w ww.finestreet. ru

Продолжение. Начало в № 3 `2007

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

Иосиф КАРШЕНБОЙМ
losifk@narod.ru

Начинаем рассматривать архитектуры...

В качестве примера многопроцессорной **архитектуры с независимыми процессорами** можно рассмотреть архитектуру сетевого процессора (рис. 8). Процессор такого типа содержит несколько специализированных процессорных ядер, выполняющих обработку поступающих из сети пакетов. Обычно в сетевой процессор входят от 4 до 8 ядер процессоров, обрабатывающих пакеты. Такие процессорные ядра имеют специфичную систему команд, ориентированную на обработку данных в пакетах. Управляющий процессор производит загрузку в процессоры обработки данных пакетов и выполняет другие «медленные» задания. Сопроцессор производит вспомогательные операции по проверке данных на соответствие паттернам, вхождению данных в таблицы и криптованию. Примером таких процессоров могут служить микросхемы семейства IXP.

Пример **архитектуры с процессорным конвейером**. Фирма Agere Systems, Inc. для своих сетевых процессоров выбрала другую архитектуру. Если в предыдущем варианте сетевого процессора обработка шла одновременно на нескольких ядрах-вычислителях, то в процессоре фирмы Agere используется конвейерная обработка данных. В состав конвейера входят Fast Pattern Processor (FPP), выполняющий классификацию пакетов, и Routing Switch Processor (RSP), выполняющий функцию роутинга пакетов. Кроме этих двух процессоров в состав микросхемы входит еще и блок системного интерфейса — Agere System Interface (ASI). На рис. 9 показана блок-схема такого процессора.

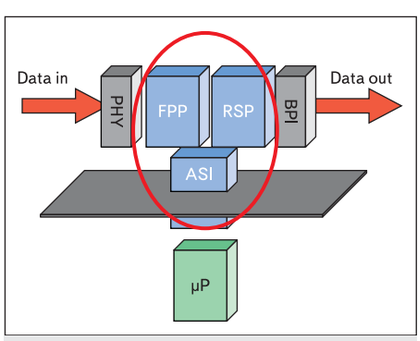

Рис. 9. Блок-схема сетевого процессора фирмы Agere

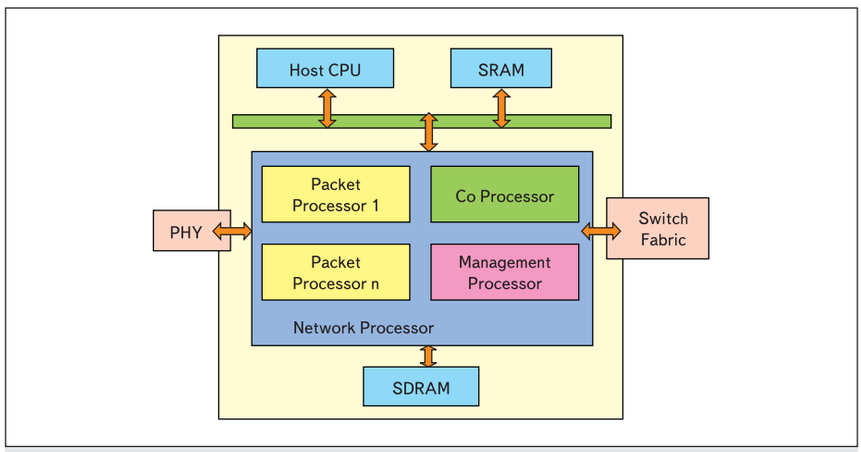

Рис. 8. Блок-схема сетевого процессора

Пример **архитектуры с параллельными процессорами**. Пример архитектуры сетевого процессора с параллельными вычислительными узлами может быть взят на сайте [11]. Для одновременного управления всеми вычислительными узлами используется слово команды очень большой длины.

Еще об одном варианте такой архитектуры кратко упоминалось в обзоре стековых процессоров. Этот проект назывался 4stack [12].

Процессор использует набор команд, ориентированных на работу со стеками. В один процессор входят четыре вычислительных узла — арифметико-логических устройства, выполненных как VLIW (слово команды очень большой длины) сопроцессоры. Процессор выполняет четыре операции со стеками, две операции load/store и две операции модификации адреса. Два блока DSP MAC и два сопроцессора для операций с плавающей точкой (один сумматор и один умножитель) позволяют производить высокоэффективную обработку сигналов и трехмерные геометрические вычисления. Применение стека, как было подробно описано в обзоре, позволяет значительно увеличивать плотность кодирования команды. В то время как «нормальный» RISC-процессор использует для одной команды 32 бита, процессор 4stack одновременно выполняет 8 операций при использовании слова команды в 64 бита. Архитектура VLIW не всегда позволяет выполнять все операции одновременно, однако даже в том случае, когда выполняются не менее чем две операции, процессор 4stack имеет выигрыш по производительности. Это дает лучшее использование памяти программы, что приводит к более дешевым чипам (меньший кэш команды) и к меньшим системным затратам (меньше требуемой памяти). Блок-схема процессора 4stack приведена на рис. 10.

Рассмотрев предыдущие примеры, можно сделать следующие предположения. Выбор архитектуры вычислителя определяется задачей. Если задачу, решаемую вычислителем, можно разделить на несколько процессоров, то применяем такой подход. При этом каждый из процессоров должен иметь полный набор команд для решения возложенных на него задач. Но в этом варианте удобнее произ-

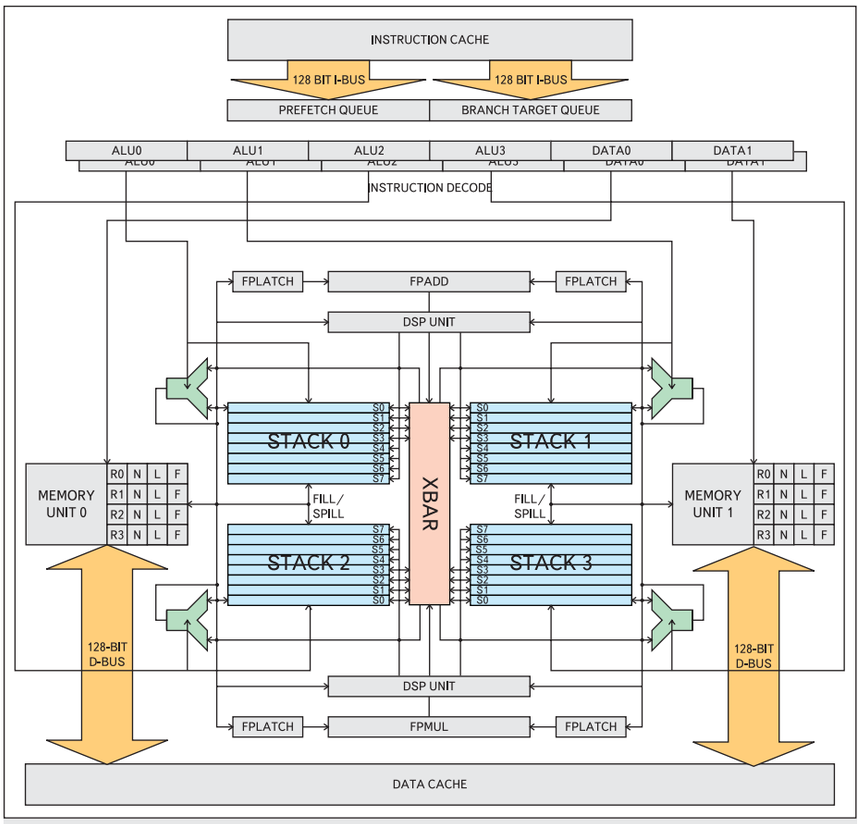

Рис. 10. Блок-схема процессора 4stack

водить масштабирование всего устройства. Хотим повысить производительность — добавляем в схему однотипные процессорные узлы. Если применяемые алгоритмы значительно отличаются для разных ветвей решения задачи, то тогда выгоднее применить конвейер из процессоров. При этом процессоры, применяемые в различных частях конвейера, могут быть оптимизированы по системе команд, а, следовательно, и по занимаемым ресурсам. Но при этом нельзя добавлять производительность. Можно лишь добавить еще такой же процессор «целиком».

А теперь давайте представим, что задача, выполняемая вычислителем, не столь велика и разнообразна, и поэтому мы можем собрать, скажем так, «несколько аппаратных вычислительных узлов в один».

Примером такой задачи может служить вычисление «бабочки» при FFT. Вот алгоритм расчета «бабочки» (рис. 11). Здесь A и B — входные данные, X и Y — выходные данные. Блок, обозначенный как «+», выполняет операцию суммирования, блок со знаком «–» выполняет операцию вычитания, а блок со знаком «×» — операцию перемножения над данными, поступающими от B и K.

При таких вычислениях необходимо взять данные из области памяти, соответствующей входным данным, выполнить вычисления по приведенному выше алгоритму, а потом записать результат в память, соответствующую области выходных данных. При таком алгоритме вычислений нет необходимости непрерывно менять параметры вычислительного устройства, а единственное, что нужно делать, так это менять коэффициенты K для умножителя.

Исходя из изложенного выше задания, можно организовать следующую архитектуру. Давайте сделаем первый вычислитель «головным». Его выходные данные направим в следующий вычислитель и так далее. Получим «многоступенчатый» вычислительный узел. Вычисления он будет производить конвейерно и за каждый такт вычислений выдаст готовый результат. Если все операции, выполняемые таким вычислителем, одинаковые, то его можно программно настроить на требуемые операции. Если во время вычислений необходимо повторять последовательность команд, то для «ведомых» вычислительных узлов можно сделать специальные буферы (стеки) команд.

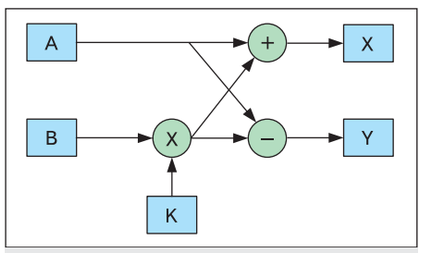

Рис. 11. Алгоритм расчета «бабочки»

А если речь идет не о вычислениях как таковых?

Обратимся к «классическим» ASIC-микроконтроллерам. Ведь в их состав с самого начала входят специализированные узлы, выполняющие работу в реальном времени и разгружающие от этой рутины их процессор. И такие блоки имеются даже у самых «мелких» микроконтроллеров. О чем же идет речь? Конечно, об UART'e и таймере. Эти узлы являются программно-настраиваемыми узлами реального времени. Так вот, именно с этой точки зрения давайте и посмотрим на архитектуру «самодельного» микропроцессора. В его состав можно вводить дополнительные узлы, выполняющие обработку информации в реальном времени — это общепринято и известно. Главное, чтобы эти узлы программно настраивались и управлялись.

То есть, если говорить об архитектуре, надо твердо усвоить, что нельзя слепо копировать микроконтроллеры общего применения. Архитектура должна выбираться с учетом оптимизации вычислений и программирования.

Прогноз. Вентили, триггер, soft-процессор, что дальше? FPOA! А за ним и «море процессоров»

Теперь самое время немного заглянуть вперед: что ждет любителей нестандартных процессоров? Сопоставим следующие факты. Когда-то «атомом» разработки цифровой техники был вентиль, упакованный по несколько штук в микросхему. Потом появились «молекулы» — триггеры, счетчики и т. д. Следующий шаг: более крупные цифровые блоки — микроконтроллеры. Потом, как мы знаем, произошел виток эволюции, и появились FPGA. Триггеры и счетчики были «понижены в звании» и превращены в «атомы», расположенные внутри FPGA. А на роль «молекул» в FPGA теперь «назначены» микроконтроллеры. Что должно произойти при следующем витке эволюции? Без сомнения, процессоры «целиком» или их крупные составные части должны превратиться в «атомы», расположенные внутри чего-то подобного FPGA. Только теперь это уже не море вентилей, а море объектов. Так что давайте знакомиться — FPOA.

И вот оно, новое поколение, мечта строителей «самодельных» процессоров — MOA1400D 1 GHz Field Programmable Object Array™. Производитель — фирма MathStar (w ww.mathstar.c om). Необходимо сразу сказать, что фирма MathStar позиционирует свои микросхемы в основном для рынка устройств обработки видео.

В таблице 4 приведено описание «начинки» такой микросхемы.

Таблица 4. Ресурсы, доступные в микросхеме FPOA — MOA1400D

Название ресурса	Количество	Рабочая частота	Разрядность	Производительность
ALU	256 объектов	до 1 ГГц	16 бит+control logic	Одна операция за clock
Register File	80 объектов	до 1 ГГц	128 байт+80 бит тег	Одна операция за clock
MAC	64 объектов	до 1 ГГц	16×16 бит умножитель	Одна операция за clock
Internal RAM	12 банков	до 500 МГц	768×76 бит	5,7 Гбайт/с
External RAM	2 интерфейса	до 200 МГц DDR	36 бит RLDRAM II	1,8 Гбайт/с
GPIO	4 банка	до 100 МГц	44 вывода на банк	176 выводов
High Speed I/O Transmit	1 порт	18–400 МГц DDR	16+1 бит LVDS	12,8 Гбайт/с
High Speed I/O Receive	1 порт	250–400 МГц DDR	16+1 бит LVDS	12,8 Гбайт/с

Фирма-производитель предлагает разработчикам комплект программных инструментов и язык программирования — OHDL (Object Hardware Description Language), ориентированный на описание ресурсов микросхемы. Здесь мы не будем останавливаться на тонкостях архитектуры микросхемы FPOA и детальном описании методики проектирования. Отметим главное. При такой методике разработки сокращается время, затрачиваемое на проектирование, так как разрабатываются более крупные «куски». Трассы для сигналов прокладываются не между ячейками с триггерами, а между объектами. А это значит, что все трассы внутри объектов уже оптимизированы по скорости, и потому вмешательство разработчика не требуется.

То, что делает фирма MathStar, — это набор «полуфабрикатов». Если же следовать нашему экскурсу в историю, то должны появиться микросхемы, где «атомом» будет не часть процессора, а сам микропроцессор целиком. И ведь мы оказались правы! Такие микросхемы уже производятся. Это pc102-100/80, которые выпускает фирма picoChip. Микросхемы содержат 308 процессоров, интерфейс памяти и т. д. За документацией об этих микросхемах можно обратиться на сайт **ht tp://ww w.picochip.c om**.

После того, как выбрана глобальная архитектура, мы можем перейти к следующему вопросу: а какой именно процессор можно сделать? Надо определить, какой тип процессора нам нужен.

Выберем тип процессора в соответствии с тем, что приводит Кен Чапман (Ken Chapman) в своей статье [13].

Чапман предлагает к рассмотрению три основных архитектуры процессоров, имеющих память данных и АЛУ. В основу этих архитектур могут быть положены регистры, аккумулятор или стек.

Архитектура на основе регистров

Блок-схема такого процессора приведена на рис. 12. Достоинства этой архитектуры — в простоте программирования. И чем больше регистров имеет процессор, тем меньше выполняется пересылок данных при решении задач, а, соответственно, выше производительность процессора. Но, вместе с тем, чем больше регистров мы хотим использовать в процессоре, тем больше нам нужно затратить аппаратных ресурсов. Для управления записью в регистры нам необходимы будут дешифратор и узлы схемы, вырабатывающие сигналы разрешения для записи данных в регистр. Увеличение числа регистров должно затрагивать и разрядность полей операндов в коде команды. Если же машина имеет поле команды с тремя операндами, то увеличение числа регистров должно сказаться на увеличении каждого из полей операндов. Но это только небольшая часть затрат аппаратных ресурсов. «Львиная доля» затрат состоит в том, что данные, считываемые из этих регистров, необходимо промультиплексировать и вывести на выходную шину. При увеличении числа регистров увеличивается и число последовательно включенных вентилей в мультиплексоре. При этом тратится не только большое количество вентилей, но еще и большое число внутренних шин в кристалле, а это очень «дорогой» ресурс.

Архитектура на основе аккумулятора

Блок-схема этого процессора приведена на рис. 13. Обычно аккумулятор связан с регистрами или памятью, для того чтобы получать и передавать различные переменные. Преимущество структуры, выполненной с использованием аккумулятора, состоит в том, что можно в набор команд ввести те, которые оперируют только с аккумулятором, а следовательно, адрес одного из операндов уже будет определен в коде операции, и этот адрес может не содержаться в поле операнда. Рассмотрим это более подробно. Если мы хотим иметь 3 операнда для того, чтобы выполнить нечто вроде «A+B=C», то нам необходимо закодировать следующие действия. Первое действие — в коде операции должно быть сказано о том, что выполняется операция между суммированными данными, находящимися в регистрах, и ее результат будет помещен в один из регистров. И далее в полях операндов, в каждом из таких полей, необходимо дать адрес, соответствующий конкретному регистру. При таком подходе, в самом общем случае, аккумулятор может иметь свой собственный адрес. И, соответственно, процессор может иметь множество аккумуляторов. Но, если в поле кодов операций есть свободная кодовая комбинация, то операции с аккумулятором можно вынести в отдельную группу команд. Такая архитектура будет наиболее приемлема в «бюджетных» моделях процессоров, имеющих только один или два аккумулятора. При этом адрес аккумулятора уже будет учтен не в поле операнда, а в самом коде операции. Это позволит сократить разрядность полей операндов в коде команд. Неудобство этой архитектуры хорошо известно тем, кто поработал с такими процессорами, начиная от Intel-8080.

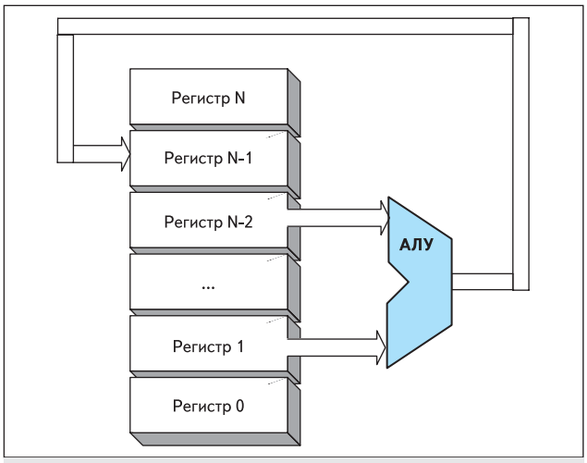

Рис. 12. Блок-схема процессора на основе регистров

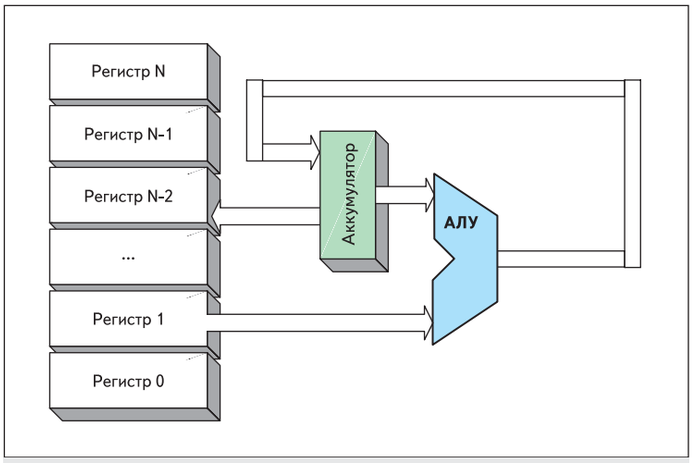

Рис.13. Блок-схема процессора на основе аккумулятора

Имея в своем распоряжении только один аккумулятор и ограниченный набор регистров, программист вынужден был каждую подпрограмму начинать с команд пересылок, необходимых для сохранения результатов работы предыдущей части программы.

Архитектура на основе стека

Такая архитектура (рис. 14) представляет собой область памяти, адресуемую как стек. Текущее положение вершины стека обозначает указатель. Данные в АЛУ могут быть считаны из вершины стека — TOS (top of stack) и из следующей за вершиной ячейки — NOS (next of stack). Данные из АЛУ помещаются в вершину стека. Кроме этого, стековая машина должна поддерживать команды обмена данными между TOS и NOS. А для того чтобы работать с вызовами подпрограмм, такая архитектура должна иметь еще и стек возвратов.

Стековая архитектура имеет наименьший объем задействованных аппаратных ресурсов. Для работы со стеком не надо указывать его адрес в поле операнда, поэтому часто можно встретить термины «0-операндная» или «безадресная» архитектура. Стековые машины имеют очень плотно упакованный код команд и за счет этого могут выполнять несколько примитивных действий одновременно, если, конечно, такие действия не противоречат друг другу. Например, записывать данные в стек и порт ввода/вывода. Но ведь на самом деле не все так однозначно. К сожалению, стековые машины не получили большого распространения из-за непривычного синтаксиса, применяемого при написании программ. Кроме того, для хранения глобальных переменных стековые машины часто дополняют регистрами. Так что количество их преимуществ над другими типами архитектур при этом несколько сокращается.

Теперь следующий этап — надо определить число операндов

«А и Б сидели на трубе, А упало, Б — пропало, что осталось на трубе?». Эта детская «считалка» сейчас нам поможет разобраться сначала с операндами, а потом и с pipeline, то есть с «трубой» в дословном переводе, а по-нашему — с конвейером.

Обратимся, для начала, к архитектуре процессора на основе регистров. Итак, чем больше регистров, тем легче делать вычисления. Причем хочется совершать их так, чтобы считывание команды из памяти команд и сама операция и запись результата вычисления производились в один такт. Конечно, это могут быть разные такты, но суть именно в том, чтобы и считывание команды, и ее выполнение занимало одинаковое время. Например, для выполнения операции «A+B=C» желательно иметь 3 регистра. Если же исходить из поставленной задачи, и нам необходимо выполнять операцию «A+B+C=D», то в этом случае нужно иметь уже 4 регистра.

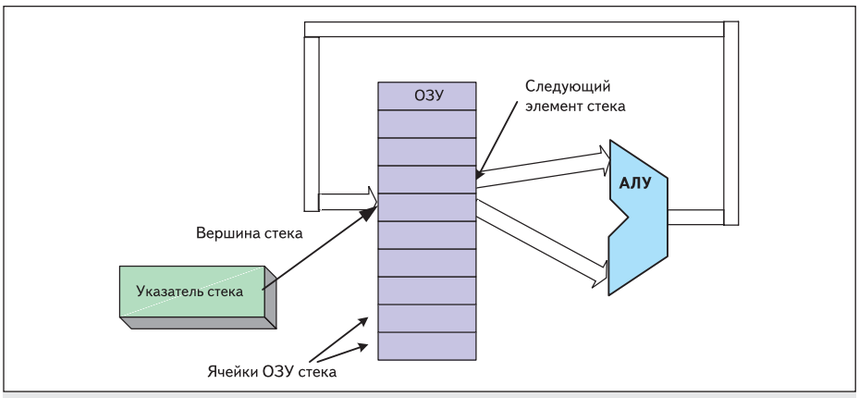

Рис. 14. Блок-схема процессора на основе стека

Далее необходимо определить, сколько же регистров понадобится для вычислений наиболее часто встречающихся алгоритмов, либо для вычисления самых критичных по времени. Если для критичных по времени участков все более или менее понятно (надо, значит надо!), то вот для других вычислений можно рассмотреть несколько вариантов. С одной стороны — минимум регистров, а следовательно, минимум и ресурсов, но требуется больше команд, дольше будут идти вычисления и память команд будет большего объема . И еще необходимо отметить, что в этом случае будет проще ассемблер, и, возможно, это вызовет меньше ошибок при его разработке. С другой стороны — больше регистров, а следовательно, больше будет задействовано ресурсов, но потребуется меньше команд, вычисления будут произведены быстрее, а память команд будет более компактной. И в этом случае ассемблер будет несколько сложнее, чем в предыдущем. Здесь, как мы и договаривались, были поставлены задачи по определению числа регистров, необходимых только для решения самой задачи. Но, кроме этого, может быть, понадобятся еще дополнительные регистры, необходимые для работы встроенной операционной системы и для смены контекста.

Какой вывод можно сделать, исходя из всех этих рассуждений? Однозначных рекомендаций и каких-либо формул тут нет. Все определяется конкретной задачей, микросхемой и волей разработчика. Что касается задачи и числа регистров, то тут должно быть уже все понятно. А вот по поводу микросхемы можно сделать некоторые дополнения. Разрабатывая новый процессор, специалист, как правило, с самого начала разработки ориентируется на какую-либо аппаратную платформу. И, выбирая архитектуру, можно рассмотреть аппаратные ресурсы, которые уже заложены в той серии микросхем, на которую будет ориентирована их разработка. Также необходимо учесть и библиотеки, поставляемые изготовителем микросхем. Первый этап работ позволяет выбрать архитектуру процессора и тщательно ее протестировать. А на заключительном этапе работ можно сделать более четкие выводы о том, сколько свободных ресурсов остается в микросхеме. И, если этот остаток ресурсов превышает технологический запас, то это позволяет разработчику не оставлять такие «излишки» ресурса незадействованными, а добавить их к разрабатываемому процессору.

В заключение можно сделать следующие предположения. Использование большого числа регистров, выполненных на триггерах, для процессора в FPGA — слишком расточительное занятие, особенно для 32-битных моделей. В этом плане предпочтительнее использовать блоки встроенной памяти как регистровые файлы, у которых мультиплексор выходной шины уже встроен. Примером здесь может служить процессор Nios. О нем будет сказано далее, в разделе, где описано переключение контекста.

Сколько надо иметь стеков?

В отличие от предыдущего пункта по поводу числа стеков автор может дать более четкую рекомендацию. Стеков должно быть два! Один — стек данных, другой — стек возвратов (рис. 15).

Если нам надо спроектировать «бюджетный» вариант процессора, то даже в этом случае применение двух стеков будет оправданным. Для такого процессора как запись в стек, так и извлечение данных из стека возвратов должны происходить аппаратно, и эти действия скрыты от программиста. Причем данные, занесенные в стек данных, «не портятся». И передача параметров через стек от одной подпрограммы к другой значительно упрощается. При такой архитектуре удается сэкономить на числе регистров, что повышает быстродействие и сокращает ресурсы, требуемые для реализации процессора. Мало того, используя команды работы со стеком, мы можем одновременно с этим выполнять еще какие-либо действия.

Что касается стека данных, то он должен иметь разрядность, равную разрядности шины данных процессора. Соответственно, стек возвратов должен иметь разрядность, равную разрядности шины адреса процессора. И тут вся хитрость

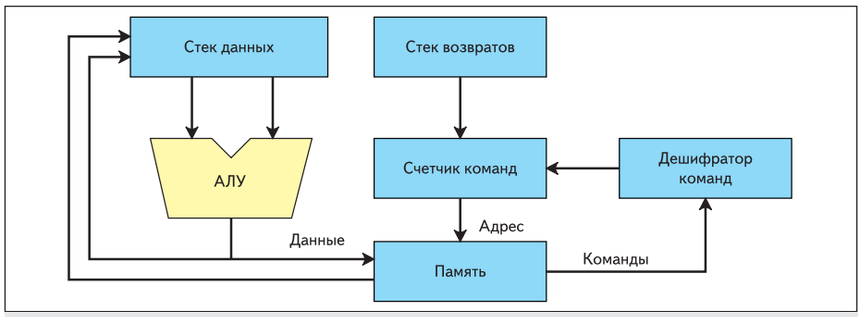

Рис. 15. Блок-схема процессора с двумя стеками

заключается в том, что даже для «бюджетных» вариантов процессора шина адреса имеет меньшую разрядность, чем шина данных. Да и глубина этих стеков может быть разной. Представим, что нам нужно передавать через стек не более 4 слов данных по 32 бита. И мы хотим иметь не более 10 вложений подпрограмм, при адресной шине в 12 разрядов. При использовании одного стека мы должны задействовать (10+4)×32 = 448 ячеек для хранения информации. При использовании двух стеков — (12×10+4×32) = 248. То есть почти вдвое меньше. А поскольку для стека необходимо использовать либо триггеры из логических ячеек, либо быстрые блоки памяти с низкой латентностью, то такой выигрыш имеет довольно большое значение. Конечно, некоторую часть из сэкономленных ресурсов придется потратить на мультиплексор, но в целом такое решение имеет определенные преимущества. Поэтому отдельный стек возвратов — это более удобное и экономное решение.

Как будем грузиться — отлаживаться?

Гарвардская или фон-неймановская архитектура? С ПЗУ — надежней. Но есть проблемы с загрузкой. С ОЗУ — намного проще, но ненадежно…

Теперь давайте заглянем в «начинку» процессора со стороны памяти. И если говорить более точно, то сейчас в первую очередь нас будет интересовать память команд. Кстати, давайте уточним, что кроме памяти команд бывает еще память данных. Здесь, и до особого упоминания, мы будем иметь в виду «обычную» память — с произвольной выборкой. Как правило, память — это две шины: адреса и данных. Кроме этих двух шин мы еще имеем набор сигналов управления. Но в FPGA память устроена несколько по-другому. Это вызвано тем, что в FPGA — двунаправленные шины с третьим состоянием, так широко распространенные «снаружи» микросхемы, совершенно не прижились «внутри». Поэтому, говоря о памяти, расположенной на кристалле, следует рассматривать шину данных для записи информации в память и шину данных для считывания информации из памяти. Более того, один и тот же блок памяти может быть сконфигурирован и как двухпортовая память. Так как же мы подключим память команд?

В соответствии со сказанным выше, давайте представим и весь наш процессор как некий черный ящик с тремя шинами. Первая, та, что будет выходить из черного ящика, — это шина адреса команды. С ней все просто. Ее надо подключить на вход адресов памяти команд. Осталась еще пара шин — входящие и выходящие данные. Применяем самое быстрое решение — соединяем «выход на вход». Все очень компактно и просто. Что получаем? Классическую архитектура фон неймана. Общая шина для памяти команд и данных. Такая машина может читать и писать в память команд и данных, может сама загружать новые программы в память.

Но есть у нее и недостатки:

1. В случае сбоя может затереть кусок кода программы.
2. Не может одновременно читать код команды и данные.
3. Если память не выполнена как двухпортовая, то не может одновременно читать код команды и писать данные.

Такая организация памяти во встроенных устройствах применяется довольно редко. Редко, потому что нельзя одновременно считывать код следующей команды и делать запись в память данных. Следовательно, при обращении к памяти данных машина будет замедляться. И, кроме этого, такой вариант применяется не часто еще и потому, что для встроенного узла управления нет необходимости менять память программ (конечно, кроме режима начальной загрузки). В ряде случаев можно использовать память на кристалле как кэш, но и тогда заменой содержимого кэша занимается отдельный специализированный блок — контроллер кэша.

Одна из поставленных задач в ТЗ — это максимальная производительность. А в варианте с архитектурой фон неймана мы ее реализовать не сможем. Поэтому сейчас мы постараемся сделать наш проект по-другому. Давайте разделим шины, по которым считываются коды команд, и те шины данных, по которым мы их читаем. Шины данных оставим «так как есть», а шину кодов команд выделим отдельно и подключим к дешифратору кодов команд и затем к АЛУ. Теперь мы имеем возможность получать коды команд, выполнять эти команды, но теперь мы уже не имеем возможности писать в эту область памяти — то есть в память команд. Что мы имеем в этом случае? Классическую гарвардскую архитектуру. Шины данных — раздельные, поэтому невозможно затереть память команд, и это несомненный плюс. Обе шины работают одновременно: в то время как по шине команд передаются команды, по шине данных производится прием/передача данных. И это тоже плюс.

Все это хорошо, и просто и надежно… Да вот только есть один вопрос, который требуется обсудить здесь же. А как мы будем грузить память команд и отлаживать программу? Как было сказано выше, процессор не имеет доступа по записи в собственную память команд. Первое, что приходит в голову — применить тезис «Не может — и не надо». Перекомпилируем проект еще раз под новую «прошивку», загрузим все заново стандартными средствами, применяемыми для загрузки FPGA, — и готово. Но ведь так можно отлаживаться только в очень небольших проектах, когда время, затрачиваемое на компиляцию, мало. Для больших проектов, когда каждая компиляция занимает от получаса и до нескольких часов, такой подход невозможен.

Но ведь должны же быть и нетривиальные решения. Да, конечно. И одно из таких решений — это внешний загрузчик. Представим себе, что память программ выполнена как двухпортовая память. Один ее порт используется основным процессором, тем, который мы разрабатываем, а второй порт этой памяти подключен к загрузчику. В качестве загрузчика мы можем использовать как простенький статический автомат, так и внешний, по отношению к микросхеме FPGA, процессор. Внешний процессор, как известно, стоит значительно меньше, чем процессор в FPGA, и может использоваться сразу для нескольких задач. Во-первых, для загрузки самой FPGA, во-вторых, для загрузки памяти команд встроенного в FPGA soft-процессора. Третья функция, выполняемая таким вспомогательным процессором, — это контроллер интерфейса для связи с хостом при отладке или работе. Кроме того, используя узел контроля напряжения питания, внешний микропроцессор может работать как супервизор питания для soft-процессора. И, наконец, такой микроконтроллер может выполнять функции шифрования проекта, сохранения серийного номера, промежуточных данных в энергонезависимой памяти и т. д. В таком случае внешний процессор поддерживает интерфейс обмена с хостом и преобразует данные для обмена с soft-процессором в последовательный код. Например, в SPI или JTAG.

Можно ли обойтись без внешнего микроконтроллера и при этом иметь возможность загрузки команд? Да, конечно. Для этого нужно сделать режим работы с памятью переключаемым. На этапе загрузки сделать режим,

когда память программ доступна на запись и на чтение, а затем, после загрузки, блокировать такую возможность.

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://w ww.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, 3, 4, 5.
9. **ht tp://w ww.xilinx.c om/ipcenter/processor_central/picoblaze/ picoblaze_user_resources.ht m**
10. Каршенбойм И. Микропроцессор своими руками-2. Битовый процессор // Компоненты и технологии. 2003. № 6, 7.
11. Tomaszewski E. Explicitly Parallel RISC (EPRISC). **ht tp://w ww.opencores.o rg/articles.cgi/view/4**
12. **http://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, **w ww.analog.c om**
15. U17135EJ1V1UM00.pdf; V850E2 32-bit Micropro-cessor 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**