CS520 Computer Architecture

Project 1 – Spring 2024

Due date: 3/13/2024

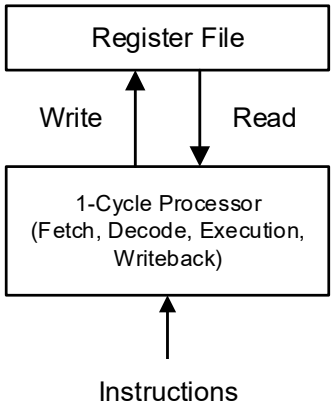
1. RULES

(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, you will construct a 1-cycle processor simulator that simulates test programs.

3. 1-Cycle Processor Simulator



The 1-cycle processor fetches, decodes, and executes one instruction every cycle with 64 integer registers. The processor supports 4 to 6 bytes varying-length instructions, which consist of an opcode and operands. The operand is either a 2B immediate number or a register address. The first operand specifies a destination register that contains the result if the instruction produces a result. The other operands contain the necessary numbers for the conducted operation. The processor supports 3 different instruction lengths as follows.

Type	Opcode	Operand 1	Operand 2	Operand 3
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)

The 1B opcode has the following format.

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

For example, an opcode 0001 1000 (0x18) indicate:

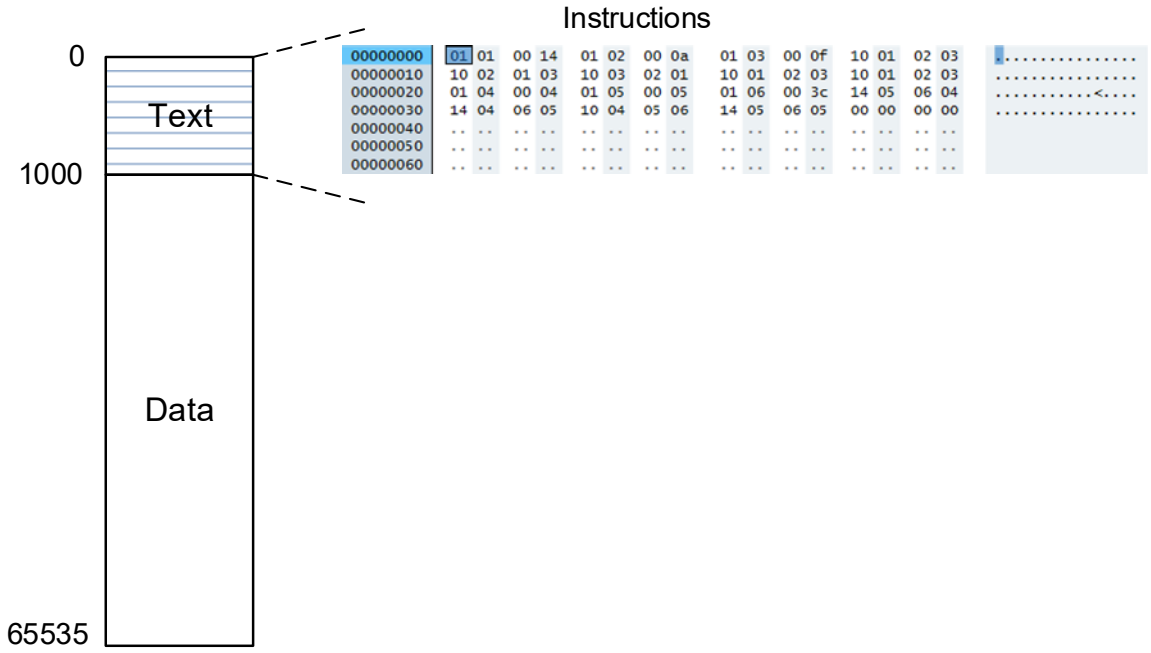
00: 4B instructions
01: arithmetic instruction
10: multiplication
00: reserved

Instruction: The processor supports 6 types of instructions, 4 arithmetic instructions (add, sub, mul, and div) and 2 others (set and ret). Each arithmetic instruction has 3 different formats with varying lengths. 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 valve (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 valve (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 valve (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 valve (2B)	Immediate value (2B)
ret	0x00	ret (exit the current program)		
		0x00	0x00	0x00

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 0.



4. Validation and Other Requirements

4.1. Validation requirements

Your simulator should print all the register values, execution cycles, and the number of instruction types on the screen. Sample outputs 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 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 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 one argument: a memory map. The input memory map contains a program that it simulates. 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)**

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_project1.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 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 -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.