

# Midterm Exam

CSC 252

1 March 2007

## Directions; PLEASE READ

This exam has 5 questions, all of which have subparts. Each question indicates its point value. The total is 90 points. Questions 4(d), 5(e), and 5(f) are for extra credit only; 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.

Please put your name on every page. That way if I lose a staple I won't lose your answers. (Please do this **NOW** so you don't forget.) Scratch paper is available if you need it, but the TA will collect **only the exams**.

In the interest of fairness, the proctor has been instructed not 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. The proctor will collect any remaining exams promptly at 4:40 pm. Good luck!

### 1. Warm-up.

- (a) (4 points) Indicate the numeric value (exponent of ten) for the following Latin prefixes:

giga	=	$10^9$	kilo	=	$10^3$
mega	=	$10^6$	micro	=	$10^{-6}$
milli	=	$10^{-3}$	nano	=	$10^{-9}$
pico	=	$10^{-12}$	tera	=	$10^{12}$

- (b) (3 points) To the nearest power of ten, how many nanoseconds does it take to access main memory on a modern PC if we miss in all levels of the cache?

**Answer:** About 100.

### 2. Integer arithmetic.

- (a) (4 points) Interpret the hexadecimal bit pattern 0x2d3 as a 16-bit 2's complement number. What is its decimal value?

**Answer:** 723.

- (b) (4 points) Express the decimal number  $-926$  as a 16-bit hexadecimal bit pattern.

**Answer:**  $0xfc62$ .

- (c) (4 points) What is  $0xc793 + 0xdb14$  as an unsigned sum? (Feel free to give the answer in hex.)

**Answer:**  $0x1a2a7 = 107175$  ( $0xa2a7 = 41639$  if truncated to 16 bits).

(5 points) As a 16-bit 2's complement sum? (Please give sign and magnitude.)

**Answer:**  $0xa2a7 = -0x5d59 = -23897$ .

(2 points) Does either sum overflow 16 bits?

**Answer:** Yes, the first sum overflows. Note that while the second sum has a carry out of column 15, it also has a carry *into* column 15; it does not overflow.

- (d) (5 points) Is it ever possible for 2's complement addition to overflow when one operand is positive and the other is negative? If yes, give an example. If no, explain why.

**Answer:** No. Consider the larger-magnitude of the two numbers. Adding a smaller-magnitude number of the opposite sign moves us closer to zero, an operation that cannot overflow.

- (e) (5 points) How can the x86 get by with only one `cmp` (compare) instruction for both signed and unsigned numbers?

**Answer:** Signed and unsigned subtraction (what `cmp` does behind the scenes) perform the same bit manipulations. All that differs is the definition of overflow. The x86 sets the condition codes to reflect the outcome of the bit manipulations and lets subsequent instructions choose the combination of codes to look at.

- (f) (5 points) Given an n-bit adder, explain how to subtract with only one additional level of combinational logic (one extra gate delay)

**Answer:** Flip all the bits of the second operand (1 gate delay), then add, providing the adder with a "carry" of 1 into the one's place.

### 3. Floating-point and memory layout.

- (a) (6 points) Give two examples of mathematical laws that hold for real numbers but not for floating point numbers.

**Answer:** There are many possibilities. Among them:

$$a + (b + c) = (a + b) + c$$

$$(a + b) - b = a$$

$$(a * b) / b = a$$

- (b) (6 points) Give the bit pattern for the value 106.5, encoded as a single-precision IEEE floating point number. Remember that single precision has 8 bits of exponent (with a bias of 127) and 23 bits of significand.

**Answer:**  $1101010.1 = 1.1010101 \times 2^6 = 1.1010101 \times 2^{133-127}$   
 significand = 101010100000000000000000  
 sign = 0  
 exponent 133 = 10000101  
 0 10000101 101010100000000000000000  
 = 0100 0010 1101 0101 0000 0000 0000 0000 = 0x42d50000.

- (c) (3 points) Suppose you write the 32-bit integer 0x12345678 to a file on a little-endian machine, transfer the file to a big-endian machine, and then read the integer back in again. What value will you get?

**Answer:** 0x78563412.

(3 points) Now suppose you read the same bits back in as two 16-bit integers, on both machines. What will you get on the little-endian machine?

**Answer:** 0x5678 and 0x1234.

(3 points) On the big-endian machine?

**Answer:** 0x7856 and 0x3412.

#### 4. ISA and assembler.

- (a) (3 points) What value does the following instruction sequence leave in `%eax`?

```
movl    $20, %eax
movl    $10, %ecx
leal    12(%eax, %ecx, 4), %eax
```

**Answer:**  $12 + 20 + 10 \times 4 = 72$ .

- (b) (4 points) Why do most processor architectures have separate integer and floating-point register sets?

**Answer:** (1) With the register set implicit in the opcode, fewer bits are required to encode instruction operands. (2) By letting the integer ALU access the integer registers and the FP ALU access the FP registers, we minimize interference between separate integer and FP pipelines. (3) On some machines the integer and FP registers are different lengths.

- (c) (4 points) Why do most recent processor architectures allow only load and store instructions to access memory?

**Answer:** To preserve predictable timing for all the other instructions, for the sake of smooth pipelining.

- (d) (Extra Credit; 5 points max) What does the following function do? (Note: There is a simple high-level answer to this question. Do *not* give me a detailed instruction-by-instruction description of the code.)

```

foo:
    pushl    %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    movl    16(%ebp), %ecx
    cmpl   %eax, %edx
    jge    .L1
    movl   %edx, %eax
.L1:
    cmpl   %ecx, %eax
    jl    .L2
    movl   %ecx, %eax
.L2:
    popl   %ebp
    ret

```

**Answer:** It returns the minimum of its three integer arguments.

## 5. Processor implementation.

- (a) (3 points) What is the principal difference between the SEQ and SEQ+ processor implementations described in chapter 4 of the textbook?

**Answer:** SEQ performs a “next-PC” calculation at end of its long cycle; SEQ+ performs a “this-PC” calculation at the beginning of the cycle (in preparation for the subsequent pipelining examples).

- (b) (3 points) What is the principal difference between PIPE– and PIPE?

**Answer:** Forwarding.

- (c) (6 points) Give three examples of things a compiler might want to know about how the processor implements the ISA (instruction set architecture).

**Answer:** There are many possible answers. Among them: How long is the load delay for a level-1 cache hit? How are branches predicted (is there, for example, a preference for reverse-taken loops)? What is the issue width (the number of instructions that can be started concurrently, if they are independent)? How long are the integer multiply, integer divide, and floating point execution delays? Are these units internally pipelined? What are the relative costs of complex instructions and equivalent multi-instruction sequences? What is the cache block size, the associativity of the L1 cache, and the total L1 cache size? (Note that we haven’t really covered caches yet.)

- (d) (5 points) In class I described a processor implementation strategy in which instructions are executed sequentially, but in multiple cycles each. If instructions can’t overlap, what advantage(s) does such a strategy have over an implementation like SEQ+?

**Answer:** (1) Less chip area is required, because a single hardware unit can be used for different purposes in different cycles of the same instruction. (2) Average time per instruction may go down, since the clock is much faster and many instructions take fewer than the maximum number of cycles to complete. NB: This is the implementation strategy used in classic microprogrammed machines.

- (e) (Extra Credit; 3 points max) Most processors have an *indirect jump* instruction, which changes the PC to the address found in a register. Indirect jumps are commonly used for **switch** statements, function pointers, and virtual method dispatch in object-oriented programs. The Y86, however, has no indirect jump instruction. How would you implement these special forms of control flow on such a restricted machine?

**Answer:** Use the **ret** instruction: push the address you want to jump to onto the stack, then execute a “return”.

- (f) (Extra Credit; 7 points max) The Y86 PIPE example in the book is a single-issue, in-order processor with a short (5-stage) pipeline. There’s nothing wrong with this *per se* (many embedded processors are similarly simple). But certain other aspects of the Y86 instruction set and the PIPE implementation are arguably unrealistic. Explain.

**Answer:** The lack of indirect jumps, as noted in the previous question, is unfortunate. So, too, are the lack of integer multiplication and division, or any sort of floating point. The implementation also embodies an extremely simplistic branch prediction strategy, and doesn’t predict returns at all. None of these are terrible. More serious omissions are (1) the lack of a mechanism to handle cache misses (the example assumes that memory always responds in a single cycle) and (2) the lack of any provision for exceptions or external interrupts.