УДК 681.325

Д.А. ПЕТРОСОВ, В.Ю. ЦУКАНОВ

MISC - КОМПИЛЯТОР

Приводятся принципы построения компилятора, работающего на структуре микропрограммируемого микропроцессора MISC – архитектуры; программные модели самого компилятора. Рассматривается также вопрос выбора базовой системы простейших команд, выполняющихся за один такт.

Введение

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только очень незначительная часть программного обеспечения, требующая особой эффективности, программируется с помощью ассемблеров. В настоящее время распространено довольно много языков программирования. Широкое распространение получили так называемые "универсальные языки" (Паскаль, Си и другие), а также некоторые специализированные (например, язык обработки списочных структур Лисп). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ. Для некоторых языков имеется довольно много реализаций. Например, реализаций Паскаля или Си для ЭВМ типа IBM/PC на рынке десятки. С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектуры ЭВМ. Это развитие идет по различным направлениям. Совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным линиям. Это можно проиллюстрировать на примере микропроцессора Intel - 80x86. Его последние версии отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, значит, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых) [6].

В рамках традиционных последовательных машин возникает большое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC. Такие ведущие фирмы как Intel, Motorola, Sun, DEC перешли на выпуск машин с RISC-ядром. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

В данной работе создано первое приближение MISC-компилятора для MISC- архитектуры процессора.

1.Принципы построения MISC- компилятора

Для создания компилятора MISC-процессора рассмотрим подробно структуру микропрограммируемого микропроцессора [1,4].

Логическая структура MISC состоит из двух частей: основная (или RISC) часть и управляющая память (микропрограммное управление). Основная часть Host — одноуровневый RISC с глобальным микропрограммированием задачи, где выполняется 40-50 простых однотактных команд. Если же необходимо выполнить команду, не принадлежащую к числу простых, то используется ПЗУ микропрограммного управления. При отсутствии блока микропрограммного управления процессор работает как чистый RISC. Преимущество такого разделения очевидно. Библиотека стандартных функций (некоторые из них CISC-процессоры выполняют одной, хотя и сложной командой) для RISC-процессора занимает много места в памяти. Обращения к памяти RISC-процессора очень часты и занимают много времени, поэтому выполнение стандартных функций может понизить быстродействие RISC. Именно поэтому в предлагаемой структуре библиотеки стандартных функций расположены в ПЗУ МУ. Подобный подход решает главную проблему: для команд микропрограммы не тратится время на выборку и дешифрацию команды, так как в ПЗУ уже дешифрованная последовательность RISC-команд, видимая программистом, как единая CISC-команда.

Компилятор для такой структуры должен выполнять следующее:

1. Программа на языке высокого уровня (ЯВУ) — исходный код преобразуется в объектный код. Любой компилятор с ЯВУ для RISC-архитектуры генерирует микрокод в простые команды. В MISC-компиляторе проблема потери времени на поэтапное выполнение стандартных функций решена.

2. Глобальная микропрограмма, полученная после первого этапа, просматривается по определенным правилам [7].

3. Исполнение микропрограммы. Простая микрокоманда обрабатывается простой однотактной командой процессора или несколькими из базовой системы команд. Сложная команда – по принципу "склеил - исполнил".

Ниже приведена часть компилятора, написанного на Си [5] и выполняющего перевод исходного кода в глобальную микропрограмму. В качестве языка высокого уровня был выбран Си.

```
// File MISC Compiler Part4
# include<stdio. h>
# include"cip.def"
extern char
*mach,
*cptr, *symtab,
# ifdef OPTIMIZE
 optimize,
```

```
    #endif
    *stagenext, ssname[NAMESIZE];
    extern int
    beglab, csp, output;
    #include "msc41.cpp"
    #include "msc42.cpp"

    FILE: misc41.cpp

    header ()
    { beglab=getlabel();}
    trailer()
    { #ifdef LINK
    if ((beglab==1)|(beglab>9000))}}
    #else
    char *ptr;
     cptr=STARTGLB;
    while(cptr<ENDGLB)
    {if(cptr[IDENT}==FUNCTION&& cptr[CLASS]==AUTOEXT)
      external (cptr+NAME);
    cprt+=SYMMAX;}
    #ifdef UPPER
      if ((ptr=findglb("MAIN"))&&(prt[OFFSET]==FUNCTION))
    #else
if((ptr=findglb("main"))&&(ptr[OFFSET]==FUNCTION))
    #endif external("Ulink");
    #endif ol("END");}
    loadargc(val) int val;
    {if(searcn("NOCCARGC",macn,NAMESIZE+2,MACNEND,MACNBR,0)==0)
    {if(val)
      {out("MVI A,");
     outdec(val);
     nl();}
     else ol("XRA A"); } }
    entry()
    {out (ssname);
    col();
    #ifdef LINK
    col();
    #endif
    nl();}
    external(name) char *name;
    { #ifdef LINK
      ot("EXT");
     ol(name);
```



```
    #endif}
    indirect(lval) int lval[];
    { if(lval[1]==CCHAR) ffcall("CCGHAR##");
    else           ffcall("CCGINT##"); }
    getmem(lval) int lval[];
    {char *sym;
    sym=lval[0];
    if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR))}
    ot("LDA");
    outstr(sym+NAME);
    nl();
    ffcall("CCSXT##");}
       else{
       ot("LHLD");
      outstr(sym+NAME);
     nl();}}
    getloc(sym) char *sym;
    { const(getint(sym+OFFSET, OFFSIZE)-csp);
    ol("DAD SP");}
    putmem(lval) int lval[];
    {char *sym;
    sym=lval[0];
    if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)
       {ol("MOV A,L");
      ol("STAX D");}
     else ffcall("CCPINT##");}
    move()
    {ol("MOV D,H");
    ol("MOV E,L");}
    swap()
    {ol("XCHG;;");}
    immed()
    {ot("LXI H,");}
    immed2()
    {ot("LXI D,");}
    push()
    {ol("PUSH H");
    csp=csp-BPW;}
    smartpop(lval, start) int lval[]; char *start;
    {if(lval[5] pop();
    else unpush(start);}
    unpush(dwst) char *dest;
    {int i;
    char *sour;
     sour="XCHG;;";
```

```
while(*sour) *dest++ = *sour++;
sour=stagenext;
   while(—sour>dest   /*adjust stack references*/
 {if(streq(sour,"DAD SP))
  {—sour;
   i=BPW;
   while(isdigit(*(—sour)))
   {if((*sour=*sour-i)<'0'
     {*sour=*sour+10;
      i=1;}
   else i=1;}}}
 csp=csp+BPW;  }
 pop()
 {ol("POP D");
 csp=csp+BPW;}
 swapstk()
 {ol("XTHL");}
 sw()
 {ffcall("CCSWITCH##");}
 ffcall(sname) char *sname:
 {ot("CALL");
 outstr(sname);
 nl(); }
 ffret()
 {ol("RET");}
 callstk()
 {ffcall("CCDCAL##");}
 jump(label) int label;
 { ot("JMP");
 printlabel(label);
 nl();}
 testjump(label) int label;
 {ol(MOV A,H");
 ol("ORA L");
 ot("JZ");
 printlabel(label);
 nl();}
 zerojump(oper, label, lval) int (*oper)(),label, lval[];
 { clearstage(lval[7],00;
 (*oper)(label);}
 defstorage(size) int size;
 { if(size==1) ot("DB");
  else          ot("DW");}
 point()
 {ol("DW $+2");}
```



```
 modstk(newsp, save) int newsp;
 int k;
 k=newsp-cspo;
 if(k==0)return newsp;
 if(>=0)
 {if(k<7){
  if(k&1){
    ol("INX SP");
    k—; }
  While(k){
     ol("PUSH B");
      k=k+BPW;}
  return newsp;}  }
 if(save)swap();
 const(k);
 ol("DAD SP");
 ol("SPHL");
 if(save)swap();
 return newsp;}
doublereg(){
 ol("DAD H");}

 FILE : misc42.cpp

 ffadd(){
 ol("DAD D");}
 ffsub()
 {ffcall("CCSUB##");}
 ffmult()
 {ffcal(CCMULT##");}
 ffdiv()
 {ffcall(CCDIV##);}
 ffmod()
 {ffdiv();swap();}
 ffor()
 {ffcall("CCOR##");}
 ffxor()
 {ffcall("CCXOR##");}
 ffand()
 {ffcall("CCAND##");}
 lneg()
 {ffcall("CCLNEG##");}}
```

На первом этапе (чтение исходного кода и сканирование) происходит заполнение таблиц, определяющих отношение имен форматов пользователя и имен констант со значениями полей бит.

На втором этапе определяются символы для форматов, символы для констант и разрядность слова микрокоманды.

Следующая фаза – генерация кода: производится чтение символической программы, присвоение меток, установка адресного счетчика, трансляция в двоичный код, формирование листингов и таблиц соответствия.

В результате выполнения второго этапа получается файл, содержащий оттранслированную микропрограмму. Этот файл используется затем как исходный для программирования ППЗУ после соответствующей обработки.

2.Система команд для MISC-процессора

Предлагается организовать систему команд по принципу дерева (рисунок). В основе дерева: базовая система команд, состоящая из 10-20 команд (простых – команд RISC-процессора, выполняющихся за один такт). Базовая система команд (БСК) вместе с префиксами для вхождения в более сложные команды образуют более сложную систему команд [3].

БСК + Префиксы

Более сложная СК + Префиксы

...

Самая сложная СК

Иерархия команд для MISC-процессора

Для каждого из наборов команд имеется своя микросхема ПЗУ микропрограмм или своя схема логического устройства.

Такое построение системы команд сложное для реализации, но весьма гибкое и задачу быстрого выполнения команд (без дополнительного обращения к памяти) решает:

1. Если в процессор с системной шины компьютера пришла простейшая команда, то она выполняется процессором выбором из базовой системы команд.

2. Если команда, поступившая в host-процессор, сложная (т.е. не может быть реализована командами из БСК), то происходит подключение того ПЗУ (на уровне которого имеется такая команда) через логическое устройство микрокоманд, где идет выбор сложной команды из сложной системы команд с помощью префиксов и БСК.

Существует и другой подход: один из наборов команд может быть сформирован в системной памяти компьютера (памяти, которая находится вне кристалла процессора) с помощью специального компилятора или ассемблера.

При разработке компилятора была выбрана система команд, состоящая из четырех групп [2].

Группа 1. Арифметико-логические команды:
1. SUB RX,RX - Вычитание
2. ADD RX,RX - Сложение
3. AND RX,RX - Логическое И
4. OR RX,RX – Логическое ИЛИ
5. NEC RX – Инвертирование НЕ
6. INC RX – Добавление 1
7. SHR RX – Сдвиг на 1 разряд с занесением 0

Группа 2. Передача данных:
8. SORA RX,(R0) – Пересылка во внутренние регистры (загрузка адреса)
9. SODR – Загрузка данных
10. SORR RX,RX –Перегрузка регистров

Группа 3. Управление программой:
11. JMP – Безусловный переход
12. JC – Условный переход
13. JSR – Переход к подпрограмме
14. RIS – Возврат из подпрограммы

Группа 4. Сравнение и анализ:
15. TST = RX,RX – Проверка на равенство
16. TST > RX,RX – Проверка на больше

Выбор такого подмножества команд из всего множества возможных обоснован единой задачей повышения быстродействия процессора.

С помощью трех основных операций (сложение, вычисление дополнительного кода и логической операции И/ИЛИ) могут быть выполнены любые другие арифметические и логические операции. Однако в большинстве случаев необходимо усложнять структуру АЛУ для повышения быстродействия МП за счет введения новых команд: вычитания, И-НЕ, исключающее ИЛИ и других.

Заключение

Для создания компилятора под MISC – архитектуру были выполнены такие исследования:

– рассмотрена ОП система возможных команд для К1804ВУ4, К1804ВМ1 и на ее основе обоснованно выбрана базовая система простейших команд для MISC – процессора;

– рассмотрены вопросы организации работы MISC – компилятора на этапах сканирования и генерирования кода.

Список литературы: 1.*Бережная М.А., Лобода В.Г., Цуканов В.Ю.* К вопросу проектирования структуры процессора // Радиоэлектроника и информатика. 1998. №2(3) .С.120-124. 2. *Комплект БИС К1804 в процессорах и контроллерах* / В.М. Мещеряков, И.Е. Лобов, С.С. Глебов и др. М.: Радио и связь, 1990. 256 с. 3. *Сизов К.А.* Микропроцессор будущего: RISC, CISC или MISC?// Библиотека информационных технологий: Сборник статей. Вып.1 / Под ред. Г.Р. Громова. М.: Наука, 1990. С.118-124. 4. *Ельчанинов Д.Б., Лобода В.Г., Цуканов В.Ю.* Модели архитектуры MISC-процессора// Радиоэлектроника и информатика. 1999. №1(06). С. 85-89. 5. *Язык программирования Си* / Бриан В. Керниган, Деннис М. Ритчи. М.: Финансы и статистика, 1992. 270 с.6. Р. Хантер. Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984. 232 с. 7. *Цуканов В.Ю.* Взаимная адаптация аппаратно-программных средств в процессоре спецназначения // Радиоэлектроника и информатика. 2000. №1. С.59-63.

Поступила в редколлегию 02.09.2000

Петросов Давид Арегович, студент ХТУРЭ. Научные интересы: алгоритмическое и программное обеспечение функционально-ориентированных процессоров. Адрес: Украина, 61137, Харьков, ул. Межлаука, 11/7, кв.67.

Цуканов Виталий Юрьевич, аспирант ХТУРЭ. Научные интересы: алгоритмическое и программное обеспечение функционально-ориентированных процессоров. Адрес: Украина, 61111, Харьков, ул. Познанская, 2, кв.67, тел. 10-42-63.

УДК 681.325:519.713

В.В. ХАНЬКО

СТРУКТУРНО-ЛОГИЧЕСКИЕ МОДЕЛИ ДЛЯ АНАЛИЗА И ТЕСТИРОВАНИЯ СЕТЕВЫХ СЕГМЕНТОВ

Предлагаются структурно-логические модели компонентов компьютерной сети, используемые для реализации системы анализа и тестирования исправного поведения и неисправностей, деструктивно влияющих на техническое состояние и возникновение коллизий в сетевых сегментах.

1. Введение

При разработке и развертывании корпоративных компьютерных сетей со сложной архитектурой, а также при изменении архитектуры уже существующих сетей возникает необходимость в предварительном тестировании правильности принятых технических решений. Такая проверка позволяет выявить и устранить ошибки в проекте ещё до его реализации, что впоследствии дает возможность значительно сократить материальные и временные затраты, необходимые для устранения тех же ошибок и просчетов в уже развернутой сети. Подход к решению задач создания и модернизации сетей особенно актуален для корпоративных сетей крупных предприятий, которые, как правило, неоднородны, имеют сложную архитектуру, в связи с чем их модернизация сопряжена с большими трудностями.

Задача предварительной проверки правильности принятых технических решений может быть решена путём моделирования спроектированной сети либо отдельных элементов уже существующей с помощью специализированных пакетов, например, NetMaker XA или COMNET Predictor [1]. Однако они весьма дороги, поэтому приобретение такого рода программ компанией, деятельность которой не связана с компьютерными технологиями, не является приоритетным. Кроме того, воспользоваться услугами компаний – системных интеграторов по ряду причин (например, отсутствие таковых в данном регионе) также бывает проблематично. В этих условиях администратору сети приходится обходиться своими силами, создавая инструментарий для диагностики своей сети самостоятельно, используя такие языки программирования, как Perl или C++, которые часто входят в дистрибутивный комплект большинства операционных систем. Несмотря на то, что данные языки являются мощным средством создания приложений, их применение для построения моделей *параллельных* процессов, происходящих в аппаратуре, связано с большими трудностями ввиду отсутствия в них