Computer Assignment 3

Fall 2025

Deadline: Sunday, Nov 23, 11:59 PM

(Upload it to Gradescope.)

Note:

The deadline for this assignment is Sunday, 23rd, but we strongly suggest that you finish this assignment before your Quiz 2 (Thursday, 20th), as the content is quite relevant (and helpful) for preparing for the quiz.

Project Description

In this project, you will construct a simulator for an out-of-order (OoO) superscalar processor. Please note that the design and terms used in this assignment are slightly different than the OoO example we covered in class. Many aspects of the processor pipeline is simplified to make the implementation more manageable. Please make sure that you read this document carefully and please pay attention to terms, stages, and timing, as will be discribied in the following.

Design Overview

We consider an N-wide (N is a user-defined parameter, and by default is N=4). Our design has five distinct stages which are combinations of stages described in the lecture. Specifically, this design has: ***Fetch, Dispatch, Schedule, Execute, and State Update.*** The frontend (Fetch and Dispatch) are significantly simplified and the main goal of this project is to design the backend (schedule, execute, and state update). The details of the design are provided later.

Traces and Outputs:

We have provided for traces for your design which can be found here: traces

The input traces are 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..127]

For example: “`ab120024 0 1 2 3`” means:

“ab120024” is the PC/address, "operation/functional unit type" is 0, destination register is R1, and source 1 and 2 registers are R2, R3/

Important: If the Function type is -1, use FU=1 for execute. Also, 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 #>)

Please note that we are NOT implementing a particular ISA but rather an ad-hoc ISA/microoperation to make the implementation simpler.

Furthermore, we have also provided an output folder to help you debug your design: output1.1. For each trace, there are two output files: ".log" and ".output". The log file shows the per cycle behavior of the simulator and the output file shows the cycles where an instruction *enters* a particular stage.

Important: For any given instruction/cycle, the "log" file indicates the cycle where the instruction enters that stage (e.g., `"14 4 5 7 9 11"` means that instruction 14 enters fetch at cycle 4, dispatch at cycle 5, schedule at cycle 7, execute at cycle 9, and state update at cycle 11). Conversely, the "output" file indicate the cycle where the instruction leaves that stage (e.g., for the same instruction 14, you can see that it leaves fetch at cycle 4 - meaning that this instruction only stays in fetch stage for one cycle - leaves dispatch at cycle 6, and leaves schedule at cycle 8. The notation for execution stage is different, which will be discussed later).

We strongly recommend that before writing any code, try to manually create these logs for first 10 instructions (of at least two different traces) by hand and compare your work with the logs/outputs to make sure that you have understood the timing and steps correctly.

Code:

The skeleton code can be downloaded from here:
Files
It has three main files:

- ***procsim_driver.*** This is the main function of your project. It reads the input parameters and the trace, and runs the simulator. It also produces and prints the stats. Details about each step are provided later.
- ***procsim.cpp.*** The main class for your design. You are allowed to add as many functions and/or define new classes as you see fit.
- ***prcosim.hpp.*** The header file for the Procsim class. You can also add new header files and define other utility functions if needed.

Your project includes a Makefile that builds a binary in your project's root directory named procsim. The program should run from this root directory: `./procsim -r R -f F -j J -k K -l L < trace_file`

The command line parameters are as follows: F – Fetch rate (instructions per cycle), J – Number of k0 function units, K – Number of k1 function units, L – Number of k2 function units, R – Result buses, trace_file – Path name to the trace file

By default, you should use the values indicated in the trace folder for each program.

Design Details

Stages:

a) Fetch

The fetch design is straightforward. At each cycle, you should fetch N new instructions. We assume that the pipeline never stalls nor we need to support any branching, so you always fetch N new instructions.

b) Dispatch

This is a combination of decode and rename stages discussed in the lecture. To simplify things, you don't need to implement the decode and/or rename logics and assume that registers are all physical register. The only thing needed during the dispatch stage is writing the fetched instructions (N) into a dispatch queue. We assume that this queue has unlimited size. To further simplify the design, instead of using address/PC, use a counter/tag for each instruction (i.e., first dispatched instruction has tag=1, second one is tag=2, and so on. This also makes the debugging and comparing with output/log file easier as we are using the same tag scheme).

c) Scheduling

This is where you implement your reservation station. Instructions move from dispatch to schedule queue (aka reservation station) only if there is space available inside the RS. The size of RS is 2*(number of k0 function units + number of k1 function units + number of k2 function units). Assume a *centralized* design *combined* ROB and RS combined. If RS is full, the instruction has to stay in the dispatch queue (but do NOT stall the processor). Also, to schedule an instruction, 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.

d) Execute and Functional units:

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

Note that to issue/fire/execute and instruction, you have to keep track of data dependencies and FU availability (i.e., ready bits for FUs and registers).

Important Notes:

1. If there are multiple independent instructions ready to be issued/executed 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.)
2. A fired instruction remains in the scheduling queue until it completes.
3. The fire rate (issue rate) is only limited by the number of available FUs.

4. There are R result buses (also called Common Data Buses, or CDBs). These are used to update/forward data to the reservation station.
   Important: If an instruction is completed but all result buses are used, it must **wait** an additional cycle inside the execution stage. The function unit is not freed in this case. Hence, a subsequent instruction might stall because the function unit is busy. The function unit is freed only when the result is put onto a result bus. Note that you should serve completed instructions in tag order (i.e., if there are R result buses and R+1 completed instructions, the one with highest tag has to wait). Example: Tag values of instructions waiting to complete: 100, 102, 110, 104, where R = 3. You should implement the following: The instruction with the tag value = 100 updates the schedule queue/register file, then instruction 102, then 104, all of these happen in parallel, at the same time, and at the beginning of the next cycle! On the next cycle, the instruction with the tag value = 110 updates the schedule queue/register file.

e) State Update/Retire
   The update/retire logic is simplified so you don't need to actually update register file/memory values. Moreover, state update can happen out-of-order (i.e., we are not implementing the reorder logic).

Important Timining Infromation:

Stage	*Name*	*Number of cycles per instruction*
1	Instruction Fetch/Decode	1
2	Dispatch	Variable, depends on resource conflicts
3	Scheduling	Variable, depends on data dependencies and conflicts
4	Execute	Depends on function unit type and conflicts
5	State update	1

Note that the actual hardware has the following structure:

```
Instruction Fetch/Decode
PIPELINE REGISTER
Dispatch
PIPELINE REGISTER
Scheduling
PIPELINE REGISTER
Execute
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 there should be at least one cycle per stage. 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.

Additionally, we assume the “half cycle” behavior for the registers as follows (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 (ready bits updated) via a result bus
	The FU of the completed instruction is marked free
	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

Statistics (Output)

The simulator outputs the following statistics after completion of a program:

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
6. The cycle-by-cycle behavior for all cycles of execution.

How to Design:

You should also follow these guidelines/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!
- Pay attention to the timings (parallel and sequential) as well as order of operations.
- Start early: this is a difficult project.
- Make each pipeline stage into a function.

- 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.
- 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.
- Ask questions on Campuswire.

What to Submit:

1. Your code (prcosim_driver, procsim.cpp and .hpp and any other cpp and hpp files you have created). Your code should be well-commented. **Do NOT zip your files** or put them in a folder. Upload them directly to Gradescope (seriously, DO NOT ZIP!!!!).
2. A report submitted as a PDF file showing:
    a. The screenshots show that your design fully matches the output files (use the diff command to show the difference).
    b. A table for the total number of cycles (for each application) for the following design configurations:
        i. Default config (baseline)
        ii. Default config but change R to 4
        iii. Default config but change N=8
        iv. Default config with N=8 and R=4

Gradescope Tests:

Similar to CA1, your code should pass the Gradescope tests (we may use different configurations). To pass the Gradescope tests, make sure that your code only prints the total number of cycles (i.e., comment out print_statistics and outer printfs in the code except the last one). You can submit as many times as you want!

Acknowledgement

This project was inspired by Georgia Tech's CS6290 course, taught by Professor Tom Conte.