

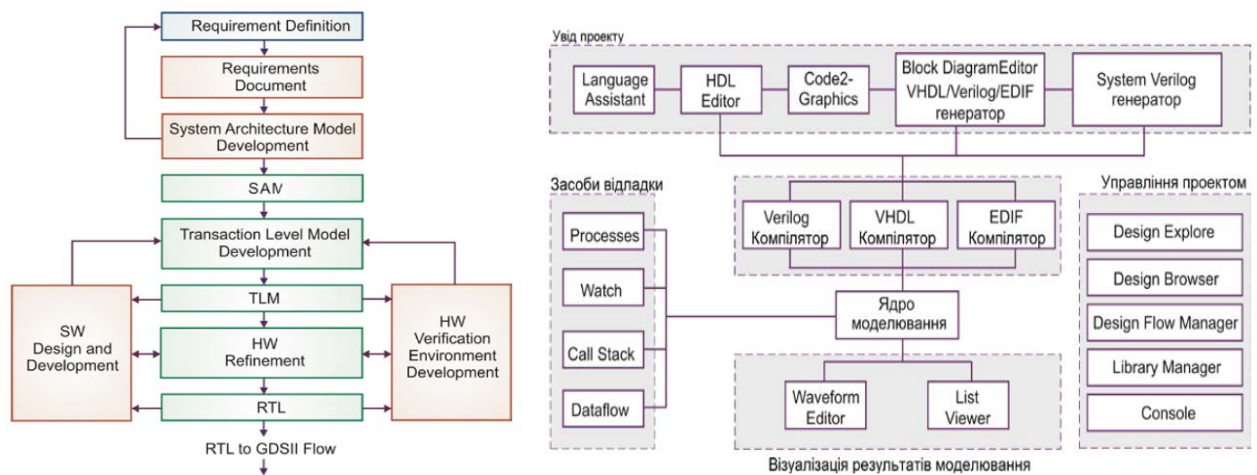
Харківський національний університет радіоелектроніки

Національний університет "Львівська політехніка"

В. Хаханов, І. Хаханова, Є. Литвинова, М. Лобур, Г. Хаханова

Схемотехнічне проектування і моделювання НВІС

Підручник



№530785-TEMPUS-1-2012-1-PL-TEMPUS-JPCR



Tempus

Підручник "Схемотехнічне проектування і моделювання НВІС" створено для допомоги вищим навчальним закладам України впровадити нову магістерську навчальну програму "Проектування мікросистем".

Підручник "Схемотехнічне проектування і моделювання НВІС" створено при підтримці Європейського Союзу за Спільним Європейським Проектом "Curricula Development for New Specialization: Master of Engineering in Microsystems Design" (MastMST), ідентифікаційний номер 530785-TEMPUS-1-2012-1-PL-TEMPUS-JPCR.

Координатор проекту проф. Збігнєв Лісік, Технічний університет м.Лодзь, Польща.

Учасники проекту:

- Національний університет "Львівська політехніка", м. Львів, Україна, координатор проф. Михайло Лобур.
- Київський Національний університет ім. Тараса Шевченка, м. Київ, Україна, координатор проф. Валерій Скришевський.
- Харківський національний університет радіоелектроніки, м. Харків, Україна, координатор проф. Володимир Хаханов.
- Донецький національний технічний університет, м. Красноармійськ, Україна, координатор проф. Володимир Святний.
- Технічний університет м.Ільменау, Німеччина, координатор проф. Іво Рангелов.
- Ліонський Національний інститут прикладних наук, Франція, координатор проф. Александра Апостолук
- Університет Павії, м.Павії, Італія, координатор проф. Паоло Ді Барба

Підручник схвалено редакційним комітетом (проф. Паоло Ді Барба (Університет Павії) - співголова, проф. Александра Апостолук (Ліонський Національний інститут прикладних наук) – співголова, члени: проф. Збігнєв Лісік (Технічний університет м.Лодзь), д-р Яцек Подгурські (Технічний університет м.Лодзь), Д-р Януш Возний (Технічний університет м.Лодзь), Д-р Валентин Іщук (Технічний університет м.Ільменау), Д-р Марія-Евеліна Могначі (Університет Павії), Д-р Роберто Галді (Університет Павії)) 6 травня 2016, м. Павія, Італія

Автори висловлюють глибоку вдячність керівництву вищеназваних університетів за всебічну підтримку Проекту.

Textbook "**VLSI Schematic Design and Simulation**" developed to help higher education institutions in Ukraine to introduce new master's educational program "Designing microsystems".

Textbook "**VLSI Schematic Design and Simulation**" was created with the support of the European Union within the Joint European Project "Curricula Development for New Specialization: Master of Engineering in Microsystems Design" (MastMST), identification number 530785-TEMPUS-1-2012-1-PL-TEMPUS-JPCR.

Project Coordinator prof. Zbigniew Lisik, Lodz University of Technology, Lodz, Poland.

Учасники проекту:

- Lviv Politechnical National University, Lviv, Ukraine ,
Coordinator prof. Mykhailo Lobur.
- Taras Shevchenko National University of Kyiv, Ukraine,
Coordinator prof. Valeriy Skryshevsky.
- Kharkiv National University of Radioelectronics, Ukraine ,
Coordinator prof. Vladimir Hahanov.
- Donetsk National Technical University, Krasnoarmiysk,
Coordinator prof. Volodymyr Sviatny.
- Ilmenau University of Technology, Germany,
Coordinator prof. Ivo Rangelow.
- Lyon Institute of Applied Sciences, France,
Coordinator prof. Alexandra Apostoluk.
- University of Pavia, Italy,
Coordinator prof. Paolo Di Barba.

The Handbook was approved by Editorial Committee (prof. Paolo Di Barba (University of Pavia) - Co-Chair, prof. Alexandra Apostoluk (Lyon Institute of Applied Sciences) – Co-Chair, members: prof. Zbigniew Lisik (Lodz University of Technology), Dr Jacek Podgorski (Lodz University of Technology), Dr Janusz Wozny (Lodz University of Technology), Dr Valentyn Ishchuk (Ilmenau University of Technology), Dr Maria Evelina Mognaschi (University of Pavia), Dr Roberto Galdi (University of Pavia) May 6, 2016, Pavia, Italy.

The authors express their deep gratitude to the aforementioned universities for full support of the project.

ВСТУП

Нанoeлектроніка, комп'ютер і кіберпростір (Інтернет) складають сьогодні три рівня ієрархії цифрової планети, що еволюціонують. На їх основі третина всіх інтелектуальних ресурсів людства в даний час працює над створенням цифрової карти або дзеркала структурної взаємно однозначної відповідності між процесами, явищами реального і віртуального світів в цілях підвищення якості життя людства шляхом створення «зеленої» кіберпланети. Це означає оцифровування не тільки всіх мобільних і стаціонарних об'єктів, а й «чистого» структурованого земного простору для точного визначення координат процесів і явищ, що відбуваються на реальній планеті. Цифрова карта останньої постійно еволюціонує від стаціонарності до динаміки процесів реального часу і вже використовується для створення в кіберпросторі хмарних сервісів точного моніторингу та оптимального (безпілотного) управління (рухомими) об'єктами реального світу у вимірах: 1D, 2D, 3D.

Прискорення розвитку знань про природу останнім часом пов'язується з технологічною сингулярністю або вибухом у розумінні законів мікро- й макросвіту на короткому проміжку часу, що неодмінно приведе до створення через два десятиліття глобального кібермозку для управління людством, а також процесами та явищами реального світу. Це стає можливим завдяки розвитку трьох компонентів: біоінженерії, штучного інтелекту кіберпростору і нанотехнологій адитивного виробництва промислових виробів. Тут мається на увазі: 1) вбудована і безпосередня інтеграція мозку людини з комп'ютером або кіберпространством шляхом усунення мовних інтерфейсів між ними; 2) створення штучного інтелекту для самонавчання і самовдосконалення небіологічних (комп'ютерних) структур, програм і процесів; 3) «нано-вирощування» комп'ютера шляхом адитивного структурування атомів. Ринкова привабливість «зелених» нанотехнологій висхідного проектування – побудова або вирощування (квантового) комп'ютера шляхом структурування атомів – полягає в безвідходності, мікромініатюрності, наднизькому енергоспоживанні, абсолютно мінімальній витратності матеріалів, а в майбутньому й вартості, надвисокій швидкодії та необхідній масштабованості, сумірній з класом сервісів, що доставляються. Сучасні технології дозволяють сьогодні не тільки сканувати атомні структури, а й послідовно будувати або вирощувати їх за допомогою технологій 3D-принтера. Однак на шляху вирішення проблеми нано-технологічного напрямку розробки квантового комп'ютера на ринку є три привабливих, але ще не вирішених завдання: 1) Відкриття технологій з високою швидкістю вирощування необхідних гетерогенних атомних структур відповідно до заданої програмної специфікації обчислювача; 2) Створення ефективного транзакційного механізму для реалізації простого моніторингу та управління квантовими станами вирощених атомних обчислювальних структур з компонентами, що адресуються; 3) Забезпечення необхідної стабільності в часі станів компонентів атомної структури, що реалізує пам'ять. Вирішення згаданих проблем у мікросвіті дозволить радикально змінити всі процеси і явища в макросвіті, який сьогодні еволюціонує у формі кіберпростору (Інтернету) планети, послідовно проходячи такі

періоди: 1) 1980-ті роки – формування парку персональних комп'ютерів; 2) 1990-ті роки – впровадження Інтернет технологій у виробничі процеси і побут людини; 3) 2000-і роки – підвищення якості життя за рахунок повсюдного використання мобільних пристроїв і хмарних сервісів; 4) 2010-ті роки – створення цифрової інфраструктури моніторингу, управління та взаємодії між собою стаціонарних і рухомих об'єктів, у тому числі повітряний, морський, наземний транспорт і роботи; 5) 2015-ті роки – створення глобальної цифрової інфраструктури кіберпростору, де всі процеси, явища ідентифікуються в часі та в тривимірному просторі, поступово перетворюючись на інтелектуальні (Internet of Things, Smart Everything).

Куди сьогодні прагне реальний кібернетичний світ? Корпоративні мережі, персональні комп'ютери, а також окремі сервіси (програмні продукти) йдуть у хмари кіберпростору, які мають яскраво виражену тенденцію до розшарування Інтернету по спеціалізованих сервісах. Якщо сьогодні 4 мільярди користувачів з'єднуються в Інтернеті за допомогою 50 мільярдів гаджетів, то через п'ять років кожен активний користувач буде мати не менше 10 пристроїв для зв'язку з кіберпростором. Стає неможливим використання персональних гаджетів і комп'ютерів без часткового або повного їх відображення та синхронізації на хмарах Інтернету. Це надає можливість вирішувати проблему віддаленого доступу до особистих даних і сервісів персонального комп'ютера при переміщенні користувачів у просторі. Економічний чинник неефективного використання придбаних додатків, розміщених в гаджетах і персональних комп'ютерах, змушує користувача відмовлятися від їх покупки на користь майже безкоштовної оренди сервісів на хмарах. Все згадане вище є істотним аргументом і незаперечним доказом неминучого переходу всього людства в кіберпростір віртуальних мереж і комп'ютерів, що розташовуються в професійно надійних хмарах сервісів. Переваги віртуального світу полягають у тому, що мікро-комірки і макро-мережі в хмарах інваріантні по відношенню до численних гаджетів кожного користувача або корпорації. Хмарні технології знімають практично всі згадані вище проблеми надійності, безпеки, сервісного обслуговування і практично не мають недоліків. У зв'язку з глобальним переходом корпорацій і користувачів в хмари надзвичайно актуальною і ринково привабливою стає проблема захисту інформації і компонентів кіберпростору від несанкціонованого доступу, деструктивних проникнень, вірусів. Необхідно створювати надійну, тестопридатну і захищену від несанкціонованих проникнень інфраструктуру кіберпростору і його компонентів (віртуальні персональні комп'ютери і корпоративні мережі) за аналогією з існуючими сьогодні рішеннями в реальному кібернетичному світі. Таким чином, кожен сервіс, що розробляється в реальному світі, повинен бути розташований у відповідній комірці хмари, яка об'єднує близькі за функціональністю і корисні людині компоненти. Сказане безпосередньо стосується й сервісів проектування цифрових систем на кристалах, які економічно і технічно доцільно зберігати в кіберпросторі для подальшого використання за призначенням. Персональний комп'ютер перетворюється в зручний інтелектуальний мікромініатюрний інтерфейс для доступу до власної цифрової комірки або до бажаного хмарного сервісу кіберпростору. Для створення пе-

рсональних інтерфейсів – гаджетів у формі цифрових систем на кристалах ринок нано-електронних технологій надає розробникам до 1 мільярда вентилів на пластині розмірністю 2x2 см при її товщині, що дорівнює 5 мікрон. При цьому сучасні технології припускають створення пакета або «сендвіча», що містить до 7 кремнієвих кристалів. Практично «бездротове» з'єднання таких пластин ґрунтується на технологічній можливості свердління порядку 10 тисяч наскрізних отворів (vias) на 1 квадратному сантиметрі. Крім того, поява тривимірних FinFETs транзисторів і 3D-технологій реалізації об'ємних цифрових систем, що базуються на них, надають нові можливості для створення більш швидкодіючих обчислювальних пристроїв за рахунок зменшення затримок. Маючи згадані технічні можливості, можна і потрібно використовувати «жадібні» до апаратури моделі та методи для створення швидкодіючих засобів паралельного вирішення практичних завдань, що сьогодні є затребуваним при створенні інтелектуальних обчислювачів для кіберпростору, хмарних структур Big Data і сервісів Інтернету; підвищенні надійності комп'ютерних пристроїв; проектуванні, моделюванні, верифікації та тестуванні цифрових систем на кристалах.

Необхідність видання підручника обумовлена підвищеним інтересом Hardware індустрії та студентської аудиторії до мов проектування енергоощадної, компактної та спеціалізованої цифрової апаратури на основі SoC і SiP. Часові витрати апаратної реалізації проекту вже практично дорівнюють періоду створення аналогічного програмного продукту. В останні роки різниця між time-to-market для Software і Hardware Design зникає завдяки технологічним досягненням у наноелектроніці силіконових кристалів і автоматизації процесів проектування виробу на всіх його стадіях (Gate level, RTL, System level). Як наслідок, комп'ютерна та програмна інженерія все більше інтегруються на полі взаємних інтересів, пов'язаних зі створенням компонентів SoC і SiP, і взаємно збагачуються своїми методологіями, утворюючи Software / Hardware Codesign або Electronic System Level (ESL) Design. Реалізація ідей – у залізо або програму – залежить від її ринкової привабливості, що формує метрику проекту: швидкодія або гнучкість, спеціалізація або універсалізація, мініатюрні або дециметрові розміри цифрової системи, високе або низьке енергоспоживання. Вибір у бік апаратури обумовлює промислове використання апаратно орієнтованих та системних мов, таких як Verilog та SystemVerilog. Однак серед студентів існує психологічний бар'єр на шляху до апаратної реалізації цікавих для ринку функціональних рішень, який автори книги, у міру можливості, хотіли б усунути.

Мета видання – надати читачеві технології проектування, синтезу й верифікації спеціалізованих і мініатюрних комп'ютерів на основі індустріально-орієнтованих мов Verilog та SystemVerilog за час, порівнянний з періодом створення програмного продукту. Для досягнення мети розглядаються дві мови опису апаратури: Verilog та SystemVerilog.

Відмінність даної книги від існуючих видань: 1) Надання новітніх технологій Design & Test для нових конструктивів SoC і SiP. 2) Порівняння та аналіз

можливостей всіх мов опису апаратури для проектування, тестової і часової верифікації цифрових систем на кристалах. 3) Компактність і структурованість викладу матеріалу з проникненням у текст англomовних термінів. 4) Різнороманіття прикладів HDL-опису, верифікації, синтезу та аналізу реальних цифрових компонентів. 5) Використання мінімальної кількості найбільш популярних зарубіжних джерел з проектування та верифікації цифрових систем, а також досвіду читання курсу Hardware Design Automation у Харківському національному університеті радіоелектроніки.

Сутність видання – на фоні вентиляльної потужності силіконового кристалу, що становить 1 млрд транзисторів, показати шляхи системного та RTL- проектування, тестової та часової верифікації, а також синтезу функціональних модулів на основі мов System Verilog і Verilog з використанням IEEE стандартів тестопридатного проектування та новітніх ESL Design технологій. У загальному випадку системний підхід означає використання мов опису апаратури: VHDL, Verilog, System C, System Verilog, UML, PSL, SVA, OVA; системного програмування: C++, Assembler, C; Web-дизайну – HTML, Flash, Java, .Net; Internet програмування: PHP, ASP, JSP, Java, .Net, C#; програмування баз даних: C#, Java, .Net. Проте, автори зупинились на двох мовах: System Verilog і Verilog, яких цілком достатньо для ефективного створення Hardware/Software компонентів SoC і SiP.

Структура видання представлена 5 частинами, 24 розділами і 204 підрозділами, в яких відображено: 1) Новітні технології та конструктиви для розміщення цифрових систем у пакетах, а також моделі системної верифікації проектів на основі механізму асерцій. 2) Мову Verilog – для опису та синтезу функціональних компонентів та інфраструктури діагностичного обслуговування цифрових систем на основі IEEE 1500 стандарту, testbench-технологій. 3) Мову System Verilog – для опису специфікації функціональності SoC та верифікації проекту на основі використання testbench та асерційної надлишковості. 4) Часову верифікацію, що розглядає технології синхронізації, моделювання й усунення змагань у сучасних цифрових системах. Всі розділи містять практично орієнтовані програми, приклади синтезу, аналізу й верифікації цифрових компонентів. Запитання та завдання, наведені у кінці кожного розділу, покривають знання та вміння, висвітлені у ньому.

Найбільш істотними і актуальними, з точки зору авторів, задачами в галузі вивчення і застосування HDL-мов є: 1) Класифікація та відмінності синтаксичних і семантичних особливостей мов опису апаратури для проектування SiP і SoC. 2) Системний опис ієрархічної інфраструктури сервісного обслуговування цифрових систем в пакетах на основі IEEE стандартів тестопридатного проектування як доповнення до основної функціональності. 3) Розробка та опис системних моделей і технологій для тестування, верифікації та виправлення помилок у HDL-кодi. 4) Ефективне використання системних бібліотек провідних компаній для верифікації та синтезу цифрових виробів і компонентів SiP: пам'ять, логіка, програмні та керувальні модулі.

ЧАСТИНА І. ПРОЕКТУВАННЯ

1. ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ ТА ВЕРИФІКАЦІЇ ЦИФРОВИХ СИСТЕМ

1.1. Апаратні засоби для розробки цифрових систем

Основна мета Hardware / Software Design and Test – надати широкому колу користувачів технології створення мініатюрних, спеціалізованих, економічних, мобільних і швидкодіючих цифрових виробів, затребуваних у світі. Впровадження або просування нових засобів на ринку електроніки починається з апробації будь-якого методу або алгоритму в системному виконанні коду HDL-мовами. Після достатньо повної верифікації проект може бути представлений на ринку у вигляді IP-core, що гарантує і авторські права розробника.

Наступним етапом може бути імплементація IP-core у кристал програмованої логіки, що дозволяє суттєво підвищити швидкодію, залишаючи можливість перепрограмування окремих функцій у процесі доопрацювання, виявлення та виправлення помилок. Винятковість технологічного рішення і ринкова привабливість є критеріями для його імплементації в кристал ASIC. У цьому випадку швидкодія реалізованої системи стає максимальною, але тут вже неможливо виправляти помилки, тому проект повинен бути ідеально коректним. Таким чином, просування нової технології на ринку пов'язане з еволюцією проекту від Software до Hardware (Hardwaring), який стає менш гнучким для модифікації, але більш продуктивним для вирішення проблеми.

Що стосується кристалів ПЛІС, то вони надають розробникам нові можливості для проектування швидкодіючих спеціалізованих убудованих цифрових систем. Доказом перспективності програмованої логіки служить поява нових поколінь ПЛІС з убудованими процесорами, що мають більш досконалу архітектуру, і зростання обсягу їх випуску провідними закордонними виробниками. Аналіз тенденції розвитку архітектур програмованої логіки дає підстави стверджувати, що найближчим часом основу елементної бази цифрових систем (що реконфігуруються) становитимуть CPLD, FPGA, HardCore, які наздоганяють за продуктивністю і вартістю ASIC, але не потребують витратної тестової і верифікаційної методології для забезпечення масового виробництва мікросхем.

На ринку електронних технологій існує причинно-наслідковий зв'язок між трьома основними компонентами, що впливають на комп'ютерну продукцію. Це перш за все – технологія виготовлення силіконових пластин для кристалів, типи пропонованих на ринку чипів (мікросхем) або кристалів, мови опису цифрової і аналогової апаратури і програмних продуктів, EDA-засоби для проектування систем і мереж на кристалах.

Враховуючи, що кожен з трьох основних типів кристалів (PLD, ASIC, CPU) має недоліки, виробники намагаються зменшити їх вплив, об'єднавши переваги. Так з'являються на ринку цікаві комбінації: (PLD + ASIC) – Lucent Technologies (OR3TP12), Quick Logic (RAM, PCI, DSP), Cyprus Semiconductor (PCI), Altera (Stratix), Xilinx (Virtex ® II-VI, Spartan ® 4-6). Далі інтегруються такі компоненти як (PLD + CPU) – Atmel (FPSLIC), Triscend (E5, A7), Xilinx (Virtex II Pro), Altera (Excalibur), (ASIC + CPU) – мікроконтролери, (PLD + ASIC + CPU), які інтегрують швидкодію ASIC, перепрограмованість PLD, гнучкість і керованість CPU. Приклад, який ілюструє останні досягнення функціональних можливостей PLD, наведено в табл. 1.1, що представляє широку палітру убудованих функціональностей або інструментів сучасним розробникам [1].

Таблиця 1.1. Характеристики кристалів сімейства Spartan®-6

Кристал FPGA XC	Комірки логіки	Логічні блоки		DSP 48A1	Блоки RAM		Розмір/Крок выводів, мм
		Slices	RAM, Kb		Kb	Max	
6SLX4	3 366	526	32	4	8	144	20x20/0,5
6SLX25	23 770	3 714	228	38	52	936	13x13/0,8
6SLX150	147 456	23 040	1 358	182	268	4 824	23x23/1,0
6SLX25T	23 770	3 714	228	38	52	936	23x23/1,0
6SLX150T	147 456	23 040	1 358	182	268	4 824	23x23/1,0

Сімейство кристалів Spartan®-6 (дана апаратура більш доступна за вартістю для навчального процесу порівняно зі Spartan ® -7) оптимізовано для їх застосування у виробках масового виробництва і забезпечує широкі можливості при відносно низькій вартості мікросхем. До нього входять два підсімейства, що включають 11 кристалів щільністю від 3 400 до 148 000 логічних комірок зі зменшеним у два рази енергоспоживанням, високошвидкісними і більш гнучкими трасувальними ресурсами. Підсімейство LX орієнтовано для виконання логічних функцій, LXT – для реалізації високошвидкісних послідовних інтерфейсів. Кристали сімейства Spartan®-6 виконані за відпрацьованою 45-нм КМОП технологією, що дозволяє встановити оптимальний баланс між вартістю, потужністю і продуктивністю. У них використовуються нові, ефективні логічні комірки з шестивходовими таблицями перетворення (Look-up Tables – LUTs) і двома тригерами. Логічні комірки об'єднуються парами у секції (slices). Кожен конфігурований логічний блок CLB містить 4 таблиці LUT і 8 тригерів (2 секції). Є великий вибір убудованих блоків системного рівня (апаратних ядер). Block RAM має пам'ять ємністю 18 Кбіт, яка може бути конфігурована як два блоки по 9 Кбіт.

Друге покоління блоку цифрової обробки сигналів DSP48A1 складається з 18x18 помножувача, 48-бітового акумулятора, передсуматора і регістрів для конвеєризації, які можуть функціонувати з частотою 250 МГц. SDRAM контролери пам'яті забезпечують смугу пропускання до 12,8 Гбіт/с. Є поліпшений блок керування та синтезу сигналів синхронізації (Clock Management Tile, CMT). Кристали забезпечують підтримку SelectIO™ технології і містять високошвидкісні GTP приймачі-передавачі з оптимізацією енергоспо-

живання, а також убудований прикінцевий пристрій для PCI Express™. Також є удосконалені режими керування живленням, автоматичного визначення конфігурації та посилений захист проекту від несанкціонованого доступу з використанням алгоритму шифрування конфігураційної послідовності AES і технології Device DNA.

Сімейство кристалів Virtex® -6 оптимізовано для використання у високотехнологічній апаратурі, характеризується поліпшеними властивостями і забезпечує найбільш широкі можливості на ринку FPGA. Застосування інноваційної архітектури ASMBL™ (Advanced Silicon Modular Block), що базується на колонках модульних логічних блоків, дозволило створити різні платформи (підсімейства), націлені на різні сфери застосування (домени). Кожен домен включає набір додатків, які розділяють загальні ресурси, такі як вискоефективна обробка сигналів, убудований процесор, високошвидкісні міжз'єднання (рис. 1.1).

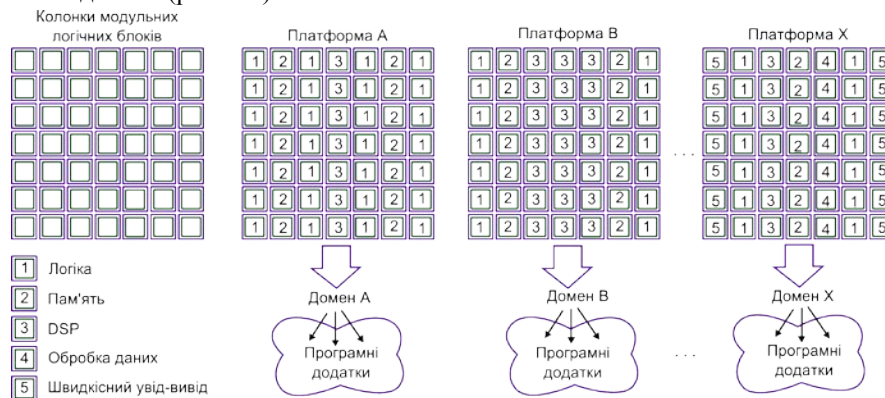


Рис. 1.1. Різноманіття платформ для різних областей застосування

Користувач може вибрати платформу з оптимальним поєднанням функціональних можливостей і параметрів для створення спеціалізованої системи на кристалі. ASMBL™ архітектура дозволяє реалізувати технологію монтажу методом перевернутого кристала (flip-chip) і усунути обмеження, пов'язані з особливостями традиційного проектування кристалів, наприклад, залежність між кількістю вихідних контактів і розмірами матриці. Архітектура також дає можливість усунути більш жорсткі обмеження на розміщення кіл живлення і заземлення на кристалі. Як наслідок – значно зменшується час розробки проекту, його вартість і збільшується надійність FPGA.

Два підсімейства LXT і SXT містять 9 кристалів різної логічної ємності. LXT оптимізовано для виконання швидкодіючих логічних операцій і реалізації високошвидкісних послідовних інтерфейсів. Кристали SXT, перш за все, орієнтовані на виконання швидкодіючих операцій цифрової обробки сигналів. Висока продуктивність логічних комірок забезпечується використанням шестивходових таблиць LUT, доповнених здвоєним тригером. FPGA сімейства Virtex® -6 містять широкий спектр убудованих блоків (апаратних IP-

ядер), що дозволяють створювати закінчені рішення системного рівня. Block RAM – блок пам'яті ємністю 36 Кбіт можна конфігурувати як два блоки по 18 Кбіт. Він також може функціонувати як FIFO і має убудовані ресурси визначення і виправлення помилок.

Третє покоління блоку цифрової обробки сигналів DSP48E1 складається з 25х18 помножувача, 48-бітового акумулятора, передсуматора і регістрів для конвеєризації. Блок може функціонувати на частоті до 600 МГц. Кристал підтримує SelectIO™ технології з цифровим контролем імпедансу і має також блок керування і синтезу сигналів синхронізації MMCMT. System Monitor орієнтований на виконання функцій знімання характеристик за напругою живлення, температури усередині кристала і на зовнішніх мікросхемах. У кристалі також є GTX – 6,5 Гбіт/с приймачі-передавачі, модуль PCI Express з підтримкою 8-lane Gen1 (2,5 Гбіт/с) і 4-lane Gen2 (5,0 Гбіт/с), TEMAC – 10/100/1000 Мбіт/с Ethernet контролер. Напруга живлення ядра 1В (0,9 В – економічний режим).

Зазначені засоби дозволяють досягти найбільш високого рівня продуктивності та функціональності у цифрових системах, реалізованих на основі FPGA. Виконані за 40-нм КМОП технологією, кристали FPGA сімейства Virtex®-6 є реальною альтернативою технології ASIC. Virtex®-6 FPGA являє собою найкраще рішення для об'єднання можливостей високопродуктивної логіки, цифрової обробки сигналів у цифровій системі на кристалі. Характеристики окремих кристалів сімейства Virtex®-6 наведено в табл. 1.2.

Сімейство FPGA Stratix® IV [2] компанії Altera виконано за 40-нм технологією та за своїми можливостями перевершує існуючі аналоги. Кристали характеризуються щільністю логічних елементів, найбільшою кількістю приймачів, найменшим енергоспоживанням (на 50% нижче, ніж в аналогічних пристроях FPGA, завдяки використанню технології програмованого енергоспоживання – Programmable Power Technology). FPGA Stratix® IV призначені для реалізації бездротових і DSP додатків, що вимагають високої продуктивності. До складу сімейства Stratix® IV входять 3 підсімейства, оптимізовані для виконання різних функцій: Stratix IV GX - орієнтовані на прийом і передачу інформації, містять до 531 200 логічних елементів, до 48 повнодуплексних приймачів, що функціонують на основі технології синхронного відновлення даних (clock data recovery, CDR) і забезпечують швидкість передачі до 8,5 Гбіт / с; Stratix IV E (Enhanced) – інтерфейси пам'яті, містять до 681 100 логічних елементів, до 27 376 Кбіт убудованої пам'яті RAM і до 1 024 убудованих помножувачів 18 × 18 біт; Stratix IV GT – оптимізовані для швидкісного прийому-передачі інформації, що містять до 531 200 логічних елементів, до 27 376 Кбіт пам'яті RAM, до 1 228 убудованих помножувачів 18 × 18 біт (табл.1.3). Послідовні приймачі-передавачі інформації підтримують швидкість передавання даних від 2,488 до 10,3125 Гбіт/с. Пристрої Stratix IV GT можна розділити на дві групи залежно від швидкості передачі даних на вході/виході - 100/40 Гбіт/с, які містять відповідно 48 (36) повно-

дуплексних CDR приймачів-передавачів, 24 (12) з яких підтримують швидкість 10,3125 Гбіт/с. Габарити кристалів знаходяться в межах від 35 до 43 мм.

Таблиця 1.2. Характеристики кристалів сімейства Virtex®-6

Кристал FPGA XC	Комірки логіки	Логічні блоки		DSP 48A1	Блоки RAM		Розмір, мм/ Виводів
		Slices	RAM Kb		18 Kb	Max Kb	
6VLX75T	74 496	11 640	1 045	288	156	5 616	23/240
6VLX240T	241 152	37 680	3 650	768	416	14 976	29/400
6VLX550T	549 888	85 920	6 200	864	720	25 920	42,5/840
6VSX315T	314 880	49 200	5 090	1 344	704	25 344	42,5/720
6VSX475T	476 160	74 400	7 640	2 016	1 064	38 304	42,5/840

Таблиця 1.3. Характеристики кристалів сімейства Stratix® IV

Кристал EP	Логічні модулі	Логічні елементи	Пам'ять, Кбіт	Убудовані помножувачі	Входів/ виходів
4SGX70 (F1152)	29 040	72 600	7 370	364	480
4SGX290 (F1517)	116 480	291 200	17 248	832	736
4SGX530 (F1932)	212 480	531 200	27 376	1 024	904
4SE110 (F780)	42 240	105 600	9 564	512	480
4SE360 (F1152)	141 440	353 600	22 564	1 040	736
4SE680 (F1760)	272 440	681 100	31 491	1 360	1,104

1.2. Технологічні конструктиви цифрових систем

Представлено основні сучасні конструктиви, використовувані для створення цифрових систем для ринку електронних технологій у порядку зростання їхньої складності. Показано проблеми, що підлягають вирішенню в частині сервісного діагностичного обслуговування функціональних блоків.

1.2.1. Система на кристалі - System on Chip (SoC) - спеціалізований цифровий виріб, конструктивно виконаний на силіконовому кристалі (рис. 1.2), що характеризується низьким енергоспоживанням, малими розмірами, високою швидкодією і має такі компоненти: процесор (CPU), пам'ять (RegM, ROM, SRAM, DRAM), логіку (UDL), інтерфейс введення-виведення (I/O, PCI, UART, USB, CAN), перетворювачі інформації (ADC, DSP, MPEG), шини обміну даними та інфраструктуру (I-IP) сервісного обслуговування на основі стандарту IEEE 1500 і технології JTAG.

Функціональні блоки SoC орієнтовані на ефективне вирішення актуальної проблеми, на відміну від мікроконтролера, що являє собою універсальний обчислювач практично будь-якої задачі. Переваги SoC: прозорість щодо убудованих деструктивних компонентів; висока продуктивність при низькому енергоспоживанні; мініатюрні розміри; ефективне використання різних типів пам'яті; висока надійність; низька вартість готового виробу і високий коефіцієнт використання функціональностей. Плата за описані переваги: істотний час тестування та верифікації проекту (70% від періоду проектуван-

ня); висока вартість розробки та макетування; складність налагодження і виробництва; інтеграція IP-cores (інтелектуальної власності) із різних джерел.

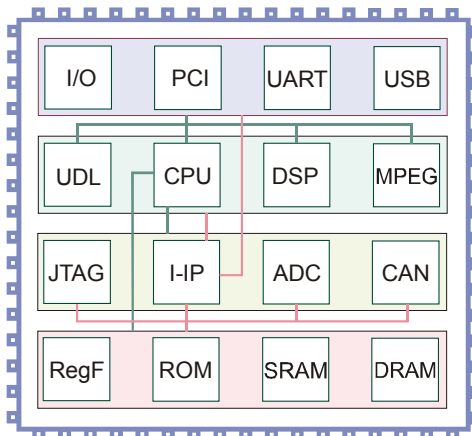


Рис. 1.2. Сучасна структура SoC

IP-Core – складний вже протестований, верифікований та оптимізований функціональний модуль, який може бути багато разів використаний як компонент при проектуванні більш складних цифрових виробів з метою скорочення часу їх розробки [3]. Використання IP-Cores дає: 1) істотне зменшення часу до моменту появи цифрового виробу на ринку; 2) спрощення процесу проектування шляхом створення інтерфейсу для зв'язку системи з IP-Core; 3) мінімізацію ризику проектування завдяки використанню вже налагоджених модулів; 4) зменшення часу верифікації всієї системи.

Області застосування SoC: конфігуровані спеціалізовані відкриті репрограмовні виробу; індивідуальні спеціалізовані мініатюрні сервери, орієнтовані на конфіденційність обчислювальних процесів і зберігання інформації під захистом апаратного ключа; інтелектуальна RAM; архітектури для стільникових телефонів; мультимедійні, бездротові та телекомунікаційні функціональності, нейронні, аналогові та квантові обчислювачі. Особливий інтерес ринку проявляє до появи індивідуальних серверів на кристалі, які орієнтовані не тільки на конфіденційне зберігання інформації, але й на захищеність обчислювальних процесів від несанкціонованого доступу. Підключивши сервер розміром трохи більше Flash Memory до порту USB персонального комп'ютера, користувач виконує характерні для його професійної діяльності обчислення, зберігаючи їх потім на сервері. Таким чином, мікросервер на кристалі можна розглядати як захищену карту Flash Memory з розширеними функціями, орієнтованими на виконання користувачем професійних обов'язків. Актуальність такої концепції підтверджується також низьким коефіцієнтом використання універсального персонального комп'ютера, що дорівнює 5-10%.

Існуючі проблеми: 1) Зменшення вартості виготовлення SoC кристалів шляхом збільшення виходу придатної продукції. 2) Скорочення періоду появи готового виробу на ринку (time-to-market) і зменшення життєвого циклу готового виробу. Якщо кожні 18 місяців змінюються технології, то термін ефективного функціонування виробу (3-4 роки) потрібно зменшувати, наближаючи його до 2 років. У цьому випадку компанії, щонайменше, будуть на хвилі використання високотехнологічних продуктів, що означає і привабливість власної продукції для світового ринку. 3) Розглядаючи традиційні рішення для досягнення ефективного виходу придатної продукції на часовій осі (15 місяців), можна зробити такий висновок: продукт, який розробляється більше 15 місяців (time-to-market), на ринок ніколи не потрапить. Скорочення даного періоду (зсув кривої вліво по осі абсцис), так само як і зменшення кількості помилок проекту (підняття кривої вгору по осі ординат), гарантує високу ринкову привабливість, а отже – додаткові доходи для компанії. Дбати про якість і надійність виробу варто, починаючи з системної стадії проектування шляхом створення нових ефективних методів і засобів тестування й верифікації, що оптимізують процеси, які відносяться до етапу Design for Manufacturability. Yield Learning Curve показує резерви для збільшення виробництва цифрових виробів (production ramp up), особливо шляхом впровадження технологій діагностування та налагодження інтегрованих у I-IP модулів. Надзвичайно важливою тут видається імплементація стандартів тестопрадності в проект для убудованого тестування, діагностування та ремонту, які суттєво підвищують вихід придатної продукції, навіть за наявності дефектів на кристалі [4].

1.2.2. Мережа на кристалі – Network on Chip (NoC) – спеціалізований мультитядерний цифровий виріб конвеєрного типу, що має всі властивості системи на кристалі, доповнений інтелектуальною структурою комунікацій, яка підтримує всі архітектури та рівні протоколу паралельного обміну даними офісної мережі.

Як ядро виступає IP-core (процесор або блок пам'яті), який організується у масштабовані сегменти. Такої ієрархії ставиться у відповідність дворівнева система комунікацій та конвеєризації. Підтримка всіх рівнів протоколу обміну даними забезпечує високу швидкість і пропускну здатність інформаційних каналів. Наявність інтелектуальних мультитядерної архітектури і комутаторів забезпечує високий рівень розпаралелювання обчислювальних процесів і приймання/передавання даних. Найбільш поширені архітектури представлені на рис. 1.3: а – мережа або матриця, як правило, однакових процесорів, з'єднана комутаторами, число яких дорівнює кількості ядер; b – блочно-функціональна сегментація дворівневої ієрархії мережі; c – багатокутна зірка, орієнтована на ефективний обмін даними між процесорними парами; d – нерегулярна структура дворівневої ієрархії, орієнтована на обслуговування процесорів різних пріоритетів і функціональностей.

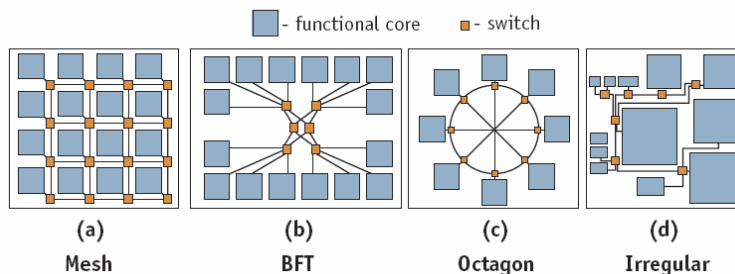


Рис. 1.3. Типові структури NoC

Типова система на кристалі має сьогодні 15-20 (IP-cores) функціональностей. Це поріг її складності та практичного застосування. Далі йде зона ефективного використання мережі. На відміну від SoC кристал NoC має багаторівневу архітектуру.

Як і глобальна (локальна) мережа, NoC стандарт має сім рівнів протоколу обміну інформацією. NoC структура ефективна не тільки з позиції обчислювальних процесів і обміну даними, але й з точки зору енергозбереження та надійності цифрової системи в цілому. Три рівні: транзакцій, транспортний і фізичний використовуються для відокремлення IP від даних, що передаються шинами. Перший необхідний для передавання пакетованих сигналів. Другий організовує маршрути проходження пакету через систему комутаторів. Третій орієнтований на обмін двійковою інформацією по проводах. Наявність системи пріоритетів, корекція помилок у мережі на кристалі, висока захищеність інтелектуальної власності (IP-core), орієнтація на створення проектів масштабованих систем великої розмірності, ефективність обміну даними, низьке енергоспоживання, висока швидкість виконання операцій, невисока вартість проектування та інші сервіси, характерні для офісної мережі, роблять NoC дуже привабливою технологією для світового ринку, де рівень продажів мережних кристалів у найближчі п'ять років досягне 1 млрд доларів.

Мережа на кристалі видається цікавою парадигмою для проектувальників на шляху можливої інтеграції великої кількості обчислювачів (IP-cores) і блоків пам'яті в одному чипі. Але шлях адаптації SoC технологій до NoC структури та практична імплементація мережі в кристал ставить багато важливих і ще не вирішених проблем перед розробниками, що відносяться до методологій проектування, тестування, а також створення нових засобів CAD [5]. Як результат істотного підвищення рівня інтеграції функціональностей виникає проблема підвищення ефективності міжмодульних зв'язків шляхом розробки інфраструктури комунікацій. В даний час існують SoC проекти, що містять мультипроцесори, які застосовуються у TV-тюнерах, безпроводових базових станціях, цифровому телебаченні високої роздільної здатності, мобільних телефонах, пристроях обробки зображень (set-top boxes, wireless base stations, HDTV, mobile hand-sets and image processing). Згадані пристрої вимагають

від розробників ефективних рішень у галузі проектування регулярної та швидкодіючої комунікаційної архітектури для організації паралельних обчислень на мультіядерних системах у кристалах.

Різні архітектури комунікацій формують типові специфічні шаблони мережних рішень на кристалі. Найцікавіші з них представлені на рис. 1.3. Слід зауважити, що складний кристал SoC може розглядатися як мікромережа множини функціональних блоків, що дозволяє дослідникам запозичити мережні моделі й технології паралельних обчислень для їх подальшого застосування в архітектурі NoC. Бібліотека «нових» компонентів NoC, що містить комутатори, маршрутизатори, з'єднання та інтерфейси, забезпечує розробнику додаткову гнучкість у проектуванні архітектури NoC на основі процесорних модулів та блоків пам'яті (processor/storage cores). Звичайно, для цього повинні існувати відповідні й досить вивірені засоби і маршрути синтезу та аналізу цифрових мережних систем, що підтримують всі рівні обміну даними, від фізичного до транзакційного.

Істотними характеристиками проекту є енергетичні витрати архітектури міжз'єднань, їхня пропускна здатність, час затримки, а також апаратна надлишковість для їх реалізації. Всі основні властивості NoC спрямовані на ефективне вирішення проблеми високопродуктивних з'єднань між процесорними компонентами і модулями пам'яті на основі інтелектуальних комутаторів, які створюють комунікаційні зв'язки з високим рівнем абстракції. Однак задача обміну даними між компонентами мережі (процесори і пам'ять) стає все більш складною в міру зростання розмірності системи і узагальненої затримки міжз'єднань. Одним з можливих рішень є представлення системи міжз'єднань мультимодульною конвеєрною платформою, де кожен комунікаційний модуль відповідає за часовими параметрами функціонуванню всієї системи. При цьому NoC архітектура підтримує конвеєрний тип, як внутрішньо-сегментних перемикачів з'єднань модулів, так і міжблокових контактів структури, що визначається конвеєризацією встановлення зв'язків, перемикачів, інваріантних до затримок компонентів мережі.

Тестування NoC. Оскільки мережа виходить із системи на кристалі, то природним є адаптація існуючих SoC технологій і механізмів тестування (ТАМ) до аналогічних проблем з обов'язковим використанням методів і засобів тестопридатного проектування (DFT). Стратегія тестування NoC систем орієнтована на вирішення трьох основних проблем: 1) Тестування функціональних модулів і блоків пам'яті та відповідної їм мережі інтерфейсів. Механізм ТАМ використовується для транспортування тестових наборів з відповідної пам'яті або генератора до модуля або блоку пам'яті, що тестується. Завдяки регулярності мережі існує можливість одночасного і паралельного завантаження тестових ресурсів для кожного модуля. Таким чином, час тестування може бути зменшений завдяки використанню просторових характеристик мережі, що дають можливість розпаралелювати процеси подання тестів на функціональні блоки та їх тестування. Спостережуваність міжз'єднань є по-

рівняно низькою, оскільки вони глибоко інтегровані у кристал. Обмеження на кількість зовнішніх виводів кристалу впливають на кількість вхідних-вихідних контактів, які можна застосовувати в процесі тестування. NoC інфраструктура повинна бути застосована для тестування власних компонентів за рекурсивною процедурою – вже перевірені модулі необхідно використовувати для транспортування тестових наборів до ще не перевірених блоків. Стратегія тестування мінімізує застосування додаткових механізмів для транспортування даних до компонентів мережі, що дозволяє зменшити час тестування завдяки використанню паралельних шляхів для групового передавання даних. Роздільне тестування функціональних блоків та інфраструктури між'єднань не є достатньою умовою отримання необхідної якості тесту. 2) Перевірка інфраструктури між'єднань (міжблокових і внутрішньоблокових). Взаємодія між функціональними модулями та структурою комунікацій перевіряється повними функціональними тестами. 3) Тестування інтегрованої системи в цілому повинно охоплювати перевірку вхідних-вихідних функцій кожного процесорного елемента, а також всі функції маршрутів проходження даних.

Для забезпечення якості та надійності NoC, за аналогією з SoC I-IP, необхідно створювати інфраструктуру сервісного обслуговування (I-IP) мережі на основі використання IEEE стандартів 11.49 і 1500, яка повинна містити такі компоненти: 1) Моніторинг станів внутрішніх і вихідних ліній у процесі функціонування, верифікації й тестування блоків на основі використання стандарту граничного сканування IEEE 1500 [3]. 2) Тестування функціональних модулів, орієнтоване на перевірку дефектів або справної поведінки. 3) Діагностування відмов шляхом аналізу інформації, отриманої на стадії тестування та використання спеціальних методів убудованого пошуку несправностей на основі стандарту IEEE 1500 [3]. 4) Ремонт функціональних модулів і пам'яті цифрових систем на кристалах після визначення місця й виду дефекту при виконанні діагностування. 5) Забезпечення надійності функціонування цифрового виробу за допомогою диверсифікації та дублювання функціональних блоків.

1.2.3. Система у пакеті System in Package (SiP) – це набір кристалів, розташованих на загальній підкладці, що являє собою функціональність, реалізовану в одному корпусі [4]. SiP містить: пасивні компоненти, такі як дискретні елементи; пасивні інтегральні схеми; пасивні елементи, убудовані в підкладку, а також компоненти спеціального призначення у вигляді фільтрів на поверхнево-акустичних хвилях, електромагнітні екрани (EMI shields); паковані кристали, зовнішні з'єднувачі та механічні елементи.

Традиційне застосування систем у пакеті: керування живленням, сенсорні пристрої, RF і wireless додатки для мобільної телефонії, пристрої для високопродуктивних обчислень, пам'ять, кишенькові комп'ютери – PDA, глобальні системи навігації та визначення місця розташування – GPS, засоби WiFi і Bluetooth™ [3].

Функціональні особливості силіконових кристалів для виготовлення цифрових виробів можуть бути класифіковані за такими ознаками [6, 7, 10]: 1) Технологія: цифрові, аналогові, RF, оптоелектронні, MEMS, пасивні. 2) Частота: від МГц діапазону для цифрових пристроїв і RF до ГГц для мікрохвильових компонентів. 3) Архітектура: аналогові пристрої малої та великої потужності; цифрові структури реального часу, керовані подіями та даними. 4) Споживання енергії: портативні пристрої з малим споживанням енергії. 5) Інші фактори: форм-фактор, вага, виділення тепла. Традиційні технології пакування кристалів характеризуються наявністю обмежень за такими характеристиками [7]: 1) Продуктивність системи на кристалі, що визначається затримками передачі сигналу між кристалами, розповсюдженням і часом наростання сигналу. 2) Мініатюризація, що визначається площею установлювального місця корпусу системи на кристалі, зазором між провідниками та кількістю шарів топології. Актуальними є задачі зменшення площі системи у пакеті кристалів і збільшення її продуктивності.

SiP є розвитком більш ранніх технологій пакування MCM (Multichip Module), MCP – Multi-Chip-Package і характеризується наявністю одного або більше кристалів, комбінуванням технологій провідного монтажу та монтажу методом перевернутого кристалу – виводами до підкладки, розміщенням пам'яті та логіки на одній підкладці разом з пасивними компонентами, фільтрами та антенами. Різноманіття компонентів MCM модуля показано на рис. 1.4,а.

В даний час ринок електронних технологій пропонує п'ять концепцій тривимірного пакування кристалів [8, 9]:

1. «Multichip Package» [7, 10]. Кристали розміщуються горизонтально (рис. 1.4, б), можуть накладатися один на одного у вигляді етажерки або утворювати комбіновану 3D структуру в цілях забезпечення мінімального значення форм-фактора. Конструкція «Multichip Package» характеризується високою продуктивністю, малими розмірами і вагою, а також мінімальною затримкою розповсюдження сигналу.

2. «System-in-Package» [7, 9, 10] базується на метриці різновидів SiP: 1) Використання компонентів, що допускають розміщення у горизонтальній площині (side-by-side) (рис. 1.4, в); технологія провідного монтажу і термокомпресійного мікрозварювання Wire Bond; монтаж кристалів на гнучкій стрічці, виготовлений з полімерного матеріалу (поліімід) Tape Automated Bonding (TAB) та/або компонентів, що монтуються методом перевернутого кристалу flip-chip. 2) Використання компонентів, що являють собою складальні одиниці типу «складений кристал» (Stacked Die) у вигляді BGA (Ball Grid Array) модулів з кульковими виводами або таких, що монтуються за технологією Wire Bond, а також компонентів Flip-chip (рис. 1.4, г). 3) Використання убудованих у підкладку компонентів. У порівнянні з розміщенням у горизонтальній площині складальні одиниці типу Stacked Die дозволяють збільшити швидкодію, зменшити габаритні розміри, споживану потужність і поліпшити температурний режим у робочій зоні.

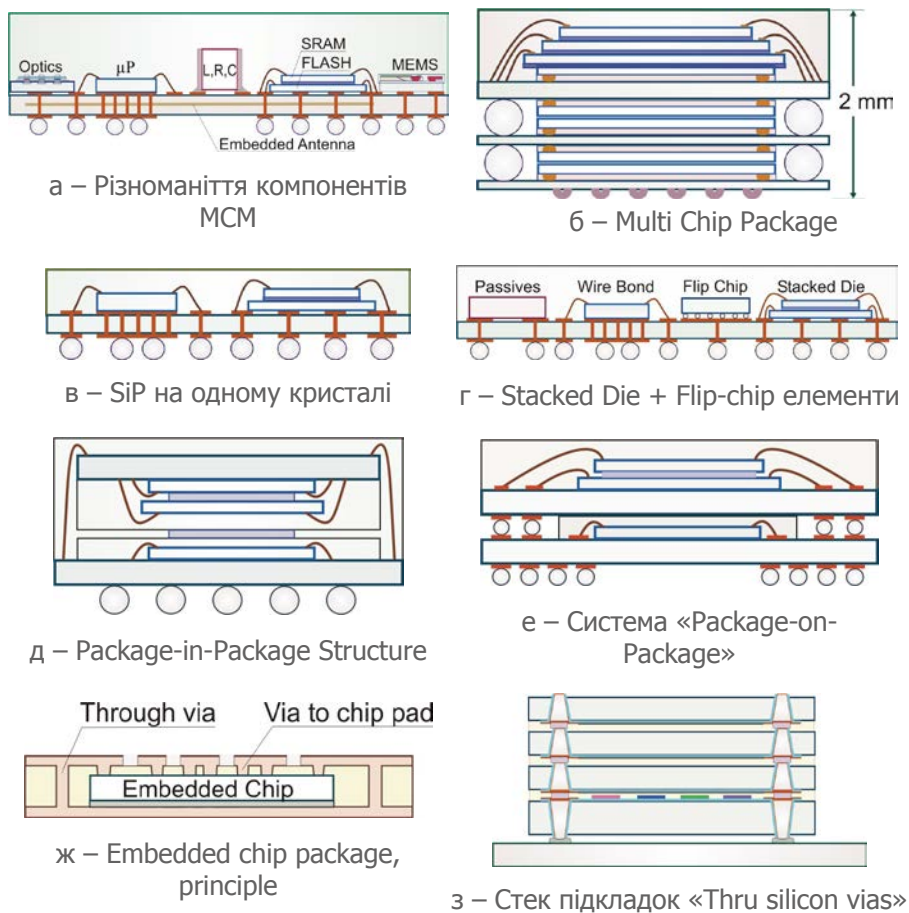


Рис. 1.4. Конструктиви цифрових систем у пакетах

Окремі кристали можуть бути досить просто переведені з штатного режиму в режим автономного тестування. Застосування технології монтажу компонентів методом перевернутого кристалу дає можливість покращити значення форм-фактора, зменшити паразитні ємності й індуктивності між елементами топології.

3. «Package-in-Package» (PiP) [9, 10]. Пакет складається з набору або стека повністю перевірених модулів (Internal Stacking Module, ISM), розміщених у базовому складальному пакеті (Base Assembly Package, BAP) в цілях формування єдиного конструктивного виробу (рис. 1.4, д).

4. «Package-on-Package» (PoP) [8, 9, 10]. Являє собою структуру, в якій повністю перевірені пакети встановлено вертикально, один на одного. При цьому ступінь інтеграції може бути різною. Дана концепція отримала найбільше застосування для створення цифрових або аналого-цифрових логічних при-

строїв з високою щільністю упаковки, що розміщуються в нижньому базовому пакеті, з одним або кількома запам'ятовуючими пристроями великої місткості, розміщеними у верхньому пакеті [8]. Лідером розробки і впровадження технологій тривимірного пакування кристалів (3D packaging technologies) є компанія Amkor Technology. Приклад конструктивного рішення PoP, при якому верхній пакет – модуль пам'яті – розміщується у корпусі FBGA (Fine Pitch BGA) або Stacked CSP (Chip Scale Package), а нижній – логічний модуль – у корпусі PSvfBGA (Package Stackable Very Thin Fine Pitch BGA) (рис. 1.4, е).

5. «Embedded chip technologies», «Substrate Technology» [7, 10]. Активні або пасивні кристали вбудовуються в підкладку (рис. 1.4, ж), де міжз'єднання реалізуються за допомогою мікроотворів. Технологія «Wafer Level» [10] дозволяє встановлювати кристал на підкладці, підкладку на підкладці (рис. 1.4, з) і реалізовувати технологію міжз'єднань Thru Silicon Vias (TSV) [8, 10]. Wafer Level технологія характеризується високим ступенем інтеграції і, разом з цим, має ряд недоліків [10]: висока вартість проектування та виробництва цифрової системи в пакеті кристалів; використання компонентів від різних виробників, що виготовляються за різними технологіями; складність процесу відновлення працездатності системи; відносно низький відсоток виходу придатних виробів у порівнянні з іншими технологіями. Стандарт із застосування багатокристалів IPC-MC-790 (Guidelines for Multichip Module Technology Utilization) визначає три типи технологій виготовлення підкладки: багатшарова (Laminated); керамічна (Cofired) з використанням технології тонких плівок (Thick Film) і підкладка, яка формується технологіями напилювання (Deposited).

Переваги тривимірного пакування кристалів на основі технологій активного використання підкладки [10]: 1) Мініатюризація – збільшення щільності пакування і топології, які визначаються таким чином:

$$P_{\text{eff}} = \text{Total Active Silicon Area} / \text{Total Active SiP Area} (\%),$$

де P_{eff} – щільність пакування; Total Active Silicon Area – активна площа кристалу; Total Active SiP Area – активна площа цифрової системи у пакеті кристалів SiP; $\text{Wiring Density} = \text{Wiring Length} / \text{Silicon Area} (\text{cm} / \text{cm}^2)$, де Wiring Density – щільність топології; Wiring Length – сумарна довжина зв'язків; Silicon Area – площа кристалу. 2) Збільшення продуктивності. Для її оцінки на системному рівні використовується параметр «час циклу»:

$$\text{CycleTime} = \text{Time for Semiconductor} / \text{Time for Package},$$

де CycleTime – час циклу; Time for Semiconductor – час циклу для напівпровідника; Time for Package – час циклу для пакету кристалів. 3) Зменшення вартості пакету за рахунок підвищення відсотка виходу придатних виробів і усунення етапу пакування системи в корпус, а отже й виключення відповід-

них операцій з виробничого процесу. 4) Поліпшення надійності системи завдяки зменшенню сумарної довжини зв'язків та кількості шарів пакету за рахунок розміщення безкорпусних кристалів безпосередньо на багатошаровій підкладці.

Проблеми використання технологій підкладки: 1) висока вартість виробництва; 2) складність процесу прийняття рішень (Decision Making Process); 3) неможливість використання існуючих моделей; 4) негарантована якість безкорпусних кристалів; 5) низький відсоток виходу придатних виробів після операцій складання пакету кристалів; 6) складний процес відновлення працездатності; 7) обмежений набір засобів автоматизації проектування (CAD Tools); 8) складність забезпечення оптимального теплового режиму.

Статистика сучасного ринку щодо використовуваних продуктів на основі SiP [10]: 33% – замовні спеціалізовані вироби; 20% – засоби телекомунікацій; 20% – компактні енергоощадні обчислювачі; 10% – військового або авіакосмічного призначення.

Технології підкладки: 58% – керамічна – Cofired; 39% – багатошарова – Laminated; 3% – напилення – Deposited. Технології монтажу: Wirebond, TAB, FlipChip. Технології забезпечення тестопридатності: 50% – Boundary-Scan; 54% – внутрішньосхемне зондування контрольних точок (Internal Test Points Probed); 69% – відновлення працездатності дефектних SiP. Типи несправностей: найбільший відсоток дефектів пов'язаний з порушенням працездатності кристалів. Технології автоматизації тестування: 46% – спеціальні тестові набори (Custom Test Sets); 37% – використання тестерів (IC Testers). Напрямки розвитку пакетних технологій: збільшення складності кристалу до 700 млн транзисторів на кристалі; зменшення площі кристалу; збільшення кількості виводів до 10 К; використання кулькових виводів з кроком 50 мкм.

1.3. Технології виготовлення і тестування цифрових систем

Загальна стратегія створення (виробництва) й тестування цифрової системи у пакеті (MCM – Multi Chip Module) містить пункти [7, 10], наведені на рис. 1.5.

1) Перевірка 300-мм пластини після її виробництва (виготовлення, сортування, розпилювання). Паралельно в часі виконуються пункти 2 і 3. 2) Створення підкладки (проектування, виробництво і тестування). 3) Створення кристалу (mount in carrier – розміщення кристалів на носії або кристалотримачі, випробування на наявність дефектів, повне тестування, видалення носія). 4) Створення (асемблювання) модуля MCM з раніше створених і перевірених компонентів (підкладки – substrate і кристалу die): складання, тестування і діагностування модуля, випробування, перевірка після випробування, інкапсуляція компонентів у пакет, остаточне тестування пакету модулів.

Тут же виконується відбраковування компонентів у кошик для сміття або на переробку, якщо це допускається технологією. Певний інтерес викликає сучасне виробництво кремнієвих пластин. Перший етап – формування монокристалу кремнію циліндричної форми.

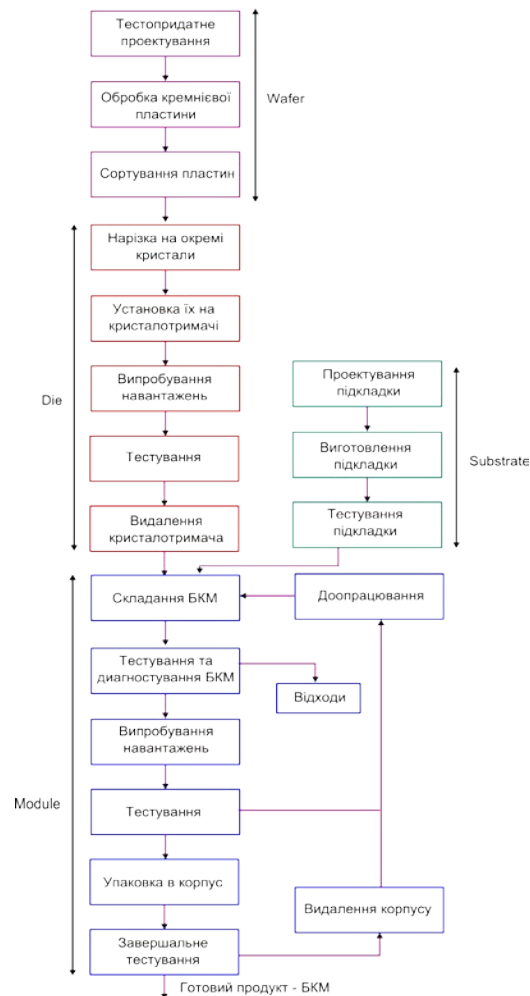


Рис. 1.5. Стратегія виробництва і тестування цифрової системи у пакеті

Він вирощується на основі початкового кристалу, який являє собою ідеальну (досконалу) структуру необхідної кристалографічної орієнтації. Початковий кристал занурюється у ванну (тигель) з розплавленим кремнієм. Тигель і кристал обертаються в протилежних напрямках для забезпечення рівномірного розподілу температури і домішок в об'ємі розплаву. Потім монокристал поволі витягується з розплаву. При цьому відбувається перехід атомів з рідкої або газоподібної фази речовини у тверду фазу на межі розділу і зростання монокристалу. У результаті виходить кремнієвий злиток високої чистоти заданого діаметра. В даний час для виробництва систем на кристалі використовуються заготовки діаметром 300 мм.

До 2013 року планується перейти на використання 450-мм кремнієвих пластин. Вартість одного нового заводу – 2 млрд. доларів. Другий етап – отримання кремнієвих пластин товщиною близько 0,6 мм шляхом розрізання монокристалу за допомогою кільцевої алмазної пилки.

Для отримання необхідної шорсткості поверхні пластини піддаються додатковій обробці: 1) шліфуванням за допомогою обертових сталевих дисків та абразивного матеріалу, при цьому усуваються нерівності висотою більше 10 мкм; 2) травленням для видалення похибок у структурі кристалу; 3) поліруванням для отримання дзеркально гладкої поверхні з нерівностями не вище 3 нм. Третій етап – нанесення захисної плівки діелектрику (діоксиду кремнію, SiO₂), який є добрим ізолятором і виконує захисну функцію при подальшій обробці підкладки. Четвертий етап – нанесення фоторезисту з метою формування на підкладці захисного рельєфу, який не піддається подальшому травленню. П'ятий етап – літографія. Під впливом ультрафіолетового випромінювання, що потрапляє на певні ділянки захисного шару через маску, фоторезист змінює властивості розчинності. Після опромінення фоторезист піддається прояву. Нова технологія літографії EUV (Extreme UltraViolet – наджорстке ультрафіолетове випромінювання), основана на використанні ультрафіолетового випромінювання з довжиною хвилі 13 нм, дозволяє формувати лінії шириною до 30 нм. Шостий етап – травлення з метою видалення плівки діоксиду кремнію з ділянок поверхні пластини, які повинні бути піддані подальшій обробці. При виробництві кристалів використовується метод сухого (плазмового) травлення з застосуванням іонізованого газу (плазми), що вступає в реакцію з поверхнею діоксиду кремнію. Відбувається руйнування діоксиду кремнію на ділянках, не вкритих захисним шаром, після чого решта фоторезисту видаляється. Сьомий етап – дифузія легуючих домішок шляхом іонної імплантації. Іони домішки «вистрілюють» з високовольтного прискорювача і, маючи достатню енергію, проникають у поверхневі шари кремнію. Таким чином, формується шар напівпровідникової структури. Для створення декількох шарів повторюють етапи з третього по сьомий. Восьмий етап – металізація. При формуванні шарів на окремих ділянках залишають області, які потім заповнюються атомами металу – провідні зони, що використовуються для встановлення зв'язків між шарами.

Пакетування електронних компонентів розглядається як частина комп'ютерного системного проектування. Найбільш відомою є Multichip module (MCM) технологія пакетування, яка інтегрує known good dies (KGD) на підкладці з високою щільністю розміщення компонентів. На основі цієї технології в дев'яності роки минулого століття розроблялися потужні і професійні (high-end) комп'ютерні системи в 2D виконанні. Наприкінці дев'яностих з'явилися 3D реалізації технології MCM з високою щільністю монтажу. Як розширення та розвиток MCM технологія SiP стає досить поширеною, спрямованою на зменшення форм-фактора, що робить її перспективною в найближчі 10 років. Технології тестування пакетів є життєздатними, продуктивними, але ефективними лише для простих пакетів. Що стосується багатокри-

сталі гетерогенні системні структури, що мають логіку, пам'ять, аналогові й радіочастотні компоненти, то їх мініатюрне виконання дуже очікуване на ринку, як виріб, що має такі переваги: невисока вартість і висока продуктивність, незначний час розробки та низькі енергетичні витрати.

Однак, існує ряд проблем в області design and test, які слід вирішувати. Необхідні узагальнені EDA засоби для інтегрування цифрових, мішаних mixed-signal і радіочастотних чипів, що містяться у SiP. Засоби для проектування та аналізу повинні вміти виконувати моделювання електричних і температурних режимів, а також використовувати KGD як компонент, хоча відсутність придатного кристала затримує проектування інтегрованої системи в цілому. Ще одна проблема пов'язана з розповсюдженням та імплементацією пасивних елементів у підкладку пакета, що має виконуватися під керівництвом SiP виробників, які зобов'язані ознайомити проектувальників з сімейством пасивних компонентів, що має високу актуальність для ефективного створення систем в пакетах та їх просування на ринок.

Стратегія тестування SiP відрізняється від перевірки SoC, оскільки система в пакеті інтегрує компоненти від різних виробників. Створення нових технологій тестування SiP є сьогодні пріоритетною проблемою, поставленою перед практичними вченими і дослідниками, які входять до design-and-test community. Найперспективнішою стратегією є використання стандарту IEEE 1500 для вбудованого тестування функціональних компонентів. Маючи високу продуктивність і рівень інтеграції компонентів у невеликому об'ємі, пакети містять декілька кристалів, з'єднаних провідними шинами, які негативно впливають на проходження сигналів у високочастотному спектрі. Інтеграція мультядерних гетерогенних систем у пакет також ставить багато питань перед design-and-test інженерами. Подолання таких бар'єрів вимагає об'єднання зусиль і засобів з боку фахівців і компаній, які займаються цифровим, аналоговим, RF- і тепловим (thermal) проектуванням. Існуючі пропозиції з тестування SiP пов'язані з адаптацією DFT-технології для перевірки вбудованих АЦП і ЦАП перетворювачів чисто цифровими тестами, які більш економічні для конвертерів. Таким чином, успіх цифрових систем у пакетах кристалів (SiP) на ринку електронних технологій залежить від вирішення проблем, пов'язаних зі створенням засобів проектування гетерогенних компонентів та їх ефективного тестування.

SiP дизайн є привабливою технологією для широкого кола застосувань, таких як: безпроводові пристрої, сенсори, графічні перетворювачі, мережеві пакетні перемикачі. З точки зору проектування і виробництва конструктив SiP пропонує гнучкі технології для створення та виготовлення пакетів з різномірних матеріалів: германій-кремнієвий кристал (SiGe) – для швидкодіючої логіки, CMOS – для аналогових компонентів і пам'яті. Таким чином, з'являються можливості створювати маленькі за розміром вироби високої продуктивності та збільшувати випуск придатних виробів. SiP також дозволяє істотно зменшити витрати на тестування, оскільки кожен тип кристала по-

винен мати власний вбудований тестер. На додаток до сказаного, кожен функціональний компонент повинен бути таким, що ремонтується або реконфігурується у разі виявлення дефектів. В даний час SiP дизайн не має інфраструктури сервісного обслуговування (як в SoC) для уніфікованого вирішення проблем проектування на основі моделі процесу, орієнтованого на спільне створення функціональностей кристалів пакета.

1.4. Тестопридатне проектування SiP на основі стандарту IEEE 1500

Система в пакеті інтегрує кілька кристалів, вироблених, як правило, різними постачальниками та розробниками. Тому тестування SiP-структури істотно відрізняється від перевірки системи на кристалі. Далі розглядаються стратегії, що базуються на використанні IEEE 1500 стандарту, і, на думку експертів [3], є перспективними при вирішенні проблем сервісного обслуговування SiP.

Вимоги ринку диктують компаніям розробляти вироби за метрикою: швидше, менше, дешевше. Тому виникли технології, здатні оперувати мініатюрними компонентами, інтегруючи їх у набір функціональних модулів на одному кристалі. Парадигма системи на кристалі гарантовано вирає у цифрових систем на друкованих платах за продуктивністю, розмірами й енергоспоживанням. SoC-виробники можуть розробляти операційні модулі самостійно або купувати готові функціональні блоки (як IP) для подальшого їх компонування у процесі проектування спеціалізованих цифрових систем. При цьому виробник повинен випускати SoC великими серіями з метою відшкодування витрат інженерної праці, необхідної для розробки кристала. З 1990-х років концепція SoC постійно зростає в частині її привабливості для ринку електронних технологій, завдяки інтеграції на одному чипі функціональностей у вигляді логіки, пам'яті, радіочастотних модулів і блоків живлення. З боку засобів проектування слід зауважити, що інструменти моделювання, верифікації та валідації здійснили гігантський стрибок у бік складності оброблюваних моделей. Тестування більш складних SoC вимагає таких платформ тестового обладнання, які пропонують більше функціональних можливостей, а також спеціалізованого вбудованого тестового обладнання, необхідного для нових маршрутів тестування SoC. Імплементація компонентів в SoC є вибір компромісу між оптимальними технологіями та бізнесцілями. Тому важливим є не тільки якнайшвидше вийти з новим виробом на ринок, а й мати технологічні засоби заміни несправних компонентів.

Переходячи до системи в пакеті, проектувальник і виробник отримують: автономні IP чипи, виконані у вигляді чистих кристалів, які нашаровані один на одного і мають внутрішньопакетні з'єднання. Кожен кристал може мати окрему функціональність або бути багатофункціональним і містити сукупність IP-cores. SiP увібрав у себе переваги як друкованої плати, так і системи на кристалі: 1) Компоненти можуть бути виготовлені окремо один від одного на основі різних технологій, а потім пакетовані. 2) Це означає, що вся система буде мати високу продуктивність і прості маршрути (фази) проектування.

3) SiP-структура добре масштабується і оптимізується, а також всі функціональні IP-модулі мають надійний захист авторських прав виробника. Стратегія «поділяй і пануй» (divide-and-conquer) тут підходить як для виробника, так і для споживача, які повинні дотримуватися жорсткого графіка створення продукту – від визначення специфікації до постачання готового виробу. Мільярди цифрових систем у пакетах вже знаходяться в руках споживачів; це, перш за все, мобільні комунікаційні вироби й різні види "твердої" пам'яті. Однак, тестування SiP все ще потребує пильної уваги з боку вчених та експертів. Активність фахівців спрямована на адаптацію SoC технологій тестування, що використовують регістри граничного сканування як навколишнє середовище (core wrapper) і механізми подачі тестів (test access mechanisms) як складові частини тестопридатного проектування. Для систем у пакетах сценарій тестування відрізняється від описаного вище, оскільки він орієнтований на два рівні опису системи: тестування кристалів окремо, а потім – остаточна перевірка асембльованої системи в цілому. Також у SiP є надлишкові нефункціональні компоненти, що забезпечують підвищення керованості та спостережуваності сигналів та інтегруються у стратегію тестування системи в цілому. Для цього створюються спеціальні засоби і структури (подібно I-IP для SoC), що обслуговують функціональні блоки системи, у тому числі аналогові та мішані – mixed-signal, технологічні між'єднання кристалів, а також проектні обмеження й технічні умови, що використовують новий стандарт 1500 Standard for Embedded Core Test (SECT).

Парадигма тестування SiP базується на передумові існування завідомо справних кристалів – known-good-die. Це означає, що на рівні системи в пакеті не завжди існує можливість протестувати всі функціональності та структурні зв'язки, як це можна зробити на рівні окремого кристала. Для цього існують кілька причин: обмежений доступ до всіх тестових режимів кристалів, зважаючи на існування лімітів на часові та матеріальні витрати. Інші невирішені проблеми відносяться до технології тестування з'єднань усередині підкладки, яка має стійку тенденцію до ускладнення структури і збільшення кількості внутрішніх компонентів, активних і пасивних. Останні елементи використовуються, у тому числі, для з'єднання окремих кристалів у систему. Крім того, важливо мати повні тести для між'єднань на рівні одного кристала, а також тести верифікації, чутливі до параметрів пакетування кристалів.

В даний час SiP архітектури орієнтовані на реалізацію ручних, обчислювальних та мобільних пристроїв, які потребують мінімального простору і злиття функціональностей, у тому числі цифрових, аналогових модулів та керованих обчислювальним процесом блоків живлення. У такій конфігурації SiP має кілька кристалів різної природи і підтримує такі режими: 1) Один чи декілька обчислювальних пристроїв (процесори, спеціальні обчислювачі, кодеки, периферійні пристрої, з'єднані шинами) здійснюють поставку користувачеві своїх функціональностей. 2) Автономні кристали пам'яті (Flash, E2PROM, DRAM), вироблювані роздільно, потім можуть бути асембльовані для одержання більш складних архітектур. 3) Аналогові кристали і чипи жи-

влення надають системам у пакетах можливість взаємодіяти з електромеханічними компонентами, а також здійснювати виведення інформації на екран і аудіосистему. 4) RF кристали надають системі можливість взаємодії з працюючим обладнанням і забезпечують збереження інформації, отриманої у процесі обчислень. 5) Підкладка служить не тільки як структура міжз'єднань кристалів на основі застосування активних і пасивних елементів, але зазначена область може бути використана для підвищення повноти тестової стратегії SiP.

Процес виробництва SoC і SiP для стадій створення кристалів має більше спільних моментів, ніж відмінностей, що також обумовлює адаптацію для SiP методів SoC тестування. Більш того, методи і механізми перевірки компонентів на рівні тестування кристалів є загальними для SoC і SiP [10]. Другий рівень тестування SoC використовує системний тест, орієнтований на перевірку інтерфейсів, міжз'єднань і всіх структур в цілому. Оскільки SiP має багатокристалну архітектуру, маршрут системного тестування складатиметься з таких пунктів: 1) Визначення вимог до повноти тестування підсистем. Кожен вироблений кристал має власний інтерфейс тестування і точний порядок виконання тестів. 2) Визначення вимог до якості перевірки системи. SiP архітектура обумовлює два важливих аспекти. По-перше, тестування кристала може бути орієнтоване на загальну структуру SiP у взаємодії з іншими компонентами, коли функціональність одного кристала SiP може використовуватися при тестуванні іншого. По-друге, необхідно мати ефективні алгоритми аналізу результатів тестування міжз'єднань кристалів, використовуючи архітектуру інтерфейсів, доступну для підсистем тестування.

Останній рівень виробничого тестування відображає структурні відмінності між SoC і SiP. Тут тестова стратегія, розроблена для всієї системи, переводиться в Automatic Test Equipment (ATE) формат: рівень 1 – опис тестової програми і тестових впливів; рівень 2 – тестові програми, згенеровані для перевірки міжз'єднань між кристалами і завершального тестування всієї структури SiP.

Стандарт IEEE 1500 SECT є ефективним рішенням виробничого тестування SoC. У ньому зосереджено методологію передавання тестових даних, яка не залежить від функціональності системи і дозволяє виконувати «plug-and-play» тестування. Стандарт визначає: масштабовану структуру інтерфейсу тестування для IP-cores, що входять до складу SoC; мова тестування IP-core – Core Test Language – CTL, яка дозволяє описати архітектуру тесту і протоколи обміну даними між фірмами-виробниками IP-core і системним інтегратором.

На рівні інтеграції цифрової системи на кристалі IP-core, поміщений у тест-інтерфейс (wrapper) IEEE 1500, підключається до спеціальної, визначеної користувачем інфраструктури тестування завдяки наявності механізму тестового доступу (Test Access Mechanism – TAM). Тестування здійснюється через тест-інтерфейс IEEE 1500, за допомогою якого зовнішнє обладнання

передає тестові набори і отримує відгуки в установленому порядку. Використані для SoC рішення, основані на стандарті IEEE 1500, можуть бути поширені на парадигму SiP шляхом застосування представлених в стандарті структур для тестування окремих кристалів, що утворюють систему в пакеті, гарантуючи, таким чином, можливість реалізації тестування на рівні кристала і на системному рівні. Розміщення тест-інтерфейсу на кристалі дозволяє поліпшити якість проекту, незважаючи на те, що часові параметри та апаратурні витрати виробу можуть дещо погіршитися.

Запропоноване у роботі [3] рішення з організації тестового доступу для SiP базується на ієрархічному підході, представленому у стандарті IEEE 1500. Переваги зазначеного підходу: 1) проста й швидка взаємодія тесту з шарами IP-cores і підсистемами незалежно від типів компонентів, спрощення проектування структури тесту для кожного модуля та приєднання до компоненту TAM; 2) ефективна підтримка тестування з'єднань між кристалами в пакеті; 3) визначення стандартного підходу до генерації тестових програм на рівні кристала і SiP в цілому. На рис. 1.6 наведено можливу конфігурацію тесту для SiP, що складається з 2-х кристалів.

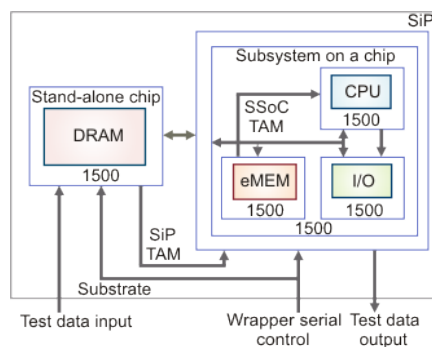


Рис. 1.6. Механізм TAM для двокристалльної системи в пакеті

Тут використовується один послідовний канал для реалізації механізму тестового доступу (мінімум, визначений стандартом IEEE 1500). Запропонована структура може бути використана у двох випадках: коли кристал тестується окремо шляхом його підключення безпосередньо до автоматичного тестового устаткування (ATE) і для тестування функціональності SiP в цілому.

Тест-інтерфейс забезпечує можливість формування тестових даних для кожного IP-core та отримання результатів, а також виконання необхідних перетворень формату даних для передавання їх у TAM, починаючи від введення тестових наборів і закінчуючи їх виведенням. Шестирозрядна шина забезпечує керування послідовними структурами інтерфейсу шляхом передавання даних до кожного кристалу на основі стандартизованого протоколу.

На рис. 1.7 наведено пропонувану структуру інтерфейсу тестування на основі стандарту IEEE 1500. Стандарт вимагає імплементації мінімального набору структур: 1) Регістр граничного сканування (Wrapper Boundary Register,

WBR) діє як boundary scan коло на рівні IP-core. 2) Буферний регістр (Wrapper Bypass Register, WBY), зазвичай складається з одного тригера, який дозволяє направляти тестові дані до іншого IP-core через інтерфейс. 3) Регістр команд (Wrapper Instruction Register, WIR), що приймає кодове слово для активізації одного з можливих регістрів сканування шляхом перемикачів мультиплексорів. На рис. 1.7 показано кола сканування, додані в IP-core, які підімкнено до системи і додаткового паралельного порту для організації тестового доступу.

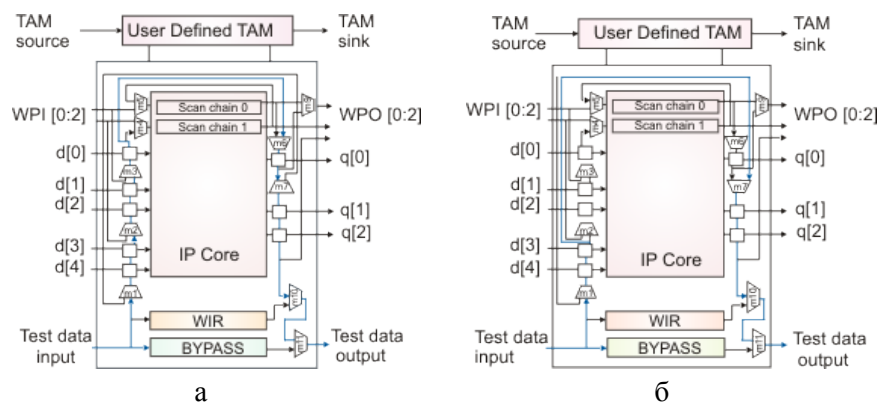


Рис. 1.7. Boundary scan інтерфейс для тестування SiP: а – повний доступ до кола граничного сканування; б – тестування міжз'єднань кристалів

Представлена архітектура інтерфейсу забезпечує два режими функціонування, орієнтовані на тестування SiP: перший (рис. 1.7, а) – надає повний доступ до кола граничного сканування, коли є можливість тестування міжз'єднань IP-cores усередині підсистеми на кристалі до і після складання SiP; другий (рис. 1.7, б) – орієнтований виключно на тестування міжз'єднань між кристалами і надає доступ тільки до комірки граничного сканування на первинних входах і виходах підсистеми на кристалі. Для наведеного прикладу лінії: d[3] – d[4] та q[1] – q[2]. Описана конфігурація повторюється для всіх IP-cores підсистеми на кристалі, забезпечує виконання функціональностей інтерфейсу IEEE 1500, що оточує кристал в цілому, без введення додаткового апаратного забезпечення або погіршення характеристик. Вибір конфігурації комірки граничного сканування – задача рівня тестування пакетованої системи. Незважаючи на те, що розроблений раніше кристал, інтегрований у систему, може згодом втратити структури тестового доступу, що відповідають стандарту IEEE 1500, є можливість отримати доступ до верхнього рівня кристала шляхом використання комірки граничного сканування, яка відноситься до іншого кристала системи. Це дозволяє застосовувати тестові набори, призначені для тестування з'єднань між кристалами на основі інтерфейсу IEEE 1500, що є складовою проекту SiP.

SiP надає велику різноманітність можливих проектних рішень, в яких використовуються компоненти, виготовлені різними виробниками за різними технологіями. Навіть для кристалів, що задовольняють тестовим структурам стандарту IEEE 1500, безліч додаткових чинників можуть впливати на розробку тестових програм. Залежно від доступного автоматичного тестового обладнання, конфігурації проекту і вимог до нього, може бути використана стратегія з одним або декількома проходами. Для прискорення тестової інтеграції в процес розробки SiP можуть застосовуватися програмні засоби автоматичної генерації тестів. Основні параметри тестових програм: 1) перевірка сумісності описаних тестових архітектур та існуючих обмежень на доступні ATE і споживану потужність в процесі тестування; 2) швидка оцінка часу тестування SiP для заданої конфігурації тесту; 3) автоматична генерація тестових програм для SiP заданою мовою, включаючи форми сигналів, які отримані і передані в ATE для виконання тесту. Основна перевага використання сервісного програмного забезпечення полягає у можливості автоматичної генерації тестових програм на рівні кристала або системному рівні на основі описів компонентів.

Як приклад у [3] розглянуто SiP, розроблену для супутникового зв'язку. Архітектура SiP містить підсистему на кристалі для прямого немодульованого (base-band) передавання даних, розроблену за 90nm технологією, та один або два додаткових елементи пам'яті: буферна пам'ять об'ємом 128 Mbyte, dynamic RAM і енергонезалежна (E2PROM). До складу системи входять 4 кристали, що реалізують RF інтерфейс і перетворення інформації (рис. 1.8). Під час тестування системи були використані рішення, описані вище, і засоби автоматичної генерації тестових програм на всіх етапах виробничого тестування SiP. Тестування підсистем на кристалі виконувалося через порт тестового доступу (TAP). Для структурного тестування SoC необхідна пам'ять великого об'єму, а також аналогові засоби функціональної перевірки вбудованих функцій обробки сигналів. На рівні підкладки для тестування SoC за допомогою тестера Agilent 93K потрібно 7 секунд на один кристал. Зазначений час збільшується на 25%, якщо система на кристалі містить RF компоненти. Більш того, система в пакеті піддається випробуванням при різних температурах від 25 до 125 ° C. Для тестування елементів пам'яті SiP були використані стандартні засоби тестування і DFT. Для пам'яті E2PROM тестування на рівні підкладки виконувалося на спеціальному обладнанні з використанням тестера Credence Kalos. Тестування DRAM здійснювалося із застосуванням стандартного автономного тестера серії Agilent V54 або Advantest. Для розглянутої системи було використано тестер Agilent 93K, в якому реалізовано метод паралельного тестування, що базується на застосуванні мультипоточного обміну даними. Метод дозволяє подавати кілька незалежних тестів на різні групи виводів паралельно, а також ізолювати окремі кристали при тестуванні або тестувати їх окремо, використовуючи стандартну інфраструктуру.

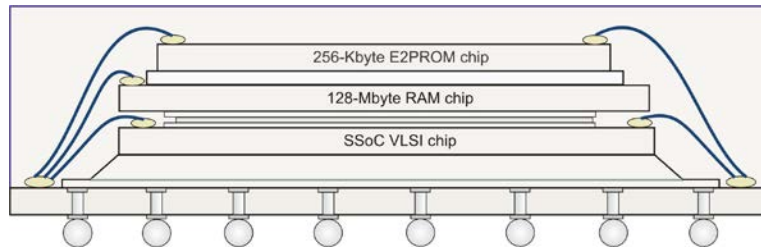


Рис. 1.8. Система в пакеті для супутникового зв'язку

1.5. Мови опису апаратури

VHDL – аббревіатура від англійського виразу VHSIC Hardware Description Language. У свою чергу, VHSIC походить від назви програми Very High Speed Integrated Circuit (високошвидкісні інтегральні схеми). Ця програма, профінансована Міністерством Оборони США, ставила за мету розвиток нового покоління високошвидкісних інтегральних схем. Перша версія мови була представлена у 1985 році. Згодом він був переданий товариству IEEE для стандартизації. У 1987 році мову було затверджено як стандарт IEEE 1076-1987. Через п'ять років він був розглянутий повторно, в результаті чого нова версія 1076-93 містить ряд додаткових можливостей. Відтоді VHDL користується постійно зростаючою популярністю серед фахівців з автоматизованого проектування (CAD) електронних систем. Перші VHDL-додатки з'явилися на початку 90-х років. Для синтезу був розроблений пакет IEEE 1164. На практиці кожен великий виробник систем автоматизованого проектування підтримує VHDL. Мова має академічну спрямованість, популярна в університетських лабораторіях.

VHDL дозволяє описувати системи на різних рівнях. Вона реалізує методологію низхідного проектування, в якій система спочатку описується на високому рівні та тестується за допомогою засобів моделювання, після чого поетапно вона зводиться до структурного опису, тісно пов'язаного з фактичною апаратною реалізацією. Мова VHDL є інваріантною до різноманіття технологій реалізації. Якщо певний проект описано мовою VHDL і реалізовано для конкретної електронної технології, то VHDL-опис може бути використаний як початковий для реалізації пристрою в новій технології.

Verilog, або Verilog HDL - мова опису апаратури, розроблена у 1985 році Філіпом Мурбі (Philip Moorby), який потребував простого, наочного та ефективного способу опису цифрових схем, моделювання та аналізу їх функціонування. Мова стає власністю Gateway Design Automation, а потім Cadence Design Systems. Перший стандарт Verilog з'являється в IEEE у 1995 році. Останнє оновлення мови виконано у 2005 році. Verilog має свою законну нішу на ринку EDA і більше ніж VHDL популярний у промисловості при проектуванні ASIC-пристроїв.

На рис. 1.9 наведено схему конструкцій, доданих у мову Verilog на різних етапах її розвитку. Як ядро виділено базові елементи, визначені стандартом

1995 року. Власне, вони є тією підмножиною, на якій розроблено основну масу Verilog-проектів.

SystemVerilog					
User Defined Port	Block Labeling	int	enum	break	continue
State Machine	Dynamic Progress	shortint	typedef	return	goto
Interface	Assertion	longint	struct union	++ - += -= *=	
Packed Array	2/4 State Variable	shortrea	casting	/= >>= <<=	
Structure	Timeunits	double	const	&= = ^= %=	
Union	unique/priority fork/case/ if	char	void		
ANSI C style port	standard file I/O	(* attribute*)	multidimensional array		
generate	Svalue \$plusarg	configuration	signed unsigned		
localparam	`ifndef `elsif `line	memory part select	automatic		
const func	@*	variable part select	**(power operator)		
module	Sfinish \$fopen \$close	initial	wire reg	begin end	+ = * /
parameter	Sdisplay \$write	disable	integer real	while	%
function/task	Smonitor	event	time	for forever	>> <<
always @	`define `ifdef `else	wait # @		packed array	if else
assign	`include `timescale	fork join		2D memory	repeat

Verilog 2001

Verilog 1995

Рис. 1.9. Схема конструкцій мови System Verilog

Далі, як розширенням до ядра слугують конструкції, які доповнюють мову, завдяки стандарту 2001 Verilog 1364-2001. І остання частина – конструкції, що утворюють SystemVerilog. Ці елементи призначені для створення складних ієрархічних проектів, а також для опису додаткових типів даних та операторів мови C. SystemVerilog розроблено у зв'язку з постійним збільшенням розмірів проектів для опису пристроїв на системному рівні.

1.5.1. Мови проектування й верифікації SoC.

Різні мови розроблялися для різних аспектів проектування систем. C/C ++ переважно використовуються для розробки програмного забезпечення вбудованих систем. HDL-мови опису апаратури, VHDL і Verilog застосовуються для проектування та синтезу цифрових схем. Мови Vera і e призначені для функціональної верифікації складних спеціалізованих інтегральних схем (ASIC). Нова мова SystemVerilog є доповненою версією мови Verilog, призначеної для розв'язання типових задач апаратно-орієнтованого системного проектування. Середовище Matlab, інші подібні інструменти і мови, такі як SPW і System Studio, широко використовуються для формування системних вимог і розробки алгоритмів цифрової обробки сигналів. На рис. 1.10 подано галузі використання мов системного проектування. SystemC був створений для проектування SoC – пристроїв на ESL-рівні (Electronic System-Level) для розробників, що використовують C/C ++. Стандарт: IEEE Std. 1666-2005 SystemC (Open SystemC Initiative, OSCI). SystemC – це єдина мова для проектування та верифікації, яка дозволяє зображати архітектурні та інші атрибути моделей системного рівня у формі класів мови C ++ з відкритим вихідним кодом.

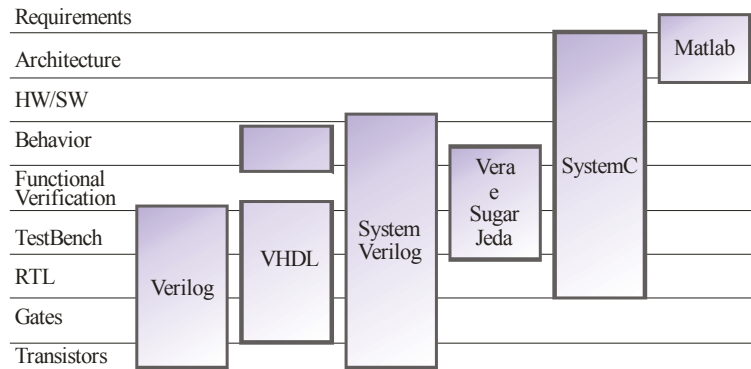


Рис. 1.10. Порівняння галузей застосування мов системного проектування

Це дає можливість виконувати проектування та верифікацію на системному рівні, незалежно від будь-яких деталей реалізації апаратури і програмного забезпечення. Також є можливість виконання спільної верифікації з RTL-проектами. Високий рівень опису дозволяє значно швидше й продуктивніше виконувати аналіз вибору компромісного рішення для архітектури, ніж це можна зробити на RTL-рівні. Більш того, верифікація системної архітектури та інших системних атрибутів відбувається швидше, ніж при використанні точної за часовими параметрами або розташуванням зовнішніх контактів RTL-моделі.

SystemVerilog – перша мова опису та верифікації апаратури (HDVL), яка є розширенням стандарту IEEE 1364 Verilog-2001 [12] і дозволяє значно підвищувати продуктивність проектування пристроїв, що складаються з великої кількості вентилів та мають велику навантаженість шини. Стандарт IEEE Std. 1800-2005 [11] являє собою мову SystemVerilog (Accellera), призначену, головним чином, для створення безперервного процесу проектування SoC, починаючи від поведінки і закінчуючи GDSII-описом. Однак, SystemVerilog також підтримує створення TLM-моделей. Такий збіг властивостей верифікації між SystemC і SystemVerilog є основою для зв'язків між системним рівнем і реалізацією кристала. Модуль SystemVerilog's Direct Programming Interface (DPI) дозволяє викликати функції C/C++/SystemC, і навпаки, виконувати спільне моделювання (див. рис. 1.10) Verilog-моделей з SystemVerilog і SystemC блоками. SystemVerilog підтримує повне середовище верифікації, використовуючи Constraint Random Generation, Assertion Based Verification і Coverage Driven Verification. Вона може бути застосована для HW-проектуювання та верифікації. SystemVerilog може бути частиною середовища Verilog or VHDL TestBench (TB), частиною проекту або зовнішнього модуля. Наприклад, проектувальник може використовувати можливість Random Generation шляхом додавання деяких властивостей SystemVerilog Random до поточного TB без необхідності мати повне середовище верифікації. Таким же чином інженери-проектувальники можуть використовувати деякі асерції, як частину повного симулятора кристала, для верифікації правильності шини або протоколів.

Мова e – перша комерційна мова верифікації. Була створена розробниками Yoav Hollander і Amos Noy. Основна ідея її появи – надання можливості створення

специфікації проекту (*e* для English), його тестування та верифікації. Для роботи з мовою *e* компанією Verisity було розроблено інструмент Specman, до складу якого входив компілятор і debugger. Мова *e* використовується також для створення testbench проектів, при цьому інструмент Specman повинен бути підключений до засобу моделювання HDL-коду. Пізніше компанію Verisity було куплено фірмою Cadence, а Specman було додано до більш складного пакету для верифікації Incisive, що розробляється Cadence. Мова *e* – це перша мова, до складу якої було введено псевдовипадковий генератор, генератор тестів і можливість перевірки результатів моделювання, що дало можливість описувати testbench за допомогою однієї мови. Раніше для генерації псевдовипадкових послідовностей використовувалися додаткові засоби, що підключалися до тестованого пристрою у вигляді спеціальних модулів. Мова *e* породила нову концепцію програмування – Aspect Oriented Programming (AOP). Це дало можливість користувачам мови *e* розширювати код, додавати додаткові константи, додавати або перезаписувати функціональності. Різниця між Object oriented і Aspect oriented технологіями полягає в можливості виконувати розширення функціональності тільки для деяких аспектів базових класів.

Мова верифікації Vera – це розробка компанії Sun Micro Systems, яка використовувала його для внутрішньої верифікації ASIC-проектів. Пізніше мова VERA з компілятором VERA були продані фірмі System Science, від якої вони згодом перейшли до компанії Synopsys. В даний час компанія Synopsys випустила нову версію мови Vera – OpenVera, яка підтримується VCS компілятором.

Методологія верифікації SystemVerilog ґрунтується на трьох типах блоків, які можуть застосовуватися окремо або всі разом: 1) Stimuli. Проект, який використовує автоматично генеровані сценарії з псевдовипадковими послідовностями (constrained-random – CR) генерування тестів. 2) Check. Перевірка функціональної поведінки проекту (асерції) та вихідних даних (scoreboard) на предмет верифікації правильності операцій. 3) Measure. Метрика функціонального покриття для забезпечення зворотного зв'язку при генеруванні та аналізі виконання процесу верифікації. Використання методів верифікації, які охоплюють процес моделювання, налагодження та оцінки покриття, називається Coverage Driven Verification.

1.5.2. Основи методології TLM.

Основною концепцією TLM (Transaction Level Modeling) є створення моделі рівня деталізації, достатнього для розв'язання задачі проектування компонентів системи. Аналіз лише окремих деталей дозволяє значно підвищити швидкість верифікації моделей. Концепція TLM є незалежною від будь-яких мов, однак ґрунтується на мовах високого рівня програмування, наприклад, таких як SystemC або SystemVerilog. Transaction Level Modeling пропонує bit-true, address-map accurate, cycle-less моделі. TLM визначає транзакцію (SoC-подію), як миттєву передачу даних (комунікацію) або синхронізацію між двома модулями, визначеними hardware/software системною специфікацією. Термін TLM визначений для відповідного ESL (Electronic System Level) рівня проектування [17-21]. Передбачається використання мов програмування високого рівня для моделювання апаратури на високому концептуальному рівні, не вдаючись у деталі її ре-

алізації. До таких мов можна віднести SystemC, SpecC, Hspascal, System Verilog, HardwareC.

Моделі абстракції. Основна концепція TLM – комунікації між системами й підсистемами та їх функціональність можуть бути розроблені і деталізовані незалежно один від одного. За принципом деталізації комунікацій і функціональностей моделі можуть бути поділені на Un-timed (UT), Approximately-timed (AT), Cycle-timed (CT) (рис. 1.11) [18].

Моделі, які підтримують детальний потактовий (cycle-timed) опис функціональностей і комунікацій, є структурами рівня передачі даних (RTL – register-transfer level). Моделі без використання деталізації часу при описі функціональностей і комунікацій (un-timed-моделі) називаються SAM (System Architectural Model) моделями. SAM-модель є засобом комунікації між групами проектувальників алгоритмів, програмного забезпечення та апаратури.

До інших чотирьох структур, представлених на рис. 1.11 та позначених TLM, належать моделі з апроксимованим часом (approximately-timed). Вони використовують статичний або розрахунковий час, або часові вимоги підсистем.

У табл. 1.4 наведено список моделей абстракції, які використовуються при проектуванні SoC. BFM (Bus Functional Model) модель є найпоширенішою з TLM-моделей. Інші три – не мають загальноприйнятих імен. Для них використовуються імена, визначені в публікації Gajski і Cai [18]. На практиці всі запропоновані моделі не є обов'язковими при проектуванні, як правило, використовуються дві або три з них. На початковій стадії проектування немає необхідності точного опису процесу передавання даних через шину, тому застосовується модель, у якій цей крок виконується за одну подію або транзакцію (component-assembly model).

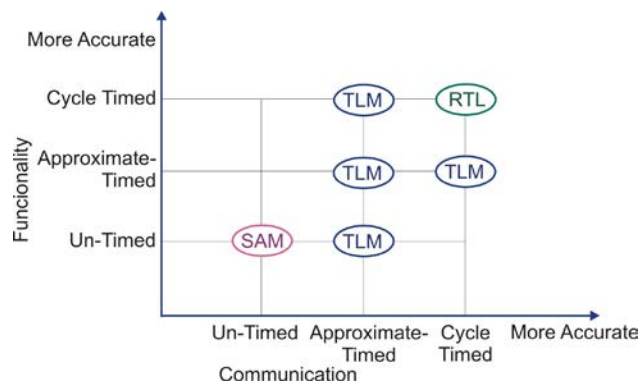


Рис. 1.11. Моделі абстракції представлення цифрових пристроїв

На наступному етапі проектування може виявитися важливим знання точної кількості тактів при передаванні інформації для кожного сигналу всередині блоків. Однак процес передачі даних по шині розглядається при цьому як одна транзакція або подія (моделі bus-arbitration або cycle-accurate). Коли специфікація стає повністю визначеною, шина моделюється за допомогою транзакцій або подій для зміни сигналу (bus functional or RTL model).

Таблиця 1.4. Моделі цифрових пристроїв

Модель		Комунікації	Функціональність
SAM		UT	UT
T L M	Component assembly	UT	AT
	Bus arbitration	AT	AT
	Bus functional (BFM)	CT	AT
	Cycle-accurate computation	AT	CT
RTL		CT	CT

У табл. 1.4 модель component assembly використовує одну подію (1 event), якій в моделі bus arbitration відповідають п'ять подій (5 events), а в RTL-моделі – близько 75 подій (75 events). Точна кількість тактів залежить від алгоритму моделювання. Діаграма ілюструє вимоги до обчислень у різних моделях і причину того, що багато команд проектувальників працюють з TLM-моделями.

Методологія проектування з TLM. Нова технологія проектування складається з двох частин: стандартної – від RTL до створення або прошивки мікросхеми (layout flow) і додаткової – від системного рівня до RTL. Проектування починається зі створення системної специфікації, що представляється у вигляді документа на папері. Грунтуючись на специфікації, системний проектувальник виконує поділ архітектури пристрою на hardware/software складові. Наступним етапом є TLM. Одним з призначень TLM є забезпечення фази паралельної розробки hardware/software. У цьому випадку TML служить універсальною вихідною точкою для software і architecture команд проектування, для забезпечення раннього проектування програмного забезпечення та структурного аналізу архітектури. Вона також використовується командами тестування для розробки середовища верифікації та тестів, забезпечуючи, таким чином, їх готовність на момент розробки RTL-платформи. Тим часом проектувальники hardware створюють RTL-проект для системи, розробленої на основі SoC RTL-платформи. Таким чином, досягається hardware/software спільне проектування, що є одним з основних відмінностей нової методології від класичної. Оскільки для проектувальника також є доступною RTL платформа, з її допомогою можуть бути вирішені задачі: перевірки сумісності при моделюванні проекту з очікуваними параметрами, hardware верифікації при низькорівневій інтеграції software з hardware. Ці задачі вирішуються паралельно з кроками емуляції, синтезу та аналізу результатів імплементації.

Потрійна абстракція. Нова методологія проектування визначає структуру потрібної абстракції: функціональне, архітектурне та мікроархітектурне представлення SoC. 1) Функціональне представлення SoC (SoC Functional View). Найвищий рівень абстракції, визначає поведінку системи з точки зору користувача. Функціональне представлення розробляється без внесення будь-яких деталей реалізації (архітектури чи адресного простору). Модель являє собою програми, що реалізують алгоритм функціонування пристрою. Параметри продуктивності системи описуються окремо у вигляді паперової специфікації. 2) Архітектурне представлення SoC (SoC Architecture View). Є більш низьким етапом проектування, який реалізується TLM-моделями. Крім того, архітектурне представлення

застосовується інженерами верифікації як довідкова модель, яка виступає в ролі золотого еталона для генерування функціональних тестів, використовуваних для верифікації пристрою на наступних стадіях проектування. 3) Мікроархітектурне представлення SoC – це найнижчий рівень потрібної абстракції. Містить всю інформацію для виконання часового моделювання за синхротактами (cycle-accurate). Загальноприйнятою практикою для цього рівня є використання RTL-моделей, створених за допомогою мов опису апаратури, таких як VHDL [13-15] або Verilog [15,16]. У [19] ілюструються ситуації використання мікроархітектурного представлення SoC, а в [20] описуються шляхи застосування багаторівневих моделей для створення проектів.

У загальному випадку сучасний процес проектування може бути представлений схемою, зображеною на рис. 1.12. Проектування починається зі створення Product Requirement Document (PRD), що складається замовником або маркетинговою групою. На основі PRD розробляється SAM, на цьому етапі вносяться необхідні зміни і уточнення до PRD. Потім SAM перетворюється в один з видів TLM-моделей, наприклад, спочатку в component assembly модель, а потім у bus arbitration модель. При виборі відповідної мови для SAM така модель може бути повторно використана (reuse) та шляхом деталізації перетворена у TLM.

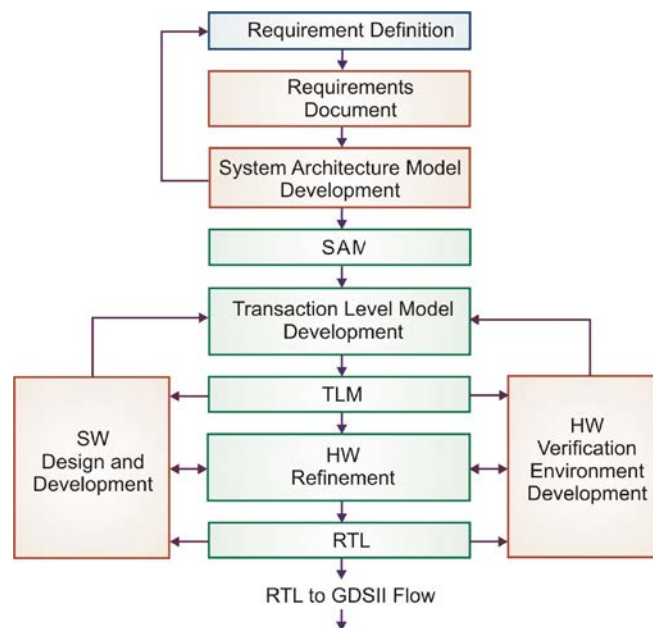


Рис. 1.12. Сучасні етапи проектування SoC

Задачі, що вирішуються з допомогою TLM-моделей: 1) Визначення властивостей імплементації, таких як HW/SW partitioning; HW partitioning між ASIC, FPGA та board реалізаціями; розробка архітектури шини; визначення або вибір сопроцесора. 2) Розробка платформи для системного програмного забезпечення.

3) Створення золотого еталона для функціональної верифікації апаратури. 4) Розробка мікроархітектури апаратури та основ для створення детальної специфікації hardware. До переваг використання TLM-моделей відносяться: можливість більш ранньої розробки software; раннє і якісне створення testbench для виконання функціональної верифікації; наявність єдиного і безперервного шляху проектування від вимог користувача до розробки детальної специфікації hardware і software.

1.6. Висновки за технологіями реалізації SoC

1. Мільярди компактних комунікаційних, обчислювальних виробів і пам'яті, що використовуються у діяльності людини, дозволяють зробити висновок про повне домінування цифрових систем у пакетах (кристалах) на ринку електронних технологій вже в найближчі п'ять років. Еволюційний шлях перетворення SoC в SiP є добрим прикладом для адаптації існуючих технологій тестування цифрових систем на кристалах до нового конструктивного покоління – цифрових систем у пакетах.

2. Актуальною для ринку електронних технологій є проблема створення технологій, моделей і методів системного проектування цифрових виробів у пакетах (кристалах) на основі використання HDL-мов (Verilog, System Verilog) і стандартів тестопридатного проектування програмних продуктів у вигляді механізму асерцій.

3. Представлені порівняльні характеристики мов опису апаратури. Мова Verilog призначена для ефективного опису функціональних компонентів і інфраструктури діагностичного обслуговування на основі IEEE 1500 стандарту, testbench-технологій, синтезу та імплементації цифрових систем на кристалах. Мова System Verilog – для опису специфікації функціональності SoC і асерційної надлишковості з подальшою верифікацією проекту.

4. Сформульовано найбільш істотні й актуальні, з точки зору авторів, задачі в галузі вивчення та застосування HDL-мов: 1) Класифікація та відмінності синтаксичних і семантичних особливостей мов опису апаратури. 2) Системний опис ієрархічної інфраструктури сервісного обслуговування цифрових систем у пакетах на основі IEEE стандартів тестопридатного проектування як доповнення до основної функціональності. 3) Розробка та опис системних моделей і технологій для тестування, верифікації та виправлення помилок у HDL-коді. 4) Ефективне використання системних бібліотек провідних компаній для верифікації та синтезу цифрових виробів, а також компонентів SiP: пам'ять, логіка, програмні та керувальні модулі.

1.7. Тестування та верифікація HDL-моделей компонентів SoC

У підрозд. 1.7-1.12 пропонуються технології тестування та верифікації цифрових систем для діагностування та виправлення помилок HDL-моделей, що базуються на спільному використанні механізму асерцій і тестопридатного проектування. Запропоновані моделі та методи дозволяють істотно підвищити якість

проектованих компонентів цифрових систем на кристалах і зменшити час розробки.

Провідні компанії планети, такі як Cadence, Synopsys, Mentor Graphics, велику увагу приділяють системній стадії створення цифрових виробів на кристалах, яка практично не піддається автоматизації. Тут на процес налагодження програмного коду втрачається до 70% загального часу, який визначається як time-to-market. Пропонується технологія тестування та верифікації системних HDL-моделей, орієнтована на суттєве підвищення якості проєктованих компонентів цифрових систем на кристалах (yield) і зменшення часу розробки (time-to-market) шляхом використання середовища моделювання, тестопридатного аналізу логічної структури HDL-програми та механізму асерцій.

Технологія дозволяє здійснювати пошук помилок із заданою глибиною в програмному HDL-коді за прийнятний для розробника час шляхом введення у критичні точки програмної моделі асерційної надлишковості. Критичні точки обчислюються за допомогою синтезованих логічних функцій тестопридатності. Останні визначають якість програмного коду шляхом синтезу диз'юнктивної нормальної форми (ДНФ), для якої оцінка за Квайном формує кількісну характеристику тестопридатності компонента або програми, представленої послідовністю операторів. Використовувані в hardware design and test критерії тестопридатності (спостережуваність та керованість) застосовані для оцінки якості програмного коду з метою його покращення та ефективного діагностування семантичних помилок.

Мета – поліпшення технології тестування та верифікації цифрових систем для діагностування та виправлення помилок HDL-моделей шляхом синтезу транзакційного графа програми, спільного використання механізму асерцій і технологій тестопридатного проєктування стандарту IEEE 1500.

Задачі дослідження: 1. Класифікація технологій тестопридатного проєктування та верифікації системних HDL-моделей для створення цифрових систем на кристалах. 2. Розробка узагальненої моделі верифікації та тестування системної HDL-моделі на основі використання асерцій як адаптація технологій IEEE 1500 стандарту до перевірки програмного HDL-коду. 3. Розробка метрики оцінювання тестопридатності HDL-моделей на основі синтезу логічних функцій тестопридатності. 4. Застосування моделі асерцій для верифікації IP-core фільтра дискретного косинусного перетворення. 5. Практичні результати й подальші дослідження.

Джерела дослідження: 1. Технології та засоби створення тестів і testbench представлені в роботах [22-23]. 2. Моделі й методи верифікації системних моделей на основі механізму асерцій описано в публікаціях [24-27]. Тестопридатне проєктування програмних продуктів використовує стандарти IEEE [28-30], а також інноваційні рішення для верифікації та аналізу тестопридатності системних HDL-моделей [31-38].

1.8. Тестопридатність програмно-апаратних продуктів

Ініціювальним ядром появи нових технологій тестування та верифікації в програмній та комп'ютерній інженерії слід вважати силіконовий кристал, що є ос-

новою для створення обчислювальних та/або комунікаційних пристроїв. Кристал розглядається як випробувальний полігон для апробації нових засобів і методів трасування, розміщення, синтезу та аналізу компонентів. Технологічні рішення, які витримали випробування часом у мікроелектроніці, далі захоплюються і адаптуються до макроелектроніки, представлені комп'ютерними системами та мережами. Ось деякі історичні факти, пов'язані зі спадковістю розвитку та взаємопроникнення технологічних інновацій у програмних і апаратних продуктах.

1. Стандарти граничного сканування [24, 26, 38] на рівні плати та кристала привели до появи механізму асерцій для тестування та верифікації програмних продуктів [32–38].
2. Метрику аналізу тестопридатності [22,23] (спостережуваності та керованості) цифрових структур можна адаптувати для оцінки програмного коду з метою визначення критичних місць і подальшого покращення програмного продукту щодо зменшення часу верифікації та підвищення його якості.
3. Технології аналізу якості покриття [38] наперед заданих несправностей тестовими послідовностями використовуються для створення таблиці покриття функціональностей програмних продуктів модулем testbench з метою оцінки достовірності тестів, встановлення діагнозу та виправлення помилок.
4. Графові моделі регістрових передач Тетга-Абрахама [35], С.Г. Шаршунова [36-37] використовуються при тестуванні програмних продуктів, які, шляхом їх структурно-логічного аналізу, приводяться до більш технологічної форми. Такою є транзакційний граф як модель HDL-коду, що записується у вигляді алгебраїчної форми представлення графа для підрахунку тестопридатності з метою визначення критичних компонентів (точок) програми для встановлення асерцій.
5. Поділ автомата на керувальну [22,27] та операційну частини застосовується для спрощення процесу синтезу testbench та верифікації програмного коду на основі синтезу графів керування та транзакційної передачі даних.
6. Крива життєвого циклу апаратного виробу [28,29] також адекватно відображає тимчасові стадії зміни yield при створенні, тиражуванні й супроводженні програмного продукту.
7. Платформено-орієнтований синтез [21] HDL-коду (platform-based electronic system-level design) з використанням існуючих наборів компонентів (chipset) під керуванням GUI ізоморфний технології об'єктно-орієнтованого програмування на основі використання напрацьованих провідними компаніями бібліотек. Застосування технології Electronic System Level (ESL) [21,31] у програмуванні дає можливість використовувати програмні компоненти готових функціональностей з базових бібліотек, що використовуються для створення нових програмних продуктів. У цьому випадку основна процедура проектування полягає у виконанні мепінга, орієнтованого на покриття функцій специфікації існуючими компонентами, де новий код становить не більше 10% проекту.
8. Поняття тесту перевірки несправностей або функціональностей, що використовується для тестування апаратних продуктів, застосовується як testbench [32-38]

для верифікації та налагодження програмних продуктів, представлених описами на рівні системних мов (C++, SystemVerilog, Vera, e).

9. Платформено-орієнтований синтез [21] testbench (platform-based test-bench synthesis) з використанням існуючих бібліотек тестів (ALINT) для reusable стандартизованих функціональних компонентів використовується при генеруванні тестів програмних модулів на основі напрацьованих бібліотек від провідних компаній планети.

10. Стандартні рішення сервісного обслуговування функціональностей F-IP в рамках інфраструктури I-IP [24-28] використовуються для створення інфраструктури вбудованої верифікації компонентів програмного продукту в цілях усунення помилок дефектного програмного модуля.

11. Забезпечення двовимірності структури взаємопов'язаних функціональних компонентів (IP-cores) створюваного програмного продукту орієнтоване на використання мультіядерних архітектур для технологічного розпаралелювання обчислювальних процесів тестування та верифікації [21,24,33,37], що істотно зменшує параметр time-to-market.

12. Створення адресного простору для функціональностей SoC, реалізованих як в апаратному, так і у програмному виконанні [29-31], надає цифровій системі чудову властивість самовідновлення працездатності програмних і апаратних компонентів шляхом використання альтернативних або надлишкових ресурсів інфраструктури сервісного обслуговування I-IP. Прикладом тому може служити мультипроцесорне виконання апаратних продуктів, яке є стійким до виникаючих дефектів. При цьому адресований компонент, що відмовив, може бути замінений резервним у процесі виконання функціональності. Властивість адресовуваності використовується при створенні критичних програмних продуктів, де наявність адресовуваних диверсних (мультиверсних) резервних компонентів забезпечує відмовостійкість системи при виникненні дефектів.

13. Цікавою і ринково-привабливою є проблема автономного внутрішньокристалного вбудованого тестування, діагностування та ремонту без застосування зовнішніх засобів [29-31], якою займаються всі провідні компанії. До вирішення проблеми залучаються сучасні бездротові й Internet технології дистанційного сервісного обслуговування. Зворотний бік медалі полягає у несанкціонованому доступі до вмісту кристала на відстані, що може призвести до небажаних деструктивних наслідків і виведення з ладу цифрового виробу. Однак, специфіка сучасних цифрових систем на кристалах полягає у надзвичайній можливості виправляти помилки на відстані, завдяки наявності зв'язку кристала з зовнішнім світом за допомогою Internet або бездротових технологій (wi-fi, wi-max, bluetooth, satellite), присутніх у кремнії. Дистанційна корекція програмних помилок стала можливою завдяки використанню пам'яті (що займає до 94% силікону) SoC для зберігання програм, куди можна, у разі виявлення некоректності, записати новий код, який не має помилок. Дистанційна корекція апаратних помилок стала можливою завдяки використанню програмовних логічних інтегральних схем, куди можна, у разі виявлення несправності, записати новий bit stream, що не має помилок, який фактично створює нову апаратуру шляхом повторного програмування кристала.

Зближення і взаємопроникнення технологій приводить до ізоморфних методів проектування, тестування та верифікації по відношенню до програмних і апаратних комплексів, що по суті є закономірним процесом асиміляції прогресивних концепцій та рішень. Цьому сприяє факт, що найбільш важливі параметри життєвого циклу виробу, такі як time-to-market та yield, стають порівнянними за часом і виходом придатної продукції. Крива життєвого циклу апаратного виробу, представлена на рис. 1.13, з точністю до ізоморфізму відображає часові етапи програмного продукту, який проходить аналогічні стадії: проектування, збільшення обсягу випуску продукції, виробництво з доопрацюванням і супроводженням виробу. У контексті життєвого циклу існують дві актуальні проблеми, які пов'язані з підняттям графіка (кривої) вгору по осі ординат, а також з компресією згаданої кривої по осі часу, що означає зменшення параметра time-to-market. Тут підвищення виходу придатної продукції відбувається на всіх стадіях: 1) проектування – за рахунок швидкого усунення помилок шляхом впровадження нових технологій; 2) прототипування – за рахунок виправлення коду, імplementованого в пам'ять системи на кристалі; 3) виробництво – за рахунок випуску програм, які коригують помилки за допомогою Internet або супутників.

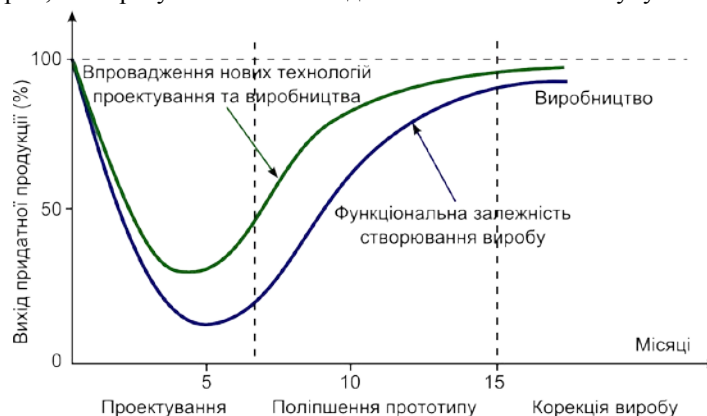


Рис. 1.13. Графік життєвого циклу програмно-апаратного комплексу

Згідно з [32-38] вартість верифікації програмно-апаратних продуктів на основі ASIC (Application-Specific Integrated Circuit – спеціалізована інтегральна мікросхема), IP-core (Intellectual Property Core – налагоджений функціональний апаратний модуль, готовий до імplementації в кристал), SoC складає 70% від загальних витрат проектування. Аналогічна оцінка, близько 80%, визначає розмірність testbench-коду від загальної довжини формального опису проекту. Задачі, вирішувані в процесі верифікації, пов'язані з усуненням помилок проектування на найбільш ранній його стадії з метою приведення коду прототипу у відповідність з його специфікацією. Пропуск помилки збільшує її вартість на порядок при переході від рівня проектування блоку до рівня кристала і далі до системи. Введення в проект програмної надлишковості – механізму асерцій [35-37] – дозволяє виконувати аналіз основних специфікованих умов у процесі моделювання

проекту і діагностувати помилки в разі їх виявлення на ранніх стадіях проектування програми (C++, System Verilog, Vera) або апаратури (HDL).

1.9. Інфраструктура процесу верифікації та тестування проекту

Модель процесу верифікації проекту на системному рівні можна представити у вигляді узагальненого рівняння діагнозу $T \oplus S = L$ або більш докладно для виявлення помилок у програмних компонентах:

$$(T, F) \oplus (S, A) = L_s. \quad (1.1)$$

Тут: T, F – тестові впливи та функціональне покриття еталонної моделі з очікуваними реакціями; S, A – HDL-модель, що перевіряється, і механізм асерцій для верифікації і точного діагностування помилок у програмному коді. Тестування апаратної реалізації базується на використанні логічної операції суми по mod 2 (Xor) для аналітичного виразу $(T) \oplus (S, B) = L_h$, де B – надлишковість у вигляді регістра граничного сканування від стандарту IEEE 1500, що використовується як доповнення до функціональної моделі для забезпечення необхідної глибини діагностування компонентів цифрової системи на кристалі. При цьому L_s, L_h – списки помилок, одержувані на стадіях верифікації проекту і тестування готового цифрового виробу.

Для розуміння співвідношень між ключовими поняттями: верифікація, валідація, асерція – вводяться такі визначення [33-35]. Верифікація є процесом аналізу системи або її компонентів для визначення коректності формальних перетворень вхідного опису у вихідний на кожній стадії проектування. Валідація є процесом визначення працездатності системи і її компонентів шляхом перевірки її відповідності основним вимогам специфікації після виконання кожної стадії проектування. Сертифікація є експертною (юридичною) гарантією валідності системи (компонентів) вимогам специфікації при її використанні за призначенням.

Асерція – це інверсне HDL-висловлювання системного рівня, призначене для раннього визначення помилок проектування відносно вимог специфікації при моделюванні проекту на тестових впливах до і після виконання синтезу. Типи і властивості асерцій: 1) Гранична кількість асерцій – це диверсна модель проекту, на практиці кількість асерційних операторів складає 5% коду проекту. 2) Миттєві або однократні (immediate) асерції належать моделі проекту, вставляються в текст програми у формі інверсних if-операторів і виконуються в загальному порядку проходження рядків коду. 3) Паралельні асерції (циклічні або багатотактні) – моделюються паралельно і незалежно від виконуваної моделі проекту. Обчислювальний процес організовується у цикли моделювання, що дозволяє аналізувати стан проекту в моменти часу, визначені синхроімпульсами введення асерцій. Assert оператори розташовуються у тілі програми. Паралельна асерція захищена від змагань сигналів у моделі наявністю таймслота симуляції. Він задає моменти зчитування (preponed) сигналів і обчислення (postponed) асерції. 4) Зовнішні асерції являють собою диверсну модель проекту у вигляді окремого модуля, записаного мовою System Verilog або PSL. Зовнішні асерції спостерігають за сигналами портів функціонального модуля і передають їх з

проекту в тестове середовище. 5) Асерції, в загальному випадку, не є синтезованою підмножиною HDL-мов. Більше того, механізм асерцій, функціональні покриття і генератори тестів, спрямовані на виявлення помилок проектування, формують HVL-мовне середовище. 6) Асерції орієнтовані на раннє виявлення функціональних помилок всіх стадій перетворення моделі, у тому числі й післясинтезних помилок, без виконання процедури синтезу.

Формально асерція записується у вигляді предикату $A_i = T_i \oplus S_i \rightarrow d(A_i)$, який на множині станів еталонної і реальної моделей приймає значення $\{\text{true} = 1, \text{false} = 0, \text{don't care} = X\}$. Тут X – невизначений стан асерції або її відсутність для аналізованого програмного компонента. Відповідно до згаданого визначення і за функціональною реакцією $d(A_i)$ на порівняння станів $T_i \oplus S_i$ асерції можна

диференціювати на три типи $d(A_i) = \{A^c, A^g, A^w\}$ – припинення верифікації HDL-моделі, перехід до перевірки наступного програмного блоку, видача повідомлення на консоль. Механізм асерцій – це система висловлювань, представлена у вигляді вектора $A = (A_1, A_2, \dots, A_i, \dots, A_n)$, а також засоби їх аналізу $d(A)$, призначені для верифікації та пошуку помилок HDL-моделі системного рівня у просторі і часі. Мета введення механізму асерцій – отримання точного діагнозу для усунення семантичних помилок HDL-коду програми. Вектор асерцій орієнтований на перевірку функціональностей і логічного стану компонентів HDL-моделі у часі і просторі. Такі ж функції – підвищення глибини діагностування – присутні також у стандартах граничного сканування (IEEE 11.49, IEEE 1500), де тестування компонентів цифрової системи регулюється контролером тестового доступу (TAP).

Структурна модель процесу проектування в частині взаємодії процедур валідації та верифікації представлена на рис. 1.14. Тут слід зазначити, що верифікація визначає коректність виконаних перетворень на розглянутому етапі проектування, а валідація визначає відповідність технічного стану системи вимогам специфікації на кожній стадії проектування (Spec – специфікація, SL – System Level – системний рівень, RTL – Register Transfer Level – рівень регістрових передач, Proto – Prototype – прототип).

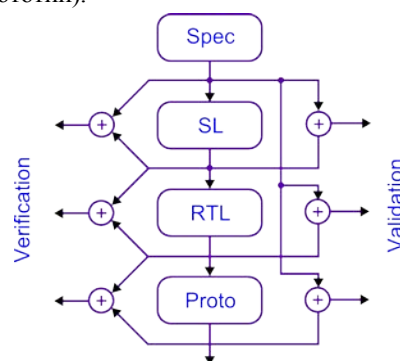


Рис. 1.14. Взаємодія процесів валідації та верифікації

```

graph LR
    A[Специфікація] -- "Проектування" --> B["HDL Модель (Netlist)"]
    B -- "Імплементація" --> C[Силіконовий кристал]
    C -- "Тестування" --> B
    B -- "Верифікація" --> A
  
```

Інфраструктура тестування та верифікації HDL-моделі зображена на рис. 1.16. де специфікація проекту, описана формальною мовою високого рівня, є вихідною інформацією для: створення метрики оцінювання якості тесту у вигляді функціонального покриття; HDL-моделі проекту; тесту з еталонними реакціями – testbench, асерційної структури, яка служить доповненням до основної моделі, що необхідно для прискорення перевірки і налагодження проекту.

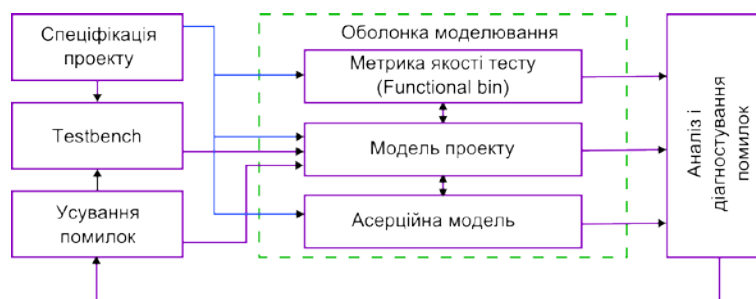


Рис. 1.16. Середовище верифікації проекту

Середовище верифікації представлено системою моделювання, блоком тестування (testbench), механізмом асерцій (Assertion Engine) і власне системним кодом моделі мовами VHDL, Verilog, System Verilog. Модуль testbench задає вхідні стимули й еталонні реакції на них, записані HDL-мовами і орієнтовані на перевірку функціональності (змінні, функції, послідовності), параметри яких визначаються у кошику функціонального покриття. Механізм асерцій – модельна надлишковість, яка доповнює testbench, в частині перевірки внутрішніх за часом і простором станів проекту, представлена вхід-вихідними висловлюваннями і призначена для прискорення тестування, верифікації, діагностування та виправлення помилок проектування у системному коді. Асерції можна згенерувати не тільки за специфікацією, але і за HDL-моделлю, усуваючи непотрібні конструкції, а решту – модифікувати до форми асерцій. При цьому існує ймовірність повторення в асерції програмної помилки HDL-моделі, яка не буде ідентифікована в процесі моделювання.

Спрощена структура взаємодії механізму асерцій з іншими компонентами середовища верифікації та діагностування показана на рис. 1.17. Testbench являє собою еталонну модель об'єкта проектування у вигляді вхідних та вихідних реакцій. Часто замість testbench використовується вже перевірена і доступна модель іншої компанії, відносно якої перевіряється модель (MUV - Model Under Verification), що верифікується. У цьому випадку необхідно мати генератор вхідних послідовностей або testbench без вихідних реакцій. Блок механізму асерцій є надбудовою для testbench і моделі, що верифікується, і служить для порівняння результатів моделювання в цілях формування двійкового вектора експериментальної перевірки, у ролі якого тут виступає вектор (станів) асерцій:

$$A = (A_1, A_2, \dots, A_i, \dots, A_n), A_i = T_i \oplus S_i, A_i = \{0, 1, X\}.$$

Хоча асерційні оператори можуть бути присутніми в тілі програми testbench або MUV, вони не є приналежністю згаданих програм. Assert-оператори обробляються окремо і паралельно з моделюванням MUV, тому вони не впливають на функціонування HDL-моделі проекту. Крім того, асерції не синтезуються і закінчують своє існування після налагодження системного коду. Маючи аналоги операторів у HDL-мовах у вигляді умовних команд, assert-оператори призначені для порівняння технічних станів компонентів еталонної та проектової моделей з метою подальшого прийняття рішення про продовження процесу моделювання, припинення або переходу до виконання аналізу іншого фрагменту програмного коду.

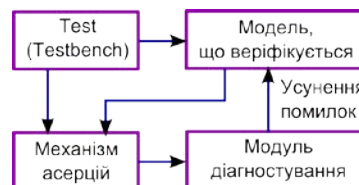


Рис. 1.17. Технологія використання асерцій для верифікації проекту

Інтерес викликає і послідовність дій по створенню середовища верифікації, де єдиним аргументом є специфікація проекту, все інше – похідні від неї.

На рис. 1.18 зображено структуру взаємозв'язків процесу проектування і діагностування для виправлення помилок у HDL-кодї програми.

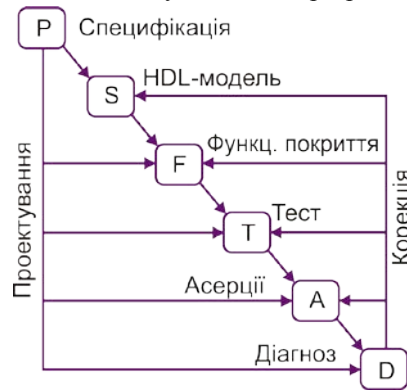


Рис. 1.18. Маршрут верифікації проекту

Практично, якщо виконані умови тестопридатності та правильно розставлені асерції у критичних точках програмного коду для діагностування всіх компонентів, то процедура спільного моделювання механізму асерцій та HDL-моделі може однозначно ідентифікувати послідовність рядків програмного коду з семантичною помилкою. Testbench і вектор (матриця) асерцій повинні створюватися незалежно від HDL-моделі та іншим розробником. Це забезпечує диверсифікацію згаданих моделей щодо єдиної специфікації, а також виявлення та виправлення помилок у HDL-проекті. Таким чином, в процесі створення системного коду проекту розробник повинен орієнтуватися на технологічні та прості процедури процесу верифікації HDL-моделі, що означає – виконувати досить прості умови тестопридатності (див. рис. 1.17, 1.18).

Тут необхідно вирішувати задачі: 1) Написання testbench та генерування функціональних покриттів. 2) Визначення критичних точок програмного коду для установки асерцій. 3) Коректне написання самих асерцій. 4) Створення ефективних алгоритмів діагностування семантичних помилок у HDL-кодї на основі аналізу системи асерцій, як реакції на тест еталонної і реальної моделей в процесі моделювання проекту.

1.10. Аналітична модель інфраструктури верифікації

Для ідентифікації узагальненого стану HDL-моделі формуються еталонні реакції (сигнатури) критичних точок (змінні, регістри, пам'ять) у часі і просторі. Потім виконується аналіз HDL-моделі в цілях її діагностування шляхом порівняння еталонних та експериментальних реакцій (сигнатур) для формування асерційного вектора, що описує порівняння станів (еталонного і реального) компонентів об'єкта в часі і просторі. Доцільно формувати асерційний вектор незалежно від HDL-моделі проекту. Асерційна і функціональна моделі обробляються паралельно і незалежно одна від одної середовищем моделювання. Асерційна модель визначає відхилення від поведінки об'єкта в істотних точках просторово-часової еталонної структури. Формат асерцій повинен відповідати формату HDL-моделі

проекту. Асерції орієнтовані на діагностування семантичних помилок HDL-моделі шляхом використання testbench і MUV.

Аналітична модель інфраструктури верифікації подана у такому вигляді (P – специфікація проекту, S – soft-модель, A – асерційна модель, T – testbench, F – кошик покриття функціональностей, d – модуль діагностування помилок і C – умови діагностування помилок):

$$M = \{P, S, A, T, F, d, C\},$$

$$1) S = f_1(P) = (S_1, S_2, \dots, S_i, \dots, S_n);$$

$$2) F = f_2(P, S) = \{F_1, F_2, \dots, F_i, \dots, F_n\};$$

$$3) T = f_3(P, S, F) = \{T_1, T_2, \dots, T_i, \dots, T_n\};$$

$$4) A = f_4(P, S, F, T) = (A_1, A_2, \dots, A_i, \dots, A_n);$$

$$5) d = f_5(P, S, F, T, A) = (L_1, L_2, \dots, L_i, \dots, L_n) \in \{L_s, L_h\};$$

$$6) C = [\bigcup_{i=1}^n F_i \in F = P] \wedge [\bigcup_{i=1}^n T_i \in T = F]; \quad (1.2)$$

$$7) L_s = (T, F) \oplus (S, A); 8) L_h = (T) \oplus (S, B).$$

Формула 6 у (1.2) визначає умови повноти перевірки функціональностей тестом (testbench) відносно специфікації. Рівняння 7 задає функцію обчислення помилок проектування при переході від системного до регістрового рівня шляхом використання атрибутів верифікаційної інфраструктури.

Асерційна надлишковість є функцією від критичних точок HDL-моделі, гранична кількість яких може дорівнювати кількості часових фреймів функціональних компонентів, визначених специфікацією.

Апріорно координатам вектора асерцій присвоюються значення X. Потім визначаються критичні координати, кількість яких буде достатньою для проведення верифікаційного експерименту з метою пошуку помилкових програмних блоків із заданою глибиною діагностування. Такі координати ідентифікуються одиницями. У процесі моделювання координати вектора модифікуються у бік зменшення одиниць. Кожній координаті вектора A ставиться у відповідність список всіх вершин попередників транзакційного графа програмного коду. Координатам вектора відповідає матриця досяжності транзакційного графа або списки вершин-попередників, задані у будь-якій іншій формі. За фактичним бінарним станом елементів вектора A виконується безумовна процедура діагностування списку L несправних програмних блоків d(A), що визначається такими виразами [38]:

$$\begin{cases} L_s(A) = (\bigcap_{\forall i(A_i = 1)} A_i) \setminus (\bigcup_{\forall i(A_i = 0)} A_i); \\ L_m(A) = (\bigcup_{\forall i(A_i = 1)} A_i) \setminus (\bigcup_{\forall i(A_i = 0)} A_i). \end{cases} \quad (1.3)$$

Рівняння призначені для пошуку поодиноких і кратних помилок шляхом використання вектора асерційних станів. Довжина асерційного вектора дорівнює кількості вершин у графі або програмних блоків у функціонально-логічній структурі HDL-коду. Векторна модель середовища верифікації:

$$\begin{bmatrix} T_1 \\ T_2 \\ T_i \\ T_n \end{bmatrix} \oplus \begin{bmatrix} S_1 \\ S_2 \\ S_i \\ S_n \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_i \\ A_n \end{bmatrix} \xrightarrow{d} \begin{bmatrix} L_1 \\ L_2 \\ L_i \\ L_n \end{bmatrix} \quad (1.4)$$

У процесі моделювання здійснюється порівняння реакцій testbench та HDL-моделі, що формує стани координат асерційного вектора:

$$A_i = f(T_i, S_i) = T_i \oplus S_i, A_i = \{0, 1, X\}.$$

Потім істотні (0,1)-координати вектора асерцій маскують матрицю досяжності для отримання списку програмних блоків з помилками шляхом виконання однієї з процедур, визначених у (1.3).

1.10.1. Діагностування за матрицею досяжності графа. Демонстрація технології діагностування несправних блоків представлена таким прикладом. Припустимо є функціонально-логічний граф HDL-моделі, зображений на рис. 1.19.

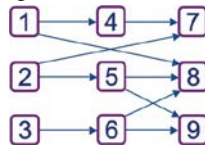


Рис. 1.19. Функціонально-логічний граф HDL-моделі

Структура взаємозв'язків графа представлена матрицею досяжності:

M	1	2	3	4	5	6	7	8	9
1	1								
2		1							
3			1						
4	1			1					
5		1			1				
6			1			1			
7	1	1		1			1		
8	1	1	1		1	1		1	
9		1	1		1	1			1

Векторна модель діагностування, описана виразом (1.4), містить списки блоків-попередників для кожної вершини графа, одержані з матриці досяжності. Кожній вершині графа ставиться у відповідність асерція, яка в процесі моделювання може бути довізначена значенням $\{0, 1\}$. Векторна модель пошуку помилкових програмних блоків (1.4) для графа (рис. 1.19) трансформується до вигляду:

T ₁	⊕	S ₁	=	A ₁ = X	d →	L ₁ = S ₁
T ₂		S ₂		A ₂ = X		L ₂ = S ₂
T ₃		S ₃		A ₃ = X		L ₃ = S ₃
T ₄		S ₄		A ₄ = 0		L ₄ = S ₁ , S ₄
T ₅		S ₅		A ₅ = X		L ₅ = S ₂ , S ₅
T ₆		S ₆		A ₆ = X		L ₆ = S ₃ , S ₆
T ₇		S ₇		A ₇ = 1		L ₇ = S ₁ , S ₂ , S ₄ , S ₇
T ₈		S ₈		A ₈ = 1		L ₈ = S ₁ , S ₂ , S ₃ , S ₅ , S ₆ , S ₈
T ₉		S ₉		A ₉ = X		L ₉ = S ₂ , S ₃ , S ₅ , S ₆ , S ₉

В результаті діагностування за системою рівнянь (1.3), що полягає в перетині всіх несправних компонентів, відповідних одиничним координатам вектора асерцій, з наступним відніманням об'єднання всіх несправних модулів, відповідних нульовим координатам вектора A, виходить список дефектних програмних блоків (за умови існування тільки одного помилкового модуля):

$$L_s(A_4 = 0, A_7 = 1, A_8 = 1) = L_7 \cap L_8 \setminus L_4 = \\ = S_1, S_2, S_4, S_7 \cap S_1, S_2, S_3, S_5, S_6, S_8 \setminus S_1, S_4 = S_2.$$

При використанні другого рівняння з виразу (1.3) можна отримати список всіх програмних блоків, які можуть мати помилки, за умови існування кількох дефектних компонентів:

$$L_m(A_4 = 0, A_7 = 1, A_8 = 1) = L_7 \cup L_8 \setminus L_4 = \\ = S_1, S_2, S_4, S_7 \cup S_1, S_2, S_3, S_5, S_6, S_8 \setminus S_1, S_4 = S_2, S_3, S_5, S_6, S_7, S_8.$$

Для ілюстрації наступного прикладу діагностичного експерименту можна виключити з розгляду перші два вектори, які не є суттєвими для процесу обробки списків несправних блоків на основі аналізу координат асерційного вектора $L = d(A, S)$. Процедура обчислення списку одиночних дефектів має вигляд:

A ₁ = X	d →	L ₁ = S ₁	(L ₈ ∩ L ₉) – – (L ₅ ∪ L ₇)	S ₃ , S ₆
A ₂ = X		L ₂ = S ₂		
A ₃ = X		L ₃ = S ₃		
A ₄ = X		L ₄ = S ₁ , S ₄		
A ₅ = 0		L ₅ = S ₂ , S ₅		
A ₆ = X		L ₆ = S ₃ , S ₆		
A ₇ = 0		L ₇ = S ₁ , S ₂ , S ₄ , S ₇		
A ₈ = 1		L ₈ = S ₁ , S ₂ , S ₃ , S ₅ , S ₆ , S ₈		
A ₉ = 1		L ₉ = S ₂ , S ₃ , S ₅ , S ₆ , S ₉		

Для множинних несправних блоків на одному й тому ж векторі асерцій результат містить більшу кількість дефектних модулів:

$A_1 = X$	d →	$L_1 = S_1$	$(L_8 \cup L_9) -$ $-(L_5 \cup L_7)$	$S_3, S_6,$ S_8, S_9
$A_2 = X$		$L_2 = S_2$		
$A_3 = X$		$L_3 = S_3$		
$A_4 = X$		$L_4 = S_1, S_4$		
$A_5 = 0$		$L_5 = S_2, S_5$		
$A_6 = X$		$L_6 = S_3, S_6$		
$A_7 = 0$		$L_7 = S_1, S_2, S_4, S_7$		
$A_8 = 1$		$L_8 = S_1, S_2, S_3, S_5, S_6, S_8$		
$A_9 = 1$		$L_9 = S_2, S_3, S_5, S_6, S_9$		

1.10.2. Діагностування за векторно-логічною формою графа. Цікавою є векторна форма моделі проведення діагностичного експерименту. Нижче наведено матрицю досяжності, вбудовану в модель діагностування, де асерції зображено у вигляді трійкового вектора:

A	L	1	2	3	4	5	6	7	8	9	$L_s = (L_8 \wedge L_9) \wedge$ $\wedge (\overline{L_5 \vee L_7}) =$ $= (001001000) =$ $= S_3, S_6$
X	L_1	1									
X	L_2		1								
X	L_3			1							
X	L_4	1			1						
0	L_5		1			1					
X	L_6			1			1				
0	L_7	1	1		1			1			
1	L_8	1	1	1		1	1		1		
1	L_9		1	1		1	1			1	
	L_s			1			1				

Результат пошуку програмних блоків отриманий за умови існування в проєкті одного несправного модуля, записаного в останньому рядку. Процедура діагностування на основі асерційного вектора A шляхом модифікації (1.3) до виконання векторних операцій кон'юнкції, диз'юнкції і заперечення

$$\begin{cases} L_s(A) = \left(\bigwedge_{\forall i(A_i = 1)} A_i \right) \wedge \left(\bigvee_{\forall i(A_i = 0)} \overline{A_i} \right); \\ L_m(A) = \left(\bigvee_{\forall i(A_i = 1)} A_i \right) \wedge \left(\bigvee_{\forall i(A_i = 0)} \overline{A_i} \right) \end{cases}$$

визначена у правій частині матричної моделі діагностування. Аналогічні обчислення для випадку існування кратних дефектів у програмних модулях проєкту дають такий результат:

A	L	1	2	3	4	5	6	7	8	9
X	L ₁	1								
X	L ₂		1							
X	L ₃			1						
X	L ₄	1			1					
0	L ₅		1			1				
X	L ₆			1			1			
0	L ₇	1	1		1			1		
1	L ₈	1	1	1		1	1		1	
1	L ₉		1	1		1	1			1
	L _m			1			1		1	1

$$\begin{aligned}
 L_m &= (L_8 \vee L_9) \wedge \\
 &\wedge \overline{(L_5 \vee L_7)} = \\
 &= (001001011) = \\
 &= S_3, S_6, S_8, S_9
 \end{aligned}$$

У реальності результат діагностування гарантує наявність хоча б одного дефектного блоку зі списку компонентів, підозрюваних у наявності несправностей, визначених у [38].

1.10.3. Діагностування за алгебологічною формою графа. Пропонується процедура діагностування HDL-моделі за структурно-логічним (транзакційним) графом, представленим алгебраїчною формою опису графових структур [38]. Її переваги полягають у компактності задання матриці досяжності, яка до того ж має властивість структуризації графа у формі визначення всіх шляхів, навантажених на кожну вершину. Крім того, поєднання дефектних компонентів, що визначаються кон'юнктивними термами, дають більш точний результат діагностування, у порівнянні з формуванням несправних модулів у вигляді неупорядкованої множини елементів.

Для структури, зображеної на рис. 1.19, алгебраїчна форма графа та обчислення списку поодиноких несправностей мають такий вигляд:

A	L
X	$L_1 = S_1$
X	$L_2 = S_2$
X	$L_3 = S_3$
X	$L_4 = S_1 S_4$
0	$L_5 = S_2 S_5$
X	$L_6 = S_3 S_6$
0	$L_7 = S_1 S_4 S_7 \vee S_2 S_7$
1	$L_8 = S_1 S_8 \vee S_2 S_5 S_8 \vee S_3 S_6 S_8$
1	$L_9 = S_2 S_5 S_9 \vee S_3 S_6 S_9$
	$L_s = S_3, S_6$

$$\begin{aligned}
 L_s &= (L_8 \wedge L_9) - (L_5 \vee L_7) = \\
 &= (S_2 S_5 \vee S_3 S_6) - S_2 S_5 = \\
 &= S_3 S_6
 \end{aligned}$$

Процедура аналізу логічних функцій графової структури містить три пункти:

1. Всі терми і змінні в ДНФ, яка відповідає нульовому значенню асерційної координати, дорівнюють нулю
2. В інших ДНФ, відповідних одиничному значенню асерційної координати, ліва частина кон'юнктивного терма, включаючи змінну, раніше визначену нулем, видаляється.

3. Для поодиноких функцій здійснюється перетин (об'єднання) термів, що залишилися, якщо виконується діагностування поодиноких (кратних) несправних блоків.

Такі перетворення ДНФ шляхом спеціального моделювання нульових сигналів змінних в одиничних, щодо асерцій, логічних функціях представлення графа в цілях отримання списку несправних програмних модулів наведено в правій частині алгебологічної моделі діагностування.

Наступний приклад також підтверджує можливість здійснення аналізу логічних функцій для обчислення множинних несправних блоків:

A	L	$L_m = (L_8 \vee L_9) - (L_5 \vee L_7) =$ $(S_1S_8 \vee S_2S_5S_8 \vee S_3S_6S_8 \vee$ $S_2S_5S_9 \vee S_3S_6S_9) -$ $(S_2S_5 \vee S_1S_4S_7 \vee S_2S_7) =$ $S_3S_6S_8 \vee S_3S_6S_9$
X	$L_1 = S_1$	
X	$L_2 = S_2$	
X	$L_3 = S_3$	
X	$L_4 = S_1S_4$	
0	$L_5 = S_2S_5$	
X	$L_6 = S_3S_6$	
0	$L_7 = S_1S_4S_7 \vee S_2S_7$	
1	$L_8 = S_1S_8 \vee S_2S_5S_8 \vee S_3S_6S_8$	
1	$L_9 = S_2S_5S_9 \vee S_3S_6S_9$	
	$L_m = S_3, S_6, S_8, S_9$	

Для отриманої функції L_m застосування додаткової асерційної точки $A_6 = S_3S_6$ дозволяє підвищити глибину діагностування до двох несправних блоків:

$$L_m = S_3S_6S_8 \vee S_3S_6S_9 = \begin{cases} S_3S_6 \leftarrow A_6 = 1; \\ S_8 \vee S_9 \leftarrow A_6 = 0. \end{cases}$$

Перевірка асерційної точки $A_6 = 1$ не виключає блок S_8 зі списку несправних, але гарантує факт наявності помилки в компонентах S_3S_6 . Тому після процедури виправлення некоректностей у блоках S_3S_6 необхідно повторювати діагностичний експеримент. Що стосується перевірки асерційної точки $A_6 = 0$, то вона виключає наявність помилок у блоках S_3S_6 . Тому нульова перевірка завжди доставляє більш цінну інформацію для діагностичного експерименту. З метою зменшення підозрюваної множини несправних блоків виконується ще одна перевірка $(A_3 \vee A_9)$, яка встановлює точний діагноз:

$$L_m = \begin{cases} = S_3S_6 = \begin{cases} S_3 \leftarrow A_3 = 1; \\ S_6 \leftarrow A_3 = 0; \end{cases} \\ = S_8 \vee S_9 = \begin{cases} S_8 \leftarrow A_9 = 0; \\ S_9 \leftarrow A_9 = 1. \end{cases} \end{cases}$$

Після усунення помилок діагностичний експеримент повторюється для пошуку інших складових кратних несправностей HDL-блоків, які можуть бути присутніми у програмному коді [10].

Запропонована технологія верифікації є досить простою, орієнтована на обробку HDL-моделей великої розмірності, які містять тисячі строк коду, що описує системні структури мовами VHDL, Verilog.

Технологічний маршрут верифікації програмного коду представлений матрицею виконання послідовно-паралельних процедур:

		P	
1	$F = f_2(P, S)$	$S = f_1(P)$	$T = f_3(P, S, F)$
2		$G = f_4(S, F)$	
3		$U = f_5(G, S, F)$	
4		$A = f_6(G, U, S, F, T)$	
5		$L_s = d(T, F, S, A)$	

які мають такий зміст у вигляді 5 пунктів:

1. Створення HDL-моделі, testbench, функціонального покриття за специфікацією проекту в паралельному режимі.
2. Синтез транзакційного графа (вершини – елементи зберігання інформації та дуги – HDL-оператори, які виконують транзакції між вершинами), що представляє структуру програмного коду за допомогою компонентів та операторів HDL-мови. Оператори циклів розміщуються в одному блоці. Граф не повинен містити глобальних зворотних зв'язків.
3. Визначення тестопридатності графа шляхом підрахунку спостережуваностей всіх вершин для планування верифікаційного діагностичного експерименту.
4. Розміщення асерцій в n% (25%) вершин, які мають мінімальні оцінки спостережуваності. Їх кількість повинна забезпечувати задану глибину діагностування. Асерції повинні бути керованими, що забезпечує їх підключення або відключення в процесі ітеративної верифікації в залежності від результату попередньої перевірки. Така властивість може істотно скоротити час налагодження програмного коду. Зазначений підхід умовного (асерційного) зондового діагностування широко використовується при тестуванні складних апаратних цифрових виробів.
5. Діагностування і виправлення помилок шляхом спільного моделювання HDL-коду, асерцій на тестових послідовностях testbench за умов повноти, визначених функціональним покриттям. Після усунення помилок у дефектному блоці процедура моделювання та діагностування повторюється.

1.11. Аналіз тестопридатності HDL-проектів

Досить суттєва надлишковість HDL-моделі передбачає її ефективне використання в цілях підвищення тестопридатності структури розробленого або написаного коду. Існуючі стандарти тестопридатного проектування апаратури (IEEE 1500, 11.49) можна адаптувати до верифікації HDL-коду системних і регістро-

вих програмних моделей. Для цього використовується граф регістрових або транзакційних передач С.Г. Шаршунова [30], що надає користувачеві інформацію про взаємозв'язки булевих і регістрових змінних, пам'яті та інтерфейсних шин, які називають транзакторами. Дані для синтезу графа отримують шляхом синтаксичного аналізу рядків HDL-коду на предмет встановлення джерел і приймачів інформації, які є вершинами (транзакторами) графа, з'єднаними орієнтованими дугами. Кожну з дуг можна позначити кількістю навантажених на неї операторів. Побудований таким чином транзакційний граф (TG – Transaction Graph) [38] покриває всі функціональності (транзакції) програмної моделі і задає зв'язки між вершинами, які відповідають прийому, передачі та перетворенню інформації.

Роль транзакційного графа полягає у створенні моделі передачі даних в цілях визначення тестопридатності всіх вершин. Потім виділяється підмножина вершин, які мають мінімальне значення тестопридатності, що задовольняє умові:

$$V = \forall i [Q(V_i) \leq Q_{\min}] \rightarrow Q^t \geq Q_{\min}^t, \text{ де } Q^t - \text{якість покриття тестом}$$

(testbench) несправностей (кошика функціональних покриттів) при модифікації структури цифрового проекту або HDL-коду шляхом додаткового моніторингу станів критичних вершин, для яких апаратна (програмна) надлишковість у реальних проектах становить близько 5% (Yervant Zorian).

Основними критеріями ефективності проектування виробу на ринку EDA є вихід придатної продукції ($Y - \text{Yield}$) та відносний час створення продукту $T^{\Delta} - \text{time-to-market}$. Разом з відносними апаратними витратами проекту Z^{Δ} вони формують оцінку E ефективності проектування цифрового виробу, представлену в (1.6), як зведений до загальних часових витрат час, витрачений на створення програмної й апаратної функціональності $T^{\Delta}(S), T^{\Delta}(H)$, помножений на аналогічно зведені програмні й апаратні витрати $Z^{\Delta}(S) \times Z^{\Delta}(H)$, а також на вихід придатної продукції Y . Останній параметр $Y = (1 - P)^{n(1-Q)}$ залежить від тестопридатності (якості) проекту Q , ймовірності P існування в кристалі несправних областей і кількості виявлених дефектів n або D . Критерій часових витрат T^{Δ} також визначається тестопридатністю проекту Q та розмірністю його програмної $Z(S)$, апаратної моделі $Z(H)$, зведеної до денної або годинної продуктивності $Z^1(S) [Z^1(H)]$ розробника коду (апаратури). Коефіцієнти k_q, k_w задають частини часового інтервалу, необхідного для написання HDL-коду – $k_w = 0,3$ і верифікації – $k_q = 0,7$ проекту, $k_q + k_w = 1$. Параметри $T^+(S), T^+(H)$ визначають час створення програмної та апаратної функціональності, а $T^-(S), T^-(H)$ – витрати для їх сервісного обслуговування, до яких входять:

$$T(F^S), T(T^S), T(A), T(G^S) [T(F^S), T(T^S), T(A), T(G^S)]$$

– додатковий період часу на створення функціонального покриття, тестових послідовностей testbench, механізму асерцій [регістра граничного сканування] і транзакційного графа програмної [апаратної] моделі. Розмірність HDL-коду та апаратури для проекту визначається програмними [апаратними] компонентами:

$$Z^{\Sigma}(S) = Z(S), Z(F^S), Z(T^S), Z(A), Z(G^S),$$

$$[Z^{\Sigma}(H) = Z(H), Z(F^h), Z(T^h), Z(B), Z(G^h)],$$

де в правій частині рівності представлено компоненти: функціональність, функціональне покриття, testbench [тест], механізм асерцій [регістр граничного сканування], транзакційний граф програмної [апаратної] моделі. Загальна модель ефективності проектування SoC має вигляд:

$$E = \frac{1}{3} (Y + T^{\Delta} + Z^{\Delta}),$$

$$Y = (1 - P)^{n(1-Q)}; \quad D = 1 - Y^{(1-Q)};$$

$$T^{\Delta} = T^{\Delta}(S) \times T^{\Delta}(H);$$

$$Z^{\Delta} = Z^{\Delta}(S) \times Z^{\Delta}(H);$$

$$T^{\Delta}(S) = \frac{T^{+}(S)}{T^{+}(S) + T^{-}(S)}; \quad T^{\Delta}(H) = \frac{T^{+}(H)}{T^{+}(H) + T^{-}(H)};$$

$$T^{+}(S) = \left(k_w + k_q \frac{1-Q}{1+Q} \right) \frac{Z(S)}{Z^1(S)};$$

$$T^{-}(S) = \frac{1-Q}{1+Q} \left[\frac{Z(F^S)}{Z^1(F^S)} + \frac{Z(T^S)}{Z^1(T^S)} + \frac{Z(A)}{Z^1(A)} \right] + \frac{Z(G^S)}{Z^1(G^S)};$$

$$T^{+}(H) = \left(k_w + k_q \frac{1-Q}{1+Q} \right) \frac{Z(H)}{Z^1(H)};$$

$$T^{-}(H) = \frac{1-Q}{1+Q} \left[\frac{Z(F^h)}{Z^1(F^h)} + \frac{Z(T^h)}{Z^1(T^h)} + \frac{Z(B)}{Z^1(B)} \right] + \frac{Z(G^h)}{Z^1(G^h)};$$

$$T(S) = T^{+}(S) + T^{-}(S); \quad T(H) = T^{+}(H) + T^{-}(H);$$

$$\begin{aligned}
Z^{\Delta}(S) &= \frac{Z(S)}{Z(S) + Z(F^S) + Z(T^S) + Z(A) + Z(G^S)}; \\
Z^{\Delta}(H) &= \frac{Z(H)}{Z(H) + Z(F^h) + Z(T^h) + Z(B) + Z(G^h)}; \\
Z^{\Sigma}(S) &= Z(S) + Z(F^S) + Z(T^S) + Z(A) + Z(G^S); \\
Z^{\Sigma}(H) &= Z(H) + Z(F^h) + Z(T^h) + Z(B) + Z(G^h); \\
Q &= \{Q^h, Q^s\} = \frac{1}{n} \sum_{i=1}^n (U_i \times N_i); \\
U_i &= \frac{1}{\tau} \sum_{j=1}^{x_i} T_j^i \times \frac{1}{d_i^x \vee t_i^x}; \quad N_i = \frac{1}{\tau} \sum_{j=1}^{y_i} T_j^i \times \frac{1}{d_i^y \vee t_i^y}.
\end{aligned} \tag{1.6}$$

Тут (1.6) інтегральна оцінка тестопридатності $Q = \{Q^h, Q^s\}$ транзакційних графів HDL-коду та апаратної моделі регістрового рівня функціонально залежить від керованості та спостережуваності їх вершин U_i, N_i , де n – кількість вершин транзакційного графа. Керованість та спостережуваність є метрикою оцінювання тестопридатності не з'єднувальних ліній, а компонентів HDL-коду, таких як: регістр, лічильник, пам'ять або масиви, вхід-вихідні шини, вектори, логічні або арифметичні змінні цифрового проекту.

Керованість вершини має функціональну залежність від структурної глибини d_i^x знаходження транзактора відносно входів або довжини кон'юнктивного терма – t_i^x . Спостережуваність вершини має аналогічну залежність $d_i^y \vee t_i^y$ відносно виходів. Для підрахунку тестопридатності можна використовувати один з параметрів $d_i^x, t_i^x (d_i^y, t_i^y)$. Оцінки керованості та спостережуваності залежать

також від процентного відношення кількості команд $\frac{1}{\tau} \sum_{j=1}^{x_i} T_j^i$, які мають вхід-

ний (вихідний – $\frac{1}{\tau} \sum_{j=1}^{y_i} T_j^i$) доступ до вершини при аналізі даної програми, до за-

гальної кількості команд τ , де x_i – кількість команд, які формують доступ до вхідної вершини; (y_i) – кількість команд, які визначають вершину як джерело даних. Тестопридатність Q , представлена в (1.6), залежить від керованості (U), спостережуваності (N), а також від вартості реалізації (Z) компонентів: метрика функціонального покриття (F), testbench (B), механізм асерцій (A), функціональність (S). Керованість (спостережуваність) є функцією кількості операторів, що входять у вершину (виходять з вершини) транзакційного графа, а також від структурної глибини розглядуваного елемента – відстані від входів (виходів) або від кількості часових тактів, необхідних для керування (спостереження) компо-

нента у заданому стані на часовій осі. Вплив потужності L^m ліній моніторингу проекту на змінення (збільшення, зменшення) всіх суттєвих параметрів процесу проектування цифрової системи на кристалі визначається таким виразом:

$$[(L^m \uparrow \rightarrow (Z(B) \uparrow, Z(A) \uparrow, T(B) \uparrow, T(A) \uparrow)) \rightarrow [(Y \uparrow, Q \uparrow, D^f \uparrow) \& \\ \& (T^\Delta \downarrow, T^{\text{sim}} \downarrow, T^{\text{t-gen}} \downarrow, T^{\text{diag}} \downarrow, T^{\text{t-hw}} \downarrow, T^{\text{v-sw}} \downarrow, L^{\text{ud-f}} \downarrow)].$$

Вербальне і послідовне пояснення всіх символів, що входять у вираз – збільшення кількості точок спостереження в програмі або апаратурі відбувається за рахунок додавання апаратної надлишковості у вигляді регістра граничного сканування, програмної надлишковості у вигляді механізму асерцій, а також за рахунок додаткового часу створення зазначених компонентів. Це дає можливість істотно збільшити вихід придатної продукції, тестопридатність проекту, глибину діагностування дефектів і помилок. Крім того, істотно зменшується час: виходу виробу на ринок, моделювання несправностей, генерації тестів, діагностування дефектів і помилок, тестування функціональності апаратури та верифікації програмного коду. Також істотно зменшується кількість невиявлених дорогих дефектів і помилок, що впливає на вихід придатної продукції Yield.

Інтегральна нормована середня адитивна оцінка ефективності процесу проектування представлена сумою функціоналів, які формують витратні (f_i^-) і вигашні (f_i^+) функції, що залежать від кількості ліній моніторингу програмного або апаратного проектів:

$$E = \frac{1}{n} \sum_{i=1}^n k_i f_i,$$

$$f(L^m) = \begin{cases} \{f_1^- = Z(B); f_2^- = Z(A); f_3^- = T(B); f_4^- = T(A)\}; \\ \{f_5^+ = Y; f_6^+ = Q; f_7^+ = D^f\}; \\ \{f_8^+ = T^\Delta; f_9^+ = T^{\text{sim}}; f_{10}^+ = T^{\text{t-gen}}; f_{11}^+ = T^{\text{diag}}; \\ f_{12}^+ = T^{\text{t-hw}}; f_{13}^+ = T^{\text{v-sw}}; f_{14}^+ = L^{\text{ud-f}}\}. \end{cases}$$

Тут коефіцієнт k_i визначає вагомозначність функціонала у формуванні інтегральної оцінки, яка використовується для знаходження оптимізованої стратегії проектування цифрового виробу.

Інтегральній оцінці E можна поставити у відповідність графіки залежностей найбільш істотних функціоналів від кількості ліній моніторингу, які представлено на рис. 1.20.

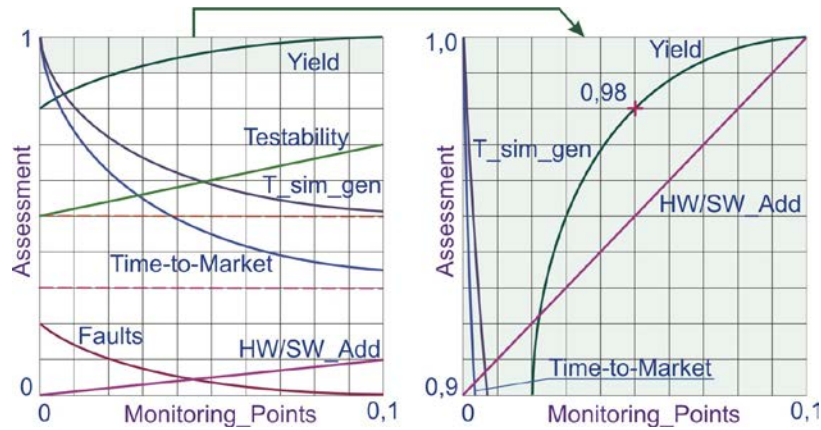


Рис. 1.20. Вплив точок контролю на ефективність проектування

У лівій частині рис. 1.20 зображено залежності ефективностей (Yield – вихід придатної продукції, Testability – тестопридатність проекту, T_{sim_gen} – час моделювання, створення тестів та пошуку дефектів, Time-to-Market – час виходу виробу на ринок, HW / SW_Add – додаткові апаратні і програмні витрати для реалізації IEEE 1500 регістра граничного сканування і механізму асерцій, Faults – кількість неперевіраних дефектів) від кількості ліній спостереження в інтервалі $[0-0,1]$. Права частина рис. 1.20 ілюструє поведінку окремих функціоналів у збільшеному масштабі (10:1) інтервалу $[0,9-1]$ по осі ординат лівого графіка.

Наведений в (1.6) критерій тестопридатності може бути також використаний для оцінення якості граф-схеми керування обчислювальним процесом. Тут розглядаються операторні вершини, навантажені вхідними умовами, а також позиція вершини відносно початку або закінчення схеми керування. Позиція операторної вершини корелюється з часовим тактом керування обчислювальним процесом. Кількість умов виконання сукупності операцій в кожній вершині, об'єднана операцією Or, підвищує тестопридатність графа в частині керованості. Аналогічно обчислюється спостережуваність, на яку впливає структурна глибина і потужність умов, створювана операціями And, Or.

У загальному випадку тестопридатність розглядуваної вершини орієнтованого графа може бути представлена логічної функцією, що задана у вигляді кон'юнктивної нормальної форми (КНФ). При цьому керованість і спостережуваність визначатимуться оцінкою за Квайном обчислювальної складності КНФ.

У загальному випадку логічні функції керованості та спостережуваності поточної вершини (транзакційного) графа задаються кон'юнкцією диз'юнктивних термів – перший рядок (1.7):

$$\begin{aligned}
1) \quad U_r^f &= \bigwedge_{i=1}^{n_r^x} \left(\bigvee_{j=1}^{x_i} T_{rij}^x \right); \quad N_r^f = \bigwedge_{i=1}^{n_r^y} \left(\bigvee_{j=1}^{y_i} T_{rij}^y \right); \\
2) \quad U_r^f &= \bigvee_{i=1}^{x_r} \left(\bigwedge_{j=1}^{n_i^x} T_{ij}^x \right); \quad N_r^f = \bigvee_{i=1}^{y_r} \left(\bigwedge_{j=1}^{n_i^y} T_{ij}^y \right).
\end{aligned}
\tag{1.7}$$

Тут функція керованості U_r^f (спостережуваності N_r^f) визначається кон'юнкцією всіх вершин-попередників n_r^x (наступників n_r^y), де кожна з них має x_i вхідних (y_i вихідних) дуг-транзакцій, з'єднаних знаками диз'юнкції.

Потужність диз'юнктивних термів відповідає кількості вхідних у вершину дуг, а кількість кон'юнкцій являє собою структурну глибину розташування розглядуваного компонента в транзакційному графі. Далі кон'юнктивна форма перетворюється у ДНФ – другий рядок (1.7), де кількість термів для функції керованості (спостережуваності) $x_r(y_r)$ дорівнює всім можливим шляхам формування стану розглядуваної вершини, а довжина терму керованості (спостережуваності) $n_i^x(n_i^y)$ є умовою досяжності вершини – структурна глибина від входів (виходів).

Цікавим видається нестандартне рішення, коли критерії керованості та спостережуваності поточної вершини транзакційного графа обчислюються за допомогою побудованих логічних функцій керованості й спостережуваності (U_i, N_i) та інтегральної оцінки тестопридатності (Q) при використанні апарату – алгебраїчної форми представлення графа [10]. Формули підрахунку зазначених оцінок мають такий вигляд:

$$\begin{aligned}
U_i &= \frac{1}{t_{\max}^x \times n_t^x} \times \sum_{i=1}^{n_t^x} \sum_{j=1}^{k_i^x} (t_{\max}^x - |t_{ij}^x| + 1); \\
N_i &= \frac{1}{t_{\max}^y \times n_t^y} \times \sum_{i=1}^{n_t^y} \sum_{j=1}^{k_i^y} (t_{\max}^y - |t_{ij}^y| + 1); \\
Q &= \frac{1}{n} \sum_{i=1}^n (U_i \times N_i),
\end{aligned}
\tag{1.8}$$

де $t_{\max}^x, n_t^x, k_i^x, |t_{ij}^x|$ – кон'юнктивний терм максимальної довжини для визначення критерію керованості; кількість термів в логічній функції керованості; кількість транзакцій (букв) у поточному термі функції; потужність розглядуваної транзакції в термі. Аналогічні позначення використовуються також при підра-

хунку критерію спостережуваності – $t_{\max}^y, n_t^y, k_1^y, |t_{ij}^y|$ кожної вершини транзакційного графа.

Для фрагмента графа, наведеного на рис. 1.21, перетворення кон'юнктивної форми у диз'юнктивну структуру для вершин $V_1 - V_3$ відповідно до виразу (1.7) формує логічні функції керованості:

$$U^f(V_1) = T_1 \vee T_2 \vee T_3;$$

$$U^f(V_2) = (T_1 \vee T_2 \vee T_3)T_5 \vee T_4 \vee T_6 = T_1T_5 \vee T_2T_5 \vee T_3T_5 \vee T_4 \vee T_6;$$

$$\begin{aligned} U^f(V_3) &= (T_1 \vee T_2 \vee T_3)T_5T_8 \vee (T_4 \vee T_6)T_8 \vee (T_7 \vee T_9) = \\ &= T_1T_5T_8 \vee T_2T_5T_8 \vee T_3T_5T_8 \vee T_4T_8 \vee T_6T_8 \vee T_7 \vee T_9. \end{aligned}$$

Дотримуючись правил (1.8), визначаємо керованість компонента V_3 :

$$U(V_3) = \frac{1}{t_{\max}^x \times n_t^x} \times \sum_{i=1}^{n_t^x} \sum_{j=1}^{k_i^x} (t_{\max}^x - |t_{ij}^x| + 1) = \frac{1}{3 \times 7} \times (1 + 1 + 1 + 2 + 2 + 3 + 3) = 0,61.$$

Для іншого фрагмента графа, представленого на рис. 1.22, перетворення кон'юнктивної форми в диз'юнктивну структуру дозволяє визначити логічну функцію спостережуваності вершини V_1 :

$$N^f(V_1) = T_6 \vee T_7(T_3 \vee T_5 \vee T_4(T_1 \vee T_2)) = T_6 \vee T_7T_3 \vee T_7T_5 \vee T_7T_4T_1 \vee T_7T_4T_2;$$

$$N^f(V_2) = T_3 \vee T_5 \vee T_4(T_1 \vee T_2) = T_3 \vee T_5 \vee T_4T_1 \vee T_4T_2; \quad V_3 = T_1 \vee T_2.$$

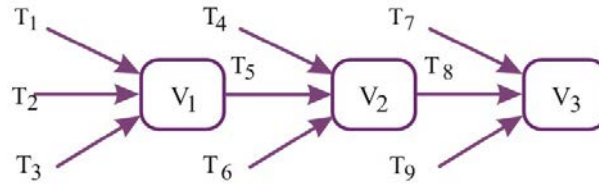


Рис. 1.21. Фрагмент графа для підрахунку керованості

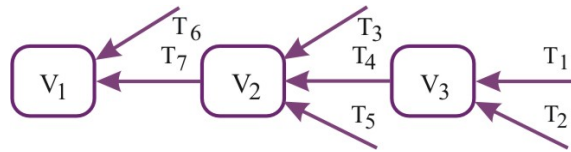


Рис. 1.22. Фрагмент графа для підрахунку спостережуваності

За правилами (1.8) також визначається спостережуваність компонента V_1 :

$$N(V_1) = \frac{1}{t_{\max}^y \times n_t^y} \times \sum_{i=1}^{n_t^y} \sum_{j=1}^{k_i^y} (t_{\max}^y - |t_{ij}^y| + 1) = \frac{1}{3 \times 5} \times (3 + 2 + 2 + 1 + 1) = \frac{9}{15} = 0,6.$$

Для випадку, коли дуги в транзакційному графі мають вагові коефіцієнти b_{ij}^x (b_{ij}^y), що показують кількість операторів, задіяних у передаванні інформації між вершинами (мультидуги), формули підрахунку тестопридатності ускладнюються:

$$U_i = \frac{\sum_{i=1}^{n_t^x} \prod_{j=1}^{k_i^x} b_{ij}^x (t_{\max}^x - |t_{ij}^x| + 1)}{t_{\max}^x \times (\sum_{i=1}^{n_t^x} \prod_{j=1}^{k_i^x} b_{ij}^x)}; N_i = \frac{\sum_{i=1}^{n_t^y} \prod_{j=1}^{k_i^y} b_{ij}^y (t_{\max}^y - |t_{ij}^y| + 1)}{t_{\max}^y \times (\sum_{i=1}^{n_t^y} \prod_{j=1}^{k_i^y} b_{ij}^y)}. \quad (1.9)$$

1.12. Верифікація DCT IP-core, Xilinx

Представлені моделі верифікації програмного HDL-коду перевірено на реальному проекті Xilinx (Open Source) DCT IP-core в цілях визначення наявності в ньому помилок. При цьому отримано позитивний результат щодо невірної семантики роботи програми для подальшого виправлення коду. Фрагмент модуля дискретного косинусного перетворення представлений лістингом 1.1 [Xilinx.com]. Повна HDL-модель налічує 900 рядків коду System Verilog.

Лістинг 1.1.

```
module Xilinx
`timescale 1ns/10ps
module dct ( CLK, RST, xin,dct_2d,rdy_out);
output [11:0] dct_2d;
input CLK, RST;
input[7:0] xin; /* input */
output rdy_out;
wire[11:0] dct_2d;
.....
/* The first 1D-DCT output becomes valid after 14 +64 clk cycles. For the first 2D-DCT output
to be valid it takes 78 + 1clk to write into the ram + 1clk to write out of the ram + 8 clks to shift
in the 1D-DCT values + 1clk to register the 1D-DCT values + 1clk to add/sub + 1clk to take
compliment + 1 clk for multiplying + 2clks to add product. So the 2D-DCT output will be valid
at the 94th clk. rdy_out goes high at 93rd clk so that the first data is valid for the next block*/
endmodule
```

Відповідно до правил тестопридатного аналізу, наведених вище, спроектовано транзакційний граф, наведений на рис. 1.23, який для DCT-module Xilinx містить 28 вершин-компонентів (вхідна й вихідна шини, логічні та регістрові змінні, вектори і пам'ять).

Ідентифікатор дуги має верхній індекс, що позначає кількість транзакцій у програмі між вихідною та вхідною вершинами. Для кожної вершини будуються логічні функції керованості та спостережуваності. Приклад обчислення функції керованості для вершини V_2 має такий вигляд:

$$\begin{aligned}
U^f(B_2) &= T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 (T_5^1 \vee T_6^1 \vee \\
&\vee T_{12}^4 T_{13}^2 T_{14}^2 T_{15}^2 T_{15}^{64} T_{17}^{64} T_{18}^4 T_{19}^4 T_{20}^{64} T_{21}^8 T_{22}^8 T_{23}^8 (T_1^{64} T_7^4 \vee \\
&\vee T_{11}^4 T_2^{64} T_8^8 T_9^8 T_{10}^4 \vee T_{11}^4 T_3^1 \vee T_{11}^4 T_4^1)) = \\
&= T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_5^1 \vee T_6^1 T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 \vee \\
&\vee T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_{12}^4 T_{13}^2 T_{14}^2 T_{15}^2 T_{15}^{64} T_{17}^{64} T_{18}^4 T_{19}^4 T_{20}^{64} T_{21}^8 T_{22}^8 T_{23}^8 T_1^{64} T_7^4 \vee \\
&\vee T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_{12}^4 T_{13}^2 T_{14}^2 T_{15}^2 T_{15}^{64} T_{17}^{64} T_{18}^4 T_{19}^4 T_{20}^{64} T_{21}^8 T_{22}^8 T_{23}^8 T_{11}^4 T_2^{64} T_8^8 T_9^8 T_{10}^4 \vee \\
&\vee T_{11}^4 T_3^1 T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_{12}^4 T_{13}^2 T_{14}^2 T_{15}^2 T_{15}^{64} T_{17}^{64} T_{18}^4 T_{19}^4 T_{20}^{64} T_{21}^8 T_{22}^8 T_{23}^8 \vee \\
&\vee T_{11}^4 T_4^1 T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_{12}^4 T_{13}^2 T_{14}^2 T_{15}^2 T_{15}^{64} T_{17}^{64} T_{18}^4 T_{19}^4 T_{20}^{64} T_{21}^8 T_{22}^8 T_{23}^8.
\end{aligned}$$

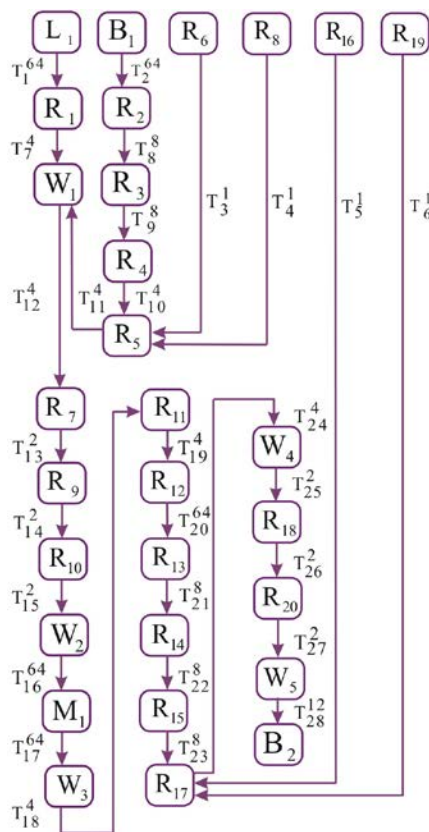


Рис. 1.23. Транзакційний граф Xilinx DCT-моделі

Для вершини L_1 ДНФ функція спостережуваності має вигляд:

$$N^f(L_1) = T_{28}^{12} T_{27}^2 T_{22}^2 T_{25}^2 T_{24}^4 T_{23}^8 T_{22}^8 T_{21}^8 T_{20}^{64} T_{19}^4 T_{18}^4 T_{17}^{64} T_{16}^{64} T_{15}^2 T_{14}^2 T_7^4 T_1^{64}.$$

Синтезовані логічні функції визначають усі можливі шляхи керування як у часі, так і в просторі, що можна вважати новою аналітичною формою опису тестопридатності проекту. За ДНФ, відповідно до виразів (1.8), можна визначити критерій керованості (спостережуваності) для всіх компонентів HDL-моделі.

Тут можна розглядати два варіанти (сценарію) обрахування програмної моделі. 1) Враховується тільки графова структура, де вага кожної дуги дорівнює 1, незалежно від кількості транзакцій у програмному коді. 2) Всі дуги графа відзначаються реальною кількістю транзакцій, що мають місце між двома вершинами транзакційного графа. Оцінки тестопридатності описаних процедур можуть істотно відрізнятися одна від одної. Користувач повинен визначитися, що важливіше – тільки структура програмного коду – застосувати перший сценарій, чи мати більш складну і точну модель транзакцій, розподілених у часі, на множині графових компонентів. Як приклад нижче наведено процедуру обчислення керованості для B_2 :

$$U(B_2) = \frac{1}{22 \times 6} \times (6 + 6 + 19 + 22 + 19 + 19) = 0,54.$$

Застосування аналогічних обчислень керованості (спостережуваності) для інших вершин графа (див. рис. 1.23) дає результат у вигляді графіка, представленого на рис. 1.24, який дозволяє визначити критичні точки для розміщення необхідних асерцій. Такою вершиною може бути компонент R_{15} , якщо транзакційний граф представлений поодинокими дугами. Для випадку, коли дуги відзначені реальною кількістю транзакцій, критичні вершини належать компонентам, які містяться найближче до вихідної шини B_2 . Тут важливою є не структура графа, а вага вхідної дуги, яка має більший негативний вплив, якщо структурна глибина розглянутого компонента досить висока. Використовується формула (1.8) обчислення тестопридатності $U_i \times N_i$, що дає оцінку нижчу, ніж будь-який із співмножників (керованість, спостережуваність).

Після визначення спостережуваності та керованості вершин транзакційного графа здійснюється підрахунок узагальненого критерію тестопридатності програмного коду за формулою (1.8). Для Xilinx DCT-моделі така оцінка дорівнює 0,382. Вона характеризує якість проектного варіанту, що уявляється досить істотним при порівнянні декількох альтернативних рішень. Як приклад позитивного використання розроблених моделей та методів був виконаний аналіз тестопридатності програмного коду дискретного косинусного перетворення (DCT) з Xilinx бібліотеки. Побудовано транзакційну модель, обчислено характеристики тестопридатності та визначено критичні точки (R_1, R_5, R_9, R_{15}). У відповідності з числом і типами компонентів було розроблено функціональне покриття, фрагмент якого представлений лістингом 1.2.

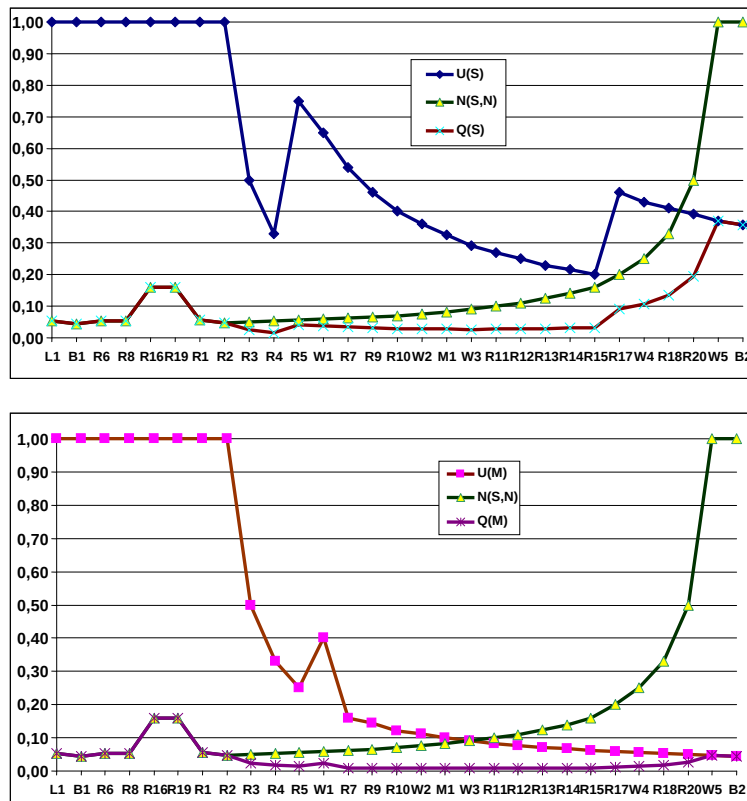


Рис. 1.24. Графіки тестопридатності Xilinx DCT-моделі

Лістинг 1.2.

```
c0: coverpoint xin
{ bins minus_big={{[128:235]}};
  bins minus_sm={{[236:255]}};
  bins plus_big={{[21:127]}};
  bins plus_sm={{[1:20]}};
  bins zero={{0}}; }
c1: coverpoint dct_2d
{ bins minus_big={{[128:235]}};
  bins minus_sm={{[236:255]}};
  bins plus_big={{[21:127]}};
  bins plus_sm={{[1:20]}};
  bins zero={{0}};
  bins zero2={{0=>0}}; }
endgroup
```

Для критичних точок, визначених в результаті аналізу тестопридатності транзакційного графа, розроблено асерційну модель перевірки основних характерис-

тик дискретного косинусного перетворення. Істотний фрагмент коду механізму асерцій представлено лістингом 1.3.

Лістинг 1.3.

```
sequence first( reg[7:0] a, reg[7:0]b);
    reg[7:0] d;
    (!RST,d=a)
    ##7 (b==d);
endsequence
property f(a,b);
    @(posedge CLK)
    // disable iff(RST||$isunknown(a)) first(a,b);
    !RST | => first(a,b);
endproperty
odin:assert property (f(xin,xa7_in))
    // $display("Very good");
    else $error("The end, xin =%b,xa7_in=%b", $past(xin, 7),xa7_in);
```

В результаті верифікації програмної HDL-моделі дискретного косинусного перетворення у середовищі моделювання Active-HDL було знайдено неточності у восьми рядках вихідного коду HDL-моделі:

```
// add_sub1a <= xa7_reg + xa0_reg; //
```

Подальше виправлення помилок дозволило отримати фрагмент коду, наведений у лістингу 1.4.

Лістинг 1.4.

```
add_sub1a <= ({xa7_reg[8],xa7_reg} + {xa0_reg[8],xa0_reg});
add_sub2a <= ({xa6_reg[8],xa6_reg} + {xa1_reg[8],xa1_reg});
add_sub3a <= ({xa5_reg[8],xa5_reg} + {xa2_reg[8],xa2_reg});
add_sub4a <= ({xa4_reg[8],xa4_reg} + {xa3_reg[8],xa3_reg});
end
else if (toggleA == 1'b0)
begin
add_sub1a <= ({xa7_reg[8],xa7_reg} - {xa0_reg[8],xa0_reg});
add_sub2a <= ({xa6_reg[8],xa6_reg} - {xa1_reg[8],xa1_reg});
add_sub3a <= ({xa5_reg[8],xa5_reg} - {xa2_reg[8],xa2_reg});
add_sub4a <= ({xa4_reg[8],xa4_reg} - {xa3_reg[8],xa3_reg});
```

1.13. Висновки з верифікації HDL-коду на основі асерцій

Розглянуто інноваційні технології тестопридатного проектування програмних і апаратних продуктів [22-37], орієнтовані на істотне зменшення часу верифікації системних програмних моделей цифрових виробів на кристалах. 1. Показано основні напрямки використання технологій тестопридатного проектування цифрових систем на кристалах в задачах тестування та верифікації програмних HDL-моделей. 2. Представлено універсальну модель програмного компонента у

вигляді транзакційного графа, на якому вирішуються задачі аналізу та оцінки тестопридатності з метою визначення критичних точок для розміщення асерцій і досягнення необхідної глибини діагностування HDL-коду. 3. Запропоновано логічні функції тестопридатності HDL-моделей на основі використання транзакційного графа в цілях обчислення оцінок керованості та спостережуваності програмних HDL-моделей, що дає можливість визначити критичні точки для подальшого поліпшення коду проекту шляхом встановлення асерцій. 4. Розглянуто приклади аналізу тестопридатності шляхом підрахунку керованості та спостережуваності транзакційного та керувального графів з метою визначення критичних точок з наступним розв'язанням практичної проблеми пошуку і усунення помилок у реальному DSP-проекті від компанії Xilinx. 5. Практична значущість запропонованих методик та моделей полягає в ринковій привабливості та високій зацікавленості технологічних компаній в інноваційних рішеннях проблеми тестування й верифікації програмно-апаратних виробів на системному рівні проектування в цілях зменшення time-to-market та підвищення виходу придатної продукції – yield. 6. Подальші дослідження спрямовано на розробку інтерфейсів для інтеграції моделей, методів і програмних продуктів у технологічні маршрути проектування цифрових систем на кристалах.

1.14. Контрольні запитання та завдання

1. Назвіть чотири провідні компанії планети в області EDA – Electronic Design Automation.
2. Як називаються моделі, що підтримують cycle-timed опис функціональності та комунікацій?
3. Як називаються моделі з un-timed деталізацією часу для функціональності та комунікацій?
4. Яка з TLM-моделей є найпоширенішою на ринку EDA?
5. Дайте стислу характеристику мови SystemC.
6. Дайте стислу характеристику мови SystemVerilog.
7. Назвіть конструктиви виконання цифрових систем на кристалах.
8. Назвіть основні типи функціональних компонентів цифрової системи на кристалі.
9. Опишіть типові структури NoC та їх призначення.
10. Опишіть мету і функції стандарту IEEE 1500 в інфраструктурі SoC.
11. Стадії виробництва і тестування силіконових кристалів для створення цифрових систем.
12. Охарактеризуйте взаємний вплив технологій проектування програмних і апаратних продуктів на два критерії: time-to-market, yield?
13. Наведіть криву графіка створення продукту, що включає фази проектування, налагодження та виробництва.
14. Дайте визначення понять: тестування, верифікація, валідація, специфікація, асерція.
15. Що являє собою транзакційний граф програмного коду?

16. Яке смислове навантаження тестопридатності несе ДНФ керованості та спостережуваності?
17. Як обчислюється тестопридатність транзакційного графа за логічними функціями тестопридатності?
18. Які методи діагностування програмного коду на основі асерцій використовуються для пошуку дефектних блоків HDL-моделі?
19. Як виглядає графік тестопридатності графа при послідовному і паралельному з'єднанні вершин?
20. З яких компонентів складається інфраструктура верифікації програмного коду?

1.15. Література

1. Spartan-6 Family Overview. Advance Product Specification.– XILINX.– February 2, 2009.– www.xilinx.com/
2. Stratix IV Device Handbook. – Altera.– 2009. www.altera.com/support
3. Appello D., Bernardi P., Grosso M., Reorda M.S. System-in-package testing: problems and solutions // IEEE Design & Test of Computers.– Vol. 23, Issue 3.– May-June, 2006.– P. 203-211.
4. Yervant Zorian. Tutorial on EWDTS 2007. Embedding Infrastructure IP for SOC Yield Improvement // Proceedings of the 39th Design Automation Conference.– New Orleans, LA, USA.– 2002.– P. 709-712.
5. P.P. Pande, C. Grecu, A. Ivanov, R. Saleh. Destination network-on-chip // EDA Tech Forum. University of British Columbia, chart the path from SoC to NoC.– June, 2005.
6. Horgan J. SiP or System-in-Package.– EDACafe.– September 29, 2008.– <http://EDACafe.com>
7. Zorian Yervant. Test Strategies for System-in-Package // The Plenary Paper of IEEE East-West Design & Test Symposium (EWDTS'08).– Lviv, Ukraine.– October 9-12, 2008.
8. L. Smith. 3D Packaging Applications, Requirements, Infrastructure and Technologies // Fourth Annual International Wafer-Level Packaging Conference. – San Jose, California. – September, 2007.
9. The next Step in Assembly and Packaging: System Level Integration in the package (SiP) / Editors: William Chen, W. R. Bottoms, Klaus Pressel, Juergen Wolf // SiP White Paper V9.0.– International Technology Roadmap for Semiconductors. – 2007.
10. Проектирование и тестирование цифровых систем на кристаллах / В.И. Хаханов, Е.И. Литвинова, О.А. Гузь.– Харьков: ХНУРЭ.– 2009.– 484 с.
11. IEEE Std 1800-2005.– IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language.– Published by The Institute of Electrical and Electronics Engineers, Inc.– USA.– 2005.– 664 p.

12. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).— IEEE Standard Verilog Hardware Description Language.— Published by The Institute of Electrical and Electronics Engineers, Inc.— USA.— 2001.— 791 p.
13. Ashenden, Peter J. The designer's guide to VHDL.— San Francisco, California: Morgan Kaufmann Publishers, Inc, 1996.— 688 p.
14. Pong P. Chu. RTL hardware design using VHDL. Coding for Efficiency, Portability, and Scalability.— Wiley-Interscience publication.— 2006.— 696 p.
15. Хаханов В.И., Хаханова И.В. VHDL + Verilog = Синтез за минуты.— Харьков: СМИТ.— 2007.— 264 с.
16. Donald E. Thomas, Philip R. Moorby. The Verilog Hardware Description Language.— New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.— 2001.— 404 p.
17. Schaumont P., Verbauwhere I. A Component-Based Design Environment for ESL Design.— Design & Test on computers.— 2006. — P. 338 - 347.
18. L. Cai, D. Gajski. Transaction Level Modeling: an overview // Hardware/ Software Codesign and System Synthesis.— 2003.— P. 19-24.
19. Chevallaz C., Mareau N. and Gonier A. Advanced Methods for SoC Concurrent Engineering.— In Proc. of Design, Automation and Test in Europe Conference (DATE'02).— 2002.— P. 59-63.
20. Nicolescu G., Yoo S. and Jerraya A. Mixed-Level Co-simulation for Fine Gradual Refinement of Communication in SoC Design.— In Proc. of Design, Automation and Test in Europe Conference (DATE).— 2001.— P. 754-759.
21. Douglas Densmore, Roberto Passerone, Alberto Sangiovanni-Vincentelli. A Platform-Based taxonomy for ESL design // Design&Test of computers.— September-October 2006.— P. 359-373.
22. Abramovici M., Breuer M.A. and Friedman A.D. Digital System Testing and Testable Design. Computer Science Press.— 1998.— 652 p.
23. Bayraktaroglu Ismet, Orailoglu Alex. The Construction of Optimal Deterministic Partitionings in Scan-Based BIST Fault Diagnosis: Mathematical Foundations and Cost-Effective Implementations // IEEE Transactions on Computers.— 2005.— P.61-75.
24. Francisco DaSilva, Yervant Zorian, Lee Whetsel, Karim Arabi, Rohit Kapur. Overview of the IEEE P1500 Standard // ITC Conference. 2003. P.988–997.
25. Rashinkar P., Paterson P., Singh L. System-on-chip Verification: Methodology and Techniques.— Kluwer Academic Publishers.— 2002.— 324 p.
26. Zorian Yervant. What is Infrastructure IP? // Design & Test.— 2002.— P. 5-7.
27. Zorian Yervant, Gizopoulos Dmytris. Gest editors' introduction: Design for Yield and reliability // IEEE Design & Test of Computers.— 2004.— P. 177-182.
28. Zorian Yervant. Guest Editor's Introduction: Advances in Infrastructure IP // IEEE Design and Test of Computers.— 2003.— 49 p.
29. Thatte S.M., Abraham J.A. Test generation for microprocessors // IEEE Trans. Comput.— 1980.— C-29, No 6.— P. 429-441.

30. Шаршунов С.Г. Построение тестов микропроцессоров. 1. Общая модель. Проверка обработки данных // Автоматика и телемеханика.– 1985.– №11.– С. 145-155.
31. Jerraya A.A. System Level Synthesis SLS. TIMA.– 2002.– P. 65-75.
32. Frank Ghenassia. Transaction Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Published by Springer.– 2005.– 282 p.
33. Bergeron, Janick. Writing testbenches: functional verification of HDL models.– Boston: Kluwer Academic Publishers.– 2001.– 354 p.
34. Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale. Verification Methodology. Manual for SystemVerilog.– Springer.– 2005.– 528 p.
35. Harry Foster, Adam Krolnik, David Lacey. Assertion-based design.– Second edition.– Kluwer Academic Publishers.– Springer.– 2005.– 392 p.
36. Rashinkar P., Paterson P., Singh L. System-on-chip Verification: Methodology and Techniques.– Kluwer Academic Publishers.– 2002.– 393 p.
37. Meyer A.S. Principles of Functional Verification. Elsevier Science.– 2004.– 206 p.

ЧАСТИНА II. VERILOG

Розглядаються конструктивні особливості, синтаксис і семантика мови опису апаратури Verilog, орієнтованої на створення цифрових систем на кристалах. У розділах 2-8 наведено структури проектів, конструкції операторів, (системних) директив, блоків, циклів, приклади створення моделей цифрових компонентів, інтерфейсів і пристроїв поведінкового, регістрового і вентильного рівнів опису.

Мета – надання широкому колу практично орієнтованих читачів технологічної культури проектування, аналізу та синтезу спеціалізованих цифрових систем на кристалах шляхом використання ефективного мовного середовища Verilog і численних прикладів апаратної реалізації алгоритмів.

Задачі: 1. Історія створення мови Verilog. 2. Основні поняття й конструкції: модуль як базова одиниця мови Verilog, рівні абстракції опису моделей, лексичні угоди, класи та типи даних, опис портів модуля, структурні моделі, параметри, директиви компілятора. 3. Засоби опису моделей вентильного рівня: стандартні логічні елементи, примітиви, що визначаються користувачем, затримки елементів; Dataflow моделей: оператор безперервного призначення, оператор умовного призначення, системні задачі та функції керування моделюванням; поведінкові моделі: блоки Initial і Always, процедурні оператори призначення, часове та подієве керування операторами в блоках, оператори керування (IF, CASE, Casez і Casex), цикли, спеціальні властивості блоків, процедурні оператори безперервного призначення. 4. Приклади поведінкових моделей пристроїв, підпрограми Task і Function, автоматичні функції та задачі, системні задачі роботи з файлами, засоби генерування операторів, оператор generate та змінні genvar, оператор generate з циклом for, умовний оператор generate, оператор generate з оператором case, приклади використання оператора generate, Programming Language Interface (PLI).

Джерела: стандарт мови опису апаратури Verilog [1, 2]; мова опису апаратури Verilog [4-6]; проектування цифрової апаратури за допомогою мови Verilog [3, 9-10]; інтерфейс програмування PLI [7, 8].

2. ОСНОВНІ ПОНЯТТЯ ТА КОНСТРУКЦІЇ МОВИ VERILOG

2.1. Модуль – базова одиниця мови Verilog

Основним елементом опису пристроїв у Verilog є модуль, який складається з декларацій та операторів [1, 2, 5, 6]. Він починається ключовим словом module та закінчується – endmodule. Синтаксис модуля:

```
// Старий стиль
module identifier (ports_list);
    ports_declaration;
    module_body;
```

```

endmodule
// Новий стиль
module identifier
    (ports_declaration);
    module_body;
endmodule

```

Лістинг 2.1 містить приклад пристрою, що реалізує операцію логічне І для двох 4-розрядних входів a і b , де результат надходить на вихід c .

Лістинг 2.1. Приклад опису модуля

```

// Старий стиль
module module_1(a, b, c);
    input [3: 0] a, b;
    output [3: 0] c;
    assign c = a & b;
endmodule
// Новий стиль
module module_1
    (input [3: 0] a, b, output [3: 0] c);
    assign c = a & b;
endmodule

```

2.2. Рівні абстракції опису моделей на Verilog

Для опису функцій модуля можуть бути використані конструкції одного з чотирьох рівнів абстракції: поведінковий або алгоритмічний (behavioral); рівень передачі даних (dataflow); вентильний рівень (gate-level); транзисторний рівень (switch-level) [3,4].

Поведінковий або алгоритмічний – це верхній рівень опису проєктів, що відображає алгоритм їх функціонування без деталей реалізації. У даному випадку модель поведінки подібна опису мовою C. Dataflow – описує процес зміни даних при їх передаванні між регістрами, але не належить до RTL (Register Transfer Level) опису. RTL структури призначені для синтезу моделей, які можуть містити конструкції поведінкового та dataflow-рівнів абстракції. На вентильному рівні модель представлена логічними елементами та з'єднаннями між ними. Транзисторний рівень (switch-level) – це найнижчий рівень абстракції. Мова Verilog допускає одночасне використання в модулі конструкцій всіх чотирьох рівнів опису.

Лістинг 2.2 являє собою модель D-тригера на поведінковому рівні: d – вхід даних, clk – вхід синхронізації, q – вихід. Для реалізації функції цього пристрою використовується блок always, в якому q отримує значення від входу d по кожному передньому фронту сигналу синхронізації clk .

Лістинг 2.2. Поведінковий опис Verilog-моделі D-тригера.

```

module dff
    (input clk, d, output reg q);
    always @(posedge clk)
        q = d;
endmodule

```


Лістинги 2.1 і 2.3 містять приклади моделей пристроїв dataflow-рівня. При цьому лістинг 2.3 реалізує мультиплексор 4-в-1 за допомогою булева рівняння:

$$\text{Out} = \text{Sel1}' \cdot \text{Sel0}' \cdot \text{In0} + \text{Sel1}' \cdot \text{Sel0} \cdot \text{In1} + \text{Sel1} \cdot \text{Sel0}' \cdot \text{In2} + \text{Sel1} \cdot \text{Sel0} \cdot \text{In3}.$$

Лістинг 2.3. Модель мультиплексора 4-в-1 рівня dataflow.

```
module mux4_to_1
    //Декларація портів
    (output out,
     input i0, i1, i2, i3, // Входи даних
     input s1, s0);        // Керувальні входи
    assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
                (s1 & ~s0 & i2) | (s1 & s0 & i3);
endmodule
```

Лістинг 2.4 являє собою вентильний опис мультиплексора 4-в-1, що відповідає схемі на рис. 2.1. Модель складається з убудованих в стандарт мови Verilog логічних елементів I (and), АБО (or) та НЕ (not).

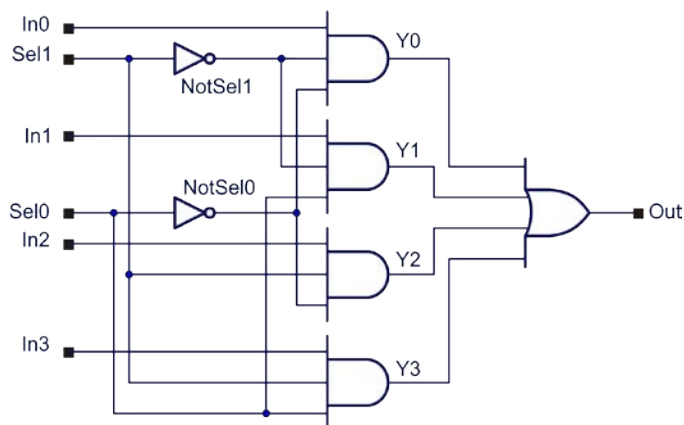


Рис. 2.1. Вентильна схема мультиплексора 4-в-1

Лістинг 2.4. Модель мультиплексора 4-в-1 вентильного рівня.

```
module mux_4_to_1
    (output Out,
     input In0, In1, In2, In3, Sel1, Sel0);
    wire NotSel0, NotSel1;
    wire Y0, Y1, Y2, Y3;
    and (Y1, In1, NotSel1, Sel0);
    not (NotSel0, Sel0);
    and (Y3, In3, Sel1, Sel0);
    or (Out, Y0, Y1, Y2, Y3);
    and (Y0, In0, NotSel1, NotSel0);
    not (NotSel1, Sel1);
    and (Y2, In2, Sel1, NotSel0);
endmodule
```

Рис. 2.2 містить МОП-схему вентиля АБО-НЕ (nor), а лістинг 2.5 – її Verilog-опис на транзисторному рівні.

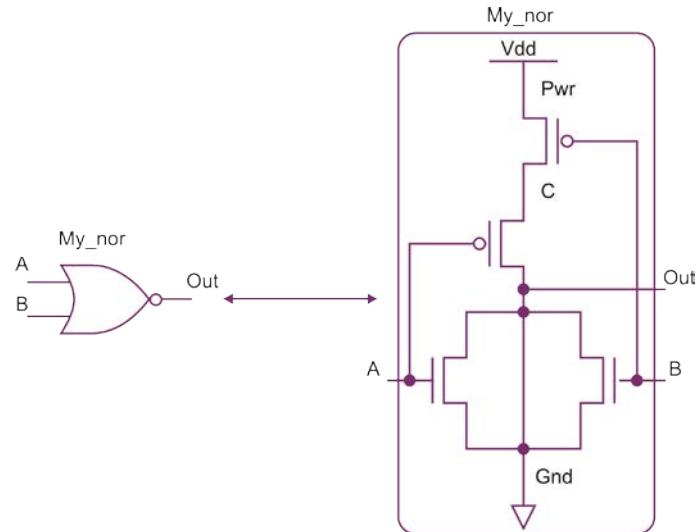


Рис. 2.2. Транзисторна схема вентиля АБО-НЕ (nor)

Лістинг 2.5. Модель елемента АБО-НЕ вентиляного рівня.

```
// Модель власного nor вентиля
module my_nor
    (output out, input a,b);
//внутрішні лінії
    wire c;
//створення ліній живлення та землі
    supply1 pwr;           // pwr з'єднується з Vdd (живлення)
    supply0 gnd;           // gnd з'єднується з Vss (земля)
// реалізації pmos-транзистор
    pmos (c, pwr, b);
    pmos (out, c, a);
// реалізації nmos-транзистор
    nmos (out, gnd, a);
    nmos (out, gnd, b);
endmodule
```

2.2.1. Структурні моделі.

Verilog дозволяє створювати ієрархічні моделі, в яких описи більш високого рівня можуть містити посилання на компоненти більш низького рівня ієрархії. При цьому використовується структурний стиль опису. Лістинг 2.6 являє собою Verilog-код пристрою, який складається з двох D-тригерів (див. лістинг 2.2). Імена inst_1 та inst_2 є індивідуальними мітками кожної копії компонента dff у пристрої top. Наступні за мітками списки містять імена факти-

чних сигналів, які будуть з'єднані з зазначеними у заголовку модуля dff портами.

Лістинг 2.6. Приклад структурної Verilog-моделі пристрою.

// модуль верхнього рівня, що використовує компонент dff (див. лістинг 2.2)

```
module top;
    reg data, clock;
    wire q_out, net_1;
    dff inst_1 (clock, data, net_1);
    dff inst_2 (clock, net_1, q_out);
endmodule
```

2.3. Лексичні норми

2.3.1. Опис імен.

Ідентифікатор імен у Verilog:

- може складатися з букв, цифр, \$, символа підкреслювання "_";
- повинен починатися з букви або символа підкреслювання;
- не може містити пробіли.

Як приклад можна навести такі конструкції:

```
reg enable;                wire _ready;
integer group_a;           reg and5;
tri clk$1;
```

2.3.2. Коментарі.

Для виділення коментарів, що являють собою рядок, використовуються дві похилі риски //. Символи зірочки /* і */ можуть бути застосовані для того, щоб відзначити початок і кінець блоку коментарів.

Оператори.

У мові Verilog розрізняють унарні, бінарні й тернарні оператори. В унарних операторах символ операції розміщується перед операндом, у бінарному – між операндами. Тернарний оператор використовує подвійний символ операції, що розділяє три операнди:

```
a = ~b;    // ~ – унарний оператор, b – операнд
a = b && c;    // && – бінарний оператор, b та c – операнди
a = b ? c : d;    // ? – тернарний оператор, b, c та d – операнди
```

2.3.3. Задання числових значень.

Числові значення можуть бути представлені з зазначенням або без зазначення розміру, можуть мати або не мати знак, бути цілими або дійсними.

Синтаксис цілих значень:

```
[sign] [size] 'base number
```

де sign – знак, size – кількість бітів для представлення значення number, base – система числення: d (D) – десяткова, h (H) – шістнадцяткова, o (O) – вісімкова, b (B) – двійкова. Число може складатися з цифр 0-9 та символів x, z.

Щоб розділити групи цифр у числі для поліпшення зорового сприйняття коду, можна використовувати символ підкреслення "_". Символ "?" є альтернативою z та ідентифікує високий імпеданс. При цьому даний символ у шістнадцятковій системі позначає 4 біти, що дорівнюють z, у вісімковій – три, у двійковій – 1.

Якщо беззнакове число менше відведеної для нього кількості розрядів, то число вирівнюється по правому краю з доповненням нулями до зазначеного розміру. Якщо лівий старший значущий символ дорівнює z або x, то додаток буде виконано цим символом. Приклади значень цілого типу:

```
15          'h f          'o 17          'd 15
'b 1111      'b 1_1_1_1    -5'b1_1011    10'd 20
8'h z        6'o 71
8'b0 (8'b00000000) 8'b1 (8'b00000001)
8'bz (8'bzzzzzzzz) 8'bx (8'bxxxxxxx)
4'b10?? (4'b10zz)  8'h4? (8'b0100zzzz)
12'h13x // 12-бітове шістнадцяткове число,
        // у якого значення останніх 4 бітів невідоме.
6'hx          // 6-бітове шістнадцяткове число
32'bz          // 32-розряди із значенням високого імпедансу
```

Знак у від'ємних числах ставиться перед символом системи числення. Числа представляються у додатковому коді:

```
4'd-2 // неправильний опис
-8'd3 // від'ємне число представлено у додатковому коді 8 розрядами
```

Синтаксис дійсних типів даних:

```
sign unsigned_number.unsigned_number
sign unsigned_number.unsigned_number e sign unsigned_number
sign unsigned_number.unsigned_number E sign unsigned_number
```

Приклади дійсних значень:

```
17.5          10e5
0.5            0.5694_e-5
1_000_000.0
```

2.3.4. Рядки.

Рядкові значення у Verilog записуються у подвійних лапках:

```
var = "Hello world!";
```

У рядках можуть бути використані спеціальні символи:

```
\n – новий рядок
\t – табуляція
\\ – символ \
\" – символ " .
```

2.4. Класи й типи даних

2.4.1. Сигнали.

На відміну від мови VHDL, у Verilog немає таких понять, як сигнал і змінна [4]. Всі дані носять назву змінних (variable). Виконання операцій призначення і збереження станів змінних визначаються їх класом (net або register) або

способом запису оператора присвоювання. Для моделювання поведінки реальної апаратури пропонується чотири можливих значення змінної, представлені у табл. 2.1.

Таблиця 2.1. Значення сигналів

Значення	Опис
0	Логічний 0, умова false
1	Логічна 1, умова true
x	Невідоме значення
z	Високий імпеданс

Змінні у Verilog поділяються на два класи: кола (nets) та регістри (register).

Кола (nets) є моделлю фізичного провідника між структурними компонентами пристрою, такими як вентиля. Само по собі коло не зберігає стан. Він повинен керуватися драйвером, який реалізується за допомогою оператора паралельного присвоювання (continuous assignment) або вентиля. Якщо коло не має драйвера, іншими словами, змінна не отримує значення, то вона приймає стан високого імпедансу z.

Регістр (register) як тип даних є моделлю елементів, здатних зберігати значення між двома виконаннями оператора присвоєння. Ключовим типом даних у регістровому класі є регістр – reg. За замовчуванням змінна регістрового класу ініціалізується значенням x.

Клас кола.

У загальному випадку синтаксис кіл виглядає так:

```
net_type [range] [delay3] list_of_net_identifiers ;
```

Параметр net_type може приймати одне з наступних значень, що позначають тип даних класу кола: wire, tri, tri1, supply0, wand, triand, tri0, supply1, wor, trior або trireg. Параметр range дозволяє створювати вектори зазначених типів даних для моделювання шин. Вони будуть розглянуті далі. Затримка лінії описується параметром delay3.

Для розв'язання конфліктів при одночасному присвоєнні значень у Verilog визначено вісім рівнів інтенсивності (strength level) сигналу. Більшість з них використовується для моделей транзисторного рівня. Синтаксис змінної класу кола з зазначенням інтенсивності (drive_strength або charge_strength) виглядає так:

```
net_type [drive_strength] [range] [delay3] list_of_net_decl_assignments ;
```

Типи wire і tri застосовуються для моделювання з'єднувальних ліній між елементами. Вони однакові за своїм синтаксисом і функціями. Два імені введено в мову для розрізнення мети використання змінних. При цьому wire вживається для сигналів, що мають в ролі джерела один драйвер або вентиль, а tri – для сигналів з кількома драйверами. Типи даних wand, wor,

triand та trior призначені для моделювання монтажною логіки: wand/triand – монтажне І, wor/trior – монтажне АБО.

У табл. 2.2 описуються функції розв’язання конфліктів, якщо змінна має кілька драйверів, за умови їх однакової інтенсивності.

Таблиця 2.2. Таблиці істинності для wire/tri, wand/triand та wor/trior

wire/tri					wand/triand					wor/trior				
	0	1	x	z		0	1	x	z		0	1	x	z
0	0	x	x	0	0	0	0	0	0	0	0	1	x	0
1	x	1	x	1	1	0	1	x	1	1	1	1	1	1
x	x	x	x	x	x	0	x	x	x	x	x	1	x	x
z	0	1	x	z	z	0	1	x	z	z	0	1	x	z

Наступні типи даних створено для моделей низького рівня. Тип trireg використовується для опису заряду запам'ятовуючого вузла (storage node). Кола типів даних: tri0 та tri1 – являють собою моделі резистивних пристроїв, які знижують (pulldown) та підвищують (pullup) рівні сигналів. Supply0 і supply1 моделюють напругу живлення.

Приклади опису змінних класу кола:

```
wire a;           // однобітова лінія a
wire b, c;        // дві однобітові лінії b та c
wire d = 1'b0;    // однобітова лінія d з початковим значенням 0
tri tristate_buffer; // тристабільна лінія tristate_buffer
wand #5 sig_1;    // однобітова лінія з монтажною логікою І та затримкою 5 одиниць
```

Регістровий клас даних.

На відміну від кіл регістрові змінні не мають потреби в драйвері, а самі зберігають своє значення. До регістрових відноситься кілька типів даних. Їх синтаксис:

```
reg [range] list_of_identifiers;
integer list_of_identifiers;
real list_of_identifiers;
time list_of_identifiers;
realtime list_of_identifiers;
```

Тип reg є основним для моделювання апаратури. Всі інші – допоміжні. Типи даних reg і time зберігають інформацію у вигляді беззнакових чисел. Integer використовується для маніпулювання та зберігання інформації, яка не розглядається як реальний регістр апаратури. Розмір integer за замовчуванням рівний 32 бітам, дані зберігаються у знаковій формі, арифметичні операції виконуються у додатковому коді. Тип Time подібний за функціями 64-бітовому регістру та використовується для зберігання параметрів часу.

Verilog також має дійсний тип даних real. Він подібний до решти регістрових типів, за винятком кількох обмежень: значення real дозволяється для використання не у всіх операторах Verilog та за замовчуванням ініціалізується нулем (решта регістрових типів ініціалізується символом x).

Також в Verilog існує дійсний аналог типу `time` для зберігання дійсних значень часу – `realtime`.

Приклад регістрових даних:

```
reg reset; integer i; time t; real r; realtime rt1, rt2;
```

2.4.2. Вектори.

Вектори є моделями шин в апаратурі і можуть бути визначені тільки для типів класу "кола" і для регістрового типу `reg`.

Синтаксис:

```
net_type [msb : lsb] list_of_net_identifiers;
```

```
reg [msb : lsb] list_of_register_identifiers;
```

де `msb` (найзначущий біт) – константа, що визначає ліву границю вектора, а `lsb` (наймолодший біт) – праву. Константи `msb` та `lsb` можуть мати будь-яке значення – позитивне, негативне або нуль. Константа `lsb` може бути більшою, рівною або меншою за `msb`.

Приклад векторів:

```
reg [3:0] addr; // 4-розрядний регістр addr
wire [-3:4] d; // змінна d розміром 8 бітів складається
               // з елементів: d[-3] (msb), d[-2], d[-1], d[0],
               // d[1], d[2], d[3], d[4] (lsb).
```

```
tri [5:0] x, y, z;
```

```
busA[7] // елемент шини busA з індексом 7
```

За замовчуванням вектори задають беззнакові цілі числа. Проте з використанням ключового слова `signed` можна визначити знакові числа. Найстарший біт розглядається як знак. Наприклад:

```
reg signed [3:0] signed_reg;
```

задає 4-бітовий вектор, який може приймати значення в діапазоні від -8 до 7.

2.4.3. Масиви.

Допускається створення масивів з елементів регістрових типів (`reg`, `integer`, `time`, `real`, `realtime`) і кіл. Елементи масиву (або слова) доступні за його індексом. Присвоювання та читання значень пам'яті можливе тільки словами, заборонено звернення до окремих бітів у слові, діапазоні або цілій пам'яті.

Синтаксис пам'яті:

```
reg memory_width memory_identifier memory_depth;
```

```
wire memory_width memory_identifier memory_depth;
```

```
integer | time memory_identifier memory_length;
```

Параметр апаратурного блоку пам'яті `memory_depth` задає розмір пам'яті, `memory_width` – розмір слова пам'яті типу `reg`. Одновимірний масив регістрів є моделлю апаратурного блоку пам'яті. Для позначення таких конструкцій використовується термін пам'ять або масив.

Приклади декларацій масивів:

```
reg [7:0] meta[0:255]; // декларація пам'яті meta розміром 256 8-бітових слів
                       // з індексацією від 0 до 255
```

```
reg arrayb[7:0][0:255]; // декларація двовимірного масиву однобітових регістрів
```

```

wire w_array[7:0][5:0]; // декларація двовимірного масиву ліній
time chng_hist[1:1000] // масив з 1000 значень типу time
integer count[0:7] // масив з 8 слов цілого типу
reg [4:0] port_id [0:7]; //масив port_id з восьми п'ятирозрядних слів

arrayb[1][0] = 0; // Присвоювання значення 0 елементу з індексом [1][0]
chng_hist[t_index] = $time; // Занесення поточного часу
//моделювання в комірку масиву з індексом t_index
count[5]; // звертання до п'ятого елемента масиву count
port_id[3]; // 3-й елемент масиву port_id шириною у 5 бітів

reg [7:0] mem [3:0], r; // Регістри й пам'ять можна визначати в одному рядку
// пам'ять mem з чотирьох 8-бітових слів і
// 8-бітовий регістр r

mem[0] = 7;
r = mem[3];
mem[1] = r;
reg [7:0] vect; // 8-бітовий вектор vect типу reg
reg array [7:0]; // пам'ять array з 8 однобітових слів типу reg
vect = 8'b11001010; // ініціалізація вектора значенням 11001010
array= 8'b11001010; // Неправильне звертання до пам'яті
array[7] = 1'b1; // Правильна ініціалізація пам'яті
array[6] = 1'b1;
array[5] = 1'b0;
array[4] = 1'b0;
array[3] = 1'b1;
array[2] = 1'b0;
array[1] = 1'b1;
array[0] = 1'b0;

```

2.5. Опис портів модуля

Для обміну інформацією із зовнішнім середовищем кожен модуль використовує порти, імена яких повинні бути зазначені в дужках після ідентифікатора імені модуля. Далі в декларативній частині вказується режим і тип кожного порту. Синтаксис модуля для декларації портів і змінних має такий вигляд:

```

module name_of_module (port, port);
    (input_declaration,
    output_declaration,
    inout_declaration);
    [net_declaration]
    [reg_declaration]
    [time_declaration]
    [integer_declaration]
    [real_declaration]
    other module_body ;
endmodule

```

У мові Verilog визначено три режими портів: input (вхід), output (вихід), inout (двонаправлений). Якщо тип даних порту явно не вказано, то він буде wire.

У прикладі (лістинг 2.7) для модуля описано два 4-розрядних входи і один 4-розрядний вихід типу `wire`, оскільки жоден інший тип даних для них не вказано. У прикладі, записаному в лістингу 2.8, вхідні порти отримують тип `wire`, а вихідний – `reg`.

Для портів природно мати тип `wire`, однак якщо вихідний порт зберігає своє значення, йому може бути присвоєний тип `reg`. Не допускається використовувати тип `reg` для портів з режимами `input` та `inout`.

Модуль може не мати ніяких портів.

Лістинг 2.7. Приклад декларації портів.

```
module module_1
    (input [3: 0] a, b,
     output [3: 0] c);
    ...
endmodule
```

Лістинг 2.8. Приклад декларації портів з явним зазначенням їх типів.

```
module module_1
    (input [3: 0] a, b,
     output [3: 0] c);
    wire [3: 0] a, b ;
    reg [3: 0] c;
endmodule
```

2.6. Структурні моделі

2.6.1. Способи з'єднання сигналів у структурній моделі.

На рис. 2.3 в графічній формі представлено правило використання типів портів і сигналів у випадку, якщо модуль є компонентом іншого, більш складного модуля. Як видно з рисунка, ті лінії, значення з яких тільки читаються (вихід модуля і лінія, з'єднана з входом модуля), можуть бути типу клас кола і типу `reg`. Для решти ліній дозволені тільки типи клас кола.

Синтаксис оператора реалізації копії модуля в структурній моделі:

```
module_name [parameter_assignment] name_of_instance
    (list_of_module_connections);
```

де `parameter_assignment` – фактичні значення параметрів (будуть розглянуті нижче), `module_name` – ім'я модуля, `name_of_instance` – ім'я копії модуля, `list_of_module_connections` – список фактичних змінних. Наприклад:
`my_module #(7, 25) inst_1(Clk, D, Q);`

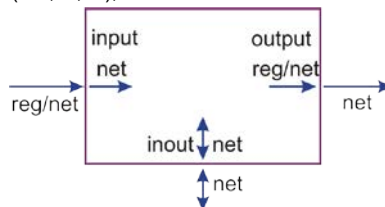


Рис. 2.3. Правило з'єднання портів

Існує два способи з'єднання змінних, зазначених у списку копії компонента, і фактичних портів модуля: по порядку та по імені. При зв'язуванні портів по порядку (лістинг 2.9) з'єднання змінних з портами визначається їх розташуванням у списку. При цьому перша змінна оператора реалізації підключається до першого порту зі списку опису модуля. Так, у прикладі змінна `clock` з модуля `top` буде підключена до порту `clk` модуля `dff`. У даному випадку можна залишати порти не зв'язаними, тоді при моделюванні передбачається, що на них подається значення `z`. Наприклад:

```
dff inst_1 (clock, , net_1);
```

Лістинг 2.9. З'єднання по порядку.

```
module dff
    (input clk, d,
     output reg q);
    ...
endmodule

module top;
    reg data, clock;
    wire q_out, net_1;
    dff inst_1 (clock, data, net_1);
    dff inst_2 (clock, net_1, q_out);
endmodule
```

Лістинг 2.10 являє собою приклад з'єднання портів по імені. У даному випадку спочатку пишеться ім'я порту, а потім у дужках — ім'я змінної, з якою він з'єднується. Пари порт–змінна можуть розташовуватися в будь-якому порядку. Незв'язані порти просто не вказуються.

Лістинг 2.10. З'єднання по імені.

```
module top;
    reg data, clock;
    wire q_out, net_1;

    dff inst_1 (.d(data), .q(net_1), .clk(clock));
    dff inst_2 (.clk(clock), .d(net_1), .q(q_out));
endmodule
```

Одним оператором можна створювати кілька копій компонента. Лістинг 2.11 ілюструє випадок створення чотирьох копій модуля `my_module` у компоненті `top`, завдяки використанню 4-розрядних змінних і лічильника `[3:0]` в операторі реалізації копії модуля.

Лістинг 2.11. Масив копій модулів.

```
module my_module
    (input a, b, output c);
    assign c = a & b;
endmodule

module top
    (input [3:0] a, b;
```

```

        output [3:0] c);
my_module inst [3:0] (a,b,c); // створює 4 копії модуля my_module
endmodule

```

2.6.2. Ієрархічні імена.

Verilog підтримує ієрархічну структуру проекту. При цьому кожен модуль, його копія і сигнал мають власний ідентифікатор або, іншими словами, ім'я. Кожен об'єкт має унікальне місце в структурі проекту, а використання ієрархічних імен дозволяє називати його унікальним ім'ям. Ієрархічне ім'я являє собою список ідентифікаторів, розділених крапкою (.), для кожного рівня ієрархії. Таким чином, використовуючи повне ім'я, можна звертатися до будь-якого об'єкта з будь-якого місця проекту.

Опис самого верхнього рівня називається модулем і з його імені починається ієрархічний ідентифікатор. Наприклад, для пристрою, представленого листингом 2.11, модуль `top` є верхнім рівнем ієрархії. У ньому оголошено ідентифікатори `a`, `b`, `c`, а також реалізовано копію модуля `my_module` з ім'ям `inst`. У модулі `my_module`, у свою чергу, визначено власні ідентифікатори `a`, `b`, `c`. У цьому випадку об'єкти модуля `top` будуть мати такі ієрархічні імена:

`top.a top.b top.c` и `top.inst`,

а об'єкти модуля `my_module` представлено списком:

`top.inst.a, top.inst.b` и `top.inst.c`.

2.7. Параметри

Параметри не відносяться до класу ліній і/або регістрових значень. Це константи і їх не можна використовувати як змінні. Значення параметра визначається індивідуально для кожної копії модуля на етапі компіляції. Декларація параметрів виконується перед описом портів і змінних модуля.

Синтаксис:

```

parameter [ signed ] [ range ] identifier = const_expr{, identifier = const_expr};
parameter integer identifier = const_expr{, identifier = const_expr};
parameter real identifier = const_expr{, identifier = const_expr};
parameter realtime identifier = const_expr{, identifier = const_expr};
parameter time identifier = const_expr{, identifier = const_expr};

```

Приклади визначення параметрів:

```

parameter lsb = 7;
parameter size = 8 , word = 32 ;
parameter number = 3.92, frequency = 100;
parameter clk_cycle = frequency / 2;
parameter byte_size = 8, byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;
parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;
parameter [31:0] dec_const = 1'b1; // значення перетворюється в 32 біта
parameter newconst = 3'h4; // використовуваний діапазон [2:0]
parameter newconst = 4; // використовуваний діапазон [31:0]

```

Лістинг 2.12 містить приклад, в якому розмір шин даних і затримка пристрою визначені за допомогою параметрів width і delay. Параметр width використовується в описі портів модуля, а delay – в операторі assign, який присвоює значення входу D виходу Q із затримкою 10 одиниць часу.

Лістинг 2.12. Приклад використання параметрів.

```
module my_module (Clk, D, Q) ;
    #(parameter width = 2, delay = 10)
    (input [width - 1:0] D, input Clk,
     output [width:0] Q);
assign #delay Q = D;
endmodule
```

2.7.1. Перевизначення параметрів.

В операторі реалізації модуля можна перевизначити значення параметра для даної копії компонента. Також це можна зробити за допомогою конструкції defparam. Тоді необхідно буде вказати ієрархічний шлях до модуля і його параметра. Синтаксис:

```
defparam hierarchical_path = constant_expression ;
```

Лістинг 2.13 являє собою приклад перевизначення параметрів в операторі реалізації копії модуля my_module (лістинг 2.12). При цьому параметри отримують нові значення: width = 7, delay = 25. Лістинг 2.14 ілюструє використання оператора defparam для зміни значення параметра width у додатковому модулі override.

При зміні параметра, від якого залежать значення інших параметрів, необхідно враховувати, що при зміні першого другі автоматично оновлюються.

Лістинг 2.13. Перевизначення параметрів в операторі модуля.

```
module top;
    reg Clk ;
    reg [7:0] D ;
    wire [7:0] Q ;
my_module #(7, 25) inst_1(Clk, D, Q) ;
endmodule
```

Лістинг 2.14. Перевизначення параметрів за допомогою defparam

```
module top;
    reg Clk;
    reg [7:0] D;
    wire [7:0] Q;
my_module inst_1(Clk, D, Q) ;
endmodule
module override;
    defparam top.inst_1.width = 7;
    // ієрархічний шлях top.inst_1 вказує розташування параметра width
    // в структурі вкладених модулів
endmodule
```

2.7.2. Локальні параметри.

Локальні параметри `localparam` ідентичні параметрам, за винятком того, що вони не можуть бути модифіковані за допомогою `defparam` або оператора реалізації модуля. Локальні параметри можуть отримувати значення в константних виразах, що містять параметри, які допускають перевизначення. Синтаксис:

```
localparam [signed] [range] identifier = const_expr{, identifier = const_expr};
localparam integer identifier = const_expr{, identifier = const_expr};
localparam real identifier = const_expr{, identifier = const_expr};
localparam realtime identifier = const_expr{, identifier = const_expr};
localparam time identifier = const_expr{, identifier = const_expr};
```

2.8. Директиви компілятора

У мові Verilog визначено такі директиви компілятора [5, 6].

2.8.1. Директива ``define`.

Використовується для визначення текстових макросів. Вона подібна до `#define` конструкції мови C. Синтаксис:

```
`define <text_macro_name> <MACRO_TEXT>
```

У декларації і при використанні текстових макросів застосовується символ зворотного апострофа ```. Наприклад:

```
`define TESTEQ1 4'b1101 // визначено текстовий макрос 1101
`define myzero 0 // визначено текстовий макрос myzero, що дорівнює нулю
assign mysig = `myzero; // використання текстового макросу myzero
// у будь-якому місці Verilog-коду замість нуля
`define WORD_REG reg [31:0] // ім'я WORD_REG може тепер бути
// використано для визначення 32-розрядних регістрів
`define WORDSIZE 64
reg [WORDSIZE-1 : 0] data_bus;
```

Макроси можуть містити параметри і можуть бути використані для реалізації функцій. Наступні оператори представляють приклад застосування макросу для створення функції порівняння двох чисел.

```
`define max(a,b) ((a) > (b) ? (a) : (b))
n = `max(p+q, r+s);
```

2.8.2. Конструкція ``ifdef`.

Verilog пропонує використовувати конструкцію ``ifdef` і ``endif`, що виконує перевірку декларацій текстових макросів для проведення умовної компіляції. Приклад з лістингу 2.15 ілюструє використання директиви ``ifdef` для вибору способу реалізації функції I.

Лістинг 2.15. Приклад використання директиви ``ifdef`.

```
module and_op (a, b, c);
    output a;
    inout b, c;
// вибір між RTL/GATE моделями за допомогою директиви `ifdef
```

```

`ifdef RTL
    wire a = b and c;
`else
    and a1(a, b, c);
`endif
.....
endmodule

```

2.8.3. Директива ``include`.

Verilog дозволяє розділити вихідний код на декілька файлів. Для використання інформації, що міститься в іншому файлі, можна застосовувати директиву ``include`:

```
`include "path/file-name-to-be-included"
```

Якщо файл визначений за допомогою директиви ``include`, він не вимагає додавання до проекту:

```
`include "/design/library/cells.v"
`include "/design/system/rest_of_system.v"
```

2.8.4. Директива ``timescale`.

Директива компілятора ``timescale` використовується для опису одиниць вимірювання і точності обчислення параметрів часу. Синтаксис:

```
`timescale time_unit / time_precision
```

де `time_unit` – одиниці виміру часу; `time_precision` – точність обчислення (округлення). Значення точності обчислення повинно бути менше або дорівнювати величині, зазначеній в одиницях виміру часу. Допустимі чисельні значення: 1, 10, 100; одиниці виміру s, ms, us, ns, ps, fs. Наприклад:

```
`timescale 10 ns / 100 ps
```

```
assign #5 Q = D; // Q отримає значення D після 10 ns затримки.
```

2.9. Контрольні завдання

1. Написати декларацію модуля для 4-розрядного регістра (рис. 2.4) з ім'ям `shift_reg`. Описати вхідні порти без внутрішньої реалізації пристрою.



Рис. 2.4. 4-розрядний регістр `shift_reg`

2. Створити модуль верхнього рівня `stimulus` з такими змінними: `REG_IN` (4 біти) та `CLK` (1-біт) типу `reg` і `REG_OUT` (4 біти) типу `wire`. Створити в ньому копію модуля `shift_reg` з ім'ям `sr1`. Використовувати спосіб з'єднання портів по порядку.

3. Виконати попереднє завдання, використовуючи для з'єднання портів – з'єднання по імені.

4. Записати ієрархічне ім'я змінних REG_IN та REG_OUT.
5. Написати ієрархічне ім'я для копії компонента srl. Написати ієрархічне ім'я для копії його портів.

Тестові запитання.

1. Яка з мов краще підходить для створення складних ієрархічних проектів цифрових пристроїв?
2. Якою організацією був розроблений стандарт мови Verilog?
3. Що описують моделі поведінкового рівня?
4. Що описують моделі dataflow-рівня?
5. Що описують моделі вентильного рівня?
6. Які класи змінних існують у Verilog ?
7. Який з атрибутів не є правильним Verilog-ідентифікатором?
8. Який опис відповідає 8-бітовому вектору типу лінія?
9. Яка з декларацій описує пам'ять з 20 елементів цілого типу?
10. Який опис відповідає пам'яті з 20 слів цілого типу?
11. Записано оператор присвоювання: A=23456; який формат має це число у Verilog?
12. Який вираз описує 16-бітове двійкове число?
13. Який елемент вектора (reg [7:0] bus) містить найстарший біт значення?
14. До змінної якого типу може бути підключений вихід модуля при побудові структурних моделей пристроїв?
15. Вказати назву стандартного примітиву.
16. Скільки виходів може мати визначений користувачем примітив?
17. Як описується поведінка визначеного користувачем примітива?

3. ВЕНТИЛЬНИЙ РІВЕНЬ ОПИСУ МОДЕЛЕЙ

3.1. Стандартні логічні елементи та моделі вентильного рівня

Вентильні й транзисторні моделі – це моделі низького рівня, які точно описують функцію і структуру схеми, використовуючи логічні елементи або примітиви МОП-транзисторів [9, 10]. Синтаксис для застосування вентилів у Verilog-кодi:

```
GATETYPE [strength] [delay] [name_of_instance] (terminals ),
      [name_of_instance] (terminals) ;
```

Оператор GATETYPE представлено множиною, що складається з таких ключових слів: and, nand, or, nor, xor, xnor, buf, bufif0, bufif1, not, notif0, notif1, pulldown, pullup, nmos, rnmos, pmos, rpmos, cmos, rcmos, tran, rtran, tranif0, rtranif0, tranif1, rtranif1; strength – інтенсивність драйвера; delay – затримка елемента; name_of_instance – необов'язкове ім'я копії логічного елемента; terminals – список з'єднувальних ліній, який може містити лише один, описуваний першим вихід, за яким зазначаються вхідні лінії. Запропоновані у Verilog логічні елементи для побудови моделей вентильного рівня представлено на рис. 3.1.

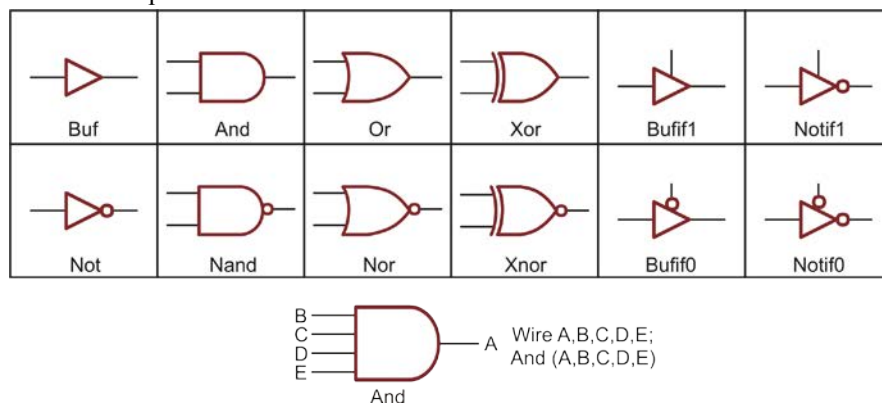


Рис. 3.1. Стандартні логічні елементи Verilog

Вентилі bufif0 bufif1 notif1 notif0 використовуються для справного моделювання тристабільних буферів, навантажених на один спільний вихід. При цьому стан загального виходу може приймати значення з множини сигналів {z, 0, 1}. Представлені вище вентилі мають три термінальні лінії, які записуються у список terminals в такому порядку: <вихід, вхід даних, керувальний вхід>. У табл. 3.1 наведено функції тристабільних вентилів, де символ L означає, що вихід може приймати одне з двох значень 0 або Z, символ H – 1 або Z. Затримки символів H і L подібні до затримок при переході лінії у стан X.

Приклад використання стандартних елементів для побудови цифрових схем дано на листингу 2.4.

Таблиця 3.1. Таблиці істинності тристабільних вентилів

bufif0		Control			
		0	1	x	z
D	0	0	z	L	L
a	1	1	z	H	H
t	x	x	z	x	x
a	z	x	z	x	x

bufif1		Control			
		0	1	x	z
D	0	z	0	L	L
a	1	z	1	H	H
t	x	z	x	x	x
a	z	z	x	x	x

notif0		Control			
		0	1	x	z
D	0	1	z	H	H
a	1	0	z	L	L
t	x	x	z	x	x
a	z	x	z	x	x

notif1		Control			
		0	1	x	z
D	0	z	1	H	H
a	1	z	0	L	L
t	x	z	x	x	x
a	z	z	x	x	x

3.2. Моделі комбінаційних схем

У лістингу 3.1 наведено ще один варіант вентиляної моделі мультиплексора 4-в-1, яка складається з 7 примітивів стандартних вентилів. На відміну від лістинга 2.4, описи вентилів групуються за їх типами, що робить модель більш компактною і ясною для розуміння.

Лістинг 3.1. Модель мультиплексора 4-в-1 вентиляного рівня

```
module mux_4_to_1
    (output Out,
     input In0, In1, In2, In3, Sel1, Sel0);
    wire NotSel0, NotSel1;
    wire Y0, Y1, Y2, Y3;
    and (Y1, In1, NotSel1, Sel0),
        (Y3, In3, Sel1, Sel0),
        (Y0, In0, NotSel1, NotSel0),
        (Y2, In2, Sel1, NotSel0);
    not (NotSel0, Sel0),
        (NotSel1, Sel1);
    or (Out, Y0, Y1, Y2, Y3);
endmodule
```

3.3. Визначені користувачем примітиви

Verilog дозволяє проектувальникові створювати свої власні елементи – визначені користувачем примітиви (UDP – User-defined primitives). Примітиви бувають двох типів: комбінаційні та послідовні. Алгоритм їх функціонування задається в табличній формі. Для цього використовуються спеціальні символи (табл. 3.2). Всі термінальні лінії повинні бути однобітовими. Вхідними значеннями примітива може бути 0, 1 та x. Значення z приймається рівним x. Для будь-якого користувального примітива допускається описувати один або кілька входів (input), але тільки єдиний вихід (output), який розташовується першим у списку термінальних ліній. Не можна задавати для

портів режим inout. Примітив описується на тому ж рівні, що і модуль, тому він не може бути вкладеним в останній. Синтаксис UDP-примітива:

```
primitive udp_name ( port_list );
    output output_port;
    input list_of_inputs;
initial output_port = value;    // ініціалізація вихідного значення
    table
        combinational_input_list : output_value;
        sequential_input_list : current_state : next_state;
    endtable
endprimitive
```

Лістинг 3.2 є прикладом комбінаційного примітива – мультиплексора 2-в-1 (рис. 3.2). Стан виходів таких примітивів залежить тільки від значень сигналів на його входах, тому вони не мають секції ініціалізації (ключове слово initial). Таблиця станів містить дві частини, які розділяють двокрапкою: область вхідних сигналів і область виходів. Порядок стовпців у таблиці визначено відповідним порядком запису вхідних змінних у списку портів port_list. Лістинг 3.3 містить приклад UDP послідовного примітива – D-тригера (рис. 3.3). Модель виконує ініціалізацію виходу q шляхом задання йому значення логічного нуля. Таблиця станів послідовного примітива містить три області: входи, поточний стан і наступний стан. Для того щоб виключити можливість появи невизначеного значення x на виході, необхідно покрити в таблиці всі можливі комбінації вхідних сигналів.

Таблиця 3.2. Опис символів для таблиць станів в UDP

Символ	Інтерпретація	Коментар
0	Логічний 0	
1	Логічна 1	
x	Невідоме значення	Не дозволяється для виходів
b	Замінює 0 та 1	
?	Замінює 0, 1 та x	
-	Відсутність змін	Використовується з виходами послідовних UDP
(v w)	Перемикання із значення v в w	Значення v та w можуть бути рівні 0, 1, b та ?*
*	Еквівалентно (??)	Будь-яка зміна значення*
r	Еквівалентно (01)	Передній фронт*
f	Еквівалентно (10)	Задній фронт*
p	Еквівалентно (01),(0x),(x1)	Позитивний фронт*
n	Еквівалентно (10),(1x),(x0)	Негативний фронт*
* Символи використовуються тільки для входів послідовних UDP		

Лістинг 3.2. Примітив мультиплексора 2-в-1.

```
primitive Mux_2to_1(Out, Sel, In0, In1);
    output Out;
    input Sel, In0, In1;
// Немає ініціалізації
Table
```

```
// Sel In0 In1 : Out
// -----
0 0 ? : 0; // Якщо Sel=0, то значення з In0 поступає на вихід
0 1 ? : 1;
1 ? 0 : 0; // Якщо Sel=1, то значення з In1 поступає на вихід
1 ? 1 : 1;
x ? ? : x; // Якщо Sel=x, то й значення виходу не визначено
// -----
endtable
endprimitive
```

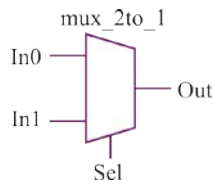


Рис. 3.2. Мультиплексор 2-в-1

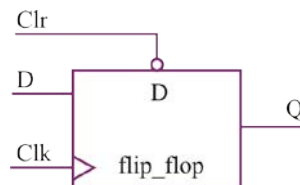


Рис. 3.3. D-тригер з інверсним входом скидання

Лістинг 3.3. Примітив D-тригера

```
primitive d_ff (q, d, clk, clr);
    input d, clk, clr;
    initial
        q=0;
    table
    //      d clk clr : q  : q+
    // -----
    ? ? 0 : ? : 0; // скидання тригера в 0
    0 r 1 : ? : 0; // запис 0 по передньому фронту clk
    1 r 1 : ? : 1; // запис 1 по передньому фронту clk
    ? n 1 : ? : -; // збереження при перемиканні clk (10), (1x) або (x0)
    * ? 1 : ? : -; // збереження значення за відсутності зміни на clk:
    ? ? * : ? : -;
    // -----
    endtable
endprimitive
```

Синтаксис оператора реалізації копії примітива UDP точно такий же, як і для будь-якого стандартного вентиля в Verilog. Лістинг 3.4 є прикладом використання UDP-примітива JK-тригера для створення моделі схеми з рис. 3.4. Кожній копії примітива присвоєна необов'язкова мітка Bit0, Bit1, Bit2 і Bit3.

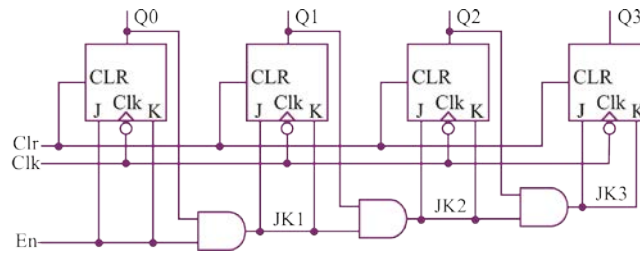


Рис. 3.4. Приклад схеми

Лістинг 3.4. Приклад використання UDP-примітивів.

```

module Count4En (Q, Clr, Clk, En);
output  [3:0] Q;
wire    [3:0] Q;
input   Clr, Clk, En;
wire    JK1, JK2, JK3;
JKMS Bit0 (Q[0], Clr, Clk, En, En);
JKMS Bit1 (Q[1], Clr, Clk, JK1, JK1);
JKMS Bit2 (Q[2], Clr, Clk, JK2, JK2);
JKMS Bit3 (Q[3], Clr, Clk, JK3, JK3);
and (JK1, Q[0], En);
and (JK2, Q[1], JK1);
and (JK3, Q[2], JK2);
endmodule

```

3.4. Затримки елементів

На практиці будь-якій електронній схемі необхідний кінцевий час на перемикання виходів у відповідь на зміну входів (затримка елемента) або на поширення сигналу вздовж провідника (затримка лінії). Вводяться такі затримки перемикання сигналів: затримка переднього фронту (rise delay) – час перемикання в 1 (рис. 3.5, а); затримка заднього фронту (fall delay) – час перемикання в 0 (рис. 3.5, б); затримка відключення (turn-off) – час, що асоціюється з часом переходу-виходу елемента з будь-якого стану в стан z.

Якщо значення виходу перемикається в x, то для моделювання використовується одна з описаних вище затримок, що має мінімальну величину.

```

#(delay)
#(rise_delay, fall_delay)
#(rise_delay, fall_delay, turn-off_delay)

```

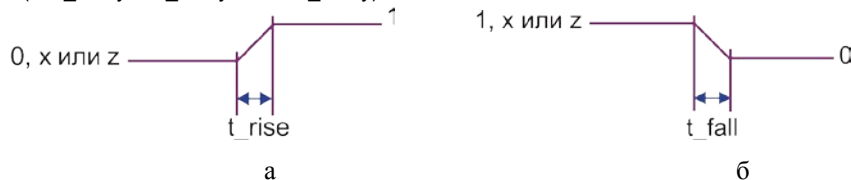


Рис. 3.5. Моделі затримок

Коли вказано один параметр, то для всіх типів перемикачів буде використано одне й те ж значення. Якщо зазначено дві величини, то вони розглядаються як затримки переднього і заднього фронтів, затримка відключення буде прийнята рівною мінімальному з вказаних значень. При наявності всіх трьох параметрів вони розглядаються у такому порядку: <затримка переднього фронту, затримка заднього фронту, затримка відключення>. При цьому кожне значення затримки може бути задано числом, ідентифікатором або у форматі: <мінімальна : типова : максимальна>:

#(number) | #(identifier) | #(min : typ : max, min : typ : max, min : typ : max).

Якщо затримка не описана, вона вважається рівною нулю. Одиниці виміру часу при записі затримок не вказуються, вони задаються для всього Verilog-коду директивою компілятора `timescale. Приклади опису затримок:

```
and #(5) a1(out, i1, i2); // затримка елемента для всіх типів перемикачів дорівнює 5
and #(4, 6) a2(out, i1, i2); // rise_delay=4, fall_delay=6, turn-off_delay=4
`timescale 1 ns / 1 ps
```

```
...
and #(3, 4, 5) a3(out, i1, i2); // rise_delay=3 ns, fall_delay=4 ns, turn-off_delay=5 ns
```

Лістинг 3.5 ілюструє використання затримок елементів при побудові моделі пристрою (рис. 3.6) вентиляного рівня.

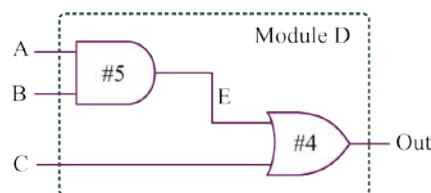


Рис. 3.6. Використання затримок при описі вентиляльних схем

Часові діаграми, отримані при тестуванні моделі, наведено на рис. 3.7. Модуль верхнього рівня, який формує тестові послідовності, представлений лістингом 3.6. Він не має портів, його внутрішні змінні A, B та C підключені до входів модуля D, мають тип reg і отримують значення за допомогою конструкції initial. Оператори в блоці виконуються послідовно, починаючи з нульового моменту часу. Одиниця виміру часу (1 ns) вказана у директиві `timescale. Системна задача \$finish використовується для завершення моделювання протягом 30 одиниць часу після його початку.

Лістинг 3.5. Verilog-модель вентиляного рівня з використанням опису затримок.

```
`timescale 1 ns / 1 ps
module D(out, a, b, c);
    // декларація вхідних портів
    output out;
    input a, b, c;
    wire e; // внутрішня лінія
    // реалізація копії примітивів вентилів
    and #(5) a1(e, a, b); // затримка вентиля 5 одиниць часу
```

```
or #(4) o1(out, e, c);           // затримка вентиля 4 одиниці часу
endmodule
```

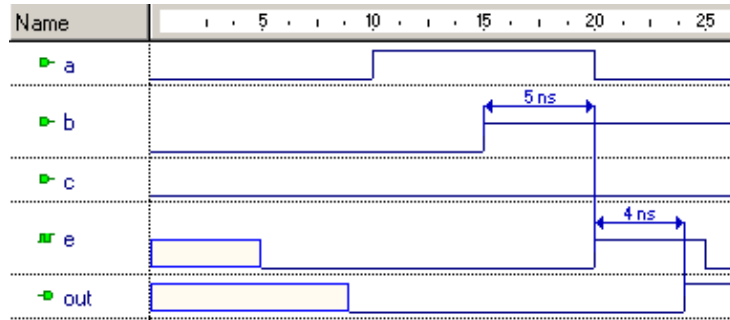


Рис. 3.7. Часові діаграми тестування пристрою з лістинга 3.4

Лістинг 3.6. Testbench для модуля D з лістинга 3.4.

```
`timescale 1ns/1ps
module stimulus_simple;
    // Декларація внутрішніх змінних
    reg A, B, C;
    wire OUT;
    // Реалізація копії модуля
    D d1(OUT, A, B, C);
    // Формування тестових послідовностей
    initial begin
        A = 1'b0; B = 1'b0; C = 1'b0;
        #10 A = 1'b1;    // A=1 в 10 ns
        #5  B = 1'b1;    // B=1 в 15 ns
        #5  A = 1'b0;    // A=0 в 20 ns
        #10 $finish;     // Моделювання завершиться за 30 ns
    end
endmodule
```

Verilog пропонує додатковий спосіб контролю для кожного типу затримок, дозволяючи описувати їх мінімальне, типове та максимальне значення. Будь-яка з цих величин може бути використана для моделювання. Цей підхід застосовується для тестування поведінки пристроїв, для яких параметри затримок вказуються діапазоном з мінімальною і максимальною границями. Метод вибору значень для моделювання залежить від програми-симулятора:

```
and #(4:5:6) a1(out, i1, i2);
and #(3:4:5, 5:6:7) a2(out, i1, i2);
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1, i2);
```

3.5. Контрольні завдання

1. Створити свої власні 2-входові логічні Verilog-елементи з іменами `my_and`, `my_or` та `my_not`, використовуючи тільки 2-входові вентиля `nand`.

Протестувати їх поведінку, побудувати файл, що генерує тестову послідовність.

2. Вентиль хог на два входи може бути побудований за допомогою логічних елементів `my_and`, `my_or` та `my_not`. Створити модуль `my_хог`, що реалізує функцію $z = xy' + x'y$. Написати тестовий модуль, що подає всі чотири можливі комбінації x та y .

3. Однобітовий суматор реалізований булевими рівняннями

$$\text{sum} = a \cdot b \cdot c\_in + a' \cdot b \cdot c\_in' + a' \cdot b' \cdot c\_in + a \cdot b' \cdot c\_in';$$

$$c\_out = a \cdot b + b \cdot c\_in + a \cdot c\_in.$$

Побудувати схему однобітового суматора, використовуючи стандартні логічні елементи `and`, `not`, `or`. Створити її Verilog-опис. Написати тестовий модуль для всіх можливих вхідних комбінацій a , b та c_in .

4. На рис. 3.8 наведено логічну схему RS-тригера і таблицю переходів.

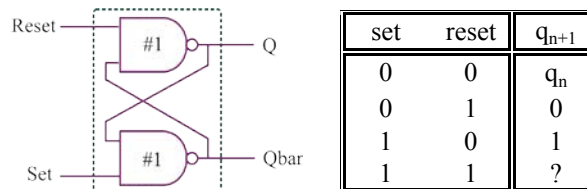


Рис. 3.8. Логічна схема RS-тригера і таблиця переходів

Створити Verilog-опис для RS-тригера, задаючи затримку в 1 одиницю часу при створенні копій вентильного лог. Написати тестовий модуль і перевірити функціонування RS-тригера, використовуючи його таблицю переходів.

5. Створити модель мультиплексора 2-в-1, використовуючи вентильні елементи `bufif0` і `bufif1`, як показано на рис. 3.9. Опис затримок для вентилів `b1` і `b2` наведено в табл. 3.3.

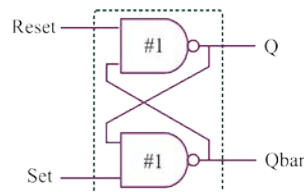


Рис. 3.9. Вентильні елементи `bufif0` та `bufif1`

Таблиця 3.3. Затримки для вентилів `b1` та `b2`

	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

6. Використовуючи стандартні примітиви, записати структурну модель схеми на рис. 3.10.

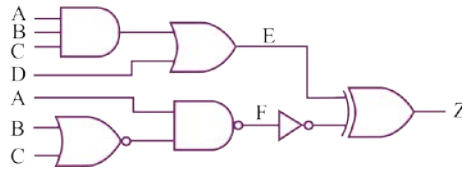


Рис. 3.10. Приклад схеми

7. Розробити стандартний примітив, що реалізує функцію: $f=abc+bcd$.

8. Спроекувати примітив UDP мультиплексора 4-в-1. Пристрій має керувальні сигнали $s0$ та $s1$, входи даних $i0, i1, i2, i3$, вихід out . Якщо керувальний сигнал $s0$ або $s1$ дорівнює x , то вихід $out = 0$.

9. Написати таблицю істинності та створити UDP для функції $Y = (A \& B) | (C \wedge D)$. Вхід ніколи не повинен набувати значення x .

10. Створити примітив користувача UDP, що має п'ять входів $a1, a2, a3, b1, b2$ та один вихід, поведінка якого описується функцією:

$$out = (a1 \& a2 \& a3) | (b1 \& b2)$$

Виконати моделювання пристрою з використанням такого блоку для формування тестових значень і спостереження за результатами:

```
initial begin
    $monitor($time, a1, a2, a3, b1, b2, out);
    for (i=0; i < 32; i=i+1)
        #10 {a1,a2,a3,b1,b2} = i;
end
```

11. Створити примітив UDP RS-тригера:

set	reset	q_{n+1}
0	0	q_n
0	1	0
1	0	1
1	1	?

12. Створити примітив JK-тригера з синхронізацією негативним фронтом, з асинхронними прямими сигналами устанавлення (preset) в 1 та скидання (clear) в 0:

J	K	q_{n+1}
0	0	q_n
0	1	0
1	0	1
1	1	q'_n

4. DATAFLOW МОДЕЛІ ЦИФРОВОГО ПРИСТРОЮ

4.1. Dataflow моделі та оператор безперервного визначення

Dataflow – це більш високий рівень опису моделей у порівнянні з вентильним рівнем. У даному випадку описується спосіб передавання даних між регістрами [5, 6, 9]. Основною конструкцією такого рівня моделей є оператор безперервного визначення (continuous assignment). Його синтаксис:

```
assign [drive_strength] [delay] net_value = expression {,net_value = expression};
```

де net_value – скалярна або векторна величина класу net (коло) або конкатенація цих елементів, де не допускається застосування змінних класу register. У виразі expression можуть бути використані операнди класів net та register або виклик функції.

Оператор безперервного визначення завжди буде активним. Як тільки один з операндів з правого боку змінює своє значення, обчислюється нове значення виразу, яке присвоюється змінній з лівої сторони:

```
assign out=i1&i2;
assign addr[15:0]=addr1[15:0]^addr2[15:0];
assign {c_out, sum[3:0]}=a[3:0]+b[3:0]+c_in;
```

У мові Verilog є можливість виконувати декларацію лінії в одному операторі та присвоювати їй значення:

```
//Звичайна форма
wire out;
assign out = i1 & i2;
//Спрощена форма
wire out = i1 & i2;
```

Затримки в операторі визначення.

Опис затримок в операторі безперервного призначення та їх властивості аналогічні затримкам вентилів:

```
wire out;
assign #10 out = i1 & i2;      // Звичайна форма
wire #10 out = i1 & i2;      // Спрощена форма
```

Затримку лінії можна також зазначити при її декларуванні:

```
wire #10 out;                // Затримка лінії
assign out = i1 & i2; // Значення лінії out буде присвоєне після
                        // затримки, що дорівнює 10 одиницям часу
```

4.2. Оператори

Verilog використовує символи для опису операторів, по аналогії з мовою C. У табл. 4.1 наведено список символів в порядку убутання пріоритету, а в табл. 4.2 – їх описи.

Таблиця 4.1. Пріоритети операторів

Оператор	Опис
+ - ! ~	Унарні
* / %	Арифметичні
+ - (binary)	Бінарні
<< >>	Зсуву
< <= > >=	Відношення
== != === !==	Рівності
& ~&	and nand
^ ~^ ^~	xor xnor
~	or nor
&&	Логічне and
	Логічне or
? :	Умовний оператор

Таблиця 4.2. Оператори Verilog

Арифметичні		Побітові	
Оператор	Опис	Оператор	Опис
a + b	a плюс b	~ b	not
a - b	a мінус b	a & b	and
a * b	a помножити на b	a b	or
a / b	a ділити на b	a ^ b	xor
a % b	a за модулем b	a ^~ b, a ~^ b	xnor

Відношення		Рівності	
Оператор	Опис	Оператор	Опис
a < b	менше	a == b	дорівнює
a > b	більше	a != b	не дорівнює
a <= b	менше або дорівнює	a === b	дорівнює
a >= b	більше або дорівнює	a !== b	не дорівнює

Логічні	
Оператор	Опис
a && b	a and b
a b	a or b
!a	not a

Арифметичні оператори наведено у табл. 4.2. Приклад їх використання:

A = 4'b0101; // визначено 4-розрядні регістри A, B

B = 4'b0010;

D = 5; E = 4; // змінні цілого типу D, E

A * B // результат операції множення – 4'b1010

D / E // результат операції ділення D на E – 1, остача відкидається

A + B // результат операції множення – 4'b0111

A - B // результат операції віднімання – 4'b0011

Якщо хоча б один біт операнда дорівнює x, результат арифметичної операції буде невідомий (дорівнює x):

```
in1 = 4'b110x;
in2 = 4'b1010;
sum = in1 + in2;           // результат операції додавання – 4'bx
```

Результат операції за модулем є остачею від ділення двох чисел. Знак результату збігається зі знаком першого операнда:

```
10 % 3 // результат 1, остача від ділення 10 на 3 дорівнює 1
17 % 3 // результат 2, остача від ділення 17 на 3 дорівнює 2
12 % 3 // результат 0, немає остачі від ділення
-7 % 3 // результат -1, знак першого операнда
11 % -3 // результат 2
-4'd12 % 3 // результат 1, операнд -4'd12 розглядається як велике
           // додатне число, остача від ділення якого на 3 дорівнює 1
```

Для останнього прикладу число -d12 d у додатковому коді буде представлене такими п'ятьма бітами b10100, однак вираз -4'd12 втрачає старший знаковий розряд і набуває значення 4'b0100. А результат буде дорівнювати $4'b0100 \% 3 = 1$.

Логічні оператори (табл. 4.2) використовуються для створення логічних виразів, для об'єднання операцій відношення й порівняння. Результатом застосування логічного оператора може бути 0 (false), 1 (true) або x.

Для цілочислових типів даних результат операції логічне І (&&) буде 0, якщо один з операндів дорівнює 0; результат операції логічне АБО (||) буде дорівнювати 1, якщо один з операндів має ненульове значення. Унарна логічна інверсія повертає значення 1, якщо операнд дорівнює 0, та 0 – в іншому випадку.

Приклад такої операції може мати вигляд:

```
A=3;    B=0;
A&&B=0  A||B=1  !A=0    !B=1
// Операнд з невідомими елементами
A = 2'b0x; B = 2'b10;
A && B    // Результат x, еквівалентно (x && 1)
```

Оператори відношення (табл. 4.2). Результат операції може бути 0 (false) або 1 (true). Якщо один з розрядів операндів дорівнює x або z, то результат оператора відношення дорівнює x:

```
reg [3:0] a, b;
a = 4'b1100;
b = 4'b0110;
a < b           // дорівнює логічному 0
a > 8           // дорівнює логічній 1
a <= b         // дорівнює логічному 0
a >= 10        // дорівнює логічній 1
a < 4'b1zzz    // результат дорівнює x
a < 4'b1x01    // результат дорівнює x
```

Оператори рівності (табл. 4.2). Оператори == та != видають значення x, якщо хоча б один розряд операнда дорівнює x або z. Оператори === та !== здійснюють побітове порівняння, дозволяючи використовувати в операндах сим-

воли x або z. У цьому випадку результат завжди є визначеним і дорівнює 0 або 1, він ніколи не може набувати значення x:

```
reg [3:0] a, b;
a = 4'b1100;
b = 4'b101x;
a == 4'b1100    // істина – логічна 1
a != 4'b1100    // неправда – логічний 0
b == 4'b101x    // результат дорівнює x
b != 4'b101x    // результат дорівнює x
b === 4'b101x   // істина – логічна 1
b !== 4'b101x   // неправда – логічний 0
```

Порозрядні або побітові оператори (табл. 4.2, 4.3). Оператор виконує логічні операції порозрядно над парами елементів – операндами. Якщо один з них коротший, то він доповнюється нулями з боку старших розрядів. Символ z обробляється як x.

Таблиця 4.3. Таблиці істинності побітових операторів

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

~	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

~	0	1	x	z
1	0	X	X	X

Приклади використання побітових операторів:

```
reg [7:0] a, b;
a = 8'b1010xzxz;
b = 8'b10010011;
a & b    // операція порозрядного І, результат дорівнює 8'b100000xx;
a | b    // операція порозрядного АБО, результат дорівнює 8'b1011xx11;
a ^ b    // операція порозрядного по модулю 2, результат дорівнює 8'b0011xxxx;
a ~^ b   // операція порозрядного по модулю 2 з інверсією, результат дорівнює
// 8'b1100xxxx;
~ a      // операція порозрядної інверсії, результат дорівнює 8'b0101xxxx;
```

Оператори редукції або згортки. Група унарних операторів: & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ^~ або ~^ (xnor) повертає однорозрядний результат, виконуючи зазначену логічну операцію між бітами операнда. Приклад застосування операторів редукції до змінного x=4'b1010:

```
&x      // 1&0&1&0=1'b0
|x      // 1|0|1|0=1'b1
^x      // 1^0^1^0=1'b0.
```

Наступний приклад представляє використання операторів редукції для формування контролю парності parity і рівності коду 11...1 all_ones.

```
module par
    (input [7:0] in,
     output parity, all_ones);
```

```

    assign parity = ^in;
    assign all_ones = & in;
endmodule

```

Оператори зсуву виконують зсув операнда, записаного ліворуч від оператора, на число розрядів, зазначених справа:

```

//x=4'b1100
y=x>>1 //y=4'b0110
y=x<<1 //y=4'b1000
y=x<<2 //y=4'b0000
reg [3:0] a;
a = 4'b1111;
a << 3 = 4'b1000
a >> 3 = 4'b0001
a << 1'bz = 4'bxxxx
a >> 1'bx = 4'bxxxx

```

Оператор конкатенації {,} пропонує механізм з'єднання декількох змінних в одну. Операнди записуються у фігурних дужках, через кому. При декларації операндів для них явно повинен бути зазначений розмір. Синтаксис:

```
{expression | multiplier{expression}, expression | multiplier{expression}, ... }
```

де expression - операнд, який може бути класу net або register: однобітовий, вектор або діапазон вектора. Операндам можна задавати константи повторення (multiplier):

```

//A=1'b1, B=2'b00,
//C=2'b10, D3'b110
Y={B, C} //Y=4'b0010
Y={A, B, C, D, 3'b001} //Y=11'b10010110001
Y={2{B}, C} //Y=6'b000010
Y={3{A}, 2{B}, C} //Y=9'b111000010

```

Лістинг 4.1 містить приклад використання побітових операторів і безперервного оператора призначення для створення моделі мультиплексора dataflow рівня. Функція пристрою описана у вигляді булевого рівняння.

Лістинг 4.1. Модель мультиплексора в-1 рівня dataflow.

```

//З використанням логічного рівняння
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
    //Декларація портів
    output out;
    input i0, i1, i2, i3; // входи даних
    input s1, s0; // керуючі входи
    assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
        (s1 & ~s0 & i2) | (s1 & s0 & i3);
endmodule

```

4.3. Оператор умовного призначення

Умовний оператор (?:) може приймати три операнда. Його синтаксис:

```
condition_expr ? true_expr: false_expr;
```

де `condition_expr` - описує умову, що визначає, який з наступних виразів буде обрано. Якщо `condition_expr=1` (має значення `true`), обчислюється вираз `true_expr`, інакше - вираз `false_expr`. Якщо умова дорівнює `x`, то обчислюються обидва вирази `true_expr` і `false_expr` і їхні значення порівнюються побітно. У кожній позиції, в якій біти різні, ставиться `x`, інакше значення бітів, що збіглися.

Лістинг 4.2 містить приклад побудови моделі мультиплексора 2-в-1 (рис. 2.2) з використанням умовного оператора. Вихід `out` отримає значення від входу `In1`, якщо `Sel=1`, інакше `out = In0`.

Лістинг 4.2. Умовний оператор у моделі мультиплексора 2-в-1.

```
module mux_2to_1 (out, In0, In1, Sel);
    output Out;
    input In0, In1, Sel;
    assign out = Sel ? In1 : In0;
endmodule
```

Умовні оператори можуть бути вкладеними один в інший. Приклад такого оператора, використаного для опису мультиплексора 4-в-1, наведений у лістингу 4.3. Обидві моделі мультиплексора 4-в-1, представлені лістингами 4.1 і 4.3, є моделями `dataflow`-рівня. Проте для реалізації функції пристрою в них використовувалися різні оператори Verilog.

Лістинг 4.3. Модель мультиплексора 4-в-1 з умовним оператором.

```
// Модель рівня передачі даних (dataflow) із застосуванням умовного оператора
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
//Декларація портів
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
//Вкладений умовний оператор
    assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
endmodule
```

4.4. Системні завдання й функції для керування моделюванням

4.4.1. `$display`.

В Verilog пропонується інструмент системних завдань для запуску підпрограм, що дозволяють виконувати вивід інформації на екран, моніторинг значень змінних, припинення й завершення процесу моделювання [5]. Такі завдання записуються у форматі `$keyword`. Нижче розглянуті рішення, які найчастіше використовуються на практиці.

Системне завдання `$display` служить для виведення інформації на екран. Її синтаксис:

```
$display | $displayb | $displayh | $displayo (p1, p2, p3, ...);
```

де `p1, p2, p3, ...` - можуть бути рядками, змінними або виразами. За замовчуванням для `$display` чисельні значення виводяться в десятковій формі.

Управляти форматом подання чисел можна за допомогою символів, перелічених у табл. 4.4. Варіанти виклику завдання з закінченнями `b`, `h` і `o` означають, що дані, для яких не зазначений формат, будуть виводитися у двійковій, шістнадцятковій і вісімковій системах числення. Лістинг 4.4 являє собою приклади для виведення інформації і значень змінних з використанням системного завдання `display`.

Таблиця 4.4. Символи подання формату для виведення даних

Символ	Опис формату
<code>%d</code> or <code>%D</code>	Десятковий формат
<code>%b</code> or <code>%B</code>	Двійковий формат
<code>%h</code> or <code>%H</code>	Шістнадцятковий формат
<code>%o</code> or <code>%O</code>	Вісімковий формат
<code>%c</code> or <code>%C</code>	Символи у форматі ASCII
<code>%v</code> or <code>%V</code>	Інтенсивності сигналу класу <code>net</code>
<code>%m</code> or <code>%M</code>	Ієрархічне ім'я
<code>%s</code> or <code>%S</code>	У вигляді рядка
<code>%t</code> or <code>%T</code>	Поточний формат часу

Лістинг 4.4. Приклади використання системного завдання `display`.

```
//виводиться рядок з лапок
$display("Hello Verilog World");
---i Hello Verilog World

//виводиться поточний час моделювання
$display($time);
---i 230

// виводиться час і значення параметра addr у цей момент
reg [0:40] addr;
$display("At time %d address is %h", $time, addr);
---i At time 200 address is 1fe0000001c

// виводиться значення 5-розрядного двійкового регістра
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
---i ID of the port is 11001

// виводиться у двійковій формі значення 4-бітові шини, що містить символ x
reg [3:0] bus;
$display("Bus value is %b", bus);
---iBus value is 10xx

// виводиться ієрархічне ім'я модуля, немає необхідності вказувати аргумент
$display("This string is displayed from %m level of hierarchy");
---i This string is displayed from top.p1 level of hierarchy

// виводяться спеціальні символи перекладу рядка й %
$display("This is a \n multiline string with a %% sign");
---i This is a
---i multiline string with a % sign
```

4.4.2. \$monitor.

Завдання \$monitor пропонує механізм моніторингу для спостереження за змінами сигналів. Його синтаксис:

```
$monitor | $monitorb | $monitorh | $monitord (p1, p2, p3, ...);
```

де p1, p2, p3, ... - може бути ім'я змінної або строкове значення в лапках. Дані виводяться при кожній зміні змінних, наведених у списку. Завдання із закінченнями b, h і o функціонують подібно до аналогічних модифікацій завдання \$display. Якщо в моделі описано декілька операторів моніторингу, активним буде тільки останній з них. Для відключення й включення моніторингу можуть бути використані такі завдання:

```
$monitoron;      // включити моніторинг
$monitoroff;     // виключити моніторинг
```

За замовчуванням, після запуску моделювання моніторинг вважається активним. Далі наводиться приклад використання системного завдання \$monitor для спостереження за сигналами синхронізації clock і скидання reset:

```
// Моніторинг значення сигналів clock і reset із вказівкою часу
// Сигнал clock перемикається кожні 5 одиниць часу
// Сигнал reset установиться в 0 в 10 одиниць часу
initial      // блок ініціалізації
    $monitor($time, "clock = %b, reset = %b", clock, reset);

//Фрагмент виведених при моніторингу даних
---i 0 clock = 0, reset = 1
---i 5 clock = 1, reset = 1
---i 10 clock = 0, reset = 0
```

Оператор initial описаний у розд. 5.

4.4.3. \$stop і \$finish.

Системні завдання \$stop і \$finish служать для керування процесом моделювання. Завдання \$stop дозволяє виконувати моделювання в інтерактивному режимі, припиняючи його, після чого проектувальник може перевіряти значення сигналів у точці зупинки. Завдання \$finish завершує процес моделювання й передає керування операційній системі. Синтаксис обох завдань:

```
$stop [(n)];
$finish [(n)];
```

де n – вираз, що визначає умову припинення або завершення процесу моделювання (табл. 4.5). Якщо цей параметр не зазначений, то за замовчуванням його вважають рівним 1. Приклади використання:

```
$stop;      // припинення процесу моделювання з виводом часу моделювання
#150 $finish(2); // завершення процесу моделювання з виводом часу моделювання,
                // часу роботи процесора й розміру використаної пам'яті
```


Таблиця 4.5. Режими діагностичних повідомлень завдань \$stop і \$finish

№ п/п	Повідомлення
0	Відсутнє
1	Час моделювання
2	Час моделювання і роботи CPU, розмір використовуваної пам'яті

4.4.4. \$printtimescale.

Наступні системні завдання й функції використовуються для відображення параметрів часу моделювання:

```
$printtimescale [ ( hierarchical_name ) ];
// функції
integer $time
integer $stime
real $realtime
```

Завдання \$printtimescale - виводить одиниці виміру й точність обчислення часу для зазначеного модуля hierarchical_name. Системні функції \$time і \$stime повертають поточний час моделювання у вигляді 64- і 32-бітового цілого числа, а функція \$realtime - у речовинному форматі. Наступний код містить приклад використання системного завдання \$printtimescale:

```
'timescale 1 ms / 1 us
module a_dat;
  initial
    $printtimescale;
endmodule
```

4.4.5. \$random.

Системна функція \$random забезпечує механізм для псевдовипадкової генерації значень. Розмір значень, що повертаються, відповідає типу integer. Типовий приклад використання функції \$random:

```
parameter SEED = 33;
reg [31:0] vector;
always @(posedge clock)
  vector = $random (SEED);
```

4.5. Контрольні завдання

1. Повний пристрій вирахування має три входи: x (зменшуване), y (від'ємник) і z (позиція з попереднього розряду). Виходи - D (різниця) і B (позиція). Функція може бути описана булевими рівняннями:

$$D = \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot z,$$

$$B = \bar{x} \cdot y + \bar{x} \cdot z + y \cdot z.$$

Розробити повний Verilog-опис пристрою вирахування. Написати модель верхнього рівня, сформувані тестові послідовності із усіх восьми комбінацій вхідних змінних. Виконати верифікацію пристрою вирахування:

x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

2. Амплітудний компаратор, що використовується для порівняння чисел, має два 4-розрядних входи $A[3:0]$ і $B[3:0]$. Порівняння може бути виконане за допомогою побітової операції, що відповідає операції $x \text{ nor } y$:

$$x(i) = A(i) \cdot B(i) + \bar{A}(i) \cdot \bar{B}(i) .$$

При будь-якій розбіжності операндів $x(i)$ дорівнює 0. Компаратор має три виходи: A_gt_B ($A > B$), A_lt_B ($A < B$) і A_eq_B ($A = B$), значення яких обчислюються за допомогою таких рівнянь:

$$A\_gt\_B = A(3) \cdot \bar{B}(3) + x(3) \cdot A(2) \cdot \bar{B}(2) + x(3) \cdot x(2) \cdot A(1) \cdot \bar{B}(1) + x(3) \cdot x(2) \cdot x(1) \cdot A(0) \cdot \bar{B}(0)$$

$$A\_lt\_B = \bar{A}(3) \cdot B(3) + x(3) \cdot \bar{A}(2) \cdot B(2) + x(3) \cdot x(2) \cdot \bar{A}(1) \cdot B(1) +$$

$$+ x(3) \cdot x(2) \cdot x(1) \cdot \bar{A}(0) \cdot B(0) ,$$

$$A\_eq\_B = x(3) \cdot x(2) \cdot x(1) \cdot x(0) .$$

Написати Verilog-модель компаратора й модуль верхнього рівня для виконання верифікації з використанням декількох значень A і B .

3. Синхронний лічильник може бути побудований з використанням двоступінчастого JK-тригера.

Нижче представлена вентильна схема JK-тригера (рис. 4.1). Якщо $clear = 0$, то тригер встановлюється в нульовий стан. Перший ступінь тригера зберігає значення по позитивному фронту сигналу $clock$, друга - по негативному.

Для лічильника (рис. 4.2) операція рахунку дозволена, коли сигнал $count\_enable = 1$. Розробити Verilog-модель лічильника $dataflow$ -рівня. Розробити тестовий файл.

4. Яким буде результат виконання такого оператора?

а) `latch = 4'd12;`

`$display("The current value of latch = %b\n", latch);`

б) `in_reg = 3'd2;`

`$monitor($time, "In register value = %b\n", in_reg[2:0]);`

в) `'define MEM_SIZE 1024;`

`$display("the maximum memory size is %h", 'MEM_SIZE);`

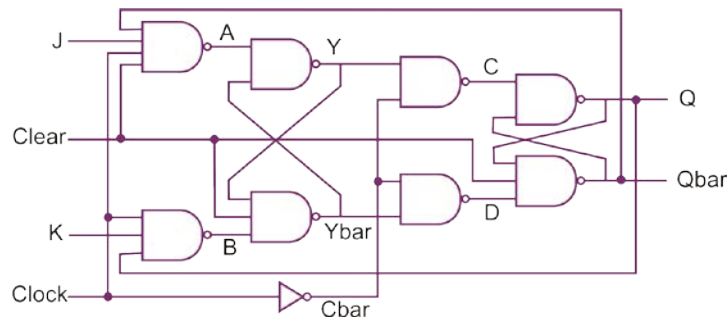


Рис. 4.1. Двоступінчастий JK-тригер

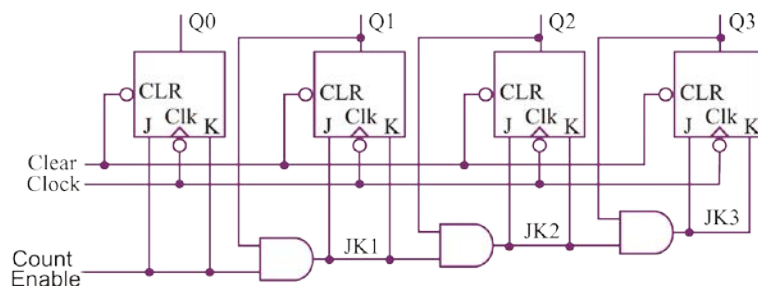


Рис. 4.2. Чотирирозрядний синхронний лічильник

Тестові запитання.

1. Що означає оператор $a === b$ в Verilog?
2. Що означає оператор $a !== b$ в Verilog?
3. Чому дорівнює результат операції $Y = \{4\{A\}\}$, де $A = 1'b1$?
4. Чому дорівнює результат операції $Y = \{3\{A\}\}$, де $A = 2'b0$?
5. Чому дорівнює результат операції $X|Y$ для змінних $X = 5'b10101$, $Y = 5'b00111$?
6. Чому дорівнює результат операції $X \& Y$ для змінних $X = 5'b10101$, $Y = 5'b00111$?
7. Чому дорівнює результат операції $X \wedge Y$ для змінних $X = 5'b10101$, $Y = 5'b00111$?
8. Чому дорівнює результат операції $\&X$ для $X = 5'b10101$?
9. Чому дорівнює результат операції $|X$ для $X = 5'b10101$?
10. Чому дорівнює результат операції $Y = \{A, B, C, D, 3'B101\}$ для $A = 1'b1$, $B = 1'b0$, $C = 2'b11$, $D = 2'b00$?
11. Укажіть правильний оператор, що описує затримку лінії в Verilog:

a) wire #10 out;	б) wire #10 ns out;
в) wire #(10 ps) out;	г) assign #10 ns out = in1 & in2;
12. Укажіть правильний оператор, що описує затримку лінії в Verilog:

a) #10 and (out, a, b);	б) wire #10 ns out;
в) assign #10 out = in1 & in2;	г) assign #10 ns out = in1 & in2;

5. ПОВЕДІНКОВІ МОДЕЛІ

5.1. Блоки Initial і Always

Поведінкові моделі – це моделі високого рівня, які описують алгоритм функціонування пристрою без деталізації на логічні вентиля або шляхи передачі даних [5, 6, 9]. Їхніми основними конструкціями є `initial` і `always`, усередині яких розміщуються всі інші оператори поведінкового рівня опису моделей.

Оператори `initial` і `always` не можуть бути вкладеними. Кожний з них являє собою окремий потік, що починає своє виконання в момент часу $t=0$. Присвоювати значення в блоках `initial` і `always` можна тільки змінним класу `register`.

Оператор ініціалізації починається із ключового слова `initial`. Він виконується один раз, відразу після початку моделювання. Якщо операторів `initial` декілька, кожний функціонує й закінчує своє виконання незалежно від іншого. У випадку, коли конструкція `initial` містить більше одного оператора, вони записуються між ключовими словами `begin` і `end` (лістинг 5.1).

Лістинг 5.1. Приклад використання ініціалізації.

```
module stimulus;
    reg x, y, a, b;      // визначення регістрових змінних
initial
    begin
        #5 a = 1'b1;      // a отримає 1 у момент часу 5,
                          // після старту моделювання
        #25 b = 1'b0;     // b отримає значення 0 у момент часу 30
    end
initial
    begin
        #10 x = 1'b0;     // x отримає значення 0 у момент часу 10
        #25 y = 1'b1;     // y отримає значення 1 у момент часу 35
    end
initial
    #50 $finish;         // завершення моделювання в момент часу 50
endmodule
```

Оператори, що містяться в `always`, виконуються безупинно, подібно циклу, починаючи з моменту часу $t=0$ (лістинг 5.2). Якщо модуль не має виразів `$stop` і `$finish`, моделювання відбувається в нескінченному циклі.

Лістинг 5.2. Приклад використання конструкції `always`.

```
module clock_gen;
    reg clock;           // визначення регістрової змінної
initial
    clock = 1'b0;        // ініціалізація синхросигналу 0
always
    // Перемикає clock з періодом 20 одиниць часу
    #10 clock = ~clock;
initial
    #1000 $finish;
endmodule
```

5.2. Процедурні оператори призначення

Процедурні оператори призначення сигналу використовуються в конструкціях `initial` і `always` і застосовуються для присвоєння значень змінним класу `register`: `reg`, `integer`, `real`, `realtime` і `time`. Синтаксис:

```
register_identifier = expression;
register_identifier <= expression;
```

де `register_identifier` може бути змінною або вектором регістрового типу, елементом пам'яті, фрагментом вектора регістрового типу або конкатенацією перелічених елементів. У виразі `expression` можуть бути використані змінні класів `net` і `register`. Розрізняють два види процедурних операторів призначення: блокуючі (`=`) і неблокуючі (`<=`).

Блокуючі (`blocking assignments`) процедурні оператори призначення виконуються в порядку, як вони записані в послідовному блоці. У паралельному блоці вони не заважають виконанню наступних за ними операторів. У прикладі з лістинга 5.3 у нульовий момент часу відбувається ініціалізація скалярних змінних `x`, `y` і `z`; змінної `count`, типу `integer` і двох регістрових векторів `reg_a` і `reg_b`. Потім, через 15 одиниць часу другий елемент вектора `reg_a` отримує значення 1. Наступний оператор виконується після затримки в 10 одиниць від поточного часу моделювання (15 одиниць часу) у момент, рівним 25 одиницям часу. Потім змінна `count` отримає значення в момент часу, рівний 25 одиницям, оскільки даний оператор має нульову затримку.

Лістинг 5.3. Приклад використання блокуючих операторів.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin                                // час виконання операторів
    x=0; y=1; z=1;                          // time 0
    count=0;                                // time 0
    reg_a=16'b0; reg_b = reg_a;             // time 0
    #15 reg_a[2]=1'b1;                      // time 15
    #10 reg_b[15:13]={x, y, z}              // time 25
    count=count+1;                          / time 25
end
```

Неблокуючі оператори починають виконуватися одночасно зі стартом усього їхнього блоку, що містить їх, залежно від зазначеного в них механізму керування. Вони дозволяють моделювати кілька шляхів зміни даних, що ініціалізуються однією загальною подією. У прикладі (лістинг 5.4) для присвоєння значень змінним `reg_a[2]`, `reg_b[15:13]` і `count` використовуються неблокуючі оператори, запуск яких буде ініціалізований одночасно після виконання зазначених перед ними блокуючих операторів у момент часу 0. Змінна `reg_a[2]` одержить значення після затримки в 15 одиниць часу, а `reg_b[15:13]` - після 10 одиниць часу від початку моделювання.

Лістинг 5.4. Приклад використання неблокуючих операторів.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin                                // час виконання операторів
    x=0; y=1; z=1;
    count=0;
    reg_a=16'b0; reg_b = reg_a;
    #15 reg_a[2] <= 1'b1;                    // time 15
    #10 reg_b[15:13] <= {x, y, z}           // time 10
    count <= count+1;                        // time 0
end
```

Лістинг 5.5 ілюструє механізми виконання блокуючих і неблокуючих операторів. У прикладі 1 використання блокуючих операторів призведе до виникнення гонок між змінними a і b. У цьому випадку конкретне значення змінних буде залежати від реалізації програми моделювання. У прикладі 2 обчислене значення заноситься в тимчасовий буфер і тільки потім присвоюється змінним, тому ніяких конфліктів не виникає. Виконання таких операторів рівносильно використанню блокуючих операторів із прикладу 3.

Лістинг 5.5. Порівняння блокуючих і неблокуючих операторів.

```
// Приклад 1
always @(posedge clock)
    a = b;
always @(posedge clock)
    b = a;

// Приклад 2
always @(posedge clock)
    a <= b;
always @(posedge clock)
    b <= a;

// Приклад 3
always @(posedge clock)
begin
    tmp_a = a;        //збереження значення a у часовій змінній tmp_a
    tmp_b = b;        // збереження значення b у часовій змінній tmp_b
    a = tmp_b;        // присвоєння значення a часовій змінній tmp_b
    b = tmp_a;        // присвоєння значення b часовій змінній tmp_a
end
```

5.3. Часове й подієве керування операторами в блоках

Існує три способи керування процесом виконання операторів: керування затримкою (delay control), використовує символ #; керування подією (event control), використовує символ @; керування значенням (level-sensitive timing control), використовує оператор wait.

5.3.1. Керування затримкою.

Керування затримкою описує час між читанням оператора і його виконанням. Синтаксис затримки такий же, як і при описі затримок вентилів і безперервних операторів призначення:

#NUMBER | #identifier | #(min:typ:max)

Якщо операторові задана ненульова затримка, то він буде виконаний через зазначений проміжок модельного часу. Наприклад:

```
reg x, y;
initial
begin
    x=0;      // немає керування, виконається відразу після початку моделювання
              // в time=0
    #10 y = 1; // затримка дорівнює 10 одиницям.
              // Змінна y отримає значення 1 через 10 одиниць часу,
              // у момент часу time = 10
    #y x=x+1; // використовується ідентифікатор для вказівки величини
              // затримки, оператор буде виконаний у момент часу time = 10+y
end
```

Для усунення гонок операторів, що обчислюються одночасно, використовується керування нульовою затримкою (zero-delay control), що гарантує послідовне їхнє виконання. На практиці звичайно не рекомендується одночасне присвоєння значення змінним у декількох операторах. Проте нульова затримка являє собою зручний механізм для керування порядком виконання операторів при моделюванні:

```
initial
begin
    x=0;
    y=0;
end
initial
begin
    #0 x=0;
    #0 y=0;
end
```

5.3.2. Керування подією (Event control).

Подія - це зміна значення регістра або ланцюга. Існує чотири типи подієвого керування порядком виконання операторів: звичайне подієве керування (regular event control), іменне подієве керування (named event control), подієве керування по АБО (event OR control) і керування з використанням оператора циклу repeat.

Звичайне подієве керування (regular event control) для опису використовує символ @. Вираз виконується після зміни або появи фронту, зазначеного в дужках сигналу. Для опису фронту застосовуються ключові слова posedge і negedge. Pposedge означає позитивний фронт - перемикання зі значення 0, x, z в 1 і з 0 в z або x (табл. 5.1).

Negedge являє собою негативний фронт - перемикання зі значення 1, x, z в 0 і з 1 в z або x. Приклади використання звичайного подієвого керування:

```
@(clock) q=d;           // q буде отримувати значення змінної d,
                        // при будь-якій зміні clock
@(posedge clock) q=d;    // q буде отримувати значення змінної d,
                        // по позитивному фронту clock
@(negedge clock) q=d;    // q буде отримувати значення змінної d,
                        // по негативному фронту clock
```

Таблиця 5.1. Визначення позитивного й негативного фронтів

Iz\B	0	1	x	z
0		posedge	posedge	posedge
1	negedge		negedge	negedge
x	negedge	posedge		
z	negedge	posedge		

Подієве керування по АБО (event OR control) дозволяє задавати кілька сигналів через ключове слово or. Виконання оператора відбувається при зміні хоча б одного із цих сигналів:

```
always @(reset or clock or d)
// блок буде ініціалізований при зміні хоча б одного із сигналів
    if(reset)
        q=1'b0;
    else
        q=d;
```

Новий стандарт дозволяє замінити слово or комою. Таким чином, представлений вище приклад може мати вигляд:

```
always @(reset, clock, d)
    if(reset)
        q=1'b0;
    else
        q=d;
```

Базова форма комбінаційної логіки, що реалізована за допомогою процедурних операторів, представлена лістингом 5.6 а). Список чутливості оператора always повинен у цьому випадку містити всі вхідні лінії схеми. Розподіленою помилкою в таких моделях є не повністю описаний список чутливості, у ньому пропущені сигнали, значення яких читається в межах оператора always. У нову версію стандарту внесли зміни, які дозволяють спростити опис оператора й уникнути помилок такого роду. Лістинг 5.6 б) містить змінену модель із лістингу 1 з використанням конструкції @(*).

Лістинг 5.6. Приклад комбінаційної схеми.

```
a)
module synCombinationalAlways
    (output reg f, input a, b, c);
always @ (a, b, c)
    if (a == 1)
```



```

        f = b ;
    else
        f = c;
endmodule

б)
module synAutoSensitivity
    (output reg f, input a, b, c);
always @ (*)
    if (a == 1)
        f = b ;
    else
        f = c ;
endmodule

```

Іменне подієве керування (named event control) вимагає виконання декларації події й створення умов для його перемикавання. Таку подію можна використовувати як будь-яку іншу, проте вона не зберігає ніякого значення. Для декларації використовується ключове слово `event`, для позначення перемикавання - символ `"->"`. Синтаксис:

```

// Декларація події
event event_identifier {, event_identifier } ;
//Перемикавання події
-> event_identifier ;

```

Тут `event_identifier` - ім'я події. Приклад використання:

```

event received_data; // визначена подія з ім'ям received_data
always @(posedge clock) // перевірка переднього фронту сигналу clock
begin
    if (last_data_packet->received_data; // якщо last_data_packet=1,
        // то відбувається перемикавання
    end // події received_data
always @(received_data); // очікує перемикавання події received_data
    data_buf = {data_pkt[0],data_pkt[1],data_pkt[2],data_pkt[3]}

```

Приклад використання іменних подій ілюструється моделлю пристрою генератора чисел Фібоначчі (лістинг 5.7). Модуль `numberGen` формує значення кожні 100 одиниць часу й перемикає подію `ready`. Модуль `fibNumCalc` одержує подію, до якої звертається через ієрархічне ім'я `@ng.ready`, де `ng` - ім'я копії модуля `numberGen`'s у структурній моделі `.topFib`. Результатом моделювання модуля `topFib` є числа Фібоначчі `numberOut`, що формуються на основі значень `number`, які генеруються модулем `numberGen`.

Лістинг 5.7. Генератор чисел Фібоначчі.

```

module topFib;
    wire [15:0] number, numberOut;
    numberGen ng (number);
    fibNumCalc fnc (number, numberOut);
endmodule

```

```

module numberGen
    (output reg [15:0] number = 0);
    event ready; // Декларація іменної події
always begin
    #50 number = number + 1;
    #50 -> ready; // Генерування події
end
endmodule

module fibNumCalc
    (input [15:0] startingValue,
    output reg [15:0] fibNum);
    reg [15:0] count, oldNum, temp;
always begin
    @ng.ready // Очікування події
        count = startingValue;
        oldNum = 1;
        for (fibNum = 0; count != 0; count = count - 1)
            begin
                temp = fibNum;
                fibNum = fibNum + oldNum;
                oldNum = temp;
            end
        $display("%d, fibNum=%d", $time, fibNum);
end
endmodule

```

5.3.3. Часове керування значенням.

Керування значенням (level-sensitive timing control) використовує оператор wait, що включає в себе функції керування подією й циклу while. Оператор виконується, якщо якась умова приймає значення істини. Синтаксис:

wait (expression) statement_or_null

де statement_or_null - означає який-небудь оператор або його відсутність. У наступному прикладі виконання оператора (#10 a = b;) буде відкладено, поки змінна enable не стане рівною 0:

```

begin
    wait (!enable) #10 a = b;
    #10 c = d;
end

```

5.3.4. Внутрішньооператорний контроль.

За місцем запису виразу, подієвого або часового керування, в операторах присвоєння значень виділяють внутрішньооператорне часове керування (intra assignment timing control). Його синтаксис:

```

reg_lvalue = [ delay_control | event_control | repeat_event_control ]
expression ;

```

або

```
reg_lvalue <= [delay_control | event_control | repeat_event_control]
expression ;
```

де delay_control - керування затримкою, event_control - подієве керування, repeat_event_control - конструкція repeat, що буде розглянута в підрозділі 5.4 "Цикли".

При внутрішньооператорному керуванні змінна отримає значення після зазначеної затримки (керування затримкою) або виконання умови (подієвий контроль). Однак обчислення цього значення відбувається відразу після прочитання оператора з поточними на той момент значеннями змінних. Після цього отримане значення зберігається програмою моделювання до виконання умови затримки або події. Наприклад:

```
reg x, y, z;           // Опис регістрових змінних
initial
```

```
    begin
        x=0; z=0;
        y = #5 x + z;
    end
```

//Описаний блок еквівалентний наступному

```
initial
    begin
        x=0; z=0;
        temp_xz = x + z;
        #5 y = x + z;
    end
```

q = @(posedge clock) d;

У наступному фрагменті коду:

```
always @(posedge clock)
```

```
begin
    reg1 <= #1 in1;           // у момент часу 1
    reg2 <= @(negedge clock) in2 ^ in3; // по задньому фронту clock
    reg3 <= #1 reg1;         // у момент часу 1
end
```

Після появи переднього фронту clock виконується читання розташованих із правого боку змінних (in1, in2, in3, reg1). Обчислюються значення виразів з правого боку, які зберігаються в пам'яті програми моделювання. Запис значень у змінні, розташовані з лівого боку, буде виконуватися відповідно до описів затримок і подій. Для неблокуючих операторів порядок їхнього виконання не залежить від послідовності запису, а визначається тільки параметрами керування. Черговість виконання операторів не вплине на результуюче значення reg2, оскільки програма моделювання зберігає старе значення reg1, що було зчитане при надходженні події @(posedge clock).

5.4. Оператори керування: if, case, casez і casex. Цикли

5.4.1. Оператор if. Синтаксис оператора if:

// Тип 1

```

if (expression) true_statement;

// Тип 2
if (expression) true_statement;
else faulse_statement;

// Тип 3
if (expression1) true_statement1;
else if (expression2) true_statement2;
else if (expression3) true_statement3;
else default_statement;
    faulse_statement;

```

Приклади використання оператора if представлені лістингом 5.8.

Лістинг 5.8. Приклади використання оператора if.

```

// Приклад 1
if (index > 0)
    if (rega > regb)
        result = rega;
    else // гілка else належить попередньому оператору if
        result = regb;

// Приклад 2
if (index > 0) begin
    if (rega > regb)
        result = rega;
    end
else result = regb;

// Приклад 3 з використанням конструкції else if
if (a)
    b = 4;    // якщо a не дорівнює 0, то b отримає значення 4
else if (d)
    b = 5;    // якщо a=0, а d не дорівнює 0, то b отримає значення 5
else
    b = 1;    // інакше, якщо a=0 і d=0, то b отримає значення 4

// Приклад 4
if (a) //гілка буде виконана, якщо a не дорівнює 0
    begin
        counter = counter + 1;
        data_out = counter;
    end
else
    data_out = 8'bz;

// Приклад 5
if ( counter > 10) // гілка не має оператора (нульовий оператор)
    ;
else
    counter = counter + 1;

```

5.4.2. Оператор case. Оператор case - це оператор множинного вибору.

Його синтаксис:

```
case (expression)
    expression: statement1;
    expression {, expression}: statement2;
    . . .
    default [:] default_statement;
endcase
```

Наступний приклад ілюструє використання case для вибору різних операторів, залежно від значення дворозрядної змінної address:

```
reg [1:0] address;
case (address)
    2'b00 : statement1;
    2'b01, 2'b10 : statement2;
    default : statement3;
endcase
```

Лістинг 5.9 містить поведінкову модель мультиплексора 4-в-1 з використанням оператора case. Для порівняння, лістинги 3.5 і 3.7 представляють цей пристрій на рівні dataflow. Мультиплексор має два керуючих входи s0 і s1. Щоб використовувати їх як єдиний операнд в операторі case, застосовується конкатенація. Гілка default буде виконана, якщо хоча б один з керуючих входів отримає значення, відмінне від нуля або одиниці.

Лістинг 5.9. Модель мультиплексора 4-в-1 з використанням case.

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
    //Декларація портів
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out; // визначення регістрової змінної виходу
            // для використання її в операторі always
    always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1,s2}) // конкатенація керуючих сигналів в 2-бітове значення
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default : $display("Invalid control signals");
    endcase
endmodule
```

5.4.3. Оператори casez і casex.

В Verilog визначені ще два різновиди оператора case: casex і casez. Вони дозволяють розглядати окремі біти у виразах вибору, як ті, що не мають значення. Такі позиції можуть бути позначені символом "?". Casez розглядає всі позиції із символом z як ті, що не мають значення, а для casex - це x і z. При використанні casex і casez виконується порівняння тільки не-x і не-z позицій.

Синтаксис:

```

casez (expression)
    expression: statement1;
    expression {, expression}: statement2;
    ...
    default: default_statement;
endcase
casex (expression)
    expression: statement1;
    expression {, expression}: statement2;
    ...
    default: default_statement;
endcase

```

Лістинг 5.10 містить приклади використання операторів casex і casez. Приклад 1 ілюструє застосування оператора casex для декодування бітів стану автомата. Нехай `encoding=4'b10xz`, тоді змінна `next_state` прийме значення 3. Приклад 2 представляє використання оператора casez. Декодування операції виконується по її старших значущих бітах. Якщо самий значущий біт дорівнює 1, то буде викликана команда `instruction1`, незалежно від значень інших бітів `ir`. У прикладі 3 за допомогою оператора casex демонструється динамічне керування процесом моделювання. Завдяки використанню маски `mask` при `r = 8'b01100110` буде обране завдання `stat2`.

Лістинг 5.10. Приклади використання операторів casex і casez.

```

// Приклад 1
reg [3:0] encoding;
integer state;
casex (encoding)
    4'b1xxx: next_state=3;
    4'bx1xx: next_state=2;
    4'bxx1x: next_state=1;
    4'bxxx1: next_state=0;
    default: next_state=0;
endcase

// Приклад 2
reg [7:0] ir;
casez (ir)
    8'b1??????: instruction1(ir);
    8'b01?????: instruction2(ir);
    8'b00010??? : instruction3(ir);
    8'b000001?? : instruction4(ir);
endcase

// Приклад 3
reg [7:0] r, mask;
mask = 8'bx0x0x0x0;
casex (r ^ mask)          // r ^ mask=8'bx1x0x1x0 для r=8'b01100110
    8'b001100xx: stat1;
    8'b1100xx00: stat2;

```

```

        8'b00xx0011: stat3;
        8'bxx010100: stat4;
    endcase

```

5.4.4. Константний вираз в операторі case.

Якщо оператор case використовує константні величини, то значення для варіантів вибору повинні являти собою вираз. Наприклад, з використанням константного виразу може бути побудований трьохбітовий пріоритетний декодер. Буде обрана перша гілка, для якої encode[i] = 1:

```

reg [2:0] encode ;
case (1)
    encode[2] : $display("Select Line 2") ;
    encode[1] : $display("Select Line 1") ;
    encode[0] : $display("Select Line 0") ;
    default : $display("Error: One of the bits expected ON" );
endcase

```

5.4.5. Цикли.

Verilog пропонує чотири види циклів: while, forever, repeat і for. Їхній синтаксис:

```

while (expression) statement;
forever statement;
repeat (expression) statement;
for (assignment; expression; assignment) statement;

```

Цикл repeat повторює виконання оператора statement із числом разів, заданим виразом expression. Якщо значення останнього невідомо або зафіксований високий імпеданс, то воно приймається рівним нулю. Це означає, що оператор ніколи не буде виконаний. Цикл while виконується, поки вираз expression не прийме значення false або 0. Конструкція forever виконує вираз нескінченне число разів. Для нього необхідно використовувати часове керування, інакше оператор може підвісити програму моделювання. В операторі for для керуючого параметра задається початкове значення, умова завершення й вираз модифікації.

Лістинг 5.11 містить приклади опису різних циклів.

У прикладі 1 цикл for використовує керуючу змінну count, що змінюється від 0 до 127 із кроком 1. Цикл forever із прикладу 2 застосовується для формування синхросигналу з періодом 20 одиниць часу, сигнал clock перемикає своє значення через кожні 10 одиниць часу. У прикладі 3 оператори, що містяться в циклі repeat, будуть оброблені 128 разів. Цикл while з прикладу 4 буде повторювати виконання операторів, поки tempreg = 1.

Лістинг 5.11. Приклади використання операторів циклу.

```

// Приклад 1
integer counter;
initial
    for(count=0; count<128; count=count+1) ... ;

// Приклад 2

```

```

initial
begin
    clock=1'b0;
    forever #10 clock = ~clock;    // генерує синхросигнал з періодом
                                   // 20 одиниць часу
    end

// Приклад 3
integer count;
initial
begin
    count = 0;
    repeat(128)          //цикл виконується 128 разів
    begin
        $display("Count = %d", count);
        count = count + 1;
    end
end

// Приклад 4
while (tempreg)
begin
    if (tempreg[0])
        count = count + 1;
        tempreg = tempreg >> 1;
    end
end

```

Наступний вираз ілюструє використання оператора repeat для внутрішнього подієвого керування в неблокуючих операторах присвоєння:

```
a <= repeat(5) @(posedge clk) data;
```

Значення data буде збережено відразу після появи оператора, змінна a отримає своє значення після надходження п'яти позитивних фронтів синхросигналу clk (рис. 5.1).

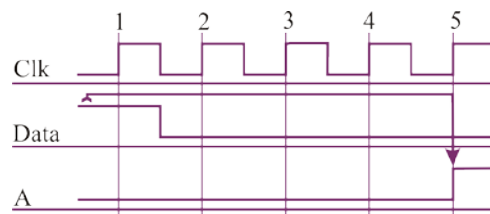


Рис. 5.1. Використання подієвого контролю з оператором repeat

Аналогічним чином використовується внутрішньооператорне подієве керування з repeat у процедурному блокуючому операторі присвоєння. Для виразу `a = repeat(num) @(clk) data;`

значення `data` буде зберігатися програмою моделювання з моменту появи оператора і до надходження `num` подій на `clk`, після чого воно буде присвоєно змінній `a`. Також в операторі `repeat` можна використовувати суму двох змінних, щоб указати необхідне число повторень подій:

```
a <= repeat(a+b) @(posedge phi1 or negedge phi2) data;
```

5.5. Спеціальні властивості блоків

Блоки дозволяють групувати кілька операторів, виконання яких розглядається як один. В Verilog є два типи блоків:

- послідовні, або `begin – end` блоки;
- паралельні, або `fork – join` блоки.

У попередніх прикладах для групування операторів використовувалися послідовні блоки із ключовими словами `begin – end`. У них послідовні оператори виконуються в тому порядку, у якому вони описані, за винятком команд із внутрішньооператорним часовим контролем (intra-assignment timing control). Якщо зустрічається затримка або подія, то час їхнього виконання відраховується від моменту завершення попереднього оператора.

Оператори в паралельних блоках виконуються одночасно; порядок виконання операторів контролюється затримками та подіями. Виконання всіх операторів починається в момент входу в блок. Тому затримки й події відносяться до початку виконання блоку, порядок запису операторів у блоці є несуттєвим. Після завершення виконання всіх операторів у паралельному блоці керування передається за блок.

Синтаксис блоків:

// послідовний блок	//паралельний блок
<code>begin : name</code>	<code>fork : name</code>
<code>statement;</code>	<code>statement;</code>
<code>...</code>	<code>...</code>
<code>end</code>	<code>join</code>

Опис послідовних і паралельних блоків ілюструється лістингом 5.12. Приклад 1 показує використання послідовного блоку, який не містить часове керування. Оператори блоку будуть виконані один по одному, відразу після його ініціалізації. Приклад 2 містить ті ж конструкції, що й приклад 1, але з використанням механізму керування затримкою. Тому перший оператор буде виконаний у момент часу 0, другий - у момент часу 5, третій - в 15, а останній - в 35. Приклад 3 представляє паралельний блок, оператори якого обробляються паралельно після зазначених у них затримок. Блоки можуть бути вкладеними або змішаними, приклад 4 ілюструє дану ситуацію. Приклад 5 розглядає ситуацію, коли паралельний блок використовується для перевірки виконання двох подій `@Aevent` і `@Bevent`, які можуть відбуватися в будь-якому порядку. Після цього блок передає керування наступному за ним оператору. Якби в прикладі замість паралельного використовувався послідовний блок, то черговість появи подій була б дуже істотна й приводила б до різних результатів моделювання.

Лістинг 5.12. Приклади використання блоків.

//Приклад 1

```
reg x,y; reg [1:0] z; reg [2:0] w;
initial
begin
```

```
    x = 1'b0;
    y = 1'b1;
    z = {x, y};
    w = {y,z};
```

```
end
```

// Приклад 2 - послідовний блок із затримками

```
reg x,y; reg [1:0] z; reg [2:0] w;
initial
begin
```

```
    x = 1'b0;
    #5 y = 1'b1;
    #10 z = {x, y};
    #20 w = {y,z};
```

```
end
```

//Приклад 3 - паралельний блок

```
fork
    #50 r = 1'b0;
    #100 r = 1'b1;
    #150 r = 1'b0;
    #200 r = 1'b1;
    #250 -> end_wave;
```

```
join
```

// Приклад 4 - паралельний блок є вкладеним у послідовний

```
initial
```

```
begin
```

```
    x = 1'b0;
    fork
        #5 y = 1'b1;
        #10 z = {x, y};
    join
    #20 w = {y, x};
```

```
end
```

// Приклад 5

```
begin
```

```
    fork
        @Aevent;
        @Bevent;
```

```
    join
```

```
areg = breg;
```

```
end
```

Кожному блоку може бути поставлено у відповідність ім'я. Такі іменні блоки є частиною ієрархії проекту. Вони можуть мати локальні змінні, які будуть доступні через ієрархічне ім'я. До них можна застосовувати механізм припинення або завершення виконання блоку:

```
module top;
initial
begin: block1          // послідовний блок з ім'ям block1
    integer i;          // ієрархічне ім'я top.block1.i
    ...

end
initial
fork: block2            // паралельний блок з ім'ям block2
    reg i;              // ієрархічне ім'я top.block2.i
    ...

join
```

Оператор `disable` надає спосіб для завершення виконання блоку. Він може бути використаний для виходу із циклу, обробки умов помилок, виходу з будь-якого іменного блоку проекту:

```
// виявляє перший біт зі значенням 1
reg [15:0] flag;
integer i;
initial
begin
    flag = 16'b0010_0000_0000_0000;    i = 0;
    begin: block1
        while (i<=16)
        begin
            if (flag [i])
            begin
                $display ("Encountered a TRUE bit at element number %d", i);
                disable block1;
            end
            i = i + 1;
        end
    end
end
end
```

5.6. Процедурні оператори безперервного призначення

Процедурні оператори безперервного призначення (procedural continuous assignments) (ключові слова `assign` and `force`) - це оператори, які дозволяють виконувати безперервний запис даних у змінні класів `register` або `net`. Їхній синтаксис:

```
assign reg_lvalue = expression;
deassign reg_lvalue;
force reg_lvalue = expression;
force net_lvalue = expression;
release reg_lvalue;
```

```
release net_lvalue;
```

В операторі `assign`, у лівій частині виразу, може бути використаний регістр або конкатенація регістрових змінних, але не можна застосовувати змінні класу `net`, елементи пам'яті, окремі біти або діапазон регістра. Пара `assign` – `deassign` – перекриває дію звичайних процедурних операторів призначення. Застосовується для керування сигналами в певний проміжок часу. Наприклад, вона може бути використана для моделювання асинхронних сигналів тригера (лістинг 5.13). Оператор `deassign` скасовує дію оператора `assign`, після чого значення регістра залишається незмінним, поки для нього не буде виконаний оператор процедурного призначення.

Лістинг 5.13. Поведінкова модель D-тригера.

```
module dff (q, qbar, d, clear, preset, clock);
    output q, qbar;
    input d, clear, preset, clock;
    reg q, qbar;
always @(clear or preset)
    if (!clear)
        assign q = 0;          // скидання в 0, якщо clear = 0
    else if (!preset)
        assign q = 1;          // установка в 1, якщо preset = 0
    else
        deassign q; // скасування дії оператора assign, якщо clear = 1 і preset=1
always @(posedge clock)
begin
    q = d;          // прямий вихід
    qbar = ~d;       // інверсний вихід
end
endmodule
```

Пара `force` і `release` може бути використана для перевантаження значень, що привласнюються змінним класів `net` і `register` процедурними операторами призначення або безперервного призначення. У лівій частині виразу може бути використана змінна або конкатенація змінних, не допускається вказувати елементи пам'яті, окремі біти або діапазон вектора. Оператор `force` застосовується при налагодженні в блоках, що формують тестові послідовності. Не рекомендується його використання в модулях проекту пристрою. Регістрова змінна буде мати значення, присвоєне виразом `force` до виконання оператора `release` (лістинг 5.14, приклад 1). Після цього вона збереже це значення до зміни його оператором призначення. Для змінних класу `net` виконання оператора `release` поверне змінній її нормальне значення (лістинг 5.14, приклади 2 і 3).

Лістинг 5.14. Приклади використання оператора `force`.

```
// Приклад 1 - для змінних класу register
module stimulus;
...
```

```

edge_dff dff(q, qbar, d, clk, reset);
...
initial begin
    #50 force dff.d = 1'b1;
    #50 release dff.q;
end
...
endmodule

// Приклад 2 - для змінних класу net
module top;
...
    assign out = a & b & c;
...
initial begin
    #50 force out = a | b | c;
    #50 release out;
end
...
endmodule

// Приклад 3
module test;
    reg a, b, c, d;
    wire e;
    and and1 (e, a, b, c);
initial begin
    $monitor("%d d=%b,e=%b", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10 $stop;
    release d;
    release e;
    #10 $finish;
end
endmodule

```

Результат виконання системного завдання \$monitor:

```

0 d=0,e=0
10 d=1,e=1
20 d=0,e=0

```

5.7. Приклади поведінкових моделей пристроїв

У лістингу 5.15 представлена модель лічильника, що виконує рахунок на збільшення від 0 до 15. Для установки в початковий нульовий стан використовується вхід clear, для перевірки максимальної межі - операція по модулю 16.

Лістинг 5.15. Модель 4-розрядного лічильника.

```
module counter (Q, clock, clear);
    output [3:0] Q;
    input clock, clear;
    reg [3:0] Q;
always @(posedge clear or negedge clock)
begin
    if (clear)
        Q = 4'd0;
    else
        Q = (Q + 1) % 16; // у деяких випадках оператор по модулю
                        // можна не використовувати
end
endmodule
```

Далі пропонується Verilog-модель пристрою, керуючого світлофором. Світлофор розташований на перетині головної (Main highway) і другорядної (Country road) магістралей. Головна магістраль має пріоритет, тому на ній завжди за замовчуванням горить зелене світло. На другорядній дорозі зелене світло спалахує тільки на час, необхідний для перетину машиною перехрестя. Якщо на другорядній дорозі відсутні машини ($X=0$), то загоряється червоне світло світлофора, поки датчик не виявить машину. Тоді він згенерує сигнал $X=1$, який ініціює процес перемикання світлофора. Граф переходів автомата й таблиця значень виходів представлені на рис. 5.2.

Перемикання між станами S1 і S2, S2 і S3, S4 і S0 виконується після контрольованої затримки. Лістинг 5.16 містить Verilog-код автомата. Для зручності й наочності стани автомата й значення виходів визначені за допомогою текстових макросів з використанням конструкції 'define. У моделі застосовується оператор initial для ініціалізації початкового стану автомата й значення його виходів. Поводження пристрою реалізовано за допомогою трьох операторів always: для відновлення по передньому фронту синхросигналу стану автомата, для обчислення значень виходів, для визначення наступного стану автомата. Testbench автомата й результати моделювання представлені лістингом 5.17.

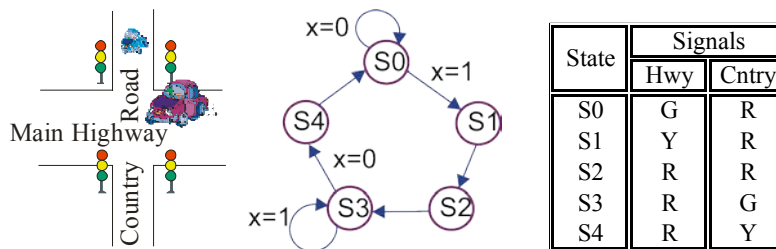


Рис. 5.2. Граф станів і таблиця значень виходів автомата

Лістинг 5.16. Поведінкова Verilog-модель автомата Мура.

```
// створення текстових макросів для вихідних функцій автомата
`define RED 2'd0          // червоний колір світлофора
`define YELLOW 2'd1       // жовтий колір світлофора
`define GREEN 2'd2        // зелений колір світлофора
//стану автомата  HWY      CNTRY
`define S0 3'd0           // GREEN    RED
`define S1 3'd1           // YELLOW   RED
`define S2 3'd2           // RED     RED
`define S3 3'd3           // RED     GREEN
`define S4 3'd4           // RED     YELLOW
// Затримки
`define Y2RDELAY 3        // Затримка перемикання з жовтого в червоний
`define R2GDELAY 2        // Затримка перемикання із червоного в зелений
module sig_control(hwy, cntry, X, clock, clear);
//Опис вхідних і вихідних портів
    output [1:0] hwy, cntry;
    reg [1:0] hwy, cntry;
    input X;
    input clock, clear;
// Внутрішні змінні для поточного й майбутнього станів автомата
    reg [2:0] state, next_state;
initial // Ініціалізація стану автомата й значень виходів
    begin
        state = `S0;      next_state = `S0;
        hwy = `GREEN;     cntry = `RED;
    end
always @(posedge clock) // відновлення значення стану автомата
    state = next_state; // по передньому фронту синхросигналу
always @(state)
    begin
        case (state)
        `S0: begin
                    hwy = `GREEN;
                    cntry = `RED;
                end
        `S1: begin
                    hwy = `YELLOW;

```

```

        cntry = `RED;
    end
`S2: begin
        hwy = `RED;
        cntry = `RED;
    end
`S3: begin
        hwy = `RED;
        cntry = `GREEN;
    end
`S4: begin
        hwy = `RED;
        cntry = `YELLOW;
    end
endcase
end
always @(state or clear or X)
begin
    if (clear)
        next_state = `S0;
    else
        case (state)
        `S0: if (X) next_state = `S1;
            else next_state = `S0;
        `S1: begin
            repeat (`Y2RDELAY) @(posedge clock);
            next_state = `S2;
        end
        `S2: begin
            repeat (`R2GDELAY) @(posedge clock);
            next_state = `S2;
        end
        `S3: if (X) next_state = `S1;
            else next_state = `S0;
        `S4: begin
            repeat (`Y2RDELAY) @(posedge clock);
            next_state = `S2;
        end
        default: next_state = `S0;
        endcase
    end
end
endmodule

Лістинг 5.17. Testbench для перевірки автомата Мура.
`define TRUE 1'b1
`define FALSE 1'b0
module stimulus;
    wire [1:0] MAIN_SIG, CNTRY_SIG;
    reg CAR_ON_CNTRY_RD; // якщо сигнал дорівнює TRUE, це означає,

```



```

        // що на другорядній дорозі перебуває автомобіль
        reg CLOCK, CLEAR;
sig_control sc(MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD,
               CLOCK, CLEAR);
initial      // Ініціалізація моніторингу
    $monitor($time, " Main Sig = %b Country Sig = %b car_on_cntry = %b",
             MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD);
initial begin // Генерування синхросигналу
    CLOCK = 'FALSE;
    forever #10 CLOCK = ~CLOCK;
end
initial begin // Керування сигналом скидання
    CLEAR = 'TRUE;
    repeat (5) @(negedge CLOCK);
    CLEAR = 'FALSE;
end
initial begin // Генерування вхідних імпульсів
    CAR_ON_CNTRY_RD = 'FALSE;
    repeat (3) begin
        #200 CAR_ON_CNTRY_RD = 'TRUE;
        #100 CAR_ON_CNTRY_RD = 'FALSE;
    end
    #100 $stop;
end
endmodule

//результати моделювання для 'timescale 1ns/1ps
# :      0 Main Sig = 10 Country Sig = 00 car_on_cntry = 0
# :     200 Main Sig = 10 Country Sig = 00 car_on_cntry = 1
# :     210 Main Sig = 01 Country Sig = 00 car_on_cntry = 1
# :     270 Main Sig = 00 Country Sig = 00 car_on_cntry = 1
# :     300 Main Sig = 00 Country Sig = 00 car_on_cntry = 0
# :     500 Main Sig = 00 Country Sig = 00 car_on_cntry = 1
# :     600 Main Sig = 00 Country Sig = 00 car_on_cntry = 0
# :     800 Main Sig = 00 Country Sig = 00 car_on_cntry = 1
# :     900 Main Sig = 00 Country Sig = 00 car_on_cntry = 0
# RUNTIME: RUNTIME_0069 stimulus.v (36): $stop called.

```

Таким чином, блоки `initial` і `always` - це основні оператори поведінкових моделей. Всі інші поведінкові оператори містяться усередині цих структур.

Блоки дозволяють групувати кілька операторів, виконання яких розглядається як один оператор. Блоки бувають послідовні й паралельні. Послідовні оператори виконуються в порядку їхнього проходження в коді програми.

Оператори в паралельних блоках виконуються паралельно, порядок їх виконання контролюється затримками й подіями; затримки й події відносяться до початку виконання блоку.

5.8. Контрольні завдання

1. Створити синхросигнал, який дорівнює нулю в початковий момент часу та що перемикається через кожні 30 одиниць часу. Використовувати для цього цикл `forever`.

2. Застосовуючи оператори `initial` і `always`, спроектувати синхросигнал з періодом 10 і тривалістю робочого циклу 25%. Початкове значення синхросигналу дорівнює 0.

3. У який момент часу буде виконаний кожен, наведений нижче, оператор? Чому рівні проміжні й остаточні значення сигналів `a`, `b`, `c`, `d`?

```
initial
    begin
        a = 1'b0;
        b = #10 1'b1;
        c = #5 1'b0;
        d = #20 {a, b, c};
    end
```

4. У який момент часу буде виконаний кожен, наведений нижче, оператор? Чому рівні проміжні й остаточні значення сигналів `a`, `b`, `c`, `d`?

```
initial
    begin
        a<=1'b0;
        b<=#10 1'b1;
        c<=#5 1'b0;
        d<=#20 {a, b, c};
    end
```

5. Який порядок виконання операторів у наступному Verilog-коді? Указати остаточні значення сигналів `a`, `b`, `c`, `d`?

```
initial
    begin
        a=1'b0;
        #0 c = b;
    end
begin
        b = 1'b1;
        #0 d = a;
    end
```

6. Вказати кінцеве значення `d` у такому прикладі:

```
initial begin
    b = 1'b1;
    c = 1'b0;
    #10 b = 1'b0;
end
initial
    d = #25 (b|c);
```

7. Обчислити результат моделювання таких блокуючих та неблокуючих операторів:

```
module e
```

```

        (output out);
        reg a,b,c;
        reg d,e,f;
    initial begin
        $monitor ("time=%d a=%d b=%d c=%d e=%d \n", $time,a,b,c,d,e);
        a=0; b=1; c=0; d=0; e=1;
        #30 $finish;
    end

    always c = #5 ~c;

    always @(posedge c) begin
        a = b;
        b = a;
        d<= e;
        e <= d;
    end
endmodule

```

8. Спроекувати D-тригер (D_FF) з негативним фронтом і синхронним скиданням, активним по 1. Використовувати тільки поведінкові оператори. Спроекувати тестовий модуль із синхросигналом, що має період 10.

9. Спроекувати D-тригер (D_FF) з негативним фронтом і асинхронним скиданням, активним по 1. Використовувати тільки поведінкові оператори. Спроекувати тестовий модуль із синхросигналом, що має період 10.

10. Побудувати D-тригер із синхронізацією за рівнем (latch), використовуючи оператор wait. Тригер має вхід даних d і вхід синхронізації clk, один вихід q.

11. Побудувати модель мультиплексора 4-в-1, використовуючи оператори if-else.

12. Побудувати пристрій керування світлофором (див. рис. 5.2), застосовуючи оператори if-else.

13. Використовуючи оператор case, спроекувати ALU, що виконує 8 функцій. Пристрій має два 4-розрядних входи даних a і b, один 3-розрядний вхід вибору операції OP, 5-розрядний вихід out. Функції ALU описані табл. 5.2.

14. Використовуючи оператор циклу while, побудувати генератор синхросигналу з початковим значенням 0 і періодом 10.

15. Застосовуючи оператор циклу for, виконати 0-ініціалізацію пам'яті.

16. Використовуючи оператор циклу forever, побудувати генератор синхросигналу з початковим значенням 0, періодом 10 і робочим циклом 40%.

17. Застосовуючи оператор repeat, створити затримку виконання оператора a = a + 1 на 20 позитивних фронтів сигналу clk.

18. Перетворити наступний цикл repeat у цикли for і while:

```

parameter DivsLength = 31
parameter DivdLength 63
parameter QuoLenght 31
parameter HiDminimum = 32

```

```

repeat(DivsLength+1)
begin
    quotient = quotient << 1;
    dividend = dividend << 1;
    dividend[DivdLength: HiDminimum] = dividend[DivdLength :
        HiDminimum] - divisor;
    if (!dividend [DivddLength])
        quotient = quotient + 1;
    else begin
        dividend[DivdLength:Himinimum] =
            dividend[DivdLength:Himinimum] + divisor;
    end
end
end

```

Таблиця 5.2. Функції ALU

OP	Функція
3'b000	out = a
3'b001	out = a + b
3'b010	out = a - b
3'b011	out = a / b
3'b100	out = a % b
3'b101	out = a << b
3'b110	out = a >> b
3'b111	out = (a > b), порівняння амплітуд

19. Коли закінчиться виконання таких вкладених послідовних і паралельних блоків? У якому порядку вони будуть виконані?

```

initial begin
    x = 1'b0;
    #5 y = 1'b0;
    fork
        #20 a = x;
        #15 b = y;
    join
    #40 x = 1'b0;
    fork
        #10 p = x;
        begin
            #10 a = y;
            #30 b = x;
        end
        #5 m = y;
    join
end

```

20. Спроекувати 8-розрядний лічильник, використовуючи цикл forever, іменні блоки й оператор disable для виходу із блоку. Пристрій починає рахунок з 5 і закінчує на 67, працює по позитивному фронту синхросигналу, пе-

ріод якого дорівнює 10. Цикл рахунку виконується тільки один раз, після чого за допомогою оператора `disable` здійснюється вихід із блоку.

21. Спроектувати D-тригер із синхронізацією по передньому фронту, з асинхронними сигналами скидання (`reset`) і установки (`set`). Використовувати `assign` і `deassign`.

22. Побудувати модель повного суматора, застосовуючи стандартні логічні елементи Verilog. Побудувати тестовий модуль, використовуючи оператор `force`, призначити виходу значення (`a & b & c_in`) у проміжок часу 15-35 одиниць.

23. Однобітовий суматор визначений логічними елементами й затримками:

```
module fulladd(sum, c_out, a, b, c_in);
    parameter d_sum = 0, d_cout = 0;
    //Декларація портів
    output sum, c_out;
    input a, b, c_in;
    // Внутрішні лінії
    wire s1, c1, c2;
    // Копії стандартних елементів
    xor (s1, a, b);
    and (c1, a, b);
    xor #(d_sum) (sum, s1, c_in);
    and (c2, s1, c_in);
    or #(d_cout) (c_out, c2, c1);
endmodule
```

Використовуючи однобітовий суматор, розробити структурну модель 4-розрядного суматора `fulladd4`. Установити значення параметрів для копій модулів з табл. 5.3:

Таблиця 5.3. Значення параметрів для копій модулів

Копія модуля	Значення параметрів
fa0	d_sum = 1, d_cout = 1
fa1	d_sum = 2, d_cout = 2
fa2	d_sum = 3, d_cout = 3
fa3	d_sum = 4, d_cout = 4

а) Побудувати структурну модель `fulladd4`, використовуючи оператор `defparam` для зміни значень параметрів.

б) Побудувати структурну модель `fulladd4`, перевизначаючи параметри в операторі створення копії модуля.

Побудувати тестовий модуль. Виконати верифікацію обох моделей. Зрівняти результати верифікації обох моделей.

24. З моделі повного суматора (див. попередню задачу) створити проект, що використовує умовну компіляцію (`'ifdef`). Версія проекту із застосуванням оператора `defparam` для зміни значень параметрів компілюється, якщо в проєкті з використанням директиви (`'define`) визначений макрос `DPARAM`, інакше компілюється код зі зміною параметрів в операторах копій модулів.

25. Чому буде дорівнювати змінна result після виконання таких операторів, якщо rega = 16'd3?

```
reg [15:0] rega;
reg [9:0] result;
case (rega)
  16'd0: result = 10'b0111111111;
  16'd1: result = 10'b1011111111;
  16'd2: result = 10'b1101111111;
  16'd3: result = 10'b1110111111;
  16'd4: result = 10'b1111011111;
  16'd5: result = 10'b1111101111;
  16'd6: result = 10'b1111110111;
  16'd7: result = 10'b1111111011;
  16'd8: result = 10'b1111111101;
  16'd9: result = 10'b1111111110;
```

```
default result = 'bx;
endcase
```

26. Намалювати часову діаграму (рис. 5.3, 5.4), що буде отримана в результаті виконання таких операторів:

```
initial x = 1'b0;
initial
  repeat (3) @(posedge clk) x=~x;
```

27. Який сигнал буде згенерований після виконання таких операторів?

```
initial
begin
  clock=1'b0;
  forever #20 clock = ~clock;
end
```

28. Яким буде результат виконання оператора, якщо num = 2?

```
a = repeat(num) @(posedge clk) data;
```

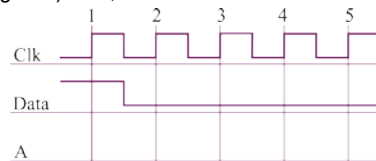


Рис. 5.3. Часова діаграма

29. Яким буде результат виконання оператора, якщо num = 2?

```
a = repeat(num) @(clk) data;
```

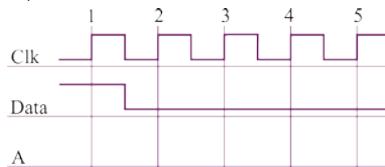


Рис. 5.4. Часова діаграма

6. ПІДПРОГРАМИ

6.1. Підпрограми Task Function

Як і в будь-якій іншій мові програмування, підпрограми дозволяють розбивати великий код на окремі фрагменти. Verilog має у своєму розпорядженні два типи підпрограм: завдання (task) і функції (function) [5, 6, 9]. Основна мета функції - згенерувати значення, що може бути використане у виразі. Функція завжди повертає одне значення й повинна мати хоча б один вхідний аргумент. Завдання може формувати кілька значень вихідних змінних або взагалі жодного, може не мати вхідних аргументів. У завданнях допускається застосування затримок, тимчасового й подієвого керування, у функціях - ні. У функціях не повинно бути операторів, які починаються із символів #, @ або wait. У підпрограмах для декларації аргументів використовуються такі ж ключові слова, як і для портів модуля. Синтаксис:

//Синтаксис опису функції

```
function [ automatic ] [ signed ] [ range_or_type ] identifier ;
    function_item_declaration { function_item_declaration }
    function_statement
endfunction
```

```
function [automatic] [signed] [range_or_type] identifier (function_port_list ) ;
    block_item_declaration { block_item_declaration }
    function_statement
endfunction
```

// виклик функції

```
function_identifier ( expression { , expression } )
```

//Синтаксис опису завдання

```
task [ automatic ] task_identifier ;
    { task_item_declaration }
    statement
endtask
```

```
task [ automatic ] task_identifier ( task_port_list ) ;
    { block_item_declaration }
    statement
endtask
```

```
task identifier;
    parameter_declaration;
    input_declaration;
    output_declaration;
    inout_declaration;
    register_declaration;
```

```

event_declaration;
    statement;
endtask

```

```
// Виклик завдання
```

```
task_identifier [ ( expression { , expression } ) ] ;
```

Параметр identifier описує ім'я функції або завдання. Для функції - це також ім'я значення, що повертається, для якого параметром range_of_type задається тип і розмір. Якщо останній (розмір) пропущений, то за замовчуванням функція повертає однобітове значення типу reg. Можна також присвоїти значенню, що повертається, тип: integer, real, realtime або time. Аргументи задач, оголошені як inout і out, можуть використовуватися тільки в процедурних операторах і повинні бути: змінною регістрового класу (reg, integer, real, realtime або time), пам'яттю, конкатенацією регістрів змінних пам'яті, елементом або діапазоном регістрового вектора.

Входи функції можуть бути описані одним із двох способів. У першому випадку після ім'я функції ставиться крапка з комою. Після цього йде опис одного або декількох вхідних портів. У другому випадку опис входів записується в круглих дужках, через кому після ім'я функції. Приклад визначає функцію getbyte, з використанням першого (лістинг 6.1 а)) або другого (лістинг 6.1 б)) способу опису вхідних параметрів. Аналогічним чином визначаються вхідні параметри і для завдань. Лістинг 6.2 ілюструє обидва способи визначення завдань на прикладі завдання my_task, що має п'ять аргументів: два вхідних, два вихідних і один двонаправлений.

Лістинг 6.1. Приклади опису вхідних параметрів функції.

```

a) function [7:0] getbyte;
    input [15:0] address;
    begin
        ...
        getbyte = result_expression;
    end
endfunction

```

```

б) function [7:0] getbyte (input [15:0] address);
    begin
        ...
        getbyte = result_expression;
    end
endfunction

```

Лістинг 6.2. Приклади опису вхідних параметрів завдань.

```

a) task my_task;
    input a, b;
    inout c;
    output d, e;
begin

```



```

... // оператори, що визначають роботу завдання
    c = foo1; // присвоєння результуючих значень
    d = foo2;
    e = foo3;
end
endtask
б) task my_task (input a, b, inout c, output d, e);
    begin
        ... // оператори, що визначають роботу завдання. . .
        c = foo1; // присвоєння результуючих значень
        d = foo2;
        e = foo3;
    end
endtask

```

Обидва види підпрограм визначаються в модулі і є локальними для нього. У табл. 6.1 наведена порівняльна характеристика властивостей функцій і завдань. Підпрограми можуть мати локальні змінні, але не допускається використання змінних класу ланцюга. Реалізуються вони тільки за допомогою поведінкових операторів. Однак вони не можуть містити в собі конструкції `always` і `initial`, але здатні викликатися з них або інших підпрограм (функції не можуть викликати завдання).

Таблиця 6.1. Властивості функцій і завдань

	Функція	Завдання
1	Може викликати іншу функцію, завдання - ні	Може викликати як завдання, так і функції
2	Завжди виконується миттєво	На виконання може знадобитися ненульовий проміжок часу
3	Не може містити часові або подієві оператори управління	Може містити часові або подієві оператори управління
4	Має тільки вхідні аргументи (input)	Може мати аргументи типу input , output и inout
5	Завжди повертає єдине значення	Не повертає значення, але може передавати через аргументи типу output и inout

У наступному виразі одна з гілок умовного оператора містить конкатенацію двох викликів однієї функції:

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```

Лістинг 6.3 являє собою приклад використання в завданні параметрів, подій, вхідних, вихідних і двонаправлених аргументів. При виклику завдання перший аргумент повинен бути значенням типу `integer`, другий - 4-розрядною змінною типу `reg`, третій - однобітовою змінною типу `reg`. Подія `e`, визначена в завданні `first_task`, може бути доступна через ієрархічне ім'я `first_task.e`.

Лістинг 6.4 містить приклад створення функції для обчислення речовинних значень. Ще один приклад (лістинг 6.5) ілюструє використання функцій для

створення моделі світлофора. Функція управляє часом між перемиканнями квітів світлофора.

Лістинг 6.3. Приклад створення й використання завдання.

```
task first_task;
    parameter size=4;
    input a;   integer a;           // Вхідний аргумент
    inout [size-1:0] b;           // Двонаправлений аргумент
    output c;   // Вихідний аргумент
    reg [size-1:0] d;             // Внутрішня змінна
    event e;           // Декларація події

begin
    d = b;
    c = |d;
    b = ~b;
    if (!a) -> e;

end
endtask
...
integer x;
reg a, b, y;
reg [3:0] z;
reg [7:0] w;
...
first_task(x, z, y);           // x – змінна integer, z – 4-розрядний вектор,
                               // y - 1-бітовий reg
first_task(x, w[7:4], w[1]);   // x – змінна integer, 4-розрядний діапазон
                               // вектора w, w[1] - 1-бітовий reg
first_task(1,{a,b,w[3], x[0]}, y);
                               // константа типу integer, оператор конкатенації
                               // формує- 4-розрядний вектор, x[0] y - 1-бітовий reg
always @(first_task.e)       // Використання іменної події із завдання
    count= count+1;
```

Лістинг 6.4. Приклад використання функції.

```
function real multiply;
    input a, b;
    real a, b;
    multiply = ((1.2 * a) * (b * 0.17)) * 5.1;
endfunction
//...
real a;
initial begin
    a = multiply(1.5, a);
end
```

Лістинг 6.5. Використання завдання для моделі світлофора.

```
module traffic_lights;
```

```

        reg clock, red, amber, green;
        parameter on = 1, off = 0, red_tics = 350;
        parameter amber_tics = 30, green_tics = 200;
// Ініціалізація значень кольору
initial red = off;
initial amber = off;
initial green = off;
always begin // послідовність керування світлофором
    red = on; // включення червоного кольору
    light(red, red_tics); // очікування
    green = on; // включення зеленого кольору
    light(green, green_tics); // очікування
    amber = on; // включення жовтого кольору
    light(amber, amber_tics); // очікування
end
// Формується затримка на tics позитивних фронтів синхросигналу
// до включення світла color
task light;
    output color;
    input [31:0] tics;
begin
    repeat (tics) @ (posedge clock);
    color = off; // вимикання світла
end
endtask
always begin // формування сигналу синхронізації
    #100 clock = 0;
    #100 clock = 1;
end
endmodule

```

6.2. Автоматичні функції й завдання

Завдання й функції без ключового слова `automatic` реалізуються як статичні. Таким чином, всі елементи таких підпрограм оголошені статичними і є загальними для всіх викликів таких завдань і функцій.

В автоматичних завданнях і функціях (оголошених із ключовим словом `automatic`) всі елементи є динамічними й створюються при кожному виклику підпрограми. Автоматичні функції дозволяють реалізовувати рекурсію. Елементи автоматичних функцій і завдань не можуть бути доступні через ієрархічне ім'я.

Для автоматичних завдань не дозволяється використовувати: 1) неблокуючі оператори; 2) безперервні процедурні оператори `assign` і `force`; 3) внутрішньооператорний контроль; 4) сканування об'єктів за допомогою системного завдання `$monitor`.

Лістинг 6.6 представляє приклад використання автоматичної функції `factorial` для обчислення факторіала з поверненням значення в 32-бітовому регістрі.

Лістинг 6.6. Приклад автоматичної функції.

```

module tryfact;
    // опис функції
function automatic [31:0] factorial;
    input [3:0] operand;
    reg [3:0] i;

begin
    factorial = 1;
    for (i = 2; i <= operand; i = i + 1)
        factorial = i * factorial;

end
endfunction

//використання функції
integer result;
integer n;

initial
begin
    for (n = 0; n <= 7; n = n+1)
    begin
        result = factorial(n);
        $display("%0d factorial=%0d", n, result);
    end
end
endmodule

// результати використання системного завдання $display
0      factorial=1
1      factorial=1
2      factorial=2
3      factorial=6
4      factorial=24
5      factorial=120
6      factorial=720
7      factorial=5040

```

6.3. Системні завдання роботи з файлами

6.3.1. Запис інформації у файл.

Verilog пропонує можливість виконувати роботу з файлами із застосуванням системних завдань. Відкрити й закрити файл можна за допомогою \$fopen і \$fclose. Відкритий файл може бути використаний для завдань виводу результатів і моніторингу: \$fdisplay і \$fmonitor, відповідно. Синтаксис:

```

integer $fopen("filename");
$fclose(multi_channel_descriptor);
file_output_task_names(multi_channel_descriptor, list_of_arguments);
file_output_task_name ::=
$fdisplay | $fdisplayb | $fdisplayh | $fdisplayo |

```

\$fwrite | \$fwriteb | \$fwriteh | \$fwriteo |
 \$fstrobe | \$fstrobeb | \$fstrobeh | \$fstrobeo |
 \$fmonitor | \$fmonitorb | \$fmonitorh | \$fmonitorto

Функція \$fopen повертає 32-бітовий дескриптор файла multi_channel_descriptor, що має цілочисельний тип. Якщо файл неможливо відкрити для запису, то функція \$fopen повертає 0.

Для тестування суматора ha_1 розроблений testbench (лістинг 6.7). Результат моделювання записується у файл ha_f_rslt.txt. Файл відкривається й зв'язується з ім'ям "info". Пізніше завдання \$fmonitor записує значення описаних змінних у відкритий файл. При завершенні моделювання файл закривається автоматично. Можна закрити файл автоматично через процедуру \$close(info).

Лістинг 6.7. Приклад виводу результатів моніторингу у файл

```
timescale 1ns/1ps
module ha_1
    (input a,b,
     output s, ca);
xor #(1,2) (s, a, b);
and #(3,4) (ca, a, b);
endmodule

// test-bench
module tsthaf();
    integer info;
    reg a,b;
    wire s,ca;
    ha_1 hh(.s(s), .ca(ca), .a(a), .b(b));
    initial begin
        a = 0; b = 0;
        info = $fopen("ha_f_rslt.txt");
    end
    always begin
        #5 a = 1; b = 0;
        #5 a = 0; b = 1;
        #5 a = 1; b = 1;
        #5 a = 0; b = 0;
    end
end
initial
    $fmonitor(info, $time, "a = %b, b = %b, out carry = %b,
        outsum = %b", a, b, ca, s);
    initial begin
        #30 $display(info);
        #0 $stop;
    end
end
endmodule
```

Результати моніторингу, отримані при моделюванні напівсуматора (див. лістинг 6.1) і записані у файл ha_f_rslt.txt:

```

0a = 0, b = 0, out carry = x, outsum = x
2a = 0, b = 0, out carry = x, outsum = 0
4a = 0, b = 0, out carry = 0, outsum = 0
5a = 1, b = 0, out carry = 0, outsum = 0
6a = 1, b = 0, out carry = 0, outsum = 1
10a = 0, b = 1, out carry = 0, outsum = 1
15a = 1, b = 1, out carry = 0, outsum = 1
17a = 1, b = 1, out carry = 0, outsum = 0
18a = 1, b = 1, out carry = 1, outsum = 0
20a = 0, b = 0, out carry = 1, outsum = 0
24a = 0, b = 0, out carry = 0, outsum = 0
25a = 1, b = 0, out carry = 0, outsum = 0
26a = 1, b = 0, out carry = 0, outsum = 1

```

6.3.2. Введення інформації у пам'ять.

Єдиною формою введення інформації з файла, яка підтримується IEEE 1364 Verilog HDL, є системні завдання \$readmemb і \$readmemh, що зчитують інформацію прямо в пам'ять. Завдання \$readmemb і \$readmemh працюють із двійковим і шістнадцятковим форматом подання даних, відповідно. Синтаксис:

```

$readmemh("file_nam", memory_name, [start_addr, end_addr]);
$readmemb("file_nam", memory_name, [start_addr, end_addr]);

```

де filename - строкове ім'я файла; memname - описує Verilog-ідентифікатор пам'яті для завантаження даних; start_addr - необов'язкова початкова адреса для запису даних. Якщо не зазначена, то використовується ліва адреса пам'яті. Якщо зазначена <finish_addr>, то занесення даних в пам'ять закінчується в цій позиції.

Формат даних у файлі може бути двійковим або шістнадцятковим. Довжина та система числення не вказується. Числа розділяються пробілами. Дозволяється використовувати Verilog- коментарі.

Лістинг 6.8 представляє типовий приклад використання функції \$readmemh для зчитування тестових послідовностей з файла mem.dat для верифікації напівсуматора ha_1.

Лістинг 6.8. Приклад читання даних з файла за допомогою \$readmemh.

```

module test_ha_1;
    reg [1:0] mem [1:16];
    integer i;
    reg a, b;
    wire s,ca;
    ha_1 hh(.s(s), .ca(ca), .a(a), .b(b));
    initial
        $readmemh("mem.dat", mem);
    initial begin
        $monitor("time=%d a = %b, b = %b, out carry = %b,
            outsum = %b", $time, a, b, ca, s);
        for(i=1; i<=16; i = i+1)
            #5 {a, b} = mem[i];
    end
endmodule

```

Початкова адреса для запису інформації в пам'ять може бути також вказана у файлі у форматі "@hhh...h". Дані завантажуються послідовно, починаючи із зазначеної адреси. Використовується тільки **шістнадцятковий** формат для подання адреси. Файл може містити кілька початкових адрес, що дозволяє завантажувати в пам'ять декілька підблоків.

6.4. Контрольні завдання

1. Створити функцію, що обчислює факторіал 4-бітового числа. Результат - 32-бітове число. Написати тестовий модуль для виклику функції й перевірки результату її роботи.
2. Створити функцію, що множить два 4-бітових числа та формує 32-бітовий результат. Написати тестовий модуль для виклику функції й перевірки результату її роботи.
3. Створити функцію, що моделює восьмиопераційне АЛП. Функція приймає 3-бітовий код операції, два 4-розрядних операнда, а повертає 5-бітовий результат. Написати тестовий модуль для виклику функції й перевірки результату її роботи.

Таблиця 6.2. Приклади функцій

ОР	Функція
3'b000	out = a
3'b001	out = a + b
3'b010	out = a - b
3'b011	out = a / b
3'b100	out = a % b
3'b101	out = a << b
3'b110	out = a >> b
3'b111	out = (a > b), порівняння амплітуд

4. Розробити функцію, що обчислює скалярний добуток двох векторів A і B за формулою $C = \sum a_i \cdot b_i$. Наприклад:

$$A[3:1] = (1, 2, 3); B[3:1] = (4, 5, 6);$$

$$C = 3 \cdot 6 + 2 \cdot 5 + 1 \cdot 4 = 32.$$

Виклик функції повинен мати вигляд DOT(A,B), де A і B - масиви елементів типу integer. Результат обчислення, що повертається функцією, також має тип integer.

5. Для функції із завдання 6.4 розробити testbench, у якому визначені два масиви A і B, які використовуються як тестові послідовності. Масиви ініціалізуються значеннями з файла "input_data.dat", інформація в якому представлена в шістнадцятковій формі.

6. Створити завдання, що обчислює факторіал 4-бітового числа. Результат - 32-бітове число призначається після затримки в 10 одиниць часу. Написати тестовий модуль для виклику завдання й перевірки результату роботи.

7. Створити завдання, що виконує перевірку на парність 16-бітового числа. Результат - 31-бітове число. На нього надходить значення після трьох позитивних фронтів синхронізації. Використовувати оператор repeat.

8. а) Написати завдання, що обчислює суму квадратів вхідних значень A і B. Вхідні значення мають тип real. Завдання повертає значення через речовинну змінну sum_of_squares.

б) Розробити testbench для тестування роботи завдання з п. а). Тестові значення формувати за допомогою системної функції \$random. Результати моделювання виводити в текстовий файл у форматі:

A = значення B = значення sum_of_squares = значення.

Наприклад:

A = 3 B = 2 sum_of_squares = 13.

9. Написати завдання, що генерує імпульси синхросигналу syn_clk довжиною 1 од. часу з періодом в 20 од. часу. Коли сигнал reset = 1, то завдання перемикає syn_clk в 0 і повертає керування.

7. ГЕНЕРУВАННЯ ОПЕРАТОРІВ

7.1. Оператор generate і змінні genvar

Оператор generate спрощує процес створення параметризованих моделей. Є можливість перевірки декларації змінних, функцій і завдань, а також створення копій модулів і примітивів, операторів безперервного присвоєння, блоків initial і always. При цьому також може бути використана умовна форма даної конструкції. Кожна копія, що генерується, має унікальне ім'я й може бути доступна через ієрархічне ім'я [5, 6, 9].

Для індексації оператора generate використовується індексна змінна genvar. Її синтаксис:

```
genvar list_of_genvar_identifiers ;
```

де list_of_genvar_identifiers - список змінних genvar.

Змінна декларується в модулі, у якому використовується. Допускається її визначення в операторі generate і за його межами. В останньому випадку вона може бути використана декількома блоками generate. Має цілочисельний тип і може приймати значення більші або рівні 0. Перше присвоєння значення змінної genvar не повинне відноситися до правої границі змінної. Якщо два оператори generate використовують ту саму змінну як індекс, то вони не можуть бути вкладеними.

Параметри genvars існують тільки під час підготовки проекту до моделювання, під час моделювання вони недоступні.

Для підтримки з'єднань між структурними елементами та/або процедурними блоками в операторі generate допускається виконувати декларування змінних таких типів даних: net, reg, integer, real, time, realtime і event.

Приклад використання оператора generate представлений лістингом 7.1. Модуль xorGen містить параметри, що визначають число розрядів у пристрої width і величину затримки delay. Для керування оператором generate визначена змінна i, яка змінюється в діапазоні від 1 до величини width, за замовчуванням рівної 4.

Лістинг 7.1. Генерування копій операторів assign.

```
module xorGen
    #(parameter width = 4, delay =10)
    (output [1:width] xout,
    input [1:width] xin1, xin2);
generate
    genvar i;
    for (i = 1; i <= width; i=i + 1) begin: xi
        assign #delay xout[i] = xin1[i] ^ xin2[i];
    end
endgenerate
endmodule
```

Для розглянутого приклада (див. лістинг 7.1) генеруються такі чотири копії операторів безперервного присвоєння:

```
assign #delay xout[1] = xin1[1] ^ xin2[1];
assign #delay xout[2] = xin1[2] ^ xin2[2];
assign #delay xout[3] = xin1[3] ^ xin2[3];
assign #delay xout[4] = xin1[4] ^ xin2[4];
```

Оскільки оператор generate виконується під час елаборації, ці чотири оператори стають частиною модуля xorGen, замінюючи конструкцію generate ... endgenerate...

7.2. Оператор generate із циклом for

Оператори, які можуть використовуватися в конструкції generate, обмежуються for, if-then-else і case. Синтаксис оператора generate із циклом for:

```
generate {
    for ( genvar_assignment ; constant_expression ; genvar_assignment )
    begin: generate_block_identifier
    { generate_item }
    end
} endgenerate
```

де genvar_assignment - початкове значення змінної genvar, constant_expression - умова завершення циклу, genvar_assignment - крок циклу, generate_item – оператори, що генеруються. Вкладений блок begin-end має обов'язкову унікальну мітку generate_block_identifier і може бути доступний через ієрархічне ім'я. Фігурні дужки означають можливість присутності декількох копій конструкцій, що обмежуються ними.

Оператор generate може бути використаний для створення копій логічних елементів. Кожна копія отримує індивідуальне ім'я, яке має як суфікс строкове подання значення індексу genvar. Наприклад, код моделі пристрою, розглянутого в лістингу 7.1, з реалізацією на вентилях xor може виглядати так:

```
generate
    genvar i;
    for (i = 1; i <= width; i=i+1) begin: xi
    xor #delay a (xout[i], xin1[i], xin2[i]);
    end
endgenerate
```

Аналогічно можна створювати копії оператора always:

```
generate
    genvar i;
    for (i = 1; i <= width; i=i+1) begin: xi
    always @(*)
    xout[i] = xin1[i] ^ xin2[i];
    end
endgenerate
```

У результаті буде згенеровано 4 блоки always з індексами від 1 до 4.

Лістинг 7.2, а), б) представляє ще один приклад застосування оператора generate із циклом for для створення моделі перетворювача коду Грея у двійковий послідовний код gray2bini. Модулі параметризуються параметром size, за замовчуванням рівним 8.

Лістинг 7.2. Перетворювач коду Грея у двійковий послідовний код.

а) // із застосуванням оператора безперервного присвоєння

```
module gray2bin1
    #(parameter SIZE = 8)
    (output [ SIZE-1:0] bin, input [ SIZE-1:0] gray);
    genvar i;
    generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[ SIZE-1-i];
    end endgenerate
endmodule
```

б) // із застосуванням оператора always

```
module gray2bin2
    #(parameter SIZE = 8)
    (output [ SIZE-1:0] bin, input [ SIZE-1:0] gray);
    reg [ SIZE-1:0] bin;
    genvar i;
    generate for (i=0; i<SIZE; i=i+1) begin:bit
        always @(gray[ SIZE-1-i]) // fixed part select
            bin[i] = ^gray[ SIZE-1-i];
    end endgenerate
endmodule
```

7.3. Умовний оператор generate

Умовний оператор generate використовує конструкцію if-else-if, що дозволяє застосовувати умови, які обчислюються на етапі елаборації, для створення копій модулів, стандартних і користувацьких примітивів, безперервних операторів присвоєння, блоків. Синтаксис:

```
generate
    if ( constant_expression )
        generate_item_or_null
    [else generate_item_or_null ]
endgenerate
```

де constant_expression - умова генерування операторів.

В моделі пристрою множення multiplier, що параметризується, яка представлена лістингом 7.3, параметри a_width і b_width визначають розрядність операндів. Якщо один з параметрів має значення менше 8 (бітів), створюється пристрій множення CLA (carry look-ahead adder - суматор з попереднім переглядом у схемі прискореного переносу). Якщо обидва параметри a_width і b_width більші або дорівнюють 8 (бітам), то використовується пристрій

множення Wallace tree (Дерево Уоллеса). Пристрій, що генерується, має мітку u1.

Лістинг 7.3. Пристрій множення, що параметризується.

```
module multiplier
    #(parameter a_width = 8, b_width = 8)
    (input [a_width-1:0] a, [b_width-1:0] b,
     output [a_width+b_width-1:0] product);
generate
    if((a_width < 8) || (b_width < 8))
        CLA_multiplier #(a_width,b_width) u1(a, b, product);
    else
        WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
endgenerate
endmodule
```

7.4. Оператор generate з оператором case

Для умовного вибору в операторі generate може бути також використана конструкція case, як це зроблено в прикладі (лістинг 7.4), що генерує різні копії суматорів, залежно від параметра WIDTH:

Лістинг 7.4. Використання оператора case у конструкції generate.

```
generate
    case (WIDTH)
        1: adder_1bit x1(co, sum, a, b, ci); // 1-бітовий суматор
        2: adder_2bit x1(co, sum, a, b, ci); // 2-бітовий суматор
        default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
        // суматор CLA
    endcase
endgenerate
```

Синтаксис:

```
generate
    case ( constant_expression )
        genvar_case_item { genvar_case_item } endcase
        genvar_case_item ::= constant_expression { , constant_expression } :
        generate_item_or_null | default [ : ] generate_item_or_null
    endgenerate
```

Для перевизначення значення параметрів використовується формат ordered або named parameter = value assignment or defparam, що може бути також оголошений у межах оператора generate.

У декларативній частині оператора (не в циклі) generate можуть бути описані завдання й функції. При цьому вони повинні мати унікальне ім'я й можуть бути доступні через ієрархічне ім'я. Оператор generate може бути створений з використанням одного із трьох методів: generate-loop, generate-conditional or generate-case.

7.5. Приклади використання оператора generate

7.5.1. Приклад суматора з наскрізним переносом. Приклад представляє реалізацію вентиляної моделі суматора з наскрізним переносом (лістинг 7.5). Оператором generate виконується генерування копій імен для вентилів хог: bit[0].g1, bit[1].g1, bit[2].g1, bit[3].g1, bit[0].g2 bit[1].g2 bit[2].g2, bit[3].g2; для вентилів and: bit[0].g3, bit[1].g3, bit[2].g3, bit[3].g3, bit[0].g4, bit[1].g4, bit[2].g4, bit[3].g4; для вентилів or: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5. Згенеровані копії вентилів з'єднуються за допомогою багатомірного масиву t типу wire, визначеного поза оператором generate (лістинг 7.5 а)). У прикладі з лістинга 7.5 б) для зв'язку вентилів використовуються три лінії t1, t2, t3, декларація яких виконана усередині оператора generate. Для кожної ітерації циклу створюються власні копії цих змінних: bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1

```
// bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
```

```
// bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
```

Лістинг 7.5. Дві моделі суматора з наскрізним переносом.

а) module addergen1

```
    #(parameter SIZE = 4)
    (output [ SIZE-1:0] sum, output co,
    input [ SIZE-1:0] a, b, input ci);
    wire [SIZE :0] c;
    wire [ SIZE-1:0] t [1:3];
    genvar i;
    assign c[0] = ci;
    generate
        for(i=0; i<SIZE; i=i+1) begin:bit
            xor g1 ( t[1][i], a[i], b[i]);
            xor g2 ( sum[i], t[1][i], c[i]);
            and g3 ( t[2][i], a[i], b[i]);
            and g4 ( t[3][i], t[1][i], c[i]);
            or g5 ( c[i+1], t[2][i], t[3][i]);

        end
    endgenerate
    assign co = c[SIZE];
```

endmodule

б) module addergen2

```
    #(parameter SIZE = 4)
    (output [ SIZE-1:0] sum, output co,
    input [ SIZE-1:0] a, b, input ci);
    wire [SIZE :0] c;
    genvar i;
    assign c[0] = ci;
    generate
        for(i=0; i<SIZE; i=i+1) begin:bit
            xor g1 ( t1, a[i], b[i]);
            xor g2 ( sum[i], t1, c[i]);
            and g3 ( t2, a[i], b[i]);
```

```

                                and g4 ( t3, t1, c[i]);
                                or g5 ( c[i+1], t2, t3);
                                end
                                endgenerate
                                assign co = c[SIZE];
                                endmodule

```

7.5.2. Приклад суматора.

Нижче представлений приклад моделювання n-бітового суматора (де $n > 1$), що також має умовні виходи, які позначають негативний результат, перенос або переповнення додаткового коду (лістинг 7.6). У цьому випадку не генеруються копії суматора. Оператори if-then-else і case можуть бути використані для генерування різних конфігурацій. Лістинг 7.6 представляє модуль, що використовує оператор case для створення різних конфігурацій схеми суматора adder залежно від того, які біти будуть генеруватися. Три різних ситуації обробляються окремо. Для більшості станів carry in з'єднується з carry out попереднього стану. При обчисленні молодшого біта використовується перенос у пристрій (cIn). Для самого старшого розряду (визначеного параметром width) повинні бути реалізовані: перенос у старший розряд (cOut), overFlow і neg.

Лістинг 7.6. Модель повного суматора.

```

module adderWithConditionCodes
    #(parameter width = 1)
    (output reg [ width-1:0] sum,
    output reg cOut, neg, overFlow,
    input [ width-1:0] a, b,
    input cIn);
    reg [width -1:0] c;
    generate
    genvar i;
    for (i = 0; 1<= width-1; i=i+1) begin: stage
        case(i)
            0: begin
                always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ cIn;
                    c[i] = a[i]&b[i] | b[i]&cIn | a[i] & cIn;
                end
            end
            width-1: begin
                always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ c[ i-1];
                    cOut = a[i]&b[i] | b[i]&c[ i-1] | a[i] & c[ i-1];
                    neg = sum[i];
                    overFlow = cOut^ c[ i-1];
                end
            end
        endcase
    end
end

```

```

        default: begin
            always @(*) begin
                sum[i] = a[i] ^ b[i] ^ c[i-1];
                c[i] = a[i]&b[i] | b[i]&c[i-1] | a[i] &c[i-1];
            end
        end
    endcase
end
endgenerate
endmodule

```

7.5.3. Використання вкладених операторів generate.

Модель, представлена лістингом 7.7, ілюструє використання вкладених циклічних операторів, циклічних і умовних операторів generate.

Лістинг 7.7. Багаторівневий оператор generate.

```

parameter SIZE = 2;
genvar i, j, k, m;
generate
    for (i=0; i<SIZE+1; i=i+1) begin:B1 // область дії B1[i]
        M1 N1(); // копії B1[i].N1[i] компонента M1
        for (j=0; j<SIZE; j=j+1) begin:B2 // область дії B1[i].B2[j]
            M2 N2(); // копії B1[i].B2[j].N2 компонента M2
            for (k=0; k<SIZE; k=k+1) begin:B3
                // область дії B1[i].B2[j].B3[k]
                // копії B1[i].B2[j].B3[k].N3
                M3 N3();
            end
        end
    end
    if (i>0)
        for (m=0; m<SIZE; m=m+1) begin:B4
            // область дії B1[i].B4[m]
            M4 N4(); // копії B1[i].B4[m].N4
        end
    end
endgenerate

```

Нижче представлені деякі імена копій компонентів, що генеруються, у багаторівневому операторі generate:

B1[0].N1	B1[1].N1
B1[0].B2[0].N2	B1[0].B2[1].N2
B1[0].B2[0].B3[0].N3	B1[0].B2[0].B3[1].N3
B1[0].B2[1].B3[0].N3	
B1[1].B4[0].N4	B1[1].B4[1].N4

7.6. Контрольні завдання

1. Розробити структурну модель 32-бітного двонаправленого трансівера, що використовує як компонент функціональну модель 1-бітного трансівера, який має два двонаправлених входи даних - а і b.

Асинхронний, активний по низькому рівню порт дозволу виходів oe_n . Порт напрямку dir . Коли $oe\_n = 0$ і $dir = 0$, дані передаються з b в a ; якщо $oe\_n = 0$ і $dir = 1$, то дані передаються з a на b . Якщо $oe\_n = 1$, то a і b мають значення високого імпедансу.

2. Намалювати схему, що відповідає такому оператору generate:

```
generate
    genvar stage
    for (stage = 1; stage <= 4; stage = stage + 1) begin: Ui
        d_ff delay_ff (.clk(sys_clk),
            d(delayed_data[stage-1]), .q(delayed_data[stage]));
    end
endgenerate
```

3. Створити умовний оператор generate, що з'єднає змінну $external\_clk$ із сигналом $internal\_clk$, якщо параметр $positive\_clk = 1$. У протилежному випадку на лінію $internal\_clk$ подається інверсне значення $external\_clk$.

4. Розробити структурну модель пристрою, що складається з n мультиплексорів 2-в-1, з'єднаних паралельно (рис. 7.1). Входами пристрою є сигнал керування Sel , два n -бітових вектори A і B . Вихід описаний n -бітовим сектором C . Величина n визначена за допомогою директиви 'define.

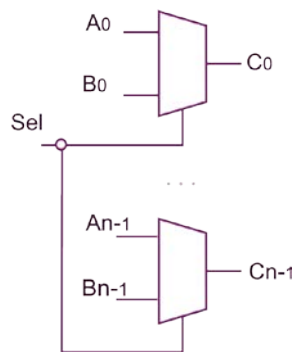


Рис. 7.1. Мультиплексори 2-в-1, з'єднані паралельно

5. Розробити структурну модель двійкового лічильника з наскрізним переносом, яка реалізована на D-тригерах (рис. 7.2). Кількість компонентів задається через параметр n , що за замовчуванням дорівнює 4.

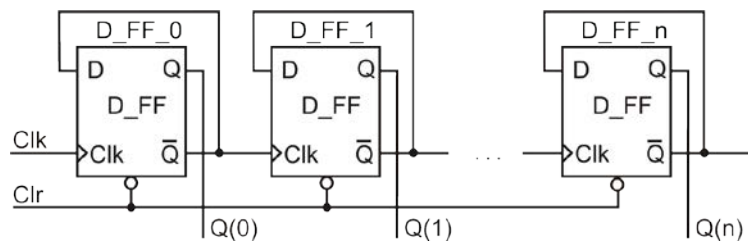


Рис. 7.2. Двійковий лічильник з наскрізним переносом, реалізований на D-тригерах

8. PROGRAMMING LANGUAGE INTERFACE (PLI)

Програмний інтерфейс PLI (рис. 8.1) використовується для [7, 8]:

- створення додаткових системних завдань і функцій: завдань моніторингу, генерації тестів, налагодження фрагментів програми;
- побудови різних програмних додатків: транслятори, аналіз тимчасових параметрів;
- одержання інформації про проект: ієрархія, зв'язки;
- спеціальних і користувацьких форм керування виводом інформації;
- створення підпрограм, що формують тестові послідовності;
- створення будь-якого заснованого на Verilog програмного додатка.



Рис. 8.1. PLI-Інтерфейс

PLI дозволяє користувачеві читати й модифікувати структури внутрішніх даних, отримувати доступ до середовища моделювання.

Проектувальник пише власне користувацьке системне завдання, застосовуючи PLI-підпрограми, дод. А. Воно підключається до програми моделювання, що обробляє посилання на завдання, знаючи місце розташування відповідної функції мовою Сі.

Для ілюстрації механізму використання PLI представлений приклад створення елементарного користувацького системного завдання \$welcome_task і функції \$my_function. При виклику дане завдання виводить повідомлення "You are welcome!".

Спочатку створюється підпрограма мовою Сі з використанням бібліотеки PLI:

```
#include "veriusertfs.h"
void welcome_task()
{
    io_printf("You are welcome!\n");
}
int function_call()
{
    io_printf("This PLI function works !\n");
    return 0;
}
```

Для того щоб програма моделювання знала про існування підпрограми `welcome_task` і могла викликати її, зустрівши конструкцію `$welcome_task`, необхідно виконати підключення PLI-підпрограми до Verilog-симулятора. Різні програми моделювання можуть мати різний механізм лінковки PLI. Симулятори Aldec Active-HDL, Cadence Verilog-XL™, MTI ModelSim® пропонують механізм реєстрації системних завдань і функцій з використанням масиву `veriusertfs`. Його синтаксис:

```
s_tfcell veriusertfs[ ] =
{
    {usertask | userfunction | userrealfunction, data; checktf; sizetf; calltf;
    mistcf; "$tfname"},
{0}
}
```

Перший параметр визначає, чи буде це користувальницька функція (`userfunction`, `userrealfunction`) або завдання (`usertask`). Різні функції зворотного виклику (`checktf`, `sizetf`, `calltf`, and `mistcf`) докладно описані в стандарті Verilog IEEE Std 1364-1995. Вони, як правило, є необов'язковими до використання. Програма моделювання залишає тільки функцію `calltf`, що виконується при виклику системної функції або процедури в Verilog-кодi. Функція `userfunction` повертає значення `reg`, а `userrealfunction` - значення типу `real`. Параметр `"$tfname"` - це ім'я системної функції або завдання в Verilog-кодi, що повинне починатися із символу долара (\$). Кожне завдання й функція повинні мати запис у таблиці `veriusertfs`. Останнім обов'язковим елементом є `{0}`.

При підключенні кожного PLI-дodatка програма моделювання шукає експортовану таблицю `veriusertfs`. У випадку, якщо вона знайдена, програма моделювання має можливість викликати певні в ній завдання й функції. Далі представлений приклад реєстрації користувальницького завдання й функції:

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, welcome_task, 0, "$welcome_task"},
    {userfunction, 0, 0, 0, function_call, 0, "$my_function"},
    {0} // останній елемент повинен бути 0
};
```

Представлена вище таблиця, що визначає підключення завдання `welcome_task` і функції `function_call`, повинна бути присутня у Сі-файлі.

Для Microsoft Windows орієнтованих версій програм моделювання PLI-додаток повинен бути відкомпільований й зібраний в 32-розрядну динамічну бібліотеку (DLL). Тому в коді Сі необхідно визначити таку Windows-функцію DllMain():

```
BOOL WINAPI DllMain(HINSTANCE hInstance,DWORD dwReason,
LPVOID lpReserved)
{
    return TRUE;
}
```

а також додати у файл визначення (PliApp.def) для проекту PliApp в MS Visual C++ наступний рядок:

```
EXPORTS
    veriusertfs DATA
```

Повністю код Сі-програми, що ілюструє найпростіший спосіб використання функцій PLI, представлений лістингом 8.1, а приклад застосування в Verilog користувальницького системного завдання й функції - лістингом 8.2.

Лістинг 8.1. Повна версія коду С із файла PliApp.c.

```
#include "veriusert.h"
#include "aldecp1.h"
#include <windows.h>
int welcome_task()
{
    io_printf("You are welcomet!\n");
    return 0;
}
int function_call()
{
    io_printf("This PLI function works !\n");
    return 0;
}
extern "C" __declspec(dllexport) s_tfc11 veriusertfs[ ] =
{
    {usertask, 0, 0, 0, welcome_task, 0, "$welcome_task"},
    {userfunction, 0, 0, 0, function_call, 0, "$my_function"},
    {0}
};
BOOL WINAPI DllMain( HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved )
{ return TRUE;
}
```

Лістинг 8.2. Приклад використання системної функції та завдання.

```
module start_PLI;
initial
begin
    $welcome_task;
    $my_function;
end
endmodule
// Результат моделювання модуля для середовища ALDEC Active-HDL
Simulation has been initialized
Selected Top-Level: start_PLI (start_PLI)
```

```

: You are welcome!
:
: This PLI function works !
:

```

KERNEL: Simulation has finished. There are no more test vectors to simulate.

PLI-додатки повинні включати файл `veriusr.h` при використанні TF-підпрограми й файл `acc_user.h` - для АСС-підпрограм.

8.1. Внутрішнє подання даних у програмі моделювання

Програмою моделювання модуль розглядається як набір елементів, що представляють типи Verilog-об'єктів, і може включати: копії модулів, порти модуля, pin-to-pin шляхи модуля, intermodule-шляхи; модулі верхнього рівня; копії примітивів, термінали примітивів; ланцюги, регістри, параметри, `spesragam`; регістри `integer`, `time`, `real`; тимчасове керування; іменні події.

Кожний тип має відповідний набір, що ідентифікує об'єкти даного типу. Рис. 8.2 відображає концепцію подання даних у симуляторі.

Кожний набір містить всі елементи типів об'єктів модуля. Набори з'єднані між собою двонаправленими зв'язками. Всі внутрішні структури даних для одержання інформації про модуль можуть бути переглянуті за допомогою PLI-підпрограм.

Для ілюстрації подання структур даних використовується модель мультиплексора 2-в-1 (рис. 8.3). Його код міститься в лістингу 8.3. Внутрішнє подання даних мультиплексора зображене на рис. 8.4.

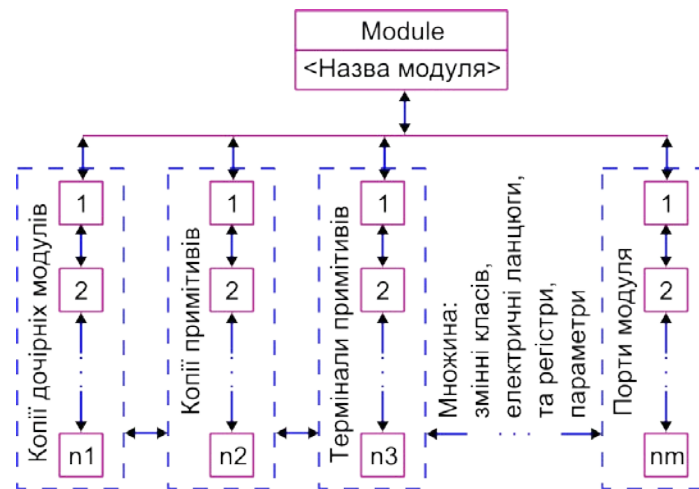


Рис. 8.2. Концепція внутрішнього подання даних

Лістинг 8.3. Модель мультиплексора 2-в-1.

```

module mux2_to_1(out, i0, i1, s);
    output out;
    input i0, i1;

```

```

input s;
wire sbar, y1, y2;
not n1(sbar, s);
and a1(y1, i0, sbar);
and a2(y2, i1, s);
or o1(out, y1, y2);
endmodule

```

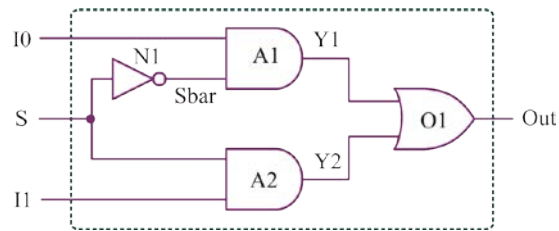


Рис. 8.3. Вентильна схема мультиплексора 2-в-1

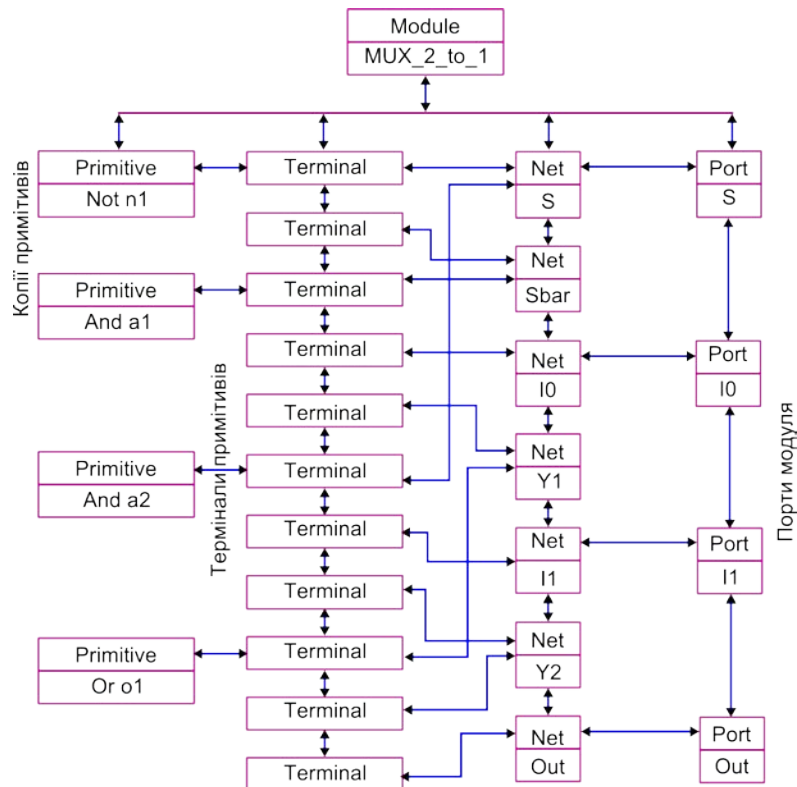


Рис. 8.4. Внутрішнє подання даних для мультиплексора 2-в-1

8.2. Підпрограми бібліотеки PLI

PLI-бібліотека пропонує стандартний інтерфейс для подання внутрішніх даних у проєкті. Існує два класи PLI-підпрограм: доступу (access routine) і обслуговування даних (utility routine) (vpi - не розглядається).

Підпрограми access routine забезпечують доступ до подання внутрішніх структур даних. Вони дозволяють переглядати й витягати необхідну інформацію про проєкт. Обслуговуючі підпрограми (utility routine) головним чином використовуються для переміщення даних через Verilog/Programming Language межу й надають користувачеві різноманітні допоміжні функції. Рис. 8.5 ілюструє роль обох класів PLI-підпрограм.

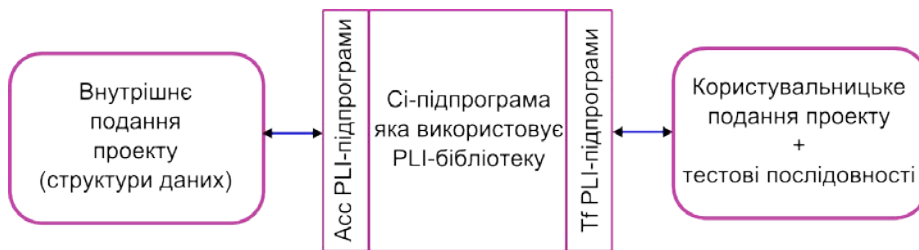


Рис. 8.5. Використання підпрограм доступу й обслуговування

8.3. Підпрограми доступу (access routine)

Підпрограми доступу (access routine), що позначаються асс, можуть зчитувати інформацію про який-небудь об'єкт із внутрішніх структур даних або записувати її туди.

Всі імена починаються із приставки асс_. Сі-підпрограма спочатку виконує ініціалізацію середовища за допомогою підпрограми асс_initialize(). Потім, по завершенню даної фази, необхідно закрити середовище шляхом використання підпрограми асс_close(). Щоб підключити файл із визначеннями підпрограм доступу, код повинен містити рядок `#include "асс_user.h"`.

Підпрограми доступу для звертання до об'єктів використовують поняття абстрактного ідентифікатора (handle). Він призначений для до визначення типів даних, що вказують на окремі об'єкти проєкту. Будь-яка інформація про проєкт може бути доступна, якщо отримано його абстрактний ідентифікатор. Дане поняття подібно до концепції абстрактних ідентифікаторів файлів для роботи з ними із програм. Декларація ідентифікатора об'єкта починається із ключового слова handle:

```
handle top_handle;
```

Існує шість типів підпрограм доступу:

1. Handle routine. Повертає абстрактний ідентифікатор об'єкта в проєкті. Ім'я таких підпрограм завжди починається з асс_handle_.

2. Next routine. Повертає ідентифікатор наступного об'єкта з множини заданого типу об'єктів проекту. Завжди починається з `acc_next_` і отримує посилання на об'єкти як аргументи.
3. Value Change Link (VCL) routine. Дозволяє користувальницькому системному завданню додавати й видаляти об'єкти зі списку спостережуваних. Завжди починається з `acc_vcl_` і не повертає значення.
4. Fetch routine. Витягує різноманітну інформацію про об'єкт: повне ім'я ієрархічного шляху, відносне ім'я. Починається з `acc_fetch_`.
5. Utility access routine. Виконує допоміжні операції для підпрограм доступу. Наприклад, `acc_initialize()` і `acc_close()`.
6. Modify routines. Може змінювати внутрішні структури даних.

8.3.1. Приклади використання підпрограм доступу.

У першому прикладі користувальницьке системне завдання застосовується для читання й підрахунку портів у модулі, а в другому - для моніторингу значення ланцюга.

Приклад 8.1. Спочатку описується користувальницьке системне завдання `$get_ports`, що призначене для пошуку повних ієрархічних імен вхідних, вихідних і двонаправлених портів. Код підпрограми `get_ports` мовою Cі представлений лістингом 8.4.

Для ініціалізації доступу до програми моделювання використовується підпрограма `acc_initialize()`, а `acc_close()` закриває доступ. Підпрограма `acc_handle_tfrag` повертає аргумент, переданий у розроблювальну користувачем функцію `$get_ports`. Для отримання покажчиків на порти модулів використовувалася функція `acc_handle_port`, а `acc_next_port` - для покажчиків на наступні за списком порти. У завданні `acc_fetch_direction`, що повертає напрямки портів, застосовуються відповідні константи `accInput`, `accOutput`, `accInout`. Виведення результатів реалізується за допомогою функції `io_print` TF-класу бібліотеки PLI. Дана функція подібна до функції `printf` і виконує виведення повідомлення у вихідний канал продукту, який її вікликає.

Лістинг 8.5 містить приклад застосування завдання `$get_ports` для аналізу й підрахунку портів заданого модуля.

Лістинг 8.4. Cі-підпрограма для пошуку ієрархічних імен портів.

```
#include "acc_user.h"
int get_ports()
{
    handle mod, port;
    int input_ctr = 0;
    int output_ctr = 0;
    int inout_ctr = 0;
    acc_initialize();

    // Отримання покажчика на копію модуля через перший аргумент mod =
    acc_handle_tfrag(1);

    // Отримання визначника першого порту модуля в списку
    port = acc_handle_port(mod,0);
```

```

while (port!=null) // Цикл перебору всіх портів
{ if (acc_fetch_direction(port) == accInput) // Вхідний порт
  { // Виведення повного ієрархічного ім'я вхідного порту
    io_printf("Input Port %s \n", acc_fetch_fullname(port));
    input_ctr++;
  }
  else if (acc_fetch_direction(port) == accOutput) // Вихідний порт
  { io_printf("Output Port %s \n", acc_fetch_fullname(port));
    output_ctr++;
  }
  // Двонаправлений порт
  else if (acc_fetch_direction(port) == accInout)
  {
    io_printf("Inout Port %s \n", acc_fetch_fullname(port));
    inout_ctr++;
  }
  port = acc_next_port(mod, port); // Перехід до наступного порту
}
io_printf("Input Ports = %d Output Ports = %d, Inout ports = %d\n\n",
          input_ctr, output_ctr, inout_ctr);

acc_close();

return 0;
}

```

...
 // Рядок ініціалізації завдання в структурі veriusertfs[] Сі-коду
 {usertask, 0, 0, 0, get_ports, 0, "\$get_ports"},

Лістинг 8.5. Verilog-модель.

```

module top;
    wire OUT;
    reg I0, I1, S;

    mux2_to_1 my_mux(OUT, I0,I1, S); /*Копія модуля mux2_to_1*/
    initial
    begin
        $get_ports("top.my_mux"); /*Виклик завдання $get_ports, що генерує список
    портів*/
    end
endmodule

```

// Результат моделювання:

```

# Output Port top2.my_mux.out
# Input Port top2.my_mux.i0
# Input Port top2.my_mux.i1
# Input Port top2.my_mux.s
# Input Ports = 3 Output Ports = 1, Inout ports = 0

```

Приклад 8.2. Ілюструє використання підпрограм доступу групи Value Change Link (VCL).

Замість застосування стандартного завдання \$monitor визначене користувацьке завдання \$my_monitor для спостереження за окремими лініями проекту (лістинг 8.6).

Моніторинг реалізується функцією acc_vcl_net, що має такий формат виклику:

```
void acc_vcl_add(object_handle, consumer_routine, user_data, vcl_flag);
    handle object_handle;
    int (*consumer_routine)();
    char * user_data;
    int vcl_flag;
```

де object_handle - визначник об'єкта, що додається в список моніторингу. Користувальницька функція, що викликається при кожному вимірі спостережуваного об'єкта, задається за допомогою покажчика (*consumer_routine)(). У розглянутому прикладі - це функція display_net. Вона виводить ім'я і час зміни сигналу. Повне ім'я спостережуваного об'єкта передається через рядок *user_data. Четвертий параметр у функції acc_vcl_add - це прапор vcl_flag, що може приймати значення vcl_verilog_logic для моніторингу зміни сигналу й vcl_verilog_strength сили сигналу (у даному навчальному посібнику не розглядається).

Лістинг 8.6. Сі-код користувацького завдання моніторингу значень сигналів.

```
#include "acc_user.h"
char convert_to_char();
int display_net();
int my_monitor()
{
    handle net;
    char *netname;           // покажчик для зберігання імені лінії
    char *malloc();
    acc_initialize();         // ініціалізація середовища*/
    net = acc_handle_tfarg(1); // Отримання ідентифікатора
                             // спостережуваної лінії*/
    // Пошук і отримання ієрархічного імені лінії
    netname = malloc(strlen(acc_fetch_fullname(net)));
    strcpy(netname, acc_fetch_fullname(net));
    // Виклик підпрограми VCL для додавання сигналу
    // у список моніторингу
    acc_vcl_add(net, display_net, netname, vcl_verilog_logic);
    acc_close();
    return 0;
}
```

При зміні значення лінії підпрограма acc_vcl_add викликає модуль display_net і передає покажчик на структуру даних типу p_vc_record. Користувальницька підпрограма мовою Сі виконує дію, визначену користувачем при кожному виклику підпрограми acc_vcl_add. Тип p_vc_record визначений у файлі acc_user.h так:

```
typedef struct t_vc_record
{int vc_reason;      // причина зміни значень
  int vc_hightime;   // Старші 32-розряду й
  int vc_lowtime;    // молодші 32-розряду
                      // 64-бітового часу моделювання
char *user_data;     // Рядок, переданий аргументом 3 в acc_vcl_add
  union              // Нове значення спостережуваного сигналу
  {unsigned char      logic_value;
    double            real_value;
    handle            vector_handle;
    s_strengths       strengths_s;
  } out_value;
} *p_vc_record;
```

У даному прикладі користувальницька підпрограма `display_net` просто виводить час зміни, ім'я і нове значення лінії (лістинг 8.7). Інша підпрограма `convert_to_char` створена для перетворення значень логічних констант в ASCII-символи.

Лістинг 8.7. Сі-код додаткових функцій моніторинга.

```
//Користувальницька підпрограма.
//Викликається при кожній зміні спостережуваних сигналів.
display_net(vc_record)
p_vc_record vc_record; //Структура p_vc_record з пакета acc_user.h
{
    // Виводиться час, ім'я й нове значення лінії, що змінилася
    io_printf ("%d New value of net %s is %c \n", vc_record->vc_lowtime,
               vc_record->user_data,
               convert_to_char(vc_record->out_value.logic_value));
}
// Змішані підпрограми для перетворення константи
// визначеного символу в ASCII символ
char convert_to_char(logic_val)
char logic_val;
{
    char temp;
    switch(logic_val)
    {
// vc10, vc11, vc1x визначені в пакеті acc_user.h
    case vcl0: temp = '0';
               break;
    case vcl1: temp = '1';
               break;
    case vcl: temp = 'X';
               break;
    case vcl: temp = 'Z';
               break;
    }
    return(temp);
}
```

Для перевірки створеного користувацького PLI-завдання застосовується модуль мультиплексора 2-в-1 з рис. 8.3. Модуль верхнього рівня, що формує тестові послідовності й використовує завдання \$my_monitor, представлений лістингом 8.8.

Лістинг 8.8. Використання підпрограми моніторингу.

```
module top;
    wire OUT;
    reg I0, I1, S;
    mux2_to_1 my_mux(OUT, I0, I1, S);    // Копія модуля mux2_to_1
    initial                               // Додавання сигналу в список моніторингу
    begin
        $my_monitor("top.my_mux.sbar");
        $my_monitor("top.my_mux.y1");
    end
    initial
    begin
        I0 = 1'b0; I1 = 1'b1; S = 1'b0;
        #5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
        #5 I0 = 1'b0; I1 = 1'b1; S = 1'bx;
        #5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
    end
endmodule

// Результат моделювання
0 New value of net top.my_mux.y1 is 0
0 New value of net top.my_mux.sbar is 1
5 New value of net top.my_mux.y1 is 1
5 New value of net top.my_mux.sbar is 0
5 New value of net top.my_mux.y1 is 0
10 New value of net top.my_mux.sbar is X
15 New value of net top.my_mux.y1 is X
15 New value of net top.my_mux.sbar is 0
15 New value of net top.my_mux.y1 is 0
```

8.4. Обслуговуючі підпрограми (utility routine)

Обслуговуючі підпрограми (utility routine) - це різноманітні PLI-процедури для передачі даних в обох напрямках через межу "Verilog - Сі-програма користувача". Також їх часто називають tf-підпрограмами, оскільки вони завжди починаються із приставки tf_. Якщо такі підпрограми використовуються в Сі-кодї, то до нього повинен бути підключений файл veriusr.h, що містить опис всіх обслуговуючих підпрограм:

```
#include "veriusr.h".
```

Завдання, розв'язувані із застосуванням обслуговуючих підпрограм:

- отримання інформації про викликані системні завдання Verilog і про список аргументів. Зчитування й зміна значення аргументу для викликаного системного завдання. Спостереження за змінами значень аргументів;

- виконання допоміжних завдань, таких як збереження робочої області, покажчика на завдання;
- виконання складних обчислень;
- виведення повідомлення;
- отримання інформації про час моделювання й черги подій. Зупинка, завершення, збереження та відновлення процесу моделювання.

В описаних вище прикладах застосовувалася тільки одна tf-підпрограма `io_printf()`. Лістинг 8.9 містить приклад використання обслуговуючих підпрограм для створення користувальницького системного завдання `$my_stop_finish`, що управляє процесом моделювання. Verilog має два системні завдання для зупинки й завершення моделювання `$stop` і `$finish`. Розроблене завдання `$my_stop_finish` поєднує обидві ці функції. Її специфікація представлена в табл. 8.1.

Таблиця 8.1. Специфікація для `$my_stop_finish`

Аргументи		Дія
1-й	2-й	
0	Ні	Зупинка моделювання. Виведення повідомлення та модельного часу
1	Ні	Завершення моделювання. Виведення повідомлення та модельного часу
0	Будь-яке значення	Зупинка моделювання. Виведення повідомлення, ім'я модуля, з якого був зроблений виклик завдання й модельного часу
1	Будь-яке значення	Завершення моделювання. Виведення повідомлення, ім'я модуля, з якого був зроблений виклик завдання й модельного часу

У даному прикладі функція `tf_nump()` використовується для визначення числа переданих у системне завдання аргументів, а `tf_getp()` – для одержання самих аргументів. Функція `tf_gettime` повертає поточний час моделювання. Зупинка та завершення процесу моделювання виконується за допомогою `tf_dostop()` і `tf_dofinish()`. При відсутності аргументів користувальницьке системне завдання `$my_stop_finish()` буде видавати попередження, для чого використовується функція `tf_warning()`.

Лістинг 8.9. Сі-код користувальницького системного завдання. `$my_stop_finish`

```
int my_stop_finish()
{
    if(tf_nump() == 1) // Якщо в завдання передається 1 аргумент, то
        //виводиться тільки повідомлення про час моделювання
        // Отримання значення першого аргументу. Якщо аргумент дорівнює 0,
        if (tf_getp(1)==0) // моделювання зупиняється
        {
            io_printf("Mymessage: Simulation stopped at time %d\n", tf_gettime());
            tf_dostop(); // Зупинка моделювання
        }
        // Якщо аргумент дорівнює 1, моделювання завершується
        else if (tf_getp(1) == 1)
        {
            io_printf("Mymessage: Simulation finished at time %d\n", tf_gettime());
        }
}
```

```

        tf_dofinish();          // Завершення моделювання
    }
    else
        // Виведення попередження
        tf_warning("Bad arguments to $my_stop_finish at time %d\n",
            tf_gettime());
    }
    // Якщо в завдання my_stop_finish передається два аргументи,
    // то виводиться повідомлення з ім'ям модуля, з якого
    // виконувався виклик завдання, і часом
else if (tf_nump() == 2)
    { // Якщо аргумент дорівнює 0, моделювання зупиняється
        if (tf_getp(1) == 0)
        { io_printf("Mymessage: Simulation stopped at time %d in instance %s\n",
            tf_gettime(), tf_mipname());
            tf_dostop();          // Зупинка моделювання
        }
        // Якщо аргумент дорівнює 1, моделювання завершується
        else if (tf_getp(1) == 1)
        { io_printf("Mymessage: Simulation finished at time
            %d in instance %s\n", tf_gettime(), tf_mipname());
            tf_dofinish(); // Завершення моделювання
        }
        else
            // Виведення попередження
            tf_warning("Bad arguments to $my_stop_finish at time %d\n",
                tf_gettime());
        }
    return 0;
}

```

Лістинг 8.10 являє собою Verilog-модуль, що використовує створене завдання \$my_stop_finish для тестування пристрою mux2_to_1, і результат моделювання.

Лістинг 8.10. Verilog-модуль для тестування завдання \$my_stop_finish.

```

module top;
    wire OUT;
    reg I0, I1, S;
    mux2_to_1 my_mux(OUT, I0, I1, S);          // Копія модуля mux2_to_1
    initial // Формування тестових наборів
    begin
        I0 = 1'b0; I1 = 1'b1; S = 1'b0;
        $my_stop_finish(0);                    // Зупинка моделювання без виведення
                                                // імені копії модуля
        #5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
        $my_stop_finish(0,1);                  // Зупинка моделювання,
                                                // виводиться ім'я копії модуля
        #5 I0 = 1'b0; I1 = 1'b1; S = 1'b1;
    end

```

```

        $my_stop_finish(2,1);           // Завданню передане
                                         //неправильне значення аргумента
#5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
        $my_stop_finish(1,1);           // Завершення моделювання,
                                         // виводиться ім'я копії модуля
    end
endmodule
// Результат тестування Verilog-модуля програмою Active-HDL
run
# Mymessage: Simulation stopped at time 0
# RUNTIME: RUNTIME_0069 top.v (9): $stop called.
# KERNEL: Time: 0 ps, Iteration: 0, Instance: \.
# KERNEL: stopped at time: 0 ps
run
# Mymessage: Simulation stopped at time 5 in instance top
# RUNTIME: RUNTIME_0069 top.v (9): $stop called.
# KERNEL: Time: 5 ps, Iteration: 0, Instance: \.
# KERNEL: stopped at time: 5 ps
run
# : Bad arguments to $my_stop_finish at time 10
# : : $my_stop_finish f:\My_Designs\new\my_new/src/top1.v(13)
# Mymessage: Simulation finished at time 15 in instance top
# RUNTIME: RUNTIME_0068 top1.v (15): $finish called.
# KERNEL: Time: 15 ps, Iteration: 0, Instance: \.
# KERNEL: stopped at time: 15 ps

```

8.5. Приклади створення користувацьких системних завдань

Приклад 8.3. Користувальницьке завдання, представлене лістингом 8.11, виконує зміни затримок переднього та заднього фронту для примітива; отримує три аргументи, перший з яких є ім'ям вентиля, інші - константними значеннями затримок. Приклад виклику такої функції:

```
$timing_task("top.g12", 8.4, 9.2 );
```

Перший аргумент зчитується за допомогою функції `acc_handle_tfarg`, інші два - `tf_getrealp`. Підпрограма `acc_handle_tfarg` повертає визначники для аргументів у поточній копії користувацького системного завдання або функції. Аргументом може бути ланцюг, модуль або примітив. Аргумент задається номером, починаючи з 1. Ім'я примітива задається в подвійних лапках, імена модуля й ланцюга - без лапок:

```
$mytask(top.mod1,"top.mod1.nand6");
```

Лістинг 8.11. Користувальницьке завдання: для роботи із затримками примітива.

```

#include "acc_user.h"
new_timing()
{
    handle gate_handle;
    double new_rise, new_fall;
    // Підпрограми доступу ініціалізації й конфігурації

```

```

acc_initialize();
acc_configure( accDevelopmentVersion, "1.6a" );
acc_configure(accToHiZDelay, "max");
// Отримання визначника вентиля
gate_handle = acc_handle_tfarg( 1 );
// Отримання величин нових затримок
new_rise = tf_getrealp( 2 );
new_fall = tf_getrealp( 3 );
// Передача нових затримок вентилів
acc_replace_delays( gate_handle,new_rise,new_fall);
io_printf("Primitive %s has new delays %d %d\n",
          acc_fetch_fullname( gate_handle ),
          new_rise, new_fall );
acc_close();
}

```

Приклад 8.4. Повертає визначник на реалізацію батьківського примітива або модуля для заданого об'єкта, для цього використовується завдання `acc_handle_parent()` (лістинг 8.12). Батьківським вважається об'єкт, що містить інші об'єкти. Наприклад, для терміналів примітива батьківським вважається примітив, якому вони належать. Модуль верхнього рівня не має батьків.

Лістинг 8.12. Користувальницьке завдання: для роботи з затримками примітива.

```

#include "acc_user.h"
get_primitives(net_handle)
handle net_handle;
{
    handle primitive_handle;
    handle driver_handle;
    // Отримання примітива, якому належить кожний термінал,
    // який керує лінією
    driver_handle = null;
    while( driver_handle = acc_next_driver( net_handle, driver_handle ) )
    {
        primitive_handle = acc_handle_parent( driver_handle );
        io_printf( "Primitive %s drives net %s\n",
                  acc_fetch_fullname( primitive_handle ),
                  acc_fetch_fullname( net_handle ) );
    }
}

```

8.6. Контрольні завдання

1. Створити користувальницьке системне завдання `$get_in_port`, що формує повідомлення про вхідні порти заданого модуля. Ієрархічне ім'я модуля буде вхідним параметром завдання.
2. Написати користувальницьке системне завдання `$count_and_gates`, що підраховує число стандартних примітивів `and` у модулі. Для тестування використовувати таку модель мультиплексора:

```

module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);
    output Out;
    input In0, In1, In2, In3, Sel1, Sel0;

```

```

        wire NotSel0, NotSel1;
        wire Y0, Y1, Y2, Y3;
    and (Y1, In1, NotSel1, NotSel0);
    not (NotSel0, Sel0);
    and (Y3, In3, Sel1, Sel0);
    or (Out, Y0, Y1, Y2, Y3);
    and (Y0, In0, NotSel1, NotSel0);
    not (NotSel1, Sel1);
    and (Y2, In2, Sel1, NotSel0);
endmodule

```

3. Створити користувацьке системне завдання \$monitor_mod_output, що знаходить всі вихідні сигнали модуля й додає їх у список спостереження. Після будь-якої зміни вихідного сигналу модуля повинне виводитися повідомлення "Output signal has changed". Використовувати VCL-підпрограми.

4. Створити функцію, яка б виводила ім'я та список портів модуля, з якого вона викликається.

5. Створити функцію, яка б виводила ім'я та список примітивів, з якого вона викликається.

8.7. Література

1. IEEE Std 1364-1995. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language- IEEE Standard Verilog Hardware Description Language.- IEEE Computer Society Sponsored by the Design Automation Standards Committee.- Published by The Institute of Electrical and Electronics Engineers, Inc.- USA.- 1996.- 675 p.
2. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).- IEEE Standard Verilog Hardware Description Language.- IEEE Computer Society Sponsored by the Design Automation Standards Committee.- Published by The Institute of Electrical and Electronics Engineers, Inc.- USA.- 2001.- 791 p.
3. Samir Palnitkar. Verilog HDL. A guide to digital design and synthesis.- SunSoft Press.- 1996.- 396 p.
4. Хаханов В.И., Хаханова И.В. VHDL + Verilog = Синтез за минуты.- Харьков: СМІТ.- 2007.- 264 с.
5. Donald E. Thomas, Philip R. Moorby. The Verilog Hardware Description Language.- New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.- 2002.- 404 p.
6. James M. Lee, VERILOG QUICKSTART. A Practical Guide to Simulation and Synthesis in Verilog.- New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.- 2002.- 378 p.
7. Stuart Sutherland. The Verilog PLI Handbook. A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface.- New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.- 2002.- 809 p.
8. Verilog-HDL PLI. Reference Manual.- Open Verilog International.-1991.-350 p.
9. Michael D. Ciletti. Advanced Digital Design with the Verilog HDL.-Prentice, Hall of India.- 2005.- 1014 p.
10. Поляков А.К. Языки VHDL и Verilog в проектировании цифровой аппаратуры.- М: СОЛОН-Пресс.- 2003.-320 с.

ЧАСТИНА ІІІ. СИНТЕЗ МОДЕЛЕЙ НА МОВІ VERILOG

Розглядаються основні етапи процесу логічного синтезу цифрових систем на кристалах засобами мови Verilog: перетворення Verilog-операторів в елементи й схемні конструкції RTL-рівня, а також технології оптимізації післясинтезних рішень. У розд. 9-11 представлена сучасна методологія проектування SoC на основі HDL-моделі цифрових пристроїв RTL-рівня, особливості перетворення Verilog-операторів у схему, що складається зі стандартних елементів, шаблони комбінаційних компонентів і послідовної логіки.

Мета - вивчення та практичне використання технологій синтезу спеціалізованих цифрових систем на кристалах на основі HDL-моделей компонентів, представлених на RTL-рівні опису засобами ефективного мовного середовища Verilog, шляхом розгляду численних прикладів апаратної реалізації комплектуючих компонентів.

Завдання: 1. Синтезовані конструкції в Verilog: загальне поняття синтезу; принципи побудови синтезованого коду; синтезовані конструкції в Verilog. 2. Інтерпретація операторів Verilog: типи даних; масиви; використання векторів - присвоєння вектору значення вектора меншої довжини, використання діапазону та окремих елементів векторів; синтез комбінаційних схем - оператор безперервного призначення assign, поведінковий код, процедурні оператори призначення; синтез послідовних пристроїв - моделі D-тригерів, реалізація послідовних схем за допомогою always; особливі ситуації синтезу – блокуючі та неблокуючі оператори, оператор if, оператор case, використання операторів casex і casez, директива синтезу parallel_case, використання неконстантних варіантів вибору для оператора case; оператори циклів; локальне використання змінних; відмінність між блокуючими та неблокуючими змінними; оператори Assign і Deassign; синтез підпрограм; використання параметрів; використання значення x. 3. Шаблони стандартних рішень: комбінаційні компоненти - мультиплексори, дешифратори, пристрої зсуву, арифметичні пристрої, спільне використання ресурсів (Resource Sharing); послідовна логіка - тригери із синхронізацією по фронту, тригери-засувки, лічильники, зсувні регістри; автомати; моделі пам'яті. 4. Контрольні запитання й завдання.

Джерела: синтез цифрової апаратури за допомогою мови Verilog [1, 2]; мова опису апаратури Verilog [3, 4]; стандарт мови опису апаратури Verilog [5-7].

9. СИНТЕЗОВАНІ КОНСТРУКЦІЇ В СЕРЕДОВИЩІ VERILOG

9.1. Загальне поняття синтезу

Синтез - це процес перетворення високорівневого опису проекту в оптимізовану модель вентильного рівня, що складається з елементів цільової бібліотеки стандартних комірок або компонентів програмувальної логіки, яка також має назву технологічної або цільової бібліотеки.

Логічний синтез існував давно, з моменту створення перших цифрових пристроїв. Однак раніше він виконувався вручну. Поява ж мов опису апаратури й створення спеціальних програм для перетворення HDL-коду в схемне подання привело до революції в проектуванні обчислювальних систем. Автоматизоване перетворення високорівневого подання моделі в схему дозволило значно підвищити продуктивність, скоротити час проектування й виключити помилки, обумовлені людським фактором. Робота, на виконання якої раніше було потрібно кілька місяців, тепер може бути виконана за години. Не викликає ніяких труднощів створення декількох версій реалізації проекту з різними параметрами для вибору більш оптимального варіанту. Можливе використання одного і того ж проекту для різних технологій реалізації.

Фірми, орієнтовані на CAD (Computer-Aided Design), пропонують різні програми для виконання синтезу логічних схем з HDL-коду, наприклад Synopsys Synplify Pro і Synplify Premier, Synopsys Design Compiler Ultra, Xilinx XST, Cadence Encounter HDL-Compiler, Mentor Graphics Leonardo Spectrum. Одні з них призначені для проектування на програмованій логіці, інші - на ASIC. Вони використовують різні алгоритми та принципи перетворення HDL-коду в схему, і як наслідок, можуть підтримувати підмножини, що відрізняються, конструкцій мов опису апаратури й генерувати різні за розміром та швидкістю схеми.

Типова схема виконання процесу синтезу зображена на рис. 9.1 [1, 2]. Вихідні дані представлені HDL-моделлю цифрового пристрою RTL-рівня. Перший етап Translation являє собою процес перетворення конструкцій мови опису апаратури в схему, що складається зі стандартних елементів. Програма замінює оператори HDL своїми бібліотечними елементами, формуючи схему RTL-рівня. Під стандартними елементами в одних програмах (наприклад, Synplify і XST) мають на увазі макроси: суматори, лічильники, автомати, пам'ять. В інших - це можуть бути тільки логічні елементи та тригери. Як правило, за першим принципом виконується RTL-синтез пристроїв, що реалізуються на FPGA, а другий підхід використовується для ASIC. Цей етап залежить тільки від вихідного HDL-коду, і для всіх цільових технологій реалізації пристрою програма синтезу буде генерувати ту саму схему.

Наступним етапом є оптимізація отриманої схеми (блок Logic Optimization), результат перетвориться в схему, що складається з елементів заданої технології Technology Mapping and Optimization. Це можуть бути таблиці перетворень LUT і тригери (для FPGA) або стандартні комірки (для ASIC). На цьому

етапі програми використовують технологічні бібліотеки (блок Technology Library), що описують реалізацію стандартних елементів у заданому елементному базисі. Константи проектування (Design Constraints) представляють механізм керування процесом синтезу й можуть нести різноманітну додаткову інформацію щодо формування результуючої схеми, як, наприклад, вимоги до розміру або швидкодії пристрою, розташування зовнішніх контактів.

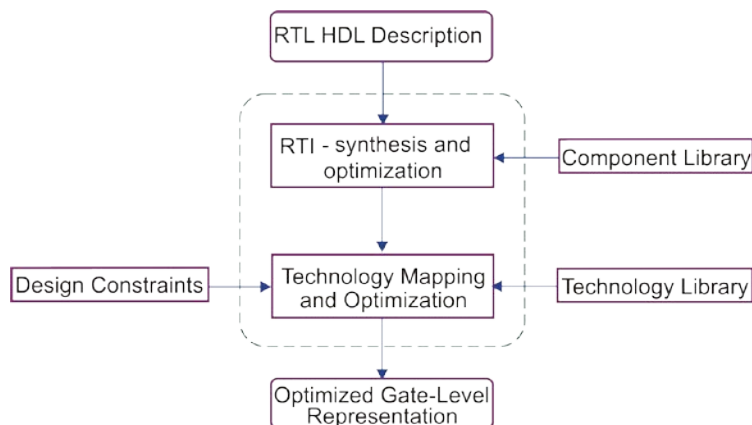


Рис. 9.1. Етапи синтезу

9.2. Принципи побудови синтезованого коду

У даний час компанії використовують стандарти синтезу для VHDL (IEEE Std 1076.6-1999: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis) і Verilog (IEEE Std 1364.1-2002: IEEE Standard for Verilog Register Transfer Level Synthesis) [5-7]. Однак різні програми синтезу підтримують різні підмножини конструкцій мов опису апаратури. Та сама конструкція, застосовна для однієї програми, може бути не синтезована в іншій. Проте існує якийсь єдиний підхід до написання синтезованого HDL-коду, в основу якого покладені загальні принципи виконання синтезу:

1. HDL-модель повинна бути представлена на RTL-рівні. Поводження пристрою повинно бути описане з точністю до такту синхросигналу. У загальному випадку, будь-яка конструкція, що моделює роботу пристрою на кожному такті функціонування RTL-схеми, може бути синтезована.
2. Не допускається використання конструкцій, що включають часові параметри.
3. Не допускається застосування механізмів ініціалізації засобами HDL-мови, тільки за допомогою зовнішніх входів, наприклад, скидання й установки.
4. Рекомендується явно вказувати розмір (розрядність) даних.

Найголовніше, при написанні коду дуже важливо, щоб проектувальник уявляв, з яких апаратних елементів буде складатися його проект, як він буде реалізований в апаратурі. Під апаратними елементами мається на увазі стан-

дартний набір компонентів: тригери, регістри, суматори, вентилі, мультиплексори, дешифратори.

9.3. Синтезовані конструкції в Verilog

Набір підтримуваних для синтезу конструкцій мови Verilog представлений табл. 9.1 [3, 4], однак він може змінюватися для різних програм синтезу.

Таблиця 9.1. Синтезовані конструкції мови Verilog

Тип конструкції	Ключове слово або опис	Примітки
Модуль	module	
Порти	input, inout, output	
Параметри	parameter	#() і defparam підтримуються тільки на один рівень вниз
Сигнали та змінні	wire, tri reg, integer	Дозволені вектори та масиви
Події	@()	Не підтримуються іменні події
Функції та завдання	function, task	Ігноруються тимчасові параметри, виклик системних завдань та функцій
Поведінковий опис	always	Initial ігнорується
	begin - end іменні блоки, disable	Не підтримується fork-join, не дозволяється використання disable для іменних блоків
	always, if-then-else, case, casex, casez	Не підтримуються deassign, wait, force, release
	Цикл for	Підтримується повністю
	Цикли: while, forever, repeat	Стандартом синтезу не підтримуються. Окремі програми можуть виконувати синтез із обмеженнями
Рівень data flow	assign	Інформація про затримки ігнорується
Вентильний рівень	Стандартні примітиви	
Реалізація компонентів	Реалізація модуля, реалізація вентильного примітива	Наприклад, mux m1(out, i0, i1, s); nand (out, a, b);

Оскільки для синтезу не підтримується попередня ініціалізація даних в HDL-кодї, то в мові Verilog блок initial не є синтезованим. Для ініціалізації даних варто використовувати механізм скидання, що реалізується за допомогою спеціальних входів.

При виконанні синтезу ігноруються всі параметри часу, що були описані у форматі #<затримка>.

Всі оператори мови Verilog, крім `===` і `!==`, синтезовані.

Символи `x` і `z` не мають значення в апаратурі, тому `===` і `!==` є спеціальними операторами для моделювання. При синтезі вони замінюються на `==` і `!=`, відповідно.

Програмами синтезу підтримуються тільки послідовні блоки, які обмежуються ключовими словами `begin` і `end`. Використання паралельних блоків `fork-join` не дозволено.

Підтримується застосування стандартних примітивів для опису вентильних схем. Однак визначені користувачем примітиви ігноруються. Більш докладно реалізація операторів Verilog при виконанні RTL-синтезу розглядається в главі 10.

Табл. 9.2 представляє основні директиви компіляції мови Verilog і їхню інтерпретацію програмами синтезу.

Таблиця 9.2. Директиви компіляції

Підтримуються	
<code>`define</code>	<code>`undef</code> , <code>`indef</code> , <code>`elsif</code> ,
<code>`include</code>	<code>`ifdef</code> <code>`else</code> <code>`endif</code>
<code>`file</code> , <code>`line</code>	<code>`default_nettype</code>
Ігноруються	
<code>`timescale</code>	<code>`resetall</code>

9.4. Контрольні запитання

1. Що відбувається при синтезі тимчасовими параметрами моделі?
2. Які цикли є повністю синтезованими?
3. Які блоки синтезуються?
4. Чи можна в синтезованих моделях використовувати параметри?
5. Чи є примітиви синтезованими?

10. ІНТЕРПРЕТАЦІЯ ОПЕРАТОРІВ VERILOG

Далі на прикладах розглянутий перший етап синтезу - перетворення Verilog-операторів в елементи схеми RTL-рівня, що складається зі стандартних бібліотечних компонентів програми синтезу й не залежить від конкретної технології реалізації пристрою. Для реалізації прикладів синтезу використовувався програма XST.

10.1. Типи даних

Перелік підтримуваних для синтезу типів даних Verilog представлений табл. 10.1 [3, 4]. Типи даних integer і real рекомендується застосовувати для побудови тестів або абстрактних моделей. Для створення ж моделей реальної апаратури переважніше використання типів reg або wire, які дозволяють точно описувати архітектуру пристрою. В Verilog 2001 вектори обох типів даних можуть бути використані для створення як знакових, так і беззнакових даних:

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

Таблиця 10.1. Типи даних, які підтримуються програмами синтезу

Клас даних	XST	Synplify
Ланцюги (Net)	wire, tri, triand/wand, trior/wor supply0, supply1	wire, tri, supply1, supply0
Регістри (Register)	reg, integer	reg, integer time (64 bit reg)

10.1.1. Масиви.

Масиви в Verilog синтезовані, дозволяється використовувати масиви типів reg і wire, наприклад:

```
reg [3:0] mem_array [31:0];
wire [7:0] mem_array [63:0];
```

Підтримуються багатомірні масиви розміром до трьох вимірів:

- a) wire [7:0] array2 [0:255][0:15];
- b) reg [63:0] regarray2 [255:0][7:0];
- c) wire [7:0] array3 [0:15][0:255][0:15];

Не допускається виконувати присвоєння більш ніж одному елементу масиву одночасно. У попередньому прикладі дозволене звернення до масиву: а) 8-розрядними словами; б) 64-розрядними словами; с) 8-розрядними словами.

10.2. Використання векторів

10.2.1. Присвоєння вектору значення вектора меншої довжини.

Якщо вектор, що отримує значення, має більшу довжину, ніж вектор, з якого дані передаються, то вектор-джерело доповнюється бітами з боку старших розрядів за таким правилом [3, 4]:

- 1) якщо вектор із правого боку є знаковим, то вектор з лівого боку доповнюється нулями, якщо він є позитивним - одиницями, наявність високого імпедансу - z, значення вектора не визначене - x (рис. 10.1, а);
- 2) якщо вектор із правого боку є беззнаковим, то вектор з лівого боку доповнюється нулями (рис. 10.1, б);
- 3) якщо для констант z і x розмір не зазначений, то вони доповнюються значеннями z і x відповідно, при цьому не має значення - знакова беззнакова змінна із правого боку виразу.

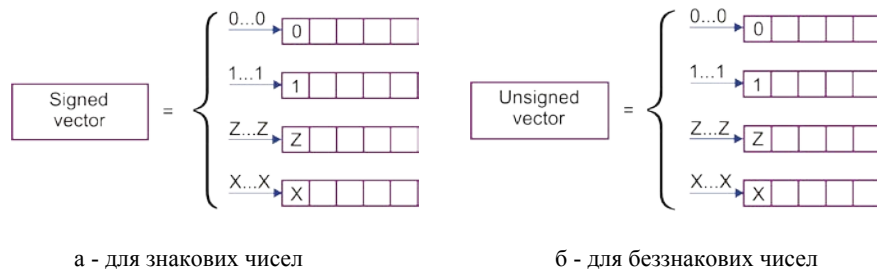


Рис. 10.1. Розширення векторів

10.2.2. Використання діапазону та окремих елементів векторів.

В Verilog-моделях RTL-рівня можна вільно використовувати як весь вектор цілком, так і його піддіапазони або окремі елементи (лістинг 10.1).

Лістинг 10.1. Приклад використання елементів вектора

```
module ConstantIndex
    (input[3:0] A, C, RegFile,
     output [3:0] ZCat);
    assign ZCat [2:0] ={A[2], C[3:2]};
    assign ZCat[0] = RegFile[3];
endmodule
```

Якщо в правій частині виразу для вибору елементів вектора як індекс використовується змінна, то такий вираз реалізується мультиплексором (лістинг 10.2, рис. 10.2).

Якщо змінний індекс застосовується з лівого боку виразу, то в результаті синтезу формується пам'ять.

У прикладі (лістинг 10.3, рис. 10.3) апаратний пристрій реалізується на тригерах-засувках з використанням логічних елементів для вибору необхідної комірки для занесення значення.

Лістинг 10.2. Використання вектора зі змінним індексом у виразі.

```
module NonComputeRight
    (input[0:3] Data, input [1:0] Index,
     output Dout);
    assign Dout = Data [Index];
endmodule
```

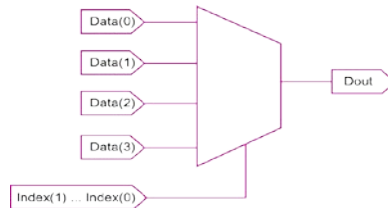


Рис. 10.2. Мультиплексор з вектора зі змінним індексом

Лістинг 10.3. Використання вектора зі змінним індексом.

```
module NonComputeLeft
    (input Data, input [2:0] Addr,
     output reg [7:0] Mem);
    always @(Data, Addr)
        Mem[Addr] = Data;
endmodule
```

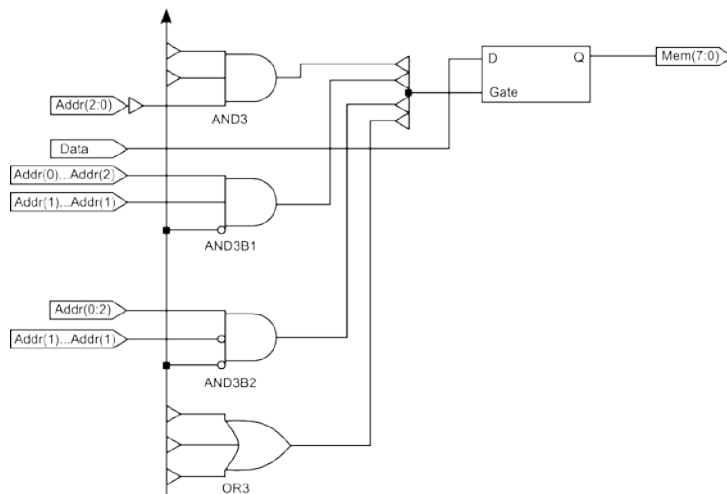


Рис. 10.3. Результат синтезу (лістинг 10.3)

10.3. Синтез комбінаційних схем

Модель комбінаційного пристрою може бути описана за допомогою [1-4]:

- 1) стандартних примітивів логічних елементів (лістинг 10.4, рис. 10.4);
- 2) операторів assign в data-flow моделях;
- 3) процедурного оператора always, при цьому список чутливості повинен містити всі входи комбінаційної функції (всі змінні, зазначені із правого боку виразу або як умова операторів if і case) або мати форму @(*).

При використанні операторів if або case необхідно, щоб вихідний сигнал отримував значення при кожній можливій ситуації, тобто при кожній ітерації виконання блоку begin-end вихідний сигнал повинен обновляти своє значення.

Лістинг 10.4. Модель мультиплексора 4-в-1 вентиляного рівня.

```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);
    output Out;
    input In0, In1, In2, In3, Sel1, Sel0;
    wire NotSel0, NotSel1;
    wire Y0, Y1, Y2, Y3;
    and (Y1, In1, NotSel1, Sel0);
    not (NotSel0, Sel0);
    and (Y3, In3, Sel1, Sel0);
    or (Out, Y0, Y1, Y2, Y3);
    and (Y0, In0, NotSel1, NotSel0);
    not (NotSel1, Sel1);
    and (Y2, In2, Sel1, NotSel0);
endmodule
```

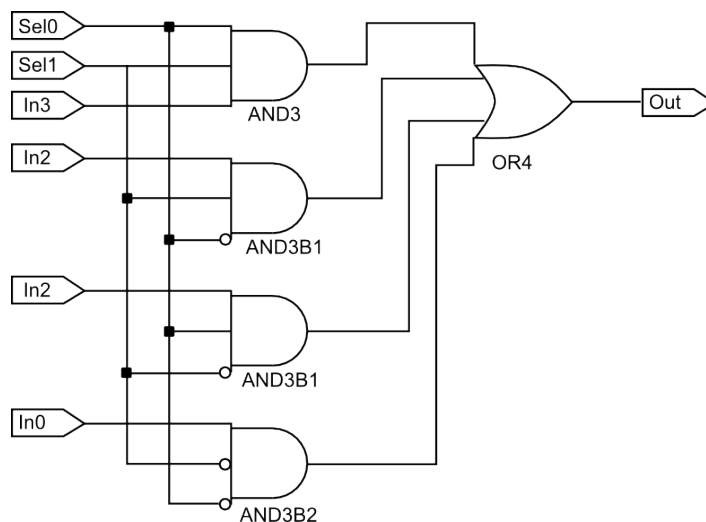


Рис. 10.4. Результат синтезу (лістинг 10.4)

10.3.1. Оператор безперервного призначення assign.

Оператор безперервного призначення assign використовується для подання комбінаційної логіки. Наприклад, для виразу:

```
assign out=(a&b)|c;
```

схема RTL-рівня буде мати вигляд, представлений рис. 10.5.

Якщо a, b і c - двоелементні вектори, то буде згенеровано дві схеми відповідно до рис. 10.5.

Арифметичні оператори реалізуються за допомогою арифметичних блоків, доступних у бібліотеці програми. Наприклад, рис. 10.6. представляє реалізацію програмою Synplify суматора, описаного виразом:
`assign{c_out, sum} = a + b + c_in;`

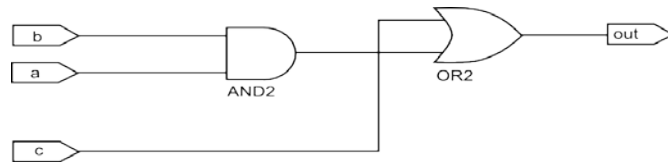


Рис. 10.5. Вентильна схема

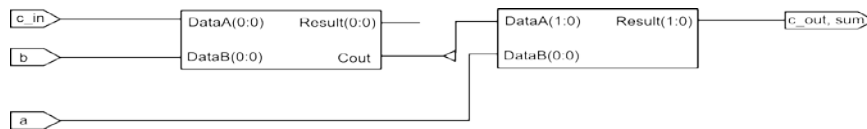


Рис. 10.6. Реалізація повного суматора

Умовний оператор реалізується мультиплексором. Таким чином, наступний вираз буде відповідати рис. 10.7:

`assign out=(s)? a: b;`

Щоб уявити, скільки цей пристрій зажадає апаратури, варто пам'ятати, що мультиплексор 2-в-1 описується булевим рівнянням трьох змінних:
`out=s'a+sb.`

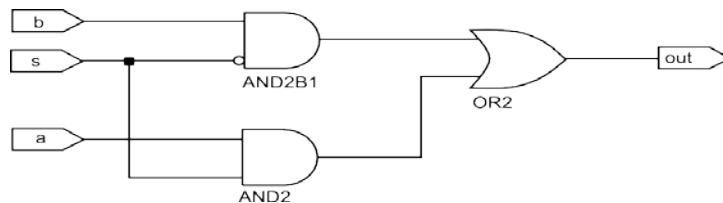


Рис. 10.7. Мультиплексор

10.3.2. Поведінковий код. Процедурні оператори призначення.

Блок `always` може бути використаний для реалізації послідовної й комбінаційної логіки. Наступний приклад представляє код комбінаційного суматора, результат синтезу якого представлений рис. 10.7:

```
module mux_4_to_1 (output reg c_out, sum, input a, b, c_in);
  always @(*)
    {c_out, sum} = a + b + c_in;
endmodule
```

Допускається використання скороченого запису опису всіх портів у списку чутливості:

```
always @(*)      // або always @*
  {c_out, sum} = a + b + c_in;
```

Дана конструкція генерує схему, аналогічну попередній. Використання блокуючих та неблокуючих операторів не зробить ніякого впливу на результуючу схему, різниця буде тільки в результатах моделювання. Рекомендується для комбінаційних схем використовувати блокуючі оператори, а для послідовних - неблокуючі.

10.4. Синтез послідовних пристроїв

Для проектування послідовної логіки використовується оператор `always` [1-4]. При написанні якісного коду слід дотримуватися таких правил:

- 1) У списку чутливості `always` допускається використовувати перевірку тільки подій синхросигналів або тільки фронтів синхросигналів. Вказуються тільки сигнали, що безпосередньо впливають на зміну стану.
- 2) В операторі `always` умови, що відповідають сигналам `reset` і `preset`, описуються першими.
- 3) У середині блоку `begin-end` умова синхронізації не описується.
- 4) Не можна описувати в одному блоці комбінаційну й послідовну схеми.
- 5) Для моделей послідовних схем переважніше використання неблокуючих операторів "`<=`".

10.4.1. Моделі D-тригерів.

Для послідовної логіки блок `always` повинен управлятися зміною сигналу `clk`. Лістинг 10.5 а) представляє D-тригер із синхронізацією по передньому фронту, його схема представлена на рис. 10.8, а. Щоб згенерувати тригеразасувку із синхронізацією за рівнем (рис.10.8, б), треба в управлінні блоку `always` указати синхросигнал і вхід даних, а також включити в нього оператор `if` (лістинг 10.5, б).

Лістинг 10.5. Verilog -моделі D-тригера.

<pre>a) module FlipFlop (input C, D, output reg Q); always @(posedge C) Q <= D; endmodule</pre>	<pre>б) module Latch (input Clk, D, output reg Q); always @(Clk, D) if (Clk) Q <= D; endmodule</pre>

Лістинг 10.6 представляє модель D-тригера з асинхронним скиданням (рис. 10.9). Для того щоб програма синтезу не видала помилку, у списку чутливості оператора `always` слід використовувати перевірку фронтів (`negedge` або `posedge`) для всіх керуючих сигналів.

Лістинг 10.6. Verilog-модель тригера із синхронізацією по задньому фронту й асинхронним скиданням.

```
module flop1
    (input C, D, CLR,
     output reg Q);
    always @(negedge C, posedge CLR)
```

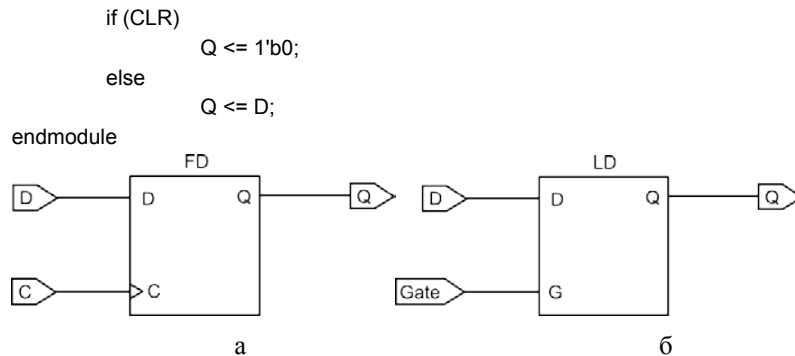


Рис. 10.8. Результати синтезу моделей D-тригера

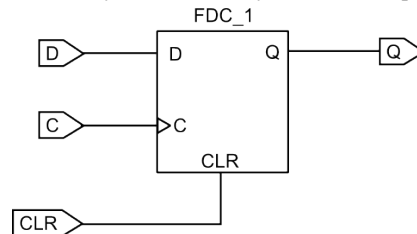


Рис. 10.9. Тригер із синхронізацією по передньому фронту й асинхронним скиданням

10.4.2. Реалізація послідовних схем за допомогою always.

Список чутливості може містити події, що визначають фронт синхросигналу та керуючі сигнали. Блок always може включати тільки один оператор if-else. Асинхронна частина може з'являтися до синхронної частини в перших гілках if-else. В асинхронній частині сигнали можуть отримувати константні значення '0', '1', 'X' або 'Z', або комбінацію цих значень. Лістинг 10.7 представляє шаблон з асинхронними керуючими сигналами. Лістинг 10.8 представляє модель 8-бітового регістра, що використовує асинхронний керуючий сигнал скидання. Результат синтезу зображений на рис. 10.10.

Лістинг 10.7. Шаблон послідовного пристрою з асинхронними керуючими сигналами.

```

always @(posedge A, negedge B, negedge C, ... posedge Clock)
    if (A)           //асинхронний сигнал A, активний 1
        <statement> //асинхронна логіка
    else if (!B)      //асинхронний сигнал B, активний 0
        <statement> //асинхронна логіка
    else if (!C)      // асинхронний сигнал C, активний 0
        <statement> // асинхронна логіка
    . . .             //Будь-яка кількість конструкцій else if
    else              // передній фронт Clock
        <statement> //синхронна логіка

```

Лістинг 10.8. Регістр із асинхронним скиданням розміром в 8-біт.

```

module EXAMPLE
    (input [7:0] DI, input CLK, RST,
     output reg [7:0] DO);
always @(posedge CLK or posedge RST)
    if (RST == 1'b1)
        DO = 8'b00000000;
    else
        DO = DI;
endmodule

```

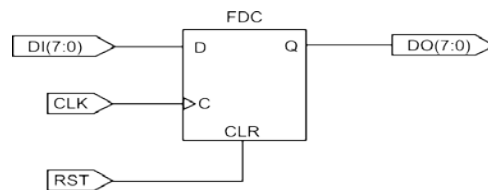


Рис. 10.10. Результат синтезу регістра (лістинг 10.8)

Лістинг 10.9 представляє реверсивний лічильник, напрямок рахунку в якому залежить від синхронного керуючого сигналу UpDown. Розрядність лічильника задається за допомогою параметра Num_bits. Пристрій містить асинхронне скидання Clear, перевірка переднього фронту цього синхросигналу зазначена в списку чутливості. Синхронний сигнал Preset виконує попередню установку лічильника значенням із входу Data. Рис. 10.11 зображує результат синтезу лічильника: інтерфейс (рис. 10.11, а) і RTL-схему (рис. 10.11, б).

Лістинг 10.9. Лічильник з асинхронними й синхронними керуючими сигналами.

```

module AsynCounter
    #(parameter Num_bits = 2)
    (input Clk, Preset, UpDown, Clear,
     input [Num_bits-1:0] Data,
     output reg [Num_bits-1:0] Counter);
always @(posedge Clear, posedge Clk)
    if (Clear) Counter <= 0;
    else if (Preset) Counter <= Data;
    else if (UpDown) Counter <= Counter + 1;
    else Counter <= Counter - 1;
endmodule

```

10.5. Особливі ситуації синтезу

10.5.1. Блокуючі та неблокуючі оператори.

Не дозволяється, щоб сигнал отримував значення з використанням блокуючих та неблокуючих операторів одночасно, наприклад, як у такому фрагменті коду [1-4]:

```

always @(*)
    if (in2) out1 = in1;
    else out1 <= in2;

```

Це обмеження діє також для елементів або діапазонів вектора. Не можна використовувати блокуючі та неблокуючі оператори для присвоєння значень діапазонам одного вектора, навіть якщо немає реального змішування стилю для окремо взятих бітів, наприклад, таким чином:

```
if (in2) begin
    out1[0] = 1'b0;
    out1[1] <= in1;
end
else begin
    out1[0] = in2;
    out1[1] <= 1'b1;
end
```

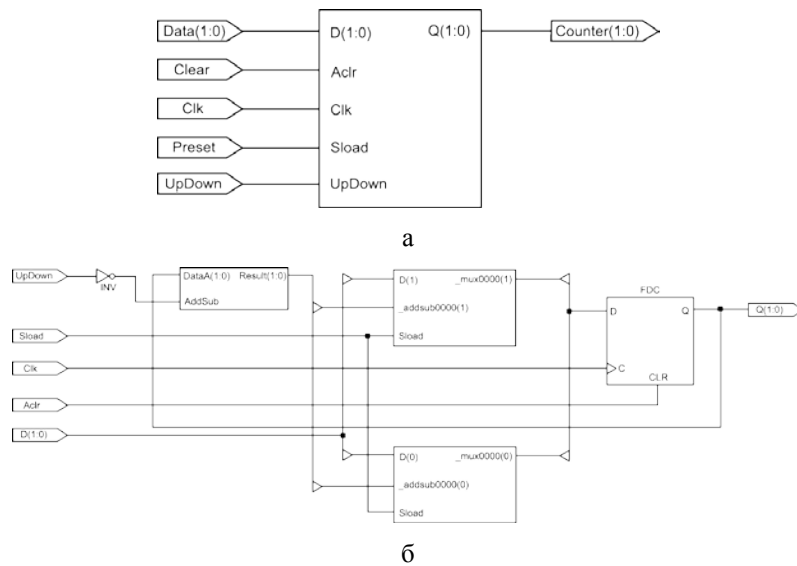


Рис. 10.11. Результат синтезу лічильника (лістинг 10.9)

10.5.2. Оператор if.

Оператор if може генерувати комбінаційну логіку (лістинг 10.10, а), рис. 10.12) або послідовну (лістинг 10.10, б), рис 10.13). Тут діє правило: якщо значення змінної явно визначене у всіх гілках оператора if, то при синтезі створюється комбінаційна схема, у противному випадку - послідовна.

Лістинг 10.10. Використання оператор if.

а) module ex1 (input A, B, output reg Z); always @(A, B) if (A>B) Z=A; else Z=B; endmodule	б) module ex2 (input A, B, output reg Z); always @(A, B) if (A>B) Z=A; endmodule

Оператор `if` може бути вкладеним. У цьому випадку на нього діє таке ж саме правило. Приклад з лістинга 10.11 ілюструє використання вкладеного оператора `if` і оператора `always` для створення моделі й синтезу мультиплексора 4-в-1 (рис. 10.14). Всі вхідні змінні вказані в списку чутливості `always`.

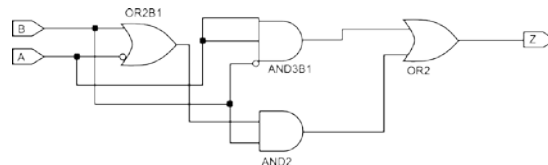


Рис. 10.12. Комбінаційна схема з оператора `if`

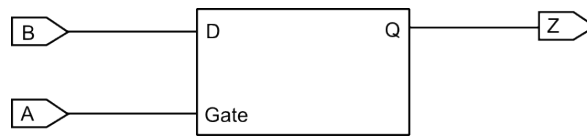


Рис. 10.13. Тригер-засувка схема з оператора `if`

Лістинг 10.11. Вкладений оператор `if` для опису мультиплексора.

```
module mux4
    (input [1:0] sel, a, b, c, d,
     output reg [1:0] outmux);
    always @(sel, a, b, c, d)
    begin
        if (sel[1])
            if (sel[0]) outmux = d;
            else outmux = c;
        else
            if (sel[0]) outmux = b;
            else outmux = a;
    end
endmodule
```

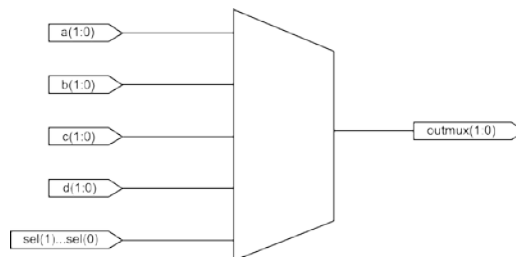


Рис. 10.14. Мультиплексор (лістинг 10.11)

10.5.3. Оператор `case`.

Оператор `case`, подібно до оператора `if`, може генерувати як комбінаційну, так і послідовну логіку. Точно так само, як і для `if`, якщо значення змінної явно визначене для всіх варіантів вибору оператора `case`, при синтезі ство-

рюється комбінаційна схема (лістинг 10.12, рис. 10.15), інакше - послідовна (лістинг 10.13, рис. 10.16). Реалізація схеми із тригером-засувкою може бути різною й залежить від програми синтезу.

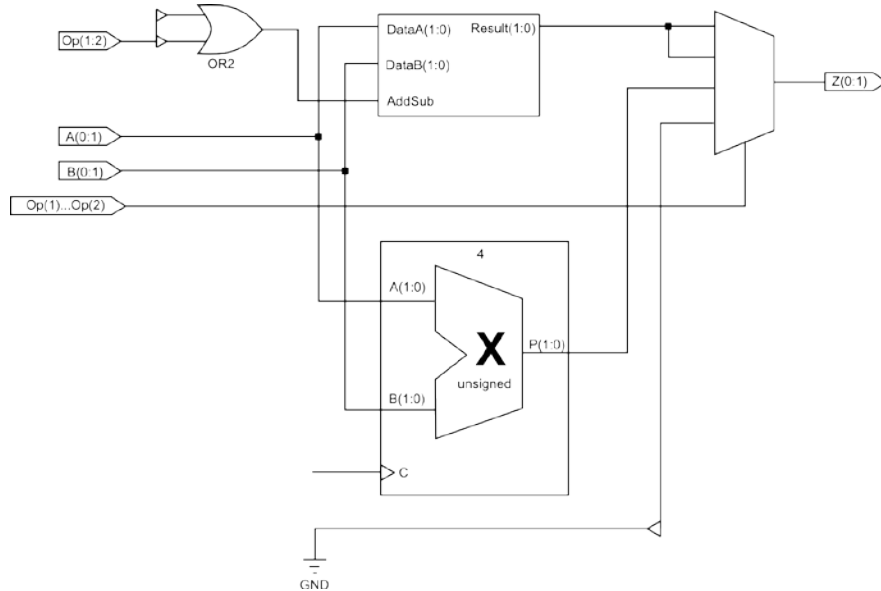


Рис. 10.15. Комбінаційний АЛП

Лістинг 10.12. Оператор case для комбінаційного АЛП.

```
module ALU
    #(parameter ADD = 'b00, SUB = 'b01, MUL = 'b10, DIV = 'b11)
    (input [1:2] Op, input [0:1] A, B,
     output reg [0:1] Z);
    always @(Op, A, B)
        case (Op)
            ADD : Z = A + B;
            SUB : Z = A - B;
            MUL : Z = A * B;
            DIV : Z = A / 2;
        endcase
endmodule
```

Лістинг 10.13. Оператор case, що приведений до появи тригерів.

```
module NextStateLogic
    (input [1:0] Toggle,
     output reg [1:0] NextToggle);
    always @(Toggle)
        case (Toggle)
            2'b01: NextToggle = 2'b10;
            2'b10: NextToggle = 2'b01;
        endcase
endmodule
```

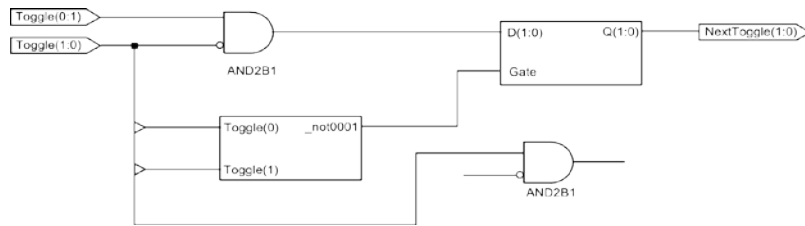



Рис.

10.16. Результат синтезу коду (лістинг 10.13)

За допомогою атрибутів можна керувати процесом синтезу. Так, атрибут `full_case` дозволить уникнути появи тригера, навіть якщо не всі можливі варіанти значень перебрані в операторі `case`. У наступних прикладах у коді лістинга 10.14 використовується гілка `default`, а в коді лістинга 10.15 - директива `full_case`. У результаті синтезу генеруються комбінаційні схеми (рис. 10.17 і 10.18).

Лістинг 10.14. Оператор `case` з гілкою `default`.

```
module NextStateLogic
    (input [1:0] Toggle, output reg [1:0] NextToggle);
    always @(Toggle)
        case (Toggle)
            2'b01: NextToggle = 2'b10;
            2'b10: NextToggle = 2'b01;
            default: NextToggle = 2'b01;
        endcase
endmodule
```



Рис. 10.17. Результат синтезу коду (лістинг 10.14)

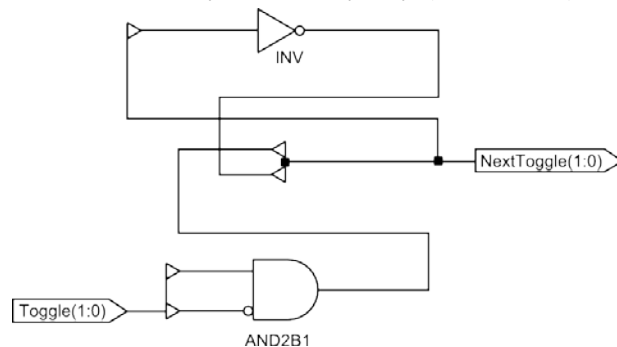


Рис. 10.18. Результат синтезу коду (лістинг 10.15)

Лістинг 10.15. Оператор `case` з атрибутом `full_case`

```
module NextStateLogic
    (input [1:0] Toggle, output reg [1:0] NextToggle);
```

```

always @(Toggle)
    case (Toggle)
        //synthesis full_case
        2'b01: NextToggle = 2'b10;
        2'b10: NextToggle = 2'b01;
    endcase
endmodule

```

10.5.4. Використання операторів casex і casez.

В операторі casez символ z розглядається як той, що не має значення. Символи z і x не допускається використовувати в змінній вибору оператора case. Лістингами 10.16 і 10.17 представлені приклади використання операторів casez і casex у моделях, результати синтезу зображені на рис. 10.19 і 10.20.

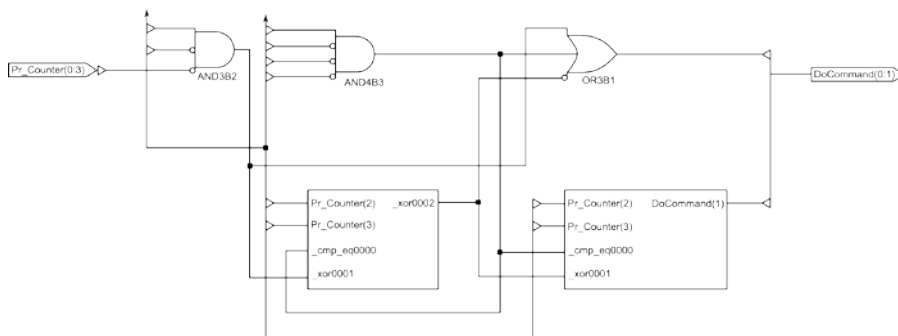


Рис. 10.19. Результат синтезу коду (лістинг 10.16)

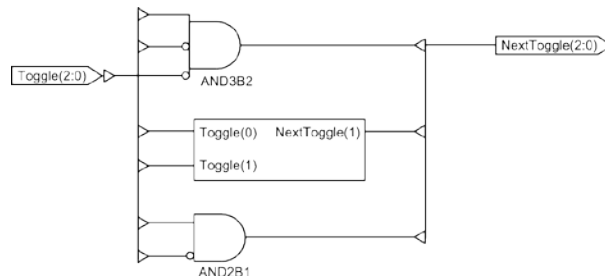


Рис. 10.20. Результат синтезу коду (лістинг 10.17)

Лістинг 10.16. Приклад коду з оператором casez.

```

module Casez_Ex
    (input [0:3] Pr_Counter,
     output reg [0:1] DoCommand);
always @(Pr_Counter)
    casez(Pr_Counter)
        4'b???1 : DoCommand = 0;
        4'b???10 : DoCommand = 1;
        4'b?100 : DoCommand = 2;
        4'b1000 : DoCommand = 3;
        default : DoCommand = 3;
    endcase
endmodule

```

```

        endcase
    endmodule

```

Лістинг 10.17. Приклад коду з оператором `casex`.

```

module PriorityLogic
    (input [2:0] Toggle,
     output reg [2:0] NextToggle);
always @(Toggle)
    casex (Toggle)
        3'bxx1: NextToggle = 3'b010;
        3'bx1x: NextToggle = 3'b110;
        3'b1xx: NextToggle = 3'b001;
        default: NextToggle = 3'b000;
    endcase
endmodule

```

10.5.5. Директива синтезу `parallel_case`.

В Verilog оператор `case` має пріоритетну природу. Якщо кілька варіантів вибору задовольняють заданому значенню, то будуть обрані оператори з першої по порядку гілки для оператора. Однак в Verilog є механізм одночасного вибору оператора - паралельний `case`. Його можна використовувати в ситуації, коли неможливий одночасний вибір декількох гілок. Паралельний `case` реалізується за допомогою атрибута `parallel_case` (лістинг 10.18). У ряді випадків це дозволяє отримати більш просту й швидку апаратну реалізацію (рис. 10.21).

Лістинг 10.18. Приклад використання директиви `parallel_case`.

```

module ParallelCase (input [2:0] Toggle, output reg [2:0] NextToggle);
always @(Toggle)
    casex (Toggle) // synthesis parallel_case
        3'bxx1: NextToggle = 3'b010;
        3'bx1x: NextToggle = 3'b110;
        3'b1xx: NextToggle = 3'b001;
        default: NextToggle = 3'b000;
    endcase
endmodule

```

10.5.6. Використання неконстантних варіантів вибору для оператора `case`.

В Verilog можлива ситуація, коли варіанти вибору оператора `case` представлені змінними й мають неконстантні значення (лістинг 10.19).

У цьому випадку необхідно обов'язково використовувати директиву `synthesis full_case`, щоб уникнути появи тригерів у процесі синтезу. На рис. 10.22 представлений результат синтезу такої Verilog-моделі без використання директиви синтезу `full_case`, а на рис. 10.23 – з використанням. У першому випадку відсутність директиви привело до появи тригерів.

Лістинг 10.19. Використання оператора `case` зі змінними значеннями для варіантів вибору.

```

module Encoder

```

```

(input [0:3] Bus, output reg [0:1] Address);
always @(Bus)
  casez(1'b1) //synthesis full_case
    Bus[0] : Address = 2'b00;
    Bus[1] : Address = 2'b01;
    Bus[2] : Address = 2'b10;
    Bus[3] : Address = 2'b11;
  endcase
endmodule

```

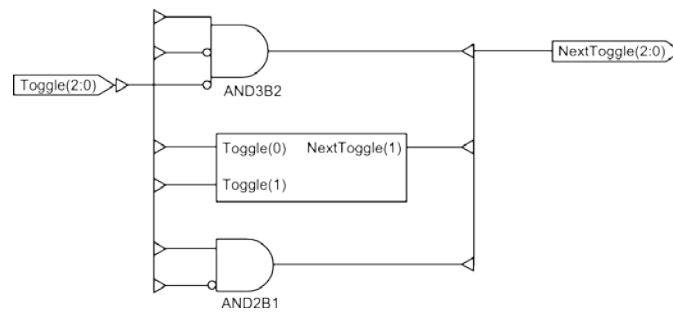


Рис. 10.21. Результат синтезу моделі (лістинг 10.18)

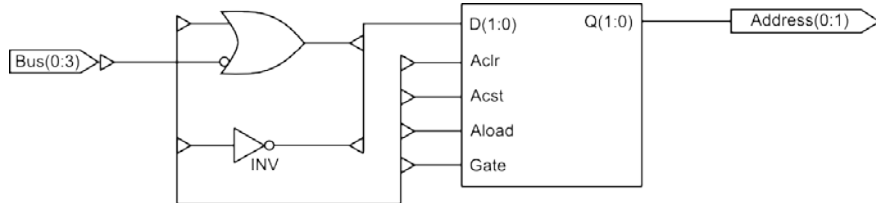


Рис. 10.22. Без використання директиви full_case

Уникнути появи тригера можна за допомогою ініціалізації змінної Address в операторі case:

```

always @(Bus)
begin
  Address = 2'b00; ...

```

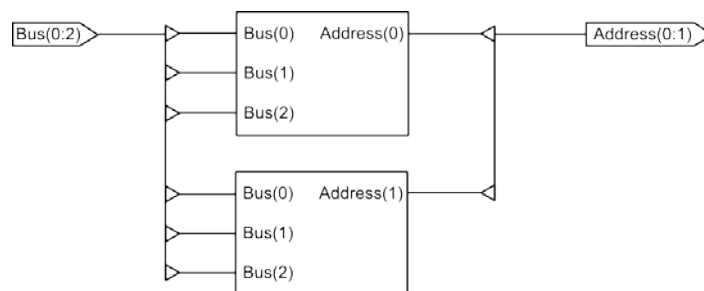


Рис. 10.23. З використанням директиви full_case

10.6. Оператори циклів

У мові Verilog є чотири цикли: while, for, forever і repeat [3, 4]. Повністю для синтезу, як це зазначено у стандарті, підтримується цикл for, оскільки в ньому точно і ясно задається число ітерацій (лістинг 10.20). Для логіки, що відповідає операторам, які записані в циклі, створюється стільки копій елементів, скільки ітерацій має цикл. Тому необхідно, щоб межі циклу були представлені константами.

Лістинг 10.20. Приклад використання циклу for.

```
module Multiplexer
    (input [1:0] Address,
     output reg [3:0] Line);
    integer J;
    always @(Address)
        for (J = 3; J >= 0; J = J-1)
            if (Address == J)
                Line[J] = 1;
            else
                Line[J] = 0;
endmodule
```

Для приклада з лістинга 10.20 у момент синтезу обробляється 4 копії оператора if:

```
if (Address == 3) Line[3] = 1; else Line[3] = 0;
if (Address == 2) Line[2] = 1; else Line[2] = 0;
if (Address == 1) Line[1] = 1; else Line[1] = 0;
if (Address == 0) Line[0] = 1; else Line[0] = 0;
```

Результат синтезу представлений на рис. 10.24.

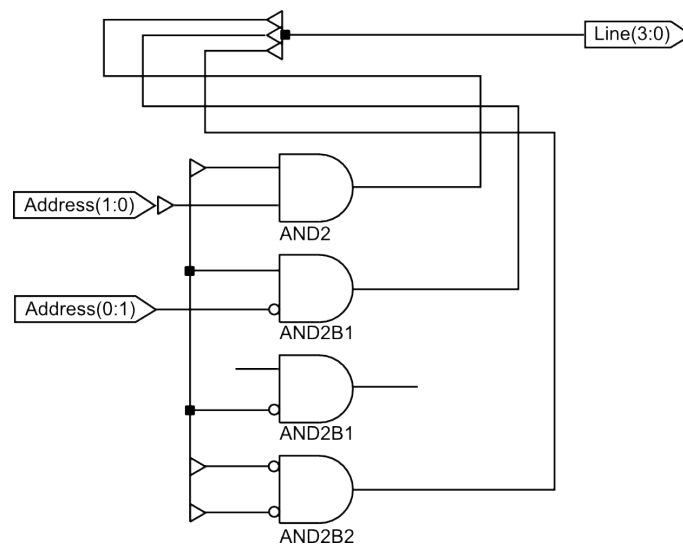


Рис. 10.24. Результат синтезу оператора for

Синтез інших трьох видів циклів, якщо й підтримується, то визначається тільки програмами синтезу, і в різних системах можуть бути присутні різні вимоги та обмеження щодо запису таких операторів.

Синтез оператора `repeat` можливий, якщо вказано константне значення числа повторень. Забороняється використання конструкцій виду `repeat (n) @(event)`,

де `n` - деяке константне значення, а `event` - подія сигналу або перевірка появи заданого фронту сигналу.

При використанні блоків `always` для повторення виконання операторів може бути застосований цикл `while`. Він виконує інші оператори, поки вираз, що перевіряється, не прийме значення `false`. Вираз, що перевіряється, може бути будь-яким виразом мови Verilog. Оператор `disable` не підтримується. Лістинг 10.21 представляє приклад використання циклу `while`, а рис. 10.25 - результат його синтезу.

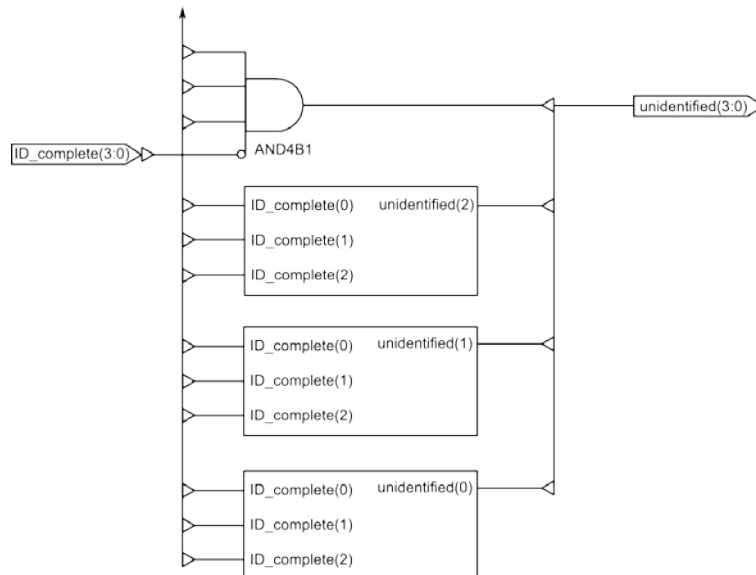


Рис. 10.25. Результат синтезу циклу `while`

Лістинг 10.21. Використання циклу `while`.

```
module test1(ID_complete, unidentified);
    input [3:0] ID_complete;
    output reg [3:0] unidentified;
    parameter P = 4;
    always @(ID_complete)
    begin : UNIDENTIFIED
        integer i;
        reg found;
        unidentified = 0;
```

```

i = 0;
found = 0;
while (!found && (i < P))
begin
    found = !ID_complete[i];
    unidentified[i] = !ID_complete[i];
    i = i + 1;
end
end
endmodule

```

10.7. Локальне використання змінних

Навіть якщо змінна оголошена глобально, але використовується локально під керуванням синхросигналу, то тригер для неї формуватися не буде [1- 4]. У наступному прикладі (лістинг 10.22) оператор `always` запускається по задньому фронту синхросигналу `Clk`.

Першою значення отримує змінна `Temp`, але воно відразу ж використовується в наступному неблокуючому операторі, який надає значення змінній `NextState`. Тому для даного прикладу буде згенерований тільки один тригер для змінної `NextState` (рис. 10.25).

Лістинг 10.22. Використання змінної без створення тригера.

```

module GlobalReg
    (input Clk, input [3:0] State, output reg [3:0] NextState);
    reg [3:0] Temp;
    always @(negedge Clk)
    begin
        Temp = State;
        NextState <= Temp;
    end
endmodule

```

Якщо в попередньому прикладі (див. лістинг 10.22) поміняти місцями блокуючий та неблокуючий оператор, що не блокує (лістинг 10.23), то значення змінної `Temp` буде зберігатися між ітераціями виконання блоку `always`, що приведе до появи додаткового тригера (рис. 10.27).

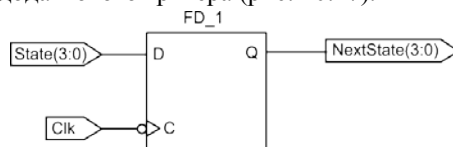


Рис. 10.26. Результат синтезу моделі (лістинг 10.22)

Лістинг 10.23. Використання змінної зі створенням тригера.

```

module GlobalReg
    (input Clk, input [3:0] State, output reg [3:0] NextState);
    reg [3:0] Temp;
    always @(negedge Clk)
    begin

```

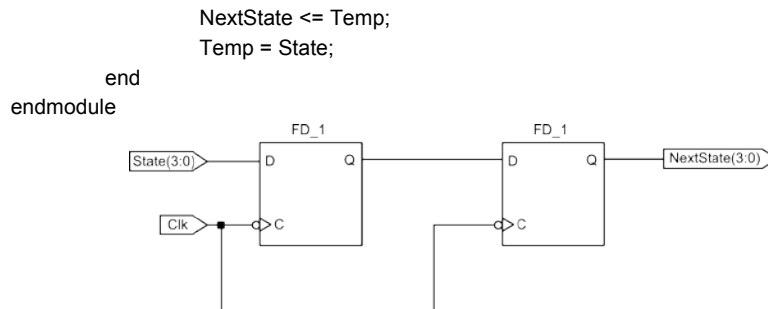


Рис. 10.27. Результат синтезу моделі (лістинг 10.23)

Для локально визначених у блоці змінних, так само як і для інших змінних, може генеруватися або не генеруватися тригер.

Тригер буде генеруватися, якщо значення змінної не повністю визначене в операторах керування if або case і якщо змінна використовується до присвоєння їй значення та зберігається до наступної ітерації виконання блоку.

10.8. Відмінність між блокуючими та неблокуючими змінними

Основна відмінність між цими двома формами оператора присвоєння лежить у принципі їхньої обробки під час моделювання. В блокуючих операторах змінна отримує значення до виконання наступного за ним оператора (немає відмінності між двома типами цих операторів, якщо він єдиний в операторі always і initial) [3, 4]. В неблокуючих операторах змінна, яка знаходиться в правій частині виразу, отримує нове значення тільки наприкінці циклу моделювання.

У наступному прикладі блок always містить три оператора: один блокуючий і два неблокуючих. Оскільки всі три присвоєння виконуються під керуванням синхросигналу, то для кожного з них генерується тригер із синхронізацією заднім фронтом. У лістингу 10.24 змінна RightShift використовує для обчислення своє попереднє значення, тому для неї при синтезі створюється тригер (рис. 10.28). Відмінність полягає в способі приєднання інформації до входів регістра. Оскільки для змінної RightShift використовується блокуючий оператор, то її нове значення стане доступним раніше виконання наступного оператора. Тому виходить, що дані із входу тригера RightShift повинні використовуватися для обчислення значення, що надходить на вхід SelectFirst. А оскільки змінна SelectFirst отримує значення в неблокуючому операторі, то на наступний тригер дані надходять із виходу SelectFirst.

Лістинг 10.24. Використання змінної без створення тригера.

```

module FB
    (input Clock, Strobe, Xflag, Mask,
     output reg RightShift, SelectFirst, CheckStop);
always @(negedge Clock)
begin
    RightShift = RightShift ^ Strobe;
    SelectFirst <= RightShift | Xflag;
end

```

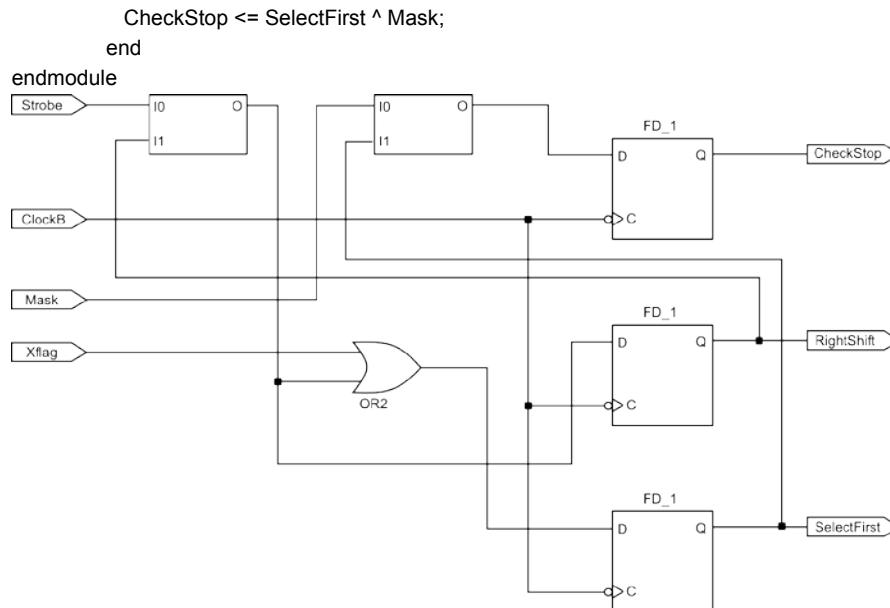



Рис. 10.28. Результат синтезу (лістинг 10.24)

Лістинги 10.25 і 10.26 ілюструють відмінність між блокуючими та неблокуючими операторами. У першому випадку FCR отримує значення за допомогою неблокуючого оператора, тому в операторі if використовується старе значення змінної FCR, а для FCR генерується тригер (рис. 10.29). У другому прикладі для FCR використовуються блокуючі оператори, тому при обчисленні значення змінної Claim застосовується оновлене значення FCR, а тригер для FCR не генерується, оскільки немає необхідності зберігати її значення (рис. 10.30).

Лістинг 10.25. Приклад із блокуючими та неблокуючими операторами.

```

module NonBlockingExample
  (input Clk, Merge, ER, Xmit, FDDI, output reg Claim);
  reg FCR;
  always @(posedge Clk)
  begin
    FCR<=ER|Xmit;
    if(Merge)
      Claim <= FCR & FDDI;
    else
      Claim <= FDDI;
  end
endmodule

```

Лістинг 10.26. Приклад із блокуючими та неблокуючими операторами.

```

module NonBlockingExample
  (input Clk, Merge, ER, Xmit, FDDI, output reg Claim);

```

```

    reg FCR;
always @(posedge Clk)
begin
    FCR = ER|Xmit;
    if(Merge)
        Claim = FCR & FDDI;
    else
        Claim = FDDI;
    end
endmodule

```

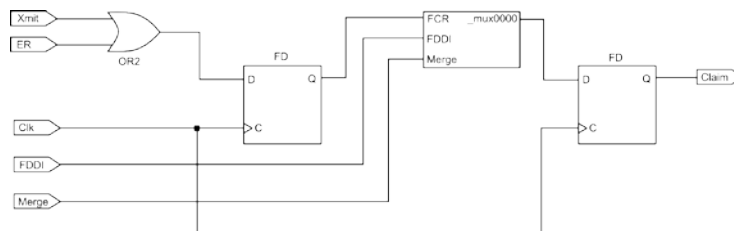


Рис. 10.29. Результат синтезу (лістинг 10.25)

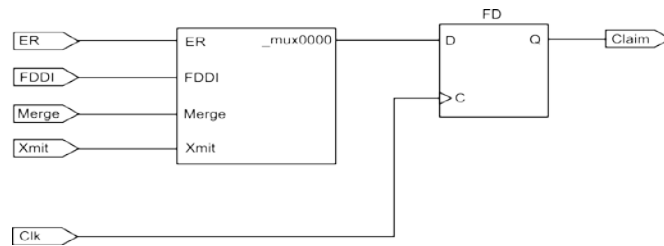


Рис. 10.30. Результат синтезу (лістинг 10.26)

10.9. Оператори Assign і Deassign

Синтез процедурних операторів assign/deassign підтримується в XST [3, 4]. Використання операторів assign/deassign представлено лістингом 10.27, а результат - рис. 10.31.

Лістинг 10.27. Приклад використання операторів assign/deassign.

```

module assign
    (input RST, SELECT, CLOCK, input [0:3] DATA_IN,
     output reg [0:3] STATE);
always @ (RST)
    if(RST) assign STATE = 4'b0;
    else deassign STATE;
always @ (posedge CLOCK)
    STATE = DATA_IN;
endmodule

```

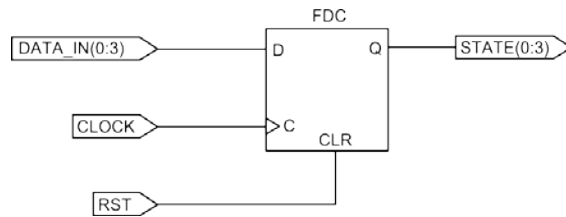


Рис. 10.31. Результат синтезу операторів assign/deassign в XST

На використання операторів assign/deassign накладаються обмеження:

- 1) Для одного сигналу допускається тільки одна конструкція assign/deassign (лістинг 10.28, а).
- 2) Оператори assign/deassign повинні бути описані в одному блоці always за допомогою конструкції if-else (лістинг 10.28, б).
- 3) Не можна застосовувати оператори assign/deassign, якщо вектор отримує значення в розділі (лістинг 10.28, в).

Синтез моделей, представлених лістингом 10.28, призведе до помилки синтезу.

Лістинг 10.28. Неприпустиме використання операторів assign/deassign.

```
a) module dflop (input RST, SET, CLOCK, DATA_IN,
    output reg STATE);
    always @ (RST) // block b1
        if(RST) assign STATE = 1'b0;
        else deassign STATE;
    always @ (SET) // block b1
        if(SET) assign STATE = 1'b1;
        else deassign STATE;
    always @ (posedge CLOCK) // block b2
        STATE = DATA_IN;
endmodule
```

Повідомлення про помилку

Analyzing top module <dflop>.

ERROR:Xst:855 - "m1.v" line 26: Unsupported procedural assignment for signal <STATE>.

```
б) module dflop (input RST, SET, CLOCK, DATA_IN,
    output reg STATE);
    always @ (RST or SET) // block b1
        case ({RST,SET})
            2'b00: assign STATE = 1'b0;
            2'b01: assign STATE = 1'b0;
            2'b10: assign STATE = 1'b1;
            2'b11: deassign STATE;
        endcase
    always @ (posedge CLOCK) // block b2
        STATE = DATA_IN;
endmodule
```

Повідомлення про помилку

Analyzing top module <dflop>.

ERROR:Xst:855 - "m1.v" line 25: Unsupported procedural assignment for signal <STATE>.

```

b) module assign (input RST, SET, CLOCK, DATA_IN,
    output reg STATE);
    always @ (RST) // block b1
        if (RST) assign STATE[0:7] = 8'b0;
        else deassign STATE[0:7];
    always @ (posedge CLOCK) // block b2
        if (SELECT)
            STATE [0:3]= DATA_IN[0:3];
        else
            STATE [4:7]= DATA_IN[4:7];
endmodule

```

10.10. Синтез підпрограм

Функція синтезується комбінаційною схемою з одним виходом, який може бути скалярним або вектором (лістинг 10.29, рис. 10.32) [1-4].

Лістинг 10.29. Декларація й використання функції.

```

module comb15
    (input [3:0] A, B,    input CIN,
     output [3:0] S, output COUT);
    wire [1:0] S0, S1, S2, S3;
    function signed [1:0] ADD (input A, B, CIN);
        reg S, COUT;

        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            ADD = {COUT, S};
        end
    endfunction
    assign S0 = ADD (A[0], B[0], CIN),
           S1 = ADD (A[1], B[1], S0[1]),
           S2 = ADD (A[2], B[2], S1[1]),
           S3 = ADD (A[3], B[3], S2[1]),
           S = {S3[0], S2[0], S1[0], S0[0]},
           COUT = S3[1];
endmodule

```

Функції завжди відповідає комбінаційна схема. Однак наступний код приведе до появи latch-тригера (рис. 10.33), оскільки змінна у буде зберігати своє значення між ітераціями виконання оператора always:

```

module latch1b (output reg y, input d, en);
    always @*
        y <= lat(d, en);
    function lat (input d, en);
        if (en) lat = d;
    endfunction
endmodule

```

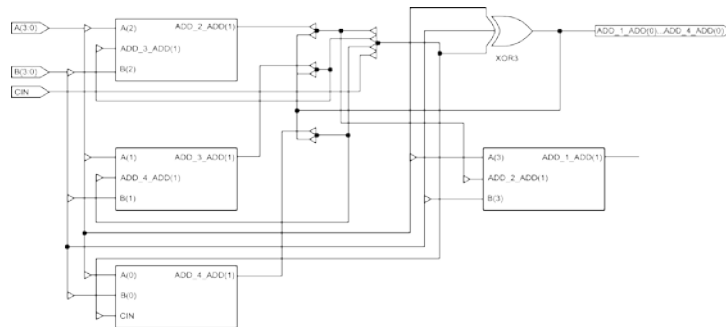


Рис. 10.32. Результат синтезу (лістинг 10.29)



Рис. 10.33. Latch-тригер

Модель буде поводитися як тригер-засувка, але синтезуватися буде комбінаційною схемою, що приведе до різних результатів моделювання до й після синтезу.

Завдання може бути представлено як комбінаційною, так і послідовною логікою залежно від контексту виконання (лістинг 10.30, рис. 10.34). У завданнях ігноруються всі тимчасові параметри. Не допускається використовувати іменні події.

Лістинг 10.30. Декларація й використання завдання.

```
module ex_task
    (input [3:0] A, B, input CIN,
     output reg [3:0] S, output reg COUT);
    reg [1:0] S0, S1, S2, S3;
    task ADD (input A, B, CIN, output reg [1:0] C);
        reg S, COUT;
    begin
        S = A ^ B ^ CIN;
        COUT = (A&B) | (A&CIN) | (B&CIN);
        C = {COUT, S};
    end
endtask
always @(A, B, CIN)
begin
    ADD (A[0], B[0], CIN, S0);
    ADD (A[1], B[1], S0[1], S1);
    ADD (A[2], B[2], S1[1], S2);
    ADD (A[3], B[3], S2[1], S3);
    S = {S3[0], S2[0], S1[0], S0[0]};
    COUT = S3[1];
end
endmodule
```

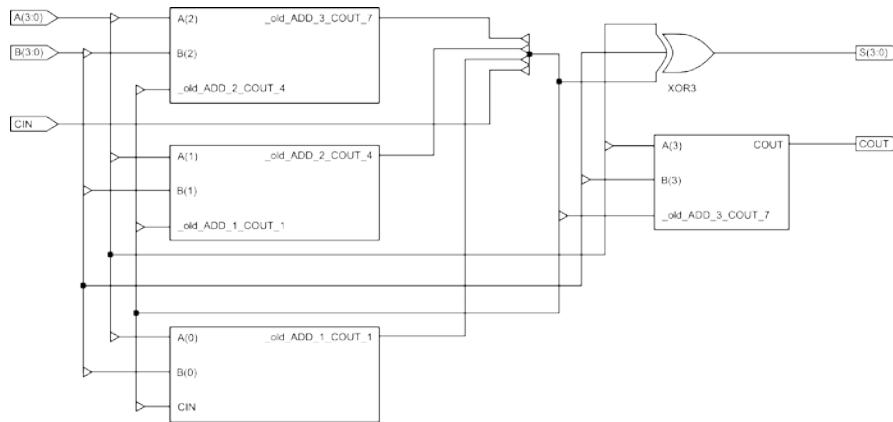


Рис. 10.34. Результат синтезу (лістинг 10.30)

10.11. Використання параметрів

У процесі синтезу параметри використовуються для визначення розмірів [1, 2]: розрядності ліній, кількості елементів, кодування станів автомата. Затримки в синтезованих моделях не задаються, тому параметри, що визначають тимчасові параметри пристрою, ігноруються.

Значення параметра може бути перевизначене в операторі реалізації модуля через конструкцію `#( )`. Також дозволене використання конструкції `defparam`, але не більше ніж на один рівень ієрархії.

У прикладі (лістинг 10.31) за допомогою параметра задається розрядність вхідної `in` та вихідної `out` шин даних у модулі `lpm_reg`, що за замовчуванням дорівнює 1. У модулі `top` в операторі реалізації копії модуля `lpm_reg` це значення перевизначається на 8. Рис. 10.35, а представляє результат синтезу 1-розрядного регістра з модуля `lpm_reg`, а рис. 10.35, б – результат синтезу модуля `top`, що формує 8-розрядний регістр із `lpm_reg` шляхом перевизначення параметра `SIZE`.

Лістинг 10.31. Визначення розрядності шин даних через параметр.

```
module lpm_reg
    #( parameter SIZE = 1)
    (input [ SIZE-1 : 0] in, input en, reset, clk,
     output reg [ SIZE-1 : 0] out);
always @(posedge clk, negedge reset)
    begin
        if (!reset) out <= 'b0;
        else if (en) out <= in;
        else out <= out;
    end
endmodule
module top
    (input [7:0] sys_in, output [7:0] sys_out,
     input sys_en, sys_reset, sysclk);
```

```

    lpm_reg #8 buf373 (sys_in, sys_en, sys_reset, sysclk, sys_out);
endmodule

```

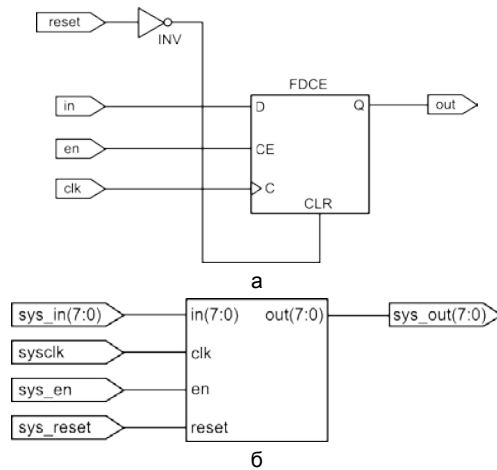


Рис. 10.35. Синтез 1- і 8-розрядних регістрів

10.12. Використання значення x

Присвоєння значення x сигналу в Verilog повідомляє програмі моделювання, що сигнал перебуває в невизначеному стані, а програма синтезу розглядає таку величину як ту, що не має значення [3, 4]. Таким чином, під час синтезу виконується оптимізація вентилів, що формують вихідне значення, рівне x. Це також означає, що результати моделювання для сигналів, що отримують x, до й після синтезу будуть різні.

У прикладі (лістинг 10.32) вихід мультиплексора ініціалізується в x, а потім обновляється, ґрунтуючись на значенні сигналу sel. У проекті маєтись на увазі, що вхідний сигнал sel ніколи не буде дорівнювати 2'b11. Якщо в результаті моделювання моделі виникне така ситуація, то вихідне значення у буде дорівнювати x, у той час як для моделі після синтезу у - визначене значення 0 або 1.

Лістинг 10.32. Модель мультиплексора з використанням x

```

module mux3_x (output reg y, input [2:0] a, input [1:0] sel);
always @* begin
    y = 1'bx;           // synthesis "don't-care"
    case (sel)
        2'b00: y = a[0];
        2'b01: y = a[1];
        2'b10: y = a[2];
    endcase
end
endmodule

```

Використання значення x може бути корисним для пошуку помилок при виконанні постсинтезного моделювання. Це також дозволяє виконувати оптимізацію схеми проекту, що генерується, ґрунтуючись на невизначених (don't care) значеннях змінних (рис. 10.36).

10.13. Контрольні запитання

1. Знаковий чи беззнаковий результат арифметичної операції присвоюється змінній типу reg?

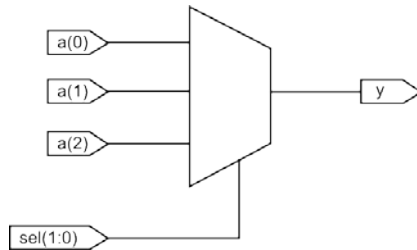


Рис. 10.36. Використання x дозволило уникнути появи тригера

2. Скільки тригерів буде згенеровано з наступного коду, якщо допустити, що кожна змінна відповідає одному біту?

```

always @(posedge CLK)
begin
    STAGE1 = A&B;
    STAGE2 <= STAGE1|C;
end
  
```

3. Чи буде синтезований тригер-засувка з наступного коду? Якщо так, то скільки?

```

always @(A, EN)
begin: Design
    integer Tmp;
    if (EN)
    begin
        Q <= Tmp;
        Tmp = A;
    end
end
  
```

4. Яка логіка буде синтезована з наступного коду?

```

reg [3:0] Q;
wire [3:0] Shift;
wire [3:0] D;
always @(posedge Clk)
begin
    Q <= Shift >> D;
end
  
```

5. Чи будуть синтезовані тригери або тригер з наступного коду?

```

always @(Sel)
case (Sel) // synthesis full_case
  
```



```

2'b01: Out <= 2'b10;
2'b10: Out <= 2'b01;
endcase

```

6. Як розв'язується конфліктна ситуація при використанні оператора `case`, якщо значення вибору в різних гілках збігаються або перекриваються?

7. Яка логіка генерується з наступного коду?

```

wire en, d;
reg q;
always
begin
    wait (en);
    #10 q <= d;
end

```

8. Наступний код моделює два регістри. Результати моделювання після синтезу не збіглися з результатами функціонального моделювання. Чому?

```

always @(posedge CLK)          always @(posedge CLK)
    if (CLK) A = A_in;          if (CLK) B = B_in;
    else A = B;                  else B = A;

```

9. Яка логіка буде згенерована з наступного коду?

```

always @ (Din, E)
    if(E) assign Q = Din;
    else deassign Q;

```

10. Яка логіка буде згенерована при синтезі наступного коду?

```

wire [3:0] D, sel;
reg Q;
always @(R, sel)
casez (sel) //synthesis parallel_case
    4'b???1: Q <= R[3];
    4'b??1?: Q <= R[1];
    4'b?1??: Q <= R[0];
    4'b1???: Q <= R[2];
    default : Q <= 1'bx;
endcase

```

11. Який знак буде мати результат наступної операції?

```

integer A, B, C;
always @(A, B)
    C <= A + B;

```

12. Чому результати моделювання після синтезу не збіглися з результатами моделювання до синтезу?

```

always @(Ready)
    Stop <= Ready & Wait;

```

13. Комбінаційною чи послідовною логікою реалізуються оператори безперервного призначення `assign`?

14. Що буде згенеровано в результаті синтезу наступного коду?

201

```
always @(i*)
begin
    if (in1) O = in1;
    else O <= in2;
end
```

15. Що буде згенеровано в результаті синтезу наступного коду?

```
if (in2) begin
    out1[0] = in0;
    out1[1] <= in1;
end
else begin
    out1[0] = in2;
    out1[1] <= in3;
end
```

16. Що буде згенеровано в результаті синтезу наступного коду?

```
module Logic
    (input [1:0] T, output reg [1:0] Res);
always @(*)
    case (T)
        2'b01: Res = 2'b10;
        2'b10: Res = 2'b01;
    endcase
endmodule
```

17. Що буде згенеровано в результаті синтезу наступного коду?

```
module Logic
    (input [1:0] T, output reg [1:0] Res);
always @(*)
    case (T)
        2'b01: Res = 2'b10;
        2'b10: Res = 2'b01;
        default: Res = 2'b01;
    endcase
```

18. Що буде згенеровано в результаті синтезу наступного коду?

```
module Logic
    (input [1:0] D, output reg [1:0] Q);
always @(*)
    case (D) //synthesis full_case
        2'b01: Q = 2'b10;
        2'b10: Q = 2'b01;
    endcase
endmodule
```

11. ШАБЛони СТАНДАРТНИХ КОМПОНЕНТІВ

11.1. Комбінаційні компоненти

11.1.1. Мультиплексори.

Модель мультиплексора може бути описана за допомогою оператора if (лістинг 11.1, а) і case (лістинг 11.1, б) [1-4]. Використання того або іншого оператора є справою смаку й принципового значення для сформованої в результаті схеми не має. В обох випадках результат синтезу однаковий (рис. 11.1). Зручніше, на думку авторів, використовувати оператор case.

Лістинг 11.1. Мультиплексор 4-в-1.

<pre> a) module mux_if (input a,b,c,d, input [1:0] s, output reg o); always @(a, b, c, d, s) if (s == 2'b00) o = a; else if (s == 2'b01) o = b; else if (s == 2'b10) o = c; else o = d; endmodule </pre>	<pre> б) module mux_case (input a,b,c,d, input [1:0] s; output reg o); always @(a, b, c, d, s) case (s) 2'b00 : o = a; 2'b01 : o = b; 2'b10 : o = c; default : o = d; endcase endmodule </pre>

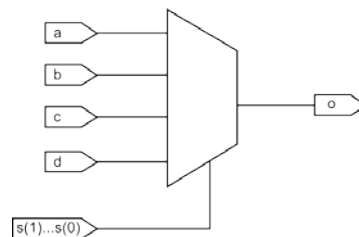


Рис. 11.1. RTL-схема мультиплексора

Мультиплексор може бути реалізований за допомогою тристабільних буферів. У цьому випадку число керуючих входів дорівнює числу входів даних, при цьому використовується позиційний код для вибору активного входу (лістинг 11.2, рис. 11.2).

Лістинг 11.2. Мультиплексор 4-в-1 на тристабільній логіці.

```

module mux_tri_st
    (input a,b,c,d,
     input [3:0] s,
     output o);
    assign o = s[3] ? a : 1'bz;
    assign o = s[2] ? b : 1'bz;
    assign o = s[1] ? c : 1'bz;
    assign o = s[0] ? d : 1'bz;
endmodule

```

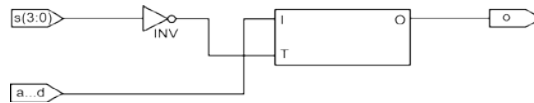


Рис. 11.2. RTL-схема мультиплексора на тристабільних буферах

11.1.2. Дешифратори.

Моделі дешифраторів створюються за допомогою оператора case. Пропонується два приклади дешифраторів із прямими (лістинг 11.3, а) та інверсними виходами (лістинг 11.3, б). Результат RTL-синтезу для обох моделей однаковий і представлений рис. 11.3.

Лістинг 11.3. Verilog-модель дешифратора (One-Hot).

<pre> a) module dec1 (input [2:0] sel, output reg [7:0] res); always @(sel, res) case (sel) 3'b000 : res = 8'b00000001; 3'b001 : res = 8'b00000010; 3'b010 : res = 8'b00000100; 3'b011 : res = 8'b00001000; 3'b100 : res = 8'b00010000; 3'b101 : res = 8'b00100000; 3'b110 : res = 8'b01000000; default : res = 8'b10000000; endcase endmodule </pre>	<pre> б) module dec0 (input [2:0] sel, output reg [7:0] res); always @(sel, res) case (sel) 3'b000 : res = 8'b111111110; 3'b001 : res = 8'b111111101; 3'b010 : res = 8'b111111011; 3'b011 : res = 8'b111110111; 3'b100 : res = 8'b111011111; 3'b101 : res = 8'b110111111; 3'b110 : res = 8'b101111111; default : res = 8'b011111111; endcase endmodule </pre>

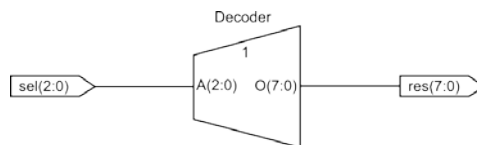


Рис. 11.3. RTL-схема дешифратора

Якщо в дешифраторі використовуються не всі можливі комбінації 2^n , то застосування конструкції виду `default : res = 8'bxxxxxxxx`; допоможе отримати оптимальну комбінаційну схему, як це представлено в лістингу 11.4.

Лістинг 11.4. Verilog-модель дешифратора зі значенням за замовчуванням.

```

module dec
    (input [2:0] sel, output reg [7:0] res);
always @(sel , res)
    case (sel)
        3'b000 : res = 8'b00000001;
        3'b001 : res = 8'b00000010;
        3'b010 : res = 8'b00000100;
        3'b011 : res = 8'b00001000;
        3'b100 : res = 8'b00010000;
        3'b101 : res = 8'b00100000;

```

```

        // 110 and 111 selector values are unused
        default : res = 8'bxxxxxxx;
    endcase
endmodule

```

11.1.3. Пристрої зсуву.

Xilinx проектує логічні схеми зсуву як комбінаційну схему із двома входами й одним виходом (рис. 11.4):

- 1) перший вхід DI – дані, що зрушуються;
- 2) другий вхід SEL визначає число розрядів, на які виконується зсув.

На вихід SO надходить результат операції зсуву (лістинг 11.5, рис. 11.5). Варто використовувати один тип операції зсуву. Значення n для операції зсуву повинне збільшуватися або зменшуватися тільки на 1 для кожного наступного двійкового значення селектора. Значення n може бути тільки позитивним. Всі значення селектора повинні бути присутніми.

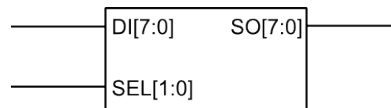


Рис. 11.4. Інтерфейс пристрою зсуву

Лістинг 11.5. Verilog-модель зсувача

```

module lshift
    (input [7:0] DI, input [1:0] SEL,
     output reg [7:0] SO);
always @(DI, SEL)
    case (SEL)
        2'b00 : SO <= DI;
        2'b01 : SO <= DI << 1;
        2'b10 : SO <= DI << 2;
        default : SO <= DI << 3;
    endcase
endmodule

```

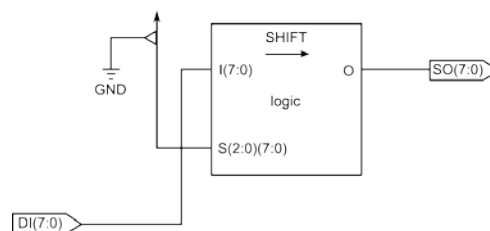


Рис. 11.5. RTL-Схема пристрою зрушення

11.1.4. Арифметичні пристрої.

Для синтезу дозволені арифметичні операції: додавання, віднімання й множення, а також операція порівняння.

Операнди можуть бути знакові й беззнакові. Розподіл дозволений тільки якщо дільник дорівнює 2^n , де n - будь-яке позитивне число або 0, і апаратно реалізується зсувом на n розрядів вправо.

Шаблони опису арифметичних пристроїв представлені: беззнаковими й знаковими суматорами без переносу (лістинг 11.6, рис.11.6), з переносами (лістинг 11.7, рис.11.7), пристроєм віднімання (лістинг 11.8, рис.11.8), компаратором (лістинг 11.9, рис.11.9), пристроєм множення (лістинг 11.10, мал.11.10), пристроєм розподілу (лістинг 11.11).

Лістинг 11.6. Verilog-моделі суматора без переносу.

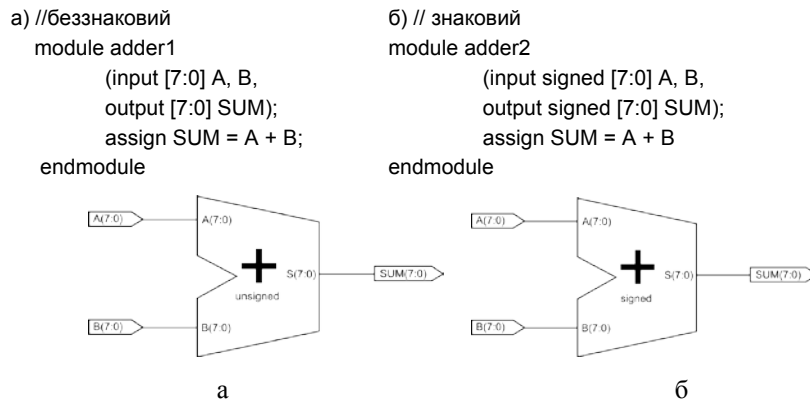
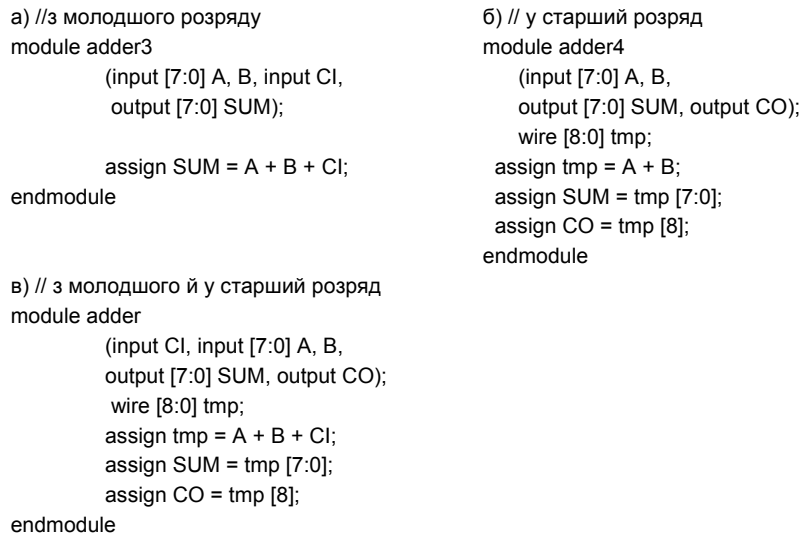


Рис. 11.6. RTL-схема суматора без переносу

Лістинг 11.7. Verilog-модель суматора з переносом.



Лістинг 11.8. Verilog-модель пристрою віднімання.

```

module subtr
    (input [7:0] A, B, output [7:0] RES);

```

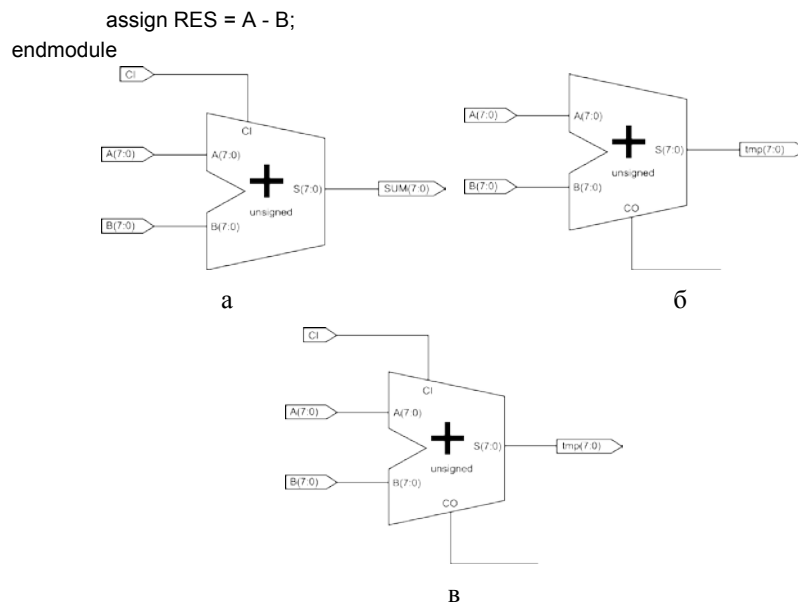


Рис. 11.7. RTL-схема суматора з переносом

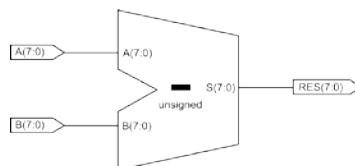


Рис. 11.8. RTL-схема пристрою віднімання

Лістинг 11.9. Verilog-модель компаратора.

```

module compare
  (input [7:0] A, B, output CMP);
  assign CMP = A >= B ? 1'b1:1'b0;
endmodule

```

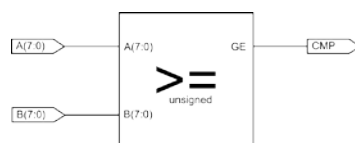


Рис. 11.9. RTL-схема компаратора

Лістинг 11.10. Verilog-модель беззнакового множника.

```

module mult
  (input [7:0] A, input [3:0] B, output [11:0] RES);
  assign RES = A * B;
endmodule

```

Лістинг 11.11. Verilog-модель дільника.

```

module divider

```

```

(input [7:0] DI, output [7:0] DO);
assign DO = DI / 2;
endmodule

```

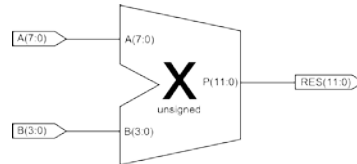


Рис. 11.10. RTL-схема пристрою множення

11.1.5. Спільне використання ресурсів (Resource Sharing).

Мета - зменшити розмір схеми, ґрунтуючись на принципі, що два подібних арифметичних пристрої можуть бути реалізовані на одній схемі, якщо вона ніколи не використовується ними одночасно. Вибір того або іншого стилю визначається засобами середовища проектування. XST підтримує спільне використання ресурсів для суматорів, від'ємників, суматорів/від'ємників і множників.

Модель найпростішого АЛП (рис. 11.11, лістинг 11.12) застосовується для ілюстрації синтезу зі спільним і роздільним використанням ресурсів (рис. 11.12 - 11.13).

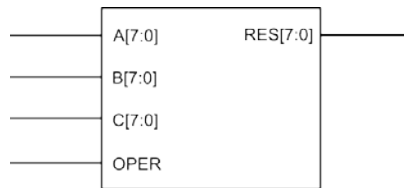


Рис. 11.11. Спільне використання ресурсів для АЛП

Лістинг 11.12. Verilog-модель найпростішого АЛП.

```

module addsub
    (input OPER, input [7:0] A, B, C,
     output reg [7:0] RES);
always @(A or B or C or OPER)
    if (OPER==1'b0) RES = A + B;
    else RES = A - C;
endmodule

```

11.2. Послідовна логіка

Далі представлені шаблони опису тригерів.

11.2.1. Тригери із синхронізацією по фронту.

Модель D-тригера, що керується переднім фронтом, представлена лістингом 11.13, а), результат синтезу - рис. 11.14, а. Інтерфейс пристрою включає такі порти: C - вхід синхронізації, D - вхід даних, Q - вихід даних.

Наступна модель (лістинг 11.13, б), рис. 11.14, б) представляє пристрій із синхронізацією по задньому фронту й використанням сигналу асинхронного

скидання CLR. Останні дві моделі ілюструють застосування синхронної установки S (лістинг 11.13, в), рис. 11.14, в) і сигналу дозволу синхронізації CE (лістинг 11.13, г), рис. 11.14, г).

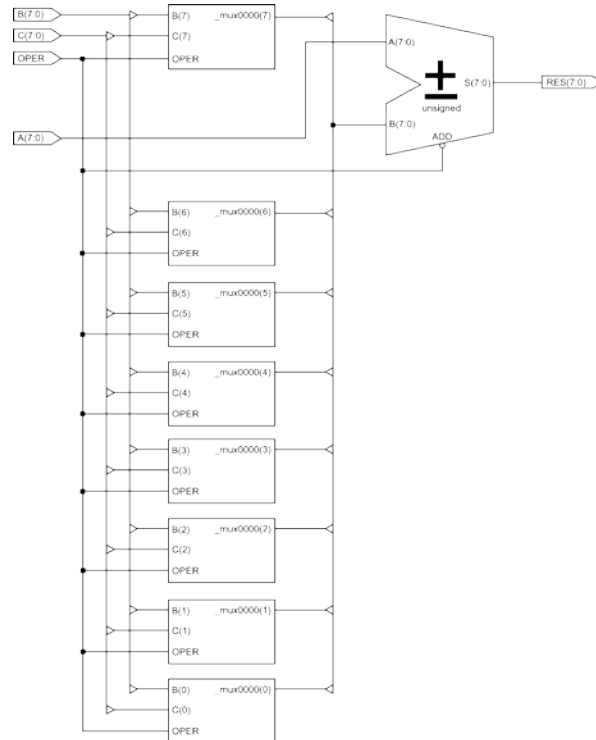


Рис. 11.12. RTL-схема АЛП зі спільним використанням ресурсів

Лістинг 11.13. Verilog-модель D-тригера.

а) // D-Тригер

```
module flop1
    (input C, D,
     output reg Q);
    always @(posedge C)
        Q = D;
endmodule
```

в) // Із синхронною установкою

```
module flop3
    (input C, D, S,
     output reg Q);
    always @(posedge C)
        if (S) Q = 1'b1;
        else Q = D;
endmodule
```

б) // З асинхронним скиданням

```
module flop2
    (input C, D, CLR,
     output reg Q);
    always @(negedge C, posedge CLR)
        if (CLR) Q = 1'b0;
        else Q = D;
endmodule
```

г) // Із сигналом дозволу синхронізації

```
module flop4
    (input C, D, CE,
     output reg Q);
    always @(posedge C)
        if (CE)
            Q = D;
endmodule
```

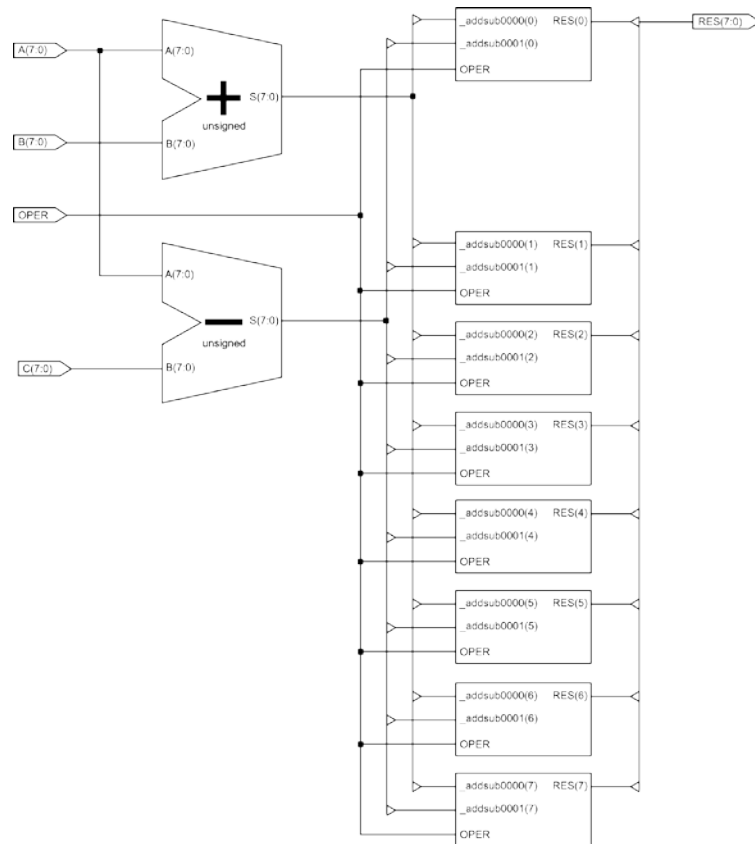


Рис. 11.13. RTL-схема АЛП без спільного використання ресурсів

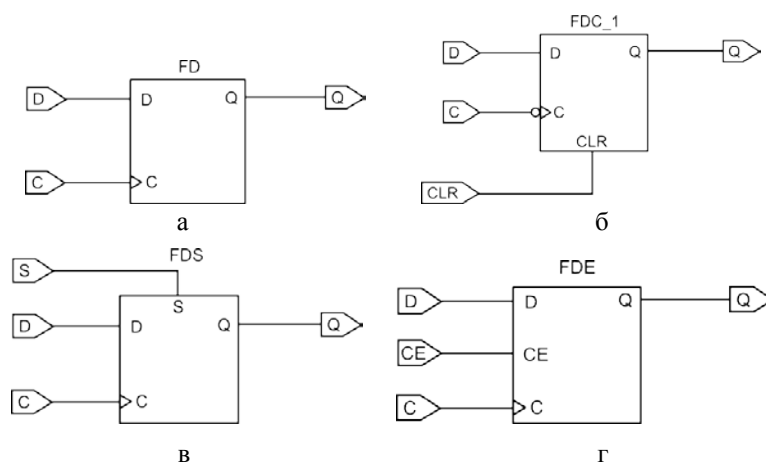


Рис. 11.14. Синтез моделей D-тригера із синхронізацією по фронту

11.2.2. Тригери-засувки.

Тригери-засувки описуються за допомогою блоку `always` [1-4] (лістинг 11.14, а), рис. 11.15, а), можуть мати асинхронний сигнал установки або скидання (лістинг 11.14, б), рис. 11.15, б). Список чутливості оператора `always` може мати скорочену форму `always @(*)`.

Лістинг 11.14. Verilog-модель D-тригера-засувки.

```

а) // D-тригер-засувка
module latch1
    (input G, D,
     output reg Q);
always @(G, D)
    if (G)
        Q = D;
endmodule

б) // D-тригера-засувки зі скиданням
module latch2
    (input G, D, CLR,
     output reg Q);
always @(G, D, CLR)
    if (CLR) Q = 1'b0;
    else if (G) Q = D;
endmodule

```

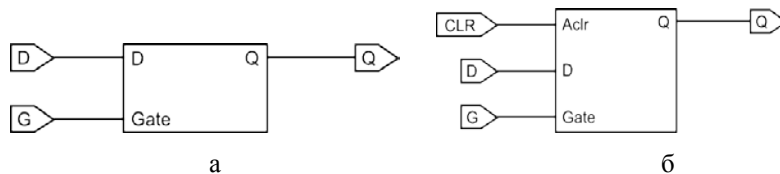


Рис. 11.15. Синтез D-тригера-засувки

11.2.3. Лічильники.

Найпростіший 4-розрядний лічильник з асинхронним скиданням представлений лістингом 11.15, а). Якщо сигнал CLR дорівнює 1, то виконується скидання пристрою в нульовий стан, інакше по кожному передньому фронту синхросигналу значення лічильника збільшується на 1. Результат RTL-синтезу зображений на рис. 11.16. Лістинг 11.15, б) - це модель лічильника, що параметризується, розрядність якого задається параметром `MAX_SQRT`, а верхня межа рахунку - параметром `MAX`. Модель із синхронним входом S, по якому значення лічильника скидається в 1111, представлений лістингом 11.16, а результат синтезу даного пристрою зображений на рис. 11.17. Ще один приклад лічильника (лістинг 11.17, рис. 11.18) має вхід паралельного завантаження D, що може бути використаний для встановлення початкового стану, коли сигнал `ALOAD=1`.

Лістинг 11.15. Беззнаковий лічильник з асинхронним скиданням.

```

а) module counter
    (input C, CLR, output reg [3:0] Q);
always @(posedge C, posedge CLR)
    if (CLR) Q = 4'b0000;
    else Q = Q + 1'b1;
endmodule

б) module v_counters_8
    #(parameter MAX_SQRT = 4,

```

```

MAX = MAX_SQRT*MAX_SQRT)
(input C, CLR,
output [MAX_SQRT-1:0] Q);
reg [MAX_SQRT-1:0] cnt;
always @(posedge C, posedge CLR)
  if (CLR)
    cnt <= 0;
  else
    cnt <= (cnt + 1) %MAX;
    assign Q = cnt;
endmodule

```

Лістинг 11.16. Беззнаковий лічильник, що віднімає, із синхронною установкою.

```

module counter
  (input C, S, output reg [3:0] Q);
always @(posedge C)
  if (S) Q = 4'b1111;
  else Q = Q - 1'b1;
endmodule

```

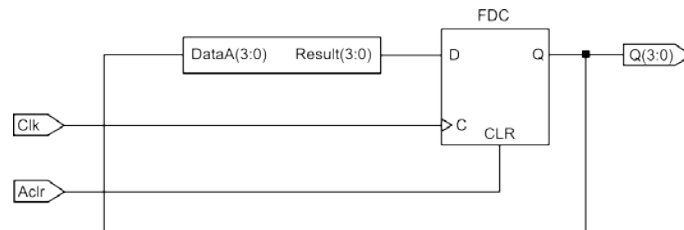


Рис. 11.16. RTL-схема беззнакового лічильника з асинхронним скиданням

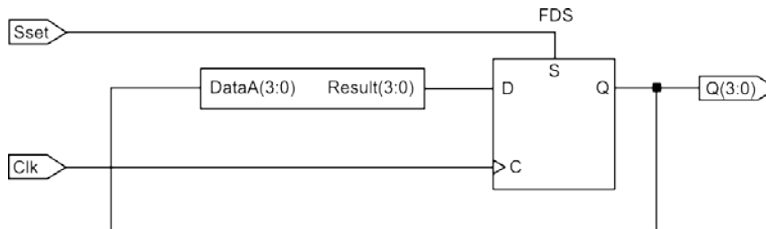


Рис.11.17. RTL-схема беззнакового лічильника, що віднімає

Лістинг 11.17. Беззнаковий лічильник з паралельним завантаженням.

```

module counter
  (input C, ALOAD, input [3:0] D,
   output reg [3:0] Q);
always @(posedge C)
  if (ALOAD) Q = D;
  else Q = Q + 1'b1;
endmodule

```

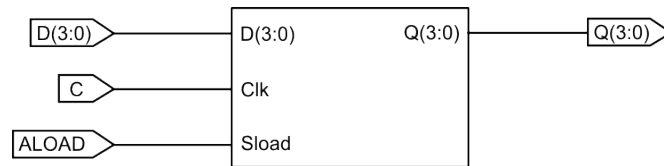


Рис. 10.18. RTL-схема беззнакового лічильника з паралельним завантаженням

Акумулятор відрізняється від лічильника операндами, що обчислюють наступне значення. У лічильнику це його поточний стан і константа - крок рахунку. Наприклад, $A \leq A + 1$. В акумуляторі перший оператор - його поточний стан, а другий - змінна: $A \leq A + B$. Акумулятор може працювати на збільшення свого значення, на зменшення або в обох напрямках. Приклад (лістинг 11.18) представляє чотирирозрядний підсумовуючий акумулятор з асинхронним скиданням CLR. Для схем, що представляють в XST результати RTL-синтезу, використовується спеціальний макрос (рис. 11.19).

Лістинг 11.18. Акумулятор з асинхронним скиданням.

```
module accum
    (input C, CLR, input [3:0] D,
     output reg [3:0] Q);
    always @(posedge C, posedge CLR)
        if (CLR) Q = 4'b0000;
        else Q = Q + D;
endmodule
```

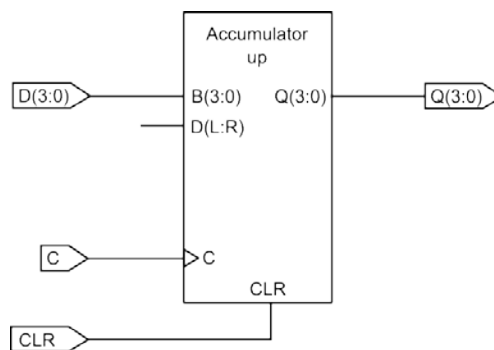


Рис. 11.19. Акумулятор з асинхронним скиданням

11.2.4. Зсувні регістри.

Існує кілька способів реалізації зсуву при створенні зсувних регістрів, вибір яких визначається перевагою проектувальника й розв'язуванням завданням:

1. За допомогою оператора конкатенації. Наприклад, зсув вліво (убік старших розрядів) на одну позицію із занесенням значення SI у молодший розряд може бути реалізований оператором:

```
shreg = {shreg [6:0], SI};
```

2. За допомогою операторів циклу. Наприклад,

```
for (i = 0; i <= 6; i = i + 1) in 0 to 6
```

```

begin
    shreg(i+1) <= shreg(i);
end loop;
shreg(0) <= SI;

```

3. За допомогою визначених у мові операторів зсуву << i >>. Прикладом використання цих операторів є лістинг 11.19, що реалізує зсувний 8-розрядний регістр із послідовним входом SI і послідовним виходом SO (лістинг 11.19, а), рис. 11.20, а), із входом дозволу (лістинг 11.19, б), рис. 11.20, б), з асинхронним скиданням (лістинг 11.19, в)), двонаправлений (лістинг 11.19, г), рис. 11.20, в)).

Лістинг 11.19. Моделі 8-розрядного зсувного регістра.

а) // Послідовний регістр <pre> module shift1 (input C,SI, output SO); reg [7:0] tmp; always @(posedge C) begin tmp = tmp << 1; tmp[0] = SI; end assign SO = tmp[7]; endmodule </pre>	б) // Із входом дозволу <pre> module shift2 (input C,SI, CE, output SO); reg [7:0] tmp; always @(negedge C) if (CE) begin tmp = tmp << 1; tmp[0] = SI; end assign SO = tmp[7]; endmodule </pre>
в) //Послідовний регістр зі скиданням <pre> module shift3 (input C,SI,CLR, output SO); reg [7:0] tmp; always @(posedge C, posedge CLR) if (CLR) tmp = 8'b00000000; else tmp = {tmp[6:0], SI}; assign SO = tmp[7]; endmodule </pre>	г) // Двонаправлений зсувний регістр <pre> module shift4 (input C,SI,LEFT_RIGHT, output [7:0] PO); reg [7:0] tmp; always @(posedge C) if (LEFT_RIGHT==1'b0) tmp = {tmp[6:0], SI}; else tmp = {SI, tmp[7:1]}; assign PO = tmp; endmodule </pre>

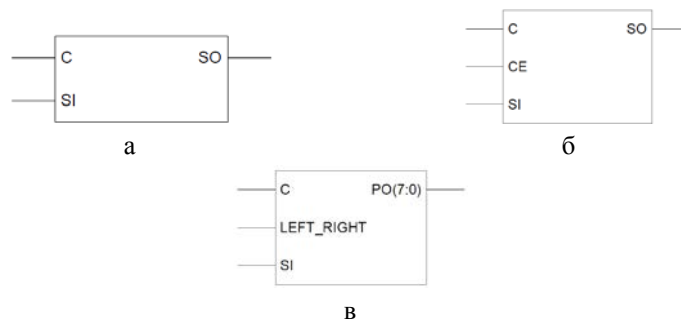


Рис. 11.20. Послідовний зсувний регістр

11.3. Автомати

У загальному випадку керуючий автомат може бути представлений комбінацією автоматів Мілі й Мура (рис. 11.21). При цьому структура пристрою ділиться на три основні блоки, двом з яких відповідає комбінаційна схема (функції переходів і виходів), а третьому – послідовна (регістр станів). Входами для функції переходів є поточний стан автомата й входи. Функція виходів для автомата Мура залежить від поточного стану пристрою й вхідних сигналів [1, 2].

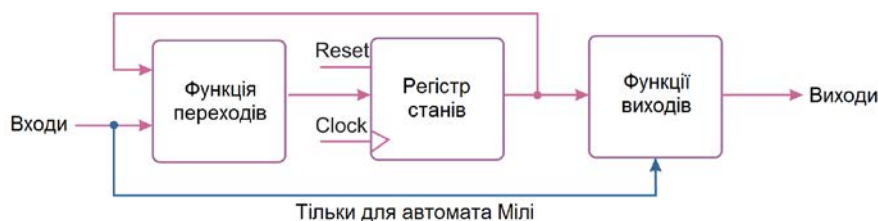


Рис. 11.21. Структурна схема керуючого автомата

Для опису автомата можуть бути використані різні стилі й заходи. Програми синтезу мають можливість визначати в коді компонентів автомата й застосовувати специфічні для даного типу пристроїв методи оптимізації. В основному вони торкаються способу кодування станів. Як правило, використовуються три способи кодування: послідовний двійковий код, унарний (або позиційний) код і код Грея. Вибір способу кодування залежить від числа станів у пристрої.

Бажано при опису моделі автомата окремо описувати комбінаційну частину пристрою, а окремо - послідовну. Це спростить процес тестування пристроїв і скоротить імовірність внесення помилок. Для написання якісного коду варто ознайомитися з документацією використовуваної системи синтезу й розглянути пропонувані нею шаблони написання коду.

В Verilog немає типів перелічення як в VHDL. Для запису станів автомата і їхніх двійкових кодів використовують параметри `parameter` або директиву компіляції `define`.

Приклад автомата Мілі (рис. 11.22, лістинг 11.20) представляє найпоширеніший спосіб опису коду моделі, що складається із двох блоків `always`. Перший блок `always` моделює регістр станів, другий відповідає комбінаційним функціям переходів і виходів. Стани автомата визначені через параметри. Кодування станів виконується трьохрозрядним унарним кодом. Крім цього, використовується параметр `deflt=3'bxxx`, що дозволяє отримати оптимальну комбінаційну схему функції переходів.

Результат синтезу представлений на рис. 11.23. Перша схема (рис. 11.23, а) відповідає всьому пристрою й включає автомат з функціями виходу, друга (рис. 11.23, б) - розшифровує блок FSM, що містить регістр станів і реалізацію функції переходів. Лістинг 11.21 представляє фрагмент звіту про синтез схеми автомата. З нього видно, що система синтезу виявила автомат і зміни-

ла запропоноване Verilog-моделлю кодування станів. Замість унарного коду використовується послідовний, регістр станів автомата скорочений до двох розрядів.

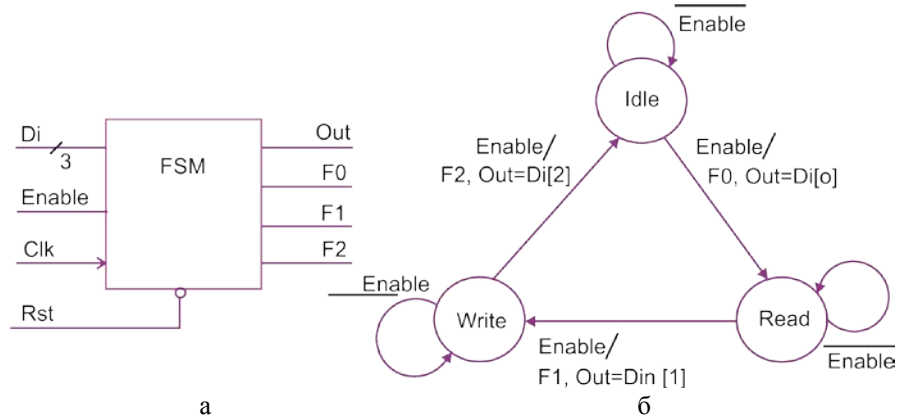


Рис. 11.22. Інтерфейс і граф переходу керуючого автомата

Лістинг 11.20. Керуючий автомат із двома блоками always.

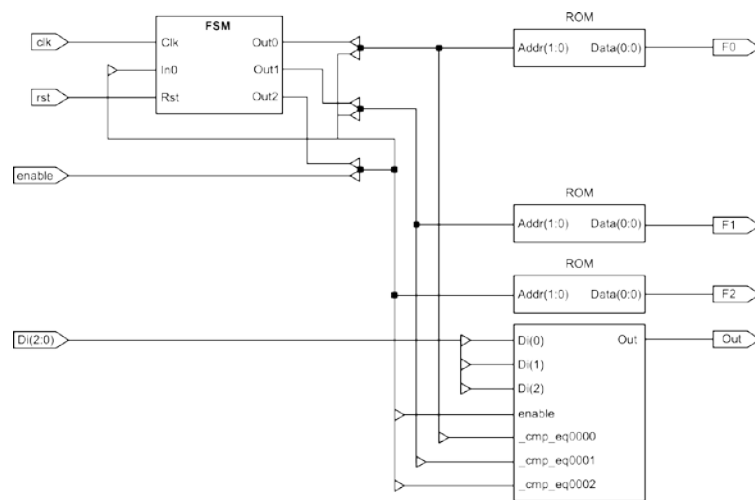
```
module FSM1
// Визначення міток станів автомата і їхнє кодування
#( parameter deflt=3'bxxx, idle=3'b001,
  read=3'b010, write=3'b100)
(input clk, rst, enable, input [2:0] Di,
  output reg Out, F0, F1, F2);
reg [2:0] state, next_state;
// Оператор always, що описує послідовну логіку
always @(posedge clk, negedge rst)
  if (!rst) state <= idle;
  else state <= next_state;
// Оператор always, що описує комбінаційну логіку
always @(*)
begin
// Значення виходів автомата за замовчуванням
F0 <= 1'b0; F1 <= 1'b0; F2 <= 1'b0;
Out <= 1'b0;
case (state)
idle : if (enable) begin
F0 <= 1'b1;
Out <= Di[0];
next_state <= read;
end
else next_state <= idle;
read : if (enable) begin
F1 <= 1'b1;
Out <= Di[1];
next_state <= write;
end
write : if (enable) begin
F2 <= 1'b1;
Out <= Di[2];
next_state <= idle;
end
endcase
end
endmodule
```



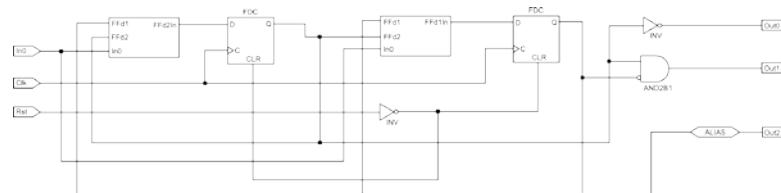
```

        end
        else next_state <= read;
write : if (enable) begin
    F2 <= 1'b1;
    Out <= Di[2];
    next_state <= idle;
end
else next_state <= write;
default : next_state <= deflt;
endcase
end
endmodule

```



a



6

Рис. 11.23. Результат синтезу моделі автомата із двома блоками always

Лістинг 11.21. Фрагмент звіту про синтез XST.

Synthesizing Unit <FSM1>.

Found finite state machine <FSM_0> for signal <state>.

[illegible]

States	3
Transitions	6
Inputs	1

```

| Outputs          | 3          |
| Clock            | clk (rising_edge) |
| Reset            | rst (negative) |
| Reset type       | asynchronous |
| Reset State      | 001        |
| Encoding         | automatic   |
| Implementation   | LUT        |
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
Found 4x1-bit ROM for signal <F2>.
Found 4x1-bit ROM for signal <F1>.
Found 4x1-bit ROM for signal <F0>.
Summary:
    inferred 1 Finite State Machine(s).
    inferred 3 ROM(s).
Unit <FSM1> synthesized.
...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state> on signal <state[1:2]> with gray encoding.
-----
State | Encoding
-----
001 | 00
010 | 01
100 | 11
-----
...
Macro Statistics
# FSMs          : 1
# ROMs          : 3
4x1-bit ROM     : 3
# Registers     : 2
Flip-Flops      : 2
Verilog-модель автомата Мура (рис. 11.24) з використанням одного блоку
always представлена лістингом 11.22. Треба враховувати, що в цьому випадку
результуюча схема має додаткові вихідні регістри, у представленому при-
кладі - це блок FDP (рис. 11.25) з результатами синтезу моделі. Таким чи-
ном, кількість тригерів у схемі збільшується, а вихідні значення надходять
на вихід із затримкою на один синхротакт стосовно зміни стану автомата.
Лістинг 11.22. Verilog-модель автомата з одним блоком.
module fsm_1xst
    #(parameter s1 = 2'b00,
      s2 = 2'b01,
      s3 = 2'b10,
      s4 = 2'b11)
    (input clk, reset, x1, output reg outp);
    reg [1:0] state;
always @(posedge clk or posedge reset)

```

```

begin
    if (reset) begin
        state = s1; outp = 1'b1;
    end
else
    case (state)
        s1: begin
            if (x1==1'b1) state = s2;
            else state = s3;
            outp = 1'b1;
        end
        s2: begin state = s4;
            outp = 1'b1;
        end
        s3: begin state = s4;
            outp = 1'b0;
        end
        s4: begin state = s1;
            outp = 1'b0;
        end
    endcase
end
endmodule

```

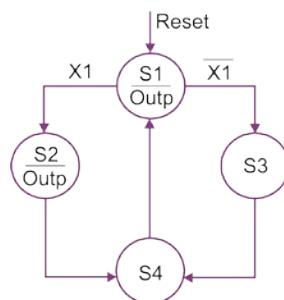


Рис. 11.24. Граф-переходів автомата Мура

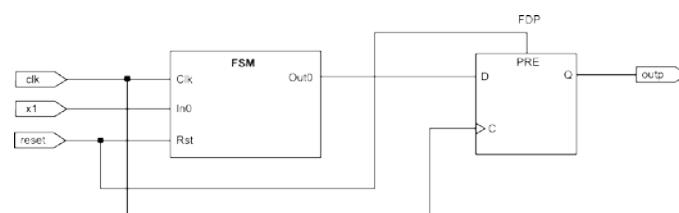


Рис. 11.25. Результат синтезу моделі автомата з одним блоком always

Лістинг 11.22, також як і приклад лістинга 11.20, використовує два блоки always. Однак відмінність полягає в тому, що один блок always реалізує функцію переходів і регістр станів, а другий – комбінаційну схему функції ви-

ходів. Також цей приклад ілюструє використання директиви компілятора ``define` для визначення станів автомата. Рис. 11.26 представляє результат синтезу моделі автомата (лістинг 11.23). У лістингу 11.24 наводиться фрагмент звіту про результат синтезу, з якого видно, що кодування станів автомата послідовним кодом збережено, проте коди для станів s3 і s4 змінилися.

Для опису комбінаційних функцій виходів може бути використаний оператор безперервного присвоєння `assign` з умовним оператором. Наприклад, другий блок `always` у лістингу 11.23 може бути замінений таким оператором: `assign outp = (state == 's1) || (state == 's2) ? 1'b1:1'b0;`

У цьому випадку змінна `outp` повинна мати тип `wire`.

Лістинг 11.23. Модель автомата Мура.

```
'define s1 2'b00
'define s2 2'b01
'define s3 2'b10
'define s4 2'b11
module FSM1
    (input clk, reset, x1, output reg outp);
    reg [1:0] state;
    initial
        state = 2'b00;
    always @(posedge clk, posedge reset)
        if (reset)
            state <= 's1;
        else
            case (state)
                's1: if (x1==1'b1)
                    state <= 's2;
                else
                    state <= 's3;
                's2: state <= 's4;
                's3: state <= 's4;
                's4: state <= 's1;
            endcase
    always @(state)
        case (state)
            's1: outp = 1'b1;
            's2: outp = 1'b1;
            's3: outp = 1'b0;
            's4: outp = 1'b0;
        endcase
endmodule
```

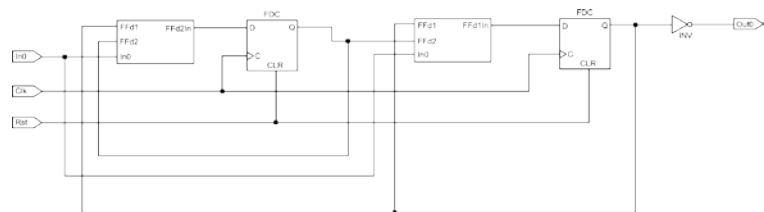


Рис. 11.26. Результат синтезу моделі автомата (лістинг 11.23)

Лістинг 11.24. Фрагмент звіту про синтез XST автомата.
Synthesizing Unit <FSM1>.

```
Found finite state machine <FSM_0> for signal <state>.
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| States                | 4 |
| Transitions           | 5 |
| Inputs                | 1 |
| Outputs               | 1 |
| Clock                 | clk (rising_edge) |
| Reset                 | reset (positive) |
| Reset type            | asynchronous |
| Reset State           | 00 |
| Power Up State        | 00 |
| Encoding               | automatic |
| Implementation        | LUT |
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
Summary:
      inferred 1 Finite State Machine(s).
Unit <FSM1> synthesized.
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state> on signal <state[1:2]> with gray encoding.
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
State | Encoding
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
00    | 00
01    | 01
10    | 11
11    | 10
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
Macro Statistics
# FSMs                : 1
# Registers            : 2
Flip-Flops            : 2
```

11.4. Моделі пам'яті

Далі представлені шаблони моделей пам'яті. Будь-яка пам'ять має набір параметрів, що визначає її тип: синхросигнал (clk), сигнал дозволу запису (we), адреса для операцій читання й запису (a, dpra), вхід даних (di.), вихід даних (do, spo, dpo) [1, 2].

Читання з пам'яті може бути асинхронним (лістинги 11.25 і 11.27) або синхронним (лістинги 11.26 і 11.28), тобто дані з'являються на виході відразу після надходження адреси або після приходу активного фронту синхросигналу. Зараз у додатках активно використовується пам'ять із двома портами для читання (лістинги 11.27 і 11.28), а також і для запису. Наприклад, для побудови буферної пам'яті між двома пристроями. Приклад RTL-синтезу моделі пам'яті представлений на рис 11.27. Апаратна реалізація пам'яті залежить від стилю її опису, а також від обраної для реалізації технології.

Лістинг 11.25. Модель пам'яті з асинхронним читанням.

```
module raminfr
    (input clk, we, input [4:0] a, input [3:0] di,
     output [3:0] do);
    reg [3:0] ram [31:0];
    always @(posedge clk)
        if (we) ram[a] <= di;
    assign do = ram[a];
endmodule
```

Лістинг 11.26. Модель пам'яті із синхронним читанням.

```
module raminfr
    (input clk, we, input [8:0] a, input [3:0] di,
     output [3:0] do);
    reg [3:0] ram [512:0];
    reg [8:0] read_a;
    always @(posedge clk) begin
        if (we) ram[a] <= di;
        read_a <= a;
    end
    assign do = ram[read_a];
endmodule
```

Лістинг 11.27. Модель двопортової пам'яті з асинхронним читанням.

```
module raminfr
    (input clk, we, input [4:0] a, dpra, input [3:0] di,
     output [3:0] spo, dpo);
    reg [3:0] ram [31:0];
    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end
    assign spo = ram[a];
    assign dpo = ram[dpra];
endmodule
```

Лістинг 11.28. Модель двопортової пам'яті із синхронним читанням.

```
module raminfr
    (input clk, we, input [4:0] a, dpra, input [3:0] di,
     output [3:0] spo, dpo);
    reg [3:0] ram [31:0];
```

```

    reg [4:0] read_a, read_dpra;
always @(posedge clk) begin
    if (we)
        ram[a] <= di;
        read_a <= a; read_dpra <= dpra;
end
assign spo = ram [read_a];
assign dpo = ram [read_dpra];
endmodule

```

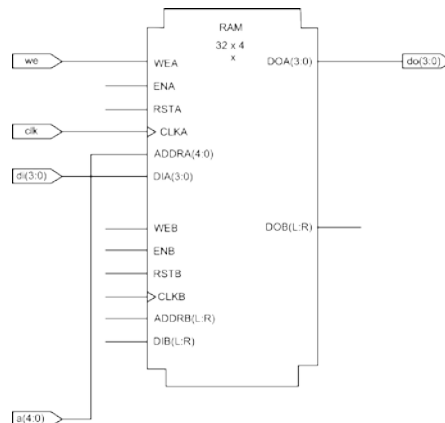


Рис. 11.27. Результат RTL-синтезу моделі пам'яті (лістинг 11.24)

11.5. Контрольні запитання й завдання

1. Який блок буде згенерований у результаті синтезу наступного коду?

```

module m
    (input C, D, CLR, output reg Q);
always @(negedge C, posedge CLR)
    if (CLR) Q = 1'b0;
    else Q = D;
endmodule

```

2. Який блок буде згенерований у результаті синтезу наступного коду?

```

module m
    (output OUT, input A, SEL);
    assign OUT = SEL ? A : 1'bz;
endmodule

```

3. Який блок буде згенерований у результаті синтезу наступного коду?

```

module m
    (output reg [3:0] COUNTER, input CLK, RESET);
always @(posedge CLK, posedge RESET)
    if (RESET) COUNTER = 'd0;
    else COUNTER = COUNTER + 'd1;
endmodule

```

4. Що буде згенеровано в результаті синтезу наступного коду?

```

module m

```

```

        (output reg Q, input D, CLK);
    always @(CLK, D)
        if (CLK) Q = D;
endmodule

```

5. Який блок буде згенерований в результаті синтезу наступного коду?

```

module m
    (output reg [7:0] Data_out,
     input [7:0] Data_in,
     input load, clock, reset);
always @(posedge reset, posedge clock)
    if (reset == 1'b1) Data_out <= 8'b0;
    else if (load == 1'b1) Data_out <= Data_in;
    else Data_out <= { Data_out[6:0], Data_in[7]};
endmodule

```

6. Записати синтезовану Verilog-модель 16-розрядного зсувного регістра з послідовними входом і виходом. SI - послідовний вхід даних, EN - вхід дозволу синхронізації й CK - синхронізація. Зсув виконується по передньому фронту. Послідовний вихід SO (рис.11.28).

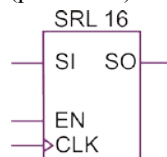


Рис. 11.28. 16-розрядний зсувний регістр

7. Записати синтезовану Verilog-модель двонаправленого 4-розрядного зсувного регістра, що має вхід скидання Clrb, асинхронний і активний за низьким рівнем, що скасовує дію інших керуючих входних сигналів. Всі зміни стану відбуваються по передньому фронту синхронізації. Якщо керуючі входи S1 = S0 = 1 (табл. 11.1), то виконується паралельне завантаження даних у регістр. Якщо S1 = 1 і S0 = 0 - зсув вправо й дані з SDR записуються в Q3. Якщо S1 = 0 і S0 = 1, то виконується зсув вліво й дані з SDL записуються в Q0. Якщо S1 = S0 = 0, - збереження стану (рис.11.29).

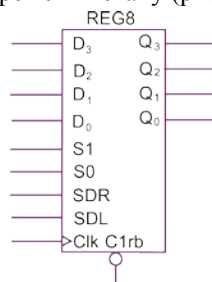


Рис. 11.29. Двонаправлений 4-розрядний зсувний регістр

Таблиця 11.1. Режими роботи зсувного регістра

Керуючі сигнали		Наступний стан				Опис
S1	S0	Q3+	Q2+	Q1+	Q0+	
0	0	Q3	Q2	Q1	Q0	Збереження стану
0	1	Q2	Q1	Q0	SDL	Зсув вліво
1	0	SDR	Q3	Q2	Q1	Зсув вправо
1	1	D3	D2	D1	D0	Паралельне завантаження

8. Записати синтезовану Verilog-модель 4-розрядного реверсивного десятичного лічильника з виходом Q і синхронізований переднім фронтом CLK, крім асинхронного скидання CLR. Коли CLR = 0, лічильник скидається в 0, незалежно від значень на інших входах. Якщо LOAD = 0, то в лічильник завантажуються дані від входу D. Якщо LOAD = ENT = ENP = UP = 1, то значення в лічильнику збільшується на 1. Якщо LOAD = ENT = ENP = 1 і UP = 0, то значення в лічильнику зменшується на 1. Якщо ENT = UP = 1, вихід переносу C0 = 1, коли лічильник перебуває в стані 9. Якщо ENT = 1 і UP = 0, вихід переносу C0 = 1, коли лічильник - у стані 0.

9. Записати синтезовану Verilog-модель N-бітового двонаправленого зсувного регістра, що має N-бітовий паралельний вхід і вихід даних, лівий послідовний вхід LSI, правий послідовний вхід RSI, вхід синхронізації й керуючі сигнали. Операція Load - паралельне завантаження даних у регістр, має більший пріоритет, ніж команди зсуву. Коли Rsh=1, виконується зсув регістра вправо, а значення з LSI заноситься в лівий розряд. Якщо Lsh =1, це означає зсув регістра вліво й сигнал RSI надходить у правий розряд.

10. Записати синтезовану Verilog-модель N-бітового двійкового реверсивного лічильника з керуючими входами U і D. При U = 1 - виконується рахунок на збільшення; D = 1 - рахунок на зменшення; U = D = 0 - збереження стану; U = D = 1 - заборонена комбінація. Значення N за замовчуванням дорівнює 8.

11. Розробити синтезовану Verilog-модель кінцевого автомата, що має вхід X, виходи D і B. На вхід X послідовно подається двійкове 4-бітве число N, починаючи з молодшого біта. D представляє 4-бітве число, рівне N-N-2. Значення N надходить на вихід, починаючи з молодшого розряду. Під час прийому четвертого біта вихід B=1, якщо N-N-2 - негативне число, B=0 - у протилежному випадку. Стан автомата скидається після надходження четвертого біта на вхід X. Зміни стану відбуваються по передньому фронту синхросигналу.

12. Послідовний автомат Мура із двома входами (X1 і X2) і одним виходом (Z) має таку таблицю переходів і виходів:

	00	01	10	11	Z
1	1	2	2	1	0
2	2	1	2	1	1

13. Написати Verilog-код автомата, представленого таблицею переходів:

Q ₁ Q ₂	Q ₁ ⁺ Q ₂ ⁺		Z	
	X = 0	X = 1	X = 0	X = 1
0 0	01	10	0	1
0 1	10	00	1	1
1 0	00	01	1	1

11.6. Література

1. Samir Palnitkar. Verilog HDL. A guide to digital design and synthesis.— Sun-Soft Press.— 1996.— 396p.
2. Bhasker J. Verilog HDL Synthesis. A Practical Primer. — Allentown: Star Galaxy Publishing, 1998. — 215 p.
3. Donald E. Thomas. Philip R. Moorby. The Verilog Hardware Description Language.— New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.— 2002.— 404 p.
4. James M. Lee, VERILOG QUICKSTART. A Practical Guide to Simulation and Synthesis in Verilog. — New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.— 2002.— 378 p.
5. IEEE Std 1364.1-2002. IEEE Standard for Verilog® Register Transfer Level Synthesis — IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 3 Park Avenue, New York, NY 10016-5997, USA— 2002.—109 p.
6. IEEE Std 1364-1995. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language— IEEE Standard Verilog Hardware Description Language.— IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 345 East 47th Street, New York, NY 10017-2394, USA — 1996.— 675 p.
7. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).— IEEE Standard Verilog Hardware Description Language.— IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 3 Park Avenue, New York, NY 10016-5997, USA.— 2001.— 791 p.

Частина IV. МОДЕЛЮВАННЯ Й ВЕРИФІКАЦІЯ ЧАСОВИХ ХАРАКТЕРИСТИК ЦИФРОВИХ КОМПОНЕНТІВ

Розглядаються питання, практично орієнтовані на аналіз тимчасових параметрів компонентів і цифрових систем на кристалах в цілому для виявлення змагань, гонок і ризиків збоїв, які деструктивно впливають на функціональні й часові характеристики проєктованих виробів. Описуються приклади моделювання часових характеристик у середовищі Xilinx ISE для ідентифікації змагань у комбінаційних і послідовних базових елементах, описаних мовою Verilog.

Мета - вивчення сучасних, практично орієнтованих методів і засобів моделювання для часової верифікації цифрових систем на кристалах і в пакетах, спрямованих на пошук і усунення часових колізій, що призводять до втрати швидкодії й до порушення функціональних режимів проєктованого обчислювального пристрою.

Завдання: 1. Аналіз часових характеристик і типів затримок цифрової схеми: затримки вентиля й тригера; часові параметри послідовних систем; зворотний зв'язок у тригері; час установки й зберігання для зовнішніх входів. 2. Імплементация проєкту в кристал: етапи імплементации; використання пакета Xilinx ISE; звіти, що формуються в процесі імплементации; Translate Report. 3. Моделювання параметрів часу в середовищі Verilog. Моделі затримок: розподілена (distributed delay) і зосереджена (lumped delay). Затримка сигналу між виводами (pin-to-pin delay) і логічного шляху (path delay). Моделювання затримки шляху в Verilog. Паралельне й повне з'єднання. Параметри часу спрерагам, чутливі до фронту сигналу, і умовні шляхи затримок – rise, fall і turn-off. Обробка x- переходів. Перевірка часових параметрів для установки й зберігання даних. Вплив ширини імпульсу. 4. Послідовний суматор: RTL-модель пристрою; синтез керуючого автомата; синтез операційного пристрою в цілому; оцінка часових параметрів.

Джерела: промисловий синтез і часова верифікація цифрових систем на кристалах і в пакетах за допомогою мови Verilog [1, 2, 4, 8]; синтаксис і семантика мови опису апаратури Verilog [3]; стандарт мови опису апаратури Verilog [5-7].

12. ОПИС ЧАСОВИХ ПАРАМЕТРІВ

12.1. Терміни

1. Propagation Delay (затримка поширення) - період часу між подією на вході й появою відповідної реакції на виході. Подією є будь-яка зміна сигналу на лінії.

2. Setup Time (час установки) – період часу до появи активного фронту синхросигналу, протягом якого описаний вхідний сигнал повинен залишатися незмінним (рис. 12.1).

3. Hold Time (час зберігання) – період часу після надходження активного фронту синхроімпульсу, протягом якого вхідний сигнал не може змінювати своє значення (рис. 12.1).

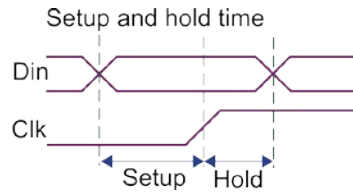


Рис. 12.1. Час установки й час зберігання

4. Release Time (час звільнення) – мінімальний проміжок часу між перемиканням асинхронного (set, reset) вхідного сигналу в активний стан і появою синхрофронту (подібно до часу установки для синхронних входів) (рис. 12.2).

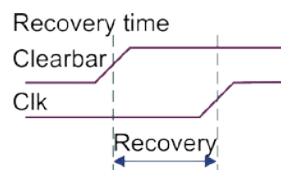


Рис. 12.2. Час звільнення

5. Removal Time (recovery time, час відновлення) – мінімальний проміжок часу між появою синхрофронту й установкою асинхронного (set, reset) вхідного сигналу в активне значення (подібно до часу зберігання для синхронних сигналів) (рис. 12.3).

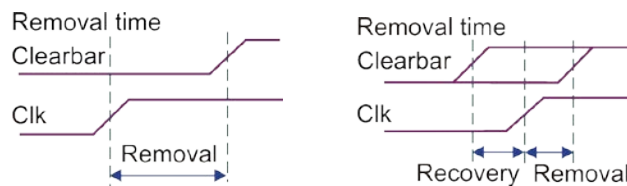


Рис. 12.3. Час відновлення

6. Period – час між активними фронтами синхросигналу (рис. 12.4).



Рис. 12.4. Period

7. Width – мінімальна ширина імпульсу (рис. 12.5).

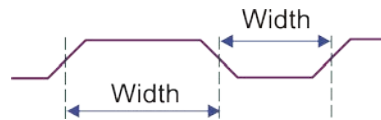


Рис. 12.5. Width

8. Skew – представляє зсув між парою сигналів (рис. 12.6).



Рис. 12.6. Skew

9. Незмінюваність (NOCHANGE) – мінімальний параметр для перевірки збереження сигналом незмінного значення (рис. 12.7). Ця перевірка споріднена з перевіркою ширини керуючого імпульсу. Період setup встановлюється до керуючого імпульсу, а hold - після. Сигнал, що перевіряється відносно керуючого сигналу, повинен залишатися стабільним протягом setup period, ширини імпульсу й hold period. Зазвичай використовується в часових моделях пам'яті, у яких адресна лінія не повинна змінюватися під час появи сигналу запису, плюс додатковий час до й після цього сигналу.

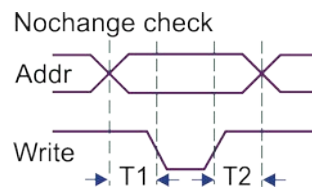


Рис. 12.7. Незмінюваність

10. Interconnect path delay - представляє зовнішню затримку лінії до порту, або затримку міжз'єднань, що спрощена до простої затримки лінії розглянутого компонента.

12.2. Аналіз часових характеристик цифрової схеми

12.2.1. Затримка вентиля.

Для передачі сигналу через вентиль після зміни входу потрібен кінцевий час, який називається затримкою розповсюдження. Чим менший за розміром транзистор, тим коротший час його перемикання.

Тому компанії, що виробляють напівпровідникові пристрої, намагаються мінімізувати розміри транзисторів, щоб підвищити швидкість роботи пристрою.

Затримки поширення одновходового елемента на прикладі інвертора (рис. 12.8):

- 1) t_{plh} - час між подією на вході елемента й зміною його виходу з 0 в 1. Величина вимірюється між 50% точками вхідного й вихідного сигналів. Фрагмент індексу lh позначає зміну вихідного сигналу;
- 2) t_{phl} - час між подією на вході елемента й зміною його виходу з 1 в 0. Величина вимірюється між 50% точками вхідного й вихідного сигналів. Фрагмент індексу hl позначає зміну вихідного сигналу.

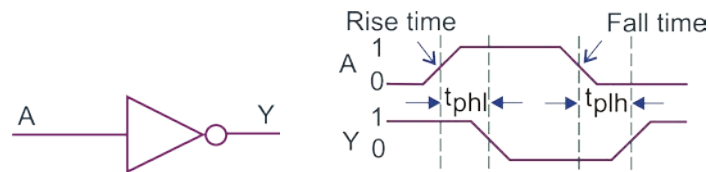


Рис. 12.8. Затримки інвертора

Для вентилів, що мають більше одного входу (рис. 12.9), записуються затримки від кожного входу: $A2Y\_tplh$, $A2Y\_tphl$, $B2Y\_tplh$, $B2Y\_tphl$. Ці параметри можуть мати різні значення. Для простоти може задаватися одне з них Y_tpd . У технічній документації часто вказуються типова й максимальна затримки.

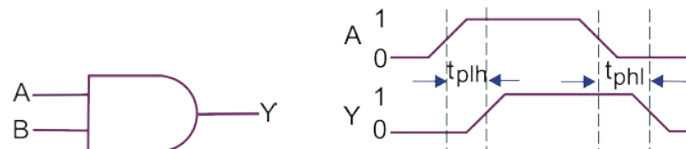


Рис. 12.9. Затримки вентиля

12.2.2. Затримки тригера.

Затримки поширення послідовних компонентів представлені на прикладі D-тригера (рис. 12.10) з асинхронними входами скидання (R) і установки (S):

- 1) $TC2Q$ – час, необхідний для зміни виходу Q у відповідь на зміну C. Вихідне значення Q залежить від входу D;
- 2) $TS2Q$, $TR2Q$ - час, необхідний для зміни виходу Q після зміни асинхронних входів S і R.

Не існує затримки між синхронним входом D і виходом Q, оскільки при зміні значення на вході D значення виходу зміниться тільки після надходження фронту синхросигналу - через період часу $TC2Q$.

Для синхронних входів (D) визначають час установки (t_{su}) і зберігання (t_{hd}) стосовно їхнього входу синхронізації (рис. 12.11). Setup Time - період часу до активного фронту синхросигналу, протягом якого синхронний вхід D не повинен змінювати своє значення. Hold Time - період часу після активного фронту синхросигналу, протягом якого синхронний вхід D не повинен змінювати своє значення.

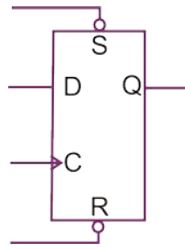


Рис. 12.10. Затримки послідовного елемента

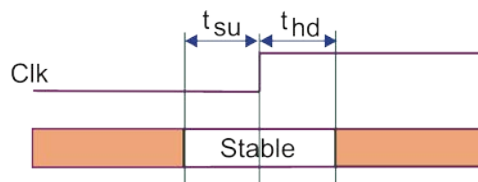


Рис. 12.11. Час установки й зберігання

Якщо сигнал на вході D змінюється під час установки (Setup) або зберігання (Hold), то це може призвести до неправильної роботи тригера FF.

12.2.3. Часові параметри послідовних систем.

Модель послідовного пристрою може бути представлена схемою, наведеною на рис. 12.12.

Максимальна частота пристрою визначається за формулою:

$$Freq_{max} = 1/(T_{d_{max}}),$$

де $T_{d_{max}}$ - найдовший шлях у послідовній схемі, визначається на основі параметрів, наведених нижче, з яких вибирається максимальний:

1) Clock-to-Output: затримка між синхровходом і виходом

$$T_{c2q} + T_{comb\_Q2O_{max}},$$

де $T_{comb\_Q2O_{max}}$ - найдовший шлях від виходу тригера Q до будь-якого виходу схеми;

2) Register-to-Register: затримка між регістрами

$$T_{c2q} + T_{comb\_Q2Dmax} + T_{su},$$

де T_{comb_Q2D} - найдовший шлях від виходу тригера Qdff до будь-якого входу тригера Ddff;

3) Pin-to-Pin: комбінаційний шлях у схемі

$$T_{comb\_I2Omax}.$$

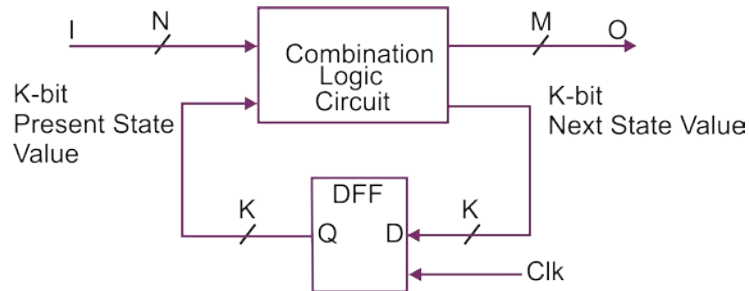


Рис. 12.12. Структура послідовного пристрою

У більшості схем другий з перелічених параметрів має максимальне значення й визначає робочу частоту пристрою.

Найчастіше всі входи й виходи схеми є регістровими (рис. 12.13). Таким чином, максимальна робоча частота схеми практично завжди залежить від затримки Register-to-Register:

$$T_{dmax} = T_{c2q} + T_{pdmax} + T_{setup}.$$



Рис. 12.13. Пристрій з регістровими входами-виходами

Для виконання умови стабільності сигналу після фронту (Hold time) мінімальна затримка шляху між регістрами повинна задовольняти умові (рис. 12.14.):

$$T_{c2q} + T_{pdmin} \geq T_{hold},$$

яка, як правило, легко виконується для послідовних систем.



Рис. 12.14. Параметр Hold

12.2.4. Зворотний зв'язок і тригер.

Для тригера зі зворотним зв'язком (рис. 12.15) розглядаються такі параметри:

- 1) частота перемикання $F_{\text{req max}} = 1 / (C2Q + T_{\text{setup}})$;
- 2) затримка, достатня для виконання умов стабільності сигналу
 $T_{c2q} - [T_{pd(\text{min})} \rightarrow 0] \geq T_{\text{hold}}$.

Затримка $T_{pd \text{ min}}$ лінії вважається незначною і її величина не враховується, тому $T_{c2q} > T_{\text{hold}}$.

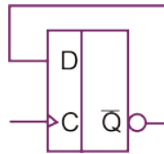


Рис. 12.15. Тригер зі зворотним зв'язком

12.2.5. Час установки й зберігання для зовнішніх входів.

На шляху між зовнішніми входами й синхронними входами тригера знаходяться буфери, крім цього може бути присутня деяка комбінаційна схема. Зазначені елементи мають власну затримку розповсюдження й впливають на величину часу розповсюдження й зберігання щодо зовнішніх входів пристрою.

Найгірша ситуація для часу установки T_{su} , коли затримка комбінаційного шляху ($T_{pd \text{ DIN}}$) перевищує затримку лінії синхронізації ($T_{pd \text{ CLK}}$) (рис. 12.16).

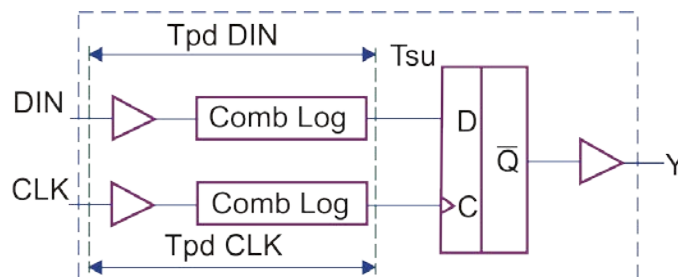


Рис. 12.16. Обчислення часу зберігання для зовнішніх входів

Синхросигнал досягає входу тригера раніше, ніж дані - вхід D. Таким чином, на зовнішній вхід даних DIN інформація повинна надходити раніше, ніж активний фронт на вхід CLK:

$$T_{su \text{ DIN}} = T_{su} + T_{pd \text{ DIN max}} - T_{pd \text{ CLK min}}.$$

Найгірша ситуація виникає, коли затримка шляху синхронізації ($T_{pd \text{ CLK}}$) перевищує затримку комбінаційного шляху даних ($T_{pd \text{ DIN}}$) (рис. 12.17).

У цьому випадку синхросигнал досягає входу тригера із запізненням, що призводить до необхідності довше зберігати сигнал незмінним на вході даних:

$$Thd_{DIN} = Thd + Tpd_{CLK\ max} - Tpd_{DIN\ min}.$$

Якщо $Tpd_{DIN\ min}$ перевищує величину $Thd + Tpd_{CLK\ max}$, то значення Thd_{DIN} буде негативним. Це відповідає ситуації, зображеної на рис. 12.18, коли значення входу може змінюватися до надходження синхросигналу. Негативні значення замінюються 0.

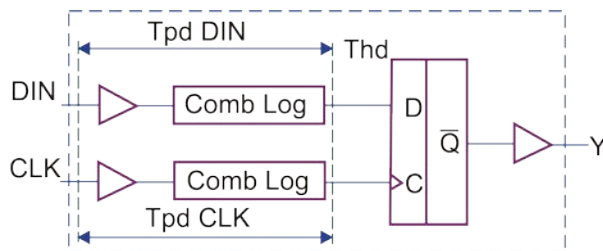


Рис. 12.17. Обчислення часу зберігання для зовнішніх входів

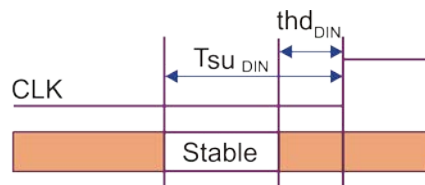


Рис. 12.18. Негативне значення часу зберігання

12.3. Приклад обчислення часових параметрів

Пропонується аналіз часових параметрів на прикладі схеми рис. 12.19, яка містить два тригери (U1, U2), три логічних елементи (U3 – U5) і три буфери (U6 – U8). Часові параметри всіх компонентів наведені на рис. 12.19.

1. Register-to-Register max:

$$Tc2q2 + Tpd3 + Tsu1 = 5 + 8 + 3 = 16\ ns.$$

2. Asu:

$$\begin{aligned} Tsu + Tpd2D\ max - TpdCLK\ min &= \\ &= Tsu + (Tpd3 + Tpd7) - Tpd8 = 3 + (8 + 1) - 2 = 10\ ns. \end{aligned}$$

3. Ahd:

$$\begin{aligned} Thd + TpdCLK\ max - Tpd2D\ min &= \\ &= Thd + Tpd8 - (Tpd4 + Tpd7) = 4 + 2 - (7 + 1) - 2 = -2\ ns. \end{aligned}$$

4. Clock-to-Out max:

$$Tpd8 + Tc2q2 + Tpd5 + Tpd6 = 2 + 5 + 9 + 6 = 22\ ns.$$

5. Pin-to-Pin max (від А до Y):

$T_{pd7} + T_{pd5} + T_{pd6} = 1 + 9 + 6 = 16 \text{ ns.}$

6. Freq max:

$1 / \text{Max(Reg-to-Reg, Clock-to-Out, Pin-to-Pin)} =$
 $= 1 / \text{Max}(16, 22, 16) = 45,5 \text{ MHz.}$

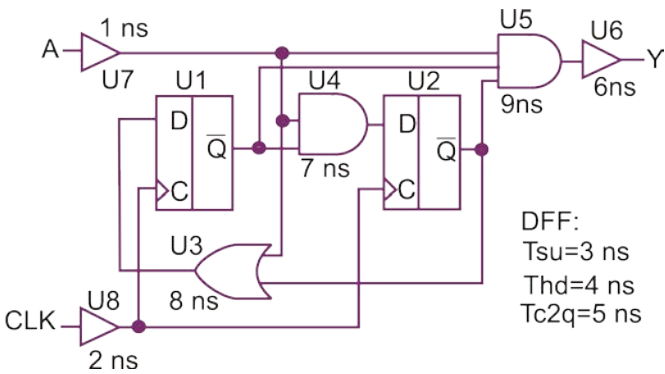


Рис. 12.19. Послідовна схема

Отримані характеристики пристрою представлені в табл. 12.1.

Таблиця 12.1. Часові характеристики пристрою (рис. 12.19)

Parameter	Description	Min	Max	Units
Tclk	Clock period(період)	22		ns
Fclk	Clock frequency		45,5	MHz
Atsu	A setup time	10		ns
Athd	A hold time	-2		ns
A2Y	A to Y Tpd		16	ns
Ck2Y	Clock to Y Tpd		22	ns

Поліпшити часові характеристики пристрою можна, перетворивши вхід А і вихід Y у регістри. Для цього в схему (див. рис. 12.19) додаються два тригери U8 і U9 (мал. 12.20). Тепер схема не містить комбінаційного шляху.

Приклад обчислення часових параметрів для регістрових входів і виходів наведений нижче.

1. Register-to-Register max:

$T_{c2q2} + T_{pd5} + T_{su9} = 5 + 9 + 3 = 17 \text{ ns.}$

2. Asu:

$T_{su} + T_{pd2D} \text{ max} - T_{pdCLK} \text{ min} =$
 $= T_{su} + (T_{pd7}) - T_{pd8} = 3 + (1) - 2 = 2 \text{ ns.}$

3. Ahd:

$T_{hd} + T_{pdCLK} \text{ max} - T_{pd2D} \text{ min}$
 $= T_{hd} + T_{pd8} - (T_{pd7}) = 4 + 2 - (1) = 5 \text{ ns.}$

4. Clock-to-Out max:

$$T_{pd8} + T_{c2q10} + T_{pd6} = 2 + 5 + 6 = 13 \text{ ns.}$$

5. Freq max:

$$1 / \text{Max}(\text{Reg-to-Reg, Clock-to-Out, Pin-to-Pin}) = 1/17 = 58.8 \text{ MHz.}$$

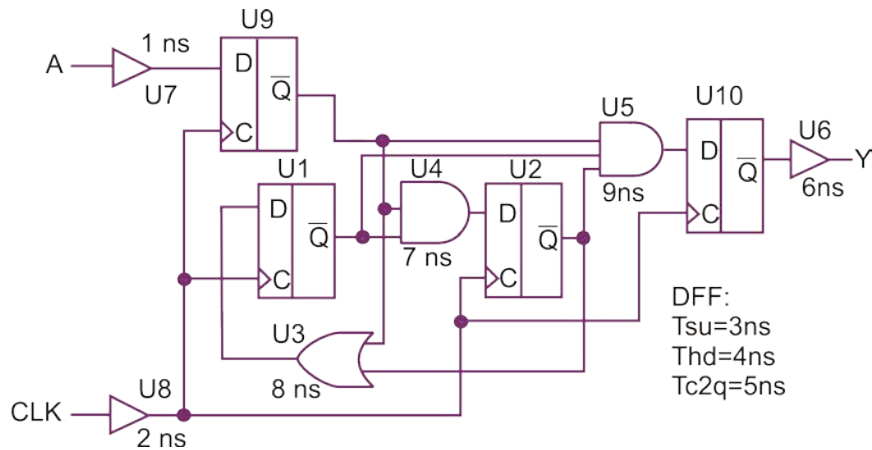


Рис. 12.20. Приклад пристрою з регістровими входами й виходами

Отримані характеристики пристрою представлені в табл. 12.2.

Таблиця 12.2. Часові характеристики пристрою (рис. 12.20)

Parameter	Description	Min	Max	Units
T_{CLK}	Clock period(період)	17		ns
Freq	Clock frequency		58.8	MHz
Asu	A setup time	2		ns
Ahd	A hold time	5		ns
A2Y	A to Y Tpd		-	ns
CLK2Y	Clock to Y Tpd		13	ns

12.4. Імплементація проекту в силіконовий кристал

У підрозділі описуються етапи, що виконуються в процесі імплементації проектів для FPGA і CPLD. Представлено тимчасові моделі й способи оцінки часових параметрів проєктованих пристроїв.

12.4.1. Етапи імплементації. Використання пакета Xilinx ISE.

Після синтезу формується схема (netlist), що складається з компонентів цільового пристрою (технології). Для того щоб помістити проєкт у мікросхему, необхідно виконати операцію імплементації (Implementation).

Для мікросхем програмованої логіки імплементація включає такі етапи (рис. 12.21):

- 1) FPGA: Translate, Map, Place and Route, Configure;
- 2) CPLD: Translate; Fit; Configure.

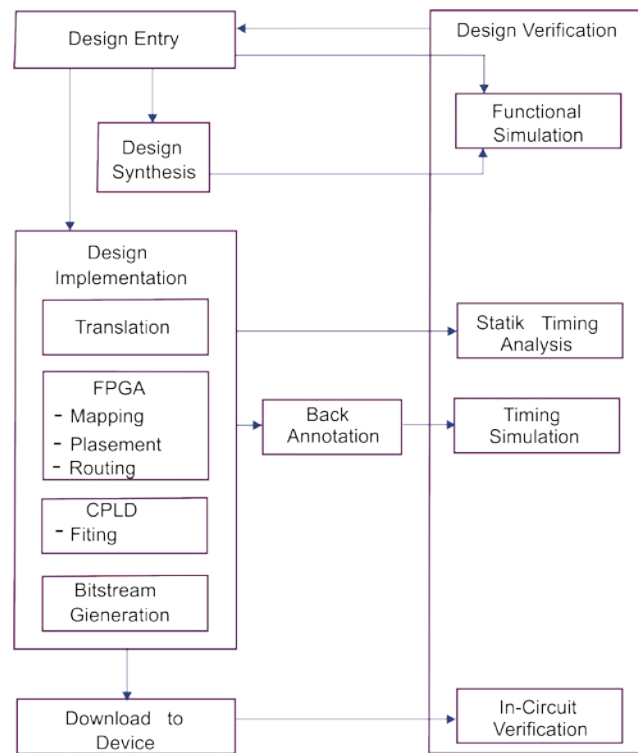


Рис. 12.21. Етапи проектування. Місце імплементації в процесі проектування

Translate - інтерпретація проекту й перевірка його на правильність опису (design rule check). Складається з набору різних програм, які використовуються для імпортування netlist проекту й підготовки його до розміщення. Набір програм може змінюватися від виробника до виробника, але, як правило, включає оптимізацію, трансляцію у фізичні елементи, перевірку правильності моделі (наприклад, наявність необхідної кількості синхробуферів, вхідних/вихідних портів). Етап трансляції звичайно закінчується створенням звіту.

Map (English) - обчислення й розподіл ресурсів цільової мікросхеми для розроблюваного проекту.

Place and Route - розміщення CLB у логічні позиції мікросхеми й створення засобів маршрутизації між ними. Більшість розроблювачів пропонують автоматичні інструменти Place and Route, тому у користувача немає необхідності докладно вникати в деталі цього процесу. Деякі розроблювачі мають інструменти, що дозволяють виконувати операції Place and/or Route вручну для найбільш критичних шляхів, зберігаючи кращі властивості автоматичних інструментів. Одним із прикладів таких інструментів є програма Floorplanner з пакета Xilinx ISE.

Процес Place and Route вимагає значних тимчасових витрат, оскільки розміщення великого проекту по компонентах мікросхеми й виконання міжз'єднань, при забезпеченні їхньої правильності й відповідності вимогам проектувальника, є досить складним завданням.

Fit (Fitting) - виконується при проектуванні на CPLD і означає припасування ("Fit") проекту до цільового пристрою, що має фіксовану структуру, з якого вибираються вентиля й з'єднання для формування проекту. Зазвичай цей процес не займає багато часу. Потенційним джерелом проблем у цьому випадку є попередній розподіл вхідних/вихідних контактів, який зазвичай має назву Pin Locking. Найчастіше його формують на попередніх ітераціях проектування, розподіл контактів залежить від топології друкованої плати.

Configure - створення двійкового файлу конфігурації FPGA або JED-файлу для програмування CPLD.

Downloading or Programming - завершальний етап імплементації. Термін Download звичайно ставиться до енергозалежних пристроїв, таким як SRAM FPGAs. У цьому випадку виконується завантаження конфігурації мікросхеми в пам'ять пристрою. Переданий потік даних Bitstream містить всю інформацію про використовувані ресурси і є індивідуальним для кожного проекту. Оскільки SRAM пристрої втрачають свою інформацію після вимикання живлення, конфігурація повинна зберігатися в іншому місці. Зазвичай для цього використовується послідовне ПЗП. Program - застосовується для програмування енергонезалежних програмувальних логічних пристроїв, включаючи послідовні ПЗП. Програмування виконує ті ж функції, що й завантаження, за винятком того, що інформація про конфігурацію зберігається після вимикання живлення. Для антиф'юзних компонентів програмування виконується тільки один раз. (Звідси термін - одноразово програмувальний пристрій - One-Time Programmable, OTP).

Завантаження конфігурації в FPGA і програмування CPLD виконується в системі (In-System) через інтерфейс JTAG або Boundary Scan, також відомий як стандарт IEEE/ANSI 1149.1_1190, що являє собою набір правил проектування для спрощення процесу тестування, програмування й налагодження мікросхеми, плати або системи. Перевага програмування в системі полягає в тому, що змінити конфігурацію пристрою можна, не виймаючи мікросхему із плати.

12.5.2. Звіти, що формуються в процесі імплементації. Translate Report.

На кожному етапі імплементації генерується звіт:

1. Translate Report - показує помилки в проекті або UCF (User Constraints File).
2. Map Report - показує ресурси, використовувані в проекті. Докладний звіт map може бути заданий у властивостях команди. Він описує trimmed (усічену) і merged (сполучену) логіку, а також показує, де і яка частина проекту розміщається в реальному пристрої.
3. Post-Map Static Timing Report - показує тільки затримки елементів, не враховуючи затримки ліній. Якщо затримки логіки не задовольняють тимчасо-

вим параметрам, то додані до неї затримки зв'язків тільки погіршать ситуацію.

4. Place and Route Report - відображає покроково процес виконання етапу Place&Route. Інструменти, що виконують Place&Route, повинні отримати інформацію про часові вимоги. Звіт може містити список обмежень (констант) і опис заходів відповідності сформованої схеми заданим характеристикам.

5. Asynchronous Delay Report - містить затримки найбільш довгого шляху в проєкті з урахуванням затримок елементів і шляхів.

6. Pad Report - містить список розподілів зовнішніх контактів з інформацією про drive strength і стандарт сигналу.

7. Post Place and Route Static Timing Report - враховує затримки між'єднань проєкту.

12.5.3. Тестування пристрою на різних стадіях проєктування.

Functional Simulation (див. рис. 12.21) - використовується для перевірки поводження пристрою. Якщо на цьому етапі виявляються помилки, проєктувальник виправляє їх у моделі й повторює процес моделювання. На цей етап витрачається 50% часу розробки пристрою.

Потім виконується синтез схеми пристрою й постсинтезне моделювання його поводження, при якому перевіряється правильність синтезу схеми з RTL. Виробляється аналіз часових параметрів, що враховує затримки елементів.

На етапі Place&Route програма Timing-Driven Place&Route (TDPR) дає можливість користувачу вказувати часові параметри, які будуть застосовуватися під час розміщення проєкту.

Static Timing Analyser - частина розроблювального користувачем програмного забезпечення, дозволяє одержати інформацію про логічні шляхи поширення сигналів у проєкті.

Крім того, користувач може отримати докладну інформацію про розміщення проєкту в мікросхемі, що може бути передана назад у програму моделювання разом з точною інформацією про тимчасові параметри системи. Цей процес називається Back-Annotation і його перевага полягає в точному описі часових параметрів одночасно з поводженням проєкту. В обох випадках часові параметри будуть відображати затримки і логічних блоків, і між'єднань.

12.5. Контрольні запитання

1. Якого з перелічених часових параметрів не існує для елемента, представленого на рис. 12.22?

а) TC2Q - затримка зміни виходу Q після надходження фронту синхросигналу C;

б) TS2Q - затримка зміни виходу Q після зміни асинхронного входу установки S;

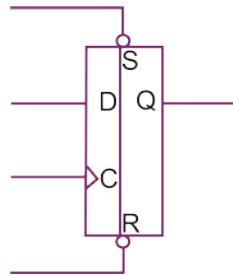


Рис. 12.22. Приклад пристрою

в) TR2Q - затримка зміни виходу Q після зміни асинхронного входу скидання R;

г) TD2Q - затримка зміни виходу Q після зміни синхронного входу D.

2. Чому дорівнює максимальна затримка шляху між регістрами (Register to Register delay) для схеми на рис. 12.23?

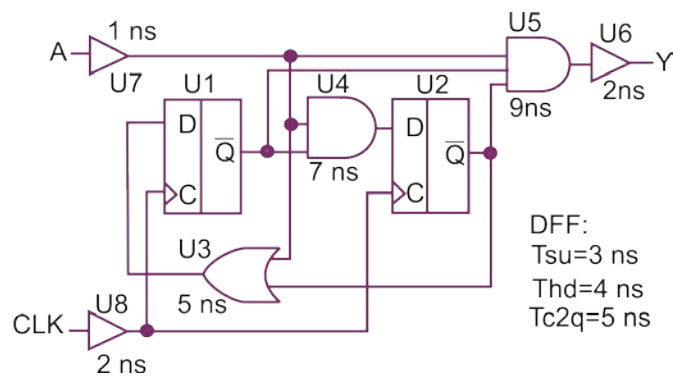


Рис. 12.23. Приклад схемы прибораю

3. Чому дорівнює затримка від синхросигналу до виходу (Clock to Output delay) для схеми на рис. 12.24?

4. Чому дорівнює затримка комбінаційного шляху (Pin to Pin delay) для схеми на рис. 12.25?

5. Для цифрового пристрою були отримані такі часові параметри:

a) Pin to Pin delay = 12 ns (затримка комбінаційного шляху);

б) Clock to Output delay = 16 ns (затримка від синхросигналу до виходу);

в) Register to Register delay = 15 ns (затримка між регістрами).

Чому дорівнює мінімальний припустимий для цифрового пристрою період синхросигналу?

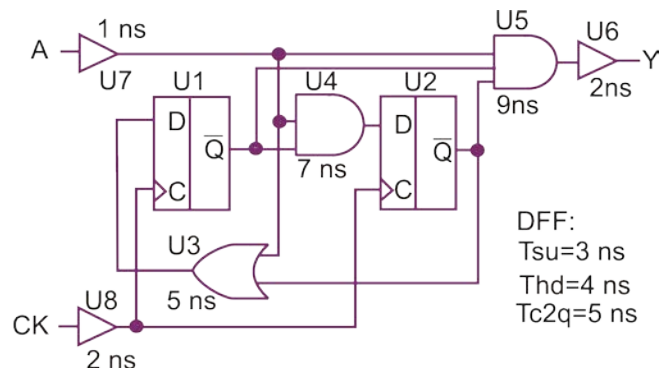


Рис. 12.24. Приклад схеми пристрою

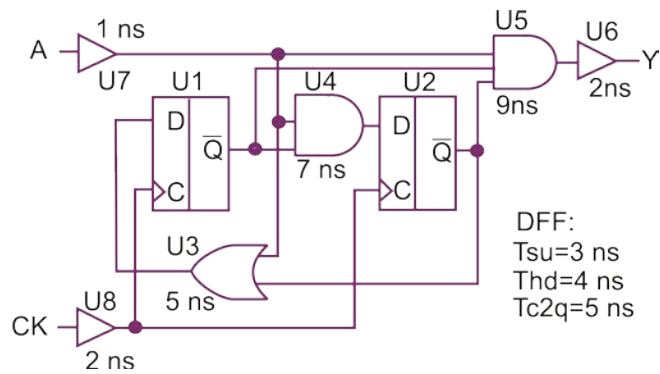


Рис. 12.25. Приклад схеми пристрою

13. МОДЕЛЮВАННЯ ЧАСОВИХ ПАРАМЕТРІВ ПРОЕКТУ В СЕРЕДОВИЩІ VERILOG

13.1. Моделі затримок у середовищі Verilog

13.1.1. Розподілена затримка (distributed delay).

Описується на основі інформації про затримки елементів. Кожний елемент схеми має власну затримку (рис. 13.1).

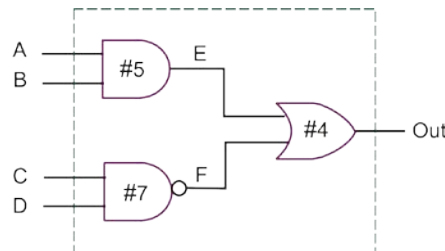


Рис. 13.1. Розподілена затримка

Розподілена модель може бути побудована шляхом присвоєння затримок окремим вентилям (лістинг 13.1, а) або в операторах assign (лістинг 13.1, б).

Лістинг 13.1. Розподілена затримка.

а) // Розподілена затримка в моделі вентильного рівня
module M

```
(output out, input a, b, c, d);
  wire e, f;
```

```
//Затримка розподіляється по кожному вентилю
```

```
and #5 a1 (e, a, b);
```

```
nand #7 a2 (f, c, d);
```

```
or #4 a3 (out, e, f);
```

```
endmodule
```

б) // Розподілена затримка в моделі dataflow-рівня

```
module M
```

```
(output out, input a, b, c, d);
```

```
  wire e, f;
```

```
//Затримка описується в операторі assign
```

```
assign #5 e = a & b;
```

```
assign #7 f = ~(c & d);
```

```
assign #4 out = e | f;
```

```
endmodule
```

13.1.2. Зосереджена затримка (lumped delay).

Зосереджена (схемна) затримка складається на основі інформації про затримки елементів. Вона представлена максимальною затримкою серед всіх логічних шляхів, що ведуть до виходу (рис. 13.2, лістинг 13.2).

Лістинг 13.2. Зосереджена затримка.

```
// Розподілена затримка в моделі вентиляного рівня
module M
    (output out, input a, b, c, d);
    wire e, f;
    and a1 (e, a, b);
    nand a2 (f, c, d);
    or #11 a3 (out, e, f); // Затримку має тільки вихідний вентиль
endmodule
```

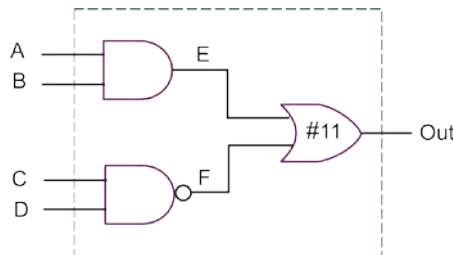
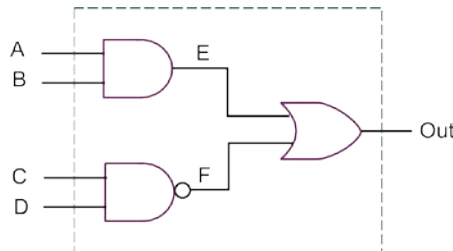


Рис. 13.2. Зосереджена, або схемна затримка

13.1.3. Затримка сигналу між виводами (pin-to-pin delay) або затримка логічного шляху (path delay).

Затримка присвоюється кожному логічному шляху між входами й виходами схемної структури (рис. 13.3.).



$T_{a\_out} = 9, T_{b\_out} = 9, T_{c\_out} = 11, T_{d\_out} = 11$

Рис. 13.3. Затримка сигналу між виводами

Затримку сигналу між виводами часто отримують із документації.

Хоча така затримка є дуже детальною, для більших схем вона є кращою, ніж розподілена затримка, оскільки проєктувальникові досить знати тільки вхідні й вихідні контакти схеми без її внутрішньої структури. Остання може бути представлена у вигляді опису вентиляного, dataflow, поведінкового рівнів або бути комбінацією різних рівнів. При цьому величина затримки шляху залишається незмінною. Затримка між вхід-вихідними виводами також називається затримкою шляху (path delay).

13.2. Моделювання затримки шляху в середовищі Verilog

Затримка шляху в Verilog описується за допомогою конструкції `specify` і `endspecify`. Оператори усередині цього блоку є описовими й виконують такі завдання:

- 1) описувати затримку між виводами;
- 2) виконувати перевірку часових параметрів у схемі;
- 3) визначати константи `specparam`.

Для прикладу (рис. 13.3) опис моделі із затримкою між виводами представлено лістингом 13.3. Описовий блок є незалежним і не може з'являтися під керуванням операторів `initial` або `always`.

Лістинг 13.3. Затримка між виводами.

```
// Розподілена затримка в моделі вентиляного рівня
module M
    (output out, input a, b, c, d);
    wire e, f;
//Опис блоку із затримками шляху
specify
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
endspecify
// копії вентилів
and a1 (e, a, b);
nand a2 (f, c, d);
or a3 (out, e, f);
endmodule
```

У прикладі сигнали `a`, `b`, `c` і `d` являють собою вихідне поле (`source field`), `out` - поле призначення (`destination field`).

13.2.1. Паралельне і повне з'єднання.

Паралельне з'єднання позначається символом `=>` і має такий синтаксис:

```
(<source_field> => <destination_field>) = <delay_value>;
```

При паралельному з'єднанні кожний біт з поля джерела з'єднується з кожним бітом поля приймача (рис. 13.4). Якщо джерело й приймач є векторами, вони повинні мати однакове число біт.

Існує кілька видів з'єднань:

- 1) Однобітове з'єднання, коли джерело й приймач являють собою однорядні змінні:

```
(a => out) = 9;
```

- 2) Векторне з'єднання, якщо `a` і `out` є векторами розмірністю `[3:0]`.

```
(a => out) = 9;
```

що відповідає результату:

```
(a[0] => out[0]) = 9;
```

```
(a[1] => out[1]) = 9;
```

```
(a[2] => out[2]) = 9;
```

```
(a[3] => out[3]) = 9;
```

Повне з'єднання описується символом `*>` і використовує синтаксис:

```
(<source_field> *> <destination_field>) = <delay_value>;
```



Рис. 13.4. Паралельне з'єднання

Кожний біт джерела з'єднується з кожним бітом приймача (рис. 13.5). У цьому випадку, якщо приймач і джерело являють собою вектори, то вони не обов'язково повинні бути однакової довжини.

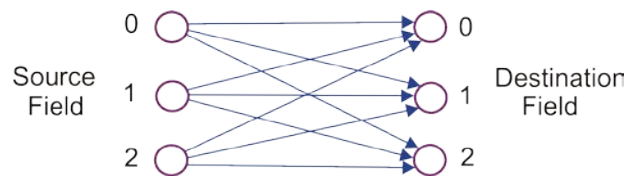


Рис. 13.5. Повне з'єднання

Приклад застосування повного з'єднання представлений лістингом 13.4.

Лістинг 13.4. Використання моделі повного з'єднання.

```
module M
    (output out, input a, b, c, d);
    wire e, f;
    // full connection
    specify
        (a, b *> out) = 9;
        (c, d *> out) = 11;
    endspecify
    and a1 (e, a, b);
    and a2 (f, c, d);
    and a3 (out, e, f);
endmodule
```

Ще один приклад ілюструє переваги використання повного з'єднання. Нехай `a[31:0]` - це 32-розрядний вхідний вектор, а `out[15:0]` - 16-розрядний вихідний вектор. Затримка між зміною кожного вхідного біта й появою реакції на виходах дорівнює 9 одиницям часу.

Для опису затримки можна використовувати код з оператором повного з'єднання:

```
specify
    (a *> out) = 9;
endspecify
```

Для порівняння необхідно відзначити, що при опису даної ситуації треба було б $32 \cdot 16 = 352$ оператори паралельного з'єднання.

13.2.2. Параметри часу спсрагам.

Конструкція `спсрагам` застосовується усередині блоку `specify`. Замість використання фіксованих значень для опису затримок `pin-to-pin` набагато зручніше визначити параметр, застосовуючи конструкцію `спсрагам`, і використовувати його (лістинг 13.5). Цей параметр часто служить для збереження значень для інструментів, які не є симуляторами, а вирішують завдання, наприклад, обчислення затримок, логічний синтез, розміщення блоків. Блок, що містить параметри `спсрагам`, представлений лістингом 13.5.

Лістинг 13.5. Використання `спсрагам`.

```
specify
    спсparam d_to_q = 9;
    спсparam clk_to_q = 11;
    (d => q) = d_to_q;
    (clk => q) = clk_to_q;
endspecify
```

Параметри `спсрагам` використовуються тільки в межах блоку, у якому вони описані. Рекомендується, щоб всі затримки були задані із застосуванням параметрів `спсрагам`.

13.2.3. Чутливі до фронту сигналу шляху.

Якщо шлях модуля описується за допомогою фронту сигналу як джерело, його називають чутливим до фронту. Такі шляхи використовуються для моделювання затримок від входу до виходу, які виникають при певному фронті сигналу.

Синтаксис:

```
// Паралельне з'єднання
([edge_identifier] source_field => destination_field [polarity] : data_source_expression))
// Повне з'єднання
([edge_identifier] source_field *> destination_field [polarity] : data_source_expression))
```

Параметр `edge_identifier` може бути описаний з використанням ключового слова `posedge` або `negedge` і ставиться до вхідного й двонаправленого порту. Якщо порт є вектором, то фронт ставиться до молодшого біта. Якщо фронт не зазначений, то шлях розглядається при будь-якій зміні вхідного сигналу. Чутливий до фронту шлях сигналу може бути описаний з використанням повного (`*>`) або паралельного (`=>`) з'єднання. Параметр `data_source_expression` є необов'язковим виразом, що служить для опису по-

току даних до шляху призначення. Цей необов'язковий параметр не має впливу на реальну передачу даних або події моделі. Параметр *polarity* вказує, шлях даних є інверсним чи не інверсним.

Наступний код описує чутливий до фронту шлях з позитивним оператором полярності:

```
(posedge clock => ( out +: in ) ) = (10, 8);
```

По позитивному фронту синхросигналу *clock* значення із входу *in* до виходу *out* надійде через 10 одиниць часу, якщо вихід переходу з 0 -> 1, або через 10 одиниць у протилежному випадку. Шлях даних від входу *in* до виходу *out* прямий - значення від *in* не інвертується при надходженні на вихід *out*.

Декларація чутливого до фронту шляху з оператором негативної полярності може виглядати так:

```
(negedge clock[0] => ( out -: in ) ) = (10, 8);
```

У даному прикладі перемикання пристрою по задньому фронту *clock[0]* використовує затримку переднього фронту, рівну 10, і затримку заднього фронту, рівну 8. Полярність негативна, що означає: величина із входу *in* інвертується при передачі її на вихід *out*.

Наступний приклад демонструє використання декларації чутливого до фронту шляху без ідентифікатора фронту:

```
(clock => ( out : in ) ) = (10, 8);
```

У цьому прикладі вихід повинен змінюватися при будь-якій зміні *clock*.

13.2.4. Умовні шляхи затримок.

Величина затримки *pin-to-pin* може змінюватися залежно від значення вхідного сигналу. У цьому випадку при описі затримок використовуються умови. Умовну величину затримки часто називають залежною від стану затримкою шляху (*state dependent path delays*(SDPD)).

Синтаксис:

```
if ( conditional_expression ) simple_path_declaration
| if ( conditional_expression ) edge_sensitive_path_declaration
| ifnone simple_path_declaration
```

Операнди можуть бути входом (*input*) або двонаправленим (*inout*) портом, скалярним вектором, елементом або діапазоном вектора класу ланцюга і регістра, константою часу компіляції. У виразі можуть бути використані оператори: логічної, побітової, редукції конкатенації або умови. В операторі *if* не повинно бути гілки *else*.

Ключове слово *ifnone* використовується для позначення шляху, коли інші умови для нього мають значення *false*, у цьому випадку може бути описаний простий шлях модуля. Умовний шлях, що відповідає *if*, може бути або простим шляхом модуля, або чутливим до рівня. Не можна одночасно описувати той самий шлях за допомогою умовного *ifnone* і безумовного шляху:

```
if (C1) (IN => OUT) = (1,1);
```

```

ifnone (IN => OUT) = (2,2);
// add operation
if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
// pass-through i1 operation
if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
// pass-through i2 operation
if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
// all other operations
ifnone (i2 => o1) = (15.0, 15.0);
(posedge CLK => (Q += D)) = (1,1);
ifnone (CLK => Q) = (2,2);

```

Наступний опис містить помилку, оскільки для шляху `a => out` описана затримка з використанням умовного шляху `ifnone` і безумовного шляху:

```

if (a) (b => out) = (2,2);
if (b) (a => out) = (2,2);
ifnone (a => out) = (1,1);
(a => out) = (1,1);

```

Комбінаційна схема (лістинг 13.6) містить опис умовних затримок, результат її моделювання представлений на рис. 13.6.

Лістинг 13.6. Вентильна схема з умовними затримками.

```

module M1 (out, a, b, c, d);
    output out;
    input  a, b, c, d;
    wire e, f;
    specify
        //різні затримки шляху залежно від значення сигналу a
        if (a)(a => out) = 9;
        if (~a)(a => out) = 10;
        // умовний вираз містить два сигнали b і c
        if (b & c)(b => out) = 9;
        if ~(b & c)(b => out) = 13;
        // використання операторів конкатенації й повного з'єднання
        if ({c, d} == 2'b01) (c, d *> out) = 11;
        if ({c, d} != 2'b01) (c, d *> out) = 13;
    endspecify
    and a1 (e, a, b);
    and a2 (f, c, d);
    or a3 (out, e, f);
endmodule

```

Наступний приклад представляє умовний чутливий до фронту шлях, що описує затримку тригера, який має керуючі сигнали `reset` і `clear`:

```

if (!reset && !clear)
    (posedge clock => (out + : i n ) ) = (10, 8 );

```

Якщо передній фронт `clock` трапляється, коли `reset = 0` і `clear = 0`, шлях модуля поширюється від `clock` до `out`, використовуючи затримку переднього фронту 10 і заднього - 8.

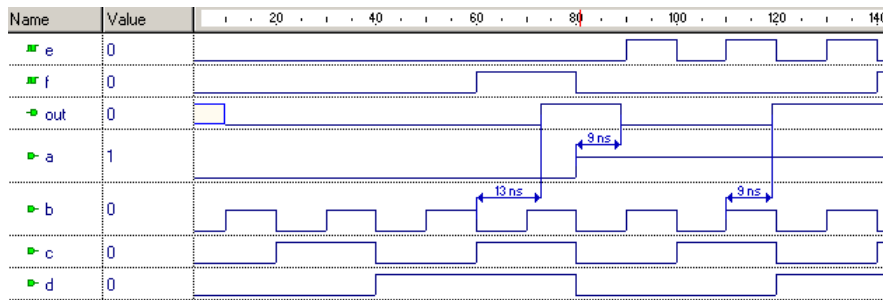


Рис. 13.6. Результат моделювання

Лістинг 13.7 представляє три декларації для чутливого до фронту шляху. Кожний шлях використовує унікальний фронт або умову.

Лістинг 13.7. Множинні описи умовного чутливого до рівня фронту шляху.

specify

```
(posedge clk => (q[0]:data)) = (10, 5);
(negedge clk => (q[0]:data)) = (20, 12);
if (reset)
  (posedge clk => (q[0]:data)) = (15, 8);
```

endspecify

Наступна декларація шляхів із двома станами - неправильна, навіть якщо вони використовують різні умови. У першому випадку виконується декларація затримки для діапазону елементів вектора, а в другому - для єдиного біта:

specify

```
if (reset)
  (posedge clk => (q[3:0]:data)) = (10, 5);
if (!reset)
  (posedge clk => (q[0]:data)) = (15, 8);
```

endspecify

13.2.5. Rise, fall і turn-off затримки.

У моделі затримок pin-to-pin може бути виконаний детальніший опис. Для кожного шляху може бути задано 1, 2, 3, 6 або 12 значень. Не дозволено вказувати 4, 5, 7, 8, 9, 10 або 11 параметрів. При цьому порядок опису затримок є фіксованим (лістинг 13.8).

Лістинг 13.8. Моделі затримок.

// Опис тільки однієї затримки. Використовується для всіх переходів

specparam t_delay = 11;

(clk => q) = t_delay;

// Опис двох затримок: rise і fall

//Rise використовується для перемикань 0-0->1, 0-0->z, z-z->1

//Fall використовується для перемикань 1-1->0, 1-1->z, z-z->0

specparam t_rise = 9, t_fall = 13;

(clk => q) = (t_rise, t_fall);

```
// Опис трьох затримок: rise, fall і turn-off
//Rise використовується для перемикачів 0-0->1, z-z->1
//Fall використовується для перемикачів 1-1->0, z-z->0
// Turn-off використовується для перемикачів 0-0->z, 1-1->z
спсparam t_rise = 9, t_fall = 13, t_turnoff = 11;
(clk => q) = (t_rise, t_fall, t_turnoff);
// Опис шести затримок
//Затримки описуються в порядку 0-0->1, 1-1->0, 0-0->z, z-z->1, 1-1->z, z-z->0
//Порядок запису затримок не можна змінювати
спсparam t_01 = 9, t_10 = 13, t_0z = 11;
спсparam t_z1 = 9, t_1z = 11, t_z0 = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);
// Опис дванадцяти затримок
// Затримки описуються в порядку 0-0->1, 1-1->0, 0-0->z, z-z->1, 1-1->z, z-z->0
// 0-0->x, x-x->1, 1-1->x, x-x->0, x-x->z, z-z->x
спсparam t_01 = 9, t_10 = 13, t_0z = 11;
спсparam t_z1 = 9, t_1z = 11, t_z0 = 13;
спсparam t_0x = 4, t_x1 = 13, t_1x = 5;
спсparam t_x0 = 9, t_xz = 11, t_zx = 7;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0,
             t_0x, t_x1, t_1x, t_x0, t_xz, t_zx);
```

Будь-яка затримка може бути представлена у формі min:typ:max. Наприклад:

```
// Опис трьох затримок: rise, fall і turn-off
//Кожна затримка має min:typ:max значення
спсparam t_rise = 8:9:10, t_fall = 12:13:14, t_turnoff = 10:11:12;
(clk => q) = (t_rise, t_fall, t_turnoff);
```

За замовчуванням при моделюванні використовується типова величина затримки. Вибір мінімальної, максимальної або типової величини затримок задається програмою моделювання.

13.2.6. Обробка x-переходів.

Verilog використовує песимістичний метод при обчисленні затримок для переходів у стан x. Це означає, що якщо затримка x-переходу явно не описана, то для переходу з x у визначений стан використовується максимальна можлива затримка, а для переходу з визначеного значення в x - мінімальна можлива затримка.

При моделюванні пристрою, затримки якого описані кодом:

```
// Опис шести затримок
//Затримки описуються в порядку 0-0->1, 1-1->0, 0-0->z, z-z->1, 1-1->z, z-z->0
//Порядок запису затримок не можна змінювати
спсparam t_01 = 9, t_10 = 13, t_0z = 11;
спсparam t_z1 = 9, t_1z = 11, t_z0 = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);
```

для x-переходів між станами використані значення з табл. 13.1.

Таблиця 13.1. Величина затримки для x-переходів

Перехід	Величина затримки
0->x	$\min(t_{01}, t_{0z}) = 9$
1->x	$\min(t_{10}, t_{1z}) = 11$
z->x	$\min(t_{z0}, t_{z1}) = 9$
x->0	$\max(t_{01}, t_{z0}) = 13$
x->1	$\max(t_{01}, t_{z1}) = 9$
x->z	$\max(t_{1z}, t_{0z}) = 11$

13.3. Перевірка часових параметрів

Дана процедура використовується для перевірки можливості порушення часових параметрів під час моделювання. Це необхідно для проектування критичних до часу високошвидкісних пристроїв, наприклад мікропроцесорів або DSP.

В Verilog існує велика кількість системних завдань для перевірки тимчасових параметрів. Всі перевірки часових параметрів можуть бути виконані тільки в блоці `specify`.

13.3.1. Перевірка часу установки й зберігання.

Для перевірки часу установки й зберігання послідовних елементів проекту використовуються завдання `$setup` і `$hold`, відповідно.

Синтаксис виклику системного завдання `$setup`:

`$setup(data_event, reference_event, limit [, notify_reg]);`

де `data_event` - спостережуваний сигнал, `reference_event` - відповідна подія, `limit` - значення часу установки.

Повідомлення про помилку буде видано, якщо:

$$(T_{\text{reference\_event}} - T_{\text{data\_event}}) < \text{limit}.$$

`specify`

`$setup(data, posedge clock, 3);`

`endspecify`

Приклад (лістинг 13.9) представляє використання функції `$setup` для перевірки часу установки. Функція виконує подію `notif_reg`, що запускає блок `always` з функцією `$display`, яка виводить повідомлення про порушення часу установки.

Лістинг 13.9. Модель із перевіркою часу установки.

```
module m( );
    reg notif_reg;
    wire data, clock;
    specify
        $setup(posedge data, posedge clock, 10, notif_reg);
    endspecify
    always @ notif_reg
```

```

$display("Setup violation in %m at time %t", $time);
endmodule

```

Синтаксис виклику системного завдання \$hold:

```
$hold(reference_event, data_event, limit);
```

Повідомлення про помилку буде видано, якщо:

$$(T_{data\_event} - T_{reference\_event}) < limit .$$

```
specify
```

```
$hold(posedge clear, data, 3);
```

```
endspecify
```

13.3.2. Перевірка ширини імпульсу.

Іноді необхідно виконувати перевірку ширини імпульсу.

Системне завдання \$width використовується для перевірки відповідності довжини імпульсу вимозі до мінімальної його ширини. Синтаксис завдання:

```
$width (reference_event, limit);
```

Параметр data_event не вказується безпосередньо при виклику системного завдання, він обчислюється як наступна подія сигналу reference_event.

Помилка виникає, якщо $(T_{data\_event} - T_{reference\_event}) < limit$.

```
specify
```

```
$width (posedge clock, 6);
```

```
endspecify
```

13.4. Системні завдання для визначення часових параметрів

1. Перевірка часу установки

```
$setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] );
```

2. Перевірка часу зберігання

```
$hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
```

3. Комбінована перевірка часу установки і зберігання

```
$setuphold (reference_event , data_event , timing_check_limit , timing_check_limit [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ] [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] );
```

4. Перевірка параметрів керуючих сигналів

```
$recovery (reference_event, data_event, timing_check_limit [ , [ notify_reg ] ] );
```

```
$removal (reference_event, data_event, timing_check_limit [ , [ notify_reg ] ] );
```

```
$recrem (reference_event, data_event, timing_check_limit , timing_check_limit
```

```
[ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
```

```
[ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] );
```

5. Перевірка зміщень між двома сигналами

```
$skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
```

```
$timeskew ( reference_event , data_event , timing_check_limit
```

```
[ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] );
```

```
$fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
```

```
[ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] );
```

6. Перевірка періоду синхросигналу

```
$period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] );
```

7. Перевірка ширини імпульсу

```
$width ( controlled_reference_event , timing_check_limit ,  
threshold [ , [ notify_reg ] ] );
```

8. Перевірка незмінюваності

```
$nochange ( reference_event , data_event , start_edge_offset ,  
end_edge_offset [ , [ notify_reg ] ] );
```

Лістинг 13.10 представляє приклад використання системних завдань для опису тимчасових параметрів тригера. Подібні моделі застосовуються для опису пристроїв після імплементації при виконанні часового аналізу системи.

Лістинг 13.10. Приклад тригера, що використовує блок specify.

```
primitive negdff(q, clock, data, preset, clear, notifier);  
    output q; reg q;  
    input clock, data, preset, clear, notifier;  
    table  
    //clock data p c notifier state q  
        f 0 1 1 ? : ? : 0;  
        f 1 1 1 ? : ? : 1;  
        n 1 1 ? 1:1 : 1;  
        n 0 1 ? ? : 0:0;  
        p ? ? ? ? : ? : -;  
        ? ? 0 1 ? : ? : -;  
        ? ? * 1 ? : 1:1;  
        ? ? 1 0 ? : ? : 0;  
        ? ? 1 0 ? : ? : 0;  
        ? ? ? ? * : ? : x;  
    endtable  
endprimitive  
module dff(q, qBar, clk, d, p, c);  
    output q, qBar;  
    input clk, d, p, c;  
    reg notifier;  
    and (e, p, c);  
    not (qBar, udp_out);  
    buf (q, udp_out);  
    negdff n(udp_out, clk, d, p, c, notifier);  
    specify  
    // Визначення значення часових параметрів specparam  
        specparam tsetup = 10, thold = 2, tclkwidth = 20, trecover = 5, t_pc_width;  
        specparam tPLHc = 4:6:9, tPHLc = 5:8:11;  
        specparam tPLHpc = 3:5:6, tPHLpc = 4:7:9;  
    // Затримки шляху даних  
        (clk *> q, qBar) = (tPLHc, tPHLc);  
        (p, c *> q, qBar) = (tPLHpc, tPHLpc);  
    // Перевірка часових параметрів  
        $setup(d, posedge clk && e, tsetup, notifier);
```

```

$hold(d, negedge clk && e, thold, notifier);
$period(negedge clk, twidth, notifier);
$width(negedge c, t_pc_width, notifier);
$width(negedge p, t_pc_width, notifier);
$recovery(posedge c, posedge clk, (recover, notifier);

endspecify
endmodule

```

13.5. Контрольні запитання й завдання

1. Який тип затримки використовується в схемі на рис. 13.7? Розробити Verilog-опис модуля Y.

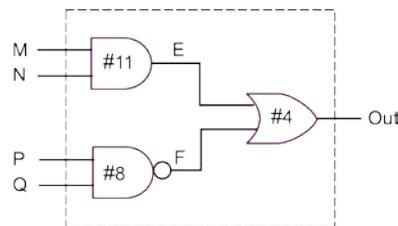


Рис. 13.7. Приклад схеми пристрою

2. Використовуючи максимальну затримку модуля, перетворити його модель в lumped delay. Написати Verilog-код модуля, застосовуючи затримку lumped delay.

3. Для завдання 1 обчислити затримку від кожного входу до виходу. Створити Verilog-опис модуля, використовуючи модель path delay.

4. D-тригер із синхронізацією заднім фронтом і асинхронним скиданням представлений на рис. 13.8. Створити Verilog-модель тригера D_FF. Показати тільки входні/вихідні порти й опис затримок шляху. Описати затримки шляху, використовуючи паралельне з'єднання.

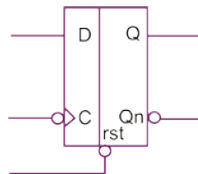


Рис. 13.8. D-тригер

Параметри пристрою:

D → Q = 5, D → Qn = 5, C → Q = 6, C → Qn = 7, rst → Q = 2, rst → Qn = 3.

5. Модифікувати D-тригер із завдання 4 таким чином, щоб всі затримки шляху від входів до виходів були рівні 5 одиницям часу. Описати затримки, використовуючи повне з'єднання для q і qbar.

6. Нехай для всіх шляхів вказано шість значень затримок. У блоці specify визначені параметри t_01 = 4, t_10 = 5, t_0z = 7, t_z1 = 2, t_1z = 3, t_z0 = 8. Використовувати ці значення для моделі D-тригера із завдання 4.

7. Модифікувати модель D-тригера із завдання 4 таким чином, щоб затримка пристрою залежала від значення на вході d. Якщо $d = 0$, то $\text{clock} \rightarrow q = 5$ і $\text{clock} \rightarrow \bar{q} = 4$; інакше $\text{clock} \rightarrow q = 6$, а $\text{clock} \rightarrow \bar{q} = 7$. В інших випадках затримка дорівнює 5.

8. Для моделі D-тригера із завдання 7 додати перевірку часових параметрів. Мінімальний час установки для входу d дорівнює 8, а час зберігання - 4. Сигнал reset активний по 1, мінімальна довжина імпульсу для сигналу reset дорівнює 42.

9. Промодельовати значення виходу для наступної моделі, що використовує блок specify:

```
module dff(q, d, clk, reset);
    input clk, d, reset;
    output q;
    specify
        specparam tRise_clk_q = 150, tFall_clk_q = 200;
        specparam tSetup = 70;
        (clk ==> q) = (tRise_clk_q, tFall_clk_q);
        (reset ==> q) = (20,10);
    endspecify
    dff_logic dffi (q, d, clk, reset);
endmodule

module dff_logic(q, data, clk, reset);
    reg q;
    output q;
    wire clk, reset, data;
    always @posedge clk
        q = data;
    always @reset
        if (reset == 1)
            deassign q;
        else
            assign q = 0;
endmodule
```

14. СИНТЕЗ І ВЕРИФІКАЦІЯ ПОСЛІДОВНОГО СУМАТОРА

14.1. RTL- модель пристрою

Проектування системи, що складається з керуючого і операційного автоматів, представлено на прикладі послідовного 4-розрядного суматора, структурна схема якого зображена на рис. 14.1. Два зсувних регістри використовуються для зберігання чотирибітних доданків, X і Y . Квадрат ліворуч від кожного регістра зображує входи: Sh (зрушення), SI (послідовний вхід) і $Clock$ (синхронізація). Якщо $Sh=1$, при надходженні активного синхроімпульсу значення SI заноситься в x_3 (або в y_3), а весь вміст регістра зсувається вправо на одну позицію. Регістр X є акумулятором, і після виконання четвертої операції зсуву він містить суму $X + Y$. Другий регістр - це циклічний зсувний регістр. Після четвертої операції зсуву він повертається в початковий стан і значення Y не втрачається. Операція додавання реалізується повним суматором і тригером, що зберігає перенос. В один момент часу виконується додавання однієї пари бітів. Коли $Sh = 1$, по задньому фронту синхроімпульсу біт суми заноситься в акумулятор, а біт переносу зберігається в тригері. Додаткові зв'язки, необхідні для початкового завантаження регістрів (X і Y) і для очищення тригера переносу, на рис. 14.1 не показані.

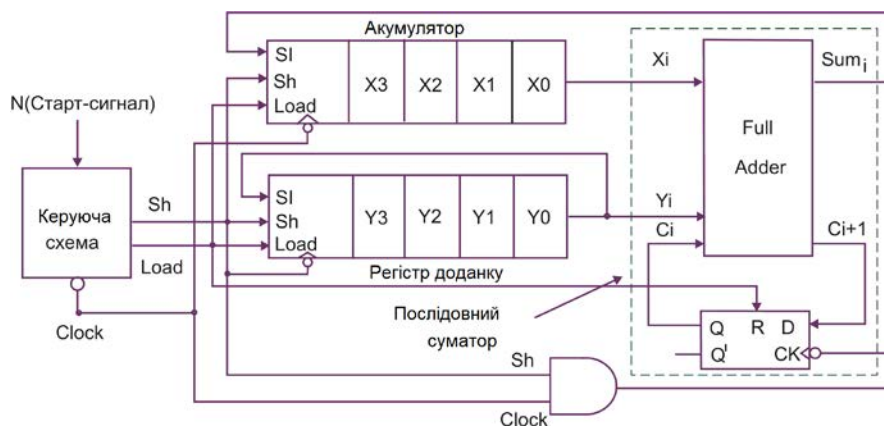


Рис. 14.1. Структурна схема послідовного 4-розрядного суматора

Табл. 14.1 описує операції послідовного суматора. Тут представлені моменти часу: t_0 – до першого синхроімпульсу, t_1 – після першого синхроімпульсу, t_2 – після другого синхроімпульсу і т.п. У початковий момент часу акумулятор містить число X , другий регістр містить Y і тригер переносу перебуває в стані 0.

Оскільки повний суматор – це комбінаційна схема, то додавання $x_0 = 1$, $y_0 = 1$ і $c_0 = 0$ буде виконано після невеликої затримки, наприклад 10 ns, з формуванням результату: $sum_0 = 0$ і перенос $c_1 = 1$. Після першого фронту синхросигналу сума sum_0 заноситься в акумулятор і число, що залишилося в ньому, зсувається вправо на одну позицію. Тоді ж перенос c_1 зберігається в тригері й виконується зсув другого регістра вправо на 1 позицію. На входи повного суматора надходить наступна пара бітів: $x_1 = 0$ і $y_1 = 1$. Результат додавання: $sum_1 = 0$ і $c_1 = 1$. По другому синхроімпульсу сума sum_1 надходить в акумулятор, c_2 зберігається в тригері переносу, виконується циклічний зсув другого регістра. Біти x_2 й y_2 подаються тепер на входи суматора, і процес моделювання триває, поки не буде виконане додавання всіх пар бітів. Після четвертого синхроімпульсу в акумуляторі перебуває сума X і Y , а другий регістр повертається в початковий стан.

Таблиця 14.1. Операції послідовного суматора

	X	Y	c_i	sum_i	c_{i+1}
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)

Керуюча схема реалізується за допомогою автомата Мура, граф переходів якого зображений на рис. 14.2. Пристрій містить один керуючий вхід N і два виходи Sh і $Load$. Стан автомата - S 0-S5. У вершинах графа автомата зазначені тільки ті виходи, значення яких у даному стані рівні 1. Наприклад, у стані S1: $Load = 1$, а $Sh = 0$; у стані S2: $Load = 0$, а $Sh = 1$. Тоді в стані S0 $Load = Sh = 0$.

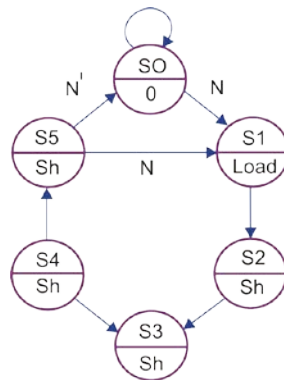


Рис. 14.2. Керуючий автомат суматора

Для керуючого автомата й регістрів розроблені окремі компоненти (лістинги 14.1 і 14.2). Модель верхнього рівня, що формує послідовний суматор (лістинг 14.3), містить спільні компоненти, повний суматор, який реалізований булевими функціями в операторі assign, а також представлений оператором always регістр переносу.

Лістинг 14.1. Verilog-модель керуючого автомата.

```
'timescale 1 ns / 10 ps
module FSM
    (input Clk, Reset, N, output Sh, Load);
// Визначення міток станів
    parameter S0=3'b000;    parameter S1=3'b001;
    parameter S2=3'b010;    parameter S3=3'b011;
    parameter S4=3'b100;    parameter S5=3'b101;
    reg [2:0] state, next_state;
// Блок Always для послідовної логіки
    always @(posedge Clk or posedge Reset)
        if (Reset) state <= S0;
        else state <= next_state;
// Блок Always для комбінаційної логіки
    always @(state or N)
        case (state)
            S0 : if (N)next_state <= S1;
                  else    next_state <= S0;
            S1 : next_state <= S2;
            S2 : next_state <= S3;
            S3 : next_state <= S4;
            S4 : next_state <= S5;
            S5 : if (N) next_state <= S1;
                  else next_state <= S0;
            default : next_state <= S0;
        endcase
    assign Sh = (state==S2)||((state==S3)||((state==S4)||((state==S5)? 1'b1:1'b0;
    assign Load = (state==S1)? 1'b1:1'b0;
endmodule
```

Лістинг 14.2. Verilog-модель регістра.

```
'timescale 1 ns / 10 ps
module register
    (input Clk, Load, Sh, Sl, input [3:0] D,
     output reg [3:0] Q);
always @(negedge Clk)
    if (Sh)
        Q <= {Sl, Q[3:1]};
    else if (Load)
        Q<=D;
endmodule
```

Лістинг 14.3. Verilog-модель послідовного суматора.

```

'timescale 1 ns / 10 ps
module seq_adder
    (input N, clk, reset, input [3:0] X, Y,
     output [4:0] result);
wire Sh, Load, sum, Cout;
reg Cin;
wire [3:0] Xin, Yin;
    FSM U1(.Clk(clk), .Reset(reset), .N(N), .Sh(Sh), .Load(Load));
    register Reg_X (.Clk(clk), .Load(Load), .Sh(Sh), .Sl(sum), .D(X), .Q(Xin));
    register Reg_Y (.Clk(clk), .Load(Load), .Sh(Sh), .Sl(Yin[0]), .D(Y), .Q(Yin));
    // Full Adder
    assign sum = Xin[0]^Yin[0]^Cin;
    assign Cout = (Xin[0]&Yin[0])|(Xin[0]&Cin)|(Yin[0]&Cin);
    // flip-flop Cin
    always @(negedge clk or posedge Load)
        if (Load) Cin <= 1'b0;
        else if (Sh) Cin <= Cout;
    assign result={Cin,Xin};
endmodule

```

Для тестування суматора розроблений testbench (лістинг 14.4). Вхідні тестові набори описані масивами константних значень X_test[5:0] і Y_test[5:0], які задаються в операторі initial. Другий оператор initial використовується для формування синхросигналу і виконання скидання в початковий стан перед тестуванням. Результати моделювання представлені в табл. 14.2.

Лістинг 14.4. Testbench для тестування послідовного суматора.

```

'timescale 1 ns / 10 ps
module seq_adder_tb;
// Декларації внутрішніх сигналів
reg N, clk, reset;
reg [3:0]X, Y;
wire [4:0]result;
reg [3:0] X_test [5:0], Y_test [5:0];
parameter period = 'd 20;
reg ERROR;
// Тестований модуль
    seq_adder UUT (.N(N), .clk(clk), .reset(reset),
                  .X(X), .Y(Y), .result(result));
initial
    begin
        X_test [0] = 4'b0000;    X_test [1] = 4'b0001;
        X_test [2] = 4'b1000;    X_test [3] = 4'b1111;
        X_test [4] = 4'b1010;    X_test [5] = 4'b0101;
        Y_test [0] = 4'b0000;    Y_test [1] = 4'b0001;
        Y_test [2] = 4'b1000;    Y_test [3] = 4'b1111;
        Y_test [4] = 4'b1010;    Y_test [5] = 4'b0101;
    end
endmodule

```

```
// Формування сигналів синхронізації й скидання
initial
begin
    clk=1'b0; reset=1'b1;
    #5 reset<=1'b0;
    forever #(period/2) clk=~clk;
end
integer i, j;
reg [3:0]A,B;
reg [4:0] SUM, Sum_etalon;
always
    for(i=0; i<=5; i=i+1)
        begin
            X = X_test[i];
            for(j=0; j<=5; j=j+1)
                begin
                    N = 1'b1;
                    Y = Y_test[j];
                    N = repeat (2) @(negedge clk) 1'b0;
                    //читання результатів додавання
                    #(5*period) A=X; B=Y;
                    SUM=result;
                    Sum_etalon={1'b0,X} + Y;
                    ERROR = (SUM!=Sum_etalon)? 1'b1:1'b0;
                    $display($time," ns: A=%d, B=%d, Res=%d, E= %b ",
                        A, B, SUM, ERROR);
                end
            end
        end
endmodule
```

14.2. Синтез керуючого автомата

Для синтезу пристрою використовувалася програма ISE WebPack. На рис. 14.3 зображена схема RTL-синтезу пристрою. Лістинг 14.5 приводить фрагмент результату синтезу. З нього видно, що система виявила автомат і для кодування станів використовувала код Грея.

Лістинг 14.5. Фрагмент звіту про синтез керуючого автомата.

```
=====
*                      HDL Analysis                      *
=====
Analyzing top module <FSM>.
    S0 = 3'b000
    S1 = 3'b001
    S2 = 3'b010
    S3 = 3'b011
    S4 = 3'b100
    S5 = 3'b101
Module <FSM> is correct for synthesis.
```

```
=====
*                      HDL Synthesis                      *
=====

Synthesizing Unit <FSM>.
  Found finite state machine <FSM_0> for signal <state>.

-----
| States           | 6 |
| Transitions      | 8 |
| Inputs           | 1 |
| Outputs          | 2 |
| Clock            | Clk (rising_edge) |
| Reset            | Reset (positive)  |
| Reset type       | asynchronous       |
| Reset State      | 000                |
| Encoding         | automatic           |
| Implementation   | LUT                 |
-----

Summary:
inferred 1 Finite State Machine(s).
Unit <FSM> synthesized.

=====
*                      Advanced HDL Synthesis              *
=====

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state> on signal <state[1:3]> with gray encoding.

-----
State | Encoding
-----
000 | 000
001 | 001
010 | 011
011 | 010
100 | 110
101 | 111
-----

=====
Advanced HDL Synthesis Report
Macro Statistics
# FSMs           : 1
# Registers      : 3
Flip-Flops       : 3
=====
```

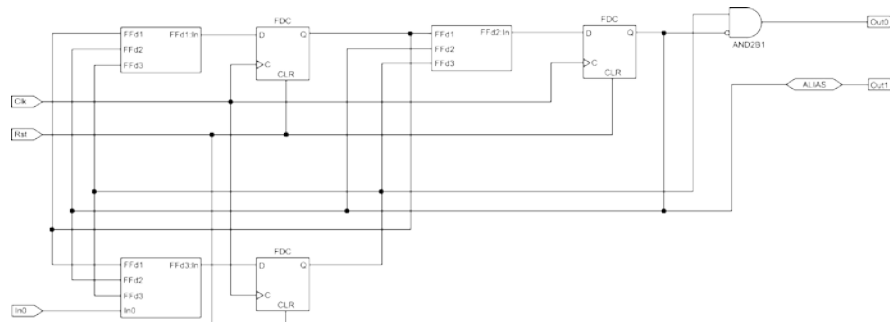


Рис. 14.3. Післясинтезна RTL-схема керуючого автомата

Таблиця 14.2. Результати тестування послідовного суматора

TIME	A	B	SUM	Sum_etalon	ERROR
0.000	?	?	?	?	0
140.000 ns	0	0	0	0	0
280.000 ns	0	1	1	1	0
420.000 ns	0	8	8	8	0
560.000 ns	0	15	15	15	0
700.000 ns	0	10	10	10	0
840.000 ns	0	5	5	5	0
980.000 ns	1	0	1	1	0
1.120 us	1	1	2	2	0
1.260 us	1	8	9	9	0
1.400 us	1	15	16	16	0
1.540 us	1	10	11	11	0
1.680 us	1	5	6	6	0
1.820 us	8	0	8	8	0
1.960 us	8	1	9	9	0
2.100 us	8	8	16	16	0
2.240 us	8	15	23	23	0
2.380 us	8	10	18	18	0
2.520 us	8	5	13	13	0
2.660 us	15	0	15	15	0
2.800 us	15	1	16	16	0
2.940 us	15	8	23	23	0
3.080 us	15	15	30	30	0
3.220 us	15	10	25	25	0
3.360 us	15	5	20	20	0
3.500 us	10	0	10	10	0
3.640 us	10	1	11	11	0
3.780 us	10	8	18	18	0
3.920 us	10	15	25	25	0
4.060 us	10	10	20	20	0
4.200 us	10	5	15	15	0

14.3. Синтез операційного пристрою в цілому

Схема пристрою, сформована в результаті RTL-синтезу, зображена на рис. 14.4. Частота пристрою після синтезу:

Minimum period: 8.398ns (Maximum Frequency: 119.076MHz).

Розмір проекту з мар-звіту (лістинг 14.6) дорівнює 189 еквівалентних венти-лів. Використовується 12 тригерів і 15 блоків LUT.

Лістинг 14.6. Фрагмент мар-звіту.

Design Summary

---i ---i ---i ---i ---i ---i

Number of errors: 0

Number of warnings: 2

Logic Utilization:

Number of Slice Flip Flops: 12 out of 1,920 1%
Number of 4 input LUTs: 15 out of 1,920 1%

Logic Distribution:

Number of occupied Slices: 8 out of 960 1%
Number of Slices containing only related logic: 8 out of 8 100%
Number of Slices containing unrelated logic: 0 out of 8 0%
*See NOTES below for an explanation of the effects of unrelated logic

Total Number of 4 input LUTs: 15 out of 1,920 1%
Number of bonded IOBs: 16 out of 66 24%
Number of GCLKs: 1 out of 24 4%

Total equivalent gate count for design: 189

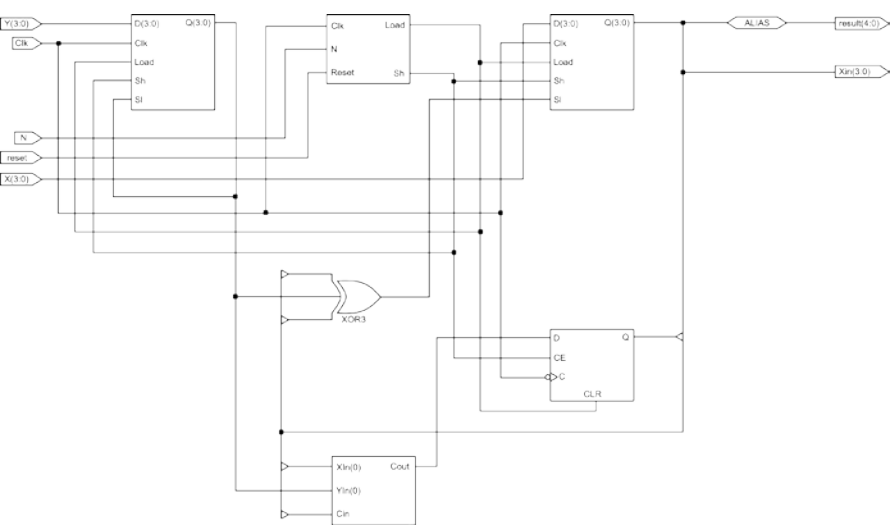


Рис. 14.4. Сформована в результаті синтезу RTL-схема пристрою

14.4. Оцінка часових параметрів

Часові характеристики пристрою наведені в лістингу 14.7, на основі яких отримано, що максимальне значення:

1) Clock Clk to Pad = 7,396 ns;

2) Clock to Setup on destination clock Clk = 8,106 ns.

Таким чином, мінімальний період суматора дорівнює 8,106 ns, а частота 123,3 МГц.

Лістинг 14.7. Фрагмент файлу звіту.

```
Setup/Hold to clock clk
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
| Setup to | Hold to | | Clock |
Source | clk (edge) | clk (edge) | Internal Clock(s) | Phase |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
N | 1.616(R) | -0.111(R) | clk_BUFGP | 0.000 |
X<0> | 1.111(F) | 0.293(F) | clk_BUFGP | 0.000 |
X<1> | 0.665(F) | 0.649(F) | clk_BUFGP | 0.000 |
X<2> | 0.705(F) | 0.617(F) | clk_BUFGP | 0.000 |
X<3> | 1.695(F) | -0.174(F) | clk_BUFGP | 0.000 |
Y<0> | 1.211(F) | 0.213(F) | clk_BUFGP | 0.000 |
Y<1> | 0.866(F) | 0.489(F) | clk_BUFGP | 0.000 |
Y<2> | 0.953(F) | 0.419(F) | clk_BUFGP | 0.000 |
Y<3> | 1.174(F) | 0.242(F) | clk_BUFGP | 0.000 |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
Clock clk to Pad
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
| clk (edge) | | Clock |
Destination | to PAD | Internal Clock(s) | Phase |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
result<0> | 8.061(F) | clk_BUFGP | 0.000 |
result<1> | 8.106(F) | clk_BUFGP | 0.000 |
result<2> | 7.893(F) | clk_BUFGP | 0.000 |
result<3> | 7.803(F) | clk_BUFGP | 0.000 |
result<4> | 8.052(F) | clk_BUFGP | 0.000 |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
Clock to Setup on destination clock clk
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
| Src:Rise | Src:Fall | Src:Rise | Src:Fall |
Source Clock | Dest:Rise | Dest:Rise | Dest:Fall | Dest:Fall |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
clk | 3.293 | | 3.698 | 3.000 |
---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+---i---i---i---i---i+
```

14.5. Контрольні запитання

1. Який програмний продукт був використаний для синтезу керуючого автомата?
2. Чи можна testbench для верифікації послідовного суматора використовувати при тестуванні готового цифрового виробу на кристалі?
3. Як називається граф, у якому представлений керуючий автомат суматора?

4. Скільки операцій містить цикл послідовного суматора для виконання функціональності?
5. Яке функціональне навантаження виконують керуючий і операційний автомати послідовного суматора?
6. Скільки станів має керуючий автомат Мура послідовного суматора?
7. Яку інформацію надає користувачу файл звіту про результати синтезу?

14.6. Література

1. Samir Palnitkar. Verilog HDL. A guide to digital design and synthesis.— SunSoft Press.— 1996.— 396p.
2. Bhasker J. Verilog HDL Synthesis. A Practical Primer.— Allentown: Star Galaxy Publishing, 1998. — 215 p.
3. Donald E. Thomas, Philip R. Moorby. The Verilog Hardware Description Language.— New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.— 2002.— 404 p.
4. James M. Lee. Verilog Quickstart. A Practical Guide to Simulation and Synthesis in Verilog.— New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers.— 2002.— 378 p.
5. IEEE Std 1364.1-2002. IEEE Standard for Verilog® Register Transfer Level Synthesis— IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 3 Park Avenue, New York, 2002.— 109 p.
6. IEEE Std 1364-1995. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language— IEEE Standard Verilog Hardware Description Language.— IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 345 East 47th Street, New York, NY 10017-2394, USA — 1996.— 675 p.
7. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).— IEEE Standard Verilog Hardware Description Language.— IEEE Computer Society Sponsored by the Design Automation Standards Committee.— Published by The Institute of Electrical and Electronics Engineers, Inc.— 3 Park Avenue, New York, NY 10016-5997, USA.— 2001.— 791 p.
8. Cunningham P., Swinnen M., Wilcox S. Clock Concurrent Optimization.— February, 2009.— www.azuro.com

ЧАСТИНА V. SYSTEM VERILOG

Розглядаються конструктивні особливості, синтаксис і семантика мови опису апаратури SystemVerilog, орієнтовані на проектування і верифікацію гетерогенних (цифрових) систем на кристалах. У розд. 15-17 представлені типи даних SystemVerilog, структури проєктів, конструкції операторів, процедурних блоків, завдань і функцій, основи об'єктно-орієнтованого програмування (ООП).

Мета - надання вже не новачкові, але ще не конструктору, технологічної культури від провідних компаній планети (Cadence, Synopsys, Mentor Graphics, Aldec) у галузі проектування, аналізу й синтезу спеціалізованих цифрових систем на кристалах шляхом використання мовного середовища високого рівня - SystemVerilog до опису й функціональної верифікації численних прикладів апаратної реалізації компонентів.

Завдання вивчення: 1. Типи даних SystemVerilog: простор для декларації об'єктів в SystemVerilog - пакети, поняття одиниці компіляції; одиниці часу і точність їхнього обчислення; нововведення в описі символічних значень - опис типів даних, вбудовані типи даних, типи ланцюгів (Net Types); логічні типи; типи даних із двома станами; рядки; інші абстрактні типи даних SystemVerilog; перетворення типів; розмір виразів; константи; користуальницькі типи даних (User-defined types); спільно використовуване визначення типу typedef; типи перелічення - визначення значень типів перелічення, послідовності міток у типі перелічення, завдання базового типу для типу міток, підпрограми й функції для типів перелічення. 2. Складні типи даних SystemVerilog: Arrays, Structures і Unions: структури (Structures) - створення визначених користувачем структур, упаковані структури; об'єднання (union) - упаковані об'єднання, приклад використання структур і об'єднань; масиви фіксованого розміру - декларація й ініціалізація, базові операції масивів (for і foreach), базові операції масивів (copy і compare); упаковані масиви (Packed arrays); призначення упакованих масивів; приклад використання масивів; динамічні масиви; черги - методи черг, пам'ять FIFO, що параметризується; асоціативні масиви, методи асоціативних масивів; зв'язані списки (Linked Lists); методи масивів - методи редукції масивів, методи місця розташування. 3. Процедурні блоки, завдання й функції; Void функції; процедурні оператори й підпрограми; оператори переходу - значення тимчасових параметрів, Finite State Machines; SystemVerilog ієрархія проєкту - прототип модуля, з'єднання портів, псевдоніми ланцюгів, передача даних через порти модуля, посилальні порти (Reference ports), типи даних, що параметризуються; приклади використання інтерфейсів і пам'яті; основи ООП - перший клас, місце визначення класів, ООП термінологія, створення нових об'єктів; блоки синхронізації модулів і тестових послідовностей (Stimulus Timing), блок clocking; об'єднання блоків, приклад Four-Port ATM Router; SystemVerilog Assertions - прямі (процедурні) асерції, паралельні асерції; створення випадкових значень (Randomization), найпростіший клас із псевдовипадковими змінними.

Джерела: синтез, моделювання й верифікація гетерогенних компонентів цифрової апаратури за допомогою мовного середовища SystemVerilog [5-8]; синтаксичні й семантичні особливості мови опису SystemVerilog [1-4, 9]; стандарт мови опису апаратури SystemVerilog і його використання в технологічних процесах EDA [1-11].

Мова SystemVerilog особливо актуальна в даний час, коли апаратна реалізація ідеї для ринку електронних технологій представляється найкращою. Свідченням тому може служити опублікована в журналі IEE Times [2009, December] топ-десятка від Gartner Research Group (Gary Smith) найбільш актуальних технологій:

1. Біологічний зворотний зв'язок або електроніка, керована думкою.
2. Друкована електроніка на основі використання органічних матеріалів.
3. Пластикові пам'яті на основі полімерів, що проявляють фероелектричні властивості.
4. Безмасочна літографія на основі використання електронного пучка для створення топології схеми.
5. Паралельна обробка даних для багатоядерних гетерогенних графічних процесорів.
6. Збір енергії від механічних, електромагнітних і електричних процесів у навколишньому середовищі.
7. Біоелектроніка й wetware - сполучення біологічних об'єктів та електроніки для медицини.
8. Резистивний ОЗП або мемристор, що демонструє ефект пам'яті і є четвертим пасивним елементом електронної схеми, поряд з резистором, конденсатором і індуктивністю.
9. Перехідні отвори в кремнії (Through-Silicon-Via - TSV) для створення реальних 3 D-SiP і кристалів.
10. Різні технології виготовлення батарей на основі нікелю й літію.

У топ-десятці знайшли відображення тільки технології, пов'язані з розробками на апаратному рівні, хоча розвиток програмних технологій також буде впливати на стан ринку електроніки. Глобально важливими залишаються технології зниження споживаної потужності й застосування рішень, спрямованих на зменшення змісту кошових матеріалів у продукті. Ці технології виступають рушіями багатьох напрямків розвитку електроніки, пов'язаних з людською діяльністю.

15. ТИПИ ДАНИХ SYSTEM VERILOG

15.1. Простір для декларації об'єктів в SystemVerilog

15.1.1. Пакети.

Verilog дозволяє визначати змінні тільки всередині модуля або блоків (begin-end і fork-join). В SystemVerilog введені пакети, що мають аналоги в VHDL. Пакети в SystemVerilog визначаються між ключовими словами package-endpackage, можуть включати синтезовані конструкції, опис яких може бути включено в пакет:

- 1) Визначення параметрів parameter і localparam. Параметр може бути перевизначений, локальний параметр ні. Однак у пакеті не допускається перевизначення параметрів, і в цьому випадку parameter і localparam стають синонімами.
- 2) Визначення констант const.
- 3) Визначені користувачем типи даних typedef.
- 4) Визначення автоматичних завдань і функцій (task і function).
- 5) Оператори імпорту з інших пакетів import.
- 6) Визначення переважання операторів.

Пакет може також містити глобальні змінні, визначення статичних змінних і функцій. Однак ці конструкції не синтезовані.

Пакет є незалежним декларативним об'єктом і ніколи не записується всередині модуля. Приклад опису пакета представлений лістингом 15.1. У ньому використовуються нові типи даних, які докладно будуть розглянуті далі: 1) logic – 1-бітова змінна, що має 4 стани, подібна до типу reg мови Verilog, може бути використана для створення вектора будь-якого розміру; 2) enum – тип перелічення, що представляє собою безліч міток; 3) typedef – визначений користувачем тип даних, що створюється з вбудованих і користувацьких типів даних; 4) struct – колекція змінних, до яких можна звертатися окремо і всім разом.

Лістинг 15.1. Визначення пакета.

```
package definitions;
    parameter VERSION = "1.1";
    typedef enum {ADD, SUB, MUL} opcodes_t;
    typedef struct {
        logic [31:0] a, b;
        opcodes_t opcode;
    } instruction_t;
    function automatic [31:0] multiplier (input [31:0] a, b);
        // code for a custom 32-bit multiplier goes here
        return a * b; // abstract multiplier (no error detection)
    endfunction
endpackage
```

Звертання до об'єктів пакета в модулі може бути виконане через:

- 1) пряме посилання, за допомогою оператора зони видимості (scope resolution(::)) (лістинг 15.2);
- 2) імпорт елементів пакета в модуль або інтерфейс (лістинг 15.3);
- 3) груповий імпорт елементів пакета в модуль або інтерфейс (імпортуються тільки використовувані елементи) (лістинг 15.4);
- 4) декларативний простір \$unit.

Лістинг 15.2. Використання оператора зони видимості.

```
module ALU
    (input definitions::instruction_t IW,
     input logic clock,
     output logic [31:0] result);

    always_ff @(posedge clock) begin
        case (IW.opcode)
            definitions::ADD : result = IW.a + IW.b;
            definitions::SUB : result = IW.a - IW.b;
            definitions::MUL : result = definitions::multiplier(IW.a, IW.b);
        endcase
    end
endmodule
```

Лістинг 15.3. Імпорт окремих елементів.

// Імпортування для визначення типів портів не допускається

```
module ALU
    (input definitions::instruction_t IW,
     input logic clock,
     input logic [31:0] result);
    import definitions::ADD;
    import definitions::SUB;
    import definitions::MUL;
    import definitions::multiplier;

    always_comb begin
        case (IW.opcode)
            ADD : result = IW.a + IW.b;
            SUB : result = IW.a - IW.b;
            MUL : result = multiplier(IW.a, IW.b);
        endcase
    end
endmodule
```

Лістинг 15.4. Груповий імпорт елементів.

```
module ALU
    (input definitions::instruction_t IW,
     input logic clock,
     output logic [31:0] result);
    import definitions::*; // wildcard import

    always_comb begin
        case (IW.opcode)
```

```

        ADD : result = IW.a + IW.b;
        SUB : result = IW.a - IW.b;
        MUL : result = multiplier(IW.a, IW.b);
    endcase
end
endmodule

```

15.1.2. Поняття одиниці компіляції.

SystemVerilog вводить концепцію, яка називається одиницею компіляції (compilation unit), що складається із всіх файлів, компільованих в один і той же час. Одиниця компіляції являє собою засіб для незалежної компіляції блоків, з яких складається цілий проект.

SystemVerilog дозволяє виконувати декларацію за межами пакета, модуля, інтерфейсу або програми. Така зовнішня декларація є compilation-unit scope (зона видимості одиниці компіляції) і її елементи доступні у всіх модулях, які декларуються в той же самий час.

Зона видимості одиниці компіляції може включати:

- 1) декларації одиниць виміру й точності тимчасових параметрів;
- 2) декларації змінних, ланцюгів і констант;
- 3) визначені користувачем типи даних, створені за допомогою конструкцій typedef, enum або class;
- 4) визначення завдань і функцій.

Приклад з лістингу 15.5 ілюструє використання зовнішніх декларацій констант, користувацького типу й функції. Даний код є несинтезованим.

Лістинг 15.5. Зовнішні декларації в області одиниць компіляції

```

// ---i ---i ---i ---i ---i ---i ---i ---i Зовнішня декларація ---i ---i ---i ---i ---i ---i ---i
parameter VERSION = "1.2a";           // зовнішня константа
    reg reset = 1;                     // зовнішня змінна (активна при 0))
    typedef struct packed {           // зовнішній, визначений користувачем тип
        reg [31:0] address;
        reg [31:0] data;
        reg [ 7:0] opcode;
    } instruction_word_t;
    function automatic int log2 (input int n); // зовнішня функція
        if (n <=1) return(1);
            log2 = 0;
            while (n > 1) begin
                n = n/2;
                log2++;
            end
        return(log2);
    endfunction
// ---i ---i ---i ---i ---i ---i ---i ---i Визначення модуля ---i ---i ---i ---i ---i ---i ---i

```

```
// зовнішні декларації використовуються для визначення типів портів module register
(output instruction_word_t q,
    input instruction_word_t d,
    input wire clock );
always @(posedge clock, negedge reset)
    if (!reset) q <= 0; // Використання зовнішнього сигналу скидання
    else q <= d;
endmodule
```

Небажано виконувати декларації в зоні видимості одиниць компіляції. Всі декларації варто робити в пакетах, що мають індивідуальні імена, а в зоні видимості виконувати імпортування цього пакета.

Зовнішнє декларування може бути використане для спрощення опису портів модуля ALU із прикладу, представленого лістингом 15.4, як показано в лістингу 15.6.

Лістинг 15.6. Імпорт пакета через декларативний простір \$unit.

```
import definitions::instruction_t;
module ALU
    (input instruction_t IW,
     input logic clock,
     output logic [31:0] result
    );
    або
    //Символ групового імпорту елементів пакета
    import definitions::*;
    module ALU
        (input instruction_t IW,
         input logic clock,
         output logic [31:0] result
        );
```

15.2. Одиниці часу і точність їхнього обчислення

В Verilog при описі часових параметрів у моделі одиниці виміру часу не вказувалися. Для їхнього визначення використовувалася директива 'timescale. У цьому випадку величина затримок модулів, що не має директиви 'timescale, залежала від порядку компіляції файлу (рис. 15.1). У наступному прикладі затримка модуля В буде дорівнювати 5 ns або 5 ms.

SystemVerilog дозволяє вказувати одиниці виміру разом з величиною затримки:

```
forever #5ns clock = ~clock;
```

Визначені стандартом одиниці виміру представлені в табл. 15.1.

Не допускається ставити пропуск між величиною затримки й одиницями виміру:

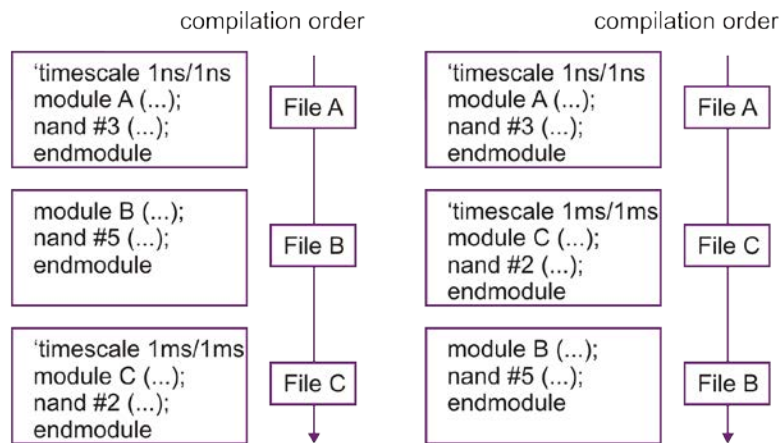
```
#3.2ps // legal
```

```
#4.1 ps // illegal: no space allowed
```

SystemVerilog дозволяє виконувати оголошення одиниць виміру та точності часових параметрів локально для кожного модуля, інтерфейсу або програми.

Наприклад:

```
module chip (...);
    timeunit 1ns;
    timeprecision 10ps;
    ...
endmodule
```



Module B delays are in nanoseconds Module B delays are in milliseconds

Рис. 15.1. Залежність одиниць часу від порядку компіляції модулів

Таблиця 15.1. Одиниці виміру часу в SystemVerilog

Unit	Description
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds
step	the smallest unit of time being used by the software tool (used in SystemVerilog testbench clocking blocks)

Як і для директиви `'timescale`, одиниці виміру можуть бути представлені в масштабі 1, 10 або 100. Опис виконується відразу після заголовка, перед всіма іншими операторами:

```
module adder (input wire [63:0] a, b,
              output reg [63:0] sum,
              output reg carry);
```



```

        timeunit 1ns;
        timeprecision 10ps;
        ...
    endmodule

```

15.3. Нововведення в описі символьних значень

У мові Verilog наступні оператори використовують операції доповнення до 1 (ones complement operator) і до 2 (two's complement operator) для заповнення вектора одиницями:

```

data = ~0;          // зворотний код
data = -1;          // додатковий код

```

В SystemVerilog для заповнення векторів значеннями пропонуються спеціальні символи:

- 1) '0 заповнює всі біти з лівого боку 0;
- 2) '1 заповнює всі біти з лівого боку 1;
- 3) 'z або 'Z заповнює всі біти з лівого боку z;
- 4) 'x або 'X заповнює всі біти з лівого боку x.

15.3.1. Опис типів даних.

В SystemVerilog введені такі нові типи даних:

- 1) Із двома станами (ключове слово bit): краща продуктивність, мінімізація використання пам'яті.
- 2) Черги (Queues), динамічні й асоціативні масиви (dynamic і associative arrays), автоматична пам'ять (automatic storage): мінімізація використання пам'яті, вбудована підтримка пошуку й сортування.
- 3) Об'єднання (Unions) і пакети (packed): пропонують множинне подання тих самих даних.
- 4) Класи й структури (Classes і structures): підтримують структури абстрактних даних.
- 5) Рядки (Strings): вбудована підтримка рядків.
- 6) Типи перелічення (Enumerated types): спрощення запису та аналізу коду.

15.3.2. Вбудовані типи даних.

Verilog-1995 містить два класи типів даних: змінні (register) і ланцюги (net), які містять змінні чотирьох станів: 0, 1, Z і X. Код RTL використовує змінні для збереження комбінаційних і послідовних значень. Змінні можуть бути беззнакові (reg [7:0] m), знакові 32-бітові (integer), беззнакові 64-бітові (time), із плаваючою точкою (real). Змінні можуть групуватися в масиви фіксованого розміру. Вся пам'ять є статичною, і підпрограми не використовують стек для зберігання аргументів і локальних значень. Ланцюги (net) застосовуються для з'єднання частин проекту, таких як вентильні примітиви і копії модулів.

SystemVerilog додає багато нових типів даних для проектування й верифікації апаратури.

15.3.3. Типи ланцюгів (Net Types).

Verilog дозволяє використовувати змінні типу ланцюга без їхньої декларації. Однак ця властивість може призвести до помилок, які важко виявляються. В Verilog-2001 з'явилася можливість за допомогою директиви 'default_nettype none виконувати блокування декларації типів даних ланцюга за замовчуванням:

```
'default_nettype none
module first;
....
```

15.4. Логічні типи

В SystemVerilog розширили можливості типу даних reg, він тепер може отримувати значення в операторі безперервного присвоєння (continuous assignments), точно так само, як і типи даних ланцюга. Також він може бути змінною. Такий модифікований тип отримав назву logic:

```
logic reset;           // 1-бітова змінна з 4 станами
logic [63:0] data;      // 64-бітова змінна
logic [0:7] array [0:255]; // масив з 256 8-бітових змінних
```

Єдине обмеження, введене для типу logic: він не може управлятися декількома драйверами, які використовуються для створення моделей двонаправлених шин. У цьому випадку варто застосовувати тип даних класу ланцюга, наприклад wire. Модуль із лістинга 15.7 представляє приклад використання типу даних logic.

Лістинг 15.7. Використання типу даних logic.

```
module logic_data_type(input logic rst_h);
    parameter CYCLE = 20;
    logic q, q_l, d, clk, rst_l;
    initial begin
        clk <= 0; // Procedural assignment
        forever #(CYCLE/2) clk = ~clk;
    end
    assign rst_l = ~rst_h; // Continuous assignment
    not n1(q_l, q); // q_l is driven by gate
    my_dff d1(q, d, clk, rst_l); // d is driven by module
endmodule
```

15.5. Типи даних із двома станами

Типи введені для підвищення продуктивності процесу моделювання і мінімізації застосування пам'яті в порівнянні з типами, що мають 4 стани. Вони можуть бути використані для опису моделей системного рівня та рівня транзакцій. Визначено такі типи:

1) bit - 1-бітовий цілочисельний тип з 2-ма станами;

- 2) byte - 8-бітовий цілочисельний тип з 2-ма станами, подібний до типу char у Cі;
- 3) shortint - 16-бітовий цілочисельний тип з 2-ма станами, подібний до типу short у Cі;
- 4) int - 32-бітовий цілочисельний тип з 2-ма станами, подібний до типу int у Cі;
- 5) longint - 64-бітовий цілочисельний тип з 2-ма станами, подібний до типу long у Cі.

Найпростіший з типів з 2-ма станами - беззнаковий тип bit. Інші 4 типи: byte, shortint, int і longint є знаковими.

Лістинг 15.8. Типи даних із двома станами.

```
bit b;           // 1 біт з 2-ма станами
bit [31:0] b32;  // 32-бітове беззнакове ціле число
int i;          // 32-бітове знакове ціле число
byte b8;        // 8-бітове знакове ціле число
shortint s;     // 16-бітове знакове ціле число
longint l;      // 64-бітове знакове ціле число
```

Варто бути уважним зі спробою заміни виразу logic [7:0] на найкоротше byte, оскільки перший з них беззнаковий, а другий - знаковий. Таким чином, максимальне позитивне число для об'єкта типу byte буде 127, а не 256.

Потрібно бути обережним при з'єднанні сигналів (особливо виходів) DUT із сигналами, що мають два стани. Рекомендується використовувати оператор \$isunknown, що повертає 1 у випадку, якщо вираз дорівнює X або Z:

```
if ($isunknown(iport)
    $display("@%0d: 4-state value detected on input port", $time, iport);
```

Типи із двома станами можуть бути використані в ситуації, коли немає необхідності в докладній деталізації за допомогою 4-х станів. Наприклад, для опису індексації в циклі for. Навіть у моделях RTL-рівня ця змінна не є присутня у реальній апаратурі:

```
logic [7:0] bus;    // змінна з 4-ма станами
bit [31:0] bus;     // змінна з 2-ма станами
```

SystemVerilog пропонує можливість застосування необов'язкового слова var перед описом типу даних. Наприклад:

```
var logic [7:0] a;   // 8-бітова змінна з 4-ма станами
var bit [31:0] b;    // 32-бітова змінна з 2-ма станами
var int i;           // 32-бітова змінна з 2-ма станами
```

Слово var не впливає на моделювання, а робить код більш читабельним:

```
typedef enum bit {FALSE, TRUE} bool_t;
var bool_t c; // variable of user-defined type
```

За замовчуванням використання ключового слова `var` має на увазі тип `logic`:

```
var [7:0] d; // 4-state 8-bit variable
```

Перетворення значень 4 станів у значення 2 станів наведене в табл. 15.2.

Таблиця 15.2. Перетворення значень

Значення 4 станів	Значення 2 станів
0	0
1	1
Z	0
X	0

Лістинг 15.9 ілюструє використання змінних у моделях SystemVerilog.

Лістинг 15.9. Використання змінних.

```
module compare
    (output logic lt, eq, gt,
     input logic [63:0] a, b);
    always @(a, b)
        if (a < b) lt = 1'b1; // Процедурне присвоєння
        else lt = 1'b0;
    assign gt = (a > b); // Безперервне присвоєння
    comparator u1 (eq, a, b); // Створення копії модуля
endmodule

module comparator (output logic eq,
    input [63:0] a, b);
    always @(a, b)
        eq = (a==b);
endmodule
```

Типи даних `byte`, `shortint`, `int` і `longint` в SystemVerilog є знаковими. Їх можна перетворити в беззнакову форму за допомогою ключового слова `unsigned`:

```
int s_int; // 32-бітова знакова змінна
int unsigned u_int; // 32-бітова беззнакова змінна
```

15.6. Рядки

Строковий тип даних `string` в SystemVerilog підтримує створення рядків змінної довжини. Кожний символ має тип `byte`. Елементи рядка нумеруються починаючи з нуля, рядок довжиною в `N` елементів буде мати індекси `0 : N-1`. Рядки не містять символ кінця рядка. Для них використовується динамічне виділення пам'яті.

Розширено можливості оператора конкатенації для роботи з рядками. Він може бути використаний для об'єднання декількох рядків в одну:

```
string sv = "SystemVerilog";
string s;
s = {sv, " ", "is easy"};
$display ("%s\n", s); // виведення рядка «SystemVerilog is easy»
s = {s, ", enjoy"};
$display ("%s\n", s); // виведення рядка «SystemVerilog is easy, enjoy»
```

Методи й завдання SystemVerilog для роботи з рядками:

- 1) `getc(N)` - повертає значення з позиції `N`;
- 2) `toupper` - повертає копію рядка у верхньому регістрі;
- 3) `tolower` - повертає копію рядка в нижньому регістрі;
- 4) фігурні дужки `{}` використовуються для операції конкатенації;
- 5) завдання `putc(M)` записує в рядок у позицію `M`, яка для рядка в `N` елементів може приймати значення від 0 до `N-1`;
- 6) `len` - повертає довжину рядка;
- 7) функція `substr(start,end)` добуває символи з позицій від `start` до `end`.

Лістинг 15.10 представляє використання операцій над рядками.

Лістинг 15.10. Застосування методів рядків.

```
string s;
initial begin
    = "SystemVerilog";
    display(s.getc(0));           // Виводиться «83 ('S')»
    display(s.toupper());        // Виводиться: «SYSTEMVERILOG»
    = {s, "3.1b"};               // «SystemVerilog3.1b»
    .putc(s.len()-1, "a");        // Заміна символу b на a
    display(s.substr(2, 5));      // Виводиться: «stem»
                                // Створюється тимчасовий рядок
    y_log($psprintf("%s %5d", s, 42));
end
task my_log(string message);
    // Виводиться повідомлення на консоль або в log-файл
    display("@%0d: %s", $time, message);
endtask
```

Замість функції `$sformat` з Verilog-2001 може бути використана функція `$psprintf`. Вона повертає форматований тимчасовий рядок, який безпосередньо може бути переданий в іншу функцію.

15.7. Інші абстрактні типи даних SystemVerilog

Мова SystemVerilog також дозволяє використовувати додаткові абстрактні типи даних:

- 1) `void` - означає відсутність збереження, який використовується для типу даних `union` і функцій, що не повертають значень;
- 2) `shortreal` - коротке речовинне значення, що має довжину 32 біта, на відміну від 64-бітового типу `real`, що не є синтезованим.

В Verilog-95 всі змінні були статичні, в Verilog-2001 з'явилися автоматичні змінні для функцій і завдань (лістинг 15.11). В SystemVerilog можна явно повідомляти змінні як статичні (`static`) (лістинг 15.12) або автоматичні (`automatic`). Такі описи є частиною декларації змінної й можуть бути використані в завданнях, функціях, блоках `begin...end` і `fork...join`. Такі визначення не можна зробити на рівні модуля, всі змінні на рівні модуля є статичними.

Лістинг 15.11. Використання автоматичних змінних у функції.

```
function int count_ones (input [31:0] data);
    automatic logic [31:0] count = 0;
    automatic logic [31:0] temp = data;
    for (int i=0; i<=32; i++) begin
        if (temp[i]) count++;
        temp >>= 1;
    end
    return count;
endfunction
```

Лістинг 15.12. Використання статичної змінної в автоматичній функції.

```
typedef struct packed {...} packet_t;

task automatic check_results
    (input packet_t sent, received);
    output int total_errors;
    static int error_count;
    ...
    if (sent != received) error_count++;
    total_errors = error_count;
endtask
```

В SystemVerilog допускається ініціалізація значень у змінній разом з декларацією як змінних модулів (дозволене й в Verilog), так і змінних функцій і завдань (в Verilog забороняється). Однак такі конструкції не є синтезованими.

Правила використання статичних і автоматичних змінних:

- 1) У блоках `initial` і `always` використовуються статичні змінні, якщо вони не мають ініціалізації при декларації, і автоматичні - у протилежному випадку. Такі змінні будуть повторно ініціалізовані при кожному виклику блоку.
- 2) Якщо функції й завдання викликаються кілька разів, то вони повинні бути автоматичними, якщо тільки немає причини зберігати значення змінної між різними викликами. Наприклад, як для змінної, що зберігає число викликів процедури.
- 3) Якщо завдання або функція моделює фрагмент апаратури, то вони і їх змінні повинні бути оголошені як статичні.

15.8. Перетворення типів

В Verilog-2001 знакове-беззнакове перетворення типів виконувалося за допомогою системної функції `$signed` і `$unsigned`. В SystemVerilog розширені можливості перетворення значень між різними типами даних.

1. Синтаксис перетворення типів в SystemVerilog:

`<type>(<expression>)` - перетворення значень у будь-який тип, включаючи користувацькі типи даних.

Наприклад:

`7+ int'(2.0 * 3.0);` // Значення $(2.0 * 3.0)$ перетвориться в `int` і потім підсумується з 7

2. Синтаксис перетворення розмірів в SystemVerilog:

<size>'(<expression>) - перетворення значення у вектор зазначеної довжини

Наприклад:

```
logic [15:0] a, b, y;
y = a + b**16'(2); // cast literal value 2 to be 16 bits wide
```

3. Синтаксис перетворення знака в SystemVerilog:

<sign>'(<expression>) - casts a value to signed or unsigned.

Наприклад:

```
shortint a, b;
int y;
y = y - signed'({a,b}); // cast concatenation result to a signed value
```

Оскільки операція перетворення типу відноситься до часу компіляції, а реальне присвоювання відбувається в момент моделювання, то перевірка сумісності типів і даних не виконується.

Для здійснення такої перевірки в динамічному режимі може бути використане системне завдання \$cast, що приймає два аргументи: змінна призначення, змінна джерела. Її синтаксис:

```
$cast(dest_var, source_exp);
```

Наприклад:

```
int radius, area;
    always @(posedge clock)
        $cast(area, 3.154 * radius ** 2); // У підсумку результат операції
        // cast речовинне значення перетвориться в ціле
```

Помилки, які можуть виникнути в результаті перетворення типів:

- 1) присвоєння речовинного числа цілому, якщо перше більше розрядної сітки другого;
- 2) перетворення в тип перелічення, коли значення даного типу не існує.

Наприклад, при моделюванні коду може виникнути ситуація:

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;
always_latch begin
    $cast(next_state, state + 1);
end

$cast може бути використана у форматі системної функції
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;
int status;
always_comb begin
    status = $cast(next_state, state + 1);
    if (status == 0) // if cast did not succeed...
        next_state = S1;
end
```

Системне завдання/функція \$cast не підтримується синтезом.

15.9. Розмір виразів

Розмір виразів визначається виходячи з розмірів операндів. У прикладі (лістинг 15.13) виконується операція $1+1$ з використанням різних стилів. Сума А отримана додаванням двох однобітових змінних, з такою точністю операторів $1+1=0$. У сумі В використана 8-бітова точність операндів, тому результат буде також 8-бітовий. У цьому випадку $1+1=2$. Сума С застосовує фіктивну константу, що приводить до використання двобітової точності. В останній сумі D перший операнд являє собою приведення до дворозрядного значення, таким чином, $1+1=2$.

Лістинг 15.13. Залежність розрядності виразу від контексту.

```
bit [7:0] b8;
bit one = 1'b1; // Single bit
$displayb(one + one); // A: 1+1 = 0
b8 = one + one; // B: 1+1 = 2
$displayb(b8);
$displayb(one + one + 2'b0); // C: 1+1 = 2 with constant
$displayb(2'(one) + one); // D: 1+1 = 2 with cast
```

Існує кілька способів уникнути проблеми втрати значень. По-перше, виключити ситуації, коли може втратитися перенос, як у сумі А. Використовувати тимчасові змінні, як, наприклад, b8, для того щоб досягти потрібної розрядності результату. Також для цього може бути використаний додатковий операнд, як $2'b0$. Останній спосіб - в SystemVerilog можна застосувати операцію приведення для отримання операнда потрібної розрядності.

15.10. Константи

В Verilog існує три види констант:

- 1) parameter - константи, значення яких може бути перевизначене в момент елаборації за допомогою оператора defparam або при прямому перевизначенні;
- 2) spsrparam, значення якого може бути перевизначене в момент елаборації значеннями з файлу SDF;
- 3) localparam - константи часу елаборації, які не можуть бути безпосередньо змінені, але їхнє значення може залежати від значення інших констант.

SystemVerilog дозволяє визначити будь-яку змінну як константу за допомогою ключового слова const. Такі константи отримують значення після етапу елаборації та мають такі властивості:

- 1) можуть бути декларовані в динамічних контекстах, таких як автоматичні завдання й функції;
- 2) можуть отримувати значення змінних і ліній, а не тільки константних виразів;
- 3) можуть отримувати значення елементів, оголошених у будь-якому місці ієрархії проекту.

Декларація констант `const` повинна містити декларацію типу даних.

Наприклад:

```
const logic [23:0] C1 = 7;    // 24-бітова константа
const int C2 = 15;          // 32-бітова константа
const real C3 = 3.14;       // Речовинна константа
const C4 = 5;               // Помилка, тип не зазначений
task automatic C;
    const int N = 5;        // N - константа
    ...
endtask
```

У наступному прикладі ініціалізація `colon` виконується при вході в блок `initial`:

```
initial begin
    const byte colon = ":";
    ...
end
```

15.11. Користувальницькі типи даних (User-defined types)

В SystemVerilog користувальницькі типи даних визначаються за допомогою ключового слова `typedef`. Після визначення новий тип може бути використаний для створення змінних.

Новий тип даних може бути заданий локально в модулі, у пакеті або в зоні видимості одиниці компіляції. У наступному фрагменті коду визначений користувальницький тип даних `nibble`, що використовується при декларації змінної в модулі `alu`. Оскільки `nibble` оголошений локально, його зона видимості обмежується тільки модулем `alu`:

```
module alu (...);
    typedef logic [3:0] nibble;
    nibble op, op;          // змінна типу nibble
    nibble [7:0] data;      // 32-бітовий вектор, створений з
                           // 8 елементів типу nibble
    ...
endmodule
```

Можна створити модель ALU, що у процесі компілювання буде зконфігурована для роботи з 8, 16, 24 або 32-бітовими операндами. У мові Verilog це можна реалізувати за допомогою макросу (лістинг 15.14).

Лістинг 15.14. Визначені користувачем типи даних в Verilog.

```
// Old Verilog style
'define OPSIZE 8

'define OPREG reg [' OPSIZE-1:0]
'OPREG op_a, op_b;
```

При цьому реальний тип даних не створюється, виконується тільки заміна одного тексту на інший. SystemVerilog дозволяє створювати новий тип даних (лістинг 15.15).

Лістинг 15.15. Визначені користувачем типи даних в SystemVerilog.

```
// New SystemVerilog style
parameter OPSIZE = 8;
typedef reg [ OPSIZE-1:0] opreg_t;
opreg_t op_a, op_b;
```

Одним з найбільш корисних типів даних може бути беззнакове ціле із двома значеннями. Більшість значень в testbench є позитивними цілими числами, наприклад, довжина поля або кількість отриманих транзакцій. Розмістивши наступне визначення uint в \$root, можна отримати можливість його використання в будь-якому місці при моделюванні:

```
typedef bit [31:0] uint;           // 32-бітовий беззнаковий з 2-ма станами
typedef int unsigned uint; // Еквівалентне попереднє визначення
```

15.12. Спільно використовуване визначення типу typedef

Якщо необхідно, щоб користувальницький тип був видний у декількох модулях, то він визначається в пакеті або в одиниці компіляції.

Лістинг 15.16 ілюструє створення користувальницького типу даних dtype_t у пакеті. При цьому застосовується умовна директива компіляції 'ifdef, що дозволяє визначити тип даних dtype_t за допомогою типу даних з 2-ма станами bit або типу даних з 4-ма станами logic.

Лістинг 15.16. Пряме звертання до типу даних з пакета.

```
package chip_types;
    'ifdef TWO_STATE
        typedef bit dtype_t;
    'else
        typedef logic dtype_t;
    'endif
endpackage
module counter
    (output chip_types::dtype_t [15:0] count,
     input chip_types::dtype_t clock, reset);
always @(posedge clock, negedge reset)
    if (!reset) count <= 0;
    else count <= count + 1;
endmodule
```

Вираз для імпортування пакета можна розмістити в просторі одиниці компіляції \$unit. Це зручно, коли велика кількість портів визначена за допомогою користувальницького типу даних (лістинг 15.17).

Лістинг 15.17. Імпортування визначень пакета в простір одиниці компіляції \$unit.

```
package chip_types;
    'ifdef TWO_STATE
        typedef bit dtype_t;
    'else
        typedef logic dtype_t;
    'endif
endpackage
import chip_types::dtype_t; // import definition into $unit
module counter
    (output dtype_t [15:0] count,
     input dtype_t clock, reset);

    always @(posedge clock, negedge reset)
        if (!reset) count <= 0;
        else count <= count + 1;

endmodule
```

Якщо пакет містить багато визначень користувацьких типів, можна застосовувати спеціальний груповий символ:

```
import chip_types::*; // Груповий імпорт
```

15.13. Типи перелічення

Перелічення - це тип даних, що являє собою обмежену безліч спеціальних імен, таких як команди або стани автомата. Використання таких типів спрощує читання коду.

Найпростіша декларація типу перелічення містить список імен і одну або декілька змінних. Таким чином, створюється тип даних без імені:

```
enum {RED, BLUE, GREEN} color;
```

За допомогою typedef може бути створений іменний тип перелічення (лістинг 15.18).

Лістинг 15.18. Типи перелічення.

```
// Створення типу даних для значень 0, 1, 2
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate; // declare typed variables
initial begin
    case (pstate)
        IDLE: nstate = INIT; // data assignment
        INIT: nstate = DECODE;
        default: nstate = IDLE;
    endcase
    $display("Next state is %0s", nstate.name); // Use name function
end
```

15.13.1. Визначення значень типів перелічення.

Реальні значення відповідають послідовності невід'ємних цілих чисел, починаючи з 0. Також елементам типу перелічення можуть бути присвоєні будь-

які довільні значення. У наступному прикладі призначені: 0 - для INIT, 2 - для DECODE і 3 - для IDLE:

```
typedef enum {INIT, DECODE=2, IDLE} fsmtype_e;
```

Константи перелічення, такі як INIT з попереднього прикладу, використовують ті ж правила видимості, що й змінні. Відповідно, якщо використовуються ті самі імена в декількох типах перелічення (наприклад, INIT у різних керуючих автоматах), вони повинні декларуватися в різних зонах видимості, таких як модулі, блоки програми, підпрограми або класи. Мітки типів перелічення зберігаються як int, якщо їхній тип не заданий іншим способом. Потрібно бути обережним при присвоєнні значень константам типів перелічення, тому що за замовчуванням int отримує значення 0. У прикладі (лістинг 15.19, а) position ініціалізується в 0, що не дозволено для змінної ordinal_e. Така поведінка не є помилкою середовища - вона відповідає специфікації мови. Таким чином, варто завжди описувати константи типів перелічення зі значенням 0 (лістинг 15.19, б), щоб мати можливість виявлення цієї помилки.

Лістинг 15.19. Приклади типів перелічення.

а) // Неправильний опис значень типів перелічення

```
typedef enum {FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

б) //Правильний опис значень типів перелічення

```
typedef enum {ERR_O=0, FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

Лістинг 15.20 представляє приклад використання типу для визначення станів автомата. Спеціальні процедурні блоки always_ff і always_comb, використані в прикладі, будуть розглянуті далі.

Лістинг 15.20. Використання типу перелічення для опису станів керуючого автомата.

```
package chip_types;
    typedef enum {FETCH, WRITE, ADD, SUB, MULT, DIV, SHIFT, NOP }
    instr_t;
endpackage

import chip_types::*; // import package definitions into $unit
module controller (output logic read, write,
    input instr_t instruction,
    input wire clock, reset);
    enum {WAITE, LOAD, STORE} State, NextState;
    always_ff @(posedge clock, negedge reset)
        if (!reset) State <= WAITE;
        else State <= NextState;
    always_comb begin
        case (State)
            WAITE: NextState = LOAD;
            LOAD: NextState = STORE;
            STORE: NextState = WAITE;
        endcase
    end
```

```

end
always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)
        read = 1;
    else if (State == STORE && instruction == WRITE)
        write = 1;
end
endmodule

```

15.13.2. Послідовності міток у типі перелічення.

На додаток до опису множини й унікальних міток SystemVerilog пропонує два спрощених способи створення діапазону міток у типі перелічення (табл. 15.3).

Таблиця 15.3. Опис послідовності міток типу перелічення

state	Створення однієї мітки з іменем state
state[N]	Створення послідовності міток з іменем state0, state1, ... , stateN-1
state[N:M]	Створення послідовності міток, які починаються зі stateN та закінчуються stateM. Якщо N менше M, послідовність буде збільшуватися від N до M. Якщо N більше M, послідовність буде зменшуватися від N до M

Приклад використання можливостей SystemVerilog для опису міток типу перелічення:

```
enum {RESET, S[5], W[6:9]} state;
```

У цьому випадку змінна state може приймати одне з таких значень: RESET, S0, S1, S2, S3, S4, W6, W7, W8, W9.

15.13.3. Завдання базового типу для міток.

Типи перелічення є змінними або ланцюгами з безліччю значень міток. За замовчуванням базовим типом для перелічення є int (32-бітовий з 2-ма станами). Для більш детального подання апаратури дозволяється явно задавати базовий тип для перелічень. Наприклад:

```
// тип перелічення з 1-1- розрядним кодуванням міток
```

```
// типом з 2-ма станами
```

```
enum bit {TRUE, FALSE} Boolean;
```

```
// тип перелічення з 2-розрядним кодуванням міток
```

```
// типом з 4-ма станами
```

```
enum logic [1:0] {WAITE, LOAD, READY} state;
```

Якщо мітки типу перелічення мають явне присвоєння значень, їхній розмір повинен збігатися з розміром базового типу:

```
enum logic [2:0] {WAITE = 3'b001, LOAD = 3'b010, READY = 3'b100} state;
```

Буде помилкою привласнити мітці значення, що відрізняється за розміром від базового типу, як це зроблено в наступному прикладі:

```
enum {WAITE = 3'b001,      // ПОМИЛКА!  
      LOAD = 3'b010, READY = 3'b100} state;
```

або ж описати більше міток, ніж може бути представлено базовим типом:

```
enum logic {A=1'b0, B, C} list5; // ПОМИЛКА: занадто багато міток  
      // для однорозрядного коду
```

Якщо базовий тип має 4 значення, то можна використовувати для присвоєння значення *X* або *Z* для міток перелічення:

```
enum logic {ON=1'b1, OFF=1'b0} out;
```

При кодуванні мітки із застосуванням значень *X* або *Z* наступна мітка повинна мати явно заданий код:

```
enum logic [1:0] {WAITE, ERR=2'bxx, LOAD, READY} state;  
      // ПОМИЛКА: Не визначене значення для LOAD
```

Тип перелічення може бути визначений за допомогою конструкції `typedef` (типізований тип перелічення - `typed enumerated type`) - це дає можливість декларувати за допомогою такого типу кілька змінних. Інакше, тип перелічення є анонімним (`anonymous enumerated type`):

```
typedef enum {WAITE, LOAD, READY} states_t;  
states_t state, next_state;
```

15.13.4. Підпрограми й функції для типів перелічення.

SystemVerilog пропонує кілька методів для роботи з елементами типів перелічення:

- 1) `first` повертає перший елемент;
- 2) `last` повертає останній елемент;
- 3) `next` повертає наступний елемент;
- 4) `next(N)` повертає *N* наступних елементів;
- 5) `prev` повертає попередній елемент;
- 6) `prev(N)` повертає *N* попередніх елементів.

Функції `next` і `prev` йдуть по колу, коли вони досягають кінця послідовності елементів перелічення.

Немає простого способу, щоб виконати перебір всіх елементів типу перелічення в циклі `for`. Перший елемент може бути отриманий за допомогою функції `first`, а наступні - за допомогою функції `next`. Проблема виникає у визначенні останнього елемента послідовності. Якщо використовувати `current!=current.last`, то цикл завершиться, не досягнувши останнього елемента. Використання виразу:

```
current<=current.last,
```

приведе до створення нескінченного циклу, тому що функція `next` ніколи не поверне значення, що перевищує останнє в послідовності.

Для перебору значень може бути використаний цикл `do...while`, як це показано в прикладі з лістингу 15.21.

Лістинг 15.21. Перебір всіх елементів типу перелічення.

```
enum {RED, BLUE, GREEN} color;
    color = color.first;
do begin
    $display("Color = %0d/%0s", color, color.name);
    color = color.next;
end
while (color != color.first); // Done at wrap-around
```

Оскільки за замовчуванням мітки типу перелічення представлені типом `int` (2 значення), то значення змінних типів перелічення можуть бути присвоєні змінним типів `integer` або `int`. Однак не дозволяється в SystemVerilog зберігати 4-значкові `integer` в `enum` без виконання перетворення типу. SystemVerilog вимагає явного приведення типу, щоб виключити випадкове присвоєння значення, що виходить за межі типу (лістинг 15.22). У прикладі функція `$cast` намагається виконати присвоєння правого значення в ліву змінну. Якщо присвоєння виконане успішно, то функція `$cast` повертає 1, інакше 0.

Лістинг 15.22. Призначення між типами `integer` і перелічення.

```
typedef enum {RED, BLUE, GREEN} COLOR_E;
COLOR_E color, c2;
integer c;
initial begin
    c = color;          // Перетворить із типу enum в integer
    c++;               // Інкремент integer
    if (!$cast(color, c)) // Перетворення integer назад в enum
        $display("Cast failed for c=%0d", c);
    $display("Color is %0d / %0s", color, color.name);
    c2 = COLOR_E'(c);   // Без виконання перевірки типу
end
```

Можна для перетворення типів використовувати вираз `type'(val)`, однак при цьому не виконується ніякої перевірки, таким чином, результат може виходити за припустимі межі. Даний стиль не рекомендується використовувати.

Лістинг 15.23 ілюструє використання керуючого автомата з послідовністю станів, які можуть бути перебрані за допомогою описаних вище функцій. Приклад представляє найпростіший прихований лічильник, де:

- 1) вихід `in_sync` спочатку дорівнює 0; коли лічильник досягає значення 8, `in_sync` встановлюється в 1;
- 2) якщо входи прапорів `compare` і `synced` мають значення `false`, то лічильник зберігає поточне значення;
- 3) якщо прапори `compare` і `synced` мають значення `true`, то лічильник збільшується на 1, але не може бути більше 15;
- 4) якщо прапор `compare` має значення `true`, але прапор `synced` дорівнює `false`, лічильник зменшується на 2, але не може бути менше 0.

Лістинг 15.23. Використання спеціальних методів для переміщення за списком типу перелічення.

```

module confidence_counter(input logic synced, compare, reset, clock,
    output logic in_sync);
    enum {cnt[0:15]} State, Next;
    always_ff @(posedge clock, negedge reset)
        if (!reset) State <= cnt0;
        else State <= Next;
    always_comb begin
        Next = State; // default NextState value
        case (State)
            cnt0 : if (compare && synced) Next = State.next;
            cnt1 : begin
                if (compare && synced) Next = State.next;
                if (compare && !synced) Next = State.first;
            end
            cnt15: if (compare && !synced) Next = State.prev(2);
            default begin
                if (compare && synced) Next = State.next;
                if (compare && !synced)
                    Next = State.prev(2);
            end
        endcase
    end
    always_ff @(posedge clock, negedge reset)
        if (!reset) in_sync <= 0;
        else begin
            if (State == cnt8) in_sync <= 1;
            if (State == cnt0) in_sync <= 0;
        end
endmodule

```

На друк може виводитися мітка типу перелічення або його значення (лістинг 15.24).

Лістинг 15.24. Друк значень та імен міток типів перелічення.

```

module FSM (input logic clock, reset,
    output logic [3:0] control);
    enum logic [2:0] {WAITE=3'b001,
        LOAD =3'b010, READY=3'b010} State, Next;
    always @(posedge clock, negedge reset)
        if (!reset) State <= WAITE;

        else State <= Next;
    always_comb begin
        $display("\nCurrent state is %s (%b)", State.name, State);
        case (State)
            WAITE: Next = LOAD;
            LOAD: Next = READY;
            READY: Next = WAITE;
        endcase
    end
endmodule

```



```

        $display("Next state will be %s (%b)", Next.name, Next);
    end
    assign control = State;
endmodule

```

15.14. Контрольні запитання

1. Які з перелічених типів даних відносяться до типів даних із двома станами?

bit, byte, shortint, int, longint wire, reg, integer, time, real, shortint

2. Що означає вираз: `int'(2.0 * 3.0);`?

3. Що означає вираз: `signed'({a,b});`?

4. Для чого призначена функція `$cast`?

5. Який номер має елемент RED у типі перелічення, визначеного так?

```
enum {RED, BLUE, GREEN} color;
```

6. Яка помилка міститься в такому виразі?

```
typedef enum {FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

7. Які мітки містить такий тип перелічення?

```
enum {RESET, S[5], W[6:9]} state;
```

8. Який тип даних мають мітки наступного типу перелічення?

```
enum logic [1:0] {WAITE, LOAD, READY} state;
```

9. Яка помилка міститься в такому виразі?

a) `enum {WAITE = 3'b001, LOAD = 3'b010, READY = 3'b100} state;`

б) `enum logic {A=1'b0, B, C} list5;`

в) `enum logic [1:0] {WAITE, ERR=2'bxx, LOAD, READY} state;`

16. СКЛАДНІ ТИПИ ДАНИХ SYSTEM VERILOG: ARRAYS, STRUCTURES I UNIONS

16.1. Структури (Structures)

16.1.1. Створення структур, що визначаються користувачем.

В SystemVerilog надана можливість створення структур за допомогою struct, подібних використуванню у мові Сі. Структури зручні для об'єднання спорідненої інформації:

```
struct {
    int a, b; // 32-bit variables
    opcode_t opcode; // user-defined type
    logic [23:0] address; // 24-bit variable
    bit error; // 1-bit 2-state var.
} Instruction_Word;
```

Структури вважають виродженими класами і в деяких випадках рекомендують віддавати перевагу останнім.

Кілька змінних можна об'єднати в структуру. Наприклад, структура pixel має три беззнакових байти для опису червоної (r), зеленої (g) і блакитної (b) складових кольору:

```
struct {bit [7:0] r, g, b;} pixel;
```

Даний приклад створює тільки один елемент заданого типу. Для того щоб мати можливість використовувати даний тип для опису портів і в підпрограмах, його варто описати у вигляді нового типу даних:

```
typedef struct {bit [7:0] r, g, b;} pixel_s;
pixel_s my_pixel;
```

Найбільш корисний оператор typedef для створення типів, об'єднань (unions), типів перелічення та віртуальних інтерфейсів.

Звертання до елементів структури:

```
<structure_name>.<variable_name>
```

Наприклад:

```
Instruction_Word.address = 32'h000001E;
```

Структури можуть бути оголошені для змінних і ланцюгів. Для оголошення структури як змінної в загальному виді може бути використане ключове слово var, а для ланцюгів - wire:

```
var struct { // Змінна - структура
    logic [31:0] a, b;
    logic [ 7:0] opcode;
    logic [23:0] address;
} Instruction_Word_var;
```

```
wire struct { // Структура класу ланцюга ( net)
    logic [31:0] a, b;
    logic [ 7:0] opcode;
    logic [23:0] address;
} Instruction_Word_net;
```

Елементи структури можуть бути ініціалізовані при створенні змінної. Для цього використовується конструкція '{and}' (лістинг 16.1, а).

Можна присвоювати значення окремим полям структури (лістинг 16.1, б) або всій структурі в цілому за допомогою оператора '{and}' (лістинг 16.1, в).

Лістинг 16.1. Присвоєння значень елементам структури.

а) // У момент декларації

```
typedef struct {
    logic [31:0] a, b;
    logic [ 7:0] opcode;
    logic [23:0] address;
} instruction_word_t;
instruction_word_t IW = '{100, 3, 8'hFF, 0};
```

б) // Присвоєння значень окремим елементам структури

```
typedef struct {
    logic [31:0] a, b;
    logic [ 7:0] opcode;
    logic [23:0] address;
} instr_t;
instr_t IW;
always @(posedge clock, negedge reset)
    if (!reset) begin
        IW.a = 100; // Звертання до поля а структури IW
        IW.b = 5;
        IW.opcode = 8'hFF;
        IW.address = 0;
    end
    else begin
        ...
    end
```

в) // Присвоєння значень всій структурі

```
always @(posedge clock, negedge reset)
    if (!reset) IW = '{100, 5, 8'hFF, 0};
    else begin
        ...
    end
```

При цьому присвоєння виконується по порядку проходження елементів. Можна виконати ініціалізацію значень із вказівкою їхнього імені, як це зроблено в такому прикладі:

```
IW = '{address:0, opcode:8'hFF, a:100, b:5};
```

Однак не допускається змішувати обидва стилі опису:

```
IW = '{address:0, 8'hFF, 100, 5}; // ERROR
```

Можна ініціалізувати всі елементи структури за допомогою оператора за замовчуванням:

```
IW = '{default:0}; // Установка всіх елементів IW в 0
```

Також допускається виконувати ініціалізацію даних певного типу:

```
typedef struct {
    real r0, r1;
    int i0, i1;
    logic [ 7:0] opcode;
    logic [23:0] address;
} instruction_word_t;
instruction_word_t IW;
always @(posedge clock, negedge reset)
    if (!reset)
        IW = '{real:1.0, default:0};
        // Всі речовинні змінні отримують значення 1.0,
        // інші елементи встановлюються в 0
    else begin
        ...
    end
```

Існує пріоритет присвоєння значень даним, при цьому оператор default має найнижчий пріоритет і перевизначається будь-яким типом даних. У наступному прикладі поле r0 отримує значення 1.0, r1 - 3.1415, а всі інші поля - 0:

```
typedef struct {
    real r0, r1;
    int i0, i1;
    logic [ 7:0] opcode;
    logic [23:0] address;
} instruction_word_t;
instruction_word_t IW;
IW = '{real:1.0, default:0, r1:3.1415};
```

16.1.2. Запаковані структури.

SystemVerilog надає можливість керувати способом збереження даних у пам'яті, використовуючи запаковані структури. Вони зберігаються як безперервна безліч бітів без додаткового невикористовуваного простору. Наведена в попередніх прикладах структура struct містить три значення і, отже, займає три довгих слова, навіть якщо реально досить трьох байт. Щоб заощадити пам'ять, можна перетворити структуру в запаковану:

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Запаковані структури використовуються, коли базові біти представляють чисельні структури або коли виконується спроба економії пам'яті.

Наприклад, можна запакувати разом кілька бітових полів для формування єдиного регістра. Можна запакувати разом код команди opcode і поля операндів, отримавши таким чином єдину інструкцію для процесора.

Оскільки запаковані структури зберігаються у вигляді вектора, то математичні й логічні операції можуть бути реалізовані для запакованих структур, як і для векторів:

```
typedef struct packed {
    logic valid;
    logic [ 7:0] tag;
    logic [31:0] data;
} data_word_t;
data_word_t packet_in, packet_out;
always @(posedge clock)
    packet_out <= packet_in << 2;
```

Запакована структура може бути оголошена як беззнакова (unsigned) або знакова (signed), що визначає спосіб її обробки при реалізації арифметичних операторів:

```
typedef struct packed signed {
    logic valid;
    logic [ 7:0] tag;
    logic signed [31:0] data;
} data_word_t;
data_word_t A, B;
always @(posedge clock)
    if ( A < B ) // Знакове порівняння
    ...
```

Обидва види структур (запаковані й ні) є синтезованими.

16.2. Об'єднання (Union)

В апаратурі ті самі дані, об'єднані в регістри, можуть мати різні значення й формати для пристрою, що їх обробляє. Наприклад, через той самий регістр можуть передаватися як команди, так і дані. У прикладі (лістинг 16.2) ціле і речовинне *f* числа зберігаються в тому самому місці.

Лістинг 16.2. Приклади об'єднань.

```
// Використання typedef для створення об'єднань
typedef union { int i; real f; } num_u;
num_u un;
un.f = 0.0; // установка значень у форматі із плаваючою точкою
// Змінна, що має тип об'єднання
union {
    int i;
    int unsigned u;
} data;
...
data.i = -5;
$display("data is %d", data.i);
data.u = -5;
$display("now data is %d", data.u);
typedef union { // typed union
    int i;
```

```

        int unsigned u;
    } data_t;
    data_t a, b;           // два змінні типи data_t

```

Незапаковані об'єднання не синтезуються. Можуть містити будь-які типи змінних, включаючи `real`, неупаковані структури й масиви.

16.2.1. Упаковані об'єднання.

Об'єднання можуть бути оголошені як упаковані, подібно до структур. Тоді кількість бітів, які займає кожний елемент, буде однаковим, що дозволяє запаковані пакети зробити синтезованими.

Приклад, представлений лістингом 16.3, визначає запаковане об'єднання, у якому дані можуть бути представлені двома способами: як пакет даних (за допомогою структури) або як масив байтів (рис. 16.1).

Лістинг 16.3. Приклад запакованих об'єднань.

```

typedef struct packed {
    logic [15:0] source_address;
    logic [15:0] destination_address;
    logic [23:0] data;
    logic [ 7:0] opcode;
} data_packet_t;
union packed {
    data_packet_t packet;    // Запакована структура
    logic [7:0][7:0] bytes;  // Запакований масив
} dreg;

```

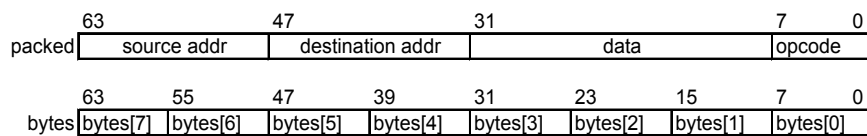


Рис. 16.1. Запаковане об'єднання із двома формами подання тих самих даних

16.2.2. Приклад використання структур і об'єднань.

Лістинг 16.4 містить модель найпростішого арифметичного логічного пристрою, що може оперувати або знаковими, або беззнаковими даними. Структура, що представляє команду, містить: код команди ALU, два операнда й прапори для індикації знакових або беззнакових даних. Пристрій може оперувати одним з видів даних, але не обома відразу.

Лістинг 16.4. Використання структури й об'єднання.

```

package definitions;
    typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;
    typedef enum {UNSIGNED, SIGNED} operand_type_t;
    typedef union packed {
        logic [31:0] u_data;
        logic signed [31:0] s_data;
    }

```

```

    } data_t;
    typedef struct packed {
        opcode_t opc;
        operand_type_t op_type;
        data_t op_a;
        data_t op_b;
    } instr_t;
endpackage
import definitions::*; // імпорт пакета в простір $unit
module alu
    (input instr_t IW,
     output data_t alu_out);
always @(IW) begin
    if (IW.op_type == SIGNED) begin
        case (IW.opc)
            ADD : alu_out.s_data = IW.op_a.s_data + IW.op_b.s_data;
            SUB : alu_out.s_data = IW.op_a.s_data - IW.op_b.s_data;
            MULT: alu_out.s_data = IW.op_a.s_data * IW.op_b.s_data;
            DIV : alu_out.s_data = IW.op_a.s_data / IW.op_b.s_data;
            SL : alu_out.s_data = IW.op_a.s_data <<< 2;
            SR : alu_out.s_data = IW.op_a.s_data >>> 2;
        endcase
    end
    else begin
        case (IW.opc)
            ADD : alu_out.u_data = IW.op_a.u_data + IW.op_b.u_data;
            SUB : alu_out.u_data = IW.op_a.u_data - IW.op_b.u_data;
            MULT: alu_out.u_data = IW.op_a.u_data * IW.op_b.u_data;
            DIV : alu_out.u_data = IW.op_a.u_data / IW.op_b.u_data;
            SL : alu_out.u_data = IW.op_a.u_data << 2;
            SR : alu_out.u_data = IW.op_a.u_data >> 2;
        endcase
    end
end
endmodule

```

16.3. Масиви фіксованого розміру

Базовий синтаксис масивів

<data_type> <vector_size> <array_name> <array_dimensions>

Наприклад:

```
reg [15:0] RAM [0:4095]; // Масив пам'яті
```

В Verilog-2001 дозволяється створювати масиви змінних і ланцюгів:

```
// Одновимірний незапакований масив з 1024
```

```
// 1-бітових елементів класу net
```

```
wire n [0:1023];
```

```
// Одновимірний незапакований масив з 256f
```

```
// 8-бітових змінних
reg [7:0] LUT [0:255];
```

```
// Одновимірний з масив з 1024
// речових змінних
real r [0:1023];
```

```
// 3-вимірний незапакований масив
// з 32-бітових цілочисельних змінних
integer i [7:0][3:0][7:0];
```

16.3.1. Декларація та ініціалізація.

Дозволяється виконувати декларацію масиву, вказуючи тільки його розмір, подібно до того, як це робиться в Сі. Наступні два вирази є цілком рівнозначними:

```
int lo_hi[0:15]; // 16 ints [0]..[15]
int c_style[16]; // 16 ints [0]..[15]
```

Дозволяється створювати багатовимірні масиви фіксованої довжини, описуючи розмірність масиву після імені змінної. Це незапаковані масиви (unpacked array). У прикладі створюються два двовимірних масиви цілих чисел розміром 8x4. Останній елемент масиву array2 встановлюється в 1. Багатовимірні масиви були введені в Verilog-2001, нововведенням є можливість компактної декларації:

```
int array2 [0:7][0:3];           // Докладна декларація
int array3 [8][4];               // Компактна декларація
array2[7][3] = 1;                // Установка значення останнього
                                // елемента масиву
bit [63:0] d_array [1:128];      // Масив векторів
shortreal cosines [0:89];        // Масив речових чисел of floats
typedef enum {Mo, Tu, We, Th, Fr, Sa, Su} Week;
Week Year [1:52];                 // Масив елементів типу Week
```

Кожний елемент в SystemVerilog зберігається у вигляді довгого слова (32-bit). Таким чином, byte, shortint і int зберігаються в одному довгому слові longword, у той час як longint - у двох. (Симулятор часто зберігає чотиризначні типи, такі як logic і integer у двох або більше словах).

Наприклад, елементи з масиву

```
bit [7:0] b_unpacked[3];
```

будуть займати три довгих слова (рис. 16.2).

Ініціалізація масивів може бути виконана за допомогою апострофа та фігурних дужок (лістинг 16.5).

b_array[0]				7	6	5	4	3	2	1	0				
b_array[1]	Unused space							7	6	5	4	3	2	1	0
b_array[2]				7	6	5	4	3	2	1	0				

Рис. 16.2. Розміщення з масиву в пам'яті

Лістинг 16.5. Ініціалізація масиву.

```
int ascend[4] = '{0,1,2,3};           // Ініціалізація 4 елементів
int descend[5];
int md[2][3] = '{0,1,2}, {3,4,5}';
descend = '{4,3,2,1,0};              // Установка 5 елементів
descend[0:2] = '{5,6,7};              // Установка 3 елементів
ascend = '{4(8)};                    // Чотири повторення значення 8
```

На відміну від Verilog в SystemVerilog дозволяється звертатися до всього незапакованого масиву або діапазону з масиву:

```
int a1 [7:0][1023:0];                // Незапакований масив
int a2 [1:8][1:1024];                // Незапакований масив
a2 = a1;                             // Копіювання цілого масиву
a2[3] = a1[0];                       // Копіювання сектора масиву
```

16.3.2. Базові операції масивів: for і foreach.

У прикладі (лістинг 16.6) змінна *i* є локальною для циклу for. SystemVerilog функція \$size повертає розмір масиву. В операторі foreach вказується ім'я масиву й у квадратних дужках індекс. SystemVerilog автоматично перебирає всі елементи масиву.

Лістинг 16.6. Використання масивів із циклами for і foreach.

```
initial begin
    bit [31:0] src[5], dst[5];
    for (int i=0; i<$size(src); i++)
        src[i] = i;
    foreach (dst[j])
        dst[j] = src[j] * 2; // dst doubles src values
end
```

Для багатовимірного масиву синтаксис звертання до елемента: [i,j]. Лістинг 16.7 представляє приклад роботи з елементами багатовимірного масиву.

Лістинг 16.7. Ініціалізація та перебір елементів багатовимірного масиву.

```
initial begin
    $display("Initial value:");
    foreach (md[i,j]) // Yes, this is the right syntax
        $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
    $display("New value:");
    md = '{9, 8, 7}, 3{5}'; // Replicate last 3 values
    foreach (md[i,j]) // Yes, this is the right syntax
        $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

// Результат моделювання

Initial value:

md[0][0] = 0

md[0][1] = 1

md[0][2] = 2

md[1][0] = 3

md[1][1] = 4

md[1][2] = 5

New value:

md[0][0] = 9

md[0][1] = 8

md[0][2] = 7

md[1][0] = 5

md[1][1] = 5

md[1][2] = 5

16.3.3. Базові операції масивів: copy і compare.

Дозволяється виконувати операції порівняння та копіювання масивів без використання циклів. Операція копіювання працює із цілими масивами (лістинг 16.8). Порівняння обмежене результатами рівно та нерівно.

Лістинг 16.8. Операції копіювання та порівняння масивів.

```
initial begin
    bit [31:0] src[5] = '{0,1,2,3,4},
    dst[5] = '{5,4,3,2,1};
    // Порівняння двох масивів
    if (src==dst)
        $display("src == dst");
    else
        $display("src != dst");
    // Копіювання елементів масиву src в dst
    dst = src;
    // Порівняння одного елемента
    src[0] = 5;
    // Порівняння всіх елементів масиву та виведення повідомлення про це
    $display("src %s dst", (src == dst) ? "==" : "!=");
    // Порівнює останні елементи масиву 1-4
    $display("src[1:4] %s dst[1:4]", (src[1:4] == dst[1:4]) ? "==" : "!=");
end
```

Не можна виконувати об'єднання арифметичних операцій, таких як складання масивів. Замість цього можна використовувати цикли. Для логічних операцій, таких як хог, використовуються цикли або запаковані масиви.

Verilog-1995 не дозволяє використовувати разом слово і піддіапазони. В Verilog-2001 цього обмеження немає для масивів фіксованого розміру. У прикладі з лістингу 16.9 роздруковуються перший елемент масиву (1012), молодший біт (1) і наступні два старших біти (binary 10).

Лістинг 16.9. Використання слова та біта масиву.

```
initial begin
```

```

bit [31:0] src[5] = `5{5};
$displayb(src[0],, // 'b101 или 'd5
src[0][0],,        // 'b1
src[0][2:1]);      // 'b10
end

```

16.3.4. Запаковані масиви (Packed arrays).

Запаковані масиви - безперервні послідовності бітів, які не мають невикористовуваних просторів на відміну від незапованих масивів даних.

Запаковані масиви - це зручний спосіб розподілу векторів на поля, доступ до яких стає можливим через індекси масиву. Якщо запакований масив оголошується як signed, то він розглядається як один вектор з лівим старшим бітом у вигляді знака.

Розмірність запакованих масивів задається перед іменем. Розмірність описується у форматі [lo:hi]. У прикладі (лістинг 16.10, рис. 16.3) змінна bytes являє собою запакований масив розміром в одне довге слово.

Лістинг 16.10. Декларація й використання запакованого масиву.

```

bit [3:0] [7:0] bytes; // 4 bytes packed into 32-bits
bytes = 32'hdead_beef;
$displayh(bytes,, // Show all 32-bits
bytes[3], // most significant byte "de"
bytes[3][7]); // most significant bit "1"

```

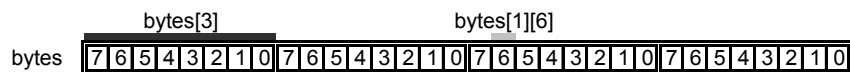


Рис. 16.3. Структура запакованого масиву

Запаковані масиви можуть бути створені з бітових типів (logic, bit або reg), інших запакованих масивів, структур і об'єднань, а також з ланцюгів (wire, uwire, wand, tri, triand, trior, tri0, tri1 або trireg):

```

typedef struct packed {
    logic [ 7:0] crc;
    logic [63:0] data;
} data_word;
data_word [7:0] darray; // 1-D запакований масив
                      // з запакованих структур

```

В запакованих масивах можна звертатися: до всього масиву, до окремих елементів або піддіапазонів:

```

logic [3:0][7:0] data; // 2-D запакований масив
wire [31:0] out = data; // Цілий масив
wire sign = data[3][7]; // Виділення біта
wire [3:0] nib = data [0][3:0]; // Виділення діапазону
byte high_byte;
assign high_byte = data[3]; // 8-бітовий діапазон
logic [15:0] word;
assign word = data[1:0]; // 2 діапазон

```

Можна змішувати запаковані та незапаковані виміри, наприклад, для подання пам'яті, доступної по бітах, байтах і довгим байтах. У прикладі (лістинг 16.11, рис. 16.4) `barray` - незапакований масив із трьох запакованих елементів.

Лістинг 16.11. Декларування змішаних масивів.

```
bit [3:0] [7:0] barray [3]; // Packed: 3x 32-bit
barray[0] = 32'h0123_4567;
barray[0][3] = 8'h01;
barray[0][1][6] = 1'b1;
```

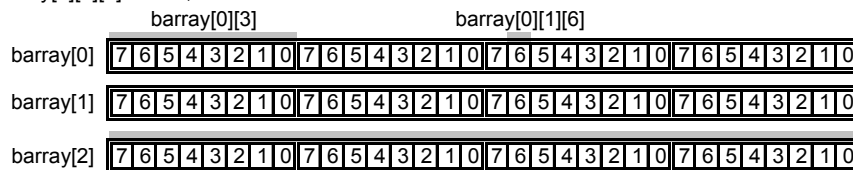


Рис. 16.4. Структура 3-вимірного запакованого масиву

При ініціалізації першим отримує значення довге слово `barray[0]`, наступним ініціалізується байт даних `barray[0][3]`, останнім - біт `barray[0][1][6]`. Слід звернути увагу, що оскільки один вимір описується після імені, `barray[3]`, то він є незапакованим, тому необхідно використовувати як мінімум один індекс.

Рис. 16.5 ілюструє порядок запису індексів при звертанні до елемента масиву зі змішаною індексацією.

Selection order for mixed packed/unpacked multi-dimensional array

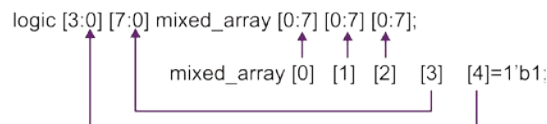


Рис. 16.5. Порядок запису індексів у багатовимірному масиві

16.3.5. Призначення запакованих масивів.

Якщо необхідно виконувати перетворення в скалярні значення і навпаки, використовуються запаковані масиви. Наприклад, при звертанні до пам'яті по-байтно або довгими словами. Із цією метою може бути використаний описаний у прикладі (див. лістинг 16.11) масив `barray`. Запакованими можуть бути тільки масиви фіксованого розміру, тому, якщо необхідно використовувати зміни усередині масиву, варто застосовувати запаковані масиви.

Наприклад, операція `@` може бути використана зі скалярними значеннями і запакованими масивами. У розглянутих вище прикладах буде легальним використання оператора `@` зі змінною `lw` і масивом `barray[0]`, але ніяк не із цілим масивом `barray`. Однак припустимий вираз: `@(barray[0] or barray[1] or barray[2])`.

Ініціалізація запакованих масивів:

```
logic [3:0][7:0] a = 32'h0; // Присвоєння значень вектору
```

```
logic [3:0][7:0] b = {16'hz,16'h0};      // Оператор конкатенації
logic [3:0][7:0] c = {16{2'b01}};        // Оператор дублювання
```

Ініціалізація з масивів виконується за допомогою конструкції '{and}' для кожного виміру масиву:

```
int d [0:1][0:3] = '{ {7,3,0,5}, {2,0,1,6} };
// d[0][0] = 7
// d[0][1] = 3
// d[0][2] = 0
// d[0][3] = 5
// d[1][0] = 2
// d[1][1] = 0
// d[1][2] = 1
// d[1][3] = 6
int e [0:1][0:3] = '{ 2{7,3,0,5} };
// e[0][0] = 7
// e[0][1] = 3
// e[0][2] = 0
// e[0][3] = 5
// e[1][0] = 7
// e[1][1] = 3
// e[1][2] = 0
// e[1][3] = 5
```

Використання значення за замовчуванням:

```
int a1 [0:7][0:1023] = '{default:8'h55};
```

Присвоєння значень незапакованим масивам в SystemVerilog:

```
byte a [0:3][0:3];
a[1][0] = 8'h5; // Присвоєння значень одному елементу
a = '{0,1,2,3},
      '{4,5,6,7},
      '{7,6,5,4},
      '{3,2,1,0}};
```

// Присвоєння списку значень цілому масиву

```
a[3] = '{h, 'h, 'h, 'h};
```

// Присвоєння списку значень діапазону масиву

або з використанням значення за замовчуванням:

```
always @(posedge clock, negedge reset)
if (!reset) begin
    a = '{default:0}; // Ініціалізація цілого масиву
    a[0] = '{default:4}; // Ініціалізація діапазону масиву
end
else begin
    //...
end
```

В запакованих масивах значення можна привласнювати всьому масиву, окремим елементам і піддіапазонам.

```

logic [1:0][1:0][7:0] a;           // 3-вимірний запакований масив
a[1][1][0] = 1'b0;                // Присвоєння значення одному біту
a = 32'h1A3C5E7; // Присвоєння значень цілому масиву
a[1][0][3:0] = 4'h;               // Присвоєння значень діапазону масиву
a[0] = 16'hFACE;                  // Присвоєння значень сектору масиву
a = {16'bz, 16'b0};               // Присвоєння конкатенації елементів i

```

16.3.6. Приклад використання масивів.

Наступний приклад (лістинг 16.12) ілюструє використання запакованого масиву для регістра з 32 інструкцій. Кожна інструкція являє собою дані, описані запакованою структурою. Операнди в інструкції можуть бути знаковими або беззнаковими, для чого використовуються об'єднання двох типів. Входами регістра команд є вибір операнда (separate operands), код команди (opcode) і прапор (flag) індикації знакових або беззнакових операндів. Модель завантажує ці окремі фрагменти даних у регістр інструкцій. Вихід моделі - масив з 32 інструкцій.

Лістинг 16.12. Використання масивів структур для створення моделі регістра команд.

```

package definitions;
    typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;
    typedef enum {UNSIGNED, SIGNED} operand_type_t;
    typedef union packed {
        logic [31:0] u_data;
        logic signed [31:0] s_data;
    } data_t;
    typedef struct packed {
        opcode_t opc;
        operand_type_t op_type;
        data_t op_a;
        data_t op_b;
    } instr_t;
endpackage
import definitions::*; // Імпорт пакета в простір $unit
module instruction_register (
    output instr_t [0:31] instr_reg,    // Запаковані масиви структур
    input data_t operand_a,
    input data_t operand_b,
    input operand_type_t op_type,
    input opcode_t opcode,
    input logic [4:0] write_pointer
);
always @(write_pointer) begin
    instr_reg[write_pointer].op_type = op_type;
    instr_reg[write_pointer].opc = opcode;
    // Використання op_type для визначення типів операндів,
    // збережених в об'єднанні вхідних операндів
    if (op_type == SIGNED) begin

```

```

instr_reg[write_pointer].op_a.s_data = operand_a.s_data;
instr_reg[write_pointer].op_b.s_data = operand_b.s_data;
end
else begin
instr_reg[write_pointer].op_a.u_data = operand_a.u_data;
instr_reg[write_pointer].op_b.u_data = operand_b.u_data;
end
end
endmodule

```

Спеціальні функції для роботи з масивами:

- 1) \$dimensions(array_name) - повертає число вимірів масивів;
- 2) \$left(array_name, dimension) - повертає крайній лівий індекс масиву:
 logic [1:2][7:0] word [0:3][4:1];
 \$left(word,1) will return 0
 \$left(word,2) will return 4
 \$left(word,3) will return 1
 \$left(word,4) will return 7
- 3) \$right(array_name, dimension) - повертає крайній правий індекс масиву;
- 4) \$low(array_name, dimension) - повертає наймолодший біт масиву:
 logic [7:0] word [1:4];
 \$low(word,1) returns 1, and \$low(word,2) returns 0.
- 5) \$high(array_name, dimension) - повертає найстарший біт масиву;
- 6) \$size(array_name, dimension) - повертає число елементів масиву;
- 7) \$increment(array_name, dimension) - повертає 1, якщо \$left - більше або дорівнює \$right; -1, якщо \$left менше, ніж \$right:

```

logic [3:0][7:0] array [0:1023];
int d = $dimensions(array);
if (d > 0) begin // об'єкт є масивом
for (int j = $right(array,1);
j = ($left(array,1)+ $increment(array,1) );
j += $increment(array,1))
begin
...
end
end

```

Наприклад:

```

$right(array,1) // повертає 1023
$left(array,1) // повертає 0
$increment(array,1) // повертає -1
$bits(expression) // повертає кількість бітів,
// представлених виразом

bit [63:0] a;

logic [63:0] b;

```

```
wire [3:0][7:0] c [0:15];
struct packed {byte tag; logic [31:0] addr;} d;

$bits(a)          // повертає 64
$bits(b)          // повертає 64
$bits(c)          // повертає 512
$bits(d)          // повертає 40
$bits(a+b)        // повертає 128
```

Описувані далі типи масивів не синтезовані. Вони призначені для використання в testbench.

16.4. Динамічні масиви

Динамічні масиви декларуються без вказівки їхньої довжини з порожніми дужками []. Для виділення пам'яті під масив використовується оператор new[]. Якщо вказати додаткове ім'я в операторі new[], то значення буде копіюватися в новий елемент (лістинг 16.13.).

Лістинг 16.13. Використання динамічних масивів.

```
int dyn[], d2[];      // Порожній динамічний масив
initial begin
    dyn = new[5];      // Виділення пам'яті для 5 елементів
    foreach (dyn[j])
        dyn[j] = j; // Ініціалізація елементів
    d2 = dyn;          // Копіювання динамічного масиву
    d2[0] = 5;         // Зміна копії
    $display(dyn[0],d2[0]); // Виведення обох значень (0 і 5)
    dyn = new[20](dyn); // Збільшення розміру та копіювання
    dyn = new[100];     // Виділення пам'яті для 100 нових
                        // елементів. Старі значення при цьому будуть загублені
    dyn.delete;         // Видалення всіх елементів
end
```

Функція \$size повертає розмір масивів з фіксованою і динамічною довжинами. Динамічні масиви мають кілька спеціальних методів, таких як delete і size (лістинг 16.14.). Останній повертає розмір тільки для динамічних масивів. Прототипи методів size і delete:

```
function int size();
function void delete();
```

Лістинг 16.14. Використання методів size і delete.

```
int j = addr.size;
addr = new[addr.size()*4](addr); // збільшений в 4 рази масив addr

int ab[] = new[N]; // Створення тимчасового масиву в N елементів
... // Використання ab
ab.delete; // Видалення вмісту масиву
$display ("%d", ab.size); // Виведення 0
```


Можна задавати масиви фіксованої довжини, не вказуючи їхній розмір безпосередньо. У цьому випадку елементи масиву отримують значення в момент його декларації, а компілятор обчислює розмір масиву, виходячи з кількості зазначених елементів.

У лістингу 16.15 масив `mask` одержує значення дев'яти 8-бітових масок.

Лістинг 16.15. Використання динамічного масиву для створення списку.

```
bit [7:0] mask[] = {8'b0000_0000, 8'b0000_0001, 8'b0000_0011,
                   8'b0000_0111, 8'b0000_1111, 8'b0001_1111,
                   8'b0011_1111, 8'b0111_1111, 8'b1111_1111};
```

Можна привласнювати між собою значення масивів з динамічною і фіксованою довжиною, якщо вони мають однаковий тип даних `int` і однакову довжину. Якщо привласнити масив з фіксованою довжиною динамічному масиву, то компілятор викличе конструктор `new[]`, щоб виділити пам'ять під масив, а потім скопіює в неї нові значення.

16.5. Черги

Новий тип даних. Спрощує процес пошуку та сортування в структурах. Такий же швидкий, як і масиви з фіксованою довжиною; різноманітний як зв'язаний список. Подібно до динамічних масивів черги можуть збільшуватися і зменшуватися в розмірі під час моделювання, також можна легко додавати і видаляти будь-які елементи, як це показано в наступному прикладі. Декларується як масив, але з використанням символу долара `$` (лістинг 16.16). Розмір масиву може бути зазначений, але це необов'язково.

Черга може зберігати дані типу тих, які він одержав у момент декларації черги:

```
int q1 [$]; //порожня черга, без вказівки розміру
int q2 [$] = {1,2,3,5,8}; // безрозмірна черга,
                        // ініціалізується п'ятьма елементами
typedef struct {int a, b; bit flag} packet_t;
packet_t q3 [$:16]; //черга розміром в 16 елементів
```

Лістинг 16.16. Операції над чергами

```
int j = 1, b[$] = {3,4}, q[$] = {0,2,5}; // {0,2,5} Ініціалізація черги
initial begin
    q.insert(1, j); // {0,1,2,5}, поміщає 1 перед 2
    q.insert(3, b); // {0,1,2,3,4,5}, поміщає значення b[] після 2
    q.delete(1); // {0,2,3,4,5}, видаляє елемент із індексом 1
    // Наступні оператори найшвидші
    q.push_front(6); // {6,0,2,3,4,5}, додає елемент у початок списку
    j = q.pop_back; // {6,0,2,3,4} j = 5
    q.push_back(8); // {6,0,2,3,4,8}, додає елемент у кінець списку
    j = q.pop_front; // {0,2,3,4,8} j = 6
    foreach (q[i])
        $display(q[i]);
end
```

Коли створюється черга, SystemVerilog виділяє додатковий простір, що дозволяє швидко додавати елементи. Для черг немає необхідності використовувати оператор `new[]`. Якщо в чергу додається більше елементів, перевищуючи, таким чином, розмір вільного простору, SystemVerilog автоматично виділяє додаткове місце. Швидко виконується операція читання (`pop`) і запису (`push`) з початку або кінця масиву. Для її виконання використовується фіксований час. Більше часу буде потрібно на виконання цих операцій із середини масиву, особливо якщо він має більшу довжину, оскільки симулятору буде необхідно змістити половину масиву.

У чергу можна копіювати значення масивів фіксованої та динамічної довжини.

Черги можуть бути використані для створення моделей FIFO, LIFO або інших типів пам'яті з послідовним доступом.

16.5.1. Методи черг.

В SystemVerilog для черг визначені вбудовані методи:

- 1) `insert(value)` - додає елемент у зазначену індексну позицію;
- 2) `delete(value)` - видаляє елемент із описаної індексної позиції;
- 3) `push_front(<value>)` - додає значення змінної в нову позицію на початку черги;
- 4) `push_back(<value>)` - додає значення змінної в нову позицію наприкінці черги;
- 5) `variable = pop_front()` - видаляє перший елемент із черги і повертає в його значення;
- 6) `variable = pop_back()` - видаляє останній елемент із черги й повертає в його значення;
- 7) `insert(<index>,<value>)` - змінює значення елемента в зазначеній позиції без зміни розміру черги;
- 8) `variable = <queue_name>[<index>]` - повертає значення елемента із зазначеної позиції без зміни розміру черги;
- 9) `variable = size()` - повертає поточну кількість елементів у черзі.

Запис у заповнену чергу й читання з порожньої черги приведе до помилки часу виконання.

16.5.2. Пам'ять FIFO, що параметризується.

Лістинг 16.17 ілюструє використання черг і їхніх методів у моделях, розроблених за допомогою SystemVerilog.

Приклад являє собою найпростішу пам'ять FIFO, що створюється із застосуванням черг в SystemVerilog. Розмір і розрядність даних FIFO контролюється параметрами. Розміщення в пам'ять й видалення даних з пам'яті FIFO, точно так само як і перевірка елементів в FIFO, реалізується за допомогою методів черг.

Черга також моделюється за допомогою SystemVerilog типів із двома станами `int` і `bit`, де зміст шини й дозвіл ланцюга не важливі, типи із двома станами поліпшують продуктивність.

У середовищі верифікації черги можуть використовуватися для збору інформаційних даних.

Лістинг 16.17. Модель FIFO з використанням черг.

```
'timescale 1ns / 1ns
module FIFO #(parameter int DEPTH = 31, parameter int WIDTH = 8) (
    input bit [ WIDTH-1:0] DATA,
    input bit CLK, RSTb, WENb, RENb,
    output bit FULL, EMPTY,
    output bit [ WIDTH-1:0] Q;
    bit [ WIDTH-1:0] mem [ $:DEPTH];
    // Запис даних в FIFO
    always @(posedge CLK, negedge RSTb)
        if (RSTb == 0)
            mem = '{};
        else if (WENb == 0 && mem.size() < DEPTH)
            mem.push_back(DATA);

    // Читання даних з FIFO
    always @(posedge CLK)
        if (RENb == 0 && mem.size() > 0)
            Q <= mem.pop_front();
    // FIFO control flags
    assign EMPTY = (mem.size() == 0) ? 1:0;
    assign FULL = (mem.size() == DEPTH) ? 1:0;
endmodule
```

16.6. Асоціативні масиви

Для зберігання в пам'яті більших обсягів даних SystemVerilog пропонує асоціативні масиви, що зберігають входи для розріджених матриць. На рис.16.6 представлений масив, що зберігає інформацію в позиціях 0:3, 42, 1000, 4521, 200 000. Для створення його моделі при використанні асоціативних масивів буде потрібно значно менше пам'яті, ніж якби це був масив з фіксованою й динамічною довжиною з 200 001 елементів.

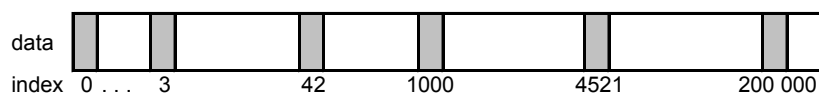


Рис. 16.6. Приклад асоціативного масиву

Синтаксис декларації асоціативного масиву:

```
data_type array_id [index_type];
```

де `data_type` - тип даних елементів масиву, може бути будь-який тип, дозволений для масивів фіксованої довжини; `array_id` - ім'я декларованого масиву; `index_type` - тип даних, використовуваний для індексу або *. Символ * озна-

час, що масив може індексуватися будь-яким цілим виразом. При вказівці типу індексу використовуваний вираз для індексації повинен бути тільки зазначеного типу.

Лістинг 16.18 представляє декларування, ініціалізацію й використання асоціативних масивів. Їхня декларація виконується за допомогою групового символу [*].

Лістинг 16.18. Декларування, ініціалізація та використання асоціативних масивів.

```
initial begin
    logic [63:0] assoc[*], idx = 1;
    // Ініціалізація розрізненими значеннями
    repeat (64) begin
        assoc[idx] = idx;
        idx = idx << 1;
    end
    // Перебір всіх індексів за допомогою foreach
    foreach (assoc[i])
        $display("assoc[%h] = %h", i, assoc[i]);
    // Перебір всіх індексів за допомогою функції
    if (assoc.first(idx))
        begin // Отримання першого індексу
            do
                $display("assoc[%h]=%h", idx, assoc[idx]);
                while (assoc.next(idx)); // Отримання наступного індексу
            end
            // Пошук і видалення першого елемента
            assoc.first(idx);
            assoc.delete(idx);
        end
end
```

Асоціативний масив assoc складається з розсіяних елементів 1, 2, 4, 8, 16. Для перебору елементів не зручно використовувати оператор for, для цієї мети краще підходить операція foreach. Для більш точного контролю можна також використовувати функції first і next у циклі do...while. Ці функції приймають значення індексу як аргумент і повертають 0 або 1, залежно від наявності елементів у масиві. Асоціативні масиви можуть бути також адресовані за допомогою строкового індексу. У прикладі (лістинг 16.19) виконується читання пар імен з файлу в асоціативний масив. Якщо спробувати прочитати елемент, що ще не існує, то SystemVerilog поверне 0 або X для типів з 2-ма або 4-ма значеннями, відповідно. Функція exists може бути використана для перевірки існування елемента.

Лістинг 16.19. Застосування асоціативного масиву з індексом string.

```
// Вхідний файл містить: 42 min_address 42 max_address 1492
int switch[string], min_address, max_address;
initial begin
```

```

int i, r, file;
string s;
file = $fopen("switch.txt", "r");
while (! $feof(file)) begin
    r = $fscanf(file, "%d %s", i, s);
    switch[s] = i;
end
$fclose(file);
// Одержання мінімальної адреси, за замовчуванням 0
mid_address = switch["min_address"];
// Одержання максимальної адреси, за замовчуванням 1000
if (switch.exists("max_address"))
    max_address = switch["max_address"];
else
    max_address = 1000;
end

```

Асоціативний масив може бути збережений симулятором у вигляді дерева. Крім того, допускається використовувати додаткові службові сигнали (overhead), коли необхідно зберегти масив зі значеннями індексів, які відстоять далеко один від одного, наприклад для пакетів, адресованих 32-розрядним або 64-розрядним значенням даних.

16.6.1. Методи асоціативних масивів.

Визначено кілька методів для аналізу і маніпулювання елементами асоціативних масивів.

1. Функція num():

```
function int num();
```

Метод num() повертає число елементів в асоціативному масиві. Повертає 0, якщо масив порожній:

```

int item[*];
item[ 2'b3 ] = 1;
item[ 16'hffff ] = 2;
item[ 4b'1000 ] = 3;
$display( "%0d entries\n", item.num ); // prints "3 entries"

```

2. Функція delete():

```
function void delete( [input index] );
```

де index - необов'язковий параметр індексу відповідного типу для масиву.

Якщо індекс описаний, то метод delete() видаляє елемент, що відповідає індексу; якщо видаляється елемент, що не існує, ніяких попереджень згенеровано не буде. Якщо індекс не описаний, то метод видаляє всі елементи масиву:

```

int map[ string ];
map["systemverilog"] = 1;
map["is"] = 2;
map["not easy"] = 3;
map.delete("sad"); // видаляє елемент із індексом "sad"

```

```
map.delete;          // видаляє всі елементи асоціативного масиву map
```

3. Функція exists():

```
function int exists( input index );
```

де index - значення відповідного типу індексу масиву.

Функція exists() перевіряє існування в заданій індексній позиції елемента.

Повертає 1, якщо елемент є, інакше - 0:

```
if ( map.exists("SystemVerilog"))
    map["SystemVerilog"] += 1;
else
    map["SystemVerilog"] = 0;
```

4. Функція first():

```
function int first( ref index );
```

де index - значення відповідного типу індексу масиву.

Метод first() передає через посилальний параметр index значення першого (найменшого) індексу масиву. Повертає 0, якщо масив порожній, і 1 - у протилежному випадку:

```
string s;
if (map.first( s ))
$display("First entry is : map[ %s ] = %0d\n", s, map[s]);
```

5. Функція last():

```
function int last( ref index );
```

де index - значення відповідного типу індексу масиву.

Метод last() передає через посилальний параметр index значення останнього (максимального) індексу масиву. Повертає 0, якщо масив порожній, і 1 - у протилежному випадку:

```
string s;
if (map.last( s ))
$display("Last entry is : map[ %s ] = %0d\n", s, map[s]);
```

6. Функція next():

```
function int next( ref index );
```

де index - значення відповідного типу індексу масиву.

Метод next() виконує пошук елемента, чий індекс більший від заданого індексу. Якщо в масиві є ще елементи, то виконується зв'язок із сусіднім елементом і функція повертає 1. Інакше індекс не міняється, і функція повертає 0:

```
string s;
if (map.first( s ))
do
    $display("%s : %d\n", s, map[s]);
```

```
while (map.next( s ));
```

7. Функція prev():

```
function int prev( ref index );
```

де index - значення відповідного типу індексу масиву.

Метод `prev()` виконує пошук елемента, чий індекс менший від заданого індексу. Якщо в масиві є ще елементи, то виконується зв'язок із сусіднім елементом і функція повертає 1. Інакше індекс не змінюється, і функція повертає 0.

```
string s;
if (map.last( s ))
    do
        $display("%s : %d\n", s, map[ s ]);
    while ( map.prev( s ) );
```

Якщо аргумент, переданий в будь-який із чотирьох методів переміщення для асоціативних масивів, менший ніж розмір відповідного індексу, то функція повертає значення -1. Тому необхідно копіювати стільки даних, скільки може підійти аргументу.

Наприклад:

```
string aa[*];
byte ix;
int status;
aa[1000] = "a";
status = aa.first(ix);
// status is -1
// ix is 232 (останні 8 значущих бітів від 1000)
```

16.6.2. Зв'язкові списки (Linked Lists).

SystemVerilog підтримує зв'язкові списки, аналогічні STL (Standard Template Library) контейнерам списків. Контейнер визначається як параметризований клас. Це означає, що він може бути змінений для зберігання даних будь-якого типу. Однак в SystemVerilog зв'язковим спискам слід віддавати перевагу чергам, які ефективніші й простіші у використанні.

16.7. Методи масивів

Існує багато методів масивів, які можуть бути використані для будь-яких незапакованих масивів: фіксованих, динамічних, черг і асоціативних.

16.7.1. Методи редукції масивів.

Базовий метод редукції приймає масив даних і повертає скалярне значення.

Найзагальнішим методом редукції є `sum`, що складає разом всі значення масиву. Потрібно бути обережним з визначенням розміру результату в SystemVerilog. За замовчуванням, якщо скласти елементи однобітового масиву, то результат буде однорозрядний (лістинг 16.20). Однак якщо скласти або порівняти цей результат з 32-розрядним числом, то SystemVerilog вирівнює розміри у бік збільшення.

Лістинг 16.20. Створення суми масивів.

```
bit on[10];           // Масив бітів
int yum;
initial begin
    foreach (on[i])
        on[i] = i;           // on[i] отримує значення 0 або 1
```

```

        // Виведення на друк однобітової суми
        $display("on.sum = %0d", on.sum); // on.sum = 1
        // Сума значень привласнюється 32-бітовій змінній yum
        yum = on.sum;
        $display("sum = %0d", yum); // sum = 5
        // Порівняння суми з 32-бітовим значенням
        if (on.sum >= 32'd5) // True
            $display("sum has 5 or more 1's");
    end

```

Інші методи редукції: product, and, or і xor.

16.7.2. Методи місця розташування.

Використовуються для пошуку даних в незапакованих масивах. Ці методи завжди повертають чергу.

Приклад використовує масив фіксованого розміру f[6], динамічний масив d[] і чергу q[\$]. Функції min і max знаходять мінімальний і максимальний елементи масиву. Ці методи також працюють і для асоціативних масивів. Метод unique повертає чергу унікальних значень масиву, дублікати не включаються (лістинг 16.21).

Лістинг 16.21. Використання методів: min, max, unique.

```

int f[6] = '{1,6,2,6,8,6};
int q[$] = '{1,3,5,7}, tq[$];
tq = q.min;           // {1}
tq = q.max;           // {7}
tq = f.unique;        // {1,6,2,8}

```

Можна виконувати операцію пошуку за допомогою оператора циклу foreach, але в System Verilog простіше виконати це за допомогою методів місця розташування (locator) (лістинг 16.22). Вираз with використовується SystemVerilog для виконання пошуку.

Лістинг 16.22. Використання методу місця розташування find.

```

int d[] = '{9,1,8,3,4,4}, tq[$];
        // Пошук всіх елементів, більших ніж 3
tq = d.find with (item > 3); // {9,8,4,4}
        // Еквівалентний код
tq.delete;
foreach (d[i])
    if (d[i] > 3) tq.push_back(d[i]);
tq = d.find_index with (item > 3);           // {0,2,4}
tq = d.find_first with (item > 99);          // {} – елементи не знайдені
tq = d.find_first_index with (item==8);      // {2} d[2]=8
tq = d.find_last with (item==4);             // {4}
tq = d.find_last_index with (item==4);       // {6} d[6]=4

```

У лістингу 16.23 оператор sum видає кількість, для якого вираз має значення істина. Для першого оператора вертається число елементів більше 7 - це 9 і 8, тому count = 2. Варто звернути увагу, що sum-with - це оператор, а не ви-

раз, таким чином, результат необхідно зберігати в часову змінну, а не використовувати прямо.

Лістинг 16.23. Приклади застосування методів місця розташування.

```
int count, d[] = {9,1,8,3,4,4};
count = d.sum with (item > 7);           // 2: {9, 8}
count = d.sum with (item < 8);           // 4: {1, 3, 4, 4}
count = d.sum with (item == 4);          // 2: {4, 4}
```

Вибір типів пам'яті (Choosing a Storage Type). Вибір здійснюється на основі необхідних можливостей: гнучкості, використання пам'яті, швидкості та сортування. Дані правила можуть змінюватися залежно від компілятора.

Гнучкість. У випадку використання послідовних позитивних індексів застосовуються масиви фіксованої або динамічної довжини: фіксованої довжини, якщо розмір масиву відомий на момент компіляції, і динамічної - у протилежному випадку.

16.8. Контрольні запитання і завдання

1. Визначено структуру

```
typedef struct {
    logic [31:0] a, b;
    logic [ 7:0] opcode ;
    logic [23:0] address ;
} instr_t ;
instr_t IW;
```

Що означають такі вирази?

- a) IW = '{100, 5, 8'hFF, 0};
- б) IW = '{address:0, opcode:8'hFF, a:100, b:5};
- в) IW = '{address:0, 8'hFF, 100, 5};
- г) IW = '{default:0};

2. Що таке запакована структура?

3. Що означає вираз?

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
```

4. Що означає даний код?

```
typedef struct packed {
    logic valid;
    logic [ 7:0] tag;
    logic [31:0] data;
} data_word_t;
```

```
data_word_t packet_in, packet_out;
always @(posedge clock)
    packet_out <= packet_in << 2;
```

5. Які обмеження існують при визначенні запакованих об'єднань?

6. Що означає даний вираз?

- a) wire n [0:1023];
- б) reg [7:0] LUT [0:255];

в) `real r [0:1023];`

г) `integer i [7:0][3:0][7:0];`

д) `int array3 [8][4];`

7. Дати поняття запакованого та незапакованого масивів.

8. Що означає такий вираз?

а) `int ascend[4] = '{0,1,2,3};`

б) `int md[2][3] = '{0,1,2}, '{3,4,5}};`

в) `bit [3:0] [7:0] bytes;`

г) `bit [3:0] [7:0] barray [3];`

д) `foreach (dst[j])`

9. Що таке динамічний масив?

10. Дати визначення черги.

11. Дати визначення асоціативного масиву.

12. Що означає даний вираз?

а) `int dyn[], d2[];`

б) `b[$] = {3,4};`

в) `q[$] = {0,2,5};`

13. Що означає даний вираз?

а) `q[$] = {0,2,5};`

...

`q.insert(1, j);`

б) `q[$] = {0,2,5};`

...

`q.delete(1);`

14. Що означає даний вираз?

`logic [63:0] assoc[*], idx = 1;`

15. Яке значення отримає змінна `summ`?

`bit on[8];`

`int summ;`

...

`on = '{1,0,0,1,1,1,0,0};`

`summ = on.sum;`

16. Яке значення отримає змінна `tq`?

`int f[6] = '{1,6,2,6,8,6};`

...

`tq = f.unique;`

17. ПРОЦЕДУРНІ БЛОКИ, ЗАВДАННЯ Й ФУНКЦІЇ

17.1. Процедурні блоки

Щоб виключити неоднозначність блоку `always`, в SystemVerilog додані три спеціальні його версії: `always_comb`, `always_latch` і `always_ff`.

Ці три блоки мають нескінченну природу моделювання, як і оператор `always`. Однак вони поліпшують стиль опису апаратури та призначені для розробки синтезованих моделей RTL-рівня.

Програмним засобам немає необхідності визначати тип апаратури з контексту опису операторів. Якщо стиль запису операторів не задовольняє характеру блоку, то система розробки генерує попередження.

17.1.1. Комбінаційний процедурний блок.

На відміну від звичайного блоку `always`, блок `always_comb` не вимагає вказівки спеціального списку чутливості. Він створюється автоматично і у нього включаються всі змінні, значення яких читається в процедурному блоці, за винятком тих, що оголошені в ньому. У наступному прикладі оператор `always_comb` буде виконуватися при кожній зміні змінних `a` або `b`:

```
always_comb
if (!mode)
    y = a + b;
else
    y = a - b;
```

Лістинг 17.1 представляє приклад використання `always_comb` для опису комбінаційних фрагментів схеми контролера.

Лістинг 17.1. Модель контролера з використанням `always_comb`.

```
module controller (output logic read, write,
    input instr_t instruction,
    input logic clock, reset);
    enum {WAITE, LOAD, STORE} State, NextState;
    always @(posedge clock, negedge reset)
        if (!reset) State <= WAITE;
        else State <= NextState;
    always_comb begin
        case (State)
            WAITE: NextState = LOAD;
            LOAD: NextState = STORE;
            STORE: NextState = WAITE;
        endcase
    end
    always_comb begin
        read = 0; write = 0;
```

```

        if (State == LOAD && instruction == FETCH) read = 1;
        else if (State == STORE && instruction == WRITE) write = 1;
    end
endmodule

```

Існує розходження в моделюванні між `always_comb` і `always`. Перший виконується один раз в нульовий момент моделювання, після активації всіх процедурних блоків.

Verilog-2001 пропонує використовувати в списку чутливості блоку `always` груповий символ `@*` або `@(*)`. Однак це тільки більш короткий запис і не дає таких переваг, як `always_comb` (лістинг 17.2).

Лістинг 17.2. Розходження між `always_comb` і `always`.

```

always @* begin                // Позначає @(data)
    a1 = data << 1;
    b1 = decode ();
    ...
end
always_comb begin              // Позначає @(data, sel, c, d, e)
    a2 = data << 1;
    b2 = decode ();
    ...
end
function decode;                // Функція не має входів
begin
    case (sel)
        2'b01 : decode = d | e;
        2'b10 : decode = d & e;
        default : decode = c;
    endcase
end

```

17.1.2. Послідовний процедурний блок `always_latch`.

Другий спеціалізований оператор `always` - `always_latch`. Це процедурний блок, що моделює тригер-засувку.

```

always_latch
    if (enable) q <= d;

```

Приклад з лістинга 17.3 використовує `always_latch`. 5-бітовий лічильник виконує рахунок від 0 до 31. Вхід `ready` контролює початок виконання рахунку, має значення 1 короткий період часу. Тому, коли `ready` перемикається в 1, модель зберігає це значення для внутрішнього сигналу `enable`. Засувка зберігає значення 1, поки лічильник не досягне значення 21, і потім виконується очищення сигналу `enable`, не дозволяючи лічильнику виконуватися знову, до наступного надходження сигналу `ready`.

Лістинг 17.3. Використання процедурного блоку `always_latch`.

```

module register_reader (input clk, ready, reset,
                        output logic [4:0] read_pointer);
    logic enable; // internal enable signal for the counter

```

```

        logic overflow; // internal counter overflow flag
always_latch begin // latch the ready input
    if (!reset)
        enable <= 0;
    else if (ready)
        enable <= 1;
    else if (overflow)
        enable <= 0;
    end
always @(posedge clk, negedge reset) begin // 5-bit counter
    if (!reset)
        {overflow,read_pointer} <= 0;
    else if (enable)
        {overflow,read_pointer} <= read_pointer + 1;
    end
endmodule

```

17.1.3. Послідовний процедурний блок.

Блоки, що описують послідовну логіку, можуть моделюватися за допомогою `always_ff`:

```

always_ff @(posedge clock, negedge reset)
    if (!reset) q <= 0;
    else q <= d;

```

Всі сигнали в списку чутливості повинні бути записані із вказівкою фронту `posedge` або `negedge`. Подієвий контроль усередині блоку не допускається.

Блоки `always_comb`, `always_latch` і `always_ff` є синтезованими.

17.2. Завдання, функції й void-функції

Verilog проводить чітку границю між функціями та завданнями. Функції не можуть містити оператори, що керують часом або подіями, завжди повинні повертати одне значення.

В SystemVerilog, якщо необхідно при виклику функції ігнорувати значення, що повертається, то її результат описується за допомогою `void`. Наприклад, при ігноруванні значення, що повертається функцією

```
void'(my_func(42));
```

Деякі симулятори дозволяють ігнорувати значення, що повертається, без використання синтаксису `void`.

Void-Функції викликаються як завдання, у вигляді оператора, але на них накладають такі ж обмеження, як і на звичайні функції. В них не можна використовувати події й затримки, з них не допускається виконувати виклик завдань.

Приклад (лістинг 17.4.) представляє void-функцію, що отримує дані у вигляді 64-розрядного вхідного вектора з елементів logic і інформацію, що повертає, у вигляді структури packet_t.

Лістинг 17.4. Функція void.

```
typedef struct {
    logic valid;
    logic [ 7:0] check;
    logic [63:0] data;
} packet_t;
function void fill_packet (
    input logic [63:0] data_in,
    output packet_t data_out );
    data_out.data = data_in;
    for (int i=0; i<=7; i++)
        data_out.check[i] = ^data_in[(8*i)+:8];
    data_out.valid = 1;
endfunction
```

17.2.1. Повернення з підпрограми.

В SystemVerilog, щоб підвищити контроль за послідовністю виконання операторів підпрограми, доданий оператор return. У наступному прикладі (лістинг 17.5) повернення з підпрограми виконується у випадку наявності помилки - неправильного значення оператора. Це ж завдання можна вирішити за допомогою конструкції else, однак це ускладнить читання коду. Оператор return може бути використаний і з функціями (лістинг 17.6).

Лістинг 17.5. Використання оператора повернення в завданні.

```
task load_array(int len, ref int array[]);
    if (len <= 0) begin
        $display("Bad len");
    return;
end
    // Інший код завдання
...
endtask
```

Лістинг 17.6. Повернення з функції

```
function bit transmit(...);
    // Передача транзакції
    ...
    return ~ifc.cb.error; // Повернення статусу: 0=error
endfunction
```

Нижче наведене порівняння стилів опису функції в Verilog (лістинг 17.7, а) і SystemVerilog (лістинг 17.7, б, в).

Лістинг 17.7. Функції Verilog і SystemVerilog.

а) // Приклад Verilog-Функції

```
function [31:0] add_and_inc (input [31:0] a,b);
```

```

begin
    add_and_inc = a + b + 1;
end
endfunction

```

б) // Приклад System Verilog-функції

```

function int add_and_inc (input int a, b);
    return a + b + 1;
endfunction

```

в) // Приклад System Verilog-функції

```

function int add_and_inc (input int a, b);
    add_and_inc = a + b;
    return ++add_and_inc;
endfunction

```

Verilog дозволяє виконувати групування виразів за допомогою блоку begin...end. У завданнях дозволено також групувати трохи операторів, застосовуючи fork...join. SystemVerilog дозволяє обійтися без використання блоків, у цьому випадку оператори виконуються послідовно:

```

function states_t NextState(states_t State);
    NextState = State; // default next state
    case (State)
        WAITE: if (start) NextState = LOAD;
        LOAD: if (done) NextState = STORE;
        STORE: NextState = WAITE;
    endcase
endfunction

```

Допускається передавати аргументи функції через їхнє ім'я:

```

// Стил System Verilog виклику функції
always @(posedge clock)
    result <= divide(.denominator(b), .numerator(a) );

```

В SystemVerilog за замовчуванням функція повертає значення, що має тип logic. Якщо формальний аргумент отримує значення в момент декларації, то при виклику функції його можна не вказувати:

```

function int incrementer(int count=0, step=1);
    incrementer = count + step;
endfunction

```

Допускається передавати посилання на аргумент за допомогою конструкції ref. Це використовується в автоматичних функціях (лістинг 17.8).

Лістинг 17.8. Приклад використання автоматичної функції.

```

module chip (...);
    typedef struct {
        logic valid;
        logic [ 7:0] check;
        logic [63:0] data;
    } packet_t;

```

```

        packet_t data_packet;
        logic [7:0] raw_data [0:7];
always @(posedge clock)
    if (data_ready)
        fill_packet (.data_in(raw_data), .data_out(data_packet) );
function automatic void fill_packet (
    ref logic [7:0] data_in [0:7], // ref arg
    ref packet_t data_out ); // ref arg
    for (int i=0; i<=7; i++) begin
        data_out.data[(8*i)+:8] = data_in[i];
        data_out.check[i] = ^data_in[i];
    end
    data_out.valid = 1;
endfunction
...
endmodule

```

17.3. Процедурні оператори і підпрограми

SystemVerilog успадкував багато операторів з C і C++.

17.3.1. Нові оператори.

1) Інкремент - декремент (табл. 17.1).

Таблиця 17.1. Оператори інкремента і декремента

Вираз	Операція	Опис
<code>j = i++;</code>	пост-інкремент	j отримує значення i, після чого i збільшується на 1
<code>j = ++i;</code>	пре-інкремент	i збільшується на 1 та j отримує оновлене значення i
<code>j = i--;</code>	пост-декремент	j отримує значення i, після чого i зменшується на 1
<code>j = --i;</code>	пре-декремент	i зменшується на 1 та j отримує оновлене значення i

Приклад використання оператора інкремента в циклі for:

```
for (i = 0; i <= 31; i++) begin
```

```
    ...
```

```
end
```

Застосування операторів інкремента в циклах while:

```
while (i++ < LIMIT) begin: loop1
```

```
    ... // Останнє значення i повинне бути LIMIT
```

```
end
```

```
while (++j < LIMIT) begin: loop2
```

```
    ... // Останнє значення j повинне бути LIMIT -1
```

```
end
```

Оператори інкремента ++ і декремента - обробляються програмами моделювання як блокуючі. Це синтезовані оператори.

2) Оператори присвоєння.

SystemVerilog отримав кілька нових операторів (табл. 17.2), які комбінують арифметичну або логічну операцію із присвоєнням.

Наприклад, оператор `+=`:

```
out += in;
```

підсумовує `in` з `out` і присвоює результат `assign out`. Оператор `+=` - це коротка форма виразу:

```
out = out + in;
```

Таблиця 17.2. Нові оператори присвоєння в SystemVerilog

Оператор	Опис
<code>+=</code>	Підсумовується оператор з лівого боку виразу з оператором із правого боку, потім виконується присвоєння
<code>-=</code>	З оператора з лівого боку виразу віднімається значення із правого боку, потім виконується присвоєння
<code>*=</code>	Множення
<code>/=</code>	Ділення оператора з лівого боку на значення виразу із правого
<code>%=</code>	Ділення оператора з лівого боку на значення виразу із правого, присвоюється залишок
<code>&=</code>	Побітова операція І між лівим і правим операндом
<code> =</code>	Побітова операція АБО між лівим і правим операндом
<code>^=</code>	Побітова операція по модулю два між лівим і правим операндом
<code><<=</code>	Зсув оператора з лівого боку вліво виразу на кількість біт, що задаються правим операндом
<code>>>=</code>	Зсув оператора з лівого боку вправо виразу на кількість біт, що задаються правим операндом
<code><<<=</code>	Арифметичний зсув оператора з лівого боку вліво виразу на кількість біт, що задаються правим операндом
<code>>>>=</code>	Арифметичний зсув оператора з лівого боку вправо виразу на кількість біт, що задаються правим операндом

Оператори присвоєння з додатковою операцією моделюються як блокуючий процедурний оператор присвоєння. Вони можуть бути використані в синтезованих моделях, але з обмеженнями (лістинг 17.9).

Лістинг 17.9. Використання операторів присвоєння в SystemVerilog.

```
package definitions;
    typedef enum logic [2:0] {ADD,SUB,MULT,DIV,SL,SR} opcode_t;
    typedef enum logic {UNSIGNED, SIGNED} operand_type_t;
    typedef union packed {
        logic [23:0] u_data;
        logic signed [23:0] s_data;
    } data_t;
    typedef struct packed {
        opcode_t opc;
        operand_type_t op_type;
    }
```

```

        data_t op_a;
        data_t op_b;
    } instruction_t;
endpackage
import definitions::*; // Імпорт пакета в простір компіляції $unit
module alu (input instruction_t instr, output data_t alu_out);
    always_comb begin
        if (instr.op_type == SIGNED) begin
            alu_out.s_data = instr.op_a.s_data;
            unique case (instr.opc)
                ADD : alu_out.s_data += instr.op_b.s_data;
                SUB : alu_out.s_data -= instr.op_b.s_data;
                MULT : alu_out.s_data *= instr.op_b.s_data;
                DIV : alu_out.s_data /= instr.op_b.s_data;
                SL : alu_out.s_data <<= 2;
                SR : alu_out.s_data >>= 2;
            endcase
        end
        else begin
            alu_out.u_data = instr.op_a.u_data;
            unique case (instr.opc)
                ADD : alu_out.u_data += instr.op_b.u_data;
                SUB : alu_out.u_data -= instr.op_b.u_data;
                MULT : alu_out.u_data *= instr.op_b.u_data;
                DIV : alu_out.u_data /= instr.op_b.u_data;
                SL : alu_out.u_data <= 2;
                SR : alu_out.u_data >= 2;
            endcase
        end
    end
endmodule

```

17.3.2. Оператори порівняння.

Додано нових операторів `==?` and `!=?` (wildcard equality operator). Для них Z і X тільки в правому операнді розглядаються як груповий символ, що порівнюється з відповідною позицією іншого операнда (табл. 17.3).

Таблиця 17.3. Оператори порівняння в SystemVerilog

a	b	a == b	a === b	a ==? B	a != b	a !== b	a !=? b
0000	0000	true	true	true	false	false	false
0000	0101	false	false	false	true	true	true
010Z	0101	unknown	false	unknown	unknown	false	unknown
010Z	010Z	unknown	true	true	unknown	false	false
010X	010Z	unknown	false	true	unknown	true	false
010X	010X	unknown	true	true	unknown	false	false

Приклад:

```
logic [7:0] opcode;
```

```
...
if (opcode ==? 8'b11011???) // маскуються тільки молодші біти
```

Оператор синтезується, якщо оператор, що маскує, є константою:

```
logic [3:0] a, b;
```

```
logic y1, y2;
```

```
assign y1 = (a ==? 4'b1??1); // Синтезований
```

```
assign y2 = (a ==? b); // Несинтезований
```

17.3.3. Оператор inside.

SystemVerilog пропонує оператор inside, що призначений для пошуку значення змінної в заданій множині:

```
logic [2:0] a;
```

```
if (a inside {3'b001, 3'b010, 3'b100})
```

Така форма спрощує запис множинних порівнянь. Без оператора inside представлений вище вираз буде виглядати так:

```
if ((a==3'b001) || (a==3'b010) || (a==3'b100))
```

В операторі inside безліч значень для порівняння може бути представлено безліччю сигналів

```
if ( data inside {bus1, bus2, bus3, bus4} )
```

або бути масивом

```
int d_array [0:1023];
```

```
if ( 13 inside {d_array} ).
```

В операторі inside значення Z можна замінити символом «?».

Наступний вираз буде мати значення істина, якщо a прийме одне із значень:

3'b101, 3'b111, 3'b1x1 або 3'b1z1:

```
logic [2:0] a;
```

```
if (a inside {3'b1?1})
```

...

Точно так само, як і в операторі if, конструкція inside може бути використана в операторі case:

```
always_comb begin
```

```
case (instruction) inside
```

```
4'b0???: opc = instruction[2:0];
```

```
4'b1000, 4'b1100: opc = 3'b000;
```

```
default: opc = 3'b111;
```

```
endcase
```

```
end
```

Оператор inside є синтезованим.

17.3.4. Цикл for.

В SystemVerilog змінна, яка використовується для індексації циклу for, може декларуватися безпосередньо в ньому (лістинг 17.10). В Verilog така змінна повинна була бути оголошена до створення циклу.

Лістинг 17.10. Стилї циклів SystemVerilog.

```
module chip (...);
    ...
    always_ff @(posedge clock) begin
        for (bit [4:0] i = 0; i <= 15; i++)
            ...
    end
    always_ff @(posedge clock) begin
        for (int i = 1; i <= 1024; i += 1)
            ...
    end
endmodule
```

Змінна, декларована в операторі циклу, є автоматичною. До неї не можна звертатися через ієрархічне ім'я, і вона не доступна за межами циклу. Допускається виконувати оголошення декількох змінних у циклі, наприклад:

```
for (int i=1, j=0; i*j < 128; i++, j+=3)
    ...
```

Змінні циклу можуть мати різні типи даних:

```
for (int i=1, byte j=0; i*j < 128; i++, j+=3)
```

Якщо блоки begin або fork мають мітки, то вони можуть бути присвоєні конструкціям end або join, які завершають блок. Це спрощує пошук початку й кінця блоку. Мітки можуть мати й інші оператори SystemVerilog, такі як endmodule, endtask, endfunction та інші. Приклад (лістинг 17.11) представляє використання деяких нових конструкцій.

Лістинг 17.11. Використання міток.

```
initial
    begin : example
        integer array[10], sum, j;
        // Декларація i для оператора for
        for (int i=0; i<10; i++)          // Інкремент i
            array[i] = i;
        // Підсумовування значень елементів масиву
        sum = array[9];
        j=8;
        do                                // Цикл do...while
            sum += array[j];              // Акумулятор
        while ( j--);                     // Перевірка j на рівність 0
        $display("Sum=%4d", sum);         // %4d -кількість символів
    end : example // Мітка кінця блоку
```

17.3.5. Новий цикл do...while.

В SystemVerilog доданий новий цикл do ... while, відсутній в Verilog. Його синтаксис

```
do <statement or statement block>
```

```
...
```

```
while (<condition>);
```

Такий цикл виконується як мінімум один раз (лістинг 17.12).

Лістинг 17.12. Приклад використання циклу do.

```
always_comb begin
  do begin
    done = 0;
    OutOfBound = 0;
    out = mem[addr];
    if (addr < 128 || addr > 255) begin
      OutOfBound = 1;
      out = mem[128];
    end
    else if (addr == 128) done = 1;
      addr -= 1;
    end
  while (addr >= 128 && addr <= 255);
end
```

Цикл do синтезується, але з обмеженнями.

Перебір всіх елементів масиву в SystemVerilog можна виконувати також за допомогою оператора foreach.

17.4. Оператори переходу

В Verilog для виходу з іменних блоків і циклів використовувався оператор disable (лістинг 17.13). Крім того, оператор disable може також застосовуватися для переривання виконання завдань (лістинг 17.14).

Лістинг 17.13. Використання disable для виходу з іменних блоків.

```
// Пошук першого біта множини в заданому діапазоні бітів
always @* begin
  begin: loop
    integer i;
    first_bit = 0;
    for (i=0; i<=63; i=i+1) begin: pass
      if (i < start_range)
        disable pass; // Триває робота блоку loop
      if (i > end_range)
        disable loop; // Вихід із блоку loop
      if ( data[i] ) begin
        first_bit = i;
        disable loop; // Вихід із блоку loop
      end
    end
  end
end
```

```

        end
    end                                // Завершення блоку pass
end                                    // Завершення блоку loop
...                                  // Інші оператори, що обробляють дані
end

```

Лістинг 17.14. Використання disable у завданнях.

```

task add_up_to_max
    (input [ 5:0] max, output [63:0] result);
    integer i;

    begin
        result = 1;
        if (max == 0)
            disable add_up_to_max; // Вихід із завдання
        for (i=1; i<=63; i=i+1) begin
            result = result + result;
            if (i == max)
                disable add_up_to_max; // Вихід із завдання
        end
    end
endtask

```

В SystemVerilog додаються оператори з мови Cі: break, continue і return, що дає можливість створювати зрозумілий і компактний код. Ці оператори можуть бути застосовані й до поточного потоку.

Оператор continue виконує перехід до наступної ітерації циклу (лістинг 17.15). При цьому немає необхідності використовувати іменні блоки begin...end, як у випадку застосування оператора disable. Оператор break здійснює негайне завершення роботи циклу (лістинг 17.16).

Лістинг 17.15. Використання continue у циклі.

```

logic [15:0] array [0:255];
always_comb begin
    for (int i = 0; i <= 255; i++) begin : loop
        if (array[i] == 0)
            continue; // Пропуск порожніх елементів
        transform_function(array[i]);
    end // Кінець циклу
end

```

Лістинг 17.16. Використання break для завершення циклу.

```

// Пошук першого біта множини в заданому діапазоні бітів
always_comb begin
    first_bit = 0;
    for (int i=0; i<=63; i=i+1) begin
        if (i < start_range) continue;
        if (i > end_range) break; // Вихід із циклу
        if ( data[i] ) begin
            first_bit = i;
        end
    end
end

```

```

                                break;           // Вихід із циклу
                                end
                                end               // Кінець циклу
... // Інші оператори, що обробляють дані
end

```

Оператор `return` (лістинг 17.17) використовується для повернення з `non-void` і `void`-функцій та завдань. Може бути застосований у будь-який момент виконання функції або завдання.

Лістинг 17.17. Використання оператора `return` для виходу з підпрограм.

```

task add_up_to_max (input [ 5:0] max,
                    output [63:0] result);
    result = 1;
    if (max == 0) return;           // Вихід із завдання
    for (int i=1; i<=63; i=i+1) begin
        result = result + result;
        if (i == max) return;      // Вихід із завдання
    end
endtask

function automatic int log2 (input int n);
    if (n <=1) return 1;           // Вихід з функції
    log2 = 0;
    while (n > 1) begin
        n = n/2;
        log2++;
    end
    return log2;
endfunction

```

В SystemVerilog дозволяється використання оператора `return` для дострокового виходу як із завдань, так і з функцій, у той час як оператор Verilog `disable` призначений тільки для завдань і не може з'являтися у функціях.

У завданнях і `void`-функціях оператор `return` не може мати виразу для повернення, для `non-void`-функцій наявність такого виразу обов'язкова.

Оператори переходу `break`, `continue` і `return` є синтезованими конструкціями і обробляються так само, як `disable` в Verilog.

17.4.1. Мітки операторів.

SystemVerilog дозволяє використовувати мітки для операторів, що спрощує їхню ідентифікацію (лістинг 17.18). Синтаксис

```
<label> : <statement>
```

Лістинг 17.18. Приклад створення міток операторів.

```

always_comb begin : decode_block
    decoder : case (opcode)
        2'b00:
            outer_loop: for (int i=0; i<=15; i++)

```

```

                                inner_loop: for (int j=0; j<=15; j++)
                                    //...
                                ... // Декодування інших значень opcode
                                endcase
end : decode_block

```

17.4.2. Модифікатори unique і priority.

SystemVerilog пропонує спеціальні модифікатори unique і priority для операторів case, casex і casez:

```

unique case (<case_expression>)
    ... // Мітки оператора case
endcase
priority case (<case_expression>)
    ... // Мітки оператора case
endcase

```

Оператор unique case вказує, що тільки одне значення вибору може й повинне збігатися з виразом case. У цьому операторі не враховується пріоритет для вибору операцій:

```

always_comb
    unique case (opcode)
        2'b00: y = a + b;
        2'b01: y = a - b;
        2'b10: y = a * b;
        2'b11: y = a / b;
    endcase

```

Оператор priority case вказує, що як мінімум одне значення вибору повинне збігатися з виразом case, у разі збігання більш ніж одного виразу буде обраний перший збіг.

Модифікатор unique if...else означає, що порядок умов не має значення:

```

logic [2:0] sel;
always_comb begin
    unique if (sel == 3'b001) mux_out = a;
    else if (sel == 3'b010) mux_out = b;
    else if (sel == 3'b100) mux_out = c;
end

```

Модифікатор priority if...else позначає, що порядок умов важливий:

```

always_comb begin
    priority if (irq0) irq = 4'b0001;
    else if (irq1) irq = 4'b0010;
    else if (irq2) irq = 4'b0100;
    else if (irq3) irq = 4'b1000;
end

```


17.5. Значення часових параметрів

Поряд з директивою компілятора 'timescale для визначення параметрів часу в SystemVerilog можуть бути використані декларації timeunit і timeprecision. Однак якщо використовуються директиви, то їх необхідно вказувати в кожному модулі, що містить опис затримок. SystemVerilog дозволяє описувати затримки у форматі 0.1ns або 20ps (лістинг 17.19). Для роботи з часовими параметрами можна використовувати системні завдання Verilog: \$timeformat і \$realtime.

Лістинг 17.19. Символьне подання часу й \$timeformat.

```
module timing;
    timeunit 1ns;
    timeprecision 1ps;
    initial begin
        $timeformat(-9, 3, "ns", 8);
        #1 $display("@%t", $realtime);    // @1.000ns
        #2ns $display("@%t", $realtime);  // @3.000ns
        #0.1ns $display("@%t", $realtime); // @3.100ns
        #41ps $display("@%t", $realtime); // @3.141ns
    end
endmodule
```

17.6. Finite State Machines

Приклад з лістинга 17.20 представляє синтезовану поведінкову модель автомата, що використовує три процедурних блоки для обчислення наступного стану, значення вихідної функції й регістра стану. Приклад представляє простий контролер світлофора. Три стани автомата описані за допомогою змінної типу перелічення.

Лістинг 17.20. SystemVerilog-модель керуючого автомата.

```
module traffic_light (output logic green_light, yellow_light, red_light,
    input sensor,
    input [15:0] green_downcnt, yellow_downcnt,
    input clock, reset);
    // Використання типу перелічення для змінних станів
    enum {RED, GREEN, YELLOW} State, Next;
    always_ff @(posedge clock, negedge reset)
        if (!reset) State <= RED;           // Скидання в стан RED
        else State <= Next;
    always_comb begin: set_next_state
        Next = State;                       // Значення за замовчуванням
        unique case (State)
            RED: if (sensor) Next = GREEN;
            GREEN: if (green_downcnt == 0) Next = YELLOW;
            YELLOW: if (yellow_downcnt == 0) Next = RED;
        endcase
    end: set_next_state
    always_comb begin: set_outputs
```

```

{green_light, yellow_light, red_light} = 3'b000;
unique case (State)
    RED: red_light = 1'b1;
    GREEN: green_light = 1'b1;
    YELLOW: yellow_light = 1'b1;
endcase
end: set_outputs
endmodule

```

17.7. System Verilog FIFO example

Представлено приклад опису досить популярної синхронної пам'яті FIFO розміром у чотири 16-бітових слова й найпростішого testbench, орієнтованого на перевірку функціональності. FIFO, по суті, – це циклічна черга, інтерфейс якої представлений на рис. 17.1.

17.7.1. System Verilog-модель пристрою.

Пристрій містить показники для точок запису (регістр head) і читання (регістр tail) даних (лістинг 17.21). Лічильник count являє собою кількість інформації, записаної в пам'ять FIFO. Сигнали empty (порожня пам'ять) і full (пам'ять заповнена) формуються на основі даних лічильника counts. Скидання стану пам'яті в початковий, коли всі регістри дорівнюють нулю, виконується сигналом rstp=1. Вхід din і вихід dout даних описаний із застосуванням типу даних packet_t. Цей же тип даних може бути використаний при описі fifomem масиву пам'яті. Запис у пам'ять виконується по фронту синхросигналу, якщо writep=1. Процедура читання даних управляється сигналом readp.

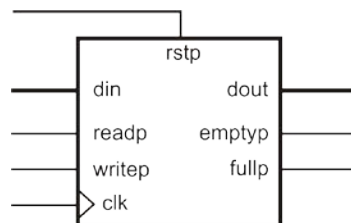


Рис.17.1. Інтерфейс пам'яті FIFO

Лістинг 17.21. SystemVerilog-модель синхронної пам'яті FIFO.

```

typedef struct packed {
    int src;
    int dst;
    logic [31:0] data;
} packet_t;

module fifo (
    input clk, input rstp,
    input packet_t din,
    input readp, writep,
    output packet_t dout,
    output logic emptyp,
    output logic fullp );

```

```

int i;
parameter DEPTH = 2, // 2-бітовий параметр, що задає розмір пам'яті,
// тобто 4 слова FIFO
MAX_COUNT = (1<<DEPTH); // Формування максимальної адреси FIFO
reg [(DEPTH-1):0] tail; // Показчик на точку читання
reg [(DEPTH-1):0] head; // Показчик на точку запису
reg [(DEPTH):0] count; // Лічильник розміру пам'яті
packet_t fifomem[0:MAX_COUNT]; // Масив пам'яті
always @(posedge clk)
    if (rstp == 1)
        dout <= '{default:0};
    else
        dout <= fifomem[tail];
// Відновлення пам'яті FIFO
always @(posedge clk)
    if (rstp)
        fifomem = '{default:0};
    else if (writep == 1'b1 && fullp == 1'b0)
        fifomem[head] <= din;
// Відновлення показчика точки запису head
always @(posedge clk)
    if (rstp == 1'b1)
        head <= 0;
    else if (writep == 1'b1 && fullp == 1'b0)
        head <= head + 1; // WRITE
// Відновлення регістра читання tail
always @(posedge clk)
    if (rstp == 1'b1)
        tail <= 0;
    else if (readp == 1'b1 && empty == 1'b0)
        tail <= tail + 1; // READ
// Відновлення лічильника кількості даних count
always @(posedge clk)
    if (rstp == 1'b1)
        count <= 0;
    else
        case ({readp, writep})
            2'b00: count <= count;
            2'b01: // Запис даних
                if (!fullp) count <= count + 1;
            2'b10: // Читання даних
                if (!empty) count <= count - 1;
            2'b11: // Одночасне виконання операцій // читання й запису,
                // лічильник count не змінюється
                count <= count;
        endcase
always @(count)
    if (count == 0)

```

```

        emptyp = 1'b1;
    else
        emptyp = 1'b0;
always @(count)
    if (count < MAX_COUNT)
        fullp = 1'b0;
    else
        fullp = 1'b1;
endmodule

```

17.7.2. System Verilog FIFO testbench.

Лістинг 17.22 являє собою testbench для верифікації поведінки FIFO. Він містить два завдання: читання (read_word) і запису (write_word) даних в пам'ять, за допомогою яких реалізовані два послідовно виконуваних тести - test1 і test2.

Test2 моделює нормальну роботу FIFO за допомогою двох паралельних потоків, що виконують читання і запис. Потік Writer спостерігає за значенням сигналу fullp і випадково формує вхідні значення, а також затримку між подачею цих значень. Крім того, він фіксує кількість записаних в пам'ять даних. Потік Reader спостерігає сигнал emptyp і виконує зчитування даних з випадковим інтервалом. Значення прапорів empty/full змінюються залежно від затримки, формованої випадково.

Лістинг 17.22. Testbench для верифікації поведінки FIFO.

```

module test_fifo;
    reg clk, rstp;
    packet_t packet_in, packet_out;
    reg readp, reg writep;
    wire emptyp, fullp;
    parameter depth = 2;
    fifo #(depth+1) U1 (.clk (clk), .rstp (rstp), .din (packet_in), .readp (readp),
        .writep (writep), .dout (packet_out), .emptyp (emptyp), .fullp (fullp));
    task read_word;
        @(negedge clk);
        readp = 1;
        @(posedge clk) #5;
        $display ("Read %h from FIFO", packet_out.src);
        readp = 0;
    endtask
    task write_word(byte src, dst, logic [3:0] data);
        @(negedge clk);
        packet_in = '{src,dst,data};
        writep = 1;
        @(posedge clk);
        $display ("Write %h to FIFO", src);
    endtask
endmodule

```

```

        #5;
        packet_in = '{default:'z';
        writep = 0;
    endtask
    initial begin
        clk = 0;
        forever #10 clk = ~clk;
    end
    initial begin
        test1;
        test2;
        @(posedge clk);
        $finish;
    end
    task test1;                                     // Тест1
        begin
            packet_in = '{default:'z';
            writep = 0;
            readp = 0;
            rstp = 1;                                // Скидання
            #50;
            rstp = 0;
            #50;
            write_word (0,1,16'h1111);               // Запис трьох значень
            write_word (1,2,16'h2222);
            write_word (3,4,16'h3333);
            read_word;                                // Читання двох значень
            read_word;
            write_word (5,6,16'h4444);               // Запис значення ге
            repeat (6) read_word;                    // Читання пакета значень
            write_word (0,1,16'h0001);               // Запис пакета значень
            write_word (0,1,16'h0002);
            write_word (0,1,16'h0003);
            write_word (0,1,16'h0004);
            write_word (0,1,16'h0005);
            write_word (0,1,16'h0006);
            write_word (0,1,16'h0007);
            write_word (0,1,16'h0008);
            repeat (8) read_word;                    // Читання пакета значень
            $display ("Done TEST1.");
        end
    endtask
    task test2;                                     // Тест2
        reg [15:0] writer_counter;
        begin
            packet_in = '{default:'z';
            writer_counter = 16'h0001;
            writep = 0;

```

```

readp = 0;
// Скидання
rstp = 1;
#50 rstp = 0;
#50;
fork
    begin: Writer                // Запис значень в пам'ять
    repeat (500) begin
        @(negedge clk);
        if (fullp == 1'b0) begin
            write_word ($random,$random,$random);
            #5 writer_counter = writer_counter + 1;
        end
        else $display ("WRITER is waiting..");
        // Псевдовипадкове значення затримки від 0ns до 100ns
        #(50 + ($random % 50));
    end
    $display ("Done with WRITER fork..");
    #200 $finish;
end
begin: Reader                // Читання інформації
    forever begin
        @(negedge clk);
        if (empty == 1'b0)
            read_word;
        else
            $display ("READER is waiting..");
        // Значення затримки від 0ns до 100ns, що формується випадково.
        #(50 + ($random % 50));
    end
end
join
end
endtask
always @(fullp)
    $display ("fullp = %0b", fullp);
always @(empty)
    $display ("empty = %0b", empty);
always @(U1.head)
    $display ("head = %0h", U1.head);
always @(U1.tail)
    $display ("tail = %0h", U1.tail);
endmodule

```

На рис. 17.2 представлені часові діаграми тестування й верифікації основних режимів пам'яті шляхом візуалізації її змінних і параметрів.

17.8. Контрольні запитання

1. Що означає такий вираз?

```
logic [7:0] opcode;
```

...

```
if (opcode ==? 8'b11011???)...
```

2. Яке значення отримає змінна у, якщо a=1001?

```
assign y = (a ==? 4'b1???1);
```

3. Яке значення отримає змінна у, якщо a=1111?

```
assign y = (a ==? 4'b1???1);
```

4. Яке значення отримає змінна у, якщо a=0001?

```
assign y = (a ==? 4'b1???1);
```

5. Яке значення отримає змінна у, якщо a=1???1?

```
assign y = (a ==? 4'b1???1);
```

6. Яку помилку містить такий вираз?

a) always_comb @(clk)

б) always_ff @(clk)

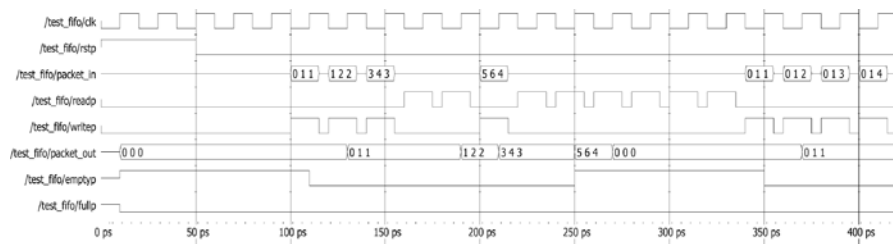


Рис. 17.2. Часові діаграми моделювання пам'яті

18. SYSTEM VERILOG ІЄРАРХІЯ ПРОЕКТУ

18.1. Прототип модуля

Прототип модуля створюється за допомогою ключового слова `extern`, за яким іде декларація модуля і його портів.

// Прототип модуля з використанням стилю Verilog-1995

```
extern module counter (cnt, d, clock, reset);
```

// Прототип модуля з використанням стилю Verilog-2001

```
extern module counter #(parameter N = 15)
```

```
    (output logic [N:0] cnt,
```

```
    input wire [N:0] d,
```

```
    input wire clock,    load, reset);
```

Якщо прототип модуля існує, при його визначенні можна не повторювати оголошення портів, а використовувати груповий символ `(.*)` (лістинг 18.1).

Лістинг 18.1. Створення моделі модуля при використанні його прототипу.

```
extern module counter #(parameter N = 15)
```

```
    (output logic [N:0] cnt,
```

```
    input wire [N:0] d,
```

```
    input wire clock, load, reset);
```

```
module counter ( .* );
```

```
    always @(posedge clock, negedge reset) begin
```

```
        if (!reset) cnt <= 0;
```

```
        else if (load) cnt <= d;
```

```
        else cnt <= cnt + 1;
```

```
    end
```

```
endmodule
```

Ім'я модуля може бути зазначене наприкінці коду:

```
endmodule : <module_name>
```

В SystemVerilog з'являється можливість створювати вкладені модулі, які визначаються всередині іншого модуля (лістинг 18.2, рис. 18.1).

Лістинг 18.2. Ієрархічна модель із вкладеними модулями.

```
module chip
```

```
    (input wire clock); // Проект верхнього рівня
```

```
    dreg i1 (clock);
```

```
    ip_core i2 (clock);
```

```
endmodule: chip
```

```
module dreg
```

```
    (input wire clock); // Декларація глобального модуля
```

```
    ...
```

```
endmodule: dreg
```

```
module ip_core
```

```
    (input wire clock); // Декларація глобального модуля
```

```
    sub1 u1 (...);
```

```
    sub2 u2 (...);
```

```
    module sub1(...); // Декларація вкладеного модуля
```



```

...
endmodule: sub1
module sub2(...); // Декларація вкладеного модуля
...
endmodule: sub2
endmodule: ip_core

```

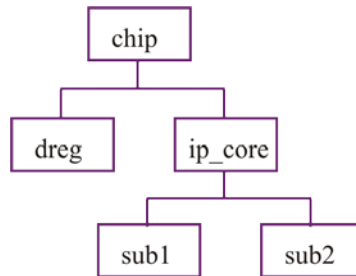


Рис. 18.1. Ієрархічна модель

Вкладені модулі можуть бути розташовані в різних файлах. Для включення їх у модель може бути використана директива `'include` (лістинг 18.3). Це спрощує аналіз і роботу з такими моделями.

Лістинг 18.3. Використання директиви `'include` для вкладених модулів.

```

module ip_core (input logic clock);
...
`include sub1.v // Вкладений модуль sub1
`include sub2.v // Вкладений модуль sub2
...
endmodule
module sub1(...); // Декларація модуля у файлі sub1.v
...
endmodule
module sub2(...); // Декларація модуля у файлі sub2.v
...
endmodule

```

Після компіляції вкладений файл буде включений у батьківський модуль (лістинг 18.4). Вкладені модулі можуть бути використані усередині модуля, у якому вони визначені, і вниз від нього по ієрархії (рис. 18.2).

Лістинг 18.4. Ієрархічна модель із вкладеними модулями.

```

module ip_core (input clock);
    sub1 u1 (...); // Копія вкладеного модуля sub1
    module sub1 (...); // Визначення вкладеного модуля
    ...
    sub2 u2 ();
endmodule: sub1
module sub2; // Визначення вкладеного модуля
// модуль sub2 не має портів, але може звертатися
// до коду батьківського модуля (ip_core)
...

```

```

        sub3 u3 (...);
    endmodule: sub2
    module sub3 (...);          // Визначення вкладеного модуля
    ...
    endmodule: sub3
endmodule: ip_core

```

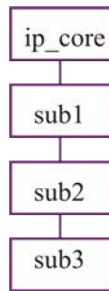


Рис. 18.2. Ієрархічна модель із вкладеними компонентами

18.2. З'єднання портів

В Verilog використовувалося з'єднання сигналів по позиції або по імені. SystemVerilog пропонує три спрощені форми опису зв'язків портів:

- 1) .name ("dot-name") з'єднання портів;
- 2) .* ("dot-star") з'єднання портів;
- 3) за допомогою інтерфейсів.

SystemVerilog виконує з'єднання портів з сигналами, що збігаються по імені, спрощуючи вирази Verilog .data(data) (лістинг 18.5, а) до .data в SystemVerilog (лістинг 18.5, б). З'єднання .* (лістинг 18.5, в) зв'язує всі порти з відповідними по імені сигналами.

Лістинг 18.5. Приклади з'єднань портів.

а) //Verilog стиль

```

// Копія модуля з іменним з'єднанням портів
pc_stack pcs (
    .program_counter(program_counter),
    .program_address(program_address),
    .clk(clk),
    .reset(reset),
    .instruct_reg(instruct_reg),
    .data_bus(data_bus),
    .status_reg(status_reg));
prom prom (.dout(program_data),
    .clk(clk), .address(program_address));

```

б) // SystemVerilog стиль з використанням.name

```

// Копія модуля із з'єднанням портів.name
pc_stack pcs (.program_counter, .program_address,
    .clk, .reset, .instruct_reg, .data_bus, .status_reg);
prom prom (

```

```
.dout(program_data),
.clk, .address(program_address));
```

```
в) // SystemVerilog стиль із використанням .*
// Копія модуля із з'єднанням портів .*
pc_stack pcs (. *);
prom prom (. *, .dout(program_data),
.address(program_address));
```

18.3. Псевдоніми ланцюгів

Псевдоніми ланцюгів дозволяють використовувати два різних імені для звертання до однієї лінії:

```
wire clock;
wire clk;
alias clk = clock;
```

У прикладі множинних псевдонімів

```
wire reset, rst, reset, rst;
alias rst = reset;
alias reset = rst;
alias reset = rst;
```

всі імена вказують на одну лінію. Не важливо, у якому порядку виконується зв'язок імен, оскільки оператор `alias` не є оператором присвоєння. Псевдоніми можуть бути використані тільки для ланцюгів і повинні бути одного типу:

```
wire [3:0][7:0] n2;
alias n2 = n1; // n1 і n2 мають розмір 32 біта
wire [39:0] d_in;
wire [7:0] crc;
wire [31:0] data;
alias data = d_in[31:0]; // Ланцюг розміром в 32 біта
alias crc = d_in[39:32]; // Ланцюг розміром в 8 біт
```

Використання псевдонімів з конструкціями `.name i .*` дозволяє значно спростити запис зв'язків між портами в ієрархічних проектах.

Лістинг 18.6 представляє структурну SystemVerilog-модель пристрою (рис. 18.3) з використанням символу «`.*`» зв'язку портів без псевдонімів. Цей же приклад із застосуванням псевдонімів наведений у лістингу 18.7.

Лістинг 18.6. Використання з'єднання портів SystemVerilog `.*` без псевдонімів.

```
module chip
    (input wire master_clock,
     input wire master_reset, ...);
    wire [31:0] address, new_address, next_address;
    ROM i1 ( . *, // Позначає .address(address)
            .data(new_address),
            .clk(master_clock) );
    program_count i2 ( . *, // Позначає .next_address(next_address)
```

```

        .jump_address(new_address),
        .clock(master_clock),
        .reset_n(master_reset) );
address_reg i3 ( . *,      // Не відповідає ніяким з'єднанням
        .next_addr(next_address),
        .current_addr(address),
        .clk(master_clock), .rst(master_reset) );
endmodule
module ROM (output wire [31:0] data,
        input wire [31:0] address,
        input wire clk);
...
endmodule
module program_count (output logic [31:0] next_address,
        input wire [31:0] jump_address,
        input wire clock, reset_n);
...
endmodule
module address_reg (output wire [31:0] current_addr,
        input wire [31:0] next_addr,
        input wire clk, rst);
...
endmodule

```

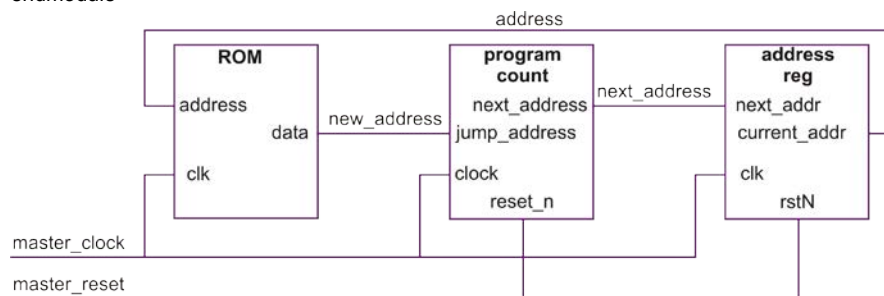


Рис 18.3. Структурна схема пристрою

Лістинг 18.7. Використання з'єднання портів SystemVerilog `.*` із застосуванням псевдонімів.

```

module chip
    (input wire master_clock,
     input wire master_reset, ...);
    wire [31:0] address, data, new_address, jump_address,
                next_address, next_addr, current_addr;
    alias clk = clock = master_clock;
    alias rst = reset_n = master_reset;
    alias data = new_address = jump_address;
    alias next_address = next_addr;
    alias current_addr = address;
    ROM i1 ( . * );

```

```

        program_count i2 ( .* );
        address_reg i3 ( .* );
    endmodule
    module ROM (output wire [31:0] data,
        input wire [31:0] address,
        input wire clk);
        ...
    endmodule
    module program_count (output logic [31:0] next_address,
        input wire [31:0] new_count,
        input wire clock, reset_n);
        ...
    endmodule
    module address_reg (output wire [31:0] address,
        input wire [31:0] next_address,
        input wire clk, rst);
        ...
    endmodule

```

18.4. Передача даних через порти модуля

В SystemVerilog зняли обмеження на використання типів даних при зв'язуванні із вхідними та вихідними портами в структурних моделях (лістинг 18.8). Через порт можуть бути передані: 1) будь-які скалярні дані; 2) запаковані і незапаковані масиви будь-якої розмірності; 3) структури та об'єднання. Лістинг 18.8. Передача структури і масиву через порти.

```

typedef struct packed {
    logic [ 3:0] opcode;
    logic [15:0] operand;
} instruction_t;
module decoder
    (output logic [23:0] microcode,
    input instruction_t instruction,
    input logic [23:0] LUT [0:(2**20)-1] );
    ... // Інші оператори моделі
endmodule
module DSP
    (input logic clock, reset,
    input logic [ 3:0] opcode,
    input logic [15:0] operand,
    output logic [23:0] data );
    logic [23:0] LUT [0:(2**20)-1]; // Визначення таблиці
    instruction_t instruction;
    logic [23:0] microcode;
    decoder i1 (microcode, instruction, LUT);
    ... // Оператори моделі
endmodule

```

18.5. Посилальні порти (Reference ports)

В SystemVerilog доданий четвертий тип портів - посилальний, який позначається як `ref`. У цьому випадку виконується передача ієрархічного посилання на порт замість самого значення (лістинг 18.9). Посилальні порти не синтезуються.

Лістинг 18.9. Передача посилання на масив через `ref`-порт модуля.

```
typedef struct packed {
    logic [ 3:0] opcode;
    logic [15:0] operand;
} instruction_t;
module decoder (output logic [23:0] microcode,
    input instruction_t instruction,
    ref logic [23:0] LUT [0:(2**20)-1] );
    ... // Оператори моделі
endmodule
module DSP (input logic clock, reset,
    input logic [ 3:0] opcode,
    input logic [15:0] operand,
    output logic [23:0] data );
    logic [23:0] LUT [0:(2**20)-1]; // Визначення таблиці
    instruction_t instruction;
    logic [23:0] microcode;
    decoder i1 (microcode, instruction, LUT);
    ... // Оператори моделі
endmodule

// Два різних стилі опису портів:
// Verilog-2001 стиль
module accum (inout wire [31:0] data,
    output reg [31:0] result,
    output reg co,
    input [31:0] a, b,
    input tri1 ci );
    ...
endmodule

// SystemVerilog стиль
module accum (wire [31:0] data,
    output reg [31:0] result, reg co,
    input [31:0] a, b, tri1 ci );
    ...
endmodule
```

Якщо напрямок порту не зазначено, то в SystemVerilog за замовчуванням використовується тип порту `inout`.

18.6. Параметризовані типи даних

Verilog дозволяє використовувати параметри та локальні параметри для на-строювання розмірів типів даних. SystemVerilog пропонує параметризовані

типи даних, які оголошуються за допомогою пари ключових слів `parameter type`. Якщо в Verilog параметри використовувалися для визначення розміру векторів, то в SystemVerilog параметризовані типи даних використовуються для зміни типів даних.

Параметризовані типи є синтезованими, якщо використовують синтезовані типи даних. У прикладі (лістинг 18.10) тип даних суматора `ADDERTYPE` є параметризованим і за замовчуванням визначається як `shortint`. Модуль `big_chip` містить три копії суматора. Копія `i1` використовує тип даних суматора, заданий за замовчуванням, створюючи 16-розрядний знаковий суматор. Копія `i2` змінює тип змінної `int`, формуючи 32-розрядний знаковий суматор. Копія `i3` перетворює тип змінної в `int unsigned`, формуючи 32-бітовий беззнаковий суматор.

Лістинг 18.10. Поліморфний суматор з параметризованими даними.

```
module adder
    #(parameter type ADDERTYPE = shortint)
    (input ADDERTYPE a, b,      // перевизначений тип
     output ADDERTYPE sum,    // перевизначений тип
     output logic carry);
    ADDERTYPE temp;           // локальна змінна перевизначеного типу
    ... // Функціональність суматора
endmodule

module big_chip( ... );
    shortint a, b, r1;
    int c, d, r2;
    int unsigned e, f, r3;
    wire carry1, carry2, carry3;
    // 16-бітовий беззнаковий суматор
    adder i1 (a, b, r1, carry1);
    // 32-бітовий знаковий суматор
    adder #(.ADDERTYPE(int)) i2 (c, d, r2, carry2);
    // 32-бітовий беззнаковий суматор
    adder #(.ADDERTYPE(int unsigned)) i3 (e, f, r3, carry3);
endmodule
```

18.7. Контрольні запитання

1. Які методи зв'язку портів, відмінні від Verilog, введені в SystemVerilog?
2. Створити прототипи компонентів і структурні моделі пристрою з використанням стилів `i.*` (рис. 18.4).
3. Визначити псевдоніми для `reset_n` і `clr_n` і структурні SystemVerilog-моделі пристрою (рис. 18.5). Для компонентів записати тільки декларацію інтерфейсів модулів.

4. Створити модель пристрою множення з параметризованими типами даних операндів. За замовчуванням пристрій повинен працювати 8-бітовим вектором типу `logic`. Створити модуль верхнього рівня, що підключає копію компонента пристрою множення, у якому тип даних операндів перевизначається на `int`.

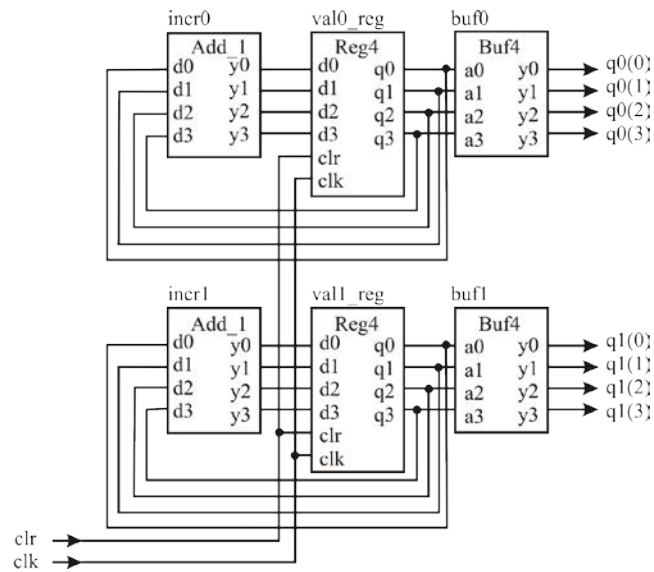


Рис 18.4. Приклад схемы пристрою

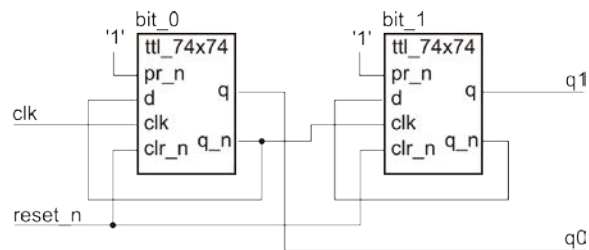


Рис 18.5. Приклад схемы пристрою

19. ВИКОРИСТАННЯ ІНТЕРФЕЙСІВ

19.1. Концепція інтерфейсу

Для виконання зв'язків між модулями Verilog використовує порти. Для більших проектів запис такого способу з'єднань стає дуже складним. Приклад (рис. 19.1, лістинг 19.1) представляє п'ять блоків, з'єднаних разом за допомогою шини `main_bus`, крім цього, присутні додаткові прямі зв'язки між окремими блоками.

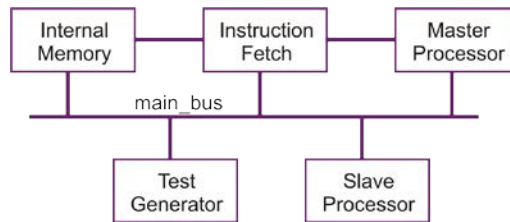


Рис. 19.1. Структурна схема пристрою із шинною архітектурою

Лістинг 19.1. Структурна Verilog-модель пристрою.

```
// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i Модуль верхнього рівня ---i ---i ---i ---i ---i ---i ---i
module top (input wire clock, reset, test_mode);
    wire [15:0] data, address, program_address, jump_address;
    wire [ 7:0] instruction, next_instruction;
    wire [ 3:0] slave_instruction;
    wire slave_request, slave_ready;
    wire bus_request, bus_grant;
    wire mem_read, mem_write;
    wire data_ready;
    processor proc1 (
        // Порти для зв'язку із шиною main_bus
        .data(data), .address(address),
        .slave_instruction(slave_instruction),
        .slave_request(slave_request),
        .bus_grant(bus_grant),
        .mem_read(mem_read),
        .mem_write(mem_write),
        .bus_request(bus_request),
        .slave_ready(slave_ready),
        // Інші порти
        .jump_address(jump_address),
        .instruction(instruction),
        .clock(clock), .reset(reset), .test_mode(test_mode) );
    slave slave1 (
        // Порти для зв'язку із шиною main_bus
        .data(data), .address(address),
        .bus_request(bus_request),
```

```

        .slave_ready(slave_ready),
        .mem_read(mem_read),
        .mem_write(mem_write),
        .slave_instruction(slave_instruction),
        .slave_request(slave_request),
        .bus_grant(bus_grant),
        .data_ready(data_ready),
        // Інші порти
        .clock(clock), .reset(reset) );
dual_port_ram ram (
    // Порти для зв'язку із шиною main_bus
    .data(data), .data_ready(data_ready),
    .address(address),
    .mem_read(mem_read),
    .mem_write(mem_write),
    // Інші порти
    .program_address(program_address),
    .data_b(next_instruction));
test_generator test_gen(
    // Порти для зв'язку із шиною main_bus
    .data(data), .address(address),
    .mem_read(mem_read),
    .mem_write(mem_write),
    // Інші порти
    .clock(clock), .reset(reset), .test_mode(test_mode));
instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .reset(reset));
endmodule
// ---і---і---і---і---і---і---і---і---іВизначення модулів ---і---і---і---і---і---і-/
module processor (
    // Порти для зв'язку із шиною main_bus
    inout wire [15:0] data,
    output reg [15:0] address,
    output reg [ 3:0] slave_instruction,
    output reg slave_request,
    output reg bus_grant,
    output wire mem_read,
    output wire mem_write,
    input wire bus_request,
    input wire slave_ready,
    // Інші порти
    output reg [15:0] jump_address,
    input wire [ 7:0] instruction,

```

```

        input wire clock,
        input wire reset,
        input wire test_mode);
... // Функціональний код модуля
endmodule

module slave (
    // Порти для зв'язку із шиною main_bus
    inout wire [15:0] data,
    inout wire [15:0] address,
    output reg bus_request,
    output reg slave_ready,
    output wire mem_read,
    output wire mem_write,
    input wire [3:0] slave_instruction,
    input wire slave_request,
    input wire bus_grant,
    input wire data_ready,
    // Інші порти
    input wire clock,
    input wire reset);
... // Функціональний код модуля
endmodule

module dual_port_ram (
    // Порти для зв'язку із шиною main_bus
    inout wire [15:0] data,
    output wire data_ready,
    input wire [15:0] address,
    input tri0 mem_read,
    input tri0 mem_write,
    // Інші порти
    input wire [15:0] program_address,
    output reg [7:0] data_b);
... // Функціональний код модуля
endmodule

module test_generator (
    // Порти для зв'язку із шиною main_bus
    output wire [15:0] data,
    output reg [15:0] address,
    output reg mem_read,
    output reg mem_write,
    // Інші порти
    input wire clock,
    input wire reset,
    input wire test_mode);
... // Функціональний код модуля
endmodule

module instruction_reg (
    output reg [15:0] program_address,

```

```

        output reg [ 7:0] instruction,
        input wire [15:0] jump_address,
        input wire [ 7:0] next_instruction,
        input wire clock,
        input wire reset);
    ... // Функціональний код модуля
endmodule

```

До недоліків портів у більших Verilog-проектах можна віднести:

- 1) декларації дублюються в декількох модулях;
- 2) комунікаційні протоколи дублюються в декількох модулях, що підвищує можливість виникнення помилки в деклараціях різних модулів;
- 3) зміна в специфікації проекту вимагає виконання модифікації в декількох модулях.

SystemVerilog пропонує нову потужну конструкцію для опису портів - інтерфейс. Інтерфейс дозволяє групувати сигнали разом, при цьому кожний модуль, що використовує сигнали з інтерфейсу, розглядає його як єдиний порт (лістинг 19.2). В інтерфейсі `main_bus` описуються сигнали пристрою, а в модулі верхнього рівня `top` створюється копія інтерфейсу `bus`, що виконує зв'язок компонентів. Крім цього, декларація інтерфейсу вказується в списку портів кожного модуля.

Лістинг 19.2. Виконання зв'язків модулів за допомогою SystemVerilog.

```

// ---i---i---i---i---i---i---i---i---i---i--- Декларація інтерфейсу ---i---i---i---i---i---i---i---i---i---i---
interface main_bus;
    wire [15:0] data;
    wire [15:0] address;
    logic [ 7:0] slave_instruction;
    logic slave_request;
    logic bus_grant;
    logic bus_request;
    logic slave_ready;
    logic data_ready;
    logic mem_read;
    logic mem_write;
endinterface

// ---i---i---i---i---i---i---i---i---i---i--- Модуль верхнього рівня ---i---i---i---i---i---i---i---i---i---i---
module top (input logic clock, reset, test_mode);
    logic [15:0] program_address, jump_address;
    logic [ 7:0] instruction, next_instruction;
    main_bus bus ( ); // копія інтерфейсу з ім'ям bus
    processor proc1 ( // Порти шини main_bus
        .bus(bus), // Підключення інтерфейсу
        // Інші порти
        .jump_address(jump_address),
        .instruction(instruction),

```

```

        .clock(clock), .reset(reset), .test_mode(test_mode));
slave slave1 (
    // Порти шини main_bus
    .bus(bus), /      / Підключення інтерфейсу
    // Інші порти
    .clock(clock), .reset(reset));
dual_port_ram ram (
    // Порти шини main_bus
    .bus(bus),        // Підключення інтерфейсу
    // Інші порти
    .program_address(program_address), .data_b(next_instruction));
test_generator test_gen(
    // Порти шини main_bus
    .bus(bus),        // Підключення інтерфейсу
    // Інші порти
    .clock(clock), .reset(reset), .test_mode(test_mode));
instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .reset(reset));
endmodule

// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i Визначення модулів ---i ---i ---i ---i ---i ---i ---i ---i
module processor (
    // Порти інтерфейсу main_bus
    main_bus bus,      // Порти інтерфейсу
    // Інші порти
    output logic [15:0] jump_address,
    input logic [ 7:0] instruction,
    input logic clock,
    input logic reset,
    input logic test_mode);
... // Функціональний код модуля
endmodule
module slave (
    // Порти інтерфейсу main_bus
    main_bus bus, // Порти інтерфейсу
    // other ports
    input logic clock,
    input logic reset);
... // Функціональний код модуля
endmodule
module dual_port_ram (
    // Порти інтерфейсу main_bus
    main_bus bus, // Порти інтерфейсу
    // other ports
    input logic [15:0] program_address,
    output logic [ 7:0] data_b);

```

```

... // Функціональний код модуля
endmodule
module test_generator (
    // Порти інтерфейсу main_bus
    main_bus bus, // Порти інтерфейсу
    // other ports
    input logic clock,
    input logic reset,
    input logic test_mode);
... // Функціональний код модуля
endmodule
module instruction_reg (
    output logic [15:0] program_address,
    output logic [7:0] instruction,
    input logic [15:0] jump_address,
    input logic [7:0] next_instruction,
    input logic clock,
    input logic reset);
    ... // Функціональний код модуля
endmodule

```

Інтерфейс дозволяє: дискретні сигнали і порти можуть бути визначені в одному місці; комунікаційний протокол; виконувати перевірку й верифікацію підпрограм прямо в інтерфейсі. Інтерфейс може містити декларацію типів, завдань, функцій, процедурних блоків, програмних блоків і асерцій. Інтерфейс дозволяє визначати множинні його види. Наприклад, для кожного модуля, що підключається за допомогою інтерфейсу, сигнал `data_bus` може бути входом, виходом і двонаправленим портом. Переважніше використовувати тип `logic` для опису сигналів в інтерфейсі, це дозволить передавати їм значення за допомогою процедурних операторів.

Основні відмінності між інтерфейсом і модулем:

- 1) інтерфейс не може містити опис ієрархії проекту;
- 2) не можна використовувати модуль у списку портів;
- 3) інтерфейс може включати конструкцію `modport`.

Лістинг 19.3 представляє визначення інтерфейсу `main_bus`, що використовує три зовнішніх сигнали, які входять в інтерфейс: `clock`, `reset` і `test_mode`. Вони можуть бути підключені до кожного модуля через інтерфейс без виконання явного з'єднання сигналів у кожному модулі. Для опису зовнішніх сигналів (у прикладі це `clock`, `reset` і `test_mode`) інтерфейсу використовується той же синтаксис, що й для з'єднання сигналів у модулі.

Лістинг 19.3. Визначення інтерфейсу для `main_bus` із зовнішніми входами

```

// ---i---i---i---i---i---i---i---i---i---i---i---i- Декларація інтерфейсу ---i---i---i---i---i---i-
i-
interface main_bus (input logic clock, reset, test_mode);
    wire [15:0] data;
    wire [15:0] address;

```

```

        logic [ 7:0] slave_instruction;
        logic slave_request;
        logic bus_grant;
        logic bus_request;
        logic slave_ready;
        logic data_ready;
        logic mem_read;
        logic mem_write;
    endinterface
// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---iМодуль верхнього рівня ---i ---i---i-i-i-i-i-i-i-i-
module top (input logic clock, reset, test_mode);
    logic [15:0] program_address, jump_address;
    logic [ 7:0] instruction, next_instruction;
    main_bus bus ( .clock(clock), .reset(reset), .test_mode(test_mode)); // копія інтер-
фейсу
    processor proc1 ( // main_bus ports
        .bus(bus), // interface connection
        // other ports
        .jump_address(jump_address),
        .instruction(instruction));
    ...
    /*** remainder of netlist and module definitions are not ***/
    /*** listed - they are similar to example 10-2, but ***/
    /*** clock and reset do not need to be passed to each ***/
    /*** module instance as discrete ports. ***/

```

Спрощене з'єднання портів SystemVerilog за допомогою стилів `.name i .*` може бути використано і для інтерфейсу. Приклад з лістингу 19.4 може мати більш компактний вид, завдяки комбінації використання інтерфейсів і з'єднання портів `.*`.

Лістинг 19.4. Спрощення з'єднання портів.

```

// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i Декларація інтерфейсу ---i ---i ---i ---i---i-i-i-i-i-i-i-
i-
interface main_bus (input logic clock, reset, test_mode);
    wire [15:0] data;
    wire [15:0] address;
    logic [ 7:0] slave_instruction;
    logic slave_request;
    logic bus_grant;
    logic bus_request;
    logic slave_ready;
    logic data_ready;
    logic mem_read;
    logic mem_write;
endinterface
// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i ---iМодуль верхнього рівня ---i ---i---i-i-i-i-i-i-i-i-
module top (input logic clock, reset, test_mode);
    logic [15:0] program_address, jump_address;

```

```

        logic [ 7:0] instruction, next_instruction, data_b;
        main_bus bus ( .* );
        processor proc1 ( .* );
        slave slave1 ( .* );
        instruction_reg ir ( .* );
        test_generator test_gen ( .* );
        dual_port_ram ram ( .*, .data_b(next_instruction) );
    endmodule
    /*** remainder of netlist and module definitions are not ***/
    /*** listed - they are similar to example 10-2, but ***/
    /*** clock and reset do not need to be passed to each ***/
    /*** module instance as discrete ports. ***/

```

19.2. Опис портів за допомогою інтерфейсу та звертання до них

Опис портів модуля може бути виконано за допомогою ключового слова `interface` або зазначення конкретного типу інтерфейсу. Синтаксис:

```

module <module_name> (interface <port_name>);
або
module <module_name> (<interface_name> <port_name>);

```

Наприклад:

```

module CACHE (interface pins, // interface port
               input clock);
...
endmodule
interface chip_bus;
...
endinterface
module CACHE (chip_bus pins, // interface port
               input clock);
...
endmodule

```

Звертання до сигналів в інтерфейсі виконується за допомогою ієрархічного імені, у першій частині якого вказується ім'я копії інтерфейсу.

Синтаксис:

```

internal_interface_signal_name>.<port_name>

```

Сигнали інтерфейсу повинні отримувати значення за допомогою неблокуючих операторів.

У прикладі визначення інтерфейсу `main_bus` містить декларацію сигналів `clock` і `reset`. Модуль `slave` використовує інтерфейсний порт із `bus`. Модуль `slave` має доступ до змінної інтерфейсу `clock`, застосовуючи посилання `bus.clock`. Наприклад:

```

always @(posedge bus.clock, negedge bus.reset)
...

```


Лістинг 19.5 представляє фрагмент коду модуля `slave`. Модуль містить кілька звертань до сигналів інтерфейсу `main_bus`.

Лістинг 19.5. Звертання до сигналів інтерфейсу.

```
module slave (
    // Порти інтерфейсу main_bus
    main_bus bus
    // other ports);
    // internal signals
    logic [15:0] slave_data, slave_address;
    logic [15:0] operand_A, operand_B;
    logic mem_select, read, write;
    assign bus.address = mem_select? slave_address: 'z;
    assign bus.data = bus.slave_ready? slave_data: 'z;
    enum logic [4:0] {RESET = 5'b00001, START = 5'b00010,
        REQ_DATA = 5'b00100, EXECUTE = 5'b01000,
        DONE = 5'b10000} State, NextState;
    always_ff @(posedge bus.clock, negedge bus.reset) begin: FSM
        if (!bus.reset) State <= START;
        else State <= NextState;
    end
    always_comb begin : FSM_decode
        unique case (State)
            START: if (!bus.slave_request) begin
                bus.bus_request = 0;
                NextState = State;
            end
            else begin
                operand_A = bus.data;
                slave_address = bus.address;
                bus.bus_request = 1;
                NextState = REQ_DATA;
            end
            ... // decode other states
        endcase
    end: FSM_decode
endmodule
```

19.2.1. З'єднання інтерфейсів і портів.

Для модуля, порти якого не можуть бути змінені з використанням інтерфейсу, можна просто з'єднати сигнали інтерфейсу з окремими портами, як це зроблено для модуля `arb_port`:

Лістинг 19.6. З'єднання сигналів інтерфейсу.

```
module top;
    bit clk;
    always #5 clk = ~clk;
    arb_if arbif(clk); // створення копії інтерфейсу
```

```

arb_port a1 (.grant (arbif.grant),          // підключення компонента до
            .request (arbif.request), // інтерфейсу
            .reset (arbif.reset),
            .clk (arbif.clk));
test t1(arbif); // підключення модуля, що використовує інтерфейс
endmodule : top

```

19.3. Групування сигналів за допомогою modport

У попередніх прикладах застосовувалося з'єднання без зазначення напрямку сигналу в інтерфейсі. Однак той самий сигнал у різних модулях може відігравати різну роль. Наприклад, для одного він буде входом, а для іншого виходом. У цьому випадку такий порт усередині інтерфейсу оголошується як modport.

Конструкція modport дозволяє групувати сигнали та описувати їхній напрямок, що дає можливість виконувати додаткову перевірку при передачі інформації і виключати пов'язані із цим помилки.

Modport визначає напрямок порту, що відповідає модулю. Інтерфейс може містити будь-яке число декларацій modport, кожна описує один або кілька модулів, розглядаючи сигнали в інтерфейсі.

Лістинг 19.7. Визначення напрямку порту.

```

interface chip_bus (input logic clock, reset);
    logic interrupt_request, grant, ready;
    logic [31:0] address;
    wire [63:0] data;
    modport master (input interrupt_request,
                    input address, output grant, ready,
                    inout data, input clock, reset);
    modport slave (output interrupt_request,
                   output address, input grant, ready,
                   inout data, input clock, reset);
endinterface

```

У конструкції modport не вказують тип даних або розмір вектора. Ця інформація вказується при декларації сигналів інтерфейсу. Декларація modport описує тільки напрямок порту: input, output, inout або ref.

SystemVerilog пропонує два методи для зазначення застосовуваного виду modport:

- 1) в операторі реалізації копії компонента;
- 2) у декларації портів модуля при його визначенні.

Обидва стилі специфікації є синтезованими.

Перший стиль при створенні копії модуля та копії інтерфейсу з'єднується з портом копії модуля, при цьому modport може бути зазначений так:

```
<ім'я копії модуля>. <ім'я modport>
```

Наприклад:

```
chip_bus bus; // instance of an interface
primary i1 (bus.master); // use master modport
```

Приклад (лістинг 19.8) ілюструє з'єднання двох модулів разом з інтерфейсом `chip_bus`. Модуль `primary` з'єднується з інтерфейсом у режимі `master`, а модуль `secondary` - у режимі `slave`:

Лістинг 19.8. Вибір режиму `modport` у момент створення копії модуля.

```
interface chip_bus (input logic clock, reset);
    modport master (...);
    modport slave (...);
endinterface
module primary (interface pins);          // Абстрактний порт інтерфейсу
    ...
endmodule
module secondary (chip_bus pins);        // Визначений порт інтерфейсу
    ...
endmodule
module chip (input logic clock, reset);
    chip_bus bus (clock, reset); // Копія інтерфейсу
    primary i1 (bus.master);      // Використання режиму modport
    secondary i2 (bus.slave);     // Використання режиму slave
endmodule
```

Якщо `modport` вказується в декларації інтерфейсу модуля, то застосовується синтаксис:

```
<ім'я інтерфейсу>.<ім'я modport>
```

Наприклад, для лістинга 19.8:

```
module secondary (chip_bus.slave pins);
    ...
endmodule
```

У цьому випадку має бути вказано явне ім'я інтерфейсу. Тоді в операторі реалізації копії модуля наводиться тільки ім'я копії інтерфейсу без опису режиму `modport`. Приклад (лістинг 19.9) представляє запис режиму `modport` у декларації портів модуля.

Лістинг 19.9. Вибір режиму `modport` у момент опису портів модуля.

```
interface chip_bus (input logic clock, reset);
    modport master (...);
    modport slave (...);
endinterface
module primary (chip_bus.master pins);    // Використання режиму master
    ...
endmodule
module secondary (chip_bus.slave pins);   // Використання режиму slave
    ...
endmodule
```

```

module chip (input logic clock, reset);
    chip_bus bus (clock, reset);           // Копія інтерфейсу
    primary i1 (bus);                     // Використання режиму master
    secondary i2 (bus);                   // Використання режиму slave
endmodule

```

Таким чином, вибрати `modport` можна або для копії модуля, або при описі його портів, але ніколи одночасно (лістинг 19.10). В один момент допускається вказувати тип `modport` тільки в одній із цих двох конструкцій. Якщо напрями портів не описані, то за замовчуванням всі порти інтерфейсу мають режим `inout` і тип `ref`. Крім цього, в інтерфейсі можна оголошувати внутрішні сигнали, які є локальними для нього.

Лістинг 19.10. Використання інтерфейсу з конструкціями `modport`.

```

// ---i ---i ---i ---i ---i ---i ---i ---i ---i- Визначення інтерфейсу ---i ---i ---i ---i ---i ---i ---i-
interface main_bus (input logic clock, reset, test_mode);
    wire [15:0] data;
    wire [15:0] address;
    logic [ 7:0] slave_instruction;
    logic slave_request, bus_grant, bus_request;
    logic slave_ready, data_ready;
    logic mem_read, mem_write;
    modport master (inout data,
        output address,
        output slave_instruction, slave_request,
        output bus_grant,
        output mem_read, mem_write,
        input bus_request,
        input slave_ready, data_ready,
        input clock, reset,
        input test_mode
    );
    modport slave (inout data,
        inout address,
        output mem_read, mem_write,
        output bus_request, slave_ready,
        input slave_instruction, slave_request,
        input bus_grant,
        input data_ready,
        input clock, reset
    );
    modport mem (inout data,
        output data_ready,
        input address,
        input mem_read, mem_write
    );
endinterface
// ---i ---i ---i ---i ---i ---i ---i ---i ---i- Модуль верхнього рівня ---i ---i ---i ---i ---i ---i ---i-
module top (input logic clock, reset, test_mode);

```

```

logic [15:0] program_address, jump_address;
logic [ 7:0] instruction, next_instruction, data_b;
    main_bus bus ( . * );           // Копія інтерфейсу
    processor proc1 (.bus(bus.master), . * );
    slave slave1 (.bus(bus.slave), . * );
    instruction_reg ir ( . * );
    test_generator test_gen (.bus(bus), . * );
    dual_port_ram ram (.bus(bus.mem), . * ,
        .data_b(next_instruction) );
endmodule

```

19.4. Використання завдань і функцій в інтерфейсі

Інтерфейс може успадковувати всі деталі комунікаційного протоколу між блоками.

Наприклад, протокол `main_bus` з попереднього прикладу (див. лістинг 19.10) містить сигнал встановлення зв'язку між головним процесом (`master processor`) і підлеглим пристроєм (`slave`). Головний процесор в Verilog моделі повинен був містити код, який міг бути використаний для встановлення і видалення сигналу квитирування у відповідні моменти часу та відстеження його зміни для другорядного процесора.

Опис усього протоколу при створенні інтерфейсу модуля веде до можливості виникнення повторів коду.

19.4.1. Імпорт завдань і функцій.

Якщо інтерфейс зв'язується через `modport`, то для імпортування методів необхідно використовувати ключове слово `import`, при цьому вони застосовуються як частина визначення `modport`. Найпростіша форма імпортування завдання або функції просто описує їх ім'я. Синтаксис:

```
modport (import <task_function_name>);
```

Приклад використання:

```

modport in          (import Read,
                    import parity_gen,
                    input clock, reset );

```

Другий стиль декларації `import` описує повний прототип аргументів завдання або функції. Цей стиль вимагає, щоб ключові слова `task` або `function` слідували за ключовим словом `import`. Також потрібно, щоб за іменем завдання і функції була присутня безліч формальних аргументів. Базовий синтаксис:

```

modport (import task <task_name>(<task_formal_arguments> ) ;
modport (import function <function_name> (<formal_args> ) );

```

Наприклад:

```

modport in (
    import task Read (input [63:0] data, output [31:0] address),
    import function parity_gen (input [63:0] data),
    input clock, reset);

```

Виклик імпортованих методів:

```
<interface_port_name>.<method_name>
```

Конструкція `modport` пропонує спосіб використання різних методів і протоколів в одному інтерфейсі. Інтерфейс може містити різні методи, які застосовують різні протоколи або типи.

Наступний фрагмент коду ілюструє інтерфейс `math_bus`. В інтерфейсі визначені різні методи читання, що працюють із цілими та речовинними типами даних. Два модулі викликають `integer_math_unit` і `floating_point_unit`, обидва мають той самий інтерфейс шини. Кожний модуль має доступ до різних типів інформації.

Лістинг 19.11. Використання в інтерфейсі `modport` для вибору альтернативних методів.

```
interface math_bus (input logic clock, reset);
    int a_int, b_int, result_int;
    real a_real, b_real, result_real;
    ...
    task IntegerRead (output int a_int, b_int);
    ... // Встановлення зв'язку для захоплення значень a і b
    endtask
    task FloatingPointRead (output real a_real, b_real);
    ... // Встановлення зв'язку для захоплення значень a і b
    endtask
    modport int_io (import IntegerRead,
        input clock, reset,
        output result_int);
    modport fp_io (import FloatingPointRead,
        input clock, reset,
        output result_real);
endinterface
// ---i ---i ---i ---i ---i ---i ---i ---i ---i ---i Модуль верхнього рівня ---i ---i ---i ---i ---i-i-i-i-i
module dual_mu (input logic clock, reset);
    math_bus bus_a (clock, reset);          // Перша копія інтерфейсу
    math_bus bus_b (clock, reset);          // Друга копія інтерфейсу
    integer_math_unit i1 (bus_a.int_io);
    // Підключення до інтефейсу з використанням типу integer
    floating_point_unit i2 (bus_b.fp_io);
    // Підключення до інтефейсу з використанням типу real
endmodule
// ---i ---i ---i ---i ---i ---i ---i ---i ---i- Визначення модуля ---i ---i ---i ---i ---i-i-i-i-i-i-i-i-i-i
module integer_math_unit (interface io);
    int a_reg, b_reg;
    always @(posedge io.clock)
    begin
        io.IntegerRead(a_reg, b_reg);      // Виклик методу інтерфейсу
        ... // Інші математичні операції
    end
endmodule
```

```

        end
    endmodule
    module floating_point_unit (interface io);
        real a_reg, b_reg;
        always @(posedge io.clock)
        begin
            io.FloatingPointRead(a_reg, b_reg); // Виклик методу інтерфейсу
            ... // Інші математичні операції
        end
    endmodule

```

Модулі з імпортованими з інтерфейсу функціями і завданнями є синтезованими. Системи синтезу створюють локальну копію для кожної функції. Імпортовані завдання і функції повинні бути автоматичними.

19.4.2. Експортування завдань і функцій.

SystemVerilog пропонує можливість виконувати визначення завдань і функцій в одному модулі, а потім експортувати їх в інші модулі через інтерфейс (лістинг 19.12 і 19.13). Експортовані завдання не синтезуються, допускається їхнє використання тільки в абстрактних моделях.

Лістинг 19.12. Експорт функції з модуля через modport інтерфейсу.

```

interface chip_bus (input logic clock, reset);
    logic request, grant, ready;
    logic [63:0] address, data;
    modport master (output request, ...
        export check );
    modport slave (input request, ...
        import check );
endinterface
module CPU (chip_bus.master io);
    function check (input parity, input [63:0] data);
    ...
    endfunction
endmodule

```

Конструкція export дозволяє експортувати завдання і функції з модуля в інтерфейс через використання modport інтерфейсу. Завдання і функція може також експортуватися в інтерфейс без використання modport. Це робиться за допомогою декларації зовнішнього прототипу завдання або функції.

Лістинг 19.13. Експорт функції з модуля в інтерфейс.

```

interface chip_bus (input logic clock, reset);
    logic request, grant, ready;
    logic [63:0] address, data;
    extern function check(input logic parity,
        input logic [63:0] data);
    modport master (output request, ...);

```

```

        modport slave (input request, ...
            import function check (input logic parity, input logic [63:0] data) );
    endinterface
    module CPU (chip_bus.master io);
        function check (input logic parity, input logic [63:0] data);
            ...
        endfunction
        ...
    endmodule

```

Обмеження в експортуванні завдань і функцій:

1) не можна виконувати експорт того самого імені функції з різних копій модуля; 2) не можна експортувати ім'я завдання з декількох інтерфейсів, використовуючи декларацію `extern forkjoin`.

Завдання і функції можуть бути визначені в інтерфейсі (лістинг 19.14) або в одному, або в декількох підключених до інтерфейсу модулях. Це дозволяє виконувати моделювання на більш абстрактному рівні. Наприклад, “read” і “write” можуть бути визначені як завдання, без звернення до будь-яких ліній, а master може просто викликати ці завдання. У конструкції `modport` такі завдання декларуються за допомогою `import`.

Прототип функції або завдання описує типи даних і напряду аргументів, значення функції, що повертається, яка може бути визначена в будь-якому місці. При імпорті функції або завдання тільки один вираз може містити конструкцію `import`. Завдання (і тільки, але не функція) може бути визначене в модулі, який ініціалізується двічі або більше разів. Такі завдання декларуються в інтерфейсі за допомогою конструкції `extern forkjoin`.

Лістинг 19.14. Приклад використання завдання в інтерфейсі.

```

interface simple_bus (input bit clk);    // Визначення інтерфейсу
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
    task masterRead(input logic [7:0] raddr); // метод masterRead
        ...
    endtask: masterRead
    task slaveRead;                        // метод slaveRead
        ...
    endtask: slaveRead
endinterface: simple_bus
module memMod(interface a);              // Використовується абстрактний інтерфейс
    logic avail;
    always @(posedge a.clk)              // Сигнал clk з інтерфейсу
        a.gnt <= a.req & avail          // Сигнали gnt і req з інтерфейсу
    always @(a.start)

    a.slaveRead;

```



```

endmodule
module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;
    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // Виклик методу інтерфейсу
    ...
endmodule
module top;
    logic clk = 0;
    simple_bus sb_intf(clk); // Копія інтерфейсу
    memMod mem(sb_intf);
    cpuMod cpu(sb_intf);
endmodule

```

Приклад інтерфейсу (лістинг 19.15) представляє спосіб запису сигналів в modport для контролю їхнього напрямку, а також завдань доступу до сигналів для виконання операцій читання/запису (read/write).

Лістинг 19.15. Приклад використання завдань у конструкції modport.

```

interface simple_bus (input bit clk); // Визначення інтефейсу
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
modport slave (input req, addr, mode, start, clk,
    output gnt, rdy,
    ref data,
    import slaveRead,
    slaveWrite); // Імпорт у модуль, що використовує даний modport
modport master(input gnt, rdy, clk,
    output req, addr, mode, start,
    ref data,
    // Імпорт завдань у модуль, що використовує даний modport
    import masterRead, masterWrite);
    task masterRead(input logic [7:0] raddr); // Метод masterRead
    ...
    endtask
    task slaveRead; // Метод slaveRead
    ...
    endtask
    task masterWrite(input logic [7:0] waddr);
    ...
    endtask
    task slaveWrite;
    ...
    endtask

```

```

endinterface: simple_bus
module memMod(interface a);           // Використання тільки інтерфейсу
    logic avail;
    always @(posedge a.clk)           // Сигнал clk з інтерфейсу
        a.gnt <= a.req & avail;      // Сигнали gnt і req з інтерфейсу
    always @(a.start)
        if (a.mode[0] == 1'b0)
            a.slaveRead;
        else
            a.slaveWrite;
endmodule
module cpuMod(interface b);
    enum {read, write} instr = $rand();
    logic [7:0] raddr = $rand();
    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr);      // Виклик методу інтерфейсу
            ...
        else
            b.masterWrite(raddr);
endmodule
module omniMod( interface b);
    ...
endmodule: omniMod
module top;
    logic clk = 0;
    simple_bus sb_intf(clk);           // Копія інтерфейсу
    memMod mem(sb_intf.slave);         // Доступ тільки до завдань slave...
    cpuModcpu(sb_intf...master);       // Доступ тільки до завдань master...
    omniModomni(sb_intf);              //Доступні всі завдання інтерфейсу
endmodule

```

Приклад інтерфейсу з лістингу 19.16 представляє спосіб визначення завдань в одному модулі і виконання їхнього виклику в іншому, при цьому керування доступом до завдань здійснюється через modport.

Лістинг 19.16. Приклад експортування завдань і функцій.

```

interface simple_bus (input bit clk); // Визначення інтерфейсу
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
    modport slave( input req, addr, mode, start, clk,
        output gnt, rdy,
        ref data,
        export Read,Write);           // Експорт із модуля, що використовує modport
    modport master(input gnt, rdy, clk,
        output req, addr, mode, start,

```

```

    ref data,
    import task Read(input logic [7:0] raddr),
    task Write(input logic [7:0] waddr);    // Імпорт вимагає
                                           // повного прототипу завдання
endinterface: simple_bus
module memMod(interface a);              // Використання абстрактного інтерфейсу
    logic avail;
    task a.Read;                          // Метод Read
        avail = 0;
        ...
        avail = 1;
    endtask
    task a.Write;
        avail = 0;
        ...
        avail = 1;
    endtask
endmodule
module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;
    always @(posedge b.clk)
        if (instr == read)
            b.Read(raddr);    // Виклик методу через інтерфейс
            ...
        else
            b.Write(raddr);
endmodule
module top;
    logic clk = 0;
    simple_bus sb_intf(clk);    // Копія інтерфейсу
    memMod mem(sb_intf.slave); // Експорт завдань Read і Write
    cpuMod cpu(sb_intf.master); // Імпорт завдань Read і Write
endmodule

```

19.4.3. Приклад множинного експорту завдання.

Помилкою є ситуація, якщо більше одного модуля експортують однакове по імені завдання. Проте існують ситуації, коли необхідно використовувати кілька копій того самого типу `modport`, наприклад, модулів пам'яті. Такі завдання повинні бути визначені за допомогою ключових слів `extern forkjoin`. Виклик `extern forkjoin task countslaves( );` буде поводитися як наступний оператор:

```

fork
    top.mem1.a.countslaves;
    top.mem2.a.countslaves;
join

```

Для завдання читання `read task` тільки один модуль створить активну відповідь при його виклику, тобто тільки один буде містити відповідну адресу. Завдання в інших модулях будуть вертатися без ефекту. Тільки після цього буде викликане активне завдання для запису змінних результату. На відміну від завдань, множинний виклик функцій не дозволяється, оскільки вони завжди повертають результат.

Ефект використання оператора `disable` у завданні, що визначене `extern forkjoin task`, буде таким:

- 1) якщо звертання до завдання виконується через копію інтерфейсу, то всі виклики завдань будуть завершені;
- 2) якщо до завдання звертаються через копію модуля, то тільки одне завдання, що відповідає цьому модулю, буде завершене.

Якщо інтерфейс містить завдання `extern forkjoin task` і жоден модуль, що визначає завдання, не підключений до інтерфейсу, то кожне звертання до такого завдання призведе до появи помилки часу виконання.

Приклад інтерфейсу (лістинг 19.17) представляє завдання, визначені в декількох модулях, і їхній виклик через `extern forkjoin`. Механізм експортування множинних завдань також може бути використаний для підрахунку копій певного типу `modport`, які підключені до кожної копії інтерфейсу.

Лістинг 19.17. Приклад експортування завдань і функцій.

```
interface simple_bus (input bit clk);    // Визначення інтерфейсу
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
    int slaves = 0;
    // Завдання, виконувани паралельно, як блок fork/join
    extern forkjoin task countSlaves();
    extern forkjoin task Read (input logic [7:0] raddr);
    extern forkjoin task Write (input logic [7:0] waddr);
modport slave (input req,addr, mode, start, clk,
    output gnt, rdy,
    ref data, slaves,
    // Експорт із модуля, що використовує modport
    export Read, Write, countSlaves);
modport master ( input gnt, rdy, clk,
    output req, addr, mode, start,
    ref data,
    // При імпорті потрібен повний прототип завдання,
    import task Read(input logic [7:0] raddr),
        task Write(input logic [7:0] waddr));
initial begin
    slaves = 0;
    countSlaves;
    $display ("number of slaves = %d", slaves);
```

```

end
endinterface: simple_bus
module memMod #(parameter int minaddr=0, maxaddr=0;) (interface a);
    logic avail = 1;
    logic [7:0] mem[255:0];
    task a.countSlaves();
        a.slaves++;
    endtask
    task a.Read(input logic [7:0] raddr);        // Метод Read
        if (raddr >= minaddr && raddr <= maxaddr) begin
            avail = 0;
            #10 a.data = mem[raddr];
            avail = 1;
        end
    endtask
    task a.Write(input logic [7:0] waddr);       // Метод Write
        if (waddr >= minaddr && waddr <= maxaddr) begin
            avail = 0;
            #10 mem[waddr] = a.data;
            avail = 1;
        end
    endtask
endmodule
module cpuMod(interface b);
    typedef enum {read, write} instr;
    instr inst;
    logic [7:0] raddr;
    integer seed;
    always @(posedge b.clk) begin
        inst = instr'($dist_uniform(seed, 0, 1));
        raddr = $dist_uniform(seed, 0, 3);
        if (inst == read) begin
            $display("%t begin read %h @ %h", $time, b.data, raddr);
            callr:b.Read(raddr);
            $display("%t end read %h @ %h", $time, b.data, raddr);
        end
        else begin
            $display("%t begin write %h @ %h", $time, b.data, raddr);
            b.data = raddr;
            callw:b.Write(raddr);
            $display("%t end write %h @ %h", $time, b.data, raddr);
        end
    end
end
endmodule
module top;
    logic clk = 0;
    function void interrupt();
        disable mem1.a.Read;        // Завдання з копії модуля

```

```

        disable sb_intf.Write;           // Завдання з копії інтерфейсу
        if (mem1.avail == 0) $display ("mem1 was interrupted");
        if (mem2.avail == 0) $display ("mem2 was interrupted");
    endfunction
always #5 clk++;
initial begin
    #28 interrupt();
    #10 interrupt();
    #100 $finish;
end
simple_bus sb_intf(clk);
memMod #(0, 127) mem1(sb_intf.slave);
memMod #(128, 255) mem2(sb_intf.slave);
cpuMod cpu(sb_intf.master);
endmodule

```

19.5. Реконфігуровані інтерфейси

Інтерфейс може містити процедурні блоки (always, always_comb, always_ff, always_latch, initial або final) і оператори безперервного присвоєння (assign).

Інтерфейси можуть використовувати параметри для настройки розмірів, які можуть змінюватися при створенні копії інтерфейсу (лістинг 19.18 і 19.19).

Лістинг 19.18. Параметризовані типи даних в інтерфейсі.

```

interface math_bus
    #(parameter type DTYPE = int)
    (input logic clock);
    DTYPE a, b, result;           // Параметризований тип
    ...
    task Read (output DTYPE a, b);
        ... // Зчитування значень a і b
    endtask
    modport int_io (import Read,
        input clock, output result);
    modport fp_io (import Read,
        input clock, output result);
endinterface
module top (input logic clock, reset);
    math_bus bus_a(clock);           // Використання даних int
    math_bus (#DTYPE(real)) bus_b(clock); // Використання даних real
    integer_math_unit i1 (bus_a.int_io);
        // Підключення до інтерфейсу, що використовує цілочисельний тип
    floating_point_unit i2 (bus_b.fp_io);
        // Підключення до інтерфейсу, що використовує речовинний тип
    floating_point_unit i2 (bus_b.fp_io);
endmodule // end of module top

```

Лістинг 19.19. Використання параметрів в інтерфейсі.

```

interface simple_bus
    #(AWIDTH = 8, DWIDTH = 8)

```

```

        (input bit clk);
        logic req, gnt;
        logic [ AWIDTH-1:0] addr;
        logic [ DWIDTH-1:0] data;
        logic [1:0] mode;
        logic start, rdy;
    modport slave( input req, addr, mode, start, clk,
        output gnt, rdy,
        ref data,
        import task slaveRead,          // Імпорт завдань у модуль,
            task slaveWrite);          // який використовує даний modport
    modport master(input gnt, rdy, clk,
        output req, addr, mode, start,
        ref data,
        // Імпорт вимагає повного прототипу завдання
        import task masterRead(input logic [ AWIDTH-1:0] raddr),
            task masterWrite(input logic [ AWIDTH-1:0] waddr));
    task masterRead(input logic [ AWIDTH-1:0] raddr);          // Метод masterRead
    ...
endtask
task slaveRead;          // Метод slaveRead
    ...
endtask
task masterWrite(input logic [ AWIDTH-1:0] waddr);
    ...
endtask
task slaveWrite;
    ...
endtask
endinterface: simple_bus
module memMod(interface a); // Використання ключового слова interface
    logic avail;
    always @(posedge a.clk)    // Сигнал clk з інтерфейсу
        a.gnt <= a.req & avail;    // Сигнали gnt і req з інтерфейсу
    always @(a.start)
        if (a.mode[0] == 1'b0)
            a.slaveRead;
        else
            a.slaveWrite;
endmodule
module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;
    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // Виклик методу інтерфейсу
        ...
    else

```

```

        b.masterWrite(raddr);
    endmodule
    module top;
        logic clk = 0;

        // Копія інтерфейсу з параметрами за замовчуванням
        simple_bus sb_intf(clk);
        // Інтерфейс із 16-розрядними даними
        simple_bus #(DWIDTH(16)) wide_intf(clk);
        initial repeat(10) #10 clk++;
        memMod mem(sb_intf.slave); // Доступне тільки завдання slaveRead
        cpuMod cpu(sb_intf.master); // Доступне тільки завдання masterRead
        memMod mem(wide_intf.slave); // 16-розрядна пам'ять
        cpuMod cpu(wide_intf.master); // 16-розрядне cpu
    endmodule

```

19.6. Компроміси інтерфейсу

Інтерфейс не може містити ніяких вкладень - копій модулів або інших інтерфейсів. Існують відмінності у використанні інтерфейсів з modport у порівнянні із традиційним з'єднанням портів.

Переваги:

- 1) Інтерфейс ідеальний для повторного використання. Коли для комунікації застосовується протокол, сигнали повторюються.
- 2) Інтерфейс збирає безліч різноманітних сигналів, які декларуються в різних модулях або програмах, і розміщає їх в одному місці, що зменшує можливість помилки неправильного підключення сигналів.
- 3) Щоб додати нові сигнали, досить оголосити їх тільки в інтерфейсі, що також дозволяє мінімізувати ймовірність помилки.
- 4) Modport дозволяє модулю більш простим способом зв'язувати сигнали з інтерфейсом. Можна описувати напрямок сигналу для додаткової перевірки.

Недоліки:

- 1) Для з'єднання point-to-point інтерфейс, що використовує modport, має майже такий же великий розмір, як і застосування портів у списку сигналів. Однак всі декларації перебувають в одному місці, зменшуючи ймовірність внесення помилки.
- 2) Необхідність використання імені інтерфейсу на додаток до імені сигналу, що збільшує розмір модуля.
- 3) Якщо два блоки проекту з'єднуються за допомогою одного протоколу, що не буде повторно використовуватися, то побудова інтерфейсу вимагатиме більше зусиль, ніж безпосереднє з'єднання портів.
- 4) Досить складно виконати з'єднання двох різних інтерфейсів. Новий інтерфейс (bus_if) може містити, у добавок до нових сигналів, всі сигнали, які існують в іншому інтерфейсі (arb_if). Але оскільки інтерфейси не можуть бу-

ти ієрархічними, можна розкрити окремі сигнали і направити їх відповідним чином.

19.7. Моделі пам'яті

Лістинг 19.20 представляє найпростішу модель пам'яті, у якій реалізовані функції читання та запису за допомогою завдань ReadMem і WriteMem. У момент ініціалізації виконується сканування пам'яті на правильність виконання операцій читання із запису.

Для цього в операторі initial реалізується цикл for, що містить два блоки fork-join. Якщо операція запису/читання завершилася успішно, то ознака помилки $E = 0$. Інакше $E = 1$. Через 1000 од. часу керування передається наступному за блоком fork-join оператору.

Лістинг 19.20. Підсистема пам'яті із завданнями читання і запису.

```
module TopTasks;
  logic [20:0] A;
  logic [15:0] D;
  logic      E;
  parameter  LOWER = 20'h00000;
  parameter  UPPER = 20'h7ffff;
  logic [15:0] Mem[LOWER:UPPER];
  task ReadMem(input logic [19:0] Address,
               output logic [15:0] Data,
               output bit Error);
    if (Address >= LOWER && Address <= UPPER) begin
      Data = Mem[Address];
      Error = 0;
    end
    else Error = 1;
  endtask
  task WriteMem(input logic [19:0] Address,
                input logic [15:0] Data,
                output bit Error);
    if (Address >= LOWER && Address <= UPPER) begin
      Mem[Address] = Data;
      Error = 0;
    end
    else Error = 1;
  endtask
  initial begin
    // 2 операції читання запису/читання виконуються без помилки
    $display("\nExpect 2 write/read transactions with no errors, ");
    // 2 операції читання запису/читання з помилкою
    $display("followed by 2 write/read transactions with bus errors...\n");
    for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
      fork
        #1000;
        WriteMem(A[19:0], 0, E);    // Операція запису
```

```

join
  if (E) $display ("%t bus error on write %h", $time, A);
  else $display ("%t write OK %h", $time, A);
  fork
    #1000;
    ReadMem(A[19:0], D, E);    // Операція читання
  join
  if (E) $display ("%t bus error on read %h", $time, A);
  else $display ("%t read OK %h", $time, A);
end
$finish;
end
endmodule : TopTasks

```

Результати моделювання

```

1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000

```

19.7.1. Створення моделі рівня транзакції із застосуванням інтерфейсу.

У наступному прикладі (лістинг 19.21, рис. 19.2) система пам'яті поділяється на три підблока: два модулі і один testbench. Модулі з'єднуються за допомогою інтерфейсу. У цьому проекті область адрес визначена в компонентах пам'яті. В один момент часу тільки одна пам'ять відповідає за операцію читання або запису. Якщо немає модуля, що відповів на запит про транзакцію, то видається помилка шини. Широкомовний запит з однією відповіддю може бути легко промодельований за допомогою конструкції `extern forkjoin task` в інтерфейсі SystemVerilog, що обробляється подібно до `fork...join`, що містить множинний виклик завдань. Проте відмінність між ними полягає в тому, що в першому випадку кількість викликів підпрограм не визначено, а це дає можливість застосовувати той самий код інтерфейсу для будь-якої кількості компонентів пам'яті. Вихідні значення записуються в актуальні аргументи виклику кожного завдання, і відповідна затримка виклику завдання затримує відповідь, таким чином, відбувається перезапис неправильних значень.

Лістинг 19.21. Дві підсистеми пам'яті, підключені до інтерфейсу.

```

module TopTLM;
  Mmbus Mbus();
  Tester T(Mbus);
  Memory #(.Lo(20'h00000), .Hi(20'h3ffff))
    M1(Mbus);          // Молодша адреса

```

```

        Memory #(.Lo(20'h40000), .Hi(20'h7ffff))
            M2(Mbus);          // Старша адреса
endmodule : TopTLM

// Заголовок інтерфейсу
interface Mmbus;
    extern forkjoin task ReadMem (input logic [19:0] Address,
        output logic [15:0] Data, bit Error);
    extern forkjoin task WriteMem (input logic [19:0] Address,
        input logic [15:0] Data, output bit Error);
endinterface

module Tester (interface Bus);
    logic [15:0] D;
    logic E;
    int A;
    initial begin
        $display("\nExpect 2 write/read transactions with no errors, ");
        $display("followed by 2 write/read transactions with bus errors...\n");
        for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
            fork
                #1000;
                Bus.WriteMem(A[19:0], 0, E);
            join
            if (E) $display ("%t bus error on write %h", $time, A);
                else $display ("%t write OK %h", $time, A);
            fork
                #1000;
                Bus.ReadMem(A[19:0], D, E);
            join
            if (E) $display ("%t bus error on read %h", $time, A);
                else $display ("%t read OK %h", $time, A);
            end
            $display("");
            $finish;
        end
    endmodule

// Модулі пам'яті
module Memory(interface Bus);
    parameter Lo = 20'h00000;
    parameter Hi = 20'h3ffff;
    logic [15:0] Mem[Lo:Hi];
    task Bus.ReadMem(input logic [19:0] Address,
        output logic [15:0] Data, output bit Error);
        if (Address >= Lo && Address <= Hi) begin
            #100 Data = Mem[Address];
            Error = 0;
        end
    end
endmodule

```

```

    else Error = 1;
endtask
task Bus.WriteMem(input logic [19:0] Address,
    input logic [15:0] Data, output bit Error);
    if (Address >= Lo && Address <= Hi) begin
        #100 Mem[Address] = Data;
        Error = 0;
    end
    else Error = 1;
endtask
endmodule

```

Результати моделювання:

```

1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000

```

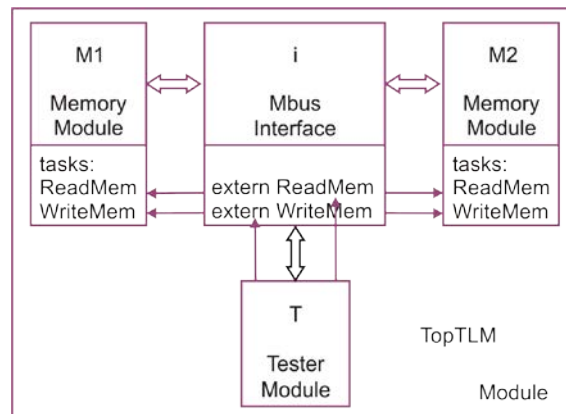


Рис. 19.2. Структурна схема моделі рівня транзакцій

19.8. Керування доступом до шини

Якщо існує два master компоненти шини, необхідно запобігти одночасному доступу обох пристроїв master до шини. Контроль над шиною в кожний момент часу може мати тільки один пристрій. Абстрактним механізмом для моделювання таких загальних ресурсів є «семафор». В SystemVerilog включений вбудований клас semaphore. Проте у даному прикладі використовується модель інтерфейсу, що ілюструє, як поводження семафора моделюється за допомогою інтерфейсу і його методів (лістинг 19.22). Інтерфейс Semaphore у наступному прикладі містить кількість ключів keys, що відповідає ресурсам - за замовчуванням один.

Завдання `get` чекає доступний ключ, а потім забирає його. Повертає ключ завдання `put`. Приклад містить модуль `Arbiter`, у якому використовується семафор. Інакше, семафор міг бути розміщений в інтерфейсі, але така модель не буде відповідати RTL-рівню.

Лістинг 19.22. TLM-модель із арбітражем шини за допомогою семафора.

```
module TopArbTLM;
    Membus Mbus();
    Tester T1(Mbus);
    Tester T2(Mbus);
    Arbiter A(Mbus);
    Memory #(.Lo(20'h00000), .Hi(20'h3ffff)) M1(Mbus);
    Memory #(.Lo(20'h40000), .Hi(20'h7ffff)) M2(Mbus);
endmodule : TopArbTLM
interface Membus;
    // Повторення прикладу 19.20
    extern forkjoin task ReadMem (input logic [19:0] Address,
                                output logic [15:0] Data, bit Error);
    extern forkjoin task WriteMem (input logic [19:0] Address,
                                input logic [15:0] Data, output bit Error);
    extern task Request();
    extern task Relinquish();
endinterface
interface Semaphore
    #(parameter int unsigned initial_keys = 1);
    int unsigned keys = initial_keys;
    task get(int unsigned n = 1);
        wait (n <= keys);
        keys -= n;
    endtask
    task put (int unsigned n = 1);
        keys += n;
    endtask
endinterface
module Arbiter (interface Bus);
    Semaphore s (); // Для вбудованого типу semaphore цей рядок
                    // мав би вигляд: semaphore s = new;

    task Bus.Request();
        s.get();
    endtask
    task Bus.Relinquish();
        s.put();
    endtask
endmodule
module Tester (interface Bus);
    logic [15:0] D;
    logic E;
    int A;
```

```

initial begin : test_block
  $display("\nExpect 2 sets of double-write/double-read transactions with no errors, ");
  $display("followed by 2 sets of double-write/double-read transactions with bus errors...\n");
  for (A = 0; A < 21'h100000; A = A + 21'h40000)
    begin : loop
      fork
        #1000;
        begin
          Bus.Request;
          Bus.WriteMem(A[19:0], 0, E);
          if (E) $display("%t bus error on write %h", $time, A);
          else $display ("%t write OK %h", $time, A);
          Bus.Relinquish;
        end
      join
      fork
        #1000;
        begin
          Bus.Request;
          Bus.ReadMem(A[19:0], D, E);
          if (E) $display("%t bus error on read %h", $time, A);
          else $display ("%t read OK %h", $time, A);
          Bus.Relinquish;
        end
      join
    end : loop
  $display("");
  $finish;
end : test_block
endmodule

```

```

// Модулі пам'яті
module Memory (interface Bus); // Повторення прикладу 19.20
  parameter Lo = 20'h00000;
  parameter Hi = 20'h3ffff;
  logic [15:0] Mem[Lo:Hi];
  task Bus.ReadMem(input logic [19:0] Address,
    output logic [15:0] Data,
    output bit Error);
    if (Address >= Lo && Address <= Hi) begin
      #100 Data = Mem[Address];
      Error = 0;
    end
    else Error = 1;
  endtask
  task Bus.WriteMem(input logic [19:0] Address,
    input logic [15:0] Data,
    output bit Error);

```

```

if (Address >= Lo && Address <= Hi) begin
    #100 Mem[Address] = Data;
    Error = 0;
end
else Error = 1;
endtask
endmodule

```

```

// Результати моделювання
100 write OK 00000000
200 write OK 00000000
1100 read OK 00000000
1200 read OK 00000000
2100 write OK 00040000
2200 write OK 00040000
3100 read OK 00040000
3200 read OK 00040000
4000 bus error on write 00080000
4000 bus error on write 00080000
5000 bus error on read 00080000
5000 bus error on read 00080000
6000 bus error on write 000c0000
6000 bus error on write 000c0000
7000 bus error on read 000c0000
7000 bus error on read 000c0000

```

19.9. Транзактор, адаптер і функціональні моделі шини

При використанні TLM-блоків для проектування апаратури їх підключають до RTL-моделей за допомогою конструкцій, званих транзакторами, адаптерами або функціональними моделями шини (bus functional models (BFMs)). Адаптери можуть працювати в режимі master або slave, залежно від напрямку керування. Master адаптер містить завдання для підсистем TLM master, які інкапсулюють протокол і маніпулюють сигналами для комунікації з RTL-моделлю slave-підсистеми. Slave-адаптер містить процеси, які спостерігають за змінами сигналів RTL-моделі master підсистеми і викликають завдання або функції в TLM підсистемах slave.

19.9.1. Master-адаптер у вигляді модуля.

Одним зі способів створення адаптерів є оформлення їх у вигляді модулів, які переводять інтерфейс рівня транзакцій в інтерфейс контактів, і навпаки (лістинг 19.23). Адаптер має два інтерфейсних порти - рівня транзакцій (TLM) і рівня контактів (TLM).

Лістинг 19.23. Модель із адаптером у вигляді модуля.

```

module TopTLMPLM;
    //Реалізація копії інтерфейсу для рівня транзакцій
    Multibus TLMbus();

```

```

//Реалізація копії інтерфейсу для нижнього рівня
Multibus PLMbus();
Tester T(TLMbus);           //Блок ініціалізації тестування
MultibusMaster MM (TLMbus, PLMbus);    //Адаптер
MultibusArbiter MA (PLMbus);           //Арбітр
Clock CLK(PLMbus);
MultibusMonitor MO(PLMbus);
MemoryPIN #(.Lo(20'h00000), .Hi(20'h3ffff))
M1 (PLMbus.ADR, PLMbus.DAT, PLMbus.MRDC, PLMbus.MWTC,
    PLMbus.XACK, PLMbus.BCLK);
MemoryPIN #(.Lo(20'h40000), .Hi(20'h7ffff))
M2 (PLMbus.ADR, PLMbus.DAT, PLMbus.MRDC, PLMbus.MWTC,
    PLMbus.XACK, PLMbus.BCLK);
endmodule : TopTLMPLM

```

Архітектура моделі зображена на рис 19.3. Tester - ініціалізує процес тестування пам'яті. Адаптер MultibusMaster реалізує операції читання і запису, запиту і звільнення шини. Арбітр MultibusArbiter стежить за порядком виділення шини. У модулях пам'яті не використовується інтерфейс і він реалізується на нижньому рівні. Два допоміжних блоки: Clock і MultibusMonitor генерують синхросигнал і спостерігають за зміною ліній інтерфейсу відповідно.

Інтерфейс Multibus (лістинг 19.24) є спрощеною версією шини, до якої може бути підключено кілька master компонентів, кожному з яких відповідає лінія запиту з масиву BREQ, що йде до арбітра, і вхідна лінія пріоритету BPRN, що йде від арбітра. Таким чином, застосовується техніка паралельних пріоритетів.

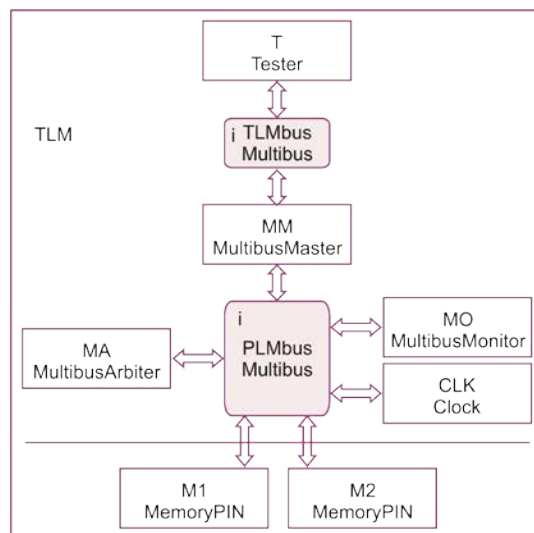


Рис. 19.3. Архітектура моделі з окремим адаптером

Лістинг 19.24. Модель шини із множинними master і slave

```

interface Multibus;
parameter int MASTERS = 1;          / Параметр задає кількість master,
                                     / підключених до шини

// Структурні комунікації
tri [19:0]      ADR;                  / Адресна шина (інвертується)
tri [15:0]      DAT;                  / Шина даних (інвертується)
wand /*active0*/ MRDC, MWTC;         / Команди читання/запису в пам'ять
wand /*active0*/ XACK;                / Підтвердження передачі
wand /*active0*/ [1:MASTERS] BREQ;    / Вектор запитів
                                     / master шини

wand /*active0*/ CBRQ;                / Загальний запит шини
wire /*active0*/ BUSY;                / Ознака зайнятості шини
wire /*active0*/ [1:MASTERS] BPRN;    / Шина пріоритетів master
logic           BCLK;                / Синхросигнал шини;
                                     / управляється одним master

logic           CCLK;                / Постійний синхросигнал
wand            INIT;                / Ініціалізація

// Завдання - поведінкові комунікації
extern task Request (input int n);
extern task Relinquish (input int n);
extern forkjoin task ReadMem (input logic [19:0] Address,
                              output logic [15:0] Data, bit Error);
extern forkjoin task WriteMem (input logic [19:0] Address,
                               input logic [15:0] Data, output bit Error);
endinterface

module Clock (Multibus Bus);
always begin                          / Синхронізація
    #50 Bus.CCLK = 0;
    #50 Bus.CCLK = 1;
end
endmodule : Clock

```

Адаптер MultibusMaster реалізує завдання, які управляють лініями і працюють із slave прототипом рівня транзакцій. Якщо тільки один драйвер дозволений для лінії, то може бути використаний тип даних logic. Інакше, якщо дозволені множинні драйвери, то адаптер має потребу в операторі безперервного призначення для буферизуючої лінії. Якщо master не має дозволу на керування шиною, то він формує запит до арбітра, очікує пріоритет, що буде йому призначений, і потім, якщо необхідно, звільнення шини попереднім master пристроєм. Ці дії інкапсулюються в завданні GetBus. Якщо жоден slave не відповідає адресі, виникає тайм-аут (перевищення часу виконання завдання) і вертається прапор помилки (лістинг 19.25).

Лістинг 19.25. TLM-модель шини з адаптером у вигляді модуля.

```

module MultibusMaster (interface Tasks, interface Wires);
    // Номер master, що отримувє доступ до шини
parameter int Number = 1;

```

```

enum {IDLE, READY, READ, WRITE} Master_State;
logic [19:0] adr = 'z; assign Wires.ADR = adr;
logic [15:0] dat = 'z; assign Wires.DAT = dat;
logic      mrdc = 1; assign Wires.MRDC = mrdc;           // Читання
logic      mwtc = 1; assign Wires.MWTC = mwtc;           // Запис
logic      breq = 1; assign Wires.BREQ[Number] = breq;
logic      cbrq = 1; assign Wires.CBRQ = cbrq; // Загальний запит шини
logic      busy = 1; assign Wires.BUSY = busy; // Шина зайнята
assign Wires.BCLK = Wires.CCLK;
task Tasks.ReadMem (input logic [19:0] Address,
                    output logic [15:0] Data,
                    output bit      Error);
    if (Master_State == IDLE) GetBus();
    else assert (Master_State == READY);
    Master_State = READ;
    Data = 'x; Error = 1;           // Значення за замовчуванням,
                                   // якщо не отримана відповідь від slave
adr = ~Address;
#50 mrdc = 0;                       //Мінімальна затримка
fork
    begin: ok
        @(negedge Wires.XACK) Data = ~ Wires.DAT;
        EndRead();
        @(posedge Wires.XACK) Error = 0;
        disable timeout;
    end
    begin: timeout // Тайм-аут, якщо не надійшло підтвердження запиту
        #900 Error = 1;
        EndRead();
        disable ok;
    end
join
FreeBus();
endtask
task Tasks.WriteMem (input logic [19:0] Address,
                    input logic [15:0] Data,
                    output bit      Error);
    if (Master_State == IDLE) GetBus();
    else assert (Master_State == READY);
    Master_State = WRITE;
    Error = 1;           // Значення за замовчуванням,
                        // якщо не отримана відповідь від slave
    GetBus();
    adr = ~Address;
    dat = ~Data;
    #50 mwtc = 0;
    fork
        begin: ok

```

```

    @(negedge Wires.XACK) EndWrite();
    @(posedge Wires.XACK) Error = 0;
    disable timeout;
end
begin: timeout // Тайм-аут, якщо не прийшло підтвердження запиту
    #900 Error = 1;
    EndWrite();
    disable ok;
end
join
FreeBus();
endtask
task EndRead();
    mrdc = 1;
    #50 adr = 'z;
endtask
task EndWrite();
    mwtc = 1;
    #60 adr = 'z;
    dat = 'z;
endtask
task GetBus();
    @(negedge Wires.BCLK) breq = 0;           // Формування запиту шини
    cbrq = 0;                                // Формування загального запиту шини
    @(negedge Wires.BPRN[Number]);
    @(negedge Wires.BCLK iff !Wires.BPRN[Number]);
    #50 busy = 0;
    cbrq = 1;
endtask
task FreeBus();
    breq = 1;
    if (Wires.CBRQ) Master_State = READY;
    else begin
        Master_State = IDLE;
        busy = 1;           // Звільнення шини, якщо встановлено
                           // загальний запит шини CBRQ
    end
endtask
endmodule: MultibusMaster
module Tester (interface Bus);
    logic [15:0] D;
    logic      E;
    int        A;
    initial begin
        $display("\n*** Expected results are " ***\n");
        for (A = 0; A < 21'h100000; A = A + 21'h40000)
            begin
                fork

```

```

        #1000;
        Bus.WriteMem(A[19:0], 0, E);
    join
    if (E) $display ("%t bus error on write %h", $time, A);
    else $display ("%t write OK %h", $time, A);
    fork
        #1000;
        Bus.ReadMem(A[19:0], D, E);
    join
    if (E) $display ("%t bus error on read %h", $time, A);
    else $display ("%t read OK %h", $time, A);
end
end
initial # 10000 $finish;
endmodule

module MultibusArbiter #(parameter MASTERS = 1)(interface Bus);
    // масив master, для запитів шини
    logic [1:MASTERS] bprn = '1; assign Bus.BPRN = bprn;
    int last = 0; // Останній, що відробив master
    int i;
    always @(negedge Bus.BCLK)
    if (Bus.CBRQ == 0) begin // Загальний запит шини
        i = last+1;
        forever begin
            if (i > MASTERS) i = 1;
            if (Bus.BREQ[i] == 0) break; //Запитів немає
            // Асерція видає повідомлення про помилку, якщо i = last
            assert (i != last) else $fatal(0, "no bus master");
            i++;
            if (i > MASTERS) i = 1;
        end
        last = i;
        #50 bprn [i] = 0;
        $display("bprn[%b] = %b", i, bprn);
    end
    else if (Bus.BUSY == 0) begin // Звільнення системного ресурсу
        #50 bprn [last] = 1;
    end
endmodule : MultibusArbiter

module MultibusMonitor (interface Bus);
    initial $monitor(
        "ADR=%h DAT=%h MRDC=%b MWTC=%b XACK=%b BREQ=%b CBRQ=%b
        BUSY=%b BPRN=%b",
        Bus.ADR, Bus.DAT, Bus.MRDC, Bus.MWTC, Bus.XACK,
        Bus.BREQ, Bus.CBRQ, Bus.BUSY, Bus.BPRN);
endmodule

```

```

// Memory Module with pin level interface
module MemoryPIN (
    input  [19:0]    ADR,           // Адресна шина
    inout  [15:0]    DAT,           // Шина даних
    input  /*active0*/ MRDC,         // Читання з пам'яті
    input  /*active0*/ MWTC,         // Запис на згадку
    output logic /*active0*/ XACK,    // Підтвердження запиту
    input          CCLK
);
parameter Lo = 20'h00000;
parameter Hi = 20'h3ffff;
logic [15:0] Mem[Lo:Hi];
logic [15:0] Bufdat;
logic      Bufena = 0;             // Буфер за замовчуванням відключений
initial XACK = 1; // Ознака відповіді системи, за замовчуванням, відповіді немає
assign DAT = Bufena ? Bufdat : 'z;
always @(posedge CCLK)
begin
    automatic logic [19:0] Address = ~ADR;
    if (MRDC == 0 && Address >= Lo && Address <= Hi) // Читання
    begin
        Bufdat <= ~Mem[Address];
        Bufena <= 1;
        XACK <= 0;
    end
    else if (MWTC == 0 && Address >= Lo && Address <= Hi)
    begin // Запис
        Mem[Address] = ~DAT;
        XACK <= 0;
    end
    else begin
        XACK <= 1;
        Bufena <= 0;
    end
end
endmodule: MemoryPIN

```

19.9.2. Адаптер в інтерфейсі.

Іншим способом створення моделі адаптера є розміщення його в інтерфейсі. У цьому випадку використовується один адаптер, через можливий конфлікт імен у протилежному випадку. Slave-адаптери, з іншого боку, викликають завдання або функції в slave, і може виникнути помилка елаборації, якщо slave буде відсутній. Таким чином, необхідна оновлена версія інтерфейсу. Лістинг 19.26 представляє приклад адаптера в інтерфейсі. Структурна схема компонентів такої моделі представлена на рис. 19.4. Дана модель відноситься до рівня TLM.

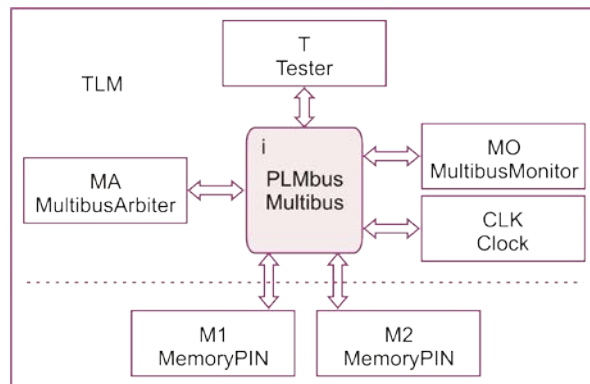


Рис.19.4. Архітектура моделі з адаптером в інтерфейсі

Лістинг 19.26. Приклад простої Multibus TLM з master-адаптером, реалізованим в інтерфейсі.

```

module TopInterfaceAdapter;
    Multibus Mbus();
    Tester T(Mbus);
    MultibusArbiter MA(Mbus);
    Clock Clk(Mbus);
    MultibusMonitor MO(Mbus);
    MemoryPIN #(Lo(20'h00000), .Hi(20'h3fff)) M1 (Mbus.ADR,
        Mbus.DAT, Mbus.MRDC, Mbus.MWTC, Mbus.XACK, Mbus.BCLK);
    MemoryPIN #(Lo(20'h40000), .Hi(20'h7fff)) M2 (Mbus.ADR,
        Mbus.DAT, Mbus.MRDC, Mbus.MWTC, Mbus.XACK, Mbus.BCLK);
endmodule : TopInterfaceAdapter

// Заголовок інтерфейсу
interface Multibus;
    parameter int MASTERS = 1;           // Число master шини
    parameter int Number = 1;
    // Структурні комунікації
    tri [19:0]    ADR;                   // Адресна шина
    tri [15:0]    DAT;                   // Шина даних
    wand /*active0*/ MRDC, MWTC;         // Команди читання/запису
    wand /*active0*/ XACK;               // Підтвердження прийому
    wand /*active0*/ [1:MASTERS] BREQ;
    wand /*active0*/ CBRQ;
    wire /*active0*/ BUSY;
    wire /*active0*/ [1:MASTERS] BPRN;
    logic        BCLK;
    logic        CCLK;
    wand         INIT;

    // Master-адаптер перетворить виклик ReadMem/WriteMem
    // у тимчасові діаграми
    enum {IDLE, READ, WRITE} Master_State;
    logic [19:0] adr = 'z; assign ADR = adr;

```

```

logic [15:0] dat = 'z; assign DAT = dat;
logic   mrdc = 1; assign MRDC = mrdc;
logic   mwtc = 1; assign MWTC = mwtc;
logic   breq = 1; assign BREQ[Number] = breq;
logic   cbrq = 1; assign CBRQ = cbrq;
logic   busy = 1; assign BUSY = busy;
task ReadMem (input logic [19:0] Address,
              output logic [15:0] Data,
              output bit   Error);
  assert (Master_State == IDLE);
  Master_State = READ;
  Data = 'x;
  Error = 1;      // За замовчуванням, якщо жоден slave не відповів
  GetBus();
  adr = ~Address;
  #50 mrdc = 0;   // Мінімальна затримка
  fork
    begin: ok
      @(negedge XACK) Data = ~ DAT;
      EndRead();
      @(posedge XACK) Error = 0;
      disable timeout;
    end
    begin: timeout// Час простою, якщо немає підтвердження прийому
      #900 Error = 1;
      EndRead();
      disable ok;
    end
  join
  FreeBus();
  Master_State = IDLE;
endtask
task WriteMem (input logic [19:0] Address,
              input logic [15:0] Data,
              output bit   Error);
  assert (Master_State == IDLE);
  Master_State = WRITE;
  Error = 1;      // За замовчуванням, якщо жоден slave не відповів
  GetBus();
  adr = ~Address;
  dat = ~Data;
  #50 mwtc = 0;
  fork
    begin: ok
      @(negedge XACK) EndWrite();
      @(posedge XACK) Error = 0;
      disable timeout;
    end
  join
  FreeBus();
  Master_State = IDLE;
endtask

```

```

        begin: timeout // Час простою, якщо немає підтвердження прийому
            #900 Error = 1;
            EndWrite();
            disable ok;
        end
    join
    FreeBus();
    Master_State = IDLE;
endtask
task EndRead();
    mrdc = 1;
    #50 adr = 'z;
endtask
task EndWrite();
    mwtc = 1;
    #60 adr = 'z;
    dat = 'z;
endtask
task GetBus();
    breq = 0;
    cbrq = 0;
    @(negedge BCLK iff !BPRN[Number]);
    #50 busy = 0;
    cbrq = 1;
endtask
task FreeBus();
    breq = 1;
    busy = 1;
endtask
endinterface
module Clock (Multibus Bus);
    always begin                // Синхронізація
        #50 Bus.BCLK = 0;
        #50 Bus.BCLK = 1;
    end
    initial # 10000 $finish;
endmodule : Clock
module Tester (interface Bus);
    logic [15:0] D;
    logic E;
    int A;
    initial begin
        for (A = 0; A < 21'h100000; A = A + 21'h40000)
            begin
                fork
                    #1000;
                    Bus.WriteMem(A[19:0], 0, E);
                join
            end
        end
    end
endmodule

```



```

        if (E) $display ("%t bus error on write %h", $time, A);
        else $display ("%t write OK %h", $time, A);
        fork
            #1000;
            Bus.ReadMem(A[19:0], D, E);
        join
        if (E) $display ("%t bus error on read %h", $time, A);
        else $display ("%t read OK %h", $time, A);
    end
end
endmodule

module MultibusArbiter #(parameter MASTERS = 1)(interface Bus);
    logic [1:MASTERS] bprn = '1; assign Bus.BPRN = bprn;
    int last = 0;
    int i;
    always @(negedge Bus.BCLK)
        if (Bus.CBRQ == 0) begin // Запити
            i = last+1;
            forever begin
                if (i > MASTERS) i = 1;
                if (Bus.BREQ[i] == 0) break;
                assert (i != last); else $fatal(0, "no bus master");
                i++;
                if (i > MASTERS) i = 1;
            end
            last = i;
            #50 bprn [i] = 0; // $display("bprn[%b] = %b", i, bprn);
        end
        else if (Bus.BUSY == 0) begin //Звільнення системного ресурсу
            #50 bprn [last] = 1;
        end
endmodule : MultibusArbiter

module MultibusMonitor (interface Bus);
    initial $monitor(
        "ADR=%h DAT=%h MRDC=%b MWTC=%b XACK=%b BREQ=%b
        CBRQ=%b BUSY=%b BPRN=%b", Bus.ADR, Bus.DAT, Bus.MRDC,
        Bus.MWTC, Bus.XACK, Bus.BREQ, Bus.CBRQ, Bus.BUSY, Bus.BPRN);
endmodule

// Memory Module with pin level interface
module MemoryPIN (
    input [19:0] ADR, // Адресна шина
    inout [15:0] DAT, // Шина даних
    input /*active0*/ MRDC, // Читання з пам'яті
    input /*active0*/ MWTC, // Запис в пам'ять
    output logic /*active0*/ XACK, // Підтвердження запиту
    input CCLK
);
parameter Lo = 20'h00000;

```

```

parameter Hi = 20'h3ffff;
logic [15:0] Mem[Lo:Hi];
logic [15:0] Bufdat;
logic      Bufena = 0;                // Відключення буферів за замовчуванням
initial XACK = 1; // За замовчуванням відповідей немає
assign DAT = Bufena ? Bufdat : 'z;
always @(posedge CCLK) begin
    automatic logic [19:0] Address = ~ADR;
    if ( MRDC == 0 && Address >= Lo && Address <= Hi) // Читання
    begin
        Bufdat <= ~Mem[Address];
        Bufena <= 1;
        XACK <= 0;
    end
    else if (MWTC == 0 && Address >= Lo && Address <= Hi)
    begin
        // Запис
        Mem[Address] = ~DAT;
        XACK <= 0;
    end
    else begin
        XACK <= 1;
        Bufena <= 0;
    end
end
endmodule: MemoryPIN

```

Представлені моделі транзакції є простими в тому плані, що тільки одна транзакція може мати місце в один момент часу. Це означає, що життєвий цикл транзакції відповідає життєвому циклу транзакції, що викликає її. Завдання містить дані, що відповідають транзакції, наприклад, такі як початковий час. У системах допускається починати виконання нової транзакції до завершення попередньої (перекриття часу або конвеєр). Вони навіть можуть дозволяти асинхронне виконання (розподіл транзакцій). У такому випадку дані про транзакцію не можуть міститися в одному завданні. Або новий процес(потік) повинен бути породжений для керування чи моніторингу транзакції і містити відповідні дані, або об'єкт динамічних даних повинен бути створений для зберігання інформації про транзакцію.

19.10. Контрольні запитання й завдання

1. Дати визначення призначення конструкції інтерфейс.
2. Яке співвідношення апаратних витрат для компонентів «пам'ять-логіка» у сучасних цифрових системах на кристалах?
3. Що таке modport? У яких випадках використовується дана конструкція?
4. Розробити модель і структурну модель пристрою на рис. 19.5.
5. Як реалізується імпорт і експорт завдань в інтерфейсі? Навести приклади використання.
6. Наведіть структуру опису портів модуля.

7. Опишіть існуючі моделі пам'яті.
8. Опишіть архітектуру моделі з окремим адаптером.
9. Опишіть архітектуру моделі з адаптером в інтерфейсі.
10. Як здійснюється керування доступом до шини?
11. Поясніть структуру TLM-моделі з арбітражем шини за допомогою семафора.
12. Перелічіть технології виготовлення пам'яті.
13. Як використовуються параметризовані типи даних в інтерфейсі?
14. Як можна запобігти одночасному доступу декількох пристроїв master до шини?

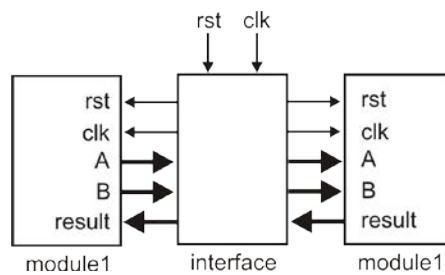


Рис.19.5. Архітектура моделі з адаптером в інтерфейсі

20. ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

20.1. Визначення класу в SystemVerilog

Поняття класу дозволяє поєднувати дані разом з підпрограмами, що маніпулюють ними. Найпростіший синтаксис:

```
class class_name
    //список аргументів
endclass
```

Лістинг 20.1 Представляє клас BusTran, що описує типовий мережний пакет, який включає адресу джерела і призначення, а також масив значень даних. Клас BusTran містить функції для виведення змісту пакета і обчислення CRC (cyclic redundancy check) даних. Для того щоб можна було легко визначити початок і кінець класу, використовуються мітки.

Лістинг 20.1. Найпростіший клас BusTran.

```
class BusTran;
    bit [31:0] addr, crc, data[8];
    function void display;
        $display("BusTran: %h", addr);
    endfunction : display
    function void calc_crc;
        crc = addr ^ data.xor;
    endfunction : calc_crc
endclass : BusTran
```

В SystemVerilog класи можуть бути оголошені в program, module, package або в іншому місці. Вони можуть бути використані в програмах і модулях.

Програми містять один тест і містять у собі testbench, блок initial для створення, ініціалізації й запуску тесту. Багато груп верифікації розміщують автономний клас або групу споріднених класів у файл. Набір класів оформляється в пакет package. Наприклад, разом в один пакет можуть бути згруповані всі транзакції SCSI/ATA. Пакет може бути скомпільований окремо, незалежно від тесту системи. Неспоріднені класи, такі, як транзакції, scoreboards або різні протоколи, повинні бути оформлені в окремі файли.

20.2. Об'єктно-орієнтоване програмування (ООП). Термінологія ООП

1. Клас - основний блок, що містить підпрограми і змінні. Аналогом в Verilog є модуль.
2. Об'єкт - копія класу. Подібно до створення в Verilog копії модуля можливості його наступного використання.
3. Визначник (Handle) - покажчик на об'єкт. В Verilog використовується ім'я копії модуля для звертання до сигналів і підпрограм поза модулем. Покаж-

чик в ООП подібний до адреси на об'єкт, але він зберігається в покажчику, що може посилатися на будь-який тип.

4. Властивості (Property) - змінні, що зберігають дані. В Verilog - це сигнали: регістри або ланцюги.

5. Метод (Method) - процедурний код для маніпулювання змінними, що оформляється у вигляді завдань і функцій. В Verilog модулі містять завдання й функції, а також блоки initial і always.

6. Прототипи (Prototype) - заголовки підпрограм, що містять імена, типи і списки аргументів. Тіло підпрограми містить виконуваний код.

20.3. Створення нових об'єктів

У наступному прикладі b - це визначник (handle) об'єкта типу BusTran.

```
BusTran b;           // Декларація об'єкта класу
b = new;             // Виділення місця під об'єкт BusTran
```

В момент декларація-визначник b ініціалізується нульовим покажчиком null. Потім для створення об'єкта BusTran може бути викликана функція new, що виділяє простір під BusTran, ініціалізує змінні значенням за замовчуванням (0 - для 2-значових змінних і X - для 4-значових) і повертає адресу об'єкта. В SystemVerilog для кожного класу створюється функція new, призначена для виділення пам'яті та ініціалізації значень об'єктів.

20.3.1. Користувальницькі конструктори.

SystemVerilog дозволяє користувачу створювати свою функцію new (лістинг 20.2). Вона не має типу, оскільки завжди повертає об'єкт того ж типу, що й клас.

Лістинг 20.2. Визначення користувальницької функції new.

```
class BusTran;
    logic [31:0] addr, crc, data[8];
    function new;
        addr = 3;
        foreach (data[i])
            data[i] = 5;
    endfunction
endclass
```

Тут функція присвоює змінним addr і data певні значення, але залишає для змінної crc значення за замовчуванням X (SystemVerilog виділяє місце для об'єкта автоматично). Можна вказувати для аргументів функції значення за замовчуванням, щоб зробити конструктор більш гнучким (лістинг 20.3).

Лістинг 20.3. Функція new з аргументами.

```
class BusTran;
    logic [31:0] addr, crc, data[8];
    function new(logic [31:0] addr=3, d=5);
        this.addr = addr;
```

```

        foreach (data[i])
            data[i] = d;
        endfunction
    endclass
    initial begin
        BusTran b;
        b = new(10); // поле data отримує значення 5 за замовчуванням
    end

```

У прикладі (лістинг 20.4) виконується виклик new для BusTran з конструктора Driver. Оскільки bt - це дескриптор типу BusTran, SystemVerilog створює об'єкт, що має тип BusTran.

Лістинг 20.4. Виклик функції new у класі.

```

class BusTran;
    ...
endclass : BusTran
class Driver;
    BusTran bt;
    function new();    // Функція new класу Driver
        bt = new();    // Виклик функції new класу BusTran
    endfunction
endclass : Driver

```

Слід уникати виконання в одному операторі декларації дескриптора і виклику конструктора new. Хоча це припустимо з погляду синтаксису, але може створити ряд проблем, тому що конструктор викликається до виконання операторів, особливо, якщо необхідно виконати декларацію об'єкта в певному порядку. Якщо не використовувати режим automatic storage, конструктор буде викликаний на початку моделювання.

20.3.2. Звільнення об'єктів.

Коли дескриптор припиняє вказувати на об'єкт, то він видаляється з пам'яті. Наприклад:

```

BusTran b;        // Створення визначника
b = new;          // Виділення пам'яті під об'єкт
b = new;          // Виділення пам'яті під другий об'єкт, перший видаляється
b = null;         // Видалення другого об'єкта

```

У другому рядку виклик конструктора new виділяє пам'ять під об'єкт і присвоює показник на нього дескриптору b. Наступний виклик конструктора new створює новий об'єкт і записує адресу на нього в b, перепризначаючи попереднє значення і видаляючи перший об'єкт. У четвертому рядку відбувається видалення другого об'єкта.

SystemVerilog ніколи не видаляє об'єкт, на який указує дескриптор. Видалення необхідно робити вручну за допомогою оператора null.

20.3.3. Використання об'єктів.

Для доступу до змінних і підпрограм класу можна використовувати позначення крапки “.”. Наприклад:

```
BusTran b;           // Створення визначника BusTran
b = new;             // Виклик конструктора BusTran
b.addr = 32'h42;     // Присвоєння значення змінної
b.display();         // Виклик підпрограми
```

У строгому ООП доступ до змінної здійснюється тільки через публічні (public) методи, наприклад `get()` і `put()`. Це підвищує стабільність розроблювальних програм, крім випадкової зміни полів, однак знижує гнучкість, необхідну для побудови testbench. Тому для мов верифікації кращим рішенням є наявність прямого доступу до змінних класу.

20.4. Статичні і глобальні змінні

В SystemVerilog є можливість створювати статичну змінну, що є загальною для всіх копій класу. У лістингу 20.5 статична змінна `count` зберігає кількість створених на даний момент об'єктів (рис. 20.1). Вона ініціалізується в 0 при декларації, тому що на початку моделювання відсутні транзакції. Щоразу, при створенні нового об'єкта, значення `count` збільшується на 1. У прикладі (рис. 20.1) є присутньою тільки одна копія статичної змінної `count`, незалежно від того, скільки об'єктів `BusTran` буде створено. Змінна `id` не є статичною, тому кожний об'єкт `BusTran` має свою власну копію, і немає необхідності в створенні глобальної змінної для `count`.

Лістинг 20.5. Клас зі статичною змінною.

```
class BusTran;
    static int count = 0;           // Лічильник створюваних об'єктів
    int id;                         // Унікальний ID копії об'єкта
    function new;
        id = count++;              // Установка ID за допомогою лічильника
    endfunction
endclass
BusTran b1, b2;
initial begin
    b1 = new;                       // Перша копія, id=0
    b2 = new;                       // Друга копія, id=1
    $display("Second id=%d, count=%b", b2.id, b2.count);
end
```

Статичні змінні, як правило, ініціалізуються в момент декларації. Не можна використовувати для цього функцію `new`, оскільки вона викликається кожного разу при створенні нової копії класу. Для більш складної ініціалізації може бути використаний блок `initial`. Необхідно тільки гарантувати, що статична змінна ініціалізована до створення першої копії класу. У прикладі (лістинг 20.6) дескриптор `s` дорівнює `null` у момент виклику функції `initialize`. Це припустимо, оскільки завдання використовує тільки статичні змінні, які не створюються конструктором.

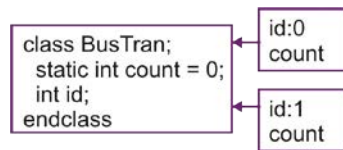


Рис. 20.1. Статична змінна в класі

Лістинг 20.6. Ініціалізація статичної змінної за допомогою завдання.

```
class MyStatic;
    static int count;
    task initialize(int val);
        count = val;
    endtask
endclass
MyStatic s;
initial
    s.initialize(42);
```

20.5. Підпрограми або методи класу

Підпрограми або методи (task або function) класу визначаються всередині нього. Лістинг 20.7 представляє приклад визначення підпрограми display() для класів BusTran і PCI_Tran. SystemVerilog викликає підпрограму, ґрунтуючись на типі дескриптора.

Лістинг 20.7. Підпрограми в класі.

```
class BusTran;
    bit [31:0] addr, crc, data[8];
    function void display();
        $display("@%0d: BusTran addr=%h, crc=%h", addr, crc);
        $write("\tdata[ 0-7]=");
        foreach (data[i]) $write(data[i]);
        $display();
    endfunction
endclass
class PCI_Tran;
    bit [31:0] addr, data;
    function void display();
        $display("@%0d: PCI: addr=%h, data=%h", addr, data);
    endfunction
endclass
BusTran b;
PCI_Tran pc;
initial begin
    b = new();           // Конструктор BusTran
    b.display();         // Виведення значення BusTran
    pc = new();          // Конструктор транзакції PCT
    pc.display();        // Виведення значень транзакції PCI
end
```


Підпрограми в класі завжди використовують автоматичний розподіл пам'яті, отже, вони не мають потреби в застосуванні модифікатора `automatic`.

20.5.1. Визначення підпрограм за межами класу.

Емпіричне правило говорить: «Щоб код програми був зрозумілим, він повинен бути не більше однієї сторінки».

В SystemVerilog підпрограму можна розбивати на прототип (ім'я підпрограми і аргументи) всередині класу і тіло (процедурний код), що може бути записано поза класом. У цьому випадку перед іменем підпрограми в класі додається слово `extern`. При визначенні тіла підпрограми перед її іменем ставиться ім'я класу, відокремлене від нього двома дужками (::).

Класи з лістингу 20.7 можна було б визначити з використанням підходу, представленого лістингом 20.8.

Лістинг 20.8. Зовнішнє визначення підпрограм.

```
class BusTran;
    bit [31:0] addr, crc, data[8];
    extern function void display();
endclass
function void BusTran::display();
    $display("@%0d: BusTran addr=%h, crc=%h", addr, crc);
    $write("\tdata[ 0-7]=");
    foreach (data[i]) $write(data[i]);
    $display();
endfunction
class PCI_Tran;
    bit [31:0] addr, data;
    extern function void display();
endclass
function void PCI_Tran::display();
    $display("@%0d: PCI: addr=%h, data=%h", addr, data);
endfunction
```

20.6. Правила меж видимості

SystemVerilog дотримується основних правил Verilog, але має ряд доповнень. Межі видимості визначаються блоками коду, такими як модуль, програма, завдання, функція, клас або блок `begin-end`. Для циклів `for` і `foreach` автоматично створюється блок. Таким чином, індексні змінні є локальними для цієї межі видимості.

Нові змінні можуть бути визначені всередині блоку. SystemVerilog допускає створення змінних у безіменних блоках `begin-end`. Ім'я може бути відносним або абсолютним, починається з `$root`. Для відносних імен SystemVerilog переглядає список меж видимості, поки не знайде збіг. Для того щоб уникнути двозначності, слід використовувати `$root` перед іменем.

У наступному прикладі (лістинг 20.9) застосовується одне й те саме ім'я в різних межах видимості. Ім'я `limit` відповідає глобальній змінній, змінній програми, класу, завдання і локальній змінній в блоці `initial`. Останній з них є безіменним блоком, тому ім'я мітки, пов'язаної з іменем змінної, залежить від середовища симулятора.

Лістинг 20.9. Межі видимості імен.

```
int limit;                                // $root.limit
program p;
    int limit;                            // $root.p.limit
    class Foo;
        int limit, array[];              // $root.p.Foo.limit
        task print (int limit);           // $root.p.Foo.print.limit
            for (int i=0; i<limit; i++)
                $display("%m: array[%0d]=%0d", i, array[i]);
        endfunction
    endclass
    initial begin
        int limit = $root.limit;          // $root.p.$unnamed.limit
        Foo bar;
        bar = new;
        bar.array = new[limit];
        bar.print (limit);
    end
endprogram
```

Слід виконувати декларування класів за межами програми або модуля. Лістинг 20.10 представляє функцію `Bug::display`, що не декларує змінну циклу `i`, а замість цього використовує змінну `i` програмного рівня, звертаючись до неї за допомогою ієрархічного імені `test.i`.

Лістинг 20.10. Неприпустиме використання зовнішніх змінних у класі.

```
program test;
    int i;                                // Змінні програмного рівня
    class Bug;
        logic [31:0] data[9];
        // Calling this function changes the program variable
        function void display;
            // Forgot to declare i in next statement
            for (i = 0; i<data.size; i++)
                $display("data[%0d]=%x", i, data[i]);
        endfunction
    endclass
endprogram
```

Ключове слово `this` дозволяє уникнути неоднозначності при визначенні змінних у класі, а у випадку використання однакових імен - різних зон видимості. Наприклад, у наступному коді `opate` є локальною змінною функції і змінною класу `Scoring`.

```

class Scoping;
    string oname;
    function new(string oname);
        this.oname = oname; // Змінна класу отримує
                           // значення локальної змінної
    endfunction
endclass

```

Варто бути обережним при визначенні змінних. Якщо компілятор не знаходить змінну всередині блоку, він шукає її ззовні. Це може призвести до одночасного використання однієї змінної декількома потоками, і помилка буде виявлена тільки на етапі моделювання.

Лістинг 20.11. Неправильне використання загальної змінної програми.

```

program bug;
    class Buggy;
        int data[10];
        task transmit;
            fork
                for (i=0; i<10; i++) // Змінна i не оголошена в класі
                    send(data[i]);
            join_none
        endtask
    endclass

    int i; // програмний рівень
    Buggy b;
    event receive;
    initial begin
        b = new;
        for (i=0; i<10; i++) b.data[i] = i;
        b.transmit;
        for (i=0; i<10; i++)
            @(recieve) $display(data[i]);
    end
endprogram

```

20.7. Використання одного класу в межах іншого

Один клас може містити копію іншого, використовуючи дескриптор як об'єкт, подібно до того, як модуль в Verilog може містити інший модуль. Основна ідея застосування вкладень полягає в можливості повторного використання і керування складністю проекту. Наприклад, кожна транзакція може містити блок статистики з тимчасовими мітками початку і завершення транзакції, а також інформацію про всі транзакції (рис. 20.2, лістинг 20.12).

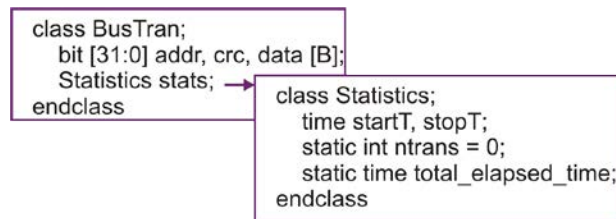


Рис. 20.2. Вкладені об'єкти

Лістинг 20.12. Використання класу в класі.

```

// Клас Statistics.
class Statistics;
    time start, stop;                // Час транзакції
    static int ntrans = 0;           // Лічильник транзакцій
    static time total_elapsed_time = 0;
    function time how_long;
        how_long = stop - start;
        ntrans++;
        total_elapsed_time += how_long;
    endfunction

    function void start;
        start = $time;
    endfunction
endclass

// Використання класу Statistics у класі BusTran
class BusTran;
    bit [31:0] addr, crc, data[8];
    Statistics stats;                // Визначник класу Statistics

    function new();
        stats = new();              // Виділення пам'яті під об'єкт stats
    endfunction
    task create_packet();
        ...                          // Заповнення пакета даними
    stats.start();                  // Передача пакета
    endtask
endclass

```

Зовнішній клас BusTran може звертатися до елементів класу Statistics за допомогою ієрархічного синтаксису, наприклад stats.start. Необхідно пам'ятати, що значення дескриптора вкладеної копії об'єкта дорівнює null і виклик start призведе до помилки. Ініціалізацію копій вкладених класів краще зробити в конструкторі зовнішнього класу BusTran.

20.8. Порядок компіляції

Якщо необхідно виконати компіляцію класу до визначення вкладеного в нього класу, то останній може бути оголошений за допомогою конструкції typedef, наприклад, у такий спосіб:

```
// Використання оператора typedef для оголошення класу.
typedef class Statistics;           // Декларація класу нижнього рівня
class BusTran;
    Statistics stats;               // Використання класу Statistics
    ...
endclass
class Statistics;                  // Опис класу Statistics
    ...
endclass
```

20.9. Поняття динамічного об'єкта

При виклику підпрограми в неї передається дескриптор на об'єкт, але не сам об'єкт. На рис. 20.3 завдання generator викликає завдання transmit. Два дескриптори generator.b і transmit.btrans посилаються на той самий об'єкт.

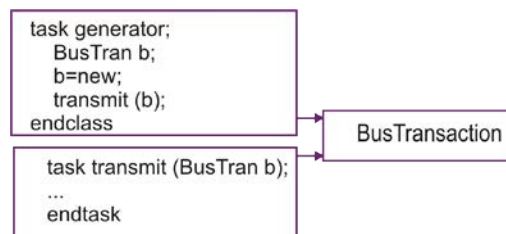


Рис. 20.3. Передача визначників і об'єктів між підпрограмами

При виклику підпрограми зі скалярними змінними і використанням ключового слова ref SystemVerilog передає адресу. Інакше - SystemVerilog передає копію змінних. При цьому їхні зміни в підпрограмі не вплинуть на оригінальні значення змінних.

У прикладі, представленому лістингом 20.13, у блоці initial створюється об'єкт BusTran, що викликає завдання transmit з визначником, що вказує на об'єкт. За допомогою визначника завдання transmit може читати і записувати дані в об'єкт. Проте, якщо transmit спробує модифікувати дескриптор, результат не буде видний у блоці initial, оскільки bt не оголошений як ref. Підпрограма може модифікувати об'єкт, навіть якщо аргумент дескриптора не має модифікатора ref. Як показано нижче, функція transmit може записати timestamp в об'єкт. Якщо немає необхідності модифікації об'єкта в підпрограмі, можна передати його копію.

Лістинг 20.13. Пасивні об'єкти.

```
// Передача пакета в 32-бітову шину
task transmit(BusTran bt);
    CBbus.rx_data <= bt.data;
    bt.timestamp = $time;
    ...
endtask
BusTran b;
initial begin
```

```

        b = new();           // Виділення пам'яті під об'єкт
        b.addr = 42;         // Ініціалізація значення
        transmit(b);         // Передача об'єкта в завдання
    end

```

20.10. Зміна визначника в завданні

Частою помилкою кодування є пропуск ключового слова `ref` в описі аргументів підпрограми. Наприклад, у наступному коді аргумент `b` не має модифікатора `ref`, таким чином, будь-які його зміни не будуть видні в кодї, що викликає:

```

task create_packet(BusTran bt);           // Неправильна декларація аргументів
                                           // функції, пропущений ref

        bt = new();
        bt.addr = 42;
        // Initialize other fields
    endtask
BusTran b;
initial begin
        create_packet(b);                 // Виклик підпрограми з помилкою
        $display(b.addr);                 // Призводить до помилки, тому що b=null
    end

```

Хоча `create_packet` змінив аргумент `bt`, визначник `b` повинен залишатися рівним `null`. Щоб виправити ситуацію, аргумент `bt` повинен бути оголошений як `ref`.

```

task create_packet(ref BusTran bt);       // Правильно
    ...
endtask

```

20.11. Зміна об'єктів «на льоту»

Ще однією часто виникаючою помилкою є відсутність створення нового об'єкта при виконанні кожної транзакції. У прикладі завдання `generate_trans` створює об'єкт `BusTran` з випадковим значенням і передає його в проект, що займає кілька тактів.

```

// Неправильний опис генератора
task generator_bad(int n);
    BusTran b;
    b = new();           // Створення одного нового об'єкта
    repeat (n) begin
        b.addr = $random();           // Ініціалізація змінних
        $display("Sending addr=%h", b.addr);
        transmit(b);                 // Передача об'єкта в DUT
    end
endtask

```

Код створює тільки один об'єкт `BusTran`. Таким чином, щоразу у циклі відбувається зміна об'єкта `generator_bad` одночасно з його передачею. У момент використання `$display` виведе багато значень `addr`, але в `BusTrans` буде переданий той самий стан `addr`.

Помилка виникає, якщо transmit зберігає об'єкт навіть після повернення з transmit. Якщо завдання transmit зберігає посилання на об'єкт, то він може бути використаний знову й знову. Необхідно створювати новий об'єкт BusTran при кожному проході циклу.

```
// Правильний опис генератора
task generator_good(int n);
    BusTran b;
    repeat (n) begin
        b = new();                // Створення одного нового об'єкта
        b.addr = $random();        // Ініціалізація змінних
        $display("Sending addr=%h", b.addr);
        transmit(b);              // Передача об'єкта в DUT
    end
endtask
```

20.12. Масиви визначників

Testbench може оперувати багатьма об'єктами. Можна створити масив дескрипторів, кожний з яких посилається на свій власний об'єкт. Приклад представляє спосіб зберігання десяти транзакцій шини в масиві:

```
// Використання масиву дескрипторів
task generator();
    BusTran barray[10];
    foreach (barray[i]) begin
        barray[i] = new();        // Конструктор кожного об'єкта
        transmit(barray[i]);
    end
endtask
```

Масив barray створений з дескрипторів, але не об'єктів, тому необхідно створювати кожний об'єкт масиву до його використання. Немає можливості застосовувати функцію new для цілого масиву.

20.13. Копіювання об'єктів

20.13.1. Копіювання об'єкта за допомогою оператора new.

Використання new для копіювання об'єктів - просте і надійне рішення. Створюється новий об'єкт, а значення всіх змінних копіюються в нього з існуючого об'єкта.

```
// Копіювання простого класу за допомогою new
class BusTran;
    bit [31:0] addr, crc, data[8];
endclass
BusTran src, dst;
initial begin
    src = new;                    // Створення першого об'єкта
    dst = new src;                // Створення копії за допомогою new
end
```

Так створюється проста копія. Якщо клас містить дескриптори на об'єкти інших класів, то тільки об'єкти верхнього рівня будуть скопійовані, а об'єкти нижніх рівнів - ні. У прикладі (лістинг 20.14) клас BusTran містить дескриптор на клас Statistics, представлений лістингом 20.12.

Лістинг 20.14. Копіювання складного класу за допомогою оператора new.

```
class BusTran;
    bit [31:0] addr, crc, data[8];
    static int count = 0;
    int id;
    Statistics stats;
    function new;
        stats = new;
        id = count++;
    endfunction
endclass

BusTran src, dst;
initial begin
    src = new;                // Створення першого об'єкта
    src.stats.start = 42;
    dst = new src;            // Копіювання src в dst
    dst.stats.start = 84; // Зміна stats для dst і src
end
```

Блок initial створює перший об'єкт BusTran і модифікує змінну у вкладеному об'єкті Statistics (рис. 20.4).

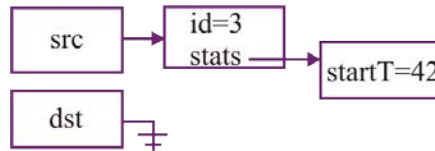


Рис. 20.4. Об'єкти і визначники до копіювання за допомогою new

Використання new створює копію об'єкта BusTran, але не об'єкта Statistics. Тому при звертанні до функції new не виконується виклик власної функції new (рис. 20.5). Замість значень змінних копіюється дескриптор. Таким чином, обидва об'єкти BusTran вказують на той самий об'єкт Statistics і мають те ж саме значення id.

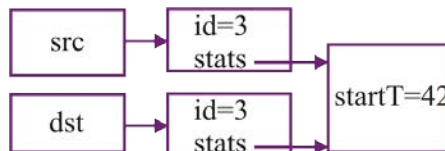


Рис. 20.5. Об'єкти і визначники після копіювання за допомогою new

20.13.2. Створення власної функції копіювання

Якщо клас не містить посилань на інші аналоги, то функція копіювання `copy` може мати вигляд лістинга 20.15.

Лістинг 20.15. Найпростіший клас із функцією копіювання.

```
class BusTran;
    bit [31:0] addr, crc, data[8];

    function BusTran copy;
        copy = new;           // Конструктор для копії
        copy.addr = addr;     // Заповнення значень даних
        copy.crc = crc;
        copy.data = data;     // Копія масиву
    endfunction
endclass
...
// Використання функції copy
BusTran src, dst;
initial begin
    src = new;               // Створення першого об'єкта
    dst = src.copy;
end
```

Для більш складних класів варто мати власні функції копіювання `copy`. Наприклад, необхідно створити функцію копіювання для класу `Statistics` і для кожного класу ієрархії (рис. 20.6, лістинг 20.16).



Рис. 20.6. Об'єкти і визначники після копіювання (лістинг 20.16)

Лістинг 20.16. Складні класи з функцією `copy`.

```
class BusTran;
    bit [31:0] addr, crc, data[8];
    Statistics stats;
    static int count = 0;
    int id;

    function new;
        stats = new;
        id = count++;
    endfunction

    function BusTran copy;
        copy = new;           // Конструктор для копії
        copy.addr = addr;     // Заповнення значень даних
        copy.crc = crc;
        copy.data = data;
```

```

        copy.stats = stats.copy;    // Створення першого об'єкта
        id = count++;
    endfunction
endclass

```

20.14. Публічні або приватні об'єкти класів

Звичайні мови програмування, що використовують ідею ООП, припускають, що всі об'єкти класу є приватними (невидимими за межами класу), якщо вони не оголошені як `public`. Для побудови `testbench` необхідна більша гнучкість, тому в SystemVerilog всі об'єкти є публічними (`public`), якщо тільки вони не оголошені як приватні (`private`).

20.15. Спадкування й підкласи

Ключовою властивістю класів SystemVerilog є спадкування, що дозволяє створювати класи, засновані на інших аналогах. Таким чином, вони автоматично включають деякі члени батьківського класу.

Наприклад, у наступному коді (лістинг 20.17) `Circle` успадковує властивості класу `Shapes`. Частина базового класу завжди створюється першою і видаляється останньою. Частина підкласу конструюється останньою і видаляється першою.

Лістинг 20.17. Приклад спадкування класів.

```

class Shapes;
    // Кожний об'єкт повинен містити розмір і початок координат
    int total_size;
    int start_x, start_y;
    bit[7:0] color;
    // Для кожної фігури необхідні функції рисування і зміни кольору
    task draw();
    endtask
// Визначення коду в даному місці є необов'язковим
// для наслідуваних методів, оскільки тут вони є
// віртуальними та будуть переписані пізніше в класах-спадкоємцях
    task setcolor (bit [7:0] color);
        this.color = color;
    endtask
endclass

class Circle extends Shapes;
    bit [30:0] circle_d_data;
    function new(int size = 4);
        total_size = size * 3;
    endfunction
    task draw();
// Код, що виконує рисування кола, записується тут
        $display("drawing a circle...\n");
    endtask
endclass

```

Ключове слово `super` використовується з породженого класу для доступу до елементів батьківського класу (лістинг 20.18).

Лістинг 20.18. Застосування ключового слова `super`.

```
class Packet; // Батьківський клас
integer value;
function integer delay();
    delay = value * value;
endfunction
endclass
class LinkedPacket extends Packet; // Клас спадкування
integer value;
function integer delay();
    delay = super.delay() + value * super.value;
endfunction
endclass
```

Елемент класу може бути декларований на рівень вище або успадкований класом на один рівень вище. Отримати доступ на більш високий рівень ієрархії немає можливості. Наприклад, неприпустиме використання конструкції `super.super.count`. Підкласи або породжені аналоги - це класи, що успадковують поточний аналог.

Конструкція `super.new` повинна бути першою у виконуваному конструкторі. Це необхідно тому, що суперклас повинен бути ініціалізований до поточного класу. Якщо користувальницький аналог не реалізує ініціалізацію суперкласу, то компілятор додає конструкцію `super.new` автоматично.

Породжений клас переслідує досягнення двох цілей. Перша - розширити функціональність базового класу, друга - скоротити, обмежити або уточнити функціональність базового класу.

Віртуальні класи не можуть бути використані прямо для створення об'єктів, але можуть бути застосовані як базові класи. Всі змінні з них і методи (функції класів) сумісні з усіма об'єктами породжених класів.

20.16. Приклади використання масивів класів

20.16.1. Приклади асоціативних масивів.

Наступний приклад (лістинг 20.19) представляє використання визначника класу. У ньому застосовується асоціативний масив об'єктів класу з індексацією рядком `string`.

Коли копія нового класу заноситься в масив, то в ньому реально зберігається визначник на нього. При цьому можна отримати два варіанти моделювання. Перший виконується без виклику методу (функції класу) `new` між кожним присвоєнням значення масиву. Значення, що буде отримано, є поточним значенням класу. При другому виконанні реалізується виклик методу (функції класу) `new` між кожним присвоєнням значення масиву. Виконання функції щоразу створює новий показник. Кожний елемент у масиві має різний визначник і являє собою значення, що зберігається в класі в зазначений час.

У результаті моделювання масиву визначників класів будуть привласнені три різних значення.

Лістинг 20.19. Використання асоціативних масивів об'єктів класів.

```
module top;
  class nam_dat;
    string name;
    int data;
  endclass:nam_dat

  class men_reg;
    // Асоціативний масив nam_dat зі строковим індексом string
    nam_dat reg_fields[string];
    task add_field (input nam_dat field);
      reg_fields[field.name] = field;
    endtask:add_field
  endclass:men_reg
  nam_dat field;
  men_reg foo = new;
  string s;
  initial begin
    field = new;                // Створення поля First
    field.name = "Verilog";
    field.data = 99;
    // Занесення значення field у динамічний масив foo
    foo.add_field(field);
    `ifdef FIELD
      field = new;
    `endif
    field.name = "Systemverilog"; // Створення поля Second
    field.data = 199;
    foo.add_field(field);
    `ifdef FIELD
      field = new;
    `endif
    field.name = "VHDL";        // Створення поля Third
    foo.add_field(field);
    if ( foo.reg_fields.first( s ) )
      do
        $display( "%s : %d\n", s, foo.reg_fields[ s ] );
        while (foo.reg_fields.next(s) );
      end
  end
endmodule
```

20.17. Контрольні завдання

1. Створити клас, що визначає інтерфейс процесора. Клас повинен містити поля:

instruction [17:0], in_port [7:0], interrupt, address [9:0], port_id [7:0], write_strobe,
out_port [7:0], read_strobe, interrupt_ack.

2. Розробити клас, що описує транзакції для передачі і читання даних з пам'яті FIFO. Клас повинен містити поля:

width - розрядність шини даних;

depth = $\log_2(n)$, де n - глибина пам'яті;

Data [width-1, 0] - шина даних;

wr_en - сигнал дозволу запису;

rd_en - сигнал дозволу читання;

sinit - ініціалізація пам'яті;

Dout[width-1:0] - вихід даних;

full - ознака того, що пам'ять заповнена повністю;

empty - ознака того, що пам'ять порожня;

data_count[depth-1:0] - кількість даних, записаних в пам'ять;

конструктор, що ініціалізує розмір пам'яті та розрядність шини;

методи для передачі інформації в пам'ять, читання результатів, визначення кількості даних у пам'яті.

3. Для класу, представленого в завданні 2, створити функцію копіювання даних сорту.

20.18. Література

1. IEEE Std. 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog™ Hardware Description Language 1995. ISBN 0-7381-3065-6. IEEE Product No. WE94418-TBR.

2. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).– IEEE Standard Verilog Hardware Description Language.– IEEE Computer Society Sponsored by the Design Automation Standards Committee.– Published by The Institute of Electrical and Electronics Engineers, Inc.– 3 Park Avenue, New York, NY 10016-5997, USA.– 2001.– 791 p.

3. IEEE Std 1800-2005.– IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language.– Sponsor: Design Automation Standards Committee of the IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group.– Published by The Institute of Electrical and Electronics Engineers, Inc.– 3 Park Avenue. New York, NY 10016-5997, USA.– 2005.– 664 p.

4. SystemVerilog 3.1a. Language Reference Manual. – Accellera's Extensions to Verilog®.– Accellera Organization, Inc. 1370 Trancas Street #163, Napa, CA 94558, USA.– 2004.
5. Spear C. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, 2nd ed.– Springer-Verlag, New York, Inc. Secaucus, NJ, USA.– 2006.– 429 p.
6. Mintz M., Ekendahl R. Hardware Verification with System Verilog. An Object-Oriented Framework.– New York : Springer.– 2007.– 314 p.
7. Bergeron J. Writing Testbenches using SystemVerilog.– Springer US.–2006.– 414 p.
8. Bergeron J., Cerny E., Hunter A., Nightingale A. Verification Methodology. Manual for SystemVerilog.– Springer US.–2005.– 510 p.
9. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas. 101 Common Coding Errors and How to Avoid Them.– New York : Springer.– 2007.– 214 p.
10. Vijayaraghavan Srikanth, Ramanathan Meyyappan. A Practical Guide for SystemVerilog Assertions.– Springer US.– 2005.– 334 p.
11. Haque F., Michelson J., Khan K. The Art of Verification with System Verilog Assertions.– Verification Central. First ed.– 2006.

21. СЕРЕДОВИЩЕ ВЕРИФІКАЦІЇ

Представлено моделі, методи і типові HDL-процедури, засновані на широких можливостях мови System Verilog конструювати ефективні рішення для верифікації цифрових проектів, описаних на системних мовах опису апаратури.

Ціль - створення технологічних процедур, маршрутів і структур верифікації, орієнтованих на пошук семантичних помилок, що змінюють функціонування пристрою та приводять проект у непрацездатний стан щодо заданої специфікації.

Завдання - вивчення найбільш технологічних процедур верифікації шляхом використання широких можливостей мови System Verilog, орієнтованих на перевірку апаратних рішень проєктованих цифрових систем на кристалах.

Джерела: [5-11] - методи тестування і верифікації різних типів цифрових структур; [1-11] - технології програмування процедур для створення середовища верифікації проєктів на основі System Verilog.

21.1. Методологія та інфраструктура верифікації SoC

Для створення середовища верифікації використовуються такі принципи:

1. Псевдовипадкові тестові послідовності.
2. Функціональне покриття.
3. Поділ testbench на шари з використанням транзакторів.
4. Загальні конструкції testbench для всіх тестів.
5. Код, орієнтований на створення тестів, що підтримує рішення, відмінні від testbench.

21.1.1. Псевдовипадкові тестові послідовності.

Випадковий (псевдовипадковий) тест (рис. 21.1) завжди покриває більший простір станів, чим вузьконаправлений або детермінований. При завершенні тестування можна використовувати спеціальні тести, для верифікації областей, недоступних або неперевірених випадковим тестом.

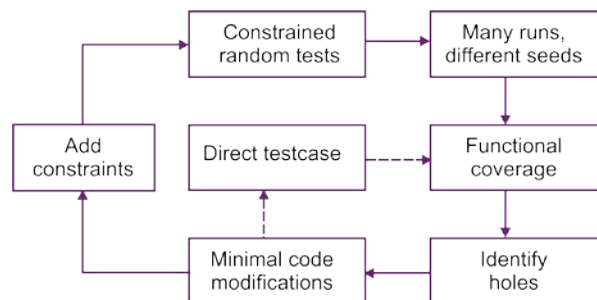


Рис. 21.1. Взаємодія тесту і покриття

Основні типи помилок, що виявляються випадковими тестами, - це невірні шляхи передачі даних, можливо, з помилками розрядності.

Нижче представлені вихідні дані про проект, які необхідно враховувати при створенні або виборі тестів:

1. Конфігурація пристрою (Device configuration).
2. Конфігурація середовища (Environment configuration).
3. Вхідні дані.
4. Виключення в протоколах.
5. Затримки.
6. Помилки і виключення при подачі некоректних вхідних сигналів.

Випадковий тест використовується зі зворотним зв'язком (рис. 21.2), що дозволяє оцінити повноту покриття тестом функціональності та скоригувати генерування наступних вхідних наборів. Для реалізації даної стратегії існує «coverage-driven verification». Подібні зворотні зв'язки широко використовуються в промислових системах моделювання, Magellan (Synopsys, 2003).

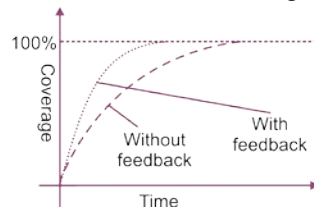


Рис 21.2. Процес тестування з використанням зворотних зв'язків і без них

21.1.2. Компоненти testbench.

Testbench - спеціалізована структура операційного пристрою, реалізована в HDL-кодi, призначена для тестування і верифікації цифрового проекту за допомогою засобів моделювання, асерційної надмірності, а також компонентів, що забезпечують керування, спостереження й ухвалення рішення про технічний стан об'єкта, що перевіряється. Він включає згенеровані вручну або автоматично вхідні і вихідні дані, що описують ідеальну модель пристрою та складові основи testbench (рис. 21.3).

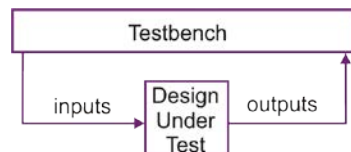


Рис.21.3. Testbench як середовище верифікації

Блок testbench заснований на використанні безлічі функціональних моделей шини Bus Functional Model (BFM). Якщо реальний пристрій підключається до шин AMBA, USB, PCI і SPI, можна будувати конструктивні компоненти testbench, які будуть генерувати тести і аналізувати результати моделювання (рис. 21.4). Тут не використовується детальна, синтезована модель проекту.

Замість цього застосовується високорівневий транзактор (перетворювач), що підкоряється протоколу шини тестування, але виконується значно швидше. У випадку створення прототипу пристрою на основі FPGA або емулятора BFM не буде синтезований.

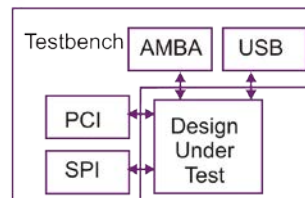


Рис. 21.4. Компоненти testbench

21.1.3. Багаторівневий testbench (layered testbench).

Багаторівневий підхід - це ключова концепція сучасної методології верифікації. Незважаючи на видимі ускладнення процесу верифікації, проблема перевірки пристрою істотно спрощується завдяки поділу завдання на кілька простих підзавдань, що становлять ієрархію інфраструктури тестування, які розглядаються далі від нижнього рівня ієрархії до вищого.

1) Сигнальний і командний рівень (рис. 21.5).

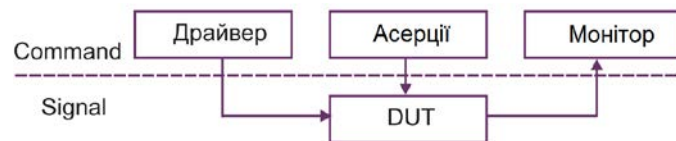


Рис. 21.5. Сигнальний і командний рівні

Найнижчим є рівень сигналів, що являє собою тестований пристрій Design Under Test (DUT) і сигнали, які з'єднують його з testbench. Наступний рівень - командний (command layer). Входи DUT керуються драйвером (driver), що виконує окремі команди, такі як читання і запис шини. Виходи DUT спостерігаються монітором (monitor), що отримує зміни сигналів і формує із них команди. Асерції (assertions) належать обом рівням (command/signal layer), вони спостерігають за окремими сигналами та здійснюють пошук змін протягом виконання всього коду.

2) Функціональний рівень (рис. 21.6).

Функціональний шар керує командним рівнем.

Блок-Агент (agent block), називаний також транзактором, отримує високорівневі транзакції, такі як пряме читання та запис у пам'ять, а потім розбиває їх на окремі команди. Вони передаються драйверу, а також посилають в scoreboard, що пророкує результат транзакції.

Перевірочний блок отримує значення від монітора та відправляє для аналізу в scoreboard.

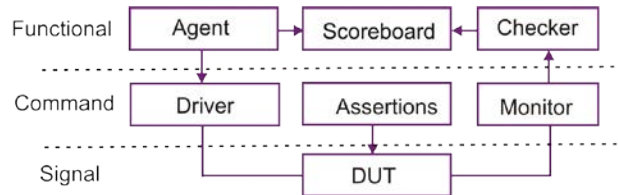


Рис. 21.6. Сигнальний, командний і функціональний рівні

3) Рівень сценарію (рис. 21.7).

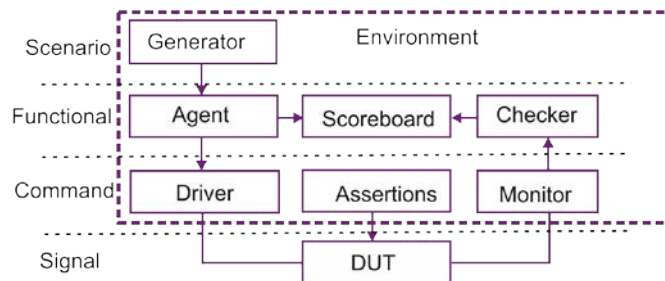


Рис. 21.7. Сигнальний, командний, функціональний і рівень сценарію

Функціональний шар управляється генератором сценарного рівня. Тут формується завдання з параметрами обмежень для псевдовипадкової генерації тестів. Блоки, виділені пунктиром, створюються на початку етапу розробки. Вони можуть модифікуватися, доповнюватися функціями, але ніколи не змінюються при виконанні окремого тесту.

4) Рівень тесту створює повну і замкнуту інфраструктуру верифікації проєктованого виробу разом з функціональним покриттям, сценарієм тестування, що перевіряється пристроєм і компонентами, які обслуговують процес тестування (рис. 21.8).

Таким чином, на підставі розглянутих структур верифікації можна зробити такі практично важливі узагальнення. Компоненти кожного рівня ієрархії являють собою транзактори. Подібна організація відкриває широкі можливості по обробці даних і повторному використанню компонентів. Тестовий рівень ієрархії складається з тестових компонентів або шарів: 1) Test - компонент верхнього рівня задає обмеження генератору послідовностей і конфігурує режими моделювання при кожному запуску. 2) Рівень сценаріїв містить генератор (generator), що створює послідовності з обмеженнями, отриманими від test, а також всі сценарії по псевдовипадковій генерації тестів. 3) Функціональний рівень формує структури даних для їхньої обробки та передачі на командний (command) рівень. Тут перевіряється правильність даних, які видаються DUT, за допомогою компонентів: агент (agent), лічильник (scoreboard) і аналізатор (checker).

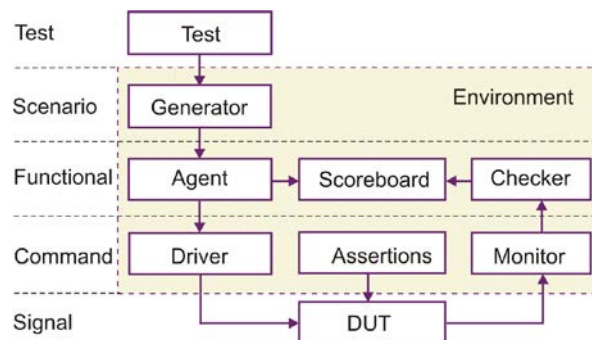


Рис. 21.8. Повний testbench з усіма шарами ієрархії

Агент використовується для обробки даних, переданих драйверу (driver). Лічильник верифікаційного середовища містить високорівневу «золоту» модель проекту. Він отримує однакові з DUT тестові послідовності та визначає коректну реакцію DUT. Аналізатор порівнює вихідну інформацію з DUT, доставлену за допомогою монітора (monitor) з очікуваною еталонною реакцією, отриманою від лічильника. 4) Командний рівень забезпечує зв'язок з DUT. Драйвер управляє значеннями на входах DUT. Монітор стежить за вихідною інформацією з DUT і відправляє її аналізатору. Зовнішні асерції використовуються як складова частина проекту командного рівня. 5) Сигнальний (signal) рівень містить верифікований пристрій і інтерфейс вхідних, вихідних сигналів. Асерції - блоки, що додаються у вихідний код проекту для спостереження і керування поведінкою моделі проекту. Вони можуть бути представлені операторами if для повідомлень про помилки, що проявляються в процесі тестування DUT. Асерції створюються розроблювачем або можуть бути взяті з існуючих бібліотек для перевірки типових функцій. Фірма Synopsys поставляє систему моделювання VCS з бібліотекою SystemVerilog-асерцій.

SystemVerilog дозволяє реалізувати два види асерцій: миттєві і паралельні. Миттєві запускаються відразу при передачі керування відповідному оператору у вихідному коді. Паралельна асерція перевіряє вираз по кожному імпульсу синхросигналу та спрацьовує у випадку фіксації хибності виразу. Асерції, поміщені в код DUT, називаються внутрішніми, у тестове середовище - зовнішніми. Вони стежать за сигналами, переданими між DUT і тестовим середовищем. Мова SystemVerilog має можливість для підключення проекту до модуля, що містить зовнішні асерції. Підвищення рівня абстракції проекту для ранньої верифікації - це шлях до скорочення часу виходу придатної продукції на ринок. Статистика компанії Synopsys показує вартість помилок у кристалі: дефекти через помилки проектування становлять 82%; дефекти від помилок специфікації - 47%; дефекти від помилок багаторазово використовуваних IP-cores компонентів - 14%.

Модель верифікаційного середовища. В HDL Verilog базова функціональна одиниця - це модуль, що містить структуру і інтерфейси для підключення до інших модулів. Ключова можливість модуля - формування границі навколо структури, що дає засоби спілкування із зовнішнім світом через однозначно визначений інтерфейс. Модуль - це конструкція, що залежить від конкретної мови. Компонент - незалежний об'єкт, що має властивість інкапсуляції. Наприклад, компоненти будуються з модулів SystemVerilog і System, але можуть бути побудовані на класах в C++ або entity/architecture в VHDL. Ефективні повторно використовувані компоненти - це «чорні ящики», що зв'язуються між собою через інтерфейси. Транзакція - це кількість активності, що виникає в моделі за певний період часу, або одиниця інформації при передачі керування або даних між двома компонентами (запис/читання шини, передача пакета даних), що має структуру: розміщення, добування і передача.

Ієрархію верифікаційного середовища можна представити у вигляді організації компонентів, зображеної на рис. 21.9. Внутрішня частина є DUT компонент RTL-рівня абстракції, до якого підключається верифікаційне оточення TLM-рівня за допомогою верифікаційних компонентів-транзакторів. Роль кожного з них - конвертування потоку даних TLM-рівня для їхнього сприйняття на RTL-рівні, і навпаки. Оточення (див. рис. 21.9) - набір високорівневих компонентів із транзакційними інтерфейсами, які надають все необхідне для функціонування DUT. Компоненти оточення - генератори тестів ведучих (masters) і ведених (slaves) блоків.

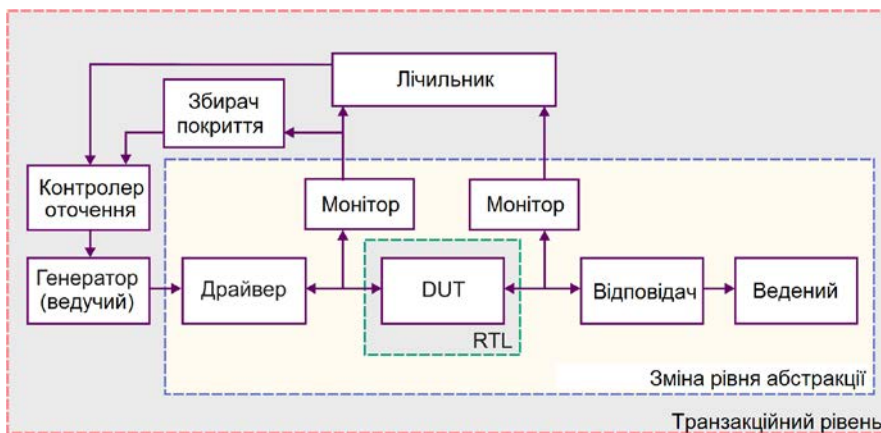


Рис. 21.9. Організація верифікаційного оточення

Генератори тестових впливів створюють потік транзакцій для моделювання DUT. Вони можуть бути псевдовипадковими, детермінованими або змішаними; самостійними або мати керуючі сигнали; незалежними або синхронізованими. Найпростіший генератор виконує псевдовипадкову функцію синтезу тестів, які відправляє драйверу. Генератор сценаріїв створює детерміновані або змішані послідовності, спрямовані на ініціалізацію специфічних

функцій тестуємим пристроєм. Ведучий - двонаправлений компонент, відсилає запити та приймає відгуки, ініціалізує активність, аналізує відгуки для завдання наступного сценарію. Ведений - двонаправлений компонент, обробляє запити та повертає відгуки. Аналізатори (scoreboard - золота модель, coverage collector - повнота тестового покриття транзакцій, адресного простору, кількості помилок) отримують інформацію про верифікаційне середовище та роблять висновок про правильність і закінчення процесу верифікації. Контролер формує потік даних у верифікаційному середовищі і керує її активністю, пересилає дані від лічильника і колектора покриття компонентам оточення, запускає та зупиняє генератор тестів.

Виділення операційного та аналізуючого доменів. Компоненти об'єктно-орієнтованого верифікаційного середовища зручно розглядати в приналежності до двох доменів: операційного і керуючого (аналізуючого) (рис. 21.10). Операційний домен - це набір компонентів, який створює середовище для виконання процесу тестування пристрою DUT. Генератори тестових впливів, а також відповідач і драйвер створюють сигнали, що управляють моделюванням. Керуючий домен містить набір компонентів (монітори, лічильники, збирачі покриття, контролер), які виконують аналіз процесу верифікації і ухвалюють рішення щодо технічного стану проекту.

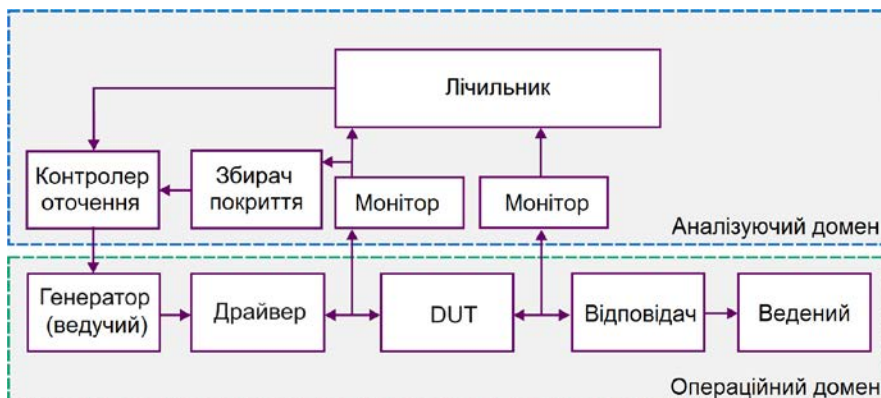


Рис. 21.10. Виділення операційного і аналізуючого доменів

Збирач покриття (coverage collector) і лічильник підключаються до виходу монітора через порт аналізу (рис. 21.11). Функція покриття - визначити повноту або якість тесту щодо перевіреної безлічі станів тестованого об'єкта. Якщо при заданій 100% повноті щодо введених станів або несправностей верифікаційне середовище не видало помилок, робиться висновок про справність моделі проєктованого виробу.

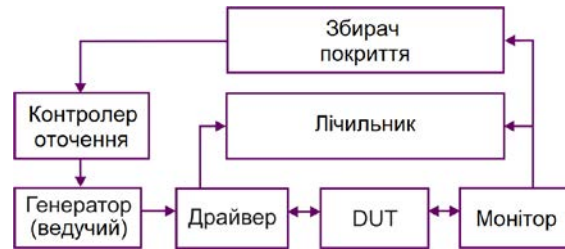


Рис. 21.11. Приклад середовища зі збирачем покриття

Узагальнення наведених структур верифікації створює ESL (Electronic System Level design) технологію, засновану на SystemVerilog, що дозволяє отримати високорівневу TLM-модель (TLM - Transaction Level Model), яка емулює поведінку SoC. TLM-структура дає можливість почати процес розробки і верифікації задовго до наявності проекту RTL-рівня або кремнієвого прототипу пристрою. TLM-модель працює на рівні функціональних запитів і передач пакетів даних, що задають системну поведінку проекту. Верифікаційні середовища, створені на System Verilog TLM, інтегруються в будь-яку архітектуру: SoC TLM або SoC HDL. Це дає можливість, використовуючи верифікаційне середовище, визначити оптимальну архітектуру програмно-апаратних засобів розглянутого проекту. TLM-методологія прискорює інтеграцію верифікаційних рішень і дозволяє швидко змінювати цільові проекти, використовуючи TLM-шаблон як легко модифіковану базову платформу. Рівень абстракції TLM набагато вищий, ніж RTL, який у деталях відображає внутрішньоблокові схемні стани, наносекундну точність передач і побітові зміни шин. Як наслідок, використання TLM прискорює програмну верифікацію та HW/SW ко-верифікацію в 1000 і більше разів у порівнянні з RTL.

Ряд закордонних компаній має цікаві розробки в області системної верифікації: Cadence, Synopsys, Mentor Graphics, Aldec. Остання з них є лідером в області створення апаратних і програмних рішень для верифікації проектів, реалізованих на технологіях ASIC (Application-Specific Integrated Circuits) і FPGA (Field Programmable Gate Arrays). Один з напрямків - це забезпечення повної підтримки актуального стандарту SystemVerilog'2009. Цікавий маршрут проектування від Aldec представлений на рис. 21.12, що інтегрує кращі технології, у тому числі від Cadence, Synopsys, Mentor Graphics.

Інфраструктура використовує досвід і технології від провідних компаній планети в області тестування і верифікації цифрових проектів. Тут основні компоненти: сімейство мовних засобів для введення проекту; компілятори з мов опису апаратури; засоби налагодження проекту; орієнтовані на користувача засоби керування проектом і компоненти візуалізації результатів моделювання.

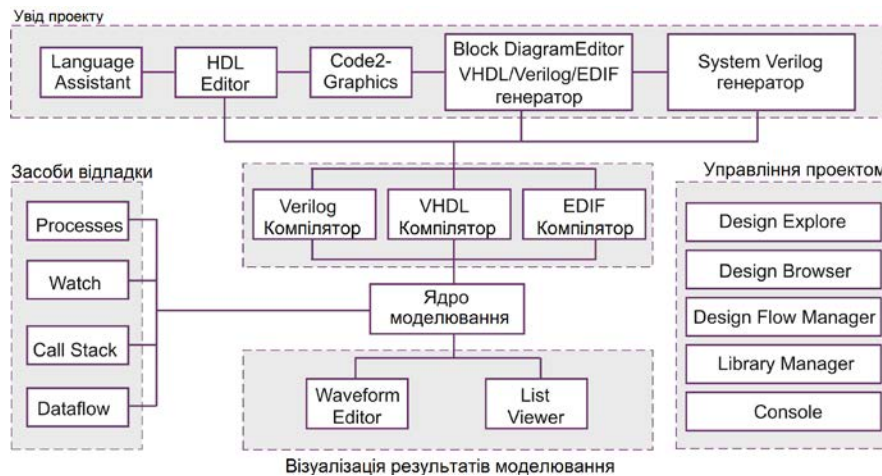


Рис. 21.12. Інфраструктура верифікації проектів компанії Aldec

21.2. Програми (Programs)

Модуль - основна конструкція для створення компонентів апаратури в Verilog і SystemVerilog. Він може містити ієрархічні підключення інших модулів, ліній, декларації завдань і функцій, процедурні оператори в блоках `always` і `initial`. Ці конструкції добре підходять і для опису апаратної моделі. При побудові `testbench` акцент робиться не на деталі апаратного рівня, лінії, структурну ієрархію та міжз'єднання, а на створення середовища верифікації. Увага приділяється ініціалізації змінних, синхронізації виконання операцій, усуненню можливості виникнення гонок між проектом і `testbench`, автоматизації генерування вхідних сигналів, повторному використанню існуючих бібліотечних моделей. Для рішення завдань в SystemVerilog вводиться спеціальна конструкція - `program`, завдання якої:

1. Забезпечити точку входу для виконання `testbench`.
2. Сформувати область, що є оболонкою даних, завдань і функцій.
3. Забезпечити синтаксичний зміст, що описує порядок формування і подачі тестів, а також аналіз результатів тестування.
4. Усунути можливість появи гонок між проектом і `testbench`.

Програма служить чітким роздільником проекту від `testbench`. Разом із блоками синхронізації програма створює вільні від гонок взаємозв'язки між проектом і `testbench` і дозволяє використовувати рівні абстракції, такі як циклічне (`cycle`) моделювання і транзакційний рівень обміну даними. Конструкції абстракції і моделювання в SystemVerilog спрощують створення і використання `testbench`.

Можливість створення програмних екземплярів і виконання їхнього індивідуального підключення дозволяє використовувати програми як узагальнену модель. Програма може містити декларації типів і даних, підпрограми з'єднань із проектом, один або кілька потоків процедурного коду. Зв'язки між

проектом і testbench реалізуються за допомогою механізмів з'єднання портів в SystemVerilog, включаючи інтерфейси.

Синтаксис програм:

```
program [static | automatic] program_identifier
    [ parameter_port_list ] list_of_ports | [ list_of_port_declarations ];
    [ timeunits_declaration ] { program_item }
endprogram [: program_identifier ]
або
program program_identifier ( .* );
    [ timeunits_declaration ] { program_item }
endprogram [: program_identifier ]
```

До program_item ставиться декларація портів і опис таких конструкцій:

```
{ attribute_instance } continuous_assign // оператори безперервного призначення
| { attribute_instance } module_or_generate_item_declaration // копії модулів або
// оператори generate
| { attribute_instance } initial_construct // конструкції initial або final
| { attribute_instance } final_construct
| { attribute_instance } concurrent_assertion_item // Паралельні асерції
| loop_generate_construct | conditional_generate_construct | generate_region
```

Наприклад, декларації заголовків програм з використанням портів і інтерфейсу:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
    initial ...
endprogram
або
program test ( interface device_ifc );
    initial ...
endprogram
```

У програмі можна розміщувати завдання, функції, класи і блоки initial, але не можна використовувати блоки always. Програма має одну або кілька точок входу та закінчує свою роботу після завершення останнього блоку initial. Замість always, якщо необхідно нескінченне виконання операцій, можна використовувати «initial forever», а завершувати роботу за допомогою системного завдання \$exit.

21.3. Кроки моделювання систем в SystemVerilog

В Verilog більшість подій виконується в активний період (Active region). Також існують періоди для неблокуючих присвоєнь і PLI інтерфейсу. В SystemVerilog пропонується новий розподіл тимчасових слотів (рис. 21.13, табл. 21.1). Перший період тимчасового слота є підготовчим - prepare region, використовуваним для testbench. У цей момент збираються значення сигналів перед активним періодом. Наступний за ним активний період (active region) призначений для виконання подій проекту. У даний період обробляється RTL і вентильний код, а також генератор синхросигналу. Третій - період спостереження (observed region) служить для аналізу асерцій - спеціалізо-

ваних конструкцій формальної верифікації. Останнім є період реактивації (reactive region), у якому знову виконується testbench. Зазначений порядок періодів не є строгим. Події з періодів спостереження та реактивації можуть виникати раніше від подій проекту з активного періоду.

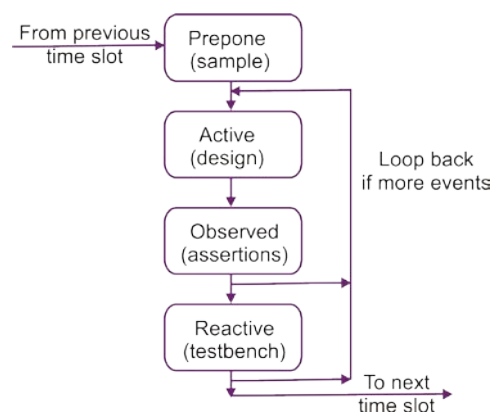


Рис. 21.13. Етапи моделювання в межах одного тимчасового кроку

Таблиця 21.1. Области планування робіт (primary scheduling regions)

Період	Дія
Підготовчий (prepone)	Зчитування значень сигналів до активності проекту. Для входів testbench
Активації (active)	Моделювання коду проекту, описаного в моделях
Спостереження (Observed)	Обчислення асерцій
Реактивації (reactive)	Обчислення коду в testbench, описаного в програмі

21.4. Синхронізація тестових послідовностей (Stimulus Timing)

21.4.1. Блок синхронізації (clocking).

Це новий блок для опису синхросигналів і вимог до тимчасових параметрів і синхронізації моделі. Він спрощує використання часу. Блок clocking використовує cycle-based методологію й дозволяє писати моделі на більш високому абстрактному рівні. Замість того щоб звертати увагу на сигнали і транзакції в часі, блок тестування може оперувати поняттям такту і транзакції. Залежно від середовища (environment) testbench може включати один або кілька блоків clocking, кожний з яких містить один синхросигнал і необов'язкові додаткові сигнали.

Блок clocking відокремлює опис тимчасових параметрів і синхронізації від структурних, функціональних і процедурних елементів. Таким чином, час для отримання і керування сигналами блоку стає більш ясным і залежить від відповідного блоку. Безліч ключових операцій може бути записано більш стисло без прямого використання синхронізації або вказівки тимчасових

параметрів. До таких операцій відносяться: події синхронізації, отримання вхідних значень, драйвери синхронізації.

Синтаксис:

```
[ default ] clocking [ identifier ] clk_event;
    { default input clk_skew | output clk_skew | input clk_skew output clk_skew;
      | input [ clk_skew ] | output [ clk_skew ] | input [ clk_skew ] output [ clk_skew ] |
inout
      signal_identifier [ = expression ] { , signal_identifier [ = expression ] }; }
endclocking [ : clocking_identifier ]
```

Тут *identifier* - ім'я блоку, а подія синхронізації *clk_event* може бути представлена у формі

@ *identifier* | @ (*event_expression*)

Фігурні дужки вказують на повторювані елементи, а квадратні - на необов'язкові.

Зсув *clk_skew* визначає затримку стосовно події синхросигналу, коли може бути отримано або передано значення сигналу. Вхідний зсув завжди негативний, оскільки відноситься до часу до фронту синхронізації, у той час як вихідний сигнал відноситься до періоду після фронту (рис. 21.14). Зсув може бути заданий у форматі:

posedge | negedge [# *delay_value* | # (*mintypmax_expression*)] | # *delay_value* | # (*mintypmax_expression*)

Загальний зсув визначається за допомогою ключового слова *default*.

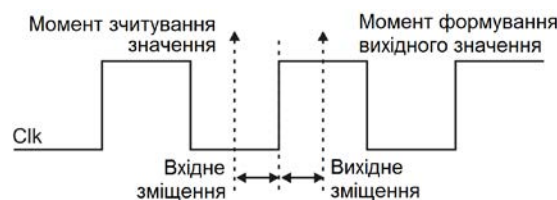


Рис. 21.14. Моменти надходження вхідних тестів і зчитування результатів

Лістинг 21.1,а представляє блок, сигнали якого за замовчуванням управляються переднім фронтом *clk*. Входи мають зсув на один крок (1 step) - значення на вхід повинне бути подане за один крок моделювання до надходження переднього фронту синхросигналу. Вихідні значення зчитуються по задньому фронту. У другому прикладі (лістинг 21.1,б) не зазначений фронт, тому за замовчуванням подією синхронізації буде вважатися будь-яка зміна сигналу *clk*. Виходи, як і в попередньому випадку, управляються заднім фронтом.

Перший рядок у третьому прикладі (лістинг 21.1,в) оголошує блок синхронізації з іменем *bus*, що управляється переднім фронтом синхросигналу *clock1*. Друга лінія вказує, що за замовчуванням всі вхідні сигнали мають зсув, рівний 10ns, а вихідні - 2ns. Третя лінія задає керовані синхронізацією вхідні сигнали: *data*, *ready* і *enable*; останній з них пов'язаний з ієрархічним сигналом *top.mem1.enable*.

У четвертій лінії додається інформація про сигнал `ask`, керований заднім фронтом, що перевизначає задане за замовчуванням значення. Останній рядок додає вхідний сигнал `addr` і перевизначає заданий за замовчуванням зсув таким чином, що сигнал подається за один крок (`step`) до переднього фронту синхросигналу. Це означає, що сигнал стає стабільним відразу перед появою фронту.

Останній приклад (лістинг 21.1,г) представляє блок синхронізації `dram`, де значення на вхід `address` повинне бути подане за `1ps` до появи події на `clk`. Для інших входів цей параметр дорівнює 5 одиницям часу. На виходах інформація повинна бути готова через 6 одиниць часу після перемикання `clk`.

Лістинг 21.1. Приклади синхроблоків.

```
a) clocking ck1 @(posedge clk);
    default input #1step output negedge; // виходи управляються заднім фронтом
clk
    input ... ;
    output ... ;
endclocking
б) clocking ck2 @(clk); // Фронт не описаний
    default input #1step output negedge;
    input ... ;
    output ... ;
endclocking
в) clocking bus @(posedge clock1);
    default input #10ns output #2ns;
    input data, ready, enable = top.mem1.enable;
    output negedge ack;
    input #1step addr;
endclocking
г) clocking dram @(clk);
    input #1ps address;
    input #5 output #6 data;
endclocking
```

21.4.2. Інтерфейси і блоки синхронізації.

Час між `testbench` і модулем повинен бути ретельно спланованим.

Інтерфейс може містити блоки синхронізації, по одному на кожний синхродомен. Можна описати використовувану за замовчуванням синхронізацію, застосовуючи оператор `default`. За замовчуванням вхідний сигнал надходить до початку виконання операцій проекту, а вихідний передається назад у проект під час поточного тимчасового слота.

При описі синхронізації `testbench` може використовувати вираження виду `@my_interface.cb` замість визначення точного значення синхросигналу або фронту. У прикладі (лістинг 21.2) блок синхронізації `cb` декларує, що всі сигнали в блоці активні по передньому фронту синхросигналу. Напрямок сигналу залежить від `modport`. Таким чином, `request` - вихід блоку `TEST modport` і `grant` - вхід в `modport`.

Лістинг 21.2. Інтерфейс із блоком синхронізації.

```
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    logic reset;
    clocking cb @(posedge clk); // Declare cb
        output request;
        input grant;
    endclocking
    modport TEST (clocking cb, // Use cb
        output reset);
    modport DUT (input request, reset, output grant);
endinterface
```

Блоки синхронізації є оболонкою для безлічі сигналів, що використовують той самий синхросигнал. Застосування таких блоків в інтерфейсах дозволяє значно скоротити кількість коду, необхідного для підключення testbench.

Більш того, оскільки напрямки сигналів у блоках синхронізації інтерфейсів стають залежними від testbench, а не від тестованого пристрою, то декларація modport може відповідним чином описати один із цих напрямків. Програма testbench може використовувати порти інтерфейсу залежно від призначення кожного блоку синхронізації. Лінії інтерфейсу в блоці синхронізації повинні мати той же напрямок, який описано в блоці для сторони testbench (modport test), або бути інверсними стосовно сторони DUT (modport dut). Описана ситуація представлена в лістингу 21.3.

Лістинг 21.3. Використання блоків синхронізації для тестових блоків.

```
interface bus_A (input clk);
    logic [15:0] data;
    logic write;
    modport test (input data, output write);
    modport dut (output data, input write);
endinterface

interface bus_B (input clk);
    logic [8:1] cmd;
    logic enable;
    modport test (input enable);
    modport dut (output enable);
endinterface

program test( bus_A.test a, bus_B.test b );
    clocking cd1 @(posedge a.clk);
        input a.data;
        output a.write;

        inout state = top.cpu.state;
    endclocking
    clocking cd2 @(posedge b.clk);
        input #2 output #4ps b.cmd;
```

```

        input b.enable;
    endclocking
    initial begin
        // Оператори програми
        ...
        // Можуть бути використані cd1.a.data , cd2.b.cmd , etc...
    end
endprogram

```

Компоненти проекту описуються у вигляді модулів, testbench - у вигляді програми, interfaces - представляє сигнали, які можуть бути використані для зв'язку між окремими блоками. Усі компоненти об'єднуються в модулі верхнього рівня (лістинг 21.4).

Лістинг 21.4. Об'єднання компонентів системи в модулі верхнього рівня.

```

module top;
    logic phi1, phi2;
    bus_A a(phi1);
    bus_B b(phi2);
    test main( a, b );
    cpu cpu1( a );
    mem mem1( b );
endmodule

```

Інакше, в описаному прикладі (лістинг 21.3) блок clocking міг бути реалізований з використанням ієрархічних імен з інтерфейсу:

```

clocking cd1 @(posedge a.clk);
    input data = a.data;
    output write = a.write;
    inout state = top.cpu.state;
endclocking
clocking cd2 @(posedge b.clk);
    input #2 output #4ps cmd = b.cmd;
    input enable = b.enable;
endclocking

```

21.4.3. Опис подій через блок синхронізації. Події синхронізації блоку clocking стають безпосередньо доступними через використання імені блоку clocking, безвідносно до реальної події синхронізації, декларованої у блоці clocking.

Наприклад:

```

clocking dram @(posedge phi1);
    inout data;
    output negedge #1 address;
endclocking

```

Події синхронізації clocking-блоку можуть бути використані для опису події: @(dram) , що еквівалентно запису @(posedge phi1).

Оператор `##` служить для опису затримки на зазначену кількість подій синхросигналу або синхротактів - циклічної затримки (Cycle delay: `##`). Синтаксис операторів затримки синхротактів:

```
## integral_number | ## identifier | ## ( expression )
```

Вираз `expression` може бути будь-яким з SystemVerilog, значення якого обчислюється як позитивне ціле число.

Циклічна затримка відповідає синхронізації за замовчуванням. Якщо для поточного модуля, інтерфейсу або програми не описана синхронізація за замовчуванням, то компілятор видає повідомлення про помилку. Приклади опису затримок:

```
## 5;      // очікування 5 синхроподій з використанням синхронізації за замовчуванням
## (j + 1); // очікування j+1 синхроподій з використанням синхронізації
           // за замовчуванням
```

Один блок `clocking` може бути зазначений як синхронізація за замовчуванням для всіх операцій синхрозатримки в модулі, інтерфейсі або програмі. Синтаксис оператора специфікації синхронізації за замовчуванням:

```
// Опис у модулі існуючого блоку синхронізації
    default clocking clocking_identifier ;
// Декларація блоку синхронізації
    [ default ] clocking [ clocking_identifier ] clocking_event ;
    { clocking_item }
    endclocking [ : clocking_identifier ]
```

Тут `clocking_identifier` повинен бути іменем блоку `clocking`.

Тільки один `clocking`-блок може бути описаний як блок синхронізації за замовчуванням у програмі, модулі або інтерфейсі. Запис виразу `default clocking` більш одного разу в одній програмі або модулі може призвести до помилки компіляції.

Конструкція `default clocking` має силу тільки в границях області, що містить її опис. Ці границі включають модуль, інтерфейс або програму, що містить декларацію, точно так само, як вкладений модуль або інтерфейс. Це не стосується реалізації копії модуля та інтерфейсу. Лістинги 21.4 і 21.5 представляють декларацію блоку синхронізації за замовчуванням всередині програми в першому випадку, з використанням для цієї мети зовнішнього блоку - у другому.

Лістинг 21.5. Декларація синхронізації за замовчуванням.

```
program test( input bit clk, input reg [15:0] data );
    default clocking bus @(posedge clk);
        inout data;
    endclocking
    initial begin
        ## 5;
        if ( bus.data == 10 )
            ## 1;
```

```

        else
            ...
        end
    endprogram

```

Лістинг 21.6. Використання існуючого блоку clocking для синхронізації за замовчуванням.

```

module processor ...
    clocking bus @(posedge clk1); ... endclocking
    clocking bus @(negedge clk2); ... endclocking
    module cpu( interface y );
        default clocking bus ;
        initial begin
            ## 5; // use bus => (posedge clk1)
            ...
        end
    endmodule
endmodule

```

Всі входи блоку clocking (input або inout) є тестами, які синхронізуються подією синхросигналу. Коли сигнал з'являється у виразі, його значення замінюється вхідним, яке було подано в останній точці. Коли сигнал з'являється в декількох clocking, кожен блок clocking зчитує значення сигналу по своїй події синхронізації.

21.4.4. Синхронізація подій і керування ними.

Керування порядком виконання операторів включає керування затримкою:

```
# delay_value | # ( mintypmax_expression ) ,
```

і керування подією

```
@ identifier | @ ( expression ) | @* | @ ( * ) | @ sequence_instance
```

де sequence_instance - послідовність, нова форма опису складних подій в SystemVerilog, розглядається нижче.

Також для керування за допомогою повторюваних подій може бути використаний цикл

```
repeat ( expression ) event_control
```

Вираз expression може включати кілька подій, перелічених через оператор or або кому:

```
event or event | event , event
```

Допускається використання ключових слів, що описують передній і задній фронти: posedge і negedge.

Вирази, які застосовуються в керуванні подіями, можуть позначати входи блоку clocking (input або inout) або його сектора. Сектор може включати динамічні індекси, які обчислюються, коли обчислюється @-вираз. Нижче наведені деякі приклади виразів синхронізації:

1. Очікування зміни сигналу `ask_1`, що синхронізується блоком `clocking ram_bus`:
`@(ram_bus.ack_1);`
2. Очікування наступної події блоку `clocking ram_bus`:
`@(ram_bus);`
3. Очікування переднього фронту сигналу `ram_bus.enable`:
`@(posedge ram_bus.enable);`
4. Очікування заднього фронту однобітового сектора `dom.sign[a]` :
`@(negedge dom.sign[a]);`
5. Очікування переднього фронту `dom.sig1` або зміни сигналу `dom.sig2`:
`@(posedge dom.sig1 or dom.sig2);`
6. Очікування заднього фронту `dom.sig1` або переднього фронту `dom.sig2`:
`@(negedge dom.sig1 or posedge dom.sig2);`

SystemVerilog до подієвого керування `@` додає специфікатор `iff`. Подія виразу буде обчислюватися, якщо вираз після `iff` буде мати значення «істина». В описаному нижче прикладі, якщо `enable = 1`,

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
    always @(a iff enable == 1)
        y <= a;    //тригер-засувка в режимі прийому даних
endmodule
```

Так само, як і в подібних подієвих виразах, операторові `iff` може передувати вираз `or`. Пріоритет може бути зазначений більш явно за допомогою дужок.

Якщо вираз вказує на `input` або `inout`, синхронізований блоком `clocking`, то оператор подієвого контролю використовує синхронні значення сигналу, які є входними послідовностями, отриманими в момент появи події синхронізації. Вираз може також управлятися іменним блоком синхронізації.

Змінні, використовувані із подієвим контролем, можуть бути будь-якого інтегрального типу або рядком, простими змінними або посилальними `ref` (змінними, переданими по посиланню), вони можуть бути елементами масиву, асоціативним масивом або об'єктом класу. Подієвий вираз повинен повертати одиничне значення. Члени об'єкта або об'єднаних елементів можуть бути будь-якого типу, але результат виразу повинен бути скалярним значенням.

Якщо подієвий вираз звертається до визначника (`handle`) простого об'єкта, то подія фіксується, коли значення, що записується в змінну, не дорівнює попередньому. У подієвих виразах допускається використання невіртуальних і вбудованих методів або системних функцій, якщо вони повертають одиничне значення і метод визначений як функція, а не завдання. Зміни значень елементів об'єктів або розмірів динамічних масивів призводять до повторного обчислення подієвого виразу:

```
real AOR[];                // Динамічний масив речовинних значень
byte stream[$];            // Черга з байтів
initial wait(AOR.size() > 0) ...; // Очікування, поки під масив не буде виділена пам'ять
initial wait($bits(stream) > 60)...; // Очікування, поки загальне число бітів у потоці стане
```



```

// більше ніж 60
Packet p = new; // Пакет 1
Packet q = new; // Пакет 2
initial fork
    @(p.status); // Очікування зміни статусу status пакета 1
    @q; // Очікування зміни визначника q
    # 10 q = p; // Зміна @q.
// тепер очікування @(p.status) для пакета 2 не буде відрізнятися від пакета 2
join

```

21.4.5. Керування сигналами в інтерфейсі.

Асинхронні сигнали, такі як reset, передаються через інтерфейс без затримки. Для синхронізації сигналів в testbench використовуються оператори @ і wait (лістинг 21.7).

Лістинг 21.7. Синхронізація сигналів.

```

program automatic test(bus_if.TB bus);
    initial begin
        @bus.cb; // Виконання триває по активному фронту
        // блоку синхронізації
        repeat (3) @bus.cb; // Очікування трьох активних фронтів
        @bus.cb.grant; // Очікування будь-якої зміни сигналу grant
        @(posedge bus.cb.grant); // Очікування переднього фронту сигналу
        @(negedge bus.cb.grant); // Очікування заднього фронту сигналу
        wait (bus.cb.grant==1); // Очікування виконання умови
        // Затримки не буде, якщо умова має вже значення ІСТИНА
        @(posedge bus.cb.grant or negedge bus.reset); // Очікування подій
        // для декількох сигналів
    end
endprogram

```

Наступний приклад (лістинг 21.8) передає значення grant в DUT з arb.

Лістинг 21.8. Синхронізована передача значень через інтерфейс.

```

program test(arb_if.TEST arbif);
    initial begin
        $monitor("@%0d: grant=%h", $time, arbif.cb.grant);
        #50;
    end
endprogram
module arb(arb_if.DUT arbif);
    initial begin
        arbif.grant = 1; // @ 0ns
        #12 arbif.grant = 2; // @ 12ns
        #18 arbif.grant = 2; // @ 30ns
    end
endmodule

```

На тимчасовій діаграмі (рис. 21.15) показано, що в програмі сигнал `arbif.cb.grant` отримує значення до фронту синхросигналу. При зміні входу інтерфейсу, з появою фронту синхросигналу (time 30ns), значення не передається в `testbench` до наступного синхроциклу.

Нижче представлена найпростіша версія арбітра тесту програми.

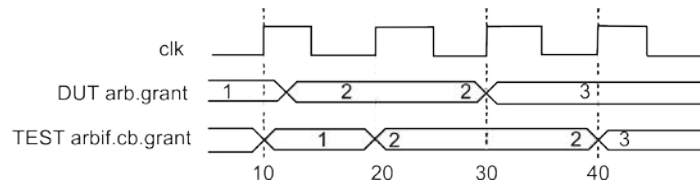


Рис. 21.15. Отримання значень у синхронізованому інтерфейсі

Лістинг 22.9. Testbench, що використовує інтерфейс, у якому визначений блок синхронізації.

```
program automatic test (arb_if.TEST arbif);
    initial begin
        arbif.cb.request <= 2'b01;
        $display("@%0d: Drove req=01", $time);
        repeat (2) @arbif.cb;
        if (arbif.cb.grant != 2'b01)
            $display("@%0d: a1: grant != 2'b01", $time);
    end
endprogram : test
```

Будь-який сигнал синхронізації, подібний до `request`, повинен мати дві прис-тавки: ім'я інтерфейсу (`arbif`) та ім'я блоку синхронізації(`cb`), якщо цей сигнал не використовується більш ніж в одному блоці синхронізації. Таким чином, `arbif.cb.grant` - правильне ім'я, а `arbif.grant` - ні. Інтерфейсні сигнали повинні завжди отримувати значення в неблокуючих операторах `<=`, так що сигнал проекту не змінюється безпосередньо після виконання присвоєння, оскільки `testbench` змінюється тільки в період реактивації, у той час як код проекту змінюється тільки в період активності. Якщо `testbench` формує сигнал `arbif.cb.request` у момент часу 100ns, одночасно й відповідно до операцій `arbif.cb`, `clocking`, `@(posedge clk` змінюється сигнал `request`. Але якщо `testbench` спробує змінити `arbif.cb.request` у момент часу 101ns, то ця зміна не буде поширюватися до наступного синхроциклу. Таким чином, драйвери завжди синхронізовані. У прикладі з лістинга 21.10 `arbif.grant` отримує значення з модуля та може отримувати значення в блокуючому операторі. Якщо `testbench` управляє синхронізованими сигналами інтерфейсу по активному фронту, значення передається негайно в проект, тому що за замовчуванням вихідна затримка дорівнює `#0`. У протилежному випадку зміни будуть вико-нані по такому активному фронту синхросигналу:

```
// Передача значень сигналам інтерфейсу
busif.cb.request <= 1;      // синхронне неблокуюче присвоєння
busif.cb.cmd <= cmd_buf;   // синхронне неблокуюче присвоєння
```

Приклад (лістинг 21.10) інформує, що відбудеться при завданні сигналів у тестовому блоці і передачі їх через інтерфейс у модуль арбітра arb.

Лістинг 21.10. Передача тестових значень через синхронний інтерфейс.

```
program test(arb_if.TEST arbif);
    initial begin
        # 2 arbif.cb.request <= 3;    // @ 2ns
        #10 arbif.cb.request <= 2;    // @ 12ns
        #18 arbif.cb.request <= 1;    // @ 30ns
        #50 finish;
    end
endprogram
module arb(arb_if.DUT arbif);
    initial
        $monitor("@%0d: req=%h", $time, arbif.request);
endmodule
```

Якщо необхідно почекати два синхротакти до виконання присвоєння сигналу, можна використовувати конструкцію repeat (2) @bus.cb; або затримку циклу ##2. Ця конструкція працює тільки із сигналами в синхронізованому блоці, оскільки необхідно знати, який синхросигнал використовується для затримки (рис. 21.16):

```
// Передача значень сигналам інтерфейсу
##2 arbif.cb.request <= 0;    // Очікування два такти до виконання присвоєння
##3;                          // Неправильно, необхідно використовувати разом із присвоєнням
```

21.4.6. Двонаправлені сигнали в інтерфейсі.

В Verilog, якщо необхідно управляти двонаправленим сигналом і зв'язати його з регістровою змінною із процедурного коду, то застосовується оператор безперервного призначення, щоб з'єднати змінні типів даних reg з wire. В SystemVerilog синхронні двонаправлені сигнали в інтерфейсі реалізуються за допомогою безперервного оператора присвоєння. Коли в програмі створюється змінна класу лінія, SystemVerilog генерує тимчасову змінну для її керування. Програма читає безпосередньо з wire значення, які передаються із всіх драйверів. Код проекту в модулях також використовує змінні класу register і безперервні оператори присвоєння (лістинг 21.11).

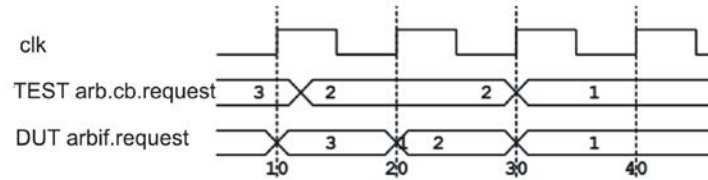


Рис. 21.16. Передача значень у синхронний інтерфейс

Лістинг 21.11. Двонаправлені сигнали в програмі і інтерфейсі.

```

interface master_if (input bit clk);
    wire [7:0] data;                // Двонаправлені сигнали
    clocking cb @(posedge clk);
        inout data;
    endclocking
    modport TEST (clocking cb);
endinterface

program test(master_if mif);
    initial begin
        mif.cb.data <= 'z;          // Тристабільна шина
        @mif.cb;
        $displayh(mif.cb.data);     // Читання даних із шини
        @mif.cb;
        mif.cb.data <= 7'h5a;        // Передача значень на шину
        @mif.cb;
        mif.cb.data <= 'z;          // Звільнення шини
    end
endprogram

```

21.5. Приклад. Four-Port ATM Router

Чотирипортовий ATM маршрутизатор. Його схема підключення до testbench для виконання тестування зображена на рис. 21.17.

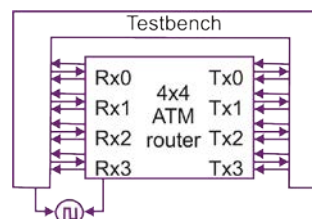


Рис. 21.17. Інтерфейс ATM маршрутизатор і testbench

Наступний приклад представляє групи ліній, які використовуються для з'єднання модуля рівня RTL з testbench.

Лістинг 21.12. Заголовок моделі маршрутизатора АТМ без інтерфейсу.

```
module atm_router(
    // 4 x Level 1 Utopia ATM layer Rx Interfaces
    output Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
    input [7:0] Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
    input Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
    output Rx_en_0, Rx_en_1, Rx_en_2, Rx_en_3,
    input Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
    // 4 x Level 1 Utopia Tx Interfaces
    output Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
    output [7:0] Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
    output Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
    output Tx_en_0, Tx_en_1, Tx_en_2, Tx_en_3,
    input Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
    // Miscellaneous control interfaces
    input rst, clk);
    ...
endmodule
```

Блок верхнього рівня, що поєднує компоненти системи, наведений у лістингу 21.13.

Лістинг 21.13. Модуль верхнього рівня без використання інтерфейсу.

```
module top;
    bit clk;
    always #5 clk = !clk;
    wire Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
        Rx_en_0, Rx_en_1, Rx_en_2, Rx_en_3,
        Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
        Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
        Tx_en_0, Tx_en_1, Tx_en_2, Tx_en_3,
        Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3, rst;
    wire [7:0] Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
        Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3;
    atm_router a1(Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
        Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
        Rx_en_0, Rx_en_1, Rx_en_2, Rx_en_3,
        Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
        Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
        Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
        Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
        Tx_en_0, Tx_en_1, Tx_en_2, Tx_en_3,
        Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
        rst, clk);
    test t1 (Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
        Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
```

```

Rx_en_0, Rx_en_1, Rx_en_2, Rx_en_3,
Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,
Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
Tx_en_0, Tx_en_1, Tx_en_2, Tx_en_3,
Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,
rst, clk);

```

```
endmodule
```

Лістинг 21.14 представляє тестовий модуль. Порти і лінії в ньому займають основну частину коду.

Лістинг 21.14. Використання портів testbench.

```

module test(
// 4 x Level 1 Utopia ATM layer Rx Interfaces
input Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
output reg [7:0] Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
output reg Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
input Rx_en_0, Rx_en_1, Rx_en_2, Rx_en_3,
output reg Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,
// 4 x Level 1 Utopia Tx Interfaces
input Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
input [7:0] Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
input Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
input Tx_en_0, Tx_en_1, Tx_en_2, Tx_en_3,
output reg Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,
// Miscellaneous control interfaces
output reg rst, input clk);
initial begin
    // Reset the device
    rst <= 1;
    Rx_data_0 <= 0;
    ...
end
endmodule

```

Використання інтерфейсу спрощує опис моделі. Застосовуються два інтерфейси Rx і Tx з конструкціями modports і clocking (рис. 21.18).

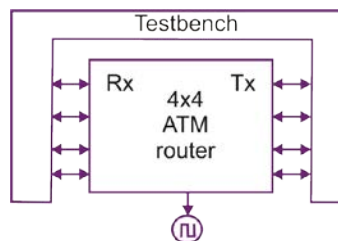


Рис. 21.18. Використання інтерфейсу для підключення testbench

Опис портів пристрою з використанням інтерфейсів представлено в лістингу 21.15.

Лістинг 21.15. Rx- і Tx-інтерфейси.

```
// Rx- інтерфейс із конструкціями modport і блоками синхронізації
interface Rx_if (input logic clk);
    logic [7:0] data;
    logic soc, en, clav, rclk;
    clocking cb @(posedge clk);
        output data, soc, clav;      // Напрямок відносно testbench
        input en;

    endclocking : cb
    modport DUT (output en, rclk,
        input data, soc, clav);
    modport TB (clocking cb);
endinterface : Rx_if

// Tx- інтерфейс із конструкціями modport і блоками синхронізації
interface Tx_if (input logic clk);
    logic [7:0] data;
    logic soc, en, clav, tclk;
    clocking cb @(posedge clk);
        input data, soc, en;
        output clav;

    endclocking : cb
    modport DUT (output data, soc, en, tclk,
        input clk, clav);
    modport TB (clocking cb);
endinterface : Tx_if

// ATM router model with interface using modports
module atm_router(Rx_if.DUT Rx0, Rx1, Rx2, Rx3,
    Tx_if.DUT Tx0, Tx1, Tx2, Tx3,
    input logic clk, rst);

    ...

endmodule

// Модель верхнього рівня з інтерфейсами
module top;
    bit clk, rst;
    always #5 clk = !clk;
    Rx_if Rx0 (clk), Rx1 (clk), Rx2 (clk), Rx3 (clk);
    Tx_if Tx0 (clk), Tx1 (clk), Tx2 (clk), Tx3 (clk);
    atm_router a1 (Rx0, Rx1, Rx2, Rx3,
        Tx0, Tx1, Tx2, Tx3, clk, rst);
    test t1 (Rx0, Rx1, Rx2, Rx3,
        Tx0, Tx1, Tx2, Tx3, clk, rst);
endmodule : top

21.5.1. ATM testbench з інтерфейсом.
```

Лістинг 21.16. представляє частину testbench, що охоплює вхідні комірки, які йдуть від портів ТХ-маршрутизатора.

Лістинг 21.16. Testbench-інтерфейс із блоком синхронізації.

```

program test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
            Tx_if.TB Tx0, Tx1, Tx2, Tx3,
            input logic clk, output logic rst);
    bit [7:0] bytes[ATM_CELL_SIZE];
    initial begin
        // Reset the device
        rst <= 1;
        Rx0.cb.data <= 0;
        ...
        receive_cell0();
        ...
    end
task receive_cell0();
    @(Tx0.cb);
    Tx0.cb.clav <= 1; // Assert ready to receive
    wait (Tx0.cb.soc == 1); // Wait for Start of Cell
    for (int i=0; i<ATM_CELL_SIZE; i++) begin
        wait (Tx0.cb.en == 0); // Wait for enable
        @(Tx0.cb);
        bytes[i] = Tx0.cb.data;
        @(Tx0.cb);
        Tx0.cb.clav <= 0; // Deassert flow control
    end
endtask : receive_cell0
endprogram : test

```

21.6. Контрольні запитання і завдання

1. Що таке програма? Чим програма відрізняється від модуля?
2. Які об'єкти в testbench відносяться до рівня сигналів?
3. Які об'єкти в testbench відносяться до рівня команд?
4. Які об'єкти в testbench відносяться до функціонального рівня?
5. Які об'єкти в testbench відносяться до рівня сценарію?
6. У чому відмінність псевдовипадкових і детермінованих методів генерації тестів для цифрових систем?
7. Дайте характеристику одиночним константним і функціональним несправностям цифрових пристроїв.

22. АСЕРЦІЇ

Представлено моделі, методи і типові асерційні HDL-процедури спостереження та керування процесом налагодження програмного коду, засновані на широких можливостях мови System Verilog конструювати ефективні рішення для скорочення часу тестування цифрових проектів, описаних на системних мовах опису апаратури.

Ціль - дослідження технологій створення процедур, маршрутів і структур асерційної верифікації, орієнтованих на пошук семантичних помилок, що змінюють функціонування пристрою і приводять проект у непрацездатний стан щодо заданої специфікації.

Завдання - визначення найбільш технологічних структур і процедур асерційної верифікації шляхом використання широких можливостей мови System Verilog, орієнтованих на ранню перевірку апаратних рішень проєктованих цифрових систем на кристалах.

Джерела: [1-5] - методи й структури асерційної верифікації різних типів цифрових структур; [6-11] - технології програмування процедур для створення середовища асерційної верифікації проєктів на основі System Verilog.

22.1. Послідовності (Sequence)

SystemVerilog має можливість опису специфічних послідовностей булевих виразів із явними тимчасовими взаємовідносинами між ними. Для визначення збігу послідовностей булеві вирази обчислюються в кожній точці зчитування значень, обумовленій синхросигналом, пов'язаним з розглянутою послідовністю.

Найпростіша послідовна поведінка є лінійною. Лінійна послідовність - це кінцевий список булевих виразів SystemVerilog у порядку зростання часу. Кажуть, що лінійна послідовність мала місце на кінцевому інтервалі наступних одна за одною синхроподій, якщо перший логічний вираз приймає значення «істина» на першій синхроподії, другий логічний вираз отримує значення «істина» при надходженні другої синхроподії, і так далі, аж до останнього логічного виразу, який повинен прийняти значення «істина» на останній синхроподії. Прикладом простої лінійної послідовності є булевий вираз. Його присутність на зазначеній синхроподії означає, що він приймає значення «істина» у цей момент часу.

Більш складне поводження в SystemVerilog описується за допомогою послідовності (sequence), яка визначається як упорядкована сукупність булевих виразів, що включає безліч нулів, кінцеве або нескінченне число лінійних послідовностей. Якщо хоча б одна з них, що входять в безліч, має місце в кінцевому інтервалі послідовних синхроподій, то вважається, що послідовність відбулася на даному інтервалі часу. Сам базовий послідовний вираз - це порядок - «а іде за b на наступному синхротакті» на SystemVerilog може виглядати так:

a ##1 b

У цьому прикладі «##1» означає затримку на один синхротакт між наступним булевим виразом. В SystemVerilog кожний елемент послідовності може бути або булевим виразом, або іншою послідовністю. У термінах послідовності булевий вираз - це спрощений вироджений випадок послідовності довжиною в 1. Таким чином, вираз конкатенації послідовностей

s1 ##1 s2

означає, що послідовність s2 починається на синхротакт після завершення послідовності s1, як це зображено на рис. 22.1. У прикладі послідовностей, що перекриваються

s1 ##0 s2

упорядкований набір s2 почне своє виконання в синхротакті, у якому послідовність s1 завершить свою функцію (рис. 22.2).

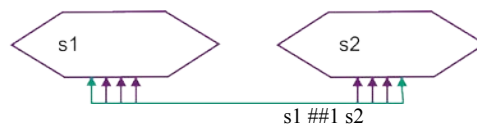


Рис. 22.1. Конкатенація послідовностей

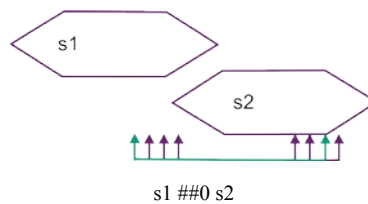


Рис.22.2. Перекриття послідовностей

Хоча послідовності корисні для опису тимчасових відносин між виразами, важливим є їхнє оформлення у вигляді елементів мови таким чином, щоб вони могли бути повторно використані або на них можна було посилалися. SystemVerilog визначає такий елемент мови - sequence. Крім перелічених можливостей, послідовності можуть бути використані при побудові інших упорядкованих наборів або як частина властивостей асерцій.

22.1.1. Булеві вирази в послідовностях.

Використовувані в послідовностях вирази обчислюються на основі значень змінних, які представлені у виразі. Результат обчислення має булевий тип і інтерпретується аналогічно оператору if. Існує ряд обмежень для виразів, які застосовуються в послідовностях і асерціях.

У булевих виразах, що складають послідовності, не дозволені: нецілочисельні дані (shortreal, real і realtime), рядки, події, chandle, класи, асоціативні і динамічні масиви. Вирази дозволяють включати виклики функцій, але існують деякі семантичні обмеження. Функції, які з'являються у виразах, не можуть містити вихідний аргумент посилального типу ref (дозволяються const

ref). Функції повинні бути автоматичні або захищені від статичної інформації. Масиви фіксованого розміру, запаковані чи ні, можуть бути використані повністю, у вигляді діапазонів або по окремих елементах. Індекс може бути константою, параметром або змінною. Наступний приклад показує деякі припустимі форми порівняння компонентів структур і об'єднань. Нехай виконане визначення:

```
typedef int [4] array;
typedef struct { int a, b, c, d } record;
union { record r; array a; } p, q;
```

Тоді наступний вираз є легальним:

```
p.a == q.a
або
p.r == q.r
```

Приклади використання масивів у виразах:

```
logic [7:0] array [0:15], array[0:15];
array == array;
array != array;
array[i] >= array[j];
array[i][j+:2] == array[k][ m-m-:2];
(array[i] & (~array[j])) == 0;
```

Змінні, які з'являються у виразах, повинні бути статичними змінними проекту або викликами функцій, що повертають значення припустимих типів. Статичні змінні, оголошені в програмах, інтерфейсах або блоках синхронізації, можуть бути також використані в асерціях. Якщо посилання на статичну змінну декларується в завданні, то значення такої змінної може бути прочитане в будь-який момент часу, незалежно від виклику функції. Дозволені всі оператори, які можуть бути використані із припустимими в послідовностях типами даних, крім операторів присвоєння, інкремента і декремента. SystemVerilog містять оператори з мови Cі, такі як +=, ++ and ---і. Вони не можуть бути використані в обчисленні виразів, які застосовуються в асерціях.

22.1.2. Синтаксис.

Послідовність може бути задекларована в модулі, інтерфейсі, програмі, блоці синхронізації, пакеті та області одиниці компіляції.

SystemVerilog синтаксис для декларування й створення копій послідовностей:

```
// Декларація
sequence sequence_identifier [ ( [ tf_port_list ] ) ];
    { assertion_variable_declaration } // Будуть розглянуті нижче
    sequence_expr ;
endsequence [ : sequence_identifier ]
```

```
// Створення копії послідовності
sequence_identifier [ ( [ list_of_arguments ] ) ]
```

Необов'язковий список формальних параметрів дозволяє описувати послідовності, як загальні тимчасові взаємини. Формальні аргументи замінюються актуальними, які передаються в послідовність при створенні копії. Наприклад, послідовність

```
sequence seq1 (a, b);
    a ##2 b;
endsequence
```

представляє набір із двох виразів. Коли створюється копія послідовності seq1(e,f),

актуальні аргументи e і f замінюють формальні аргументи (a і b, відповідно), визначені в послідовності. Таким чином, тимчасові взаємини між e і f будуть виглядати як

```
e ##2 f
```

Табл. 22.1 поєднує операції, які можуть бути використані в послідовностях. Послідовності можуть містити синхроподії. Послідовність може декларуватися за допомогою необов'язкових формальних аргументів. Коли створюється копія послідовності, реальні аргументи замість формальних передаються в послідовність. Актуальні аргументи можуть бути представлені: ідентифікатором, виразом, виразом подієвого контролю, верхнім рівнем, як \$(upper range as \$). Змінні, використовувані в послідовностях, що не є формальними аргументами, обробляються відповідно до області видимості для границь, у яких послідовність декларується:

```
sequence s1;
    @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
    @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
    @(negedge clk) g ##1 h ##1 i;
endsequence
```

У прикладі послідовності s1 і s2 обчислюються по передньому фронту clk, а s3 - по задньому.

Приклад декларації послідовності, що містить аргументи:

```
sequence s20_1(data,en);
    (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
Endsequence
```

Послідовність s20_1 не включає опису синхронізації. У цьому випадку синхросигнал повинен бути успадкований від зовнішнього джерела, такого як оператор property або assert. До послідовності можна звертатися по імені. Може бути використане ієрархічне ім'я. При описі затримок у послідовностях можуть застосовуватися оператори повторення:

1) Послідовне повторення (consecutive repetition ([*])). Означає повторення послідовності. Наприклад,

(a ##1 b [*3] ##1 c) відповідає (a ##1 b ##1 b ##1 b ##1 c).

Таблиця 22.1. Операції послідовностей

Операції	Синтаксис	Пояснення
Конкатенація	seq1 ##1 seq2	Послідовність seq2 починається з наступного синхротакту після завершення seq1
Перекриття	seq1 ##0 seq2	Послідовність seq2 починається в тому ж синхротакті, в якому закінчується послідовність seq1
Виявлення кінця послідовності	seq1 ##1 seq2.ended	Послідовність seq2 завершується на один синхротакт після завершення послідовності seq1, незалежно від моменту початку послідовності seq2
Повторення	seq1[*n:m]	Послідовність seq1 повторюється мінімум n і максимум m разів. Може формувати множинний збіг послідовностей
Виявлення першого збігу	first_match(seq1)	Якщо послідовність seq1 має безліч збігів, використовується перший з них та ігноруються ті, що залишилися
АБО	seq1 or seq2	Формує послідовність, яка повертає значення "істина", якщо виявлена хоча б одна з послідовностей seq1 чи seq2
І	seq1 and seq2	Формує послідовність, яка повертає значення "істина", якщо одна послідовність з'являється до або після появи іншої послідовності Matches
Перетинання	seq1 intersect seq2	Повертає значення "істина" в синхроциклах, де виконуються обидві послідовності seq1 і seq2
Уточнення умовою	cond throughout seq	Умова cond має значення "істина" для кожного синхротакту seq
У рамках	seq1 within seq2	Послідовність seq1 починається разом або після початку послідовності seq2 і закінчується разом з нею або до її завершення

Для опису кінцевого, але необмеженого числа повторень використовується символ долара (\$). Наприклад,

a ##1 b [*1:\$] ##1 c

2) Повторення переходом (goto repetition ([->)). Як операнди частіше використовуються булеві вирази, а не послідовності. Описують ітеративні збіги булевих виразів за синхроподіями, які не обов'язково є послідовними і закінчуються на останній ітерації, що повертає значення «істина». Наприклад,

(a ##1 b [->2:10] ##1 c) відповідає (a ##1 ((!b[*0:\$] ##1 b) [*2:10]) ##1 c).

3) Непослідовна побудова (nonconsecutive repetition ([=))). Подібно до повторення переходом, за єдиною відмінністю, що перевірка повторень не обов'язково повинна закінчуватися на останньому виразі, що повертає значення «істина». Наприклад,

(a ##1 b [=2:10] ##1 c) відповідає (a ##1 ((!b [*0:\$] ##1 b) [*2:10]) ##1 !b[*0:\$] ##1).

22.1.3. Локальні змінні в послідовностях.

Послідовності описують тимчасові взаємини між сигналами. Проте, щоб описати складну поведінку, необхідно явно зберігати значення в деяких точках послідовності таким чином, щоб до неї можна звертатися пізніше. Розглянемо послідовність «коли дані надходять у конвеєр, вони будуть передані назовні через 3-5 синхротактів і збільшені на 1»:

```
sequence s5;
    int d;
    @(posedge clk) (d = data, valid) ##[3:5] (dout == (d+1));
ends0equence
```

Коли дані надходять на конвеєр і кількома (3-5) тактами пізніше на виході з конвеєра, dout мітки правильного значення сигналу повинні дорівнювати вхідним даним, збільшеним на 1. Значення сигналу, що надходить у конвеєр, зберігається, а через 3-5 синхротактів вихід конвеєра dout повинен бути рівним вхідному значенню, збільшеному на 1. Захоплення даних, виконане у послідовності, дозволяє уникнути побудови окремого автомата або іншого допоміжного коду для захоплення даних і необхідності редагування його для кожної послідовності. Також немає необхідності повідомляти інші інструменти, що допоміжний код не є частиною проекту, що спрощує використання таких інструментів.

22.1.4. Подія послідовності (Sequence events).

Копія конструкції sequence може бути використана в подієвому виразі для керування виконанням процедурних операторів, заснованих на позитивному результаті порівняння послідовностей. Це дозволяє точці завершення іменної послідовності sequence перемикає множинні дії в інших процесах. Коли копія sequence описується в подієвому виразі, то процес, що виконує подієвий контроль, повинен блокуватися, поки зазначена послідовність не досягне кінцевої точки. Послідовність досягає точки завершення щоразу, коли відбувається збіг із цілою послідовністю. Процес продовжує виконання observe region, у якому кінцева точка була виявлена.

Приклад використання послідовності sequence для подієвого керування:

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence
program test;
    initial begin
        @ abc $display( "Saw a-b-c" );
        L1 : ...
    end
```

endprogram

Коли іменна послідовність *abc* досягне кінцевої точки, блок *initial* у програмі *test* буде розблокований. Виконається оператор *\$display*, що виводить рядок "Saw a-b-c", і наступний оператор з міткою *L1*. У цьому випадку кінцева точка послідовності *sequence* розглядається як перемикання, що є подією.

Для синхронізації використовується точка виходу з послідовності, а точка входу ігнорується. Аргументи повинні бути статичними, автоматичні змінні призведуть до помилки.

22.1.5. Послідовності і події послідовностей (Level-sensitive sequence controls).

Виконання процедурного коду може бути припинено, поки статус завершення послідовності не прийме значення істина. Це реалізується шляхом використання чутливого до рівня оператора *wait* спільно (у кон'юнкції) з вбудованим методом *triggered*, що повертає статус поточного кінця іменної послідовності (лістинг 22.1.). Метод *triggered* повертає значення «істина», якщо дана послідовність досягає кінцевої точки в певний момент часу (поточний крок моделювання), і «неправда» - у протилежному випадку. Статус перемикання для послідовності встановлюється в області спостереження (*observe region*) і триває протягом тимчасового кроку, що залишився (до наступного часу моделювання).

Лістинг 22.1. Використання методів послідовностей.

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence
sequence de;
    @(negedge clk) d ##[2:5] e;
endsequence
program check;
    initial begin
        wait( abc.triggered || de.triggered );
        if( abc.triggered )
            $display( "abc succeeded" );
        if( de.triggered )
            $display( "de succeeded" );
        L2 : ...
    end
endprogram
```

У представленому прикладі блок *initial* перевіряє оператором *wait* завершення послідовності *abc* або *de*. Коли один з обчислених виразів прийме значення «істина», оператор *wait* розблокує процес, виведе повідомлення та продовжить виконання наступних операторів, починаючи з мітки *L2*.

22.2. Асерції в SystemVerilog

Асерції SystemVerilog розроблені для надання можливості опису поведінки проекту в явній і однозначній манері. Асерції в основному використовую-

ються для перевірки поведження проекту. Крім того, вони можуть бути використані для отримання інформації про функціональне покриття і генерацію вхідних стимулів для перевірки. Асерції реалізуються подібно до інших блоків проекту й активні для всього процесу моделювання. Програма моделювання відслідковує зміни асерцій, щоб зібрати функціональне покриття даних.

SystemVerilog має вбудовані ефективні уніфіковані асерції, які можуть бути використані для моделювання і формальної верифікації. Основні переваги їхнього використання полягають в такому: 1) Легко налаштовуються, тому що будуються на подібній мові і синтаксисі. 2) Код асерцій - невеликий за розміром, завдяки автоматичному концептуальному розумінню керуючої логіки проекту. 3) Просте створення зв'язків між асерціями і testbench без необхідності розробки спеціального інтерфейсу. 4) Налагодження і керування виведеними повідомленнями, можливість опису декількох рівнів помилок. 5) Можливість зв'язку між Verilog і функціями мови C. 6) Можливість уникнути неузгодженості між моделюванням і формальним обчисленням з певною семантикою розподілу робіт. 7) Можливість підвищення продуктивності процесу верифікації.

Семантика SVA визначена так, що обчислення асерцій гарантує еквівалентність між моделюванням, яке засноване на події (event-based), і асерціями, заснованими на використанні синхротакту (cycle-based). Це гарантує, що різні інструменти будуть інтерпретувати поведження SVA тим самим способом. Крім того, уніфікація асерцій з кодом проектування і верифікації підвищує потужність інструмента асерцій, завдяки безперервному процесу. Зокрема, SystemVerilog дозволяє асерціям зв'язувати інформацію з testbench і дає можливість останньому реагувати на стани асерцій без використання додаткового інтерфейсу. SystemVerilog підтримує два типи асерцій: прямі (процедурні) і паралельні.

Прямі (миттєві) асерції використовують семантику моделювання подій і виконуються, як оператори в процедурному блоці. Прямі асерції, у першу чергу, призначені для верифікації в процесі моделювання проекту. Паралельні асерції засновані на семантиці синхронізації і захвату значень змінних у певні моменти часу. Однією із завдань асерцій в SystemVerilog є забезпечення загальної семантики, щоб вони могли бути використані для керування різними проектами та інструментами верифікації. Реальна потужність SVA, для моделювання і формальної верифікації, полягає в можливості короткого опису послідовного поведження і обчислення асерційних значень у дискретних точках процесу моделювання, зазвичай за синхротактом. Концепції і компоненти, що становлять паралельні асерції, можуть розглядатися як безліч шарів, з яких кожний будується на нижчележачому шарі (рис. 22.3).

Базова функція асерцій описує безліч поведжень, які повинні мати значення істини для даного проекту або компонента. Шар булевих виразів є найбільш базовим і описує значення елементів у конкретній точці часу, у той час як шар послідовностей будується на булевому рівні та описує часові взаємоз-

в'язки між елементами протягом періоду часу. Шар властивостей використовує послідовності для опису реального поведіння, а шар директив асерції зв'язує це поведіння з інструментами верифікації і проектування.

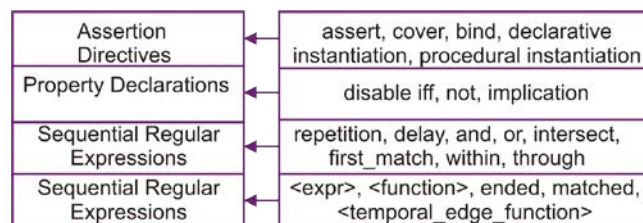


Рис.22.3. Шари побудови асерцій

22.2.1. Вибірка значень змінних (Sampling).

Для забезпечення стійкості між системою моделювання і інструментами формальної верифікації паралельні асерції в SystemVerilog використовують вибірку значень сигналів для обчислення виразів. Ці зразки визначені як значення сигналів наприкінці останнього кроку моделювання перед появою синхрподії. Таким чином, може бути отриманий передбачуваний результат, незалежно від внутрішньої реалізації черговості і обчислення подій симулятором.

Можливість виникнення гонок - це недолік семантики планування подієвого моделювання в Verilog, що дозволяє декільком подіям траплятися в заданий період моделювання. Оскільки інструменти синтезу і формальної верифікації використовують синхронний підхід до проекту, умови гонок завжди вирішуються відповідно до семантики зчитування значень для побудови асерцій в SystemVerilog. Це означає, що асерції дають користувачеві постсинтезний погляд на проект під час досинтезної RTL-верифікації, крім можливості виникнення різного поведіння проекту до та після синтезу.

22.3. Прямі (процедурні) асерції

Миттєві (прямі) асерції використовуються в процедурному коді в будь-якому місці блоків initial і always.

Синтаксис прямих асерцій:

```
assert ( expression ) [ [ pass_stmt ] else fail_stmt ];
```

Вираз обчислюється відразу, при виконанні оператора. Оператори, що втримуються в галузях pass і fail, якщо вони присутні, виконуються миттєво після обчислення виразу.

Процедурний код testbench може перевіряти значення сигналів проекту, змінних і виконувати необхідні дії, у випадку виявлення помилки. Наприклад, у проекті виконується перевірка запиту шини request і очікується, що відповідь буде отримана двома тактами пізніше. Тестування цієї ситуації можна виконати за допомогою оператора if:

```
// Перевірка сигналу за допомогою оператора if
bus.cb.request <= 1;
@bus.cb;
if (bus.cb.grant != 2'b01)
    $display("Error, grant != 1");
// Інша частина тесту
```

Оператор асерції більш компактний, ніж оператор if. При цьому значення умови є протилежним записуваному в if. Інакше кажучи, якщо умова має значення «істина», то виконання асерції було успішним:

```
// Проста процедурна асерція
bus.cb.request <= 1;
@bus.cb;
a1: assert (bus.cb.grant == 2'b01);
// інший код тесту
```

Якщо сигнал grant отриманий правильно (grant=2'b01), тестування буде продовжено. Інакше буде видане таке повідомлення:

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
```

Повідомлення говорить, що в лінії 7 файлу test.sv асерція top.t1.a1 почала своє виконання в момент 55ns для перевірки сигналу bus.cb.grant і відразу ж виявила помилку.

Процедурні асерції мають галузі then- і else-, а при необхідності в них можуть бути додані оператори:

```
// Створення користувальницького повідомлення про помилку
a1: assert (bus.cb.grant == 2'b01)
else $error("Grant not asserted");
```

Якщо сигнал grant не має очікуваного значення, то буде виведено відповідне повідомлення:

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
Grant not asserted
```

SystemVerilog має чотири функції для виведення повідомлень: \$info, \$warning, \$error і \$fatal, які можуть бути використані тільки в операторі асерції. Можна застосовувати гілку then для запису кількості успішних виконань оператора асерції:

```
// Створення користувальницького повідомлення про помилку
a1: assert (bus.cb.grant == 2'b01)
    grants_received++; // Кількість успішних виконань
else
    $error("Grant not asserted");
```

Оператори прямих асерцій виконують тестування зазначених умов у момент свого виконання в процедурному коді. Вираз інтерпретується тим же способом, що і умова в операторі if. Якщо значення виразу дорівнює X, Z або 0,

воно розглядається як «неправда» і асерція не пройшла (fail), інакше - «істина» і асерція пройшла (pass). Оператор, пов'язаний з успішним виконанням асерції, записується першим, називається pass-оператором і виконується, якщо значення виразу є «істина». Pass-оператор може, наприклад, записувати число успішних проходів у файл звіту (log), але може взагалі бути відсутнім. Якщо оператор для гілки успішного виконання асерцій не зазначений, то ніякі дії в цьому випадку виконуватися не будуть. Оператор, асоційований з віткою else, називається fail-оператор і виконується, якщо значення виразу є «неправда». Присутність цього оператора також є необов'язковим. Зазначені дії виконуються миттєво, після обчислення асерції. Необов'язкова мітка оператора створює іменну конструкцію, і вона може бути виведена за допомогою символу форматування %m. Наприклад,

```
assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");
```

Оскільки асерція - це оператор, що може приймати значення «неправда», то провал асерції повинен мати пов'язаний з ним рівень серйозності помилки. За замовчуванням, такий рівень дорівнює error. Задати або явно вказати рівень помилки можна, включаючи одне з наступних системних завдань, що реалізують рівень серйозності провалу асерції: \$fatal - фатальна помилка; \$error - помилка часу виконання; \$warning - попередження про надмірний час виконання завдання системою моделювання; \$info - означає, що провалу асерції не призначений певний рівень серйозності.

Всі ці завдання повинні виводити певні повідомлення, що означають рівень серйозності провалу асерції і певну інформацію, яка повинна включати таке: 1) ім'я файлу й номер лінії оператора асерції; 2) ієрархічне ім'я оператора асерції, якщо він має ім'я, або межі видимості асерції, якщо немає мітки.

Для інструментів моделювання ці завдання повинні повідомляти добовий час моделювання завдання. Кожне системне завдання може також включати додаткову, описувану користувачем інформацію, використовуючи той же самий формат подання даних, що й оператор Verilog \$display.

Якщо гілка else містить більш ніж одну із цих системних завдань, то кожна з них повинна бути описана зазначеним способом. У наступному прикладі завдання рівня серйозності виконується не в момент провалу асерції, але, тим не менш, виведений реальний час виявлення помилки:

```
time t;
always @(posedge clk)
    if (state == REQ)
        assert (req1 || req2)
    else begin
        t = $time;
        #5 $error("assert failed at time %0t",t);
    end
```

Якщо асерція провалиться в момент часу 10, повідомлення про помилку буде виведено в момент часу 15, але при цьому містити воно буде "assert failed at time 10".

Виведення інформаційних повідомлень і попереджень може контролюватися опціями, формованими командами. Оскільки fail-оператори, як і pass-оператори, є легальними процедурними операторами SystemVerilog, то вони можуть бути також використані для формування умов виконання асерцій в іншій частині testbench:

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0) else flag = 1;
```

22.4. Паралельні асерції

Інший тип - паралельні асерції виконують перевірку сигналу протягом усього процесу моделювання. При їхньому описі використовується синхронізація. Паралельні асерції відрізняються від прямих двома важливими аспектами. По-перше, на додаток до прямого запису в проєкті, подібно до операторів у блоках always або initial, вони можуть бути реалізовані декларативно, на рівні безперервних операторів присвоєння і реалізації модулів (подібно до безперервного присвоєння) за межами процедурних блоків. Друга відмінність полягає в тому, що паралельні асерції дозволяють описувати поведінку системи в часі замість комбінаційних умов у прямих асерціях.

Синтаксис:

```
assert property ( sequential_expr_or_property )
                [[ pass_stmt ] else fail_stmt ]
```

Ключове слово property відрізняє паралельну асерцію від прямої. Послідовний вираз обчислюється з використанням значень сигналів і дозволяє операторам pass/fail зв'язуватися з testbench. Оскільки асерції є внутрішньою частиною мови, ці оператори можуть застосовувати SystemVerilog у повному обсязі для перемикання подій, запису інформації про покриття або інших впливів на порядок коду верифікації, включаючи виклик функцій мови C. Окрема мова асерцій ефективна тільки в режимі «read-only». Вона дозволяє спостерігати за поведінкою проєкту, але не допускає елементів впливу на проєкт і testbench. В наступному прикладі виконується перевірка асерції - сигнал арбітра не приймає значення X або Z, крім ситуації скидання пристрою, коли reset=1 (лістинг 22.2).

Лістинг 22.2. Паралельна асерція для перевірки значення X/Z.

```
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    logic reset;
    property request_2state;
        @(posedge clk) disable iff (reset) $isunknown(request) == 0;
    endproperty
    assert_request_2state: assert property request_2state
endinterface
```

Паралельні асерції описують поведінку, яка охоплює великий проміжок часу. На відміну від прямих асерцій, обробка моделі ґрунтується на синхронізації. Паралельна асерція обчислюється тільки з появою події синхронізації. При цьому використовується вибірка значень змінних. Незалежно від внут-

рішнього механізму обробки подій симулятором, у результаті обчислень може бути отриманий передбачуваний результат. Ця здійснена модель відповідає також синтезованій моделі апаратної реалізації RTL-опису. Значення змінних, використовуваних в асерціях, збираються в підготовчий період (Preponed) часового слота, асерції обчислюються в періоди спостереження (Observe). Якщо змінна, використовувана в асерції, є вхідною для блоку синхронізації, то її значення зчитується # 1step кроком керування. Будь-які інші типи вибірок для змінних блоків синхронізації призведуть до помилки. Асерції, що використовують змінні блоків синхронізації, не створюють свій спосіб вибірки значень, а застосовують процедуру, зазначену в блоках синхронізації.

Часові моделі, застосовувані в специфікаціях паралельних асерцій, засновані на синхроподіях і використовують узагальнене поняття синхротакту. Визначення синхронізації, як правило, створюється користувачем і може змінюватися від одного виразу до іншого. Подія синхронізації - це окремий момент часу, який сам не має тривалості. Подія синхронізації може мати місце тільки один раз протягом будь-якого часу моделювання. Значення вибірки сигналів для цього періоду моделювання використовуються при обчисленні паралельних асерцій. Рис. 22.4 представляє стан змінної при зміні синхронізації.

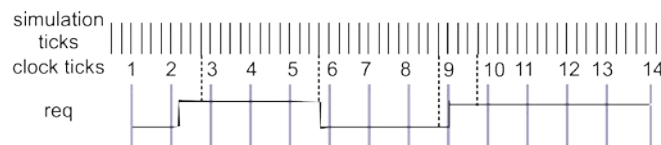


Рис. 22.4. Періоди отримання значень

Значення сигналу *req* дорівнюють нулю в моменти подій синхронізації 1 і 2. При події 3 значення вибірки перемикається в одиницю й залишається в такому стані до синхроподії 6. Значення змінної *req* у момент події 6 стає рівним нулю й залишається таким для подій 7, 8 і 9. Синхроподія 9 відбувається разом з перемиканням сигналу *req*, однак значення вибірки для цього моменту залишається рівним нулю, оскільки, відповідно до семантики мови, значення зчитується за один крок (1step) до появи події.

Вираз, використовуваний в асерціях, завжди пов'язаний з визначенням синхронізації. Значення вибірки, використовувані для обчислення змінюваних або булевих виразів, необхідні для виявлення послідовності. Для паралельних асерцій потрібно виконувати такі умови: 1. Гарантувати вільне від збоїв визначення поведінки синхросигналу. У протилежному випадку можуть бути отримані помилкові значення вибірок. 2. Якщо змінна, котра з'являється у виразі синхронізації, також з'являється в асерції, то її значення може бути різним. Поточне її значення використовується у виразі синхронізації, у той час як значення вибірки змінної використовується всередині асерції. Вираз синхронізації, що контролює обчислення послідовності, може бути більш складним, ніж просте ім'я одного сигналу. Для опису синхронізації можуть

бути використані вирази виду: (clk && gating_signal) і (clk iff gating_signal). Можливі інші, більш складні конструкції. Однак з метою забезпечення правильної поведінки системи й відповідності семантиці синхронного моделювання, сигнали у виразах синхронізації повинні бути вільні від ризиків збоїв і мати тільки одне перемикання. Приклад паралельної асерції наведений нижче:

```
base_rule1: assert property (cont_prop(rst,in1,in2)) $display("%m, passing");
else $display("%m, failed");
```

Властивість може виконувати перевірку однієї або декількох послідовностей, що стартують у різний час. Обчислення кожного з них починається з пошуку збігу початку послідовності на заданій синхроподії. Щоб визначити наявність збігу послідовності, обчислюються відповідні булеві вирази, починаючи з певної синхроподії, і тривають при кожній наступній синхроподії, поки вирази мають значення «істина». Якщо зафіксовано значення «неправда», то збігу немає.

22.5. Властивості (Properties)

У багатьох випадках послідовності дозволяють описувати поведінку системи. Задавати перевірку поведінки можна за допомогою властивостей. Вони дозволяють інвертувати зміст послідовності, відключити її обчислення або застосувати послідовність до інших подій.

Для визначення властивостей використовується такий синтаксис:

```
// Декларація властивостей
property name [ ( formal_item {, formal_item } ) ] ;
    { assertion_variable_declaration }
    property_spec ;
endproperty [:name]
```

Опис властивостей складається з послідовностей або їхньої імплікації, а також інших властивостей:

```
property_spec ::=
    [clocking_event] [disable iff( expression )] [ not ] property_expr
property_expr ::= sequence_expr | implication_expr
```

Визначення властивостей виконується в межах property-endproperty і підтримує формальні аргументи поводження послідовності.

Наприклад:

```
property p1;
    @(posedge clk) disable iff (test) not abort_seq;
endproperty
```

Оператор not інвертує зміст послідовності abort_seq, таким чином, вираз буде мати значення «неправда» з появою послідовності abort_seq. Вираз disable iff відключає обчислення послідовності тоді і тільки тоді, якщо сигнал test=1. Можливо повторне використання ключових слів disable і iff, які застосовуються в інших конструкціях SystemVerilog. Це властивість повинна інтерпретуватися як «поки сигнал test=0, варто перевіряти відсутність появи послі-

довності `abort_seq`». Інша корисна поведінкова концепція - «це виключається» (that of implication). Нехай необхідно перевірити появу послідовності після фіксації `init_seq`. Така властивість може бути описана так:

```
@(posedge clk) init_seq | => abort_seq;
```

де `|=>` оператор імплікації, що не перекривається. Це вираження властивостей стверджує, що кожна успішно завершена послідовність `init_seq` першого операнда має на увазі, що `abort_seq` із другого операнда виконується на наступному синхротакті. Для кожного такту, у якому успішне завершення послідовності першого операнда не є присутнім, послідовність другого операнду не буде обчислена й властивість розглядається як «істина». Властивість визначає поведінку проекту. Вона може бути використана для верифікації, як специфікація припущення, перевірки або покриття. Для того щоб використовувати поведінку для верифікації, застосовуються оператори `assert`, `assume` або `cover`. Декларація властивості не приводить до результату. Властивість `property`, як і послідовність, декларується з необов'язковими формальними аргументами. Коли властивість обчислюється, в неї передаються фактичні аргументи. Механізм їхньої передачі у властивості аналогічний передачі аргументів у послідовність. Тут вона отримує розширення з реальними аргументами, що замінюють формальні аргументи.

Результат обчислення властивостей: «істина» або «неправда». Існує сім видів властивостей: послідовність (sequence), заперечення (negation), диз'юнкція (disjunction), кон'юнкція (conjunction), `if...else`, імплікація (implication) і створення копії (instantiation).

1) Властивість типу послідовність є послідовністю, що обчислюється при значенні «істина», тоді й тільки тоді, коли існує непустий збіг послідовності. Якщо вона допускає порожній збіг, то такий результат неприпустимий для властивостей, оскільки збіг існує, тоді й тільки тоді існує перший збіг. Обчислення такої властивості є повне трансформування `sequence_expr` в `first_match (sequence_expr)`. Як тільки перший збіг `sequence_expr` виявляється, обчислення властивості розглядається як «істина» і немає необхідності в пошуку інших збігів.

2) Властивість типу заперечення має такий вигляд:

```
not property_expr
```

При кожній спробі обчислення властивостей обчислюється `property_expr`. Ключове слово `not` означає, що при його обчисленні вертається протилежне значення для `property_expr`. Таким чином, якщо `property_expr` приймає значення «істина», то `not property_expr` відповідає значенню «неправда», і якщо `property_expr` оцінюється як «неправда», то `not property_expr` буде «істина».

3) Диз'юнкція властивостей має такий вигляд:

```
property_expr1 or property_expr2
```

Властивість відповідає значенню «істина», якщо й тільки якщо одне із властивостей `property_expr1` і `property_expr2` «істина».

4) Кон'юнкція властивостей має таку форму:

`property_expr1 and property_expr2`

Властивість оцінюється як «істина», якщо й тільки якщо `property_expr1` і `property_expr2` мають значення «істина».

5) Властивість «if...else» має таку форму:

`if (expression_or_dist) property_expr1`

або

`if (expression_or_dist) property_expr1 else property_expr2`

Властивість у першому випадку оцінюється як «істина», якщо й тільки якщо `expression_or_dist` приймає значення «неправда» або `property_expr1` - «істина». У другому випадку властивість повертає значення «істина», коли `expression_or_dist` і `property_expr1` повертають значення «істина», інакше `expression_or_dist` обчислюється в «неправду», а `property_expr2` - «істина».

6) Властивість типу імпліканта має таку форму:

`sequence_expr |-> property_expr`

або

`sequence_expr |=> property_expr`

7) Копія іменної властивості може бути використана у виразі `property_expr` or `property_spec`. У загальному випадку, використання копії є законним, якщо тіло іменної властивості `property_spec` може замінити його копію в `property_expr` or `property_spec`, підставляючи реальні параметри замість формальних аргументів і ігноруючи локальні змінні. Якщо копія іменної властивості використовується як операнд `property_expr` у будь-якому формулюючому властивість операторі, то ця властивість не повинна включати вираз `disable iff`. Аналогічним чином, події синхронізації в іменних властивостях повинні відповідати правилам мультисинхронної підтримки, у випадку якщо властивість входить у вираз `property_expr` (`property_spec`), що також управляється іншою синхроподією. Табл. 22.2 представляє списки операторів для послідовностей і властивостей, починаючи з найвищого пріоритету, і показує сполучуваність для неунарних операторів.

Вираз `disable iff` може бути підключений до (`a property_expr`), щоб сформулювати `property_spec`:

`disable iff (expression_or_dist) property_expr`

Вираз `disable iff` називається умовою скидання (`reset expression`) і дозволяє описувати пріоритетне скидання. Для обчислення `property_spec` оцінюються складові `property_expr`. Якщо до завершення такого обчислення вираз скидання приймає значення «істина», то стан властивості `property_spec` - «істина». У протилежному випадку значення `property_spec` залежить від результату обчислення `property_expr`. Скидання виразу перевіряється незалежно для різних спроб обчислення `property_spec`. Значення сигналів у виразі скидання є значеннями змінних у поточному циклі моделювання, а не їхніми вибірками.

Вирази можуть містити посилання на кінцеву крапку послідовності за допомогою методу її перемикавання. Методи порівняння й виявлення кінця послідовності й локальні змінні не можуть бути використані у виразі скидання. Якщо значення вибірки функції використовується у виразі скидання, то синхронізація вибірки повинна бути явно описана в списку актуальних аргументів.

Таблиця 22.2. Оператори послідовностей і властивостей

Sequence operators	Property operators	Associativity
[*], [=], [->]		--
##		Left
throughout		Right
within		Left
intersect		Left
	not	--
and	and	Left
or	or	Left
	if...else	Right
	->, =>	Right

Властивості ніколи не обчислюються для перевірки виразів. Їх використовують в операторі верифікації, що встановлює одну з таких функцій: `assert` - для перевірки опису властивості, щоб забезпечити її місце в проекті; `assume` - для опису властивостей як припущення для середовища; `cover` - для моніторингу обчислення властивостей покриття.

22.6. Системні функції, використовувані в асерціях

Асерції використовуються для обчислення специфічних особливостей реалізації проекту, наприклад, певний сигнал містить тільки одну одиницю. Наступні системні функції введені для опису функціональності асерції: `$onehot` (`<expression>`) повертає значення «істина», якщо тільки один біт виразу дорівнює 1; `$onehot0` (`<expression>`) повертає значення «істина», якщо як мінімум один біт виразу дорівнює 1; `$isunknown` (`<expression>`) повертає значення «істина», якщо будь-який біт виразу дорівнює X або Z. Це еквівалентно `^ <expression> == 'bx`.

Всі перелічені системні функції повертають значення типу `bit`. Повернення значення `1'b1` відповідає «істині», а значення `1'b0` – «неправда». Інша корисна функція, надавана для логічного виразу `$countones`, підраховує кількість одиниць у бітовому векторі:

`$countones(expression)`

Значення X и Z не зараховуються в число одиниць.

22.7. Паралельні й прямі асерції

На практиці більшість миттєвих (immediate) асерцій виражається в деякій формі імплікації, що вимагає їхнього запису для формування попереднього виразу з метою його перемикавання. Як вказувалося раніше, одне із ключових переваг асерцій - простота їхнього вбудовування в проект для інженерів-

проектувальників. Проте декларація асерцій часто вимагає додаткових зусиль від проектувальника для ефективного їхнього використання. Далі розглядається модель керуючого автомата. Асерції в даному типі проектів часто формують висловлення «якщо цей стан, то повинен відбутися» і «якщо в стані АСК деяка змінна foo дорівнює 1, тоді регістр повинен зберігати значення 0 протягом 5 синхротактів». Паралельна асерція для перевірки даної умови може бути записана в такий спосіб:

```
P1: assert property(
  @(posedge clk) (st == АСК) && (foo == 1) |-> !req[*5]);
```

При цьому RTL-код автомата повинен виглядати так:

```
always @(posedge clk)
case (st)
  АСК:
    if (foo == 1)
      begin          // у цьому стані req=0 протягом 5 синхротактів
        ...
      end
    ...
endcase
```

Отже, два фрагменти інформації, які дублюються в проекті, також асерції: синхронізації й перемикання станів. Замість того, щоб жадати від проектувальника дублювання цієї інформації, асерції, що є частиною мови проектування, можуть бути процедурно вбудовані в RTL-код і автоматично виводити інформацію при їхньому використанні, наприклад:

```
always @(posedge clk)
case (st)
  АСК:
    if (foo == 1)
    begin
      P2: assert property (!req[*5]);
      ...
    end
```

Пряма асерція (P2) є семантично еквівалентною декларації паралельної асерції (P1), наведеної вище, але набагато простіша у використанні й обслуговуванні. Процедурні вбудовані асерції використовують отримані значення сигналів для оцінки умов перемикання, як роблять це декларативні асерції. Правила логічного виведення для попередників містять у собі case і if-else оператори, що дозволяє вводити в процедурні асерції перемикання довільної складності. У результаті перевагою процедурних асерцій є просте обслуговування. Якщо змінюється код кінцевого автомата, умови перемикання для асерції автоматично оновлюються, а користувачеві варто модифікувати код вручну, щоб оновити перемикання, що відповідають декларативним асерціям.

22.8. Взаємодія асерцій і testbench

Як вказувалося раніше, перевагою асерцій, які взаємодіють із проектом і мовою верифікації є можливість для асерцій передавати інформацію в testbench. В окремій мові асерцій не існувало такого механізму комунікацій. В SystemVerilog у гілках pass і fail асерції можуть виконувати будь-які процедурні оператори, дозволяючи користувачеві передавати інформацію в testbench про виконання послідовності, оновляти лічильники покриття, робити установку значень сигналів, виклик методів об'єктів класу, або навіть виклик коду Cі.

Нехай шина включає властивість «новий цикл шини не може початися раніше, ніж через 2 синхротакта після появи сигналу завершення циклу abort_cycle». Ця властивість може кодуватися як

```
property wait_after_abort;
    @(posedge clk) abort_cycle |=> !cycle_start[*2];
endproperty
```

P3: assert property (wait_after_abort);

Коли з'являється послідовність abort_cycle, властивість диктує, що сигнал cycle_start буде дорівнювати нулю протягом двох синхротактів. Властивість P3 кодує цю поведінку й може бути використана для моніторингу під час моделювання й перевірки поведінки, а також у формальних інструментах, щоб довести неможливість деякого стану.

Проте, якщо ця властивість визначає поведінку інтерфейсу верхнього рівня, testbench буде генерувати транзакцію шини для підтвердження стану інтерфейсу. Інженери верифікації, створюючи testbench, можуть повторно використовувати інформацію асерцій, як константу, що допомагає гарантувати, що символи, які генеруються, ніколи не порушують поведінку. Такий testbench може виглядати подібним чином:

```
program manual_stimulus_generator;
repeat(1000)
begin
    generate_transaction(addr,data);
    while(wait_cnt > 0)
        @(posedge clk) wait_cnt ---i;
    end
endprogram
```

Лічильник wait_cnt використовується для опису затримки між транзакціями шини, що генеруються модулем testbench. Послідовність abort_cycle може бути застосована для установки лічильника, використовуючи директиву покриття:

```
cover property( abort_cycle ) wait_cnt = 2;
```

Використання cover дозволяє виявляти послідовність abort_cycle без формування помилки, якщо вона не присутня. Якщо послідовність виявлена, то лічильник wait_cnt встановлюється в 2, змушує testbench чекати 2 такти до

генерування наступної транзакції. Семантика формування порядку обробки подій SystemVerilog гарантує, що тут не буде гонок між виконанням `pass statement` і `testbench`, що не можна було б гарантувати, якби асерції були реалізовані за допомогою іншої мови.

SystemVerilog включає можливість опису явного синхронного інтерфейсу між `testbench` і проектом. Конструктор області синхронізації визначає, коли щодо синхронізації `testbench` буде захоплювати й передавати тестованому пристрою (DUT) сигнали. За замовчуванням, захват значень сигналів в `testbench` аналогічний захвату з використанням асерцій, що гарантує - `testbench` має доступ до тих же значень, що й асерції. Це не тільки дозволяє уникнути гонок між асерціями `testbench`, асерціями й тестованим пристроєм, але також дає `testbench` подання про «post-synthesis» поведінку пристрою щодо гонок, відповідно до правил синтезу, під час RTL-верифікації, виконаної до синтезу.

22.9. Підтримка багатодоменої синхронізації

Послідовності й властивості можуть використовувати багатодоменну синхронізацію.

22.9.1. Послідовності із множинною синхронізацією (Multiply-clocked sequences).

Послідовності із множинною синхронізацією будуються шляхом конкатенації декількох наборів за допомогою оператора конкатенації з одиничною затримкою `##1`. Цей оператор не перекриває й не синхронізує з'єднання синхросигналів двох послідовностей. Одинична затримка, позначувана `##1`, розуміється як затримка між закінченням першої послідовності, що управляється першим синхросигналом, і найближчою активною подією або фронтом другого синхросигналу, з якого починається виконання другої синхропослідовності. Наприклад,

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

Пошук послідовності починається з порівняння `sig0` після появи переднього фронту `clk0`. Потім `##1` ініціює затримку до найближчого переднього фронту `sig1`. Пошук послідовностей завершується із закінченням набору `sig1`. Якщо `clk0` і `clk1` не ідентичні, то синхроподія перемикається після `##1`. Інакше, якщо `clk0` і `clk1` рівні, синхронізуюча подія не змінюється після `##1` і представлена послідовність еквівалентна набору з одним синхросигналом:

```
@(posedge clk0) sig0 ##1 sig1
```

Коли виконується конкатенація по-різному синхронізованих послідовностей, максимальний набір з одиничною синхронізацією необхідний, щоб допустити непустий збіг. Якщо послідовності `s1` і `s2` є виразами наборів без подій синхронізації, то тоді послідовність із множинною синхронізацією

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

припустима, якщо `s1` або `s2` збігаються з порожнім словом. Подія переднього фронту синхросигналу `clk1` застосовується на всьому протязі пошуку послідовності `s1`, у той час як передній фронт `clk2` є подією синхронізації під час

пошуку послідовності s2. Якщо збіг s1 - не порожній, то ця подія є кінцевою точкою збігу переднього фронту clk1. Синхронізація між кінцевою точкою послідовності й появою переднього фронту clk2 після цього досягається за допомогою оператора ##1. Передній фронт clk2 є початковою точкою пошуку послідовності s2.

22.9.2. Властивості з мультисинхронізацією (Multiply-clocked properties).

Для властивостей з мультисинхронізацією результат обчислюється так само, як і для випадку з єдиним синхросигналом. Властивості із множинною синхронізацією можуть бути описані кількома способами. Послідовність із мультисинхронізацією є сама по собі властивістю з мультисинхронізацією. Наприклад,

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

є властивість із множинною синхронізацією. Якщо послідовність із множинною синхронізацією обчислюється як властивість, що починається в деякій точці, обчислення повертає значення істини, якщо й тільки якщо існує збіг у послідовності із множинною синхронізацією в цій точці.

Булеві оператори властивостей (not, and, or) можуть бути використані вільно для комбінування одно- і мультисинхронної властивості. Значення операторів булевих властивостей дорівнюють одиниці, як і для випадку властивостей, що синхронізуються окремо. Наприклад,

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

мультисинхронна властивість, але не мультисинхронна послідовність. Ця властивість повертає true у точці, якщо й тільки якщо дві послідовності

```
@(posedge clk0) sig0
```

та

```
@(posedge clk1) sig1
```

збігаються на початку цієї точки.

Два оператори імплікації |=>, що не перекриваються, можна використовувати для створення мультисинхронної властивості з попередньої послідовності й наступних властивостей, що означає їх мультисинхронізованість. Значення мультисинхронізованої і односинхронізованої імплікацій, що не перекриваються, подібні. Наприклад, якщо s0, s1 - послідовності з несинхронізуємою подією, тоді

```
@(posedge clk0) s0 |=> @(posedge clk1) s1 |=>
```

синхронізуються переднім фронтом clk0 і clk1. Починаючи із точки, у якій імплікація була обчислена, кожний збіг s0 синхронізується із сигналом clk0. При цьому зміни в точці відбуваються по передньому фронту clk1. Крім того, для формування значення точки повинен виконуватися збіг s1 із синхросигналом clk1 у проміжку часу між передніми фронтами clk0 і clk1. Починаючи із точки, у якій обчислюється імплікація, для кожного збігу s0, синхронізованого сигналом clk0, час рухається від кінцевої точки збігу до найближчої, котра буде активізована переднім фронтом clk1. При цьому тут повинен існувати збіг сигналу s1 із синхросигналом clk1. Синхронізація між окремими синхросигналами завжди вимагає строгого передування в часі.

Два оператори, що створюють властивості, вимагають спеціальну властивість із множинною синхронізацією для перекриття імплікацією $|-| \rightarrow$ і якщо if/ if...else. Пізніше імплікація $|-| \rightarrow$ повинна забезпечувати синхронізацію перекриття кінця попередника з початком спадкоємця. Наприклад, якщо clk0 і clk1 не ідентичні й s0, s1, s2 - це послідовності з несинхронізованими подіями, тоді

```
@(posedge clk0) s0 |-|> @(posedge clk1) s1 ##1 @(posedge clk2) s2
```

не є правильним виразом, але

```
@(posedge clk0) s0 |-|> @(posedge clk0) s1 ##1 @(posedge clk2) s2
```

є правильним.

22.9.3. Синхропотік (Clock flow).

Синхропотік дозволяє масштабувати події синхронізації для того, щоб поширювати його звичайним способом через різні частини мультисинхронних послідовностей і скорочувати число місць, у яких однакові події синхронізації повинні бути описані. Потік синхронізації забезпечує в мультисинхронній послідовності або властивості масштабованість події синхронізації, що йде зліва направо через лінійні оператори (повторення, конкатенація, заперечення, імплікація). Крім того, виконується розподіл операндів через оператори розгалуження (кон'юнкція, диз'юнкція, перетинання, if ... else), поки процес не буде замінений новою подією синхронізації.

Наприклад,

```
@(c) x |>= @(c) y ##1 @(d) z
```

може бути записано в більш простій формі

```
@(c) x |>= y ##1 @(d) z ,
```

тому що синхросигнал проходить через $| \Rightarrow$.

Синхропотік позбавляє розроблювача від необхідності записувати синхроподії в позиціях, де не допускається зміна синхронізації. Наприклад:

```
@(c) x |-|> @(c) y ##1 @(d) z
```

може бути записане як

```
@(c) x |-|> y ##1 @(d) z,
```

що означає - підсилити обмеження на те, що синхронізація не зміниться при переході через $| \rightarrow$. Аналогічно конструкція

```
@(c) if (b) @(c) w ##1 @(d) x else @(c) y ##1 @(d) z
```

може бути записана, як

```
@(c) if (b) w ##1 @(d) x else y ##1 @(d) z,
```

що означає - підсилити обмеження на те, що синхронізація не зміниться з булевою умовою b c, а також з умовою розгалуження if і else властивості.

Синхропотік також робить з'єднані відносини між конкатенацією й імплікацією ясними для мультисинхронних властивостей:

```
@(c) x ##1 y |>= @(d) z
```

еквівалентно виразу

```
@(c) x |>= y |>= @(d) z
```

та

@(c) x ##0 y | => @(d) z

еквівалентно виразу

@(c) x | -> y | => @(d) z

Межі подій синхронізації відповідають підвиразам, обмеженим дужками, і якщо підвираз є послідовністю, вона також передається зліва направо, через дужки підвиразу. Проте межі видимості подій синхронізації обмежуються дужками.

22.9.4. Приклади.

Приклади мультисинхронного опису (multiple-clock specifications):

```
sequence s1;
```

```
@(posedge clk1) a ##1 b; // послідовність із єдиним синхросигналом
```

```
endsequence
```

```
sequence s2;
```

```
@(posedge clk2) c ##1 d; // послідовність із єдиним синхросигналом
```

```
endsequence
```

1) Мультисинхронна послідовність:

```
sequence mult_s;
```

```
@(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
```

```
endsequence
```

2) Властивість із мультисинхронною послідовністю:

```
property mult_p1;
```

```
@(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
```

```
endproperty
```

3) Властивість із іменною мультисинхронною послідовністю :

```
property mult_p2;
```

```
    mult_s;
```

```
endproperty
```

4) Властивості з імплікацією мультисинхронізації:

```
property mult_p3;
```

```
@(posedge clk) a ##1 @(posedge clk1) s1 | => @(posedge clk2) s2;
```

```
endproperty
```

5) Властивості з іменними послідовностями, що мають різну синхронізацію.

У цьому випадку, якщо s1 містить синхросигнал, то він повинен бути ідентичним (posedge clk1). Подібним чином, якщо s2 містить синхросигнал, він повинен бути ідентичним (posedge clk2):

```
property mult_p5
```

```
    @(posedge clk1) s1 | => @(posedge clk2) s2;
```

```
endproperty
```

6) Властивості з імплікацією, де попередник і послідовник є іменними мультисинхронними послідовностями:

```
property mult_p6;
```

```
    mult_s | => mult_s;
```

```
endproperty
```

7) Властивість, що використовує синхропотік і імплікацію, що перекривається:

```
property mult_p7;
    @(posedge clk) a ##1 b |-> c ##1 @(posedge clk1) d;
endproperty
```

Тут a, b і c синхронізуються переднім фронтом clk.

8) Властивість, що використовує синхропотік і if...else:

```
property mult_p8;
    @(posedge clk) a ##1 b |->
if (c)
    (1 |> @(posedge clk1) d)
else
    e ##1 @(posedge clk2) f ;
endproperty
```

Тут a, b, c і e синхронізуються переднім фронтом clk.

22.10. Контрольні завдання

1. Створити послідовність (набір), що перевіряє одночасне виконання наборів seq1 і seq2, керованих переднім фронтом синхросигналу clk.
2. Створити послідовність, що перевіряє виконання хоча б одного із двох наборів seq1 і seq2, керованих переднім фронтом clk.
3. Написати паралельну асерцію, що виконує перевірку значень A=01, set=1, по передньому фронту clk, що виводить повідомлення у випадку вдалого й невдалого виконання асерції із вказівкою мітки асерції.
4. Написати пряму іменну асерцію, що виконує перевірку значень A=01, set=1, що виводить повідомлення у випадку вдалого й невдалого виконання асерції із вказівкою мітки асерції.
5. Написати паралельну асерцію, що виконує перевірку значень, що сигнал grant представлений унарним кодом, set=1, якщо reset не дорівнює 1. Здійснити виведення повідомлення про помилку у випадку невдалого виконання асерції.
6. Написати паралельну асерцію, що виконує перевірку проходження сигналу a=1, через 5 тактів після d=1 або b=1. Управляється заднім фронтом clk. У випадку неуспішного виконання асерції виводити попередження.
7. Написати паралельну асерцію, що виконує перевірку проходження сигналу a=1, через 2-5 тактів після z=1 і b=1. Управляється заднім фронтом clk. У випадку неуспішного виконання асерції виводити попередження.
8. Написати паралельну асерцію, що виконує перевірку проходження сигналу a=1 або b=1, через 3 такти після z=1. Управляється переднім фронтом clk. У випадку неуспішного виконання асерції виводити попередження.
9. Написати паралельну асерцію, що виконує перевірку проходження сигналу a=1, через 5 тактів після d=1 і через 8 тактів після b=1. Управляється переднім фронтом clk. У випадку неуспішного виконання асерції виводити попередження.

10. Написати паралельну асерцію, що виконує перевірку проходження сигналу $a=1$ або $d=1$ і через такт після $b=1$. Управляється переднім фронтом clk . У випадку неуспішного виконання асерції виводити повідомлення про фатальну помилку.
11. Створити послідовність без перекриття, що з'єднає послідовності a і b , позначити набори за допомогою формальних параметрів, які дозволять їх перевизначити при використанні послідовності.
12. Створити набір, що перевіряє послідовність через 2 такти проходження один за одним через події a , b і c , керовану переднім фронтом синхросигналу clk .
13. Написати паралельну асерцію, що виконує перевірку появи $set=1$ і через такт - $A=01$, керовану переднім фронтом clk . Активізувати виведення повідомлення із вказівкою мітки асерції у випадку вдалого й невдалого її виконання.
14. Написати пряму асерцію, що виконує перевірку значень $grant=01$ і $request=1$. Активізувати виведення повідомлення у випадку вдалого й невдалого виконання асерції із вказівкою її мітки.
15. Написати паралельну асерцію, що виконує перевірку значень, що сигнал $grant$ містить як мінімум одну «1», якщо сигнал $reset$ не дорівнює «1». Активізувати виведення повідомлення про помилку у випадку невдалого виконання асерції.
16. Написати паралельну асерцію, що виконує перевірку проходження сигналу $a=1$, через 2-5 тактів після $z=1$ і $b=1$. Керування здійснюється заднім фронтом clk . У випадку неуспішного виконання асерції виводити попередження.
17. Написати паралельну асерцію, що виконує перевірку проходження сигналу $a=1$ або $b=1$, через 3 такти після $z=1$. Керування здійснюється переднім фронтом clk . У випадку неуспішного виконання асерції виводити попередження.

22.11. Література

1. IEEE Std. 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog[®] Hardware Description Language 1995. ISBN 0-7381-3065-6. IEEE Product No. WE94418-TBR.
2. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).– IEEE Standard Verilog Hardware Description Language.– IEEE Computer Society Sponsored by the Design Automation Standards Committee.– Published by The Institute of Electrical and Electronics Engineers, Inc.– 3 Park Avenue, New York, NY 10016-5997, USA.– 2001.– 791 p.
3. IEEE Std 1800-2005.– IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language.– Sponsor: Design Automation Standards Committee of the IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group.– Published by The Institute of Electrical and

Electronics Engineers, Inc.– 3 Park Avenue. New York, NY 10016-5997, USA.– 2005.– 664 p.

4. SystemVerilog 3.1a. Language Reference Manual. – Accellera’s Extensions to Verilog®.– Accellera Organization, Inc. 1370 Trancas Street #163, Napa, CA 94558, USA.– 2004.

5. Spear C. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, 2nd ed.– Springer-Verlag, New York, Inc. Secaucus, NJ, USA.– 2006.– 429 p.

6. Mintz M., Ekendahl R. Hardware Verification with System Verilog. An Object-Oriented Framework.– New York : Springer.– 2007.– 314 p.

7. Bergeron J. Writing Testbenches using SystemVerilog.– Springer US.–2006.– 414 p.

8. Bergeron J., Cerny E., Hunter A., Nightingale A. Verification Methodology. Manual for SystemVerilog.– Springer US.–2005.– 510 p.

9. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas. 101 Common Coding Errors and How to Avoid Them.– New York : Springer.– 2007.– 214 p.

10. Vijayaraghavan Srikanth, Ramanathan Meyyappan. A Practical Guide for SystemVerilog Assertions.– Springer US.– 2005.– 334 p.

11. Haque F., Michelson J., Khan K. The Art of Verification with System Verilog Assertions.– Verification Central. First ed.– 2006.

23. ПСЕВДОВИПАДКОВЕ ТЕСТУВАННЯ

Представлено моделі й процедури створення різних типів псевдовипадкових тестів (randomization) для верифікації цифрових проектів, описаних на системних мовах опису апаратури.

Мета - створення технологічних процедур синтезу псевдовипадкових тестів, орієнтованих на пошук дефектів, детермінованих у часі, а також переможованих несправностей, що змінюють функціонування пристрою й приводять проект у непрацездатний стан.

Завдання - вивчення найбільш технологічних процедур синтезу псевдовипадкових тестів, орієнтованих на перевірку апаратних рішень цифрових систем на кристалах.

Джерела: [1, 2] – методи синтезу детермінованих і псевдовипадкових тестів для різних типів цифрових структур; [3-13] – технології програмування процедур створення псевдовипадкових тестів на основі HDL мов опису апаратури.

У процесі розробки програмного коду проектувальники допускають велику кількість некоректностей. Їх можна розділити на такі основні категорії: 1) неминучі синтаксичні помилки й помилки у вихідному програмному коді, що виявляються практично миттєво при запуску компілятора; 2) дрібні технічні помилки, які виникають при початковому складанні змінених частин програми, що виявляються при перших запусках на виконання; 3) логічні помилки у функціональності програми, що виявляються не по станах виходів, але частина з них може бути виявлена в процесі тестування; 4) помилки й недоліки алгоритмів і моделей, які проявляються під час виконання програмного коду у вигляді підвищеної витрати оперативної пам'яті, неприйнятних тимчасових витрат на виконання програми.

Перші дві категорії синтаксичних неточностей виявляються розроблювачами відносно швидко, однак семантичні помилки останніх двох категорій специфічні для розв'язуваного завдання й вимагають використання продуманої стратегії тестування. Основна складність при тестуванні цифрових систем полягає в неочевидності симптомів багатьох некоректностей. Якщо помилка проявляється у вигляді неправильних результатів моделювання, візуально виявити її й усунути набагато складніше. Вирішити це можна за допомогою спеціальних засобів додаткового аналізу, заснованого на створенні тестів, що покривають функціональні режими й/або несправності заданого класу.

У мові System Verilog тест створюється за допомогою генератора псевдовипадкових значень, що перенастроюється (Constrained-Random Tests - CRT). Проблемно-орієнтований або прямий тест (directed test) призначений для пошуку наперед заданих дефектів, у той час як CRT-тест виявляє випадкові або незаплановані помилки: 1) конфігурації пристроїв (Device configuration); 2) конфігурації середовища (Environment configuration); 3) вихідних вхідних даних (Primary input data); 4) інкапсульованих вхідних даних (Encapsulated

input data); 5) виключень протоколів (Protocol exceptions); 6) затримок (Delays); 7) статусу транзакцій (Transaction status); 8) несправності й проти-річчя (Errors and violation).

23.1. Найпростіший клас коду із псевдовипадковими змінними

Найпростіший клас, що формує псевдовипадкові послідовності, які перевіряють функціональність тесту, представлений лістингом 23.1.

Лістинг 23.1. Найпростіший клас рандомізації.

```
class Packet;
    // Випадкові змінні
    rand bit [31:0] src, dst, data[8];
    randc bit [7:0] kind;
    // Обмеження значень src
    constraint c {src > 10; src < 15;}
endclass
Packet p;
    initial begin
        p = new; // Створення пакета
        assert (p.randomize());
        transmit(p);
    end
```

Клас має чотири змінні із псевдовипадковими значеннями. Перші три використовують модифікатор rand, таким чином, що щоразу, при рандомізації класу, змінні можуть отримувати нові значення або повторювати які-небудь із попередніх. Змінна kind описується як randc, що означає псевдовипадковий цикл, де формовані значення не будуть повторюватися, поки не будуть перебрані всі можливі.

Обмежуючий вираз constraint групується за допомогою фігурних дужок {}. Вони використовуються для ідентифікації декларативного коду, але не процедурного, де необхідно застосовувати begin...end. Функція randomize повертає 0, якщо проблема використовує обмеження. Для перевірки результатів формування випадкових значень застосовуються прямі асерції. Лістинг 23.2 містить клас коду із псевдовипадковою генерацією значень із обмеженнями.

Лістинг 23.2. Псевдовипадковий клас із обмеженнями.

```
class Stim;
    const bit [31:0] SRC_CONGEST_ADDR = 42;
    typedef enum {READ, WRITE, CONTROL} stim_t;
    randc stim_t type; // Змінна типу перелічення
    rand bit [31:0] len, src, dst;
    bit congestion_test;
    constraint c_stim {
        len < 1000;
        len > 0;
        src inside {0, [2:10], [100:107]};
        if (congestion_test) {
            dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
        }
    }
```

```

    }
}
endclass

```

У прикладі (лістинг 23.3) представлений клас коду із блоком обмежень, що містить кілька виразів. Перші два контролюють значення для змінної `len`. Як видно із прикладу, змінна може з'являтися в множинних виразах. На один вираз допускається використовувати тільки один оператор відношення (`<`, `<=`, `==`, `>=`, або `>`).

Лістинг 23.3. Змінні обмеження описані в певному порядку.

```

class bad;
    rand bit [15:0] a, b, c;
    constraint good {0 < a;          // Правильний опис
                    a < b;
                    b < c;}
    constraint bad {0 < a < b < c;} // Неправильно, призведе до помилки
endclass

```

Безліч значень може бути створена за допомогою оператора `inside` (лістинг 23.4). SystemVerilog створює значення змінних з рівною ймовірністю, якщо немає інших умов або обмежень для змінних. Можна також використовувати змінні для вказівки безлічі. Рядок `len inside {[1:999]}` заміщає описані вище вирази.

Лістинг 23.4. Псевдовипадкова безліч значень.

```

rand int c;          // Псевдовипадкова змінна
int lo, hi;          // Звичайна змінна, використовувана для опису меж
constraint c_range {
    c inside {[lo:hi]}; // lo <= c <= hi
}

```

У прикладі змінні `lo` і `hi` визначають діапазон можливих значень. При цьому описується інверсний діапазон - вибираються значення, що не входять у зазначену безліч:

```

constraint c_range {
    !(c inside {[lo:hi]}); // c < lo або c > hi
}

```

Всі значення безлічі мають однакову ймовірність вибору, навіть якщо вони з'являються в тексті кілька разів:

```

constraint c_even_weight {
    (c inside {0,1,1,1,1,1}); // 0 і 1 мають однакову ймовірність
}

```

Якщо необхідно додати іншу вагу деяким значенням змінних, це можна зробити за допомогою оператора `dist`.

23.2. Використання масивів як границі безлічі псевдовипадкових значень

Допускається виконувати вибір значення з безлічі, представленого масивом. У прикладі (лістинг 23.5) - це вибір дня тижня, описаного списком перелічень. Список `choices` може бути легко змінений, якщо зробити його псевдовипадковою змінною. У випадку використання оператора `randc` система генерування перебирає всі можливі значення перед повтором вже привласнених чисел.

Лістинг 23.5. Вибір з безлічі можливих значень.

```
class Days;
    typedef enum {SUN, MON, TUE, WED,
                  THU, FRI, SAT} DAYS_E;
    DAYS_E choices[$];
    rand DAYS_E choice;
    constraint cday {choice inside choices;}
endclass

Days days;
initial begin
    days = new;
    days.choices = '{Days::SUN, Days::SAT};
    assert (days.randomize());
    $display("Random weekend day %s\n", days.choice.name);
    days.choices = '{Days::MON, Days::TUE, Days::WED, Days::THU, Days::FRI};
    assert (days.randomize());
    $display("Random week day %s", days.choice.name);
end
```

Якщо необхідно динамічно додати або видалити дані з безлічі, то варто обережно використовувати оператор `inside` через його особливості. Наприклад, існує безліч значень у черзі, з яких повинно бути обрано й видалено тільки одне. Це зажадає створити N обмежень, де N - кількість елементів, що залишаються в черзі. Краще використовувати `randc` (лістинг 23.6), що пов'язано з масивом виборів. Вибір значення `randc` займає значно менше часу, тоді як створення великої кількості обмежень вимагає значних тимчасових витрат, особливо якщо констант більше 10.

Лістинг 23.6. Використання `randc` для вибору значень масиву у випадковому порядку.

```
class RandcInside;
    int array[]; // Значення для вибору
    randc bit [15:0] index; // Індексація масиву
    function new(input int a[]); // Конструктор і ініціалізація
        array = a;
    endfunction
    function int pick; // Повернення нового результату вибору
        return array[index];
    endfunction
    constraint c_size {index < array.size;}
```

```

endclass
initial begin
    RandcInside ri;
    ri = new('{1,3,5,7,9,11,13});
    repeat (ri.array.size) begin
        assert(ri.randomize());
        $display("Picked %2d [%0d]", ri.pick(), ri.index);
    end
end
end

```

23.3. Зважений розподіл

Оператор `dist` дозволяє задавати зважене випадкове генерування значень таким чином, що деякі числа (коди) будуть з'являтися частіше, ніж інші. Оператор `dist` (лістинг 23.7) приймає список значень і ваг, розділених символами `:=` або `:/`. Значення й ваги можуть бути константами або змінними. Значення може бути задано числом або діапазоном виду `[lo:hi]`. Оператор `:=` означає однакову вагу для кожного описаного діапазону, у той час як `:/` означає - вага повинна бути поділена між всіма значеннями діапазону. Вага визначається відсотками й не може бути в сумі більше 100.

У прикладі (лістинг 23.7) `src` отримує значення 0, 1, 2 або 3. Вага 0 дорівнює 40, а 1, 2 і 3 - 60, усього в сумі виходить 220. У цьому випадку ймовірність вибору 0 дорівнює 40/220, а ймовірності появи 1, 2 або 3 - 60/220. Далі `dst` отримує значення 0, 1, 2 або 3. Вага 0 дорівнює 40, а 1, 2 і 3 розділили загальну вагу 60, отже, повна вага всіх значень дорівнює 100. У цьому випадку ймовірність вибору нуля дорівнює 40/100, а ймовірність вибору 1, 2 або 3 дорівнює 20/100.

Лістинг 23.7. Зважене псевдовипадкове генерування значень.

```

rand int src, dst;
constraint c_dist {
    src dist {0:=40, [1:3]:=60};
        // src = 0, weight = 40/220
        // src = 1, weight = 60/220
        // src = 2, weight = 60/220
        // src = 3, weight = 60/220
    dst dist {0:/40, [1:3]:/60};
        // dst = 0, weight = 40/100
        // dst = 1, weight = 20/100
        // dst = 2, weight = 20/100
        // dst = 3, weight = 20/100
}

```

Використання змінних для ваги вибору значення дозволяє динамічно міняти розподіл випадкових змінних, як це представлено в лістингу 23.8.

Лістинг 23.8. Динамічна зміна розподілу ваг.

// Оператори шини довжиною в байт, слово або довге слово
class BusOp;

```

        // Довжина операндів
typedef enum {BYTE, WORD, LWRD } length_t;
rand length_t len;
        // Псевдовипадкові ваги для обмежень dist
bit [31:0] w_byte=1, w_word=3, w_lwr=5;
constraint c_len {
        len dist {BYTE := w_byte,    // Вибір псевдовипадкового значення
        WORD := w_word,              // Використання довжини
        LWRD := w_lwr;               // Змінна вага
        }
}
endclass

```

23.4. Двонаправлені обмеження

У наступному прикладі (лістинг 23.9) SystemVerilog розглядає всі чотири обмеження на змінні одночасно: b менше d, що, у свою чергу, менше 30. Але b має обмеження на рівність c, що повинно бути більше 25. Навіть якщо тут немає прямого обмеження на мінімальне значення d, дана обставина формує обмеження для змінної c (табл. 23.1).

Лістинг 23.9. Двонаправлені обмеження.

```

rand logic [15:0] b, c, d;
constraint c_bidir {
        b < d;
        c == b;
        d < 30;
        c > 25;
}

```

Таблиця 23.1. Рішення для двонаправлених обмежень

Рішення	b	c	d
1	26	26	27
2	27	27	28
3	28	28	29

23.5. Умовні обмеження

Звичайно обмеження змінних активні на всьому проміжку часу тестування. У ситуаціях, коли необхідно відключити дію обмежень на якийсь час, SystemVerilog пропонує два умовних оператори -> і if-else:

```

// Блок обмежень із оператором імплікації
class BusOp;
...
constraint c_io {
        (io_space_mode) -> addr[31] == 1'b1;
}

```

Якщо присутні вирази для обох гілок умови true-false, то краще використувати оператор if-else:


```
// Блок обмежень із оператором if-else
class BusOp;
...
constraint c_len_rw {
    if (op == READ)
        len inside {[BYTE:LWRD]};
    else
        len == LWRD;
}
```

У блоках обмеження використовуються фігурні дужки {}, оскільки це декларативний код. У процедурному коді застосовується begin...end.

23.6. Використання арифметичних операторів

Для обмежень можуть бути використані прості арифметичні оператори: додавання, віднімання, виділення бітів, зсув. За замовчуванням обмеження мають розмір 32 біта.

Також для завдання обмежень може бути використаний оператор solve...before (лістинг 23.10). Конструкція solve...before не впливає на простір вибору рішень, а тільки на ймовірність формування значень змінних. Вибір стану x виконується з рівною ймовірністю для (0, 1). З 1000 випадкових значень x 500 буде дорівнювати 0, а 500 - 1. Коли x дорівнює 0, у повинен дорівнювати 0. Коли x приймає значення 1, то у може бути 0, 1, 2 або 3 з рівною ймовірністю (табл. 23.2).

Таблиця 23.2. Значення x і у для обмежень solve x before y

Рішення	x	y	Ймовірність
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/9
H	1	3	1/8

Лістинг 23.10. Клас із імплікацією й solve...before.

```
class SolveBefore;
    rand bit x; // 0 or 1
    rand bit [1:0] y; // 0, 1, 2, or 3
    constraint c_xy {
        (x==0) -> y==0;
        solve x before y;
    }
endclass
```

Але у випадку використання обмеження solve у before х буде отриманий інший розподіл, заданий табл. 23.3.

Таблиця 23.3. Значення х і у для обмежень solve у before х

Рішення	х	у	Ймовірність
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4

23.7. Керування декількома блоками обмежень

Клас може містити кілька блоків обмежень. Наприклад, одна група керує розрядністю даних транзакції, а інша - довжиною транзакції. Тоді можна застосовувати метод `constraint_mode()` для того, щоб включати або відключати обмеження. При використанні формату `<ім'я обмеження>.constraint_mode()` цей метод керує єдиним обмеженням.

Лістинг 23.11. Використання `constraint_mode`.

```
class Packet;
    rand int length;
    constraint c_short {length inside {[1:32]}; }
    constraint c_long {length inside {[1000:1023]}; }
endclass
Packet p;
initial begin
    p = new;
// Створення довгих даних long через відключення обмежень для коротких short
    p.c_short.constraint_mode(0);
    assert (p.randomize());
    transmit(p);
// Створення коротких даних через відключення всіх обмежень,
// а потім дозволяється використання обмежень для коротких даних
    p.constraint_mode(0);
    p.c_short.constraint_mode(1);
    assert (p.randomize());
transmit(p);
end
```

23.8. Обмеження для керування правильністю даних

Обмеження можуть бути використані для керування формуванням тільки припустимих для моделі значень змінних. Наприклад, у наступному коді операція read-modify-write дозволена тільки для довгих даних.

```
// Перевірка довжини змінної з використанням обмеження valid
class BusTrans;
    rand enum {BYTE, WORD, LWRD, QWRD} length;
    rand enum {READ, WRITE, RMW, INTR} opc;
    constraint valid_RMW_LWRD {
        (opc == RMW) -> length == LWRD;
    }
endclass
```

23.8.1. Лінійні обмеження.

Додаткове керування обмеженнями реалізується за допомогою конструкції `randomize() with` (лістинг 23.12).

Лістинг 23.12. Оператор `randomize() with`.

```
class Transaction;
    rand bit [31:0] addr, data;
    constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t = new();
initial begin
    int s;
    t = new();
    // addr може приймати значення 50-100, 1000-1500, data < 10
    assert(t.randomize() with {addr >= 50; addr <= 1500; data < 10;});
    driveBus(t);
    // Формування певного значення для addr, data > 10
    assert(t.randomize() with {addr == 2000; data > 10;});
    driveBus(t);
end
```

З метою відключення конфліктуючих обмежень для змінних можна використувати функцію `constraint_mode`.

23.9. Функції `pre_randomize` і `post_randomize`

SystemVerilog має функції для нелінійного розподілу даних, що генеруються випадково (рис. 23.1). Це `void`-функції - функції, що не повертають значення. Однак на відміну від завдань вони не допускають використання параметрів часу. За допомогою даних функцій можна виконувати дії перед запуском функції формування псевдовипадкових умов або відразу після неї (лістинг 23.13).

Лістинг 23.13. Вбудований U-образний розподіл.

```
class Bathtub;
    int value;
    // Псевдовипадкова змінна з U-образним розподілом
    int WIDTH = 50, DEPTH=4, seed=1;
    function void pre_randomize();
        // Обчислення лівої частини кривої
        value = $dist_exponential(seed, DEPTH);
        if (value > WIDTH) value = WIDTH;
    // Псевдовипадкове розміщення крапок у лівій або правій частині кривої
endclass
```

```

        if ($random_range(1))
            value = WIDTH - value;
        endfunction
    endclass

```

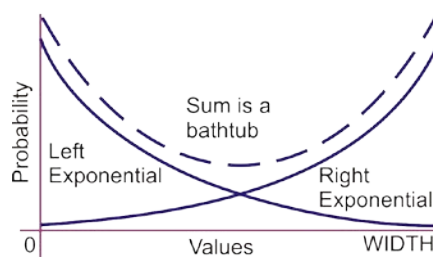


Рис. 23.1. Вбудований U-образний розподіл

Тут існують такі функції: `$dist_exponential` - експонентне загасання (див. рис. 23.1); `$dist_normal` - дзвонообразний розподіл; `$dist_poisson` - дзвонообразний розподіл; `$dist_uniform` - плоский розподіл; `$random` - плоский розподіл, повертає знакове 32-бітове значення; `$urandom` - плоский розподіл, повертає беззнакове 32-бітове значення; `$urandom_range` - плоский розподіл через діапазон.

23.10. Настроювані тести з випадковою генерацією

Для вказівки меж псевдовипадкової генерації можуть бути використані змінні:

```

class bounds;
    rand int size;
    int max_size = 100;
    constraint c_size {
        size inside {[1:max_size]};
    }
endclass

```

За замовчуванням, клас привласнює випадковій змінній `size` значення від 1 до 100. Змінюючи величину змінної `max_size`, можна модифікувати верхню межу діапазону.

Є також можливість використовувати змінну з обмеженнями `dist` для включення й відключення значень і діапазонів. У прикладі (лістинг 23.14) кожна команда шини має різні вагові змінні.

Лістинг 23.14. Обмеження `dist` з вагами змінної

```

typedef enum (READ8, READ16, READ32) read_t;
class ReadCommands;
    rand read_t read_cmd;
    int read8_wt=1, read16_wt=1, read32_wt=1;
    constraint c_read {
        read_cmd dist {READ8 := read8_wt,
            READ16 := read16_wt,

```

```

        READ32 := read32_wt};
    }
endclass

```

23.11. Використання не випадкових значень

Оператор `rand_mode` дозволяє зробити змінну детермінованою в часі або не випадковою (лістинг 23.15).

Лістинг 23.15. Відключення псевдовипадкової генерації значень за допомогою `rand_mode`.

```

// Клас Packet зі змінними length і payload
class Packet;
    rand bit [7:0] length;
    rand bit [7:0] payload[];
    constraint c_valid {length > 0; payload.size == length;}
    function void display(string msg);
        $write("Packet len=%0d, payload size=%0d, bytes = ", length,
payload.size);
        for(int i=0; (i<4 && i<payload.size); i++)
            $write(" %0d", payload[i]);
        $display;
    endfunction
endclass
Packet p;
initial begin
    p = new();
    // Псевдовипадкове генерування всіх змінних
    assert (p.randomize());
    p.display("Simple randomize");
    // Відключення псевдовипадкової генерації значень length, потім
    // генерування значень для пакета
    p.length.rand_mode(0);
    p.length = 42;
    assert (p.randomize());
    p.display("Randomize with rand_mode");
end

```

Якщо виконується псевдовипадкове генерування значень сигналів об'єкта, а потім деякі змінні модифікуються, то необхідно перевірити виконання обмежень. Виклик `handle.randomize(null)` SystemVerilog розглядає всі змінні як не випадкові й гарантує, що всі обмеження будуть виконуватися (лістинг 23.16).

Лістинг 23.16. Включення й відключення обмежень `constraint_mode`.

```

class Instruction;
    rand OPCODE_T opcode;
    ...
    constraint c_no_operands {
        opcode == NOP || opcode == HALT;}
    constraint c_one_operand {

```

```

opcode == CLR || opcode == NOT;}

endclass
Instruction instr;
initial begin
    instr = new;
    // Генерування instruction без операндів
    instr.constraint_mode(0);           // Відключення всіх обмежень
    instr.c_no_operands.constraint_mode(1);
    assert (instr.randomize());
    // Генерування instruction з одним операндом
    instr.constraint_mode(0);           // Відключення всіх обмежень
    instr.c_one_operand.constraint_mode(1);
    assert (instr.randomize());
end

```

23.12. Використання зовнішніх визначень обмежень

Обмеження визначаються за межами пакета. У прикладі (лістинг 23.17) оголошені обмеження `c_external`, які задані в програмі `test`.

Лістинг 23.17. Клас із зовнішніми обмеженнями і їхнє використання.

```

// Визначення класу
class Packet;
    rand bit [7:0] length;
    rand bit [7:0] payload[];
    constraint c_valid {length > 0;
                        payload.size == length;}
    constraint c_external;
endclass
// Визначення зовнішніх обмежень у програмі
program test;
    constraint Packet::c_external {length == 1;}
    ...
endprogram

```

23.13. Проблеми рандомізації

У прикладі (лістинг 23.18) обмеження `total_len` визначає, що сума пари змінних (`pkt1_len + pkt2_len`) не повинна перевищувати 64. При цьому припустимі значення (32, 32) і (2, 62). Однак цій умові задовольняє також пара значень (-64, 128). Якщо необхідно уникнути появи негативних значень, то варто використовувати в тесті беззнакові типи даних (лістинг 23.19).

Лістинг 23.18. Знакові змінні є причиною проблем псевдовипадкової генерації.

```

class SignedVars;
    rand byte pkt1_len, pkt2_len;
    constraint total_len {
        pkt1_len + pkt2_len == 64;
    }
endclass

```

Лістинг 23.19. Формування значень для беззнакових 32-бітових змінних.

```
class Vars32;
    rand logic [31:0] pkt1_len, pk2_len;    // беззнаковий тип
    constraint total_len {
        pkt1_len + pk2_len == 64;
    }
endclass
```

Навіть дана версія коду призводить до помилок. Більші значення `pkt1_len` і `pkt2_len`, як `32'h80000040` і `32'h80000000`, при додаванні дають `32'd64`. Варто подумати про додавання іншої пари констант для обмеження значень цих двох змінних. Але кращий підхід - зробити їх меншої довжини й уникнути використання 32-бітових змінних в обмеженнях (лістинг 23.20).

Лістинг 23.20. Формування значень для беззнакових 8-бітових змінних.

```
class Vars8;
    rand logic [7:0] pkt1_len, pk2_len;    // 8-бітів
    constraint total_len {
        pkt1_len + pk2_len == 8'd64;    // 8-бітів
    }
endclass
```

23.14. Обмеження, ітеративні й масиви

Форму розподілу значень сигналів можна змінювати за допомогою оператора `foreach` і декількох функцій масивів. Найпростішим обмеженням є функція `size`. Вона застосовується при визначенні кількості елементів для динамічного масиву або черги:

```
class dyn_size;
    rand reg [31:0] d[];
    constraint d_size {d.size inside {[1:10]}; }
endclass
```

У прикладі використання обмеження `inside` дозволяє задати нижню й верхню границі масиву.

23.14.1. Сума елементів.

Можна послати випадковий масив даних у проект або використовувати його для керування потоком. Наприклад, задано інтерфейс, що передає чотири слова даних. Слова можуть бути послані послідовно або протягом декількох циклів. Сигнал `strobe` означає наявність правильних даних. Існує кілька припустимих шаблонів стробування, які можуть бути використані для передачі значень під час виконання десяти циклів (рис. 23.2, лістинг 23.21).

Ці шаблони можуть бути створені за допомогою випадкових масивів. Обмеження використовують чотири біти й функцію підсумовування `sum`.

Лістинг 23.21. Використання шаблону стробування.

```
// Клас для формування псевдовипадкового шаблону стробування
parameter MAX_TRANSFER_LEN = 10;
```

```

class StrobePat;
    rand bit strobe[MAX_TRANSFER_LEN];
    constraint c_set_four { strobe.sum == 3'h4; }
endclass
// Використання класу шаблону стробування
initial begin
    StrobePat sp;
    int count = 0;           // Індекс у масиві даних
    sp = new();
    assert (sp.randomize);
    foreach (sp.strobe[i]) begin
        bus.cb.strobe = sp.strobe[i];
        // If strobe is enabled, drive out next data word
        if (sp.strobe[i])
            bus.cb.data = data[count++];
    end
end
end

```

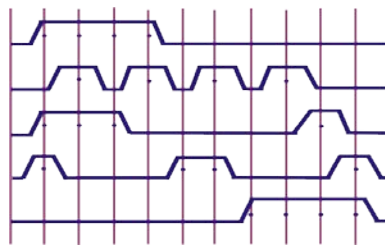


Рис. 23.2 Випадкова форма сигналів стробування

Таким чином, у розділі розглянуті типові процедури створення псевдовипадкових тестів на основі використання конструкцій мови System Verilog. Процедури можуть бути досить корисними при створенні тестів перевірки справності для функціональних модулів цифрових систем на стадії проектування архітектури виробу.

23.15. Контрольні запитання й завдання

1. Де застосовується псевдовипадкова генерація значень змінних?
2. Чим відрізняється генерування значень змінних при використанні ключових слів `rand` і `randc`?
3. Створити клас, що містить поля `a` і `b`, які можуть приймати випадкові значення чисел в інтервалі від 0 до 100. При цьому значення `a` повинно бути завжди менше `b`.
4. Створити клас, який містить поле `a`, що приймає значення 1, 10, 20 і 100. Імовірність появи значення 1 дорівнює 10, імовірність появи чисел 10 і 20 дорівнює 20, а ймовірність значення 100 дорівнює 50.
5. Створити клас, що містить поле `a`, значення якого вибираються з масиву `b`, довжиною 20.

24. ФУНКЦІОНАЛЬНЕ ПОКРИТТЯ

Представлено моделі й процедури створення різних типів функціональних покриттів, необхідних для оцінки якості тестування й верифікації цифрових проектів, описаних на системних мовах опису апаратури.

Мета - розробка надлишкових відносно HDL-моделі структур даних для оцінювання якості тестів верифікації проєктованих цифрових виробів, орієнтованих на вичерпну перевірку функціональностей, пошук константних дефектів і перемешованих несправностей, що призводять пристрій у непридатний стан.

Завдання - створення типових процедур для синтезу кошика (безлічі найбільш істотних параметрів і режимів) функціонального покриття проєктованого виробу для оцінки якості тестів, орієнтованих на перевірку компонентів цифрових систем на кристалах.

Джерела: [1-5] - методи моделювання несправностей для оцінки якості детермінованих і псевдовипадкових тестів цифрових структур; [6-11] - технології й програмні процедури створення функціональних покриттів на основі застосування System Verilog для оцінки якості тестування й верифікації проєктованих цифрових систем.

24.1. Покриття (Coverage)

Покриття (тестом функціонального простору й несправностей) є модельна надмірність функціонального HDL-коду, орієнтована на оцінку якості проєктованого виробу шляхом моделювання тестів для визначення повноти покриття (перевірки) функціональних і спеціальних режимів цифрових систем на кристалах, заданих специфікацією.

Для рішення завдання функціональної верифікації існує велика кількість програмних ресурсів, орієнтованих на підтвердження правильності функціонування складних систем або виявлення помилок у процесі проєктування. Однією з основних технологій оцінювання якості тестування є повнота покриття тестом функціональних режимів керування, передачі й перетворення даних, що дозволяє визначити у відсотках, яка частина об'єкта, що верифікується, перевірена й вільна від помилок проєктування.

Існує дві метрики покриття для оцінювання якості тестів або якості верифікації: 1) code coverage - автоматично генерується список істотних змінних для перевірки на основі аналізу коду проєкту; 2) functional coverage застосовує користувальницьку метрику, яка визначає частину специфікації проєкту, що підлягає перевірці й зазначена в плані тестування. Використовується для виміру у відсотках потрібного сценарію, виняткових ситуацій, компонентів специфікації й умов, які були перевірені й оцінені щодо правильного функціонування проєктованого виробу.

Є два типи функціонального покриття: 1) Визначається користувачем і не може бути отриманий автоматично з коду проєкту. 2) Ґрунтується на специфікації проєкту й не залежить від реального коду моделі або структури.

Оскільки властивості функціонального покриття повністю визначаються користувачем, то для їхнього опису не потрібно більших тимчасових витрат. Тут необхідний структурний підхід до верифікації. Хоча функціональне покриття скорочує в цілому тимчасові витрати по верифікації й підвищує продуктивність праці розроблювача і якість проектування, його недоліки, пов'язані із синтезом і аналізом тестів, можуть перешкодити його застосуванню.

SystemVerilog пропонує конструкції, що спрощують створення специфікації для функціонального покриття моделі виробу. Специфікація може бути ефективно створена за допомогою SystemVerilog-симуляторів. Вона дозволяє здійснювати маніпулювання даними з покриття й пропонує інструменти аналізу, що підвищують продуктивність розробки якісного тесту. Поліпшена безліч вхідних наборів може виконувати тестування функціональності пристрою у виняткових ситуаціях і за необхідним сценарієм.

Конструкції функціонального покриття в SystemVerilog дозволяють виконувати такі дії: 1) Збір інформації про покриття змінних і виразів, а також вивчення перехресного покриття між ними. 2) Автоматичне або користувацьке створення кошиків покриття. 3) Фільтрація умов на декількох рівнях ієрархії покриття. 4) Використання подій і послідовностей для автоматичного перемикавання умови для збору значень (sampling) покриття. 5) Отримання покриття викликів, процедур і черг. 6) Використання необов'язкових директив для керування й регулювання покриттям.

24.2. Визначення моделі покриття: **covergroup**

Конструкція **covergroup** успадковує специфікацію моделі покриття й може включати: 1) Події синхронізації, що визначають моменти зчитування значень. 2) Безліч точок покриття (coverage points). 3) Перехресне покриття між крапками моделі для оцінки якості тестів. 4) Необов'язкові формальні аргументи. 5) Опції покриття.

Конструкція **covergroup** є типом, визначеним користувачем. Після формування запису конструкція може мати кілька копій, які можуть бути реалізовані в різних контекстах. Конструкція **covergroup** створюється в пакеті, модулі, програмі, інтерфейсі або класі. Вона може збирати значення будь-яких видимих змінних програми або модуля, сигналів з інтерфейсу або будь-яких інших сигналів за допомогою ієрархічних імен. Група покриття класу може спостерігати його змінні й вбудовані класи. Синтаксис:

```
covergroup covergroup_identifier [ ( [ tf_port_list ] ) ] [ coverage_event ] ;
    { {attribute_instance} coverage_spec |
      {attribute_instance} coverage_option ;
    }
endgroup [ : covergroup_identifier ]
```

Тут **covergroup_identifier** - ім'я групи покриття, **tf_port_list** - список формальних параметрів, **coverage_event** - подія синхронізації для вибірки значень спостережуваних змінних. Тіло групи покриття може включати опції **coverage_spec**, записувані у форматі:

```
option.member_identifier = expression | type_option.member_identifier = expression,
```

а також опції покриття `coverage_option`, що описують його точки або перехресне покриття. Для створення копії груп покриття використовується оператор `new`:

```
covergroup_identifier [variable_identifier] = new [ ( list_of_arguments ) ],
```

де `list_of_arguments` - список реальних аргументів.

Ідентифікатор, пов'язаний з декларацією `covergroup`, визначає ім'я моделі покриття. Використовуючи це ім'я, можна створити довільне число копій моделей покриття. Наприклад,

```
// Декларація групи
```

```
covergroup cg;
```

```
...
```

```
endgroup
```

```
// Створення копії групи покриття й ініціалізація
```

```
cg cg_inst = new;
```

У запропонованому прикладі визначена модель покриття `covergroup` з іменем `cg`. За допомогою оператора `new` створена `cg_inst` копія `cg`. Конструкція `covergroup` може містити список необов'язкових аргументів. Вони обчислюються після виконання оператора `new`. Аргумент `ref` дозволяє розрізнити змінні, спостережувані кожною копією `covergroup`. Вхідні аргументи передають значення для оператора `new`.

Якщо зазначена подія синхронізації, то вона визначає моменти часу для збору значень із точок покриття. У протилежному випадку користувач повинен задавати синхронізацію за допомогою процедурних операторів, наприклад, через вбудований метод `sample()`. Опція `strobe` може бути використана для зміни моментів зчитування даних. Коли опція `strobe` не встановлена, значення точок покриття зчитуються за подією копії його моделі. Якщо подія синхронізації зустрічається кілька разів під час моделювання, то значення точок покриття повинні бути також лічені кілька разів. Як альтернатива для синхронізації може бути використаний вираз блокової події, де зчитування контрольних точок покриття може бути виконано на початку або в кінці роботи заданого іменного блоку, завдання, функції або методу класу. Якщо вираз блокової події починається із ключового слова `begin`, за яким слідує ієрархічний ідентифікатор, що означає іменний блок, завдання, функцію або метод класу, то подія перемикається до виконання першого оператора зазначених конструкцій. Вираз блокової події, що описано за допомогою ключового слова `end`, виконується відразу після останнього оператора зазначеної конструкції.

Група покриття `covergroup` може містити одну або кілька крапок, які задаються змінною або виразом. З кожною крапкою покриття зв'язується безліч значень змінної, необхідних для перевірки, що оформляється в структурний компонент покриття, називаний кошиком. Вона може бути явно визначена користувачем або автоматично створена за допомогою інструментів моделювання:

```
enum { red, green, blue } color;
```

```
covergroup g1 @(posedge clk);
    c: coverpoint color;
endgroup
```

Попередній приклад представляє групу покриття g1 з однією точкою, пов'язаною зі змінною color. Значення color вибирається за позначеною подією синхронізації - позитивним фронтом clk.

Оскільки точка покриття не має явно визначених кошиків, то інструмент автоматично створює три згаданих компоненти, по одному для кожного значення типу перелічення. Група покриття може задавати перехресні точки між двома або більше точками покриття або змінними. Допускається використання будь-яких комбінацій більш, ніж двох змінних або попередньо декларованих точок покриття.

Наприклад:

```
enum { red, green, blue } color;
bit [3:0] pixel_adr, pixel_offset, pixel_hue;
covergroup g2 @(posedge clk);
    Hue: coverpoint pixel_hue;
    Offset: coverpoint pixel_offset;
    Ax: cross color, pixel_adr;    //перетинання 2 змінних (явне декларування точок покриття)
    all: cross color, Hue, Offset; // перетинання 1 змінної й 2 точок покриття
endgroup
```

У прикладі створюється група покриття g2, що містить дві точки й дві декларації перетинання покриттів cross. Явно задані точки покриття з мітками Offset і Hue визначені для змінних pixel_offset і pixel_hue. Для змінних color і pixel_adr задається аналіз спільного покриття. У групі покриття можуть бути також описані одна або кілька опцій, що контролюють і регламентують структуру й спосіб збору інформації про покриття. Опції можуть бути задані для групи покриття в цілому або для окремих її пунктів. В SystemVerilog група покриття повинна бути визначена на відповідному рівні абстракції. Він може розмежовувати testbench і проект у транзакторі, який виконує читання й запис даних в класі, що конфігурує середовище. Захват будь-яких транзакцій необхідно виконувати при реальному отриманні їх з моделі проекту DUT. Якщо внести помилку в транзакцію, що приводить до її скасування, необхідно визначитися, як ця ситуація буде впливати на функціональне покриття. Варто використовувати різні крапки покриття, які будуть створені для керування помилками. Клас може містити кілька груп покриття. Це дозволяє застосовувати різний набір груп покриття, дозволяючи або забороняючи їх якщо буде потреба. Групи можуть мати різні перемикачі, які дозволяють збирати дані з різних джерел.

Приклад з лістинга 24.1 представляє опис і використання групи покриття CovPort у програмі.

Лістинг 24.1. Функціональне покриття простих об'єктів.

```
program automatic test(busifc.TB ifc);
    class Transaction;
```

```

        rand bit [31:0] data;
        rand bit [ 2:0] port;          // Номер одного з восьми портів
    endclass
    covergroup CovPort;
        coverpoint tr.port;            // Точка, спостережувана групою покрит-
    тя
    endgroup
    Transaction tr = new;
    initial begin
        CovPort ck = new;              // Створення копії групи
        repeat (32) begin              // Виконання декількох циклів
            assert(tr.randomize);       // Створення транзакції
            ifc.cb.port <= tr.port;     // і передача
            ifc.cb.data <= tr.data;     // в інтерфейс
            ck.sample();                // Збір даних про покриття
            @ifc.cb; // Wait a cycle
        end
    end
endprogram

```

У наступному прикладі (лістинг 24.2) група покриття розміщена в класі, тому вона не вимагає для себе окремого імені реалізації.

Лістинг 24.2. Функція покриття всередині класу.

```

class Transactor;
    Transaction tr;
    mailbox mbx_in;
    covergroup CovPort;
        coverpoint tr.port;
    endgroup
    function new(mailbox mbx_in);
        CovPort = new;                // Створення копії групи покриття
        this.mbx_in = mbx_in;
    endfunction
    task main;
        forever begin
            tr = mbx_in.get;           // Отримання наступної транзакції
            ifc.cb.port <= tr.port;     // Передача даних в DUT
            ifc.cb.data <= tr.data;
            CovPort.sample();           // Збір покриття
        end
    endtask
endclass

```

24.2.1. Перемикання групи покриття

Існує два основних моменти у функціональному покритті - це отримання даних і час їхнього прийому. Коли дані готові, testbench виконує перемикання групи покриття. Це можна зробити за допомогою завдання sample() (лістинги 24.1 і 24.2) або блокуючого оператора у визначенні групи покриття

covergroup. Блокуючий вираз може використовувати wait або @ для блокування сигналів або подій. Метод sample() застосовується для простого перемикавання групи покриття або коли існує кілька реалізацій групи покриття, що перемикаються окремо. Блокуючі оператори covergroup варто застосовувати якщо буде потреба використання існуючих сигналів або подій для перемикавання групи покриття.

24.3. Застосування групи покриття із класами

Розміщення групи покриття у визначенні класу забезпечує найпростіший спосіб створення підмножини покриття для властивостей даного класу. Інтеграція покриття в клас забезпечує інтуїтивний і ясний механізм для визначення моделі покриття, пов'язаної із класом. Наприклад, у лістингу 24.3 показано, що значення полів m_x і m_y класу xyz збираються при кожній зміні змінної m_z. Якщо група покриття визначається всередині класу й у ній немає явної декларації змінної даної групи в межах класу, то вона створюється автоматично й має ім'я, яке група отримала при описі. Таким чином, у згаданому прикладі змінна cov1 відповідає групі покриття cov1.

Лістинг 24.3. Використання групи покриття в класах.

```
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;
    covergroup cov1 @m_z;      // Вбудовані групи покриття
        coverpoint m_x;
        coverpoint m_y;
    endgroup
    function new();
        cov1 = new;
    endfunction
endclass
```

Вбудована в клас група покриття може визначати модель покриття для захищених і локальних властивостей даного класу без можливості її перевизначення в класі-спадкоємці. Останній може мати кілька груп покриття. Ще один приклад (лістинг 24.4) представляє клас MC, що містить дві групи покриття. Перша група cv1 є членом класу, де значення змінної m_x збирається по кожному позитивному фронту сигналу clk. Значення локальної змінної m_z збирається групою cv2. Кожна з них використовує різні події синхронізації.

Лістинг 24.4. Кілька груп покриття в класі.

```
class MC;
    logic [3:0] m_x;
    local logic m_z;
    bit m_e;
    covergroup cv1
        @(posedge clk);
        coverpoint m_x;
```

```

    endgroup
    covergroup cv2
        @m_e ;
        coverpoint m_z;
    endgroup
endclass

```

Вбудована група покриття повинна бути явно реалізована за допомогою методу `new`. У протилежному випадку вона не створюється, що означає відсутність збору даних. У наступному прикладі (лістинг 24.5) група вбудованого покриття не має переданих у неї аргументів і використовує явну реалізацію для синхронізації з іншими об'єктами. У прикладі група покриття `Cov` належить класу `MyClass`, що також містить об'єкт класу `Helper` з ім'ям `m_obj`. Подія синхронізації для вбудованої групи покриття ставиться до поля `m_ev` об'єкта `m_obj`. Оскільки група `Cov` використовує `m_obj`, то останній створюється до копіювання групи `Cov`. Передача аргументів у вбудовану групу покриття ілюструється прикладом з лістинга 24.6.

Лістинг 24.5. Ініціалізація групи покриття в конструкторі.

```

class Helper;
    int m_ev;
endclass
class MyClass;
    Helper m_obj;
    int m_a;
    covergroup Cov @(m_obj.m_ev);
        coverpoint m_a;
    endgroup
    function new();
        m_obj = new;
        Cov = new; // Створення вбудованої групи покриття після об'єкта
                   //класу m_obj
    endfunction
endclass

```

Лістинг 24.6. Використання групи покриття з аргументами.

```

class C1;
    bit [7:0] x;
    covergroup cv (int arg) @(posedge clk);
        option.at_least = arg;
        coverpoint x;
    endgroup
    function new(int p1);
        cv = new(p1);
    endfunction
endclass
initial begin
    C1 obj = new(4);
end

```

24.4. Збір даних

Для збору даних використовуються спеціальні кошики. Інформація заноситься в них у процесі моделювання, а потім на основі зібраної інформації формуються статистичні дані.

24.4.1. Окремі кошики.

Для обчислення покриття в точці необхідно визначити загальне число можливих значень, які називаються доменом. Для одного кошика це може бути одно або кілька значень. Покриття визначається числом зібраних даних, розділеним на кількість кошиків у домені.

Точка покриття, що є 3-бітовою змінною, має 0:7 доменів і звичайно ділиться на 8 кошиків. Якщо під час моделювання в сім кошиків заносяться значення, звіт буде - 7/8 або 87,5% покриття в даній точці. Всі такі точки поєднуються для формування покриття всієї групи, а потім всі групи з'єднуються для визначення якості покриття (в %), що заноситься в базу даних моделювання.

24.4.2. Автоматичне створення кошиків.

SystemVerilog автоматично створює кошики для точок покриття. Для всіх значень, що мають n-розрядів, існує 2^n можливих значень. Таким чином, для 3-розрядних портів є 8 можливих значень сигналів.

Обмеження на число кошиків пов'язане з опцією `auto_bin_max`, що задає максимальну кількість автоматично створюваних кошиків. За замовчуванням – це 64. Якщо домен значень точки покриття змінної або виразу більше описаних в опції, то SystemVerilog виконує розподіл діапазону між кошиками. Наприклад, якщо 16-бітова змінна має 65 536 можливих значень, то кожний з 64 кошиків буде відповідати 1024 значенням. У реальності цей підхід може виявитися непрактичним. Тому краще обмежувати можливе число кошиків до 8-16 або явно вказувати їхню кількість. У наступному прикладі параметр `auto_bin_max` встановлюється рівним 2. Вхідний порт має розмір 3 розряди, що відповідає 8-ми можливим значенням. Першому кошику відповідають 4 молодших, а другому – 4 старших значення:

```
// Використання auto_bin_max, встановленого в 2
covergroup CovPort;
    coverpoint tr.port
        { options.auto_bin_max = 2; } // Розподіл на 2 кошики
endgroup

// Приклад звіту моделювання
Bin # hits at least
=====
auto[0:3] 15 1
auto[4:7] 17 1
=====
```


Приклад використовує опцію `auto_bin_max` для однієї точки покриття. Ця опція може бути застосована для цілої групи:

```
// Використання auto_bin_max для всіх точок покриття
covergroup CovPort;
    options.auto_bin_max = 2;          // Діє для port і data
    coverpoint tr.port;
    coverpoint tr.data;
endgroup
```

24.4.3. Спостереження виразів.

Можна спостерігати прості вирази, але завжди варто перевіряти звіт покриття, щоб бути впевненим в отриманні очікуваних значень. Може також знадобитися налаштування розрядності для підсумкового виразу.

Приклад (лістинг 24.7) представляє групу покриття, що збирає загальну довжину транзакції. Точка покриття має мітку, що полегшує читання звіту покриття. Вираз також має додаткову макетну (dummy) константу. Таким чином, довжина транзакції обчислюється з точністю 5 біт для максимального числа (32) кошиків, що генеруються автоматично.

Лістинг 24.7. Використання виразу в точці покриття.

```
class Transaction;
    rand bit [2:0] hdr_len;          // діапазон: 0:7
    rand bit [3:0] payload_len;     // діапазон: 0:15
    ...
endclass
Transaction tr;
covergroup CovLen;
    len16: coverpoint (tr.hdr_len + tr.payload_len);
    len32: coverpoint (tr.hdr_len + tr.payload_len + 5'b0);
endgroup
```

24.4.4. Користувальницькі кошики знаходять помилки.

Кошики, що генерують автоматично, прийнятні для анонімних значень даних, таких як лічильник, адреса. Для інших значень варто явно вказувати ім'я кошика з метою поліпшення точності аналізу й полегшення формування звіту покриття. SystemVerilog автоматично створює імена кошиків не тільки для типів даних перелічення, але й для інших типів, якщо буде потреба. Найпростіший спосіб визначення імен кошиків за допомогою синтаксису дужок [] (лістинг 24.8).

Лістинг 24.8. Визначення кошиків для довжини транзакції.

```
covergroup CovLen;
    len: coverpoint (tr.hdr_len + tr.payload_len + 5'b0)
        {bins len[] = {[0:23]}; }
endgroup

// Звіт про покриття для довжини транзакції
Bin # hits at least
=====
```

481

```
len_00 13 1
len_01 36 1
len_02 51 1
len_03 60 1
len_04 72 1
len_05 88 1
len_06 127 1
len_07 122 1
len_08 133 1
len_09 138 1
len_0a 115 1
len_0b 128 1
len_0c 125 1
len_0d 111 1
len_0e 115 1
len_0f 134 1
len_10 107 1
len_11 102 1
len_12 70 1
len_13 65 1
len_14 39 1
len_15 30 1
len_16 19 1
len_17 0 1
```

=====

24.4.5. Імена кошиків точок покриття.

Лістинг 24.9 представляє 4-бітову змінну `kind`, що має 16 можливих значень. Перший кошик має ім'я `zero` і виконує підрахунок моментів, коли змінна `kind` дорівнює 0. Наступні три значення 1-3 групуються в один кошик `lo`. Старші 8 значень 8-15 відповідають окремим кошикам `hi_8`, `hi_9`, `hi_a`, `hi_b`, `hi_c`, `hi_d`, `hi_e` і `hi_f`. Останній кошик `misc` зберігає всі інші значення, які не потрапили в попередні кошики - 4-7. Додаткова інформація групується за допомогою `{}` фігурних дужок.

Лістинг 24.9. Опис імен кошиків.

```
covergroup CovKind;
    coverpoint tr.kind {
        bins zero = {0};           // Кошик для kind==0
        bins lo = {[1:3]};         // Один кошик для значень 1:3
        bins hi[] = {[8:$]};       // 8 окремих кошиків
        bins misc = default;       // Один кошик для всіх інших
    }
endgroup // CoverKind

// Звіт для подання імен кошиків
Bin #    hits    at least
=====
hi_8     0       1
```

```

hi_9      5      1
hi_a      3      1
hi_b      4      1
hi_c      2      1
hi_d      2      1
hi_e      9      1
hi_f      4      1
lo        16     1
misc      15     1
zero      1      1
=====

```

Дані, що не потрапили в кошики, ігноруються. Для визначення функціонального покриття враховуються тільки дані, що попадають у кошик. Можна й бажано використовувати кошик default для всіх інших даних.

24.4.6. Умовне покриття.

Використовуючи ключове слово `iff`, можна створювати умови для вибору значень при формуванні покриття. Наприклад, у коді лістинга 24.10, а збір інформації не виконується в момент скидання пристрою, коли `reset = 1`. Інакше, можна використовувати функції `start` і `stop` для контролювання окремих реалізацій груп покриття, як це зроблено в прикладі лістинга 24.10, б.

Лістинг 24.10. Використання умов для збору інформації про покриття.

```

а) // З використанням ключового слова iff
covergroup CoverPort;
    // Інформація про покриття не збирається, якщо reset==1
    coverpoint port iff (!bus_if.reset);
endgroup

б) З використанням функцій stop і start
initial begin
    CovPort ck = new;          // Створення копії групи покриття
    // Виконання скидання зупиняє збір даних про покриття
    #1ns bus_if.reset = 1;
    ck.stop();
    #100ns bus_if.reset = 0;    // Завершення скидання
    ck.start();
    ...
end

```

24.4.7. Створення кошиків для типів перелічення.

Для типів перелічення SystemVerilog створює кошик для кожного можливого значення (лістинг 24.11).

Лістинг 24.11. Функціональне покриття типу перелічення.

```

typedef enum {INIT, DECODE, IDLE} fsmstate_t;
fsmstate_t pstate, nstate; // declare typed variables
covergroup cg_fsm;
    coverpoint pstate;

```

```

endgroup
// Звіт для auto_bin_max, рівного 2
Bin                # hits    at least
=====
auto_DECODE        11        1
auto_IDLE           11        1
auto_INIT           10        1
=====

```

Якщо необхідно групувати декілька значень в один кошик, то для цього варто визначити власний кошик. Опція `auto_bin_max` для типів перелічення не працює.

24.4.8. Покриття переходів.

Можна описувати певний перехід для точок покриття. У цьому випадку можна задавати не тільки певні значення, але й порядок їхнього проходження. Наприклад, у наступному коді виконується перевірка перемикання сигналу `port` з 0 в 1, 2 або 3:

```

//Опис переходів для точок покриття
covergroup CoverPort;
coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
}
endgroup

```

Кілька переходів можна легко описати за допомогою діапазону. Вираз $(1,2 \Rightarrow 3,4)$ відповідає чотирьом переходам $(1 \Rightarrow 3)$, $(1 \Rightarrow 4)$, $(2 \Rightarrow 3)$ і $(2 \Rightarrow 4)$. Допускається описувати переходи будь-якої довжини. Кожне значення в переході повинно бути отримано один раз. Таким чином, $(0 \Rightarrow 1 \Rightarrow 2)$ відрізняється від $(0 \Rightarrow 1 \Rightarrow 1 \Rightarrow 2)$ або $(0 \Rightarrow 1 \Rightarrow 1 \Rightarrow 1 \Rightarrow 2)$. Якщо є необхідність у повторенні значення 3 рази, варто використовувати форму $(0 \Rightarrow 1[*3] \Rightarrow 2)$. Щоб мати можливість повторення значення одиниці 3, 4 або 5 разів, можна використовувати вираз $1[*3:5]$.

24.4.9. Групові символи для опису станів і переходів.

Для свідомості множинних станів і переходів використовуються групові символи. Будь-який символ `x`, `z` або `?` розглядається як 0 або 1:

```

// Кошики для групових символів у точці покриття
bit [2:0] port;
covergroup CoverPort;
coverpoint port {
    wildcard bins even = {3'b??0};
    wildcard bins odd = {3'b??1};
}
endgroup

```

24.4.10. Ігноровані значення.

Якщо в деякій точці не можна досягти всіх можливих значень, то можна вказати ігноровані значення за допомогою кошика `ignore_bins`:

```
bit [2:0] low_ports_0_5;           // Використовуються тільки значення 0-5
covergroup CoverPort;
    coverpoint low_ports_0_5 {
        ignore_bins hi = {[6,7]}; // Ігнорування верхніх двох кошиків
    }
endgroup
```

Оригінальний діапазон `low_ports_0_5` має 3-розрядні значення з 0:7. У прикладі ігноруються останні два кошики, що дозволяє скоротити діапазон до 0:5. Якщо використовується опція `auto_bin_max` і вказуються ігноровані кошики, то останні не враховуються при обчисленні повноти покриття:

```
// Точки покриття з опцією auto_bin_max і ігноруванням кошиків ignore_bins
bit [2:0] low_ports_0_5;           // Використовуються тільки значення 0-5
covergroup CoverPort;
    coverpoint low_ports_0_5 {
        options.auto_bin_max = 4; // 0:1, 2:3, 4:5, 6:7
        ignore_bins hi = {[6,7]}; // Ігноруються старші 2 значення
    }
endgroup
```

24.4.11. Неприпустимі кошики.

Деякі приклади значень не тільки ігноруються, але не повинні з'являтися. Їх краще вказати в коді монітора для `testbench` або за допомогою `illegal_bins`. Це є додаткова перевірка правильності визначення кошиків або побудови `testbench`:

```
// Крапки покриття з illegal_bins
bit [2:0] low_ports_0_5;           // Використовуються тільки значення 0-5
covergroup CoverPort;
    coverpoint low_ports_0_5 {
        illegal_bins hi = {[6,7]}; // Повідомлення про помилку, якщо значення
з'явиться
    }
endgroup
```

24.5. Перехресне покриття

В SystemVerilog конструкція `cross` дозволяє комбінувати значення двох або більше точок покриття в групу. Оператор `cross` допускає тільки точки покриття або прості імена змінних. Якщо необхідно використовувати вирази, ієрархічні імена або змінні, то їх слід зазначити оператором `coverpoint`, а потім застосовувати цю мітку в операторі `cross`. У прикладі (лістинг 24.12) створюються точки покриття для змінних `tr.kind` і `tr.point`. Потім ці точки використовуються для створення умов перехресного покриття для всіх можливих комбінацій. SystemVerilog створює 128 (8x16) кошиків. Testbench із псевдовипадковою генерацією даних створює 200 транзакцій і формує звіт про покриття. Варто звернути увагу, що якщо були згенеровані всі можливі

значення kind і port для перехресного покриття, то тільки 1/8 частина всіх можливих комбінацій не була досягнута.

Лістинг 24.12. Базове перехресне покриття.

```
class Transaction;
    rand bit [3:0] kind;
    rand bit [2:0] port;
endclass
Transaction tr;
covergroup CovPort;
    kind: coverpoint tr.kind;    // Створення точки покриття kind
    port: coverpoint tr.port    // Створення точки покриття port
    cross kind, port;           // Перехресчування точок kind і port
endgroup

// Звіт про сумарне покриття для базового перехресного покриття
Cumulative report for Transaction::CovPort
Summary:
Coverage: 95.83
Goal: 100


| Coverpoint | Coverage | Goal | Weight |
|------------|----------|------|--------|
| kind       | 100.00   | 100  | 1      |
| port       | 100.00   | 100  | 1      |


Cross Coverage Goal Weight
Transaction::CovPort 87.50 100 1
Cross Coverage report
CoverageGroup: Transaction::CovPort
Cross: Transaction::CovPort
Summary
Coverage: 87.50
Goal: 100
Coverpoints Crossed: kind port
Number of Expected Cross Bins: 128
Number of User Defined Cross Bins: 0
Number of Automatically Generated Cross Bins: 112
Automatically Generated Cross Bins
kind port # hits at least
auto[0] auto[0] 1 1
auto[0] auto[1] 4 1
auto[0] auto[2] 3 1
auto[0] auto[5] 1 1
...
```

24.5.1. Мітки кошиків перехресного покриття.

Для підвищення читаності коду перехресного покриття кошики можуть мати мітки, представлені в лістингу 24.13.

Лістинг 24.13. Опис імен кошиків для перехресних покриттів.

```
covergroup CovPortKind;
    port: coverpoint tr.port
        {bins port[] = {[0:$]};
        }
    kind: coverpoint tr.kind
        { bins zero = {0};           // Кошик для kind==0
          bins lo = {[1:3]};         // Один кошик для значень 1:3
          bins hi[] = {[8:$]};       // Вісім роздільних кошиків
          bins misc = default;       // Один кошик для всіх, що залишилися
        }
cross kind, port;
endgroup

// Звіт про перехресне покриття з мітками кошиків
Summary
Coverage: 90.91
Number of Coverpoints Crossed: 2
Coverpoints Crossed: kind port
Number of Expected Cross Bins: 88
Number of Automatically Generated Cross Bins: 80
Automatically Generated Cross Bins
port kind # hits at least
=====
port_0 hi_8 3 1
port_0 hi_a 1 1
port_0 hi_b 4 1
port_0 hi_c 4 1
port_0 hi_d 4 1
port_0 hi_e 1 1
port_0 lo 7 1
port_0 misc 6 1
port_0 zero 1 1
port_1 hi_8 3 1
...
```

Скорочення числа можливих кошиків при використанні перехресного покриття виконується за допомогою ігнорування кошиків `ignore_bins` (лістинг 24.14).

Лістинг 24.14. Виключення кошиків з перехресного покриття.

```
covergroup CovCovport;
    port: coverpoint tr.port
        {bins port[] = {[0:$]};
        }
```

```

kind: coverpoint tr.kind
{
  bins zero = {0};           // Кошик для kind==0
  bins lo = {[1:3]};         // Один кошик для значень 1:3
  bins hi[] = {[8:$]};       // Вісім роздільних кошиків
  bins misc = default;       // Один кошик для всіх, що залишилися
}
cross kind, port {
  ignore_bins hi = binsof(port) intersect {7};
  ignore_bins md = binsof(port) intersect {0} &&
    binsof(kind) intersect {[9:10]};
  ignore_bins lo = binsof(kind.lo);
}
endgroup

```

24.5.2. Виключення точок покриття із загальної метрики покриття.

Загальне значення покриття для групи ґрунтується на використанні всіх простих точок перехресного покриття. Якщо деякі значення необхідно виключити, то їм можна привласнити нульову вагу (лістинг 24.15).

Лістинг 24.15. Опис ваги перехресного покриття.

```

covergroup CovPort;
  kind: coverpoint tr.kind
  {bins kind[] = {[0:$]};
    weight = 0;           // Не враховуються
  }
  port: coverpoint tr.port
  {bins zero = {0};
    bins lo = {[1:3]};
    bins hi[] = {[8:$]};
    bins misc = default;
    weight = 5;           // Враховуються
  }
  cross kind, port
    {weight = 10;}       // Додання точкам перетинання додаткової ваги
endgroup

```

24.5.3. Змішування даних з різних доменів.

Основна проблема для перехресного покриття може полягати в необхідності захоплення значень із різних тимчасових доменів. Може знадобитися інформація про отримання процесором події про половину заповнення кеш-пам'яті. Апаратура, що керує перериваннями, скоріше всього буде реалізована окремим блоком, що використовує інші тимчасові домени стосовно кеш-пам'яті. Рішенням буде створення незалежних тимчасових доменів для кеш-пам'яті й апаратури переривання. Потім створюються копії сигналів у вигляді тимчасових змінних. Вони використовуються для створення групи, що формує перехресне покриття.

24.6. Опції покриття

Додаткова інформація може бути описана за допомогою опцій. Вони можуть бути поміщені в групу. У цьому випадку опції будуть належати всім точкам покриття групи або будуть прив'язані до однієї точки для кращого керування процесом аналізу, що представлено нижче.

1) Завдання коментарів для спрощення аналізу:

```
covergroup CoverPort;
    option.comment = "Section 3.2.14 Port numbers";
    coverpoint port;
endgroup
```

2) Якщо testbench за замовчуванням створює копії груп покриття кілька разів, то System Verilog збирає значення покриття від всіх реалізацій групи. Але в проекті може бути кілька генераторів, кожний з яких створює різні потоки транзакцій, що вимагає додаткового аналізу окремих звітів. Наступна група може бути реалізована в різних генераторах, а інформація про покриття буде зберігатися для кожної копії:

```
covergroup CoverLength;
    coverpoint tr.length;
    option.per_instance = 1;
endgroup
```

3) Поріг покриття з використанням at_least (див. лістинг 24.6).

4) Друк порожніх кошиків. За замовчуванням у звіт про покриття виводяться тільки кошики, що містять елементи. Опція cross_num_print_missing пропонує програмі моделювання й інструментам звіту показувати всі кошики, особливо ті, які виявилися порожніми:

```
//Звіт про кошики, включаючи порожні
covergroup CovPort;
    kind: coverpoint tr.kind;
    port: coverpoint tr.port
    cross kind, port;
    option.cross_num_print_missing = 1_000;
endgroup
```

5) Мета покриття. Метою кожної групи або точки є рівень, на якому групи або точки розглядаються як повністю покриті. За замовчуванням - це 100% покриття. Якщо встановити рівень нижче 100%, то запит буде меншим повного покриття, що може бути недосяжним. Опція діє тільки для звіту покриття:

```
// Опис мети покриття
covergroup CoverPort;
    coverpoint port;
    option.goal = 90; // Установка для часткового покриття
endgroup
```

24.7. Параметризація групи покриття

24.7.1. Передача параметрів через значення.

У прикладі (лістинг 24.16) використовується значення, що ділить дані по двох кошиках. Значення формується при створенні групи за допомогою функції new.

Лістинг 24.16. Приклад параметра.

```
bit [2:0] port;           // Значення: 0:7
covergroup CoverPort (int mid);
    coverpoint port
        {bins lo = {[0: mid-1]};
         bins hi = {[mid:$]};
        }
endgroup
CoverPort cp;
initial
    cp = new(5);          // lo=0:4, hi=5:7
```

24.7.2. Передача параметрів через посилання.

Можна описати змінну значень, які спостерігаються за допомогою pass-by-reference (лістинг 24.17).

Лістинг 24.17. Передача параметрів по посиланню.

```
bit [2:0] port_a, port_b;
covergroup CoverPort (ref bit [2:0] port, int mid);
    coverpoint port {
        bins lo = {[0: mid-1]};
        bins hi = {[mid:$]};
    }
endgroup
CoverPort cpa, cpb;
initial begin
    cpa = new(port_a, 4); // port_a, lo=0:4, hi=5:7
    cpb = new(port_b, 2); // port_b, lo=0:1, hi=3:7
end
```

24.8. Контрольні запитання й завдання

1. Які види покриттів використовуються при верифікації й у чому їхня відмінність?
2. Які опції можуть бути використані в групах покриття?
3. Керуючий автомат містить 9 станів. Визначити тип даних State і змінну St, що описують стани автомата, а також групу покриття, що збирає значення St.
4. Створити клас, що містить 8-бітові поля a і b. Визначити в класі групу покриття для перевірки значень цих полів і метод new() для ініціалізації даної групи.

5. Створити клас, що містить 8-бітові поля *a* і *b*. Визначити в класі групи покриття для кожного поля й метод `new()` для їхньої ініціалізації. Перша група повинна збирати інформацію з переднього фронту `clk`, а друга - з заднього.

24.9. Література

1. Michael Nicolaidis, Yervant Zorian, Dhiraj K. Pradan. On Line-Testing for VLSI. – Kluwer Academic Publishers. – 1998. –168 p.
2. Samiha Mourad, Yervant Zorian. Principles of testing electronic systems.– Wiley, John & Sons, Incorporated.– 2000.– 444 p.
3. IEEE Std. 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog™ Hardware Description Language 1995. ISBN 0-7381-3065-6. IEEE Product No. WE94418-TBR.
4. IEEE Std 1364-2001. (Revision of IEEE Std 1364-1995).– IEEE Standard Verilog Hardware Description Language.– IEEE Computer Society Sponsored by the Design Automation Standards Committee.– Published by The Institute of Electrical and Electronics Engineers, Inc.– 3 Park Avenue, New York, NY 10016-5997, USA.– 2001.– 791 p.
5. IEEE Std 1800-2005.– IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language.– Sponsor: Design Automation Standards Committee of the IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group.– Published by The Institute of Electrical and Electronics Engineers, Inc.– 3 Park Avenue. New York, NY 10016-5997, USA.– 2005.– 664 p.
6. SystemVerilog 3.1a. Language Reference Manual. – Accellera's Extensions to Verilog®.– Accellera Organization, Inc. 1370 Trancas Street #163, Napa, CA 94558, USA.– 2004.
7. Spear C. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, 2nd ed.– Springer-Verlag, New York, Inc. Secaucus, NJ, USA.– 2006.– 429 p.
8. Mintz M., Ekendahl R. Hardware Verification with System Verilog. An Object-Oriented Framework.– New York : Springer.– 2007.– 314 p.
9. Bergeron J. Writing Testbenches using SystemVerilog.– Springer US.–2006.– 414 p.
10. Bergeron J., Cerny E., Hunter A., Nightingale A. Verification Methodology. Manual for SystemVerilog.– Springer US.–2005.– 510 p.
11. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas. 101 Common Coding Errors and How to Avoid Them.– New York : Springer.– 2007.– 214 p.
12. Vijayaraghavan Srikanth, Ramanathan Meyyappan. A Practical Guide for SystemVerilog Assertions.– Springer US.– 2005.– 334 p.
13. Haque F., Michelson J., Khan K. The Art of Verification with System Verilog Assertions.– Verification Central. First ed.– 2006.

ВИСНОВОК

Підручник являє собою підсумок еволюції суб'єктивних знань авторів про сучасні технології проектування, тестування й верифікації цифрових систем на кристалах на основі мов опису апаратури (Verilog) і архітектурних рішень (System Verilog).

Сутність викладеного матеріалу полягає в поданні мовних засобів, технологій, моделей, методів і процедур проектування й верифікації цифрових систем на кристалах, імплементованих у сучасні й компактні конструктиви. Більша частина запропонованого матеріалу є перероблені під точку зору авторів наукові й освітні видання провідних вчених і університетів планети, орієнтовані на підвищення технологічної культури студентської молоді, що вивчає курси по напрямку «Комп'ютерна інженерія».

Важливе місце в книзі займає досить свіжа технологія системної верифікації цифрових проектів, що йде від компаній Mentor Graphics, Synopsys, Cadence, Aldec і що використовує тестопридатний механізм асерцій, орієнтований на істотне зменшення часу виправлення помилок в HDL-кодi.

Ринкова привабливість видання визначається помітним ростом інтересу з боку IT-індустрії й університетів пострадянського простору до проблеми створення спеціалізованих енергозберігаючих компактних цифрових виробів, як альтернатива програмному продукту, для функціонування якого необхідна наявність універсального, дорогого й енерговитратного комп'ютера.

Зміст видання по суті або технологічно покриває традиційно усталені курси, пов'язані з автоматизацією проектування й цифровою технікою: «Мови опису апаратури», «Прикладна теорія цифрових автоматів», «Автоматизація проектування комп'ютерних систем», «Проектування систем на програмувальній логіці», «Проектування пристроїв цифрової обробки сигналів».

Апробація матеріалу, викладеного в книзі, на студентах факультету Комп'ютерної інженерії Харківського Національного Університету Радіоелектроніки дає підставу до видання даного посібника як підручника українською мовою, через відсутність аналогів у форматі з'єднання технологій з мовами опису апаратури й верифікації.

Суттєвий вплив на зміст книги і її поява зробили безсумнівні авторитети, колеги й друзі по цеху комп'ютерної інженерії: Володимир Тарасенко, Стенлі Гайдук, Раймунд Убар, Yervant Zorian, Paolo Prinetto, Дмитро Сперанський, Аркадій Закревський, В'ячеслав Харченко, Генадій Кривуля, Олексій Романкевич, Анжела Матросова, Юрій Скобцов, Олександр Дрозд, Олександр Баркалов. Виражаємо також подяку колегам і друзям по університету: Олександрі Петрівні Гужві, Анні Хахановій, Світлані Чумаченко, Ірині Побеженко, Василю Василенко, Юлії Хахановій, Сергію Зайченко, Дмитру Мельнику, Олексію Сушанову, які надали неоціненну допомогу в підготовці, оформленні й перевірці рукопису.

СПИСОК СКОРОЧЕНЬ

AC – Assertion Controller – контролер асерцій
 Ad-Hoc – спеціалізовані та проблемно-орієнтовані технології
 ADTG – Adder Test Generator – генератор тестів для суматора
 AMBA – Advanced Microcontroller Bus Architecture
 ASIC – Application-Specific Integrated Circuit
 ASSP – Application Specific Standard Product
 CAD – Computer-Aided Design – САПР
 CLB – Complex Logic Block
 CPLD – Complex PLD
 CPU – Central Processor Unit
 CTL – Core Test Language
 DFM – Design-for-Manufacturability
 DFT – Design-for-Testability
 DSP – Digital Signal Processing
 EDA – Electronic Design Automation
 ESL – Electronic System Level
 FDT – Fault Detection Table – таблиця несправностей
 F-IP – Functional Intellectual Property
 FPGA – Field Programmable Gate Array
 FS – Fault Simulator
 FV – Formal Verification – формальна верифікація
 GUI – Graphical User Interface
 HDL – Hardware Description Language – мова опису апаратури
 HDTV – High Definition Television
 HESTM – Hardware Embedded Simulator
 HFS – Hardware Fault Simulator
 HW-SW – Hardware-Software
 IEEE – Institute of Electrical and Electronics Engineers
 I-IP – Infrastructure Intellectual Property
 IP-core – Intellectual Property Core
 LUT – Look-up Table
 MCP – Multichip Package – багатокристальний пакет
 NoC – Network on Chip – мережа на кристалі
 PCI – Peripheral Connect Interconnect
 PiP – Package-in-Package
 PLD – Programmable Logic Device – програмований логічний пристрій
 PoP – Package-on-Package
 PRTG – Pseudo-Random Test Generator – генератор псевдовипадкових тестів
 RAM – Random Access Memory
 RTL – Register-Transfer Level

SAM – System Architectural Model
SiP – System in Package – система в пакеті кристалів
SoC – System on Chip – система на кристалі
STA – Static Timing Analysis
TAM – Test Access Mechanism
TAP – Test Access Port Controller
TLM – Transaction-Level Modeling
VHDL – Very high speed integrated circuits Hardware Description Language
VLSI – Very Large Circuit Integrated
BIC – велика інтегральна схема
ВПН – вектор підозрюваних несправностей
ВЭП – вектор експериментальної перевірки
ГРП – граф регістрових передач
ДІ – діагностична інформація
ДНФ – диз'юнктивна нормальна форма
ДС – деревовидна структура
КНФ – кон'юнктивна нормальна форма
КП – кубічне покриття
МП – мікропроцесор
БТН – багатозначна таблиця несправностей
ОЗП – оперативний запам'ятовуючий пристрій
ОКН – одиночна константна несправність
ПЛІС – програмована логічна інтегральна схема
ППЗП – постійний запам'ятовуючий пристрій
ПЕ – примітивний елемент
НВІС – надвелика інтегральна схема
ДДНФ – досконала диз'юнктивна нормальна форма
ТН – таблиця несправностей
ТФН – таблиця функцій несправностей
ЦАП – цифро-аналоговий перетворювач
ЦС – цифрова система
ЦП – цифровий пристрій

ДОДАТОК А. ДОВІДНИК РЛІ-ПІДПРОГРАМИ

А.1. Позначення

Таблиця А.1

Позначення	Значення
char *format	Передача форматowanego рядка
char*	Передача імені об'єкта у вигляді рядка
аргумент з підкресленням	Наявність аргументу не обов'язкова
*	Показчик на тип даних
...	Більше аргументів одного типу

А.2. Підпрограми доступу

А.2.1. Handle Routines.

Повертають абстрактні ідентифікатори або визначники об'єктів. Ім'я функцій даної групи завжди починається із приставки `acc_handle_`.

Таблиця А.2

Тип даних, що повертається	Ім'я	Список аргументів	Опис
handle	acc_handle_by_name	(char *name, handle scope)	Повертає визначник об'єкта, заданого іменем і областю видимості
handle	acc_handle_condition	(handle object)	Повертає визначник на вирази умови для шляху модуля або часової перевірки
handle	acc_handle_conn	(handle terminal)	Повертає визначник на лінію, підключену до примітиву, до шляху модуля або терміналу перевірки часових параметрів
handle	acc_handle_datapath	(handle modport)	Повертає визначник шляху даних

Продовження табл. А.2

Тип даних, що повертається	Ім'я	Список аргументів	Опис
handle	acc_handle_hiconn	(handle port)	Повертає ієрархічну старшу лінію, підключену до модуля порту
handle	acc_handle_interactive_scope	();	Показчик на поточну область видимості середовища моделювання
handle	acc_handle_loconn	(handle port)	Отримання ієрархічно молодшої лінії, підключеної до порту модуля
handle	acc_handle_modpath	(handle module, char *src, char *dest); или (handle module, handle src, handle dest);	Отримання визначника для шляху модуля, для якого описано джерело або приймач. Шлях модуля може бути описаний за іменем або за вказівниками
handle	acc_handle_notifier	(handle tchk);	Отримання оголошення регістрів, пов'язаних із зазначеною часовою перевіркою
handle	acc_handle_object	(handle object);	Отримання визначника будь-якого об'єкта, заданого через повне або відносне ієрархічне ім'я
handle	acc_handle_parent	(handle output, handle inport);	Отримання визначника батьківського об'єкта для примітиву, модуля або об'єкта
handle	acc_handle_path	(handle output, handle inport);	Отримання визначника для шляху між виходом одного і входом іншого модуля

Продовження табл. А.2

Тип даних, що повертається	Ім'я	Список аргументів	Опис
handle	acc_handle_pathout	(handle modpath);	Отримання визначника для першої лінії, підключеної до виходу модуля
handle	acc_handle_port	(handle module, int port#);	Повертає визначник порту модуля. Port# - це номер порту, починаючи з 0
handle	acc_handle_score	(handle object);	Повертає визначник на область видимості об'єкта
handle	acc_handle_simulated_net	(handle collapsed_net_handle);	Повертає визначник на лінію, пов'язану зі слабкою лінією
handle	acc_handle_tchk	(handle module, int tchk_type, char *netname1, int edge1, ...);	Повертає визначник для зазначеної часової перевірки модуля або комірки
handle	acc_handle_tchkarg2	(handle tchk);	Повертає лінію, пов'язану з другим аргументом часової перевірки
handle	acc_handle_terminal	(handle module, int terminal#);	Отримання визначника для терміналу примітиву. Terminal# - позиція в списку аргументів
handle	acc_handle_tfarg	(int arg#);	Отримання визначника для аргументу arg# системного завдання або функції, що викликає PLI-підпрограму
handle	acc_handle_tfinst	();	Отримання визначника для поточного, визначеного користувачем, системного завдання або функції

A.2.2. Next Routines.

Повертають визначник для наступного об'єкта у зв'язаному списку заданого типу. Підпрограми даного класу завжди починаються в `acc_next_` і приймають посилання на об'єкти як аргументи.

Таблиця А.3

handle	acc_next	(int obj_type_array[], handle module, handle current_object);	Повертає наступний об'єкт на деякий тип даних в заданій області видимості. Типи об'єктів, такі як accNet accRegister, визначені в obj_type_array
handle	acc_next_bit	(handle vector, handle current_bit);	Повертає наступний біт у векторі або масиві
handle	acc_next_cell	(handle module, handle current_cell);	Повертає копію наступної комірки в модулі. Комірки визначаються в бібліотеці
handle	acc_next_cell_load	(handle net, handle current_cell_load);	Повертає наступну комірку, завантажувану лінією
handle	acc_next_child	(handle module, handle current_child);	Повертає копію наступного модуля в структурній моделі
handle	acc_next_driver	(handle net, handle current_driver_termin al);	Отримання наступного терміналу примітиву, керуючого лінією
handle	acc_next_hiconn	(handle port, handle current_net);	Отримання наступного високого з'єднання лінії
handle	acc_next_input	(handle path_or_tchk, handle current_terminal);	Отримання наступного вхідного терміналу для описаного часового шляху або часової перевірки
handle	acc_next_load	(handle net, handle current_load);	Отримання наступного терміналу примітиву, керуючого лінією, незалежною від ієрархії
handle	acc_next_loconn	(handle port, handle current_net);	Получение следующего соединения низкой линии для портов модуля

Продовження табл. А.3

Тип даних, що повертається	Ім'я	Список аргументів	Опис
handle	acc_next_loconn	(handle port, handle current_net);	Отримання наступного з'єднання низької лінії для портів модуля
handle	acc_next_modpath	(handle module handle path);	Отримання наступного порту всередині модуля
handle	acc_next_net	(handle module, handle current_net);	Отримання наступної лінії в модулі
handle	acc_next_output	(handle path, handle current_terminal);	Отримання наступного вихідного терміналу шляху модуля або шляху даних
handle	acc_next_parameter	(handle module, handle current)	Отримання наступного параметра в модулі
handle	acc_next_port	(handle module, handle current_port);	Отримання наступного порту в списку портів модуля
handle	acc_next_portout	(handle module, handle current_port);	Отримання наступного порту output або inout в модулі
handle	acc_next_primitive	(handle module, handle)	Отримання наступного примітиву в модулі
handle	acc_next_scope	(handle module, handle current_scope);	Отримання наступної області видимості по відношенню до поточної
handle	acc_next_specparam	(handle module, handle current_specparam);	Отримання наступного спресрагата, декларованого в модулі
handle	acc_next_tchk	(handle module, handle current_tchk);	Отримання наступної часової перевірки в модулі
handle	acc_next_terminal	(handle primitive, handle)	Отримання наступного терміналу в примітиві
handle	acc_next_topmod	(handle current_topmod);	Отримання наступного вгору за рівнем модуля в проекті

A.2.3. Value Change Link (VCL) Routines

Підпрограма VCL дозволяє системним завданням додавати й видаляти об'єкти зі списку об'єктів для спостереження. Такі підпрограми завжди починаються із приставки `acc_vcl_` і не повертають значень.

Таблиця A.4

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	<code>acc_vcl_add</code>	(handle object, int (*consumer_routine)(), char *user_data, int VCL_flags);	Повідомляє Verilog-симулятор для виклику користувальницької підпрограми зі змінним значенням інформації при кожній зміні значення об'єкта
void	<code>acc_vcl_delete</code>	(handle object, int (*consumer_routine)(), char *user_data, int VCL_flags);	Видаляє вказаний об'єкт зі списку спостереження

A.2.4. Fetch Routines

Fetch-підпрограма дозволяє отримувати різну інформацію про об'єкти, таку як повне ієрархічне ім'я, відносне ім'я й інші атрибути. Підпрограми даної групи починаються із приставки `acc_fetch_`.

Таблиця A.5

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	<code>acc_fetch_argv</code>	();	Отримати кількість аргументів командного рядка виклику
char **	<code>acc_fetch_argc</code>	();	Отримати масив аргументів командного рядка виклику
double	<code>acc_fetch_attribute</code>	(handle object, char *attribute, double default);	Отримати атрибути параметра або <code>specparam</code>

Продовження табл. А.5

Тип даних, що повертається	Ім'я	Список аргументів	Опис
char **	acc_fetch_defname	(handle module);	Отримати ім'я визначення для копії модуля або примітиву
int	acc_fetch_delay_mode	(handle module);	Отримати режим затримки копії модуля
bool	acc_fetch_delays	(handle object, double *rise, double *fall, double *turnoff); (handle object, double *d1, *d2, *d3, *d4, *d5, *d6);	Отримати типові значення затримок для примітивів, шляхів модуля, часової перевірки або вхідних портів модуля
int	acc_fetch_direction	(handle object);	Отримати напрям порту або терміналу - input, output, inout
int	acc_fetch_edge	(handle path_or_tchk_ter m);	Отримати заданий тип фронту для шляху терміналу входу або виходу, або часової перевірки вхідного терміналу
char*	acc_fetch_fullname	(handle object);	Отримання повного ієрархічного імені для будь-якого іменного об'єкта або шляху модуля
int	acc_fetch_fulltype	(handle object);	Отримання типу даних об'єкта. Повертає зумовлені константи, що означають типи даних

Продовження табл. А.5

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	acc_fetch_index	(handle port or terminal);	Отримання індексу для порту або терміналу для вентиля, перемикача, копії UDP, модуля і т.п. Нуль повертається для першого терміналу
void	acc_fetch_location	(p_location loc_p, handle object);	Повертає розташування об'єкта у вихідному файлі Verilog. Зумовлений тип p_location є структурою, яка містить ім'я файла та номер лінії в файлі
char*	acc_fetch_name	(handle object);	Отримання копії об'єкта або шляху модуля в модулі
int	acc_fetch_paramtype	(handle parameter);	Отримання типу даних параметра, integer, string, real і т.д.
double	acc_fetch_paramvalue	(handle parameter);	Отримання значення параметра або спецparam. Повинна повертати значення integer, string або double
int	acc_fetch_polarity	(handle path);	Отримання полярності шляху
int	acc_fetch_precision	(handle path);	Отримати точність обчислення часу моделювання
bool	acc_fetch_pulse	(handle path, double *r1, double *e1, double *r2, double *e2, ...);	Отримати значення імпульсного контролю для шляху модуля, що складається зі значення відображення і значення для передачі
int	acc_fetch_range	(handle vector, int *msb, int *lsb);	Отримання найбільш і найменш значущих бітів значень діапазону для вектора

Продовження табл. А.5

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	acc_fetch_size	(handle object);	Отримання кількості бітів для ланцюга, регістра або порту
double	acc_fetch_tfarg	(int arg#);	Отримання значення аргументів для системного завдання або функції індексованого arg#
int	acc_fetch_tfarg_int	(int arg#);	Отримання строкового значення для аргументу системного завдання або функції індексованого arg#
char*	acc_fetch_tfarg_str	(int arg#);	Отримання значення integer для аргументу системного завдання або функції індексованого arg#
void	acc_fetch_timescale_info	(handle object, p_timescale_info timescale_p);	Отримання інформації про масштаб часу для об'єкта p_timescale_info, який є покажчиком на зумовлену структуру часової шкали даних
int	acc_fetch_type	(handle object);	Отримання типу об'єкта. Повертає зумовлену цілу константу, таку як accIntegerVar, accModule
char*	acc_fetch_type_str	(handle object);	Отримання типу об'єкта в строковому форматі. Повертає рядок accIntegerVar, accParameter
char*	acc_fetch_value	(handle object, char *format);	Отримання логічного значення або сили для ланцюга, регістра або змінної в описаному форматі

A.2.5. Utility Access Routines

Утиліти доступу виконують різноманітні завдання, що відносяться до доступу.

Таблиця А.6

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	acc_close	();	Звільняє внутрішню пам'ять, використовувану підпрограмами доступу, і скидає всі параметри конфігурації в значення, задані за замовчуванням
handle*	acc_collect	(handle *next_routine, handle ref_object, int *count);	Збирає всі об'єкти, які відносяться до певного посиляльного об'єкта, для успішного виклику підпрограми acc_next. Повертає масив визначників
bool	acc_compare_handles	(handle object1, handle object2);	Повертає значення істина, якщо обидва визначника відносяться до одного і того ж об'єкта
void	acc_configure	(int config_param, char *config_value);	Встановлює параметри, які контролюють операції для різних підпрограм доступу
int	acc_count	(handle *next_routine, handle ref_object);	Підраховує кількість об'єктів в посиляльному об'єкті, такому як модуль. Об'єкти підраховуються при успішному виклику підпрограми acc_next
void	acc_free	(handle *object_handles);	Звільняє пам'ять, виділену acc_collect, для збереження визначників об'єктів

Продовження табл. А.6

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	acc_initialize	();	Скидання параметрів конфігурації для всіх підпрограм доступу. Викликається на початку користувацької PLI-підпрограми
bool	acc_object_in_typelist	(handle object, int object type[]);	Шукає наявність зазначеної властивості або типу об'єкта в масиві властивостей або типів
bool	acc_object_in_type	(handle object, int object_type);	Порівнює властивість або тип об'єкта із заданою властивістю або типом
int	acc_product_type	();	Повертає тип використовуваного програмного продукту
char*	acc_product_version	();	Повертає версію використовуваного програмного продукту
int	acc_release_object	(handle object);	Перерозподіл пам'яті, пов'язаної зі шляхом входу і виходу терміналу
void	acc_reset_buffer	();	Скидання буфера рядка
handle	acc_set_interactive_scope	();	Установка інтерактивної області видимості для реалізації програми
void	acc_set_scope	(handle module, [char ^module_name]);	Установка області видимості для пошуку об'єктів в ієрархії проекту
char*	acc_version	();	Отримання версії використовуваної підпрограми доступу

A.2.6. Modify Routines

Таблиця А.7

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	acc_append_delays	(handle object, double rise, double fall, double z); или (handle object, double d1, ..., double d6); или (handle object, double limit);или (handle object double delay[]);	Додати затримку до існуючої затримки примітивів, шляхів модуля, часової перевірки або вхідних шляхів модуля. Можна задавати rise/fall/turn-off або 6 затримок, часову перевірку або формат min:typ:max
bool	acc_append_pulsere	(handle path, double r1, ..., double r12, double e1, ..., double e12);	Додати значення про шлях модуля до існуючого імпульсного контролю
void	acc_replace_delays	(handle object, double rise, double fall, double z); или (handle object, double d1, ..., double d6); или (handle object, double limit);или (handle object double delay[]);	Замінити існуючі затримки примітивів, шляхів модуля, часової перевірки або вхідних портів модуля. Можна задавати rise/fall/turn-off або 6 затримок, часову перевірку або формат min:typ:max
bool	acc_replays_pulsere	(handle path, double r1, ..., double r12, double e1, ..., double e12);	Встановлює значення імпульсного контролю для шляху модуля у відсотках від затримки шляху
void	acc_set_pulsere	(handle path, double reject, double e);	Встановлює відсотки імпульсного контролю для шляху модуля
void	acc_set_value	(handle object, p_setval_value value P,	Встановлює значення для регістру або послідовної UDP

A.3. Службові підпрограми (Utility (tf_) Routines)

Службові (tf_) підпрограми використовуються для передачі даних в обох напрямках через границю Verilog/Ci. Всі tf_ підпрограми припускають, що операції будуть виконуватися для поточних копій. Кожна tf-підпрограма має tf_i прототип, де показчик копії, у якій виконується операція, передається як додатковий аргумент у кінець списку аргументів.

A.3.1. Get Calling Task/Function Information

Таблиця A.8

Тип даних, що повертається	Ім'я	Список аргументів	Опис
char*	tf_getinstance	(<i>i</i>);	Отримання показчика на поточну копію завдання або функції моделювання, яка викликається користувальницькою PLI-програмою
char*	tf_ispname	(<i>i</i>);	Отримання імені ієрархічного шляху модуля, що містить виклик користувальницької PLI-програми
char*	tf_mipname	(<i>i</i>);	Отримання імені ієрархічного шляху області видимості, що містить виклик користувальницької PLI-програми

A.3.2. Get Argument List Information

Таблиця A.9

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	tf_nump	(<i>i</i>);	Отримати число параметрів у списку аргументів
int	tf_typep	(int param_index#);	Отримати тип зазначеного параметра в списку аргументів
int	tf_sizep	(int param_index#);	Отримати довжину параметрів у бітах
t_tfexprinfo*	tf_expinf	(int param_index#, struct t_tfexprinfo *expinf_p);	Отримати інформацію про вираження параметрів
t_tfexprinfo*	tf_nodeinfo	(int param_index#, struct t_tfexprinfo *expinf_p);	Отримати інформацію про значення параметрів вузла

A.3.3. Get Parameter Values

Таблиця А.10

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	tf_getp	(int param_index#);	Отримання значення параметра в цілочисельній формі
double	tf_getrealp	(int param_index#);	Отримання значення параметра в форматі з плаваючою точкою з подвійною точністю
int	tf_getlongp	(int *aof_highvalue, int para_index#);	Отримання значення параметра в 64-бітовій цілочисельній формі
char*	tf_strgetp	(int param_index#, char format_character);	Отримання значення параметра у вигляді Сі рядка символів
char*	tf_getcstringp	(int param_index#);	Отримання значення параметра у вигляді форматованого рядка
void	tf_evaluatep	(int param_index#);	Обчислення вираження параметрів і отримання результатів

A.3.4. Put Parameter Values

Таблиця А.11

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_putp	(int param_index#, int value);	Передача назад цілого значення для завдання або функції, що викликає
void	tf_putlongp	(int param_index#, double value);	Передача назад речовинного значення з подвійною точністю в форматі з плаваючою точкою для завдання або функції, що викликає
void	tf_putrealp	(int param_index#, int lowvalue, int highvalue);	Передача назад цілого значення з подвійною точністю 64 біта для завдання або функції, що викликає

Продовження табл. А.11

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_propagatep	(int param_index#);	Поширення значення параметра вузла
int	tf_strdelputp	(int param_index#, int bitlength, char format_char, int delay, int delaytype, char *value_p);	Передача назад значення та розкладу подій параметра. Значення представляється у вигляді форматowanego рядка і затримки, як цілого значення
int	tf_strealdeputp	(int param_index#, int bitlength, char format_char, int delay, double delaytype, char *value_p);	Передача назад значення рядка з доданою речовинною затримкою
int	tf_strlongdeputp	(int param_index#, int bitlength, char format_char, int lowdelay, int highdelay, double delaytype, char *value_p);	Передача назад значення рядка з доданою затримкою в форматі довгого речового числа

A.3.5. Monitor Parameter Value Change

Таблиця А.12

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_asynchon	();	Дозволяє користувальницькій PLI-підпрограмі бути викликаного при кожній зміні значення параметра
void	tf_asynchoff	();	Відключає асинхронний виклик
void	tf_synchronize	();	Синхронізує зміну параметрів з закінченням поточного часового слота моделювання
void	tf_rosynchronize	();	Синхронізує зміну параметрів і забороняє генерування нових подій протягом поточного часового слота моделювання
int	tf_getpchange	(int param_index#);	Отримує номер параметра, який змінив значення
int	tf_copypvc_flag	(int param_index#);	Копіює прапор зміни значення параметра
int	tf_movepvc_flag	(int param_index#);	Зберігає прапор зміни параметра
int	tf_testpvc_flag	(int param_index#);	Тестує прапор зміни значення параметра

A.3.6. Synchronize Tasks

Таблиця А.13

Тип даних, що повертається	Ім'я	Список аргументів	Опис
int	tf_gettime	();	Отримання поточного часу моделювання в цілочисельній формі
int	tf_getrealtime	();	Те ж саме
int	tf_getlongtime	(int *aof_hightime);	Отримання поточного часу моделювання у вигляді довгого цілого
char*	tf_strgettext	();	Отримання поточного часу моделювання у вигляді рядка
int	tf_getnextlongtime	(int *aof_lowtime, int *aof_hightime);	Отримання часу наступної події в списку
int	tf_setdelay	(int delay);	Приводить до реактивації користувальницького завдання в майбутній момент моделювання, представлений у вигляді цілочисельного значення затримки
int	tf_setlongdelay	(int lowdelay, int highdelay);	Приводить до реактивації користувальницького завдання після цілочисельної затримки, значення якої представлено у вигляді довгого цілого
int	tf_setrealdelay	(double delay, char *instance);	Активация miscf-додатку в зазначений час моделювання
void	tf_scale_longdelay	(char *instance, int lowdelay, int hidelay, int *aof_lowtime, int *aof_hightime);	Перетворює 64-бітову цілочисельну затримку у внутрішній масштаб представлення часу

Продовження табл. А.13

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_scale_realdelay	(char *instance, double delay, double *aof_realdelay);	Перетворює затримку в форматі з подвійною точністю і плаваючою точкою у внутрішній масштаб представлення часу
void	tf_unscale_longdelay	(char *instance, int lowdelay, int hidelay, int *aof_lowtime, int *aof_hightime);	Перетворює затримку з внутрішнього представлення в часову шкалу заданого модуля
void	tf_unscale_realdelay	(char *instance, double delay, double *aof_realdelay);	Перетворює затримку з внутрішнього представлення в часову шкалу заданого модуля
void	tf_clearalldelays	();	Очищення всіх затримок реактивації
int	tf_strdelputp	(int param_index#, int bitlength, char format_char, int delay, int delaytype, char *value_p);	Передача назад значень і розкладу подій для параметра. Значення представляється у вигляді форматowanego рядка символів, а затримка - цілочисельним значенням
int	tf_strrealdelputp	(int param_index#, int bitlength, char format_char, int delay, double delaytype, char *value_p);	Передача назад строкового значення з затримкою в речовинному форматі
int	tf_strlongdelputp	(int param_index#, int bitlength, char format_char, int lowdelay, int highdelay, int delaytype, char *value_p);	Передача назад строкового значення з затримкою в форматі довгого цілого

A.3.7. Long Arithmetic

Таблиця А.14

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_add_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Виконує додавання двох 64-бітових чисел
void	tf_subtract_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Виконує віднімання одного 64-бітового цілого числа від іншого
void	tf_multiply_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Виконує множення двох довгих цілих чисел
void	tf_divide_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Ділить одне довге число на інше
int	tf_compare_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Порівнює два довгих числа
char*	tf_longtime_tostr	(int lowtime, int hightime);	Перетворює довге ціле число в рядок
void	tf_real_to_long	(double real, int *aof_low, int *aof_high);	Перетворює дійсне число в 64-бітове ціле
void	tf_long_to_real	(int low, int high, double *aof_real);	Перетворює довге ціле число в речовинний формат

A.3.8. Display Messages

Таблиця А.15

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	io_print	(char *format, arg1, ...);	Виведення повідомлень в стандартне виведення і в log-файл

Продовження табл. A.15

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	io_mcdprint	(char *format, arg1, ...);	Виведення повідомлень в багатоканальний файл дескриптора
void	tf_error	(char *format, arg1, ...);	Виводить повідомлення про помилку
void	tf_warning	(char *format, arg1, ...);	Виводить повідомлення з попередженням
void	tf_message	(int level, char facility, char code, char *message);	Повідомляє про помилку і попереджає, використовуючи стандартні можливості поводження з помилками програми моделювання Verilog
void	tf_text	(char *format, arg1, ...);	Зберігає інформацію про повідомлення про помилку в буфері. Виводиться при виклику tf_message

A.3.9. Miscellaneous Utility Routines

Таблиця A.16

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_dostop	();	Зупиняє моделювання і переводить систему в інтерактивний режим
void	tf_dofinish	();	Завершує моделювання
char*	mc_scnplus	(char *startarg);	Повертає командну лінію плюс (+) опції, введені користувачем в інтерактивному режимі

Продовження табл. А.16

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_write_save	(char *blockptr, int blocklength);	Записати дані PLI-додатку в файл
int	tf_read_restart	(char *blockptr, int blocklength);	Отримати блок даних, раніше збережених у файлі
void	tf_read_restore	(char *blockptr, int blocklength);	Відновити дані, збережені у файлі
void	tf_dumpflush	();	Змінює значення параметра в системному dump-файлі
char*	tf_dumpfilename	();	Отримати ім'я системного dump-файла

A.3.10. Housekeeping Task

Таблиця А.17

Тип даних, що повертається	Ім'я	Список аргументів	Опис
void	tf_setworkarea	(char *workarea);	Зберегти покажчик на робочу область копії завдання або функції PLI-додатку
char*	tf_getworkarea	();	Відновити покажчик на робочу область
void	mc_setroutine	(char (*routine)());	Зберегти покажчик на завдання або функцію PLI-додатку
char*	tf_getroutine	();	Відновити покажчик на завдання або функцію PLI-додатку
void	tf_settflash	(char *tflash);	Зберегти покажчик на завдання або функцію PLI-додатку
char*	tf_gettflash	();	Відновити покажчик на копію завдання або функції PLI-додатку

ЗМІСТ

ВСТУП	3
ЧАСТИНА I. ПРОЕКТУВАННЯ	7
1. ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ ТА ВЕРИФІКАЦІЇ ЦИФРОВИХ СИСТЕМ	7
1.1. Апаратні засоби для розробки цифрових систем	7
1.2. Технологічні конструктиви цифрових систем	11
1.3. Технології виготовлення й тестування цифрових систем	20
1.4. Тестопридатне проектування SiP на основі стандарту IEEE 1500	24
1.5. Мови опису апаратури	30
1.6. Висновки за технологіями реалізації SoC	37
1.7. Тестування й верифікація HDL-моделей компонентів SoC	37
1.8. Тестопридатність програмно-апаратних продуктів	38
1.9. Інфраструктура процесу верифікації та тестування проекту	42
1.10. Аналітична модель інфраструктури верифікації	46
1.11. Аналіз тестопридатності HDL-проектів	53
1.12. Верифікація DCT IP-core, Xilinx	61
1.13. Висновки з верифікації HDL-коду на основі асерцій	65
1.14. Контрольні запитання та завдання	66
1.15. Література	67
ЧАСТИНА II. VERILOG	70
2. ОСНОВНІ ПОНЯТТЯ ТА КОНСТРУКЦІЇ МОВИ VERILOG	70
2.1. Модуль – базова одиниця мови Verilog	70
2.2. Рівні абстракції опису моделей на Verilog	71
2.3. Лексичні норми	74
2.4. Класи й типи даних	75
2.5. Опис портів модуля	79
2.6. Структурні моделі	80
2.7. Параметри	82
2.8. Директиви компілятора	84
2.9. Контрольні завдання	85
3. ВЕНТИЛЬНИЙ РІВЕНЬ ОПИСУ МОДЕЛЕЙ	87
3.1. Стандартні логічні елементи та моделі вентильного рівня	87
3.2. Моделі комбінаційних схем	88
3.3. Визначені користувачем примітиви	88
3.4. Затримки елементів	91
3.5. Контрольні завдання	93
4. DATAFLOW МОДЕЛІ ЦИФРОВОГО ПРИСТРОЮ	96
4.1. Dataflow моделі та оператор безперервного призначення	96
4.2. Оператори	96
4.3. Оператор умовного призначення	100
4.4. Системні завдання й функції для керування моделюванням	101
4.5. Контрольні завдання	104

5. ПОВЕДІНКОВІ МОДЕЛІ	107
5.1. Блоки Initial і Always	107
5.2. Процедурні оператори призначення	108
5.3. Часове й подієве керування операторами в блоках	109
5.4. Оператори керування: if, case, casez і casex. Цикли	114
5.5. Спеціальні властивості блоків	120
5.6. Процедурні оператори безперервного призначення	122
5.7. Приклади поведінкових моделей пристроїв	125
5.8. Контрольні завдання	129
6. ПІДПРОГРАМИ	134
6.1. Підпрограми Task Function	134
6.2. Автоматичні функції й завдання	138
6.3. Системні завдання роботи з файлами	139
6.4. Контрольні завдання	142
7. ГЕНЕРУВАННЯ ОПЕРАТОРІВ	144
7.1. Оператор generate і змінні genvar	144
7.2. Оператор generate із циклом for	145
7.3. Умовний оператор generate	146
7.4. Оператор generate з оператором case	147
7.5. Приклади використання оператора generate	148
7.6. Контрольні завдання	150
8. PROGRAMMING LANGUAGE INTERFACE (PLI)	152
8.1. Внутрішнє подання даних у програмі моделювання	155
8.2. Підпрограми бібліотеки PLI	157
8.3. Підпрограми доступу (access routine)	157
8.4. Обслуговуючі підпрограми (utility routine)	162
8.5. Приклади створення користувальницьких системних завдань	165
8.6. Контрольні завдання	166
8.7. Література	167
ЧАСТИНА III. СИНТЕЗ МОДЕЛЕЙ VERILOG	168
9. СИНТЕЗОВАНІ КОНСТРУКЦІЇ В СЕРЕДОВИЩІ VERILOG	169
9.1. Загальне поняття синтезу	169
9.2. Принципи побудови синтезованого коду	170
9.3. Синтезовані конструкції в Verilog	171
9.4. Контрольні питання й завдання	172
10. ІНТЕРПРЕТАЦІЯ ОПЕРАТОРІВ VERILOG	173
10.1. Типи даних	173
10.2. Використання векторів	174
10.3. Синтез комбінаційних схем	175
10.4. Синтез послідовних пристроїв	178
10.5. Особливі ситуації синтезу	180
10.6. Оператори циклів	188
10.7. Локальне використання змінних	190
10.8. Відмінність між блокуючими та неблокуючими змінними	191
10.9. Оператори Assign і Deassign	193
10.10. Синтез підпрограм	195

10.11. Використання параметрів	197
10.12. Використання значення x	198
10.13. Контрольні запитання	199
11. ШАБЛони СТАНДАРТНИХ КОМПОНЕНТІВ	202
11.1. Комбінаційні компоненти	202
11.2. Послідовна логіка	207
11.3. Автомати	214
11.4. Моделі пам'яті	220
11.5. Контрольні запитання й завдання	222
11.6. Література	225
Частина IV. МОДЕЛЮВАННЯ Й ВЕРИФІКАЦІЯ ЧАСОВИХ ХАРАКТЕРИСТИК ЦИФРОВИХ КОМПОНЕНТІВ	226
12. ОПИС ЧАСОВИХ ПАРАМЕТРІВ	226
12.1. Терміни	226
12.2. Аналіз часових характеристик цифрової схеми	228
12.3. Приклад обчислення часових параметрів	233
12.4. Імплементация проекту в силіконовий кристал	235
12.5. Контрольні запитання	238
13. МОДЕЛЮВАННЯ ЧАСОВИХ ПАРАМЕТРІВ ПРОЕКТУ В СЕРЕДОВИЩІ VERILOG	241
13.1. Моделі затримок у середовищі Verilog	241
13.2. Моделювання затримки шляху в середовищі Verilog	243
13.3. Перевірка часових параметрів	250
13.4. Системні завдання для визначення часових параметрів	251
13.5. Контрольні запитання й завдання	253
14. СИНТЕЗ І ВЕРИФІКАЦІЯ ПОСЛІДОВНОГО СУМАТОРА	255
14.1. RTL- модель пристрою	255
14.2. Синтез керуючого автомата	259
14.3. Синтез операційного пристрою в цілому	262
14.4. Оцінка часових параметрів	262
14.5. Контрольні запитання	263
14.6. Література	264
ЧАСТИНА V. SYSTEM VERILOG	265
15. ТИПИ ДАНИХ SYSTEM VERILOG	267
15.1. Простір для декларації об'єктів в SystemVerilog	267
15.2. Одиниці часу й точність їхнього обчислення	270
15.3. Нововведення в описі символьних значень	272
15.4. Логічні типи	273
15.5. Типи даних із двома станами	273
15.6. Рядки	275
15.7. Інші абстрактні типи даних SystemVerilog	276
15.8. Перетворення типів	277
15.9. Розмір виразів	279
15.10. Константи	279
15.11. Користувальницькі типи даних (User-defined types)	280
15.12. Спільно використовуване визначення типу typedef	281

15.13. Типи перелічення	282
15.14. Контрольні запитання	288
16. СКЛАДНІ ТИПИ ДАНИХ SYSTEMVERILOG: ARRAYS, STRUCTURES I UNIONS	289
16.1. Структури (Structures)	289
16.2. Об'єднання (Union)	292
16.3. Масиви фіксованого розміру	294
16.4. Динамічні масиви	303
16.5. Черги	304
16.6. Асоціативні масиви	306
16.7. Методи масивів	310
16.8. Контрольні запитання й завдання	312
17. ПРОЦЕДУРНІ БЛОКИ, ЗАВДАННЯ Й ФУНКЦІЇ	314
17.1. Процедурні блоки	314
17.2. Завдання, функції й void-функції	316
17.3. Процедурні оператори й підпрограми	319
17.4. Оператори переходу	324
17.5. Значення часових параметрів	328
17.6. Finite State Machines	328
17.7. System Verilog FIFO example	329
17.8. Контрольні запитання	334
18. System Verilog ієрархія проекту	335
18.1. Прототип модуля	335
18.2. З'єднання портів	337
18.3. Псевдоніми ланцюгів	338
18.4. Передача даних через порти модуля	340
18.5. Посилальні порти (Reference ports)	341
18.6. Параметризовані типи даних	341
18.7. Контрольні запитання й завдання	342
19. ВИКОРИСТАННЯ ІНТЕРФЕЙСІВ	344
19.1. Концепція інтерфейсу	344
19.2. Опис портів за допомогою інтерфейсу та звертання до них	351
19.3. Групування сигналів за допомогою modport	353
19.4. Використання завдань і функцій в інтерфейсі	356
19.5. Реконфігуровані інтерфейси	365
19.6. Компроміси інтерфейсу	367
19.7. Моделі пам'яті	368
19.8. Керування доступом до шини	371
19.9. Транзактор, адаптер і функціональні моделі шини	374
19.10. Контрольні запитання й завдання	385
20. ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ	387
20.1. Визначення класу в SystemVerilog	387
20.2. Об'єктно-орієнтоване програмування (ООП). Термінологія ООП	387
20.3. Створення нових об'єктів	388
20.4. Статичні й глобальні змінні	390
20.5. Підпрограми або методи класу	391

20.6. Правила границь видимості	392
20.7. Використання одного класу в межах іншого	394
20.8. Порядок компіляції	395
20.9. Поняття динамічного об'єкта	396
20.10. Зміна визначника в завданні	397
20.11. Зміна об'єктів «на льоту»	397
20.12. Масиви визначників	398
20.13. Копіювання об'єктів	398
20.14. Публічні або приватні об'єкти класів	401
20.15. Спадкування й підкласи	401
20.16. Приклади використання масивів класів	402
20.17. Контрольні завдання	404
20.18. Література	404
21. СЕРЕДОВИЩЕ ВЕРИФІКАЦІЇ	406
21.1. Методологія та інфраструктура верифікації SoC	406
21.2. Програми (Programs)	414
21.3. Кроки моделювання систем в SystemVerilog	415
21.4. Синхронізація тестових послідовностей (Stimulus Timing)	416
21.5. Приклад. Four-Port ATM Router	427
21.6. Контрольні запитання й завдання	431
22. АСЕРЦІЇ	432
22.1. Послідовності (Sequence)	432
22.2. Асерції в SystemVerilog	438
22.3. Прямі (процедурні) асерції	440
22.4. Паралельні асерції	443
22.5. Властивості (Properties)	445
22.6. Системні функції, використовувані в асерціях	448
22.7. Паралельні й прямі асерції	448
22.8. Взаємодія асерцій і testbench	450
22.9. Підтримка багатодоменої синхронізації	451
22.10. Контрольні завдання	455
22.11. Література	456
23. ПСЕВДОВИПАДКОВЕ ТЕСТУВАННЯ	458
23.1. Найпростіший клас коду із псевдовипадковими змінними	459
23.2. Використання масивів як границі безлічі псевдовипадкових значень	461
23.3. Зважений розподіл	462
23.4. Двонаправлені обмеження	463
23.5. Умовні обмеження	463
23.6. Використання арифметичних операторів	464
23.7. Керування декількома блоками обмежень	465
23.8. Обмеження для керування правильністю даних	465
23.9. Функції pre_randomize і post_randomize	466
23.10. Налаштовувані тести з випадковою генерацією	467
23.11. Використання невипадкових значень	468
23.12. Використання зовнішніх визначень обмежень	469

23.13. Проблеми рандомізації	469
23.14. Обмеження, ітеративні й масивів	470
23.15. Контрольні питання й завдання	471
24. ФУНКЦІОНАЛЬНЕ ПОКРИТТЯ	472
24.1. Покриття (Coverage)	472
24.2. Визначення моделі покриття: covergroup	473
24.3. Застосування групи покриття із класами	477
24.4. Збір даних	479
24.5. Перехресне покриття	484
24.6. Опції покриття	488
24.7. Параметризація групи покриття	489
24.8. Контрольні питання й завдання	489
24.9. Література	490
ВИСНОВОК	491
СПИСОК СКОРОЧЕНЬ	492
ДОДАТОК А. ДОВІДНИК PCL-ПІДПРОГРАМИ	494
ЗМІСТ	515

СХЕМОТЕХНІЧНЕ ПРОЕКТУВАННЯ І МОДЕЛЮВАННЯ НВІС (VLSI Schematic Design and Simulation)

ХАХАНОВ Володимир Іванович

ХАХАНОВА Ірина Віталіївна

ЛИТВИНОВА Євгенія Іванівна

ЛОБУР Михайло Васильович

ХАХАНОВА Ганна Володимирівна

Редактор *О.П. Гужва*

Дизайн-макетування *Є.І. Литвинова*

Художник *Е.Б. Лук'янченко*

Комп'ютерний набір та верстка ХНУРЕ

Підписано до друку 06.05.2016. Формат 60x84¹/₈.

Умов. друк. арк. 61,15. Зам. № 371. Тираж 500 прим. Ціна договірна.

ХНУРЕ, 61166, Харків, просп. Леніна, 14.

Надруковано у видавництві ЧП "Степанов В.В."

61168, Харків, вул. акад. Павлова, 311