О.Г. Аврунін
Т.В. Носова
І.В. Прасол
В.В. Семенець
Є.А. Чугуй

ОСНОВИ МОВ SYSTEMVERILOG ТА VHDL ДЛЯ ПРОЄКТУВАННЯ ЦИФРОВИХ ПРИСТРОЇВ НА ПЛІС У ПРИКЛАДАХ І ЗАДАЧАХ

Харків-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

О.Г. Аврунін, Т.В. Носова,
І.В. Прасол, В.В. Семенець, Є.А. Чугуй

ОСНОВИ МОВ *SYSTEMVERILOG* ТА *VHDL* ДЛЯ ПРОЄКТУВАННЯ ЦИФРОВИХ ПРИСТРОЇВ НА ПЛІС У ПРИКЛАДАХ І ЗАДАЧАХ

Навчальний посібник
Електронне видання

Харків-2025

УДК 004.03

Рекомендовано до друку рішенням Вченої ради
Харківського національного університету радіоелектроніки
(протокол № 15/7 від 31.10.2024)

Аврунін О.Г., Носова Т.В., Прасол І.В., Семенець В.В., Чугуй Є.А.
Основи мов *SystemVerilog* та *VHDL* для проєктування цифрових пристроїв на ПЛІС у прикладах і задачах: навч. посіб. / О.Г. Аврунін, Т.В. Носова, І.В. Прасол, В.В. Семенець, Є.А. Чугуй. – Електронне видання. – Харків: ХНУРЕ, 2025. – 383 с. – pdf 5,26 Mb.

ISBN 978-966-659-419-1

У навчальному посібнику розглянуто елементну базу ПЛІС з огляду на мови описання апаратури *SystemVerilog* та *VHDL*. Детально розглянуто синтаксис мов для програмування внутрішніх вузлів.

У кожному розділі наведено приклади програм керування та виконання базових функцій оброблення сигналів, що супроводжуються коментарями. З метою закріплення знань, набутих на практичних заняттях, подано умови задач для самостійної роботи.

Рекомендовано для здобувачів вищої освіти всіх рівнів і форм, які навчаються за галузями знань: 16 «Хімічна інженерія та біоінженерія», 17 «Електроніка та телекомунікації».

ISBN 978-966-659-419-1
DOI: 10.30837/978-966-659-419-1

© О.Г. Аврунін, Т.В. Носова, І.В. Прасол, В.В. Семенець, Є.А. Чугуй, 2025
© Харківський національний університет радіоелектроніки, 2025

ЗМІСТ

1 Від нуля до одиниці 8
1.1 Вступ 8
1.2 Мистецтво управління складністю 8
1.2.1 Абстракція 9
1.2.2 Конструкторська дисципліна 11
1.2.3 Три базові принципи 12
1.3 Цифрова абстракція 12
1.4 Системи числення 13
1.4.1 Десяткова система числення 13
1.4.2 Двійкова система числення 14
1.4.3 Шістнадцяткова система числення 16
1.4.4 Байт, напівбайт 18
1.4.5 Додавання двійкових чисел 19
1.4.6 Знак двійкових чисел 21
1.5 Логічні елементи 26
1.5.1 Логічний вентиль НІ 27
1.5.2 Буфер 27
1.5.3 Логічний вентиль І 28
1.5.4 Логічний вентиль АБО 29
1.5.5 Інші логічні елементи із двома вхідними сигналами 29
1.5.6 Логічні елементи з кількістю входів більше двох 31
1.6 За межею цифрової абстракції 32
1.6.1 Напруга живлення 32
1.6.2 Логічні рівні 33
1.6.3 Допустимі рівні шумів 33
1.6.4 Передавальна характеристика 34
1.6.5 Статична дисципліна 35
1.7 КМОН-транзистори 37
1.7.1 Напівпровідники 38
1.7.2 Діоди 39
1.7.3 Конденсатори 39
1.7.4 n-МОН- та p-МОН-транзистори 40
1.7.5 Логічний вентиль НІ на КМОН-транзисторах 43
1.7.6 Інші логічні вентилі на КМОН-транзисторах 44

1.7.7 Передавальний логічний вентиль 48
1.7.8 Псевдо n-МОН-логіка 48
1.8 Споживана потужність 50
1.9 Резюме 51
Вправи 53
Запитання для співбесіди 67

2 Проектування комбінаційної логіки 68
2.1 Вступ 68
2.2 Булеві рівняння 72
2.2.1 Термінологія 72
2.2.2 Диз'юнктивна форма 73
2.2.3 Кон'юнктивна форма 75
2.3 Булева алгебра 76
2.3.1 Аксіоми 77
2.3.2 Теореми однієї змінної 78
2.3.3 Теореми з кількома змінними 80
2.3.4 Правда про все це 82
2.3.5 Спрощення рівнянь 83
2.4 Від логіки до логічних елементів 85
2.5 Багаторівнева комбінаційна логіка 90
2.5.1 Мінімізація апаратури 90
2.5.2 Переміщення інверсії 92
2.6 Що за X та Z? 94
2.6.1 Неприпустиме значення: X 95
2.6.2 Третій стан: Z 95
2.7 Карти Карно 98
2.7.1 Думайте про овали 99
2.7.2 Логічна мінімізація на картах Карно 100
2.7.3 Байдужі змінні 105
2.7.4 Підбиваючи підсумки 106
2.8 Базові комбінаційні блоки 106
2.8.1 Мультиплексори 106
2.8.2 Дешифратори 112
2.9 Часові характеристики 113
2.9.1 Затримка поширення та затримка реакції 114
2.9.2 Імпульсні перешкоди 119

2.10 Резюме 123
Вправи 123
Запитання для співбесіди 131

3 Проєктування схем послідовної логіки 133
3.1 Вступ 133
3.2 Засувки та тригери 133
3.2.1 RS-тригер 135
3.2.2 D-засувка 138
3.2.3 D-тригер 139
3.2.4 Регістр 140
3.2.5 Тригер із функцією дозволу 141
3.2.6 Тригер із функцією скидання 141
3.2.7 Проєктування тригерів та засувок на транзисторному рівні 143
3.2.8 Загальний огляд 144
3.3 Проектування синхронних логічних схем 145
3.3.1 Деякі проблемні схеми 146
3.3.2 Синхронні послідовні схеми 148
3.3.3 Синхронні та асинхронні схеми 150
3.4 Кінцеві автомати 151
3.4.1 Приклад проєктування кінцевого автомата 152
3.4.2 Кодування станів 158
3.4.3 Автомати Мура та Мілі 161
3.4.4 Декомпозиція кінцевих автоматів 165
3.4.5 Відновлення кінцевих автоматів за електричною схемою 167
3.4.6 Огляд кінцевих автоматів 170
3.5 Синхронізація послідовних схем 171
3.5.1 Динамічна дисципліна 172
3.5.2 Часові властивості системи 173
3.5.3 Розфазування тактових сигналів 180
3.5.4 Метастабільність 184
3.5.5 Синхронізатор 186
3.5.6 Обчислення часу дозволу 188
3.6 Паралелізм 192
3.7 Резюме 196
Вправи 197
Запитання для співбесіди 207

4 Мови опису апаратури 209
4.1 Вступ 209
4.1.1 Модулі 209
4.1.2 Походження мов SystemVerilog та VHDL 211
4.1.3 Симуляція та Синтез 212
4.2 Комбінаційна логіка 215
4.2.1 Побітові оператори 215
4.2.2 Коментарі та прогалини 218
4.2.3 Оператори скорочення 219
4.2.4 Умовне присвоєння 220
4.2.5 Внутрішні змінні 224
4.2.6 Пріоритет 226
4.2.7 Числа 227
4.2.8 Стани Z та X 229
4.2.9 Маніпуляція бітами 232
4.2.10 Затримки 233
4.3 Структурне моделювання 235
4.4 Послідовна логіка 240
4.4.1 Регістри 240
4.4.2 Регістри зі скиданням 242
4.4.3 Регістри із сигналом дозволу 245
4.4.4 Групи регістрів 247
4.4.5 Засувки 248
4.5 І знову комбінаційна логіка 250
4.5.1 Оператори case 253
4.5.2 Оператори if 257
4.5.3 Таблиці істинності з невизначеними бітами 260
4.5.4 Блокувальні та неблокувальні присвоєння 262
4.6 Кінцеві автомати 267
4.7 Типи даних 275
4.7.1 SystemVerilog 275
4.7.2 VHDL 276
4.8 Параметризовані модулі 280
4.9 Середовище тестування 286
4.10 Резюме 295
Вправи 295
Запитання для співбесіди 310

5 Цифрові функціональні вузли ... 311
5.1 Вступ ... 311
5.2 Арифметичні схеми ... 311
5.2.1 Додавання ... 311
5.2.2 Віднімання ... 321
5.2.3 Компаратори ... 322
5.2.4 АЛП ... 324
5.2.5 Схеми зсуву та циклічного зсуву ... 326
5.2.6 Множення ... 328
5.2.7 Ділення ... 330
5.2.8 Додаткова література ... 331
5.3 Подання чисел ... 331
5.3.1 Числа з фіксованою точкою ... 332
5.3.2 Числа з рухомою точкою ... 333
5.4. Функціональні вузли послідовної логіки ... 337
5.4.1 Лічильники ... 338
5.4.2 Регістри зсуву ... 339
5.5. Матриці пам’яті ... 343
5.5.1 Огляд ... 344
5.5.2 Динамічне ОЗП (DRAM) ... 348
5.5.3 Статичний ОЗП (SRAM) ... 349
5.5.4 Площа та затримки ... 349
5.5.5 Регістрові файли ... 350
5.5.6 Постійний запам’ятовувальний пристрій ... 351
5.5.7 Реалізація логічних функцій із використанням матриць пам’яті ... 354
5.5.8 Мови опису апаратури та пам’яті ... 355
5.6 Матриці логічних елементів ... 358
5.6.1 Програмовані логічні матриці ... 358
5.6.2 Програмовані користувачем матриці логічних елементів ... 360
5.6.3 Схемотехніка матриць ... 366
5.7 Резюме. ... 367
Вправи ... 368
Запитання для співбесіди ... 380

Перелік джерел посилання ... 382

1 ВІД НУЛЯ ДО ОДИНИЦІ

1.1 Вступ

За останні тридцять років мікропроцесори буквально змінили світ до невпізнанності. Нині ноутбук визначається більшою вимірною потужністю, ніж комп'ютер нещодавнього минулого розміром з кімнату. Усередині сучасного автомобіля представницького класу можна знайти близько 50 мікропроцесорів. Саме прогрес у царині мікропроцесорної техніки уможливив появу мобільних телефонів та інтернету, сприяв значному розвитку медицини й радикально змінив тактику й стратегію сучасної війни. Обсяг продажів світової напівпровідникової промисловості зріс з 21 млрд доларів 1985 року до 300 млрд доларів 2011 року, зокрема мікропроцесори становили значну частку цих продажів. Не має сумніву, що мікропроцесори важливі не тільки з технічного, економічного та соціального погляду, але й стали одним із найбільш цікавих винаходів у історії людства.

У цьому посібнику основна увага приділяється розробленню цифрових систем, які застосовують для своєї роботи два рівні напруги, що подані одиницею та нулем. Почнемо з простіших цифрових логічних елементів – вентилів (цифрових логічних воріт), які приймають певну комбінацію одиниць і нулів на вході та перетворюють їх у комбінацію одиниць і нулів на виході. Після цього роглянемо, як об'єднувати ці простіші логічні елементи в більш складні модулі, такі як підсумовувачі та блоки пам'яті. Величезною перевагою цифрових систем над аналоговими є те, що необхідні для їх побудови блоки надзвичайно прості, оскільки працюють не з безперервними сигналами, а з одиницями та нулями [1].

Побудова цифрової системи не потребує складних математичних обчислень або глибоких знань у галузі фізики. Замість цього, завдання, що стоїть перед розробником цифрових пристроїв, полягає в тому, щоб зібрати з простих блоків складну систему, що працює.

1.2 Мистецтво управління складністю

Однією з властивостей, що розрізняють професійного інженера-електроніка або програміста, є систематичний підхід до управління складністю багаторівневої системи. Сучасні цифрові системи побудовані з мільйонів

та мільярдів транзисторів. Людський мозок не спроможний передбачити поведінку подібних систем за допомогою складання рівнянь, що описують рух кожного електрона в кожному транзисторі системи, та подальшого розв'язання цієї системи рівнянь. Для того, щоб розробити вдалий мікропроцесор і водночас не потонути в морі надлишкової інформації, необхідно навчитися керувати складністю системи, що розробляється.

1.2.1 Абстракція

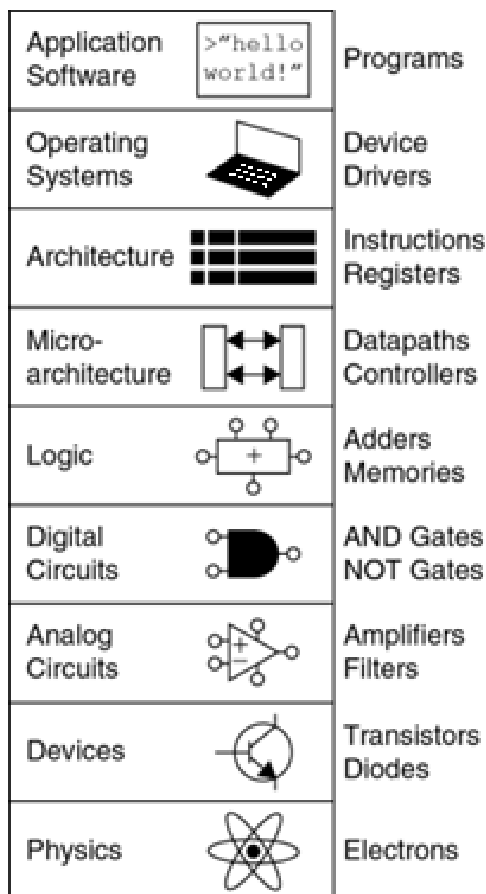


Рисунок 1.1 – Рівень абстракції електронної обчислювальної системи

Критично важливий принцип управління складністю системи – абстракція, що передбачає виняток із розгляду тих елементів, які в цьому разі не суттєві для розуміння роботи системи. Будь-яку систему можна аналізувати з різних рівнів абстракції.

На рис. 1.1 зображено рівні абстракції, типові для будь-якої електронної комп'ютерної системи разом із блоками, властивими для кожного рівня абстракції цієї системи. На найнижчому рівні абстракції розташована фізика, що вивчає рух електронів. Їх поведінку описує квантова механіка й система рівнянь Максвелла.

Сучасна електронна система містить напівпровідникові пристрої (*devices*), наприклад транзистори. Кожен пристрій має чітко визначені точки з'єднання з іншими подібними пристроями. Ці точки називатимемо контактами (в англомовній літературі використовується термін *terminal*). Будь-який електронний пристрій може бути поданий абстрактною математичною моделлю, яка описує взаємозалежність струму й напруги, що змінюється в часі. Такі самі зміни струму та напруги можна спостерігати на екрані осцилографа, якщо його під'єднати до контактів реального пристрою. Цей підхід означає, що за умови розгляду системи лише на рівні пристроїв, функції яких однозначно визначені, можна зважати на поведінку електронів усередині окремих пристроїв цієї системи.

Наступний рівень абстракції – це аналогові схеми (*analog circuits*), у яких напівпровідникові пристрої з'єднані так, щоб вони утворювали функціональні компоненти, зокрема підсилювачі. Напруга на вході та на виході такого ланцюга змінюється безперервно в деякому діапазоні.

На відміну від аналогових ланцюгів, цифрові схеми (*digital circuits*), такі як логічні вентилі, використовують два чітко обмежені дискретні рівні напруги. Один із цих рівнів – це логічний нуль, інший – логічна одиниця. У розділах цього посібника, присвячених розробленню цифрових схем і пристроїв, використовуватимемо найпростіші цифрові схеми для побудови складних цифрових модулів, зокрема суматорів та блоків пам'яті.

Мікроархітектурний рівень абстракції, чи навіть мікроархітектура (*microarchitecture*), пов'язує логічний і архітектурний рівні абстракції. Архітектурний рівень, або архітектура (*architecture*), описує комп'ютер з погляду програміста. Наприклад, архітектура *Intel* ×86, яку застосовують мікропроцесори більшості персональних комп'ютерів (ПК), визначається набором інструкцій та регістрів (пам'яті для тимчасового зберігання змінних), доступним для програміста з метою використання. Мікроархітектура – це поєднання найпростіших цифрових елементів у логічні блоки, призначені для виконання команд, визначених якоюсь конкретною архітектурою. Окремо взята архітектура може бути реалізована із застосуванням різних варіантів мікроархітектур із неоднаковим співвідношенням ціни, продуктивності та споживаної енергії, і таке співвідношення найчастіше обирається як баланс між цими трьома факторами. Процесори *Intel Core i*7, *Intel* 80486 і *AMD Athlon*, наприклад, використовують ту саму архітектуру *x*86, але реалізовану за допомогою трьох різних мікроархітектурних рішень.

Тепер перейдемо до сфери програмного забезпечення. Операційна система (*operating system*) керує операціями нижнього рівня, зокрема доступом до жорсткого диска або управлінням пам'яттю. І, нарешті, програмне забезпечення застосовує ресурси операційної системи для виконання конкретних завдань користувача.

1.2.2 Конструкторська дисципліна

Конструкторська дисципліна – це навмисне обмеження самим конструктором вибору можливих варіантів розробки, що дає змогу працювати більш продуктивно на вищому рівні абстракції. Використання взаємозамінних частин – це, мабуть, найбільш добре знайомий приклад практичного застосування конструкторської дисципліни. Одним із перших прикладів використання взаємозамінних деталей та вузлів стала уніфікація в процесі виробництва крем'яних рушниць. До початку XIX ст. такі рушниці виготовляли вручну. Висококваліфікований збройовий майстер ретельно підточував і підганяв деталі для комплектування, вироблені кількома не пов'язаними один з одним ремісниками. Конструкторська дисципліна для забезпечення взаємозамінності деталей та вузлів зробила революцію у збройовій промисловості. Обмеження асортименту деталей для комплектування до стандартного набору із жорстко встановленими допусками для кожної деталі дало змогу збирати та ремонтувати рушниці набагато швидше та водночас використовувати менш кваліфікований персонал. Збройовий майстер перестав витрачати свій час на розв'язання проблем, пов'язаних із нижніми рівнями абстракції, такими як доведення якогось конкретного ствола або виправлення форми окремого приклáда.

У контексті цього посібника дотримання конструкторської дисципліни у вигляді максимального застосування цифрових схем відіграє важливу роль. У цифрових схемах використовуються дискретні значення напруги, тоді як в аналогових схемах напруга змінюється безперервно. Отже, цифрові схеми, що можна розглядати як підмножину аналогових ланцюгів, у певному сенсі поступаються за своїми властивостями ширшому класу аналогових ланцюгів. Проте цифрові ланцюги набагато простіше проєктувати. Обмеживши використання аналогових схем і наскільки можна замінивши їх цифровими, можемо легко об'єднувати окремі компоненти в складні системи, що зрештою перевершать за своїми параметрами системи, побудовані на аналогових ланцюгах. Прикладом цього можуть бути цифрові телевізори,

компакт-диски (CD) й мобільні телефони, які вже практично витіснили своїх аналогових попередників.

1.2.3 Три базові принципи

Крім абстрагування від несуттєвих деталей і конструкторської дисципліни, розробники електронних систем застосовують ще три базових принципи для управління складністю системи: ієрархічність, модульність конструкції та регулярність. Зазначені принципи стосуються як програмного забезпечення, так і апаратної частини комп'ютерних систем.

● Ієрархічність – передбачає поділ системи на окремі модулі, а потім подальший поділ кожного модуля на фрагменти до рівня, що дає змогу легко зрозуміти поведінку всіх конкретних фрагментів.

● Модульність – вимагає, щоб кожен модуль у системі мав чітку функціональність і набір інтерфейсів та міг легко й без непередбачених побічних ефектів з'єднуватися з іншими модулями системи.

● Регулярність – потребує дотримання однаковості в проєктуванні окремих модулів системи. Стандартні модулі загального призначення, наприклад блоки живлення, можуть використовуватися багаторазово й водночас чимало разів знижувати кількість модулів, необхідних для розроблення нової системи.

1.3 Цифрова абстракція

Більшість фізичних величин змінюється безперервно. Наприклад, напруга в електричному дроті, частота коливань або розподіл маси – це параметри, що змінюються безперервно. Цифрові системи, з іншого боку, надають інформацію у вигляді змінних, що дискретно змінюються з кінцевим числом конкретних значень.

У разі двійкового коду висока напруга – це одиниця, а низька напруга – нуль, оскільки набагато легше оперувати двома рівнями напруги, ніж десятьма.

Обсяг інформації *D*, що передається однією дискретною змінною, яка може перебувати в різних станах, вимірюється в одиницях – бітах, і обчислюється за такою формулою:

$$\mathrm{D} = \log_2 N \text{ bits.} \tag{1.1}$$

Двійкова змінна передає один біт інформації. Тепер, мабуть, зрозуміло, чому одиниця інформації називається бітом.

Біт – це скорочення англійського словосполучення *binary digit*, що перекладається як двійковий розряд. Теоретично безперервний сигнал

може передавати нескінченну кількість інформації, оскільки здатний набувати необмежену кількість значень. Однак шум і помилки вимірювання обмежують інформацію, що передається більшістю безперервних сигналів діапазоном 10–16 бітів. Якщо ж рівень сигналу має вимірюватися дуже швидко, то обсяг передавальної інформації буде ще нижчим (за умови 10 бітів, наприклад, обсяг становитиме тільки 8 бітів).

Предметом цього навчального посібника є цифрові схеми, що використовують двійкові змінні – нуль та одиницю. Джордж Буль розробив систему логіки, що застосовує двійкові змінні, і цю систему нині називають його ім'ям – «булева логіка». Булеві змінні можуть набувати значення ІСТИНА (*TRUE*) або НЕПРАВДА (*FALSE*). В електронних комп'ютерах позитивна напруга зазвичай становить одиницю, а нульова – нуль. У цьому посібнику застосовуватимемо поняття одиниця (1) – ІСТИНА (*TRUE*) та ВИСОКИЙ (*HIGH*) як синоніми. Так само нуль (0) – НЕПРАВДА (*FALSE*) і НИЗЬКИЙ (*LOW*) для нас будуть взаємозамінні терміни.

1.4 Системи числення

Ми всі звикли працювати з десятковими числами. Однак у цифрових системах, побудованих на одиницях і нулях, використання двійкових або шістнадцяткових чисел найчастіше більш зручне. Нижче розглянемо системи числення, застосовані в цьому посібнику.

1.4.1 Десяткова система числення

Ще в початковій школі нас навчили рахувати та виконувати різні арифметичні операції в десятковій (*decimal*) системі числення. Ця система використовує десять арабських цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 – стільки ж, скільки пальців на руках. Числа понад 9 записуються у вигляді рядка цифр. До того ж цифра, розташована в кожній наступній позиції такого рядка, починаючи з крайньої правої цифри, має вагу, яка вдесятеро перевищує вагу цифри, що є в попередній позиції. Саме тому десяткову систему числення називають системою на основі (*base*) 10. Справа наліво вага кожної позиції збільшується таким чином: 1, 10, 100, 1000 і т.д. Позицію, що цифра посідає в рядку десяткового числа, називають розрядом чи декадою.

Щоб уникнути непорозумінь за умови одночасної роботи з більш ніж однією системою числення, основа системи зазвичай вказується способом додавання цифри за основне число й трохи нижче від нього: 9742_{10}.

На рис. 1.2 продемонстровано, наприклад, як десяткове число 9742_{10} може бути записано у вигляді суми цифр, що становлять це число, помножених на вагу розряду, що відповідає кожній конкретній цифрі.

1's column
10's column
100's column
1000's column

$$9742_{10} = 9 \times 10^3 + 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

nine thousands
seven hundreds
four tens
two ones

Рисунок 1.2 – Подання десяткового числа

N-розрядне десяткове число може бути одною з 10^N цифрових комбінацій: 0, 1, 2, 3, … $10^N - 1$. Це називається діапазоном *N*-розрядного числа. Десяткове число, що містить три цифри (розряди), наприклад, становить одну з 1000 можливих цифрових комбінацій у діапазоні від 0 до 999.

1.4.2 Двійкова система числення

Одиночний біт може приймати одне з двох значень – 0 або 1. Кілька бітів, з’єднаних в одному рядку, утворюють бінарне число. Кожна наступна позиція у двійковому рядку має удвічі більшу вагу, ніж попередня позиція, тож двійкова система числення – це система на основі 2. У двійковому числі вага кожної позиції збільшується (так само, як і в десятковому – справа наліво) таким чином: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 і т.д. У роботі з двійковими числами дуже корисно зберегти час і запам’ятати значення ступенів двійки до 256 [1].

Довільне *N*-розрядне двійкове число може бути одною з 2^N цифрових комбінацій: 0, 1, 2, 3, … $2^N - 1$. У табл. 1.1 зібрані 1-, 2-, 3- й 4-бітні двійкові числа та їх десяткові еквіваленти.

Таблиця 1.1 – Таблиця двійкових чисел та їх десяткових еквівалентів

1-бітні двійкові числа	**2-бітні двійкові числа**	**3-бітні двійкові числа**	**4-бітні двійкові числа**	**Десяткові еквіваленти**
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4

Продовження таблиці 1.1

1-бітні двійкові числа	**2-бітні двійкові числа**	**3-бітні двійкові числа**	**4-бітні двійкові числа**	**Десяткові еквіваленти**
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Приклад 1.1.

Перетворення чисел із двійкової системи числення в десяткову

Перетворіть двійкове число 101102 на десяткове.

Виконання. Необхідні перетворення подано на рис. 1.3.

16's column
8's column
4's column
2's column
1's column

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

one sixteen — no eight — one four — one two — no one

Рисунок 1.3 – Перетворення двійкового числа на десяткове

Приклад 1.2.

Перетворення чисел із десяткової системи числення у двійкову

Перетворіть десяткове число 84_{10} на двійкове.

Виконання. Визначте, що має стояти в кожній позиції двійкового результату: 1 або 0. Ви можете робити це, починаючи з лівої чи правої позиції.

Якщо почати ліворуч, знайдіть найбільший степінь 2, менший ніж задане число або рівний йому (у прикладі такий степінь – це 64). 84 > 64, тому ставимо 1 у позиції, що відповідає 64. Залишається 84–64 = 20, 20 < 32, тож у позиції 32 необхідно поставити 0, 20 > 16, тож у позиції 16 ставимо 1. Залишається 20–16 = 4. 4 < 8, тому 0 у позиції 8. 4 ≥ 4 – ставимо 1 у позицію 4.

4–4 = 0, тому будуть 0 у позиціях 2 та 1. Зібравши все разом, отримуємо 84_{10} =1010100_2.

Якщо почати справа, послідовно ділитимемо вихідне число на 2. Залишок іде в чергову позицію. 84/2 = 42, тому 0 – у правій позиції. 42/2 = 21, 0 – на другу позицію. 21/2 = 10, залишок 1 іде в позицію, що відповідає 4. 10/2 = 5, тому 0 – у позицію, що відповідає 8. 5/2 = 2, залишок 1 – у позицію 16. 2/2 = 1, 0 у – 32 позицію. Нарешті, 1/2 = 0 із залишком 1, який іде у позицію 64. Знову 84_{10} =1010100_2.

1.4.3 Шістнадцяткова система числення

Використання довгих двійкових чисел для запису й виконання математичних розрахунків на папері – спосіб не ефективний і може спричинити помилки. Однак довге двійкове число можна розбити на групи по чотири біти, кожна з яких становить одну з 2^4 = 16 цифрових комбінацій. Саме тому найчастіше зручно використовувати для роботи систему числення на основі 16, що називається шістнадцятковою (*hexadecimal*). Для запису шістнадцяткових чисел застосовуються цифри від 0 до 9 та літери від A до F, як показано в табл. 1.2. У шістнадцятковому числі вага кожної позиції змінюється так: 1, 16, 16^2 (або 256), 16^3 (або 4096) і т.д.

Таблиця 1.2 – Шістнадцяткова система числення

Шістнадцяткова цифра	**Десятковий еквівалент**	**Двійковий еквівалент**
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Приклад 1.3.

Перетворення шістнадцяткового числа у двійкове й десяткове

Перетворіть шістнадцяткове число $2ED_{16}$ у двійкове та десяткове.

Виконання. Перетворення шістнадцяткового числа в двійкове і назад – дуже просте, оскільки кожна шістнадцяткова цифра прямо відповідає чотирирозрядному числу $2_{16} = 0010_2$, $E_{16} = 1110_2$ та $D_{16} = 1101_2$, так що $2ED_{16} = 001011101101$. Перетворення в десяткову систему числення вимагає арифметики, продемонстрованої на рис. 1.4.

1's column
16's column
256's column

$$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$$

two two hundred fifty six's; fourteen sixteens; thirteen ones

Рисунок 1.4 – Перетворення шістнадцяткового числа на десяткове

Приклад 1.4.

Перетворення двійкового числа в шістнадцяткове

Перетворіть двійкове число 1111010_2 на шістнадцяткове.

Виконання. Повторимо ще раз: це просто. Починаємо справа. 4 найменш значних біти $1010_2 = A_{16}$. Наступні біти $111_2 = 7_{16}716$. Звідси $1111010_2 = 7_{16}$.

Приклад 1.5.

Перетворення десяткового числа в шістнадцяткове та двійкове

Перетворіть десяткове число 333_{10} у шістнадцяткове та двійкове.

Виконання. Як і в разі перетворення десяткового числа на двійкове, можна почати як зліва, так і справа.

Якщо почати зліва, знайдіть найбільший степінь шістнадцяти, менший від заданого числа або рівний йому (у нашій ситуації це $16^2 = 256$). Число 256 міститься в числі 333 лише один раз, тому в позицію з вагою 256 записуємо одиницю. Залишається число 333-256 = 77. Число 16 міститься в 77 чотири рази, тому в позицію з вагою 16 записуємо четвірку. Залишається 77 – 16 × 4 = 13. $13_{10} = D_{16}$, тому в позицію з вагою 1 записуємо цифру *D*. Отже, $333_{10} = 14D_{16}$, це число легко перетворити на двійкове, як ми показали в прикладі 1.3: $14D_{16} = 101001101_2$.

Якщо починати справа, повторюватимемо поділ на 16. Щоразу залишок іде в черговий стовпчик. 333/16 = 20 із залишком $13_{10} = D_{16}$, що йде в праву позицію. 20/16 = 1 із залишком 4, який іде в позицію з вагою 16. 1/16 = 0 із залишком 1, що йде в позицію з вагою 256. Зрештою знову отримуємо $14D_{16}$.

1.4.4 Байт, напівбайт

Група із восьми бітів називається «байт» (*byte*). Байт – це 2^8 = 256 цифрових комбінацій. Розмір модулів, збережених у пам'яті комп'ютера, зазвичай вимірюється саме в байтах, а не бітах.

Група з чотирьох бітів (половина байта) називається напівбайт (*nibble*). Напівбайт – це 2^4 = 16 цифрових комбінацій. Одна шістнадцяткова цифра займає один напівбайт, а дві шістнадцяткові цифри – один байт. Нині напівбайти широко не застосовуються, проте цей термін все ж таки варто знати, та й звучить він кумедно, адже в англійській мові *nibble* означає «відкушувати щось маленькими шматочками».

Мікропроцесор обробляє інформацію не повністю, а невеликими блоками, так званими словами. Розмір слова (*word*) перестав бути величиною, встановленою раз і назавжди, а визначається архітектурою кожного конкретного мікропроцесора. На момент написання цього розділу абсолютна більшість комп'ютерів використовувала 64-бітні процесори, вони обробляють інформацію блоками (словами) завдовжки 64 біти. А ще недавно межею досконалості були комп'ютери, що обробляють інформацію словами завдовжки 32 біти. Цікаво, що й нині найпростіші мікропроцесори, особливо ті, що керують роботою побутових пристроїв, зокрема тостерів або мікрохвильових печей, застосовують слова завдовжки 16 бітів або навіть 8 бітів.

У межах однієї групи бітів кінцевий біт, розташований на одному кінці цієї групи (зазвичай правим), називається найменш значущим бітом (*least significant bit,* LSB), або молодшим, а біт на іншому кінці групи – найбільш значущим бітом (*most significant bit msb*), або старшим. Рис. 1.5, *а* демонструє найменш значущі біти в разі шестибітного двійкового числа. Так само всередині одного слова можна виокремити найменш значущий байт (*least significant byte*, LSB), або молодший, і найбільш значущий байт (*most significant byte*, MSB), або старший. На рис. 1.5, *b* продемонстровано, який це має вигляд у разі чотирибайтного числа, записаного вісьмома шістнадцятковими цифрами.

Унаслідок удалого збігу 2^{10}= 1024 $\approx 10^3$. Цей факт дає змогу використовувати префікс «кіло-» (грецька назва тисячі) для скороченої позначки 2^{10}. Наприклад, 2^{10} байти – це один кілобайт (1 КБ). Так само «мега»

(грецька назва мільйона) позначає $2^{20} \approx 10^6$, а «гіга» (грецька назва мільярда) вказує на $2^{30} \approx 10^9$. Знаючи, що $2^{10} \approx 1$ тисяча, $2^{20} \approx 1$ мільйон, $2^{30} \approx 1$ мільярд і до 2^9 включно, буде легко приблизно подумки розрахувати будь-який інший степінь двох.

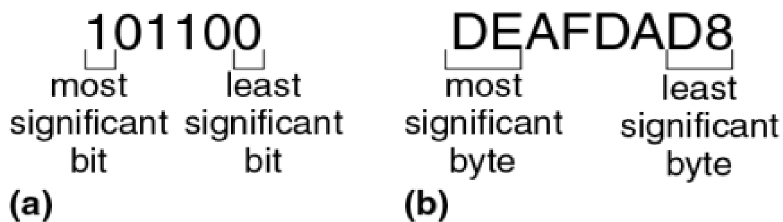


Рисунок 1.5 – Найменш важливі біти й байти

Приклад 1.6.

Оцінка степенів двійки

Знайдіть приблизне значення 2^{24} без використання калькулятора.

Виконання. Подайте експоненту як число, кратне десяти, і залишок.

$2^{24} = 2^{20} \times 2^4$, 220 ≈ 1 мільйон. $2^4 = 16$. Отже, $2^{24} \approx 16$ мільйонів. Насправді $2^{24} = 16777216$, але 16 мільйонів – досить гарне наближення для маркетингових цілей.

Так само як 1024 байти називають кілобайтом (КБ), 1024 біти – кілобітом (Кб або кбіт). Аналогічно МБ, Мб, ГБ та Гб використовуються для скороченої познаки мільйона та мільярда байтів та бітів. Розміри елементів пам’яті зазвичай вимірюються в байтах. А ось швидкість передачі інформації вимірюється в бітах за секунду. Максимальна швидкість передачі інформації телефонним модемом, наприклад, становить 56 кілобітів за секунду.

1.4.5 Додавання двійкових чисел

Двійкові числа додаються так само, як і десяткові, з тією лише різницею, що двійкове додавання виконати набагато простіше (див. рис. 1.6). Як і в разі додавання десяткових чисел, якщо сума двох чисел перевищує значення, що міститься в одному розряді, переносимо 1 у наступний розряд. На рис. 1.6 для порівняння показано додавання десяткових і двійкових чисел. У крайньому правому стовпці на рис. 1.6, *a* складаються числа 7 і 9. Сума 7 + 9 = 16, що перевищує 9, а отже, більше, ніж може вмістити один десятковий розряд. Тому записуємо в перший розряд 6 (перший стовпець), і переносимо 10 у наступний розряд (другий стовпець) як 1. Так само в процесі додавання двійкових чисел, якщо сума двох чисел перевищує 1, переносимо 2 в наступний розряд як 1 правому стовпцю на рис. 1.6, *b*, наприклад сума $1 + 1 = 2_{10} = 10_2$,

що не може вміститися в одному двійковому розряді. Тому записуємо 0 у першому розряді (перший стовпець) та 1 у наступному розряді (другий стовпець). У другому стовпці знову складаються 1 і 1 та додається 1, перенесена сюди після складання чисел у першому стовпці. Сума $1 + 1 + 1 = 3_{10} = 11_2$. Ми записуємо 1 до першого розряду (другий стовпець) і знову додаємо 1 до наступного розряду (третій стовпець). З очевидної причини біт, доданий до сусіднього розряду (стовпця), називається бітом перенесення (*carry bit*).

```
     11    ← carries →     11
   4277                  1011
 + 5499                + 0011
   9776                  1110
(a)                   (b)
```

Рисунок 1.6 – Приклади перенесення: а – десяткове; b – двійкове

Приклад 1.7.

Двійкове додавання

Обчисліть $0111_2 + 0101_2$.

Виконання. На рис. 1.7 показано, що сума дорівнює 1100_2. Перенесення виділено зверху. Можемо перевірити нашу роботу, повторивши обчислення в десятковій системі числення. $0111_2 = 7_{10}$, $0101_2 = 5_{10}$. Сума дорівнює $12_{10} = 1100_2$.

```
   111
   0111
 + 0101
 ------
   1100
```

Рисунок 1.7 – Приклад двійкового додавання

Цифрові системи зазвичай оперують числами із заздалегідь визначеною та фіксованою кількістю розрядів. Ситуацію, коли результат складання перевищує виділену йому кількість розрядів, називають переповненням (*overflow*). Чотирибітна комірка пам’яті, наприклад, може зберігати значення в діапазоні [0, 15]. Така комірка переповнюється, якщо результат додавання перевищує число 15. У цьому разі додатковий п’ятий біт відкидається, а результат, що залишився в чотирьох бітах, буде помилковим. Переповнення можна виявити, якщо стежити за перенесенням біта з найбільш значущого розряду двійкового числа (див. рис. 1.6), з найбільш крайнього стовпця ліворуч.

Приклад 1.8.

Додаток з переповненням

Обчисліть $1101_2 + 0101_2$. Чи буде переповнення?

Виконання. На рис. 1.8 показано, що сума дорівнює 10010_2. Результат виходить за межі чотирибітного двійкового числа. Якщо його потрібно запам'ятати в чотирьох бітах, найбільш значущий біт пропаде, залишивши некоректний результат – 0010_2. Якщо обчислення виконуються з числами з п'ятьма або більше бітами, результат буде коректним.

```
 11 1
  1101
+ 0101
------
 10010
```

Рисунок 1.8 – Приклад двійкового додавання з переповненням

1.4.6 Знак двійкових чисел

Досі ми розглядали двійкові числа без знака (*unsigned*) – тобто лише позитивні числа. Часто (проте для обчислень потрібні й позитивні, і негативні числа) це означає, що для знака двійкового числа нам знадобиться додатковий розряд. Існує кілька способів подання двійкових чисел зі знаком (*signed*). Найбільш широко застосовуються два: прямий код (*Sign/Magnitude*) та додатковий код (*Two's Complement*).

Прямий код

Подання негативних двійкових чисел із застосуванням прямого коду інтуїтивно здається найбільш привабливим, оскільки збігається зі звичним способом запису негативних чисел, коли спочатку йде мінус, а потім абсолютне значення числа. Двійкове число, що містить *N* бітів і записане в прямому коді, використовує найбільш значний біт для знака, інші *N*-1 бітів для запису абсолютного значення цього числа. Якщо найбільш значущий біт 0, число позитивне. Якщо найбільший біт 1, то число негативне.

Приклад 1.9.

Подання чисел у прямому коді

Запишіть 5 та –5 як чотирибітні числа в прямому коді.

Виконання. Обидва числа мають абсолютну величину $5_{10} = 101_2$. Отже, $5_{10} = 0101_2$ та $-5_{10} = 1101_2$.

На жаль, стандартний спосіб додавання не спрацьовує в разі двійкових чисел зі знаком, записаних у прямому коді. Двійкова змінна завдовжки N бітів у прямому коді може бути числом у діапазоні $[-2^{N-1}+1, 2^{N-1}-1]$.

Іншою, дещо дивною, особливістю прямого коду є наявність +0 і −0, до того ж обидва ці числа відповідають одному нулю. Неважко припустити, що подання однієї й тієї самої величини двома різними способами загрожує помилками.

Додатковий код

Двійкові числа, записані з використанням додаткового коду, та двійкові числа без знака ідентичні, за винятком того, що в разі додаткового коду вага найбільш значущого біта -2^{N-1} замість 2^{N-1}, як за умови двійкового числа без знака. Додатковий код гарантує однозначне подання нуля, допускає складання чисел за звичною схемою, а отже, позбавлений недоліків прямого коду.

У разі додаткового коду нульове значення подано нулями в усіх розрядах двійкового числа: $00..000_2$. Максимальне позитивне значення подано як нуль у найбільш значущому розряді та як одиниці в усіх інших розрядах двійкового числа: $01..111_2 = 2^{N-1} - 1$. Максимальне негативне значення має одиницю в найбільш значущому розряді та нулі – у всіх інших розрядах: $10..000_2 = -2^{N-1}$. Негативна одиниця подана одиницями в усіх розрядах двійкового числа: $11..111_2$.

Найбільш значущий розряд у всіх позитивних чисел – це 0, тоді як у негативних чисел – 1, тобто найбільш значущий біт додаткового коду можна розглядати як аналог знакового біта прямого коду. Однак на цьому подібність закінчується, оскільки інші біти додаткового коду інтерпретуються не так, як біти прямого коду.

У разі додаткового коду знак негативного двійкового числа змінюється на протилежний унаслідок виконання спеціальної операції, що називається доповненням до двох (*taking the two's complement*). Сутність цієї операції полягає в тому, що інвертуються всі біти цього числа й до значення найменш значущого біта додається 1. Подібна операція дає змогу знайти двійкове подання негативного числа або визначити його абсолютне значення.

Приклад 1.10.

Подання негативних чисел у додатковому коді

Знайдіть подання -2_{10} як чотирибітного числа в додатковому коді.

Виконання. Почніть з $+2_{10} = 0010_2$. Для отримання -2_{10} інвертуйте біти та додайте одиницю. Інвертуючи 0010_2, отримаємо 1101_2. 1101_2+1 = 1110_2.

Отже, – 2_{10} =1110_2.

Приклад 1.11.

Значення негативних чисел у додатковому коді

Знайдіть десяткове значення числа 1001_2 у додатковому коді.

Виконання. Число 1001_2 має старшу 1, тому воно має бути негативним. Щоб знайти його модуль, інвертуємо всі біти й додаємо 1. Інвертуючи 1001_2, отримаємо 0110_2. 0110_2 + 1 = 0111_2 = 7_{10}. Звідси 1001_2=-7_{10}.

Безперечною перевагою додаткового коду є те, що звичний спосіб додавання працює як у разі позитивних, так і негативних чисел. Нагадаємо, однак, що в процесі додавання *N*-бітних чисел *N*-й біт (тобто N + 1-й біт результату) не переноситься.

Приклад 1.12.

Додавання чисел, поданих у додатковому коді

Обчисліть (a) -2_{10} + 1_{10} та (b) -7_{10} + 7_{10} за допомогою чисел у додатковому коді.

Виконання.

(a) -2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}.

(b) -7_{10} + 7_{10} = 1001_2 +111_2=10000_2. П’ятий біт відкидається, залишаючи правильний чотирибітний результат 0000_2.

Віднімання одного двійкового числа від іншого здійснюється за допомогою перетворення числа, яке віднімається, в додатковий код і подальшого його складання зі зменшуваним.

Приклад 1.13.

Віднімання чисел у додатковому коді

Обчисліть (a) $5_{10} - 3_{10}$ та (b) $3_{10} - 5_{10}$, використовуючи чотирирозрядні числа в додатковому коді.

Виконання.

a) 3_{10} =0011_2. Обчислюючи його додатковий код, отримаємо -3_{10} =1101_2. Тепер складемо 0011_2 + 1101_2 = 0010_2 = 2_{10}. Зазначимо, що перенесення з найбільш значущої позиції скидається.

b) Обчислюючи додатковий код від 5_{10}, отримаємо -5_{10} = 1011_2. Тепер складемо 0011_2 +1011_2 = 1110_2 = -2_{10}.

Доповнення нуля до двох також здійснюється за допомогою інвертування всіх бітів (це дає $11..111_2$) і наступним додатком 1, що робить значення всіх бітів рівним 0. У цьому разі перенесення найбільш значущого біта ігнорується. Унаслідок нульове значення завжди подане набором тільки нульових бітів. На відміну від прямого коду, додатковий не має негативного нуля. Нуль завжди вважається позитивним числом, оскільки його знаковий біт завжди є 0.

Так само, як і двійкове число без знака, довільне *N*-бітне число, записане в додатковому коді, може набувати одного з 2^N можливих значень. Однак увесь цей діапазон розділено між позитивним і негативним числами. Наприклад, чотирибітне двійкове число без знака може набувати 16 значень від 0 до 15. У разі додаткового коду чотирибітне число також набуває 16 значень, але вже від –8 до 7. Загалом діапазон *N*-бітного числа, записаного в додатковому коді, охоплює $[-2^{N-1}, 2^{N-1}-1]$. Легко зрозуміти, чому в негативному діапазоні виявилося одне значення більше, ніж у позитивному – у додатковому коді відсутній негативний нуль. Максимальне від'ємне число, що можна записати з використанням додаткового коду $10..000_2 = -2^{N-1}$, іноді називають «дивним» числом (*weird number*). Щоб доповнити це число до двох, інвертуємо всі його біти (це дасть нам $01..111_2$), додамо 1 і отримаємо зрештою 10.000_2 – знову це саме «дивне» число. Тобто це єдине негативне число, яке не має позитивної пари.

У разі додаткового коду додавання двох позитивних чи негативних *N*-бітних чисел може призвести до переповнення, якщо результат буде більшим ніж 2^{N-1} – 1, або меншим ніж -2^{N-1}. Додавання позитивного й негативного числа, навпаки, ніколи не спричиняє переповнення. На відміну від двійкового числа, без знака перенесення найбільш значущого біта не є ознакою переповнення. Натомість індикатором переповнення є ситуація, коли після додавання двох чисел з однаковим знаком знаковий біт суми не збігається зі знаковими бітами доданків.

Приклад 1.14.

Додавання чисел у додатковому коді з переповненням

Обчисліть $4_{10} + 5_{10}$, використовуючи чотирибітні числа додаткового коду. Чи відбудеться переповнення?

Виконання. $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. Результат не вмістився в діапазон позитивних чотирибітних чисел у додатковому коді,

виявляючись негативним. Якби обчислення виконувалося з п’ятьма чи більше бітами, результат був би правильним.

У разі необхідності збільшення кількості бітів довільного числа, записаного в додатковому коді, значення знакового біта має бути скопійовано найбільш значущі розряди модифікованого числа. Ця операція називається знаковим розширенням (*sign extension*). Наприклад, числа 3 і −3 записуються в чотирибітному додатковому коді як 0011 та 1101 відповідно. Якщо збільшуємо число розрядів до семи бітів, маємо скопіювати знаковий біт у три найбільш значущі біти модифікованого числа, що дає 0000011 та 1111101.

Порівняння способів подання двійкових чисел

Три найбільш часто застосовуваних на практиці способи подання двійкових чисел – це двійкові числа без знака, прямий код і додатковий код. У табл. 1.3 наведено порівняння діапазону *N*-бітних чисел для кожного з перелічених способів. Переваги додаткового коду полягають у тому, що його можна використовувати для подання як позитивних, так і негативних цілих чисел, а звичний спосіб додавання працює для всіх чисел, поданих у додатковому коді. Віднімання здійснюється за допомогою перетворення числа, що віднімається в негативне число (тобто способом доповнення цього числа до двох) і подальшого додавання зі зменшуваним. Надалі в цьому посібнику, якщо не зазначено інше, передбачається, що всі двійкові числа подані в додатковому коді.

Таблиця 1.3 – Діапазон *N*-бітних чисел

Система	**Діапазон**
Двійкові числа без знака	$[0, 2^N - 1]$
Прямий код	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Додатковий код	$[-2^{N-1}, 2^{N-1} - 1]$

На рис. 1.9 зображена десяткова числова шкала з відповідними десятковими та чотирибітними двійковими числами, які подано трьома переліченими вище способами. Двійкові числа без знака перебувають у діапазоні [0, 15] і розташовуються у звичному порядку. Чотирибітні двійкові числа, подані в додатковому коді, містяться в діапазоні [–8, 7]. До того ж позитивні числа [0, 7] використовують таке саме кодування, як і двійкові без знака. Негативні числа [–8, –1] кодуються таким чином, що найбільше двійкове значення кожного такого числа без знака є числом, найближчим до 0.

Зауважте, що «дивне» число 1000 відповідає десятковому значенню −8 і не має позитивної пари. Числа, подані в прямому коді, містяться в діапазоні [–7, 7]. У цьому разі найбільш значущий біт є знаковим. Позитивні числа [0, 7] використовують таке саме кодування, як і двійкові числа без знака. Негативні числа симетричні щодо позитивних, з тією лише різницею, що їх знаковий біт має значення 1. Нуль подано двома значеннями – 0000 і 1000. Унаслідок того, що два числа відповідають одному нулю, будь-яке довільне N-розрядне двійкове число в прямому коді може бути подано тільки 2^N – 1 цілих числа.

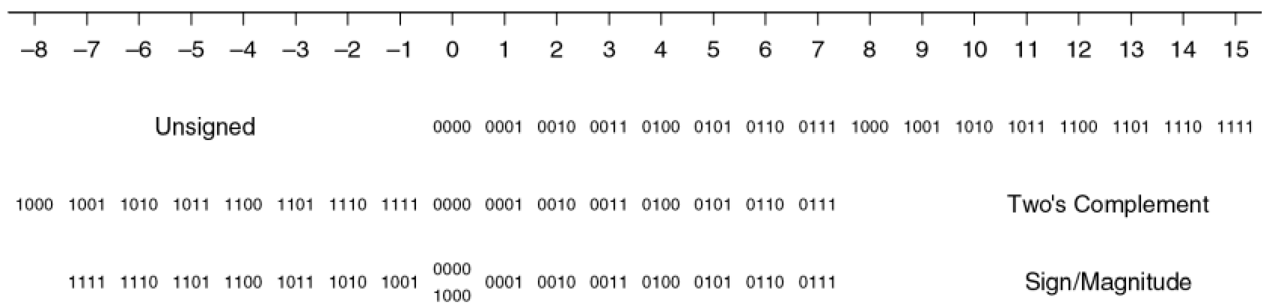


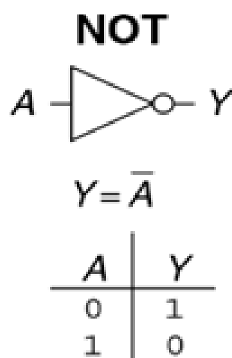
Рисунок 1.9 – Числова шкала та чотирибітне двійкове кодування

1.5 Логічні елементи

Тепер, коли ми знаємо, як використовувати бінарні змінні для подання інформації, розглянемо цифрові системи, що здатні виконувати різні операції з цими змінними. Логічні вентилі (*logic gates*) – це найпростіші цифрові схеми, що отримують один або більше двійкових сигналів на вході та виробляють новий двійковий сигнал на виході. За умови графічного зображення логічних вентилів для позначення одного чи кількох вхідних сигналів і вихідного сигналу використовуються спеціальні символи. Якщо дивитися на зображення логічного елемента, то вхідні сигнали зазвичай розміщуються ліворуч (чи вгорі), а вихідні сигнали – праворуч (чи внизу). Розробники цифрових систем зазвичай застосовують перші літери латинського алфавіту для позначення вхідних сигналів та латинську букву Y, щоб позначити вихідний сигнал. Взаємозв’язок між вхідними сигналами та вихідним сигналом логічного вентиля може бути описано за допомогою таблиці істинності (*truth table*) або рівнянням булевої логіки. Ліворуч таблиці істинності подані значення вхідних сигналів, а праворуч – значення відповідного вихідного сигналу. Кожен рядок у такій таблиці відповідає одній із можливих комбінацій вхідних сигналів. Рівняння булевої логіки – це математичний вираз, що описує логічний елемент за допомогою бінарних змінних.

1.5.1 Логічний вентиль НІ

Логічний вентиль НІ (*NOT gate*) має один вхід *A* і вихід *Y*, як зображено на рис. 1.10. У цьому разі вихідний сигнал *Y* – це сигнал, зворотний вхідному сигналу *A*, або, як кажуть, інвертований *A* (*inversed A*). Якщо сигнал на вході *A* – це НЕПРАВДА, сигнал на виході *Y* буде ІСТИНА. Таблиця істинності та рівняння булевої логіки на рис. 1.10 підсумовують цей зв'язок вхідного та вихідного сигналів. У рівнянні булевої логіки лінія над познакою сигналу читається як «ні», тобто математичний вираз $Y = \overline{A}$ вимовляється як «*Y* дорівнює не *A*». Саме тому логічний вентиль також називають інвертором (*inverter*).



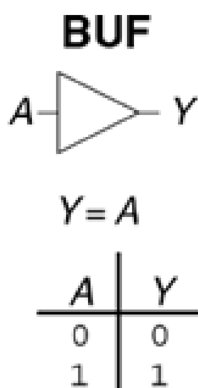
A	Y
0	1
1	0

Рисунок 1.10 – Вентиль НІ

Для позначення логічного вентиля НІ використовують інші способи запису, зокрема: $Y = A'$, $Y = \neg A$, $Y = !A$ *і* $Y = \sim A$. У цьому навчальному посібнику користуватимемося лише записом $Y = \overline{A}$, проте не дивуйтеся, якщо в науковій та технічній літературі ви зіткнетеся з іншими позначками.

1.5.2 Буфер

Іншим прикладом логічного вентиля з одним входом є буфер (*buffer*), зображений на рис. 1.11.



A	Y
0	0
1	1

Рисунок 1.11 – Буфер

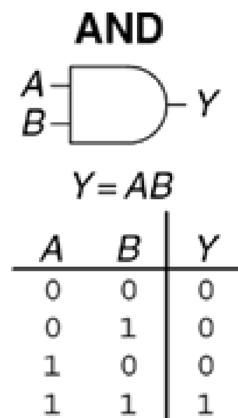
Буфер просто копіює вхідний сигнал на вихід. Якщо розглядати буфер як частину логічної схеми, такий елемент нічим не відрізняється від простого дроту й може здатися марним. Водночас на аналоговому рівні буфер може забезпечити властивості, необхідні для нормального функціонування пристрою, що розробляється.

Буфер, наприклад, необхідний для передачі великого струму електродвигуна чи швидкої передачі сигналу відразу кільком логічним елементам. Це ще один приклад, який доводить важливість розгляду будь-якої системи з кількох рівнів абстракції, якщо хочемо повною мірою зрозуміти цю систему. Розгляд буфера лише з позиції цифрового рівня абстракції не дає змогу встановити його реальну функцію.

У логічних схемах буфер позначається трикутником. Кружок на виході логічного елемента в англомовній літературі, часто названий міхуром (*bubble*), вказує на інверсію сигналу, як, наприклад, показано на рис. 1.10.

1.5.3 Логічний вентиль І

Логічні вентилі з двома вхідними сигналами набагато цікавіші, ніж вентиль НІ та буфер. Логічний вентиль (*AND gate*), продемонстрований на рис. 1.12, видає значення ІСТИНА на вихід *Y* тільки за умови, якщо обидва вхідні сигнали *A* і *B* мають значення ІСТИНА. В іншому разі вихідний сигнал *Y* має значення НЕПРАВДА.



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

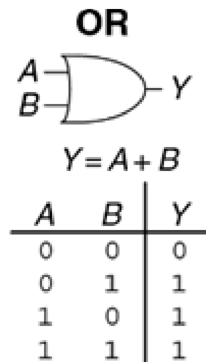
Рисунок 1.12 – Вентиль І

У випадку, що використовується, вхідні сигнали перелічені в порядку 00, 01, 10, 11, як у разі підрахунку у двійковій системі числення. Рівняння булевої логіки для логічного елемента І може бути записано декількома способами: $Y = A \bullet B$, $Y = AB$, або $Y = A \cap B$. Символ ∩ читається як «перетин» і більше за інших подобається фахівцям у математичній логіці. Однак у цьому посібнику вважаємо за краще застосовувати вираз $Y = AB$, що вимовляється як

«Y дорівнює A і B», лише тому, що ми досить ліниві, щоб обрати те, що коротше [1].

1.5.4 Логічний вентиль АБО

Логічний вентиль АБО (*OR gate*), зображений на рис. 1.13, видає значення ІСТИНА на вихід Y, якщо хоча б один із двох вхідних сигналів A або B має значення ІСТИНА. Рівняння булевої логіки для логічного елемента АБО записується як $Y = A + B$, або $Y = A \cup B$.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Рисунок 1.13 – Вентиль АБО

Символ ∪ читається як «об’єднання» та знову ж таки найбільше подобається математикам. Розробники цифрових систем зазвичай користуються простим символом +. Математичний вираз $Y = A + B$ вимовляється так: «Y дорівнює A або B».

1.5.5 Інші логічні елементи з двома вхідними сигналами

На рис. 1.14 зображені інші поширені логічні вентилі з двома вхідними сигналами. Додавання кружка на виході будь-якого логічного вентиля перетворює цей вентиль на протилежний йому, тобто інвертує його. Отже, наприклад, з вентиля І виходить вентиль І-НІ (*NAND gate*). Значення вихідного сигналу Y вентиля І-НІ буде ІСТИНА доти, доки обидва вхідні сигнали A і B не набудуть значення ІСТИНА. Так само з логічного вентиля АБО виходить вентиль АБО-НІ (*NOR gate*). Його вихідний сигнал Y буде ІСТИНА в тому разі, якщо жоден із вхідних сигналів (ні A, ні B) не має значення ІСТИНА. Вентиль, який виключає АБО і має кількість входів, рівну N (*N-input XOR gate*), іноді ще називають елементом контролю за парністю (*parity gate*). Такий вентиль видає на вихід сигнал ІСТИНА, якщо непарна кількість вхідних сигналів має значення ІСТИНА. Як і в разі елемента з двома вхідними сигналами, комбінації сигналів для елемента з входами N перелічені в логічній таблиці відповідно до підрахунку в двійковій системі числення.

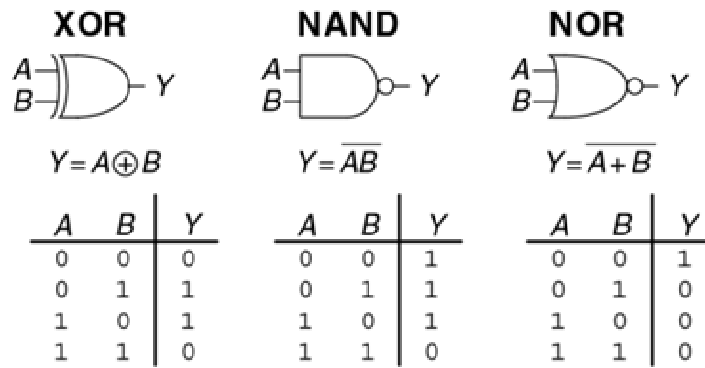


Рисунок 1.14 – Інші логічні елементи з двома вхідними сигналами

Приклад 1.15.

Вентиль, що виключає АБО-НІ

На рис. 1.15 продемонстровано познаки й булеве рівняння для вентиля, що виключає АБО-НІ (*XNOR*) з двома входами, який виконує інверсію, що виключає АБО. Заповніть таблицю істинності.

Виконання. На рис. 1.16 подано таблицю істинності. Вихід виключає АБО-НІ і є ІСТИНА, якщо обидва входи мають значення НЕПРАВДА або обидва входи мають значення ІСТИНА.

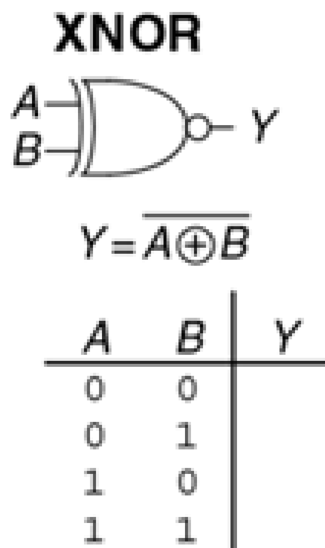


Рисунок 1.15 – Вентиль, що виключає АБО-НІ

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Рисунок 1.16 – Таблиця істинності вентиля, що виключає АБО-НІ

Вентиль що виключає АБО-НІ з двома входами, іноді називають вентилем рівності, оскільки його вихід є ІСТИНА, коли входи збігаються.

1.5.6 Логічні елементи з кількістю входів більше ніж два

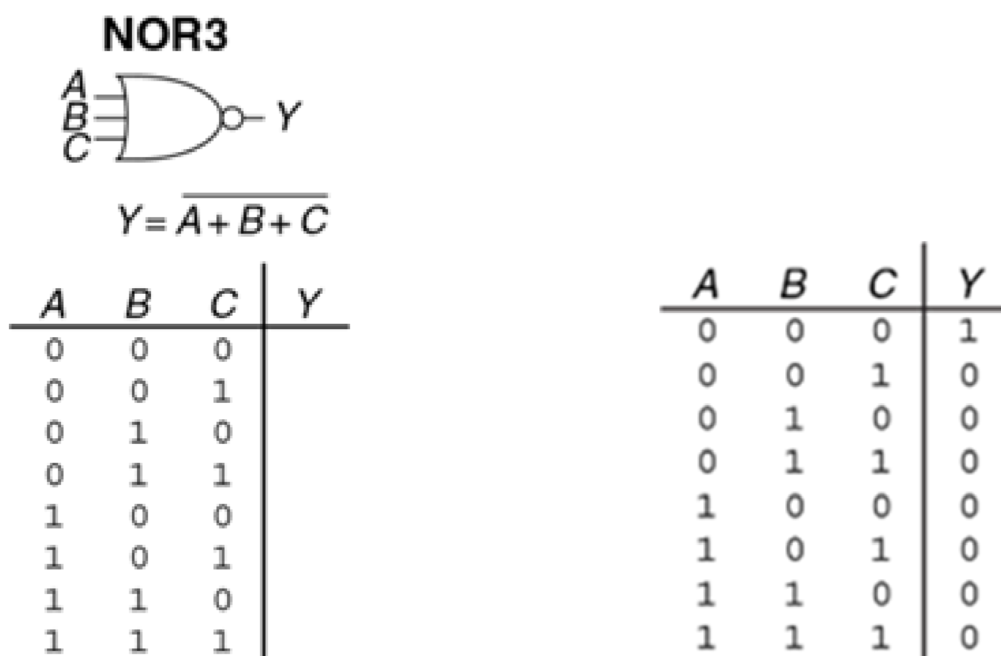
Чимало логічних функцій, а отже, і логічні вентилі, які необхідні для їх реалізації, оперують трьома й більше вхідними сигналами. Найпоширеніші з таких вентилів – це І, АБО, що виключає АБО, І-НІ, АБО-НІ і виключне АБО-НІ. Логічний вентиль І з кількістю входів, що дорівнює N, видає значення ІСТИНА, коли значення на всіх N входах цього логічного вентиля ІСТИНА. Логічний вентиль АБО з кількістю входів, що дорівнює N, видає ІСТИНА, коли значення хоча б одного з його входів ІСТИНА.

Приклад 1.16.

Вентиль АБО-НІ з трьома входами

На рис. 1.17 продемонстровано познаки та булеве рівняння для вентиля АБО-НІ з трьома входами. Заповнити таблицю істинності.

Виконання. На рис. 1.18 подано таблицю істинності. Вихід є ІСТИНА за умови, якщо немає жодного входу зі значенням ІСТИНА.



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Рисунок 1.17 – Вентиль АБО-НІ з трьома входами

Рисунок 1.18 – Таблиця істинності вентиля АБО-НІ з трьома входами

Приклад 1.17.

Вентиль із чотирма входами

На рис. 1.19 продемонстровано познаки та булеве рівняння для вентиля І з чотирма входами. Заповніть таблицю істинності.

Виконання. На рис. 1.20 подано таблицю істинності. Вихід є ІСТИНА за умови, якщо всі входи мають значення ІСТИНА.

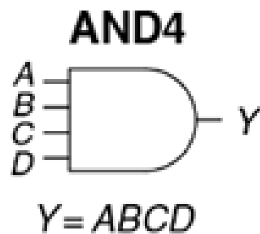


Рисунок 1.19 – Вентиль І з чотирма входами

A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рисунок 1.20 – Таблиця істинності вентиля І з чотирма входами

1.6 За межею цифрової абстракції

Цифрова система оперує дискретними змінними. Однак для їх подання використовуються безперервні фізичні величини, такі як напруга в електричному ланцюзі, положення шестерень у механічній передачі або рівень рідини в гідравлічному циліндрі. Завдання розробника цифрової системи – визначити, яким чином величина, що безперервно змінюється, співвідноситься з конкретним значенням дискретної змінної.

Розглянемо, наприклад, завдання подання двійкового сигналу A напругою в електричному ланцюзі. Припустимо, що напруга 0 відповідає значенню $A = 0$, а напруга 5 – $A = 1$. Однак реальна цифрова система має бути стійкою до неминучого в такій ситуації шуму, так що значення 4,97 В, імовірно, також необхідно тлумачити як $A = 1$. А що робити, якщо напруга дорівнює 4,3? Або 2,8? Або 2,500000?

1.6.1 Напруга живлення

Припустимо, що мінімальна напруга в електронній цифровій системі, яка називається також напругою землі (*ground voltage*, або просто *ground*, або *GND*), становить 0 В. Найвища напруга в системі надходить від блока живлення і, як правило, позначається V_{DD}.

Транзисторні технології 70–80-х рр. минулого століття здебільшого використовували V_{DD}, що дорівнює 5 В. З переходом на транзистори меншого розміру V_{DD} послідовно знижували до 3,3, 2,5, 1,8, 1,5, 1,2 В і навіть нижче для економії електроенергії та уникнення перевантаження транзисторів.

1.6.2 Логічні рівні

Безперевно змінювальна на різні значення дискретна двійкова змінна відтворюється способом визначення логічних рівнів (див. рис. 1.21). Перший логічний елемент у схемі називається «джерело» (*driver*), а другий – «приймач» (*receiver*). Вихідний сигнал джерела під’єднується до входу приймача. Джерело видає вихідний сигнал низької напруги (0) в діапазоні від 0 до V_{OL} або вихідний сигнал високої напруги (1) в діапазоні від V_{OH} до V_{DD}. Якщо приймач отримує на вхід сигнал у діапазоні від 0 до V_{IL}, він розглядає такий сигнал як нуль. Якщо приймач отримує на вхід сигнал у діапазоні від V_{IH} до V_{DD}, він розглядає такий сигнал як одиницю. Якщо ж з будь-якої причини, наприклад наявності шумів або несправності одного з елементів схеми, напруга сигналу на вході приймача падає настільки, що потрапляє в заборонену зону (*forbidden zone*) між V_{IL} і V_{IH}, то цей логічний елемент стає непередбачуваним. V_{OH} та V_{OL} називаються, відповідно, високим і низьким логічними рівнями виходу (*output high and low logic levels*), а V_{IH} та V_{IL} – високим та низьким логічними рівнями входу відповідно (*input high and low logic levels*).

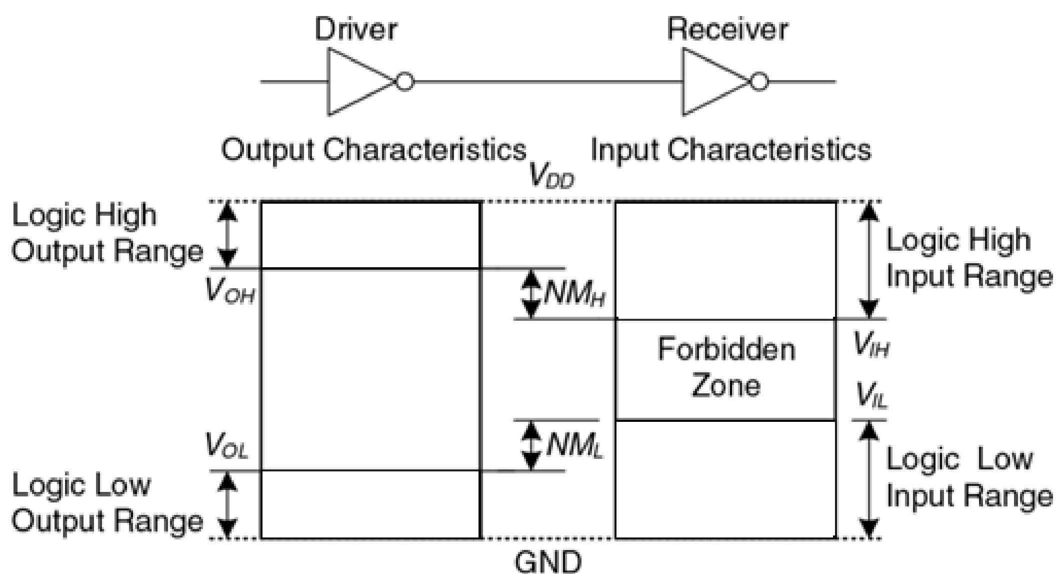


Рисунок 1.21 – Логічні рівні та рівні шуму

1.6.3 Допустимі рівні шумів

Щоб вихідний сигнал джерела було правильно інтерпретовано на вході приймача, необхідно, щоб $V_{OL} < V_{IL}$ і $V_{OH} > V_{IH}$. У цьому разі навіть якщо вихідний сигнал джерела буде забруднений шумами, приймач, як і раніше, зможе правильно визначити логічний рівень вхідного сигналу. Допустимий рівень шумів (*noise margin*) – це максимальна кількість шуму, присутність якого у вихідному сигналі джерела не заважає приймачу коректно

інтерпретувати значення отриманого сигналу. Як можна побачити на рис. 1.21, значення допустимого нижнього рівня шумів (*low noise margin*) і верхнього рівня шумів (*high noise margin*) визначають таким чином:

$$NM_L = V_{IL} – V_{OL}, \quad (1.2)$$

$$NM_H = V_{OL} – V_{IH}. \quad (1.3)$$

Приклад 1.18.

Розрахунок рівня шуму

Розглянемо схему з інверторами на рис. 1.22. V_{O1} – це напруга на виході інвертора I1, а V_{I2} – напруга на вході інвертора I2. Обидва інвертори мають такі властивості: V_{DD} = 5; V_{IL} = 1,35; V_{IH} = 3,15; V_{OL} = 0,33; V_{OH} = 3,84 В. Які нижній та верхній рівні шуму? Чи може схема коректно обробити рівень шуму 1 В між V_{O1} і V_{I2}?

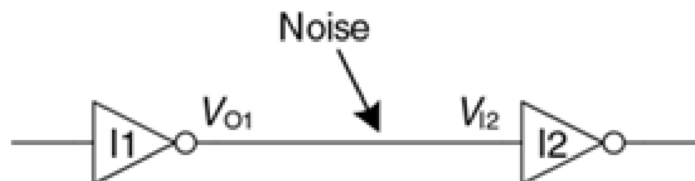


Рисунок 1.22 – Схема з інверторами

Виконання. Межі рівня шуму інвертора такі: NM_L = V_{IL} – V_{OL} = = (1,35 – 0,33 В) = 1,02 В; NM_H = V_{OH} – V_{IH} = (3,84 – 3,15 В) = 0 ,69 В. Схема може коректно обробити шум в 1 В, коли на виході низький рівень (NM_L = 1,02 В), але не коли на виході високий рівень (NM_H = 0,69 В). Наприклад, припустимо, що інвертор I1 має на виході в найгіршому разі ВИСОКЕ значення, V_{O1} = V_{OH} = 3,84 В. Якщо наявність шуму спричинить падіння напруги на 1 В на вході інвертора I2, тоді V_{I2} = (3,84 – 1 В) = 2,84 В. Це менше, ніж допустиме вхідне значення ВИСОКОГО рівня V_{IH} = 3,15 В, тому інвертор I2 може не набути правильного вхідного значення ВИСОКОГО рівня.

1.6.4 Передавальна властивість

Для розуміння межі цифрової абстракції необхідно розглянути поведінку логічних вентилів з аналогового аспекта. Передавальна властивість (*D transfer characteristics*) будь-якого логічного вентиля описує напругу на виході цього елемента як функцію напруги на його вході, коли вхідний сигнал змінюється настільки повільно, що вихідний сигнал встигає змінюватися слідом за ним. Така властивість називається передавальною, оскільки описує взаємозв’язок між вхідною та вихідною напругою.

За умови ідеального інвертора перемикання буде різким у точці $V_{DD}/2$, як зображено на рис. 1.23, *a*. Для $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. Для $V(A) > V_{DD}/2$, $V(Y) = 0$. У цьому разі $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ та $V_{OL} = 0$.

Напруга під час перемикання реального інвертора змінюється поступово між граничними значеннями (див. рис. 1.23, *b*). Якщо вхідна напруга $V(A)$ дорівнює 0, то напруга на виході $V(Y) = V_{DD}$. Якщо $V(A) = V_{DD}$, то $V(Y) = 0$. Однак перехід між цими кінцевими точками плавний і може розташовуватися більшою мірою праворуч або ліворуч від значення $V_{DD}/2$. У зв'язку з цим виникає закономірне запитання: як у цьому разі визначити логічні рівні?

Розумно обрати логічними рівнями ті дві точки, де нахил передавальної властивості $dV(Y)/dV(A)$ дорівнює −1. Такі точки називаються «граничні коефіцієнти передачі» (*unity gain points*). Подібний вибір зазвичай максимізує допустимі рівні шумів. Якщо V_{IL} зменшується, V_{OH} незначно збільшується. Однак, якщо V_{IL} зростає, V_{OH} падає практично прямовисно.

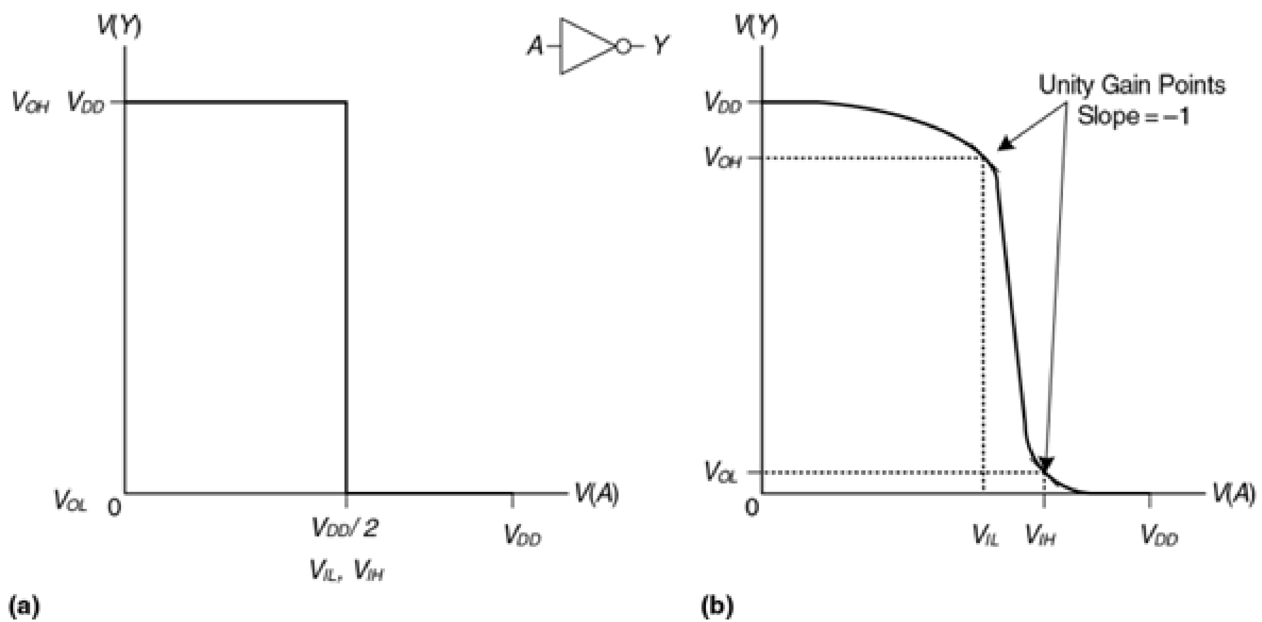


Рисунок 1.23 – Передавальні властивості та рівні шуму

1.6.5 Статична дисципліна

Щоб уникнути потрапляння вхідних сигналів у заборонені зони, логічні вентилі необхідно розробляти відповідно до принципу статичної дисципліни (*static discipline*). Цей принцип вимагає, щоб за наявності логічно коректних сигналів на вході кожен елемент системи видавав логічно коректні сигнали на виході.

Застосування принципу статичної дисципліни обмежує свободу розробника у виборі аналогових елементів для побудови цифрових систем, проте допомагає

забезпечити простоту й надійність цифрових схем, що розробляються. Використовуючи цей принцип, розробник піднімається з аналогового рівня абстракції на цифровий, що збільшує продуктивність проєктувальника й позбавляє його розгляду зайвих деталей.

Вибір V_{DD} та логічних рівнів може бути довільним, однак цей вибір має забезпечити сумісність усіх логічних вентилів, що обмінюються інформацією в межах однієї цифрової системи. Тому вентилі зазвичай групуються в сімейства логіки (*logic families*) таким чином, що будь-який елемент з одного сімейства внаслідок з'єднання з будь-яким іншим елементом цього самого сімейства автоматично забезпечує дотримання принципу статичної дисципліни. Логічні вентилі одного сімейства з'єднуються один з одним так само легко, як і блоки конструктора Лего, оскільки вони повністю сумісні за напругою джерела живлення та логічними рівнями.

Чотири основні сімейства логічних вентилів домінували з 70-х до 90-х рр. минулого століття – це ТТЛ – транзисторно-транзисторна логіка (*Transistor-Transistor Logic*, або TTL), КМОН – логіка, побудована на комплементарній структурі метал-оксид-напівпровідник (*Complementary Metal-Oxide-Semiconductor Logic*, або CMOS), НТТЛ – низьковольтна транзисторно-транзисторна логіка (*Low-Voltage Transistor-Transistor Logic*, або LVTTL) і НКМОН – низьковольтна логіка на комплементарній структурі метал-оксид-напівпровідник (*Low-Voltage Complementary Metal-Oxide-Semiconductor Logic*, або LVCMOS). Логічні рівні цих сімейств подано в табл. 1.4. Починаючи з 90-х рр. минулого століття, чотири перелічені вище сімейства розпалися на значну кількість дрібніших сімейств у зв'язку з усе більшим поширенням пристроїв, що вимагають значно нижчої напруги живлення.

Таблиця 1.4 – Сімейства логіки з рівнями напруги 5 і 3,3 В

Сімейство логіки	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4,75−5,25)	0,8	2,0	0,4	2,4
CMOS	5 (4,5−6)	1,35	3,15	0,33	3,84
LVTTL	3,3 (3−3,6)	0,8	2,0	0,4	2,4
LVCMOS	3,3 (3−3,6)	0,9	1,8	0,36	2,7

Приклад 1.19.

Сумісність логічних сімейств

Які з логічних сімейств з табл. 1.4 можуть надійно взаємодіяти між собою?

Виконання. У табл. 1.5 перелічено логічні сімейства, що мають сумісні логічні рівні. Зауважимо, що п'ятивольтові логічні сімейства, зокрема TTL

і CMOS, можуть видавати на вихід ВИСОКИЙ рівень 5 В. Якщо п'ятивольтовий сигнал подається на вхід сімейству з рівнем 3,3 В, такому як LVTTL або LVCMOS, це може пошкодити приймач, якщо у специфікації останнього не зазначено чітко «5 В-сумісний».

Таблиця 1.5 – Сумісність логічних сімейств

		Наслідувач			
		TTL	**CMOS**	**LVTTL**	**LVCMOS**
Джерело	TTL	ТАК	НІ: $V_{OH} < V_{IH}$	МОЖЛИВО[a]	МОЖЛИВО[a]
	CMOS	ТАК	ТАК	МОЖЛИВО[a]	МОЖЛИВО[a]
	LVTTL	ТАК	НІ: $V_{OH} < V_{IH}$	ТАК	ТАК
	LVCMOS	ТАК	НІ: $V_{OH} < V_{IH}$	ТАК	ТАК

1.7 КМОН-транзистори*

*Цей розділ, як і інші, позначені знаком *,*
є додатковим і не обов'язковим
для засвоєння матеріалу
цього навчального посібника.

Аналітична Машина Беббіджа була механічним пристроєм із пружинами та шестернями, а в перших комп'ютерах використовувалися реле або вакуумні трубки. У сучасних комп'ютерах застосовують транзистори, оскільки вони дешеві, мають незначні розміри та високу надійність. Транзистор – це перемикач із двома положеннями «увімкнути» й «вимкнути», контрольований способом подання напруги або струму на керуючу клему. Існують два основних типи транзисторів: біполярні (*bipolar junction transistors*) і МОН-транзистори – метал-оксид-напівпровідник-транзистори (іноді їх називають польовими; *metal-oxide-semiconductor field effect transistors*, або MOSFET).

У 1958 р. Джек Кілбі з *Texas Instruments* створив першу інтегральну схему з двох транзисторів. У 1959 р. Роберт Нойс, який працював тоді у *Fairchild Semiconductor*, запатентував метод з'єднання кількох транзисторів на одному кремнієвому кристалі. Тоді один транзистор коштував близько 10 американських доларів.

Роберт Нойс народився в місті Берлінгтон штату Айова й здобув ступінь бакалавра в галузі фізики в Ґріннелльському коледжі, а ступінь доктора наук у галузі фізики – у Массачусетському технологічному інституті. Роберту Нойсу дали прізвисько «мер Силіконової долини» за його неабиякий внесок у розвиток мікроелектроніки.

Нойс став співзасновником *Fairchild Semiconductor* 1957 р. та корпорації *Intel* 1968 р. Він також є одним із винахідників інтегральної мікросхеми. Інженери з груп, очолюваних Нойсом, надалі заснували низку видатних напівпровідникових компаній. (Відтворюється з дозволу *Intel Corporation*© 2006.)

Нині, після понад трьох десятиліть безпрецедентного розвитку напівпровідникової технології, інженери можуть «упакувати» приблизно один мільярд польових МОН-транзисторів на одному квадратному сантиметрі кристала кремнію, до того ж кожен із цих транзисторів коштуватиме менше ніж десять мікроцентів. Щільність розміщення транзисторів на кристалі зростає на порядок, а собівартість одного транзистора падає що вісім років.

На цей час польові МОН-транзистори – це ті «цеглинки», з яких збираються майже всі цифрові системи. У цьому розділі вийдемо за межі цифрової абстракції та уважно розглянемо, як можна побудувати логічні вентилі з польових МОН-транзисторів.

1.7.1 Напівпровідники

МОН-транзистори виготовляють з кремнію – елемента, що переважає в скельній породі та піску. Кремній (Si) – це елемент IV атомної групи, тобто він має чотири валентні електрони, може утворювати зв'язки з чотирма сусідніми атомами і, отже, формувати кристалічну ґратку (*lattice*). На рис. 1.24, *a* для спрощеного сприйняття кристалічна ґратка подана у двовимірній системі координат, проте корисно пам'ятати, що реальна кристалічна ґратка має форму куба. Лінія на рис. 1.24, *a* позначає ковалентний зв'язок. За своєю природою кремній – поганий провідник, тому що всі електрони зайняті в ковалентних зв'язках. Однак провідність кремнію покращується, якщо додати до нього незначну кількість атомів іншої речовини, яка називається домішкою (*dopant*). Якщо як домішка використовується елемент V атомної групи, наприклад миш'як (As), то в кожному атомі домішки виявиться додатковий електрон, який не бере участі в утворенні ковалентних зв'язків. Цей вільний електрон може легко переміщатися всередині кристалічних ґраток. У цьому разі атом миш'яку, що втратив електрон, перетворюється на позитивний іон (As^+), як показано на рис. 1.24, *b*. Електрон має негативний заряд (*negative charge*), тому миш'як прийнято називати домішкою *n*-типу (*n-type dopant*). Якщо ж як домішка застосовується елемент III атомної групи, наприклад бор (B), то в кожному з атомів домішки не вистачатиме одного електрона, як продемонстровано на рис. 1.24, *c*. Відсутній електрон називають діркою (*hole*). Електрон із сусіднього атома кремнію може перейти до атома бору й заповнити зв'язок,

якого бракує. У цьому разі атом бору, який отримав додатковий електрон, перетворюється на негативний іон (B-), а в атомі кремнію виникає дірка. Отже, дірка може мігрувати в кристалічній ґратці подібно до електрона. Дірка – це лише відсутність негативного заряду, але вона поводиться в напівпровіднику як позитивно заряджена частинка.

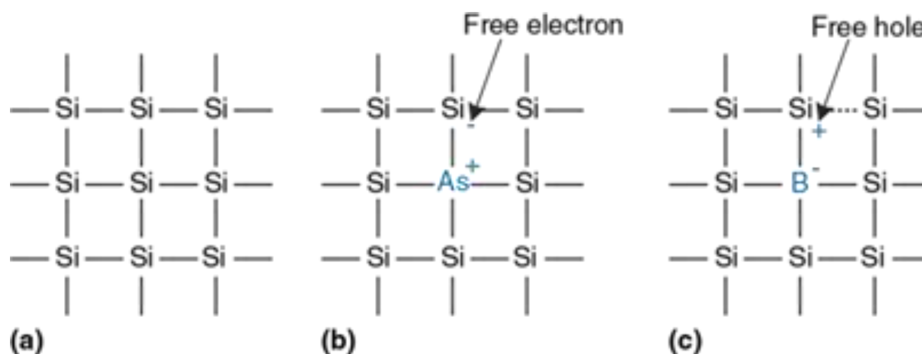


Рисунок 1.24 – Кремнієві ґратки й атоми домішок

Саме тому бор називають домішкою *p*-типу (*p-type dopant*). Оскільки провідність кремнію може значно змінюватися залежно від концентрації домішки, кремній називають напівпровідником (*semiconductor*).

1.7.2 Діоди

Діод (*diode*) – це з'єднання напівпровідника *p*-типу з напівпровідником *n*-типу (рис. 1.25). У цьому разі ділянку *p*-типу називають анодом (*anode*), а ділянку *n*-типу – катодом (*cathode*). Коли напруга на аноді перевищує напругу на катоді, діод відкритий (*forward biased*), і струм крізь нього тече від анода до катода. Якщо ж напруга на аноді нижча за напругу на катоді, то діод закритий (*reverse biased*), і струм крізь діод не тече. Символ діода дуже інтуїтивний і показує, що струм крізь діод може протікати тільки в одному напрямку.

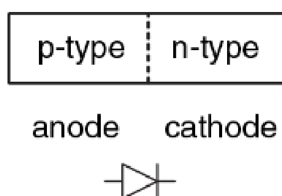


Рисунок 1.25 – Структура діода з *p*-*n*-з'єднанням та його позначка

1.7.3 Конденсатори

Конденсатор (*capacitor*) має два провідники, відділені один від одного ізолятором. Якщо до одного з провідників докласти напругу *V*, за деякий час

цей провідник накопичить електричний заряд Q, а інший провідник – протилежний електричний заряд $-Q$. Ємністю (*capacitance*) C конденсатора називається відношення заряду до відповідної напруги $C = = Q/V$. Ємність прямо пропорційна розміру провідників і зворотно пропорційна відстані між ними. Символ, що застосовується для позначки конденсатора, зображено на рис. 1.26.

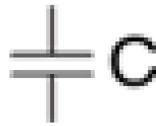


Рисунок 1.26 – Позначка конденсатора

Ємність – це дуже важливий параметр електричної схеми, оскільки зарядка чи розрядка будь-якого провідника потребує часу та енергії. Вища ємність означає, що електрична схема працюватиме повільніше й вимагатиме для свого функціонування більше енергії. До понять швидкості та енергії постійно повертатимемося в цьому посібнику.

1.7.4 n-МОН та p-МОН-транзистори

Польовий МОН-транзистор є «сендвічем» з декількох шарів провідних та ізоляційних матеріалів. «Фундамент», з якого починається побудова польових МОН-транзисторів, – це тонка кругла крем'яна пластина (*wafer*) діаметром приблизно 15–30 см. Виробничий процес починається з порожньої підкладки та передбачає певну послідовність операцій. На підкладці вирощуються тонкі плівки кремнію та діоксиду кремнію й накладається шар металу. Після кожної операції на підкладку, як маска, наноситься певний малюнок (*pattern*), щоб матеріал, який нашаровується на наступній операції, залишався лише в тих місцях, де він необхідний. Оскільки розміри одного транзистора – це частки мікрона, а вся підкладка обробляється під час одного виробничого процесу, коли одночасно виробляються мільярди транзисторів, собівартість одного транзистора значно знижується. Після завершення всіх операцій підкладка нарізається на прямокутні кристали, що називаються в англомовній літературі *chip* або *dice*, до того ж у кожному з цих прямокутників розміщуються тисячі, мільйони й навіть мільярди транзисторів. Кожен такий кристал тестується, а потім поміщається в пластиковий або керамічний корпус-упаковку (*package*) з металевими контактами (*pins*), щоб його можна було встановити на монтажній платі.

«Сендвіч» польового МОН-транзистора містить шар провідника, так званий затвор (*gate*), накладений на шар ізолятора – діоксиду кремнію (SiO_2), що зі свого боку накладений на крем'яну пластину – підкладку. Спочатку для виготовлення

затвора використовувався тонкий шар металу, звідси назва цього типу транзисторів – «метал-оксид-напівпровідник». У сучасних же технологічних процесах як матеріал затвора застосовується полікристалічний кремній, оскільки він не плавиться за умови подальшого високотемпературного оброблення кристала. Діоксид кремнію – це добре відоме всім скло, і в напівпровідниковій промисловості цей матеріал часто називають оксидом. Шари метал – оксид – напівпровідник утворюють конденсатор, у якому тонкий шар оксиду, що називається діелектриком, ізолює металеву пластину від напівпровідникової.

Існують два види польових МОН-транзисторів: *n*-МОН і *p*-МОН (англійською мовою *n-MOS* і *p-MOS*, що вимовляється як «н-мосс» і «пі-мосс»). На рис. 1.27 схематично зображено переріз кожного з цих двох типів транзисторів так, ніби ми розпиляли кристал і тепер дивимося на транзистор збоку. У транзисторах *n*-типу, або *n*-МОН, ділянки, де розташовані напівпровідникові домішки *n*-типу (що називаються витоком (*source*) і стоком (*drain*)), розташовані поруч із затвором (*gate*), до того ж уся ця структура розміщується на підкладці *p*-типу. У транзисторах *p*-МОН і виток, і стік – це ділянки *p*-типу, розміщені на підкладці *n*-типу.

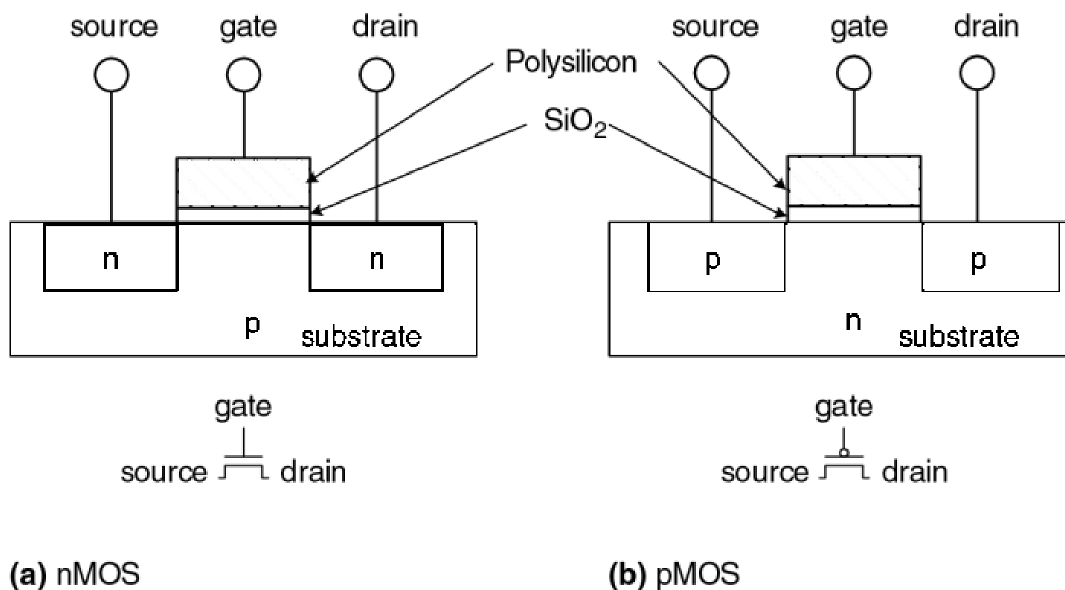


Рисунок 1.27 – *n*-МОН- та *p*-МОН-транзистори

Польовий МОН-транзистор поводиться як перемикач, керований прикладеною до нього напругою. У такому транзисторі напруга переходу створює електричне поле, що вмикає або вимикає лінію зв’язку між джерелом і стоком. Термін «польовий транзистор» (*field effect transistor*) є прямим відтворенням принципу такого пристрою. Ознайомлення з роботою напівпровідникових пристроїв розпочнемо з вивчення *n*-МОН-транзистора.

Підкладка *n*-МОН-транзистора зазвичай розміщена під напругою землі *GND*, що є мінімальною напругою в системі. Спочатку розглянемо випадок, коли напруга на затворі також дорівнює 0 В (див. рис. 1.28, *a*). Діоди між витоком або стоком і підкладкою перебувають у стані, що називається зворотним зміщенням (*reverse bias*), оскільки напруга на витоку та стоку не є негативною. Унаслідок цього канал для руху струму між витоком і стоком залишається закритим, а транзистор вимкнено. Тепер розглянемо ситуацію, коли напруга на затворі підвищується до V_{DD} (див. рис. 1.28, *b*). Якщо прикласти позитивну напругу до затвору (верхньої пластини конденсатора), це створює електричне поле між затвором і підкладкою, зрештою в зоні між витоком і стоком під шаром оксиду формується надлишок електронів. За досить високої напруги на нижній межі затвора накопичується настільки багато електронів, що ділянка з напівпровідником *p*-типу перетворюється на ділянку з напівпровідником *n*-типу. Така інвертована ділянка називається каналом (*channel*). У цей момент у транзисторі утворюється зона провідності джерела *n*-типу, крізь канали *n*-типу до стоку *n*-типу, і крізь цей канал електрони можуть безперешкодно переміщатися від витоку до стоку. Транзистор увімкнено. Напруга переходу, яка потрібна для увімкнення транзистора, називається пороговим значенням напруги (*threshold voltage*) *VT* і зазвичай становить від 0,3 до 0,7 В.

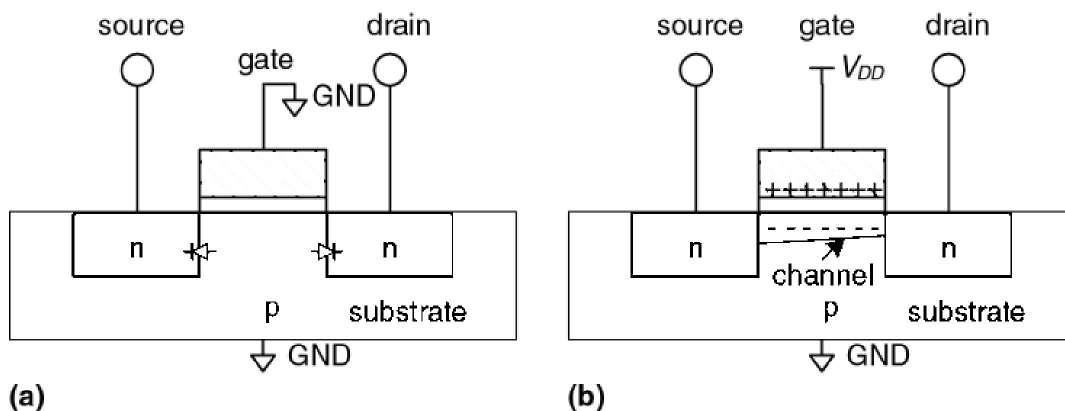


Рисунок 1.28 – Робота *n*-МОН-транзистора

Транзистор *p*-МОН працює з точністю до навпаки (на рис. 1.29 у позначці цього транзистора наявна точка). Підкладка *p*-МОН-транзистора перебуває під напругою V_{DD}. Якщо затвор також є під напругою V_{DD}, *p*-МОН-транзистор вимкнено. Якщо на затвор подається напруга землі *GND*, провідність каналу інвертується, перетворюючись на провідність *p*-типу, і транзистор вмикається.

На жаль, польові МОН-транзистори у ролі перемикача працюють далеко не ідеально. Зокрема *n*-МОН-транзистори добре передають 0, але погано 1.

Якщо перехід *n*-МОН-транзистора перебуває під напругою V_{DD}, то напруга на стоку коливатиметься між 0 і $V_{DD} - VT$. Так само *p*-МОН-транзистори добре передають 1, але погано 0. Однак, як побачимо надалі, можна побудувати логічний вентиль, що добре працюватиме, використовуючи тільки ті режими *n*-МОН- і *p*-МОН-транзисторів, у яких їх робота близька до ідеальної.

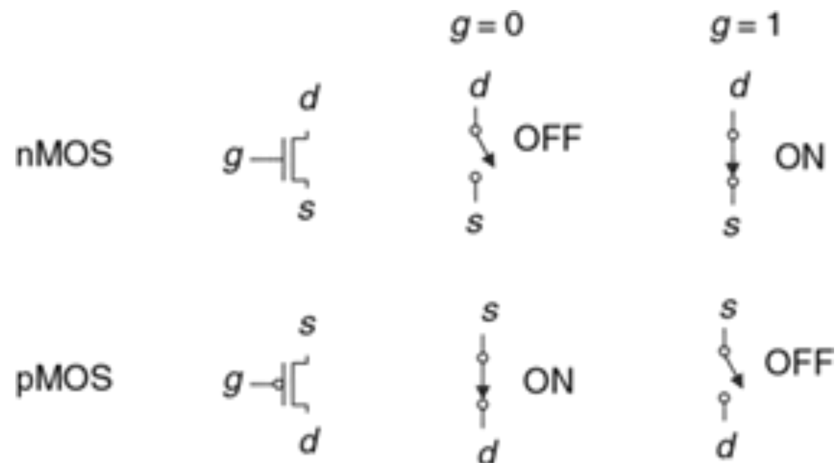


Рисунок 1.29 – Моделі перемикання польових МОН-транзисторів

Для виготовлення *n*-МОН-транзистора потрібна підкладка з провідністю *p*-типу, а для виготовлення транзистора *p*-МОН – підкладка *n*-типу. Щоб розмістити обидва типи транзисторів на одному кристалі, виробничий процес, як правило, починається з підкладки *p*-типу, який потім імплантують ділянки для розміщення *p*-МОН-транзисторів *n*-типу, що називаються колодязями (*wells*). Такий процес має назву «комплементарний МОН» чи КМОН (*Complementary* MOS, чи CMOS). На цей час КМОН-процес використовується для виготовлення переважної більшості транзисторів і мікросхем.

Підіб’ємо підсумок. КМОН-процес дає змогу розмістити МОН-транзистори *n*- та *p*-типу (див. рис. 1.29) на одному кристалі. Напруга на затворі (*g*) керує струмом між джерелом (*s*) і стоком (*d*). Транзистори *n*-МОН вимкнені, коли значення напруги на переході відповідає логічному 0, і увімкнені, коли значення напруги на переході відповідає логічному 1. Транзистори *p*-МОН, навпаки, вмикаються, коли значення напруги на переході відповідає логічному 0, і вимкнені, коли значення напруги на переході відповідає логічному 1.

1.7.5 Логічний вентиль НІ на КМОН-транзисторах

Схема, зображена на рис. 1.30, демонструє, як побудувати логічний елемент НІ за допомогою КМОН-транзисторів.

На цій схемі трикутник позначає напругу землі *GND*, а горизонтальна лінія – напругу живлення V_{DD}.

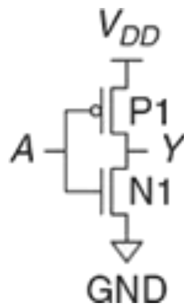


Рисунок 1.30 – Схема вентиля НІ

На всіх наступних схемах у цьому посібнику ми не використовуватимемо буквені позначки V_{DD} та GND.

Транзистор *n*-МОН *N*1 увімкнено між землею *GND* та вихідним контактом *Y*. Зі свого боку *p*-МОН-транзистор *P*1 увімкнено між напругою живлення V_{DD} та вихідним контактом *Y*. Напруга на вхідному контакті *A* управляє переходами обох транзисторів.

Якщо напруга *A* дорівнює 0, то транзистор *N*1 вимкнено, а транзистор *P*1 увімкнено. Зауважимо, що напруга на контакті *Y* дорівнює напрузі живлення V_{DD}, а не землі, що відповідає логічній одиниці. У цьому разі кажуть, що *Y* «підтягнутий» до одиниці (англ. *pulled up*). Увімкнений транзистор *P*1 добре передає логічну одиницю (рівну напрузі живлення), тобто напруга контакту *Y* дуже близька до V_{DD}. Якщо ж напруга на контакті *A* дорівнює логічній одиниці, то транзистор *N*1 увімкнено, а транзистор *P*1 вимкнено, і напруга на контакті *Y* дорівнює напрузі землі, що відповідає логічному нулю. У цьому разі кажуть, що *Y* «підтягнутий» до нуля (англ. *pulled down*). Увімкнений транзистор *N*1 добре передає логічний нуль, тобто напруга контакту *Y* дуже близька до *GND*. Перевірка таблиці істинності на рис. 1.10 підтверджує, що ми маємо справу з логічним вентилем НІ.

1.7.6 Інші логічні вентилі на КМОН-транзисторах

На рис. 1.31 зображено схему для побудови логічного елемента І-НІ з двома вхідними контактами за допомогою МОН-транзисторів.

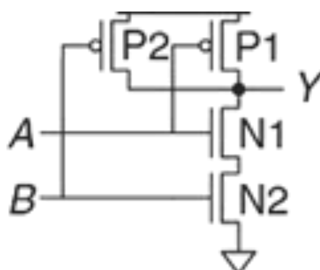


Рисунок 1.31 – Схема вентиля І-НІ з двома входами

На електронних схемах прийнято, що якщо немає жодних додаткових зауваг чи познак, то мається на увазі, що дві лінії поєднуються одна з одною в тому разі, якщо одна з них закінчується в точці перетину (перетин у формі літери Т). Якщо ж обидві лінії продовжуються за точкою перетину, то для позначення контакту цих двох ліній у точці перетину ставиться точка. Якщо точка відсутня, це значить, що лінії не перетинаються, і одна з них проходить над іншою. На рис. 1.31 *n*-МОН-транзистори *N*1 та *N*2 з'єднано послідовно. До того ж щоб замкнути вихідний контакт на землю *GND*, тобто знизити логічний рівень (*pull down*), обидва ці транзистори мають бути увімкнені. Водночас як *p*-МОН-транзистори *P*1 і *P*2 з'єднані паралельно, і тільки один з них має бути увімкнений, щоб з'єднати вихідний контакт із напругою живлення V_{DD}, тобто підвищити логічний рівень (*pull up*). У табл. 1.6 перелічено всі можливі стани для частини схеми, що знижує логічний рівень (*pull-down network*), для частини схеми, що підвищує логічний рівень (*pull-up network*), і виходу. З табл. 1.6 видно, що електрична схема, зображена на рис. 1.31 справді працює як логічний вентиль І-НІ. Наприклад, якщо *A* дорівнює 1 і *B* дорівнює 0, то транзистор *N*1 увімкнено, однак транзистор *N*2 вимкнено й блокує зв'язок контакту *Y* з напругою землі *GND*. У цьому разі транзистор *P*1 вимкнено, а транзистор *P*2 увімкнено та з'єднує напругу живлення V_{DD} з контактом *Y*. Тобто на контакті *Y* маємо 1.

Таблиця 1.6 – Робота вентиля І-НІ

A	***B***	**Схема зниження логічного рівня**	**Схема підвищення логічного рівня**	***Y***
0	0	ВИМК.	УВІМК.	1
0	1	ВИМК.	УВІМК.	1
1	0	ВИМК.	УВІМК.	1
1	1	УВІМК.	ВИМК.	0

На рис. 1.32 в узагальненому вигляді зображені блоки, необхідні для побудови будь-якого інвертувального логічного вентиля, такого як НІ, І-НІ, АБО-НІ.

Транзистори *n*-МОН добре передають 0, тому схема, що знижує логічний рівень (*pull-down network*) і містить такі транзистори, розташована між вихідним контактом і землею *GND* для передачі 0 на вихід. Транзистори *p*-МОН добре передають 1, тому схема, що підвищує логічний рівень (*pull-up network*) і містить такі транзистори, розташована між вихідним контактом і напругою живлення V_{DD} для передачі 1 на вихід. Схема, що знижує, і схема, що підвищує, можуть містити транзистори, з'єднані як паралельно, так і послідовно.

Зауважимо, що за умови паралельного з’єднання транзисторів усю схему увімкнено, якщо увімкнено хоча б один із транзисторів. У разі послідоного з’єднання схема увімкнена тільки тоді, якщо обидва транзистори увімкнено.

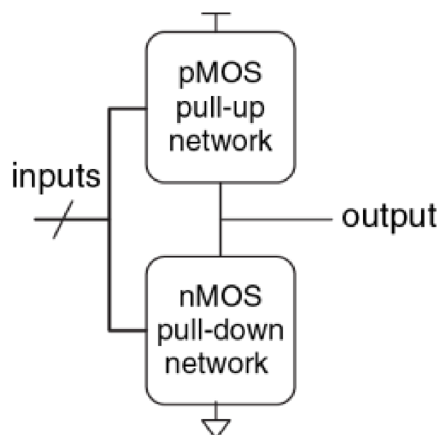


Рисунок 1.32 – Загальна форма інвертувального логічного вентиля

Коса лінія на вхідній лінії свідчить про те, що логічний елемент має кілька входів.

Якщо і понижувальну, і підвищувальну частини схеми увімкнути одночасно, то в усій схемі виникне коротке замикання між напругою живлення V_{DD} та землею *GND*. Сигнал на вихідному контакті може опинитися в забороненій зоні, а транзистори, що споживають значну кількість енергії, можуть перегоріти. З іншого боку, якщо і понижувальну, і підвищувальну частини схеми одночасно вимкнути, вихідний сигнал буде від’єднаний і від V_{DD}, і від *GND*. І тут кажуть, що вихідний сигнал рухається, або плаває (*floats*). Його значення, як і в разі одночасно увімкнених схем, не визначено. Наявність рухомого сигналу на виході системи зазвичай небажана. У правильно функціональному логічному вентилі в будь-який момент часу одна зі схем має бути увімкнена, а інша – вимкнена, і необхідно, щоб напруга на виході була або високою (V_{DD}), або низькою (*GND*). Ні коротке замикання, ні рухоме значення сигналу не допускається. Щоб гарантувати це, користуються правилом доповнення провідності (*conduction complements*). Якщо *n*-МОН-транзистори в будь-якому ланцюгу з’єднані послідовно, *p*-МОН-транзистори в цьому самому ланцюгу мають бути з’єднані паралельно. Якщо ж *n*-МОН-транзистори з’єднані паралельно, тоді *p*-МОН-транзистори мають з’єднуватися послідовно.

Приклад 1.20.

Схема вентиля І-НІ з трьома входами

Наведіть схему вентиля І-НІ з трьома входами, використовуючи КМОН-транзистори.

Виконання. Вентиль І-НІ має видати 0 тільки в тому разі, якщо всі входи дорівнюють 1. Отже, необхідно, щоб схема, яка знижує логічний рівень, мала три послідовно увімкнених *n*-МОН-транзистори. За правилом доповнень (*Conduction complements rule*) *p*-МОН-транзистори мають бути увімкнені паралельно. Такий вентиль зображений на рис. 1.33. Ви можете переконатись у правильності функціонування перевіркою таблиці істинності.

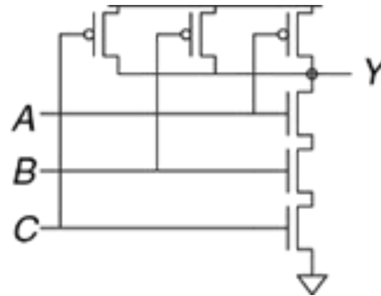


Рисунок 1.33 – Схема вентиля І-НІ з трьома входами

Приклад 1.21.

Схема вентиля АБО-НІ з двома входами

Наведіть схему вентиля АБО з двома входами, використовуючи КМОН-транзистори.

Виконання. Вентиль АБО не має видавати 0, якщо хоча б один із входів дорівнює 1. Отже, необхідно, щоб схема, яка знижує логічний рівень, мала два *n*-МОН-транзистори, увімкнених паралельно. За правилом доповнень (*Conduction complements rule*) *p*-МОН-транзистори мають бути увімкнені послідовно. Такий вентиль зображено на рис. 1.34.

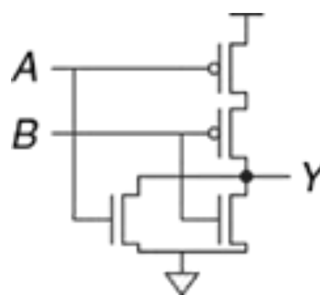


Рисунок 1.34 – Схема вентиля АБО-НІ з двома входами

Приклад 1.22.

Схема вентиля І з двома входами

Накресліть схему вентиля І з двома входами, застосовуючи КМОН-транзистори.

Виконання. Схему І неможливо побудувати на основі одного КМОН-вентиля. Однак побудова І-НІ- та НІ-вентилів – річ досить проста.

Отже, найкращий спосіб побудувати вентиль І, використовуючи КМОН-транзистори, полягає в тому, щоб застосовувати І-НІ, за яким слідує НІ, як показано на рис. 1.35.

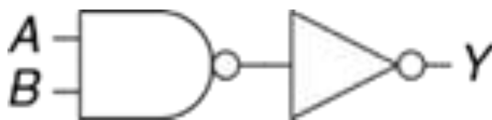


Рисунок 1.35 – Схема вентиля І з двома входами

1.7.7 Передавальний логічний вентиль

Іноді розробнику необхідний ідеальний перемикач, що може однаково добре передавати як 0, так і 1. Згадаємо, що *n*-МОН-транзистори добре передають 0, а *p*-МОН-транзистори добре передають 1, і паралельне з'єднання цих двох транзисторів має добре передавати обидва ці значення.

На рис. 1.36 показано так званий передавальний логічний елемент (*transmission gate*), прохідний логічний вентиль (*pass gate*) або аналоговий ключ. Виводи цього елемента позначаються *A* і *B*, оскільки передача сигналу в такому логічному вентилі може йти у двох напрямках, і жоден з них не має переваг. Сигнали управління (в англомовній літературі мають назву *enables*), позначаються EN і $\overline{EN}$. Якщо EN дорівнює 0, а $\overline{EN}$ дорівнює 1, то обидва транзистори вимкнені. У цьому разі весь передавальний логічний вентиль вимкнено й контакт *A* не має зв'язку з контактом *B*.

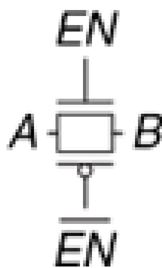


Рисунок 1.36 – Передавальний вентиль

Якщо EN дорівнює 1, а $\overline{EN}$ дорівнює 0, то передавальний логічний вентиль увімкнено, і будь-яке логічне значення передається від *A* до *B*.

1.7.8 Псевдо-n-МОН-логіка

Побудований за технологією КМОН логічний вентиль АБО, у якого кількість вхідних контактів дорівнює *N*, використовує *N* паралельно увімкнених *n*-МОН-транзисторів і *N* послідовно увімкнених *p*-МОН-транзисторів. Послідовно увімкнені транзистори передають сигнал повільніше, ніж ті, що увімкнені паралельно, так само як опір резисторів, увімкнених послідовно, буде більшим,

ніж опір резисторів, увімкнених паралельно. Крім того, *p*-МОН-транзистори передають сигнали повільніше, ніж *n*-МОН-транзистори, оскільки дірки не можуть переміщатися кристалічною ґраткою кремнію так само швидко, як електрони. Як наслідок, з'єднані паралельно *n*-МОН-транзистори працюють швидко, а з'єднані послідовно *p*-МОН-транзистори – повільно, особливо якщо їх багато [1].

Як продемонстровано на рис. 1.37, у процесі використання псевдо-*n*-МОН-логіки (*pseudo-nMOS logic*), або просто псевдологіки, повільний стік з *p*-МОН-транзисторів замінюють одним «слабким» *p*-МОН-транзистором, що завжди перебуває у відкритому стані. Такий транзистор часто називають слабким транзистором, що підтягує (*weak pull-up*). Фізичні параметри *p*-МОН-транзистора обираються таким чином, що він до високого логічного рівня (1) вихід *Y* «підтягує слабко», тобто тільки в тому разі, коли всі *n*-МОН-транзистори вимкнені. Але якщо хоча б один з *n*-МОН-транзисторів вмикається, то він, перевершуючи за потужністю слабкий транзистор, «перетягує» вихід *Y* настільки близько до напруги землі *GND*, що на виході маємо логічний 0.

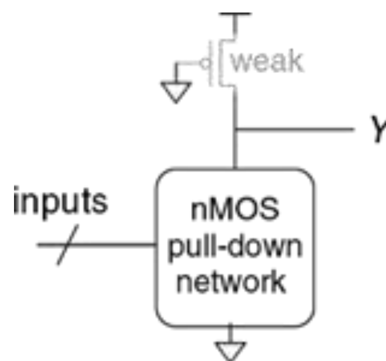


Рисунок 1.37 – Узагальнений псевдо-*n*-МОН-вентиль

Перевага псевдологіки полягає в тому, що її можна використовувати для створення швидких АБО-НІ-вентилів із значною кількістю входів. Наприклад, на рис. 1.38 проілюстровано вентиль АБО-НІ з чотирма входами, що побудований із застосуванням псевдологіки.

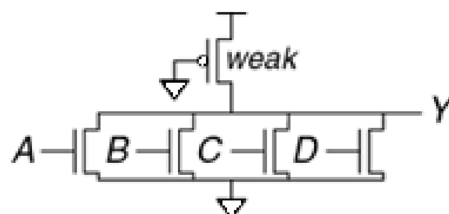


Рисунок 1.38 – Псевдо-*n*-МОН-вентиль АБО-НІ з чотирма входами

Логічні вентилі, які використовують псевдологіку, можуть бути дуже ефективними для побудови деяких видів пам'яті та логічних масивів.

Недолік псевдологіки – наявність короткого замикання між живленням V_{DD} та землею *GND*, коли сигнал на виході – це логічний нуль (0). Слабкі *p*-МОН- та *n*-МОН-транзистори вимкнені. У цьому разі через коротке замикання постійно протікає струм, і електрична енергія від джерела живлення витрачається марно. Саме тому псевдо-*n*-МОН-логіка використовується обмежено.

Термін «псевдо-*n*-МОН-логіка» впроваджено в минулому столітті. Тоді існував виробничий процес виготовлення лише *n*-МОН-транзисторів. Слабкі *n*-МОН-транзистори використовувалися для «підтягування» вихідного сигналу до логічної одиниці (1), оскільки *p*-МОН-транзисторів просто не було.

1.8 Споживана потужність

Потужність – це кількість енергії, що споживається системою за одиницю часу. Енергоспоживання має вагоме значення в цифрових системах. Саме споживана потужність визначає час автономної роботи без підзарядки батареї будь-якого портативного пристрою, наприклад мобільного телефона або ноутбука. Не варто думати, однак, що споживана потужність – другорядний параметр для стаціонарних пристроїв. Електрика дорога, і до того ж будь-який пристрій може перегрітися, якщо споживає занадто багато електроенергії.

Цифрова система споживає енергію як у динамічному режимі, коли виконує будь-які операції, і в статичному, коли система перебуває в стані спокою (*idle*). У динамічному режимі енергія витрачається на зарядку ємностей елементів системи, коли ці елементи перемикаються між 0 і 1. І хоча в статичному режимі жодних перемикань не відбувається, система все одно витрачає електричну енергію.

І самі логічні вентилі, і провідники, що з'єднують їх один з одним, є конденсаторами й мають певну ємність. Енергія, яку отримують від блоку живлення і яку необхідно витратити на зарядку ємності *C* до напруги V_{DD}, дорівнює $C \times V^2$. Якщо напруга на конденсаторі перемикається з частотою *f* (тобто *f* разів за секунду), то конденсатор заряджається *f*/2 раза та розряджається *f*/2 раза за секунду. І оскільки в процесі розрядження конденсатор не споживає енергію від джерела живлення, тоді споживання енергії в динамічному режимі можна обчислити як

$$\text{Pdynamic} = \frac{1}{2} C * V_{DD}^2 * f. \quad (1.4)$$

Струм у системі реєструється, навіть якщо вона перебуває в стані спокою. У деяких типів електронних схем, таких як псевдо-*n*-МОН-логіка, розглянута в п. 1.7.8, існує шлях, що з'єднує напругу живлення V_{DD} із землею *GND*,

крізь який струм протікає постійно. Сумарна величина струму, що протікає в системі в її статичному стані I_{DD}, називається струмом витоку (*leakage current*), або струмом спокою (*quiescent supply current*). Потужність, що споживається системою в статичному стані, пропорційна величині струму витоку й може бути обчислена як

$$Pstatic = I_{DD}*V_{DD}. \quad (1.5)$$

Приклад 1.23.

Споживана потужність

Стільниковий телефон певної моделі має акумулятор ємністю 6 Вт·год та працює від напруги 1,2 В. Припустимо, що під час використання пристрій працює на частоті 300 МГц і середня ємність цифрової схеми в будь-який конкретний момент становить 10 нФ (10^{-8} Фаради). У процесі роботи телефон також видає сигнал потужністю 3 Вт на антену. Коли пристрій не використовується, динамічна споживана потужність падає практично до нуля, оскільки оброблення сигналів вимкнене. Але телефон також споживає 40 мА струму спокою в будь-якому разі, працює він чи ні. Розрахуйте час, на який вистачить акумулятора пристрою для таких випадків:

(а) якщо телефон не використовується;

(б) якщо телефон використовується безперервно.

Виконання. Статична потужність *Pstatic* дорівнює (0,040 А) * (1,2 В) = =48 мВт. Якщо телефон не застосовується – це єдине споживання потужності, тому час життя акумулятора дорівнює (6 Вт·год)/(0,048 Вт) = 125 год (приблизно п'ять днів). Якщо телефон використовується, динамічна потужність *Pdynamic* дорівнює (0,5)*(10^{-8} Ф)*(1,2 В)2 *(3 * 108 Гц) = 2,16 Вт. Загальна потужність, яка є сумою *Pdynamic, Pstatic* та потужності мовлення, становитиме 2,16 Вт + 0,048 Вт + 3 Вт = 5,2 Вт, тому час життя акумулятора дорівнюватиме 6 Вт·год / 5,2 Вт = 1,15 год. Цей приклад подає фактичну роботу телефона в дещо спрощеному вигляді, але водночас він ілюструє ключові ідеї щодо потужності споживання.

1.9 Резюме

Існує два види людей: ті, хто ознайомлений з двійковою системою числення, і ті, хто не знає про неї нічого.

У цьому розділі ми висвітлили основні принципи, необхідні для розуміння та проєктування складних електронних систем. І хоча фізичні

величини в реальному світі здебільшого аналогові, тобто змінюються безперервно, розробники цифрових систем обмежуються розглядом наприкінці підмножини дискретних величин безперервно змінних сигналів. Зокрема бінарні змінні можуть набувати лише двох значень – 0 і 1, які ще називаються НЕПРАВДА (*FALSE*) та ІСТИНА (*TRUE*) або НИЗЬКИЙ (*LOW*) і ВИСОКИЙ (*HIGH*). Логічні вентилі певним чином перетворюють сигнали з одного або кількох двійкових входів на двійковий сигнал на виході. Деякі з найчастіше використовуваних логічних вентилів перелічені нижче.

● НІ: має на виході значення *TRUE*, якщо сигнал на вході має значення *FALSE*.

● І: має на виході значення *TRUE*, якщо всі сигнали на вході мають значення *TRUE*.

● АБО: має на виході значення *TRUE*, якщо хоча б один сигнал на вході має значення *TRUE*.

● Виключне АБО: має на виході значення *TRUE*, якщо непарна кількість сигналів на вході має значення *TRUE*.

Для побудови логічних вентилів зазвичай використовують КМОН-транзистори, що, власне, є перемикачами з електричним управлінням. Транзистор *n*-МОН вмикається, якщо затвор перебуває під напругою V_{DD}, що відповідає логічній одиниці. Транзистор *p*-МОН вмикається, якщо затвор перебуває під напругою *GND*, який відповідає логічному нулю.

У розділах 2–5 продовжимо вивчати цифрову логіку. Зокрема в розділі 2 проаналізовано комбінаторну логіку (*combinational logic*), яка передбачає, що сигнал на виході логічного вентиля залежить від станів на вхідних контактах цього елемента в конкретний момент часу. Ті логічні вентилі, описані вище, можуть слугувати прикладом використання комбінаторної логіки. З розділу 2 також стане зрозуміло, як можна спроєктувати схему з кількох логічних вентилів таким чином, щоб усі можливі стани цієї схеми відповідали станам, наперед описаним у таблиці істинності або за допомогою рівняння булевої логіки.

У розділі 3 подано послідовну логіку (*sequential logic*), яка вже припускає, що результат на виході логічного вентиля залежить як від поточного стану на вході, так і від його станів. Регістр (*Register*) – це найпоширеніший елемент послідовної логіки, який «запам’ятовує» попередній стан на своєму вході. Кінцевий автомат (*finite state machines*), побудований на базі регістрів та комбінаторної логіки, є потужним засобом створення складних систем на системній основі. У розділі 3 також розглянемо часові співвідношення

сигналів у цифровій системі, щоб визначити максимально можливу швидкість, де ця система здатна нормально працювати.

У розділі 4 розглянуто мови опису обладнання (*hardware description languages*, або *HDL*). *HDL* – родичі звичайних мов програмування, але використовуються здебільшого для моделювання та створення апаратного, а не програмного забезпечення. Більшість сучасних цифрових систем розробили з використанням *HDL*. *SystemVerilog* і *VHDL* – дві найпоширеніші мови для опису та верифікації апаратури, й обидві вони розглядаються в цьому посібнику. *VHDL* (*Very high-speed integrated circuits Hardware Description Language*) перекладається як мова для опису та верифікації апаратури на дуже високошвидкісних інтегральних схемах.

ВПРАВИ

Вправа 1.1. Поясніть щонайменше три рівні абстракції, що застосовують:

a) біологи, які вивчають роботу клітин;

b) хіміки, які вивчають склад будь-якого матеріалу. (Ваше пояснення не має перевищувати один абзац.)

Вправа 1.2. Поясніть, як методи ієрархічності, модульності та регулярності можуть використовувати:

a) конструктори автомобілів;

b) будь-який бізнес для управління щоденними операціями. (Ваше пояснення не має перевищувати один абзац.)

Вправа 1.3. Бен Бітдідл будує будинок. (*Прим. перекладача*: Бен Бітдідл – персонаж, створений Стівом Уордом (*Steve Ward*) упродовж 1970-х рр. і відтоді широко використовується як герой збірників завдань у Массачусетському технологічному інституті (*Massachu of Technology*) і поза ним. Прізвище Бена походить від терміна *bit diddling*, який можна перекласти як «бітове жонглювання» – програмування на рівні машинних кодів із маніпулюванням бітами, прапорами, напівбайтами та іншими елементами розміром меншим за символ.) Поясніть йому, як він може використовувати принципи ієрархічності, модульності та регулярності, щоб заощадити час та ресурси.

Вправа 1.4. Припустимо, що напруга аналогового сигналу в нашій системі змінюється не більше ніж 0 вольтів до 5 вольтів. Якщо можемо виміряти цю напругу з точністю до ±50 мілівольтів, то яку максимальну кількість інформації в бітах цей сигнал може передавати?

Вправа 1.5. На стіні висить старий годинник з відламаною хвилинною стрілкою.

a) Якщо використовувати тільки годинникову стрілку, ви можете визначити поточний час з точністю до 15 хв. Скільки бітів інформації про час ви можете отримати, дивлячись на цей годинник?

b) Якщо ви знатимете, яка зараз половина дня – до або після полудня, то скільки додаткових бітів інформації про поточний час ви отримаєте?

Вправа 1.6. Приблизно 4000 років тому вавилоняни розробили шістдесяткову (на підставі 60) систему числення. Скільки бітів інформації передає одна шістдесяткова цифра? Як можна записати число 400010, використовуючи шістдесяткову систему числення?

Вправа 1.7. Як багато різних чисел може бути подано 16 бітами?

Вправа 1.8. Яке максимальне значення може бути подано 32-розрядним двійковим числом?

Вправа 1.9. Яке максимальне 16-розрядне двійкове число ви можете подати, застосовуючи системи подання двійкових чисел, наведені нижче?

a) Двійкове число без знака (*unsigned*).

b) Додатковий код (*two's complement*).

c) Прямий код (*sign / magnitude*).

Вправа 1.10. Яке максимальне 32-розрядне двійкове число ви можете подати, використовуючи системи подання двійкових чисел, наведені нижче?

a) Двійкове число без знака (*unsigned*).

b) Додатковий код (*two's complement*).

c) Прямий код (*sign / magnitude*).

Вправа 1.11. Яке мінімальне (найменше негативне) 16-розрядне двійкове число ви можете подати, застосовуючи системи подання двійкових чисел, наведені нижче?

a) Двійкове число без знака (*unsigned*).

b) Додатковий код (*two's complement*).

c) Прямий код (*sign / magnitude*).

Вправа 1.12. Яке мінімальне (найменше негативне) 32-розрядне двійкове число ви можете подати, використовуючи системи подання двійкових чисел, наведені нижче?

a) Двійкове число без знака (*unsigned*).

b) Додатковий код (*two's complement*).

c) Прямий код (*sign / magnitude*).

Вправа 1.13. Перетворіть двійкові числа без знака на десяткові.

a) 10_2;

b) 110110_2;

c) 11110000_2;

d) $0001000101001 1_2$.

Вправа 1.14. Перетворіть наступні двійкові числа без знака на десяткові.

a) 1110_2;

b) 100100_2;

c) 11010111_2;

d) $011101010110010 0_2$.

Вправа 1.15. Перетворіть двійкові числа без знака з вправи 1.13 у шістнадцяткові.

Вправа 1.16. Перетворіть двійкові числа без знака з вправи 1.14 у шістнадцяткові.

Вправа 1.17. Перетворіть подані шістнадцяткові числа на десяткові.

a) $\mathrm{A5}_{16}$;

b) $\mathrm{3B}_{16}$;

c) $FFFF_{16}$;

d) $D0000000_{16}$.

Вправа 1.18. Перетворіть подані шістнадцяткові числа на десяткові.

a) $4E_{16}$;

b) $7C_{16}$;

c) $ED3A_{16}$;

d) $403FB00_{16}$.

Вправа 1.19. Перетворіть шістнадцяткові числа із вправи 1.17 у двійкові числа без знака.

Вправа 1.20. Перетворіть шістнадцяткові числа з вправи 1.18 у двійкові числа без знака.

Вправа 1.21. Перетворіть двійкові числа, подані в додатковому коді, на десяткові.

a) 1010_2;

b) 110110_2;

c) 01110000_2;

d) 10011111_2.

Вправа 1.22. Перетворіть двійкові числа, подані в додатковому коді, на десяткові.

a) 1110_2;

b) 100011_2;

c) 01001110_2;

d) 10110101_2.

Вправа 1.23. Перетворіть двійкові числа з вправи 1.21 у десяткові, вважаючи, що ці двійкові числа подані не в додатковому, а в прямому коді.

Вправа 1.24. Перетворіть двійкові числа з вправи 1.22 в десяткові, вважаючи, що ці двійкові числа подані не в додатковому, а в прямому коді.

Вправа 1.25 Перетворіть наведені десяткові числа на двійкові числа без знака.

a) 4210;

b) 6310;

c) 22910;

d) 84510.

Вправа 1.26. Перетворіть наведені десяткові числа на двійкові числа без знака.

a) 14_{10};

b) 52_{10};

c) 339_{10};

d) 7111_{10}.

Вправа 1.27. Перетворіть десяткові числа із вправи 1.25 у шістнадцяткові.

Вправа 1.28. Перетворіть десяткові числа із вправи 1.26 у шістнадцяткові.

Вправа 1.29. Перетворіть десяткові числа на 8-бітні двійкові числа, подані в додатковому коді. Укажіть, чи має місце переповнення.

a) 42_{10};

b) -63_{10};

c) 124_{10};

d) -128_{10};

e) 133_{10}.

Вправа 1.30. Перетворіть десяткові числа на 8-бітні двійкові числа, подані в додатковому коді. Укажіть, чи має місце переповнення.

a) 24_{10};

b) -59_{10};

c) 128_{10};

d) -150_{10};

e) 127_{10}.

Вправа 1.31. Перетворіть десяткові числа із вправи 1.29 у 8-бітні двійкові числа, подані в прямому коді.

Вправа 1.32. Перетворіть десяткові числа із вправи 1.30 у 8-бітні двійкові числа, подані в прямому коді.

Вправа 1.33. Перетворіть 4-розрядні двійкові числа, подані в додатковому коді, на 8-розрядні двійкові числа, подані в додатковому коді.

a) 0101_2;

b) 1001_2.

Вправа 1.34. Перетворіть наступні 4-розрядні двійкові числа, подані в додатковому коді, на 8-розрядні двійкові числа, подані в додатковому коді.

a) 0111_2;

b)$1001_{2.}$

Вправа 1.35. Перетворіть 4-розрядні двійкові числа з вправи 1.33 у 8-розрядні, вважаючи, що це двійкові числа без знака.

Вправа 1.36. Перетворіть 4-розрядні двійкові числа з вправи 1.34 у 8-розрядні, вважаючи, що це двійкові числа без знака.

Вправа 1.37. Система числення на підставі 8 називається вісімковою (*octal*). Подайте кожне з чисел у вправі 1.25 у вісімковому вигляді.

Вправа 1.38. Система числення на підставі 8 називається вісімковою (*octal*). Подайте кожне з чисел у вправі 1.26 у вісімковому вигляді.

Вправа 1.39. Перетворіть кожне з поданих вісімкових чисел у двійкове, шістнадцяткове та десяткове.

a) 42_8;

b) 63_8;

c) 255_8;

d) $3041_{8.}$

Вправа 1.40. Перетворіть кожне з наведених вісімкових чисел у двійкове, шістнадцяткове та десяткове.

a) 23_8;

b) 41_8;

c) 371_8;

d) 2560_8.

Вправа 1.41. Скільки 5-розрядних двійкових чисел, поданих у додатковому коді, мають значення понад 0? Скільки – менше ніж 0? Якою буде правильна відповідь у разі 5-розрядних двійкових чисел, поданих у прямому коді?

Вправа 1.42. Скільки 7-розрядних двійкових чисел, поданих у додатковому коді, мають значення понад 0? Скільки – менше ніж 0? Якою буде правильна відповідь у разі 7-розрядних двійкових чисел, поданих у прямому коді?

Вправа 1.43. Скільки байтів у 32-бітному слові? Скільки напівбайтів?

Вправа 1.44. Скільки байтів у 64-бітному слові?

Вправа 1.45. Якщо *DSL*-модем працює зі швидкістю 768 кбіт/сек, скільки байтів може передати за 1 хв?

Вправа 1.46. *USB*3.0 передає інформацію зі швидкістю 5 Гбіт/сек. Скільки байтів *USB*3.0 може передати за хвилину?

Вправа 1.47. Виробники жорстких дисків вимірюють обсяги інформації в мегабайтах, що означає 10^6 байтів, та гігабайтах, що означає 10^9 байтів. Скільки гігабайтів музики можна зберегти на 50-гігабайтному жорсткому диску?

Вправа 1.48. Без використання калькулятора розрахуйте приблизно 2^{31}.

Вправа 1.49. Пам'ять процесора *Pentium* II організована як прямокутний масив бітів, що містить 28 рядків та 29 стовпців. Без застосування калькулятора розрахуйте приблизну кількість бітів у цьому масиві.

Вправа 1.50. Накресліть цифрову шкалу, аналогічну зображеній на рис. 1.11 для 3-бітного двійкового числа, поданого в додатковому коді та прямому коді.

Вправа 1.51. Накресліть цифрову шкалу, аналогічну до зображеної на рис. 1.11 для 2-бітного двійкового числа, поданого в додатковому коді та прямому коді.

Вправа 1.52. Додайте такі двійкові числа без знака:

a) $1001_2 + 0100_2$;

b) $1101_2 + 1011_2$.

Укажіть, якщо сума переповнює 4-бітний регістр.

Вправа 1.53. Додайте такі двійкові числа без знака:

a) $10011001_2 + 01000100_2$;

b) $11010010_2 + 1011110_2$.

Вкажіть, якщо сума переповнює 8-бітний регістр.

Вправа 1.54. Виконайте вправу 1.52 за умови, що двійкові числа у вправі подано в додатковому коді.

Вправа 1.55. Виконайте вправу 1.53 за умови, що двійкові числа у вправі подано в додатковому коді.

Вправа 1.56. Перетворіть десяткові числа на 6-бітні двійкові числа, які подано в додатковому коді, та складіть їх.

a) $16_{10} + 9_{10}$;

b) $27_{10} + 31_{10}$;

c) $-41_{10} + 19_{10}$;

d) $31_{10} + -32_{10}$;

e) $-16_{10} + -9_{10}$;

f) $-27_{10} + -31_{10}$.

Укажіть, якщо сума переповнює 6-бітний регістр.

Вправа 1.57. Перетворіть десяткові числа на 6-бітні двійкові числа, які подано в додатковому коді, та складіть їх.

a) $7_{10} + 13_{10}$;

b) $17_{10} + 25_{10}$;

c) $-26_{10} + 8_{10}$;

d) $31_{10} + -14_{10}$;

e) $-19_{10} + -22_{10}$;

f) $-21_{10} + -29_{10}$.

Укажіть, якщо сума переповнює 6-бітний регістр.

Вправа 1.58. Складіть такі шістнадцяткові числа без знака:

a) $7_{16} + 9_{16}$;

b) $13_{16} + 28_{16}$;

c) $AB_{16} + 3E_{16}$;

d) $8F_{16} + AD_{16}$.

Укажіть, якщо сума переповнює 8-бітний регістр (два шістнадцяткові числа).

Вправа 1.59. Складіть такі шістнадцяткові числа без знака:

a) $22_{16} + 8_{16}$;

b) $73_{16} + 2C_{16}$;

c) $7F_{16} + 7F_{16}$;

d) $C2_{16} + A4_{16}$.

Укажіть, якщо сума переповнює 8-бітний регістр (два шістнадцяткові числа).

Вправа 1.60. Перетворіть десяткові числа на 5-розрядні двійкові числа, які подано в додатковому коді, і відніміть одне від одного:

a) $9_{10} - 71_{10}$;

b) $12_{10} - 15_{10}$;

c) $-61_{10} - 11_{10}$;

d) $4_{10} - 8_{10}$.

Укажіть, якщо різниця переповнює 5-бітний регістр.

Вправа 1.61. Перетворіть десяткові числа на 6-розрядні двійкові числа, які подано в додатковому коді, і відніміть одне від одного:

a) $18_{10} - 12_{10}$;

b) $30_{10} - 9_{10}$;

c) $-28_{10} - 3_{10}$;

d) $-16_{10} - 21_{10}$.

Укажіть, якщо різницю переповнює 6-бітний регістр.

Вправа 1.62. У *N*-бітній двійковій системі числення зі зміщенням *B* (*N-bit binary number system with bias B*) позитивні та негативні числа видаються як значення цих чисел у звичайній двійковій системі, плюс зміщення *B*. Наприклад, для 5-бітної двійкової системи числення з усуненням 15 число 0 подається як 01111, а число 1 – як 10000 і т.д. Системи числення зі зміщенням іноді використовуються для виконання математичних операцій з рухомою комою, що буде розглянуто в розділі 5. Дайте відповідь на подані нижче запитання щодо 8-бітної системи числення зі зміщенням 127_{10}.

a) Яке десяткове значення відповідає двійковому числу 10000010_2?

b) Яке двійкове число відповідає значенню 0?

c) Як у такій системі матиме вигляд мінімальне негативне двійкове число та яким буде його десятковий еквівалент?

d) Як у такій системі матиме вигляд максимальне позитивне двійкове число та яким буде його десятковий еквівалент?

Вправа 1.63. Наведіть цифрову шкалу, аналогічну зображеній на рис. 1.11 для 3-бітного двійкового числа з усуненням рівним 3. Що таке система числення зі зміщенням, пояснюється у вправі 1.62.

Вправа 1.64. У двійково-кодованій десятковій системі числення (*binary-coded decimal system,* або *BCD*) 4 біти використовуються для подання десяткових чисел від 0 до 9. Наприклад, 37_{10} записується як 00110111_{BCD}.

Дайте відповідь на наведені нижче запитання щодо двійково-кодованої десяткової системи числення.

a) Який вигляд має 289_{10} у двійково-кодованій десятковій системі числення?

b) Який вигляд має десятковий еквівалент 100101010001_{BCD}?

c) Який вигляд має двійковий еквівалент 01101001_{BCD}?

d) Які, на вашу думку, переваги має двійково-кодована десяткова система числення?

Вправа 1.65. Дайте відповідь на подані нижче запитання щодо двійково-кодованої десяткової системи числення.

a) Який вигляд має 371_{10} у двійково-кодованій десятковій системі числення?

b) Який вигляд має десятковий еквівалент 000110000111_{BCD}?

c) Який вигляд має двійковий еквівалент 10010101_{BCD}?

d) Які, на вашу думку, недоліки має двійково-кодована десяткова система числення порівняно з двійковою?

Що таке двійково-кодована десяткова система числення зі зміщенням, пояснюється у вправі 1.64.

Вправа 1.66. Археологи виявили з-поміж уламків запис із формулами, поданими в деякій системі числення. Одна з формул має такий вигляд: 325 + 42 = 411. Якщо ця формула записана без помилок, у якій системі числення вона записана?

Вправа 1.67. У Бена Бітдідла та Аліси П. Хакер виникла суперечка. (В англомовному варіанті ім'я *Alyssa P. Hacker* співзвучне вислову *a LISP hacker*, тобто *LISP*-хакер (*LISP* – сімейство функціональних мов програмування)). Бен стверджує, що в усіх цілих чисел, які більше за нуль і кратні шести, є точно дві одиниці у двійковому поданні. Аліса не погоджується. На її думку, всі такі числа мають парну кількість одиниць у їх поданні. Ви погоджуєтеся з Беном, чи з Алісою, чи з обома, чи із жодним з них? Поясніть.

Вправа 1.68. Бен Бітділ та Аліса П. Хакер ще раз сперечаються. Бен каже: «Я можу отримати двійкове доповнення числа способом віднімання 1, а потім інвертую всі біти». Аліса відповідає: «Ні, я можу це зробити способом перевірки кожного біта, починаючи з найменш значущих. Коли стикнуся з першою 1, інвертую кожен наступний біт». Ви погоджуєтеся з Беном, чи з Алісою, чи з обома, чи із жодним з них? Поясніть.

Вправа 1.69. Напишіть програму вашою улюбленою мовою (наприклад, *C, Java, Perl*) для перетворення двійкових чисел на десяткові. Користувач має ввести беззнакове двійкове число. Програма має видати його десятковий еквівалент.

Вправа 1.70. Повторіть вправу 1.69, але для перетворення чисел у системі числення з довільною базою $b1$ у числа в системі числення з іншою базою $b2$. Підтримайте всі бази до 16, для цифр понад 9 використовуйте літери алфавіту. Користувач має ввести $b1$, $b2$, а потім число в системі числення з базою $b1$. Програма має видати еквівалентне число в системі числення з базою $b2$.

Вправа 1.71. Накресліть позначку, подайте логічне рівняння та таблицю істинності для

a) вентиля АБО з трьома входами;

b) вентиля, що виключає АБО, з трьома входами;

c) вентиля, що виключає АБО-НІ, з чотирма входами.

Вправа 1.72. Накресліть позначку, подайте логічне рівняння й таблицю істинності для

a) вентиля АБО з чотирма входами;

b) вентиля, що виключає АБО-НІ, з трьома входами;

c) вентиля І-НІ з п'ятьма входами.

Вправа 1.73. Мажоритарний вентиль видає значення ІСТИНА тоді й лише тоді, коли понад половина його входів мають значення ІСТИНА.

Заповніть таблицю істинності для мажоритарного вентиля, як показано на рис. 1.39.

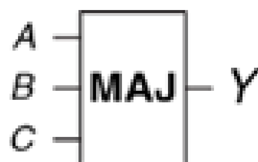


Рисунок 1.39 – Мажоритарний вентиль із трьома входами

Вправа 1.74. Вентиль І-АБО з трьома входами, зображений на рис. 1.40, видає значення ІСТИНА, якщо входи *A* і *B* мають значення ІСТИНА або вхід *C* має значення ІСТИНА. Заповніть таблицю істинності цього вентиля.

Рисунок 1.40 – Вентиль І-АБО з трьома входами

Вправа 1.75. Вентиль АБО-І з інверсією з трьома входами, зображений на рис. 1.41, видає значення НЕПРАВДА, якщо вхід *C* має значення ІСТИНА і входи *A* або *B* мають значення ІСТИНА. Інакше вентиль видає значення ІСТИНА. Заповніть таблицю істинності цього вентиля.

Рисунок 1.41 – Інвертований вентиль АБО-І з трьома входами

Вправа 1.76. Є 16 різних таблиць істинності для булевих функцій двох змінних. Дослідіть ці таблиці, даючи кожній одне коротке описове ім'я (наприклад, АБО, І-НІ тощо).

Вправа 1.77. Скільки різних таблиць істинності для булевих функцій від *N* змінних?

Вправа 1.78. Чи можна призначити логічні рівні так, щоб пристрій із передавальною властивістю (яку подано на рис. 1.42), міг слугувати інвертором? Якщо так, то якими є вхідні та вихідні низькі та високі рівні (*VIL, VOL, VIH* і *VOH*) та рівні шуму (*NML* та *NMH*)? Якщо це не так, то поясніть чому.

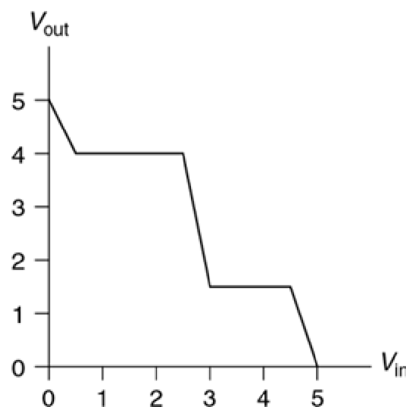


Рисунок 1.42 – Передавальна властивість

Вправа 1.79. Повторіть вправу 1.78 для передавальної властивості, яку подано на рис. 1.43.

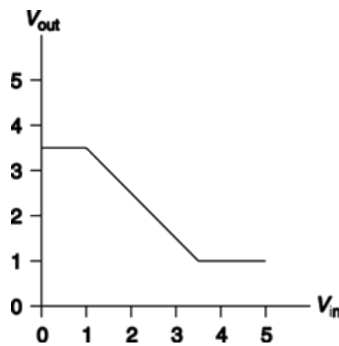


Рисунок 1.43 – Передавальна властивість

Вправа 1.80. Чи можна призначити логічні рівні так, щоб пристрій із передавальною властивістю (яку подано на рис. 1.44) міг слугувати буфером? Якщо так, то якими є вхідні та вихідні низькі та високі рівні (*VIL, VOL, VIH* і *VOH*) та рівні шуму (*NML* та *NMH*)? Якщо це не так, то поясніть чому.

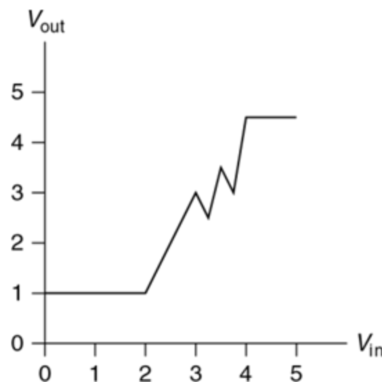


Рисунок 1.44 – Передавальна властивість

Вправа 1.81. Бен Бітдідл вигадав схему з передавальною властивістю (яку показано на рис. 1.45), щоб використовувати її як буфер. Чи працюватиме ця схема? Так чи ні й чому? Бітдідл стверджує, що вона сумісна з низьковольтними КМОН- та НТТЛ-структурами. Чи буфер Бена може коректно отримувати вхідні сигнали від цих логічних структур? Чи може її вихід керувати цими логічними структурами? Поясніть.

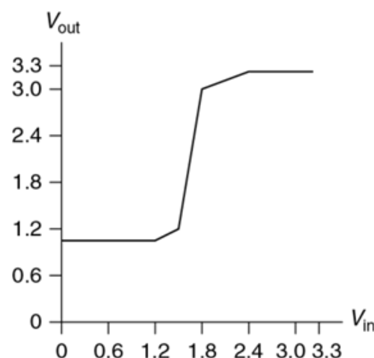


Рисунок 1.45 – Передавальна властивість буфера Бена

Вправа 1.82. Під час прогулянки темною алеєю Бен Бітдідл побачив вентиль із двома входами та передавальною функцією (див. рис. 1.46). Входи позначені як *A* та *B*, а вихідний сигнал – *Y*.

a) Якого типу логічний вентиль побачив Бен?

b) Якими є приблизні значення високого та низького логічних рівнів?

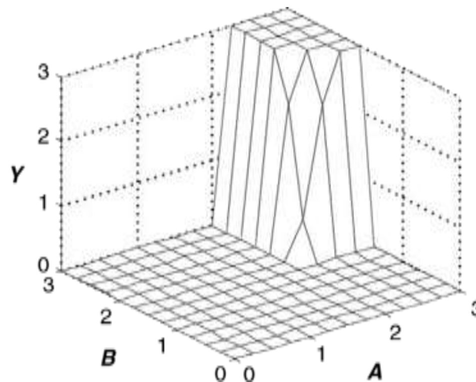


Рисунок 1.46 – Передавальні властивості вентиля з двома входами

Вправа 1.83. Повторіть вправу 1.82 для рис. 1.47.

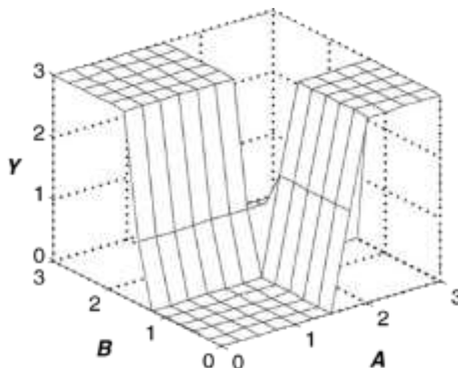


Рисунок 1.47 – Передавальні властивості вентиля з двома входами

Вправа 1.84. Накресліть схему лише на рівні транзисторів наведених нижче КМОН-вентилів. Використовуйте мінімальну кількість транзисторів.

a) Вентиль І-НІ з чотирма входами.

b) Вентиль інвертований АБО-І з трьома входами (див. вправу 1.75).

c) Вентиль І-АБО з трьома входами (див. вправу 1.74).

Вправа 1.85. Накресліть схему лише на рівні транзисторів поданих нижче КМОН-вентилів. Застосовуйте мінімальну кількість транзисторів.

a) Вентиль АБО-НІ з трьома входами.

b) Вентиль І з трьома входами.

c) Вентиль АБО з двома входами.

Вправа 1.86. Вентиль меншості видає значення ІСТИНА тоді й лише тоді, коли менше ніж половина його входів мають значення ІСТИНА.

В іншому разі він видає значення НЕПРАВДА. Накресліть схему лише на рівні транзисторів для КМОН-вентиля меншини. Використовуйте мінімальну кількість транзисторів.

Вправа 1.87. Запишіть таблицю істинності функції вентиля, який зображено на рис. 1.48. Необхідно, щоб таблиця мала два входи: *A* та *B*. Як називається ця функція?

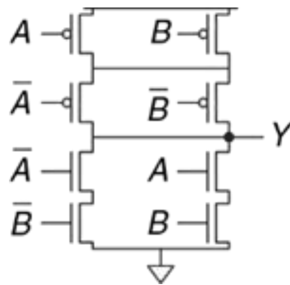


Рисунок 1.48 – Таємнича схема

Вправа 1.88. Напишіть таблицю істинності функції вентиля на рис. 1.49. Необхідно, щоб таблиця мала три входи: *A, B* та *C*.

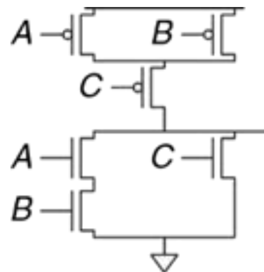


Рисунок 1.49 – Таємнича схема

Вправа 1.89. Реалізуйте подані нижче вентилі з трьома входами, використовуючи лише псевдо-*n*-МОН-логічні вентилі. Необхідно забезпечити мінімальну кількість транзисторів.

a) Вентиль АБО-НІ.

b) Вентиль І-НІ.

c) Вентиль І.

Вправа 1.90. Резисторно-транзисторна логіка (РТЛ) застосовує *n*-МОН-транзистори для видачі значення НИЗЬКИЙ (*LOW*) та резистор з малим опором для видачі значення ВИСОКИЙ (*HIGH*), коли жоден із шляхів до заземлення не активний. Вентиль НІ, побудований за допомогою РТЛ, зображений

на рис. 1.50. Накресліть схему РТЛ-вентиля АБО-НІ з трьома входами. Використовуйте мінімальну кількість транзисторів.

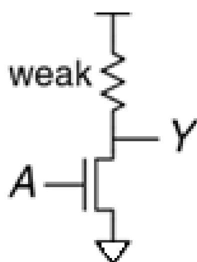


Рисунок 1.50 – Вентиль НІ

ЗАПИТАННЯ ДЛЯ СПІВБЕСІДИ

Ці питання часто ставлять
розробникам цифрових систем
на співбесіді під час їхнього працевлаштування.

Запитання 1.1. Накресліть КМОН-схему лише на рівні транзисторів для вентиля АБО-НІ з чотирма входами.

Запитання 1.2. Король отримав 64 золоті монети у вигляді податків, однак він має підстави вважати, що одна з них є підробленою. Король доручив вам виявити підроблену монету. У вас є ваги, на чашки яких можна покласти скільки завгодно монет на кожному боці. Скільки разів необхідно зробити зважування, щоб знайти легшу фальшиву монету?

Запитання 1.3. Професор, викладач, студент, що проєктує цифрові схеми, і першокурсник, чемпіон із бігу, хочуть перейти хиткий міст темної ночі. Міст має настільки поганий стан, що безпечно ним пройти можуть одночасно лише дві людини. У нашої групи лише один ліхтарик, без нього йти страшно, а міст занадто довгий, щоб перекинути ліхтарик через нього. Отже, після кожного переходу хтось має перенести ліхтар назад до людей, що залишилися.

Першокурсник може перетнути міст за 1 хв. Старший студент – за 2 хв. Викладач здатний перетнути міст протягом 5 хв. Професор завжди відволікається, тому йому для цього завдання потрібно 10 хв.

Як організувати перехід, щоб усі перейшли міст за найкоротший час?

2 ПРОЄКТУВАННЯ КОМБІНАЦІЙНОЇ ЛОГІКИ

2.1 Вступ

У цифровій електроніці під схемою розуміють електричний ланцюг, що обробляє дискретні сигнали. Таку схему можна подати як «чорну скриньку», як зображено на рис. 2.1. Схема має:

- один або більше дискретних входів;
- один або більше дискретних виходів;
- функціональну специфікацію (*functional specification*), що описує взаємозв'язок між входами та виходами;
- часову специфікацію (*timing specification*), що описує затримку між зміною сигналів на вході та відгуком вихідних сигналів.

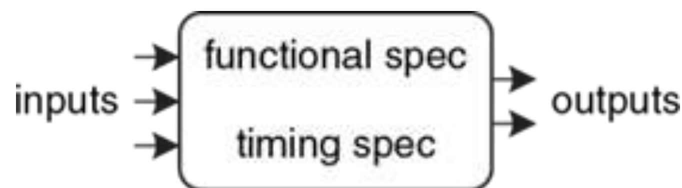


Рисунок 2.1 – Схема як «чорна скринька»
із входами, виходами та специфікаціями

Якщо заглянути всередину такої «чорної скриньки», побачимо, що схеми містять вузли (*nodes*) та елементи [1].

Елемент також є схемою з входами, виходами та специфікацією. З'єднання – це провідник, напруга в якому відповідає дискретній змінній. З'єднання поділяються на входи, виходи та внутрішні з'єднання. Входи отримують сигнали ззовні. Виходи надсилають сигнали в зовнішній світ. З'єднання, що не є входами чи виходами, називаються внутрішніми. На рис. 2.2 зображено схему з трьома елементами *E*1, *E*2 та *E*3 та із шістьма сполуками. З'єднання *A, B* та *C* – входи, *Y* та *Z* – виходи, а *n*1 – внутрішнє з'єднання між *E*1 та *E*3.

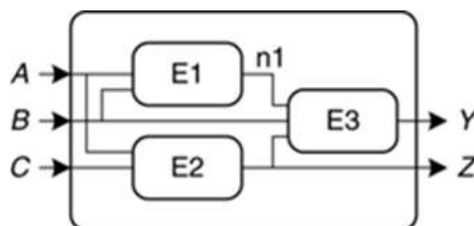


Рисунок 2.2 – Елементи та з'єднання

Цифрові схеми поділяються на комбінаційні (*combinational*) і послідовні (*sequential*). Виходи комбінаційних схем залежать тільки від поточних значень на входах; іншими словами, такі схеми комбінують поточні значення вхідних сигналів для обчислення значення на виході. Наприклад, логічний елемент – це комбінаційна схема. Виходи послідовних схем залежать і від поточних, і від попередніх значень на входах, тобто від послідовності зміни вхідних сигналів. У комбінаційних схем, на відміну послідовних, пам'ять відсутня. Цей розділ присвячений комбінаційним схемам, а в розділі 3 розглянемо послідовні схеми.

Функціональна специфікація комбінаційної схеми описує залежність значень на виходах від поточних вхідних значень. Часова специфікація комбінаційної схеми містить нижнє та верхнє граничне значення затримки сигналу на шляху від входу до виходу. У цьому розділі спочатку розглянемо функціональну специфікацію, а потім повернемося до часової.

На рис. 2.3 зображена комбінаційна схема з двома входами та одним виходом. Входи *A* і *B* розташовані ліворуч, праворуч зображено вихід *Y*. Символ прямокутника означає, що цей елемент реалізовано з використанням тільки комбінаційної логіки. У цьому прикладі функція *F* визначена як АБО: $Y = F(A, B) = A + B$.

$$Y = F(A, B) = A + B$$

Рисунок 2.3 – Комбінаційна логічна схема

Інакше кажучи, ми говоримо, що вихід *Y* – це функція двох входів: *A* і *B*, саме *Y* = *A* АБО *B*. На рис. 2.4 продемонстровано два можливі способи побудови комбінаційної логічної схеми (див. рис. 2.3). Як неодноразово зможемо побачити в цьому посібнику, часто існує безліч способів реалізації однієї й тієї самої функції. Ви самі обираєте, як реалізувати необхідну функцію з огляду на «будівельні блоки», що є в розпорядженні, а також ваших проєктних обмежень. Ці обмеження часто передбачають зайняту на кристалі мікросхемою площу, швидкість роботи, споживану потужність і час розроблення.

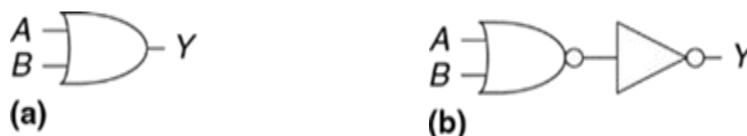


Рисунок 2.4 – Два варіанти схеми АБО

На рис. 2.5 зображено комбінаційну схему з декількома виходами. Ця схема називається повним суматором. Ми повернемося до неї в п. 5.2.1. Два рівняння визначають значення на виходах *S* і *Cout* як функції вхідних сигналів *A, B* і *Cin*.

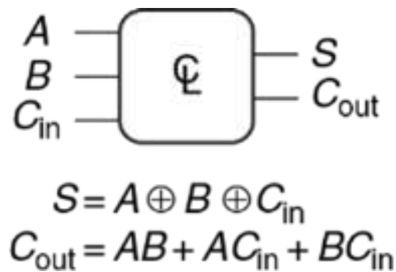


Рисунок 2.5 – Багатовихідна комбінаційна схема

Для спрощення креслеників часто використовуємо перекреслену косу лінію та число поряд з нею для позначення шини (*bus*), тобто групи сигналів. Число вказує, скільки сигналів у шині (воно зазвичай називається шириною шини). Наприклад, на рис. 2.6, *a* продемонстровано блок комбінаційної логіки з трьома входами та двома виходами. Якщо кількість розрядів не має значення або очевидна з контексту, то коса лінія може бути поряд.

На рис. 2.6, *b* показано два блоки комбінаційної логіки з довільною кількістю виходів одного блоку, які є входами іншого блоку.

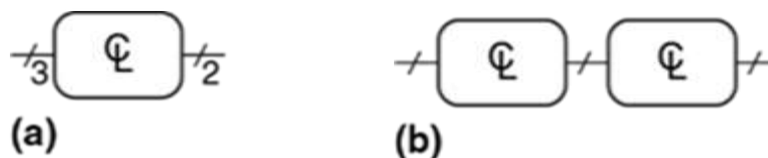


Рисунок 2.6 – Познаки шин на схемах

Правила комбінаційної композиції говорять, як ми можемо побудувати велику комбінаційну схему з дрібніших комбінаційних елементів. Схема є комбінаційною, якщо містить з'єднані між собою елементи й виконані такі умови:

● кожен елемент схеми сам є комбінаційним;

● кожне з'єднання схеми є або входом, або приєднано до одного-єдиного виходу іншого елемента схеми;

● схема не містить циклічних шляхів: кожен шлях у схемі проходить крізь будь-яке з'єднання не більше ніж один раз.

Приклад 2.1.

Комбінаційні схеми

Які зі схем на рис. 2.7 відповідно до правил комбінаційної композиції є комбінаційними?

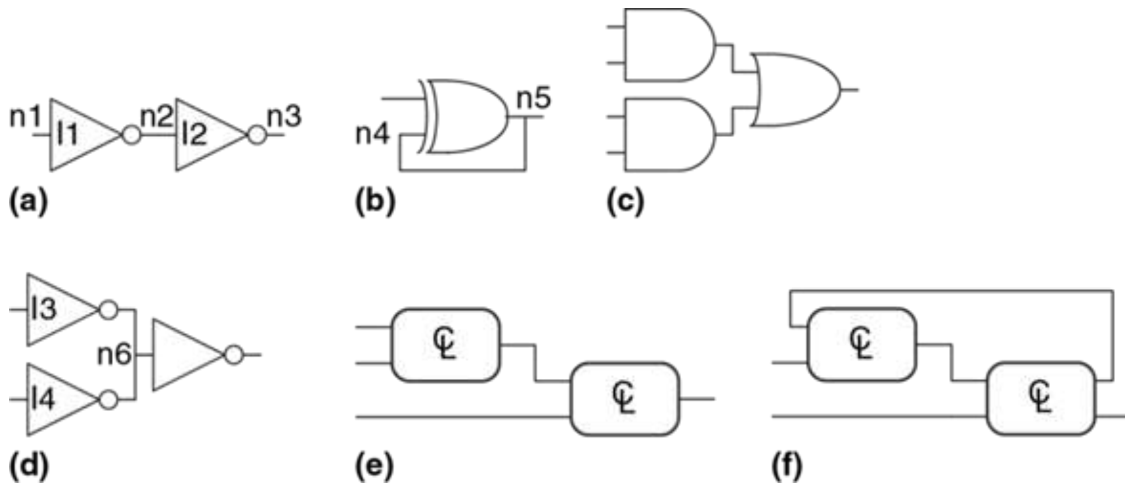


Рисунок 2.7 – Приклади схем

Виконання. Схема (*a*) – комбінаційна. Вона містить два комбінаційні елементи (інвертори I1 та I2) та має три сполуки: *n*1, *n*2 та *n*3. З'єднання *n*1 – вхід схеми та вхід для I1; *n*2 – внутрішнє з'єднання, що є виходом I1 і входом I2; *n*3 – вихід схеми та вихід I2. Схема (*b*) не комбінаційна, оскільки в ній є циклічний шлях: вихід елемента «що виключає АБО» під'єднано до одного з його власних входів, тобто циклічний шлях, починаючи в *n*4, проходить крізь «що виключає АБО» до *n*5, який веде назад до *n*4.

Схема (*c*) – комбінаційна, а (*d*) – не комбінаційна, оскільки з'єднання *n*6 під'днано до виходів двох елементів (I3 та I4). Схема (*e*) – комбінаційна, оскільки містить дві комбінаційні схеми, що з'єднані між собою та утворюють більшу комбінаційну схему. Схема (*f*) не відповідає правилам комбінаційної композиції, оскільки в ній є циклічний шлях крізь два елементи. Залежно від функцій цих елементів схема може бути, а може й не бути комбінаційною.

Великі схеми, такі як мікропроцесори, можуть бути дуже складними, тому застосовуватимемо принципи, описані в розділі 1, щоб упоратися зі складністю. Розгляд схеми як «чорної скриньки» з ретельно визначеними інтерфейсом та функцією є впровадженням принципів абстракції та модульності. Побудова схеми з більш дрібних елементів є використанням ієрархічного підходу до розробки. Правила комбінаційної композиції є сутністю застосування дисципліни.

Функціональна специфікація комбінаційної схеми зазвичай визначається як таблиці істинності чи булеві рівняння. У наступних розділах описано, як вивести ці рівняння з будь-якої таблиці істинності та як застосовувати булеву алгебру й карти Карно для спрощення рівнянь. Розглянемо, як реалізовувати ці рівняння із використанням логічних елементів і як аналізувати швидкість роботи таких схем.

2.2 Булеві рівняння

Булеві рівняння використовують змінні, що мають значення ІСТИНА або НЕПРАВДА, тому вони ідеально підходять для опису цифрової логіки. У цьому підрозділі спочатку подамо термінологію, що часто застосовується в булевих рівняннях, а потім покажемо, як записати булеві рівняння для будь-якої логічної функції за таблицею істинності.

2.2.1 Термінологія

Доповнення (*complement*) змінної *A* – це її заперечення $\overline{\mathrm{A}}$. Змінна або її доповнення називаються літералом. Наприклад, *A,* $\overline{\mathrm{A}}$*,* *B* і $\overline{\mathrm{B}}$ – літерали. Називатимемо *A* прямою формою змінної, а $\overline{\mathrm{A}}$ – комплементарною формою. Пряма форма не має на увазі, що значення *A* дорівнює ІСТИНІ, а свідчить лише про те, що в *A* відсутня риска зверху.

Операція І над одним або декількома літералами називається кон'юнкцією, твором (*product*), або імплікантою. $\overline{\mathrm{A}}\mathrm{B}$, $\mathrm{A}\overline{\mathrm{B}}\ \overline{\mathrm{C}}$ і B – імпліканти для функції трьох змінних. Мінтерм (*minterm*; елементарна кон'юнктивна форма) – це множення, що містить усі входи функції. $\mathrm{A}\overline{\mathrm{B}}\ \overline{\mathrm{C}}$ – це мінтерм для функції трьох змінних A, B і C, а $\overline{\mathrm{A}}\mathrm{B}$ не є мінтермом, оскільки не має *C*. Аналогічно, операція АБО над одним або більше літералами називається диз'юнкцією, або сумою. Макстерм (*maxterm*; елементарна диз'юнктивна форма) – це сума всіх входів функції. $A + \overline{\mathrm{B}} + C$ є макстермом функції трьох змінних *A, B* і *C*.

Порядок операцій важливий під час аналізу булевих рівнянь.

У булевих рівняннях найбільший пріоритет має операція НІ, потім іде І, потім АБО. Як і в звичайних рівняннях, обчислюються до обчислення сум. Отже, правильно рівняння читається як *Y* = *A* АБО (*B* І *C*). Рівняння (2.1) – ще один приклад, який показує порядок операцій.

$$\overline{\mathrm{A}}\,\mathrm{B} + \mathrm{BC}\overline{D} = ((\overline{\mathrm{A}}\,\mathrm{B}) + (\mathrm{BC}\,(\overline{D}))). \qquad (2.1)$$

2.2.2 Диз'юнктивна форма

Таблиця істинності функції N змінних містить $2N$ рядків, по одному для кожної можливої комбінації значень входів. Кожному рядку таблиці істинності відповідає мінтерм, що має значення ІСТИНА для цього рядка. На рис. 2.8 подано таблицю істинності функції двох змінних A і B. У кожному рядку подано відповідний йому інтерм. Наприклад, мінтерм для першого рядка – це $\overline{A} * \overline{B}$, оскільки A B має значення ІСТИНА тоді, коли $A = 0$ і $B = 0$. Мінтерми нумерують починаючи з 0; перший рядок відповідає мінтерму 0 (m_0), наступний рядок – мінтерму 1 (m_1) і т. д.

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	0	$A\ B$	m_3

Рисунок 2.8 – Таблиця істинності та мінтерми

Можна написати рівняння для будь-якої таблиці істинності за допомогою підсумовування всіх тих мінтермів, для яких вихід Y має значення ІСТИНА. Наприклад, на рис. 2.8 є лише один рядок (мінтерм), для якого вихід Y має значення ІСТИНА (обведено). Отже, $Y = \overline{A}\ B$. На рис. 2.9 подано таблицю, у якій вихід має значення ІСТИНА для декількох рядків. Підсумовування зазначених мінтермів дає $Y = \overline{A}\ B + AB$.

Така сума мінтермів називається досконалою диз'юнктивною нормальною формою функції (*sum-of-products canonical form*). Вона є сумою (операцій АБО) добутків (операцій І), що утворюють мінтерми. Хоча існує чимало способів записати одну й ту саму функцію, зокрема $Y = \overline{A}\ \ B + AB$, будемо записувати мінтерми за тим самим порядком, як у таблиці істинності, щоб завжди отримувати один і той самий булевий вираз для однієї і тієї самої таблиці істинності. Досконала диз'юнктивна нормальна форма може бути записана за допомогою символу суми Σ. У разі використання цієї позначки функція на рис. 2.9 матиме такий вигляд:

$$F(A, B) = \Sigma(m1, m3)$$

або (2.2)

$$F(A, B) = \Sigma(1, 3).$$

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

Рисунок 2.9 – Таблиця істинності з кількома мінтермами, рівними ІСТИНІ

Приклад 2.2.

Диз'юнктивна форма

У Бена Бітдідла планується пікнік. Він не зрадіє, якщо йтиме дощ чи з'являться мурахи. Побудуйте схему, в якій вихід набуватиме значення ІСТИНА тільки в тому разі, якщо Бену пікнік сподобається.

Виконання. Спочатку визначимо входи та виходи. Входами будуть змінні A і R, що означають мурахи (*ants*) та дощ (*rain*). A набуває значення ІСТИНА, коли мурахи є, і НЕПРАВДА, коли мурах немає. Так само R має значення ІСТИНА, коли йде дощ, і НЕПРАВДА, коли світить сонце. Вихід E (*enjoyment*, радість) показує настрій Бена. E має значення ІСТИНА, коли чоловік радіє пікніку, і НЕПРАВДА, коли він страждає. На рис. 2.10 подано таблицю істинності вражень Бена від пікніка.

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Рисунок 2.10 – Таблиця істинності Бена

Використовуючи диз'юнктивну форму, запишемо рівняння так: $E = \overline{\mathrm{A}} * \overline{R}$ або $E = \Sigma$ (0). Можемо реалізувати відповідну схему за допомогою двох інверторів і двовходового елемента І (див. рис. 2.11, *a*). Ви могли помітити, що ця таблиця є такою самою, як і таблиця для функції АБО-НІ, яку розглянуто в п. 1.5.5: $E = A$ АБО-НІ $R = \overline{A + R}$.

На рис. 2.11, *b* продемонстровано реалізацію на базі елемента АБО-НІ. У підрозділі 2.3 покажемо, що вирази $\overline{\mathrm{A}} * \overline{R}$ та $\overline{A + R}$ еквівалентні.

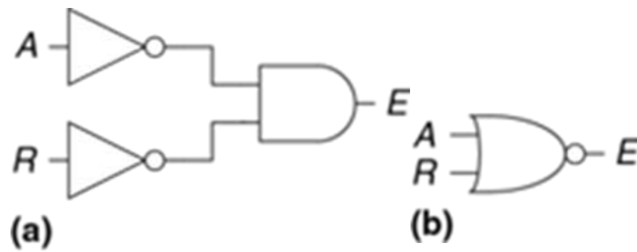


Рисунок 2.11 – Схема Бена

Досконала диз’юнктивна нормальна форма дає змогу записати булеве рівняння для будь-якої таблиці істинності з будь-якою кількістю змінних. На рис. 2.12 подано довільну таблицю істинності для тривходового елемента. Досконала диз’юнктивна нормальна форма відповідної логічної функції має такий вигляд:

$$Y = \overline{A}*\overline{B}*\overline{C} + A\overline{B}*\overline{C} + A\overline{B}C$$

або (2.3)

$$Y = \Sigma(0, 4, 5).$$

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Рисунок 2.12 – Довільна таблиця істинності з трьома входами

На жаль, досконала диз’юнктивна нормальна форма не завжди дає змогу отримати просте рівняння. У підрозділі 2.3 покажемо, як записати ту саму функцію із використанням меншої кількості членів рівняння.

2.2.3 Кон’юнктивна форма

Альтернативний спосіб вираження булевих функцій – це досконала кон’юнктивна нормальна форма (*products-of-sum forms*). Кожен рядок таблиці істинності відповідає макстерму, що має значення *FALSE* для цього рядка. Наприклад, макстерм для першого рядка для двовходової таблиці істинності – це $(A + B)$, оскільки $(A + B)$ має значення *FALSE*, коли $A = 0$ і $B = 0$. Для будь-якої схеми, задану таблицею істинності, можемо записати її булеве рівняння як логічне І всіх макстермів, для яких вихід має значення *FALSE*.

Досконала кон'юнктивна нормальна форма також може бути записана з використанням символу П.

Приклад 2.3.

Кон'юнктивна форма

Запишіть рівняння в досконалій кон'юнктивній нормальній формі для таблиці істинності на рис. 2.13.

Виконання. Таблиця істинності має два рядки, у яких вихід має значення *FALSE*. Отже, функція може бути записана в кон'юнктивній формі так: $Y = (A + B)\ (\overline{\mathrm{A}} + B)$. Також функцію можна записувати як $Y = \Pi(\mathrm{M}_0, \mathrm{M}_2,)$ або $Y = \Pi(0, 2)$. Перший макстерм $(A + B)$ гарантує, що $Y = 0$ для $A = 0$ і $B = 0$, оскільки логічне І будь-якого значення й нуля дає нуль. Так само другий макстерм $((\overline{\mathrm{A}} + B)$ гарантує, що $Y = 0$ для комбінації $A = 1$ та $B = 0$. На рис. 2.13 подано таку саму таблицю істинності, як і на рис. 2.9, щоб продемонструвати, що одна й та сама функція може бути записана більш ніж одним способом.

A	B	Y	maxterm	maxterm name
0	0	0	$\mathrm{A} + \mathrm{B}$	M_0
0	1	1	$\mathrm{A} + \overline{\mathrm{B}}$	M_1
1	0	0	$\overline{\mathrm{A}} + \mathrm{B}$	M_2
1	1	1	$\overline{\mathrm{A}} + \overline{\mathrm{B}}$	M_3

Рисунок 2.13 – Таблиця істинності з макстермами

Аналогічно булеве рівняння для пікніка Бена (рис. 2.10) може бути записано в досконалій кон'юнктивній нормальній формі, якщо обвести три рядки з нулями для того, щоб отримати

$$\mathrm{E} = (\mathrm{A} + (\overline{R})\ ((\overline{\mathrm{A}} + \mathrm{R})\ ((\overline{\mathrm{A}} + (\overline{R}) \quad \text{або} \quad \mathrm{E} = (1, 2, 3).$$

Це не такий гарний запис, як диз'юнктивне рівняння, $E = \overline{\mathrm{A}} * \overline{R}$, але ці два рівняння логічно еквівалентні. Диз'юнктивна форма дає більш коротке рівняння, коли вихід має значення ІСТИНА лише в кількох рядках таблиці істинності; кон'юнктивна ж форма простіша, коли вихід має значення *FALSE* лише в кількох рядках таблиці істинності.

2.3 Булева алгебра

У попередньому розділі ми вивчили, як записувати булеві вирази за наявності таблиці істинності. Однак вираз, отриманий у такий спосіб,

не обов’язково приводить до найпростішого набору логічних елементів. Ви можете застосовувати булеву алгебру для спрощення булевих рівнянь так само, як використовуєте алгебру для спрощення математичних рівнянь. Правила булевої алгебри дуже схожі на правила звичайної алгебри, але в деяких випадках вони простіші, тому що змінні мають тільки два можливі значення: 0 або 1.

Булева алгебра основана на наборі аксіом, які вважаємо правильними. Аксіоми є недоведеними, тому що визначення не може бути доведено. За допомогою цих аксіом доводимо всі теореми булевої алгебри.

Ці теореми мають величезну практичну значущість, тому що за їх допомогою вчимося спрощувати логічні рівняння, щоб отримувати більш дешеві та компактні схеми. Аксіоми та теореми булевої алгебри підпорядковуються принципу двоїстості. Якщо взаємно замінити символи 0 і 1, а також взаємно замінити оператори • (І) та + (АБО), то булевий вираз залишиться правильним. Ми використовуємо символ «штрих» (‘) для позначення двоїстого виразу.

2.3.1 Аксіоми

У табл. 2.1 наведено аксіоми булевої алгебри. Ці п’ять аксіом і подвійні їм аксіоми визначають булеви змінні та значення операторів НІ, І, АБО. Аксіома А1 показує, що булева змінна *B* має значення 0, якщо вона не має значення 1.

Подвійний вираз для аксіоми А1 стверджує, що змінна набуває значення 1, якщо вона не має значення 0. Разом аксіоми А1 і А1’ показують, що ми працюємо в булевому, тобто бінарному, полі, що містить значення нулів і одиниць. Аксіоми А2 і А2’ визначають операцію НІ. Аксіоми з А3 до А5 визначають операцію І, а їх двоїсті аксіоми (А3’–А5’) – операцію АБО.

Таблиця 2.1 – Аксіоми булевої алгебри

	Аксіома		**Подвійна аксіома**	**Назва**
A1	$B = 0$, якщо $B \neq 1$	A1′	$B = 1$, якщо $B \neq 0$	Бінарне поле
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	НІ
A3	0 • 0 = 0	A3′	1 + 1 = 1	І/АБО
A4	1 • 1 = 1	A4′	0 + 0 = 0	І/АБО
A5	0 • 1 = 1 • 0 = 0	A5′	1 + 0 = 0 + 1 = 1	І/АБО

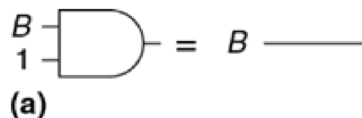
2.3.2 Теореми однієї змінної

Теореми з Т1 до Т5 у табл. 2.2 описують, як спростити рівняння, що містять одну змінну.

Теорема ідентичності Т1 стверджує, що для будь-якої булевої змінної виконано B І 1 = B. Подвійна їй теорема констатує, що B АБО 0 = B. В апаратурі, як продемонстровано на рис. 2.14, Т1 означає, що якщо рівень сигналу на одному з входів двовходового елемента завжди дорівнює 1, то можемо вилучити цей елемент і замінити його дротом, що з’єднує вихід цього елемента із входом, значення якого може змінюватися. Так само теорема Т1’ стверджує, що якщо один вхід двовходового елемента АБО завжди дорівнює 0, ми можемо замінити цей елемент на дріт, з’єднаний із входом B. Як правило, елементи мають певну вартість, енергоспоживання та затримку проходження сигналу, тому заміна елемента на дріт є доцільною.

Таблиця 2.2 – Булеві теореми для однієї змінної

	Теорема	**Подвійна теорема**		**Назва**
Т1	$B \bullet 1 = B$	Т1′	$B + 0 = B$	ідентичність
Т2	$B \bullet 0 = 0$	Т2′	$B + 1 = 1$	нульовий елемент
Т3	$B \bullet B = B$	Т3′	$B + B = B$	ідемопотентність
Т4		$\overline{\overline{B}} = B$		інволюція
Т5	$B \bullet \overline{B} = 0$	Т5′	$B + \overline{B} = 1$	додатковість



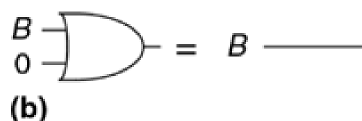


Рисунок 2.14 – Теорема ідентичності в апаратурі: (a) Т1, (b) Т1

Теорема про нульовий елемент Т2 стверджує, що B І 0 завжди дорівнює 0. Отже, 0 називають нульовим елементом для операції І, оскільки він обнуляє ефект будь-якого іншого входу. Подвійна їй теорема свідчить, що $B + \overline{B} = 1$. Отже, 1 – це нульовий елемент для операції АБО. В апаратурі (як показано на рис. 2.15) якщо один вхід елемента І дорівнює 0, можемо замінити елемент І дротом, під’єднаним до низького логічного рівня (0). Так само, якщо один

із входів елемента АБО дорівнює 1, можемо замінити елемент АБО на дріт, під'єднаний до високого логічного рівня (1).

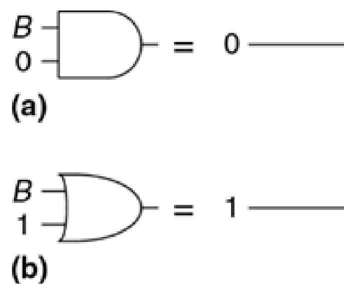


Рисунок 2.15 – Теорема про нульовий елемент в апаратурі: (a) T2, (b) T2'

Теорема про ідемпотентність T3 стверджує, що операція логічного І двох рівних одна одній змінних має значення, що дорівнює цій змінній. Аналогічне твердження правильне для операції АБО з двома однаковими значеннями на входах. Назва теореми походить від латинських слів: *idem* – той самий, такий самий і *potent* – сила. Операції повертають ті самі значення, які ви подаєте їм на вхід. На рис. 2.16 продемонстровано, як ідемопотентність дає змогу замінити елемент схеми на дріт.

Теорема про інволюцію T4 – це спосіб опису того, що подвійне заперечення змінної дає її вихідне значення. Два послідовно увімкнені інвертори логічно скасовують один одного, тобто вони еквівалентні дроту (див. рис. 2.17). Подвійною їй теоремою є вона сама.

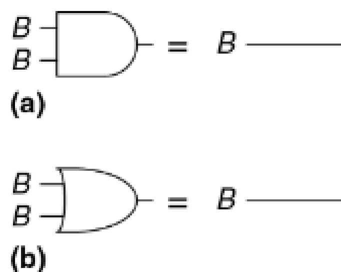


Рисунок 2.16 – Теорема про ідемпотентність в апаратурі: (a) T3, (b) T3

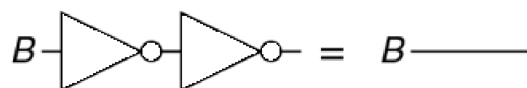


Рисунок 2.17 – Теорема про інволюцію в апаратурі: T4'

Теорема про додатковість T5 (рис. 2.18) стверджує, що операція І над змінною та її інверсним значенням дає 0 (бо одна з них завжди дорівнюватиме нулю). І відповідно до принципу двоїстості операція АБО над змінною та її інверсним значенням завжди дає 1 (оскільки одна з них завжди дорівнюватиме 1).

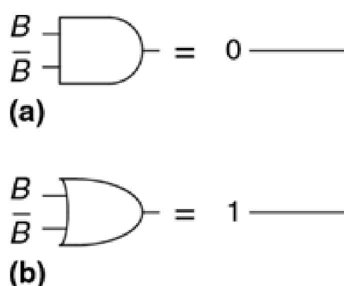


Рисунок 2.18 – Теорема інволюції в апаратурі: (a) T5, (b) T5

2.3.3 Теореми з кількома змінними

Теореми T6–T12 (див. табл. 2.3) описують, як спростити рівняння, у яких понад одна булева змінна.

Таблиця 2.3 – Булеві теореми для кількох змінних

	Теорема		Подвійна теорема	Назва
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	комутативність
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	асоціативність
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	дистрибутивність
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	поглинання
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	склеювання
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) =$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D) =$ $= (B + C) \bullet (\overline{B} + D)$	узгодженість
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots)$	теорема де Моргана

Теореми T6 про комутативність і T7 про асоціативність працюють так само, як і в традиційній алгебрі. Відповідно до принципу комутативності порядок входів для функцій І чи АБО не впливає на значення виходу. За принципом асоціативності будь-яке групування входів не впливає на значення виходу.

Теорема про дистрибутивність T8 є такою самою, як і в традиційній алгебрі, а подвійна їй теорема T8’ – ні. Відповідно до теореми T8 оператор І дистрибутивний щодо операції АБО. T8’ стверджує, що оператор АБО дистрибутивний щодо операції І. У традиційній алгебрі оператор множення дистрибутивний щодо операції додавання, але не навпаки, тобто

$$(B + C) \times (B + D) \neq B + (C \times D).$$

Теореми поглинання, склеювання та узгодженості T9–T11 дають змогу вилучати зайві змінні. Якщо ви трохи подумаєте, то зможете переконатись, що ці теореми справедливі.

Теорема де Моргана Т12 є особливо потужним інструментом у розробленні цифрових пристроїв. Вона пояснює, що доповнення результату множення всіх термів дорівнює сумі доповнень кожного терму. Так само доповнення суми всіх термів дорівнює результату множення доповнень кожного терму.

За теорією де Моргана елемент І-НІ еквівалентний елементу АБО з інвертованими входами. Аналогічно АБО-НІ еквівалентний елементу І з інвертованими входами. На рис. 2.19 зображено еквівалентні елементи І-НІ та АБО-НІ (відповідно до теорії де Моргана). Вони логічно еквівалентні та взаємозамінні.

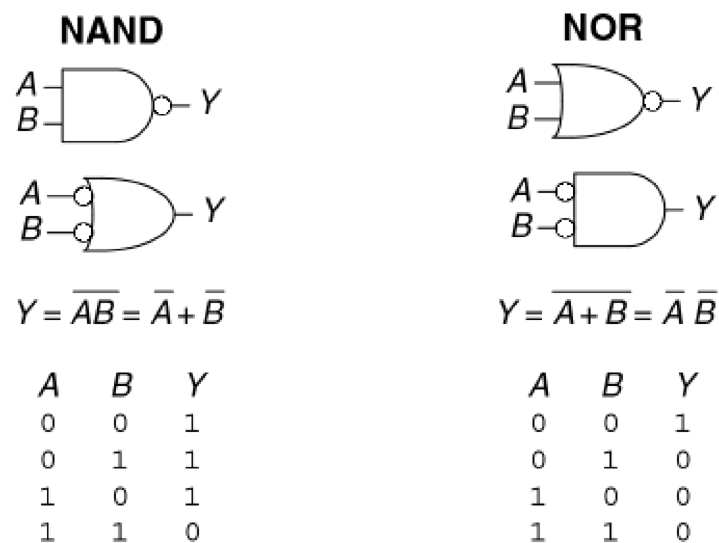


Рисунок 2.19 – Еквівалентні елементи за де Морганом

Кружок на графічній схемі елементів позначає заперечення (інверсію). Інтуїтивно можете уявити: якщо «вдавити» цей кружок з одного боку логічного елемента, то він «вискочить» на інший, у цьому разі тип елемента зміниться з І на АБО (і навпаки). Це називається «переміщення інверсії». Наприклад, елемент І-НІ (рис. 2.19) містить елемент І із запереченням на виході. Переміщення інверсії наліво приводить до отримання елемента АБО з двома запереченнями на входах. Наведемо базові правила для переміщення інверсії.

● Переміщення інверсії назад (від виходу) або вперед (від входів) змінює тип елемента з І на АБО й навпаки.

● Переміщення інверсії з виходу назад до входів приводить до того, що на всіх входах з’являється інверсія.

● Переміщення інверсії з усіх входів елемента до виходу приводить до появи інверсії на виході.

У п. 2.5.2 принцип руху інверсії використовується для аналізу схем.

Приклад 2.4.

Отримайте кон’юнктивну форму

На рис. 2.20 подано таблицю істинності для булевої функції *Y* та її доповнення *Y’*. Унаслідок застосування теореми де Моргана отримайте нормальну кон’юнктивну форму функції *Y* з диз’юнктивної форми *Y*.

Виконання. На рис. 2.21 обведені мінтерми, що містяться у функції *Y*. Диз’юнктивна нормальна форма функції *Y* має такий вигляд:

$$\overline{Y} = \overline{A}\ \overline{B} + \overline{A}\ \mathrm{B}. \qquad (2.4)$$

Застосовуючи операцію інверсії до обох частин рівняння та двічі використовуючи теорему де Моргана, отримуємо:

$$\overline{\overline{Y}} = (\mathrm{Y}) = \overline{\overline{A}\overline{B} + \overline{A}\,\mathrm{B}} = (\overline{\overline{A}\overline{B}})(\overline{\overline{A}\mathrm{B}}) = (\mathrm{A} + \mathrm{B})(\mathrm{A} + \overline{B}). \qquad (2.5)$$

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Рисунок 2.20 – Таблиця істинності для *Y* і *Y*

A	B	Y	$\overline{Y}$	minterm
0	0	0	1	$\overline{A}\ \overline{B}$
0	1	0	1	$\overline{A}\ B$
1	0	1	0	$A\ \overline{B}$
1	1	1	0	$A\ B$

Рисунок 2.21 – Таблиця істинності для *Y* і *Y’*

2.3.4 Правда про все це

Допитливий читач може поставити запитання: як довести правильність теореми. У булевій алгебрі доказ теорем з кінцевим числом змінних є простим: необхідно показати, що теорема правильна для всіх можливих значень змінних. Цей метод називається досконалою індукцією та може бути виконаний з використанням таблиці істинності.

Приклад 2.5.

Доказ теореми узгодженості методом повного перебору

Доведіть теорему узгодженості T11 з табл. 2.3.

Виконання. Перевірте обидві частини рівняння всіх восьми комбінацій змінних *B, C* і *D*. Таблиця істинності на рис. 2.22 ілюструє всі ці комбінації.

Оскільки рівність $BC + \overline{B}D + CD = BC + \overline{B}D$ правильна для всіх випадків, теорему доведено.

B	C	D	$BC+\overline{B}D+CD$	$BC+\overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Рисунок 2.22 – Таблиця істинності, що доводить Т11

2.3.5 Спрощення рівнянь

Теореми булевої алгебри допомагають спрощувати булеві рівняння. Наприклад, візьмемо диз'юнктивну форму виразу з таблиці істинності (рис. 2.9): $Y = \overline{A} * \overline{B} + A\overline{B}$. Відповідно до теореми Т10 рівняння можна спростити до $Y = \overline{B}$. Можливо, це очевидно, якщо подивитися на таблицю істинності. В іншому разі може знадобитися кілька кроків для спрощення більш складних рівнянь.

Основний принцип спрощення диз'юнктивних рівнянь – це комбінування термів із використанням відношення $PA + P\overline{A} = P$, де P може бути будь-якою імплікантою. Наскільки може бути спрощено рівняння? За визначенням рівняння диз'юнктивної форми є мінімізованим, якщо воно містить мінімально можливу кількість імплікант. Якщо є кілька рівнянь з однаковою кількістю імплікант, мінімальним буде рівняння, у якому менше літералів.

Імпліканта називається простою (*prime implicant*), якщо не може бути поєднана з іншими імплікантами в рівнянні для того, щоб утворити нову імпліканту з меншою кількістю літералів. Усі імпліканти в мінімальному рівнянні мають бути простими. Інакше вони можуть бути об'єднані, щоб зменшити кількість літералів.

Приклад 2.6.

Мінімізація рівняння

Мінімізуйте рівняння (2.3): $\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\overline{B}\,C$.

Виконання. Починаємо з вихідного рівняння та застосовуємо булеві теореми крок за кроком, як показано в табл. 2.4.

Таблиця 2.4 – Мінімізація виразів

Крок	Вираз	Пояснення
	$\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\overline{B}\,C$	
1	$\overline{B}\,\overline{C}\,(\overline{A} + \overline{A}) + A\overline{B}\,C$	Т8: дистрибутивність
2	$\overline{B}\,\overline{C}\,(1) + A\overline{B}\,C$	Т5: додатковість
3	$\overline{B}\,\overline{C} + A\overline{B}\,C$	Т1: ідентичність

Чи спростили ми повністю рівняння на цій стадії? Пропонуємо подивитися уважно. В оригінальному рівнянні мінтерми $\overline{A}\ \overline{B}\ \overline{C}$ і $A\ \overline{B}\ \overline{C}$ розрізняються тільки змінною *A*. Тому об’єднуємо мінтерми, отримуємо $\overline{B}\ \overline{C}$. Однак, якщо подивимося на початкове рівняння, помітимо, що останні два мінтерми $A\ \overline{B}\ \overline{C}$ і $A\overline{B}\ C$ також розрізняються одним літералом (C і $\overline{C}$). Так, використовуючи той самий метод, ми могли б об’єднати ці два мінтерми й отримати мінерм $A\overline{B}$. Можна сказати, що імпліканти $\overline{B}\ \overline{C}$ і $A\overline{B}$ ділять між собою мінтерм $A\ \overline{B}\ \overline{C}$.

Отже, ми зупинилися на спрощенні лише однієї пари мінтермів чи можемо спростити обидві? Використовуючи теорему про ідемпотентність, можемо дублювати мінтерми стільки разів, скільки необхідно: $B = B + B + B + B$… Упроваджуючи зазначений принцип, ми повністю спрощуємо рівняння до його простих імплікант, $\overline{B}\ \overline{C} + A\overline{B}$, як подано в табл. 2.5.

Таблиця 2.5 – Покращена мінімізація виразів

Крок	Вираз	Пояснення
	$\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\overline{B}\,C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\overline{B}\,C + A\overline{B}\,C$	Т3: ідемопотентність
2	$\overline{B}\,\overline{C}\,(\overline{A} + \overline{A}) + A\overline{B}\,(\overline{C} + C)$	Т8: дистрибутивність
3	$\overline{B}\,\overline{C}\,(1) + A\overline{B}\,(1)$	Т5: додатковість
4	$\overline{B}\,\overline{C} + A\overline{B}$	Т1: ідентичність

Хоча це трохи не логічно, розширення імпліканти (наприклад, перетворення *AB* в $ABC + AB\overline{C}$) іноді корисне за умови мінімізації рівнянь. У такий спосіб можете повторювати один із розширених мінтермів для його поєднання з іншим мінтермом.

Ви могли помітити, що повне спрощення булевих рівнянь за допомогою теорем булевої алгебри може потребувати кількох спроб, деякі з яких будуть хибними. У підрозділі 2.7 описано методику, що дає змогу спростити процес мінімізації – карти Карно.

Навіщо ж працювати над спрощенням булевого рівняння, якщо воно залишається логічно еквівалентним? Спрощення зменшує кількість елементів, що використовуються в процесі фізичного втілення функції в апаратурі, тим самим роблячи схему меншою, дешевшою та, можливо, більш швидкою. У наступному розділі розглядається, як втілювати булеві рівняння за допомогою логічних елементів.

2.4 Від логіки до логічних елементів

Принципова схема – це зображення цифрової схеми, що показує елементи й провідники, які з’єднують їх. Наприклад, схема на рис. 2.23 демонструє можливу апаратну реалізацію логічної функції (рівняння (2.3)):

$$Y = \overline{A}*\overline{B}*\overline{C} + A\overline{B}*\overline{C} + A\overline{B}C.$$

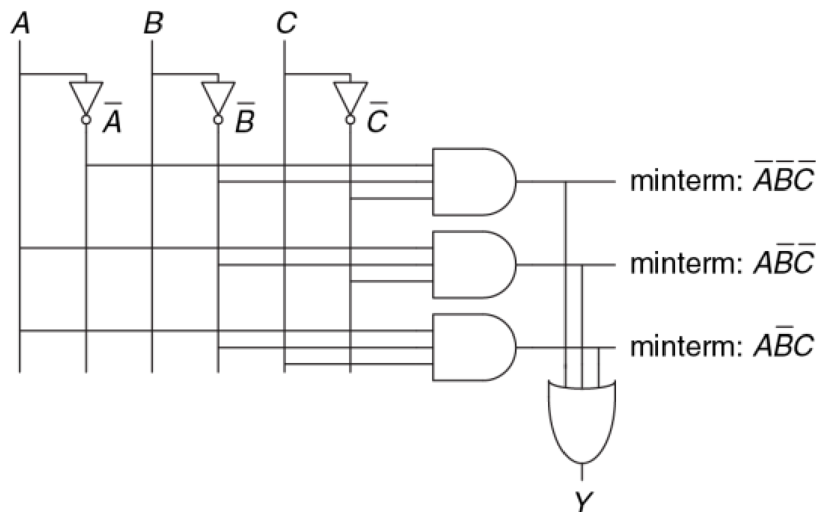


Рисунок 2.23 – Апаратна реалізація логічної функції $Y = \overline{A}*\overline{B}*\overline{C} + A\overline{B}*\overline{C} + A\overline{B}C$

Якщо подавати принципові схеми в уніфікованому вигляді, їх легше читати й налагоджувати. Здебільшого дотримуватимемося таких правил:

- входи зображують на лівій (або верхній) частині схеми;
- виходи позначають на правій (або нижній) частині схеми;
- завжди, коли це можливо, елементи необхідно зображати зліва направо;
- провідники краще зображати прямими лініями, ніж лініями з безліччю кутів (нерівні рвані лінії відволікають увагу: доводиться стежити, куди ведуть дроти, а не думати про те, що робить схема);

● провідники завжди мають з’єднуватись у вигляді літери Т;

● крапка у місці перетину провідників позначає їх з’єднання;

● провідники, що перетинаються без точки, не мають з’єднання один з одним.

Три останні правила продемонстровано на рис. 2.24.

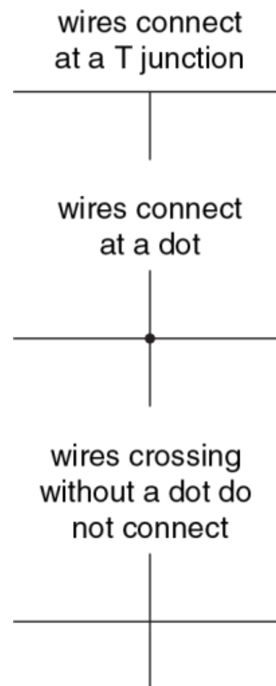


Рисунок 2.24 – Перетин провідників

Будь-яке булеве рівняння в диз’юнктивній формі може бути зображено у вигляді принципової схеми з використанням систематичного підходу (див. рис. 2.23). Спершу накресліть вертикальні провідники для входів. Розташуйте інвертори на сусідніх вертикальних лініях для отримання комплементарних входів, якщо це необхідно. Намалюйте горизонтальні лінії, що ведуть до елементів І для кожного мінтерму. Потім для кожного виходу намалюйте елемент АБО, з’єднаний з мінтермом, що відповідає цьому виходу. Такий стиль зображення називається програмованою логічною матрицею (ПЛМ, *PLA*), оскільки інвертори, елементи І та АБО систематично об’єднані в масиви. Програмовані логічні матриці розглядатимемо в підрозділі 5.6.

На рис. 2.25 продемонстровано реалізацію спрощеного рівняння, отримане за допомогою булевої алгебри в прикладі 2.6. Очевидно, що спрощена схема має значно менше апаратних елементів, ніж схема на рис. 2.23. Також її швидкодія може бути вищою, оскільки схема використовує елементи з меншою кількістю входів.

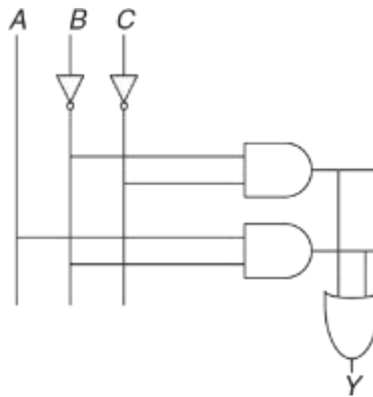


Рисунок 2.25 – Схема реалізації функції Y $=\overline{B}*\overline{C} + A\overline{B}$

Можемо навіть ще зменшувати кількість елементів (нехай хоча б на один інвертор), якщо скористаємося перевагою логічних елементів, які інвертують. Зауважте, що $\overline{B}*\overline{C}$ – це елемент І з інвертованими входами. На рис. 2.26 зображена схема, що використовує цю оптимізацію для вимкнення інвертора на вході *C*. Згадайте, що за теоремою де Моргана логічний елемент І з інвертованими входами еквівалентний елементу АБО-НІ. Залежно від технології реалізації використання найменшого числа елементів або елементів певного типу, замість інших, може бути більш вигідним. Наприклад, у технології КМОН елементи І-НІ та АБО-НІ кращі, ніж І або АБО.

Чимало схем мають кілька виходів, кожен з яких реалізує незалежні булеві функції для входів. Можемо записати окремі таблиці істинності кожного виходу, але часто зручно записати всі виходи на одну таблицю істинності й накреслити одну схему для всіх виходів.

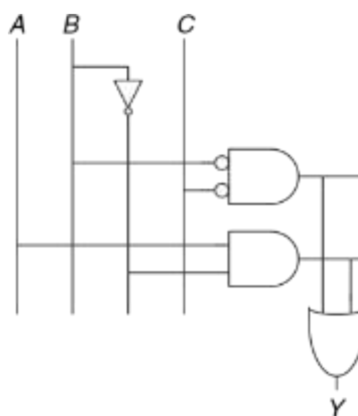


Рисунок 2.26 – Схема, що використовує менше елементів

Приклад 2.7.

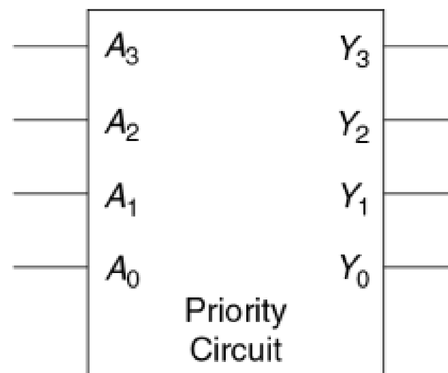
Схеми з кількома виходами

Декан, завідувач кафедри, аспірант та голова ради гуртожитку працюють в одній аудиторії. На жаль, іноді аудиторія потрібна їм одночасно,

що призводить до катастроф, як, наприклад, коли зустріч декана з літніми та шановними членами опікунської ради була запланована на той самий час, що й пивна вечірка студентів гуртожитку. Алісу Хакер було запрошено для того, щоб розробити систему резервування кімнати.

Система має чотири входи ($A3$, …, $A0$) та чотири виходи ($Y3$, …, $Y0$). Ці сигнали можуть бути записані у вигляді $A3{:}0$ і $Y3{:}0$. Кожен користувач активує свій вхід, коли запитує аудиторію наступного дня. Система активує лише один вихід, підтверджуючи, що аудиторію використовує найвища пріоритетна особа. Декан, який оплачує систему, потребує найвищого пріоритету (3). Завідувач кафедри, аспірант та голова ради гуртожитку мають пріоритети за спаданням. Запишіть таблицю істинності та булеві рівняння для цієї системи. Накресліть схему, яка виконує цю функцію.

Виконання. Ця функція називається чотиривходовою схемою пріоритету. Її кресленик і таблиця істинності зображені на рис. 2.27.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Рисунок 2.27 – Схема пріоритетів

Можна було б записати кожен вихід у диз’юнктивній формі та спростити рівняння, використовуючи булеву алгебру. Однак достатньо подивитися на функціональний опис (таблицю істинності), щоб зрозуміти, які можуть бути спрощені рівняння: *Y*3 має значення ІСТИНА завжди, коли сигнал *A*3 подається, отже *Y*3 = *A*3. *Y*2 дорівнює ІСТИНІ, якщо подано сигнал *A*2 і не подано сигналу *A*3, отже, *Y*2 = $\overline{\mathrm{A3}}$A2. *Y*1 має значення ІСТИНА, якщо подано сигнал *A*1 і на жодний з більш пріоритетних входів сигнал не подано: *Y*1 = $\overline{\mathrm{A3}}$ $\overline{\mathrm{A2}}$ A1. *Y*0 має значення ІСТИНА за умови поданого сигналу *A*0 і коли жоден з інших виходів не активовано: *Y*1 = $\overline{\mathrm{A3}}\,\overline{\mathrm{A3}}\,\overline{\mathrm{A1}}$A0. Схему зображено на рис. 2.28. Досвідчений розробник часто може реалізувати логічну схему, безпосередньо дивлячись на вихідні дані. За наявності чітко заданої специфікації, просто перетворіть слова на рівняння, а рівняння – на логічні елементи схеми.

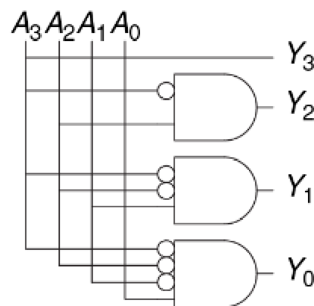


Рисунок 2.28 – Принципова схема

Зверніть увагу, якщо в схемі пріоритету подається сигнал *A*3, то виходи схеми не залежать від того, які сигнали є на інших входах. Ми використовуємо символ *X* для опису стану входів, які нам байдужі, оскільки не впливають на вихід. На рис. 2.29 продемонстровано, що таблиця істинності чотиривходової пріоритетної схеми стає набагато меншою, якщо прибрати значення входів, якими можна знехтувати. З цієї таблиці істинності можемо легко отримати булеві рівняння в диз’юнктивній формі, пропустивши входи з *X*. Значення, якими можна знехтувати, також, імовірно, виникнуть на виходах таблиці істинності, як побачимо в п. 2.7.3.

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Рисунок 2.29 – Таблиця істинності схеми пріоритетів

2.5 Багаторівнева комбінаційна логіка

Комбінаційна логіка, побудована як диз’юнкція кон’юнкцій (сума), називається дворівневою, оскільки містить літерали, з’єднані з елементами І (що утворюють перший рівень), виходи яких з’єднані з елементами АБО (що утворюють другий рівень). Розробники часто створюють схеми із значною кількістю рівнів логічних елементів. Така багаторівнева комбінаційна схема може використовувати менше логічних елементів, ніж її дворівнева реалізація. Еквівалентні перетворення за законами де Моргана та переміщення інверсії особливо корисні для аналізу та розроблення багаторівневих схем.

2.5.1 Мінімізація апаратури

Деякі логічні функції вимагають величезної кількості апаратури, якщо будувати їх із використанням дворівневої логіки. Показовий приклад – це функція ЩО ВИКЛЮЧАЄ АБО (*XOR*) кількох змінних. Наприклад, розглянемо побудову трився тривходового елемента *XOR*, застосовуючи дворівневу техніку, яку вивчали досі.

Згадаємо, що *N*-входовий *XOR* видає на вихід значення ІСТИНА, якщо непарне число вхідних операндів мають значення ІСТИНА. На рис. 2.30, *а* подано таблицю істинності тривходового елемента *XOR*. У таблиці обведені рядки, для яких значення виходу буде ІСТИНА. З таблиці істинності розуміємо форму логічного виразу, яка відповідає диз’юнкції кон’юнкцій рівняння (2.6). На жаль, цей вираз неможливо спростити в меншу кількість імплікант [1].

$$Y = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\overline{B}\,C + ABC. \quad (2.6)$$

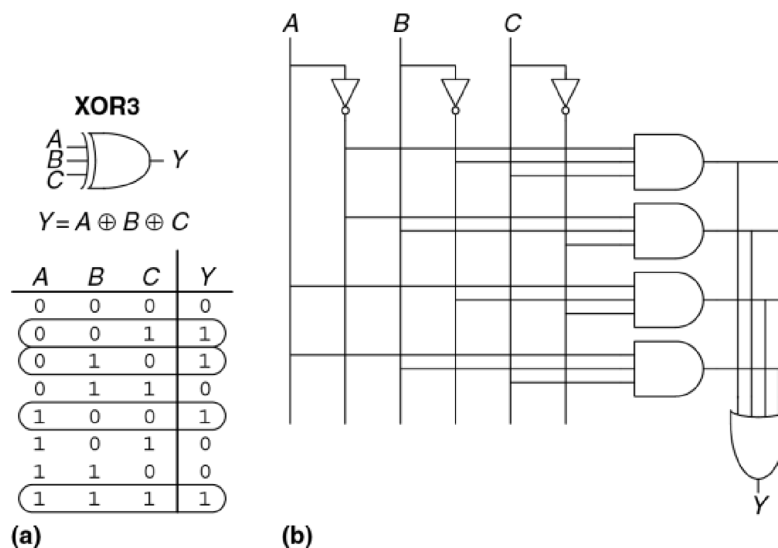


Рисунок 2.30 – Тривходовий елемент *XOR*:
функціональна специфікація (a) та реалізація з двома рівнями логіки (b)

З іншого боку, $A \wedge B \wedge C = (A \wedge B) \wedge C$ (якщо ви сумніваєтеся, доведіть це самостійно за допомогою досконалої індукції). Отже, тривходовий елемент *XOR* можна реалізувати каскадом двовходових елементів *XOR*, як зображено на рис. 2.31.

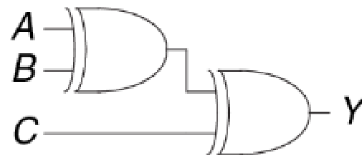


Рисунок 2.31 – Тривходовий елемент *XOR*,
зібраний із двох двовходових елементів *XOR*

Аналогічно восьмивходовий *XOR* вимагатиме 128 восьмивходових елементів І та одного 128-входового елемента АБО для дворівневої реалізації диз'юнкції кон'юнкцій. Набагато найкращою альтернативою буде використання дерева двовходових елементів *XOR*, як продемонстровано на рис. 2.32.

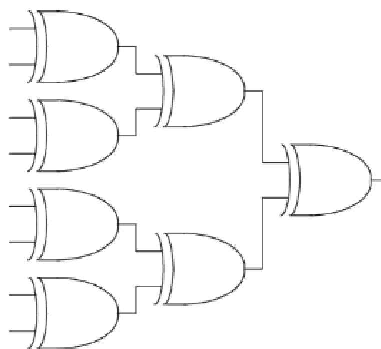


Рисунок 2.32 – Восьмивходовий елемент *XOR*,
зібраний із семи двовходових

Вибір найкращої багаторівневої реалізації заданої логічної функції – це складний процес (обирати найкращу багаторівневу реалізацію заданої логічної функції не просто). Крім того, «найкраще» має чимало значень: найменша кількість елементів, найкраща швидкодія, найкоротший час розроблення, найнижча вартість, найменше енергоспоживання.

У розділі 5 пояснюється, що «найкраща» схема для однієї технології не обов'язково є найкращою для іншої. Наприклад, ми використовували елементи І та АБО, але для КМОН-технології більш ефективні елементи І-НІ та АБО-НІ. З досвідом побачите, що для більшості схем ви зможете знаходити хорошу багаторівневу реалізацію, якщо просто розглядати ці схеми (і діяти інтуїтивно).

Деякий досвід ви напрацюєте, якщо вивчите приклади схем, наведі в посібнику. Це залежатиме від того, як ви навчаєтесь, досліджуйте різні

варіанти розроблення та думайте про компроміси. Зараз також доступні системи автоматизованого проєктування (САПР), що дають змогу розглядати величезний простір можливих багаторівневих реалізацій (здійснювати пошук у багатовимірному просторі рішень) та обирати те рішення, що найкраще задовольняє ваші критерії оптимальності з огляду на наявні будівельні блоки.

2.5.2 Переміщення інверсії

Як було розглянуто в п. 1.7.6, для КМОН-схем краще підходять елементи І-НІ та АБО-НІ, а не І та АБО. Однак читання рівнянь багаторівневих схем з елементами І-НІ та АБО-НІ може виявитися досить важким. На рис. 2.33 наведено приклад багаторівневої схеми, функція якої очевидна безпосередньо із схеми. За допомогою переміщення інверсії можна перетворити подібні схеми так, що інверсія скоротиться й функція може стати більш зрозумілою. Побудовані на принципах, поданих у п. 2.3.3, правила для переміщення інверсії такі:

- починайте з виходу ланцюга та рухайтеся назад до входів;
- перемістіть інверсію із загального виходу на входи так, щоб ви могли читати вираз у термінах виходу (наприклад, *Y*), а не інвертованого виходу $\overline{Y}$;
- просуваючись у зворотному напрямку, змінюйте кожен елемент так, щоб кількість інверсій виявилася парною та їх можна було скоротити. Якщо поточний елемент має вхідні заперечення, накресліть попередній елемент із вихідним запереченням. Якщо поточний елемент не має вхідного заперечення, накресліть попередній елемент без вихідного заперечення.

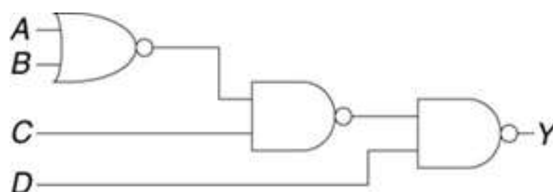


Рисунок 2.33 – Багаторівнева схема на елементах І-НІ та АБО-НІ

На рис. 2.34 показано, як змінити схему з рис. 2.33, дотримуючись викладених правил. Починаємо з виходу *Y*. Елемент І-НІ має заперечення на виході, яке хочемо усунути. Переставляємо вихідне заперечення «назад», формуючи елемент АБО з інверсними входами, що продемонстровано на рис. 2.34, *a*. Рухаючись наліво за схемою, помічаємо, що правий елемент тепер має вхідне заперечення, яке може бути відкинуте разом із вихідним запереченням середнього елемента І-НІ так, що інверсій у цьому шляху не залишиться (див. 2.34, *b*). Середній елемент не має вхідних інверсій,

тому трансформуємо лівий елемент так, щоб він не мав вихідного заперечення, як зображено на рис. 2.34, *c*. Зараз усі заперечення в схемі прибрані, за винятком входів, так що функція може бути прочитана в термінах елементів І та АБО з дійсними або комплементарними входами: $Y = = \overline{A}\,\overline{B}C + \overline{D}$.

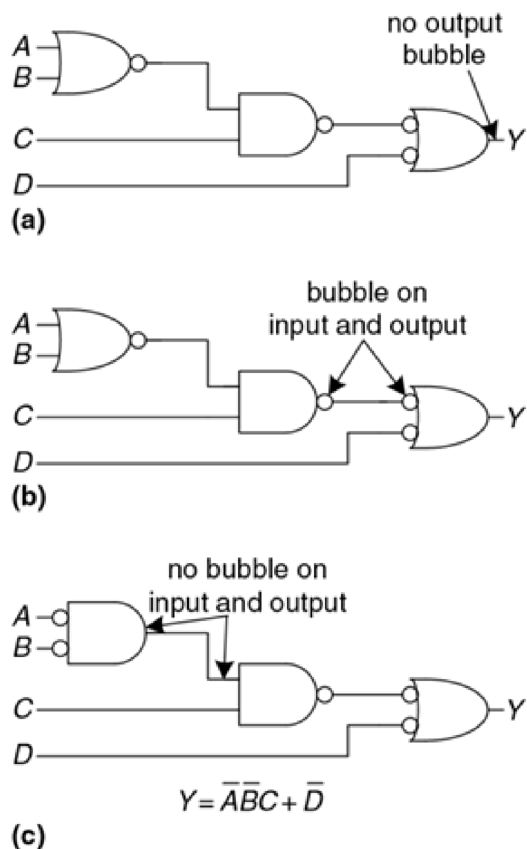


Рисунок 2.34 – Схема з вилученими інверсіями

Оскільки послідовні заперечення можуть бути відкинуті, можемо ігнорувати інверсії на виході середнього та на вході правого елементів. Тоді схема, яка логічно еквівалентна схемі, зображеній на рис. 2.34, подана на рис. 2.35.

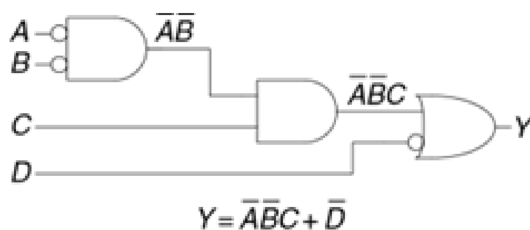


Рисунок 2.35 – Логічно еквівалентна схема

Приклад 2.8.

Переміщення інверсії в КМОН-логіці

Більшість розробників думають у термінах елементів І та АБО, але припустимо, що ви хотіли б реалізувати схему, зображену на рис. 2.36,

у КМОН-логіці, для якої кращі елементи І-НІ та АБО-НІ. Використовуйте переміщення інверсії, щоб перетворити схему на елементи І-НІ, АБО-НІ та НІ.

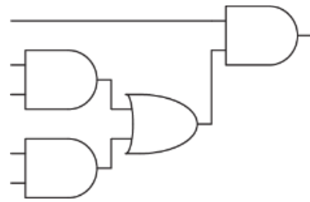


Рисунок 2.36 – Схема на елементах І та АБО

Виконання. Прямолінійне рішення полягає в простій заміні кожного елемента І на І-НІ з інвертором, а кожного елемента АБО – на АБО-НІ з інвертором, як це подано на рис. 2.37. Така схема вимагатиме вісім елементів. Зверніть увагу, що інвертори зображені із запереченням на вході, а не на виході, щоб наголосити на тому, що послідовне подвійне заперечення не змінює логіки роботи схеми та, ймовірно, буде відкинуто.

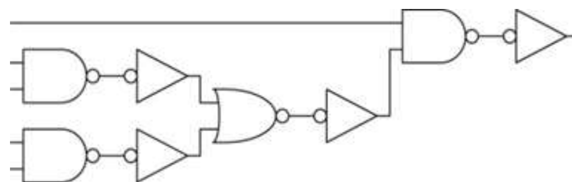


Рисунок 2.37 – Погана схема на елементах І-НІ та АБО-НІ

Зауважимо, що заперечення можуть бути додані на вихід елемента й на вхід наступного елемента без зміни функції, як показано на рис. 2.38, *a*. Вихідний елемент І перетворюється на елемент І-НІ та інвертор (рис. 2.38, *b*). Це рішення потребує лише п’яти елементів.

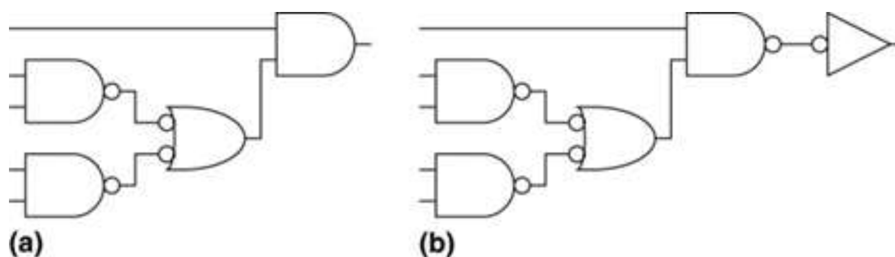


Рисунок 2.38 – Поліпшена схема на елементах І-НІ та АБО-НІ

2.6 Що за *X* і *Z*?

Булева алгебра обмежена значеннями 0 і 1. Однак реальні схеми можуть мати неприпустимий і рухомий стани, подані символами *X* і *Z* відповідно.

2.6.1 Неприпустиме значення: X

Символ *X* означає невідоме логічне значення або неприпустиме значення фізичної напруги в з'єднанні, яке не відповідає рівням логічних 0 і 1. Це зазвичай відбувається, якщо до з'єднання під'єднані виходи інших елементів схеми, що видають значення 0 і 1 одночасно. На рис. 2.39 продемонстрований випадок, коли вихід *Y* під'єнаний до елементів, що мають на виході ВИСОКИЙ та НИЗЬКИЙ рівні.

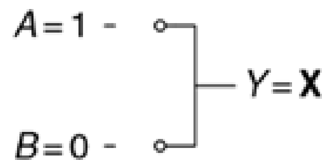


Рисунок 2.39 – Схема з неприпустимим значенням на виході

Ця ситуація, що називається змаганням, чи конфліктом (*contention*), вважається помилкою, тому її необхідно уникати. Реальна (фізична) напруга на виході з конфліктом може бути десь між нулем і напругою живлення залежно від співвідношення потужностей елементів, що видають у ланцюг ВИСОКУ та НИЗЬКУ напругу. Часто, але не завжди, значення напруги виявляється в «забороненій» зоні. Змагання також може спричинити підвищене споживання енергії елементами, що конфліктують, унаслідок чого схема нагрівається й може бути пошкоджена.

Значення *X* також іноді використовується програмами моделювання для позначення неініціалізованого значення. Наприклад, якщо ви забули визначити вхідне значення, симулятор надасть йому значення *X*, щоб попередити вас про проблему.

Як згадувалося в підрозділі 2.4, розробники цифрових схем також використовують символ *X* для позначення в таблицях істинності байдужих змінних, від яких залежить стан виходів. Коли *X* у таблицях істинності, він показує, що значення цієї змінної може бути і нулем, і одиницею. Коли *X* з'являється в схемі, це значить, що ланцюг має невідоме чи заборонене значення.

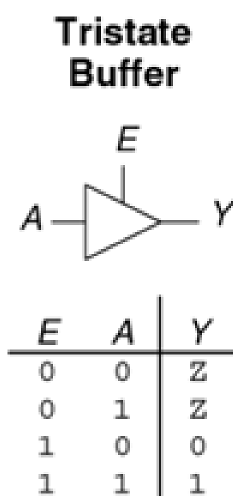
2.6.2 Третій стан: Z

Символ *Z* вказує, що напруга в ланцюзі не визначається ні джерелом ВИСОКОЇ, ні джерелом НИЗЬКОЇ напруги. Вважають, що такий ланцюг від'єднаний, перебуває в стані високого імпедансу або в третьому стані. Типово неправильне твердження – це що непід'єднаний, або рухомий, ланцюг має значення логічного 0. Насправді логічний стан непід'єднаного ланцюга

можливий як 0, так і 1, а його напруга може прийняти якесь проміжне значення залежно від історії зміни стану системи. Непід'єднаний ланцюг не обов'язково означає наявність помилки в схемі. Наприклад, якийсь інший елемент схеми може задати ланцюгу допустимий логічний рівень саме в той момент, коли цей ланцюг впливає на роботу схеми.

Один із поширених способів отримати невизначене значення – це забути під'єднати вхід схеми до джерела напруги логічного рівня або припустити, що непід'єднаний вхід – те саме, що вхід зі значенням 0. Ця помилка може призвести до того, що поведінка ланцюга буде хаотичною, оскільки невизначені значення на вході здатні випадково змінюватися з 0 в 1. Справді, торкання схеми може бути достатньо, щоб спричинити зміну через слабку статичну електрику тіла. Ми бачили схему, яка коректно працювала, аж доки студент тримав палець на мікросхемі.

Буфер із трьома станами, який зображено на рис. 2.40, має три можливі вихідні значення: ВИСОКИЙ (1), НИЗЬКИЙ (0) і від'єднаний, або рухомий, (*Z*) стан (саме тому рухомий стан називають третім). Буфер із трьома станами має вхід *A*, вихід *Y* та сигнал управління *E*. Коли сигнал дозволу (управління) має значення ІСТИНА, буфер із трьома станами працює як простий буфер, передаючи вхідне значення на вихід. Коли сигнал управління має значення *FALSE*, вихід буфера перемикається в третій стан і стає рухомим (*Z*). Буфер із трьома станами (рис. 2.40) має активний високий рівень. Це значить, що коли сигнал дозволу ВИСОКИЙ (1), передача дозволена.

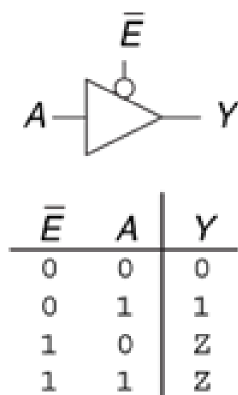


E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Рисунок 2.40 – Буфер із трьома станами

На рис. 2.41 зображено буфер із трьома станами з активним низьким рівнем. Коли сигнал управління низький (0), передача дозволена. Бачимо, що сигнал має активний низький рівень через заперечення на його вхідному ланцюзі. Ми часто позначаємо вхід з активним низьким рівнем,

використовуючи риску (символ заперечення) над його ім'ям (E) або додаючи букву *b* або *bar* після імені, *Eb* або *Ebar*.



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Рисунок 2.41 – Буфер із трьома станами з активним низьким рівнем

Буфери з третім станом зазвичай використовуються в шинах, що з'єднують кілька мікросхем. Наприклад, мікропроцесор, відеоконтролер та *Ethernet*-контролер можуть потребувати взаємодії з підсистемою пам'яті в персональному комп'ютері. Кожна мікросхема здатна під'єднуватися до загальної шини пам'яті, застосовуючи буфери з третім станом (див. рис. 2.42). У цьому лише одна мікросхема має право виставити свій сигнал дозволу, щоб видати значення на шину. Виходи інших мікросхем мають перебувати в третьому стані, щоб не стати причиною колізії з мікросхемою, яка здійснює обмін даними з пам'яттю. Однак будь-яка мікросхема може читати інформацію із загальної шини в будь-який час. Такі шини на основі буферів із трьома станами колись були дуже поширеними. Однак у сучасних комп'ютерах найвищі швидкості можливі лише за умови з'єднання мікросхем одна з одною безпосередньо (*point-to-point*), а не за допомогою загальної шини.

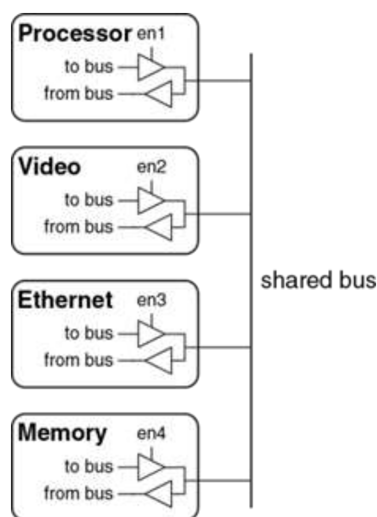


Рисунок 2.42 – Шина з третім станом, що з'єднує кілька мікросхем

2.7 Карти Карно

Після того, як ви здійсните кілька перетворень з мінімізації булевих рівнянь, використовуючи булеву алгебру, зрозумієте, що без дотримання належної акуратності іноді можна отримати рішення, що дуже різниться від необхідного спрощеного рівняння. Карти Карно є наочним методом для спрощення булевих рівнянь. Їх винайшов 1953 р. Моріс Карно, телекомунікаційний інженер з фірми *Bell Labs*. Карти Карно дуже зручні, коли рівняння містить до чотирьох змінних. Але, найважливіше, вони дають розуміння сутності під час маніпулювання логічними висловлюваннями.

Як пам'ятаємо, логічна мінімізація здійснюється способом склеювання термів. Два терми, що містять імпліканту P і два логічних значення деякої змінної A, об'єднуються, у цьому разі змінна A вимикається. Карти Карно дають змогу легко знаходити терми, які можна склеїти, розташовуючи їх у вигляді таблиці.

На рис. 2.43 подано таблицю істинності та карту Карно для функції трьох змінних. Верхній рядок дає чотири можливі значення для змінних A і B. Лівий стовпець дає два можливі значення змінної C. Кожна комірка карти Карно відповідає рядку таблиці істинності та містить значення функції Y з цього рядка. Наприклад, верхня ліва комірка відповідає першому рядку таблиці істинності та показує, що значення функції Y дорівнюватиме 1, коли $ABC = 000$. Як і кожен рядок у таблиці істинності, будь-яка комірка карти Карно є окремим мінтермом. Для кращого розуміння на рис. 2.43, *c* подано мінтерми, що відповідають кожній комірці карти Карно.

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(a)

Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

(b)

Y: C \ AB	00	01	11	10
0	$\overline{A}\overline{B}\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
1	$\overline{A}\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

(c)

Рисунок 2.43 – Функція трьох змінних:
таблиця істинності (a), карта Карно (b), карта Карно з мінтермами (c)

Кожна комірка, або мінтерм, розрізняється від сусідньої зміною лише однієї змінної. Це означає, що сусідні комірки різняться лише значенням одного літерала. Наприклад, комірки, подані мінтермами $A\ B\ C$ і $A\ B\ \overline{C}$, – сусідні й різняться тільки в змінній C. Ви, мабуть, також звернули увагу,

що змінні *A* і *B* комбінуються у верхньому рядку за особливим порядкам: 00, 01, 11, 10. Цей порядок називається кодом Грея (*Gray code*). На відміну від бітового порядку зростання величини (00, 01, 10, 11), у коді Грея сусідні записи розрізняються лише одним розрядом. Наприклад, 01 : 11 різниться лише зміною *A* з 0 на 1, тоді як 01 : 10 вимагає зміни *A* з 1 в 0 і *B* з 0 в 1. Отже, звичайний послідовний побітний порядок не дає необхідної нам властивості сусідніх комірок, що мають розрізнятися тільки в одній змінній.

Код Грея 1953 р. запатентував Френк Грей, дослідник з *Bell Labs*. Цей код особливо корисний для електромеханічних перетворювачів (наприклад, датчиків кута повороту), оскільки він дає змогу позбутися хибних спрацьовувань. Код Грея може бути будь-якої розрядності. Наприклад, трибітний код виглядає так: 000, 001, 011, 010, 110, 111, 101, 100.

Льюїс Керролл 1879 р. опублікував схожу загадку в журналі *Vanity Fair*. «Правила прості. Дано два слова однакової довжини. Потрібно з’єднати їх ланцюжком слів, у якому два сусідні слова розрізняються лише однією літерою», – написав він.

Наприклад, слово *SHIP* можна перетворити на *DOCK* у такий спосіб: *SHIP, SLIP, SLOP, SLOT, SOOT, LOOT, LOOK, LOCK, DOCK*. Чи можете ви знайти коротший ланцюжок?

Карти Карно також «закільцьовані». Комірка з правого краю таблиці є сусідньою з тою, що розташована з лівого краю, оскільки вони розрізняються тільки в одній змінній (*A*). Можна згорнути карту в циліндр, якщо з’єднати краї, і навіть у цьому разі сусідні комірки також розрізнятимуться лише в одній змінній.

2.7.1 Думайте про овали

На карті Карно (рис. 2.43) є тільки дві одиниці, що відповідає числу мінтермів у рівнянні ($\overline{A}\ \overline{B}\ \overline{C}$ і $\overline{A}\ \overline{B}\ C$). Читання мінтермів з карт Карно точно відповідає читанню диз’юнктивної нормальної форми (ДНФ) з таблиці істинності.

Як і раніше, ми могли б застосовувати булеву алгебру для мінімізації:

$$Y = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C = \overline{A}\ \overline{B}\ (\overline{C} + C) = \overline{A}\ \overline{B}. \qquad (2.7)$$

Карти Карно допомагають робити це спрощення графічно, якщо окреслити одиниці в сусідніх комірках овалами, як зображено на рис. 2.44. Для кожного овала пишемо відповідну йому імпліканту. Згадайте з підрозділу 2.2, що імпліканта – це добуток одного або кількох літералів. Змінні, пряма й комплементарна форми яких потрапляють у один овал, вилучаються з імпліканти. У нашій ситуації обидві форми змінної *C* потрапляють в овал,

тому не додаємо її в імпліканту. Іншими словами, Y = ІСТИНА, коли $A = B = 0$ незалежно від C. Так що імплікантою буде $\overline{A}\ \overline{B}$: карта Карно дає ту саму відповідь, яку отримали за допомогою булевої алгебри.

Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

Рисунок 2.44 – Мінімізація за допомогою карти Карно

2.7.2 Логічна мінімізація на картах Карно

Карти Карно забезпечують простий візуальний спосіб мінімізації логічних виразів. Просто окресліть всі прямокутні блоки з одиницями на карті, використовуючи найменшу можливу кількість овалів. Кожен овал має бути максимально великим. Потім прочитайте всі обведені імпліканти.

Нагадаємо, що формально рівняння булевої алгебри є мінімальними, лише коли записані як сума найменшого числа первинних імплікант. Кожен овал на карті Карно є імплікантою. Найбільш можливий овал є первинною імплікантою.

Наприклад, на карті Карно (рис. 2.44) $\overline{A}\ \overline{B}\ \overline{C}$ і $\overline{A}\ \overline{B}\ C$ – імпліканти, але не первинні. На цій карті тільки $\overline{A}\ \overline{B}$ є первинною імплікантою. Правила для знаходження мінімального рівняння з карток Карно такі:

- необхідно використовувати найменшу кількість овалів для покриття всіх 1;
- усі комірки в кожному овалі мають містити 1;
- кожен овал має охоплювати блок, кількість комірок якого в кожному напрямку дорівнює степеню двійки (тобто 1, 2 або 4);
- кожен овал має бути настільки великим, наскільки це можливо;
- овал може пов’язувати краї картки Карно;
- одиниця на карті Карно може бути обведена скільки завгодно разів, якщо це дає змогу зменшити кількість овалів, які застосовуватимуться.

Приклад 2.9.

Мінімізація функції трьох змінних за допомогою карти Карно

Припустимо, ми маємо функцію $Y = F(A, B, C)$ з картою Карно (див. рис. 2.45). Спростіть цей вираз, використовуючи карту Карно.

Y: C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Рисунок 2.45 – Карта Карно для прикладу 2.9

Виконання. Окреслимо одиниці на карті Карно, застосовуючи найменшу можливу кількість овалів, як зображено на рис. 2.46. Кожен овал на карті Карно є первинною імплікантою, а його розмір кратний степеню двійки (2 × 1 і 2 × 2).

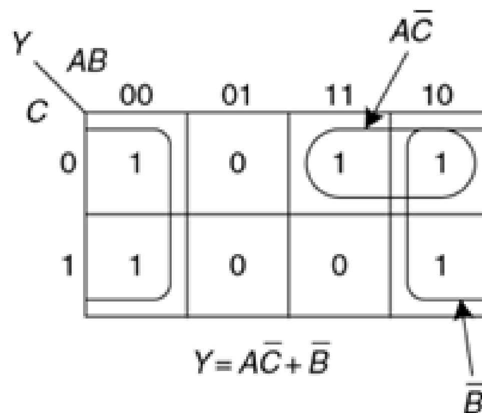


Рисунок 2.46 – Рішення для прикладу 2.9

Сформуємо первинну імпліканту для кожного виділеного овала, виписуючи тільки ті змінні, що з'являються в ньому лише в прямій або комплементарній формах. Наприклад, овал розміром 2 × 1 містить пряму та комплементарну форми змінної B, так що ми не додаємо змінну B у первинну імпліканту. Однак у цьому овалі є тільки пряма форма змінної A і комплементарна форма змінної C ($\overline{C}$), так що ми додаємо ці змінні в первинну імпліканту A ($\overline{C}$). Подібним способом овал розміром 2 × 2 покриває всі комірки, де $B = 0$, так що первинна імпліканта буде $\overline{B}$.

Зауважте, що права верхня комірка (мінтерм) використовується двічі, щоб зробити овали первинних імплікант якомога більшими. Як бачили в булевій алгебрі, це еквівалентно спільному використанню мінтерму для зменшення розміру імпліканти. Також зверніть увагу на те, що овал, який покриває чотири комірки, обертається через краї карти Карно.

Приклад 2.10.

Дешифратор семисегментного індикатора

Дешифратор семисегментного індикатора отримує на вхід чотирибітні дані *D*[3:0] і формує сім виходів для керування світлодіодами для показу цифр від 0 до 9. Сім виходів часто називають сегментами від *a* до *g*, або *Sa*-*Sg* (див. рис. 2.47). Самі цифри подані на рис. 2.48. Сформуємо таблицю істинності для виходів та використаємо карти Карно з метою знаходження логічного рівняння для виходів *Sa* та *Sb*. Припустимо, що заборонені вхідні значення (10–15) нічого не виводять на індикатор.

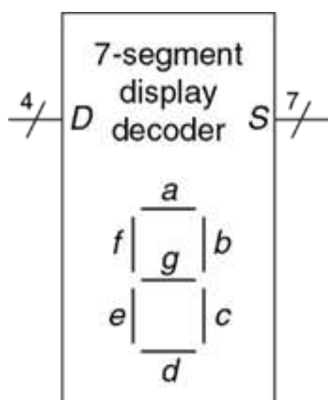


Рисунок 2.47 – Семисегментний індикатор

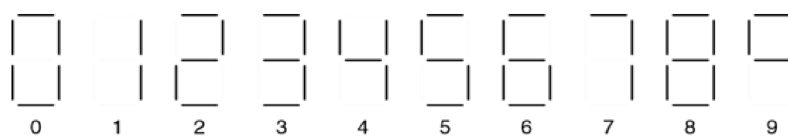


Рисунок 2.48 – Цифри на семисегментному індикаторі

Виконання. Нижче подана таблиця істинності (табл. 2.6). Наприклад, вхід 0000 має містити всі сегменти, за винятком *Sg*.

Таблиця 2.6 – Таблиця істинності дешифратора семисегментного індикатора

***D*3:0**	***Sa***	***Sb***	***Sc***	***Sd***	***Se***	***Sf***	***Sg***
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
Інші	0	0	0	0	0	0	0

Кожен із семи виходів є незалежною функцією від чотирьох змінних. Карти Карно для виходів *Sa* та *Sb* продемонстровано на рис. 2.49. Пам’ятайте, що сусідні комірки можуть розрізнятися тільки однією змінною, так що промаркуємо рядки та стовпці в коді Грея: 00, 01, 11, 10. Будьте уважні й пам’ятайте цей порядок, коли вписуватимете значення виходів у комірки.

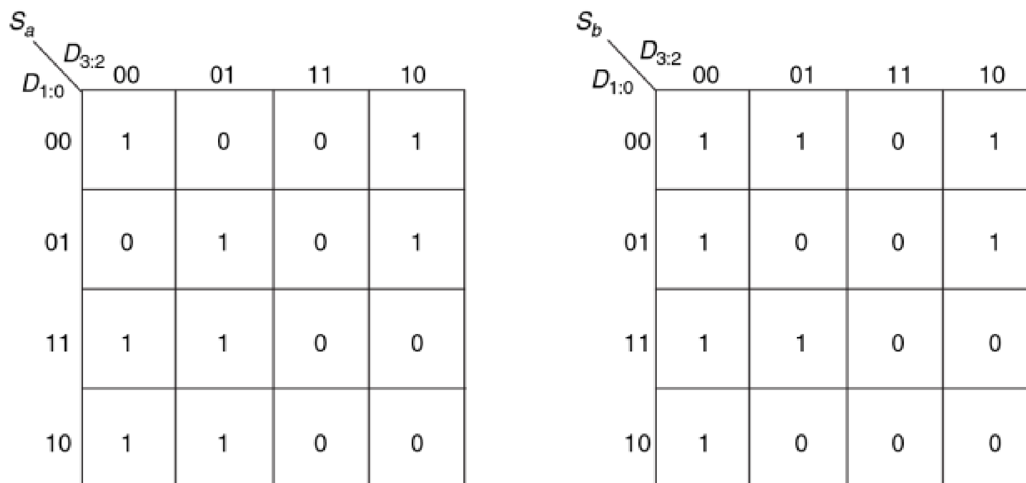


Рисунок 2.49 – Карти Карно для *Sa* та *Sb*

Потім окреслимо первинні імпліканти. У цьому разі використовуємо мінімально потрібну кількість овалів для покриття всіх одиниць. Овали можуть пов’язувати краї (вертикальні та горизонтальні), а кожна одиниця може бути виділена кілька разів. На рис. 2.50 зображено первинні імпліканти та спрощені логічні рівняння.

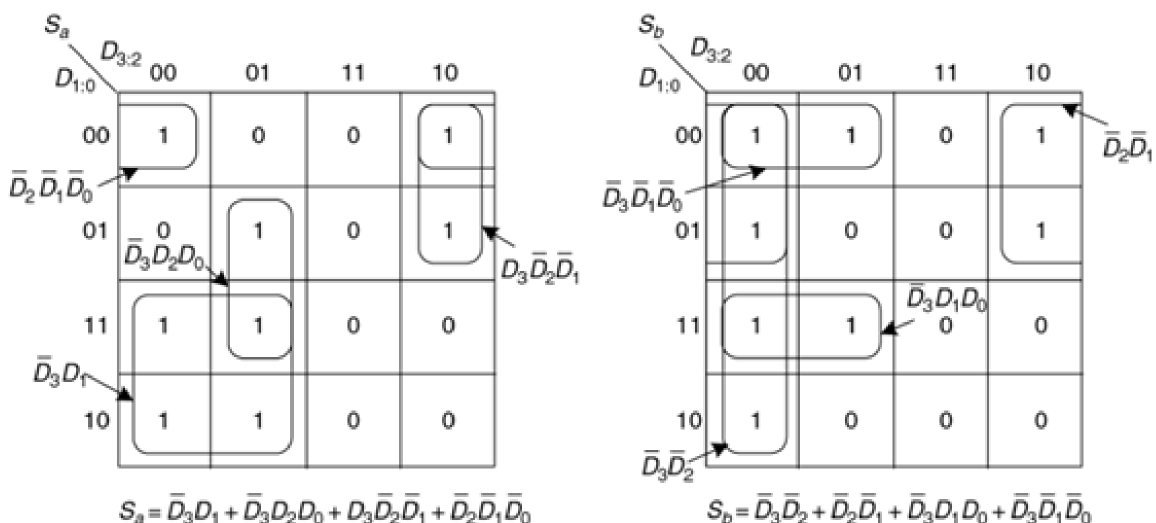


Рисунок 2.50 – Виконання вправи 2.10

Зауважте, що мінімальний набір первинних імплікант не єдиний можливий. Наприклад, запис 0000 на карті Карно для *Sa* може бути виділений

разом із записом 1000, отримуючи мінтерм $\overline{D}2\overline{D}1\overline{D}0$. Але замість цього овал може містити запис 0010, отримуючи мінтерм $\overline{D}\ 3\overline{D}\ 2\overline{D}\ \ 0$, як позначено пунктирною лінією на рис. 2.51.

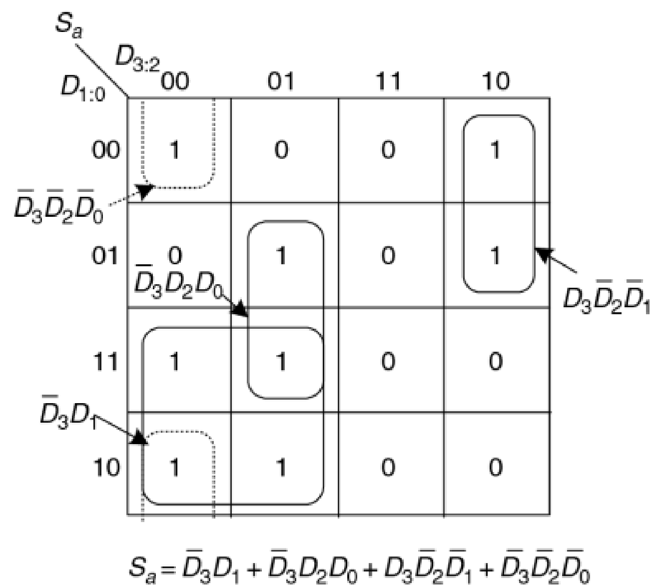


Рисунок 2.51 – Альтернативна карта Карно для *Sa*,
що використовує інший набір первинних імплікант

Рис. 2.52 ілюструє поширену помилку, коли первинна імпліканта обирається для покриття 1 у лівому верхньому куті. Цей мінтерм $\overline{D}3\overline{D}2\overline{D}1\overline{D}0$ дає диз'юнкцію кон'юнкцій, яка не мінімізована. Його можна було б скомбінувати з будь-яким із двох сусідніх мінтермів для отримання овала більшого розміру, як було зроблено на попередніх двох рисунках.

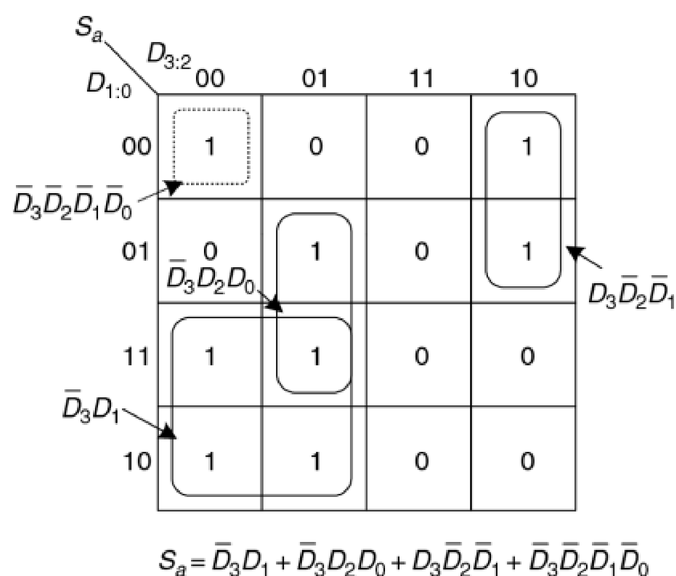


Рисунок 2.52 – Карта Карно для *Sa*, що застосовує некоректну імпліканту

2.7.3 Байдужі змінні

Згадайте, що байдужі змінні в таблиці істинності були додані в підрозділі 2.4 для зменшення кількості її рядків у тому разі, коли відповідні змінні не впливають на вихід. Вони позначені символом *X*, який означає, що значення вхідної змінної може бути 0 або 1.

Не тільки входи, але й виходи можуть бути байдужими, якщо стан виходу не є важливим або відповідна комбінація входів ніколи не виникає. Такі виходи можуть трактуватися або як 0, або як 1 залежно від того, як вирішить розробник.

У картах Карно байдужі змінні дають змогу провести ще більшу логічну мінімізацію. Їх можна додавати до овалів, якщо це допомагає покрити одиниці або меншою кількістю овалів, або овалами, більшими за розміром, але їх можна і не покривати, якщо це не сприяє мінімізації.

Приклад 2.11.

Дешифратор семисегментного індикатора з байдужими змінними

Повторимо приклад 2.10 для ситуації, коли нас цікавлять значення виходів за умови заборонених вхідних значень від 10 до 15.

Виконання. Карту Карно з байдужими елементами, позначеними як *X*, подано на рис. 2.53. Оскільки такі елементи можуть дорівнювати як 0, так і 1, використовуємо їх там, де це допоможе покрити одиниці або меншою кількістю овалів, або овалами, більшими за розміром. Окреслені значення *X* трактуються як 1, неокреслені – як 0. Подивіться, як для сегмента *Sa* можна виділити овал розміром 2×2, що об’єднує всі чотири кути. Використовуйте комірки з байдужими значеннями для спрощення логіки.

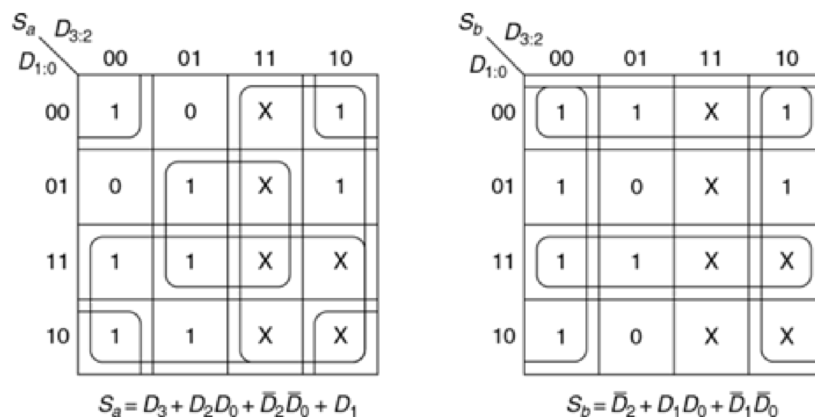


Рисунок 2.53 – Карта Карно з байдужими змінними

2.7.4 Підбиття підсумків

Булева алгебра та карти Карно – два методи логічного спрощення. Зрештою, метою є знаходження найменш витратного методу реалізації конкретної логічної функції.

У сучасній інженерній практиці комп'ютерні програми, що називаються синтезаторами логіки (*logic synthesizers*), спрощують схеми з опису логічних функцій, як ми побачимо в розділі 4. Для великих завдань програми логічного синтезу набагато ефективніші, ніж люди. Для маленьких завдань людина з деяким досвідом може знайти хороше рішення «на око». Рідко хто використовує карти Карно в реальному житті для розв'язання практичних завдань. Але розуміння принципів, що лежать в основі карт Карно, дуже важливе. Крім того, знання карт Карно часто запитують на співбесідах.

2.8 Базові комбінаційні блоки

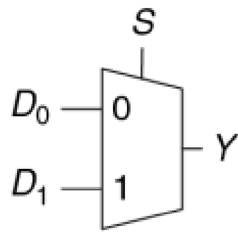
Комбінаційні логічні елементи часто групуються в «будівельні блоки», що застосовуються для створення складних систем. Це дає змогу абстрагуватися від зайвої деталізації рівня логічних елементів та зосередитися на функції «будівельного блоку». Ми вже вивчили три такі блоки: повний суматор (див. підрозділ 2.1), схеми пріоритету (див. підрозділ 2.4) та дешифратор семисегментного індикатора (див. підрозділ 2.7). Цей розділ описує два типи блоків, які ще частіше використовуються в проєктуванні: мультиплексори та дешифратори. У розділі 5 ітиметься про інші комбінаційні «будівельні блоки».

2.8.1 Мультиплексори

Мультиплексори є комбінаційними схемами, що найчастіше використовуються. Вони дають змогу обрати одне вихідне значення із кількох вхідних залежно від значення сигналу вибору.

Двовходовий мультиплексор (2:1)

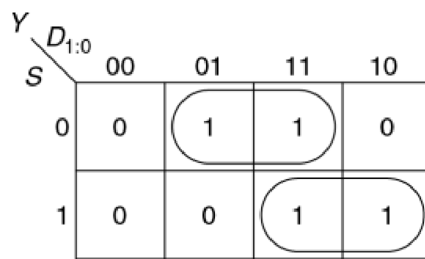
На рис. 2.54 зображена умовна графічна познака й таблиця істинності для двовходового мультиплексора (2:1) з двома входами даних $D0$ та $D1$, входом вибору S та одним виходом Y. Мультиплексор передає на вихід один із двох вхідних сигналів даних, ґрунтуючись на сигналі вибору: якщо $S = 0$, вихід $Y = D0$, і якщо $S = 1$, то вихід $Y = D1$. S також називають сигналом управління, оскільки він керує поведінкою мультиплексора.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Рисунок 2.54 – Умовна познака й таблиця істинності двовходового мультиплексора

Двовходовий мультиплексор може бути побудований з використанням диз'юнкції кон'юнкцій (суми), як зображено на рис. 2.55. Його логічний вираз може бути отримано за допомогою карт Карно чи складено з огляду на опис ($Y = 1$, якщо $S = 0$ І $D0 = 1$ АБО якщо $S = 1$ І $D1 = 1$).



$$Y = D_0\overline{S} + D_1 S$$

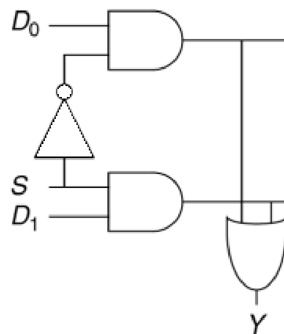


Рисунок 2.55 – Реалізація двовходового мультиплексора з використанням дворівневої логіки

Мультиплексор може бути побудований на буферах із третім станом (див. рис. 2.56). Сигнали дозволу буферів із третім станом організовані так, що постійно активний лише один буфер. Коли $S = 0$, то увімкнено тільки елемент $T0$, що дає змогу сигналу $D0$ передаватися на вихід Y. Коли $S = 1$, активний тільки елемент $T1$, що передає на вихід сигнал $D1$.

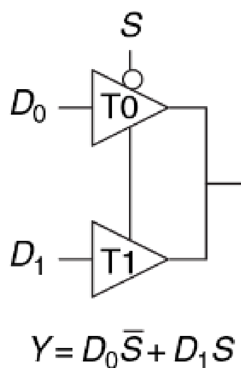


Рисунок 2.56 – Мультиплексор на буферах із трьома станами

Багатовходові мультиплексори

Чотиривходовий мультиплексор (4:1) має чотири входи даних і один вихід (рис. 2.57). Для вибору одного з чотирьох входів даних потрібен дворозрядний сигнал керування.

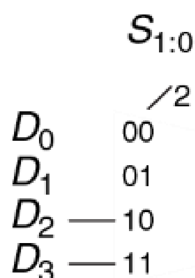


Рисунок 2.57 – Чотиривходовий мультиплексор

Чотиривходовий мультиплексор може бути побудований з використанням диз'юнкції кон'юнкцій буферів із трьома станами або застосуванням двовходових мультиплексорів (рис. 2.58).

Кон'юнкції, під'єднані до сигналів дозволу роботи буферів із трьома станами можуть бути побудовані з використанням елементів І та інверторів. Вони також можуть бути сформовані дешифратором, який розглянемо в п. 2.8.2.

Мультиплексори із значною кількістю входів (наприклад, восьмивходові або 16-входові) можуть бути побудовані простим масштабуванням методів, продемонстрованих на рис. 2.58. У цьому разі мультиплексор N:1 вимагає $log2N$

сигналів керування. Вибір найкращої реалізації, як і раніше, залежить від застосованої технології.

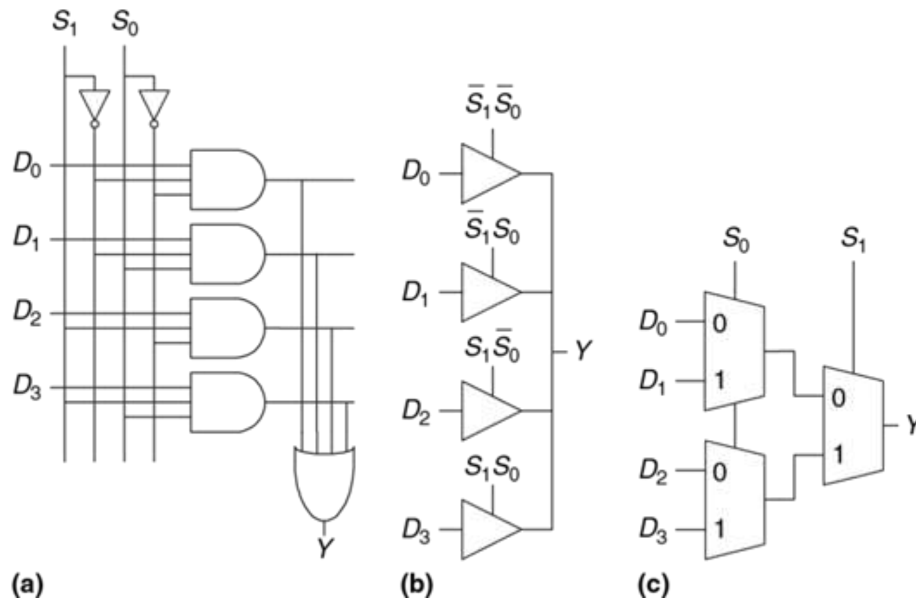


Рисунок 2.58 – Реалізація чотиривходового мультиплексора: дворівнева логіка (a); буфер з трьома станами (b); ієрархічна реалізація (c)

Логіка на мультиплексорах

Мультиплексори можна використовувати як таблиці перетворення (*lookup tables*) до виконання логічних функцій. На рис. 2.59 зображено чотиривходовий мультиплексор, що застосовується для реалізації двовходового елемента І.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

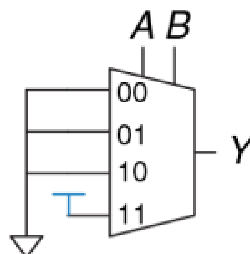


Рисунок 2.59 – Отримання двовходового елемента І з чотиривходового мультиплексора

Входи A і B є лініями управління. Входи даних мультиплексора під’єднано до 0 та 1 згідно з відповідним рядком таблиці істинності. Узагалі,

вхідний мультиплексор 2^N можна запрограмувати для виконання будь-якої N-входової логічної функції за допомогою 0 і 1 для відповідних входів інформації. Справді, зміною вхідних даних мультиплексор може бути перепрограмований на виконання різних функцій.

Трохи кмітливості – і зможемо зменшити розмір мультиплексора наполовину, використовуючи тільки вхідний мультиплексор 2^{N-1} для виконання будь-якої N-входової логічної функції. Спосіб полягає в тому, щоб подати один з літералів, як і 0, і 1, на вхід даних мультиплексора.

Для ілюстрації цього принципу на рис. 2.60 подано функції двовходових елементів І та ВИКЛЮЧНЕ АБО, реалізованих на двовходових мультиплексорах. Ми почали зі звичайної таблиці істинності й потім скомбінували пари рядків, щоб виключити праву вхідну змінну (B), і виразити вихід в термах цієї змінної. Наприклад, за умови елемента І, коли A = 0, то Y = 0 незалежно від B. Коли A = 1, то Y = 0; коли B = 0, то Y = 1, коли B = 1, то Y = B. Потім використовуємо мультиплексор як таблицю підстановки відповідно до цієї нової зменшеної таблиці істинності.

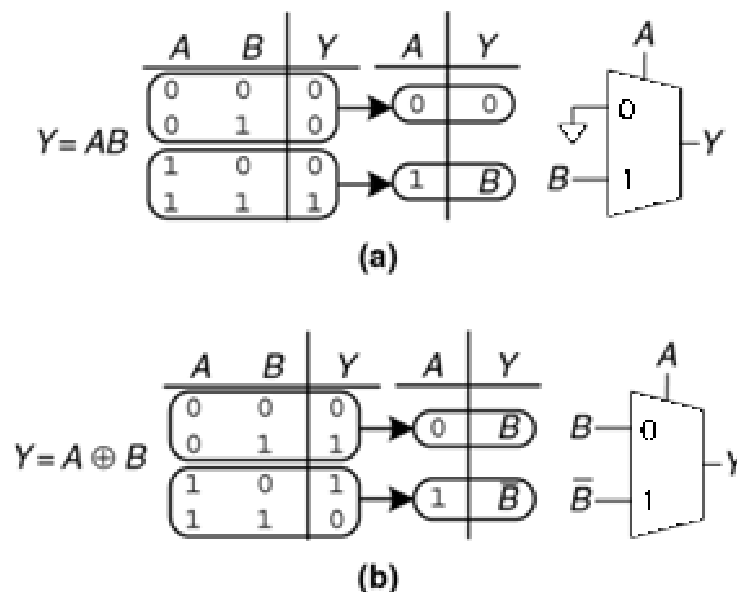


Рисунок 2.60 – Реалізація логічних функцій на мультиплексорах

Приклад 2.12.

Логіка з мультиплексорами

Алісі Хакер необхідно реалізувати функцію $Y = A\overline{B} + \overline{B}\ \overline{C} + \overline{A}\ BC$ для завершення курсового проєкту. Коли вона подивилася, які мікросхеми доступні їй у лабораторії, то побачила, що там залишився лише восьмивходовий мультиплексор. Як реалізувати цю функцію?

Виконання. На рис. 2.61 зображено схему, яку розробила Аліса з використанням одного восьмивходового мультиплексора. Він відіграє роль таблиці перетворення, де кожен рядок таблиці істинності відповідає входу мультиплексора.

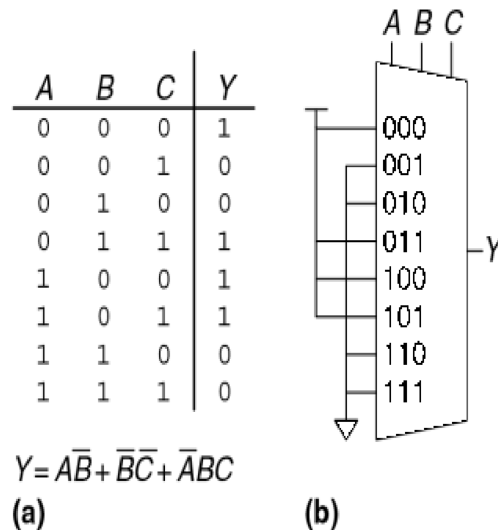


Рисунок 2.61 – Схема Аліси:
таблиця істинності (a); реалізація на восьмивходовому мультиплексорі (b)

Приклад 2.13.

Логіка з мультиплексорами, повторення

Аліса ще раз увімкнула свою схему перед захистом проєкту й спалила єдиний восьмивходовий мультиплексор (дівчина після безсонної ночі випадково подала напругу 20 В замість 5 В).

Тепер Аліса просить у своїх друзів запасні елементи, і їй дають чотиривходовий мультиплексор та інвертор. Чи зможе вона зібрати свою схему, використовуючи лише ці елементи?

Виконання. Аліса зменшила свою таблицю істинності до чотирьох рядків, зробивши вихід залежним від *C*. (Дівчина могла б також вимкнути будь-який з двох інших стовпців таблиці істинності, зробивши вихід залежним від *A* або *B*). Нова схема зображена на рис. 2.62.

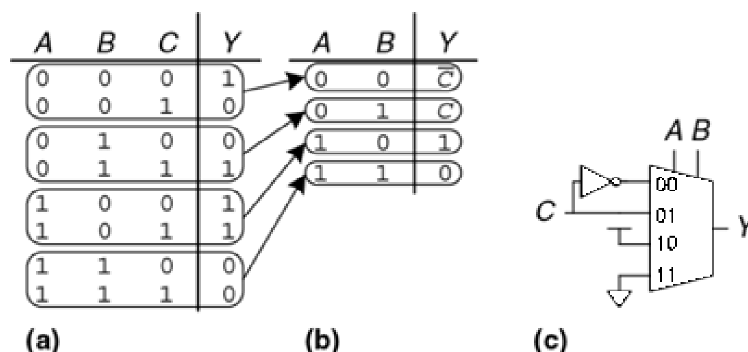
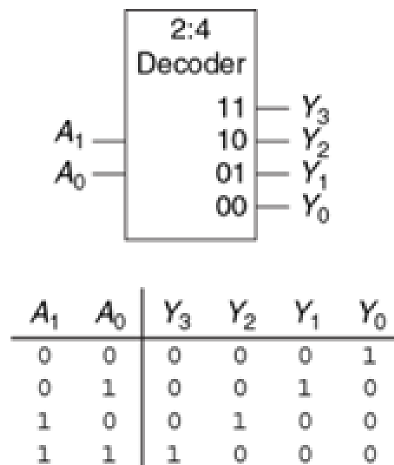


Рисунок 2.62 – Нова схема Аліси

2.8.2 Дешифратори

Дешифратор має *N* входів і 2*N* виходів. Він видає одиницю тільки на одному з виходів залежно від набору вхідних значень. На рис. 2.63 продемонстровано дешифратор 2:4. Якщо *A*[1:0] = 00, *Y*0 = 1. Якщо *A*[1:0] = 01, *Y*1 = 1 тощо. Виходи утворюють прямий унітарний код (*one-hot code*), назва якого зумовлена тим, що в будь-який час тільки один із виходів може приймати високий рівень.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Рисунок 2.63 – Дешифратор 2:4

Приклад 2.14.

Реалізація дешифратора

Реалізуйте дешифратор 2:4 на елементах І, АБО та НІ.

Виконання. На рис. 2.64 показано реалізацію дешифратора 2:4, що використовує чотири елементи І. Кожен елемент залежить або від дійсної, або від комплементарної форми кожного входу. Взагалі, дешифратор N:2^N може бути побудований з 2^N *N*-входових елементів, до яких підходять різні комбінації дійсних і комплементарних входів. Кожен вихід у дешифраторі є одиночним мінтермом. Наприклад, *Y*0 – мінтерм $\overline{A_1}\ \overline{A_0}$. Ця обставина буде зручна у використанні дешифратора з іншими цифровими базовими блоками.

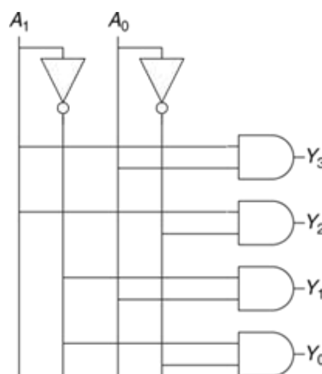


Рисунок 2.64 – Реалізація дешифратора 2:4

Логіка на дешифраторах

Дешифратор може комбінуватися з елементами АБО для побудови логічних функцій. На рис. 2.65 подана двовходова функція ВИКЛЮЧАЄ АБО-НІ (*XNOR*), що використовує дешифратор 2:4 та один елемент АБО. Оскільки кожен вихід дешифратора – одиночний мінтерм, функція побудована як логічне АБО всіх мінтермів у цій функції. На рис. 2.65 зображено реалізацію функції.

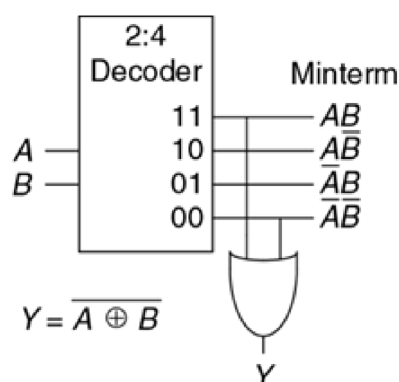


Рисунок 2.65 – Реалізація логічної функції на дешифраторі

У використанні дешифраторів для реалізації логічних функцій найпростіше подати функцію таблицею істинності або записати її у диз’юнктивній нормальній формі. N-входова функція, що має M одиниць у таблиці істинності, може бути побудована з використанням N:2^N дешифратора і M-входового елемента АБО, під’єднаним до всіх мінтерм, що містить одиницю таблиці істинності. Ця ідея буде застосована для створення постійного запам’ятовувального пристрою (ПЗП) у п. 5.5.6.

2.9 Часові властивості

У попередніх розділах ми зосереджувалися насамперед на роботі схеми, яка в ідеалі використовує найменшу кількість елементів. Однак, як підтвердить будь-який досвідчений розробник, одне з найскладніших завдань у створенні схем – це облік усіх обмежень, що накладаються на часові властивості роботи схеми, адже хороша схема має працювати дуже швидко й водночас без збоїв.

Зміна вихідного значення у відповідь на зміну входу займає час. На рис. 2.66 продемонстровано затримку між зміною входу буфера та наступною зміною його виходу. Цей рисунок називається часовою діаграмою; він зображує перехідну властивість схеми буфера за умови зміни входу.

Перехід від низького рівня до високого називається позитивним перепадом, або фронтом. А перехід від ВИСОКОГО рівня до НИЗЬКОГО (на рисунку не показано) називається, відповідно, негативним перепадом, чи зрізом. Ліва стрілка вказує, що позитивний фронт сигналу *Y* викликається позитивним фронтом сигналу *A*. Величина затримки вимірюється від часу, коли вхідний сигнал *A* досягає рівня 50%, до досягнення рівня 50% вихідним сигналом *Y*. Рівень 50% – це точка, у якій сигнал розташований рівно посередині між низьким і високим логічними рівнями.

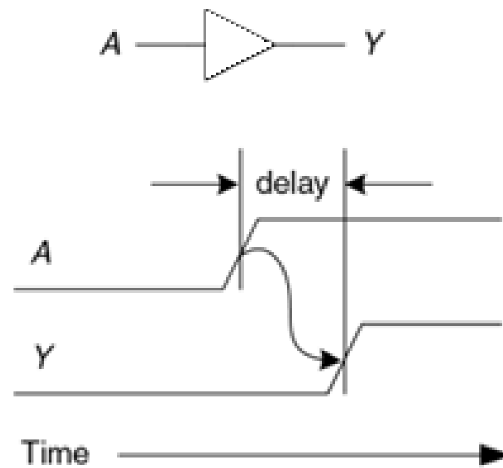


Рисунок 2.66 – Затримка схеми

2.9.1 Затримка поширення та затримка реакції

Комбінаційній логіці властива затримка поширення (*propagation delay*) та затримка реакції, або відгуку (*contamination delay*). Затримка поширення *tpd* – це максимальний час від початку зміни входу до моменту, коли всі виходи досягнули значень, що встановилися. Затримка реакції *tcd* – це мінімальний час від моменту, коли вхід змінився, до моменту, коли будь-який із виходів почне змінювати своє значення.

На рис. 2.67 ліва та права стрілки позначають затримки поширення та реакції буфера відповідно. На рисунку продемонстровано, що вхід *A* спочатку мав або високе, або низьке значення, і воно змінюється на протилежне в певний момент часу. Нас цікавить лише те, що значення *A* змінилося, але з його конкретного значення. У відповідь за деякий час змінюється *Y*. Стрілки показують, що *Y* може почати змінюватися через часовий інтервал t_{CD} після зміни *A* і що *Y* точно встановиться в нове значення не пізніше, ніж через інтервал t_{PD}.

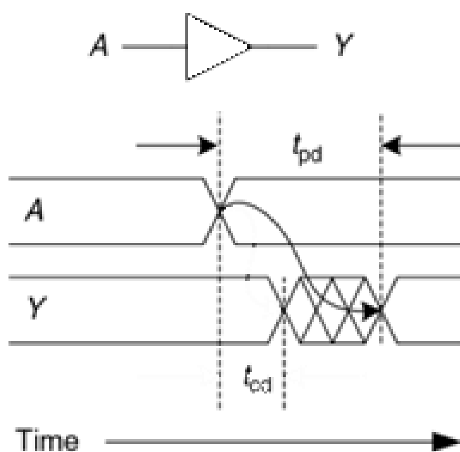


Рисунок 2.67 – Затримка поширення та затримка реакції

Основні причини затримок у схемах полягають у часі, необхідному для перезарядження ємностей ланцюга, а також кінцевої швидкості поширення електромагнітних хвиль у середовищі. Величини t_{PD} і t_{CD} можуть розрізнятися з багатьох причин, що передбачають:

● різні затримки наростання та спаду сигналу;

● кілька входів і виходів, одні з яких швидші за інші;

● уповільнення роботи схеми за умови підвищення температури та прискорення внаслідок охолодження.

Обчислення *tpd* і *tcd* вимагає заглиблення в нижні рівні абстракцій, що виходить за межі цього посібника. Проте виробники зазвичай надають документацію зі специфікацією цих затримок кожного елемента.

Поряд з переліченими факторами, затримки поширення та реакції також визначаються шляхом, що проходить сигнал від входу до виходу. На рис. 2.68 показано чотиривходову схему. Критичний шлях (*critical path*) – це шлях від входу *A* або *B* до виходу *Y*. Він відповідає ланцюгу з найбільшою затримкою, оскільки вхідному сигналу необхідно пройти три елементи до виходу. Цей шлях критичний і обмежує швидкість схеми. Найкоротший шлях у схемі від входу *D* до виходу *Y*. Він найшвидший у схемі, оскільки вхідному сигналу до виходу потрібно пройти лише один елемент.

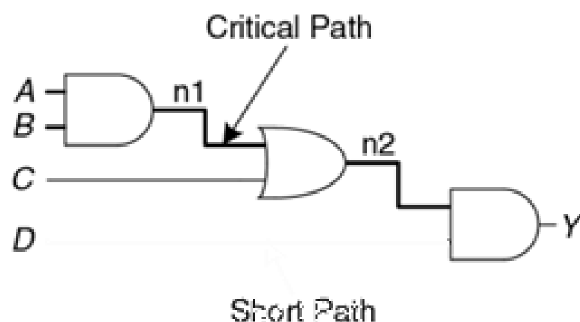


Рисунок 2.68 – Найкоротший ланцюг і ланцюг з найбільшою затримкою

Затримка розповсюдження комбінаційної схеми – це сума затримок розповсюдження всіх елементів у критичному шляху. Затримка реакції – сума затримок реакції всіх елементів у найкоротшому шляху. Ці затримки продемонстровані на рис. 2.69 та описані такими рівняннями:

$$t_{PD} = 2t_{PD}_and + t_{PD}_or, \qquad (2.8)$$

$$t_{CD} = t_{CD}_and. \qquad (2.9)$$

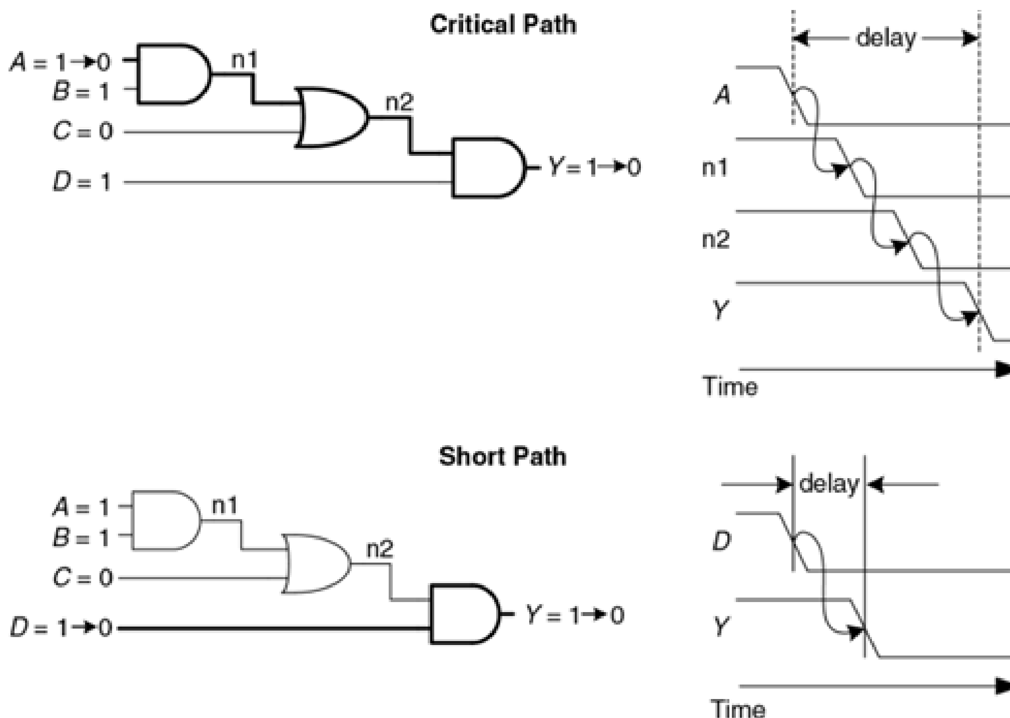


Рисунок 2.69 – Часові діаграми для найкоротшого ланцюга й ланцюга з найбільшою затримкою

Приклад 2.15.

Знаходження затримки

Бену необхідно знайти затримки поширення та відгуку схеми, яку зображено на рис. 2.70. Відповідно до довідника кожен елемент має затримку поширення 100 пікосекунд (пс) та затримку відгуку 60 пс.

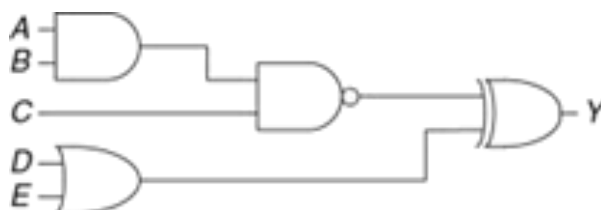


Рисунок 2.70 – Схема Бена

Виконання. Бен почав із знаходження критичного й найкоротшого шляхів у схемі. Критичний шлях, виділений на рис. 2.71, – це шлях від входу *A* чи *B*

крізь три елементи до виходу *Y*. Отже, t_{PD} – це потрійна затримка поширення одиночного елемента чи 300 пс.

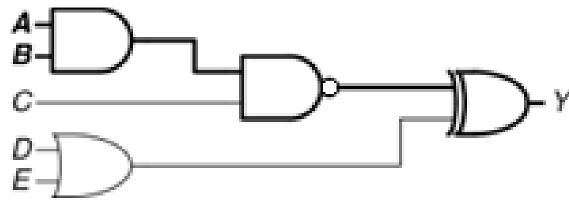


Рисунок 2.71 – Ланцюг найбільшої довжини

Найкоротший шлях (рис. 2.72) – це шлях від входів *C, D* або *E* крізь два елементи до виходу *Y*. У найкоротшому шляху тільки два елементи, так що t_{CD} дорівнює 120 пс.

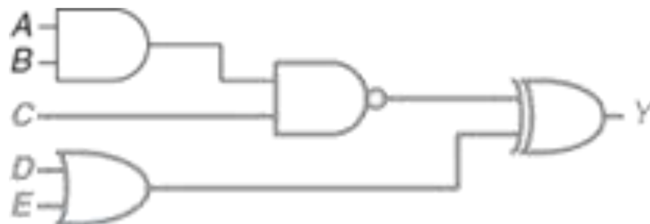


Рисунок 2.72 – Найкоротший ланцюг

Приклад 2.16.

Часові властивості мультиплексора: порівняння критичних шляхів

Порівняйте найгірші часові властивості кожної з трьох реалізацій чотиривходового мультиплексора (див. рис. 2.58, п. 2.8.1). Затримки поширення компонентів перелічені в табл. 2.7. Яким буде критичний шлях кожної реалізації? З огляду на аналіз часових властивостей якій схемі ви віддасте перевагу над іншими й чому?

Таблиця 2.7 – Часові властивості елементів у схемах мультиплексорів

Елемент	***tpd*** **(пс)**
НІ	30
Двовходовий І	60
Триходовий І	80
Чотиривходовий АБО	90
Буфер із трьома станами (від A до Y)	50
Буфер із трьома станами (від E до Y)	35

Виконання. Один з критичних шляхів для кожного з трьох варіантів зображено на рис. 2.73 та рис. 2.74. t_{PD_SY} показує затримку поширення від входу управління *S* до виходу *Y*; t_{PD_DY} – від входу даних до виходу *Y*; t_{PD} – найгірше з двох: *max* (t_{PD_SY}, t_{PD_DY}).

Як для дворівневої логіки, так і для реалізації на буферах із третім станом (див. рис. 2.73) критичним є шлях від одного із сигналів керування *S* до виходу *Y:* $t_{PD} = t_{PD_SY}$. Ця схема критична за управлінням, оскільки критичний шлях іде від сигналів управління до виходу. Будь-яка додаткова затримка в сигналах управління додасться безпосередньо в найгіршу затримку. Затримка від *D* до *Y* (рис. 2.73, *b*) – всього 50 пс порівняно із затримкою від *S* до *Y* до 125 пс.

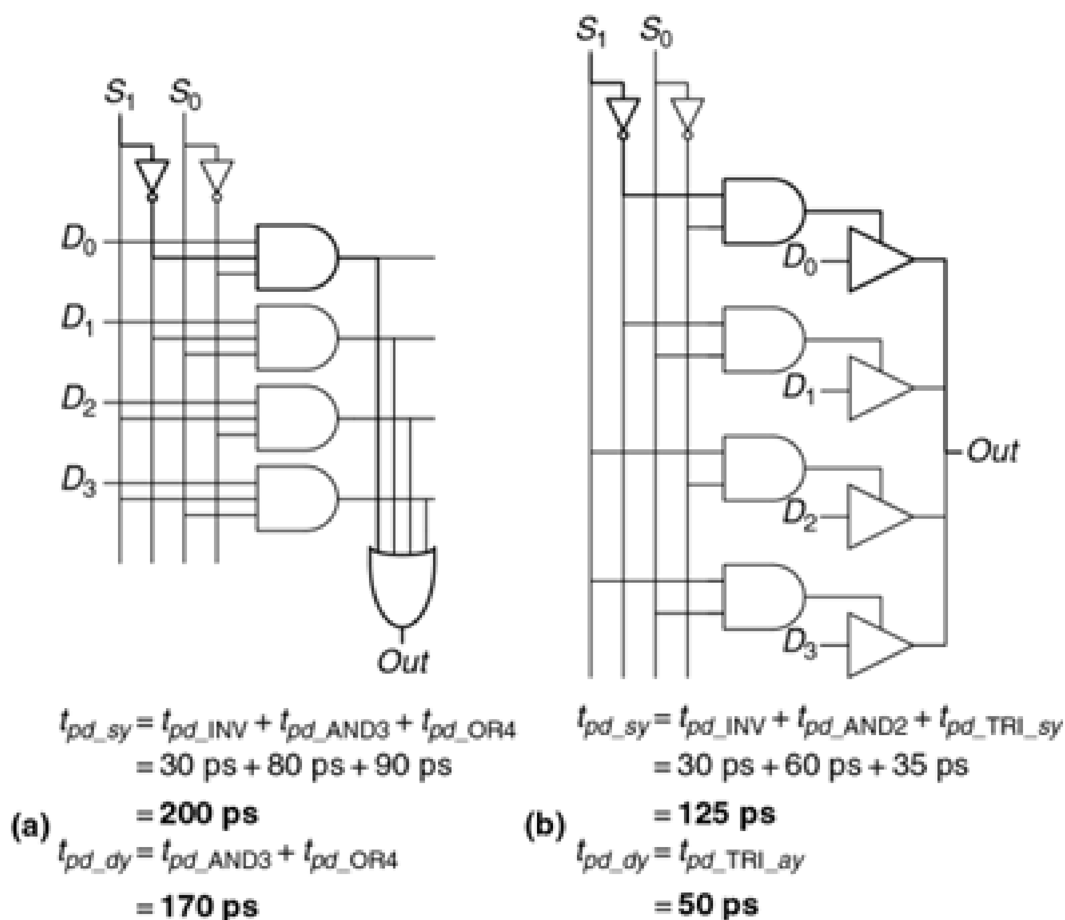


Рисунок 2.73 – Затримки поширення в чотиривходовому мультиплексорі: дворівнева логіка (а), буфера з трьома станами (b)

На рис. 2.74 показано ієрархічну реалізацію мультиплексора 4:1, що використовує два каскади мультиплексорів 2:1. Критичний шлях у ній від будь-якого входу даних *D* до виходу. Ця схема критична за даними, оскільки критичний шлях від входу даних до виходу: $t_{PD} = t_{PD_DY}$.

Якщо дані надходять на входи задовго до сигналів керування, маємо віддати перевагу схемі з найкоротшою затримкою від управління до виходу (ієрархічна схема, зображена на рис. 2.74). Аналогічно, якщо сигнали керування надходять набагато раніше, ніж вхідні дані, маємо віддати перевагу схемі з найкоротшою затримкою від даних до виходу (реалізація на буферах із третім станом на рис. 2.73, *b*).

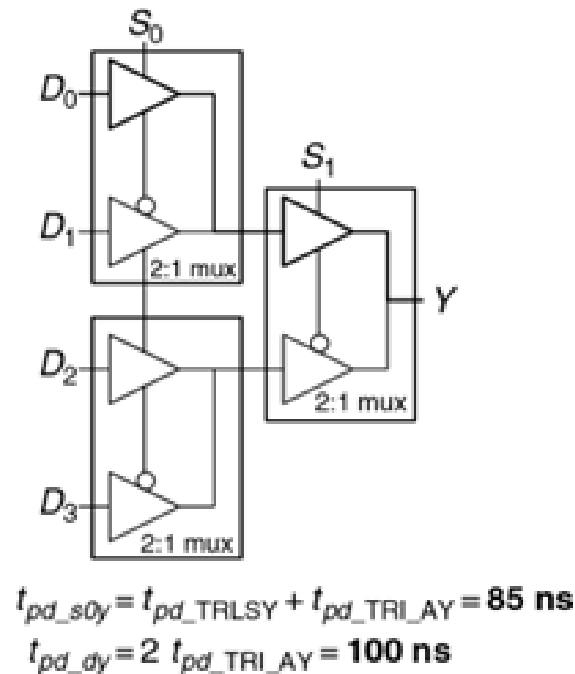


Рисунок 2.74 – Затримки поширення в чотиривходовому мультиплексорі, побудованому з двовходових

Найкращий вибір залежатиме не тільки від ланцюга з найбільшою затримкою, а й від споживання електроенергії, вартості та наявності компонентів.

2.9.2 Імпульсні перешкоди

Вище ми розглядали ситуацію, коли одиночна зміна вхідного сигналу викликає одиночну зміну виходу. Однак може виявитися, що поодинока зміна на вході викликає кілька вихідних змін. Це називається імпульсною перешкодою чи паразитним імпульсом. Хоча паразитний імпульс зазвичай не викликає проблем, важливо розуміти, що він є, і необхідно вміти розпізнавати його на часових діаграмах. На рис. 2.75 зображена схема, яка схильна до паразитних імпульсів, і карта Карно для неї.

Логічне рівняння мінімізовано коректно, проте подивіться, що відбувається, коли $A = 0$, $C = 1$ і B змінюється з 1 до 0. Рис. 2.76 ілюструє цей сценарій. Короткий шлях проходить крізь два елементи: І та АБО. Критичний шлях проходить крізь інвертор і два елементи: І та АБО.

Як тільки B зміниться з 1 в 0, $n2$ (у короткому шляху) опуститься в 0 до того, як $n1$ (у критичному шляху) зможе встановитися в 1. До підйому $n1$ обидва входи елемента АБО набуватимуть значення 0, і його вихід скинеться в 0. Коли $n1$ зрештою підніметься, Y повернеться до 1. Як показано на часових діаграмах (рис. 2.76) Y починається з 1 і закінчується 1, але на короткий час перемикається в 0.

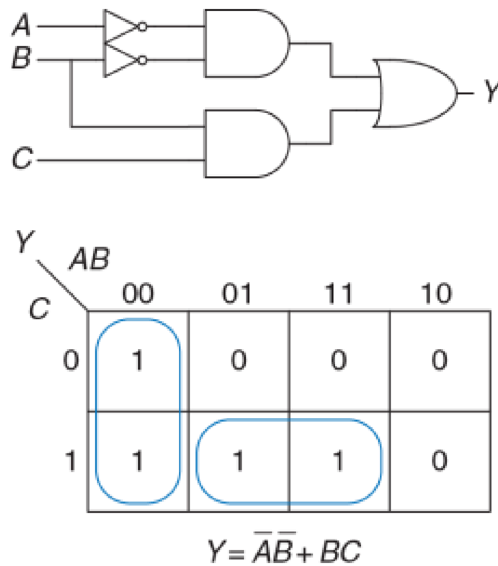


Рисунок 2.75 – Схема, що зазнає імпульсних перешкод

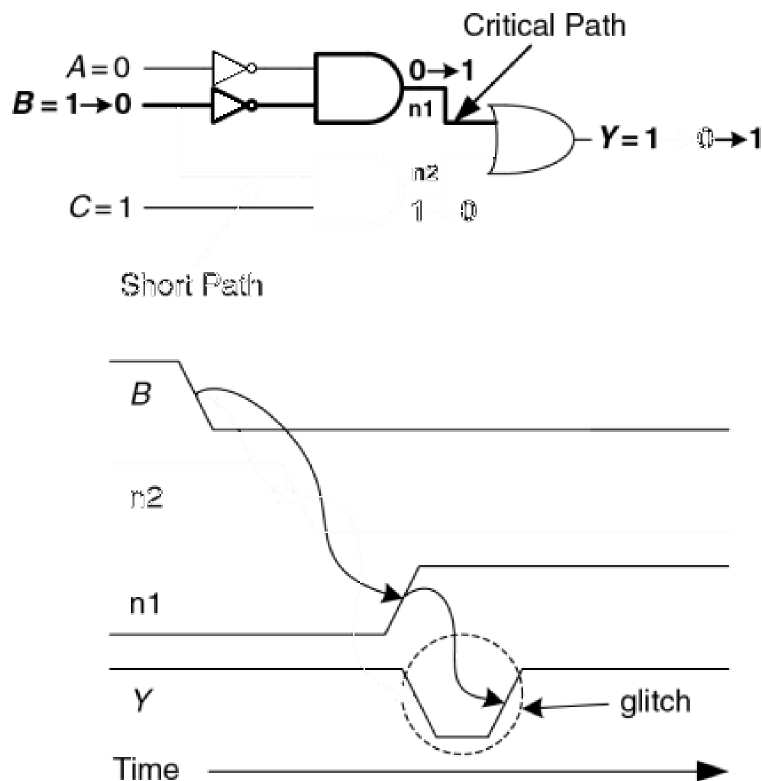


Рисунок 2.76 – Часова діаграма імпульсної перешкоди

Доки витримуємо інтервал, рівний часу затримки поширення, перш ніж використовувати значення з виходу, імпульсна перешкода не становить проблеми, оскільки вихід зрештою встановиться у правильне значення.

Якщо бажаємо, можемо уникнути цього імпульсу додаванням додаткового елемента в схему. Це простіше зрозуміти в термах карти Карно.

На рис. 2.77 зображено, як зміна входу B за умови переходу з ABC = 001 до ABC = 011 приводить до переходу від однієї первинної імпліканти до іншої. Перехід крізь межу двох первинних імплікант у карті Карно свідчить про можливу появу імпульсної перешкоди.

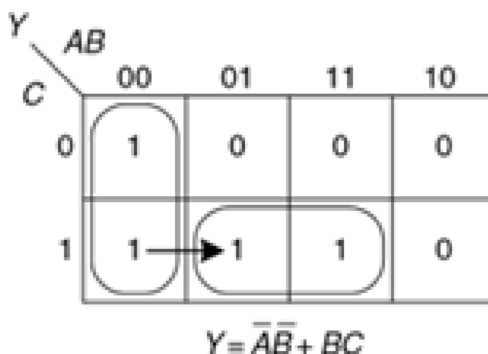


Рисунок 2.77 – Перехід від однієї імпліканти до іншої

Як бачимо на часових діаграмах (рис. 2.76), якщо схема реалізації однієї первинної імпліканти вимикається до того, як може увімкнутися схема іншої первинної імпліканти, виникне імпульсна перешкода. Щоб виправити це, ми додали інший ланцюг, що охоплює межу первинних імплікант (рис. 2.78). Ви могли б пізнати в цьому теорему узгодженості, де доданий терм $\overline{A}$ C – це узгоджений, або надлишковий терм.

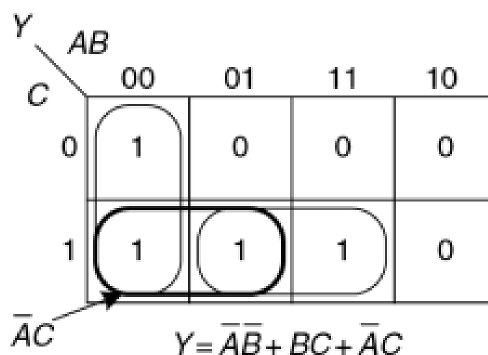


Рисунок 2.78 – Карта Карно без імпульсних перешкод

На рис. 2.79 подано схему, яка стійка до паразитних імпульсів. Зараз перемикання B, якщо A = 0 і C = 1, не викликає паразитного імпульсу на виході, оскільки доданий елемент формує на виході 1 під час цього переходу.

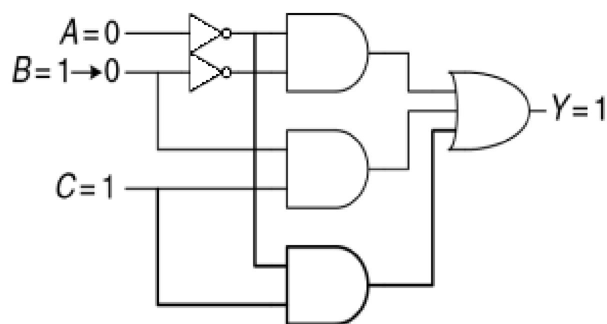


Рисунок 2.79 – Схема без імпульсних перешкод

Загалом паразитний імпульс може виникати, коли одна змінна перетинає межу між двома первинними імплікантами в карті Карно. Можемо усунути ці імпульси додаванням надлишкових імплікант у карту Карно, щоб покрити ці межі. Звичайно, це буде зроблено ціною додаткових апаратних витрат.

Однак одночасне перемикання кількох входів також може спричинити паразитні імпульси. Їх неможливо знайти додаванням додаткових елементів у схемі. Оскільки переважна більшість систем, що нас цікавлять, мають одночасні (або майже одночасні) перемикання безлічі входів, виникнення паразитних імпульсів у них неминуче. Хоча ми описали, як усунути один вид імпульсних перешкод, сенс дискусії про паразитні імпульси не в тому, щоб вилучити їх, а в тому, щоб знати про їх наявність. Це особливо важливо в аналізі часових діаграм у симуляторі або на екрані осцилографа.

2.10 Резюме

Цифрова схема – це модуль із дискретними значеннями входів та виходів і специфікацією, що описує його функціональні й часові властивості. Цей розділ присвячено комбінаційним схемам, виходи яких залежить лише від поточних значень на їх входах.

Функціональний опис комбінаційної схеми може бути задано таблицею істинності чи логічним виразом. Логічний вираз будь-якої таблиці істинності може бути отримано у вигляді досконалої диз’юнктивної нормальної форми або досконалої кон’юнктивної нормальної форми. У першому випадку функція записується як диз’юнкція кон’юнкцій, тобто булева сума (логічне АБО) однієї чи більше імплікантів. Імплікант є перемноження (логічне І) літералів. Літерали ж – це пряма чи комплементарна форма вхідних змінних.

Логічні вирази можуть бути спрощені внаслідок використання правил булевої алгебри. Зокрема їх можна спростити, об’єднуючи імпліканти,

які розрізняються лише прямою та комплементарною формами одного з літералів: $PA + P\overline{A} = P$.

Карти Карно – візуальний інструмент для мінімізації функцій двох-чотирьох змінних. Насправді розробники зазвичай можуть спростити функції кількох змінних, зважаючи лише на досвід. Системи автоматизованого проєктування застосовують для складніших функцій; такі методи та інструменти обговоримо в розділі 4.

Логічні елементи з'єднують у тому, щоб створити комбінаційну схему, яка виконує необхідну функцію. Будь-яка функція в диз'юнктивній нормальній формі може бути побудована із застосуванням дворівневої логіки: елемент НІ утворює комплементарну форму входів, елемент І формує добуток і елемент АБО формує суму. Залежно від функції та доступності базових елементів багаторівнева логічна реалізація з елементами різних типів може виявитися більш ефективною. Наприклад, для КМОН-схем більше підходять елементи І-НІ та АБО-НІ, тому що можуть бути побудовані безпосередньо на КМОН-транзисторах без використання додаткового інвертора. Коли застосовують елементи І-НІ та АБО-НІ, для скорочення кількості інверторів корисно використовувати переміщення інверсії.

Логічні елементи комбінуються, щоб створити складніші схеми, такі як мультиплексори, дешифратори та схеми пріоритету. Мультиплексор обирає один із входів даних, ґрунтуючись на вході управління. Дешифратор встановлює один із виходів у високе значення відповідно до входів. Пріоритетна схема видає 1 на вихід, що вказує на вхід із найвищим пріоритетом.

Часові властивості комбінаційної схеми передбачають затримки поширення та відгуку. Вони вказують на найбільший та найменший час між зміною входу та відповідною зміною виходів. Обчислення затримки поширення полягає у визначенні критичного шляху в схемі, потім у додаванні разом затримок поширення всіх елементів на цьому шляху. Існує безліч різних способів реалізації складної комбінаційної схеми. Ці способи припускають компроміс між швидкістю її роботи та ціною.

ВПРАВИ

Вправа 2.1. Запишіть логічний вираз у досконалій диз'юнктивній нормальній формі для всіх таблиць істинності, які наведено на рис. 2.80.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рисунок 2.80 – Таблиці істинності для вправ 2.1 та 2.3

Вправа 2.2. Запишіть логічний вираз у досконалій диз’юнктивній нормальній формі для всіх таблиць істинності, які наведено на рис. 2.81.

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

	B	C	A D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

	B	C	A D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Рисунок 2.81 – Таблиці істинності для вправ 2.2 та 2.4

Вправа 2.3. Запишіть логічний вираз у досконалій кон’юнктивній нормальній формі для всіх таблиць істинності, які наведено на рис. 2.80.

Вправа 2.4. Запишіть логічний вираз у досконалій кон’юнктивній нормальній формі для всіх таблиць істинності, які наведено на рис. 2.81.

Вправа 2.5. Мінімізуйте всі логічні вирази, отримані у вправі 2.1.

Вправа 2.6. Мінімізуйте всі логічні вирази, отримані у вправі 2.2.

Вправа 2.7. Накресліть прості комбінаційні схеми, що реалізують вирази, отримані у вправі 2.5. Під досить простою схемою мається на увазі така, що містить незначну кількість елементів, але її розробник не витрачає багато часу на перевірку кожної з можливих реалізацій схеми.

Вправа 2.8. Накресліть прості комбінаційні схеми, що реалізують вирази, отримані у вправі 2.6.

Вправа 2.9. Повторіть вправу 2.7, використовуючи лише елементи НІ, І та АБО.

Вправа 2.10. Повторіть вправу 2.8, використовуючи лише елементи НІ, І та АБО.

Вправа 2.11. Повторіть вправу 2.7, використовуючи лише елементи НІ, І-НІ та АБО.

Вправа 2.12. Повторіть вправу 2.8, використовуючи лише елементи НІ, І-НІ та АБО.

Вправа 2.13. Спростіть подані нижче логічні вирази, застосовуючи булеві теореми. Перевірте правильність результатів, використовуючи таблиці істинності або карти Карно.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + (\overline{A + \overline{C}})$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Вправа 2.14. Спростіть подані нижче логічні вирази, використовуючи булеві теореми. Перевірте правильність результатів, застосовуючи таблиці істинності або карти Карно.

(a) $Y = \overline{A}BC + \overline{A}B\overline{C}$

(b) $Y = \overline{ABC} + A\overline{B}$

(c) $Y = ABC\overline{D} + A\overline{BCD} + (\overline{A + B + C + D})$

Вправа 2.15. Накресліть прості комбінаційні схеми, що реалізують вирази, отримані у вправі 2.13.

Вправа 2.16. Накресліть прості комбінаційні схеми, що реалізують вирази, отримані у вправі 2.14.

Вправа 2.17. Спростіть кожний із наступних логічних виразів. Накресліть прості комбінаційні схеми, що реалізують отримані вирази.

(a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D$
$+ \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Вправа 2.18. Спростіть кожний із поданих нижче логічних виразів. Накресліть прості комбінаційні схеми, що реалізують отримані вирази.

(a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$

(b) $Y = \overline{(A + B + C)}D + AD + B$

(c) $Y = ABCD + \overline{A}B\overline{C}D + \overline{(\overline{B} + D)E}$

Вправа 2.19. Наведіть приклад таблиці істинності, що містить від 3 млрд до 5 млрд рядків, яка може бути реалізована схемою, що використовує менше ніж 40 двовходових логічних елементів (але не менше ніж один).

Вправа 2.20. Наведіть приклад схеми з циклічним шляхом, яка є комбінаційною.

Вправа 2.21. Аліса Хакер стверджує, що будь-який логічний вираз може бути записано у вигляді мінімальної диз'юнктивної нормальної форми, тобто у вигляді булевої суми простих імплікант. Бен Бітдідл наголошує, що існують такі вирази, мінімальні форми яких не містять усіх простих імплікант. Поясніть, чому Аліса має рацію, або наведіть контрприклад, що підтверджує думку Бена.

Вправа 2.22. Доведіть подані нижче теореми, використовуючи досконалу індукцію. Вам не потрібно доводити подвійні їм теореми.

a) Теорема про ідемпотентність (T3).

b) Теорема дистрибутивності (T8).

c) Теорема склеювання (T10).

Вправа 2.23. Доведіть теорему де Моргана (T12) для трьох змінних, застосовуючи досконалу індукцію.

Вправа 2.24. Напишіть логічні вирази для схеми, поданої на рис. 2.82. Ви не маєте мінімізувати ці вирази.

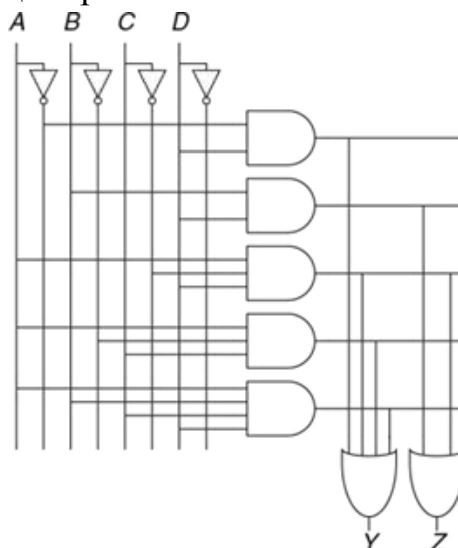


Рисунок 2.82 – Принципова схема

Вправа 2.25. Мінімізуйте логічні вирази, отримані у вправі 2.24, та накресліть удосконалену схему, що реалізує ці функції.

Вправа 2.26. Використовуючи елементи, еквівалентні де Моргану, і метод переміщення інверсії, перекресліть схему, зображену на рис. 2.83, щоб ви могли знайти її логічний вираз «на око». Запишіть цей логічний вираз.

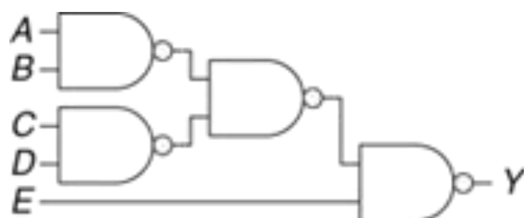


Рисунок 2.83 – Принципова схема

Вправа 2.27. Повторіть вправу 2.26 для схеми (рис. 2.84).

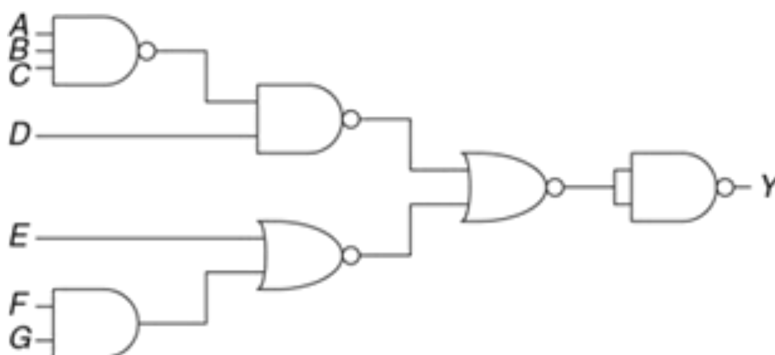


Рисунок 2.84 – Принципова схема

Вправа 2.28. Знайдіть мінімальний логічний вираз функції, яку задано на рис. 2.85. Не забудьте в цьому разі скористатися наявністю байдужих значень таблиці істинності.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рисунок 2.85 – Таблиця істинності

Вправа 2.29. Накресліть схему, що реалізує функцію, яку отримано у вправі 2.28.

Вправа 2.30. Чи можуть у схемі з вправи 2.29 з'явитися потенційні паразитні імпульси за умови зміни стану одного із входів? Якщо ні, поясніть чому. Якщо так, покажіть, як необхідно змінити схему, щоб усунути паразитні імпульси.

Вправа 2.31. Знайдіть мінімальний логічний вираз функції, заданої на рис. 2.86. Не забудьте в цьому разі скористатися наявністю байдужих значень таблиці істинності.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рисунок 2.86 – Таблиця істинності

Вправа 2.32. Накресліть схему, що реалізує функцію, отриману у вправі 2.31.

Вправа 2.33. Бен Бітдідл насолоджуватиметься пікніком у сонячний день, якщо не буде мурах. Він також із задоволенням відпочиватиме на пікніку в будь-який день, якщо побачить колібрі, а також у ті дні, коли сонячно і є мурахи.

Запишіть логічний вираз для Бенової радості (E) у термінах наявності сонця (S), мурах (A), колібрі (H) та сонечка (L).

Вправа 2.34. Завершіть проєктування дешифратора семисегментного індикатора сегментів від Sc до Sg (див. приклад 2.10).

а) Виведіть логічний вираз для виходів від Sc до Sg за умови, що в процесі подачі на вхід значення понад 9 вихід має бути нулем.

b) Виведіть логічний вираз для виходів від *Sc* до *Sg* за умови, що в процесі подачі на вхід значення понад 9 стан виходу байдужий.

c) Накресліть досить просту реалізацію лише на рівні логічних елементів для випадку (b). За потреби використовуйте спільні логічні елементи для кількох виходів.

Вправа 2.35. Схема має чотири входи та два виходи. На входи *A*3:0 подається число від 0 до 15. Вихід *P* має дорівнювати ІСТИНІ, якщо число на вході просте (0 і 1 не є простими, а 2, 3, 5 і так далі – є). Вихід *D* має дорівнювати ІСТИНІ, якщо число ділиться на 3. Запишіть спрощений логічний вираз для кожного з виходів і накресліть схему.

Вправа 2.36. Пріоритетний шифратор має 2^N входи. Він формує на *N*-розрядному виході номер найстаршого вхідного біта, що набуває значення ІСТИНА. Він також формує на виході *NONE* значення ІСТИНА, якщо жоден із входів не набуває значення ІСТИНА. Спроєктуйте восьмивходовий пріоритетний шифратор із входом *A*7:0 та виходами *Y*2:0 та *NONE*.

Наприклад, якщо вхід *A* набуває значення 00100000, вихід *Y* має бути 101, а *NONE* – 0. Запишіть спрощений логічний вираз для кожного з виходів і накресліть схему.

Вправа 2.37. Спроєктуйте модифікований пріоритетний шифратор (див. вправу 2.36), який має восьмирозрядний вхід *A*7:0, а також трирозрядні виходи *Y*2:0 та *Z*2:0. На виході *Y* формується номер найстаршого вхідного біта, що набуває значення ІСТИНА. На виході *Z* формується номер другого відповідно до старшинства вхідного біта, що набуває значення ІСТИНА. *Y* набуває значення 0, якщо всі біти входу – *FALSE*. *Z* набуває значення 0, якщо один біт входу – ІСТИНА. Запишіть спрощений логічний вираз для кожного з виходів і накресліть схему.

Вправа 2.38. *M*-бітний унарний код числа k містить k одиниць у молодших розрядах і *M*–k нулів у всіх старших розрядах. Перетворювач бінарного коду в унарний має N входів та 2^N–1 виходів. Він формує (2^N-1)-бітний унарний код для числа, встановленого на вході. Наприклад, якщо на вході 110, то на виході має бути 0111111. Спроєктуйте перетворювач трибітного бінарного коду в семибітний унарний. Запишіть логічний вираз кожного з виходів і накресліть схему.

Вправа 2.39. Запишіть мінімізований логічний вираз для функції, яку виконує схема, зображена на рис. 2.87.

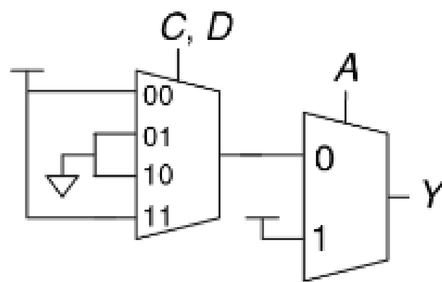


Рисунок 2.87 – Схема на мультиплексорах

Вправа 2.40. Запишіть мінімізований логічний вираз для функції, яку виконує схема, зображена на рис. 2.88.

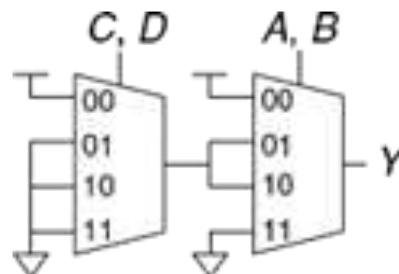


Рисунок 2.88 – Схема на мультиплексорах

Вправа 2.41. Спроєктуйте схему, що реалізує функцію (див. рис. 2.80, *b*), використовуючи:

a) восьмивходовий мультиплексор (8:1);

b) чотиривходовий мультиплексор (4:1) та один інвертор;

c) двовходовий мультиплексор (2:1) та два будь-які інші логічні елементи.

Вправа 2.42. Спроєктуйте схему, що реалізує функцію з вправи 2.17, *a*, використовуючи:

a) восьмивходовий мультиплексор (8:1);

b) чотиривходовий мультиплексор (4:1) без інших логічних елементів;

c) двовходовий мультиплексор (2:1), один елемент АБО та один інвертор.

Вправа 2.43. Розрахуйте затримку поширення *tpd* та затримку реакції *tcd* для схеми, зображеної на рис. 2.83. Значення затримок елементів подано в табл. 2.8.

Вправа 2.44. Розрахуйте затримку поширення та затримку реакції для схеми, зображеної на рис. 2.84. Значення затримок елементів подано в табл. 2.8.

Вправа 2.45. Накресліть схему швидкодіючого дешифратора 3:8. Значення затримок елементів подано в табл. 2.8 (використовуйте лише зазначені в таблиці елементи). Спроєктуйте дешифратор таким чином, щоб він мав мінімально можливий критичний шлях і знайдіть цей шлях. Які затримки поширення та реакції?

Таблиця 2.8 – Значення затримок елементів для вправ 2.43–2.47

Елемент	***tpd*** **(пс)**	***tcd*** **(пс)**
НІ	15	10
двовходовий І-НІ	20	15
тривходовий І-НІ	30	25
двовходовий І АБО-НІ	30	25
двовходовий АБО-НІ	45	35
двовходовий І	30	25
двовходовий І	40	30
двовходовий АБО	40	30
двовходовий АБО	55	45
двовходовий виключний АБО	60	40

Вправа 2.46. Змініть схему із вправи 2.35 так, щоб вона була з максимальною швидкодією. Використовуйте лише елементи з табл. 2.8. Накресліть нову схему та визначте критичний шлях. Які затримки поширення та реакції?

Вправа 2.47. Змініть пріоритетний дешифратор із вправи 2.36 так, щоб він працював максимально швидко. Використовуйте лише елементи з табл. 2.8. Накресліть нову схему та визначте критичний шлях. Які затримки поширення та реакції?

Вправа 2.48. Спроєктуйте восьмивходовий мультиплексор так, щоб затримка від входів до виходів була мінімальною. Використовуйте лише елементи з табл. 2.7. Накресліть схему. Використовуючи значення затримок елементів із таблиці, визначте затримку від входів до виходів.

ЗАПИТАННЯ ДЛЯ СПІВБЕСІДИ

Подаємо приклади типових запитань,
які можуть бути поставлені претендентам
під час пошуку роботи
в галузі проєктування цифрових пристроїв.

Запитання 2.1. Накресліть схему, що реалізує функцію «що виключає АБО», використовуючи логічні елементи І-НІ. Яка мінімальна кількість елементів І-НІ для цього необхідна?

Запитання 2.2. Спроєктуйте схему, яка показує, чи містить заданий місяць 31 день. Місяць задається чотирирозрядним входом $A3{:}0$. Наприклад, значенню 0001 на вході відповідає січень, а значенню 1100 – грудень. Вихід схеми Y має набувати значення ІСТИНА тільки тоді, коли на вхід подано номер місяця, у якому 31 день. Напишіть спрощений вираз та накресліть схему, використовуючи мінімальну кількість елементів. (*Підказка: не забудьте скористатися байдужими станами*.)

Запитання 2.3. Що таке буфер із трьома станами? Як і навіщо він застосовується?

Запитання 2.4. Елемент чи набір елементів є універсальним, якщо може бути використаний для реалізації будь-якої логічної функції. Наприклад, набір {І, АБО, НІ} є універсальним.

a) Чи є елемент І універсальним? Чому?

b) Чи є набір елементів {АБО, НІ} універсальним? Чому?

c) Чи є елемент І-НІ універсальним? Чому?

Запитання 2.5. Чому затримка реакції схеми може бути меншою або дорівнює затримці поширення?

3 ПРОЄКТУВАННЯ ПОСЛІДОВНОЇ ЛОГІКИ

3.1 Вступ

У попередньому розділі ми розглянули процес аналізу та проєктування комбінаційних логічних схем. Значення на виході комбінаційної схеми залежить лише від значень на вході зараз. Можемо створити оптимізовану схему згідно з технічним завданням у вигляді таблиці істинності або логічного виразу.

У цьому розділі проаналізуємо та спроєктуємо послідовні логічні схеми. Значення на виході послідовної логічної схеми залежить як від поточних, так і від попередніх вхідних значень. Отже, послідовні логічні схеми мають пам’ять і можуть явно запам’ятовувати попередні значення певних входів, а можуть «стискати» попередні значення певних входів у меншу кількість інформації, яка називається станом системи. Стан цифрової послідовної схеми – набір біт, так звані змінні стану. Ці біти містять всю інформацію про минуле, необхідну для визначення майбутньої поведінки схеми.

Матеріал починається з вивчення засувок та тригерів. Вони є простими послідовними схемами, що запам’ятовують один біт інформації. Загалом, послідовні схеми досить складно аналізувати. З метою спрощення проєктування ми обмежимося лише синхронними схемами, які передбачають комбінаційну логіку та набір тригерів, що зберігають інформацію про стан системи. У розділі описуються кінцеві автомати, за допомогою яких можна легко проєктувати послідовні схеми. Крім того, проаналізуємо швидкодію послідовних схем та обговоримо паралельні обчислення як спосіб підвищення швидкодії [1].

3.2 Засувки й тригери

Основним блоком для побудови пам’яті є бістабільний осередок – елемент із двома стійкими станами. На рис. 3.1, *a* зображено простий бістабільний осередок, що містить пару інверторів, замкнутих у кільце. Цю схему можна переробити так, щоб рисунок виглядав симетрично (рис. 3.1, *b*). Тепер видно, що інвертори з’єднані перехресно, тобто вхід $I1$ з’єднано з виходом $I2$ і навпаки. У схеми немає жодного входу, проте є два виходи Q і $\overline{Q}$. Аналіз цієї схеми різниться від аналізу комбінаційної схеми, оскільки схема є циклічною: Q залежить від $\overline{Q}$, а $\overline{Q}$ залежить від Q.

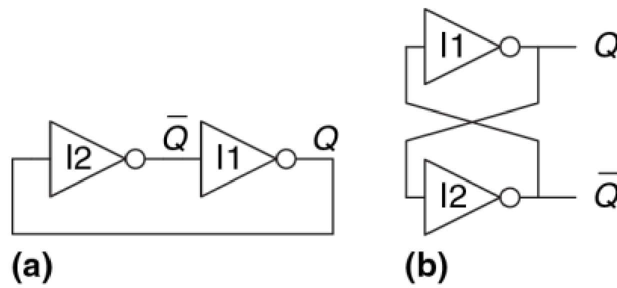


Рисунок 3.1 – Перехресно з’єднані інвертори

Розглянемо два випадки: Q = 0 та Q = 1

● Випадок I: Q = 0.

Як зображено на рис. 3.2, *a*, на вхід *I*2 надходить сигнал Q = 0. *I*2 інвертує сигнал і подає на вхід *I*1 сигнал $\overline{Q}$ = 1. Відповідно, на виході *I*1 – логічний 0. У розглянутому випадку схема перебуває в стабільному стані.

● Випадок II: Q = 1.

Як зображено на рис. 3.2, *b*, на вхід *I*2 надходить 1 (Q). *I*2 інвертує сигнал і подає на вхід *I*1 0 $\overline{Q}$. Відповідно, на виході *I*1 – логічна 1. І тут схема також перебуває в стабільному стані.

Оскільки інвертори, увімкнені перехресно, мають два стабільні стани Q = 0 і Q = 1, то кажуть, що схема бістабільна. У схеми є і третій стан, коли обидва виходи перебувають у стані між 0 і 1. Такий стан називається метастабільним, і його розглянемо в п. 3.5.4.

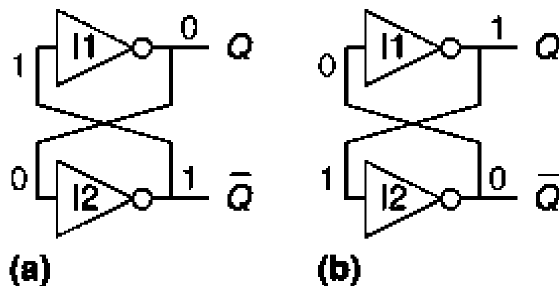


Рисунок 3.2 – Бістабільний режим перехресно з’єднаних інверторів

Елемент із *N* стабільними станами зберігає *log2N* бітів інформації. Отже, бістабільний осередок зберігає 1 біт. Стан перехресно увімкнених інверторів подається значенням Q. Значення Q повідомляє нам всю інформацію про минуле, необхідну для визначення майбутньої поведінки схеми. Зокрема, якщо Q = 0, то він і завжди буде 0, а якщо Q = 1, тоді й залишиться 1. У схеми є ще один вихід – $\overline{Q}$. Але $\overline{Q}$ не містить жодної додаткової інформації, оскільки

якщо Q відомо, то $\overline{Q}$ визначено однозначно. З іншого боку, $\overline{Q}$ можна було б також розглядати як змінну стану.

За умови увімкнення живлення вихідний стан послідовної схеми невідомий і зазвичай непередбачуваний. Він може бути різним щоразу, коли схему вмикають.

Незважаючи на те, що перехресно увімкнені інвертори можуть зберігати біт інформації, вони не використовуються на практиці, оскільки схема не має входів, за допомогою яких користувач міг би контролювати її стан. Однак інші елементи, зокрема засувки та тригери, мають входи, що дають змогу керувати змінною стану. Ці схеми проаналізуємо нижче в межах цього розділу.

3.2.1 RS-тригер

Однією з найпростіших послідовних схем є *RS*-тригер (від англ. *Reset* і *Set*), що має два перехресно увімкнених елементи АБО-НІ (рис. 3.3). У засувки є два входи (R та S) і два виходи (Q та $\overline{Q}$). Принципи роботи *RS*-тригера та схеми з перехресно увімкненими інверторами аналогічні, але стан засувки контролюються входами R і S, які скидають та встановлюють вихід Q [1].

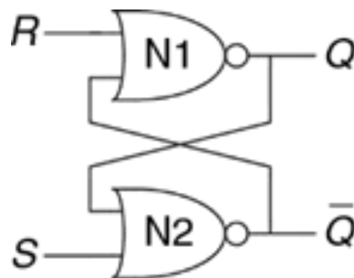


Рисунок 3.3 – *RS*-тригер (засувка)

Щоб зрозуміти, як працює невідомий ланцюг, зазвичай будують його таблицю істинності. Пригадаємо, що у виході елемента АБО-НІ з’являється логічний нуль, якщо на якийсь із його входів подана логічна одиниця. Розглянемо чотири можливі комбінації R і S.

● Випадок I: $R = 1$, $S = 0$.

На вході $N1$ як мінімум одна одиниця – вхід R, отже, вихід $Q = 0$. Обидва входи $N2$ – у стані логічного нуля ($Q = 0$ і $S = 0$), тому вихід $\overline{Q} = 1$.

● Випадок II: $R = 0$, $S = 1$.

На вхід $N1$ надходить 0 і $\overline{Q}$. Оскільки ми поки не знаємо значення $\overline{Q}$, то не можемо визначити значення Q. На вхід $N2$ надходить як мінімум одна одиниця S, тому на виході $\overline{Q}$ нуль. Тепер можна повернутися до стану виходу елемента $N1$. Ми знаємо, що на обох його входах 0, отже, $Q = 1$.

● Випадок III: $R = 1$, $S = 1$.

Як на вході $N1$, так і на вході $N2$ як мінімум по одній одиниці (R та S), тому на виході кожної засувки – логічний 0. Отже, $Q = 0$ і $\overline{Q} = 0$.

● Випадок IV: $R = 0$, $S = 0$.

На вхід $N1$ надходить 0 і $\overline{Q}$. Оскільки ми не знаємо значення $\overline{Q}$, то не можемо визначити значення на виході елемента $N1$. На вхід $N2$ надходить 0 і Q. Оскільки ми ще не знаємо значення Q, відповідно, не можемо встановити значення на виході елемента $N2$. Здається, ми зайшли в глухий кут. Цей випадок аналогічний випадку з двома перехресно увімкненими інверторами. Ми знаємо, що Q має дорівнювати або 0, або 1. Отже, зможемо розв'язати проблему, якщо розглянемо кожен із цих двох випадків.

● Випадок IV, *a*: $Q = 0$.

Оскільки S і Q дорівнюють 0, то на виході $N2$ буде логічна 1, $\overline{Q} = 1$, як подано на рис. 3.4, а. Тепер на вході $N1$ є одна одиниця – $\overline{Q}$, тому на його виході $Q = 0$, як і передбачали.

● Випадок IV, *b*: $Q = 1$.

Оскільки $Q = 1$, то на виході $N2$ буде 0, $\overline{Q} = 0$, як подано на рис. 3.4, *b*. Тепер на обох входах $N1$ нулі (R та $\overline{Q}$), тому на його виході логічна 1, $Q = 1$, як і передбачали.

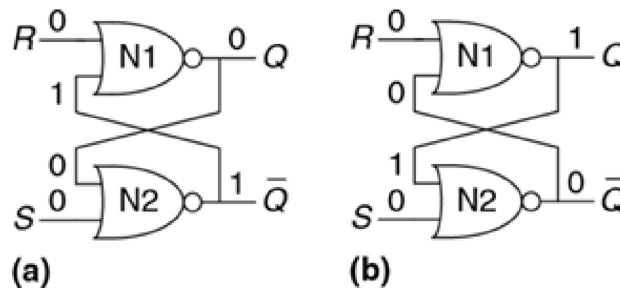


Рисунок 3.4 – Бістабільні стани *RS*-тригера

З огляду на сказане вище, припустимо, що Q – певне значення, встановлене до настання випадку IV, яке назвемо Q_{PREV} і яке може бути або 0, або 1. Q_{PREV} позначає стан системи. Коли R і S дорівнюють 0, на виході Q зберігатиметься старе значення Q_{PREV}, а $\overline{Q}$ буде його булевим доповненням.

Таблиця істинності (див. рис. 3.5) ілюструє ці чотири випадки. Входи R та S відповідають за скидання та встановлення значень відповідно.

Встановити біт – значить перевести його в логічну одиницю, а скинути – у логічний нуль. Зазвичай $\overline{Q}$ є булевим доповненням Q. Коли надходить команда скидання $R = 1$, вихід Q набуває значення 0, а вихід $\overline{Q}$ – протилежне

(логічна одиниця). Коли надходить команда втановлення біта $S = 1$, вихід Q стає одиницею, а $\overline{Q}$ – нулем. Якщо на жодний із входів не надходить логічна одиниця, на обох виходах зберігається попереднє значення Q_{PREV}. Подання на входи одночасно $R = 1$ і $S = 1$ не має особливого сенсу, оскільки це означає, що вихід має бути одночасно і встановлений, і скинутий, що неможливо. Засувка, не знаючи, що їй робити, встановлює як на прямому, так і на інверсному виході логічний 0.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Рисунок 3.5 – Таблиці істинності *RS*-тригера

Умовна познака *RS*-тригера подана на рис. 3.6. Умовні познаки використовуються в модульному проєктуванні схеми з метою абстрагування внутрішньої структури елемента.

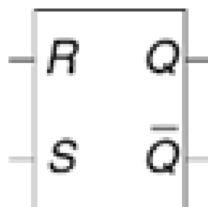


Рисунок 3.6 – Зображення *RS*-тригера

Існує кілька способів побудови *RS*-тригера, наприклад використання логічних елементів або транзисторів. Проте будь-який елемент схеми, специфікований таблицею істинності (див. рис. 3.5), позначається символом (рис. 3.6) і називається *RS*-тригером.

Так само як і перехресно увімкнені інвертори, *RS*-тригер є бістабільним елементом з одним бітом стану, що зберігається в Q. Станом можна управляти за допомогою входів R та S. Коли на R надходить високий рівень, вихід скидається в 0. Коли високий рівень приходить на S, вихід встановлюється в 1. Якщо на жодний вхід не прийшла логічна одиниця, тригер зберігає свій попередній стан, значення виходів не змінюється. Зазначимо, що вся історія сигналів, поданих на вхід, може бути зосереджена в одній змінній стану Q. Не має значення, що відбувалося раніше. Усе, що потрібно, щоб передбачити подальшу поведінку *RS*-тригера, – це знати, чи остання зміна стану тригера була скиданням чи установкою.

3.2.2 D-засувка

RS-тригер незручний через незвичайну поведінку, якщо на обидва входи тригера одночасно надходить високий рівень сигналу. Більш серйозна проблема полягає в тому, що питання ЩО й КОЛИ у змінних стану тригера об'єднані його *R*- та *S*-входами. Подача логічної одиниці на ці входи визначає не тільки, ЩО станеться, а й КОЛИ це відбудеться. Розроблення схем спрощується, якщо питання ЩО та КОЛИ розділені. *D*-тригер-засувка (рис. 3.7, *a*) розв'язує ці проблеми. Тригер має два входи: вхід даних *D*, що визначає, яким буде наступний стан, і вхід тактового сигналу *CLK*, що визначає, коли він зміниться.

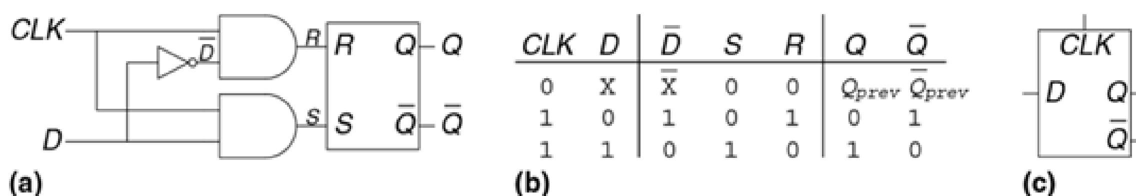


Рисунок 3.7 – *D*-тригер-засувка: a – схема; b – таблиця істинності; c – познака

Для аналізу засувки знову створемо таблицю істинності (рис. 3.7, *b*). Спочатку розглянемо внутрішні лінії *D, R* і *S*. Якщо *CLK* = 0, то обидва сигнали *R* та *S* нульові, незалежно від значення *D*. Якщо *CLK* = 1, на виході одного елемента І буде одиниця, а на виході іншого – нуль. Елемент І, на виході якого буде 1, визначається входом *D*. Значення *Q* та $\overline{Q}$ визначаються як *R* і *S* відповідно до таблиці, поданої на рис. 3.5. Зауважимо, що доки *CLK* = 0, *Q* зберігає попереднє значення *Q*попр. Якщо *CLK* = 1, *Q* = *D*. Очевидно, що $\overline{Q}$ завжди є булевим доповненням *Q*. У *D*-засувці унеможливлений випадок незвичайної поведінки в разі одночасно поданих сигналах скидання та установлення (*R* = 1 і *S* = 1).

Отже, бачимо, що тактовий сигнал контролює, КОЛИ дані проходять крізь тригер. Коли *CLK* = 1, засувка «прозора», тобто пропускає дані *D* на вихід *Q*, якби він був звичайним буфером. Якщо *CLK* = 0, засувка буде «непрозора», тобто не пропускає нові дані з входу *D* на вихід *Q*, а *Q* зберігає своє значення. *D*-тригер іноді називають прозорим тригером, або тригером, синхронізованим рівнем. Умовна позначка *D*-засувки зображена на рис. 3.7, *c*.

Стан *D*-тригера-засувки змінюється безперервно, поки *CLK* = 1. Нижче в цьому розділі побачимо, що найчастіше зручно змінювати стан схеми лише в певний момент часу. У наступному пункті описано *D*-тригер, синхронізований фронтом.

Іноді стан засувки називають «відкритим» або «закритим», а не «прозорим» або «непрозорим».

3.2.3 D-Тригер

D-тригер, або синхронізований фронтом тригер (далі – тригер), може бути побудований з двох увімкнених послідовно *D*-засувок. Як подано на рис. 3.8, *a*, тактові сигнали, що подаються на них, є булевими доповненнями один одного. Першу засувку називають основною (*master*), а другу – підпорядкованою (*slave*). Засувки з'єднані лінією *N*1. Умовна познака *D*-тригера зображена на рис. 3.8, *b*. Коли вихід $\overline{Q}$ не використовується, познака може бути спрощена до вигляду, поданого на рис. 3.8, *c*.

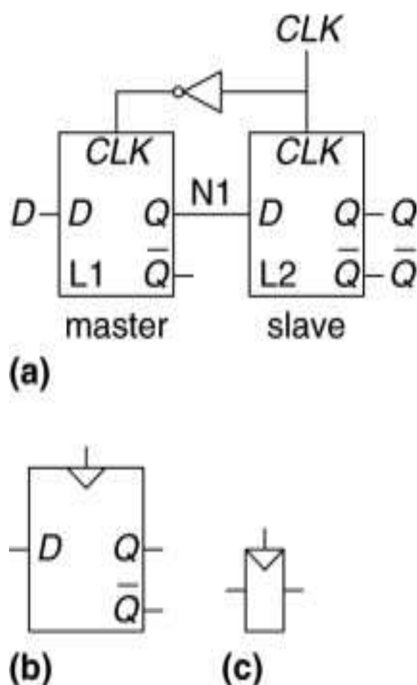


Рисунок 3.8 – *D*-тригер: а – схема; b – познака; c – спрощена познака

Коли *CLK* = 0, *master*-засувка відкрита, а *slave*-засувка закрита. Отже, значення входу *D* проходить до лінії *N*1. Коли *CLK* = 1, *master*-засувка закривається, а *slave*-засувка відкривається. Значення *N*1 проходить на вихід *Q*, але *N*1 стає відрізаним від *D*. Отже, те значення, що було на вході *D* безпосередньо перед переходом *CLK* з 0 в 1, відразу ж потрапляє на вихід *Q* після того, як тактовий сигнал встановлюється в 1. Решта *Q* зберігає своє колишнє значення, оскільки закритий тригер постійно блокує шлях між *D* та *Q*.

Іншими словами, *D*-тригер копіює значення з *D* на *Q* по передньому фронту тактового імпульсу й пам'ятає цей стан решту часу. ***Перечитайте це визначення, доки ви його не запам'ятаєте.*** Одна з найпоширеніших помилок розробників-початківців цифрових схем – вони забувають, що таке синхронізація фронтом. Часто передній фронт тактового імпульсу називають просто фронтом. Вхід *D* визначає новий, подальший стан тригера. Фронт визначає час, коли стан буде оновлено.

D-тригер також відомий як *MS*-тригер, *master-slave*-тригер і як тригер, що синхронізується фронтом. Трикутник у познаці свідчить про те, що вхід синхронізується фронтом. У багатьох тригерів вихід $\overline{Q}$ відсутній, і їх зазвичай використовують, коли $\overline{Q}$ не потрібен.

Приклад 3.1.

Кількість транзисторів у тригері

Скільки транзисторів міститься в *D*-тригері, описаному в цьому розділі?

Виконання. В елементі АБО-НІ або І-НІ використовується по чотири транзистори, в інверторі – два транзистори. Елемент І містить елементи І-НІ та НІ (інвертора), тому в ньому використовуються шість транзисторів. *RS* має два елементи АБО-НІ або вісім транзисторів. У *D*-засувки використовується *RS*-засувка, два елементи І і один елемент НІ або 22 транзистори. У *D*-тригері застосовується дві *D*-засувки та один елемент НІ або 46 транзисторів. У п. 3.2.7 описуються більш ефективні способи реалізації тригера на основі КМОН-технології з використанням прохідних ключів.

3.2.4 Регістр

N-розрядний регістр – набір із *N* тригерів із загальним тактовим сигналом. Отже, всі біти регістра оновлюються одночасно. Регістр є ключовим блоком у побудові більшості схем послідовності. На рис. 3.9 подано схему та познаку чотирирозрядного регістра з входами *D*3:0 та виходами *Q*3:0. *D*3:0 та *Q*3:0 є чотирирозрядними шинами.

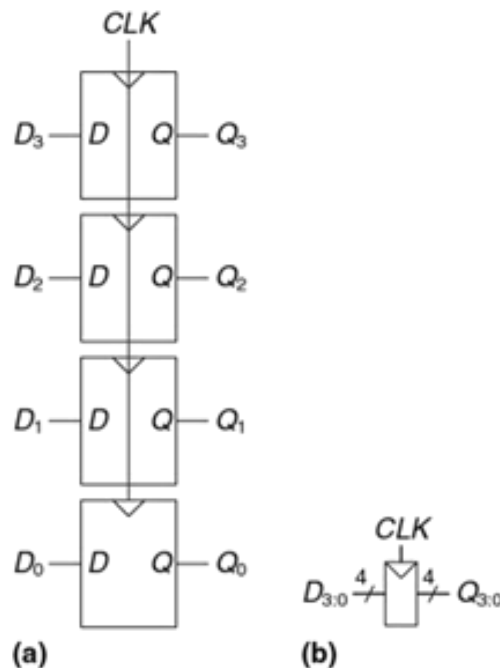


Рисунок 3.9 – Чотирирозрядний регістр: a – схема; b – познака

3.2.5 Тригер із функцією дозволу

У деяких тригерів є ще один вхід, так званий *EN*, або *ENABLE* (дозволити). Цей вхід визначає, чи будуть дані завантажені фронтом чи ні. Коли на *EN* подається логічна одиниця, то такий *D*-тригер поводиться так само, як і звичайний *D*-тригер. Якщо ж на *EN* надходить логічний нуль, то тригер ігнорує тактовий сигнал і зберігає свій стан. Такі тригери корисні, якщо хочемо завантажувати значення тригера лише протягом якогось часу, а не по кожному фронту тактового імпульсу.

На рис. 3.10 подано два способи додавання дозволу входу до звичайного *D*-тригера.

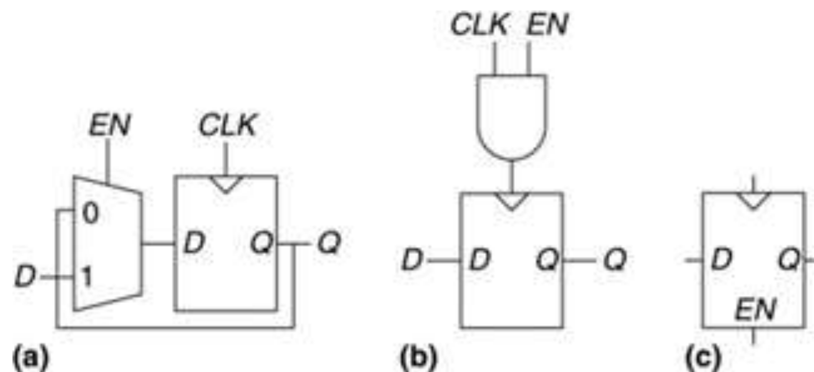


Рисунок 3.10 – Тригер із функцією дозволу: a, b – схеми; c – позначка

На рис. 3.10, *a* вхідний мультиплексор обирає, чи подавати дані на вхід *D*, якщо *EN* – логічна одиниця, або подавати на вхід *D* старе значення з виходу *Q*, якщо на *EN* подається логічний нуль. На рис. 3.10, *b* тактовий сигнал проходить, якщо *EN* дорівнює одиниці; імпульси на вхід тактового сигналу подаються у звичайному режимі. Якщо *EN* – логічний нуль, тоді й на *CLK* також нуль, і тригер зберігає свій попередній стан. Зауважимо, що сигнал *EN* має змінюватися, поки *CLK* = 1, щоб уникнути збою (викиду) тактового сигналу (перемикання у неправильний час). Узагалі, додавання логічних елементів до тракту тактування – погана ідея. Управління тактуванням спричиняє затримку в тактовому сигналі й може призвести до часових помилок, про що буде зазначено в п. 3.5.3. Тобто рекомендуємо робити так лише в тому разі, якщо впевнені в своїх діях. Познаку тригера з функцією дозволу зображено на рис. 3.10, *c*.

3.2.6 Тригер із функцією скидання

У тригері з функцією скидання додається ще один вхід, що називається *RESET* (скидання). Коли на *RESET* подано 0, тригер поводиться як звичайний *D*-тригер. Коли на *RESET* подано 1, такий тригер ігнорує вхід *D* і скидає

вихід у 0. Тригери з функцією скидання корисні, коли хочемо прискорити встановлення певного стану (тобто 0) у всіх тригерах системи під час першого увімкнення.

Такі тригери можуть скидатися як синхронно, так і асинхронно. Тригери, що скидаються синхронно, скидаються тільки по фронту сигналу *CLK*. Тригери, що скидаються асинхронно, роблять це відразу ж за умови надходження логічної одиниці на вхід *RESET*, незалежно від тактового сигналу.

На рис. 3.11, *a* продемонстровано, як побудувати тригер, що синхронно скидається, із звичайного D-тригера і елемента І. Коли на $\overline{RESET}$ надходить логічний нуль, елемент І подає 0 на вхід тригера. Коли на $\overline{RESET}$ надходить логічна одиниця, елемент І пропускає сигнал D на вхід тригера. У цьому прикладі $\overline{RESET}$ – сигнал із активним низьким рівнем (інверсна логіка). Це означає, що скидання відбувається, коли на цей вхід надходить 0, а не 1. Додавши інвертор, ми б отримали схему з активним високим рівнем (пряма логіка). На рис. 3.11, *b* та *c* подано познаки тригера з прямим скиданням.

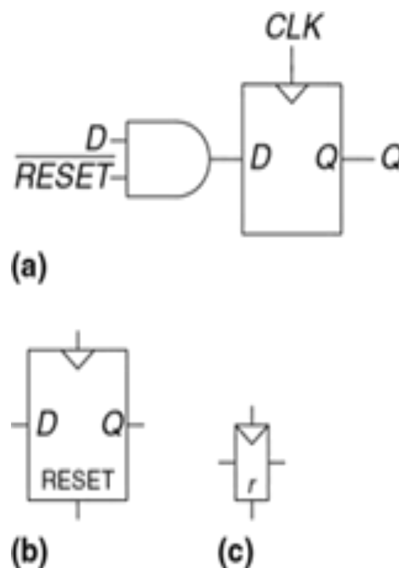


Рисунок 3.11 – Тригер, що синхронно скидається: a – схема; b, c – познаки

Тригери асинхронного скидання вимагають зміни своєї внутрішньої структури й залишені для самостійного розбору (вправа 3.13), проте й вони найчастіше доступні розробникам як стандартний компонент.

Як ви могли легко здогадатися, іноді використовуються тригери з функцією установки. Коли встановлено сигнал *SET*, у такий тригер завантажується логічна 1, і він працює в синхронному й асинхронному режимах. У таких тригерів може бути вхід *ENABLE*, і вони, ймовірно, згруповані в *N*-розрядні регістри.

3.2.7 Проєктування тригерів і засувок на транзисторному рівні

У прикладі 3.1 продемонстровано, що якщо тригери побудовані з логічних елементів, то в них використовується значна кількість транзисторів. Але фундаментальна функція засувки (тригера, синхронізованого рівнем) – бути відкритою або закритою – робить її схожою з ключем. У п. 1.7.7 зазначалося, що застосування прохідного вентиля є ефективним способом створення КМОН-ключа. Отже, можемо скористатися перевагами прохідних ключів для зменшення кількості транзисторів.

Як зображено на рис. 3.12, *a*, компактний D-клапан може бути спроєктований із застосуванням одного прохідного ключа. Якщо CLK = 1, а $\overline{CLK}$ = 0, прохідний ключ замкнуто, отже, D проходить на Q, і засувка відкрита. Якщо CLK = 0, а $\overline{CLK}$ = 1, прохідний ключ розімкнуто, отже, вихід Q ізольований від входу D, і засувка закрита. Однак такий тригер має суттєві недоліки. Розглянемо їх.

● Потенціал на виході рухається. Коли засувка закрита, значення виходу Q не підтягнуто до жодного логічного рівня. У цьому разі вузол Q називають рухомим, або динамічним. За деякий час шуми та витік заряду можуть змінити значення виходу Q.

● Відсутність буферів. Цей призводило до некоректної роботи кількох комерційних мікросхем. Випадковий викид, що спричиняє появу на вході *D* негативної напруги, може увімкнути *n*-канальний транзистор, відкриваючи засувку, навіть якщо *CLK* = 0. Так само викид на вході *D,* вищий за напругу живлення, може відкрити *p*-канальний транзистор, навіть якщо *CLK* = 0. Але прохідний ключ симетричний, отже, він може бути відкритий викидами на виході *Q*, впливаючи цим на значення входу *D*. Основне правило: ні вхід прохідного ключа, ні вузол стану логічної послідовної схеми ніколи не мають застосовуватися там, де існує ймовірність виникнення перешкод чи шумів.

На рис. 3.12, *b* зображено надійнішу 12-транзисторну *D*-засувку, що використовується в сучасних комерційних мікросхемах. Хоча вона й побудована на основі прохідних ключів, що тактуються, в ній додані інвертори I1 і I2, які виконують роль вхідного й вихідного буферів. Стан засувки визначається станом вузла *N*1. Інвертор I3 і буфер з трьома станами T1 утворюють зворотний зв'язок та усувають ефект рухомого потенціалу на *N*1. Якщо вузол *N*1 відхилиться від стаціонарного стану під впливом перешкод або шуму, тоді на *CLK* = 0 буфер T1 поверне його в цей стан.

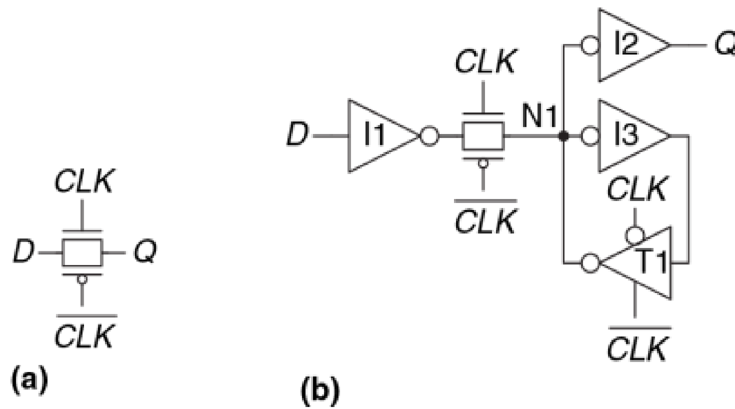


Рисунок 3.12 – Схема *D*-тригера-засувки

На рис. 3.13 зображено *D*-тригер, що має дві засувки, які управляються сигналами *CLK* і $\overline{CLK}$. Ми вилучили деякі зайві інвертори, і тепер для створення тригера потрібно лише 20 транзисторів.

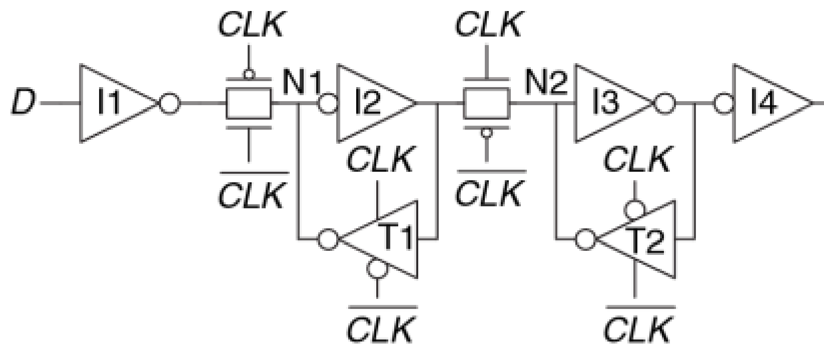


Рисунок 3.13 – Схема *D*-тригера

3.2.8 Загальний огляд

Засувки та тригери є фундаментальними функціональними вузлами послідовних логічних схем. *D*-засувка відкрита, коли *CLK* = 1, даючи цим змогу значенню з входу *D* потрапити на вихід *Q*. *D*-тригер передає значення з *D* на *Q* тільки по фронту тактового сигналу. У решті випадків тригери й засувки зберігають свій попередній стан. Регістром називається набір з кількох *D*-тригерів із загальним тактовим сигналом.

Приклад 3.2.

Порівняння засувок і тригерів

Бен Бітдідл подав сигнали *D* та *CLK*, як продемонстровано на рис. 3.14, на *D*-засувку й на *D*-тригер. Допоможіть визначити значення виходу *Q* для кожного пристрою.

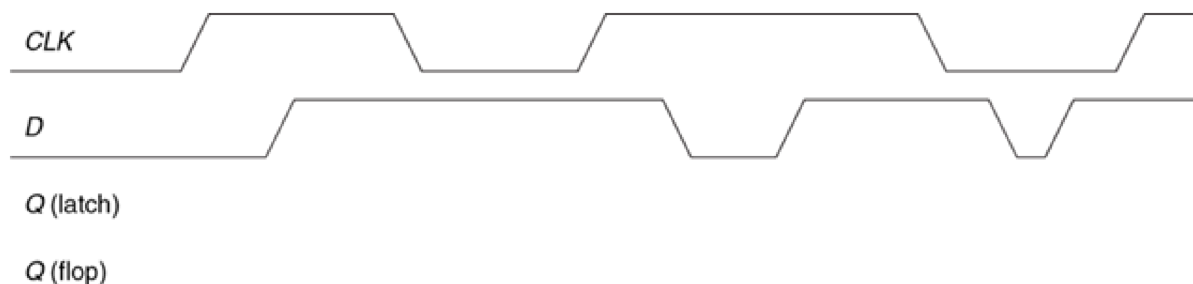


Рисунок 3.14 – Вихідні часові діаграми

Виконання. На рис. 3.15 зображено часові діаграми вихідних сигналів з огляду на незначні затримки в тригері та засувці. Стрілки вказують на фактор, що спричинив перемикання сигналу на виході. Вихідне значення *Q* не відоме, це продемонстровано двома горизонтальними лініями на початку діаграми.

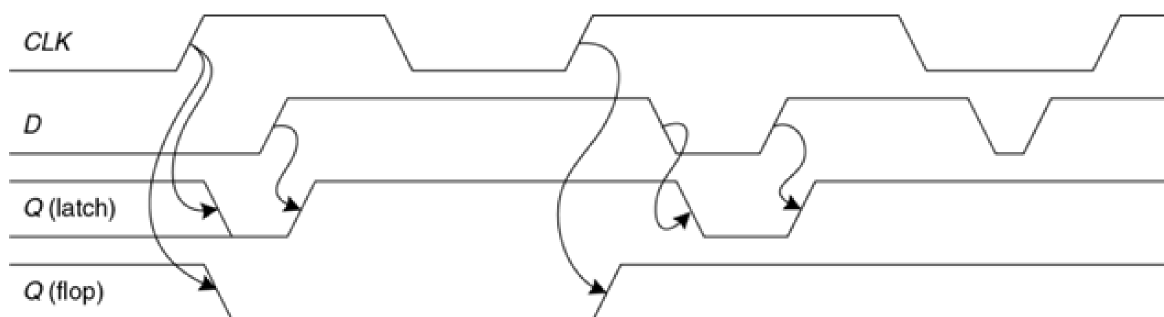


Рисунок 3.15 – Рішення прикладу 3.2

Спочатку розглянемо засувку. Під час проходження першого фронту тактового сигналу *CLK* значення *D* = 0, тому *Q* стане 0. Щоразу, коли *D* змінюватиметься, тоді як *CLK* = 1, *Q* також змінюватиметься. Якщо *D* змінюватиметься, коли *CLK* = 0, змін на виході *Q* не буде.

Тепер розглянемо тригер, що синхронізується фронтом. Значення на виході *Q* стає рівним значенню на вході *D* по кожному фронту тактового сигналу *CLK*. В іншому разі *Q* не змінюється.

3.3 Проєктування синхронних логічних схем

Узагалі, послідовні схеми передбачають всі схеми, які не є комбінаційними, тобто послідовними є схеми, значення виходу яких не можна однозначно визначити, знаючи лише поточні значення входів. Поведінка деяких послідовних схем може бути дуже складною. Цей підрозділ розпочнемо з аналізу кількох таких схем. Потім запропонуємо поняття синхронних послідовних схем і динамічної дисципліни. Обмеживши себе розглядом лише синхронних послідовних схем, зможемо сформулювати прості систематичні підходи до аналізу та проєктування таких схем.

3.3.1 Деякі проблемні схеми

Приклад 3.3.

Нестійкі схеми

Аліса Хакер зіткнулася зі схемою, що має три інвертори, замкнуті в кільце (рис. 3.16). Вихід третього інвертора подається на перший вхід. Затримка поширення кожного з інверторів дорівнює 1 нс. Визначте, що відбувається в такій схемі.

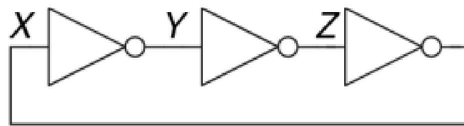


Рисунок 3.16 – Кільце з трьох інверторів

Виконання. Припустимо, що в початковий момент часу сигнал X дорівнює логічному 0. Тоді $Y = 1$, $Z = 0$, отже, $X = 1$, що не збігається з нашим припущенням. Ця схема не має стабільних станів, тому вона називається нестабільною або нестійкою. На рис. 3.17 подано поведінку схеми. Якщо сигнал X переходить з 0 в 1 у початковий момент часу, Y перейде з 1 в 0 у момент часу $T = 1$ нс, а Z з 0 в 1 – у $T = 2$ нс, а потім X перейде назад з 1 в 0 у час $T = 3$ нс. Зі свого боку Y перейде з 0 в 1 у момент $T = 4$ нс, Z перейде з 1 в 0 під час $T = 5$ нс, а X знову перейде з 0 в 1 у момент часу $T = 6$ нс, і далі така поведінка схеми повторюється. Кожен вузол коливатиметься між 0 та 1 з періодом $T = 6$ нс. Така схема називається кільцевим генератором.

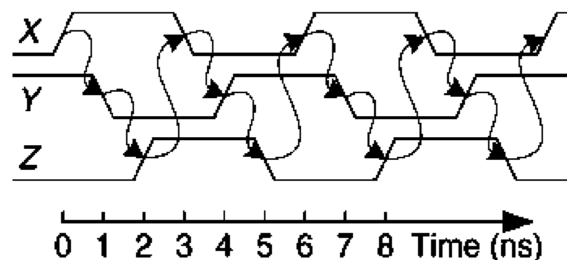


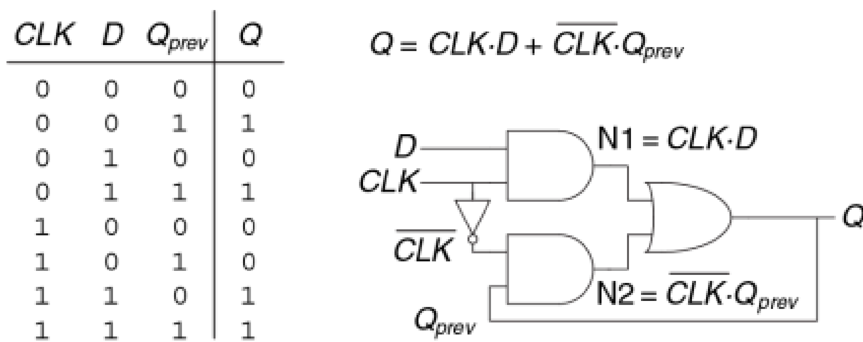
Рисунок 3.17 – Діаграми в часі кільцевого генератора

Період коливань кільцевого генератора залежить від затримки розповсюдження кожного інвертора. Ця затримка залежить від того, як виготовлений інвертор, від напруги живлення й навіть температури. Тому точно визначити період коливань кільцевого генератора складно. Іншими словами, кільцевий генератор – послідовна схема без входів і з одним виходом, значення якого періодично змінюються.

Приклад 3.4.

Гонки в послідовних схемах

Бен Бітдідл спроєктував нову *D*-засувку, що, як він вважає, працює краще, ніж зображена на рис. 3.7, оскільки в ній використовується менше елементів. Чоловік склав таблицю істинності для виходу *Q* за даними двох входів *D* і *CLK* та попереднього стану *Qprev*. Ґрунтуючись на цій таблиці, Бен написав логічні рівняння. Для отримання *Qprev* використовується зворотний зв'язок із виходу *Q*. Його схема зображена на рис. 3.18. Чи працює його засувка коректно, незалежно від затримок кожного елемента?



CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Рисунок 3.18 – «Удосконалена» *D*-засувка

Виконання. На рис. 3.19 показано, що схема може працювати некоректно через появу гонок (англ. *race condition*), що призводить до збою, якщо певні елементи повільніші за інші. Нехай *CLK* = *D* = 1. Засувка відкрита, пропускає дані, і на виході *Q* з'являється логічна 1. Тепер сигнал *CLK* переходить з 1 в 0. Тригеру потрібно запам'ятати своє попереднє значення, тобто зберегти *Q* = 1.

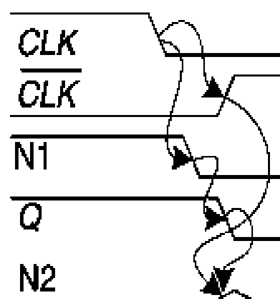


Рисунок 3.19 – Часові діаграми засувки, що ілюструють гонки

Припустимо, що затримка поширення інвертора значно більша, ніж затримки елементів І та АБО. У такому разі сигнали *N*1 та *Q* перейдуть з 1 до 0 раніше, ніж сигнал $\overline{CLK}$ стане 1. У цьому разі сигнал *N*2 ніколи не набуде значення логічної одиниці, і вихідний сигнал схеми *Q* залишиться нульовим. Це приклад проєкту асинхронної схеми, у якій виходи безпосередньо пов'язані зворотним зв'язком із входами. Асинхронні схеми не популярні

через непередбачуваність поведінки, пов’язаною зі швидкодією елементів, коли поведінка схеми залежить від того, який сигнал усередині схеми пройде швидше за інших. Одна схема може працювати, тоді як інша, що здається ідентичною, зібрана з елементів із незначними затримками, може не працювати. Або схема може працювати тільки за певних температур або напруги, за яких затримки відповідають розрахунковим. Подібні помилки проєктування дуже важко виявляти.

3.3.2 Синхронні послідовні схеми

У попередніх прикладах були присутні циклічні шляхи, у яких виходи безпосередньо з’єднані зворотним зв’язком із входами. Це скоріше послідовні, ніж комбінаційні схеми. У комбінаційній логіці немає циклічних шляхів, відсутні залежності стану виходу від часу проходження сигналу. Якщо на входи комбінаційної логічної схеми подано певні сигнали, то її вихід за деякий час завжди встановиться в певний коректний стан. Однак у послідовних схемах із циклічними шляхами може з’явитися небажана нестабільність або гонки. Перевірка таких схем потребує чимало часу, і багато видатних проєктувальників припускалися подібних помилок.

Щоб уникнути таких проблем, розробники розривають циклічні шляхи й додають у розрив регістри. Це перетворює схему на набір комбінаційної логіки та регістрів. У регістрах міститься стан системи, що змінюється лише з фронту тактового імпульсу. У цьому разі говорять, що стан синхронізовано з тактовим сигналом. Якщо період тактового сигналу досить великий, щоб усі входи регістрів встигли встановитися до фронту наступного тактового імпульсу, ефекти, пов’язані з гонками, усуваються. Дотримання правила «завжди використовувати регістри у зворотному зв’язку» призводить до формального визначення синхронної послідовної схеми.

Нагадаємо, що схема визначається набором входів і виходів і функціональними та часовими параметрами. У послідовної схеми існує кінцевий набір дискретних станів {S0, S1, …, Sk-1}. У синхронної послідовності є вхід тактового сигналу, передні фронти тактових імпульсів визначають послідовність точок на осі часу, у яких відбуваються зміни стану. Ми часто послуговуватимемося термінами «поточний стан» і «наступний стан» для того, щоб розрізняти стан системи в теперішньому стані системи, від того, в який вона перейде по фронту наступного тактового імпульсу. Функціональний опис визначає наступний стан та значення кожного виходу для кожної можливої комбінації поточних станів і вхідних сигналів.

Часова специфікація передбачає верхню межу t_{PCQ} і нижню межу t_{CCQ} тривалості часового проміжку від переднього фронту тактового імпульсу до моменту зміни вихідного сигналу, а також із часів установлення та утримання t_{SETUP} і t_{HOLD}, які визначають проміжок часу до і після надходження фронту такту. Значення на входах не мають змінюватися.

Правила побудови синхронних послідовних схем свідчать, що схема є синхронною послідовністю, якщо її елементи задовольняють такі умови:

- кожен елемент схеми є або регістром, або комбінаційною схемою;
- як мінімум один елемент схеми є регістром;
- усі регістри тактуються єдиним тактовим сигналом;
- у кожному циклічному шляху є щонайменше один регістр.

Послідовні схеми, що не є синхронними, називають асинхронними.

Тригер є найпростішою синхронною послідовною схемою з двома станами {0,1}. Він має один вхід даних *D*, один вхід тактового сигналу *CLK*, один вихід *Q*.

Функціональний опис *D*-тригера полягає в тому, що його наступним станом є значення входу *D*, а значення виходу *Q* є поточним станом (див. рис. 3.20). Ми часто позначатимемо поточний стан змінної *S*, а наступний стан – змінної *S*’, тобто *S*’ (штрих) позначає наступний стан, а не інверсію. Часові діаграми послідовних схем будуть розглянуті в підрозділі 3.5. Два інші види синхронних послідовних схем – кінцеві автомати та конвеєри. Вони розглядатимуться нижче в цьому розділі.

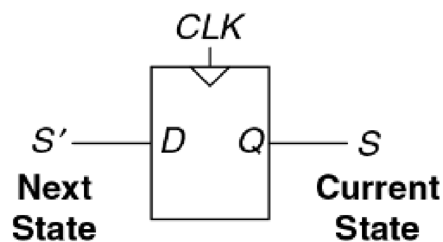


Рисунок 3.20 – Поточний і наступний стан тригера

Приклад 3.5.

Синхроні послідовні схеми

Які з наведених на рис. 3.21 схеми є послідовними синхронними?

Виконання. Схема (a) є комбінаційною, а не послідовною, оскільки в ній відсутні регістри. Схема (b) – проста послідовна, тому що в ній немає зворотного зв’язку. Схема (c) не є ні комбінаційною, ні послідовною синхронної, оскільки вона містить засувку, яка не є ні регістром, ні комбінаційною схемою. Схеми (d) та (e) – синхронні послідовні логічні; вони є

двома класами кінцевих автоматів, які обговорюватимуться в підрозділі 3.4. Схема (f) – ні комбінаційна, ні синхронна послідовна, оскільки у неї є циклічний шлях із виходу комбінаційної схеми на її вхід, у цьому разі в тракті зворотного зв’язку відсутній регістр. Схема (g) є синхронною послідовною у вигляді конвеєра, який вивчатимемо в підрозділі 3.6. Схема (h) не є синхронною послідовною, оскільки тактовий сигнал другого регістра різниться від першого з причини затримки, що виникає через два інвертори.

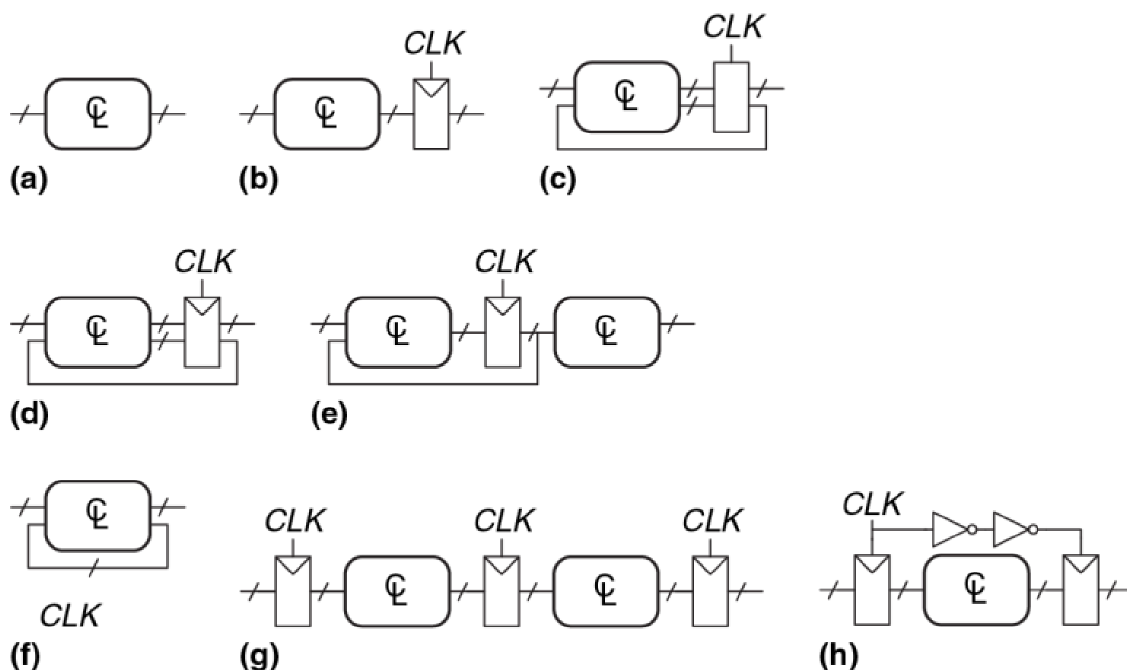


Рисунок 3.21 – Приклади схем

3.3.3 Синхронні та асинхронні схеми

Теоретично через відсутність часових обмежень, що накладаються на систему регістрами, які тактуються, у проєктуванні асинхронних схем розробник має більшу свободу, ніж у процесі проєктування синхронних. Так само як аналогові схеми менш формалізовані порівняно з цифровими, тому що в аналогових схемах можуть використовуватися довільні напруги, асинхронні схеми менш формалізовані, ніж синхронні, оскільки зворотний шлях у них може бути будь-який. Проте, виявляється, що синхронні схеми проєктувати та застосовувати простіше, ніж асинхронні, як і цифрові схеми простіше проєктувати, ніж аналогові. Незважаючи на багаторічні наукові дослідження асинхронних схем, майже всі сучасні цифрові схеми є синхронними.

Асинхронні схеми іноді використовуються для взаємозв’язку систем із різними тактовими сигналами або для зчитування значень із входів у довільний час, як і аналогові схеми необхідні для взаємодії з реальним світом аналогових (безперервних) напруг. Крім того, серед розробок у сфері

асинхронних схем є справді визначні, деякі з них можуть також покращити властивості синхронних схем.

3.4 Кінцеві автомати

Послідовні логічні схеми можуть бути зображені у формі, запропонованій на рис. 3.22. Такі припущення називаються кінцевими автоматами (КА). Вони отримали таку назву внаслідок того, що схема з *k*-регістрів може бути в одному з $2^к$, тобто в кінцевому числі, станів. У КА *M* входів, *N* виходів і *k* бітів станів. На вхід КА подається тактовий сигнал *i*, можливо, сигнал скидання. КА містить два блоки комбінаційної логіки – логіки переходу в наступний стан та вихідної логіки – і регістр, у якому зберігається поточний стан. По фронту кожного тактового імпульсу автомат перетворюється на такий стан, що визначається поточним станом і значеннями на входах.

Існує два основних класи кінцевих автоматів, що розрізняються за функціональними описами. В автоматі Мура вихідні значення залежать лише від поточного стану, тоді як в автоматі Мілі вихід залежить і від поточного стану, і вхідних даних. Кінцеві автомати надають систематичний спосіб проєктування синхронних послідовних схем за заданим функціональним описом. Цей метод опишемо нижче, а зараз розглянемо найпростіший приклад.

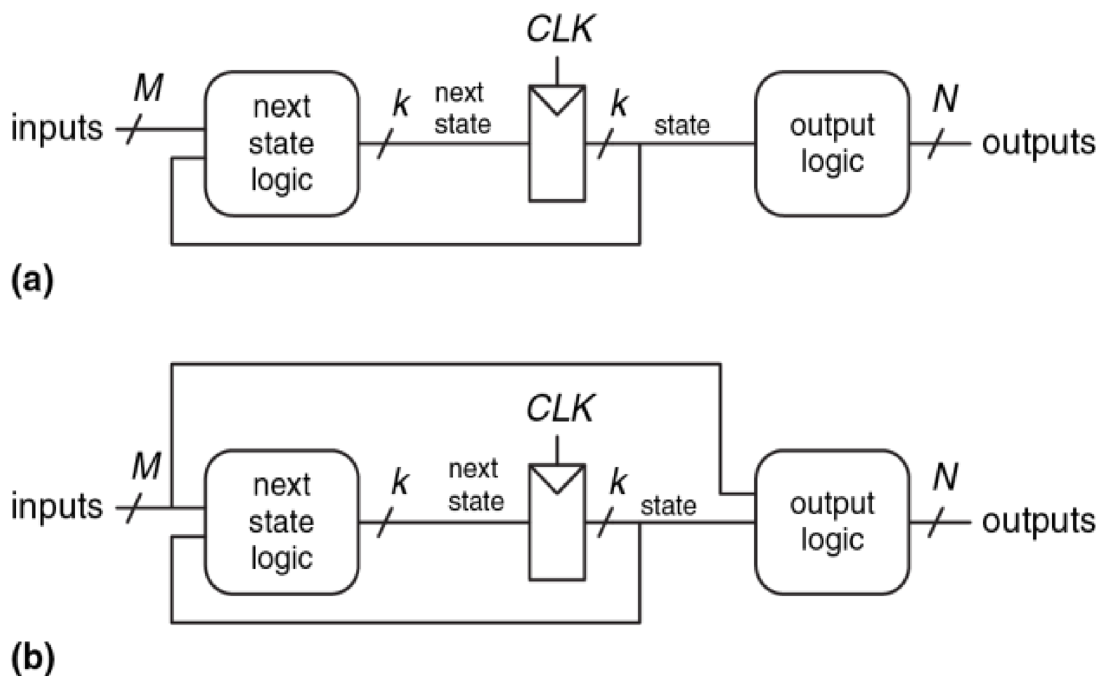


Рисунок 3.22 – Кінцеві автомати: a – автомат Мура; b – автомат Мілі

3.4.1 Приклад проєктування кінцевого автомата

Для того, щоб проілюструвати процес проєктування кінцевого автомата, розглянемо проблему створення контролера світлофора для завантаженого перехрестя в студентському містечку. Студенти-інженери гуляють Академічною вулицею, на якій розташовані навчальні корпуси та гуртожиток. Молоді люди не мають часу читати про кінцеві автомати й ідуть, не дивлячись під ноги. Футболісти носяться між спортзалом та їдальнею Біговою вулицею. Вони ганяють м'яч туди-сюди і також не дивляться під ноги. Декілька студентів вже отримали серйозні травми на перехресті, і декан попросив Бена Бітдідла встановити світлофор, доки не стався інцидент з летальним кінцем.

Бен вирішив упоратися із проблемою за допомогою кінцевого автомата. Чоловік установив два датчики руху, T_A та T_B, на Академічній та Біговій вулицях відповідно. Кожен датчик видає одиницю, якщо студенти присутні на вулиці, та нуль, якщо нікого немає. Він також установив два світлофори для управління рухом – L_A та L_B. Кожен світлофор отримує вхідний цифровий сигнал, що визначає, яким світлом він має світити: червоним, жовтим або зеленим. Отже, КА має два входи, T_A і T_B, і два виходи, L_A і L_B. Перехрестя з двома світлофорами й датчиками зображено на рис. 3.23. Бен подає тактові імпульси раз на 5 с. На передньому фронті кожного імпульсу колір світлофора може змінитися залежно від показників датчиків руху. Також є кнопка скидання, щоб техніки могли скидати контролер після подачі живлення у відомий вихідний стан. На рис. 3.24 автомат зображено у вигляді «чорної скриньки».

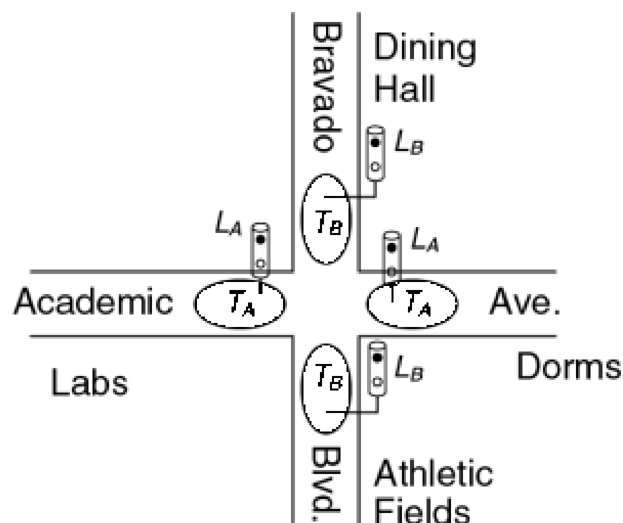


Рисунок 3.23 – Карта кампусу

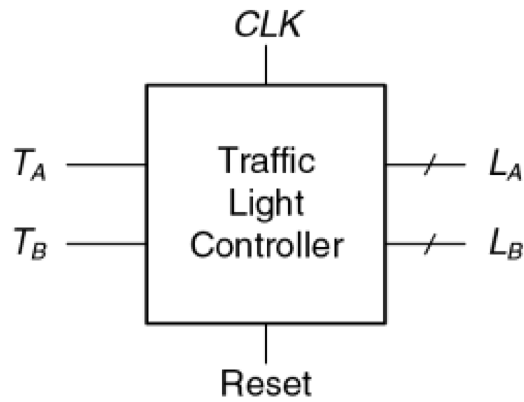


Рисунок 3.24 – Кінцевий автомат як «чорна скринька»

Наступний крок першокурсника – зробити рисунок діаграми переходів (чи графа) (див. рис. 3.25, на якому наведені всі можливі стани системи та переходи між ними).

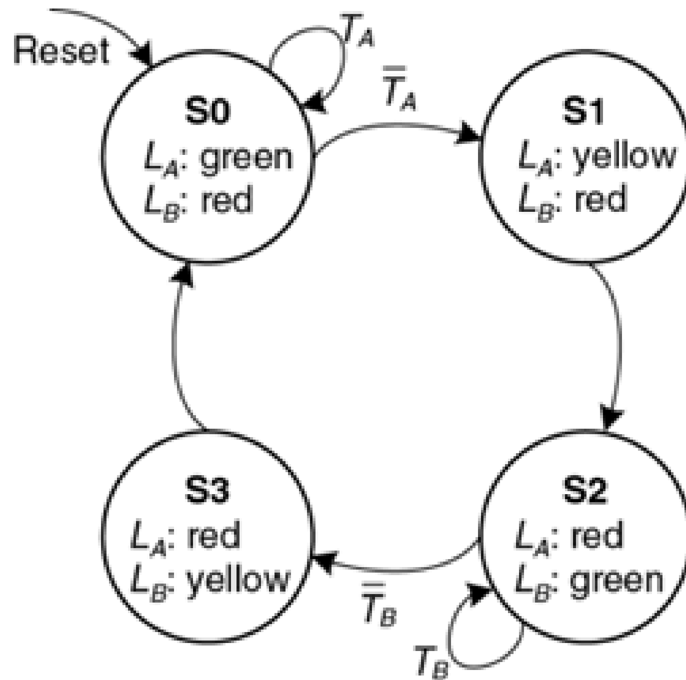


Рисунок 3.25 – Таблиця переходів

Після скидання світлофор горить зеленим на Академічній вулиці та червоним – на Біговій. Кожні 5 с контролер аналізує рух і вирішує, що робити далі. Якщо рух присутній на Академічній вулиці, колір не змінюється. Щойно Академічна вулиця звільняється, на її світлофорі впродовж 5 с горить жовтий, потім спалахує червоний, а на Біговий – зелений. Аналогічно зелене світло на Біговій вулиці зберігається доти, доки вулиця стане вільною, потім світлофор перемикається на жовтий, а потім – на червоний.

Круги на діаграмі переходів позначають стани, а дуги зі стрілками між ними – переходи між цими станами. Переходи здійснюються на передньому фронті тактового імпульсу. Ми не зображатимемо тактовий сигнал на діаграмі, оскільки він завжди присутній у синхронних логічних схемах. Зазначимо, що тактовий сигнал лише визначає, коли станеться перехід, тоді як діаграма визначає, який саме перехід відбудеться. Стрілка, позначена як скидання, вказує на перехід ззовні в стан *S*0, указуючи на те, що система перейде в цей стан відразу після скидання незалежно від того, в якому вона стані була до цього. Якщо є кілька стрілок, що виходять із деякого стану, то їх підписують, щоб позначити, який вхідний сигнал викликав цей перехід. Наприклад, система перебуває в стані *S*0. Система залишиться у стані *S*0, якщо $T_A = 1$, і перейде в стан *S*1, якщо $T_A = 0$. Якщо з цього стану виходить лише одна стрілка, це значить, що такий перехід відбудеться незалежно від стану входів. Наприклад, зі стану *S*1 система завжди переходитиме в стан *S*2, коли L_A – червоний, а L_B – зелений.

На основі цієї діаграми переходів Бен Бітдідл записав таблицю переходів (див. табл. 3.1), що відтворює, яким має бути наступний стан *S*', що відповідає поточному стану та вхідним сигналам. Зауважимо, що в таблиці використовуються символи *X*, які означають, що такий стан залежить від конкретного входу. Також зазначимо, що сигнал скидання вилучено з цієї таблиці. Натомість ми використовували тригери, що скидаються. Вони переходять у стан *S*0 відразу після скидання незалежно від даних на вході.

Таблиця 3.1 – Таблиця переходів

Current State *S*	**Inputs T_A T_B**	**Next State *S'***
S0	0 X	S1
S0	1 X	S0
S1	X X	S2
S2	X 0	S3
S2	X 1	S2
S3	X X	S0

Діаграма переходів абстрактна в тому сенсі, що вона застосовує стани, позначені як {*S*0, *S*1, *S*2, *S*3}, і виходи, позначені як {червоний, жовтий, зелений}.

Для побудови реальної схеми станам і виходам мають бути поставлені у відповідність двійкові коди.

Бен обрав просте кодування (див. табл. 3.2 і 3.3). Кожен стан і кожне вихідне значення закодовано двома бітами: $S_{1:0}$,$L_{A1:0}$ та $L_{B1:0}$.

Таблиця 3.2 – Кодування станів

State Encoding $S_{1:0}$	
S0	00
S1	01
S2	10
S3	11

Таблиця 3.3 – Кодування виходів

Output	**Encoding $L_{1:0}$**
green	00
yellow	01
red	10

Бен переписує таблицю переходів, використовуючи двійкове кодування, як подано в табл. 3.4. Вона є таблицею істинності, що визначає логіку наступного стану. Таблиця переходів визначає наступний стан *S*’ як функцію входів і поточного стану.

Таблиця 3.4 – Таблиця переходів з двійковим кодуванням

Current State		**Inputs**		**Next State**	
S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Аналіз поданої таблиці дає змогу легко записати булеве рівняння для наступного стану в досконалій диз’юнктивній нормальній формі (СДНФ):

$$S_1'=\overline{S_1}S_0+S_1\overline{S_0}*\overline{T_B}+S_1\overline{S_0}T_B\ ; \quad (3.1)$$

$$S_0'=\overline{S_1}*\overline{S_0}*\overline{T_A}+S_1*\overline{S_0}*\overline{T_B}.$$

Рівняння можуть бути спрощені за допомогою карт Карно, але це часто простіше зробити в голові, уважно вивчивши рівняння. Наприклад, T_B і $\overline{T_B}$ очевидно скорочуються. Вирази (3.2) є результатом спрощення виразів (3.1).

$$S_1'=S_0\text{^}S_1;$$

$$S_0'=\overline{S_1}*\overline{S_0}*\overline{T_A}+S_1*\overline{S_0}*\overline{T_B}\ . \quad (3.2)$$

Подібним чином Бен записує таблицю виходів (табл. 3.5), визначаючи, яким має бути вихід для кожного стану. Потім він знову складає і спрощує булеві вирази для виходів. Наприклад, L_{A1} = 1 у рядках, де S_1= 1.

$$L_{A1} = S_1;$$
$$L_{A0} = S_1 * \overline{S_0}; \quad (3.3)$$
$$L_{B1} = \overline{S_1};$$
$$L_{B0} = S_1 S_0.$$

Таблиця 3.5 – Таблиця виходів

Current State		**Outputs**			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Нарешті, Бен креслить автомат Мура у вигляді, поданому на рис. 3.22, *a*. Спочатку він накреслює дворозрядний регістр станів (див. рис. 3.26, *a*). За кожним переднім фронтом тактового сигналу регістр станів фіксує наступний стан $S'_{1:0}$, і, отже, він стає поточним станом $S_{1:0}$. Регістр станів отримує сигнал синхронного або асинхронного скидання для ініціалізації КА після подачі живлення. Потім Бен накреслює схему визначення наступного стану, ґрунтуючись на рівняннях (3.2), що обчислюють наступний стан за значенням на входах і за поточним станом. Цю схему зображено на рис. 3.26, *b*. Нарешті, за рівняннями (3.3) Бен накреслює схему (див. рис. 3.26, *c*), яка обчислює значення на виходах за поточним станом.

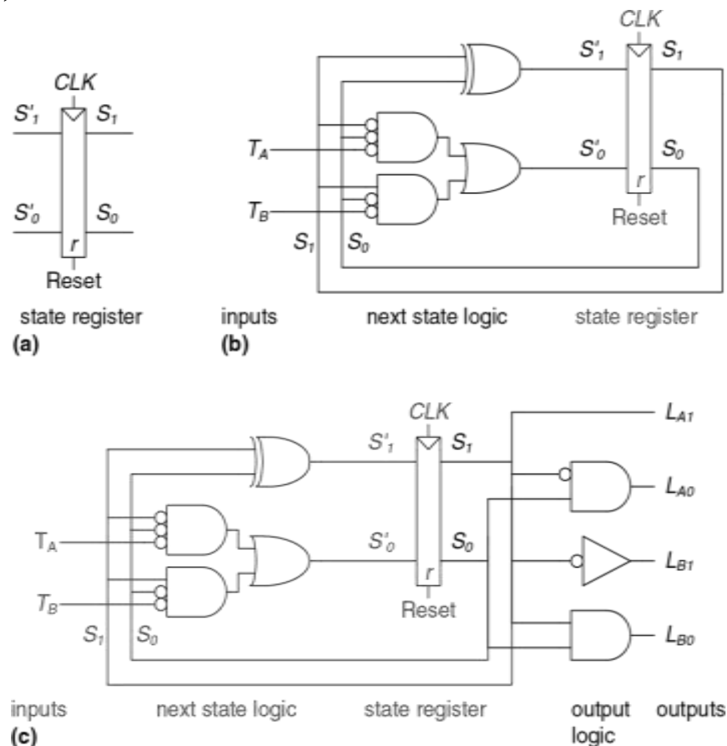


Рисунок 3.26 – Схема контролера кінцевого автомата світлофора

На рис. 3.27 показано часову діаграму переходу контролера світлофора з одного стану до іншого. На діаграмі показані сигнал *CLK, Reset,* входи T_A та T_B, наступний стан *S*', поточний стан *S* та виходи L_A і L_B. Стрілки показують причинний зв'язок. Наприклад, зміна стану викликає зміну виходів. Пунктирні лінії відповідають передньому фронту сигналу *CLK*, тобто часу, коли стан кінцевого автомата змінюється.

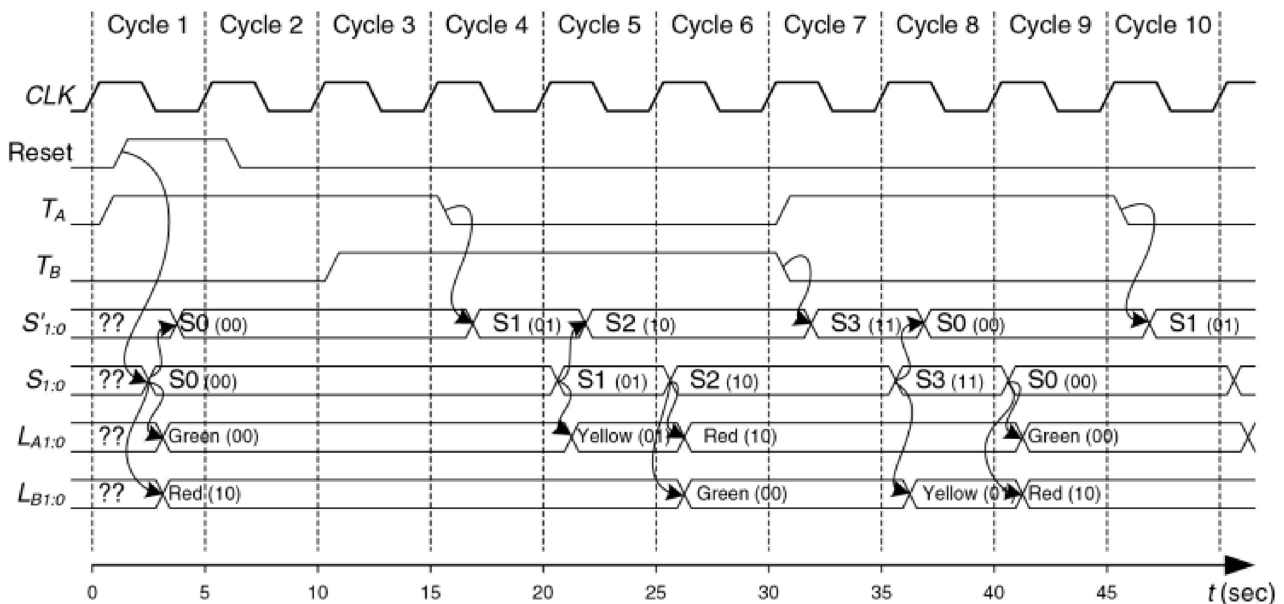


Рисунок 3.27 – Часова діаграма контролера світлофора

Період тактового сигналу дорівнює 5 с, тому сигнали світлофора можуть перемикатися максимум один раз за 5 с. Коли кінцевий автомат лише увімкнено, його стан не відомий, це показують знаки питання. Отже, система має бути скинута для переведення її у відомий стан. На цій часовій діаграмі *S* негайно скидається в *S*0, наголошуючи на тому, що використовуються тригери з асинхронним скиданням. У стані *S*0 світло L_A зелене, а світло L_B червоне.

У цьому прикладі рух на Академічній вулиці починається відразу. Отже, контролер залишається в стані *S*0, залишаючи на світлофорі L_A зелене світло, навіть якщо на Біговій вулиці хтось з'являється. За 15 с потік на Академічній вулиці розсотується, і T_A скидається. Контролер переходить у стан *S*1 фронтом відповідного тактового імпульсу й запалює жовте світло на L_A. Ще за 5 с контролер переходить у стан *S*2, у якому на L_A спалахує червоне, а на L_B – зелене світло. Контролер залишається в стані *S*2 доти, доки Бегова вулиця не спорожніє. Потім він перетворюється на стан *S*3, на L_B з'являється жовте світло. За 5 с контролер переходить у стан *S*0, перемикаючи L_B на червоне, а L_A – на зелене світло й т.д.

3.4.2 Кодування станів

У попередньому прикладі кодування станів і виходів було обрано довільно. Вибір іншого кодування спричинив би іншу схему. Основна проблема полягає в тому, як визначити кодування, що вимагатиме найменшу кількість елементів і призведе до найменших затримок у схемі. На жаль, простого способу знайти найкраще кодування не існує, окрім як перепробувати всі можливі варіанти, що не раціонально, якщо кількість станів значна. Однак часто можна знайти хороше кодування так, щоб пов’язані стани або виходи мали спільні біти. У пошуку набору можливих кодувань та вибору найбільш раціонального часто використовуються системи автоматизованого проєктування (САПР).

Одне з важливих рішень у кодуванні станів – вибір між двійковим кодуванням (00, 01, 10) та прямим кодуванням (001, 010, 100), яке також називається кодуванням «1 з *N*». За умови двійкового кодування, як у прикладі з контролером світлофора, кожному стану відповідає двійкове число (номер цього стану). Оскільки *K* двійкових чисел можна записати в *log2K* розрядах, системі з *K* станами потрібно всього *log2K* бітів стану.

У прямому кодуванні кожного стану використовується один біт стану. Англійською воно називається *one-hot*, оскільки тільки один розряд буде гарячим, тобто тільки в одному з розрядів міститься логічна одиниця в будь-який момент часу. Наприклад, у КА з прямим кодуванням і трьома станами коди станів будуть 001, 010 та 100. Кожен біт стану зберігається в тригері; отже, пряме кодування вимагає більше тригерів, ніж двійкове. Однак за умови застосування прямого кодування схема визначення наступного стану та схема формування вихідних сигналів часто спрощується; тож потрібно менше елементів. Найкращий вибір кодування залежить від особливостей конкретного автомата.

Приклад 3.6.

Кодування стану кінцевого автомата

У лічильника з діленням на *N* є один вихід, а входи відсутні. Вихід *Y* розташований на високому рівні протягом одного періоду кожні *N* періодів тактового сигналу. Іншими словами, вихід поділяє тактову частоту на *N*.

На рис. 3.28 наведено часову діаграму та діаграму переходів для лічильника-дільника на 3. Накресліть схему такого лічильника з використанням двійкового й прямого кодувань.

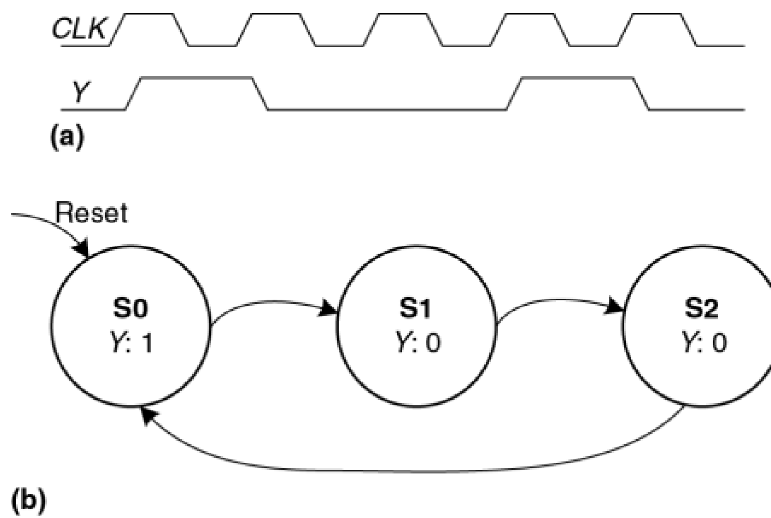


Рисунок 3.28 – Лічильник-дільник на 3:
a – часова діаграма; b – діаграма переходів

Виконання. У табл. 3.6 і 3.7 подано абстрактні таблиці переходів між станами та виходу до кодування.

Таблиця 3.6 – Таблиця переходів лічильника-дільника на 3

Current State	**Next State**
S0	S1
S1	S2
S2	S0

Таблиця 3.7 – Таблиця виходів лічильника-дільника на 3

Current State	**Output**
S0	1
S1	0
S2	0

У табл. 3.8 порівнюються двійкове та пряме кодування для трьох станів.

Таблиця 3.8 – Двійкове та пряме кодування лічильника-дільника на 3

State	**One-Hot Encoding**			**Binary Encoding**	
	S_2	S_1	S_0	S'_1	S'_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

У двійковому кодуванні використовуються два розряди. Табл. 3.9 є таблицею переходів цього кодування. Зверніть увагу, що входи відсутні; наступний стан залежить від поточного стану. Таблицю виходу залишимо читачеві як домашнє завдання. З наведених таблиць легко отримати вирази для виходу та для наступного стану:

$$S'_1=\overline{S_1}S_0;$$
$$S'_0=\overline{S_1}*\overline{S_0}; \quad (3.4)$$
$$Y=\overline{S_1}*\overline{S_0}. \quad (3.5)$$

Таблиця 3.9 – Таблиця переходів з двійковим кодуванням

Current State		**Next State**	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

У разі прямого кодування використовується 3 біти стану. Табл. 3.10 – таблиця переходів цього кодування, а таблицю виходу ми також залишимо читачеві для самостійного виконання. Вирази для виходу та наступного стану будуть такими:

$$S'_2=S_1;$$
$$S'_1=S_0; \quad (3.6)$$
$$S'_0=S_2;$$
$$Y=S_0. \quad (3.7)$$

Таблиця 3.10 – Таблиця переходів з прямим кодуванням

Current State			**Next State**		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

Зауважимо, що залізо для двійкового кодування може бути оптимізовано способом використання одного елемента для Y і S_0'. В процесі застосування прямого кодування для ініціалізації автомата в стан *S*0 в момент скидання необхідно використовувати тригери з входами скидання та установлення (*resettable and settable*). Вибір найкращої реалізації залежить від відносної

вартості елементів (вентилів) і тригерів, але пряме кодування зазвичай більш ефективне в цьому конкретному прикладі.

Ще одним різновидом прямого кодування є кодування *one-cold*, коли біт, що відповідає стану системи на цей момент, скинутий, тоді як інші біти встановлені: 110, 101, 011.

3.4.3 Автомати Мура та Мілі

Досі ми розглядали приклади автоматів Мура, вихід яких залежить лише від стану системи. Тому на діаграмах переходів для автоматів Мура значення виходів пишуться всередині кружків. Згадаємо, що автомати Мілі дуже схожі на автомати Мура, але значення на їх виходах можуть залежати від значень на входах таким самим чином, як вони залежать від поточного стану системи. Тому на діаграмах переходів для автоматів Міля значення виходів пишуться над стрілками. У блоці комбінаційної логіки, що обчислює вихідні значення, використовуються значення поточного стану та входів (див. рис. 3.29, *b*).

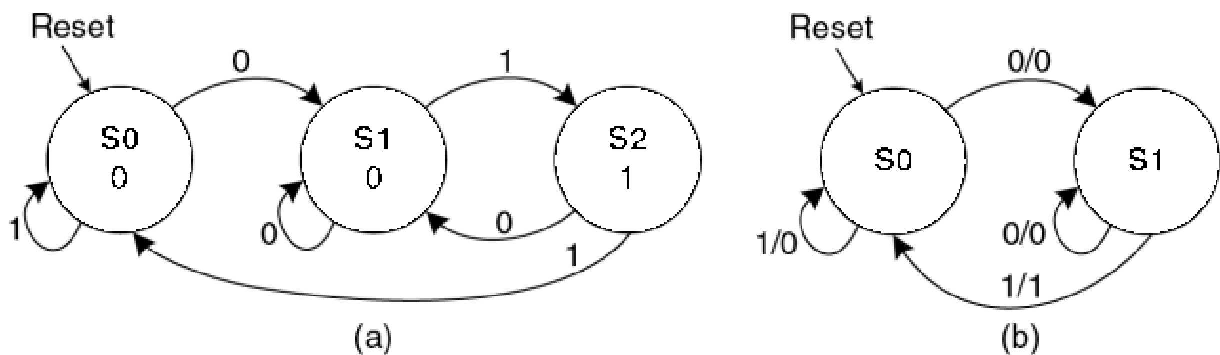


Рисунок 3.29 – Діаграми переходів КА: a – автомат Мура; b – автомат Мілі

Приклад 3.7.

Порівняння автоматів Мура та Мілі

Аліса має равлик-робот з «мозком» у вигляді кінцевого автомата. Равлик рухається зліва направо перфострічкою (перфоровані паперові стрічки активно використовувалися в обчислювальній техніці в 80-х рр.), що містить послідовність нулів і одиниць. Відповідно до кожного тактового імпульсу равлик переповзає на наступний біт.

Равлик усміхається, якщо послідовність з двох останніх бітів, крізь які він переповз, дорівнює 01. Спроєктуйте автомат, що визначає, коли равлику потрібно усміхнутися. На вхід *A* надходить значення біта під пристроєм, що зчитує равлики. На виході *Y* встановлюється логічна одиниця, коли равлик усміхається. Порівняйте реалізації на автоматах Мура та Мілі.

Накресліть часові діаграми для кожного автомата; зобразіть на них вхід, стан та вихід; равлик проповзає послідовність 0100110111.

Виконання. Для автомата Мура потрібно три стани (див. рис. 3.30, *a*). Переконайтеся, що діаграма переходів зображена правильно, зокрема поясніть, чому є стрілка з *S*2 в *S*1, коли на вході 0.

На відміну від автомата Мура, автомату Мілі потрібно лише два стани, що проілюстровано на рис. 3.30, *b*. Кожну стрілку підписано за принципом *A/Y*. *A* – значення входу, що викликало перехід, а *Y* – це відповідний вихідний сигнал.

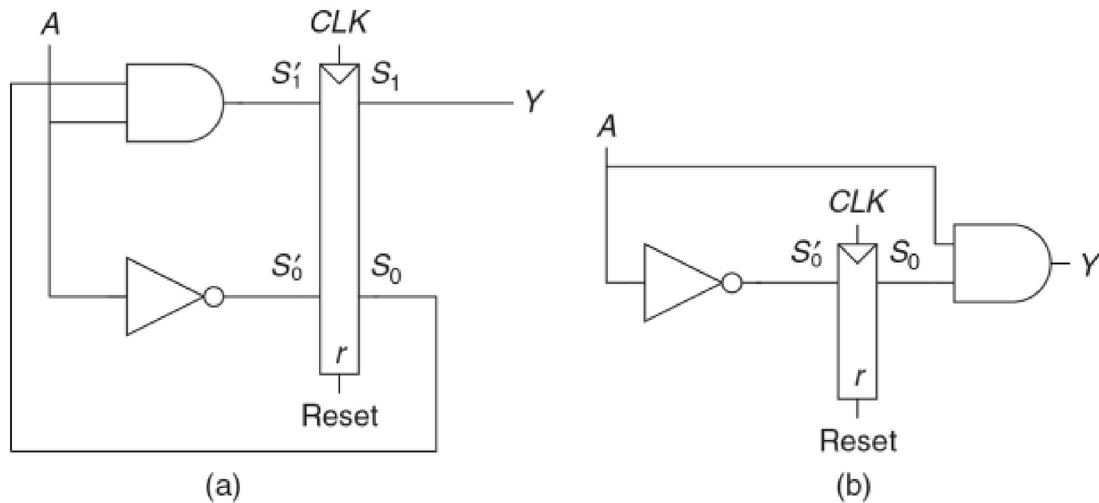


Рисунок 3.30 – Схеми КА: a – Мура; b – Мілі

У табл. 3.11 та 3.12 зображена діаграма переходів і таблиця станів виходу для автомата Мура. Автомату Мура знадобиться щонайменше два біти стану. Застосуємо двійкове кодування: *S*0=00, *S*1=01, *S*2=10. Табл. 3.13 та 3.14 є результатом переписування табл. 3.11 і 3.12 з таким кодуванням.

Таблиця 3.11 – Таблиця переходів автомата Мура

Current State ***S***	**Input** ***A***	**Next State** ***S'***
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Таблиця 3.12 – Таблиця виходів автомата Мура

Current State *S*	**Output** *Y*
S0	0
S1	0
S2	1

Таблиця 3.13 – Таблиця переходів автомата Мура з кодуванням станів

Current State		**Input**	**Next State**	
S_1	S_0	*A*	S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Таблиця 3.14 – Таблиця виходів автомата Мура з кодуванням станів

Current State		**Output**
S_1	S_0	*Y*
0	0	0
0	1	0
1	0	1

Отже, значення наступного стану й значення виходу для цього стану ні на що не впливають (*X*) (не показано в таблицях). Ми користуємося тим, що цей стан нам байдужий для спрощення виразів.

Далі складемо по цих таблицях вирази для наступного стану й для виходу. Зауважимо, що ці вирази спрощені з огляду на те, що стану 11 не існує.

$$S'_1 = S_0 \text{ A}; \tag{3.8}$$

$$S'_0 = \overline{A};$$

$$\text{Y} = S_1. \tag{3.9}$$

Табл. 3.15 – зведена таблиця переходів та виходу для автомата Мілі. Автомату Мілі необхідний лише один біт стану. Застосуємо двійкове кодування: *S*0=0 та *S*1=1. Перепишемо табл. 3.15 у табл. 3.16, використовуючи таке кодування.

Таблиця 3.15 – Таблиця переходів та виходів автомата Мілі

Current State *S*	**Input** *A*	**Next State** *S'*	**Output** *Y*
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Таблиця 3.16 – Таблиця переходів і виходів автомата Мілі з кодуванням станів

Current State S_0	**Input** *A*	**Next State** S'_0	**Output** *Y*
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

За цими таблицями складемо вирази для наступного стану й для виходу:

$$S'_0=\overline{A}; \qquad (3.10)$$

$$Y=S_0A. \qquad (3.11)$$

Схеми автоматів Мілі та Мура зображено на рис. 3.31. Часові діаграми кожного з автоматів подані на рис. 3.32.

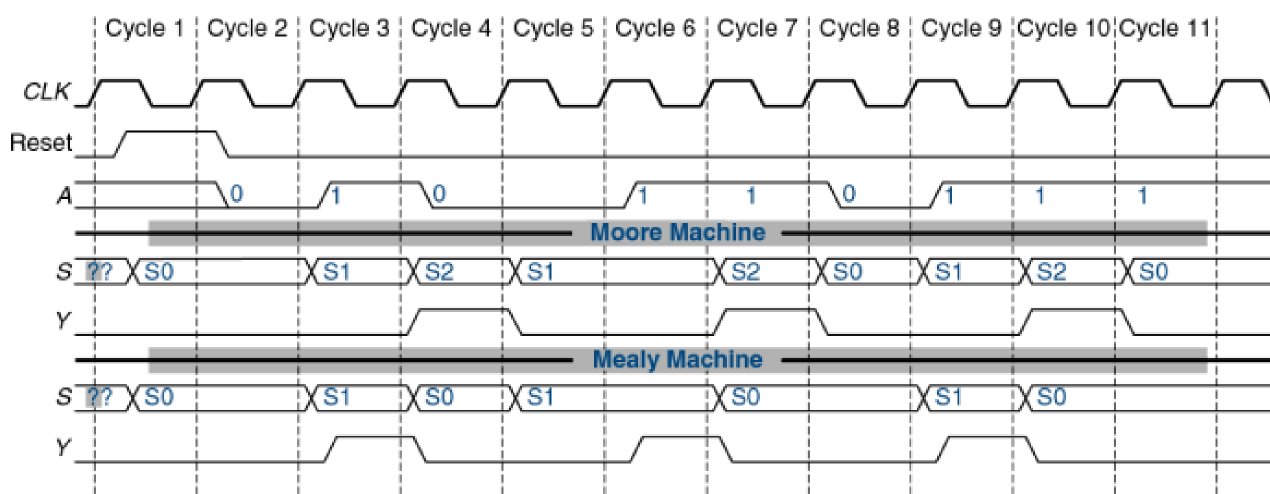


Рисунок 3.31– Часові діаграми автомата Мура й автомата Мілі

Кожен з автоматів проходить крізь різну послідовність станів. Крім того, вихід автомата Мілі випереджає вихід автомата Мура на один період, оскільки він реагує на вхід, а не чекає зміни стану. Якщо на виході автомата Мілі поставити тригер, додавши цим затримку, то за часовими параметрами така

конструкція стане еквівалентною автомату Мура. Коли обиратимете тип автомата для вашого проєкту, подумайте, у який момент ви хочете побачити реакцію виходів.

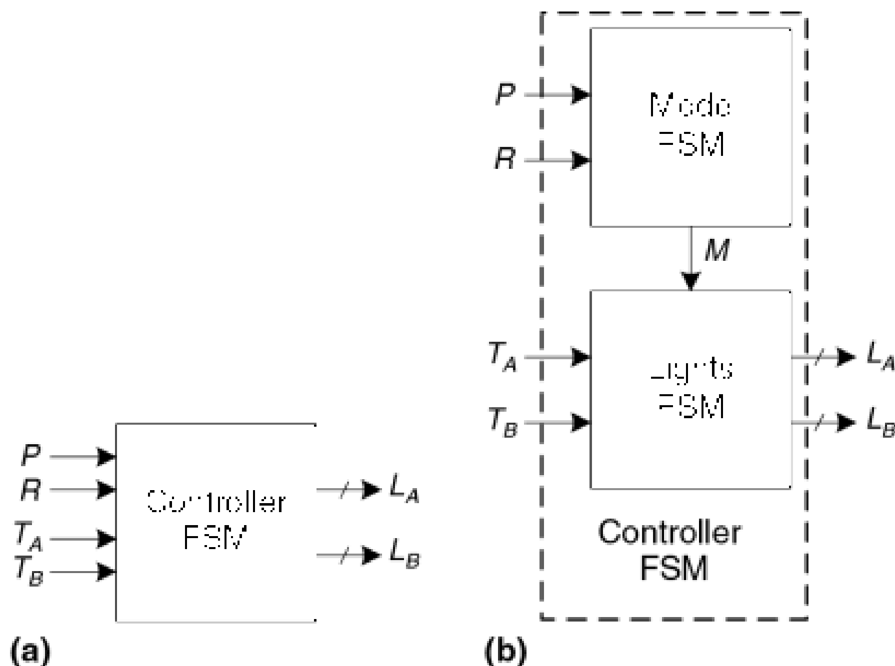


Рисунок 3.32 – Немодульна (a) та модульна (b) моделі КА модифікованого контролера світлофора

3.4.4 Декомпозиція кінцевих автоматів

Проєктування складних кінцевих автоматів часто спрощується, якщо їх можна розбити на кілька більш простих автоматів, що взаємодіють один з одним у такий спосіб, що вихід одних автоматів є входом інших. Таке застосування принципів ієрархічної організації та модульного проєктування називається декомпозицією кінцевих автоматів.

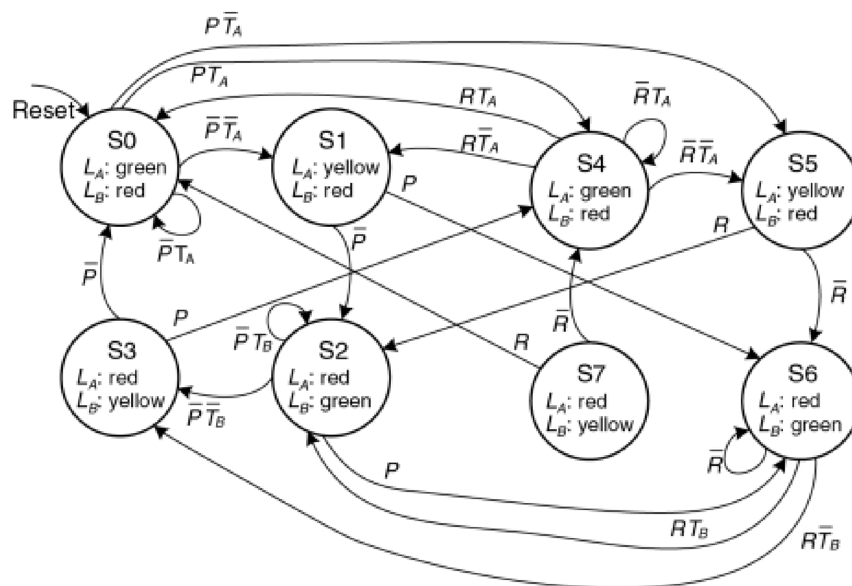
Приклад 3.8.

Модульні та немодульні кінцеві автомати

Модифікуйте контролер світлофора з п. 3.4.1 так, щоб у ньому з'явився режим «параду». У цьому режимі світлофор на Біговій вулиці залишається зеленим, коли команда та глядачі йдуть на футбольні ігри розрізненими групами. У контролера з'являються ще два входи: P і R. Отримуючи сигнал P, контролер хоча б на один цикл входить у режим параду, а отримуючи сигнал Р, хоча б на один цикл виходить із цього режиму. Перебуваючи в режимі параду, контролер проходить свою звичайну послідовність перемикань, доки L_B не стане зеленим, а потім залишається в цьому стані доти, доки режим параду не закінчиться.

Спочатку накреслимо діаграму переходів для одного-єдиного КА, як продемонстровано на рис. 3.32, *a*. Потім накреслимо діаграму переходів двох взаємодіючих КА (див. рис. 3.32, *b*). Автомат вибору режиму виставляє вихід M в одиницю, коли він перетворюється на режим параду. Автомат світлових сигналів керує світлофорами залежно від M і датчиків руху T_A та T_B.

Виконання. На рис. 3.33, *a* подано реалізацію з одним-єдиним автоматом. Стани *S*0–*S*3 відповідають за нормальний режим роботи, а стани *S*4–*S*7 – за режим параду. Дві половини діаграми практично ідентичні, за винятком того, що в режимі параду КА залишається в стані *S*6, зокрема зелене світло на вулиці Біговій. Входи *P* і *R* управляють переходами між двома половинами. Такий автомат надто складний і важкий у розробленні. На рис. 3.33, *b* запропоновано модульну реалізацію КА. У КА вибору режиму буде лише два стани: коли світлофор у нормальному й у парадному режимі. Автомат світлових сигналів модифіковано таким чином, щоб залишатися в стані *S*2, поки $M = 1$.



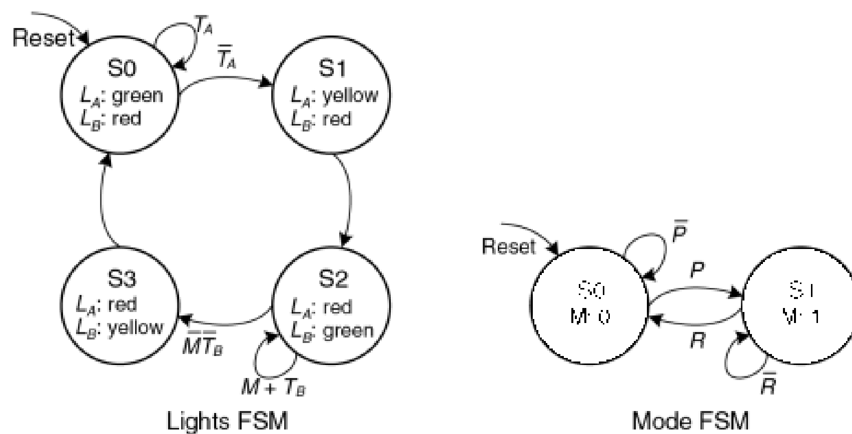


Рисунок 3.33 – Діаграми переходів: a – немодульна; b – модульна

3.4.5 Відновлення кінцевих автоматів за електричною схемою

Відновлення кінцевих автоматів за електричною схемою є процесом, оберненим до проєктування КА. Цей процес необхідний, наприклад, під час розгляду проєкту з неповною документацією чи реверсивного проєктування чиєїсь системи.

● Проаналізуйте схему, можливі стани входів, виходів та регістр станів.

● Складіть вирази для наступного стану та виходів.

● Складіть таблицю виходів і таблицю переходів.

● Викресліть із таблиці переходів стани, до яких система ніколи не потрапляє.

● Надайте ім'я кожному набору біт-станів, що використовується.

● Перепишіть таблиці виходів та переходів, застосовуючи подані позначки.

● Накресліть діаграму переходів.

● Опишіть словами те, що робить автомат.

На останньому кроці не бійтеся розгорнуто описувати цілі та функції автомата, щоб уникнути переформулювання кожного переходу з діаграми переходів.

Приклад 3.9.

Відновлення КА за його схемою

Аліса Хакер приїхала додому, але в її кодовому замку замінили проводку, тому старий код більше не працює. До замка прикріплений аркуш папера зі схемою, зображеною на рис. 3.34.

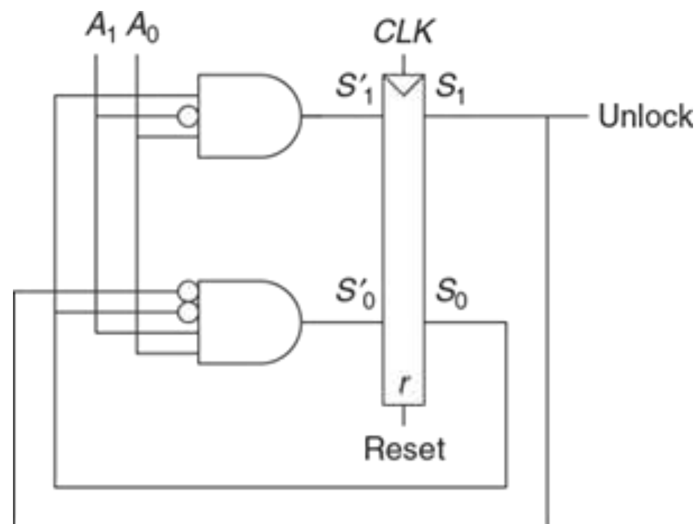


Рисунок 3.34 – Схема автомата з прикладу 3.9

Аліса вважає, що схема може бути кінцевим автоматом і вирішує відновити діаграму переходів, щоб дізнатися, чи їй це допоможе потрапити всередину.

Виконання. Аліса починає вивчати схему. Входом є A1:0, а виходом – розблокування. Біти станів вже позначено на рис. 3.34. Це автомат Мура, оскільки виходи залежить тільки від бітів стану. Прямо за схемою Аліса записує вирази для наступного стану й для виходу:

$$S_1' = S_0\, \overline{A_1}\, A_0;$$
$$S_0' = \overline{S_1}\ \overline{S_0}\, A_1\, A_0; \qquad (3.12)$$
$$U_{nlok} = S_1.$$

Потім вона складає таблиці переходів та виходу (табл. 3.17, 3.18) за написаними рівняннями. Спочатку Аліса розставляє одиниці (останні два стовпці таблиці) за виразами (3.12), а в інших місцях пише нулі.

Таблиця 3.17 – Таблиця наступних станів, відновлена за схемою (рис. 3.35)

Current State		**Input**		**Next State**	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Таблиця 3.18 – Таблиця виходів, відновлена за схемою (рис. 3.35)

Current State		**Output**
S_1	S_0	***Unlock***
0	0	0
0	1	0
1	0	1
1	1	1

Аліса скорочує таблицю, способом викреслення невикористовуваних станів і комбінування рядків. Стан $S_{1:0}$ = 11 ніде не трапляється в табл. 3.17 як можливо наступний стан, тому рядки із цим станом можна викреслити. Для поточного стану $S_{0:1}$ =10 наступний стан завжди $S_{1:0}$=00, незалежно від входів. Табл. 3.19 і 3.20 є результатом скорочення.

Таблиця 3.19 – Скорочена таблиця наступних станів

Current State		**Input**		**Next State**	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0		1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Таблиця 3.20 – Скорочена таблиця виходів

Current State		**Output**
S_1	S_0	***Unlock***
0	0	0
0	1	0
1	0	1

Дівчина дає імена кожної комбінації бітів станів: $S0$ – це $S_{1:0}$ = 00, $S1$ – це $S_{1:0}$ = 01, а $S2$ – це $S_{1:0}$ = 10. Аліса переписує табл. 3.19 і 3.20 у табл. 3.21 та 3.22, використовуючи ці позначки.

Таблиця 3.21 – Символьна таблиця наступних станів

Current State	**Input**	**Next State**
S	***A***	***S′***
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Таблиця 3.22 – Символьна таблиця виходів

Current State *S*	**Output** ***Unlock***
S0	0
S1	0
S2	1

Відповідно до тбл. 3.21 і 3.22 вона рисує діаграму переходів, яку подано на рис. 3.35. Вивчивши її, Аліса робить висновок, що кінцевий автомат розблокує двері після виявлення поданих на вхід $A_{1:0}$ трьох одиниць. Потім двері знову блокуються. Вона намагається ввести цей код – і двері відчиняються.

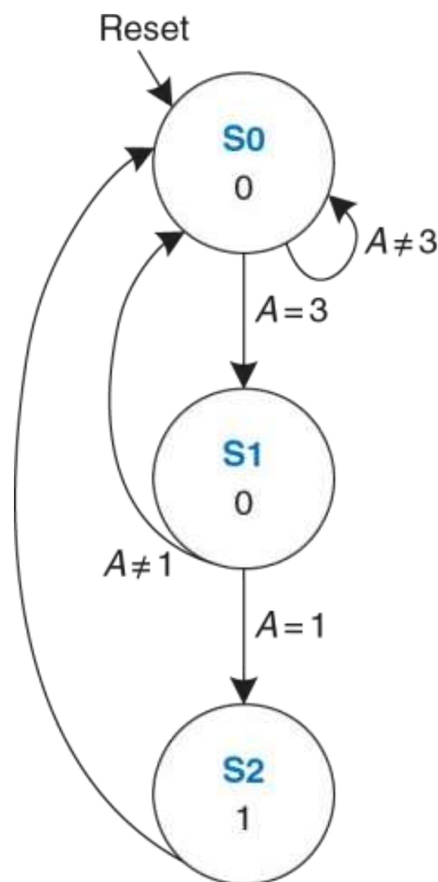


Рисунок 3.35 – Діаграма переходів отриманого КА

3.4.6 Огляд кінцевих автоматів

Кінцеві автомати є потужним інструментом для систематичного проєктування схем послідовності за технічним завданням. Використовуйте таку послідовність дій для створення КА:

- визначте входи та виходи;
- нарисуйте діаграму переходів;

● для автомата Мура:
– складіть таблицю переходів;
– складіть таблицю виходів;
● для автомата Мілі:
– складіть об'єднану таблицю виходів і переходів;
● оберіть метод кодування станів – обраний метод вплине на схемотехнічну реалізацію;
● складіть булеві вирази для наступного стану та вихідної комбінаційної схеми;
● накресліть принципову схему.

Ми неодноразово використовуватимемо КА для створення складних цифрових систем у межах цього посібника.

3.5 Синхронізація послідовних схем

Згадайте, що тригер копіює сигнал із *D*-входу на *Q*-вихід по передньому фронту тактового сигналу. Цей процес називається фіксацією (*sampling*) *D*-сигналу на фронті тактового імпульсу. Поведінка тригера коректна, якщо сигнал на *D*-вході стабільний (рівний 0 чи 1 і змінюється) протягом переднього фронту тактового сигналу. Але що станеться, якщо сигнал *D* не буде стабільним під час зміни тактового сигналу?

Ця ситуація аналогічна тій, що виникає під час спуску затвора фотокамери. Уявіть, що ви намагаєтеся зняти стрибок жаби з листка латаття, що плаває в озері. Якщо натиснете на спуск перед стрибком, то на фотографії побачите жабу на зеленому листі. Якщо натиснете на спуск після стрибка, то на фотографії будуть брижі на воді. Але якщо ви натиснете на спуск під час стрибка, то на світлині побачите змазане зображення витягнутої вздовж напрямку стрибка жаби. Однією з властивостей фотокамери є апертурний час, протягом якого об'єкт, якого фотографують, має бути нерухомий, щоб на світлині сформувалося його різке зображення. Так само послідовний елемент має апертурний час до й після фронту тактового сигналу, протягом якого його інформаційні сигнали мають бути стабільними, щоб на виході тригера сформувався коректний сигнал.

Частина апертурного часу послідовного елемента до фронту тактового імпульсу називається часом передустановлення (*setup time*), після фронту – часом утримання (*hold time*). Подібно до статичної дисципліни, що дає змогу використовувати логічні рівні тільки за межами забороненої зони, динамічна

дисципліна уможливлює застосування тільки тих сигналів, які змінюються поза апертурним часом. У процесі виконання вимог динамічної дисципліни можемо оперувати дискретними одиницями часу, що називаються тактовими циклами, аналогічно до того, як ми оперуємо дискретними логічними рівнями 1 і 0. Сигнал може змінюватися та осцилювати протягом деякого обмеженого проміжку часу. У виконанні вимог динамічної дисципліни важливим є лише його значення в кінці циклу тактового сигналу, коли він уже набув стабільного значення. Отже, для опису сигналу *A* можна використовувати його величину *A[n]* наприкінці *n*-го циклу тактового імпульсу, де *n* – ціле число, замість його величини *A(t)* у довільний час *t*, де *t* – дійсне число.

Період тактових імпульсів має бути досить великим, щоб перехідні процеси всіх сигналів встигли завершитися. Ця вимога обмежує швидкодію всієї системи. У реальних системах тактові імпульси надходять на входи тригерів неодночасно. Цей розкид часу, що називається розфазуванням, або розкидом фаз тактового сигналу (*clock skew*), змушує розробників додатково збільшувати період тактових сигналів.

Іноді неможливо задовольнити вимоги динамічної дисципліни, особливо в пристроях поєднання цифрової системи з реальним світом. Наприклад, розглянемо схему, до входу якої під'єднано кнопку. Мавпа може натиснути кнопку якраз під час фронту тактового імпульсу. Це може призвести до виникнення явища, що називається метастабільністю, у цьому разі тригер виявляється в проміжному стані між 0 і 1, до того ж перехід у коректний логічний стан (0 або 1) може відбуватися нескінченно довго. Розв'язанням проблеми асинхронних входів є використання синхронізатора, на виході якого некоректний логічний рівень може виникнути з дуже малою (але не нульовою) ймовірністю.

Ці ідеї детально розглядатимуться нижче в межах цього розділу.

3.5.1 Динамічна дисципліна

Досі розглядали функціональні специфікації послідовних схем. Згадайте, що синхронні послідовні схеми, зокрема тригери або кінцеві автомати, мають також і часову специфікацію (її приклад зображено на рис. 3.36).

Після переходу 0→1 тактового сигналу (переднього фронту тактового імпульсу) вихід (або виходи) схеми можуть почати змінюватися не раніше, ніж за час t_{CCQ} (затримка реакції *clock-to-Q, contamination delay clock-to-Q3*), і мають набути стаціонарного значення не пізніше, ніж за час t_{PCQ} (затримка розповсюдження *clk-to-Q, propagation delay clock-to-Q*). Ці величини

є найменшою й найбільшою затримкою схеми відповідно. Для того, щоб фіксування було коректним, інформаційний вхід (або входи) схеми має бути стабільним упродовж деякого часу передустановлення (*setup time*) t_{SETUP} перед переднім фронтом тактового сигналу й не має змінюватися протягом часу утримання (*hold time*) t_{HOLD} після переднього фронту тактового сигналу. Сума часу – це загальний час, упродовж якого інформаційний вхідний сигнал має бути стабільним для його фіксації на виході.

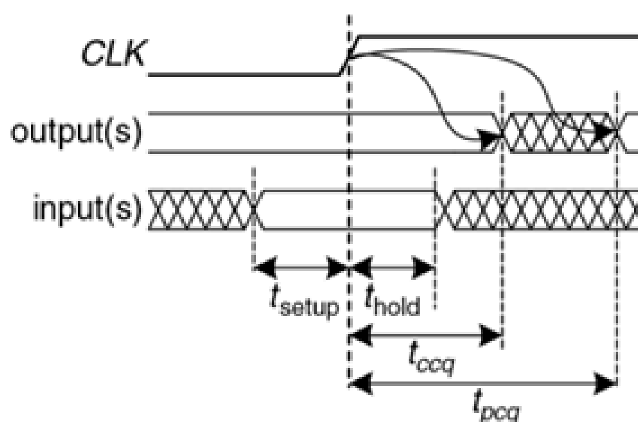


Рисунок 3.36 – Часова специфікація синхронної послідовної схеми

Динамічна дисципліна вимагає, щоб входи синхронної послідовності були стабільні протягом часу передустановлення до й часу утримання після фронту тактового імпульсу. Дотримання цих вимог гарантує, що в процесі фіксування значення інформаційного входу тригером не змінюватиметься. Оскільки розглядатимемо тільки значення вхідних сигналів, що встановилися в моменти часу, коли вони фіксуються, ми можемо вважати сигнали дискретними як за рівнем, так і за часом.

3.5.2 Часові властивості системи

Період тактового сигналу або тривалістю циклу синхронізації, Tc, називається проміжок часу між передніми фронтами послідовних тактових імпульсів. Зворотна величина $fc = 1/Tc$ називається тактовою частотою. Збільшення тактової частоти без зміни інших параметрів схеми призводить до збільшення її продуктивності. Частота вимірюється в герцах (Гц), або циклах за одну секунду: 1 мегагерц (МГц) = 10^6 Гц, і 1 гігагерц (ГГц) = 10^9 Гц.

На рис. 3.37, *а* подано структуру тракту оброблення інформації синхронної послідовної схеми, для якої розрахуємо період тактового сигналу. На передньому фронті тактового імпульсу на виході регістра *R*1 формується вихідний сигнал (або сигнали) *Q*1. Ці сигнали надходять на вхід блоку

комбінаційної логіки, вихідні сигнали цього блоку надходять на вхід (або входи) *D*2 регістра *R*2. Як зображено на рис. 3.37, *b*, вихідний сигнал блоку може почати змінюватися не раніше завершення часу реакції після припинення зміни його вхідного сигналу й набуває остаточного значення за максимальний час затримки поширення від моменту встановлення вхідного сигналу. Перші стрілки показують мінімальну затримку, зважаючи на *R*1 та комбінаційну логіку, а інші – максимальну затримку поширення у тракті регістра *R*1 – комбінаційна логіка.

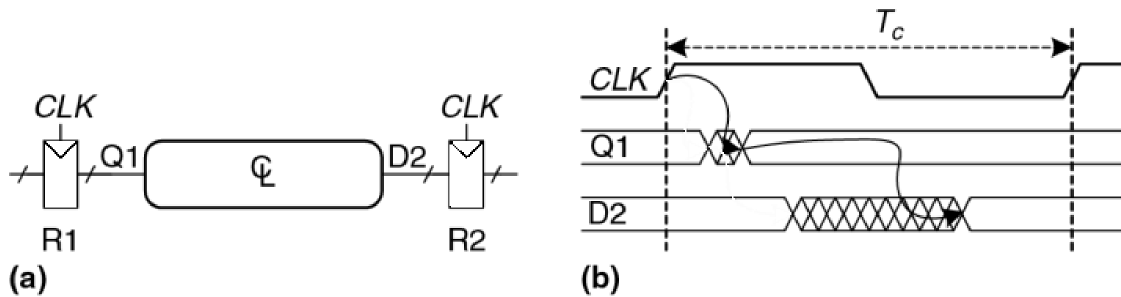


Рисунок 3.37 – Тракт між регістрами й часова діаграма

Ми проаналізуємо часові обмеження з огляду на час попереднього встановлення та утримання другого регістра R2.

Обмеження часу попереднього встановлення

На рис. 3.38 на часовій діаграмі подано лише максимальну затримку в тракті оброблення інформації.

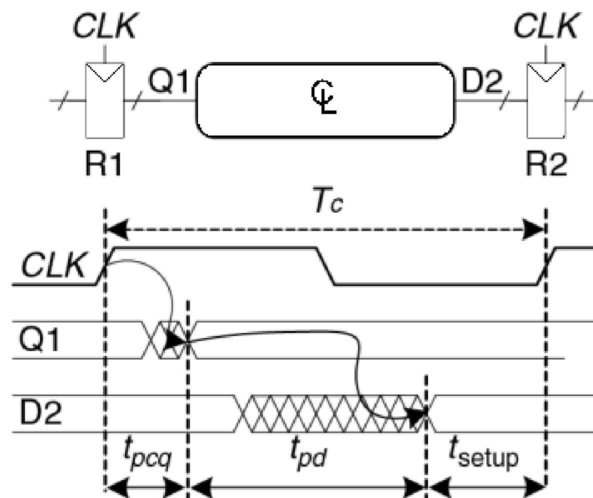


Рисунок 3.38 – Максимальна затримка для обмеження часу передустановлення

Для виконання обмеження часу передустановлення регістра *R*2 сигнал *D*2 має встановитися не пізніше, ніж за час передустановлення до фронту наступного тактового імпульсу.

Отже, можемо отримати вираз для мінімальної тривалості періоду синхросигналу:

$$Tc \geq t_{PCQ} + t_{PD} + t_{SETUP}. \quad (3.13)$$

У проєктуванні комерційних продуктів період тактового сигналу майбутнього виробу часто задає з міркувань конкурентоспроможності керівник відділу розробок чи відділу маркетингу. Крім того, затримка розповсюдження сигналу тригером від фронту тактового сигналу до виходу (*Clock-to-Q*) і час встановлення t_{PCQ} і t_{SETUP} визначені виробником. Отже, нерівність (3.13) необхідно перетворити на визначення максимальної затримки поширення комбінаційної схеми, оскільки зазвичай це єдиний параметр, що може змінювати проєктувальник:

$$t_{PD} \leq Tc - (t_{PCQ} + t_{SETUP}). \quad (3.14)$$

Доданок у дужках, $t_{PCQ} + t_{SETUP}$, називається втратами на впорядкування (*sequencing overhead*). В ідеальному випадку весь період тактового сигналу може бути витрачено на обчислення комбінаційної логіки (час t_{PD}). Проте втрати на упорядкування в тригерах зменшують цей час. Нерівність (3.14) називається обмеженням часу передустановлення або обмеженням максимальної затримки, оскільки залежить від часу передустановлення та обмежує максимальну затримку поширення в комбінаційній логічній схемі.

Якщо затримка поширення в комбінаційній схемі занадто велика, то вхід *D*2 може не встигнути прийняти свій стан до часу, коли регістр *R*2 чекає стабільності та фіксує його. Отже, *R*2 може зафіксувати некоректний результат чи навіть логічний рівень у забороненій зоні. У такому разі схема працюватиме некоректно. Проблему можна розв’язати збільшенням періоду тактового сигналу або переглядом комбінаційної схеми з метою досягти меншої затримки розповсюдження.

Обмеження часу утримання

Регістр *R*2 (рис. 3.37, *а*) має також обмеження часу утримання. Його вхід *D*2 не має змінюватися протягом деякого часу після переднього фронту тактового імпульсу.

Відповідно до рис. 3.39 *D*2 може змінитися за час $t_{CCQ} + t_{CD}$ після переднього фронту тактового імпульсу. Отже, можна записати:

$$t_{CCQ} + t_{CD} \geq t_{HOLD}. \quad (3.15)$$

Як і раніше, властивості тригера t_{CCQ} і t_{HOLD}, що використовуються в схемі, зазвичай перебувають поза впливом розробника схеми. Після простих

перетворень можемо записати нерівність мінімальної затримки комбінаційної логічної схеми:

$$t_{CD} \geq t_{HOLD} - t_{CCQ}. \quad (3.16)$$

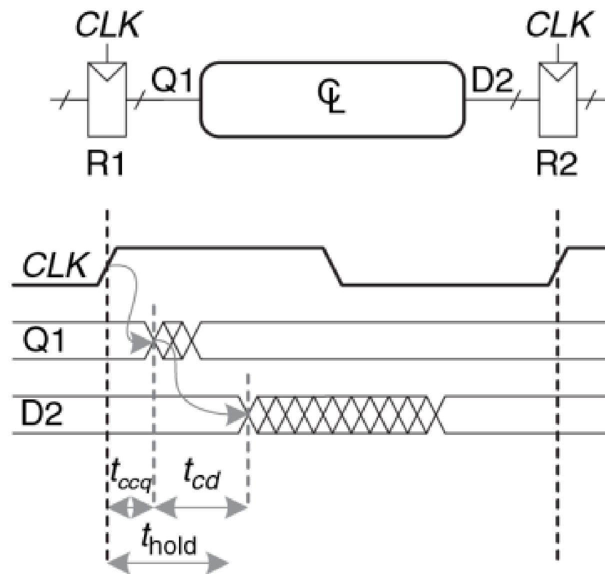


Рисунок 3.39 – Мінімальна затримка для обмеження часу утримання

Нерівність (3.16) також називається обмеженням часу утримання або обмеженням мінімальної затримки, оскільки вона обмежує мінімальну затримку комбінаційної схеми.

Припускаємо, що за умови поєднання логічних елементів між собою часові проблеми синхронізації не виникають. Зокрема, вважаємо, що з безпосереднім послідовним з'єднанням двох тригерів (рис. 3.40) проблеми, зумовлені часом утримання, не виникають.

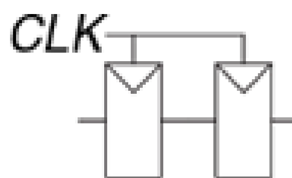


Рисунок 3.40 – Безпосереднє послідовне з'єднання тригерів

У цьому разі внаслідок відсутності комбінаційної логіки між тригерами $t_{CD} = 0$. За такої підстановки нерівність (3.16) зводиться до вимоги:

$$t_{HOLD} \leq t_{CCQ}. \quad (3.17)$$

Іншими словами, час утримання надійного тригера має бути меншим, ніж затримка його реакції. Часто тригери проєктуються так, що $t_{HOLD} = 0$, отже, нерівність (3.17) завжди виконується. У цьому посібнику, якщо не зазначено протилежне, вважатимемо таке припущення істинним та ігноруємо обмеження часу утримання.

Проте обмеження часу утримання є критично важливими. Якщо вони порушуються, то єдиним рішенням є збільшення затримки реакції комбінаційної схеми, що потребує її перепроєктування. Такі порушення, на відміну від порушень обмежень часу передустановлення, неможливо розв’язати за допомогою зміни періоду тактового сигналу. Перепроєктування інтегральної мікросхеми та виробництво її виправленого варіанта займає кілька місяців і потребує витрат у кілька мільйонів доларів за сучасних технологій, тому до порушень обмеження часу утримання потрібно ставитися вкрай серйозно.

Висновок

Послідовні схеми мають обмеження часу передустановлення та утримання, що встановлюють максимальну та мінімальну затримки в комбінаційній логічній схемі між тригерами. Сучасні тригери зазвичай спроєктовані так, що мінімальна затримка в комбінаційній логіці дорівнює нулю, тобто тригери можуть бути розміщені один за одним. Максимальна затримка обмежує кількість послідовних логічних елементів, увімкнених один за одним у критичному шляху швидкодіючої схеми.

Приклад 3.10.

Часовий аналіз

Бен Бітдідл спроєктував схему (див. рис. 3.41). Відповідно до специфікації компонентів, які він використовував, затримка реакції тактовий вхід-вихід тригерів дорівнює 30 пс, а затримка поширення – 80 пс. Вони мають час попереднього встановлення 50 пс і час утримання – 60 пс. У логічних елементів затримка поширення дорівнює 40 пс, затримка реакції – 25 пс. Допоможіть Бену визначити максимальну тактову частоту його схеми та з’ясувати, чи можуть порушуватися обмеження часу утримання в ній. Цей процес називається часовим аналізом.

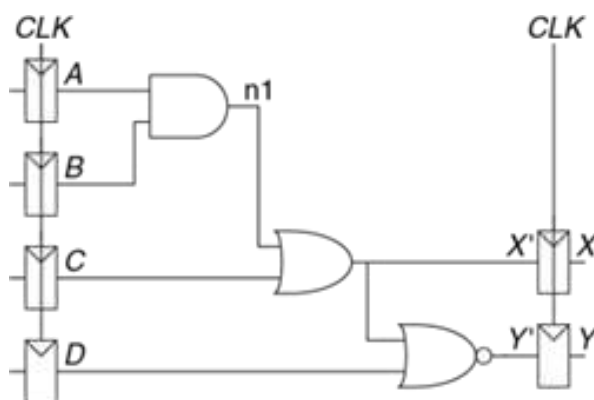


Рисунок 3.41 – Приклад схеми для часового аналізу

Виконання. На рис. 3.42, *a* наведено часові діаграми сигналів, що показують, коли вони можуть змінюватися. Сигнал на входи *A* – *D* надходить з регістрів, тому можуть змінитися за короткий час після переднього фронту сигналу *CLK*.

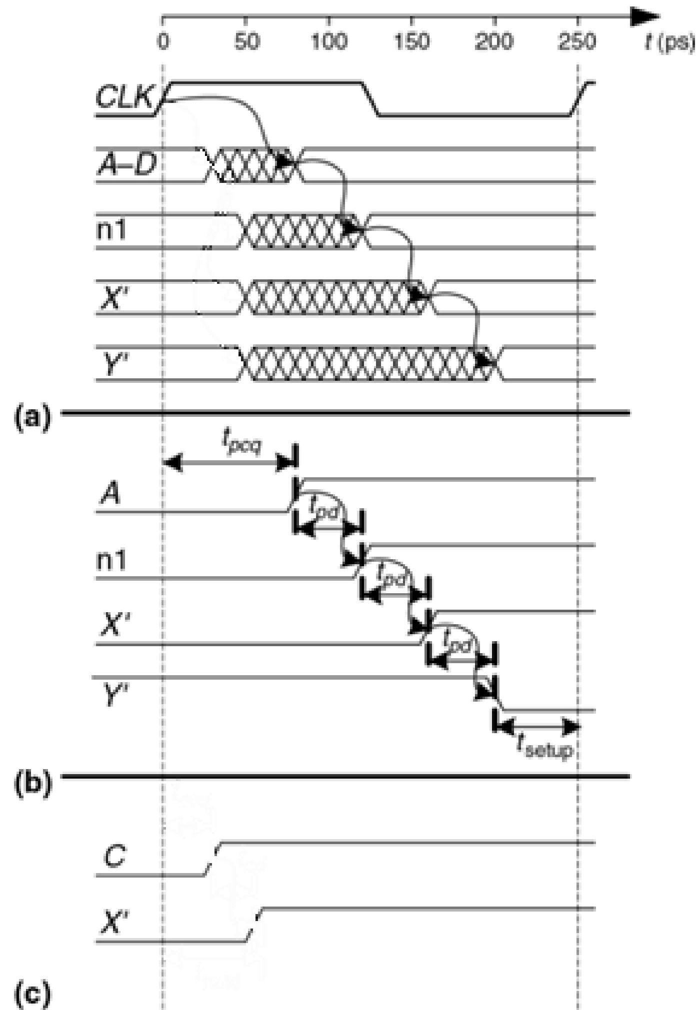


Рисунок 3.42 – Часова діаграма:
a – загальний випадок; b – критичний шлях; c – короткий шлях

Критичний шлях виникає, коли *B* = 1, *C* = 0, *D* = 0 і *A* змінюється від 0 до 1, що призводить до перемикання *n*1 в 1, *X* в 1, *Y* в 0 (див. рис. 3.42, *b*). Цей шлях передбачає затримки трьох логічних елементів. Для оцінювання затримки в критичному шляху вважатимемо, що затримка кожного елемента дорівнює затримці розповсюдження. Сигнал *Y'* має встановитись раніше від наступного переднього фронту *CLK*. Отже, мінімальна тривалість циклу дорівнює

$$Tc \geq t_{PCQ} + 3\ t_{PD} + t_{SETUP} = 80 + 3 * 40 + 50 = 250 \text{ пс.} \qquad (3.18)$$

Максимальна тактова частота дорівнює $fc = 1/Tc = 4$ ГГц.

Короткий (за часом проходження сигналу) шлях виникає, коли *A* = 0 і *C* перемикається до 1, як зображено на рис. 3.42, c.

Для короткого шляху вважатимемо, що кожен логічний елемент перемикається відразу після завершення затримки реакції. Цей шлях містить лише один елемент, тому перемикання може відбутися за $t_{CCQ} + t_{CD} = 30 + 25 = 55$ пс. Однак необхідно пам'ятати, що час утримання тригера дорівнює 60 пс. Це значить, що сигнал *X* обов'язково має бути стабільним протягом 60 пс після переднього фронту тактового сигналу *CLK*, щоб тригер зміг надійно зафіксувати величину сигналу *X*. У цьому разі протягом першого переднього фронту *CLK* $X' = 0$, тобто тригер має зафіксувати 0. Однак, оскільки *X′* не підтримується стабільним протягом часу утримання, дійсне значення *X* буде непередбачуваним. У цій схемі порушуються обмеження часу утримання, і її поведінка непередбачувана за будь-якої тактової частоти.

Приклад 3.11.

Виправлення порушень часу утримання

Аліса Хакер пропонує виправити схему Бена способом додавання буферних елементів, що уповільнюватимуть проходження сигналу крізь короткий шлях, як зображено на рис. 3.43. Буфери мають таку саму затримку, як і решта логічних вентилів. Визначте максимальну тактову частоту та перевірте, чи виникатимуть проблеми, пов'язані з часом утримання.

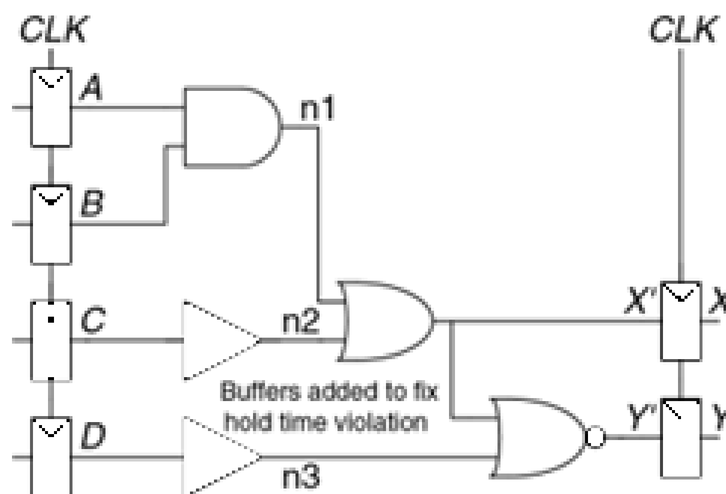


Рисунок 3.43 – Виправлена схема, у якій відсутні порушення обмеження часу утримання

Виконання. На рис. 3.44 подано часові діаграми, що показують, коли сигнали можуть змінюватися. Критичний шлях від *A* до *Y* не змінився, тому що не проходить крізь буфери. Отже, максимальна тактова частота дорівнює, як і раніше, 4 ГГц. Однак час проходження сигналу крізь короткий шлях буде

збільшено на мінімальну величину затримки буферів. Тепер *X′* не зміниться протягом $t_{CCQ} + 2\,t_{CD} = 30 + 2 * 25 = 80$ пс після фронту тактового сигналу. Отже, *X′* буде стабільний упродовж часу утримання 60 пс, тобто схема працюватиме правильно.

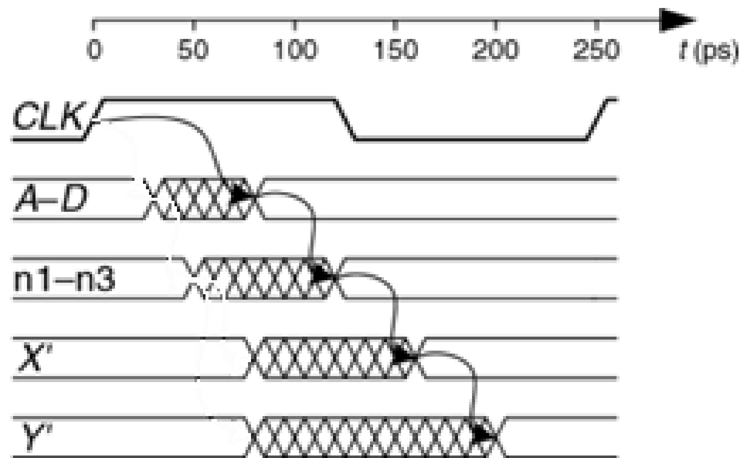


Рисунок 3.44 – Часова діаграма схеми з буферами, де відсутні порушення обмеження часу утримання

У цьому прикладі аномально великий час утримання було використано лише для демонстрації сутності проблем, пов'язаних із часом утримання. Більшість тригерів спроєктовано так, що $t_{HOLD} < t_{CCQ}$, це дає змогу уникнути таких проблем. Однак у деяких високопродуктивних мікропроцесорах, зокрема *Pentium* 4, замість тригерів використовується елемент, який називається «імпульсна засувка» (*pulsed latch*). Імпульсна засувка поводиться подібно до звичайного тригера, але має незначну затримку тактовий вхід-вихід і великий час утримання. Додавання буферів дає змогу часто, але не завжди усунути проблеми, пов'язані з обмеженням часу утримання, без збільшення часу проходження сигналу критичним шляхом.

3.5.3 Розфазування тактових сигналів

Ми розглядали, що тактові імпульси надходять на всі регістри одночасно. Насправді існує певний розкид цього часу. Ця неодночасність фронтів називається розфазуванням.

Наприклад, довжина провідників, якими тактові сигнали надходять на різні регістри, може бути різною. Це призводить до різних часів затримки, як зображено на рис. 3.45.

Шум також спричиняє різні затримки. Стробування тактових сигналів, описане в п. 3.2.5, зумовлює їх додаткову затримку. Якщо в схемі використовуються стробовані та нестробовані тактові сигнали, то між ними

буде суттєва неузгодженість. На рис. 3.45 сигнал *CLK*2 випереджатиме за часом сигнал *CLK*1 через складний шлях тактового сигналу між регістрами. Якщо трасування ланцюга тактового сигналу буде виконано інакше, ситуація може бути протилежною, *CLK*2 відставатиме від сигналу *CLK*1. У процесі виконання часового аналізу розглядаємо найгірший випадок, що дає змогу гарантувати, що схема працюватиме за всіх умов.

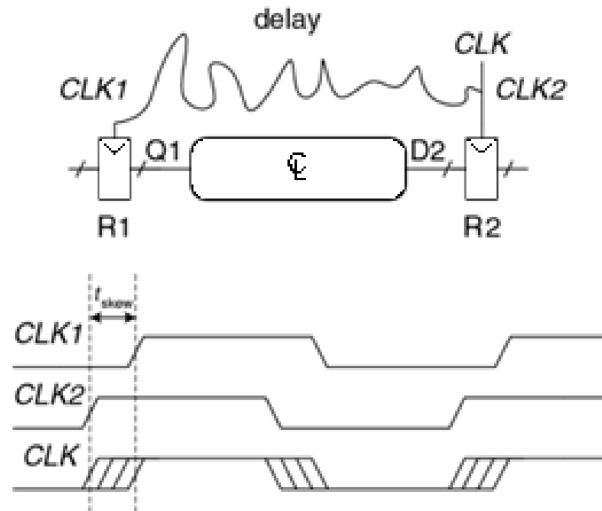


Рисунок 3.45 – Розфазування тактових сигналів, зумовлене затримками в міжз'єднаннях

Облік розфазування змінює часову діаграму, яка була зображена на рис. 3.38, модифікована діаграма подана на рис. 3.46.

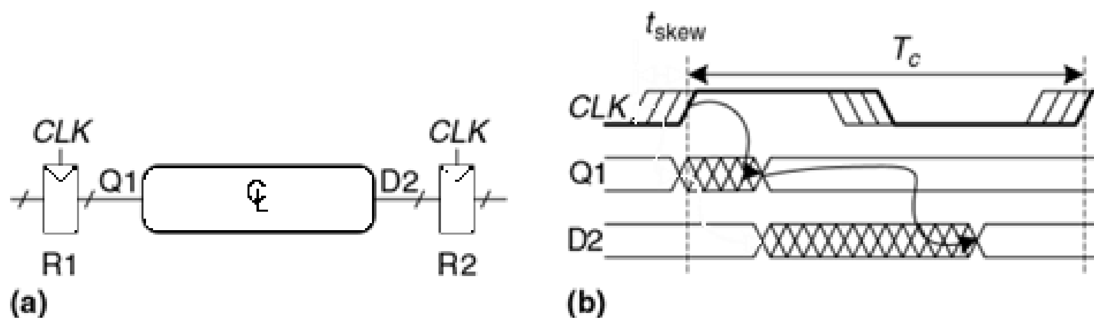


Рисунок 3.46 – Часова діаграма з огляду на розфазування тактових імпульсів

Жирною лінією позначена максимальна затримка тактового сигналу, тонкі лінії показують, що синхросигнал може з'явитися на t_{SKEW} раніше.

Спочатку розглянемо обмеження часу попереднього встановлення, (відповідні діаграми зображені на рис. 3.47).

У гіршому випадку на регістр *R*1 надходить тактовий сигнал з найбільшою затримкою, а на *R*2 – з найменшою, що залишає мінімальний час для проходження даних крізь комбінаційну схему між регістрами.

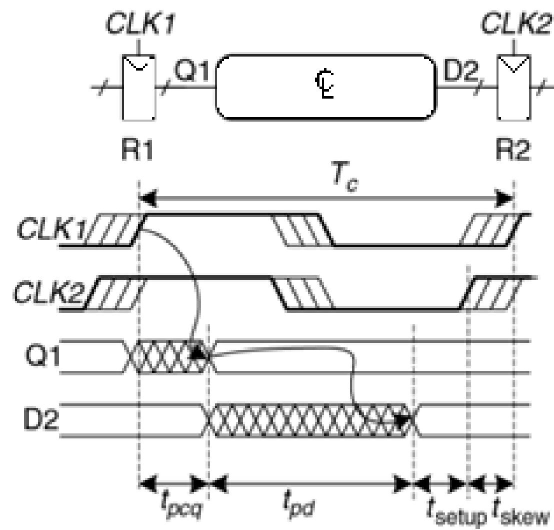


Рисунок 3.47 – Обмеження часу попереднього встановлення
з огляду на розфазування тактових імпульсів

На вхід регістра *R*2 дані надходять крізь регістр *R*1 і комбінаційну логіку. Вони мають дійти стаціонарного стану перед початком їх фіксації регістром *R*2. Отже, можна дійти невтішного висновку, що

$$Tc \geq t_{PCQ} + t_{PD} + t_{SETUP} + t_{SKEW}; \quad (3.19)$$

$$t_{CD} \geq t_{HOLD} + t_{SKEW} - t_{CCQ}. \quad (3.20)$$

Далі розглянемо обмеження часу утримання (див. рис. 3.48). У гіршому випадку на регістр *R*1 надходить тактовий сигнал із найменшою затримкою, але в *R*2 – з максимальною. Інформація може швидко пройти крізь регістр *R*1 та комбінаційну логіку, але має надійти на вхід регістра *R*2 не раніше закінчення часу утримання після переднього фронту тактового імпульсу.

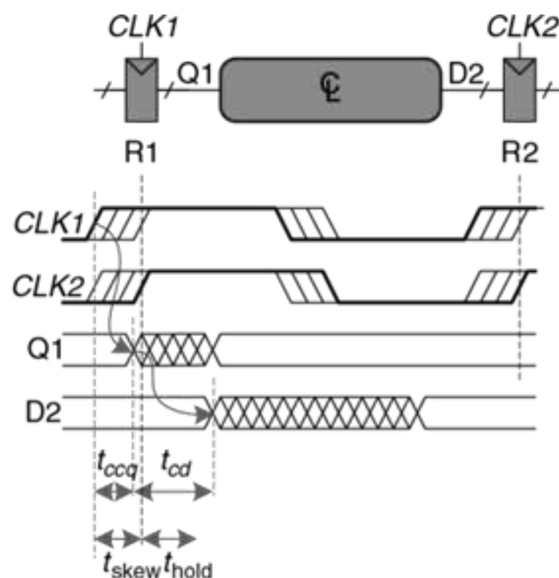


Рисунок 3.48 – Обмеження часу утримання
з огляду на розфазування тактових імпульсів

Так, можна записати:

$$t_{CCQ} + t_{CD} \geq t_{HOLD} + t_{SKEW}; \quad (3.21)$$

$$t_{CD} \geq t_{HOLD} + t_{SKEW} - t_{CCQ}. \quad (3.22)$$

Отже, розфазування тактових імпульсів призводить до ефективного збільшення часу передустановлення та часу утримання. Це призводить до зростання витрат на впорядкування та зменшує час, доступний для оброблення даних комбінаційною схемою. Навіть якщо $t_{HOLD} = 0$, пара послідовно з'єднаних тригерів порушуватиме нерівність (3.22), якщо $t_{SKEW} > t_{CCQ}$. Щоб запобігти таким серйозним порушенням обмежень часу утримання, проєктувальник має обмежувати розфазування тактових сигналів. Іноді тригери спеціально проєктуються повільними (час t_{CCQ} тривалий), щоб уникнути проблем часу утримання, навіть якщо розфазування тактових сигналів суттєве [1].

Приклад 3.12.

Часовий аналіз розфазування тактових імпульсів

Виконайте завдання прикладу 3.10 у припущенні, що в системі є розфазування тактових імпульсів величиною 50 пс.

Виконання. Критичний шлях залишається без змін, але ефективний час встановлення збільшується внаслідок розфазування. Отже, мінімальний період тактового сигналу дорівнює

$$Tc \geq t_{PCQ} + 3t_{PD} + t_{SETUP} + t_{SKEW} = 80 + 3 \cdot 40 + 50 + 50 = 300 \text{ пс}. \quad (3.23)$$

Максимальна частота тактового сигналу $f = 1/Tc$ = 3,33 ГГц.

Короткий шлях також залишається без змін, час проходження сигналу ним дорівнює 55 пс. Ефективний час утримання збільшується на величину розфазування до 60 + 50 = 110 пс, що значно більше ніж 55 пс. Отже, у схемі буде порушено обмеження часу утримання, і вона некоректно працюватиме за будь-якої частоти тактового сигналу. Нагадаємо, що в цій схемі обмеження часу утримання було порушено й без розфазування. Розфазування тактових сигналів лише погіршило ситуацію.

Приклад 3.13.

Виправлення порушення обмеження часу

Повторіть вправу прикладу 3.11 у припущенні, що в системі є розфазування тактових імпульсів величиною 50 пс.

Виконання. Критичний шлях не змінюється, тому максимальна тактова частота залишається рівною 3,33 ГГц. Час проходження сигналу коротким шляхом збільшується до 80 пс. Це все ще менше,

ніж t_{HOLD} + t_{SKEW} = 110 пс, отже, у схемі порушуються обмеження часу утримання. Щоб розв’язати проблему, до схеми необхідно додати ще кілька буферів. Оскільки вони містяться в критичному шляху, то максимальна тактова частота зменшиться. Як альтернативу можна розглянути використання інших тригерів із меншим часом утримання.

3.5.4 Метастабільність

Як було зазначено раніше, не завжди можна гарантувати, що вхід послідовної схеми буде стабільний протягом апертурного часу, особливо якщо вхідний сигнал надходить від зовнішнього асинхронного джерела. Розглянемо кнопку, приєднану до входу тригера, як показано на рис. 3.49. Коли кнопка не натиснута, *D* = 0. Коли кнопка натиснута, *D* = 1. Мавпа може натискати кнопку будь-який довільний час щодо фронту тактового сигналу. Ми хочемо знати сигнал на вихід *Q* після переднього фронту сигналу *CLK*. У випадку І, коли кнопка натискається задовго до фронту *CLK*, *Q* = 1. У випадку II кнопка натискається лише набагато пізніше за фронт *CLK*, *Q* = 0. Але у випадку III, коли кнопка натискається в проміжок, що охоплює час передустановлення перед фронтом тактового імпульсу та час утримання після нього, вхідний сигнал порушує динамічну дисципліну й вихід буде невизначеним.

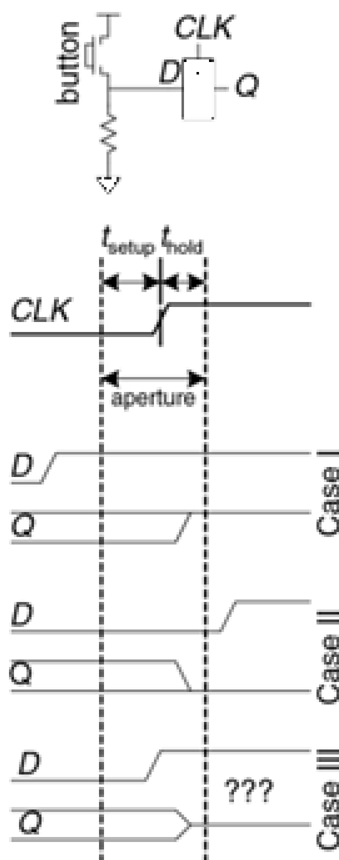


Рисунок 3.49 – Вхідний сигнал, що змінюється
до, після або протягом апертурного часу

Метастабільний стан

Коли стан інформаційного входу тригера змінюється протягом апертурного часу, з його виходу *Q* може на деякий час з’явитися напруга в діапазоні від 0 до *VDD*, тобто в забороненій зоні. Такий стан називається метастабільним. Із часом вихід тригера перейде в стабільний стан 0 або 1. Однак час дозволу, необхідний для досягнення стабільного стану, не обмежується.

Метастабільний стан тригера подібний до стану кульки на вершині між двома западинами (див. рис. 3.50). Положення в западинах є стабільними, оскільки кулька перебуватиме в них необмежено тривалий час за відсутності зовнішнього збурення.

Положення на вершині височини називається метастабільним, тому що кулька перебуватиме в ньому тільки за умови ідеального балансування. Але, оскільки у світі немає нічого досконалого, згодом кулька скотиться в одну із западин. Необхідний для цього час залежить від ступеня початкового балансування кульки. Кожен бістабільний пристрій має метастабільний стан між двома стабільними.

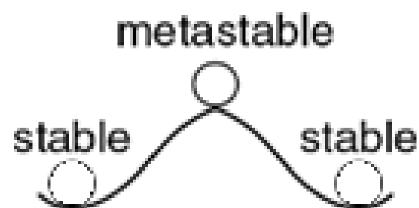


Рисунок 3.50 – Стабільний та метастабільний стан

Час дозволу

Якщо вхід тригера змінюється в довільний момент циклу тактового сигналу, час дозволу t_{RES}, необхідний для переходу в стабільний стан, також є випадковою величиною. Якщо вхід змінюється поза апертурним часом, то $t_{RES} = t_{PCQ}$. Але якщо відбудеться зміна входу в апертурний час, t_{RES} може бути значно більшим.

Теоретичний та експериментальний розгляд (див. п. 3.5.6) демонструє, що ймовірність того, що час дозволу перевищує деякий час *t*, падає експоненційно зі зростанням *t*:

$$P(t_{RES} > t) = \frac{T_0}{T_c} e^{\frac{t}{\tau}}, \quad (3.24)$$

де T_c – період тактового сигналу, T_0 та τ – властивості тригера. Вираз справедливий тільки, якщо *t* набагато триваліший, ніж t_{PCQ}.

Інтуїтивно зрозуміло, що відношення $T0/Tc$ описує ймовірність того, що вхід зміниться в невдалий час (тобто в апертурний); ця можливість зменшується зі зростанням періоду тактового сигналу Tc.

Часова константа τ позначає, наскільки швидко тригер виходить із метастабільного стану; вона пов'язана із затримкою в перехресно з'єднаних вентилях тригера.

Отже, якщо вхід бістабільного пристрою, такого як тригер, змінюється впродовж апертурного часу, його вихід може деякий час перебувати в метастабільному стані, перш ніж перейти в стабільний стан 0 або 1. Час переходу в стабільний стан не обмежений, оскільки для будь-якого кінцевого часу t ймовірність того, що тригер усе ще перебуває в метастабільному стані, не дорівнює нулю. Однак ця ймовірність експоненційно падає зі зростанням t. Отже, якщо почекати тривалий час, набагато більший, ніж t_{PCQ}, то з дуже високою ймовірністю очікується, що тригер досягне коректного логічного стану.

3.5.5 Синхронізатор

Наявність асинхронних входів цифрової системи, що приймають інформацію із зовнішнього світу, є неминучою. Наприклад, сигнали, які формує людина, асинхронні. Такі асинхронні входи, якщо до них ставитися недбало, можуть призвести до появи метастабільних станів у системі, що спричинить непередбачувані відмови, які дуже складно відстежити та виправити. За наявності асинхронних входів проєктувальник системи має забезпечити досить малу ймовірність появи метастабільних напруг. Значення слова «достатньо» залежить від контексту. Для стільникового телефона, ймовірно, одна відмова за 10 років допустима, оскільки користувач може завжди вимкнути та увімкнути апарат, якщо він «зависне». Для медичного приладу кращим є одна відмова за передбачуваний час існування всесвіту (10^{10}років). Щоб гарантувати правильність логічних рівнів, усі асинхронні входи мають пройти крізь синхронізатори.

Синхронізатор, як зображено на рис. 3.51, є пристроєм, на вхід якого надходить асинхронний сигнал D і тактовий сигнал CLK. За обмежений час він формує вихідний сигнал Q, що дуже ймовірно має коректний логічний рівень. Якщо вхід D стабільний протягом апертурного часу, вихід Q має набути значення входу. Якщо D змінюється протягом апертурного часу, Q може набути значення 0 або 1, але не має бути метастабільним.

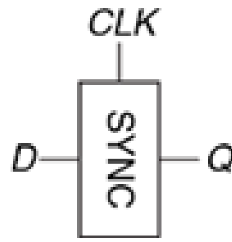


Рисунок 3.51 – Символ синхронізатора

На рис. 3.52 продемонстровано, як із двох тригерів можна побудувати простий синхронізатор. Тригер *F*1 фіксує значення вхідного сигналу *D* переднього фронту тактового сигналу *CLK*. Якщо *D* змінюється в апертурний час, вихід *D*2 на деякий час може стати метастабільним. Якщо період тактового сигналу досить великий, то з високою ймовірністю до кінця періоду *D*2 дійде коректного логічного рівня. Тригер *F*2 потім фіксує *D*2, який тепер стабільний і формує коректний вихідний сигнал.

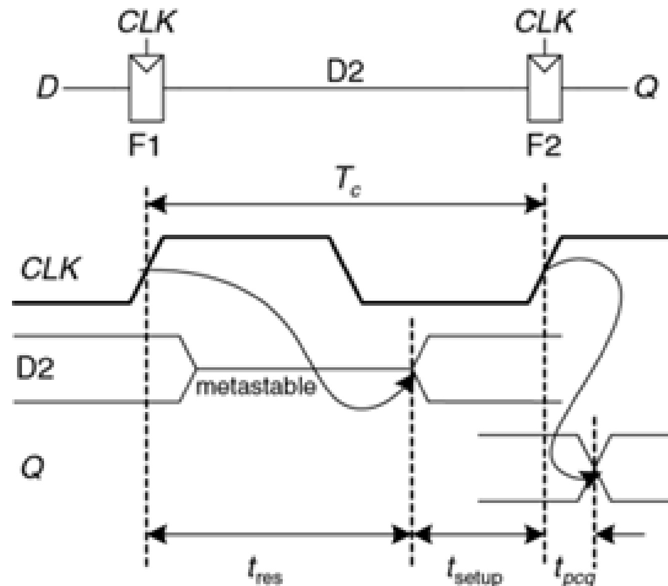


Рисунок 3.52 – Простий синхронізатор

Ми говоримо про збій синхронізатора, якщо його вихід *Q* стане метастабільним. Це може статися, якщо *D*2 не встигне прийти до коректного стану до початку часу встановлення тригера *F*2, тобто коли $t_{RES} > Tc - t_{SETUP}$. Відповідно до виразу (3.1) ймовірність збою для одиночної зміни входу в довільний час дорівнює:

$$\mathrm{P(failture)} = \frac{\mathrm{T_0}}{\mathrm{T}_c} e^{\frac{t}{\tau}}\ ,\ \mathrm{t} = T_0 - t_{\mathrm{setup}}. \tag{3.25}$$

Імовірність збою, *P(failure)*, є ймовірністю того, що вихід *Q* буде метастабільний після одноразової зміни входу *D*. Якщо *D* змінюється один раз

за секунду, то ймовірність збою за одну секунду буде просто *P(failure)*. Однак, якщо *D* змінюється *N* разів за секунду, то ймовірність помилки за секунду буде в *N* разів більшою:

$$P1(failture)= N*P(failture):sec. \quad (3.26)$$

Надійність системи зазвичай вимірюють середнім часом напрацювання на відмову (*mean time between failures, MTBF*). Як відомо з назви, *MTBF* – це середній час між відмовами системи. Ця величина обернена ймовірності збою системи за будь-яку задану секунду:

$$MTBF=1/ P1(failture). \quad (3.27)$$

Вираз (3.27) показує, що *MTBF* зростає експоненційно зі зростанням часу очікування синхронізатора, *Tc*. Для більшості систем синхронізатор, який очікує на один період тактового сигналу, забезпечує достатню величину *MTBF*. У високошвидкісних системах може знадобитися очікування триваліше, ніж періоди тактового сигналу.

Приклад 3.14.

Синхронізатор для входу кінцевого автомата

Кінцевий автомат, що керує роботою світлофора (див. п. 3.4.1) приймає асинхронні вхідні сигнали від датчиків дорожнього руху. Припустимо, що для забезпечення стабільності входів використовуються синхронізатори. У середньому за секунду датчик спрацьовує 0,2 раза. Тригер у синхронізаторі має такі властивості: τ = 200 пс, T_0 = 150 пс та t_{SETUP} = 500 пс. Яким має бути період синхронізатора, щоб середній час напрацювання на відмову (*MTBF*) перевищував 1 рік?

Виконання. 1 рік $\approx\pi * 10^7$ с.

$$\pi * 10^7 = \frac{A}{0.2*150*10^{-12}}, \; A = T_c*e^{м}, \; м = (T_c-500*10^{-12})\, 200^{-12}. \quad (3.28)$$

Для знаходження шуканого періоду необхідно розв’язати рівняння (3.27), яке має рішення в аналітичному вигляді. Однак його досить просто розв’язати методом спроб та помилок. В електронній таблиці можна спробувати декілька величин T_C і порахувати *MTBF*, доки не буде знайдена величина T_C, що дасть *MTBF*, близьке до 1 року: T_C = 3,036 нс.

3.5.6 Обчислення часу дозволу

Вираз (3.24) можна отримати, використовуючи базові знання курсів теорії ланцюгів, диференційних рівнянь і теорії ймовірностей. Цей розділ можна пропустити, якщо ви не цікавитеся виведенням цього виразу або якщо ви слабко ознайомлені з елементарною математикою.

Вихід тригера буде метастабільним за деякий час *t*, якщо тригер намагається зафіксувати вхід, який змінюється (що призводить до виникнення метастабільного стану), і вихід не встигає прийти до коректного рівня протягом цього часу після фронту тактового сигналу. Символічно це можна виразити так:

$$P(t_{RES} > t) = P(\text{samples changing input}) * P(\text{unresolved}). \qquad (3.29)$$

Обидва ймовірнісні помножувачі розглядатимуться окремо. Як показано на рис. 3.53, асинхронний вхідний сигнал переходить зі стану 0 в стан 1 протягом деякого часу t_{SWITCH}.

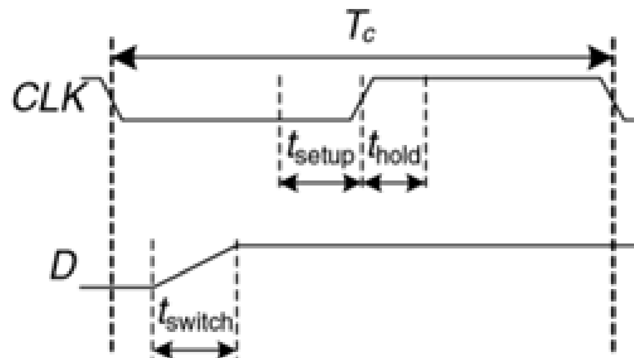


Рисунок 3.53 – Часова діаграма вхідного сигналу

На рис. 3.53 зображено, як асинхронний вхідний сигнал переходить зі стану 0 в стан 1 упродовж деякого часу t_{SWITCH}. Імовірність того, що вхід зміниться протягом апертурного часу, дорівнює

$$P(\text{samples changing input}) = (t_{SWITCH} + t_{SETUP} + t_{HOLD}):Tc\ . \qquad (3.30)$$

Якщо тригер вже перейшов у метастабільний стан з ймовірністю *P* (*samples changing input*), то час, необхідний для вирішення метастабільності, залежить від внутрішньої структури схеми. Цей час визначає ймовірність *P* (*unresolved*) – імовірність того, що тригер не встигає перейти до коректного стану (0 або 1) за час *t*. У цьому розділі буде проаналізовано просту модель бістабільного приладу та оцінено цю ймовірність [1].

Для побудови бістабільного приладу використовується пристрій з позитивним зворотним зв’язком. На рис. 3.54, *a* продемонстровано реалізацію такого зворотного зв’язку із застосуванням двох інверторів; поведінка такої схеми є репрезентативною для більшості бістабільних елементів. Пара інверторів поводиться аналогічно щодо буфера. Для побудови моделі можна вважати, що буфер має симетричну передатну властивість за постійним струмом (рис. 3.54, *b*), її нахил дорівнює *G*.

Вихідний струм буфера обмежений, цей факт можна промоделювати його вихідним опором *R*. Усі реальні схеми також мають деяку ємність *C*, яку необхідно перезаряджати в разі зміни стану схеми. Процес зарядки

конденсатора через резистор не дає змогу буферу перемикатися миттєво, властивий час цього процесу дорівнює *RC*. Отже, повну модель схеми подано на рис. 3.54, *c*, де $V_{OUT}(t)$ – напруга, що визначає стан бістабільної схеми.

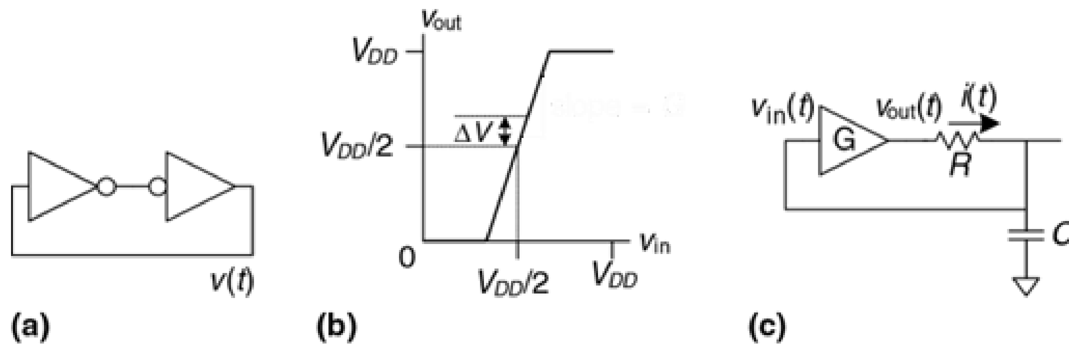


Рисунок 3.54 – Схемна модель бістабільного пристрою

Стан схеми, за умови якого $V_{OUT}(t) = V_{IN}(t) = V_{DD}/2$ є метастабільним; якщо схема стартує точно з цього стану, то за відсутності шуму вона перебуватиме в ньому невизначено тривалий час. Оскільки вся напруга є безперервними величинами, то ймовірність того, що робота схеми почнеться точно в точці метастабільності, зникаюче мала. Однак робота схеми може розпочатися в нульовий момент часу біля точки метастабільності, коли $out(0) = V_{DD}/2 + V$, де V – незначне відхилення. У такому разі позитивний зворотний зв'язок зрештою приведе $V_{OUT}(t)$ до Δ V_{DD}, якщо $\Delta V > 0$, або до 0 за умови $\Delta V < 0$. Час, необхідний для досягнення V_{DD} або 0, є часом дозволу бістабільного приладу.

Передавальна властивість буфера по постійному струму не лінійна, але на околиці точки метастабільності вона має форму, близьку до лінійної. Точніше, якщо $V_{IN}(t) = V_{DD}/2 + \Delta V/G$, то $V_{OUT}(t) = V_{DD}/2 + \Delta V$, для малих ΔV. Струм через резистор дорівнює $i(t) = (V_{OUT}(t) - V_{IN}(t))/R$. Конденсатор заряджається зі швидкістю $dvin(t)/dt = i(t)/C$. Поєднуючи ці два вирази, можна знайти рівняння для вихідної напруги:

$$duout(t)/dt = (G - 1) /RC(duout - VDD/2). \quad (3.31)$$

Це лінійне диференційне рівняння першого порядку. Розв'язуючи його з початковою умовою $V_{OUT}(0) = V_{DD}/2 + \Delta V$ можна знайти залежність вихідної напруги від часу:

$$Vout = V_{DD}/2 + \Delta V\exp(-(G - 1)t/RC) . \quad (3.32)$$

На рис. 3.55 подано графіки $V_{OUT}(t)$ для різних початкових точок. Напруга V_{OUT} експоненційно віддаляється від метастабільної точки $V_{DD}/2$, поки не досягне межі V_{DD} або 0. Вихід зрештою приходить у коректний логічний стан 0 або 1. Час, необхідний для цього, залежить від відхилення початкової напруги (ΔV) від точки метастабільності ($V_{DD}/2$).

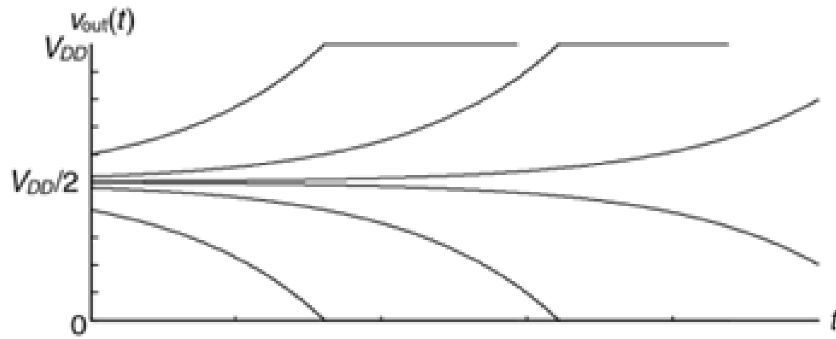


Рисунок 3.55 – Часова діаграма процесу переходу до коректного стану

Якщо в рівняння (3.32) підставити $V_{OUT}(t_{RES}) = V_{DD}$ чи 0, можна знайти час вирішення t_{RES}:

$$|\Delta V|\exp(G-1)\ t_{RES}/(RC)=V_{DD}/2; \qquad (2.33)$$

$$t_{RES} = (RC/(G-1))\ \ln\ (V_{DD}/(2*\Delta V)). \qquad (2.34)$$

Отже, час дозволу експоненційно зростає, якщо бістабільний пристрій має великий опір або ємність, що не дають змогу вихідній напрузі швидко змінюватися. Він зменшується, якщо бістабільний пристрій має значне посилення *G*. Час дозволу також логарифмічно зростає за умови наближення початкових умов схеми до точки метастабільності ($\Delta V \rightarrow 0$).

Позначимо τ як $RC/(G-1)$. З рівняння (3.34) можна отримати значення початкового відхилення, що відповідає деякому заданому часу дозволу t_{RES}:

$$\Delta V_{RES} = V_{DD}\ *\exp(-t_{RES}\ /\tau)/2\ . \qquad (3.35)$$

Припустимо, що бістабільний пристрій намагається зафіксувати вхідний сигнал під час його зміни. На його вхід надходить напруга $V_{IN}(0)$, яка передбачається рівномірно розподіленою в інтервалі від 0 до V_{DD}. Імовірність того, що вихід не досягне коректного значення за три години, залежить від імовірності того, що початкове відхилення буде досить незначним. Більш точно, початкове відхилення V_{OUT} має бути меншим, ніж $\Delta\ V_{RES}/G$. Тоді ймовірність того, що вхідний сигнал бістабільного пристрою має досить незначне відхилення, дорівнює

$$P(\text{unresolved}) = P(\ |\ V_{IN}(0) - V_{DD}/2| < \Delta V_{RES}\ /G\ ) = 2\Delta V_{RES}\ (G*V_{DD}). \qquad (3.36)$$

Отже, ймовірність того, що час дозволу перевищує задану величину *t,* задається таким виразом:

$$P(t_{RES} > t) =(\ t_{SWITCH}+\ t_{SETUP} + t_{HOLD})/(GTc)\ \exp(-t\ /\tau)\ . \qquad (3.37)$$

Зверніть увагу, що вирази (3.37) і (3.24) мають однаковий вигляд, якщо $T0 = (t_{SWITCH} + t_{SETUP} + t_{HOLD})/G$ і $\tau = RC/(G-1)$. Отже, ми вивели вираз (3.24) і показали, як величини $T0$ та τ залежать від фізичних властивостей бістабільного пристрою.

3.6 Паралелізм

Швидкість оброблення інформації системою визначається затримкою та пропускною спроможністю передачі крізь неї інформації. Визначимо токен як групу входів, яка обробляється для того, щоб отримати групу виходів. Ця назва пов'язана з методом візуалізації передачі інформації всередині системи способом розміщення в схемі токенів або маркерів та їх пересування за схемою разом із даними, що обробляються. Затримка, або латентність (*latency*), системи – час, який необхідний для проходження одного токена крізь усю систему з її входу на вихід. Пропускна спроможність (*throughput*) – кількість токенів, що може бути оброблена системою за одиницю часу.

Приклад 3.15.

Пропускна здатність і затримка під час приготування печива

Бену потрібно швидко підготуватися до вечірки з молоком та печивом, присвяченої введенню в експлуатацію його світлофора. За 5 хв він згортає печиво та укладає його на металевий лист. Протягом 15 хв печиво випікають у печі. Після цього Бен починає готувати наступний лист. Яка пропускна здатність та затримка випікання деко печива?

Виконання. У цьому прикладі деко є токеном. Затримка дорівнює 1/3 год на лист. Пропускна здатність – три листи за годину.

Досить легко зрозуміти, що пропускна здатність може бути збільшена завдяки обробленю кількох токенів в один і той самий час. Це називається паралелізмом і використовується у двох формах: просторовій та часовій. У просторовому паралелізмі застосовується кілька копій апаратних блоків, так що в один і той самий час можна виконувати декілька завдань. Часовий паралелізм передбачає розбиття завдання на кілька стадій (або ступенів), як це відбувається на складальному конвеєрі. Декілька завдань можуть бути розподілені по сходах. Хоча всі завдання мають пройти по всіх щаблях, різні завдання в будь-який заданий момент часу перебуватимуть на своєму ступені, так що кілька завдань можуть одночасно оброблятися на різних щаблях. Часовий паралелізм часто називається конвеєризацією. Просторовий паралелізм часто називають просто паралелізмом, але ми уникатимемо цієї назви через її неоднозначність.

Приклад 3.16.

Паралелізм під час приготування печива

До Бена Бітділа на вечірку прийдуть сотні друзів, і йому потрібно пекти печиво швидше. Він збирається використовувати просторовий та/або часовий паралелізм.

Просторовий паралелізм: Бен просить Алісу Хакер допомогти йому. Дівчина має власну піч і деко.

Часовий паралелізм: Бену дали другий лист. Як тільки чоловік ставить одне деко в піч, починає згортати печиво на іншому листі, а не чекає завершення випікання печива на першому деко.

Якою буде затримка та пропускна здатність у використанні просторового паралелізму? Часового? За умови застосування обох видів паралелізму?

Виконання. Затримка – це час, необхідний для завершення одного завдання від початку до кінця. У всіх випадках затримка дорівнює 1/3 год. Якщо на початку Бен не мав печива, то затримка – це час, необхідний для виробництва першого листа.

Пропускна здатність – це кількість деко з печивом, яке виробляється за одну годину. У разі застосування просторового паралелізму і Бен, і Аліса роблять по одному деко кожні 20 хв. Отже, пропускна здатність подвоюється і становить шість листів за годину. У процесі використання часового паралелізму Бен ставить нове деко в піч кожні 15 хв, пропускна здатність дорівнює чотирьом деко за годину (див. рис. 3.56).

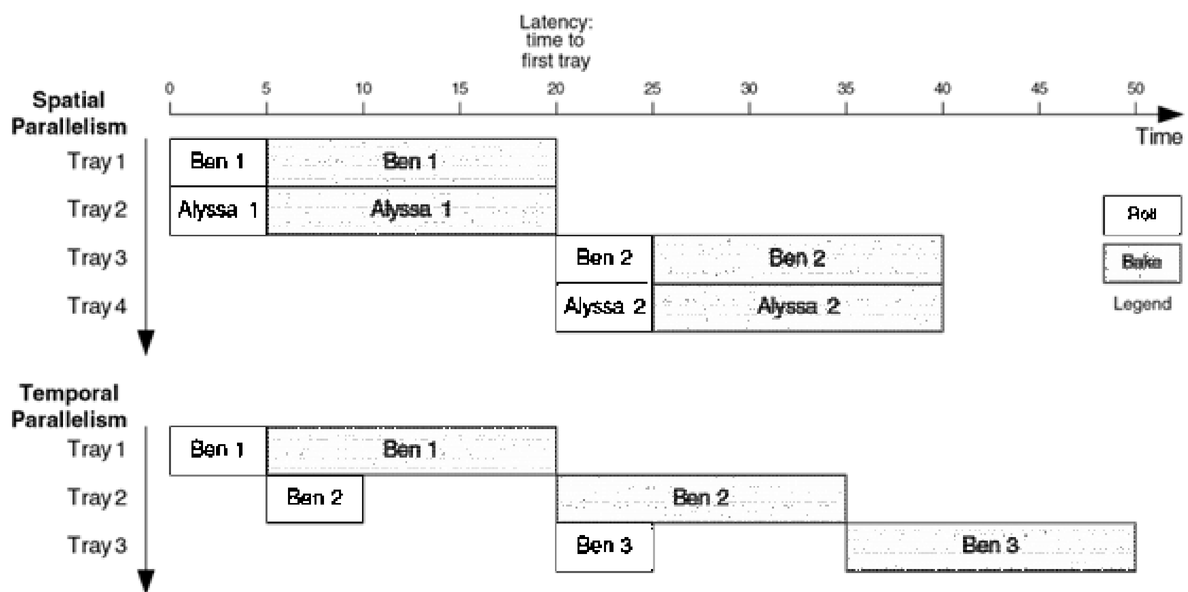


Рисунок 3.56 – Просторовий і часовий паралелізм у процесі приготування печива

Якщо Бен і Аліса застосовують два методи, вони можуть випекти вісім деко за годину.

Розглянемо систему із затримкою *L*. Якщо в системі відсутній паралелізм, то пропускна здатність становитиме $1/L$. У системі з просторовим паралелізмом, що містить *N* копій апаратних блоків, пропускна здатність буде N/L. У системі з часовим паралелізмом завдання в ідеальному випадку розбивається на *N* стадій або ступенів однакової довжини. У цьому разі пропускна спроможність також дорівнює N/L, до того ж необхідний тільки один екземпляр апаратного блоку. Однак, як показує приклад приготування печива, часто створення *N* ступенів однієї і тієї самої тривалості оброблення неможливе. Якщо найдовший ступінь має затримку $L1$, то пропускна здатність конвеєризованої системи дорівнюватиме $1/L1$.

Конвеєризація (часовий паралелізм) є особливо привабливою, оскільки збільшує швидкість схеми без збільшення апаратних витрат. Натомість регістри, встановлені між блоками комбінаційної логіки, поділяють її на короткі щаблі, які можуть працювати на вищій тактовій частоті. Регістри не дають змогу токенам, розташованим в одному ступені, наздоганяти та руйнувати токени, що перебувають у наступній стадії оброблення.

На рис. 3.57 наведено приклад схеми, де відсутня конвеєризація. Схема містить чотири блоки логіки, розташовані між двома регістрами. Критичний шлях проходить крізь блоки 2, 3 та 4. Припустимо, що регістр має затримку поширення на тактовий вхід-вихід 0,3 нс та час утримання 0,2 нс. Тоді мінімальний період тактового сигналу дорівнює $Tc = 0,3 + 3 + 2 + 4 + 0,2 = 9,5$ нс. Схема має затримку 9,5 нс та пропускну здатність 1/9,5 нс = 105 МГц.

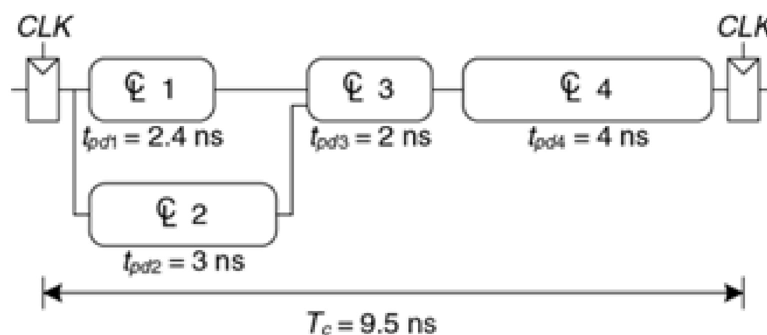


Рисунок 3.57 – Схема без конвеєризації

На рис. 3.58 зображена ця сама схема, розділена за допомогою додаткових регістрів між блоками 3 і 4, на двоступінчастий конвеєр. Перший ступінь має мінімальний період тактового сигналу 0,3 + 3 + 2 + 0,2 = 5,5 нс. Мінімальний період для другого ступеня дорівнює 0,3 + 4 + 0,2 = 4,5 нс. Тактовий сигнал має бути досить повільним для того, щоб працювали

всі щаблі. Отже, Тс = 5,5 нс. Затримка дорівнює двом періодам тактового сигналу чи 11 нс. Пропускна здатність дорівнює 1/5,5 ns = 182 МГц.

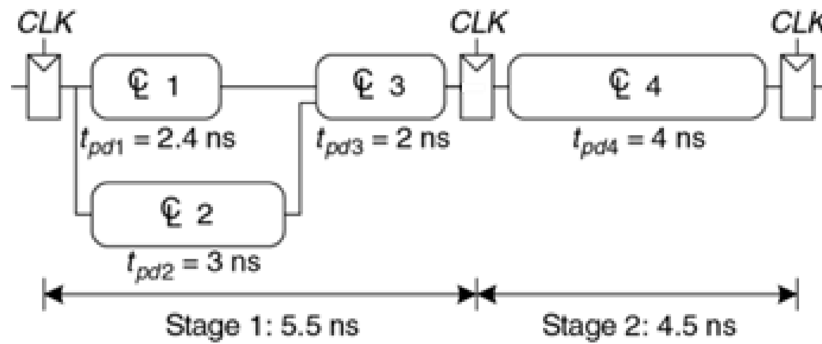


Рисунок 3.58 – Схема з двоступінчастим конвеєром

Цей приклад демонструє, що в реальних схемах конвеєризація двома ступенями майже подвоює пропускну здатність і трохи збільшує затримку. Для порівняння ідеальна конвеєризація точно подвоїла б пропускну здатність і не погіршила б затримку. Невідповідність виникає тому, що реальну схему неможливо розділити на дві абсолютно рівні частини і тому, що конвеєрні регістри вносять додаткові втрати на впорядкування.

На рис. 3.59 зображено ще один варіант тієї самої схеми, в якому використовується триступеневий конвеєр. Зауважте, що в схемі має бути на два регістри більше, вони зберігають результати блоків 1 і 2 в кінці першого ступеня конвеєра. Час циклу обмежується тепер третім ступенем і дорівнює 4,5 нс. Затримка дорівнює трьом циклам або 13,5 нс. Пропускна спроможність дорівнює 1/4,5 нс = 222 МГц. Як і в минулому варіанті схеми, додавання ще одного ступеня конвеєра покращує пропускну здатність завдяки незначному збільшенню затримки.

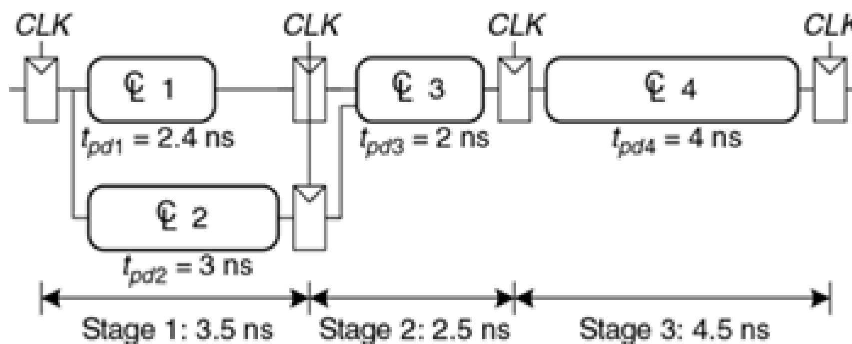


Рисунок 3.59 – Схема з триступеневим конвеєром

Хоча ці розглянуті підходи дуже ефективні, вони не можуть бути використані в усіх ситуаціях. Застосування паралелізму обмежується взаємозалежностями (*dependencies*) реальних завдань. Якщо поточне завдання

залежить від результатів попереднього, а не лише від своїх попередніх кроків, його виконання не може бути розпочато до завершення попереднього завдання. Наприклад, якщо Бен Бітдідл хоче перевірити, чи досить смачні печива з першого листа, перед приготуванням другого він має взаємозалежність, що перешкоджає використанню конвеєра або паралелізму. Паралелізм – один із найважливіших методів проєктування високопродуктивних цифрових систем. Конвеєризація далі обговорюватиметься в розділі 7, де будуть наведені приклади оброблення взаємозалежностей.

3.7 Резюме

Цей розділ було присвячено методам аналізу та проєктування послідовної логіки. На відміну від комбінаційної логіки, вихідні сигнали якої залежать тільки від поточного стану вхідних сигналів, вихідні сигнали послідовної логіки залежать як від поточного, так і від попереднього стану вхідних сигналів. Іншими словами, послідовна логіка пам’ятає інформацію про вхідні сигнали в попередні моменти часу. Ця пам’ять називається станом логіки.

Послідовні схеми можуть бути складними для аналізу, і їх легко неправильно спроєктувати, тому обмежимося використанням незначної кількості ретельно спроєктованих апаратних блоків. Найбільш важливим елементом для наших цілей є тригер, що приймає тактовий сигнал та вхідний сигнал D та формує вихідний сигнал Q. По передньому фронту тактового імпульсу тригер копіює вхід D на вихід Q, інакше він зберігає старий стан Q. Група тригерів із загальним тактовим сигналом називається регістром. На тригери можуть надходити сигнали скидання чи дозволу.

Хоча існує безліч форм послідовних схем, ми обмежимося використанням синхронних послідовних схем, оскільки їх просто розробляти. Синхронні послідовні схеми містять блоки комбінаційної логіки, розділені регістрами, що тактуються. Стан схеми зберігається в регістрах і оновлюється лише за фронтами тактового сигналу.

Один з ефективних підходів до проєктування послідовних схем ґрунтується на використанні кінцевих автоматів. Для проєктування кінцевого автомата спочатку необхідно визначити його входи та виходи, потім зробити ескіз діаграми переходів із зазначенням станів і умов переходів між ними. Потім для всіх станів автомата потрібно обрати кодування й на основі діаграми створити таблицю переходів між станами та таблицю виходів, які показують наступний стан та вихідний сигнал за умови заданого поточного

стану та вхідного сигналу. За цими таблицями проєктується комбінаційна логічна схема, що визначає наступний стан та вихідний сигнал, і створюється ескіз схеми.

Синхронні послідовні схеми мають часову специфікацію, що передбачає затримки поширення та реакції тракту тактовий вхід-вихід, t_{PCQ} і t_{CCQ}, а також час передустановлення та утримання, t_{SETUP} і t_{HOLD}. Для коректної роботи таких схем їх входи мають бути стабільними протягом апертурного часу, що передбачає час передустановлення перед переднім фронтом тактового імпульсу та часу утримання після нього. Мінімальний період *Tc* тактового сигналу системи дорівнює сумі затримок поширення комбінаційної логіки, t_{PD}, і затримок $t_{PCQ} + t_{SETUP}$ у регістрах. Для коректної роботи схеми затримка реакції регістрів та комбінаційної логіки має бути більшою, ніж t_{HOLD}. Незважаючи на поширену оману, час утримання не впливає на величину мінімального тактового періоду сигналу [1].

Загальна продуктивність системи вимірюється затримкою та пропускною здатністю. Затримка – це час, необхідний для проходження одного токена з входу системи на її вихід. Пропускна здатність – кількість токенів, які система може обробити за одиницю часу. Паралелізм підвищує пропускну здатність системи.

ВПРАВИ

Вправа 3.1. Часові діаграми вхідних сигналів *RS*-тригера-засувки зображені на рис. 3.60. Накресліть часову діаграму значень *Q*.

Рисунок 3.60 – Часові діаграми входів *RS*-тригера-засувки для вправи 3.1

Вправа 3.2. Часові діаграми вхідних сигналів *RS*-тригера-засувки подані на рис. 3.61. Накресліть часову діаграму значень *Q*.

Рисунок 3.61 – Часові діаграми входів *RS*-тригера-засувки для вправи 3.2

Вправа 3.3. Часові діаграми вхідних сигналів D-тригера-засувки зображені на рис. 3.62. Накресліть часову діаграму значень Q.

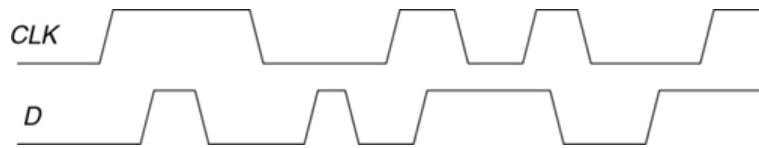


Рисунок 3.62 – Часові діаграми входів D-тригера для вправ 3.3 та 3.5

Вправа 3.4. Часові діаграми вхідних сигналів D-тригера-засувки зображені на рис. 3.63. Накресліть часову діаграму значень Q.

Рисунок 3.63 – Часові діаграми входів D-тригера для вправ 3.4 та 3.6

Вправа 3.5. На рис. 3.62 подані часові діаграми входів D-тригера (синхронізованого фронтом). Накресліть часову діаграму значень Q.

Вправа 3.6. На рис. 3.63 подані часові діаграми входів D-тригера (синхронізованого фронтом). Накресліть часову діаграму значень Q.

Вправа 3.7. Чи є схема, яку зображено на рис. 3.64, комбінаційною чи послідовною? Поясніть взаємозв’язок входів із виходами. Як називається така схема?

Рисунок 3.64 – Досліджувана схема

Вправа 3.8. Чи є схема, яку зображено на рис. 3.65, комбінаційною чи послідовною? Поясніть взаємозв’язок входів із виходами. Як називається така схема?

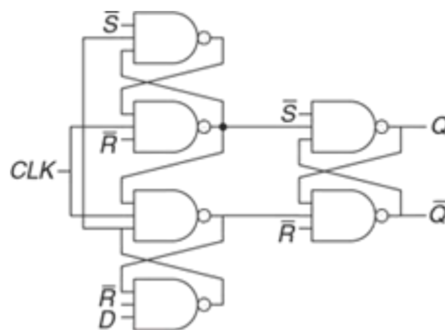


Рисунок 3.65 – Досліджувана схема

Вправа 3.9. *T*-тригер (від англ. *toggle* – перемикати) має один вхід *CLK* і один вихід *Q*. По кожному фронту тактового сигналу значення на виході тригера змінюється на протилежне. Накресліть схему *T*-тригера, використовуючи *D*-тригер та інвертор.

Вправа 3.10. На вхід *JK*-тригера надходять тактовий сигнал *CLK* і вхідні дані *J* та *K*. Тригер синхронізується по фронту тактового сигналу. Якщо *J* і *K* дорівнюють нулю, то на виході *Q* зберігається старе значення. Якщо $J = 1$, $K = 0$, то *Q* встановлюється 1. Якщо $J = 0$, $K = 1$, *Q* скидається в 0. Якщо $J = 1$, $K= 1$, *Q* приймає протилежне значення.

a) Побудуйте *JK*-тригер, використовуючи *D*-тригер і комбінаційну логіку.

b) Побудуйте *D*-тригер, використовуючи *JK*-тригер і комбінаційну логіку.

c) Побудуйте *T*-тригер (див. вправу 3.9), застосовуючи *JK*-тригер.

Вправа 3.11. Схема, зображена на рис. 3.66, називається *C*-елементом Мюллера. Поясніть взаємозв’язок входів із виходами.

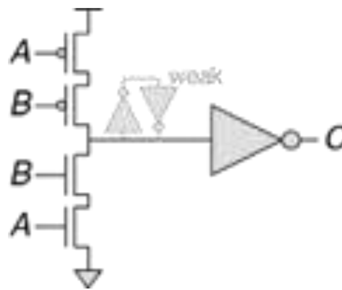


Рисунок 3.66 – *C*-елемент Мюллера

Вправа 3.12. Спроєктуйте *D*-клапан з асинхронним скиданням, використовуючи логічні елементи.

Вправа 3.13. Спроєктуйте *D*-тригер з асинхронним скиданням за допомогою логічних елементів.

Вправа 3.14. Спроєктуйте *D*-тригер, який синхронно встановлюється, застосовуючи логічні елементи.

Вправа 3.15. Спроєктуйте *D*-тригер, що асинхронно встановлюється, використовуючи логічні елементи.

Вправа 3.16. Кільцевий генератор містить *N* інверторів, замкнутих у кільце. Кожен інвертор має мінімальну t_{CD} і максимальну t_{PD} затримку. Визначте діапазон частот, у якому може працювати кільцевий генератор за умови, що *N* не парне.

Вправа 3.17. Чому число *N* із вправи 3.16 має бути непарним?

Вправа 3.18. Які із схем на рис. 3.67 є синхронними послідовними? Обґрунтуйте відповідь.

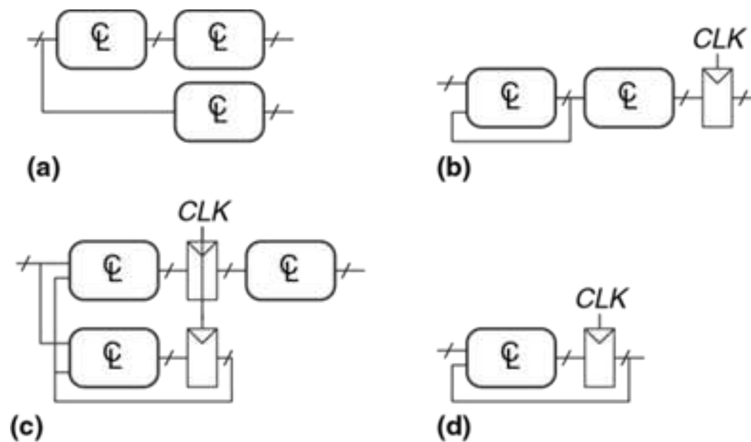


Рисунок 3.67 – Схеми

Вправа 3.19 Ви проєктуєте контролер ліфта для 25-поверхової будівлі. У контролера є два входи: ВГОРУ і ВНИЗ. Вихідними даними є номер поверху, де стоїть ліфт. 13-й поверх відсутній. Чому дорівнює мінімальна кількість бітів для зберігання стану в контролері?

Вправа 3.20. Ви проєктуєте кінцевий автомат для відстеження настрою чотирьох студентів, які працюють у лабораторії з проєктування цифрових схем. У студентів можуть бути такі настрої: ЩАСЛИВИЙ (якщо схема працює), СУМНИЙ (якщо схема згоріла), ЗАНЯТИЙ (працює над схемою), ЗАВАНТАЖЕНИЙ (думає над схемою), ДРІМЛИВИЙ (спить на робочому місці). Скільки станів буде у вашого автомата? Яка мінімальна кількість бітів станів потрібна для кодування стану автомата?

Вправа 3.21. Як би ви поділили кінцевий автомат із вправи 3.20 на кілька менш складних автоматів? Скільки станів було б у кожного такого простого автомата? Яка мінімальна кількість бітів необхідна для такого модульного проєкту?

Вправа 3.22. Опишіть словами, що робить автомат на рис. 3.68. Заповніть таблицю переходів і таблицю виходів за допомогою двійкового кодування. Складіть булеві вирази для наступного стану й виходу та накресліть схему цього кінцевого автомата.

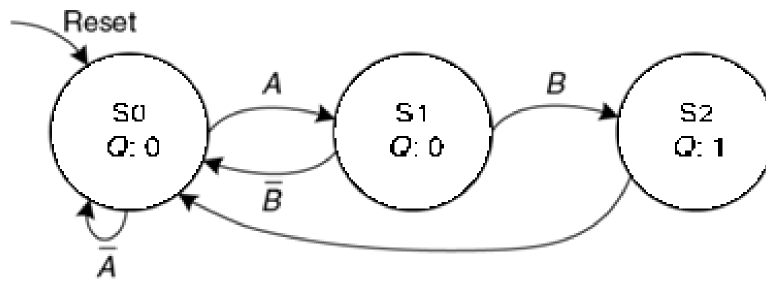


Рисунок 3.68 – Діаграма переходів

Вправа 3.23. Опишіть словами, що робить автомат на рис. 3.69. Заповніть таблицю переходів і таблицю виходів, використовуючи двійкове кодування. Складіть булеві вирази для наступного стану й виходу та накресліть схему цього кінцевого автомата.

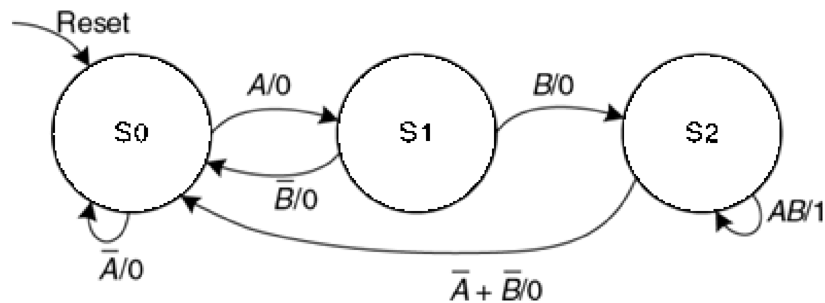


Рисунок 3.69 – Діаграма переходів

Вправа 3.24. На перетині Академічної та Бігової вулиць усе ще трапляються події. Футболісти прямують на перехрестя, як тільки на їх світлофорі спалахує зелене світло, і стикаються з ботаніками, що втратили пильність. Останні вступають на перехрестя ще на зелене світло. Удосконаліть світлофор з п. 3.4.1 так, щоб на двох вулицях горіло червоне світло протягом 5 с до того, як якийсь із світлофорів стане зеленим. Накресліть діаграму переходів автомата Мура, кодування станів, таблицю переходів, таблицю виходів, вирази для виходів та наступного стану й схему кінцевого автомата.

Вправа 3.25. У равлика Аліси з п. 3.4.3 є дочка з автоматом Мілі. Равлик-дочка усміхається, коли вона проходить послідовність 1101 або 1110. Накресліть діаграму переходів для цього веселого равлика, використовуючи якнайменше станів. Оберіть кодування станів і складіть загальну таблицю переходів та виходів. Складіть вирази для виходу й наступного стану, накресліть схему автомата.

Вправа 3.26. Вас умовили спроєктувати автомат із прохолодними напоями для офісу. Витрати на напої частково покриває профспілка, тому вони коштують лише по 5 грн. Автомат приймає монети в 1, 2 та 5 грн.

Як тільки покупець внесе необхідну суму, автомат видасть напій і поверне решту. Спроєктуйте кінцевий автомат для автомата з прохолодними напоями. Входами автомата є монети 1, 2 та 5 грн. Припустимо, що в кожному тактовому сигналі вставляється лише одна монета. Автомат має виходи: налити газовану воду, повернути 1 грн, повернути 2 грн, повернути дві по 2 грн. Як тільки в автоматі набирається 5 грн (або більше), він подає сигнал «Налити напій», а також сигнали повернення відповідної решти. Потім автомат має бути готовий знову приймати монети.

Вправа 3.27. Код Грея має корисну властивість: коди сусідніх чисел розрізняються один від одного тільки в одному розряді. У табл. 3.23 подано трирозрядний код Грея, що є числовою послідовністю від 0 до 7. Спроєктуйте трирозрядний автомат лічильника в коді Грея за модулем 8. В автомата немає входів, але є три виходи. (Лічильник за модулем *N* працює від 0 до *N*-1, потім цикл повторюється. Наприклад, у годиннику використовується лічильник по модулю 60 для того, щоб рахувати хвилини і секунди від 0 до 59.) Після скидання на лічильнику має бути 000. За кожним фронтом тактового сигналу лічильник повинен переходити до наступного коду Грея. Після досягнення коду 100 лічильник має знову перейти до коду 000.

Таблиця 3.23 – Трирозрядний код Грея

Number	**Gray code**		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Вправа 3.28. Удосконаліть свій автомат лічильника в коді Грея з вправи 3.27 так, щоб він міг рахувати як вгору, так і вниз. У лічильника з'явиться вхід ВГОРУ. Якщо ВГОРУ = 1, то лічильник переходитиме до наступного коду, а якщо ВГОРУ = 0 – тоді до попереднього.

Вправа 3.2. Ваша компанія «Детекторама» хоче спроєктувати кінцевий автомат із двома входами *A* і *B* і одним виходом *Z*. Вихід у *n*-му циклі (*Zn*) і є результатом або логічного І, або логічного АБО поточного *An* і попереднього *An*-1 значень на вході залежно від сигналу *Bn*.

$$Zn = AnAn\text{-}1, \text{ якщо } Bn = 0;$$
$$Zn = An + An\text{-}1, \text{ якщо } Bn = 1.$$

a) Накресліть часову діаграму *Z* за показниками діаграм *A* і *B*, які зображено на рис. 3.70

b) Цей автомат є автоматом Мура чи автоматом Мілі?

c) Спроєктуйте кінцевий автомат. Складіть діаграму переходів, закодовану таблицю переходів, вирази для виходів та наступного стану й накресліть схему.

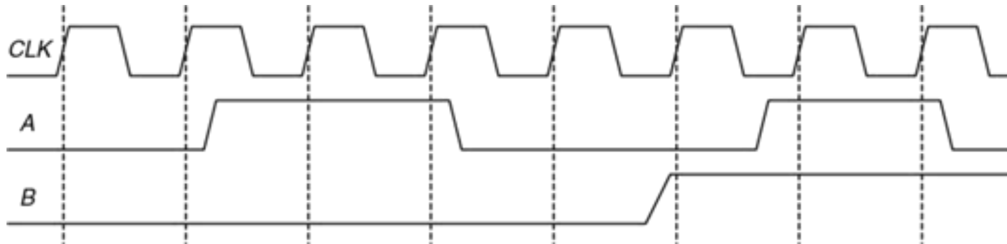


Рисунок 3.70 – Вхідні часові діаграми кінцевого автомата

Вправа 3.30. Спроєктуйте кінцевий автомат з одним входом *A* й двома виходами *X* та *Y*. На виході *X* має з’явитися 1, якщо 1 надходили на вхід як мінімум три цикли (необов’язково поспіль), а на *Y* має з’явитися 1, якщо *X* = 1 як мінімум два цикли поспіль. Складіть діаграму переходів, закодовану таблицю переходів, вирази для виходів та наступного стану й накресліть схему.

Вправа 3.31. Проаналізуйте кінцевий автомат на рис. 3.71. Складіть таблицю переходів та виходів, а також діаграму станів. Опишіть словами, що робить цей автомат.

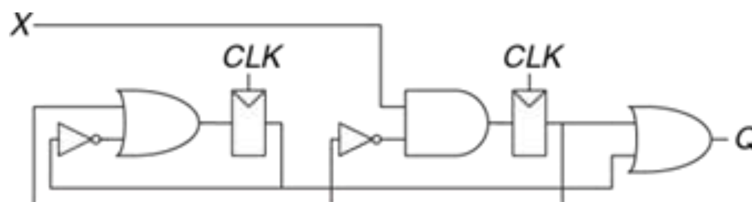


Рисунок 3.71 – Схема кінцевого автомата

Вправа 3.32. Повторіть вправу 3.31 із схемою, зображеною на рис. 3.72. Нагадаємо, що входи регістрів *s* і *r* відповідають за встановлення (*set*) та скидання (*reset*) відповідно.

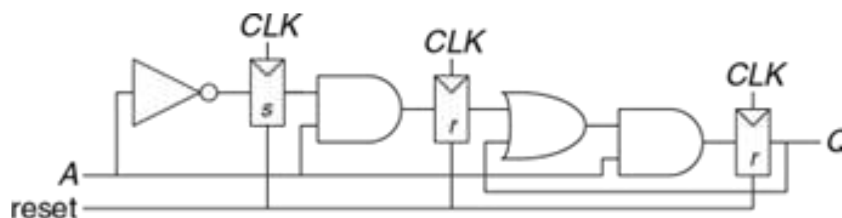


Рисунок 3.72 – Схема автомата

Вправа 3.33. Бен Бітдідл спроєктував схему обчислення функції *XOR* із чотирма входами й регістрами на вході та виході (див. рис. 3.73). Кожен двовходовий елемент *XOR* має затримку розповсюдження 100 пс та затримку реакції 55 пс. Час встановлення тригерів дорівнює 60 пс, час утримання становить 20 пс, максимальна затримка тактовий сигнал-вихід – 70 пс, мінімальна затримка – 50 пс.

a) Чому дорівнюватиме максимальна робоча частота схеми за відсутності розфазування тактових імпульсів?

b) Яке розфазування тактових імпульсів допустиме, якщо схема має працювати на частоті 2 ГГц?

c) Яке розфазування тактових імпульсів допустиме до виникнення у схемі порушень обмежень часу утримання?

d) Аліса Хакер стверджує, що вона може перепроєктувати комбінаційну логічну схему з метою підвищення її швидкості та стійкості до розфазування тактових імпульсів. У її покращеній схемі також використовується три двовходові елементи *XOR*, але вони по-іншому з'єднані між собою. Яку схему вона спроєктувала? Якою буде максимальна частота без розфазування тактових імпульсів? Яке розфазування тактових імпульсів допустиме до виникнення порушень обмежень часу утримання?

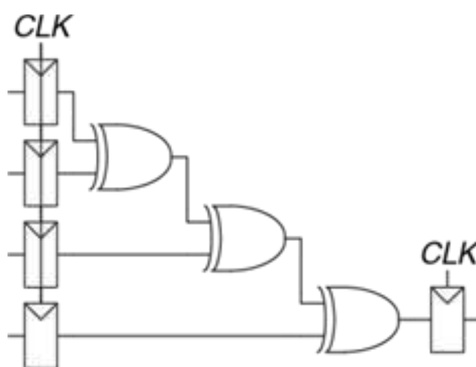


Рисунок 3.73 – Схема обчислення функції *XOR* з регістрами на вході та виході

Вправа 3.34. У межах розроблення швидкого дворозрядного процесора *RePentium* вам доручено проєктування суматора. Як зображено на рис. 3.74, суматор містить два повна суматори, вихід перенесення першого суматора приєднаний до входу другого перенесення. На вході та виході суматора розташовані регістри, суматор має виконати підсумовування за період тактового сигналу. Затримки поширення повних суматорів дорівнює: трактом вхід *Cin* – виходи *Cout* і *Sum* (*S*) – 20 пс, трактом входи *A* і *B* – вихід *Cout* – 25 пс, трактом входи *A* і *B* – вихід *S* – 30 пс. Повні суматори мають затримки реакції: трактом вхід Cin – будь-який вихід – 15 пс, трактом входи *A* і *B* –

будь-який вихід – 22 пс. Час предустановлення тригерів дорівнює 30 пс, утримання – 10 пс, час затримки тракту тактовий сигнал – вихід: поширення 10 пс, реакції 21 пс.

a) Чому дорівнюватиме максимальна робоча частота схеми за відсутності розфазування тактових імпульсів?

b) Яке розфазування тактових імпульсів допустиме, якщо схема має працювати на частоті 8 ГГц?

c) Яке розфазування тактових імпульсів допустиме до виникнення в схемі порушень обмежень часу утримання?

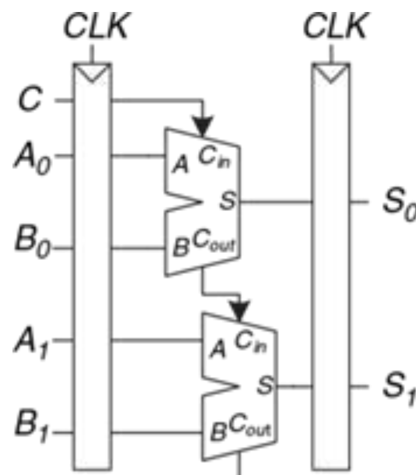


Рисунок 3.74 – Схема дворозрядного суматора

Вправа 3.35. У логічних матрицях, що програмуються користувачем, (*field programmable gate array, FPGA*) для створення комбінаційних логічних схем використовуються конфігуровані логічні блоки (*configurable logic blocks, CLBs*), а не логічні елементи.

У матрицях *Xilinx Spartan* три затримки поширення та реакції кожного *CLB* становитиме 0,61 та 0,30 нс відповідно. Вони також містять тригери, затримки поширення та реакції яких дорівнюють 0,72 та 0,50 нс, а час передустановлення та утримання – 0,53 та 0 нс відповідно.

a) Ви проєктуєте систему, яка має працювати на частоті 40 МГц. Скільки послідовно з’єднаних *CLB* можна розмістити між двома тригерами? У відповіді можна вважати, що розфазування тактових імпульсів та затримка у з’єднаннях між *CLB* відсутня.

b) Припустимо, що всі шляхи між тригерами проходять крізь принаймні один *CLB*. Яке розфазування тактових імпульсів допустиме до виникнення в схемі порушень обмежень часу утримання?

Вправа 3.36. Для побудови синхронізатора використовують два тригери з t_{SETUP} = 50 пс, $T0$ = 20 пс, і τ = 30 пс. Асинхронний вхід змінюється 108 разів за секунду. Чому дорівнює мінімальний період синхронізатора, за якого середній час між відмовами (*MTBF*) досягне 100 років?

Вправа 3.37. Вам необхідно побудувати синхронізатор, що приймає асинхронні вхідні сигнали, середній час між відмовами (*MTBF*) має бути не меншим ніж 50 років. Тактова частота системи дорівнює 1 ГГц, тригери мають такі параметри: τ = 100 пс, T0 = 110 пс і t_{SETUP} = 70 пс. На вхід синхронізатора кожні 2 с надходить новий асинхронний сигнал. Чому дорівнює ймовірність відмови, яка відповідає заданому середньому часу між відмовами (*MTBF*)? Скільки періодів тактового сигналу необхідно почекати перед зчитуванням зафіксованого вхідного сигналу досягнення цієї ймовірності?

Вправа 3.38. Ви зіткнулися зі своїм напарником з лабораторних робіт у коридорі, коли він йшов назустріч вам. Обидва ви відступили в один бік та ще досі рухаєтеся на шляху один одного. Потім ви обидва відступили в інший бік і продовжуєте заважати один одному пройти. Далі ви обидва вирішили трохи почекати, сподіваючись, що зустрічний відступить убік, і ви розійдетесь. Ви можете промоделювати цю ситуацію як метастабільну та застосувати до неї ту саму теорію, розвинену для синхронізаторів і тригерів.

Припустимо, ви створюєте математичну модель своєї поведінки та поведінки свого партнера. Стан, у якому ви заважаєте пройти один одному, можна тлумачити як метастабільний. Імовірність того, що ви залишаєтеся в цьому стані після t секунд дорівнює $exp(-t/\tau)$, величина τ описує швидкість вашої реакції. Сьогодні через недосипання ваш розум затуманений і τ = 20 с.

a) За який час із ймовірністю 99 % метастабільність буде дозволена (тобто ви зможете обійти один одного)?

b) Ви не тільки не виспалися, але й сильно зголодніли. Ситуація вкрай серйозна: ви помрете з голоду, якщо не потрапите до кав'ярні за 3 хв. Яка ймовірність того, що ваш партнер по лабораторних роботах залишиться один?

Вправа 3.39. Ви збудували синхронізатор із використанням тригерів з T0 = 20 пс і τ = 30 пс. Керівник доручив вам збільшити середній час між відмовами (*MTBF*) удесятеро. Наскільки потрібно збільшити період тактового сигналу?

Вправа 3.40. Бен Бітдідл винайшов новий покращений синхронізатор, що, як він вважає, пригнічує метастабільність за єдиний період. Схема

«поліпшеного синхронізатора» зображена на рис. 3.75. Бен пояснює, що схема в блоці *M* є аналоговим «детектором метастабільності», який видає сигнал високого логічного рівня, якщо напруга на його вході потрапляє в заборонену зону між V_{IL} і V_{IH}. Детектор метастабільності перевіряє, чи виник на виході *D*2 першого тригера метастабільний сигнал. Якщо він справді з'явився, то «детектор метастабільності» асинхронно скидає тригер і на його виході з'являється коректний логічний сигнал 0. Другий тригер фіксує сигнал *D*2 і з його виходу *Q* завжди буде коректний логічний рівень. Аліса Хакер каже Бену, що схема не працюватиме як заявлено, оскільки усунення метастабільності так само неможливе, як і побудова вічного двигуна.

Хто з них має рацію? Покажіть, де помилка Бена чи чому Аліса помиляється.

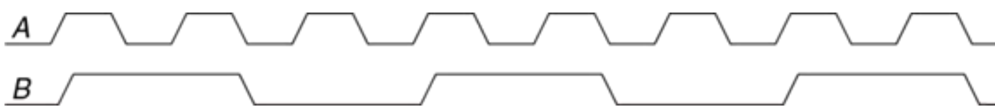


Рисунок 3.75 – «Новий покращений» синхронізатор

ЗАПИТАННЯ ДЛЯ СПІВБЕСІДИ

Подаємо приклади типових запитань,
які можуть бути поставлені претендентам
під час пошуку роботи
в галузі проєктування цифрових систем.

Запитання 3.1. Накресліть діаграму кінцевого автомата, що детектує надходження на вхід послідовності 01010.

Запитання 3.2. Спроєктуйте кінцевий автомат, що приймає послідовність бітів (один біт за раз) і виконує над ними операцію доповнення до 2. Він має два входи – *Start* і *A*, і один вихід *Q*. Двійкове число довільної довжини подається на вхід *A*, починаючи з молодшого розряду.

Відповідний вихідний біт з'являється на тому самому циклі на виході *Q*. Вхід *Start* установлюється на один цикл для ініціалізації кінцевого автомата перед надходженням молодшого біта.

Запитання 3.3. Чим розрізняються тригер-засувка і тригер, тактований фронтом імпульсу? Коли потрібно використовувати кожен з них?

Запитання 3.4. Спроєктуйте кінцевий автомат, що виконує функції п'ятирозрядного лічильника.

Запитання 3.5. Спроєктуйте схему детектування фронту сигналу. Її вихід має набувати значення 1 протягом одного періоду після переходу вхідного сигналу зі стану 0 в 1.

Запитання 3.6. Опишіть концепцію конвеєризації та методи її використання.

Запитання 3.7. Поясніть ситуацію, коли час утримання тригера негативний.

Запитання 3.8. Спроєктуйте схему, яка приймає сигнал *A* (див. рис. 3.76) і формує на виході сигнал *B*.

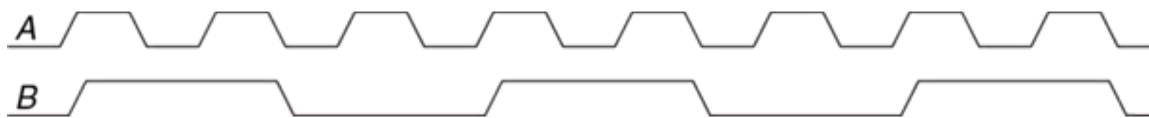


Рисунок 3.76 – Часові діаграми сигналів

Запитання 3.9. Розглянемо блок комбінаційної логіки між двома регістрами. Поясніть часові обмеження, що має задовольняти такий блок. Якщо поставити буфер на тактовому вході другого тригера, чи стануть обмеження часу попередньої установки більш пом’якшеними або, навпаки, жорсткішими?

4 МОВИ ОПИСУ АПАРАТУРИ

4.1 Вступ

Досі ми розглядали розроблення комбінаційних і послідовних цифрових схем на рівні схемотехніки. Процес пошуку найкращого набору логічних елементів для виконання цієї логічної функції трудомісткий і спричиняє помилки, оскільки вимагає спрощення логічних таблиць або виразів і перекладу кінцевих автоматів у вентилі вручну. Упродовж 1990-х років розробники виявили, що їхня продуктивність праці різко зростала, якщо вони працювали на більш високому рівні абстракції, визначаючи лише логічну функцію та надаючи створення оптимізованих логічних елементів системі автоматичного проектування (САПР). Дві основні мови опису апаратури (*Hardware Description Language, HDL*) – *SystemVerilog* та *VHDL*.

SystemVerilog і *VHDL* побудовані на схожих принципах, але їх синтаксис дуже різниться. Їх обговорення в цьому розділі поділено на два стовпці для порівняння, де *SystemVerilog* буде ліворуч, а *VHDL* – праворуч. У процесі першого читання зосередьтеся на одній мові. Як тільки розберетеся з однією, за необхідності ви зможете швидко засвоїти іншу. У наступних розділах показана апаратура й у схематичному вигляді та у формі *HDL*-моделі. Якщо ви не хочете читати цей розділ і вивчати мови опису цифрової апаратури, зможете зрозуміти принципи архітектури мікропроцесорів лише на рівні схем. Однак переважна більшість комерційних систем зараз будується з використанням мов опису цифрової апаратури, а не на рівні схемотехніки. Якщо ви коли-небудь у вашій кар'єрі збираєтеся розробляти цифрові схеми, рекомендуємо вам вивчити одну з мов опису апаратури.

4.1.1 Модулі

Блок цифрової апаратури, що має входи та виходи, називається модулем. Логічний елемент І, мультиплексор та схема пріоритетів є прикладами модулів цифрової апаратури. Є два загальноприйняті типи опису функціональності модуля – *поведінковий* та *структурний*. Поведінкова модель визначає, що модуль робить. Структурна модель визначає те, як побудовано модуль із найпростіших елементів, із застосуванням принципу ієрархії. Код *SystemVerilog* і *VHDL* з прикладу 4.1 показує поведінковий опис модуля, що розраховує булеву функцію з прикладу 2.6. Обома мовами модуль

називається *sillyfunction* і має три входи (*a, b* і *c*) і один вихід (*y*), і, як і очікували, дотримується принципу модульності. Він має цілком певний інтерфейс, що містить входи та виходи, і виконує певну функцію. Конкретний спосіб, яким модуль описаний, не важливий для тих, хто застосовуватиме модуль у майбутньому, оскільки модуль виконує свою функцію [1–7].

Приклад 4.1.
Комбінаційна логіка

SystemVerilog

```
module sillyfunction(input logic a, b, c, output logic y);
assign y = ~ a & ~ b & ~ c |
a & b & ~ c | a & b & c;
endmodule
```

Модуль *SystemVerilog* починається з імені модуля та списку входів і виходів. Оператор *assign* описує комбінаційну логіку. Тільда (~) означає НІ, амперсанд (&) – І, а вертикальна властивість (|) – АБО.

Сигнали типу *logic*, як входи та виходи в прикладі, – логічні змінні, що набувають значення 0 або 1. Вони також можуть набувати рухомого та невизначеного значення (це обговорюється в п. 4.2.8).

Тип логіки з’явився в *SystemVerilog*. Він введений для заміни типу *reg*, що був постійним джерелом труднощів *Verilog*. Тип *logic* варто використовувати скрізь, крім опису сигналів із кількома джерелами. Такі сигнали називаються ланцюгами (*net*) і пояснюються у підрозділі 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity sillyfunction is
 port(a, b, c: in STD_LOGIC; y: out STD_LOGIC);
end;
архітектура synth of sillyfunction is
 begin
   y <= (not a and not b and not c) or
   (a and not b and not c) or
   (a and not b and c);
end;
```

Код на *VHDL* містить три частини: оголошення використовуваних бібліотек і зовнішніх об’єктів (*library, use*), оголошення інтерфейсу об’єкта (*entity*) та його внутрішньої структури (*architecture*).

Конструкція для оголошення зовнішніх об’єктів, що використовуються, буде описана в п. 4.7.2. В оголошенні інтерфейсу вказується ім’я модуля та перелічуються його входи та виходи. Блок архітектури визначає, що модуль робить.

У сигналів *VHDL*, зокрема у входів і виходів, має бути зазначений тип. Цифрові сигнали необхідно оголошувати як *STD_LOGIC*. Сигнали цього типу набувають значення «0» або «1», а також рухоме й невизначене значення, що будуть описані в п. 4.2.9. Тип *STD_LOGIC* визначений у бібліотеці *IEEE.STD_LOGIC_1164*, тому бібліотеку обов’язково необхідно оголошувати. *VHDL* не визначає співвідношення пріоритетів операцій *AND* та *OR*, тому в процесі запису логічних виразів потрібно завжди використовувати дужки.

4.1.2 Походження мов SystemVerilog та VHDL

Приблизно в половині закладів вищої освіти, де викладають цифрову схемотехніку, читають *VHDL*, а в половині – *Verilog*. У промисловості схиляються до *SystemVerilog,* але чимало компаній ще використовують *VHDL*, тому багатьом розробникам потрібно володіти двома мовами. Якщо порівняти з *SystemVerilog, VHDL* більш багатослівний і громіздкий, ніж можна було б очікувати від мови, розробленої комітетом («Верблюд – це кінь, розроблений комітетом», – американський жарт).

SystemVerilog

Verilog було розроблено компанією *Gateway Design Automation* 1984 року як фірмова мова для симуляції логічних схем. У 1989 році *Gateway* придбала компанія *Cadence,* і *Verilog* став відкритим стандартом 1990 року під управлінням спільноти *Open Verilog International.* Мова стала стандартом Інституту інженерів з електротехніки та електроніки (*IEEE*) – професійна спільнота, відповідальна за багато комп’ютерних стандартів, наприклад, *Wi-Fi* (802.11), *Ethernet* (802.3), і чисел з рухомою точкою (754) 1995 року. У 2005 році мова була розширена для впорядкування та кращої підтримки моделювання й верифікації систем. Ці розширення були об’єднані в єдиний стандарт, що зараз називається *SystemVerilog* (стандарт *IEEE* 1800-2009). Файли мови *SystemVerilog* мають розширення *.sv*.

VHDL

Абревіатура *VHDL* розшифровується як *VHSIC Hardware Description Language. VHSIC* походить від скорочення *Very High Speed Integrated Circuits* – назви програми міністерства оборони США. Розроблення *VHDL* було розпочато 1981 року міністерством оборони для опису структури та функціональності електронних схем. За основу для розроблення була взята мова програмування *ADA*. Початковою метою мови була документація, але потім вона була швидко адаптована для симуляції та синтезу. *IEEE* стандартизував його 1987 року, і після цього мова оновлювалася кілька разів. Цей розділ оснований на редакції *VHDL* 2008 року (стандарт *IEEE* 1076-2008), яка впорядковує мову в багатьох аспектах. На момент написання не всі функції стандарту *VHDL* 2008 підтримуються в САПР. Цей розділ використовує ті функції, які підтримуються в *Synplicity, Altera Quartus* та *Modelsim*. Файл мови *VHDL* має розширення *.vhd.*

Двома мовами можна повністю описати будь-яку електронну систему, але кожна мова має свої особливості. Краще використовувати мову, яка вже поширена у вашій організації, або ту, що вимагають ваші клієнти. Більшість САПР зараз дають змогу змішувати мови, тому різні модулі можуть бути написані різними мовами [1–7].

4.1.3 Симуляція та синтез

Дві основні цілі *HDL* – логічна симуляція та синтез. Під час симуляції на входи модуля подаються деякі дії та перевіряються виходи, щоб переконатися, що модуль функціонує коректно. Під час синтезу текстовий опис модуля перетворюється на логічні елементи.

Симуляція

Люди регулярно припускаються помилок. Помилки в цифровій апаратурі називають багами. Зрозуміло, що усунення багів у цифровій системі дуже важливе, особливо коли від правильної роботи апаратури залежить чиєсь життя. Тестування системи в лабораторії дуже трудомістке. Дослідити причини помилок у лабораторії може бути надто складно, оскільки спостерігати можна лише сигнали на контактах чипа, а те, що відбувається всередині його, безпосередньо спостерігати неможливо. Виправлення помилок після того, як система була випущена, може бути дуже дорого. Наприклад, виправлення однієї помилки в новітніх інтегральних мікросхемах коштує понад мільйон доларів і займає кілька місяців. Сумнозвісний баг у команді поділу з рухомою точкою (*FDIV*) у процесорі *Pentium* змусив корпорацію *Intel* відкликати чипи

після того, як вони були поставлені замовникам, що коштувало їм 475 млн доларів. Логічна симуляція необхідна для тестування системи до того, як її буде випущено.

На рис. 4.1 подано графіки сигналів із симуляції попереднього модуля *sillyfunction*, які демонструють, що модуль працює коректно (симуляція була проведена в програмі *ModelSim PE Student Edition* версії 10.0c. *Modelsim* був обраний, оскільки використовується комерційно й має студентську версію з можливістю безкоштовної симуляції до 10 тис. рядків коду) й приймає лог.1, коли *a, b, c* є 000, 100 або 101, як і зазначено в логічному виразі.

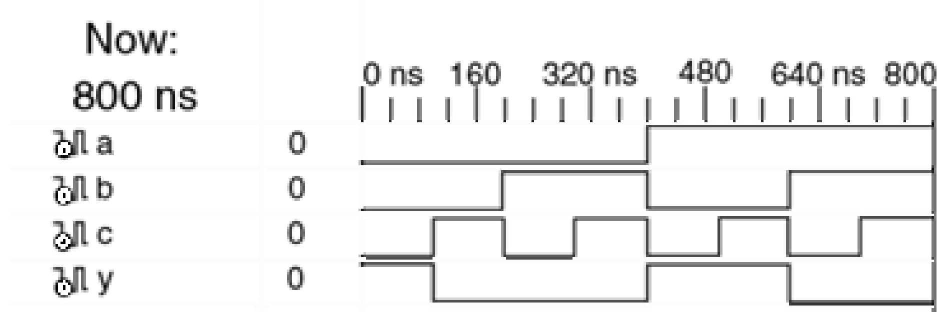


Рисунок 4.1 – Графіки сигналів

Синтез

Логічний синтез перетворює код на *HDL* в нетлист, що описує цифрову апаратуру (тобто логічні елементи й провідники, що їх з'єднують). Логічний синтезатор може виконувати оптимізацію скорочення кількості необхідних елементів. Нетлист може бути текстовим файлом або схемою, щоб було легко візуалізувати систему. На рис. 4.2 подано результати синтезу модуля *sillyfunction.* (Синтез був зроблений за допомогою програми *Synplify Premier* від *Synplicity*. Цей САПР був обраний, оскільки він є лідером серед комерційних продуктів для синтезу *HDL* в програмовані логічні інтегральні схеми (див. п. 5.6.2), а також доступний за ціною та дешевий для використання в університетах.) Зверніть увагу, що тривходові елементи I спрощені в 2 двовходових елементи I, як ми з'ясували в прикладі 2.6, використовуючи булеву алгебру.

Опис схем *HDL* нагадує програмний код. Однак ви маєте пам'ятати, що ваш код призначено для опису апаратури. *SystemVerilog* та *VHDL* – складні мови з безліччю операторів. Не всі з них можуть бути синтезовані в апаратурі: наприклад, оператор виведення результатів на екран під час симуляції не перетворюється на цифрову схему. Оскільки наше основне завдання – створення цифрової схеми, ми зосереджуємо увагу на синтезованій підмножині мов. Точніше, ми ділитимемо код *HDL* на синтезовані модулі та середовище

тестування. Синтезовані модулі описують цифрову схему. Середовище тестування містить код, що подає на входи модуля й перевіряє правильність значень його виходів, а також виводить невідповідності між очікуваними та дійсними значеннями. Код середовища тестування призначається лише симуляції та може бути синтезований.

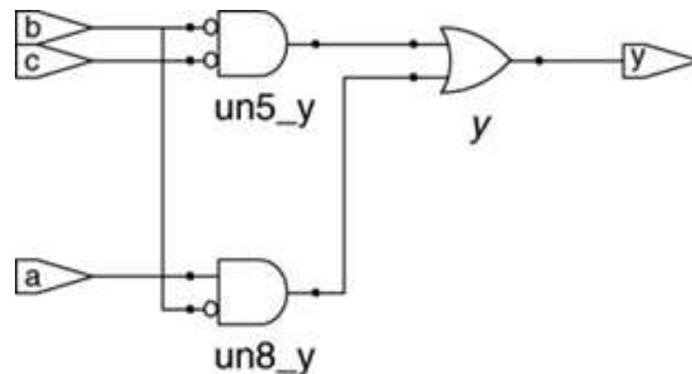


Рисунок 4.2 – Схема *sillyfunction*

Одна з основних помилок початківців полягає в тому, що вони думають про код *HDL* як про комп'ютерну програму, а не як про підмогу для опису цифрової апаратури. Якщо ви не уявляєте, хоча б приблизно, у що має синтезуватися ваш код *HDL*, то, швидше за все, результат вам не сподобається. Ваша цифрова схема може вийти набагато більшою, ніж потрібно, або виявиться, що ваш код симулюється правильно, але не може бути реалізований в апаратурі. Натомість ви маєте думати над вашою розробкою в поняттях комбінаційної логіки, регістрів та кінцевих автоматів. Накресліть ці блоки на папері та покажіть, як вони будуть під'єднані до того, як ви почнете писати код.

На наш досвід, найкращий спосіб вивчити *HDL* на прикладах. *HDL* є певні способи опису різних типів логіки; ці методи називаються ідіомами. У цьому розділі навчимо, як писати ідіоми для блоків кожного типу логіки й потім скласти блоки разом, щоб отримати робочу систему. Коли вам знадобиться описати апаратуру певного типу, подивіться на схожий приклад та адаптуйте його під свої цілі. Ми не намагатимемося суворо описувати весь синтаксис *HDL*, оскільки це нудно й сприяє уяві про *HDL* як про мови програмування, а не як про допомогу для розроблення апаратури. Якщо вам знадобиться додаткова інформація про особливості мов, зверніться до специфікацій *VHDL* і *SystemVerilog*, виданих *IEEE*, а також до літературних або інтернет-джерел, поданих у кінці посібника.

4.2 Комбінаційна логіка

Пам’ятайте, що ми тренуємося проєктувати синхронні послідовні схеми, які містять комбінаційну логіку та регістри. Стан виходів комбінаційної схеми залежить від вхідних сигналів. У цьому підрозділі описано, як створити поведінкові моделі комбінаційної логіки з використанням *HDL*.

4.2.1 Побітові оператори

Бітові оператори маніпулюють однобітовими сигналами чи багаторозрядними шинами. Так, модуль *inv* у прикладі 4.2 описує чотири інвертори, які під’єднані до чотирирозрядних шин.

Приклад 4.2.
Інвертори

SystemVerilog

```
module inv (input logic [3:0] a,
 output logic [3:0] y);
assign y = ~a;
endmodule
```

a[3:0] є чотирибітною шиною. Біти, від старшого до молодшого, записуються так: a[3], a[2], a[1] та a[0]. Такий порядок бітів називається *little-endian,* оскільки молодший біт має найменший бітовий номер. Ми могли б назвати шину a[4:1], і тоді a[4] був би старшим. Або ми могли б написати a[0:3], і тоді порядок бітів від старшого до молодшого був би таким: a[0], a[1], a[2] та a[3]. Такий порядок бітів називається *big-endian*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
 port(a: in STD_LOGIC_VECTOR(3 downto 0);
  y: out STD_LOGIC_VECTOR(3 downto 0));
end;
архітектура synth of inv is
begin
y <= not a;
 end;
```

У *VHDL* визначення шин типу *STD_LOGIC* використовується *STD_LOGIC_VECTOR. STD_LOGIC_VECTOR (3 downto 0)* є чотирибітною шиною. Біти від старшого до молодшого: a(3), a(2), a(1) та a(0). Така послідовність бітів називається *little-endian*, оскільки молодший біт має найменший бітовий номер. Ми могли б оголосити шину як *STD_LOGIC_VECTOR (4 downto 1)*, і тоді четвертий біт був би старшим. Або могли б записати *STD_LOGIC_VECTOR (0 down to 3)*, тоді порядок бітів від старшого до молодшого був би таким: a(0), a(1), a(2) та a(3). Такий порядок бітів називається *big-endian*.

Порядок прямування розрядів шини є суто умовною. (*Дивіться походження терміна на панелі посилань п. 6.2.2*.) Справді, і в прикладі порядок бітів не важливий, оскільки для набору інверторів немає значення, де який біт перебуває. Порядок бітів має значення лише для деяких операторів, наприклад, оператора додавання, в яких сума з одного стовпця переноситься в інший. Будь-який порядок є прийнятним, якщо він використовується послідовно. Ми будемо постійно застосовувати порядок бітів зліва направо від старшого до молодшого, [N − 1:0] мовою *SystemVerilog* та (*N − 1 downto 0*) мовою *VHDL* для *N*-розрядної шини.

Після кожного прикладу коду в цьому розділі наводиться схема, створена із кодом *SystemVerilog* інструментом синтезу *Synplify Premier*. Рис. 4.3 демонструє, що модуль *inv* синтезується як блок з чотирьох інверторів, позначених символом інвертора з написом *y[3:0].* Блок інверторів з’єднаний з чотирибітними вхідними та вихідними шинами. Подібна апаратна реалізація виходить із синтезованого *VHDL*-коду.

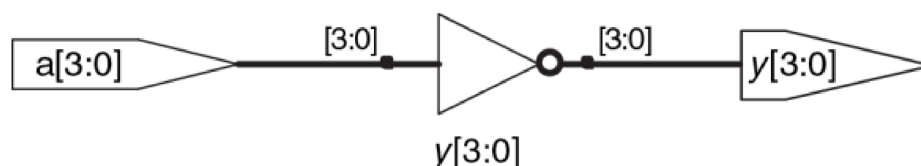


Рисунок 4.3 – Синтезована схема модуля *inv*

Модуль *gates* у прикладі 4.3 описує бітові операції, що виконуються на чотирибітних шинах для інших основних логічних функцій.

Приклад 4.3.
Логічні елементи

SystemVerilog

```
module gates(input logic [3:0] a, b,
```

```
 output logic [3:0] y1, y2, y3, y4, y5);
/*five different 2-input logic gates acting on 4-bit busses */
 assign y1 = a & b; // AND
 assign y2 = a | b; // OR
 assign y3 = a^b; // XOR
 assign y4 = ~ (a & b); // NAND
 assign y5 = ~ (a | b); // NOR
endmodule
```

Символи ~, ^ і | – це приклади операторів у мові *SystemVerilog*, тоді як *a, b* та *y*1 – операнди. Комбінація операторів та операндів, зокрема *a&b* або *~(a|b)* називається виразом. Повна команда, така як *assign y4 = ~(a&b);* називається оператором. *assign out = in1 op in2;* називається оператором безперервного присвоєння та закінчується крапкою з комою. Коли в операторі безперервного надання вхідні значення праворуч від знака «=» змінюються, результат ліворуч від знака «=» обчислюється заново. Отже, безперервне присвоєння описує комбінаційну логіку.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity gates is
port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
 y1, y2, y3, y4,y5: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of gates is
begin
-- five different 2-input logic gates
-- acting on 4-bit busses
y1 <= a and b;
y2 <= a or b;
 y3 <= a xor b;
y4 <= a nand b;
 y5 <= a nor b;
end;
```

НІ, що виключає АБО та АБО – це приклади операторів у мові *VHDL*, тоді як *a, b* та *y*1 – операнди. Комбінація операторів та операндів, зокрема *a and b* або *a nor b*, називається виразом. Повна команда, наприклад *y4 = a nand b;* називається оператором. *out = in1 op in2;* називається оператором

безперервного присвоєння сигналу. Оператори присвоєння *VHDL* закінчуються крапкою з комою. Коли в операторі безперервного надання сигналу вхідні значення праворуч від знака «=» змінюються, результат ліворуч від знака «=» обчислюється заново. Отже, оператор надання безперервного сигналу описує комбінаційну логіку.

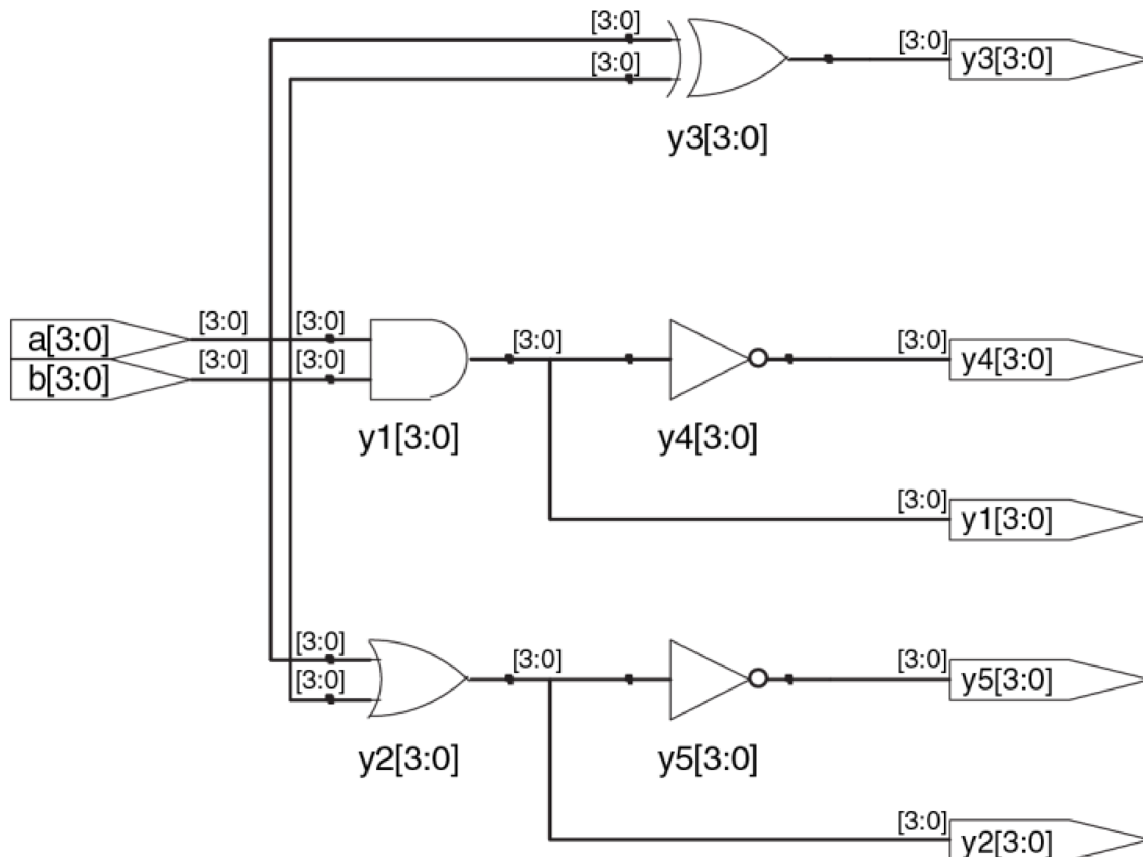


Рисунок 4.4 –Синтезована схема модуля *gates*

4.2.2 Коментарі та прогалини

Приклад із модулем *gates* демонструє, як оформляти коментарі. Мови *SystemVerilog* і *VHDL* не мають особливих вимог щодо використання вільного простору (наприклад, пробіли, табуляція та розриви рядків). Але належні відступи й застосування порожніх рядків допомагають зробити незвичайні розробки, що читаються. Будьте послідовні у використанні великих літер та підкреслень в іменах сигналів і модулів. У цьому тексті застосовуються лише малі літери. Імена сигналів і модулів не мають починатися з цифр.

SystemVerilog

Коментарі в мові *SystemVerilog* схожі на коментарі мов *C* або *Java*. Коментарі, що починаються з «/*», можуть займати кілька рядків до наступного знака «*/». Коментарі, що починаються з «//», продовжуються до кінця рядка.

SystemVerilog чутливий до регістра символів (великих і малих літер). *y1* і *Y1 SystemVerilog* – це різні сигнали.

VHDL

Коментарі, що починаються з «/*», можуть займати кілька рядків до наступного знака «*/». Коментарі, що починаються з «--», продовжуються до кінця рядка.

VHDL не чутливий до регістра символів. У *VHDL y1* і *Y1* – це той самий сигнал. Однак інші програми, що відкривають ваш файл, можуть виявитися чутливими до регістра символів, що спричиняє неприємні помилки, якщо ви змішуєте великі й малі літери.

4.2.3 Оператори скорочення

Оператори скорочення відповідають багатовходовим елементам, що працюють на одній шині. Приклад 4.4 описує восьмивходовий вентиль І з входами *a7, a6, ..., a0*. Аналогічні оператори скорочення існують для вентилів АБО, що виключає АБО, І-НІ, АБО-НІ і що виключає АБО з інверсією. Запам’ятайте, що багатовхідний вентиль, що виключає АБО, здійснює функцію контролю парності, повертаючи значення ІСТИНА, якщо непарна кількість входів має стан ІСТИНА.

Приклад 4.4.
Восьмивходовий вентиль І

SystemVerilog

```
module and8(input logic [7:0] a,
output logic          y);
 assign y = &a;
// &a is much easier to write than
// assign y = a[7] & a[6] & a[5] & a[4] &
//       a[3] & a[2] & a[1] & a[0];
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity and8 is
port(a: in STD_LOGIC_VECTOR(7 downto 0);
 y: out STD_LOGIC);
end;
```

```
architecture synth of and8 is
begin
y <= and a;
— and a is much easier to write than
— y <= a(7) and a(6) and a(5) and a(4) and
—     a(3) and a(2) and a(1) and a(0);
end;
```

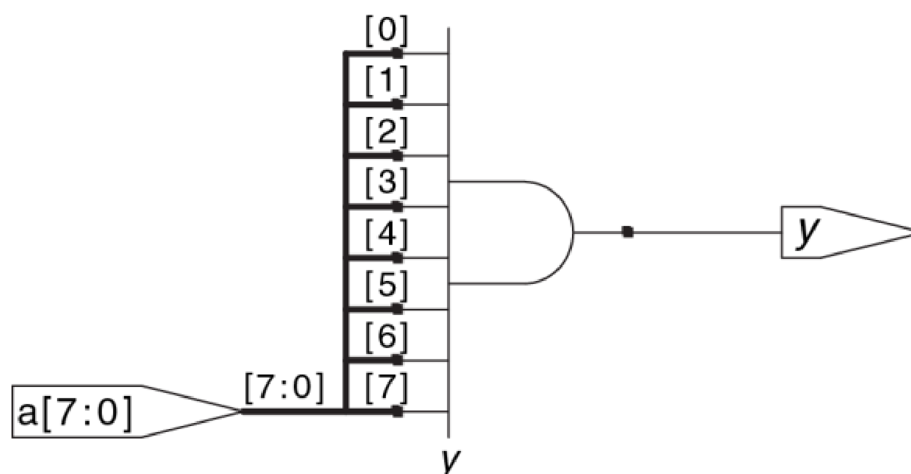


Рисунок 4.5 – Схема модуля *and8*

4.2.4 Умовне присвоєння

Оператори умовного присвоєння обирають певний вихід серед інших, з огляду на стан входу, що називається УМОВА. У прикладі 4.5 поданий двовходовий мультиплексор, який використовує умовне присвоєння.

Приклад 4.5.
Двовходовий мультиплексор

SystemVerilog

Умовний оператор *?:* обирає між другим і третім виразами, керуючись першим виразом. Перший вираз називається умовою (*condition*). Якщо умова набуває значення 1, то оператор обирає другий вираз. Якщо умова набуває значення 0, то оператор обирає третій вираз. Оператор ?: особливо корисний для опису мультиплексорів, оскільки на підставі стану першого входу він обирає між двома іншими. Наступний код демонструє програмну реалізацію двовходового мультиплексора з чотирибітними входами й виходами із застосуванням умовного оператора [1–7].

```
module mux2(input logic [3:0] d0, d1, input logic s,
output logic [3:0] y);
 assign y = s? d1: d0;
endmodule
```

Якщо *s* дорівнює 1, то *y = d1,* інакше *y = d0*. Оператор ?: також називають тернарним, оскільки він має три входи. З такою ж метою він використовується в мовах *C* та *Java*.

VHDL

Умовне присвоєння сигналу здійснює різні операції, що залежать від певних умов. Вони особливо корисні для опису мультиплексорів.

Наприклад, двовходовий мультиплексор може застосовувати умовне присвоєння сигналу для вибору одного з двох чотирибітних входів.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
 s: in STD_LOGIC;
y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of mux2 is
 begin
y <= d1 when s else d0;
 end;
```

Умовне присвоєння сигналу встановлює *y d1*, якщо *s* має значення 1. В іншому разі він встановлює *y d0*. Зауважте, що у версіях *VHDL* до 2008 р. потрібно було писати *when s = ‘1’*, а не *when s*.

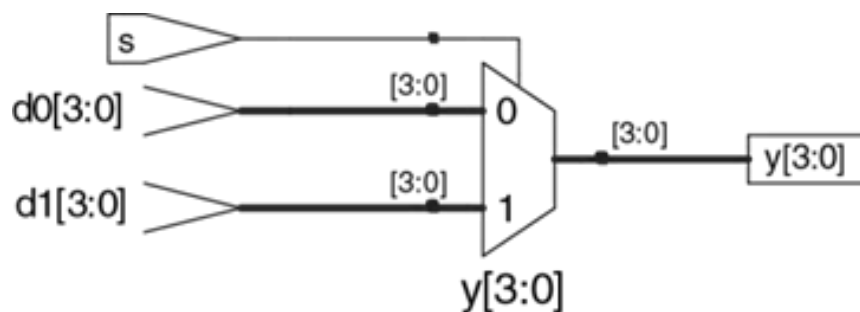


Рисунок 4.6 – Схема модуля *mux2*

У прикладі 4.6 продемонстрований чотиривходовий мультиплексор, що працює за тим самим принципом, що мультиплексор з прикладу 4.5. На рис. 4.7 зображено схему, створену за допомогою *Synplify Premier*. Програмне забезпечення використовує познаку мультиплексора, що різниться від тої, що досі наводилася в тексті. Мультиплексор має багаторозрядні входи даних (*d*) та одиночні входи роздільної здатності (*e*). Коли один із входів активовано, відповідні дані відправляються на вихід. Наприклад, коли *s*[1] = *s*[0] = 0, нижній вентиль – *un1_s_5* формує 1, активуючи нижній вхід мультиплексора, унаслідок обирається *d0[3:0]*.

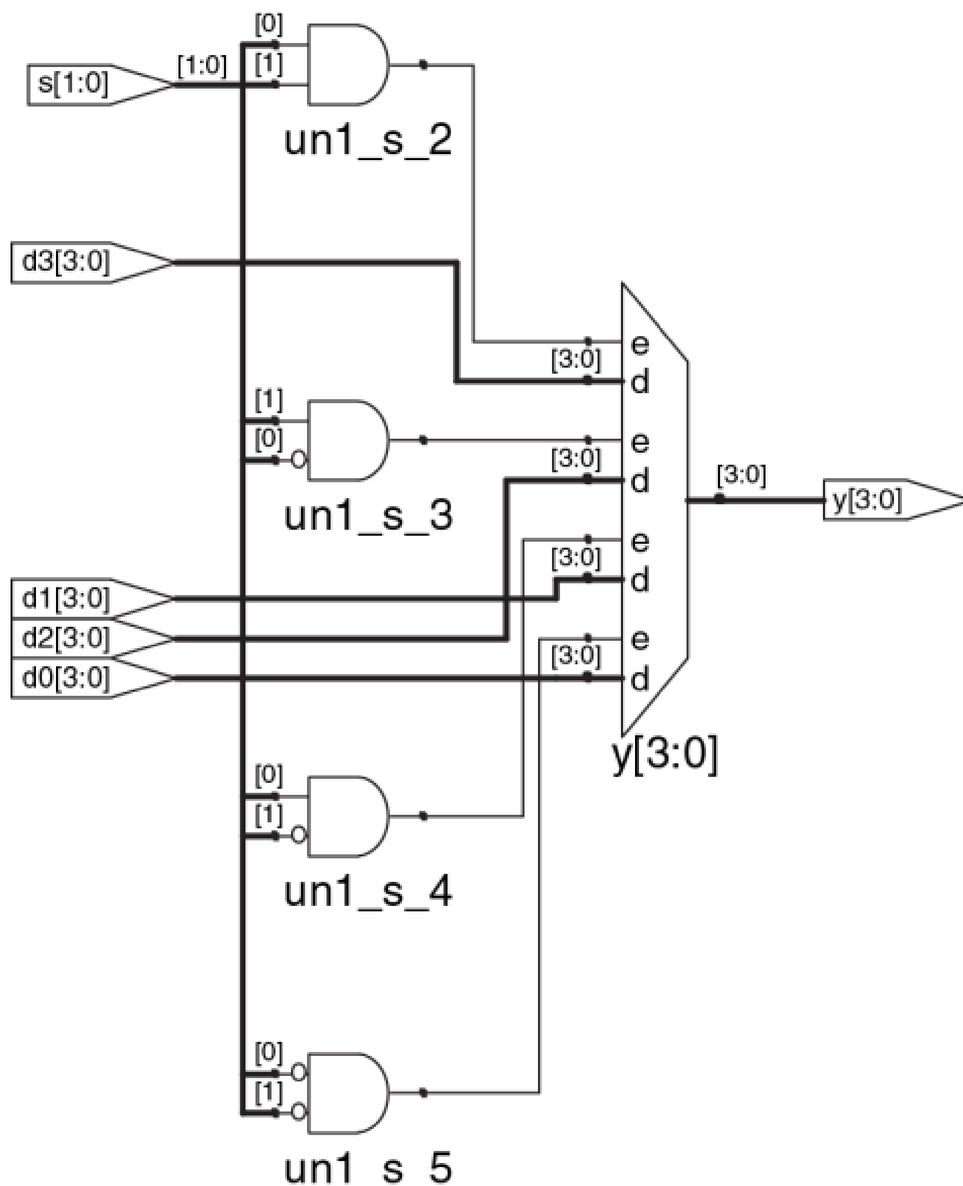


Рисунок 4.7 – Синтезована схема модуля *mux4*

Приклад 4.6.

Чотиривходовий мультиплексор

SystemVerilog

Чотириводовий мультиплексор може обрати один із чотирьох входів за допомогою вкладених умовних операторів.

```
module mux4(input logic [3:0] d0, d1, d2, d3,
            input logic [1:0] s,
            output logic [3:0] y);
 assign y = s [1]? (s[0] ? d3 : d2)
             : (s[0] ? d1 : d0);
endmodule
```

Якщо *s[1]* набуває значення 1, то мультиплексор обирає перший вираз (*s[0] ? d3 : d2*). Це вираз зі свого боку обирає або *d3*, або *d2* на основі *s[0] (y = d3*, якщо *s[0]* має значення 1 та *d2*, якщо *s[0]* має значення 0). Якщо *s[1]* має значення 0, тоді мультиплексор подібним чином обирає другий вираз, який дає *d1* або *d0* залежно від *s[0]*.

VHDL

Чотиривходовий мультиплексор може обрати один із входів за допомогою кількох умов *else* в операторі умовного присвоєння сигналу.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
    port(d0, d1,
   d2, d3: in STD_LOGIC_VECTOR(3 downto 0);
    s: in STD_LOGIC_VECTOR(1 downto 0);
    y: out STD_LOGIC_VECTOR(3 downto 0));
end;
архітектура synth1 of mux4 is
begin
     y <= d0 when s = "00" else
            d1 when s = "01" else
            d2 when s = "10" else
            d3;
end;
```

VHDL також підтримує операторів вибіркового надання сигналу для забезпечення більш короткого запису, коли обирається одна з кількох можливостей. Це аналогічно використанню операції *switch/case* замість кількох операцій *if/else* у деяких мовах програмування. Чотиривходовий мультиплексор може бути переписано із застосуванням вибіркового надання сигналу таким чином:

```
архітектура synth2 of mux4 is
begin
    with s select
y <= d0 when "00",
      d1 when "01",
      d2 when "10",
      d3 others;
end;
```

4.2.5 Внутрішні змінні

Часто зручно розбити складну функцію на кілька проміжних. Наприклад, повний суматор, що буде описано в п. 5.2.1, є схемою з трьома входами й двома виходами, що визначаються такими рівняннями:

$$S = A \wedge B \wedge Cin,$$
$$Cout = AB + ACin + BCin. \quad (4.1)$$

Якщо ми введемо проміжні сигнали *P* та *G*:

$$P = A \wedge B$$
$$G + AB, \quad (4.2)$$

зможемо переписати рівняння для повного суматора у вигляді:

$$S = P \wedge Cin,$$
$$Cout = G + PCin. \quad (4.3)$$

Змінні *P* і *G* називаються внутрішніми, оскільки вони є ні входами, ні виходами, а використовуються лише всередині модуля. Вони подібні до локальних змінних у мовах програмування. Приклад 4.7 демонструє, як ці змінні використовуються *HDL*.

Операції присвоєння *HDL* (*assign* у мові *SystemVerilog* і = в *VHDL*) відбуваються паралельно. У цьому полягає різниця від традиційних мов програмування, таких як *C* або *Java*, де оператори оцінюються в тому порядку, у якому вони записані. У традиційних мовах важливо, що вираз $S = P \wedge Cin$ слідує за виразом $P = A \wedge B$, оскільки оператори виконуються послідовно. У *HDL* порядок запису немає значення. Подібно до апаратних засобів,

оператори присвоєння *HDL* виконуються в момент, коли входи та сигнали з правого боку виразу змінюють своє значення незалежно від порядку, у якому оператори присвоєння з'являються в модулі.

Приклад 4.7.
Повний суматор

SystemVerilog

У мові *SystemVerilog* внутрішні сигнали зазвичай оголошуються як логічні.

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
   logic p, g;
 assign p = a^b;
 assign g = a & b;
 assign s = p^cin;
 assign cout = g | (p & cin);
 endmodule
```

VHDL

У *VHDL* сигнали зазвичай використовують для подання внутрішніх змінних, значення яких визначаються одночасними операторами присвоєння, таких як **p = xor b;**

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity fulladder is
    port(a, b, cin: in STD_LOGIC;
   s, cout:     out STD_LOGIC);
end;
architecture synth of fulladder is
 signal p, g: STD_LOGIC;
begin
p <= a xor b;
g <= a and b;
s <= p xor cin; cout <= g or (p and cin);
end;
```

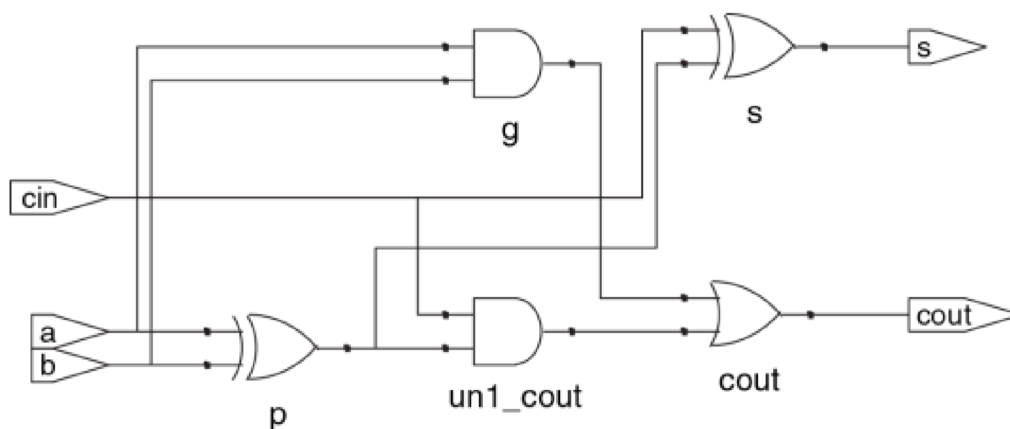


Рисунок 4.8 – Схема модуля *fulladder*

4.2.6 Пріоритет

Зверніть увагу, що ми застосовували дужки в обчисленні *cout* у прикладі 4.7, щоб визначити порядок операцій: *Cout = G + (P • Cin),* а не *Cout = (G + P) • Cin.* Якщо ми не використовуємо дужки, порядок операцій визначається за замовчуванням. Приклад 4.8 визначає пріоритет операцій від вищої до нижчої для кожної мови. У таблицях подано арифметичні операції, операції зсуву й операції порівняння, що будуть розглянуті в розділі 5.

Приклад 4.8.
Пріоритет операцій

SystemVerilog

Таблиця 4.1

Операція	**Значення**
~	Побітове заперечення (НІ)
*, /, %	Множення, ділення, залишок
+, –	Додавання, віднімання
<<, >>	Зсув вліво / вправо
<<<, >>>	Арифметичний зсув вліво / вправо
<, <=, >, >=	Порівняння на більше-менше
= =, !=	Порівняння на рівність
&, ~&	І, І-НІ
^, ~^	Виключає АБО, що виключає АБО-НІ
|, ~|	АБО, АБО-НІ
?:	Умовний оператор

Система пріоритету операторів для *SystemVerilog* подібна до систем, прийнятих в інших мовах програмування. Зокрема І має пріоритет над АБО.

Можна користуватися пріоритетом операторів, щоб унеможливити використання круглих дужок.

assign cout = g | p & cin;

VHDL

Таблиця 4.2

	Операція	Значення
Вищий	not	НІ
	*, /, mod, rem	Множення, ділення, модуль, залишок
	+, –	Додавання, віднімання
	rol, ror, srl, sll	Циклічний зсув вліво / вправо Логічний зсув вліво / вправо
Нижчий	<, <=, >, >=	Порівняння на більше-менше
	=, /=	Порівняння на рівність
	and, or, nand, nor, xor, xnor	Логічні операції

У *VHDL* множення має пріоритет над додаванням. Однак, на відміну від *SystemVerilog*, всі логічні оператори (*and, or* і т. д.) мають однаковий пріоритет. Тому дужки необхідні, інакше *cout = g or p and cin* буде інтерпретуватися зліва направо як *cout = (g or p) and cin*.

4.2.7 Числа

Числа вказують у двійковій, вісімковій, десятковій або шістнадцятковій системі числення (основа 2, 8, 10 та 16 відповідно). Розмір, тобто кількість бітів, може бути також вказано, вільні розряди заповнюються нулями. Підкреслення в числах ігноруються і можуть бути корисними лише тоді, коли потрібно розбити довге число на більш читані фрагменти. Приклад 4.9 пояснює, як числа записуються в кожній мові.

Приклад 4.9.
Числа

SystemVerilog

Формат для оголошення констант – *N'Bvalue,* де *N* – розмір у бітах, *B* – літера, що вказує на основу та *value* – значення. Наприклад, *9'h25* визначає дев'ятибітне число зі значенням 25_{16} =37_{10}= 000100101_2. *SystemVerilog*

підтримує *'b* для основи 2, *'o* – для основи 8, *'d* – для основи 10 і *'h* – для основи 16. Якщо основа опущена, то за замовчуванням вона дорівнює 10. Якщо не вказано розмір, то передбачається, що число має стільки ж бітів, скільки й вираз, у якому воно використовується. Старші (неповні) розряди доповнюються нулями автоматично до повного розміру. Наприклад, якщо *w* – шестибітна шина, то *assign w = 'b11* присвоює *w* значення 000011. Кращою практикою є явна вказівка розміру. Винятком є те, що *'0* і *'1* слугують конструкціями *SystemVerilog* для заповнення шини нулями або одиницями відповідно.

Таблиця 4.3

Запис	Число бітів	Основа	Значення	Подання
3'b101	3	2	5	101
'b11	?	2	3	0000011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	000101010

VHDL

У *VHDL* числа *STD_LOGIC* записуються в бінарному коді та беруть в одинарні лапки: '0' and '1' вказують на логічні рівні 0 і 1. Формат оголошення констант типу *STD_LOGIC_VECTOR* такий: *NB"value"*, де *N* – розмір у бітах, літера, що вказує на основу, та *value* – значення. Наприклад, *9X"25"* визначає дев'ятибітне число зі значенням $25_{16} = 37_{10} = 000100101_2$. *VHDL 2008* підтримує *B* для основи 2, *O* – для основи 8, *D* – для основи 10 та *X* – для основи 16. Якщо підстава опущена, то за замовчуванням вона дорівнює 10. Якщо розмір не вказано, то передбачається, що число має розмір, який відповідає числу бітів. Станом на жовтень 2011 року *SynplifyPremier* від *Synopsys* не підтримує вказівку розміру.

others = '0' і *others = '1'* – конструкції *VHDL* із заповненням всіх бітів нулями або одиницями відповідно.

Таблиця 4.4

Запис	Число бітів	Основа	Значення	Подання
3B“101”	3	2	5	101
B“11”	2	2	3	11
8B“11”	8	2	3	00000011
8B“1010_1011”	8	2	171	10101011
3D“6”	3	10	6	110
6O“42”	6	8	34	100010
8X“AB”	8	16	171	10101011
“101”	3	2	5	101
B“101”	3	2	5	101
X“AB”	8	16	171	10101011

4.2.8 Стани z та x

HDL

Стан *z* використовується для вказівки на рухоме значення. Застосування *z*-стану, зокрема, є корисним для опису буфера з трьома станами, стан виходу якого є рухомим, коли на вхід дозволу подано 0. Згадайте з п. 2.6.2, що шиною можна управляти кількома буферами з трьома станами, тільки один з яких має бути активним. У прикладі 4.10 подано програмну реалізацію тристабільного буфера. Якщо цей буфер активовано, стан на виході буде таким самим, як і на вході. Якщо буфер не активовано, стан на виході призначається рухомим значенням (*z*).

Приклад 4.10

Тристабільний буфер

SystemVerilog

```
module tristate(input logic [3:0] a,
                input logic en,
                output tri [3:0] y);
assign y = en? a : 4’bz;
endmodule
```

Зверніть увагу, що *y* оголошується як *tri*, а не *logic*. Сигнали типу *logic* можуть мати лише один драйвер. Тристабільні шини можуть мати кілька драйверів, тому вони мають оголошуватися як *net*. Два типи *net* у *SystemVerilog* мають назви *tri* та *trireg*. Зазвичай тільки один драйвер у мережі активний

у конкретний момент часу, і мережа набуває значень, що задаються. Якщо жоден із драйверів не активовано, то *tri* плаває (*z*), тоді як *trireg* зберігає попереднє значення. Якщо для входу чи виходу тип не вказано, передбачається, що тип – *tri.* Також зауважте, що вихід модуля типу *tri* може використовуватися як вхід типу *logic* для інших модулів. Надалі ланцюги з кількома драйверами розглядатимуться в підрозділі 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity tristate is
   port(a: in STD_LOGIC_VECTOR(3 downto 0);
   en: in STD_LOGIC;
   y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of tristate is
 begin
y <= a when en else “ZZZZ”;
end;
```

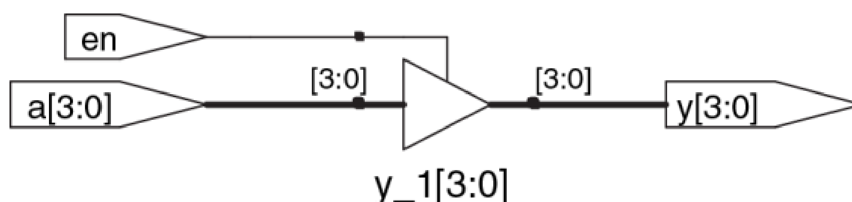


Рисунок 4.9 – Схема модуля *tristate*

Також *HDL* використовують *x* для вказівки недійсного логічного рівня. Якщо на шину одночасно потрапляє 0 і 1 із двох активних тристабільних буферів (або інших елементів), то зрештою отримуємо *x*, що вказує на конфлікт. Якщо всі тристабільні буфери, які управляють шиною, одночасно перебувають у стані *OFF*, то в шині буде рухомий стан, на що вказує *z*. На початку моделювання стану вузлів, таких як виходи тригерів, ініціалізуються невідомим станом (*x SystemVerilog і u − VHDL*). Це допомагає відстежувати помилки, які з’являються, якщо ви забули скинути тригер перед використанням його виходу.

Якщо логічний елемент отримує рухоме значення на вході, він може сформувати *x* на виході, коли йому не вдається визначити правильне вихідне значення. Якщо елемент отримує на вході недійсне або неініціалізоване значення, то на виході може сформувати *x*. Приклад 4.11 показує,

як у *SystemVerilog* та *VHDL* комбінують ці різні значення сигналів у логічних елементах. Стани *X* або *u* в моделюванні практично завжди означають помилки або поганий стиль програмування. У синтезованому ланцюгу це відповідає рухомому входу елемента, неініціалізованому стану або конфлікту. *X* або *u* можуть бути випадково інтерпретовані схемою як 0 або 1, що призведе до непередбачуваної поведінки програми.

Приклад 4.11.
Таблиці істинності з невизначеними й рухомими входами

SystemVerilog

Сигнали в *SystemVerilog* можуть набувати значення 0, 1, *z* і *x*. Константи *SystemVerilog,* що починаються з *z* або *x*, якщо необхідно, доповнюються символами *z* або *x* у старших розрядах (замість нулів) для досягнення необхідної довжини.

Табл. 4.5 демонструє таблицю істинності для вентиля І, використовуючи всі чотири можливі значення сигналу. Зауважте, що вентиль може іноді визначати вихід, незважаючи на невідомий стан деяких входів. Наприклад, *0&z* повертає 0, оскільки на виході вентиля І завжди 0, якщо якийсь із входів має стан 0. В іншому разі рухомий, або некоректний, стан на входах призводить до недійсних станів на виходах, що позначається в *SystemVerilog* як *x*.

Таблиця 4.5

&		***A***			
B		0	1	z	x
	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

VHDL

Сигнали типу *STD_LOGIC у VHDL* можуть набувати значень ‘0’, ‘1’, ‘*z*’, ‘*x*’ та ‘*u*’.

Табл. 4.6 демонструє таблицю істинності вентиля І, використовуючи п’ять можливих значень сигналу. Зауважте, що вентиль може іноді визначати вихід, незважаючи на невідомі стани деяких входів. Наприклад, ‘0’ та ‘z’ повертає ‘0’, оскільки на виході вентиля І завжди ‘0’, якщо якийсь із входів має стан ‘0’. Інакше рухомий, або некоректний, стан на входах призводить до недійсних станів на виходах, що в *VHDL* позначаються як ‘*x*’.

Неініціалізовані стани входів призводять до неініціалізованих станів сигналів на виходах, що в *VHDL* позначаються як ‘*u*’.

Таблиця 4.6

<table>
<tr><th colspan="2">AND</th><th colspan="5">A</th></tr>
<tr><td rowspan="6">B</td><td colspan="2">0</td><td>1</td><td>z</td><td>x</td><td>u</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>x</td><td>x</td><td>u</td></tr>
<tr><td>z</td><td>0</td><td>x</td><td>x</td><td>x</td><td>u</td></tr>
<tr><td>x</td><td>0</td><td>x</td><td>x</td><td>x</td><td>u</td></tr>
<tr><td>u</td><td>0</td><td>u</td><td>u</td><td>u</td><td>u</td></tr>
</table>

Приклад 4.12.
Маніпуляції з бітами

SystemVerilog

assign y = {c[2:1], {3{d[0]}}, c[0], 3’b101};

Оператор {} використовується для об’єднання шин. *{3{d[0]}}* вказує на три копії *d[0]*. Не плутайте трибітну двійкову константу *3’b101* із шиною з ім’ям *b*. Зауважте, що визначення довжини трибітної константи має вирішальне значення; в іншому разі в середині *y* могла б з’явитися невідома кількість нулів. Якби розмірність *y* перевищувала 9 бітів, то нулі були б вміщені у старших бітах.

VHDL

y <=(c(2 downto 1), d(0), d(0), d(0), c(0), 3B“101”);

Оператор агрегування використовується для поєднання шин. Має бути дев’ятибітним сигналом типу *STD_LOGIC_VECTOR.*

Інший приклад демонструє можливості оператора агрегування *VHDL.* Припустимо, що *z* – це восьмибітний сигнал типу *STD_LOGIC_VECTOR*, тоді під час виконання операції агрегування

z <= (“10”, 4 => ‘1’, 2 downto 1 => ‘1’, others => ‘0’)

z набуде значення 10010110. “10” перетворюється на старшу пару бітів. Одиниця також міститься в бітах 4, 2 і 1. Всі інші біти дорівнюють 0.

4.2.9 Маніпуляція бітами

Часто програмістам доводиться працювати із фрагментом шини чи об’єднувати сигнали для формування шин. Ці операції називаються

маніпуляціями бітами. У прикладі 4.12 y задається дев'ятибітною змінною $c_2c_1d_0d_0d_0c_0$101 з використанням маніпуляцій бітами.

4.2.10 Затримки

Оператори *HDL* можуть бути пов'язані із затримками, зазначеними в довільних одиницях. У процесі моделювання вони допомагають передбачити, наскільки швидко працюватиме схема (якщо ви вкажете відповідні затримки), також під час налагодження вони допомагають зрозуміти причину й наслідок (встановлювати джерело поганого результату складно, якщо після моделювання всі сигнали змінюються одночасно). Ці затримки ігноруються в процесі синтезу; затримка елемента, згенерованого синтезатором, залежить від значень t_{pd} і t_{cd} , а не від чисел в *HDL*-коді.

У прикладі 4.13 додана затримка до початкової функції з прикладу 4.1, $y = \overline{a}*\overline{b}*\overline{c} + a*\overline{b}*\overline{c} + a*\overline{b}*c$. Передбачається, що інвертор має затримку 1 нс, трихвходовий елемент має затримку 2 нс, а трихвходовий елемент АБО – 4 нс. На рис. 4.1 подано результати моделювання із затримкою сигналу 7 нс щодо входів. Зауважте, що y невідомо на початку моделювання.

Приклад 4.13.
Логічні елементи із затримками

SystemVerilog

```
‘timescale 1ns/1ps
module example(input logic a, b, c,
                         output logic y);
logic ab, bb, cb, n1, n2, n3;
assign #1 {ab, bb, cb} = ~{a, b, c};
assign #2 n1 = ab & bb & cb;
assign #2 n2 = a & bb & cb;
 assign #2 n3 = a & bb & c;
assign #4 y = n1 | n2 | n3;
endmodule
```

Файли *SystemVerilog* можуть містити вказівку на шкалу часу, що встановлює значення кожного проміжку часу. Цей вираз має вигляд *‘timescale unit/precision*. У цьому файлі кожна одиниця часу дорівнює 1 нс, а моделювання проводиться з точністю 1 пс. Якщо у файлі немає вказівки на шкалу часу, то одиниця часу та точність використовуються за замовчуванням (зазвичай обидва

параметри дорівнюють 1 нс). У *SystemVerilog* символ # використовується для вказівки кількості одиниць затримки. Він може міститися в операції *assign*, а також у неприсвоєних (неблокувальних) (<=) і блокувальних (=) присвоєннях, які розглянемо в п. 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity example is
    port(a, b, c: in STD_LOGIC;
    y: out STD_LOGIC);
end;
архітектура synth of example is
signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
   n1 <= ab and bb and cb after 2 ns;
   n2 <= a and bb та cb after 2 ns;
   n3 <= a and bb and c after 2 ns;
     y <= n1 or n2 or n3 after 4 ns;
end;
```

У *VHDL after* застосовується для позначення затримок. Одиниці визначаються в наносекундах.

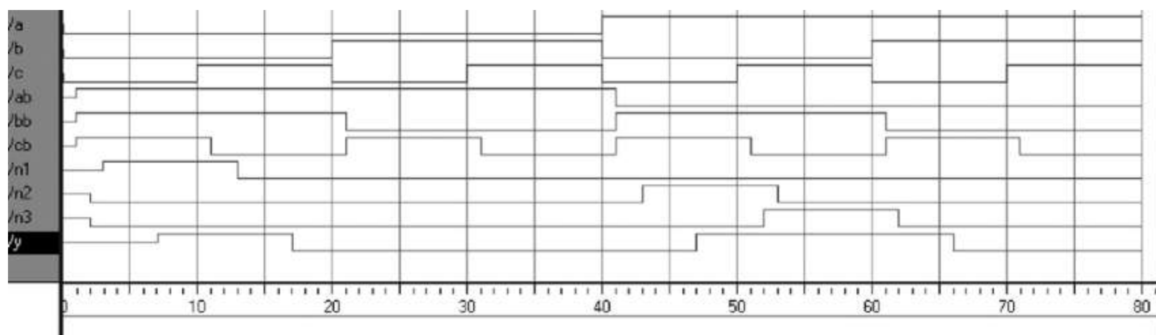


Рисунок 4.10 – Приклад моделювання сигналів із затримками (симулятор *ModelSim*)

4.3 Структурне моделювання

У попередньому підрозділі обговорювалося поведінкове моделювання, що описує модуль з погляду відношень між входами та виходами. Цей розділ вивчає структурне моделювання, що описує модуль з погляду того, як його складено з більш простих модулів.

Приклад опису *HDL* 4.14 демонструє, як збирається чотиривходовий мультиплексор із трьох двовходових мультиплексорів. Кожна копія двовходового мультиплексора називається екземпляром. Чимало екземплярів одного модуля розрізняються окремими назвами, в поданому прикладі це *lowmux, highmux* і *finalmux*. Це приклад системи, де двовходовий мультиплексор повторно використовується багато разів.

Приклад *HDL* 4.14.
Структурна модель чотиривходового мультиплексора

SystemVerilog

```
module mux4(input logic [3:0] d0, d1, d2, d3,
            input logic [1:0] s,
            output logic [3:0] y);
 logic [3:0] low, high;
 mux2 lowmux(d0, d1, s[0], low);
 mux2 highmux(d2, d3, s[0], high);
mux2 finalmux(low, high, s[1], y);
 endmodule
```

Три екземпляри модуля *mux2* називаються *lowmux, highmux* і *finalmux.* Модуль *mux2* має бути десь оголошений в *SystemVerilog*-коді (див. *HDL*-приклади 4.5, 4.15 або 4.34)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity mux4 is
    port(d0, d1,
    d2, d3: in STD_LOGIC_VECTOR(3 downto 0);
    s: in STD_LOGIC_VECTOR(1 downto 0);
    y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture struct mux4 is
```

```
component mux2
    port(d0,
    d1: in STD_LOGIC_VECTOR(3 downto 0);
    s: in STD_LOGIC;
    y: out STD_LOGIC_VECTOR(3 downto 0));
end component;
 signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
lowmux: mux2 port map(d0, d1, s(0), low);
 highmux: mux2 port map(d2, d3, s(0), high);
 finalmux: mux2 port map(low, high, s(1), y);
end;
```

В архітектурі спочатку мають бути оголошені порти *mux2* за допомогою оператора оголошення компонента. Це дає змогу інструментам *VHDL* перевірити, що компонент, який ви хочете використовувати, має ті самі порти, що й інтерфейс, який був оголошений десь ще в іншому операторі інтерфейсу. Це сприяє запобіганню помилок, викликаних зміною інтерфейсу, але не самого об'єкта. Однак оголошення компонента робить *VHDL*-код досить громіздким. Зауважте, що ця архітектура *mux4* була названа *struct*, тоді як архітектури модулів з поведінковими описами з підрозділу 4.2 називалися *synth. VHDL* дає змогу мати безліч архітектур (реалізацій) одного інтерфейсу; архітектури різняться іменами. Самі імена не мають значення для інструментів САПР, але *struct* і *synth* є загальноприйнятими. Синтезований *VHDL*-код, як правило, містить тільки одну архітектуру для кожного інтерфейсу, так що ми не обговорюватимемо *VHDL*-синтаксис, який використовується для налаштування того, яку архітектуру обирати, коли визначено безліч з них.

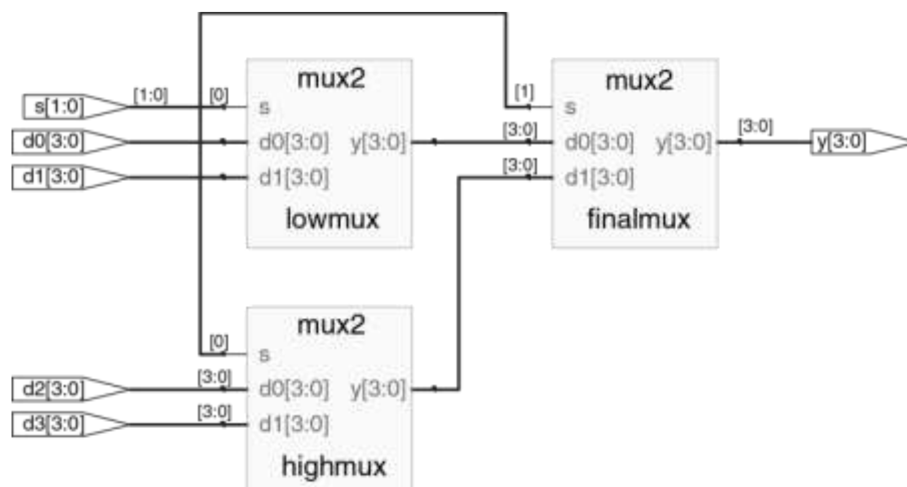


Рисунок 4.14 – Синтезована схема модуля *mux2*

У прикладі опису *HDL* 4.15 застосовується структурне моделювання створення двовходового мультиплексора з пари буферів із трьома станами. Однак побудова логіки таких буферів не рекомендується.

Приклад *HDL* 4.15.

Структурна модель двовходового мультиплексора

SystemVerilog

```
module mux2(input logic [3:0] d0, d1,
        input logic s, output tri [3:0] y);
        tristate t0(d0, ~s, y);
        tristate t1(d1, s, y);
endmodule
```

У мові *SystemVerilog* вислови, зокрема *~s*, дозволені в списку портів екземпляра. Допустимі вислови будь-якої складності, але це не заохочується, бо вони роблять код складним для читання.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity mux2 is
    port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
     s: in STD_LOGIC;
     y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture struct mux2 is
component tristate
port(a: in STD_LOGIC_VECTOR(3 downto 0);
       en: in STD_LOGIC;
       y: out STD_LOGIC_VECTOR(3 downto 0));
 end component;
signal sbar: STD_LOGIC;
begin
   sbar <= not s;
   t0: tristate port map(d0, sbar, y);
   t1: tristate port map(d1, s, y);
end;
```

У мові *VHDL* такі вислови, як *not s*, не дозволені в карті портів екземпляра. Отже, *sbar* має бути визначено як окремий сигнал.

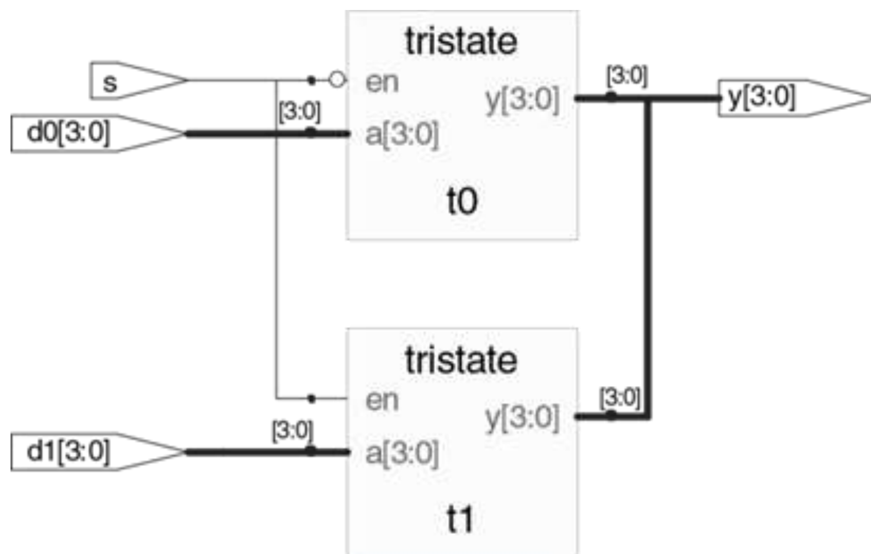


Рисунок 4.12 – Синтезована схема модуля *mux2*

Приклад опису *HDL* 4.16 демонструє, як модулі можуть отримати доступ до частини шини. Двовходовий мультиплексор розрядністю 8 бітів побудовано за допомогою двох чотирибітних двовходових мультиплексорів, що оголошені раніше та працюють з молодшим і старшим напівбайтами.

Загалом складні системи створюються ієрархічно. Система описується структурно за допомогою долучення до неї основних компонентів. Кожен із цих компонентів описується структурно зі своїх будівельних блоків і так далі рекурсивно доти, доки справа не дійде до частин досить простих для поведінкового опису. Хорошим стилем є прагнення уникнути (чи принаймні мінімізувати) змішування структурних і поведінкових описів усередині одного модуля.

Приклад *HDL* 4.16.
Звернення до частин шин

SystemVerilog
```
module mux2_8(input logic [7:0] d0, d1,
          input logic s, output logic [7:0] y);
mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2_8 is
port(d0, d1: in STD_LOGIC_VECTOR(7 downto 0);
      s: in STD_LOGIC;
      y: out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture struct of mux2_8 is
 component mux2
      port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
            s: in STD_LOGIC;
            y: out STD_LOGIC_VECTOR(3 downto 0));
end component;
begin
lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
                s, y(3 downto 0));
msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
                s, y(7 downto 4));
end;
```

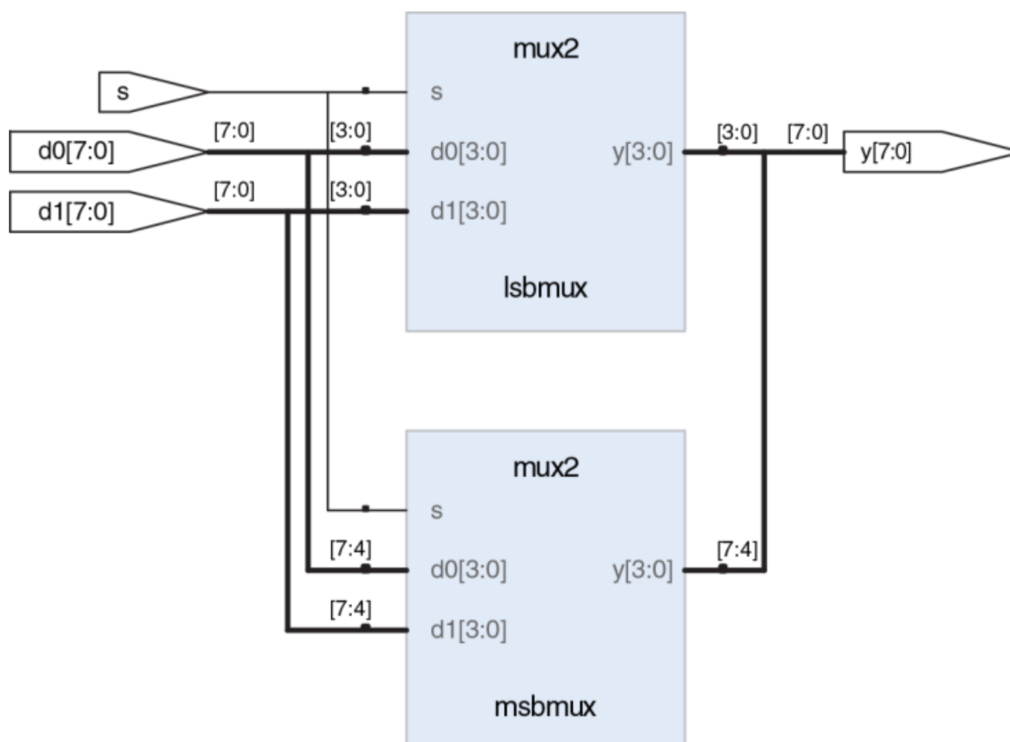


Рисунок 4.13 – Синтезована схема модуля *mux2*

4.4 Послідовна логіка

Синтезатори *HDL* розпізнають певні ідіоми та перетворюють їх на конкретні послідовні схеми. Код, написаний в іншому стилі, може правильно симулюватися, але в синтезованій схемі можуть виявитися грубі помилки, що важко розпізнати. У цьому підрозділі подані ідіоми, рекомендовані для опису регістрів та засувок.

4.4.1 Регістри

Переважна більшість сучасних комерційних систем побудована на регістрах, що використовують *D*-тригери, які спрацьовують по передньому фронту тактового імпульсу. У прикладі 4.17 показана ідіома такого тригера.

Сигнали, значення яких присвоєно в операторах *always* мови *SystemVerilog* і операторах *process* мови *VHDL*, зберігають свій стан, доки не станеться подія зі списку чутливості оператора, що призводить до зміни їх значення. Тому код, що використовує ці оператори з відповідними списками чутливості, може описувати послідовні схеми з пам’яттю. Наприклад, у тригера в списку чутливості є тільки сигнал *clk*, і тому тригер зберігає старе значення *q* до наступного переднього фронту *clk*, навіть якщо вхідний сигнал *d* змінився раніше.

На відміну від них, оператор безперервного присвоєння *SystemVerilog* (*assign*) і оператор одночасного присвоєння *VHDL* (<=) перелічується щоразу, коли змінюється якась із змінних у правій частині, тому ці оператори можуть описати тільки комбінаційну логіку (за допомогою цих операторів можна описувати й логіку, що зберігає стан, наприклад *assign q = clk ?d : q;* але робити це не рекомендується).

Приклад 4.17.
Регістр

SystemVerilog
```
module flop(input logic clk,
            input logic [3:0] d;
            output logic [3:0] q);
always_ff @ (posedge clk)
   q <= d;
endmodule
```
Загалом оператор *always* мови *SystemVerilog* має вигляд

```
always @(sensitivity list)
statement;
```

Оператор виконується лише тоді, коли трапляється подія, задана в списку чутливості. У цьому прикладі оператором є *q* <= *d* (читається «*q* набуває значення *d*»). Отже, тригер копіює *d* в *q* по передньому фронту тактового сигналу, а потім значення *q* залишається незмінним. Зазначимо, що список чутливості іноді називають списком стимулів.

<= називається неблокувальним присвоєнням; поки що вважайте його звичайним присвоєнням =. Ми повернемося до різниці між ними в підрозділі 4.5.4. Зауважте, що всередині оператора *always* неблокувальне присвоєння <= використовується замість *assign*.

Як побачимо в наступних розділах, оператори *always* можна використовувати для створення тригерів, засувок або комбінаційної логіки залежно від списку чутливості та оператора. Через подібну гнучкість мови в процесі синтезу апаратних блоків можна ненавмисно отримати небажану конфігурацію. Щоб уникнути таких помилок у *SystemVerilog* додані оператори *always_ff, always_latch* та *always_comb*. Оператор *always_ff* поводиться так само, як *always*, але використовується тільки тоді, коли мається на увазі синтез тригерів, і дає змогу інструментальному середовищу в іншому випадку видавати попередження.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity flop is
    port(clk: in STD_LOGIC;
    d: in STD_LOGIC_VECTOR(3 downto 0);
    q: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of flop is
begin
        process(clk) begin
            if rising_edge(clk) then
              q <= d;
           end if;
      end process;
end;
```

Оператор *process* мови *VHDL* має такий вигляд:

```
process(sensitivity list) begin
    statement;
end process;
```

Оператор виконується, коли змінюється якась із змінних зі списку чутливості. У цьому прикладі оператор *if* перевіряє, чи була зміна переднім фронтом тактового сигналу (такту) *clk*. Якщо так, то *q* <= *d* (читається «*q* набуває значення *d*»). Отже, тригер копіює *d* в *q* по передньому фронту сигналу *clk*, а потім значення *q* залишається незмінним.

Інший варіант ідіоми *VHDL* для запису тригера:

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

rising_edge(clk) is synonymous with clk'event and clk = '1'.

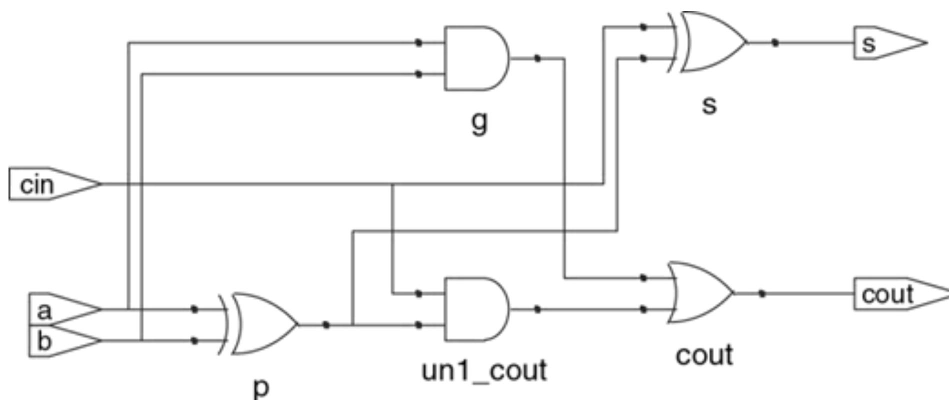


Рисунок 4.14 – Синтезована схема модуля *flop*

4.4.2 Регістри зі скиданням

На початку симуляції або відразу після подання живлення на схему значення на виході тригерів або регістрів не відомі, що позначається як значення *x* у *SystemVerilog* або як *u* в *VHDL*. На практиці корисно використовувати регістри зі скиданням, щоб після увімкнення можна було привести систему до певного стану. Скидання може бути синхронним або асинхронним. Пам'ятайте, що асинхронне скидання відбувається негайно, на відміну від синхронного, який скидає вихідний сигнал лише по наступному передньому фронту такту. У прикладі 4.18 показані ідіоми для тригерів з асинхронним і синхронним скиданням. Майте на увазі, що розрізнити

синхронне та асинхронне скидання на принциповій схемі може бути непросто. Наприклад, *Synplify Premier* розміщує на схемах асинхронне скидання на нижньому боці тригера, а синхронне – на лівому.

Приклад 4.18.
Регістр зі скиданням

SystemVerilog

```
module flopr(input logic clk,
                  input logic reset,
                  input logic [3:0] d,
                  output logic [3:0] q);
// asynchronous reset
always_ff @(posedge clk,  posedge reset)
 if (reset) q <= 4'b0;
 else q <= d;
 endmodule
module flopr(input logic clk,
                 input logic reset,
                 input logic [3:0] d,
                output logic [3:0] q);
// synchronous reset always_ff @(posedge clk)
if (reset) q <= 4'b0;
else q <= d;
endmodule
```

Сигнали в списку чутливості оператора *always* розділяються комою або словом *or*. Зауважте, що тригер з асинхронним скиданням у списку чутливості *posedge reset* є, а тригер із синхронним скиданням цього сигналу немає. Тому тригер з асинхронним скиданням реагує на передній фронт *reset* негайно, а з синхронним – тільки на передньому фронті такту.

У прикладі в обох модулях одне й те саме ім'я *flopr*, тому в схемі можна використовувати один модуль або інший.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
    port(clk, reset: in STD_LOGIC;
```

```
        d: in STD_LOGIC_VECTOR(3 downto 0);
        q: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
 begin
process(clk, reset) begin
   if reset then
      q <= "0000";
  elsif rising_edge(clk) then
     q <= d;
  end if;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity flopr is
   architecture port(clk, reset: in STD_LOGIC;
     d: in STD_LOGIC_VECTOR(3 downto 0);
     q: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synchronous of flopr is
begin
    process(clk) begin
     if rising_edge(clk) then
        if reset then q <= “0000”;
        else q <= d;
        end if;
   end if;
  end process;
 end;
```

Сигнали в списку чутливості оператора *process* розділяються комою. Зауважте, що в тригера з асинхронним скиданням у списку чутливості *reset* є, а в тригера з синхронним скиданням – немає. Тому тригер з асинхронним скиданням реагує на передній фронт *reset* негайно, а з синхронним лише по передньому фронту такту.

Пам'ятайте, що стан тригера ініціалізується як *u* внаслідок старту симуляції *VHDL*. Як згадувалося, ім'я архітектури (у цьому прикладі *synchronous* чи *asynchronous*) ігнорується інструментальним середовищем, але допомагає людям, які читають код.

Оскільки обидві архітектури описують той самий об'єкт *flopr*, у схемі можна використовувати або одну архітектуру, або іншу.

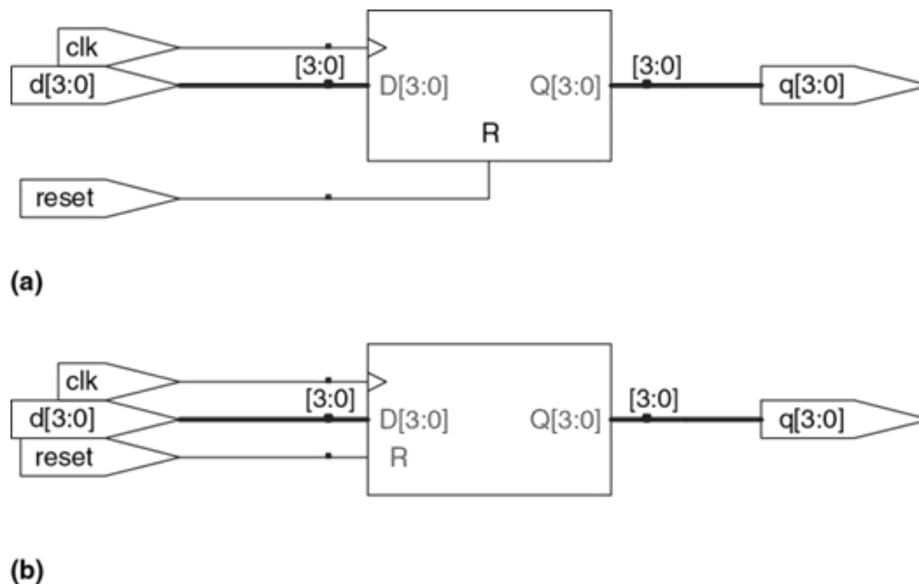


Рисунок 4.15 – Синтезована схема модуля *flopr*:
а – з асинхронним скиданням; б – із синхронним скиданням

4.4.3 Регістри із сигналом дозволу

Регістри із сигналом дозволу реагують на тактовий імпульс лише за умови подання активного рівня на лінію дозволу.

У прикладі 4.19 подано регістр з умовою *en* та асинхронним скиданням *reset,* що зберігає попереднє значення, якщо обидва сигнали мають значення *FALSE*.

Приклад 4.19.
Регістр з умовою та скиданням

SystemVerilog

```
module flopenr(input logic clk,
               input logic reset,
               input logic en;
               input logic [3:0] d;
               output logic [3:0] q);
```

```
// asynchronous reset
always_ff @(posedge clk, posedge reset)
 if (reset) q <= 4'b0;
else if (en) q <= d;
 endmodule
```
VHDL
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is
                port(clk,
                reset,
                en: in STD_LOGIC;
                d: in STD_LOGIC_VECTOR(3 downto 0);
                q: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
— asynchronous reset begin
process(clk, reset) begin
    if reset then
       q <= "0000";
    elsif rising_edge(clk) then
      if en then
     q <= d;
     end if;
   end if;
 end process;
end;
```

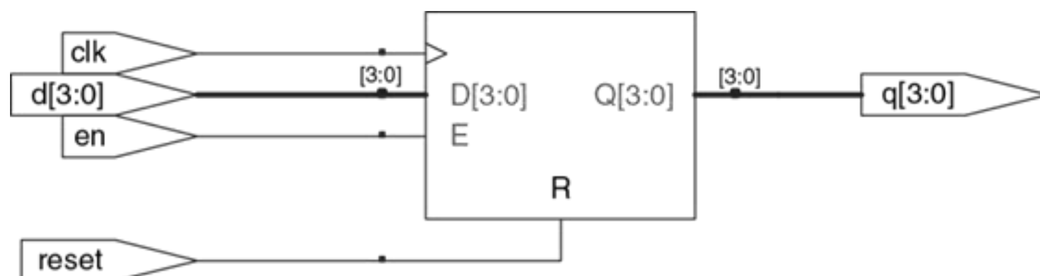


Рисунок 4.16 – Синтезована схема модуля *flopenr*

4.4.4 Групи регістрів

Один оператор *always/process* можна використовувати для опису кількох елементів апаратури. Розглянемо, наприклад, синхронізатор (п. 3.5.5), що містить два послідовні тригери (див. рис. 4.17 та приклад 4.20). По передньому фронту *clk d* копіюється в *n1* та водночас *n1* копіюється в *q*.

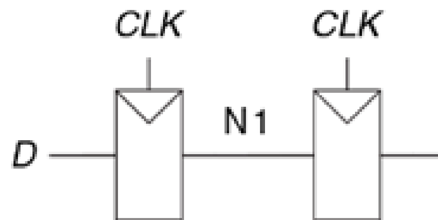


Рисунок 4.17 – Схема синхронізатора

Приклад 4.20.
Синхронізатор

SystemVerilog

```
module sync(input logic clk,
            input logic d;
            output logic q);
logic n1;
always_ff @(posedge clk)
  begin
     n1 <= d; // nonblocking
     q <= n1; // nonblocking
   end
 endmodule
```

Зверніть увагу на конструкцію *begin/end.* Вони обрамляють групу з декількох операторів, розташованих усередині оператора *always*, як дужки {} в *C* або *Java*. Конструкція *begin/end* була потрібна в прикладі *flopr*, оскільки *if/else* вважається одним оператором.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sync is
  port(clk: in STD_LOGIC;
  d: in STD_LOGIC;
  q: out STD_LOGIC);
```

```
end;
architecture good of sync is
 signal n1: STD_LOGIC;
begin
   process(clk) begin
     if rising_edge(clk) then
         n1 <= d;
         q <= n1;
    end if;
   end process;
 end;
```

Змінна *n1* має бути декларована як *signal*, тому що вона використовується всередині модуля як сигнал для з'єднання логічних елементів.

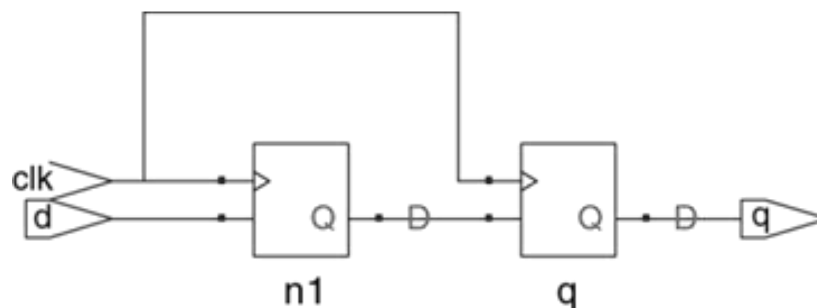


Рисунок 4.18 – Синтезована схема модуля *sync*

4.4.5 Засувки

Повертаючись до п. 3.2.2, пригадаємо, що *D*-засувка відкрита за умови високого рівня тактового сигналу, тобто пропускає сигнал даних із входу на вихід. Засувка закривається, коли рівень стає низьким, зберігаючи своє значення. Фрагмент коду в прикладі 4.21 показує ідіому для засувки *D*.

Не всі програми-синтезатори добре справляються із засувками. Якщо не впевнені, що ваш синтезатор їх підтримує, чи немає особливих причин застосовувати саме засувки, користуйтеся замість них тригерами, що працюють по фронту сигналу. Також потрібно стежити, щоб у коді на *HDL* не було конструкцій, що спричиняють появу небажаних засувок.

Чимало програм синтезу попереджають, коли створюють засувку, і якщо ви на неї не чекали, то шукайте помилку у своєму коді. А якщо не знаєте, чи потрібна вам у схемі засувка, чи ні, то це швидше за все означає, що ви пишете на *HDL*, як звичайною мовою програмування, і у вас попереду можуть бути значні проблеми [1–7].

Приклад 4.21. *D*-засувка

SystemVerilog

```
module latch(input logic clk,
             input logic [3:0] d;
             output logic [3:0] q);

always_latch
  if (clk) q <= d;
endmodule
```

У цьому разі *always_latch is* еквівалентно *always @(clk, d)* і оптимально для опису засувки на *SystemVerilog*. Оператор *always_latch* обчислюється за умови кожної зміни *clk* або *d.*

Якщо рівень *clk*, змінна *q* набуває значення *d*, тобто цей код описує засувку, активну за високим рівнем. Інакше *q* зберігає своє значення. *SystemVerilog* може видавати попередження, якщо оператор *always_latch* не описує реальну засувку.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity latch is
   port(clk: in STD_LOGIC;
        d: in STD_LOGIC_VECTOR(3 downto 0);
        q: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of latch is
  begin
    process(clk, d) begin
      if clk = ‘1’ then
        q <= d;
      end if;
    end process;
end;
```

У списку чутливості є *clk* та *d*, так що *process* обчислюється щоразу, коли *clk* або *d* змінюються. За умови високого рівня *clk* змінна *q* набуває значення *d*.

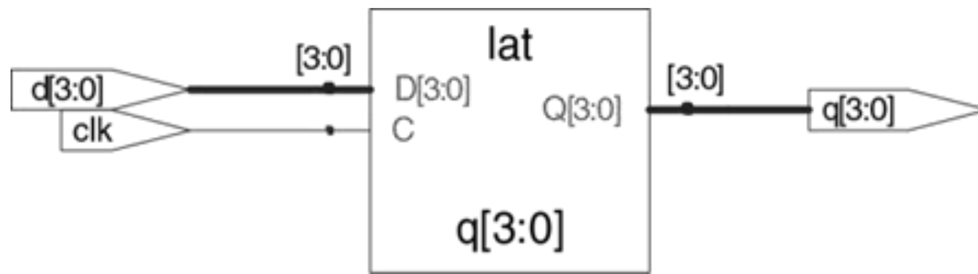


Рисунок 4.19 – Синтезована схема модуля *latch*

4.5 І знову комбінаційна логіка

У підрозділі 4.2 ми застосовували оператори присвоєння для поведінкового опису комбінаційної логіки. Оператори *always* мови *SystemVerilog* та оператори *process* мови *VHDL* використовуються для опису послідовних схем, тому що вони зберігають стан змінних, якщо не було вказано їх змінити. Однак ці оператори можна застосовувати й для поведінкового опису комбінаційної логіки, якщо список чутливості написано так, щоб відповідати на будь-яку зміну вхідних сигналів, і тіло оператора визначає значення вихідного сигналу за будь-якої комбінації значень входів. Код *HDL* у прикладі 4.22 використовує оператори *always/process* для опису групи з чотирьох інверторів (синтезовану схему див. на рис. 4.3).

Приклад 4.22.
Інвертор з використанням always/process

SystemVerilog
```
module inv(input logic [3:0] a,
           output logic[3:0] y);
    always_comb
     y = ~a;
endmodule
```

Оператор *always_comb* виконує вирази всередині оператора *always* щоразу, коли змінюється будь-який із сигналів у правій частині <= або = оператора *always*. Це еквівалентно *always @(a)*, але набагато надійніше, оскільки дає змогу уникати помилок у разі перейменування чи додавання сигналів в оператор *always*.

Якщо код всередині оператора *always_comb* не є комбінаційною логікою, тоді *SystemVerilog* видаватиме попередження. Оператор *always_comb* еквівалентний *always @(*),* але є кращим в *SystemVerilog.* Рівність = в операторі

always називається блокувальним присвоєнням, на відміну від неблокувального присвоєння <=. У *SystemVerilog* гарною практикою є використання блокувальних присвоєнь для комбінаційної логіки та неблокувальних – для послідовної. Це обговорюватиметься в п. 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
port(a: in STD_LOGIC_VECTOR(3 downto 0);
     y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture proc of inv is
begin
process(all) begin
  y <= not a;
end process;
end;
```

Оператор *process(all)* виконує всі вирази всередині *process*, як тільки змінюється будь-який із сигналів оператора *process.* Це еквівалентно *process(a),* але значно краще, оскільки допомагає уникнути помилок у разі перейменування або додавання нових сигналів.

Оператори *begin* і *end process* обов'язкові для *VHDL*, навіть якщо *process* містить лише одне присвоєння.

В обох мовах можна використовувати блокувальні та неблокувальні присвоєння в операторах *always/process*. Усередині одного оператора блокувальні присвоювання виконуються в тому порядку, в якому вони написані, точно як у звичайній мові програмування, а оновлення значень змінних у лівій частині неблокувальних присвоєнь виконується «одночасно» після того, як обчислені значення всіх правих частин неблокувальних присвоєнь.

Код у прикладі 4.23 описує повний суматор, у якому використані проміжні сигнали *p* і *g* для обчислення *s cout*. Унаслідок маємо ту саму схему, що й на рис. 4.8, але з використанням операторів *always/process* замість операторів присвоєння.

Ці два приклади не дуже вдалі для демонстрації використання *always/process* для комбінаційної логіки – у них більше рядків коду, ніж у еквівалентних *HDL*-прикладах 4.2 і 4.7 з використанням операторів

присвоєння. Однак для моделювання складнішої комбінаційної логіки зручно користуватися операторами *case* та *if*, які допускаються лише всередині операторів *always/process*. Їх розглянемо у наступних пунктах.

SystemVerilog

В операторі *always* знак = означає блокувальне присвоєння, а <= – неблокувальне (також відоме як одночасне) присвоєння. Не плутайте ці два присвоєння з неперервним присвоєння за допомогою оператора *assign.* Оператори *assign* мають використовуватися поза операторами *always* і також обчислюються одночасно.

VHDL

В операторі *process* := означає блокувальне присвоювання, а <= означає неблокувальне (або одночасне) присвоєння.

Неблокувальні присвоєння застосовуються до виходів і сигналів. Блокувальні присвоєння застосовуються до змінних, оголошених в операторах *process* (див. код у прикладі 4.23). Символ <= може використовуватися й поза операторами *process*, де також виконується одночасно.

Приклад 4.23.

Повний суматор за допомогою оператора always/process

SystemVerilog

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
  logic p, g;
  always_comb
   begin
     p = a^b; // blocking
     g = a&b; // blocking
     s = p^cin; // blocking
     cout = g | (p & cin); // blocking
  end
 endmodule
```

Тут еквівалентом *always_comb* було б *always @(a, b, cin),* але *always_comb* краще, оскільки дає змогу уникнути помилок, пов'язаних із відсутніми в списку чутливості сигналів.

З причин, які обговоримо в п. 4.5.4, для комбінаційної логіки краще використовувати блокувальні присвоєння. У цьому прикладі вони застосовані для обчислення спочатку *p,* потім *g, s* та *cout.*

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity fulladder is
     port(a, b, cin: in STD_LOGIC;
            s, cout: out STD_LOGIC);
end;
architecture synth of fulladder is
 begin
  process(all)
  variable p, g: STD_LOGIC;
    begin
      p := a xor b; — blocking
      g := a and b; — blocking
      s <= p xor cin;
      cout <= g or (p and cin);
  end process;
end;
```

Тут еквівалентом оператора *process(all)* був би *process(a, b, cin),* але *process(all)* краще, оскільки допомагає уникнути помилок, пов’язаних із відсутніми в списку чутливості сигналів.

З причин, які обговоримо в п. 4.5.4, для проміжних змінних у комбінаційній логіці краще використовувати блокувальні присвоєння. У цьому прикладі вони застосовані для обчислення нових значень *p* і *g*, необхідних для подальшого обчислення *s* і *cout*.

Так як *p* і *g* згадуються в лівій частині операторів блокувального присвоєння (:=) в операторі *process*, то вони мають бути оголошені як *variable*, а не як *signal*. Оголошення змінних пишеться до початку того процесу, в якому ці змінні використовуються.

4.5.1 Оператори case

Ось більш вдалий приклад використання операторів *always/process* для комбінаційної логіки – дешифратор для семисегментного індикатора,

виконаний із застосуванням оператора *case,* що можна писати лише всередині оператора *always/process*.

Як ви могли помітити з прикладу 2.10 дешифратора семисегментного індикатора, процес розроблення великих блоків комбінаційної логіки загрожує помилками. Мови опису апаратури полегшують цей процес, даючи змогу визначати функціональність на більш високому рівні абстракції, і потім автоматично синтезувати її у вентилі. У коді прикладу 4.24 використовується оператор *case* для опису дешифратора семисегментного індикатора таблиці істинності. Оператор *case* виконує різні дії залежно від його вхідних даних. Він синтезується в комбінаційну логіку, якщо всі можливі поєднання вхідних даних визначено; інакше вийде послідовна логіка, оскільки вихід збереже своє попереднє значення в невизначених випадках.

Приклад 4.24.
Дешифратор семисегментного індикатора

SystemVerilog

```
module sevenseg(input logic [3:0] data,
                 output logic [6:0] segments);
always_comb
  case(data)
//        abc_defg
0: segments = 7'b111_1110;
1: segments = 7'b011_0000;
2: segments = 7'b110_1101;
3: segments = 7'b111_1001;
4: segments = 7'b011_0011;
5: segments = 7'b101_1011;
6: segments = 7'b101_1111;
7: segments = 7'b111_0000;
8: segments = 7'b111_1111;
9: segments = 7'b111_0011;
default: segments = 7'b000_0000;
  endcase
 endmodule
```

Оператор *case* перевіряє значення *data*; якщо *data* дорівнює 0, виконається дія після двокрапки, тобто встановлення *segments* в 1111110.

Аналогічно перевіряються інші значення *data* аж до 9 (зверніть увагу, що за замовчуванням система десяткова). Умова *default* (за замовчуванням) – зручний спосіб визначити вихід для всіх випадків, не згаданих явно, цим гарантуючи в результаті комбінаційну логіку. У *SystemVerilog* оператори *case* мають перебувати всередині операторів *always*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity seven_seg_decoder is
     port(data: in STD_LOGIC_VECTOR(3 downto 0);
          segments: out STD_LOGIC_VECTOR(6 downto 0));
end;
architecture synth of seven_seg_decoder is
begin
     process(all) begin
         case data is
—          abcdefg
when  X"0" => segments <= "1111110";
when  X"1" => segments <= "0110000";
when  X"2" => segments <= "1101101";
when  X"3" => segments <= "1111001";
when  X"4" => segments <= "0110011";
when  X"5" => segments <= "1011011";
when  X"6" => segments <= "1011111";
when  X"7" => segments <= "1110000";
when  X"8" => segments <= "1111111";
when  X"9" => segments <= "1110011";
when others => segments <= "0000000";
   end case;
  end process;
 end;
```

Оператор *case* перевіряє значення *data*; якщо *data* дорівнює 0, виконається дія після =>, тобто встановлення *segments* в 1111110. Аналогічно перевіряються інші значення *data* аж до 9 (зверніть увагу на використання *X* для позначення шістнадцяткових чисел). Умова *others* (інші) – зручний спосіб визначити вихід для всіх випадків, не згаданих явно, цим гарантуючи в результаті комбінаційну логіку.

На відміну від *SystemVerilog,* у *VHDL* допускаються присвоєння сигналів із вибором (див. приклад 4.6), які схожі на оператори *case,* але можуть мати місце й за межами операторів *process*, так що приводів використовувати оператори *process* для опису комбінаційної логіки *VHDL* менше.

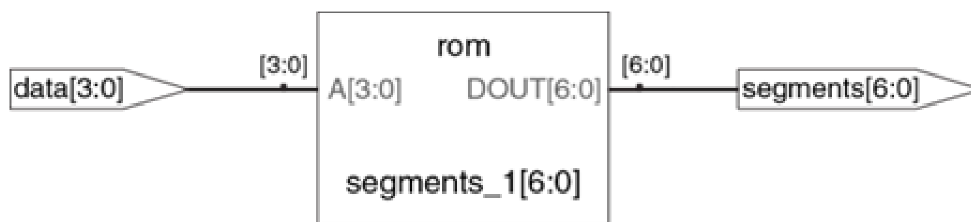


Рисунок 4.20 – Синтезована схема модуля *seven_seg_*

Synplify Premier синтезує дешифратор семисегментного індикатора як постійну пам'ять (ПЗП), що містить сім виходів кожної з 16 можливих комбінацій входів. ПЗП обговорюються у п. 5.5.6.

Якби умова *default* або *others* не була згадана в операторі *case*, то дешифратор зберігав би попереднє значення виходу, коли вхід розташований у діапазоні 10–15. Для апаратури така поведінка була б дивною.

Звичайні дешифратори також часто записуються за допомогою операторів *case*. У коді прикладу 4.25 подано дешифратор 3:8.

Приклад 4.25.
Дешифратор 3:8

SystemVerilog
```
module decoder3_8(input logic [2:0] a,
                  output logic [7:0] y);
   always_comb
     case(a)
         3'b000: y = 8'b00000001;
         3'b001: y = 8'b00000010;
         3'b010: y = 8'b00000100;
         3'b011: y = 8'b00001000;
         3'b100: y = 8'b00010000;
         3'b101: y = 8'b00100000;
         3'b110: y = 8'b01000000;
         3'b111: y = 8'b10000000;
         default: y = 8'bxxxxxxxx;
     endcase
endmodule
```

Необхідно зауважити, що умова *default* у цьому разі не потрібна для синтезу, оскільки перелічені всі можливі поєднання входів, але вона корисна для симуляції, якщо якийсь із входів дорівнює *x* або *z*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder3_8 is
  port(a: in STD_LOGIC_VECTOR(2 downto 0);
         y: out STD_LOGIC_VECTOR(7 downto 0));
  end;
architecture synth of decoder3_8 is
begin
process(all) begin
   case a is
     when "000" => y <= "00000001";
     when "001" => y <= "00000010";
     when "010" => y <= "00000100";
     when "011" => y <= "00001000";
     when  "100" => y <= "00010000";
     when "101" => y <= "00100000";
     when  "110" => y <= "01000000";
     when "111" => y <= "10000000";
     when others => y <= "XXXXXXXX";
  end case;
 end process;
end;
```

Зазначимо, що умова *others* у цьому разі не потрібна для синтезу, оскільки перелічені всі можливі поєднання входів, але вона корисна для симуляції, якщо якийсь із входів дорівнює *x, z* або *u*.

4.5.2 Оператори if

Оператори *always/process* можуть містити також оператори ***if***, за якими може йти оператор *else.* Якщо всі можливі поєднання входів оброблені, оператор синтезується в комбінаційну логіку, інакше – в послідовну (наприклад, засувка в п. 4.4.5).

У прикладі 4.26 використовуються оператори *if* для опису схеми пріоритетів, визначеної в підрозділі 2.4. Згадаємо, що *N*-вхідна схема

пріоритетів встановлює значення *TRUE* тому виходу, що відповідає найбільш пріоритетному входу, який дорівнює *TRUE*.

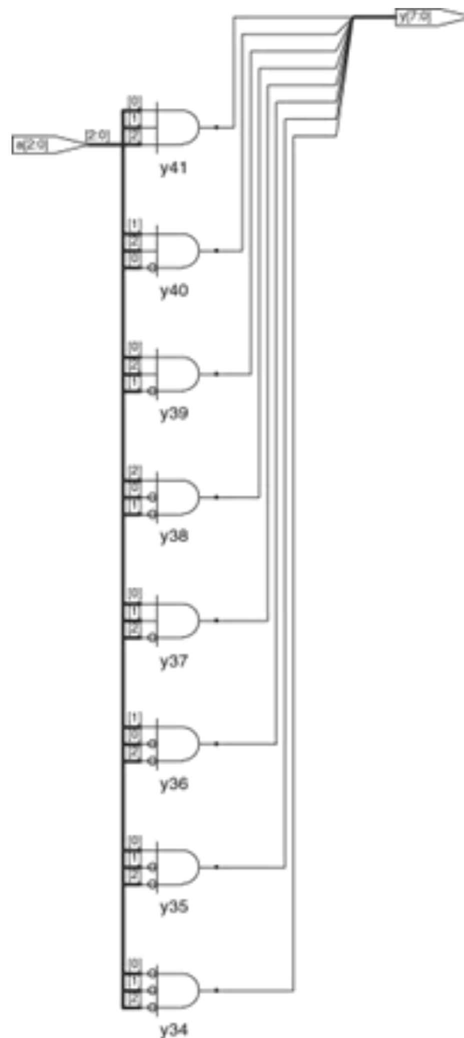


Рисунок 4.21 – Синтезована схема модуля *decoder3*

Приклад 4.26.
Схема пріоритетів

SystemVerilog

```
module priorityckt(input logic [3:0] a,
                   output logic [3:0] y);
always_comb
    if (a[3]) y <= 4’b1000;
    else if (a[2]) y <= 4’b0100;
    else if (a[1]) y <= 4’b0010;
    else if (a[0]) y <= 4’b0001;
    else y <= 4’b0000;
endmodule
```

У *SystemVerilog* оператори *if* мають бути всередині операторів *always*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity priorityckt is
    port(a: in STD_LOGIC_VECTOR(3 downto 0);
          y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of priorityckt is
begin
  process(all) begin
    if a(3) then y <= "1000";
    elsifa(2) then y <= "0100";
    elsifa(1) then y <= "0010";
    elsifa(0) then y <= "0001";
    else y <="0000";
    end if;
 end process;
 end;
```

На відміну від *SystemVerilog*, у *VHDL* є оператори умовного присвоєння (див. приклад 4.6), схожі на оператори *if*, але вони можуть мати місце й за межами операторів *process*, так що приводів використовувати процеси для опису комбінаційної логіки *VHDL* менше.

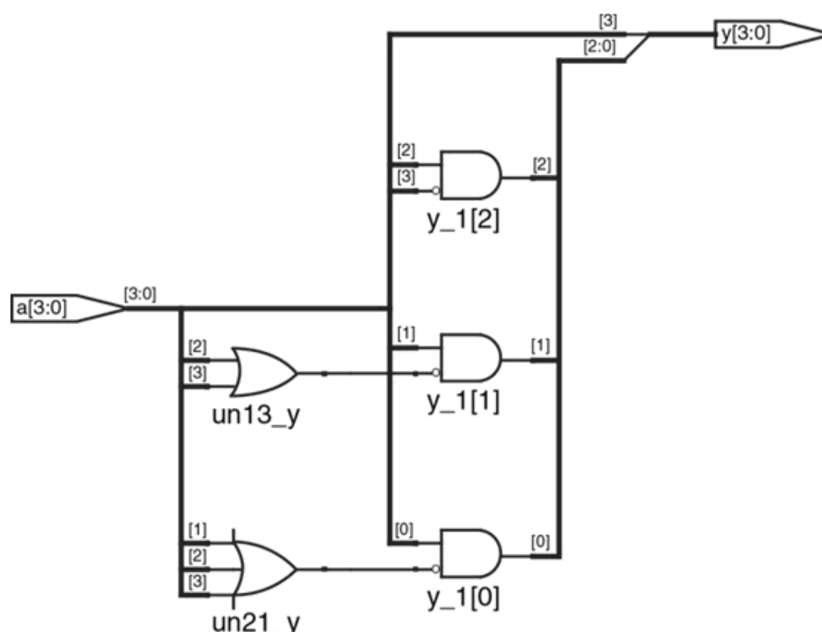


Рисунок 4.22 – Синтезована схема модуля *priorityckt*

4.5.3 Таблиці істинності з невизначеними бітами

Як показано в п. 2.7.3, у таблицях істинності можуть бути невизначені біти для спрощення логіки. У коді прикладу 4.27 показано, як описати пріоритетну схему з невизначеними бітами.

Synplify Premier синтезує схему цього модуля (див. рис. 4.23), що незначно різниться від схеми пріоритетів на рис. 4.24, але вони логічно еквівалентні.

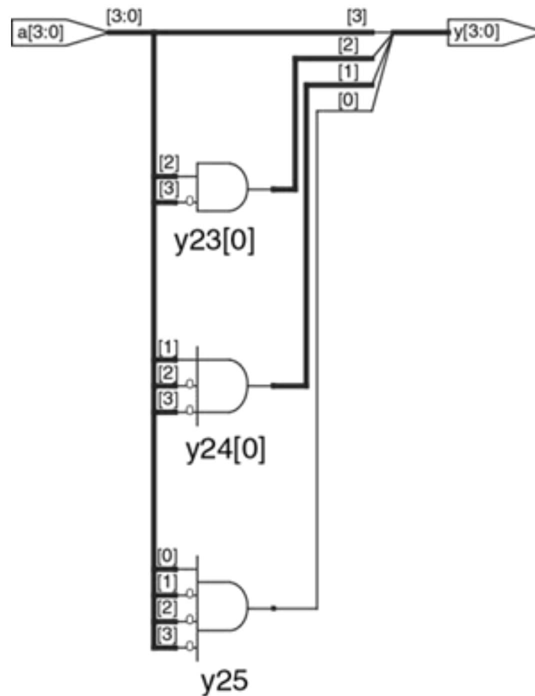


Рисунок 4.23 – Синтезована схема модуля *priority_casez*

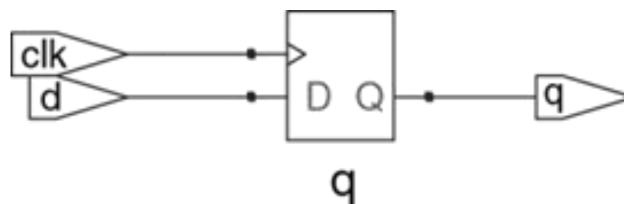


Рисунок 4.24 – Синтезована схема для *syncbad*

Щодо поданого рисунка можемо зауважити:
для моделювання послідовної логіки в операторах always/process
необхідно користуватися лише неблокувальними присвоєннями.
За допомогою різних хитрощів, наприклад зміни порядку присвоєнь,
можна домогтися правильної роботи блокувальних присвоєнь,
але вони не дають жодних переваг, а лише додають ризик небажаної поведінки.
Деякі послідовні схеми не працюватимуть з використанням блокувальних
присвоєнь незалежно від їх порядку. (Автори пропонують взяти на віру,
що не варто застосовувати в SystemVerilog блокувальне присвоєння
для послідовної логіки, навіть якщо воно в операторі always єдине.
Це є особливостями алгоритмів симуляції SystemVerilog,
у їх подробиці ми не вдаватимемося).

Приклад 4.27.
Схема пріоритетів з незначними бітами

SystemVerilog
```
module priority_casez(input logic [3:0] a,
                      output logic [3:0] y);
always_comb
  casez(a)
    4’b1???: y <= 4’b1000;
    4’b01??: y <= 4’b0100;
    4’b001?: y <= 4’b0010;
    4’b0001: y <= 4’b0001;
    default: y <= 4’b0000;
  endcase
endmodule
```

Оператор *casez* працює так само, як і *case*, але ще й розпізнає знак «?» як незначний біт.

VHDL
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity priority_casez is
   port(a: in STD_LOGIC_VECTOR(3 downto 0);
        y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture dontcare of priority_casez is
begin
  process(all) begin
    casez a is
     when "1---" => y <= "1000";
     when "01--" => y <= "0100";
     when "001-" => y <= "0010";
     when "0001" => y <= "0001";
     when others   =>y <= "0000";
    end casez;
 end process;
 end;
```

Оператор *casez* працює так само, як і *case*, але ще й розпізнає знак «–» як невизначений біт.

4.5.4 Блокувальні та неблокувальні присвоєння

Нижче подано керівництво, коли та як використовувати той чи той тип присвоєння. Якщо його не дотримуватися, то можна написати код, який, можливо, працюватиме в режимі симуляції, але синтезуватиметься в некоректну схему.

Стисле керівництво
щодо блокувальних і неблокувальних присвоєнь

SystemVerilog

1. Використовуйте *always_ff @(posedge clk)* та неблокувальні присвоєння для моделювання послідовної логіки.

```
always_ff @(posedge clk)
begin
  n1 <= d; // неблокуючий
  q <= n1; // неблокуюче
end
```

2. Використовуйте безперервні присвоєння для моделювання простої комбінаційної логіки.

```
assign y = s? d1: d0;
```

3. Застосовуйте *always_comb* та блокувальні присвоєння для моделювання більш складної комбінаційної логіки, коли зручніше використовувати оператор *always*.

```
always_comb begin
p = a^b; // блокує
 g = a & b; // блокує
 s = p ^ cin;
cout = g | (p & cin);
 end
```

4. Не надавайте значення одному і тому самому сигналу в різних операторах *always* або в операторах безперервного присвоєння.

VHDL

1. Використовуйте *process(clk)* та неблокувальні присвоєння для моделювання синхронної послідовної логіки.

```
process(clk) begin
if rising_edge(clk) then
```

```
n1 <= d; — nonblocking
q <= n1; — nonblocking
end if;
end process;
```

2. Використовуйте одночасні присвоєння поза операторами *process* для моделювання простої комбінаційної логіки.

```
y <= d0 when s = '0' else d1;
```

3. Застосовуйте *process(all)* для моделювання складнішої комбінаційної логіки, якщо оператор *process* більш зручний. Використовуйте блокувальні присвоєння локальних змінних.

```
process(all)
variable p, g: STD_LOGIC;
begin
p := a xor b; — блокуюче
 g := a and b; — блокуюче
 s <= p xor cin;
cout <= g or (p and cin);
end process;
```

4. Не надавайте значення одній і тій самій змінній у різних операторах *process* чи в операторах одночасних присвоєнь.

Комбінаційна логіка*

Повний суматор у коді прикладу 4.23 коректно змодельовано з використанням блокувальних присвоєнь. У цьому підрозділі розглянемо, як він працює і чим він розрізняється від моделі, що застосовує неблокувальні присвоєння.

Уявіть, що значення *a, b* і *cin* спочатку дорівнюють 0. Значення *p, g, s* і *cout* також дорівнюватимуть 0. У якийсь момент *a* змінюється на 1, активуючи оператор *always/process*. Чотири блокувальні присвоєння виконуються в наведеному нижче порядку. (У разі *VHDL* присвоєння *s* і *cout* виконуються одночасно). Зауважте, що *p* і *g* набувають своїх нових значень до *s* і *cout* завдяки блокувальним присвоєнням. Це важливо, оскільки ми хочемо обчислювати *s* і *cout*, використовуючи нові значення *p* і *g*.

1. **p ← 1^ 0 = 1**

2. g ← 1 • 0 = 0

3. s ← 1 ^ 0 = 1

4. cout ← 0 + 1 • 0 = 0

Приклад 4.28 ілюструє використання неблокувальних присвоєнь.

Розглянемо той самий випадок, коли *a* з 0 стає 1, тоді як *b* і *cin* дорівнюють 0. Чотири неблокувальні присвоєння виконуються одночасно:

p ← 1 ^ 0 = 1

g ← 1 • 0 = 0

s ← 0 ^ 0 = 0

cout ← 0 + 0 • 0 = 0

Приклад 4.28.

Повний суматор з неблокувальним присвоєнням

SystemVerilog

```
// nonblocking assignments (no recommended)
module fulladder(input logic a, b, cin,
                  output logic s, cout);
logic p, g;
always_comb
   begin
     p<=a^b; // nonblocking
     g <= a & b; // nonblocking
     s <= p ^ cin;
     cout <= g | (p & cin);
   end
endmodule;
```

VHDL

```
-- nonblocking assignments (not recommended)
 library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fulladder is
    port(a, b, cin: in STD_LOGIC;
         s, cout: out STD_LOGIC);
end;
architecture noblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
```

```
process(all) begin
    p <= a xor b; — nonblocking
    g <= a and b; — nonblocking
    s <= p xor cin;
    cout <= g or (p and cin);
 end process;
end;
```

Оскільки *p* і *g* згадуються в лівій частині неблокувальних присвоєнь в операторі *process*, вони мають бути оголошені як *signal*, а не як *variable*. Оголошення сигналів записуються перед *begin* в *architecture*, а не в *process*.

Отже, *s* обчислюється одночасно з *p*, тому використовує старе значення *p*. Унаслідок цього *s* залишається рівним 0, а не стає 1. Однак *p* змінюється з 0 на 1. Ця зміна викликає виконання оператора *always/process* вдруге:

p ← 1 ^ 0 = 1
g ← 1 • 0 = 0
s ← 1 ^ 0 = 1
cout ← 0 + 1 • 0 = 0

Цього разу *p* вже дорівнює 1, і *s*, як необхідно, стає рівним 1. Неблокувальні присвоєння зрештою приходять до правильної відповіді, але оператору *always/process* доводиться виконуватися двічі. Від цього симуляція відбувається повільніше, хоча код синтезується в ту саму схему.

Ще один недолік непридатних для моделювання комбінаційної логіки – під час симуляції може вийти неправильний результат, якщо забути згадати проміжні змінні в списку чутливості (у використанні *always_comb* і *process(all)* для комбінаційної логіки цей недолік не актуальний).

Найгірше, деякі синтезатори створять правильну схему, навіть якщо неправильний список чутливості призводить до неправильної симуляції. Це призведе до розбіжності результатів симуляції та реальної поведінки апаратури.

SystemVerilog

Якби список чутливості оператора *always* у коді прикладу 4.28 був написаний як *always@(a, b, cin),* а не як *always_comb*, оператор не виконався би повторно, коли змінюється *p* або *g*. У цьому разі *s* хибно залишився б рівним 0 замість 1.

VHDL

Якби список чутливості оператора *process* у коді прикладу 4.28 був записаний як *process(a, b, cin),* а не як *process(all),* оператор не виконався б повторно, коли змінюється *p* або *g*. У цьому разі *s* хибно залишився б рівним 0 замість 1.

Послідовна логіка*

Синхронізатор коду з прикладу 4.20 коректно змодельовано з використанням неблокувальних присвоєнь. По передньому фронту тактового сигналу *d* копіюється в *n1* тоді, як *n1* копіюється в *q*, так що код, як це необхідно, описує два регістри. Наприклад, нехай спочатку $d = 0$, $n1 = 1$ та $q = 0$. По передньому фронту тактового сигналу одночасно виконуються два присвоєння, так що після проходження фронту $n1 = 0$ і $q = 1$:

n1 ← d = 0

q ← n1 = 1

У коді з прикладу 4.29 робиться спроба описати той самий модуль за допомогою блокувальних присвоєнь. По передньому фронту *clk d* копіюється в *n1*. Потім це нове значення *n1* копіюється в *q,* у результаті значення *d* помилково виявляється і в *n1*, і в *q*. Присвоєння виконуються одне за одним, отже, після фронту сигналу $q = n1 = 0$.

1. n1 ← d = 0

2. q ← n1 = 0

Тому що змінна *n*1 не впливає на поведінку *q*, синтезатор ліквідує її в процесі оптимізації, як зображено на рис. 4.24.

Приклад 4.29.

Поганий синхронізатор з блокувальними присвоєннями

SystemVerilog

```
// Bad implementation of synchronizer using blocking // assignments
module syncbad(input logic clk,
                  input logic d;
                 output logic q);
logic n1;
always_ff @(posedge clk)
   begin
    n1 = d; // blocking
```

```
    q = n1; // blocking
   end
endmodule
```

VHDL

```
— Bad implementation of a synchronizer using blocking — assignment
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity syncbad is
  port(clk: in STD_LOGIC;
        d: in STD_LOGIC;
       q: out STD_LOGIC);
end;
architecture bad of syncbad is
 begin
   process(clk)
variable n1: STD_LOGIC;
 begin
  if rising_edge(clk) then
           n1 := d;      -- blocking
q <= n1;
 end if;
 end process;
end;
```

4.6 Кінцеві автомати

Як пам'ятаєте, кінцевий автомат (КА) має регістр стану та два блоки комбінаційної логіки для обчислення наступного стану та виходу за умови заданих у поточному стані та інформації на вході (див. рис. 3.22). Опис кінцевих автоматів на *HDL*, відповідно, містять три частини, що моделюють регістр стану, логіку наступного стану та логіку виходу.

Приклад 4.30.
Кінцевий автомат, що ділить на три

SystemVerilog

```
module divideby3FSM(input logic clk,
                    input logic reset,
```

```
                    output logic y);
typedef enum logic [1:0] {S0, S1, S2} statetype;
 statetype [1:0] state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
if (reset) state <= S0;
else state <= nextstate;
// next state logic
 always_comb
    case (state)
          S0: nextstate <= S1;
          S1: nextstate <= S2;
          S2: nextstate <= S0;
          default: nextstate <= S0;
   endcase
// output logic
assign y = (state = = S0);
endmodule
```

Оператор *typedef* визначає значення *Statetype* як двобітний логічний тип з трьома можливими значеннями: *S0, S1* або *S2*. Сигнали *state* та *nextstate* належать до типу *statetype*.

Константам переліку, згаданим у визначенні типу, за замовчуванням надаються порядкові значення: *S*0 = 00, *S*1 = 01 та *S*2 = 10. Вони можуть бути змінені користувачем, але програма-синтезатор розглядає їх як рекомендацію, а не як вимогу. Наприклад, наступний фрагмент кодує стани трибітним унарним (*1-hot*) кодом:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100} statetype;
```

Зверніть увагу на оператор *case*, який визначає функцію переходів. Через те, що логіка для наступного стану має бути комбінаційною, умова *default* (значення за замовчуванням) є обов’язковою, навіть незважаючи на те, що стану *2’b11* не буває.

Вихід *y* дорівнює 1, коли автомат перебуває в стані *S*0. Результат операції порівняння на рівність *a == b* дорівнює 1, коли *a* дорівнює *b* і 0 в іншому разі. Операція порівняння на нерівність *a != b*, навпаки, дає 1, коли *a* не дорівнює *b*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity divideby3FSM is
   port(clk, reset:
          in STD_LOGIC;
          y: out STD_LOGIC);
end;
архітектура synth of divideby3FSM is
typ statetype is (S0, S1, S2);
signal state, nextstate: statetype;
begin
--- state register
process(clk, reset) begin
    if reset then state <= S0;
   elsif rising_edge(clk) then
      state <= nextstate;
   end if;
end process;
— next state logic
nextstate <= S1 when state = S0 else
             S2 when state = S1 else
             S0;
— output logic
  y <= '1' when state = S0 else '0';
end;
```

У цьому прикладі визначається новий тип переліку даних *statetype* з трьома можливими значеннями: *S*0, *S*1 і *S*2. Сигнали *state* та *nextstate* належать до типу *statetype*. Завдяки використанню переліку, а не явно заданих кодів станів, *VHDL* дає змогу синтезатору обрати оптимальний код для станів.

Вихід *y* дорівнює 1, коли *state* дорівнює *S*0. Операція порівняння на нерівність записується як /=. Щоб отримати на виході 1, коли стан розрізняється від *S*0, замініть порівняння на *state / = S0*.

У прикладі 4.30 описується КА поділу на три (п. 3.4.2). Для ініціалізації КА використовується асинхронне скидання. Регістр стану застосовує стандартну ідіому для тригерів. Логіка формування наступного стану та виходу є комбінаційною.

Програма-синтезатор *Synplify Premier* породжує лише блокову діаграму та діаграму переходів для автомата; вона не показує логічні елементи або входи та виходи на вузлах та дугах, тому необхідно перевірити за діаграмою, чи правильно ви визначили КА в *HDL*-коді.

Діаграма переходів (рис. 4.25) для КА поділу на три аналогічна діаграмі, зображеній на рис. 3.28, *b*. Подвійний кружок означає, що автомат скидається у стан *S*0. Реалізацію автомата на рівні вентилів було подано в п. 3.4.2.

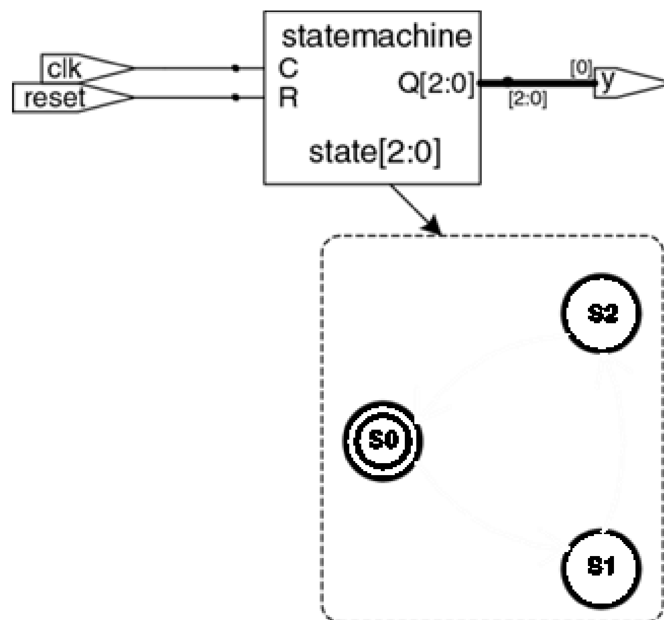


Рисунок 4.25 – Синтезована схема модуля *divideby3fsm*

Зауважте, що стани позначені константами переліку, а не бінарними значеннями. Завдяки цьому код стає більш читабельним та його легше змінювати.

Якщо з будь-якої причини ми захочемо, щоб вихід дорівнював 1 в станах *S*0 і *S*1, вихідна логіка зміниться таким чином:

```
SystemVerilog
//Вихідна логіка
assign y = (state = = S0 | state = = S1);
VHDL
— вихідна логіка
y <= '1' when (state = S0 or state = S1) else '0';
```

Наступні два приклади описують КА розпізнавача – бітового шаблону равлика з п. 3.4.3. У коді показано, як використовувати оператори *case* та *if* для оброблення наступного стану та вихідної логіки, що залежить і від входу, і від поточного стану. В автоматі Мура (приклад 4.31) вихід залежить лише

від поточного стану, а в автоматі Мілі (приклад 4.32) вихід залежить і від поточного стану, і від входів.

Приклад 4.31.
Автомат Мура для розпізнавання патерна

SystemVerilog

```
module patternMoore(input logic clk,
                    input logic reset,
                    input logic a,
                    output logic y);
typedef enum logic [1:0] {S0, S1, S2}
 statetype; statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
   if (reset) state <= S0;
   else state <= nextstate;
// next state logic
always_comb
    case (state)
       S0: if(a) nextstate = S0;
             else nextstate = S1;
      S1:if (a) nextstate = S2;
           else nextstate = S1;
     S2: if (a) nextstate = S0;
           else nextstate = S1;
    default: nextstate = S0;
endcase
// output logic
assign y = (state = = S2);
 endmodule
```

Зауважте, що неблокувальні присвоєння (<=) використовуються в регістрі стану для опису послідовної логіки, а для комбінаційної логіки наступного стану застосовуються блокувальні присвоєння (=).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity patternMoore is
```

```
  port(clk, reset: in STD_LOGIC; a:
                   in STD_LOGIC;
                   y: out STD_LOGIC);
end;
architecture synth of patternMoore is
   type statetype is (S0, S1, S2);
   signal state, nextstate: statetype;
begin
--- state register
  process(clk, reset) begin
     if reset then state <= S0;
     elsif rising_edge(clk) then state <= nextstate;
     end if;
end process;
— next state logic
   process(all) begin
     case state is
         when S0 =>
            if a then nextstate <= S0;
            else nextstate <= S1;
            end if;
         when S1 =>
            if a then nextstate <= S2;
           else nextstate <= S1;
           end if;
         when S2 =>
            if a then nextstate <= S0;
           else nextstate <= S1;
          end if;
         when others =>
                      nextstate <= S0;
    end case;
end process;
—output logic
y <= ‘1’ when state = S2 else ‘0’;
 end;
```

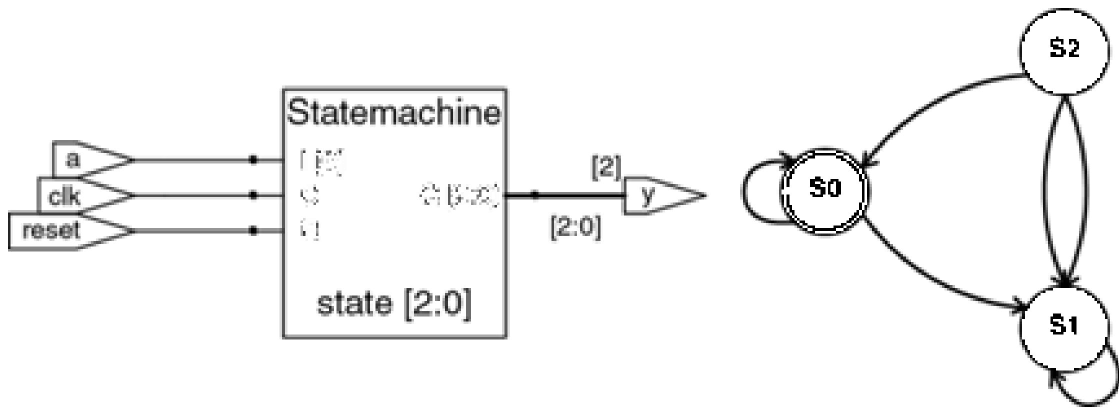


Рисунок 4.26 – Синтезована схема модуля *patternMoore*

Приклад 4.32.
Автомат Мілі для розпізнавання бітового шаблону

SystemVerilog

```
module patternMealy(input logic clk,
                    input logic reset,
                    input logic a,
                    output logic y);
typedef enum logic {S0, S1} statetype;
 statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else state <= nextstate;
// next state logic
always_comb
   case (state)
      S0: if (a) nextstate = S0;
           else nextstate = S1;
      S1: if(a) nextstate = S0;
           else nextstate = S1;
      default: nextstate = S0;
   endcase
// output logic
assign y = (a & state = = S1);
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity patternMealy is
    port(clk, reset: in STD_LOGIC;
         a: in STD_LOGIC;
         y: out STD_LOGIC);
end;
architecture synth of patternMealy is
    type statetype is (S0, S1);
    signal state, nextstate: statetype;
begin
--- state register
process(clk, reset) begin
    if reset then state <= S0;
   elsif rising_edge(clk) then state <= nextstate;
   end if;
end process;
— next state logic
 process(all) begin
   case state is
     when S0 =>
        if a then nextstate <= S0;
       else nextstate <= S1;
       end if;
    when S1 =>
        if a then nextstate <= S0;
       else nextstate <= S1;
       end if;
     when others =>
         nextstate <= S0;
end case;
end process;
— output logic
y <= ‘1’ when (a = ‘1’ and state = S1) else ‘0’;
end;
```

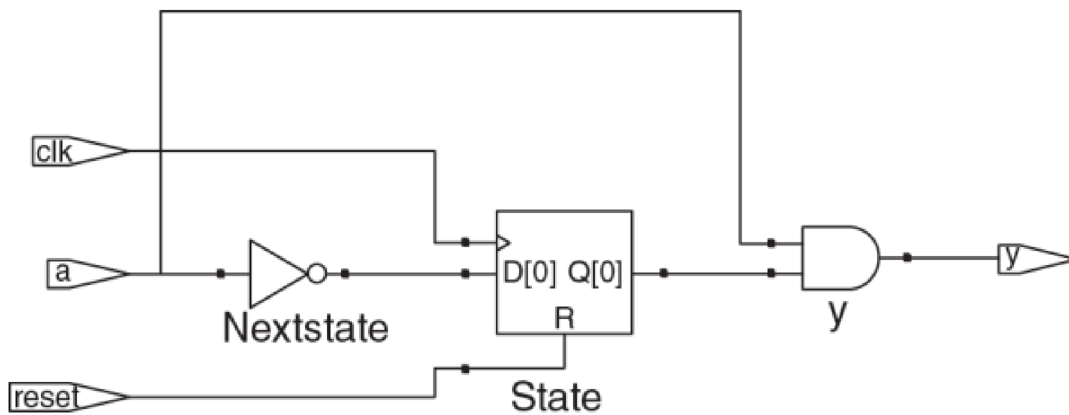


Рисунок 4.27 – Синтезована схема модуля *patternMealy*

4.7 Типи даних*

У цьому підрозділі докладніше розглянемо особливості типів даних *SystemVerilog* і *VHDL.*

4.7.1 SystemVerilog

У попередника *SystemVerilog*, мові *Verilog*, переважно використовувалися два типи: *reg* і *wire*. Незважаючи на назву, сигнал типу *reg* не має відповідати регістру, і ця плутанина ускладнювала вивчення мови. Щоб уникнути цього недаліку, до *SystemVerilog* додано тип *logic*, що й застосовується в цій книзі. У цьому підрозділі детально розповідається про типи *reg* і *wire* для тих, хто має читати старий код мовою *Verilog.*

У *Verilog,* якщо сигнал спостерігається в лівій частині оператора <= або = в *always*-блоці, він має бути оголошений як *reg*, інакше – як *wire.* Тому сигнал типу *reg* може бути виходом тригера, засувки або комбінаційної логіки, залежно від списку чутливості та оператора всередині *always*-блоку.

У вхідних і вихідних портів за замовчуванням – тип *wire*, якщо їх тип не оголошений як *reg*. Нижче показано, як тригер описується на звичайному *Verilog*. Зауважте, що сигнали *clk* і *d* належать до типу *wire* за замовчуванням, а *q* явно оголошено як *reg*, тому що він спостерігається в лівій частині оператора <= в *always*-блоці.

```
module flop(input clk,
            input [3:0] d;
            output reg [3:0] q);
always @ (posedge clk)
  q <= d;
endmodule
```

Тип *logic,* доданий в *SystemVerilog* – це синонім типу *reg*, але його назва позбавлена небажаних асоціацій із тригером. Крім того, в *SystemVerilog* ослаблені обмеження щодо використання операторів *assign* і в ієрархічних призначеннях портів, так що сигнали типу *logic* можуть бути застосовані поза блоками *always* – там, де традиційно були б потрібні сигнали типу *wire*. Отже, переважна більшість сигналів *SystemVerilog* може бути типу *logic*. Винятком є сигнал із кількома джерелами (наприклад, тристабільна високоімпедансна шина з трьома станами), який має бути оголошений як ланцюг (*net*), як подано в коді прикладу 4.10. Завдяки цьому правилу, коли сигнал типу *logic* помилково під'єнано до кількох джерел, *SystemVerilog* видає повідомлення про помилку вже під час компіляції, а не надає йому значення *x* під час симуляції.

Найбільш поширені типи ланцюгів – *wire* та *tri*. Ці два типи – синоніми, але *wire* традиційно використовується, коли джерело (*driver*) одне, а *tri* – якщо їх кілька. У *SystemVerilog* у типі *wire* немає необхідності: для сигналів з одним джерелом *logic* кращий.

Коли в усіх активних джерелах ланцюга типу *tri* – одне й те саме значення, воно набуває цього значення. Якщо всі джерела не активні, ланцюг вимкнено (*floats*) (*z*). Якщо активні джерела мають різні значення (0, 1, *x*), тоді ланцюг перебуває в стані конфлікту (*in contention*) (*x*).

Є й інші типи ланцюгів, значення яких установлюються інакше в разі неактивних джерел чи конфлікту. Ці типи використовуються нечасто, але можуть бути там само, де тип *tri* (наприклад, для ланцюгів з кількома джерелами). Вони описані в табл. 4.7.

Таблиця 4.7 – Визначення значень ланцюгів

Типи ланцюгів	**Значення в разі неактивних ланцюгів**	**Значення в разі конфліктів ланцюгів**
tri	z	x
trireg	попереднє значення	x
triand	z	0, якщо є хоча один 0
trior	z	1, якщо є хоча одна 1
tri0	0	x
tri1	1	x

4.7.2 VHDL

На відміну від *SystemVerilog*, мова *VHDL* визнається строгою типізацією, що захищає користувача від деяких помилок.

Незважаючи на те, що тип *STD_LOGIC* є принципово важливим, він не вбудований у мову *VHDL*, а є частиною бібліотеки *IEEE.STD_LOGIC_1164.* Через це в кожному файлі мають бути оператори під'єднання бібліотеки, що можна було бачити вище в прикладах.

Крім того, в *IEEE.STD_LOGIC_1164* відсутні базові операції типу складання, порівняння, зрушень та перетворення на цілі з даних типу *STD_LOGIC_VECTOR*. Їх, нарешті, додали в стандарті *VHDL* 2008 до бібліотеки *IEEE.NUMERIC_STD_UNSIGNED.*

У *VHDL* також є тип *BOOLEAN* із двома значеннями: *true* та *false*. Значення типу *BOOLEAN* повертаються операціями порівняння (наприклад, порівняння на рівність, *s = '0'*) і використовуються в умовних операторах як *when* і *if*. Здавалося б, *BOOLEAN true* має бути еквівалентно *STD_LOGIC '1',* а *BOOLEAN false* має означати те саме, що й *STD_LOGIC '0'*, але ці типи були взаємозамінні аж до *VHDL* 2008. Наприклад, у старому коді на *VHDL* доводилося писати

y <= d1 when (s = '1') else d0;

а у *VHDL* 2008, де оператор *when* автоматично перетворює *s* з *STD_LOGIC* у *BOOLEAN,* вже можна писати тільки

y <= d1 when s else d0;

Але й у *VHDL* 2008 все ще потрібно писати

q <= '1' when (state = S2) else '0';

а не

q <= (state = S2);

тому що (*state = S*2) повертає результат типу *BOOELAN*, який може бути наданий сигналу типу *STD_LOGIC*.

Хоча ми не оголошуємо жодних сигналів типу *BOOLEAN*, вони автоматично виводяться з порівнянь і використовуються в умовних операторах. Аналогічно, у *VHDL* є тип *INTEGER* для подання цілих чисел зі знаком. Сигнали типу *INTEGER* можуть набувати значення як мінімум від $-(2^{31}-1)$ до $2^{31}-1$. В індексах масивів необхідно застосовувати цілі числа. Наприклад, в операторі *y <= a(3) and a(2) and a(1) and a(0)*; 0, 1, 2 і 3 – цілі, що слугують індексами для вибору бітів сигналу *a*. Для індексації не можна використовувати сигнал типу *STD_LOGIC* або *STD_LOGIC_VECTOR*, тому потрібно перетворити його в *INTEGER*, як показано нижче у прикладі восьмивходового мультиплексора, що обирає один біт вектора за допомогою трибітного індекса. Функція *TO_INTEGER*, визначена в бібліотеці

IEEE.NUMERIC_STD_UNSIGNED, перетворює із *STD_LOGIC_VECTOR* на невід'ємні значення *INTEGER*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mux8 is
   port(d: inSTD_LOGIC_VECTOR(7 downto 0);
        s: inSTD_LOGIC_VECTOR(2 downto 0);
        y: out STD_LOGIC);
end;
architecture synth of mux8 is
 begin
   y <= d(TO_INTEGER(s));
end;
```

VHDL також суворий щодо портів типу *out*: їх можна використовувати лише як виходи. Наступний приклад дво- і тривходового вентиля *I* не коректний, оскільки *v* – вихід, але застосовується також для обчислення *w*.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity and23 is
     port(a, b, c: in STD_LOGIC;
          v, w: out STD_LOGIC);
end;
architecture synth of and23 is
 begin
   v <= a and b;
   w <= v and c;
end;
```

Для розв'язання цієї проблеми *VHDL* має окремий тип порту – *buffer*. Сигнал, під'днаний до такого порту, поводиться як вихід, але може бути використаний всередині модуля. Ось виправлений текст оголошення інтерфейсу:

```
entity and23 is
port(a, b, c: in STD_LOGIC;
       v: buffer STD_LOGIC;
       w: out STD_LOGIC);
end;
```

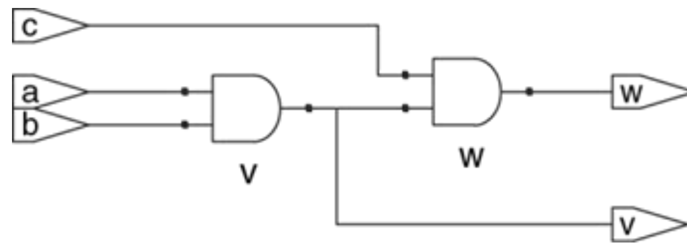


Рисунок 4.28 – Синтезована схема модуля *synth*

У *Verilog* та *SystemVerilog* цього обмеження ніколи не було, тому в них не потрібні буферні порти. У *VHDL* 2008 це обмеження також знято, але на момент написання це нововведення не підтримувалося програмою *Synplify*.

Унаслідок багатьох операцій, зокрема додавання, віднімання або операції булевої логіки, виходить те саме бітове подання результату, будь він зі знаком або без знака. На відміну від них, порівняння «більше-менше», множення та арифметичні зрушення вправо виконуються для чисел у додатковому коді зі знаком та двійкових чисел без знака по-різному. Ці операції розглядаються в розділі 5. У коді з прикладу 4.33 показано, як позначаються сигнали, що подають числа зі знаком.

Приклад 4.33.
Беззнаковий примножувач (a) і примножувач із знаком (b)

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input logic [3:0] a, b,
                  output logic [7:0] y);
   assign y = a * b;
 endmodule
// 4.33(b): signed multiplier
module multiplier(input logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a * b;
endmodule
```

У *SystemVerilog* сигнали розуміються як беззнакові за замовчуванням. Додавання модифікатора *signed* (наприклад, *logic signed [3:0] a*) призводить до того, що сигнал розглядається як число зі знаком.

VHDL

```
— 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity multiplier is
 port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture synth of multiplier is
begin
   y <= a * b;
end;
```

У *VHDL* для виконання арифметичних операцій та операцій порівняння над *STD_LOGIC_VECTOR* використовується бібліотека *NUMERIC_STD_UNSIGNED*. У цьому разі вектори вважаються беззнаковими.

Use IEEE.NUMERIC_STD_UNSIGNED.all

У *VHDL* також визначено типи даних *UNSIGNED* та *SIGNED* (у бібліотеці *IEEE.NUMERIC_STD*), але їх розгляд виходить за межі цього розділу.

4.8 Параметризовані модулі*

Досі для модулів у наведених прикладах входи та виходи були фіксованою шириною. Наприклад, нам знадобилося визначити два різних модулі для двовходового мультиплексора з чотирибітними та восьмибітними входами, але в мовах опису апаратури *HDL* можна описувати параметризовані модулі з портами змінної ширини [1–7].

У коді з прикладу 4.34 оголошується параметризований двовходовий мультиплексор із шириною входів, що за замовчуванням дорівнює восьми бітам, який потім застосовується для створення чотиривходових мультиплексорів із восьмибітними та 12-бітними входами.

Приклад 4.34.
Параметризовані N-бітні двовходові мультиплексори

SystemVerilog

```
module mux2
# (Parameter width = 8)
```

```
    (input logic [width-1:0] d0, d1,
     input logic s,
     output logic [width-1: 0]y);
   assign y = s? d1: d0;
endmodule
```

У *SystemVerilog* можлива конструкція *#(parameter ...)* перед списком входів і виходів визначення параметрів модуля. У прикладі вище оператор *parameter* містить параметр на ім’я *width* зі значенням за замовчуванням, рівним 8. Число бітів входів і виходів може залежати від параметра.

```
module mux4_8(input logic [7:0] d0, d1, d2, d3,
              input logic [1:0] s,
              output logic [7:0] y);
          logic [7: 0] low, hi;
mux2 lowmux(d0, d1, s[0], low);
mux2 himux(d2, d3, s[0], hi);
mux2 outmux(low, hi, s[1], y);
 endmodule
```

Восьмибітний чотиривходовий мультиплексор має три екземпляри двовходового мультиплексора із шириною входів, установленою за замовчуванням. На відміну від нього, у 12-бітному чотиривходовому мультиплексорі *mux4_12* необхідно перевизначити ширину входів за допомогою конструкції *#( )* перед ім’ям екземпляра *(instance):*

```
module mux4_12(input logic [11:0] d0, d1, d2, d3,
               input logic [1:0]s,
              output logic [11:0] y);
   logic [11:0] low, hi;
mux2 # (12) lowmux (d0, d1, s [0], low);
mux2 # (12) himux (d2, d3, s [0], hi);
mux2 # (12) outmux (low, hi, s [1], y);
 endmodule
```

Не плутайте застосування знака # для позначення затримок із використанням *#(...)* у процесі оголошення та перевизначення параметрів.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
generic(width: integer := 8);
port(d0,
d1: in STD_LOGIC_VECTOR(width-1 downto 0);
 s: in STD_LOGIC;
y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture synth of mux2 is
begin
   y <= d1 when s else d0;
 end;
```

Оператор *generic* містить вказівки значення 8 за замовчуванням для *width* типу ціле.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity mux4_8 is
port(d0, d1, d2,
d3: in STD_LOGIC_VECTOR(7 downto 0);
s: in STD_LOGIC_VECTOR(1 downto 0);
y: out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture synth of mux4_8 is
component mux2
generic(width: integer := 8);
port(d0,
d1: in STD_LOGIC_VECTOR(width-1 downto 0);
s: in STD_LOGIC;
y: out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
    lowmux: mux2 port map(d0, d1, s(0), low);
    himux: mux2 port map(d2, d3, s(0), hi);
    outmux: mux2 port map(low, hi, s(1), y);
end;
```

Восьмибітний чотиривходовий мультиплексор *mux4_8* містить три мультиплексори 2:1 із шириною за замовчуванням. На відміну від нього, у 12-бітному чотиривходовому мультиплексорі *mux4_12* необхідно буде перевизначити ширину за замовчуванням за допомогою *generic map:*

```
lowmux: mux2 generic map(12)
     port map(d0, d1, s(0), low);
himux: mux2 generic map(12)
     port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
     port map(low, hi, s(1), y);
```

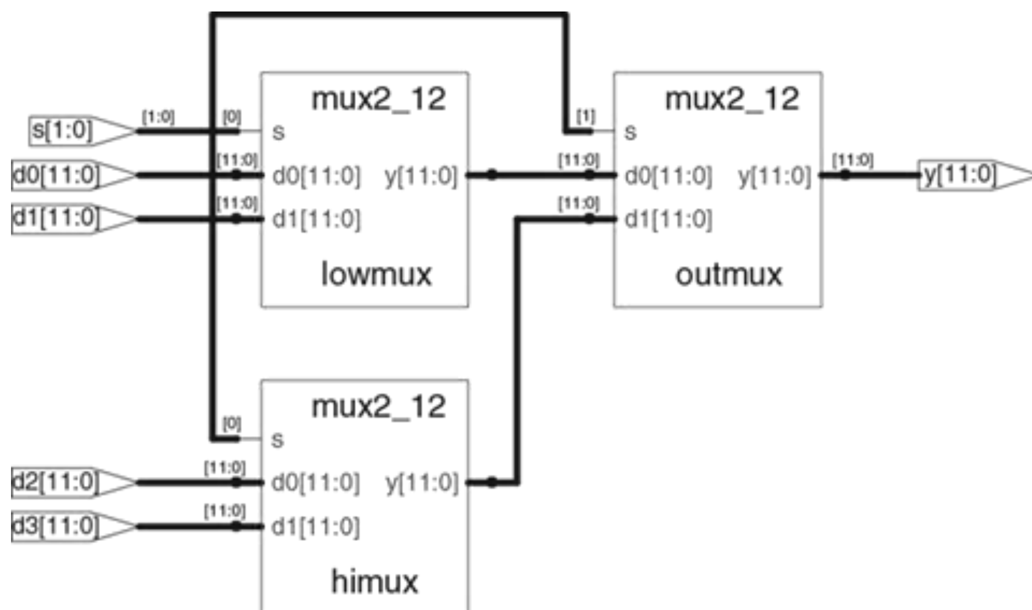


Рисунок 4.29 – Синтезована схема модуля *mux4_12*

Приклад 4.35.

Параметризований дешифратор N:2N

SystemVerilog

```
module decoder
# (Parameter N = 3)
(input logic [N-1:0] a,
output logic [2** N-1:0] y);
always_comb
begin
  y = 0;
  y[a] = 1;
end
```

```
endmodule
2**N означає 2N
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;
entity decoder is
      generic(N: integer := 3);
      port(a: in STD_LOGIC_VECTOR(N-1 downto 0);
      y: out STD_LOGIC_VECTOR(2**N–1 downto 0));
end;
architecture synth of decoder is
 begin
    process(all)
     begin
    y <= (OTHERS => '0');
     y(TO_INTEGER(a)) <= '1';
   end process;
 end;
2**N означає 2N
```

У коді прикладу 4.35 продемонстровано дешифратор, що є вдалим зразком параметризованого модуля. Широкий дешифратор *N:2N* досить важко описувати за допомогою оператора *case*, але це легко зробити за допомогою параметризованого модуля, що встановлює потрібний біт в 1. Інакше кажучи, у дешифраторі використано блокувальне присвоєння для встановлення всіх бітів в 0, а потім потрібний біт змінюється в 1.

У мовах опису апаратури також передбачено оператор *generate* для отримання різної кількості апаратури залежно від значення параметра. В операторі *generate* допускаються цикли *for* та оператори *if* для визначення кількості та властивості бажаної апаратури. У коді прикладу 4.36 демонструється, як застосовувати оператори *generate* для отримання *N*-вхідної функції *I* з каскаду двовходових вентилів *I*. Звичайно, для цієї конкретної мети краще підійшла б операція редукції, але цей приклад ілюструє загальний принцип використання оператора *generate*.

Застосовуйте оператори generate з обережністю –
через них легко можна ненавмисно отримати дуже велику схему!

Приклад 4.36.
Параметризований N-входовий вентиль I

SystemVerilog

```
module andN
# (Parameter width = 8)
(input logic [width-1:0] a, output logic y);
genvar i;
logic [width-1:0] x;
generate
  assign x[0] = a[0];
 for(i=1; i<width; i=i+1) begin: forloop
    assign x[i] = a[i] & x[i-1];
end
 endgenerate
assign y = x [width-1];
 endmodule
```

Оператор *for* проходить по *i* = 1, 2, ..., *width*-1 для отримання безлічі послідовних вентилів *I*. Після *begin* у циклі *for* всередині *generate* має бути двокрапка й довільна мітка (у цьому разі *forloop*). (*Зверніть також увагу на оголошення змінної циклу як genvar*).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity andN is
     generic(width: integer := 8);
     port(a: in STD_LOGIC_VECTOR(width-1 downto 0);
             y: out STD_LOGIC);
end;
auchitecture synth of andN is
      signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    x(0) <= a(0);
    gen: for i in 1 to width-1 generate
      x(i) <= a(i) and x(i-1);
   end generate;
y <= x (width-1);
end;
```

Змінну циклу *generate* оголошувати не потрібно.

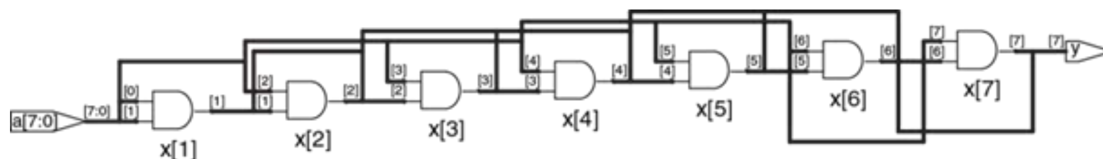


Рисунок 4.30 – Синтезована схема модуля *andN*

4.9 Середовище тестування

Середовище тестування – це модуль *HDL*, що застосовується для тестування іншого модуля і називається «тестований пристрій» (*device under test, DUT*). (Деякі програми розробки називають модуль *unit under test, UUT,* що тестується.) Середовище тестування містить оператори для генерації значень, що подаються на входи *DUT* і в ідеалі також для перевірки правильних значень, що виходять на виході. Набори вхідних і бажаних вихідних значень називаються тестовими векторами.

Спробуємо протестувати модуль *sillyfunction* з п. 4.1.1, що обчислює

```
y <= (not a and not b and not c) or
     (a and not b and not c) or
     (a and not b and c);
```

Це простий модуль, тому можна виконати вичерпне тестування, подав на входи всі вісім можливих тестових векторів.

У прикладі 4.37 продемонстровано просте середовище тестування, що передбачає тестований блок *DUT*, потім подає значення векторів на його входи. Блокувальні присвоєння та затримки потрібні для застосування значень у бажаному порядку. Користувач має переглянути результати симуляції та перевірити їх правильність. Середовища тестування симулюються так само, як і інші модулі, але не синтезуються.

Приклад 4.37.
Середовище тестування

SystemVerilog

```
module testbench1();
 logic a, b, c, y;
// instantiate device under test
 sillyfunction dut(a, b, c, y);
// apply inputs one at a time
```

```
 initial begin
a = 0; b = 0; c = 0; #10;
c = 1;               #10;
b = 1; c = 0;        #10;
c = 1;               #10;
a = 1; b = 0; c = 0; #10;
c = 1;               #10;
b = 1; c = 0;         #10;
c = 1;                #10;
end
endmodule
```

Оператор *initial* виконує оператори, що містяться в ньому, на початку симуляції. У разі він подає на входи набір 000 і чекає 10 одиниць часу. Потім він подає 001 і чекає ще 10 одиниць часу доти, доки не буде подано всі вісім можливих наборів. Оператори *initial* мають використовуватися тільки в тестуваннях для симуляції, а не в модулях, з яких буде синтезована апаратура. В апаратурі немає способу магічним чином виконати послідовність кроків під час увімкнення.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity testbench1 is -- no inputs or outputs
end;
architecture sim of testbench1 is
component sillyfunction
   port(a, b, c: in STD_LOGIC;
   y: out STD_LOGIC);
end component;
signal a, b, c, y: STD_LOGIC;
begin
— instantiate device under test
dut: sillyfunction port map(a, b, c, y);
— apply inputs one at a time
process begin
a <= ‘0’; b <= ‘0’; c <= ‘0’;  wait for 10 ns;
 c<= ‘1’;                      wait for 10 ns;
b <= ‘1’; c <= ‘0’;             wait for 10 ns;
```

```
c<= '1';                    wait for 10 ns;
a <= '1'; b <= '0'; c <= '0';   wait for 10 ns;
c<= '1';                    wait for 10 ns;
b <= '1'; c <= '0';          wait for 10 ns;
c<= '1';                    wait for 10 ns;
wait; — wait forever
end process;
 end;
```

Оператор *process* подає на входи набір 000 і чекає на 10 нс. Потім він подає 001 і чекає ще 10 нс доти, доки не буде подано всі вісім можливих наборів.

Нарешті, процес входить у вічне очікування, інакше його виконання почалося б заново, і він почав би подавати тестові вектори повторно.

Перевіряти правильність виходів вручну важко й загрожує помилками, та й тестувати подумки відносно легко, коли схема свіжа в пам'яті. Однак якщо доведеться додавати до неї виправлення за кілька тижнів, то визначати згодом, яке значення необхідно вважати правильним, буде набагато важче. Значно краще написане середовище тестування із самоперевіркою продемонстровано в коді прикладу 4.38.

Приклад 4.38.
Середовище тестування з самоперевіркою

SystemVerilog

```
module testbench2(); logic a, b, c, y;
// instantiate device under test
sillyfunction dut(a, b, c, y);
// apply inputs one at a time
// checking results
initial begin
   a = 0; b = 0; c = 0; #10;
assert (y === 1) else $error("000 failed.");
   c = 1; #10;
assert (y === 0) else $error("001 failed.");
b = 1; c = 0; #10;
assert (y === 0) else $error("010 failed."); c = 1; #10;
assert (y === 0) else $error("011 failed."); a = 1; b = 0; c = 0; #10;
assert (y === 1) else $error("100 failed."); c = 1; #10;
```

```
assert (y === 1) else $error("101 failed."); b = 1; c = 0; #10;
assert (y === 0) else $error("110 failed."); c = 1; #10;
assert (y === 0) else $error("111 failed."); end
endmodule
```

Оператор *assert* у *SystemVerilog* перевіряє, чи істинна зазначена умова. Якщо ні, то виконується оператор *else*. Системна процедура *$error* в операторі *else* друкує повідомлення про помилку із зазначенням порушеної умови. Оператори *assert* ігноруються в процесі синтезу.

У *SystemVerilog* порівняння з допомогою *== і !=* працює для сигналів, що приймають значення *x* і *z*. Середовище тестування використовує оператори *=== і !==* порівнянь на рівність і нерівність відповідно, оскільки ці оператори працюють і з операндами, значення яких може бути *x* чи *z*.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench2 is -- no inputs or outputs
end;
architecture sim testbench2 is
component sillyfunction
  port(a, b, c: in STD_LOGIC;
         y: out STD_LOGIC);
end component;
signal a, b, c, y: STD_LOGIC;
begin
— instantiate device under test
dut: sillyfunction port map(a, b, c, y);
— apply inputs one at a time
— checking results
process begin
a <= ‘0’; b <= ‘0’; c <= ‘0’;     wait for 10 ns;
  assert y = ‘1’ report "000 failed.";
c<= ‘1’;                          wait for 10 ns;
  assert y = ‘0’ report "001 failed.";
b <= ‘1’; c <= ‘0’;               wait for 10 ns;
   assert y = ‘0’ report "010 failed.";
c<= ‘1’;                          wait for 10 ns;
   assert y = ‘0’ report "011 failed.";
```

```
a <= ‘1’; b <= ‘0’; c <= ‘0’;      wait for 10 ns;
   assert y = ‘1’ report "100 failed.";
c<= ‘1’;                           wait for 10 ns;
   assert y = ‘1’ report "101 failed.";
b <= ‘1’; c <= ‘0’;                wait for 10 ns;
   assert y = ‘0’ report "110 failed.";
c<= ‘1’;                           wait for 10 ns;
    assert y = ‘0’ report "111 failed.";
  wait; --wait forever
 end process;
end;
```

Оператор *assert* перевіряє умову та друкує повідомлення, вказане після *report*, якщо умова не виконана. Оператор має сенс лише за симуляції – не в процесі синтезу.

Писати код для кожного тестового вектора також стає стомливим, особливо для модулів, що вимагають значної кількості тестових векторів. Ще краще тримати тестові вектори в окремому файлі. Тоді середовище тестування читатиме їх із файлу, подаватиме вхідний вектор на входи *DUT*, перевірятиме, що значення виходів збігаються з вихідним вектором, і повторювати, доки не досягне кінця файлу.

У прикладі 4.39 продемонстровано таке середовище тестування. Воно генерує тактовий сигнал за допомогою оператора *always/process* без списку чутливості, тому оператор виконується як нескінченний цикл. На початку симуляції середовище читає тестові вектори з текстового файлу та виставляє *reset* протягом двох тактів. Хоча тактового сигналу та скидання не потрібно для тестування комбінаційної логіки, але вони будуть важливими для тестування послідовних пристроїв.

Ось *example.tv* – файл, що має входи та очікуваний вихід у двійковому вигляді:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Приклад 4.39.

Середовища тестування з файлом тестових векторів

SystemVerilog

```
module testbench3();
logic clk, reset;
logic a, b, c, y, yexpected;
 logic [31:0] vectornum, errors;
logic [3:0] testvectors[10000:0];
// instantiate device under test
 sillyfunction dut(a, b, c, y);
// generate clock
 always
  begin
    clk = 1; # 5; clk = 0; # 5;
 end
// at start of test, load vectors
// and pulse reset
initial
    begin
          $readmemb("example.tv", testvectors);
          vectornum = 0; errors = 0;
          reset = 1; # 27;
          reset = 0;
 end
// apply test vectors on rising edge of clk
  always @(posedge clk)
     begin
       #1; {a, b, c, yexpected} = testvectors[vectornum];
     end
// check results on falling edge of clk
always @(negedge clk)
     if (~reset) begin // skip during reset
          if (y !== yexpected) begin // check result
             $display("Error: inputs = %b", {a, b, c});
             $display("outputs = %b (%b expected)", y, yexpected);
             errors = errors + 1;
    end
```

```
vectornum = vectornum + 1;
    if (testvectors[vectornum] === 4'bx) begin
   $display("%d tests completed with %d errors", vectornum, errors);
   $finish;
    end
  end
 endmodule
```

$readmemb читає файл із двійковими числами в масив *testvectors*.

$readmemh – аналогічно, але читає файл із шістнадцятковими числами.

Наступний блок коду чекає одну одиницю часу після переднього фронту тактового сигналу (щоб уникнути плутанини, якщо тактовий сигнал і дані змінюються одночасно), потім встановлює три входи (*a, b* і *c*) й очікуваний вихід (*yexpected*) згідно з чотирма бітами в поточному тестовому векторі.

Середовище порівнює отриманий вихід *y* з очікуваним виходом *yexpected* і друкує повідомлення про помилку, якщо вони не збігаються. *%b* та *%d* означають друк значень у двійковому та десятковому вигляді відповідно. *$display* – це системна процедура друку у вікні симулятора. Наприклад,

$display("%b %b", y, yexpected); друкує два значення *y* та *yexpected*, у двійковому вигляді. *%h* друкує в шістнадцятковому вигляді.

Цей процес повторюється, поки в масиві *testvectors* не закінчаться прочитані з файлу тестові вектори. *$finish* завершує симуляцію.

*Зауважте, що, хоча модуль **SystemVerilog** передбачає аж до 10001 тестових векторів, симуляція завершиться після подання восьми векторів з файлу.*

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use STD.TEXTIO.all;
entity testbench3 is -- no inputs or outputs
end;
architecture sim testbench3 is
componemt sillyfunction
    port(a, b, c: in STD_LOGIC;
           y: out STD_LOGIC);
end component;
```

```
   signal a, b, c, y: STD_LOGIC;
   signal y_expected: STD_LOGIC;
   signal clk, reset: STD_LOGIC;
begin
— instantiate device under test
dut: sillyfunction port map(a, b, c, y);
— generate clock
process begin
    clk <= ‘1’; wait for 5 ns;
    clk <= ‘0’; wait for 5 ns;
end process;
— at start of test, pulse reset
process begin
  reset <= ‘1’; wait for 27 ns;
  reset <= ‘0’;
  wait;
end process;
— run tests
 process is
   file tv: text;
  variable L: line;
  variable vector_in: std_logic_vector(2 downto 0);
  variable dummy: character;
  variable vector_out:   std_logic;
  variable vectornum: integer := 0;
  variable errors: integer := 0;
begin
   FILE_OPEN(tv, “example.tv”, READ_MODE);
   while not endfile(tv) loop
— change vectors on rising edge
   wait until rising_edge(clk);
— read the next line of testvectors and split in pieces
 readline(tv, L);
 read(L, vector_in);
 read(L, dummy); - skip over underscore
 read (L, vector_out);
 (a, b, c) <= vector_in(2 downto 0) after 1 ns;
```

```
 y_expected <= vector_out after 1 ns;
-- -- check results on falling edge
wait until falling_edge(clk);
if y /= y_expected then
      report “Error: y = ” & std_logic’image(y);
      errors: = errors + 1;
end if;
vectornum: = vectornum + 1;
end loop;
-- summarize results at end of simulation
 if (errors = 0) then
    report "NO ERRORS -- " & integer’image(vectornum) &
    “ tests completed successfully.”
    severity failure;
else
   report integer’image(vectornum) &
  “ tests completed, errors = ” &
   integer’image(errors)
  severity failure;
    end if;
  end process;
end;
```

Код *VHDL* використовує команди читання з файлу, розгляд яких не передбачений у цьому розділі, але дає зрозуміти, який вигляд має середовище тестування з самоперевіркою на *VHDL.*

Нові значення входів подаються по передньому фронту тактового сигналу, а вихід перевіряється заднім фронтом. Повідомлення про помилки видаються на момент виникнення. Наприкінці симуляції середовище тестування друкує результат: загальна кількість тестових векторів і кількість виявлених помилок.

Середовище *HDL* у прикладі 4.39 надмірне для такої простої схеми. Однак його легко змінити для тестування більш складних схем, якщо замінити файл *example.tv,* додавши в середовище інший пристрій, що тестується, і змінити кілька рядків коду для встановлення входів і перевірки виходів.

4.10 Резюме

Мови опису апаратури (*HDL*) – дуже важливі інструменти для розробників сучасної цифрової електроніки. Вивчивши *SystemVerilog* або *VHDL*, ви зможете розробляти цифрові системи набагато швидше, ніж за традиційного креслення принципових схем. Цикл налагодження також зазвичай набагато коротший, оскільки зміни полягають у редагуванні тексту, а не стомливому перепід'єднанню дротів на схемі. Однак з використанням *HDL* цикл налагодження може бути й набагато довшим, якщо ви погано уявляєте, яку апаратуру описує ваш код.

Мови опису апаратури застосовують і для симуляції, і для синтезу. Логічна симуляція – потужний спосіб протестувати систему на комп'ютері до того, як вона перетвориться на апаратуру. Симулятори дають змогу перевірити ті значення сигналів у системі, що можуть бути недоступні для вимірювання на реальній електричній схемі. Логічний синтез перетворює код на *HDL* на цифрові логічні схеми.

Найважливіше, що потрібно пам'ятати в процесі написання коду на *HDL* – це те, що ви описуєте справжню апаратуру, а не пишете програму для комп'ютера. Початківці часто роблять помилку, створюючи код *HDL,* не продумавши, яку саме апаратуру вони бажають отримати.

Якщо не знаєте, яка апаратура вийде з коду, ви швидше за все не досягнете необхідного результату. Тому починайте з ескізу блокової діаграми системи, визначивши, які її частини – комбінаційна логіка, які послідовні схеми або кінцеві автомати тощо. Потім пишіть код для кожної частини *HDL*, використовуючи правильні конструкції для потрібного типу апаратури.

ВПРАВИ

У цьому розділі вправи можна виконувати мовою, яка вам більше подобається. Якщо у вас є симулятор, необхідно випробувати, що ви написали. Виведіть значення сигналів і поясніть, як вони доводять, що схема працює правильно. Якщо у вас синтезатор, синтезуйте схему. Надрукуйте отриману принципову схему та поясніть, чому вона задовольняє очікування.

Вправа 4.1. Накресліть діаграму схеми, яку описано нижче. Спростіть схему, досягши мінімуму вентилів.

SystemVerilog

```
module exercise1(input logic a, b, c,
                 output logic y, z);
assign y = a & b & c | a & b & ~ c | a & b & c;
 assign z = a & b | ~ a & ~ b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity exercise1 is
    port(a, b, c: in STD_LOGIC;
         y, z: out STD_LOGIC);
end;
architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or (a and not b and c);
  z <= (a and b) or (not a and not b);
and;
```

Вправа 4.2. Накресліть діаграму схеми, описаної нижче. Спростіть схему, досягши мінімуму вентилів.

SystemVerilog

```
module exercise2(input logic[3:0] a,
                 output logic [1:0] y);
always_comb
if (a[0]) y = 2’b11;
   else if (a[1]) y = 2’b10;
   else if (a[2]) y = 2’b01;
   else if (a[3]) y = 2’b00;
   else y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity exercise2 is
   port(a: in STD_LOGIC_VECTOR(3 downto 0);
        y: out STD_LOGIC_VECTOR(1 downto 0));
end;
```

```
architecture synth of exercise2 is
 begin
   process(all) begin
      if a(0) then y <= "11";
     elsif a(1) then y <= "10";
     elsif a(2) then y <= "01";
     elsif a(3) then y <= "00";
     else y <= a(1 downto 0);
     end if;
end process;
```

Вправа 4.3. Напишіть модуль на ***HDL***, що обчислює чотиривходову функцію *XOR* (що виключає АБО). Вхід позначте *a3:0*, вихід – *y*.

Вправа 4.4. Напишіть середовище тестування із самоперевіркою для вправи 4.3. Створіть файл, що містить усі 16 варіантів входів. Просимулюйте схему та переконайтеся, що вона працює. Додайте помилку у файл із тестовими векторами та переконайтеся, що середовище тестування повідомляє про розбіжність результатів.

Вправа 4.5. Напишіть на *HDL* модуль *minority* з трьома входами *a, b, c* і одним виходом *y*, що приймає значення *TRUE,* якщо не менше ніж два входи дорівнює *FALSE*.

Вправа 4.6. Напишіть на *HDL* модуль для керування семисегментним індикатором шістнадцяткових цифр. Мають підтримуватися як цифри 0–9, так і *A, B, C, D, E* і *F*.

Вправа 4.7. Напишіть середовище тестування із самоперевіркою для вправи 4.6. Створіть файл, що містить усі 16 варіантів входів. Просимулюйте схему та переконайтеся, що вона працює. Додайте помилку у файл із тестовими векторами та переконайтеся, що середовище тестування повідомляє про розбіжність результатів.

Вправа 4.8. Напишіть восьмивходовий мультиплексор з ім’ям *mux8*, входами *s2:0, d0, d1, d2, d3, d4, d5, d6, d7* і виходом *y*.

Вправа 4.9. Напишіть структурний модуль для обчислення логічної функції $y = ab\sim + b\sim c\sim + a\sim bc$ за допомогою мультиплексорної логіки. Використовуйте мультиплексор із вправи 4.8.

Вправа 4.10. Повторіть вправу 4.9 за допомогою чотиривходового мультиплексора та будь-якої кількості вентилів НІ.

Вправа 4.11. У п. 4.5.4 було помічено, що синхронізатор можна описати за допомогою блокувальних присвоєнь у правильному порядку.

Придумайте просту послідовну схему, яку не можна правильно описати за допомогою блокувальних присвоєнь незалежно від їх порядку.

Вправа 4.12. Напишіть модуль на *HDL* для схеми пріоритетів із вісьмома входами.

Вправа 4.13. Напишіть модуль *HDL* для дешифратора 2:4.

Вправа 4.14. Напишіть модуль на *HDL* для дешифратора 6:64 за допомогою трьох екземплярів дешифратора 2:4 з вправи 4.13 та кількох тривходових вентилів І.

Вправа 4.15. Напишіть модуль *HDL*, що реалізує логічні вирази з вправи 2.13.

Вправа 4.16. Напишіть модуль на *HDL*, що реалізує схему вправи 2.26.

Вправа 4.17. Напишіть модуль на *HDL*, що реалізує схему вправи 2.27.

Вправа 4.18. Напишіть модуль *HDL*, що реалізує логічну функцію з вправи 2.28. Зверніть особливу увагу на те, як поводитися з незначимими бітами.

Вправа 4.19. Напишіть модуль *HDL*, що реалізує функції з вправи 2.35.

Вправа 4.20. Напишіть модуль *HDL*, що реалізує кодер із пріоритетами з вправи 2.36.

Вправа 4.21. Напишіть модуль *HDL*, що реалізує модифікований кодер із пріоритетами з вправи 2.37.

Вправа 4.22. Напишіть модуль на *HDL*, що реалізує перетворювач із бінарного в унарний код із вправи 2.38.

Вправа 4.23. Напишіть модуль *HDL*, що реалізує функцію днів місяця з питання 2.2.

Вправа 4.24. Накресліть діаграму станів кінцевого автомата, описаного нижченаведеним кодом на *HDL*:

SystemVerilog

```
module fsm2(input logicclk, reset,
                input logica, b,
                output logicy);
logic [1:0] state, nextstate;
parameter S0 = 2'b00;
parameter S1 = 2'b01;
 parameter S2 = 2'b10;
 parameter S3 = 2'b11;
always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;
    always_comb
      case (state)
        S0: if (a ^ b) nextstate = S1;
            else nextstate = SO;
       S1: if (a & b) nextstate = S2;
          else nextstate = SO;
      S2: if (a | b) nextstate = S3;
          else nextstate = SO;
      S3: if (a | b) nextstate = S3;
          else nextstate = SO;
    endcase
  assign y = (state = = S1) | (state = = S2);
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
 entity fsm2 is
      port(clk, reset: in STD_LOGIC;
                   a, b: in STD_LOGIC;
                    y: out STD_LOGIC);
end;
architecture synth of fsm2 is
   type statetype is (S0, S1, S2, S3);
   signal state, nextstate: statetype;
begin
```

```
  process(clk, reset) begin
     if reset then state <= S0;
    elsif rising_edge(clk) then
        state <= nextstate;
     end if;
end process;
process(all) begin
  case state is
     when S0 =>if (a xor b) then
           nextstate <= S1;
           else nextstate <= S0;
                 end if;
     when S1 =>if (a and b) then
           nextstate <= S2;
           else nextstate <= S0;
                 end if;
      when S2 =>if (a or b) then
           nextstate <= S3;
           else nextstate <= S0; end if;
           when S3 =>if (a or b) then
           nextstate <= S3;
           else nextstate <= S0;
                       end if;
  end case;
end process;
y <= '1' when ((state = S1) or (state = S2))
else '0';
end;
```

Вправа 4.25. Накресліть діаграму станів кінцевого автомата, описаного нижченаведеним кодом на *HDL*. Автомати такого типу використовуються д ля передбачення переходів у деяких мікропроцесорах.

SystemVerilog

```
module fsm1(input logic clk, reset,
              input logic taken, back,
              output logic predicttaken);
logic [4:0] state, nextstate;
 parameter S0 = 5'b00001;
 parameter SI = 5'b00010;
```

```
parameter S2 = 5'b00100;
parameter S3 = 5'b01000;
parameter S4 = 5'b10000;
always_ff @(posedge clk, posedge reset)
   if (reset) state <= S2;
   else state <= nextstate;
 always_comb
  case (state)
      S0: if (taken) nextstate = S1;
           else nextstate = S0;
      S1: if (tekan) nextstate = S2;
            else nextstate = S0;
      S2: if (tekan) nextstate = S3;
            else nextstate = S1;
     S3: if(taken) nextstate = S4;
           else nextstate = S2;
     S4: if(taken) nextstate = S4;
           else nextstate = S3;
    default: nextstate = S2;
endcase
assign predicttaken = (state = = S4) |
                      (state = S3) |
                      (state = = S2 && back);
Endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164. all;
 entity fsm1 is
      port(clk, reset: in STD_LOGIC;
            taken, back: in STD_LOGIC;
            predicttaken: out STD_LOGIC);
end;
architecture synth of fsm1 is
     type statetype is (S0, S1, S2, S3, S4);
     signal state, nextstate: statetype;
begin
   process(clk, reset) begin
        if reset then state <= S2;
        elsif rising_edge(clk) then
```

```
            state <= nextstate;
        end if;
  end process;
 process(all) begin
     case state is
       whtn S0 => if taken then
                          nextstate <= S1;
                      else nextstate <= S0;
                      end if;
      when S1 => if taken then
                          nextstate => S2;
                      else nextstate <= S0;
                      end if;
       when S2 => if taken then
                          nextstate <= S3;
                     else nextstate <= S1;
                     end if;
       when S3 => if taken then
                           nextstate <= S4;
                     else nextstate <= S2;
                     end if;
       when S4 => if taken then
                          nextstate <= S4;
                    else nextstate <= S3;
                    end if;
       when others =>nextstate <= S2;
   end case;
 end process;
 —— output logic
  predicttaken <= '1' when
            ((state = S4) or (state = S3) or
            (state = S2 and back = '1'))
   else '0';
 end;
```

Вправа 4.26. Напишіть модуль на *HDL* для засувки *SR*.

Вправа 4.27. Напишіть модуль на *HDL* для *JK*-тригера з входами *clk, J* і *K* і виходом *Q*. По передньому фронту тактового сигналу *Q* зберігає попередній

стан, якщо $J = K = 0$ стає рівним 1, якщо $J = 1$, скидається в 0 за умови $K = 1$ та інвертується, якщо $J = K = 1$.

Вправа 4.28. Напишіть модуль *HDL* для засувки на рис. 3.18. Використовуйте один оператор присвоєння кожному вентилю. Задайте затримку 1 (або 1 нс) для кожного вентиля. Просимулюйте засувку й переконайтеся, що вона працює правильно. Потім збільште затримку інвертора.

Наскільки тривалою може бути затримка, перш ніж засувка перестане нормально працювати через гонку сигналів?

Вправа 4.29. Напишіть модуль *HDL* для контролера світлофора з п. 4.3.1.

Вправа 4.30. Напишіть три модулі *HDL* для факторизованого контролера світлофора з режимом параду з прикладу 3.8. Назвіть ці модулі *controller, mode* та *lights* і назвіть їх входи-виходи, як на рис. 3.33, *b*.

Вправа 4.31. Напишіть модуль *HDL*, що описує схему на рис. 3.42.

Вправа 4.32. Напишіть модуль *HDL* для кінцевого автомата з діаграмою станів, зображеною на рис. 3.69 із вправи 3.22.

Вправа 4.33. Напишіть модуль *HDL* для кінцевого автомата з діаграмою станів, зображеною на рис. 3.70 із вправи 3.23.

Вправа 4.34. Напишіть модуль *HDL* для покращеного контролера світлофора з вправи 3.24.

Вправа 4.35. Напишіть модуль на *HDL* із вправи 3.25.

Вправа 4.36. Напишіть модуль *HDL* для дозатора напоїв із вправи 3.26.

Вправа 4.37. Напишіть модуль *HDL* для лічильника в коді Грея з вправи 3.27.

Вправа 4.38. Напишіть модуль на *HDL* для лічильника в коді Грея ВГОРУ / ВНИЗ із вправи 3.28.

Вправа 4.39. Напишіть модуль *HDL* для кінцевого автомата з вправи 3.29.

Вправа 4.40. Напишіть модуль *HDL* для кінцевого автомата з вправи 3.30.

Вправа 4.41. Напишіть модуль *HDL* для послідовного обчислення протилежного значення з питання 3.2.

Вправа 4.42. Напишіть модуль *HDL* для схеми з вправи 3.31.

Вправа 4.43. Напишіть модуль *HDL* для схеми з вправи 3.32.

Вправа 4.44. Напишіть модуль *HDL* для схеми з вправи 3.33.

Вправа 4.45. Напишіть модуль *HDL* для схеми з вправи 3.34, якщо бажаєте – з використанням повного суматора з п. 4.2.5.

Вправи з SystemVerilog

Вправа 4.46. Що означає, коли *SystemVerilog* сигнал оголошено як *tri*?

Вправа 4.47. Перепишіть модуль *syncbad* із прикладу 4.29. Використовуйте неблокувальні присвоєння, але змініть код так, щоб вийшов правильний синхронізатор із двома тригерами.

Вправа 4.48. Розгляньте подані два модулі *SystemVerilog*. Чи функціонально однакові вони? Наведіть схеми апаратури, яку кожен з них описує.

```
module code1(input logic clk, a, b, c,
                  output logic y);
logic x;
always_ff @(posedge clk) begin
    x <= a & b;
    y<=x | c;
end
endmodule

module code2 (input logic a, b, c, clk,
                  output logic y);
logic x;
always_ff @(posedge clk) begin
   y <= x | c;
   x <= a & b;
 end
endmodule
```

Вправа 4.49. Повторіть вправу 4.48, якщо в кожному привласненні <= замінено на =.

Вправа 4.50. У наведених нижче модулях на *SystemVerilog* показано типові помилки, помічені авторами в студентів під час виконання лабораторних робіт. Поясніть помилку в кожному модулі та вкажіть, як її виправити.

(a)
```
module latch(input logic clk,
             input logic[3:0] d,
             output reg [3:0] q);
always @(clk)
    if (clk) q <= d;
endmodule
```

(b)
```
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always @(a)
begin
   y1 = a & b;
   y2 = a | b;
   y3 = a^b;
   y4 = ~(a & b);
   y5 = ~ (a | b);
 end
endmodule
```

(c)
```
module mux2(input logic [3:0] d0, d1,
            input logic s, output logic [3:0] y);
always @ (posedge s)
   if (s) y <= d1;
   else y <= d0;
endmodule
```

(d)
```
module twoflops(input logic clk,
                input logic d0, d1,
                output logic q0, q1);
always @ (posedge clk)
  q1 = d1;
  q0 = d0;
 endmodule
```

(e)
```
module FSM(input logic clk,
           input logic a,
```

```
                     output logic out1, out2);
 logic state;
// next state logic and register (sequential)
always_ff @(posedge clk)
    if (state == 0) begin
      if (a) state <= 1;
   end else begin
       if (~a) state <= 0;
      end
always_comb
 // output logic (combinational)
      if (state == 0) out1 = 1;
      else out2 = 1;
 endmodule
```

(f)
```
module priority(input logic [3:0] a,
                         output logic [3:0] y);
always_comb
   if (a[3]) y = 4'b1000;
   else if (a[2]) y = 4'b0100;
   else if (a[1]) y = 4'b0010;
   else if (a[0]) y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input logicclk,
                                  input logicreset,
                                  output logicout);
 logic [1:0] state, nextstate;
 parameter S0 = 2'b00;
 parameter S1 = 2'b01;
 parameter S2 = 2'b10;
// State Register
always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;
// Next State Logic
always @ (state)
  case (state)
    S0: nextstate = S1;
```

```
    S1: nextstate = S2;
     S2: nextstate = S0;
endcase
// Output Logic
assign out = (state = S2);
endmodule
```

(h) **module mux2tri(input logic [3:0] d0, d1,**

```
module mux2tri(input logic [3:0] d0, d1,
                input logic s,
                output tri [3:0] y);
tristate t0(d0, s, y);
tristate t1(d1, s, y);
endmodule
```

(i) **module floprsen(input logic clk,**

```
module floprsen(input logic clk,
                input logic reset,
                input logic set,
                input logic [3:0] d,
                output logic [3:0] q);
always_ff @ (posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= d;
 always @(set)
    if (set) q <= 1;
endmodule
```

(j) **module and3(input logic a, b, c,**

```
module and3(input logic a, b, c,
output logic y); logic tmp;
always @ (a, b, c)
begin
     tmp <= a & b;
     y <= tmp & c;
end
 endmodule
```

Вправа 4.51. Навіщо в ***VHDL*** необхідно писати

q <= ‘1’ when state = S0 else ‘0’;

а не лише

q <= (state = S0);

Вправа 4.52. У кожному з наведених нижче модулів на *VHDL* є помилка. Для стислості показано лише описи архітектури; вважайте, що оголошення бібліотеки та оголошення інтерфейсу правильні. Поясніть помилку та вкажіть, як її виправити.

(a) **архітектура synth of latch is**
```
begin
   process(clk) begin
     if clk = '1' then q <= d;
    end if;
end process;
 end;
```

(b) **architecture proc of gates is**
```
 begin
     process(a) begin
        Y1 <= a and b;
        y2 <= a or b;
        y3 <= a xor b;
        y4 <= a nand b;
        y5 <= a nor b;
end process;
 end;
```

(c) **architecturw synth of flop is**
```
begin
   process(clk)
      if rising_edge(clk) then
         q <= d;
end;
```

(d) **architecture synth of priority is**
```
 begin
process(all) begin
    if a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
```

```
    end if;
end process;
end;
```

(e) **architecture synth of divideby3FSM is**

```
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
     process(clk, reset) begin
       if reset then state <= S0;
    elsif rising_edge(clk) then
        state <= nextstate;
     end if;
 end process;
process(state) begin
 case state is
    when S0 =>nextstate <= S1;
    when S1 =>nextstate <= S2;
    when S2 =>nextstate <= S0;
   end case;
 end process;
q <= '1' when state = S0 else '0';
end;
```

(f) **architectbre struct of mux2 is**

```
          component tristate
            port(a: in STD_LOGIC_VECTOR(3 downto 0);
                  en: in STD_LOGIC;
                  y: out STD_LOGIC_VECTOR(3 downto 0));
         end component;
begin
   t0: tristate port map(d0, s, y);
   t1: tristate port map(d1, s, y);
end;
```

(g) **architecture asynchronous of floprs is**

```
        begin
           process(clk, reset) begin
             if reset then
```

```
          q <= '0';
        elsif rising_edge(clk) then
          q <= d;
        end if;
 end process;
process(set) begin
   if set then
     q <= '1';
   end if;
  end process;
end;
```

ЗАПИТАННЯ ДЛЯ СПІВБЕСІДИ

Приклади типових запитань,
які можуть бути поставлені претендентам
під час пошуку роботи,
пов'язаної з проєктуванням цифрових систем.

Запитання 4.1. Напишіть рядок на *HDL*, що реалізує керування 32-бітною шиною *data* сигналом *sel*, отримуючи 32-бітний сигнал *result*. Якщо *sel* істинно, *result = data,* інакше всі біти *result* – нулі.

Запитання 4.2. Поясніть різницю між блокувальними та неблокувальними присвоєннями в *SystemVerilog*. Наведіть приклади.

Запитання 4.3. Що робить цей оператор *SystemVerilog:*

```
result = | (data[15:0] & 16'hC820);
```

5 ЦИФРОВІ ФУНКЦІОНАЛЬНІ ВУЗЛИ

5.1 Вступ

У попередніх розділах ми ознайомилися з проєктування комбінаційних та послідовних схем із використанням логічних виразів, схем і мов опису апаратури. У цьому розділі розглянемо складніші комбінаційні та послідовні функціональні вузли, що застосовуються в цифрових системах. Такі вузли містять арифметичні схеми, лічильники, схеми зсуву, матриці пам'яті та матриці логічних елементів. Ці функціональні вузли корисні як самі собою, а й як демонстрація принципів ієрархічності, модульності та регулярності. Функціональні вузли ієрархічно зібрані з кількох найпростіших компонентів, таких як логічні вентилі, мультиплексори та декодери. Кожен функціональний вузол має чітко визначений інтерфейс і може розглядатися як чорна скринька, коли не така важлива його базова реалізація. Регулярна структура кожного функціонального вузла може розширюватись до будь-якого розміру.

5.2 Арифметичні схеми

Арифметичні схеми є основним функціональним вузлом будь-якого комп'ютера. Комп'ютери та цифрові схеми виконують безліч арифметичних операцій: додавання, віднімання, порівняння, зрушення, множення та ділення. У цьому розділі опишемо апаратну реалізацію всіх перелічених операцій [1–7].

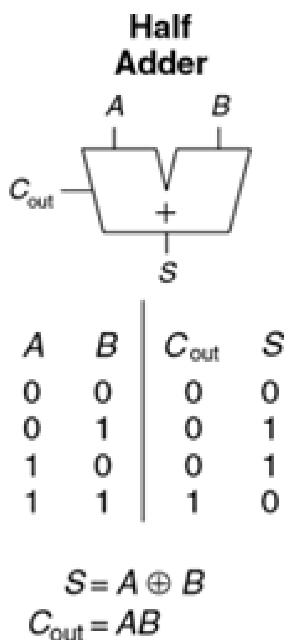
5.2.1 Додавання

Додавання – одна з найпоширеніших операцій у цифрових системах. Для початку розглянемо додавання двох однорозрядних двійкових чисел. Потім розширимо цю процедуру до *N*-розрядних чисел. Суматори демонструють компроміс між швидкістю та складністю реалізації.

Напівсуматор

Спочатку спроєктуємо однорозрядний напівсуматор (*half adder*). Як зображено на рис. 5.1, напівсуматор має два входи (A і B) та два виходи (S і C_{out}). S – це сума A і B. Якщо і A, і B дорівнюють 1, то вихід S має дорівнювати 2, таке число не може бути подано у вигляді одного двійкового розряду. У цьому разі результат вказується разом із сигналом переносу C_{out}

у наступний розряд. Напівсуматор може бути побудовано з елементів *XOR* («що виключає АБО») та *AND* («логічне І»).



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Рисунок 5.1 –Однорозрядний напівсуматор

У багаторозрядному суматорі вихід C_{out} приєднується до входу перенесення наступного розряду. Наприклад, на рис. 5.2 біт перенесення показано зверху, він є виходом C_{out} однорозрядного суматора 1-го розряду та входом C_{in} суматора наступного розряду. Однак у напівсуматора немає входу перенесення C_{in} для зв'язку з виходом C_{out} попереднього розряду. У повному суматорі, що розглядається у наступному розділі, такий вхід є.

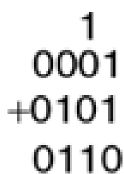
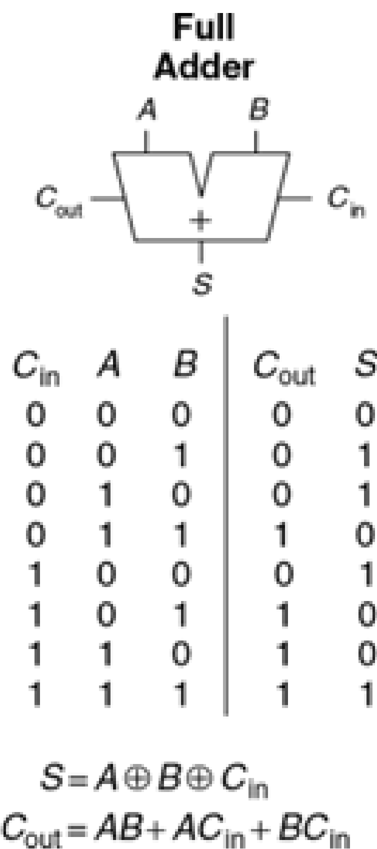


Рисунок 5.2 – Біт перенесення

Повний суматор

Як подано на рис. 5.3, повний суматор (*full adder*), описаний у підрозділі 2.1, має вхід перенесення C_{in}. На рисунку також наведено рівняння для *S* та C_{out}.



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Рисунок 5.3 – Однорозрядний повний суматор

Суматор із розповсюдженим перенесенням

N-розрядний суматор складає 2 *N*-розрядні числа (*A* і *B*), а також вхідний сигнал перенесення C_{in} і формує *N*-розрядний результат *S* і вихід перенесення C_{out}. Такий суматор називається суматором із розповсюдженим перенесенням (*carry propagate adder, CPA*), оскільки вихідний сигнал перенесення одного розряду переходить у наступний розряд. Умовна познака такого суматора зображена на рис. 5.4. Вона аналогічна познаці повного суматора, за винятком того, що входи / виходи *A, B, S* є шинами, а не окремими розрядами. Найпоширенішими реалізаціями *CPA* є суматори з послідовним перенесенням (*ripple-carry adders*), з прискореним перенесенням (*carry-lookahead adders*) та префіксні суматори (*prefix adders*).

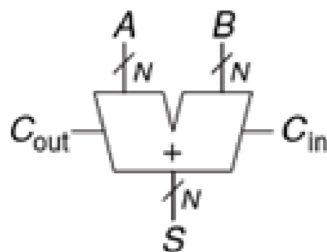


Рисунок 5.4 – Суматор з розповсюдженим перенесенням

Суматори з послідовним перенесенням

Найпростіший спосіб реалізації *N*-розрядного суматора – це об’єднання в ланцюг *N* повних суматорів. Вихід C_{out} деякого розряду надходитиме на вхід C_{in} наступного розряду і т. д. (див. рис. 5.5 для 32-розрядного суматора).

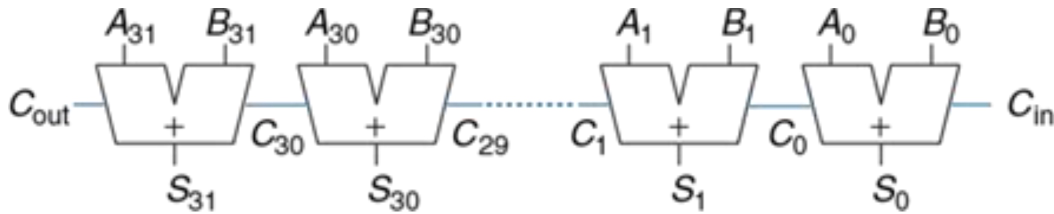


Рисунок 5.5 – 32-розрядний суматор із послідовним перенесенням

Така схема називається суматором із послідовним перенесенням (*ripple-carry adder*). У процесі її проєктування використовується принцип модульності та регулярності: модуль повного суматора багаторазово застосовується для формування більшої системи. Такий суматор має недолік: його швидкість падає зі збільшенням числа розрядів *N*. S_{31} залежить від C_{30}, що залежить від C_{29}, який зі свого боку залежить від C_{28} і так далі до C_{in} (див. рис. 5.5). Перенесення відбувається крізь весь ланцюг. Затримка такого суматора (*triple*) збільшується разом із кількістю розрядів, як показано на рівнянні (5.1), де *tFA* – це затримка повного суматора.

$$\text{tripple} = \text{NtFA}. \tag{5.1}$$

Суматори із прискореним перенесенням

Основною причиною того, що великі суматори з послідовним перенесенням працюють повільно, є те, що сигнал перенесення має пройти крізь усі біти суматора. Суматори з прискореним перенесенням (*carry-lookahead adder, CLA*) – це інший тип суматорів із розповсюдженим перенесенням, що розв’язує цю проблему способом поділу суматора на блоки і реалізує схему так, щоб визначити вихідний сигнал перенесення блоку як тільки стало відоме його вхідне перенесення. Отже, ми дивимося вперед крізь блоки й не чекаємо перенесення крізь усі повні суматори всередині блоку. Наприклад, 32-розрядний суматор може бути розділений на вісім чотирирозрядних суматорів.

Суматори з прискореним перенесенням використовують сигнали генерації (*G*) та поширення (*P*), що описують, як блок (або розряд) визначає вихід перенесення. *i*-й розряд суматора генерує перенесення, якщо він видає перенесення своєму виходу, незалежно від наявності перенесення на вході. *i*-й розряд суматора генерує C_i в тому разі, якщо і A_i, і B_i дорівнюють 1.

Отже, сигнал генерації G_i можна обчислити як $G_i = A_iB_i$. Розряд називається розповсюджувальним, якщо вихідний сигнал перенесення з’являється за наявності вхідного перенесення. Розряд поширюватиме вхідний сигнал перенесення, $C_{i–1}$, якщо або A_i, або B_i дорівнюють 1. Так, $P_i = A_i + B_i$. Використовуючи ці визначення, можемо переписати логіку формування сигналу перенесення певного розряду. *i*-й розряд суматора формуватиме вихідний сигнал перенесення C_i, якщо він або генерує перенесення G_i, або поширює вхідне перенесення $P_i\ C_{i–1}$. У вигляді рівняння це можна записати так:

$$C_i = A_i\ B_i + (A_i + B_i)\ C_{i–1} = G_i + P_i\ C_{i–1}. \quad (5.2)$$

Визначення сигналів генерації та розповсюдження належать і до багаторозрядних блоків. Блок називається генерувальним перенесення, якщо він створює вихідне перенесення незалежно від вхідного сигналу перенесення цього блоку. Блок називається розповсюджувальним перенесення, якщо вихідне перенесення виникає внаслідок надходження вхідного перенесення. $G_{i:j}$ та $P_{i:j}$ визначаються, як сигнали генерації та розповсюдження для блоків, що охоплюють розряди з *i* до *j*.

Блок генерує перенесення, якщо найстарший розряд генерує перенесення або якщо старший розряд розповсюджує перенесення, згенероване попереднім розрядом і т.д.

$$G_{3:0} = G_3 + P_3\ (G_2 + P_2\ (G_1 + P_1\ G_0)). \quad (5.3)$$

Блок розповсюджує перенесення, якщо всі вхідні до нього розряди це перенесення поширюють. Логіка розповсюдження для блоку, що охоплює розряди з 0 до 3:

$$P_{3:0} = P_3\ P_2\ P_1\ P_0. \quad (5.4)$$

За допомогою блокових сигналів генерації та розповсюдження можна швидко визначити вихідне перенесення блоку C_i, використовуючи його вхідне перенесення C_j.

$$C_i = G_{i:j} + P_{i:j}\ C_j. \quad (5.5)$$

На рис. 5.6, *a* зображено 32-розрядний суматор із прискореним перенесенням, що містить вісім чотирирозрядних блоків. Кожен блок містить чотирирозрядний суматор із послідовним перенесенням та схему прискореного перенесення, яка визначає вихідний сигнал перенесення блоку по вхідному (див. рис. 5.6, *b*). На рисунку не показано елементи І та АБО, необхідні для обчислення однорозрядних сигналів генерації та розповсюдження G_i та P_i по A_i та B_i. Суматор із прискореним перенесенням демонструє модульність і регулярність.

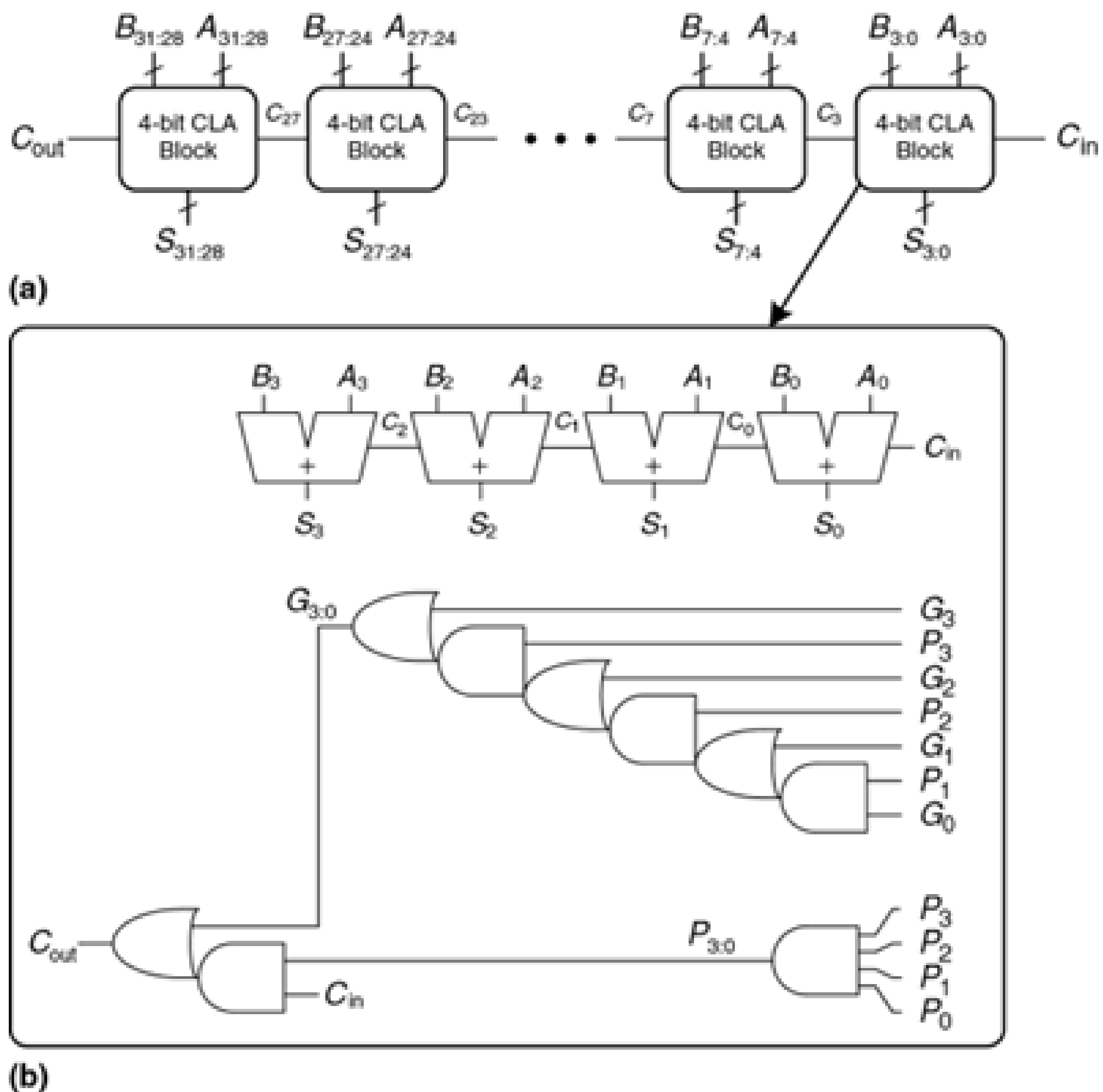


Рисунок 5.6 – 32-розрядний суматор
з прискореним перенесенням (a) та його чотирибітний блок (b)

Усі блоки суматора одночасно обчислюють однобітові та блокові сигнали генерації та розповсюдження. Критичний шлях починається з обчислення G_0 та $G_{3:0}$ у першому блоці суматора. Сигнал C_{in} потім поширюється до C_{out} через логічні елементи І/АБО всіх блоків. Для великого суматора це відбувається набагато швидше, ніж розповсюдження перенесення крізь кожний наступний розряд суматора. І, нарешті, критичний шлях крізь останній блок містить невеликий суматор із послідовним перенесенням. Отже, *N*-розрядний суматор, розділений на *k*-розрядні блоки, має затримку

$$tCLA = tpg + tpg_block + (N/k–1)tAND_OR + ktFA, \qquad (5.6)$$

де *tpg* – затримка окремих логічних елементів генерації / розповсюдження (одиночних логічних елементів І/АБО) у процесі генерації *P* і *G*.

$C_{in} – C_{out}$, що містить логіку І/АБО *k*-розрядного *CLA*-блоку. За умови $N > 16$ такий суматор працює набагато швидше, ніж суматор із послідовним

перенесенням. Однак затримка суматора, як і раніше, лінійно зростає зі зростанням N.

Приклад 5.1.

Затримки суматорів

Порівняємо затримки 32-розрядного суматора з послідовним перенесенням та 32-розрядного суматора з прискореним перенесенням, що має чотирирозрядні блоки. Припустимо, що затримка кожного двовходового логічного елемента становить 100 пс, а затримка повного суматора – 300 пс.

Виконання. Відповідно до формули (5.1) затримка поширення 32-розрядного суматора з послідовним перенесенням дорівнює 32×300 пс = 9,6 нс.

У суматора з прискореним перенесенням tpg=100 пс, $tpg_block = 6{\times}100$ пс = =600 пс, та $tAND_OR = 2{\times}100$ пс = 200 пс. Відповідно до рівняння (5.6) затримка поширення 32-розрядного суматора з прискореним перенесенням, що містить чотирирозрядних блоків, дорівнює 100 пс +600 пс +(32/4–1)×200 пс + + (4 × 300) пс = 3,3 нс, що майже втричі менше, ніж у суматора з послідовним перенесенням.

Префіксний суматор

Префіксний суматор розвиває логіку генерації та розповсюдження суматора з прискореним перенесенням для ще швидшого виконання операції складання. Спочатку він обчислює G і P для пар розрядів, далі для блоків з чотирьох розрядів, потім для блоків з 8, 16 тощо розрядів, поки сигнал генерації не буде відомий для кожного розряду. Сума визначається всіма сигналами генерації.

Інакше кажучи, стратегія префіксного суматора полягає в обчисленні вхідного сигналу перенесення C_{i-1} для кожного розряду так швидко, наскільки це можливо. Потім за формулою обчислюється сума:

$$S_i = (A_i \wedge B_i) \wedge C_{i-1}. \quad (5.7)$$

Визначимо розряд $i = -1$ обчислення $C_{in:\, G-1} = C_{in}$ і P–1 =0.

Отже, $C_{i-1} = G_{i-1:-1}$, оскільки вихідний сигнал перенесення розряду i–1 буде активним, якщо блок, що охоплює розряди від i–1 до –1, генерує перенесення. Отже, можемо переписати рівняння (5.7) як

$$S_i = (A_i \wedge B_i) \wedge G_{i-1:-1}. \quad (5.8)$$

Отже, основною проблемою є швидке обчислення всіх блокових сигналів генерації $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, . . . , $G_{N-2:-1}$.

Ці сигнали разом з $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, ..., $P_{N-2:-1}$ називають префіксними.

На рис. 5.7 зображено 16-розрядний префіксний суматор. Його робота починається з попереднього формування сигналів P_i та G_i для всіх розрядів A_i та B_i з використанням елементів І та АБО. Потім застосовується $log2N = 4$ рівня чорних осередків для формування префіксів $G_{i:j}$ та $P_{i:j}$. Чорний осередок приймає входи з верхньої частини блоку, що охоплює біти $i:k$, і нижньої частини блоку, що охоплює біти $k–1:j$. Потім ці частини об'єднуються для формування сигналів генерації та розповсюдження всього блоку, що охоплює біти $i:j$. Використовуючи рівняння (5.9) та (5.10), отримаємо

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}; \tag{5.9}$$

$$P_{i:j} = P_{i:k}\, P_{k-1:j}\,. \tag{5.10}$$

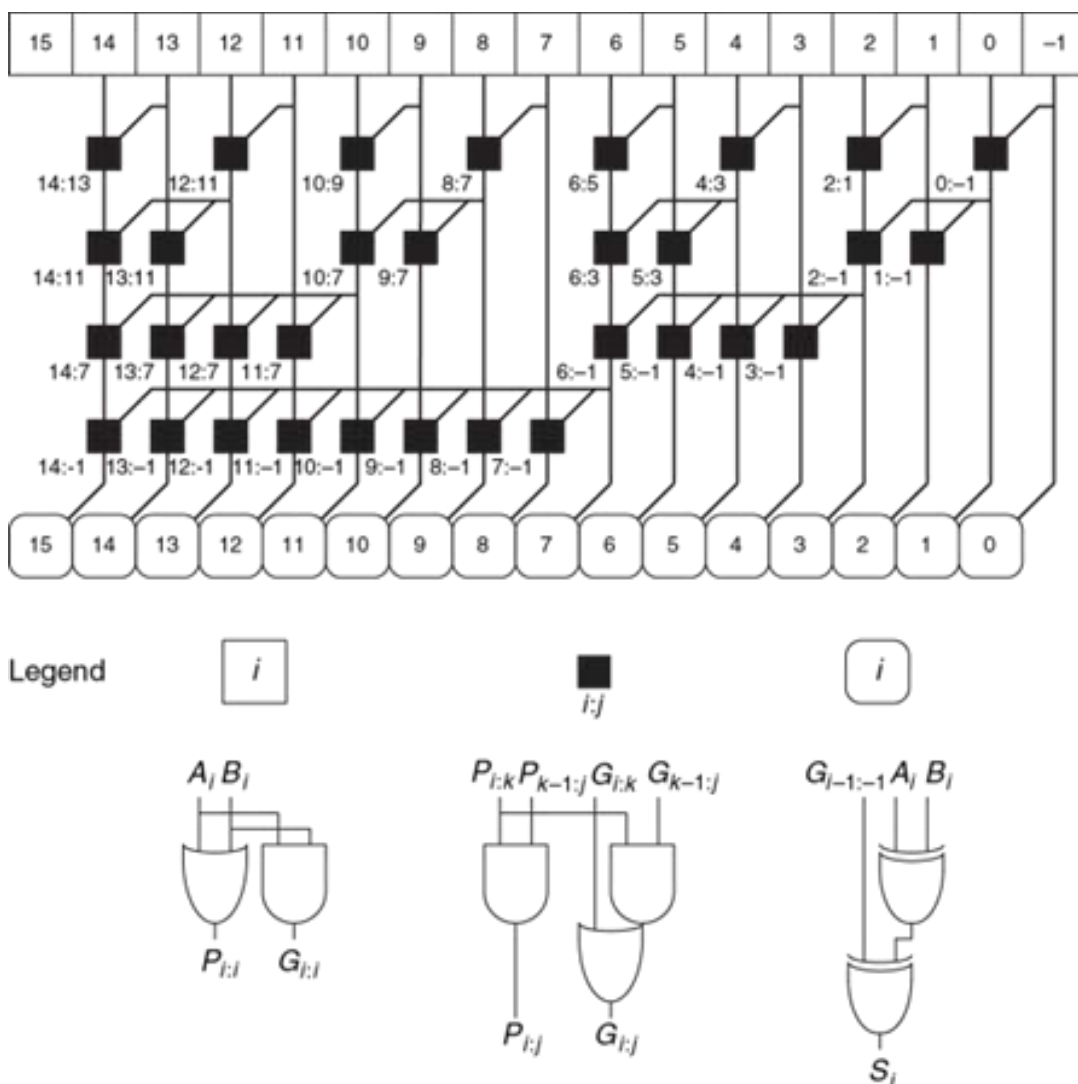


Рисунок 5.7 – 16-розрядний префіксний суматор

Іншими словами, блок, що охоплює біти $i:j$, генеруватиме сигнал перенесення, якщо верхня частина генерує перенесення або якщо вона поширює перенесення, згенероване в нижній частині. Блок розповсюджуватиме

перенесення, якщо і верхня, і нижня частини поширюють його. У результаті префіксний суматор обчислює суму на основі рівняння (5.8).

Отже, затримка префіксного суматора досягає значення, що зростає з числом розрядів суматора логарифмічно, а не лінійно. Прискорення значне, особливо для суматорів, що мають 32 і більше розрядів. Такий суматор застосовує значно більше апаратних засобів, ніж простий суматор із прискореним перенесенням. Мережа чорних осередків називається префіксним деревом.

Основний принцип використання префіксного дерева, за умови якого час обчислень збільшується логарифмічно із зростанням кількості входів, є потужною технологією. У разі деякого вміння цей принцип може бути застосовано для інших схем (див., наприклад, вправу 5.7).

Критичний шлях *N*-розрядного префіксного суматора передбачає попереднє обчислення P_i та G_i, за яким слідує *log2N* каскадів чорних осередків для отримання всіх префіксів. Потім сигнали $G_{i-1:-1}$ обробляються фінальними елементами, «що виключають АБО» в нижній частині схеми для отримання сигналу S_i.

Затримка *N*-розрядного префіксного суматора дорівнює

$$tPA = tpg + log2N(tpg_prefix) + tXOR, \quad (5.11)$$

де *tpg_prefix* – затримка чорного префіксного осередку.

Приклад 5.2.

Затримка префіксного суматора

Обчислимо затримку 32-розрядного префіксного суматора. Припустимо, що затримка кожного двовходового логічного елемента дорівнює 100 пс.

Виконання. Затримка поширення кожного чорного префіксного осередку дорівнює *tpg_prefix* = 200 пс (затримки двох логічних елементів). Отже, використовуючи рівняння (5.11), затримка поширення 32-розрядного префіксного суматора дорівнює 100 пс + *log*2(32) + 200 пс + 100 пс + 1,2 нс, що приблизно втричі менше, ніж у суматора з прискореним перенесенням із прикладу 5.1. Насправді вигода не така велика, але префіксні суматори справді працюють суттєво швидше, ніж будь-які альтернативні.

Висновок

У цьому підрозділі розглянуто напівсуматор, повний суматор і три типи суматорів із розповсюдженим перенесенням: суматори з послідовним перенесенням, прискореним перенесенням і префіксний суматор. Швидкі суматори використовують більше апаратних засобів і, отже, є дорожчими

та енерговитратними. Усе це необхідно брати до уваги під час виборів потрібного суматора в процесі проєктування.

Мови опису апаратури уможливлюють використання операції + для визначення суматора з перенесенням, що поширюється. Сучасні засоби синтезу обирають із безлічі можливих реалізацій проєкту найбільш дешеву та просту, що задовольняє вимоги щодо швидкості. Це значно полегшує роботу прєктувальника. У *HDL*-прикладі 5.1 за допомогою мов опису апаратури описано суматор з перенесенням, що має вхід і вихід перенесення.

***HDL*-приклад 5.1.**
Суматор

SystemVerilog

```
module adder #(parameter N = 8)
     (input logic [N-1:0] a, b,
      input logic cin,
      output logic [N-1:0] s,
      output logic cout);
   assign {cout, s} = a + b + cin;
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity adder is
        generic(N: integer := 8);
        port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
              cin: in STD_LOGIC;
              s: out STD_LOGIC_VECTOR(N-1 downto 0);
              cout: out STD_LOGIC);
end;
architecture synth of adder is
    signal result: STD_LOGIC_VECTOR(N downto 0);
 begin
     result <= ("0" & a) + ("0" & b) + cin;
     s <= result(N-1 downto 0);
     cout <= result(N);
end;
```

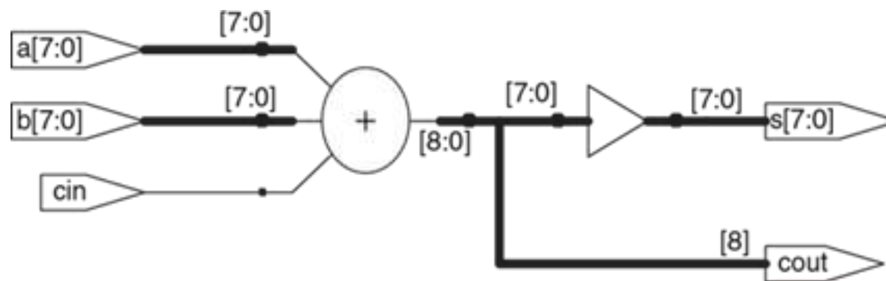


Рисунок 5.8 – Синтезований суматор

5.2.2 Віднімання

У п. 1.4.6 показано, що суматори можуть складати позитивні та негативні числа, застосовуючи подання числа додаткового коду. Віднімання здійснюється схоже: змінюється знак другого числа, потім числа складаються. Зміна знака числа додаткового коду проводиться способом інверсії бітів і додавання 1.

Для обчислення $Y = A - B$ спочатку створюється додатковий код числа B: інвертуються розряди й додається 1; $-B = \overline{B} + 1$. Отримане значення складається з A. Ця сума може бути отримана одним суматором із поширеним перенесенням способом складання $A + \overline{B}$, якщо $Cin = 1$. На рис. 5.9 зображено умовну познаку пристрою віднімання та базову апаратну реалізацію для обчислення $Y = A - B$.

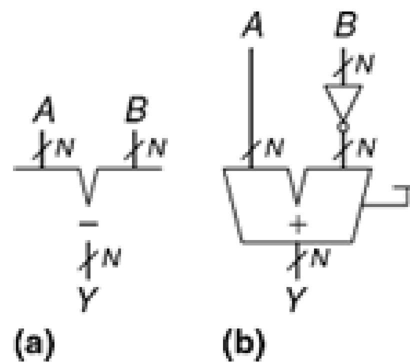


Рисунок 5.9 – Пристрій віднімання: a – умовна познака; b – реалізація

HDL-приклад 5.2.

Пристрій віднімання

SystemVerilog

```
module subtractor #(parameter N = 8)
          (input logic [N-1:0] a, b,
           output logic [N-1:0] y);
assign y = a − b;
 endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity subtractor is
	generic(N: integer := 8);
	port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
		y: out STD_LOGIC_VECTOR(N-1 downto 0));
end;
architectura synth of subtractor is
 begin
	y <= a – b;
end;
```

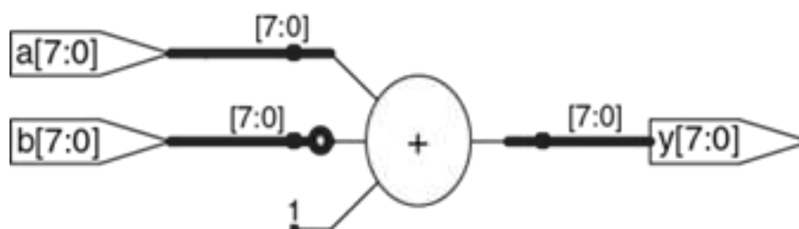


Рисунок 5.10 – Синтезований пристрій віднімання

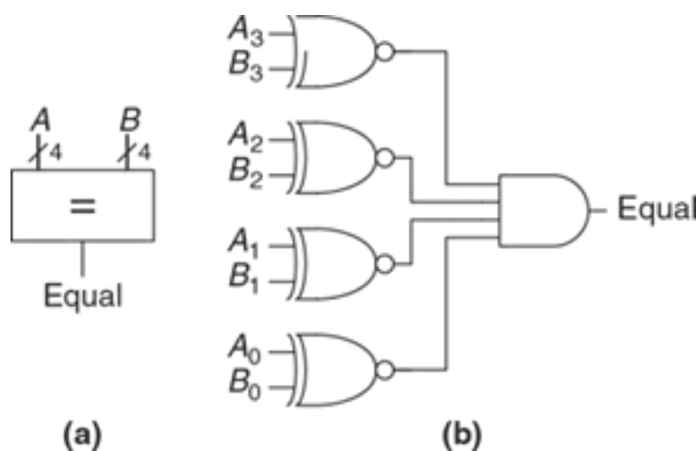
5.2.3 Компаратори

Компаратори визначають, чи два двійкових числа є рівними або одне з них більше / менше, ніж інше. Компаратор отримує два *N*-розрядні двійкові числа *A* і *B*. Існує два типи компараторів.

Компаратор рівності видає один вихідний сигнал, що показує, чи рівні *A* і *B* (*A*==*B*). Компаратор величини видає один і більше вихідних сигналів, показуючи відношення величин *A* і *B*.

Компаратор рівності має найпростішу апаратну реалізацію. На рис. 5.11 зображено познаку та реалізацію чотирирозрядного компаратора рівності. Спочатку за допомогою логічних елементів *XNOR* він перевіряє, чи є відповідні розряди *A* і *B* рівними. Значення будуть рівними, якщо всі відповідні розряди рівні [1–7].

Як подано на рис. 5.12, компаратор величини обчислює *A–B* і аналізує знак (найстарший розряд) результату. Якщо результат негативний (найстарший розряд = 1), то *A* менше від *B*. Інакше *A* більше, ніж *B*, або дорівнює йому.



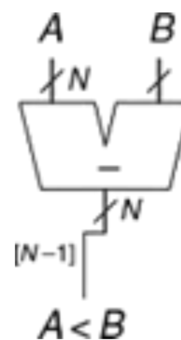


Рисунок 5.11 – Чотирирозрядний компаратор рівності: a – умовна познака; b – реалізація

Рисунок 5.12 – *N*-розрядний компаратор величини

HDL-приклад 5.3 демонструє застосування цих типів компараторів.

***HDL*-приклад 5.3.**
Компаратори

SystemVerilog

```
module comparator #(parameter N = 8)
            (input logic [N-1:0] a, b,
             output logic eq, neq, lt, lte, gt, gte);
 assign eq = (a == b);
 assign neq = (a! = b);
 assign lt = (a <b);
 assign lte = (a <= b);
 assign gt = (a> b);
 assign gte = (a> = b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity comparators is
        generic(N: integer : = 8);
         port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
                eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;
architectura synth of comparator is
begin
```

```
    eq <= '1' when (a = b) else '0';
    neq <= '1' when (a /= b) else '0';
    lt <= '1' when (a < b) else '0';
    lte <= '1' when (a <= b) else '0';
    gt <= '1' when (a > b) else '0';
    gte <= '1' when (a >= b) else '0';
end;
```

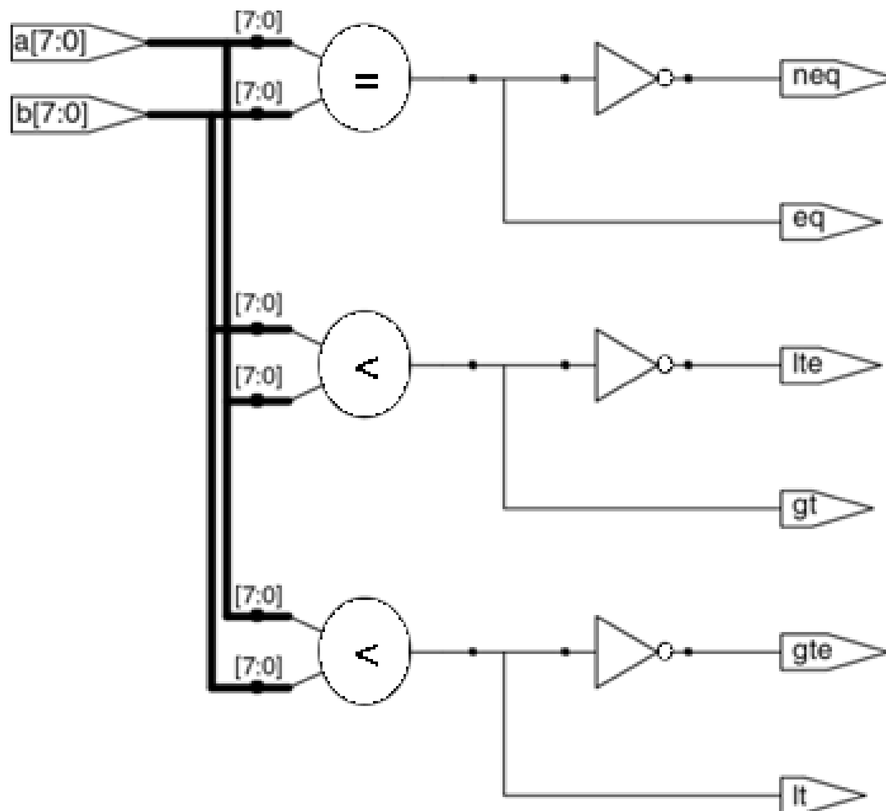


Рисунок 5.13 – Синтезований компаратор

5.2.4 АЛП

Арифметико-логічний пристрій (АЛП, *Arithmetic / Logical Unit, ALU*) об'єднує різні арифметичні та логічні операції в одному вузлі. Наприклад, типове АЛП може виконувати додавання, віднімання, порівняння величин, операції І та АБО. АЛП містить ядро більшості комп'ютерних систем.

На рис. 5.14 зображено умовну познаку *N*-розрядного АЛП з *N*-розрядними входами та виходами. В АЛП надходить сигнал керування *F*, що визначає, яку функцію необхідно виконати. Зазвичай, сигнали керування позначають блакитним кольором, щоб розрізняти їх від сигналів даних. У табл. 5.1 перелічені типові функції, що виконує АЛП. Функція *SLT* використовується для порівняння за значенням і розглядатиметься нижче в цьому розділі.

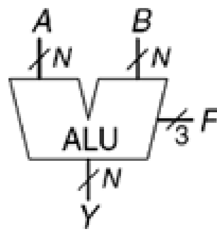


Рисунок 5.14 – Умовна познака арифметико-логічного пристрою (АЛП)

Таблиця 5.1 – Операції АЛП

***F*2:0**	**Function**
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

На рис. 5.15 подано реалізацію вузла АЛП. Він містить *N*-бітний суматор, *N* двовходові логічні елементи І та двовходові логічні елементи АБО. Також він містить інвертори та мультиплексор для інверсії бітів, коли сигнал керування *F*2 активний. Мультиплексор з організацією 4:1 обирає необхідну функцію, зважаючи на сигнали керування *F*1:0.

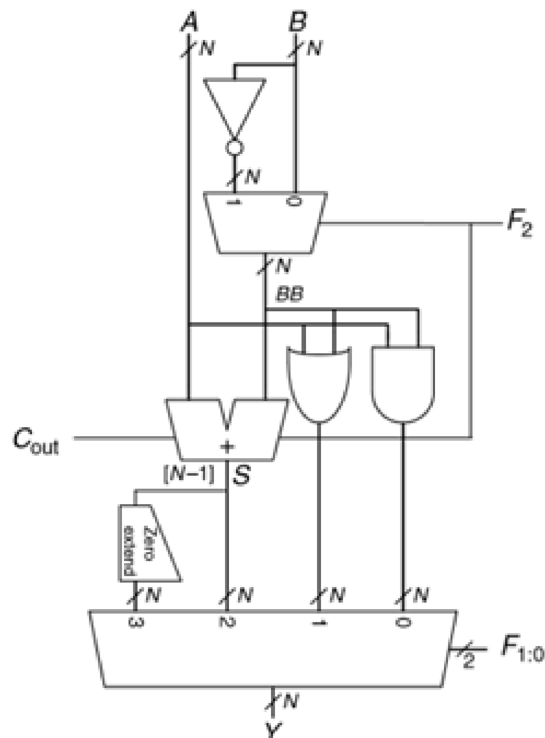


Рисунок 5.15 – *N*-розрядний АЛП

Точніше кажучи, арифметичні та логічні блоки АЛП оперують з A та BB. BB дорівнює B або $\overline{B}$ залежно від $F2$. Якщо $F1{:}0 = 00$, вихідний мультиплексор обирає операцію A AND BB. Якщо $F_{1:0} = 01$, тоді АЛП обчислює A OR BB. У разі $F_{1:0} = 10$ АЛП виконає додавання або віднімання. Зауважте, що $F2$ також є входом перенесення суматора. У використанні додаткового коду $\overline{B} + 1 = -B$. За умови $F2 = 0$ АЛП обчислює $A + B$, якщо $F2 = 1$, тоді буде обчислена різниця $A + \overline{B} + 1 = A - B$.

Коли $F_{2:0} = 111$, АЛП виконує операцію *SLT* (*set if less than* – встановити, якщо менше, ніж). Якщо $A < B$, $Y = 1$. В іншому разі $Y = 0$. Отже, Y встановлюється в 1, якщо A менше, ніж B.

Операція *SLT* виконується способом обчислення $S = A - B$. Якщо S негативне (тобто встановлено знаковий біт), то A менше за B. Модуль доповнення нулями створює N-розрядний вихідний сигнал, поєднуючи однобітний вхід з нулем у найстарших розрядах. Біт знака (розряд N–1) змінної S буде входом модуля установлення нуля.

Приклад 5.3.

Встановити, якщо менше

Побудуємо 32-розрядний АЛП з операцією *SLT*. Припустимо, $A = 25_{10}$, $B = 32_{10}$. Знайдемо сигнал керування та вихідний сигнал Y.

Виконання. Оскільки $A < B$, то Y має дорівнювати 1. Для *SLT* $F2{:}0 = 111$. Разом із сигналом $F2 = 1$ це конфігурує модуль суматора як пристрій із вихідним сигналом S: $25_{10} - 32_{10} = -7_{10} = 1111..1001_2$. З огляду на $F_{1:0} = 11$ вихідний сигнал мультиплексора буде $Y = S_{31} = 1$.

Деякі АЛП мають спеціальні вихідні сигнали, що називаються прапорами, які показують інформацію про вихід АЛП. Наприклад, прапор переповнення показує, що результат роботи суматора переповнився. Прапор нуля показує, що вихід АЛП встановився 0.

Опис N-розрядного АЛП на *HDL* залишено для вправи 5.9. Існує чимало варіацій базового АЛП, що виконують такі функції, як *XOR* або порівняння на рівність.

5.2.5 Схеми зсуву та циклічного зсуву

Схеми зсуву та схеми циклічного зсуву переміщують біти і, отже, множать або ділять число на степінь 2. Відповідно до назви схеми зсуву пересувають розряди двійкового числа вліво або вправо на певну кількість позицій. Існує кілька видів таких схем.

Розглянемо їх.

● **Логічні схеми зсуву** зсувають число вліво (*LSL*) або вправо (*LSR*) і заповнюють порожні розряди нулями.

Наприклад, 11001 *LSR* 2 = 00110; 11001 *LSL* 2 = 00100.

● **Арифметичні схеми зсуву** діють так само, як і логічні, але за умови зрушення праворуч вони заповнюють найбільш значущі розряди значенням знакового біта вихідного числа. Це необхідно у множенні та розподілі чисел зі знаком (див. п. 5.2.6 та 5.2.7). Арифметичний зсув вліво (*ASL*) працює так само, як і логічний (*LSL*).

Наприклад, 11001 *ASR* 2 = 11110; 11001 *ASL* 2 = 00100.

● **Схеми циклічного зсуву** зсувають число по колу так, що порожні місця заповнюються розрядами, які висунуті з іншого кінця.

Наприклад, 11001 *ROR* 2 = 01110; 11001 *ROL* 2 = 00111.

N-розрядна схема зсуву може бути побудована з мультиплексорів *N*:*N*:1. Вхід зсувається на 0–*N*–1 розрядів залежно від значення *log2N* ліній вибору. На рис. 5.16 зображено умовну позначку та апаратну реалізацію чотирирозрядної схеми зсуву. Оператори <<, >> та >>> зазвичай позначають зсув вліво, логічний зсув вправо та арифметичний зсув вправо відповідно. Залежно від значення дворозрядної величини зсуву *shamt*1:0, на вихід *Y* надходить вхідний сигнал *A*, зрушений на 0–3 розряду. Для всіх схем зсуву, якщо $shamt_{1:0}$ = 00, *Y* = *A*. У вправі 5.14 розглядається розроблення схем циклічного зсуву.

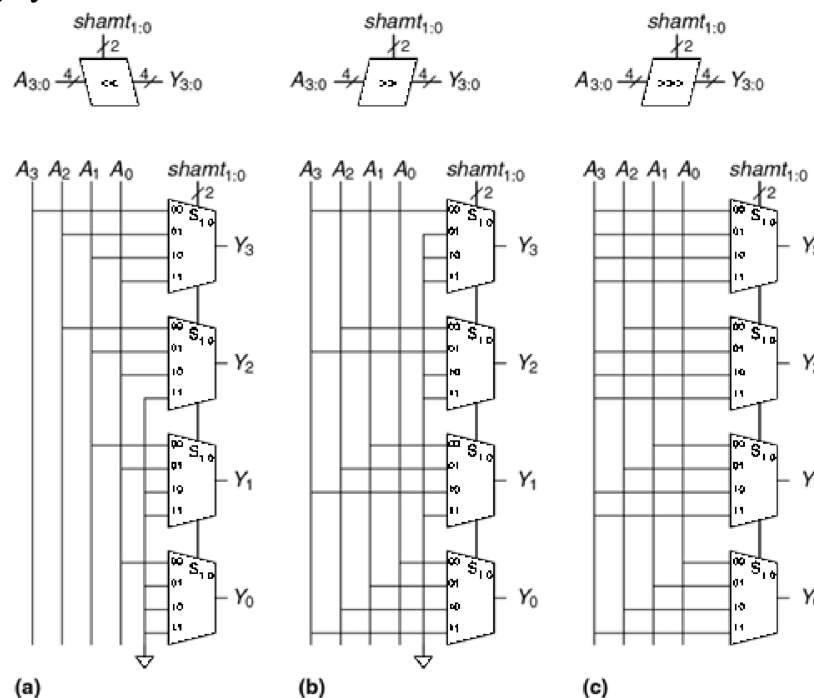


Рисунок 5.16 – Чотирирозрядні схеми зсуву:
a – зсув вліво; b – логічний зсув вправо; c – арифметичний зсув вправо

Зсув вліво – це окремий випадок множення. Зсув вліво на N бітів множить число на $2N$. Наприклад, 000011_2 << 4 = 110000_2 рівнозначно $3_{10} \times 2^4 = 48_{10}$.

Арифметичний зсув вправо – це особливий випадок поділу. Арифметичний зсув вправо на N бітів ділить число на 2^N. Наприклад, 11100_2>>> 2 =11111_2 рівнозначно $-4_{10}/2^2 = 1_{10}$.

5.2.6 Множення

Множення беззнакових двійкових чисел подібне до десяткового множення, проте воно оперує тільки з одиницями й нулями. На рис. 5.17 порівнюється множення двійкових і десяткових чисел.

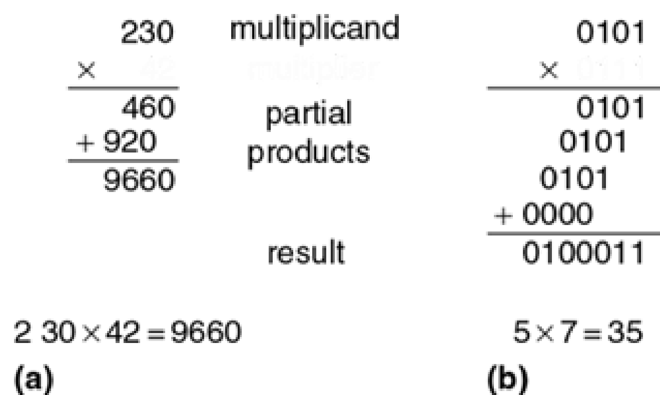


Рисунок 5.17 – Множення: a – десяткове; b – двійкове

В обох випадках часткові здобутки формуються способом множення окремих розрядів множника на все множинне. Зрушені часткові добутки потім складаються і маємо результат.

Загалом, помножувач *NxN* перемножує два *N*-розрядні числа й породжує 2*N*-розрядний результат. Часткові здобутки за умови двійкового множення дорівнюють або множнику, або нулю. Множення одного розряду двійкових чисел рівнозначно операції І, для формування часткових добутків використовують логічні елементи І.

На рис. 5.18 зображено умовну познаку, подано функціональний опис та апаратну реалізацію помножувача 4×4. Помножувач отримує множину й множник A і B та обчислює добуток P. На рис. 5.18, b показано, як формуються часткові перемноження. Кожен частковий добуток дорівнює результату операцій І, аргументами яких є окремі розряди множника (B_3, B_2, B_1 або B_0) і всі розряди множини (A_3, A_2, A_1, A_0). Для N-розрядних операндів існуватиме N часткових перемножень і N–1 каскадів (стадій) однорозрядних суматорів. Наприклад, для помножувача 4×4 частковий

добуток першого ряду – це B_0 *AND* (A_3, A_2, A_1, A_0). Цей частковий добуток додається до зрушеного другого часткового добутку B_1 *AND* (A_3, A_2, A_1, A_0). Наступні ряди логічних елементів І та суматорів формують і додають часткові добутки, що залишилися [1–7].

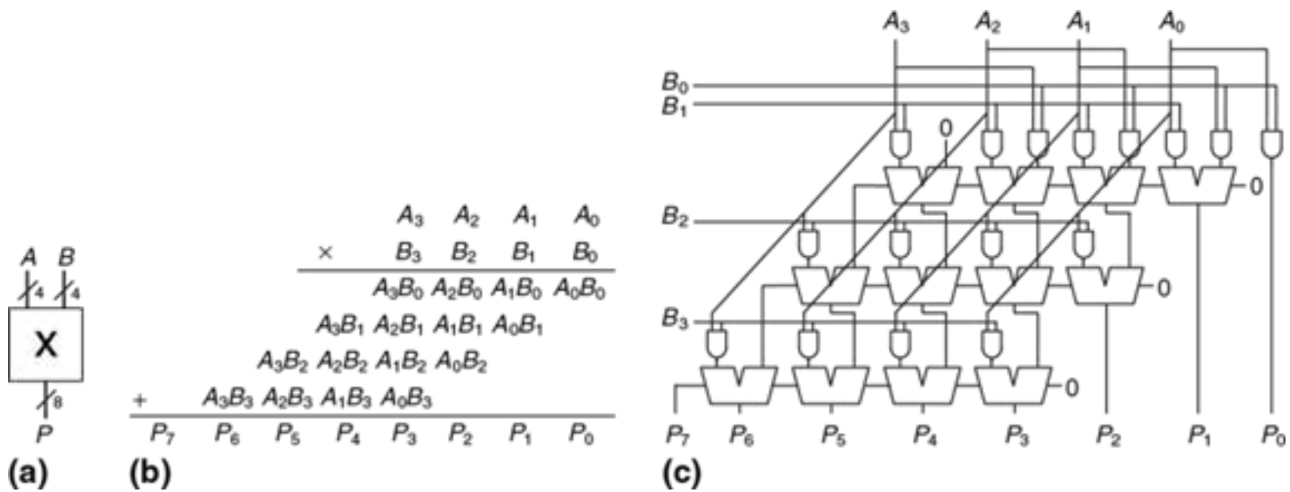


Рисунок 5.18 – Помножувач 4×4: a – умовна познака; b – функції; c – реалізація

Опис помножувача наведено в *HDL*-прикладі 5.4. Так само як і для суматорів, існує безліч реалізацій помножувачів з різними компромісами між швидкістю та вартістю. Інструментальні засоби синтезу можуть обирати найбільш підхожу реалізацію за заданими часовими обмеженнями.

***HDL*-приклад 5.4.**
Помножувач

SystemVerilog

```
module multiplier #(parameter N = 8)
                  (input logic [N-1:0] a, b,
                   output logic [2 * N-1:0] y);
   assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity multiplier is
   generic(N: integer := 8);
   port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
```

```
    y: out STD_LOGIC_VECTOR(2*N–1 downto 0));
end;
architecture synth of multiplier is
begin
    y <= a * b;
end;
```

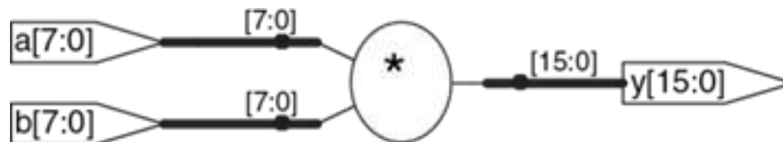


Рисунок 5.19 – Синтезований помножувач

5.2.7 Ділення

Двійкове ділення *N*-розрядних беззнакових чисел у діапазоні [0, 2^{N-1}] може бути виконано з використанням такого алгоритму:

```
R′ = 0
for i = N-1 to 0
    R = {R′ << 1, Ai}
    D = R - B
     if D < 0 then Qi = 0, R′ = R // R < B
      else Qi = 1, R′ = D // R ≥ B
R = R′
```

Частковий залишок *R* ініціалізується 0. Найбільш значущий розряд ділимого *A* потім стає найменш значущим розрядом *R*. Дільник багаторазово віднімається з часткового залишку й визначається знак різниці *D*. Якщо вона негативна (тобто знаковий розряд дорівнює 1), то частковий розряд Q_i дорівнює 0, і різниця відкидається. Інакше Q_i дорівнює 1 і частковий залишок оновлюється, він стає рівним різниці *D*. Потім частковий залишок подвоюється (зсувається вліво на один розряд) – і процес повторюється. Результат задовольняє умову $A/B = Q+R/B$.

На рис. 5.20 зображено схему чотирирозрядної матриці поділу.

Схема обчислює *A/B*, і вихід видає часткове *Q* й залишок *R*. На вставці зображена умовна познака й схеми кожного блоку в матриці поділу. Сигнал *N* показує, чи є результат *R–B* негативним. Це визначається вихідним сигналом перенесення C_{out} крайнього лівого блоку в ряду, який є знаком різниці.

Затримка *N*-розрядної матриці поділу збільшується пропорційно N^2, оскільки перенесення має пройти крізь усі *N* каскадів у ряду перед тим,

як визначиться знак і мультиплексор обере *R* або *D*. Це повторюється для всіх *N* рядів. Поділ – дуже повільна й дорога операція в апаратній реалізації, тому її необхідно використовувати якомога рідше.

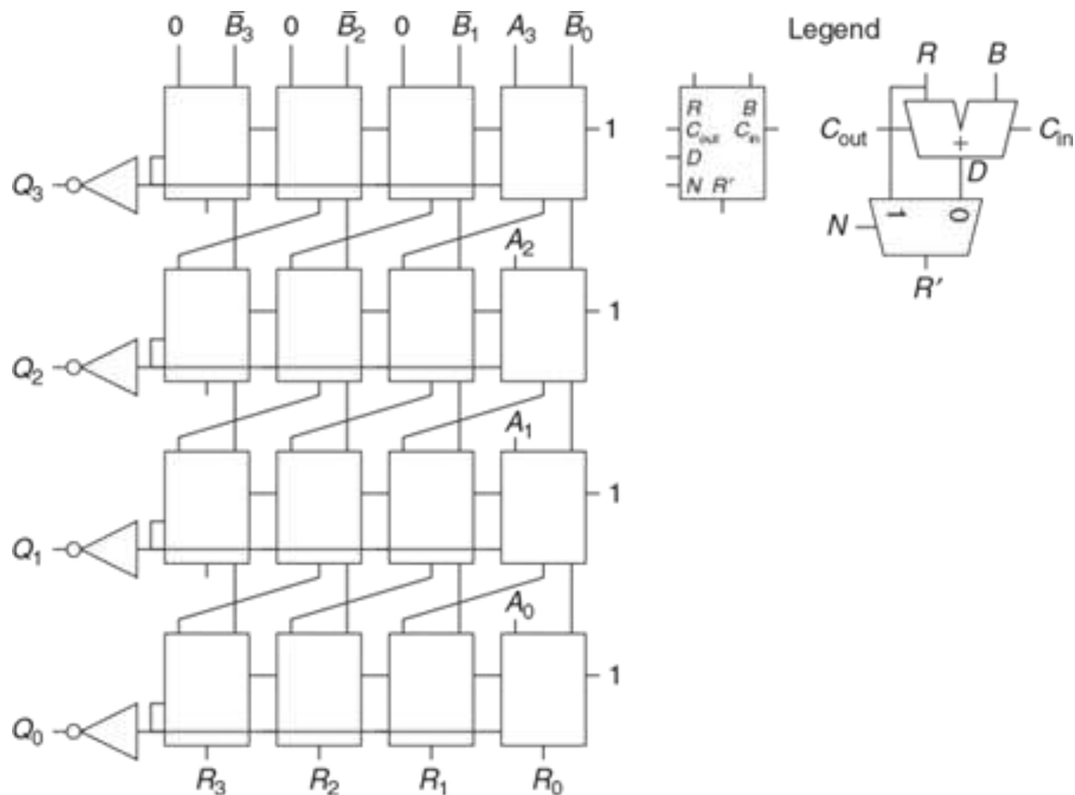


Рисунок 5.20 – Матриця поділу

5.2.8 Додаткова література

Комп'ютерна арифметика може бути предметом дослідження. Підручник *Digital Arithmetic* Ерцеговича та Ланга (*Ercegovac & Lang*) – чудовий огляд цього напряму. *CMOS VLSI Design* Весте й Харріса (*Weste & Harris*) охоплює проєктування високопродуктивних схем для арифметичних операцій.

5.3 Подання чисел

Комп'ютер працює як із цілими, так і з дробовими числами. Дотепер ми розглядали лише подання знакових і беззнакових цілих чисел, описаних у підрозділі 1.4. У цьому підрозділі описано подання чисел з фіксованою та з рухомою точкою, за допомогою якого можна подати раціональні числа. Числа з фіксованою точкою – це аналог десяткових чисел; деякі біти подають цілу частину, а ті, що залишилися, – дробову. Числа з рухомою точкою є аналогом експоненційного подання числа з мантисою та порядком.

В англомовних країнах
як роздільник цілої та дробової частин чисел
використовується точка, а не кома.
Так склалося, що в сучасній технічній літературі
термін «точка» вживається частіше,
тому й ми застосовуватимемо його.
У деяких інших джерелах використовується термін «кома».

5.3.1 Числа з фіксованою точкою

Подання з фіксованою точкою має на увазі двійкову точку між бітами цілої та дробової частини, аналогічно десятковій точці між цілою та дробовою частинами звичайного десяткового числа.

Наприклад, на рис. 5.21, *a* показано число з фіксованою точкою з чотирма бітами цілої частини та чотирма дробової.

На рис. 5.21, *b* іншим кольором позначена двійкова точка, а на рис. 5.21, *c* зображено еквівалентне десяткове число.

Знакові числа з фіксованою точкою можна використовувати як у прямому, так і додатковому коді.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Рисунок 5.21 – Подання числа 6.75 з фіксованою точкою
з чотирма бітами цілої частини та чотирма дробової

На рис. 5.22 продемонстровано два подання числа –2.375 із фіксованою точкою з використанням чотирьох цілих бітів та чотирьох дробових бітів з неявною двійковою точкою.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Рисунок 5.22 – Подання числа –2.375 з фіксованою точкою:
a – абсолютне значення; b – прямий код; c – додатковий код

У прямому коді знаковий біт застосовується для вказівки на знак. Додатковий код двійкового числа отримуємо інверсією бітів абсолютного значення та додаванням 1 до молодшого розряду. У цьому прикладі молодший розряд відповідає 2^{-4}.

Приклад 5.4.

Арифметичні операції з числами з фіксованою точкою

Обчислимо вираз 0.75±0.625, використовуючи числа з фіксованою точкою.

Виконання. Спочатку перетворимо 0.625, абсолютне значення другого числа, у стандартне подання двійкового числа з фіксованою точкою: $0.625 \geq 2^{-1}$. Отже, ставимо 1 у розряд 2^{-1}, залишаючи 0.625 – 0.5 = 0.125.

Оскільки $0.125 < 2^{-2}$, тоді ставимо 0 у розряд 2^{-2}. Тому що $0.125 \geq 2^{-3}$, ставимо 1 до розряду 2^{-3}, залишаючи 0.125 – 0.125 = 0. Так, у розряді 2^{-4} буде 0. Отже, 0.62510 = 0000.1010_2.

На рис. 5.23 продемонстровано перетворення числа –0.625 у двійкове подання в додатковому коді. На рис. 5.24 показано складання чисел із фіксованою точкою і, для порівняння, десятковий еквівалент.

```
  0000.1010   Binary Magnitude
  1111.0101   One's Complement
+         1   Add 1
  1111.0110   Two's Complement
```

Рисунок 5.23 – Подання числа в додатковому коді

```
   0000.1100          0.75
 + 1111.0110     + (–0.625)
  10000.0010         0.125
 (a)              (b)
```

Рисунок 5.24 – Додавання:

а – двійкових чисел з фіксованою точкою; b – десятковий еквівалент

Зауважте, що перший одиничний біт у двійковому поданні числа з фіксованою точкою на рис. 5.24, *а* відкинуто у восьмибітному результаті.

5.3.2 Числа з рухомою точкою

Числа з рухомою точкою відповідають експонентному поданню.

У цьому поданні подолано обмеження наявності тільки фіксованої кількості цілих і дробових бітів, тому воно дає змогу подати дуже великі та дуже маленькі числа. Як і в експоненційному поданні, числа з рухомою точкою мають знак, мантису (*M*), основу (*B*) та порядок (*E*) (рис. 5.25).

$$\pm M \times B^{E}$$

Рисунок 5.25 – Числа з рухомою точкою

Наприклад, число $4.1*10^3$ є десятковим експоненційним поданням числа 4100. Мантіса є 4.1, основа дорівнює 10, а порядок – 3. Десяткова точка переміщується на позицію правіше за найзначніший (старший) розряд. У чисел із рухомою точкою основа дорівнює 2, а мантиса буде двійковим числом. 32 біти використовуються для подання 1 знакового біта, 8 бітів порядку та 23 біти мантиси [1–7].

Приклад 5.5.

32-бітне число з рухомою точкою

Знайдіть подання десяткового числа 228 у вигляді числа з рухомою точкою.

Виконання. Для початку перетворимо десяткове число на двійкове: $228_{10}= 11100100_2 = 1.11001_2 * 2^7$. На рис. 5.26 продемонстровано 32-бітне кодування, що далі для ефективності модифіковано. Знаковий біт позитивний, дорівнює 0, 8 бітів порядку дають значення 7, а 23 біти, що залишилися, – це мантиса.

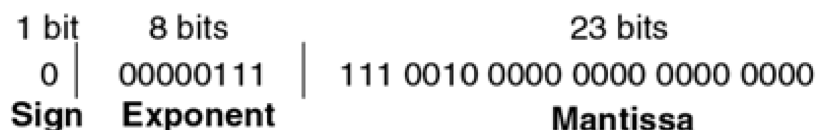


Рисунок 5.26 – 32-розрядне кодування числа з рухомою точкою: версія 1

У двійкових числах з ріхомою точкою перший біт мантиси (ліворуч від точки) завжди дорівнює 1, і тому його можна не зберігати. Це називається неявна старша одиниця. На рис. 5.27 зображено модифіковане подання: $228_{10}= 11100100_2 * 2^0 =1.11001_2 * 2^7$. Неявна старша одиниця не входить у 23 біти мантиси. Зберігаються лише дробові біти. Це звільняє додатковий біт для корисних даних.

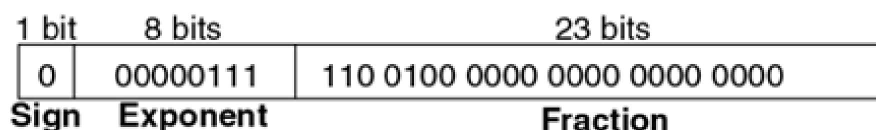


Рисунок 5.27 – Кодування числа з рухомою точкою: версія 2

Зробимо останню модифікацію подання порядку. Порядок має подавати як позитивний показник степеня, так і негативний. Для цього у форматі з рухомою точкою використовується зміщений порядок, який є початковим порядком, плюс постійне зміщення. 32-бітне подання з рухомою точкою використовує зміщення 127. Наприклад, для порядку 7 зміщений порядок виглядатиме так: 7 + 127 = 134 =10000110_2, для порядку – 4 зміщений порядок дорівнює – 4 + 127 = 123 = 01111011_2.

На рис. 5.28 продемонстровано подання числа 1.11001_2*2^7 у форматі з рухомою точкою з неявною старшою одиницею та зміщеним порядком 134 (7 + 127). Це подання відповідає стандарту *IEEE* 754.

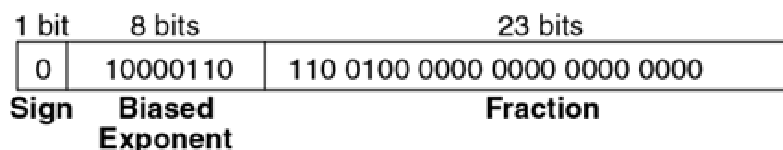


Рисунок 5.28 – Подання числа з рухомою точкою за стандартом *IEEE* 754

Особливі випадки: 0, ±∞ та NaN

Стандарт *IEEE* для чисел з рухомою точкою передбачає особливі випадки подання таких чисел, як 0, нескінченність та неприпустимі результати. Наприклад, подавати число 0 як числа з рухомою точкою неможливо через наявність неявної старшої одиниці. Для цих випадків зарезервовано спеціальні коди: порядок містить лише нулі чи одиниці. У табл. 5.2 наведено познаки 0, ±∞, *NaN*. Як і в знакових числах, рухома точка має і позитивний, і негативний 0. *NaN* використовується для чисел, що не існують, наприклад корінь з –1 і *log*2(-5).

Таблиця 5.2 – Познаки 0, ±∞ та *NaN* відповідно до стандарту *IEEE* 754

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
±∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Формати одинарної та подвійної точності

Вище ми розглядали 32-бітні числа з рухомою точкою. Такий формат називають форматом одинарної точності. Стандарт *IEEE* 754 також визначає 64-бітні числа подвійної точності, що дають змогу подавати значний діапазон чисел із великою точністю. У табл. 5.3 наведено число бітів, що використовуються в полях різних форматів.

Таблиця 5.3 – Числа з рухомою точкою з одинарною та подвійною точністю

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Якщо не брати до уваги спеціальні випадки, згадані раніше, звичайні числа одинарної точності охоплюють діапазон від $\pm1.175494*10^{-38}$ до $\pm3.402824*10^{38}$. Їх точність становить близько сім десяткових розрядів, оскільки 2–24 $\approx 10^{-7}$. Числа з подвійною точністю охоплюють діапазон від $\pm2.22507385850720*10^{-308}$ до $\pm1.79769313486232*10^{308}$ і мають точність близько 15 десяткових розрядів.

Округлення

Арифметичні результати, які виходять за межі доступної точності, необхідно округлювати до найближчих чисел. Існують такі способи округлення: у менший бік (1), у більший бік (2), до нуля (3) та до найближчого числа (4). За замовчуванням прийнято округлення до найближчого числа. У разі, якщо два числа перебувають на однаковій відстані, тоді обирається те, що матиме нуль у молодшому розряді дробової частини.

Нагадаємо, що число переповнюється, коли його величина занадто велика для будь-якого подання. Аналогічно число є зникним малим, коли воно занадто мале для подання.

За умови способу (4) переповнені числа округляються до ±∞, а зникні малі – до нуля.

Додавання чисел з рухомою точкою

Додавання чисел з рухомою точкою не така проста операція, як у разі подання чисел у додатковому коді. Для додавання двох таких чисел необхідно виконати конкретні кроки.

1. Виділити біти порядку та мантиси.
2. Приєднати неявну старшу одиницю до мантиси.
3. Порівняти порядки.
4. За потреби зрушити мантису числа, що має менший порядок.
5. Скласти мантиси.
6. За необхідності нормалізувати мантису та порядок.
7. Округлити результат.
8. Зібрати назад порядок і мантису в підсумкове число з рухомою точкою.

На рис. 5.29 продемонстровано процес складання чисел з рухомою точкою 7.875 ($1.11111*2^{2}$) та 0.1875 ($1.1*2^{-3}$). Результат дорівнює 8.0625 ($1.0000001*2^{3}$). Після вилучення мантиси та порядку, приєднання неявної старшої одиниці (кроки 1 і 2), порядки порівнюються способом віднімання меншого порядку з більшого. Результатом буде число бітів, на яке необхідно

зрушити мантису меншого число вправо (крок 4) для вирівнювання двійкової точки (тобто щоб зробити порядки рівними). Вирівняні значення складаються. Оскільки мантиса суми дорівнює або більша ніж 2.0, результат потрібно нормалізувати, зрушивши його вправо на 1 біт і збільшити порядок на 1. У цьому прикладі результат точний і жодних округлень не потрібно. Він зберігається у форматі з рухомою точкою після видалення неявної старшої одиниці мантиси й додавання знакового біта.

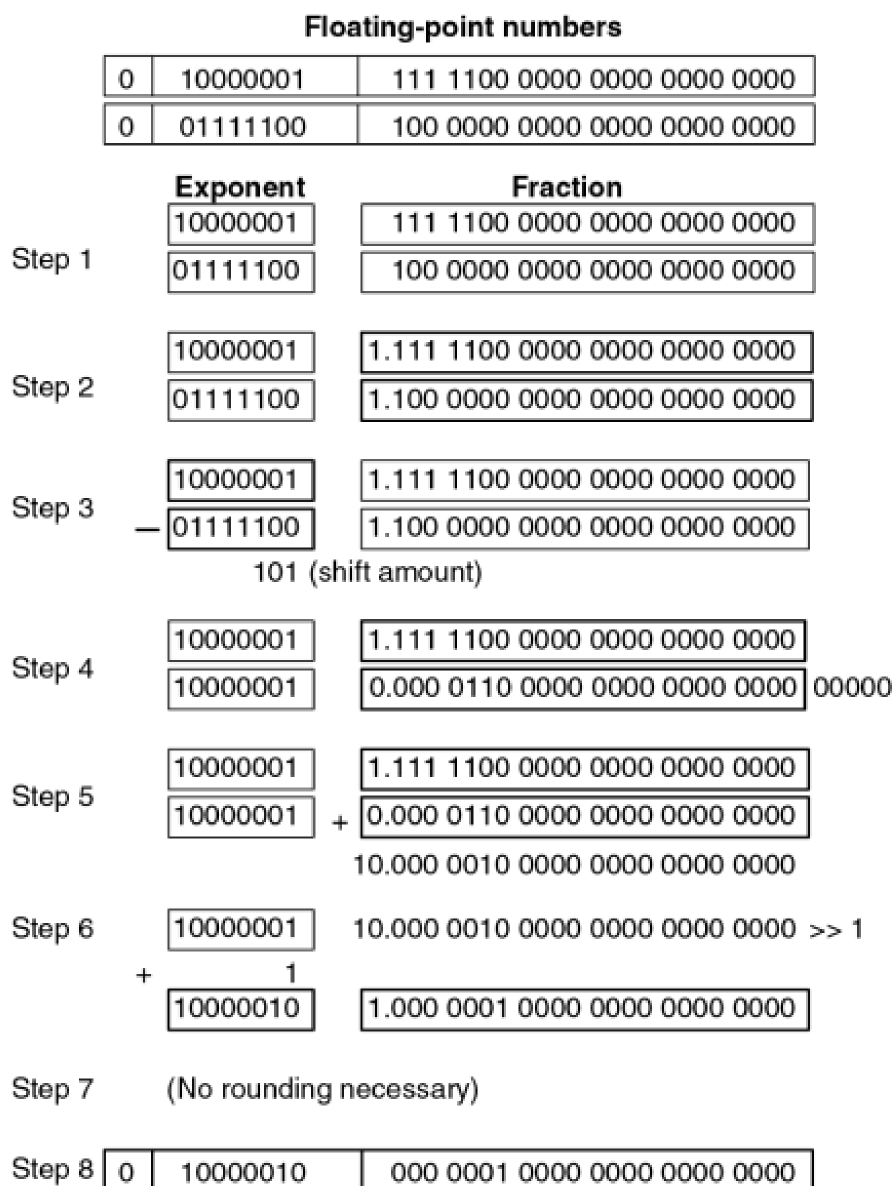


Рисунок 5.29 – Додавання чисел із рухомою точкою

5.4 Функціональні вузли послідовної логіки

У цьому розділі розглянемо функціональні вузли послідовної логіки – лічильники та зсувні регістри.

5.4.1 Лічильники

N-розрядний двійковий лічильник (див. рис. 5.30) є послідовною арифметичною схемою, що має входи тактового сигналу, скидання і *N*-розрядний вихід *Q*. Сигнал скидання ініціалізує виходи нульовим значенням. Вихід лічильник послідовно приймає всі 2^Nможливі значення *N*-розрядного двійкового числа, перехід до наступного значення відбувається за переднім фронтом тактового імпульсу.

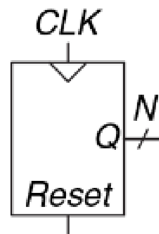


Рисунок 5.30 – Умовна познака лічильника

На рис. 5.31 зображено *N*-розрядний лічильник, що містить суматор і регістр, що має вхід скидання. На кожному циклі лічильник додає 1 до величини, що зберігається в регістрі. У прикладі 5.5 мовами *HDL* описано двійковий лічильник з асинхронним скиданням.

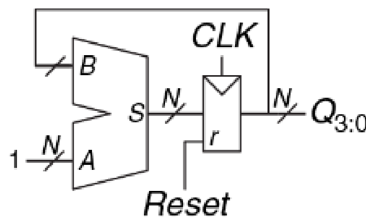


Рисунок 5.31 – *N*-бітний лічильник

Лічильники інших типів, наприклад реверсивні, розглянуті у вправах 5.43–5.46.

***HDL*-приклад 5.5.**
Лічильник

SystemVerilog

```
module counter #(parameter N = 8)
        (input logic clk,
        input logic reset,
        output logic [N-1:0] q);
    always_ff @(posedge clk, posedge reset)
        if (reset)q<=0;
```

```
        else q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity counter is
    generic(N: integer := 8);
    port(clk, reset: in STD_LOGIC;
         q: out STD_LOGIC_VECTOR(N-1 downto 0));
end;
architecture synth of counter is
begin
     process(clk, reset) begin
       if reset then q <= (OTHERS => '0');
       elsif rising_edge(clk) then q <= q + '1';
       end if;
    end process;
end;
```

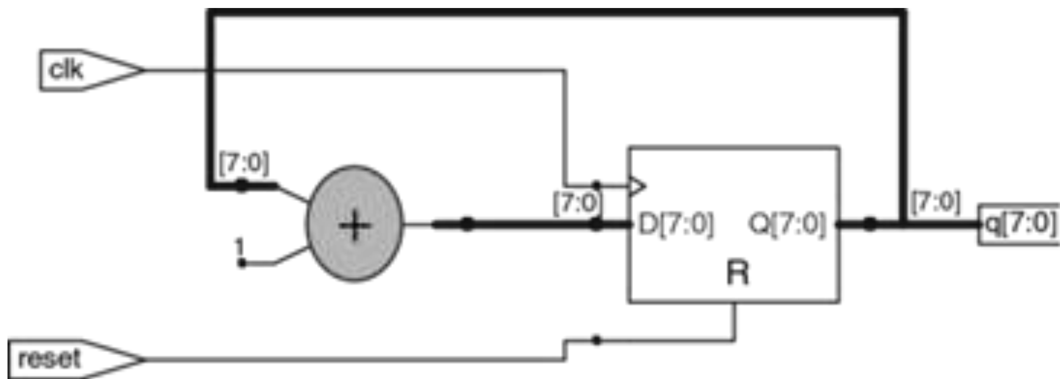


Рисунок 5.32 – Синтезований лічильник

5.4.2 Регістри зсуву

На рис. 5.33 подано регістр зсуву, що має вхід тактового сигналу, послідовний вхід S_{in}, послідовний вихід S_{out} та N паралельних виходів $Q_{N-1:0}$. По кожному передньому фронту тактового імпульсу в перший тригер регістра записується новий біт зі входу S_{in}, а вміст наступних тригерів зсувається вперед. Останній біт регістра можна рахувати з виходу S_{out}.

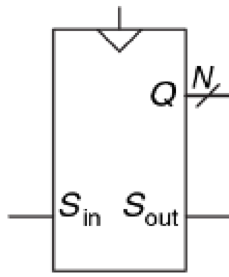


Рисунок 5.33 – Умовна позначка регістра зсуву

Регістр зсуву можна розглядати як послідовно-паралельний перетворювач. На вхід *Sin* надходять послідовні дані (по одному біту за раз). Після *N* циклів останні *N* значень вхідного сигналу можна паралельно рахувати з виходу *Q*.

Як видно на рис. 5.34, регістр зсуву може бути побудований з *N* послідовно з'єднаних тригерів. Деякі регістри мають сигнал скидання для ініціалізації всіх тригерів.

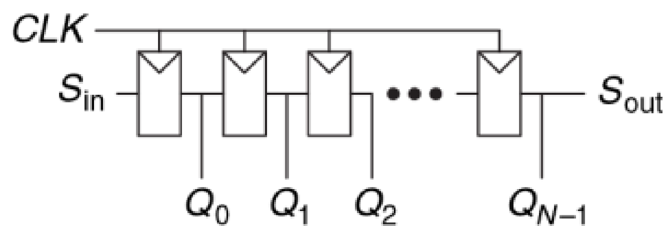


Рисунок 5.34 – Схема регістра зсуву

У паралельно-послідовний перетворювач паралельно завантажується *N* бітів, що потім послідовно (по одному біту за раз) надходять на вихід. Схемотехніка паралельно-послідовного перетворювача й регістра зсуву подібні. Регістр зсуву можна модифікувати для виконання як послідовно-паралельного, так і паралельно-послідовного перетворення, якщо до нього додати паралельний вхід $D_{N-1:0}$ і сигнал керування *Load*, як показано на рис. 5.35. Коли вхід *Load* активовано, у всі тригери паралельно завантажуються дані з входу *D*. В іншому разі регістр зсуву виконує звичайний зсув. У *HDL*-прикладі 5.6 регістр описано мовами *HDL*.

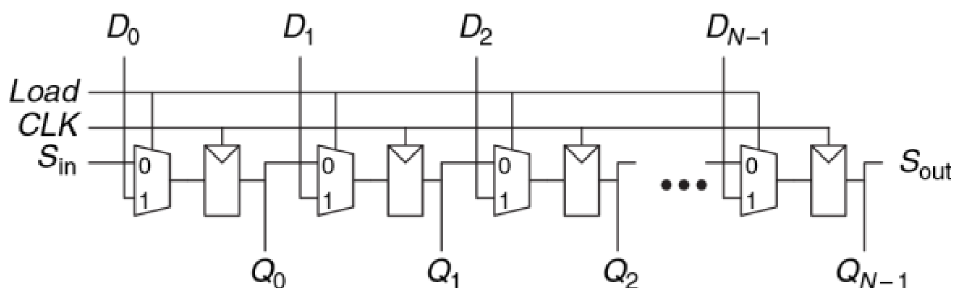


Рисунок 5.35 – Регістр зсуву з паралельним завантаженням

***HDL*-приклад 5.6.**
Регістр зсуву з паралельним завантаженням

System Verilog

```
module shiftreg #(parameter N = 8)
             (input logic clk,
              input logic reset, load,
              input logic sin,
              input logic [N-1:0] d,
              output logic [N-1:0] q,
               output logic sout);
always_ff @ (posedge clk, posedge reset)
       if (reset) q <= 0;
       else if (load) q <= d;
        else q <= {q[N–2:0], sin};
  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity shiftreg is
     generic(N: integer := 8);
     port(clk, reset: in STD_LOGIC;
     load, sin: in STD_LOGIC;
     d: in STD_LOGIC_VECTOR(N-1 downto 0);
     q: out STD_LOGIC_VECTOR(N-1 downto 0);
     sout: out STD_LOGIC);
end;
architectura synth of shiftreg is
begin
   process(clk, reset) begin
     if reset = ‘1’ then q <= (OTHERS => ‘0’);
     elsif rising_edge(clk) then
        if load then q <= d;
        else q <= q(N-2 downto 0) & sin;
        end if;
     end if;
   end process;
```

```
  sout <= q(N-1);
end;
```

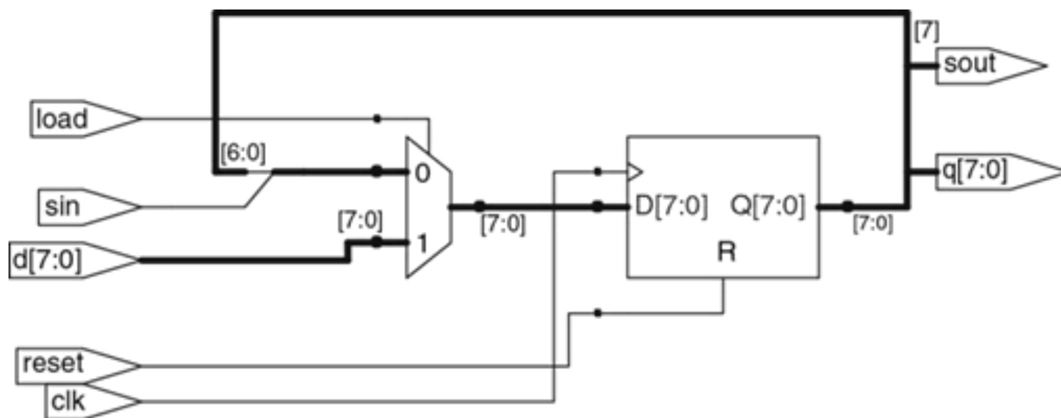


Рисунок 5.36 – Регістр зсуву з паралельним завантаженням

Сканувальні ланцюжки*

Часто для тестування послідовних схем упроваджуються сканувальні ланцюжки (*scan chains*), у яких застосовуються регістри зсуву. Тестування комбінаційних схем відносно просте. На вхід схеми подають спеціально підібрані вхідні сигнали, що називаються векторами тесту, а значення вихідних сигналів порівнюють з очікуваними результатами. Тестування послідовних схем набагато складніше, оскільки їх стан залежить від передісторії вхідних сигналів. Якщо початковий стан схеми зафіксовано, то для досягнення стану, що цікавить, може знадобитися значна кількість тестових векторів. Наприклад, для перевірки коректності роботи старшого розряду 32-бітного лічильника необхідно скинути лічильник, а потім подати на нього 2^{31} (близько двох мільярдів) тактових імпульсів!

Для розв'язання цієї проблеми бажано мати змогу безпосередньо спостерігати та змінювати всі стани схеми. Це досягається запровадженням спеціального тестового режиму, у якому вміст усіх тригерів може бути взято до уваги чи змінено належним чином. Більшість реальних систем містить надзвичайно багато тригерів, тому неможливо виділити спеціальні контакти для читання та зміни їх вмісту. Замість цього всі тригери системи з'єднані між собою в один величезний зсувний регістр, що називається сканувальним ланцюжком. За нормальної роботи тригери отримують дані зі своїх інформаційних входів *D*, а сканування вимкнено. У тестовому режимі відбувається послідовний зсув вмісту всіх тригерів, що містить сканувальний ланцюжок, їх старий вміст надходить на вихід *Sout*, а новий завантажується крізь вхід S_{in}. До складу сканованого тригера (*scannable flip-flop*), крім власне тригера, належить мультиплексор завантаження. На рис. 5.37 наведено схему

й графічну познаку тригера, що сканується, і показано, як вони з'єднуються послідовно для створення *N*-бітного сканованого регістра.

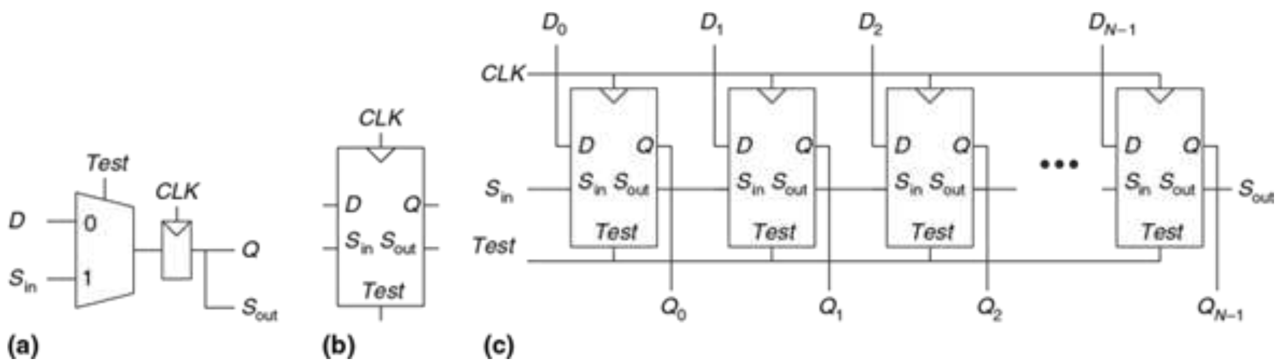


Рисунок 5.37 – Тригер, що сканується:
а – схема; b – умовна познака; c – *N*-бітовий сканований регістр

Наприклад, роботу старшого розряду 32-бітного лічильника можна протестувати таким чином: у тестовому режимі він переводиться в стан 011111...111, потім виконується один цикл рахунку в нормальному режимі, після цього в тестовому режимі зчитується стан лічильника, що має бути 100000...000. Ця послідовність дій потребує лише 32 + 1 + 32 = 65 циклів.

5.5 Матриці пам'яті

У попередніх розділах ми ознайомилися з арифметичними та послідовними схемами, що застосовуються для оброблення інформації. Для зберігання цієї інформації та результатів роботи схем у цифрових системах необхідні запам'ятовувальні пристрої (*memories*). Регістр, що містить декілька тригерів, є запам'ятовувальним пристроєм, призначеним для зберігання незначних обсягів інформації. У цьому підрозділі розглянемо матриці пам'яті, що дають ефективно зберігати чималі обсяги інформації [1–7].

Спочатку ознайомимося із загальними властивостями всіх типів матриць пам'яті. Потім розглянемо три типи матриць пам'яті: динамічний оперативний пристрій (ДОП) (*DRAM*, динамічна пам'ять з довільним доступом); статичний оперативний запам'ятовувальний пристрій (*SRAM*, статична пам'ять із довільним доступом); постійний пристрій пам'яті (ППП) (*ROM*, пам'ять тільки для читання). Ці типи матриць розрізняються способом зберігання інформації. Далі стисло проаналізуємо апаратні витрати для створення матриці пам'яті та їх швидкодію. Наприкінці підрозділу розглянемо використання матриць пам'яті виконання функцій комбінаційної логіки та їх опис за допомогою мов *HDL*.

5.5.1 Огляд

На рис. 5.38 подано графічну познаку узагальненої матриці пам’яті. Пам’ять організована як двовимірна матриця запам’ятовувальних елементів. Вміст пам’яті записується та зчитується по рядках. Рядок обирається адресою (*Address*). Записані значення називаються даними (*Data*). Матриця з *N*-бітною адресою та *M*-бітними даними має 2^N рядків та *M* стовпців. Кожен рядок даних називається словом. Отже, матриця містить 2^N *M*-бітних слів.

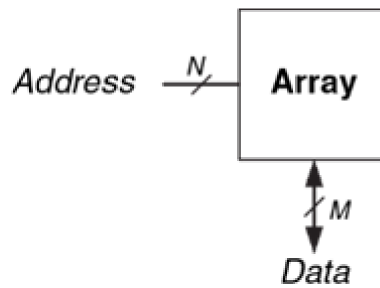
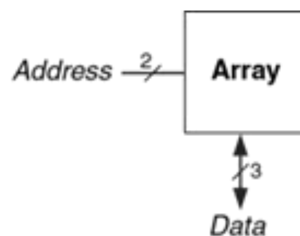


Рисунок 5.38 – Умовна познака узагальненої матриці пам’яті

На рис. 5.39 зображено матрицю пам’яті, адреса якої містить 2 біти, а дані – 3 біти. Два адресні біти обирають один із чотирьох рядків (слів даних) матриці. Ширина кожного слова даних дорівнює 3 бітам. На рис. 5.39, *b* наведено приклад можливого вмісту матриці пам’яті. Глибина матриці дорівнює кількості її рядків, а ширина – кількості стовпців, що також називається розміром слова. Розмір матриці дорівнює добутку кількості стовпців на кількість рядків. На рис. 5.39 продемонстровано матрицю «4 слова × 3 біти», або просто 4 × 3. Познаку матриці «1024 слова × 32 біти» подано на рис. 5.40. Загальний розмір цієї матриці дорівнює 32 кбайтам.



(a)

Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

(b)

Рисунок 5.39 – Матриця пам’яті 4×3: a – умовна познака; b – функція

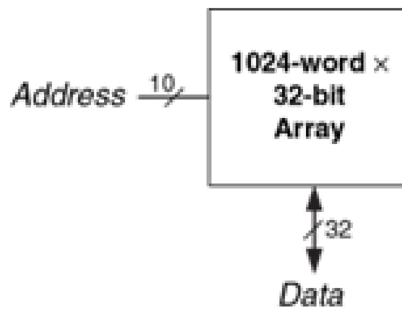


Рисунок 5.40 – Матриця 32 кбайти:
глибина становить 2^{10} = 1024 слова; ширина – 32 біти

Елементи пам'яті

Матриці пам'яті – це набір елементів пам'яті, кожен з яких зберігає один біт даних. На рис. 5.41 продемонстровано, що кожен запам'ятовувальний елемент з'єднаний з лінією слів (лінією вибірки слів) і лінією бітів (лінією запису-зчитування). За будь-якої комбінації адресних бітів активується лише одна лінія вибірки слів і цим уможливлюється доступ до елементів відповідного рядка. Коли лінія вибірки слів деякого рядка активна, його елементи можуть видавати дані лінії запису-зчитування чи приймати дані з цих ліній. Інакше запам'ятовувальні елементи від'єднані від лінії запису-зчитування. Для різних типів пам'яті схеми запам'ятовувальних елементів будуть різними.

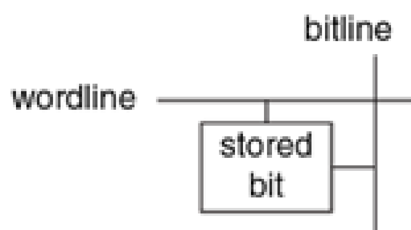


Рисунок 5.41 – Елемент пам'яті

Під час читання бітів лінія запису-зчитування спочатку перебуває у вимкнутому стані (*Z*). Потім вмикається лінія вибірки слів, і елементи, що запам'ятовують, видають збережене значення на лінію запису-зчитування. У процесі запису інформації в запам'ятовувальний елемент сигнал надходить на лінію запису-зчитування зі спеціального підсилювача запису-зчитування, що має незначний вихідний опір. Потім вмикається лінія вибірки слів, і лінії запису-зчитування з'єднуються із запам'ятовувальними елементами.

Організація

На рис. 5.42 зображена внутрішня організація матриці пам'яті 4×3. Звичайно, реальні пристрої мають набагато більший обсяг, але поведінка малих

матриць пам'яті може бути екстрапольовано на поведінку великих. У цьому прикладі матриця зберігає дані (див. рис. 5.39, *b*).

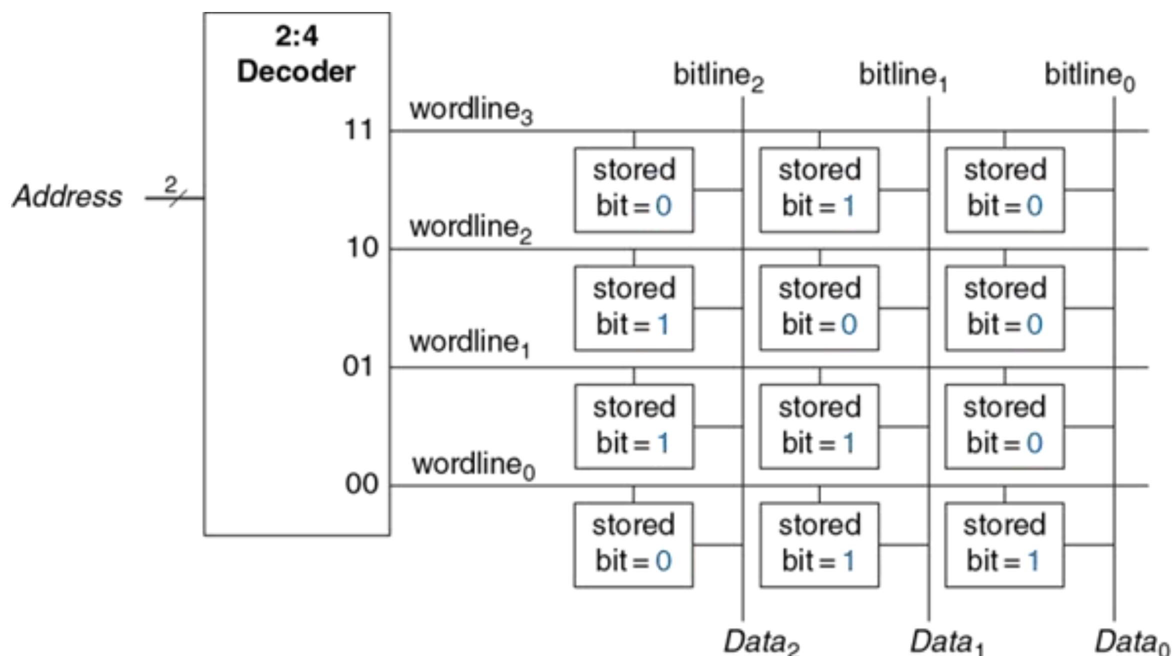


Рисунок 5.42 – Матриця пам'яті 4×3

У процесі читання вмісту пам'яті активується лінія вибірки слів, та із запам'ятовувальних елементів відповідного рядка на лінії запису-зчитування надходить напруга високого або низького логічного рівня. Під час запису на лінії запису-зчитування за допомогою підсилювача запису-зчитування подаються дані, що будуть збережені в елементах рядка, а потім активується відповідна лінія вибірки слів. Наприклад, для читання даних за адресою 10 лінії запису-зчитування залишаються у вимкненому стані, декодер активує другу лінію вибірки слів (*wordline*2), і дані, що зберігаються в цьому рядку (100), зчитуються з ліній запису-зчитування (*Data*). Для запису значення 001 з а адресою 11 на лінії запису-зчитування з підсилювача запису-зчитування надходить величина 001, потім активується третя лінія вибірки слів (*wordline*3), і нове значення зберігається в запам'ятовувальних елементах.

Порти пам'яті

Пам'ять усіх типів має один чи кілька портів (*ports*). Крізь порти здійснюється доступ до вмісту пам'яті за деякою адресою для читання, запису або читання-запису. У попередньому прикладі було розглянуто однопортову пам'ять.

Багатопортова пам'ять забезпечує одночасний доступ до вмісту за кількома адресами. На рис. 5.43 зображена трипортова пам'ять із двома портами

для читання та одним для запису. Порт 1 зчитує дані, що зберігаються за адресою *A*1, та надсилає їх на вихід *RD*1. Порт 2 подає інформацію, що зберігається за адресою *A*2, на вихід *RD*2. Порт 3 дозволяє записати дані, подані на вхід *WD*3, елемент за адресою *A*3, запис інформації здійснюється на передньому фронті тактового імпульсу за умови активного сигналу *WE*3.

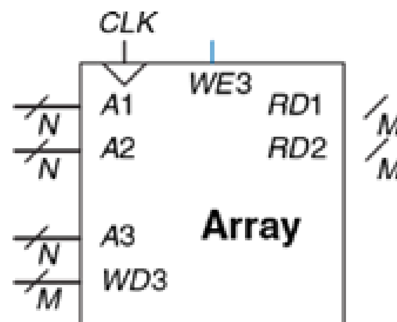


Рисунок 5.43 – Трипортова пам’ять

Типи пам’яті

Матрицям пам’яті властивий розмір (глибина × ширина), кількість та тип портів. Пам’ять усіх типів зберігає інформацію в матриці запам’ятовувальних елементів, але спосіб зберігання бітів різний.

Запам’ятовувальні елементи розрізняються за способом зберігання бітів. Запам’ятовувальні пристрої поділяють на два великі класи: оперативні (ОЗП) (*RAM*, пам’ять з довільним доступом) і постійні запам’ятовувальні пристрої (ПЗП) (*ROM*, пам’ять тільки для читання). Оперативний запам’ятовувальний пристрій є енергозалежним, тобто в разі вимкнення живлення інформація, що зберігалася в ОЗП, втрачається. Натомість ПЗП енергонезалежний і зберігає свої дані навіть за відсутності живлення.

Поділ запам’ятовувальних пристроїв на два великі класи – ОЗП і ПЗП – виник на початку комп’ютерної ери й уже застарів і не відповідає реальній ситуації. В ОЗП час доступу до всієї інформації є однаковим. Навпаки, у пам’яті з послідовним доступом, зокрема в пам’яті на магнітній стрічці, доступ до «ближньої» інформації відбувається набагато швидше, ніж доступ до «далекої» (наприклад, тої інформації, що зберігається на протилежному кінці магнітної стрічки). Історично ПЗП називається постійним, оскільки дані з такого пристрою можна було зчитувати, але не можна було записувати до нього. Проте в сучасних ПЗП є можливість записувати інформацію! Основна різниця, на яку необхідно звернути увагу, полягає в тому, що ОЗП енергозалежний, а ПЗП енергонезалежний.

Основні класи ОЗП – це динамічний (динамічна пам’ять, *DRAM*) і статичний (статична пам’ять, SRAM) оперативні пристрої пам’яті. Динамічна

пам'ять зберігає інформацію як заряд конденсаторів, а статична – як стан бістабільної схеми, що містить два перехреснoз'єднані інвертори. Існує багато різновидів ПЗП, що розрізняються методами запису та зчитування інформації. Різні типи запам'ятовувальних пристроїв будуть розглянуті нижче.

5.5.2 Динамічне ОЗП (DRAM)

У динамічному ОЗП (*DRAM*) бітовим значенням відповідає наявність і відсутність заряду конденсатора. На рис. 5.44 зображено запам'ятовувальний елемент динамічного ОЗП. Значення біта зберігається в конденсаторі. *N*-канальний МОН-транзистор (*nMOS*) є ключем, що може під'єднати конденсатор до лінії запису-зчитування або вимкнути його. Коли лінія вибірки слів активна, транзистор вмикається, і біти, що зберігаються, передаються на лінію запису-зчитування, або, навпаки, відбувається запис нової інформації в елемент.

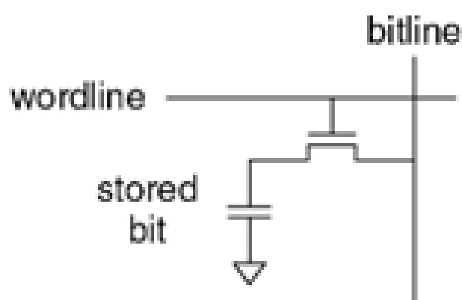


Рисунок 5.44 – Елемент динамічного запам'ятовувального ОЗП

Як показано на рис. 5.45, *a*, коли конденсатор заряджено до *VDD*, біт, що зберігається, дорівнює 1; коли він розряджений до нуля (рис. 5.45, *b*), біт, що зберігається, дорівнює 0. Вузол конденсатора буде динамічним, оскільки він фактично не керується транзистором, приєднаним до *VDD* або *GND*.

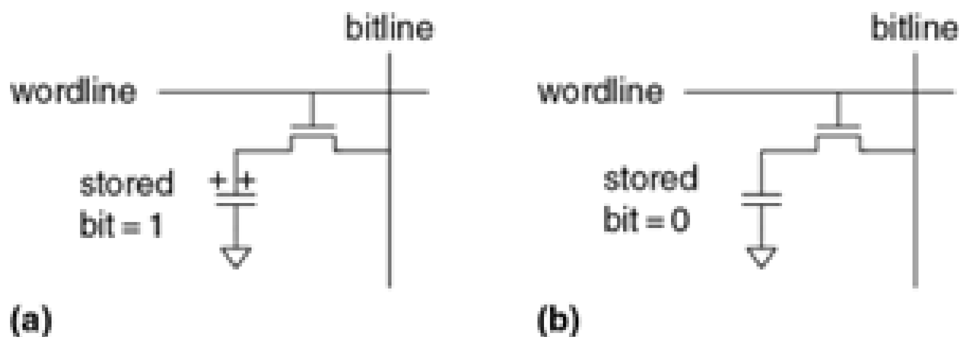


Рисунок 5.45 – Зберігання даних у динамічному ОЗП

Під час читання інформація передається від конденсатора на лінію запису-зчитування. У процесі запису дані надходять із лінії запису-зчитування на конденсатор. Читання знищує інформацію, що зберігалася в конденсаторі,

тому після кожного читання вона має бути відновлена (перезаписана). Навіть якщо з динамічного ОЗП не потрібно зчитувати дані, через саморозряд конденсаторів вони мають регенеруватися (зчитуватися й перезаписуватися) кожні кілька мілісекунд.

5.5.3 Статичний ОЗП (SRAM)

Статичним ОЗП (*SRAM*) називається тому, що в ньому відсутня необхідність регенерації даних, що зберігаються. На рис. 5.46 подано запам'ятовувальний елемент статичного ОЗП. Інформація зберігається в бістабільній схемі, що містить два перехресно з'єднаних інверторів і подібна до тих, які були розглянуті в підрозділі 3.2.

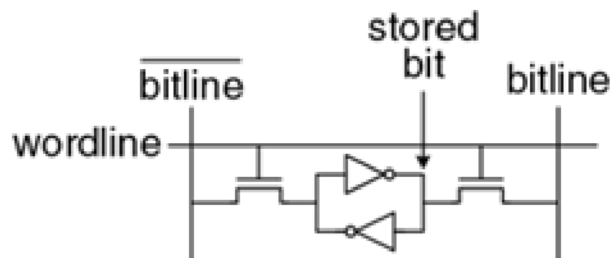


Рис. 5.46 – Запам'ятовуючий елемент статичного ОЗП 1

Кожний елемент має два виходи – *bitline* та *bitline* з інверсією. Коли лінія вибірки слів активна, обидва *n*-канальних МОН-транзистора відкриваються та інформація може бути записана в елемент або зчитана з нього. На відміну від динамічного ОЗП, перехресно з'єднані інвертори повертають запам'ятовувальний елемент у рівноважний стан, якщо вийде внаслідок випадкових відхилень.

5.5.4 Площа та затримки

Тригери, статичні та динамічні ОЗП є енергозалежними запам'ятовувальними пристроями, але вони розрізняються часовими властивостями та площею кристала, необхідною для зберігання одного біта. У табл. 5.4 подано порівняння цих трьох типів енергозалежної пам'яті. Дані, що зберігаються в тригері, безпосередньо доступні з його виходу. Але схема тригера містить принаймні 20 транзисторів. Загалом, що більше транзисторів використовується в приладі, то більшу площу він займає, споживає більше енергії та коштує дорожче. Затримка в динамічному ОЗП триваліша, ніж у статичному ОЗП, оскільки в ньому лінія запису-зчитування фактично не управляється транзистором. Затримка динамічного ОЗП обмежується щодо повільної передачі заряду з конденсатора на лінію

зчитування-запису. Через необхідність виконання періодичної регенерації та регенерації після читання динамічний ОЗП має меншу пропускну здатність, ніж статичний. Сучасні різновиди динамічного ОЗП, такі як синхронний динамічний ОЗП (*SDRAM*) і синхронний динамічний ОЗП з подвоєною швидкістю обміну (*DDR SDRAM*, або стисло *DDR*) були розроблені для подолання окресленої проблеми. У синхронному динамічному ОЗП використовується тактовий сигнал конвеєризації доступу до пам’яті. У синхронному динамічному ОЗП з подвоєною швидкістю обміну інформація передається як за переднім, так і заднім фронтом тактового імпульсу, що подвоює пропускну здатність за умови заданої частоти тактового сигналу. Синхронний динамічний ОЗП з подвоєною швидкістю обміну було вперше стандартизовано 2000 року, він працював на частотах від 100 до 200 МГц. У нових стандартах *DDR*2, *DDR*3 і *DDR*4 тактова частота була збільшена, і до 2012 року вона перевищила 1 ГГц.

Таблиця 5.4 – Порівняння типів пам’яті

Тип пам’яті	**Кількість транзисторів у елементі**	**Затримка**
тригер	~20	незначна
статичне ОЗП	6	середня
динамічне ОЗП	1	тривала

Затримка пам’яті та її пропускна здатність також залежить від розміру пам’яті; за інших рівних умов пам’ять більшого обсягу, як правило, працює повільніше, ніж меншого. Вибір найкращого типу пам’яті для конкретного проєкту залежить від вимог до швидкодії, ціни та енергоспоживання.

5.5.5 Регістрові файли

Цифрові системи часто використовують кілька регістрів для зберігання тимчасових змінних. Такі групи регістрів, що називаються регістровими файлами, зазвичай реалізуються у вигляді невеликих багатопортових матриць статичного ОЗП, оскільки вони більш компактні, ніж матриці тригерів.

На рис. 5.47 подано трипортовий регістровий файл, що містить 32 регістри по 32 біти кожен і побудований на основі трипортової пам’яті, подібної до зображеної на рис. 5.44. Регістровий файл має два порти для читання (*A*1/*RD*1 та *A*2/*RD*2) та один порт для запису (*A*3/*WD*3). П’ятирозрядні адреси *A*1, *A*2 і *A*3 забезпечують доступ до будь-якого з 2^5 = 32 регістрів. Отже, одночасно можна записувати інформацію в два регістри та зчитувати з одного.

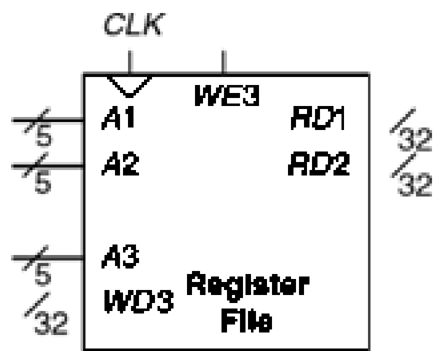


Рисунок 5.47 – Регістровий файл 32 × 32
з двома портами читання та одним портом запису

5.5.6 Постійний запам'ятовувальний пристрій

У постійному запам'ятовувальному пристрої (ПЗП, *ROM*) бітовим значенням відповідає наявність або відсутність транзистора. На рис. 5.48 зображено простий запам'ятовувальний елемент ПЗП. Під час читання інформації з елемента на лінію запису-зчитування від зовнішнього джерела подається рівень логічної 1. Потім активується лінія вибірки слів. Якщо в елементі є транзистор, він відкривається і встановлює на лінії запису-зчитування рівень логічного 0. Коли транзистор відсутній, на лінії запису-зчитування залишається рівень логічної 1. Зверніть увагу на те, що ПЗП є комбінаційною схемою і немає значення, який може бути втрачено внаслідок вимкнення живлення.

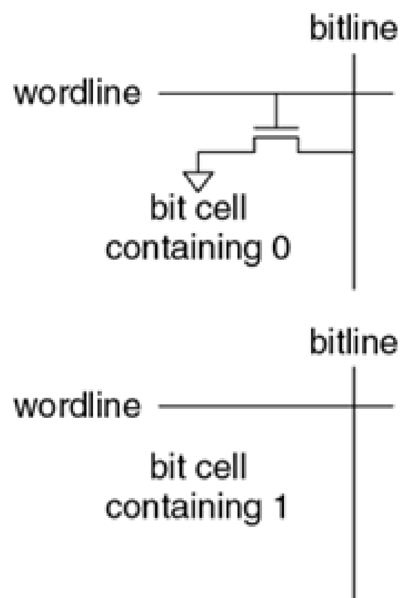


Рисунок 5.48 – Елементи пам'яті ПЗП, що містять 0 і 1

Вміст ПЗП може бути показаний за допомогою точкової нотації. На рис. 5.49 подано точкову нотацію для ПЗП «4 слова × 3 біти», яка містить інформацію з рис. 5.39. Наявність точки на перетині рядка (лінії вибірки слів)

і стовпця (лінії запису-зчитування) показує, що біт, який зберігається, дорівнює 1. Наприклад, на верхній лінії вибірки слів є тільки одна точка на її перетині з $Data_1$, отже, за адресою 11 зберігається значення 010.

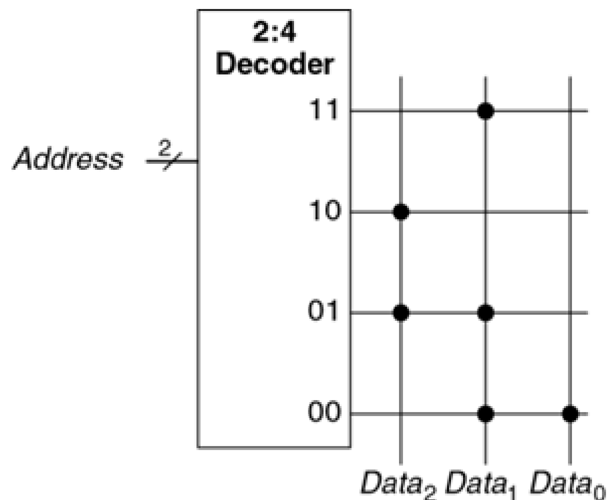


Рисунок 5.49 – ПЗП 4×3: точкова нотація

Концептуально ПЗП може бути побудовано з використанням дворівневої логіки, що містить групу логічних елементів І, за якою розташована група елементів АБО. Елементи І породжують усі можливі мінтерми і, отже, формують декодер. На рис. 5.50 зображено ПЗП (див. рис. 5.49), побудований із застосуванням декодера та елементів АБО. Кожна точка на рис. 5.49 відповідає з'єднанню рядка та входу елемента АБО (див. рис. 5.50).

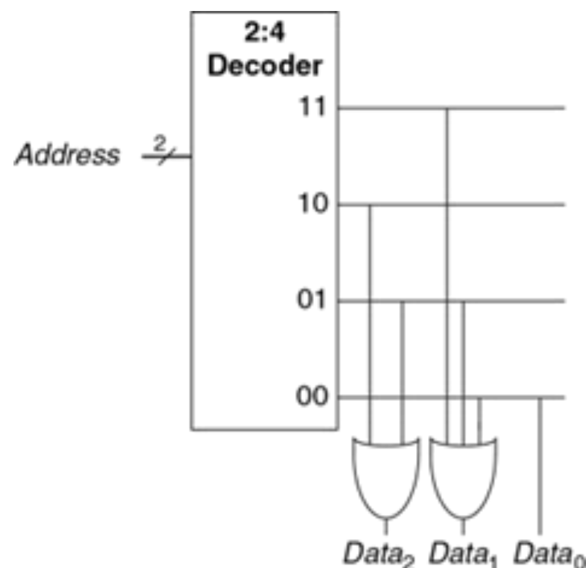


Рисунок 5.50 – Реалізація ПЗП 4×3 з використанням логічних елементів

Для вихідних бітів даних з однією точкою, таких як $Data_0$, елемент АБО не потрібен. Таке подання ПЗП свідчить, що з допомогою цього пристрою можна реалізувати довільну дворівневу логічну функцію. Реальні ПЗП

містять транзистори, а не логічні елементи, що дає змогу зменшити їх розмір та вартість. У п. 5.6.3 реалізація ПЗП на рівні транзисторів буде розглянута детально.

Вміст запам'ятовувальних елементів ПЗП (рис. 5.48) у процесі його виготовлення визначається наявністю або відсутністю транзистора в кожному осередку. У програмованому ПЗП (*PROM*, ППЗП) транзистори розміщені у всіх елементах, але в них є можливість керувати з'єднанням цих транзисторів із землею.

На рис. 5.51 зображено запам'ятовувальний елемент ПЗП, що програмується плавкими перемичками (*fuse-programmable ROM*). Користувач може програмувати ПЗП, подаючи високу напругу деяким перемичкам і цим перепалюючи їх. Якщо перемичка є, тоді транзистор з'єднаний із землею, і елемент зберігає 0. Якщо перемичка зруйнована, тоді транзистор від'єднаний від землі та елемент зберігає 1. Таке ПЗП також називають одноразово програмованим, оскільки після перепалювання перемички її неможливо відновити.

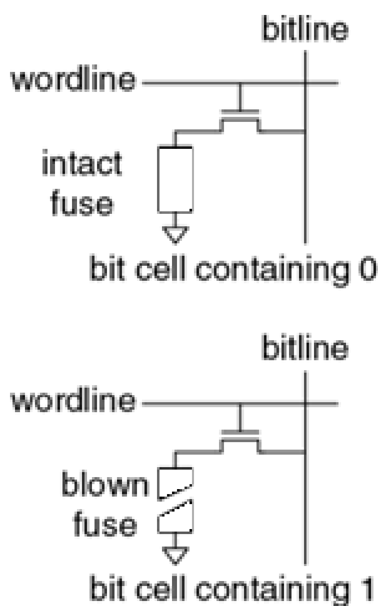


Рисунок 5.51 – Запам'ятовувальний елемент ПЗП, програмованого перемичками

У перепрограмованих ПЗП реалізований механізм оборотного з'єднання-роз'єднання транзисторів із землею. У програмованих ПЗП (СППЗП, *Erasable PROMs, EPROM*) *n*-МОП-транзистори й перемички замінені транзисторами з рухомим затвором (*floating-gate transistor*). Такий затвор не з'єднаний фізично ні з якими іншими провідниками. Коли на транзистор подається досить висока напруга, електрони тунелюють крізь ізолятор на рухомий затвор, транзистор вмикається і з'єднує лінію вибірки слів і лінію бітів (вихід декодера).

Коли СППЗП піддають ультрафіолетовим випромінюванням протягом приблизно пів години, електрони викидаються з рухомого затвора й транзистор вимикається. Ці дії називаються програмуванням і стиранням відповідно. У програмі, що електрично стирається, ПЗП (ЕСППЗП, *electrically erasable PROM, EEPROM*) і у флеш-пам'яті (*flash memory*) використовується аналогічний принцип, проте ультрафіолетове випромінювання не використовується, оскільки на кристалі присутня спеціальна схема стирання. В ЕСППЗП запам'ятовувальні елементи можна стирати індивідуально, у флеш-пам'яті стирання відбувається великими блоками, вона дешевша, оскільки в ній використовується менша кількість схем, що стирають. У 2012 році вартість флеш-пам'яті становила приблизно $1 за 1 Гб, і вона продовжувала падати приблизно на 30–40% за рік. Флеш-пам'ять стала дуже популярною для збереження значних обсягів інформації у переносних пристроях з батарейним живленням, зокрема камерах і музичних програвачах.

Отже, сучасні ПЗП є постійними в строгому значенні слова: вони можуть програмуватися, тобто інформація в них може записуватися. Різниця між ОЗП та ПЗП полягає в тому, що запис у ПЗП потребує більше часу і вони є енергонезалежними [1–7].

5.5.7 Реалізація логічних функцій із використанням матриць пам'яті

Хоча основним застосуванням матриць пам'яті є зберігання інформації, вони можуть використовуватися для реалізації комбінаційних логічних функцій. Наприклад, вихід $Data_2$ ПЗП (див. рис. 5.49) є функцію *XOR* двох входів *Address*.

Аналогічно $Data_0$ є функцією *NAND* двох входів. Матриця пам'яті розмірністю 2^N слів × *M* бітів може реалізувати довільну логічну функцію з *N* входами та *M* виходами. Наприклад, ПЗП на рис. 5.49 реалізує три функції двох аргументів.

Матриці пам'яті, що здійснюють логічні функції, називаються таблицями перетворень (*lookup tables, LUT*). На рис. 5.52 зображено матрицю пам'яті «4 слова × 1 біт», яка використовується як таблиця перетворення для реалізації функції $Y = AB$. У використанні пам'яті виконання логічної функції для заданої комбінації входів (адреси) у ній відбувається пошук відповідного значення виходу. Кожна адреса відповідає рядку в таблиці істинності, а кожен біт, що зберігається, – значенню вихідного сигналу.

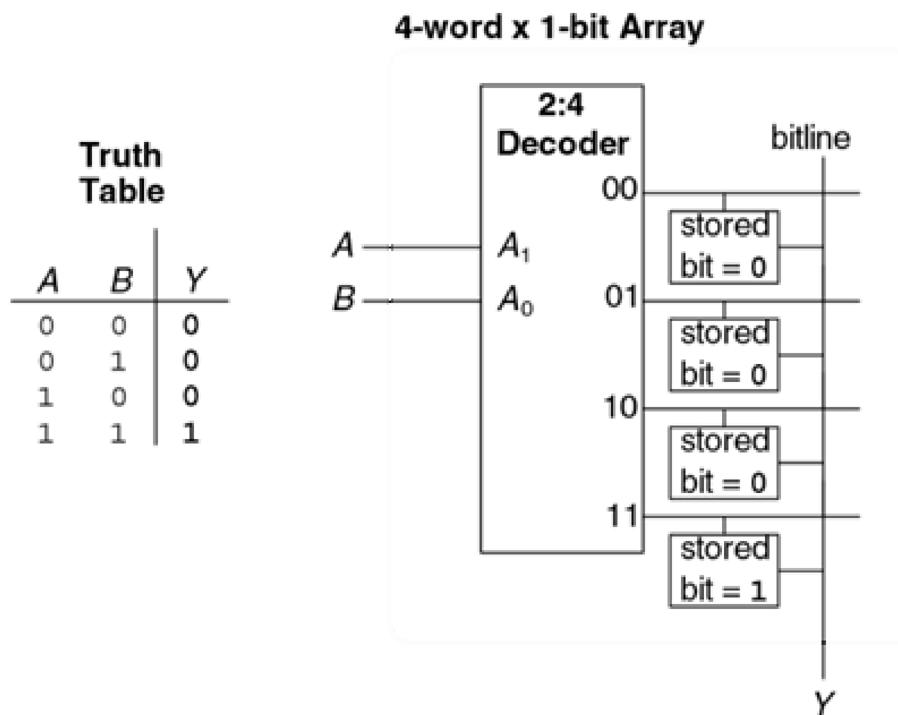


Рисунок 5.52 – Матриця «4 слова × 1 біт»
з використанням таблиці перетворення

5.5.8 Мови опису апаратури й пам'яті

В *HDL*-прикладі 5.7 мовами *HDL* описано ОЗП розмірністю 2^N слів × *M* бітів. Ця ОЗП має синхронний вхід дозволу запису. Іншими словами, запис у пам'ять відбувається по передньому фронту тактового імпульсу, якщо сигнал дозволу запису (*write enable*) не перебуває в активному стані. Читання відбувається негайно. Безпосередньо після увімкнення живлення вміст ОЗП не передбачуваний.

Приклад 5.7
HDL ОЗП

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
          (input logic clk,
           input logic we,
           input logic [N-1:0] adr,
           input logic [M-1:0] din,
           output logic [M-1:0] dout);
logic [M–1:0] mem [2**N–1:0];
always_ff @(posedge clk)
   if (we) mem [adr] <= din;
```

```
assign dout = mem [adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity ram_array is
      generic(N: integer := 6; M: integer := 32);
      port(clk,
              we: in STD_LOGIC;
               adr: in STD_LOGIC_VECTOR(N-1 downto 0);
               din: in STD_LOGIC_VECTOR(M-1 downto 0);
               dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;
architecture synth of ram_array is
       type mem_array is array ((2**N–1) downto 0)
               of STD_LOGIC_VECTOR (M–1 downto 0);
   signal mem: mem_array;
begin
     process(clk) begin
         if rising_edge(clk) then
              if we then mem(TO_INTEGER(adr)) <= din;
              end if;
        end if;
     end process;
    dout <= mem(TO_INTEGER(adr));
 end;
```

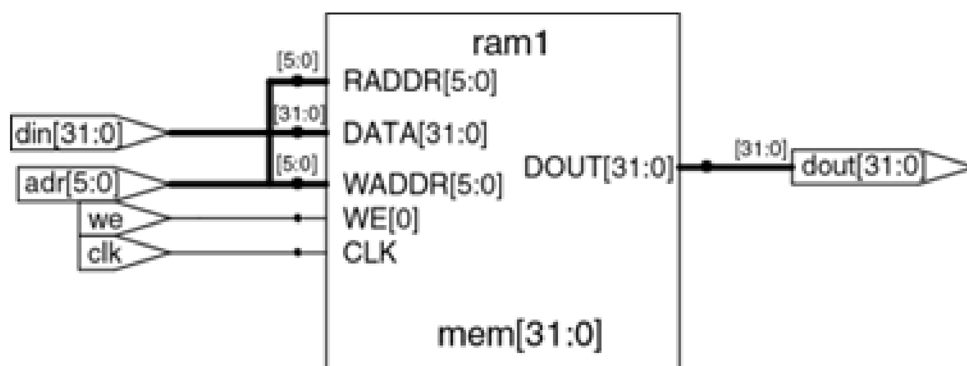


Рисунок 5.53 – Синтезована схема

В *HDL*-прикладі 5.8 мовами *HDL* описано ПЗП розмірністю 4 слова × 3 біти. Вміст ПЗП визначається оператором *case*. ПЗП є наскільки малий, що може бути синтезований у вигляді набору логічних елементів, а не матриці. Зауважте, що декодер 7-сегментного коду з прикладу 4.24 мовою *HDL* був синтезований як ПЗП на рис. 4.20.

Приклад 5.8.
HDL ПЗП

SystemVerilog

```
    module rom(input logic [1:0] adr,
                  output logic [2:0] dout):
always_comb
        case(adr)
            2’b00: dout <= 3’b011;
            2’b01: dout <= 3’b110;
            2’b10: dout <= 3’b100;
            2’b11: dout <= 3’b010;
        endcase
  endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity rom is
     port(adr: in STD_LOGIC_VECTOR(1 downto 0);
           dout: out STD_LOGIC_VECTOR(2 downto 0));
end;
architecture synth of rom is
 begin
      process(all) begin
         case adr is
            when “00” => dout <= “011”;
            when “01” => dout <= “110”;
            when “10” => dout <= “100”;
            when “11” => dout <= “010”;
        end case;
     end process;
 end;
```

5.6 Матриці логічних елементів

Логічні елементи, як і запам’ятовувальні елементи, можуть бути організовані в регулярні матриці. Якщо з’єднання між логічними елементами програмуються, такі матриці можна налаштувати для реалізації довільної логічної функції, до того ж у цьому разі не потрібно буде змінювати з’єднання між мікросхемами на платі. Регулярна структура полегшує проєктування. Матриці логічних елементів виробляються у значній кількості, що забезпечує їх малу вартість. Існує програмне забезпечення, що дає змогу перенести проєкти цифрових пристроїв на такі матриці. Більшість матриць логічних елементів реконфігурується, що уможливлює зміну проєкту без заміни апаратного забезпечення. Реконфігурованість дуже цінна в розробленні та корисна в процесі експлуатації виробу, оскільки він може бути оновлений способом простого завантаження нової конфігурації.

У цьому підрозділі розглянемо два типи матриць логічних елементів: програмовану логічну матрицю (*programmable logic arrays*, ПЛМ, *PLA*) і програмовану матрицю логічних елементів (*field programmable gate arrays*, ППМЛЕ, *FPGA*). У програмованій логічній матриці, що є більш давньою технологією, можна реалізувати лише комбінаційні логічні функції. Програмована матриця логічних елементів дає змогу створювати як комбінаційні, так і послідовні схеми [1–7].

5.6.1 Програмовані логічні матриці

Програмовані логічні матриці (ПЛМ, *PLA*) дають змогу реалізувати дворівневі комбінаційні логічні схеми, задані досконалою нормальною диз’юнктивною формою (ДДНФ). На рис. 5.54 продемонстровано, що ПЛМ містить матрицю І, за якою розташована матриця АБО. Входи (у прямій та інверсній формі) надходять на матрицю І, що створює імпліканти, які зі свого боку об’єднуються функціями АБО і формують вихідний сигнал матриці. ПЛМ розмірності $M \times N \times P$ бітів має M входів, N імплікант і P виходів.

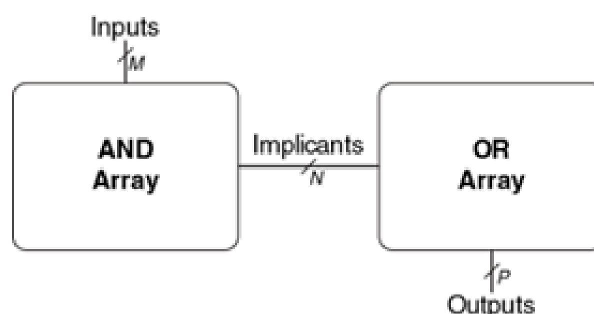


Рисунок 5.54 – Програмована логічна матриця (*PLA*) «$M \times N \times P$ бітів»

На рис. 5.55 подана точкова нотація ПЛМ 3×3×2 біти, яка реалізовує функції $X = \overline{A}\ \overline{B}\ \overline{C} + AB\overline{C}$ і $Y = A\overline{B}$. Кожен рядок у матриці І формує імпліканту. Точки в рядках матриці І показують, які літерали формують імпліканти.

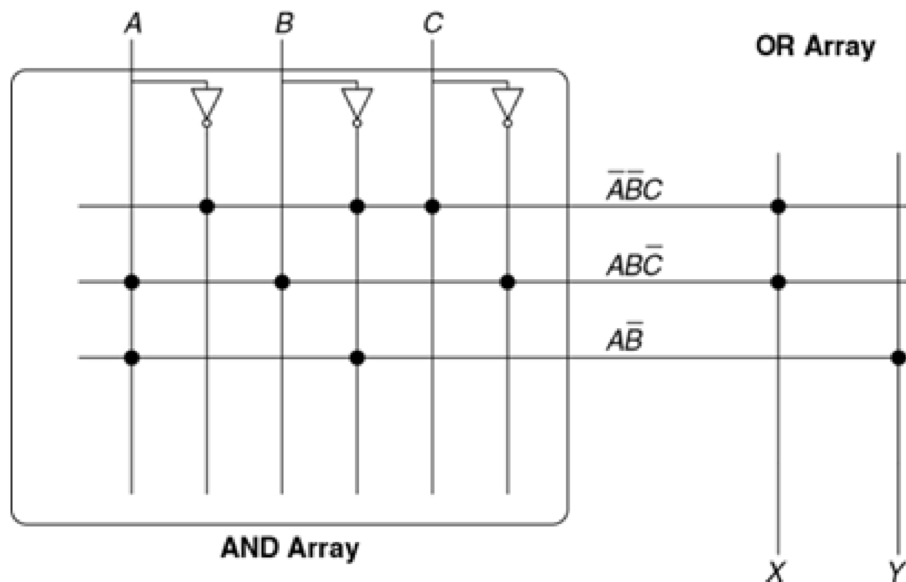


Рисунок 5.55 – Програмована логічна матриця (*PLA*) «3×3×2 біти»: точкова нотація

Матриця І на рис. 5.55 формує три імпліканти: $\overline{A}\ \overline{B}\ \overline{C} + AB\overline{C}$ і $A\overline{B}$. Точки в матриці АБО показують, які імпліканти належать до вихідної функції.

На рис. 5.56 проілюстровано, як ПЛМ можна побудувати з використанням дворівневої логіки. Альтернативну реалізацію опишемо в п. 5.6.3.

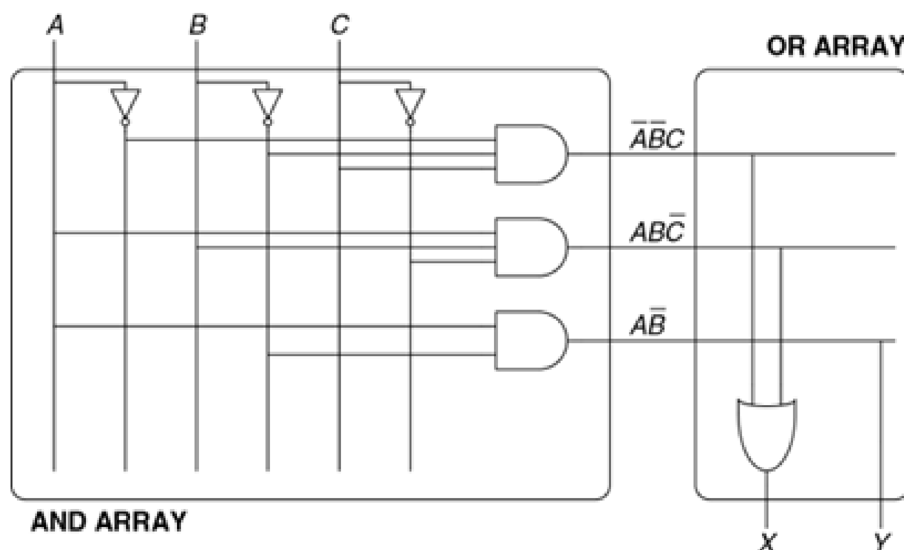


Рисунок 5.56 – Реалізація програмованої логічної матриці (*PLA*) «3×3×2 біти» з використанням дворівневої логіки

ПЗП можна розглядати як різновид ПЛМ. ПЗП з організацією 2^M слів × N бітів є ПЛМ розмірності $M \times 2^M \times N$ бітів. Декодер виконує функції матриці І та створює всі 2^M мінтерми. Масив запам'ятовувальних елементів виконує функції матриці АБО та визначає вихідні сигнали. Якщо функція залежить не від усіх 2^M мінтермів, то, можливо, реалізація з ПЛМ буде більш компактною, ніж з ПЗП. Наприклад, для виконання функцій ПЛМ розмірності 3×3×2 біти (див. рис. 5.55 та 5.56), потрібно ПЗП 8 слів × 2 біт.

У простих програмованих логічних пристроях (ППЛП, *SPLD*) базові матриці І та АБО ПЛМ доповнені регістрами та додатковими схемами. Однак на цей час ППЛУ і ПЛМ здебільшого витіснені програмованими матрицями логічних елементів (ППМЛЕ, *FPGA*), які більш гнучкі та ефективні у створенні великих систем.

5.6.2 Програмовані користувачем матриці логічних елементів

Програмовані користувачем матриці логічних елементів (ППМЛЕ, *FPGA*) є матрицями елементів, що реконфігуруються. Із застосуванням спеціального програмного забезпечення користувач може описати свій проєкт мовою опису апаратури або у вигляді схеми, а потім реалізувати його в *FPGA*. У низці випадків матриці *FPGA* потужніші та гнучкіші, ніж ПЛМ.

У *FPGA* можна реалізувати як комбінаційні, так і послідовні схеми. Вони можуть втілити багаторівневі логічні схеми, тоді як і ПЛМ здатні реалізувати лише дворівневі схеми. У сучасні *FPGA* інтегровані інші корисні вузли, такі як помножувачі, високошвидкісні пристрої введення / виведення, ЦАП, АЦП, великі ОЗП та процесори.

FPGA є матрицею конфігурованих логічних елементів (*logic elements*, ЛЕ, *LE*), які також називаються логічними блоками, що конфігуруються (*configurable logic blocks*, КЛБ, *CLB*). Кожен ЛЕ можна налаштувати для виконання функцій деякої комбінаційної або послідовної схеми. На рис. 5.57 подано узагальнену структуру FPGA. ЛЕ оточені елементами введення / виведення (*input / output elements, IOE, EBP*), призначеними для організації обміну інформацією між *FPGA* та іншими компонентами системи. Елементи введення / виведення з'єднують входи та виходи логічних елементів з контактами корпусу мікросхеми. Логічні елементи можуть бути з'єднані між собою та з елементами введення / виведення за допомогою програмованих каналів трасування.

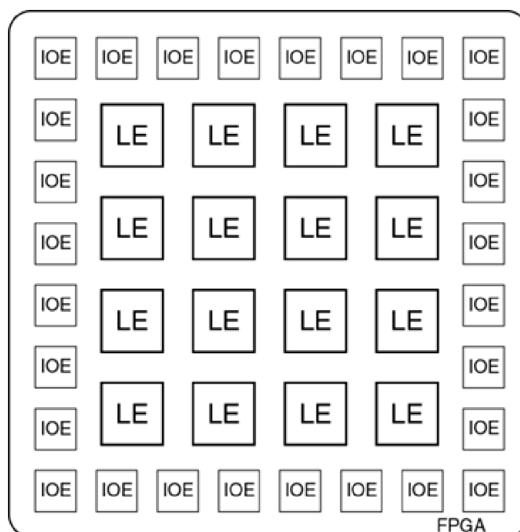


Рисунок 5.57 – Узагальнена структура *FPGA*

Лідерами ринку *FPGA* є фірми *Altera Corp*. та *Xilinx, Inc*. На рис. 5.58 зображено один логічний елемент *FPGA* фірми *Altera Cyclone IV*, виробництво якого розпочалося 2009 року. Основними компонентами логічного елемента є чотиривходова таблиця перетворення (*LUT*) та однобітовий регістр. Логічний елемент містить конфігуровані мультиплексори, призначені для комутації сигналів у логічному елементі. Під час програмування *FPGA* встановлюється вміст таблиць перетворення (*LUT*) та визначаються вхідні сигнали мультиплексорів, що проходять на їх виходи.

Логічний елемент *FPGA Cyclone IV* містить одну чотиривходову таблицю перетворення (*LUT*) та один тригер. Способом завантаження відповідних значень *LUT* вона може бути налаштована для реалізації довільної логічної функції чотирьох (або менше) аргументів. Також у процесі конфігурації *FPGA* сигналів вибору, що визначають, як мультиплексори комутуватимуть канали передачі даних у межах логічного елемента (*LE*) і між ним та сусідніми логічними елементами (*LE*) або елементами введення / виведення (*IOE*), надаються необхідні значення. Наприклад, залежно від конфігурації мультиплексора на один із входів *LUT* деякого *LE* може надходити сигнал або з входу *data* 3 або з виходу регістра цього самого *LE*. На інші три входи *LUT* сигнали завжди надходять із входів *LE data* 1, *data* 2 і *data* 4. Залежно від трасування зовнішніх з’єднань сигнал на входи *data* 1–4 надходить з *IOE* або виходів інших *LE*. Вихід *LUT* може надходити безпосередньо на вихід *LE* за умови реалізації комбінаційної логічної схеми, або крізь тригер під час створення послідовної схеми. Сигнал на вхід тригера може надходити з виходу *LUT* цього *LE*, входу *data* 3 або з виходу регістра попереднього *LE*. Крім того, в *LE* входить низка допоміжних схем: додаткові мультиплексори для трасування,

схеми керування сигналами дозволу та скидання тригера, схеми, що дають змогу реалізувати суматор з послідовним перенесенням. У *FPGA* фірми *Altera* групи із 16 *LE* об'єднані в блок логічних матриць (*logic array block, LAB*), для передачі даних між *LE* одного блоку існують спеціальні локальні з'єднання.

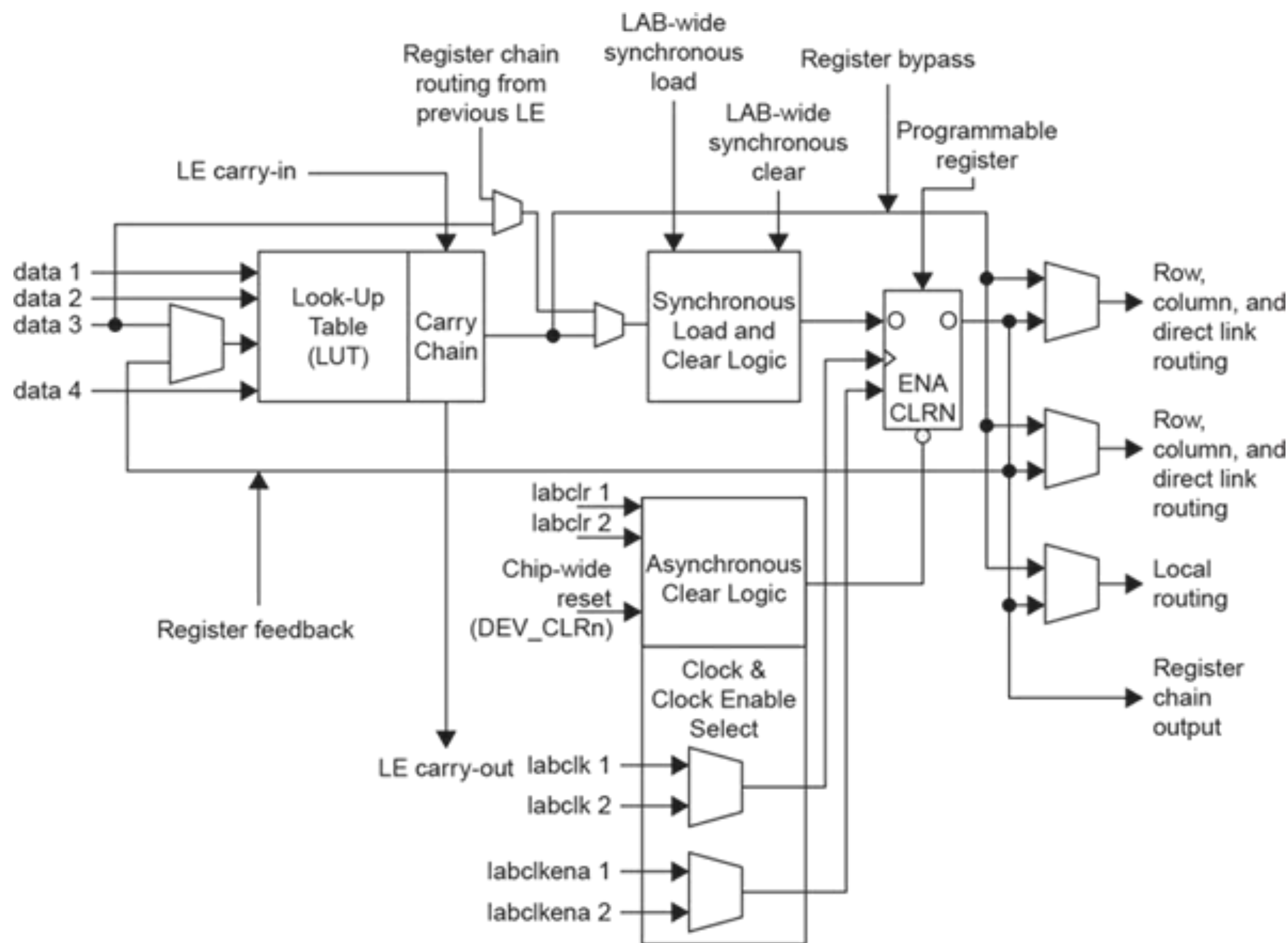


Рисунок 5.58 – Логічний елемент фірми *Cyclone IV*
(Відтворено з дозволу *Altera Cyclone™ IV Handbook* © 2010 *Altera Corporation*)

Отже, в *LE FPGA Cyclone IV* можна реалізувати одну функцію чотирьох (або менше) входів, до того ж вона може бути комбінаційною або послідовною, тобто мати на виході тригер. *FPGA* інших виробників організовані трохи інакше, але принцип побудови залишається загальним. Наприклад, у *FPGA* фірми *Xilinx* сьомої серії замість чотиривходової *LUT* використовується шестивходові.

Під час розроблення конфігурації *FPGA* проєктувальник спочатку створює схемний опис проєкту чи опис *HDL*. Потім відбувається синтез проєкту. Пакет синтезу схем визначає, як необхідно конфігурувати *LUT*, мультиплексори та канали трасування для реалізації заданих функцій. Ця конфігураційна інформація завантажується у *FPGA*. Оскільки *FPGA Cyclone IV* зберігають конфігураційну інформацію в статичному ОЗП, вони можуть

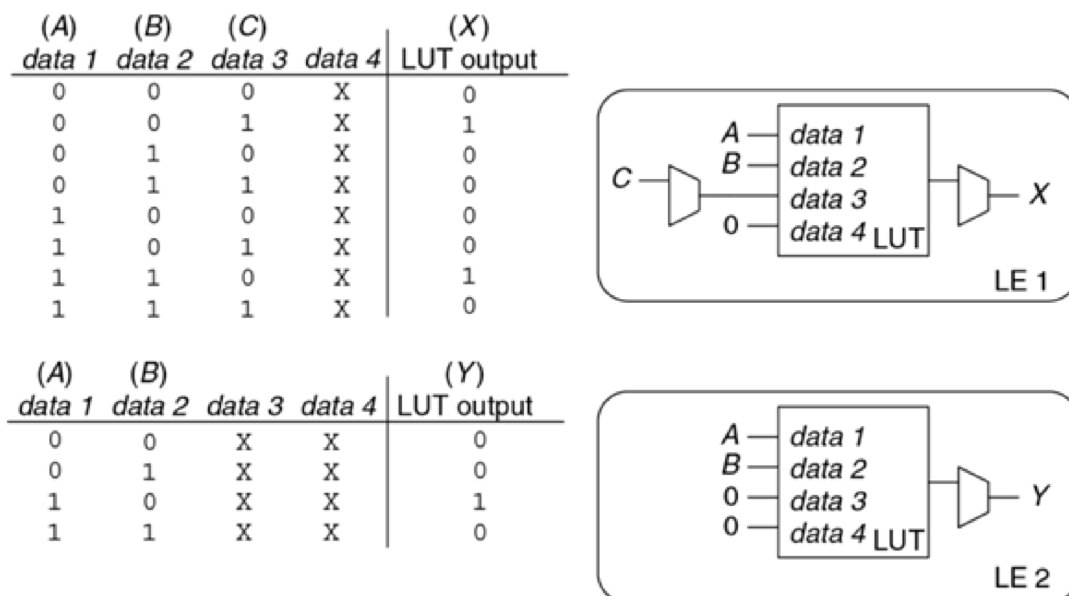
бути легко перепрограмовані. Вміст статичного ОЗП *FPGA* може бути завантажений з комп’ютера (у лабораторних умовах) або в процесі увімкнення живлення із спеціальної мікросхеми ЕСППЗП (*EEPROM*). Деякі виробники вбудовують ЕСППЗП безпосередньо в мікросхему *FPGA* або застосовують для конфігурування *FPGA* одноразові програмовані перемички.

Приклад 5.6.

Побудова функцій з використанням логічних елементів

Поясніть, як потрібно конфігурувати *LE FPGA Cyclone IV* для реалізації таких функцій: (*a*) $X = \overline{A}\ \overline{B}\ \overline{C} + AB\overline{C}$ і $Y = A\overline{B}$ (*b*) Y = JKLMPQR; (*c*) лічильник з основою 3 із двійковим кодуванням стану (див. рис. 3.29, *a*). За потреби ви можете показати зв’язок між логічними елементами.

Виконання. (*a*) Для реалізації функцій необхідно налаштувати два логічні елементи. Як показано на рис. 5.59, перша таблиця перетворення (*LUT*) обчислює *X*, друга – *Y*. На входи *data* 1, *data* 2 і *data* 3 першої таблиці перетворення подаються сигнали *A, B* і *C* (ці з’єднання встановлюються трасувальними каналами), вхід *data* 4 не використовується, але на нього потрібно подати якесь значення, наприклад 0. У другій таблиці перетворення на входи *data* 1 і *data* 2 подаються сигнали *A* та *B*; решта входів не застосовується, і на них подано 0. Вихідний мультиплексор налаштований для подання на вихід комбінаційного сигналу з таблиць перетворення, таким чином на виході формуються необхідні сигнали *X* і *Y*. Загалом, один логічний елемент дає змогу обчислити довільну функцію чотирьох (або менше) аргументів.



(*A*) data 1	(*B*) data 2	(*C*) data 3	data 4	(*X*) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(*A*) data 1	(*B*) data 2	data 3	data 4	(*Y*) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

Рисунок 5.59 – Конфігурація логічного елемента (*LE*)
для реалізації двох функцій, що мають до чотирьох входів

(*b*) Таблиця перетворення (*LUT*) першого логічного елемента має бути налаштована для обчислення *X* = *JKLM*, а другого – *Y* = *XPQR*.

Вихідні мультиплексори мають обирати комбінаційні виходи *X* та *Y* кожного логічного елемента (*LE*). Ця конфігурація зображена на рис. 5.60. Трасувальні канали між логічними елементами (*LE*), позначені пунктирними лініями, з’єднують вихід першого логічного елемента з входом другого. У цьому разі група логічних елементів дає змогу обчислити аналогічним чином функцію *N*-вхідних змінних.

(*J*) *data 1*	(*K*) *data 2*	(*L*) *data 3*	(*M*) *data 4*	(*X*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*P*) *data 1*	(*Q*) *data 2*	(*R*) *data 3*	(*X*) *data 4*	(*Y*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

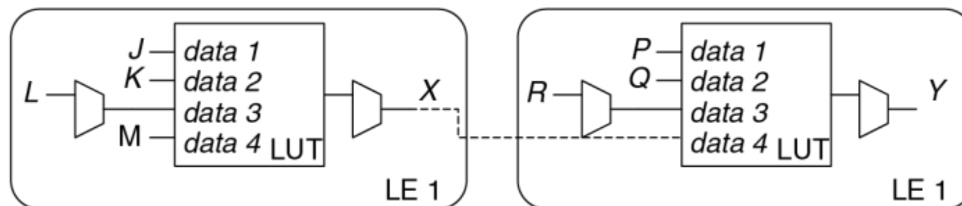
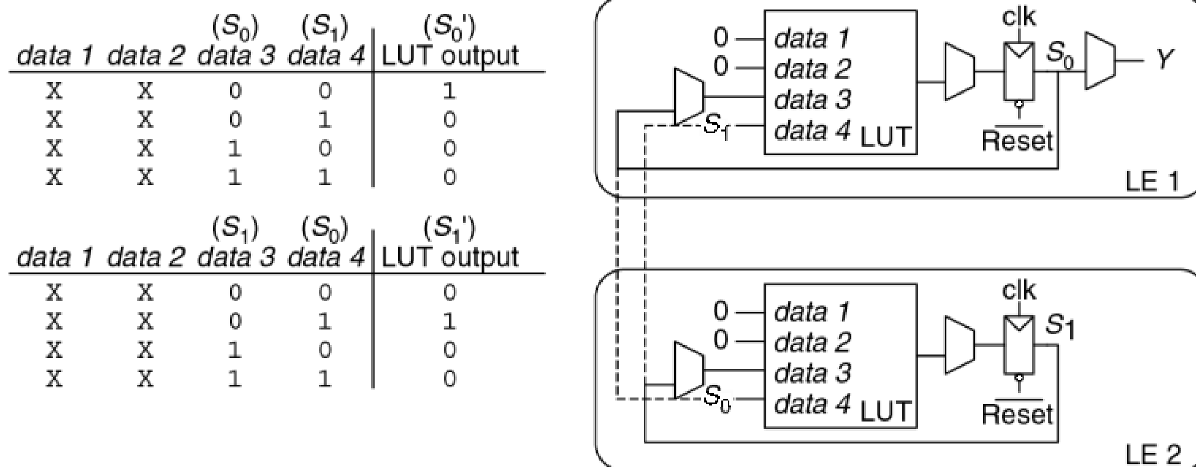


Рисунок 5.60 – Конфігурація логічних елементів (*LE*) для реалізації кінцевого автомата з двобітним станом

(*c*) Кінцевий автомат має два біти для зберігання стану (*S*1:0) та один вихід (*Y*). Наступний стан залежить від двох бітів поточного стану. Як продемонстровано на рис. 5.61, для визначення наступного стану за поточним використовуються два логічні елементи (*LE*). Два тригери, по одному з кожного логічного елемента (*LE*), зберігають цей стан. Тригери мають вхід скидання, який може бути з’єднаний із зовнішнім сигналом *Reset*. Пунктирними лініями позначений тракт подання сигналу крізь трасувальні канали та мультиплексори на входах *data* 3 з вихідних регістрів назад на входи таблиць перетворення (*LUT*). Для обчислення виходу *Y* може знадобитися додатковий логічний елемент. Проте в разі *Y* = *S*0, тобто *Y* надходить із виходу

першого логічного елемента. У такий спосіб увесь кінцевий автомат реалізовано на двох логічних елементах. Загалом, для реалізації кінцевого автомата необхідно принаймні по одному логічному елементу для кожного біта стану; якщо логіка визначення виходу чи наступного стану надто складна для однієї таблиці перетворення (*LUT*), то можуть знадобитися додаткові логічні елементи (*LE*) – один або кілька.



data 1	data 2	(S_0) data 3	(S_1) data 4	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	data 2	(S_1) data 3	(S_0) data 4	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

Рисунок 5.61 – Конфігурація логічних елементів (*LE*)
для реалізації кінцевого автомата з двобітним станом

Приклад 5.7.

Затримка в логічному елементі

Аліса розробляє кінцевий автомат, що має працювати на частоті 200 МГц. Вона використовує *FPGA Cyclone IV GX* з такими властивостями: *tLE* = 381 пс на *LE*, t_{SETUP} = 76 пс та t_{PCQ} = 199 пс для всіх тригерів. Затримка в поєднанні між *LE* дорівнює 246 пс. Час утримання тригерів установимо рівним 0. Яку максимальну кількість *LE* можна використовувати в Алісиному проєкті?

Виконання. Для визначення максимальної затримки поширення в комбінаційній логічній схемі Аліса застосовує нерівність (3.13):

$$t_{PD} \le Tc - (t_{PCQ} + t_{SETUP}).$$

Отже, t_{PD} = 5 нс − (0,199 нс + 0,076 нс), тобто $t_{PD} \le$ 4,725 нс. Затримка в кожному логічному елементі в сумі із затримкою у поєднаних логічних елементах (*tLE+wire*) дорівнює 381 пс + 246 пс = 627 пс. Максимальну кількість (*N*) логічних елементів можна визначити за умови *NtLE+wire* ≤ 4,725 нс. Отже, *N* = 7.

5.6.3 Схемотехніка матриць

Для мінімізації розмірів та ціни в ПЗП і ПЛМ, замість традиційних логічних елементів, часто використовуються псевдо-*n*-МОН (*pseudo-nMOS*) або динамічні (див. п. 1.7.8) схеми.

У динамічних елементах *p*-МОН-транзистор увімкнений не весь час, що дає змогу знижувати енергоспоживання, коли його стан не має значення і його вимкнено. У всіх інших ситуаціях (у проєктуванні та використанні) динамічні та псевдо-*n*-МОН-матриці пам'яті аналогічні.

На рис. 5.62, *a* подана точкова нотація для ПЗП 4×3 біти, що реалізує такі функції: $X = A \wedge B$, $Y = \overline{A} + B$ і $Z = \overline{A}\ \overline{B}$. Це ті самі функції, що були подані на рис. 5.49, до того ж адресні входи були перепозначені як *A* і *B*, а виходи – *X*, *Y* та *Z*. Реалізація із псевдо-*n*-МОН-елементами продемонстрована на рис. 5.62, *b*. Вихід кожного декодера з'єднано із затворами *n*-МОН-транзисторів його рядка. Як відомо, у псевдо-*n*-МОН-схемах вихід пов'язано з ланцюгом живлення *p*-МОН-транзистора із значним опором каналу. Вихід має високий потенціал тільки якщо *n*-МОН-транзистор, який пов'язує його із землею, закритий. Ці транзистори розташовані на всіх перетинах, де точка відсутня. Для порівняння на рис. 5.62, *b* збережені точки точкової нотації (див. рис. 5.62, *a*). Транзистори *p*-МОН встановлюють високий логічний рівень усіх ліній слів, у яких *n*-МОН-транзистори відсутні. Наприклад, коли *AB* = 11, лінія слів 11 має високий потенціал, з'єднані з нею *n*-МОН-транзистори відкриваються і на виходах *X* і *Z* встановлюють низьку напругу.

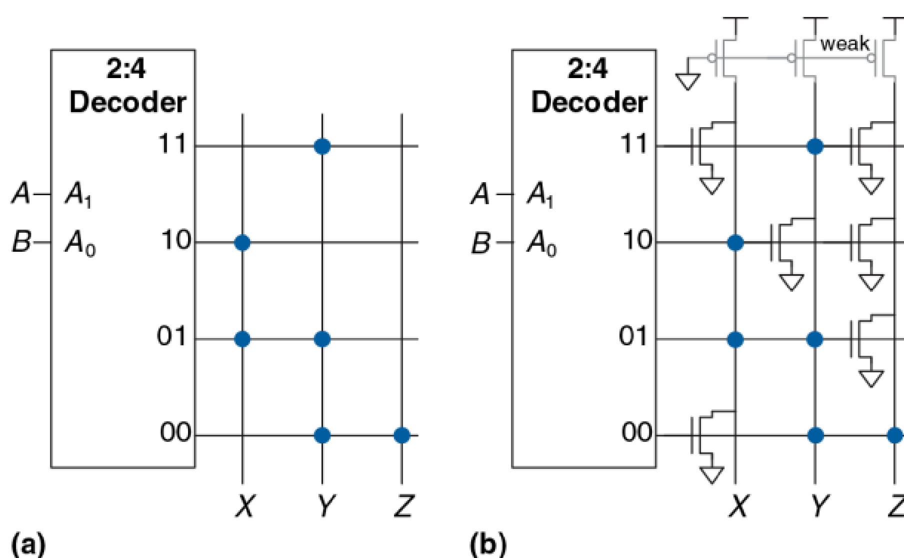


Рисунок 5.62 – Реалізація ПЗП: a – точкова нотація; b – псевдо-*n*-МОН-схема

ПЛМ також можуть бути реалізовані з використанням псевдо-*n*-МОН-схем. На рис. 5.63 продемонстровано таку реалізацію ПЛМ, що була зображена

на рис. 5.55. Транзистори *n*-МОН, що забезпечують низький рівень, розташовані на непозначених точками перетинах матриці І та в зазначених рядках матриці АБО. Стовпці матриці АБО надходять на вихід крізь інвертори. Реалізація ПЛМ подана на рис. 5.63.

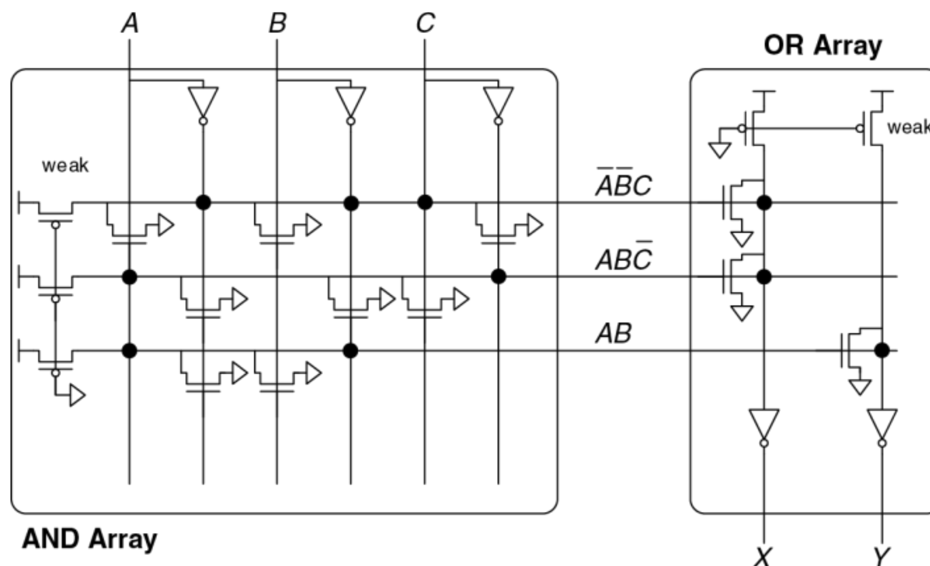


Рисунок 5.63 – Реалізація ПЛМ 3×3×2 біти
з використанням псевдо-*n*-МОН-схем

5.7 Резюме

У цьому розділі розглянуто функціональні вузли, що застосовуються в багатьох цифрових системах. Такі функціональні вузли містять арифметичні схеми, зокрема суматори, блоки віднімання, помножувачі, дільники, схеми зсуву, послідовні схеми: лічильники, регістри зсуву, логічні матриці та пристрої. У цьому розділі також описано подання дробових чисел із рухомою та фіксованою точкою.

Чимало арифметичних схем будуються за допомогою суматорів. Напівсуматор має два однобітні входи *A* і *B* та два виходи – сума й перенесення. У повному суматорі до входів напівсуматора додається вхід перенесення. *N* повних суматорів можна з’єднати послідовно і цим створити паралельний суматор, що складає два *N*-бітних числа. Такий суматор також називають суматором із послідовним перенесенням. Швидкі паралельні суматори можна створити з використанням технологій групового прискореного та префіксного перенесення.

У блоці віднімання знак другого операнда інвертується, потім виконується операція складання. Схема порівняння віднімає одне від іншого, а результат порівняння визначається за знаком різниці. У помножувачі елементи формують часткові добутки, а потім вони складаються за допомогою

повних суматорів. У схемі розподілу дільник багаторазово віднімається з часткового залишку, і за знаком різниці визначаються двійкові розряди часткового. У лічильнику зберігання стану використовується регістр, а його збільшення – суматор.

Дробові числа подаються у формах з рухомою або фіксованою точкою. Подання з фіксованою точкою аналогічне десятковому, а з рухомою – експоненційному. Для оброблення чисел із фіксованою точкою використовуються звичайні арифметичні схеми, а числа з рухомою точкою вимагають застосування більш складних схем, які виділяють і обробляють знак, порядок і мантису.

Запам’ятовувальні пристрої значного обсягу організовані у вигляді матриці слів. Пристрої мають один або більше портів для читання та/або запису слів. Вміст енергозалежної пам’яті, зокрема статичний або динамічний ОЗП, втрачається внаслідок вимкнення живлення схеми. Статичний ОЗП швидше, ніж динамічний, але використовує більше транзисторів. Регістровий файл є невеликим багатопортовим статичним ОЗП. Вміст енергонезалежної пам’яті, яка називається постійним пристроєм (ПЗП), зберігається необмежено тривалий час за відсутності живлення. Незважаючи на назву, вміст більшості сучасних ПЗП може бути змінено.

Логічні елементи також можуть бути організовані як матриці. Для виконання функцій комбінаційної логіки можуть використовуватись матриці пам’яті, в яких зберігається таблиця перетворення. ПЛМ містить з’єднані між собою конфігуровані матриці І і АБО, на ПЛМ можуть бути реалізовані тільки комбінаційні схеми. *FPGA* має значну кількість невеликих таблиць перетворення та регістрів і дає змогу реалізовувати як комбінаційні, так і послідовні схеми. Вміст таблиць перетворення та їх міжз’єднання може бути налаштовано для виконання будь-якої логічної функції. Сучасні *FPGA* можна легко перепрограмувати, вони містять значну кількість логічних елементів, що конфігуруються, дуже дешеві, що уможливлює створення на їх основі складних цифрових систем. Вони широко застосовуються як у комерційних мало- та середньосерійних виробах, так і в освітніх проєктах.

ВПРАВИ

Вправа 5.1. Чому дорівнюватиме затримка поданих 64-розрядних суматорів? Затримка будь-якого двовходового логічного елемента дорівнює 150 пс, а повного суматора – 450 пс.

a) Суматор із послідовним перенесенням.

b) Суматор із прискореним перенесенням, що містить чотирибітні блоки.

c) Префіксний суматор.

Вправа 5.2. Спроєктуйте два суматори з розповсюдженим перенесенням: 64-розрядний суматор з послідовним перенесенням та 64-розрядний суматор із прискореним перенесенням, що містить чотирибітні блоки. Використовуйте лише двовходові логічні елементи. Кожен такий елемент має площу 15 мкм^2, затримку 50 пс та повну ємність 20 пФ. Статичною потужністю можна знехтувати.

a) Порівняйте площу, затримку та споживану потужність суматорів, що працюють на частоті 100 МГц за умови напруги живлення 1,2 В.

b) Обговоріть компроміс між потужністю, площею та затримкою.

Вправа 5.3. Поясніть, чому проєктувальник може використовувати суматор із послідовним перенесенням, а не суматор із прискореним перенесенням.

Вправа 5.4. Спроєктуйте 16-розрядний сумісний префікс (рис. 5.7) з використанням мов опису апаратури. Промоделюйте та протестуйте свій модуль та продемонструйте, що він працює коректно.

Вправа 5.5. У префіксній мережі (див. рис. 5.7) для обчислення всіх префіксів застосовуються чорні осередки. Сигнали розповсюдження деяких блоків насправді не потрібні. Спроєктуйте «сіру комірку», яка отримує сигнали *G* і *P* для бітів *i:k* та *k*–1*:j*, але обчислює лише *Gi:j*, а не *Pi:j*. Переробіть кресленик префіксної мережі так, щоб, де можливо, чорні осередки були замінені на сірі.

Вправа 5.6. Префіксна мережа (див. рис. 5.7) – не єдиний спосіб обчислення всіх префіксів із логарифмічною затримкою. Мережа Когге – Стоуна є іншою поширеною префіксною мережею, що виконує ті самі функції з використанням іншого з’єднання чорних осередків. Дослідіть суматор Когге – Стоуна й накресліть схему (подібну до зображеної на рис. 5.7), на якій чорні осередки формуватимуть суматор Когге – Стоуна.

Вправа 5.7. Згадайте, що *N*-вхідний пріоритетний шифратор має *log2N* виходів, на яких формується двійкове число, що відповідає номеру найстаршого входу, на який подано 1 (див. вправу 2.36).

a) Спроєктуйте *N*-вхідний пріоритетний шифратор, у якого затримка збільшується логарифмічно зі зростанням *N*. Накресліть схему шифратора й розрахуйте його затримку, з огляду на затримки окремих логічних елементів.

b) Опишіть ваш проєкт мовою опису апаратури. Промоделюйте та протестуйте свій модуль, продемонструйте, що він працює коректно.

Вправа 5.8. Спроєктуйте подані компаратори двох 32-розрядних чисел A і B. Накресліть схеми, для яких:

a) A не дорівнює B;

b) A більше ніж B;

c) A менше ніж B або дорівнює B.

Вправа 5.9. Спроєктуйте 32-розрядний АЛП, зображений на рис. 5.15 з використанням вашої улюбленої мови опису апаратури. Модуль верхнього рівня може бути структурним або поведінковим.

Вправа 5.10. Додайте вихід *Overflow* до 32-розрядного АЛП із вправи 5.9. Цей вихід набуває значення логічного 1, якщо суматор переповнюється, інакше значення на виході 0.

a) Запишіть рівняння для виходу *Overflow*.

b) Накресліть схему, яка формує сигнал переповнення.

c) Спроєктуйте модифіковане АЛП за допомогою мови опису апаратури.

Вправа 5.11. Додайте вихід *Zero* до 32-розрядного АЛП з вправи 5.9. Вихід набуває значення логічної 1, коли $Y == 0$.

Вправа 5.12. Напишіть код середовища тестування для 32-розрядного АЛП із вправ 5.9, 5.10, 5.11 та протестуйте АЛП. Розробіть всі необхідні файли із тестовими векторами. Для переконання скептиків обов’язково детально протестуйте поведінку схеми для «незручних» даних.

Вправа 5.13. Спроєктуйте схему, яка зсуває 32-бітний вхід ліворуч на два біти. Вихід також містить 32 біти. Поясніть роботу вашого проєкту словами та накресліть його схему. Реалізуйте проєкт за допомогою вашої улюбленої мови опису апаратури.

Вправа 5.14. Розробіть 4-бітну схему циклічного зсуву вліво та вправо. Накресліть схему вашого проєкту. Реалізуйте проєкт за допомогою вашої улюбленої мови опису апаратури.

Вправа 5.15. Спроєктуйте 8-бітну схему зсуву вліво, використовуючи лише 24 мультиплексори 2:1. На вхід схеми надходить 8-бітний вхідний сигнал та 3-бітна величина зсуву, $shamt_{2:0}$. На виході схеми формується 8-бітний сигнал Y. Накресліть відповідну схему.

Вправа 5.16. Поясніть, як можна побудувати будь-яку *N*-бітну схему зсуву або циклічного зсуву, використовуючи мультиплексори *Nlog2N* 2:1.

Вправа 5.17. Дворівнева схема зсуву, зображена на рис. 5.64, може виконувати будь-яку *N*-бітну операцію зсуву або циклічного зсуву. Вона зрушує 2*N*-бітний вхід праворуч на *k* бітів. *N* молодших бітів результату надходять на вихід *Y*. Старші *N* бітів входу позначені як *B*, молодші *N* бітів – *C*.

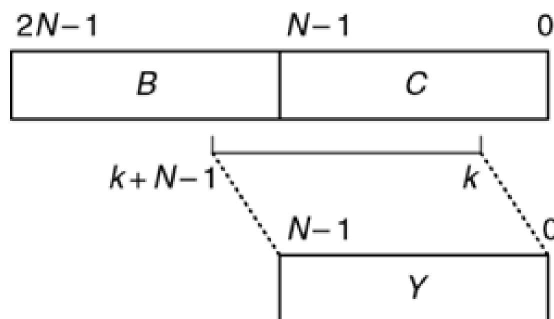


Рисунок 5.64 – Дворівнева схема зсуву

У разі відповідного вибору *B, C, k* дворівнева схема зсуву може виконувати будь-який зсув або циклічний зсув. Поясніть, як *B, C* і *k* пов’язані з *A, shamt* і *N* для виконання:

a) логічного зсуву *A* вправо на *shamt*;

b) арифметичного зсуву *A* вправо на *shamt*;

c) зсуву *A* вліво на *shamt*;

d) циклічного зсуву *A* вправо на *shamt*;

e) циклічного зсуву *A* вліво на *shamt*.

Вправа 5.18. Знайдіть критичний шлях і час проходження сигналу по ньому для помножувача 4×4, зображеного на рис. 5.18, вважаючи відомими затримки елемента І (*tAND*) та суматора (*tFA*). Чому дорівнюватиме затримка аналогічного помножувача *N*×*N*?

Вправа 5.19. Знайдіть критичний шлях та час проходження сигналу по ньому для схеми поділу 4×4, зображеної на рис. 5.20, вважаючи відомими затримки мультиплексора 2:1 (*tMUX*), суматора (*tFA*) та інвертора (*tINV*). Чому дорівнюватиме затримка аналогічної схеми розподілу *N*×*N*?

Вправа 5.20. Спроєктуйте помножувач, що працює з числами, наведеними в додатковому коді.

Вправа 5.21. Модуль розширення знака збільшує кількість розрядів числа, поданого в додатковому коді, з *M* до *N* (*N* > *M*) способом копіювання

найстаршого розряду входу до старших розрядів виходу (див. п. 1.4.6). Модуль має M-розрядний вхід A та N-розрядний вихід Y. Накресліть схему модуля розширення знака з 4-розрядним входом та 8-розрядним виходом. Опишіть ваш проєкт мовою опису апаратури.

Вправа 5.22. Модуль доповнення нулями збільшує кількість розрядів беззнакового числа M до N ($N > M$) способом присвоєння старшим розрядам виходу нульового значення. Накресліть схему модуля доповнення нулями з 4-розрядним входом та 8-розрядним виходом. Опишіть ваш проєкт мовою опису апаратури.

Вправа 5.23. Порахуйте $111001.000_2/001100.000_2$ у двійковій системі числення, використовуючи стандартний шкільний алгоритм розподілу.

Вправа 5.24. Числа якого діапазону можна подати за допомогою перелічених форматів?

a) 24-бітне беззнакове число з фіксованою точкою з 12 бітами цілої частини та 12 дробовою;

b) 24-бітне число в прямому коді з фіксованою точкою з 12 бітами цілої частини та 12 дробової;

c) 24-бітове число в додатковому коді з фіксованою точкою з 12 бітами цілої частини та 12 дробової.

Вправа 5.25. Подайте наведені десяткові числа у 16-розрядному двійковому форматі в прямому коді з 8 бітами цілої частини та 8 дробової. Подайте вашу відповідь у шістнадцятковій системі.

a) –13.5625;

b) 42.3125;

c) –17.15625.

Вправа 5.26. Подайте наведені десяткові числа у 12-розрядному двійковому форматі в прямому коді з 6 бітами цілої частини та 6 дробовою. Подайте вашу відповідь у шістнадцятковій системі.

a) –30.5;

b) 16.25;

c) –8.078125.

Вправа 5.27. Подайте десяткові числа з вправи 5.25 у 16-розрядному двійковому форматі в додатковому коді з 8 бітами цілої частини та 8 дробової. Подайте вашу відповідь у шістнадцятковій системі.

Вправа 5.28. Подайте десяткові числа з вправи 5.26 у 12-розрядному двійковому форматі в додатковому коді з 6 бітами цілої частини та 6 дробової. Подайте вашу відповідь у шістнадцятковій системі.

Вправа 5.29. Подайте десяткові числа з вправи 5.25 у форматі з рухомою точкою та одинарною точністю відповідно до стандарту *IEEE* 754. Подайте вашу відповідь у шістнадцятковій системі.

Вправа 5.30. Подайте десяткові числа з вправи 5.26 у форматі з рухомою точкою та одинарною точністю відповідно до стандарту *IEEE* 754. Подайте вашу відповідь у шістнадцятковій системі.

Вправа 5.31. Перетворіть наведені двійкові числа з фіксованою точкою, що задані в додатковому коді, на десяткові. Для простоти двійкова точка в цьому прикладі показана явно.

a) 0101.1000;

b) 1111.1111;

c) 1000.0000.

Вправа 5.32. Повторіть вправу 5.31 для поданих двійкових чисел із фіксованою точкою, заданих у додатковому коді.

a) 011101.10101;

b) 100110.11010;

c) 101000.00100.

Вправа 5.33. За умови додавання двох чисел з рухомою точкою мантиса числа з меншим порядком зрушується. Для чого це робиться? Поясніть словами та підтвердіть свій аргумент прикладом.

Вправа 5.34. Складіть подані числа, задані у форматі з рухомою точкою та одинарною точністю відповідно до стандарту *IEEE* 754.

a) C0123456 + 81C564B7;

b) D0B10301 + D1B43203;

c) 5EF10324 + 5E039020.

Вправа 5.35. Складіть подані числа, задані у форматі з рухомою точкою та одинарною точністю відповідно до стандарту *IEEE* 754.

a) C0D20004 + 72407020;

b) C0D20004 + 40DC0004;

c) (5FBE4000 + 3FF80000) + DFDE4000.

Чому отримані результати парадоксальні? Поясніть.

Вправа 5.36. Модифікуйте процедуру складання чисел із рухомою точкою, описану в п. 5.3.2, для виконання обчислень як з позитивними, так і з негативними числами.

Вправа 5.37. Розглянемо числа, задані у форматі з рухомою точкою та одинарною точністю відповідно до стандарту *IEEE* 754.

a) Скільки чисел можна подати в такому форматі? На особливі випадки ±∞ або *NaN* не потрібно брати до уваги.

b) Скільки додаткових чисел можна подати, якщо не вводити до розгляду особливі випадки ±∞ або *NaN*?

c) Поясніть, чому для ±∞ та *NaN* виділено спеціальне подання.

Вправа 5.38. Розглянемо такі десяткові числа: 245 та 0.0625.

a) Запишіть ці цифри у форматі з рухомою точкою та одинарною точністю. Подайте вашу відповідь у шістнадцятковій системі.

b) Порівняйте величини двох 32-розрядних чисел, отриманих у завданні (a). Іншими словами, інтерпретуйте два 32-розрядні числа як числа в додатковому коді та порівняйте їх. Чи даватиме порівняння таких цілих чисел коректний результат?

c) Ви вирішили запропонувати новий формат із рухомою точкою та одинарною точністю. Єдина розбіжність із стандартом *IEEE* 754 чисел з рухомою точкою та одинарною точністю полягає в тому, що ви пропонуєте для порядку використовувати додатковий код, а не зміщення. Запишіть два числа відповідно до нового стандарту. Подайте вашу відповідь у шістнадцятковій системі.

d) Чи буде ціле порівняння працювати з новим форматом із завдання (c)?

e) Чому зручно використовувати алгоритм порівняння цілих чисел для чисел з рухомою точкою?

Вправа 5.39. Спроєктуйте суматор чисел із рухомою точкою та одинарною точністю за допомогою вашої улюбленої мови опису апаратури. Перед написанням коду накресліть схему проєкту.

Промоделюйте та протестуйте ваш суматор, щоб довести скептикам, що він працює коректно. Ви можете обмежитися використанням лише позитивних чисел та виконувати округлення до нуля (виконувати усічення). Також ви не можете розглядати особливі випадки, наведені в табл. 5.2.

Вправа 5.40. У цій вправі вам необхідно спроєктувати 32-бітний помножувач із рухомою точкою. Помножувач має два 32-бітні входи для чисел

з рухомою точкою і один 32-бітний вихід. Ви можете обмежитися використанням лише позитивних чисел та виконувати округлення до нуля (виконувати усічення). Також ви не можете розглядати особливі випадки, подані в табл. 5.2.

a) Опишіть послідовність кроків, необхідних для множення 32-бітних чисел з рухомою точкою.

b) Накресліть схему 32-бітного помножувача з рухомою точкою.

c) Опишіть 32-бітний помножувач з рухомою точкою мовою опису апаратури. Промоделюйте та протестуйте ваш помножувач, щоб довести скептикам, що він працює коректно.

Вправа 5.41. У цій вправі вам потрібно спроєктувати 32-бітний префіксний суматор.

a) Накресліть схему вашого проєкту.

b) Спроєктуйте 32-бітовий сумісний префікс із використанням мови опису апаратури. Промоделюйте та протестуйте ваш суматор і покажіть, що він працює коректно.

c) Чому дорівнює затримка 32-бітного префіксного суматора, який спроєктовано в завданні (a)? Затримка кожного двовходового логічного елемента дорівнює 100 пс.

d) Спроєктуйте конвеєрну версію 32-бітного префіксного суматора, накресліть його схему. Наскільки швидко працюватиме конвеєрний префіксний суматор? Втрати на упорядкування (t_{PCQ} + t_{SETUP}) дорівнюють 80 пс. Спроєктуйте суматор так, щоб він мав максимально можливу швидкодію.

e) Спроєктуйте 32-бітний конвеєрний префіксний суматор із використанням мови опису апаратури.

Вправа 5.42. Інкрементор до *N*-розрядного числа додає 1. Побудуйте 8-розрядний інктрементор за допомогою напівсуматорів.

Вправа 5.43. Побудуйте 32-розрядний синхронний реверсивний лічильник (*Up/Down counter*). Він має входи *Reset* та *Up*. Коли вхід *Reset* встановлений в 1, усі виходи скидаються в 0. Інакше, якщо *Up* = 1, лічильник рахує вгору, а коли *Up* = 0 – вниз.

Вправа 5.44. Спроєктуйте 32-розрядний лічильник, стан якого збільшується на 4 по кожному фронту тактового імпульсу. Лічильник має входи скидання і тактових імпульсів. Після скидання всі виходи лічильника встановлюються в 0.

Вправа 5.45. Змініть лічильник із вправи 5.44 так, щоб залежно від сигналу керування *Load* лічильник або збільшував свій стан на 4 або завантажував нове 32-розрядне значення *D*. Коли *Load* = 1, лічильник завантажує нове значення, подане на вхід *D*.

Вправа 5.46. *N*-розрядний лічильник Джонсона (*Johnson counter*) містить *N*-розрядного регістра зсуву, що має сигнал скидання. Вихід регістра зсуву (S_{out}) інвертується та подається назад на його вхід (S_{in}). Коли лічильник скидається, всі його розряди набувають нульового значення.

a) Знайдіть послідовність значень $Q_{3:0}$, що з'являється на виході 4-розрядного лічильника Джонсона безпосередньо після скидання.

b) Після скількох циклів послідовність на виході *N*-розрядного лічильника Джонсона повторюватиметься? Поясніть.

c) Спроєктуйте десятковий лічильник за допомогою 5-розрядного лічильника Джонсона, 10 елементів І та інверторів. Десятковий лічильник має входи тактового сигналу та скидання та вихід $Y_{9:0}$ з прямим кодуванням «1 з 10». Після скидання активується вихід Y_0. Після кожного циклу активується наступний вихід. Після 10 циклів стан лічильника повторюється. Накресліть схему десяткового лічильника.

d) Які переваги може мати лічильник Джонсона порівняно із звичайними лічильниками?

Вправа 5.47. Створіть *HDL*-опис 4-бітного сканованого регістра, подібного до рис. 5.37. Промоделюйте та протестуйте свій *HDL*-модуль та покажіть, що він працює коректно.

Вправа 5.48. Англійська мова має велику надмірність, що дає змогу відновити спотворену передачу. Двійкові дані також можуть бути передані надмірністю, яка може використовуватися для виправлення помилок. Наприклад, число 0 буде закодовано як 00000, а число 1 як 11111. Інформація передається крізь зашумлений канал, що здатний інвертувати 1 або 2 біти. Приймач може відновити вихідні дані, якщо в посилці, що відповідає 0, буде принаймні три (з п'яти) бітів 0, аналогічно для 1 буде не менше ніж трьох бітів 1.

a) Запропонуйте кодування для передачі двобітних блоків 00, 01, 10 та 11 з використанням 5 бітів, що дає змогу виправляти всі однобітні помилки. Підказка: кодування 00000 та 11111 для 00 та 11, відповідно, не працюватиме.

b) Спроєктуйте схему, що прийматиме п'ятибітний блок кодованих даних і декодуватиме його у двобітний блок (00, 01, 10, та 11), навіть якщо 1 біт було спотворено в процесі передачі.

c) Припустимо, ви хочете використовувати альтернативне п’ятибітне кодування. Як можна реалізувати цей проєкт для забезпечення зміни кодування без заміни апаратного забезпечення?

Вправа 5.49. Флеш ЕСППЗП, або просто флеш-пам’ять, є відносно недавнім винаходом, що революційно змінив ринок споживчої електроніки. Вивчіть та поясніть, як працює флеш-пам’ять. Щоб пояснити принцип роботи рухомого затвора, використовуйте діаграми. Поясніть, як відбувається запис інформації до флеш-пам’яті. Оформіть посилання на використані ресурси.

Вправа 5.50. Учасники проєкту дослідження позаземного життя виявили, що на дні озера Моно живуть інопланетяни. Для класифікації інопланетян за можливими планетами походження на основі інформації *NASA* (зелений мул, коричневий колір шкіри, слизова оболонка, потворність) необхідно створити цифрову схему. Детальні консультації з позаземними біологами привели до таких висновків:

● якщо інопланетянин 1) зелений і слизкий або 2) потворний, коричневий та слизовий, то він може бути марсіаніном;

● якщо істота 1) потворна, коричнева та слизька або 2) зелена, не потворна й не слизька – вона може бути з Венери;

● якщо істота 1) коричнева, не потворна й не слизька або 2) зелена й слизька – вона може бути з Юпітера.

Ці дослідження все ще не зовсім точні: наприклад, форма життя з плямами зеленого та коричневого кольору, слизька, але не потворна, може бути з Марса чи Юпітера.

a) Запрограмуйте 4×4×3 ПЛМ для ідентифікації прибульця. Ви можете використовувати точкову нотацію.

b) Запрограмуйте 16×3 ПЗП для ідентифікації прибульця. Ви можете використовувати точкову нотацію.

c) Реалізуйте свій проєкт на *HDL*.

Вправа 5.51. Реалізуйте такі функції за допомогою одного 16×3 ПЗП. Застосовуйте точкову нотацію для опису вмісту пам’яті.

a) $X = AB + B\overline{C}D + \overline{A}\,\overline{B}$;

b) $Y = AB + BD$;

c) $Z = A + B + C + D$.

Вправа 5.52. Реалізуйте функції з вправи 5.50 за допомогою 4×8×3 ПЛМ. Ви можете використовувати точкову нотацію.

Вправа 5.53. Визначте розмір ПЗП, що можна застосувати для програмування поданих комбінаційних схем. Чи використання ПЗП є хорошим проєктним рішенням для реалізації цих функцій? Поясніть, чому так чи ні.

a) 16-бітний суматор / відніматор з C_{in} та C_{out};

b) помножувач 8×8;

c) 16-бітний пріоритетний шифратор (див. вправу 2.36).

Вправа 5.54. На рис. 5.65 показано низку схем, у яких використовується ПЗП. Чи можна схему в стовпці I замінити схемою зі стовпця II того самого рядка за умови належного програмування ПЗП?

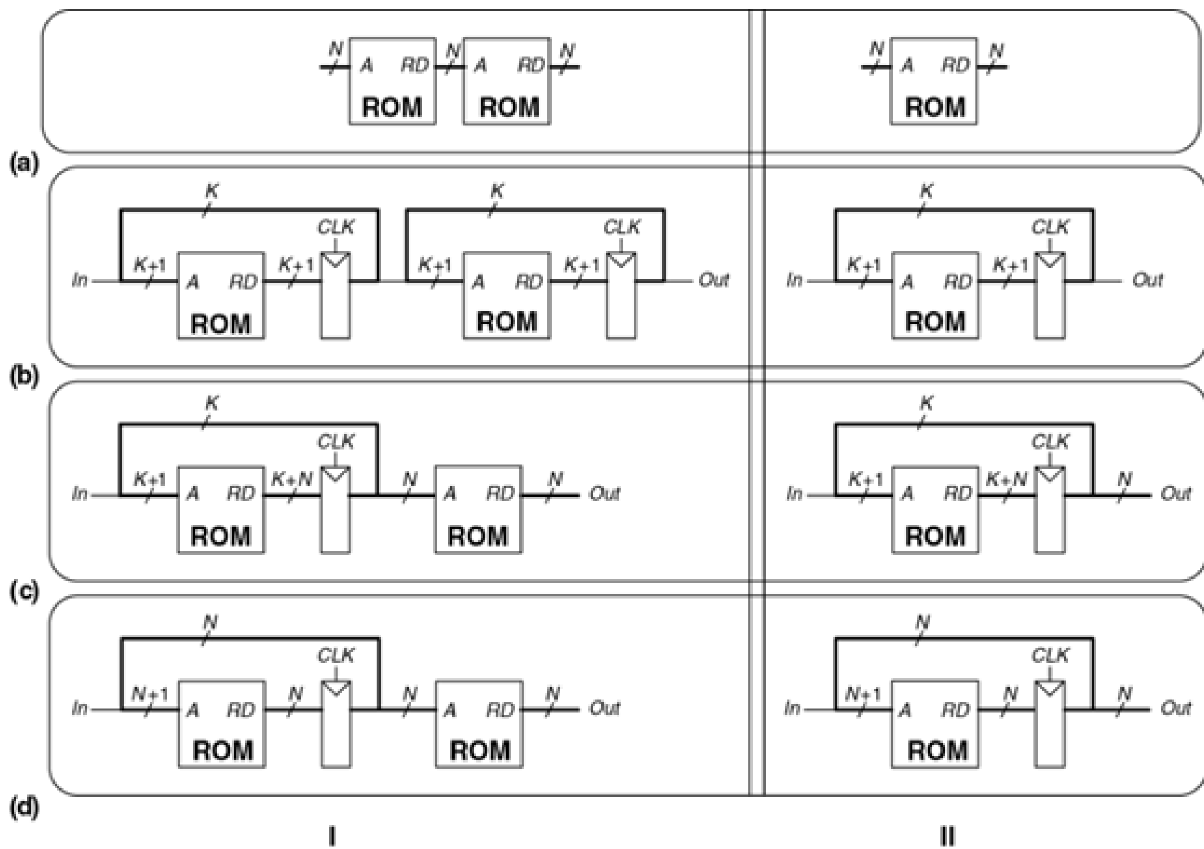


Рисунок 5.65 – Схеми на основі ПЗП

Вправа 5.55. Скільки логічних елементів *FPGA Cyclone IV* необхідно для реалізації наведених нижче функцій? Покажіть, як потрібно конфігурувати один чи кілька логічних елементів. Під час розроблення конфігурації не користуватися програмами синтезу.

a) комбінаційна функція із вправи 2.13 (c);

b) комбінаційна функція із вправи 2.17 (c);

c) функція з двома виходами із вправи 2.24;

d) функція із вправи 2.35;

e) чотиривходовий пріоритетний шифратор (див. вправу 2.36).

Вправа 5.56. Повторіть вправу 5.54 для таких опцій:

a) восьмивходовий пріоритетний шифратор (див. вправу 2.36);

b) 3:8 декодер;

c) чотирибітний суматор із послідовним перенесенням (без входу та виходу перенесення);

d) кінцевий автомат із вправи 3.22;

e) лічильник, вихід якого подано в коді Грея, із вправи 3.27.

Вправа 5.57. На рис. 5.58 зображено логічний елемент *FPGA Cyclone IV LE*. У табл. 5.5 подано його часові параметри.

a) Яка мінімальна кількість логічних елементів *FPGA Cyclone IV* необхідна для реалізації показаного на рис. 3.26 кінцевого автомата?

b) Чому дорівнює максимальна тактова частота, на якій цей кінцевий автомат стабільно працюватиме за відсутності розфазування тактових імпульсів?

c) Чому дорівнює максимальна тактова частота, на якій цей кінцевий автомат надійно працюватиме, якщо максимальне розфазування тактових імпульсів дорівнює 3 нс?

Таблиця 5.5 – Часові властивості *Cyclone IV*

Найменування	**Величина (пс)**
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (одного *LE*)	381
t_{wire} (між *LE*)	246
t_{skew}	0

Вправа 5.58. Повторіть вправу 5.56 для кінцевого автомата, зображеного на рис. 3.31, *b*.

Вправа 5.59. Ви збираєтеся використовувати *FPGA* для реалізації сортувальника льодяників *M&M*. У машині буде колірний сенсор та мотор, який відправляє червоні льодяники в одну банку, а зелені – в іншу. Проєкт буде реалізовано як кінцевий автомат із застосуванням *FPGA Cyclone IV*. Часові властивості *FPGA* подано в табл. 5.1. Кінцевий автомат працює на частоті 100 МГц. Яка максимальна кількість логічних елементів може входити в критичний шлях? Чому дорівнює максимальна частота, на якій працюватиме кінцевий автомат?

ЗАПИТАННЯ ДЛЯ СПІВБЕСІДИ

Подаємо приклади типових запитань,
які можуть бути поставлені претендентам
під час пошуку роботи
в галузі проєктування цифрових систем.

Запитання 5.1. Чому дорівнює найбільший можливий результат перемноження двох беззнакових N-бітних чисел?

Запитання 5.2. У двійково-десятковому поданні (BCD) для кожного десяткового розряду використовується 4 біти. Наприклад, 42_{10} буде подано як 01000010_{BCD}. Поясніть, чому процесор може застосовувати двійково-десяткове подання.

Запитання 5.3. Спроєктуйте суматор, що складатиме два беззнакові 8-бітні числа в двійково-десятковому поданні (див. запитання 5.2). Накресліть схему та створіть *HDL*-опис вашого суматора. Суматор має вхідні сигнали A, B та C_{in}, вихідні – S та C_{out}. Сигнали C_{in} та C_{out} є однобітним входом і виходом перенесення, A, B і S – 8-бітні числа у двійково-десятковому поданні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Тарарака В. Д. Архітектура комп'ютерних систем: навч. посіб. Житомир: ЖДТУ, 2018. 383 с.

2. Аврунін О. Г., Носова Т. В., Семенець В. В. Основи мови VDHL для проектування цифрових пристроїв на ПЛІС. Харків: ХНУРЕ, 2018. 196 с.

3. Зубков О. В., Свид І. В., Воргуль О. В., Семенець В. В. Програмування мікроконтролерів STM32 в середовищі STM32CubeIDE в прикладах і задачах: навч. посіб. Дніпро: ЛІРА ЛТД, 2022. 144 с.

4. VHDL-технології проектування електронних пристроїв: навч. посіб. / С. Ю. Леонов, Т. В. Гладких, О. І. Баленко. Київ: КАФЕДРА, 2014. 423 с.

5. Математичне моделювання інформаційних систем: навч. посіб. / І. І. Обод, І. В. Свид, І. В. Рубан, Г. Е. Заволодько; за ред. І. І. Обода. Харків: Мадрид, 2019. 270 с.

6. Svyd I., Maltsev O., Zubkov O., Saikivska L. Matlab Use in Design of Digital Systems on the FPGA in CAD Xilinx VIVADO. *First International Scientific and Practical Conference «Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs» MC&FPGA-2019* / Kharkiv, Ukraine, July 26–27, 2019. Kharkiv, 2019. P. 29–30. DOI: 10.35598/mcfpga.2019.010.

7. Кузнецов М. В. Використання Matlab при проектуванні цифрових систем на ПЛІС у САПР Xilinx Vivado / науковий керівник – к. т. н., доц. Свид І. В. *ІІ Всеукраїнська науково-практична конференція молодих учених, курсантів та студентів «АВІАЦІЯ, ПРОМИСЛОВІСТЬ, СУСПІЛЬСТВО»: зб. матеріалів форуму*. Кременчук: Кременчуцький льотний коледж, 2019. С. 222–224.

8. Reidenbach B. Practical digital design: an introduction to VHDL / Bruce Reidenbach. Purdue University Press, 2022. 445 p.

9. Harris D. Digital Design and Computer Architecture: RISC-V Edition / David Harris, Sarah L. Harris. ELSEVIER, 2022. 810 p.

10. Unsalan, Cem. Digital system design with FPGA: Implementation using Verilog and VHDL / Cem Unsalan, Bora Tar. McGray-Hill Education, 2017. 546 p.

11. Матвієнко М. П. Проектування цифрових пристроїв: підручник. Київ: Ліра-К, 2018. 364 с.

12. Chatterjee A., Das C. K. Energy-Efficient Microprocessors for Embedded Systems: Design, Simulation, and Optimization. New York: Springer, 2015.

13. Мірошник М. А., Клименко Л. А., Корольова Я. Ю. Технології та автоматизація проектування цифрових пристроїв складних комп'ютерних систем на ПЛІС: навч. посіб. Харків: УкрДУЗТ, 2021. 220 с.

14. Vorgul O., Svyd I., Zubkov O. Neuron Networks Design in Matlab and Vivado. *III International Scientific and Practical Conference Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs (MC&FPGA)*. Kharkiv, Ukraine, 2021, P. 29–31, doi: 10.35598/mcfpga.2021.010.

15. Свид І. В., Литвиненко О. В., Білоцерківець О. Г. Особливості проектування цифрових пристроїв на базі FPGA Xilinx в САПР Vivado HLx Design Suite. *Спеціалізована виставка «KharkivProm Days. Виробництво і ефективність»: збірник матеріалів форуму секції «Автоматизація, електроніка та робототехніка. Стратегії розвитку та інноваційні технології»*. Харків: ХНУРЕ; Виставкова компанія ADT, 2019. С. 43–44.

16. Старкова А. В. Використання мови опису апаратних засобів VHDL у MATLAB. *Авіація, промисловість, суспільство: матеріали IV міжнар. наук.-практ. конф.*, 18 трав. 2023 р. Харків: ХНУВС, 2023. С. 165–167.

17. Чумак В. С. Аналіз надійності ПЛІС. Порівняння ризиків, пов'язаних з використанням ПЛІС і мікропроцесорів / науковий керівник – к. т. н., доц. Свид І. В. *24-й міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті»: зб. матеріалів форуму.* Харків: ХНУРЕ, 2020. Т. 3. С. 186–187.

18. Avrunin O., Sakalo S., Semenets V. Development of up-to-date laboratory base for microprocessor systems investigation. *19th International Crimean Conference Microwave & Telecommunication Technology Sevastopol*, 2009. P. 301–302.

19. Oleg G. Avrunin, T. Nosova, V. Semenets. Experience of Developing a Laboratory Base for the Study of Modern Microprocessor Systems. *Proceedings of I International Scientific and Practical Conference "Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs" MC&FPGA-2019*. 2019. P. 6–8.

20. Special Features of the Educational Component "Design of Devices on Microcontrollers and FPGA" / I. Svyd, O. Vorgul, V. Semenets, O. Zubkov, V. Chumak, N. Boiko. *II International Scientific and Practical Conference Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs (MC&FPGA)*. Kharkiv, Ukraine, 2020. P. 55–57. doi: 10.35598/mcfpga.2020.017.

Електронне навчальне видання

АВРУНІН Олег Григорович
НОСОВА Тетяна Віталіївна
ПРАСОЛ Ігор Вікторович
СЕМЕНЕЦЬ Валерій Васильович
ЧУГУЙ Євген Анатолійович

ОСНОВИ МОВ *SYSTEMVERILOG* ТА *VHDL* ДЛЯ ПРОЄКТУВАННЯ ЦИФРОВИХ ПРИСТРОЇВ НА ПЛІС У ПРИКЛАДАХ І ЗАДАЧАХ

Навчальний посібник

Рецензенти:

Н.Г. Косуліна, д-р техн. наук, проф., проф. кафедри електромеханіки, робототехніки, біомедичної інженерії та електротехніки Державного біотехнологічного університету;

С.В. Павлов, д-р техн. наук, проф., проф. кафедри біомедичної інженерії Вінницького національного технічного університету.

Відповідальний випусковий О.Г. Аврунін
Редактор Л.В. Кузьміна
Комп’ютерна верстка Л.Ю. Свєтайло

План 2024 (друге півріччя), поз. 5.

Підп. до використання 31.10.2024 Формат pdf. Обсяг даних 5,26 Mb

ХНУРЕ. Україна. 61166, Харків, просп. Науки, 14, E-mail: info@nure.ua

Підготовлено в редакційно-видавничому відділі ХНУРЕ
Свідоцтво суб’єкта видавничої справи ДК №1409 від 26.06.2003