ECE 120: Introduction to Computing Spring 2019 ZJUI

due at start of section on Friday 10 May 2019

Homework 11: LC-3 Machine Language

1. von Neumann Machines

Imagine that we change LC-3 memory to contain 1MB of byte-addressable memory. In other words, 2^{20} addresses, each holding 8 bits. Instructions remain as 16 bits (so now each instruction takes two consecutive memory locations).

a. How many bits are now needed for the PC? 20
b. How many bits are now needed for the IR? 16
c. How many bits are now needed for the MAR? 20
d. How many bits are now needed for the MDR? 8
e. Is instruction fetch faster, slower, or unaffected? Explain your answer.
Instruction fetch requires two memory accesses instead of one memory access, so it is slower.

2. Instruction Formats

The chief architect in charge of designing your company's new processor has drafted an ISA that includes many operations: ADD, SUBTRACT, XOR, XNOR, AND, NAND, OR, NOR, and NOT. The total number of opcodes (including operations, data movement, and control flow) is 19. The chief architect suggests having 15 registers in the register file. Instructions that require three register operands then need ceiling $[\log_2 (19 \times 15 \times 15 \times 15)] = 16$ bits.

In a few sentences, explain to the chief architect why eliminating a few of the opcodes and allowing 16 registers is a better choice in terms of the microarchitecture. For credit, your response must also allow for 16-bit instructions. Be specific about which opcodes should be eliminated.

Mixing the fields used for opcode and three registers in the instruction encoding requires a significant amount of logic to pull them apart. On the other hand, removing NAND, NOR, and XNOR reduces the opcode count to 16, allowing 16-bit registers while retaining 16-bit instructions. Software that needs NAND, NOR, or XNOR can use AND, OR, or XOR (respectively) followed by NOT, so the impact in that regard is minimal. The extra register is also a slight advantage for software.

3. Machines are Busy

Do problem 5.6 from Patt and Patel. Assume that the BUSYNESS vector contains bits for 16 machines instead of the 8 discussed in Section 2.7.1. Note that your instructions must be encoded into bits.

a. ; AND R3,R2,#4 (any DR will do except R2)
0101 011 010 1 00100
b. ; AND R3,R2,#12 (any DR will do except R2)
0101 011 010 1 01100
c. ; ADD R3,R2,#1 (any DR will do except R2)
0001 011 010 1 00001
d. Yes, if the value x0040 is available in a register, we can use AND to extract machine 6's bit (such as AND R3,R2,R4, given x0040 in R4). If not, we need more than one instruction.

4. **Reasoning About Offsets**
Do problem 5.24 from Patt and Patel.

LDR uses a 6-bit 2's complement offset, so the largest offset is 31 decimal (x1F hex). Adding this offset to x4011 gives the largest address for the LDR, x4030.

If the offset were instead a 6-bit unsigned value (zero-extended), we could add 63 decimal (x3F hex) to reach address x4050. The smallest unsigned offset is 0, so the smallest address for the LDR in this case is x4011.

5. **Executing XOR**
Write a sequence of LC-3 instructions (in bits) to set R0 equal to R1 XOR R2. Assume that values have already been placed into R1 and R2 for you. You may not change the values of any other registers (only R0, R1, and R2). Include RTL or assembly comments explaining the action of each binary instruction. *Hints: You MAY change R1 and R2. Use at most eight instructions.*

Recall that A XOR B = { [A' B]' [A B']' }'. We need to use R0 to compute the first complemented factor, then use R1 or R2 to compute the second. Then we can merge the final solution back into R0. For example:

```
1001 000 001 111111   ; NOT R0,R1
0101 000 000 0 00 010 ; AND R0,R0,R2
1001 000 000 111111   ; NOT R0,R0 -- R0 is now [R1' R2]'
1001 010 010 111111   ; NOT R2,R2
0101 010 010 0 00 001 ; AND R2,R2,R1
1001 010 010 111111   ; NOT R2,R2 -- R2 is now [R1 R2']'
0101 000 000 0 00 010 ; AND R0,R0,R2
1001 000 000 111111   ; NOT R0,R0
```

6. **Understanding Induction**
The following LC-3 instructions execute starting from the point shown by the comment.

```
; start LC-3 execution here
0101 011 011 1 00000
0001 011 011 1 00001
0001 100 100 0 00 100
0000 101 111111101
; end LC-3 execution here
```

After the code reaches the end of the code (the last comment), what bits are held in R3? And in R4? And in R5? If you cannot know the bits held, explain why.

This code shifts bits left in R4 until R4 is 0, counting the number of iterations required with R3. Since the initial value of R4 is unknown, we also cannot know how many iterations the loop requires, so R3 is also unknown. R5 remains at its initial value, but we can't know those bits, either.

So R3 is unknown, R4 is 0, and R5 is unknown.

7. Understanding Memory Accesses

The following LC-3 instructions execute starting from the point shown by the comment.

```
1010 1011 1100 1101 ; this is location D
; start LC-3 execution here
0010 001 111111110
0001 001 001 1 00110
0101 011 001 1 01111
1011 011 111111011
; end LC-3 execution here
```

After the code reaches the end of the code (the last comment), what bits are held in R1? And in R3? And in memory location D? If you cannot know the bits held, explain why.

This code loads the value xABCD from D into R1, adds #6 to it to obtain xABD3 in R1, then ANDs that result with x000F to place x0003 into R3. The code then stores x0003 to memory location xABCD with an STI.

So R1 holds xABD3, R3 holds x0003, and memory location D still holds xABCD.

8. Understanding Loops

The following LC-3 instructions execute starting from the point shown by the comment.

```
0000 0000 0000 0110 ; this is location D
0000 0000 0000 0111 ; this is location D + 1
; start LC-3 execution here
0010 001 11111101
0010 010 11111101
0101 011 110 1 00000
0001 011 001 0 00 011
0001 010 010 1 11111
0000 001 111111101
1110 100 111110111
0111 011 100 000001
; end LC-3 execution here
```

After the code reaches the end of the code (the last comment), what bits are held in R1? And in R2? And in R3? And in R4? And in memory location D? And in memory location D + 1? If you cannot know the bits held, explain why.

This code loads #6 into R1, #7 into R2, and #0 into R3. Then it adds R1 into R3 for R2 loop iterations. R2 ends up as 0, while R3 ends up as #6 times #7, or #42. The code then loads address D into R4 and stores #42 into D+1.

So: R1 holds #6, R2 holds 0, R3 holds #42, R4 holds address D (which is unknown), memory location D holds #6, and memory location D + 1 holds #42.

9. Understanding Stability

The following LC-3 instructions execute starting from the point shown by the comment.

```
1111 0000 0000 0000 ; this is location D
; start LC-3 execution here
0101 011 011 1 00010
0010 101 111111101
1001 010 011 111111
0101 011 101 0 00 011
0101 101 101 1 11100
0000 100 111111100
; end LC-3 execution here
```

After the code reaches the end of the code (the last comment), what bits are held in R2? And in R3? And in R5? And in memory location D? If you cannot know the bits held, explain why.

This code never terminates, so it's fair to say that all of the values are unknown.

However, one can also reason that R3, which is first set to either 0 or 2, and is later ANDed with xF000, so R3 must reach and remain at 0 at some point during execution. Similarly, R2 will become xFFFF. R5 is fixed at xF000. And memory location D remains at xF000. These answers are also acceptable.