CS 2410 Graduate Computer Architecture
Fall 2017

Project 2: Dynamic Scheduling Simulator
Assigned: October 30, 2017.
Due: December 1, 2017 by 11:59 PM.

1 Introduction

In this project, you will implement a simulation of dynamic instruction scheduling with Tomasulo's algorithm. Suppose we name the simulator, ***tomsim***. Your simulator will mimic the behavior of a dynamic instruction scheduler to model the timing of the execution of instructions. The simulator does not need to model instruction semantics. ***tomsim*** will read an input program instruction trace and output statistics about the trace.

The simulator will also support a data cache and a memory. You do not have to develop the data cache or memory simulation yourself; you will simply reuse the components available from SST for this purpose.

2 Overview

tomsim will input an instruction trace of ***X*** instructions. An instruction trace is an ordered list of the instructions executed by a program. The instruction trace for this project will not have any branch or jump instructions.

tomsim will also input a configuration that specifies the number of functional units (FU) of different types (integer, divider, multiplier, load and store), the number of reservation stations for each FU type, and the execution latency of each functional unit type.

Using the instruction trace and the configuration, ***tomsim*** will mimic the scheduling of Tomasulo's algorithm. Instructions will go through the Issue, Read Operand, Execute and Write Register steps of the algorithm.

Once all instructions are consumed from the trace, the simulator will output statistics.

3 Simulation Model

tomsim will need to model the pipeline stages for Tomasulo's algorithm (Issue, Read Operands, Execute and Write Register). The pipeline is single instruction in-order issue.

Issue: Instructions proceed in-order through Issue and Read Operands. If there is no reservation station available in Issue for an instruction, the pipeline stalls until a reservation station of the required type is available. If there is a reservation station available in Issue, then the reservation station is allocated to the instruction (marked busy). You do not need to implement an instruction cache/memory. You can keep the instructions read from the trace in an array similar to project 1. The instructions are issued from the array one by one.

Read Operand: After Issue, an instruction moves to Read Operand with an assigned reservation station. In Read Operand, the instruction reads any available source operands from the register file, renames its unavailable source registers, and renames its destination register (to the assigned reservation station).

If some operand is not available in Read Operand, the instruction waits until the operands are available in this stage. Instructions may bypass one another in Read Operand, depending on when their operands become available.

When an instruction becomes ready to execute (i.e., it has all of its operands), a check is done in Read Operand for the availability of a FU of the type needed to execute the instruction. If an appropriate functional unit is not available, then the instruction waits in Read Operand until a FU becomes available. Multiple instructions may be ready together and need the same FU type. If there are more ready instructions than available functional units of the required type, the oldest instructions are dispatched to the free FUs. Ties can be broken arbitrarily. The younger instructions continue to wait in Read Operand.

Execute: When operands are ready and a FU is available, an instruction is dispatched to the functional unit in Execute. A dispatched instruction stays in Execute, occupying the FU and reservation station, for the number of cycles specified by the FU's latency. For instance, if an Integer unit is specified to have latency 1, then an instruction spends 1 cycle in Execute after reading operands and being dispatched to the FU. In the best case, the integer instruction takes 4 cycles in total to go through the pipeline (Issue, Read Operands, 1-cycle Execute, Write Register). You do not need to model the semantics (the operation) of the instruction in Execute.

Write Register: Once an instruction finishes executing, it broadcasts the availability of its destination value to any waiting instructions and the register file. This broadcast happens in the Write Register stage. It takes 1 clock cycle. Thus, a consumer instruction waiting on a register value can potentially be dispatched to execute on the cycle after the broadcast. You may assume an infinite number of common data busses (CDB)! That is, you do not need to model the potential structural hazard for a CDB. Note that multiple instructions can proceed through Write Register on the same clock cycle. In Write Register, the functional unit and reservation station occupied by an instruction are released. You do not need to model the actual data values, but you will need to model the renaming associated with the completion of an instruction.

Pipeline Model: There are four functional unit types: Integer, Multiplier, Divider, and Load/Store. The table below gives the opcode types that are executed by each functional unit type.

Type	**Opcodes**
Integer	add, sub, nor, and, lis, liz, lui, put, halt
Divider	div, exp, mod
Multiplier	mul
Load/Store (ls)	lw/sw

Each functional unit has reservation stations associated with it. The reservation stations are associated by functional unit type. For instance, the reservation stations for an Integer functional unit are separate from the reservation stations for a Divider functional unit. However, all functional units of the same type share the same set of reservation stations. Thus, an instruction can be dispatched from the reservation station of some type to any functional units of that type. For instance, a pipeline might have 2 Integer units sharing 8 reservation stations. The organization for this case is shown in Figure 1. An instruction from the 8 reservation stations can be dispatched to either of the Integer functional units.

4 Memory Hierarchy

In this project, you must have a L1 data cache and main memory. For the cache, you should use SST's cache component, and for the memory, use SST's simple DRAM component (the same as project 1). To handle load and store instructions, assume the following.

1. There is a *single load-store unit (L/S unit)* that handles loads and stores. This unit has an input queue, i.e., reservation stations, which are managed as a first-in, first-out queue (FIFO).

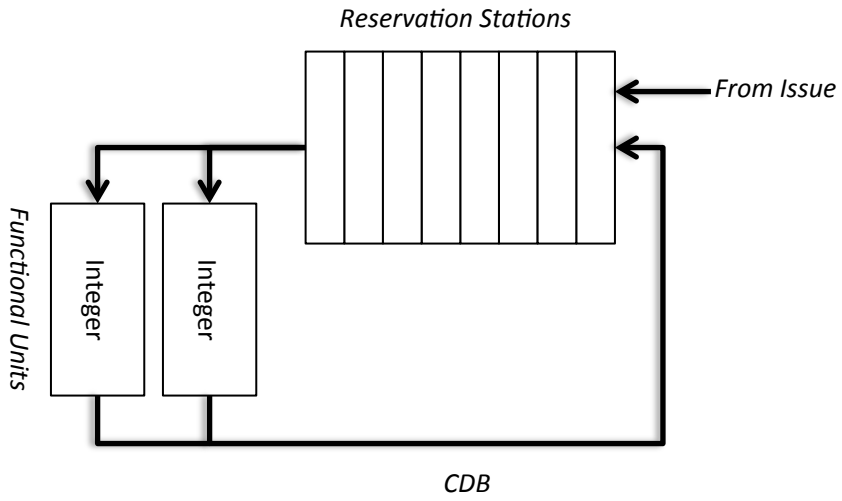

Figure 1: 8 reservation stations with two Integer functional units.

Loads/stores that are newly issued to the L/S unit are inserted at the tail of the queue. A load/store executes only when it reaches the head of the queue and its source operands are available. The load/store is removed from the queue when the cache/memory responds with an acknowledgement of completion. This structure avoids the need to schedule loads and stores, i.e., there is no need for memory disambiguation since loads and stores are executed in program order. Likewise, there is forwarding of store destination values to loads from the queue. Yes, this is L/S unit inefficient, but hey, it's simple(r)!

2. When a load/store reaches the head of the queue and has its source operands, the load/stores is dispatch to the L/S unit, which computes the effective address. After the number of cycles specified for the L/S unit's latency have elapsed, the memory operation is sent to the cache/memory. Such an operation is said to be "pending". A pending memory operation *does not block the CPU pipeline*. However, let's assume that only one memory operation can be pending at any time. When the memory operation completes, the head pointer is advanced to the next operation in the queue. (Don't forget: A load needs to broadcast the loaded value to the CDB.) The L/S unit latency in the configuration file is how many cycles it takes to compute the effective address (see below).

3. The L/S unit's FIFO queue needs to be able to receive source operand values from the register file or from a CDB broadcast. In reality, this queue is nothing more than a bunch of reservation stations, where the dispatch order to the L/S unit is enforced to be FIFO (rather than first ready to execute).

4. The data cache is configurable: the associativity and the cache size can be changed. The block size is fixed to 16 bytes (8x 16-bit words). That is, the SST cache parameter `cache_line_size` should be set to 16. The allowable associativities are 1 (direct mapped), 2, 4 and 8. The associativity is set with the SST cache paramenter, `associativity`. The allowable cache sizes are 256 bytes to 8,192 bytes. The cache size is set with the SST cache parameter, `cache_size`. The cache's frequency should be set to the CPU's clock frequency (i.e., `cache_frequency` is set to the CPU clock frequency). The access latency for the cache is 1 clock cycle (i.e., `access_latency_cycles` is 1). Be sure to set the SST cache parameter `L1` to `true`.

5. The cache and memory configuration and organization can be setup in SST's Python configuration file. See the example in the hints on how to connect the cache to the memory.

Extra credit: Do you want to try something more challenging? Yes, you say! OK, try to implement full memory disambiguation and forwarding from the stores directly to loads on a matching address. You can make this even more fun: Allow multiple pending memory operations. For this project, you probably won't notice any benefit, but generally, these "optimizations" are usually implemented. You can get some extra credit, if you decide to try this. Amount of extra credit to be determined.

5 Project Requirements

The simulator should be runnable from the command line with SST. It will take three file name arguments:

```
file names: trace.t configuration.json output.json
```

where, `trace.t` is the input trace, `configuration.json` is the configuration of the pipeline, and `output.json` is the output result.

5.1 Input Trace

The input trace is a list of ***X*** encoded instructions. It has the same format and encodings as project 1. You can re-use your input program code from project 1. However, the input trace will not have any branches or jump instructions.

For testing, you can use your ***X*** simulator to generate a trace. Simply add output statements to your simulator that outputs the instructions executed to a file (one instruction in hex format per line). When a branch or jump is executed, do not output anything. This output file is now an instruction trace that can be input to ***tomsim***.

Here is an example trace:

```
800A
7000
89FF
7020
9100
7020
6800
```

5.2 Configuration

The simulator will be configured with a JSON file. The configuration will permit specifying the number of functional units of different types, the number of reservation stations associated with a functional unit type, and the latency of executing in the functional unit. The configuration also permits configuring the cache and CPU clock speed. The JSON has the format:

```
{ "integer":{"number":2,"resnumber":4,"latency":1},
  "divider":{"number":1,"resnumber":2,"latency":20},
  "multiplier":{"number":2,"resnumber":4,"latency":10},
  "ls":{"number":1,"resnumber":8,"latency":3},
  "cache":{"associativity":1,"size":"4096B"},
  "clock":"1GHz" }
```

This configuration specifies the following:

1. There are 2 Integer FUs and 4 reservation stations. An integer operation takes 1 clock cycle.
2. There is 1 Divider FU unit with 2 reservation stations. A divide executes in 20 clock cycles.
3. There are 2 Multiplier FUs with 4 reservation stations. A multiply takes 10 clock cycles.
4. There is 1 Load/Store Unit. This unit has 8 reservation stations and takes 3 clock cycles to compute an effective address (prior to being dispatched to the cache/memory). As noted above, the reservation stations for the L/S unit are really a FIFO queue. The value for the number of L/S units will always be set to 1.
5. The data cache is direct mapped and has 4,096 bytes. The size is specified in bytes with the units specified. You can use this parameter directly to set the cache size.
6. The clock speed is 1000 MHz (1 GHz). You can use this parameter to directly set the speed of the CPU's clock in SST.

5.3 Output

The simulator will output statistics in JSON about execution. The output statistics are: number of clock cycles for the whole trace, the number of instructions executed by each FU, the number of stall cycles due to structural hazards, and the number of operands read from the register file. The output JSON has the format:

```
{"cycles" : 100,
"integer" : [{ "id" : 0, "instructions" : 10 }, { "id" : 1, "instructions" : 8  }],
"multiplier" : [{ "id" : 0, "instructions" : 1 }, { "id" : 1, "instructions" : 0 }],
"divider" : [{ "id" : 0, "instructions" : 2 }],
"ls" : [{ "id" : 0, "instructions" : 3 }],
"reg reads" : 9,
"stalls" : 13 }
```

In this output, the field `id` distinguishes multiple functional units of the same type. The field `instructions` is the number of instructions. The entries `reg reads`, `stalls` and `cycles` report the number of register file reads, the number of pipeline structural hazard stalls and the number of cycles for the simulation.

5.4 Programming Language, Libraries, etc

You must use SST to implement this project. You can use SST to provide the clock tick stimulus to drive the simulation.

6 Turning in the Project

For grading, you will both turn in the project by email (as explained below) and schedule a demo of the project with Wenchen Wang or Bruce Childers. Your project will be graded primarily during the demo with possibly additional follow-up after the demo.

6.1 What to Submit

Prior to scheduling the demo, you must submit all files for the project. You must submit an archive with your source files, header files, and a README file. Do not include SST itself. For instance, your archive might include:

`tommy.cc` (please include all source and header files, but don't include SST itself)
`README.txt` (a help file; see below)

These files should be put into a single archive; you may use either a gzipped tar-ball, or a zip file. Please name your submission with your last name. For instance, `childers-project2.zip` is a good file name for Bruce Childers' submission of project 2. I will use a Mac to unpack your submission. If I cannot unpack the submission, or there are any missing files, I will not grade the project and you may receive a 0.

In the README, put your name and e-mail address at the top of the file. The README file should list what works (i.e., instructions implemented) and any known problems. If you have known problems/issues (bugs!), then you should clearly specify the problems in this file. The explanation and bug list are critical to grading. So, if you're uncertain whether to include something, then err on the side of writing too much and just include it.

Due date is: December 1, 2017 by 11:59 PM. Demos will be held shortly after this date.

6.2 Where to Submit

Submit your files to Wenchen Wang, email `wew50@pitt.edu`.

6.3 Demonstration

You will demo your simulator for grading. W will give you some test files (configurations and traces) to run in the demonstration. These files will *not* be provided prior to the demonstration. I will give them to you on an USB key. You should be able to load the test files into your simulator, configure the simulator with the configuration, and run the simulation. You should be comfortable with demonstrating the project, including changing the configuration, loading programs and explaining your approaches.

All projects will be demonstrated shortly after December 1, 2017, likely during the week of December 4. Please plan ahead. Each demo will be 20 minutes. A sign-up sheet with appointment times will be made available prior to the demo date. If you have a class conflcit with the demo day/times, we can make accommodations. If we have not received your file submission by the due day/time, we will not grade the project and you may receive a 0.

Because the demos are intended to be short, please be prepared to quickly and smoothly show what you have done! The time allotted for each demo is strict.

7 Collaboration

In accordance with the policy on individual work, this project is strictly an individual effort. You must not collaborate with a partner. It is your responsibility to secure and back up your files.

7.1 Grading the Project

We will grade the project with multiple configurations and traces covering many (if not all) of ***X*** instructions during the demo. Your ability to easily and smoothly demonstrate the project and answer questions will also be part of the evaluation. Primarily, the concern is functional correctness and implementation of the project requirements. To ensure that your processor is working as expected, you will find it very useful to write your own test case programs to cover all instructions.

8 Hints

These hints were originally written before we started to use SST. The hints still apply, but you think about how to leverage SST's discrete event simulation, particularly to schedule callbacks in the future. I updated the hints to reflect some of SST.

The project needs to manage "simulation time". The simulation time is simply a count of cycles. The simulation proceeds in steps, which increments the simulation clock cycle count by one. You can use SST's clock tick for this purpose.

On each simulated clock cycle, multiple actions need to be performed to model the movement and execution of instructions in the pipeline. The actions need to happen according to pipeline conditions (e.g., operand availability) and latency of the actions. With this in mind, you will need some way to cause simulation actions to happen on specific simulation clock cycles.

While there are many ways to manage simulation time and actions, a classic approach is discrete event simulation, as employed by SST. In this method, you can keep a list of events with timestamps. The simulation proceeds in steps of the CPU clock cycle, i.e., your simulation receives a callback on each tick. On each clock cycle tick, the list is checked for events that have a timestamp matching the current time. A matching event is removed from the list and handled on the current clock cycle. In this approach, you are essentially managing the simulation yourself. You might want to see what SST offers to do this as well (e.g., scheduling callbacks).

It will be useful to allow multiple event types in the list, where each event type is a different action in the life of an instruction. An instruction will cause some event to be in the list, which will be processed on a matching clock cycle to update the movement of the instruction.

As an example, consider an Instruction in Read Operands. This instruction might have put a Read Operand event in the event list during Issue. The handler for the Read Operand event would read the register file (when the simulation clock cycle matches the event's timestamp). If all operands can be read from the register file and a FU is available, a new Execute event would be inserted (i.e., scheduled) for the next clock cycle. If the operands are not all ready or there is no FU, then a new event would be scheduled to Read Operands again on the next cycle. In this way, the instruction remains in Read Operands until the conditions to execute are satisfied. Other actions for in-flight instructions can be handled similarly. You will probably need at least 4 event types: Issue, Read Operands, Execute, and Write Result. Note, you may wish to have more event types, such as a Wait event type when an instruction is waiting on operands/FU.

Most events can be inserted with a timestamp corresponding to the next clock cycle. An Execute event, however, might be inserted with a further-away timestamp that corresponds to the latency of the instruction (i.e., current time + latency of instruction).

You may also want to keep the event list ordered by timestamp (soonest first). This can help the efficiency of processing the event list since you can terminate processing the list on a simulation cycle when a timestamp is found that is greater than the current simulation clock cycle.

You may also find it useful to process events in order by their type. You may wish to process events for later instructions in the pipeline before events of earlier instructions. For example, it could be useful to process Write Register before Read Operand.

I strongly recommend that you plan your simulation before trying to implement it. The data structures that you choose will likely have a big influence on your required effort. So, think carefully here!! I also strongly recommend that you implement and test in steps. It is very difficult to debug problems in this type of simulation; proceeding in small steps and carefully testing your implementation in an incremental fashion can be a big life saver! Lastly, you will probably want to write carefully crafted test cases. You can craft these test cases to exhibit certain timing behavior, which can then be checked to test your simulator.

Example of Cache/Memory Configuration

Here is an example of how to connect the L1 data cache and the memory in the SST Python configuration script.

```
import sst
core = sst.Component("XSim","XSim.core")
core.addParams(...)

# Get memory latency
memory = sst.Component("data_memory", "memHierarchy.MemController")
memory.addParams(...)

l1_cache = sst.Component("l1cache", "memHierarchy.Cache")
l1_cache.addParams(...)

cpu_cache_link = sst.Link("cpu_cache_link")
cache_data_memory_link = sst.Link("cache_data_memory_link")

cpu_cache_link.connect(
  (core,    "data_memory_link", "100ps"),
  (l1_cache,  "high_network_0",      "100ps")
)

cache_data_memory_link.connect(
  (l1_cache,    "low_network_0", "100ps"),
  (memory,  "direct_link",      "100ps")
)
```