MIPS CPU and Compiler

Authors :

Name:	E-mail:
Mario Morcos	mariomorcoswassily@gmail.com
Mohamed Mahmoud	mohamed_hesham23@outlook.com
Sameh Maher Kalach	samehkalash0@gmail.com
Mohamed Amr	elnaggarx2003@gmail.com

Introduction

Welcome to our MIPS CPU and Compiler Documentation. This comprehensive guide aims to provide an in-depth understanding of the MIPS (Microprocessor with Pipeline Stages) architecture, its CPU design, and the corresponding compiler functionalities.

This documentation is structured into several key sections, each focusing on essential aspects of MIPS CPU design and compiler construction: MIPS Architecture ,CPU Implementation ,Optimization Techniques, Compiler Design and Practical Examples

The MIPS CPU and Compiler Documentation offer a gateway to the world of computing measured in millions of instructions per second. By engaging with this guide, readers will gain a deep understanding of MIPS architecture and compiler design, enabling them to harness the full potential of MIPS-based systems for a wide range of applications.

Please refer to this documentation for comprehensive insights into using it, and feel free to reach out with any questions, feedback, or suggestions.

Contents :

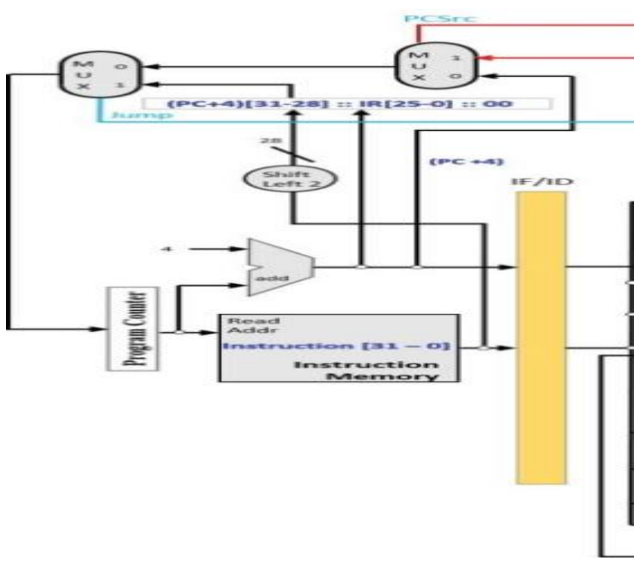
I.Data Path

A. Fetch Pipeline

Overview :

The Fetch Pipeline stage is the first step in the instruction execution process. This section covers the mechanisms involved in retrieving instructions from memory, including the program counter (PC) and instruction memory. It’s discussed how the fetch pipeline ensures a continuous flow of instructions into the CPU, setting the stage for subsequent processing stages.

Fetching Schematics :



The Fetch Pipeline module integrates two main components: Instruction_Fetch: Fetches the next instruction based on the program counter, jump signals, and branch target. IF_ID: Holds the fetched instruction and the incremented program counter value until the next clock cycle. The stall signal is used to pause the pipeline, preventing updates to the program counter and fetched instruction if necessary. This setup is typical in pipelined CPU designs, ensuring that instructions are fetched and passed to the next stage efficiently while handling hazards and control signals correctly.

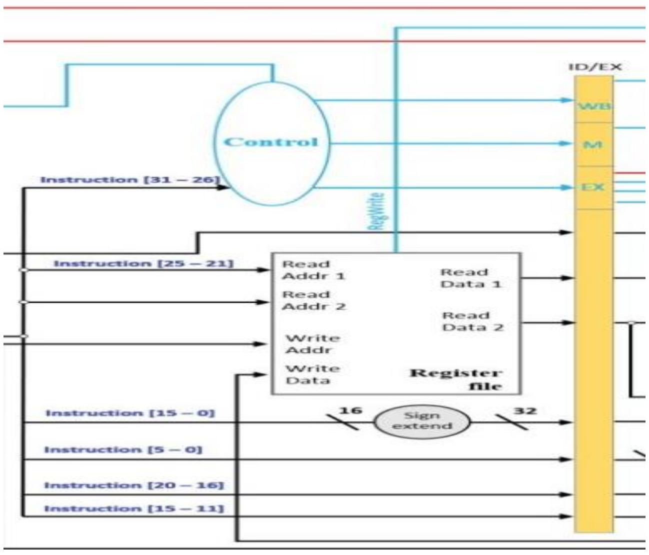
I. B. Decode Pipeline

Overview :

In the Decode Pipeline stage, the fetched instruction is interpreted to determine what action is required. The process of decoding instruction opcodes, and the significance of registers in this stage. By understanding the decode pipeline, readers will grasp how the CPU prepares instructions for execution.

Decoding Schematics :

The Decode Pipeline module integrates two main components: Instruction_Decode: Decodes the instruction, reads the register file, and prepares the necessary intermediate values and control signals. ID_EX: Holds the control signals, operand data, and intermediate values until the next clock cycle. The stall signal is used to pause the pipeline, preventing updates if necessary. This setup ensures that the decoded instruction and associated control signals and data are correctly passed to the next stage of the pipeline.



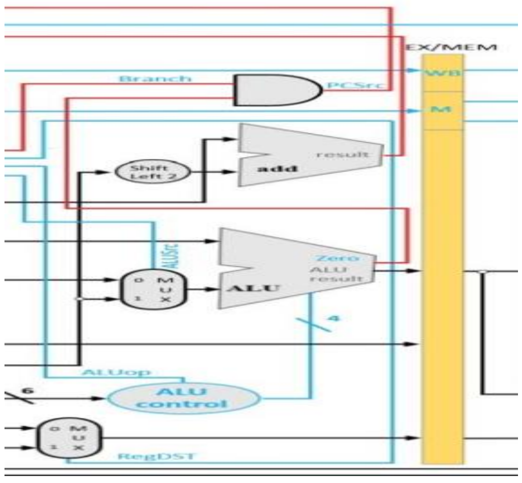
II. C. Memory Execute Pipeline

Overview :

The Execute Pipeline stage is where the actual computation or operation specified by the instruction is carried out. This section details the operation of the ALU, the handling of different instruction types (arithmetic, logic, etc.), and the execution of branch instructions. It highlights how the pipeline ensures that execution is efficient and that data hazards are minimized.

Execute Pipeline Schematics :

The Execution module integrates two main components: Excecute_Address: Executes the instruction using the ALU and determines the next program counter for branch instructions. EX_MEM: Holds the control signals, ALU result, and other data until the next clock cycle. This setup ensures that the executed instruction and associated control signals and data are correctly passed to the memory stage of the pipeline.



ALU OP of Operations :

is typically derived from a combination of the instruction's function field (for R-type instructions) and the main opcode field. The following is an example mapping for R-type instructions:

OpCode	aluOp	FunctionOp	operation
000000	10	100000	add
000000	10	100010	subtract
000000	10	100101	or
000000	10	100100	and
000000	10	011000	mul
001000	00	XXXXXX	addi
100100	00	XXXXXX	LW
101011	00	XXXXXX	SW
100001	01	XXXXXX	skip
000101	10	XXXXXX	bun

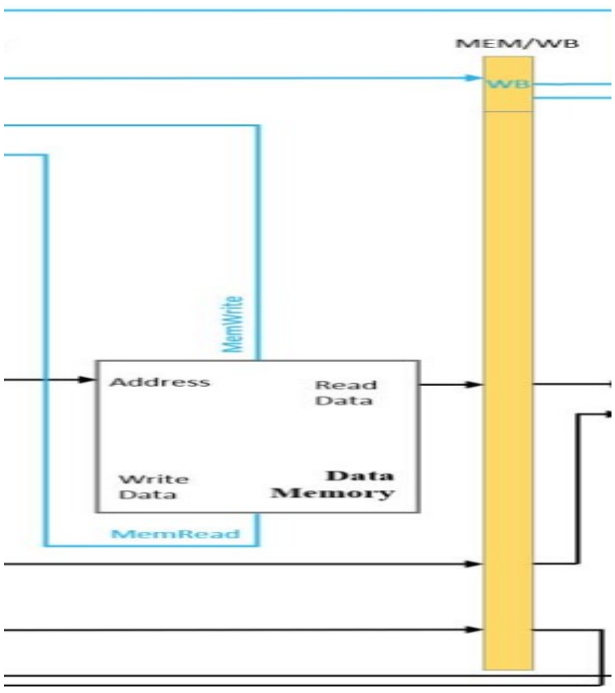
III. D. Memory Access Pipeline

Overview :

In the Memory Access Pipeline stage, the CPU interacts with memory to read from or write to it as needed by the instruction. This section covers the mechanisms for accessing data memory and the management of memory access conflicts. It provides insights into how the pipeline handles memory operations without stalling the overall instruction flow.

Access Pipeline Schematics :

The Access module integrates two main components: Memory_Access: Handles memory access, reading data from or writing data to memory. MEM_WB: Holds the control signals, ALU result, and data read from memory until the next clock cycle. This setup ensures that the data from the memory access stage is correctly passed to the write-back stage of the pipeline. The Memory_Access component performs the actual memory read/write operations, while the MEM_WB component stores the results and control signals until they are needed in the next stage.

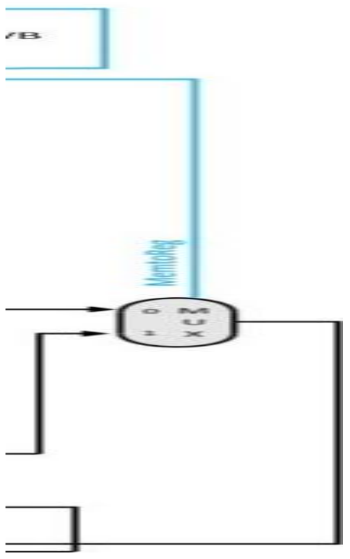


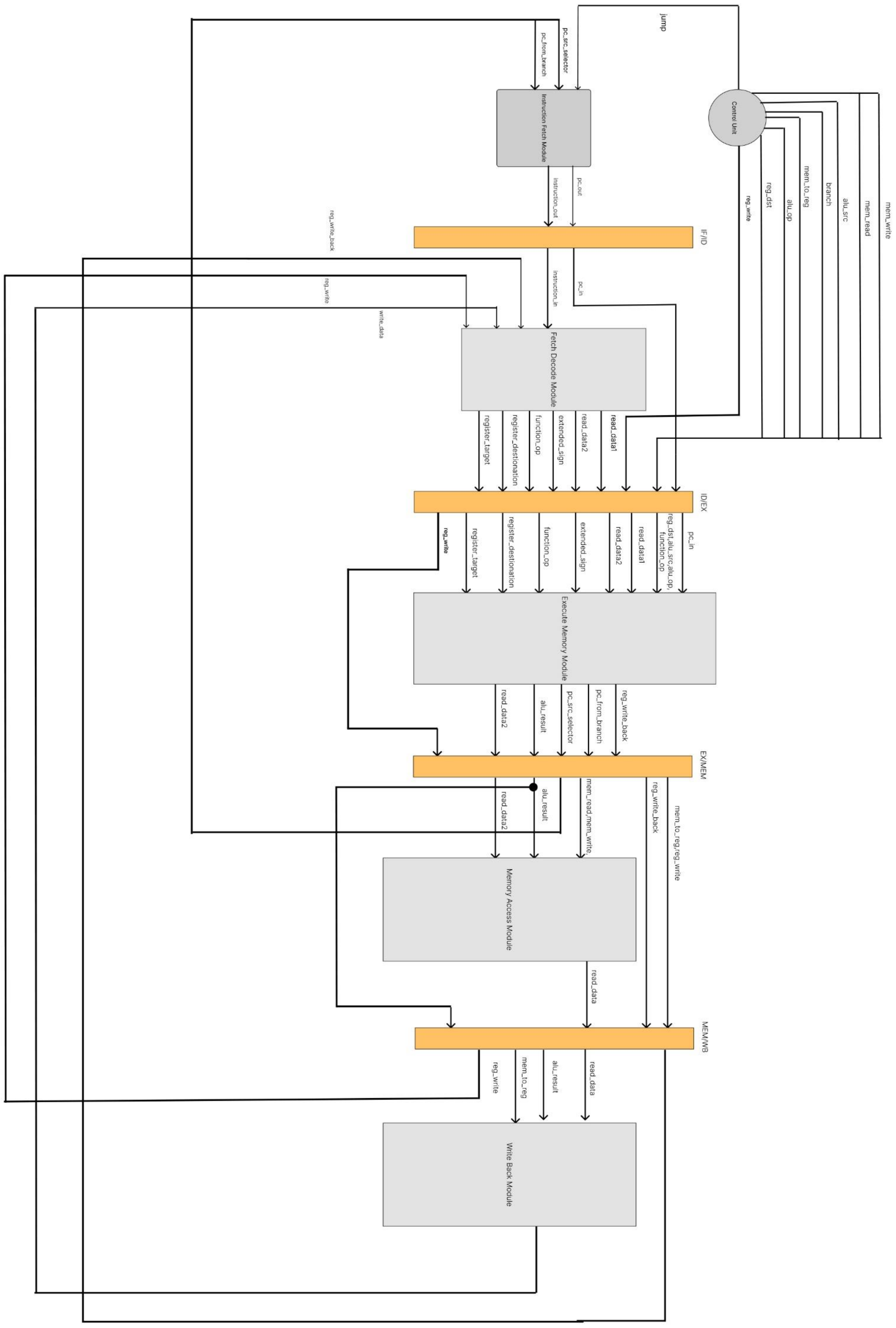
IV. E. WriteBack

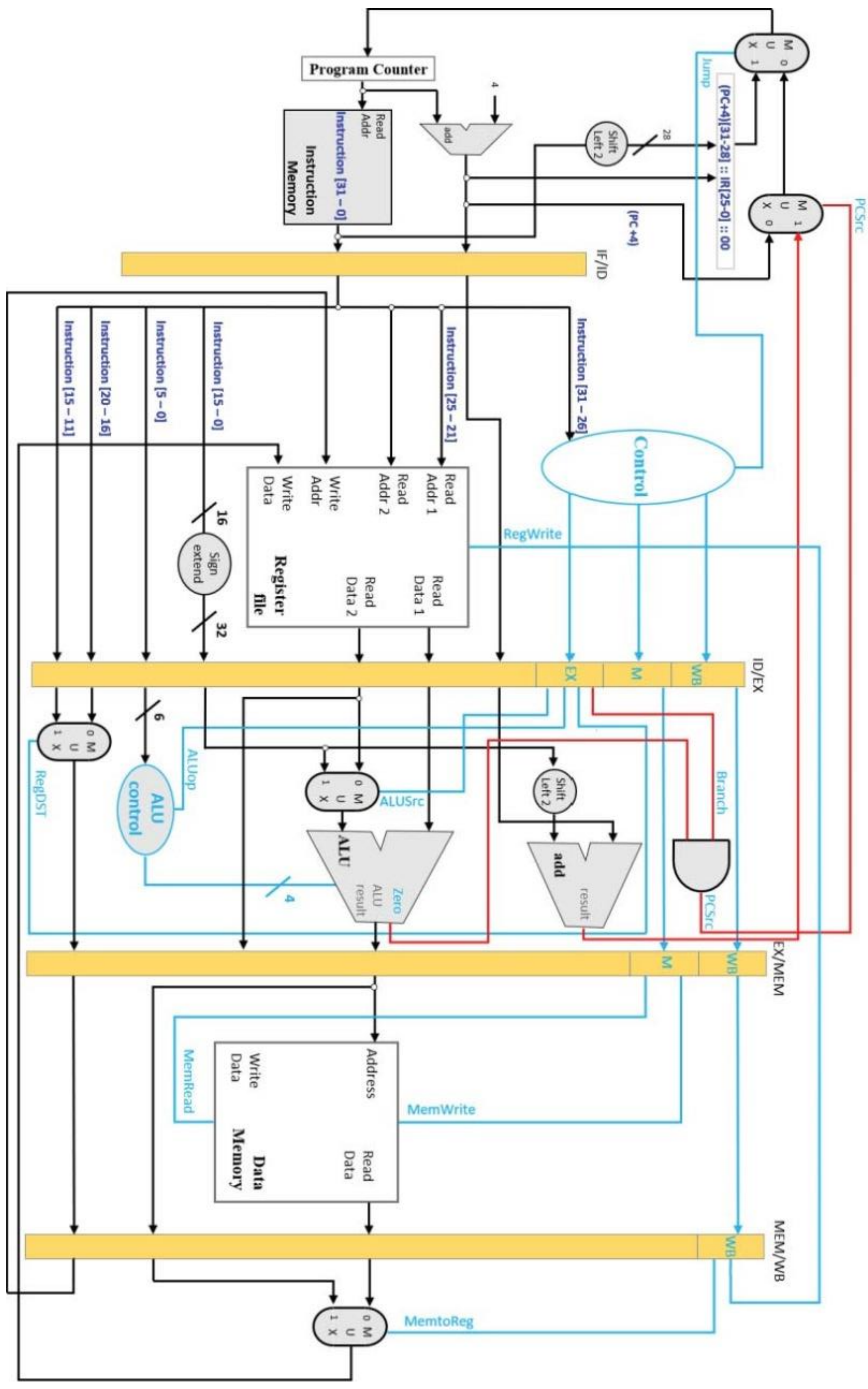
Overview :

The final stage, Memory WriteBack, involves writing the results of executed instructions back to the register file. This section explains how the pipeline ensures that data is correctly written and available for subsequent instructions. Understanding the writeback process is key to ensuring data integrity and the smooth functioning of the pipelined CPU.

Schematics :




jump
pc_src_selector
pc_from_branch
Instruction Fetch Module
Control Unit
mem_write
mem_read
alu_src
branch
mem_to_reg
alu_op
reg_dst
reg_write
instruction_out
pc_out
IF/ID
reg_write_back
reg_write
write_data
instruction_in
pc_in
Fetch Decode Module
register_target
register_destionation
function_op
extended_sign
read_data2
read_data1
ID/EX
reg_write
register_target
register_destionation
function_op
extended_sign
read_data2
read_data1
reg_dst,alu_src,alu_op, function_op
pc_in
Execute Memory Module
read_data2
alu_result
pc_src_selector
pc_from_branch
reg_write_back
EX/MEM
read_data2
alu_result
mem_read,mem_write,
reg_write_back
mem_to_reg,reg_write
Memory Access Module
read_data
MEM/WB
reg_write
mem_to_reg
alu_result
read_data
Write Back Module



Program Counter
4
add
Read Addr
Instruction [31 – 0]
Instruction Memory
Shift Left 2
28
(PC+4)[31-28] :: IR[25-0] :: 00
(PC+4)
Jump
PCSrc
M U X 0 1
M U X 1 0
IF/ID
Instruction [31 – 26]
Instruction [25 – 21]
Instruction [15 – 0]
Instruction [5 – 0]
Instruction [20 – 16]
Instruction [15 – 11]
Control
RegWrite
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
Register file
16
Sign extend
32
ID/EX
EX
M
WB
Branch
Shift Left 2
add
result
ALUSrc
ALU
Zero
ALU result
4
6
ALUop
ALU control
RegDST
EX/MEM
MemRead
MemWrite
Address
Write Data
Read Data
Data Memory
MEM/WB
MemtoReg

II. Control Unit

Overview :

The Control Unit module is a key component in a CPU pipeline responsible for generating control signals based on the current instruction's opcode. These control signals direct the operation of other components in the CPU, such as the ALU, memory, and registers.

Architecture :

The architecture of the Control_Unit module uses a process triggered by the falling edge of the clock signal. Within this process, a case statement is used to generate control signals based on the value of the opcode.

R-type Instructions (000000): Sets the reg_dst, mem_to_reg, and reg_write signals. Sets alu_op to 10. Disables jump, branch, mem_read, mem_write, and alu_src.

I-type Instructions (001100, 001000): Sets reg_dst to 0 and reg_write to 1. Adjusts other signals based on the specific I-type instruction.

Load Word (100011): Sets reg_dst to 0, mem_to_reg to 1, and alu_src to 1. Enables mem_read and reg_write. Disables jump, branch, and mem_write.

Store Word (101011): Sets alu_src to 1 and mem_write to 1. Disables reg_dst, mem_to_reg, reg_write, mem_read, branch, and jump.

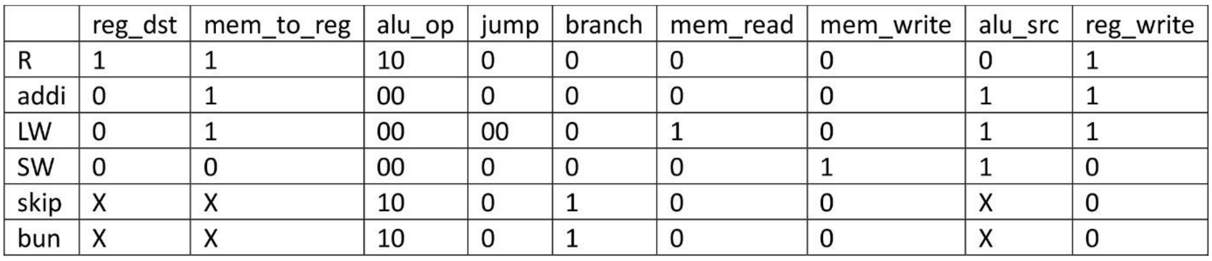
Branch Instructions (000100, 100001, 000101): Configures control signals for branching (branch set to 1). Adjusts ALU operation codes accordingly.

Jump (000010): Sets jump to 1. Disables other control signals.

Default (others): Provides default control signal values for unrecognized opcodes.

Control Unit Codes :

is typically derived from a combination of the instruction's function field (for R-type instructions) and the main opcode field. The following is an example mapping for R-type instructions:



	reg_dst	mem_to_reg	alu_op	jump	branch	mem_read	mem_write	alu_src	reg_write
R	1	1	10	0	0	0	0	0	1
addi	0	1	00	0	0	0	0	1	1
LW	0	1	00	00	0	1	0	1	1
SW	0	0	00	0	0	0	1	1	0
skip	X	X	10	0	1	0	0	X	0
bun	X	X	10	0	1	0	0	X	0

III. Hazard Detection Unit

Overview :
The Control Unit module is a key component in a CPU pipeline responsible for generating control signals based on the current instruction's opcode. These control signals direct the operation of other components in the CPU, such as the ALU, memory, and registers.

Architecture :
Architecture The architecture of the Hazard_Detection_Unit module employs a process to evaluate various conditions and determine whether stalling is required.

The process is triggered by changes in inputs related to branch instructions, register writes, ALU sources, and memory operations.

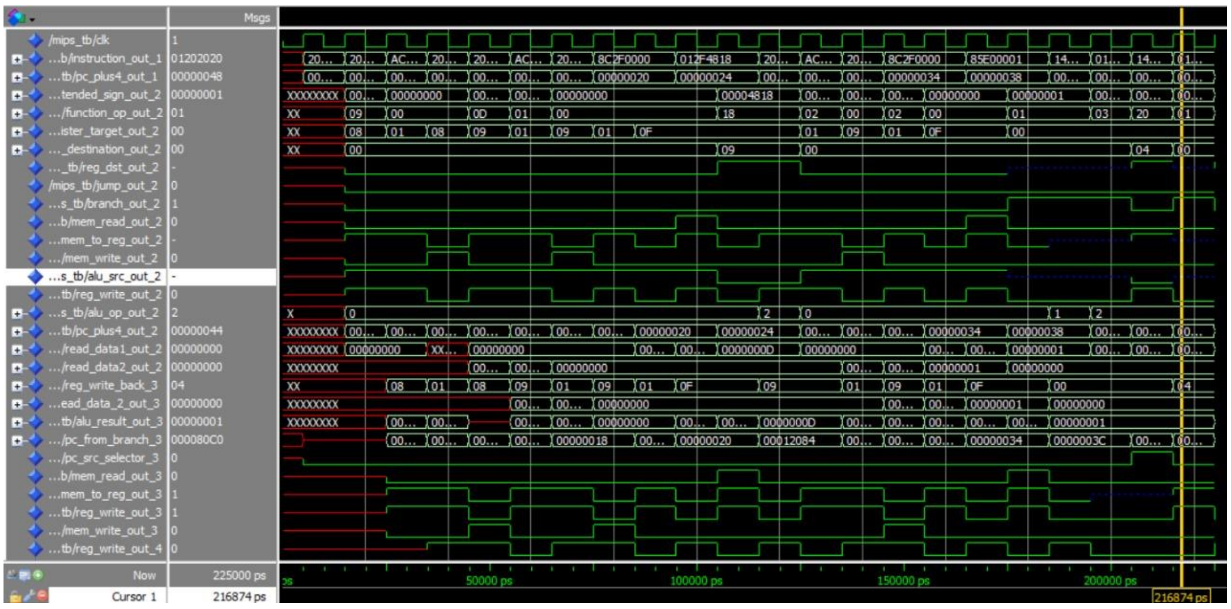
It assesses conditions involving register dependencies and memory operations to identify potential hazards.

If a branch instruction is detected (branch = '1') and there's a register write in the EX stage (reg_write_ID_EX = '1'), the module evaluates different scenarios to determine whether stalling is necessary: Checks for register dependencies between the instruction in the IF stage and the one in the EX stage (reg_source_IF_ID and reg_target_IF_ID compared to reg_target_ID_EX). Considers memory operations (mem_to_reg_ID_EX, mem_to_reg_EX_MEM) to identify potential hazards.

If there's a memory operation in the ID stage and a register write (mem_to_reg_ID_EX = '1' and reg_write_ID = '1'), the module checks for register dependencies and ALU sources to determine whether stalling is necessary.

Overall, the Hazard_Detection_Unit module ensures the smooth execution of instructions by detecting potential hazards and initiating stalling when necessary to avoid data hazards and maintain pipeline integrity.

IV. Wave Testbench Outputs



	Msgs
/mips_tb/clk	1
...b/instruction_out_1	01202020
...tb/pc_plus4_out_1	00000048
...tended_sign_out_2	00000001
.../function_op_out_2	01
...ister_target_out_2	00
..._destination_out_2	00
..._tb/reg_dst_out_2	-
/mips_tb/jump_out_2	0
...s_tb/branch_out_2	1
...b/mem_read_out_2	0
...mem_to_reg_out_2	-
.../mem_write_out_2	0
...s_tb/alu_src_out_2	-
...tb/reg_write_out_2	0
...s_tb/alu_op_out_2	2
...tb/pc_plus4_out_2	00000044
.../read_data1_out_2	00000000
.../read_data2_out_2	00000000
.../reg_write_back_3	04
...ead_data_2_out_3	00000000
...tb/alu_result_out_3	00000001
.../pc_from_branch_3	000080C0
.../pc_src_selector_3	0
...b/mem_read_out_3	0
...mem_to_reg_out_3	1
...tb/reg_write_out_3	1
.../mem_write_out_3	0
...tb/reg_write_out_4	0
Now	225000 ps
Cursor 1	216874 ps

V. Compiler

Nutshell

Convert High level language (called MIPS language) to machine code by using multiple steps.

Steps:

- Lexer
- Parser
- Assembler

Lexer:

Lexer is the lexical analyzer for text conversion into meaningful tokens based on categories. These tokens categorize various parts of the our MIPS language codebase for further processing and manipulation

Parser:

Responsible for tokenization of blocks into statements then rearrange and parse them into AST tree The code goes through lines of MIPS code and identifies patterns .

VI. Assembler

Nutshell:

It converts from assembly language to machine code (binary 0s and 1s) with the same MIPS instruction format whether (R-type , I-type) .

How it functions :

- Reading Assembly code from a text file.
- Matching the required operations with the Mips operations'
- Interpreting the matched MIPS operations with the binary code with the suitable instruction format.
- Output binary code gets transferred to another text file specified for the output

Language Used:
C++

Operations Include :

- inp
- mov
- bun
- out
- skp