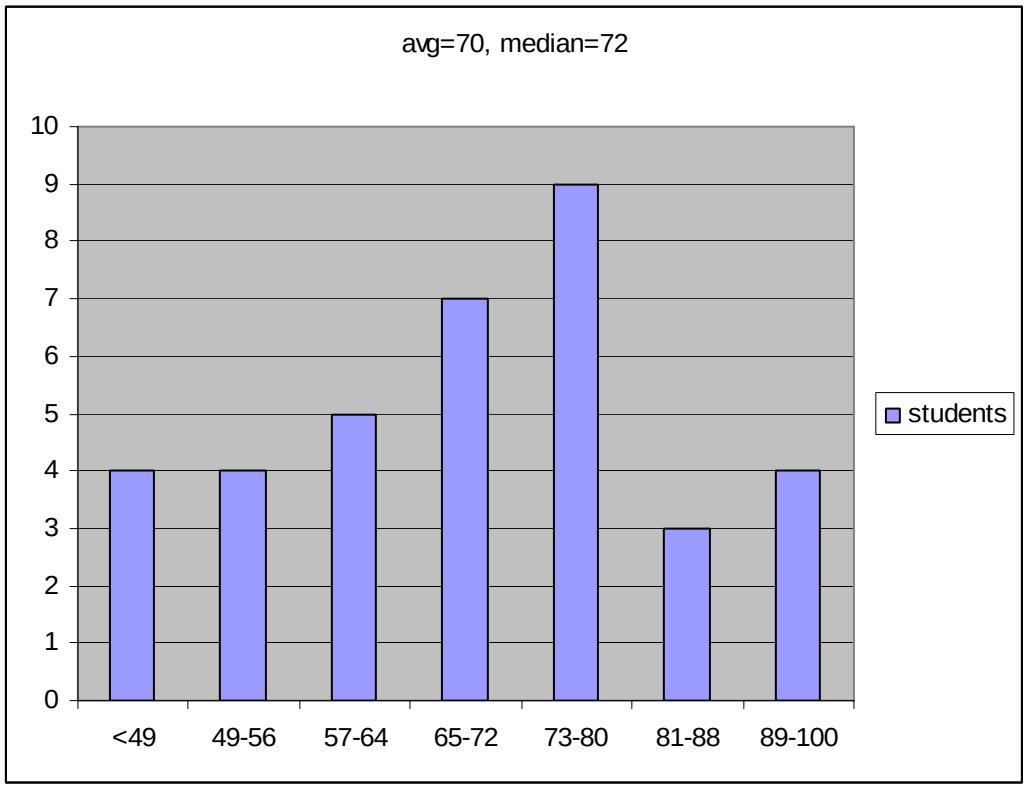
ECE 511, Computer Architecture

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
M. Frank

Midterm Exam: MY ANSWERS

October 11, 2006
6:00-9:00pm

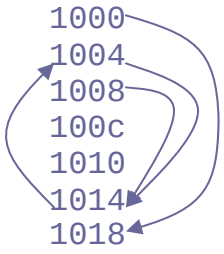
You may use class notes, textbooks, web resources, computer simulations or any other reference material you desire. No interactions with others allowed. By turning in this exam you will have attested that you have neither received nor given inappropriate aid on this quiz (i.e., interacted with any person other than the instructor.)



1. (10 pts) Consider the following pseudo-code snippet:
(address in hexadecimal, followed by pseudo assembly code)

```
1000: BRANCH <condA> 1018
1004: BRANCH <condB> 1014
1008: BRANCH <condC> 1014
100C: ADD
1010: MUL
1014: BRANCH <condE> 1004
1018: SUB
```

a) Suppose we are using a gshare predictor with a history of length 2 (just to keep the example simple). Suppose also that we have a 32 entry table of counters. Our gshare predictor hashes into this table by taking bits [6:2] of the current PC, and xoring with the current 2-bit history. Construct two paths through the code, ending at different branch instructions, but that cause the predictor to hash to the same table entry. (This is probably easier if you draw the code out as a flow graph.) **Clarification: history register left shifts, the older history bit gets xored with PC[3], the newer history bit gets xored with PC[2].**



1004 taken to 1014 taken to 1004
1000 not taken to 1004 not taken to 1008

both paths hash to table entry 2.

b) Construct two distinct legal paths through the code, both ending at the BRANCH at address 1014, but that produce the *same* 2 bit global history.

1000 not taken to 1004, 1004 taken to 1014
1004 not taken to 1008, 1008 taken to 1014

2. (20 pts) You are designing the new Illin 511 processor, a 2-wide in-order pipeline that will run at 2GHz. The pipeline front end (address generation, fetch, decode, and in-order issue) is 14 cycles deep. Branch instructions then take 6 cycles to execute in the back end, load and floating-point instructions take 8 cycles, and integer ALU operations take 4 cycles to execute.

a) Your lab partner has written some excellent assembly code that would be able to achieve a sustained throughput on the Illin 511 of 4 billion instructions/second, as long as no branches mispredict. Assume that an average of 1 out of 10 instructions is a branch, and that branches are correctly predicted at a rate of p. Give an expression for the average sustained throughput in terms of p.

Expected instructions to next mispredict is $e = 10/(1-p)$. Number of instruction slots (two slots/cycle) to execute all those and then get the next goodpath instruction to the issue unit is $t = e + 2 \cdot 20 = \frac{10}{1-p} + 40 = \frac{50-40p}{1-p}$. So the utilization (rate at which we execute useful instructions) is $e/t = \frac{10}{50-40p}$.

b) Unfortunately, it turns out that there was a bug in the original code. The bug fix made the code somewhat larger, and now you are observing instruction cache misses in the 2Kbyte L1 instruction cache. Each L1 cache miss causes a 10 cycle bubble to be inserted in the instruction stream (while we fetch the cache line from the L2). If you could only double the size of the L1 cache you could completely eliminate the L1 cache misses. Unfortunately building a 4Kbyte fully pipelined 2GHz L1 cache would add an extra 5 cycles to the front end. Under what circumstances (e.g. cache miss rate and branch prediction rate) would it be a good or bad idea to change the pipeline to include the larger cache?

If we increase the cache size the branch mispredict penalty increases from 20 cycles to 25 cycles so, by part (a), utilization would be $\frac{10}{60-50p}$. If we keep the i-cache small, it will have a non-zero miss-rate, m. We still fetch e instructions between mispredicts, but now of those e, em are 10 cycle (=20 slot) L1 misses. So the number of instruction slots to execute our instructions is $t = e(1+20m) + 2 \cdot 20 = \left(\frac{10}{1-p}\right)(1+20m) + 40 = \frac{50+200m-40p}{1-p}$. Thus utilization will be $\frac{10}{50+200m-40p}$. The larger cache will provide better utilization when $\frac{10}{60-50p} > \frac{10}{50+200m-40p}$, or $50+200m-40p > 60-50p$, or $p > 1-20m$.

3. (15 pts) I mentioned, in passing, that the return address stack predictor needs to be carefully repaired after a mispredict. Assume we are working with an architecture where it is "easy" to identify call and return instructions (as distinct from other branch instructions or non-branch instructions). Assume also that the BTB always hits and provides accurate information about the instruction being fetched.

The return address stack (RAS) is a small structure that contains an array of (say) 8 addresses and a "head pointer" that points to one entry of the array. Each time we speculatively fetch a call instruction we push the expected return point of the call (PC+4) onto the RAS. We do this by incrementing the head pointer (mod 8) and writing the return address at the corresponding RAS array location. Each time we speculatively fetch a return instruction we predict that the nextPC will be the PC currently pointed to by the head pointer. Then we decrement the head pointer (mod 8).

a) First, assume that we don't do any "repair" of the RAS after a branch mispredict. Give a (short) sequence of instructions and events that will cause the RAS to "get out of sync" (i.e., mispredict all the return addresses currently listed in the RAS).

call A
branch mispredict
call B (bumps the RAS pointer)
rollback to the branch and now no matter what we do every return will get its target from the RAS entry one above where it should be looking

b) Now, assume that we "repair" the state of the RAS as follows: with each branch instruction we carry down the pipeline the value of the RAS head pointer, as it was when the branch was fetched. (Much the same as we do with the global history). When a branch misprediction is detected, we reset the RAS head pointer to the position we carried with the mispredicted branch. Give a (short) sequence of instructions and events that will cause at least the *next* return address to be mispredicted, but not the contents of the rest of the stack.

A: call (headpointer <= 1, write value A+4 in RAS[1])
Fetch Branch that will mispredict (record headpointer = 1)
Return (headpointer <= 0)
B: call (headpointer <= 1, and we overwrite RAS[1] <= B+4)
Rollback to the branch that mispredicted. This will restore the head pointer <= 1, but RAS[1] = B+4, not A+4. Thus the next return will try to jump to B+4 and end up mispredicted.

4. (20 pts) You are designing the register renaming circuit for the new OOPS Corp. processor (I guess OOPS must be an acronym for "Out-of-Order Processing Systems"). The OOPS is a 2-wide out-of-order superscalar. The OOPS ISA is much like MIPS or DLX: there are 32 architectural registers, register 0 always contains the value 0, instructions have at most two source operands and at most one destination operand. If an instruction does not have a destination operand the decoder will tell the renamer that the instruction's "destination" is register 0.

The renamer needs to rename 2 instructions per cycle. Thus, we need to be able to simultaneously perform 4 reads on the RAT and perform 2 writes to the RAT. Note that the renamer is implemented with flip-flops, so we won't see any writes to the RAT until the end of the current cycle.

Give pseudo code for the state transitions that need to take place. Call the two input instructions in-instA, in-instB. (A comes before B in architectural order). They have fields archD, archS, archT. The two output instructions are out-instA and out-instB. They have fields physD, physS, physT. The RAT is a 32 element array. You may indicate the free list by <another free reg>. (You don't need to describe the transitions on the free list).

To get you started, here are the transition for out-instA:

```
out-instA.physS <= RAT[in-instA.archS]
out-instA.physT <= RAT[in-instA.archT]
tempAphysD = if (in-instA.archD != 0)
                then <another free reg>
                else 0
out-instA.physD <= tempAPhysD

out-instB.physS <= if (in-instB.archS == in-instA.archD)
                      then tempAphysD
                      else RAT[in-instB.archS]
out-instB.physT <= if (in-instB.archT == in-instA.archD)
                      then tempAphysD
                      else RAT[in-instB.archT]
tempBphysD = if (in-instB.archD != 0)
                then <another free reg>
                else 0
out-instB.physD <= tempBphysD
RAT[in-instB.archD] <= tempBphysD
if (in-instA.archD != in-instB.archD)
   then RAT[in-instA.archD] <= tempAphysD
```

5. (10 pts) You are designing a new out-of-order microcontroller. You have decided that the scheduler will be of size W, and the reorder buffer will be of size R. The instruction set specifies A architectural registers. Every instruction has at most two source operand registers and at most one destination operand register.

a) What is the size of the physical register file you should use if you want to *guarantee* the renamer will never stall because the free list is empty? (Don't worry about the extra register or two that you would need to keep to account for latch delay) Give an example of a sequence of instruction dispatches that would require you to allocate this many registers (assuming that W = 3, A = 4, and R = 5).

A+R.
Example: we would need 9 registers. Suppose we start with the machine empty, the RAT and RRAT each with 4 entries:
1: p1
2: p2
3: p3
4: p4
and the freelist with 5 more regs: p5, p6, p7, p8, p9.
Now we rename and dispatch 5 instructions, each with arch reg 4 as dest (but none of the instructions has retired yet.
Now the RRAT hasn't yet changed state (so p1, p2, p3, p4 are all still mapped), p5, p6, p7, p8, p9 are allocated to the 5 instructions in the ROB and the RAT has the state:
1: p1
2: p2
3: p3
4: p9

b) Explain why you should not actually make the physical register file as big as your calculation in part (a) suggests.

Neither store instructions nor branches have a dest operand. A reasonable assumption might be that between 25% and 40% of instrs are branches and stores, so even if the ROB were full we might be able to get by with A + .75R physical regs.

6. (10 pts) You are designing the issue unit for a 1-wide pipeline with *in-order* issue and out-of-order completion. As in the "standard" pipeline we've been using in class, once an instruction issues it reads its source register values on the first cycle and then proceeds to the ALU on the following cycle. Branch instructions take 6 cycles to execute in the back end, load and floating-point instructions take 8 cycles, and integer ALU operations take 4 cycles to execute (including the cycle to read the registers). Instructions write to the register file on the cycle they complete. Branch mispredictions are detected 6 cycles after the branch issues. On a mispredict all preceding pipeline stages are flushed. Luckily, none of the instructions can ever take an exception. (So you don't need to support precise exceptions).

a) One of the rules that the instruction issue unit must follow is: *an instruction must wait at the issue unit for its source operands to be written/completed before the instruction can issue*. Since this is an in-order processor the instruction issue unit also follows the rule: *an instruction must wait at the issue unit for all preceding instructions to issue*. Give the rest of the rules that the issue unit must follow.

- An instruction must wait for its dest operand to be completed before issue. (Otherwise we might have a problem with a load followed by an ALU to the same destination).
- Any integer ALU instruction immediately following a branch must stall at least one cycle before issuing (after that the branch will flush the instr before it overwrites data).

b) Describe *two* ways in which implementing renaming could improve this designs performance (even if we keep it an in-order issue design).

We get to eliminate both the extra rules given in part (a). Eliminating the bubble between branches and ALU instrs will probably help a lot. Eliminating output-deps probably won't help matter so much (output deps that aren't also a true-dep followed by an anti-dep are rare, and in-order issue enforces both true-deps and anti-deps.) The real second improvement might come because the issue logic is simpler, thus less likely to impact the critical path.

7. (15 pts) In class I described the operation of the load-queue as follows: when a load is dispatched from the renamer to the ROB and scheduler it allocates the tail entry in the load queue. When the load is issued and executed it writes the address of the memory location that it read into its entry in the load queue. When the load retires, the head pointer of the load queue should be pointing to the load instruction's allocated entry. That entry is deallocated by incrementing the head pointer.

As a store instruction dispatches from the renamer to the ROB and scheduler it records the current load queue tail along with the store instruction. Let us call this value of the tail pointer "start". When the store issues and executes it calculates the address of the memory location that it wishes to store its value to. Then the store searches the load queue from the "start" pointer location to the current tail. If any of the addresses in that range match the store's address, then we mark the corresponding load instruction as having misspeculated.

a) Suppose the following three instructions are dispatched (in the order A, B, C):

A: store r5 -> M[r6]
B: store r7 -> M[r8]
C: load M[r9] -> r10

Give a sequence of events where load C actually loads the correct value, produced by the correct store instruction, but ends up getting marked as misspeculated.

r6, r8 and r9 all contain the same pointer value. Then store B issues, followed by load C, followed by store A. load C gets the correct value (whatever store B wrote), but store A thinks load C issued early, so load C gets needlessly cancelled and reexecuted.

b) Your lab partner, Ben Bitdiddle is designing a new microprocessor that he claims is "complexity effective." The processor is designed to run at very high clock rates. Ben is worried that there is no way he could search a store queue in a single cycle, so he's come up with a "great" idea for building a processor that doesn't have a store queue. The idea is that we keep the load queue (it's okay to spend a few extra cycles searching the load queue because no instructions are really waiting for the load queue search to finish), and only issue stores when they reach the head of the ROB. Since stores don't issue until they retire, they never issue speculatively and can, therefore, write directly to memory. Ben reasons that since we still have a load queue we can still aggressively and speculatively issue load instructions, so delaying the store issue to retirement shouldn't hurt. Give a short example code sequence that will do extremely poorly under Ben's scheme.

Clarifications: it's sort of an idea for *really* issuing loads and stores out-of-order with respect to one another. Stores issue as late as possible (when they are about to retire). Loads issue as early as possible (whenever their source register is ready). In terms of "legality" Ben's idea works because as stores retire we still check the load-queue to see whether any of the loads issued too early. If they did we squash them and restart fetch at the offending instructions. So what I'm really asking for is a short sequence of instructions where delaying the stores that long will cause a performance problem.

Load miss A, followed by a store that could have issued anyway, followed by load B, dependent on the store. The load B will issue, load A will stick at the head of the ROB for a very long time. Eventually load A retires. Finally store retires, looks up in load queue and cancels load B. What a waste, we could have forwarded data (or at least cancelled and refetched) to load B dozens of cycles earlier.