Simple Scalar Pipelined Processor Design

Problem Statement

1 Objective

- To understand the working of a scalar pipelined processor by simulating the different components at the software level.
- Code must be written in C++ following the folder structure given in this pdf.
- **Team Size**: Maximum of 2 members.

2 Processor Configuration

Consider a scalar pipelined processor with a 256B instruction cache (I$) and a 256B data cache (D$), both having a read port and a write port each, and both are direct-mapped caches. Assume that both instruction and data caches are perfect, which means there won't be any cache misses in these caches. Assume the processor has a register file (RF) with sixteen 8-bit registers named R0, ..., R15. Note that R0 always stores the value '0'. The register file has two read ports and a write port. Negative numbers are stored in 2's complement form.

3 Pipelined Processor

3.1 Instructions

We consider a modified *Reduced Instruction Set Computer (RISC)* processor. This architecture has the following instruction set.

- Arithmetic Instructions
  - `ADD rd rs1 rs2` — Addition : `rd` ← `rs1 + rs2`
  - `SUB rd rs1 rs2` — Subtraction : `rd` ← `rs1 - rs2`
  - `MUL rd rs1 rs2` — Multiplication: `rd` ← `rs1 * rs2`
  - `INC r` — Increment : `r` ← `r + 1`

  In the increment instruction, the last 8 bits are discarded.
- Logical Instructions
  - `AND rd rs1 rs2` — Bit-wise AND : `rd` ← `rs1 & rs2`
  - `OR rd rs1 rs2` — Bit-wise OR : `rd` ← `rs1 | rs2`
  - `XOR rd rs1 rs2` — Bit-wise XOR : `rd` ← `rs1` ⊕ `rs2`
  - `NOT rd rs` — Bit-wise NOT : `rd` ← ∼ `rs`
- Shift Instructions
  - `SLLI rd rs1 imm(4)` — Shift Logical Left Immediate: `rd` ← `rs1 << imm(4)`
  - `SRLI rd rs1 imm(4)` — Shift Logical Right Immediate: `rd` ← `rs1 >> imm(4)`
- Memory Instructions
  - `LI rd imm(8)` — Load Immediate : `rd` ← `imm(8)`

- LD rd rs1 imm(4) — Load Memory : `rd` $\leftarrow$ `[rs1 + imm(4)]`
- ST rd rs1 imm(4) — Store Memory : `[rs1 + imm(4)]` $\leftarrow$ `rd`

Note: `imm(4)` is a 4-bit immediate value and `imm(8)` is a 8-bit immediate value, both signed

- Control InstructionsArise from pipelining of branches and other instructions that change the Program Counter (PC). For example, in a conditional Jump instruction, till the condition is evaluated, the new PC can take either the incremented PC value or the address accessed in that instruction. To avoid this, we stall the pipeline for two cycles. When a conflict is encount

  - JMP L1 — Unconditional jump by L1 instructions, last 4 bits are discarded
  - BEQZ rs L1 — Jump by L1 instructions if `rs` is zero (0).

  L1 is offset from the current program counter (PC). This is called PC-relative addressing. L1 is an 8-bit number represented in 2's complement format.

- Halt instruction

  - HLT — Program terminates, and the last 12 bits are discarded.

3.2 Pipeline

We consider a five-stage instruction pipeline: IF-ID-EX-MEM-WB. For store instruction, the WB stage remains idle. For ALU instructions, the MEM stage remains idle. The processor is pipelined at the instruction level. The instructions are to be of a fixed length of 2 bytes each.

1. **Instruction Fetch Cycle (IF)**:

$$\texttt{IR} \leftarrow \texttt{I\$[PC];}$$
$$\texttt{PC} \leftarrow \texttt{PC + 2;}$$

   Operation: Send out the Program Counter (PC) and fetch the instruction from the instruction cache (I$ cache) into the Instruction Register (IR); increment the PC by 2 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; the Register PC is updated to point to the address of the next instruction. The above describes the fetching of one instruction at a time. You should fetch one instruction at any time in the scalar pipeline architecture.

2. **Instruction Decode Cycle (ID)**:

   Operation: Decode the instruction; identify the opcode, source registers, and destination register; establish the input/output dependency information. Since there are 16 registers, 4-bit encoding is used for registers. R0 is encoded as 0000, R1 as 0001, etc. All the fields are at a fixed location in the instruction format. The 4-bit opcode is considered in the instructions as shown below:

Opcode	**Encoding**	**Opcode**	**Encoding**
ADD	0000	SLLI	1000
SUB	0001	SRLI	1001
MUL	0010	LI	1010
INC	0011	LD	1011
AND	0100	ST	1100
OR	0101	JMP	1101
XOR	0110	BEQZ	1110
NOT	0111	HLT	1111

   In the case of `LD/ST` instructions, the least significant four bits provide the offset, which can be a positive/negative number. In this pipeline stage, we also read source operands from the RF and the outputs of the general-purpose registers are read into two temporary registers (`A` and `B`) for use in later clock cycles.

$$\texttt{A} \leftarrow \texttt{RF[rs1];}$$
$$\texttt{B} \leftarrow \texttt{RF[rs2];}$$

3. **Execution/Effective Address Cycle (EX)**: The ALU operates on the operands prepared in the prior cycle, performing one of the following three functions depending on the instruction type.

- **Memory Reference: (LD and ST)**:

  `ALUOutput ← A + imm(4);`

  Operation: The ALU adds the offset (`imm(4)`) with the contents of `A` fetched in the earlier cycle to form the effective address and places the result into the temporary register `ALUOutput`.

- **Register-Register ALU Instruction**:

  `ALUOutput ← A op B;`

  Operation: The ALU performs the operation specified by the opcode on the values in registers A and B. The result is placed in the temporary register ALUOutput.

- **Branch**:

  `ALUOutput ← PC + (L1 * 2);`
  `Cond ← (A==0);`

  Operation: The ALU adds the PC to the sign-extended immediate value in L1, which is shifted left by 1 bit to create a 2-byte offset to compute the address of the branch target. Register A, read in the prior cycle, is checked to determine whether the branch has been taken. Since we are considering only one form of branch (BEQZ), the comparison is against 0.

4. **Memory Access Cycle (MEM)**:

```
LOAD:     LMD ← D$[ALUOutput];
STORE:    D$[ALUOutput] ← B;
```

   Operation: Access data cache, if needed. If the instruction is a load, data returns from the data cache and is placed in the `LMD` (load memory data) register; if it is a store, the data from the `B` register is written into the data cache. In either case, the address used is the one computed during the prior cycle and stored in the register `ALUOutput`.

5. **Write-Back Cycle (WB)**:

   - Register-Register ALU instruction:
     `RF[rd] ← ALUOutput;`
   - Load Instruction:
     `RF[rd] ← LMD`

   Operation: Write the result into the registers file, whether it comes from the memory system (which is in `LMD`) or from the ALU (which is in `ALUOutput`).

3.3 Pipelining Hazards

Hazards are situations that prevent the next instruction in the instruction stream from getting executed in its designated clock cycle. Hazards may stall the pipeline. We consider data hazards and control hazards.

1. **Read-After-Write (RAW) Hazards**:
   Consider the instruction sequence given below.

```
AND R1 R2 R3
SUB R4 R1 R5
```

   The content of R1, produced by the ADD instruction, is required for the SUB instruction to proceed.

2. **Control Hazards**:
   Arise from pipelining of branches and other instructions that change the Program Counter (PC). For example, in a conditional Jump instruction, till the condition is evaluated, the new PC can take either the incremented PC value or the address accessed in that instruction. To avoid this, we stall the pipeline for two cycles.

When a conflict is encountered, all the instructions before the stalled instructions need to continue, and all the instructions after the stalled instructions need to be stalled.

3.4 Submission

3.4.1 Input Format

- **ICache.txt**: contents of Instruction Cache.
- **DCache.txt**: contents of Data Cache.
- **RF.txt**: contents of Register File.

Sample:

```
81
21
83
22
04
13
.
.
. (total 256 lines)
```

ICache.txt

```
01
02
03
04
05
06
.
.
. (total 256 lines)
```

DCache.txt

```
00
01
10
4f
02
83
.
.
. (total 16 lines)
```

RF.txt

3.4.2 Output Format

- **DCache.txt**: contents of Data Cache reflecting all changes applied during execution.
- **Output.txt**: contains the stats described below.

Assuming that each pipeline stage takes 1 cycle, execute a given program and report:

1. Number of instructions executed.
2. Number of instructions of each type.
3. CPI (clock cycles per instruction)
4. Number of stalls and reason for each stall (eg: RAW dependency).

Sample:

```
Total number of instructions executed:8
Number of instructions in each class
Arithmetic instructions              :3
Logical instructions                 :1
Shift instructions                   :0
Memory instructions                  :3
Control instructions                 :0
Halt instructions                    :1
Cycles Per Instruction               :2.25
Total number of stalls               :6
Data stalls (RAW)                    :6
Control stalls                       :0
```

Output.txt

3.4.3 Directory Structure

Follow the directory structure given below:

```
<ROLLNO>_LAB8
├── input
│   ├── 'ICache.txt'
│   ├── 'DCache.txt'
│   └── 'RF.txt'
├── output
│   ├── 'DCache.txt'
│   └── 'Output.txt'
├── src
│   ├── blah
│   └── blah
├── 'main.cpp'
└── 'README.md'
```

- The *src* folder is optional and can be used if you wish to modularize your code into different files.
- The *README.md* file should clearly mention the instructions to run and execute the code.