Department of Electrical and Computer Engineering

The University of Texas at Austin

Name: **Xinyuan (Allen) Pan**
EE 460N, Spring 2017
Problem Set 1
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?

ISA is the specification of the interface between the software programs and the computer hardware. It specifies the set of instruction that can be used by the program and can be understood and carried out by the computer.
Microarchitecture is the underlying implementation of a machine.
The compiler does not need to know anything about the microarchitecture.

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

1. There is no subtract instruction in LC-3b.
2. The ALU of LC-3b does not have a subtract unit.
3. LC-3b has three condition code bits (n, z, and p).
4. The n, z, and p bits are stored in three 1-bit registers.

5. A 5-bit immediate can be specified in an `ADD` instruction
6. It takes *n* cycles to execute an `ADD` instruction.
7. There are 8 general purpose registers used by operate, data movement and control instructions.
8. The registers MDR (Memory Data Register) and MAR (Memory Address Register) are used for Loads and Stores.
9. A 2-to-1 mux feeds one of the inputs to ALU.
10. The register file has one input and two output ports.

ISA: 1, 3, 5, 7
Microarchitecture: 2, 4, 6, 8, 9, 10

2. Both of the following programs cause the value `x0004` to be stored in location `x3000`, but they do so at different times. Explain the difference.

1. First program:

```
        .ORIG x3000
        .FILL x0004
        .END
```

2. Second program:

```
              .ORIG x4000
x4000         AND R0, R0, #0   R0=0
x4002         ADD R0, R0, #4   R0=4
x4004         LEA R1, A        R1=x400C
x4006         LDW R1, R1, #0   R1 <- M[x400C] <- x3000
x4008         STW R0, R1, #0   M[x3000] <- 4
x400A         HALT
x400C   A     .FILL x3000
x400E         .END
```

Program 1 fills the value x0004 in location x3000 at during the assemble process (second pass)

Program 2 puts x0004 in location x3000 during the run time of the program.

3. Classify the LC-3b instructions into Operate, Data Movement, or Control instructions.
Operate: ADD, AND, NOT, LSHF, RSHFL, RSHFA, XOR
Data Movement: LDB, LDW, LEA, STB, STW
Control: BR, JUMP, RET, JSR, JSRR, RTI, TRAP

4. At location `x3E00`, we would like to put an instruction that does nothing. Many ISAs actually have an opcode devoted to doing nothing. It is usually called NOP, for NO OPERATION. The instruction is fetched, decoded, and executed. The execution phase is to do **nothing**! Which of the following three instructions could be used for NOP and have the program still work correctly?

1. `0001 001 001 1 00000`
2. [✓] `0000 111 000000000`
3. [✓] `0000 000 000000000`

For each of the three that can not be used for NOP, explain why.
instruction 1 differs from NOP because may change the condition code.

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

1. In a processor register.
2. In a memory location associated with the subroutine; i.e., a different memory location is used for each different subroutine.
3. On a stack.

Which of these possibilities supports subroutine nesting, and which supports subroutine recursion (that is, a subroutine that calls itself)?

nesting : 2, 3
recursion : 3

6. A small section of byte-addressable memory is given below:

Address	Data
x0FFE	xA2
x0FFF	x25
x1000	x0E
x1001	x1A
x1002	x11
x1003	x0C
x1004	x0B
x1005	x0A

1. x1A0E
x0C11
261F

2. x0E1A
x110C
x1F26

Add the 16-bit two's complement numbers specified by addresses `x1000` and `x1002` if

1. the ISA specifies a little-endian format x261F
2. the ISA specifies a big-endian format x1F26

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

1. the ISA is bit-addressable $32\times10^6\times8 = 256\times10^6$ $\log_2 256\times10^6 = \boxed{28 \text{ bits}}$
2. the ISA is byte-addressable $\log_2 32M = \boxed{25 \text{ bits}}$
3. the ISA is 128-bit addressable $\log_2 2\times10^6 = \boxed{21 \text{ bits}}$

8. A zero-address machine is a stack-based machine where all operations are done using values stored on the operand stack. For this problem, you may assume that its ISA allows the following operations:

- `PUSH M` - pushes the value stored at memory location M onto the operand stack.

- `POP M` - pops the operand stack and stores the value into memory location M.
- `OP` - Pops two values off the operand stack, performs the binary operation OP on the two values, and pushes the result back onto the operand stack.

Note: To compute A - B with a stack machine, the following sequence of operations are necessary: `PUSH A`, `PUSH B`, `SUB`. After execution of `SUB`, A and B would no longer be on the stack, but the value A-B would be at the top of the stack.

A one-address machine uses an accumulator in order to perform computations. For this problem, you may assume that its ISA allows the following operations:

- `LOAD M` - Loads the value stored at memory location M into the accumulator.
- `STORE M` - Stores the value in the accumulator into Memory Location M.
- `OP M` - Performs the binary operation OP on the value stored at memory location M and the value present in the accumulator. The result is stored into the accumulator (ACCUM = ACCUM OP M).

A two-address machine takes two sources, performs an operation on these sources and stores the result back into one of the sources. For this problem, you may assume that its ISA allows the following operation:

- `OP M1, M2` - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M1 (M1 = M1 OP M2).

Note 1: `OP` can be `ADD`, `SUB`, or `MUL` for the purposes of this problem.

Note 2: A, B, C, D, E and X refer to memory locations and can be also used to store temporary results.

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
```

b. In a one-address machine

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

c. In a two-address machine

```
MUL B C
ADD A B
MUL C D
ADD C E
SUB D C
MUL A D
ADD X A (X must be zero at the beginning)
```

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

2. Give an advantage and a disadvantage of a one-address machine versus a zero-address machine.
Advantage: The number of instructions used may be less. The use of accumulator (register) allows temporary storage therefore faster data fetching.
Disadvantage: zero-address machine uses a stack, which allows the storage of more temporary results. As in one-address machine, only one result can be stored in the accumulator.

9. The following table gives the format of the instructions for the LC-1b computer that has 8 opcodes.

Opcode	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADD	0	0	0	DR		A	SR	

AND	0	0	1	DR		A	SR	
BR(R)	0	1	0	N	Z	P	TR	
LDImm	0	1	1	signed immediate				
LEA	1	0	0	signed offset				
LD	1	0	1	DR		0	TR	
ST	1	1	0	SR		0	TR	
NOT	1	1	1	DR		0	0	0

Notes:

- Interpretation of all instructions is similar to that of the LC-3b, unless specifically stated otherwise.
- The destination register for the instructions `LDImm` and `LEA` is always register R0. (e.g. `LDImm #12` loads decimal 12 to register R0.)
- TR stands for Target Register. In the case of the conditional branch instruction `BR`, it contains the target address of the branch. In the case of `LD`, it contains the address of the source of the load. In the case of `ST`, it contains the address of the destination of the store.
- `ADD` and `AND` provide immediate addressing by means of a steering bit, bit[2], labeled A. If A is 0, the second source operand is obtained from SR. If A is 1, the second source operand is obtained by sign-extending bits[1:0] of the instruction. A bit is called a "steering" bit if its value "steers" the interpretation of other bits (instruction bits 1:0 in this case).
- Bits labeled 0 must be zero in the encoding of the instruction.

1. What kind of machine (n-address) does the above ISA specification represent?
   Two-address machine
2. How many general purpose registers does the machine have?
   4
3. Using the above instructions, write the assembly code to implement a register to register mov operation.
   MOV R2, R1 is implemented as: (from R1 to R2)
   AND R2, 0
   ADD R2, R1
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)
   Assume the offset is x, and the offset is relative to the first instruction in this implementation.
   LEA x
   BRcc R0

10. Consider the following LC-3b assembly language program:

```
              .ORIG x3000
x3000         AND R5, R5, #0    R5 ← 0
    2         AND R3, R3, #0    R3 ← 0
    4         ADD R3, R3, #8    R3 ← 8
    6         LEA R0, B         R0 ← B
    8         LDW R1, R0, #1    R1 ← M[R0+2] = x4000
    A         LDW R1, R1, #0    R1 ← M[R1] = M[x4000]
    C         ADD R2, R1, #0    R2 ← M[x4000]
  E AGAIN     ADD R2, R2, R2    R2 ← R2×2   ⎫
x3010         ADD R3, R3, #-1   R3 ← R3 - 1 ⎬ R2 = R2 × 2^8 (shift left 8 times)
    2         BRp AGAIN                     ⎭
    4         LDW R4, R0, #0    R4 ← xFF00
    6         AND R1, R1, R4    R1 ← M[x4000] AND xFF00
    8         NOT R1, R1        } ~~R1 ← R1~~
    A         ADD R1, R1, #1
    C         ADD R2, R2, R1    R2 ← R2 - R1
    E         BRnp NO           if not equal then end
x3020         ADD R5, R5, #1    else return R5 as 1
  2 NO        HALT
  4 B         .FILL XFF00
  6 A         .FILL X4000
x3028         .END
```

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. What does this program do? (in less than 25 words)
   If the first 8 bits and the final 8 bits of value stored in location x4000 are the same, set R5 to 1, otherwise set R5 to 0.
3. When the programmer wrote this program, he/she did not take full advantage of the instructions provided by the LC-3b ISA. Therefore the program executes too many unnecessary instructions. Show what the programmer should have done to reduce the number of instructions executed by this program.
   Instead of using a loop to shift the value in R2 left by 8, we can directly use the shift instruction: LSHF R2, R2, 8

11. Consider the following LC-3b assembly language program.

```
               .ORIG x4000        x4000
x4000   MAIN   LEA R2,L0          xE403
x4002          JSRR R2            x4080
x4004          JSR L1             x4803
x4006          HALT               xF025

x4008   L0     ADD  R0,R0,#5      x1025
x400A          RET                xC1C0

x400C   L1     ADD  R1,R1,#5      x1265
x400E          RET                xC1C0
```

1. Assemble the above program. Show the LC-3b machine code for each instruction in the program as a hexadecimal number.
2. This program shows two ways to call a subroutine. One requires two instructions: `LEA` and `JSRR`. The second requires only one instruction: `JSR`. Both ways work correctly in this example. Is it ever necessary to use `JSRR`? If so, in what situation?
   Yes. Inside a subroutine, when we call RET, we are actually calling JSRR R7, where R7 stores the return address. In this case, since we wouldn't know which address to return to at the time of programming, we must use a register to store the address and use JSRR.

12. Consider the following two LC-3b assembly language programs.

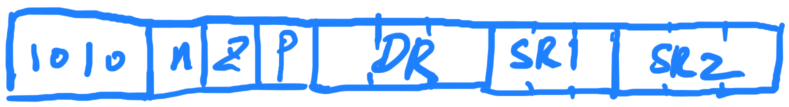
```
      .ORIG x4000                      .ORIG x5000
MAIN1 LEA R3, L1                 MAIN2 LEA R3, L2
A1    JSRR R3                    A2    JMP R3
      HALT                             HALT

L1    ADD R2, R1, R0             L2    ADD R2, R1, R0
      RET                              RET
```

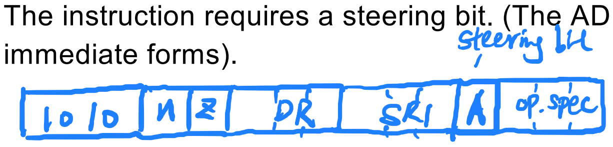
Is there a difference in the result of executing these two programs? If so, what/why is there a difference? Could a change be made (other than to the instructions at Labels A1/A2) to either of these programs to ensure the result is the same?
There is a difference in the result of executing these two programs. In program 1, JSRR instruction automatically stores the incremented PC into R7, so when RET is called, the next instruction becomes the one after JSRR. But in program 2, JMP instruction does not store the next address to R7, so RET cannot be used to return to the correct address.
Add LEA R7, #1 before A2.

13. Use one of the unused opcodes in the LC-3b ISA to implement a conditionally executed ADD instruction. Show the format of the 16 bit instruction and discuss your reasoning assuming that:

1. The instruction doesn't require a steering bit. (The ADD is a register-register operation).



2. The instruction requires a steering bit. (The ADD has both register-register and register-immediate forms).



Since one of the tree conditional codes must be 1, we only need to check 2.

Discuss the tradeoffs between a variable instruction length ISA and a fixed instruction length ISA. How do variable length instructions affect the hardware? What about the software?
Fixed length instructions are easier to decode, but it may waste bits in instructions (thus memory). Variable instruction length saves memory space, but it requires more logic to decode instructions.
Variable length instructions makes the hardware implementation more complex, and it allows software programs to be more compact.

20. Please go to the handouts section of the course web site, print and fill out the student information sheet, and turn it in with a recognizable recent photo of yourself on January 30 (the same day this problem set is due). Please submit a paper copy.