Pipeline Simulator Exercise #2

SimpleScalar Exercise #3

Due Thursday, October 7, 2003, 4:30 p.m.

This exercise is meant to extend your how pipelines function from the previous exercise and to further familiarize you with the SimpleScalar 3.0 toolset. In this phase, you will add ***bimodal branch prediction**, **single-level caches and data forwarding*** *to your pipeline. Your simulator will appropriately handle branch mispredictions and cache misses. There are 2 parts: a programming assignment and a few written exercises.*

Assumptions

For the purposes of this exercise, we are assuming the following:

- Single level caches
    - Instruction cache
        - Probed by the fetch unit
        - Miss latency: determined by the cache access function
    - Data cache
        - Probed by the MEM stage if executing a load or store
        - Miss latency: determined by the cache access function
        - Infinite write buffer, so no stalls necessary for writes to memory

- Bimodal branch predictor
    - Branch predictor will be probed on a branch in the decode stage and provide IF with a PC to fetch
    - When the branch reaches the EX, determine whether the prediction was correct, if it's incorrect, squash the wrongly fetched instructions, fetch the correct instructions

- Data forwarding:
    - If there is a data hazard, but the result is available in the EX/MEM latch or the MEM/WB latch, no stalling should occur

- Split-phase register access (writes occur in first half of clock cycle, reads in second half)

Questions to think and answer before coding (very important)

(1) When could a data hazard happen with data forwarding?
(2) What instruction could cause the data hazard?
(3) What does the pipeline need to do if there is a data hazard?
(4) What does the pipeline need to do if there is an I-cache miss?

(5) What kind of instruction could cause D-cache miss?
(6) What does the pipeline need to do if there is a D-cache miss?
(7) How to detect a branch instruction?
(8) How to do the branch prediction with bimodal branch predictor?
(9) How to decide the branch is really a taken or not-taken?
(10) How to check if the branch prediction (taken/non-taken) is correct or wrong?
(11) What does the pipeline needs to do if the branch prediction is wrong? And how to do it?

Difficult points

(1) What does the pipeline need to do for a cache-miss?
(2) How to decide branch prediction correct or wrong?

Simulator Skeleton Code

0. Download the code distribution `assign3.tar.gz` from: `~cs654/fall2003/assign3`

   You MUST use the code distributed here as your baseline for starting the assignment. Please rename the solution file to `sim-pipe2.c,` so as not to interfere with the grading of pipeline assignment part 1. (This means you'll need to edit the Makefile again, add the make rule for sim-pipe2.c, refer to the last assignment for sim-pipe.c).

   NOTE: because we're using **bpred** and **cache** modules the matching lines in your Makefile should look like this:

```
sim-pipe2$(EEXT):   sysprobe$(EEXT) sim-pipe2$(OEXT)
bpred.$(OEXT) cache.$(OEXT) $(OBJS) libexo/libexo.$(LEXT)
        $(CC) -o sim-pipe2$(EEXT) $(CFLAGS) sim-
pipe2.$(OEXT) bpred.$(OEXT) cache.$(OEXT) $(OBJS)
libexo/libexo.$(LEXT) $(MLIBS)
```

1. Include `sim-pipe.h`, `bpred.h` and `cache.h` to your simulator (sim-pipe2.c). This will give you access to the functions in the branch prediction and cache modules. The contents of `sim-pipe.h` are an assortment of stuff needed in order to do branch prediction and figure out cache miss latency.

2. You'll have to register options (sim_reg_options) and check them (sim_check_options). For hints on how to do this, look to `sim-bpred.c` and `sim-cache.c`

how to do the cache access and how to detect a cache miss?

We use L1 I-cache as an example: If we want to read an instruction at address given by fetch_pc from the L1 I-cache, and we want to know if this is a cache miss or not, we can use the following code segment. ***The function cache_access() is defined in cache.h/c***, in this example, variable cache_lat is defined as a local variable initialized to be -1, the return value of function cache_access() is a positive number if cache miss happens. In this example, cache_miss is a local variable initialized to be FALSE. You can define your own variables.

```
cache_lat = cache_access(cache_il1, Read, fetch_pc,
                          NULL, sizeof(md_inst_t), sim_cycle,
                          NULL, NULL); // cache_access defined in cache.h/c

if (cache_lat != 1)
        cache_miss = TRUE;
else
        cache_miss = FALSE;
```

In the sim-pipe2.c you get, it declares L1, L2 I-cache and D-cache, but for this assignment, we only need consider L1 cache. It also declares the TLB struct, we do not need to consider TLB in this assignment.

How to run the simulator with cache and branch predictor
In order to run the simulator with cache and branch predictor, we need to configure the branch predictor and cache. The function opt_reg_string(…) or opt_reg_int(…) is used for configuring the cache (L1/2 I-cache, L1/2 D-cache, and L1/2 I/D-cache latency). Or you can use opt_reg_note(…) to configure all the cache parameters. The functions will read the configuration from the command parameter. For cache, we need to specify the name of the cache, the sets number of the cache, the block sizes of the cache, the associativity, and the replacement policy. The same is for branch predictor configuration.

For example, if we want run the simulator with bimodal branch predictor and L1 data cache with size 4096sets,block size 32bytes,direct associated, LRU replacement policy.

```
sim-pipe2 –v –bpred:bimod -cache:dl1 dl1:4096:32:1:l …..
```

Work to do
1. Implement data forwarding/bypassing according to the assumptions given above
2. Add instruction cache, data cache, and branch predictor to the pipeline according to the assumptions given above
3. Your simulator should do the following:

   - Detect data and control hazards
   - Stall as appropriate, forward data as appropriate
   - Detect instruction cache misses and data cache misses and appropriately stall when a miss occurs for the latency required to get the instruction from memory

- Predict branches in the decode stage. Use the prediction to tell fetch the next PC to fetch
- Count the number of branches executed
- Calculate the correct prediction rate (number of correct predictions /total number of branches)

Sample Output

Sample output is can be found in
`~cs654/fall2003/assign3/assign3.sample_results.tar.gz`

The Written Assignment

1. Let us consider a single-issue, in-order system like the DLX pipeline in Chapter 3, except our pipeline only has four stages: IF, ID, EX, and WB. Loads compute their effective address and access the data cache in the EX stage. Let's assume that all operations take 1 cycle to execute, except for loads. Without a cache, loads take 10 cycles (including the cycle spent in EX to compute the effective address). Loads represent 20% of the dynamic instruction mix.

   a.) (5) What is the maximum speedup that we can attain by minimizing the load latency? Assume that, no matter what improvements are made, a load cannot take less than 1 cycle to execute (*i.e.*, it completes at the end of the EX stage). Assume that there is no impact on the clock cycle time.

   b.) (10) Suppose that 15% of the dynamic instruction mix consists of arithmetic operations that directly use the loaded values, and that the compiler can convert these sequences into arithmetic operations with a memory operand. These arithmetic-memory operations are able to compute the effective address, access the data cache, and perform the arithmetic operation in the EX stage. Since 20% of the instruction mix is loads, this means that we can convert 75% of the loads. Further suppose that this system now has a cache with a 95% hit rate (*i.e.,* 95% of loads complete at the end of the EX stage, while 5% still take 10 cycles). The ratio of hits to misses is the same for arithmetic-memory operations and for remaining load operations. The hit rate is the same for the base system and for the new system that uses arithmetic-memory operations. Finally, assume that arithmetic-memory operations complete at the end of the EX stage. Unfortunately, accommodating the arithmetic-memory operations increases the cycle time by 10%.

   Is this change worthwhile? Which system is faster and by how much?

2. Consider the following code sequence:

   lw R1 $\leftarrow$ 0(R2)
   add R4 $\leftarrow$ R1 + R3
   add R1 $\leftarrow$ R1 + R4

   a.) (5) Identify all dependences, using a 5-stage pipeline as described below.

b.) (5) Using a pipeline diagram (format of your choice), show this code sequence progressing through a five-stage pipeline (IF, ID, EX, MEM, WB) with no bypassing but with split-phase register access (*i.e.*, a register can be written in the first half of the cycle, and the result can then be read in the second half of the cycle). Loads complete at the end of the MEM phase; adds take only one cycle to complete. Registers are read in the ID stage.

Homework Submission

For the programming part:

(NOTE: We will be compiling your simulators and running our test cases on them, so make sure that your simplesim3.0 directory has group rx permissions set)

In an email to cs654@cs.virginia.edu,include the following information:

1) Group member names and email ids (no email aliases please).
2) README: This should detail how to compile your simulator and the names of files you have modified *and an overview of the design choices that you made and any assumptions that you made.*
3) Pointer to the location of the files you have modified to write your simulator. Ideally this is something like `/home/userid/cs654/assign02`

For the written part:

Follow the guidelines for labeling your assignment described in the BOC memo and submit your homework to the brown folder in the front of Olsson Hall (by the wooden mailboxes). *Please staple your assignment if it is more than one page.*

Grading Criteria:

This assignment will be graded on the following criteria:

- README file is included and describes files included and compilation instructions
- Code compiles without error in our distribution of SimpleScalar
- Each pipeline stage correctly models its behavior
- Data and control hazards are correctly identified
- Stalls for the correct number of cycles on data and control hazards
- Functional correctness (your code runs and completes successfully on the tests in bin.big)
- Simulator runs correctly on our smaller test cases