18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

HW8 Solutions [4 problems, 64 points]

Covers lectures L16 – L18 *Due: 20 November 2023*

Non-Drill Problems [64 points]

There are no drill problems this week.

1. [18 points, Lecture 17] Complete the RISC240 Assembler and Simulator Tutorial, and read the RISC240 Reference Manual. Both are available on Canvas under "Supplemental Handouts → RISC240".

   In Step 3 of the tutorial, when you are filling out the table, you are asked for **MAR** and **MDR** values. Those are registers that you will learn about soon. Nevertheless, copy the values from the simulator to the table.

   From the tutorial, you will need to turn in the following:

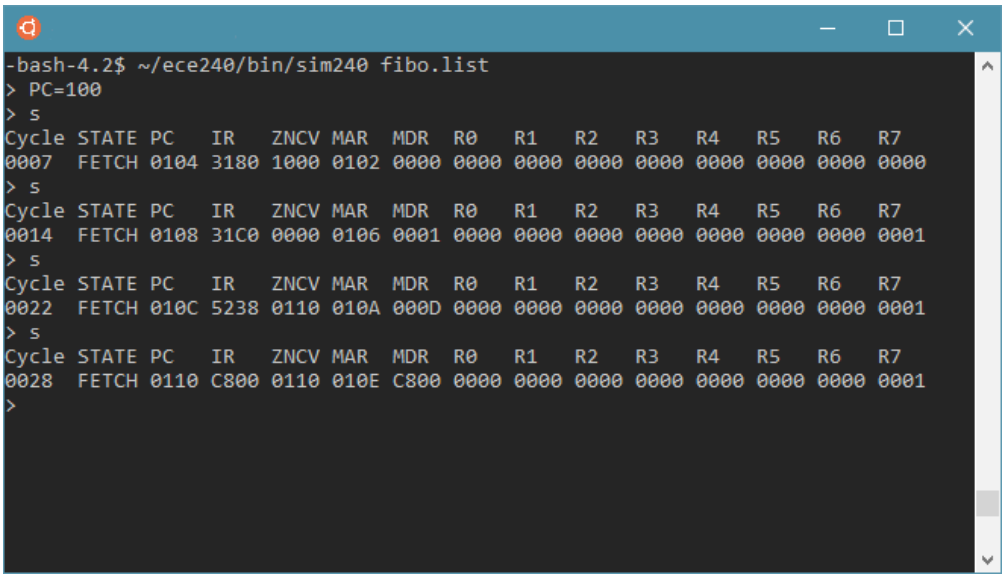
   - [6 points] Your hand-assembled binary memory listing from Step 1. It should be obvious from your submission that you have done the work to assemble this by hand.

```
//          .ORG    $100
// start    LI      R6, $0      ; R6 <- $0 (immediate)
// LI: long format, immediate addressing mode
0011 0001 1000 0000
0000 0000 0000 0000 // $0

//          LI      R7, $1      ; R7 <- $1 (immediate)
// LI: long format, immediate addressing mode
0011 0001 1100 0000
0000 0000 0000 0001 // $1

// loop     SLTI    R0, R7, $D ; R7 - D (result discarded)
// SLTI: long format, immediate addressing mode
0101 0010 0011 1000
0000 0000 0000 1101 // $D

//          BRZ     done        ; branch if R7 == D
// BRZ: long format, immediate addressing mode
1100 1000 0000 0000
0000 0001 0001 1000 // $0118
```

```
//          ADD     R6, R6, R7 ; R6 <- R6 + R7
// ADD: short format, register addressing mode
0000 0001 1011 0111

//          ADD     R7, R7, R6 ; R7 <- R7 + R6
// ADD: short format, register addressing mode
0000 0001 1111 1110

//          BRA     loop       ; branch always ("go to")
// BRA: long format, immediate addressing mode
1111 1000 0000 0000
0000 0001 0000 1000 // $0108

// done     STOP               ; all done
// STOP: short format, register addressing mode
1111 1110 0000 0000
```

- [6 points] The table from Step 4, along with a screenshot of the simulator after execution of the **`BRZ done`** instruction (the first time it is encountered).

	PC	IR	ZNCV	MAR	MDR	R6	R7
`LI R6, $0`	0104	3180	1000	0102	0000	0000	0000
`LI R7, $1`	0108	31C0	0000	0106	0001	0000	0001
`SLTI R0, R7, $D`	010C	5238	0110	010A	000D	0000	0001
`BRZ done`	0110	C800	0110	010E	C800	0000	0001
`ADD R6, R6, R7`	0112	01B7	0000	0110	01B7	0001	0001
`ADD R7, R7, R6`	0114	01FE	0000	0112	01FE	0001	0002
`BRA loop`	0108	F800	0000	0116	0108	0001	0002



- [6 points] Three assembly files for programs that you generated in Step 5. Make sure they include a comment at the beginning of the file, describing what they do.

  Your answers may vary.

2. [8 points, Lecture 18] Take a look at the datapath diagram from the end of Lecture 18's slides. Answer the following questions about that diagram.

   (a) The decoder in the upper left, as you know, converts the binary value **`dest[2:0]`** into a one-hot signal. Where does the **`dest[2:0]`** signal come from? Hint: It is shown in green, which has a particular meaning on this diagram.

   It is a *control point* and comes from the FSM, as do all green-boxed signals on this diagram.

   (b) What is the *purpose* of the decoder in this datapath?

   The destination register for an operation is specified by the control value. The output of the decoder is an enable signal to one of the various registers in the datapath: those in the register file, the **`PC`**, the **`MAR`**, etc.

   (c) If I wanted to add two values together, each of which is in the register file, and put the result in the Memory Address Register, what values would I have to provide for each of the control points?

   `aluSrcA` and **`aluSrcB`** should be **`2'b00`** so that the values in the register file are chosen to be the inputs to the ALU.

   `aluOp` should be **`2'b0000`**, which I found in the table on slide 25 ("Arithmetic Logic Unit (ALU)"). THis specifies that the ALU should do an addition operation on the incoming values.

   I should probably assert $\overline{loadCC}$ though the problem didn't specifically say that I would need to save the condition codes.

   `dest` should be **`3'b011`**, as that will enable the MAR to accept the result value coming from the ALU.

   (d) What does it indicate that the **`condCodes`** are colored purple?

   The purple boxes show the status points, signals that go back to the FSM. So far, we only know about the condition codes, but there will be more status points as we explore the RISC240 datapath in more detail.

3. [20 points, Lecture 17] Consider this RISC240 assembly language program:

Ouch! Whoever wrote this would be a horrible lab partner – no comments and single character, nonsense labels.

Answer the following questions about this code:

(a) [3 points] What is the general task that this snippet is performing? *(Hint: this is a fairly common mathematical task.)*

Find the maximum value from an array of values.

(b) [2 points] For each of the eight registers, list what the registers are being used for in this program. If a register is not used, write "not used".

r0 is the zero register

r1 is the maximum value found so far (and where the answer is at the end of the program).

r2 is not used.

r3 points to each element of the array.

r4 points to the end of the array.

r5 is not used.

r6 holds the value read from the array.

r7 is not really used. Comparison values are written here by the SLT instruction, but those values are never read.

(c) [3 points] What is the address of the first **SLT** instruction?

The code starts at address **$1000**. The **LI** and **LW** instructions are long instruction format (thus two words of memory each for a total of six) and the **MV** instruction is short format (thus one word of memory). But, remember that each word of memory is two bytes (and that all addresses are counted in bytes). Therefore the **SLT** instruction is located at **$100E**.

(d) [4 points] How many bytes of memory does the code for the program take up? Don't include the **.DW** pseudo-operations. Show your work.

The long instruction format instructions are: **LI** (qty 2), **LW**, **ADDI**, **BRNZ**, **BRZ**, **BRA**. That's seven long instructions.

The rest are short format: **MV** (qty 2), **SLT** (qty 2), and **STOP**. That's five short format instructions.

Each long format instruction takes 4 bytes of memory and each short format takes 2-bytes. $7*4+5*2=38$ bytes of memory.

(e) [1 point] When the program finishes executing, where is the answer stored?

In r1.

(f) [4 points] When the program is executed, using the data values provided, what value is the answer? There is another value that one might suspect would be the answer (but, isn't). Do you see it? Explain exactly why the program doesn't choose this value as the answer.

The correct answer is **$7FFF**. One might think that the maximum value in the array is **$FFFF**, however, the **SLT** comparison is done as a signed-comparison.

(g) [3 points] The program, as written, actually has a bug – one single character is wrong. Find and fix the bug. Explain why this is a bug.

The **ADDI** instruction should be adding two, not one, to r3 in order to get the next value from the array. Since the memory in RISC240 is word-aligned, each element of the array takes two bytes, and thus shows up in successive, even-numbered memory locations.

4. [18 points, Lecture 17] Write a RISC240 assembly program to generate a magic square of (somewhat) arbitrary size.

See **hw8prob4.asm** on Canvas.