

# CS573

## Final Exam

### Solutions

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no calculators** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your social security number, but not your name, on each sheet of paper you turn in
- show your work whenever appropriate. There can be no partial credit unless I see how answers were arrived
- be succinct. I will take off points for facts that, while true, are not relevant to the question at hand

You have until 12:30 to finish the exam. The questions are equally weighted.

1. Assume the three numbers below are eight bit integers protected with a SECDED scheme as presented in class. Assume the order of the bits is

$M_8M_7M_6M_5C_8M_4M_3M_2C_4M_1C_2C_1P$

(this is the order given in the definition in the notes). For each of the numbers, tell whether (1) there is no error, (2) there is a one-bit error (and tell which bit is wrong), or (3) there is a two-bit error.

*I'll show how to work the answers by showing a row for each check bit, with only the bits used to determine that check bit shown. Since I'm using the check bit itself in the calculation, I should get a column of all 0's; if I don't, the check bits will contain the bit number of the erroneous bit.*

(a) 0011001011100

0011001011100	0	P
00110	0	C8
0 0101	0	C4
01 01 11	0	C2
0 1 0 0 1 0	0	C1

*All check bits (including P) show OK; there is no error.*

(b) 0110010100100

```

0110010100100  1 P
01100           0 C8
0   1010       0 C4
11  10  01     0 C2
1 0 1 1 0 0   1 C1

```

The parity shows an error, and C1 shows an error: there is a one-bit error, in bit C1. Bit C1 should contain a 1 instead of a 0.

(c) 1110110001001

```

1110110001001  1 P
11101           0 C8
1   1000       0 C4
11  10  10     0 C2
1 0 1 0 1 0   1 C1

```

The parity shows an error, and so does bits C1. So there is an error in bit C1. Bit C1 should contain a 1 instead of a 0.

Grading notes:

- Didn't realize a check bit can be wrong: -3
- Parity wrong, didn't tell me how it was calculated: -3
- Got different answers for B and C after computing correctly: -3
- Conclude 2 bit error in B, C; not clear how: -5
- Conclude 2-bit error; tell me which bits: -5
- Used odd parity for C: -5
- Calculated parity on C bits only: -5

2. Here's a problem J Strother Moore described during his colloquium Monday (I'm changing the presentation a bit, but it's the same problem):

Suppose you have two processes, each running the following code

```

global int x;
local int t1, t2, t3;
t1 = x;
t2 = x;
t3 = t1 + t2;
x = t3;

```

(assume the initial value of x is 1)

- (a) Using the notation for global memory interactions that we've used in class, show how it is possible to have interleavings of reads and writes to x that will result in x having a final value of 2, 3, or 4, assuming strict consistency.

Just to be clear, the notation I mean is the one that looks something like

```

P1: R(x)1
-----
P2:
P1: R(x)1 R(x)1 W(x)
-----
P2: R(x)1 R(x)1 W(x)
gives a result of 2
P1: R(x)1 R(x)1 W(x)
-----
P2: R(x)1 R(x)2 W(x)
gives a result of 3
P1: R(x)1 R(x)1 W(x)
-----
P2: R(x)2 R(x)2 W(x)
gives a result of 4

```

- (b) Insert synchronization primitives of your choice to guarantee the result will be 2.

```

global int x;
local int t1, t2, t3;
t1 = x;
t2 = x;
barrier();
t3 = t1 + t2;
x = t3;

```

will guarantee that the results of the addition can't propagate between processes:

```

P1: R(x)1 R(x)1 B W(x)
-----
P2: R(x)1 R(x)1 B W(x)

```

Grading notes:

- Tried to use a read lock; didn't keep first process from completing before second acquired lock: -5
- Shows local reads/writes: -5
- Doesn't show all global reads/writes: -5

3. Here's a statement about instruction sets I found on Usenet a while ago:

```

Subject: Re: Is RISC dead? (was: Re: K7's FP
performance inches past P-III's)
Date: 1999/05/19
Author: Piercarlo Grandi <piercarl@Dial.PIPEX.com>

```

Well, my take here (and I said so in "comp.arch" quite a while ago) is that `_architecture_` is dead: with essentially infinite silicon/design budgets virtually any architecture, including x86, can be made to perform; in other words only implementation matters.

Have we reached the point that the instruction set is irrelevant, because with current technology any pig can be made to fly? Are some aspects of the architecture still important, and others less so? Justify your answer.

*Grading notes: the grading of this question is necessarily subjective, and I'll be trying to assign points based on (1) the extent to which you make comments relative to the question, and (2) the extent to which your comments support your position.*

*My own view is that there are aspects of architecture, traditionally defined, that are inarguably still relevant: word size comes to mind immediately. Beyond that isn't so clear; the extent to which IA32 is now implemented by taking the code as generated and translating it at run-time into something more amenable for rescheduling argues in favor of Piercarlo's point. It would appear that Intel themselves don't believe it, though, since they've devoted billions of dollars to getting a 64-bit, non upward-compatible instruction set to work precisely because they anticipate compilers can do this better than the hardware can.*

*A point brought up by one student is that ISA is important, not from any sort of performance reason, but because maintaining compatibility helps hold on to customers. Not the way a computer architect normally thinks, but very important indeed! Another student brought up a related point, that it's likely the recycling of ISA from generation to generation (specifically in IA-32) gives an advantage to the current ISA that a clean-sheet, equally complex, instruction set would not enjoy (too bad this one didn't get developed in the answer...).*

*One difficulty that arises in grading this is that the question is really aimed at narrow-view (i.e. instruction set) architecture, while our class has been about broad-view (i.e. also implementation) architecture; I'll be accepting arguments regarding implementation as well (and if you include implementation, the claim becomes patently absurd since **of course** the implementation is still important!).*

4. Suppose you are selecting a network topology for transmitting 128 byte messages. You can have either a wormhole-routed 2D square grid with with a one-byte flit, or a store-and-forward binary hypercube. Links in the two topologies have identical bandwidths.

How many nodes would you need to have before the hypercube was faster than the grid (determined by worst-case time to send a 128 byte message, assuming no blocking)?

One hard thing about this problem is finding the right way to express the number of processors — if we express the number of processors as  $n$ , then we end up comparing  $\sqrt{n}$  with  $\log_2 n$ , which is a mess. But if we consider  $n^2$  processors, we end up comparing  $n$  to  $2\log_2 n$ , which isn't nearly so painful. So, with that in mind...

Worst case in a grid is corner-to-corner; if we have  $n^2$  processors, we need  $2(n-1)$  hops to get the first flit across. We have a 128 byte message, and a flit is (conveniently) one byte, so the total time is  $2(n-1)+127 = 2n+125$ . In the hypercube, the worst case is  $2\log_2 n$  hops; since it's store-and-forward every hop takes 128 time units for a total of  $256\log_2 n$ . That means we just need to solve  $256\log_2 n < 2n + 125$ ; a little algebra transforms this into  $\log_2 n < \frac{n}{128}$  (that pesky  $1/2$  isn't going to be noticeable so we'll ignore it). At this point I'll admit to just doing an exhaustive search;  $\log_2 1024 = 10$ , while  $\log_2 2048 = 11$  so the crossover happens at  $n = 2048$ ; since we really have  $n^2$  processors this means the wormhole-routed grid is faster than the hypercube until we have more than 1,000,000 processors. This leaves open the possibility that the crossover is really at  $2^{21}$  or at  $2^{22}$ ; trying them shows that  $256 \times 10.5 > 1500 + 125$  but  $256 \times 11 < 2000 + 125$  ( $1500 > \sqrt{2^{21}}$ ).

No wonder Intel switched.

Grading notes:

- Sorry, guys, but big-O notation is no help whatever when trying to compute crossover points like this. You need all those pesky constants...
- Combined big-O with constants (like  $O(\log n) * 128$ ): -5
- Ignored constants: -8
- Numbers from Mars (near as I could tell): -15

5. Dataflow and scoreboarding (out-of-order execution as used in the Cray 1 or the Pentium Pro) are alternative ways to extract the maximum parallelism from programs. Compare the two approaches. Is one more likely than the other to be able to extract maximum parallelism?

The two are really much more similar than they appear on the surface: register reservations can be viewed as arcs in a graph; the producing instruction is the source, and any consuming instructions are sinks. With that view, the scoreboard is being used to maintain a subgraph of the dataflow graph of a program, and the CPU is executing that subgraph effectively as a dataflow computer. This becomes particularly true when we start doing register renaming so the register name becomes a really abstract label, decoupled from actual registers.

Once we've taken that view, there is no reason to suppose either approach would have an inherent advantage in exploiting parallelism. If we can have an arbitrarily large number of pipes in our conventional computer,

*we should be able to make use of as much parallelism as the flow graph makes available. Branches remain problematic, but increased parallelism makes increased use of predication more appealing too.*

*Grading notes: like 3, grading this is unavoidably subjective; the main point I'll be looking for is that scoreboarding is dataflow on a small scale.*

6. Back in the Golden Age of Supercomputers, one of the distinguishing characteristics of a supercomputer was the capability to perform vector operations. Today, current processors seem to have almost completely abandoned that technology; even things like Intel's MMX or SSE2 extensions to IA32 can only operate on trivially short vectors. Given that there is still a need to perform the sorts of problems that vector instructions were aimed at, why has this happened?

*Well, what did the vector instructions really give us? Parallelism. It let us have a number of instructions in flight simultaneously. Today, for the same (or less) money, we can wire up a pile of PCs to get vastly more parallelism.*