Homework 05 - Implement Microcode

400 pts (part 1)

Due Apr 20

Emulator at: http://www.2c-why.com/

Example Microcode

The file is attached to the homework. Let’s walk thorough it.

```

  DCL id_memory_Read id_memory_Write
  DCL id_Microcode_PC_Clr id_Microcode_PC_Ld Microcode_PC_Ld2 id_Microcode_PC_Inc
  DCL ir_decode_g1
  DCL id_ac_Out_to_ALU id_ac_Clr id_ac_Inc id_ac_Ld id_ac_Out
  DCL McJmp_0 McJmp_1 McJmp_2 McJmp_3 McJmp_4 McJmp_5 McJmp_6 McJmp_7
  DCL id_ALU_Ctl_0 id_ALU_Ctl_1 id_ALU_Ctl_2 id_ALU_Ctl_3
  DCL id_decoder_Ctl_0 id_decoder_Ctl_1
  DCL hand_out
  DCL id_input_Out
  DCL id_ir_Ld id_ir_Out
  DCL id_mar_Ld id_mar_Out
  DCL id_mdr_Ld id_mdr_Out
  DCL id_output_Ld
  DCL id_pc_Inc id_pc_Ld id_pc_Out
  DCL id_result_Ld id_result_Out

  DCL is_halted is_fetch set_execute ins_end id_ALU_Ctl do_input do_output

// -------------------- Instruction Fetch Cycle ----------------------------

// Decode 1st nibble of instruction
// Move Instruction from Main Memory to IR
// 3 Tick
    ORG        0b0_0000_000                      // 0x00
    id_mar_Ld id_pc_Out                          is_fetch id_Microcode_PC_Inc
    id_mdr_Out id_pc_Inc  id_memory_Read         id_Microcode_PC_Inc
    id_ir_Ld id_mdr_Out                          id_Microcode_PC_Inc
    id_decoder_Ctl_1 id_decoder_Ctl_0   McJmp_7  id_ir_Out ir_decode_g1    id_Micro


  STR By Your Name.....................



```

```
// ------------ Instruction Execute (Implementation) --------------------------


// OpLoad, 0x1xxx => 0x88. 3+2 Ticks.
    ORG        0b1_0001_000             // 0x88
    hand_out id_ir_Out   id_mar_Ld      set_execute id_Microcode_PC_Inc  // Move HA
    id_memory_Read                      id_Microcode_PC_Inc
    id_ac_Ld   id_mdr_Out               ins_end id_Microcode_PC_Clr      // Move MD

// OpStore, 0x2xxx => 0x90. 3+2 Ticks.
    ORG        0b1_0010_000             // 0x90
    hand_out   id_ir_Out id_mar_Ld      set_execute id_Microcode_PC_Inc // Move HAN
    id_mdr_Ld   id_ac_Out               id_Microcode_PC_Inc             // Move AC
    id_memory_Write                     ins_end id_Microcode_PC_Clr     // Write













// Jumps and Store: Stores value of PC at address X then increments PC to X+1
// OpJnS, 0x0xxx => 0x80.  3+6 Ticks.
    ORG        0b1_0000_000                             // 0x80
  // TODO

// OpAdd, 0x3xxx => 0x98. 3+3 Ticks.
    ORG        0b1_0011_000                             // 0x98
  // TODO

// OpSubt, 0x4xxx => 0xA0. 3+3 Ticks.
    ORG        0b1_0100_000                             // 0xA0
  // TODO

// OpInput, 0x5xxx => 0xA8. 3+1 Ticks.
    ORG        0b1_0101_000                             // 0xA8
  // TODO

// OpOutput, 0x6xxx => 0xA8. 3+1 Ticks.
    ORG        0b1_0110_000                             // 0xB0
  // TODO

```

```
// OpHalt, 0x7xxx => 0xB0. 3+1 Ticks.
    ORG       0b1_0111_000                            // 0xB8
  // TODO

// OpJump, 0x9xxx => 0xE8. 3+1 Ticks.
    ORG       0b1_1001_000                            // 0xC8
  // TODO

// OpClear, 0xAxxx => 0xD8. 3+1 Ticks.
    ORG       0b1_1010_000                            // 0xD8
  // TODO

// OpLoadI, 0xD => 0xE8 address.  3+4 = 7 Ticks
  ORG      0b1_1101_000                          // 0xE8
  // TODO

// OpStoreI 0xE, => 0xf8 address. Ticks 3+3
  ORG      0b1_1110_000                          // 0xF0
  // TODO

// OpUnused 0xF, => 0xf8 address. ?forever?
  ORG      0b1_1111_000                          // 0xF8
  // TODO

// OpAddI 0xB, => 0xf8 address. Ticks 3+5
  ORG      0b1_1011_000                          // 0xd8
  // TODO

// OpJumpI 0xCxxx, => 0xE0
  ORG      0b1_1100_000                          // 0xE0
  // TODO


// OpSkip[XXXX] --- OpSkipLt0, OpSkipEq0, OpSkipGt0: 3+3
    ORG       0b1_1000_000
  // TODO

    ORG       0b01_0000_00                                        // 0x40
  // TODO

    ORG       0b01_0100_00                                        // 0x50
  // TODO

    ORG       0b01_1000_00                                        // 0x60
  // TODO

  ORG       0b0010_0000                           // 0x20 (no skip)
  // TODO

  ORG       0b0010_0010                           // 0x22 (skip)
  // TODO
```

```
138
139
140
141 __end__
142
...

// Lines removed for this section of homework - see ./final.m2
```

You can bring up the emulator at http://www.2c-why.com

First take the file and change ‘STR’ (on line 32) with your name to your name. This is important because of how the assembled files are stored on the server.

Lines 2 to 18 - Have the decleration for all the liens that you can turn on and off. If you do not turn a line on it is off. For example, on line 29 McJmp_7 is turned on. This means that McJmp_0 to McJmp_6 are off.

Every register (PC, IR, AC etc - has a set of common lines. Not all of them are available in this emulation. The Lines are: - Ld Load the register from its inputs - Inc Increment the register - Clr Set the register to 0 - Out Output the values from the register Most of the registers are 16 bit registers. The exception is the Microcode_PC is only 8 bits.

Some of the registers have special lines: - IsZero - all all the values in the register 0 (on the Result register) - Out_To_ALU - output data from the AC to the ALU (A) side.

The special hardware that makes all of this work is the “hand_out” that isolates 12 bits from the IR register and puts that onto the buss. The most significant 4 bits are set to 0. To do this you have to turn on the output from the IR at the same time. Line 29 above has this.

At the same time Line 29 also loads the Microcode_PC register. The way this works is that the set of input to the Mux/Decodeer is

Bit	Loaded With
7	1 from turning on McJmp_7
6	From IR : Bit 15
5	From IR : Bit 14
4	From IR : Bit 13
3	From IR : Bit 12
2	0 from McJmp_2 - not turned on

Bit	Loaded With
1	0 from McJmp_1 - not turned on
0	0 from McJmp_0 - not turned on

This is the set of inputs that is sen on the “11” input to the Mux/Decoder. Then the mux has it’s “11” input sent to its output.

For example: A load instruction from address with a hand of 3 is 0x1003. The bits 15..12 in the instuction are 0x1, or 0b0001 so…:

Bit	Loaded With	Value
7	1 from turning on McJmp_7	1
6	From IR : Bit 15	0
5	From IR : Bit 14	0
4	From IR : Bit 13	0
3	From IR : Bit 12	1
2	0 from McJmp_2 - not turned on	0
1	0 from McJmp_1 - not turned on	0
0	0 from McJmp_0 - not turned on	0

In Binary this is 0b1000_1000 or in Hex 0x88. This is the value that is loaded into the Microcode_PC. This takes us to adress 0x88 for the next microcode instruction and that is on line 44 in the code.

Another example with the Halt instruction. It is 0x7000 so the binary for it is 0b_0111_0000_0000_0000. The Instruction is 0x7 or 0x0111.

Bit	Loaded With	Value
7	1 from turning on McJmp_7	1
6	From IR : Bit 15	0
5	From IR : Bit 14	1
4	From IR : Bit 13	1
3	From IR : Bit 12	1
2	0 from McJmp_2 - not turned on	0
1	0 from McJmp_1 - not turned on	0
0	0 from McJmp_0 - not turned on	0

This is a 0x1011_1000 or a 0xB8 address.

The code has all the locations set for the primary jumps from decoding the instructions. This is the primary decode of the instructions. One instruction has a secondary decode. That is the Skipcond set of 3 instructions. OpSkipLt0, OpSkipEq0, OpSkipGt0. These instructions will jump to 0b1_1000_000 or 0xC0. After the primary decode a secondary decode with the Mux/Decoder set to "10" has to happen. In this case the lines are set so that:

Bit	Loaded With
7	1 from turning on McJmp_7
6	1 from turning on McJmp_6
5	From IR : Bit 11
4	From IR : Bit 10
3	From IR : Bit 9
2	From IR : Bit 8
1	0 from McJmp_1 - not turned on
0	0 from McJmp_0 - not turned on

Notice that you can set McJmp_7 and McJmp_6 to the most significant 2 bits of the destination address. It is *important* that you do not send the secondary jump to the same location as other addresses that are already used. In my implementation of the microcode I left McJmp_7 as a 0 and set McJmp_6 to a 1. The locations for the instructions on lines 131 to 135 reflect this. With this assumption and these 4 lines we can then have a decode of the SkipEq instructions 0x8400 with:

Bit	Loaded With	Value
7	1 from turning on McJmp_7	0
6	1 from turning on McJmp_6	1
5	From IR : Bit 11	0
4	From IR : Bit 10	1
3	From IR : Bit 9	0
2	From IR : Bit 8	0
1	0 from McJmp_1 - not turned on	0
0	0 from McJmp_0 - not turned on	0

This takes us to an address of 0b01_0100_00 or 0x50. This is on line 127.

At that point we need to skip an instruction if the condition is matched. The “Is Zero” output from the Result register is hooked to the decoder mux. Set the mux to “10” and Load the Microcode_PC register. The key is that the other inputs to the “10” mux are:

Bit	Loaded With
7	1 from turning on McJmp_7
6	1 from turning on McJmp_6
5	1 from turning on McJmp_5
4	1 from turning on McJmp_4
3	1 from turning on McJmp_3
2	1 from turning on McJmp_2
1	Is Zero
0	0 from McJmp_0 - not turned on

This allows us to jump to some other address when “Is Zero” is 0 or 1.

We can also jump to any address we want to. Set the Mux to “00” and ALL the McJmp_XXXX are connected to the mux. This is a GOTO operation.

Bit	Loaded With
7	1 from turning on McJmp_7
6	1 from turning on McJmp_6
5	1 from turning on McJmp_5
4	1 from turning on McJmp_4
3	1 from turning on McJmp_3
2	1 from turning on McJmp_2
1	1 from turning on McJmp_1
0	1 from turning on McJmp_0

The way that I implemented the Halt instruction is turning on the is_halted line (imprtant) and the doing a jump back to the current microcode location forever in a loop.

The ALU

Our ALU has 4 control inputs:

i3	i2	i1	i0	Used	Action Taken
0	0	0	0	*	2s Compliment
0	0	0	1		
0	0	1	0	*	Increment by 1, ac + 1 -> Result
0	0	1	1		Decrement by 1, 2s compliment, result = ac - 1
0	1	0	0	*	Add: result = ac + bus (mdr usually)
0	1	0	1	*	Sub: subtract A - B
0	1	1	0		A >> B - Arithmetic - fills with MSB
0	1	1	1	*	A == B - if A == B, result <- 1
1	0	0	0		Compliment: Toggle each bit in result = ^ac
1	0	0	1	9 *	1 if AC less than 0, 2s compliment
1	0	1	0	A *	1 if AC greater than 0, 2s compliment
1	0	1	1	B	A and B
1	1	0	0	C	A or B
1	1	0	1	D	A xor B
1	1	1	0	E	A >> B - logical - 0 fill - Shift Right
1	1	1	1	F	A << B - logical - 0 fill - Shift Left

You will ned to set the ALU to do some of the operations.

For example: To do the ADD instruction you need to turn on the `id_ac_Out_to_ALU` . This is to get the data from the AC to the A side of the ALU. You need to turn on the `id_mdr_Out` so that the B side of the alu has the other side of the add operation. Then turn on `id_ALU_Ctl` and `id_ALU_Ctl_2` . The other ALU signals should be 0. Turn on the Result LD signal so that it will load data into the register. This is `id_result_Ld` Don't forget to `id_Microcode_PC_Inc` so that you go to the next Microcode Instruction!.

At the beginning of each instruction it is important to turn on the `set_execute` line. This tells the emulator that you have left the "fetch" part of the instruction and are now running the body of the instruction. At the last step in the instruction you need to `ins_end` and `id_Microcode_PC_Clr` . The `ins_end` will tell the emulator that you have finished an instruction. The `id_Microcode_PC_Clr` will clear the Microcode PC so that you start the next fetch of the next instruction.

All of the instruction lines are case sensitive. `id_ac_Ld` is not the same as `id_AC_ld` and will not work.

Example Assemble (on your system)

You can download and use the microcode assembler at the command line.

```
$ mcasm --in microcode-file.m2 --out hex-file.hex --upload
```

The upload flag tells the assembler to automatically upload the hex file to the sever. You have to connected to the web to do this. At the end it will output a long hex value. You need to cut and paste this into the http://www.2c-why.com application to tell it to load your code in the emulator.

Example Assemble online

In a similar fusion you can use the command line assembler to assemble your MARIE instructions and create a .hex file for them.

```
$ asm --in marie.mas --out marie.hex --upload
```

You will need a current (updated) version of the assemblers if you are going to use them at the command line. Note the “upload”. This will also output a large hex number that identifies your code.

2 Kinds of Assembly

The microcode has 2 different kinds of assembled hex code. The MARIE instructions that get loaded into main memory and the microcode instructions that get loaded into the microcode memory.

The web front end has both of the assemblers bit into it. You can cut/paste your code into the web front end and assemble code online. These are the bottom two buttons on the left.

Implement Add Instruction

For this homework implement the Add, Subt, JnS, Jump and Clear instructions. Turn in your microcode file. Also verify that these instructions work with test cases.

Notes

The ./final.m2 file has tables in it with useful information after the **end** . The **end** is the end of the code and the rest of the file is all just comments.

Copyright