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

Quiz 2
ECE 511, Fall 2004

You have until 6pm Saturday, December 18 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: ______________________________

1a. (10 pts) The MIPS R10000 renaming scheme that we discussed in class has a problem. As you discovered on homework 3, it is often the case that we discover a misprediction for some branch deep in the reorder buffer, but because of the way we cancel instructions, we don't tell the front end of the pipeline (the address generator and renamer) about the misprediction until the branch in question retires. Thus, as we make the reorder buffer longer, performance can actually decrease. The Pentium 4 attempts to solve this problem by telling the address generator about the misprediction as soon as the mispredicting branch completes. Unfortunately, on the Pentium 4 the renamer still can not start renaming down the correct path until the branch that mispredicted retires. Explain why. (Give an example instruction sequence where the rename Register Alias Table will be in an inconsistent state until the mispredicted branch retires.)

1b. (10 pts) The Alpha 21264 renamer *can* start renaming down the correct path as soon as a mispredicted branch completes (rather than waiting for the branch to retire) because in the 21264 every branch is given a copy of the register alias table. This means that instead of just two register alias tables (one representing the speculative state in the renamer and one representing the architectural state at retirement) the 21264 contains *many* register alias tables. As each branch passes the renamer it receives a copy of the speculative RAT at that point. If a branch completes and is mispredicted the address generator is signaled to send it down the now correctly determined fetch path, and the copy of the rename table from the mispredicted branch is copied over the rename RAT so that the renamer can go ahead and consistently rename the correct path instructions. Suppose that the reorder buffer on the 21264 has 128 entries and that extensive measurements have shown that in the Alpha ISA each basic block has an average length of 8 instructions. Thus you expect, on average, 16 branches to be in the reorder buffer, if it is full. You also know that the branch predictor on the 21264 is better than 95% accurate on average. Suppose you also know that the scoreboard has 16 entries, and that extensive measurements have shown that there are rarely more than 4 branches that have dispatched but not yet completed. How many rename alias tables would you suggest the designers of the 21264 implement? (Please explain your answer, don't just give a number. Think about the "lifetime" of a register alias table copy. Each copy gets made as a branch passes through the renamer. What's the earliest time at which it is no longer useful to hold a register alias table copy? Do we have to wait until the branch retires, or can we reuse the register alias table sooner?)

2a. (10 pts) The Store Set Prediction paper by Chrysos & Emer describes a situation with the `applu` benchmark where store set prediction is more conservative than one would like (see pages 153 and 154 of the Chrysos & Emer paper). Suppose you are designing the store buffer for a processor with out-of-order load stores, a load queue and a store set predictor. Your lab partner, Josh Fisher, argues that since the store set predictor is conservative anyway, you might as well optimize the store buffer in the following way: Rather than making the store buffer an associative cache tagged by address, it should just be a store-buffer-*table* of address/value pairs, indexed by store set number. When a store with store set number *n* is issued to the execute unit, the address and data of the store are written to entry number *n* in the table. When a load with store set number *n* is issued to the execute unit, store-buffer-table entry *n* is looked up. If the address matches the data value in the table is bypassed to the load, otherwise the load gets its value from the memory system. Josh reasons that since your system has a load queue to check whether loads and stores have incorrectly issued out-of-order, and that the memory dependence predictor will keep the loads and stores in order anyway, that his scheme will work. Give Josh an example instruction sequence where his store-buffer-table will cause loads to get the wrong value, even if the stores and loads are issued *in* order.

2b. (10 pts) Suppose we implement a load queue as a hash table with 256 entries as described in class. The address of each load and store instruction, as it completes, is hashed to the table. Suppose that your hash function is excellent (so that it is reasonable to assume that a sequence of hash bucket choices is essentially random), and that you are dealing with an instruction sequence where none of the addresses are the same. If the load queue starts out empty (e.g. we've just retired a branch mispredict), what is the probability that if n memory instructions have completed without yet retiring that all n instructions have addresses that map to different buckets? (Hint: if there is only one memory instruction that has completed but not retired, then the probability is 1, if there are two memory instructions completed but not retired the probability is 255/256, if there are three memory instructions completed but not retired the probability is (255/256)(254/256), and so on.) Find the number, *n*, of independent memory instructions such that the probability that all *n* instructions map to different buckets is less than 50%. (This is called the Birthday Paradox).

3. (15 pts.) Suppose you have a dynamically optimizing processor with a frame cache. The following sequence of instructions is in the frame cache. Determine the set of instructions that can be eliminated from the frame by the dynamic optimizer. For each instruction that you eliminate, explain why it is legal to eliminate it.

```
[beginning of frame]
Ra <- Rx + Ry
Rb <- Ra + Rz
Rc <- Rc + Rb
Rd <- Rc + Rw
Ri <- Ri + 1
Assert Ri < Rn
Ra <- Rx + Ry
Rb <- Ra + Rz
Rc <- Rc + Rb
Rd <- Rc + Rw
Ri <- Ri + 1
Assert Ri < Rn
Ra <- Rx + Ry
Rb <- Ra + Rz
Rc <- Rc + Rb
Rd <- Rc + Rw
Ri <- Ri + 1
Branch Ri < Rn target
[end of frame]
```

4. (15 pts.) You are profiling the newly release SPEC 2004 benchmark suite on your Pentium4 based Linux workstation, and discover that one of the new benchmarks spends 80% of its execution time in a procedure called "`sum_array`" (shown below):

```
int sum_array(int *a, int length) {
  int j;
  int sum = 0;
  for (j = 0; j < length; j++) {
    sum += a[j];
  }
  return sum;
}
```

Your lab partner suggests that the routine could be rewritten to get 4 times better ILP as follows:

```
int sum_array(int *a, int length) {
  int j;
  int sum0 = 0;
  int sum1 = 0;
  int sum2 = 0;
  int sum3 = 0;
  for (j = 0; (j+3) < length; j+=4) {
    sum0 += a[j];
    sum1 += a[j+1];
    sum2 += a[j+2];
    sum3 += a[j+3];
  }
  while (j < length) {
    sum0 += a[j];
    j++;
  }
  return sum0 + sum1 + sum2 + sum3;
}
```

Under what circumstances is this idea going to succeed or fail?

5. (15 pts.) You've just finished building the new Illin 512 out-of-order superscalar processor. The Illin 512 has a store buffer, but issues loads and stores in order, so no load queue was necessary to check whether loads and stores might have issued out of order. Unfortunately, the processor design team forgot that the Illin 512 is supposed to work in a multiprocessor setting, and while the cache design team has implemented a full blown MESI cache coherence protocol, there is no way for the cache to inform the processor core about coherence actions. Give an example sequence of load and store completions and retirements that will cause the following code to appear sequentially *inconsistent* when run on a system with two processors:

Thread A on Processor A:

```
a = 1
if (b == 0) then
  (critical section)
    …
   a = 0
else
   a = 0
  (retry code)
```

Thread B on Processor B:

```
b = 1
if (a == 0) then
  (critical section)
    …
   b = 0
else
   b = 0
  (retry code)
```

6. (15 pts) Consider the following C code running on two processors of a sequentially consistent shared memory multiprocessor:

Processor A:

```
for (i=0; i < 5; i++)
     x++;
```

Processor B:

```
for (j=0; j < 5; j++)
     x++;
```

a. Assume x is initialized to 0. What are the possible values x can take after both processors are done? Explain your answer. (Note: your compiler is compiling the C statement `x++` as

```
Rx <- load [address_x]
Rx <- x + 1
store Rx -> [address_x]
```

b. If the multiprocessor uses the Illinois (MESI) protocol over a bus for cache coherence what is the minimum and maximum number of bus request/reply transactions that occur when the code is run?