PART III

PICOBLAZE MICROCONTROLLER *XILINX SPECIFIC*

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.

Copyright © 2008 John Wiley & Sons, Inc.

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.

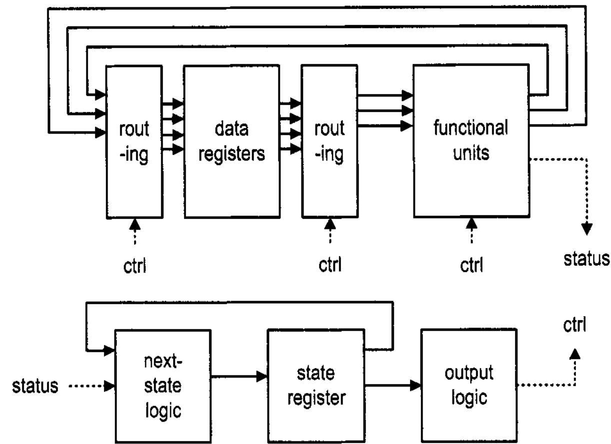
Second, we can replace the customized FSM with a *programmable state machine*, as shown in the bottom of Figure 14.1(b). Recall that operation of an FSM consists of three parts:

- 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.

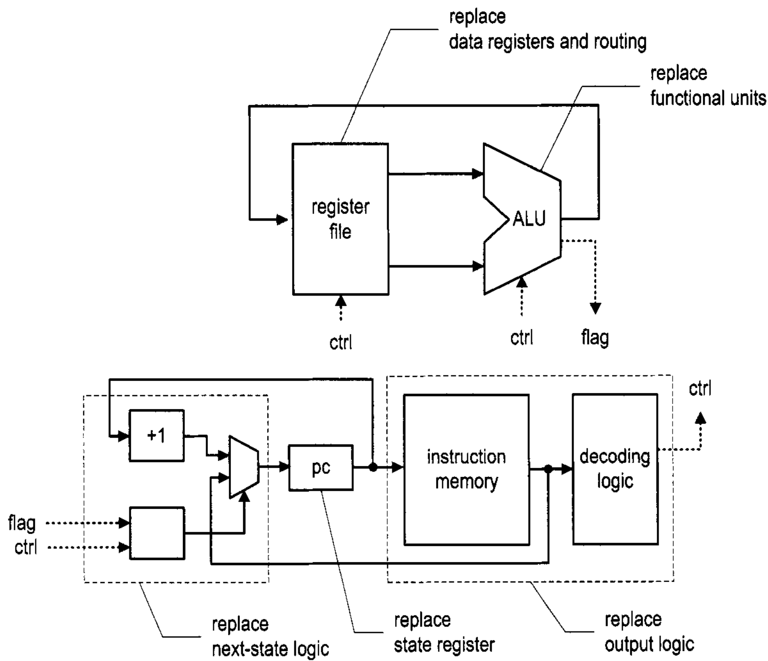
The programmable state machine modifies these operations as follows:

- 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:

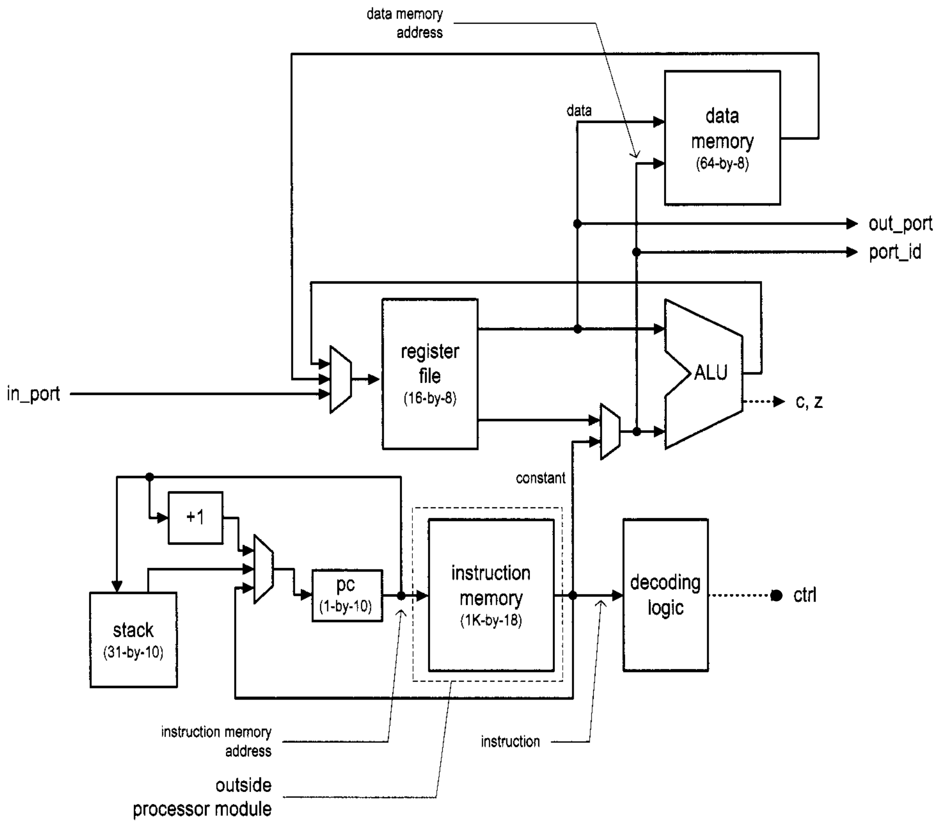


Figure 14.2 Block diagram of PicoBlaze.

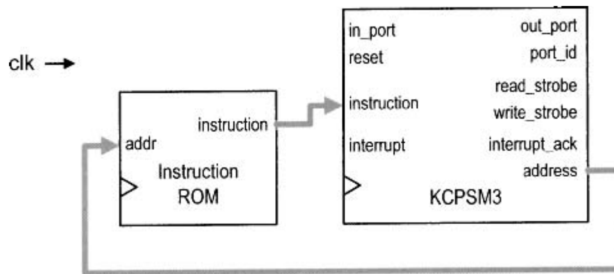


Figure 14.3 Top-level diagram of PicoBlaze.

- *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

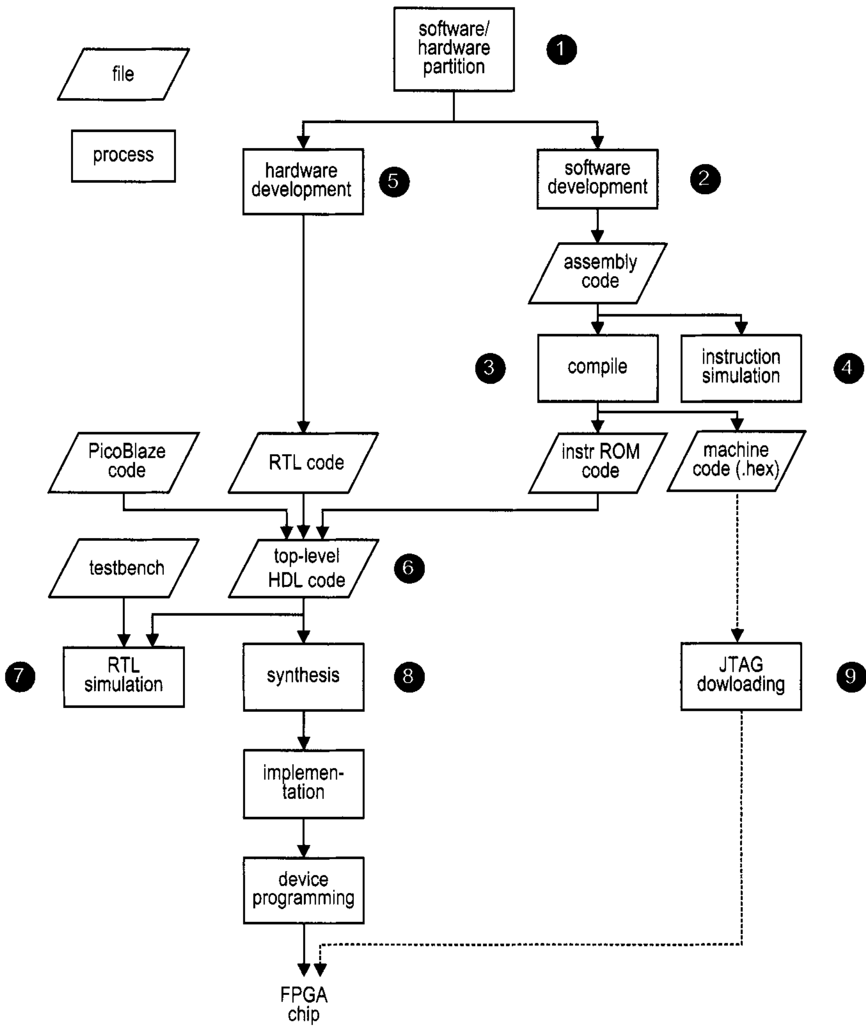


Figure 14.4 Development flow of a system with PicoBlaze.

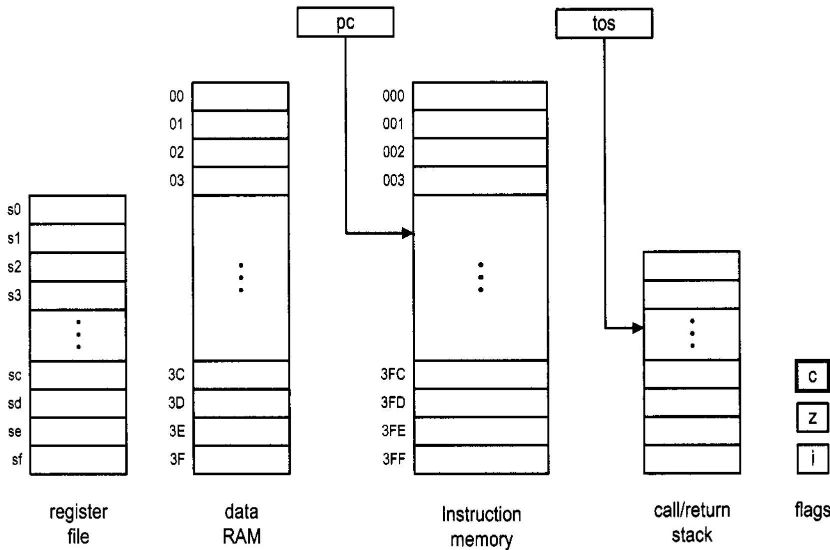


Figure 14.5 PicoBlaze programming model.

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);
    ```

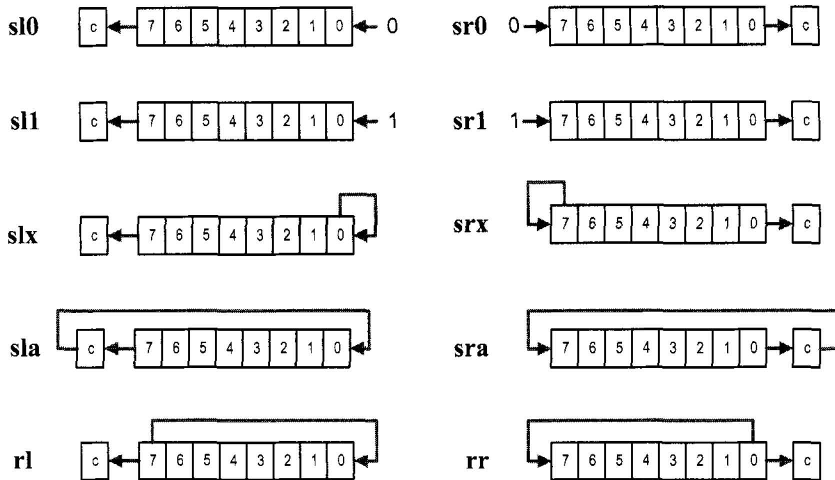


Figure 14.6 Illustration of shift and rotate instructions.

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

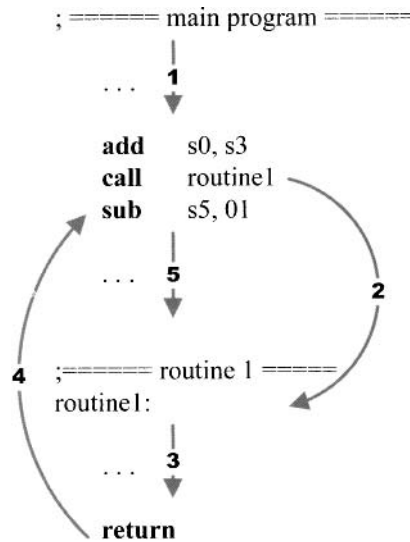


Figure 14.7 Representative flow of a subroutine call.

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.