|  |  |
| --- | --- |
| ДИСЦИЛИНА | **Схемотехника устройств компьютерных систем Часть 2** |
|  |  |
| ИНСТИТУТ | **ИТ** |
| КАФЕДРА | **вычислительной техники** |
|  |  |
| ВИД УЧЕБНОГО | **Лекция** |
| МАТЕРИАЛА |  |
| ПРЕПОДАВАТЕЛЬ | **Тарасов И.Е.** |
|  |  |
| СЕМЕСТР | 3 |
|  |  |

2

12. Процессорное ядро – дополнительные сведения о проектировании

12.1. Регистровая модель процессора и регистровый файл.

Регистровая модель процессора описывает процессор в том виде, в котором он представляет интерес для программиста. Язык ассемблера предусматривает описание команд в виде тех действий, которые производятся с регистрами процессора, которые в целом могут восприниматься как переменные программы. Кроме регистров, хранящих обрабатываемые данные, ряд регистров имеет специальное назначение.

К обязательным регистрам относится счетчик команд. Традиционно он имеет имя PC (Program Counter), однако для процессоров Intel этот регистр называется IP (Instruction Pointer). С точки зрения схемотехники, этот регистр содержит адрес следующей команды, читаемой из памяти программ.

Для ранних вариантов процессоров x86 (как и для ряда других процессоров) используется сегментированная адресация. Это означает, что содержимое регистров, содержащих адреса, выступает в качестве смещения относительно некоторого базового адреса. Такой подход требуется из-за того, что 16-разрядный регистр способен адресовать только 216 = 65536 ячеек памяти, а в ранних процессорах i8086 адресуемая память составляла 1 Мбайт, что требовало 20 разрядов адреса. Поэтому адреса в i8086 формируются следующим образом. К начальному адресу, хранящемуся в специальном сегментном регистре, добавляется смещение согласно формуле addr = segment \* 16 + offset, где segment – содержимое сегментного регистра, offset – смещение. Выполняемые команды читаются по адресу cs\*16+pc. Для удобства такое выражение записывается в i8086 также как CS:IP. Сегментные регистры в i8086 имеют предопределенное назначение:

– CS (code segment) – сегментный адрес кода;

– DS (data segment) – сегментный адрес данных, используется по умолчанию для чтения и записи данных;

– SS (stack segment) – сегментный адрес стека, используется совместно с указателем стека SP;

– ES (extra segment) – дополнительный сегмент, использующийся в некоторых командах доступа к памяти.

В процессоре i80386 были добавлены также сегментные регистры FS и GS. Способы адресации процессора i80386 были существенно расширены относительно базового варианта i8086 и в целом являются предметом отдельного рассмотрения. Здесь же имеет смысл ограничиться примером хорошо знакомой многим разработчикам архитектуры

Сегментированная адресация памяти была использована в i8086 для расширения адресного пространства, поскольку размер адреса оказался больше, чем размер регистров данных. Ее можно рассматривать как практический прием, преследующий те же цели (т.е. адресация очень больших объемов памяти), однако для практических примеров может оказаться неудобным следить за содержимым сегментного регистра. Постоянная перезагрузка сегментного регистра приведет к снижению производительности процессора.

Другим практически обязательным регистром является указатель стека. Для большинства процессоров он обозначается как SP (Stack Pointer). Стек является практически обязательным элементом процессорного проекта, так как вызов подпрограмм требует запоминания адреса возврата. Если адрес возврата запоминается в отдельном единственном регистре, вложенные вызовы подпрограмм станут невозможны. Простой путь обеспечения вложенных вызовов подпрограмм – организация области памяти в виде стека, называемого также память LIFO (Last In, First Out, т.е. «последним зашел – первым вышел»). Стек удобнее всего представлять в виде стопки листов бумаги, где последний листок, положенный на вершину стопки, будет первым с нее снят. Соответственно, команды вызова подпрограммы call и возврата из подпрограммы ret должны корректировать содержимое регистра SP, «помещая» и «снимая» числа со стека. Физически числа со стека не удаляются, вместо этого в регистр SP помещается новый адрес, указывающий на следующую ячейку памяти. Традиционно занятые ячейки стека располагаются в конце памяти, а помещение нового числа уменьшает регистр SP. Это связано с тем, что в ранних процессорах стек и данные располагались в одной и той же памяти, и было неудобно резервировать неизвестное заранее количество ячеек для стека в начальных адресах памяти. Поэтому вершина стека располагалась в наибольшем («старшем») адресе памяти и стек рос навстречу данным.

Основные операции с данными выполняются с помощью регистров общего назначения (РОН). Они обычно обозначаются буквами латинского алфавита A, B, C, или R0-Rx (например, R0-R7 или R0-R31). Для обозначения частей регистра используются символы L (Low, младшая часть) и H (High, старшая часть). Регистровая модель процессора i8086 показана на рис. 12.1. На этом рисунке видно, что регистры общего назначения разбиты на две части, каждая из которых является 8-разрядной. Части каждого регистра образуют 16-разрядный регистр, обозначаемый собственным именем, т.е. название AX соответствует 16-разрядному регистру, а AH и AL – его половинам.

Другим важным регистром, связанным с обработкой данных, является регистр флагов. Он обычно обозначается F или Flags и содержит одноразрядные признаки выполнения арифметических и логических операций. Широко используемыми флагами являются:

1. Флаг нуля (Zero Flag, обозначается как Z или ZF). Этот флаг равен 1, если результат последней выполненной операции был равен нулю.

2. Флаг переноса (Carry Flag, CF). Устанавливается в 1, если при выполнении последней математической операции результат не уместился в разрядной сетке (произошел перенос в следующий, отсутствующий разряд или заем из отсутствующего разряда). С помощью флага переноса оказывается возможной организация операций с разрядностью большей, чем регистры общего назначения, поскольку установленный флаг переноса свидетельствует, что при сложении старших частей числа необходимо учесть перенос, полученный при операции над младшими частями.

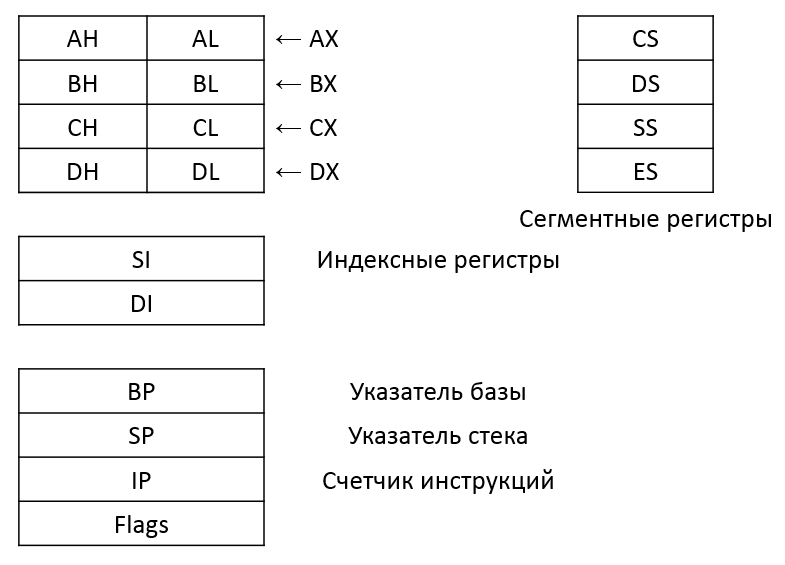
3. Флаг четности (Parity Flag, PF). Устанавливается в 1, если количество разрядов результата, равных 1, является четным. Эта проверка явно отличается от проверки на четность самого результата (у всех четных чисел младший разряд двоичного представления равен 0), и используется для организации простейшей схемы целостности данных, принимаемых по внешним интерфейсам.

4. Флаг знака (Sign Flag, SF). Копирует старший разряд результата, поскольку в дополнительном двоичном коде старший разряд свидетельствует о знаке числа. Копирование этого разряда в отдельный регистр флага позволяет использовать команды условного перехода, выполняемые по результату проверки флага. Это быстрее, чем проверять старший разряд числа отдельной командой.

5. Флаг переполнения (Overflow Flag, OF). Этот флаг является вспомогательным для флага переноса и устанавливается в 1, если результат не может быть представлен в разрядной сетке регистра назначения. Отличием от флага переноса является то, что флаг переполнения устанавливается в случаях, когда число из-за переполнения изменяет свой старший разряд. Например, для 8-разрядного регистра результат операции 127+1 формально помещается в 8 разрядов, однако получившееся число 128 из-за установленного старшего разряда будет трактоваться как отрицательное, хотя результат подразумевался положительным. Ввиду этого флаг CF будет равен 0, но флаг OF установится в 1.

Влияние различных команд на флаги является предметом отдельного рассмотрения. Например, общей практикой является то, что команды обычной загрузки в регистры не влияют на флаги, т.е. пересылка в регистр нулевого значения из другого регистра не устанавливает флаг нуля. Однако часто флаг переноса устанавливается в 0 при выполнении поразрядных логических команд, что отражает тот факт, что результат целиком поместился в регистр назначения.

Кроме флагов, устанавливаемых по результатам арифметических и логических операций, процессоры могут иметь флаги, управляющие их работой. Например, возможность реакции на прерывания может быть глобально разрешена и запрещена регулированием соответствующего флага (IF, Interrupt Flag в x86).



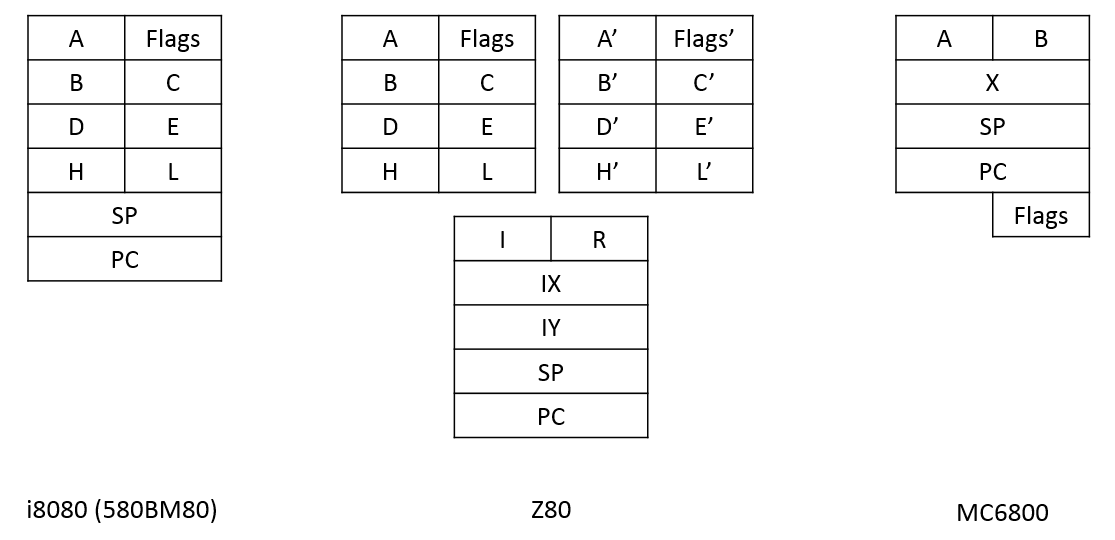
*Рис. 12.1. Регистровая модель процессора i8086*

На рис. 12.2 приведены регистровые модели процессоров i8080 (Intel), Z80 (Zilog) и MC6800 (Motorola). Все эти процессоры относятся к середине 70-х годов и достаточно показательны для изучения. Процессор i8080 (имеющий советский аналог 580ВМ80) имел регистр-аккумулятор A, регистры общего назначения B, C, D, E, H, L и регистр флагов. Отдельными 16-разрядными регистрами были описанные выше PC и SP. Для регистра-аккумулятора были доступны некоторые команды, которые не могли быть выполнены с другими регистрами общего назначения. С другой стороны, остальные регистры могли объединяться в 16-разрядные пары BC, DE и HL, которые использовались для адресации памяти и операций над 16-разрядными числами. Для пары HL был доступен более широкий перечень команд.

Таким образом, регистры процессора i8080 не полностью идентичны. Это свойство является достаточно важным при планировании совместной работы аппаратной и программной компонентов системы.

Процессор Z80 был создан в 1976 году компанией Zilog в качестве некоторого ответа на проект i8080. Сохраняя обратную совместимость по машинному коду (т.е. выполняя все программы для i8080), процессор Z80 имел также вспомогательные индексные регистры IX и IY (выполнявшие в целом ту же роль, что и регистровая пара HL), а также имел так называемый теневой набор регистров, показанный на рис. 12.2 в виде блока регистров со штрихами. В конкретный момент времени был активен только один из наборов, а быстро переключение наборов производилось всего одной командой. При этом выполнялось не физическое перемещение данных, а обычная перекоммутация, причем было невозможно определить, какой именно физический набор регистров активен в данный момент. Процессор Z80 оказал заметное влияние на российскую школу цифровой техники, поскольку был основой компьютера Sinclair ZX-Spectrum, популярного в России в начале 90-х годов и собиравшегося самостоятельно широким кругом радиолюбителей.

Процессор MC6800 компании Motorola имел всего два регистра-аккумулятора A и B, индексный регистр X, а также обязательные регистры SP и PC. Интерес представляет наличие отдельного индексного регистра (этот процессор был выпущен раньше, чем Z80 с индексными регистрами IX, IY). Кроме того, вся работа с данными должна производиться с помощью всего двух регистров-аккумуляторов, что подразумевает интенсивное использование памяти.



*Рис. 12.2. Регистровые модели некоторых процессоров*

Прежде чем приступать к перечислению возможных для процессора команд, необходимо рассмотреть основные подходы к проектированию системы команд. Бессистемное назначение кодов операций скорее всего приведет к хаосу как в схемотехнических решениях, так и в инструментальном программном обеспечении.

Наиболее известной классификацией является разделение процессоров на CISC и RISC. Эти аббревиатуры означают соответственно Complex Instruction Set Computer и Reduced Instruction Set Computer, т.е. компьютеры (процессоры) с полным набором команд и процессоры с сокращенным набором команд. Это разделение является уже достаточно старым, и соответствует выбору между сложным многотактным управляющим автоматом и простой системой работы, при который каждый машинный цикл соответствует завершенной команде. Именно RISC-подход был приведен в примере процессора, рассмотренного в предыдущей статье цикла. Современные процессоры обычно выполняются в соответствии с архитектурой RISC, однако это не означает, что их возможности в чем-то сокращены. Речь идет о том, что в состав команд RISC включены только те команды, которые имеют простую реализацию и не требуют для выполнения нескольких машинных циклов. Команды CISC обычно могут быть выражены программно с помощью нескольких команд RISC.

Для системы команд можно также упомянуть несколько важных свойств. Первое из них это **ортогональность**, которое находится в тесной связи с регистровой моделью процессора. Под ортогональностью понимается наличие множества методов адресации данных, которые могут быть использованы с любым сочетанием регистров процессора. Термин ведет происхождение от определения ортогональных (непересекающихся) векторов в геометрии. Под ортогональной системой координат, основанных на таких векторах, можно понимать некоторое пространство проектирования, в котором одно действие не оказывает влияние на другие. Например, в ортогональной системе команд можно выбирать один из операндов команды, не заботясь о том, какие ограничения это накладывает на второй операнд.

В примерах регистровых моделей, показанных выше, ортогональность в чистом виде нигде не наблюдается. Регистры часто имеют специальное назначение и с ними могут выполняться команды, недоступные для других регистров. Например, в i8086 регистр AX выступает аккумулятором, регистр BX – адресом памяти, CX – счетчиком циклов, а DX – операндом для команд умножения, деления и операций ввода-вывода. Можно указать, что введение частичной специализации позволяет в ряде случаев упростить аппаратную часть процессора и сократить разрядность команд. Примером специально введенной неортогональности является набор команд Thumb в процессорах ARM. В этом режиме команда имеет всего 16 разрядов вместо 32, однако доступны не все регистры и не все сочетания операндов.

Решение о том, следует ли использовать ортогональную систему команд, является результатом тщательных многоплановых исследований. В первую очередь играет роль используемая программная модель вычислений и типичные задачи, которые будут решаться с применением разрабатываемого процессора. Показательным примером является расширение Jazelle, которое, как и Thumb, используется как разновидность системы команд ARM. Это расширение предназначено для аппаратного ускорения выполнения приложений, написанных на языке Java, которые используют специфичную вычислительную модель – байт-код, выполняемый на виртуальной стековой машине.

Другой важной на практике характеристикой является **адресность** команд. Под адресностью понимается количество адресов (индексов регистров процессора), которые участвуют в выполнении команды. «Адрес» можно трактовать в широком смысле, как любое указание на ресурс процессора, участвующий в вычислениях или получающий результат.

Примером трехадресной команды является команда вида

R1 = R2 + R3

В данном случае команда содержит указания на три регистра – получатель результата, первый операнд и второй операнд. Для того, чтобы определить все три регистра, команда должна иметь три битовых поля, в которых будут помещены соответствующие номера. Размер этих полей зависит от количества регистров, которые можно адресовать таким способом. Например, 16 регистров потребуют 4-разрядных полей. Примером 3-адресных команд являются команды процессоров ARM.

В двухадресной команде регистр назначения совпадает с одним из операндов. Примером двухадресной архитектуры является Intel 8086. Например, рассматривая команду ассемблера *add ax, bx*, в ее описании можно увидеть, что действием команды является сложение данных из регистров *ax* и *bx* и помещение результата в *ax*. Таким образом, первый операнд команды одновременно является и получателем результата.

Одноадресные команды используются в ряде процессоров цифровой обработки сигналов (например, Texas Instruments TMS320). В таких процессорах часто используется выделенный регистр-аккумулятор, который является и первым операндом, и получателем результата.

Нуль-адресная, или безадресная система команд также существует. Как следует из названия, она должна однозначно определять все операнды команд. Может показаться, что единственным вариантом, по аналогии с одноадресной архитектурой, является использование единственно возможного сочетания регистров для выполнения всех операций. Однако существует еще одна вычислительная модель, представляемая безадресными командами – стековые процессоры. В стековых вычислениях команды автоматически применяются к операндам, находящимся на вершине стека данных, а результат помещается также на стек. Примером является виртуальная стековая машина Java, аналогичный подход использован в Common Intermediate Language виртуальной машины .net корпорации Microsoft. Особенностью стековых вычислений является их независимость от конкретной регистровой архитектуры. Нетрудно представить, что машинный код, использующий только 4 регистра, будет недостаточно эффективен для процессора, имеющего 16 или 32 регистров. Однако код, ориентированный на 32 регистра, не сможет быть перенесен на процессор с меньшим количеством регистров. Стековое представление в данном случае предоставляет промежуточную вычислительную модель, которая может быть эмулирована любой регистровой архитектурой (представлением стека в памяти), а специальные процессоры с аппаратной поддержкой стека способны существенно ускорить выполнение стековых команд.

В стековом процессоре используется отдельный стек данных, на который не помещаются адреса возврата из подпрограмм (для этого используется обычный стек). Соответственно, используется отдельный регистр – указатель вершины стека данных. Это сделано для того, чтобы отдельные подпрограммы стековой модели вычислений могли оставлять результаты на вершине стека, которые потом будут использованы другими подпрограммами.

Исторически стековая модель вычислений была реализована в языке программирования Форт (Forth). Кроме простой вычислительной модели, язык также отличается простой в реализации грамматикой, делающей его пригодным для быстрой разработки инструментального программного обеспечения. Речь идет в данном случае не столько об использовании существующих компиляторов Форта, сколько о реализации элементов Форт-машины в программном обеспечении для быстрого построения генераторов кода для новых процессоров.

Виды регистровых моделей с различной адресностью приведены на рис 12.3.



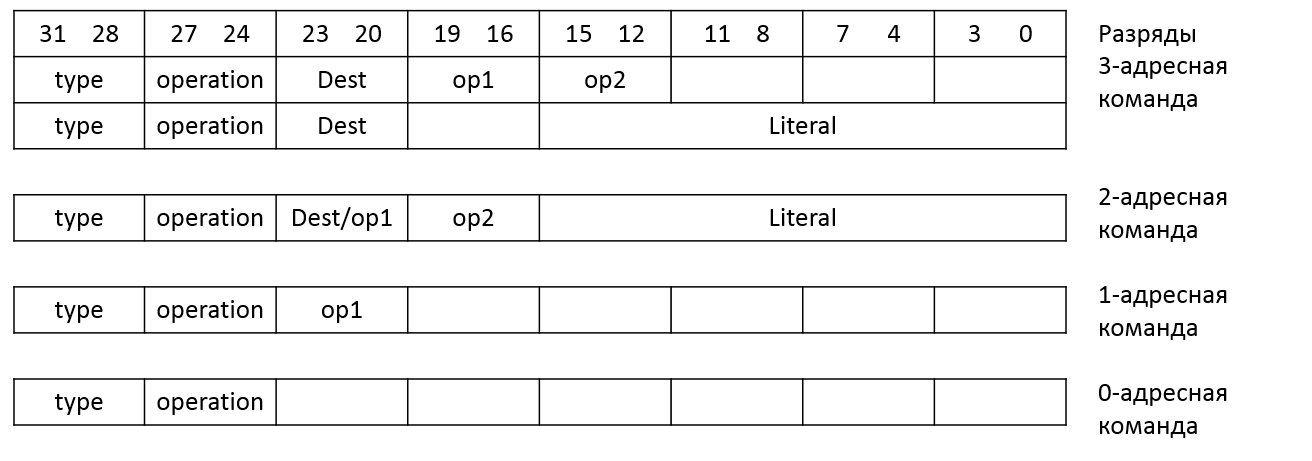
*Рис. 12.3. Виды регистровых моделей*

Команды с адресностью, большей 3, также могут быть реализованы. В данном случае речь может идти о процессорах с несколькими командами, выполняемыми параллельно – т.н. архитектура VLIW (Very Long Instruction Word). Несколько выполняемых команд требуют и несколько наборов операндов, поэтому адресность может быть равна 4, 6 и более. Архитектура VLIW в настоящее время получила развитие в виде архитектуры EPIC ((Explicitly Parallel Instruction Computing, т.е. архитектура с явно заданным параллелизмом) и может быть реализована в ПЛИС вследствие возможности описания памяти с разрядностью 64, 128 и более.

Не вполне очевидным вариантом 4-адресной команды является задание, кроме индексов операндов и получателя, также и адреса следующей команды. Можно рассматривать такую команду в качестве параллельной выполняющихся инструкций, одна из которых представляет собой обычную математическую операцию, а вторая – команду перехода. Обоснованность такого подхода определяется прежде всего интенсивностью использованию команд перехода, а также общими критериями эффективности, выбранными для процессора.

Вообще говоря, выбор архитектуры команд является предметом отдельного рассмотрения применительно к каждому конкретному проекту. Сильное влияние на эффективность выбранной организации команд оказывают типичные задачи, которые планируется решать на разрабатываемом процессоре. Поэтому невозможно заранее указать наиболее эффективную регистровую модель или подход к кодированию команд.

Пример формата команд для процессоров с различной адресностью показан на рис. 12.4. Можно убедиться, что при уменьшении адресности в команде освобождаются дополнительные поля, поэтому наибольшая компактность кода теоретически достижима в безадресной архитектуре, а трехадресная дает наибольшие возможности в генерации машинного кода.



*Рис. 12.4. Примеры организации команд для процессоров с разными видами регистровых моделей*

Процессоры в целом имеют весьма разнообразные наборы команд, состав которых зависит от назначения процессора и ограничен его программной моделью. Тем не менее, можно выделить группы команд, которые часто присутствуют в процессорах.

Команды можно сразу разделить на крупные группы. Например, команды арифметики, сдвига и манипуляций с битовыми полями объединяет то, что они работают с данными и модифицируют регистры или память. Поэтому реализация еще одной команды подобного типа будет достаточно простой. Однако команды перехода воздействуют на счетчик команд, и их реализация будет существенно отличаться от команд, работающих с АЛУ и регистрами.

Поэтому в поле type команды, показанной на рис. 5, можно занести код типа этой команды, выбирая его из списка

– безоперандные команды, такие как NOP (нет операции), прочие специальные команды, модифицирующие один из служебных регистров;

– команды работы с данными: арифметико-логические операции

– команды работы с памятью и внешними устройствами: чтение/запись, такие команды требуют формирования адреса для внешнего по отношению к процессорному ядру устройству;

– команды управления порядком выполнения программы: переходы, условные переходы, вызовы подпрограмм и возврат из подпрограмм; для этих команд следует рассмотреть организацию конвейера, а для команд вызова/возврата также и организацию работы со стеком.

Для указания на данные используются различные методы адресации. Список методов не является обязательным или исчерпывающим, однако процессор, неспособный указать на операнды ни одним из способов, не имеет практического смысла.

Регистровая адресация подразумевает указание на операнд, содержащийся в одном из регистров. Примером прямой адресации является выражение вида ax или r0, причем номер регистра в явном виде присутствует в поле команды. Это самый простой вид адресации, который, тем не менее, недостаточен для работы с процессором.

Непосредственная адресация (Indirect). Если прямую адресацию можно назвать «адресация посредством регистра», то при непосредственной адресации часть содержимого команды загружается в регистр назначения. Непосредственный операнд также называют литералом (literal – «буквальный»). Например, если команда загрузки в регистр R0 имеет код 123, то загрузить в R0 число 2 можно командами «123 2». В этом примере подразумевается, что команда самостоятельно прочитает следующее число из памяти программ и использует его как литерал. Если размер команды достаточно большой, то литерал может быть размещен и внутри кода команды, как показано на рис. 5. Такой подход можно рассматривать только в качестве альтернативного, поскольку он упрощает кодирование на HDL, однако литералы занимают дополнительное место в коде команды.

Прямая адресация указывает на данные в памяти, адрес которых содержится в команде. Если рассматривать аналогию с предыдущим примером, последовательность «123 2» должна загрузить число, содержащееся в ячейке памяти по адресу 2. Такая адресация требует дополнительной работы с памятью данных.

Косвенная адресация использует вместо адреса не число (литерал), а содержимое одного из регистров. В командах ассемблера такой способ адресации обычно обозначается скобками: mov ax, [bx]. Подразумевается, что регистр bx содержит адрес ячейки памяти, где расположены требуемые данные.

Эти методы адресации не образуют исчерпывающего списка. Например, способы адресации могут комбинироваться для достижения большей гибкости и более полного соответствия типовым алгоритмам. Например, для i80386 возможна команда

mov eax, dword ptr [ebp + 100 + ecx\*4]

В этой команде используется сложный способ вычисления адреса, однако он может быть обоснован с точки зрения программирования. Представим, что в регистр ebp загружен начальный адрес сложной структуры данных, в которой со смещения 100 начинается массив 4-байтовых полей. Номер поля загружен в регистр ecx, поэтому выражение в квадратных скобках позволяет вычислить полный адрес интересующего нас элемента в сложной структуре данных. При этом содержимое регистров ebp и ecx не изменяется и может быть использовано в дальнейшем для вычисления адреса другого элемента.

Команды перехода используются для управления порядком выполнения программы. Эти команды работают преимущественно с регистром PC, загружая в него новое значение (или сохраняя линейный порядок выполнения, если речь идет об условном переходе).

Для команды перехода в языке ассемблера обычно используют производные от слова Jump («прыжок») или Branch («ответвление»). Мнемоника ассемблерной команды может выглядеть как JMP, JP или BR. Для команд перехода также справедливо подразделение по методам адресации – переход может быть совершен по адресу, указанному в коде, по адресу, содержащемуся в указанной ячейке памяти, или по адресу, хранящемуся в регистре. Полный набор способов представления адреса не является обязательным и применяется в процессорах с различными вариациями.

Кроме указания адреса, на который необходимо совершить переход, используется также относительная адресация перехода (relative jump). При этом указывается не сам адрес, а то, на сколько он отстоит от команды, которая должна была бы выполняться при нормальном продолжении программы. Такой подход имеет как минимум два основных преимущества:

– программы становятся переносимыми, т.е. перемещение фрагмента программного кода по другому абсолютному адресу в памяти не нарушит относительное расположение команд, поэтому переходы с относительной адресацией будут по-прежнему выполняться правильно;

– для относительных переходов часто используется формат записи смещения в пределах одного или двух байтов, что сокращает размер программы, а переходы в пределах -128...+127 или -32768…+32767 адресов достаточно характерны для фрагментов программ, насыщенных проверками условий, так что команды if часто могут генерировать смещение, укладывающееся в короткий переход.

Команды условного перехода традиционно используют проверку условий, представленную флагами. Иными словами, для выполнения условного перехода сначала вычисляется условие, а затем производится переход при условии равенства нужного флага 0 или 1. Наличие в процессоре флага не делает обязательным добавлением команды условного перехода с проверкой этого флага (тем более для обоих вариантов значения флага). Перечисленные выше флаги, устанавливаемые по результатам выполнения команды в АЛУ, являются наиболее употребительными.

Команда вызова подпрограммы использует в основном те же способы представления адреса, как и команда перехода. Отличием от команды перехода является помещение на стек адреса команды, который будет использован для продолжения выполнения работы. В свою очередь, команда возврата из подпрограммы снимает со стека адрес и передает на него управление. Команды вызова подпрограмм также могут быть условными.

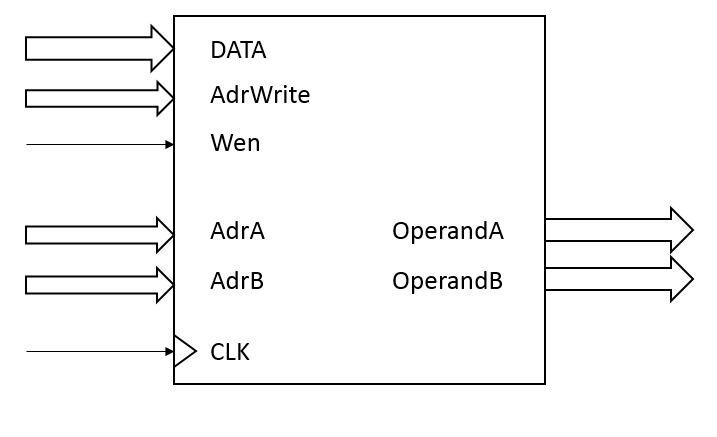
Отдельно можно упомянуть команды для работы с внешними устройствами. Несмотря на относительную простоту, управление работой внешних схем требует некоторого планирования. Например, запись данных во внешнее устройство необходимо представить отдельной командой, предусмотрев способ задания адреса устройства, данных, и назначив отдельный код команды, выполнение которого может не приводить к изменениям внутри ядра процессора, однако формировать специальный сигнал сопровождения (разрешения записи) для внешней схемы.

Регистровый файл – это группа регистров, имеющих сходное назначение. Для регистрового файла обычно используется общий интерфейс, допускающий единообразный доступ к любому регистру. Поэтому удобнее описывать регистровый файл в виде линейного массива регистров.

Рассмотрим процесс проектирования регистрового файла с учетом рассмотренных выше подходов к регистровым моделям процессоров. Модуль должен использовать тактовый сигнал и производить запись данных по фронту этого сигнала. Поскольку внутри регистрового файла имеется несколько регистров, необходимо указать номер (адрес) регистра для записи. Кроме того, в ряде случаев запись не производится (очевидно, при выполнении команды nop, но это далеко не единственный пример), поэтому необходим отдельный сигнал разрешения записи данных.

Для получения операндов необходимо указывать их индексы. В общем случае они не совпадают с индексом регистра-получателя данных. Операнды являются выходами модуля.

Графическое изображение модуля регистрового файла представлено на рис. 12.5.



*Рис. 12.5. Интерфейс регистрового файла*

Описание регистрового файла можно произвести на основе объявления двумерного массива. Запись в регистр, адресуемый сигналом AdrWrite производится по фронту тактового сигнала, где сигнал Wen (разрешение записи) глобально управляет этим процессом (т.е. if (wen) Reg[AdrWrite] <= Data). В зависимости от аппаратной платформы может быть использовано как поведенческое описание, так и структурное, с прямым указанием подключения требуемых аппаратных компонентов.

Чтение содержимого регистров с номерами AdrA, AdrB технически может производиться асинхронно, т.е. без фиксирования результата по фронту тактового сигнала. Далее будет рассмотрен вариант трехступенчатого конвейера, где операнды не просто выбираются мультиплексором, а записываются в специальные регистры операндов по фронту тактового сигнала.

12.2. Арифметико-логическое устройство.

Арифметико-логическое устройство (АЛУ) – это основной вычислительный узел процессора, выполняющий операции над данными. АЛУ выполняет операции над входными операндами (получаемыми из регистрового файла или памяти данных), основываясь на коде выполняемой команды. Соответственно, его реализация достаточно проста - простой мультиплексор, управляемый кодом команды (по крайней мере, частью этого кода).

Состав команд АЛУ может варьироваться в достаточно широких пределах в зависимости от назначения процессора. В целом существует узнаваемый набор операций, характерных для АЛУ, но это не означает, что все операции или даже все группы операций должны быть реализованы в конкретном проекте. В целом можно отметить, что для АЛУ можно ориентироваться на возможности языка описания аппаратуры, поскольку в нем уже описаны арифметические и логические команды.

Тем не менее, можно привести примерный перечень операций, характерных для АЛУ. При этом важно отметить различие в используемых форматах чисел.

Беззнаковое представление числа соответствует ситуации, когда отрицательные числа представляются в так называемом дополнительном двоичном коде. Имеется в виду, что отрицательное число не имеет специального признака, а определяется путем дополнения. Это означает, что число -X получается в результате вычисления выражения (0 – X), а поскольку при ограниченной разрядности вычитание из нуля возможно только при заеме из несуществующего разряда, число -1 представляется в виде всех разрядов, установленных в 1. Проверить это можно, прибавив к такому числу 1. Теоретически, 8-разрядное число 255 (0b11111111) при сложении с 1 даст 1\_0000\_000 (подчеркиванием выделены группы разрядов), но так как для 8-разрядного представления старший разряд будет некуда записать, результатом операции будет 0. Следовательно, число 0b11111111 можно трактовать как 8-разрядное представление числа -1.

Беззнаковое представление удобно для сложения и вычитания. Можно свободно оперировать с двоичными числами, не заботясь об отдельной обработке знака числа. Если конечный результат помещается в разрядную сетку регистра, любая последовательность сложений и вычитаний приведет к правильному результату.

Для смены знака числа, представленного в дополнительном коде, необходимо проинвертировать все разряды этого числа и прибавить к результату 1. Это можно проверить, изменяя знак числа 0000\_0001. Инвертирование дает 1111\_1110, что соответствует числу -2, тогда как исходный операнд был равен 1. Добавление к этому числу единицы даст правильное представление 1111\_1111.

Знаковое (signed) представление чисел подразумевает, что в его старшем разряде находится признак знака (0 для положительного, 1 для отрицательного числа), а в остальных разрядах – модуль числа. Таким образом, 8-разрядное число -1 будет записано как 0b1000\_0001.

Знаковое представление удобно для операций умножения и деления. Для этого достаточно умножить/разделить модули, а знак результата получить операцией ИСКЛЮЧАЮЩЕЕ ИЛИ над знаками операндов (разные знаки дадут в итоге 1, одинаковые – 0). Попытка перемножить числа, представленные в дополнительном коде, даст в итоге результат, соответствующий беззнаковому умножению. Поэтому знаковое и беззнаковое умножение должны быть реализованы отдельно.

**Арифметические операции.**

Сложение (addition) соответствует обычному сложению беззнаковых чисел. Сложение описывается выражением result <= a + b.

Сложение с учетом переноса (addition with carry, часто используется мнемоника adc) добавляет к результату значение флага переноса. Такая операция используется для обеспечения обработки чисел, имеющих разрядность больше, чем разрядность регистров. После сложения младших частей чисел командой add будет установлено соответствующее значение флага переноса, и последующие команды adc для старших частей слагаемых будут учитывать перенос, возникший при сложении предыдущих частей.

Вычитание (subtraction) в целом аналогично сложению и описывается выражением result <= op1 – op2. В отличие от сложения, важен порядок операндов.

Вычитание имеет свой аналог, учитывающий флаг переноса (subtraction with carry).

В ПЛИС умножение выполняется специализированными аппаратными блоками (DSP48 в FPGA Xilinx серии 7 и последующих, аппаратные умножители в предыдущих семействах). Синтезатор самостоятельно строит схему, состоящую из нескольких умножителей, при необходимости перемножить числа большой разрядности.

Для прототипирования СБИС необходимо учитывать, что умножитель представляет собой компонент со сложной структурой (наиболее простой подход – т.н. «дерево сумматоров», попарно складывающее частные произведения первого операнда на каждый из разрядов второго операнда). Топологическая реализация дерева сумматоров может представлять проблему в смысле достижения хороших показателей площади, тактовой частоты и потребляемой мощности. В FPGA умножители являются аппаратным компонентов с гарантированными характеристиками, поэтому могут использоваться свободно.

Операция деления в общем случае не имеет простой реализации для произвольных операндов. Как правило, эта операция выполняется последовательно, за несколько тактов, с реализацией алгоритма «двоичного деления в столбик». Поэтому деление обычно не реализуется в АЛУ в прямом виде.

Логические операции обычно имеют поразрядные (битовые) представления. Это означает, что числа рассматриваются как наборы отдельных разрядов, и логические операции применяются к каждой паре разрядов. Это отличается от операций булевой алгебры, выполняемой над числами в формате boolean, так как если считать, что любое ненулевое значение соответствует ИСТИНЕ, то результат выражения 1 AND 2 должен также дать ИСТИНУ с точки зрения проверки логического условия. Однако в поразрядных представлениях этих чисел нет разрядов, которые имели бы значение 1 одновременно, поэтому результат поразрядного И будет равен 0.

Поразрядные логические операции имеют следующие варианты.

– поразрядное И (and).

– поразрядное ИЛИ (or).

– поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (xor).

– поразрядная инверсия (not).

Возможны инверсные варианты этих операций, т.е., поразрядное И-НЕ и т.д., однако вопрос реализации полного набора поразрядных операций остается предметом изучения в каждом конкретном проекте. Частое использование какой-то операции в программах делает ее хорошим кандидатом на добавление соответствующей команды в АЛУ.

Операции сдвига имеют следующие варианты.

Логический сдвиг влево (shift left) описывается командой result <= opa sll 1, где команда sll является аббревиатурой Shift Left Logical. При выполнении этой команды битовое представление числа сдвигается влево, а в освободившийся младший разряд помещается 0. Допустим сдвиг на произвольное количество разрядов, если указать, например, opa sll 3. Сдвиг на переменное число разрядов требует реализации мультиплексора.

Логический сдвиг вправо (logical shift right) отличается от арифметического сдвига вправо (arithmetic shift right). Это связано с побочным эффектом операции сдвига. Сдвиг влево соответствует умножению на степени двойки, так что сдвиг на 1 разряд соответствует умножению на 2, сдвиг на 2 разряда – на 4 и т.д. Эта операция выполняется даже без аппаратного умножителя, который не имеет ограничений на значение сомножителя (с помощью единственного сдвига невозможно умножить на 3 или на 5). Аналогично, сдвиг вправо соответствует операции деления на степени двойки.

Если сопоставить эту информацию с наличием беззнакового представления числа, можно обнаружить некоторую проблему. Предположим, двоичное число -2 (имеющее представление 1111\_1110) делится на 2 операцией сдвига вправо. Если выполнить логический сдвиг, старший разряд станет равен 0, а младший разряд будет выдвинут за пределы разрядной сетки. Поэтому результатом сдвига будет 0111\_1111, т.е. 127. Таким образом, получен неожиданный результат: -2/2 = 127. Обязательное помещение логической единицы в старший разряд не решает проблему, поскольку теперь даже положительные числа будут превращаться в отрицательные. Правильным решением является помещение в старший разряд того же самого значения, которое было в нем до операции сдвига вправо.

Выполнять такое преобразование, или же реализовать обычный логический сдвиг, зависит целиком от смысла реализуемого алгоритма. Удобно иметь в составе АЛУ оба варианта сдвига несмотря на то, что они в принципе могут быть выражены через другие операции.

Операции вращения (rotation) подразумевают, что разряд, выдвигаемый из числа при вращении, помещается в него с другой стороны. Операция вращения также имеет разновидность, включающую в перемещение разрядов и бит переноса.

Команды сравнения используются для формирования логических (булевых) условий. Они могут быть как реализованы в АЛУ в виде отдельных операций, так и в качестве побочных эффектов при выполнении арифметико-логический действий. Например, после операции or ax, ax в процессоре i8086 флаг нуля будет установлен, если в ax содержался 0, а флаг переноса будет обязательно сброшен.

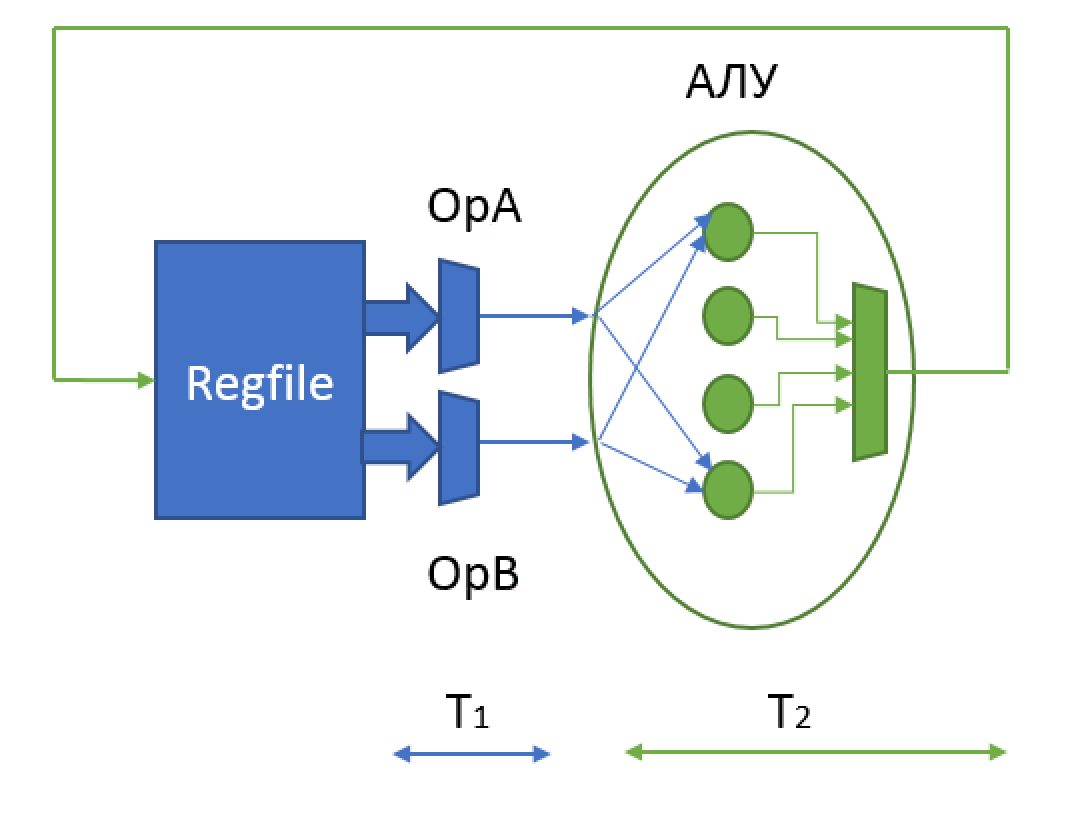
Прочие команды АЛУ добавляются по мере необходимости и могут описывать действия, являющиеся комбинацией различных известных команд. Например, можно добавить обмен старшей и младшей части операнда, смену направления разрядов (так, что число **1010**\_0001 превратится в 0001\_**0101**), наложение битовой маски на результат и т.д. Основанием для введения дополнительных команд, как и в других подобных случаях, является их частое использование в практических программах, для которых предназначается процессор.

12.3. Конвейер процессора. Архитектура с трехступенчатым конвейером.

Много процессорных архитектур являются конвейеризованными. Это означает, что преобразование данных происходит в них не за один такт, а с применением цепочки регистров, по которой данные из регистрового файла проходят последовательно, подвергаясь тем или иным преобразованиям на каждой стадии. Основной смысл конвейера – разбить линию с большой задержкой передачи данных на несколько более коротких линий, что позволит повысить тактовую частоту.

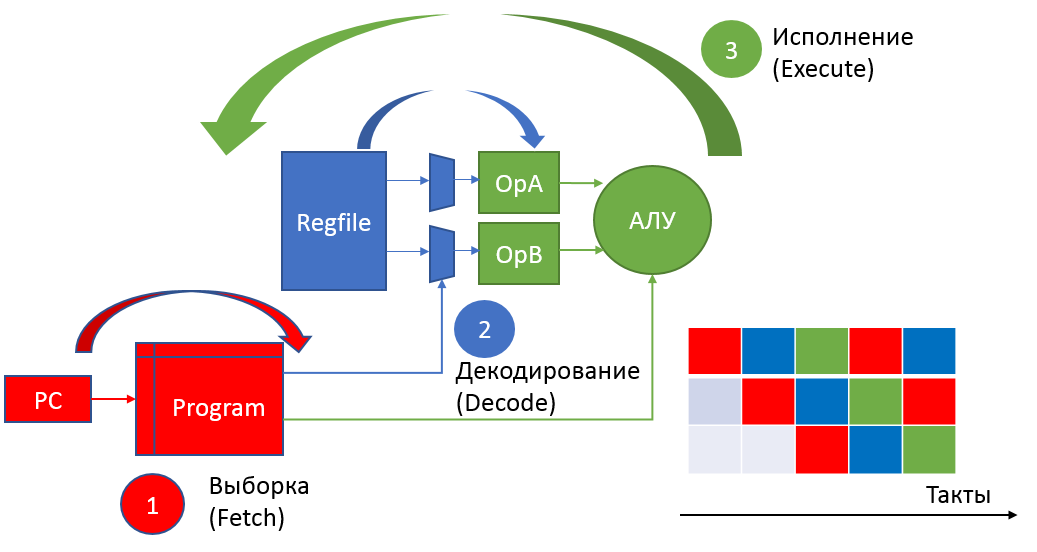
На рис. 12.6 показана структурная схема тракта данных процессора с регистровым файлом. На выходе регистрового файла находятся два мультиплексора, выбирающие из блока регистров операнды A и B. Эти операнды подаются на арифметико-логическое устройство, содержащее несколько вычислительных узлов для разных операций, выходы которых также подаются на мультиплексор, выбирающий один из результатов в зависимости от выполняемой команды.

На этом рисунке видно, что суммарная задержка такой схемы определяется задержками в ее составных частях. Если установить конвейеризующие регистры, можно будет повысить тактовую частоту. Очевидным местом для установки регистров является точка между мультиплексорами OpA, OpB и АЛУ. Возможно, она разделит задержку на не совсем равные части, однако именно здесь количество регистров будет минимально.



*Рис. 12.6. Иллюстрация к проблеме конвейеризации*

Если произвести такую установку, работа процессора выполняется за три такта, что проиллюстрировано рис. 12.7.



*Рис. 12.7. Порядок взаимодействия узлов процессора с трехступенчатым конвейером*

На первом такте (выборка, fetch) по значению счетчика команд PC производится чтение кода очередной команды.

На втором такте (декодирование, decode) на основе команды из регистрового файла выбираются операнды, участвующие в этой команде. Здесь не показаны действия с другими узлами процессора, которые тоже зависят от кода команды, однако их также можно выполнять на стадии декодирования.

На третьем такте (исполнение, execute) уже имеются операнды. В соответствии с кодом команды результат требуемой операции записывается в регистр назначения.

Можно обратить внимание, что код прочитанной команды также необходимо передать на стадию исполнения через конвейеризующий регистр. Если выполнять стадии команды последовательно, то после чтения кода команды на следующем такте на выходе памяти программ будет уже код следующей команды, а следовательно, информация о выбранной операции и номере регистра назначения должна быть также передана на следующую стадию через конвейеризующий регистр.

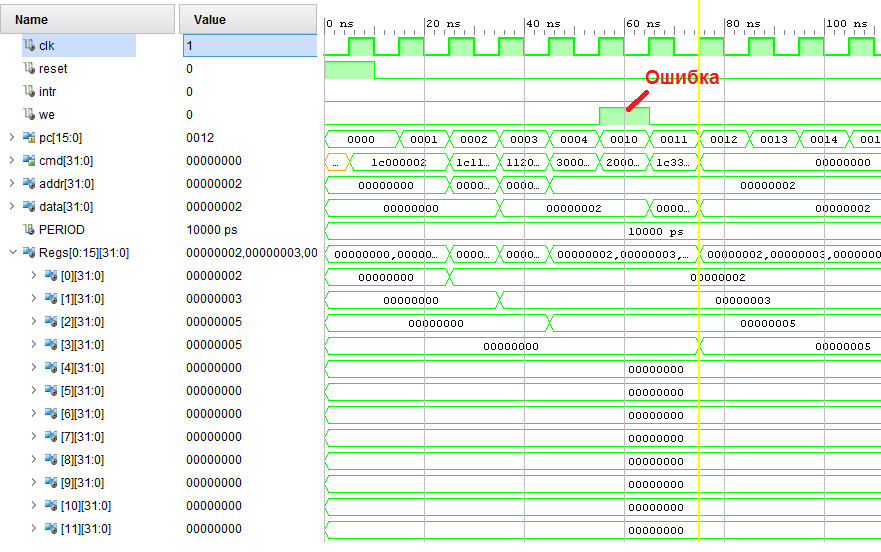
Введение дополнительных стадий конвейера образует задержку выполнения программы на дополнительное число тактов. Рассмотрим программу следующего вида, проиллюстрированную рис. 12.8.

1. (PC=3). Выполнение R2 = R0 + R1. Из памяти прочитана команда JMP, вычисление PC = PC +1.

2. (PC=4). Выполнение команды JMP, **из памяти прочитана команда вывода в порт**, присваивание PC = 0x10.

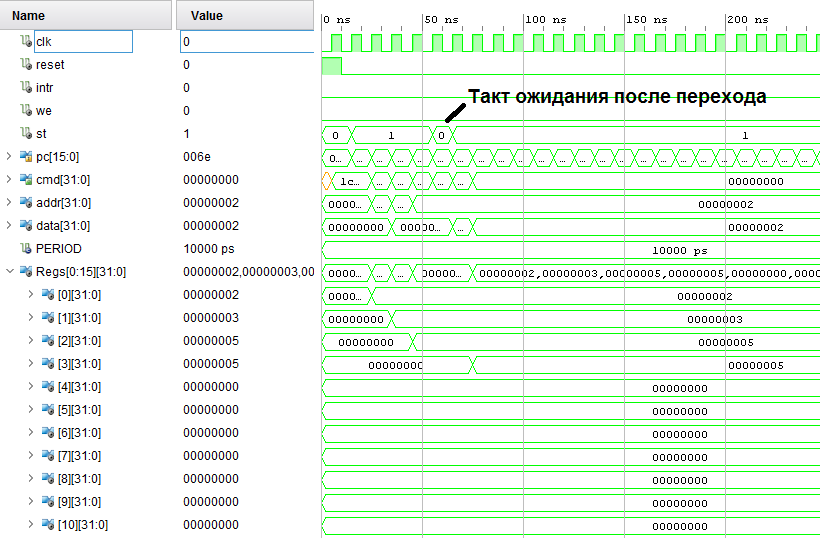
3. (PC=0x10). **Выполнение команды вывода в порт, прочитанной на предыдущем такте**, чтение новой команды.

Таким образом, задержка в конвейере на один такт привела к тому, что команда, читающаяся из последовательного адреса памяти программ одновременно с присваиванием нового значения PC, успела попасть на выход памяти программ и выполниться на следующем такте, когда новая команда по адресу PC=0x10 еще только читалась.



*Рис. 12.8. Временные диаграммы работы тестовой программы с выполнением инструкции, следующей за командой перехода*

Для решения этой проблемы необходимо ввести *состояние конвейера*, как показано на рис. 12.9. Это специальный сигнал st, который сбрасывается, если процессор переходит не к адресу PC+1, и устанавливается в противоположном случае. Сигнал st проверяется при записи данных в регистры, память или внешние устройства, если он равен 0, запись не производится.



*Рис. 12.9. Временные диаграммы выполнения тестовой программы с введением такта ожидания после нарушения линейного порядка выполнения*

Введение в конвейер процессора состояния ожидания не является единственно возможным вариантом. В некоторых процессорных ядрах существует понятие *отложенного перехода* – команды, которые успели попасть в конвейер, выполняются независимо от того, что они находятся после команды перехода. Это создает дополнительную нагрузку на компилятор, который должен размещать команду перехода в коде перед тем, как она действительно потребуется, или же компилировать команды NOP после переходов, чтобы избежать нежелательных побочных эффектов.

12.4. Зависимости по данным. Архитектура MIPS.

Рассматривая многотактные конвейеры, необходимо перейти к важной проблеме, которая неминуемо возникает при такой схеме – проблеме конфликтов и вызванных ими блокировок конвейера. Для иллюстрации можно использовать следующий псевдокод.

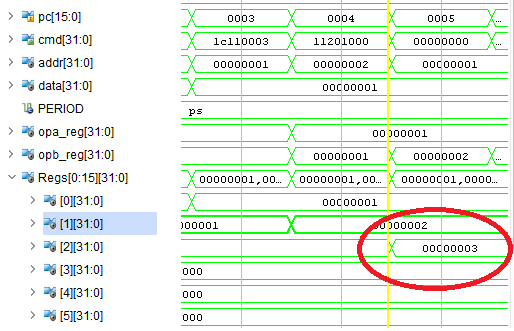
R1 = 1

R0 = 2

R1 = 3

R2 = R0 + R1

В приведенном листинге в регистры R0 и R1 загружаются операнды, которые затем складываются. Исходя из псевдокода, результатом должно быть число 5. Однако временные диаграммы, показанные на рис. 12.10, демонстрируют явную ошибку – вместо сложения чисел 2 и 3 складываются 2 и 1. Из псевдокода видно, что 1 – это ранее загруженное в R1 значение, и именно оно оказалось использовано для команды сложения.



*Рис. 12.10. Проблема зависимости по данным в конвейеризованном процессоре*

Такое поведение объясняется тем, что из-за конвейеризации процессору требуется дополнительный такт, чтобы обновить значение в регистре. В момент выполнения команды R1 = 3 следующая команда находилась на стадии «декодирование», т.е. читала операнды из регистрового файла. Поскольку новое значение для R1 находилось еще только на входе данных регистрового файла, по очередному фронту тактового сигнала произошло следующее:

1. Старое значение R1 записано в регистр операнда.

2. Новое значение записано в регистр R1.

Таким образом, в данном случае имеет место *зависимость по данным*. Она проявляется в том, что результат выполнения команды зависит от того, какие команды выполнялись непосредственно перед ней. Всего существуют 4 ситуации взаимного влияния команд.

1. «Чтение после чтения» (RAR, Read After Read) – из одного и того же физического ресурса (регистра или ячейки памяти) производятся два последовательных чтения. Эта ситуация не формирует конфликтов, поскольку первое чтение не влияет на содержимое регистра и не мешает прочитать его второй раз.

2. «Чтение после записи» (RAW, Read After Write) – наиболее проблемная ситуация, соответствующая показанной на рис. 4. При наличии конвейеризации может возникнуть ситуация, когда данные, предназначенные для записи, еще не попали в регистр (или память), однако следующая команда, находясь в конвейере, уже производит их чтение.

3. «Запись после чтения» (Write After Read)

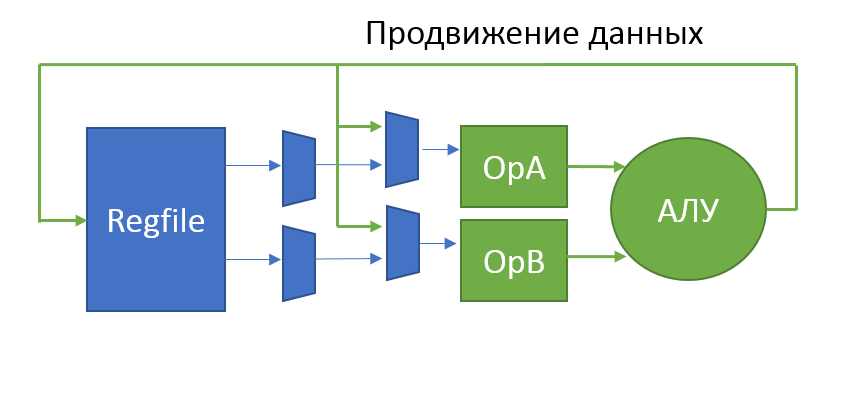
4. «Запись после записи» (Write After Write)

Эти ситуации, в принципе, не вызывают проблем. Обращать внимание на них следует только в случае более сложного варианта микроархитектуры, который еще не был рассмотрен, и соответствует случаю нескольких параллельно работающих конвейеров с внеочередным выполнение команд (обозначается термином out-of-order). По подобной схеме работает множество высокопроизводительных процессоров, где показательным примером является первый вариант процессора Intel Pentium, имевшего два конвейера (т.н. U- и V-конвейеры). В более сложном конвейере команды могут быть запущены по мере готовности данных для них и выполняться путем прохождения разного количества стадий конвейера. Может возникнуть ситуация, когда команда, прочитанная из памяти позже, запустится на исполнение и будет исполнена за меньшее количество тактов, чем предыдущая команда. В этом случае необходимо проверить, не используют ли эти команды один и тот же ресурс процессора, так как порядок записи в регистр назначения может быть нарушен.

Проектирование многотактных конвейеров с внеочередным исполнение команд требует существенно большего внимания, поскольку требуется отслеживать потенциальные конфликты по доступу к ресурсам процессора и приостанавливать выполнение команд в конвейерах, если не готовы операнды для них, или регистр, в который должна быть произведена запись, еще не был прочитан командой, запущенной на исполнение ранее.

Кроме того, в процессе доступа в память может возникнуть т.н. структурный конфликт. Если в конвейере предусмотрена последовательность «чтение данных из памяти – обработка данных – запись данных в память», то возможна ситуация, когда одна команда производит запись в память, но другая команда в то же время находится на стадии «чтение данных из памяти». В этом случае придется сделать выбор, какую из команд выполнять, поэтому либо не будет произведена запись, либо чтение из памяти не состоится. Разумеется, оба варианта приводят к ошибочному поведению процессора.

Процессор без блокировок в конвейере (Microprocessor without Interlocked Pipeline Stages) – это и способ организации конвейера, и конкретный продукт, разработанный компанией MIPS Technologies (ранее MIPS Computer Systems). В данной архитектуре используется полезный прием разрешения конфликтов по доступу к регистрам, называемый «продвижение данных» (data bypass). Его суть можно проиллюстрировать на рис. 12.11.



*Рис. 12.11. Прием «продвижение данных» для решения проблемы зависимости по данным в конвейере*

На рис. 12.11 можно видеть способ разрешения конфликта при последовательном доступе к одному и тому же регистру на запись и на чтение. Как было показано выше, такая ситуация должна вызвать конфликт, поскольку в момент записи нового значения на вход АЛУ попадает старое содержимое регистра. При реализации подхода data bypass новое значение записывается не только в сам регистр, но и в регистр операнда, если номер регистра для записи результата совпадает с номером регистра-операнда.

Несмотря на то, что на рис. 12.11 показан дополнительный мультиплексор, при должном подходе снижение тактовой частоты такой схемы окажется несущественным. Для этого необходимо убедиться, что данные из АЛУ подаются именно на последний мультиплексор в цепочке, т.е. после мультиплексирования одного из регистров в регистры OpA, OpB записывается либо значение регистра, либо данные из АЛУ. В этой случае суммарная задержка распространения сигнала от OpA, OpB через АЛУ увеличится всего на один мультиплексор 2-в-1.

12.5. Архитектуры с многоступенчатым конвейером.

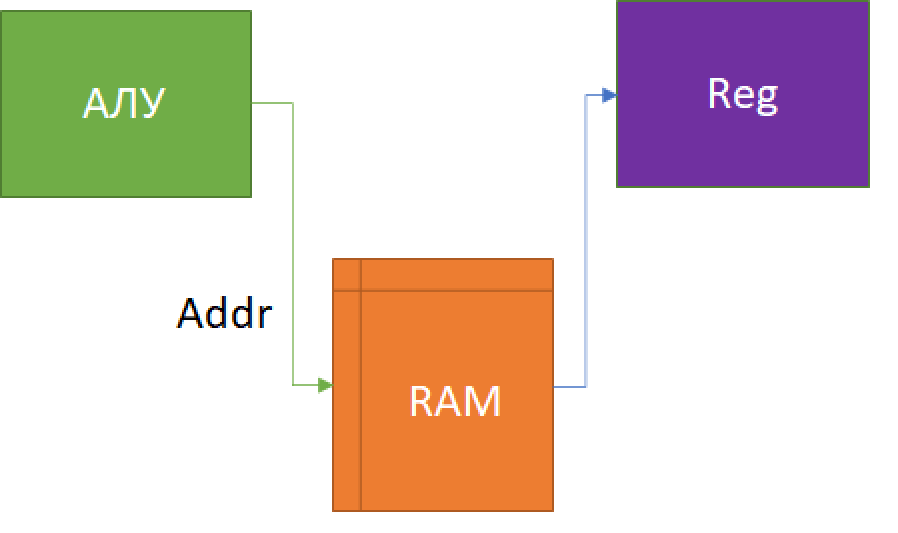
Дальнейшим развитием данной архитектуры является 5-ступенчатый конвейер, который также часто используется в процессорных ядрах. При рассмотрении 3-ступенчатого конвейера можно убедиться, что с его помощью невозможно обеспечить чтение данных из синхронной памяти за один машинный цикл (т.е. при прохождении команды по всем трем стадиям). Действительно, при выполнении команды вида mov R0, [R1] будут выполнены следующие действия:

1. Команда прочитана из памяти программ.

2. Содержимое регистра R1 записано в выходной регистр и подано на вход адреса для памяти.

3. Содержимое памяти по адресу, определяемым регистром R1, прочитано и находится на выходе блока памяти.

Соответствующая схема соединений показана на рис. 12.12. Поскольку адрес получается на стадии вычисления результата в АЛУ, он может иметь достаточно сложную структуру – например, являться суммой двух регистров, регистра и константы и т.д.

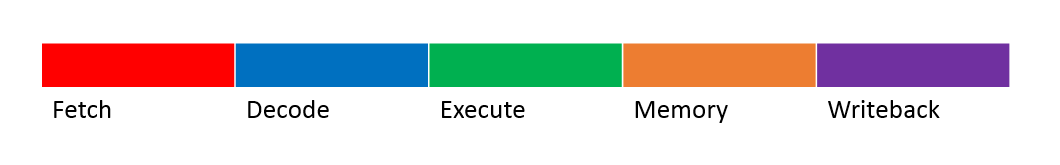


*Рис. 12.12. Стадия работы с памятью в процессоре с 5-ступенчатым конвейером*

Очевидно, что для завершения команды требуется еще один такт – запись прочитанного значения в регистр R0 (т.е. в регистр назначения). После трех тактов удалось добиться только того, что требуемое значение появилось на выходе блока памяти данных. Данная проблема имеет несколько схемотехнических решений, одно из которых – введение двухцикловых команд чтения из памяти. В таком случае первая команда инициирует собственно процесс чтения, а вторая (которая завершится на такт позже) запишет данные, находящиеся на выходе блока памяти, в регистр назначения.

Можно попытаться использовать в качестве адреса памяти значение операнда до конвейеризующего регистра. Тогда после чтения команды на стадии декодирования будет прочитана память данных, а на стадии исполнения прочитанное значение будет готово для записи. Однако такой подход потенциально вызывает конфликты другого типа, которые будут рассмотрены далее.

При интенсивной работе с памятью в конвейер могут быть добавлены такты, на которых и будут происходить чтение и запись в память. При 5-ступенчатой архитектуре эти такты носят название «Memory Phase» и «Writeback» (рис. 12.13). На 4-м такте производятся операция с памятью, а на 5-м (writeback) – запись результата по назначению.



*Рис. 12.12. Выполнение операций в процессоре с 5-ступенчатым конвейером*

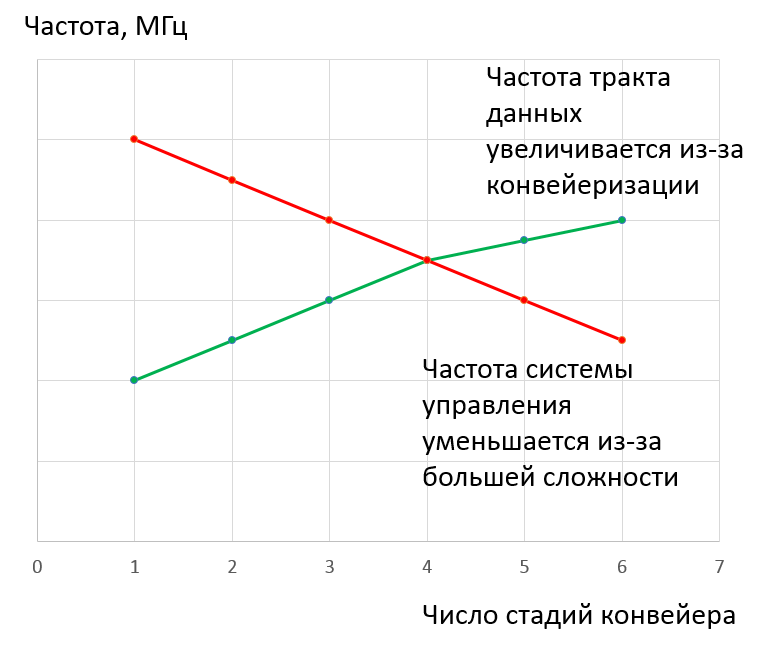
По мере того, как происходило распространение RISC-архитектур, 3- и 5- ступенчатые конвейеры получили широкое распространение в качестве наиболее употребительных. Однако в цифровой схемотехнике известно, что конвейеризация позволяет разбить критические пути распространения сигнала на отдельные стадии и тем самым увеличить тактовую частоту. Такой подход увеличивает латентность, и в определенный момент все же заставит вводить такты ожидания, если будет иметь место существенная зависимость по данным. Однако для целого ряда приложений, например, мультимедийных, или, в более широком смысле, задач цифровой обработки сигналов, можно использовать сильно конвейеризованные схемы для обработки потока входных значений.

Увеличение количества стадий конвейера можно было наблюдать в процессорах Intel Pentium. Если процессор P5 (Pentium первого поколения) имел 5 стадий, то Pentium III использовал уже 10, а Pentium 4 – 20 стадий (по некоторым данным, ядро Prescott имело 31-ступенчатый конвейер). Можно отметить, что тактовые частоты Pentium 4 при этом достигали 3-4 ГГц, что сопровождалось и ростом потребляемой мощности – до 120 Вт. В конечном итоге подход, использованный в Pentium, был заменен в процессорах Intel Core Solo/ Core Duo на более короткий конвейер, что снизило тактовую частоту, однако итоговая производительность оказалась выше.

Недостатком сильно конвейеризованной архитектуры является чрезмерное возрастание сложности контроля зависимости по данным. В длинном конвейере перед чтением данных из регистра необходимо проверять, что ни одна из стадий конвейера не занимается выполнением операции, которая при своем завершении обновит этот регистр. Подход data bypass имеет здесь ограниченное применение, поскольку может быть необходимым передача данных не просто на предыдущую стадию, а на стадию, отстоящую достаточно далеко по конвейеру.

В конечном итоге, конвейеризация имеет двойственный эффект. С одной стороны, при конвейеризации тракта данных возрастает его тактовая частота (однако этот процесс рано или поздно войдет в насыщение, поскольку сложно обеспечить совершенно равномерное разбиение тракта данных на стадии). С другой стороны, добавление стадий конвейера усложняет проверки, поэтому частота системы управления, контролирующей возможность передачи данных на каждую из стадий, будет падать. Этот процесс условно показан на рис. 12.14, где нужно отметить, что этот рисунок не соответствует какой-то определенной архитектуре.

На рис. 12.14 можно видеть, что при чрезмерно агрессивной оптимизации частота тракта данных будет достаточно высокой, но при этом общая частота процессора станет ограничиваться уже сложностью управляющих систем, проверяющих конфликты из-за зависимостей по данным на разных стадиях конвейера.



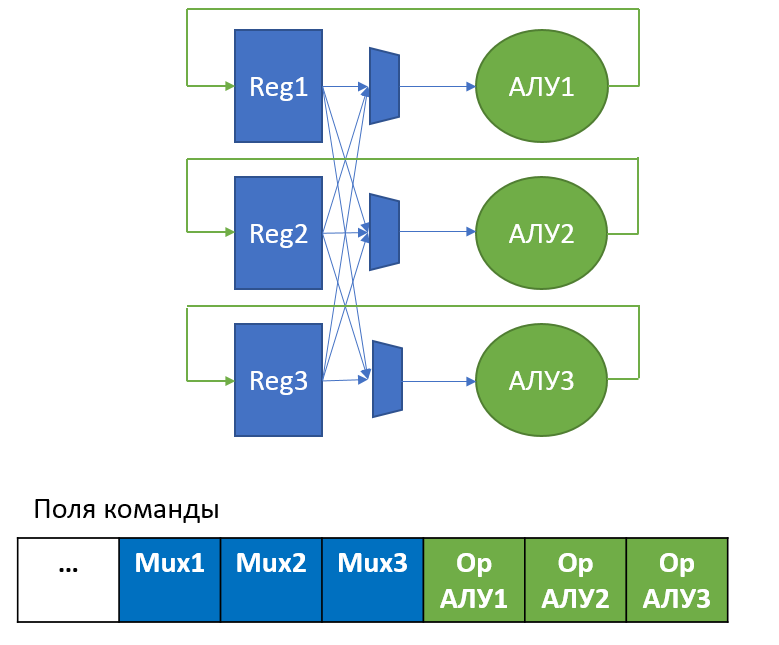
*Рис. 12.14. Пример (абстрактный) увеличения сложности управляющей схемы многоступенчатого конвейера*

12.6. Архитектура VLIW (сверхдлинное командное слово)

Сверхдлинное командное слово требуется, если в команде процессора оказывается много отдельных полей. Потребность в дополнительных полях может возникнуть, если за один такт будут параллельно выполняться несколько операций. Пример организации конвейера данных VLIW-процессора показан на рис. 12.15. На этом рисунке показано, что несколько регистровых файлов (Reg1 – Reg3) через коммутаторы подают операнды на несколько АЛУ (АЛУ1 – АЛУ3). Коммутаторы могут иметь разные варианты организации (в примере показано, что выходы любого регистрового файла могут быть поданы на входы любого АЛУ), поэтому в таком процессоре возможно параллельное выполнение трех разных команд в трех АЛУ. Подобный подход соответствует *параллелизму уровня инструкций* (ILP, Instruction Level Parallelism).

Размер команды на рис. 12.15 увеличивается, потому что каждому из АЛУ необходимо определить выполняемую на этом такте операцию, а каждому коммутатору – выбираемый им операнд. Однако это не означает, что размер команды пропорционален количеству АЛУ. Если предположить, что команда имеет 32 разряда, добавление АЛУ потребует 4-5 разрядов для выбора операции, а дополнительный мультиплексор потребует 2-3 разряда (в зависимости от количества его входов).

Размер команды для VLIW-процессоров указывается в пределах 128 - 256 разрядов. Рекордсменом является процессор Avispa+ компании Silicon Hive, имеющий 768-разрядную команду, однако при такой разрядности команды этот процессор имеет 60 слотов операций.



*Рис. 12.15. Принцип организации процессора с архитектурой сверхдлинного командного слова (VLIW)*

Преимуществом VLIW является, как можно понять, возможность выполнения нескольких операций за один такт. Однако это формальное преимущество не всегда можно использовать на практике. Для того, чтобы можно было параллельно выполнить несколько операций, для них должны быть готовы данные. Если типичные целевые алгоритмы предусматривают в основном линейные последовательности команд, в которых вычисленный результат используется как операнд следующей команды, дополнительные АЛУ, скорее всего, будут простаивать. Поэтому для VLIW большую роль играют предварительные исследования алгоритмов и возможности используемого компилятора.

Архитектура VLIW имеет разновидность, отмечающую тот факт, что в процессоре происходит прямое управление всеми вычислительными и коммутационными устройствами. Такая архитектура называется Explicit Parallel Instruction Computer (EPIC) – процессор с явным параллелизмом. Если для VLIW-процессоров часто использовались аппаратные планировщики, обеспечивавшие автоматический выбор операндов для АЛУ по мере их готовности, то в EPIC делается больший упор на возможности компилятора по прямому управлению всеми устройствами процессора. Как и для VLIW, сам факт применения архитектуры EPIC не позволяет достичь увеличения производительности, если особенности выполняемой программы не предполагают параллельных вычислений.

12.7. Прототипирование процессоров на базе ПЛИС.

Необходимо разделить проекты процессоров на прототипы будущих заказных микросхем (ASIC – Application-Specific Integrated Circuits) и на так называемые софт-процессоры – процессоры, собираемые из программируемых ячеек и изначально проектируемые для будущей эксплуатации в составе проектов в ПЛИС. Формально прототип процессора также является софт-процессором, поскольку он состоит из программируемых ресурсов ПЛИС. Однако смысл разделения заключается в том, что решения, эффективные для ПЛИС с учетом архитектуры этих микросхем, не обязательно будут эффективны для ASIC и наоборот. Поэтому важно понимать, будет ли проект в ПЛИС макетом будущей микросхемы или же самостоятельным изделием. В первом случае оптимизировать проект следует не с точки зрения САПР ПЛИС и архитектуры этих микросхем, а с точки зрения той технологической библиотеки, которая будет впоследствии использована при производстве процессора. В этом случае тактовая частота макета процессора в ПЛИС может быть невысокой.

Если ПЛИС предполагается как основная платформа для изделия, для практических целей можно ответить на следующие вопросы:

1. Предусматривает ли задача интенсивное использование несложных параллельных вычислений?

Очевидным применением являются приложения цифровой обработки сигналов, получающие существенные преимущества от независимых параллельно работающих блоков «умножение с накоплением» (компоненты DSP48 в современных FPGA Xilinx). В этом случае софт-процессор будет играть в системе вспомогательную роль и служить только для упрощения настройки цифровых фильтров, IP-ядер БПФ и прочих подобных устройств. Он может иметь весьма умеренную тактовую частоту (поскольку не является компонентом, определяющим производительность системы), однако должен обеспечивать поддержку внешних интерфейсов, осуществляющих доступ к подсистеме цифровой обработки сигналов со стороны оператора. Подобную роль в настоящее время успешно выполняют софт-процессоры MicroBlaze.

2. Имеются ли в системе независимые процессы, требующие обработки в режиме реального времени?

Данная задача является весьма неоднозначной и требующей тщательного внимания со стороны разработчиков. При попытке ее решения с помощью одного процессорного ядра (пусть даже и заявленного как real-time) перед программистом неминуемо возникает целый ряд вопросов. Каков реальный промежуток времени, проходящий между поступлением в процессор запроса на обработку и переходом к формированию выходного управляющего воздействия? Этот ответ существенно отличается от «сколько тактов процессор тратит на вход в прерывание», поскольку, например, срабатывание аварийного датчика требует отключения соответствующего силового компонента, и факт входа в прерывание еще не означает, что процессор уже послал сигнал отключения в соответствующий порт. Аппаратные решения способны обеспечить задержку реакции порядка десятков наносекунд, что существенно превосходит возможности программных решений (впрочем, подобные возможности часто избыточны).

Однако кроме обеспечения минимального времени реакции на внешние воздействия системы реального времени могут поставить и другую, более сложную задачу. Каким образом необходимо обрабатывать запросы, если подпрограммы их обработки перекрываются во времени? Наличие только одного потока исполнения команд вынуждает программиста устанавливать приоритеты в обработке запросов, причем добавление нового устройства или даже изменение основной программы требует проверять, не нарушается ли нормальная работа с внешним оборудованием, формирующим запросы на прерывания. Проблемы здесь вполне возможны, например, из-за применения библиотек или фрагментов кода, запрещающих прерывания в процессе выполнения каких-либо действий.

Простейшим решением со стороны софт-процессоров здесь является применение нескольких процессорных ядер, в том числе и специально выделенных для работы с критичными источниками запросов на обработку. Здесь можно говорить не только о датчиках аварийных состояний, но и, например, о большом количестве периферийных устройств, использующих медленные протоколы обмена данными. Чтобы не тратить ресурсы основного процессора на их реализацию, можно использовать не только аппаратные ускорители уровня конечных автоматов, но и несложные процессорные ядра, оптимизированные для простых операций с портами периферийного устройства.

Можно отметить, что популярные микроконтроллеры семейства PIC появились именно в подобном качестве. Аббревиатура PIC расшифровывается как Peripheral Interface Controller, т.е. «контроллер периферийных интерфейсов», и первый микроконтроллер PIC появился в качестве микросхемы для расширения возможностей ввода-вывода процессора CP1600. Даже ПЛИС начального уровня способны вместить несколько несложных процессоров, которые существенно облегчат работу программиста, освобождая его от необходимости «жонглировать» в основной программе несколькими независимыми процессами вычислений.

3. Имеются ли в задаче специфичные вычисления, последовательности команд или шинные циклы?

Положительные ответы на эти вопросы могут являться основанием для разработки собственного процессорного ядра. Действительно, если обратиться к первой части данного цикла статей, в ней имелась отсылка к статье, анализирующей микроконтроллеры начального уровня. Там упоминалось, что операции записи в выходной порт (микроконтроллерный hello world в виде мигания светодиода) требуют от 3 до 20 тактов в некоторых процессорных ядрах. Естественным решением было бы ускорение таких операций не только путем добавления соответствующих периферийных IP-ядер, но и путем введения в процессор специализированных команд, например, для ускорения доступа к периферии, для выполнения автоинкремента/декремента, групповых арифметических и логических операций и т.п. При этом умеренная тактовая частота процессора будет не так важна, если процессор будет выполнять значимые для пользователя фрагменты кода за меньшее число тактов.

12.8. Выводы по разделу

Разработка процессорного ядра имеет следующие основные этапы.

1. Проектирование и моделирование.

Данный этап не потребует существенных материальных и временных затрат. Можно использовать практически любой язык программирования высокого уровня для работы с программными моделями процессора и бесплатные версии САПР ПЛИС для экспериментов с RTL-описанием и моделирования на уровне регистровых передач. В итоге разработчик сможет выявить работоспособность принятых им технических решений, оценить тактовую частоту и объем ресурсов, которые будут заняты в ПЛИС.

2. Реализация в макете.

Простые отладочные платы на базе ПЛИС начального уровня вполне позволяют продемонстрировать практическую работоспособность процессора и создать примеры устройств, управляющих периферийным оборудованием. При умеренных финансовых затратах можно будет продемонстрировать практическое функционирование процессорной системы.

3. Внедрение в практику.

Наличие информации о достижимых характеристика процессора и возможных путях его доработки, расширения и адаптации позволяет говорить о его внедрении в практику, в первую очередь, по месту основной работы в профильной организации. Как было упомянуто, важное практическое преимущество оригинальной архитектуры заключается в том, что она может быть быстро адаптирована для управления сложными IP-ядрами, расположенными на кристалле ПЛИС. Без удобного в применении процессора отладка таких ядер производилась бы с помощью длительных итераций моделирования, правки управляющих непрограммируемых контроллеров и генерации новых конфигурационных файлов. Управление сигналами IP-ядер со стороны встроенного в проект процессора, по меньшей мере, сократит количество итераций создания конфигурационных файлов.

Контрольные вопросы:

1. Сколько регистров требуется процессору? В чем преимущества и недостатки большого и маленького регистрового файла?

2. Почему во многих процессорах регистры имеют дополнительные функции?

3. Что происходит с конвейером процессора при переходе к новому адресу?

4. Что такое зависимость по данным? В чем заключается негативный эффект от ситуации, когда следующая команда читает регистр, который записывается предыдущей командой?

5. Что происходит с конструкцией процессора при увеличении количества тактов конвейера? Как это отражается на выполнении программ?

6. В архитектуре со сверхшироким командным словом используется несколько АЛУ. Какие препятствия могут возникнуть, если разработчик ожидает пропорционального роста производительности такого процессора?