1. RISC/CISC [3 points]

What are the advantages of CISC over RISC?

a) **CISC can perform more advanced instructions than RISC, providing more code flexibility to programmers. (ANS)**

b) CISC is easier to decode than RISC, as CISC has a uniform instruction format.

c) CISC is always faster than RISC, as in, all programs run on CISC would execute faster than RISC.

d) All of the above.

e) None of the above.

2. Floating Point Addition [5 points]

What is the sum of 0.46875 and -1.5 in IEEE 754 notation?

a) 0 01111111 00001000000000000000000

b) **1 01111111 00001000000000000000000 (ANS)**

c) 1 00000000 00001000000000000000000

d) 1 01111111 00011000000000000000000

e) 1 10000000 00000100000000000000000

3. Symbol Table [5 points]

Given the following C file:

myfile.c:

```
int *a;
double b[20];
struct {
   int c;
   float d;
} e;
extern int f;

int foo(int g) {
   static int h = 0;
   int i = g;
   h += bar(*a) + bar(i+f);
   return (h);
}
```

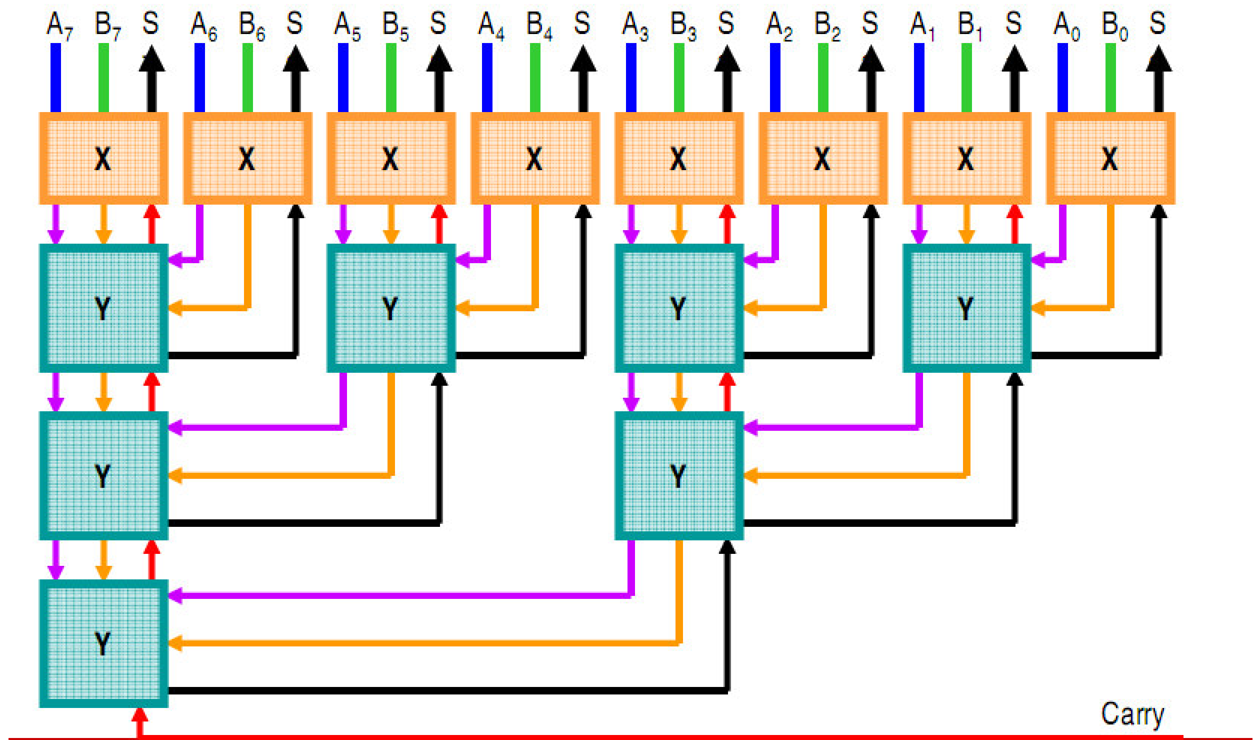
Circle all the symbols that are placed in the symbol table of myfile.o:

a **b** c d **e** **f** g **h** i **foo** **bar**

Technically variable 'h' should be in the symbol table. But we did not penalize you for not placing variable 'h' in the symbol table.

4. Adder based Propagation Delay [5 points]

Refer to the diagram below for an 8-bit Carry Lookahead Adder (CLA). Assume that the worst-case delay of an X block is 3ns and a Y block is 2ns, regardless of the actual operations within the block, what is the overall worst-case delay of a 32-bit Carry Lookahead Adder (CLA)?
Remember to compute the worst-case delay for a 32-bit CLA.



a) 2X + 5Y = 16ns
b) 1X + 7Y = 17ns
c) 2X + 7Y = 20ns
d) 1X + 9Y = 21ns
e) **2X + 9Y = 24ns (ANS)**

5. Data Structure Alignment [10 points]

Consider assigning the variables below to memory locations starting at address **100 (decimal number)**. The alignment rules are as discussed in class. The size of each data type is as follows:

DATA TYPE	SIZE
char	1 byte
short	2 bytes
int or unsigned int or pointer	4 bytes
float	4 bytes
double	8 bytes

```
short a;
double b[10];
struct {
      char c[5];
      float* d;
      short e;
} f;
int z;
```

(a) **[5 points]** What location will z be stored at?
You must show your work on this problem to be eligible for partial credit.

200-203

(b) **[5 points]** Is it possible to reduce the size of the structure? Explain your reason.

Yes, you can do so by rearranging the values inside the struct to remove whitespace generated by alignment problems.

```
struct{
      float* d;
      short e;
      char c[5];
}
```

6. Memory addressing [10 points]:

Given the initial values located in memory locations 100-105, what are the final contents of the memory after the following MIPS instruction sequence is executed?

```
lhu     $2, 104($0)
lb      $3, 101($0)
sw      $2, 100($0)
sh      $3, 104($0)
```

Recall that the register $0 always contains the value zero. The memory is organized in big endian format, the same format we studied in class.

Initial Memory Contents		Final Memory Contents	
Address	Value	Address	Value
100	0x47	100	**0x00**
101	0xB2	101	**0x00**
102	0x06	102	**0x9A**
103	0xEF	103	**0xC8**
104	0x9A	104	**0xFF**
105	0xC8	105	**0xB2**

Final contents of registers

$2	**0x00009AC8**
$3	**0xFFFFFFB2**

7. Caller-saved vs. callee-saved registers [10 points]:

You are once again using the "LowsyCompiler v.0.89" to compile the function below. Assume foo is called by main, and bar is a black-box function called from many places. The architecture that the LowsyCompiler is targeting has **2 caller-saved registers** and **2 callee-saved registers**.

```
int foo(int a)
{
    int w, x, y, z;
    w = a;
    z = 0;
    for (x = 0; x < 5; x++)
    {
        if (w == 0)
        {
            return x;
        }
        else
        {
            y = bar(w);
            z += y;
        }
    }
    return z;
}
```

The LowsyCompiler has mapped the local variables of foo as follows:
Caller-saved registers: **y** and **z**
Callee-saved registers: **w** and **x**
(Assume **a** is mapped to a register that stores parameters and does not need to be saved/restored.)

How many store/load operations did the compiler have to include **in the assembly code** of foo for the purpose of preserving register values **w**, **x**, **y**, and **z**?

(For partial credit, write down the number of added stores and loads for each variable.)

1 sw / 2 lw for w (beginning of function, after each return point)
1 sw / 2 lw for x (beginning of function, after each return point)
0 sw / 0 lw for y (old value not needed after call to bar)
1 sw / 1 lw for z (before call to bar, after call to bar)

3 sw / 5 lw total

Store operations: ____**3**_

Load operations: ____**5**_

8. Assembly Code [17 points]

The following MIPS code (with portions missing) is an implementation of the doit function in assembly. Doit is a recursive function that takes two integer input parameters ($a0, $a1) and returns an integer ($v0).

```
doit    addi $sp,$sp,-8          // allocate memory to stack
        sw $ra,0($sp)            // store return address
        sw $s0,4($sp)            // store callee-save register
        add $s0,$zero,$a1        // move second argument to $s0
        bne $s0,$zero,rec
        addi $v0,$zero,1         // set the return reg to 1
        j ret
rec     subi $s0,$s0,1
        add $a1,$s0,$zero        // pass $s0 as the second parameter
        jal doit                 // recursive call to doit
        mult $v0,$a0             // multiply the return value by $a0
        mflo $v0                 // move lower 32 bits to return reg
ret     lw $s0,4($sp)            // restore callee-save register
        lw $ra,0($sp)            // restore return address
        addi $sp,$sp,8           // deallocate memory from stack
        jr $ra                   // return
```

(a). **[7 pts]** Complete the assembly code/comments for the function doit by filling in the blanks. Note you do not necessarily need to understand what the code does for this part.

(b). **[7 pts]** What does doit() compute? Complete the C code below

```
int doit(int arg1, int arg2)
{
   if (________arg2 == 0____________)
      return 1 ;
   else
      return (____doit(arg1, arg2-1) * arg1______) ;
}
```

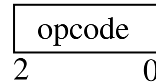
(c). **[3 pts]** What is the maximum size of the stack frame (in bytes) when you call doit(4,6)? You do not need to account for the stack of any other functions (i.e., main()).

Max size = **8bytes * 7 = 56 bytes**

9. A new ISA design [15 points]

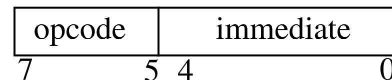
In this problem we want to design a new ISA for an embedded system. One of our key specifications is that the memory used by our assembly code should be minimal. The design is a CISC, stack-based, byte-addressable instruction set named LC-One. LC-One contains a single register, the 8-bit accumulator (called ACC), to assist with all computation. There are 3 instruction formats, each of a different bit length:

R-type instructions – 3-bit instructions:



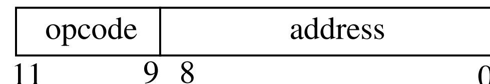
Mnemonic	Opcode	Action
push	001	Copy the value in the ACC onto the top of the stack.
pop	010	Remove value at top of stack and place it into ACC.
halt	011	Halt the machine.

I-type instructions – 8-bit instructions:
(immediate is 5-bit value in 2's complement format)



Mnemonic	Opcode	Action
pushi	100	Place the sign-extended value stored in the immediate field onto the top of the stack.
beq	101	If the value on the top of the stack is equal to the value in ACC branch to PC = PC + 1 + immediate. Otherwise PC = PC + 1.
ALU.X **.add** **.sub** **.mul** **.and** **.xor**	000	Performs some ALU operation in the form of: ACC = ACC (op) (top of stack) Imm = 00000: Add (add) Imm = 00001: Subtract (sub) Imm = 00010: Multiply (mul) Imm = 00100: AND Imm = 01000: XOR There are more possible immediate values, but they are not necessary for this problem.

U-type instructions – 12-bit instructions:
(address is a 9-bit unsigned number)



Mnemonic	Opcode	Action
loadb	110	ACC = MEM[addr] Take the byte value stored at the memory location specified by the address field and put it in ACC.
storeb	111	MEM[addr] = ACC Place the byte value contained in ACC into the memory location specified by the address field.

(a) **[4 pts]** Translate the following program into machine code.

Assembly	Binary	Hexadecimal
loadb 29	**1100 0001 1101**	**0xC1D**
beq -4	**1011 1100**	**0xBC**
ALU.xor	**0000 1000**	**0x08**

(b) **[5 pts]** Translate the following LC-One CISC assembly code segment into a single C statement. Assume that the C variable a is mapped at memory location 32. To get you started, this code segment is assigning a value to variable a.

```
loadb 32
push
ALU.add
push
ALU.mul
pushi 12
ALU.add
storeb 32
halt
```

a = **4a*a + 12**

(c) **[6 pts]** Our LC-One compiler is buggy. Although it gives the correct assembly instructions, they are not in the correct order. Instead of fixing the compiler, your boss unreasonably orders you to reorder the instructions manually. Order the assembly code to perform the following snippet of C code. The C variable *a* is located at memory location 32, and the C variable *b* is located at memory location 42.

```
if(a != 0)
{
      b = b * a;
}
```

END: …

The buggy LC-One assembly code is follows. Reorder the following code to perform the correct function. Fill the new instruction order in the table provided. The stack does not need to be empty upon code completion.

```
          ALU.mul
          loadb 32
          loadb 42
          storeb 42
          push
          pushi 0
          beq END


END:      …
```

pushi 0
loadb 32
beq END
Push
loadb 42
ALU.mul
storeb 42

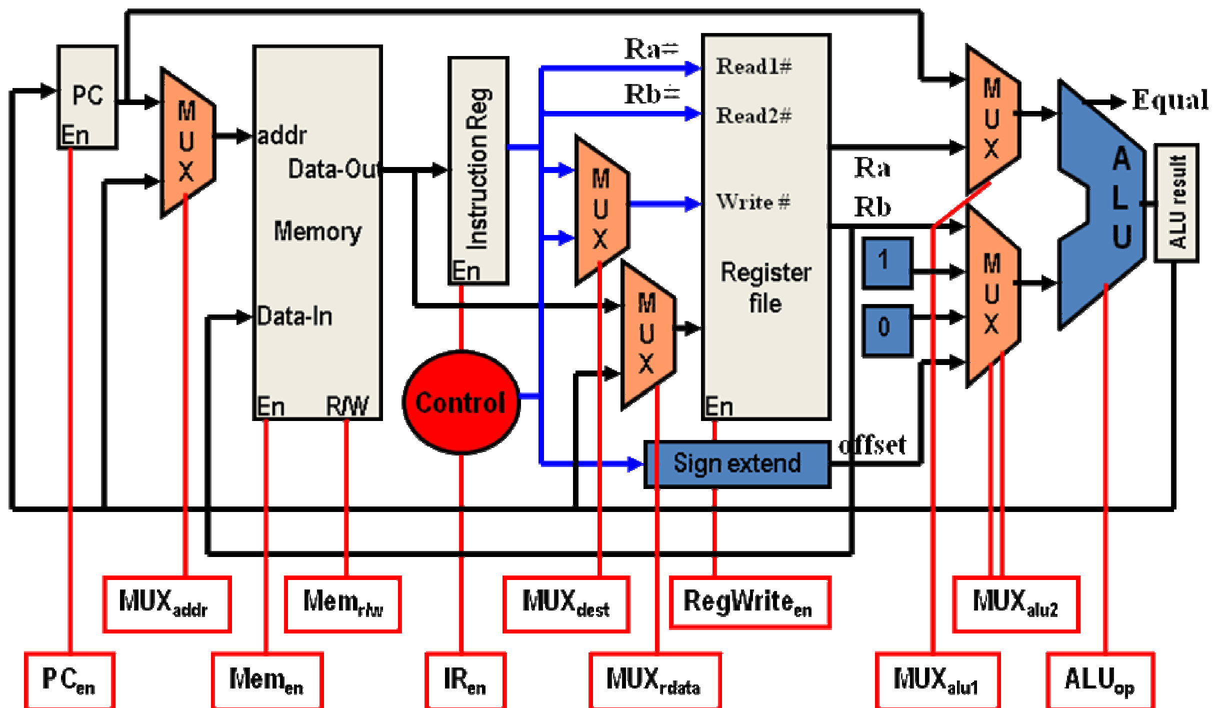
Pushi 0 and loadb 32 are interchangeable.

10. Multi-cycle Datapath [20 points]

Suppose that you want to replace the *no-op* instruction in the LC-2Kx instruction set with an I-type instruction called the indirect branch instruction : ***jmpi regB, regA (offset)***. The new *jmpi* instruction has the following semantics:

```
If (regB != 0)  then
      PC = Memory [ regA + offset]
Else
      PC = PC + 1
```

A. **12 points]** Extend the LC-2kx multi-cycle data-path below to support the *jmpi* instruction. You are only allowed to extend or add new MUXes, add wires to connect components, add new registers or new gates. Show your modifications directly on the diagram below. Below the diagram, list the changes that you have made.



List the changes you have made to the data path

1. Any additional wires to connect components?

   Wire to connect Data-Out to newly added MUX pc
   Wire to connect ALUresult to newly added MUXpc

   Connect 0 register to MUXalu1 (another option is to connect Rb to MUXalu1)

2. Any additional MUXes or extensions to existing MUXes? Any new control signals?

   One new 2:1 MUXpc to select data input to PC (ALUresult or Data-out)
   One additional control signal "jmpi-cntrl" (set only in cycle 4 of jmpi). This is used as the select signal for MUXpc.

   Extend MUXalu1 to a 3:1 MUX (selects PC or Ra or 0 as input operand to ALU).
   One additional control signal for extended MUXalu1.

3. Any additional gates?

   PCen-new = PCen + Equal' * jmpi-cntrl

   Jmpi-cntrl will be set to true only in the 4th cycle of a jmpi instruction (deasserted in all the other states). The above logic is needed to make sure that PC is overwritten with Data-out value in the 4th cycle of jmpi instruction only if Equal' is true.

4. List here any other additional changes that you might have made.

   None

B. **[8 points]** Give a cycle by cycle description when executing the new instruction using the following table. Assume that a value read from the memory can be written to a register in the same cycle. For each cycle, give the following information:

i. Single-sentence description of what the cycle is about.
ii. Psuedo-code describing the semantics of the operation such as register updates.

Use as few cycles as possible.

Cycle1	Fetch instruction. Instruction Register = Memory [PC] ALU Result = PC + 1
Cycle2	Decode Instruction, read registers, and increment PC PC = PC + 1
Cycle 3	**Compute effective address** **ALU Result = [Ra] + offset**
Cycle 4	**Read from memory;** **Compare Rb and 0;** **Write value read from memory to PC if the comparison returns false.** **If ([Rb] != 0)** **PC = Memory [ALU Result]**

C. **BONUS [5 points]** Compilers can generate efficient assembly code if the instruction set supports sophisticated instructions like the indirect jump instruction *jmpi*. Which programming construct(s) will benefit from the indirect jump instruction *jmpi*? Explain your reason in a couple of sentences.

Indirect branches are most useful for handling constructs that branch to one of the many locations in the application code based on a dynamic value. Eg: Switch-case statements and virtual function calls.

A lookup table in memory can be used to store all possible target addresses. The base address of the lookup table can be specified as the immediate value in a jmpi instruction and the register Ra can be used to specify a particular entry in the lookup table.