RV32IM PIPELINE PROCESSOR

CO502- Advanced computer architecture

E/20/024 – Ariyarathna D.B.S.

E/20/366 – Seneviratne A.P.B.P.

E/20/242- Malinga G.A.I.

GROUP 05

PIPELINE DATAPATH

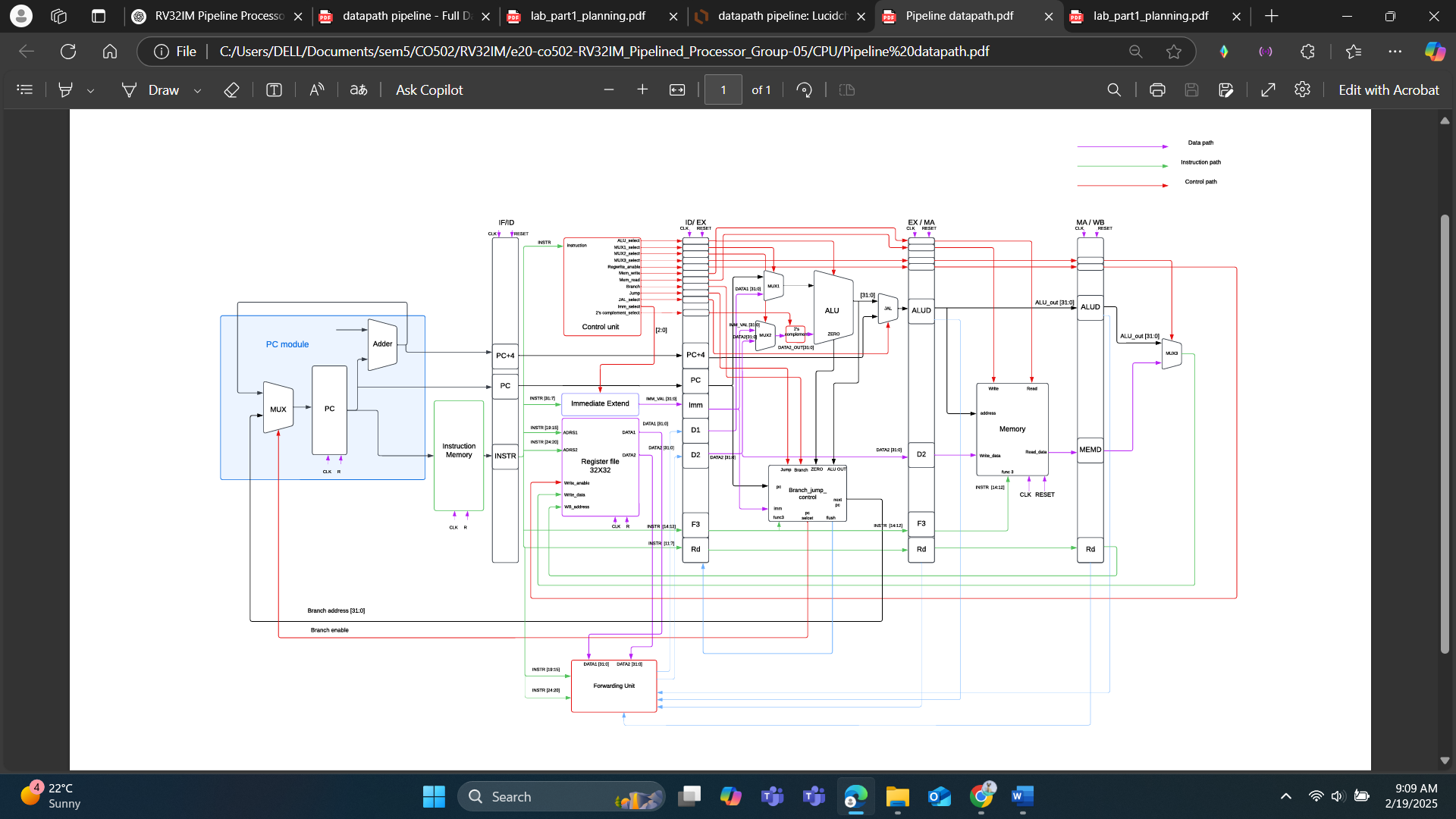


Figure 01: Pipeline Datapath of RV32Im processor

Our pipeline data path consists of 5 stages, and each stage requires 1 clock cycle.

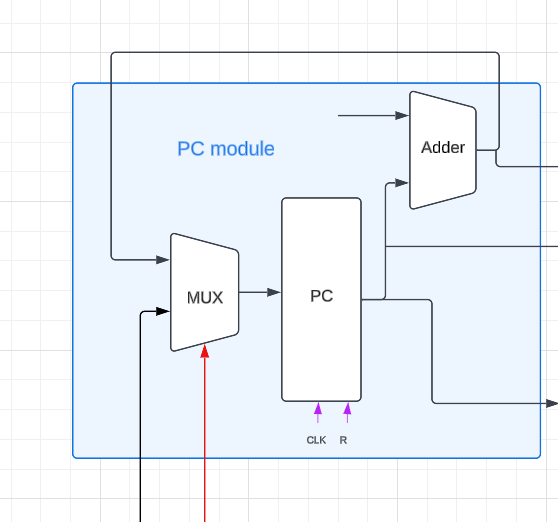
* latency (time to process one instruction) = 5 clock cycles
* Throughput (rate at which instructions are completed) = 1 instruction per clock cycle

If you start with an empty pipeline, the first instruction will take 5 cycles to complete, but after that, a new instruction will complete every clock cycle.

First Stage (Instruction fetch):

Here, first program counter should choose the next instruction between pc+4 and branch address. If branch enable signal is 1, pc is branch address and else pc equals to pc+4. Then pc fetches the instruction from instruction memory according to the instruction address. After each clock cycle 4 is added to the pc unless there is a branch.

Program counter

32-bit address 

PC+4 or Branch address

Reset -> set pc to 0

Functional blocks

* + - mux\_2x1\_32bit
    - adder\_32bit
    - pc

Why increment by 4?

reason is related to **instruction word size** and **memory alignment**.

RISC-V Uses 32-bit Instructions

* Each standard RISC-V instruction is 32 bits (4 bytes) long. The memory in RISC-V is byte-addressable (each address refers to 1 byte). To move to the next instruction, the PC must skip 4 bytes.

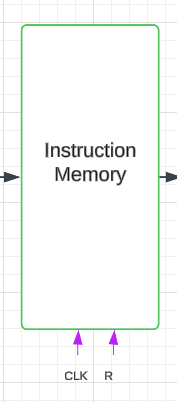
Memory Alignment in RISC-V

* RISC-V enforces word-aligned instructions, meaning instructions must start at addresses that are multiples of 4

32-bit PC connects to a memory that is **byte-addressable** (8-bit memory units), not a 32-bit-wide memory. This is because RISC-V use a byte-addressable memory model.

* PC uses byte addresses (each address refers to 8-bit memory).
* Instructions are 32-bit (4 bytes), so PC increments by 4.
* Memory can be either 8-bit-wide (multiple fetches) or 32-bit-wide (single fetch per instruction).
* PC still operates in terms of byte addresses, even if the memory bus fetches full words.

Instruction memory

Input pc 32 bit

Output Instruction 32 bit

Size: 1024 x 32-bit

All locations are initialized to NOP (0x00000013)

Pre-loaded instructions

word-aligned addressing (pc[31:2])

The instruction memory module is a crucial component in a 5-stage pipelined processor. It stores and provides 32-bit instructions based on the Program Counter (PC) value. The memory is preloaded with instructions from an external binary file (program.mem) and supports word-aligned accesses.

Control Signals and Functional Blocks

Inputs:

* clk (Clock): Synchronizes pipeline execution.
* reset: Resets the instruction output to a NOP (No Operation) instruction (0x00000013).
* pc (Program Counter) [31:0]: Specifies the address of the instruction to be fetched.

Output:

* instruction [31:0]: The fetched instruction corresponding to the given pc.

Internal Components:

* Memory Array (imem): 1024x32-bit memory storing instructions.
* PC Addressing Logic: Ensures word-aligned instruction access.

Timing Analysis

* Read Timing: Instruction memory operates asynchronously, meaning that the instruction is available immediately after the pc changes.
* Clock Cycle Impact: Since instruction memory is read asynchronously, it does not introduce additional delay in the pipeline. The instruction fetch occurs in one cycle.
* Reset Behavior: On reset, the instruction output defaults to NOP (0x00000013).

Design Choices

Memory Initialization

* Uses $readmemb("program.mem", imem); to load instructions from an external binary file.
* Default values are initialized to NOP (0x00000013) to prevent execution errors if memory is uninitialized.

Word-Aligned Access

* Instructions are stored in 32-bit words.
* The PC uses pc[31:2] for indexing to ensure word-aligned access.
* If pc is misaligned (not a multiple of 4), the module returns a NOP instruction.

Asynchronous Read

* Allows the pipeline to fetch instructions without requiring additional clock cycles, ensuring efficient execution.
* Enables seamless instruction fetching in the Instruction Fetch (IF) stage of the pipeline.

Limitations and Potential Improvements

No Write Capability

* This module is read-only, meaning it does not support self-modifying code or runtime updates.
* A separate instruction cache or memory management unit would be required for dynamic updates.

No Error Handling for Invalid PC Values

* If pc[31:2] exceeds 1023, the module defaults to NOP instead of signaling an error.
* A proper exception mechanism could be implemented to handle out-of-bounds access.

No Cache Implementation

* Directly fetching instructions from memory every cycle may be slower compared to a cache-based approach.
* Future improvements could include an instruction cache to reduce memory latency.

IF/ID Pipeline register

Design Choices:

* Pipeline Stage Isolation: Stores instruction and PC values between the Fetch and Decode stages.
* Busy Wait Signal: Introduced to stall the pipeline when required (e.g., data hazards).
* Synchronous Update: Ensures that updates happen only at the rising edge of the clock.
* Reset Handling: Clears the stored instruction and PC values to prevent incorrect execution after reset.

Control and Timing:

* Clock-Driven Updates: Updates the register values at the positive clock edge.
* Reset Priority: Reset signal takes precedence over normal updates.

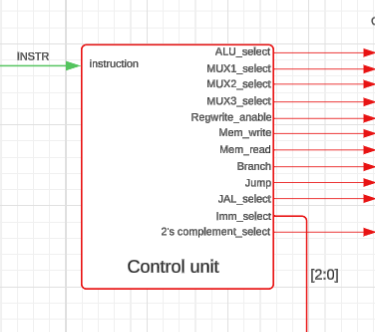
Limitations:

* Pipeline Hazards: Does not handle control hazards (e.g., branch mispredictions) or data hazards directly.
* Fixed Instruction Width: Only supports 32-bit instructions.
* No Forwarding Logic: Relies on external components to resolve hazards

Second stage (Instruction decode):

Here this stage has 4 main components. They are control unit, immediate extend unit, register file and forwarding unit. Control units generate the control signals which is necessary to smooth run of instructions through the stages. Here are the control signals which we generate for each instruction.  
Extend immediate module is used to extend the immediate value for 32 bits.

Control unit



Design Choices:

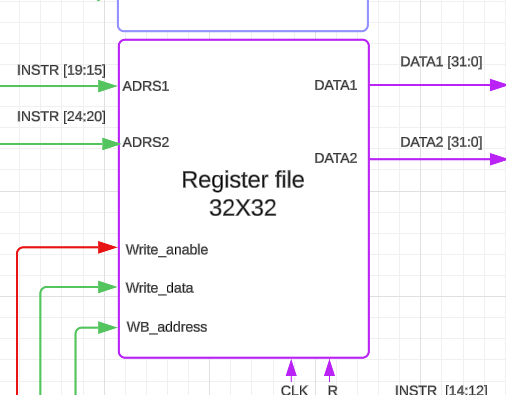
* Opcode-Based Control Signals: Generates control signals based on the instruction opcode.
* Modular Design: Separate control paths for ALU operations, memory access, and branching.
* Synchronization with Pipeline: Ensures seamless instruction execution across pipeline stages.

Control and Timing:

* Instruction Decoding: Interprets the instruction to generate appropriate control signals.
* Clock-Synchronized Updates: Ensures control signals are generated in sync with the pipeline.
* Branch and Jump Handling: Determines control flow changes and issues stall or flush signals when necessary.

Limitations:

* Limited Hazard Handling: Does not include forwarding or dependency checking mechanisms.
* Fixed Instruction Set: Designed for a specific instruction set and may require modifications for extensions.
* Potential Pipeline Stalls: Lacks dynamic hazard prediction, leading to occasional stalls in execution.

Register file

Design Choices:

* 32 General-Purpose Registers: Implements a standard RISC-V style register file.
* Read and Write Mechanism: Supports synchronous write operations and combinational reads.
* x0 Register Hardwired to Zero: Ensures compliance with RISC-V architecture constraints.
* Debugging and Monitoring: Includes $display statements and print\_registers task for debugging purposes.

Control and Timing:

* Clock-Based Write Operation: Register writes occur on the rising edge of the clock.
* Asynchronous Read: Allows immediate retrieval of register values.
* Reset Handling: Clears all register values upon reset to prevent erroneous computations.
* Write Enable Control: Ensures data is only written when WRITE\_ENABLE is asserted.

Limitations:

* Fixed Register Size: Limited to 32 registers, restricting additional storage capacity.
* No Forwarding Mechanism: Requires external logic to manage data dependencies.
* Reset Overhead: Resets all registers, which could introduce delays in recovery scenarios.

Immediate extend



Purpose

The Immediate Extend Module expands 12-bit or 20-bit immediates from RISC-V instructions into 32-bit values, ensuring proper sign extension.

Inputs & Outputs

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| imm\_value | 32 bits | Input | Raw immediate value from the instruction. |
| imm\_select | 3 bits | Input | Specifies the immediate type (I, S, B, U, J). |
| extended\_imm\_value | 32 bits | Output | The extended 32-bit immediate value. |

Design Choices

Sign Extension for Correct Arithmetic

* Immediate values are often used in arithmetic operations.
* Correct sign extension ensures negative values retain their meaning.

Why 5 Immediate Types?

* RISC-V ISA uses different formats:
  + I-type (12-bit signed) for arithmetic (addi, lw, jalr).
  + S-type (split into two 7-bit fields) for stores (sw).
  + B-type (12-bit branch offset) for conditional jumps (beq, bne).
  + U-type (20-bit upper immediate) for lui and auipc.
  + J-type (20-bit jump offset) for jal.

Timing Considerations

* Combinational Logic: Immediate extension happens instantly after the instruction is decoded.
* No Clock Dependency: No extra cycles are required, ensuring a single-cycle extension.

Limitations

* Limited to 5 Immediate Types:
  + Extensions for compressed instructions (C-extension) would require additional logic.
* Potential Delays in Multi-Cycle Pipelines:
  + In a multi-cycle or pipelined CPU, hazards may arise if immediate values are needed before they are extended.

ID\_EX pipeline register

The ID\_EX module is a pipeline register in a processor's datapath, specifically in a RISC-V processor pipeline. It acts as a bridge between the Instruction Decode (ID) stage and the Execute (EX) stage of the pipeline, holding values that need to be passed forward while ensuring proper synchronization. Below is a detailed breakdown of its components and functionality.

Stage 3 (Execution stage)  
This stage consists of four muxes, ALU and branch jump control module. Three muxes out of four muxes used to direct the correct two values as input to the ALU. Other mux is used to select between ALU result or pc+4, because if there is branch or jump, pc+4 should write back to the register. Branch jump control unit used to check whether the brach condition is true and if the condition is true, it will calculate the branch address.  
  
Stage 4 (Memory stage)  
This stage consists of data memory. It will load and store data for the relevant instruction  
Stage 5 (write back stage)  
This stage write back the data which is come from ALU result, pc+4 or data memory to the register file for the given address.

Data memory

# Data Memory Module Detailed Explanation

This is a comprehensive implementation of a data memory module for a RISC-V processor. Let me break down its functionality, features, and how it works:

## Basic Structure

- \*\*Size\*\*: 1024 words × 32 bits (4KB total memory)

- \*\*Memory Array\*\*: `memory[0:1023]` stores the actual data

- \*\*Access Control\*\*: Manages read/write operations with proper timing

## Interface Signals

- \*\*Control Inputs\*\*:

- `Read`: Activates read operations

- `Write`: Activates write operations

- `Clock`: System clock for synchronous operations

- `Reset`: Resets memory contents and control signals

- `Func3`: RISC-V function code (3 bits) that determines memory access type

- \*\*Data Interface\*\*:

- `Address[31:0]`: 32-bit memory address

- `Write\_data[31:0]`: 32-bit data to be written

- `Read\_data[31:0]`: 32-bit data read from memory

- \*\*Pipeline Control\*\*:

- `busywait`: Signals to the pipeline that memory operation is in progress

## Memory Access Control

The module implements a state machine to control memory operations:

1. When `Read` or `Write` is asserted, the module sets internal flags

2. Read and write operations are mutually exclusive (prioritizes write)

3. The `busywait` signal manages pipeline stalls during memory operations

## Memory Addressing

- The memory is word-addressed using `Address[31:2]`

- The lower bits `Address[1:0]` are used for sub-word addressing (byte/halfword selection)

## Memory Operations

This module supports all RISC-V memory access types based on `Func3`:

### Read Operations (`read\_access`)

- \*\*LB (Load Byte - Signed)\*\* - `Func3=000`:

- Reads 8 bits and sign-extends to 32 bits

- Properly handles byte alignment within words

- \*\*LH (Load Halfword - Signed)\*\* - `Func3=001`:

- Reads 16 bits and sign-extends to 32 bits

- Handles halfword alignment

- \*\*LW (Load Word)\*\* - `Func3=010`:

- Reads full 32-bit word

- \*\*LBU (Load Byte - Unsigned)\*\* - `Func3=100`:

- Reads 8 bits and zero-extends to 32 bits

- \*\*LHU (Load Halfword - Unsigned)\*\* - `Func3=101`:

- Reads 16 bits and zero-extends to 32 bits

### Write Operations (`write\_access`)

- \*\*SB (Store Byte)\*\* - `Func3=000`:

- Writes 8 bits to the specified byte position

- \*\*SH (Store Halfword)\*\* - `Func3=001`:

- Writes 16 bits to the specified halfword position

- \*\*SW (Store Word)\*\* - `Func3=010`:

- Writes full 32-bit word

## Special Features

1. \*\*Memory Reset\*\*:

- Initializes all memory locations to zero when `Reset` is asserted

- Implemented with a for-loop through all memory addresses

2. \*\*Debugging Support\*\*:

- Uses `$dumpfile` and `$dumpvars` for waveform generation

- Allows monitoring of all memory locations during simulation

## Integration with Pipeline

In a RISC-V pipeline, this module would connect to:

1. The Execute/Memory stage that generates memory addresses

2. The Writeback stage that consumes read data

3. The pipeline control logic that responds to the `busywait` signal

This data memory module provides complete support for the RISC-V load/store architecture with proper handling of all data widths and sign extension requirements.