

**CS 573**  
**Midterm Exam**  
**March 9, 2007**

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 Banner ID 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 you arrived at your answers
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 11:20 to finish the exam. The questions are equally weighted.

1. You have a choice of making exactly one of the following changes to the microarchitecture from HW2.
  - (a) Double the size of the prefetch path: change the size of the prefetch buffer to 16 bytes, and assume you can fetch 8 bytes at a time. The instruction decoders are unchanged.
  - (b) Improve the decoders so you can always decode two instructions per cycle (rather than one in some cases and two in others, as in the HW).
  - (c) Double the number of  $\mu$ -instruction slots in the decoded instruction buffer.
  - (d) Double the number of physical registers.
  - (e) Double the number of arithmetic pipelines.

In your estimation, which would show the greatest promise for improving the performance of that machine? In this question, your actual answer is almost completely unimportant: what matters is how you think about the problem and *why* you make the choice you do.

*Let's look at our choices one at a time:*

- (a) *Double the size and speed of the prefetch buffer.*  
*This won't make the slightest difference. We can already prefetch instructions faster than we can decode them: the only case in which we can decode two instructions is if the first one is only one byte, while the maximum size of an instruction is three bytes. So we can never need to decode more than four bytes on a cycle.*
- (b) *Improve the instruction decoders*  
*This looks a little more promising. Notice that we can execute three  $\mu$ -ops per cycle; this is close to the amount we can ever hope to create in a cycle. It's likely that trying to refill the pipelines after events like branches would be really slow. But, as we'll see in a moment, that's really not the biggest problem.*
- (c) *Double the number of  $\mu$ -slots.*  
*Instruction decode stalled a lot, but never for lack of space in here.*
- (d) *Double the number of physical registers.*  
*In my solution to the HW assignment, I was very surprised to discover the extent to which the number of physical registers was the bottleneck, though after a little reflection I saw I should have anticipated it. Basically, we've always got at four physical registers tied up in mappings for logical registers. Every instruction in flight will cost at least one more, the result registers need to be allocated while the old registers need to be preserved for interrupt recovery until the instruction is retired. Finally, we're likely to need some temporaries.*  
*Anyway, in my solution to the problem, I spent most of my time waiting for a register to come free so I could reallocate it. I never found myself with nothing to decode, nor with no space in the  $\mu$ -op buffer, nor with a conflict in the pipelines. It was all right here.*

(e) *Double the number of arithmetic pipelines.*

*Again, I never had to wait for one, so having more of them doesn't seem very helpful.*

*I was pretty surprised to see the answer to this in my solution to the HW: the number of physical registers is a horrible bottleneck, and my solution spends most of its time with decoding stalled because there are no available physical registers to allocate.*

*In grading the problem, I want to see answers that acknowledge the problem as one of balance; that try to address a perceived bottleneck (without grading your HW at the same time, I really can't tell whether you're seeing the actual bottleneck!).*

2. The VAX instruction set was no fun to decode: the first byte contained the opcode and the following (up to) three bytes contained a register number and addressing mode specification. Then, any values (such as immediate operands, direct memory addresses, and indexes for indexed addressing) needed for the addressing modes came next.

But this wasn't as bad as it could have been: each register/mode byte could have been followed immediately by its data. Why would this have been even worse for instruction decoding than the actual VAX? Would it have affected the orthogonality, as seen by the programmer?

*It would have been a mess. In the real VAX, once you've figured out where an instruction starts you can decode the opcode and all the reg/mode bytes in parallel (you can even start the decode of the second and third reg/mode bytes before you know you've got a three-operand instruction – just throw away the ones you don't need). Interleaving would mean you couldn't figure out where the second operand started until you'd decoded the first, so you'd be pretty much forced to decode them in sequence.*

*This would actually have made the machine code look more like the assembly code – existing assembly languages already interleave the modes and operands, however they're really arranged in the machine code. So it wouldn't have made any difference here at all.*

3. Suppose the program in HW 2 had executed its loop many more times than it actually did.

(a) What would be the asymptotic behavior of the branch predictor in the assignment? That is, over time would they converge on some good or bad behavior? If they did, what "ultimate" prediction accuracy would you expect from each of them?

*The first branch takes exactly every other branch. So it ends up oscillating between two patterns in its PBHT: 0101 and 1010. It will only index those two entries in the GPHT; every time it indexes 1010 the branch is taken, every time it indexes 0101 the branch is not taken.*

*The second branch is equally predictable, taking two and then not taking two forever. So it ends up cycling through four patterns in its PBHT: 0011, 0110, 1100, 1001. It only uses those entries in the GPHT; in its case, when the pattern is 0011 or 0110 the branch is not taken; when it is 1001 or 1100 it is taken.*

*Finally, the branch controlling the loop itself only puts pattern 1111 in its PBHT, and always takes the branch until the last time through the loop*

*Notice that the three branches use a total of seven different indexes in the GPHT, and no two branches share an index. This says that the GPHT entries will all go to 0 or 3, and will predict correctly forever. So the asymptotic behavior of all three branches is 100%.*

(b) Suppose that instead of the branch predictor from the assignment, we use Smith's Strategy 2 (each branch instruction has a simple one-bit predictor so it always predicts every branch would be taken if the previous execution of that same branch was taken). Now what happens to the asymptotic behavior?

*For B1, this is a disaster. This instruction alternates between taken and not-taken; as this Strategy 2 always predicts the next execution will be the same as the last one it will always mispredict. Accuracy 0%.*

*For B2, it's pretty bad. As it takes its branch twice, then doesn't-take twice, and so forth, it will mispredict the first branch in every pair and correctly predict the second. Accuracy 50%.*

*For B3, things work well. We make every prediction except the last one correctly, for an asymptotic accuracy of 100%*

(c) Finally, suppose that instead of the predictor from the assignment, each branch instruction had a simple two-bit predictor using saturating counters (an idealized version of Smith's Strategy 7). Now what happens to the asymptotic behavior?

*For B1, when the first branch is taken the counter is driven to "3". It then oscillates back and forth between 2 and 3, so it always predicts taken; the result is an accuracy of 50%*

*For B2, after two taken branches the counter will contain a 3. The next two iterations the branch won't be taken; it'll mispredict both of these (leaving a counter value of 2, then a value of 1); now it'll take two branches, one of which is a mispredict and the second is a correct predict. This sequence of four events will repeat forever, for an accuracy of 25%.*

*Once again, B3 is 100%, for the same reasons as before.*