CS 654 Exercise #4

Due Friday, Nov. 14, 4:30pm

If you have questions about this assignment, be sure to send email to cs654@cs and CC me.

Experimental Part

1. **Microbenchmark construction:** Your task is to write short assembly-language programs in Alpha to run on our research simulator ("sim-modes") of an Alpha 21264/21364—an MRF style machine. You need to write a program (or perhaps a set of programs) that *deduces the miss penalty of the L1 D$*. This kind of program is called a "microbenchmark". For an example of microbenchmark usage, refer to problems 5.2-5.3 in the textbook. But note that the benchmark you write will be quite different—I'm not suggesting you use the code in 5.2 as an example, only for broad insight into the general concept.

   More about what to do: You will probably want to go about this by measuring the (simulated) execution time of your program. As you vary the amount of ILP available during a cache miss, you should notice a distinctive change in execution time between cases that do hide the miss penalty and cases that do not. You will probably want to write your programs with some kind of loop to average the execution time across many iterations. You will need to compile your Alpha assembly-language programs on *courier*, an Alpha 21164. You should then run these resulting microbenchmark(s) on sim-modes on one of lava006-010. Note that we will not be using timing calls to the OS, but rather the "virtual" time reported by the simulator for executing a program. Because you only get one time statistic at the end of the simulation, you may need a set of programs rather than a single one with nested loops like problem 5.2 shows.

   Simulation details: We're using the simulator rather than a real Alpha machine so that you can test your program by varying the value of the system's simulated L1 D$ miss penalty. (We're also using simulation to reduce the load on courier, the one Alpha that is currently operational—Jack Davidson's group has been kind enough to accommodate us). But if you are curious, after the deadline passes—and after we get it operational again—you are welcome to test your programs on *krakatoa*, LAVA's Alpha 21264.

   Note that in our simulator, integer instructions *and* load/store instructions go through the integer issue queue (IQ). Load/store instructions then wait in a load-store queue for cache access. Stores commit in order, but as long as there are no hazards, loads can bypass stores and access the cache as soon as their effective address is available. L1 hit time is 2 cycles. Our pipeline consists of IF/ID/REN/EQ/IS/EX…/WB/CT, and branch and target prediction happen in IF.

   Other details: The simulator and documentation can be obtained from ~cs654/fall2003/assign4. The simulator binary is "sim-modes", and documentation is in the

README in that same directory. A configuration file—alpha.out-order.config—to make the simulator behave like a 21364 can also be found in that directory.

To compile programs on courier, you can use either gcc (/usr/cs/bin/gcc) or cc (the native Alpha compiler). I suggest gcc because its assembly language is easier to read. With both compilers, you obtain assembly using the –S command, and compile assembly by simply passing a .s file rather than a .c file. A very simple example C program and the resulting assembly program (sample.c/s) can also be found in ~cs654/fall2003/assign4. With gcc, be sure to specify "-nostdlib --static" as command-line options when compiling an assembly program for use with SimpleScalar. If you prefer to use cc, you'll have to determine the appropriate options.

We are leaving it up to you to obtain any other documentation that you feel you need regarding the microarchitecture or instruction set of the 21264/21364.

2. Again using sim-modes, our SimpleScalar simulator of an Alpha 21364, compare and discuss the performance of a sytem with an issue queue of 1, 4, 8, 16, and 32 entries using the gcc benchmark. You need to use the Alpha version of gcc, which can also be found in ~cs654/fall2003/assign4. The command to invoke gcc using SimpleScalar can be found there in a file called "InvokingGcc". Please run it to completion with the one input file that is provided, cccp.i.

For both questions, you should submit a writeup that explains what you did, why, your findings (with data), your conclusions, and how the data justifies those conclusions.

Karthik Sankarnarayanan, Siva Velusamy, and Yingmin Li are grad students in my group who work regularly with sim-modes, and can help answer questions about how the simulator behaves—especially since I may be out of the office much of the week. If you have questions for them, please see them directly.

Written Part

1. **H&P 3.16 a-c**
2. **H&P 3.22**
3. **H&P 3.24a**

4. **Instruction flow:** trace the following sequence of instructions as they flow through a ROB-style and MRF-style processor, using the format we did in class. However, you should assume a somewhat more realistic pipeline, using the following stages:

   `IF/ID/REN_EQ/ISS/EXE1/EXE2/EXE3/WB/CT`

   REN_EQ does rename and queue insertion in a single stage. ISS does issue and operand access (from wherever the operands reside) in a single stage. You can assume for simplicity that both styles (MRF and ROB) store operand *values* in the IQ. Integer instructions require

only EXE1; loads compute their effective address in EXE1 and access the cache in EXE2. Cache hits take one cycle, cache misses have an extra penalty of three cycles. FP instructions require all three EXE stages. Branch and target prediction occurs in the fetch stage; conditional and indirect control-flow is resolved in EXE1. Three instructions can be fetched/renamed/issued/committed each cycle.

LD.D	F2 ← 0(R2)
DIV.D	F0 ← F2, F4
ADD.D	F10 ← F0, F8
SUB.D	F10 ← F8, F10
ST.D	0(R2) ← F10
ADD	R2 ← R2 + #4

You are not required to use it, but the Excel file I used for the in-class examples is available at ~cs654/fall2003/assign4/654_ooe_examples.xls; you will need to modify it extensively but may find it useful as a starting point.