Assignment - In progress

Complete the form, then choose the appropriate button at the bottom.

Title	Project 2
Due	Mar 29, 2013 11:55 pm
Status	Not Started
Grade Scale	Points (max 100.0)
Modified by instructor	Mar 11, 2013 12:48 pm

Instructions

Georgia Institute of Technology

Schools of Computer Science and Electrical &Computer Engineering

CS 4290/6290, ECE 4100/6100: Spring 2013 (Conte)

Project 2: Tomasulo Algorithm Pipelined Processor

Due: Before 11:55 PM on 29 March 2013

VERSION: 2.0

Rules

1. All students (CS 4290/6290, ECE 4100/6100) must work *alone*.
2. Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy. It is acceptable for you to compare your results with other students to help debug your program. It is not acceptable to collaborate on the simulator design or the final experiments.

Project Description:

In this project, you will construct a simulator for an out-of-order superscalar processor that uses the Tomasulo algorithm and fetches N instructions per cycle. Then you will use the simulator to find the appropriate number of function units and fetch rate for each benchmark.

Specification of simulator:

1. INPUT FORMAT

The input traces will be given in the form:

<address> <function unit type> <dest reg #> <src1 reg #> <src2 reg#>
<address> <function unit type> <dest reg #> <src1 reg #> <src2 reg#>
...

Where

<address> is the address of the instruction (in hex)
<function unit type> is either “-1”, "0", "1" or "2"
<dest reg #>, <src1 reg#> and <src2 reg #> are integers in the range [0..31]
note: if any reg # is **-1**, then there is no register for that part of the instruction (e.g., a branch instruction has **-1** for its <dest reg #>). The instruction may still have immediate values or other useful work that is not pertinent for this project.

For example:

ab120024 2 1 2 3
ab120028 1 4 1 3
ab12002c -1 -1 4 7
bc213130 0 0 1 0

Means:

"operation type 2" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type -1" -, R4, R7 *branch, no destination register!*
"operation type 0" R0, R1, R0 *Destination is also a source*

Command-line parameters

Your project should include a Makefile which builds binary in your project's root directory named *proc_sim*. The program should run from this root directory as:

./proc_sim -f F -m M -j J -k K -l L < trace_file

The command line parameters are as follows:

- F – Fetch rate (instructions per cycle)
- M – Schedule queue multiplier
- J – Number of k0 FUs
- K – Number of k1 FUs
- L – Number of k2 FUs
- trace_file – Path name to the trace file

Traces will be placed on the website.

2. FUNCTION UNIT MIX

Function unit type	Number of units*	Latency
0	*parameter: k0*	1
1	*parameter: k1*	2
2	*parameter: k2*	3

Notes:

- Instructions of type -1 are branches (see below) and are executed in the type 0 functional units.
- The number of function units is a parameter of the simulation and should be adjustable along the range of 1 to 3 units each (see experiments below).
- Function units of types 1 and 2 are pipelined into stages. (2 stages for function unit type 1; 3 stages for function unit type 2).

3. PIPELINE TIMING

Assume the following pipeline structure:

Stage	*Name*	*Number of cycles per instruction*
1	Instruction Fetch/Decode	Variable, depends on ICache and Branch Predictor
2	Dispatch	Variable, depends on resource conflicts
3	Scheduling	Variable, depends on data dependencies
4	Execute	Depends on function unit type, see table above

5	State update	Variable, depends on data dependencies (see notes 6 and 7 below)

Details:

1. You should explicitly model the dispatch and scheduling queues
2. The size of the dispatch queue is unlimited.
3. **The scheduling queue is divided into three sub-queues.** There is one sub-queue each for operations of type 0, 1, or 2. The queue for type 0 operations has *m*k0* entries, the queue for type 1 operations has *m*k1* entries, and the queue for type 2 operations has *m*k2* entries (there are *k0* type 0 function units, *k1* type 1 function units and *k2* type 2 function units). Any function unit can receive instructions from any scheduling queue entry for its type.
4. If there are multiple independent instructions ready to fire during the same cycle in the scheduling queue, service them in tag order (i.e., lowest tag value to highest). (This will guarantee that your results can match ours.)
5. A fired instruction remains in the scheduling queue until it completes.
6. The fire rate (issue rate) is only limited by the number of available FUs.
7. **There are unlimited result buses (also called Common Data Buses, or CDBs).**
8. Assume that instructions retire in the same order that they complete. Instruction retirement is unconstrained (imprecise interrupts are possible).

3.1 Branch prediction

Branches are predicted at Instruction Fetch.

Branch behavior (whether the branch is taken and where the branch goes) is known after the instruction has executed (i.e., when its results are broadcast on the result bus).

Branches are predicted using a 512-entry table of 2-bit bimodal predictors, where the initial state is 01 for each counter.

The instruction's address is hashed to generate the branch predictor index using this function: (address/8)%512

Each entry in the table has a 32-bit field for the target address of the branch. This is used to predict NPC during Instruction Fetch if the predictor state predicts the branch is taken (state = 10 or 11)

3.2 FETCH/DECODE UNIT

The fetch/decode rate is N instructions per cycle. Each cycle, the Fetch/Decode unit can always supply N instructions to the Dispatch unit. Since the dispatch queue length is unlimited, there is room for these instructions in the dispatch queue.

Details:

1. **You must model a (12,5,0) instruction cache.** Assume all instructions take 4 bytes each.
2. The Fetch/Decode unit stops decoding all instructions if *any* instruction misses in the instruction cache.
3. Each block is fetched via **BLOCKING**. Once the block is brought into memory, the Fetch/Decode unit resumes decoding instructions beginning at the instruction that caused a miss.
4. The Fetch/Decode unit will send up to **N** instructions to Dispatch. Any instruction that hits in the cache is sent even if subsequent instructions miss in the cache.
5. The Fetch/Decode unit asks the cache for each instruction one at a time. For example, if one (first) instruction misses, but the next (second) instruction is in the cache, the second instruction does not cause a miss. This is because the Fetch/Decode unit does not ask the cache for a miss until after the first instruction's missing block returns from memory.
6. Each miss takes **10 cycles**. Because the cache is pipelined, a hit does not take any time.
7. Mispredicted instructions are not requested from the cache.

3.3 WHEN TO UPDATE THE CLOCK

Note that the actual hardware has the following structure:

- *Instruction Fetch/Decode*
  PIPELINE REGISTER
- *Dispatch*
  PIPELINE REGISTER
- *Scheduling*
  PIPELINE REGISTER
- *Execute*
  PIPELINE REGISTER
- *(Execute) // types 2 and 3*
  (PIPELINE REGISTER)
- *(Execute) // type 3*
  (PIPELINE REGISTER)

- *State update*

Instruction movement only happens when the latches are clocked, which occurs at the rising edge of each clock cycle. You must simulate the same behavior of the pipeline latches, even if you do not model the actual latches. For example, if an instruction is ready to move from scheduling to execute, the motion only takes effect at the beginning of the next clock cycle.

Note that the latching is not strict between the dispatch unit and the scheduling unit, since both units use the scheduling queues. For example, if the dispatch unit inserts an instruction into one of the scheduling queues during clock cycle J, that instruction must spend *at least* cycle J+1 in the scheduling unit.

In addition, assume each clock cycle is divided into two half cycles **(you do not need to explicitly model this, but please make sure your simulator follows this ordering of events)**:

Cycle half	*Action*
first half	The register file is written via a result bus
	Any independent instruction in scheduling queue is marked to fire
	The dispatch unit reserves slots in the scheduling queues
second half	The register file is read by Dispatch
	Scheduling queues are updated via a result bus
	The state update unit deletes completed instructions from scheduling queue

3.4 OPERATION OF THE DISPATCH QUEUE

Note that the dispatch queue is scanned from head to tail (in program order). When an instruction is inserted into the scheduling queue, it is deleted from the dispatch queue.

3.5 TAG GENERATION

Tags are generated sequentially for every instruction, beginning with 0. The first instruction in a trace will use a tag value of 0, the second will use 1, etc. The traces are sufficiently short so that you should not need to reuse tags.

3.6 BUS ARBITRATION

Since there are an unlimited number of result buses (common data buses), all instructions that complete in the same cycle can update the schedule queues and register file in the following cycle. Unless we agree on some ordering of this updating, multiple different (and valid) machine states can result. This will complicate validation considerably. Therefore, assume the order of update is the same as the order of the tag values.

Example:

Tag values of instructions waiting to complete: 100, 102, 110, 104

Action:

The instruction with the tag value = 100 updates the schedule queue/register file, then instruction 102, then 104, then 110, *all of these happen in parallel, at the same time, and at the beginning of the next cycle!*

3.7 INSTRUCTION OUTPUT

The simulator outputs cycle information for each instruction. With the first instruction numbered 1 and the first cycle being 0, output the cycle when every instruction enters each pipeline unit.

3.8 STATISTICS (OUTPUT)

The simulator outputs the following statistics after completion of the experimental run:

1. Average number of instructions fired per cycle
2. Total number of instructions in the trace
3. Total simulated run time for the input: the run time is the total number of cycles from when the first instruction entered the instruction fetch/decode unit until when the last instruction completed.
4. Average number of instructions completed per cycle (IPC)
5. Maximum and Average dispatch queue size (based on number of instructions in dispatch queue at the beginning of a cycle)
6. Percentage of branch instructions predicted correctly
7. Total cycles fetch is stalled from ICache misses
8. Total cycles fetch is stalled from branch mispredicts

4. EXPERIMENTS

4.1 VALIDATION

Your simulator must match the validation output that we will place on-line.

4.2 RUNS

For each trace on T-Square, use your simulator as follows:

1. Vary the following parameters:
   - Number of FUs of each type (*k0, k1, k2*) of either 1 or 2 or 3 units each.
   - Schedule queue sizes (*m*k0+m*k1+m*k2*), where m = 2 or 4 or 8
   - fetch rate, *N = 2 or* 4 or 8
2. Based on the results, select the **least amount of hardware** (*as measured by total number of FUs (k0+k1+k2), queue entries (m*k0+m*k1+m*k2), and dispatch queue size*) for **each** benchmark trace that provides **nearly the highest value** for retired IPC. As with project 1, justify your design choice.

4.3 What to turn in

1. The experimental results as described above. (As before, *there are multiple answers for each trace file, so I will know which students "collaborated" inappropriately!*)
2. The commented source code for the simulator program.
   a. The program will be run via the provided framework in order to validate its functionality.
   b. The experimental configurations will also be validated.

4.4 Hints

- Work out by hand what should occur in each unit at each cycle for an example set of instructions (e.g., the first 10 instructions in one of the traces). **Do this *before* you begin writing your program!**
- The algorithm in the notes is not intended to be implemented in C/C++. Each "step" in the algorithm represents activity that occurs *in parallel* with other steps (due to the pipelining of the machine). Therefore, you will run into difficulty if you translate the algorithm from the notes to C/C++ directly.
- Start early: this is a difficult project.
- Keep a counter for each line in the trace file. Use this for the Tomasulo tags.
- Make each pipeline stage into a function (method)
- Execute the procedures in *reverse order* from the flow of instructions (e.g., execute the state update procedure, then the procedure for the second stage of execution, then the procedure for the first stage of execution, etc.)
- It should help to have a data structure for all of the pipeline latches (e.g., a two dimensional array, one dimension for the number of execution units, another dimension for the number of pipeline stages of the execution units)
- Add a "debug" mode to your simulator that will print out what occurs during each simulated execution cycle. The debug mode should match the style of our example verification output. (see **3.7**)
- Check the webpage frequently for updates and notes. There will be a lot of clarifications required to get everyone matching our output.

4.5 Grading

0% You do not hand in anything by the due date
+50% Your simulator doesn't run, does not work, but you hand in *significant* commented code
+20% Your simulator matches the validation output posted on the website
+20% You ran all experiments outlined above (your simulator ***must be*** validated first!)
+5% You justified each design with graphs or tables and a persuasive argument

+5% You have submitted award winning code

Additional resources for assignment

project2.tar.gz (3 MB; Mar 1, 2013 11:46 pm)
traces_v2.tar.gz (3 MB; Mar 11, 2013 12:44 pm)

Submission

Assignment Text

This assignment allows submissions using both the text box below and attached documents. Type your comments in the box below and use the Add Attachments button to include other documents. Save frequently while working.

Source
Style Format Font Size

Attachments

No attachments yet

Select a file from computer Choose File No file chosen