FPGA Systems Design and Practice (ET5009701)

Term Project

Date: September 11, 2019. **Instructor: M. B. Lin**

Due: January 8, 2020

This course has one term project. To accomplish this project, you have to use the **Xilinx ISE 14.7 WebPack**, which can be downloaded freely 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 **Spartan 3: XC3S400A-4FT400**.

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 Table 2 are optional to be implemented in this project. The processor is 16 bits with eight 16-bit general-purpose registers. The *program counter* (PC) has 16 bits and hence can access up to 64-kB memory locations; that is, the address space is 65,536 bytes. *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.*

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 of 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. 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. It consists of four basic flags:

- N (negative flag): The N flag is set if the MSB of the result after an instruction executes is 1, and is cleared otherwise. In other words, the value of the N flag is set to the MSB of the result.
- Z (zero flag): The Z flag is set if the result after an instruction executes is zero and is cleared otherwise.
- C (carry flag): The C flag is set if the MSB (most-significant bit) of the result after an instruction executes has a carry-out or borrow-out, and is cleared otherwise.
- V (overflow flag): The V flag indicates whether an overflow occurs in two's complement arithmetic.

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

Mnemonic	Operation	N Z V C	Instruction format
LHI Rd, #imm8	Rd[15:8] ← #imm8; (imm8 = 0 to 255)	- - - -	00001_ddd_iiiiiiii
LLI Rd, #imm8	Rd[7:0] ← #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 - 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_yy
BCC label	$\overline{C}$: PC ← PC + SignExtend(label);	- - - -	11000_0011_disp7
BCS label	C: PC ← PC + SignExtend(label);	- - - -	11000_0010_disp7
BNE label	$\overline{Z}$: PC ← PC + SignExtend(label);	- - - -	11000_0001_disp7
BEQ label	Z: PC ← PC + SignExtend(label);	- - - -	11000_0000_disp7
B[AL] label	PC ← PC + SignExtend(label);	- - - -	11000_1110_disp7
JMP label	PC[10:0] ← label;	- - - -	10000_label11
JAL Rd,label	Rd ← PC; PC[7:0] ← label8;	- - - -	10001_ddd_label8
JAL Rd,Rm	Rd ← PC; PC ← Rm;	- - - -	10010_dddmmm_xxx_yy
JR Rd	PC ← Rd;	- - - -	10011_ddd_xxxxxxxx
HLT	Set done flag to 1 and halt CPU;	- - - -	11100_xxxx_xxxxx_00
OutR Rm	OutR ← Rm;	- - - -	11100_xxx_mmm_xxx_01

Note that both imm8 and imm5 are in units of words.

Memory Organization The memory organization of the 16-bit RISC processor is shown in Figure 2. For the current project, it can be either byte (8 bits) addressable or word (16 bits) addressable. As it is byte addressable, all bits of the memory address and the PC value are applied to the memory. As it is word addressable, only the higher 15 bits of the memory address and the PC value are applied to the memory; the LSB is left unconnected. Regardless of which type of memory, the memory data bus is 16 bits, and the PC value is incremented by 2 after fetching an instruction.

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.

Table 2: The extended instruction set of the 16-bit RISC processor.

Mnemonic	Operation	N	Z	V	C	Instruction format
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
LSR Rd, Rm, #imm3	Rd ← Rm >> imm3; (imm3 = 0 to 7)	*	*	*	*	01001_dddmmm_iii_00
ASR Rd, Rm, #imm3	Rd ← Rm >>> imm3; (imm3 = 0 to 7)	*	*	*	*	01001_dddmmm_iii_01
ASL Rd, Rm, #imm3	Rd ← Rm << imm3; (imm3 = 0 to 7)	*	*	*	*	01001_dddmmm_iii_10
LSL Rd, Rm, #imm3	Rd ← Rm << imm3; (imm3 = 0 to 7)	*	*	*	*	01001_dddmmm_iii_10
ROL Rd, Rm, #imm3	{Rd} ← Left rotate{Rm} imm3 times;	*	*	*	*	01010_dddmmm_iii_00
ROR Rd, Rm, #imm3	{Rd} ← Right rotate{Rm} imm3 times;	*	*	*	*	01010_dddmmm_iii_01
ROLC Rd, Rm, #imm3	{Rd, C} ← {C, Rm} imm3 times;	*	*	*	*	01010_dddmmm_iii_10
RORC Rd, Rm, #imm3	{C, Rd} ← {Rm, C} imm3 times;	*	*	*	*	01010_dddmmm_iii_11
AND Rd, Rm, Rn	Rd ← Rm ∧ Rn;	*	*	*	*	00100_dddmmmnnn_01
OR Rd, Rm, Rn	Rd ← Rm ∨ Rn;	*	*	*	*	00100_dddmmmnnn_10
XOR Rd, Rm, Rn	Rd ← Rm ⊕ Rn;	*	*	*	*	00100_dddmmmnnn_11
NOT Rd, Rm	Rd ← NOT Rm; (imm3 = 0)	*	*	*	*	00110_dddmmm_000_10
TST Rm, Rn	Rm ∧ Rn;	*	*	*	*	00110_xxxmmmnnn_11
POP reglist	{reglist} ← Stack;	-	-	-	-	01100_xxx_rrrrrrrr
PUSH reglist	Stack ← {reglist};	-	-	-	-	01101_xxx_rrrrrrrr

Note that both imm8 and imm5 are in units of words.

– *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.

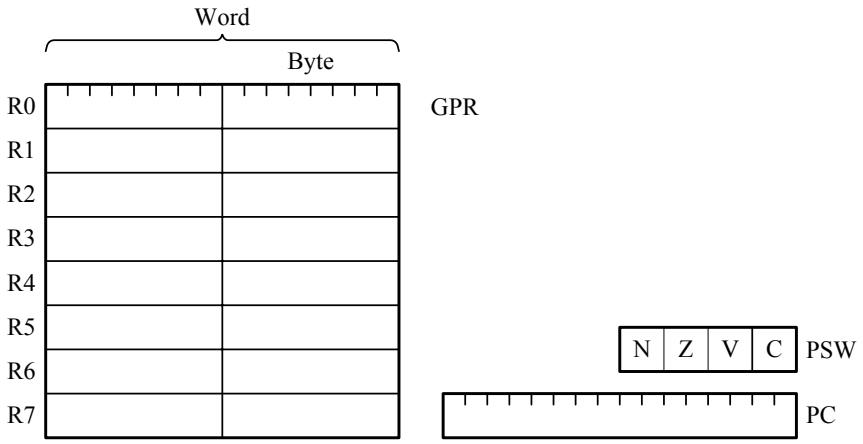

Figure 1: The programming model of the 16-bit RISC processor.

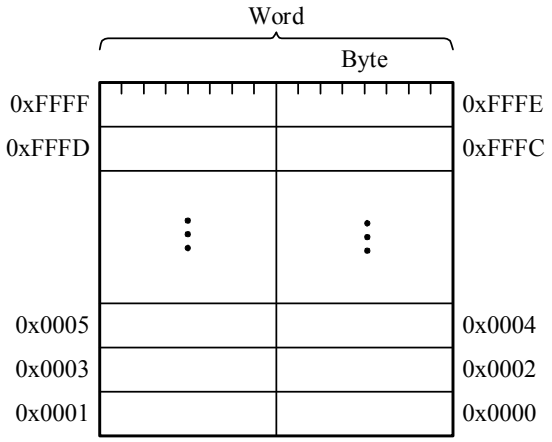

Figure 2: The memory model of the 16-bit RISC processor.

- *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
JMP  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 a 7-bit whereas in the `JMP` instruction is a 11-bit displacement (a 2's-complement value). Before they are added to the PC, they are shifted left one-bit position (aligned to word boundary). Thus, in the `Bcc` instruction, the branching range is from -256 to 254 words while in the `JMP` instruction, the branching range is from -2048 to 2046.

- *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
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, which is cleared at reset, to 1 and then halts the CPU by freezing the clock. 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 so as to verify the module that works correctly in both functionality and timing.

I. Schematic Entry

Due: November 13, 2019

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**

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.

- **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.

- **Timing Generator**

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.

II. HDL Entry

Due: January 8, 2020

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.*

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.*

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.*

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.*

• **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.*

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. Connect a multiplexer to *Cin* in order to route the proper input to the *Cin* input. 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.*

• **A 16-Bit RF-plus-ALU**

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.*

• **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.*

• **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.*

• **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.*

• **Timing Generator**

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.*

• **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.*

• **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.*

• **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.*

Supplements

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 major components of the datapath of any CPU 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. A possible datapath of the underlying ISA is shown in Figure 3. It consists of a non-registered synchronous-write memory module and a register file along with an arithmetic logic unit (ALU). In this section, we will briefly introduce how to derive this datapath from the underlying ISA.

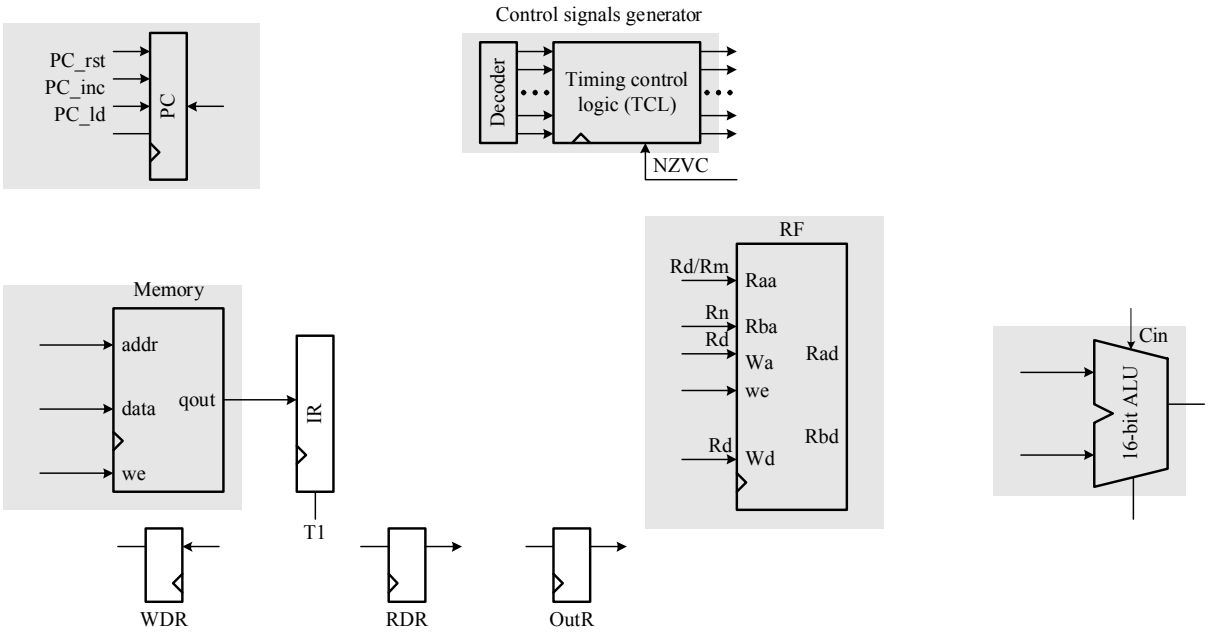

Figure 3: A simple, incomplete block diagram of the 16-bit RISC processor.

The datapath is a shared resource for the instruction set. Its functions are centered around some registers and an ALU. Therefore, the most straightforward way to obtain a datapath from an ISA is *first to consider its core parts, the register file (RF) and the ALU, and then to explore the associated routing paths through the use of multiplexers.* Nevertheless, before proceeding, we would like to deal with the memory module first.

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*) model, both write and read ports are operated asynchronously. As depicted in Figure 4, 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.

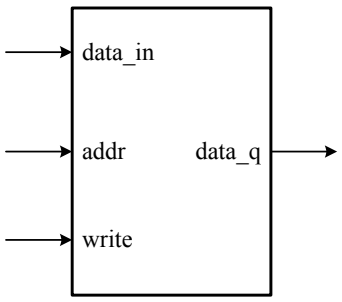

Figure 4: The general block diagram of a non-registered RAM module.

- In the *non-registered synchronous-write* (also referred to as *synchronous-write*) model, the write port is synchronous but the read port is asynchronous. As depicted in Figure 5, 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.

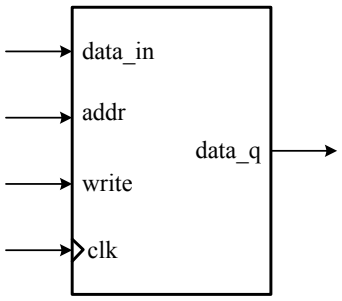

Figure 5: The general block diagram of a non-registered synchronous-write RAM module.

- 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 6, 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.

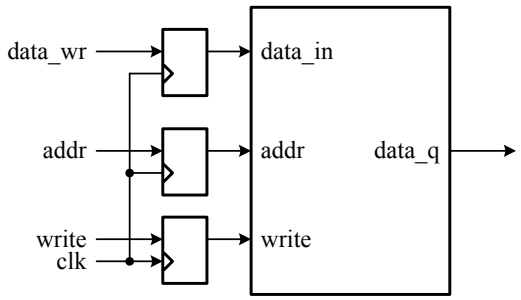

Figure 6: The general block diagram of a semi-registered RAM module.

- 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 7, 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.

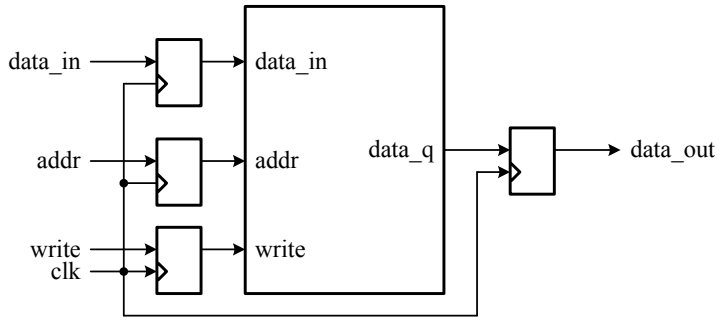

Figure 7: The general block diagram of a full-registered RAM module.

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 8, 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 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 9, 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

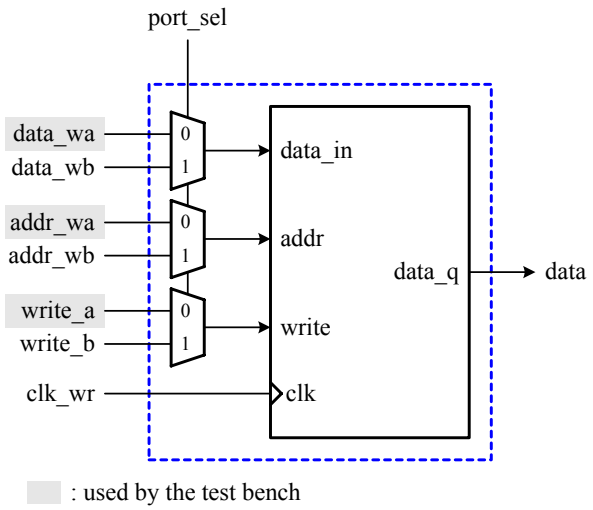

Figure 8: The general block diagram of a non-registered synchronous-write memory with 2-write and 1-read ports.

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.

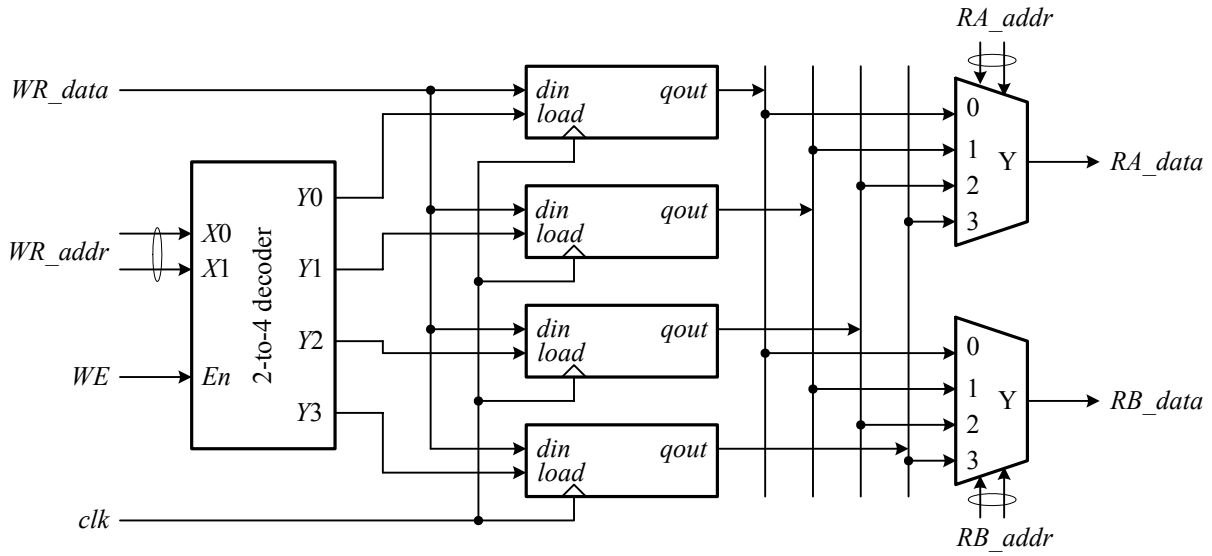

Figure 9: An example of n-bit four-register register file with one synchronous write port and two asynchronous read ports built on the basis of registers.

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 6, 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 10, 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.

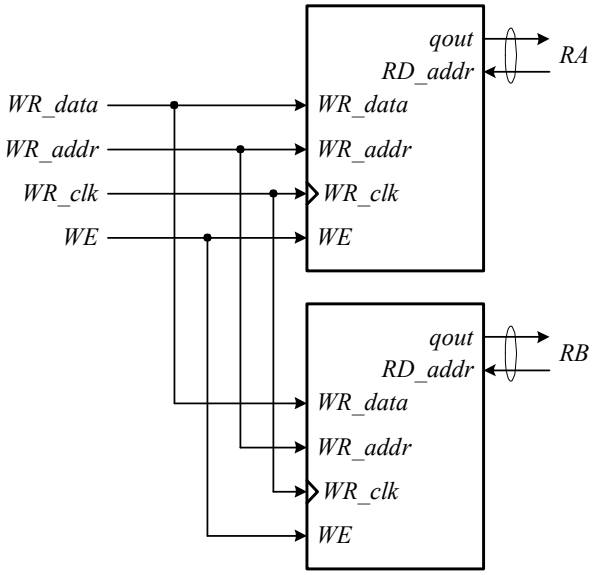

Figure 10: A register file with one synchronous write port and two asynchronous read ports constructed from two memory modules in Figure 6.

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.

D. An RF-Plus-ALU Circuit

A register file is often combined with an arithmetic logic unit (ALU) to perform the desired computations like the following ones:

$$\begin{aligned} R_i &\longleftarrow R_j + R_k; \qquad \text{for all} \quad 0 \le i, j, k \le 3 \\ R_i &\longleftarrow R_j - R_k; \end{aligned} \tag{1}$$

Of course, in practice digital systems, the arithmetic operations are not limited to addition and subtraction. Many others can also be included as needed. A generic combination of a register file and an ALU is depicted in Figure 11. To make the combination more useful, the register file also has to receive the external data in addition to the output from the ALU. Hence, three 2-to-1

multiplexers are used to double the number of write ports of the register file. These multiplexers are selected by the *test_normal* signal. As the *test_normal* signal is set to 1, external data can be written into the desired register. As the *test_normal* signal is set to 0, the register file along with the ALU operates normally, that is, carrying out Equation (1).

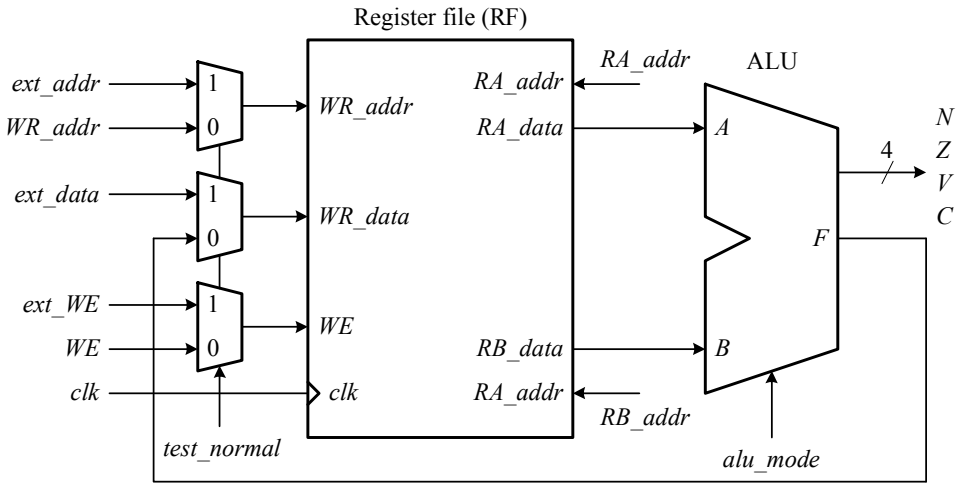

Figure 11: An RF in connection with an ALU.

Logic Circuit of the C Flag.

From Table 1, we know that both instructions ADC and SBB use the C flag as the linkage to their previous instruction to form a double- or multiple-precision operation. As the C flag is 1, the previous instruction has a carry-out or borrow-out; as the C flag is 0, the previous instruction does not have a carry-out or borrow-out. Nevertheless, in 2's complement arithmetic as the C flag is 1, the previous operation has a carry-out in addition and no borrow-out in subtraction. As the C flag is 0, the previous operation does not have a carry-out in addition but has a borrow-out in subtraction. Based on this, in hardware implementation, the carry-in input of the 2's complement adder must be cleared to 0 if the previous operation has a borrow-out. Hence, a possible logic circuit of the C flag in the 2's complement adder is depicted in Figure 12.

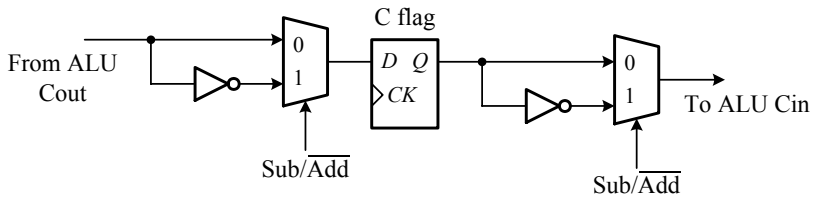

Figure 12: The logic circuit of the C flag in the ALU.

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 3. Of course, this table is incomplete. Its details are left for you to finish.

Table 3: The detailed steps of instructions of the 16-bit RISC processor.

Mnemonic	Instruction format	RTL operation	Meaning
Fetch		T0: IR ← Mem[PC], PC ← PC + 2; T1: Decode;	Fetch an instruction from an asynchronous read RAM
LHI	00001_ddd_iiiiiiii	RDR ← Mem[7:0][PC]; T3: Rd ← RDR, PC ← PC + 2;	Load high-byte Rd from memory
LLI	00010_ddd_iiiiiiii	RDR ← Mem[7:0][PC]; T3: Rd ← RDR, PC ← PC + 2;	Load high-byte Rd from memory
LDR	00011_dddmmm_iiiii	T2: RDR ← Mem[Rm + ZE(imm5<<1)]; T3: Rd ← RDR;	Load Rd from memory
LDR	00100_dddmmmnnn_00	T2: RDR ← Mem[Rm + Rn]; T3: Rd ← RDR;	Load Rd from memory
STR	00101_dddmmm_iiiii	T2: WDR ← Rd; T3: Mem[Rm + ZE(imm5<<1)] ← WDR;	Store Rd in memory
STR	00110_dddmmmnnn_00	T2: WDR ← Rd; T3: Mem[Rm + Rn] ← WDR;	Store Rd in memory
ADD	00000_dddmmmnnn_00	T2: Rd ← Rm + Rn;	Store Rm + Rn in Rd
ADC	00000_dddmmmnnn_01	T2: Rd ← Rm + Rn + C;	Store Rm + Rn in Rd
SUB	00000_dddmmmnnn_10	T2: Rd ← Rm - Rn;	Store Rm - Rn in Rd
SBB	00000_dddmmmnnn_11	T2: Rd ← Rm - Rn - C;	Store Rm - Rn - C in Rd
CMP	00110_dddmmmnnn_01	T2: Flags ← Rm - Rn;	Set flags based on Rm - Rn
...			
BCC ...	11000_0011_disp7	T2∧ $\overline{C}$: PC ← PC + SE(disp7<<1);	Branch to disp7 if C is 0
B[AL]	11000_1110_disp7	T2: PC ← PC + SE(disp7<<1);	Branch to disp7 always
JMP ...	10000_disp11	T2: PC[10:0] ← PC + SE(disp11<<1);	Jump to disp11 always
HLT	11100_xxxx_xxxxxxx	T2: Set done flag to 1 and halt CPU;	Halt the computer

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 13. 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.

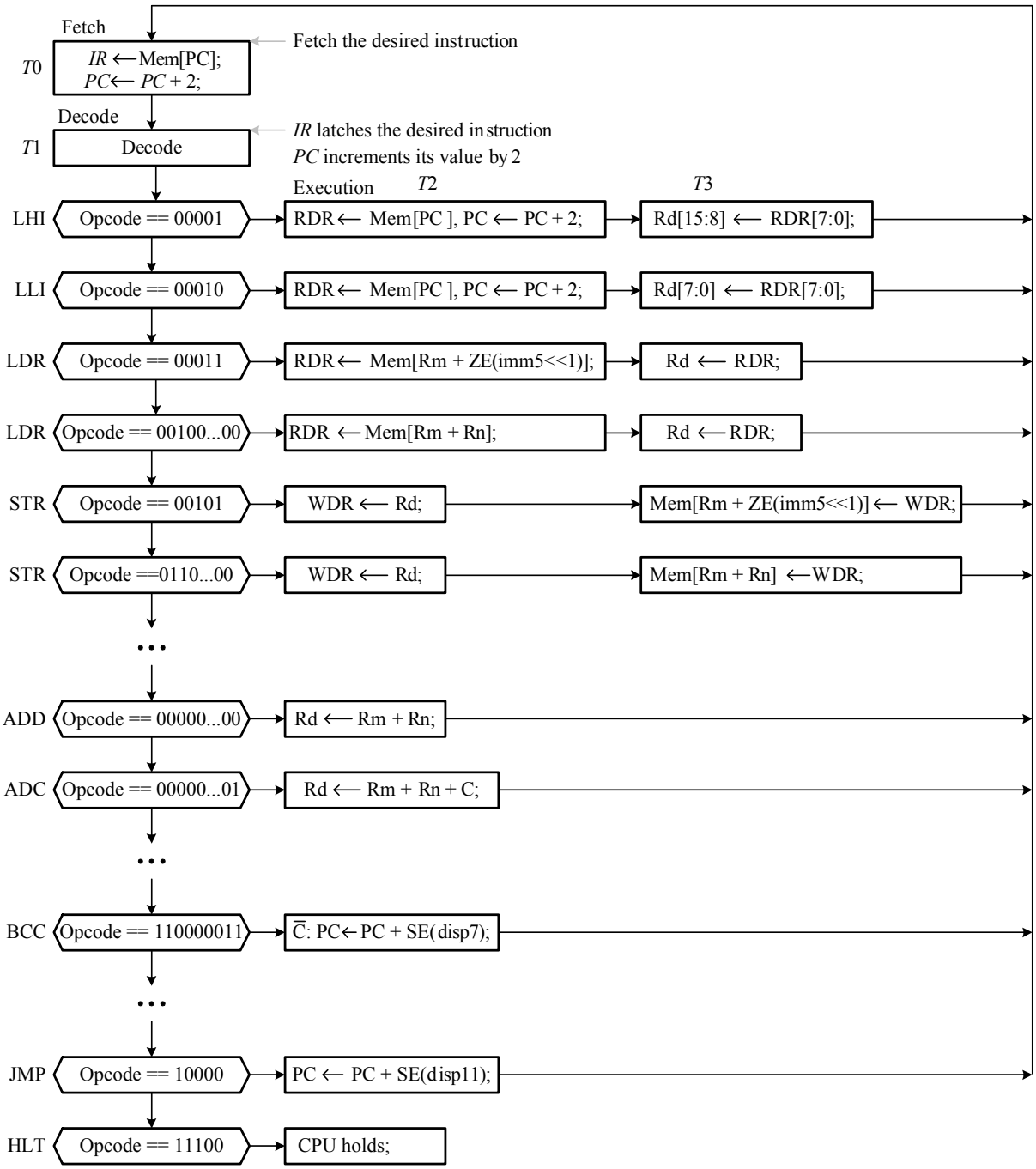

Figure 13: A reference ASM chart of controller of the 16-bit RISC processor.