Assembly language

Problem 1

Consider this C struct definition:

```
struct foo {
   int *p;
   int a[3];
   struct foo *sf;
} baz;
```

Suppose that register $16 contains the address of baz.

For each of the following C statements, indicate which of the MIPS assembly language code fragments below (A-H) could be the result of compiling it.

```
codeA: lw    $8, 0($16)
       sw    $8, 4($16)

codeB: lw    $8, 0($16)
       lw    $9, 0($8)
       sw    $9, 4($16)

codeC: lw    $8, 4($16)
       sw    $8, 0($16)

codeD: sw    $16, 16($16)

codeE: lw    $17, 6($16)

codeF: lw    $17, 12($16)

codeG: lw    $8, 0($16)
       sw    $8, 16($16)

codeH: addi  $8, $16, 4
       sw    $8, 0($16)
```

```
____ number = baz.a[2];

____ baz.p = baz.a;

____ baz.a[0] = *baz.p;

____ baz.sf = &baz;
```

```
__F_ number = baz.a[2];
__H_ baz.p = baz.a;
__B_ baz.a[0] = *baz.p;
__D_ baz.sf = &baz;
```

Problem 2

Translate the following C procedure to MIPS assembly language. Assume that arguments are passed in registers.

```
int garply (int a, int *b) {
    int c;
    c = subt(a >> 6);
    *b = a + *b;
```

```
        if (a < 0) || c < 0)
           return c;
        else
           return c | a;
    }
```

```
garply:
   addi  $sp,$sp,-12
   sw    $a0,0($sp)
   sw    $a1,4($sp)
   sw    $ra,8($sp)
   sra   $a0,$a0,6
   jal   subt
   # $v0 now contains c
   lw    $t0,0($sp)      # get a
   lw    $t1,4($sp)      # get b
   lw    $t2,0($t1)      # get *b
   add   $t2,$t2,$t0
   sw    $t2,0($t1)      # update *b
   bltz  $t0,return
   bltz  $v0,return
   or    $v0,$v0,$t0
return:
   lw    $ra,8($sp)
   addi  $sp,$sp,12
   jr    $ra
```

Problem 3

Consider the following fragment of a C program.

```
int v[10], s;
int *p;
s = 17;
for (p = &v[3]; *p != 0; p++)
        s = s + *p;
```

Here is a buggy translation in MIPS assembly language, assuming s is in \$16 and p is in \$19.

```
          or    $16, $0, $0
          lw    $19, v+12
loop:
          bne   $8, finish
          add   $16,$19,$16
          addi  $19, 1
          j     loop
finish:
```

There are six errors, including one missing instruction, in this translation. Find and fix them.

```
# Changes to the buggy code are underlined.
   li    $16,17
   la    $19,v+12
```

```
loop:
   lw    $8,0($19)
   beq   $8,finish
   add   $16,$8,$16
   addi  $19,$19,4
   j     loop
finish:
```

Problem 4

Consider the following MIPS assembly language routine. (The numbers on the left are just line numbers to help in your answer.) foo takes two integer arguments. The caller of foo and its callee bar follow the MIPS procedure call conventions. Assume var1 has been declared in the .data section with the .word directive.

```
1 foo: addi $sp, $sp, -20
2      sw   $s0, 16($sp)
3      sw   $s1, 12($sp)
4      la   $t0, var1
5      lw   $t0, 0($t0)
6      add  $t1, $a1, $a0
7      addi $s0, $t1, 10
8      add  $s2, $s0, $t1
9      add  $a0, $0, $s2
10     jal  bar
11     add  $t2, $t1, $v0
12     add  $s1, $t2, $a1
13     add  $v0, $0, $s1
14     lw   $s1, 12($sp)
15     lw   $s0, 16($sp)
16     addi $sp, $sp, 20
17     jr   $ra
```

a. Describe four bugs that are present in the code.

b. For each of these bugs, explain in one sentence either (i) why it will definitely cause the program not to work or (ii) under what condition will the program work correctly, in spite of the bug.

It's not clear what this code is supposed to do.
Some guesses at the bugs:
1. $t0 is never used.
2. $ra isn't saved, and it gets trashed by the call to bar.
3. $a1 isn't saved, and it gets trashed by the call to bar.
4. Five words of stack space were allocated, but only two were used.

#2 and #3 are serious problems that may cause the function to return a different value from what was expected.

Problem 5

Compile the following C code into MIPS.

```
struct Node {
        int data;
```

```
        struct Node *next;
};
int sumList (struct Node *nptr) {
        if (nptr == NULL) return 0;
        else return (nptr->data + sumList (nptr->next));
}
```

Your code must contain meaningful comments and adhere to the MIPS calling convention and register usage conventions. You are allowed to use pseudoinstructions to make it more readable. It should be clean and well structured. It needs to be right, not optimal, but your answer cannot be longer than 20 instructions.

```
sumlist:
   addi  $sp,$sp,8
   sw    $a0,0($sp)
   sw    $ra,4($sp)
   beqz  $$a0,return0
   lw    $a0,4($a0)     # get nptr->next
   jal   sumlist
   lw    $a0,0($sp)     # retrieve nptr
   lw    $a0,0($a0)     # get nptr->data
   add   $v0,$a0,$v0    # add to result of recursive call
   j     return
return0:
   li    $v0,0
return:
   lw    $ra,4($sp)
   jr    $ra
```

Problem 6

Translate the C function printDownUp to MIPS assembly language, retaining its recursive structure, passing its argument in the appropriate register, and following the usual register conventions. Translate putchar into a putc pseudoinstruction whose register argument contains the character to print.

```
void printDownUp (char c) {
   if (c == 'a') {
      putchar (c);
   } else {
      putchar (c);
      printDownUp (c-1);
      putchar (c);
   }
}
```

```
printDownUp:
   addi$sp,-8
   sw $s0,0($sp)
   sw $ra,4($sp)

   move$s0,$a0
   li $t0,'a'
   beq$s0,$t0,rtn
```

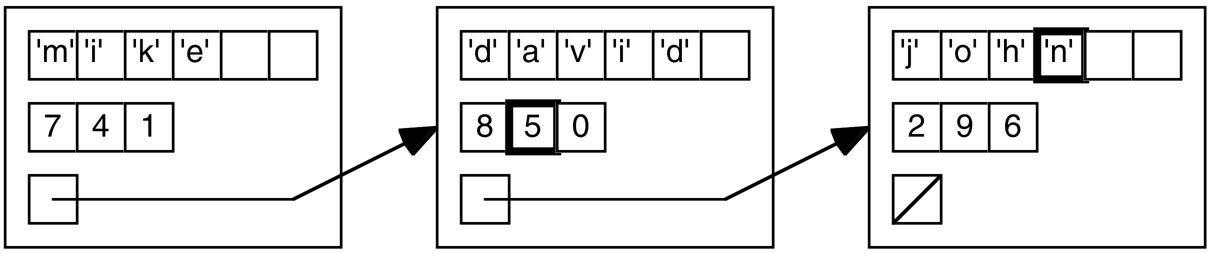
```
    putc$s0
    addi$a0,$s0,-1
    jalPrintDownUp
    putc$s0

rtn:
    lw $s0,0($sp)
    lw $ra,4($sp)
    addi$sp,8
    jr $ra
```

Problem 7

Consider a list with nodes defined in C as follows.

```
struct ListNode {
   char name[6];
   int code[3];
   struct ListNode* next;
};
```

The diagram below, not drawn to scale, gives an example of such a list.



Part a

Assume that register $a1 contains a pointer to the first node of the list. Write MIPS assembly language code that loads $s2 with the second integer in the second node in the list (with the list pictured above, this will load a 5 into $s2).

```
lw $t0,20($a1)
lw $s2,12($t0)
```

Part b

Again assume that register $a1 contains a pointer to the first node of the list. Write MIPS assembly language code that loads $s2 with the fourth character in the third node in the list (with the list pictured above, this will load 'n' into $s2).

```
lw $t0,20($a1)
lw $t0,20($t0)
lb $s2,3($t0)
```

Problem 8

Consider the following C functions that check if one string contains another as a substring. The terms "string 1" and "string 2" are used in the comments to mean the strings represented by s1 and s2 respectively.

```
int containsAsSubstring (char *s1, char *s2) {
   if (*s2 == '\0') {                      /* if string 2 has run out, */
      return 1;                         /* it's a substring of string 1. */
   } else if (*s1 == '\0') {               /* if string 1 has run out, */
      return 0;                /* string 2 isn't a substring of string 1. */
   } else if (startsWith (s1, s2)) {
      return 1;
   } else {
      return containsAsSubstring (s1+1, s2);
   }
}

int startsWith (char *s1, char *s2) {
   if (*s2 == '\0') {         /* any string starts with the empty string */
      return 1;
   } else if (*s1 == '\0') {               /* if string 1 has run out, */
      return 0;                      /* it doesn't start with string 2. */
   } else if (*s1 != *s2) {
      return 0;
   } else {
      return startsWith (s1+1, s2+1);
   }
}
```

Some examples of how containsAsSubstring behaves are listed below.

string 1	string 2	result of containsAsSubstring
"abcde"	"abc"	1
"xyabc"	"abc"	1
"axbc"	"ab"	0
"xy"	"abc"	0

Fill in the missing code in the MIPS assembly language implementation of containsAsSubstring below. (Don't worry about startsWith.) Your code should perform as described in the accompanying comments, and should follow conventions described in class and in lab and homework assignment 6 for passing arguments and managing registers and the system stack. You may assume that neither argument pointer is null.

```
containsAsSubstring:
    # save registers on the stack
addi$sp,-12# save registers on the stack
sw $s0,0($sp)
sw $s1,4($sp)
sw $ra,8($sp)

    # check base cases
   lb $t0,0($a0)# check base cases
   lb $t1,0($a1)

    beqz  $t1,returnTrue
    beqz  $t0,returnFalse

    move  $s0,$a0     # does string 1 start with string 2?
    move  $s1,$a1
    jal   startsWith
    bnez  $v0,returnTrue

    add   $a0,$s0,1   # no match; make recursive call
    move  $a1,$s1
    jal   containsAsSubstring
    j     return
returnTrue:
    # prepare to return 1
li $v0,1

    j     return
returnFalse:
    # prepare to return 0
move$v0,$0# prepare to return 0

return:
```

```
      # restore registers and return
lw $s0,0($sp)# restore registers and return
lw $s1,4($sp)
lw $ra,8($sp)
addi$sp,12
jr $ra
```

Problem 9

Here is the pwdHelper function from project 1. The declaration of struct entryNode appears on the last page of this exam.

```
void pwdHelper (struct entryNode * wd) {
   if (strcmp (wd->name, "/") != 0) {
      pwdHelper (wd->parent);
      printf ("/%s", wd->name);
   }
}
```

Write an assembly language version of pwdHelper that retains the recursive structure and follows all conventions for register use and stack management. You may use pseudoinstructions. Assume that functions named strcmp and printf are accessible from your function, and that they also follow all conventions for register use and stack management.

```
      .text
pwdHelper:
   addi  $sp,$sp,-8   # save $a0, $ra
   sw    $ra,0($sp)
   sw    $a0,4($sp)

   lw    $a0,0($a0)   # get wd->name
   la    $a1,slash
   jal   strcmp
   beq   $v0,$0,return

   lw    $a0,4($sp)   # get wd->parent
   lw    $a0,12($a0)
   jal   pwdHelper

   lw    $a1,4($sp)   # get wd->name
   lw    $a1,0($a1)
   la    $a0,fmtstring
   jal   printf

return:
   lw    $ra,0($sp)
   addi  $sp,$sp,8
   jr    $ra
```

```
      .data
   .data
slash:
   .asciiz "/"
fmtstring:
   .asciiz "/%s"
```

Problem 10

Part a

Given the following definition,

```
struct node {
   char name[12];
   int value;
};
```

what is sizeof (struct node)? _____

Assume that the sizes of chars and ints are the same as on the 271 Soda computers.

The answer is 16 bytes.

Part b

Translate the following code to assembly language in the space that follows. Your solution should adhere to conventions described in P&H. Comments in your code will help us understand your solution approach, and may earn you partial credit for an incorrect solution.

```
void exam1 (struct node **to) {
   exam2 (*to);
   (*(to-1))--;
}
```

```
# prolog: save information on stack if necessary

exam1:
   addi  $sp,-8
   sw    $a0,4($sp)# or save $s0 and put $a0 there
   sw    $ra,0($sp)
```

```
# call exam2
   lw    $a0,0($a0)# $a0 contains **to, so 0($a0) contains *to
   jal   exam2
```

```
# compute (*(to-1))--
   lw    $a0,4($sp)# not necessary if $s0 used
   lw    $t0,0($a0)# get *to
   lw    $t0,0($a0)# get to
   lw    $t1,4($t0)# get (*(to-1)) (to is ptr to 4-byte ptr)
   addi  $t1,$t1,-4# *(to-1) is a ptr to struct node
```

```
    sw    $t1,-16($t0)# decrement it (using their value from part a)




# epilog: restore necessary things and return
    lw    $ra,0($sp)
    addi  $sp,8
    jr    $ra
```

Problem 11

Suppose that the label names marks the beginning of an array of strings. In MIPS assembly language, this might appear as follows:

```
names:    .word     starting address of first string
          .word     starting address of second string
           ...
```

Give a MIPS assembly language program segment that loads the fourth character of the second string into register $t0. For example, if the array contains the strings "mike", "clancy", "dave", and "patterson", this character would be the 'n' in "clancy". Assume that there are at least two strings in the array and at least four characters in the second string.

A three-line solution is sufficient. You may use any registers you want.

```
la $t1,names
lw $t2,4($t1)# get the pointer to the 2nd string
lb $t0,3($t2)# get the 4th character
```

Also correct:

```
lw $t2,names+4# get the pointer to the 2nd string
lb $t0,3($t2)# get the 4th character
```

Problem 12

Complete the given framework to produce an assembly language function named reverse that implements the following (equivalent) Scheme and C functions:

Scheme

```
(define (reverse L soFar)
   (if (null? L) soFar
      (reverse (cdr L) (cons (car L) soFar) ) ) )
```

Equivalent C version

```
struct Thing {
   ... (as in project 1)
}
typedef struct thing *ThingPtr;
ThingPtr reverse (ThingPtr L, ThingPtr soFar) {
   if (L == NIL) {
      return soFar;
   } else {
      return reverse (L->th_cdr, cons (L->th_car, soFar));
   }
}
```

The code you supply should match the associated comments. Don't worry about memory allocation; the cons function will deal with that.

```
reverse:
      # Save relevant registers on stack.
      addi $sp,$sp,-8
      sw $a0,0($sp) # save L
      sw $ra,4($sp)

      # Check base case.
      bnez $a0,recursive
      move $v0,$a1
      j return

recursive:
      # Prepare for call to cons.
      lw $a0,0($a0) # retrieve (car L); $a1 already contains soFar
# OK to say 4($a0) since that's what project 1 would do

      jal cons
```

```
        # Prepare for recursive call to reverse.
        move $a1,$v0
        lw $a0,0($sp) # retrieve L
        lw $a0,4($a0) # retrieve (cdr L)
# OK to say 8($a0) since that's what project 1 would do




        jal reverse
return:
        # Pop stack, restore relevant registers, and return the desired result.
        lw $ra,4($sp)
        addi $sp,$sp,8
        jr $ra
```

Problem 13

Write a sequence of no more than six MIPS instructions that extracts bits 17:11 of register $s0 and inserts them into bits 8:2 of register $s1, leaving all the remaining bits of $s1 unchanged. You may use $t registers as temporaries.

```
sll   $t0,$s0,14     # turn bits 17:11 of $s0 into bits 8:2 of $t0
srl   $t0,$t0,24
sll   $t0,$t0,2
# everything else in $t0 should be 0
andi  $s1,$s1,0x1fc  # zero out bits 8:2 in $s1
ori   $s1,$s1,$t0
```

Problem 14

Consider a function isolateFloatFields that isolates components of a normalized positive floating point value in IEEE 32-bit format. Given such a value, isolateFloatFields should return

a. the exponent, and

b. the integer that results from omitting the binary point from the fraction represented by the significand.

For example, if the value 2.875 base 10 (which is $1.0111 \bullet 2^1$) is passed to isolateFloatFields, it should return the integer 1 for the exponent and the integer whose binary representation is 10111 followed by nineteen zeroes for the significand.

Complete the assignment statements in the C version of the function isolateFloatFields below.

The theBits function returns an unsigned integer whose bits are the same as those of its float argument. It's needed since bitwise operators in C may not be applied to float values.

```
void isolateFloatFields (float x, int *exponent, int *fractBits) {
   unsigned int bits = theBits (x);
   *exponent = ____________ ;
   *fractBits = ____________ ;
}
exponent = ((x & 0x7f800000) >> 23) - 127;
exponent = ((x >> 23) & 0xff) - 127;
sigBits = (x & 0x7fffff) | 0x800000;
```

Problem 15

Assume that $t0 contains an I-format MIPS instruction. In both parts of this problem, you are to write an assembly language segment that puts the *sign-extended immediate field* of the instruction into $t1. For example, if the instruction in $t0 were the machine language encoding of addi $a0,$a0,–17, you would store –17 in $t1. You may use pseudoinstructions and other temporary registers in your solution.

Part a

Give an assembly language program segment that copies the sign-extended immediate field of the machine code instruction in $t0 into $t1, that consists *only* of shift instructions.

```
sll $t1,$t0,16
sra $t1,$t1,16
```

Part b

Give an assembly language program segment that copies the sign-extended immediate field of the instruction in $t0 into $t1, that does not contain any shift instructions.

```
   andi $t1,$t0,0xFFFF
   andi $t2,$t0,0x8000
   beq  $t2,$0,next
   ori  $t1,$t0,0xFFFF0000
next:
```

Problem 16

In lab, you wrote a function that returned the contents of the various fields of a MIPS I-format instruction. In this problem, we consider a similar task for the Prune 100 computer. The Prune, like the MIPS, has 32-bit instructions. The Prune has only 16 registers. In an I-format Prune instruction, the meaning of the bits is as follows.

- The first 8 bits are the op code.
- The next 4 bits are the register to be modified by the instruction.
- The last 20 bits are the immediate operand, in 1's complement.

Thus the equivalent to the MIPS assembly language instruction addi $10,-2 might appear in hexadecimal as

```
94 af ff fd
```

if the op code for the addi instruction were 94 base 16.

On the next page, write a MIPS assembly language function splitIFormat that returns the contents of the register and immediate fields of a Prune 100 I-format instruction. If written in C, its prototype would be

```
void splitIFormat (int instr, int *register, int *immediate);
```

Follow the conventions described in class and in lab for passing arguments and managing registers and the system stack. Provide comments sufficient for the graders to understand your work.

```
# $a0 contains instr
# $a1 contains address of register
# $a2 contains address of immediate

splitIFormat:
   andi$t1,$a0,0xf00000# isolate register
   srl$t1,$t1,20
```

```
   sw $t1,0($a1)
   andi$t2,$a0,0xfffff# isolate immediate
   andi$t3,$t2,0x80000# see if negative
   beqz$t3,storeImm# no if branch
   ori$t2,0xfff00000# extend the negative sign
   addi$t2,$t2,1# convert from 1's to 2's comp

            # can also shift left 12,
            # then sra, which sign-extends
storeImm:
   sw $t2,0($a2)
   jr $ra
```

Machine language; architecture; the assembly process

Problem 17

What is the result of interpreting 0x82988000 as *a MIPS instruction*? Give your answer as an assembly language instruction, use numeric register names, and show intermediate steps.

```
   100000 10100 11000 1000 0000 0000 0000
op code = 20 => lb
rs = 20
rt = 24
sign-extended immediate field = 0xFFFF8000 = -2048

instruction = lb $24,-2048($20)
```

Problem 18

Which of the following is true of the ori instruction? Briefly explain your answer.

a. ori is always translated by the assembler into a single native MIPS instruction.

b. ori is always translated by the assembler into a sequence of two or more native MIPS instructions.

c. ori is sometimes translated by the assembler into a single native MIPS instruction and sometimes into a sequence of two or more native MIPS instructions.

Answer c. ori needs to be translated into a pair of instructions when the immediate field is bigger than 0xFFFF.

Problem 19

Why did the MIPS designers use PC-relative branch addressing (One sentence is enough!)
Most branches are to somewhere nearby, so we don't need all the bits that absolute addressing would require.

Problem 20

Assemble the following MIPS instructions into executable binary. Show the position of each field by drawing a box around the corresponding bit positions.

address	assembly language instruction	machine language instruction
0x400000	addi $a0, $a0, -4	2084FFFC
0x400004	L0: bne $s1, $t2, L1	16290003
0x400008	lw $s2, 128($sp)	8FB20080
0x40000c	j L0	08100001
0x400010	L1: subu $v0, $a0, $s0	00901023

Problem 21

Decode the following binary numbers as MIPS instructions and give the equivalent MIPS assembly language statements.

address	value
0x40	10001100101101110000000000100100
0x44	00000010111001001011000000100011
0x48	00011110110000001111111111110000

```
opcodes are 23, 0, and 7 respectively. Thus the first instruction is lw and the
third is bgtz. The second is an R-format instruction that we'll get to shortly.

Operands of the first instruction are
100011 00101 10111 0000000000100100, so we have
lw $23, 36($5)

Operands of the second instruction are
000000 10111 00100 10110 00000 100011, so we have
subu $22,$23,$4

Operands of the third instruction are
000111 10110 00000 1111111111110000, so we have
bgtz $22,-16 words from 0x4C
The latter address is 3C.
```

Problem 22

Part a

Translate the following program segment to native MIPS instructions. You may use either names or numbers for the registers.

```
      li    $t1,-5
loop: sub   $t1,$t1,3
      bgt   $t1,$a1,loop
```

Equivalent native MIPS segment:

```
first instruction: either
   lui$t1,-1# ok to use $1 instead
   ori$t1,$t1,-5# addi works here, I think
or
   addi$t1,$0,-5

second instruction:
   addi$t1,$t1,-3
or
   ori$1,$0,3
   sub$t1,$t1,$1

third instruction:
   slt$at,$a1,$t1
   bne$at,$0,loop
or
   sub$at,$t1,$a1
   bgtz$at,loop
```

Part b

Your answer to part a should include a branch instruction. Translate this branch instruction to machine language by filling in the boxes below with 0's and 1's.

```
bne$at,$0,loop000101 00001 00000 1111111111111100
bgtz$at,loop000111 00001 00000 1111111111111100
Displacement depends on their TAL code; displacement just given assumes that
two instructions are used for the subtraction.
```

Floating-point computation

Problem 23

Part a

Convert 6.25 to IEEE single precision. Show your work, and give your answer in binary.

```
6.25 = 110.01 base 2.
Biased exponent for 6.25 is 2+127 = 129.
0 10000001 (1) 100 1000 0000 0000 0000 0000
```

Part b

Show all the steps involved in computing the single-precision floating-point sum of 0x43D55555 and 0x41ADDEB7. Give the result in hexadecimal. (Don't convert anything to decimal.)

```
0x43D55555 = 0 10000111 (1) 101 0101 0101 0101 0101 0101
0x41ADDEB7 = 0 10000011 (1) 010 1101 1101 1110 1011 0111

Equalize exponents by increasing smaller exponent by 4 and shifting right by
4.
   0 10000111 (1) 101 0101 0101 0101 0101 0101
+  0 10000111 (0) 000 1010 1101 1101 1110 1011
=  0 10000111 (1) 110 0000 0011 0011 0100 0000
= 0x43E03340
```

Part c

What is the result of interpreting 0x82988000 as *a single precision IEEE floating-point value*? Give your answer as a sum of powers of 2, and show intermediate steps.

0x8298000 = 1 00000101 (1) 001 1000 1000 0000 0000 0000
Unbiased exponent = -122
Fraction is $1 + 2^{-3} + 2^{-4} + 2^{-8}$
Value = $-(2^{-122} + 2^{-125} + 2^{-126} + 2^{-130})$

Problem 24

Encode the value 17.2510 according to the single precision IEEE floating-point standard and show its representation in hexadecimal.
418A020C

Problem 25

Given below is a MIPS assembly language program segment that computes $(x+1.0)^2$ by adding x^2 to 2x, then adding 1 to that sum.

```
          .data
x:        .float
answer:   .float
```

```
one:      .float  1.0
          .text
__start:
          l.s     $f4,x
          l.s     $f6,one
          mul.s   $f8,$f4,$f4          # x^2
          add.s   $f8,$f8,$f4          # + 2*x
          add.s   $f8,$f8,$f4
          add.s   $f8,$f8,$f6          # + 1.0
          s.s     $f8,answer
```

Part a

Consider the case where x is 2.0^{12}. What is the difference between the value stored in answer and the actual value of $(2.0^{12} + 1.0)^2$? (If the answer is computed correctly, the difference will be 0.) Show your work.

The answer computed is 1 too small.

Part b

Does the sequence in which the terms are added affect the correctness of the answer? Briefly explain.

From lab, we know that 2.0^24 + 1 gives 2.0^24. The desired sum is 2.0^24 + 2.0^13 + 1, but there's no way to have a floating point value consisting of a sum of powers of two where the powers are more than 23 apart.

Problem 26

Consider the following C program segment.

```
int k, saved_k;
float x;
   ...
saved_k = k;
x = (float) k;
k = (int) x;
if (k == saved_k) {
   printf ("no change after conversion to float\n");
} else {
   printf ("change after conversion to float\n");
}
```

Recall that a cast converts the casted value to the given type. Thus if k contains the integer 3, the assignment

```
x = (float) k;
```

results in x containing the floating point value 3.0.

Assume for the following questions that an int and a float each use 4 bytes of memory, that a double uses 8 bytes of memory, and that a float and a double are stored using IEEE floating-point representation.

Part a

Find an int value k for which the above program segment produces the output

```
change after conversion to float
```

and give its hexadecimal (not decimal) representation.

Part b

Suppose that x in the above program segment was declared as double, with k being correspondingly cast to double. Would the output still be the same, using your answer to part a? Briefly explain.

```
No. Any 4-byte integer can be represented exactly in IEEE double-precision
format since the latter allows significands of 52 bits.
```

Part c

Return now to the original program segment, and give the *largest* (signed) hexadecimal integer value that k could contain and still produce the output

```
no change after conversion to float
```

Briefly explain your answer.

```
7FFFFF80. (The values for which no precision is lost are those for which the
difference between the positions of the first 1 bit in k and the last 1 bit in
k must be at most 23.)
```

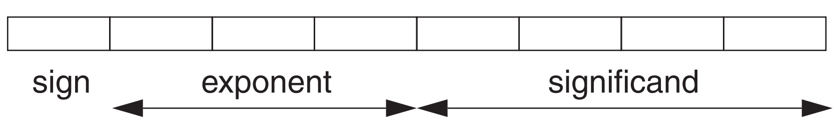
Part d

Give the 4-byte (single precision) IEEE floating-point representation (in hexadecimal) of your answer to part c. Show how you got your answer.

```
exponent = 30, so biased exponent is 30+127 = 157 = 100 1110 1 base 2
all the significand bits are 1
4EFFFFFF
```

Problem 27

Consider a representation (diagrammed below) for storing 8-bit floating point values that's exactly the same as the IEEE floating point representation except that three bits are allocated to the exponent and four to the significand.



Part a

Express in decimal the value represented by the byte 0xC1. Show your work for full credit.

```
sign is negative
biased exponent is 4
bias is 3
significand plus hidden bit is 1.0001
value is -2^(4-3) * 1.0001 = -2 * 1.0001 = -2.001 = -2.125 in decimal.
```

Part b

Let a be the value represented by the byte 0xC1. Determine a value b that, when added to a using the byte counterpart of IEEE floating point addition, produces a result that's not equal to the algebraic sum of a and b. Express this value in hexadecimal, and verify the mismatch of the computed and the algebraic sum.

Any value that produces 6 binary digits of precision (e.g. the value 2) works.

Linking

Problem 28

For each of the following utilities, specify what it takes as input and what it produces as output. Describe one key function it performs in this translation.

Compiler

Translates source code to assembler language or relocatable machine code. Handles parsing of arithmetic expressions. If producing relocatable machine code, produces a symbol table for the linker.

Assembler

Translates assembler language to relocatable machine code. Handles pseudoinstructions, and produces a symbol table for the linker.

Linker

Given several relocatable machine code files, lays them out in a memory image and fixes external references.

Loader

Given the output of the linker, loads the memory image into memory, initializes things like $sp and argv, and starts the program.

Problem 29

Given below are two assembly language program segments that are to be linked together with library code containing the getchar and malloc functions. On each line, specify *how many entries* in the relocation table would be produced by the assembler for the code on that line. (Put 0 for each line that doesn't generate a relocation table entry.)

Note that neither of these files is the result of compilation from C.

In the file main.s

In the file node.s

<pre>   .text
start:
    li $t0,0
    sw $t0,head($0)
loop:
    jal   getNode
    beqz $v0,gotAll
    lw $t0,head($0)
    sw $0,4($v0)
    sw $v0,head($0)
    j   loop
gotAll:
    ...
    .data
head:
    .word   0</pre>
	<pre> .text
	getNode:
	    addi $sp,$sp,-8
	    sw $ra,0($sp)
	    sw $s0,4($sp)
	    jal getchar
	    beqz $v0,return
	    ori $v0,0x20
	    move $s0,$v0
	    li $a0,8
	    jal malloc
	    sw $s0,0($v0)
	return:
	    lw $ra,0($sp)
	    lw $s0,4($sp)
	    addi $sp,$sp,8
	    jr $ra</pre>

```
2 entries each:
   all sw/lw involving head
1 entry each:
   all jal
   j loop
```

Problem 30

Consider the following three machine instructions, which appear in memory starting at the address 0x00400000.

address (in hex)	*contents (in hex)*
00400000	12080002
00400004	3C11FFFF
00400008	08100004

Part a

"Disassemble" the instructions; that is, give an assembly language program segment that would be translated into the given machine language. You may use numeric rather than symbolic register names. A list of op codes (Figure A.19 from P&H) appears at the end of this exam.

Handle branches and jumps specially; where you would normally have a label, provide instead a hexadecimal byte address. For example, you should list a jump to the first instruction as

```
j 0x00400000
```

and represent a branch to the first instruction, say bltz, similarly as

```
bltz $9,0x00400000
```

```
   beq$s0, $t0, match # 0x00400008
   lui$s1,0xFFFF
   j  match2 # 0x00400010
match:
   ... (one instruction)
match2:
```

Part b

For each of the instructions, indicate whether (a) it *must have contributed* an entry to the relocation table, (b) it *may have contributed* an entry to the relocation table, or (c) it *could not have contributed* an entry to the relocation table. Briefly explain your answers.

address (in hex)	*contents (in hex)*	*explanation of why this instruction must have, may have, or could not contribute relocation entry*
00400000	12080002	no entry since branch
00400004	3C11FFFF	no entry since lui entries (REFHI) are paired with REFLO entries

00400008	08100004	**certain entry since jump**

Problem 31

Consider the following assembly language program segment, which loads $t0 with the larger of $a1 and an integer labeled by value.

```
        lui$at, upper half of value
        lw    $t1, lower half of value($at)
        slt   $at, $t1, $a1
        beq   $at, $0, t1greater
        add   $t0, $0, $a1
        j     gotmax
t1greater:
        add   $t0, $0, $t1
gotmax:
     ...
```

Part a

The table below lists some of the statements in the program segment. Indicate which of the statements listed below will be represented by an entry in the relocation table.

statement	*will it contribute an entry to the relocation table? (yes or no)*
lui $at, upper half of value	yes
lw $t1, lower half of value($at)	yes
beq $at,$0,t1greater	no
j gotmax	yes

Part b

Given below is the part of the text segment of max.o that's the assembled version of the assembly language segment above. Assume that when the code is included in a program that is assembled into a file named max.o,the instruction labeled by t1greater is the 25th instruction in max.o's text segment and the word labeled by value is the third word in max.o's data segment. Fill in the missing hexadecimal digits. Show your work.

instruction	*corresponding hexadecimal value*
`lui $at,` upper half of `value`	`3C01` ________ **0000**
`lw $t1,` lower half of `value($at)`	`8C29` ________ **0008**
`slt $at,$t1,$a1`	`0125 082A`
`beq $at,$0,t1greater`	`1020` ________ **0002**
`add $t0,$0,$a1`	`0005 4020`
`j gotmax`	________ **0800 0019**
`t1greater:` `add $t0,$0,$t1`	`0009 4020`
`gotmax: ...`	

Problem 32

Consider a logic circuit that, given inputs x0, x1, and x2, produces a binary encoding in outputs q1 and q0 of how many of the xk are 1. A truth table relating q1 and q0 to the xk appears below.

x0	x1	x2	q1	q0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Using and, or, not, and xor, design Boolean equations to represent the circuit. Your equations should be simplified where possible; show your work.

```
q1 = ~x0 x1 x2 + x0 ~x1 x2 + x0 x1 ~x2 + x0 x1 x2
   = x1 x2 + x0 x2 + x0 x1
q0 = ~x0 ~x1 x2 + ~x0 x1 ~x2 + x0 ~x1 ~x2 + x0 x1 x2
   = x0 xor x1 xor x2
```