ENCM 369 Winter 2023 Lab 7 (corrected) for the Week of March 6

Steve Norman
Department of Electrical & Software Engineering
University of Calgary

March 2023

Administrative details

You may work with a partner on this assignment

If you choose to work with a partner, please make sure that both partners fully understand all parts of your assignment submission, and please follow the instructions regarding submission of your PDF document.

Partners must be in the same lab section. The reason for this rule to keep teaching assistant workloads balanced and to make it as easy as possible for teaching assistants to record marks.

Due Dates

The Due Date for this assignment is 6:00pm Friday, March 10.
The Late Due Date is 6:00pm Monday, March 13.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will not be marked.

How to package and hand in your assignments

You must submit your work as a *single PDF file* to the D2L dropbox that will be set up for this assignment. The dropbox will be configured to accept only file per student, but you may replace that file as many times as you like before the due date.

See the Lab 1 instructions for more information about preparing and uploading your PDF file.

Important update for those working with a partner: Please submit only one PDF file for both students. On the cover page, put lab section, name and ICID information in this format:

> Group Submission for [lab section]
>
> Submitted by: [submitter's name]
> UCID: [submitter's UCID]
>
> Partner: [partner's name]
> UCID: [partner's UCID]

Marking scheme

A	6 marks
B	3 marks
C	5 marks
TOTAL	14 marks

Getting files for this lab

The files you need for this week's exercises are in a folder called `encm369w23lab06`, which has been uploaded to D2L in `.zip` format. Before starting work, you must download the `.zip` file, and extract its contents in a suitable place in the file system.

Exercise A: Datapath and control signals

Read This First

The point of this exercise is to make a detailed examination of the behaviour of the single-cycle processor circuit of Figure 7.12 in the course textbook.

What to Do, Part I

Suppose the instruction

```
lw t6, 20(sp)
```

is located at address `0x0040_0130` in instruction memory. Suppose that just before the instruction is executed, the following GPRs have the given values:

```
sp = 0x7fff_fea0
t6 = 0x0001_2345
```

Finally suppose that data memory contains the following words:

```
ADDRESS        DATA
0x7fff_fea0    0x0001_1111
0x7fff_fea4    0x0002_2222
0x7fff_fea8    0x0003_3333
0x7fff_feac 12 0x0004_4444
0x7fff_feb0 16 0x0005_5555
0x7fff_feb4 20 0x0006_6666
0x7fff_feb8    0x0007_7777
```

(Note: You have been given more information than you actually need to solve the problem.)

During the clock cycle in which the instruction is executed, most of the signals in the circuit will change values. This problem asks you to find the values of some of these signals *just before the end of the clock cycle*, when all of the signals will have stabilized to their final values for the clock cycle.

Determine and write out the values of these signals:

- The values of the control signals RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, and PCSrc. Use base two for the 2-bit ImmSrc signal and the 3-bit ALUControl signal.
- The values of the 5-bit A1, A2, and A3 inputs to the Register File, as base two numbers.

- The values of these 32-bit signals: SrcA, SrcB, ALUResult, Result, and PC-Next, in hexadecimal format.
- The value of the 32-bit WD3 input to the Register File, in hexadecimal format.

(If you don't have a calculator that can display numbers in hexadecimal, you may want to use the C programs in `encm369w23lab07/exA` to help generate some of bit patterns you will need.)

What to Do, Part II

Repeat Part I, but this time assume that the instruction address is `0x0040_0134` and the instruction is

```
sub s1, s1, t5
```

Use these starting values for the source registers:

```
s1 = 0x0000_0064
t5 = 0xffff_fff0
```

For Part II, you can leave out ImmSrc, because its value is unspecified for an R-type instruction.

What to Include in Your PDF Submission

Include your answers to Parts I and II.

Exercise B: Immediate-mode instructions in the single-cycle machine

Read This First

A good way to see whether you understand the design of the circuit of Figure 7.12 in your textbook is to try to enhance it to support more instructions. When doing this kind of exercise it is important to remember this:

> *It's not enough to modify the datapath and control so that the new instructions work—the eight instructions already implemented have to continue to work properly!*

Let's consider adding support for the following useful instructions: `addi`, `slti`, `ori`, and `andi`. This is presented in Section 7.3.4 of the course textbook, which also shows how to add support for `jal`.

Figure 1 shows the instruction formats for these new instructions and the 5 R-type instructions supported by the Figure 7.12 computer. The new instructions all have 0010011 for an opcode, and they all encode their immediate operand the same way that an `lw` offset is encoded. So when the Main Decoder part of the Control Unit sees an opcode of 0010011 it should make ImmSrc = 00_2 to get a correct value of ImmExt out of the Extend unit.

Note that the funct3 field for `slti` matches the same field in `slt`. Note too that the same thing is true regarding `or` and `ori` and regarding `and` and `andi`. That's very nice, because if the Main Decoder makes ~~ImmSrc~~ ALUOp = 10_2 when it sees an opcode of 0010011, then the ALU Decoder part of the Control Unit designed for the original 8-instruction subset will do the right things for `slti`, `ori` and `andi`.

If we decide that the Main Decoder should make ~~ImmSrc~~ ALUOp = 10_2 when it sees an opcode of 0010011, that creates a problem in choosing ALUControl for

Figure 1: Formats for instructions relevant to Exercise B.

	31 25	24 20	19 15	14 12	11 7	6 0
ADD	0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
SUB	0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
SLT	0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
OR	0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
AND	0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
	funct7			funct3		opcode

	31 20	19 15	14 12	11 7	6 0
ADDI	$imm_{11:0}$	rs1	0 0 0	rd	0 0 1 0 0 1 1
SLTI	$imm_{11:0}$	rs1	0 1 0	rd	0 0 1 0 0 1 1
ORI	$imm_{11:0}$	rs1	1 1 0	rd	0 0 1 0 0 1 1
ANDI	$imm_{11:0}$	rs1	1 1 1	rd	0 0 1 0 0 1 1
			funct3		opcode

Figure 2: ALU Decoder specification to handle the 8-instruction subset and also `addi`, `slti`, `ori` and `andi`.

	Instr bits				
ALUOp	30	14:12	5	ALUControl	instruction(s) supported
00	x	xxx	x	000 (addition)	`lw`, `sw`
01	x	xxx	x	001 (subtraction)	`beq`
10	0	000	0	000 (addition)	`addi`
10	0	000	1	000 (addition)	`add`
10	1	000	0	000 (addition)	`addi`
10	1	000	1	001 (subtraction)	`sub`
10	x	010	x	101 (set less than)	`slt`, `slti`
10	x	110	x	011 (bitwise OR)	`or`, `ori`
10	x	111	x	010 (bitwise AND)	`and`, `andi`

`add`, `sub`, and `addi`. $\mathrm{Instr}_{14:12}$ is 000 for all three instructions. Instr_{30} can't by itself be used to choose between addition and subtraction because in `addi` Instr_{30} may be 0 or 1 depending on the value of the immediate operand. The solution is to involve Instr_5 in the decision—that bit is 1 for `add` and `sub` but 0 for `addi`. The resulting ALU Decoder specification is shown in Figure 2. Note that Figure 2 conveys the same information as textbook Table 7.3 in a way that might be easier to understand.

What to Do

Suppose that the control unit of the textbook Figure 7.12 computer has been enhanced as described above in "Read This First." Suppose also that the instruction

```
andi t1, t1, -256
```

is located at address `0x0040_015c` in instruction memory. Suppose that just before the instruction is executed, `t1` has a value of `0x89ab_cdef`.

During the clock cycle in which the instruction is executed, most of the signals in the circuit will change values. This problem asks you to find the values of some of these signals *just before the end of the clock cycle*, when all of the signals will have stabilized to their final values for the clock cycle.

Determine and write out the values of these signals:

- The values of the control signals RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, and PCSrc. Use base two for the 2-bit ImmSrc signal and the 3-bit ALUControl signal.
- The values of the 5-bit A1, A2, and A3 inputs to the Register File, as base two numbers.
- The values of these 32-bit signals: SrcA, SrcB, ALUResult, Result, and PC-Next, in hexadecimal format.
- The value of the 32-bit WD3 input to the Register File, in hexadecimal format.

What to Include in Your PDF Submission

Include your answers.

Exercise C: Support for `jalr` in the single-cycle machine

Read This First

As seen in Exercise B, textbook Section 7.3.4 shows how to modify the original single-cycle computer design to include support for `addi`, `slti`, `ori`, and `andi`. The same section shows how to enhance the design to support the `jal` instruction, which is essential for unconditional jumps and for function calls.

The computer of textbook Figures 7.15 and 7.16 and Tables 7.5 and 7.6 supports unconditional jumps and function calls but not function returns. It seems unsatifying to have a computer that allows function calls but not function returns, so let's look at enhancing the design to support `jalr`, the instruction used to support the `jr` pseudoinstruction and some other useful pseudoinstructions. This is the machine-code format for `jalr`:

31 20	19 15	14 12	11 7	6 0
$\text{offset}_{11:0}$	rs1	0 0 0	rd	1 1 0 0 1 1 1

`jalr` writes the sum of the GPR found with rs1 and the 32-bit sign extension of $\text{offset}_{11:0}$ into the PC. `jalr` also writes 4 plus the current PC value into the GPR found with rd, which is useful for function calls using pointers to functions in C and C++, and virtual function calls in C++. (Don't worry if you don't know what pointers to functions or C++ virtual functions are—those are good things to know about for software engineers but not things you need to know to do this exercise.)

The most common use of `jalr` in most RISC-V assembly language code is to support this pseudoinstruction:

```
jr      ra
```

For that pseudoinstruction, $\text{offset}_{11:0}$ will be 0000 0000 0000, rs1 will be 00001 so that the jump target address will come from the `ra` register, and rd will be 00000 to prevent an update to any of the GPRs.

What to Do

Figure 3 shows a very slightly modified version of the computer of textbook Figure 7.15. The modifications are an additional control signal called TargetSrc and a multiplexer controlled by that new control signal.

You can assume that the ALU Decoder design explained in Exercise B will work support `jalr` without modification. Obviously, though, changes to the Main Decoder are needed.

Make a copy of the table in Figure 4 and fill in all the blank cells with appropriate 1- or 2-bit values. Use x for don't care whenever possible. After your table is complete, write nine one-or-two-sentence explanations to support your choices of all nine Main Decoder outputs.

What to Include in Your PDF Submission

Include your completed table and your explanations for control signal choices.

Exercise D: A single-cycle machine for a different instruction set

Attention!

This exercise will not be marked. *However, please do not think of this as an optional exercise. Doing the work may be very helpful as you prepare for Midterm #2.*

Read This First

This exercise is adapted from a problem on the Winter 2006 final exam.

The Exam16 ISA (instruction set architecture) describes a system in which addresses, instructions, and data words are all 16 bits wide. It has sixteen 16-bit general purpose registers, and a 16-bit PC. The instructions of Exam16 are given in the table in Figure 5. Note that unlike with RISC-V, there are no offsets built into Exam16 load and store instructions.

Figure 6 is a nearly-complete datapath for a computer that implements the Exam16 ISA. It is very much in the style of the ***single-cycle*** RISC-V subset implementation studied in ENCM 369. Note that there are two 16-bit adders and a 16-bit ALU. The ALUOp signal works as follows: 00 asks for addition, 01 for subtraction, and 10 for set-on-less-than. The circuit labeled "All Bits 0?" is a big NOR gate—the 1-bit output is 1 if all 16 input bits are 0, and is 0 otherwise.

Data Memory in Figure 6 works very slightly differently from Data Memory in examples in the 2023 textbook and 2023 lectures. In Figure 6, MemRead and MemWrite should both be zero for instructions that do not access memory, to avoid wasting energy. Values of MemRead and MemWrite are obvious for loads and stores, I hope!

Branch target address computations in Exam16 do not work the same way as they do in RISC-V. In Exam16, the branch target address is equal to

$$\text{PC} + 2 + 2 \times (\text{16-bit sign extension of Instruction[7–0]})$$

If you recall that a left shift by 1 bit does multiplication by 2, it should be easy for you to find the part of the circuit that produces a branch target address.

What to Do

In some of the following parts, it may be helpful to draw a simple diagram or two to go along with the text you write to answer the question being asked.

Figure 3: Textbook Figure 7.15 computer enhanced to support `jalr`.

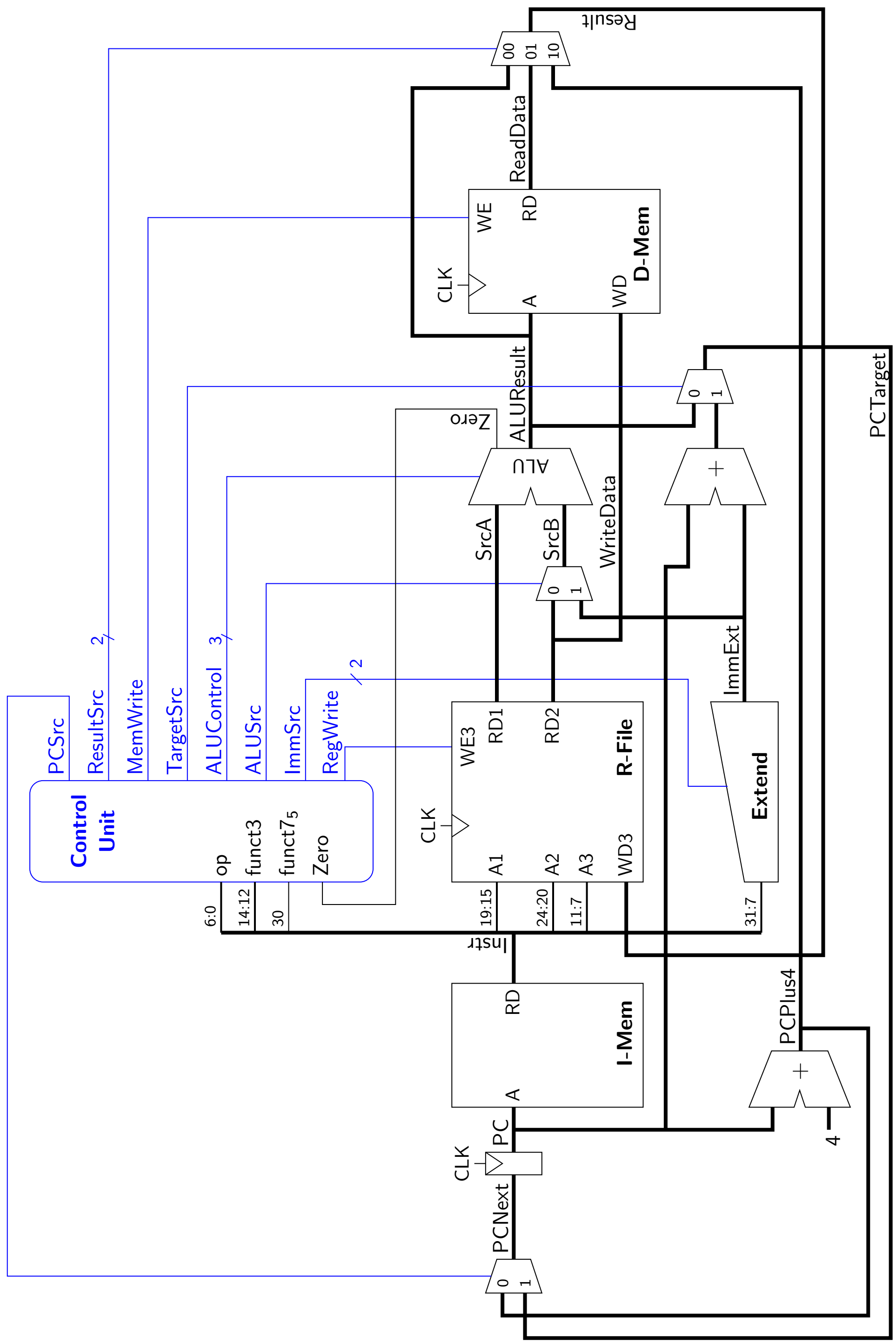


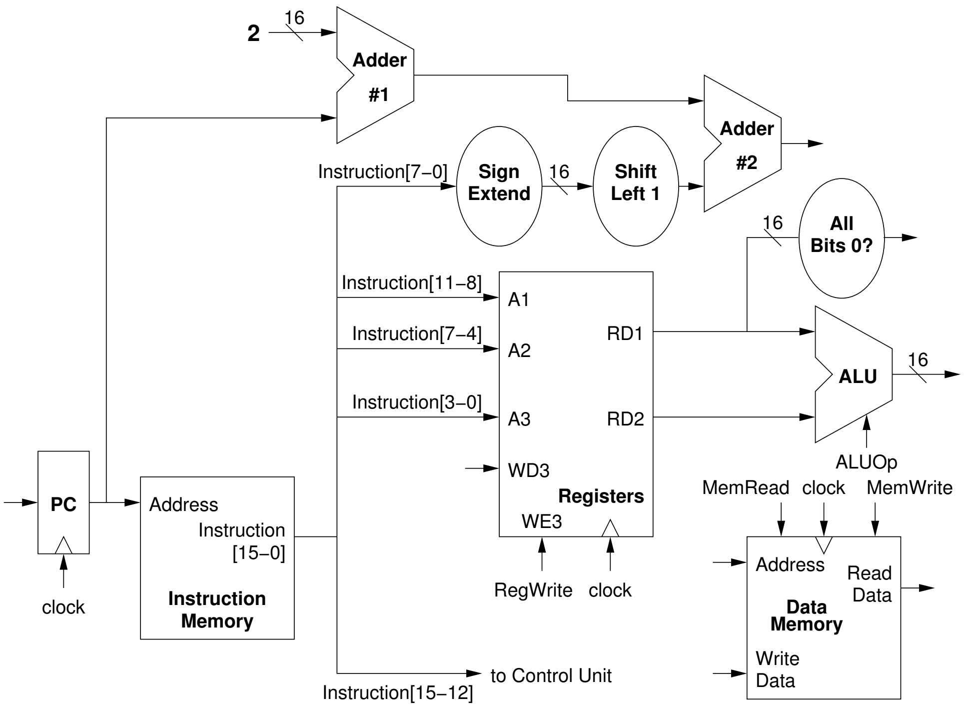
Figure 4: Incomplete Main Decoder specification for single-cycle computer with support for `jalr`. Compared to textbook Table 7.6, the table here has one more row and one more column.

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	TargetSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1		0	01	0	00	0
`sw`	0100011	0	01	1		1	xx	0	00	0
R-type	0110011	1	xx	0		0	00	0	10	0
`beq`	1100011	0	10	0		0	xx	1	01	0
I-type ALU	0010011	1	00	1		0	00	0	10	0
`jal`	1101111	1	11	x		0	10	x	xx	1
`jalr`	1100111									

Figure 5: Exam16 instruction set.

Mnemonic	Format	Description
`add`	`0000_ssss_tttt_dddd`	Add source registers selected by bits `ssss` and `tttt`, put result in register selected by bits `dddd`.
`sub`	`0001_ssss_tttt_dddd`	Same as `add`, except ALU operation is subtraction.
`slt`	`0010_ssss_tttt_dddd`	Same as `add`, except ALU operation is set-on-less-than.
`brz`	`0011_ssss_oooo_oooo`	Branch if register is zero: If register selected by bits `ssss` contains zero, branch forward or backward by number of instructions in 8-bit 2's-complement offset `oooo_oooo`.
`lw`	`0100_0000_aaaa_dddd`	Using register selected by bits `aaaa` as an address, load word from data memory into register selected by bits `dddd`.
`sw`	`0101_ssss_aaaa_0000`	Using register selected by bits `aaaa` as an address, store word from register selected by bits `ssss` into data memory.

Figure 6: Nearly complete datapath for Exam16 single-cycle implementation.



Part a. The Address and Write Data inputs to the Data Memory are not connected to anything in Figure 6. What signals should be sent to these inputs? Explain why.

Part b. The WD3 input to the Register File is not connected to anything. *How should this signal be driven?* Here is a hint: Introduce a new control signal, give it a name, and use it to control a multiplexer. *Briefly give reasons to support your design.*

Part c. The input to the PC register is not connected to anything. *How should this signal be driven?* A new control signal, a new multiplexer, and perhaps some other new, simple logic element will be needed. *Briefly give reasons to support your design.*

Part d. Make a copy of the table from Figure 7, or print the page that it is on. Then fill in all the blank cells with the correct values of control signals for the given instructions.

The last two columns are reserved for the new control signals you introduced in **parts b and c**—please write in the names of these signals.

Use "X" in table cells to indicate that a particular control signal is a "don't care" for a particular instruction.

Part e. Suppose you want to extend the Exam16 ISA to include an `addc` ("add constant") instruction with the following format:

```
1000_ssss_cccc_dddd
```

`ssss` encodes the source register, `dddd` encodes the destination register, and `cccc` encodes a constant in the range from 0 to 15. Describe all the datapath changes (not control changes) that would be needed to add support for `addc` while continuing to support the original six Exam16 instructions.

Figure 7: Table of control signals for **part d** of Exercise D.

Instruction	MemRead	MemWrite	RegWrite	ALUOp		
add						
sub						
slt						
brz						
lw						
sw						

What to Include in Your PDF Submission

Nothing. You can check your answers against solutions that will be posted on or before Tuesday, March 7.