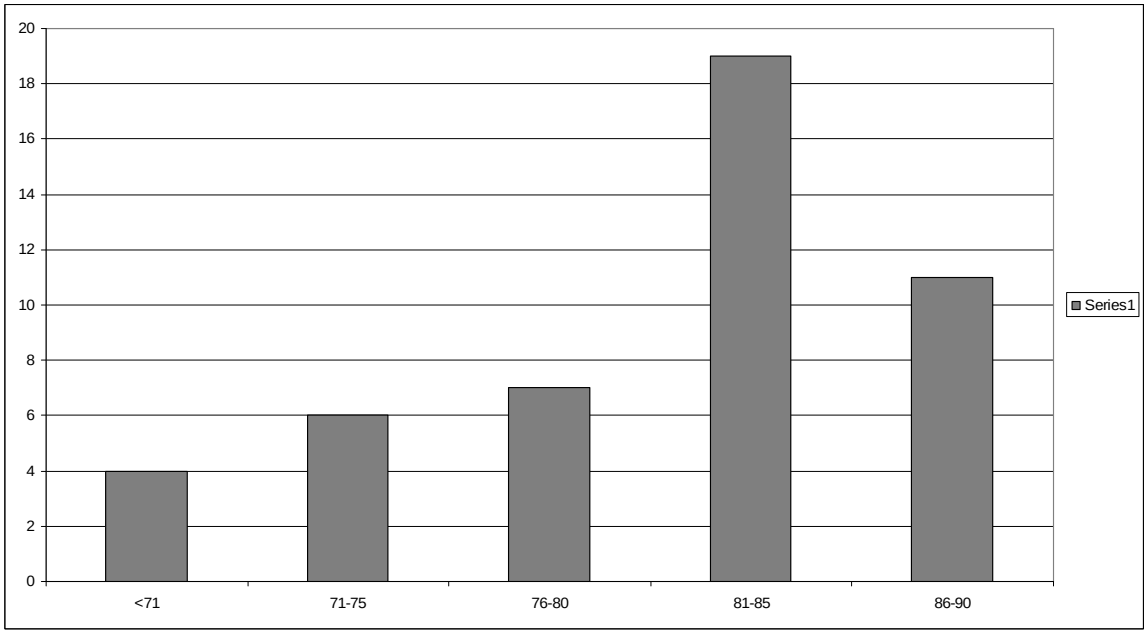
University of Illinois at Urbana-Champaign
Department of Electrical and Computer Engineering

Quiz 1
ECE 511, Fall 2004

You have until 6pm Friday, October 8th to complete this quiz. By turning in this quiz, you will have attested that you have neither received nor given inappropriate aid on this quiz from any person except the instructor. You may use class notes, textbooks, web resources, computer simulations, and published materials.

NAME: Answer Key

Histogram is below: Because question 2 was so bad I removed each person's lowest score of the six questions (which was question 2 in about 75% of cases), and so the curve is out of 90. The test was more difficult than I intended. I'd say an 80/90 is an excellent score.



1. (10 pts) The following figure describes the data cache system for a particular microprocessor. We know that the processor consists of a simple in-order pipeline that blocks on L1 cache misses. We know very little about the caches of this system except that the L1 cache, if it is associative, uses the Least Recently Used (LRU) replacement policy. We also know that the latency between the L1 and L2 cache is *p* cycles.
In order to discover the size and organization of the L1 data cache, we have decided to run a small code kernel on it, which is described in pseudo code below. The code sequentially accesses D elements of an array called A repeatedly 100 times.

```
do 100 times {
     for i = 0 to D-1 {
          access A[i];
     }
}
```

(a) Given line size w, when D < r we take D/w cold misses and all the rest of the accesses are hits. Thus

$$1 = \frac{\lceil D/w \rceil p + 100 D l_h}{100D} = l_h + \frac{p}{100w} \approx l_h$$

(b) Size is r

(c) When D > s, we get a capacity miss on every line, followed by hits for the remaining words on the line. Thus

$q = \frac{p + w l_h}{w}$, so $q = p/w + 1$, so $w = \frac{p}{q-1}$.

(d) The number of the lines in the cache is $n = r/w$. With a k-way set associative cache there are $t = n/k$ sets, and $r = tkw$. At the point where D=r each set has exactly k lines in it. At the point where D=s each set has at least k+1 lines competing for it (and thus every access to a line causes a miss because the set is replaced using LRU policy). So

$s = t(k+1)w = tkw + tw = r + tw = r + \frac{r}{k}$ and $k = \frac{r}{s-r}$.

2. (10 pts.) In class on September 13 I said that if the BTB has a miss but the branch predictor predicts taken, you have no choice but to fetch PC+4, but that the branch predictor was probably correct and the BTB incorrect. Luckily you all ignored my suggestion when you did Homework 2. Explain why I was wrong.

A BTB miss means either (a) the instruction is not a branch (in which case the branch pred *never* contained any info about the instruction) or (b) the instruction is a branch, but is a cold miss (in which case the branch pred is also going to be cold with respect to this branch) or (c) the instruction is a branch that got kicked out of the BTB for capacity/conflict reasons, the bpred has many more entries than the BTB, so the bpred is somewhat more likely to be right and the BTB wrong. Nonetheless, case (a) is much more likely than (c) because branches are usually less than 20% of the instruction mix.

3. (20 pts.) You are designing the address generation unit for a new processor, and your lab partner, Ed Davidson, suggests that rather than updating the BTB's LRU information each time a branch is found in the BTB we should only mark a branch entry as ~~Least~~Most Recently Used in the BTB when the branch retires and is seen to be taken. What are the relative advantages and disadvantages of this idea?

Let's break this down into two parts: First, updating the LRU bits at retirement time rather than fetch time avoids updating the LRU bits for entries that are miss-speculatively fetched. The added complexity is that the BTB hash and way information has to be carried through the pipeline which adds a few extra transistors here and there. Second, updating the LRU only on taken matters only if there are k+1 or more branches competing for the same set in a k-way set associative BTB. The new rule means the k most recently *taken* branches will be in the BTB, the old rule means the k most recently *seen* branches will be in the BTB. I would guess that it is more useful to have the k most recently *taken* branches, but there are almost certainly cases where having the k most recently seen is better.

4.(20 pts.) You are working on a design with a single cycle direct mapped instruction cache, but where the instruction cache miss penalty averages 25 cycles. You have discovered that your direct mapped instruction cache has a pretty bad miss rate. Your lab partner, Maurice Wilkes, suggests that you could replace the direct mapped cache with a *pseudo*-associative cache: instead of hashing the PC to find a cache line and then comparing tags, you could store a particular block of instructions in *any* of the cache lines (as with a fully associative cache), but then in the BTB you would store a prediction of the corresponding cache line number along with the target address for each branch. What are the relative advantages and disadvantages of this idea?

Advantages: Will probably reduce conflict misses. Disadvantages: (a) increases miss penalty (because we have to do a secondary lookup in the cache before we know we missed), (b) added complexity: extra storage bits in the BTB, path to update the BTB from the fetch unit and secondary lookup circuitry in the cache. (c) What do we do with non-branch instructions??? With a little extra complexity we can just keep using the same way-number as the last branch instruction, but what happens when we PC+4 off the end of a cache line? Now we have to keep BTB entries representing "non-branches-at-the-end-of-a-cache-line," and that's going to make the BTB less effective.

5. (20 pts.) You are the chief architect at Illin Corp, the makers of a processor (called the Illin E511) that issues instructions out-of-order, but does no register renaming. The scoreboard enforces true- (RAW), anti- (WAR) and output- (WAW) dependences by using the following three rules: An instruction is ready to issue only if (1) All previous instructions that write its source registers have completed, (2) All previous instructions that write its destination register have completed, (3) All previous instructions that read its destination register have issued.

A young summer intern in your group named Seymour Cray points out that rule (2) is overly conservative. The instruction in question doesn't really need to wait until the previous writer of its destination completes, it just needs to wait long enough to ensure that the writes happen in the correct order. For example, if multiplies take 10 cycles and adds take 5 cycles then in the instruction sequence:

ADD R13 <= R10 + R2
MUL R13 <= R1 * R5

It would really be okay to issue the add and then issue the multiply in the very next cycle. The add will write its result at the end of the sixth cycle, the multiply will write its result at the end of the twelfth cycle.
Would it be a good idea to use Seymour's new rule or a bad idea? What extra information would you need to keep in the scoreboard? Are output- dependences a cause of poor performance on your machine (explain)?

Bad idea. The scoreboard gets more complex because now you need to figure out the time difference in cycles between the write back time of an already issued instruction and an instruction you are considering issuing. Also, for load instructions we are not sure when they are going to complete, so we can only approximate the time difference.

So much for the cost, what's the potential benefit? Is Seymour's example a reasonable one? Not really: more likely the programmer (or at least the compiler) would remove the dead add instruction from the instruction stream. Usually there will be a use of R13 between the add and the multiply, so there will be a true dependence from the add to the use and an antidependence from the use to the multiply, so the multiply won't issue any earlier anyway. The other possibility is that there is a branch between the add and the mul, but we can't really speculate across branches in our machine, so this isn't likely to matter much either from a performance perspective.

6. (20 pts) Consider all the places that a physical register tag could be stored in the MIPS R1000 like pipeline we have been considering in class. In particular, a physical register tag could be listed (1) in the register free-list (FL), (2) in the physical destination register field of one entry in the reorder buffer (ROB), (3) in the register alias table (RAT) or (4) in the retirement register alias table (RRAT). What are the possible legal combinations of places that a physical register tag could be stored, and under what conditions will the register tag transition from one set of places to another. (You may want to start out by considering the following questions: Is it possible that a physical register tag could be stored in *none* of these four places? Is it possible that a physical register tag could be stored in both the RRAT and the ROB? Is it possible that a physical register tag could be stored in both the RAT and the RRAT?)