CHAPTER 14

PICOBLAZE OVERVIEW

14.1 INTRODUCTION

The *PicoBlaze* processor is a compact 8-bit microcontroller core for Xilinx FPGA devices. It is provided as a cell-level HDL description (which is known as *soft core*) and can be synthesized along with other logic. PicoBlaze is optimized for efficiency and occupies only about 200 logic cells, which amount to less than 5% resource of a 3S200 device. While not intended as a high-performance processor, it is compact and flexible and can be used for simple data processing and control, particularly for non-time-critical "house-keeping" and I/O operations. The PicoBlaze processor can be easily integrated into a larger system and adds another dimension of flexibility in an FPGA-based design.

Although the detailed coverage of assembly language programming and microcontrollers is beyond the scope of this book, this part provides a comprehensive overview of PicoBlaze's organization and instruction set, and illustrates the general assembly program development and I/O interface through a set of examples. We review PicoBlaze's organization and instruction set in this chapter, introduce assembly language programming in Chapter 15, and discuss the general I/O interface and interrupt interface in Chapters 16 and 17.

FPGA Prototyping by VHDL Examples. By Pong P. Chu

14.2 CUSTOMIZED HARDWARE AND CUSTOMIZED SOFTWARE

14.2.1 From special-purpose FSMD to general-purpose microcontroller

The RT-level design and FSMD discussed in Chapter 6 provide a general methodology to convert a sequential algorithm to customized hardware. The rearranged block diagram is shown in Figure 14.1(a). In an FSMD, all components, including the number of registers, the routing of registers' input and output, the number and types of functional units, and the control FSM, are tailored to the target application. The data path may contain multiple function units and multiple routing paths, as shown in the diagram.

An alternative is to keep the same hardware but use *customized software* for different applications. The transformation can be done as follows. First, we can replace the customized data path with a fixed configuration, as shown in the top of Figure 14.1(b). The data registers and customized routing networks are replaced by a register file, which has a fixed number of registers and contains only two read ports and one write port. The customized function units are replaced with an *ALU* (arithmetic and logic unit), which can only perform a set of predefined functions. The data path now can perform RT operations in the following format only:

$$\texttt{rd} \leftarrow \texttt{r1 op r2}$$

where `r1`, `r2`, and `rd` are the addresses of two source registers and one destination register, and `op` is one of the available ALU functions.

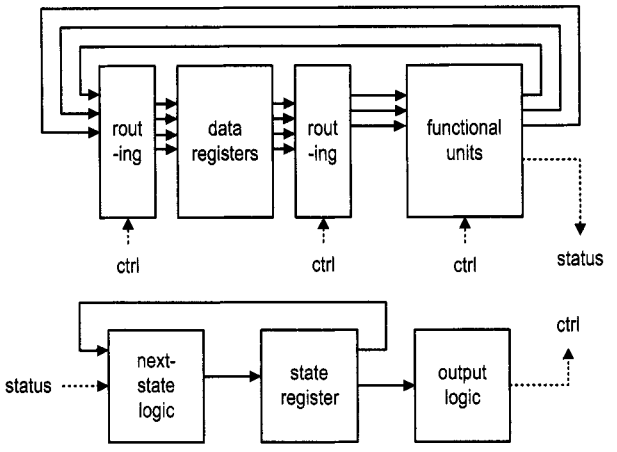
Second, we can replace the customized FSM with a *programmable state machine*, as shown in the bottom of Figure 14.1(b). Recall that operation of an FSM consists of three parts:

- The state register keeps track of the current state.
- The output logic activates certain output signals according to the current state.
- The next-state logic determines the new state.

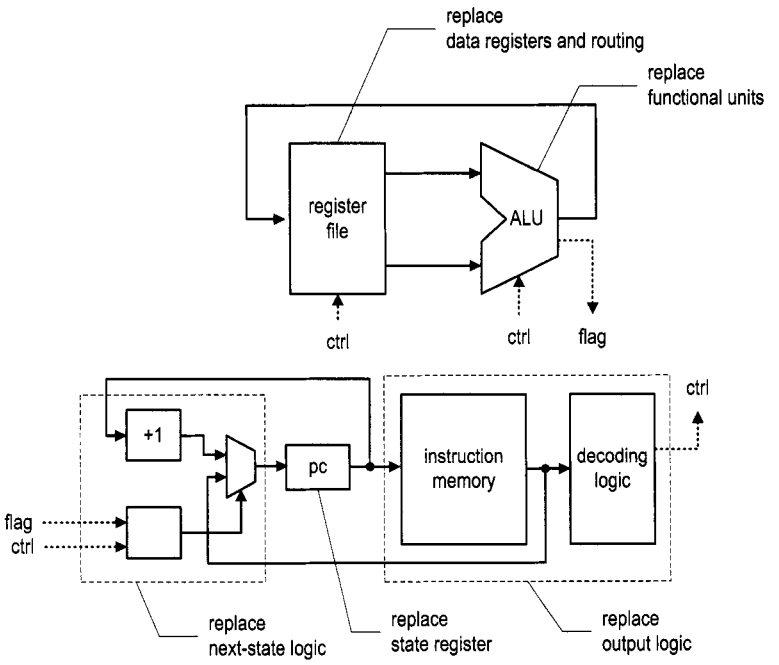
The programmable state machine modifies these operations as follows:

- It replaces the state register with the *program counter*. The content of the program counter represents the current state of the control path.
- In an FSM, each state activates certain output signals to control operation of the data path. The programmable state machine encodes these output patterns into *instructions* and stores them in a memory module, known as *program memory* or *instruction memory*. A memory address corresponds to a state (i.e., a value) of the program counter. During execution, the instruction pointed by the program counter is retrieved from the memory and decoded to generate the control signals. The instruction memory and decoding logic function as a sophisticated output logic circuit.
- In an FSM, there is no limitation on where to go next. From a given state, the FSM can check the input condition and move to one of many possible next states. In a programmable state machine, the next state is usually the value of the current state plus 1 (i.e., the program counter is incremented by 1), which reflects the nature of the sequential execution. The sequential execution may be altered only by several special instructions, such as a *jump* instruction, in which the program counter is loaded with a different value. The incrementor and the associated multiplexing logic function as a simple next-state logic circuit.

After we replace the data path with a register file and an ALU and replace the dedicated FSM with a programmable state machine, customizing the system corresponds to developing a new sequence of instructions (i.e., develop a *software program*) and loads the instructions



(a) Block diagram of an FSMD



(b) Simplified block diagram of a microcontroller

Figure 14.1 Diagrams of an FSMD and a microcontroller.

to the instruction memory. The organization of the FSMD is now the same for different applications and becomes a *general-purpose* hardware platform. The platform constitutes the basic skeleton of the PicoBlaze microcontroller.

14.2.2 Application of microcontroller

In a customized FSMD, the data path can be created to accommodate an individual application's needs. It may contain multiple customized functional units and parallel routing paths, and can complete complex computation in a single state (i.e., one clock cycle). On the other hand, the PicoBlaze microcontroller can only perform one predefined RT operation (i.e., an instruction) at a time. It may need many instructions to perform the same task and thus require much more time.

Many tasks can be done by either a customized FSMD or a microcontroller. The trade-off is between the hardware complexity, performance and ease of development. There is no exact rule on which one to choose. Because developing software is usually easier than creating customized hardware, the microcontroller option is generally preferable for non-time-critical applications. We can determine the feasibility of this option by examining the computation complexity. PicoBlaze requires two clock cycles to complete an instruction. If the system clock is 50 MHz, 25 million instructions can be perform in one second. For a task (or a collection of tasks), we can examine how frequent a request is issued and how fast the task must be completed, and then estimate the number of available instructions. For example, assume that a keyboard interface generates a new input data every 1 ms and the data must be processed within this interval. Within the 1-ms period, PicoBlaze can complete 25,000 instructions. The PicoBlaze controller will be a viable option if the required processing can be done by using less than 25,000 instructions. In general, the microcontroller is suitable for many non-time-critical I/O-interface or "house-keeping" tasks.

14.3 OVERVIEW OF PICOBLAZE

14.3.1 Basic organization

PicoBlaze is a compact 8-bit microcontroller with the following characteristics:

- 8-bit data width
- 8-bit ALU with the carry and zero flags
- 16 8-bit general-purpose registers
- 64-byte data memory
- 18-bit instruction width
- 10-bit instruction address, which supports a program up to 1024 instructions
- 31-word call/return stack
- 256 input ports and 256 output ports
- 2 clock cycles per instruction
- 5 clock cycles for interrupt handling

PicoBlaze is based on the skeleton described in Figure 14.1(b) and adds several enhancements to make it more versatile. The expanded diagram is shown in Figure 14.2. To reduce clutter, only the main data flow is shown. The sizes of main storage components are listed in round brackets. The processor makes several enhancements over the original skeleton:

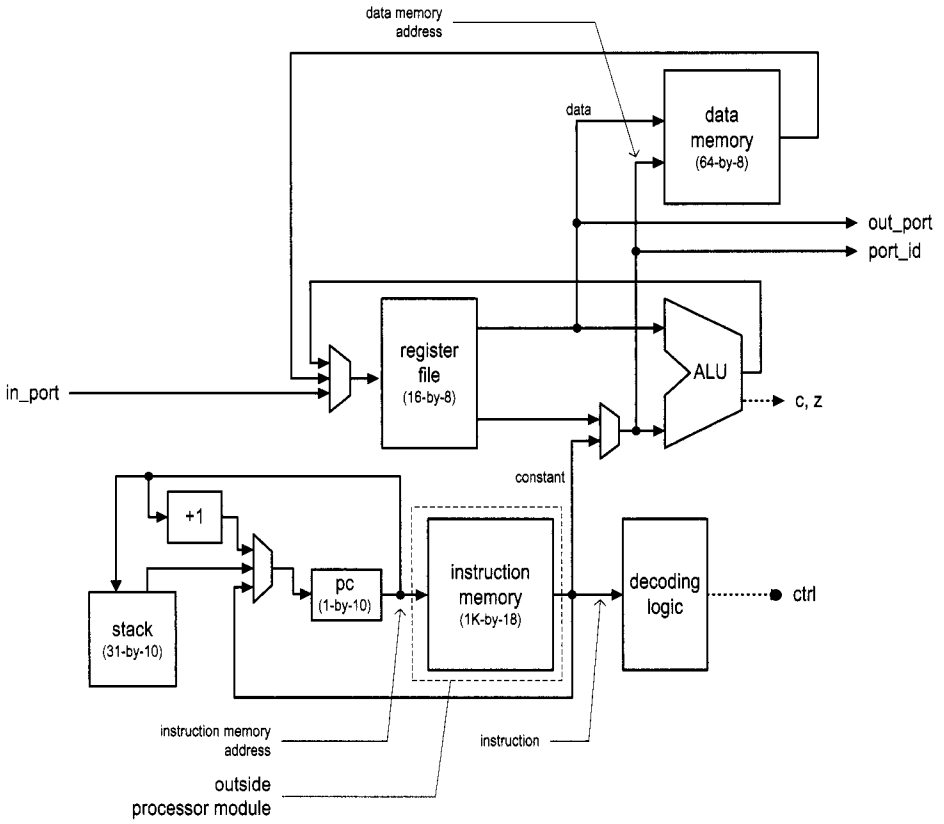


Figure 14.2 Block diagram of PicoBlaze.

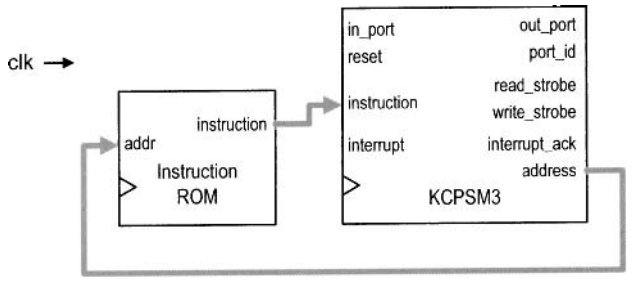


Figure 14.3 Top-level diagram of PicoBlaze.

- *Add a 64-word data memory.* It is known as *scratch RAM* in Xilinx literature but we call it *data RAM*. The data RAM can be considered as a reservoir to store additional data. Note that there is no direct path between the data RAM and ALU. Data must be fetched to a register for processing and then stored back to the data RAM.
- *Add an immediate constant field in some instructions.* This allows a constant, rather than the content of a register, to be used in ALU and other operations. The two-to-one multiplexer before the ALU's bottom input is used to select the register output or the constant field.
- *Add a 31-word stack to support the call/return functions.* We discuss the call and return procedure in more detail in Section 14.5.8.
- *Add paths to input and output external data.* An 8-bit `port_id` signal is used to identify a port and thus up to 256 input ports and 256 output ports can be supported. The I/O interface is discussed in detail in Chapter 16.
- *Add an interrupt handling circuit* (not shown in the diagram). The interrupt mechanism is discussed in detail in Chapter 17.

14.3.2 Top-level HDL modules

During synthesis, a PicoBlaze system is organized as two top-level HDL modules, as shown in Figure 14.3. The `KCPSM3` module is the PicoBlaze processor. `KCPSM3`, which stands for *constant (K) coded programmable state machine*, reflects the original name of the PicoBlaze processor. It has following input and output signals:

- `clk` (input, 1 bit): system clock signal
- `reset` (input, 1 bit): reset signal
- `address` (output, 10 bits): address of the instruction memory, which specifies the location of the instruction to be retrieved
- `instruction` (input, 18 bits): fetched instruction
- `port_id` (output, 8 bits): address of the input or output port
- `in_port` (input, 8 bits): input data from I/O peripherals
- `read_storbe` (output, 1 bit): strobe associated with the input operation
- `out_port` (output, 8 bits): output data to I/O peripherals
- `write_storbe` (output, 1 bit): strobe associated with the output operation
- `interrupt` (input, 1 bit): interrupt request from I/O peripherals
- `interrupt_ack` (output, 1 bit): interrupt acknowledgement to I/O peripherals

The second module is for the instruction memory. During the development, we usually store the compiled assembly code to memory in advance and configure it as a ROM in HDL code. It is thus known as an *instruction ROM*.

14.4 DEVELOPMENT FLOW

While developing a system based on a conventional microcontroller, we examine the required functionalities and select a processor with the proper computation capability and adequate I/O interface. Additional chips are frequently needed to perform special functions. One advantage of using a soft-core microcontroller is that we can have both a customized circuit and a microcontroller developed and implemented in the same FPGA device. A large application usually includes many different tasks. In an FPGA platform, we can implement the time-critical tasks in a customized circuit (i.e., "hardware") for performance and realize the remaining house-keeping and low-speed I/O functions in a microcontroller (i.e., "software").

The basic PicoBlaze-based development flow is shown in Figure 14.4. It consists of the following steps:

1. Determine the software–hardware partition.
2. Develop the assembly program for the software portion.
3. Compile the assembly program to generate an instruction ROM. The ROM is an HDL file.
4. Perform instruction-set-level simulation.
5. Derive HDL code for the hardware portion. The hardware includes customized circuits to perform special I/O and time-critical functions and customized circuits to interface with PicoBlaze.
6. Create the top-level HDL code that combines the codes for the PicoBlaze core, the instruction ROM, and customized hardware.
7. Develop a testbench and perform HDL simulation for the entire system.
8. Synthesize and implement the HDL code and program the FPGA chip on the prototyping board.

The subsequent chapters explain these steps in detail.

The step 9 shown in the dotted line is not a part of the normal development flow. It reloads the instruction memory after the entire system is synthesized. This step is discussed in Section 15.5.3.

14.5 INSTRUCTION SET

PicoBlaze has 57 instructions. The instructions have five general formats. We organize the instructions according to the nature of their operations and divide them into following categories:

- Logical instructions
- Arithmetic instructions
- Compare and test instructions
- Shift and rotate instructions
- Data movement instructions
- Program flow control instructions
- Interrupt related instructions

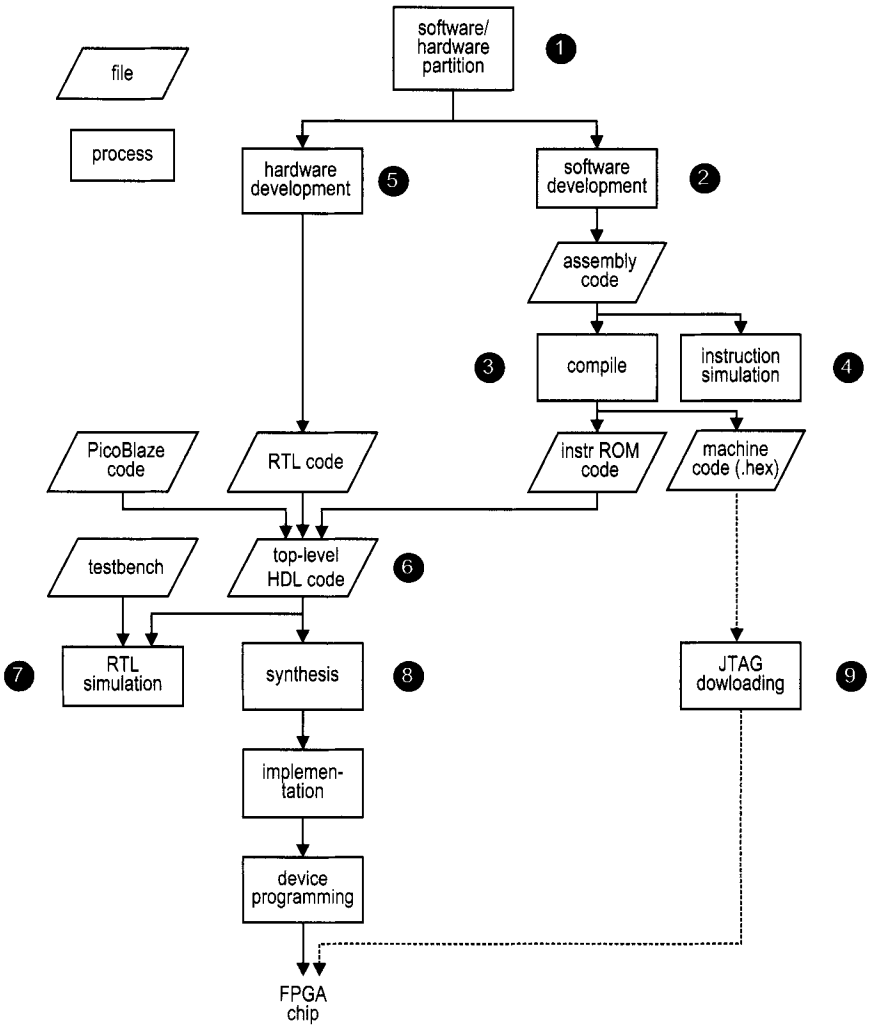


Figure 14.4 Development flow of a system with PicoBlaze.

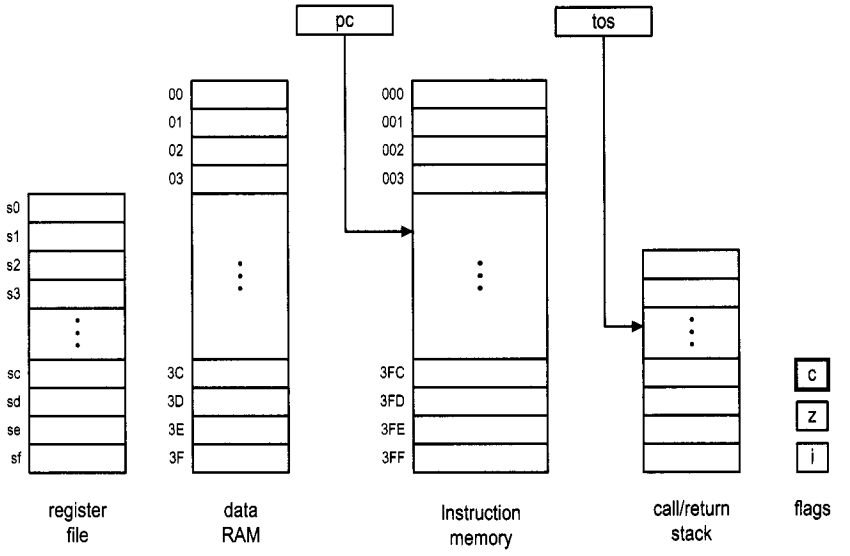


Figure 14.5 PicoBlaze programming model.

In this section, we first examine the program model and instruction format, and then list and explain each instruction.

14.5.1 Programming model

From an assembly programming point of view, PicoBlaze contains 16 8-bit registers, a 64-byte data RAM, three flags (for zero, carry and interrupt), the program counter and the top-of-stack pointer. The model, sometimes known as the instruction set architecture, is shown in Figure 14.5. After an instruction is executed, the contents of these components are modified explicitly or implicitly. The operations associated with each instruction are discussed in Section 14.5.3.

We use the following notations for these memory components and some constant definitions:

- `sX`, `sY`: each representing one of the 16 general-purpose registers, where `X` and `Y` take on hexadecimal values from `0` to `f`
- `pc`: program counter
- `tos`: top-of-stack pointer of the call/return stack
- `c`, `z`, `i`: carry, zero, and interrupt flags
- `KK`: 8-bit constant value or port id, which is usually expressed as two hexadecimal digits
- `SS`: 6-bit constant data memory address, which is usually expressed as two hexadecimal digits
- `AAA`: 10-bit constant instruction memory address, which is usually expressed as three hexadecimal digits

14.5.2 Instruction format

In an assembly program, we generally follow the conventions used in our HDL code, in which a keyword (an instruction mnemonic) is in a boldface font and a constant is in capital letters. PicoBalze's instructions have five formats:

- **op** sX, sY: *register-register format.* The **op** term specifies the operation. The sX and sY terms are the two operands and sX also serves as the destination register. It performs the sX ← sX **op** sY operation.
- **op** sX, KK: *register-constant format.* This format is similar to the register-register format except that the second operand is replaced by an immediate constant. It performs the sX ← sX **op** KK operation.
- **op** sX: *single-register format.* This format is used in shift and rotate instructions, which involve only one operand. It performs the sX ← **op** sX operation.
- **op** AAA: *single-address format.* This format is used in jump and call instructions. The AAA term is an address of the instruction memory. If the specified condition is met, AAA is loaded into the program counter.
- **op**: *zero-operand format.* This format is used in some miscellaneous instructions that do not involve any operand.

There are two assembler programs for PicoBlaze: *KCPSM3* from Xilinx and *PBlazeIDE* from Mediatronix. The two programs use different mnemonics for several instructions. In the following subsections, the alternative mnemonics used in PBlazeIDE are shown in round brackets.

14.5.3 Logical instructions

There are six logical instructions, which support the and, or, and xor operations. An instruction performs bitwise logical operation between two registers or one register and a constant. The carry flag, c, is always cleared. The zero flag, z, reflects the result of the operation. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **and** sX, sY
  - bitwise and operation
  - pseudo operation:

```
sX ← sX and sY;
c ← 0;
```

- **and** sX, KK
  - bitwise and operation
  - pseudo operation:

```
sX ← sX and KK;
c ← 0;
```

- **or** sX, sY
  - bitwise or operation
  - pseudo operation:

```
sX ← sX or sY;
c ← 0;
```

- **or** sX, KK
  - bitwise or operation

  - pseudo operation:

```
sX ← sX or KK;
c ← 0;
```

- **xor** sX, sY
  - bitwise xor operation
  - pseudo operation:

```
sX ← sX xor sY;
c ← 0;
```

- **xor** sX, KK
  - bitwise xor operation
  - pseudo operation:

```
sX ← sX xor KK;
c ← 0;
```

14.5.4 Arithmetic instructions

There are eight arithmetic instructions, which support addition and subtraction with or without the carry flag. The carry flag, c, and the zero flag, z, reflect the result of operation. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **add** sX, sY
  - add without the carry flag
  - pseudo operation:

```
sX ← sX + sY;
```

- **add** sX, KK
  - add without the carry flag
  - pseudo operation:

```
sX ← sX + KK;
```

- **addcy** sX, sY (**addc** sX, sY)
  - add with the carry flag
  - pseudo operation:

```
sX ← sX + sY + c;
```

- **addcy** sX, KK (**addc** sX, KK)
  - add with the carry flag
  - pseudo operation:

```
sX ← sX + KK + c;
```

- **sub** sX, sY
  - subtract without the carry flag
  - pseudo operation:

```
sX ← sX - sY;
```

- **sub** sX, KK
  - subtract without the carry flag
  - pseudo operation:

```
sX ← sX - KK;
```

- **subcy** sX, sY (**subc** sX, sY)
  - subtract with the carry flag (flag functioning as a borrow bit)
  - pseudo operation:

```
sX ← sX - sY - c;
```

- **subcy** sX, KK (**subc** sX, KK)
  - subtract with the carry flag (flag functioning as a borrow bit)
  - pseudo operation:

```
sX ← sX - KK - c;
```

14.5.5 Compare and test instructions

The compare and test instructions examine two registers or one register and constant, and set the carry and zero flags accordingly. The contents of the registers remain intact. These instructions are usually used in conjunction with a conditional jump or call instruction, whose operation is based on the values of the flags.

A compare instruction performs subtraction operation. The result is used to set the carry and zero flags and not stored to any register. The mnemonics, brief descriptions, and pseudo operations of the two instructions are:

- **compare** sX, sY (**comp** sX, sY)
  - compare two registers and set the flags
  - pseudo operation:

```
if sX=sY then z ← 1 else z ← 0;
if sY>sX then c ← 1 else c ← 0;
```

- **compare** sX, KK (**comp** sX, KK)
  - compare a register and a constant and set the flags
  - pseudo operation:

```
if sX=KK then z ← 1 else z ← 0;
if KK>sX then c ← 1 else c ← 0;
```

A test instruction performs an and operation. The result is used to set the flags and not stored in any register. If the result is 0, the zero flag is set to 1. The result is also fed to an eight-input xor circuit to obtain the odd parity. If there are odd number of 1's in the result, the carry flag is set to 1. The mnemonics, brief descriptions, and pseudo operations of the two instructions are shown below. The t is the 8-bit temporary result and will be discarded.

- **test** sX, sY
  - test two registers and set the flags
  - pseudo operation:

```
t ← sX and sY;
if t=0 then z ← 1 else z ← 0;
c ← t(7) xor t(6) xor ··· xor t(0);
```

- **test** sX, KK
  - test a register and a constant and set the flags
  - pseudo operation:

```
t ← sX and KK;
if t=0 then z ← 1 else z ← 0;
c ← t(7) xor t(6) xor ··· xor t(0);
```

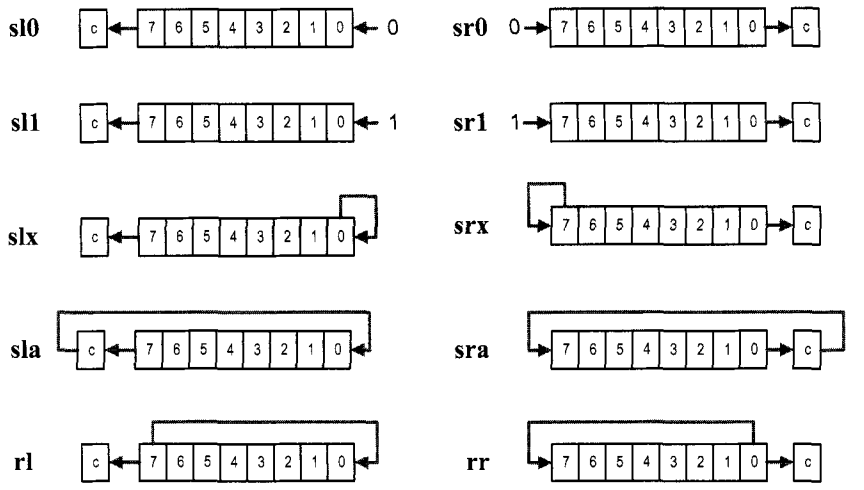


Figure 14.6 Illustration of shift and rotate instructions.

14.5.6 Shift and rotate instructions

There are four shift-left instructions, four shift-right instructions, and two rotate instructions. These instructions use the single-register format and have only one operand. The graphical representations of these instructions are shown in Figure 14.6. The mnemonics, brief descriptions, and pseudo operations of these instructions are shown below. The & symbol means to concatenate two operands.

- **sl0** sX
  - shift a register left 1 bit and shift 0 into the LSB
  - pseudo operation:

```
sX ← sX(6..0) & 0;
c ← sX(7);
```

- **sl1** sX
  - shift a register left 1 bit and shift 1 into the LSB
  - pseudo operation:

```
sX ← sX(6..0) & 1;
c ← sX(7);
```

- **slx** sX
  - shift a register left 1 bit and shift sX(0) into the LSB
  - pseudo operation:

```
sX ← sX(6..0) & sX(0);
c ← sX(7);
```

- **sla** sX
  - shift a register left 1 bit and shift c into the LSB
  - pseudo operation:

```
sX ← sX(6..0) & c;
c ← sX(7);
```

- **sr0** sX
  - shift a register right 1 bit and shift 0 into the MSB
  - pseudo operation:

```
sX ← 0 & sX(7..1);
c ← sX(0);
```

- **sr1** sX
  - shift a register right 1 bit and shift 1 into the MSB
  - pseudo operation:

```
sX ← 1 & sX(7..1);
c ← sX(0);
```

- **srx** sX
  - shift a register right 1 bit and shift sX(7) into the MSB
  - pseudo operation:

```
sX ← sX(7) & sX(7..1);
c ← sX(0);
```

- **sra** sX
  - shift a register right 1 bit and shift c into the MSB
  - pseudo operation:

```
sX ← c & sX(7..1);
c ← sX(0);
```

- **rl** sX
  - rotate a register left 1 bit
  - pseudo operation:

```
sX ← sX(6..0) & sX(7);
c ← sX(7);
```

- **rr** sX
  - rotate a register right 1 bit
  - pseudo operation:

```
sX ← sX(0) & sX(7..1);
c ← sX(0);
```

14.5.7 Data movement instructions

In PicoBlaze, the computation is done via the registers and ALU. The data RAM supplies additional storage and the I/O ports provide paths to peripherals. There are several instructions to move data between the registers, data RAM, and I/O ports. The instructions can be divided into three categories:

- *Between registers*: the **load** instruction
- *Between a register and data RAM*: the **fetch** and **store** instructions
- *Between a register and an I/O port*: the **input** and **output** instructions

The mnemonics, brief descriptions, and pseudo operations of the data movement instructions are shown below. The RAM[ ] notation represents the content of the data RAM. Note that in some instructions, the *indirect address* notation, as in (sY), is used in mnemonic to emphasize that the content of the sY register is used.

- **load** sX, sY
  - move data between two registers
  - pseudo operation:
    ```
    sX ← sY;
    ```
- **load** sX, KK
  - move a constant to a register
  - pseudo operation:
    ```
    sX ← KK;
    ```
- **fetch** sX, (sY) (**fetch** sX, sY)
  - move data from the data RAM to a register
  - pseudo operation:
    ```
    sX ← RAM[(sY)];
    ```
- **fetch** sX, SS
  - move data from the data RAM to a register
  - pseudo operation:
    ```
    sX ← RAM[SS];
    ```
- **store** sX, (sY) (**store** sX, sY)
  - move data from a register to the data RAM
  - pseudo operation:
    ```
    RAM[(sY)] ← sX;
    ```
- **store** sX, SS
  - move data from a register to the data RAM
  - pseudo operation:
    ```
    RAM[SS] ← sX;
    ```
- **input** sX, (sY) (**in** sX, sY)
  - move data from the input port to a register
  - pseudo operation:
    ```
    port_id ← sY;
    sX ← in_port;
    ```
- **input** sX, KK (**in** sX, KK)
  - move data from the input port to a register
  - pseudo operation:
    ```
    port_id ← KK;
    sX ← in_port;
    ```
- **output** sX, (sY) (**out** sX, sY)
  - move data from a register to the output port
  - pseudo operation:
    ```
    port_id ← sY;
    out_port ← sX;
    ```
- **output** sX, KK (**out** sX, KK)
  - move data from a register to the output port
  - pseudo operation:
    ```
    port_id ← KK;
    out_port ← sX;
    ```

There is no explicit instruction to move data to or from the instruction memory. However, many instructions include a field for an immediate constant. Since the constant is part of the instruction and stored in the instruction memory, it can be considered as data that is implicitly moved from the instruction memory to a register.

14.5.8 Program flow control instructions

In PicoBlaze, the program counter indicates where to fetch the instruction. By default, the execution proceeds to the next address in the instruction memory and the program counter is implicitly incremented (i.e., `pc` ← `pc + 1`). The **jump**, **call** and **return** instructions can explicitly load a value to the program counter and modify the program flow. These instructions can be executed unconditionally or conditionally based on the values of the carry and zero flags.

A **jump** instruction loads new value to the program counter if the corresponding condition is met. The program execution changes the regular flow and branches to the new address. The program flow continues normally after this point. The mnemonics, brief descriptions, and pseudo operations of these instructions are shown below. Recall that `AAA` is for the 10-bit instruction memory address and `pc` is for the program counter.

- **jump** `AAA`
  - unconditionally jump
  - pseudo operation:

    ```
    pc ← AAA;
    ```
- **jump c**, `AAA`
  - jump if the carry flag is set
  - pseudo operation:

    ```
    if c=1 then pc ← AAA else pc ← pc + 1;
    ```
- **jump nc**, `AAA`
  - jump if the carry flag is not set
  - pseudo operation:

    ```
    if c=0 then pc ← AAA else pc ← pc + 1;
    ```
- **jump z**, `AAA`
  - jump if the zero flag is set
  - pseudo operation:

    ```
    if z=1 then pc ← AAA else pc ← pc + 1;
    ```
- **jump nz**, `AAA`
  - jump if the zero flag is not set
  - pseudo operation:

    ```
    if z=0 then pc ← AAA else pc ← pc + 1;
    ```

The **call** and **return** instructions are used to implement a software function. When a function is *called*, the processor suspends the current execution and branches to the corresponding routine. When the routine computation is completed, the processor *returns* to the suspended point and continues the execution. Like a **jump** instruction, a **call** instruction loads a new value to the program counter if the corresponding condition is met. In addition, it also saves the current value of the program counter in a special buffer, known as the *stack*. The new address represents the starting point of a routine. The routine should include a **return** instruction in the end. The **return** instruction obtains the saved value from the

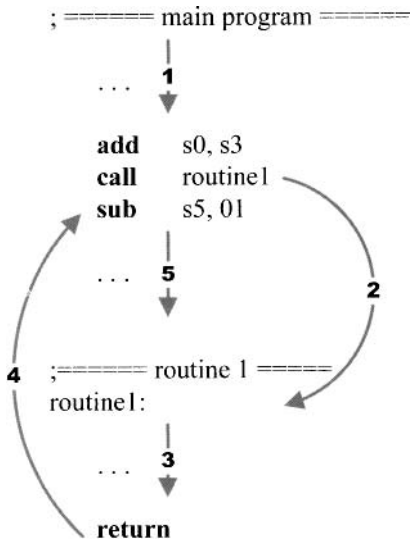


Figure 14.7 Representative flow of a subroutine call.

stack, increments the value by 1, and loads it to the program counter. This allows the execution to return to the instruction that immediately follows the original **call** instruction. A representative program flow is shown in Figure 14.7.

PicoBlaze allows nested function calls, which means that a function can be called within another function. To support this feature, a stack, which is a *last-in-first-out* buffer, is used to store the program counter's values. In this buffer, the address of the newest call is pushed to the top of the stack (i.e., the "last-in"). Assume that this routine does not contain other function call inside. It will be completed first and the saved returned address is on the top of the stack. It should be popped from the stack (i.e., "first-out") to resume the previous execution. PicoBlaze provides a 31-word stack for the nested call and return operations.

The mnemonics, brief descriptions, and pseudo operations of the **call** and **return** instructions are shown below. Recall that and `tos` is for the top-of-stack pointer. The `STACK[ ]` notation represents the content of the stack.

- **call** `AAA`
  - unconditionally call subroutine
  - pseudo operation:

```
tos ← tos + 1;
STACK[tos] ← pc;
pc ← AAA;
```

- **call c**, `AAA`
  - call subroutine if the carry flag is set
  - pseudo operation:

```
if c=1 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
```

```
pc ← pc + 1;
```

- **call nc**, AAA
  - call subroutine if the carry flag is not set
  - pseudo operation:

```
if c=0 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **call z**, AAA
  - call subroutine if the zero flag is set
  - pseudo operation:

```
if z=1 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **call nz**, AAA
  - call subroutine if the zero flag is not set
  - pseudo operation:

```
if z=0 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **return** (**ret**)
  - unconditionally return
  - pseudo operation:

```
pc ← STACK[tos] + 1;
tos ← tos - 1;
```

- **return c** (**ret c**)
  - return if the carry flag is set
  - pseudo operation:

```
if c=1 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

- **return nc** (**ret nc**)
  - return if the carry flag is not set
  - pseudo operation:

```
if c=0 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
```

```
else
   pc ← pc + 1;
```

- **return z (ret z)**
  - return if the zero flag is set
  - pseudo operation:

```
if z=1 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

- **return nz (ret nz)**
  - return if the zero flag is not set
  - pseudo operation:

```
if z=0 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

14.5.9 Interrupt related instructions

Interrupt is another mechanism to alter program execution and its detail is discussed in Chapter 17. Unlike the **jump** and **call** instructions, it is initiated from an external request. When the interrupt flag is enabled and the interrupt request is asserted, PicoBlaze completes execution of the current instruction, saves the address of the next instruction in the call/return stack, preserves the carry and zero flags, disables the interrupt flag, and loads the program counter with 3FF, which is the starting address of the interrupt service routine. PicoBlaze has two return-from-interrupt instructions, which resume the operation from the interrupted location. It also has two instructions that enable and disable the interrupt request by setting or clearing the interrupt flag, `i`. The mnemonics, brief descriptions and pseudo operations of these instructions are:

- **returni disable (reti disable)**
  - return from interrupt service routine and keep the interrupt flag disabled
  - pseudo operation:

```
pc ← STACK[tos];
tos ← tos - 1;
i ← 0;
c ← preserved c;
z ← preserved z;
```

- **returni enable (reti enable)**
  - return from interrupt service routine and keep the interrupt flag enabled
  - pseudo operation:

```
pc ← STACK[tos];
tos ← tos - 1;
i ← 1;
c ← preserved c;
z ← preserved z;
```

- **enable interrupt (eint)**
  - enable interrupt request
  - pseudo operation:
    ```
    i ← 1;
    ```
- **disable interrupt (dint)**
  - disable interrupt request
  - pseudo operation:
    ```
    i ← 0;
    ```

Note that the interrupt mechanism saves the address of the next instruction. When a **returni** instruction is executed, the address saved on the top of the stack (i.e., `STACK[tos]`) is restored. This is different from a regular **return** instruction, in which the incremented address (i.e., `STACK[tos]+1`) is restored.

14.6 ASSEMBLER DIRECTIVES

An *assembler directive* looks like an instruction in an assembly program. However, it is not part of the microcontroller's instruction set but is used to help program development. As its name suggests, a directive "directs" the assembler to perform a specific task, such as defining a constant or reserving data space. The KCPSM3 and PBlazeIDE assemblers have somewhat different directives and they are discussed in the following subsections.

14.6.1 The KCPSM3 directives

The mnemonics, descriptions, and examples of key directives used in the KCPSM3 assembler are:

- **address**
  - The directive specifies the subsequent code to be put to a specific address in the instruction ROM.
  - Example:
    ```
    address  3FF
    ```
- **namereg**
  - The directive gives a symbolic name for a register. It makes code more descriptive.
  - Example:
    ```
    namereg  s5, index
    ```
- **constant**
  - The directive gives a symbolic name for a constant. It makes code more descriptive.
  - Example:
    ```
    constant  max, F0
    ```

14.6.2 The PBlazeIDE directives

The mnemonics, descriptions, and examples of key directives used in the PBlazeIDE assembler are shown below. Note that a $ sign is needed for a number in hexadecimal format.

- **org**
  - The directive specifies the subsequent code to be put to a specific address in the instruction ROM (i.e., "originate" from this address).
  - Example:
```
org   $3FF
```
- **equ**
  - The directive "equates" a symbol to a value or register. It gives a symbolic name for a constant or a register.
  - Example:
```
max     equ   128/8
index   equ   s5
```
- **dsin, dsout, dsio**
  - These directives equate a symbolic name for an I/O port id. The corresponding port can be defined as input, output, or both input and output. The difference between these directives and **equ** is that PBlazeIDE generates "port indicators" for these directives on the simulation screen. The I/O activities can be displayed and simulated via these indicators.
  - Example:
```
keyboard   dsin   $0E
switch     dsin   $0F
led        dsout  $15
```
- **vhdl**
  - This directive generates instruction ROM in VHDL format. The detail is discussed in Chapter 15.
  - Example:
```
vhdl "template.vhd", "target.vhd", "ROM"
```

14.7 BIBLIOGRAPHIC NOTES

The PicoBlaze's manual from Xilinx, *PicoBlaze 8-bit Embedded Microcontroller User Guide*, provides detailed information about this microcontroller, including the hardware organization, instruction set, development process, and the KCPSM3 and PBlazeIDE assemblers. Ken Chapman, the designer of PicoBlaze, describes the derivation of this microcontroller in article "Creating Embedded Microcontrollers," which is available in the *TechXclusives* section of Xilinx Web site.

The KCPSM3 assembler, PicoBlaze HDL code, and instruction ROM HDL template can be downloaded from the Xilinx Web site. Searching with the "PicoBlaze" keyword will lead to the downloading page. The PBlazeIDE assembler can be downloaded from the Mediatronix Web site, `http://www.mediatronix.com`. The site also provides more detailed information about the software.

This Page Intentionally Left Blank

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.

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
```

```
    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
```

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

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

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

;=================================================================
;routine: clr_data_mem
;  function: clear data ram
;  temp register: data, i
;=================================================================
clr_data_mem:
   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
   return

;=================================================================
;routine: read switch
;  function: obtain two nibbles from input
;  input register: sw_in
;  temp register: data
;=================================================================
read_switch:
   input sw_in, sw_port    ;read switch input
```

```
    load data, sw_in
    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

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

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

;=============================================================
;routine: write_led
;  function: output 8 LSBs of result to 8 leds
;  temp register: data
;=============================================================
write_led:
    fetch data, aabb_lsb
    output data, led_port
    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, 07             ;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-out from previous +
    store data, aabb_cout    ;store carry-out 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
```

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:

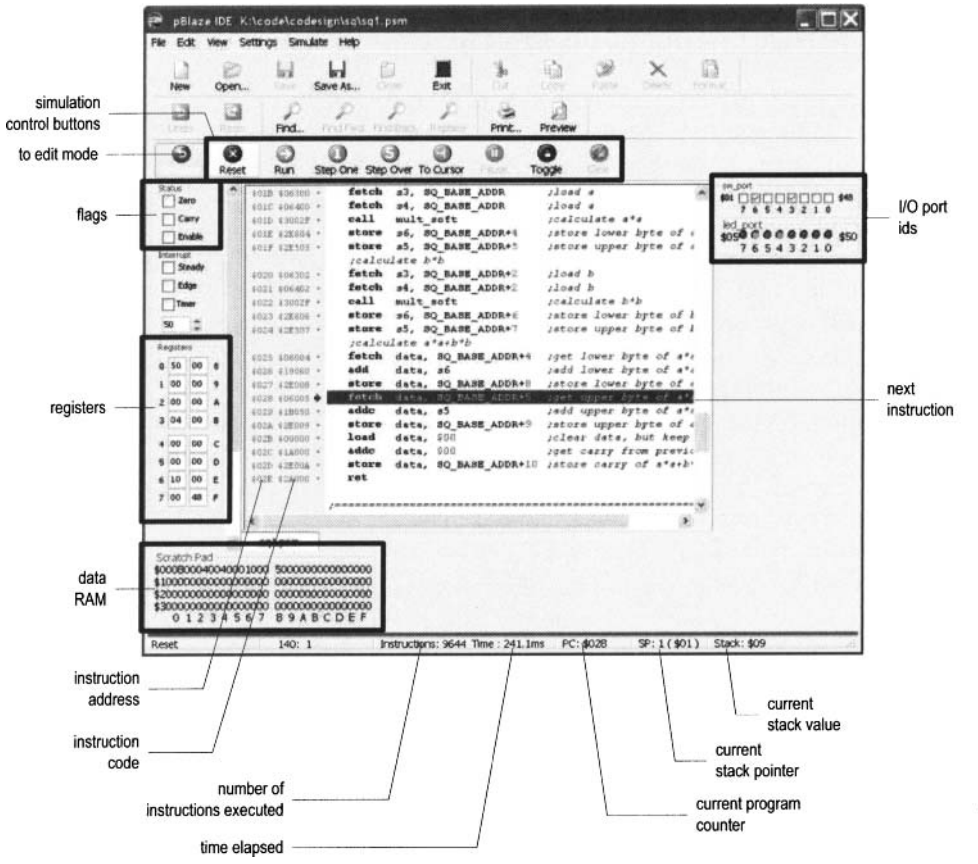


Figure 15.2 Screenshot of pBlazeIDE in simulation mode.

- 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.

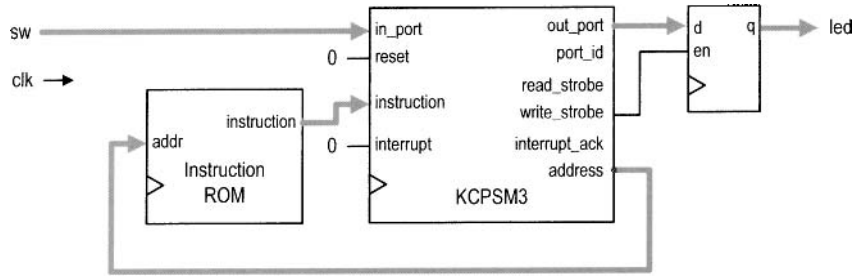


Figure 15.3 PicoBlaze with a simple I/O interface.

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
   port(
      clk, reset: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      led: out std_logic_vector(7 downto 0)
   );
end pico_sio;

architecture arch of pico_sio 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: std_logic;
    -- register signals
    signal led_reg: std_logic_vector(7 downto 0);

begin
    --=============================================================
    --  KCPSM and ROM instantiation
    --=============================================================
    proc_unit: entity work.kcpsm3
        port map(
            clk=>clk, reset=>reset,
            address=>address, instruction=>instruction,
            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
        port map(
             clk => clk, address=>address,
             instruction=>instruction);
    --=============================================================
    --  output interface
    --=============================================================
    --output register
    process (clk)
    begin
        if (clk'event and clk='1') then
            if write_strobe='1' then
                led_reg <= out_port;
            end if;
        end if;
    end process;
    led <= led_reg;
    --=============================================================
    --  input interface
    --=============================================================
    in_port <= sw;
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.

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

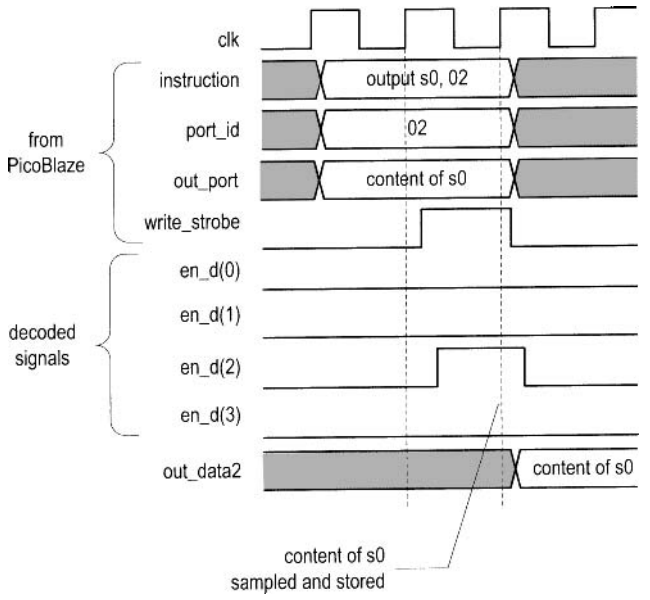


Figure 16.1 Timing diagram of an **output** instruction.

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.

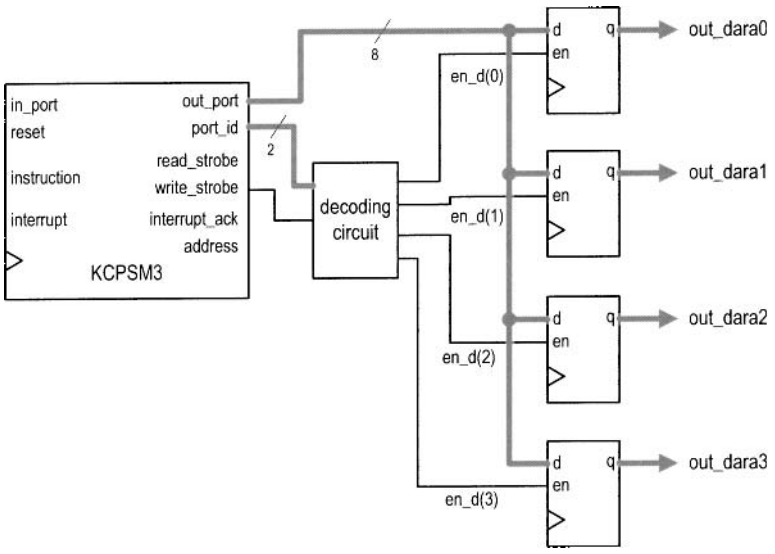


Figure 16.2 Output decoding of four output registers.

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.

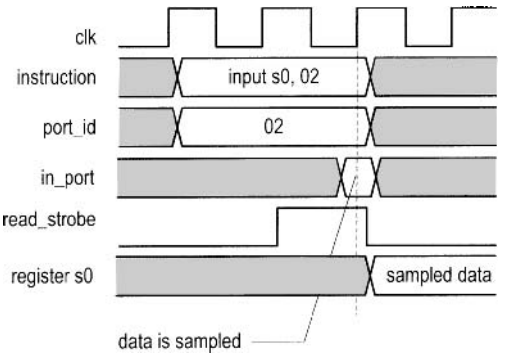


Figure 16.3 Timing diagram of an **input** instruction.

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

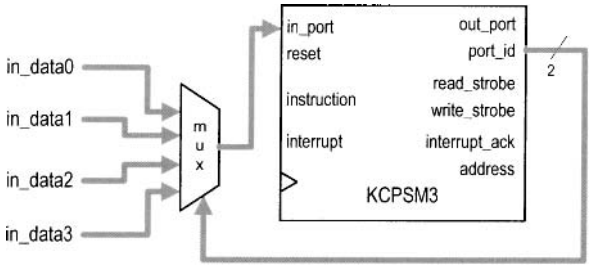


Figure 16.4 Block diagram of four continuous-access ports.

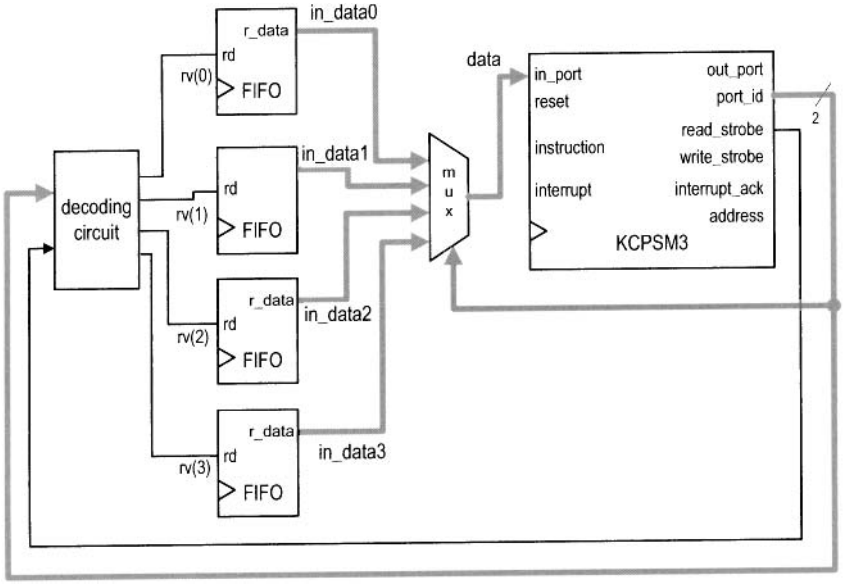


Figure 16.5 Block diagram of four single-access ports.

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

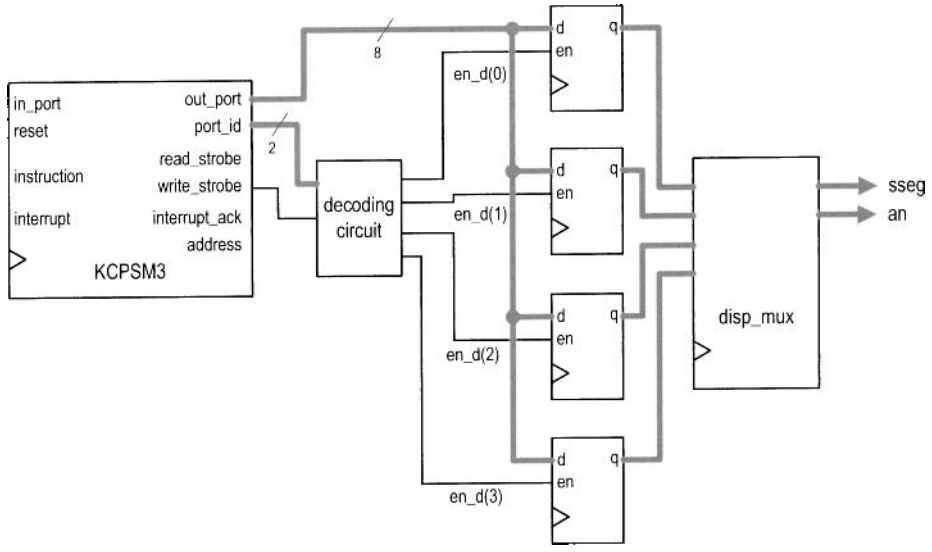


Figure 16.6 Output interface of a square circuit.

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
```

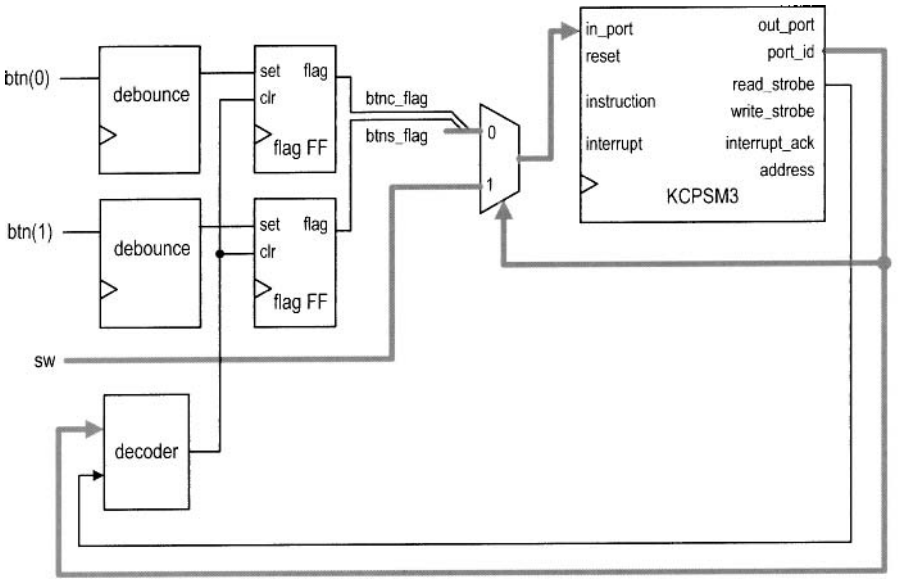


Figure 16.7 Input interface of a square circuit.

```
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
150   ;process byte 0, upper nibble
      fetch data, (addr)
      call get_upper_nibble
      call hex_to_led
      store data, led1
155   ;process byte 1, lower nibble
      add addr, 01              ;get upper addr
      fetch data, (addr)
      call get_lower_nibble
      call hex_to_led
160   store data, led2
      ;process byte 1, upper nibble
      fetch data, (addr)
      call get_upper_nibble
      call hex_to_led
165   ;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
170   test s6, 01               ;carry=1?
      jump z, led_done          ;no
      and data, 7F              ;yes, assert msb (dp) to 0
  led_done:
      store data, led3
175   return

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

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

```
  hex_to_led:
200     compare data, 00
        jump nz, comp_hex_1
        load data, 81           ;7-seg pattern 0
        jump hex_done
  comp_hex_1:
205     compare   data, 01
        jump nz, comp_hex_2
        load data, CF           ;7-seg pattern 1
        jump hex_done
  comp_hex_2:
210     compare data, 02
        jump nz, comp_hex_3
        load data, 92           ;7-seg pattern 2
        jump hex_done
  comp_hex_3:
215     compare data, 03
        jump nz, comp_hex_4
        load data, 86           ;7-seg pattern 3
        jump hex_done
  comp_hex_4:
220     compare data, 04
        jump nz, comp_hex_5
        load data, CC           ;7-seg pattern 4
        jump hex_done
  comp_hex_5:
225     compare data, 05
        jump nz, comp_hex_6
        load data, A4           ;7-seg pattern 5
        jump hex_done
  comp_hex_6:
230     compare data, 06
        jump nz, comp_hex_7
        load data, A0           ;7-seg pattern 6
        jump hex_done
  comp_hex_7:
235     compare data, 07
        jump nz, comp_hex_8
        load data, 8F           ;7-seg pattern 7
        jump hex_done
  comp_hex_8:
240     compare data, 08
        jump nz, comp_hex_9
        load data, 80           ;7-seg pattern 8
        jump hex_done
  comp_hex_9:
245     compare data, 09
        jump nz, comp_hex_a
        load data, 84           ;7-seg pattern 9
        jump hex_done
  comp_hex_a:
250     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

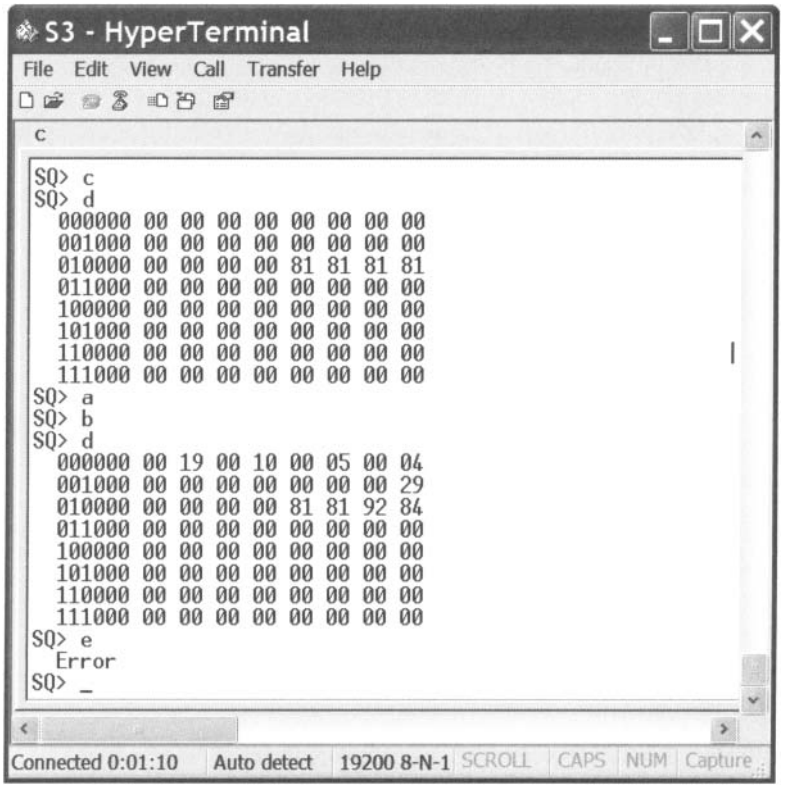


Figure 16.8 Representative console screen.

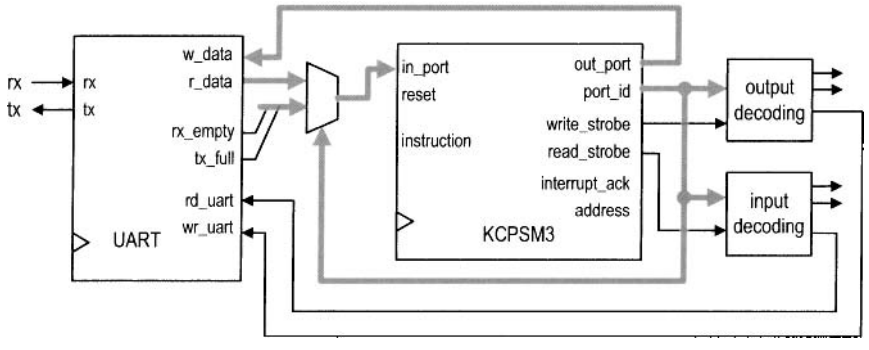


Figure 16.9 UART I/O interface.

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.

CHAPTER 17

PICOBLAZE INTERRUPT INTERFACE

17.1 INTRODUCTION

During normal program execution, a microcontroller *polls* the I/O peripherals (i.e., checks the status signals) and determines the course of action accordingly. An I/O peripheral is passive and waits for its turn. The *interrupt* is a mechanism that allows an external I/O peripheral to initiate the operation. It, as the name shows, interrupts normal program execution and starts a service routine for the I/O peripheral. For a microcontroller, the interrupt is usually reserved for a time-critical peripheral operation, which must be processed immediately. The PicoBlaze microcontroller provides support for simple interrupt-handling capability. In this chapter, we examine the PicoBlaze's interrupt mechanism and use an example to illustrate software and interface development.

17.2 INTERRUPT HANDLING IN PICOBLAZE

Interrupt handling is a coordinated effort between hardware and software. When an external peripheral needs service through interrupt, it asserts the `interrupt` signal of PicoBlaze. If the interrupt service is enabled, PicoBlaze completes execution of the current instruction, activates the `interrupt_ack` signal to acknowledge the acceptance of the interrupt request, and then implicitly executes the **call** 3FF instruction. When the instruction is executed, the current content of the program counter is saved in stack and the 3FF address is loaded to the programmer counter. Note that the 3FF address is the last location in the instruction

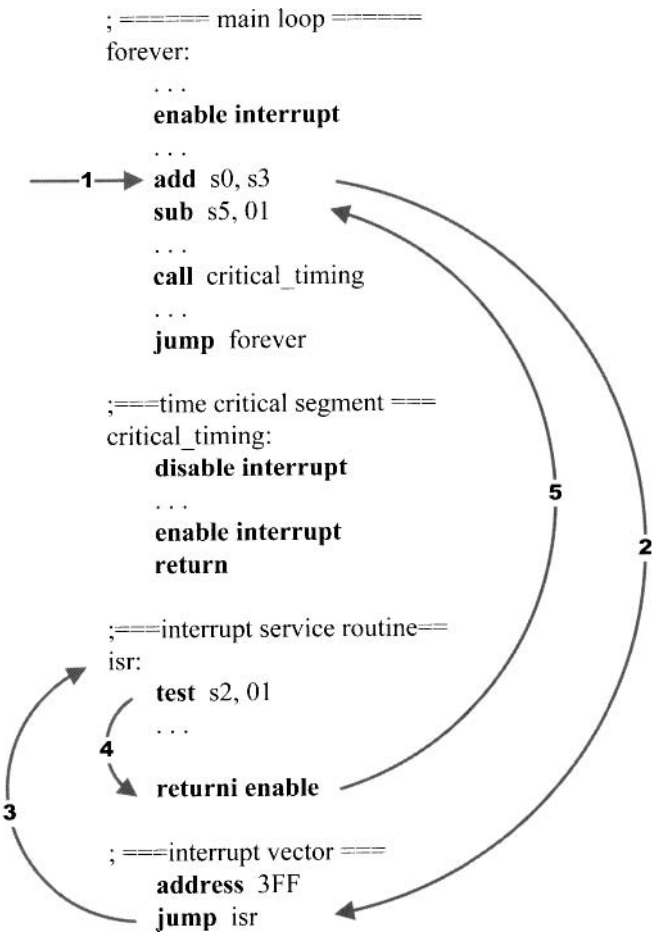


Figure 17.1 Interrupted flow.

memory and serves as the starting point of the interrupt service routine. It usually contains a **jump** instruction, which leads to the body of the service routine. The service should be ended with a **returni** instruction to return to the interrupted point and resume the previous execution.

17.2.1 Software processing

Four instructions are associated with interrupt, as discussed in Section 14.5.9. The **enable interrupt** and **disable interrupt** instructions enable and disable the interrupt request, and the two return-from-interrupt instructions, **returni enable** and **returni disable**, return execution to the interrupted point.

A typical program segment with interrupt service routine is shown in Figure 17.1. It generally consists of the following segments:

- *An initial* **enable interrupt** *instruction*: used to enable the interrupt service. This is needed since the interrupt request is disabled by default.

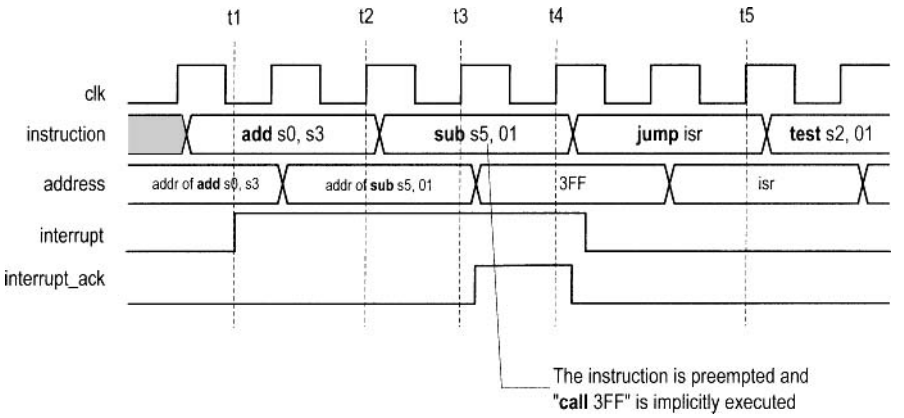


Figure 17.2 Timing diagram of an interrupt event.

- *A* **jump** *instruction in the end of the instruction memory* (i.e., 3FF): leads to the interrupt service routine.
- *Interrupt service routine*: the code that actually performs the requested service. The routine should be ended with a **returni** instruction.

A representative flow of an interrupt event is shown in Figure 17.1. We assume that the external I/O assert the `interrupt` signal in the middle of the **add** `s0,s3` instruction. PicoBlaze performs the following steps in sequence:

1. Completes execution of the current execution.
2. Saves the content of the program counter, clears the interrupt flag, `i`, to zero, preserves the zero and carry flags, and loads the program counter with 3FF.
3. Executes the **jump** `isr` instruction in the 3FF address.
4. Performs the service routine.
5. Executes the **returni** instruction, in which the saved program counter and flags are restored.
6. Resumes the interrupted program and executes the **sub** `s5,01` instruction.

17.2.2 Timing

The detailed timing diagram of the previous interrupt event is shown in Figure 17.2. The basic sequence is:

- At `t1`: The external interrupt interface asserts the `interrupt` signal. PicoBlaze continues the normal operation to complete execution of the current **add** `s0,s3` instruction.
- At `t2`: PicoBlaze recognizes the interrupt and aborts the next instruction (**sub** `s5,01`) and implicitly executes the **call** 3FF instruction.
- At `t3`: PicoBlaze asserts the `interrupt_ack` signal. It also saves the address of the **sub** `s5,01` instruction, preserves the zero and carry flags, and clears the interrupt flag to 0.
- At `t4`: PicoBlaze loads and executes the instruction in address 3FF, **jump** `isr`. The external interrupt interface circuit acknowledges the `interrupt_ack` signal and deasserts the `interrupt` signal.

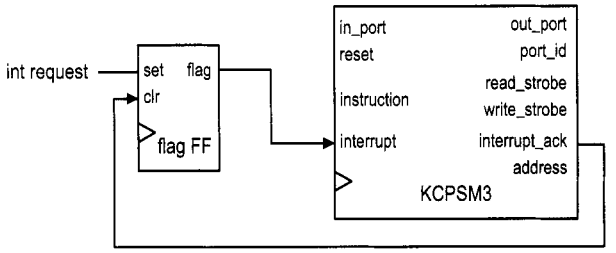


Figure 17.3 Interrupt interface with a single request.

- At `t5`: PicoBlaze starts the interrupt service routine.

Note that it requires up to five clock cycles from the time that the `interrupt` signal is asserted to the time that the first instruction of interrupt service routine is executed.

17.3 EXTERNAL INTERFACE

The nature of the interrupt request is similar to that of a single-access port discussed in Section 16.3.2. After the request is accepted, it must be cleared so that the same request will not be processed multiple times. The flag FF discussed in Section 7.2.4 can be used for this purpose.

17.3.1 Single interrupt request

If there is only one I/O peripheral in a PicoBlaze system that can generate an interrupt request, we just need a single flag FF in the interrupt interface circuit, as shown in Figure 17.3. When the service is required, the external I/O circuit asserted the `int request` signal for one clock cycle, which sets the flag FF to '1' and activates the `interrupt` input of PicoBlaze. If the interrupt is enabled in PicoBlaze, it acknowledges acceptance of the request by asserting the `interrupt_ack` signal for one clock cycle, which clears the flag FF to '0'.

17.3.2 Multiple interrupt requests

Processing a PicoBlaze system with two or more interrupt requests is more involved. The PicoBlaze microcontroller must determine which peripheral issues the request and clear the corresponding flag FF after the request is accepted. This needs the coordination of the hardware interface and the interrupt service routine.

The interrupt interface with two requests is shown in Figure 17.4. The two individual requests, `int request0` and `int request1`, are connected to two flag FFs, and the output signals of the FFs are passed to an or gate to generate the final interrupt request signal. In addition, the two signals are also routed to the input multiplexer. If at least one request is asserted, the `interrupt` signal of PicoBlaze is asserted. When PicoBlaze senses the request, it does not know which peripheral or whether both peripherals issue the request. The interrupt service routine must first input the two request signals and check their values according to the assigned priority, and then perform the corresponding service.