FPGA Systems Design and Practice (ET5009701)

Term Project

Date: September 7, 2022. **Instructor: M. B. Lin**

Due: December 14, 2022

This course has one term project. To accomplish this project, you have to use the **Xilinx ISE 14.7 WebPack**, which can be freely downloaded from the following Web site:

http://www.xilinx.com/support/download/index.htm

After downloading the file, you need to register for a free license. The FPGA device used is **Spartan6: XC6SLX25T-FGG484** or **Virtex-6: XC6VLX75T-FF484**.

For doing this project, the following two references are found much helpful.

1. David Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture*, 2nd Ed., Morgan Kaufmann, 2013. (Note that **any edition is fine.**)
2. David A. Patterson and John L. Hennessy, *Computer Organization and Design—The Hardware/-Software Interface*, 5th Ed., Morgan Kaufmann, 2014. (Note that **any edition is fine.**)

There are three common approaches that can be employed to design and implement this project. These include

- Single-cycle architecture
- Multiple-cycle architecture
- Pipeline architecture

You may choose any one of these to design and implement the project.

What about Reports

Your report must at least include the following parts:

- **Design:** This is a very important portion; you need to first complete this part and then continue the schematic and HDL entry parts and do the related behavioral (functional) and timing (gate-level) verification. **If you fail to do this, you will get a bad grade (score).** Regardless of which type of architecture you choose, your design must consist of two parts: the *datapath* and the *controller*.
- **Schematic/HDL Entry:** After you have done the design portion, enter your logic circuit schematically (in the midterm report) or in Verilog HDL (in the final report) and write test benches to verify your design with the ISE system. Please note that in the midterm report you only need to submit the schematic-entry part whereas in the final report the HDL-entry part. In the final report, the design must be the same as in the schematic entry and use the same test bench to test each corresponding module.
- **Verification:** For each module, both behavioral (RTL) and timing (post-route) simulations must be carried out and get the consistent results. You need to put the simulation results in your report.
- **Hardware Cost:** In each report, you also need to provide the total number of LUTs and FFs used in your complete system.
- **Discussion:** Give one or more paragraphs to state what problems you have encountered and how they are solved. Also, give something about what you have gotten during doing this project.

Specifications: A 16-Bit RISC Computer

In this project, you are asked to design and implement a 16-bit RISC computer. To design a processor, the most important thing is to design the desired *instruction set architecture* (ISA) along with the *programming model*. The programming model means the register set of the processor that can be accessed by its ISA. To illustrate this, we assume that the simple instruction set architecture listed in Table 1 is desired. The instructions listed in Tables 3 and 4 are optional to be implemented in this project. The processor is 16 bits with eight 16-bit general-purpose registers. For simplicity, the memory is assumed to be word-accessable; that is, the access unit is a word (16 bits). The *program counter* (PC) has 16 bits and hence can access up to 64-KW memory locations; that is, the address space is 65,536 words. *Note that in the current project, the xxx and yy parts in the instruction format can be simply ignored; they are reserved for the future extension.*

Table 1: The basic instruction set of the 16-bit RISC processor.

Mnemonic	Operation	N	Z	V	C	Instruction format
LHI Rd, #imm8	Rd ← {imm8, Rd[7:0]}; (imm8 = 0 to 255)	-	-	-	-	00001_ddd_iiiiiiii
LLI Rd, #imm8	Rd ← {8'h0, imm8}; (imm8 = 0 to 255)	-	-	-	-	00010_ddd_iiiiiiii
LDR Rd, [Rm, #imm5]	Rd ← Mem[Rm + imm5];	-	-	-	-	00011_dddmmm_iiiii
LDR Rd, [Rm, Rn]	Rd ← Mem[Rm + Rn];	-	-	-	-	00100_dddmmmnnn_00
STR Rd, [Rm, #imm5]	Mem[Rm + imm5] ← Rd;	-	-	-	-	00101_dddmmm_iiiii
STR Rd, [Rm, Rn]	Mem[Rm + Rn] ← Rd;	-	-	-	-	00110_dddmmmnnn_00
ADD Rd, Rm, Rn	Rd ← Rm + Rn;	*	*	*	*	00000_dddmmmnnn_00
ADC Rd, Rm, Rn	Rd ← Rm + Rn + C;	*	*	*	*	00000_dddmmmnnn_01
SUB Rd, Rm, Rn	Rd ← Rm - Rn;	*	*	*	*	00000_dddmmmnnn_10
SBB Rd, Rm, Rn	Rd ← Rm - Rn - $\overline{C}$;	*	*	*	*	00000_dddmmmnnn_11
CMP Rm, Rn	Rm - Rn;	*	*	*	*	00110_xxxmmmnnn_01
ADDI Rd, Rm, #imm5	Rd ← Rm + #imm5; (imm5 = 0 to 31)	*	*	*	*	00111_dddmmm_iiiii
SUBI Rd, Rm, #imm5	Rd ← Rm - #imm5; (imm5 = 0 to 31)	*	*	*	*	01000_dddmmm_iiiii
MOV Rd, Rm	Rd ← Rm;	-	-	-	-	01011_dddmmm_xxx_xx
BCC label	$\overline{C}$: PC ← PC + SignExtend(label);	-	-	-	-	1100_0011_disp8
BCS label	C: PC ← PC + SignExtend(label);	-	-	-	-	1100_0010_disp8
BNE label	$\overline{Z}$: PC ← PC + SignExtend(label);	-	-	-	-	1100_0001_disp8
BEQ label	Z: PC ← PC + SignExtend(label);	-	-	-	-	1100_0000_disp8
B[AL] label	PC ← PC + SignExtend(label);	-	-	-	-	1100_1110_disp8
JMP label	PC[10:0] ← label;	-	-	-	-	10000_label11
JAL Rd,label	Rd ← PC; PC ← PC + SignExtend(label);	-	-	-	-	10001_ddd_disp8
JAL Rd,Rm	Rd ← PC; PC ← Rm;	-	-	-	-	10010_dddmmm_xxx_yy
JR Rd	PC ← Rd;	-	-	-	-	10011_ddd_xxxxxxxx
OutR Rm	OutR ← Rm;	-	-	-	-	11100_xxx_mmm_xxx_00
HLT	Set done flag to 1 and halt CPU;	-	-	-	-	11100_xxxx_xxxxx_01

Note that the memory address in each of LDR and STR instructions corresponds to a word.

Both STR Rd, [Rm, Rn] and LDR Rd, [Rm, Rn] instructions are optional to be implemented in any architecture for simplicity. Nevertheless, it is instructive to consider how to implement them in any type of architecture and figure out their differences.

Programming Model In light of the ISA shown in Table 1, we may derive and draw the programming model as depicted in Figure 1. After reset, the program counter (PC) is cleared so that the processor begins to execute instructions at the memory address 0. Registers R0 to R7 are general-purpose registers

(GPRs). They may be used as data or address registers. All arithmetic operations are performed on these registers. Hence, memory operands must be loaded into registers before they can be operated. Nevertheless, memory operands can only be accessed by the use of LDR and STR instructions. Both LDR and STR instructions also use registers R0 to R7 as address registers to hold addresses. Hence, the computer is a GPR-based type with load-and-store structure. The *program-status word* (PSW) register memorizes the status of instruction execution. As indicated in Table 1, only arithmetic instructions may affect these flags. It consists of four basic flags:

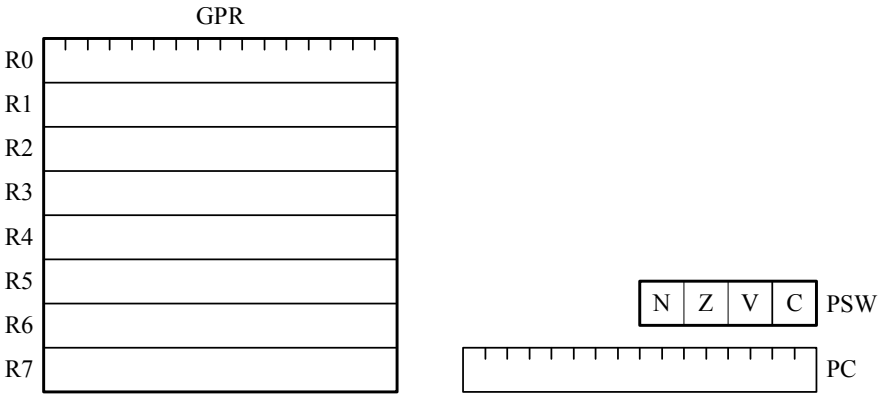

Figure 1: The programming model of the 16-bit RISC processor.

- N (negative flag): The N flag is set if the MSB (most-significant bit) of the result of execution of an arithmetic instruction, and is cleared otherwise. In other words, the value of the N flag is set to the MSB of the ALU output.
- Z (zero flag): The Z flag is set if the result of execution of an arithmetic instruction is zero and is cleared otherwise.
- C (carry flag): The C flag is set/clear if the MSB of the result of execution of an arithmetic instruction has a carry-out or borrow-out, and is cleared/set otherwise. In more precise, the C flag is just the carry-out that comes from the MSB of the two's-complement adder.
- V (overflow flag): The V flag indicates whether an overflow occurs in two's-complement arithmetic.

Basic Instruction Set The basic instruction set of the 16-bit RISC processor is summarized in Table 1 and includes:

- *Data transfer*: Data transfer instructions copy data from one register to another, immediate data into a register, and load a memory word from a specific memory location. These instructions do not affect the N, Z, V, and C flags. Data transfer instructions consist of two groups: data move and load/store.
  - *Move instructions:* The data move group includes three instructions: MOV, LHI, and LLI and has syntax as follows:

```
MOV  Rd,Rm
ADDI Rd,Rm,#0  (equivalent to MOV Rd,Rm)
SUBI Rd,Rm,#0  (equivalent to MOV Rd,Rm)
LHI  Rd,#imm8
LLI  Rd,#imm8
```

where `Rd` can be any register, `Rm` can be any register, and `#imm8` is an 8-bit immediate constant.

- *Load/store instructions:* This group of instructions facilitates the access of an operand from/to memory and has the syntax as follows

```
op  Rd,[Rm,#imm5]
op  Rd,[Rm,Rn]
```

where `op` can be LDR or STR and the memory location can be specified by immediate-offset (or called register-relative) or register-offset (or called base-index) addressing mode.

- *Arithmetic operations*: This group of instructions includes addition and subtraction and has the syntax of

```
op  Rd,Rm,Rn
op  Rm,Rn
```

All instructions in this group affect the N, Z, V, and C flags.

- *Shift and rotation operations*: (optional) Shift and rotation operations carry out shift (left and right) and rotation operations. They have the syntax

```
op  Rd,Rm,#imm3
```

All instructions in this group affect the N, Z, V, and C flags.

- *Logical operations*: (optional) Logical instructions carry out AND, OR, XOR, NOT, and TST operations in a bitwise way. They have the syntax

```
op  Rd,Rm,Rn
op  Rd,Rm
op  Rm,Rn
```

All instructions in this group affect the N, Z, V, and C flags.

- *Multiplication*: (optional) This group of instructions includes unsigned and signed multiplication and has the syntax of

```
op  Rd,Rm,Rn
op  Rd,Rm,#imm5
```

All instructions in this group affect the N, Z, V, and C flags.

- *Division*: (optional) This group of instructions includes unsigned and signed division and has the syntax of

```
op  Rd,Rm,Rn
op  Rd,Rm,#imm5
```

All instructions in this group affect the N, Z, V, and C flags.

- *Stack manipulation*: (optional) This group of instructions includes POP and PUSH and has the syntax of

```
POP  reglist
PUSH reglist
```

All instructions in this group do not affect the N, Z, V, and C flags.

- *Branch and jump*: This group contains conditional (Bcc) and unconditional branch (JMP) instructions and has the syntax

```
Bcc  label          ;8−bit displacement
JMP  label          ;11−bit label
```

  where `cc` is the combinations of condition codes, N, Z, V, and C flag statuses. For the current project, `cc` can only be CC (C = 0), CS (C = 1), NE (Z = 0), EQ (Z = 1), and [AL] (or omitted) (unconditional). The label in the `Bcc` instruction is an 8-bit displacement (here, the displacement means a 2's-complement value); thus, the branching range is from -128 to 127 words. In the `JMP` instruction, the label is 11 bits. This 11-bit label directly replaces the lowest significant 11 bits of the PC. Hence, it can make the program to jump to anywhere in the current 2k-word page.

- *Jump and link* (subroutine call and return): This group contains jump and link (JAL) and jump register (JR) instructions and has the syntax

```
JAL  Rd,label       ;8−bit displacement
JAL  Rd,Rm
JR   Rd
```

  This facilitates the subroutine call and return mechanism.

In addition, for the convenience of testing the processor, two new instructions, HLT (halt) and OutR (output register), are added to the ISA (see Table 1). The HLT instruction sets the *done* flag to 1 and then halts the CPU by freezing the clock. The *done* flag is cleared at reset. So you may put this instruction at the end of each test program to indicate the end of the program. The OutR Rm instruction facilitates the observation of the contents of registers and hence allows you to output the contents of any register to the OutR register.

Term-Project Report

In this term-project report, you need to address at least the following three parts: design block diagrams at the RTL, schematic entry, and HDL entry. In each of schematic- and HDL-entry methods, you also need to carry out both functional and timing simulations for each module in order to ensure that the module can work correctly in both functionality and timing.

I. Schematic Entry

Due: November 2, 2022

To accomplish this part, the following guidelines are listed for your reference.

• A 16-Bit Eight-Register Register File

1. Design and draw the logic diagram of an enabled-controlled 3-to-8 noninverting output decoder and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

2. Design and draw the logic diagram of an 8-to-1 multiplexer and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module. Note that you have the total freedom to design such a multiplexer.

3. Design and draw the logic diagram of a 16-bit 8-to-1 multiplexer by instantiating the 8-to-1 multiplexer and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

4. Design and draw the logic diagram of a 16-bit D-flip-flop register with clock-enable and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

5. Combine one 3-to-8 decoder, two 16-bit 8-to-1 multiplexers, and eight 16-bit registers into the desired register file. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• A 16-Bit ALU

1. Design and draw the logic diagram of a full adder and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

2. Combine sixteen full adders into a 16-bit adder. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

3. Combine the 16-bit adder with sixteen XOR gates into a 16-bit two's complement adder. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct.

4. Design the logic circuits related to N, Z, V, and C flags, and then combine them with the 16-bit two's complement adder into a 16-bit ALU. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• **A 16-Bit RF-plus-ALU** (This part may need to be modified so that it can fit your chosen architecture.)

1. Combine the 16-bit eight-register RF with the 16-bit ALU into a 16-bit RF-plus-ALU module. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• **A 256×16 Memory Module**

1. Design a 256×16 memory module by instantiating RAM 16×8s, RAM 32×8s, or other similar module and combining them together in a proper way. Draw the resulting logic diagram and then input it into the ISE system schematically. Of course, some additional logic modules, like multiplexers and/or decoders, may be also needed. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• **PC Circuitry**

1. Design and draw the logic diagram of the PC circuitry and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

Within the PC module, you may also need a simple two-complement adder to claculate the desired PC value in branch and JAL instructions.

• **A Complete Datapath Module**

1. Combine the 256×16 memory module, the PC circuitry, and the 16-bit RF-plus-ALU module into a complete datapath. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

Of course, the modules designed previously are only the major components. In constructing a complete datapath for the ISA in Table 1, you need to arrange these components appropriately and then add proper multiplexers along with other logic modules at the right places so that the datapath can accommodate the operations of all instructions. After this, all required modules are connected together to accomplish the desired datapath.

• **Timing Generator** (This module may not be needed depending on which architecture you chosen.)

1. Design and draw the logic diagram of a timing generator that generates the desired timing for the 16-bit RISC processor and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module. After this, create a symbol of this module.

• **Instruction Decoder**

1. Design and draw the logic diagram of the instruction decoder and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• **Complete Controller**

1. Combine the timing generator and instruction decoder into a complete controller and then input the resulting design into the ISE system schematically. Write a test bench to make sure that your design is correct. After this, create a symbol of this module.

• Complete Computer

Combine the datapath and the controller together into a complete 16-bit RISC in a proper way. Draw the resulting logic diagram and then input it into the ISE system schematically. Write a test bench to make sure that your design is correct. For your convenience, you may test only one program in each test bench. To ensure that your design and implementation are correct, at least the following programs have to be tested:

1. Find the minimum and maximum from two numbers in memory.
2. Add two numbers in memory and store the result in another memory location.
3. Add ten numbers in consecutive memory locations.
4. Mov a memory block of N words from one place to another.

It proves convenient to first write the above programs in assembly language and translated into their machine codes manually. Then, the resulting machine codes are embedded into the test bench so as to write into the memory of the 16-bit RISC computer.

An example of a test bench to test the complete 16-bit RISC computer.

```
// a test bench for the RISC_complete module
‘timescale 1ns / 1ps
module RISC_complete_tb;
// internal signals declarations
parameter clk_period = 20;
parameter delay_factor = 2;
reg clk, reset_n;
reg ext_we, test_normal;
reg [15:0] ext_addr, ext_data;
wire [15:0] mem_out, OutR;
wire done;
integer i;
// Unit Under Test port map
RISC_complete uut (
    .clk(clk), .reset_n(reset_n), .ext_we(ext_we),
    .test_normal(test_normal), .done(done),
    .ext_data(ext_data), .ext_addr(ext_addr),
    .mem_out(mem_out), .OutR(OutR));
// generate the clock signal
always begin
   #(clk_period/2) clk <= 1’b0;
   #(clk_period/2) clk <= 1’b1;
end
initial begin
   reset_n <=1’b0; test_normal = 1’b0;
   repeat (9) @(posedge clk)
              #(clk_period/delay_factor) reset_n <= 1’b0;
   reset_n <=1’b1; test_normal = 1’b1;
```

```
    write_mem(16'h0,16'b0001_0000_0010_0101);  // LLI R0,#25
    write_mem(16'h1,16'b0000_1000_0110_0011);  // LHI R0,#63
    write_mem(16'h2,16'b1110_0000_0000_0000);  // OUT R0 (6325H)
    write_mem(16'h3,16'b0001_1001_0000_0000);  // LDR R1,R0,#0
    write_mem(16'h4,16'b0001_1010_0000_0001);  // LDR R2,R0,#1
    write_mem(16'h5,16'b1110_0000_0010_0000);  // OUT R1 (47H)
    write_mem(16'h6,16'b1110_0000_0100_0000);  // OUT R2 (89H)
    write_mem(16'h7,16'b0000_0011_0010_1000);  // ADD R3,R1,R2
    write_mem(16'h8,16'b1110_0000_0110_0000);  // OUT R3 (D0H)
    write_mem(16'h9,16'b0000_0011_0010_1010);  // SUB R3,R1,R2
    write_mem(16'hA,16'b1110_0000_0110_0000);  // OUT R3 (FFBEH)
    write_mem(16'hB,16'b1110_0000_0000_0001);  // HLT
    write_mem(16'h25,16'h47); // data (25h, 47h)
    write_mem(16'h26,16'h89); // data (26h, 89h)
    // delay one clock to ensure the proper write to memory
    @(posedge clk) #(clk_period/delay_factor) ext_we = 1'b0;
    // read data from the dual-port memory
     test_normal = 1'b1;
    for (i = 0; i < 40; i = i + 1)
        @(posedge clk) #(clk_period/delay_factor) ext_addr = i;
     test_normal = 1'b0;
    // start the cpu to execute the program in memory
    reset_n = 1'b1;
    repeat (9) @(posedge clk)
                #(clk_period/delay_factor) reset_n = 1'b0;
    reset_n = 1'b1;
    wait (done);
end

task write_mem;
input [15:0] addr, data;
begin
    @(posedge clk) #(clk_period/delay_factor) begin
        test_normal = 1'b1;
        ext_we = 1'b1; ext_addr = addr;
        ext_data = data;
     end
end
endtask

initial #10000 $finish;
initial
    $monitor ($realtime,"ns %h %h %h %h %h %h %h %h %h \n",
              clk, reset_n, ext_we, test_normal, ext_addr,
              ext_data, mem_out, OutR, done);
endmodule
```

II. HDL Entry

Due: December 14, 2022

To accomplish this part, the following guidelines are listed for your reference.

• **A 16-Bit Eight-Register Register File**

1. Write a Verilog HDL module to describe an enabled-controlled 3-to-8 noninverting output decoder in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

2. Write a Verilog HDL module to describe a 16-bit 8-to-1 multiplexer directly in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

3. Write a Verilog HDL module to describe a 16-bit D-flip-flop register with clock-enable input directly in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

4. Write a Verilog HDL module to describe the desired register file consisting of a 3-to-8 decoder, two 16-bit 8-to-1 multiplexers, and eight 16-bit registers in structural style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **A 16-Bit ALU**

1. Write a Verilog HDL module to describe a full adder in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

2. Write a Verilog HDL module to describe the 16-bit ALU consisting of sixteen full adders and sixteen XOR gates in structural style and then input it into the ISE system. Also using **assign** continuous assignments, describe the required logic circuits for the N, Z, V, and C flags. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **A 16-Bit RF-plus-ALU** (This part may need to be modified so that it can fit your chosen architecture.)

1. Write a Verilog HDL module to describe the 16-bit RF-plus-ALU module by combining the 16-bit eight-register RF and the 16-bit ALU in structural style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **A 256×16 Memory Module**

1. Write a Verilog HDL module to describe the 256×16 memory module in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **PC Circuitry**

1. Write a Verilog HDL module to describe the PC circuitry in mixed or other style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

Within the PC module, you may also need a simple two-complement adder to claculate the desired PC value in branch and JAL instructions.

• **A Complete Datapath Module**

1. Write a Verilog HDL module to describe the complete datapath, consisting of the 256×16 memory module, the PC circuitry, and the 16-bit RF-plus-ALU module in structural style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

Of course, the modules designed previously are only the major components. In constructing a complete datapath for the ISA in Table 1, you need to arrange these components appropriately and then add proper multiplexers along with other logic modules at the right places so that the datapath can accommodate the operations of all instructions. After this, all required modules are connected together to accomplish the desired datapath.

• **Timing Generator** (This module may not be needed depending on which architecture you chosen.)

1. Write a Verilog HDL module to describe the timing generator in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **Instruction Decoder**

1. Write a Verilog HDL module to describe the instruction decoder in behavioral style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **Complete Controller**

1. Write a Verilog HDL module to describe the complete controller, consisting of the timing generator and instruction decoder, in structural style and then input it into the ISE system. Write a test bench to make sure that your design is correct. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

• **Complete Computer**

Write a Verilog HDL module to describe the complete 16-bit RISC computer by combining together the datapath and the controller in structural style and then input it into the ISE system. Write a test bench to make sure that your design is correct. For your convenience, you may test only one program in each test bench. To ensure that your design and implementation are correct, at least the following programs have to be tested:

1. Find the minimum and maximum from two numbers in memory.
2. Add two numbers in memory and store the result in another memory location.
3. Add ten numbers in consecutive memory locations.
4. Mov a memory block of N words from one place to another.

It proves convenient to first write the above programs in assembly language and translated into their machine codes manually. Then, the resulting machine codes are embedded into the test bench so as to write into the memory of the 16-bit RISC computer. *Note the test bench used here should be identical to that in the schematic counterpart. So actually the same test bench can be used here.*

Supplements — for your reference only

This supplement is only for your reference in the design and implementation of the 16-bit RISC computer. Of course, you may use your own method on condition that your final result is in consistent with the specifications of the 16-bit RISC computer.

Design Hints

To help you design and implement this simple 16-bit RISC computer, in what follows we give some design hints in addition to the examples lectured in the classroom for your reference.

Part 1: Datapath Design

In this part we give some useful hints about the design of the datapath of the 16-bit RISC processor.

A. The Structure of the Simple RISC16 Computer

The datapath is a shared resource for the instruction set. Its functions are centered around some registers and an ALU. Therefore, the major components of the datapath of any ISA (or processor) are a register file and an ALU along with some scratch registers and a few multiplexers used to route data to right places within the datapath. For a specified ISA, the datapath is usually not unique. To help you design a datapath, the major components to be used in the datapath of the underlying ISA are shown in Figure 2. They include a registered-in memory module, a register file, an arithmetic logic unit (ALU), a program counter (PC) module, and a few registers. To accomplish the datapath, you first need to design each of these modules in accordance with the requirements of the ISA defined in Table 1.

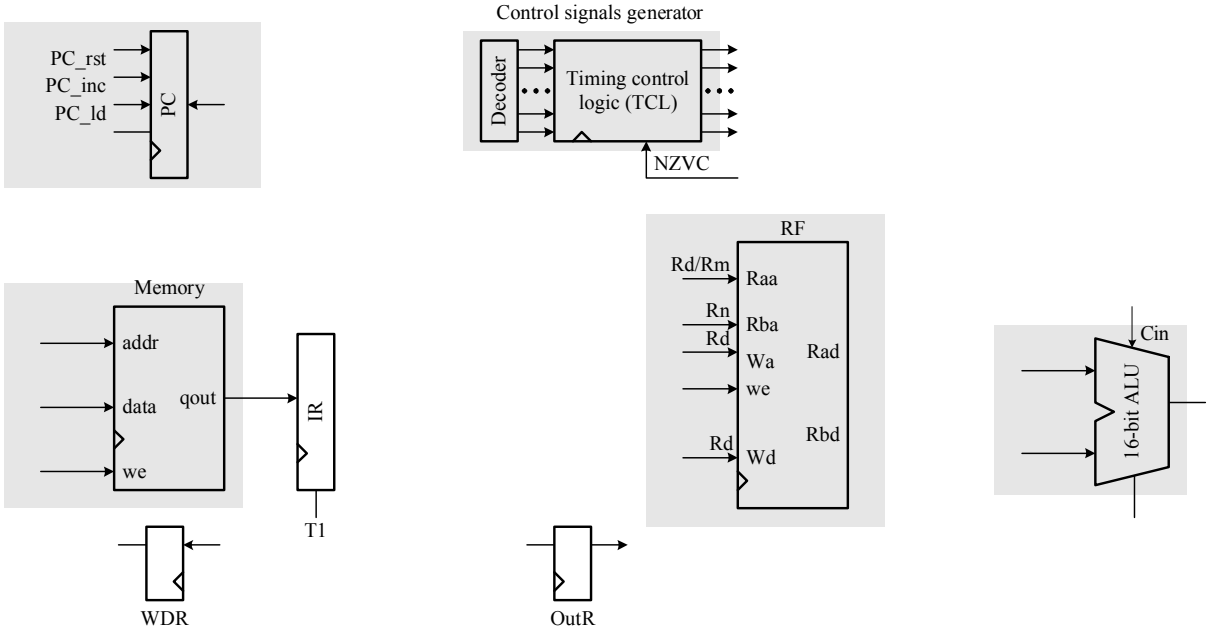

Figure 2: A simple, incomplete block diagram of the 16-bit RISC processor.

The register file (RF) must have three read ports in the single-cycle architecture and pipeline architecture. Otherwise, the STR Rd, [Rm, Rn] instruction could not be run in one clock cycle. As a consequence, both STR Rd, [Rm, Rn] and LDR Rd, [Rm, Rn] instructions can be optionally implemented in any architecture for simplicity.

Two memory modules are required in the single-cycle architecture and pipeline architecture, one for instructions and the other for data. The memory module depicted in Figure 7 is suggested to use in your project.

The WDR (write data register) is used only in the multiple-cycle architecture for implementing the STR Rd, [Rm, Rn] instruction. It is unnecessary in the other two types of architectures.

B. Memory module

Memory modules are vital in designing a digital system but unfortunately are difficult to be handled, especially, for the naive reader. Thus, in what follows we first consider four basic types of memory modules widely used in designing a digital system with FPGA devices.

Generally, memory models can be cast into the following four different types:

- In the *non-registered* (also referred to as *asynchronous* or non-registered-in, non-registered-out) model, both write and read ports are operated asynchronously. As depicted in Figure 3, when both data_in and addr appear at the inputs of the RAM and the write input is asserted at any time, the data_in will be written to the location of addr after a propagation delay of the RAM. Similarly, as the write input is deasserted at any time, the data at the memory location of addr will appear at the output, data_q, of the RAM after a propagation delay of the RAM.

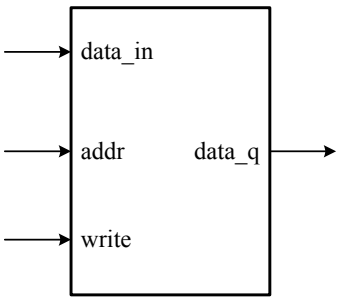

Figure 3: The general block diagram of a non-registered RAM module.

- In the *non-registered synchronous-write* (also referred to as *synchronous-write* or registered-in, non-registered-out) model, the write port is synchronous but the read port is asynchronous. As depicted in Figure 4, when both data_in and addr appear at the inputs of the RAM and the write input is asserted, the data_in will be written to the location of addr a propagation delay of the RAM after the active clock edge. Nevertheless, as the write input is deasserted at any time, the data at the memory location of addr will appear at the output, data_q, of the RAM after a propagation delay of the RAM.

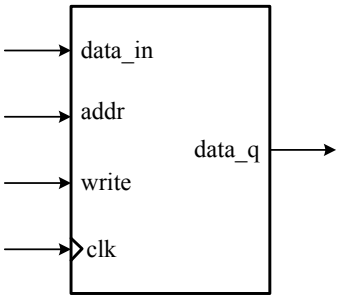

Figure 4: The general block diagram of a non-registered synchronous-write RAM module.

- In the *semi-registered* (or *registered-in*) model, the write port operates synchronously but the read port works asynchronously, meaning that the write operation is controlled by a clock signal. As depicted in Figure 5, data_wr, addr, and write signals appearing at the inputs of the RAM module are first captured by D flip-flops, and then applied to the asynchronous RAM module. On the contrary, as the write input is deasserted at any time, the data at the memory location of addr will appear at the output, data_q, of the RAM module after a propagation delay of the RAM module. It is instructive to note that in this model, the read address is first captured before it can access the memory.

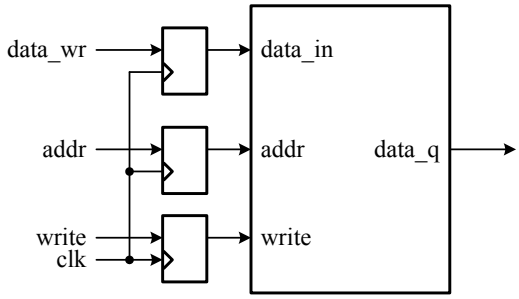

Figure 5: The general block diagram of a semi-registered RAM module.

- In the *full-registered* (or *registered-in and registered-out*) model, both write and read ports are operated synchronously, meaning that both operations are controlled by a clock signal. As depicted in Figure 6, data_in, addr, and write signals appearing at the inputs of the RAM module are first captured by D flip-flops, and then applied to the asynchronous RAM module. Similarly, as the write input is deasserted, the addr appearing at the input of the RAM module is first captured by D flip-flops, and then applied to the asynchronous RAM. After a propagation delay, the data at the memory location of addr will appear at the output, data_out, which is in turn captured by D flip-flops.

The memory module suggested to use. To test a memory module, the test bench also needs to have the access capability to the memory module, meaning that it can write and read the memory module exclusively with the normal operations of the memory module. An illustration of this is depicted in Figure 7, where one additional write port is added to the memory module. As the `port_sel` is set to 0, the test bench can write data into the memory module; as the `port_sel` is set to 1, the memory module is accessed by normal operations. The read port is asynchronous; namely, it can read the memory at any time without referring to the clock.

C. Register Files

In this section, we address the design and implementation of registers files. As mentioned, a register file can be constructed with a group of registers or memory modules. For convenience, we exemplify

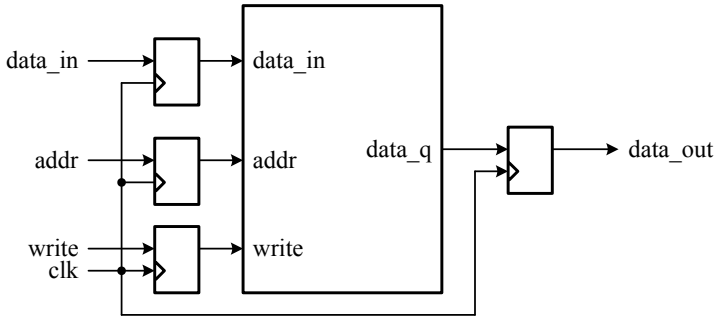

Figure 6: The general block diagram of a full-registered RAM module.

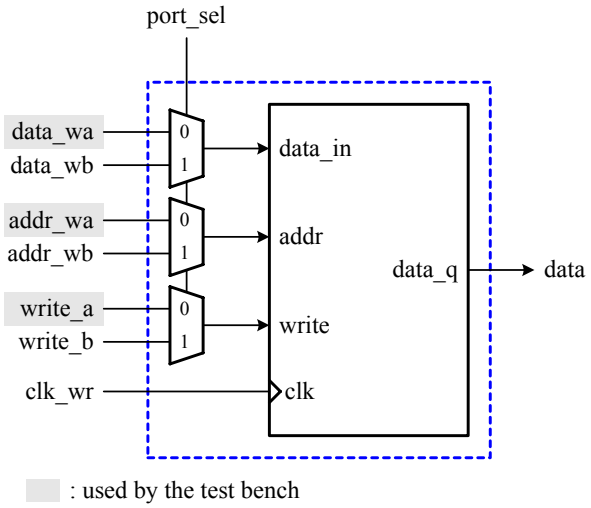

Figure 7: The general block diagram of a non-registered synchronous-write memory with 2-write and 1-read ports.

this with an n-bit four-register register file with one write port and two read ports in the register-based method.

Register-Based Register Files

As illustrated in Figure 8, to construct an n-bit four-register register file with one write port and two read ports, four n-bit registers with load control and one 2-to-4 decoder along with two n-bit 4-to-1 multiplexers are required. The 2-to-4 decoder receives the write address (*WR_addr*) and write enable (*WE*) signals and generates the *load* control signals to enable the load function of registers. Write data (*WR_data*) are connected to all the data inputs (*din*) of all four registers.

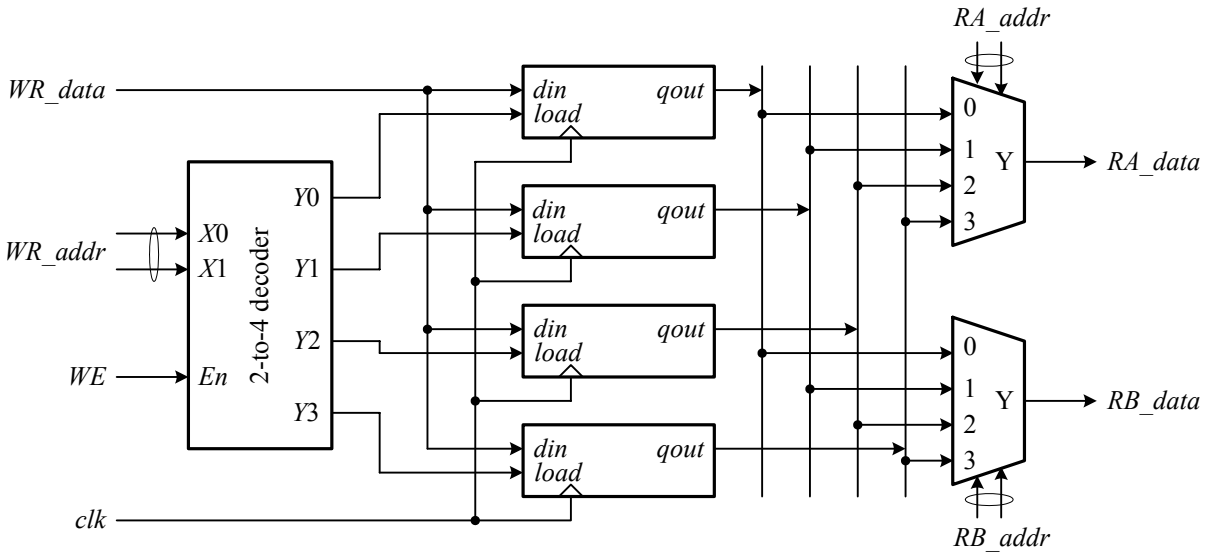

Figure 8: An example of n-bit four-register register file with one synchronous write port and two asynchronous read ports built on the basis of registers.

The data outputs (*qout*) of the four registers are routed to each of two output ports through an individual n-bit 4-to-1 multiplexer. The port A uses the port address, *RA_addr*, to select the desired register and the port data, *RA_data*, to output the register data. The port B uses the port address, *RB_addr*, to select the desired register and the port data, *RB_data*, to output the register data.

The main disadvantage of register-based register files is on the cost of output multiplexers, especially as the number of registers is large. The reader may verify this by estimating the required number of *logic elements* (LEs) or *lookup tables* (LUTs) from the hardware used in the multiplexers.

Memory-Based Register Files

In some FPGA devices, the construction of a memory module with one synchronous write port and one asynchronous read port, as illustrated in Figure 5, is quite easy and cost-effective. As this is the case, we may apply such a device to configure the desired register file. As illustrated in Figure 9, to provide one write port and two read ports, the write ports of two memory modules are connected together in parallel; that is, as a write operation is needed, the two memory modules are written to the same location simultaneously so as to maintain their data consistent at all times. The read port of each memory module is configured as a separate read port; thereby, two read ports can be independently accessed at any time. As a consequence, a register file with one synchronous write port and two asynchronous read ports is resulted.

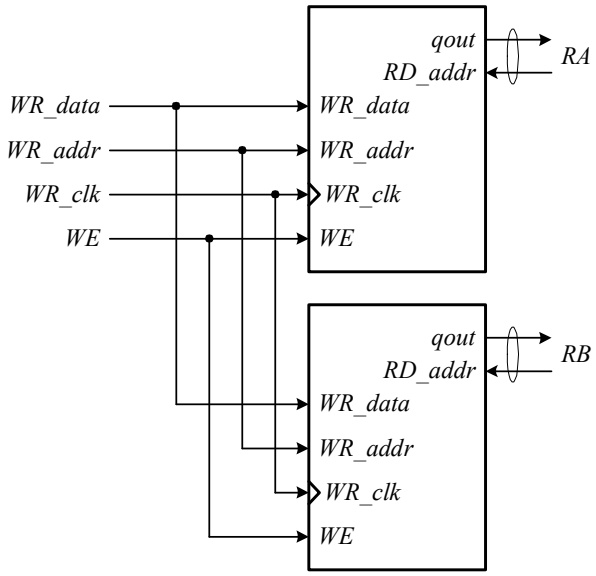

Figure 9: A register file with one synchronous write port and two asynchronous read ports constructed from two memory modules in Figure 5.

The memory-based register file often finds their use in FPGAs with RAM-based lookup tables as logic fabrics in which the lookup table can be readily converted into a RAM module. As compared to the register-based register file, the memory-based register file consumes much less hardware but has the main disadvantage that it is generally not portable.

Part 2: Controller Design

To construct the controller for the 16-bit RISC processor, we need to first refine the operations of each instruction into its RTL operations. The result is shown in Table 2. Of course, this table is incomplete. Its details are left for you to finish.

After finishing the details of RTL operations of each instruction, we may construct an ASM chart of the controller. The result is shown in Figure 10. Based on this ASM chart, we may further draw its ASMD chart of the datapath. But it and the other parts of the controller are left for you to accomplish.

Figure 10 and Table 2 are the sample ASM chart and the detailed operations of each instruction for the multiple-cycle architecture, respectivley. For the single-cycle architecture, the $T1$ state is not necessary, and the instruction output from memory is decoded directly by a

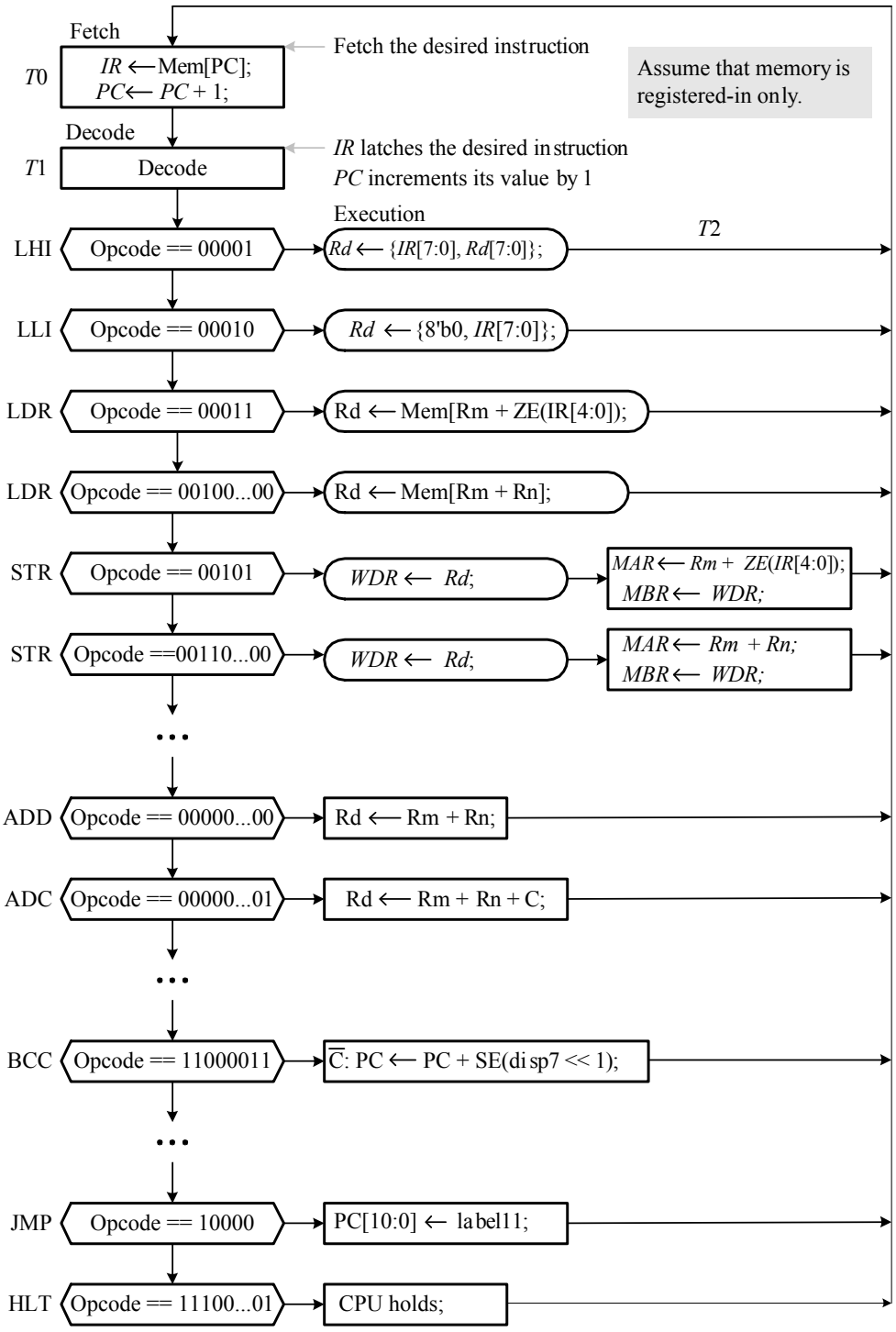

Figure 10: A reference ASM chart of controller of the 16-bit RISC processor.

Table 2: The detailed steps of instructions of the 16-bit RISC processor.

Mnemonic	Instruction format	RTL operation	Meaning
Fetch		T0: IR ← Mem[PC], PC ← PC + 1; T1: Decode;	Fetch an instruction from an asynchronous read RAM
LHI	00001_ddd_iiiiiiii	T1∧op==00001: Rd ← {IR[7:0], Rd[7:0]};	Load high-byte to Rd
LLI	00010_ddd_iiiiiiii	T1∧op==00010: Rd ← {8'b0, IR[7:0]};	Load low-byte to Rd
LDR	00011_dddmmm_iiiii	T1∧op==00011: Rd ←Mem[Rm + ZE(IR[4:0])];	Load Rd from memory
LDR	00100_dddmmmnnn_00	T1∧op==00100: Rd ← Mem[Rm + Rn];	Load Rd from memory
STR	00101_dddmmm_iiiii	T1∧op==00101: WDR ← Rd; T2: MAR ← Rm + ZE(IR[4:0]), MBR ← WDR;	Store Rd in memory
STR	00110_dddmmmnnn_00	T1∧op==00110: WDR ← Rd; T2: MAR ← Rm + Rn, MBR ← WDR;	Store Rd in memory
ADD	00000_dddmmmnnn_00	T1∧op==00000_x_00: Rd ← Rm + Rn;	Store Rm + Rn in Rd
ADC	00000_dddmmmnnn_01	T1∧op==00000_x_01: Rd ← Rm + Rn + C;	Store Rm + Rn in Rd
SUB	00000_dddmmmnnn_10	T1∧op==00000_x_10: Rd ← Rm - Rn;	Store Rm - Rn in Rd
SBB	00000_dddmmmnnn_11	T1∧op==00000_x_11: Rd ← Rm - Rn - $\overline{C}$;	Store Rm - Rn - $\overline{C}$; in Rd
CMP	00110_dddmmmnnn_01	T1∧op==00110_x_01: Flags ← Rm - Rn;	Set flags based on Rm - Rn
...			
BCC	1100_0011_disp8	T1∧op==1100_0011∧$\overline{C}$: PC ← PC + SE(disp8);	Branch to disp8 if C is 0
...			
B[AL]	1100_1110_disp8	T1∧op==1100_1110: PC ← PC + SE(disp8);	Branch to disp8 always
JMP	10000_label11	T1∧op==10000: PC ← {PC[15:11], label11};	Jump to label11 always
...			
HLT	11100_xxxx_xxxxx_01	T1∧op==11100_x_01: Set done flag to 1;	Halt the computer

☞ combinational logic circuit to yield the desired control signals. In addition, the $T2$ state is removed by the use of a three read-port register file. Notice that you need to check the ASM chart very carefully and to modify as well as complete it so as to fit into the architecture you want to use.

☞ Table 3 lists the extended instruction set of the 16-bit RISC processor. These instructions are optionally. In the term project, you are not asked to implement them. The condition code for the branch instruction Bcc is defined and listed in Table 4.

Table 3: The optionally extended instruction set of the 16-bit RISC processor.

Mnemonic	Operation	N	Z	V	C	Instruction format
LSR Rd, Rm, #imm3	Rd ← Rm >> imm3; (imm3 = 0 to 7)	*	*	-	*	10100_dddmmm_iii_00
ASR Rd, Rm, #imm3	Rd ← Rm >>> imm3; (imm3 = 0 to 7)	*	*	-	*	10100_dddmmm_iii_01
LSL Rd, Rm, #imm3	Rd ← Rm << imm3; (imm3 = 0 to 7)	*	*	-	*	10100_dddmmm_iii_10
ASL Rd, Rm, #imm3	Rd ← Rm << imm3; (imm3 = 0 to 7)	*	*	-	*	10100_dddmmm_iii_11
ROR Rd, Rm, #imm3	Rd ← Right rotate{Rm} imm3 times;	*	*	-	*	10101_dddmmm_iii_00
ROL Rd, Rm, #imm3	Rd ← Left rotate{Rm} imm3 times;	*	*	-	*	10101_dddmmm_iii_01
RORC Rd, Rm, #imm3	{C, Rd} ← {Rm, C} imm3 times;	*	*	-	*	10101_dddmmm_iii_10
ROLC Rd, Rm, #imm3	{Rd, C} ← {C, Rm} imm3 times;	*	*	-	*	10101_dddmmm_iii_11
LSR Rd, Rm, Rn	Rd ← Rm >> Rn; (Rn = 0 to 7)	*	*	-	*	10110_dddmmm_nnn_00
ASR Rd, Rm, Rn	Rd ← Rm >>> Rn; (Rn = 0 to 7)	*	*	-	*	10110_dddmmm_nnn_01
LSL Rd, Rm, Rn	Rd ← Rm << Rn; (Rn = 0 to 7)	*	*	-	*	10110_dddmmm_nnn_10
ASL Rd, Rm, Rn	Rd ← Rm << Rn; (Rn = 0 to 7)	*	*	-	*	10110_dddmmm_nnn_11
ROR Rd, Rm, Rn	Rd ← Right rotate{Rm} Rn times;	*	*	-	*	10111_dddmmm_nnn_00
ROL Rd, Rm, Rn	Rd ← Left rotate{Rm} Rn times;	*	*	-	*	10111_dddmmm_nnn_01
RORC Rd, Rm, Rn	{C, Rd} ← {Rm, C} Rn times;	*	*	-	*	10111_dddmmm_nnn_10
ROLC Rd, Rm, Rn	{Rd, C} ← {C, Rm} Rn times;	*	*	-	*	10111_dddmmm_nnn_11
AND Rd, Rm, Rn	Rd ← Rm ∧ Rn;	*	*	-	0	00100_dddmmmnnn_01
OR Rd, Rm, Rn	Rd ← Rm ∨ Rn;	*	*	-	0	00100_dddmmmnnn_10
XOR Rd, Rm, Rn	Rd ← Rm ⊕ Rn;	*	*	-	0	00100_dddmmmnnn_11
NOT Rd, Rm	Rd ← NOT Rm; (imm3 = 0)	*	*	-	0	00110_dddmmm_000_10
TST Rm, Rn	Rm ∧ Rn;	*	*	-	0	00110_xxxmmmnnn_11
MUL Rd, Rm, Rn	Rd ← Rm × Rn;	*	*	*	*	11010_dddmmm_nnn_00
MULI Rd, Rm, #imm5	Rd ← Rm × #imm5; (imm5 = 0 to 31)	*	*	*	*	11011_dddmmm_iiiii
SMUL Rd, Rm, Rn	Rd ← Rm × Rn;	*	*	*	*	11010_dddmmm_nnn_01
SMULI Rd, Rm, #imm5	Rd ← Rm × #imm5; (imm5 = -16 to 15)	*	*	*	*	11100_dddmmm_iiiii
DIV Rd, Rm, Rn	Rd ← Rm ÷ Rn;	*	*	*	*	11010_dddmmm_nnn_10
DIVI Rd, Rm, #imm5	Rd ← Rm ÷ #imm5; (imm5 = 0 to 31)	*	*	*	*	11101_dddmmm_iiiii
SDIV Rd, Rm, Rn	Rd ← Rm ÷ Rn;	*	*	*	*	11010_dddmmm_nnn_11
SDIVI Rd, Rm, #imm5	Rd ← Rm ÷ #imm5; (imm5 = -16 to 15)	*	*	*	*	11110_dddmmm_iiiii
PUSH Rm	SP ← SP + 1, Mem[SP] ← Rm;	-	-	-	-	11100_111_mmmxxx_10
POP Rd	Rd ← Mem[SP], SP ← SP - 1;	-	-	-	-	11100_dddx_xx111_11
POP reglist	{reglist} ← Stack;	-	-	-	-	01100_xxx_rrrrrrrr
PUSH reglist	Stack ← {reglist};	-	-	-	-	01101_xxx_rrrrrrrr
Bcond disp8	*cond*: PC ← PC + SignExtend(disp8);	-	-	-	-	1100_cccc_disp8

Note that R7 is defaulted to be the SP. In implementing the POP instruction, an extra 2-to-1 multiplexer is needed to select the write address of the RF since two registers, Rd and R7, are required to write in this instruction.

Table 4: Condition code summary.

Type	Symbol	If...then	Code	Condition
Single bit	CC	Carry clear	0011	C = 0
	CS	Carry set	0010	C = 1
	PL	Positive or zero	0101	N = 0
	MI	Negative	0100	N = 1
	NE	Not equal	0001	Z = 0
	EQ	Equal	0000	Z = 1
	VC	No overflow	0111	V = 0
	VS	Overflow	0110	V = 1
Signed number	LT	$<$ (less than)	1011	N $\oplus$ V = 1
	LE	$\leq$ (less than or equal)	1101	(N $\oplus$ V) $\vee$ Z = 1
	GT	$>$ (greater than)	1100	(N $\oplus$ V) $\vee$ Z = 0
	GE	$\geq$ (greater than or equal)	1010	N $\oplus$ V = 0
Unsigned number	HI	$>$ (higher)	1000	$\bar{\text{C}}$ $\vee$ Z = 0
	LS	$\leq$ (lower or same)	1001	$\bar{\text{C}}$ $\vee$ Z = 1
	LO	$<$ (lower)	0011	C = 0
	HS	$\geq$ (higher or same)	0010	C = 1
	$\{$AL$\}$	Always (this suffix is normally omitted.)	1110	None tested