CS520 Computer Architecture

Project 1 – Spring 2020

Due: 4/10/2020, 11:59 pm

1. RULES

(1) All students must work alone. Cooperation is not allowed.
(2) Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
(3) You must do all your work in the C/C++.
(4) Your code must be compiled on remote.cs.binghamton.edu or the machines in the EB-G7 and EB-Q22. This is the platform where the TAs will compile and test your simulator. As I know, they all have the same software environment.

2. Project Description

In this project, you will construct a simple pipeline and optimizations.

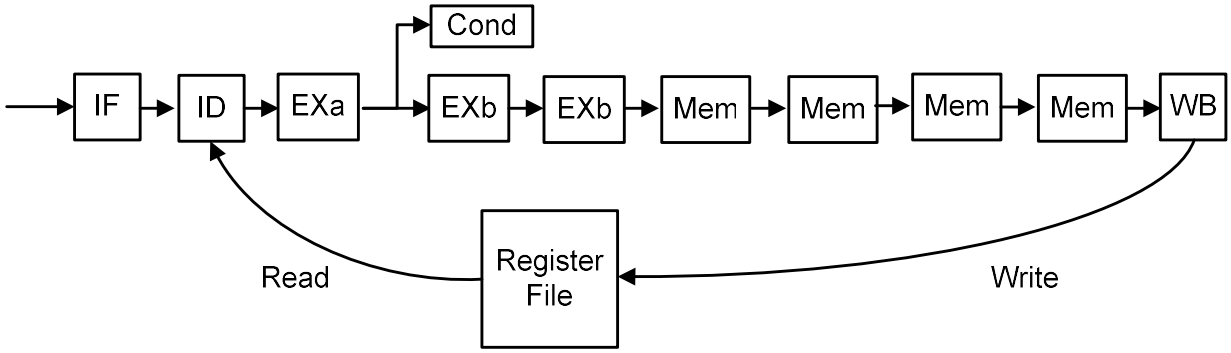
3. Simple Pipeline

Model simple pipeline with the following stages.

- 1 stage for fetch (IF)
- 1 stage for decode (ID)
- 1 stages for addition and subtraction (Execution unit A: EXa)
- 2 stages for multiplication and division (Execution unit B: EXb)
- 4 stages for memory operation (Memory unit: MEM)
- 1 stage for write back (WB)
- 16 x 4B registers
- 64KB memory (code for 0-999, data for 1000 – 65535)
  memory.map is the memory map file for this project.
- Instruction formats

```
set Rx, #Imm   // set value to register Rx
ld  Rx, #Addr      // load into register Rx the data stored in the address #Addr. E.g.) ld R1, 0x0010
ld  Rx, Ry         // load into register Rx the data stored in the address at Ry
st  Ry, #Addr      // store the content of register Rx into the address #Addr. E.g.) st R1, 0x0010
st  RX, Ry         // store the content of register Rx into the address at Ry
add  Rx, Ry, Rz    // Compute Rx = Ry + Rz
add  Rx, Ry, #Imm   // Compute Rx = Ry + #Imm (an immediate valve)
sub  Rx, Ry, Rz       // Compute Rx = Ry – Rz
sub  Rx, Ry, #Imm   // Compute Rx = Ry - #Imm (an immediate valve)
mul  Rx, Ry, Rz       // Compute Rx = Ry * Rz
mul  Rx, Ry, #Imm  // Compute Rx = Ry * #Imm (an immediate valve)
```

```
div  Rx, Ry, Rz        // Compute Rx = Ry / Rz
div  Rx, Ry, #Imm   // Compute Rx = Ry / #Imm (an immediate valve)
bez Rx, #Imm     // branch to #Imm if Rx==0
bgez Rx, #Imm   // branch to #imm if Rx >= 0
blez Rx, #Imm   // branch to #imm if Rx <= 0
bgtz Rx, #Imm   // branch to #imm if Rx > 0
bltz Rx, #Imm   // branch to #imm if Rx < 0
ret                  // exit the current program
```



The dependency check and register read is done at ID stage. If there is dependency, the instruction must wait at ID stage until the result becomes available. The branch target address is computed at EXa stage and the branch condition is checked at the next cycle. PC is updated at the same cycle. The register is updated at WB stage.

3.1. Simple pipeline

Your register file only has one read port and one write port. Each register only allows one operation at a cycle. If one register receives both read and write operations at the same cycle, write operation is always performed first. If your instruction has two register operands, the instruction needs two cycles to read both operands (structural hazard). Your simulator should output the following:

(1) (10 points) Print the program name, the number of each instruction type executed in the simulation, and register read and write count.

Program name:

ADD:
SUB:
MUL:
DIV:
LD:
ST:
BEZ:
BGEZ:
BLEZ:
BGTZ:

BLTZ:
RET:

Total instruction counts:

Register READ:
Register WRITE:

(2) (10 points) Print the final contents in the register. Also, the content in the memory file must be correct too.

R0:
R1:
...
R15:

(3) (30 points) Print the performance results of this simulation with the final memory map file (memory_0.map).

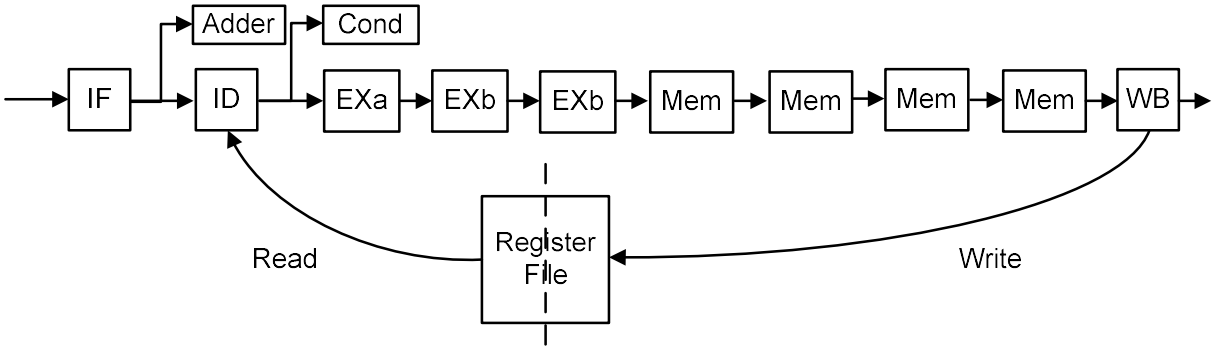
> Simple pipeline <

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

3.2. Pipeline update #1

(1) (15 points) Update simple pipeline as follows. Your register file still has one read port and one write port. However, the register file allows the write operation in the first half of the clock cycle and read operation in the second half of the cycle. We have additional adder so that the branch target address is computed at ID stage and the branch condition is checked at the next cycle. Again, PC is updated at the same cycle. Print the performance results of this simulation with the final memory map file (memory_1.map).



> Pipeline update #1 <

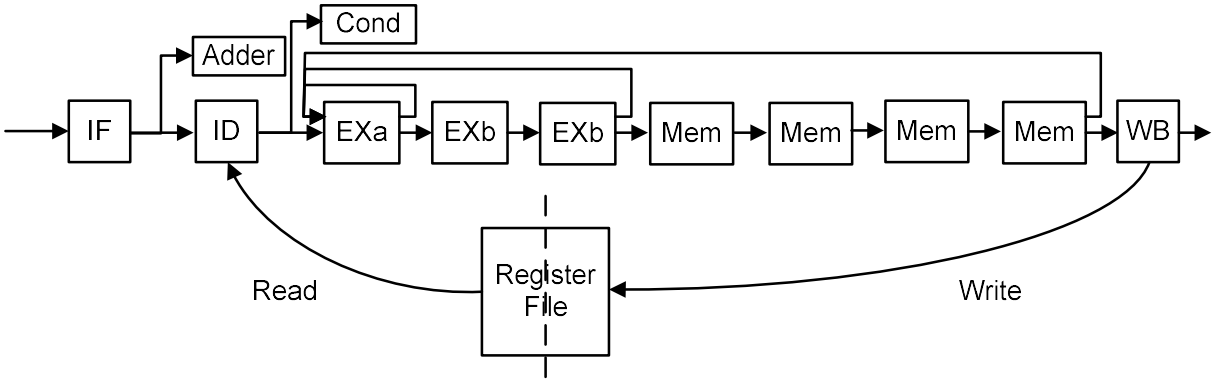
R0:
R1:
...
R15:

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

3.3. Pipeline update #2

(1) (15 points) Update the above pipeline as follows. The pipeline now has bypassing/forwarding. The instruction proceeds to EXa stage when the operand is ready to be forwarded. It receives the forwarded data at the beginning of EXa stage. The result of each instruction becomes ready after the last pipeline stage of the corresponding execution unit (e.g. if MEM unit needs 4 cycles to complete an instruction, your result will be available after the 4th cycle in the Mem unit). Also, the register file now has two read ports and one write port so that it can read two registers at the same cycle. Print the performance results of this simulation with the final memory map file (memory_2.map).



> Pipeline update #2 <

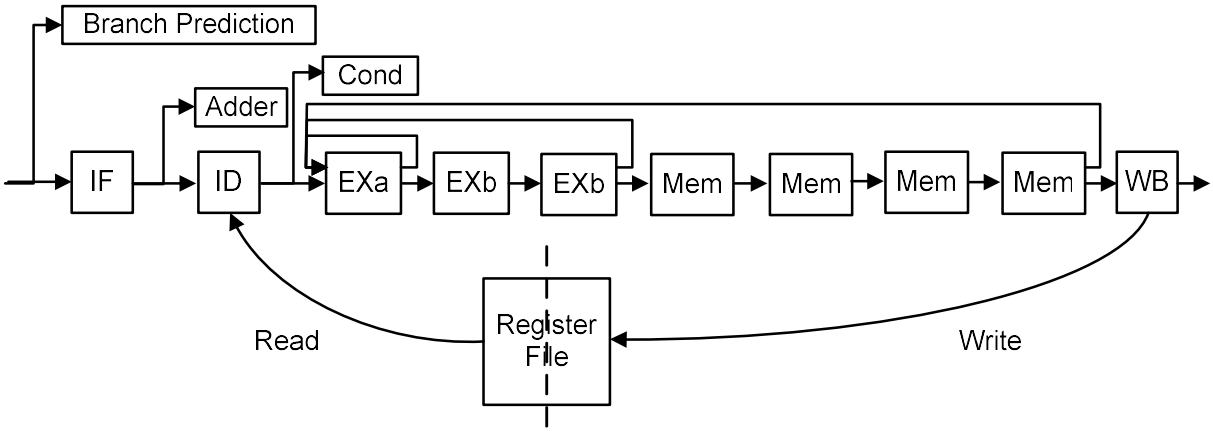
R0:
R1:
...
R15:

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

3.4. Pipeline update #3

(1) (15 points) Update the above pipeline as follows. Now your pipeline has branch prediction. Assume that the hit rate is always 90% with the fixed pattern that always experience 1 miss after 9 hits (e.g. H, H, H, H, H, H, H, H, M, H, ...). Print the performance results of this simulation with the final memory map file (memory_3.map).



> Pipeline update #3 <

R0:
R1:
...
R15:

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

3.5. Experiments and report

Programs: test program #1 and test program #2

(5 points) Plot four bar graphs of speedup over the simple pipeline on the y-axis versus pipeline updates on the x-axis for each program. You don't need to include the simple pipeline (because its speedup is always 1). Thereby, each graph contains 3 bars.

4. Validation and Other Requirements

4.1. Validation requirements

Sample simulation outputs will be provided on the website. You must run your simulator and debug it until it matches the simulation outputs. Your simulator must print outputs to the console (i.e., to the screen).

Your output must match both numerically and in terms of formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each program:

1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing "> your_output_file" after the simulator command.

2) Test whether or not your outputs match properly, by running this unix command:
"diff –iw <your_output_file> <posted_output_file>"
The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the sample outputs have whitespace. Both your outputs and final memory.map must be the same as the solution.

3) Your simulator must run correctly not only with the given programs. Note that TA will validate your simulator with hidden programs.

4.2. Compiling and running simulator

We will provide you the instruction parsing code as a library. We recommend you use the provided libraries for your simulator.

You will hand in source code and the TAs will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see section "Grading").

1. You must be able to compile and run your simulator on machines in EB-G7 and EB-Q22. This is required so that the TAs can compile and run your simulator. You also can access the machine with the same environment remotely at remote.cs.binghamton.edu via SSH.

2. Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named "sim". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.

3. Your simulator must accept command-line arguments as follows:

The pipeline must be able to receive the 2 parameters, pipeline mode (0: simple pipeline, 1: pipeline update #1, 2: pipeline update #2, 3: pipeline update #3) and a program name.

e.g. sim 0 test_01.asm

4. Your simulator must print outputs to the console (i.e., to the screen) except the final memory map. This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results. Your simulator must leave the final memory map file as follows (simple pipeline: memory_0.map, pipeline update #1: memory_1.map, pipeline update #2: memory_2.map, pipeline update #3: memory_3.map).

4.3. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons.

First, the TAs need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TAs as they may be able to help you speed up your simulator.

Second, you will be running many experiments during your implementation. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

5. What to submit

You must hand in a single zip file called project1.zip. Below is an example showing how to create project1.zip from a Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), and the Makefile, and your project report (report.doc).

zip project1 *.cc *.h Makefile

project1.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

(a) Project report. This must be a single PDF document named report.pdf. (No Word documents please.) The report must include the following:

- A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A cover page will be posted on the project website.

(b) Source code. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.

(c) Makefile. See Section 4.2 for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

6. Penalties

Various deductions (out of 100 points):
-2 points for each hour late.

Up to -10 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.

Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.