Performance Modelling - RISC-V processor

This project will require you to implement cycle-accurate simulators of a 32-bit RISC-V processor in C++ or Python. The skeleton code for the assignment is given in file (NYU_RV32I_6913.cpp or NYU_RV32I_6913.py).

The simulators should take in two files as inputs: imem.text and dmem.txt files
The simulator should give out the following:

- cycle by cycle state of the register file (RFOutput.txt)
- Cycle by cycle microarchitectural state of the machine (StateResult.txt)
- Resulting dmem data after the execution of the program (DmemResult.txt)

The imem.txt file is used to initialize the instruction memory and the dmem.txt file is used to initialize the data memory of the processor. Each line in the files contain a byte of data on the instruction or the data memory and both the instruction and data memory are byte addressable. This means that for a 32 bit processor, 4 lines in the imem.txt file makes one instruction. Both instruction and data memory are in "Big-Endian" format (the most significant byte is stored in the smallest address).

The instructions to be supported by the processor are categorized into the following types:

Bit	*31*	*25, 24*	*20, 19*	*15, 14*	*12, 11*	*7, 6*	*0*
R-type	funct7 (7 bits)	rs2 (5 bits)	rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	Opcode (7 bits)	
I-type	imm[11:0]		rs1	funct3	rd	Opcode	
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	Opcode	
B-type	imm[12, 10:5]	rs2	rs1	funct3	imm[4:1, 11]	Opcode	
U-type	imm[31:12]				rd	Opcode	
J-type	imm[20, 10:1, 11, 19:12]				rd	Opcode	

The simulator should support the following set of instructions.

Mnemonic	Type	Full Name	Psuedocode	Details
ADD	**R**	**Addition**	**rd = rs1 + rs2**	**Store the result of rs1 + rs2 in register rd.**
SUB	**R**	**Subtraction**	**rd = rs1 - rs2**	**Store the result of rs1 - rs2 in register rd.**
XOR	**R**	**Bitwise XOR**	**rd = rs1 ^ rs2**	**Store the result of rs1 ^ rs2 in register rd.**
OR	**R**	**Bitwise OR**	**rd = rs1 | rs2**	**Store the result of rs1 | rs2 in register rd.**

AND	R	Bitwise AND	rd = rs1 & rs2	Store the result of rs1 & rs2 in register rd.
ADDI	I	Add Immediate	rd = rs1 + sign_ext(imm)	Add the sign-extended immediate to register rs1 and store in rd. Overflow bits ignored.
XORI	I	XOR Immediate	rd = rs1 ^ sign_ext(imm)	Bitwise XOR the sign-extended immediate to register rs1 and store result in rd.
ORI	I	OR Immediate	rd = rs1 | sign_ext(imm)	Bitwise OR the sign-extended immediate to register rs1 and store result in rd.
ANDI	I	AND Immediate	rd = rs1 & sign_ext(imm)	Bitwise AND the sign-extended immediate to register rs1 and store result in rd.
JAL	J	Jump and Link	rd = PC + 4; PC = PC + sign_ext(imm)	Jump to PC = PC + sign_ext(imm) and store the current PC + 4 in rd.
BEQ	B	Branch if equal	PC = (rs1 == rs2)? PC + sign_ext(imm) : PC + 4	Take the branch (PC = PC + sign_ext(imm)) if rs1 is equal to rs2.
BNE	B	Branch if not equal	PC = (rs1 != rs2)? PC + sign_ext(imm) : PC + 4	Take the branch (PC = PC + sign_ext(imm)) if rs1 is not equal to rs2.
LW	I	Load Word	rd = mem[rs1 + signa(imm)][31:0]	Load 32-bit value at memory address [rs1 + signPext(imm)] and store it in rd.
SW	S	Store Word	data[rs1 + sign_ext(imm)][31:0] = rs2	Store the 32 bits of rs2 to memory address [rs1 value + sign_ext(imm)].
HALT	-	Halt execution		

Instruction encoding:

Mnemonic	Bit Fields						
	31:27	26:25	24:20	19:15	14:12	11:7	6:0
ADD	0000000		rs2	rs1	000	rd	0110011
SUB	0100000		rs2	rs1	000	rd	0110011
XOR	0000000		rs2	rs1	100	rd	0110011

OR	**0000000**	**rs2**	**rs1**	**110**	**rd**	**0110011**
AND	**0000000**	**rs2**	**rs1**	**111**	**rd**	**0110011**
ADDI	**imm[11:0]**		**rs1**	**000**	**rd**	**0010011**
XORI	**imm[11:0]**		**rs1**	**100**	**rd**	**0010011**
ORI	**imm[11:0]**		**rs1**	**110**	**rd**	**0010011**
ANDI	**imm[11:0]**		**rs1**	**111**	**rd**	**0010011**
JAL	**imm[20|10:1|11|19:12]**				**rd**	**1101111**
BEQ	**imm[12|10:5]**	**rs2**	**rs1**	**000**	**imm[4:1|11]**	**1100011**
BNE	**imm[12|10:5]**	**rs2**	**rs1**	**001**	**imm[4:1|11]**	**1100011**
LW	**imm[11:0]**		**rs1**	**000**	**rd**	**0000011**
SW	**imm[11:0]**	**rs2**	**rs1**	**010**	**imm[4:0]**	**0100011**
HALT	**x**	**x**	**x**	**xxx**	**x**	**1111111**

The simulator should have the following five stages in its pipeline:

- **Instruction Fetch:** Fetches instruction from the instruction memory using PC value as address.
- **Instruction Decode/ Register Read:** Decodes the instruction using the format in the table above and generates control signals and data signals after reading from the register file.
- **Execute:** Perform operations on the data as directed by the control signals.
- **Load/ Store:** Perform memory related operations.
- **Writeback:** Write the result back into the destination register. Remember that R0 in RISC-V can only contain the value 0.

Each stage must be preceded by a group of flip-flops to store the data to be passed on to the next stage in the next cycle. Each stage should contain a nop bit to represent if the stage should be inactive in the following cycle.

The simulator must be able to deal with two types of hazards.

1. RAW Hazards: RAW hazards are dealt with using either only forwarding (if possible) or, if not, using stalling + forwarding. Use EX-ID forwarding and MEM-ID forwarding appropriately.
2. Control Flow Hazards: The branch conditions are resolved in the ID/RF stage of the pipeline.

The simulator deals with branch instructions as follows:

1. Branches are always assumed to be NOT TAKEN. That is, when a beq is fetched in the IF stage, the PC is speculatively updated as PC+4.
2. Branch conditions are resolved in the ID/RF stage.

3. If the branch is determined to be not taken in the ID/RF stage (as predicted), then the pipeline proceeds without disruptions. If the branch is determined to be taken, then the speculatively fetched instruction is discarded and the nop bit is set for the ID/RR stage for the next cycle. Then the new instruction is fetched in the next cycle using the new branch PC address.

Tasks:

1) Draw the schematic for a single stage processor and fill in your code in the to run the simulator. (20 points)
2) Draw the schematic for a five stage pipelined processor and fill in your code to run the simulator. The processor should be able ot take care of RAW and control hazards by stalling and forwarding. (20 points)
3) Measure and report average CPI, Total execution cycles, and Instructions per cycle for both these cores by adding performance monitors to your code. (Submit code and print results to console or a file.) (5 points)
4) Compare the results from both the single stage and the five stage pipelined processor implementations and explain why one is better than the other. (5 points)
5) What optimizations or features can be added to improve performance? (Extra credit 1 point)

Your work will be evaluated against the 10 test cases, 3 of which will be revealed one week before the deadline. (50 points - 5 points each)

Useful References:

- More details on the full ISA specification can be found at https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf
- bitset library for C++: https://en.cppreference.com/w/cpp/utility/bitset
- g++: https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/G_002b_002b-and-GCC.html
- python: https://www.python.org/downloads/