1) [10 points]
(a) Add the following base-10 numbers using 6-bit 2s complement math: -3, -4. Show your work!

To get -3 in binary, start with 3 and negate (flip all bits and add one):

000011
111100 < bits flipped
111101 < added one
^ this is -3

Same to get -4 in binary:

000100
111011 < bits flipped
111100 < added one
^ this is -4 in binary

Now we add:

1111 < carries
111101
+ 111100
--------
111001 < sum

Check our work -- let's convert the sum to decimal. First we negate it to make it positive:

111001
000110 < bits flipped
000111 < added one
^ this is the negation of our sum

111 in binary is 7 in decimal
this makes sense, as -3 + -4 = -7
2) Assume that $2 = 2000$ and $3=12$. Assume that memory holds the values at the addresses shown on the left. “lw” = load word, and “sw” = store word.

(a) If the computer executes sw $3, 4($2), then what is the value of $3$ after this instruction?

12

(the store doesn’t change the register, it changes the memory)

(b) If, after the instruction in part (a), the computer executes lw $3, 0($2), what is the value of $3$ after this instruction?

130

(c) What single instruction could you use to write the value in $5$ into address 2008?

sw $5, 8($2)

or as a joke answer: sw $5, 1878($3)

(d) What single instruction could you use to read the word of memory at address 1996 and put the result in $8$?

lw $8, -4($2)
3) [10] The IEEE 754 floating point standard specifies that 32-bit floating point numbers have one sign bit, an 8-bit exponent (with a bias of 127), and a 23-bit significand (with an implicit “1”). Represent the number -11.75 in this format.

- Sign bit: 1 (negative)
- Fractional representation: -11 3/4
- Binary representation: -1011.11
- Normalized: -1.01111 * 2^3
- Mantissa with the first one removed: 01111
- Exponent with bias added: 3+127 = 130
- Biased exponent in binary: 1000010

\[1\ 1000010\ 011110000000000000000000\]
4) [10] The following questions are based on the following code snippet.

(a) What is *(array+7)*? Please give its datatype and its value.

Same as array[7]
Type: int
Value: 49

(b) On a MIPS machine, how big (how many bytes) is the variable array?

The variable array, like all pointers on a system with 32-bit words, is 32-bits long, which is 4 bytes long.

(c) On a MIPS machine, how big (how many bytes) is array[2]?

It's the size of an integer, which on MIPS, is 32-bits, or 4 bytes.

(c) What is the datatype of fun?

int**
(A pointer to a pointer to an int. Size is still 4 bytes, since it's a pointer)

```c
int* array = (int*) malloc(42*sizeof(int));
int** fun = &array;
for (int i=0; i<42; i++){
    array[i] = i*i;
}
free (array);
```
5) [25] Convert the following C code for the function foo() into MIPS code. Use appropriate MIPS conventions for procedure calls, including the passing of arguments and return values, as well as the saving/restoring of registers. Assume that there are 2 argument registers ($a0-$a1), 2 return value registers ($v0-$v1), 3 general-purpose callee-saved registers ($s0-$s2), and 3 general-purpose caller-saved registers ($t0-$t2). Assume $ra is callee-saved. The C code is obviously somewhat silly and unoptimized, but YOU MAY NOT OPTIMIZE IT -- you must simply translate it as is.

1: int foo (int num){
2:   int temp = 0; //temp MUST be held in $t0
3:   if (num <0) {
4:     temp = num + 2;
5:   } else{
6:     temp = num - 2;
7:   }
8:   int sumA = bar(temp); // sumA MUST be held in $s0
9:   int sumB = sumA + temp + num; // sumB MUST be held in $s1
10:  return (sumB + 2);
11:}

12: int bar (int arg){
<table>
<thead>
<tr>
<th>line(s) of C</th>
<th>instruction(s)</th>
<th>what code MUST do (if not obvious from C code)</th>
</tr>
</thead>
</table>
| 1           | `# need 20 bytes for s0,s1,t0,t1,ra 
# why t1 even though its not needed in the problem? 
# because i need to backup a0 before the call 
addiu $sp,$sp,-20 
sw $s0,0($sp) 
sw $s1,4($sp) 
sw $ra, 8($sp)`                                                                 | create stack frame large enough for callee-saved and callee-saved registers; save callee-saved registers (ONLY necessary ones) |
| 2           | `li $t0, $t0, 0  # alternately, i could do "move $t0,$0"`                                                                                                                                          |                                               |
| 3-7         | `beqz $a0, else  # invert the compare to get to the else 
 #then 
 addi $t0, $a0, 2 
 j end_if  # bypass the else 
 else: 
 addi $t0, $a0, -2 
 end_if:`                                                                 |                                               |
| 8           | `move $t1,$a0  # backup num 
move $a0, $t0 
sw $t0, 12($sp) 
sw $t1, 16($sp) 
jal bar`                                                                                                                                          | save caller-saved registers (ONLY necessary ones); call bar() with appropriate arguments |
| after line 8 | `lw $t0, 12($sp) 
 lw $t1, 16($sp) 
mov $s0, $v0`                                                                                                                                                      | restore caller-saved registers; get value returned from bar() and put it in appropriate place |
| 9           | `add $s1, $s0, $t0  # sumA+temp 
add $s1, $s1, $t1  # += num`                                                                                                                                         |                                               |
| 10          | `addi $v0, $s1, 2 
lw $s0,0($sp) 
lw $s1,4($sp) 
lw $ra, 8($sp) 
addiu $sp, $sp, 20 
jr $ra`                                                                                                                                            | pass return value back to whoever called foo(); restore callee-saved registers; destroy stack frame; return to caller |
1) [10 points] Write the truth table for the output of the following boolean expression that has three inputs (a, b, c): output = abc + \overline{ac} + b\overline{c}

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2) [10 points] Convert the following truth table into a boolean expression in product-of-sums format. Note that there are three inputs (a, b, c) and one output. Do NOT simplify or optimize in any way.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(\overline{a} & \overline{b} & \overline{c}) | (\overline{a} & b & c) | (a & \overline{b} & c) | (a & b & \overline{c}) | (a & b & c)
Question 1 [20 points]: Consider the circuit below. Assuming the two flip flop start with a value of zero, what will the state of the flip flops be for the clock cycles shown? The initial state is done for you.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>FF1</th>
<th>FF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Same exercise, but with a different starting condition:

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>FF1</th>
<th>FF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Question 2 [17 points]: Draw a finite state machine that will output a 1 if and only if a sequence of characters of the following form is received: exactly one ‘D’, zero or more ‘O’s, and exactly one ‘G’. (If you happen to know regular expression notation, this is the expression /DO*G/.) Examples of matching inputs include: “DG”, “DOG”, “DOOOOG”. Your machine can be of the Mealy or Moore variety. It doesn’t matter what your machine does after it outputs 1.