CHAPTER 16

PICOBLAZE I/O INTERFACE

16.1 INTRODUCTION

To interact with the external environment, a regular microcontroller chip consists of a variety of built-in I/O peripherals, such as a UART, SPI (serial peripheral interface), timer, etc. When starting a new development, we select a microcontroller chip according to the I/O requirements of the application and may sometimes need to use additional chips to realize less commonly used functions.

Unlike a regular microcontroller, PicoBlaze has no built-in I/O peripherals. It just provides a simple generic input and output structure for an I/O interface. I/O peripherals are constructed as needed and thus are customized to each application. PicoBlaze uses the **input** and **output** instructions to transfer data between its internal registers and I/O ports, and its interface consists of the following signals:

- `port_id`: an 8-bit signal that specifies the port id (i.e., port address) of an **input** or **output** instruction
- `in_port`: an 8-bit signal where PicoBlaze obtains input data during operation of an **input** instruction
- `out_port`: an 8-bit signal where PicoBlaze places output data during operation of an **output** instruction
- `read_strobe`: a 1-bit signal that is asserted in the second clock cycle of an **input** instruction
- `write_strobe`: a 1-bit signal that is asserted in the second clock cycle of an **output** instruction

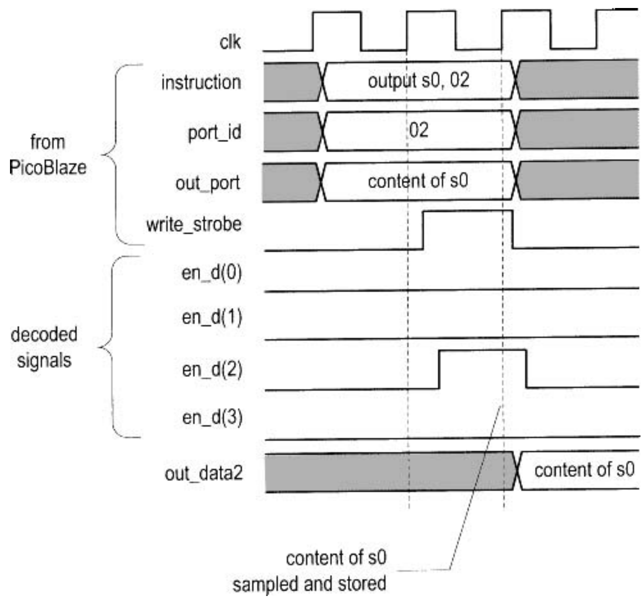

Figure 16.1 Timing diagram of an **output** instruction.

Although there are only two 8-bit ports to input and output data, the 8-bit `port_id` signal can be used to distinguish different peripherals, and thus it is said that PicoBlaze can support up to 256 (i.e., 2^8) input ports and 256 output ports.

In the remaining chapter, we examine the detailed I/O timing of PicoBlaze and illustrate the I/O interface development by adding a series of peripherals for the square circuit of Chapter 15.

16.2 OUTPUT PORT

16.2.1 Output instruction and timing

The **output** instruction writes data to the output port. It has two forms:

```
output sX, (sY)
output sX, port_name
```

In the first form, the port id is stored in the `sY` register. In the second form, the port id is specified explicitly by `port_name`, which is a two-digit hexadecimal number or a previously defined symbolic constant. The output data is always stored in the `sX` register.

The timing diagram of an **output** instruction,

```
output s0, 02
```

is shown in the top five traces of Figure 16.1. Recall that each PicoBlaze instruction takes two clock cycles. When the instruction is executed, the content of `s0` is placed on `out_port` and 02 is placed on `port_id` for two clock cycles. The `write_strobe` signal is asserted in the second clock cycle. It can be used as an enable tick to store data in an output register or to initiate the designated peripheral operation.

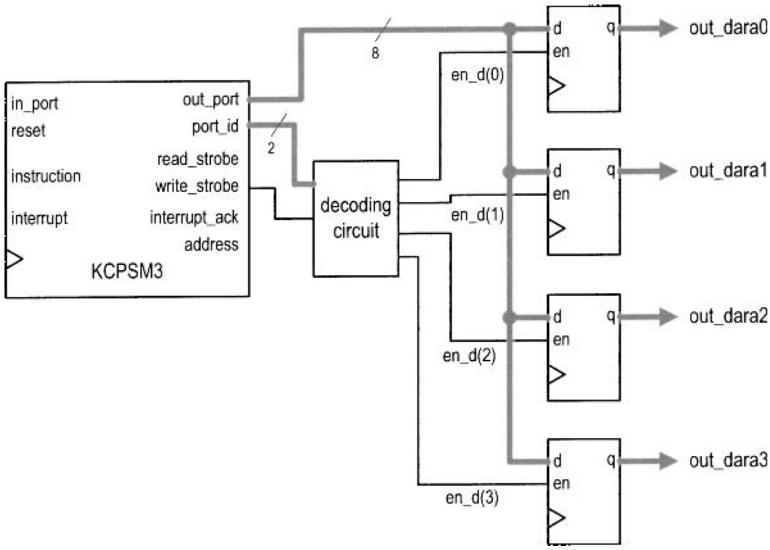

Figure 16.2 Output decoding of four output registers.

Table 16.1 Truth table of a decoding circuit

input			output
`write_strobe`	`port_id(1)`	`port_id(0)`	`en_d`
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

16.2.2 Output interface

The output interface between PicoBlaze and an output peripheral usually consists of a decoding circuit and necessary output buffers, which are normally an array of registers. The decoding circuit decodes the port id and generates an enable tick accordingly. After the **output** instruction, the data will be stored in the designated buffer.

To illustrate the construction, let us consider a PicoBlaze interface with four output buffers. We assign 00_{16}, 01_{16}, 02_{16}, and 03_{16} as their port ids. Note that the six MSBs of the port addresses are identical and only two LSBs are needed to distinguish a port. The block diagram is shown in Figure 16.2. The key is the decoding circuit, whose function table is shown in Table 16.1. It is a 2-to-2^2 decoder. In the second clock cycle of an **output** instruction, `write_strobe` is asserted and 1 bit of the 4-bit `en_d` signal is asserted accordingly. The one-clock-cycle enable tick activates the corresponding output register to retrieve data from the `out_port` signal. The decoding timing diagram of the instruction

```
output s0, 02
```

is shown at the bottom of Figure 16.1. During the second clock cycle of the **output** instruction, the `en_d(2)` signal is asserted and the data value on `out_port` is stored in the corresponding buffer at the rising edge of the next clock.

Once understanding the basic operation, we can derive the HDL code accordingly. The code segment is

```
process(write_strobe,port_id)
begin
   if write_strobe='0' then
      en_d <= "0000";
   else
      case port_id(1 downto 0) is
         when "00" =>
            en_d <= "0001";
         when "01" =>
            en_d <= "0010";
         when "10" =>
            en_d <= "0100";
         when others =>
            en_d <= "1000";
      end case;
   end if;
end process;
```

This scheme is very general and can be applied to any number of output ports.

The choice of the port address is somewhat arbitrary. We use the binary code in the previous example. If the number of the output port is smaller than eight, one-hot code can be used to simplify the decoding circuit. For example, we can define the four previous port ids as 01_{16} (i.e., 00000001_2), 02_{16} (i.e., 00000010_2), 04_{16} (i.e., 00000100_2), and 08_{16} (i.e., 00001000_2). The decoding logic can be simplified to

```
process(write_strobe,port_id)
begin
   if write_strobe='0' then
      en_d <= "0000";
   else
      en_d <= port_id(3 downto 0);
   end if;
end process;
```

Note that no decoding logic is needed if there is only a single output port. The `write_strobe` signal can be connected to the register's enable signal, as shown in Figure 15.3.

As discussed in Section 15.4.2, it is good practice to use symbolic aliases for I/O ports and declare its binary address in the header. For example, the initial output port address assignment can be declared as

```
;--------------output port definitions-------------
constant out_port_a, 00
constant out_port_b, 01
constant out_port_c, 02
constant out_port_d, 04
```

If the assignment is changed, we need to modify the header but keep the remaining assembly code intact. Using a clear header also allows us easily to identify the port ids when the companion HDL code is developed.

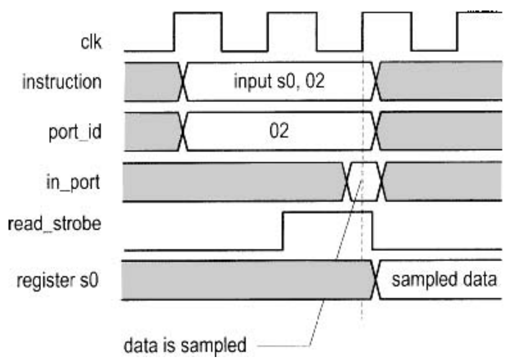

Figure 16.3 Timing diagram of an **input** instruction.

16.3 INPUT PORT

16.3.1 Input instruction and timing

The **input** instruction reads data from the input port. Similar to the **output** instruction, it has two forms:

```
input sX, (sY)
input sX, port_name
```

The sY register or port_name specifies the read port id. The retrieved data is stored in the sX register.

The timing diagram of an **input** instruction,

```
input s0, 02
```

is shown in Figure 16.3. When the instruction is executed, 02 is placed on port_id. After two clock cycles, in_port will be sampled at the rising edge of the clock and its value is stored in the s0 register. The external circuit must ensure that the input data is stable during the sampling edge to avoid timing violation.

As in the **output** instruction, the read_strobe signal is asserted in the second clock cycle. The function of the read_strobe signal is less obvious and is discussed in the next subsection.

16.3.2 Input interface

The input interface between PicoBlaze and input peripherals usually consists of a multiplexing circuit, which uses port_id as the selection signal to route the desired value to in_port. Sometimes, a decoding circuit similar to the one in the output interface is also necessary to signal the completion of the data access.

For the purpose of input interface design, an input port can be classified as a *continuous-access* or *single-access port*. For a continuous-access port, the data is presented continuously, such as the switch input of Section 15.4.1. On the other hand, the availability of data of a single-access port is triggered by a single discrete event, such as receiving a character in an UART buffer. The flag FF and buffers discussed in Section 7.2.4 are in this category. After the data is retrieved, we must remove it from the buffer to prevent the same data from

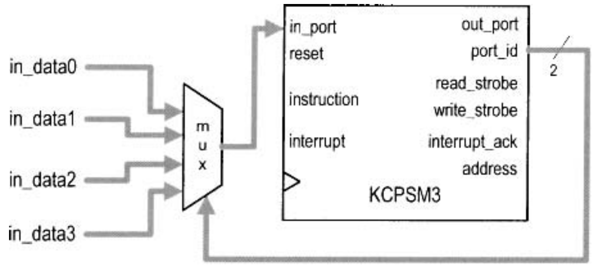

Figure 16.4 Block diagram of four continuous-access ports.

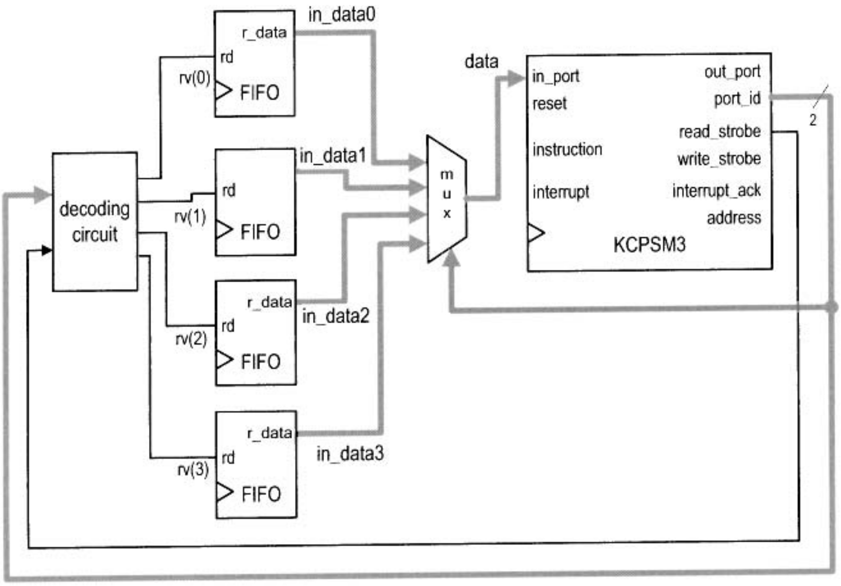

Figure 16.5 Block diagram of four single-access ports.

being processed again. This is usually done by utilizing a one-clock-cycle tick to clear the flag FF or remove a word from a FIFO buffer.

The interface for continuous-access ports involves only a multiplexing circuit. Consider an interface with four such ports. The block diagram is shown in Figure 16.4.

The interface for single-access ports needs a mechanism to remove the retrieved data from the buffer in the end of an **input** instruction. This can be done by using a decoding circuit that decodes the `port_id` and `read_strobe` signals. The circuit is identical to the decoding circuit of the output interface except that `write_strobe` is replaced by `read_strobe`. The decoded output can be considered as a "removal" signal, which is asserted for one clock cycle and removes the previously retrieved data. Consider an interface with four FIFOs. The diagram of the complete decoding and multiplexing circuit is shown in Figure 16.5. The `rv` signal is the decoded removal signal. In the end of an **input** instruction, 1 bit of this 4-bit signal is asserted and the corresponding FIFO performs a read operation, in which the first word is removed from the buffer. Assume that 00_{16}, 01_{16}, 02_{16}, and 03_{16} are assigned as the port ids. The HDL code segment for the interface is

```
-- multiplexing circuit
with port_id(1 downto 0) select
   data <= in_data0 when "00",
           in_data1 when "01",
           in_data2 when "10",
           in_data3 when others;
-- decoding circuit
process(reade_strobe,port_id)
begin
   if read_strobe='0' then
      rv <= "0000";
   else
      case port_id(1 downto 0) is
         when "00" =>
            rv <= "0001";
         when "01" =>
            rv <= "0010";
         when "10" =>
            rv <= "0100";
         when others =>
            rv <= "1000";
      end case;
   end if;
end process;
```

In a real application, it is likely that the input interface contains both continuous- and single-access ports. A decoding circuit is only needed for single-access ports.

16.4 SQUARE PROGRAM WITH A SWITCH AND SEVEN-SEGMENT LED DISPLAY INTERFACE

To demonstrate the construction of the PicoBlaze I/O interface, we add more versatile input and output peripherals to the square routine of Chapter 15. Recall that the square routine calculates $a^2 + b^2$, where a and b are 8-bit unsigned integers.

We use the 8-bit switch and a pushbutton to enter the values of a and b. The pushbutton generates a one-clock-cycle tick when pressed. The tick indicates that the current value of the switch should be loaded. The values of a and b are loaded alternately; i.e., the first pressing loads a, the second pressing loads b, the third pushing loads a, and so on. A second pushbutton is also included to clear the PicoBlaze's data RAM and relevant registers.

We use four seven-segment LEDs to display the inputs and computed results. The LEDs are arranged as four hexadecimal numbers. Since the range of $a^2 + b^2$ is up to 17 bits, the decimal point of the leftmost LED is used for the MSB. The three lower bits of the switch select what to display, which can be a, b, a^2, b^2, or $a^2 + b^2$.

In summary, the interface consists of the following:

- *Switch*: provides the values of a and b and selects the content of the LED display
- *Pushbutton 0*: loads the a and b alternatively when pressed
- *Pushbutton 1*: clears data RAM and relevant registers when pressed
- *Seven-segment LED*: displays the selected 17-bit value in four hexadecimal digits

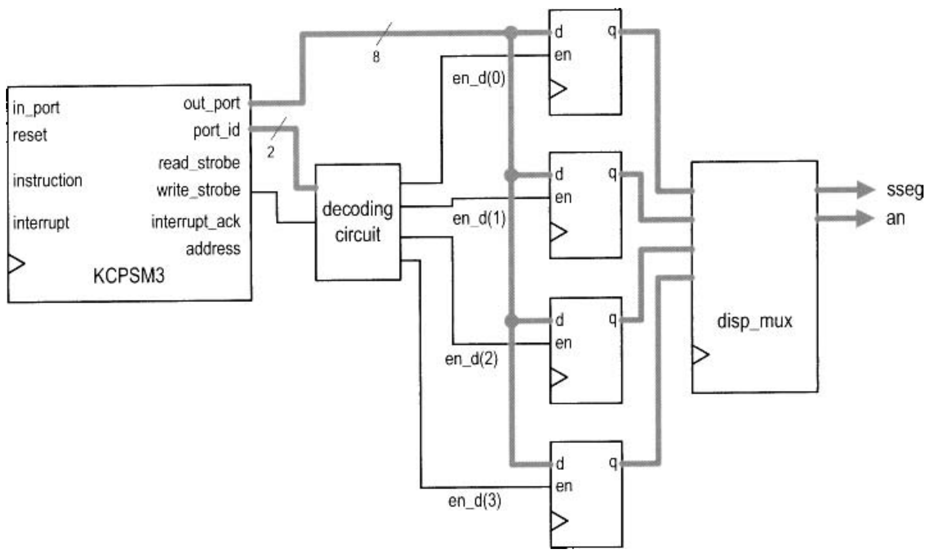

Figure 16.6 Output interface of a square circuit.

16.4.1 Output interface

Recall that the four seven-segment LEDs on the prototyping board share the same input pins, and a time-multiplexing circuit is required. For a PicoBlaze-based design, the multiplexing can be done by either an external circuit or a software routine. We use the external-circuit approach, which is simpler for assembly code development, in this section and discuss the software approach in Chapter 17. The LED time-multiplexing circuit designed in Section 4.5.1 can be used for this purpose. This circuit shields the timing and appears as four independent seven-segment LEDs for external system. The block diagram of the PicoBlaze output interface is shown in Figure 16.6. The interface consists of four 8-bit output ports, each port representing a seven-segment LED pattern.

In the assembly code, the four LED patterns are stored in PicoBlaze's data RAM with symbolic addresses of `led0`, `led1`, `led2`, and `led3`. The corresponding code segment is

```
...
;data RAM address alias
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13
...
;output port definitions
constant sseg0_port, 00     ;7-seg led 0
constant sseg1_port, 01     ;7-seg led 1
constant sseg2_port, 02     ;7-seg led 2
constant sseg3_port, 03     ;7-seg led 3
...
disp_led:
   fetch data, led0
   output data, sseg0_port
```

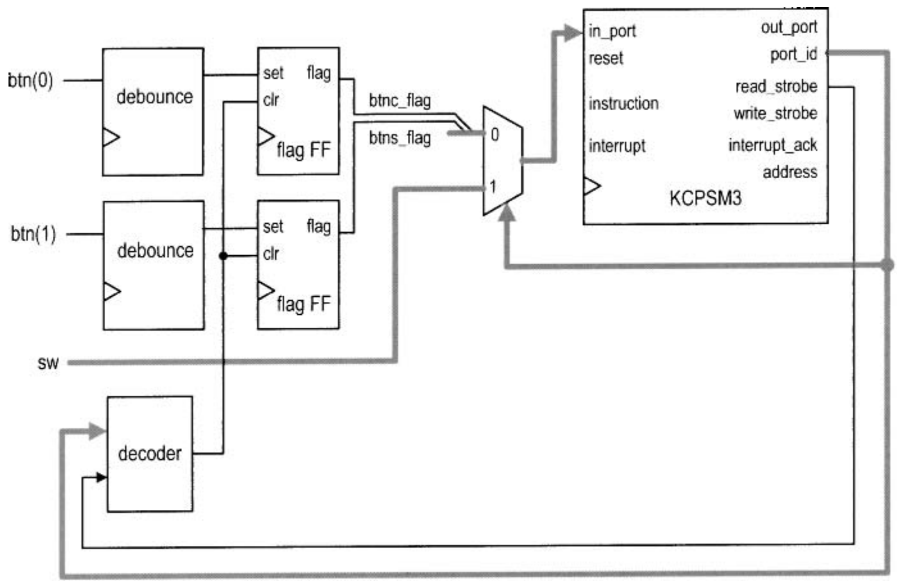

Figure 16.7 Input interface of a square circuit.

```
fetch data, led1
output data, sseg1_port
fetch data, led2
output data, sseg2_port
fetch data, led3
output data, sseg3_port
return
```

16.4.2 Input interface

The input interface consists of an 8-bit switch and two 1-bit pushbuttons. The former is a continuous-access port since the value is always present. The latter is a single-access port since pressing a button leads to only a single event (e.g., loading a to the register once rather than continuously). Because of the mechanical glitches, a debouncing circuit is needed to generate a clean one-clock-cycle tick. Since PicoBlaze's port can take up 8-bit data, inputs from the two pushbuttons can be grouped together as a single input port. The block diagram of the input interface is shown in Figure 16.7. The interface consists of two debouncing circuits, a two-to-one multiplexer, a decoding circuit, and two flag FFs. The function of the two flag FFs is discussed in Section 7.2.4. They provide a mechanism to set and clear the "button-pressing event." When a button is pressed, the debouncing circuit's output sets the flag. It remains asserted until it is retrieved by the PicoBlaze's **input** instruction, which sets the selection signal of the multiplexer to route the desired value to PicoBlaze's input port, and activates the clear signal. For clarity, we name the pushbutton 1 as the s button (for setting the value) and pushbutton 0 as the c button (for clearing the data RAM).

The pseudo code to process the input is

```
;input the button flags
;if c=1 then
```

```
    ;   call the clearing-ram routine
    ;if s=1 then
    ;   input switch value
    ;   store it to data ram
    ;   toggle a/b address offset
```

Since the s button inputs the values of a and b alternately, we use a global register, switch_a_b, to keep track of which one is being read currently. The register serves as the data RAM address offset, which can be 0 or 2, and its value toggles when the s button is pressed. The corresponding assembly code subroutine is

```
;input port definitions
constant rd_flag_port, 00   ;2 flags (xxxxxxsc):
constant sw_port, 01        ;8-bit switch
...
proc_btn:
   input s3, rd_flag_port   ;get flag
   ;check and process c button
   test s3, 01              ;check c button flag
   jump z, chk_btns         ;flag not set
   call init                ;flag set, clear
   jump proc_btn_done
chk_btns:
   ;check and process s button
   test s3, 02              ;check s button flag
   jump z, proc_btn_done    ;flag not set
   input data, sw_port      ;get switch
   load addr, a_lsb         ;get addr of a
   add addr, switch_a_b     ;add offset
   store data, (addr)       ;write data to ram
   ;update current disp position
   xor switch_a_b, 02       ;toggle between 00, 02
proc_btn_done:
   return
```

16.4.3 Assembly code development

After designing the I/O interface, we can derive the assembly program. The development follows the divide-and-conquer approach discussed in Chapter 15 and partitions the main program into several subroutines. The main program is

```
   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
   call disp_led            ;output led pattern
   jump forever
```

The complete code is shown in Listing 16.1.

The square subroutine is from Chapter 15, and the proc_btn and disp_led subroutines are discussed in the previous two subsections. The init subroutine performs system initialization. It uses a loop to load 0's to data RAM (i.e., clear the RAM) and sets the switch_a_b register to 0 (i.e., read a). The load_led_pttn subroutine reads the switch input, retrieves the desired values from the data RAM, converts the values to seven-segment LED patterns, and stores them to the corresponding locations in the data RAM. These patterns are then written to the output ports in the subsequent disp_led routine. The load_led_pttn routine consists of the get_upper_nibble and get_lower_nibble routines to extract the two hexadecimal digits and the hex_to_led routine to convert a hexadecimal digit to the corresponding seven-segment LED pattern.

The program requires more storage. In addition to the data RAM and registers required for the square subroutine, this program utilizes a new global register switch_a_b to keep track of whether a or b is being read, and 4 bytes in data RAM, whose addresses are labeled led0, led1, led2, and led3, to store four seven-segment LED patterns.

Listing 16.1 Square program with a switch and seven-segment LED interface

```
;=========================================================
; square circuit with 7-seg LED interface
;=========================================================
;program operation:
;   - read a and b from switch
;   - calculate a*a + b*b
;   - display data on 7-seg led

;=========================================================
; 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
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13

;=========================================================
; 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, switch_a_b  ;ram offset for current switch input

;=========================================================
; port alias
;=========================================================
;--------------input port definitions---------------------
```

```
constant rd_flag_port, 00   ;2 flags (xxxxxxsc):
constant sw_port, 01        ;8-bit switch
;------------output port definitions------------------
constant sseg0_port, 00     ;7-seg led 0
constant sseg1_port, 01     ;7-seg led 1
constant sseg2_port, 02     ;7-seg led 2
constant sseg3_port, 03     ;7-seg led 3

;=======================================================
; main program
;=======================================================
;calling hierarchy:
;
;main
;   - init
;   - proc_btn
;       - init
;   - square
;       - mult_soft
;   - load_led_pttn
;       - get_lower_nibble
;       - get_upper_nibble
;       - hex_to_led
;   - disp_led
;
; =======================================================

   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
   call disp_led            ;output led pattern
   jump forever

;=======================================================
;routine: init
;  function: perform initialization, clear register/ram
;  output register:
;     switch_a_b: cleared to 0
;  temp register: data, i
;=======================================================
init:
   ;clear memory
   load i, 40               ;unitize loop index to 64
   load data, 00
clr_mem_loop:
   store data, (i)
   sub i, 01                ;dec loop index
   jump nz, clr_mem_loop    ;repeat until i=0
   ;clear register
   load switch_a_b, 00
```

```
    return

;=============================================================
;routine: proc_btn
;  function: check two buttons and process the display
;  input reg:
;     switch_a_b: ram offset (0 for a and 2 for b)
;  output register:
;     s3: store input port flag
;     switch_a_b: may be toggled
;  temp register used: data, addr
;=============================================================
proc_btn:
    input s3, rd_flag_port   ;get flag
    ;check and process c button
    test s3, 01              ;check c button flag
    jump z, chk_btns         ;flag not set
    call init                ;flag set, clear
    jump proc_btn_done
chk_btns:
    ;check and process s button
    test s3, 02              ;check s button flag
    jump z, proc_btn_done    ;flag not set
    input data, sw_port      ;get switch
    load addr, a_lsb         ;get addr of a
    add addr, switch_a_b     ;add offset
    store data, (addr)       ;write data to ram
    ;update current disp position
    xor switch_a_b, 02       ;toggle between 00, 02
proc_btn_done:
    return

;=============================================================
;routine: load_led_pttn
;  function: read 3 LSBs of switch input and convert the
;            desired values to four led patterns and
;            load them to ram
;            switch: 000:a; 001:b; 010:a^2; 011:b^2;
;                    others: a^2 + b^2
;  temp register used: data, addr
;    s6: data from sw input port

;=============================================================
load_led_pttn:
    input s6, sw_port        ;get switch
    sl0 s6                   ;*2 to obtain addr offset
    compare s6, 08           ;sw>100?
    jump c, sw_ok            ;no
    load s6, 08              ;yes, sw error, make default
sw_ok:
    ;process byte 0, lower nibble
    load addr, a_lsb
    add addr, s6             ;get lower addr
```

```
    fetch data, (s6)          ;get lower byte
    call get_lower_nibble     ;get lower nibble
    call hex_to_led           ;convert to led pattern
    store data, led0
    ;process byte 0, upper nibble
    fetch data, (addr)
    call get_upper_nibble
    call hex_to_led
    store data, led1
    ;process byte 1, lower nibble
    add addr, 01              ;get upper addr
    fetch data, (addr)
    call get_lower_nibble
    call hex_to_led
    store data, led2
    ;process byte 1, upper nibble
    fetch data, (addr)
    call get_upper_nibble
    call hex_to_led
    ;check for sw=100 to process carry as led dp
    compare s6, 08            ;display final result?
    jump nz, led_done         ;no
    add addr, 01              ;get carry addr
    fetch s6, (addr)          ;s6 to store carry
    test s6, 01               ;carry=1?
    jump z, led_done          ;no
    and data, 7F              ;yes, assert msb (dp) to 0
led_done:
    store data, led3
    return

;=========================================================
;routine: disp_led
;  function: output four led patterns
;  temp register used: data
;=========================================================
disp_led:
    fetch data, led0
    output data, sseg0_port
    fetch data, led1
    output data, sseg1_port
    fetch data, led2
    output data, sseg2_port
    fetch data, led3
    output data, sseg3_port
    return

;=========================================================
;routine: hex_to_led
;  function: convert a hex digit to 7-seg led pattern
;  input register: data
;  output register: data
;=========================================================
```

```
hex_to_led:
   compare data, 00
   jump nz, comp_hex_1
   load data, 81           ;7-seg pattern 0
   jump hex_done
comp_hex_1:
   compare  data, 01
   jump nz, comp_hex_2
   load data, CF           ;7-seg pattern 1
   jump hex_done
comp_hex_2:
   compare data, 02
   jump nz, comp_hex_3
   load data, 92           ;7-seg pattern 2
   jump hex_done
comp_hex_3:
   compare data, 03
   jump nz, comp_hex_4
   load data, 86           ;7-seg pattern 3
   jump hex_done
comp_hex_4:
   compare data, 04
   jump nz, comp_hex_5
   load data, CC           ;7-seg pattern 4
   jump hex_done
comp_hex_5:
   compare data, 05
   jump nz, comp_hex_6
   load data, A4           ;7-seg pattern 5
   jump hex_done
comp_hex_6:
   compare data, 06
   jump nz, comp_hex_7
   load data, A0           ;7-seg pattern 6
   jump hex_done
comp_hex_7:
   compare data, 07
   jump nz, comp_hex_8
   load data, 8F           ;7-seg pattern 7
   jump hex_done
comp_hex_8:
   compare data, 08
   jump nz, comp_hex_9
   load data, 80           ;7-seg pattern 8
   jump hex_done
comp_hex_9:
   compare data, 09
   jump nz, comp_hex_a
   load data, 84           ;7-seg pattern 9
   jump hex_done
comp_hex_a:
   compare data, 0A
   jump nz, comp_hex_b
```

```
    load data, 88            ;7-seg pattern a
    jump hex_done
  comp_hex_b:
    compare data, 0B
    jump nz, comp_hex_c
    load data, E0            ;7-seg pattern b
    jump hex_done
  comp_hex_c:
    compare data, 0C
    jump nz, comp_hex_d
    load data, B1            ;7-seg pattern C
    jump hex_done
  comp_hex_d:
    compare data, 0D
    jump nz, comp_hex_e
    load data, C2            ;7-seg pattern d
    jump hex_done
  comp_hex_e:
    compare data, 0E
    jump nz, comp_hex_f
    load data, B0            ;7-seg pattern E
    jump hex_done
  comp_hex_f:
    load data, B8            ;7-seg pattern F
hex_done:
    return

;=========================================================
;routine: get_lower_nibble
;  function: get lower 4 bits of data
;  input register: data
;  output register: data
;=========================================================
get_lower_nibble:
    and data, 0F             ;clear upper nibble
    return

;=========================================================
;routine: get_upper_nibble
;  function: get upper 4 bits of in_data
;  input register: data
;  output register: data
;=========================================================
get_upper_nibble:
    sr0 data                 ;right shift 4 times
    sr0 data
    sr0 data
    sr0 data
    return

;=========================================================
;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
;=========================================================
square:
   ;calculate a*a
   fetch s3, a_lsb          ;load a
   fetch s4, a_lsb          ;load a
   call mult_soft           ;calculate a*a
   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
   call mult_soft           ;calculate b*b
   store s6, bb_lsb         ;store lower byte of b*b
   store s5, bb_msb         ;store upper byte of b*b
   ;calculate a*a+b*b
   fetch data, aa_lsb       ;get lower byte of a*a
   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
   load data, 00            ;clear data, but keep carry
   addcy data, 00           ;get carry from previous +
   store data, aabb_cout    ;store carry of a*a+b*b
   return

;=========================================================
;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
   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
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
   sub i, 01                ;dec loop index
```

```
jump nz, mult_loop      ;repeat until i=0
return
```

16.4.4 VHDL code development

The complete HDL code simply combines the PicoBlaze processor, instruction ROM, the input interface and peripherals shown in Figure 16.7, and the output interface and peripherals shown in Figure 16.6. It is shown in Listing 16.2.

Listing 16.2 PicoBlaze with a switch and seven-segment LED interface

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pico_btn is
   port(
      clk, reset: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(1 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end pico_btn;

architecture arch of pico_btn is
   -- KCPSM3/ROM signals
   signal address: std_logic_vector(9 downto 0);
   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, read_strobe: std_logic;
   signal interrupt, interrupt_ack: std_logic;
   signal kcpsm_reset: std_logic;
   -- I/O port signals
   -- output enable
   signal en_d: std_logic_vector(3 downto 0);
   -- four-digit seven-segment led display
   signal ds3_reg, ds2_reg: std_logic_vector(7 downto 0);
   signal ds1_reg, ds0_reg: std_logic_vector(7 downto 0);
   -- two pushbuttons
   signal btnc_flag_reg, btnc_flag_next: std_logic;
   signal btns_flag_reg, btns_flag_next: std_logic;
   signal set_btnc_flag, set_btns_flag: std_logic;
   signal clr_btn_flag: std_logic;
begin
   --=====================================================
   --  I/O modules
   --=====================================================
   disp_unit: entity work.disp_mux
      port map(
         clk=>clk, reset=>'0',
         in3=>ds3_reg, in2=>ds2_reg, in1=>ds1_reg,
         in0=>ds0_reg, an=>an, sseg=>sseg);
```

```
btnc_db_unit: entity work.debounce
   port map(
      clk=>clk, reset=>reset, sw=>btn(0),
      db_level=>open, db_tick=>set_btnc_flag);
btns_db_unit: entity work.debounce
   port map(
      clk=>clk, reset=>reset, sw=>btn(1),
      db_level=>open, db_tick=>set_btns_flag);
--=================================================
-- KCPSM and ROM instantiation
--=================================================
proc_unit: entity work.kcpsm3
   port map(
      clk=>clk, reset =>kcpsm_reset,
      address=>address, instruction=>instruction,
      port_id=>port_id, write_strobe=>write_strobe,
      out_port=>out_port, read_strobe=>read_strobe,
      in_port=>in_port, interrupt=>interrupt,
      interrupt_ack=>interrupt_ack);
rom_unit: entity work.btn_rom
   port map(
       clk => clk, address=>address,
       instruction=>instruction);
-- unused inputs on processor
kcpsm_reset <= '0';
interrupt <= '0';
--=================================================
--  output interface
--=================================================
--     outport port id:
--        0x00: ds0
--        0x01: ds1
--        0x02: ds2
--        0x03: ds3
--=================================================
-- registers
process (clk)
begin
   if (clk'event and clk='1') then
      if en_d(0)='1' then ds0_reg <= out_port; end if;
      if en_d(1)='1' then ds1_reg <= out_port; end if;
      if en_d(2)='1' then ds2_reg <= out_port; end if;
      if en_d(3)='1' then ds3_reg <= out_port; end if;
   end if;
end process;
-- decoding circuit for enable signals
process(port_id,write_strobe)
begin
   en_d <= (others=>'0');
   if write_strobe='1' then
      case port_id(1 downto 0) is
         when "00" => en_d <="0001";
         when "01" => en_d <="0010";
```

```
            when "10" => en_d <="0100";
            when others => en_d <="1000";
         end case;
      end if;
   end process;
   --=========================================================
   --  input interface
   --=========================================================
   --    input port id
   --      0x00: flag
   --      0x01: switch
   --=========================================================
   -- input register (for flags)
   process(clk)
   begin
      if (clk'event and clk='1') then
         btnc_flag_reg <= btnc_flag_next;
         btns_flag_reg <= btns_flag_next;
      end if;
   end process;

   btnc_flag_next <= '1' when set_btnc_flag='1' else
                     '0' when clr_btn_flag='1' else
                      btnc_flag_reg;
   btns_flag_next <= '1' when set_btns_flag='1' else
                     '0' when clr_btn_flag='1' else
                      btns_flag_reg;
   -- decoding circuit for clear signals
   clr_btn_flag <='1' when read_strobe='1' and
                           port_id(0)='0' else
                  '0';
   -- input multiplexing
   process(port_id,btns_flag_reg,btnc_flag_reg,sw)
   begin
      case port_id(0) is
         when '0' =>
            in_port <= "000000" &
                       btns_flag_reg & btnc_flag_reg;
         when others =>
            in_port <= sw;
      end case;
   end process;
end arch;
```

16.5 SQUARE PROGRAM WITH A COMBINATIONAL MULTIPLIER AND UART CONSOLE

In this section, we add two more I/O peripherals to the previous design. One is a combinational multiplier, which accelerates the multiplication, and the other is an UART, which provides a communication link to a PC.

16.5.1 Multiplier interface

Since PicoBlaze does not contain a hardware multiplier, the multiplication is done by a software routine, mult_soft. It uses a shift-and-add algorithm to iterate through the 8-bit multiplier and requires about 60 instructions in the worst-case scenario. An alternative is to utilize the Spartan-3 device's built-in combinational multiplier.

Since PicoBlaze provides no mechanism to use a coprocessor, the multiplier must be configured as an I/O peripheral. We can create an 8-bit combinational multiplier that takes two 8-bit operands and returns a 16-bit product. To facilitate this peripheral, the PicoBlaze's interface requires two additional output ports and buffers for the two operands and two additional input ports for the 16-bit product. The assembly routine now only needs to pass the operands to the output ports and then retrieve the results from the input ports. The code becomes

```
;input port definitions
constant mult_prod0_port, 03 ;multiplication product 8 LSBs
constant mult_prod1_port, 04 ;multiplication product 8 MSBs
;output port definitions
constant mult_src0_port, 05  ;multiplier operand 0
constant mult_src1_port, 06  ;multiplier operand 1
...
mult_hard:
   output s3, mult_src0_port
   output s4, mult_src1_port
   input s5, mult_prod1_port
   input s6, mult_prod0_port
   return
```

Note that the combinational multiplier can complete the computation with one instruction (i.e., two clock cycles), and thus no additional timing mechanism is needed in the code. This routine can be used in place of the previous mult_soft routine.

16.5.2 UART interface

With the UART interface, information can be entered and displayed in Windows HyperTerminal, which is more flexible and versatile than switches and LEDs. We use it as a simple control console for the square routine. A representative screen is shown in Figure 16.8. The console generates an SQ> prompt and a user can respond with a lowercase a, b, c, or d character. The a and b characters are used to input values for a and b of the square routine. When the key is pressed, the value of the 8-bit switch is read and stored into the corresponding data RAM location. The c character is used to clear the data RAM and reinitialize the program. Its function is identical to that of the c button. The d character leads to a "data RAM dump," in which the 64 bytes of the data RAM are displayed on screen. This allows us to observe the various values of the square routine and the four seven-segment LED patterns. An Error message is returned for all other characters.

The UART module designed in Section 7.4 can be used for this purpose. Since the transmission and receiving FIFO buffers provide a storage and flagging mechanism, no additional circuit is needed. We need only expand the decoding and multiplexing circuits to accommodate the additional I/O ports. The UART interface block diagram is sketched in Figure 16.9, in which the other I/O peripherals are omitted to reduce clutter. PicoBlaze's output port, out_port, is connected to w_data of UART. The decoded enable signal is connected to wr_uart, and the data is written to UART transmitting FIFO when it is

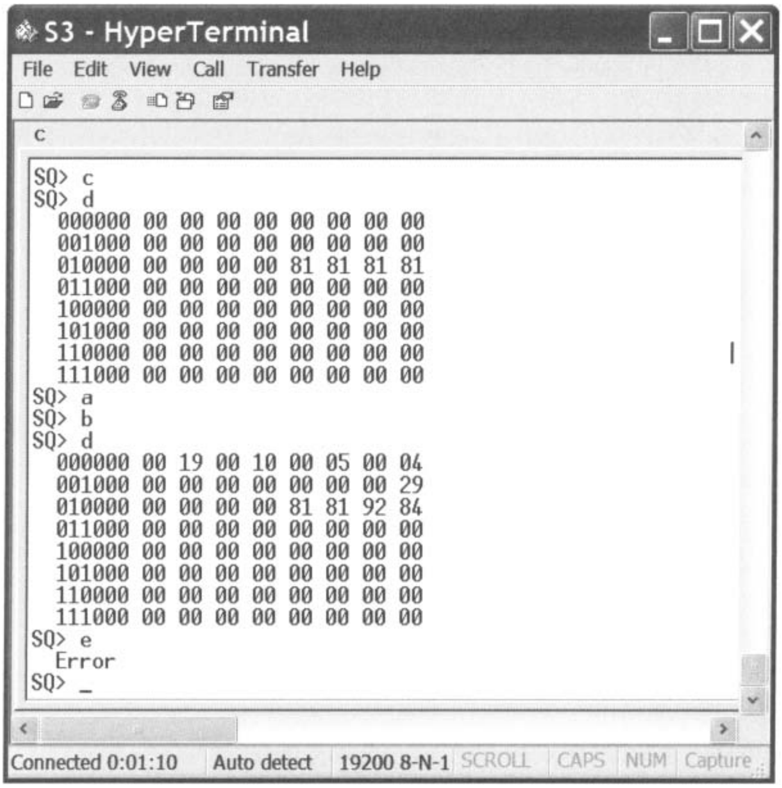

Figure 16.8 Representative console screen.

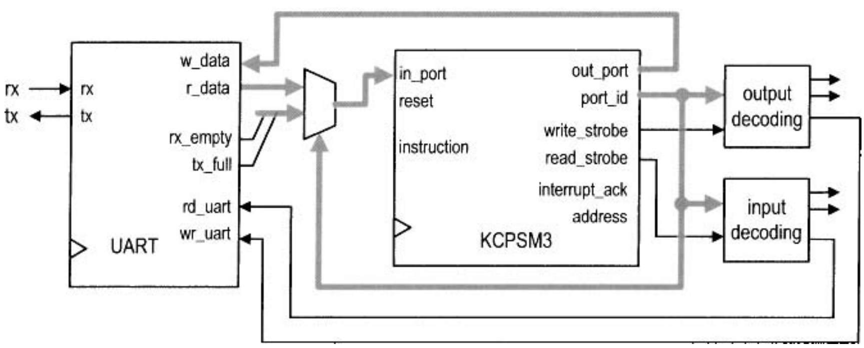

Figure 16.9 UART I/O interface.

asserted. Similarly, r_data of UART is routed to PicoBlaze's input multiplexing circuit, and the decoded clear signal is connected to rd_uart. When the UART receiving FIFO port is specified in an **input** instruction, the receiving FIFO's output is routed to PicoBlaze's input port, in_port, and the decoded remove signal is asserted one clock cycle to remove one word from the receiving FIFO. The UART interface also needs to route the two status signals, rx_empty and tx_full, to PicoBlaze's input multiplexing circuit. The assembly program needs to check the status before reading or writing the UART's FIFOs. Since the signals are only 2 bits wide, they can be grouped with the previous s and c buttons in the same input port.

16.5.3 Assembly code development

Since the previous assembly code is developed in a modular fashion, we can expand the program by adding a routine, proc_uart, to process UART transactions. The main program becomes

```
    call init                ;initialization
forever:
    ;main loop body
    call proc_btn            ;check & process buttons
    call proc_uart           ;check & process uart rx
    call square              ;calculate square
    call load_led_pttn       ;store led patterns to ram
    call disp_led            ;output led pattern
    jump forever
```

Because of the complexity of the required console operation, the proc_uart is quite involved. The pseudo code of this routine is

```
;   if (no character in UART receiving FIFO) then
;      return
;   input characters from FIFO
;   if (characters is a) then
;      input switch value
;      store it to data ram
;      display prompt
;      return
;   if (characters is b) then
;      input switch value
;      store it to data ram
;      display prompt
;      return
;   if (characters is c) then
;      perform initialization
;      return
;   if (characters is d) then
;      dump data ram
;      return
;   display error message
;   return
```

We follow the modular development approach and further divide this routine into simpler routines. A key low-level routine is tx_one_byte, which transmits 1 byte via the UART port. Its code is

```
;input port definitions
constant rd_flag_port, 00
; 4 flags (xxxxtrsc):
;   t: uart tx full, r: uart rx not empty
;   s: s button flag, c: c button flag
;output port definitions
constant uart_tx_port, 04     ;uart receiver port
;register alias
namereg sd, tx_data           ;data to be tx by uart
...
tx_one_byte:
   input s6, rd_flag_port
   test s6, 08                  ;check uart_tx_full
   jump nz, tx_one_byte         ;yes, keep on waiting
   output tx_data, uart_tx_port ;no, write to uart tx fifo
   return
```

Since PicoBlaze's processing speed is much higher than the UART's transmission speed, we must prevent buffer overflow. The routine keeps on checking the status of the transmitting FIFO buffer, and writes data only when the buffer is not full.

The task of dumping data RAM requires the most work. It displays the data RAM address and contents as an 8-by-8 table, which lists the byte address first and then the 8 bytes of data in hexadecimal format, as in

```
001000 00 0F 00 09 00 04 00 03
010000 00 00 FF 1D 00 00 00 19
 . . .
111000 00 00 00 00 00 FF FF FF
```

The routine consists of three major routines: `disp_ram_addr`, which sends ASCII codes to display the 5-bit base address in binary format; `disp_ram_data`, which sends ASCII codes to display 8 bytes of data; and `hex_to_ascii`, which converts a hexadecimal digit to the corresponding ASCII code.

The complete code is shown in Listing 16.3. It includes detailed comments to explain operation of the subroutines. The unmodified subroutines of Listing 16.1 are omitted.

Listing 16.3 Square program with a UART console

```
;=========================================================
; square circuit with UART and multiplier interface
;=========================================================
;program operation:
;   - read a and b from switch
;   - calculate a*a + b*b
;   - display data on HyperTerminal and 7-seg led

;=========================================================
; data constants
;=========================================================
;selected ASCII codes
constant ASCII_0, 30
constant ASCII_1, 31
constant ASCII_2, 32
constant ASCII_3, 33
constant ASCII_a, 61
```

```
constant ASCII_b,  62
constant ASCII_c,  63
constant ASCII_d,  64
constant ASCII_o,  6F
constant ASCII_r,  72
constant ASCII_E,  45
constant ASCII_S,  53
constant ASCII_Q,  51
constant ASCII_D_U,44    ; uppercase D
constant ASCII_GT, 3E    ; >
constant ASCII_SP, 20    ; space
constant ASCII_CR, 0D    ; carriage return
constant ASCII_LF, 0A    ; line feed

;=====================================================
; 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
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13

;=====================================================
; 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 sc, switch_a_b  ;ram offset for current switch input
namereg sd, tx_data     ;data to be tx by uart

;=====================================================
; port alias
;=====================================================
;------------input port definitions------------------
constant rd_flag_port, 00
; 4 flags (xxxxtrsc):
;    t: uart tx full
;    r: uart rx not empty
;    s: s button flag
;    c: c button flag
constant sw_port,          01    ;8-bit switches
```

```
constant uart_rx_port,     02    ;uart receiver port
constant mult_prod0_port, 03    ;multiplication product 8 LSBs
constant mult_prod1_port, 04    ;multiplication product 8 MSBs
;--------------output port definitions------------------------
constant sseg0_port,      00    ;7-seg led 0
constant sseg1_port,      01    ;7-seg led 1
constant sseg2_port,      02    ;7-seg led 2
constant sseg3_port,      03    ;7-seg led 3
constant uart_tx_port,    04    ;uart receiver port
constant mult_src0_port, 05    ;multiplier operand 0
constant mult_src1_port, 06    ;multiplier operand 1

;=========================================================
; main program
;=========================================================
;calling hierarchy:
;
;main
;   - init
;       - tx_prompt
;           - tx_one_byte
;   - proc_btn
;       - init
;   - proc_uart
;       - tx_prompt
;       - init
;       - proc_uart_err
;           - tx_one_byte
;       - dump_mem
;           - tx_prompt
;           - disp_ram_addr
;               - tx_one_byte
;           - disp_ram_data
;               - tx_one_byte
;               - get_upper_nibble
;               - get_lower_nibble
;               - hex_to_ascii
;   - square
;       - mult_hard
;   - load_led_pttn
;       - get_lower_nibble
;       - get_upper_nibble
;       - hex_to_led
;   - disp_led
;
;=========================================================
   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call proc_uart           ;check & process uart rx
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
```

```
   call disp_led            ;output led pattern
   jump forever

;==========================================================
;routine: init
;  function: perform initialization, clear register/ram
;  output register:
;    switch_a_b: cleared to 0
;  temp register: data, i
;==========================================================
init:
   ;clear memory
   load i, 40               ;unitize loop index to 64
   load data, 00
clr_mem_loop:
   store data, (i)
   sub i, 01                ;dec loop index
   jump nz, clr_mem_loop    ;repeat until i=0
   ;clear register
   load switch_a_b, 00
   call tx_prompt
   return

;==========================================================
;routine: proc_uart
;  function: read uart input char:
;    a or b: read a or b from switch;
;    c: clear;  d: dump/display data ram  other: error
;  input reg: s3 (input port flag)
;  temp register used: data
;    s4: store received uart char or 00 (no uart input)
;==========================================================
proc_uart :
   test s3, 04              ;check uart rx status
   jump z, uart_rx_done     ;go to done if rx empty
   ;process received char
   input s4, uart_rx_port   ;get char
   ;check if received char is a
   compare s4, ASCII_a      ;check ASCII a
   jump nz, chk_ascii_b     ;no, check next
   input data, sw_port      ;get switch
   store data, a_lsb        ;write a to data ram
   call tx_prompt           ;new prompt line
   jump uart_rx_done
chk_ascii_b:
   ;check if received char is b
   compare s4, ASCII_b      ;check ASCII b
   jump nz, chk_ascii_c     ;no, check next
   input data, sw_port      ;get switch
   store data, b_lsb        ;write b to data ram
   call tx_prompt           ;new prompt line
   jump uart_rx_done
chk_ascii_c:
```

```
    ;check if received char is c
    compare s4, ASCII_c      ;check ASCII c
    jump nz, chk_ascii_d     ;no check next
    call init                ;clear
    jump uart_rx_done
 chk_ascii_d:
    ;check if received char is d
    compare s4, ASCII_d        ;check ASCII d
    jump nz, ascii_undefined
    call dump_mem            ;dump/display ram
    jump uart_rx_done
 ascii_undefined:
    ;undefined char
    call proc_uart_error
 uart_rx_done:
    return

 ;=========================================================
 ;routine: proc_uart_error
 ;  function: display "Error" for unknown uart char
 ;=========================================================
 proc_uart_error:
    load tx_data, ASCII_LF
    call tx_one_byte         ;transmit LF
    load tx_data, ASCII_CR
    call tx_one_byte         ;transmit CR
    load tx_data, ASCII_SP
    call tx_one_byte         ;transmit SP
    call tx_one_byte         ;transmit SP
    load tx_data, ASCII_E
    call tx_one_byte         ;transmit E
    load tx_data, ASCII_r
    call tx_one_byte         ;transmit r
    load tx_data, ASCII_r
    call tx_one_byte         ;transmit r
    load tx_data, ASCII_o
    call tx_one_byte         ;transmit o
    load tx_data, ASCII_r
    call tx_one_byte         ;transmit r
    call tx_prompt
    return

 ;=========================================================
 ;routine: dump_mem
 ;  function: when d received, dump 64 bytes of ram as
 ;     001000 XX XX XX XX XX XX XX XX
 ;     010000 XX XX XX XX XX XX XX XX
 ;       . . .
 ;     111000 XX XX XX XX XX XX XX XX
 ;  temp register used:
 ;     s3: as outer loop index
 ;     s4: ram base address
 ;=========================================================
```

```
dump_mem:
    load s3, 00              ;addr used as loop index
 dump_loop:
    ;loop body
    load s4, s3              ;get ram base addr (xxx000)
    sl0 s4
    sl0 s4
    sl0 s4
    call disp_ram_addr
    call disp_ram_data
    add s3, 01               ;inc loop index
    compare s3, 08
    jump nz, dump_loop       ;loop not reach 8 yet
    call tx_prompt           ;new prompt
    return

;=========================================================
;routine: tx_prompt
;  function: generate prompt "SQ>"
;  temp register: tx_data
;=========================================================
tx_prompt:
    load tx_data, ASCII_LF
    call tx_one_byte         ;transmit LF
    load tx_data, ASCII_CR
    call tx_one_byte         ;transmit CR
    load tx_data, ASCII_S
    call tx_one_byte         ;transmit S
    load tx_data, ASCII_Q
    call tx_one_byte         ;transmit Q
    load tx_data, ASCII_GT
    call tx_one_byte         ;transmit >
    load tx_data, ASCII_SP
    call tx_one_byte         ;transmit SP
    return

;=========================================================
;routine: disp_ram_addr
;  function: display 6-bit ram addr
;      bbb000
;  input register:
;      s4: base address
;  temp register:
;      i, s7: 1-bit mask
;=========================================================
disp_ram_addr:
    ;new line
    load tx_data, ASCII_LF
    call tx_one_byte         ;transmit LF
    load tx_data, ASCII_CR
    call tx_one_byte         ;transmit CR
    load tx_data, ASCII_SP
    call tx_one_byte         ;transmit SP
```

```
    call tx_one_byte          ;transmit SP
    ;initialize the loop index and mask
    load i, 06                ;addr used as loop index
    load s7, 20               ;set mask to 0010_0000
tx_loop:
    ;loop body
    load tx_data, ASCII_1     ;load default ASCII 1
    test s7, s4               ;check the bit
    jump nz, tx_01            ;the bit is 1
    load tx_data, ASCII_0;    ;the bit is 0, load ASCII 0
tx_01:
    call tx_one_byte          ;transmit the ASCII 1 or 0
    ;update loop index and mask
    sr0 s7                    ;shift mask bit
    sub i, 01                 ;dec loop index
    jump nz, tx_loop          ;loop not reach 0 yet
    ;done with loop, send ASCII space
    load tx_data, ASCII_SP    ;load ASCII SP
    call tx_one_byte          ;transmit SP
    return

;=====================================================
;routine: disp_ram_data
;   function: 8-byte data in form of
;       00 11 22 33 44 55 66 77 88
;   input register:
;       s4: ram base address (xxx000)
;   temp register: i, addr, data
;=====================================================
disp_ram_data:
    ;initialize the loop index and mask
    load i, 08                ;addr used as loop index
d_ram_loop:
    ;loop body
    load addr, s4
    add addr, i
    sub addr, 01               ;calculate addr offset
    ;send upper nibble
    fetch data, (addr)
    call get_upper_nibble
    call hex_to_ascii         ;convert to ascii
    load tx_data, data
    call tx_one_byte
    ;send lower nibble
    fetch data, (addr)
    call get_lower_nibble
    call hex_to_ascii         ;convert to ascii
    load tx_data, data
    call tx_one_byte
    ;send a space
    load tx_data, ASCII_SP;
    call tx_one_byte          ;transmit SP
    sub i, 01                 ;dec loop index
```

```
    jump nz, d_ram_loop      ;loop not reach 0 yet
    return

;==========================================================
;routine: hex_to_ascii
;   function: convert a hex number to ascii code
;             add 30 for 0-9, add 37 for A-F
;   input register: data
;==========================================================
hex_to_ascii:
    compare data, 0a
    jump c, add_30           ;0 to 9, offset 30
    add data, 07             ;a to f, extra offset 07
add_30:
    add data, 30
    return

;==========================================================
;routine: tx_one_byte
;   function: wait until uart tx fifo not full;
;             then write a byte to fifo
;   input register: tx_data
;   temp register:
;      s6: read port flag
;==========================================================
tx_one_byte:
    input s6, rd_flag_port
    test s6, 08                  ;check uart_tx_full
    jump nz, tx_one_byte         ;yes, keep on waiting
    output tx_data, uart_tx_port ;no, write to uart tx fifo
    return

;==========================================================
;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
;==========================================================
square:
    ;calculate a*a
    fetch s3, a_lsb          ;load a
    fetch s4, a_lsb          ;load a
    call mult_hard           ;calculate a*a
    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
    call mult_hard           ;calculate b*b
    store s6, bb_lsb         ;store lower byte of b*b
    store s5, bb_msb         ;store upper byte of b*b
    ;calculate a*a+b*b
    fetch data, aa_lsb       ;get lower byte of a*a
```

```
    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
    load data, 00              ;clear data, but keep carry
    addcy data, 00             ;get carry from previous +
    store data, aabb_cout      ;store carry of a*a+b*b
    return

;=====================================================================
;routine: mult_hard
;   function: 8-bit unsigned multiplication using
;               external combinational multiplier;
;   input register:
;      s3: multiplicand
;      s4: multiplier
;   output register:
;      s5: upper byte of product
;      s6: lower byte of product
;   temp register:
;=====================================================================
mult_hard:
    output s3, mult_src0_port
    output s4, mult_src1_port
    input s5, mult_prod1_port
    input s6, mult_prod0_port
    return

;=====================================================================
;The following are the same as the previous Listing:
;   proc_btn, load_led_pttn, disp_led
;   hex_to_led, get_lower_nibble, get_upper_nibble
;   ...
;=====================================================================
```

16.5.4 VHDL code development

The new square circuit adds a UART and a combinational multiplier to an I/O interface. The former is the module discussed in Section 7.4, and the latter can be inferred from the HDL's * operator. The decoding and multiplexing parts of HDL code in Listing 16.2 can be expanded to accommodate the two new peripherals. The complete VHDL code is shown in Listing 16.4. The detailed I/O port address assignment can be found in the header section of Listing 16.3.

Listing 16.4 PicoBlaze with UART console and multiplier interface

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pico_uart is
   port(
      clk, reset: in std_logic;
```

```
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(3 downto 0);
      rx: in std_logic;
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0);
      tx: out  std_logic
   );
end pico_uart;

architecture arch of pico_uart is
   -- KCPSM3/ROM signals
   signal address: std_logic_vector(9 downto 0);
   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, read_strobe: std_logic;
   signal interrupt, interrupt_ack: std_logic;
   signal kcpsm_reset: std_logic;
   -- I/O port signals
   -- output enable
   signal en_d: std_logic_vector(6 downto 0);
   -- four-digit seven-segment led display
   signal ds3_reg, ds2_reg: std_logic_vector(7 downto 0);
   signal ds1_reg, ds0_reg: std_logic_vector(7 downto 0);
   -- two pushbuttons
   signal btnc_flag_reg, btnc_flag_next: std_logic;
   signal btns_flag_reg, btns_flag_next: std_logic;
   signal set_btnc_flag, set_btns_flag: std_logic;
   signal clr_btn_flag: std_logic;
   -- uart
   signal w_data: std_logic_vector(7 downto 0);
   signal rd_uart, rx_not_empty, rx_empty: std_logic;
   signal wr_uart, tx_full: std_logic;
   signal rx_char: std_logic_vector(7 downto 0);
   -- multiplier
   signal m_src0_reg, m_src1_reg: std_logic_vector(7 downto 0);
   signal prod: std_logic_vector(15 downto 0);
begin
   --=====================================================
   --  I/O modules
   --=====================================================
   disp_unit: entity work.disp_mux
      port map(
         clk=>clk, reset=>'0',
         in3=>ds3_reg, in2=>ds2_reg, in1=>ds1_reg,
         in0=>ds0_reg, an=>an, sseg=>sseg);
   uart_unit: entity work.uart(str_arch)
      port map(
         clk=>clk, reset=>reset, rd_uart=>rd_uart,
         wr_uart=>wr_uart, rx=>rx,
         w_data=>out_port, tx_full=>tx_full,
         rx_empty=>rx_empty, r_data=>rx_char, tx=>tx);
   btnc_db_unit: entity work.debounce
```

```
      port map(
         clk=>clk, reset=>reset, sw=>btn(0),
         db_level=>open, db_tick=>set_btnc_flag);
   btns_db_unit: entity work.debounce
      port map(
         clk=>clk, reset=>reset, sw=>btn(1),
         db_level=>open, db_tick=>set_btns_flag);
   -- combinational multiplier
   prod <= std_logic_vector
           (unsigned(m_src0_reg) * unsigned(m_src1_reg));
   -- =================================================
   --  KCPSM and ROM instantiation
   -- =================================================
   proc_unit: entity work.kcpsm3
      port map(
         clk=>clk, reset =>kcpsm_reset,
         address=>address, instruction=>instruction,
         port_id=>port_id, write_strobe=>write_strobe,
         out_port=>out_port, read_strobe=>read_strobe,
         in_port=>in_port, interrupt=>interrupt,
         interrupt_ack=>interrupt_ack);
   rom_unit: entity work.uart_rom
      port map(
          clk => clk, address=>address,
          instruction=>instruction);
   -- unused inputs on processor
   kcpsm_reset <= '0';
   interrupt <= '0';
   -- =================================================
   --  output interface
   -- =================================================
   --    outport port id:
   --      0x00: ds0
   --      0x01: ds1
   --      0x02: ds2
   --      0x03: ds3
   --      0x04: uart_tx_fifo
   --      0x05: m_src0
   --      0x06: m_src1
   -- =================================================
   -- registers
   process (clk)
   begin
      if (clk'event and clk='1') then
         if en_d(0)='1' then ds0_reg <= out_port; end if;
         if en_d(1)='1' then ds1_reg <= out_port; end if;
         if en_d(2)='1' then ds2_reg <= out_port; end if;
         if en_d(3)='1' then ds3_reg <= out_port; end if;
         if en_d(5)='1' then m_src0_reg <= out_port; end if;
         if en_d(6)='1' then m_src1_reg <= out_port; end if;
      end if;
   end process;
   -- decoding circuit for enable signals
```

```
process(port_id,write_strobe)
begin
   en_d <= (others=>'0');
   if write_strobe='1' then
      case port_id(2 downto 0) is
         when "000" => en_d <="0000001";
         when "001" => en_d <="0000010";
         when "010" => en_d <="0000100";
         when "011" => en_d <="0001000";
         when "100" => en_d <="0010000";
         when "101" => en_d <="0100000";
         when others => en_d <="1000000";
      end case;
   end if;
end process;
wr_uart <= en_d(4);
-- =====================================================
--  input interface
-- =====================================================
--    input port id
--      0x00: flag
--      0x01: switch
--      0x02: uart_rx_fifo
--      0x03: prod lower byte
--      0x04: prod upper byte
-- =====================================================
-- input register (for flags)
process(clk)
begin
   if (clk'event and clk='1') then
      btnc_flag_reg <= btnc_flag_next;
      btns_flag_reg <= btns_flag_next;
   end if;
end process;

btnc_flag_next <= '1' when set_btnc_flag='1' else
                  '0' when clr_btn_flag='1' else
                   btnc_flag_reg;
btns_flag_next <= '1' when set_btns_flag='1' else
                  '0' when clr_btn_flag='1' else
                   btns_flag_reg;
-- decoding circuit for clear signals
clr_btn_flag <='1' when read_strobe='1' and
                        port_id(2 downto 0)="000" else
               '0';
rd_uart <= '1' when read_strobe='1' and
                    port_id(2 downto 0)="010" else
           '0';
-- input multiplexing
rx_not_empty <= not rx_empty;
process(port_id,tx_full,rx_not_empty,
      btns_flag_reg,btnc_flag_reg,sw,rx_char,prod)
begin
```

```
      case port_id(2 downto 0) is
         when "000" =>
            in_port <= "0000" & tx_full & rx_not_empty &
                       btns_flag_reg & btnc_flag_reg;
         when "001" =>
            in_port <= sw;
         when "010" =>
            in_port <= rx_char;
         when "011" =>
            in_port <=prod(7 downto 0);
         when others =>
            in_port <= prod(15 downto 8);
      end case;
   end process;
end arch;
```

16.6 BIBLIOGRAPHIC NOTES

The basic bibliographic information for this chapter is similar to that for Chapter 14. The downloaded kcpsm file contains a comprehensive UART and timer design example. The Xilinx Web site has pages for "PicoBlaze Forum" and "PicoBlaze User Resources," where additional PicoBlaze examples are available.

16.7 SUGGESTED EXPERIMENTS

16.7.1 Low-frequency counter I

An accurate low-frequency counter is discussed in Section 6.3.5. We can treat the period counter, division circuit, and binary-to-BCD conversion circuit as three I/O modules, and replace the top-level FSM with PicoBlaze. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.2 Low-frequency counter II

We can reduce the hardware of the frequency counter of Experiment 16.7.1 by replacing the division circuit and binary-to-BCD conversion circuit with software subroutines. Redesign the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.3 Auto-scaled low-frequency counter

An auto-scaled low-frequency counter is discussed in Experiment 6.5.5. We can use PicoBlaze to perform all non-time-critical functions. Redesign the circuit with PicoBlaze and minimal external hardware. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.4 Basic reaction timer with a software timer

The reaction timer is discussed in Experiment 6.5.6. We can redesign the circuit using PicoBlaze. One task of the design is to keep track of the elapsed time interval. This can be done by a software counting routine. Recall that a 50-MHz clock is used on the prototyping board and each instruction takes two clock cycles. We can create a counting loop to record the number of instructions executed and derive the time interval accordingly. Since the interval is at least in the millisecond range, multiple registers are needed for this purpose. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.5 Basic reaction timer with a hardware timer

We can repeat Experiment 16.7.4 with a customized hardware timer. The timer should be treated as an I/O peripheral. PicoBlaze can output a command to clear, start, or pause the timer, and can input the counter's content. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.6 Enhanced reaction timer

An enhanced reaction timer keeps track of the last four response times and the fastest response time, and displays the data on Windows HyperTerminal. We can design a console similar to that of Section 16.5. There should be three commands:

- `c`: clears all data
- `f`: displays the fastest response
- `r`: displays the time of the last four responses
- All other characters: displays "error"

Expand the design in Experiment 16.7.4 or 16.7.5 to include this feature. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.7 Small-screen mouse scribble circuit

A small-screen mouse scribble circuit is discussed in Experiment 12.7.10. We can use PicoBlaze to monitor the activities of the mouse and update the video memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.8 Full-screen mouse scribble circuit

A full-screen mouse scribble circuit is discussed in Experiment 12.7.11. We can use PicoBlaze to monitor the activities of the mouse and update the video memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.9 Enhanced rotating banner

A VGA rotating banner circuit is discussed in Experiment 13.6.1. Instead of a fixed message, we can enhance this circuit by using a keyboard to enter the message dynamically. Assume that the message buffer is 20 characters long and its characters are updated in a first-in-first-out fashion. Redesign the circuit with PicoBlaze. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.10 Pong game

The complete pong game is discussed in Section 13.4. Some functions of the design can be implemented by PicoBlaze:

- Top-level control FSM
- Top-level two-second timer and two-digit decade counter
- The circuit that updates the paddle position, ball position, and ball velocities in Listing 12.5

Modify the original circuit, design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

16.7.11 Text editor

A UART terminal is discussed in Experiment 13.6.5. We can use PicoBlaze to obtain data and commands from the UART and update the tile memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.