CHAPTER 15

PICOBLAZE ASSEMBLY CODE DEVELOPMENT

15.1 INTRODUCTION

Because of its simplicity, PicoBlaze cannot effectively support high-level programming languages and the code is generally developed in assembly language. In this chapter, we provide an overview of code development, which is illustrated in a bottom-up fashion. We first introduce the segments of frequently used data and control operations and then examine the use of a subroutine and finally outline the derivation of overall program structure.

15.2 USEFUL CODE SEGMENTS

The PicoBlaze microcontroller contains instructions for byte-oriented data manipulation and simple conditional branch. In this section, we illustrate how to construct code to perform bit and multiple-byte operations and to realize frequently used high-level language control constructs.

15.2.1 KCPSM3 conventions

The KCPSM3 assembler uses the following conventions in an assembly program:

- Use a "`:`" sign after a symbolic address in code, as in "`done:`".
- Use a "`;`" sign before a comment.
- Use `HH` for a constant, in which `H` is a hexadecimal digit.

Copyright © 2008 John Wiley & Sons, Inc.

An example of a code segment follows:

```
    ;this is a demo segment
    test s0, 82        ;compare s0 with 1000_0010
    jump z, clr_s1     ;if MSB of s0 is 0, go to clr_s1
    load s1, FF        ;no, load 1111_1111 to s1
clr_s1:
    load s1, 01        ;load 0000_0001 to s1
```

15.2.2 Bit manipulation

PicoBlaze's instruction set is primarily for byte-oriented operations. Bit-oriented operations are frequently needed to control low-level I/O activities, such as testing, setting, and clearing a 1-bit flag signal.

To manipulate a single bit, we first define a *mask* to isolate and preserve (i.e., mask) the unrelated bits and then apply the designated operation on the desired bits (i.e., unmasked bits). We can set, clear, and toggle (i.e., invert) some bits of a data byte by performing **or**, **and**, and **xor** instructions with a proper mask. The following code segment shows how to set, clear, and toggle the second LSB of the s0 register:

```
    constant SET_MASK, 02    ;mask=0000_0010
    constant CLR_MASK, FD    ;mask=1111_1101
    constant TOG_MASK, 02    ;mask=0000_0010

    or  s0, SET_MASK     ;set 2nd LSB to 1
    and s0, CLR_MASK     ;clear 2nd LSB to 0
    xor s0, TOG_MASK     ;toggle 2nd LSB
```

The toggle operation is based on the observation that for any Boolean variable x, $x \oplus 0 = x$ and $x \oplus 1 = x'$. The same principle can be applied to multiple bits. For example, we can clear the upper nibble (i.e., four MSBs) by using

```
    and s0, 0F           ;mask=0000_1111
```

We can also apply the concept of the and mask to the **test** instruction to check a single bit. For example, the following code segment tests the MSB of the s0 register and branches to a proper routine accordingly:

```
    test s0, 80        ;mask=1000_0000
    jump nz, msb_set   ;MSB is 1, branch to msb_set
    ;code for MSB not set
    jump  done
msb_set:
    ;code for MSB set
    ...
done:
    ...
```

A single bit can be extracted by applying the previous code. For example, the following code segment extracts the MSB of the s0 register and stores it in the s1 register:

```
    load s1, 00
    test s0, 80        ;mask=1000_0000, extract MSB
    jump z, done       ;yes, MSB is 0
    load s1, 01        ;no, load 1 to s1
```

```
done:
   ...
```

15.2.3 Multiple-byte manipulation

A microcontroller sometimes needs to handle wide, multiple-byte data, such as a large counter. Since the data width of PicoBlaze is 8 bits, processing this type of data requires a mechanism to propagate information between two successive instructions. PicoBlaze uses the carry flag for this purpose. For the arithmetic instructions, there are two versions for addition and subtraction, one with carry and one without carry, as in the **add** and **addcy** instructions. For the shift and rotate instructions, carry can be shifted into the MSB or LSB of a register, and vice versa.

Assume that x and y are 24-bit data and each occupies three registers. The following code segment illustrates the use of carry in multiple-byte addition:

```
namereg s0, x0   ;least significant byte of x
namereg s1, x1   ;middle byte of x
namereg s2, x2   ;most significant byte of x
namereg s3, y0   ;least significant byte of y
namereg s4, y1   ;middle byte of y
namereg s5, y2   ;most significant byte of y

;add: {x2,x1,x0} + {y2,y1,y0}
add x0, y0        ;add least significant bytes
addcy x1, y1      ;add middle bytes with carry
addcy x2, y2      ;add most significant bytes with carry
```

The first instruction performs normal addition of the least significant bytes and stores the carry-out bit into the carry flag. The second instruction then includes the carry flag when adding the middle bytes. Similarly, the third instruction uses the carry flag from the previous addition to obtain the result for the most significant bytes.

The incrementing and subtraction of multiple bytes can be achieved in a similar fashion:

```
;increment: {x2,x1,x0} + 1
add x0, 01        ;inc least significant byte
addcy x1, 00      ;add carry to middle byte
addcy x2, 00      ;add carry to most significant byte

;subtract: {x2,x1,x0} - {y2,y1,y0}
sub x0, y0        ;sub least significant byte
subcy x1, y1      ;sub middle byte with borrow
subcy x2, y2      ;sub most significant byte with borrow
```

Multiple-byte data can be shifted by including the carry flag in the individual shift instruction. For example, the **sla** instruction shifts data left one position and shifts the carry flag into LSB. The code for shifting a 3-byte data left can be written as

```
;shift {x2,x1,x0} via carry
sl0 x0            ;0 to LSB of x0, MSB of x0 to carry
sla x1            ;carry to LSB of x1, MSB of x1 to carry
sla x2            ;carry to LSB of x2, MSB of x2 to carry
```

15.2.4 Control structure

A high-level programming language usually contains various control constructs to alter the execution sequence. These include the if-then-else, case, and for-loop statements. On the other hand, PicoBlaze provides only simple conditional and unconditional **jump** instructions. Despite its simplicity, we can use them with a **test** or **compare** instruction to implement the high-level control constructs. The following examples illustrate the construction of the if-then-else, case, and for-loop statements.

Let us first consider the if-then-else statement:

```
if (s0==s1) {
   /* then-branch statements */
}
else {
   /* else-branch statements */
}
```

The corresponding assembly code segment is

```
    compare s0, s1
    jump nz, else_branch
    ;code for then branch
    ...
    jump if_done
else_branch:
    ;code for else branch
    ...
if_done:
    ;code following if statement
    ...
```

The code uses the **compare** instruction to check the s0==s1 condition and to set the zero flag. The following **jump** instruction examines the flag and jumps to the else branch if the flag is not set.

The case statement can be considered as a multiway jump, in which the execution is transferred according to the value of the selection expression. The following statement uses the s0 variable as the selection expression and jumps to the corresponding branch:

```
switch (s0) {
   case value1:
      /* case value1 statements */
      break;
   case value2:
      /* case value2 statements */
      break;
   case value3:
      /* case value3 statements */
      break;
   default:
      /* default statements */
}
```

The multiway jump can be implemented by a hardware feature known as "index address mode" in some processors. However, since PicoBlaze does not support this feature, the case statement has to be constructed as a sequence of if-then-else statements. In other words, the previous case statement is treated as:

```
if (s0==value1) {
   /* case value1 statements */
}
else if (s0==value2) {
   /* case value2 statements */
}
else if (s0==value3) {
   /* case value3 statements */
}
else{
   /* default statements */
}
```

The corresponding assembly code segment becomes

```
   constant value1, ...
   constant value2, ...
   constant value3, ...

   compare s0, value1     ;test value1
   jump nz, case_2        ;not equal to value1, jump
   ;code for case 1
   ...
   jump case_done
case_2:
   compare s0, value2     ;test value2
   jump nz, case_3        ;not equal to value2, jump
   ;code for case 2
   ...
   jump case_done
case_3:
   compare s0, value3     ;test value3
   jump default           ;not equal to value3, jump
   ;code for case 3
   ...
   jump case_done
default:
   ;code for default case
   ...
case_done:
   ;code following case statement
   ...
```

The for-loop statement executes a segment of the code repetitively. The loop statement can be implemented by using a counter to keep track of the iteration number. For example, consider the following:

```
for(i=MAX, i=0, i-1) {
   /* loop body statements */
}
```

The assembly code segment is

```
namereg s0, i          ;loop index
constant MAX, ...      ;loop boundary
```

```
    load i, MAX             ;load loop index
loop_body:
    ;code for loop body
    ...
    sub i, 01               ;dec loop index?
    jump nz, loop_body      ;done?
    ;code following for loop
    ...
```

15.3 SUBROUTINE DEVELOPMENT

A subroutine, such as a function in C, implements a section of a larger program. It is coded to perform a specific task and can be used repetitively. Using subroutines allows us to divide a program into small, manageable parts and thus greatly improve the reliability and readability of a program. It is the base of modern programming practice and is supported by all high-level programming languages.

PicoBlaze uses the **call** and **return** instructions to implement the subroutine. The **call** instruction saves the current content of the program counter and transfers the program execution to the starting address of a subroutine. A subroutine ends with a **return** instruction, which restores the saved program counter and resumes the previous execution. A representative flow is shown in Figure 14.7. Note that PicoBlaze only saves and restores the content of the program counter during a function call and return. We have to manage the register and data RAM use manually to ensure that the original system state is not altered after a subroutine call.

The following multiplication example illustrates the development of subroutines. We assume that the inputs are two 8-bit numbers in unsigned integer format and the output is a 16-bit product. The algorithm is based on a simple shift-and-add method. This method iterates through 8 bits of multiplier. In each iteration, the multiplicand is shifted left one position. If the corresponding multiplier bit is '1', the shifted multiplicand is added to the partial product. The assembly code is shown in Listing 15.1. The multiplicand and multiplier are stored in the s3 and s4 registers. The individual bit of multiplier is obtained by repetitively shifting s4 to the right, which moves the LSB to the carry flag. Note that instead of actually shifting the multiplicand to the left, we shift the partial product, which consists of 2 bytes and is stored in s5 and s6, to the right.

Listing 15.1 Software integer multiplication

```
;==========================================================
;routine: mult_soft
;  function: 8-bit unsigned multiplier using
;            shift-and-add algorithm
;  input register:
;     s3: multiplicand
;     s4: multiplier
;  output register:
;     s5: upper byte of product
;     s6: lower byte of product
;  temp register: i
;==========================================================
mult_soft:
    load s5, 00             ;clear s5
```

```
15    load i, 08                      ;initialize loop index
   mult_loop:
      sr0 s4                          ;shift LSB to carry
      jump nc, shift_prod             ;LSB is 0
      add s5, s3                      ;LSB is 1
20 shift_prod:
      sra s5                          ;shift upper byte right,
                                      ;carry to MSB, LSB to carry
      sra s6                          ;shift lower byte right,
                                      ;LSB of s5 to MSB of s6
25    sub i, 01                       ;dec loop index
      jump nz, mult_loop              ;repeat until i=0
      return
```

Because of the primitive nature of the assembly language, thorough documentation is instrumental. A subroutine should include a descriptive header and detailed comments. A representative header is shown in Listing 15.1. It consists of a short function description and the use of registers. The latter shows how the registers are allocated and is crucial to preventing conflict in a large program.

15.4 PROGRAM DEVELOPMENT

Developing a complete assembly program consists of the following steps:

1. Derive the pseudo code of the *main program.*
2. Identify tasks in the main program and define them as subroutines. If needed, continue refining the complex subroutines and divide them into smaller routines.
3. Determine the register and data RAM use.
4. Derive assembly code for the subroutines.

Steps 1, 2, and 4 basically follow a *divide-and-conquer* approach and are applicable for any software development. A microcontroller-based application is normally for a simple embedded system, in which the processor monitors the I/O activities continuously and responds accordingly. Its main program usually has the following structure:

```
    call initilaization_routine
forever:
    call task1_routine
    call task2_routine
    ...
    call taskn_routine
    jump forever
```

Step 3 is unique for assembly code development. Unlike a high-level language program, in which the compiler automatically allocates storage to variables, we must manually manage the data storage in assembly code. PicoBlaze has 16 registers and 64 bytes of data RAM to store data. The registers can be considered as fast storage, in which the data can be manipulated directly. The data RAM, on the other hand, is "auxiliary" storage. Its data needs to be transferred to a register for processing. For example, if we want to increment a data item located in the RAM, it must first be loaded into a register, incremented there, and then stored back to the RAM.

Because of the limited space for data storage, its use has to be planned carefully in advance, particularly when the code is complex and involves nested subroutines. To assist

00	lower byte of a
01	unused
02	lower byte of b
03	unused
04	lower byte of a^2
05	upper byte of a^2
06	lower byte of b^2
07	upper byte of b^2
08	lower byte of $a^2 + b^2$
09	upper byte of $a^2 + b^2$
0A	carry of $a^2 + b^2$

Figure 15.1 Data RAM memory allocation.

coding, we can first identify the needed *global storage* or *local storage*. The former keeps data that is needed in the entire program. The latter provides space to store intermediate results, and the data will be discarded after the required computation is completed.

15.4.1 Demonstration example

The development process can best be explained by an example. Let us consider a program that uses the previous multiplication subroutine. It reads two inputs, a and b, from the switch, calculates $a^2 + b^2$, and displays the result on eight discrete LEDs. Since the I/O interface is to be discussed in Chapter 16, we limit the I/O to a single input port, the 8-bit switch, and a single output port, the 8-bit LEDs. We assume that a and b are obtained from the upper nibble (i.e., the four MSBs) and the lower nibble (i.e., the four LSBs) of the switch. The main program is

```
    call clear_data_ram
forever:
    call read_switch
    call square
    call write_led
    jump forever
```

The subroutines are defined as follows:

- `clr_data_mem`: clears data memory at system initialization
- `read_switch`: obtains the two nibbles from the switch and stores their values to the data RAM
- `square`: uses the multiplication subroutine to calculate $a^2 + b^2$
- `write_led`: writes the eight LSBs of the calculated result to the LED port

For demonstration purposes, we create two smaller routines, `get_upper_nibble` and `get_lower_nibble`, within the `read_switch` routine to obtain the upper nibble and lower nibble from a register.

The next step in development is to plan the register and data RAM use. For global storage, we introduce a global register, `sw_in`, to store the input value of switch and allocate 11 bytes of data RAM to store the inputs and result of the `square` routine. Allocation of the data RAM is shown in Figure 15.1. Note that the addresses 01 and 03 are not actually used. They are reserved to simplify the seven-segment LED display code, which is discussed in Chapter 16. All remaining registers are used as local storage. For program clarity, we

define three symbolic names, data, addr, and i, as temporary registers for data, port and memory address, and loop index.

The last step is to derive the assembly code for the subroutines. The complete code is shown in Listing 15.2. The clr_data_mem uses a loop to clear data memory. The i register is the loop index and initialized with 64 (i.e., 40_{16}). The index is decremented in each loop and 0 is loaded to the corresponding data RAM address. The write_led routine fetches the eight LSBs of the calculated result from the data RAM and outputs them to the LED port.

The read_switch routine includes two smaller routines. The get_upper_nibble routine shifts the data register right four times to move the upper nibble to the four LSBs. The get_lowe_nibble routine clears the four MSBs of the data register to 0's and thus removes the upper nibble. The "glue instructions" of read_switch input the switch values, set up the input for the two nibble routines, and store the result in the data RAM.

The square routine fetches data from the data RAM, utilizes the mult_soft routine to calculate a^2 and b^2, performs addition, and stores the result back to the data RAM.

Listing 15.2 Square program with simple nibble input

```
;==========================================================
; square circuit with simple I/O interface
;==========================================================
;program operation:
;   - read switch to a (4 MSBs) and b (4 LSBs)
;   - calculate a*a + b*b
;   - display data on 8 leds

;==========================================================
; data constant
;==========================================================
constant UP_NIBBLE_MASK, 0F  ;00001111

;==========================================================
; data ram address alias
;==========================================================
constant a_lsb, 00
constant b_lsb, 02
constant aa_lsb, 04
constant aa_msb, 05
constant bb_lsb, 06
constant bb_msb, 07
constant aabb_lsb, 08
constant aabb_msb, 09
constant aabb_cout, 0A

;==========================================================
; register alias
;==========================================================
;commonly used local variables
namereg s0, data  ;reg for temporary data
namereg s1, addr  ;reg for temporary mem & i/o port addr
namereg s2, i     ;general-purpose loop index
;global variables
namereg sf, sw_in
```

```
  ;=====================================================
  ; port alias
  ;=====================================================
40 ;-----------input port definitions------------------
  constant sw_port, 01 ;8-bit switches
  ;-----------output port definitions-----------------
  constant led_port, 05

45 ;=====================================================
  ; main program
  ;=====================================================
  ;calling hierarchy:
  ;
50 ;main
  ;   - clr_data_mem
  ;   - read_switch
  ;       - get_upper_nibble
  ;       - get_lower_nibble
55 ;   - square
  ;       - mult_soft
  ;   - write_led
  ;

60    call clr_data_mem
  forever:
     call read_switch
     call square
     call write_led
65    jump forever

  ;=====================================================
  ;routine: clr_data_mem
  ;  function: clear data ram
70 ;  temp register: data, i
  ;=====================================================
  clr_data_mem:
     load i, 40              ;unitize loop index to 64
     load data, 00
75 clr_mem_loop:
     store data, (i)
     sub i, 01               ;dec loop index
     jump nz, clr_mem_loop   ;repeat until i=0
     return
80
  ;=====================================================
  ;routine: read switch
  ;  function: obtain two nibbles from input
  ;  input register: sw_in
85 ;  temp register: data
  ;=====================================================
  read_switch:
     input sw_in, sw_port    ;read switch input
```

```
   load data, sw_in
90 call get_lower_nibble
   store data, a_lsb          ;store a to data ram
   load data, sw_in
   call get_upper_nibble
   store data, b_lsb          ;store b to data ram
95
;=========================================================
;routine: get_lower_nibble
;  function: get lower 4 bits of data
;  input register: data
100 ;  output register: data
;=========================================================
get_lower_nibble:
   and data, UP_NIBBLE_MASK ;clear upper nibble
   return
105
;=========================================================
;routine: get_upper_nible
;  function: get upper 4 bits of data
;  input register: data
110 ;  output register: data
;=========================================================
get_upper_nibble:
   sr0 data                   ;right shift 4 times
   sr0 data
115 sr0 data
   sr0 data
   return

;=========================================================
120 ;routine: write_led
;  function: output 8 LSBs of result to 8 leds
;  temp register: data
;=========================================================
write_led:
125 fetch data, aabb_lsb
   output data, led_port
   return

;=========================================================
130 ;routine: square
;  function: calculate a*a + b*b
;     data/result stored in ram started w/ SQ_BASE_ADDR
;  temp register: s3, s4, s5, s6, data
;=========================================================
135 square:
   ;calculate a*a
   fetch s3, a_lsb            ;load a
   fetch s4, a_lsb            ;load a
   call mult_soft             ;calculate a*a
140 store s6, aa_lsb           ;store lower byte of a*a
   store s5, aa_msb           ;store upper byte of a*a
```

```
    ;calculate b*b
    fetch s3, b_lsb           ;load b
    fetch s4, b_lsb           ;load b
145 call mult_soft            ;calculate b*b
    store s6, bb_lsb          ;store lower byte of b*b
    store s5, 07              ;store upper byte of b*b
    ;calculate a*a+b*b
    fetch data, aa_lsb        ;get lower byte of a*a
150 add data, s6              ;add lower byte of a*a+b*b
    store data, aabb_lsb      ;store lower byte of a*a+b*b
    fetch data, aa_msb        ;get upper byte of a*a
    addcy data, s5            ;add upper byte of a*a+b*b
    store data, aabb_msb      ;store upper byte of a*a+b*b
155 load data, 00             ;clear data, but keep carry
    addcy data, 00            ;get carry-out from previous +
    store data, aabb_cout     ;store carry-out of a*a+b*b
    return

160 ;==========================================================
 ;routine: mult_soft
 ;  function: 8-bit unsigned multiplier using
 ;            shift-and-add algorithm
 ;  input register:
165 ;    s3: multiplicand
 ;    s4: multiplier
 ;  output register:
 ;    s5: upper byte of product
 ;    s6: lower byte of product
170 ;  temp register: i
 ;==========================================================
 mult_soft:
    load s5, 00               ;clear s5
    load i, 08                ;initialize loop index
175 mult_loop:
    sr0 s4                    ;shift lsb to carry
    jump nc, shift_prod       ;lsb is 0
    add s5, s3                ;lsb is 1
 shift_prod:
180 sra s5                    ;shift upper byte right,
                              ;carry to MSB, LSB to carry
    sra s6                    ;shift lower byte right,
                              ;lsb of s5 to MSB of s6
    sub i, 01                 ;dec loop index
185 jump nz, mult_loop        ;repeat until i=0
    return
```

15.4.2 Program documentation

Developing an assembly program is a tedious process. The use of symbolic names and good documentation can make the code clear and reduce many unnecessary errors. It also helps future revision and maintenance. For the KCPSM3 assembler, we can use the **constant**

directive to assign a symbolic name (alias) to a data constant, a memory address, or a port id, and use the **namereg** directive to assign a symbolic name to a register.

A representative main program header is shown in Listing 15.2. It contains the following segments:

- *General program description*: provides a general description for the purpose, operation, and I/O of the program
- *Data constants*: declares symbolic names for constants
- *Data RAM address alias*: declares symbolic names for data RAM addresses
- *Register alias*: declares symbolic names for registers
- *Port alias*: declares symbolic names for I/O ports
- *Program calling hierarchy*: illustrates the calling structure and subroutines

The aliases and directives have no effect on the final machine code. When the assembly code is processed, they are replaced with the actual constant values. However, using aliases can greatly enhance the readability of the assembly code and reduce unnecessary errors. The following code segment further illustrates the impact of the alias and documentation. The purpose of this segment is to obtain values for variables a, b, and c, and store them in proper data RAM locations. The location is specified by the UART input, which is the ASCII code of character a, b, or c. The segment with aliases and proper comments is

```
;constant alias
   constant ASCII_a, 61            ;ASCII code for a
   constant ASCII_b, 62            ;ASCII code for b
   constant ASCII_c, 63            ;ASCII code for c
;data ram address alias
   constant a_addr, 02
   constant b_addr, 04
   constant c_addr, 06
;register alias
   namereg s0, data                ;reg for temporary data
   namereg s1, addr                ;reg for temporary addr
   namereg sF, sw_in               ;switch input
;port alias
   constant sw_port, 01            ;switch input
   constant uart_rx_port, 02       ;UART input

;assembly code with alias
   ;get input
   input sw_in, sw_port            ;get switch
   input data, uart_rx_port        ;get char
   ;check received char
   compare data, ASCII_a           ;check ASCII a
   jump nz, chk_ascii_b            ;no, check next
   store  sw_in, a_addr            ;yes, store a to data ram
   jump done
chk_ascii_b:
   compare data, ASCII_b           ;check ASCII b
   jump nz, chk_ascii_c            ;no, check next
   store sw_in, b_addr             ;yes, store b to data ram
   jump done
chk_ascii_c:
   compare data, ASCII_c           ;check ASCII c
   jump nz, ascii_err              ;no, error
```

```
      store sw_in, c_addr            ;yes, store b to data ram
      jump done
   ascii_err:
      ...
   done:
      ...
```

If we use hard literals and strip the comments, the code becomes

```
   ;assembly code with no alias or comments
      input sf, 01
      input s0, 02
      compare s0, 61
      jump nz, addr1
      store sf, 02
      jump addr4
   addr1:
      compare s0, 62
      jump nz, addr2
      store sf, 04
      jump addr4
   addr2:
      compare s0, 63
      jump nz, addr3
      store sf, 06
      jump addr4
   addr3:
      ...
   addr4:
      ...
```

While the functionality of this code segment is the same, it is very difficult to comprehend, debug, or modify.

15.5 PROCESSING OF THE ASSEMBLY CODE

PicoBlaze-based development flow is reviewed in Section 14.4. After the assembly code is developed, it is then compiled (translated) to machine instruction in step 3. The instruction-set-level simulation can also be performed to verify the correctness of the code, as in step 4. The two steps and the direct downloading process (step 9) are discussed in detail in this section.

Xilinx provides an assembler known as *KCPSM3* for compiling in step 3 and downloading utility programs in step 9. The programs, HDL codes for the PicoBlaze processor, and relevant template files can be downloaded from the Xilinx's web site. A program known as *PBlazeIDE* from Mediatronix can perform the instruction-set-level simulation in step 4. It can also be used as an assembler. PBlazeIDE can be downloaded from Mediatronix's Web site.

15.5.1 Compiling with KCSPM3

Assembler is the software that translates the instruction mnemonics to machine instructions, which are represented as 0's and 1's, and substitutes the aliases and symbolic branch addresses with actual values. The machine instructions are then downloaded to the instruction

memory of a microcontroller. Since PicoBlaze is embedded inside FPGA, the instruction ROM becomes an HDL ROM module with the compiled assembly code. The ROM will be instantiated later in the top-level HDL code and synthesized along with PicoBlaze and the I/O interface circuit.

Xilinx provides the *KCPSM3* assembler for this task. It is a command-line, DOS-based program. KCPSM3 basically takes an assembly program, along with the necessary template files, and generates the HDL code for the instruction ROM. The procedure of compiling an assembly program is as follows:

1. Create a directory for the project and copy `kcpsm3.exe`, `ROM_form.vhd`, `ROM_form.v`, and `ROM_form.coe` to the directory. The latter three are code templates used by KCPSM3.
2. Create the assembly program and save it as plain text file with an extension of `.psm`. Any PC-based editor, such as Notepad, can be used for this purpose.
3. Invoke a DOS window by selecting Start ≻ Programs ≻ Accessories ≻ Command Prompt. In the DOS window, navigate to the project directory.
4. Type `kcpsm3 myfile.psm` to run the program.
5. Correct syntax errors if necessary and recompile.
6. After successful compiling, the file containing the instruction ROM, `myfile.vhd`, is generated.

In addition to the HDL file, KCPSM3 also generates files that are suitable for block RAM initialization and other utilities. The file with the `.hex` extension can be used for JTAG downloading, which is discussed in Section 15.5.3, and the file with the `.fmt` extension is a reformatted `.psm` file for "pretty printing."

15.5.2 Simulation by PBlazeIDE

As the name indicates, instruction-set-level simulation simulates the operation of a PicoBlaze system instruction by instruction. The *PBlazeIDE* program can be used for this purpose. PBlazeIDE is a Windows-based program with an integrated development environment, which includes a text editor, an assembler, and an instruction-set-level simulator.

PBlazeIDE uses slightly different instruction mnemonics and directives, as discussed in Section 14.5. Thus, the code written for by KCPSM3 cannot be used directly by PBlazeIDE, and vice versa. The mnemonic differences are summarized in Table 15.1, and the directive examples are shown in Table 15.2. Note that the PBlazeIDE assembler uses both decimal and hexadecimal format for constants. A hexadecimal number is started with a `$` sign, as in `$1A`.

The procedure of using PBlazeIDE for KCPSM3 code is as follows:

1. Compile the assembly code with KCPSM3.
2. Launch PBlazeIDE.
3. Select Settings ≻ PicoBlaze 3. This specifies the version 3 of PicoBlaze, which is used in the Spartan-3 device.
4. Select File ≻ Import and a dialog window appears. Select the corresponding `.fmt` file. The "import" function converts the KCPSM3 code to the PBlazeIDE code. The formatted program is easier for conversion. The converted file may sometimes need minor manual editing.
5. Manually specify the **dsin**, **dsout**, and **dsio** directives for I/O ports. When one of these directives is used, a port indicator will be added to the simulation screen to show the activities of the port.

Table 15.1 Mnemonic differences between KCPSM3 and PBlazeIDE

KCPSM3	PBlazeIDE
addcy	**addc**
subcy	**subc**
compare	**comp**
store sX, (sY)	**store** sX, sY
fetch sX, (sY)	**fetch** sX, sY
input sX, (sY)	**in** sX, sY
input sX, KK	**in** sX, $KK
output sX, (sY)	**out** sX, sY
output sX, KK	**out** sX, $KK
return	**ret**
returni	**reti**
enable interrupt	**eint**
disable interrupt	**dint**

Table 15.2 Directive examples of KCPSM3 and PBlazeIDE

Function	KCPSM3	PBlazeIDE
code location	**address** 3FF	**org** $3FF
constant	**constant** MAX, 3F	MAX **equ** $3F
register alias	**namereg** addr, s2	addr **equ** s2
port alias	**constant** in_port, 00	in_port **dsin** $00
	constant out_port, 10	out_port **dsout** $10
	constant bi_port, 0F	bi_port **dsio** $0F

6. Enter the simulation mode by selecting Simulate ≻ Simulate. Perform simulation.
7. If the assembly code needs to be revised, it must be done outside PBlazeIDE. Simply close the current file, invoke an external editor to edit the original .psm file, save the file, and restart from step 1. If the file is edited within PBlazeIDE, it cannot be converted back to KCPSM3 code.

A representative simulation screenshot is shown in Figure 15.2. The simulator displays the assembly code in the central window and highlights the next instruction to be executed. The instruction address, instruction code, and breakpoints are shown next to the code. The current state of PicoBlaze is shown at the left, which includes the status of the flags, the content of the registers, and the content of the data RAM. The values of the program counter and stack pointer as well as some execution statistics are shown in the bottom row.

The emulated I/O ports created by the **dsin**, **dsout**, and **dsio** directives are shown at the right. There are an input port, switch, and an output port, led, on this particular screen. Since PBlazeIDE has no information about I/O behavior, the input port data must be entered and modified manually during simulation.

During simulation, the assembly program can be executed continuously, by one step, by one instruction, or to pause at a specific breakpoint. The simulation action is controlled by the commands of the Simulate menu or the icons on the top:

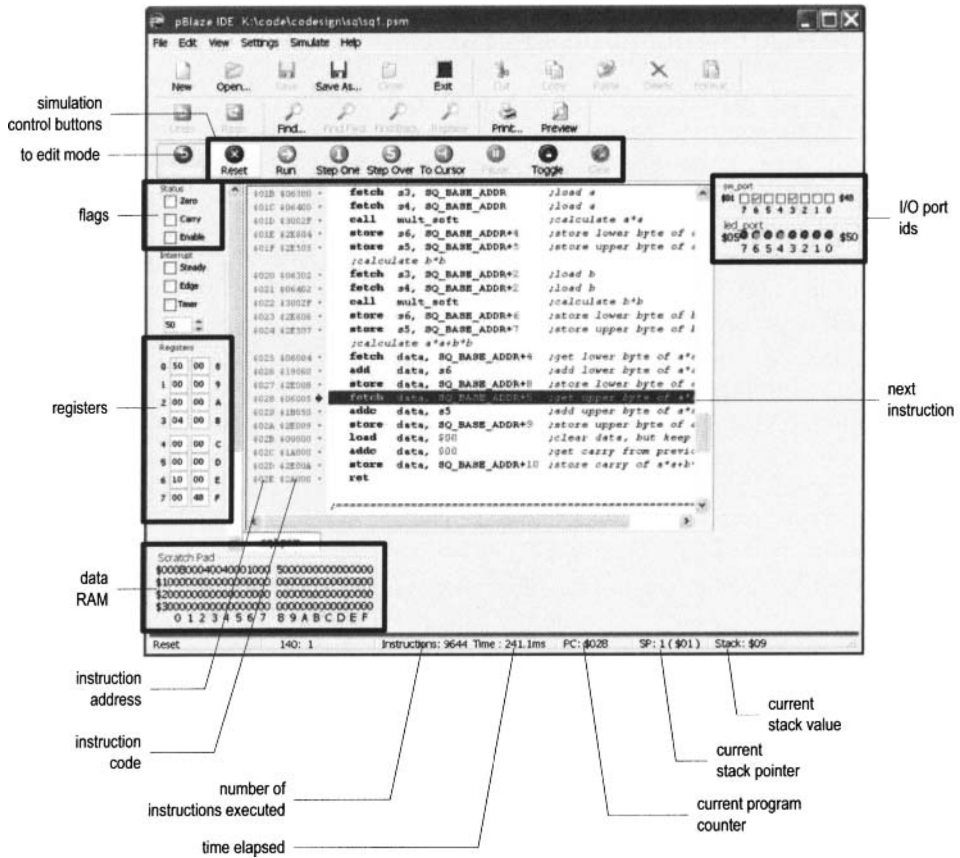

Figure 15.2 Screenshot of pBlazeIDE in simulation mode.

- Reset: clears the program counter and stack pointer
- Run: runs the program continuously until a breakpoint
- Single step: executes one instruction
- Step over: executes the entire subroutine for a **call** instruction and executes one instruction for other instructions
- Run to cursor: runs the program to the current cursor position
- Pause: pauses the simulation
- Toggle breakpoint: sets or clears a breakpoint at the current cursor position
- Remove all breakpoints: clears all breakpoints

15.5.3 Reloading code via the JTAG port

After the instruction ROM HDL is generated, we can continue steps 6 and 8 in Figure 14.4 to synthesize the entire code and download the configuration file to the FPGA chips. Note that the synthesis flow must be repeated each time the assembly code is modified.

Since synthesis is a complex process, it requires a significant amount of computation time. When the I/O configuration is fixed, resynthesizing the entire circuit after each assembly program modification is not really needed. It is possible to reload the machine code to the ROM, which is implemented by a block RAM, by using the FPGA's JTAG interface. This corresponds to the dotted line of step 9 in Figure 14.4. The basic procedure is as follows:

1. Replace the original ROM template with one that contains the JTAG interface circuit.
2. Use KCPSM3 to compile the assembly code as usual.
3. Synthesize the top-level HDL code and program the FPGA chip.
4. In subsequent assembly program modifications, compile the program as usual. Recall that a file in `hex` format (ended with the `.hex` extension) is generated.
5. Use the Xilinx utility to embed the `.hex` file to a JTAG programming file and download the file to the FPGA's block RAM via the JTAG interface.

The detailed procedure and the relevant programs and templates can be found in the `JTAG_loader` directory of the downloaded `KCPSM` file.

15.5.4 Compiling by PBlazeIDE

As discussed earlier, PBlazeIDE is an integrated program that contains an assembler and editor. If the program is developed with PBlazeIDE mnemonics, PBlazeIDE can replace the KCPSM3 assembler. The instruction ROM VHDL file is generated by a directive. If the HDL file is needed, simply include the **vhdl** directive in the assembly code. Its syntax is

```
vhdl "ROM_form.vhd", "rom_target.vhd", "rom_entity_name"
```

The `"ROM_form.vhd"` term specifies a VHDL template file, which is the same file as that discussed in Section 15.5.1. It should be copied to the directory where the assembly program file resides. The `"rom_target.vhd"` term specifies the name of the generated ROM VHDL file, and the `"rom_entity_name"` term indicates the desired entity name of the previously generated VHDL file. The VHDL file is generated automatically when PBlazeIDE is switched from the edit mode to the simulation mode.

Note that since PBlazeIDE does not generate a `hex` file, the reloading scheme discussed in Section 15.5.3 cannot be applied directly.

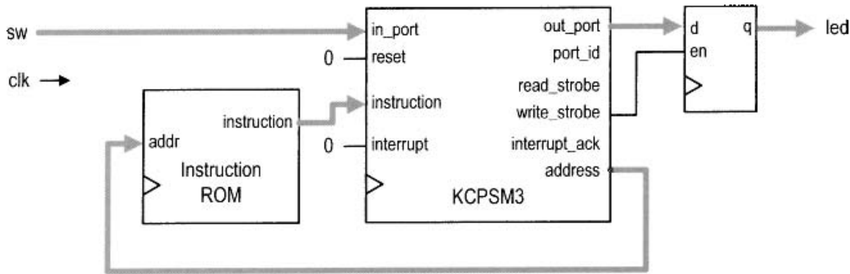

Figure 15.3 PicoBlaze with a simple I/O interface.

15.6 SYNTHESES WITH PICOBLAZE

After generating the HDL file for the instruction ROM, we can combine it with PicoBlaze to synthesize the entire system in an FPGA chip. Unlike a normal microcontroller, PicoBlaze has no built-in I/O peripherals. The I/O interface is created and customized as needed. The circuit is described in HDL code. Since the focus in this chapter is assembly program development, we use a simple I/O configuration, which contains only one switch input port and one led output port, for synthesis. The development of more sophisticated I/O interface is discussed in detail in Chapters 16 and 17.

The top-level block diagram of this design is shown in Figure 15.3. It contains the PicoBlaze processor, which is labeled `kcpsm3`, the instruction ROM, and a register. The register functions as a buffer for the eight LEDs. When PicoBlaze executes the **output** instruction, it places the data on `out_port` and asserts the `write_strobe` signal, which enables the register and stores the data in the register. The `sw` signal is connected to `in_port`. When PicoBlaze executes the **input** instruction, it retrieves the value of the `sw` signal and stores it in an internal register. The corresponding HDL code is shown in Listing 15.3. It consists of instantiations of the PicoBlaze processor and instruction ROM, and a segment for the output buffer. The `kcpsm3` entity is the name of the PicoBlaze processor, and its code is stored in an HDL file of the same name. The `sio_rom` entity is from the previously generated instruction ROM file.

Listing 15.3 PicoBlaze with a simple I/O configuration

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pico_sio is
5    port(
        clk, reset: in std_logic;
        sw: in std_logic_vector(7 downto 0);
        led: out std_logic_vector(7 downto 0)
     );
10 end pico_sio;

architecture arch of pico_sio is
   -- KCPSM3/ROM signals
   signal address: std_logic_vector(9 downto 0);
15 signal instruction: std_logic_vector(17 downto 0);
   signal port_id: std_logic_vector(7 downto 0);
```

```
   signal in_port, out_port: std_logic_vector(7 downto 0);
   signal write_strobe: std_logic;
   -- register signals
20 signal led_reg: std_logic_vector(7 downto 0);
begin
   -- =======================================================
   --  KCPSM and ROM instantiation
25 -- =======================================================
   proc_unit: entity work.kcpsm3
      port map(
         clk=>clk, reset=>reset,
         address=>address, instruction=>instruction,
30       port_id=>open, write_strobe=>write_strobe,
         out_port=>out_port, read_strobe=>open,
         in_port=>in_port, interrupt=>'0',
         interrupt_ack=>open);
   rom_unit: entity work.sio_rom
35    port map(
          clk => clk, address=>address,
          instruction=>instruction);
   -- =======================================================
   --  output interface
40 -- =======================================================
   --output register
   process (clk)
   begin
      if (clk'event and clk='1') then
45       if write_strobe='1' then
            led_reg <= out_port;
         end if;
      end if;
   end process;
50 led <= led_reg;
   -- =======================================================
   --  input interface
   -- =======================================================
   in_port <= sw;
55 end arch;
```

15.7 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 14. The procedure of reloading compiled code via JTAG port is explained in the article, "PicoBlaze JTAG Loader Quick User Guide," by Kris Chaplin and Ken Chapman, which appears in the JTAG_loader directory of the downloaded KCPSM file.

15.8 SUGGESTED EXPERIMENTS

15.8.1 Signed multiplication

The subroutine in Listing 15.1 assumes that the inputs are in unsigned integer format. Modify the subroutine to perform the signed multiplication, in which the two inputs and output are interpreted as signed integers, and use simulation to verify its operation.

15.8.2 Multi-byte multiplication

The subroutine in Listing 15.1 assumes that the inputs are 8 bits wide. Some application may need more precision and we want to extend the subroutine to take 16-bit unsigned inputs. An operand now requires two registers and the result needs four registers. Develop the subroutine and use simulation to verify its operation.

15.8.3 Barrel shift function

PicoBlaze can only shift or rotate a single bit. A "barrel" shifting function can perform the shift and rotate operation for multiple bits. This function has three input registers. The first register contains data to be shifted or rotated; the second register specifies the amount, which is between 0 and 7; and the third register indicates the types of operation, which can be shift left, shift right, rotate left, or rotate right. We assume that 0 will be shifted in for the two shift operations. Develop the subroutine and use simulation to verify its operation.

15.8.4 Reverse function

A reverse function reverses the bit order of an input. For example, if the input is "01010011", the output becomes "11001010". We can use the 8-bit switch as input and the 8-bit discrete LEDs as output. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

15.8.5 Binary-to-BCD conversion

Binary-to-BCD conversion is discussed in Section 6.3.3. This function can be implemented by using assembly code as well. Assume that the input is an 8-bit binary number and the output is a two-digit 8-bit BCD number. If the input exceeds 99, the output generates a special overflow pattern, "11111111". We can use the 8-bit switch as input and the 8-bit discrete LEDs as output. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

15.8.6 BCD-to-binary conversion

Repeat Experiment 15.8.5, but develop the assembly code and circuit for BCD-to-binary conversion.

15.8.7 Heartbeat circuit

A "heartbeat circuit" is discussed in Experiment 4.7.4. We can create a similar pattern using the eight discrete LEDs as well. Derive and simulate the assembly code, obtain the

instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

15.8.8 Rotating LED circuit

We want to design a circuit that rotates a simple LED pattern to the left or right at four different speeds. The four patterns are "00000001", "00000011", "00001111", and "00001101". The pattern, direction, and rotation speed can be selected from the 8-bit switch (only 5 bits are used). The speed should be properly chosen so that all four patterns are visually observable. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

15.8.9 Discrete LED dimmer

The concept of PWM and LED dimmer are discussed in Experiment 4.7.2. In this experiment, we want to use eight discrete LEDS to show the various degrees of the brightness. This can be done by changing the "on" fraction of an LED. The "on" fraction of the eight LEDS will be $\frac{8}{8}, \frac{7}{8}, \frac{6}{8}, \ldots, \frac{1}{8}$. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.