CS 61CL (Clancy) **Solutions and grading standards for exam 1**
Spring 2009

169 students took the exam. The average score was 43.6; the median was 46. Scores ranged from 1 to 59. There were 89 scores between 45 and 60, 62 between 30 and 44, 15 between 15 and 29, and 3 between 1 and 14. (Were you to receive grades of 46 out of 60 on each 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 31 may be projected to a B–.)

There were four 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 (6 points)

You were to give the sizeof values for the given arguments. The info field in versions A and C was an 8-element array; in versions B and D, it was a 16-element array. Here are the answers:

expression	*value*	*explanation*
struct node (versions A and C)	8 + 4 = 12	both the array and the pointer take up space in the struct; a character takes up 1 byte
struct node (versions B and C)	16 + 4 = 20	
p	4	all pointers take up 4 bytes
struct node, using an int array instead of a char array (versions A and C)	8*4 + 4 = 36	each int is 4 bytes
struct node (versions B and D)	16*4 + 4 = 68	

Answers were 2 points each. If you consistently used the wrong size of a pointer but had everything else right, you received 2 on parts a and c and 0 on part b. Otherwise, partial credit was given only for small arithmetic errors.

Problem 2 (10 points)

Part a of this problem was to decode a machine language instruction. Solutions appear in the table below. All the shamt bit fields were 0.

version	machine instruction	op code	rs	rt	rd	funct	assembly language version
A	0x023B3821	0	0x11	0x1b	0x07	0x21	addu $7 $17 $27
B	0x02E36822	0	0x17	0x03	0x0d	0x22	sub $13 $23 $3
C	0x00AFC824	0	0x05	0x0f	0x19	0x24	and $25 $5 $15
D	0x03B34825	0	0x1d	0x13	0x09	0x25	or $9 $29 $19

This part was worth 2 points. You may have earned partial credit for a small error, for instance, getting the register fields out of order. Other errors included the following:

- reading the funct field as decimal rather than binary;
- misassociating the function field with the opcode of the same value (for example, decoding addu as lh).

Parts b and c of this problem were to write functions in C and assembly language to determine if the argument is a certain kind of instruction: sub in version A, and in version B, or in version C, and addu in version D. This involved isolating the op code and making sure it's 0, then checking the funct field for the relevant bit pattern. Here are some sample C solutions for isSub and their assembly language counterparts.

```
int isSub (unsigned int instr) {
   return !(instr & 0xFC000000)
     && (instr & 0x3F == 0x22);
}
```

```
isSub:
   srl      $t0,$a0,26
   bne      $t0,$0,returnfalse
   andi     $a0,$a0,0x3F
   addi     $t1,$0,0x22
   bne      $t1,$a0,returnfalse
returntrue:
   addi     $v0,$0,1
   jr       $ra
returnfalse:
   add      $v0,$0,$0
   jr       $ra
```

```
   return (instr & 0xFC00003F == 0x22);
```

```
isSub:
   lui      $t0,0xFC00
   ori      $t0,$t0,0x3F
   and      $t0,$t0,$a0
   addi     $t1,$0,0x22
   addi     $v0,$0,1
   bne      $t0,$t1,returnfalse
   jr       $ra
returnfalse:
   add      $v0,$0,$0
   jr       $ra
```

The C and the assembly versions were each worth 4 points. Deductions were as follows:

–1 for each small bug, e.g. using a mask that is off by one, shifting incorrectly, using & for && or vice versa, using a hex value without preceding it with 0x, switching the true and false cases, not using $a0 or $v0, omitting jr $ra, or forgetting the label naming the function in the assembly language version;

–2 for forgetting to check for a 0 op code, thinking that the funct value is in the op code, or for lui errors in loading an immediate operand;

–3 for assuming that everything but the funct field is 0.

You were allowed to require that the shamt field was 0. Some of you may have incorrectly lost points for doing so.

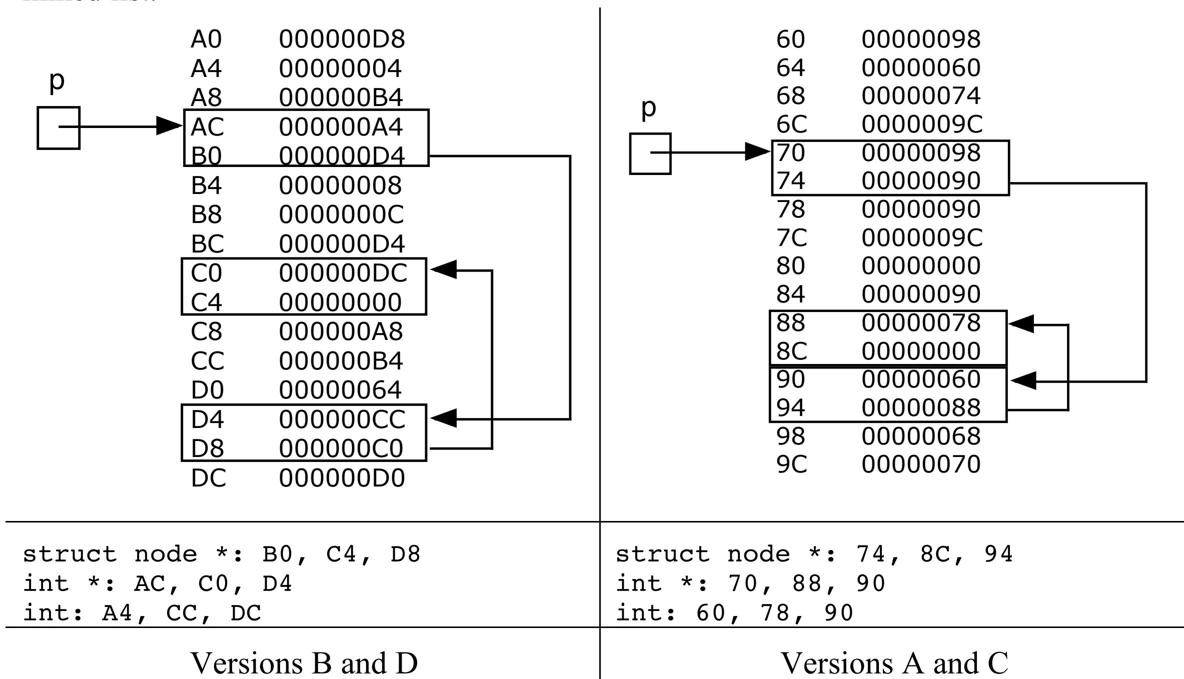
Mistakes in part b were usually repeated in part c. In this case, you lost points in *both* parts, reflecting the intent of the question to assess understanding of instruction format rather than C programming. In addition, you lost 1 point for each pseudoinstruction used in part c. Examples of pseudoinstructions are

bge, bgt, ble, blt	j with a register argument
move, li	andi with an argument longer than 16 bits

Some of these were listed on the MIPS reference card handed out with the exam. li was an especially popular pseudoinstruction.

Problem 3 (14 points)

This problem involved making sense of a layout of memory. There were two versions, diagrammed below. In both, the variable p points to the first node in a three-element linked list.



Part a was to use the memory diagram to identify the type and value of some C expressions. There were ten blanks; you received 1 point for each blank correctly filled in.

Versions A (same as version C, but with choices permuted)

expression	*type*	*value (in hex)*
`p->values`	`int *`	`98`
`p.values`	illegal	no possible value
`*(p->next)`	`struct node`	`{60, 88}`
`(p->values)+4`	`int *`	`A8`
`&(p->values)`	`int **`	`70`

Version B (same as version D, but with choices permuted)

expression	*type*	*value (in hex)*
`p->values`	`int *`	`A4`
`p.values`	illegal	no possible value
`*(p->next)`	`struct node`	`{CC, C0}`
`(p->values)+4`	`int *`	`B4`
`&(p->values)`	`int **`	`AC`

In part b, you had to identify the int values in memory. These were the locations pointed to by int * values. Only the first thing pointed to by each int * is guaranteed to be an int.

In versions A and C, the ints are 60 (pointed to by 90), 78 (pointed to by 88), and 98 (pointed to by 70). Version B/D's counterparts are A4 (pointed to by AC), CC (pointed to by D4), and DC (pointed to by C0).

This part was worth 4 points. You lost 1 point for each missing or incorrectly identified location, except that you earned some partial credit if you confused ints and int *'s.

Problem 4 (14 points)

Part a of this problem required translating an assembly function to C. Solutions are below.

versions A and C

```
update:
   lw    $t0,0($a0)
   lw    $t1,0($a1)
   ble   $t0,$t1,update0
update1:
   addi  $t1,$t1,1
   sw    $t1,0($a1)
   jr    $ra
update0:
   addi  $t0,$t0,1
   sw    $t0,0($a0)
   jr    $ra
```

```
void update (int * a, int * b) {
   if (*a > *b) {
      (*b)++;
   } else {
      (*a)++;
   }
}
```

```
void update (int a[], int b[]) {
   if (a[0] > b[0]) {
      b[0]++;
   } else {
      a[0]++;
   }
}
```

versions B and D

```
update:
   lw    $t0,0($a0)
   lw    $t1,0($a1)
   bgt   $t0,$t1,update0
update1:
   addi  $t1,$t1,-1
   sw    $t1,0($a1)
   jr    $ra
update0:
   addi  $t0,$t0,-1
   sw    $t0,0($a0)
   jr    $ra
```

```
void update (int * a, int * b) {
   if (*a <= *b) {
      (*b)--;
   } else {
      (*a)--;
   }
}
```

```
void update (int a[], int b[]) {
   if (a[0] <= b[0]) {
      b[0]--;
   } else {
      a[0]--;
   }
}
```

This part was worth 8 points, split 3 for the function header, 3 for pointer or array use within the function, 1 for a correct comparison, and 1 for a correct increment/decrement. If you didn't use arrays or pointers, you could earn at most 2 for this part. Note that you need parentheses around the incremented or decremented pointer; forgetting this lost you the increment/decrement point.

In part b, you were to explain the effect of inserting a jal at the start of the function, and then to correct the problem. Executing the jal would certainly overwrite $ra and probably the argument registers, so the fix is to save them prior to executing the jal and then to restore them on return. Here's a solution (the same in all versions).

```
update:
      addi  $sp,$sp,-12
      sw    $ra,0($sp)
      sw    $a0,4($sp)
      sw    $a1,8($sp)
      jal   printargs
      lw    $ra,0($sp)
      lw    $a0,4($sp)
      lw    $a1,8($sp)
      addi  $sp,$sp,12
        …   (the remainder of update goes here)
```

This part was worth 6 points, 3 for $ra and 3 for $a0+$a1. 2 points were deducted for using an $s register to store $ra without saving and restoring the $s register's previous value, or for using $s registers to store $a0 and $a1 without saving. If you did both, you lost 4 points.

Problem 5 (14 points)

This problem was the same in all versions. In part a, you were to provide a C implementation of addNcopies. This problem was the same in all versions. Here are iterative and recursive implementations.

```
struct node * addNcopies (int n, char * str, struct node * p) {
   int k;
   struct node * temp;
   for (k=0; k<n; k++) {
      temp  = (struct node *) malloc (sizeof (struct node));
      temp->info = (char *) malloc ((strlen(str)+1)*sizeof(char));
      strcpy (temp->info, str);
      temp->next = p;
      p = temp;
   }
   return p;
}

struct node * addNcopies (int n, char * str, struct node * p) {
   int k;
   struct node * temp;
   if (n == 0) {
      return p;
   }
   temp  = (struct node *) malloc (sizeof (struct node));
   temp->info = (char *) malloc ((strlen(str)+1)*sizeof(char));
   strcpy (temp->info, str);
   temp->next = p;
   return addNcopies (n-1, str, temp);
}
```

The malloc calls were an important part of this exercise. Maximum score was 10 points, allocated 5 for the inner malloc (temp->info in the solutions above), 3 for the outer malloc (temp in the solutions) and the linkage of the copies into the list, and 2 for essentially everything else. Saying only temp->info = s; without calling malloc for the string lost you all 5 of the inner malloc points. Copying characters into temp->info without doing malloc lost you 3 points. Both errors were quite common.

1 point was deducted for each of the following:

- orphaned storage, usually due to an extra malloc (you lost 2 points for O(n) extra mallocs);
- a pointer or parameter error;
- use of sizeof(s) or strlen(s) rather than strlen(s)+1;
- a missing return; or
- an easy-to-fix off-by-one error (harder-to-fix off-by-one errors lost 2 points).

In general, approaches that attached each node to the *front* of the list did better than approaches that added each node to the *end* of a list of new nodes and then hooked the end up to the argument list prior to return.

You were allowed to omit the cast of malloc's result, and to assume sizeof(char) == 1. We did not penalize you for copying the list argument as well as the new nodes.

In part b, you were to provide an assembly language segment that called addNcopies, by translating the statement

```
q = addNcopies (2, str, p);
```

to MIPS assembly language. Here's a solution.

```
li     $a0,2
la     $a1,str
lw     $a2,p
jal    addNcopies
sw     $v0,q
```

This part was worth 4 points. You lost 1 point for each wrong load. Saying la $a2,p was an especially common error. You also lost 1 point for using a wrong register (say, $a1-$a3, or $a0, $a1, and $a3), saving $a0-$a2 or $v0 prior to the call, or for using incorrect assembly language syntax (e.g. with 0(p)).

We incorrectly deducted a point for saving $ra. This point will be restored. Moreover, we concluded after deducting a point for not updating q that the wording of the problem may have led you to believe that you didn't need to worry about it. This point will also be restored.