KiT#57(4).qxd 4/10/2006 1:00 PM Page 8

Окончание. Начало в № 3`2006

Микропроцессор своими руками—3.
Ассемблер и софт-симулятор

Иосиф КАРШЕНБОЙМ
www.iosifk.narod.ru

Описание работы ассемблера

Примеры работы ассемблера приведены на рис. 5. Данный пример не является фрагментом какой-либо программы, а только представляет работу ассемблера при кодировке команд. На взгляд автора, кодировка команд, приведенная в примерах, гораздо проще для понимания, следовательно, работа с ней более эффективна. К этому надо добавить, что при таком способе работы отпадает надобность в сложных алгоритмах проверки ассемблерного кода на ошибки синтаксиса. Поля команды, отделенные двоеточиями, могут быть выровнены путем добавления пробелов или табуляции. Единственное, что остается проверить — это правильность расстановки меток.

Необходимо также сказать и об окне ввода адреса. Если нужно оставить часть поля адресов свободным, можно установить в окне ORG тот адрес, где будут расположены следующие команды. При нажатии на кнопку Add command к тексту добавится команда Org <Addr>, например так, как показано на рис. 5 (строка 2).

Как выполняется ассемблирование

При нажатии на кнопку Assembler программа начинает создавать файл листинга. На рис. 6 показано окно программы с листингом, выполненным в результате ассемблирования исходного текста пользовательской программы.

При генерации листинга в его начало помещается заголовок вида, представленного на рис. 7.

```
////////////////////////////////////////////////////////////
// File Name — D:\FPGAExpr_Prj\.....\Cmd_rev3ax_111.pmu
// Date, Time — 18.10.2004 13:16:06
////////////////////////////////////////////////////////////
```

Рис. 7. Пример заголовка листинга, сгенерированного программой

Имя файла и дата его обработки помогают избежать ошибок, связанных с архивацией файлов и с заменой версии «прошивки».

Далее программа проходит по строкам листинга в окне «Program Source» и производит построчную обработку текста.

```
530007FF        : LDI Dest-acc                  : Lit-0x7FF          : // Comment
Org 5
5129FFFF        : LDI Dest-RAM-0x129            : Lit-0xFFFF         : // Cm29
40140B00        : MOV Src-acc                   : Dest-RAM-0x14      : WSt-0  :  Push-0 :// only 3 bits
A0000000        : SKPLp0 //
80000069        : JMP ADDR-OBC_EXECUTE1         : //
9800000A WaitOBC1 : SKPFlg0 Fl-0                : input_flag[10] — ~ obc_busy //
4A100003        : FLAG output_flag[3] — dac_cs  : Fl-0               : WSt-0  :  //
90000000        : RET //
08060A08        : ACC_AND Src-dn_limit          : // CMG2
59001A04 Label0 : MOVD Src-adc_data[11:0]       : Dest-RAM-0x100     : WSt-1  :  Push-0  :  //
```

Рис. 5. Примеры синтаксиса команд, сгенерированных программой

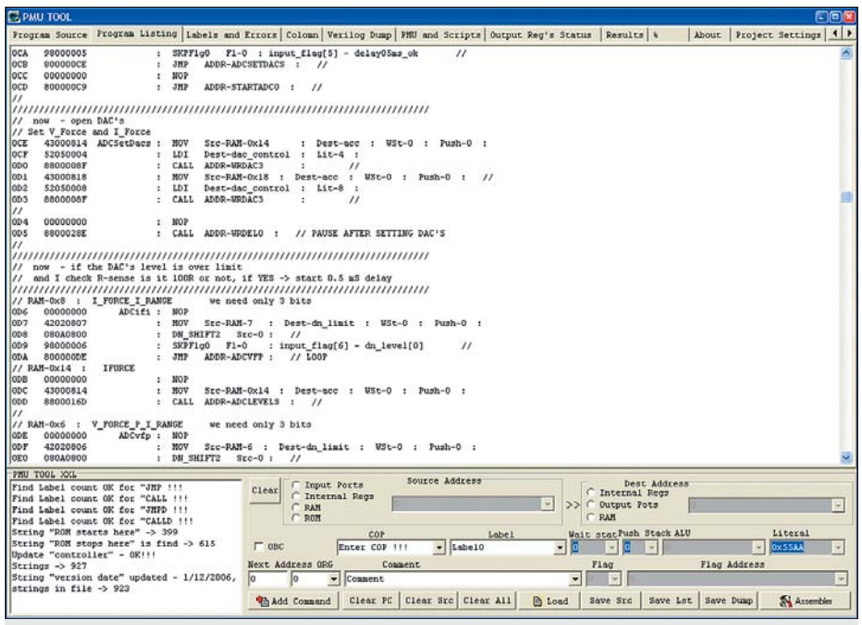

Рис. 6. Пример листинга, сгенерированного программой

При обработке текста выполняются следующие действия:

1. Выделяется текущая строка, удаляются служебные символы и пробелы в начале и в конце строки.
2. Первые символы в этой строке проверяются на соответствие «//».Если это так, то в этой строке есть только комментарии, и программа должна перейти к разборке следующей строки. Строка с комментариями при этом копируется в окно Program Listing.
3. При переборе строк производится подсчет строк, начиная с 0-й, при этом строки с комментариями не учитываются. Счетчик строк будет соответствовать адресу строки программы. Если данная строка представляет собой строку с командой, то в окно Program Listing копируется ее адрес, а затем содержимое.
4. Если в текущей строке программа находит директиву ORG, то производится заполнение памяти командами NOP до адреса, предшествующего указанному в директиве (до Addr — 1, где Addr взят из директивы Org Addr).
5. Проверяется, есть ли в данной строке подстрока JMP, CALL, JMPD или CALLD. Если такая строка найдена, то ищется название метки перехода, которое должно быть

KiT#57(4).qxd 4/10/2006 1:00 PM Page 9

помещено в строке после JMP ADDR-. Если название метки не найдено — выход по ошибке. Если название метки найдено, то программа начинает сканировать весь листинг в поисках метки, куда должен быть сделан переход. Для этого так же, как в п. 1, выделяется строка и в ней производится поиск подстроки с именем метки. Когда адрес метки найден, производится окончательное кодирование команд JMP, CALL, JMPD и CALLD. Так же как и для других команд, в окно «Program Listing» копируется адрес данной строки, а затем ее содержимое. Для определения правильности выполнения этой операции производится подсчет найденных команд переходов и числа найденных меток. Если эти счетчики имеют одинаковые значения, то все операции по кодированию адресов переходов выполнены правильно. На рис. 6 и 8 приведены фрагменты листинга пользовательской программы. Для удобства работы с листингом можно использовать любые редакторы текста, позволяющие производить выделение цветом для ключевых слов, например EditPlus2.

6. После того как кодировка команд завершена и листинг сформирован, необходимо сделать эквивалент «прошивки». Поскольку аппаратная платформа известна, то следующим проходом программа преобразовывает листинг к формату данных для инициализации памяти. На рис. 9 представлено окно программы Verilog Dump, в котором формируется дамп данных для инициализации памяти программ.
7. В «обычных» микроконтроллерах работа ассемблера на этом заканчивается. Но в данном проекте выполняется еще два шага. В окне настроек программы указываются следующие параметры:
   - объем памяти программ микроконтроллера — 0,5 Кслов, 1 Кслово или 2 Кслова;
   - расположение в папке проекта и название файла для микроконтроллера, в котором описано применение памяти программ, например …\controller.v;
   - необходимость производить замену «прошивки» в файле.

```
////////////////////////////////////////////////////////////////////////////////
// Cold Start
000 8000002F : JMP ADDR-INIT : //
//
////////////////////////////////////////////////////////////////////////////////
// Main cycle
001 00000000 Start0 : NOP // Comment
// check incoming handshake
// if OK — new command is valid.
002 9800000B : SKPFlg0 Fl-0 : input_flag[11] — begin_cycle // Comment
003 80000006 : JMP ADDR-COP0 : //
004 00000000 : NOP //
005 80000001 : JMP ADDR-START0 : //
//
006 43040A00 COP0 : MOV Src-inc_cop[3:0] : Dest-displ : WSt-0 : Push-0 : //
007 A8000009 : JMPD ADDR-COP1 : //
008 00000000 : NOP //
//
```

Рис. 8. Внешний вид листинга пользовательской программы

Программа проверяет флажок, разрешающий производить замену «прошивки» в файле. Если эта опция разрешена, то программа открывает файл с описанием памяти программ. Для того, чтобы произвести замену «прошивки», в программе используются два буфера типа TStringList. Исходный файл программа помещает в первый внутренний буфер. Таким файлом может быть файл микроконтроллера или отдельный файл, в котором описан только блок ROM. Далее программа сканирует строки в буфере до строки «//ROM starts here». Когда файл в первом буфере сканируется построчно, то строки, взятые из исходного файла и находящиеся до ключевой строки — «//ROM starts here», копируются во второй буфер. Если программа находит строку «//ROM starts here», то далее во второй буфер добавляется описание блока памяти нужного объема, при этом описание выполнено таким образом, что к блоку памяти подключены все требуемые шины и сигналы управления. Затем добавляется фрагмент текста, описывающий инициализацию этой памяти. После этого добавляется строка «//ROM stops here». В исходном тексте также ищется строка «//ROM stops here», и после того как она находится, весь остальной текст из старого файла, находящегося в первом буфере, добавляется к новому файлу, который формируется во втором буфере. Далее исходный файл переименовывается в файл с расширением *.bak. Новый файл из второго буфера записывается «поверх» старого файла.

На рис. 10 приведен фрагмент файла описания микроконтроллера со «вставленными» в него кодами инициализации памяти программ. Здесь приведены коды только для одной «половинки» ПЗУ — для старшего слова. Для младшего слова коды инициализации памяти программ формируются аналогичным образом.

```
// ROM starts here
RAMB16_S18 ROMA // program space
                ( .WE (1'b0),
                  .EN (ram_rd_ena),
... далее фрагмент описания блоков памяти пропущен.
...
...

// File Name — D:\....REV30X_VD\Cmd_rev3ax_106.pmu
// Date, Time — 12.10.2004 11:17:56
// 1K word 2 x 1024 x 18
// A — Hi word
// B — Low word
//
//synthesis attribute INIT_00 of ROMA is «0….B»
//synthesis translate_off

defparam ROMA.INIT = 18'h0;
defparam ROMA.SRVAL = 18'h0;
defparam ROMA.WRITE_MODE = «WRITE_FIRST»;

defparam ROMA.INIT_00 = 256'h0….B;

//synthesis translate_on
// ROM stops here
```

Рис. 10. Фрагмент файла описания микроконтроллера со «вставленными» в него кодами инициализации памяти программ

8. Программа ищет в новом файле строку «assign version_date = 24'h121004;» и производит замену последних цифр на текущее число, месяц и две цифры года. В дальнейшем это значение считывается как дата изготовления «прошивки».

Таким образом программа ассемблера подготавливает проект к компиляции. Остается только запустить компиляцию проекта в ISE или симуляцию в Modelsim и проверить результат.

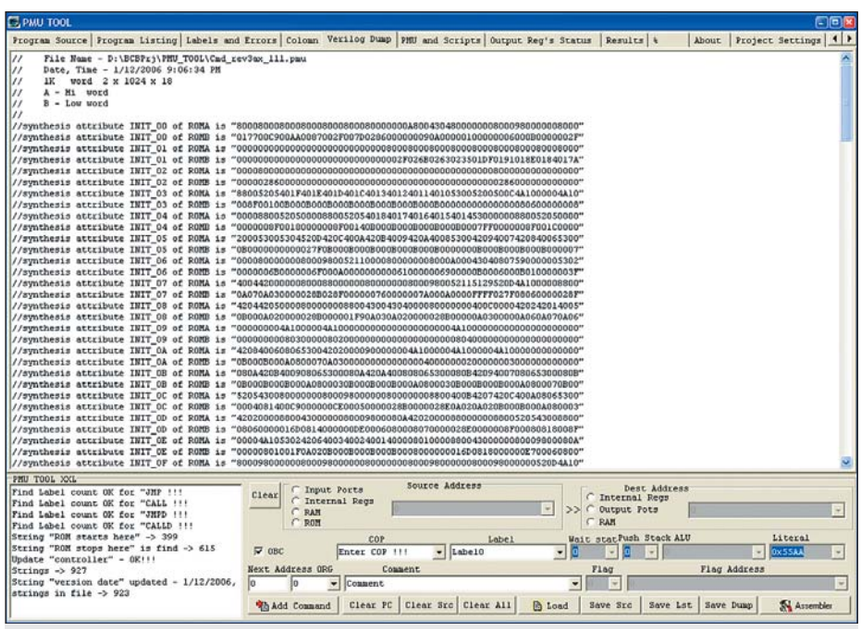

Рис. 9. Окно программы Verilog Dump, в котором показано, как программа формирует дамп данных для инициализации памяти программ

KiT#57(4).qxd 4/10/2006 1:00 PM Page 10

Как ассемблер ищет метку

Опишем подробнее процесс поиска метки ассемблером (см. п. 5 предыдущего раздела). Рассмотрим формат строки, содержащей метку (рис. 5):

```
006    43040A00    COP0  :  MOV....
```

Строка состоит из нескольких слов, разделенных пробелами. Первое слово — адрес этой строки в поле памяти программ, второе слово — команда в машинных кодах, затем пробелы, затем метка. После метки — снова пробелы и символ «:»(двоеточие). Следовательно, чтобы выделить метку, надо выполнить следующие действия:

1. Установить указатель на начало строки и произвести выделение двух первых слов.
2. Проанализировать символы, находящиеся в строке между концом второго слова и символом «:». Если все эти символы являются пробелами, то это значит, что в данной строке метки нет. Если после второго слова появился символ пробела, а после него — любые другие символы, то это будут символы метки. Символы метки должны закончиться символом пробела или двоеточием.

Исходя из этого, целесообразно иметь две функции — функцию проверки наличия метки в строке и функцию, возвращающую имя метки.

Софт-симулятор

Когда проект, для которого разрабатывался описываемый микроконтроллер, только начинался, автору казалось, что проверить правильность работы пары сотен команд — дело не очень трудное. И это подтверждалось предыдущим опытом использования встроенных микроконтроллеров. Тем более, что обычно встроенные микроконтроллеры не выполняют какой-либо сложной обработки данных. Обычный алгоритм для контроллера ввода-вывода — это проверить и получить данные в одном месте, проверить и передать в другое место. Для проверки работы микроконтроллера хватало листа бумаги и карандаша. Кроме того, часть программ удавалось отладить в Modelsim еще на стадии разработки ядра микроконтроллера или его периферии.

Но задачи, с которыми пришлось столкнуться автору в описываемом проекте, потребовали значительно более развитого алгоритма управления. Мало того, аппаратура, для управления которой проектировался микроконтроллер, была выполнена «на грани выгорания», то есть любое неправильное действие по управлению ЦАП или реле приводило к аварийному отказу. Большое количество блокировок, порядок подачи команд в ЦАП, состояния реле, которых было около 50 штук, и другие усложнения алгоритма заставили взяться за разработку софт-симулятора.

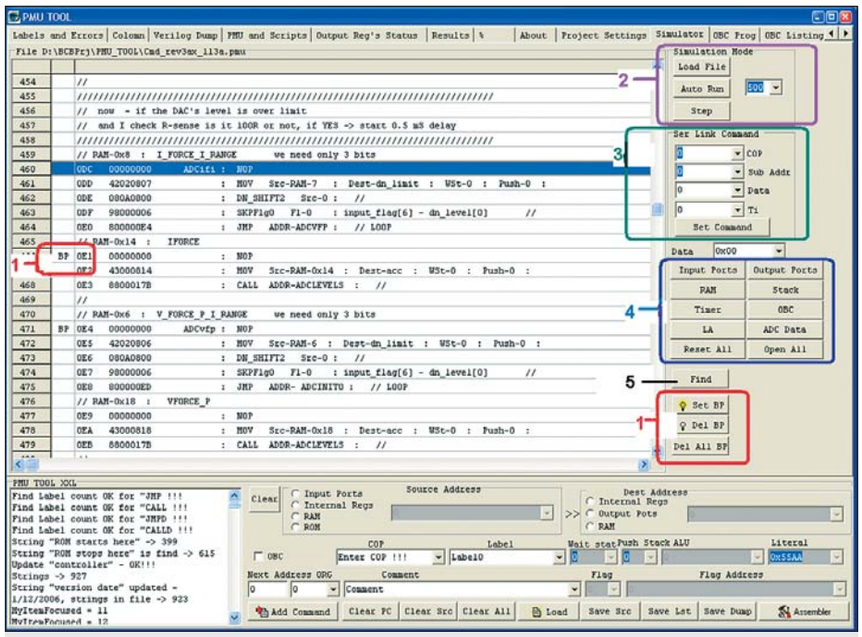

Рис. 11. Окно симуляции выполнения программы встроенного микроконтроллера
1 — слева — установленная точка останова; справа — кнопки управления остановом программы;
2 — загрузка файла и шаговая/циклическая трассировка;
3 — имитация данных, проходящих от линии связи с хостом;
4 — управление окнами, показывающими состояние процессора и периферии;
5 — кнопка вызова формы поиска

Итак, что должен делать софт-симулятор? Есть листинг программы. Этот листинг является результатом работы ассемблера. Есть описание команд. Известно, какие команды выполняются и за сколько тактов. Надо показать, как выполняются команды в микроконтроллере. Как сделать такую программу?

Дальнейшее описание будет ориентировано на использование Borland C Builder-6, однако применение именно этого программного обеспечения не является необходимым для выполнения софт-симулятора.

Проект софт-симулятора состоит из следующих частей:

1. Поле редактирования, в которое будет загружаться листинг.
2. Узлы микроконтроллера представлены набором полей редактирования или форм с полями редактирования.
3. Панель управления с кнопками, определяющими режим работы софт-симулятора.

Поле редактирования (рис. 11) выполнено в виде таблицы, имеющей три столбца. Первый столбец — номер строки, второй — точки останова и третий — текст команды программы пользователя. При этом очень удобно использовать табличное представление данных для чтения из ячейки, записи в ячейку и выделения ячейки цветом. А главное, очень удобно производить построчную прокрутку и сдвиг при выполнении команд переходов. Для поиска текста в окне редактирования используется окно для поиска (рис. 12).

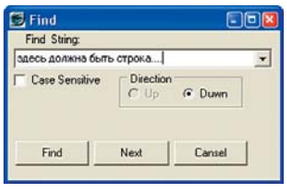

Рис. 12. Окно для поиска текстовой строки в симуляторе программы

Узлы микроконтроллера представлены в виде отдельных форм с полями редактирования, выполненных как таблицы или как небольшие окна ввода (рис. 13).

Поле памяти данных тоже выполнено в виде таблицы. Во многих симуляторах поле памяти данных представляет собой дамп шестнадцатеричных кодов. Однако для данного проекта объем памяти данных потребовался относительно небольшой. Кроме того, выполнение поля памяти в виде таблицы позволило несколько повысить удобство работы с ним. Дело в том, что для проекта было применено статическое распределение данных по адресам. А комментарии с описаниями типов данных и их «принадлежностью» подгружались из файла. Таким образом, в каждой строке находились значение адреса ячейки памяти, значение данных в этой ячейке и комментарий.

Дополнительным сервисом является форма с окном «логического анализатора». Автору было привычнее видеть последовательность

KiT#57(4).qxd 4/10/2006 1:00 PM Page 11

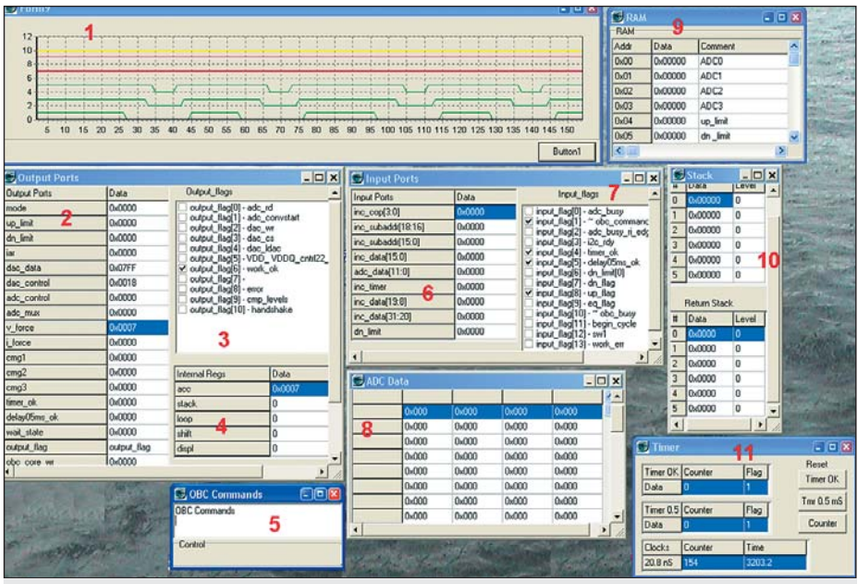

Рис. 13. Дополнительные окна симулятора. Эти окна показывают состояние памяти, регистров и других узлов микроконтроллера
1 — окно логического анализатора; 2 — выходные порты; 3 — выходные флаги; 4 — регистры процессора; 5 — команды в сопроцессор; 6 — входные порты; 7— входные флаги; 8 — данные от АЦП; 9 — память данных; 10 — стек данных и стек возвратов; 11 — программируемый интервальный таймер, таймер на 0,5 мс и счетчик тактов при выполнении программы

обмена сигналами с внешними по отношению к микросхеме FPGA узлами модуля в виде «осциллограмм». Эти внешние узлы, например сигналы готовности АЦП, имитировались в софт-симуляторе программным способом.

Кроме имитации сигналов готовности от АЦП в данном софт-симуляторе оказалось удобно имитировать чтение данных из АЦП путем загрузки в соответствующую форму файла с информацией, имитирующей данные, прочитанные из АЦП.

Управление режимом работы симулятора

Название кнопки Load File (см. рис. 11) говорит само за себя, поэтому описывать ее использование мы не будем. Кнопка Set BP — установить точку останова, Del BP — снять точку останова, Del All BP — снять все точки останова. Кроме того, есть набор кнопок, позволяющих открывать окна для работы с отдельными узлами микроконтроллера.

При нажатии на кнопку «Step» симулятор обрабатывает текущую строку. Любая строка может быть установлена как текущая, если по ней дважды щелкнуть мышкой. При этом она выделяется цветом и перемещается в окне на 4 позицию сверху. Если ни одна из строк не выбрана пользователем как текущая, то программа начинает с обработки первой строки в таблице. Строка анализируется на наличие в начале подстроки символов «//». Если символы найдены, то считается, что эта строка содержит только комментарии. Программа переходит к обработке следующей строки. Для этого выполняется прокрутка вниз на одну строку и анализируется следующая строка. Если данная строка не содержит комментарии, то из нее выделяется команда в машинных кодах. Из этой команды выделяется поле, соответствующее КОП, и кодировка этого поля проверяется на соответствие с разрешенными для данного проекта кодировками команд. Затем выделяются поля по полученному КОП в соответствии с описаниями на данную команду. Остается только «выполнить» действия, определяемые данной командой. Например, для команды «Записать литерал по адресу XX» надо из команды выделить поле литерала и эти данные записать в таблицу памяти в ячейку, соответствующую адресу XX. После выполнения очередной команды производится прокрутка листинга вниз и выделение синим цветом следующей строки.

Несколько слов о таймерах. Если таймер был взведен, то при симуляции очередной команды в таймере должен производиться счет. При выполнении команд переходов, которые выполняются за несколько тактов, необходимо делать изменение счетчиков в таймере на это же число тактов, а в форме с окном «логического анализатора» необходимо перемещать графики по шкале времени на два такта.

Порядок выполнения симуляции

Симуляция может выполняться в пошаговом режиме при нажатии на кнопку Step. Если нажать на кнопку Auto Run, то симуляция будет выполняться последовательно команда за командой с интервалом, указанным в окне, которое расположено рядом с кнопкой Auto Run. Циклическая симуляция будет выполняться либо до конца файла, либо до первой из установленных точек останова.

При повторном нажатии на кнопку Auto Run циклическая симуляция останавливается.

Точки останова

Программа софт-симулятора может содержать столько точек останова, сколько строк имеется в отлаживаемой программе. Для того, чтобы установить точку останова в данной строке кода, надо заполнить пустое поле во второй колонке, расположенной слева от адреса команды (рис. 11). При выполнении симуляции проверяется, записано ли что-нибудь в колонке точек останова. Если эта ячейка в таблице не пустая, то симулятор выполняет останов. Если текущая ячейка для точки останова пуста, то симулятор переходит к следующей строке.

Изменение значения данных и флагов во время симуляции

Для того, чтобы можно было оперативно менять данные в регистрах и в счетчиках таймера, а также значения битовых переменных, необходимо выполнить программу таким образом, чтобы при выполнении симуляции эти значения считывались не из внутренних переменных, а непосредственно из соответствующих окон ввода, расположенных на окнах программы (на формах).

Тогда в окнах ввода можно задавать любое допустимое значение данных, а битовые сигналы переключать двойным щелчком мышки.

Несколько строк кода

Приведенные на рис. 14–19 несколько строк кода на C++ предназначены только для иллюстрации отдельных деталей проекта. Основное назначение данных фрагментов — показать читателю примеры, которые помогут им в дальнейшем при разработке аналогичных программ. Подробно комментировать каждый фрагмент кода мы не будем.

Запись данных в ячейку таблицы:

```
Form1->StringGrid1->Cells[3][0] = «Voltage»;

Пример работы с полем со списком:
Form1->ComboBox8->Items->Clear(); — очистка выпадающих строк
Form1->ComboBox8->Text = «input_flags»; — запись текста
Form1->ComboBox8->Items->Add(«input_flag[0] — adc_busy»); — запись в выпадающие строки
```

Рис. 14. Пример кодов для работы с таблицей и полем со списком

KiT#57(4).qxd 4/10/2006 1:00 PM Page 12


```
//--------------------------------
if(MyItemFocused>4)
 Form1->StringGrid10->TopRow=MyItemFocused-4;
else
 Form1->StringGrid10->TopRow=MyItemFocused;
//--------------------------------
TGridRect myRect;
myRect.Left = 2;
myRect.Top = MyItemFocused;
myRect.Right = 2;
myRect.Bottom = MyItemFocused;
Form1->StringGrid10->Selection = myRect;
//--------------------------------
```

Рис. 15. Этот фрагмент показывает, как можно перемещать строки таблицы и выделять их цветной полосой

```
[RAM_Form]
Top=219
Left=-1119
Width=300
Height=687
Monitor=0
InitMax=0
```

Рис. 18. Фрагмент файла инициализации

```
// проверка на наличие в строке символов «//»
FindStr = (RichEdit1->Lines->Strings[j]).Pos(«//»);
 if(!((FindStr!=NULL) && ((FindStr<5))))
 {
 if((Form1->RichEdit1->Lines->Strings[j])!=»»)
 {
 // здесь делается новая строка, если она в начале не содержит «ORG» или «//»…
```

Рис. 16. Работа со строкой, не являющейся комментарием

```
void LoadIniRAM (void)
{
 TIniFile *ini;
 ini = new TIniFile(ChangeFileExt( Application->ExeName, «.INI» ));
 Form4->Top = ini->ReadInteger( «RAM_Form», «Top», 100 );
 Form4->Left = ini->ReadInteger( «RAM_Form», «Left», 100 );
 Form4->Width = ini->ReadInteger( «RAM_Form», «Width», 300 );
 Form4->Height = ini->ReadInteger( «RAM_Form», «Height», 695 );
 Screen->Monitors[ini->ReadInteger( «RAM_Form», «Monitor», 0)];
// ini->ReadBool( «RAM_Form», «InitMax», false ) ? Form4->WindowState == wsMaximized : Form4->WindowState == wsNormal;

 delete ini;
}
```

Рис. 17. Чтение параметров из файла инициализации

```
Form2->CheckListBox1->Checked[4]=true; // «input_flag[4] — timer_ok»
```

Рис. 19. Фрагмент файла управления битовыми переменными

Способ управления размером и местоположением формы (рис. 17 и 18) позволяет позиционировать окна программы симулятора и зафиксировать их размеры по выбору пользователя. Программа запоминает, на каком из мониторов находилось окно при завершении программы и заносит эти данные в файл инициализации. Значение «Left=-1119» определяет, что окно должно запуститься на левом мониторе. Код, необходимый для записи параметров, почти аналогичен приведенному коду для чтения.

Фрагмент формы, на котором отображаются битовые переменные, выполнен как CheckListBox (см. рис. 13). Битовые переменные представлены в виде флажков. Управление ими показано на рис. 19.

Из приведенных здесь примеров видно, что и ассемблер, и софт-симулятор представляют собой программы обработки символьных строк, которые должны загрузить файл, разобрать текстовую строку на составляющие, проанализировать содержимое этих составляющих и из одних текстовых строк сформировать другие текстовые строки.

Заключение

Когда-то, в начале 90-х, автору довелось впервые увидеть многооконный симулятор микроконтроллера, работавший под первыми версиями Windows. Тогда, после симуляторов для DOS, он показался чудом. Окна можно было передвигать, группировать, изменять их размер. Теперь, когда появились такие мощные программы, как Visual C и BCB, разработка многооконных пользовательских интерфейсов стала нормой. При использовании библиотечных компонентов, входящих в состав BCB, разработка софт-симуляторов, аналогичных описанному в данной статье, значительно облегчается.

При разработке встроенных в FPGA контроллеров необходимо иметь комплект программных инструментов, в который входят:

- ассемблер;
- формирователь кода «прошивки»;
- софт-симулятор;
- программный инструмент, соединяющий описание выводов проекта на языке Verilog с проектом на PCB.

И несколько слов о контроллерах. Еще лет 5 назад было распространено мнение о том, что в FPGA нет смысла размещать АЛУ, аналогичные x186 или x386. Потом были выпущены встроенные процессоры Nios. Одновременно с ними появились открытые проекты множества АЛУ — от 8-разрядных до 32-разрядных. Среди них были копии распространенных ядер, таких как MCS51, AVR и x186. Через некоторое время были анонсированы PicoBlaze и MicroBlaze. Для всех этих микроконтроллеров большая часть программных инструментов уже существует. Но, тем не менее, встроенные контроллеры с системой команд, определяемой пользователем, тоже активно развиваются.

Поэтому автор надеется, что материал, изложенный в данной статье, будет полезен как при изучении встроенных контроллеров, так и при разработке проектов с их применением.

И последнее: для того, чтобы отладить проект со встроенным в FPGA контроллером, очень полезно бывает иметь еще и возможность проводить отладку встроенного программного обеспечения на реально работающей аппаратуре. И здесь необходимо иметь возможность подключить к проекту:

- встроенный логический анализатор,
- встроенное в FPGA отладочное средство для микроконтроллера.

Что касается первого, то тут можно применить логический анализатор, поставляемый в комплекте программ для работы с FPGA, или разработать «самодельный» [7]. Второе же средство — это тема отдельной статьи.

Литература

1. Семенов Н., Каршенбойм И. Микропрограммные автоматы на базе специализированных ИС // Chip News. 2000. № 7. http://chipnews.gaw.ru/
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. Между ISE и ViewDraw // Компоненты и технологии. 2005. № 6.
9. Ken Chapman. Creating Embedded Microcontrollers (Programmable State Machines). Xilinx. 2002. http://www.xilinx.com
10. 8-bit Microcontroller for Virtex Devices. Application Note 213. Xilinx.