Окончание. Начало в № 3`2007

Микропроцессор своими руками-5.
По поводу начала проекта встроенного в FPGA микроконтроллера

Иосиф КАРШЕНБОЙМ
Iosifk@narod.ru

Время выполнения команды и конвейер команд

Со времен Генри Форда идея конвейера состоит в том, чтобы как можно меньше дать тем, кто трудится, но при этом получить как можно более эффективное производство. Сейчас мы рассмотрим это на примере конвейера команд в микропроцессоре.

Вот одно, самое главное, замечание о пользе конвейера. Вспомните такую картину: расходящиеся круги на поверхности озера от брошенного в воду камня. Точно такая же «картина» имеет место и в кристалле, если схема не имеет регистров. Изменение счетчика команд действует подобно описанному выше камню. В схеме начинаются многочисленные колебания от переходных процессов. А там, где происходят изменения потенциалов, — там происходит и тепловыделение. Применение же регистров в конвейере препятствует «расхождению волн» переходных процессов. А это значит, что схема, работающая без конвейера, будет иметь большее тепловыделение на той же самой тактовой частоте. И поскольку нет волн от переходных процессов, значит, сам переходный процесс завершается быстрее. Ну а теперь приведем те же самые рассуждения, но только в «процессорных терминах».

При поступлении тактового импульса на счетчик команд происходит инкрементирование счетчика адреса команды. Новое значение кода адреса с выходов счетчика адресов поступает на адресный вход памяти команд. После того как память команд выдает код команды, данный код поступает в ALU, где он дешифруется и где вырабатываются управляющие воздействия для обработки данных, находящихся в ALU, в памяти или в регистрах микропроцессора. Последним происходит собственно то действие, которое призвана выполнить данная команда: например, переслать результат вычислений из ALU в память или в регистр, взвести триггер признака и так далее.

Как видно из приведенного выше описания, полный цикл выполнения команды состоит из нескольких этапов:

$$T\text{-сум} = T(1) + T(2)\ \dots\ T(n),$$

где T-сум — суммарное время задержки, $T(1)\dots T(n)$ — времена задержки на $1\dots n$-й частях тракта обработки команды.

Естественно, что выполнить каждую команду быстрее, чем за T-сум, невозможно. Если микропроцессор не имеет конвейера, то каждая следующая команда будет запускаться на исполнение и будет выполняться не раньше, чем через T-сум.

Моделируя различные части тракта обработки команды, можно определить, сколько времени займет обработка в каждой части тракта, и определить наиболее длительные операции. Теперь попробуем организовать конвейерную обработку. Разделим весь цикл обработки на несколько равных частей. Каждая часть времени, назовем ее T-конв, должна иметь длительность больше, чем самая большая задержка из $T(1)\dots T(n)$. Допустим, что мы разобьем весь тракт обработки на n частей. Теперь мы можем запускать на исполнение в конвейер команды не через T-сум, как в предыдущем случае, а через T-конв. А это значит, что теперь время выполнения каждой отдельно взятой команды станет больше, поскольку команда выполняется не за время T-сум, а за время T-конв, умноженное на n. Но конвейер обрабатывает данные потоком, поэтому каждая следующая команда будет выходить из конвейера через время T-конв, которое значительно меньше, чем T-сум.

Как же определить, какой конвейер нам нужен? Для этого рассмотрим последовательность обработки данных в гипотетическом микропроцессоре (рис. 16). Число ступеней

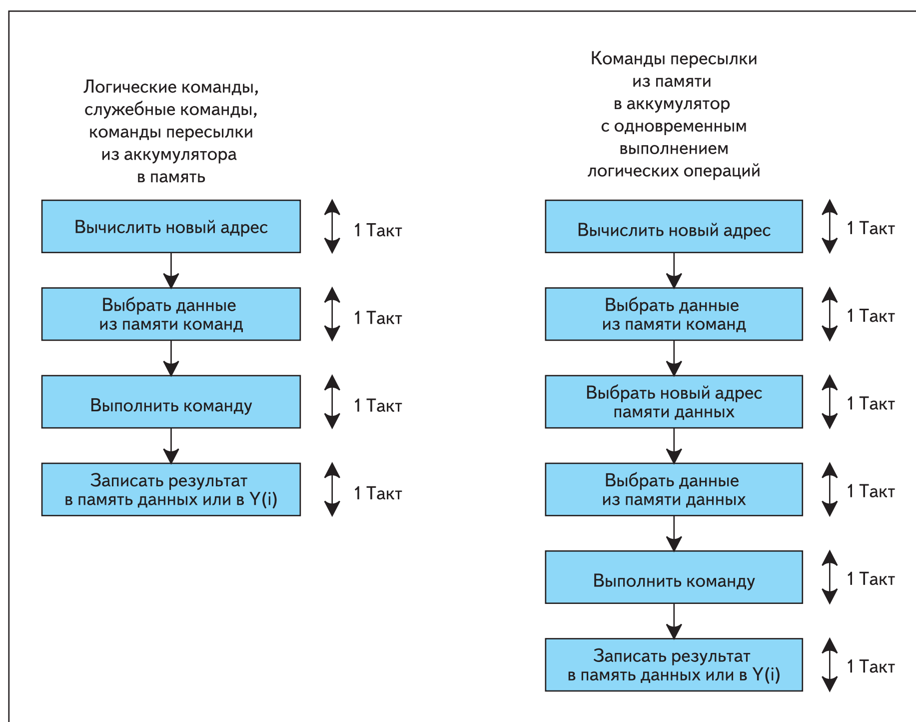

Рис. 16. Гипотетический микропроцессор

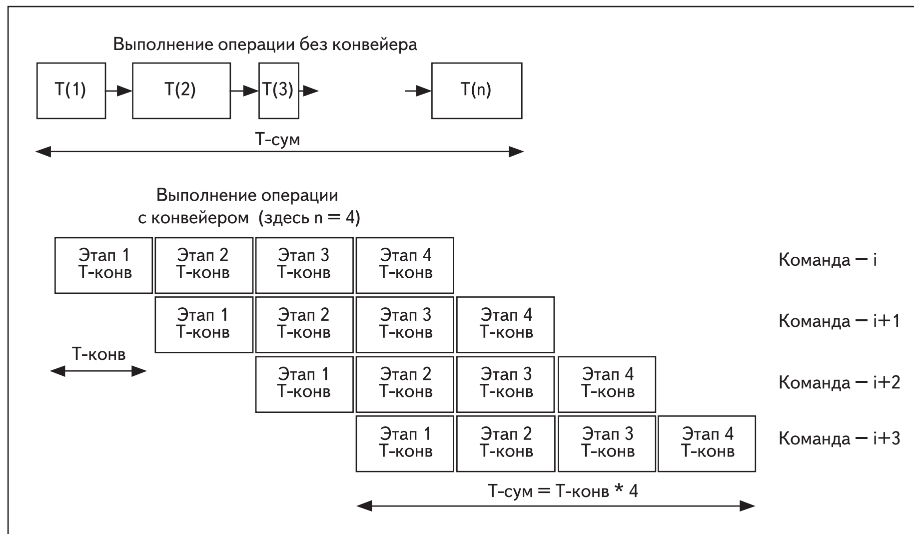

Рис. 17. Работа конвейера, состоящего из четырех стадий

конвейера определяется в зависимости от того, сколько и каких действий нам необходимо выполнить для того, чтобы обработать команду. Перебирая все действия, которые выполняют команды данного процессора, можно выбрать ту команду, которая для своего исполнения требует больше всего элементарных действий. На рис. 17 показано, что если в процессоре должны выполняться команды с разным числом последовательных действий, то конвейер надо выбирать по той команде, которая для своего исполнения требует больше всего элементарных действий, т. е. для такого процессора глубина конвейера равна 6.

Конвейерная обработка

Обычно конвейер для современного RISC-процессора представляет собой пять ступеней. Пример организации конвейера для такого процессора и его краткое описание приведены ниже.

1. Instruction Fetch — выборка инструкции.
2. Instruction Decode/Registers — декодирование инструкции.
3. Forwarding/Branch — пересылка/ветвление.
4. Execute — выполнение.
5. Write Back — обратная запись.

Блок-схема типового ядра RISC-процессора приведена на рис. 18. В первой стадии код команды читается из памяти команд, этот этап называется «выборка инструкции». Вторая стадия декодирует инструкцию и устанавливает биты контроля, которыми производится управление процессором. Результаты этой операции фиксируются в регистре. Данные, выходящие из регистра, передаются в следующий узел конвейера, где производится обработка команды в случае ветвления. От этой стадии конвейера данные посылаются блоку выполнения. И эти данные также можно послать внешнему интерфейсу или блокам памяти данных. В стадии выполнения производится обработка команд, связанная с обработкой двух операндов при работе с регистрами, или команд, при выполнении которых производится непосредственная загрузка данных, т. е. загрузка тем кодом, который поступает в самом коде команды. В последней стадии конвейера результирующие данные выбираются или из ALU, или из памяти данных. Эти данные могут быть переданы назад ко второй стадии конвейера, и там они могут быть записаны в регистры.

Ну а для более быстрых реализаций процессора приходится увеличивать глубину конвейера до 7 ступеней. Сейчас мы не будем рассматривать все аспекты этого вопроса, связанного с увеличением тактовой частоты и с соответствующим увеличением глубины конвейера. Давайте рассмотрим, как это работает.

Итак, обычно, если мы говорим о команде, то имеем в виду RISC-процессор и подразумеваем команду, которая делает «что-то» за один машинный такт. Но в случае применения конвейера это происходит не всегда именно так. Да, большинство команд, выполняемых RISC-процессором, выполняются за один такт, но часть команд просто «не имеет такой возможности». И это и есть та ложка дегтя, которая портит бочку меда! Самое очевидное в этом вопросе — это перезагрузка конвейера команд при нарушении очереди команд. Такая ситуация возникает при выполнении команд типа CALL, JMP, при отработке прерываний и т. д. В таких ситуациях процессор выдает сигнал, блокирующий прием данных из конвейера до момента прихода новых данных из конвейера. При этом может снижаться быстрота действия процессора. Именно может снижаться, потому что в данной ситуации мы можем сделать следующий ход: «перенести» команду, например JMP, на одну команду «вперед» относительно того места, где мы «привыкли видеть» данную команду.

Обычно пишется:

```
Addr_a:
Команда 1;
Команда 2;
JMP Addr_b;
Addr_b:
Команда 3;
```

И выполняется такая последовательность так: сначала команда 1, затем команда 2, затем команда JMP Addr_b, после чего счетчик адресов меняется на Addr_b, из памяти по новому адресу извлекается команда 3, и она исполняется.

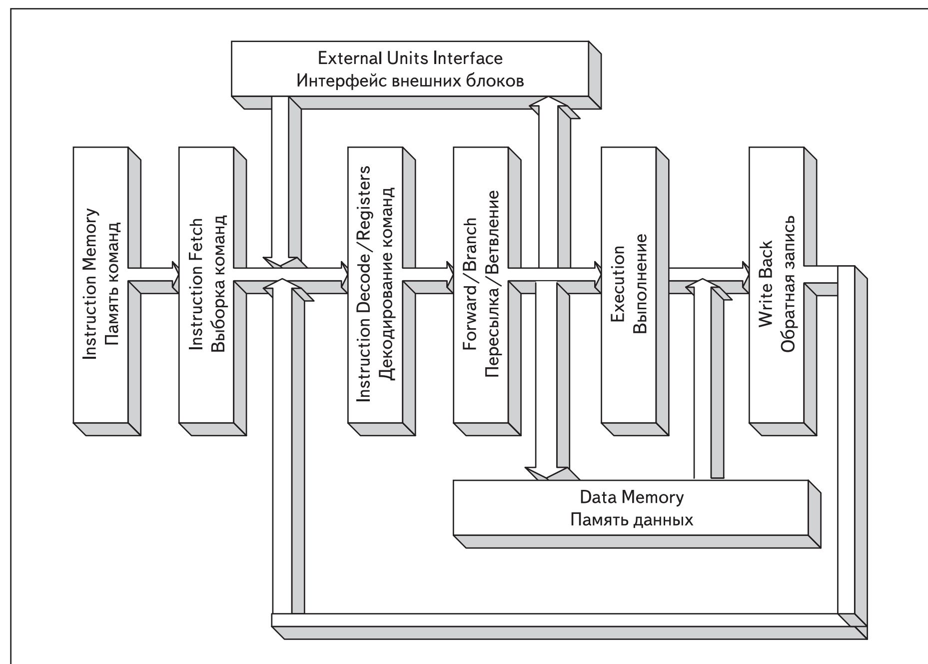

Рис. 18. Блок-схема типового ядра RISC-процессора и соответствующий ему 5-уровневый конвейер команд

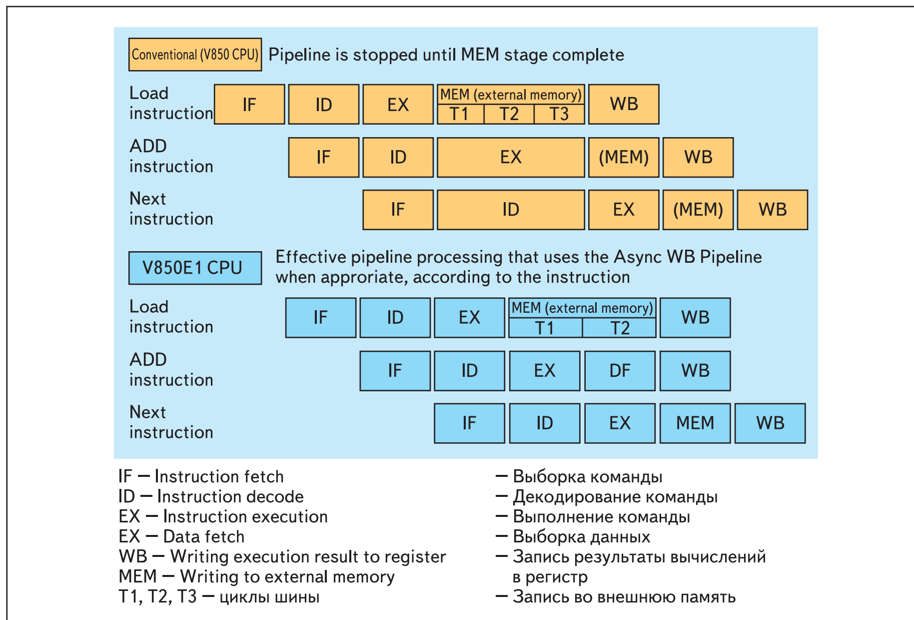

Рис. 19. Пример работы V850 с очередью конвейера

В новом варианте пишется:

```
Addr_a:
Команда 1;
JMP Addr_b
Команда 2;
Addr_b:
Команда 3;
```

Выполняется такая новая последовательность команд следующим образом: сначала, как и в предыдущем случае, выполняется команда 1, затем из памяти извлекается команда JMP Addr_b, после чего счетчик адресов меняется на Addr_b, но данные по новому адресу поступят из памяти команд на один такт позже, так как конвейер осуществляет запись нового адреса в регистр управления памятью команд и, соответственно, задерживает поступление новых данных. А пока в следующем такте придут данные от команды 2. И вот только в следующем такте уже придет команда 3. Как видно из приведенного примера, команды выполняются одна за другой и перезагрузки конвейера не требуется. То же самое делает и процессор ADSP-219x [14]. Описание это довольно подробное, и поэтому читатели сами смогут разобраться с работой конвейера ADSP-219x.

Фирма ADI называет такие инструкции «задержанными» — Delayed Jumps/Calls и Delayed RTI/RTS.

Но кроме команд переходов, когда приходится очищать весь конвейер, есть еще команды, которые не требуют очищать и затем заново заполнять весь конвейер. Таким командам необходим только один, иногда два такта работы конвейера. К ним относятся команды, выполняющие обратную запись. Представим себе такую последовательность команд: первая команда выполняет какое-либо действие над операндом или операндами, и результат этой операции записывается куда-либо для промежуточного хранения. А следующая за этой команда по результатам действия предыдущей команды выполняет какое-либо условное действие. В таком варианте работы первой из этих двух команд требуется дополнительное время для того, чтобы записать результат работы и приостановить выполнение последующей операции до того момента, пока новый результат вычислений не будет готов. За примером работы в этом случае обратимся к процессору V850, производимому фирмой НЕК. В верхней части рис. 19 показана работа конвейера процессора V850. В том случае, когда процессору требуются дополнительные такты синхронизации (T1 — основной и T2, T3 — дополнительные такты) для записи данных во внешнюю память, производится приостанов конвейера.

Как решаются эти вопросы в «больших» процессорах? Фирма НЕК в своих следующих разработках процессоров произвела сначала модернизацию регистрового файла и выполнение им операции WB. Теперь процессор с ядром V850E1 имеет возможность делать две записи в регистровый файл за один период синхрочастоты. Это позволяет не приостанавливать конвейер команд для выполнения команд при обращении к памяти. В нижней части рис. 19 показана работа конвейера процессора V850E1. В этом процессоре применена асинхронная запись в регистровый файл, поэтому две операции WB могут выполняться одновременно и приостанов конвейера не производится.

Следующим шагом этой фирмы стало разделение выполняемых команд на два потока и выполнение этих двух потоков на двух аккумуляторах. На рис. 20 показана структура процессора V850E2 [15]. На рис. 21 показана диаграмма работы такого конвейера.

Сначала поток команд проходит через 3 блока, которые на рис. 20 обозначены как Instruction fetch pipeline (Fpipe), и затем разделяется на две ветви — левую и правую.

Первый из трех субблоков в блоке Fpipe — блок выборки инструкции — Instruction fetch unit (Bpipe). Он может принимать до 8 16-бит-

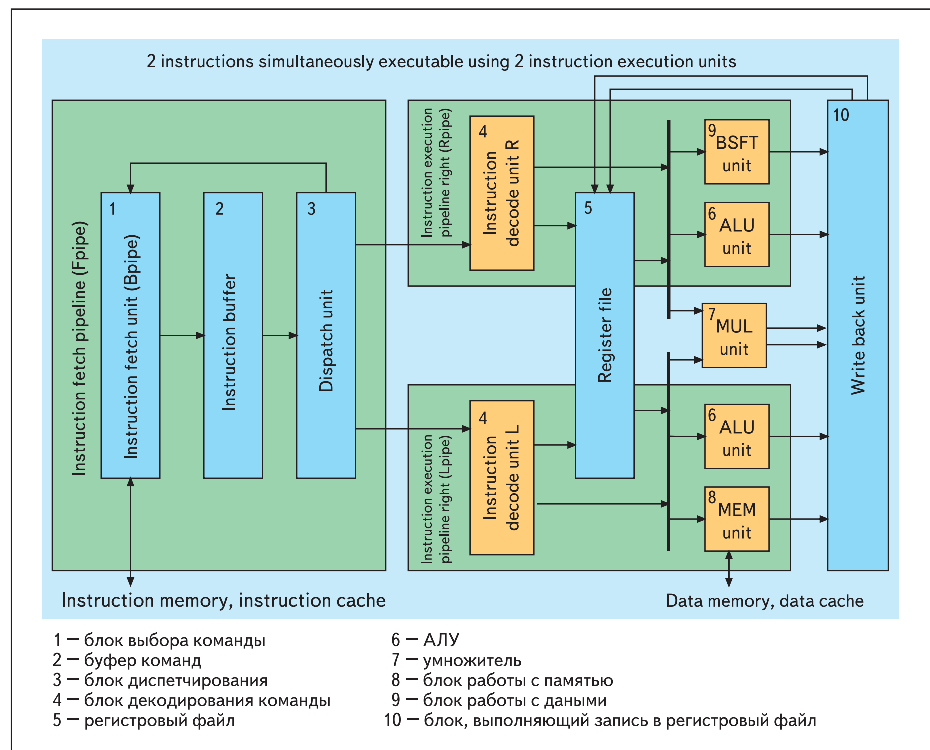

Рис. 20. Структура процессора V850E2

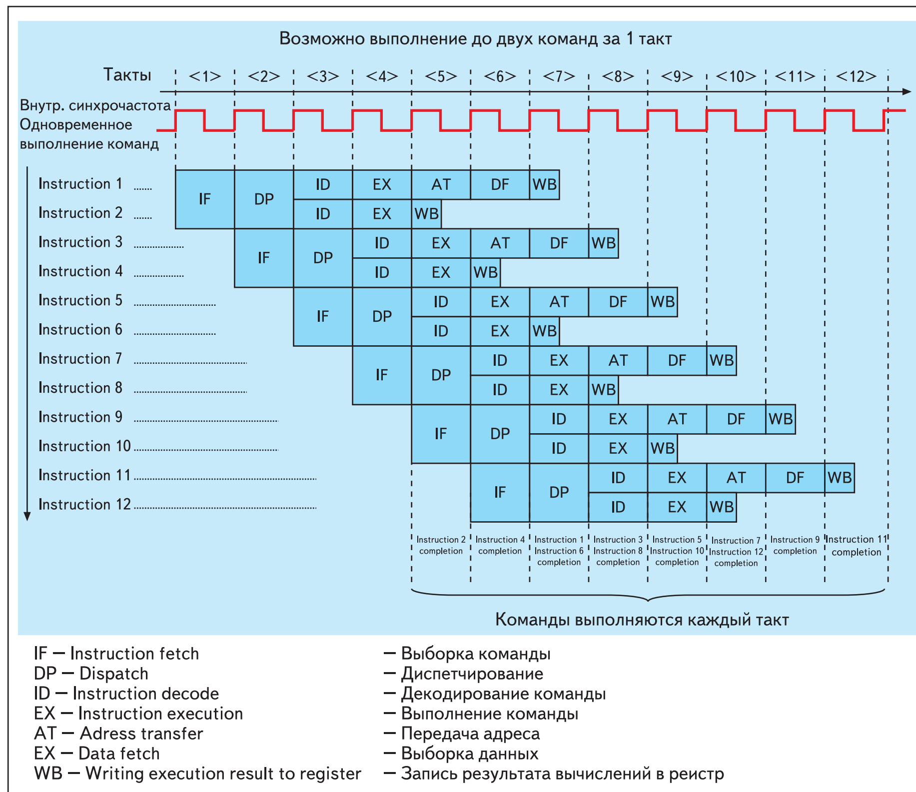

Рис. 21. Диаграмма работы конвейера процессора V850E2

ных инструкций в одном цикле по 128-битной шине iLB. Следующий блок — буфер инструкций. Третий блок — диспетчер (Dispatch unit). В нем накапливаются две 128-битные инструкции. В этом блоке идет проверка инструкций на то, что их можно выполнять одновременно и независимо. Также в этом блоке инструкции распределяются на две ветви. Левая и правая ветви работают почти одинаково. В обеих ветвях производится декодирование инструкции, затем выполнение. Далее в левой ветви, в блоке MEM, производится, если необходимо, запись в память или чтение из памяти. А в правой ветви, в блоке BSFT unit, производится обработка данных. Блок MUL unit предназначен для перемножения данных. Блок Write back unit предназначен для записи данных в регистровый файл.

А теперь пора вспомнить о том, что бывают не только RISC-процессоры, но и более сложные варианты процессоров, а также имеются и более сложные команды, которые выполняют несколько примитивных последовательностей. Соответственно, здесь необходимо учитывать все задержки от выполнения всех шагов таких последовательностей.

Циклограмма и переключение контекста

В самом начале этой статьи был рассмотрен вопрос о том, как производится опрос датчиков и в каком порядке. Теперь давайте еще раз обсудим этот момент более подробно. Было показано, что практически при одинаковой тактовой частоте процессора можно улучшить время реакции микропроцессорного узла управления на несколько порядков. В этом рассмотрении говорилось об обработке информации от датчиков как об одной задаче, выполняемой микроконтроллером. Но ведь в реальной обстановке микроконтроллер выполняет одновременно несколько задач. При этом одни задачи могут добавляться, а другие задачи сниматься. Запуск новых или активация временно «законсервированных» задач требует определенных временных затрат. Точно так же требуется некоторое время для того, чтобы завершить исполнение задачи или ее приостанов. Переключение с задачи на задачу тоже требует определенных затрат времени.

Чтобы успевать работать быстро с несколькими задачами, процессор должен как можно быстрее переключаться с задачи на задачу. Именно так и мы и будем рассматривать этот вопрос. И он только косвенно связан с тактовой частотой процессора. Рассмотрим, как и какие аппаратные решения влияют на время переключения задач.

Обработка вложенных прерываний и переключение контекста

Представим себе, что в микропроцессоре работает встроенная операционная система, которая имеет единый квант времени для выполнения задач. Например, вот такой алгоритм: «получил прерывание от часов, проверил расписание обхода и опросил». При таком алгоритме работы все задержки от переключения задачи имеют несколько составляющих. Начнем рассмотрение с того, как таймер выставил сигнал запроса прерывания. Этот сигнал поступает в контроллер прерываний. Здесь нам необходимо учитывать латентность (задержку на выполнение) контроллера прерываний. Контроллер прерываний — статический автомат, шифрующий состояния приоритетов. На входе у него регистр маски прерываний, триггер глобального разрешения-запрета. На прохождение сигнала через эти цепи требуется некоторое время. Далее сигнал запроса приходит в АЛУ и … Но перед этим должна завершиться текущая выполняемая команда. И эта команда не всегда бывает однотактная.

В качестве примера возьмем AduC7025 с ядром процессора ARM7TDMI [16].

В самом худшем случае время ожидания для FIQ состоит из самого большого отрезка времени, которое может потребоваться для сигнала запроса, чтобы пройти через синхронизатор, плюс время для завершения самой длинной команды — LDM, которая загружает все регистры, включая PC, плюс время, необходимое для аварийного прекращения работы данных, плюс время для входа в FIQ. В конце этого времени ARM7TDMI будет выполнять команду, расположенную по адресу 0x1C (адрес вектора прерывания FIQ). Максимальное время — 50 циклов процессора — равно 1,2 мкс для системы, использующей для процессора синхрочастоту в 41,78 МГц. Максимальное вычисление времени ожидания запроса на прерывание IRQ подобно приведенному выше, но необходимо учесть тот факт, что FIQ имеет более высокий приоритет и может задержать вход в запрос на прерывание IRQ, поскольку обслуживание прерывания верхнего уровня может проводиться в течение произвольного отрезка времени. Минимальное время ожидания может быть уменьшено до 42 циклов, если не используется команда LDM , поэтому некоторые компиляторы имеют такую опцию, позволяющую компилировать, не используя данную команду. Другая опция, позволяющая уменьшить латентность до 22 циклов, состоит в том, что программа должна выполниться в режиме THUMB. Минимальное время ожидания для FIQ или прерываний IRQ — всего пять циклов — состоит из времени, необходимого для того, чтобы запрос мог пройти через синхронизатор, плюс время, требуемое для выполнения режима исключения. Далее надо рассмотреть, как работает память — внутренняя или внешняя, да и много чего еще, связанного с аппаратной реализацией микроконтроллера. Но для данного этапа рассмотрения давайте пока ограничимся этим.

Жизнь показывает нам, что внешний мир, находящийся вокруг микроконтроллера, с каждым годом становится все больше и сложнее. Что можно было получить от микроконтрол-

лера с 1 К памяти и как можно было его использовать? Пара прерываний и все. Главный цикл программы с вызовом подпрограмм обслуживания прерываний. Теперь, поскольку микроконтроллеры стали значительно мощнее, требований, а значит и задач, выполняемых ими, стало значительно больше. Больше внешних воздействий приходит на микроконтроллер. Значит ли это, что число запросов прерываний стало больше? С одной стороны — да. Входов запросов прерываний действительно стало больше. Но как это число связано с количеством обслуживаемых задач? Известно, что в любом микроконтроллере, кроме блоков памяти, есть два самых больших узла, если судить по объему занимаемого ими оборудования. Это дешифратор команды и шифратор приоритета в контроллере прерываний. Поэтому невозможно значительно увеличить число запросов на прерывание и при этом получить быстрый шифратор. А это значит, что в FPGA дело усугубляется еще в большей степени, чем в ASIC. Следовательно, нам будет необходимо принимать какие-то дополнительные меры. Сейчас мы просто зафиксируем факт того, что контроллер прерываний — узел, требующий дополнительного внимания. И перейдем к рассмотрению переключения контекста.

Переключение контекста, кто и как с этим справляется

Как работало переключение контекста в первых 8-разрядных микроконтроллерах? Адрес возврата заносился в стек, а содержание регистров сохранялось и восстанавливалось программистом «вручную». Стек был выполнен аппаратно и располагался в общем поле памяти данных. Теперь на смену 8-разрядным процессорам приходят 16- и 32-битные процессоры. Например, процессоры с архитектурой ARM7.

У процессора ARM7TDMI аппаратного стека как такового нет, кроме одного регистра, в котором производится сохранение только одного адреса возврата. А все манипуляции с остальными регистрами производятся также «вручную». И обработка вложенных прерываний точно так же поддерживается программным способом.

Основа процессора, выполненного на основе регистров, — это именно регистры общего назначения. И в большинстве случаев они задействуются при работе вполне определенным образом, определяемым компилятором или другим программным инструментом. А следовательно, при переключении контекста необходимо производить сохранение регистров и загружать их новыми значениями. Ну, что-то вроде лозунга: «Поработал — убери за собой рабочее место»… А что же можно здесь сделать, кроме этого? Давайте рассмотрим несколько примеров. Что мы хотим увидеть? Бывает ли так, чтобы не делать эту самую «уборку», а просто продолжить работу на другом «рабочем месте»? Да, без сомнений, именно такой режим работы и позволяет делать переключение контекста очень быстрым. И признанный чемпион в этом деле — стековый процессор. Представьте себе, что вы имеете стек данных очень большой величины. Далее вы разбиваете этот стек на несколько рабочих областей, так чтобы каждой задаче «досталась» своя область. Теперь, чтобы переключиться с задачи на задачу, вам не надо что-то сохранять или загружать. Надо просто перенаправить указатель вершины стека на другую область, соответствующую новой задаче.

И еще один пример хочется привести в этом разделе. Так уж получилось, что всплеск числа проектов, выполненных с применением «самодельных» soft-процессоров, выпал на те годы, когда еще только-только появились FPGA достаточного объема, такого, чтобы в них мог поместиться проект со встроенным процессором. Это было то время, когда еще не набрали полной силы soft-процессоры, предлагаемые фирмами — изготовителями микросхем. А дальше, примерно после 2002–2004 годов, число проектов «самодельных» soft-процессоров резко сократилось именно потому, что фирмы — изготовители микросхем учли потребность рынка в soft-процессорах и предложили пользователю полный комплект soft-процессоров вместе с программными инструментами. Здесь, в этой части описания о переключении контекстов хочется показать читателям оригинальный проект soft-процессора SILVERBIRD [17]. Этот проект выполнялся в 2000 году студентами в Integrated Systems Laboratory of the Swiss Federal Institute of Technology, Zurich, Switzerland. Сама структура конвейера выбрана совершенно стандартной (рис. 22), но некоторые новшества были применены для ускорения отработки переключения контекста по прерываниям.

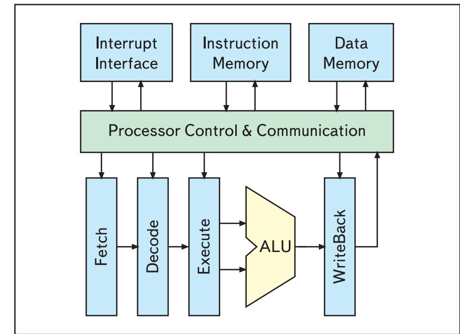

Рис. 22. Структура конвейера процессора «SILVERBIRD»

Порядок работы конвейера стандартный — выборка, декодирование, выполнение, запись.

В этом проекте очень удачно скомбинированы достоинства двух архитектур, выполненных на основе регистров и на основе стека. Процессор с регистрами легче программировать и для него легче сделать компилятор языка высокого уровня. Но в случае переключения контекста регистры такого процессора надо долго перезагружать. Для процессора, выполненного на основе стека, сложнее программировать (в том числе и потому, что нет достаточного числа квалифицированных программистов), но, как мы помним, одним из преимуществ стекового процессора является быстрое переключение контекста. Комбинируя эти преимущества, швейцарские студенты сделали процессор, в котором регистры общего назначения выполнены в виде вершины стека. И точно так же, как и для управления обычным стеком данных, в этом проекте для того, чтобы переключить контекст, достаточно просто выдать команды, аналогичные командам push или pop, или перезагрузить указатель стека. На рис. 23 показано, что регистры общего назначения (они обведены красной рамкой) выполнены как

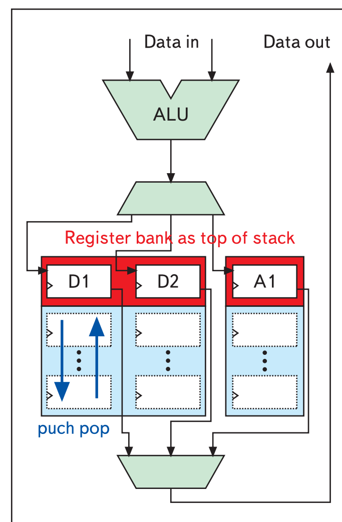

Рис. 23. Регистры общего назначения (они обведены красной рамкой) выполнены как вершина стека

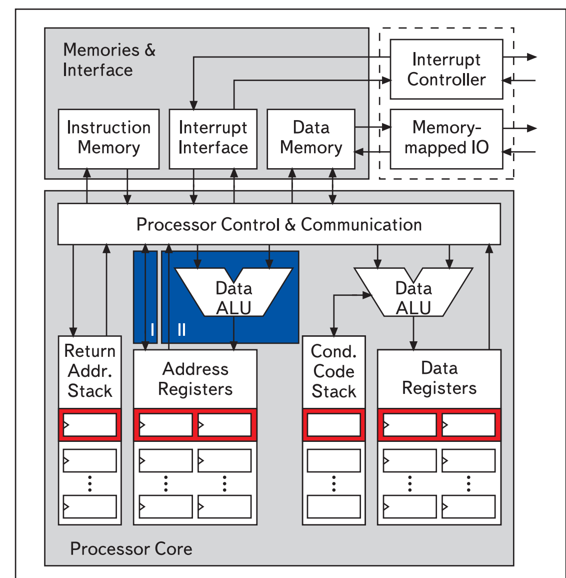

Рис. 24. Блок-схема процессора SILVERBIRD

вершина стека. На рис. 24 показана вся блок-схема процессора SILVERBIRD. Там же показано и то, что и остальные регистры этого проекта (они тоже, как и на предыдущем рисунке, обведены красной рамкой) выполнены как вершина стека.

Кто еще ведет себя таким же образом, как и процессор SILVERBIRD? Процессор Nios! В нем вместо регистров используется регистровый файл. Причем для работы этот файл доступен через окно, так что можно работать одновременно с 32 адресами так же, как с 32 регистрами. А вместо сохранения регистров в этом регистровом файле происходит переключение окна на следующую группу из 32 регистров. Когда указатель достигает предела по переполнению или по опустошению регистрового файла, то сначала вырабатывается прерывание, которое инициирует перенос программой содержимого регистрового файла в основную память. Таким образом, для определенного числа одновременно обрабатываемых задач смена контекста будет проходить довольно быстро благодаря переключениям указателя окна регистрового файла. Но при этом необходимо всегда отслеживать то, что у нас есть запас по ресурсам в регистровом файле.

В заключение раздела по переключению контекста необходимо отметить, что здесь были рассмотрены только аппаратные узлы. Они обычно скрыты от программиста. Но вместе с тем можно сделать еще несколько шагов по ускорению работы soft-процессора. Правда, эти шаги уже будут «задевать» и программистов. Поэтому их описание будет приведено в разделах, описывающих кодировку команд.

Переключение контекста и кодировка команд

Переключение контекста — это мощный инструмент повышения скорости реакции процессора на внешние воздействия. Как переключение контекста может быть связано с кодировкой команд?

Давайте представим себе то, как процессор обрабатывает флаги, поступающие от битовых переменных. Простейший вариант — собрать флаги в слово и читать это слово процессором. Далее накладывать маску, выделять требуемый бит и по состоянию регистра судить — равен искомый бит 0 или 1. Для очень бюджетных проектов может быть и неплохо. Но довольно долго. А давайте применим следующий формат команды. Сведем все флаги в битовое поле и направим их в аппаратный мультиплексор типа N бит в 1 бит. И, соответственно, в состав команд процессора введем команду чтения бита, находящегося по известному адресу в поле битовых переменных. Теперь нам не нужно читать битовый массив и накладывать на него маску. Все выполняется только одной командой: «Прочитать бит по адресу N». Но вот в чем беда. Для адресации битового поля нам требуется занять определенное количество разрядов в поле команды. А как быть в том случае, когда это поле команды не позволяет адресовать все требуемые по проекту биты? Или скажем так: если мультиплексор, требующийся для такой операции, получится довольно медленный. Вот тут давайте вспомним, что в отличие от процессора общего применения «самодельный» процессор всегда делается и оптимизируется под конкретную задачу. Следовательно, мультиплексор битового поля можно разбить на два. Один из них будет переключаться в соответствии с контекстом, а второй, как и раньше, управляться кодом, приходящим в команде. Так один и тот же бит в битовом поле будет иметь разные физические «корни» в зависимости от того, какую задачу решает процессор или какое он обрабатывает прерывание. Неудобно? Да нет, просто необходимо сгруппировать эти биты таким образом, чтобы они представляли собой одинаковые «сюжеты». Например, бит готовности на прием может находиться на той же позиции, что и бит готовности на передачу. Вошли в программу обработки передатчика, проверили бит и передали. То же и для готовности на прием. То же может быть сделано и с битами ошибок.

А как же медленный мультиплексор? Раньше у него был только один такт на то, чтобы промультиплексировать бит. Ну а если не успевает этот мультиплексор, то тогда ввести все сигналы за один такт RISC нельзя. А во втором варианте у мультиплексора появились два времени для работы. Первое — это время, определенное одним тактом для RISC, а второе — дополнительное время, которое требуется для прохождения сигнала через первую ступень дешифрации. Но теперь уже процессор с такой командой работать будет. При этом просто надо будет учесть появившуюся задержку в один такт, или в несколько тактов, в зависимости от реализации мультиплексора. Простейший вариант в таком случае — это вставить команду NOP.

Команда тестирования бита и команда условного перехода

Что делается после тестирования бита? Обычно выполняется условный переход или условный вызов подпрограммы. Как тестировать бит, мы уже «прошли». Теперь давайте рассмотрим, как можно облегчить себе жизнь при обработке ветвлений. Команды ветвлений — это, можно сказать, самое тонкое место в сегодняшнем процессоре. Чем быстрее работает процессор, тем большая у него глубина конвейера. А чем больше глубина конвейера, тем больше потери при перезагрузке конвейера. Невероятные усилия применяются разработчиками процессоров, чтобы не терять драгоценные такты конвейера. А между тем для бюджетных вариантов будет вполне достаточно применить команду пропуска — SKIP. А если говорить точнее, то пропуска по условию. Например, по флагу — SKIPF. Сводим биты флагов в битовое поле, а то, как это должно выглядеть, мы рассматривали в предыдущем разделе. Формируем команду, которая в зависимости от состояния бита флага, находящегося по адресу N, разрешает выполнение следующей за этой команды или блокирует ее, превращая ее код в NOP. Что мы выигрываем в таком случае? Если мы имеем процессор с конвейером в 5 ступеней, то при выполнении условного перехода мы потеряем 5 тактов конвейера. При выполнении команды SKIPF мы потеряем только один такт. А то, что флаги мы можем формировать по своему усмотрению, означает, что таким образом мы получим все необходимые нам команды ветвлений из связки команд SKIPF+JMP или SKIPF+CALL или SKIPF+RET или SKIPF+IRET.

По поводу команды перехода необходимо отметить следующий момент. Поле команды, в которое должен помещаться адрес перехода, должно соответствовать разрядности поля адресов. Но это может оказаться не всегда выполнимым. В таком случае в дело вступают различные приемы формирования кода адреса. Например, выполнение «коротких» переходов. Такие переходы делаются как переходы на 127 адресов вперед или назад от текущего положения счетчика команд. При этом необходимо учитывать то, что счетчик команд «опережает» текущую выполняемую команду на величину, равную глубине конвейера. Другим вариантом является формирование адреса перехода как суммы значений, содержащихся в коде команды и в одном из регистров.

Здесь автор считает, что вопрос «с чего начать проект» описан достаточно подробно. Теперь остается только разработать сам микроконтроллер. Но описать все это в одной статье сложно. Как и что надо разрабатывать — определяется конкретной задачей. А привести рекомендации на все случаи просто невозможно. Посему на этом данную статью хочется закончить.

Вот теперь уже можно подвести некоторый итог

Цикл «Микропроцессор своими руками» и другие статьи этого же направления печатаются уже семь лет, поскольку первая статья на эту тему была напечатана в 2000 году. Большая часть из указанных далее статей были напечатаны в журнале «Компоненты и технологии». Если статьи были написаны совместно с другими авторами, это дополнительно указывается. В том случае, когда статьи были напечатаны в других журналах, об этом тоже сообщается дополнительно. Вот теперь мы можем расставить эти статьи в определенном порядке, так, чтобы они представляли собой единый сюжет.

Расставим их вот таким образом.

1. «Микропроцессор своими руками-5. По поводу начала проекта встроенного в FPGA микроконтроллера». В этой статье излагаются вопросы разработки задания на разработку и приводятся общие вопросы синтеза микропроцессора в соответствии с требуемым заданием.
2. «Квадрига Аполлона и микропроцессоры». В этой статье излагаются вопросы определения производительности труда разработчиков и вопросы разбиения задания на разработку на части для того, чтобы уменьшить трудоемкость выполнения проекта.
3. «Микроконтроллер для встроенного применения — NIOS. Конфигурация шины и периферии».
4. «Перевод описания микропроцессора NIOS. Описание и практическое руководство. Шина, арбитраж, подключение периферийных блоков. Лабораторные работы». ht tp://ww w.altera. ru/cgi-bin/go?519
5. «Система команд микропроцессора NIOS».
6. Семенов Н., Каршенбойм И. Микропрограммные автоматы на базе специализированных ИС // Chip News. 2000. № 7. — Небольшой обзор встроенных микроконтроллеров.
7. «Микропроцессор своими руками», часть 1. Пример разработки встроенного микроконтроллера.
8. «Микропроцессор своими руками-2. Битовый процессор». Пример «нестандартной» архитектуры микропроцессора.
9. «Стековые процессоры». Обзор стековых процессоров.
10. «Микропроцессор своими руками-3». Ассемблер и софт-симулятор.
11. Каршенбойм И., Паленов К. Встроенный логический анализатор — инструмент разработчика встроенных систем // Схемотехника. 2001. № 12.
12. «Микропроцессор своими руками-4. Как отладить встроенный в FPGA микроконтроллер»? Описание методик отладки встроенных микроконтроллеров. Применение порта JTAG для отладки проектов пользователя.

Что же теперь получается? Получается небольшой курс по встроенным в FPGA микроконтроллерам, начиная от разработки задания на разработку и до отладки проекта в кристалле. К этому курсу можно только добавить «Записки инженера», в которых рассказывается о некоторых случаях, произошедших с автором во время отладки и «боевой» работы систем управления технологией на Байконуре. С «Записками инженера» читатели также смогут ознакомиться на сайте автора.

Конечно, на данную тему написано уже достаточно много. Но все же жизнь идет вперед, и вслед за сегодняшними горизонтами открываются новые. Поэтому возможно, что данный цикл будет в дальнейшем продолжен.

Литература

1. Каршенбойм И. Квадрига Аполлона и микропроцессоры // Компоненты и технологии. 2006. № 4, 5.
2. ht tp://en.wikipedia.o rg/wiki/SystemC
3. ht tp://w ww.mentor.c om/products/c-based_design/index.cfm
4. ht tp://w ww.celoxica.c om/products/dk/default.asp
5. ht tp://ww w.impulsec.c om/
6. w ww.altera.c om/c2h
7. Каршенбойм И. Микропроцессор своими руками. Часть 1 // Компоненты и технологии. 2002. № 6, 7.
8. Каршенбойм И. Микроконтроллер для встроенного применения — NIOS. Конфигурация шины и периферии // Компоненты и технологии. 2002. № 2–5.
9. ht tp://w ww.xilinx.c om/ipcenter/processor_central/picoblaze/picoblaze_user_resources.h tm
10. Каршенбойм И. Микропроцессор своими руками-2. Битовый процессор // Компоненты и технологии. 2003. № 6, 7.
11. Tomaszewski E. Explicitly Parallel RISC (EPRISC), ht tp://ww w.opencores.o rg/articles.cgi/view/4
12. ht tp://w ww.jwdt.c om/~paysan/4stack.ht ml
13. Chapman K. Creating Embedded Microcontrollers (Programmable State Machines). Part 1, 2, 3. 03/28/2002, w ww.xilinx.c om
14. An Overview of the ADSP-219x Pipeline. Engineer To Engineer Note. EE-123, ww w.analog.c om
15. U17135EJ1V1UM00.pdf; V850E2 32-bit Microprocessor Core Architecture. ht tp://w ww.eu.necel.c om
16. ADuC7024_25_PrD.pdf, w ww.analog.c om
17. ht tp://w ww.trash.n et/~luethi/study/silverbird/silverbird.ht ml