CS520 Computer Architecture

Project 3 – Spring 2023

Due date: 4/13/2023

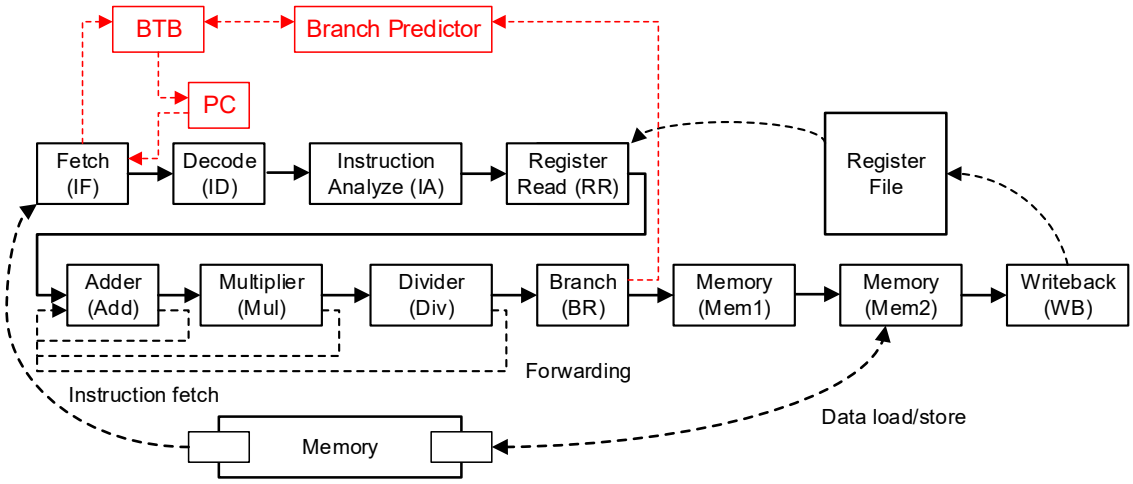
1. RULES

(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 improve the pipeline in project 2 by adding branch instructions.

3. Simple Pipeline



Model simple pipeline with the following three stages.

- 1 stage for fetch (**IF**): fetch an instruction from memory
- 1 stage for decode (**ID**): decode an instruction
- 1 stage for instruction analyze (**IA**): analyze an instruction's dependency
- 1 stage for register read (**RR**): access the register file to read registers

- 1 stage for adder (**Add**): Adder operates on the operands if needed
- 1 stage for multiplier (**Mul**): Multiplier operates on the operands if needed
- 1 stage for divider (**Div**): Divider operates on the operands if needed
- 1 stage for branch (**BR**): PC is replaced with the branch destination address if needed
- 2 stages for memory (**Mem1, Mem2**): Access memory if needed
- 1 stage for register writeback (**WB**): Write the result into the register file

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 (from Project 2): The pipeline has 11 stages. An instruction is fetched at the IF stage. The instruction is decoded at the ID stage, and its dependency is analyzed at the IA stage. The necessary registers are read at the RR stage before the execution stages. After the RR stage, six execution stages follow for different instruction types. The Add stage executes set, add, and sub instructions. The Mul stage executes mul instructions, and the Div stage executes div instructions. At last, the two Mem stages execute ld instructions. The loaded data is only available at the end of the second Mem stage (Mem2). Finally, destination registers are updated at the WB stage. Each stage takes 1 cycle to complete. The memory unit has two read ports and one write port, supporting all simultaneous read/write operations in the pipeline without stalls (No structural hazards).

The instructions may exhibit a true dependency. The pipeline solves such a RAW hazard by stalling the dependent instruction at the RR stage. Note that the register file does not allow the write and the read operations on the same register in the same clock cycle. Once the waiting register is updated, the stalled instruction reads the register next cycle.

The pipeline supports forwarding. The result can be forwarded from the Add, Mul, and Div stages at the end of its cycle. The forwarding in the later stages (Mul and Div) can also forward the earlier stages' results (Add and Mul). No forwarding is provided from other stages, including the end of the Mem stages. The stalled instruction can advance to the Add stage when the dependent data is forwarded.

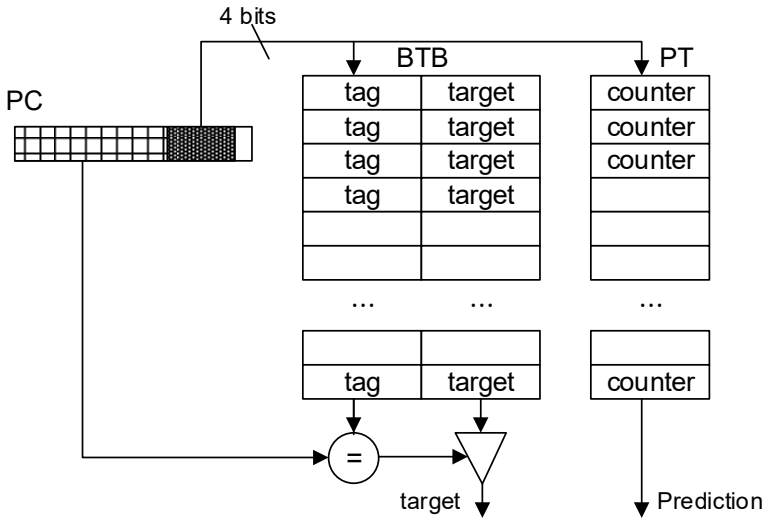
Note that the forwarded data is not stored in this pipeline. When an instruction depends on two instructions, both instructions must forward the data simultaneously.

Branch: The branch target address is computed at the ADD stage. The branch condition is checked at the BR stage, and the PC is updated in the same cycle if the branch is a taken branch. 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 a not-taken branch, nothing happens.

Branch Predictor: The pipeline supports branch prediction. With branch prediction, the PC is updated while a branch instruction is being fetched during the IF stage. Once a branch is mis-predicted, all wrongly fetched instructions must be squashed.

The branch predictor has a branch target buffer (BTB) that has 16 entries, each of which contains a branch's tag and branch target address. The 4 bits of the low-order PC bits are used to form the instruction's index into the BTB. **Note**: discard the lowest two bits of the PC since these are always zero, i.e., use bits 5 through 2 of the PC. The remaining bits, bits 6 through 31 of the PC, become a tag. The BTB is looked up while an instruction is being fetched. The associated target address is retrieved and used to update the PC if there is a tag match. The prediction table is also looked up if the branch is conditional. The BTB is updated at the end of the BR stage with an actual branch outcome.

The branch predictor has a prediction table (PT) with 16 entries, each containing a 3-bit counter for prediction. The 4 bits of the low-order PC bits are used to form the prediction table index. You must discard the lowest two bits of the PC since these are always zero, i.e., use bits 5 through 2 of the PC.



The instruction's index into the prediction table is used to get the branch's counter from the prediction table. If the counter value is greater than or equal to 4, then the branch is predicted taken; else, it is predicted not-taken. When it is taken, the instruction located at the address specified in the branch's operand is fetched next cycle. The prediction table is updated at the end of the BR stage with an actual branch outcome, not affecting the fetched branch's prediction at the same cycle. All counters are initialized to 3 when a new program starts. Counters in the pattern table are global and shared by all branch instructions.

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.

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 2. 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 three c files, main.c, cpu.c, and cpu.h. Please follow the following naming rule.

LASTNAME_FIRSTNAME_project3.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 will be a part of all the following projects. That's why this project's portion in final grading is small. We allocate a long enough time, allowing you to flexibly manage your time to work on this project, although we highly recommend you start this homework as soon as possible.

On the other hand, we, therefore, have limited flexibility in the project's due date because our next project will start immediately after this one. Late submission is only allowed the first five days after the due date, with a penalty. Also, no extension will be allowed.

Various deductions (out of 100 points):
-8 points for each date late during the first 5 days.

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.