Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 460N, Spring 2017
Problem Set 1
Due: January 30, before class
Yale N. Patt, Instructor
Chirag Sakhuja, Sarbartha Banerjee, Jon Dahm, Arjun Teh, TAs

Instructions

You are encouraged to work on the problem set in groups and turn in one problem set for the entire group. The problem sets are to be submitted on Canvas. Only one student should submit the problem set on behalf of the group. The only acceptable file format is PDF. Include the name of all students in the group in the file.

Note: You still need to bring a hard copy of your student information sheet to the class on Monday.

You will need to refer to the assembly language handouts and the LC-3b ISA on the course website.

Update 01/26/17: The questions in red (14-19, inclusive) have been moved to PS2. Please make sure to submit the student information sheet with PS1.

Questions

1. Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy. What information does the compiler *need* to know about the microarchitecture of the machine in order to compile the program correctly?

Classify the following attributes of LC-3b as either a property of its microarchitecture or ISA:

1. ISA
2. Microarchitecture
3. ISA
4. Microarchitecture
5. ISA
6. Microarchitecture
7. ISA
8. Microarchitecture
9. Microarchitecture

10. Microarchitecture

2. The first program causes x0004 to be stored in location x3000 when the assembled code is loaded into the memory. The second program causes x0004 to be stored in x3000 during the execution of the program.

3.

Instruction	**Operate**	**Data Movement**	**Control**
ADD	X		
AND	X		
BR			X
JMP/RET			X
JSR/JSRR			X
LDB		X	
LDW		X	
LEA		X	
RTI			X
SHF	X		
STB		X	
STW		X	
TRAP			X
XOR/NOT	X		

4. Both the second and the third instruction (never branch) can be used as a NOP. The first instruction (ADD) sets the condition codes based on the value in R1, therefore it is also not a NOP.

5. Consider the following possibilities for saving the return address of a subroutine:

1. The first option does not support either subroutine nesting or subroutine recursion.
2. The second option supports subroutine nesting, but does not support recursion.
3. The third option supports both subroutine nesting and subroutine recursion.

6.

1. Little endian: MEM[x1000] + MEM[x1002] = x1A0E + x0C11 = x261F
2. Big endian: MEM[x1000] + MEM[x1002] = x0E1A + x110C = x1F26

7. Say we have 32 megabytes of storage, calculate the number of bits required to address a location if

1. The memory consists of 2^{28} addressable locations. 28 bits are required to uniquely address each location.
2. The memory consists of $2^{28}/2^{3}$ addressable locations. 25 bits are required to uniquely address each location.
3. The memory consists of $2^{28}/2^{7}$ addressable locations. 21 bits are required to uniquely address each location.

8.

1. Write the assembly language code for calculating the expression (do not simplify the expression):

   X = (A + (B × C)) × (D - (E + (D × C)))

   a. In a zero-address machine

   ```
   PUSH A
   PUSH B
   PUSH C
   MUL
   ADD
   PUSH D
   PUSH E
   PUSH D
   PUSH C
   MUL
   ADD
   SUB
   MUL
   POP X
   ```

   Advantages: Operate instructions only require an opcode so they can be encoded very densely.
   Disadvantages: Not flexible in terms of manipulating operands, more instructions required to write programs.

   b. In a one-address machine

   ```
   LOAD B
   ```

```
MUL C
ADD A
STORE A
LOAD C
MUL D
ADD E
STORE E
LOAD D
SUB E
MUL A
STORE X
```

Advantages: Only a single register required, fewer instructions compared to the stack machine.
Disadvantage: Not as flexible as 2 or 3-address machines. We need instructions to move data to and from the accumulator (We don't need these in a 3-address machine that supports memory-to-memory operations).

c. In a two-address machine

```
MUL B, C
ADD A, B
MUL C, D
ADD C, E
SUB D, C
MUL A, D
SUB X, X   ; these two instructions emulate a MOV X, A
ADD X, A
```

d. In a three-address machine like the LC-3b, but which can do memory to memory operations and also has a `MUL` instruction.

```
MUL X, B, C
ADD A, X, A
MUL X, D, C
ADD X, E, X
SUB X, D, X
MUL X, X, A
```

9.

1. 1 address machine
2. 4 registers
3. Let's say you want `MOV R2, R1` (R2 = R1)

```
AND R2, #0
ADD R2, R1
```

or

```
LEA TEMP   ; Absolute address of TEMP in R0
ST  R1, R0
LD  R2, R0
...
...
TEMP .FILL xDEAD
```

4. How can we make a PC-relative branch? (HINT: You will need more than one LC-1b instruction; also, note that the LEA instruction does NOT set condition codes)

10.
1. The assembler creates a symbol table after the first pass. Show the contents of this symbol table.

Symbol	**Address**
AGAIN	x300E
NO	x3022
B	x3024
A	x3026

2. If the high and low byte of the word stored at location x4000 are equal, R5 is set to 1, else to 0.
3. There are several possible answers to this code optimization question
   a. The programmer used a loop to left shift a value in R2 by 8 bits. He/she could have done this using a single LC-3b instruction, LSHF. He/she should have replaced the loop

```
AGAIN   ADD R2, R2, R2
        ADD R3, R3, #-1
        BRp AGAIN
```

   with

```
        LSHF R2, R2, #8
```

   b. Instead of using "subtraction" to compare the high and low byte, the programmer could have used a single XOR instruction to check the equality of the two bytes. He/she could have replaced the following instructions

```
        NOT R1, R1
        ADD R1, R1, #1
```

```
        ADD R2, R2, R1
```

with

```
        XOR R2, R1, R2
```

c. The program is comparing the high byte and low byte of the word in memory location x4000. The programmer could have utilized the LDB instruction to load the high byte and low byte into two separate registers and compare them. This way there would be no need for shifting and masking. The optimized program could look like this:

```
        .ORIG  x3000
        AND    R5, R5, #0
        LEA    R0, A
        LDW    R0, R0, #0
        LDB    R1, R0, #0
        LDB    R2, R0, #1
        XOR    R2, R1, R2
        BRnp   NO
        ADD    R5, R5, #1
NO      HALT
A      .FILL   x4000
       .END
```

11.

1.

```
        .ORIG x4000
MAIN    LEA   R2,L0           x4000 xE403
        JSRR  R2              x4002 x4080
        JSR   L1              x4004 x4803
        HALT                  x4006 xF025
        ;
L0      ADD   R0,R0,#5        x4008 x1025
        RET                   x400A xC1C0
        ;
L1      ADD   R1,R1,#5        x400C x1265
        RET                   x400E xC1C0
```

2. As long as the subroutine we are calling is located at most 1023 instructions before the JSR instruction or at most 1024 instructions after the JSR instruction, there is no need to use the first way, which requires at least two instructions. This is due to the fact that JSR instruction can change the PC to an address within the range PC + 2 - 2048 and PC + 2 + 2046, because it uses a limited offset of 11 bits.

The method that requires two is necessary if the subroutine we are calling is not within the range of the JSR instruction. Note that the JSRR instruction can change the PC to any address residing in its base register. The address in the base register can be set to any address in memory.

For example if we have a program starting at memory location x3000 and we would like to call a subroutine starting at memory location 0xF000, there is no way to do this by just using a JSR instruction at location x3000. However, we can call the subroutine at 0xF000 using JSRR with the following sequence of instructions:

```
           .ORIG   x3000
            LEA    R0, SUBADDR
            LDW    R1, R0, #0
            JSRR   R1
            HALT
SUBADDR     .FILL  xF000
            .END
```

12. Yes, there is a difference. The program at x5000 does not save the return address from the subroutine at L2 because it uses a JMP instruction. Thus, that program will not work correctly.

A possible change that could be made:

```
      .ORIG x5000
      LEA   R7, B
MAIN2 LEA   R3, L2
A2    JMP   R3
B     HALT
      ;
L2    ADD   R2, R1, R0
      RET
```

13. Possible solutions:
    1. 4 opcode bits, 3 NZP bits, 3 DR bits, 3 SR1 bits, 3 SR2 bits
    2. A 2 address operation can be used instead of a 3 address. Thus, we have 4 opcode bits, 3 NZP bits, 3 DR/SR1 bits, 1 steering bit. The remaining 5 bits will either be used as 5 immediate bits, or 2 unused bits + 3 SR2 bits.

       Variable instruction length ISAs have more complex decode logic. Variable instruction length ISA programs can be encoded more densely. Variable instruction length ISAs also generally imply a richer instruction set than that of a fixed length ISA. Since a richer instruction set has more instructions that directly correspond to higher level language programming constructs, the compilation process can be easier.