Микропроцессор своими руками — 3
Ассемблер и софт-симулятор

Несколько лет назад автором был опубликован цикл статей [1—3], в которых было дано описание разработки проекта встроенного в FPGA микроконтроллера. В первой «серии» было рассказано о том, как сделать микропроцессор с набором команд, заданным пользователем. Во второй «серии» было дано описание битового процессора. Оба проекта были выполнены на AHDL, как на наиболее популярном тогда языке. Процессоры и их система команд были описаны очень схематично. С тех пор автором было получено довольно много писем с просьбами продолжить данный «сериал».

Иосиф КАРШЕНБОЙМ
www.iosifk.narod.ru

В этой статье будет показан пример выполнения набора команд для реального микроконтроллера и описано то, как сделать «самодельный» ассемблер, формирователь кода «прошивки» и софт-симулятор для встроенного в FPGA микропроцессора. Термин «софт-симулятор» применен здесь для того, что под термином «симулятор», программисты понимают симулятор микроконтроллера, показывающий то, как он выполняет свою программу, а дизайнеры FPGA под этим термином обычно понимают свои симуляторы VHDL или Verilog, показывающие то, как работает «железо». В этой статье термин «софт-симулятор» применяется в «программистском» смысле, то есть для симуляции выполнения программ в микроконтроллере. В качестве примера приводится часть описания от реально работавшего проекта.

Зачем нужны встроенные микропроцессоры

Итак, основой для разработок управляющих устройств в цифровой технике, и именно в FPGA, служат статические автоматы. Для небольших проектов в FPGA было вполне достаточно автомата на несколько десятков состояний. Однако за последнее десятилетие произошел качественный рост микросхем FPGA. Цены на них упали, а это значит, что появилась возможность применять микросхемы с гораздо большей «начинкой». FPGA стали значительно быстрее, увеличился объем «классических» ресурсов, таких как число логических ячеек, включающих в себя логику и триггеры, а также увеличился объем памяти на кристалле. Появились новые узлы: умножители, блоки управления частотой — DLL и PLL, сдвиговые регистры и т. д. Новые технологии для маршрутизации сигналов по кристаллу позволили поднять тактовые частоты проектов до 300 МГц и более. Кроме того, программные инструменты — применение технологии привязки части проекта к фрагменту кристалла по технологии LogicLock (Altera) и RLock (Xilinx) — позволяют сократить межсоединения для различных частей проекта, а значит, повысить быстродействие.

Проекты стали более быстродействующими и сложными. А это, в свою очередь приводит к усложнению узлов управления. При каждом (!) изменении статического автомата производится новая перекомпиляция всего узла. А, следовательно, именно здесь и может быть источник ошибок. И чтобы провести полное тестирование, необходимо каждый раз проводить полную проверку заново скомпилированной части проекта в том случае, если проводится частичная компиляция проекта, или всего проекта полностью. Причем с ростом сложности проекта увеличивается и время на симуляцию и компиляцию. Если десять лет назад время в 15 минут, требуемое на симуляцию проекта в MAX Plus II на PC-Intel 486-100 казалось очень большим, то сейчас на симуляцию сложного проекта может потребоваться время от получаса до нескольких часов. То же можно сказать и о компиляции проекта. Как только в проекте появляются критические параметры по объему ресурсов или по быстродействию, время, затрачиваемое на компиляцию проекта, значительно возрастает.

Так в чем же выигрыш в случае применения встроенного микропроцессора? Выигрыш в том, что значительно упрощается отладка проекта. «Схема», то есть софт-ядро микроконтроллера, отлаживается только один раз. Далее она фиксируется на кристалле, если это необходимо, для получения большой скорости работы. Для фиксации взаимного расположения узлов схемы на кристалле используется непосредственная привязка схемы к ресурсам кристалла. Так же можно произвести только взаимную привязку отдельных частей схемы, позволив компилятору перемещать весь узел микроконтроллера по кристаллу. И, поскольку микроконтроллер как функциональный узел проверяется отдельно, то идеальным вариантом работы по отладке проекта будет тот вариант, когда память программ для микропроцессора загружается в работающем изделии, так же как и в режиме ISP для традиционных микроконтроллеров. При этом нет необходимости каждый раз запускать компиляцию проекта. Но, даже в случае, когда такой способ работы не реализован, процесс отладки программ при помощи софт-симулятора происходит гораздо быстрее, чем при отладке статических автоматов. В качестве примера можно привести следующий факт. В данной статье автор описывает узел управления для платы, содержавшей регулируемые источники напряжения, измерители напряжения, тока и т. д. Автор здесь выполнял только узел управления, который находился в FPGA для данной платы. И этот узел был выполнен как встроенный микроконтроллер. Однако задание на разработку самой платы было дополнено и потребовалась модернизация функциональных узлов.

В процессе работ по модернизации функциональных узлов на плате проект был передан для отладки этих узлов другому инженеру, не владевшему технологией встроенных микроконтроллеров. В результате его работы в новую версию платы в устройстве управления был встроен статический автомат с 206 состояниями!

О приведенном выше примере можно конечно сказать и так: «Но ведь заработало же!». Да, можно сказать и так. Вот только насколько

просто или сложно будет модернизировать такой проект, сопровождать его во время «жизни» изделия — это каждый разработчик пусть сам решает для себя.

Общая тенденция на сегодняшний день такова — при числе состояний автомата от 50 и выше целесообразно применять встроенный микроконтроллер. Это же подтверждается публикациями [1, 9, 10].

Именно поэтому целью данной статьи является рассмотрение методики проектирования «маленьких» «самодельных» микроконтроллеров, так как они являются хорошей альтернативой для замены статических автоматов с большим числом состояний.

Микроконтроллер: стандартный или «самодельный»?

При сравнении «самодельного» микроконтроллера и «стандартного» речь пойдет о IP-ядрах микроконтроллеров. «Самодельный» — это значит, что набор команд определяется пользователем. «Стандартный» — набор команд соответствует набору команд какого-либо стандартного микроконтроллера, например AVR или MCS51.

К приведенному ниже примеру необходимо добавить только одно замечание. Не имеет смысла сравнивать любой «самодельный» микроконтроллер с любым «стандартным». Речь идет о том, что «самодельный» микроконтроллер не имеет смысла использовать при больших объемах программ, ибо при этом необходимо иметь хорошую инструментальную поддержку. Делать же специализированные компиляторы языков высокого уровня, симуляторы и отладчики для каждого проекта — довольно трудоемкое занятие, и оно может быть оправдано только при очень большой серийности выпускаемых изделий. По мнению автора, встроенные в FPGA микроконтроллеры предназначены, в основном, для локальных задач предварительной обработки данных. При необходимости проводить более сложную обработку в FPGA может быть выделен отдельный ресурс, в котором эта обработка и будет производиться. Но, конечно, все зависит от конкретных областей применения. Для этих целей может быть использован и «большой штатный» микроконтроллер, предлагаемый производителем FPGA и оптимизированный по занимаемым ресурсам и быстродействию для конкретной серии микросхем. Но, еще раз хочется уточнить, что целью данной статьи являются именно «маленькие» «самодельные» микроконтроллеры и методики их проектирования.

Набор команд: стандартный или «самодельный»?

Итак, как сделать микропроцессор, уже написано. В последнее время появилось достаточно много публикаций и открытых проектов на эту тему. Но при разработке любого микропроцессорного средства успех дела определяет не только сам проект в «железе», но и возможность быстро произвести отладку программ для этого «железа». При разработке встроенных в FPGA микроконтроллеров существует трудная задача по выбору оптимальной системы команд. Если выбрать стандартную систему команд, уже применяемую в каком-либо микроконтроллере, то не будет проблем с инструментальным программным обеспечением. Правда, сам проект микроконтроллера будет далеко не оптимальным, потребует много ресурсов в FPGA. Если же пойти по пути выбора оптимальной системы команд, то получим хороший проект в FPGA, но при этом надо будет сделать «самодельные» инструментальные программные средства. Какой путь выбрать? Ответ на этот вопрос должен даваться только при рассмотрении конкретных условий для конкретного проекта.

Что касается автора, то он для своих разработок выбирал второй путь. Почему? Да, он сложней, чем первый. Но он позволяет делать проект более гибким. Если у вас есть возможность добавить регистр, флаг, тестовый выходной сигнал и так далее, то процесс отладки проекта пойдет гораздо быстрее. Мало того, вы не ограничиваетесь жесткими рамками «стандартного микроконтроллера». Число портов, объем ОЗУ — все это можно менять и настраивать под задачу проекта. И последний довод в пользу «самодельных» микропроцессоров: «самодельный» микропроцессор всегда можно сделать так, что он будет решать задачу пользователя быстрее, чем стандартный. Правда, еще раз надо подчеркнуть, что здесь сравниваются только встроенные в FPFA микроконтроллеры, и сравниваются только при выполнении конкретной локальной задачи.

Теперь, непосредственно, сам пример. Представим, что мы хотим получить очень компактный по объему занимаемого в FPGA ресурса микроконтроллер. И пусть он обслуживает некоторую «самодельную» периферию. Например, узел связи с микросхемой АЦП, имеющей параллельную интерфейсную шину. Алгоритм обмена будет такой: если есть готовность, то прочитать данные из порта и переложить их во внутреннюю память микроконтроллера. «Стандартный» микроконтроллер выполнит этот алгоритм следующим образом (конечно, это только пример, и возможны другие варианты):

- протестирует бит порта;
- если есть готовность, то переход на обслуживание, иначе — выход;
- производится сохранение аккумулятора;
- вводятся данные на аккумулятор;
- устанавливается указатель на память;
- выполняется пересылка данных из аккумулятора в память;
- восстанавливаются данные в аккумуляторе;
- выполняется возврат из подпрограммы обслуживания.

«Самодельный» микроконтроллер выполнит этот же алгоритм так:

- если флаг, то выполнить следующую команду, иначе следующую команду пропустить;
- выполнить пересылку данных из порта в память и инкрементировать указатель.

Как видно из приведенного примера, выигрыш по производительности в данном фрагменте будет четырехкратный. А если «стандартный» микроконтроллер имеет конвейер команд, то выигрыш по времени будет еще больше.

Если же требуется производить более сложную обработку, например суммирование с накоплением, то выигрыш по производительности может быть еще выше. В «самодельном» микроконтроллере добавляется столько регистров — указателей на память, сколько нужно для эффективной работы с таблицами. А операция суммирования может быть совмещена с операцией пересылки в память. Отсюда следует, что при выполнении определенной локальной задачи «самодельный» микроконтроллер всегда будет побеждать «стандартный» по скорости работы и по занимаемым ресурсам на кристалле.

И если с разработкой аппаратной части проекта всё более или менее обстоит благополучно, то вот с описаний того, как сделать программные, сейчас очень мало.

Поэтому автор считает, что описание методики разработки программных инструментов сейчас более актуально.

О стиле работы и о программных инструментах

Еще необходимо добавить, что есть разные стили работы над проектом. Первый путь — «быстро-быстро руками». Второй путь — сначала сделать программные инструменты, а потом «нажать кнопку». При этом все доработки и изменения в проекте — это тоже только «нажать кнопку». Этот подход изложен в предыдущих статьях автора о программных инструментах [7, 8]. То, что описано в статье [8], являлось только частью интегрированного программного инструмента, использованного при разработке встроенных в FPGA-микроконтроллеров для узла управления модулем. В данной статье будет описана еще одна часть этого интегрированного программного инструмента.

Несколько слов о микроконтроллере и его системе команд

Как было сказано выше, в качестве примера здесь будет использоваться описание и система команд реально работавшего проекта. Данный микроконтроллер должен был обслуживать разнообразную периферию:

1. Набор регистров.
2. Порт связи с АЦП.
3. Порт связи с ЦАП.
4. Порт связи с реле.

5. Порт связи с последовательной линией связи.
6. Входные и выходные битовые флаги.

Периферия 1–3 представляла собой параллельные регистры и сигналы синхронизации к ним.

Периферия 4 представляла собой последовательный сдвиговый регистр. Для работы на сдвиговый регистр применялся встроенный в FPGA битовый сопроцессор, который выдавал в свой внешний выходной порт последовательность битов. Этот сопроцессор производил обработку сигналов, получаемых от основного процессора, что позволяло проводить программное обслуживание большого числа блокировок, определять возможность включения или выключения реле при различных режимах работы контроллера. Для основного процессора сопроцессор по записи представлялся как набор регистров, а по чтению — как набор битовых флагов. Такое представление сопроцессора удобно тем, что основной процессор записывает в битовый процессор команду, представляющую собой код адреса, и значение бита команды для битового сопроцессора. А по чтению основной процессор использует битовые флаги, так как их удобно использовать в командах перехода по значению бита. Таким образом, разделение алгоритма на два микроконтроллера позволило упростить программирование и разработку обоих узлов.

Периферия 5 представляла собой набор регистров, узел приема-передачи данных и узел пересинхронизации с частоты процессора на частоту работы линии связи.

В качестве элементной базы был задан кристалл серии Virtex2 — xc2v250 fg456-6.

На рис. 1 представлена блок-схема микроконтроллера.

По условиям проекта необходимо было иметь следующие параметры:

- разрядность шины данных — 17 бит;
- тактовая частота — 48 МГц.

Для упрощения и ускорения работы устройства были приняты следующие положения. Все операции должны выполняться за один такт — кроме команд, в которых задается несколько тактов ожиданий. Поскольку быстродействие кристалла и заданная тактовая частота позволяли иметь минимальную длину конвейера команд, то для упрощения проекта был выбран конвейер, содержащий только две ступени.

Исходя из перечисленных выше требований, была выбрана следующая разрядность шин:

- шина данных — 17 бит;
- шина ОЗУ — 17 бит;
- шина ПЗУ — 32 бита.

Процессор был выполнен по гарвардской архитектуре. Память команд загружалась при инициализации FPGA.

Шина команд была выбрана достаточной ширины для размещения в одной команде адреса источника информации и адреса приемника информации. Таким образом, адресная сетка была сделана так, чтобы можно было адресовать память до 2 Кслов внутренней

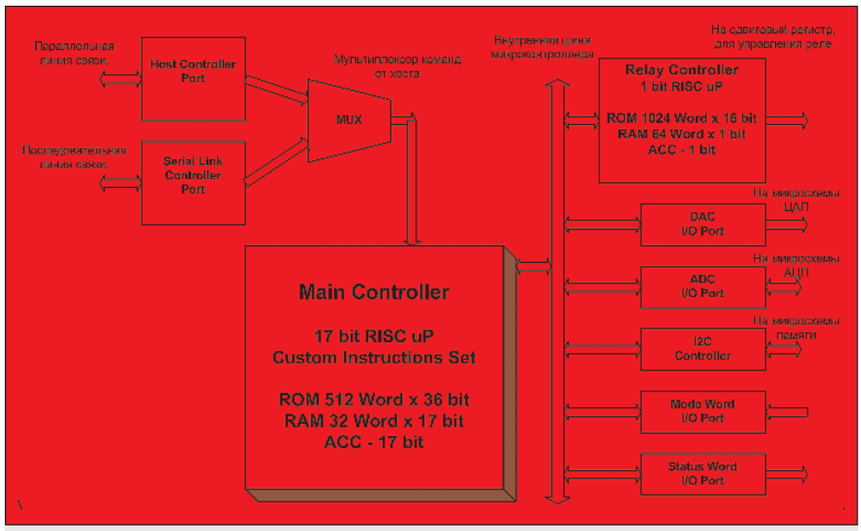

Рис. 1. Блок-схема микроконтроллера

Таблица 1. Адресное поле микроконтроллера

Описание		Адрес											
Addr	Dest	27	26	25	24	23	22	21	20	19	18	17	16
Addr	Src	11	10	9	8	7	6	5	4	3	2	1	0
Src — yes	Dest — yes	1	0	1	1	256 ardd locations — Internal Reg's							
		1	0	1	0	256 ardd locations — I/O Ports							
		1	0	0	1	256 ardd locations — OBC							
		1	0	0	0	256 ardd locations — Internal RAM							
Src — yes	Dest — no	0 (2K Words)	1	1	1	2K Words — Internal ROM							
		0	1	1	0								
		0	1	0	1								
		0	1	0	0								
		0	0	1 (1K Words)	1	1K Words — Internal ROM							
		0	0	1	0								
		0	0	0	1 (512 Words)	512 Words — Internal ROM							
		0	0	0	0								

Примечания:
1. Internal Reg's — область внутренних регистров.
2. I/O Ports — область портов ввода-вывода.
3. OBC — область двухпортовой памяти для записи команд в сопроцессор.
4. Internal RAM — область памяти ПЗУ.
5. Internal ROM — область памяти ОЗУ.
6. Src — область — источник данных.
7. Dest — область — приемник данных.

памяти программ и до 1 Кслова портов и внутренней памяти данных. Также было учтено, что при командах пересылки память команд может выступать только как источник информации, в то время как все остальные узлы могут передавать и принимать информацию. В таблице 1 приведена адресная сетка микроконтроллера и расположение ПЗУ, ОЗУ, портов. В таблице 2 приведены названия внутренних регистров микроконтроллера.

Шина данных и ОЗУ должны были позволять принимать данные от 12-битного АЦП. Данные от АЦП должны были суммироваться 32 раза на сумматоре-аккумуляторе, а затем полученный результат должен был быть разделен на 32. В данном проекте был сделан небольшой компромисс. Поскольку все остальные порты не нуждались в 17-битной арифметике, то команда записи литерала была выбрана 16-битной. Это позволило более свободно оперировать с разрядностью ПЗУ. Но шина данных RAM и стек данных выполнены как 17-битные.

Для упрощения работы с вызовом подпрограмм было сделано два стека — стек данных и стек возвратов. Это позволило отказаться от команд манипуляции данными в стеке и от команд манипуляции указателем стека, кроме обычных команд PUSH и POP. Кроме того, эти стеки принципиально имеют разную разрядность и глубину. Стек возвратов имеет разрядность, соответствующую разрядности счетчика адресов команд. Его глубина определяется числом вложенных вызовов подпрограмм. Стек данных имеет разрядность, соответствующую разрядности шины данных. Его глубина определяется числом слов данных, вложенных в стек.

Процессор имел специальную команду условного перехода SKPLP0, позволяющую

Таблица 2. Внутренние регистры микроконтроллера

#	Название регистра	Адрес	Разрядность	Описание
1	acc	0	17 бит	Аккумулятор
2	stack	1	17 бит	Вершина стека данных
3	loop	2	8 бит	Регистр, в котором производится счет числа циклов
4	shift	3	4 бит	Регистр сдвига данных для проверки битов
5	displ	4	6 бит	Регистр смещения для выполнения команд со смещением

упростить счетчик числа циклов. Эта команда выполнялась по условию, определяемому по флагу, который выставлялся при выполнении заданного числа циклов. Если флаг, указывающий на выполнение числа циклов, имеет значение 0, то команда, следующая за этой командой пропускается. В случае, если флаг установлен в состояние 1, команда SKPLP0 выполняется.

Также процессор имел только одну команду условного ветвления SKPFL0, которая выполнялась по условию, определяемому по одному из флагов. Если флаг, указанный в команде, имеет значение 0, то команда, следующая за этой командой пропускается. Связка из двух инструкций — команды типа SKIP и следующей за ней командой — позволяет иметь достаточно много команд, выполняемых по условию [4]. И, что самое важное в данном случае, это позволяет иметь очень компактный дешифратор команд.

Микропроцессор в данном проекте работал в режиме «запрос-ответ», поэтому контроллер прерываний не использовался.

Описание форматов команд микроконтроллера

Команды можно разделить на группы, название которых приведено (табл. 3). Для упрощения дешифратора команды переходов вынесены в отдельную область в пространстве КОП и имеют КОП от 10H до 1FH. В скобках указаны первые байты кодов команд. Группа команд ALU имеет расширение — подгруппу команд ALU (табл. 4). Как видно из табл. 3 и 4, для данного формата команд имеется достаточно много резервных позиций, так что при необходимости можно легко дополнить набор команд новыми, необходимыми пользователю. Описание действий, выполняемых командами, приводится ниже.

Запись литерала

- **LDI-> ADRR** — запись литерала по непосредственному адресу.
- **LDID** -> ADRR = ADDR + DISPL — запись литерала по адресу, вычисляемому по формуле: адрес, указанный в команде, плюс значение, записанное в регистре смещения.

31 27	26	25 16	15 0
КОП 5 бит	1 бит	Адрес приемника 10 бит	Литерал 16 бит

Таблица 3. Группы команд микроконтроллера

КОП	Команда	КОП	Команда
0	NOP	10	JMP (80)
1	ACC (08)	11	CALL (88)
2	Резерв	12	RET (90)
3	Резерв	13	SKPFL0 (98)
4	PUSH (20)	14	SKPLP0 (A0)
5	POP (28)	15	JMPD (A8)
6	Резерв	16	CALLD (B0)
7	Резерв	17	Резерв
8	MOV (40)	18	Резерв
9	FLAG (48)	19	Резерв
A	LDI (50)	1A	Резерв
B	MOVD (58)	1B	Резерв
C	LDID (60)	1C	Резерв
D...F	Резерв	1D...1F	Резерв

Таблица 4. Подгруппа команд ALU микроконтроллера

КОП	Команда	КОП	Команда
0	Резерв	10	DN_SHIFT2
1	ACC_CLR	11	DN_SHIFT3
2	ACC_NEG	12	ACC_SHR5
3	ACC_DEC	13	ACC_ABS
4	ACC_ADD	14	DISPL_INC
5	ACC_OR	15	DISPL_DEC
6	ACC_AND	16	Резерв
7	ACC_INC	17	Резерв
8	DISPL_SHIFT	18	Резерв
9	DN_SHIFT 1	19	Резерв
A...F	Резерв	1A...1F	Резерв

Команды пересылок

- **MOV** — эта команда пересылает данные и одновременно записывает в стек данные от источника.
  - Регистр — регистр + запись в стек данных источника.
  - Регистр — память + запись в стек данных источника.
  - Память — регистр + запись в стек данных источника.

При записи в поле Wait-State кода, отличного от 0, получается задержка выполнения команды на время, равное значению, записанному в поле «код» + 3 такта синхрочастоты.

- **MOVD** -> ADRR = ADDR + DISPL — пересылка данных по адресу, равному адресу, указанному в команде, плюс значение, записанное в регистре смещения.

Остальное — аналогично команде **MOV**.

31 27	26	25 16	15 12	11 0
КОП 5 бит	Запись в стек данных 1 бит	Адрес приемника 10 бит	Wait-State 4 бит	Адрес источника 11 бит

Команды управления выходными битовыми флагами

FLAG — данная команда производит установку-сброс битовых флагов. В поле «Адрес приемника» записывается значение адреса порта, «отвечающего» за дешифрацию команды. В поле «Адрес флага» записывается значение адреса флага, в поле «Состояние флага» записывается то состояние, в которое флаг должен быть установлен данной командой.

31 27	26	25 16	15 12	11	10 5	4 0
КОП 5 бит	1 бит	Адрес приемника 10 бит	Wait-State 4 бит	Состояние флага 1 бит	Резерв 6 бит	Адрес флага 5 бит

При записи в поле Wait-State кода, отличного от 0, получается задержка выполнения команды на время, равное значению, записанному в поле «код» + 3 такта синхрочастоты.

Команды ветвления безусловные

- **JMP** — переход по адресу, указанному в команде.
- **CALL** — вызов подпрограммы по адресу, указанному в команде.
- **JMPD** -> ADRR = ADDR + DISPL — переход по адресу, равному адресу, указанному в команде, плюс значение, записанное в регистре смещения.
- **CALLD** -> ADRR = ADDR + DISPL — вызов подпрограммы по адресу, равному адресу, указанному в команде, плюс значение, записанное в регистре смещения.

31 27	26 12	11 0
КОП 5 бит	Резерв 15 бит	Адрес перехода 12 бит

- **RET**
- **NOP**

31 27	26 0
КОП 5 бит	Резерв 16 бит

Команды ветвления условные

- **SKIP – SKPFL0** — если флаг равен 0, то следующая команда будет пропущена.
- **SKPLP0** — если флаг равен 0, то циклы будут продолжены, а если флаг равен 1, то произойдет выход из цикла.

31 27	26 4	3 0
КОП 5 бит	Резерв 23 бит	Адрес флага 4 бит

Команды работы со стеком данных

- **PUSH** — сохранение данных, адрес которых указан в поле «Адрес источника» в стеке данных.

31 27	26 12	11 0
КОП 5 бит	Резерв 15 бит	Адрес источника 12 бит

- **POP** — восстановление данных из стека данных по адресу, который указан в поле «Адрес приемника».

31 27	26	25 16	15 0
КОП 5 бит	Резерв 1 бит	Адрес приемника 10 бит	Резерв 16 бит

Команды, выполняемые в ALU и в специальных регистрах

1. **ACC_CLR** — очистка аккумулятора.
2. **ACC_NEG** — побитная инверсия данных аккумулятора.
3. **ACC_DEC** — декремент данных в аккумуляторе.
4. **ACC_ADD** — сложение данных в аккумуляторе с данными источника информации.

5. **ACC_OR** — операция «ИЛИ» данных в аккумуляторе с данными источника информации.
6. **ACC_AND** — операция «И» данных в аккумуляторе с данными источника информации.
7. **ACC_INC** — инкремент данных в аккумуляторе.
8. **DISPL_SHIFT** — сдвиг данных в регистре смещения на 1 позицию.
9. **DN_SHIFT1** — сдвиг данных в регистре DN на 1 позицию.
10. **DN_SHIFT2** — сдвиг данных в регистре DN на 2 позицию.
11. **DN_SHIFT3** — сдвиг данных в регистре DN на 3 позицию.
12. **ACC_SHR5** — деление знакового числа на 32.
13. **ACC_ABS** — преобразование знакового числа в абсолютную величину.
14. **DISPL_INC** — инкремент данных в регистре смещения.
15. **DISPL_DEC** — декремент данных в регистре смещения.

31 27	26 16	15 12	11 0
КОП 5 бит	Код операции ALU 11 бит	Резерв 4 бит	Адрес источника 12 бит

Ассемблер и кодировка команд микроконтроллера

Для «классических» микроконтроллеров кодировка команд и их мнемоника на ассемблере были очень простыми. Пример наиболее известной команды пересылки — «MOV B, A;». Здесь есть сама команда, то есть КОП, источник данных и приемник данных. Такая кодировка удобна в тех случаях, когда команда выполняет только одно действие. Для ее описания необходимы только три параметра. Однако во встроенных в FPGA микроконтроллерах выгоднее применять команду с «длинным словом». При этом появляется необходимость закодировать в мнемонике все действия, выполняемые данной командой. Примером здесь может служить команда пересылок, описанная выше и имеющая 5 полей, или команда управления флагами, также имеющая 5 полей. В том случае, когда эти действия понятны программисту, строка на ассемблере выглядит как, например, строка в Форте [6].

```
0C03 mova mem [cs] -t
2C13 rpop [ss] -n
214B load 2 (a2) [ds] -t
07EB load +(a4) [ss] –n
```

Рис. 2. Примеры строк команд на Форте

Такая мнемоника удобна только в том случае, когда сам микроконтроллер уже «устоялся», и его состав, адресация и название составных узлов уже «забронзовели». Но при такой мнемонике сделать ассемблер довольно затруднительно, поскольку нужно производить разборку довольно сложной текстовой строки.

Как показывает опыт, такие проекты, в которых имеются устоявшиеся значения адресов и набор регистров уже определен, отлажен и не будет изменяться в течение длительного времени, встречаются довольно редко. Чаще бывает наоборот, когда приходится очень оперативно добавлять или убирать регистры и флаги. При этом, как показала практика, очень удобно производить корректировки сразу в трех документах:

- в ассемблере, симуляторе,
- в файле на языке Verilog, описывающем микроконтроллер,
- в техническом описании.

Если реализация фрагмента программы для микроконтроллера вызывает трудности, то тут же добавляется регистр или флаг, позволяющий облегчить задачу. Если какой-либо регистр или команда окажутся незадействованными, они тут же удаляются из проекта. При этом, возможно, поменяется и адресация у других регистров или флагов. Поэтому в таких проектах хочется, чтобы мнемоника или описание команды было связано с именем периферии, а не с номером регистра или флага. И, как показалось автору в начале работы над проектом, здесь было непреодолимое противоречие. Ассемблер получался неимоверно сложным и не гибким. Но вся сложность была в том, что эту задачу нельзя было решать известными старыми способами. То есть все дело было в том, что старые программы-ассемблеры вышли из времен CM и DOS: ручная набивка команд и последующая их декодировка за два прохода. Вот здесь-то и есть то самое «бутылочное горлышко». Как только это стало понятно, то решение пришло само! Теперь оставалось только заменить DOS-технологию на современную Windows-технологию. Команды не надо набивать вручную. Надо обычным для Windows способом при помощи мыши, флажков и выпадающих меню выбрать параметры для полей каждой команды. При этом программа сама произведет кодировку команды в машинные коды, добавит к ним комментарии и т. д. Единственное, что не может сделать такая программа в первом проходе, так это расставить адреса переходов. Но это действие легко выполняется при ассемблировании полученного кода, так как теперь от всего ассемблирования остается только процедура поиска адресов переходов и кодирование их в машинных кодах. Процесс неимоверно упрощается.

Некоторые из читателей могут возразить: «это, мол, надо каждую команду «наколотить» мышкой, да я руками много быстрее… «. Возможно, дело в опыте и в привычке. Но для встроенного микроконтроллера длина кода не так важна. Скорее наоборот, код для «самодельного» микроконтроллера будет небольшим, не более пары килослов. Отдельные его фрагменты будут копироваться несколько раз. Так что попробуйте, возможно, все не так сложно, как кажется в начале.

Далее будет показано, как сделать удобный для разработчика программный инструмент.

Гибкость и еще раз гибкость

Где ассемблер выполняет свою работу? В кодировании полей команды. Это значит, что если список мнемоник команд подгружать из файла, то мы получаем единый файл для всех документов, входящих в проект. То же относится и к наименованию портов и флагов. Достаточно просто вставить их названия в исходный файл описания портов и флагов, который будет подгружаться при запуске программы, и проблемы с названиями и адресами будут закрыты. И теперь пользователю не надо будет вспоминать, какая периферия подключена к какому порту и по какому адресу.

Программный инструмент, примененный автором, показан на рис. 3.

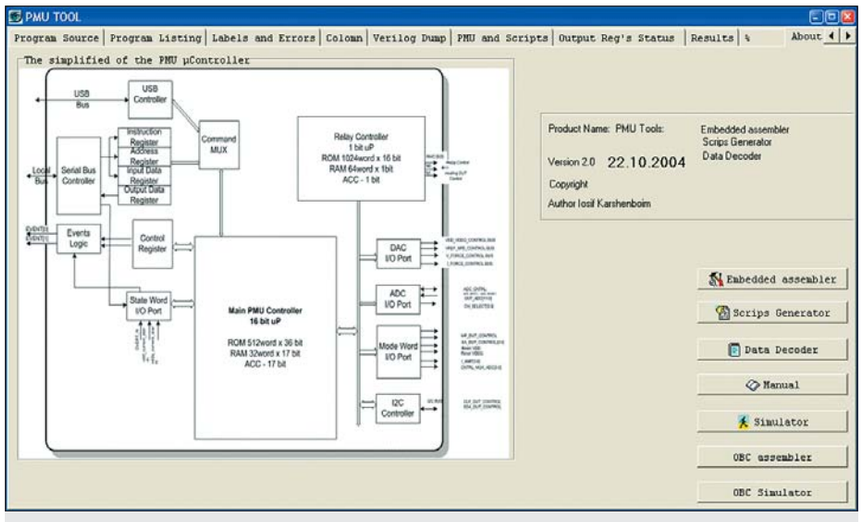

Рис. 3. Программный инструмент для работы со встроенным микроконтроллером

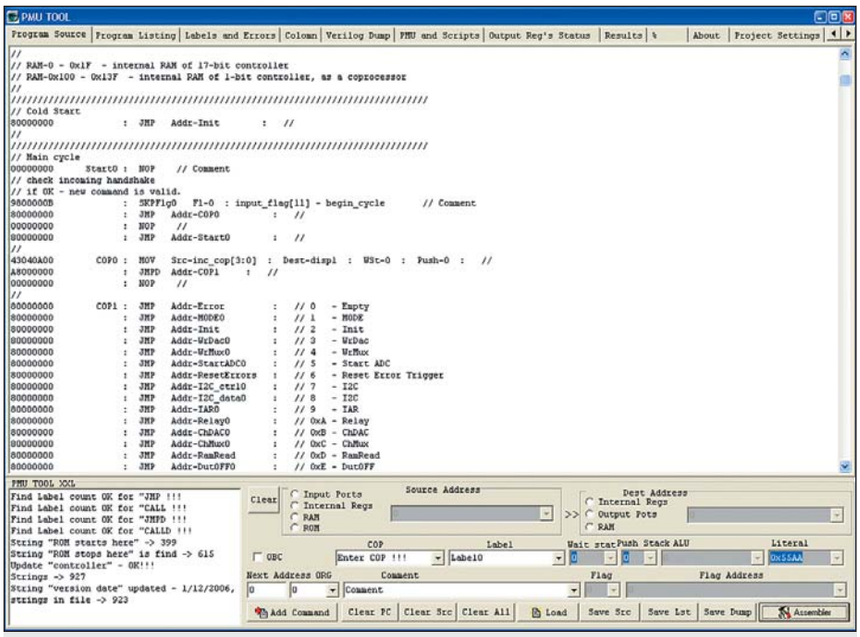

Рис. 4. Окно создания исходного ассемблерного файла — «Program Source»

Программа была написана на Borland C Builder-6. В данной статье на примере этой программы показана методика разработки подобных программных инструментов.

Окно создания исходного ассемблерного файла показано на рис. 4. Программа работает следующим образом. Пользователь выбирает код команды в поле со списком «COP». Когда код команды выбран, программа подгружает в поле со списком источника данных список узлов микроконтроллера, которые могут быть источниками данных. В поле со списком приемника данных программа подгружает список узлов микроконтроллера, которые могут быть приемниками данных. Если у данной команды нет источника данных или приемника данных, то соответствующее поле становится серым. Соответственно, подгружаются и окна ввода для комментариев по данной команде. Пользователь имеет возможность изменить текст комментария к данной строке программы. При обращении к ячейке ОЗУ пользователь указывает адрес требуемой ячейки. Затем он выбирает из поля со списком источника данных требуемый ему для данной команды источник, например, «данные из АЦП». Далее, пользователь выбирает из поля со списком приемника данных требуемый ему для данной команды приемник, например, «регистр-аккумулятор». Устанавливаются дополнительные параметры, такие как запись в стек и такты ожидания. При нажатии кнопки «Add command» код данной команды добавляется к листингу, находящемуся в окне «Program Source» (рис. 4). Кодировщик команды в таком варианте очень прост. Нет необходимости разбирать сложный синтаксис строки на ассемблере. Вместо этого надо просто считать индексы из полей со списком источника и приемника, состояние дополнительных кнопок для кодировки полей, сдвинуть коды индексов и поместить их в нужное поле команды.

Продолжение следует.

Литература

1. Семенов Н., Каршенбойм И. Микропрограммные автоматы на базе специализированных ИС // Chip News. 2000. № 7.
2. Каршенбойм И. Микропроцессор своими руками // Компоненты и технологии. 2002. № 6, 7.
3. Каршенбойм И. Микропроцессор своими руками — 2. Битовый процессор // Компоненты и технологии. 2003. № 7, 8.
4. Каршенбойм И. Микропроцессор для встроенного применения Nios. Система команд и команды, определяемые пользователем // Компоненты и технологии. 2002. № 8.
5. Каршенбойм И. Микропроцессор для встроенного применения Nios. Конфигурация шин и периферии // Компоненты и технологии. 2002. № 2–5.
6. Каршенбойм И. Стековые процессоры, или Новое это хорошо забытое Новое // Компоненты и технологии. 2003. № 9; 2004. № 1, 2.
7. Каршенбойм И., Паленов К. «Встроенный» логический анализатор — инструмент разработчика «встроенных» систем // Схемотехника. 2001. № 12.
8. Каршенбойм И. Г. Между ISE и ViewDraw. Компоненты и технологии. 2005. № 6.
9. Ken Chapman. Creating Embedded Microcontrollers (Programmable State Machines). Xilinx. 03/28/2002. http://www.xilinx.com
10. 8-bit Microcontroller for Virtex Devices. Xilinx Application Note 213.