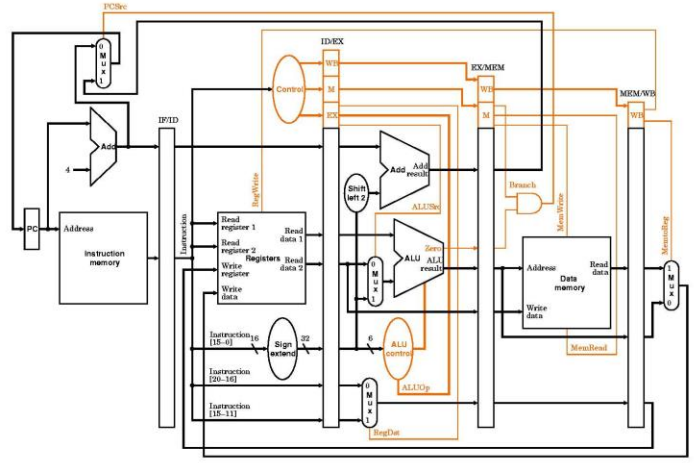
Pipelining Simulator

CMPUT 229 LAB 6

Saumya Patel

GOAL

- Simulate the operation of a 5-stage pipelined RISC-V CPU in software.
- Allow users to step through program execution one instruction at a time.
- Visualize instruction flow through each pipeline stage:
  - Instruction Fetch (IF)
  - Instruction Decode/Register Fetch (ID)
  - Execute/Address Calculation (EX)
  - Memory Access (MEM)
  - Write Back (WB)



GOAL (Continued)

- Enable observation of how instructions overlap in the pipeline during parallel execution.
- Demonstrate how the pipeline handles different types of hazards:
  - **Data hazards**: when one instruction depends on the result of another.
  - **Control hazards**: typically caused by branch instructions.
  - **Structural hazards**: due to insufficient hardware resources (described but not simulated).
- Model cache behavior for memory instructions (loads and stores):
  - Check for cache hits and misses.
  - Introduce memory delays (stalls) on a cache miss to show impact on pipeline execution.

- Collect and display performance metrics throughout simulation.
- Allow analysis of the performance impact of hazards, stalls, and cache behavior.

BACKGROUND

- Pipelining:
  - Technique used in computer architecture to improve CPU performance
  - Allows multiple instructions to be executed simultaneously
  - Divides processing into stages with each performing specific operations
  - Output of one stage becomes input for the next stage

Background (Continued)

- Five pipeline stages:
  - **IF (Instruction Fetch**)
    - Instruction is fetched from memory
  - **ID (Instruction Decode/Register Fetch)**
    - Instruction is decoded and registers read
  - **EX (Execute/Address Calculation)**
    - Operation performed or effective address calculated
  - **MEM (Memory Access)**
    - Data memory accessed for load/store instructions
  - **WB (Write Back)**
    - Result written back to register file

Background (Continued)

- Pipeline hazards:
  - **Data hazards**
    - When instruction depends on result of previous instruction not yet completed
  - **Control hazards**
    - When pipeline makes decision based on unresolved branch instruction
  - **Structural hazards**
    - When multiple instructions require same resource simultaneously

Memory Mapped Register and Register Table

- Register Table (or register file) = array of 32 words.
- Each word represents a register.
- If the address of the Register table is 0x40004000 then the first register ie x0 has the address 0x40004000, register x1 has the value 0x40004004 , etc
- Register Table stored in the global variable ***RegisterTable***

The Simulator

- The entire simulator requires using and calling the 5 stages of the pipeline in a reverse order.
- Example

```
WriteBack();
Memory();
Execution();
Decode();
InstructionFetch();
```

- Simulate Control Hazards like handling branches along with data hazards.
- Modeling cache behaviors and introducing cache delays for cache misses.

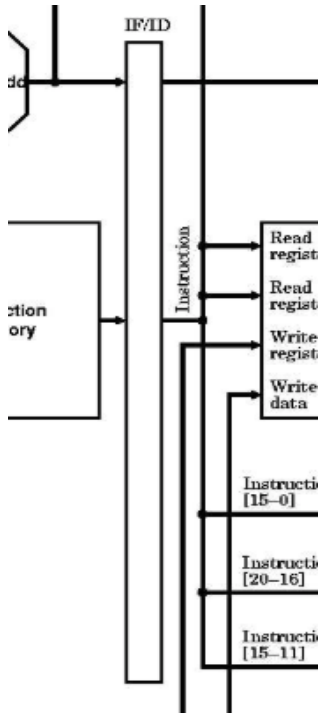
Pipeline Latches - Overview

- Pipeline latches hold instruction information between stages.
- Each latch stores:
  - ***Instn*** (32 bit)
  - ***Valid*** bit (shows if actual instruction is in latch)
  - Instruction-specific fields required for the next pipeline stage

IF/ID Latch

```
struct ifidlatch {
  unsigned instn;
  int      pc;
  byte     valid;
};
```

- Purpose: Holds instruction just fetched, ready for decode.
- Holds:
  - Fetched instruction (raw 32 bits)
  - Program Counter (PC) of fetched instruction
  - Valid bit (1 if latch holds a real instruction, 0 otherwise)
- Note: Enables step-wise handoff of instruction to decoding stage.



ID/EX Latch

- Purpose: Holds decoded information, ready for execution.
- Holds:
  - Instruction
  - Instruction type (R, I, S, SB, UJ)
  - Register indices: `rs1`, `rs2`, `rd`
  - Immediate value (sign-extended, format-dependent)
  - funct3 field
  - Funct7 field
  - Control signals
  - Program Counter (PC)
  - Valid bit

```
struct ID_EX_LATCH {
  unsigned   instn;
  int        instn_type;
  int        pc;
  int        rd;
  int        rs1;
  int        rs2;
  int        imm;
  int        funct3;
  int        funct7;
  byte       valid;
  byte       control_signals;
};
```

EX/MEM Latch

- Purpose: Holds execution/ALU results and address calculation for memory access.
- Holds:
  - Result from ALU (arithmetic/logical result, or effective memory address)
  - Second source register (rs2)
  - Register destination (rd)
  - Branch evaluation (taken/not taken) (if instruction is branch)
  - Value to write to memory (for store instructions)
  - Control signals
  - Valid bit

```
struct EX_MEM_LATCH {
  unsigned  instn;
  int       instn_type;
  int       rd;
  int       rs2;
  int       alu_result;
  byte      valid;
  byte      control_signals;
};
```

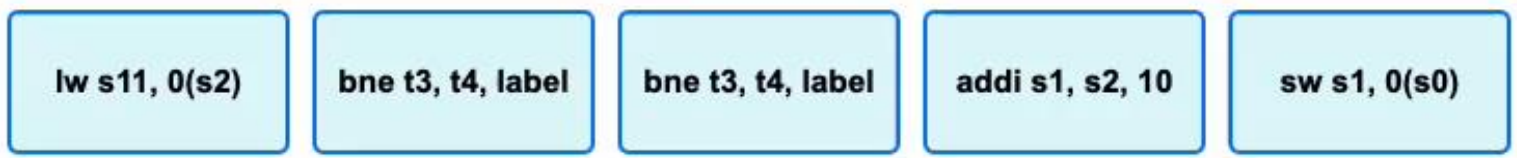
MEM/WB Latch

- Purpose: Holds results of memory or ALU operation for final write-back.
- Holds:
  - Data from memory (for load)
  - Data from ALU (if not a load)
  - Destination register (rd)
  - Control signals
  - Valid bit
  - InstructionType

```
struct memwblatch {
  int       rd;
  int       alu_out;
  int       mem_out;
  byte      valid;
  unsigned  instn;
  byte      control_signals;
  int       instn_type;
};
```

Inserting into the pipeline

- A helper function called InsertIntoPipeline gives a detailed overview.
- "CurrentPipeline" is used to store the current state of the pipeline at any given cycle. It is an array of 5 words representing the 5 stages of the pipeline.
- When an instruction is inserted, all the other instructions shift to the right and the last instruction (write back stage) gets retired from the pipeline.



Current Pipeline

Pipeline Stages

Instruction Fetch

- Reads the program counter
- Gets the instructions at the program counter. Inserts the instructions into the pipeline.
- Turns on valid bit for the IF/ID latch
- Increments Program Counter

Decode

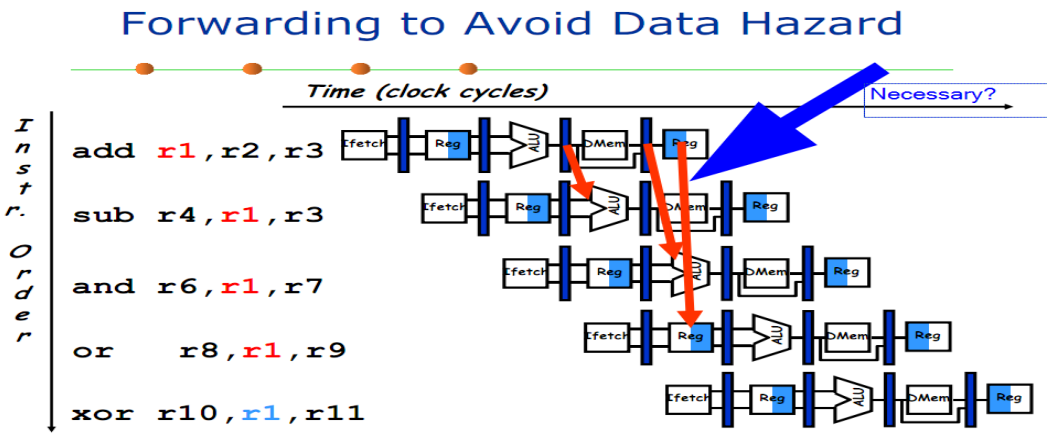
- Extracts the instructions from the IF/ID latch
- Calculate all the operands needed for the ID/EX latch
- Sets the control signals
- Transfers the values to the ID/EX latch

Execute Stage

- Extracts the values from the ID/EX latch.
- Calculates the values stored in each of the registers
- Implements the forwarding and/or load use hazard detection and handles them
- Gets the outcome of a branch and changes the PC if the branch is taken but keeps the PC the same when the branch is not taken.

Data Hazards and Forwarding

- In the case of a data hazard where the destination register of the instruction in the memory stage is the same as one of the source registers, forwarding will need to be applied.
- In the case of a load use dependency, involving a use of after a load, the following things will happen in the simulator
  - The value will be taken from the necessary latch and there will be a stall.



Branch Prediction

- A simple static branch predictor is required to be implemented in this lab. All branches are predicted to be not taken.
- If a branch is taken you increment the number of mispredictions by 1.
  - This would require converting the instructions in the latches and the current pipeline into a ***NOOP*** signified by a –2 in the pipeline
- If a branch is not taken you increment the number of correct predictions by 1.

Memory Stage

- Extracts the instructions from the EX/MEM latch
- Gets the control signal and masks the control signals based on whether a memory access is needed or not.
- There are instructions that don’t need to access memory, and that can be checked by masking the ***ControlSignal*** byte for specific control signals

Memory Stage (Continued)

- If memory access is needed, you might have to load from memory or write to memory. Use the ***ControlSignal*** field of the EX/MEM latch to determine if memory should be accessed.
- Write the necessary values to the MEM/WB latch

Modeling the Cache Behavior

- The pipeline also simulates cache delays caused by memory accesses.
- Cache misses cause a 3 cycle delay.
- This lab requires implementing the ***SimulateHitOrMiss*** which generates a pseudorandom number using the LCG (Linear congruential generator) algorithm, to determine if a memory access results in a hit or miss.

Modeling the Cache Behaviour (Continued)

- The values of a , c , m , and the initial X_n (seed) is given in ***Common.s***
- X_{n+1} is written back into $X_n$which is a variable (mentioned in"common.s")
- (X_{n+1} < 10) ? (Cache miss) : (Cache hit)

LCG: Linear Congruential Generator

$$X_{n+1} = (a \cdot X_n + c) \bmod m$$

X_n : Current value (the seed is the initial X_0)
a : Multiplier constant
c : Increment constant
m : Modulus constant
X_{n+1} : Next value to compute

Write Back Stage

- Extract values from the MEM/WB latch
- This stage involves writing values into the register file if necessary.
- That can be done by masking out the ***ControlSignal*** field of the MEM/WB latch and extracting the necessary signals.

Output

- The output is divided into 3 parts
  - The Pipeline Stages
  - The Register Table
  - Simulation summary
- The pipeline stages involve printing the current state of the pipeline after every cycle
- The register table section visualizes the values of the memory-mapped registers in the ***RegisterTable***.

Output (Continued)

- The simulation summary contains
  - Total Cycles
  - Instructions Executed
  - Cycles Per Instruction (CPI)
  - Number of Branches
  - Cache Hits and Misses
  - Branch mispredictions and Correct Predictions
  - Instructions that involve memory accesses

Functions to implement

- PipeliningSimulator
  - Main function that calls the functions i.e the 5 stages, updates the cycle count, handles the stalls, and is involved with printing the current state of the pipeline, the register table and the end statistics.
- InstructionFetchStage
  - Performs the instruction fetch stage by fetching the instruction at the current program counter, and performs the operations of the instruction fetch stage, and setting the IF/ID latch

Functions to implement (Continued)

- DecodeStage
  - Processes the instruction fetched in the previous stage. This stage is responsible for determining the instruction type, Extracting operand registers, immediates and funct3 codes, and also generating the correct control signals for each type of instruction.
- ExecutionStage
  - Performs arithmetic, logical, and address calculations based on instruction type. This stage is responsible for executing ALU operations for R-type and I-type instructions, computing branch conditions, preparing effective addresses for S-type and UJ-type instructions, propagating results and control signals to the EX/MEM pipeline latch.

Functions to implement (Continued)

- MemoryStage
  - Handles all memory access operations as determined by the control signals. This stage is responsible for performing memory reads or writes using the address computed in the EX stage, propagating relevant data and control information to the MEM/WB latch

Functions to implement (Continued)

- WriteBackStage
  - Write-back stage: writes result to register file if RegWrite is set and valid. Selects value from memory or ALU based on the control signal;
- PreDecode
  - Takes in the instruction type and instruction hex as arguments and returns indeces of rs1, rs2, rd, funct3, funct7, immediate
- SimulateHitOrMiss
  - Simulates a cache hit or miss for a given register using a Linear Congruential Generator (LCG). Updates global counters for cache hits and misses, and increments cycles on a miss.

Data Structures And Global Variables

- Global variables are given to manage the state of the pipeline, print the simulation stats, etc
- The data structures include the pipeline latches.
- The following global variables are given
  - ***CurrentPipeline***
  - ***ZeroFlag***
  - ***Cycles***
  - ***CorrectBranchPredictions***
  - ***BranchMispredictions***
  - ***NumBranches***
  - ***PCWrite***
  - ***RegisterTable***
  - ***SimulatedMemory***
  - ***NumMemAccess***
  - ***NumCacheMisses***
  - ***NumCacheHits***
  - ***NumInstructions***
  - ***CacheMissFlag***
  - ***DelayCounter***
  - ***CacheMissFlag***

Test Cases

- All test cases are standalone RISC-V assembly files (no assembler directives or global variables).
- ***Common.s*** sets up a 256-byte ***SimulatedMemory*** (found in the data structures section) and initializes a0 to point to it.
- Supported Instructions
  - R-type: add, sub
  - I-type: lw, addi
  - S-type: sw
  - SB-type: beq, bne

```
1    add a2 , a0 , zero
2    addi t0, zero, 0  #sum = 0
3    addi t2, zero, -1 #sentinel -1
4    loop:
5    lw   t3, 0(a2)    #get val
6    addi a2, a2, 4    #move ptr
7    add  t0, t0, t3   #add to sum
8    bne  t3, t2, loop #if not -1 → keep going
9    sw   t0, 0(a0)    #store result
```

Test Cases (Continued)

- Execution Context
  - Test cases run without inputs and only access ***SimulatedMemory*** for load/store.
- Purpose
  - Validate pipeline correctness with varied scenarios (array sums, arithmetic, branching).
- Students are encouraged to design additional test cases beyond those provided.

Tips and Tricks

- Read Chapter 4 : **The Processor** in the textbook!
- Use a Disassembler to decode the binary representation of the instruction to debug the pipelining stages functions! You can find one online (https://luplab.gitlab.io/rvcodecjs/)