Harvard 5 Stages Pipeline CPU Report

December 10, 2024

0.1 Contributors information

Name	Section	ID
Youssef Tarek	2	9220990
Marwan Mohamed	2	9220808
Youssef Roshdy	2	9220985
Moamen Hefny	2	9220886

Contents

Chapter 1

Introduction

This is the introduction section of the report. Here you can provide an overview of the topic, background information, and the purpose of the report.

Chapter 2

Instruction Set Architecture

Instruction Types and Classification

This document defines the instruction set architecture (ISA) for a hypothetical processor. The instructions are categorized into three main types: **R-Type**, **I-Type**, and **J-Type**, with an additional category for miscellaneous instructions referred to as **Others**. The bit allocation for each instruction type is detailed along with the format used for decoding.

Instruction Categories

R-Type (Register-to-Register Instructions)

R-Type instructions perform operations between registers and store the result in a destination register. They typically involve arithmetic and logical operations.

Opcode	Function	Rsrc1	Rsrc2	Rdst
[15-14]	[13-11]	[10-8]	[7-5]	[4-2]

Unused bits: [1-0]
Function field: 3 bits to specify the operation.
Supported Instructions:

- `NOT Rdst, Rsrc1`
- `INC Rdst, Rsrc1`
- `MOV Rdst, Rsrc1`
- `ADD Rdst, Rsrc1, Rsrc2`
- `SUB Rdst, Rsrc1, Rsrc2`
- `AND Rdst, Rsrc1, Rsrc2`

I-Type (Immediate and Memory Instructions)

I-Type instructions involve immediate values or memory access, typically for arithmetic or data transfer operations.

Opcode	Function	Rsrc1	Rsrc2	Rdst
[15-14]	[13-11]	[10-8]	[7-5]	[4-2]

Unused bits: [1-0]
Second Fetch (Immediate or Offset):

Immediate / Offset
[15-0]

Supported Instructions:

1. `IADD Rdst, Rsrc1, Imm`
2. `LDM Rdst, Imm`
3. `LDD Rdst, offset(Rsrc1)`
4. `STD Rsrc1, offset(Rsrc2)`

J-Type (Control Flow Instructions)

J-Type instructions handle program control flow, such as jumps and calls.

Opcode	Function	Rsrc1	Index
[15-14]	[13-11]	[10-8]	[7-6]

Unused bits: [5-0]
Supported Instructions:

1. `JZ Rsrc1`
2. `JN Rsrc1`
3. `JC Rsrc1`
4. `JMP Rsrc1`
5. `CALL Rsrc1`
6. `RET`
7. `RTI`
8. `INT index`

Other Instructions

These instructions do not fit into the R-Type, I-Type, or J-Type categories and typically control processor state or interact with peripherals.

Opcode	Function	Rsrc1	Rdst
[15-14]	[13-11]	[10-8]	[7-5]

Unused bits: [4-0]
Supported Instructions:

1. `NOP` – No operation.
2. `HLT` – Halt the processor.
3. `SETC` – Set the carry flag.
4. `OUT Rsrc1` – Output data from `Rsrc1`.
5. `IN Rdst` – Input data into `Rdst`.
6. `PUSH Rsrc1` – Push `Rsrc1` onto the stack.
7. `POP Rdst` – Pop value from the stack into `Rdst`.

Bit Allocation Overview

- **Opcode:** 2 bits `[15-14]`.
- **Function:** 3 bits `[13-11]`.
- **Register Fields:**
  - `Rsrc1`: 3 bits `[10-8]`.
  - `Rsrc2`: 3 bits `[7-5]` (R-Type and I-Type).
  - `Rdst`: 3 bits `[4-2]` (R-Type, I-Type, and Others).

Encoding Summary

R-Type Encoding

Utilizes two registers as sources and one destination register. The function field specifies the operation.

[Opcode — Function — Rsrc1 — Rsrc2 — Rdst — Unused(2 bits)]

I-Type Encoding

Involves one source register and one destination register, with a second fetch for immediate values or offsets.

[Opcode — Function — Rsrc1 — Rsrc2 — Rdst — Unused(2 bits)]

[Immediate/Offset (Second Fetch)]

J-Type Encoding

Uses a single register as the jump target.

[Opcode — Function — Rsrc1 — Index — Unused(6 bits)]

Other Instructions Encoding

Instructions that do not fit into the R-Type, I-Type, or J-Type categories.

[Opcode — Function — Rsrc1 — Rdst — Unused(5 bits)]

Chapter 3

Instruction Set Encoding

R-Type Instructions

Instruction	Opcode	Function	Rsrc1	Rsrc2	Rdst	Unused
NOT Rdst, Rsrc1	00	000	xxx	000	xxx	00
INC Rdst, Rsrc1	00	001	xxx	000	xxx	00
MOV Rdst, Rsrc1	00	010	xxx	000	xxx	00
ADD Rdst, Rsrc1, Rsrc2	00	011	xxx	xxx	xxx	00
SUB Rdst, Rsrc1, Rsrc2	00	100	xxx	xxx	xxx	00
AND Rdst, Rsrc1, Rsrc2	00	101	xxx	xxx	xxx	00

Table 3.1: R-Type Instructions

I-Type Instructions

Instruction	Opcode	Function	Rsrc1	Rsrc2	Rdst	Unused
IADD Rdst, Rsrc1, Imm	01	000	xxx	000	xxx	00
LDM Rdst, Imm	01	001	000	000	xxx	00
LDD Rdst, offset(Rsrc1)	01	010	xxx	000	xxx	00
STD Rsrc1, offset(Rsrc2)	01	011	xxx	xxx	000	00

Table 3.2: I-Type Instructions

J-Type Instructions

Instruction	Opcode	Function	Rsrc1	Index	Unused
JZ Rsrc1	10	000	xxx	00	00000000
JN Rsrc1	10	001	xxx	00	00000000
JC Rsrc1	10	010	xxx	00	00000000
JMP Rsrc1	10	011	xxx	00	00000000
CALL Rsrc1	10	100	xxx	00	00000000
RET	10	101	000	00	00000000
RTI	10	110	000	00	00000000
INT index	10	111	000	xx	00000000

Table 3.3: J-Type Instructions

Other Instructions

Instruction	Opcode	Function	Rsrc1	Rdst	Unused
NOP	11	000	000	000	00000
HLT	11	001	000	000	00000
SETC	11	010	000	000	00000
OUT Rsrc1	11	011	xxx	000	00000
IN Rdst	11	100	000	xxx	00000
PUSH Rsrc1	11	101	xxx	000	00000
POP Rdst	11	110	000	xxx	00000

Table 3.4: Other Instructions

Control Unit Signals

Instruction	Control Signals
NOT Rdst, Rsrc1	ALUOp = NOT, RegWrite = 1
INC Rdst, Rsrc1	ALUOp = INC, RegWrite = 1
MOV Rdst, Rsrc1	ALUOp = MOV, RegWrite = 1
ADD Rdst, Rsrc1, Rsrc2	ALUOp = ADD, RegWrite = 1
SUB Rdst, Rsrc1, Rsrc2	ALUOp = SUB, RegWrite = 1
AND Rdst, Rsrc1, Rsrc2	ALUOp = AND, RegWrite = 1
IADD Rdst, Rsrc1, Imm	ALUOp = ADD, ALUSrc2 = 1[Imm], RegWrite = 1, prevOp = 1
LDM Rdst, Imm	MemRead = 1, ALUSrc2 = 1[Imm] MemToReg = 1, RegWrite = 1, prevOp = 1
LDD Rdst, offset(Rsrc1)	MemRead = 1, ALUSrc2 = 1[Imm] MemToReg = 1, RegWrite = 1, prevOp = 1
STD Rsrc1, offset(Rsrc2)	MemWrite = 1, ALUSrc2 = 1[Imm], prevOp = 1
JZ Rsrc1	JZ = 1
JN Rsrc1	JN = 1
JC Rsrc1	JC = 1
JMP Rsrc1	JMP = 1
CALL Rsrc1	CALL = 1, RegWrite = 1, SP = SP - 1
RET	JMP = 1, SP = SP + 1
RTI	JMP = 1, SP = SP + 1
INT index	INT = 1
NOP	NOP = 1
HLT	-
SETC	CF = 1
OUT Rsrc1	OUTPORT = 1
IN Rdst	INPORT = 1, RegWrite = 1
PUSH Rsrc1	SP = SP - 1
POP Rdst	RegWrite = 1, SP = SP + 1

Table 3.5: Control Unit Signals for Each Instruction

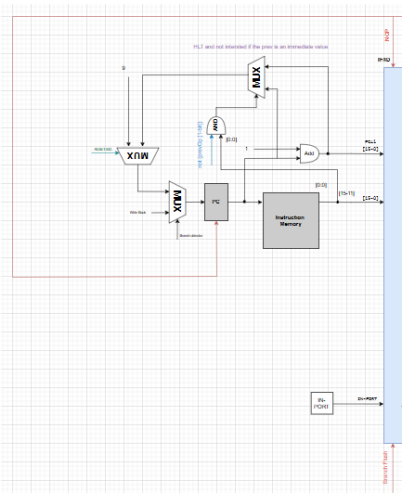
Chapter 4

Pipeline Stages Design

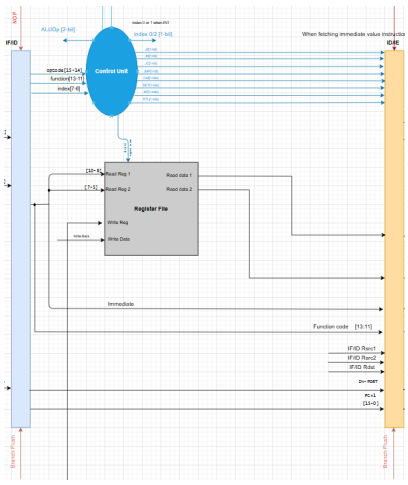
The processor pipeline is composed of five distinct stages: **IF** (Instruction Fetch), **ID** (Instruction Decode), **EX** (Execution), **MEM** (Memory Access), and **WB** (Write Back). Each stage is responsible for specific tasks in the instruction execution process.

IF (Instruction Fetch)

- Fetch the instruction from memory using the Program Counter (PC).
- Increment the PC to point to the next instruction.
- Store the fetched instruction in the Instruction Register (IR).
- Update the PC accordingly in case of a branch or jump instruction.
- Stall the PC counter value in case of a halting instruction to stop fetching new instructions.

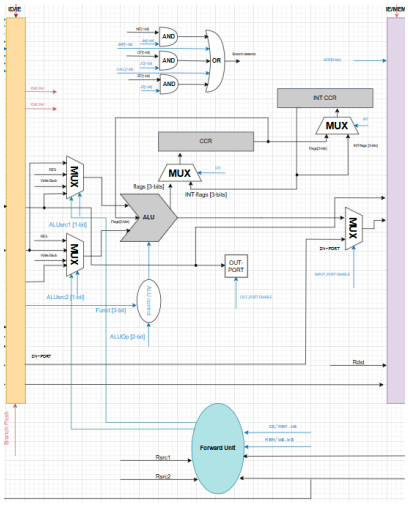


ID (Instruction Decode)



- Decode the instruction opcode and extract the necessary fields.
- Read the register file to obtain the values of the source registers.
- Determine the type of instruction and generate the required control signals.
- Calculate the effective address for memory operations if applicable.
- Forward register values and immediate operands to the next stage (EX).
- Stall the pipeline if a data hazard is detected and cannot be resolved with forwarding.

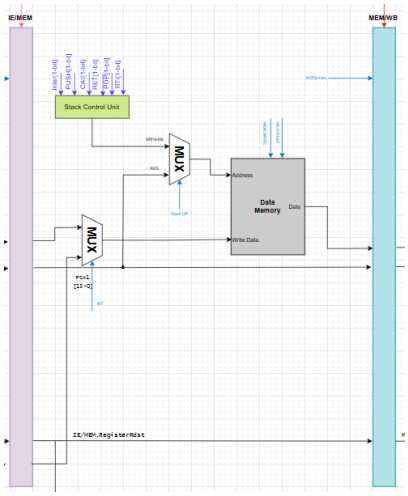
EX (Execution)



- Select ALU inputs through multiplexers based on control signals (`ALUsrc1` and `ALUsrc2`).
- Perform the ALU operation based on the control signals (`ALUOp`) and the instruction's function field (`Func[3-bit]`).
- Generate the condition flags (`ZF, NF, CF`) based on the ALU result.
- Calculate the branch target address if the instruction is a branch or jump.
- Evaluate the branch condition using flags and control signals (`JMP`, `CALL`, `JC`, `JZ`, `JN`).
- Forward the ALU result to the Memory Access (MEM) stage.
- Stall the pipeline if a data hazard is detected and cannot be resolved with data forwarding.
- Flush the pipeline if a branch instruction is mispredicted.
- Generate control signals to update the Program Counter (PC) if a branch is taken.

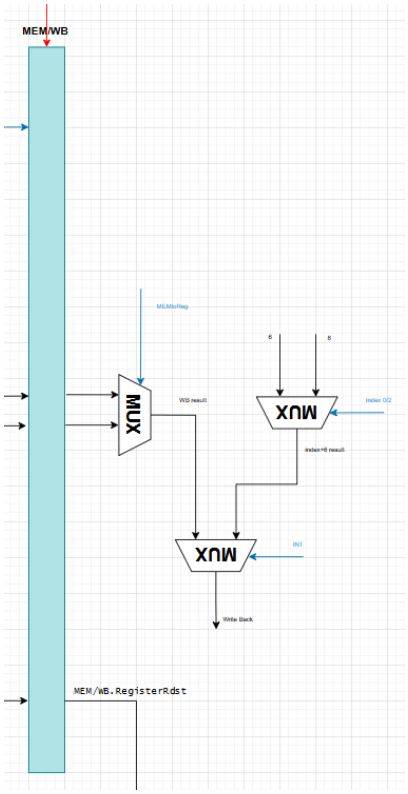
MEM (Memory Access)

- Access memory to read or write data based on the instruction type.
- Forward the memory read data to the Write Back (WB) stage.
- Stall the pipeline if a data hazard is detected and cannot be resolved with data forwarding.
- Handle stack operations (PUSH, POP) through the stack control unit.
- Generate signals to enable memory read/write operations.
- Forward the ALU result to the WB stage if the instruction does not involve memory access.



WB (Write Back)

- Update the register file with the new value if the instruction is a register operation.
- Update the Program Counter (PC) with the branch target address if a branch is taken.
- Stall the pipeline if a data hazard is detected and cannot be resolved with data forwarding.
- Update the PC with the target address if the instruction is a jump or call.
- Update the PC with the return address if the instruction is a return.
- Halt the processor if the instruction is a halt.



Chapter 5

Hazard Handling

Handling Structural Hazards

Structural hazards are managed by ensuring that the pipeline has enough resources to handle multiple instructions simultaneously. This can be achieved by duplicating resources or by scheduling instructions to avoid conflicts.

Resource Duplication

Duplicating critical resources, such as ALUs or memory ports, allows multiple instructions to use these resources concurrently, reducing the likelihood of structural hazards.

Instruction Scheduling

To further mitigate structural hazards, we have chosen to schedule the read and write operations in different parts of the cycle. This approach ensures that:

- **Write Operations:** Are performed in the first half of the clock cycle.
- **Read Operations:** Are performed in the second half of the clock cycle.

By separating read and write operations within the same cycle, we minimize resource conflicts and improve the overall efficiency of the pipeline.

Handling Data Hazards

Data hazards are managed using data forwarding and pipeline stalling:

- **Data Forwarding:** The result of an instruction is forwarded to subsequent instructions that need it before it is written back to the register file.
- **Pipeline Stalling:** The pipeline is stalled until the required data is available. This is done by inserting no-operation (NOP) instructions.

Example of Data Forwarding

Consider the following sequence of instructions:

```
ADD R1, R2, R3              IF     ID     EX --|  MEM     WB
SUB R4, R1, R5              -      IF     ID   |->EX      MEM     WB
```

The result of the ADD instruction is forwarded to the SUB instruction to avoid stalling the pipeline.

Example of Pipeline Stalling

Consider the following sequence of instructions:

```
LDD R1, 0(R2)               IF     ID     EX      MEM --|  WB
ADD R3, R1, R4              -      IF     ID------ID    |->EX      MEM     WB
```

The pipeline is stalled until the LDD instruction completes and the data is available for the ADD instruction.

Handling Control Hazards

Control hazards are handled using branch prediction techniques. We employ a static branch prediction technique known as "Backward Taken, Forward Not Taken" (BTFNT). This technique predicts that a branch will be taken if it jumps backward (typically for loops) and not taken if it jumps forward.

Backward Taken, Forward Not Taken (BTFNT)

The BTFNT technique helps in reducing the number of pipeline flushes by making educated guesses about the branch direction based on its target address.

Example of BTFNT

Consider the following sequence of instructions:

```
LOOP:   SUB R1, R1, #1          IF     ID     EX     MEM     WB
        JNZ LOOP                -      IF     ID     EX      MEM     WB
        IN Rsrc2   --flush      -      -      IF
```

In this example, the branch instruction 'JNZ LOOP' jumps backward to the label 'LOOP'. Using BTFNT, the pipeline predicts that the branch will be taken.

```
ADD R2, R3, R4                IF     ID     EX     MEM     WB
JMP END                       -      IF     ID     EX      MEM     WB
SUB R5, R6, R7     --flush    -      -      IF
END:   HLT                    -      -      -      IF      ID      EX      MEM     WB
```

In this example, the branch instruction 'JMP END' jumps forward to the label 'END'. Using BTFNT, the pipeline predicts that the branch will not be taken.

Pipeline Flushing

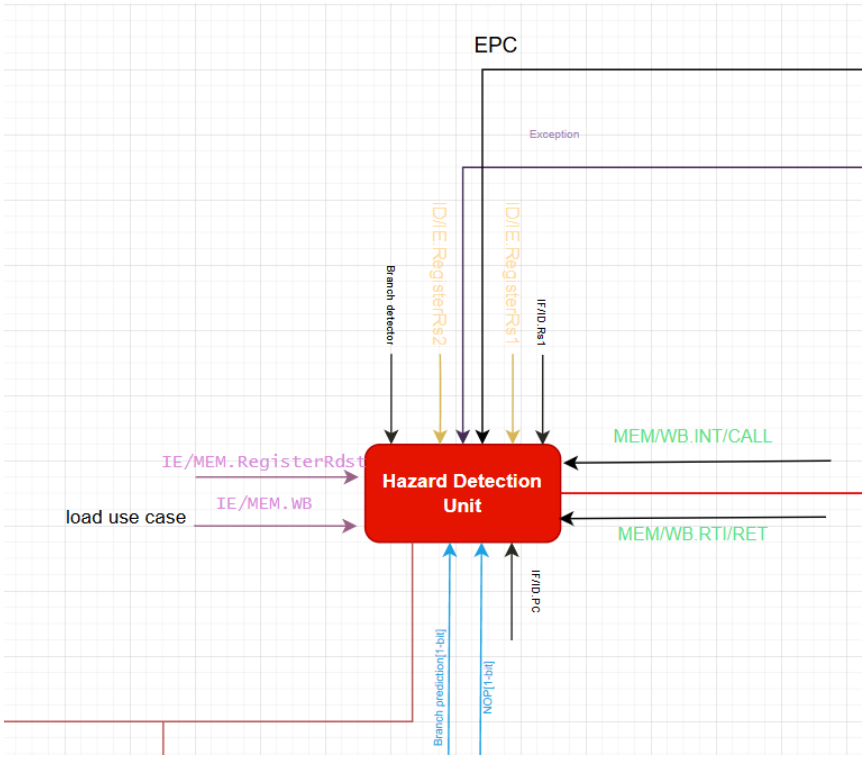
If the prediction is incorrect, the pipeline is flushed, and the correct instructions are fetched. This involves invalidating the instructions in the pipeline stages and fetching the correct instructions from memory.

Advantages of BTFNT

- Effective for typical loop structures and forward jumps.
- Reduces the number of pipeline flushes compared to always predicting taken or not taken.

Pipeline Hazard Detection Unit

A hazard detection unit is implemented to identify hazards and take appropriate actions. It monitors the pipeline stages and generates control signals to handle hazards.



- **Structural Hazard Detection:** Ensures that the pipeline has enough resources to handle multiple instructions simultaneously by duplicating resources or scheduling instructions to avoid conflicts.
- **Data Hazard Detection:** Manages data hazards using data forwarding to pass results directly to dependent instructions or pipeline stalling by inserting no-operation (NOP) instructions until the required data is available.

- **Control Hazard Detection:** Uses branch prediction techniques, specifically "Backward Taken, Forward Not Taken" (BTFNT), to predict branch directions and manage pipeline flushing if the prediction is incorrect.

Chapter 6

Exception Handling

Exception Types

Exceptions are unexpected events that occur during program execution, requiring special handling to ensure the correct operation of the processor. The following exception types are supported by the processor:

- **Stack Underflow:** Occurs when the stack pointer exceeds its limits.
- **Invalid Memory Access:** Indicates an attempt to access an invalid memory location.