Problem #1: CPU Performance [12 points]

a) Consider the following C++ code. Assume you generated MIPS assembly code for the C++ code using two different compilers. Compiler 1 generated the code in the first column while compiler 2 generated the code in the second column. Assume that all the instructions used by the compilers are real instructions (as opposed to pseudo-instructions) in the MIPS architecture. Assume arithmetic/logic instructions require 2 cycles to execute, loads 5 cycles, branches 3 cycles, and jumps 1 cycle.

```
void probx(int arr[ ], int n)
{
        for(int i=0 ; i<n ; i=i+1)
                arr[i] = i+1;
}
```

<table>
<tr><th>Compiler 1</th><th>Compiler 2</th></tr>
<tr><td>
<pre>
        add $t0, $zero, $zero
        add $t1, $zero, $zero
loop: bge $t1, $a1, back
        sll $t2, $t1, 2
        add $t2, $t2, $a0
        addi $t3, $t1, 1
        sw $t3, 0($t2)
        addi $t1, $t1, 1
        j loop
back: jal $ra
</pre>
</td><td>
<pre>
        sll $a1, $a1, 2
        sll $t3, $zero, 2
        sll $t1, $zero, 2
        bge $t1, $a1, back
loop: add $t2, $t1, $a0
        addi $t3, $t3, 1
        sw $t3, 0($t2)
        addi $t1, $t1, 4
        blt $t1, $a1 loop
back: jal $ra
</pre>
</td></tr>
</table>

a) Fill the table below with the **number** of instructions of each type that will be executed by the code generated by compiler 1 and by compiler 2 when the function **probx(arr, 100)** is invoked.

Instruction Type	Compiler 1 Code	Compiler 2 Code
Arithmetic/logic	**402**	**304**
Loads	**100**	**100**
Branches	**101**	**101**
Jumps	**101**	**1**

b) Compute the average number of clock cycles per instruction (**CPI**) for each version of the program.

Average CPI (Compiler 1)	**(402*2 + 100*5 + 101*3 + 101*1)/704 = 1,708/704**
Average CPI (Compiler 2)	**(304*2 + 100*5 + 101*3 + 1*1)/506 = 1,412/506**

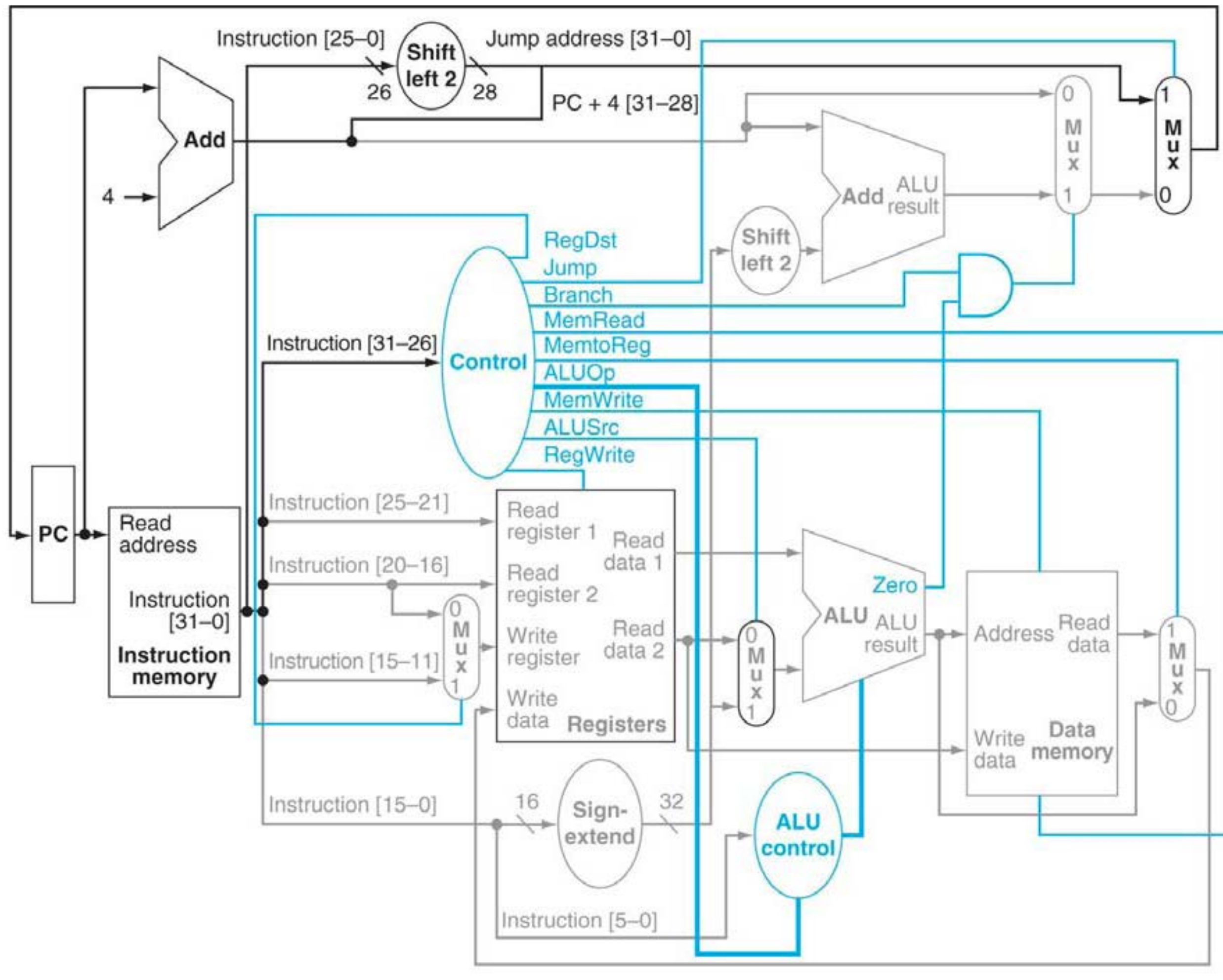
c) Which version of **probx()** is faster if you run it on a 1 GHz processor and by how much?

Answer: ________**1708/1412 = 1.209 → 21%**_______________

Problem #2: Single-Cycle MIPS Control [12 Points]

Consider the single cycle MIPS datapath shown in the figure below. In the Table below, fill in the corresponding values of the 9 control signals when executing the instructions shown on the left hand-side of the table. The ALU control bits are shown in a table at the bottom. Use X to denote don't-care entries where necessary. Do not mark entries with 0 or 1 when a don't-care condition applies.

Instruction	RegDest	Jmp	Brnch	MemRd	MemtoReg	ALUOp	MemWr	ALUSrc	RegWr
or $2, $5,$7	1	0	0	0	0	10	0	0	1
beq $3, $4, L	X	0	1	0	X	01	0	0	0
lw $1, 8($2)	0	0	0	1	1	00	0	1	1
j FINISH	X	1	0	0	X	XX	0	X	0



Instruction opcode	ALUOp	Instruction operation
LW	00	load word
SW	00	store word
Branch equal	01	branch equal
R-type	10	add
R-type	10	subtract
R-type	10	AND
R-type	10	OR
R-type	10	set on less than

Problem #3: MIPS Programs [12 points]

Consider the following MIPS assembly code.

```
origami:
            li $t0, 0
            li $v0, 0
pear:       mul $t1, $t0, 4
            add $t1, $a0, $t1
            lw $t1, 0($t1)
            blt $t1, $0, pizza
            mul $t2, $v0, 4
            add $t2, $a1, $t2
            sw $t1, 0($t2)
            add $v0, $v0, 1
pizza:      add $t0, $t0, 1
            blt $t0, $a2, pear
            jr $ra
```

a) Translate the function `origami` above into C++. You should include a header that lists the types of any arguments and return values. Also, your code should be as concise as possible, without pointers. We will not deduct points for syntax errors unless they are significant enough to alter the meaning of your code.

```
int origami(int A[], int B[], int a2) {
   // array A[] starts at a0 and array B[] starts at a1
   int t0 = 0;
   int v0 = 0;
   do {
        int t1 = A[t0];
        if ( t1 >= 0) {
             B[v0] = A[t0];
             v0++;
        }
        t0++;
   }while (t0 < a2);

  return v0;
}
```

b) Describe briefly, in English, what this function does.

This function inspects first a2 elements of array A and copies them into array B if they are nonnegative. The return value is the number of copies.

Problem #4: Compiling Recursive Functions [12 points]

Compile the following function into MIPS instructions and add comments to your code. The function **pow** takes two arguments (*n* and *m*, both 32-bit numbers) and returns n^m (i.e., *n* raised to the mth power). This function assumes that *m* is greater than or equal to one.

```
int pow(int n, int m) {
     if (m == 1)
          return n;
     return n * pow(n, m-1);
}
```

Argument registers $a0 and $a1 will correspond to *n* and *m*, and the return value should be placed in $v0 as usual. You will not be graded on the efficiency of your code, but you *must* follow all MIPS conventions. Comment your code.

```
pow: bne $a1, 1, rec    # if m == 1, return a0
     move $v0, $a0      # base case: set return value to n
     j $ra              # jump back to the caller

rec:
     addi $sp, $sp, -8  # adjust the stack to store $ra
     sw $ra, 0($sp)     # save $ra
     sw $a0, 4($sp)     # save $a0
     addi $a1, $a1, -1  # recursive case: m = m - 1
     jal pow            # call pow with n and m - 1
     lw $a0, 4($sp)     # restore $a0
     mult $v0, $v0, $a0 # multiply the result by n
     lw $ra, 0 ($sp)    # restore the return address of pow
     addi $sp, $sp, 8   # readjust the stack pointer $sp
     j $ra              # jump back to the caller
```

Notes:

- Since pow does not manipulate $a0, it is OK (but perhaps bad style) not to preserve its value in this program.
- Contents of $a1 need not be saved, because $a1 is not used by pow after the recursive call.

Problem #5: Short Questions, 4 points each [32 Points]

a) Convert to **Decimal** the following single-precision IEEE 754 floating point number:

1 10000111 11011000000000000000000

-472.0

b) Convert **$-128.875_{(10)}$** from decimal to **single precision** IEEE 754 floating-point representation. Express your result in **hexadecimal**.

0xC300E000 = 11000011 00000000 11100000 00000000

c) The hexadecimal number **0xC08F400000000000** represents a double precision IEEE floating-point number. What is the corresponding decimal value?

$-1000_{(10)}$

d) Perform the following floating-point addition as computed in a single-precision FP hardware adder with a total of 4 precision bits. Show all steps and give the final result in:

a. Normalized binary (4 bits of precision)

b. Normalized decimal (4 digits of precision)

$$-1.101_2 \times 2^{-2} + 1.011_2 \times 2^{-3}$$

$$\mathbf{-1.111_{(2)} \times 2^{-3} = -0.2344_{(10)} = -2.344_{(10)} \times 10^{-1}}$$

e) If the instruction **beq $s0, $s1, Exit** is located at address **0x004000BC**, and encoded as **0x10800007**, what is the byte address of the label **Exit**? Show your steps.

0x004000DC

Instruction **j Label** is stored at **0x004400A4**, and **Label** is at instruction address **0x0044008C**. Encode this instruction and express your answer in hexadecimal. The opcode for the **jump** instruction is (**000010**).

000010 [0000 0100 0100 0000 0000 1000 11] =
0000 1000 0001 0001 0000 0000 0010 0011 = 0x 08110023

f) Write **MIPS** instructions in the body of the C++ function **`flip_nb(x,n)`** below that takes a 16-bit unsigned integer *x* ($b_{15}b_{14}b_{13}\ldots b_1b_0$) and returns the same value with the "*n*"th bit (b_n) inverted (i.e., b_n flipped from 1 to 0 or from 0 to 1).

For example, calling **`flip_nb(0x04F2,7)`** returns **`0x0472`**.

```
unsigned flip_nb(unsigned x, unsigned n)
{
      li $t0, 1
      sll $t0, $t0, $a1
      xor $v0, $t0, $a0
      jr $ra
}
```

g) Consider the following sequence of MIPS instructions. Where does the **`jr $ra`** instruction jump?

```
      jal L1
L1:   addi  $ra, $ra, 8
```

```
L2:   jr $ra
```

Second instruction after L2

Problem #6: MIPS Compilation [12 points]

Compile the following C++ statements shown on the left column. Assume the following register-to-variable mappings: **i:$s0, j:$s1, A:$s2**. You can only use the following MIPS instructions: **sll, srl, add, addi, andi, sub, lw, sw, slt, beq, bne, j**. NO OTHER INSTRUCTIONS ARE ALLOWED. The variables **i**, **j**, and the array **A** are all integers.

C++	MIPS

`A[2*i+3*j] = A[7*i+5*j];`	`sll $t0, $s0, 1   //t0=2*i`
	`sll $t1, $s1, 1   //t1=2*j`
	`add $t1, $t1, $s1 //t1=3*j`
	`add $t1, $t0, $t1 //t1=2*i+3*j`
	`sll $t1, $t1, 2   //t1=4*(2*i+3*j)`
	`add $t1, $t1, $s2 //t1=t1+base`
	`lw $t0, (0)$t1 //t0=A[t1]`
	`sll $t0, $s0, 3   //t0=8*i`
	`sub $t0, $t0, $s0 //t0=7*i`
	`sll $t1, $s1, 2   //t1=4*j`
	`add $t1, $s1, $t1 //t1=5*j`
	`add $t1, $t0, $t1 //t1=7*i+5*j`
	`sll $t1, $t1, 2   //t1=4*(7*i+5*j)`
	`add $t1, $t1, $s2 //t1=t1+base`
	`sw $t0, (0)$t1 //A[2*i+3*j]=A[7*i+5*j]`

`if (i ≤ j)`
`    i = (i % 16) + 1;`
 	
`//Note: i%16 is i modulo 16`
	`slt $t0, $s1, $s0  //t0=(j<i)`
	`   bne $t0, $0, EXIT  //branch if t0=1`
	`   andi $s0, $s0, 0xF`
	`   addi $s0, $0, 1`
	`EXIT:`

`for(i=5; i<100 || j>0; i++)`
`    A[j] = i;`
	`addi $s0, $0, 5`
	`       addi $t0, $0, 100`
	`START: slt $t1, $s0, $t0`
	`       beq $t1, $0, EXIT`
	`       slt $t2, $0, $s1`
	`       beq $t2, $0, EXIT`
	`       sll $t3, $s1, 2`

Problem #7: Pseudo-Instructions [8 points]

Consider the SWAP pseudo-instruction that swaps the contents of two registers $s0 and $s1:

```
SWAP $s0, $s1      // swaps contents $s0 and $s1
```

a) You are told that you can implement the SWAP instruction without using any registers other than $s0 and $s1 in your MIPS code. Can you come up with such a sequence of instructions to implement SWAP?

```
xor $s0,$s0,$s1  // $s0 = $s0 ⊕ $s1
xor $s1,$s0,$s1  // $s1 = ($s0 ⊕ $s1) ⊕ $s1 = $s0
xor $s0,$s0,$s1  // $s0 = ($s0 ⊕ $s1) ⊕ $s1
                        = ($s0 ⊕ $s1) ⊕ $s0
                        = $s1
```

b) Use the above SWAP pseudo-instruction to implement a three-way swapping among three registers `$s0`, `$s1`, `$s2` as follows:

```
$s0 <- $s1, $s1 <- $s2, $s2 <- $s0
SWAP $s0, $s1  //$s0 = $s1, $s1 = $s0;
```

```
SWAP $s1, $s2  //$s2 = $s1 = $s0 (old), $s1 = $s2
```