

Midterm Exam

CSC 252

24 February 2005

Directions; PLEASE READ

This exam has 5 questions, all of which have subparts. Each question indicates its point value. The total is 100 points. **Questions 4(f), 4(g), 5(f), and 5(g) 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 (or *way* too big). Scratch paper is available if you need it, but the TA 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 TA has been instructed not to try to answer questions during the exam.

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

1. Binary integers.

- (a) (4 points) Express the decimal number 432 in hexadecimal.

0x1b0

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

-52

- (c) (4 points) Suppose that a is a k -bit 2's complement number and b is its k -bit additive inverse (negative). If we reinterpret their bit patterns as k -bit *unsigned* numbers, add them, and keep any carry out of the leftmost place, what number do we get?

2^k . That's why it's called 2's complement.

- (d) (4 points) Suppose we use an integer divide instruction to divide -20 by -3 . What will be the quotient? What will be the remainder?

Quotient 6; remainder -2 .

2. Floating point.

- (a) (5 points) What are *denormal* numbers? What purpose do they serve?

They're numbers close to zero that have an implied zero bit to the left of the binary point, and therefore a reduced number of significant bits. They provide

gradual underflow, so we don't have a (comparatively) big gap between zero and the smallest magnitude numbers.

- (b) (5 points) A positive single precision floating point number with significand f and exponent e has value $1.f \times 2^{e-127}$. The value 127 is called the *bias*. What purpose does it serve?

It ensures that larger numbers always have "larger" exponent values, when interpreted as an unsigned integer. This allows us to use the same comparator circuit for both integer and FP numbers.

- (c) (5 points) If x , y , and z are C variables of type `double`, will $(x + y) + z$ have the same value as $x + (y + z)$? Explain.

Not necessarily. If x and y have opposite sign and approximately the same large magnitude, but z has much smaller magnitude, then z may be lost when added to y in the second expression, but may be quite significant in the first, because x and y will roughly cancel out.

3. Memory layout.

- (a) (10 points) What does the following program print on a 32-bit Pentium? Explain. (Be careful! This problem tests your understanding of array layout, byte order, and alignment. Remember that fields of a union lie on top of each other.)

```
#include <stdio.h>

int main() {
    int i;
    union {
        unsigned char A[8];
        struct {
            unsigned char c;
            int n;
        } S;
    } U;

    for (i = 0; i < 8; i++) U.A[i] = 0;
    U.A[4] = 0x3;
    printf("%x\n", U.S.n);
}
```

It prints a 3. The 8 bytes of A lie on top of S. There is a 3-byte hole after c, so that n can be longword-aligned. A[4] therefore coincides with the first byte of n in memory. Because the Pentium is a little-endian machine, this is the least significant byte.

- (b) (5 points) Here is the header for a C function to copy row i of an $N \times N$ matrix into an N -element vector. Fill in the body of the code.

```
void copy_row(int *M, int i, int N, int *V) { ...
```

```
int j; for (j = 0; j < N; j++) V[j] = M[N*i+j];
```

Equivalently:

```
int *end = V+N; for (M+=(N*i); V < end; M++, V++) *V = *M;
```

- (c) (4 points) Explain the difference in C between `int *a[n]` and `int (*a)[n]`.

*int *a[n] is an array of n pointers to integers; int (*a)[n] is a pointer to an array of n integers.*

- (d) (4 points) What is the value of `n` after executing the following C code on a 32-bit Pentium? Explain.

```
int A[100];
int *p = &A[10];
int *q = &A[20];
int n = q - p;
```

10. The size of the array elements is immaterial. The C compiler computes the numeric difference in indices.

4. Understanding assembler.

- (a) (4 points) Why do C compilers usually push function arguments in reverse order (last one first)?

So the first argument will be at a known offset from the frame pointer. This is necessary for functions that accept variable numbers of arguments.

- (b) (4 points) Why do `switch` statements exist? Why not just use `if...else if...else`?

Because compilers can generate particularly efficient code for `switch` statements using techniques like array indexing, hashing, and binary search.

- (c) (4 points) Name four different instruction set architectures currently in widespread use.

I've mentioned the following in class: ARM, x86, IA-64, Sparc, MIPS, Power/PowerPC, 680x0, IBM 360/370/3090.

- (d) (5 points) *Branch* instructions change the program counter by adding or subtracting an offset. *Jump* instructions change the program counter by setting it to some absolute value. Why does the x86 provide both?

Branch instructions can be encoded in fewer bits, and can be relocated in memory without changing the value in the instruction, but they can only reach a limited distance from the current location. Jump instructions are longer, and not relocatable, but can reach arbitrary locations.

- (e) (5 points) The stack generally grows downward, toward smaller addresses. Is this merely a software convention on the x86, or is it reflected somehow in the hardware? Explain.

It's reflected in the hardware: the `push`, `call`, and `enter` instructions subtract from `%esp`; the `pop`, `leave`, and `ret` instructions add to it.

- (f) (Extra Credit; 6 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. Be sure to pay careful attention to the difference between `%eax` and `%edx`.)

```
foo:
    pushl    %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    popl    %ebp
    leal   (%edx,%edx,4), %eax
    leal   (%edx,%eax,4), %eax
    ret
```

It multiplies its argument by 21 and returns the result.

- (g) (Extra Credit; 4 points max) Describe *two* problems posed by the ability to execute code in the stack on a Pentium.
- (1) *It leaves programs with inadequate bounds checking vulnerable to buffer overflow attacks.* (2) *It makes it difficult for hardware designers to implement the first-level caches: we have to notice when a store instruction modifies a location currently in the instruction cache.*

5. Processor implementation.

- (a) (5 points) Explain the difference between combinational and sequential circuits. *Combinational circuits implement a pure Boolean function from inputs to outputs. They have no feedback (wiring cycles). Sequential circuits may have multiple stable states, allowing them to store values. They require feedback.*
- (b) (4 points) What is the time and space complexity of a carry-lookahead adder? (Use big-O notation.)
 $O(\log n)$ time, $O(n)$ space, where n is the number of bits in the numbers to be added.
- (c) (5 points) Name the five stages of the Y86 PIPE implementation in the textbook. *Fetch, Decode, Execute, Memory, Writeback.*
- (d) (5 points) Why does the Y86 PIPE implementation predict its way through conditional branches, but not through `ret` instructions? More specifically, it speculatively fetches instructions from the target of a conditional branch, but stalls the pipeline at a `ret`. Why?
A conditional branch instruction contains the target address of the branch, so it's available in the first stage of the pipeline. A `ret` instruction obtains the return address from the stack, so it's not available until the fourth stage of the pipeline.
- (e) (5 points) Briefly explain the purpose of the reorder buffer (ROB) in an out-of-order processor. (The book calls this the *retirement unit*.)

It serves to make interrupts precise. An instruction is not allowed to commit (retire) until all instructions from earlier in program order have committed. Exceptions, if any, are generated at commit time.

- (f) (Extra Credit; 4 points max) The PowerPC has multiple sets of condition codes. An instruction like `test` or `compare` specifies which set of codes to set. A branch instruction specifies which set of codes to inspect. What advantage does this provide over the single set of the x86?

Like software register renaming, multiple condition codes allow the compiler to avoid artificial dependences among instructions. The machine can have more than one set of test-and-branch instructions “in flight” at the same time.

- (g) (Extra Credit; 6 points max) The PowerPC G5 and the Pentium 4 have comparable computational power, comparable implementation complexity, comparable heat dissipation, and comparable manufacturing cost. The x86 has more instructions, but not a lot more. Yet the x86 is supposedly a CISC machine and the PowerPC is supposedly a RISC machine. What’s the difference, and does it really matter?

The G5 has uniform length instructions, a shorter pipeline, wider issue width (it’s more superscalar), many more registers, and 64-bit native operation. It keeps many more instructions in flight at one time. It has no need for front-end translation from machine language to micro-ops. By doing more per cycle, it achieves comparable computational power with a substantially slower clock. These differences suggest that it will enjoy lower development cost for new versions, less vulnerability to mispredicted branches, and more “headroom” to increase the clock rate and (further) increase instruction-level parallelism. These advantages may not mean much, however, if neither IBM nor Intel is able to cool more complex chips.