Микропроцессор своими руками – 2
Битовый процессор

Иосиф Каршенбойм

iosifk@narod.ru

Введение

В предыдущих статьях [1, 4] был приведен пример разработки микроконтроллера в FPGA. В зависимости от конкретных требований проекта, разработчик может выбрать ресурсы микропроцессора и систему команд. Но обычный набор команд ориентирован на обработку байтов или слов. Обработка информации на уровне отдельных битов обычно в таких микропроцессорах не поддерживается.

Данная статья представляет собой продолжение предыдущей статьи, как цикла о микропроцессорах, ориентированных на оптимальное выполнение задачи пользователя при значительном сокращении ресурсов кристалла. В статье приведено описание *битового процессора*, предназначенного для управления объектами, требующими только дискретного (не пропорционального) регулирования. Показано, что такой процессор может быть наиболее эффективным управляющим процессором для небольших систем управления или выполнять функции битового сопроцессора для процессора, оперирующего байтами или словами. Наиболее целесообразно применение данного управляющего процессора может быть в тех случаях, когда нет возможности применить стандартный микропроцессор с операционной системой реального времени или трудозатраты на программирование такой системы велики. Битовый процессор дает существенный выигрыш в быстродействии по сравнению со стандартным микропроцессором при одновременном упрощении программирования. Статья также содержит описание конвейера команд и методики разработки микропроцессора с конвейером команд.

Статья может быть интересна как студентам и начинающим разработчикам, так и профессионалам, которые имеют опыт разработки устройств, и в том числе и в FPGA, но, по тем или иным причинам, не имели возможности разрабатывать и использовать «встроенные в FPGA» микропроцессоры. Все файлы, приведенные в данной статье, будут, как и в предыдущем случае, доступны в Интернете на сайте автора www.iosifk.narod.ru.

Управляющий процессор

Эпоха бума микропроцессорных систем управления давно прошла и на смену ей пришла эпоха систем, требующих «модной» теперь DSP-обработки. Однако задачи управления дискретными объектами, не требующими DSP-обработки, не потеряла своей актуальности.

Если, например, объект управления описывается терминами «уставка», «большой расход», «клапан открыть», «не норма» и так далее, или объект управления представляет часть тракта приема-передачи данных и описывается терминами «соединение», «перезапрос», «начало кадра» и так далее, то для управления подобными объектами необходим совершенно «немодный» сегодня управляющий процессор, оптимизированный на решение задач булевой алгебры. Цель настоящей статьи — показать один из вариантов решения подобной задачи в FPGA. При разработке встроенного в FPGA микропроцессора задача обработки битов, находящихся в слове данных микропроцессора, может потребовать больших затрат как по аппаратным ресурсам, так и по времени на обработку задач управления битами данных. Однако данная задача легко решается в FPGA, если обработку битов выделить из общего потока управления и для этой задачи применить отдельный управляющий микропроцессор, оперирующий только битовыми данными. На примере управляющего микропроцессора, описываемого в этой статье, будут подробно описаны преимущества, появляющиеся при использовании конвейера команд.

Конечный автомат или микропроцессор?

В предыдущей статье [1] были приведены все аспекты применения статического автомата в FPGA. Позволю себе коротко повторить основной тезис данного пункта.

Чтобы сравнить микроконтроллер с конечным автоматом, необходимо сравнить трудоемкость работ.

Чтобы в новом проекте реализовать заданную последовательность действий, можно либо каждый раз заново создавать конечный автомат, либо взять уже готовый микроконтроллер, адаптировать его к заданным условиям, и, написав небольшую программу для выполнения последовательности команд, запустить. Причем написание программы для микроконтроллера намного проще написания и отладки конечного автомата на языках HDL.

Если рассмотреть применение статического автомата с точки зрения управления битовыми переменными, то можно задать только один вопрос: «Сколько ресурсов потребуется такому автомату при управлении хотя бы не сотней, а парой десятков дискретных параметров, если все эти параметры управляются по разным алгоритмам?» Ответ будет очевиден — реализовать такой автомат либо невозможно, либо нецелесообразно.

Преимущества микропроцессора, «встроенного в FPGA»

В предыдущей статье [1] были приведены аспекты применения «стандартного» микропроцессора в FPGA. Под термином «стандартный» здесь понимается микропроцессор, выполняющий операции регистр-регистр, регистр-память. Обработку данных такой процессор выполняет словами или байтами. Микропроцессор, имеющий «стандартный» набор команд, ориентирован на широкий круг задач пользователя. Примерами таких процессоров могут быть Nios [2, 3], MCS-51 или AVR. Ядра этих микропроцессоров могут быть встроены в FPGA.

Позволю себе очень коротко повторить основной тезис данного пункта. «Встроенные в FPGA» микропроцессоры и микроконтроллеры на их основе имеют главное преимущество перед обычными микроконтроллерами средней производительности: они абсолютно синхронны со всем остальным проектом, расположенным в этой же микросхеме. Если устройство, которое вы проектируете, работает в реальном времени и с большими потоками данных, которые вы должны извлекать из периферии и отдавать в периферию, то задача синхронизации становится достаточно серьезной.

Все быстрые «мелкие» микроконтроллеры работают асинхронно (относительно периферии в FPGA), и не имеют аппаратного входа «Готовность», поэтому они могут синхронизироваться с периферией только программно, а для программной привязки их к синхронному проекту в FPGA нужно, во-первых, несколько команд процессора, что займет несколько тактов синхрочастоты, во-вторых, это требует ресурса микросхемы FPGA и, в-третьих, занимает довольно много места на плате. Быстрые «крупные» процессоры имеют возможность аппаратной синхронизации по входу «Готовность», но дороги и занимают еще больше места на плате. Да и применение «крупного» процессора для небольших задач нецелесообразно. А это значит, что при том же быстродействии ядра процессора получится выигрыш по производительности в 2–3 раза.

Недостатки «внешнего» «стандартного» микропроцессора, и «стандартного» микропроцессора, «встроенного в FPGA».

К недостаткам «стандартного» микропроцессора, «встроенного в FPGA», можно отнести то, что в FPGA сложно организовать развитую систему команд. Увеличение числа команд микропроцессора, равно как и увеличение способов доступа к данным различной разрядности, приводит к значительным затратам ресурса в FPGA. Одно из таких «узких мест» — это обработка битовых банных. Блоки памяти, находящиеся в FPGA, с которыми оперирует микропроцессор, обычно организованы как байтные или словные. Таким образом, выделение бита из байта или слова требует достаточно сложной обработки, а прямое управление битами при такой организации памяти невозможно. Особенно это относится к тем применениям, где нет необходимости реализовывать мощный микропроцессор. Более того, обычно ядро встроенного стандартного микропроцессора уже содержит всю периферию, которую «положено» иметь данному микропроцессору. Однако, в том случае, если часть этой периферии не нужна в данном проекте, то она будет занимать ресурс кристалла, но не будет использоваться. Конечно, разработчик может выделить и убрать неиспользуемые периферийные блоки. Но это представляет собой дополнительную задачу, так как неизбежно встает вопрос верификации доработанного микропроцессора. А в том случае, когда мегафункция микропроцессора зашифрована или поставляется как файл списка связей, доработка микропроцессора под требования проекта может оказаться невозможной.

Если рассматривать задачи управления дискретными переменными, то, как в случае применения «внешнего» «стандартного» микропроцессора, то есть микропроцессора, находящегося вне FPGA, так и в случае применения «стандартного» микропроцессора внутри FPGA, здесь необходимо дополнительно рассмотреть два дополнительных пункта, первый из которых — это работа «стандартного» микропроцессора с командами обработки битовых данных. Если битовые переменные «упакованы» в слова данных байтного, двухбайтного или словного формата, то микропроцессор должен потратить несколько команд на то, чтобы выделить нужный бит из слова, поместить его в нужную позицию перед обработкой, и только потом произвести необходимую операцию с битом. Перед записью битовых данных в словную память также необходимо произвести операцию «упаковки» бита в слово данных. Конечно, можно использовать для хранения бита данных целое слово, но при этом снижается эффективность использования памяти. Второй пункт, на который также необходимо обратить внимание, это наличие в алгоритмах управления большого числа программных таймеров. При формировании алгоритмов все ветви технологического процесса обычно «перекрываются» программными таймерами. То есть «процесс должен завершиться через N секунд, а в противном случае делать…». Так например, процесс соединения двух абонентов по протоколу X25 может потребовать до 7 таймеров различной длительности. Таким образом, управляющий микропроцессор должен одновременно формировать сотни программных таймеров различной длительности от единиц миллисекунд до секунд и, возможно часов (в зависимости от задачи управления). Но, поскольку у реального микропроцессора число аппаратных таймеров ограничено, то выдержки времени обычно формируются программно, путем подсчета квантов времени, получаемых по прерыванию от одного аппаратного таймера. Это значит, что кроме прерываний от периферии и от системного таймера, микропроцессор будет тратить свой ресурс и на обработку прерываний от аппаратного таймера, а так же и на формирование десятков программных таймеров. В случае использования FPGA появляется возможность легко распараллелить процессы вычислений и выделить формирование программных таймеров в отдельный блок многоканального таймера. Такой блок будет представлять собой специализированный вычислитель, выполняющий счет, например, по 128 16-битным каналам.

В FPGA «мы пойдем другим путем»

Ресурс, который можно использовать в FPGA, может быть произвольно задействован по усмотрению разработчика. Именно поэтому абсолютно неважно, как и где будет производиться обработка битовых данных. И, если обработка битовых данных в составе главного управляющего микропроцессора представляет собой трудную и ресурсоемкую задачу, то, как было сказано во введении, данная задача легко решается в FPGA, если обработку битов выделить из общего потока управления и для этой задачи применить отдельный управляющий микропроцессор, оперирующий только с битовыми данными. Назовем такой микропроцессор *битовым процессором (БПр)*. В наиболее общем случае, когда необходимо производить обработку слов различной разрядности, битовый процессор может быть выполнен по отношению к главному управляющему микропроцессору как сопроцессор. В том случае, когда такой необходимости нет, и все данные могут быть представлены в виде набора битов, битовый процессор будет представлять собой по отношению к объекту управления автономный управляющий автомат, полностью выполняющий все функции управления данным устройством.

Рассматриваемый в данной статье БПр будет выполнять команды булевой алгебры и команды пересылки данных в памяти данных. Поскольку при изменении одного из входных данных X(i) реакцией системы может быть изменение одного выхода системы Y(k) или множества выходов системы Y(k..m), что определяется алгоритмом работы конкретного объекта, то с целью упрощения работы БПр произведем обработку полного набора выходных параметров системы — Y(). При этом нам будут не нужны команды ветвления и прерывания. БПр будет циклически проводить обработку выходных параметров системы Y(). Общее время на обработку всех выходных параметров будет зависеть от числа этих параметров и от числа операций, которые БПр будет выполнять при обработке каждого выходного параметра Y(i). При необходимости сократить время на обработку всех параметров системы можно разделить объект управления на несколько частей (если это возможно) и для каждой части объекта управления применить свой БПр.

Если необходимо сократить время на обработку одного или нескольких параметров системы, например для аварийных датчиков, то можно производить вычисление выходных значений для этих параметров несколько раз за один полный цикл обработки всего массива. Это достигается тем, что коды команд, по которым производится обработка «быстрых»

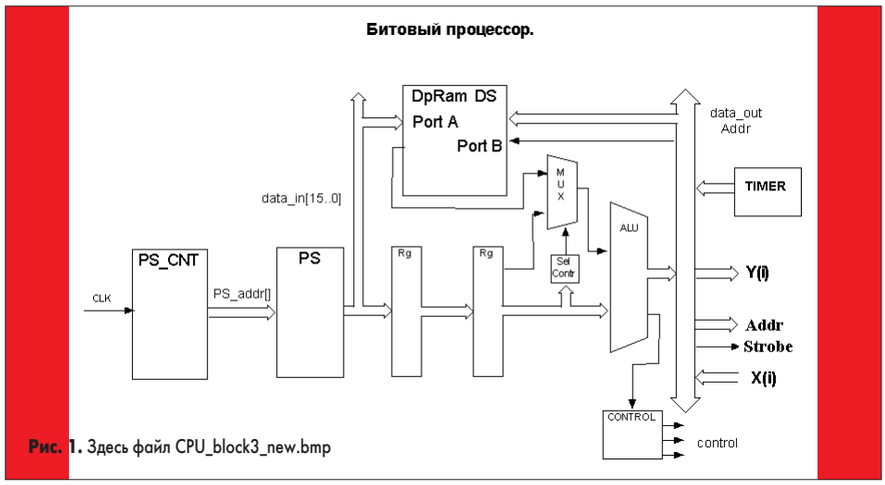

Рис. 1. Здесь файл CPU_block3_new.bmp

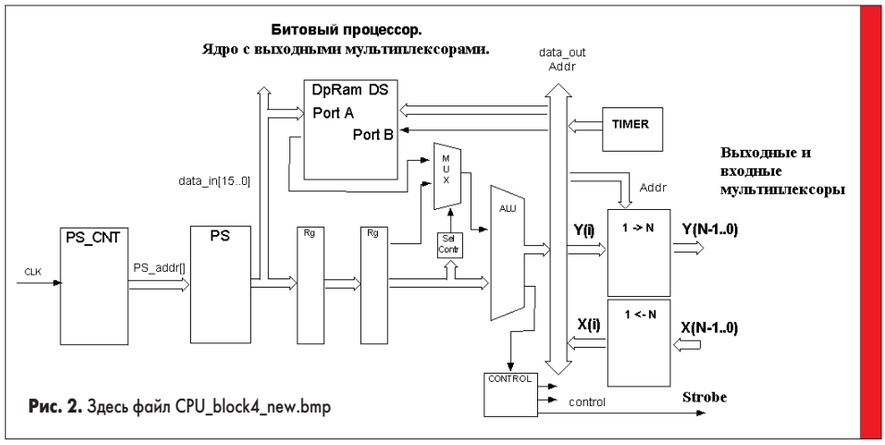

Рис. 2. Здесь файл CPU_block4_new.bmp

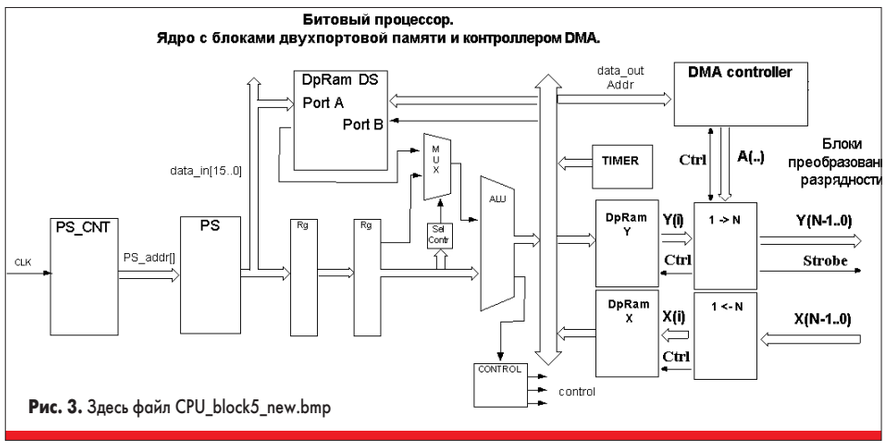

Рис. 3. Здесь файл CPU_block5_new.bmp

датчиков, несколько раз включаются в общий файл команд обработки всех датчиков.

Блок-схема ядра БПр приведена на рис. 1. Для работы ядра БПр необходима битовая память переменных, которые будут являться результатами промежуточных вычислений, и блок таймеров. Битовые переменные, необходимые для вычислений, удобнее всего расположить в блоке двухпортовой памяти — DpRam DS. Эта память является составной частью ядра процессора. Применение двухпортовой памяти здесь наиболее удобно потому, что по одному порту производится запись переменных при выполнении команд типа LDI — непосредственной записи в память, а по другому входу производится запись переменных, находящихся на аккумуляторе по командам MOV. Таким образом, отпадает необходимость в применении мультиплексоров шин данных и сигналов управления для записи. Входные и выходные переменные передаются по двум шинам — Y(i) и X(i). Шины представляют собой набор сигналов адреса, битов данных и сигнала стробирования.

Число таймеров и их параметры должны соответствовать требованиям к конкретному объекту управления. Более подробно о блоке таймеров будет сказано далее.

Для варианта автономного использования ядра БПр в качестве основного управляющего устройства его связь со входами и выходами будет осуществляться напрямую через входные и выходные мультиплексоры. В таком случае внешние по отношению к микропроцессору шины будут представлять собой набор входных и выходных переменных и сигнала стробирования. Вариант такого БПр приведен на рис. 2. Такое применение БПр требует минимальных затрат ресурсов, но предъявляет высокие требования по скорости ввода и вывода данных. Наиболее целесообразно применять такую структуру управляющего устройства в том случае, если сам объект управления находится в том же кристалле FPGA.

Для варианта использования ядра БПр как сопроцессора может быть применено включение БПр через блоки двухпортовой памяти.

Вариант исполнения БПр, работающего через блоки двухпортовой памяти, приведен на рис. 3. Поскольку блоки памяти, имеющиеся в FPGA, не позволяют иметь разную разрядность по двум портам, то будет целесообразно применить блоки памяти с разрядностью 1 бит. Переход от битовой двухпортовой памяти к байтовой или словной шине производится на блоках преобразования разрядности, представляющих собой микропрограммные автоматы, которые «собирают» в байт или в n-разрядное слово n битовых циклов. Это позволяет соответственно понизить скорость обмена по n-разрядной шине.

Если необходимо применить ядро БПр для автономного использования, как основного управляющего устройства, в том случае, когда нет возможности выполнить высокие требования по скорости ввода и вывода данных, то целесообразно использовать n-разрядные внешние шины. При этом, кроме блоков преобразования разрядности в устройство должны также входить два канала DMA, которые будут принимать входные n-разрядные данные и передавать, соответственно, n-разрядные данные. В наиболее общем случае разрядность входной и выходной шин может быть различной.

Несколько слов о блоке таймеров

В данном случае реализация таймеров подробно рассматриваться не будет. Простейшей реализацией многоканального блока таймеров может быть микропрограммный автомат, выполняющий только несколько команд, а именно команду загрузки аккумулятора, команду декрементирования аккумулятора, команду сохранения результата в памяти и команду записи в поле памяти данных микропроцессора бита флага, сообщающего процессору об окончании счета в конкретном канале.

Блок таймеров состоит из счетчика каналов, блока памяти, регистра-аккумулятора, выходного регистра и автомата управления. Код адреса с выхода счетчика адреса поступает на вход блока памяти. Данные из памяти поступают на аккумулятор. На следующем такте, если значение кода данных в аккумуляторе не равно нулю, то производится декремент аккумулятора. И, далее, на следующем такте, новое значение данных записывается обратно в память по адресу этого же канала. После чего, автомат управления дает команду на обработку следующего канала таймера. В том случае, если значение кода

данных в аккумуляторе становится равным нулю, то блок управления устанавливает в собственном поле памяти по адресу данного канала признак окончания обработки данного канала и производит запись бита данных в поле памяти данных микропроцессора, соответствующее расположению битовых переменных для блока таймера.

Запуск таймера на счет производится так же по команде процессора, при записи соответствующего бита в поле памяти блока таймеров. Обычно при разработке алгоритма программы становится известно сколько и каких временных таймеров необходимо для работы конкретного устройства, поэтому для инициализации таймеров используют несколько значений выдержки времени. Весь массив таймеров разбивается на группы и каждой группе таймеров назначают свою выдержку времени. Каждый таймер приписывается к своей ветви технологического процесса, поэтому при работе микропроцессора не возникает необходимость перезагружать таймер различными значениями данных, соответствующих разной выдержке времени для таймера. Такой подход значительно упрощает программирование, уменьшает число команд, а следовательно, увеличивает быстродействие микропроцессора.

На вход блока таймеров могут подаваться как системные тактовые импульсы, так и любые другие синхрочастоты (синхросигналы) в зависимости от требований проекта. Часть разрядов слова данных таймера может использоваться для выбора частоты счета для данного канала таймера. Многоканальный блок таймеров может быть выполнен во многих вариантах, но главное что хочется подчеркнуть еще раз, это то, что вычисление выдержек времени происходит в нем по всем каналам квазиодновременно и автономно, и не занимает ресурс микропроцессора.

Для примера представим, что мы можем выделить для реализации таймеров 2 блока памяти в микросхеме APEX или 1 блок в микросхеме ACEX. Тогда мы получим массив из 128 16-разрядных таймеров или 256 8-разрядных таймеров, что будет достаточно для большинства задач.

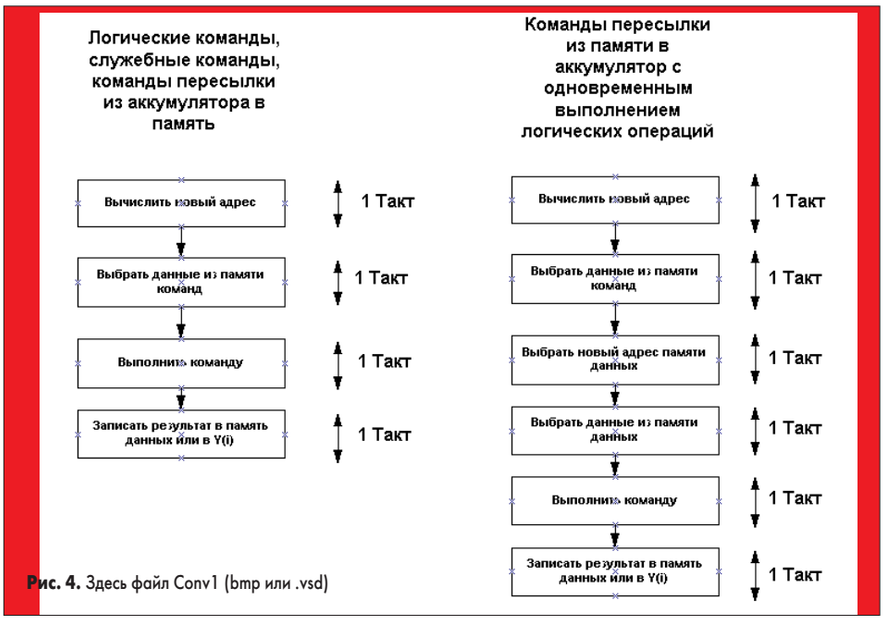

Рис. 4. Здесь файл Conv1 (bmp или .vsd)

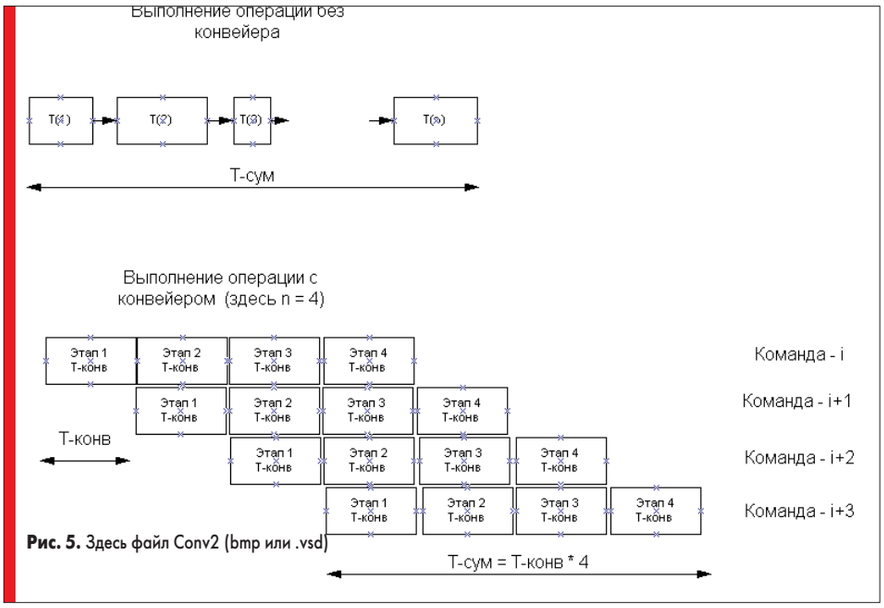

Рис. 5. Здесь файл Conv2 (bmp или .vsd)

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.

Последовательность обработки данных в микропроцессоре приведена на рис. 4, 5.

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, код команды поступает в ALU, где дешифрируется и где вырабатываются управляющие воздействия для обработки данных, находящихся в ALU, в памяти или в регистрах микропроцессора. Последним этапом происходит собственно то действие, которое призвана выполнить данная команда, например, переслать результат вычислений из ALU в память или в регистр, взвести триггер признака и т. д. Рекордсменом среди команд по длительности выполнения является команда непосредственной загрузки аккумулятора данными из памяти данных по адресу, который прочитается в данной команде. В таком случае, ко всем затратам времени, приведенным выше, добавится время на извлечение данных из памяти данных.

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

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

где ***T-сум*** — суммарное время задержки, ***T(1)… T(n)*** — времена задержки на 1..n-ых 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 часть 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)… 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*** умно-

женное на ***n***. Но конвейер обрабатывает данные потоком, поэтому каждая следующая команда будет выходить из конвейера через время ***T-конв***, что значительно меньше, чем ***T-сум***.

А где же та ложка дегтя, которая портит бочку меда? Да вот она — 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 быстродействие процессора. Именно может снижаться, потому что разработчики ядра ARM9 (см. систему команд процессора NIOS [2, 3]) делают в данной ситуации следующий ход: они «переносят» 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 последовательно 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.

В новом варианте пишется:

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

А выполняется такая последовательность команд так: 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 *команда 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.

В данном конкретном случае с БПр ситуация такова: либо надо вводить команды переходов, что усложнит дешифратор команд и, соответственно увеличит время на обработку каждой команды, либо вообще отказаться от команд перехода. В таком случае мы получаем наиболее простое устройство, как в аппаратном плане, так и в плане программирования. При таком подходе обработка производится циклически, последовательно для всех выходных параметров системы. Поскольку нет команд процессора, требующих перезагрузки очереди команд процессора, то появляется возможность организовать эффективный конвейер требуемой глубины. После выполнения всех команд пользователя БПр должен перейти к первой команде и снова начать поочередное выполнение команд пользователя. Для такого перехода введем команду SKIP, обозначающую, что все команды пользователя выполнены, дальше в поле памяти команд кодов операций нет и БПр должен перейти к команде по адресу 0.

Выберем команды, отвечающие нашим задачам

Опишем группы команд, которые должен исполнять микропроцессор.

1. Служебные команды:

NOP	Нет операции
SKIP	Дальше в поле памяти команд кодов операций нет

2. Группа команд пересылки из аккумулятора в память:

MOV Acc, [Mem]	Запись содержимого аккумулятора в память
MOVI Acc, [Mem]	Запись инверсного содержимого аккумулятора в память
MOVCL Acc, [Mem]	Запись содержимого аккумулятора в память и очистка аккумулятора
MOVICL Acc, [Mem]	Запись инверсного содержимого аккумулятора в память и очистка аккумулятора

3. Группа команд пересылки из памяти в аккумулятор с одновременным выполнением логических операций:

MOV_AND [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «И»
MOV_NAND [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «НЕ-И»
MOV_OR [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «ИЛИ»
MOV_NOR [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «НЕ-ИЛИ»
MOV_XOR [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «Исключающее ИЛИ»
MOV_XNOR [Mem], Acc	Запись данных из памяти в аккумулятор с выполнением операции «НЕ-Исключающее ИЛИ»

4. Логические команды:

INV Acc	Инверсия аккумулятора
SET Acc	Загрузить в аккумулятор 1

5. Команды непосредственной записи в память:

LDI 0, [Mem]	Записать в память 0
LDI 1, [Mem]	Записать в память 1

Для работы битового микропроцессора, такого набора команд будет достаточно.

Определим поля команд

Для работы битового процессора необходимо только два поля памяти: одно поле для кода операции и другое — для адресации памяти.

Поле кодов операций имеет разрядность 4 бита, что позволяет иметь 16 команд. Поскольку нам необходимо иметь только 16 команд, то такое поле кодов операций нас вполне устраивает. Очевидно, что для данного набора команд самое большое поле требуется для команды пересылки из памяти в аккумулятор и из аккумулятора в память. Разрядность поля для адресации памяти задается исходя из требований к объему памяти. С другой стороны, мы можем определить область памяти, которую будет адресовать процессор, если мы зададимся общей разрядностью шины памяти, например 16 бит и вычтем из нее четыре бита на поле КОП. Оставшиеся 12 бит памяти позволят адресовать 2^12 бит.

Команды записи из аккумулятора в память и команды чтения из памяти в аккумулятор будут выглядеть так:

15....КОП — 4 бита...12	11...Mem — 12 бит.. 0

Остальные команды будут выглядеть так:

15....КОП — 4 бита...12	

Определим коды операций команд

После того, как поля команд определены, и мы знаем разрядность поля кодов операций, можно определить коды операций команд.

Наиболее просто определить код команды NOP — здесь нет никаких требований, кроме одного — этот код команды должен быть на входе ALU при получении им сигнала «Сброс». Поскольку схема формирования адреса и память команд выполнены по синхронной схеме, то после сброса на вход АЛУ код команды, заданный пользователем, поступит только на том такте синхрочастоты, когда данные из памяти команд, соответствующие первой выбранной команде, пройдут через весь конвейер команд, а на первом такте АЛУ получит «пустой» код, который мы представим как код команды NOP. А так как после сброса в выходном регистре блока памяти будет находиться «00», то выберем этот код, как код команды NOP. Ко всем остальным командам никаких дополнительных требований нет, поэтому коды команд можно выбрать произвольно.

Тогда шестнадцатеричные коды операций команд будут такие:

0 — NOP
1 — MOV Acc, [Mem] — из аккумулятора в память
2 — MOVI Acc, [Mem] — из аккумулятора в память с инверсией
3 — MOVCL Acc, [Mem] — из аккумулятора в память с очисткой аккумулятора
4 — MOVICL Acc, [Mem] — из аккумулятора в память с инверсией и с очисткой аккумулятора
5 — MOV_AND [Mem], Acc — из памяти в аккумулятор с выполнением операции «И»
6 — MOV_NAND [Mem], Acc — из памяти в аккумулятор с выполнением операции «НЕ-И»
7 — MOV_OR [Mem], Acc — из памяти в аккумулятор с выполнением операции «ИЛИ»
8 — MOV_NOR [Mem], Acc — из памяти в аккумулятор с выполнением операции «НЕ-ИЛИ»
9 — MOV_XOR [Mem], Acc — из памяти в аккумулятор с выполнением операции «исключающее ИЛИ»
A — MOV_XNOR [Mem], Acc — из памяти в аккумулятор с выполнением операции «НЕ-исключающее ИЛИ»
B — INV Acc — инверсия аккумулятора
C — SET Acc — установить аккумулятор в «1»
D — SKIP — пропустить остальное поле памяти
E — LDI 0, [Mem] — записать в память «0»
F — LDI 1, [Mem] — записать в память «1»

Примеры кодов команд

Команда NOP будет иметь код 0000, MOV Acc, 234 — команда записи содержимого ак-

кумулятора в память по адресу 234 будет иметь код команды — 1234, а MOV_NOR 9, Acc — команда пересылки данных из адреса 9 в аккумулятор с выполнением операции NOR будет иметь вид 8009 и т. д.

Определим требования к программному обеспечению

Одним из основных требований к разработке встроенного ПО для микроконтроллеров является требование о необходимости инструментального ПО. Обычно к таким программным средствам относят различные редакторы, ассемблеры, компиляторы и т. д. Для более сложных проектов могут потребоваться симуляторы работы программы, языки высокого уровня, операционные системы и т. д.

Что касается редактора, то существуют такие редакторы как EditPlus2 (**http://www.editplus.com**), Prisma, и другие редакторы текста, позволяющие выделять ключевые слова по списку, формируемому пользователем.

Для данного класса процессоров необходим свой язык высокого уровня — язык, оперирующий в терминах булевой алгебры.

Такой компилятор может иметь в качестве переменных набор следующих параметров:

X(i) — значение переменной, соответствующее входному сигналу с номером «i»;

Y(i) — значение переменной, соответствующее выходному сигналу с номером «i»;

T(i) — значение переменной, соответствующее сигналу ячейки таймера с номером «i»;

V(i) — значение внутренней переменной, имеющей номер «i».

В зависимости от конкретной задачи пользователя возможно задать и большее количество переменных для компилятора.

Для обозначения логических операций, выполняемых над переменными, можно принять любые стандартные символы, например те, что приняты в AHDL или в VHDL.

Текст программы может выглядеть так:

Y25 = X3 OR (NOT T5) AND V43;

V90 = X1 XOR Y100; и т. д.

Компилятор преобразует строки программы команд в машинные коды и формирует файл инициализации памяти команд, совместимый с ПО MaxPlus. К командам, задаваемым пользователем, компилятор добавляет в конец файла код команды SKIP, в том случае, если объем файла кодов команд меньше, чем полный объем памяти команд.

Поскольку описываемый здесь проект посвящен только описанию микропроцессора, то дальнейшее описание компилятора будет опущено.

Окончание следует.

Литература

1. Каршенбойм И. Микропроцессор своими руками // Компоненты и технологии. 2002. № 6–7.
2. Каршенбойм И. Микропроцессор для встроенного применения Nios. Система команд и команды, определяемые пользователем // Компоненты и технологии. 2002. № 8–9.
3. Каршенбойм И. Микропроцессор для встроенного применения Nios. Конфигурация шин и периферии // Компоненты и технологии. 2002. №2–5.
4. И. Каршенбойм, Н. Семенов. Микропрограммные автоматы на базе специализированных ИС // Chip News. 2000. № 7.
5. И. Каршенбойм, К. Паленов. «Встроенный» логический анализатор — инструмент разработчика «встроенных» систем // Схемотехника. 2001. № 12.