5

Single-Cycle CPU Design in Verilog HDL

Most computers in use today are synchronous computers. A synchronous computer is one in which all operations are controlled by a clock.

Referring to Figure 5.1, a clock is a periodic signal that oscillates instantaneously between 0 and 1. The transition from 0 to 1 is called a rising edge (or positive edge) and the transition from 1 to 0 is called a falling edge (or negative edge). A clock cycle is defined as the time from one rising edge to the next rising edge.

A single-cycle CPU (central processing unit) executes each instruction in one clock cycle. CPU states (program counter (PC) and registers) are updated at the rising edges of the clock. The cycle time is determined so that the most complex instruction can complete the execution correctly in one clock cycle. Compared to other CPUs, the single-cycle CPU is the most cost-effective but time-consuming CPU.

This chapter describes the design method of the single-cycle CPU and gives the Verilog HDL (hardware description language) code as well as the simulation waveforms.

5.1 The Circuits Required for Executing an Instruction

Instructions are executed by hardware circuits. This section describes what kinds of circuits are required for executing the instructions listed in Table 4.4.

5.1.1 The Circuits Required by Instruction Fetch

Instructions are stored in the memory. CPU must fetch instruction from the memory first. There is a PC in CPU that points to a memory location. The instruction in that location will be executed by the CPU. That is, CPU uses PC as the address to read an instruction from the memory, as shown in Figure 5.2.

If the fetched instruction is neither a branch instruction nor a jump instruction, the PC will be incremented by 4, pointing to the next instruction, because an instruction has 32 bits (4 bytes) while the PC is a byte address. The multiplexer (`mux`) in the figure is used to select the next PC, which will be written to the PC at the rising edge of the clock. The other inputs of the multiplexer will be described later.

5.1.2 The Circuits Required by Instruction Execution

After fetching an instruction, the CPU will execute it. The main components for executing an instruction include an arithmetic logic unit (ALU), a register file, and a control unit.

Computer Principles and Design in Verilog HDL, First Edition. Yamin Li.

Companion Website: www.wiley.com/go/li/verilog

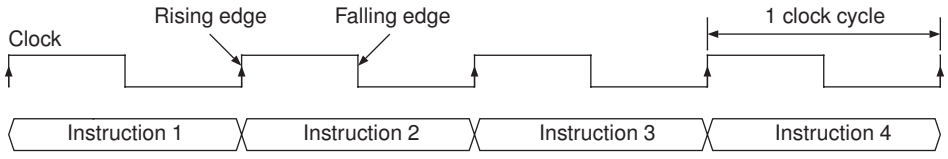

Figure 5.1 Clock and clock cycle

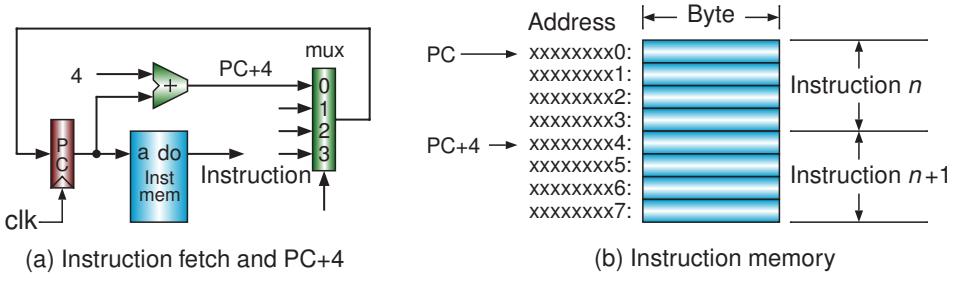

Figure 5.2 Block diagram of instruction fetch

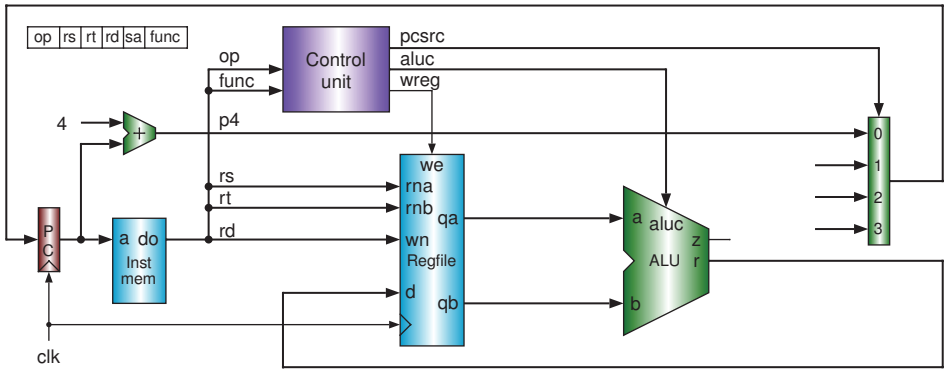

Figure 5.3 Block diagram for R-format arithmetic and logic instructions

5.1.2.1 The Circuits Required by R-Format Instructions

Figure 5.3 shows the circuits required for executing the following instructions:

```
add rd, rs, rt          # rd <-- rs ADD rt
sub rd, rs, rt          # rd <-- rs SUB rt
and rd, rs, rt          # rd <-- rs AND rt
or  rd, rs, rt          # rd <-- rs OR  rt
xor rd, rs, rt          # rd <-- rs XOR rt
```

Two source operands are read from `qa` and `qb` of the register file based on `rs` and `rt`, respectively. These two operands are sent to the inputs of the ALU for calculation. The difference between these five instructions is only on the ALU's operations. A control unit generates the control signals. `pcsrc`

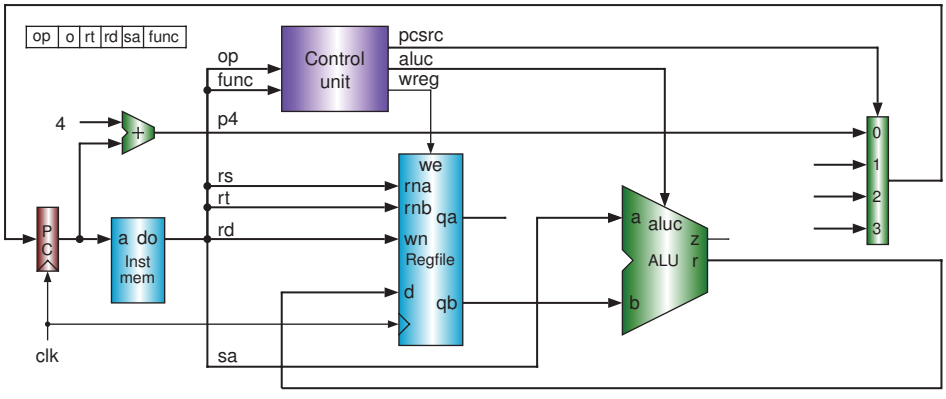

Figure 5.4 Block diagram for R-format shift instructions

(PC source) is the selection signal of the multiplexer. `aluc` (ALU control) controls the operations of ALU. `wreg` (write register) is a write enable signal. If `wreg` is a 1, the result calculated by the ALU will be written to the register `rd` of the register file.

Figure 5.4 shows the circuits required for executing the following instructions:

```
sll rd, rt, sa          # rd <-- rt SLL sa
srl rd, rt, sa          # rd <-- rt SRL sa
sra rd, rt, sa          # rd <-- rt SRA sa
```

Different from Figure 5.3, the input `a` of the ALU in Figure 5.4 is the 5-bit `sa` (shift amount). The high 27 bits of ALU's input `a` can be any value which will be ignored by the shift operations. The `rs` field is not used. We will use a multiplexer to select `sa` or `qa` of the register file based on the instruction.

5.1.2.2 The Circuits Required by I-Format Instructions

Figure 5.5 shows the circuits required for executing the following instructions:

```
addi rt, rs, immediate   # rt <-- rs ADD (sign)immediate
andi rt, rs, immediate   # rt <-- rs AND (zero)immediate
ori  rt, rs, immediate   # rt <-- rs OR  (zero)immediate
xori rt, rs, immediate   # rt <-- rs XOR (zero)immediate
lui  rt,     immediate   # rt <--    LUI       immediate
```

These five instructions use immediate as the input `b` of ALU. The 16-bit `immediate` must be extended to 32 bits. Similarly, a multiplexer will be inserted in front of the ALU's input `b`. `rt` is the destination register number.

5.1.2.3 The Circuits Required by Load/Store Instructions

Figure 5.6 shows the circuits required for executing the following instructions:

```
lw rt, offset(rs)        # rt <-- memory[rs + offset]
sw rt, offset(rs)        # memory[rs + offset] <-- rt
```

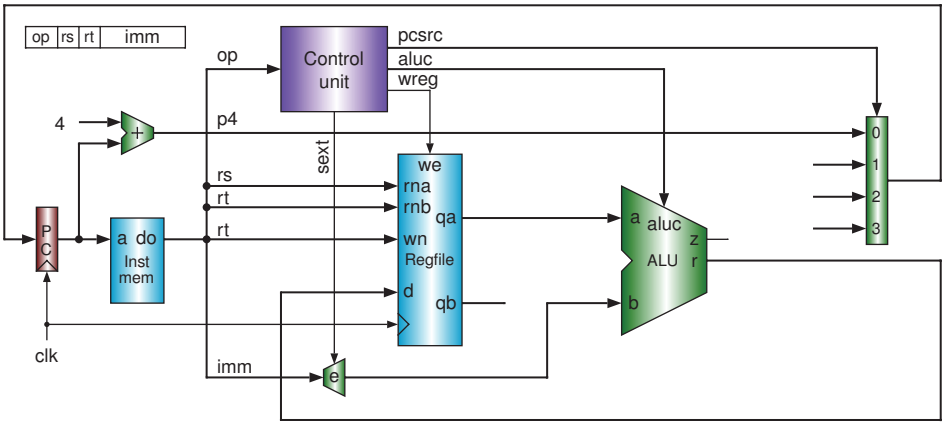

Figure 5.5 Block diagram for I-format arithmetic and logic instructions

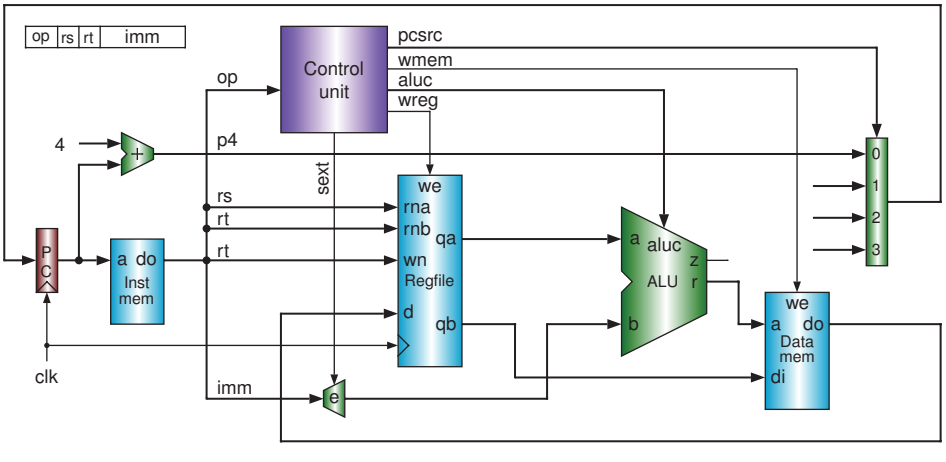

Figure 5.6 Block diagram for I-format load and store instructions

These two instructions access the data memory (`Data mem` component in the figure). The memory address is calculated by the ALU in the same way as the `addi` instruction does. For the `lw` instruction, the data word read from the data memory is written to the `rt` register of the register file. The `sw` instruction writes the data held in the `rt` register to the data memory. `wmem` (write memory), one of the control signals generated by the control unit, is the memory write enable signal.

5.1.2.4 The Circuits Required by Conditional Branch Instructions

Figure 5.7 shows the circuits required for executing the following instructions:

```
beq rs, rt, label        # if (rs == rt) PC <-- label
bne rs, rt, label        # if (rs != rt) PC <-- label
```

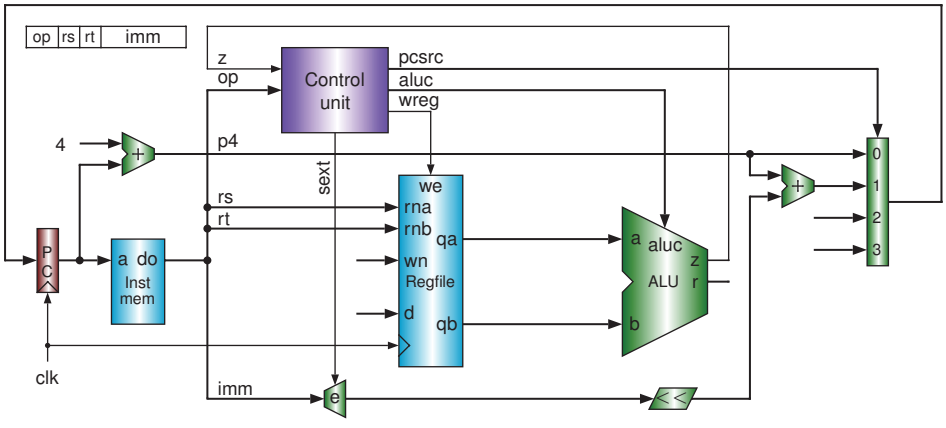

Figure 5.7 Block diagram for I-format branch instructions

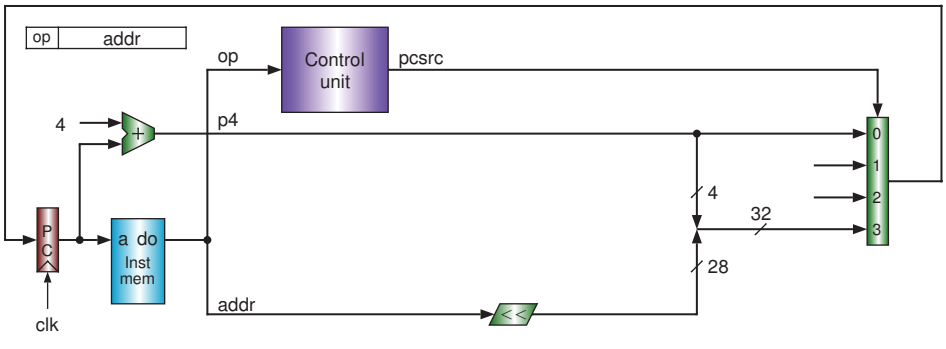

Figure 5.8 Block diagram for J-format jump instruction

The two operands in registers `rs` and `rt` are compared by the ALU. The result of the comparison `z` is sent to the control unit to determine whether to branch or not. If the branch is taken, the 2-bit `pcsrc` signal will be 01 to select the branch target address. An additional 32-bit adder is used to calculate the branch target address, which is the sum of the PC+4 (`p4` in the figure) and the 2-bit left-shifted sign-extended immediate. These two instructions do not write the register file (`wreg` = 0).

5.1.2.5 The Circuits Required by Jump Instructions

Figure 5.8 shows the circuits required for executing the jump instruction:

```
j target                    # PC <-- target
```

This instruction uses neither the ALU nor the register file. The jump target address is obtained by the concatenation of the high 4 bits of PC+4 and 2-bit left-shifted `addr`. `pcsrc` will be 11 to select the jump target address.

Figure 5.9 shows the circuits required for executing the `jal` instruction:

```
jal target                  # $31 <-- PC + 4; PC <-- target
```

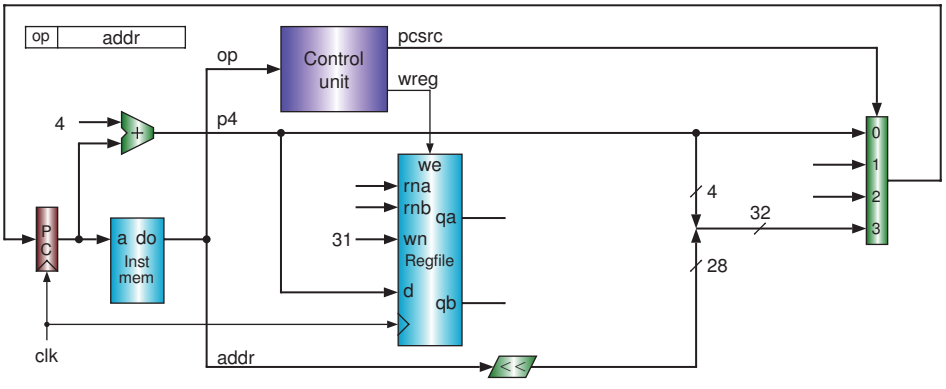

Figure 5.9 Block diagram for J-format jump-and-link instruction

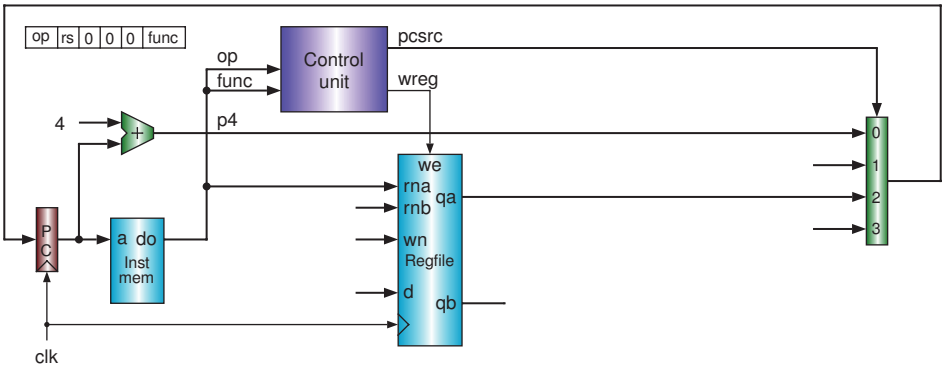

Figure 5.10 Block diagram for R-format jump-register instruction

In addition to what the `j` instruction does, the `jal` instruction saves the return address to register `$31` of the register file. The number 31 does not appear in the instruction; we must create it by hardwire. We used PC+4 as the return address in our single-cycle CPU design. MIPS ISA defines it as PC+8. We will use PC+8 as the return address in the design of a pipelined CPU in Chapter 8.

Figure 5.10 shows the circuits required for executing the `jr` instruction:

```
jr rs  # PC <-- rs
```

The jump target address is obtained from register `rs` of the register file. `pcsrc` will be 10 to select the jump target address.

The circuits required by executing the 20 instructions have been given separately. We will describe how to combine these circuits together to form a CPU. Before that, we describe how to design the register file.

5.2 Register File Design

The register file contains 32 general-purpose registers. Figure 5.11 shows the symbol of the register file that will be used in our CPU design. It has two read ports (port A and port B) and a write port. Each port

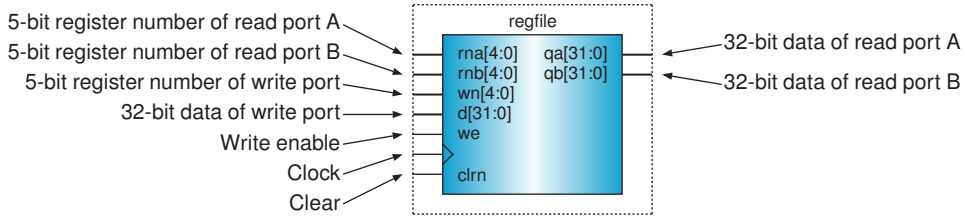

Figure 5.11 Symbol of the register file

has a 5-bit address input (register number). A 32-bit data input and a write enable are provided for the write port.

This section first gives the schematic circuit of the register file and its Verilog HDL code. Then a behavioral-style Verilog HDL code is given which also implements the register file but is short.

5.2.1 Register File Schematic Circuit

A register in the register file has 32 bits. It can be designed with 32 D flip-flops, as shown in Figure 5.12. We name it `dffe32`. The input `e` is the write enable.

The register file has 32 registers but the register `$0` always contains a constant 0. Thus we can use 31 `dffe32s` to implement registers `$1–$31`, and use a `gnd` (ground) for register `$0`, as shown in Figure 5.13. We name it `reg32`. It has thirty-two 32-bit outputs: there are 32 registers, and each register has a 32-bit output.

The final schematic circuit of the register file is shown in Figure 5.14. `reg32` is the storage body. It has thirty-two 32-bit outputs. Each read port uses a 32-bit 32-to-1 multiplexer (`mux32x32`) to select one register output according to its 5-bit read register number. The two read ports are independent of each other.

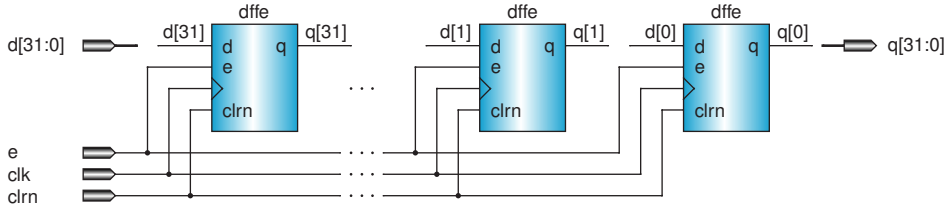

Figure 5.12 Block diagram of 32-bit D flip-flops with enable control

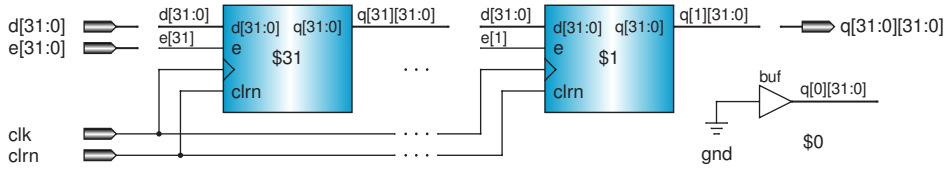

Figure 5.13 Block diagram of thirty-two 32-bit registers

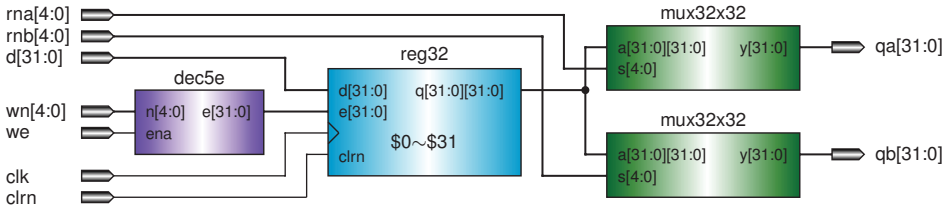

Figure 5.14 Block diagram of the register file

A 5-32 decoder (`dec5e`) generates at most one output that is a 1 among 32 outputs according to the 5-bit write register number. Each output of the decoder is connected to the enable input of the corresponding register so that at most one register will be written with the 32-bit input data at the rising edge of the clock. If the `we` signal is 0, no register is updated.

5.2.2 Register File Design in Dataflow-Style Verilog HDL

The following Verilog HDL code implements the schematic circuit shown in Figure 5.14. It invokes the `dec5e` (5-32 decoder with enable control) module and the `dffe32` (32-bit register with enable control) module, which will be given next.

```
module regfile_dataflow (rna,rnb,d,wn,we,clk,clrn,qa,qb);
    input   [4:0] rna,rnb,wn;                                   // reg numbers
    input  [31:0] d;                                            // write data
    input         we;                                           // write enable
    input         clk, clrn;                                    // clock, reset
    output [31:0] qa,qb;                                        // read ports
    wire   [31:0] e;                                            // enables
    wire   [31:0] r00,r01,r02,r03,r04,r05,r06,r07;
    wire   [31:0] r08,r09,r10,r11,r12,r13,r14,r15;
    wire   [31:0] r16,r17,r18,r19,r20,r21,r22,r23;
    wire   [31:0] r24,r25,r26,r27,r28,r29,r30,r31;
    dec5e decoder (wn,we,e);                                    // wn decoder
    assign        r00 = 0;                                      // $0
    dffe32 reg01  (d,clk,clrn,e[01],r01);                       // $1
    dffe32 reg02  (d,clk,clrn,e[02],r02);                       // $2
    dffe32 reg03  (d,clk,clrn,e[03],r03);                       // $3
    dffe32 reg04  (d,clk,clrn,e[04],r04);                       // $4
    dffe32 reg05  (d,clk,clrn,e[05],r05);                       // $5
    dffe32 reg06  (d,clk,clrn,e[06],r06);                       // $6
    dffe32 reg07  (d,clk,clrn,e[07],r07);                       // $7
    dffe32 reg08  (d,clk,clrn,e[08],r08);                       // $8
    dffe32 reg09  (d,clk,clrn,e[09],r09);                       // $9
    dffe32 reg10  (d,clk,clrn,e[10],r10);                       // $10
    dffe32 reg11  (d,clk,clrn,e[11],r11);                       // $11
    dffe32 reg12  (d,clk,clrn,e[12],r12);                       // $12
    dffe32 reg13  (d,clk,clrn,e[13],r13);                       // $13
```

```
dffe32 reg14  (d,clk,clrn,e[14],r14);                    // $14
dffe32 reg15  (d,clk,clrn,e[15],r15);                    // $15
dffe32 reg16  (d,clk,clrn,e[16],r16);                    // $16
dffe32 reg17  (d,clk,clrn,e[17],r17);                    // $17
dffe32 reg18  (d,clk,clrn,e[18],r18);                    // $18
dffe32 reg19  (d,clk,clrn,e[19],r19);                    // $19
dffe32 reg20  (d,clk,clrn,e[20],r20);                    // $20
dffe32 reg21  (d,clk,clrn,e[21],r21);                    // $21
dffe32 reg22  (d,clk,clrn,e[22],r22);                    // $22
dffe32 reg23  (d,clk,clrn,e[23],r23);                    // $23
dffe32 reg24  (d,clk,clrn,e[24],r24);                    // $24
dffe32 reg25  (d,clk,clrn,e[25],r25);                    // $25
dffe32 reg26  (d,clk,clrn,e[26],r26);                    // $26
dffe32 reg27  (d,clk,clrn,e[27],r27);                    // $27
dffe32 reg28  (d,clk,clrn,e[28],r28);                    // $28
dffe32 reg29  (d,clk,clrn,e[29],r29);                    // $29
dffe32 reg30  (d,clk,clrn,e[30],r30);                    // $30
dffe32 reg31  (d,clk,clrn,e[31],r31);                    // $31
assign qa = select(r00,r01,r02,r03,r04,r05,r06,r07,      // read port a
                   r08,r09,r10,r11,r12,r13,r14,r15,
                   r16,r17,r18,r19,r20,r21,r22,r23,
                   r24,r25,r26,r27,r28,r29,r30,r31,rna);
assign qb = select(r00,r01,r02,r03,r04,r05,r06,r07,      // read port b
                   r08,r09,r10,r11,r12,r13,r14,r15,
                   r16,r17,r18,r19,r20,r21,r22,r23,
                   r24,r25,r26,r27,r28,r29,r30,r31,rnb);
function  [31:0] select;
    input [31:0] r00,r01,r02,r03,r04,r05,r06,r07,
                 r08,r09,r10,r11,r12,r13,r14,r15,
                 r16,r17,r18,r19,r20,r21,r22,r23,
                 r24,r25,r26,r27,r28,r29,r30,r31;
    input [4:0]  s;                                      // reg number
    case (s)
        5'd00: select = r00;          5'd01: select = r01;
        5'd02: select = r02;          5'd03: select = r03;
        5'd04: select = r04;          5'd05: select = r05;
        5'd06: select = r06;          5'd07: select = r07;
        5'd08: select = r08;          5'd09: select = r09;
        5'd10: select = r10;          5'd11: select = r11;
        5'd12: select = r12;          5'd13: select = r13;
        5'd14: select = r14;          5'd15: select = r15;
        5'd16: select = r16;          5'd17: select = r17;
        5'd18: select = r18;          5'd19: select = r19;
        5'd20: select = r20;          5'd21: select = r21;
        5'd22: select = r22;          5'd23: select = r23;
        5'd24: select = r24;          5'd25: select = r25;
        5'd26: select = r26;          5'd27: select = r27;
```

```
            5'd28: select = r28;            5'd29: select = r29;
            5'd30: select = r30;            5'd31: select = r31;
        endcase
    endfunction
endmodule
```

The following Verilog HDL code implements the 5-32 decoder with enable control.

```
module dec5e (n,ena,e);                  // 5-32 decoder with an enable
    input   [4:0] n;                     // 5-bit number
    input         ena;                   // master enable
    output [31:0] e;                     // 32-bit enables
    assign        e = ena? decoder(n) : 32'h00000000;
    function [31:0] decoder;
        input [4:0] n;
        case (n)
            5'd00: decoder=32'h00000001; // 00000000000000000000000000000001
            5'd01: decoder=32'h00000002; // 00000000000000000000000000000010
            5'd02: decoder=32'h00000004; // 00000000000000000000000000000100
            5'd03: decoder=32'h00000008; // 00000000000000000000000000001000
            5'd04: decoder=32'h00000010; // 00000000000000000000000000010000
            5'd05: decoder=32'h00000020; // 00000000000000000000000000100000
            5'd06: decoder=32'h00000040; // 00000000000000000000000001000000
            5'd07: decoder=32'h00000080; // 00000000000000000000000010000000
            5'd08: decoder=32'h00000100; // 00000000000000000000000100000000
            5'd09: decoder=32'h00000200; // 00000000000000000000001000000000
            5'd10: decoder=32'h00000400; // 00000000000000000000010000000000
            5'd11: decoder=32'h00000800; // 00000000000000000000100000000000
            5'd12: decoder=32'h00001000; // 00000000000000000001000000000000
            5'd13: decoder=32'h00002000; // 00000000000000000010000000000000
            5'd14: decoder=32'h00004000; // 00000000000000000100000000000000
            5'd15: decoder=32'h00008000; // 00000000000000001000000000000000
            5'd16: decoder=32'h00010000; // 00000000000000010000000000000000
            5'd17: decoder=32'h00020000; // 00000000000000100000000000000000
            5'd18: decoder=32'h00040000; // 00000000000001000000000000000000
            5'd19: decoder=32'h00080000; // 00000000000010000000000000000000
            5'd20: decoder=32'h00100000; // 00000000000100000000000000000000
            5'd21: decoder=32'h00200000; // 00000000001000000000000000000000
            5'd22: decoder=32'h00400000; // 00000000010000000000000000000000
            5'd23: decoder=32'h00800000; // 00000000100000000000000000000000
            5'd24: decoder=32'h01000000; // 00000001000000000000000000000000
            5'd25: decoder=32'h02000000; // 00000010000000000000000000000000
            5'd26: decoder=32'h04000000; // 00000100000000000000000000000000
            5'd27: decoder=32'h08000000; // 00001000000000000000000000000000
            5'd28: decoder=32'h10000000; // 00010000000000000000000000000000
            5'd29: decoder=32'h20000000; // 00100000000000000000000000000000
            5'd30: decoder=32'h40000000; // 01000000000000000000000000000000
```

```
            5'd31: decoder=32'h80000000; // 10000000000000000000000000000000
        endcase
    endfunction
endmodule
```

The following Verilog HDL code implements the 32-bit register with enable control.

```
module dffe32 (d,clk,clrn,e,q);                          // a 32-bit register
    input      [31:0] d;                                 // input d
    input             e;                                 // e: enable
    input             clk, clrn;                         // clock and reset
    output reg [31:0] q;                                 // output q
    always @(negedge clrn or posedge clk)
        if (!clrn)  q <= 0;                              // q = 0 if reset
        else if (e) q <= d;                              // save d if enabled
endmodule
```

5.2.3 Register File Design in Behavioral-Style Verilog HDL

The code given in the previous section is too "hard"; it is almost the same as the hardware circuit shown in Figure 5.14. In this section, we show that the register file can be designed with Verilog HDL in a "soft" way.

The Verilog HDL supports a two-dimensional array (matrix) statement. For example, the statement `reg [31:0] register [0:31]` defines a 32×32-bit storage: `[31:0]` defines a register that has 32 bits and `[0:31]` declares that there are 32 such registers. Then we can use an index to read or write a 32-bit register. A behavioral-style Verilog HDL code that implements the register file is listed below.

```
module regfile (rna,rnb,d,wn,we,clk,clrn,qa,qb);     // 32x32 regfile
    input  [31:0] d;                                 // data of write port
    input   [4:0] rna;                               // reg # of read port A
    input   [4:0] rnb;                               // reg # of read port B
    input   [4:0] wn;                                // reg # of write port
    input         we;                                // write enable
    input         clk, clrn;                         // clock and reset
    output [31:0] qa, qb;                            // read ports A and B
    reg    [31:0] register [1:31];                   // 31 32-bit registers
    assign qa = (rna == 0)? 0 : register[rna];       // read port A
    assign qb = (rnb == 0)? 0 : register[rnb];       // read port B
    integer i;
    always @(posedge clk or negedge clrn)            // write port
        if (!clrn)
            for (i = 1; i < 32; i = i + 1)
                register[i]  <= 0;                   // reset
        else
            if ((wn != 0) && we)                     // not reg[0] & enabled
                register[wn] <= d;                   // write d to reg[wn]
endmodule
```

This register file has one write port. In Chapter 10, we will give the Verilog HDL code for implementing a register file with two write ports.

5.3 Single-Cycle CPU Datapath Design

A CPU consists of a datapath and a control unit. A datapath is a collection of functional units, such as the ALU, register file, and multiplexers, that perform data processing operations. A control unit is a component that manages the operations of the datapath. This section introduces the design method of the datapath of the single-cycle CPU.

5.3.1 Using Multiplexers

If there are two or more signals that will be connected to the same input of a component, a multiplexer can be used for selecting a signal based on the instruction.

5.3.1.1 Selection for Next PC (Selection Signal: pcsrc)

The PC will be updated on the rising edge of the clock in the single-cycle CPU. If the current instruction is neither a branch nor a jump, PC+4 will be written to the PC. But the following five instructions may transfer control to a branch or jump target address that is not the PC+4.

```
beq/bne rs, rt, label # if (eq/ne) pc <-- label
jr      rs            # pc <-- rs
j/jal   address       # pc <-- address << 2
```

op	rs	rt	offset

op	rs	0	0	0	func

op	address

A 4-to-1 multiplexer can be used to select an address for the next PC as shown in Figure 5.15. The input 0 of the multiplexer is PC+4 (neither branch nor jump); input 1 is the branch target address; input 2 is the jump target address coming from the register `rs` of the register file; and input 3 is the jump target address coming from `addr` and PC+4. The 2-bit `pcsrc` (PC source) is the selection signal of the multiplexer.

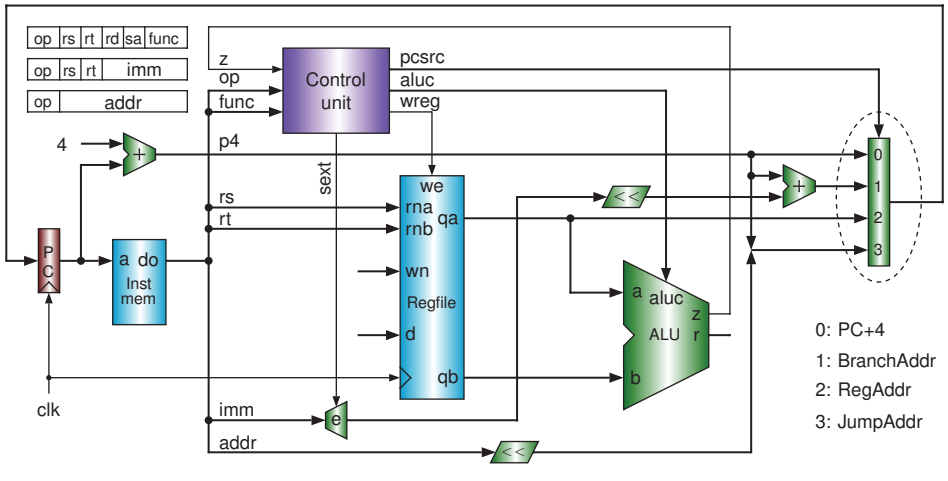

Figure 5.15 Using multiplexer for the next PC selection

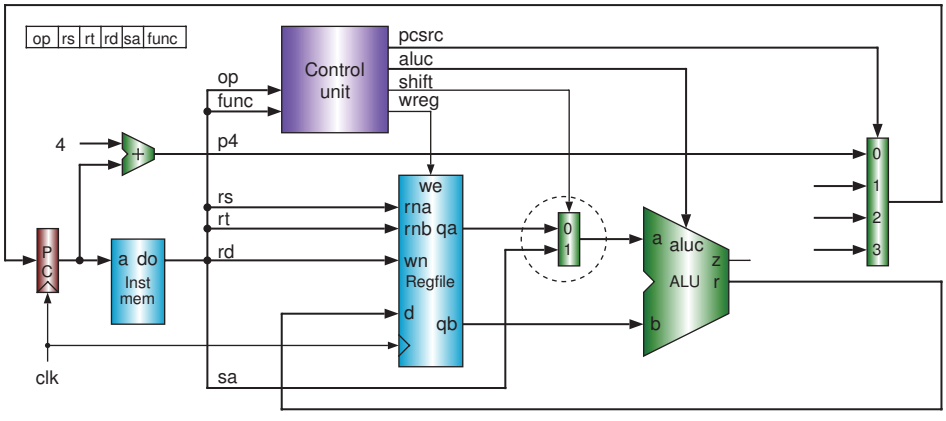

Figure 5.16 Using a multiplexer for ALU A-input selection

5.3.1.2 Selection for ALU Input A (Selection Signal: shift)

Shift instructions use `sa` as the shift amount that will be sent to the input `a` of the ALU for the operation. Other instructions may use the value in the register `rs` of the register file; see the following two examples.

```
add rd, rs, rt        # rd <-- rs + rt
sll rd, rt, sa        # rd <-- rt << sa
```

op	rs	rt	rd	0	func

op	0	rt	rd	sa	func

`sa` or the value of the register `rs` can be selected by a 2-to-1 multiplexer, as shown in Figure 5.16. The 1-bit `shift` is the selection signal of the multiplexer. If `shift` is a 1, `sa` is selected. Otherwise, `qa` of the register file is selected.

5.3.1.3 Selection for ALU Input B (Selection Signals: aluimm and regrt)

The immediate (`imm`) of the I-format computational instructions will be sent to the input `b` of the ALU for the calculation. Other instructions may use the value in the register `rt` of the register file; see the following two examples.

```
add  rd, rs, rt       # rd <-- rs + rt
addi rt, rs, imm      # rt <-- rs + imm
```

op	rs	rt	rd	0	func

op	rs	rt	imm

In addition to the input `b` of the ALU, the destination register number also needs to be selected from `rd` and `rt`. Figure 5.17 shows these selections by two 2-to-1 multiplexers. The multiplexer in the front of the ALU input `b` has 32 bits and its selection signal is `aluimm`. If `aluimm` is a 1, the extended immediate is selected. Otherwise, `qb` of the register file is selected. The multiplexer in front of the register file's input `wn` has 5 bits, and its selection signal is `regrt`. If `regrt` is a 1, `rt` is selected. Otherwise, `rd` is selected.

5.3.1.4 Selection for Register File Inputs (Selection Signals: m2reg and jal)

The data that will be written to the register file may be the output of the ALU, the data in the data memory, or the return address (for `jal` instruction). The destination register number may be `rd`, `rt`, or a constant 31 (for `jal` instruction).

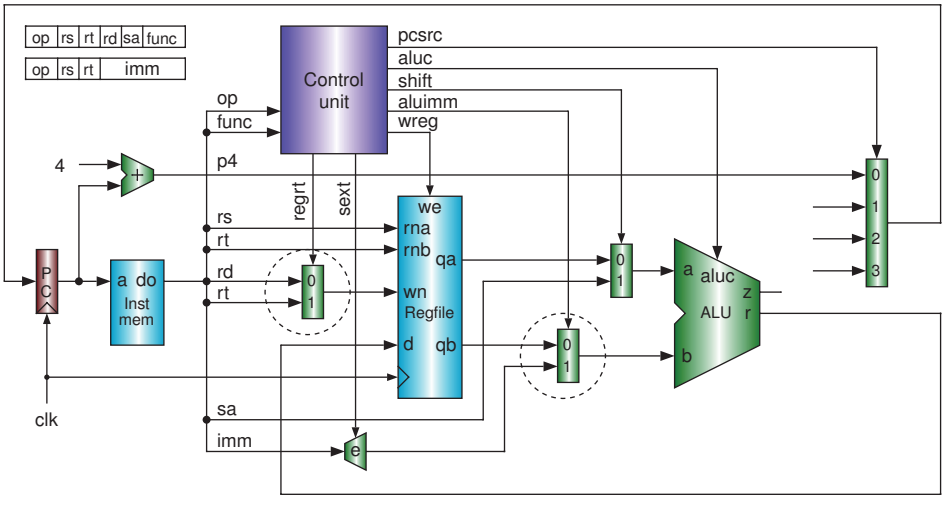

Figure 5.17 Using a multiplexer for ALU B-input selection

The load (`lw`) instruction writes the data read from the memory to the register `rt` of the register file. Other instructions may write the output of the ALU to the register `rd` or `rt` of the register file. The subroutine call `jal` instruction saves the return address to register `$31`. See the following three examples.

```
add rd, rs, rt        # rd <-- rs + rt
lw  rt, imm(rs)       # rt <-- mem[rs+imm]
jal address # $31 <-- pc + 4; pc <-- address << 2
```

op	rs	rt	rd	0	func

op	rs	rt	imm

op	address

Two 32-bit 2-to-1 multiplexers are used for selecting the data that will be written to the register file, as shown in Figure 5.18. The selection signal of the multiplexer on the right side of the figure is `m2reg`. If `m2reg` is a 1, the data read from the memory is selected. The selection signal of the multiplexer on the left side is `jal`. If `jal` is a 1, the return address `p4` is selected.

For the `jal` instruction, because the destination register number 31 does not appear in the instruction, we must generate it by hardware (the component `f` in the figure). The following Verilog HDL code is a hardware implementation for generating the 5-bit destination register number:

```
assign wn = reg_dest | {5{jal}};
```

It performs a bitwise logical OR operation. If `jal` is a 1, a 5-bit pattern 11111 (decimal 31) will be assigned to the destination register number `wn`. Otherwise, `reg_dest`, which is `rd` or `rt`, will be assigned to `wn`.

5.3.2 Single-Cycle CPU Schematic Circuit

By summarizing the discussions in the previous section, we got the schematic circuit of the single-cycle CPU plus the instruction memory and the data memory, as shown in Figure 5.19, which can execute all the instructions listed in Table 4.4.

Putting the memory modules outside, Figure 5.20 shows the single-cycle computer that consists of a single-cycle CPU and two memory modules.

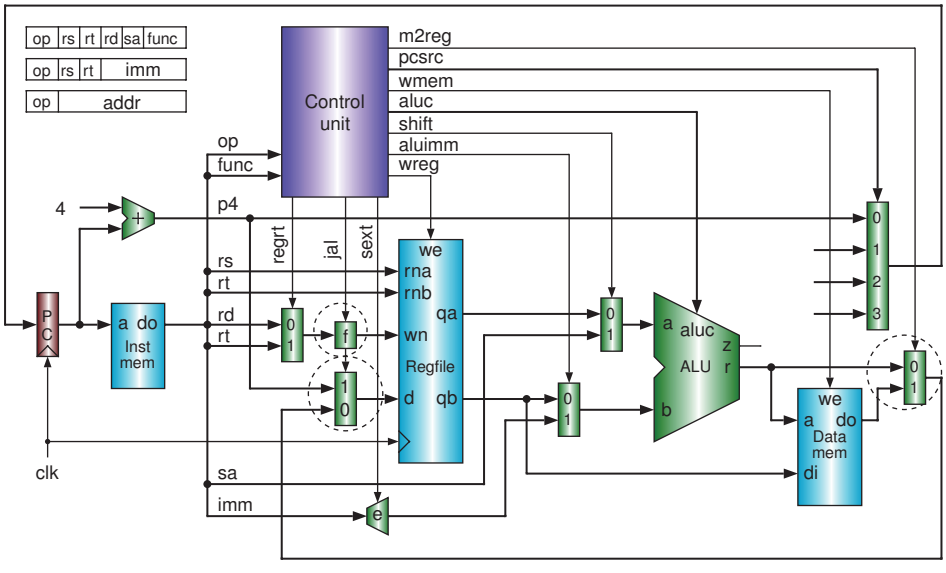

Figure 5.18 Using a multiplexer for register file written data selection

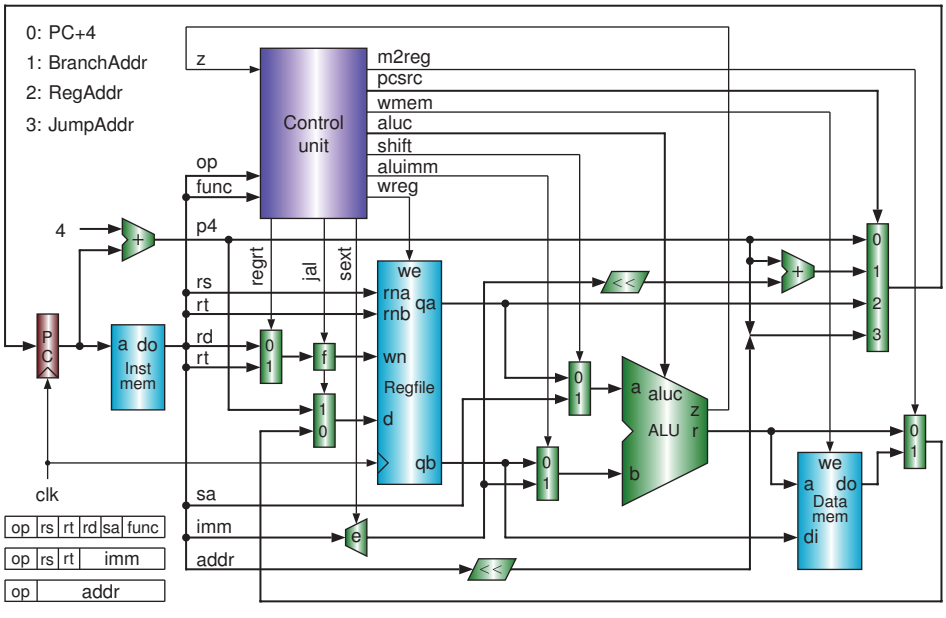

Figure 5.19 Block diagram of a single-cycle CPU

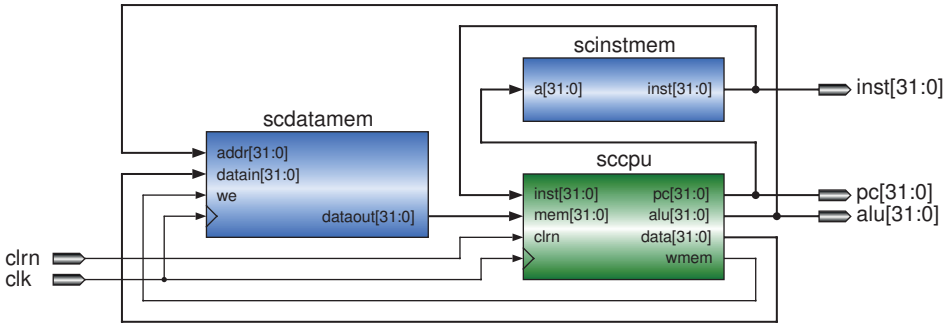

Figure 5.20 Block diagram of single-cycle computer

The reason why we use separate instruction memory and data memory is that the single-cycle CPU completes the execution of an instruction, including instruction fetch and data memory access, in one clock cycle. We can implement an instruction cache and a data cache inside the CPU so that only one off-chip memory module can be used for storing both the program (instructions) and data.

5.3.3 Single-Cycle CPU Verilog HDL Codes

The following Verilog HDL code implements the single-cycle computer shown in Figure 5.20. It invokes `sccpu` (single-cycle CPU), `scinstmem` (instruction memory), and `scdatamem` (data memory). The first module is given next, and the last two modules will be given later.

```
module sccomp (clk,clrn,inst,pc,aluout,memout);   // single cycle computer
    input         clk, clrn;                      // clock and reset
    output [31:0] pc;                             // program counter
    output [31:0] inst;                           // instruction
    output [31:0] aluout;                         // alu output
    output [31:0] memout;                         // data memory output
    wire   [31:0] data;                           // data to data memory
    wire          wmem;                           // write data memory
    sccpu cpu (clk,clrn,inst,memout,pc,wmem,aluout,data);   // cpu
    scinstmem imem (pc,inst);                                // inst memory
    scdatamem dmem (clk,memout,data,aluout,wmem);            // data memory
endmodule
```

The following Verilog HDL code that implements a single-cycle CPU is almost identical to the schematic circuit shown in Figure 5.19.

```
module sccpu (clk,clrn,inst,mem,pc,wmem,alu,data);
    input  [31:0] inst;                           // inst from inst memory
    input  [31:0] mem;                            // data from data memory
    input         clk, clrn;                      // clock and reset
    output [31:0] pc;                             // program counter
    output [31:0] alu;                            // alu output
    output [31:0] data;                           // data to data memory
    output        wmem;                           // write data memory
```

```
// instruction fields
wire    [5:0] op   = inst[31:26];           // op
wire    [4:0] rs   = inst[25:21];           // rs
wire    [4:0] rt   = inst[20:16];           // rt
wire    [4:0] rd   = inst[15:11];           // rd
wire    [5:0] func = inst[05:00];           // func
wire   [15:0] imm  = inst[15:00];           // immediate
wire   [25:0] addr = inst[25:00];           // address
// control signals
wire    [3:0] aluc;                         // alu operation control
wire    [1:0] pcsrc;                        // select pc source
wire          wreg;                         // write regfile
wire          regrt;                        // dest reg number is rt
wire          m2reg;                        // instruction is an lw
wire          shift;                        // instruction is a shift
wire          aluimm;                       // alu input b is an i32
wire          jal;                          // instruction is a jal
wire          sext;                         // is sign extension
// datapath wires
wire   [31:0] p4;                           // pc+4
wire   [31:0] bpc;                          // branch target address
wire   [31:0] npc;                          // next pc
wire   [31:0] qa;                           // regfile output port a
wire   [31:0] qb;                           // regfile output port b
wire   [31:0] alua;                         // alu input a
wire   [31:0] alub;                         // alu input b
wire   [31:0] wd;                           // regfile write port data
wire   [31:0] r;                            // alu out or mem
wire   [31:0] sa  = {27'b0,inst[10:6]};     // shift amount
wire   [15:0] s16 = {16{sext & inst[15]}};  // 16-bit signs
wire   [31:0] i32 = {s16,imm};              // 32-bit immediate
wire   [31:0] dis = {s16[13:0],imm,2'b00};  // word distance
wire   [31:0] jpc = {p4[31:28],addr,2'b00}; // jump target address
wire    [4:0] reg_dest;                     // rs or rt
wire    [4:0] wn  = reg_dest | {5{jal}};    // regfile write reg #
wire          z;                            // alu, zero tag
// control unit
sccu_dataflow cu (op,func,z,wmem,wreg,      // control unit
                  regrt,m2reg,aluc,shift,
                  aluimm,pcsrc,jal,sext);
// datapath
dff32 i_point (npc,clk,clrn,pc);            // pc register
cla32 pcplus4 (pc,32'h4,1'b0,p4);           // pc + 4
cla32 br_addr (p4,dis,1'b0,bpc);            // branch target address
mux2x32 alu_a (qa,sa,shift,alua);           // alu input a
mux2x32 alu_b (qb,i32,aluimm,alub);         // alu input b
mux2x32 alu_m (alu,mem,m2reg,r);            // alu out or mem
```

```
    mux2x32 link  (r,p4,jal,wd);                    // r or p4
    mux2x5 reg_wn (rd,rt,regrt,reg_dest);           // rs or rt
    mux4x32 nextpc(p4,bpc,qa,jpc,pcsrc,npc);        // next pc
    regfile rf (rs,rt,wd,wn,wreg,clk,clrn,qa,qb); // register file
    alu alunit (alua,alub,aluc,alu,z);              // alu
    assign data = qb;                               // regfile output port b
endmodule
```

The Verilog HDL code of the module `sccu_dataflow` (control unit) invoked by `sccpu_dataflow` (CPU) will be given in the following section.

5.4 Single-Cycle CPU Control Unit Design

As mentioned in the previous section, a CPU consists of a datapath and a control unit. A control unit is a component that directs the operations of the datapath. This section introduces the control unit design of the single-cycle CPU.

5.4.1 Logic Design of the Control Unit

A control unit generates control signals based on the instruction that is currently being executed. The first step of designing the control unit is to decode the instruction based on the instruction's opcode (`op`), as listed in Table 5.1. The function code (`func`) needs to be checked also if the instruction has an R-format. In order to avoid conflict with the keywords of Verilog HDL, an `i_` is prefixed to the name of each instruction.

From the table, we can get each instruction decode expression as shown below. A temporary wire, `rtype`, is used for decoding all the R-format instructions.

```
rtype = ~op[5] ~op[4] ~op[3] ~op[2] ~op[1] ~op[0];
i_add =  rtype func[5] ~func[4] ~func[3] ~func[2] ~func[1] ~func[0];
i_sub =  rtype func[5] ~func[4] ~func[3] ~func[2] func[1] ~func[0];
```

Table 5.1 Instruction decode

R-format			I- and J-format	
Inst.	op[5:0]	func[5:0]	Inst.	op[5:0]
i_add	000000	100000	i_addi	001000
i_sub	000000	100010	i_andi	001100
i_and	000000	100100	i_ori	001101
i_or	000000	100101	i_xori	001110
i_xor	000000	100110	i_lw	100011
i_sll	000000	000000	i_sw	101011
i_srl	000000	000010	i_beq	000100
i_sra	000000	000011	i_bne	000101
i_jr	000000	001000	i_lui	001111
			i_j	000010
			i_jal	000011

```
 i_and =  rtype func[5] ~func[4] ~func[3] func[2] ~func[1] ~func[0];
  i_or =  rtype func[5] ~func[4] ~func[3] func[2] ~func[1] func[0];
 i_xor =  rtype func[5] ~func[4] ~func[3] func[2] func[1] ~func[0];
 i_sll =  rtype ~func[5] ~func[4] ~func[3] ~func[2] ~func[1] ~func[0];
 i_srl = rtype ~func[5] ~func[4] ~func[3] ~func[2] func[1] ~func[0];
 i_sra =  rtype ~func[5] ~func[4] ~func[3] ~func[2] func[1] func[0];
  i_jr =  rtype ~func[5] ~func[4] func[3] ~func[2] ~func[1] ~func[0];
i_addi = ~op[5] ~op[4] op[3] ~op[2] ~op[1] ~op[0];
i_andi = ~op[5] ~op[4] op[3] op[2] ~op[1] ~op[0];
 i_ori = ~op[5] ~op[4] op[3] op[2] ~op[1] op[0];
i_xori = ~op[5] ~op[4] op[3] op[2] op[1] ~op[0];
  i_lw = op[5] ~op[4] ~op[3] ~op[2] op[1] op[0];
  i_sw = op[5] ~op[4] op[3] ~op[2] op[1] op[0];
 i_beq = ~op[5] ~op[4] ~op[3] op[2] ~op[1] ~op[0];
 i_bne = ~op[5] ~op[4] ~op[3] op[2] ~op[1] op[0];
 i_lui = ~op[5] ~op[4] op[3] op[2] op[1] op[0];
   i_j = ~op[5] ~op[4] ~op[3] ~op[2] op[1] ~op[0];
 i_jal = ~op[5] ~op[4] ~op[3] ~op[2] op[1] op[0];
```

The control signals can be classified into the following four categories: (i) the selection signals of the multiplexers; (ii) the ALU operation control signal; (iii) the register file and memory write enables; and (iv) others such as sign or zero extension. Table 5.2 lists all the control signals of the single-cycle CPU.

Table 5.2 Control signals of single-cycle CPU

Signal	Meaning	Action
`wreg`	Write register	1: write; 0: do not write
`regrt`	Destination register is `rt`	1: select `rt`; 0: select `rd`
`jal`	Subroutine call	1: is `jal`; 0: is not `jal`
`m2reg`	Save memory data	1: select memory data; 0: select ALU result
`shift`	ALU A uses `sa`	1: select `sa`; 0: select register data
`aluimm`	ALU B uses immediate	1: select immediate; 0: select register data
`sext`	Immediate sign extend	1: sign-extend; 0: zero extend
`aluc[3:0]`	ALU operation control	x000: ADD; x100: SUB; x001: AND x101: OR; x010: XOR; x110: LUI 0011: SLL; 0111: SRL; 1111: SRA
`wmem`	Write memory	1: write memory; 0: do not write
`pcsrc[1:0]`	Next instruction address	00: select PC+4; 01: branch address 10: register data; 11: jump address

Table 5.3 Truth table of the control signals

Inst.	z	wreg	regrt	jal	m2reg	shift	aluimm	sext	aluc[3:0]	wmem	pcsrc[1:0]
i_add	x	1	0	0	0	0	0	x	x 0 0 0	0	0 0
i_sub	x	1	0	0	0	0	0	x	x 1 0 0	0	0 0
i_and	x	1	0	0	0	0	0	x	x 0 0 1	0	0 0
i_or	x	1	0	0	0	0	0	x	x 1 0 1	0	0 0
i_xor	x	1	0	0	0	0	0	x	x 0 1 0	0	0 0
i_sll	x	1	0	0	0	1	0	x	0 0 1 1	0	0 0
i_srl	x	1	0	0	0	1	0	x	0 1 1 1	0	0 0
i_sra	x	1	0	0	0	1	0	x	1 1 1 1	0	0 0
i_jr	x	0	x	x	x	x	x	x	x x x x	0	1 0
i_addi	x	1	1	0	0	0	1	1	x 0 0 0	0	0 0
i_andi	x	1	1	0	0	0	1	0	x 0 0 1	0	0 0
i_ori	x	1	1	0	0	0	1	0	x 1 0 1	0	0 0
i_xori	x	1	1	0	0	0	1	0	x 0 1 0	0	0 0
i_lw	x	1	1	0	1	0	1	1	x 0 0 0	0	0 0
i_sw	x	0	x	x	x	0	1	1	x 0 0 0	1	0 0
i_beq	0	0	x	x	x	0	0	1	x 0 1 0	0	0 0
i_beq	1	0	x	x	x	0	0	1	x 0 1 0	0	0 1
i_bne	0	0	x	x	x	0	0	1	x 0 1 0	0	0 1
i_bne	1	0	x	x	x	0	0	1	x 0 1 0	0	0 0
i_lui	x	1	1	0	0	x	1	x	x 1 1 0	0	0 0
i_j	x	0	x	x	x	x	x	x	x x x x	0	1 1
i_jal	x	1	x	1	x	x	x	x	x x x x	0	1 1

Table 5.3 is the truth table for the control unit. We take the `lw` instruction as an example to explain how to fill the table. The ALU performs addition to calculate the memory address (`aluc[3:0] = x000`); one operand of the addition comes from register `rs` of the register file (`shift = 0`); the other operand is the immediate (`aluimm = 1`); the immediate is sign-extended (`sext = 1`); the result will be written to register file (`wreg = 1`); it is the memory data (`m2reg = 1`); the destination register number is `rt` (`regrt = 1`); it is not a `jal` instruction (`jal = 0`); it does not write memory (`wmem = 0`); and the address of the next instruction is PC+4 (`pcsrc[1:0] = 00`).

From the truth table, we can get the logic expression of each control signal as shown below (not simplified). If there are multiple bits in a control signal, `aluc` for instance, we must write a logic expression for each bit individually.

```
wreg    = i_add + i_sub + i_and + i_or + i_xor + i_sll + i_srl + i_sra +
          i_addi + i_andi + i_ori + i_xori + i_lw + i_lui + i_jal;
regrt   = i_addi + i_andi + i_ori + i_xori + i_lw + i_lui;
m2reg   = i_lw;
shift   = i_sll + i_srl + i_sra;
aluimm  = i_addi + i_andi + i_ori + i_xori + i_lw + i_lui + i_sw;
sext    = i_addi + i_lw + i_sw + i_beq + i_bne;
aluc[3] = i_sra;
aluc[2] = i_sub + i_or + i_srl + i_sra + i_ori + i_lui;
aluc[1] = i_xor + i_sll + i_srl + i_sra + i_xori + i_beq + i_bne + i_lui;
aluc[0] = i_and + i_or + i_sll + i_srl + i_sra + i_andi + i_ori;
```

```
wmem     = i_sw;
pcsrc[1] = i_jr + i_j + i_jal;
pcsrc[0] = i_beq z + i_bne ~z + i_j + i_jal;
```

5.4.2 *Verilog HDL Codes of the Control Unit*

Based on the logic expressions given above, we can write the Verilog HDL code for implementing the control unit of the single-cycle CPU, as listed below. The first half is the code for decoding instructions, and the second half is for generating control signals.

```
module sccu_dataflow (op,func,z,wmem,wreg,regrt,m2reg,aluc,shift,aluimm,
                      pcsrc,jal,sext);            // control unit
  input  [5:0] op, func;                          // op, func
  input        z;                                 // alu zero tag
  output [3:0] aluc;                              // alu operation control
  output [1:0] pcsrc;                             // select pc source
  output       wreg;                              // write regfile
  output       regrt;                             // dest reg number is rt
  output       m2reg;                             // instruction is an lw
  output       shift;                             // instruction is a shift
  output       aluimm;                            // alu input b is an i32
  output       jal;                               // instruction is a jal
  output       sext;                              // is sign extension
  output       wmem;                              // write data memory
  // decode instructions
  wire rtype  = ~|op;                                          // r format
  wire i_add  = rtype& func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
  wire i_sub  = rtype& func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
  wire i_and  = rtype& func[5]&~func[4]&~func[3]& func[2]&~func[1]&~func[0];
  wire i_or   = rtype& func[5]&~func[4]&~func[3]& func[2]&~func[1]& func[0];
  wire i_xor  = rtype& func[5]&~func[4]&~func[3]& func[2]& func[1]&~func[0];
  wire i_sll  = rtype&~func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
  wire i_srl  = rtype&~func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
  wire i_sra  = rtype&~func[5]&~func[4]&~func[3]&~func[2]& func[1]& func[0];
  wire i_jr   = rtype&~func[5]&~func[4]& func[3]&~func[2]&~func[1]&~func[0];
  wire i_addi = ~op[5]&~op[4]& op[3]&~op[2]&~op[1]&~op[0];       // i format
  wire i_andi = ~op[5]&~op[4]& op[3]& op[2]&~op[1]&~op[0];
  wire i_ori  = ~op[5]&~op[4]& op[3]& op[2]&~op[1]& op[0];
  wire i_xori = ~op[5]&~op[4]& op[3]& op[2]& op[1]&~op[0];
  wire i_lw   =  op[5]&~op[4]&~op[3]&~op[2]& op[1]& op[0];
  wire i_sw   =  op[5]&~op[4]& op[3]&~op[2]& op[1]& op[0];
  wire i_beq  = ~op[5]&~op[4]&~op[3]& op[2]&~op[1]&~op[0];
  wire i_bne  = ~op[5]&~op[4]&~op[3]& op[2]&~op[1]& op[0];
  wire i_lui  = ~op[5]&~op[4]& op[3]& op[2]& op[1]& op[0];
  wire i_j    = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]&~op[0];       // j format
  wire i_jal  = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]& op[0];
  // generate control signals
```

```
  assign regrt   = i_addi | i_andi | i_ori  | i_xori | i_lw  | i_lui;
  assign jal     = i_jal;
  assign m2reg   = i_lw;
  assign wmem    = i_sw;
  assign aluc[3] = i_sra;                          // refer to alu.v for aluc
  assign aluc[2] = i_sub  | i_or   | i_srl  | i_sra  | i_ori  | i_lui;
  assign aluc[1] = i_xor  | i_sll  | i_srl  | i_sra  | i_xori | i_beq |
                   i_bne  | i_lui;
  assign aluc[0] = i_and  | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
  assign shift   = i_sll  | i_srl  | i_sra;
  assign aluimm  = i_addi | i_andi | i_ori  | i_xori | i_lw  | i_lui | i_sw;
  assign sext    = i_addi | i_lw   | i_sw   | i_beq  | i_bne;
  assign pcsrc[1]= i_jr   | i_j    | i_jal;
  assign pcsrc[0]= i_beq & z | i_bne &~z | i_j | i_jal;
  assign wreg    = i_add  | i_sub  | i_and  | i_or   | i_xor | i_sll  |
                   i_srl  | i_sra  | i_addi | i_andi | i_ori | i_xori |
                   i_lw   | i_lui  | i_jal;
endmodule
```

5.5 Test Program and Simulation Waveform

The design of the single-cycle CPU has been completed. This section gives the test program that is used to verify the correctness of the CPU. The following Verilog HDL code implements an instruction memory with a read-only memory (ROM). Although an ROM is called a memory, it is actually a combinational circuit.

```
module scinstmem (a,inst);              // instruction memory, rom
    input  [31:0] a;                    // address
    output [31:0] inst;                 // instruction
    wire   [31:0] rom [0:31];           // rom cells: 32 words * 32 bits
    // rom[word_addr] = instruction     // (pc) label   instruction
    assign rom[5'h00] = 32'h3c010000;   // (00) main:   lui  $1, 0
    assign rom[5'h01] = 32'h34240050;   // (04)         ori  $4, $1, 80
    assign rom[5'h02] = 32'h20050004;   // (08)         addi $5, $0,  4
    assign rom[5'h03] = 32'h0c000018;   // (0c) call:   jal  sum
    assign rom[5'h04] = 32'hac820000;   // (10)         sw   $2, 0($4)
    assign rom[5'h05] = 32'h8c890000;   // (14)         lw   $9, 0($4)
    assign rom[5'h06] = 32'h01244022;   // (18)         sub  $8, $9, $4
    assign rom[5'h07] = 32'h20050003;   // (1c)         addi $5, $0,  3
    assign rom[5'h08] = 32'h20a5ffff;   // (20) loop2:  addi $5, $5, -1
    assign rom[5'h09] = 32'h34a8ffff;   // (24)         ori  $8, $5, 0xffff
    assign rom[5'h0A] = 32'h39085555;   // (28)         xori $8, $8, 0x5555
    assign rom[5'h0B] = 32'h2009ffff;   // (2c)         addi $9, $0, -1
    assign rom[5'h0C] = 32'h312affff;   // (30)         andi $10,$9, 0xffff
    assign rom[5'h0D] = 32'h01493025;   // (34)         or   $6, $10, $9
    assign rom[5'h0E] = 32'h01494026;   // (38)         xor  $8, $10, $9
    assign rom[5'h0F] = 32'h01463824;   // (3c)         and  $7, $10, $6
```

```
    assign rom[5'h10] = 32'h10a00001;    // (40)         beq  $5, $0, shift
    assign rom[5'h11] = 32'h08000008;    // (44)         j    loop2
    assign rom[5'h12] = 32'h2005ffff;    // (48) shift:  addi $5, $0, -1
    assign rom[5'h13] = 32'h000543c0;    // (4c)         sll  $8, $5, 15
    assign rom[5'h14] = 32'h00084400;    // (50)         sll  $8, $8, 16
    assign rom[5'h15] = 32'h00084403;    // (54)         sra  $8, $8, 16
    assign rom[5'h16] = 32'h000843c2;    // (58)         srl  $8, $8, 15
    assign rom[5'h17] = 32'h08000017;    // (5c) finish: j    finish
    assign rom[5'h18] = 32'h00004020;    // (60) sum:    add  $8, $0, $0
    assign rom[5'h19] = 32'h8c890000;    // (64) loop:   lw   $9, 0($4)
    assign rom[5'h1A] = 32'h20840004;    // (68)         addi $4, $4,  4
    assign rom[5'h1B] = 32'h01094020;    // (6c)         add  $8, $8, $9
    assign rom[5'h1C] = 32'h20a5ffff;    // (70)         addi $5, $5, -1
    assign rom[5'h1D] = 32'h14a0fffb;    // (74)         bne  $5, $0, loop
    assign rom[5'h1E] = 32'h00081000;    // (78)         sll  $2, $8, 0
    assign rom[5'h1F] = 32'h03e00008;    // (7c)         jr   $31
    assign inst = rom[a[6:2]];           // use word address to read rom
endmodule
```

The test data are stored in the data memory; see the following Verilog HDL code.

```
module scdatamem (clk,dataout,datain,addr,we);          // data memory, ram
    input         clk;                  // clock
    input         we;                   // write enable
    input  [31:0] datain;               // data in (to memory)
    input  [31:0] addr;                 // ram address
    output [31:0] dataout;              // data out (from memory)
    reg    [31:0] ram [0:31];           // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]];    // use word address to read ram
    always @ (posedge clk)
        if (we) ram[addr[6:2]] = datain; // use word address to write ram
    integer i;
    initial begin                       // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data        // (byte_addr) item in data array
        ram[5'h14] = 32'h000000a3;      // (50)  data[0]    0 +  A3 =  A3
        ram[5'h15] = 32'h00000027;      // (54)  data[1]   a3 +  27 =  ca
        ram[5'h16] = 32'h00000079;      // (58)  data[2]   ca +  79 = 143
        ram[5'h17] = 32'h00000115;      // (5c)  data[3]  143 + 115 = 258
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```

Figures 5.21 and 5.22 show the simulation waveforms of the single-cycle CPU when executing the test program.

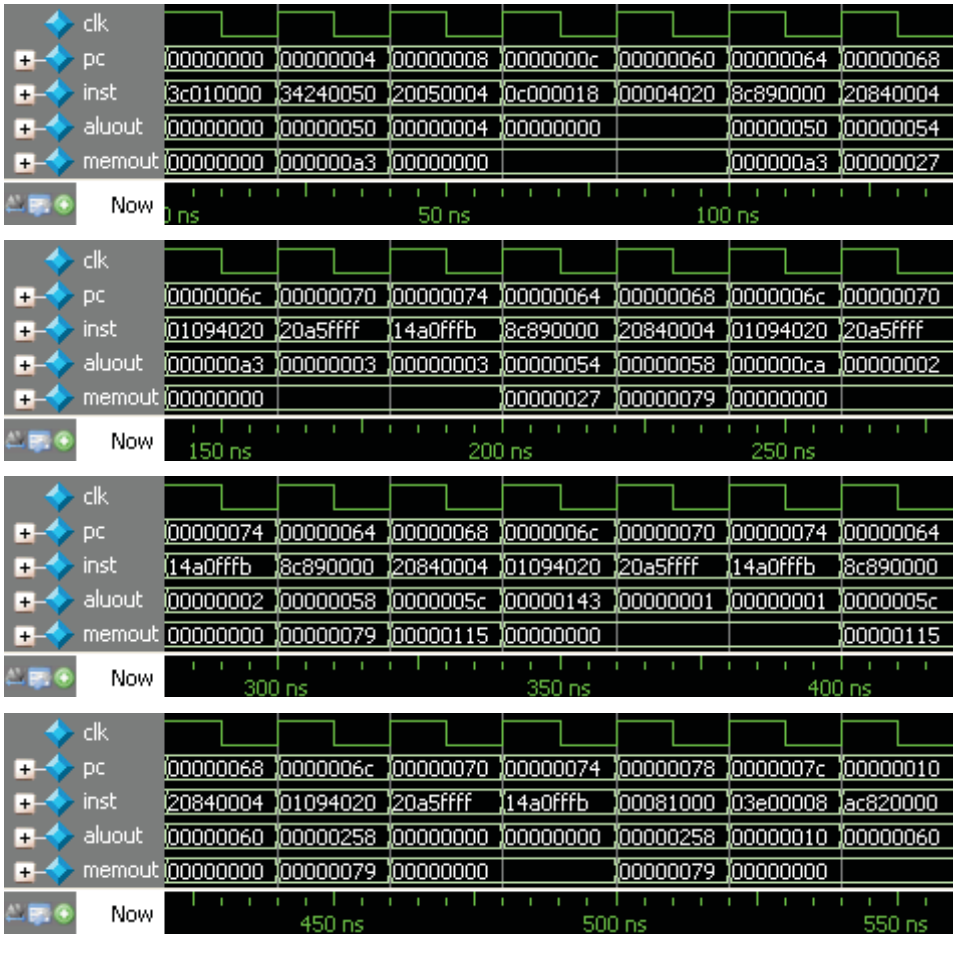

Figure 5.21 Waveform 1 of a single-cycle CPU

From the simulation waveforms, we know that our single-cycle CPU works correctly logically. The single-cycle CPU executes each instruction in one clock cycle. In the following chapters, we will design CPUs with different organizations.

Exercises

5.1 What is a single-cycle CPU? Explain its advantages and disadvantages.

5.2 Use behavioral-style Verilog HDL to design a single-cycle CPU. Hint: there is no need to have an ALU control signal; use `case` statement to deal with each instruction. The following code shows a design example of a single-cycle CPU which can execute three instructions: `add`, `lw`, and `beq`. Although the control signals are declared as `reg`, combinational circuits will be generated.

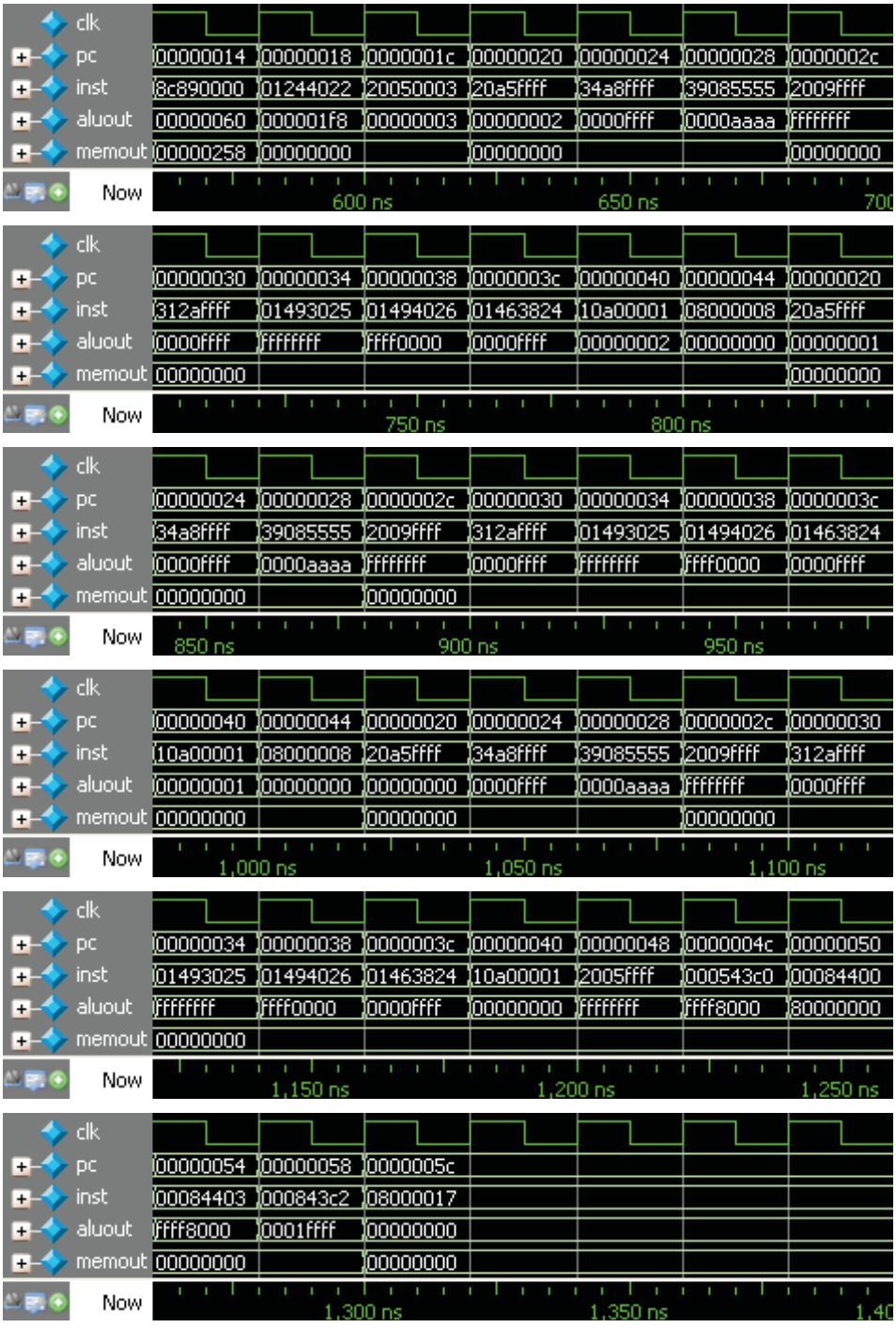

Figure 5.22 Waveform 2 of a single-cycle CPU

```
// internal signals (instruction format)
wire [05:00] opcode    = inst[31:26];
wire [04:00] rs        = inst[25:21];
wire [04:00] rt        = inst[20:16];
wire [04:00] rd        = inst[15:11];
wire [04:00] sa        = inst[10:06];
wire [05:00] func      = inst[05:00];
wire [15:00] imm       = inst[15:00];
wire [25:00] addr      = inst[25:00];
wire         sign      = inst[15];
wire [31:00] br_offset = {{14{sign}},imm,2'b00};
// instruction decode
wire i_add  = (opcode == 6'h00) & (func == 6'h20);
wire i_lw   = (opcode == 6'h23);
wire i_beq  = (opcode == 6'h04);
reg        wreg;    // write enable
reg [31:0] ALU_out; // ALU output
reg  [4:0] dest_rn; // destination register number
reg [31:0] next_pc; // next PC
// program counter
reg [31:0] pc;
always @ (posedge clk or negedge clrn) begin
    if (!clrn) pc <= 0;
    else pc <= next_pc;
end
// data written to register file
wire [31:0] data_to_regfile = i_lw ? d_f_mem : ALU_out;
// register file
reg  [31:0] regfile [1:31];
wire [31:0] a = (rs==0) ? 0 : regfile[rs];
wire [31:0] b = (rt==0) ? 0 : regfile[rt];
always @ (posedge clk) begin
    if (wreg && (dest_rn != 0)) begin
        regfile[dest_rn] <= data_to_regfile;
    end
end
// pc + 4
wire [31:0] pc_plus_4 = pc + 4;
// output signals
assign m_addr = ALU_out;
// control signals and ALU output
// will be combinational circuit
always @(*) begin
    ALU_out = 0;
    dest_rn = rd;
    wreg    = 0;
    next_pc = pc_plus_4;
```

```
        case (1'b1)
            i_add: begin // add
                ALU_out = a + b;
                wreg    = 1; end
            i_lw: begin // lw
                ALU_out = a + {{16{sign}},imm};
                dest_rn = rt;
                wreg    = 1; end
            i_beq: begin // beq
                if (a == b)
                  next_pc = pc_plus_4 + br_offset; end
            default: ;
        endcase
    end
```

5.3 Use Xilinx's BMG (block memory generator) or Altera's LPM (library of parameterized modules) to design the instruction memory and data memory. And simulate your CPU designed in Verilog HDL of behavioral style.

5.4 Design a single-cycle CPU that can execute all the MIPS integer instructions.