CSCI 463 Assignment 3 – Microcode Machine

20 Points – Due Tuesday, November 12, 2019 at 23:59

Abstract

In this assignment, you will use the machine simulator demonstrated in lecture to complete the design and implementation of a microcode program that will implement the instruction set described herein.

1 Input

The simulator reads two files:

1. Microcode expressed as discussed below.
2. Up to 256 bytes expressed in hex that will be loaded into the system memory before the microcode starts executing.

The microcode for this machine can be developed by starting with the instruction *fetch*, *decode* and *execute* example code discussed in class, given on blackboard and reproduced in Figure 1.

The headings in Figure 1 match the columns and number of bits in each field of the microcode input file that you must create. Note that, in this microcode format, all fields have their values expressed in hex.

2 Instruction Definitions

The descriptions in this section specify the operations that have to take place *in addition to* those in the instruction fetch and decode logic. Specifically, these descriptions assume that the PC register has been incremented prior to the pseudocode that is given below for each instruction.

2.1 NOP

Do nothing.

This instruction does nothing except to advance the PC register to the next instruction. Therefore there is nothing to do after the instruction fetch and decode logic has advanced the PC register.

2.2 LDI Ra,imm

Load the contents of a memory byte that appears in the instruction stream immediately after the instruction opcode into register Ra.

Pseudocode:

MAR ← PC
MBR_IN ← mem
Ra ← MBR_IN
PC ← PC+1

2.3 ST Ra,imm

Store the contents of register Ra into memory at the absolute address that appears in the instruction stream immediately after the instruction opcode.

Pseudocode:

MAR ← PC
MBR_IN ← mem
MAR ← MBR_IN
MBR_OUT ← Ra
mem(MAR) ← MBR_OUT
PC ← PC+1

2.4 ADD Ra,Rb

Add the contents of Rb to Ra.

Pseudocode:

Ra ← Ra + Rb
FLAGS ← ALU_status

2.5 SUB Ra,Rb

Add the contents of Rb from Ra.

Pseudocode:

Ra ← Ra - Rb
FLAGS ← ALU_status

2.6 XOR Ra,Rb

Exclusive–or the contents of Ra with Rb.

Pseudocode:

Ra ← Ra $\oplus$ Rb
FLAGS ← ALU_status

2.7 AND Ra,Rb

Add the contents of Ra with Rb.

Pseudocode:

Ra ← Ra $\wedge$ Rb
FLAGS ← ALU_status

2.8 OR Ra,Rb

OR the contents of Ra with Rb.

Pseudocode:

Ra ← Ra $\vee$ Rb
FLAGS ← ALU_status

2.9 MOV Ra,Rb

Copy the contents of Rb into Ra.

Pseudocode:

Ra ← Rb

Note that the MOV instruction does NOT change the ALU flag register(s).

2.10 LD Ra,mem(imm)

Load the contents of a memory byte from the address that appears in the instruction stream immediately after the opcode into register Ra.

Pseudocode:

MAR ← PC
MBR_IN ← mem
MAR ← MBR_IN
MBR_IN ← mem
Ra ← MBR_IN
PC ← PC+1

2.11 B imm

Branch to the *absolute* address in the byte that appears in the instruction stream immediately after the instruction opcode. (Note that this is similar to the LDI Ra,imm instruction except that it will store the fetched address byte into the into the PC register (rather than Ra.)

Pseudocode:

MAR ← PC
MBR_IN ← mem
PC ← MBR_IN

2.12 BR imm

Branch to the *relative* address in the byte that appears in the instruction stream immediately after the instruction opcode.

A *relative* branch is implemented by adding the `imm` operand (as am 8-bit signed value) to the PC register.

Pseudocode:

MAR ← PC
MBR_IN ← mem
PC ← PC + 1
PC ← PC + MBR_IN

Note that the PC register should be pointing to the em next instruction before the addition is performed! Therefore, the following is an endless loop:

```
BR 0xfe
```

2.13 BZ PC+imm

If the Z flag is set then branch to the relative address that appears in the instruction stream immediately after the opcode.

Pseudocode:

if (Z)
    MAR ← PC
    PC ← PC + 1
    MBR_IN ← mem
    PC ← PC + MBR_IN
else
    PC ← PC + 1

Note that this instruction is implemented using a branch table (similar to the instruction decode) but with the `uc_addr_mux` set to 1 (as opposed to 2.) When `uc_addr_mux = 1`, the next microcode instruction address is determined by replacing the 4 LSBs if the `uc_next_addr` field with the current value of the FLAGs register. To implement a conditional branch set up a branch table and provide a `uc_next_addr` value for those table entries where Z is true to that of the unconditional relative branch microcode and for the other table entries provide a `uc_next_addr` value that will execute microcode that will add 1 to the PC register and proceed to the instruction fetch code.

2.14 BNZ PC+imm

If the Z flag is *not* set then branch to the relative address that appears in the instruction stream immediately after the opcode.

Pseudocode:

if (!Z)
    MAR ← PC
    PC ← PC + 1
    MBR_IN ← mem
    PC ← PC + MBR_IN
else
    PC ← PC + 1

This can be implemented identically to the BZ instruction but by using a different set of branch table entries.

3 Your Microcode File

As seen below, your microcode can have comments. Anything after the pound-sign '#' is ignored as are blank lines and lines that only have comments in them. Be as verbose as is necessary so that you can remember your intentions as you debug your code.

Example Microcode Input File Format With Comments

```
1 ##############################
2 # LDI Ra,imm
3 # fetch the byte in memory that the PC is pointing to now
4 1100 0  0 0 0  0 1 0 0 0  0 0 7  0 4 0 7 1101   # MAR <- PC
5 1101 0  0 0 0  0 0 0 0 0  0 0 7  0 4 0 7 1102   #
6 1102 0  0 0 0  0 0 0 0 1  0 0 7  0 7 0 7 1103   # MBR_IN <- d_in
7 1103 0  0 0 0  0 0 0 0 0  0 0 7  0 7 0 7 1104   #
8 1104 0  0 0 0  0 0 0 0 0  1 1 7  0 7 0 4 1105   # Ra <- MBR_IN
9 1105 0  0 0 0  0 0 0 0 0  0 1 7  0 7 0 4 1106   #
```

ROM address	uc_addr_mux[2]	uc_alu_func[2]	uc_alu_comp_b	uc_alu_ci	uc_alu_flags_clk	uc_mar_we	uc_mem_we	uc_mbr_out_we	uc_mbr_in_we	uc_reg_we_clk	uc_reg_addr_ir	uc_reg_addr[3]	uc_alu_reg_a_ir	uc_alu_reg_a[3]	uc_alu_reg_b_ir	uc_alu_reg_b[3]	uc_next_addr[16]	note
0000	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	0001	do nothing
opcode fetch																		
0001	0	0	0	0	0	1	0	0	0	0	0	5	0	4	0	7	0002	MAR ← PC
0002	0	0	0	0	0	0	0	0	1	0	0	5	0	4	0	7	0003	MBR_IN ← mem
0003	0	0	0	0	0	0	0	0	0	0	0	5	0	4	0	7	0004	
0004	0	0	0	0	0	0	0	0	0	1	0	5	0	7	0	4	0005	IR ← MBR_IN
0005	0	0	0	0	0	0	0	0	0	0	0	5	0	7	0	4	0006	falling uc_reg_we_clk
increment PC register																		
0006	0	0	0	1	0	0	0	0	0	1	0	4	0	4	0	7	0007	PC ← PC+1
0007	0	0	0	1	0	0	0	0	0	0	0	4	0	4	0	7	0010	
opcode decode																		
0010	2	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	00f0	branch using opcode in IR
opcode branch table																		
00f0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1000	opcode 0 NOP
00f1	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1100	opcode 1 LDI Ra,imm
00f2	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1200	opcode 2 ST Ra,imm
00f3	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1300	opcode 3 ADD Ra,Rb
00f4	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1400	opcode 4 SUB Ra,Rb
00f5	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1500	opcode 5 XOR Ra,Rb
00f6	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1600	opcode 6 AND Ra,Rb
00f7	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1700	opcode 7 OR Ra,Rb
00f8	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1800	opcode 8 MOV Ra,Rb
00f9	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1900	opcode 9 LD Ra,mem(imm)
00fa	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1a00	opcode a B imm
00fb	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1b00	opcode b BR PC+imm
00fc	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1c00	opcode c BZ PC+imm
00fd	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	1d00	opcode d BNZ PC+imm
00fe	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	ffff	opcode e HALT
00ff	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	ffff	opcode f HALT
NOP																		
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	0001	go to insn fetch
LDI Ra,imm # Rd ← mem(PC), PC ← PC+1																		
1100	0	0	0	0	0	1	0	0	0	0	0	7	0	4	0	7	1101	MAR ← PC
1101	0	0	0	0	0	0	0	0	0	0	0	7	0	4	0	7	1102	
1102	0	0	0	0	0	0	0	0	1	0	0	7	0	7	0	7	1103	MBR_IN ← d_in
1103	0	0	0	0	0	0	0	0	0	0	0	7	0	7	0	7	1104	
1104	0	0	0	0	0	0	0	0	0	1	1	7	0	7	0	4	1105	Ra ← MBR_IN
1105	0	0	0	0	0	0	0	0	0	0	1	7	0	7	0	4	1106	
1106	0	0	0	1	0	0	0	0	0	1	0	4	0	4	0	7	1107	PC ← PC+1
1107	0	0	0	1	0	0	0	0	0	0	0	4	0	4	0	7	1108	
1108	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	7	0001	go to insn fetch

Figure 1: Microcode Fetch, Decode And Execute Example Code

```
1106 0  0 0 1  0 0 0 0 0  1 0 4  0 4 0 7 1107   # PC <- PC+1
1107 0  0 0 1  0 0 0 0 0  0 0 4  0 4 0 7 1108   #
1108 0  0 0 0  0 0 0 0 0  0 0 0  0 7 0 7 0001   # go to insn fetch
```

A memory image dump file is formatted as a set of whitespace-separated hex byte values as many or as few per line as you prefer as shown below. The byte values will be placed into memory starting at address zero in the order appearing in your input file.

Anything after a pound-sign '#' is ignored as are blank lines and lines that only have comments in them.

An Example Memory Dump Input File

```
00      # nop
10 55   # ldi R0,0x55
ff ff   # halt
```

4 Output

The simulator will reset the CPU, load the input memory dump, print the initial memory contents, the register state every time it encounters an microcode instruction with `uc_addr_mux` set to 2 (effectively printing the instruction stream each time an opcode is fetched) and then another memory dump when it is halted.

The interpreter will halt when it encounters an undefined instruction. For debugging purposes, you can take advantage of this by branching to an address that has no microcode instruction in it.

Your program must properly execute the instructions from the given memory image.

The test memory image will expect to terminate by executing an opcode whose value is `0xff`. (Hence the definition of the `HALT` instruction above.)

5 File You Must Write

You will create and submit one file for this assignment.

Create a directory named `prog3` and place your source files for this assignment within. Implement your solution for this assignment in the file named `prog3.uc`.

When we grade your assignment, we will compile and run it on `hopper.cs.niu.edu` using this command with a special test memory dump file designed to rigorously test your CPU:

`~winans/csci463/a3/ucsim prog3.uc test.mem`

6 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only file in your `prog3` directory is the `prog3.uc` file discussed above. Then, in the parent of your `prog3` directory, use the `mailprog.463` command to send the contents of the files in your `prog3` project directory to your TA like this:

```
mailprog.463 prog3
```

If `mailprog.463` detects and problems, it will inform you that you have not followed the instructions given above and provide some hints how to proceed. If you followed these instructions you will see the following:

```
winans@hopper:~/$ mailprog.463 prog3

*********************************************************************
* WARNING : Do NOT use this program to mail notes to your Instructor *
*           Doing so may result in the loss of your program !!       *
*********************************************************************

Enter program number for your assignment : 3
shar: Saving /tmp/mailprog.11111 (text)
winans@hopper:~/$
```

7 Grading

The grade you receive on this programming assignment will scored according to the syllabus and its ability to execute on the CSCI Department's computer.

It is your responsibility to test your program thoroughly. While grading, a variety of instructions will be executed with different operand values.

8 The `ucsim` Simulator

For this assignment, the `ucsim` command may be executed with different levels of verbose output:

- `ucsim prog3.uc test.mem`
  Run with minimal output. This is how your program will be graded and is shown in the example test run below. This will dump the contents of the CPU regs and a description of the instruction that *should* be executed each time the simulator encounters a microcode instruction with `uc_addr_mux = 2`.
- `ucsim prog3.uc -v1 test.mem`
  Run with a dump of the CPU signal state after each microcode instruction is executed. Use this mode to debug individual microcode instructions. Remember that you can easily halt the simulator at any point in your microcode logic by specifying the value `ffff` as the instruction's `uc_next_addr` field.
- `ucsim prog3.uc -v2 test.mem`
  This will add a memory dump after each microcode instruction
- `ucsim prog3.uc -v3 test.mem`
  This is an abbreviated version of `-v1` that you might find useful.

9 Hints

Start by creating a memory image with NOP and HALT instructions to verify that you can execute the simulator and recognize the output.

Do the easy ones first! The ADD, SUB and other RR instructions that only use the ALU are the easiest to complete. Implement and test the ADD instruction first. Proceed implementing the rest of the RR instructions by copying the ADD microcode as a template. The only things you should have to change for each RR instruction will be the microcode address and `uc_alu_XXX` fields.

In order to set FLAGs register as part of an instruction, such as ADD for example, the ALU outputs `alu_n`, `alu_z`, `alu_uo` and `alu_so` have to be stored into the FLAG register at the same time that the sum is being calculated (and, perhaps, stored into the `Ra` register). In other words, you should expect to have to assert

both of the `uc_reg_we_clk` and `uc_alu_flags_clk` signals at the same time if you want to store both the sum and the flags.

In order to make copying microcode segments easier, do not do more than one thing in each microcode instruction. For example, note the contrast in the way that the given LDI code above at addresses 1100-1103 is implemented using pairs of microcode instructions versus the same logic in the fetch code at addresses 0001-0002. The former is easier to cleanly extract just one of the two operations without cleaning up the interference of the other operation afterword.

Don't forget to save the ALU state in the flags register during operations such as ADD, SUB... so that they can be used later on in conditional branch instructions. (Note that MOV does *not* set the ALU flags. If you copy and paste your code for opcodes 3-7 then make sure that you don't accidentally set the ALU flags for opcode 8 too!)

When executing a conditional branch instruction, make sure that you remember to advance the PC register to skip over the immediate operand (if you use it or not). Look closely at the operand values in the example memory dump and corresponding output below.

10 An Example Test Run

Here is an example of a test run with matching output.

Example Memory Dump

```
00      # NOP
a0 10   # B 0x10

# some junk data to jump over
ff ff ff ff ff ff ff ff ff ff ff ff

10 11   # LDI R0,0x11
10 0    # LDI R0,0
10 60   # LDI R0,0x60
30      # ADD R0,R0
20 f0   # ST R0,0xf0
40      # SUB R0,R0

14 55   # LDI R1,0x55   R1=0x55
18 50   # LDI R2,0x50   R2=0x50
56      # XOR R1,R2     R1=0x05, R2=0x50

76      # OR R1,R2      R1=0x55, R2=0x50
66      # AND R1,R2     R1=0x50, R2=0x50

8d      # MOV R3,R1     R1=0x50, R2=0x50, R3=0x50

90 f0   # LD R0,(0xf0)  R0=0xc0

b0 02   # BR 0x02
ff ff   # (this is to be branched over to get to next insn)
00      # NOP


# Count from 4 to 0 using R1 as the counter
10 00   # LDI R0,0x00
14 04   # LDI R1,0x04
18 01   # LDI R2,0x01
1c 00   # LDI R3,0x00   R0=0x00, R1=0x04, R2=0x01, R3=0x00
```

```
35
36 46       # SUB R1,R2
37 d0 fd    # BNZ 0xfd       PC=PC-3
38
39 c0 01    # BZ +1 (skip over the following HALT
40 ff       # HALT
41
42 ee       # HALT
43 ff       # HALT
```

Example Output

```
1  Memory before execution begins:
2  0000: 00 A0 10 FF FF FF FF FF  FF FF FF FF FF FF FF FF
3  0010: 10 11 10 00 10 60 30 20  F0 40 14 55 18 50 56 76
4  0020: 66 8D 90 F0 B0 02 FF FF  00 10 00 14 04 18 01 1C
5  0030: 00 46 D0 FD C0 01 FF EE  FF 00 00 00 00 00 00 00
6  0040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
7  0050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8  0060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
9  0070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
10 0080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
11 0090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
12 00A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
13 00B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
14 00C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
15 00D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
16 00E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
17 00F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
18
19 00: 00    NOP             #  R0=00 R1=00 R2=00 R3=00 ----
20 01: a0 10 B    0x10        #  R0=00 R1=00 R2=00 R3=00 ----
21 10: 10 11 LDI  R0,0x11     #  R0=00 R1=00 R2=00 R3=00 ----
22 12: 10 00 LDI  R0,0x00     #  R0=11 R1=00 R2=00 R3=00 ----
23 14: 10 60 LDI  R0,0x60     #  R0=00 R1=00 R2=00 R3=00 ----
24 16: 30    ADD  R0,R0       #  R0=60 R1=00 R2=00 R3=00 ----
25 17: 20 f0 ST   R0,0xf0     #  R0=c0 R1=00 R2=00 R3=00 -N-S
26 19: 40    SUB  R0,R0       #  R0=c0 R1=00 R2=00 R3=00 -N-S
27 1a: 14 55 LDI  R1,0x55     #  R0=00 R1=00 R2=00 R3=00 Z---
28 1c: 18 50 LDI  R2,0x50     #  R0=00 R1=55 R2=00 R3=00 Z---
29 1e: 56    XOR  R1,R2       #  R0=00 R1=55 R2=50 R3=00 Z---
30 1f: 76    OR   R1,R2       #  R0=00 R1=05 R2=50 R3=00 ---S
31 20: 66    AND  R1,R2       #  R0=00 R1=55 R2=50 R3=00 ----
32 21: 8d    MOV  R3,R1       #  R0=00 R1=50 R2=50 R3=00 ---S
33 22: 90 f0 LD   R0,(0xf0)   #  R0=00 R1=50 R2=50 R3=50 ---S
34 24: b0 02 BR   0x02        #  R0=c0 R1=50 R2=50 R3=50 ---S
35 28: 00    NOP              #  R0=c0 R1=50 R2=50 R3=50 ---S
36 29: 10 00 LDI  R0,0x00     #  R0=c0 R1=50 R2=50 R3=50 ---S
37 2b: 14 04 LDI  R1,0x04     #  R0=00 R1=50 R2=50 R3=50 ---S
38 2d: 18 01 LDI  R2,0x01     #  R0=00 R1=04 R2=50 R3=50 ---S
39 2f: 1c 00 LDI  R3,0x00     #  R0=00 R1=04 R2=01 R3=50 ---S
40 31: 46    SUB  R1,R2       #  R0=00 R1=04 R2=01 R3=00 ---S
41 32: d0 fd BNZ  0xfd        #  R0=00 R1=03 R2=01 R3=00 ----
42 31: 46    SUB  R1,R2       #  R0=00 R1=03 R2=01 R3=00 ----
43 32: d0 fd BNZ  0xfd        #  R0=00 R1=02 R2=01 R3=00 ----
44 31: 46    SUB  R1,R2       #  R0=00 R1=02 R2=01 R3=00 ----
45 32: d0 fd BNZ  0xfd        #  R0=00 R1=01 R2=01 R3=00 ----
46 31: 46    SUB  R1,R2       #  R0=00 R1=01 R2=01 R3=00 ----
47 32: d0 fd BNZ  0xfd        #  R0=00 R1=00 R2=01 R3=00 Z---
```

```
34: c0 01 BZ   0x01        #  R0=00 R1=00 R2=01 R3=00 Z---
37: ee    HALT             #  R0=00 R1=00 R2=01 R3=00 Z---
Execution halted after 477 operations.

Memory after execution ends:
0000: 00 A0 10 FF FF FF FF FF  FF FF FF FF FF FF FF FF
0010: 10 11 10 00 10 60 30 20  F0 40 14 55 18 50 56 76
0020: 66 8D 90 F0 B0 02 FF FF  00 10 00 14 04 18 01 1C
0030: 00 46 D0 FD C0 01 FF EE  FF 00 00 00 00 00 00 00
0040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00F0: C0 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```