CS520 Computer Architecture

Project 4 – Spring 2023

Due date: 5/8/2023

1. RULES

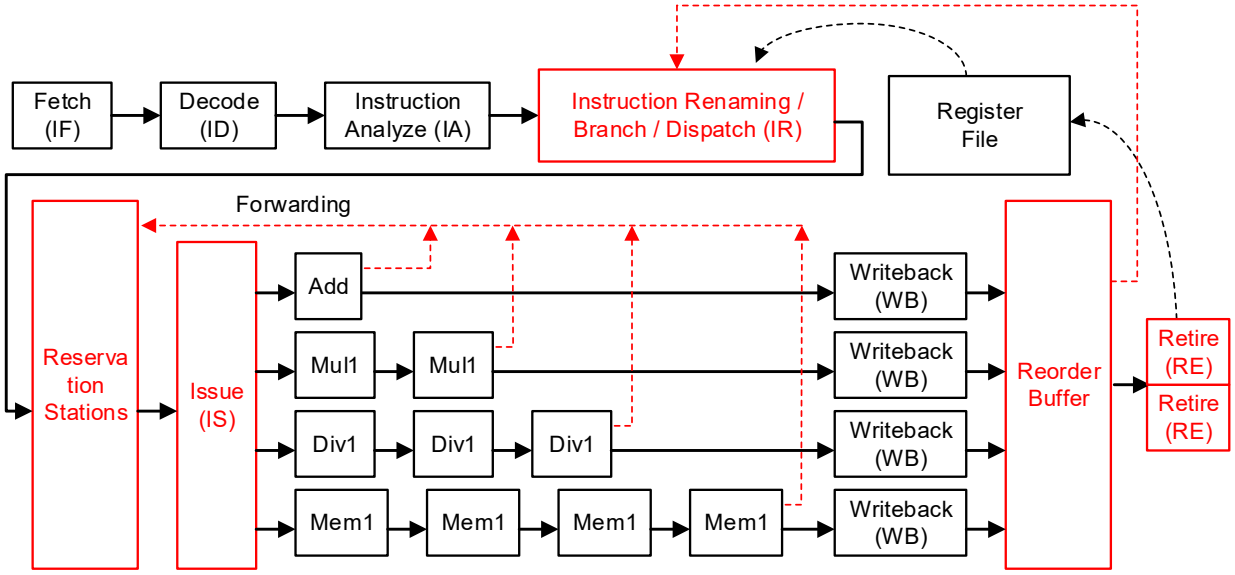
(1) You are allowed to work in a group of up to two students per group, where both members must have an important role in making sure all members are working together. Besides, you are not allowed to form a new group with a new member. Both members must work in a group on previous projects.
(2) All groups must work separately. Cooperation between groups is not allowed.
(3) Sharing of code between groups 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.
(4) You must do all your work in C/C++.
(5) 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. They all have the same software environment.

2. Project Description

In this project, you will construct an out-of-order pipeline.

3. Superscalar Pipeline

Model superscalar pipeline with the following stages.



- 1 stage for fetch units (**IF**): fetch two instructions from memory per clock cycle
- 1 stage for instruction dispatch (**ID**): dispatch two instructions per clock cycle
- 1 stage for instruction analyzers (**IA**): analyze an instruction's dependency
- 1 stage for renaming + branch + dispatch (**IR**): access the register file to read registers and rename, replace PC with the branch destination address if needed, and dispatch instruction
- 1 stage for adders (**Add**): Adder operates on the operands if needed
- 2 stages for multiplier (**Mul1, Mul2**): Multiplier operates on the operands
- 3 stages for divider (**Div1, Div2, Div3**): Divider operates on the operands
- 4 stages for memory (**Mem1, Mem2, Mem3, Mem4**): Access memory
- 1 stage for writeback units (**WB**): Write the result into the reorder buffer
- 2x 1 stage for retire units (**RE**): Retire 2 instructions per cycle in-order

The pipeline supports 4B fixed-length instructions, which have 1B for opcode, 1B for destination, and 2B for two operands. The destination and the left operand are always registers. The right operand can be either register or an immediate value.

Opcode (1B)	Destination (1B)	Left Operand (1B)	Right Operand (1B)
Opcode (1B)	Destination (1B)	Operand (2B)	
Opcode (1B)	None (3B)		

Instruction: The supported instructions have 19 different types, as listed in the following table. The arithmetic operations (add, sub, mul, and div) require at least 1 register operand. The pipeline only supports integer arithmetic operations with 16 integer registers (R0 – R15), each having 4B. All numbers between 0 and 1 are discarded (floor). Instructions' operands are only immediate values, and the destinations are registers; thus, no dependencies among the instructions. The load/store instructions read/write 4B from/to the specified address in the memory map file. Additionally, the pipeline now supports 5 different branch instructions (bez, bgez, blez, bgtz, and bltz).

Mnemonic	Description		
	Destination (1B)	Left Operand (1B)	Right Operand (1B)
	Operand (1B)	Immediate value (2B)	
set	set Rx #Imm (Set an immediate value to register Rx)		
	Register Rx	Immediate value	
add	add Rx Ry Rz (Compute Rx = Ry + Rz)		
	Register Rx	Register Ry	Register Rz
add	add Rx Ry #Imm (Compute Rx = Ry + an immediate valve)		
	Register Rx	Register Ry	Immediate value
sub	sub Rx Ry Rz (Compute Rx = Ry – Rz)		
	Register Rx	Register Ry	Register Rz
sub	sub Rx Ry #Imm (Compute Rx = Ry - an immediate valve)		
	Register Rx	Register Ry	Immediate value
mul	mul Rx Ry Rz (Compute Rx = Ry * Rz)		
	Register Rx	Register Ry	Register Rz

mul	mul Rx Ry #Imm (Compute Rx = Ry * an immediate valve)		
	Register Rx	Register Ry	Immediate value
div	div Rx Ry Rz (Compute Rx = Ry / Rz)		
	Register Rx	Register Ry	Register Rz
div	div Rx, Ry, #Imm (Compute Rx = Ry / an immediate valve)		
	Register Rx	Register Ry	Immediate value
ld	ld Rx #Addr (load the data stored in #Addr into register Rx)		
	Register Rx	Immediate value	
ld	ld Rx Rz (load into register Rx the data stored in the address at Rz)		
	Register Rx		Register Rz
st	st Rx #Addr (store the content of register Rx into the address #Addr. E.g.)		
	Register Rx	Immediate value	
st	st Rx Rz (store the content of register Rx into the address at Rz)		
	Register Rx		Register Rz
bez	bez Rx #Imm (branch to #Imm if Rx==0)		
	Register Rx	Immediate value	
bgez	bgez Rx #Imm (branch to #imm if Rx >= 0)		
	Register Rx	Immediate value	
blez	blez Rx #Imm (branch to #imm if Rx <= 0)		
	Register Rx	Immediate value	
bgtz	bgtz Rx #Imm (branch to #imm if Rx > 0)		
	Register Rx	Immediate value	
bltz	bltz Rx #Imm (branch to #imm if Rx < 0)		
	Register Rx	Immediate value	
ret	ret (exit the current program)		

Pipeline (Dynamic + Diversified): One instruction is fetched at the IF stage, decoded at the ID stage, and analyzed at the IA stage every cycle. The instruction's registers are register-renamed and dispatched in the IR stage in the same cycle. The instructions are dispatched in order. A dispatched instruction takes a free reservation station and waits for their operands. When the instruction becomes ready, it is issued, routed to an associated functional unit, and executed. The issue bandwidth is 4, issuing a maximum of 4 instructions per cycle as long as the functional units are available.

The instructions are executed out of order when their operands are ready. We assume that **if two instructions to the same functional unit are ready at the same cycle, they are issued to the functional unit in order at two successive cycles.** All functional units are pipelined with a different number of stages. Set, add, sub, and branch (bez, bgez, blez, bgtz, bltz, and ret) instructions take the path to the 1-stage adder unit. Mul instructions take the path to the 2-stage multiplier. Div instructions take the path to the 3-stage divider unit. Ld and St instructions take the path to the 4-stage memory unit. At the end of the last execution stage, the execution result is forwarded to the reservation stations. Also, the result is

written to the re-order buffer during the writeback stage. The instructions retire (commit) in order (i.e., the pipeline supports precise exceptions). The retire bandwidth is 2, retiring up to two instructions per cycle.

Branch: The branch condition is checked at the IR stage, and the PC is updated in the same cycle if the branch is taken. All wrongly fetched instructions also must be squashed in the same cycle. After the PC update, a new instruction is fetched at the next cycle. If the branch is not-taken, nothing happens. For simplicity, we assume that all instructions get squashed, and no further instructions are fetched once a ret instruction is executed in the IR stage. The branch instruction is issued and routed to the Add stage but does nothing in that stage.

In-order Execution: The execution of the first 4 stages in the pipeline is centralized and in order, taking one common path until issued. The dependency check is analyzed at the IA stage, and the register read is performed at the IR stage and updated during the RE stage. The register file offers 2 read and 2 write ports, simultaneously handling all memory read/write operations from both stages. The register file allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle. The branch instruction is executed in the IR stage in order, implying that it may stall the pipeline when it is dependent on preceding instructions. The instructions are renamed at the IR stage and dispatched to the reservation stations while reserving their entries in the re-order buffer. The re-order buffer and reservation stations have enough bandwidth to handle all simultaneous read/write operations without structural hazards. The instructions are retired in order from the re-order buffer.

Out-of-order Execution: The pipeline supports out-of-order processing with renaming. The renaming is performed at the IR stage. The pipeline has two new buffers, reservation stations and a re-order buffer. The size of each buffer is as follows.

4 Reservation Stations
8-entry Re-order Buffer

Each reservation station corresponds to one instruction. Reservation stations can keep instruction operands when they are read/forwarded. Once an instruction is dispatched, a reservation station and an entry in the re-order buffer are allocated. If the buffers do not have a free entry, the instruction must be stalled at the IR stage. The instructions stay in the reservation station until their operands are ready. When the instruction is issued (to the execution unit), the occupied entry in the reservation station is freed. The re-order buffer is a queue with a first-in, first-out (FIFO) policy, maintaining the in-order processing. **The re-order buffer also allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle.** Once the instruction is retired, the entry in the re-order buffer is freed.

Memory: The memory map file contains the snapshot of the system's main memory. The file position 0 to 65535 is mapped to the main memory address 0 to 65535. The data at the file position presents the data in the corresponding location of the main memory. Although the programs are in separate files, they are mapped to the memory address 0 to 999. You do not need to copy the programs to the memory map file.

Note #1. You should refer to the course slides for further detail on the reservation station and out-of-order execution.

Note #2. Your simulator must measure the pipeline performance until the time when all the instructions retire, including the ret instruction.

4. Validation and Other Requirements

4.1. Validation requirements

Sample simulation outputs are posted with the following file names: **program1_result.txt**, **program2_result.txt**, **program3_result.txt**, and **program4_result.txt**. You must run your simulator and debug it until it matches the simulation outputs. The print format is already coded in the provided codes, which are the same as in project 3. You must print your registers using the print format on the terminal at the end of each cycle. We also post log files (programX_pipeline.txt) for each program describing every cycle's pipeline status, including the branch predictor's status, to help your debug.

Each simulator must produce a separate output memory map file, **mmap_<program name>.txt**, after the simulation. Assume that your output memory is the same as memory_map.txt initially. You need to update the file whenever memory is written during the simulation. The content in the output memory map file must be correct, matching the content in the provided output memory.

Your output must match both numerically and in terms of formatting because the TAs will "diff" your output with the correct output. You must confirm the 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> <program name>_result.txt"

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 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

You will hand in source code and the TA will compile and run your simulator. As such, 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. A make file is provided with two commands, make and make clean.

The pipeline simulator receives a program name as follows. The below command must generate your outputs.

e.g., sim program_1.txt

5. What to submit

You must hand in only three .c/.h files, main.c, cpu.c, and cpu.h. Do not include other files. Please follow the following naming rule.

LASTNAME_FIRSTNAME_project4.tar.gz

Also, if you work as a group, you must submit another file, group.txt, explaining each member's role in this project. Finally, you must submit a cover page with the project title, the Honor Pledge, and your full name as an electronic signature of the Honor Pledge. A cover page is posted on the project website.

6. Late submissions/Penalties

This project is the last in this course. Submissions must be made on time, and no late submissions are allowed. Considering the project's difficulties, we highly recommend you start this homework as soon as possible.

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.