Problem 1 (10 points)

Parts a and b involved isolating the Rs field of an instruction. Here are solutions.

	isolating Rs

C	`// two-shift version`
	`return (inst << 6) >> 27;`
	`// shift-and-mask version`
	`return (inst >> 21) & 0x1F;`

assembly language	`# two-shift version`
	`sll  $v0,$a0,6`
	`srl  $v0,$v0,27`
	`jr   $ra`
	`# shift-and-mask version`
	`srl  $v0,$a0,21`
	`andi $v0,$v0,0x1F`

More of you provided the two-shift version, though (we think) the shift-and-mask version is somewhat simpler.

Part c involved a C program segment to convert a lower-case letter to upper-case. You were to translate the C code to assembly language. Here's a solution.

```
      li   $t1,'a'
      li   $t2,'z'
      blt  $t0,$t1,ok      # ch < 'a' if branch
      bgt  $t0,$t2,ok      # ch > 'z' if branch
      sub  $t0,$t0,$t1     # compute ch – 'a'
      addi $t0,$t0,'A'     # compute ch – 'a' + 'A'
ok:
```

Finally, part d involved translating a C switch to assembly language. Here's a solution.

```
      li  $t1,'y'
      bne $t0,$t1,checkn
      li  $v0,1
      j   switchend
checkn:
      li  $t1,'n'
      bne $t0,$t1,default
      li  $v0,0
      j   switchend
default:
      li  $v0,-1
switchend:
```

Problem 2

In this problem, you were to translate machine language instructions to assembly language. The instructions were 8D28FFF8 and 01022020.

We start by expressing each instruction as binary, in order to access the instruction's bit fields.

hexadecimal	binary
8D28FFF8	100011 01001 01000 1111111111111000
01022020	000000 01000 00010 00100 00000 100000

We observe from the op codes that 8D28FFF8 is lw and the other is an R-format instructions. The function fields of the latter indicate that each is an add.

In an assembly language lw, the Rt field is the *first* operand. Rt is the *second* operand in machine language. The offset for each is –8. (Note that the offset is in *bytes*, unlike the operand in a branch or jump, which is a *word* offset or address.)

The resulting instruction is

```
lw  $8,-8($9)
```

In the assembly language add instruction, the operands are Rd, Rs, and Rt. In machine language, they appear in the order Rs, Rt, Rd. Thus 01022020 translates to

```
add $4,$8,$2
```

Problem 3

This problem involved translation of truth table values to Boolean expressions. Answers are

$U_0 = N_2 + N_1 + N_0$

$U_4 = N_2\ N_1\ N_0$

$U_2 = !N_2\ N_1\ N_0 + N_2\ !N_1\ !N_0 + N_2\ !N_1\ N_0$ (sum of products)

$= N_2 + N_1\ N_0$ (simplified)

Each part was worth 1 point. You didn't need to simplify U_4 or U_0, and you didn't need to simplify U_2 all the way. Some of you provided a sum-of-products expression for U_0, which was maximally unsimplified!

Problem 4

In this problem, you were to provide a simplified Boolean expression representing a given circuit.

A good approach is to make a truth table:

	S 0	1
A		
0	0	0
1	0	1

Simplifying, we find that the output X = S A.

Problem 5

Here, you had to supply arguments to an assembly language version of snprintf. This problem was the same on both versions. (We announced at the exam that the format string should be changed to "%s%d %c", i.e. with no blank after the "%s".) Here is a solution.

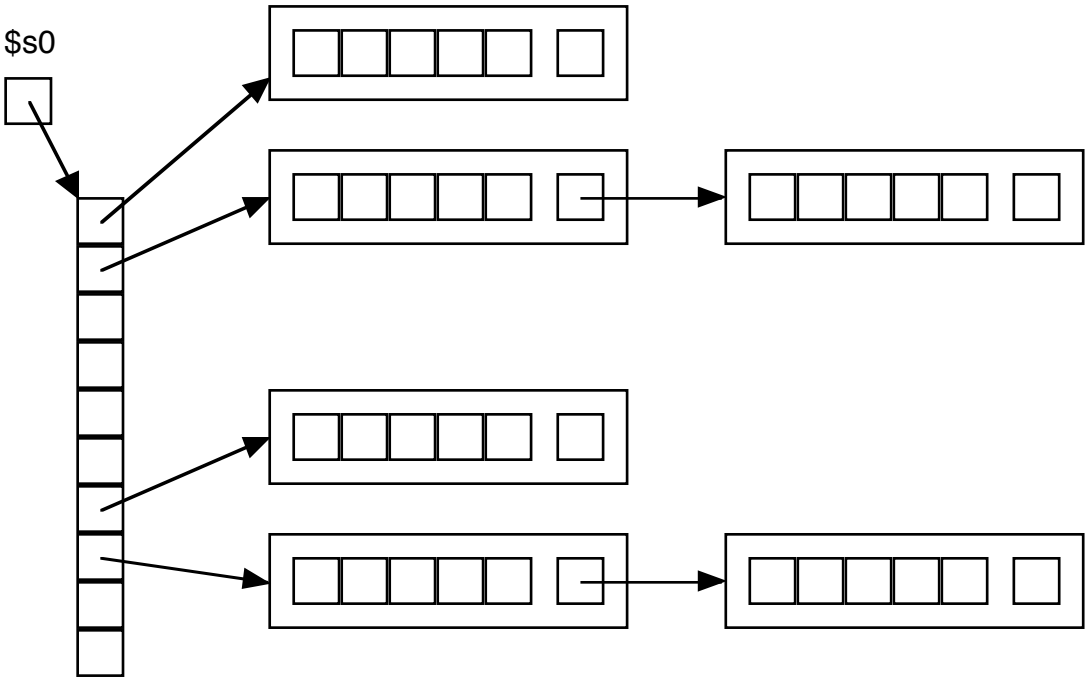
```
# argument 4 (in $a3): the string "N = "
la  $a3,chars+5

# argument 5 (on the stack): the integer 112
lb  $t0,more
sw  $t0,0($sp)

# argument 6 (on the stack): the character semicolon
lb  $t0,more+5
sw  $t0,4($sp)
```

Problem 6

In this problem, you were to give the C equivalent of assembly language accesses to a data structure. The data structure is pictured below.



$s0 corresponds to a struct node ** in C.

The assembly language segments and their C translation appears below.

assembly language	C
`addi $t0,$s0,4` `sw  $0,0($t0)`	`lists[1] = 0;` addi points $t0 at lists[1]; sw zeroes that element.
`lw  $t0,8($s0)` `sw  $t0,24($s0)`	`lists[6] = lists[2];` lw gets lists[2]; sw stores it into lists[6].

`lw  $t0,20($s0)`
`lw  $t0,20($t0)`
`sw  $0,20($t0)`
	`lists[5]->next->next = 0;`
	lw gets lists[5]; the next lw gets lists[5]->next;
	sw zeroes lists[5]->next->next.

Problem 7

Part a involved converting two values from decimal to their IEEE floating point representations. Here are solutions.

decimal	IEEE floating point
`4.5`	The sign is 0. The exponent is 2, so the biased exponent is 129. The fraction is (1).001, the result of shifting 100.1 two places to the right and then hiding the hidden bit. The result is 0 10000001 001000 … = 0x40900000.
`–0.625`	The sign is 1. The exponent is –1, so the biased exponent is 126. The fraction is (1).010, the result of shifting .101 left one place and then hiding the hidden bit. The result is 1 01111110 010 … = 0xBF200000.

Adding the two values in part b involved increasing the exponent and shifting the fraction of the smaller value to equalize exponents, adding the values, then renormalizing as shown below.

Compute $1.001 * 2^2 – 1.01 * 2^{-1}$.

Shift the fraction of the second value three places to equalize exponents:

$= 1.00100 * 2^2 – .00101 * 2^2 = .11111 * 2^2$

Renormalize:

$= 1.1111 * 2^1 = 3.875$

Problem 8

This problem involved exploring the consequences of adding a bit to the exponent in the IEEE floating point representation and simultaneously removing a bit from the fraction. In particular, you were to decide if the smallest `x` for which `x = x+1` would decrease, increase, or stay the same. This problem was the same on both versions.

The smallest `x` for which `x = x+1` would decrease from 2^{24} to 2^{23}. The problem arises when the exponents of the summands are equalized; the fraction for 1.0 must be shifted right as many places as the exponent is increased to match that of the bigger value. Shifting the hidden bit 24 places in IEEE format essentially zeroes it. If the number of fraction bits were reduced by 1, we only need a shift of 23 places to render 1.0 meaningless.

Problem 9

In this problem, you were to translate a C function (similar to the code in problem 1) to assembly language. Here's a solution.

```
answer:
   addi $sp,$sp,-4
   sw   $ra,0($sp)
   move $a1,$a0
   la   $a0,format
   jal  printf
   jal  getchar
   li   $t0,'y'
   bne  $t0,$v0,return0
   li   $v0,1
   j    return
return0:
   li   $v0,0
return:
   lw   $ra,0($sp)
   addi $sp,$sp,4
   jr   $ra

   .data
format:
   .asciiz "%s"
```

We told you at the exam not to use syscall. Some of you did it anyway. To avoid deductions, you had to use it correctly: "print string" requires a 4 in $v0 and the address of the first character of the string to print in $a0; "get character" requires a 12 in $v0, and *returns* the character in $a0 (contrary to MIPS register use conventions).

Problem 10

Part a was to identify which instructions in the given code would produce entries in the relocation table. The code appears below, with relevant instructions underlined and boldfaced.

Assembly language, .text section	Relocatable binary, .text section	
# Argument is the number of bytes # the caller wants to allocate. # Address of the requested storage # is returned, or 0 if request # can't be satisfied.	Address	Contents
`stackalloc:`		
`lw $v0,nextfree`	00	**3c010000**
	04	**8c220064**
`add $t0,$a0,$v0`	08	00824020
`la $t1,nextfree`	0c	**3c010000**
	10	**34290064**
`ble $t0,$t1,ok`	14	0128082a

	18	10200003
`add $v0,$0,$0`	1c	00001020
`j return`	20	**0800000b**
`ok:`		
`sw $t0,nextfree`	24	**3c010000**
	28	**ac280064**
`return:`		
`jr $ra`	2c	03e00008

Assembly language, .data section	Relocatable binary, .data section	
`stg:`	00	00000000
`.space 100`	…	...
	60	00000000
`nextfree:`		
`.word stg`	64	**00000000**

The jr does not produce a relocation entry, since the relevant absolute address will be in a register rather than in the instruction itself.

Note that some of the assembly language instructions—specifically, lw, la, and sw—expand to *two* machine language instructions, and both instructions in the pair will contribute relocation entries.

Part b was to do the relocation by adjusting absolute addresses in the machine language instructions. The following adjustments are necessary:

- Change the right half of each lui—at locations 00, 0c, and 24—to 1001.
- Change the j instruction at location 20 to 0x0810000b.
- Change the word at location 64 to 0x10010000.

The lw at location 04, the ori at location 0c, and the sw at location 28 would merely get changed to their existing values in the relocation process.