name: Johnny Exampleface

Duke students are bound by an academic integrity standard:

1. I will not lie, cheat, or steal in my academic endeavors, nor will I accept the actions of those who do.

2. I will conduct myself responsibly and honorably in all my activities as a Duke student.

Please sign your name below to acknowledge that you follow this standard:

Johnny Exampleface

--------------------------------------------------------------------------------
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 \texttt{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 \texttt{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?

\texttt{sw $5, 8($2)}

or as a joke answer: \texttt{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 $8$?

\texttt{lw $8, -4($2)}
3) 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
Binary representation, normalized: -1.01111 * 2^3
Mantissa with the first one removed: 01111
Exponent with bias added: 3+127 = 130
Biased exponent in binary: 10000010

1 10000010 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)

```
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 $0, $1, $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 caller-saved registers; save callee-saved registers (ONLY necessary ones) |
| 2 | li $t0, $t0, 0  # alternately, i could do "move $t0,$0" | |
| 3-7 | bgez $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 | |
| 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 |
6) [10] Explain the von Neumann model of computers.

- Implicit model of all modern ISAs
  - “von NOY-man” (German name)
  - Everything is in memory (and perhaps elsewhere)
    - instructions and data

- Key feature: **program counter (PC)**
  - PC is the memory address of the currently executing instruction
  - Next PC is PC + length_of_instruction unless instruction specifies otherwise

- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts