

Midterm Exam

CSC 252

28 February 2002

Directions; PLEASE READ

This exam has 6 questions, most of which have subparts. Each question indicates its point value. The total is 100 points. **Questions 3(b) and 4 are for extra credit only**; they are not part of the 100 points.

Please show your work here on the exam, in the space given. Do not write on the backs or in the margins. Put your name on every page. If your answer won't fit in the given space then you're trying to write too much. Scratch paper is available if you need it, but I will collect **only the exams**.

I have tried to make the questions as clear and self-explanatory as possible. If you do not understand what a question is asking, make some reasonable assumption and *write that assumption down* next to your answer. The proctor has been instructed not to answer questions during the exam.

You will have a maximum of 75 minutes to work. Good luck!

- (5 points) Express the decimal number 1234 in hexadecimal.
0x4d2
 - (5 points) Express the unsigned hexadecimal number 0x2ae in decimal.
686
 - (5 points) Interpret the hexadecimal bit pattern 0xffd9 as a 16-bit 2's complement number. What is its decimal value?
-39
 - (5 points) Suppose that n is a negative integer represented as a k -bit 2's complement bit pattern. If we reinterpret this bit pattern as an unsigned number, what is its numeric value as a function of n and k ?
 $n + 2^k$
- Short answer:
 - (6 points) What will the following code print on a little-endian machine such as a Pentium? What will it print on a big-endian machine such as a Sun?

```
unsigned short n = 0x1234;
unsigned char *p = (unsigned char *) &n;
printf ("%d\n", *p);
```

On a little-endian Pentium: 52; on a big-endian Sun: 18.

- (b) (6 points) How does computer hardware tell whether a 2's complement addition operation has overflowed?

The carries into and out of the high order bit are different. Equivalently: two negative numbers sum to an apparently non-negative number, or two non-negative numbers sum to an apparently negative number.

- (c) (6 points) Why do C programmers sometimes use variables of type `double` rather than `long` to store certain integer values?

Because a double has 52 bits of precision, while a long (on many machines) has only 32.

- (d) (6 points) Consider the following C declaration, compiled on a Pentium machine:

```
struct {
    int n;
    char c;
} A[10][10];
```

If the address of `A[0][0]` is 1000 (decimal), what is the address of `A[3][7]`?

$$1000 + (3 \times 10 \times 8) + (7 \times 8) = 1296.$$

3. Consider the following loop:

```
before
while (A < B) {
    body
}
after
```

- (a) (6 points) Why do compilers usually choose the translation at right rather than the one at left?

before	before
L1: <code>cmp A, B</code>	<code>cmp A, B</code>
<code>jge L2</code>	<code>jge L2</code>
body	L1: <code>body</code>
<code>j L1</code>	<code>cmp A, B</code>
L2: <code>after</code>	<code>j1 L1</code>
	L2: <code>after</code>

Because it has only one branch in each iteration of the loop, instead of two.

- (b) (Extra Credit; 6 points) Suppose the condition in the original code was something much more complicated than `(A < B)` — something that took a lot of instructions to evaluate. Can you suggest another translation that would avoid duplicating the testing code, but still have only one branch per iteration?

```

        before
        j L2          # executes only once, not once per iteration
L1: body
L2: cmp A, B        # or whatever the test should be
        jl L1
        after

```

4. (Extra Credit; 14 points) What does the following code do? Note that I am *not* looking for a detailed, instruction-by-instruction description of the code. There is a nice, short, easy-to-state, high-level answer. Your job is to find that answer.

```

movl    %edx, %eax
andl    $0xf0f0, %eax
shrl    $3, %eax
andl    $0xf0f, %edx
leal    (%eax,%eax,4), %eax
addl    %edx, %eax
movzbl  %al,%edx
movzbl  %ah, %eax
leal    (%eax,%eax,4), %eax
leal    (%eax,%eax,4), %eax
leal    (%edx,%eax,4), %edx

```

It converts the unsigned 4-digit binary-coded decimal number in register %edx to binary.

5. (a) (5 points) Imagine a 100MHz single-cycle implementation of a certain instruction set. How many nanoseconds (ns) does it take to execute each instruction?

10ns.

- (b) (7 points) Now suppose we create a 1GHz multi-cycle implementation of the same instruction set. Suppose further than this new implementation requires 3 cycles to implement a branch, 4 cycles to implement an integer arithmetic operation, 5 cycles to implement a store, 6 cycles to implement a load, and 12 cycles to implement a floating point arithmetic operation. Finally, suppose that a particular application contains 10% branch instructions, 50% integer arithmetic instructions, 10% store instructions, 20% load instructions, and 10% floating-point arithmetic operations. How much faster will the new, multi-cycle implementation run this application, compared to the old, single-cycle implementation? Hint: you'll probably find it easiest to calculate everything in terms of ns per instruction.

$0.3 + 2 + 0.5 + 1.2 + 1.2 = 5.2ns$ per instruction on average, compared to 10ns on the single-cycle machine. That's almost twice as fast.

- (c) (7 points) Now suppose we create a pipelined implementation of the same instruction set. Suppose that we have to slow the clock down to 750MHz in order to accommodate the extra control logic and the pipeline registers. If we have

perfect pipelining (no bubbles), how much faster will this processor run than the multi-cycle implementation did?

0.75 = 3/4 instructions per ns, or 4/3 = 1.33ns per instruction on average. That's about 4 times as fast as the multi-cycle implementation.

- (d) (7 points) Finally, suppose that we are able to issue a new instruction, on average, every 1.25 cycles, rather than every cycle, due to pipeline bubbles. How much faster is our processor, in practice, than the multi-cycle processor was?

1.33 × 1.25 = 4/3 × 5/4 = 5/3 = 1.67ns per instruction on average. That's a little more than 3 times as fast as the multi-cycle implementation.

6. Short answer:

- (a) (6 points) Why does the standard subroutine calling convention on the Pentium give the caller responsibility for saving half the registers and the callee responsibility for saving the other half?

To balance the desire to minimize code size with the desire to avoid unnecessary work. If only a few registers are used in the callee, the callee gets to save them, and we don't have to duplicate the save/restore code at all the call sites. If many registers are used in the callee, we only do the work of saving the rest of them in the places where the caller needs them, too.

- (b) (6 points) Why do standard stack management conventions use a frame pointer in addition to the stack pointer?

Because the stack pointer moves up and down frequently. The frame pointer provides a stable base from which to access local variables and arguments.

- (c) (6 points) Why does the IA32 ISA have a `leave` instruction, when the sequence `movl %ebp, %esp; popl %ebp` would do the same thing?

To reduce program length. The `leave` instruction is only one byte long. The equivalent two-instruction sequence is three bytes long.

- (d) (6 points) Why do RISC machines typically pass subroutine parameters in registers rather than on the stack?

Because it's faster, and because RISC machines have more registers to work with. Registers can be accessed in a single cycle. Loads from memory, even when they hit in the first-level cache, usually take at least two cycles.