Midterm Exam

CSC 252

26 February 2009

Directions; PLEASE READ

This exam has 7 questions, all of which have subparts. Each question indicates its point value. The total is 90 points. Questions 3(d) and 7(e) are for extra credit only, and not included in the 90; 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) How many bytes are in a kilobyte?

   Answer: 1024.

   (b) (3 points) How many milliseconds are in a second?

   Answer: 1000.

   (c) (3 points) To the nearest power of ten, how many milliseconds does it take to access data on a modern hard disk?

   Answer: About 10.

3. Integer arithmetic.

   (a) (4 points) Express the bit pattern `1011 0110 1001 0101` in hexadecimal.

   Answer: `0xb695`.

   (b) (6 points) Interpret this bit pattern as a 16-bit 2's complement number. What is its decimal value?

Answer: Flipping the bits and adding one we get `0100 1001 0110 1011`. So the answer is $(-1) \times (4 \times 4096 + 9 \times 256 + 6 \times 16 + 11) = -18795$.

(c) (8 points) What is the largest magnitude negative number that can be added to this value without causing 16-bit 2's complement overflow? (Feel free to calculate your answer in decimal or in hex.)

Answer: We want to add up to the largest magnitude negative number that can be expressed in 16-bit 2's complement, namely $-2^{15} = -32768$. So the answer is $-32768 - (-18795) = -13973$.

Alternatively, if you prefer to think in bit patterns, you can "fill in the holes" in the original value, flip the "sign" bit, and add one: `1100 1001 0110 1011` = `0xc96b`. When this is added to the original bit pattern we'll get `8000`, which is the largest magnitude negative number expressible.

(d) (Extra Credit; 10 points max) Prove that first decrementing and then complementing is equivalent to complementing and then incrementing. That is, for any signed value `x`, the C expressions `~x+1` and `~(x-1)` yield identical results.

Answer: Assume we are working with n-bit numbers. Suppose $-2^{n-1} + 1 < x \leq 2^{n-1} - 1$. In this range, $-(x-1)$ does not overflow, and equals $\tilde{}(x-1)+1$. Subtracting 1 from both terms, we have $-x = \tilde{}(x-1)$. But we also know that in this range $-x = \tilde{}x + 1$. So $\tilde{}x + 1 = \tilde{}(x-1)$.

The only values of x we have not considered are the two most negative values, represented by the binary strings 10...00 and 10...01. It is easy to verify that the equality holds for these as well, though the computations above would overflow.

4. Floating point.

(a) (7 points) Give the bit pattern for the value -9.25, 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: $-1001.01 = -1.00101 \times 2^3 = -1.00101 \times 2^{130-127}$
significand = 001 0100 0000 0000 0000 0000
sign = 1
exponent = 130 = 1000 0010
answer = 1 1000 0010 001 0100 0000 0000 0000 0000
= 1100 0001 0001 0100 0000 0000 0000 0000 = 0xc1140000.

(b) (3 points) Which is larger, when interpreted as a single-precision IEEE floating-point number, `0x43210000` or `0x12340000`? Justify your answer. (Hint: this is not a difficult problem!)

Answer: `0x43210000`. As long as the sign bit is zero, floating-point numbers are ordered the same as integers with the same bit pattern.

5. C arrays. Consider the following global variable declarations in C:

```
int A[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    // contiguous layout
int r1[3] = {1, 2, 3};
int r2[3] = {4, 5, 6};
int r3[3] = {7, 8, 9};
int *B[3] = {r1, r2, r3};
    // row-pointer layout
```

   (a) (3 points) What syntax would you use to access element 1 of row 2 (the value 8) in each of `A` and `B`?

   Answer: The same: `A[2][1]` and `B[2][1]`.

   (b) (7 points) Suppose we have declared

```
int *p = A[0];
int **q = B;
```

   Show how to access the same elements of `A` and `B` (the 8s) using pointer arithmetic on `p` and `q`, respectively.

   Answer: `*(p+7)`; `*(*(q+2)+1)`.

6. ISA and assembler.

   (a) (3 points) What is the most likely purpose of the x86 instruction `movl 8(%ebp), %eax`? (Express the answer in high-level terms if you can.)

   Answer: Load an argument of the current subroutine (specifically, the first argument) into register `%eax`.

   (b) (10 points) Consider the following C code (left) and assembler (right):

```
struct S {
    int a;
    int b;
    int c;
};

void foo(struct S[] A, int v) {
    int i;
    for (i = 0; i < 10; i++) {
        A[i].a = v;
    }
}
```

```
.globl _foo
_foo:
    movl  $0, %ecx
L3:
    movl  %ecx, %edx
    imull $12, %edx
    addl  4(%esp), %edx
    movl  8(%esp), %eax
    movl  %eax, (%edx)
    addl  $1, %ecx
    cmpl  $9, %ecx
    jle   L3
    ret
```

   The assembler is a correct but sub-optimal translation of the C. Identify two ways the compiler could make the loop faster. (You don't have to show new code; just sketch the answer in words.)

Answer: (1) Hoist invariants out of the loop. Trivially, there is no need to load `%eax` from memory in every iteration. We could load it before the loop and just keep it in the register. With the allocation of one additional register (which we would need to save at the beginning of the routine, because we've already used all three caller-saves registers), we could also keep the address of `A` in a register, and avoid loading it on every iteration.

(2) Strength reduce the index calculation. Instead of setting `%edx` to 12 times `%ecx` on every iteration, *add* 12 to it at the bottom of the loop. If we do that, we could test for `%edx` == 108 at the bottom of the loop, instead of testing for `%ecx` == 9. Then we wouldn't need `%ecx` at all, and could use it instead of a caller-saves register to hold the address of `A`. Better yet, we could initialize `%ecx` with the address of `A` and `%edx` with the address of the last element to be modified, at which point we wouldn't have to add the two registers in every iteration.

(3) Unroll the loop.

You didn't have to show new code to get full credit on this question, but if you did the version without unrolling might look like this:

```
.globl _foo
_foo:
    movl  4(%esp), %ecx
    movl  %ecx, %edx
    addl  $108, %edx
    movl  8(%esp), %eax
L3:
    movl  %eax, (%ecx)
    addl  $12, %ecx
    cmpl  %edx, %ecx
    jle   L3
    ret
```

(c) (5 points) The MIPS, SPARC, and PowerPC instruction sets have 32 integer registers each. The x86 has only 8. Why?

Answer: Because the x86 was designed a long time ago, when it was more important to economize on the number of bits devoted to register names in each instruction than to provide the compiler with a lot of scratch registers.

7. Processor implementation.

(a) (3 points) What is the principal source of pipeline bubbles in a modern processor?

Answer: Cache misses.

(b) (8 points) Explain the basic idea behind out-of-order execution.

Answer: Instead of forcing instructions to execute in order, keep a *set* of upcoming instructions in the processor, each tagged with an indication of the inputs it needs. On every cycle, choose one or more instructions whose inputs are available and execute those instructions, even if there are earlier instructions that have not yet been able to execute (e.g., because they are waiting on a cache miss).

(c) (5 points) What is the principal downside of out-of-order execution? Why isn't it used in, say, the recent Sun Niagara processors?

Answer: It requires a *lot* of bookkeeping, which consumes chip area and, crucially, energy. The increase in performance achieved by out-of-order processing is significantly less than the increase in energy consumption (and heat dissipation). The Niagara processor improves performance per joule by using in-order cores.

(d) (6 points) In the pipelined version of the Y86 processor (see next page), the data memory takes three inputs: a control signal that indicates whether to read, write or neither in the current cycle; an address signal that indicates the location (if any) to be accessed; and a data signal that indicates the value (if any) to be written. Consider the circuit that generates the address signal (the "Addr" box in the processor diagram). When does it take its input from the valE field of the M register? When does it take its input from the valA field? What control input (not shown) does it need to make this decision?

Answer: For `push`, `call`, `rmmov`, and `mrmov` instructions, Addr tells the memory to access the location specified by valE. This will have been computed by the Execute state in the previous cycle, by adding 4 to `%esp` (in the case of `push` and `call`), or (for `rmmov` and `mrmov`) by adding a displacement (possibly zero) to a value from a register or the immediate field of the instruction. For `pop` and `ret` instructions, Addr tells the memory to access the location specified by valA. All it needs for a control input is the icode field of the M register.

(e) (Extra Credit; 8 points max) Suppose we were developing a new generation of Y86 processor. As part of our design effort, we might study programs run on the current model to find sequences of instructions that are frequently executed together. For each such sequence, we could introduce a new instruction that does the work of the entire sequence. We might, for example, replace

```
mrmov (%ebp), %ebp
rrmov %ebp, %esp
```

with

```
leave
```

(as in the x86). Or we might replace

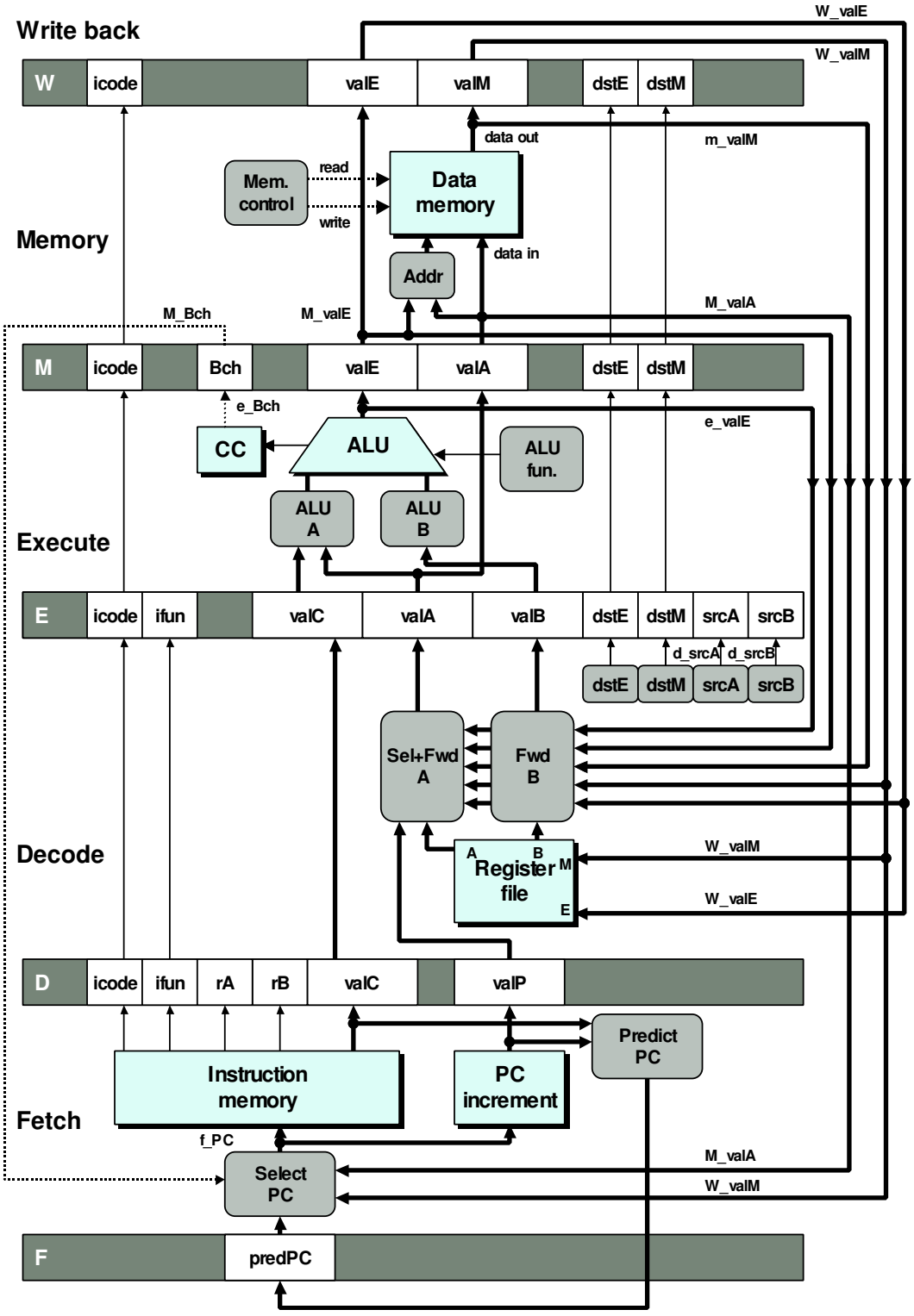
```
irmov val, reg1
op    reg1, reg2
```

with

```
iop   val, reg2
```

Under what circumstances could we expect such extensions to the instruction set to lead to better performance?

Answer: If every stage of the new instruction can be completed as quickly as the slowest stage of any existing instruction, then programs should run faster, because they'll execute fewer instructions. If the new instruction lengthens the slowest stage, we'll have to decide whether the improvement in number of instructions outweighs the reduction in cycle time (it probably won't). But the impact may also be more subtle: By consuming chip area, the new instruction may force the use of longer wires, which at some point in the future (in a new generation of chips) may become the bottleneck on cycle time. By increasing the complexity of instruction encodings, the new instruction (or a set of new instructions) may preclude subdividing the decode stage and building a deeper pipeline.


Write back
W_valE
W_valM
W
icode
valE
valM
dstE
dstM
data out
m_valM
Mem. control
read
write
Data memory
Memory
data in
Addr
M_Bch
M_valE
M_valA
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
e_valE
CC
ALU
ALU fun.
ALU A
ALU B
Execute
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
Decode
A
B
Register file
M
E
W_valM
W_valE
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
Fetch
f_PC
M_valA
W_valM
Select PC
F
predPC