

CS 573

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 electronic aids** 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. You may lose 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. Some computers (including VAX) use a single set of registers to hold both integer and floating point values. Most, however, use a set of integer registers and a separate set of floating point registers. How does this decision affect the ease of constructing the CPU - in which case is the register file(s) easier? In which case are the datapaths simpler?

Having two sets of registers – integer and floating point – makes the datapath implementation simpler, by allowing us to have essentially two completely separate datapaths. First, the integer registers only need to have enough read ports to feed the integer units, and enough write ports to support their results. Similarly, the floating point registers only need to support the floating point units. One complexity that does arise is that we need to have a mechanism to do a register transfer from one side to the other. One student brought up that we'd need more space for two register sets; a valid point, but it's not clear how important a point it might be.

A number of students brought up some other points, which weren't what I was asking for but seem worth considering. It's likely it will be possible to have more registers by using separate integer and floating point sets, due to the space constraints in the instruction set. If we only have five bits to specify a register number, using a single set of registers lets us have 32 registers while having two sets lets us have 64. This shouldn't end up transferring information to op code space, since we already have to have, for instance, separate integer and floating point addition instructions. It might appear that we could use the registers more flexibly if we have a single register set, but that's only true if the total number of registers is the same in both situations (everything else being equal). The Alpha paper mentions this, and a bit of reflection shows the author hadn't clearly thought it out... A number of people thought building the register file would be easier in the combined case; I wound up looking to see if there was something in the Alpha paper that would lead people to believe that and came up empty...

A huge number of people brought up the claim (from the Alpha paper) that using a single register file would make passing parameters easier. First, it isn't clear to me that this is true for a language that uses proper function prototypes, and second what does it have to do with the question? Similarly, while it undoubtedly allowed for an easier two-chip implementation, what relevance does that have today (other than pointing to the separate-datapaths argument I wanted people to make, and which I don't think anybody who brought up this red herring followed)? And what does the possibility of an integer-only version have to do with the question I asked?

2. The Intel Pentium 4 processor uses a “trace cache” to store decoded instructions, instead of a normal instruction cache storing them in their original non-decoded form. So far as I've been able to find, they have been quite secretive regarding exactly what goes in a decoded instruction in the trace cache (as well they should be!). One thing they *could* put there would be branch prediction information (so every decoded instruction in the trace cache would also have information regarding whether or not it is a branch, and a prediction as to whether it would be taken if it is). How this would compare to using a normal branch target buffer - would it cost more or less? would it be more or less accurate? would it be a good idea?

In answering this question, it's important to remember what the Pentium 4 already does regarding branch prediction. It does in fact have a predictor in the trace cache (this was stated in the paper), but Intel is pretty vague about just how it works. It's a good assumption, though, that it's a fairly normal BTB-style predictor (answers based on doing prediction based on decoded instruction being a bad idea took a bad left turn right there!).

So the question is whether it would be good to tag each decoded branch instruction with the information that would normally be in the branch predictor. Doing so could increase accuracy (since there could never be a case of two instructions getting confused with each other in the predictor), but would be much, much more expensive since the decoded instructions would have to be larger. On balance, it probably wouldn't be a good idea.

There were many, many answers based on comparing doing branch prediction in the trace cache rather than based on IA-32 instructions; that wasn't the question. The answers that were just cutting-and-pasting the paragraphs on trace caches from the paper were particularly off the mark.

3. Calculate the frame check sequence for the following 12 bit message and 4 bit polynomial:

Message: 0x9b3
 Polynomial: $x^3 + x^2 + 1$

My notes on CRC are unclear; I show how to do the division, not the full CRC computation. Because of this, I'm accepting two answers. The first one is just dividing the message by the polynomial, it isn't actually computing a FCS. Here it is:

```

100110110011
1101
10010110011
1101
1000110011
1101
101110011
1101
11010011
1101
0011
  
```

So the "FCS" is 011

To compute the actual FCS of a message, you need to multiply it by the degree of the polynomial before doing the division. So it looks like this:

```

100110110011000
1101
10010110011000
1101
1000110011000
1101
101110011000
1101
11010011000
1101
0011000
1101
010
  
```

So the real FCS is 010. If we replace the 000 that was appended to the original message with the FCS of 010, then when we perform the division we'll end up with a remainder of 0 which will tell us the message was transmitted OK.

- 3 Went a step too far or not far enough
- 3 Mistook hexadecimal for duodecimal
- 5 Used polynomial notation???
- 5 Didn't use modulo-2 arithmetic
- 3 Unexplained extra bits tacked on

Amazingly, several students attempted to compute the FCS working entirely in polynomial notation. Not surprisingly, none made it through without an error. This was going so far out of their way to make the problem harder that I felt I just had to take points.

4. Consider the following snippet of C code, to be executed on 2 processors. x is a shared variable, and the system uses serial consistency.

P1		P2
x = 1;		x = 2;
x = x + 2;		x = x + 1;

In answering the following questions, use the notation for memory accesses we've been using throughout the semester.

- (a) Show how this code can produce a sequence of memory reads and writes such that x can end up with a value of 2.

P1:	w(x)1	r(x)1	w(x)3
P2:	w(x)2	r(x)1	w(x)2

- (b) Show how it can produce a sequence of memory reads and writes such that x can end up with a value of 4.

P1:	w(x)1	r(x)2	w(x)4
P2:	w(x)2	r(x)2	w(x)3

- (c) Show how a sync point S can be inserted in the code which will guarantee that x will end up with a value of 3.

If the code is changed to

P1		P2
S;		
x = 1;		x = 2;
x = x + 2;		x = x + 1;
		S;

then the memory actions become

P1:		S	w(x)1	r(x)1	w(x)3
P2:	w(x)2	r(x)2	w(x)3	S	

- 3 processor writes wrong value
- 5 accesses put out of order
- 7 missing accesses
- 5 no sync point forcing result
- 3 Doesn't show sync point works
- 3 sync in wrong place
- 7 claims impossible
- 5 didn't use notation
- 2 extra accesses

Probably the biggest surprise in grading this one was the number of people who seem to be under the impression that "x = x + 1;" is an atomic operation. One particular class of wrong solutions to part 4c looked basically like this:

P1:	w(x)1	r(x)1	S		w(x)3
P2:		w(x)2	S	r(x)2	w(x)3

The thing is, with the sync point there, the sequence of memory operations could just as easily have wound up as

P1:	w(x)1	r(x)2	S		w(x)4
P2:	w(x)2	S	r(x)2	w(x)3	

so it didn't force the result to be 3 (there are quite a few variations here, but the point is that it doesn't **guarantee** the reads and writes must occur in an order that will give the required answer).

5. Some of the relevant characteristics of the Massively Parallel Processor are:

- Operations can be performed on all of the processing elements simultaneously, or can be "masked" by a special bit plane called the G plane. A masked operation is only performed by PEs whose element of the G plane has a value of 1 (of course, they are still performed simultaneously for all enabled PEs).
- It is possible to perform Boolean operations involving the G plane. In particular, it is possible to set elements of the G plane to 0 when an arbitrary condition is satisfied for some other plane.
- There is a one-bit output available called sum_or. When an operation is performed in the machine, this bit takes a value of 1 if the result of the operation is 1 on any PE in the machine.

So... it turns out MPP is able to find the largest element of an array in time linear in the number of bits in the values in the array. Describe, using either pseudocode or clearly in English, how this can be performed. *Hint: start with the most significant bit of the array, and see if any of the PEs have a 1 in that bit. Remember that once a PE has been established as not containing the greatest value, it doesn't ever need to be considered again.*

Assume the array is in a set of N planes called array[N], and that we have access to the G plane and sum-or tree output. We can solve the problem in C-like code as:

```
G = 1;
for (i = N-1; i >= 0; i--) {
  where G {
    if (sum_or(array[i]) == 1) {
      val[i] = 1;
      G = array[i];
    } else
      val[i] = 0;
  }
}
```

Of course, other forms of pseudocode are also OK.

6. Draw a 9×9 Benes network using 3×3 crossbars switches. Since I've put a figure showing the switches on this page, that might be a good starting point.

