Проектирование процессорных ядер. Часть 2. Программная модель и микроархитектура

В статье рассматриваются вопросы практического проектирования конвейеризованных процессорных ядер. С учетом большого разнообразия доступных процессоров и развития компонентной базы программируемых логических интегральных схем появляется возможность не только изучить подходы к проектированию процессоров, но и применить софт-процессор в практическом проекте на базе ПЛИС. Подразумевается использование языка описания аппаратуры VHDL и ПЛИС Xilinx.

Илья ТАРАСОВ,
д. т. н.
ilya_e_tarasov@mail.ru

Введение

В предыдущей статье [1] рассмотрен порядок проектирования конечного автомата, управляемого потоком команд. Фактически такой автомат, изменяющий состояние нескольких регистров по фронту тактового сигнала под управлением данных, поступающих из специальной памяти, является процессором. Изменяя содержимое памяти программ, можно управлять последовательностью операций, производимых над содержимым регистров. Однако представленный простейший пример, демонстрируя основные операции, характерные для процессора, все же имел множество недостатков с практической точки зрения.

Регистровая модель процессора

Регистровая модель процессора описывает процессор в том виде, в каком он представляет интерес для программиста. Язык ассемблера предусматривает описание команд в виде тех действий, которые выполняются с регистрами процессора и в целом могут восприниматься как переменные программы. Кроме регистров, хранящих обрабатываемые данные, есть регистры, имеющие специальное назначение.

К обязательным регистрам относится счетчик команд, традиционно именуемый PC (Program Counter), однако для процессоров Intel он называется IP (Instruction Pointer). С точки зрения схемотехники такой 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, где находятся адреса, выступает в качестве смещения относительно некоторого базового адреса. Такой подход требуется потому, что 16-разрядный регистр способен адресовать только 2^{16} = 65 536 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, так же 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, применяется совместно с указателем стека SP;
- ES (extra segment) — дополнительный сегмент, предусмотренный в некоторых командах доступа к памяти.

В процессоре i80386 добавлены 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 стоит рассматривать как практический прием, преследующий те же цели (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.

Другой практически обязательный регистр — указатель стека, так как вызов подпрограмм требует запоминания адреса возврата. Для большинства процессоров он обозначается как SP (Stack Pointer). 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, то есть «последним зашел — первым вышел»). Стек удобнее всего представлять в виде стопки листов бумаги, где последний листок, положенный сверху, 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. Работу стека иллюстрирует рис. 1. 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 находи-

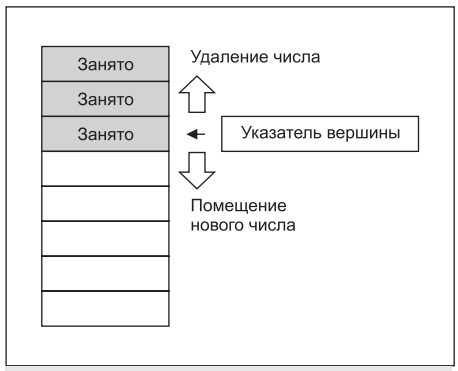

Рис. 1. Организация стека в памяти с помощью указателя вершины

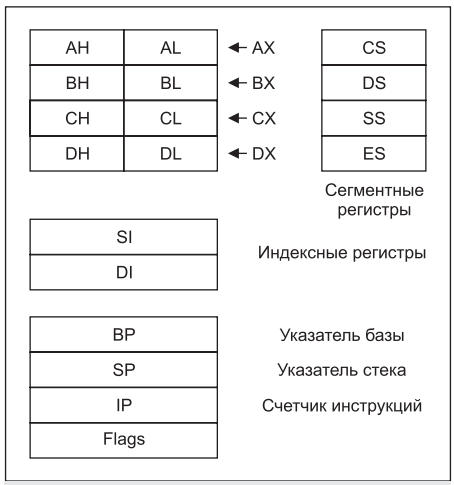

Рис. 2. Регистровая модель 16-разрядного процессора i8086

лась в наибольшем («старшем») адресе памяти и стек рос «навстречу» данным.

Кроме адресов возврата, на стеке обычно хранят локальные параметры, передаваемые подпрограммам. Для этого удобно, чтобы разрядность адреса и данных совпадала, однако для i8086 приходится помещать адрес на стек в виде двух чисел — сегментного регистра и смещения. Вопросы организации вызовов подпрограмм — предмет отдельного анализа при проектировании архитектуры. Здесь будет важно понять, в какой памяти располагается стек, можно ли поместить адрес для возврата одной командой процессора, используется ли сегментированная адресация и т. п.

Особым приемом, применимым для FPGA, является размещение стека в отдельном компоненте накристальной памяти. Это возможно, поскольку блоки памяти в FPGA независимы и есть смысл эффективнее использовать высокую пропускную способность подсистемы накристальной памяти, выполняя за один такт несколько операций.

Основные операции с данными осуществляются 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 предлагаются символы L (Low, младшая часть) и H (High, старшая часть). Регистровая модель процессора i8086 показана на рис. 2, где 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: название AX соответствует 16-разрядному регистру, а AH и AL — его половинам.

Другой важный регистр, связанный с обработкой данных, — 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 0.
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 возможна 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, так как в дополнительном двоичном коде старший разряд свидетельствует о знаке числа. Копирование этого разряда в отдельный регистр флага позволяет применять команды условного перехода, выполняемые по результату проверки флага. Это быстрее, чем проверять старший разряд числа отдельной командой.
5. **Флаг переполнения (Overflow Flag, OF).** Это вспомогательный флаг 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 — предмет отдельного рассмотрения. Например, к общей практике следует отнести то, что команды обычной загрузки в регистры не влияют на флаги, то есть пересылка в регистр нулевого значения из другого регистра не устанавливает флаг нуля. Однако часто флаг переноса устанавливается в 0 при выполнении поразрядных логических команд и отражает тот факт, что результат целиком поместился в регистр назначения.

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

На рис. 3, 4 показаны некоторые регистровые модели процессоров. На рис. 3 приведены регистровые модели процессоров i8080 (Intel), Z80 (Zilog) и MC6800 (Motorola). Все эти процессоры относятся к середине 1970-х годов и достаточно показательны для изучения. Процессор i8080 (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 команды, которые не могли быть выполнены с другими регистрами общего назначения. Остальные регистры могли объединяться в 16-разрядные пары BC, DE и HL, предназначенные 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 достаточно важно при планировании совместной работы аппаратной и программной компонентов системы.

Процессор Z80 создан в 1976 году компанией Zilog в качестве некоего ответа на проект i8080. Сохраняя обратную совместимость по машинному коду (выполняя все программы для i8080), процессор Z80 имел и вспомогательные индексные регистры IX и IY (игравшие в целом ту же роль, что и регистровая пара HL), а также имел теневой набор регистров, показанный на рис. 3 в виде блока регистров со штрихами. В конкретный момент времени был активен только один из наборов, а быстрое 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

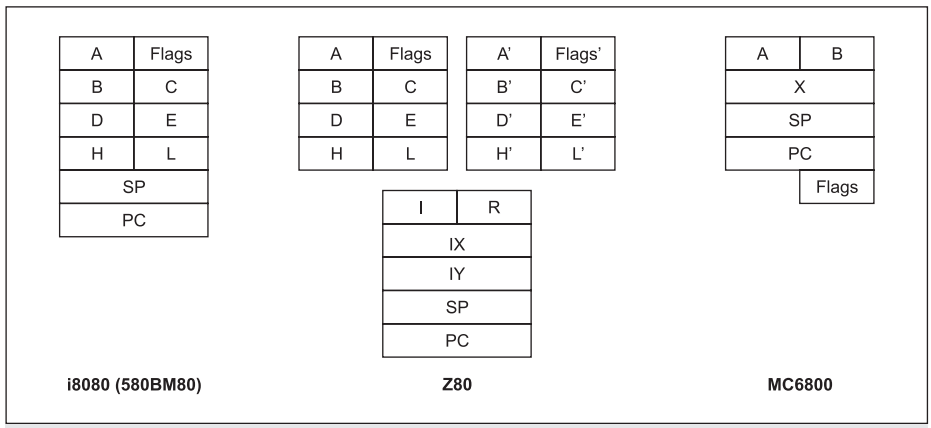

Рис. 3. Регистровые модели процессоров i8080, Z80 и MC6800

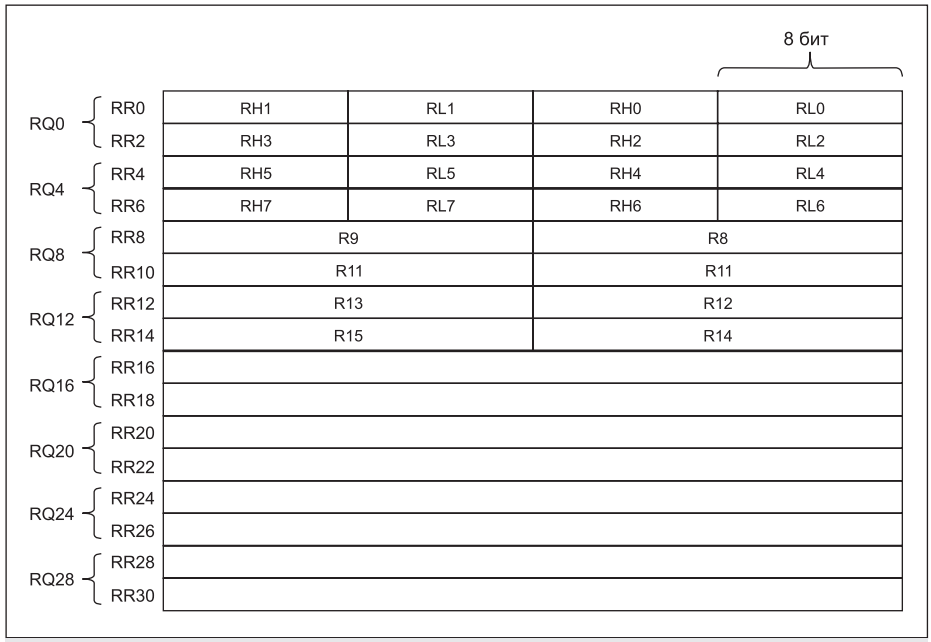

Рис. 4. Регистры общего назначения процессора Z80000 (1986 г.)

момент. Процессор Z80 оказал заметное влияние на российскую школу цифровой техники, поскольку стал основой компьютера Sinclair ZX-Spectrum, популярного в России в начале 1990-х и самостоятельно собиравшегося 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.

Для сравнения можно показать регистры общего назначения 32-разрядного процессора Zilog Z80000 (рис. 4). Такой подход расширяет возможности программиста по выбору регистра, в котором будут размещены те или иные программные объекты. На рисунке видна некоторая неоднородность, поскольку старшая половина регистров является строго 32-разрядной (RR16–RR31), регистры R8–R15 могут использоваться как 16-разрядные или 32-разрядная пара, а регистры R0–R7 дополнительно разделены на 8-разрядные части.

В архитектуре процессоров MCS-96 компании Intel регистровый файл состоял из 256 регистров общего назначения (располагавшихся в памяти). Тем не менее в команде можно было указать 8 бит, соответствующие номеру регистра.

Количество регистров, их разрядность и доступные команды являются важными составными частями проекта процессорной архитектуры.

Проектирование системы команд процессора

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

Следует также упомянуть несколько важных свойств системы команд. Первое — ортогональность, тесно связанная с регистровой моделью процессора. Под ортогональностью понимается наличие множества методов адресации данных, которые могут использоваться с любым сочетанием регистров процессора. Термин ведет происхождение от определения ортогональных (непересекающихся) векторов в геометрии. Под ортогональной системой координат, основанных на таких векторах, понимают некоторое пространство проектирования, где одно действие не оказывает влияния 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. Нужно отметить, что в ряде случаев введение частичной специализации позволяет упростить аппаратную часть процессора и сократить разрядность команд. Примером специально введенной неортогональности служит 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 применять ортогональную систему команд, является результатом тщательных многоплановых исследований. В первую очередь играет роль программная модель вычислений и типичные задачи, которые будут решаться с помощью разрабатываемого процессора. Показательный пример — расширение Jazelle, оно, как и Thumb, является разновидностью 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 [3, 4].

Другая важная практическая характеристика — адресность команд. Под адресностью понимается количество адресов (индексов регистров процессора), участвующих в выполнении команды. «Адрес» можно трактовать в широком смысле, как любое указание на ресурс процессора, участвующий в вычислениях или получающий результат.

Примером трехадресной команды служит 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, где будут помещены соответствующие номера. Размер полей зависит от количества регистров, которые можно адресовать таким способом. Так, 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. Скажем, рассматривая команду ассемблера **add ax, bx**, в ее описании можно увидеть, что действие команды — это сложение данных из регистров **ax** и **bx** и помещение результата в **ax**. Таким образом, первый операнд команды одновременно становится и получателем результата.

Одноадресные команды используются в ряде процессоров цифровой обработки сигналов (например, Texas Instruments TMS320). В данных процессорах часто применяется 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, будет использование единственно возможного сочетания регистров для выполнения всех операций.

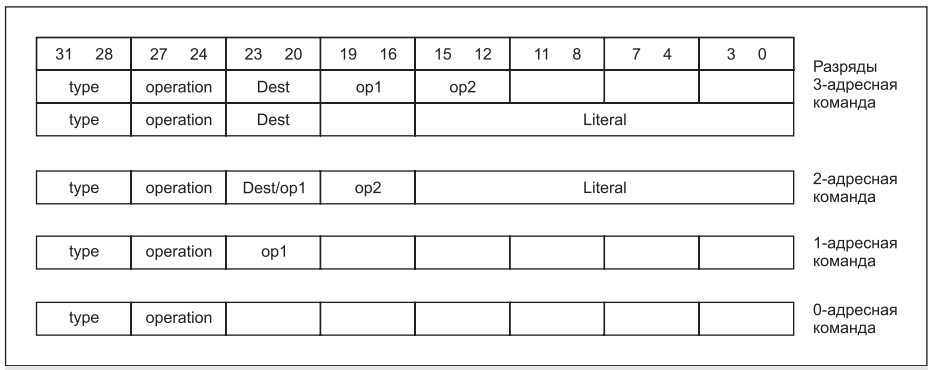

Рис. 5. Пример формата команд для процессоров с различной адресностью

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 служит виртуальная стековая машина Java, аналогичный подход использован в Common Intermediate Language виртуальной машины *.net* корпорации Microsoft [5]. Особенность стековых вычислений заключается в их независимости от конкретной регистровой архитектуры. Нетрудно представить, что машинный код, использующий только 4 регистра, окажется недостаточно эффективным для процессора, имеющего 16 или 32 регистра. 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 значительно ускорить выполнение стековых команд.

В стековом процессоре имеется отдельный стек данных, куда не помещаются адреса возврата из подпрограмм (для этого предназначен 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 применены другими подпрограммами.

Исторически стековая модель вычислений реализована на языке программирования Форт (Forth), который был удачно описан в [6]. Кроме простой вычислительной модели, язык отличается доступной 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.

Команды с адресностью больше, чем 3, также могут быть реализованы. В данном случае имеются в виду процессоры с несколькими командами, выполняемыми параллельно, так называемая архитектура VLIW (Very Long Instruction Word). Несколько осуществляемых команд требуют и нескольких наборов операндов, поэтому адресность может быть равна 4, 6 и более. В настоящее время архитектура VLIW получила развитие в виде EPIC ((Explicitly Parallel Instruction Computing — архитектуры с явно заданным параллелизмом) и реализуется в ПЛИС вследствие возможности описания памяти с разрядностью 64, 128 и выше.

Не вполне очевидным вариантом 4-адресной команды является задание, кроме индексов операндов и получателя, адреса следующей команды. Можно рассматривать такую команду в качестве параллельно выполняющихся инструкций, одна из которых представляет собой обычную математическую операцию, а вторая — команду перехода. Обоснованность такого подхода определяется, прежде всего, интенсивностью использования 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. 5. 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. На рис. 6 представлены соответствующие адресности регистровые модели.

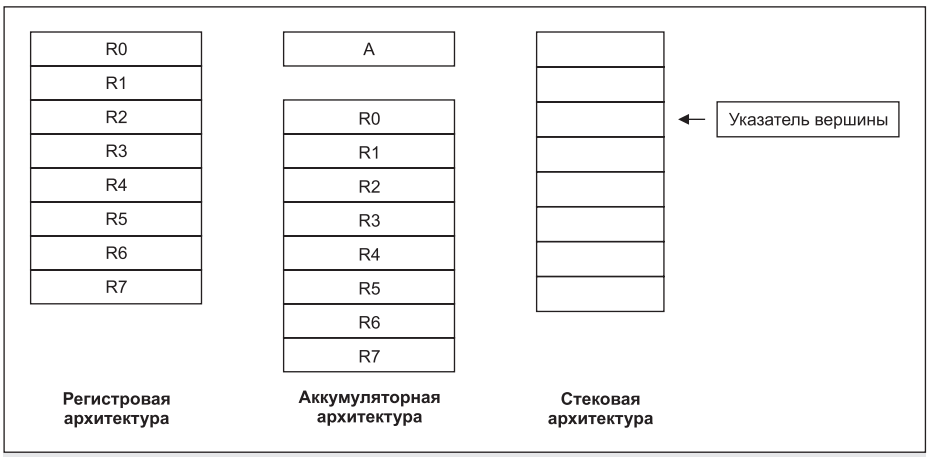

Рис. 6. Регистровая, аккумуляторная и стековая архитектуры, соответствующие системам команд с различной адресностью

Поля команды на рис. 5 изображены достаточно условно, однако они имеют и практическое применение. Для определения того, что именно следует закодировать, необходимо рассмотреть группы команд процессоров.

Группы команд процессорного ядра

В целом процессоры имеют весьма разнообразные наборы команд, чей состав зависит от назначения процессора и ограничен его программной моделью. Тем не менее следует выделить крупные группы команд, часто присутствующие в процессорах.

Например, команды арифметики, сдвига и манипуляций с битовыми полями объединяет то, что они работают с данными и модифицируют регистры или память. Поэтому осуществление еще одной команды подобного типа достаточно просто. 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 устройства;
- 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 и организацию работы со стеком.

Для указания на данные предусмотрены различные методы адресации. Список методов не является обязательным или исчерпывающим, однако процессор, который не может 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 типа **ax** или **r0**, причем номер регистра явно 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 именуют литералом (literal — «буквальный»). Так, если команда загрузки в регистр R0 имеет код 123, 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. Данный 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, чей адрес 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. В частности, 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**, а потому 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 предназначены для управления порядком выполнения программы и 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, существует и 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, то есть перемещение фрагмента программного кода по другому абсолютному адресу в памяти не нарушит относительное расположение команд, и переходы с относительной адресацией будут по-прежнему выполняться правильно;
- для относительных переходов часто используется формат записи смещения в пределах 1 или 2 байт, 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 добавление команды условного перехода с проверкой этого флага (тем более для обоих вариантов значения флага). Перечисленные выше флаги, устанавливаемые по результатам выполнения команды в АЛУ, наиболее употребительные.

Команда вызова подпрограммы применяет 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. Так, 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.

Команды арифметико-логического устройства описаны далее при анализе проектирования этого устройства на VHDL.

Проектирование основных компонентов процессора

Проанализируем порядок проектирования основных компонентов процессора, которые можно было бы использовать для построения вариантов микроархитектуры.

Регистровый файл — это группа регистров, имеющих сходное назначение. Для регистрового файла обычно предусмотрен 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 указанных 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 нужен отдельный сигнал разрешения записи данных.

Для получения операндов следует 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. 7. Описание регистрового файла на языке VHDL приведено в листинге 1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity regfile is
   Port ( clk : in STD_LOGIC;
we : in STD_LOGIC;
aw : in STD_LOGIC_VECTOR (3 downto 0);
dw : in STD_LOGIC_VECTOR (31 downto 0);
addra : in STD_LOGIC_VECTOR (3 downto 0);
addrb : in STD_LOGIC_VECTOR (3 downto 0);
opa : out STD_LOGIC_VECTOR (31 downto 0);
opb : out STD_LOGIC_VECTOR (31 downto 0)
);
end regfile;

architecture Behavioral of regfile is

type TRegs is array(0 to 15) of std_logic_vector(31 downto 0);
signal Regs : TRegs := (others => (others => '0'));

begin

process(clk)
begin
 if rising_edge(clk) then
if we = '1' then Regs(to_integer(unsigned(aw))) <= dw; end if;
 end if;
end process;

opa <= Regs(to_integer(unsigned(addra)));
opb <= Regs(to_integer(unsigned(addrb)));

end Behavioral;
```

Листинг 1. Описание регистрового файла

В листинге 1 используется тип TRegs, описывающий массив из 16 регистров, каждый из которых имеет разрядность 32 бит. Легко заметить, что как количество регистров, так и разрядность могут быть изменены. Таким образом, увеличение разрядности процессора не представляет сложности с точки зрения инструментов описания схем. Размер регистрового файла в 16 регистров выбран для того, чтобы 4-разрядный номер регистра мог быть представлен одним шестнадцатеричным символом, это упрощает чтение машинного кода программы в простых примерах.

Выходы регистрового файла читаются асинхронно. Соответственно, после обновления содержимого какого-либо регистра его значение будет доступно на выходе (с учетом задержки распространения сигнала).

В описании всем регистрам было задано нулевое начальное значение. Это сделано в первую очередь в целях обеспечения нормального моделирования на функциональном уровне. Можно также убедиться, что модуль регистрового файла не имеет отдельного сигнала сброса (reset). Сигнал сброса необходим при проектировании компонентов СБИС, однако не требуется для FPGA, где начальное состояние всех триггеров проекта инициализируется в явном виде при загрузке конфигурации. Введение сигнала сброса всего массива регистров ограничит возможности реализации регистров только триггерами логических ячеек ПЛИС, которые имеют такой аппаратный сигнал. В приведенном же варианте остается возможность выбора ресурсов ПЛИС, в частности, показанный модуль можно синтезировать на базе распределенной памяти LUT, что дает более компактную реализацию по сравнению с триггерами.

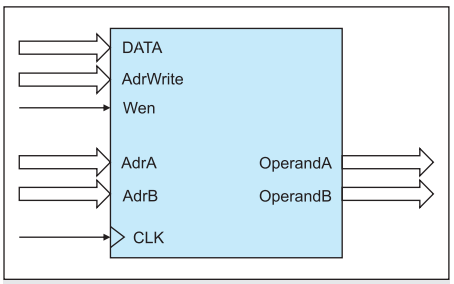

Рис. 7. Графическое изображение модуля регистрового файла

Описание регистрового файла модифицируется не только путем изменения разрядности и количества регистров. Для архитектур с большой адресностью можно использовать регистровый файл с тремя и более выходами, а также с несколькими входами для записи данных.

Если добавление выходного порта не представляет принципиальной сложности, то при добавлении еще одного входа возникает проблема электрического конфликта входных сигналов. При попытке просто продублировать код процесса записи в регистр синтезатор сформирует схему, способную подключить один регистр к двум входам данных сразу. Поэтому нужно предусмотреть проверку совпадения адресов регистров и выполнить запись по второму порту лишь в том случае, если она производится в несовпадающий регистр. Пример приведен в листинге 2.

```
process(clk)
begin
 if rising_edge(clk) then
  if wea = '1' then Regs(to_integer(unsigned(awa))) <= dwa; end if;
  if web = '1' and (awb /= awa) then Regs(to_integer(unsigned(awb)))
<= dwb; end if;
 end if;
end process;
```

Листинг 2. Реализация двупортового интерфейса записи в регистровый файл

Можно заметить, что синтезированная схема занимает существенно больший размер в ячейках ПЛИС по сравнению с однопортовой реализацией. Кроме громоздкой схемы проверки совпадений, регистры оказались реализованы не на базе распределенной памяти логических ячеек (LUTRAM),

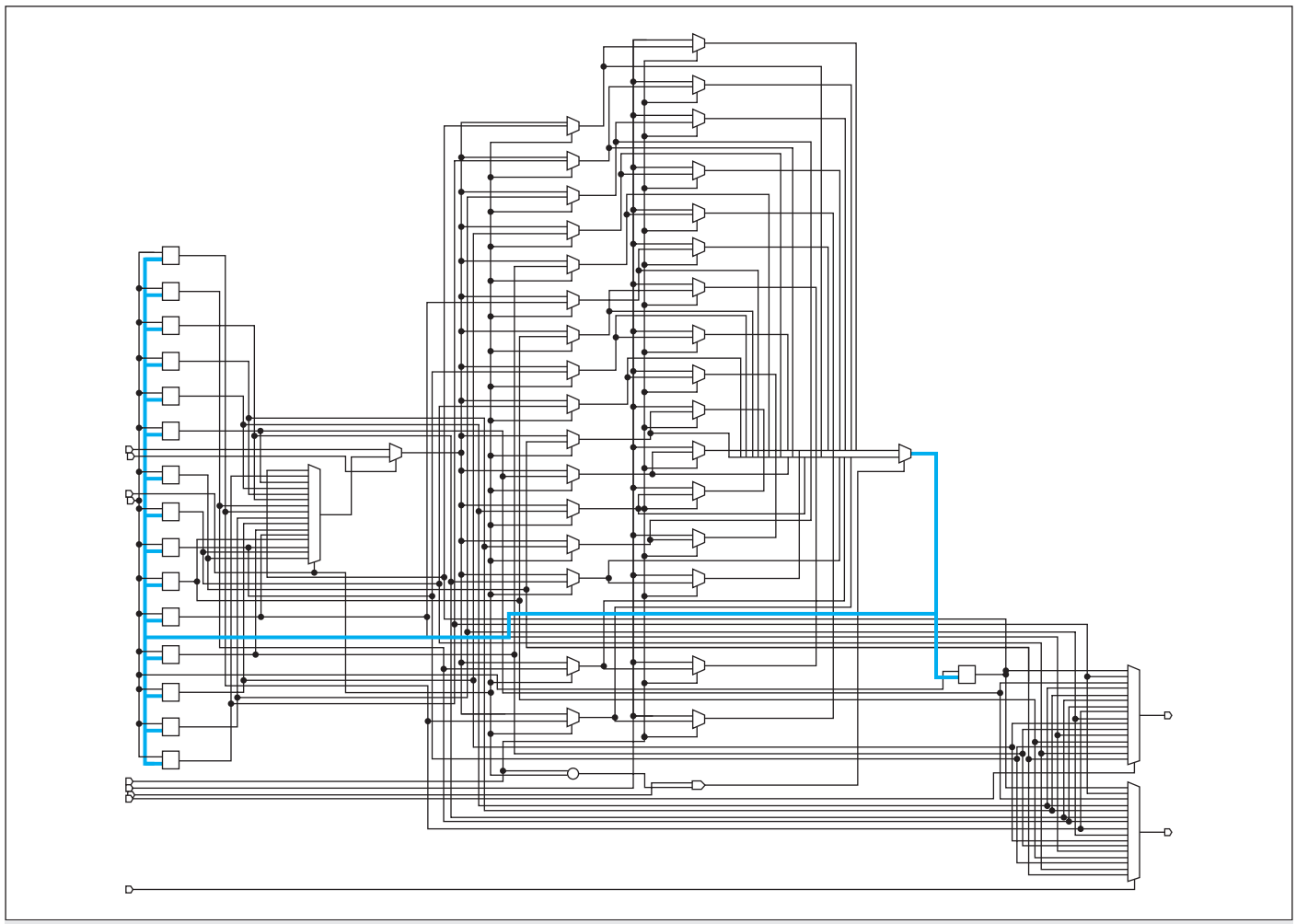

Рис. 8. Синтезированная схема регистрового файла с двумя портами записи

а на триггерах. Поэтому применение такого подхода оправдано в случаях, когда выполняемые программы интенсивно используют запись в регистры, что и подтверждается реализованными моделями. Синтезированная схема показана на рис. 8. Синим цветом выделена шина данных, полученная после разрешения конфликтов по записи, так что при совпадении адресов данные второго порта игнорируются.

Теоретически для регистрового файла возможно описание и более сложных схем с большим количеством портов записи и чтения. Видно, что при увеличении количества интерфейсов сложность схемы растет, и это должно быть обосновано соответствующим выигрышем от уменьшения количества тактов процессора на выполнение тех алгоритмов, на которые данный процессор ориентирован.

Альтернативным вариантом является применение подхода VLIW/EPIC с несколькими независимыми регистровыми файлами, каждый из которых имеет собственный интерфейс записи. Выходы таких регистровых файлов могут быть объединены общим мультиплексором, и каждый операнд получен из любого регистрового файла. Однако наличие нескольких регистровых файлов не позволяет произвести две и более записи в регистры, относящиеся к одному блоку.

Арифметико-логическое устройство (АЛУ) — основной вычислительный узел процессора, выполняющий операции над данными. Упрощенное представление интерфейса АЛУ показано на рис. 9. Видно, что АЛУ производит операции над входными операндами (получаемыми из регистрового файла или памяти данных), основываясь на коде выполняемой команды. Соответственно, реализация АЛУ достаточно проста. Ее пример показан в листинге 3, а результат синтеза — на рис. 10. Видно, что АЛУ представляет собой простой мультиплексор, управляемый кодом команды (по крайней мере, частью этого кода). В примере листинга 3 тип выполняемой операции задается в разрядах (27–24) кода команды.

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 следует ориентироваться на возможности языка описания аппаратуры, поскольку в нем уже представлены 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.

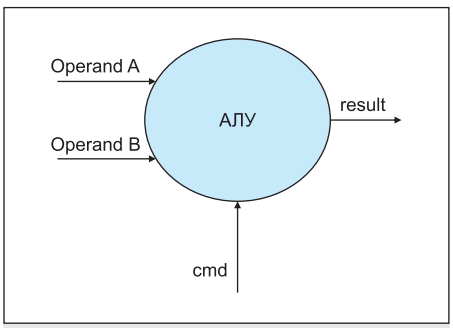

Рис. 9. Упрощенное представление АЛУ

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity alu is
   Port ( opa : in STD_LOGIC_VECTOR (31 downto 0);
opb : in STD_LOGIC_VECTOR (31 downto 0);
din : in STD_LOGIC_VECTOR (31 downto 0);
cmd : in STD_LOGIC_VECTOR (31 downto 0);
result : out STD_LOGIC_VECTOR (31 downto 0));
end alu;

architecture Behavioral of alu is

signal add, sub, mult : integer;

begin
-- описание команд, требующих данных типа integer
add <= to_integer(unsigned(opa)) + to_integer(unsigned(opb));
sub <= to_integer(unsigned(opa)) - to_integer(unsigned(opb));
mult <= to_integer(unsigned(opa)) * to_integer(unsigned(opb));

with cmd(27 downto 24) select
result <= opa when “0000”,
std_logic_vector(to_unsigned(add, 32)) when “0001”,
std_logic_vector(to_unsigned(sub, 32)) when “0010”,
opa and opb when “0011”,
opa or  opb when “0100”,
opa xor opb when “0101”,
opa(30 downto 0) & ‘0’ when “0111”,
‘0’ & opa(31 downto 1) when “1000”,
opa(31) & opa(31 downto 1) when “1001”,
not(opa) when “1010”,
din when “1011”,
opa(31 downto 16) & cmd(15 downto 0) when “1100”,
cmd(15 downto 0) & opa(31 downto 16)  when “1101”,
std_logic_vector(to_unsigned(mult, 32)) when “1111”,
(others => ‘0’) when others;

end Behavioral;
```

Листинг 3. Пример реализации арифметико-логического устройства

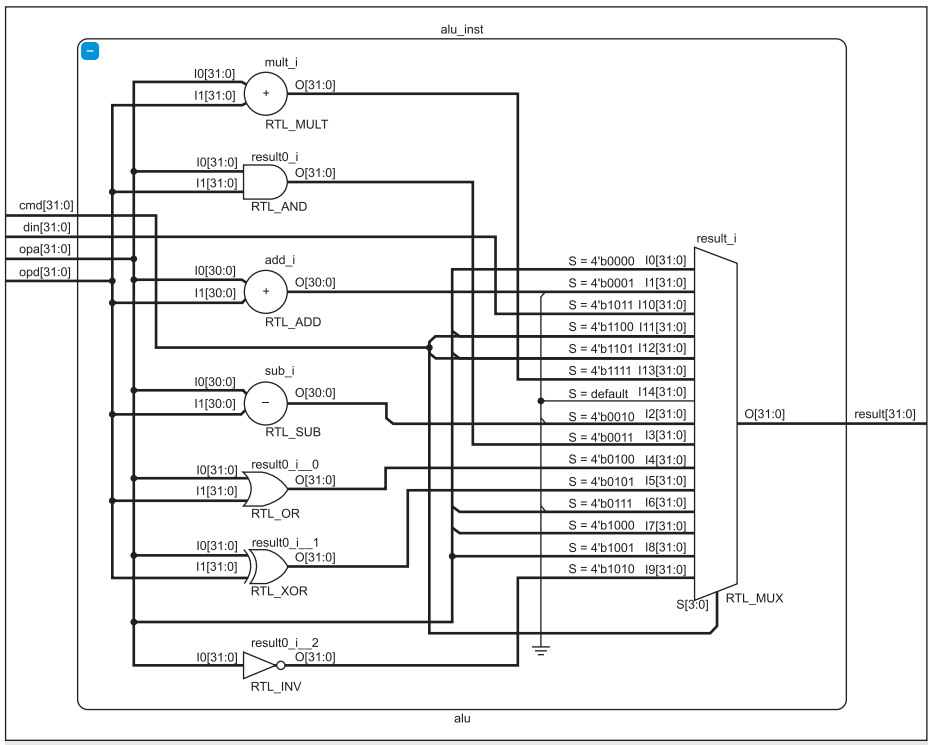

Рис. 10. Результат синтеза проекта АЛУ в САПР Vivado

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 станет 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 должно быть реализовано отдельно.

Рассмотрим 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 применяется 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).

Умножение с помощью оператора op1*op2 поддерживается языком VHDL и синтезаторами. Важно учитывать, какое именно представление чисел используется. При подключении библиотеки *numeric.std* возможно указание обоих вариантов:

```
mult <= to_integer(unsigned(opa)) * to_integer(unsigned(opb));
mult <= to_integer(signed(opa)) * to_integer(signed(opb));
```

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 перемножает числа большой разрядности.

Для прототипирования СБИС необходимо учитывать, что умножитель представляет собой компонент со сложной структурой (наиболее простой подход, так называемое 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 компонентом с гарантированными характеристиками, поэтому могут использоваться свободно.

Операция деления в общем случае не имеет простой реализации для произвольных операндов. Как правило, данная 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 имеют поразрядные (битовые) представления. А значит, 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, то есть поразрядное И-НЕ и т. д., однако вопрос реализации полного набора поразрядных операций остается предметом изучения в каждом конкретном проекте. Частое применение 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.

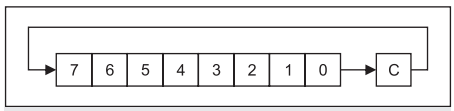

Рис. 11. Иллюстрация к операции вращения через бит переноса

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, то есть 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, хотя 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 разряды и бит переноса. Порядок вращения проиллюстрирован на рис. 11.

Операции сдвига и вращения могут быть реализованы на VHDL с помощью оператора сцепления &. Например: Result <= ‘0’ & opa (31 downto 1) описывает логический сдвиг на 1 разряд вправо.

Команды сравнения используются для формирования логических (булевых) условий. Они могут быть как реализованы в АЛУ в виде отдельных операций, так и выступать в качестве побочных эффектов при выполнении арифметико-логических 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.

Тем не менее операции сравнения могут быть реализованы и в явном виде:

```
Result <= x“FFFFFFFF” when opa = opb else x“00000000”.
```

Данное выражение дает в результате число с разрядами, установленными в 1, если входные операнды были равны, и 0, если они не равны.

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_0000 превратится в 0000_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.

Микроархитектура

Под микроархитектурой понимается структура связей между отдельными компонентами процессорного ядра и их взаимодействие.

В первой части цикла статей рассмотрена простая микроархитектура, работающая по двухтактному циклу «выборка – исполнение». На первом такте работы команда выбиралась из памяти, то есть происходил переход **PC → CMD**. На втором такте, зная код команды, можно было выполнить соответствующие ей действия, включая запись нового значения в регистр **PC**.

Такой подход отличается единообразием и возможностью простого расширения системы команд. Однако очевидно, что в то время, когда команда выполняется, память программ, по сути, простаивает, ожидая вычисления нового значения **PC**. Если программа осуществляется линейно, можно ожидать, что будет выполнено **PC <= PC**+1, то есть следующая команда находится сразу за текущей. Попытаемся реализовать такой процессор, в котором команды выполня-

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cpu_ctrl is
   Port ( clk : in STD_LOGIC;
reset : in STD_LOGIC;
intr : in STD_LOGIC;
cmd : in STD_LOGIC_VECTOR (31 downto 0);
opa : in STD_LOGIC_VECTOR (31 downto 0);
we_reg : out STD_LOGIC;
we_bus : out STD_LOGIC;
pc_out : out STD_LOGIC_VECTOR (15 downto 0));
end cpu_ctrl;

architecture Behavioral of cpu_ctrl is

signal pc : integer := 0;

-- signal st : integer range 0 to 1;

begin

process(clk)
begin
 if rising_edge(clk) then
   if reset = ‘1’ then pc <= 0; -- st <= 0;
    elsif cmd(31 downto 28) = “0011” then
pc <= to_integer(unsigned(cmd(15 downto 0))); -- st <= 0;
    elsif cmd(31 downto 28) = “0100” and opa = x“00000000” then
pc <= to_integer(unsigned(cmd(15 downto 0))); -- st <= 0;
    else pc <= pc + 1; -- st <= 1;
   end if;
  end if;
end process;

pc_out <= std_logic_vector(to_unsigned(pc, 16));

we_reg <= ‘1’ when cmd(31 downto 28) = “0001” and st = 1 else ‘0’;
we_bus <= ‘1’ when cmd(31 downto 28) = “0010” and st = 1 else ‘0’;
```

Листинг 4. Реализация устройства управления конвейеризованным процессором

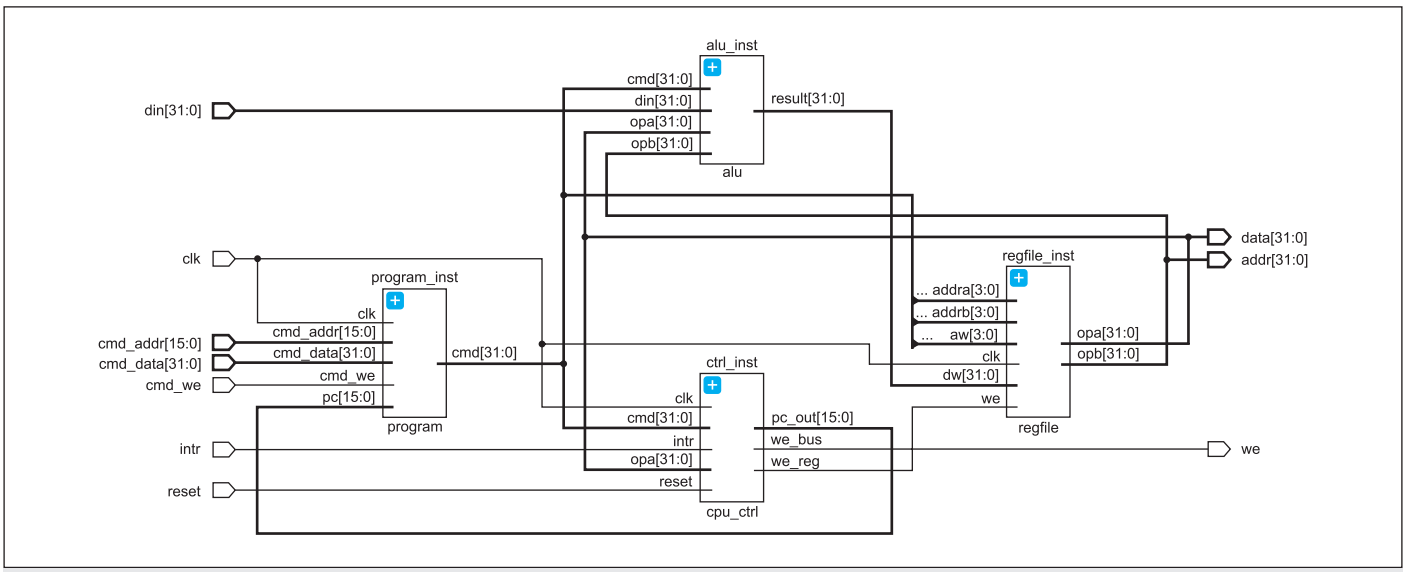

Рис. 12. Схема конвейеризованного процессора в САПР Vivado

ются на каждом такте, а значение счетчика команд вычисляются одновременно с ними. Это формирует конвейер, в котором параллельно с выполнением команды из памяти программ читается следующая команда. Пример устройства управления приведен в листинге 4. Жирным шрифтом выделены закомментированные строки, относящиеся к состоянию процессора **st**, которое понадобится чуть позже.

```
entity program is
   Port ( clk : in STD_LOGIC;
pc : in STD_LOGIC_VECTOR (15 downto 0);
cmd : out STD_LOGIC_VECTOR (31 downto 0);
cmd_addr : in STD_LOGIC_VECTOR (15 downto 0);
cmd_data : in STD_LOGIC_VECTOR (31 downto 0);
cmd_we : in STD_LOGIC);
end program;

architecture Behavioral of program is

type TProgram is array(16383 downto 0) of std_logic_vector(31 downto 0);
shared variable ProgramMem : TProgram :=
( 0  => x"1C000002",  -- R0 = 2
  1  => x"1C110003", -- R1 = 3
  2  => x"11201000",  -- R2 = R0 + R1
  3  => x"30000010",  -- jmp 0x10
  4  => x"20000000", -- вывод в порт, тестовая команда
                     -- для проверки конвейера
 16 => x"1C330005", -- R3 = 5
 others => x"00000000"
);

attribute RAM_STYLE : string;
attribute RAM_STYLE of ProgramMem : variable is "BLOCK";

begin

process(clk)
begin
 if rising_edge(clk) then
  if cmd_we = '1' then ProgramMem(to_integer(unsigned(cmd_addr)))
:= cmd_data; end if;
 end if;
end process;

process(clk)
begin
 if rising_edge(clk) then
   cmd <= ProgramMem(to_integer(unsigned(pc)));
 end if;
end process;

end Behavioral;
```

Листинг 5. Описание памяти программ

Процессор использует регистровый файл и имеет структуру команды, показанную на рис. 5. Старшие 4 бита команды соответствуют типу операции:

- 0000 — нет операции (т. е. код 0 соответствует команде **nop**);
- 0001 — операция АЛУ;
- 0010 — операция с устройством ввода/вывода;
- 0011 — переход по адресу, заданному в поле literal команды;
- 0100 — условный переход по адресу, заданному в поле literal команды.

Разряды 27–24 кодируют тип команды АЛУ. Последующие разряды определяют номера регистра назначения и регистров-операндов в соответствии с рис. 5.

Такой набор команд достаточен для демонстрации основных эффектов и построения временных диаграмм. Для того чтобы проверить работу процессора, необходимо добавить память команд с моделью программы, представленную в листинге 5.

В листинге 5 описана простая программа, помещающая в регистры R0 и R1 операнды и выполняющая затем команду сложения. Далее производится переход по адресу 16, а за командой перехода расположена команда вывода в порт. Подразумевается, что эта команда не будет выполнена из-за ее обхода командой **jmp**.

Схема собранного конвейеризованного процессора показана на рис. 12. Листинг верхнего уровня проекта не приводится, поскольку содержит только соединение ранее описанных модулей на структурном уровне. Схема на рис. 12 соответствует автоматически сгенерированному представле-

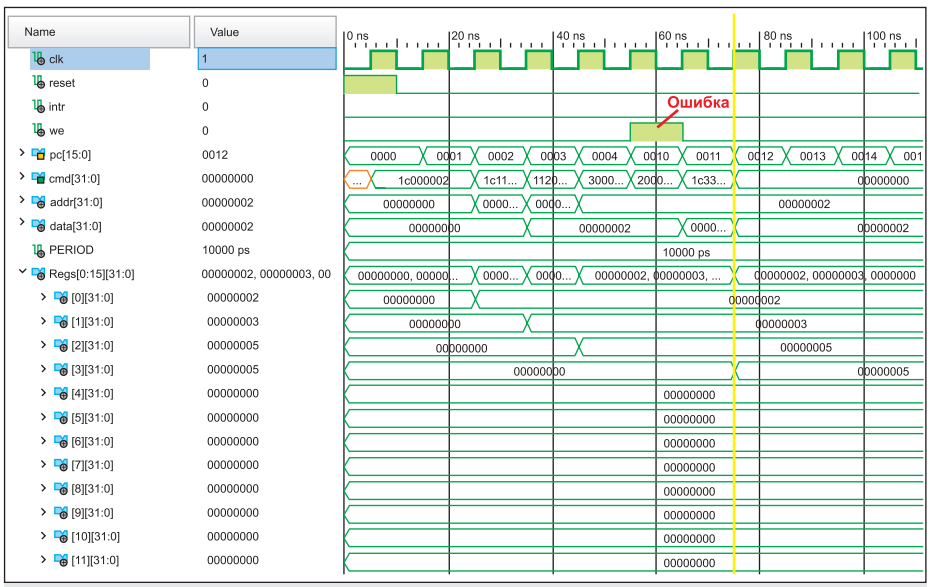

Рис. 13. Временные диаграммы работы тестовой программы с выполнением инструкции, следующей за командой перехода

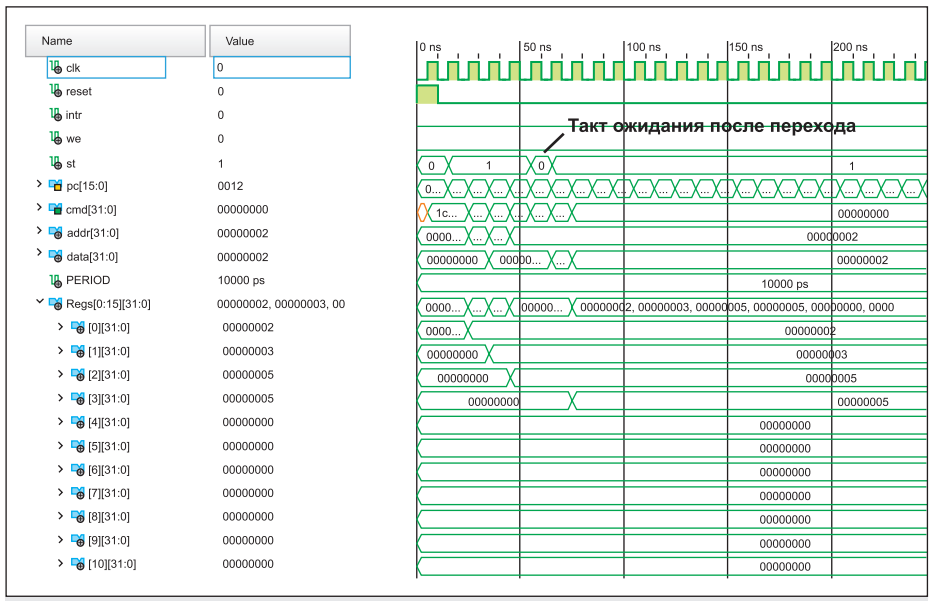

Рис. 14. Временные диаграммы выполнения тестовой программы с введением такта ожидания после нарушения линейного порядка выполнения

нию Schematic на уровне Elaborated Design в САПР Vivado.

Временная диаграмма работы процессора показана на рис. 13. Сигнал **we** формируется для команды записи во внешний порт, и в данном примере предполагается, что эта команда не будет выполняться. Однако на рис. 13 видно, что в момент времени около 60 нс появляется сигнал **we**, свидетельствующий о выполнении команды записи во внешний порт. Это произошло потому, что реализованный конвейер команд успел прочитать команду записи во внешний порт, пока выполнялась команда перехода. Таким образом, на последовательных тактах происходили следующие действия:

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.

Соответственно, 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.

Для решения этой проблемы необходимо ввести состояние конвейера, что показано в листинге 4 закомментированными фрагментами кода, относящимися к сигналу **st**. Видно, что регистр **st** устанавливается в 0 каждый раз, когда нарушается линейный порядок выполнения программы. Для того чтобы прочитанная в конвейер команда не выполнялась, в данном проекте достаточно установить сигналы **write enable** для регистрового файла и внешней шины в 0, если **st** = 0.

Введение в конвейер процессора состояния ожидания не является единственно возможным вариантом. В некоторых процессорных ядрах существует понятие «отложенный переход». Подобная ситуация показана на рис. 14 — команды, которые успели попасть в конвейер, выполняются независимо от того, что они находятся после команды перехода. Это накладывает 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.

Заключение

Рассмотренный материал позволяет перейти к проектам, обладающим важными чертами современных процессорных ядер — наличием конвейеризации и управления состоянием конвейера. Следующим этапом рассмотрения аппаратных архитектур будет анализ более сложных, 3- и 5-ступенчатых конвейеров, а также подходы к планированию и анализу аппаратной архитектуры исходя из особенностей аппаратной платформы и требований программного обеспечения.

Литература

1. Тарасов И. Проектирование процессорных ядер. Цели, задачи, инструменты // Компоненты и технологии. 2018. № 2.
2. Казаченко В. Ф. Микроконтроллеры. Руководство по применению 16-разрядных микроконтроллеров INEL MCS-196/296 во встроенных системах управления. М.: Эком, 1997.
3. Lindholm T., Yellin F., Bracha G., Buckley A. The Java Virtual Machine Specification. Java SE. 7 Edition, 2011.
4. Meyer J., Downing T. Java Virtual Machine. O'Reilly, 1997.
5. www.ecma-international.org/publications/standards/Ecma-335.htm
6. Баранов С. Н., Ноздрунов Н. Р. Язык ФОРТ и его реализации. Серия: ЭВМ в производстве. М.: Машиностроение, 1988.
7. Паттерсон Д., Хеннесси Дж. Архитектура компьютера и проектирование компьютерных систем. Классика computer science. Изд 4-е. СПб: Питер, 2012.