Name and ID:

CS / ECE 6810 Midterm Exam - Oct 21st 2008

Notes: This is an open notes and open book exam. If necessary, make reasonable assumptions and clearly state them. The only clarifications you may ask for during the exam are definitions of terms. You may use calculators. Laptops are allowed if you want to browse through class material (textbook CD, your notes, etc.), but you are not allowed to get on-line. Complete your answers in the space provided (including the back-side of each page). Confirm that you have 7 questions on 6 pages, followed by a blank page. Turn in your answer sheets before 10:40am.

1. **Performance Summaries.** You have designed an innovation that improves the CPI of two programs in a benchmark suite in the following way:

	Program A	B
CPI of baseline processor	1.2	0.6
CPI with new innovation	1.0	0.5

   The additional circuits required by your innovation cause the clock speed to reduce by 5%. What is the speedup from your innovation if your workload is such that each program executes for an equal number of cycles? **(8 points)**

2. **Branch Predictors.** Consider the following tournament branch predictor that employs a selector with 4K entries (2-bit saturating counters). The selector picks a prediction out of either a global predictor (10-bit global history is XOR-ed with 10 bits of branch PC to index into 3-bit saturating counters) or a local predictor (8 bits of branch PC index into level-1, 8 bits of local history are concatenated with 2 bits of branch PC to generate the index into level-2 that has 2-bit saturating counters). What is the total capacity of the entire branch prediction system? **(6 points)**

3. **Bypassing.** Consider the following in-order pipeline:

```
Bpred:I-cache:Decode:Decode:Regrd:IntAdd:WB
                                  :Effadd:DCache:Dcache:Dcache:WB
                                  :FPA1:FPA2:FPA3:WB
```

After register-read, Integer-adds go through "IntAdd" and "WB" (register writeback). Loads and stores go through "Effadd" (where the load/store address is calculated), then three data-cache stages, and finally the "WB" stage. Floating-point adds go through three "FPA" stages and then "WB". What are the stall cycles introduced between the following pairs of successive instructions **with and without** full bypassing? Assume the write happens in the first half of the cycle and the read happens in the second half of the cycle. (Show at least a couple of pipeline diagrams to convince me that you have understood the concepts; don't just provide the final numbers.) **(20 points)**

(a) Int-add, providing the data for a store

(b) Int-add, providing the input for an Int-add

(c) Load, providing data for an Int-add

(d) FP-add, followed by dependent FP-add

(e) FP-add, providing the data for a store

4. **Out-of-Order Execution.** Consider an out-of-order processor similar to the one described in class and in assignment 4. The assumptions are re-stated below. The only notable changes are (i) that the architecture has 32 logical registers and 34 physical registers, and (ii) the processor has a width of 2 (only up to two instructions can be fetched, decoded, renamed, issued, committed in a cycle). On power up, the following program starts executing (to simplify the problem, some of the initialization code is not shown and you can ignore that code).

```
L.D    R1, 0(R2)
L.D    R3, 0(R4)
ADD.D  R1, R1, R3
ADD.D  R2, R2, R4
BNE    R2, R4, brtarget
ADD.D  R2, R1, R3
```

(i) Show the renamed version of this code. **(5 points)**

(ii) Show when each instruction gets placed in the issue queue, leaves the issue queue, completes, and commits. **(15 points)**

Assumptions: Assume that branch prediction is perfect for a simple program like this. With the help of a trace cache, even fetch is perfect. Assume that caches are perfect as well. Assume that the dependent of an ADD.D instruction can leave the issue queue in the cycle right after the ADD.D. Assume that the dependent of an L.D cannot leave in the next cycle, but the cycle after that. Assume a ROB, an issue queue, and an LSQ with 20 entries each. When the thread starts executing, its logical register LR1 is mapped to physical register PR1, LR2 is mapped to PR2, and so on. An instruction goes through 5 pipeline stages before it gets placed in the issue queue and an additional 5 pipeline stages (6 for a LD/ST) after it leaves the issue queue (in other words, an instruction will take a minimum of 11 cycles to go through the pipeline). When determining if a L.D can issue, you need not check to see if previous store addresses have been resolved (just to make the problem simpler).

5. **Multi-Core Processors.** Today's best Intel processors are quad-core. What kind of processor will you architect for 2013, if your workload resembles the following and you are attempting to maximize performance? (Note that you receive points for thoughtful arguments, not an arbitrary design.) **(12 points)**

    - 30% of the programs you run are single-threaded and have a high degree of instruction-level parallelism (ILP)
    - 40% of the programs you run are single-threaded and have a low degree of ILP
    - 25% of the programs you run can be partitioned into a small number of threads (less than 16 threads)
    - 5% of the programs you run can be partitioned into a large number of threads (greater than 16 threads)

6. **Memory Dependences.** Describe the design of a prediction mechanism that helps overcome stalls in the LSQ because of unresolved store addresses. What is the impact of this innovation on power and energy of the processor? **(8 points)**

   Can you come up with another simple prediction mechanism for the exact same situation (a load address in the LSQ that cannot issue because of an earlier unresolved store address) that will help save cycles and that will not require any corrective measures in case the store address happens to conflict with the load address? **(6 points)**

7. **Loop Scheduling.** Consider a basic in-order pipeline with bypassing (one instruction in each pipeline stage in any cycle). The pipeline has been extended to handle FP add and FP mult. Assume the following delays between dependent instructions:

   (a) Load feeding any instruction: 2 stall cycles

   (b) FP MULT feeding any instruction (except stores): 4 stall cycles

   (c) FP MULT feeding store: 3 stall cycles

   (d) Int add feeding any instruction: 0 stall cycles

   (e) A conditional branch has 1 delay slot (an instruction is fetched in the cycle after the branch without knowing the outcome of the branch and is executed to completion)

   Below is the source code and default assembly code for a loop.

```
    Assembly code                      Source code
Loop: L.D     F2, 0(R1)            for (i=1000;i>0;i--) {
      MULT.D  F3, F2, F1             x[i] = y[i] * s;
      S.D     F3, 0(R2)            }
      DADDUI  R1, R1, #-8
      DADDUI  R2, R2, #-8
      BNE     R1, R3, Loop
      NOP
```

   (i) Show the schedule (what instruction issues in what cycle) for the default code. **(3 points)**
   (ii) How should the compiler order instructions to minimize stalls (without unrolling)? Show the schedule. How many cycles can you save per iteration, compared to the default schedule? **(4 points)**
   (iii) How many times must the loop be unrolled to eliminate stall cycles? Show the schedule for the unrolled code. **(8 points)**
   (iv) Show the software pipelined version of the code. **(5 points)**