CS 61C   Fall 2003   Midterm 1 solutions

Most of the short-answer questions were worth 1 point or 1/2 point, with no possibility of partial credit. Scoring information is given below only for the problems where it's not obvious from the above. We carried half points to the front of the exam, but after adding the points we truncated the result to an integer.

1a. Negation.

23 = 16+7 = 16+4+2+1 = 00010111 in 8-bit binary.

The sign-magnitude representation of -23 just turns on the leftmost bit, so 00010111 -> 10010111.

The ones-complement representation reverses all the bits, so 00010111 -> 11101000.

The twos-complement representation is the ones-complement representation plus 1, so 00010111 -> 11101001.

So the answers in order are B, C, A, D.

1b. twos complement range

With N bits you can represent 2^N values, but when you are representing signed integers about half those values are negative, so the largest magnitude is around 2^(N-1). For N=8, 2^N=256 and 2^(N-1)=128, so the answer has to be close to -128.

More precisely, in N-bit twos complement, the most negative number is -(2^(N-1)), which is represented as 100...00. (The most positive number is (2^(N-1))-1, which is 011...11.

For eight bits, the range is from -(2^7) to (2^7)-1, or -128 to 127. So the exact answer is indeed -128.

1c. unsigned range

In N-bit unsigned representation, the smallest number is 0, and the largest number is (2^N)-1, represented as 111...11.

For eight bits, the range is 0 to 255, so the answer is 255.

1d. hex addition

0xFA25 + 0xB705. The easiest way to do this is the same way you'd do addition of decimal integers: add from right to left, carrying when the result is 16 or more.

5+5 = 10 decimal = A hex, no carry.
0+2 = 2, no carry.
A+7 = 10 decimal + 7 = 17 = 16+1 = 1, carry 1.
F+B+1 = 15 + 11 + 1 decimal = 27 = 16+11 = B, carry 1.

So the result is 0x1B12A. But we only have 16 bits, which is 4 hex digits, so the leftmost digit is lost and the result is 0xB12A.

If you didn't know the addition table for hex, another way to solve this problem is to convert to binary and add the binary values:

```
carry          1 1111 11        1 1
first number     1111 1010 0010 0101
second number    1011 0111 0000 0101
               ---------------------
               1 1011 0001 0010 1010
```

Dropping the leftmost bit and converting back to hex gives 0xB12A.

If we'd allowed calculators, you could have converted the numbers to decimal and added them. Since they're signed numbers and both have the leftmost bit on, both are negative, so we should take their twos complements to get their absolute value.

We get the ones complement by subtracting each hex digit from 15 decimal, then we add 1 (to the entire number, not to each digit) to get the twos complement:

```
Given number    Ones complement Twos complement

FA25            05DA            05DB = 5*16^2 + 13*16 + 11 = 1499
B705            48FA            48FB = 4*16^3 + 8*16^2 + 15*16 + 11 = 18683
```

So our problem is (-1499)+(-18683) = -20182 = -4ED6 hex

0x4ED6's ones complement is 0xB129; its twos complement is 0xB12A.

1e. Overflow?

No, there is no overflow. In *unsigned* addition, a carry out from the leftmost bit is always an overflow, but in *twos complement* addition, overflow occurs when the carry out from the leftmost bit is unequal to the carry into the leftmost bit. In this case, both carries are 1. Another way to see this is that the range of representable numbers in 16-bit twos complement is -2^15 to (2^15)-1, or -32768 to 32767. So -20182, which is the correct answer, is representable, so there will be no overflow.

2a1. static char str[] = "thing";

This allocates a six-byte character array (including a byte for the null at the end) in global data space. Every call to set() refers to this same array. So each of the three calls changes one character: thing -> thong -> whong -> wrong. So the result that's printed is "wrong".

2a2. char str[] = "thing";

This allocates a new six-byte array on the stack for each call, then returns the address of that stack array, but the stack frame containing it is deallocated when set() returns. So what will be printed is whatever the call to printf() puts at that address on the stack! The result is therefore undefined, or a runtime error.

2a3. char *str = malloc(6); strcpy(str, "thing");

This heap-allocates a new six-byte array for each call. Each array has the initial value "thing" and then one character is changed. So the first two calls have essentially no effect; the third call changes thing -> tring, and "tring" is printed.

2b. What's legal C?

```
char a[14]; char c; char *p1; char **p2;

p1 = a+5;       The "a" without a subscript means &a[0], a constant of type
                pointer-to-char.  Adding an integer to a pointer is legal, and
                returns a pointer-to-char, which matches the type of p1, so
                the assignment is LEGAL.

&p1 = &a[0];    Any expression starting with & is a constant, not a variable,
                so this is an attempt to assign a value to a constant, like
                saying 3 = 4; so it's ILLEGAL.

p2 = a;         Again, "a" means &a[0], a constant of type pointer-to-char.
                But p2 is of type pointer-to-pointer-to-char, so this is a
                type mismatch and is ILLEGAL.  It would be legal, although
                weird, with an explicit cast:
                        p2 = (char **)a;

*(a+10) = 't';  The "a+10" is a valid expression of type pointer-to-char, like
                a+5 in the first statement.  So *(a+10) is a variable of type
                char, and we are assigning a char value to it, so this is
                LEGAL.

*p2 = &c;       p2 is of type pointer-to-pointer-to-char, so *p2 is a variable
                of type pointer-to-char.  &c is a constant of type
                pointer-to-char.  These match, so this assignment is LEGAL.
```

[Scoring: You started with 2 points, and lost 1/2 point for each error -- each statement that either should be crossed out but isn't, or shouldn't be but is -- with a floor of zero points.]

2c. Why stack frames?

Many people wrongly said "both" to this question.

The advantage of the stack over the heap is that allocation and deallocation are just a single instruction each:

```
        addi $sp, $sp, -framesize       # allocate
        addi $sp, $sp, framesize        # deallocate
```

By contrast, heap allocation requires searching the free list to find a large enough free block, removing that block (or part of it) from the free list, and updating the block header.

Access to local variables requires that we keep a pointer to the frame in a register, but once we've allocated the space, it doesn't matter whether that pointer points into the stack or into the heap; in either case we can use it in load and store instructions.

So the answer is #1.

3a. Branch MAL->machine language

During the exam, several people asked "what is the address of LOOP?" But branch instructions don't contain the target address; they contain the offset from the current PC (which points just after the branch) to the target address. In this case, you don't need to know where LOOP is; you can see that LOOP is the instruction before the branch, and that's all you need. Perhaps the reason for the confusion is that the *assembly language* branch instruction gives the target address (in the form of a label), while the *machine language* instruction gives the offset. The assembler computes the offset and uses it in the machine instruction that it generates.

BNE has opcode 5, which is 000101 in six bits.

$t0 = $8 = 01000 in five bits.

$zero = $0 = 00000.

LOOP is the instruction before the branch. The offset would be zero to branch to the instruction after the branch, -1 to branch to the branch instruction itself, and -2 to branch to the previous instruction. -2 is 1111111111111110 as 16 bits.

```
        000101 01000 00000 1111111111111110
```

Regrouping in four-bit chunks gives

```
        0001 0101 0000 0000 1111 1111 1111 1110
```

which is 0x1500fffe.

A popular wrong answer was 0x1500fff8, which would be correct if branch offsets were measured in bytes, like load/store offsets, rather than in words as they actually are.

3b. Why stack pointer in register?

Note that this is not the same as question 2c, which was about stack vs. heap allocation.

The main reason to keep pointers in registers is that all MIPS memory references use I-format (register+offset) addressing. So in order to get to things on the stack, we need a register that contains an address near the thing we want. The most straightforward way to do this is to keep the address of the current stack frame in a register. This is answer 2.

Answer 1 is wrong because (unlike some other architectures) there is nothing specifically about stacks in the MIPS architecture. (Other architectures, for example, have instructions named PUSH that allocate one stack word and add an item at the new stack address, and POP for the reverse.)

Answer 3 is nonsense; stack frames point to variables, not to procedures (unless a local variable happens to be of type pointer-to-procedure). And 4 is clearly wrong.

3c. MAL pseudo-instruction to real MIPS

The reason why the given ADDI instruction is not an actual machine instruction is that its operand doesn't fit in 16 bits. So we have to get it into a register, namely $at, the one reserved for use by the assembler.

```
        lui   $at, 0x7f
        ori   $at, $at, 0xf333
        add   $s0, $s1, $at
```

It is incorrect to use ADDI or ADDIU instead of ORI in the second instruction. These instructions both sign-extend their immediate operand, so you would be adding 0xffff f333, not 0x0000f333, to the result of the LUI. The sum would be 0x7ef333, not 0x7ff333. Since this is the main point of the question, this error got no credit.

We gave 1/2 point credit for a translation that used a register other than $at (or the equivalent $1) to hold the temporary result. The assembler is not allowed to use other registers, such as $t0, for this purpose, because your program might need the value in $t0. (But it's okay, in this case, to use $s0 instead of $at, since $s0 is the register whose value we're trying to change here. We gave full credit for $s0.)

3d. Structs and unions.

The key point to notice here is that fields i and d share memory. Each of the unions (x and y) are therefore 8 bytes long (the larger of sizeof(int) and sizeof(double)). sizeof(struct point) is therefore 16.

```
        la   $8, p              # $8 points to p[0]
        addi $9, $8, 160        # $9 points to p[10] ($8 + 10*16)
L1:     bge  $8, $9, L2
        sw   $0, 0($8)          # store 0 ($0) into x.i (offset 0)
        sw   $0, 8($8)          # store 0 into y.i (offset 8)
        addi $8, $8, 16         # add sizeof(struct point) to pointer
        b    L1
L2:
```

Note: Given that x and y are unions of different-sized alternatives, it's not obvious whether &x.i is the same as &x.d or whether it comes 4 bytes later. But the code we gave you settles that question, because we used an offset of 0 for x.i in the first SW instruction. This is the correct answer, according to K&R page 213: "A union may be thought of as a structure all of whose members begin at offset 0..."

There was some confusion about the second instruction, which produces a pointer to a nonexistent array element. The array has 10 elements, namely p[0] through p[9]. So why do we make a pointer past the end of the array? Because we aren't going to dereference this pointer; we use it only for the test for the end of the loop! When $8 reaches the value in $9, we've gone past the end of the array, so we stop looping.

Scoring: We counted the two uses of $0 as a single answer. Thus there are four answers here: 160, $0, 8, and 16. Each of these was worth 1/2 point.

4a. Floating point representation

Many people had trouble with this question, partly because you didn't read, or didn't believe, the part about "the same as IEEE." So, during the exam, we got questions like "is the exponent biased?" and "does this include denorms?" We answered by saying "the same as IEEE" but of course that implies "yes" to both questions.

What is the exponent bias? In IEEE single precision, with an 8-bit exponent field, the bias is 127, which is (2^7)-1. For our format, with a 3-bit exponent field, the bias will be (2^2)-1, which is 3. An all-zero exponent field is used for zero and denorms; an all-one exponent field is used for infinity and NaN, so the range representing normalized numbers is 001-110, which after bias conversion means -2 to 3.

There are four significand bits, plus (except for denorms) an implicit one before the binary point.

That means the largest representable number is

```
        1.1111 * 2^3 = 1111.1 = 15 1/2 = 15.5 decimal.
```

The smallest representable positive number is a denorm, meaning that
there is no implicit 1 before the significand. Figuring the exponent is
tricky; even though the exponent field is all zeros, which in bias-3 would
be expected to represent -3, denorms actually use an exponent value one
more than that (-2). This is so that the largest denorm, 0.1111 * 2^(-2),
is just below the smallest normalized, 1.0000 * 2^(-2). (Remember that
the whole point of denorms is to avoid a big gap among the smallest
magnitudes.) So it's

```
        0.0001 * 2^(-2) = 2^(-4) * 2^(-2) = 2^(-6) = 1/64
```

or 0.015625 decimal. (We accepted the 1/64 form, as the question says.)

With 8 bits there are 2^8 = 256 possible values. But 1/8 of those (32 of
them) have exponent 4, and are therefore infinite or NaN, leaving 224. And
two of those, +0 and -0, are equal, so there are 223 distinct numbers exactly
representable in this notation. We also accepted 224, because several people
asked during the exam whether to count +0 and -0 as different. But strictly
speaking the answer has to be 223, because the question asks "how many
numbers," not "how many number representations." And +0 is definitely the
same number as -0!

4b. The significand of a float (not including denorms) always has a value
between 1 and 2. (More precisely, 1 <= sig < 2.) This "fills up" the space
between consecutive powers of 2, as represented by the exponent field. In
other words, the number of exactly representable numbers between any two
consecutive powers of two is a constant. That is, the number of representable
numbers between 1 and 2 is equal to the number of numbers between 2 and 4,
which is equal to the number of numbers between 4 and 8, which is equal to the
number between 8 and 16, etc. The numbers get "thinner" as the magnitude gets
larger.

If the number of representable numbers between 1 and 2 is equal to the number
of representable numbers between 2 and 4, it must be *larger* than the number
of representable numbers between 2 and 3. So the answer is b1. (The
"inclusive" doesn't really affect the outcome; it's just in the question to
make the question unambiguous. Adding one or two to the number of numbers in
a range of 2^23 is insignificant compared to taking only half as many
numbers!)

5a. Clock vs. instruction count

```
microseconds     cycles      instructions   microseconds
------------ * ----------- * ------------ = ------------
   cycle       instruction      program       program
```

500 MHz = 500,000,000 cycles/second

```
1 / 500,000,000 = 2 / 1,000,000,000 seconds/cycle
                = .002 / 1,000,000 = 0.002 microseconds/cycle
```

So 0.002 * 3 * X = 15

X = 15 / (0.002 * 3) = 15 / 0.006 = 15,000 / 6 = 2500 instructions.

Another way to do it, avoiding having to work out the cycle time from the
clock rate, is to convert the fundamental equation above to this form:

```
  cycles      instructions   microseconds     cycles
----------- * ------------ = ------------ * -----------
```

```
instruction      program       program      microsecond
```

That gives 3 * X = 15 * 500, because MHz means cycles/microsecond.

Some people asked about the "micro" prefix. You should know these!

pico	trillionths	p
nano	billionths	n
micro	millionths	mu
milli	thousandths	m
kilo	thousands	k
mega	millions	M
giga	billions	G
tera	trillions	T

So MHz = Megahertz = millions of Hertz = millions of cycles/second
       = cycles/microsecond

Generally, capital letters abbreviate more-than-one multipliers, and lower case letters abbreviate less-than-one. The exceptions are k for kilo and the Greek letter mu for micro.

5b. The weighted average is
        (.3 * 2) + (.6 * 1) + (.1 * 10) = .6 + .6 + 1 = 2.2

6. Memory allocation.

If all blocks are the same size, we should maintain a free list that contains only blocks of that exact size. So we're never in the situation in which we allocate less than an entire free block, so there's no fragmentation. Similarly, there's no need to coalesce small free blocks to make a big one, since the size we want is exactly the size we have.

On the other hand, we do still need to maintain a list of free blocks, because over time they'll be scattered among other (same size) blocks that are in use. And it's still a good idea to keep the freeing of memory out of the hands of human beings by using a garbage collection system. (Think about a Lisp system, in which all pairs are the same size, and they're garbage collected!)

So the answers are No, Yes, No, Yes.

7. MIPS mergesort.

Because this procedure calls other procedures (including itself), we have to save $ra, and we have to save our one argument $a0. We also have to save the result from the first recursive call while we're working on the second recursive call. That's three things to save, so we need three words of stack frame. (Alternatively, we can save $s0 and $s1, and use those for the list argument and for the intermediate result.)

```
mergesort:
        addi $sp, -12           # prologue:
        sw   $ra, 0($sp)        # we accept either order of saving
        sw   $a0, 4($sp)        # the two things we know at start

        add  $v0, $a0, $zero    # body:
        beqz $a0, epi           # end test: list == 0
        lw   $t0, 4($a0)        # list->next
        beqz $t0, epi           # end test: list->next == 0
        jal  evens              # evens(list)
        add  $a0, $v0, $zero    # ret val becomes arg
        jal  mergesort          # mergesort(evens(list))
        sw   $v0, 8($sp)        # save the result
        lw   $a0, 4($sp)        # list (my arg)
        jal  odds               # odds(list)
        add  $a0, $v0, $zero    # ret val becomes arg
        jal  mergesort          # mergesort(odds(list))
        add  $a1, $v0, $zero    # ret val becomes 2nd arg
        lw   $a0, 8($sp)        # saved result becomes 1st arg
        jal  merge              # call merge

epi:    lw   $ra, 0($sp)        # epilogue:
        addi $sp, 12            # restore $ra and stack
        jr   $ra                # return
```

This is a completely straightforward translation, in standard prologue-body-epilogue form. It's possible to shave off a few instructions by being clever at the cost of clarity; for example, by putting the end tests *before* the prologue we could avoid allocating and deallocating stack space in the base case. [Extra for experts:] More interestingly, 61A alumni should recognize the call to merge() as a tail call -- when it returns, we return. We could take advantage of that by doing the epilogue things before calling merge, and then turning the JAL instruction into a plain jump:

```
        ...
        add  $a1, $v0, $zero    # ret val becomes 2nd arg
        lw   $a0, 8($sp)        # saved result becomes 1st arg
epi:    lw   $ra, 0($sp)        # epilogue:
        addi $sp, 12            # restore $ra and stack
        j    merge              # goto merge
```

In this version our stack frame and merge's stack frame aren't both on the stack at the same time, so the memory required to run the program doesn't grow because of the call to merge. This is exactly what a Scheme interpreter or compiler does to implement tail call elimination.

But you shouldn't try to be clever when taking an exam. Just do the straightforward thing, as above.

The scoring of this problem was less straightforward than the others. We wanted a solution with several minor errors, but still basically having the right structure and many correct details, to get some credit. So we divided all the errors we found into three categories:

Minor errors: One point off per error, up to three errors. So a solution with a lot of minor errors but no others got 2 points. Note: "Minor" doesn't mean "unimportant"! Many of these errors showed important misunderstandings.

Major errors: These seemed to reflect a serious misunderstanding of the way the MIPS hardware works or the whole idea of procedure calling. Two points off per error, up to two errors.

Disasters: These suggested a complete lack of understanding, and got zero points.

There were too many minor errors to list them all, but here's a representative sample:

Using saved registers ($s0 - $s7, $16 - $23) without saving and restoring them.

Not saving and restoring $ra ($31). But note that not saving and restoring *anything* counts as a disaster.

Not reloading the argument list into $a0 before calling odds().

In the base case test, loading list->next from memory before checking whether list == NULL. (This leads to a seg fault!)

In the base case test, checking (... && ...) instead of (... || ...). Typically this means BNEZ NOT_BASE instead of BEQZ BASE.

Losing the return value ($v0, $2) from one call after another call.

Thinking that the temporary registers ($t0 - $t7, $8 - $15) are preserved over a procedure call.

Not setting $v0 to list in the base case.

Using a LW instruction in the list == NULL test. (This really tests list->value == 0.)

Here's a sample of the major errors:

Not using a LW instruction in the list->next == NULL test. (This tests &(list->next)==0, which is never true!)

Using LW or SW to copy a value from one register to another. (LW and SW always copy between a register and a memory location.)

B MERGESORT instead of JAL MERGESORT. (We think this means you never learned that recursion doesn't mean "go back to the beginning" in 61A!)

Using a memory address as an operand of an arithmetic instruction. (Arithmetic uses only register or immediate operands.)

And here are the typical disasters:

A string of consecutive JAL instructions with no transfer of values between registers.

No stack space allocated at all.

Nothing saved and restored at all.

Huge gaps in the code, e.g., only two procedure calls instead of five (odds, evens, mergesort twice, merge).

Composition of instructions: LW $A0, JAL EVENS (an attempt to put the value returned by the JAL into $a0).

If you don't like your grade, first check these solutions. If we graded your paper according to the standards shown here, and you think the standards were wrong, too bad -- we're not going to grade you differently from everyone else. If you think your paper was not graded according to these standards, bring it to your TA. We will regrade the entire exam carefully; we may find errors that we missed the first time around. :-)

If you believe our solutions are incorrect, we'll be happy to discuss
it, but you're probably wrong!