ОГЛАВЛЕНИЕ

Введение

В методических указаниях по лабораторным работам изложены практические вопросы методологии разработки программно-аппаратного обеспечения вычислительных устройств, применимых в информационных и автоматизированных системах различного назначения.

Методические указания предназначены для магистров по направлению 09.04.04 «Программная инженерия».

Методические указания содержат описание трех лабораторных работ по темам:

1. Разработка модели процессорного ядра программно-аппаратного комплекса.
2. Разработка модели аппаратного ускорителя вычислений для программно-аппаратного комплекса.
3. Разработка ассемблера для модели процессора.

Описание каждой лабораторной работы содержит цель работы, теоретическое введение, описание проекта, порядок выполнения работы и оформления отчета, а также контрольные вопросы.

Лабораторная работа № 1. Разработка модели процессорного ядра программно-аппаратного комплекса

Цель работы: освоение маршрута проектирования процессорных ядер на уровне системного моделирования путем разработки модели процессорного ядра на языке программирования высокого уровня.

Введение

При проектировании процессоров важную роль играет предварительное моделирование. При использовании программы, имитирующей работу будущего процессора, становится возможным проверить результаты выполнения на этом процессоре программ и убедиться, что для этого имеются достаточные ресурсы, набор поддерживаемых команд, а алгоритм выполняется корректно и за приемлемое число шагов. Предварительное моделирование, выполняемое с помощью широко распространенной компьютерной техники, может быть организовано быстрее и не требует безвозвратных финансовых потерь по сравнению с проектированием процессора в виде интегральной микросхемы.

Такое моделирование процессора называется *эмуляцией*.

«Имитация функционирования одного устройства посредством другого устройства или устройств вычислительной машины, при которой имитирующее устройство воспринимает те же данные, выполняет ту же программу и достигает того же результата, что и имитируемое» [из п. 53 табл. 1 ГОСТ 15971-90].

«Эмуляция – комплекс программных, аппаратных средств или их сочетание, предназначенное для копирования функций одной вычислительной системы на другой, отличной от первой, вычислительной системе таким образом, чтобы эмулированное поведение как можно ближе соответствовало поведению оригинальной системы. Примечание - Целью эмуляции является максимально точное воспроизведение поведения в отличие от разных форм моделирования, в которых имитируется поведение некоторой абстрактной модели. Например, моделирование физического процесса или явления не является эмуляцией» [из п. 3.7 ГОСТ Р 57721-2017]

Эмуляция может выполняться на нескольких уровнях, различающихся детализацией моделирования. При необходимости определить поведение процессора при выполнении алгоритмов может быть выполнено моделирование на уровне программной модели. Пример программной модели процессора может выглядеть, как показано на рис. 1.1.

Рис. 1.1. Программная модель процессора

В модели показаны базовые компоненты процессора. Регистр PC (Program Counter) называется также «счетчик команд». Этот регистр содержит адрес памяти, из которого в данный момент исполняется команда. Память команд и данных может быть физически разделена (т.е. процессор не может выполнять команды из памяти данных). Это решение соответствует *гарвардской архитектуре* процессора. Если же команды и данные хранятся в одной и той же памяти, говорят об *архитектуре фон Неймана*.

Спецификой памяти является ограниченное количество чисел, которые можно читать из нее в каждый момент времени (обычно это одно число, однако существует и многопортовая память). Поэтому для выполнения вычислений процессор обычно загружает данные из памяти во внутренние узлы для хранения чисел – регистры данных. Существует множество вариантов организации регистров данных – их количества, разрядности, допустимых операций между этими регистрами и т.д. (рис.1.2, 1.3, 1.4).

Моделирование процессора производится путем программной имитации действий, выполняемых аппаратными компонентами процессора в ходе его

работы. При этом игнорируются технические детали работы таких схем – например, для имитации чтения данных из памяти достаточно прочитать значение из массива, который имитирует память, подключенную к процессору.

Рис. 1.2. Регистровая модель 16-разрядного процессора i8086.

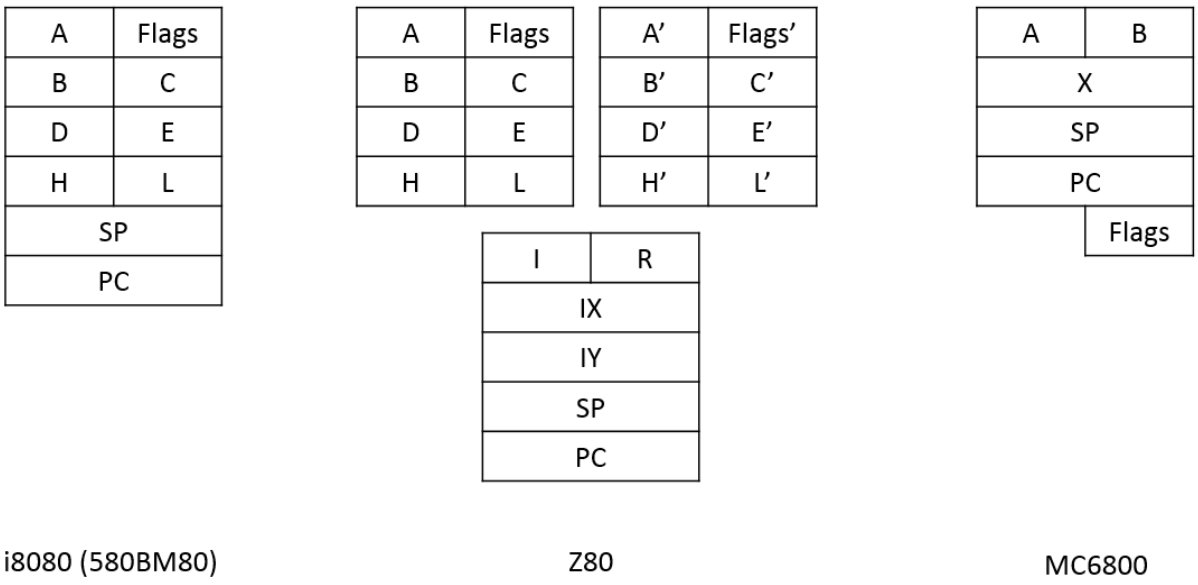

Рис. 1.3. Регистровые модели процессоров i8080, Z80 и MC6800

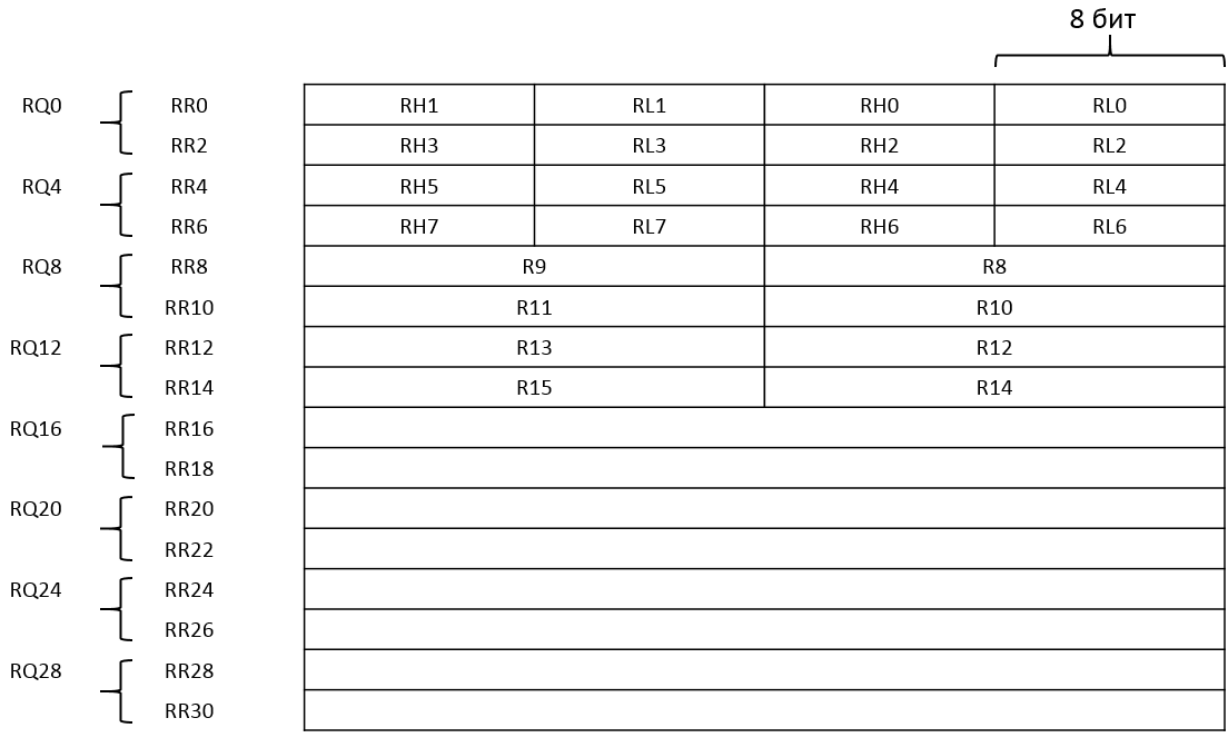

Рис. 1.4. Регистры общего назначения процессора Z80000 (1986 г.)

Состояние управляющих сигналов памяти, времена распространения сигналов и прочие существенные для микроэлектроники параметры не учитываются. Подразумевается, что при правильном подключении памяти к процессору операции чтения и записи будут производиться корректно.

В простейшей модели типичный процессор циклически выполняет следующие действия:

1. Чтение команды из ячейки памяти, адрес которой содержится в регистре PC.

2. Выполнение действий, кодируемых этой командой, включая обязательное вычисление нового значения для регистра PC.

В простейшем случае фрагмент программы, имитирующей работу процессора, будет выглядеть следующим образом:

```
int cmd;
int pc;
int cmem[1024];
int dmem[1024];

while(1)
{
  cmd = cmem[pc];
```

```
        // здесь впоследствии необходимо описать
        // модель выполнения принятой команды
        pc = pc + 1;
    }
```

В этом примере имеются упрощения:

– не предусматривается выход из цикла, что в целом соответствует нормальной работе процессора, но неприемлемо для программы, которая в таком виде будет работать бесконечно;

– использован фиксированный размер массива, имитирующего память команд процессора и фиксированные целочисленные типы для регистра PC и ячеек памяти;

– отсутствует имитация исполнения команд.

Для имитации выполнения команд необходимо выполнить планирование системы команд. Команда, в примере выше записанная в переменную cmd, обычно содержит отдельные разряды, кодирующие выполнение тех или иных действий. Для показанного на рис. 1 примера можно представить простой набор команд, задаваемые разрядами числа, как показано на рис. 2. Сгруппированные двоичные разряды называют также полями, т.е. разряды с 3 по 0 можно называть «поле второго операнда» (рис. 1.5).

Разряды двоичного представления команды	31-28	27-12	11-8	7-4	3-0
Название	cmdtype	literal	dest	Op1	Op2
Описание	Тип команды	Непосредственное значение («литерал»)	Номер регистра для записи результата	Номер регистра – первого операнда	Номер регистра – второго операнда

Рис. 1.5. Формат команды для примера процессора

Поскольку цифровая схема не ограничена использованием форматов чисел, принятых в программировании, двоичное представление команды может использоваться внутренними схемами процессора как набор независимых сигналов. Например, если в процессоре имеется 16 регистров, то номер регистра может быть задан в 4-разрядном двоичном числе. Для кодирования команды следует задать номера ее операндов, регистр-получатель результата, а

также тип команды. В примере на рис. 1.2 для типа команды отведено также 4 разряда, поэтому возможно кодировать 16 вариантов команды. Например, если принять, что сложение регистров имеет код команды 1, то команда для сложения регистров 3 и 4, помещающая результат в регистр 7, будет выглядеть следующим образом (рис. 1.6):

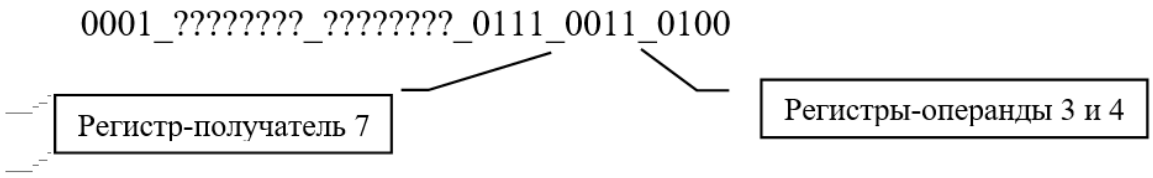

Рис. 1.6. Результат выполнения команды

Знаками вопроса показаны разряды, которые не используются в команде и могут принимать любые значения.

Поскольку для примера выбраны 4- и 16-разрядные поля команды, удобно задавать код команды в шестнадцатеричном формате. Чтобы поместить показанную команду в память, достаточно выполнить присваивание:

cmem[0] = 0x10000734

Тогда при pc = 0 процессор прочитает из памяти эту команду, поля которой будут кодировать требуемые действия. Удобно использовать вспомогательные переменные, в которые можно записать значения отдельных полей команды. Чтобы выделить из команды отдельные разряды, можно использовать операцию «поразрядное И». В результате поразрядного «И» каждый бит результата становится равным 1, только если соответствующие биты обоих операндов равны 1. Например, если выполнить операцию cmd & 1, то результатом будет число 0 или 1, в зависимости от того, был ли равен 1 самый младший разряд двоичного представления переменной cmd. Чтобы выделить разряды с 3 по 0 и записать результат во вспомогательную переменную op2, необходимо выполнить операцию

Op2 = cmd & 15

Число 15 имеет двоичное представление 1111, т.е. содержит единицы в четырех младших разрядах. Для того, чтобы выделять поля, находящиеся в других разрядах, следует сначала выполнить сдвиг двоичного представления, чтобы требуемый фрагмент команды оказался в младших разрядах. Например, для выделения поля первого операнда необходимо выполнить операцию

Op1 = (cmd >> 4) & 15

Сначала оператор сдвига перемещает разряды 7 – 4 в позиции 3 – 0, затем

оператор & выделит 4 младших разряда.

Аналогично следует организовать выделение остальных полей команды.

Далее следует организовать моделирование выполнения принятой команды. Поскольку номера регистров, участвующих в команде, представляют собой индексы массива регистров, имитацию сложения можно выполнить следующим образом:

```
Switch (cmdtype)
{
   Case 1 : reg[dest] = reg[op1] + reg[op2]; break;

   Default : break;
}
```

Подобным же образом можно описать другие команды, выполняющие арифметическую или логическую операцию над входными операндами op1 и op2. К числу операций, реализуемых в процессорах, обычно относятся:

– вычитание;

– поразрядное И (and);

– поразрядное ИЛИ (or);

– поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (xor);

– инверсия одного из операндов (not);

– сдвиги влево и вправо.

Операция умножения, очевидная для арифметики, должна использоваться в процессорах с осторожностью. Схема умножения двоичных чисел занимает существенно больше ресурсов по сравнению с другими операциями, поэтому ряд процессорных ядер не имеет команды умножения, или реализует ее последовательными шагами по схеме «сложение со сдвигом» (аналогично умножению в столбик, записанному для двоичных чисел).

Операция целочисленного деления не имеет общепринятой схемы на базе логических элементов (могут существовать реализации деления на базе таблиц, однако они существенно ограничены по разрядности операндов). Как правило, используется команда «шаг деления», вычисляющая один бит результата по алгоритму, аналогичному делению в столбик.

Кроме представленного формата команды, для моделирования процессора следует реализовать как минимум два дополнительных варианта.

Первый из этих вариантов – так называемая непосредственная адресация

(Indirect). Если представить программу, выполняющую сложение чисел 2 и 3, то из рассмотренного ранее формата команд видно, что попытка указать эти числа в полях op1, op2 приведут к использованию регистров 2 и 3, но не чисел 2 и 3. Поэтому необходимо добавить поле команды, содержащее не номер регистра, а непосредственное (Indirect), также обозначаемое как literal, («буквальный») представление числа. Эта команда может быть смоделирована оператором вида:

```
Case 2 : reg[dest] = literal; break;
```

Недостатком такого формата является ограниченный размер поля literal. В приведенном примере он имеет размер 16 бит, что не позволяет загружать одной командой 32-разрядные регистры. Кроме того, поле literal не требуется для команд, использующих в качестве операндов регистры процессора. Однако в учебных целях, для сохранения единообразия представления команд, можно проиллюстрировать работу процессора на упрощенном примере.

Второй необходимый вариант команд – команды переходов. Без этих команд процессор будет иметь возможность выполнять только линейную последовательность операторов, не поддерживая ни условные операторы, ни циклы, ни вызовы подпрограмм. Для реализации практических примеров необходимо, чтобы кроме перехода к следующей команде, была доступна возможность перехода к заданному адресу.

Примеры имитации переходов приведены ниже:

```
Case 3 : pc = literal; break;
Case 4 : if (reg[op1] == reg[op2]) pc = literal; else pc = pc + 1; break;
```

В первом случае (тип команды указан равным 3) счетчик команд загружается значением, переданным в поле литерала. Это вариант безусловного перехода (jump). Во втором случае в pc либо загружается литерал, либо выполнение программы продолжается со следующего адреса. Это вариант условного перехода (т.е. перехода, выполняющегося при наступлении определенного условия). В примере показано условие, состоящее в равенстве значений регистров процессора с номерами op1 и op2.

Как правило, в процессорах используются разные варианты условий перехода. Обычно это переходы, выполняемые в зависимости от состояния однобитных переменных, устанавливаемых по результатам предварительно

выполненных операций. Такие переменные называются флагами и входят в состав программной модели процессора (в рассматриваемом примере флаги не используются). Наиболее распространенные флаги:

– флаг нуля (Zero Flag), равен 1, если последний из вычисленных процессором результатов был равен 0;

– флаг переноса (Carry Flag), равен 1, если последний из вычисленных процессором результатов потребовал перенос в несуществующий старший разряд или заем из несуществующего старшего разряда.

При использовании флагов команды условного перехода выглядят следующим образом:

```
If (cf == 1) pc = literal; else pc = pc + 1;
If (cf == 0) pc = literal; else pc = pc + 1;
```

Рассмотренная модель имеет целый ряд упрощений:

– используются только целочисленные форматы данных;

– не учитывается количество тактов, затрачиваемое на выполнение команд;

– не моделируется работа системной шины и внешних устройств.

1. Описание проекта

Проект для лабораторной работы представляет собой программу в виде консольного или графического приложения для PC на языке программирования высокого уровня. Допустимо использование языков C/C++/C#/Python на базе соответствующих сред программирования.

Проект заключается в описании модели регистров и памяти процессора, а также проверки модели путем составления тестовой программы с предсказуемым результатом исполнения. После записи команд тестовой программы в массив, имитирующий память процессора, запуск модели должен привести к пошаговому моделированию состояний процессора при выполнении каждой из команд. Корректность модели и программы определяется достижением заранее определенного результата вычислений,

2. Порядок выполнения работы

2.1. Выбрать и согласовать с преподавателем регистровую модель процессора и набор команд.

2.2. Создать программный проект в выбранной среде программирования. Допустимо использование широкого спектра языков и сред программирования, включая MS Visual Studio, Eclipse, Qt Creator, Java Builder, варианты IDE на

основе языка Python и т.д. Язык программирования должен допускать объявление переменных и массивов, а также выполнение основных операторов процедурных языков. Среда программирования должна обеспечивать, по крайней мере, создание консольных приложений с возможность вывода текстовых сообщений.

2.3. Разработать описание последовательности действий процессора при выполнении основных команд. Описание рекомендуется создавать на базе оператора case/switch (в зависимости от используемого языка программирования).

2.4. Разработать тестовую программу по заданию преподавателя. Тестовая программа создается в режиме прямого программирования полей каждой команды, которые записываются в ячейки массива cmem[].

2.5. Выполнить моделирование работы процессора при выполнении тестовой программы. Для этого необходимо организовать циклическое выполнение последовательности «чтение команды из памяти – имитация выполнения команды оператором case» с выводом на экран состояния регистров и фрагмента памяти данных процессора.

3. Порядок оформления отчета

3.1 Отчет должен содержать:

– описание регистровой модели процессора с перечислением регистров и их назначения;

– описание использованных форматов команд процессора;

– описание архитектуры программной модели (с перечислением основных переменных, функций и описанием выполняемых ими действий);

– протокол испытаний модели процессора;

– вывод по проделанной работе с перечислением достигнутых результатов.

Исходный текст программы оформляется в виде приложения к отчету.

Контрольные вопросы:

1. Что такое эмуляция? С какой целью можно использовать эмуляцию процессора перед началом его проектирования?

2. В чем состоит различие между гарвардской архитектурой и архитектурой фон Неймана?

3. Для чего необходимы регистры процессора?

4. Что такое счетчик команд?

5. Сколько разрядов необходимо, чтобы указать номер используемого регистра, если в процессоре 8 регистров? 16 регистров? 32 регистра?

6. Что такое непосредственный операнд (литерал)?

7. Какие варианты условного перехода можно предложить для процессора?

8. Если размер команды меньше или равен разрядности регистров, какие способы загрузки непосредственных значений можно предложить?

9. Какую последовательность команд нужно выполнить, чтобы перейти к следующей итерации цикла со счетчиком?

Лабораторная работа № 2. Разработка модели аппаратного ускорителя вычислений для программно-аппаратного комплекса

Цель работы: разработка модели аппаратного ускорителя вычислений на языке программирования высокого уровня.

Введение

В настоящее время к программно-аппаратным комплексам могут предъявляться различающиеся требования по характеристикам, в зависимости от их назначения и области применения. Поэтому на практике невозможно разработать универсальную аппаратную платформу, одинаково хорошо подходящую для множества отличающихся областей применения. Например, для настольных компьютеров, ноутбуков, смартфонов и роутеров требуются процессоры, отличающиеся производительностью, энергопотреблением и возможностью выполнения операций определенных типов.

Кроме того, даже в составе одного комплекса часто выполняются задачи, относящиеся к разным классам по составу выполняемых операций. Например, в персональном компьютере (десктопе, ноутбуке) требуются как вычисления универсального характера с широким набором поддерживаемых операций, так и специфические операции над объектами в трехмерном пространстве для построения трехмерных изображений на экране дисплея. Целесообразно разделить такие задачи, реализовав в составе комплекса несколько вычислительных узлов. Можно убедиться, что в современных компьютерах используются как процессоры общего назначения (CPU), так и графические процессоры (GPU), ориентированные на выполнение операций, характерных для построения трехмерных изображений на двумерном дисплее.

В отличие от *гомогенной* («однородной») архитектуры, подразумевающей использование множества одинаковых вычислительных устройств, такие комплексы имеют *гетерогенную* («неоднородную») архитектуру.

Универсальной платформой для реализации аппаратных ускорителей являются микросхемы программируемой логики с архитектурой FPGA. Они представляют собой матрицу программируемых логических ячеек с

конфигурируемыми соединениями. Кроме базовых ячеек, на микросхеме FPGA размещены также блоки статической памяти, аппаратные компоненты «умножение с накоплением» и высокоскоростные последовательные приемопередатчики. Поэтому с применением FPGA можно разрабатывать ускорители, предназначенные для следующих задач:

1. На базе компонентов «умножение с накоплением» – цифровые фильтры, блоки для операций с матрицами, ускорители преобразования Фурье, нейросети.

2. На базе логических ячеек – вспомогательные контроллеры, устройства аппаратной защиты, преобразователи интерфейсов, устройства сбора данных.

Высокоскоростные последовательные приемопередатчики являются единственным способом для передачи сигналов со скоростями 3, 6 и более Гбит/с (до 58 Гбит/с на базе 7 нм FPGA). Такие компоненты используются в основном в магистральном сетевом оборудовании, разработка которого требует специальных технологий проектирования печатных плат и приборов для отладки высокоскоростных аналоговых цепей.

Эффективно продемонстрировать взаимодействие процессора и аппаратного ускорителя можно на примере ускорения цифровой обработки сигналов.

Важной задачей цифровой обработки сигналов является задача фильтрации длинной последовательности чисел. Устройство, реализующее эту задачу, называется цифровой фильтр. Линейный ЦФ, по определению, есть дискретная система (физическое устройство или программа для компьютера), которая преобразует последовательность $\{x(i)\}$ числовых отсчетов входного сигнала в модифицированную последовательность $\{y(i)\}$ отсчетов выходного сигнала:

$$(x_0, x_1, x_2 \ldots) \Rightarrow (y_0, y_1, y_2 \ldots). \quad (1)$$

Цифровые фильтры имеют ряд преимуществ, основные из них – стабильность характеристик и надежность в работе, недостижимые в аналоговых фильтрах. Фильтруемый сигнал представляет собой поток непрерывных данных или чисел. Наиболее общий вид цифрового фильтра первого порядка определяется разностным уравнением:

$$y(i) = \sum_{k=0}^{K} b_k x(i-k) - a_1 y(i-1). \quad (2)$$

Текущий выход фильтра определяется линейной комбинацией K предшествующих значений входа, текущим значением входа и одним текущим значением выхода. Такой фильтр называется фильтром первого порядка

потому, для получения нового значения выхода из линейной комбинации значений входов вычитают только одно предыдущее значение выхода (умножение на постоянный множитель). Вообще порядок фильтра определяет число входящих в разностное уравнение значений предшествующих выходов.

Важным понятием в цифровой фильтрации является импульсная характеристика фильтра. Импульсной характеристикой системы называется её реакция на единичный импульс при нулевых начальных условиях.

По импульсной характеристике фильтры делятся на два основных класса:

– КИХ фильтры, фильтры с конечной импульсной характеристикой;

– БИХ фильтры, фильтры с бесконечной импульсной характеристикой, называемые еще авторегрессионными или рекурсивными фильтрами.

В случае КИХ фильтра импульсная характеристика фильтра тождественна ее весовым коэффициентам. Для БИХ фильтров подобного тождества не наблюдается.

На рис. 2.1 показан пример амплитудно-частотной характеристики КИХ фильтра, построенного в специализированной утилите FIR Compiler, входящей в САПР Vivado. Утилита предназначена для создания аппаратных фильтров, поэтому результаты моделирования могут быть впоследствии проверены на базе FPGA.

Коэффициенты этого фильтра имеют следующие значения:

{6, 0, -4, -3, 5, 6, -6, -13, 7, 44, 64, 44, 7, -13, -6, 6, 5, -3, -4, 0, 6}

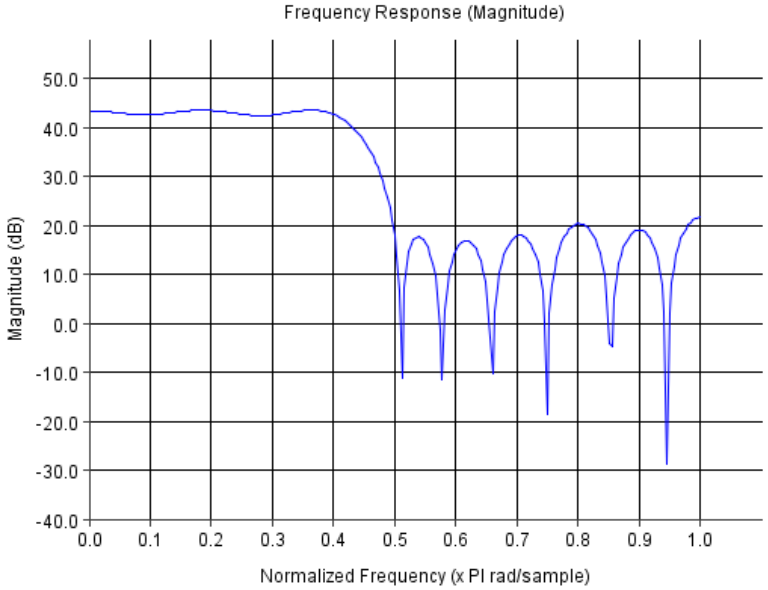

Рис. 2.1. Пример АЧХ КИХ фильтра

По горизонтальной оси на рис. 2.1 отложена частота в радианах на цифровой отсчет. Значение 1.0 соответствует предельному для цифрового фильтра случаю – частоте равной половине частоты дискретизации, т.о. АЧХ

фильтра приводится в пределах выполнения условия Котельникова-Найквиста-Шеннона.

Операция фильтрации КИХ фильтром сводится в операции умножения с накоплением:

$$y(m) = \sum_{i=0}^{m} x_i h_{m-i}\,,\ m = 0,1,2... \qquad (3)$$

Цифровой фильтр реализуется либо как специализированное вычислительное устройство на основе так называемой жесткой логики из элементов типа сумматор, сдвиговый регистр и умножитель, либо как программа, выполняемая на процессорном устройстве.

Как видно из рис. 2.2, КИХ фильтр представляет собой многоразрядный сдвиговый регистр, выходы которого хранят число со входа в различные моменты времени. Так число x_0 соответствует числу на выходе АЦП в момент времени t_0, x_{-1} – число на предшествующем такте преобразования или в момент времени t_{-1} и т.д. Числа, соответствующие различным моментам времени преобразования умножаются каждое на свой коэффициент K_i и поступают на сумматор. В результате на выходе появляется новая числовая последовательность.

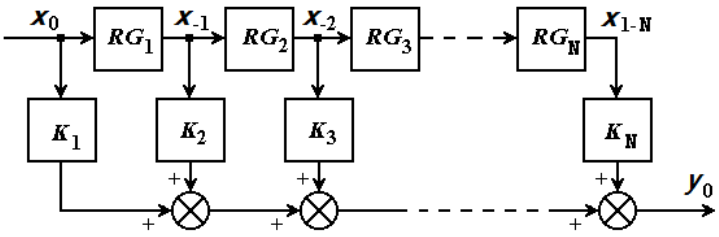

Рис. 2.2. КИХ фильтр.

Процесс фильтрации можно дополнительно пояснить графически, рис. 2.3.

На рис. 2.3. показаны коэффициенты фильтра *K(i)* и фильтруемый сигнал *X(i*). Точки с одинаковым *i* индексом перемножаются, после чего произведения складываются. В результате получается новая точка модифицированной последовательности *y*. Для получения следующей точки модифицированной последовательности сигнал сдвигается на единицу и операция повторяется. Получается, что на сигнал накладывают скользящее окно фильтра (рис. 2.4).

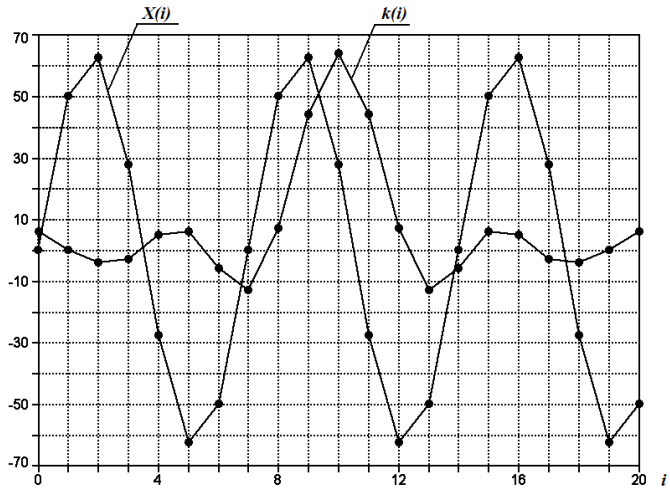

Рис. 2.3. КИХ фильтр.

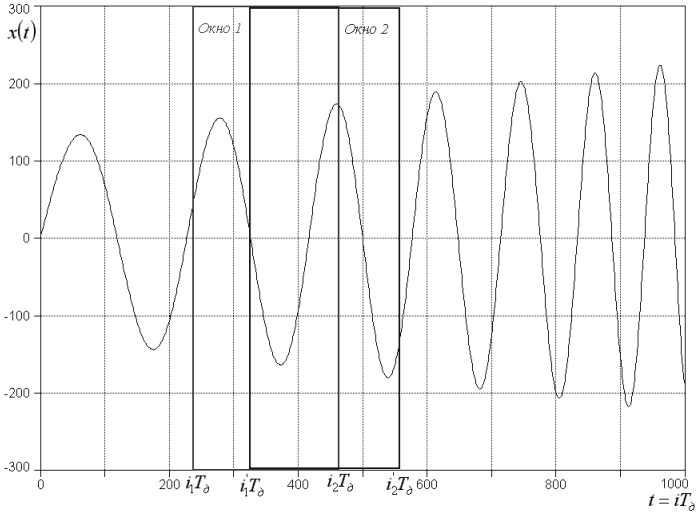

Рис. 2.4. Входная последовательность и наложенные на нее окна

У КИХ фильтров есть важное свойство, – если количество коэффициентов нечетное и коэффициенты обладают симметрией, то фазовая характеристика становится линейной. Поэтому необходимо отметить, что весовые коэффициенты не обязательно должны перемножаться со входными отсчетами, расположенными левее их. Т.е., выходной отсчет может получаться от симметрично расположенных относительно него весовых коэффициентов, и тогда весовые коэффициенты будут представлены положительными и отрицательными индексами. Это важно для получения нулевых фазовых искажений, т.к. если делать свертку, как описано в примере, фазовая характеристика будет линейно возрастать.

При моделировании фильтра необходимо иметь в виду, что при умножении двоичных чисел разрядность результата может превысить 32. В общем, для представления результата умножения требуется разрядность,

равная сумме разрядностей операндов. Кроме того, если последовательно складывать числа, требуемая разрядность также может возрасти. Например, при сложении 2 чисел может потребоваться дополнительный бит для представления результата, при сложении 4 чисел – 2 бита, и т.д. Однако разрядность, требуемая для конкретного набора коэффициентов, зависит от фактического заполнения разрядной сетки этих коэффициентов.

1. Описание проекта

Проект для лабораторной работы представляет собой программу в виде консольного или графического приложения для PC на языке программирования высокого уровня. Допустимо использование языков C/C++/C#/Python на базе соответствующих сред программирования.

Проект заключается в разработке программы, имитирующей работу аппаратного ускорителя в составе программно-аппаратного комплекса. Имитация производится с соблюдением основных ограничений цифровой схемотехники – в частности, в пределах одного цикла работы допускается только одно изменение каждой переменной, моделирующей аппаратные ресурсы ускорителя.

2. Порядок выполнения работы

2.1. Определить и согласовать с преподавателем тип аппаратного ускорителя и сформулировать основную задачу, решаемую ускорителем в составе программно-аппаратного комплекса.

2.2. Создать программный проект в выбранной среде программирования. Допустимо использование широкого спектра языков и сред программирования, включая MS Visual Studio, Eclipse, Qt Creator, Java Builder, варианты IDE на основе языка Python и т.д. Язык программирования должен допускать объявление переменных и массивов, а также выполнение основных операторов процедурных языков. Среда программирования должна обеспечивать, по крайней мере, создание консольных приложений с возможность вывода текстовых сообщений.

2.3. Разработать программу, моделирующую работу ускорителя путем имитации вычислений, характерных для такого устройства. В программе необходимо предусмотреть подсчет количества операций, выполняемых за один цикл работы (т.е. период тактовой частоты моделируемого устройства).

2.4. Разработать модель входных воздействий (сигналов, начального состояния памяти ускорителя и т.д.) и провести моделирование работы ускорителя.

3. Порядок оформления отчета

3.1 Отчет должен содержать:

– описание ускорителя, его назначения в составе программно-аппаратного комплекса и обоснование нецелесообразности выполнения задач ускорителя на базе процессора;

– протокол испытаний модели ускорителя;

– вывод по проделанной работе с перечислением достигнутых результатов.

Исходный текст программы оформляется в виде приложения к отчету.

Контрольные вопросы:

1. Что такое аппаратный ускоритель?

2. Почему GPU используют для построения трехмерной графики вместо CPU?

3. Что общего у цифровых фильтров и нейронов?

4. Существует ли универсальный цифровой фильтр с единственным набором коэффициентов? Каким образом следует организовать обновление коэффициентов фильтра?

5. Как следует организовать подачу входного сигнала на вход цифрового фильтра – со стороны процессора или независимого входа? Можно ли использовать оба способа?

6. Сколько операций CPU потребуется, чтобы реализовать фильтр, описанный во введении данной работы? Какой должна быть частота такого CPU, если частота отсчетов входного сигнала равна 100 МГц?

7. Сколько разрядов потребуется для представления результата цифрового фильтра, если входные отсчеты представлены 16 битами, коэффициенты фильтра – 32 битами, а фильтр может иметь до 32 коэффициентов?

8. Почему теорема Котельникова (в зарубежной литературе теорема Найквиста – Шеннона) утверждает, что для представления синусоидального сигнала требуется не менее 2 точек на период?

9. Будет ли корректной работа цифрового фильтра, реализуемого на базе CPU, если CPU в процессе вычисления выхода фильтра выполнит обработку прерывания в течении 1000 тактов, а входные отсчеты поступают каждые 100 тактов?

Лабораторная работа № 3. Разработка ассемблера для модели процессора

Цель работы: разработка ассемблера для модели процессора на языке программирования высокого уровня.

Введение

Инструментальное программное обеспечение является важнейшей составной частью проекта процессорной системы. Ручное формирование машинных кодов крайне непродуктивно и его невозможно использовать даже для отладки сколько-нибудь объемных программ. Поэтому для разработки процессора необходимо иметь хотя бы базовые инструменты для создания кода, пусть даже и с ограниченными возможностями.

Согласно определению в ГОСТ 19781-90, «кросс-система программирования – система программирования, программные компоненты которой порождают программы на машинном языке, отличном от того, в среде которого они работают».

Под кросс-компиляцией (cross-compilation) понимается процесс перевода программы в машинный код, который должен исполняться на процессоре с архитектурой, отличной от той, на которой запускается сам компилятор. Иными словами, речь идет о том, чтобы на ПК с процессором x86 была запущена программа, преобразующая некий исходный текст в машинные коды для процессора с другой архитектурой. Созданный машинный код может быть использован как для загрузки в макет процессора в ПЛИС, так и для проведения с его помощью моделирования.

Хорошо известным трудом в области разработки компиляторов является т.н. «Книга дракона» («Dragon book») [1]. Ее настоящее название «Компиляторы: принципы, технологии и инструментарий», а «книга дракона» вошла в обиход из-за оформления обложки, на которой процесс разработки компилятора представлен в виде борьбы рыцаря-программиста с драконом-компилятором. Книга при ее большом объеме описывает различные аспекты разработки компиляторов, не все из которых обязательны для построения практического продукта. В ней дается общий маршрут работы компилятора, который представлен на рис. 3.1.

Рассмотрим вкратце шаги, которые могли бы помочь разработать простой компилятор, переводящий исходный текст программы в последовательность машинных кодов. Исходя из рис. 3.1, первым шагом компиляции является лексический анализ, переводящий исходный текст программы в последовательность *токенов*. Под токенами понимаются пары «имя –

значение», которые строятся для каждого найденного элемента программы. Например, ассемблерная команда mov r0, r1 будет разобрана на элементы «mov» «r0» «r1».

Рис. 3.1. Стадии компиляции

Синтаксический анализ имеет целью построить *промежуточное представление* программы. Существуют различные способы представление порядка операций программы, например, в виде *синтаксического дерева*. Однако такое представление может быть несколько избыточно для простого ассемблера.

На стадии семантического анализа проверяется смысловая корректность построенного промежуточного представления. Например, строка x = y + z является синтаксически корректной, однако если x, y – это целые числа, а z – строка, то в представленном виде такой оператор будет ошибочным. Необходимо или констатировать ошибку, или, если разработанный язык это допускает, выполнить преобразование строковой переменной в число перед сложением.

В результате проверки корректности промежуточного представления образуется машинно-независимый код. С его получением завершается т.н. *компиляция переднего плана* (front-end). Данное представление все еще не является последовательностью машинным кодов, за получение которой отвечает *компилятор заднего плана* (back-end). Его центральным элементом является собственно генератор кода, однако как с промежуточным

представлением, так и с итоговым кодом возможно выполнение оптимизирующих преобразований.

Сложность разработки компилятора, особенно на уровне синтаксического анализа, существенно зависит от класса реализуемого языка. Классификация грамматик была предложена Хомским и включает четыре типа: фразовую, контекстно-зависимую, контекстно-свободную и регулярную грамматики.

Представим программу на ассемблере в виде потока *лексем* («элементов языка»). Например, команда *nop* не имеет аргументов и может быть использована в исходном виде, а для команды mov требуется два операнда. Если рассматривать строки вида

<command> <op1>, <op2>

можно видеть, что в строке за командой следуют два операнда, разделенные запятой. Тогда алгоритм анализа строки может содержать следующие шаги:

1. Выделить команду (ограниченную пробелом) и сравнить ее со списком допустимых команд.

2. Выделить первый операнд (ограниченный пробелом или запятой).

3. Выделить второй операнд.

Поскольку данные действия происходят последовательно, а от найденной команды зависит список требуемых операндов (например, если найдена команда nop, поиск операндов необходимо остановить, а если они найдены, это следует считать ошибкой), можно реализовать синтаксический разбор с помощью конечного автомата.

Фрагмент диаграммы для простого синтаксического анализатора показан на рис. 3.2.

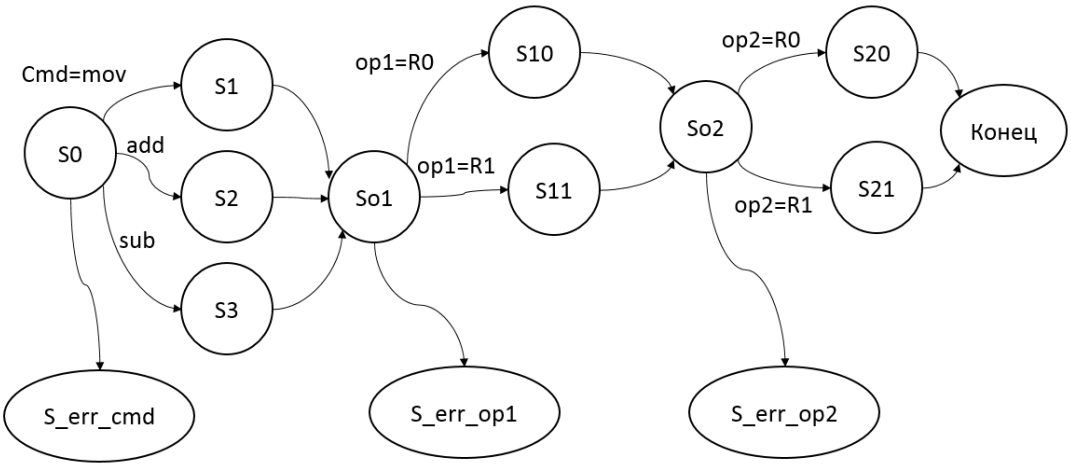

Рис. 3.2. Диаграмма переходов конечного автомата для простого синтактического анализатора

Пример реализации подобного синтаксического анализа показан в листинге. В примере разбираются строки вида «cmd op1, op2», причем команда может быть *mov* или *add*, а в качестве операндов принимаются только r0 и r1. Можно видеть, что в процессе анализа определяются команда и оба операнда, что позволяет получить значение для машинного кода проанализированной операции, просто подставив номера команды и операндов в соответствующие поля машинного кода.

```
void syntax(char * str)
{
	int op1, op2;
	char * token = strtok(str, " ");

	if (strcmp(token, "mov") == 0)
	{
		token = strtok(NULL, " ,");
		op1 = -1;
		if (strcmp(token, "r0") == 0)
		{
			op1 = 0;
		}
		if (strcmp(token, "r1") == 0)
		{
			op1 = 1;
		}

		token = strtok(NULL, " ");
		op2 = -1;
		if (strcmp(token, "r0") == 0)
		{
			op2 = 0;
		}
		if (strcmp(token, "r1") == 0)
		{
			op2 = 1;
		}

		printf("mov: op1 = %d op2 = %d \n\r", op1, op2);
	}

	if (strcmp(token, "add") == 0)
	{
		token = strtok(NULL, " ,");
		op1 = -1;
		if (strcmp(token, "r0") == 0)
		{
			op1 = 0;
		}
		if (strcmp(token, "r1") == 0)
```

```
            {
                  op1 = 1;
            }

            token = strtok(NULL, " ");
            op2 = -1;
            if (strcmp(token, "r0") == 0)
            {
                  op2 = 0;
            }
            if (strcmp(token, "r1") == 0)
            {
                  op2 = 1;
            }

            printf("add: op1 = %d op2 = %d \n\r", op1, op2);
      }

   }

int main(void) {
    char s1[] = "mov r0, r1";
    char s2[] = "add r1, r0";

    syntax(s1);
    syntax(s2);
    return 0;
   }
   // Результаты работы:
   mov: op1 = 0 op2 = 1

   add: op1 = 1 op2 = 0
```

Листинг. Простейший синтаксический анализ

Явно видимым недостатком представленного подхода является большая трудоемкость описания всех возможных комбинаций команд и их операндов. Поэтому при проектировании процессора следует придерживаться *ортогональной* архитектуры команд. Это свойство подразумевает возможность использования единообразного формата команд, что упрощает кодирование и позволяет в частности, после определения индексов регистров формировать машинный код без дополнительных проверок на допустимость сочетания именно этой пары операндов.

Организация переходов между командами в простейшем случае может быть реализована непосредственно в процессе ассемблирования. Для этого необходимо отметить команду, на которую требуется организовать переход.

Например, ключевое слово label не генерирует команду процессора, но отмечает текущую команду для последующей ссылки на нее. После ключевого слова должен следовать номер или обозначение метки.

Например,

```
Label 1
```

Отмечаемые метки записываются в специальную структуру ассемблера – таблицу меток. Впоследствии при упоминании метки ее адрес будет уже известен из таблицы.

Однако таким образом невозможно описать переход вперед, поскольку такая метка еще не объявлена. Для этого места в программе, упоминающие переход к еще не объявленной метке, также запоминаются в таблице в качестве *неразрешенных* (unresolved, более точное понятие – «пока не решенных», «пока не выясненных»). При объявлении метки необходимо проверить, нет ли в этой таблице отмеченных ранее переходов на такую метку и вписать в соответствующие команды программы выясненный адрес.

1. Описание проекта

Проект для лабораторной работы представляет собой программу в виде консольного или графического приложения для PC на языке программирования высокого уровня. Допустимо использование языков C/C++/C#/Python на базе соответствующих сред программирования.

Проект заключается в разработке программы, преобразующей исходный текст программы на языке ассемблера в машинные коды процессора, реализованного в виде модели.

2. Порядок выполнения работы

2.1. Определить и согласовать с преподавателем программную модель процессора. Рекомендуется использовать модель, разработанную в ходе выполнения лабораторной работы №1.

2.2. Создать программный проект в выбранной среде программирования. Допустимо использование широкого спектра языков и сред программирования, включая MS Visual Studio, Eclipse, Qt Creator, Java Builder, варианты IDE на основе языка Python и т.д. Язык программирования должен допускать объявление переменных и массивов, а также выполнение основных операторов процедурных языков. Среда программирования должна обеспечивать, по крайней мере, создание консольных приложений с возможность вывода текстовых сообщений.

2.3. Разработать программу ассемблера, преобразующую входной текст в машинные коды процессора, выбранного в качестве целевого.

2.4. Проверить работоспособность созданного ассемблера путем компиляции тестовых примеров.

3. Порядок оформления отчета

3.1 Отчет должен содержать:

– описание регистров и команд процессора;

– описание примененного подхода по разбору исходного текста;

– описание поддерживаемых форматов команд ассемблера;

– протокол испытаний ассемблера с приведенным текстом программы и созданным машинным кодом;

– вывод по проделанной работе с перечислением достигнутых результатов.

Исходный текст программы оформляется в виде приложения к отчету.

Контрольные вопросы:

1. Что такое кросс-компиляция?

2. Какие классы грамматик подразумевает классификация Хомского?

3. Какие стадии существуют у процесса компиляции?

4. Какой практический смысл у разделения компиляции на компиляцию переднего плана (front-end) и компиляцию заднего плана (back-end)?

5. Требуется ли при определении имени команды знать положение битового поля, кодирующего эту команду, в двоичном представлении команды?

6. Ассемблер процессора x86 использует команду mov для всех вариантов пересылки между ресурсами процессора (регистр-регистр, регистр-память, загрузка литерала в регистр и т.д.). В процессоре Intel 8080 использовались модификации команды mov для разных сочетаний операндов (mov для регистр-регистр, mvi для загрузки литерала и т.д.). Какой вариант проще реализовать? В чем могут заключаться возможные недостатки более простого варианта?

7. Какие имена команд ассемблера используются для выполнения перехода к заданному адресу? Используйте для ответа справочные материалы производителей процессоров.

8. Как организуется переход назад и переход вперед в простейшем варианте ассемблирования?

Сведения об авторах

Потехин Дмитрий Станиславович, доктор технических наук, доцент, профессор кафедры «Вычислительная техника» РТУ МИРЭА.

Тарасов Илья Евгеньевич, доктор технических наук, доцент, профессор кафедры «Корпоративные информационные системы» РТУ МИРЭА.