

## Project 3: w75 Pipeline Timing Simulator

Prof. Loh

CS3220 - Processor Design - Spring 2005

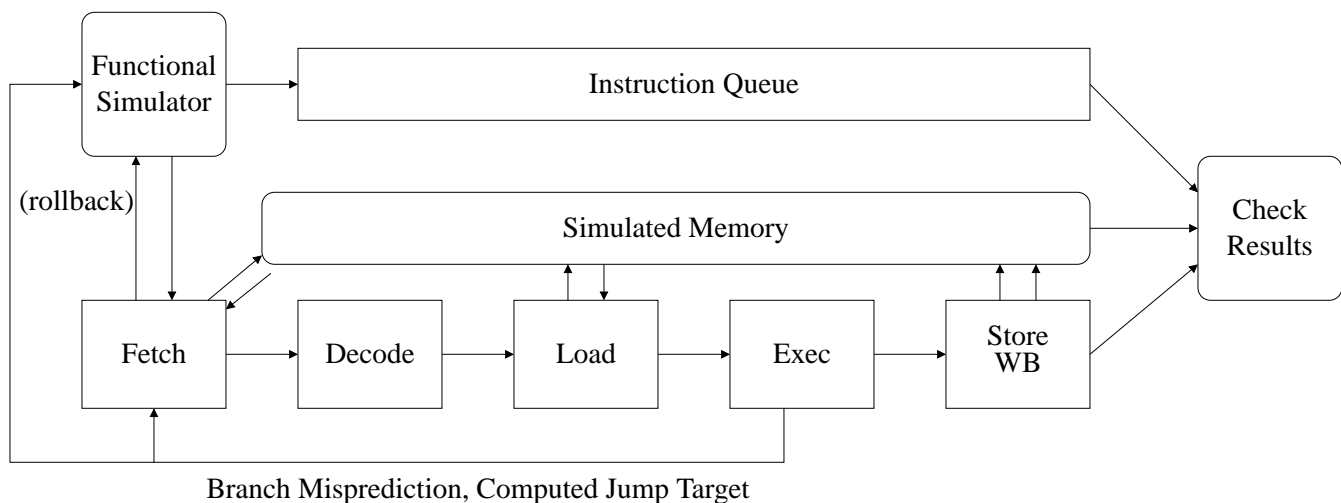
Handed Out: 19 Mar 2005

Due: 21 Apr 2005

In this third project, you will implement a complete pipeline timing simulator for the 5-stage w75 pipeline that we have been working with in class. You will implement the version of the pipeline that only has a single execute stage (fetch-decode-load-execute-store).

The simulator you will implement is called an “execute-at-execute” simulator which means that you will simulate all of the data movement and computation during the appropriate pipeline stages. Contrast this to an “execute-at-fetch” simulator that performs all computation in the fetch stage and then merely simulates the *timing* (but not that actual data movement and computations) of instructions as they progress through the pipeline.

As part of the simulator, you will extend your functional simulator to provide a *checker* infrastructure. When you fetch an instruction, the checker simulator will execute the instruction to compute the outcome of the instruction. This can let you, for example, perform simulations to find out what kind of performance you can get with a theoretical perfect branch predictor. After an instruction has made its way to the end of the pipeline and is writing back its result to the register file or memory, then the checker can verify that the timing simulator computed the correct answer (for example a bug in your simulator could cause you to bypass an operand from the wrong stage, resulting in an incorrect answer). The figure below graphically depicts the organization and interaction of the functional simulator/checker and the pipelining timing component.



The functional simulator/checker component will also include an instruction queue that simply holds all of the information for all instructions that are somewhere in the pipeline. When an instruction is fetched, we stash a copy of all of its information (including anything we need to undo its operations) in a queue entry. When the instruction exits the pipeline, we can remove it from the queue and compare it to what the pipeline actually did.

Your main loop should look something like:

```
while(1)
{
```

- store-and-writeback(); // invokes checker to verify pipeline results

- execute();
- load();
- decode();
- fetch(); // notifies functional simulator of execution of new instruction

}

The fetch function should cause the functional simulator to execute the instruction and place the relevant data in the instruction queue for later verification. It then reads the instruction bytes from memory, does the minimum amount of decode to determine if the instruction is 2 or 4 bytes, and then increments the PC to point to the next instruction.

As your functional simulator executes each instruction, you will want to assign each one of them a unique ID (you can just use an unsigned int and increment it for each instruction - we won't be running programs that execute 4B instructions). If you have a loop, the same PC may show up multiple times, and using a sequence number will allow you to differentiate between different instances when debugging.

Your timing simulator will support the following options:

- -max:inst *nnn* Similar to the functional simulator. Stop after *nnn* instructions have *exited* the pipeline. (I.e. instructions fetched after a mispredicted branch do not count toward the limit.)
- -v Similar to the functional simulator. Print out instructions as they *leave* the pipeline.
- -pipetrace Perform a cycle-by-cycle pipe trace. Each cycle, print out the contents of each pipestage (decoded instruction information, PC, sequence number, operand and result values, etc.). The pipetrace is basically a stage-by-stage version of the -v flag.
- -bpred:perfect Run the simulator with perfect branch prediction. The fetch stage utilizes the “oracle” information from the functional simulation to always fetch the correct next instruction with no bubbles.

I suggest building you simulator in smaller, easier-to-manage steps:

1. Make an instruction queue where the functional simulator inserts instructions in one end, and then after some delay, dequeues the instruction. These entries should contain all necessary information to “unroll” execution if necessary (if the instruction overwrites R1, you should save the previous value).
2. Make a dummy instruction pipeline. The fetch stage calls the functional simulator to provide an instruction. Let the instruction simply advance one stage per cycle. When it reaches the last stage of the pipeline, “execute” the instruction by reading its results from the corresponding entry in the functional simulator’s instruction queue.
3. When an instruction exits the timing pipeline, compare its result to the checker’s version. At this point in time, since we are only “executing” instructions by copying the functional simulator’s information, we will never encounter an error. To test the checker code, randomly inject errors into the execution process. For example, you could have 5% of instructions generate a random number for their result to see if the checker catches it. When the checker detects an error, you will need to flush the pipeline and rollback the functional simulator to the point before the error occurred. This step tests both the checking component and the rollback component of the simulator.
4. Implement the stages of the pipeline using perfect branch prediction. In this case, the fetch stage never injects bubbles and never needs to recover from a branch misprediction (although other errors in the simulator may still result in a checker error detection and subsequent pipeline flush and functional simulator rollback). This will involve implementing a scoreboard-like structure to determine when functional units and data resources are busy/ready, and where data should be forwarded from/bypassed to.
5. Make the fetch stage assume a default not-taken branch prediction.

At the end of the simulation (either due to a HALT interrupt or reaching the instruction limit), print out the following statistics:

1. cycles Total number of cycles simulated.

2. instructions Total number of instructions that exited the pipeline.
3. fetched Total number of instructions fetched (includes wrong-path).
4. mispredictions Total number of branch mispredictions.
5. flushed Total number of instructions thrown away due to branch mispredictions.
6. int-flushed Total number of instructions thrown away due to interrupts/system calls.
7. errors Total number of errors detected by the checker (should be zero).
8. CPI cycles per instruction (just the ratio of the first two stats).
9. data-bubbles Number of stalls/bubbles due to data-dependence hazards.
10. memory-bubbles Number of stalls/bubbles due to store-to-load memory-dependence hazards.
11. rf-reads Number of values read from the register file.
12. bypass-reads Number of values read from a bypass path.
13. rf-writes Number of values written back to the register file.

Use the following instruction latencies, assuming entirely *non*-pipelined functional units.

- Fetch: perfect instruction cache - always hits, latency of 1 cycle.
- Load/Store: perfect data cache - always hits, but latency of 2 cycles.
- Integer Multiply: 3 cycles
- Integer Divide, Mod: 16 cycles (that's a lot of stalling)
- All other integer: 1 cycle
- FP Multiply: 5 cycles
- FP divide, recip: 12 cycles
- Int ↔ FP convert: 3 cycles
- FP Transcendentals (sin, log, etc.): 12 cycles
- All other FP: 1 cycle
- All control instructions: 1 cycle

A special note on simulating interrupts: When the processor executes one of the interrupt/system call instructions, what would really happen is that the pipeline flushes, and then jumps to the library/kernel/system code that implements the system call. When the fetch stage provides the functional simulator with a PC that corresponds to the system call, the functional simulator *should not execute the system call* until the pipeline is empty. If the functional simulator executes the system call right away, it may cause side-effects that shouldn't happen because the system call may actually be on the wrong path following a mispredicted branch. So fetch should stall until the pipeline is empty (which guarantees that you are not in the shadow of a mispredicted branch), then allow the functional simulator to handle the system call.

Because the system call may modify multiple registers and/or memory locations, a normal scoreboard would not be able to track all of the potential hazards correctly. To get around this, the fetch stage should again stall until the system call has exited the pipeline (at which point all of the register values will have been written-back to the RF and memory values will have been updated).

You will email me (loh@cc.gatech.edu) a tarball containing:

1. All of your source code and a Makefile.
2. A two-page (or more) writeup. In particular, explain how you organized your instruction state, the functional simulator state, the pipeline stages, the register file/scoreboard/bypass interaction, and any other “tricky” things you had to do to make the simulator work.