CS520 Computer Architecture

Project 3 – Spring 2024

Due date: 5/4/2024

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 C programming language. C++ is not allowed.
(4) You are not allowed to add more C files and libraries.
(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.

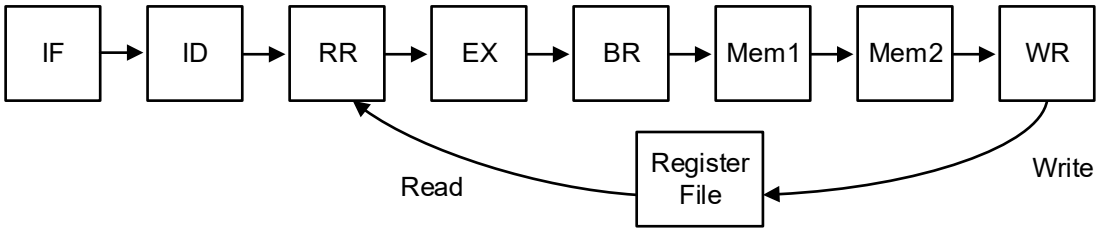
2. Project Description

In this project, we will construct an 8-stage processor simulator that simulates test programs.

3. 8-stage Processor Simulator

This new processor is an updated version of the processor that was implemented in the prior project. This processor now has 8 pipeline stages as follows. Each pipeline stage takes 1 clock cycle to complete.

- 1 stage for instruction fetch (IF)
- 1 stage for instruction decode (ID)
- 1 stage for instruction register read (RR)
- 1 stages for arithmetic instructions (EX)
- 1 stages for branch instruction (BR)
- 2 stages for memory operation (Memory stages: Mem1, Mem2)
- 1 stage for write back (WB)



The instruction formats are the same as those in the prior project.

Type	Opcode	Operand 1	Operand 2	Operand 3
Inst-3B	Opcode (1 byte)	Register (1 byte)	Register (1 byte)	-
Inst-4B	Opcode (1 byte)	Register (1 byte)	Register (1 byte)	Register (1 byte)
	Opcode (1 byte)	Register (1 byte)	Immediate (2 bytes)	
Inst-5B	Opcode (1 byte)	Register (1 byte)	Register (1 byte)	Immediate (2 bytes)
Inst-6B	Opcode (1 byte)	Register (1 byte)	Immediate (2 bytes)	Immediate (2 bytes)

Like the prior processors, the 1B opcode has the following format.

2-bit instruction length	2-bit instruction group	2-bit instruction type	2-bit reserved

In this format, the first four bits indicate as follows.

2-bit instruction length:

- 00: 4-byte instruction
- 01: 5-byte instruction
- 10: 6-byte instruction
- 11: 3-byte instruction

2-bit instruction group:

- 00: non-arithmetic instruction
- 01: arithmetic instruction

Instruction: The processor supports the same instructions of the prior project: 4 arithmetic instructions (add, sub, mul, and div), 2 memory instructions (ld and sd), and 3 others (set, branch, and ret). Arithmetic instructions have no updates, supporting 3 different formats with varying lengths. New memory instructions allow a user to access memory. The ld instruction loads 4 bytes from the specified address. The sd instruction stores 4 bytes in a register to the specified address. The processor supports little-endian. The processor also supports 3 different branch instructions: bez, bgtz, and bltz. As in the prior simulator, the processor only supports integer arithmetic operations with 64 integer registers (R0 – R63), each with a 4B size. All numbers between 0 and 1 are discarded (floor).

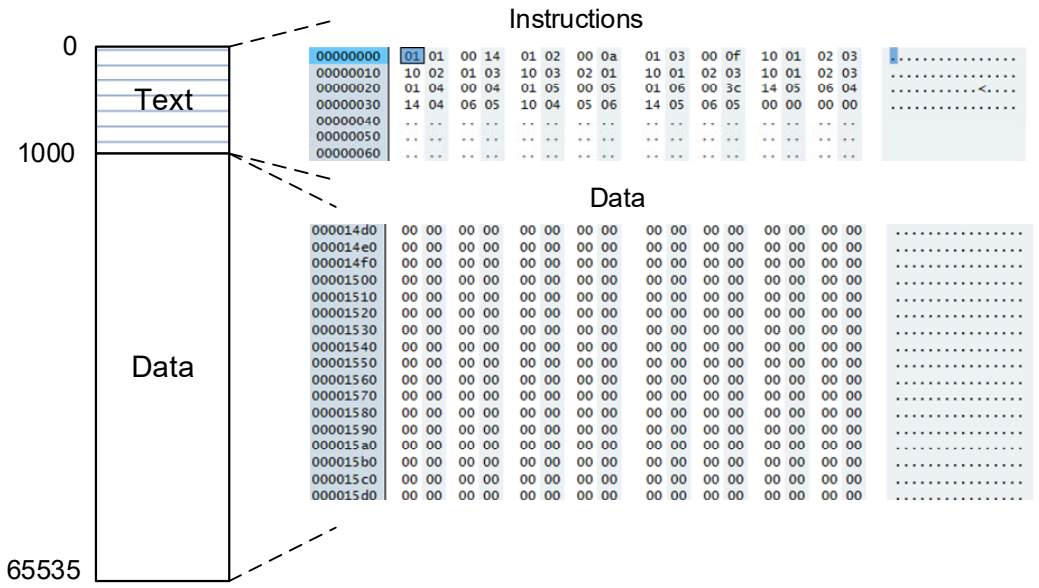
Mnemonic (1B)	Opcode	Description		
		Destination (1B)	Left Operand (1B or 2B)	Right Operand (1B or 2B)
set	0x01	set Rx, #Imm	Set an immediate value to register Rx	
		Register Rx	Immediate value (2B)	
add	0x10	add Rx, Ry, Rz	Compute Rx = Ry + Rz	
		Register Rx	Register Ry (1B)	Register Rz (1B)
add	0x50	add Rx, Ry, #Imm	Compute Rx = Ry + immediate valve	
		Register Rx	Register Ry (1B)	Immediate value (2B)
add	0x90	add Rx, #imm, #Imm	Compute Rx = immediate valve + immediate valve	
		Register Rx	Immediate value (2B)	Immediate value (2B)
sub	0x14	sub Rx, Ry, Rz	Compute Rx = Ry – Rz	
		Register Rx	Register Ry (1B)	Register Rz (1B)

sub	0x54	sub Rx, Ry, #Imm	Compute Rx = Ry - immediate valve	
		Register Rx	Register Ry (1B)	Immediate value (2B)
sub	0x94	sub Rx, #imm, #Imm	Compute Rx = immediate valve - immediate valve	
		Register Rx	Immediate value (2B)	Immediate value (2B)
mul	0x18	mul Rx, Ry, Rz	Compute Rx = Ry × Rz	
		Register Rx	Register Ry (1B)	Register Rz (1B)
mul	0x58	mul Rx, Ry, #Imm	Compute Rx = Ry × immediate valve	
		Register Rx	Register Ry (1B)	Immediate value (2B)
mul	0x98	mul Rx, #imm, #Imm	Compute Rx = immediate valve × immediate valve	
		Register Rx	Immediate value (2B)	Immediate value (2B)
div	0x1C	div Rx, Ry, Rz	Compute Rx = Ry ÷ Rz	
		Register Rx	Register Ry (1B)	Register Rz (1B)
div	0x5C	div Rx, Ry, #Imm	Compute Rx = Ry ÷ immediate valve	
		Register Rx	Register Ry (1B)	Immediate value (2B)
div	0x9C	div Rx, #imm, #Imm	Compute Rx = immediate valve ÷ immediate valve	
		Register Rx	Immediate value (2B)	Immediate value (2B)
bez	0x04	bez Rx, #Imm	branch to #Imm if Rx==0	
		Register Rx	Immediate value (2B)	
bgtz	0x05	bgtz Rx, #Imm	branch to #imm if Rx > 0	
		Register Rx	Immediate value (2B)	
bltz	0x06	bltz Rx, #Imm	branch to #imm if Rx < 0	
		Register Rx	Immediate value (2B)	
ld	0xC8	ld Rx, Ry	Load into register Rx the data stored in the address at Ry	
		Register Rx	Register Ry	-
sd	0xCC	sd Rx, Ry	Store the content of register Rx into the address at Ry	
		Register Rx	Register Ry	-
ret	0x00	ret (exit the current program)		
		0x00	0x00	0x00

Branch: The branch target address is computed at the EX stage, the branch condition is evaluated at the BR stage, and the PC is updated at the same cycle. After the PC update, the pipeline squashes all subsequent instructions after the branch instruction if the instruction is a taken branch. A new instruction is fetched at the next cycle. Overall, the branch instruction is processed across three cycles, i.e., clock cycle 1: calculate target address (EX), clock cycle 2: condition check + pc update (BR), and clock cycle 3: fetch a new instruction.

Dependency: The dependency check (RAW) and register read are done at the RR stage. If there is a dependency, the instruction must wait at the RR stage until the result becomes available. The register read requires one cycle. Since there is no forwarding, the instruction can read a register at the next cycle after the register is updated.

Memory: The memory map file (mmap#.in) contains a snapshot of the system's 64KB main memory, indicating that all data and instructions are encoded in a binary code format. 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. The programs are mapped to the text area of the memory, address 0 to 999. The instructions are stored in the memory in order starting from address 0. The other memory space is reserved for data.



4. Validation and Other Requirements

4.1. Validation requirements

In this project, you update your previous submission for Project 2.

Your simulator must print all the register values, execution cycles, and the number of instruction types on the screen. Your simulator must also generate correct memory map files (mem#.out). Sample outputs and memory maps are provided on the websites. Your simulator does not need to create log files (test#.log); those are provided to help your debugging.

You must run your simulator and debug it until it matches the simulation outputs. Your simulator must print the final contents in the register and performance results correctly.

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> <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 a whitespace. Both your outputs must be the same as the solution.

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

4) Since the correct answers are already provided, we will treat submissions that print exact outputs without correct implementations as cheating.

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 can also access the machine remotely with the same environment at remote.cs.binghamton.edu via SSH. A make file is provided with two commands: make and make clean.

The simulator receives two arguments: input and output memory maps. The first argument is the input memory map, and the second is the output. The below command must generate your simulation output. The simulation results must be printed on the terminal (standard output).

e.g., **sim mem1.in (input) mem1.out (output)**

5. What to submit

You must hand in two c files, cpu.c, and cpu.h. Please do not include other files including your outputs. Please follow the following naming rule.

LASTNAME_FIRSTNAME_project3.tar.gz

You also need to 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

Late submission is not allowed **the first 4 days** after the due date, with a penalty. Also, no extension will be allowed.

Various deductions (out of 100 points):
-10 points for each date late during the first 4 days.

Up to -20 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. Note that we are not only using a tool. We check your codes for ourselves and flag the codes that look suspicious as cheating.