Midterm Exam

CSC 252

3 March 2011

Directions; PLEASE READ

This exam has 7 questions, all but the first of which have subparts. Each question indicates its point value. The total is 100 points. Questions 3(d) and 6(c) are for extra credit only, and not included in the 100; they won't factor into your exam score, but may help to raise your letter grade at the end of the semester.

This is a *closed-book* exam. You must put away all books and notes (except for a dictionary, if you want one). Please confine your answers to the space provided.

In the interest of fairness, I will decline to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You must complete the exam in class. I will collect any remaining exams promptly at 4:40 pm. Good luck!

1. (3 points) Put your name on every page (so if I lose a staple I won't lose your answers).

2. Warm-up.

   (a) (3 points) Moore's Law has been giving us about a factor of 2 increase in transistors per square millimeter every year and a half. In nice round numbers, about how much increase is that per decade?

   Answer: About a factor of 100.

   (b) (3 points) About how many different values can be expressed with 32 bits?

   Answer: A little over 4 billion.

   (c) (3 points) What is the cycle time of a 2.5GHz processor?

   Answer: 400 picoseconds.

   (d) (3 points) Identify the world's most widely used instruction set architecture (the one with the largest annual number of manufactured units). Hint: it's used in the iPhone, and it's not the x86.

   Answer: ARM.

3. Integer arithmetic.

   (a) (4 points) Express the decimal value 4321 in hexadecimal.

Answer: $4321 = 4096 + 128 + 64 + 32 + 1 =$ `0x10e1`.

(b) (4 points) Interpret the hexadecimal value `0xfebc` as a 16-bit 2's complement number. What is its decimal value?

Answer: The most significant bit is a 1, so this is a negative number. Flipping the bits we get `0x0143`; adding 1 we get `0x0144`. So the answer is $(-1) \times (1 \times 256 + 4 \times 16 + 4) = -324$.

(c) (5 points) What is the sum of `0x5300` and `0x2d00`? Does this sum overflow as a 16-bit unsigned integer? As a 16-bit 2's complement number?

Answer: The sum is `0x8000`, which fits in 16 bits but is a negative number in 2's complement. Since the inputs were positive, the sum overflows in 2's complement, but not in unsigned math.

(d) (Extra Credit; 8 points max) What does the following code do to the values of `x` and `y`? Justify your answer.

```
int x;  int y;
...
x ^= y;  y ^= x;  x ^= y;
```

Answer: It swaps the values originally held in `x` and `y`. (Interestingly, it does this without requiring an intermediate temporary variable—something that many people find surprising.) Suppose originally `x` $= a$ and `y` $= b$. After `x ^= y` we have `x` $= a$^b and `y` $= b$. After `y ^= x` we have `x` $= a$^b and `y` $= b$^$(a$^$b)$. Exclusive or is commutative and associative (you can prove this with a truth table), so b^$(a$^$b)$ $= (b$^$b)$^$a = 0$^$a = a$. Then after `y ^= x` we have `x` $= (a$^$b)$^$a = b$ and `y` $= a$.

4. Floating point.

(a) (8 points) Consider the bit pattern `1:101 1111 1:110 0000 0000 0000 0000 0000`, which I have conveniently divided into sign, exponent, and significand fields for you as an IEEE single-precision floating point value. Give its value in the form $x \times 2^y$, where x and y are decimal values.

Answer: The sign bit is 1, so the value is negative. The exponent is $191 = 64 + 127$, so the power of 2 is 64. The mantissa is 1.significand $= 1 + {}^1/_2 + {}^1/_4$, giving us a final value of -1.75×2^{64}.

(b) (6 points) Suppose you square the value in part (a) (multiply it by itself). Does the computation cause floating-point overflow? Explain your answer.

Answer: Yes, the result overflows. $(-1.75 \times 2^{64})^2 = 1.75^2 \times 2^{128}$, but the largest power of two we can accommodate in single precision is 127.

(c) (5 points) Give values a and b, $b \neq 0$, such that $a + b = a$ in single-precision floating point. Explain your answer.

Answer: All this requires is that a exceed b by more than a factor of 2^{23}. We might, for example, choose $a = 2^{24}$ and $b = 1$. Then when b is scaled (shifted) to perform the addition, all its bits will fall off the bottom of the significand.

5. Programming in C.

(a) (8 points) Explain why the following code might perform more poorly than expected. What might you do about it?

```
int s;                  // global variable
extern int f(int);      // separately compiled function
void map_reduce(int A[], int len) {
    int i;
    for (i = 0; i < len; i++) {
        s += f(A[i]);
    }
}
```

Answer: This is an aliasing problem similar to the one discussed in class. The compiler can't be sure that `f()` does not modify `s`, so it can't keep `s` in a register during the execution of the loop: it has to write `s` to memory before calling `f()` and read it back afterward. We can solve the problem by creating our own temporary variable:

```
int s;                  // global variable
extern int f(int);      // separately compiled function
void map_reduce(int A[], int len) {
    int i;
    int t = 0;
    for (i = 0; i < len; i++) {
        t += f(A[i]);
    }
    s += t;
}
```

(This will be incorrect, of course, if `f()` actually *does* modify `s`.)

(b) (6 points) What will the following program print on a 32-bit little-endian machine like the x86? What would it print on a 32-bit big-endian machine? Explain.

```
#include <stdio.h>
int main() {
    union {
        unsigned long i;
        unsigned char s[4];
    } bytes;

    bytes.s[0] = 1;  bytes.s[1] = 2;
    bytes.s[2] = 3;  bytes.s[3] = 4;
    printf("0x%08lx\n", bytes.i);
}
```

(Hint: the magic incantation `%08lx` tells `printf` that its 2nd argument is a long (32-bit) integer that should be printed in hexadecimal, padding on the left with zeros if necessary to yield a width of 8 characters.)

Answer: On the x86 it prints `0x04030201`. On a 32-bit bit-endian machine (e.g., the SPARC) it would print `0x01020304`.

(c) (6 points) Describe at least three specific features of C that reflect the fact that it was intended to support low-level "systems programming."

Answer: There is a long list. Some possibilities include: It guarantees not to reorder the fields of structs, so they can be used to mirror hardware structures (it even allows fields to be manually aligned at arbitrary bit positions). It provides unions that can be used to break the type system. It has no garbage collection. It supports both signed and unsigned arithmetic. It avoids all run-time safety checks. It allows pointers to be created to arbitrary data. It supports pointer arithmetic. It doesn't specify whether characters (8-bit integers) are signed or unsigned (because to do so could ruin performance on non-matching machines).

6. ISA and assembler.

(a) (5 points) I have claimed in class that the direction of stack growth in the x86 (downward, toward lower addresses) is not just a software convention, but is built into the hardware. Explain.

Answer: The `push` and `call` instructions decrement the stack pointer (`%esp`); the `pop` and `return` instructions increment it.

(b) (6 points) Consider the following C function, compiled with the version of gcc we've been using in class.

```
unsigned char nth_char(int n) {
    unsigned char buf[100];
    fgets(buf, 100);
    return buf[n];
}
```

Suppose that the code for `nth_char` has loaded parameter `n` into register `%esi`. After calling `fgets()`, `nth_char()` can use a single `movzbl _____,%eax` instruction to load `buf[n]` into the return-value register. Fill in the blank in that instruction with an appropriate expression (effective address). Explain your answer.

Answer: Function `nth_char()` is simple enough that it probably doesn't have to save any registers other than `%ebp`. That means that once the frame is set up, `buf` is probably at an offset of -100 from `%ebp`. The effective address we want is `-100(%ebp, %esi, 1)`. The 1 can be omitted.

(c) (Extra Credit; 7 points max) Consider the following x86 assembly routine.

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    jmp L2
L7:
    movl    %eax, %edx
L2:
    movl    (%edx), %eax
    testl   %eax, %eax
    jne L7
    movl    %edx, %eax
    leave
    ret
```

Describe in English the likely purpose of this routine. (Do *not* tell me in low-level detail what the individual instructions do.)

Answer: Given a pointer to the head of a linked list, where the "next" pointer lies in the first word of each node, the function returns a pointer to the last node in the list.

7. Processor implementation.

(a) (8 points) The textbook describes a very simple implementation of the "Y86" instruction set (SEQ), in which each instruction executes in a single (very long) cycle. However, even though they executed only one instruction at a time, simple processors of the 1970s were not designed this way. Instead, they devoted multiple, shorter cycles to each instruction. Explain why.

Answer: There were two main reasons. The less important reason was the opportunity to use the same hardware unit (e.g., the ALU) for different purposes in different cycles (e.g., to increment the PC and later to add two registers). The more important reason was the opportunity to execute simple instructions in a small number of cycles and more complex instructions in a larger number of cycles. Because the cycle time of the single-cycle machine was determined by the length of the most complex instruction, the average time per instruction could be significantly shorter on the multi-cycle machine.

(b) (6 points) Explain why high-quality branch prediction is especially important for machines with very deep (many-stage) pipelines.

Answer: The deeper the pipeline, the more branches, on average, must generally be predicted to keep the pipeline full. When we go through several branches, misprediction rates compound. After three branches, a 90%-accurate branch predictor is wrong more than a quarter of the time ($.9 \times .9 \times .9 = .729$).

(c) (8 points) Suppose we were to modify the Y86 PIPE implementation to include a small, hidden hardware stack for return address prediction. We'd push this stack on `call` instructions (in addition to doing what we normally due to the programmer-visible stack). We'd pop it in the fetch stage of `ret` instructions, and use it to predict the address of the next instruction. If the average subroutine is 50 cycles long, and if our prediction is right 90% of the time, what percentage improvement in performance can we expect?

Answer: In the PIPE design, `ret` forces three bubbles into the pipeline, expanding our 50 cycles to 53. If we get rid of this penalty 90% of the time, we can expect to cut the average subroutine latency from 53 cycles to 50.3, an improvement of 2.7/53, or just over 5%.

By the way: you might expect the hardware predictor to be right more than 90% of the time, but only in the absence of recursion. The hidden hardware stack will be limited in size, and once recursion goes deeper than that, prediction will cease to be effective.