CS 61CL (Clancy/Culler) Solutions and grading standards for exam 2
Fall 2008

149 students took the exam. The average score was 42.3; the median was 44. Scores ranged from 7 to 60. There were 71 scores between 46 and 60, 60 between 31 and 45, 16 between 16 and 30, and 2 between 7 and 15. (Were you to receive scaled grades of 23 out of 30 on your two in-class exams and 46 out of 60 on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 16 may be projected to a B–.)

There were two versions of the test. (The version indicator appears at the bottom of the first page.)

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

Problem 0 (2 points)

You lost 1 point on this problem if you did any one of the following:

- you earned some credit on a problem and did not put your login name on the page,
- you did not adequately identify your lab section, or
- you failed to put the names of your neighbors on the exam.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we want to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to verify where students were sitting in the classroom while the exam was being administered.

Problem 1 (10 points)

Parts a and b, worth 2 points each, involved isolating the Rs field (in version A) or the Rt field (in version B) of an instruction. Here are solutions.

	isolating Rs	isolating Rt

C	`// two-shift version`
	`return (inst << 6) >> 27;`
	
	`// shift-and-mask version`
	`return (inst >> 21) & 0x1F;`
		`// two-shift version`
		`return (inst << 11) >> 27;`
		
		`// shift-and-mask version`
		`return (inst >> 16) & 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`
		`# two-shift version`
		`sll $v0,$a0,11`
		`srl $v0,$v0,27`
		`jr $ra`
		
		`# shift-and-mask version`
		`srl $v0,$a0,16`
		`andi $v0,$v0,0x1F`

More of you provided the two-shift version, though (we think) the shift-and-mask version is somewhat simpler. Each error lost 1 point. The most common bug was misunderstanding of the shift and masking operations.

Part c, worth 3 points, involved a C program segment to convert a lower-case letter to upper-case (version A) or vice versa (version B). You were to translate the C code to assembly language. Here's a solution to version A.

```
      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:
```

Correct logic was worth 2 points and the computation 1 point. This generally worked out to –1 per error.

Finally, part d involved translating a C switch to assembly language. It was also worth 3 points, and was the same on both versions. 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:
```

The 3 points were divided into 2 for the logic, 1 for the return value. Again, this generally worked out to –1 per error.

Problem 2 (4 points)

In this problem, you were to translate machine language instructions to assembly language. In version A, the instructions were 8D28FFF8 and 01022020; in version B, they were AD09FFF8 and 00881020.

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
AD09FFF8	101011 01000 01001 1111111111111000
00881020	000000 00100 01000 00010 00000 100000

We observe from the op codes that 8D28FFF8 is lw, AD09FFF8 is sw, and the others are R-format instructions. The function fields of the latter indicate that each is an add.

In an assembly language lw and a sw, 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 instructions are

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

In the assembly language add instructions, 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
```

and 00881020 translates to

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

Each instruction was worth 2 points. 1 point partial credit was given only for the following, in which all the bit fields were parsed correctly but operands were out of order.

hexadecimal	1 point partial credit answer
8D28FFF8	lw $9,-s($8)
01022020	add $8,$2,$4
AD09FFF8	sw $8,-8($9)
00881020	add $4,$8,$2

Most of you got this correct.

Problem 3 (4 points)

This problem involved translation of truth table values to Boolean expressions. It was the same in both versions. 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! Common errors mainly involved faulty simplification of U_2.

Problem 4 (4 points)

In this problem, you were to provide a simplified Boolean expression representing a given circuit. The circuits differed slightly in the two versions: in version A, the bottom multiplexor had A and 0 as the 0 and 1 inputs, while in version B those inputs were exchanged.

A good approach is to make a truth table:

version A	version B
S 0 1 A 0 0 0 1 0 1	S 0 1 A 0 0 0 1 1 0

Simplifying, we find that the output X = S A (version A) or X = !S A (version B),

You received 1 point out of 4 for getting started; you received 3 points out of 4 for an insufficiently simplified expression.

Problem 5 (6 points)

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)
```

Each argument was worth 2 points, for a possible total of 6.

Deductions were made as follows:

–2 confusing the type of an argument, for example, putting characters of a string into a register or treating a character like a string

–2 not saving an argument to the stack that should have been saved there (you lost this twice by not putting anything on the stack)

–1, each occurrence wrong operator or offset (e.g. lw for lb, sb for sw, lw for sw)

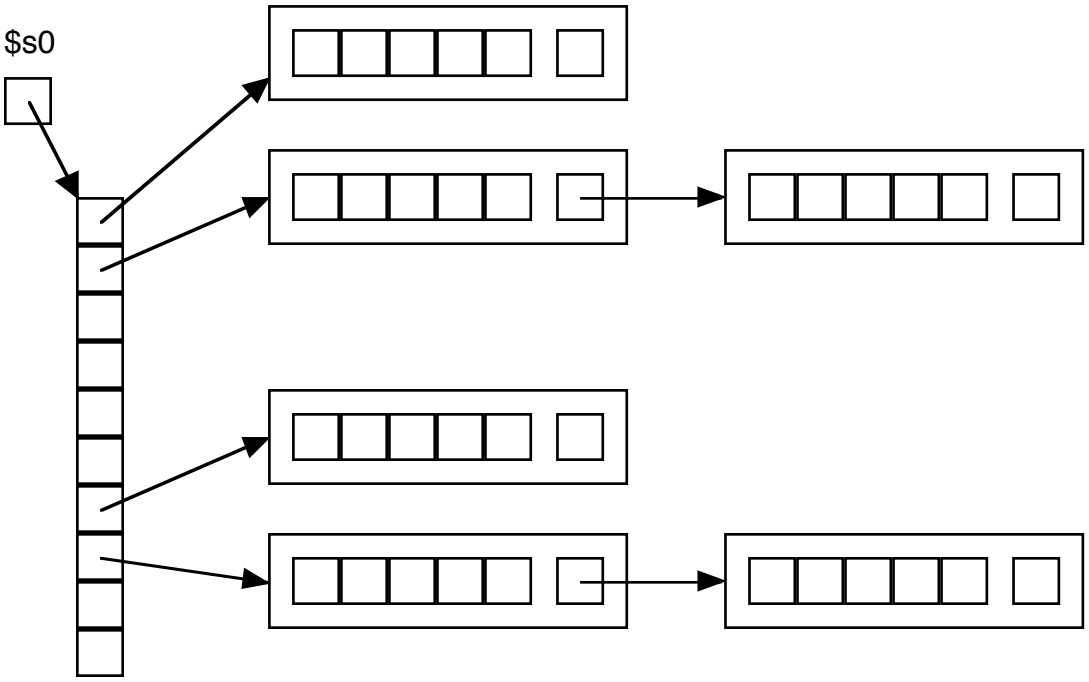
–1 wrong stack index (only deducted once)

–1 forgetting to use $a3

You were allowed to place arguments 5 and 6 on the stack in either order.

Problem 6 (5 points)

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 two sets of assembly language segments and their C translation for each of the two versions appears below.

Version A

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.

Version B

assembly language	C
`addi $t0,$s0,8` `sw $0,0($t0)`	`lists[2] = 0;`
`lw $t0,12($s0)` `sw $t0,16($s0)`	`lists[4] = lists[3];`

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

The first program segment was worth 1 point, and the second and third were worth 2 points each. Answers that displayed one of the following common misconceptions could have earned 4 out of the 5 points:

a. assuming that lists was an array of struct nodes (possibly including a struct node * at the start) rather than an array of pointers;

b. consistently being off by one level of indirection.

The first of those misconceptions might have resulted in version A answers of

```
lists[0].values[1] = 0;
lists[0].next = lists[0].values[2];
lists[0].next->next->next = 0;
```

Finally, a solution to the third program segment that assumed that lists[k]->next and lists[k]->next->next were *adjacent in memory* (i.e. occupied consecutive elements of the array) lost 2 points. This was a common error.

Problem 7 (6 points)

Part a, worth 4 points, involved converting two values from decimal to their IEEE floating point representations. Here are solutions.

Version A

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.

Version B

decimal	IEEE floating point
`4.25`	The sign is 0. The exponent is 2, so the biased exponent is 129. The fraction is (1).0001, the result of shifting 100.01 two places to the right and then hiding the hidden bit. The result is 0 10000001 0001000 … = 0x40880000.
`–0.75`	The sign is 1. The exponent is –1, so the biased exponent is 126. The fraction is (1).10, the result of shifting .11 left one place and then hiding the hidden bit. The result is 1 01111110 100 … = 0xBF400000.

You received 1 point for correctly computing the biased exponents, 1 point for correctly finding the fractions with the hidden bits, 1 point for the signs, and—if all these were correct—1 point for the correct hexadecimal value. An error involving an incorrect exponent or an incorrect fraction on *either* value lost you the corresponding point.

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.

Version A	Version B
Compute $1.001 * 2^2 – 1.01 * 2^{-1}$.	Compute $1.0001 * 2^2 – 1.1 * 2^{-1}$.
Shift the fraction of the second value three places to equalize exponents:	Shift the fraction of the second value three places to equalize exponents:
$= 1.00100 * 2^2 – .00101 * 2^2 = .11111 * 2^2$	$= 1.0001 * 2^2 – .0011 * 2^2 = .1110 * 2^2$
Renormalize:	Renormalize:
$= 1.1111 * 2^1 = 3.875$	$= 1.110 * 2^1 = 3.5$

You earned 1 point for shifting and 1 point for renormalizing, for a maximum of 2 in this part.

Problem 8 (4 points)

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.

Points were allocated as follows:

- 1 point for saying "decrease", and 1 more point for saying how much ("by half", or "by a factor of 2");
- 1 point for mentioning the need to equalize exponents, or for saying that the values were "far apart";
- 1 point for mentioning the need to shift the fraction, and that a fraction shifted 23 bits in the new system would disappear.

You may have lost one or both of the last two points by being insufficiently specific about how the modified bit fields related to the operation of addition.

If you got it backward (by saying x would increase) but had the right explanation, you received 3 points out of 4. You received no penalty for saying that x would decrease by a *power* of 2.

Problem 9 (7 points)

In this problem, the same in both versions, 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"
```

2 points were awarded for saving and restoring registers, 3 for procedure calls, and 2 for determining the return value. Common 1-point errors were using $s0 without saving it, using incorrect argument registers, failing to pass the format string to printf, and forgetting to pop the stack. Passing no arguments to printf at all lost 2 points.

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 (8 points)

This problem was the same on both versions. Part a, worth 4 points, 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	
	Address	Contents

`# 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.`		

`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.

You lost ½ point in this part for each missing entry and 1 for each wrong entry. A fractional score was truncated.

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.

You received 2 points in this part for correctly changing all the lui instructions, 1 point for correctly changing the j, and 1 point for correctly changing the word at location 64. (Omitting the latter was a common error.) 1 point was deducted for each incorrectly modified instruction.