Проектирование конфигурируемых процессоров на базе ПЛИС

Илья ТАРАСОВ,
к. т. н.
tile@kc.ru

Долгое время программируемые логические интегральные схемы выступали в качестве дополнения к микропроцессору, обеспечивая эффективное выполнение вспомогательных операций и определенный запас гибкости при построении цифровых интерфейсов. Однако в последнее время улучшение технических характеристик ПЛИС позволяет им приобретать все большую функциональность процессоров, превращаясь из «рассыпной логики-на-кристалле» в «систему-на-кристалле». Данная статья посвящена вопросам проектирования конфигурируемых, или софт-процессоров, которые способны придать проекту на базе ПЛИС все элементы стандартной микроконтроллерной системы, включая возможность программирования полученного устройства с помощью обычных языков высокого уровня.

В последнее время в технических публикациях все чаще и чаще поднимается вопрос об использовании ПЛИС для разработки не просто цифровых систем, а систем-на-кристалле — то есть устройств, наряду с прочими цифровыми узлами содержащих на кристалле и процессор. В качестве наглядного примера можно привести [1], где описаны возможности мощных FPGA семейств Virtex-II Pro и Virtex-4 FX фирмы Xilinx, которые содержат на кристалле аппаратное ядро процессора PowerPC 405. На базе данных ПЛИС можно построить мощную вычислительную систему с весьма богатыми возможностями, сопоставимыми с возможностями PC-совместимых настольных компьютеров.

Однако сочетание терминов «процессор» и «ПЛИС» не обязательно подразумевает реализацию высокопроизводительной вычислительной системы на базе устройства стоимостью в несколько сотен долларов. Обращаясь к прошлым публикациям в КиТ, можно выделить, например, серию статей, посвященных процессорам MicroBlaze и PicoBlaze фирмы Xilinx [2]. Эти проекты, по сути, представляют собой особым образом организованные цифровые устройства, выполняемые на базе обычных логических ячеек ПЛИС. Впрочем, для разработчика такие устройства выглядят как процессоры со всеми сопутствующими атрибутами: регистровой моделью, системой команд, возможностью выполнения программы, записанной в память, и т. д. Важно, что подобные проекты, реализованные с помощью языков описания аппаратуры, не привязаны к конкретной модели ПЛИС, а могут встраиваться в различные семейства в качестве устройства управления прочими цифровыми узлами. Достаточно интересен и тот факт, что разработка процессора как такового не обязательно превращается в чрезмерно сложный процесс, доступный только высококвалифицированным инженерам с богатейшим опытом работы. Например, в [3] приводилась аналогичная методика разработки оригинального процессора на базе ПЛИС Altera, где все необходимые сведения были указаны в рамках одной журнальной публикации.

Следует сразу подчеркнуть, что процессоры в ПЛИС (так называемые софт-процессоры) ни в коей мере не становятся универсальными решениями, одинаково пригодными для разных классов задач. В данной статье будет по возможности представлена конструктивная критика подхода, основанного на использовании софт-процессоров. Представляется, что подробный анализ их преимуществ и недостатков принесет читателю больше практической пользы, чем простой набор листингов на каком-либо языке описания аппаратуры.

Итак, первое же возражение против софт-процессоров (и ПЛИС вообще) выглядит следующим образом: «...долгая практическая работа и опыт производства коммерчески успешной продукции говорят нам о том, что массово выпускаемые микроконтроллеры вполне способны решить любую стоящую перед нами задачу». Данный аргумент действительно весом и существенен. В самом деле, сравнивая объемы продаж ПЛИС и микроконтроллеров, нельзя не увидеть, что, несмотря на бурный рост рынка программируемой логики и возрастающий интерес к этой технологии, существует множество областей, в которых долго и плодотворно используются массовые решения из мира ASIC (то есть «жесткой» логики). Можно даже предположить, что для разработчиков, давно и успешно применяющих микроконтроллерные системы управления, попытки перехода к ПЛИС по соображениям «технической моды» могут оказаться попросту вредными. Действительно, если цифровая система управления не является определяющей частью проекта, то замена одного технического решения другим, скорее всего, слабо скажется на потребительских качествах выпускаемого изделия, но заставит разработчиков приложить немало усилий при переходе на иную элементную базу (и, соответственно, методы разработки). И рыночные доли производителей ПЛИС и микроконтроллеров в настоящее время наглядно демонстрируют степень востребованности программируемой логики. В каких-то применениях ни производительность, ни специальные возможности управляющего устройства не являются критичными, а основным требованием к микроконтроллеру становится обеспечение процесса сбора данных, их несложной обработки и передачи с помощью стандартных интерфейсов. Эти задачи вполне по силам микроконтроллерам минимальной стоимости, чьи возможности хорошо известны разработчикам. Очевидно, при достаточно четком видении путей использования существующих микроконтроллеров обращаться к ПЛИС вряд ли целесообразно.

В то же время можно указать основные технические задачи, которые эффективно решаются посредством ПЛИС. Исходя из наличия в современных FPGA большого количества аппаратных умножителей, имеет смысл рассмотреть возможность построения на базе таких FPGA систем распределенной обработки, многоканальной цифровой фильтрации и прочих вычислительных устройств, требующих выполнения значительного объема однообразных вычислений. Полезным может оказаться и выполнение внутри FPGA промежуточных преобразований потока данных, приходящих из внешнего АЦП с высокой частотой. Многоканальная обработка, возможность выполнения на базе ПЛИС нескольких независимых (или слабо зависящих друг от друга) цифровых устройств, минимизация габаритов изделия — все это аргументы «за».

При реализации алгоритмов цифровой обработки основная нагрузка ложится на аппаратные умножители в ПЛИС. Однако кроме собственно вычислительных операций от кристалла может потребоваться поддержка более или менее сложных протоколов обмена с внешними устройствами, работа со вспомогательным оборудованием и прочие операции, которые без труда решаются управляющей программой любого микроконтроллера. Иногда возможность оперативного внесения несущественных изменений является если не решающим, то очень важным фактором при определении жизнеспособности изделия в целом. Именно в этой связи часто появляются решения вида «микроконтроллер + ПЛИС», причем микроконтроллер обычно хорошо известен специалистам и не вызывает трудностей при работе, а ПЛИС выполняет основной объем однообразных вычислений, и ее выбирают исходя именно из требований обеспечения нужной производительности.

При таком подходе резонно задать вопрос: если уж ПЛИС так или иначе используется в системе, нельзя ли исключить из нее микроконтроллер, переложив его функции на ПЛИС? А если говорить точнее, нельзя ли разместить в ПЛИС некое подобие микроконтроллера, чтобы появилась возможность привлечь к разработке программиста, а не возлагать дополнительную нагрузку на инженера, проектирующего на HDL? В этом случае при уменьшении числа микросхем сохраняется общий порядок отладки системы: для уточнения алгоритмов достаточно внести изменения в исходные тексты управляющей программы, написанной на одном из языков высокого уровня и загружаемой в систему по достаточно простому интерфейсу. Таким образом, после создания основного блока вычислительных устройств работа с ПЛИС может быть продолжена в том же стиле, что и с микроконтроллерами.

Другой не менее убедительный контраргумент — проектирование системы на базе ПЛИС обычно требует более высокой квалификации разработчиков. Да и процесс проектирования с нуля, когда кроме самих функций изделия и алгоритмов проектируется и отлаживается процессор, чреват большим количеством ошибок. Их источником может стать и софт-процессор (ввиду ошибок, допущенных при его проектировании), и системное программное обеспечение, некорректно отражающее особенности только что спроектированного устройства. В то же время ошибки в аппаратной части ASIC-микроконтроллеров крайне редки, а ошибки в компиляторах для них, если даже и встречаются чаще, то по крайней мере известны и своевременно устраняются производителями. В конечном итоге коллектив, принявший решение о разработке собственного процессора на базе ПЛИС, будет вынужден воспроизводить процесс создания и отладки системного и прикладного программного обеспечения. Конечно, всегда существует возможность использовать готовые процессорные ядра, такие как MicroBlaze и PicoBlaze у Xilinx или Nios у Altera. В данном случае и разработка самих ядер, и создание системного ПО уже выполнены.

И все же остановимся на вопросах создания уникальных процессорных устройств, когда разработчик имеет все возможности для выбора архитектуры, системы команд и прочих характеристик проектируемого устройства. Реально ли это, и самое главное, применимо ли за пределами «научного интереса»? Кстати, хочется напомнить, как в России (причем со времен СССР) начиналось массовое использование ЭВМ. Речь идет именно о радиолюбительской практике и том переломе в сознании инженеров, когда стало понятно, что компьютер вполне реальный инструмент, способный не только помогать в процессе проектирования, но и заменять собой печатные платы, состоящие из триггеров, счетчиков, дешифраторов и прочих цифровых узлов. Нельзя не согласиться, что применение микропроцессорной техники резко сократило и финансовые, и временные затраты на разработку сложных систем. Если прежде для внесения изменений было необходимо переработать принципиальную электрическую схему, создать печатную плату, выполнить монтаж и наладку, и только потом приступать к тестированию нового алгоритма, то использование процессорной системы позволяло добиться аналогичных результатов простой перекомпиляцией управляющей программы. Причем к данному периоду были вполне применимы аргументы типа: «компьютер слишком сложное устройство», «разработка программы доступна только профессионалам», «отладить программу не проще, чем увидеть ошибку в электрической схеме» и прочее. Однако на сегодняшний день самые разные микроконтроллеры прочно вошли в инженерную практику, а их освоение не представляется чем-то чрезмерно сложным.

В связи с этим хотелось бы задать еще несколько отвлеченных вопросов. Сопоставимы ли характеристики ЭВМ РК-86 с зарубежными аналогами, выполненными на базе процессора i8080? Превосходили ли российские Spectrum-совместимые компьютеры оригинальный ZX-Spectrum? Думается, во времена появления всех перечисленных устройств такие вопросы возникали далеко не в первую очередь. И гораздо более важным является вклад, внесенный в общую копилку инженерных кадров благодаря тому, что многие разработчики когда-то изготовили и отладили собственный Specrtum-совместимый компьютер и в большей или меньшей степени изучили и программирование, и схемотехнику систем на базе микропроцессоров. В конце концов, речь не о том, насколько эффективными были выполняемые повсеместно разработки, намного ценнее инициирование процесса изучения микропроцессорных систем «снизу». Точно так же чуть позже, с переходом IBM PC в разряд обычной домашней ЭВМ, появились программисты, получавшие начальные знания самостоятельно. Следовательно, можно было рассчитывать на то, что поступающий на работу сотрудник не станет тратить время на освоение азов. И главное, большое количество инициативных исследований, студенческих проектов и прочих разработок, выполняемых массово, несомненно способствует формированию той «питательной среды», благодаря которой рождаются по-настоящему эффективные технические решения. Поэтому без появления аналогичной «питательной среды» (только уже в отношении софт-процессоров) обсуждать эффективность учебных проектов вряд ли имеет смысл. Цель публикаций, подобных этой, заключается именно в формировании представлений о софт-процессорах вообще, и только потом в предложении практических решений.

Кроме всего прочего, представления об архитектурах процессоров и практические навыки их проектирования помогают формировать круг разработчиков, способных заниматься не только созданием процессоров на базе ПЛИС, но и проектированием процессоров как таковых. Пробуя различные архитектуры и технические решения, используя столь удобный инструмент, как программируемая аппаратная платформа, специалисты получат необходимые базовые навыки, которые впоследствии окажутся полезными при выполнении проектов процессорных и контроллерных устройств с аппаратными соединениями. В рамках организации это означает, что наличие у разработчиков общих представлений о возможностях программируемой элементной базы позволит им выполнять реальные проекты, хорошо понимая все выгоды софт-процессоров или используя опыт одного

из предыдущих проектов, выполненных при освоении данной технологии.

Итак, резюмируя, можно отметить, что процессоры в ПЛИС не только интересное хобби и возможность приобщения к миру разработчиков микропроцессорных систем. Владение навыками проектирования процессоров формирует хороший задел на будущее, когда ранее выполненные учебные проекты неожиданно находят практическое применение. К тому же в российских условиях проектирование процессоров в ПЛИС представляется очень перспективным в плане развития отечественной элементной базы. Действительно, при затратах, на порядки меньших по сравнению с затратами на создание технологических линий для выпуска микросхем, российские студенты и инженеры получают возможность изучить методики проектирования процессоров и опробовать на практике свои идеи. Будут ли эти проекты реализованы в промышленных масштабах? Вероятно, подавляющее большинство не будет осуществлено, но вернемся к программированию для IBM PC. Именно массовое распространение этих компьютеров и методик программирования для них в очень большой степени способствовало тому взрывному росту информационных технологий, который наблюдается сегодня. Аналогично, формирование «питательной среды» из простых и понятных методик проектирования несложных процессорных устройств поможет широкому кругу разработчиков глубже понять проблемы, возникающие у конструкторов массово выпускаемых процессоров, а возможно, и предложить полезные архитектурные и схемотехнические решения.

ПЛИС как аппаратная основа для софт-процессоров

Итак, почему же именно ПЛИС предлагаются в качестве аппаратной платформы для создания конфигурируемых процессоров? В общих чертах можно ответить так: потому что именно ПЛИС позволяют получить макетный вариант цифрового устройства в кратчайшие сроки и с возможностью неограниченного внесения изменений без затрат материалов или комплектующих. В конечном итоге процесс проектирования процессора на базе ПЛИС очень напоминает разработку программы. Если говорить о полном перечне программного и аппаратного обеспечения, необходимого для получения макетного варианта, то сегодня он не предусматривает существенных затрат и состоит из следующих пунктов:

- ПЛИС или макетная плата на базе ПЛИС;
- САПР ПЛИС, поддерживающая выбранную микросхему;
- программатор.

Сведения о доступности перечисленных устройств и программного обеспечения для удобства и наглядности приведены в таблице 1.

Таблица 1. Аппаратные и программные инструменты для реализации софт-процессора

Фирма-производитель	Xilinx	Altera
Стартовый набор на базе семейства ПЛИС начального уровня	Spartan-3 Starter Kit	Cyclone-II Starter Kit
САПР ПЛИС	ISE Webpack	Quartus Web Edition
Программатор	Parallel Cable III	ByteBlaster

В этой таблице приведены сведения, относящиеся к двум ведущим производителям ПЛИС. Тот факт, что основная часть рынка почти поровну (примерно 50% у Xilinx и 35% у Altera) поделена между двумя компаниями, выпускающими приблизительно аналогичную продукцию, благоприятно сказывается на их ценовой политике и отношению к предоставлению САПР и подробной технической документации. Из приведенного в таблице 1 списка необходимого оборудования и ПО фактически только первая строка влечет сколько-нибудь существенные финансовые затраты. Обе фирмы бесплатно предоставляют версию САПР с ограничениями по логическому объему ПЛИС (не поддерживаются ПЛИС, емкость которых исчисляется миллионами логических вентилей, а стоимость — сотнями долларов). Достаточно просто собираются и программаторы. В случае Xilinx потребуются две микросхемы 74125 (555ЛП8), а программатор фирмы Altera собирается на одной микросхеме 74244 (555АП5). Собственно, ПЛИС можно приобрести и отдельно, причем ее стоимость составит $10–15 (Spartan-3 для фирмы Xilinx и Cyclone-II для Altera). Следует добавить, что в состав стартовых наборов (starter kit) входит все перечисленное в таблице. Для начала работы, таким образом, оказывается достаточным найти PC со свободным разъемом LPT и операционной системой Windows XP. Даже для радиолюбителей такие наборы вполне доступны — их цена составляет около $130–150. Разумеется, проектирование процессоров далеко не единственная задача, которая может быть решена с их помощью.

Приведенные в таблице сведения относятся к продукции двух самых известных производителей ПЛИС, однако не следует думать, что только эти устройства могут быть использованы для разработки софт-процессоров. При ориентации на современные языки описания аппаратуры вполне возможно создавать проекты, переносимые как между семействами ПЛИС одной фирмы, так и между ПЛИС различных фирм. Поэтому выбор элементной базы — вопрос личных предпочтений каждого разработчика. Различия в архитектуре и возможностях ПЛИС Xilinx и Altera несущественны на уровне языков описания аппаратуры, а на практике оказывается возможным подобрать архитектурное решение, хорошо соответствующее используемой микросхеме. Примеры, приведенные в данной статье, выполнены на базе Spartan-3 Starter Kit фирмы Xilinx.

Процессор с точки зрения теории

Развитие процессоров и переход компьютера в разряд домашней техники привели к тому, что с термином «процессор» часто ассоциируются понятия из области современных высокопроизводительных процессоров с x86-архитектурой. В массовом представлении частота процессора должна измеряться гигагерцами, объем адресуемой памяти — гигабайтами, а сама архитектура должна быть как минимум суперскалярной, поддерживающей out-of-order-выполнение команд, многоуровневый кэш, предсказание переходов и прочие атрибуты флагманских изделий мировых лидеров в области разработки процессоров. Разумеется, возможности ПЛИС в этом плане достаточно скромны, хотя необходимо иметь в виду, что все перечисленное выше совершенно не обязательно должно присутствовать в устройстве, которое формально может называться процессором. Более того, программируемая платформа сама по себе предполагает использование несколько иных методик проектирования по сравнению с ASIC, а потому большинство понятий из мира x86 на первоначальном этапе окажутся лишними.

В популярной технической литературе очень часто можно встретить рисунки, изображающие структуру процессора. Внутри процессора помещаются регистры, арифметико-логическое устройство (АЛУ), контроллеры памяти и периферийных устройств, некое абстрактное «устройство управления»… и на этом перечень, в общем-то, заканчивается. Такое изложение является вполне достаточным для ознакомления пользователей с внутренним устройством компьютера, но совершенно не дает представления о том, какие процессы происходят внутри, как именно взаимодействуют между собой узлы процессора, и наконец, как необходимо приступать к его проектированию?

Для пояснения методологии проектирования процессоров необходимо обратиться с абстрактной модели — машине Тьюринга. Хотя эта машина в чистом виде не может быть реализована, она является теоретической основой для выполнения всех известных в настоящее время алгоритмов. В основе машины Тьюринга (рис. 1) лежит бесконечная лента, на которой записаны символы, и указатель, способный перемещаться по ленте. На каждом шаге работы можно просмотреть символ, находящийся под указателем, заменить его другим символом, а также переместить указатель на новое место. Перечень предполагаемых действий на первый взгляд выглядит довольно ограниченным, но все многообразие операций с данными скрывается за терминами «заменить символ» и «переместить указатель». Правила замены и перемещения, по большому счету, не регламентированы, поэтому любые операции, реализуемые цифровой логикой, не проти-

воречат такой модели. В действительности, все существующие процессоры так или иначе приближаются к машине Тьюринга, поскольку адресуемая ими память может рассматриваться как фрагмент бесконечной ленты с символами, а с помощью системы команд выражаются те правила, по которым один символ заменяется другим. Впрочем, здесь есть и свои тонкости. Например, с указателем в машине Тьюринга может быть сопоставлен тот регистр, посредством которого адресуются данные во внешней памяти (по крайней мере, так должно следовать из представленной модели). Процессоры могут иметь несколько таких указателей на данные, зато уникальным регистром-указателем является счетчик команд, не отраженный в абстрактной модели. Его поведение как раз очень похоже на поведение указателя в машине Тьюринга, поскольку в ходе выполнения команд он способен переходить к произвольному адресу программы, но указывает не на символы ленты (данные в памяти), а на выполняемую команду. Чем вызвана подобная путаница, становится понятным, если учесть, что перемещать указатель в абстрактной машине может и человек (по тем правилам, которые он «хранит» в голове), а вот все действия процессора должны быть где-то зафиксированы. В конечном итоге оказывается, что можно использовать две «ленты с символами»: одна хранит обрабатываемые данные, а другая — «управляющие символы», то есть команды процессора. В микропроцессорной технике такое различие зафиксировано явно: фон-неймановская архитектура предполагает хранение команд и данных в общем адресном пространстве (другими словами, «на одной ленте»), а гарвардская архитектура разделяет устройства хранения команд и данных. В последующих частях статьи будет показано, как с помощью явного разделения адресных пространств можно повысить производительность процессора.

Реализовать процессор проще всего с помощью так называемого конечного автомата (КА). В англоязычной литературе используется термин finite state machine (FSM). Под конечным автоматом подразумевается устройство, состояние которого может быть однозначно описано неким набором переменных. На каждом элементарном шаге своей работы КА переходит в новое состояние, зависящее от внешних сигналов (всегда) и от его предыдущего состояния (в случае так называемого автомата Мили). Именно эта последняя разновидность автомата представляет особый интерес в контексте рассматриваемого вопроса. Представим, что процессор выполняет операцию инкремента одного из регистров. В какое состояние он придет после выполнения? Очевидный ответ: регистр увеличится на единицу относительно своего предыдущего состояния. Следовательно, для успешного выполнения операции инкремента требуется знать предыдущее состояние инкрементируемого регистра (в наиболее общем случае — всех регистров). Таким образом, для выполнения последовательности команд процессор должен организовать регистры наподобие переменных в языках программирования. Тогда в процессе исполнения каждой команды какие-то регистры изменят свое состояние.

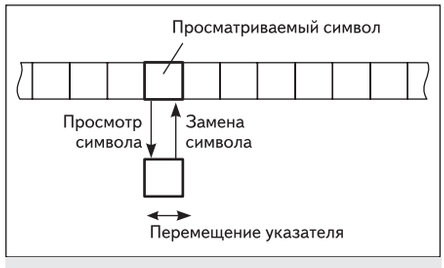

Рис. 1. Машина Тьюринга

От конечного автомата к процессору

Рассмотрим на каком-либо простейшем примере последовательность проектирования процессорного устройства. В качестве такого примера можно выбрать устройство управления светофором. Это не перегрузит изложение излишними деталями, однако позволит наглядно проиллюстрировать все этапы перехода от конечного автомата к процессору.

В общих чертах порядок работы светофора выглядит следующим образом.

1. Включение красного сигнала.
2. Включение желтого сигнала.
3. Включение зеленого сигнала, выключение красного и желтого.
4. Мигание зеленого сигнала.
5. Включение желтого сигнала, выключение зеленого.

Приведенный порядок переключения сигналов светофора широко известен, и на первый взгляд, полностью корректен. Можно выделить переменные состояния, управления которыми окажется достаточно для реализации произвольного порядка переключения сигналов. К таким переменным очевидно относятся однобитные сигналы, управляющие зажиганием соответствующих сигналов светофора. Обозначим их как *red*, *yellow*, *green*. Можно достаточно просто переписать алгоритм переключения сигналов с использованием операторов присваивания этим переменным значений 0 и 1, однако с точки зрения цифровой схемотехники, данный подход содержит существенный недостаток. Действительно, рассматривая алгоритм, содержащий строки вида *red=1 yellow=1* не трудно заметить, что такая запись не содержит информации о временных интервалах между отдельными переключениями сигналов. Более того, устройство, описанное подобным образом, нельзя реализовать с использованием только асинхронной логики.

Таблица 2. Порядок изменения переменных состояния конечного автомата

st	red	yellow	green
0–10	1	0	0
11–12	1	1	0
13–19	0	0	1
20	0	0	0
21	0	0	1
22	0	0	0
23–24	0	1	0

Для обеспечения требуемых временных интервалов необходимо разбить последовательность работы светофора на отдельные шаги и ввести дополнительную переменную состояния, определяющую номер шага. Обозначим ее как *st* (от слова «state» — состояние). Тогда, с учетом соотношения длительностей отдельных режимов работы светофора, определим порядок работы, как указано в таблице 2.

Обратите внимание, что в таблице отсутствует такое состояние, как «зеленый мигающий». Вместо этого мигание зеленого сигнала определено как последовательная смена состояний «включено» и «выключено». Альтернативой является введение переменной состояния для зеленого сигнала, способной отображать состояния «зеленый не горит», «зеленый горит», «зеленый мигает», что несколько усложняет аппаратную реализацию контроллера.

Полученная последовательность смены состояний переменных легко реализуется в виде конечного автомата. Переменной-селектором является номер состояния *st*, на основе которого реализуется оператор выбора. Пример реализации конечного автомата на VHDL приведен в следующем листинге:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity svetofor is
        Port ( clk : in std_logic;
                        red : out std_logic;
                        yellow : out std_logic;
                        green : out std_logic);
end svetofor;

architecture Behavioral of svetofor is

signal st : integer range 0 to 31;

begin

process(clk)
begin
   if clk'event and clk='1' then
      case st is
         when 0 =>   red <= '1';
                     yellow <= '0';
                     green <= '0';
                     st <= st + 1;
         when 11 =>  yellow <= '1'; st <= st + 1;
         when 13 =>  red <= '0';
                     yellow <= '0';
                     green <= '1';
         st <= st + 1;
         when 20 =>  green <= '0'; st <= st + 1;
         when 21 =>  green <= '1'; st <= st + 1;
         when 22 =>  green <= '0'; st <= st + 1;
         when 23 =>  yellow <= '1'; st <= st + 1;
         when 24 => st <= 1;
         when others => st <= st + 1;
      end case;
   end if;
end process;

end Behavioral;
```

Шаблон конечного автомата, описанный выше, достаточно универсален и пригоден для построения самых разнообразных устройств на его основе. Порядок проектирования конечного автомата на VHDL выглядит следующим образом:

1. Выбор переменных состояния КА и распределение их по переменным вида *std_logic_vector* или *integer*.
2. Перечисление возможных состояний КА и присвоение им номеров.
3. Описание для каждого из состояний КА вариантов модификации переменных состояния и перехода к новому состоянию.

При подобном подходе HDL-описание может быть сведено к единственному оператору case, выполняющему изменение состояния по фронту тактового сигнала. Ввиду того, что КА является стандартным цифровым узлом, существующие САПР ПЛИС обычно обладают возможностью оптимального способа реализации таких устройств.

Теперь рассмотрим порядок модификации алгоритма работы КА. В приведенном примере актуальным может являться, например, изменение длительности свечения одного из сигналов. Следуя описанному порядку проектирования, необходимо скорректировать таблицу переходов между состояниями, увеличив количество состояний, в которых должен оставаться активным интересующий нас сигнал (например, красный). В этом случае вся таблица должна быть «сдвинута» на нужное число состояний, и HDL-описание конечного автомата отредактировано таким образом, чтобы все последующие переключения сигналов осуществлялись на N тактов позже (где N — требуемое увеличение длительности свечения красного сигнала). Порядок переключения остальных сигналов при этом не изменится.

Несмотря на простоту данной коррекции, она, тем не менее, потребует изменения исходного текста HDL-описания и новой трансляции проекта в ПЛИС. Изменения аппаратной части КА могут в данном случае коснуться, прежде всего, изменения разрядности счетчика состояний (например, если после внесения изменений в листинг будет недостаточно 5 разрядов переменной *st*).

Рассмотрим семантику операций, выполняемых по отдельным переходам внутри проектируемого КА. В приведенном примере встречаются следующие операции (табл. 3).

Таблица 3. Перечень операций, выполняемых конечным автоматом

№	VHDL-код	Описание
1	red <= '0';	Выключить красный сигнал
2	red <= '1';	Включить красный сигнал
3	yellow <= '0';	Выключить желтый сигнал
4	yellow <= '1';	Включить желтый сигнал
5	green <= '0';	Выключить зеленый сигнал
6	green <= '1';	Включить зеленый сигнал
7	st <= 0;	Переход к начальному состоянию

Операция увеличения номера состояния в таблице не приведена, поскольку оно происходит одновременно со всеми остальными операциями, за исключением операции № 7, при которой номер состояния принудительно изменяется на начальный. Можно сформулировать порядок работы КА следующим образом. По каждому фронту тактового сигнала происходит:

1. Номер состояния или увеличивается на единицу, или загружается некоторым фиксированным значением.
2. Происходит изменение произвольного количества переменных состояния.

Модифицируем разработанный КА таким образом, чтобы переменные состояния изменялись не в соответствии с номером состояния *st*, а по некоторому внешнему сигналу *cmd*, кодирующему изменение переменных состояния в соответствии с таблицей. Очевидно, что содержимое таблицы должно поступать для обработки последовательно, с каждым новым тактом. Для выбора номера сигнала необходимо завести еще один регистр — счетчик команд. В микропроцессорной технике встречаются обозначения PC (Program Counter), а также IP (Instruction Pointer). В данном случае речь идет о регистре, описываемом на HDL, чье имя может быть практически любым, поэтому вопросы выбора обозначения оставлены на усмотрение читателей. Примем для счетчика адреса обозначение *ip*, привычное по опыту использования процессоров с архитектурой x86.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity svetofor_p is
        Port (  clk : in std_logic;
                cmd : in std_logic_vector(3 downto 0);
                ip : inout std_logic_vector(7 downto 0);
                red : out std_logic;
                yellow : out std_logic;
                green : out std_logic);
end svetofor_p;

architecture Behavioral of svetofor_p is

begin

process(clk)
begin
   if clk'event and clk = '1' then
      case conv_integer(cmd) is
         when 0 => ip <= ip + 1;
         when 1 => red <= '0'; ip <= ip + 1;
         when 2 => red <= '1'; ip <= ip + 1;
         when 3 => yellow <= '0'; ip <= ip + 1;
         when 4 => yellow <= '1'; ip <= ip + 1;
         when 5 => green <= '0'; ip <= ip + 1;
         when 6 => green <= '1'; ip <= ip + 1;
         when 7 => ip <= conv_std_logic_vector(0, 8);
         when others => ip <= ip + 1;
         end case;
      end if;
end process;

end Behavioral;
```

Сравнение листингов позволяет выделить различия между ними. Прежде всего, второй листинг не содержит описания порядка переходов между состояниями, а только описывает семантику этих переходов — изменение переменных состояния. Однако для работы устройства, описанного в новом листинге, требуется внешний модуль памяти, обеспечивающий поток команд в соответствии с анализом сигнала *ip*. Создадим такой модуль, учитывая, что он должен представлять собой асинхронную память.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity svetofor_code is
        Port ( ip : in std_logic_vector(7 downto 0);
               cmd : out std_logic_vector(3 downto 0));
end svetofor_code;

architecture Behavioral of svetofor_code is

begin

with conv_integer(ip) select
   cmd <= conv_std_logic_vector(2,4) when 0,
          conv_std_logic_vector(4,4) when 11,
          conv_std_logic_vector(6,4) when 13,
          conv_std_logic_vector(1,4) when 14,
          conv_std_logic_vector(3,4) when 15,
          conv_std_logic_vector(5,4) when 20,
          conv_std_logic_vector(6,4) when 21,
          conv_std_logic_vector(5,4) when 22,
          conv_std_logic_vector(4,4) when 23,
          conv_std_logic_vector(3,4) when 24,
          conv_std_logic_vector(7,4) when 25,
          conv_std_logic_vector(0,4) when others;

end Behavioral;
```

Остается соединить полученные модули, как это показано на рис. 2.

На рисунке видна произошедшая трансформация конечного автомата. Он больше не выглядит как «черный ящик», выходами которого являются собственно значения переменных состояния. Разорвав внутреннюю цепочку между выходом сигнала *ip* и входом его перезаписи, мы приходим к тому, что в схему добавляется выход номера команды (или проще — адрес команды) и вход команды (или же шина данных, подключаемая к внешней памяти команд). Собственно говоря, произошедшие изменения достаточно плавные и естественные. Однако теперь можно не изменять логику переходов внутри HDL-описания автомата, а все изменения в порядке переключения сигналов производить за счет коррекции содержимого внешней памяти.

На схеме также видно, что образованное устройство содержит обратную связь: новое значение команды на линии *cmd* появляется с некоторой задержкой относительно установления нового значения *ip*. В то же время сигнал *ip* устанавливается через некоторое время после фронта тактового сигнала. Временные диаграммы его работы представлены на рис. 3.

Фронт тактового сигнала вызывает обновление переменных состояния, к числу которых относится и счетчик адреса *ip*. Через некоторое время T_1, соответствующее задержке распространения сигнала до выхода регистра *ip*, этот сигнал устанавливает новое значение адреса на входе блока памяти.

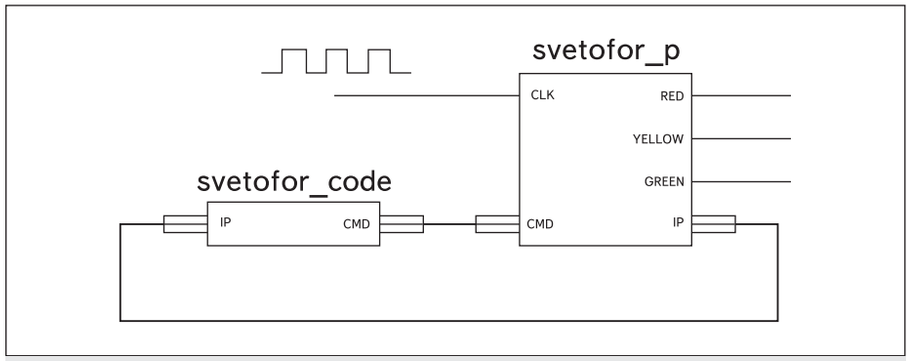

Рис. 2. Процессор: конечный автомат с внешней памятью программ

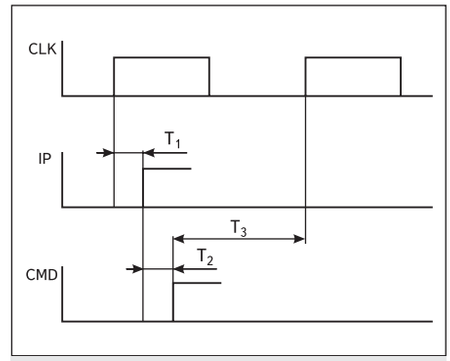

Рис. 3. Временные диаграммы работы процессора

Данный блок, работая асинхронно, обновляет сигнал *cmd* на своем выходе через время T_2. Следующий фронт тактового сигнала может быть подан на вход управляющего автомата при том условии, что с момента установления нового значения *cmd* прошло время не менее t_{setup} (время установления сигнала, приводимое в технической документации). Таким образом, время T_3 от момента появления нового значения *cmd* до фронта тактового сигнала, должно быть не меньше, чем суммарное время установления сигнала и время распространения *cmd* от выхода блока памяти до входа управляющего блока. Сумма времен T_1, T_2 и T_3 определяет минимальный период тактового сигнала.

Следует отметить, что блоки, использованные для построения программируемого конечного автомата, являются стандартными для современных САПР, основанных на HDL. Приведенные в листингах модули реализуются с помощью стандартных шаблонов проектирования, использующих оптимальное относительное размещение логических ячеек, что позволяет получать устройства с высокими рабочими частотами даже без применения сложных приемов оптимизации размещения ячеек.

Приведенная схема конечного автомата с внешней памятью, задающей последовательность переходов, фактически является процессором. Хотя функциональные возможности и список регистров данного устройства не вполне соответствуют классическим схемам широко распространенных процессоров, качественной разницы нет. Приведенный шаблон конечного автомата легко может быть дополнен таким образом, чтобы полностью соответствовать набору регистров, арифметико-логических устройств и интерфейсов, характерных для микропроцессоров различных типов.

С практической точки зрения можно сделать два совершенно противоречивых замечания:

1. Схема в принципе работоспособна.
2. Не стоит использовать ее в практической конструкции.

Сущность замечаний становится понятна, если учесть особенности реализации цифровых устройств в ПЛИС, где задержки распространения сигналов по программируемым трассировочным линиям достигают достаточно больших значений и вполне сопоставимы с задержками сигналов в логических ячейках. Прежде всего, если попытаться реализовать память именно в виде блока внутри ПЛИС, очень легко столкнуться с так называемыми гонками фронтов. Особенно легко получить этот эффект, если подать тактовую частоту с делителя — например, чтобы пронаблюдать работу процессора, тактируемого частотой 1 Гц. При этом внутри ПЛИС образуется так называемый gated clock — вместо глобальной тактирующей сети тактовый сигнал будет подан на триггеры через обычные ячейки и комбинаторную логику. Тогда задержки распространения могут оказаться непредсказуемыми (разумеется, их можно определить при трассировке, однако в процессе разработки описания нельзя заранее сказать, как она будет выполнена). Наблюдались ситуации, когда отдельные биты *cmd* изменялись раньше, чем все переменные состояния, что нарушало нормальное функционирование управляющего блока. Ввиду всего перечисленного приведенный пример можно рассматривать только в качестве промежуточного этапа трансформации простого конечного автомата в реальное устройство, пригодное к практическому использованию.

Описанная ситуация, вообще говоря, весьма характерна для FPGA. В ряде руководств (например, [4]) настоятельно рекомендуется использовать асинхронное формирование сигналов наряду с синхронным изменением номера состояния. В приведенном примере следует исключить из веток оператора case присвоение новых значений сигналам *red*, *green*, *yellow*, а вместо этого формировать такие сигналы асинхронно, анализируя текущее состояние *ip*.

Далее, в примере был создан модуль на 256 ячеек по 4 бита, что вполне реализуемо в современных ПЛИС. Однако стоит иметь в виду, что все «массовые» ресурсы в ПЛИС обладают весьма высокой «удельной стоимостью». В ряде случаев можно воспользоваться внутренней блочной памятью (об этом будет рассказано в следующей части статьи), но если некая стандартная функция реализуется дешевым внешним устройством, то практически со всех точек зрения выгоднее использовать именно его. В данном примере подключить следует микросхему флэш-ПЗУ, где и будет храниться весь программный код. Какую же задержку даст такая микросхема? По всей видимости, 70 или 55 нс (ориентируясь на современные устройства таких производителей, как Amd, Atmel, Intel и др.). Добавляя сюда задержку внутри ПЛИС, округленно получаем период около 100 нс, что соответствует тактовой частоте 10 МГц. Показатель не слишком высокий, поскольку очевидно, что большая часть задержки вносится внешней микросхемой.

Есть ли простой выход? Очевидное решение — использовать накристальную блочную память с синхронным интерфейсом, которая в современных ПЛИС имеется в достаточно большом количестве. Синхронный интерфейс сам по себе быстрее и надежнее, но позволяет получать новую команду после установки нового значения *ip* — потребуется еще один фронт тактового сигнала, чтобы новое значение появилось на выходе блочной памяти. В следующей части статьи будет приведена методика проектирования процессора, использующего для хранения программы блочную память ПЛИС с синхронным интерфейсом.

Продолжение следует

Литература

1. Тарасов И. Системы на кристалле на базе FPGA Xilinx с встроенными процессорами PowerPC // Компоненты и технологии. 2005. № 7–9.
2. Зотов В. PicoBlaze — семейство восьмиразрядных микропроцессорных ядер, реализуемых на основе ПЛИС фирмы Xilinx // Компоненты и технологии. 2003. № 4.
3. Каршенбойм И. Микропроцессор своими руками // Компоненты и технологии. 2002. № 6–7.
4. Грушвицкий Р. И., Мурсаев А. Х., Угрюмов Е. П. Проектирование систем на микросхемах программируемой логики. СПб.: БХВ-Петербург, 2002. 608 с.: ил.