Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Computer Architecture ENCS 4370

Design and Verification of a Pipelined RISC Processor Using Verilog

Prepared by:	ID:	Sec No.	Instructor:
Islam Zayed	1230007	3	Dr. Ahmad Afaneh
Sahar Atari	1230971	1	Dr. Ayman Hroub
Sama Alkhader	1230079	1	Dr. Ayman Hroub

January – 2026

Table of Contents

List of Tables

1. Datapath Description and Component Assembly:

1.1 Overview of the Datapath Architecture:

The project implements a 5-stage pipelined datapath designed to handle data hazards through forwarding and stalling. The datapath stages are:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory (MEM)
5. Write Back (WB)

The top-level datapath module connects all stages, pipeline registers, control units, and hazard-handling components.

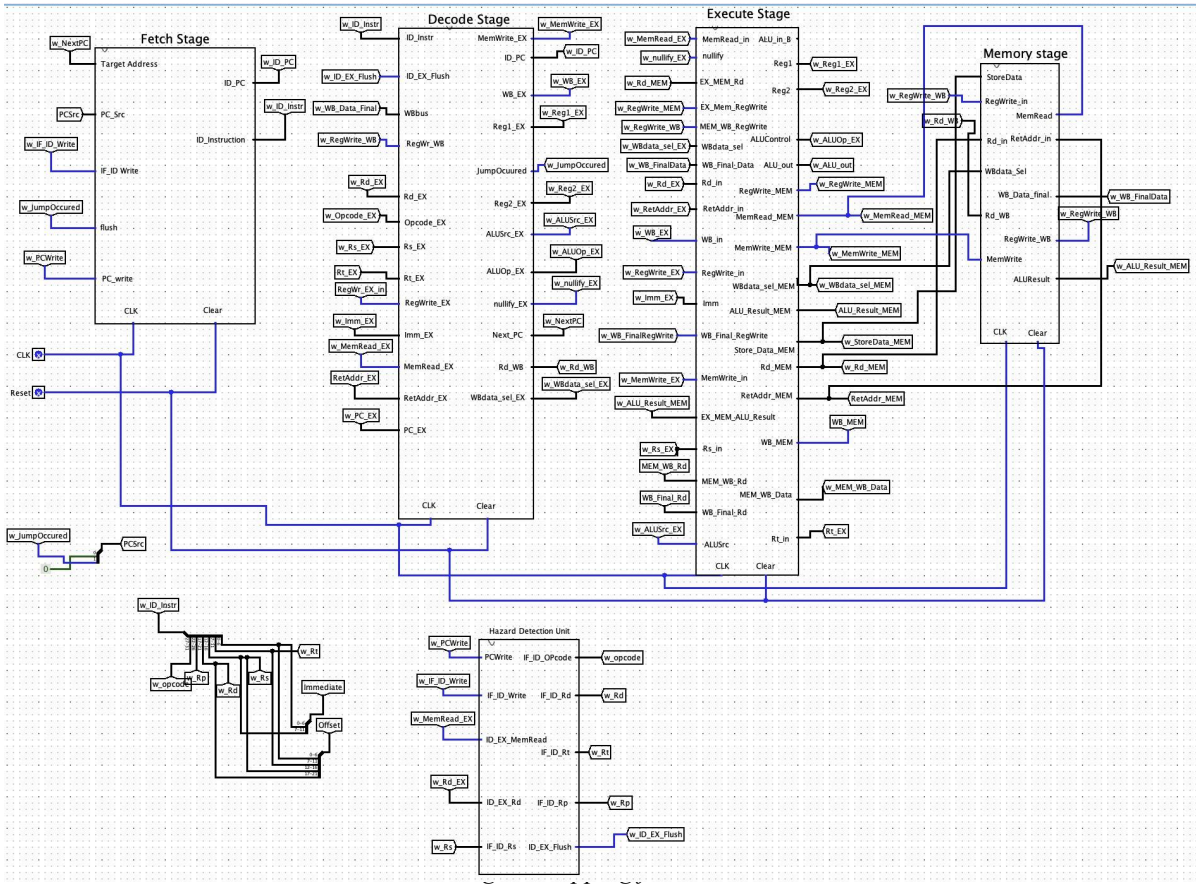


Figure 1: *Complete Datapath Overview of the Pipelined Processor*

1.2 Control Signals, Truth tables, and Boolean Equations:

The main control unit translates the 5-bit opcode into the execution signals required by the rest of the pipeline.

1.2.1. Main Control Unit:

Table 1.1 shows how the main control unit decodes instructions. The signals determine how data flows through the ALU, Memory, and Register File.

Table 1.1: *Control Signal Mapping for Processor Instructions*

Instruction	Opcode	ExtOp	ALUSrc	ALUOp	MemRead	MemWrite	WB data	RegWrite	PCSrc
ADD	0	0= Zero	0=Rt	000	0	0	00	1	00
SUB	1	0= Zero	0=Rt	001	0	0	00	1	00
OR	2	0= Zero	0=Rt	011	0	0	00	1	00
NOR	3	0= Zero	0=Rt	100	0	0	00	1	00
AND	4	0= Zero	0=Rt	010	0	0	00	1	00
ADDI	5	1=Sign	1=Imm	000	0	0	00	1	00
ORI	6	0= Zero	1=Imm	000	0	0	00	1	00
NORI	7	0= Zero	1=Imm	000	0	0	00	1	00
ANDI	8	0= Zero	1=Imm	000	0	0	00	1	00
LW	9	1=Sign	1=Imm	000	1	0	01	1	00
SW	10	1=Sign	1=Imm	000	0	1	x	0	00
J	11	1=Sign	x	xxx	0	0	x	0	10
CALL	12	1=Sign	x	xxx	0	0	10	1	10
JR	13	x	0=Rt	xxx	0	0	x	0	10

Control Signals Description:

- ExtOp: determines if the 12-bit immediate is zero extended (0) or sign extended (1).

$$ExtOp = (ADDI + LW + SW + J + CALL)$$

- ALUSrc: selects the second operand for the ALU (0 for Rt and 1 for immediate).

$$ALUSrc = \overline{(ADD + SUB + OR + NOR + AND + JR)}$$

- MemRead: enables reading from the data memory.

$$MemRead = LW$$

- MemWrite: enables writing to the data memory.

$$MemWrite = SW$$

- RegWrite: enables writing a value back to the register file.

$$RegWrite = \overline{(SW + J + JR)}$$

- WB Data:
  - 00: write ALU result to register.
  - 01: write data memory output to register.
  - 10: write return address (PC+1) to register (specifically used for Call to R31).

1.2.2. PC Control Unit:

The PC control unit is responsible for determining the value of the next program counter in the processor. It supports conditional execution through register-based predication, control-flow changes (jumps and calls), and predicated execution.

Table 1.2 summarizes the behaviour of the PC control unit for different instruction opcodes.

Table 1.2: *PC Control Unit Program Counter Update Rules*

Opcode	Instruction Type	PC Update Rule
11	Jump (PC-Relative)	PC + sign_extended_offset
12	Call (PC-Relative)	PC + sign_extended_offset
13	Jump Register	Rs Data
Others	Normal execution	PC+1

1.2.3. Prediction Logic:

An instruction is allowed to modify the PC only if the predicate condition is met:

$$predicate\_met = (RpIdx = 0) \vee (RpData \neq 0)$$

- If $Rp = R0$, the instruction always executes (unconditional).
- If $Rp \neq R0$ and $RpData \neq 0$, the instruction executes.
- If $Rp \neq R0$ and $RpData = 0$, the instruction doesn't execute.

The PC control unit checks if the predicate condition is satisfied, if the instruction is a jump or call, the next PC is updated accordingly. Otherwise, the default behaviour of the PC control unit is to increment the PC.

1.2.4. Jump Detection Logic:

A jump is detected when the decoded instruction opcode corresponds to a control-flow instruction (Jump, Call, or Jump Register) and the predicate condition associated with the instruction is satisfied. If these conditions are met, the PC control unit asserts the JumpOccured control signal.

$$JumpOccured = predicate\_met \,\&\, (Opcode = Jump \vee Call \vee Jump\ Register)$$

The JumpOccured signal serves two purposes:

1. It informs the fetch stage to select the jump target address instead of PC+1.
2. It triggers the IF/ID register to discard the incorrectly fetched instruction.

1.2.5. ALU Control Unit:

The ALU control unit is responsible for selecting the arithmetic or logical operation performed by the ALY during the execution stage of the pipeline. It generates a 3-bit control signal (`ALUOp`) based on the decoded instruction opcode provided by the main control unit.
Table 1.3 shows the arithmetic and logic operations supported by the ALU.

Table 1.3: *ALU Control Unit Supported Operations*

ALUOp	Operation	Description
000	ADD	Used for arithmetic operations, address calculation, and PC-relative operations

001	SUB	Performs Subtraction
010	AND	Bitwise logical AND
011	OR	Bitwise logical OR
100	NOR	Bitwise logical NOR

1.2.6. Forwarding Unit:

The forwarding unit is the component designed to eliminate data hazards in the pipelined processor without introducing stalls. It allows the processor to use the most recent result of an instruction before it is written back to the register file by directly forwarding data from later pipeline stages to the execute stage. The forwarding unit follows this logic:

- If the destination register in the EX/MEM stage matches the source register in the ID/EX stage and register write-back is enabled, forwarding is performed from the EX/MEM stage.
- Otherwise, if the destination register in the MEM/WB stage or WB stage matches the source register in the ID/EX stage and write-back is enabled, the operand is forwarded directly, allowing the instruction to use the value without waiting for it to be written to the register file.
- If neither condition is satisfied, no forwarding is required.

Table 1.4 shows the encoding of the Forward control signal generated by the forwarding unit.

Table 1.4: *Forwarding Control Signal Encoding*

Forward	Selected Source
00	Use value from ID/EX pipeline register (no hazard)
01	Forward result from EX/MEM stage

10	Forward result from MEM/WB stage
11	Forward result from WB stage

EX/MEM Forwarding:

$$Forward\ 01 \rightarrow EX\_MEM\_RegWrite \wedge (EX\_MEM\_Rd \neq 0) \wedge (EX\_MEM\_Rd = ID\_EX\_Rs)$$

MEM/WB Forwarding:

$$Forward\ 10 \rightarrow MEM\_WB\_RegWrite \wedge (MEM\_WB\_Rd \neq 0) \wedge (MEM\_WB\_Rd = ID\_EX\_Rs)$$

WB Forwarding:

$$Forward\ 11 \rightarrow WB\_RegWrite \wedge (WB\_Rd \neq 0) \wedge (WB\_Rd = ID\_EX\_Rs)$$

1.2.6. Hazard Detection Unit:

The hazard detection unit is responsible for detecting data hazards caused by the load instruction. When a load hazard is detected, the unit temporarily stalls the pipeline by disabling updates to the program counter and the IF/ID pipeline register, while simultaneously flushing the ID/EX pipeline register. This inserts a bubble into the pipeline, allowing the required data to become available before execution continues.

Load-Use Hazard Condition:

Hazard =

$$ID\_EX\_MEMRead \wedge (ID\_EX\_Rd \neq 0) \wedge ((ID\_EX\_Rd = IF\_ID\_Rs) \vee (ID\_EX\_Rd = IF\_ID\_Rt))$$

Table 1.5 summarizes the control decisions generated by the hazard detection unit for detecting load hazards.

Table 1.5: *Hazard Detection Unit Control Logic*

ID/EX.MEMRead	ID/EX.Rd= IF/ID.Rs or IF/ID.Rt	ID/EX.Rd≠0	PCWrite	IF/ID_Write	ID/EX_flush	Action Taken
0	x	x	1	1	0	No hazard, normal execution
1	0	x	1	1	0	No hazard,

						normal execution
1	1	0	1	1	0	No hazard (R0 ignored)
1	1	1	0	0	1	Load-hazard, use stall and flush

2. Pipelined Architecture:

2.1. Instruction Fetch (IF) Stage Architecture:

The implementation of the simplified predicated RISC five-stage pipelined processor is initiated with the architectural design and development of the Instruction Fetch stage. This stage essentially retrieves the pre-programmed instructions from a dedicated instruction memory and calculates the subsequent Program Counter value to be updated on the following clock cycle. The Verilog code implementing the hierarchical structural design of the Instruction Fetch stage is documented in Appendix I as Listing I.1.

The IF stage was constructed by connecting the following four primary components:

- **PC Register:** A 32 - bit register applied to keep track of the address of the current instruction.
- **Next-PC Mux:** A multiplexer employed to select between the sequential address (`PC + 1`) and the jump target address (`Target_Addr`) computed in the Instruction Decode Stage when `JumpOccurred` is asserted.
- **Instruction Memory:** A word-addressable memory module dedicated solely to containing the program's binary code (thus avoiding structural hazards prominent in single-ported memories)
- **IF/ID Pipeline Register:** A synchronization buffer located between the IF and ID stages, required to store the fetched instruction and the PC value to pass them to the following stage.

RTN (Register Transfer Notation)**:**

The operation of the Instruction Fetch stage may be summarized using Register Transfer Notation as follows:

$$Instruction \leftarrow InstructionMem[PC]$$

$$NextPC \leftarrow (JumpOccurred)\ ?\ Target\_Addr : PC + 1$$

$$IF/ID \leftarrow \{PC, Instruction\}$$

Operation:

The next address of the pipelined processor is determined on every rising edge of the system clock. Typically, sequential execution requires the PC to be incremented by 1 (since the

memory is word addressable). Nevertheless, the datapath is subjected to account for two critical conditions that override sequential execution:

- **Jump Instructions:** In an effort to handle control hazards, the PC source is determined in the Decode stage. If a jump instruction is taken, the `PCControlUnit` asserts the `JumpOccured` signal, redirecting the PC to the target address and flushing the IF/ID pipeline register, hence discarding the incorrectly fetched instruction.
- **Stalls:** The `PCWrite` signal is disabled if the `HazardDetectionUnit` of the pipelined processor detects a load-use hazard. The Fetch stage is effectively "frozen," for one cycle as the PC is prevented from updating and a NOP is inserted into the pipeline instead.

Table 2.1 below summarizes the primary control signals and data values associated with the Instruction Fetch stage.

Table 2.1: *Control and Data Signals of the Instruction Fetch (IF) Stage*

Signal	Direction	Description
`PCWrite`	Input	Enables PC updates, disabled by HDU to stall PC during load-use hazards
`IF_ID_Write`	Input	Controls writing to the IF/ID register, disabled by the HDU during load-use hazards
`flush`	Input	Flushes instruction in the IF/ID register if a jump is taken
`ID_PC`	Output	Current PC value passed to the ID stage
`ID_Instruction`	Output	Instruction fetched from the instruction memory and sent to the ID stage

2.2. Instruction Decode (ID) Stage Architecture:

The Instruction Decode stage functions as the control center of the pipelined processor. Its primary responsibilities include decoding the fetched instruction, reading source operands from the register file, generating control signals, and resolving control hazards early in the pipeline. By utilizing complex hardware, this stage is enabled to achieve crucial predication checks and prepare necessary data and control information for the Execute stage. The Verilog code implementing the hierarchical structural RTL design of the Instruction Decode stage is provided in Appendix I as Listing I.2.

The ID stage was constructed by connecting several primary components as follows:

- **Register File**: A module consisting of thirty-two 32-bit general-purpose registers. It has been redesigned to support three simultaneous reads (R_s, R_t and R_p) and a single write operation, diverging from typical two read-port configurations.
- **Main Control Unit**: A combinational logic module that decodes a 5-bit Opcode to generate control signals essential for arithmetic operations, memory access and write-back selection.
- **PC Control Unit**: Evaluates jump and call instructions, predicates instruction execution based on the predicate register, and computes the subsequent Program Counter value. Whenever a control transfer occurs, the unit asserts the `JumpOccurred` signal to resolve potential control hazards.
- **Sign/Zero Extender**: Combinational logic required for extending immediate fields to 32 bits. It performs either sign or zero extension based on the instruction type to ensure operand compatibility with the ALU.
- **ID/EX Pipeline Register**: A synchronization buffer located between the ID and EX stages, required to capture the decoded data and generated control signals before passing them to the Execute stage in the subsequent clock cycle. This register is directly controlled by the `HazardDetectionUnit`, enabling it to be cleared (flushed) or held (stalled) to handle load-use hazards.

RTN (Register Transfer Notation)**:**

The operation of the Instruction Decode stage may be summarized using Register Transfer Notation as follows:

$$Registers \leftarrow RegFile[R_s],\ RegFile[R_t],\ RegFile[R_p]$$

$$ExtImm \leftarrow Extend(Instruction[Imm])$$

$$if(Reg[R_p] == 0\ and\ R_p \neq 0)\ then\ Controls \leftarrow 0$$

$$ID/Ex \leftarrow \{RegValues, R_d,\ Imm,\ Controls\}$$

$$If(JumpTaken)\ then\ NextPC \leftarrow target$$

Operation:

The instruction stored in the IF/ID pipeline register is processed by this stage on each rising edge of the system clock. The opcode, source registers, destination register and predicate register are primarily identified by the stage through parallel decoding of the instruction fields. Consequently, `RegisterFile` access for proper operand retrieval, `MainControlUnit` generation of appropriate control signals based on the opcode, and immediate value extension are all effectively conducted in this stage in preparation for execution in the Execute stage. The Instruction Decode stage is obligated to handle two exceptional conditions as follows:

- **Control Hazard Management**: In an attempt to reduce jump penalties, the processor was designed to resolve jump instructions in the Decode stage.
  - **Target Calculation**: Regardless of whether a jump instruction is present in the stage, the jump target is computed by the `PCControlUnit` as follows:

    $$NextPC = CurrentPC + SignExt(Offset)$$

  - **Jump Resolution**: Given that the predicate condition is met, if the opcode corresponds to a Jump (J), Call (Call), or Jump Register (JR) instruction, the signal `JumpOccurred` is asserted.
  - **Link Register Support**: To support the Call instruction, the logic automatically forces the destination register to 31 (R_{31} is hardwired to the link register), enabling the return address to be saved for subsequent function returns.
- **Predicated Execution and Nullification**: A distinctive feature of the Decode stage is the support for predicated instructions. Every instruction, regardless of the type, contains a predicate register field.

- **Logic**: The value of R_P fetched from the `RegisterFile` is inspected by the hardware. If R_P isn't R_0 and its content equal to 0, the internal signal `w_nullify` is asserted. The instruction is effectively nullified by suppressing register-write and memory-access control signals, so that the instruction occupies a pipeline stage without producing architectural side effects.
- **Signal Gating**: The instruction is prevented from altering the state of the machine by turning it into a No-Operation (NOP). This is effectively achieved by setting `w_nullify` high, thereby forcing the control signals bound for succeeding stages (particularly `RegWrite`, `MemRead` and `MemWrite`) to be zero.

➢ **Hazard and Flushing Handling**: The Instruction Decode stage is obligated to respond to external signals from the `HazardDetectionUnit`.

- `ID_EX_Flush`: If a load-use hazard is detected or a jump is taken, the ID/EX register is cleared by the `ID_EX_Flush` signal. Invalid instructions are thus prevented from advancing into the Execute stage, effectively protecting the processor state from incorrect operations.

Table 2.2 summarizes the primary control signals and data values associated with the Instruction Decode stage.

Table 2.2: *Control and Data Signals of the Instruction Decode (ID) Stage*

Signal	Direction	Description
`RegWrite_EX`	Output	Enables register write in succeeding pipeline stages
`MemRead_EX`	Output	Indicates a memory read operation
`MemWrite_EX`	Output	Indicates a memory write operation

ALUSrc_EX	Output	Selects immediate or register operand for the ALU
ALUOp_EX	Output	Specifies the ALU operation
WBdata_sel_EX	Output	Selects the write-back data source
NextPC	Output	The previously computed next Program Counter value
JumpOccurred	Output	Indicates that a control transfer has occurred

2.3. Execute (EX) Stage Architecture:

The Execute stage of the RISC five-stage pipelined processor conducts all arithmetic and logical operations. Simultaneously, it resolves data hazards through operand forwarding and prepares the resulting data and control signals for the Memory stage. Decoded operands, immediate values, and control signals are provided by the ID/EX pipeline register to this stage. Consequently, this stage produces the computational results and effective addresses required by all subsequent pipeline stages. The Verilog code implementing the hierarchical structural design of the Execute stage is provided in Appendix I as Listing I.3.

The EX stage is composed of the following primary components:

- **Forwarding Units**: Three forwarding units are employed to resolve RAW hazards affecting ALU operands and store-data paths. These units compare the destination registers of instructions currently residing in the EX/MEM and MEM/WB pipeline registers against the source register identifiers (R_s and R_t), as well as the store-data register, of the instructions in the EX stage. In accordance with the detected dependencies, the appropriate forwarding control signals are generated to select the most recent operand values through the `ALU` input multiplexers and the store-data forwarding path.
- **Operand Selection Muxes**: Forwarding signals control the selection of combinational multiplexers selecting between the register file outputs

(`Reg1`/`Reg2`), the EX/MEM ALU result (`EX_MEM_ALU_Result`) and the MEM/WB write-back data (`MEM_WB_Data`). The `ALUSrc` signal controls an additional multiplexer placed at the second input of the **`ALU`** to select either the forwarded second operand or the immediate value.

- **Arithmetic Logic Unit (ALU)**: The `ALUControl` signal is responsible for dictating the specific arithmetic or logical operation executed by the **`ALU`** on its selected operands. The result of the **`ALU`** operation is subsequently propagated to the EX/MEM pipeline register for passage into the next pipeline stage.
- **EX/MEM Pipeline Register**: This pipeline register responsible for capturing the ALU result, the data for store operations, the return address (particularly for the JR instruction), the destination register identifier and all relevant control signals on the rising edge of the system clock cycle. In an effort to prevent incorrect state updates, the nullify signal is asserted to clear the register's control signals. The instruction is, therefore, effectively converted into a No-Operation instruction for the subsequent MEM and WB stages.

RTN (Register Transfer Notation)**:**

The operation of the Execute stage may be outlined using Register Transfer Notation as follows:

$$A \leftarrow Forward(Reg1)$$

$$B \leftarrow Forward(Reg2)$$

$$ALUOut \leftarrow A\ op\ (ALUSrc\ ?\ Imm : B)$$

$$if\ (nullify)\ then\ Controls \leftarrow 0$$

$$EX/MEM \leftarrow \{ALUOut,\ StoreData,\ R_d,\ Controls\}$$

Operation:

The instruction forwarded from the ID/EX pipeline register is processed by the Execute stage, synchronous with each rising edge of the system clock. The foremost responsibility of this stage is to execute arithmetic and logical operations, resolve the majority of RAW data hazards

through operand forwarding, and prepare both results and control signals for the following Memory stage. Commencing with the execution of the instruction, the source operands, immediate values, destination register identifier, and control signals are retrieved from the ID/EX register. Prior to ALU execution, the stage examines register dependencies between the current instruction and instructions residing in the EX/MEM and MEM/WB pipeline stages in order to evaluate potential RAW data hazards.

- **Data Hazard Resolution**: Within the Execute stage, data hazards arising from dependent instructions are resolved through operand forwarding. If a required operand is produced by an instruction in a later pipeline stage, the forwarding logic dynamically reroutes the most recent value to the ALU inputs, and in that way, avoids unnecessary pipeline stalls.
- **Instruction Nullification**: Nullification is enabled in the EX stage through the dedicated `nullify` signal. When high, this signal suppresses side effects by masking write-enable control signals before they are passed into the EX/MEM pipeline register. This technique is crucial, as it completely disables flushed or predicated-false instructions from modifying the architectural state of the pipelined processor.

Table 2.3 presents the primary control signals and data values associated with the Execute stage.

Table 2.3: *Control and Data Signals of the Execute (EX) Stage*

Signal	Direction	Description
`Reg1, Reg2`	Input	Source register operands from the ID stage
`Imm`	Input	Immediate value for ALU operations
`ALUSrc`	Input	Selects either immediate or register operand
`ALUControl`	Input	Specifies the ALU operation
`forwardA, forwardB`	Internal	Control signals from the forwarding units
`nullify`	Input	Suppresses instruction execution by masking control signals

ALU_Result_MEM	Output	The ALU result forwarded to the MEM stage
StoreData_MEM	Output	The store instruction operand forwarded to the MEM stage
Rd_MEM	Output	Destination register identifier
RegWrite_MEM	Output	Register write enable for the MEM stage
MemRead_MEM	Output	Memory read control signal
MemWrite_MEM	Output	Memory write control signal
WBdata_sel_MEM	Output	Write-back data selection

2.4. Memory (MEM) Stage Architecture:

The Memory (MEM) stage of the simplified predicated RISC five-stage pipelined processor is primarily obliged to perform data memory accesses and prepare the final data value to be written back to the register file. Owing to the fact that instructions entering this stage have already completed execution and hazard resolution, the Memory stage is granted operation on fully resolved operands and control signals. The hierarchical structural design implementation of the Memory stage in Verilog code is provided in Appendix I as Listing I.4.

The MEM stage is constructed by interconnecting the following primary components:

- **Data Memory**: Required for load and store instructions support, a synchronous, word-addressable memory module is utilized. The memory is accessed using the effective addresses generated by the **ALU** in the Execute stage. Control signals MemRead and MemWrite control read and write operations, respectively.
- **Write-Data Mux**: A multiplexer whose selection for the appropriate value to be forwarded to the Write-Back stage is controlled by the WBdata_sel signal. The instruction type dictates whether this final value is the data

read from memory, the ALU result, or the return address associated with call instructions.

- **MEM/WB Pipeline Register**: The MEM/WB pipeline register buffers the selected write-back data, destination register identifier, and register-write control signal. This register ensures correct timing synchronization between data and control information as instructions advance into the final pipeline stage (WB stage).

RTN (Register Transfer Notation)**:**

The behavior of the Memory stage may be summarized using the following Register Transfer Notation:

$$if(MemRead)\ then\ MemData \leftarrow DataMem[ALUResult]$$

$$if\ (MemWrite)\ then\ DataMem[ALUResult] \leftarrow StoreData$$

$$WB\_Data \leftarrow (WBdata\_sel == 01)\ ?\ MemData:\ (WBdata\_sel == 10)\ ?\ RetAddr:\ ALUResult$$

$$MEM/WB\ \{WB\_Data,\ Rd,\ RegWrite\}$$

Operation:

On each rising edge of the system clock, the instruction passed by the EX/MEM pipeline register is processed by the Memory stage. At this moment in the pipeline, all operand values and control signals have been fully resolved by the preceding stages, authorizing the Memory stage to operate deterministically without further hazard resolution.

- **Memory Access Handling**: The `DataMemory` module is accessed in read mode, providing that the `MemRead` signal is asserted, so that the data stored at the desired address may be retrieved. Conversely, if the `MemWrite` signal is asserted, the value located on the `StoreData` bus is written to the addressed memory location on the rising edge of the clock. Instructions that do not require memory access leave the Data Memory unit idle during this stage.
- **Write-Back Data Selection**: Succeeding memory access, a combinational mux, controlled by the `WBdata_sel` signal, selects the value to be forwarded to the write-back stage as follows:
  - Load instructions pass on the data read from memory.
  - Arithmetic and logical instruction pass on the ALU result.

- Call instructions pass on the return address provided by the Execute stage.

➢ **Pipeline Register Update**: The selected write-back data, destination register identifier, and register-write enable signal are transferred to the MEM/WB pipeline register on the rising edge of the system clock.

Table 2.4 presents the primary control signals and data values associated with the Memory stage.

Table 2.4: *Control and Data Signals of the Memory (MEM) Stage*

Signal	Direction	Description
ALU_Result	Input	ALU computation result or effective address
StoreData	Input	Data needed to be written to memory only in cases of store instructions
MemRead	Input	Data memory read operation enable
MemWrite	Input	Data memory write operation enable
WBdata_sel	Input	Write-back data source selection
RetAddr_in	Input	Return address for call instructions
WB_Data_final	Output	Selected data forwarded to Write-Back stage
Rd_WB	Output	Destination register identifier
RegWrite_WB	Output	Register write enable for Write-Back stage

2.5. Write-Back (WB) Stage Architecture:

The Write-Back stage is accountable for committing the final result of an instruction to the architectural register file. In our design, the Write-Back stage does not exist as a standalone module as opposed to the preceding stages; instead, it is implemented through the combination of the MEM/WB pipeline register and the register file write interface.

RTN (Register Transfer Notation)**:**

The operation of the Write-Back stage may be outlined using Register Transfer Notation as follows:

$$if\ (RegWrite\_WB)\ then\ RegFile[Rd\_WB] \leftarrow WB\_Data\_Final$$

Operation:

At the end of the Memory stage, the value selected by the write-back multiplexer the following are passed into the MEM/WB pipeline register:

- Either the ALU result, memory read data, or the return address
- The destination register identifier and
- the register-write enable signal

Doing so ensures that all write-back information is fully synchronized before any modifications to the state of the processor.

On the subsequent rising edge of the system clock, the MEM/WB register outputs directly manage the write port of the register file. If the `RegWrite` control signal is asserted and the destination register is valid, the stored value is written into the specified register. The register file remains unchanged in cases when the instruction executed does not produce a register result. Table 2.5 presents the primary control signals and data values associated with the Write-Back stage.

Table 2.5: *Control and Data Signals of the Write-Back (WB) Stage*

Signal	Direction	Description
`RegWrite_WB`	Input	Enables writing to the register file
`WB_Data_Final`	Input	Final data written to the register
`Rd_WB`	Input	Destination register index

3. Processor Component Design and Implementation:

3.1. Register File:

The architectural register storage of the pipelined processor is implemented within the `RegisterFile` module. Designed to support operative reads, controlled writes, and restricted hazard mitigation, this module provided the operand access required for instruction execution in

the pipelined datapath. The Verilog code implementation of the Register File is documented in Appendix I as Listing I.5.

Technical Specifications and Architectural Decisions:

- ➔ Implements 32 general-purpose 32-bit registers (in conformity with the project requirements); therefore, registers are addressed using 5-bit register identifiers.
- ➔ Operand values are available within the same cycle as instruction decoding, enabled by the three combinational read ports (Rs1, Rs2, Rp) provided in the design.
- ➔ One synchronous write port is supported, with register updates occurring on the rising edge of the system clock to ensure precise architectural state updates.
- ➔ Register R0 is hardwired to zero (in conformity with the project requirements) and maintains architectural consistency due to writes being disabled to it.
- ➔ Register R30 is hardwired to the Program Counter (as specified in the design requirements), and enables efficient access to control flow information by automatically updating every cycle with the current PC, eliminating any need for additional datapath logic.
- ➔ Write-back data is forwarded directly to read ports, when source and destination registers match, by local bypass forwarding logic. Write-after-read hazards and pipeline stalls are effectively reduced by the logic.
- ➔ Simulation, verification, and debugging were simplified by initializing some registers with some deterministic values.

3.2. Program Counter (PC) Register:

The maintenance of the address of the current instruction and control of the progression of instruction fetch within the pipelined processor is carried out by the `PC_Register`. Both pipeline stalling and resets are supported by the module to ensure proper sequencing under normal instruction execution and hazard conditions. Appendix I, Listing I.6 presents the Verilog code for the PC register module, which supports synchronous resets and conditional PC updates controlled by the `PCWrite` signal.

Technical Specifications and Architectural Decisions:

- ➔ A 32-bit synchronous register is implemented to store the program counter value

- ➔ Alignment is maintained with the rest of the pipeline registers, with all updates occurring on the rising edge of the system clock.
- ➔ A well defined starting state after system reset is enabled by an asynchronous reset that initializes PC to zero.
- ➔ Incorporates a `PCWrite` enable signal to conditionally allow PC updates; therefore, the design inherently provides direct support for pipeline stalls resulting from hazards or control dependencies.
- ➔ Instruction fetch is prevented from advancing when the PC value is held constant (by setting the `PCWrite` signal to low), thereby preserving pipeline correctness.

3.3. Instruction Memory:

Read-only access to program instructions required during the Instruction Fetch stage is provided by the `InstructionMemory` module. The assumption of a fixed program image loaded prior to execution is reflected by implementing the module as a statically initialized memory array. The Verilog code implementation of the Instruction Memory module is documented in Appendix I as Listing I.7.

Technical Specifications and Architectural Decisions:

- ➔ Memory Organization: The module is implemented as a 1024-word array of 32-bit instructions, supporting word-addressable access. The instruction located at the address specified by the PC is directly outputted along with additional control signals.
- ➔ Combinational Read Access: Prevention of additional latency introduced into the IF stage is ensured by the purely combinational instruction fetch.
- ➔ Static Initialization: Instructions are loaded in an initial block to define a deterministic test program. The program was deliberately structured to execute:
  - ◆ Predicated execution and nullification
  - ◆ Load-use data hazards
  - ◆ Control hazards due to CALL and JR instructions
  - ◆ Pipeline flushing and stalling behavior
- ➔ The absence of write ports enforces read-only operation, consistent with conventional instruction memory.

3.4. Control Units:

3.4.1. Main Control Unit:

The `MainControlUnit` is accountable for control signal generation based solely on the instruction opcode. Instruction decoding is performed by the module during the ID stage, as well as setting the datapath configuration for subsequent pipeline stages. The module lacks any contribution to pipeline latency, given that the unit is implemented purely as combinational logic. The Verilog code implementation of the Main Control Unit module is provided in Appendix I as Listing I.8.

Technical Specifications and Architectural Decisions:

- ➔ Opcode-Based Decoding: Control signals for each instruction are derived exclusively from the 5-bit opcode field, simplifying instruction decoding.
- ➔ Combinational Design: The control logic is implemented using an `always @(*)` block to ensure that control outputs update instantaneously in response to opcode changes, independent of the system clock.
- ➔ Default Initialization: Upon commencing with the decoding process, all control outputs are initially cleared to prevent unintended side effects from invalid opcodes.
- ➔ ALU Control: An `ALUOp` code is generated by the unit to be refined later by the ALU control logic.
- ➔ Immediate and Memory Support: Supports Immediate and Memory instructions by asserting `ALUSrc` (for operand selection) and setting `ExtOp` (for sign extension). Load and store instructions additionally assert memory access signals and select the appropriate write-back source.
- ➔ Call Instruction Handling: Efficient link register updates are ensured by enabling the CALL instruction to bypass the ALU result and memory data paths by simply selecting a return address through `WBdata = 2'b10`.

3.4.2. PC Control Unit:

Determination of the next program counter value and control-flow instruction resolution are performed by the `PCControlUnit` during the Instruction Decode stage. The unit enables rapid control-flow redirection while minimizing jump penalties by evaluating opcodes and

predicate conditions early in the pipeline. The Verilog code implementation of the PC Control Unit module is provided in Appendix I as Listing I.9.

Technical Specifications and Architectural Decisions:

- ➔ Combinational Control Logic: Implemented entirely as combinational logic, the unit enables the next PC value to be computed without waiting for a clock edge, further supporting early jump resolution.
- ➔ Predicated control flow: Control-flow instructions are conditionally executed after evaluating the predicate. Provided that the predicate register index is R0 or the predicate register value is non-zero, a jump is permitted.
- ➔ Sign Extended Offset Computation: Both forward and backward control transfers relative to the current PC are enabled because jump and call targets are calculated using a sign-extended 22-bit offset.
- ➔ Opcode-Based PC Selection: The module decodes the opcode to distinguish between the following instructions requiring distinctive handling:
  - ◆ Jump (J) and Call (CALL) instructions: the target is calculated as `PC + SignExtOffset`
  - ◆ Jump Register (JR) instructions: the target is loaded directly from a source register
- ➔ Default Sequential Execution: When no valid jump instruction is detected, the next PC defaults to `PC+1`, thereby preserving normal sequential instruction execution flow.
- ➔ Jump Detection: When a control-flow redirection is taken, the `JumpOccurred` signal is asserted. This enables subsequent logic to flush invalid instructions from the pipeline and maintain overall correctness.

3.4.3. Hazard Detection Unit:

To ensure correct pipeline execution, the `HazardDetectionUnit` detects load-use hazards that cannot be resolved through forwarding. If a hazard is identified, the unit temporarily stalls the pipeline to prevent consumption of unavailable data. The Verilog code implementation of the Hazard Detection Unit module is provided in Appendix I as Listing I.10.

Technical Specifications and Architectural Decisions:

- ➔ Identification of Load-Use Hazards: The module monitors the instruction in the ID/EX stage and inspects whether it performs a memory read (`ID_EX_MemRead`). If the destination register of this instruction is determined to match any operand required by the instruction in the IF/ID stage, a hazard is detected.
- ➔ Dependency checking with predicate constraints: Beyond the conventional source registers (Rs and Rt), the predicate register (Rp) is also examined to preserve validity under predicated execution.
- ➔ Store Instruction Handling: The unit determines dependencies involving the register that supplies the store data. This rigorous checking effectively prevents store operations from using incorrect, stale values that may follow a preceding load instruction, thereby ensuring data consistency in memory.
- ➔ Pipeline Stall Implementation:

  Upon recognition of a hazardous condition:
  - ◆ `PCWrite` is set low to effectively freeze the Program Counter
  - ◆ `IF_ID_Write` is set low to stall the IF/ID register
  - ◆ `ID_EX_Flush` is asserted to insert a bubble into the Execute stage.
- ➔ Default Operation: In the absence of detected hazards, all control signals remain enabled, authorizing continuous, uninterrupted flow of instructions through the pipeline.

3.4.4. Forwarding Unit:

RAW data hazard resolution in the Execute stage without introducing pipeline stalls is accomplished by the **`Forwarding_Unit`**. It dynamically selects the most recent value of a source register by comparing the source register identifier in the ID/EX stage with destination registers in the EX/MEM and MEM/WB stages. The Verilog code implementation of the Forwarding Unit module is provided in Appendix I as Listing I.11.

Technical Specifications and Architectural Decisions:

- ➔ To detect dependencies, `ID_EX_Rs` is compared against `EX_MEM_Rd` and `MEM_WB_Rd`.
- ➔ Priority is given to the EX/MEM stage so that the newest available result is forwarded.

- ➔ Forwarding from the EX/MEM stage is excluded when the instruction is a load (detected if `EX_MEM_MemRead` is asserted), as the load data isn't available.
- ➔ A 2-bit control signal (`Forward`) is outputted by the module, which is needed to drive operand muxes in the Execute stage.
- ➔ Unnecessary pipeline stalls are eliminated, improving overall pipeline throughput

3.5. Arithmetic Logic Unit (ALU):

The Arithmetic Logic Unit (**`ALU`**) conducts all arithmetic and logical computations required by the pipelined processor. It is implemented using purely combinational logic and executes the operation specified by a dedicated control signal generated in the Decode stage. The Verilog code implementation of the ALU module is provided in Appendix I as Listing I.12.

Technical Specifications and Architectural Decisions:

- ➔ Both arithmetic (ADD, SUB) and logical (AND, OR, NOR) operations are supported
- ➔ Control logic is simplified by directly selecting the ALU operation using the 3-bit `ALUControl` signal. This dedicated signal, derived from the instruction opcode and the main control unit, efficiently dictates the required arithmetic or logical computation.
- ➔ Either computation results or effective memory addresses, needed for load and store instructions, are produced by the unit.

3.6. Data Memory:

The data storage subsystem of the pipelined processor is implemented in a dedicated **`DataMemory`** module. This unit also provides support for both load and store instructions during the Memory stage. It is addressed using the effective address generated by the Execute stage and operates under the control of read and write enable signals. The Verilog code implementation of the Data Memory module is provided in Appendix I as Listing I.13.

Technical Specifications and Architectural Decisions:

- ➔ A word-addressable memory array with 1024 entries is implemented to suffice for program data storage and testing.
- ➔ Read and write behavior of the Data Memory unit is strictly separated as follows:
  - ◆ Read operations are controlled by the `MemRead` signal

- Write operations are synchronized to the rising clock edge of the system clock and controlled by the `MemWrite` signal

➔ A small delay is introduced to ensure that the memory inputs have stabilized. This stabilization is necessary because the memory address and the store data update simultaneously with the Memory stage's attempt to access (read or write) the memory array.

➔ Some memory contents were pre-initialized to simplify functional verification of load and store behavior.

3.7. Pipeline Registers:

Pipeline registers are employed to isolate the five stages of the processor and ensure proper timing synchronization of data and control signals as instructions advance through the pipeline. Implemented as a synchronous storage device, each pipeline register incorporates explicit support for:

➢ Reset: To initialize the pipeline state

➢ Stalling: To halt the flow of instructions for data hazards

➢ Flushing: To nullify instructions for control hazards

This integrated support is essential for hazard and control-flow management as instructions traverse the pipeline.

3.7.1. IF/ID:

The fetched instruction and its corresponding program counter value are buffered by the IF/ID register between the Instruction Fetch and Instruction Decode stages. The Verilog code implementation of the IF/ID module is provided in Appendix I as Listing I.14.

Technical Specifications and Architectural Decisions:

➔ The current PC and instruction word are stored here for use in the Decode stage

➔ A `write_en` signal is included to support pipeline stalling during load-use hazards

- ➔ Flushing is supported to discard incorrectly fetched instructions following control-flow changes
- ➔ An asynchronous reset is used to ensure the pipeline starts from a valid, known state.

3.7.2. ID/EX:

Decoded operands, immediate values, register identifiers and control signals are transferred from the Decode stage to the Execute stage by the ID/EX register. The Verilog code implementation of the ID/EX module is provided in Appendix I as Listing I.15.

Technical Specifications and Architectural Decisions:

- ➔ Both data operands and all control signals relative to execution are carried by the unit
- ➔ A NOP is inserted on hazards or jumps by an explicit `ID_EX_Flush` mechanism incorporated in the unit.
- ➔ Propagation of the nullify signal enables support for predicated execution.
- ➔ Decoded instructions are selectively advanced by an enable control signal.
- ➔ Control signals cleared on flushes to prevent invalid instructions from executing.

3.7.3. EX/MEM:

Execution results and control signals relevant to the memory are buffered by the EX/MEM register between the Execute and Memory stages. The Verilog code implementation of the EX/MEM module is provided in Appendix I as Listing I.16.

Technical Specifications and Architectural Decisions:

- ➔ Stores ALU results, store data, the return address, and the destination register identifier
- ➔ Preserves memory access and write-back control signals
- ➔ Proper synchronization of data and control is ensured for memory operations.

3.7.4. MEM/WB:

Final results and control information are held by the MEM/WB register prior to the Write-Back stage. The Verilog code implementation of the MEM/WB module is provided in Appendix I as Listing I.17.

Technical Specifications and Architectural Decisions:

➔ The selected write-back and destination register identifier are buffered.

➔ Register write enable signal is propagated in a controlled manner.

➔ Correct write-back ordering and architectural state updates are guaranteed.

4. Simulation and Verification:

The primary objective of this simulation is to perform a Stress-Based Functional Verification of the 5-stage Datapath. A complete comprehensive test case log for the pipelined processor may be referred to in Appendix II under Listing II.1. Rather than testing individual instructions in isolation, this "Full-Spectrum" test validates the complex interactions between the following hardware sub-systems:

- **A. Hazard Detection Unit (HDU):** Verification of hardware-level interlocks (stalling) when a Load-Use dependency occurs.
- **B. Forwarding Unit (FU):** Validation of Data Bypass paths (EX-EX and MEM-EX) to ensure zero-cycle latency for R-type dependencies.
- **C. Predication Logic (Conditional Execution):** Verification of the "Killed Instruction" mechanism where the hardware nullifies the RegWrite signal based on a predicate register (Rp).
- **D. Control Flow & Branch Recovery:** Verification of the Branch/Jump logic, specifically the CALL instruction and the subsequent flushing of the "Jump Shadow" to maintain instruction atomicity.

Simulation Environment Setup:

Before analyzing the hardware behavior, we define the initial state of the processor and the specific instruction sequence used for the stress test.

Register File Initialization

To ensure a clean environment for testing data dependencies, the Register File is initialized with known values. Registers R0 through R16 are loaded with their own index values (e.g., R1=1, R2=2), while R17 through R31 are cleared to zero.

```
 initial begin
        for (i = 0; i < 32; i = i + 1) registers[i] = (i < 17) ? i :
32'h0;
```

```
        registers[0] = 0;
    end
```

Instruction Memory (Program Code)

The following assembly-level equivalent is loaded into the instruction memory. This sequence is carefully crafted to trigger every hazard-handling mechanism in the datapath.

```
// --- PHASE 1: Forwarding & Priority (I-Type & R-Type) ---
    // Format I: Op(5), Rp(5), Rd(5), Rs(5), Imm(12)
    instMemory[0] = {5'd5, 5'd0, 5'd1, 5'd0, 12'd10};  // ADDI R1, R0, 10
    instMemory[1] = {5'd5, 5'd0, 5'd1, 5'd1, 12'd5};   // ADDI R1, R1, 5  (EX-EX
Forward: R1=15)
    instMemory[2] = {5'd5, 5'd0, 5'd1, 5'd1, 12'd2};   // ADDI R1, R1, 2  (Priority
Test: R1=17)

    // Format R: Op(5), Rp(5), Rd(5), Rs(5), Rt(5), Unused(7)
    instMemory[3] = {5'd0, 5'd0, 5'd2, 5'd1, 5'd1, 7'd0}; // ADD R2, R1, R1 (R2
should be 34)

    // --- PHASE 2: Load-Use & Memory (I-Type) ---
    instMemory[4] = {5'd10, 5'd0, 5'd2, 5'd0, 12'd100}; // SW R2, 100(R0) (Store
34)
    instMemory[5] = {5'd9,  5'd0, 5'd3, 5'd0, 12'd100}; // LW R3, 100(R0) (R3 = 34)
    instMemory[6] = {5'd0,  5'd0, 5'd4, 5'd3, 5'd3, 7'd0}; // ADD R4, R3, R3 (STALL
check: R4=68)

    // --- PHASE 3: Predication (Nullify) ---
    instMemory[7] = {5'd5, 5'd0, 5'd5, 5'd0, 12'd0};    // R5 = 0 (Predicate False)
    instMemory[8] = {5'd5, 5'd0, 5'd6, 5'd0, 12'd1};    // R6 = 1 (Predicate True)

    // KILLED: Rp=R5 (0), so ADDI R7 shouldn't happen
    instMemory[9] = {5'd5, 5'd5, 5'd7, 5'd0, 12'd100};
    // EXECUTED: Rp=R6 (1), so ADDI R8 should happen
    instMemory[10]= {5'd5, 5'd6, 5'd8, 5'd0, 12'd100};  // R8 = 100

    // --- PHASE 4: J-Type Call & Flush ---
    // Format J: Op(5), Rp(5), Offset(22)
    // CALL +3 if R6 != 0. Jumps to PC 14, writes PC+1 (12) to R31.
    instMemory[11] = {5'd12, 5'd6, 22'd3};

    instMemory[12] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED
    instMemory[13] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED

    // --- PHASE 5: Indirect Jump (I-Type style) ---
    // Some architectures use I-Type for JRs to keep the Rs field.
    instMemory[14] = {5'd5, 5'd0, 5'd9, 5'd0, 12'd17};  // R9 = 17
```

```
    instMemory[15] = {5'd13, 5'd6, 5'd0, 5'd9, 12'd0};  // J Rs (Jump to R9=17)
    instMemory[16] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED

    instMemory[17] = {5'd5, 5'd0, 5'd11, 5'd31, 12'd0}; // Final Check: R11 = R31
(12)
```

Phase 1 Analysis: Forwarding & Priority Logic:

Objective: To verify that the Forwarding Unit (FU) can resolve back-to-back data dependencies without stalling and that it correctly prioritizes the most recent data (EX/MEM stage over MEM/WB).

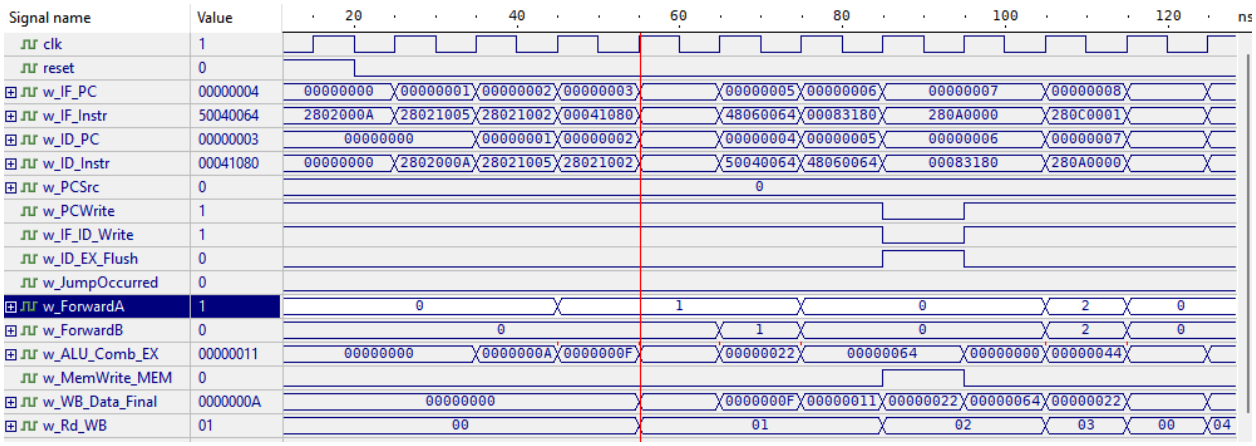
Results & Discussion:

Phase 1 executes a sequence of three ADDI instructions targeting R1. This creates a "Moving Target" dependency where each instruction requires the result of the one immediately preceding it.

- **Dependency Handling:** The hardware identifies that the source register for the current instruction in the Execute (EX) stage matches the destination register of the instruction in the Memory (MEM) stage.
- **Bypass Execution:** Instead of waiting for a Write-Back to the Register File, the Forwarding Unit asserts the control signals to pull data directly from the pipeline registers.
- **Verification of Priority:** The test confirms that the FU correctly prioritizes the most recent result (from the instruction at PC+1) over older values, preventing "stale" data from entering the ALU.

Waveform Verification:

The provided waveform serves as the primary proof of this architectural behavior:



- **Control Signal Assertions:** At the **45ns** mark, the signal w_ForwardA transitions to **1**. This is the physical "trigger" showing the multiplexer has switched from the Register File input to the forwarded MEM-stage input.
- **Data Flow:** Following the `w_ALU_Comb_EX` signal, we observe the values updating in real-time:
    - **First,** the ALU output is `0000000A` (10).
    - **Then,** the ALU output updates to `0000000F` (15).
    - **Lastly,** the output settles at `00000011` (17).
- **Conclusion:** The lack of any "flat" or "zero" gaps in the ALU output during these transitions proves the forwarding paths are correctly balanced and functioning at the full clock frequency.

Kernel Snapshot:

To verify the final architectural commit after the waveform sequence, we look at the specific Kernel print for the Write-Back stage:

```
[CYCLE 5] [WB STAGE] WriteBack: Writing Data 00000011 to Register R 1
```

Phase 2 Analysis: Load-Use Dependency & Stalling:

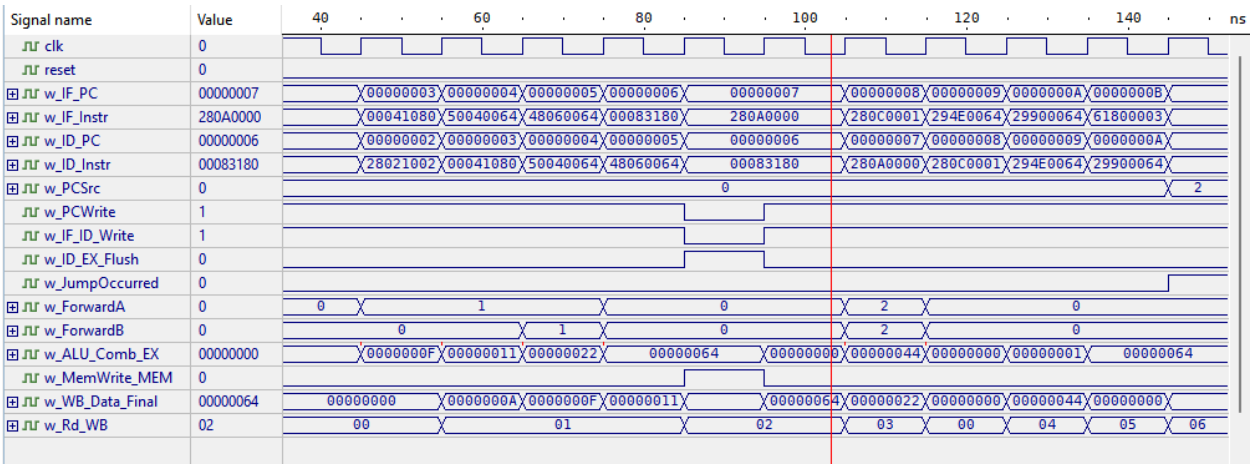
Objective: To verify the **Hazard Detection Unit (HDU)**. The hardware must detect a Load-Use hazard, where an instruction requires data from a LW still in progress and force a 1-cycle pipeline stall to maintain data integrity.

Results and Discussion

In this phase, a `LW R3, 100(R0)` is immediately followed by an `ADD R4, R3, R3.`

- **The Problem:** The data for R3 is not available until the end of the Memory (MEM) stage.
- **The Solution:** The HDU must freeze the Fetch and Decode stages for one cycle to allow the LW to reach the MEM stage, where the data can then be forwarded.
- **Control Action:** The hardware de-asserts `PCWrite` and `IF/ID_Write`, while asserting `ID_EX_Flush` to insert a "bubble" (NOP) into the execution flow.

Waveform Verification:



The waveform provides the direct physical proof of this interlock mechanism:

- **Signal Interlock:** At **85ns**, both `w_PCWrite` and `w_IF_ID_Write` drop to **0**. This is the HDU "freezing" the pipeline.
- **PC Stall:** Observe `w_IF_PC`. It remains at 00000007 for two consecutive clock periods. This confirms the Fetch stage was successfully halted.
- **Bubble Injection:** The signal `w_ID_EX_Flush` transitions to **1**. This ensures the ADD instruction in the Decode stage does not move into Execution with incorrect data.
- **Post-Stall Forwarding:** After the stall is released at **95ns**, the hardware successfully executes the ADD.

Kernel Snapshot

To confirm exactly why the stall occurred, we look at the specific Kernel output during the hazard detection cycle:

```
[CYCLE 6] Stall/Flush: PCWrite=0 | IF/ID_Write=0 | Flush=0 Control : ID_EX_Flush=1 [ID STAGE] Decoding : Op= 0 | Rs= 3 | Rt= 3
```

Phase 3 Analysis: Predication (Instruction Nullification)

Objective: To verify the **Conditional Execution** (Predication) sub-system. The goal is to prove the hardware can "kill" an instruction in the pipeline, preventing it from writing to registers if the value in its assigned predicate register (Rp) does not meet the execution criteria.

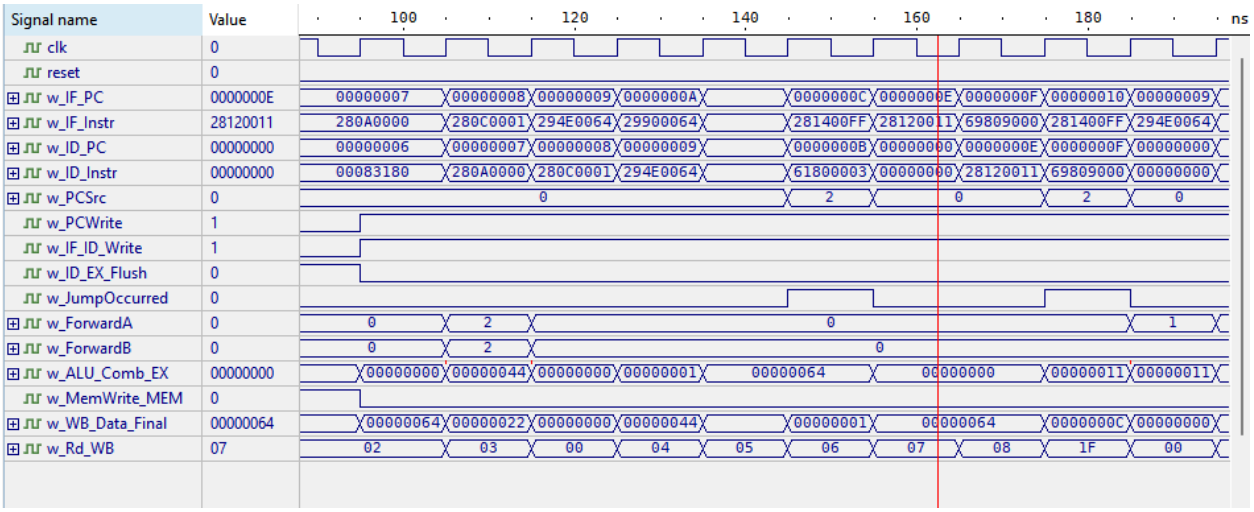
Results and Discussion

In this phase, the test targets the interaction between the Decode stage (where the predicate is read) and the Execute stage (where the write-enable signal is suppressed).

- **The Condition:** Instruction at PC 0x09 is an ADDI that specifies R5 as its predicate register (Rp = 5).
- **The State:** Earlier in the test, R5 was initialized to 0.
- **The Logic:** Upon decoding, the hardware detects that the value in the specified predicate register is zero, triggering the Predicate_Kill signal.
- **The Result:** The instruction remains in the pipeline to maintain timing, but it is effectively converted into a "ghost" instruction with no architectural impact.

Waveform Verification

We verify the success of the predication logic by tracking the instruction from Decode to its expected Write-Back point on the waveform:



- **Instruction Identification:** The instruction `29900064` (`PC 0x09`) is processed in the `w_ID_Instr` bus.
- **Execution Outcome:** In the EX stage, the `w_ALU_Comb_EX` signal shows the calculated result `00000001`.
- **The Nullification Proof:** Observation of the `w_Rd_WB` (Register Destination) bus shows that it does not transition to register **07** for this instruction.
- **Observation:** The physical absence of R7 on the destination bus confirms that the instruction was successfully nullified and prevented from writing to the Register File.

Kernel Snapshot

The Kernel Trace provides the cycle-by-cycle confirmation of the internal control states that drive the behavior observed in the waveform:

```
[CYCLE 10] [ID STAGE] Decoding : Op= 5 | Rd= 7 | Rp= 5 Control :
Predicate_Kill=1
[CYCLE 11] [EX STAGE] Output : Dest Reg=R 7 | WriteEn=0
```

Phase 4 Analysis: CALL and Pipeline Flush

Objective: To verify the Call instruction's ability to perform a control flow redirect while simultaneously preserving the return address (R31) and clearing the "jump shadow" via a pipeline flush.

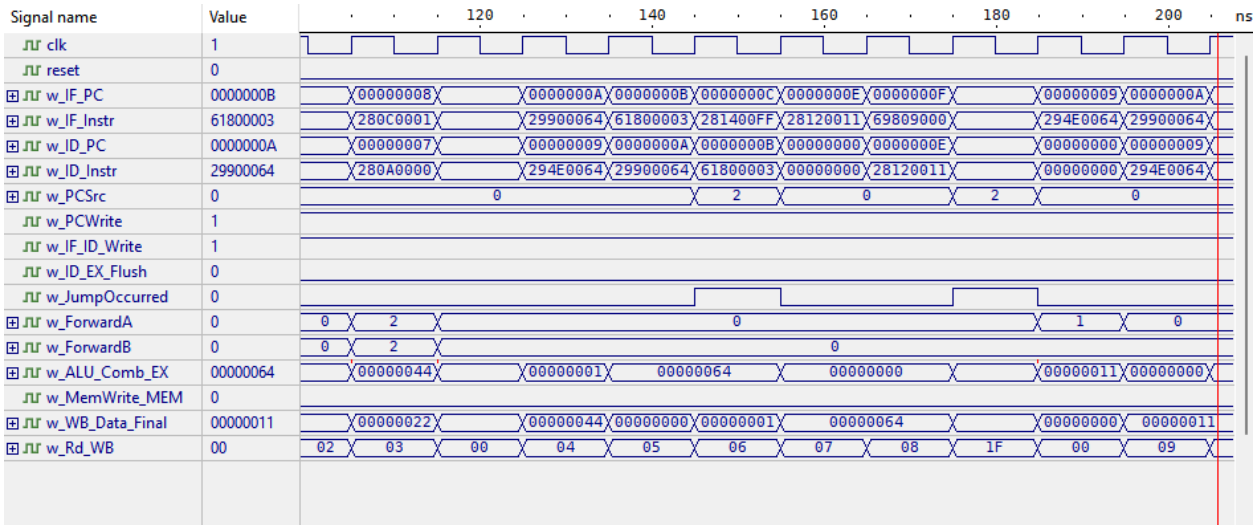
Results and Discussion

In this phase, the Call instruction instruction is fetched at `PC` `0x0B`. Because the target address is non-sequential, the hardware must resolve the branch and clean the pipeline of any incorrectly fetched instructions.

- **The Problem:** The pipeline has already fetched instructions at `PC` `0x0C` and `0x0D` before the Call instruction is fully decoded.
- **The Solution:** The Hazard Detection and Control units must identify the jump, redirect the Program Counter to the target (`0x0E`), and invalidate the instructions currently in the Decode stage.
- **Control Action:** The hardware asserts the `Flush` and `JumpOccurred` signals. Simultaneously, the address following the Call instruction (`0x0C`) is calculated and passed down the pipeline to be stored in the link register (R31).

Waveform Verification

The waveform provides the physical evidence of the control flow redirection and register commit:



- **The PC Jump:** Observe **w_IF_PC**. After the Call instruction at `0x0B`, the PC snaps directly to `0x0E`, successfully bypassing the sequential addresses.

- **The Flush Signal:** During the cycle where the jump is resolved, **w_JumpOccurred** transitions to **1**. Immediately following this, the instruction buses for the ID stage show zeros or NOPs, confirming the pipeline was cleared.
- **Return Address Commit:** Observe the Write-Back stage signals `w_Rd_WB` and `w_WB_Data_Final`. The destination register `w_Rd_WB` transitions to `1F` (Decimal 31). The corresponding data in `w_WB_Data_Final` is `0000000C`, proving the return address `(PC + 1)` was saved correctly.

Kernel Snapshot

The Kernel Trace provides the specific cycle-by-cycle logic that governed the hardware behavior seen in the waveform:

```
[CYCLE 12] [IF STAGE] Next PC : 0000000e Stall/Flush: PCWrite=1 |
IF/ID_Write=1 | Flush=1 [ID STAGE] Control : JumpOccurred=1
[CYCLE 15] [WB STAGE] WriteBack : Writing Data 0000000c to Register
R31 Enable : 1
```

Phase 5 Analysis: Indirect Jump (JR)

Objective: To verify the **JR (Jump Register)** instruction logic, ensure the Program Counter (PC) correctly redirects using a dynamic address stored in a register (R9) and that the pipeline successfully flushes the jump shadow.

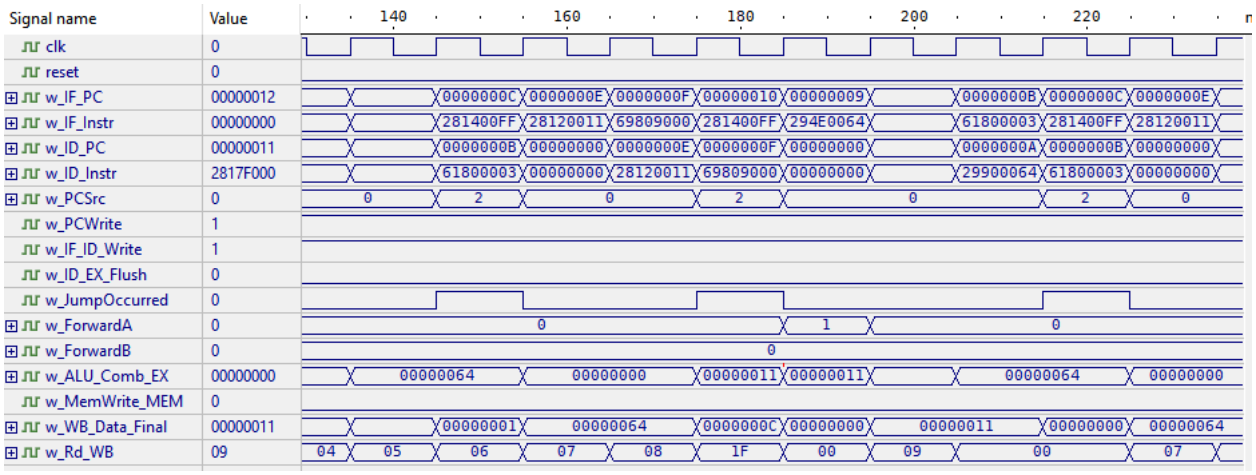
Results and Discussion

In this phase, the instruction at `PC 0x0F` is identified as a JR instruction targeting the address held in **R9**.

- **The Logic:** Unlike a standard jump with a fixed immediate, this instruction requires the hardware to read the value from register **R9** during the Decode stage and feed it into the "Next PC" logic.
- **The Action:** The hardware must detect the jump, assert the flush signal to clear the instruction fetched at `PC 0x10` (the jump shadow), and update the PC to the retrieved address.
- **Control Action:** The `JumpOccurred` signal is asserted, and the `w_PCSrc` MUX is set to select the register-sourced address.

Waveform Verification

The physical evidence of this indirect jump is captured in the final segment of the waveform:



- **Target Identification:** In the Decode stage for **PC 0x0F**, the hardware reads **R9**. The waveform shows `w_WB_Data_Final` was previously loaded with `00000009` for register `09`.
- **The PC Snap:** Observe `w_IF_PC`. Immediately following the execution of the JR at `0x0F`, the PC does not increment to 0x10. Instead, it snaps back to `00000009`.
- **Shadow Flush:** At the moment of redirection, `w_JumpOccurred` pulses high. This triggers a flush of the `w_ID_Instr` bus, which transitions to a NOP state (`00000000`) before receiving the instruction from the new target address.
- **Loop Confirmation:** The PC is then seen following the sequential path from `0x09` to `0x0A` once again, confirming the jump successfully restarted the conditioned code block.

Kernel Snapshot:

The Kernel Trace for **Cycle 15** confirms the exact register-to-PC transfer:

```
[CYCLE 15] > [IF STAGE] Next PC : 00000009
[ID STAGE] Decoding : Op=13 | Rs= 9
Control : JumpOccurred=1 | Flush=1
Values Read: Reg[ 9]=00000009
```

Final Architectural State: Register File Dump:

The following table presents the final values stored in the Register File as of the last simulation cycle, providing empirical proof that all pipeline operations, including memory access, forwarding, predication, and control flow, executed correctly.

Register	Hex Value	Verification Logic
R1	`00000011`	Initial setup or intermediate loop state.
R2	`00000022`	Data correctly loaded from memory address `0x64`.
R3	`00000022`	Verified data dependency handling via forwarding.
R4	`00000044`	Result of R3 + R3 (0x22 + 0x22).
R5	`00000000`	Condition register for Predication (False).
R6	`00000001`	Condition register for Predication (True).
R7	`00000007`	**Proves Nullification:** Remained at its base value because the predicate (R5=0) killed the write.
R8	`00000064`	**Proves Predication:** Successfully updated because the predicate (R6=1) was true.
R9	`00000011`	Final address used for the dynamic `JR` instruction.
R11	`0000000c`	Value moved from R31 during the clean-up phase.
R30	`00000014`	Final PC tracking or loop counter status.
R31 (RA)	`0000000c`	**Proves CALL:** Correctly captured the Return Address (PC+1) from the `CALL` at `0x0B`.

Simulation Conclusion:

The hardware successfully passed all functional tests across five phases:

1. **Forwarding**: Verified via the RAW hazard handling in Phase 1.
2. **Stalling**: Confirmed by the 1-cycle "bubble" injected during the Load-Use hazard.
3. **Predication**: Proved by the suppression of the `WriteEn` signal and the static state of **R7**.
4. **Control Hazards (CALL)**: Verified through the jump to `0x0E` and the storage of the link address in **R31**.
5. **Indirect Jumps (JR)**: Confirmed by the successful loop back to `0x09` using the value in **R9** and the accompanying pipeline flush.

5. Teamwork:

We strived to divide the work amongst the three of us as evenly as possible. Each of us took a certain amount of modules as well as pipeline stages, so that the time and effort put from each student was equal and fair. Discussions on the processor design were held constantly, and oftentimes we asked each other to help review and debug each other's code. Overall, all three of us are more than satisfied by the amount of effort we saw from each other.

6. Disclosure of AI:

AI proved to be somewhat useful in regards to the division of the project, and its provision for a deeper understanding of the project. At times, we've attempted to use it in debugging, though only to find ourselves taking longer than usual as it insists we change things we don't think is necessary to, as we are more than satisfied by our design.

Appendix I:

Listing I.1: *Verilog code Implementation of the Instruction Fetch Stage*

```
module Fetch_Stage (
    input wire clk, reset,
    input wire PCWrite,
    input wire [1:0] PCSrc,
    input wire [31:0] Target_Addr,
    input wire IF_ID_Write, flush,
    output wire [31:0] ID_PC, ID_Instruction
);
    wire [31:0] w_PC, w_NextPC, w_Instruction;

    PC_Register pc_reg (.clk(clk), .reset(reset),.PCWrite(PCWrite),
.next_pc(w_NextPC), .pc(w_PC));

    assign w_NextPC = (PCSrc == 2'b10) ? Target_Addr : (w_PC + 32'd1);

    InstructionMemory inst_mem (.clk(clk), .addr(w_PC),
.instruction(w_Instruction));

    IF_ID_Register if_id_reg (
        .clk(clk), .reset(reset),
        .write_en(IF_ID_Write),
        .flush(flush),
        .PC_in(w_PC), .Instr_in(w_Instruction),
        .PC_out(ID_PC), .Instr_out(ID_Instruction)
    );
endmodule
```

Listing I.2: *Verilog code Implementation of the Instruction Decode Stage*

```
module Decode_Stage (
    input wire clk, reset,
    input wire ID_EX_Flush,
    input wire [31:0] ID_PC, ID_Instruction,
    input wire RegWr_WB, [4:0] Rd_WB, [31:0] WBbus,

    //INPUTS FOR DECODE FORWARDING
    input wire [31:0] ALU_EX, ALU_MEM,
    input wire [4:0]  Rd_EX_in, Rd_MEM_in,
    input wire        RegWr_EX_in, RegWr_MEM_in,
```

```
    output wire [31:0] Reg1_EX, Reg2_EX, Imm_EX, PC_EX, RetAddr_EX,
    output wire [4:0]  Rd_EX, Opcode_EX, Rs_EX, Rt_EX,
    output wire [2:0]  ALUOp_EX,
    output wire RegWrite_EX, MemRead_EX, MemWrite_EX, WB_EX, ALUSrc_EX, nullify_EX,
    output wire [1:0]  WBdata_sel_EX,
    output wire [31:0] NextPC,
    output wire JumpOccurred
);
    wire [31:0] w_RsData, w_RtData, w_RpData, w_ExtImm;
    wire [4:0]  w_Rs, w_Rt, w_Rd, w_Rp, w_Opcode;
    wire        w_ExtOp, w_ALUSrc, w_MemRd, w_MemWr, w_RegWr, w_nullify;
    wire [1:0]  w_WBdata; wire [2:0] w_ALUOp;

    assign w_Opcode = ID_Instruction[31:27];
    assign w_Rp     = ID_Instruction[26:22];
    assign w_Rd     = ID_Instruction[21:17];
    assign w_Rs     = ID_Instruction[16:12];
    assign w_Rt     = ID_Instruction[11:7];
    wire [4:0] w_ReadAddr2;

    // Mux to select between Rt and Rd as the second source register
    assign w_ReadAddr2 = (w_Opcode == 5'd10) ? w_Rd : w_Rt;

    RegisterFile rf (
        .clk(clk), .curr_pc(ID_PC), .Rs1(w_Rs), .Rs2(w_ReadAddr2), .Rp(w_Rp),
        .Rd(Rd_WB), .RegWr(RegWr_WB), .WBbus(WBbus),
        .Bus1(w_RsData), .Bus2(w_RtData), .BusP(w_RpData)
    );

    // --- PREDICATE (Rp) FORWARDING LOGIC ---
    wire [31:0] w_RealRpData;
    assign w_RealRpData = (w_Rp == 5'd0) ? 32'b0 :
                          ((w_Rp == Rd_EX_in)  && RegWr_EX_in)  ? ALU_EX  :
                          ((w_Rp == Rd_MEM_in) && RegWr_MEM_in) ? ALU_MEM :
                          w_RpData;

    MainControlUnit mcu (.Opcode(w_Opcode), .ExtOp(w_ExtOp), .ALUSrc(w_ALUSrc),
.ALUOp(w_ALUOp), .MemRd(w_MemRd), .MemWr(w_MemWr), .WBdata(w_WBdata),
.RegWr(w_RegWr));

    // Pass the CORRECTED RpData to PC Control Unit
    PCControlUnit pcu (
        .PC(ID_PC), .PCPlus1(ID_PC + 32'd1), .Opcode(w_Opcode),
        .RpIdx(w_Rp), .RpData(w_RealRpData), .RsData(w_RsData),
        .Offset(ID_Instruction[21:0]), .NextPC(NextPC), .JumpOccurred(JumpOccurred)
    );
```

```
    // Use CORRECTED RpData for nullifying the current instruction
    assign w_nullify = (w_Rp != 5'd0) && (w_RealRpData == 32'b0);
    assign w_ExtImm  = (w_ExtOp) ? {{20{ID_Instruction[11]}}, ID_Instruction[11:0]}
: {20'b0, ID_Instruction[11:0]};

    wire [4:0] w_Rd_to_EX = (w_Opcode == 5'd12) ? 5'd31 : w_Rd;

    ID_EX_Register id_ex_reg (
        .CLK(clk), .reset(reset),.ID_EX_Flush(ID_EX_Flush),
        .RegWrite_in(ID_EX_Flush ? 1'b0 : (w_nullify ? 1'b0 : w_RegWr)),
        .Mem_read_in(ID_EX_Flush ? 1'b0  : (w_nullify ? 1'b0 : w_MemRd)),
        .Mem_write_in(ID_EX_Flush ? 1'b0 : (w_nullify ? 1'b0 : w_MemWr)),
        .WB_in (ID_EX_Flush ? 1'b0 : (w_nullify ? 1'b0 : w_RegWr)),
        .ALUControl_in(w_ALUOp), .ALUSrc_in(w_ALUSrc),
        .WBdata_in(w_WBdata), .Opcode_in(w_Opcode),
.nullify_in(w_nullify),.enable(1'b1),
        .Rd_in(w_Rd_to_EX), .Rs_in(w_Rs), .Rt_in(w_Rt), .Reg1_in(w_RsData),
.Reg2_in(w_RtData),
        .Imm_in(w_ExtImm), .PC_in(ID_PC), .RetAddr_in(ID_PC + 32'd1),
        .RegWrite_out(RegWrite_EX), .Mem_read_out(MemRead_EX),
.Mem_write_out(MemWrite_EX),
        .WB_out(WB_EX), .ALUControl_out(ALUOp_EX), .ALUSrc_out(ALUSrc_EX),
.WBdata_out(WBdata_sel_EX),
        .Opcode_out(Opcode_EX), .Rd_out(Rd_EX), .Rs_out(Rs_EX), .Rt_out(Rt_EX),
        .Reg1_out(Reg1_EX), .Reg2_out(Reg2_EX), .Imm_out(Imm_EX), .PC_out(PC_EX),
.RetAddr_out(RetAddr_EX), .nullify_out(nullify_EX)
    );
endmodule
```

Listing I.3: *Verilog code Implementation of the Execute Stage*

```
module Execute_Stage (
    input wire clk, reset,
    input wire [31:0]  Reg1, Reg2, Imm, RetAddr_in,
    input wire [4:0]   Rs_in, Rt_in, Rd_in,
    input wire [2:0]   ALUControl,
    input wire [1:0]   WBdata_sel,
    input wire         ALUSrc, RegWrite_in, MemRead_in, MemWrite_in, WB_in,
nullify,


    // Pipeline Data Inputs
    input wire [31:0]  EX_MEM_ALU_Result,
    input wire [31:0]  MEM_WB_Data_Reg,
```

```
    // Hazard/Forwarding Inputs
    input wire [4:0]   EX_MEM_Rd,
    input wire [4:0]   MEM_WB_Rd,
    input wire         EX_MEM_RegWrite,
    input wire         MEM_WB_RegWrite,

    output wire [31:0] ALU_Result_MEM, StoreData_MEM, RetAddr_MEM,
    output wire [31:0] ALU_Result_Comb,
    output wire [4:0]  Rd_MEM,
    output wire        RegWrite_MEM, MemRead_MEM, MemWrite_MEM, WB_MEM,
    output wire [1:0]  WBdata_sel_MEM
);

    wire [31:0] alu_in_B, alu_out;
    wire [1:0]  forwardA, forwardB, forwardD;
    reg  [31:0] fa_mux_out, fb_mux_out, fd_mux_out;

    // --- FORWARDING UNIT INSTANTIATIONS ---
    // Forwarding for Rs
   Forwarding_Unit fwA (
    .ID_EX_Rs(Rs_in),
    .EX_MEM_Rd(EX_MEM_Rd),
    .EX_MEM_RegWrite(EX_MEM_RegWrite),
    .EX_MEM_MemRead(MemRead_MEM),
    .MEM_WB_Rd(MEM_WB_Rd),
    .MEM_WB_RegWrite(MEM_WB_RegWrite),
    .Forward(forwardA)
);


    // Forwarding for Rt
    Forwarding_Unit fwB (
        .ID_EX_Rs(Rt_in),
        .EX_MEM_Rd(EX_MEM_Rd), .EX_MEM_RegWrite(EX_MEM_RegWrite),
.EX_MEM_MemRead(MemRead_MEM),
        .MEM_WB_Rd(MEM_WB_Rd), .MEM_WB_RegWrite(MEM_WB_RegWrite),
        .Forward(forwardB)
    );

    // Forwarding for Rd (Data to store for SW)
    Forwarding_Unit fwD (
        .ID_EX_Rs(Rd_in),
        .EX_MEM_Rd(EX_MEM_Rd), .EX_MEM_RegWrite(EX_MEM_RegWrite),
.EX_MEM_MemRead(MemRead_MEM),
        .MEM_WB_Rd(MEM_WB_Rd), .MEM_WB_RegWrite(MEM_WB_RegWrite),
        .Forward(forwardD)
    );
```

```
always @(*) begin
    case(forwardA)
        2'b01:   fa_mux_out = EX_MEM_ALU_Result; // From EX/MEM
        2'b10:   fa_mux_out = MEM_WB_Data_Reg;   // From MEM/WB
        default: fa_mux_out = Reg1;              // From RegFile
    endcase
end

always @(*) begin
    case(forwardB)
        2'b01:   fb_mux_out = EX_MEM_ALU_Result;
        2'b10:   fb_mux_out = MEM_WB_Data_Reg;
        default: fb_mux_out = Reg2;
    endcase
end

// Use ALUSrc to decide between Rt_forwarded or the Immediate
assign alu_in_B = ALUSrc ? Imm : fb_mux_out;

// Store data (for SW) also needs to be forwarded
always @(*) begin
    case(forwardD)
        2'b01:   fd_mux_out = EX_MEM_ALU_Result;
        2'b10:   fd_mux_out = MEM_WB_Data_Reg;
        default: fd_mux_out = Reg2;
    endcase
end

ALU alu_unit (
    .A(fa_mux_out),
    .B(alu_in_B),
    .ALUControl(ALUControl),
    .ALUResult(alu_out)
);

// Assign the combinational ALU output to the module port
assign ALU_Result_Comb = alu_out;

// Pipeline Register
EX_MEM_Register ex_mem_reg (
    .CLK(clk), .reset(reset),
    .RegWrite_in(RegWrite_in & ~nullify),
    .MemRead_in(MemRead_in & ~nullify),
    .MemWrite_in(MemWrite_in & ~nullify),
    .WB_in(WB_in),
```

```
        .WBdata_in(WBdata_sel),
        .ALU_Result_in(alu_out),
        .StoreData_in(fd_mux_out),
        .RetAddr_in(RetAddr_in),
        .Rd_in(Rd_in),
        .NPC_in(32'b0),
        .RegWrite_out(RegWrite_MEM), .MemRead_out(MemRead_MEM),
.MemWrite_out(MemWrite_MEM), .WB_out(WB_MEM),
        .WBdata_out(WBdata_sel_MEM), .ALU_Result_out(ALU_Result_MEM),
.StoreData_out(StoreData_MEM),
        .RetAddr_out(RetAddr_MEM), .NPC_out(), .Rd_out(Rd_MEM)
    );
endmodule
```

Listing I.4: *Verilog code Implementation of the Memory Stage*

```
module Memory_Stage (
    input wire clk, reset,
    input wire [31:0] ALU_Result, StoreData, RetAddr_in,
    input wire [4:0] Rd_in,
    input wire MemRead, MemWrite, RegWrite_in, [1:0] WBdata_sel,
    output wire [31:0] WB_Data_final,
    output wire [4:0] Rd_WB,
    output wire RegWrite_WB
);
    wire [31:0] w_MemData, w_MuxOut;
    DataMemory dmem (.clk(clk), .MemRead(MemRead), .MemWrite(MemWrite),
.Address(ALU_Result), .WriteData(StoreData), .ReadData(w_MemData));
    assign w_MuxOut = (WBdata_sel == 2'b01) ? w_MemData : (WBdata_sel == 2'b10) ?
RetAddr_in : ALU_Result;
    MEM_WB_Register mem_wb_reg (.clk(clk), .reset(reset),
.RegWrite_in(RegWrite_in), .Data_in(w_MuxOut), .Rd_in(Rd_in),
.RegWrite_out(RegWrite_WB), .Data_out(WB_Data_final), .Rd_out(Rd_WB));
endmodule
```

Listing I.5: *Verilog code Implementation of the Register File*

```
module RegisterFile(
    input clk,
    input [4:0] Rs1, Rs2, Rp, Rd,
    input [31:0] curr_pc,
    input RegWr,
```

```
    input [31:0] WBbus,
    output [31:0] Bus1, Bus2, BusP
);
    reg [31:0] registers [0:31];
    integer i;

    initial begin
        for (i = 0; i < 32; i = i + 1) registers[i] = (i < 17) ? i : 32'h0;
        registers[0] = 0;
    end

    wire bypass_Rs1 = (Rs1 == Rd) && RegWr && (Rs1 != 0) && (Rs1 != 5'd30);
    wire bypass_Rs2 = (Rs2 == Rd) && RegWr && (Rs2 != 0);
    wire bypass_Rp  = (Rp  == Rd) && RegWr && (Rp  != 0);

    assign Bus1 = (Rs1 == 5'd0)  ? 32'b0 :
                  (Rs1 == 5'd30) ? curr_pc :
                  (bypass_Rs1)   ? WBbus :
                  registers[Rs1];

    assign Bus2 = (Rs2 == 5'd0)  ? 32'b0 :
                  (bypass_Rs2)   ? WBbus :
                  registers[Rs2];

    assign BusP = (Rp == 5'd0)   ? 32'b0 :
                  (bypass_Rp)    ? WBbus :
                  registers[Rp];

    always @(posedge clk) begin
        if (RegWr && (Rd != 5'd0) && (Rd != 5'd30)) begin
            registers[Rd] <= WBbus;
        end
        registers[30] <= curr_pc;
    end
endmodule
```

Listing I.6: *Verilog code Implementation of the PC Register*

```
module PC_Register (input clk, reset, PCWrite, [31:0] next_pc, output reg [31:0]
pc);
    always @(posedge clk or posedge reset)begin
    if (reset) pc <= 0;
    // only enable update if not stalling
    else if (PCWrite)
    pc <= next_pc;
```

```
    end
endmodule
```

Listing I.7: *Verilog code Implementation of the Instruction Memory*

```
module InstructionMemory (input clk, [31:0] addr, output [31:0] instruction);
    reg [31:0] instMemory [0:1023];
       initial begin
    for (int i = 0; i < 1024; i = i+1) instMemory[i] = 0;
    // --- PHASE 1: Forwarding & Priority (I-Type & R-Type) ---
    // Format I: Op(5), Rp(5), Rd(5), Rs(5), Imm(12)
    instMemory[0] = {5'd5, 5'd0, 5'd1, 5'd0, 12'd10};  // ADDI R1, R0, 10
    instMemory[1] = {5'd5, 5'd0, 5'd1, 5'd1, 12'd5};   // ADDI R1, R1, 5  (EX-EX
Forward: R1=15)
    instMemory[2] = {5'd5, 5'd0, 5'd1, 5'd1, 12'd2};   // ADDI R1, R1, 2  (Priority
Test: R1=17)
    // Format R: Op(5), Rp(5), Rd(5), Rs(5), Rt(5), Unused(7)
    instMemory[3] = {5'd0, 5'd0, 5'd2, 5'd1, 5'd1, 7'd0}; // ADD R2, R1, R1 (R2
should be 34)
    // --- PHASE 2: Load-Use & Memory (I-Type) ---
    instMemory[4] = {5'd10, 5'd0, 5'd2, 5'd0, 12'd100}; // SW R2, 100(R0) (Store
34)
    instMemory[5] = {5'd9,  5'd0, 5'd3, 5'd0, 12'd100}; // LW R3, 100(R0) (R3 = 34)
    instMemory[6] = {5'd0,  5'd0, 5'd4, 5'd3, 5'd3, 7'd0}; // ADD R4, R3, R3 (STALL
check: R4=68)
    // --- PHASE 3: Predication (Nullify) ---
    instMemory[7] = {5'd5, 5'd0, 5'd5, 5'd0, 12'd0};    // R5 = 0 (Predicate False)
    instMemory[8] = {5'd5, 5'd0, 5'd6, 5'd0, 12'd1};    // R6 = 1 (Predicate True)
    // KILLED: Rp=R5 (0), so ADDI R7 shouldn't happen
    instMemory[9] = {5'd5, 5'd5, 5'd7, 5'd0, 12'd100};
    // EXECUTED: Rp=R6 (1), so ADDI R8 should happen
    instMemory[10]= {5'd5, 5'd6, 5'd8, 5'd0, 12'd100};  // R8 = 100
    // --- PHASE 4: J-Type Call & Flush ---
    // Format J: Op(5), Rp(5), Offset(22)
    // CALL +3 if R6 != 0. Jumps to PC 14, writes PC+1 (12) to R31.
    instMemory[11] = {5'd12, 5'd6, 22'd3};
        instMemory[12] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED
    instMemory[13] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED
    // --- PHASE 5: Indirect Jump (I-Type style) ---
    // Some architectures use I-Type for JRs to keep the Rs field.
    instMemory[14] = {5'd5, 5'd0, 5'd9, 5'd0, 12'd17};  // R9 = 17
    instMemory[15] = {5'd13, 5'd6, 5'd0, 5'd9, 12'd0};  // J Rs (Jump to R9=17)
    instMemory[16] = {5'd5, 5'd0, 5'd10, 5'd0, 12'd255}; // Should be FLUSHED
    instMemory[17] = {5'd5, 5'd0, 5'd11, 5'd31, 12'd0}; // Final Check: R11 = R31
(12)
```

```
end
   assign instruction = instMemory[addr];
endmodule
```

Listing I.8: *Verilog code Implementation of the Main Control Unit*

```
module MainControlUnit (input [4:0] Opcode, output reg ExtOp, ALUSrc, MemRd, MemWr,
RegWr, output reg [1:0] WBdata, output reg [2:0] ALUOp);
    always @(*) begin
        {ExtOp, ALUSrc, MemRd, MemWr, RegWr} = 5'b0;
        WBdata = 2'b00; ALUOp = 3'b000;
        case(Opcode)
            5'd0:  begin ALUOp=3'b000; RegWr=1; end // ADD
            5'd1:  begin ALUOp=3'b001; RegWr=1; end // SUB
            5'd2:  begin ALUOp=3'b011; RegWr=1; end // OR
            5'd3:  begin ALUOp=3'b100; RegWr=1; end // NOR
            5'd4:  begin ALUOp=3'b010; RegWr=1; end // AND
            5'd5:  begin ALUOp=3'b000; ALUSrc=1; RegWr=1; ExtOp=1; end // ADDI
            5'd6:  begin ALUOp=3'b011; ALUSrc=1; RegWr=1; end // ORI
            5'd7:  begin ALUOp=3'b100; ALUSrc=1; RegWr=1; end // NORI
            5'd8:  begin ALUOp=3'b010; ALUSrc=1; RegWr=1; end // ANDI
            5'd9:  begin ALUOp=3'b000; ALUSrc=1; MemRd=1; WBdata=2'b01; RegWr=1;
ExtOp=1; end // LW
            5'd10: begin ALUOp=3'b000; ALUSrc=1; MemWr=1; ExtOp=1; end // SW
            5'd12: begin RegWr=1; WBdata=2'b10; end // CALL
            default: ;
        endcase
    end
endmodule
```

Listing I.9: *Verilog code Implementation of the PC Control Unit*

```
module PCControlUnit(
    input [31:0] PC, PCPlus1, RpData, RsData,
    input [4:0] Opcode, RpIdx,
    input [21:0] Offset,
    output reg [31:0] NextPC,
    output reg JumpOccurred
);
    wire [31:0] SignExtOffset = {{10{Offset[21]}}, Offset};
    wire predicate_met = (RpIdx == 5'd0) || (RpData != 32'd0);

    always @(*) begin
```

```
        NextPC = PCPlus1;
        JumpOccurred = 0;
        if (predicate_met) begin
            case (Opcode)
                5'd11: begin NextPC = PC + SignExtOffset; JumpOccurred = 1; end
                5'd12: begin NextPC = PC + SignExtOffset; JumpOccurred = 1; end
                5'd13: begin NextPC = RsData;             JumpOccurred = 1; end
                default: ;
            endcase
        end
    end
endmodule
```

Listing I.10: *Verilog code Implementation of the Hazard Detection Unit*

```
module HazardDetectionUnit(
    input wire ID_EX_MemRead,
    input wire [4:0] ID_EX_Rd,
    input wire [4:0] IF_ID_Rs,
    input wire [4:0] IF_ID_Rt,
    input wire [4:0] IF_ID_Rp,
    input wire [4:0] IF_ID_Rd_Input,
    input wire [4:0] IF_ID_Opcode,
    output reg PCWrite,
    output reg IF_ID_Write,
    output reg ID_EX_Flush
    );

    always @(*) begin
        PCWrite = 1; IF_ID_Write = 1; ID_EX_Flush = 0;

        if (ID_EX_MemRead && (ID_EX_Rd != 0)) begin
            if ((ID_EX_Rd == IF_ID_Rs) || (ID_EX_Rd == IF_ID_Rt) || (ID_EX_Rd ==
IF_ID_Rp)) begin
                PCWrite = 0; IF_ID_Write = 0; ID_EX_Flush = 1;
            end
            else if ((IF_ID_Opcode == 5'd10) && (ID_EX_Rd == IF_ID_Rd_Input)) begin
                PCWrite = 0; IF_ID_Write = 0; ID_EX_Flush = 1;
            end
        end
    end
endmodule
```

Listing I.11: *Verilog code Implementation of the Forwarding Unit*

```
module Forwarding_Unit (
    input [4:0] ID_EX_Rs,
    input [4:0] EX_MEM_Rd,
    input EX_MEM_RegWrite,
    input EX_MEM_MemRead,
    input [4:0] MEM_WB_Rd,
    input MEM_WB_RegWrite,
    output reg [1:0] Forward
);
    always @(*) begin
    Forward = 2'b00;

    if (EX_MEM_RegWrite && !EX_MEM_MemRead && (EX_MEM_Rd != 0) && (EX_MEM_Rd ==
ID_EX_Rs))
        Forward = 2'b01;
    else if (MEM_WB_RegWrite && (MEM_WB_Rd != 0) && (MEM_WB_Rd == ID_EX_Rs))
        Forward = 2'b10;
    end
endmodule
```

Listing I.12: *Verilog code Implementation of the ALU*

```
module ALU (input [31:0] A, B, input [2:0] ALUControl, output reg [31:0]
ALUResult);
    always @(*) begin
        case (ALUControl)
            3'b000: ALUResult = A + B;
            3'b001: ALUResult = A - B;
            3'b010: ALUResult = A & B;
            3'b011: ALUResult = A | B;
            3'b100: ALUResult = ~(A | B);
            default: ALUResult = 32'b0;
        endcase
    end
endmodule
```

Listing I.13: *Verilog code Implementation of the Data Memory*

```
module DataMemory (input clk, MemRead, MemWrite, [31:0] Address, WriteData, output
```

```
reg [31:0] ReadData);
    reg [31:0] memory [0:1023];

    initial  begin
        for (integer i = 0; i < 1024; i = i + 1)
            memory[i] = 32'd0;
        memory[10]= 32'd35;
        memory[11]= 32'd8;
        memory[20]= 32'd9;
    end

    always @(*) begin
        #1
        if (MemRead) ReadData <= memory[Address[9:0]];
    end

    always @(posedge clk) begin
        #1
        if (MemWrite) begin
        memory[Address[9:0]] <= WriteData;
    end
    end
endmodule
```

Listing I.14: *Verilog code Implementation of the IF/ID*

```
module IF_ID_Register (input clk, reset, write_en, flush, [31:0] PC_in, Instr_in,
output reg [31:0] PC_out, Instr_out);
    always @(posedge clk or posedge reset) if (reset || flush) {PC_out, Instr_out}
<= 0; else if (write_en) {PC_out, Instr_out} <= {PC_in, Instr_in};
endmodule
```

Listing I.15: *Verilog code Implementation of the ID/EX*

```
module ID_EX_Register (
    input wire CLK, reset, ID_EX_Flush, RegWrite_in, Mem_read_in, Mem_write_in,
WB_in, ALUSrc_in, nullify_in,
    input wire enable,
    input wire [1:0] WBdata_in, input wire [2:0] ALUControl_in, input wire [4:0]
Opcode_in, Rd_in, Rs_in, Rt_in,
    input wire [31:0] Reg1_in, Reg2_in, Imm_in, PC_in, RetAddr_in,
    output reg RegWrite_out, Mem_read_out, Mem_write_out, WB_out, ALUSrc_out,
nullify_out,
```

```
    output reg [1:0] WBdata_out, output reg [2:0] ALUControl_out, output reg [4:0]
Opcode_out, Rd_out, Rs_out, Rt_out,
    output reg [31:0] Reg1_out, Reg2_out, Imm_out, PC_out, RetAddr_out
);
always @(posedge CLK or posedge reset) begin
    if (reset || ID_EX_Flush) begin
        RegWrite_out <= 0; Mem_read_out <= 0; Mem_write_out <= 0; WB_out <= 0;
ALUSrc_out <= 0; nullify_out <= 0;
        WBdata_out <= 0; ALUControl_out <= 0; Opcode_out <= 0; Rd_out <= 0; Rs_out
<= 0; Rt_out <= 0;
        Reg1_out <= 0; Reg2_out <= 0; Imm_out  <= 0; PC_out   <= 0; RetAddr_out <=
0;
    end
    else if (enable) begin
        RegWrite_out <= RegWrite_in; Mem_read_out <= Mem_read_in; Mem_write_out <=
Mem_write_in; WB_out <= WB_in; ALUSrc_out <= ALUSrc_in; nullify_out <= nullify_in;
        WBdata_out <= WBdata_in; ALUControl_out <= ALUControl_in; Opcode_out <=
Opcode_in; Rd_out <= Rd_in; Rs_out <= Rs_in; Rt_out <= Rt_in;
        Reg1_out <= Reg1_in; Reg2_out <= Reg2_in; Imm_out  <= Imm_in; PC_out   <=
PC_in; RetAddr_out <= RetAddr_in;
    end
end
endmodule
```

Listing I.16: *Verilog code Implementation of the EX/MEM*

```
module EX_MEM_Register (
    input wire CLK, reset, RegWrite_in, MemRead_in, MemWrite_in, WB_in, [1:0]
WBdata_in,
    input wire [31:0] ALU_Result_in, StoreData_in, RetAddr_in, NPC_in, input wire
[4:0] Rd_in,
    output reg RegWrite_out, MemRead_out, MemWrite_out, WB_out,
    output reg [1:0] WBdata_out,
    output reg [31:0] ALU_Result_out, StoreData_out, RetAddr_out, NPC_out,
    output reg [4:0] Rd_out
);
    always @(posedge CLK or posedge reset) begin
        if (reset) begin
            {RegWrite_out, MemRead_out, MemWrite_out, WB_out} <= 0;
            {WBdata_out, ALU_Result_out, StoreData_out, RetAddr_out, NPC_out,
Rd_out} <= 0;
        end else begin
            RegWrite_out <= RegWrite_in; MemRead_out <= MemRead_in; MemWrite_out <=
MemWrite_in;
            WB_out <= WB_in; WBdata_out <= WBdata_in; ALU_Result_out <=
```

```
ALU_Result_in;
            StoreData_out <= StoreData_in; RetAddr_out <= RetAddr_in; NPC_out <=
NPC_in; Rd_out <= Rd_in;
        end
    end
endmodule
```

Listing I.17: *Verilog code Implementation of the MEM/WB*

```
module MEM_WB_Register (
    input wire clk, reset, RegWrite_in, [31:0] Data_in, [4:0] Rd_in,
    output reg RegWrite_out,
    output reg [31:0] Data_out,
    output reg [4:0] Rd_out
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            RegWrite_out <= 0; Data_out <= 0; Rd_out <= 0;
        end else begin
            RegWrite_out <= RegWrite_in; Data_out <= Data_in; Rd_out <= Rd_in;
        end
    end
endmodule
```

Appendix II:

Listing II.1: *Full Log of Test Cases for the Pipelined Processor*

```
# KERNEL:
================================================================================
# KERNEL:                      PIPELINED PROCESSOR DETAILED TRACE
# KERNEL:
================================================================================
# KERNEL:
# KERNEL: [CYCLE  0] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000001
# KERNEL:     Next PC    : 00000002
# KERNEL:     Instr Raw  : 28021005
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000000
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 1 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
```

```
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE  1] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000002
# KERNEL:     Next PC    : 00000003
# KERNEL:     Instr Raw  : 28021002
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000001
# KERNEL:     Decoding   : Op= 5 | Rs= 1 | Rt= 0 | Rd= 1 | Rp= 0
# KERNEL:     Values Read: Reg[ 1]=00000001 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=0000000a | Result=0000000a
# KERNEL:     Output     : Dest Reg=R 1 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE  2] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000003
# KERNEL:     Next PC    : 00000004
# KERNEL:     Instr Raw  : 00041080
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000002
# KERNEL:     Decoding   : Op= 5 | Rs= 1 | Rt= 0 | Rd= 1 | Rp= 0
# KERNEL:     Values Read: Reg[ 1]=00000001 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=01 (Val=0000000a) | FwdB=00 (Val=00000000) |
FwdD=01 (Val=0000000a)
# KERNEL:     ALU Calc   : A=0000000a | B=00000005 | Result=0000000f
# KERNEL:     Output     : Dest Reg=R 1 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=0000000a | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
```

```
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  3] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000004
# KERNEL:     Next PC    : 00000005
# KERNEL:     Instr Raw  : 50040064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000003
# KERNEL:     Decoding   : Op= 0 | Rs= 1 | Rt= 1 | Rd= 2 | Rp= 0
# KERNEL:     Values Read: Reg[ 1]=0000000a | Reg[ 1]=0000000a | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=01 (Val=0000000f) | FwdB=00 (Val=00000000) |
FwdD=01 (Val=0000000f)
# KERNEL:     ALU Calc   : A=0000000f | B=00000002 | Result=00000011
# KERNEL:     Output     : Dest Reg=R 1 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=0000000f | WriteData=0000000a
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 0000000a to Register R 1
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  4] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000005
# KERNEL:     Next PC    : 00000006
# KERNEL:     Instr Raw  : 48060064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000004
# KERNEL:     Decoding   : Op=10 | Rs= 0 | Rt= 0 | Rd= 2 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 2]=00000002 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=01 (Val=00000011) | FwdB=01 (Val=00000011) |
FwdD=00 (Val=0000000a)
# KERNEL:     ALU Calc   : A=00000011 | B=00000011 | Result=00000022
# KERNEL:     Output     : Dest Reg=R 2 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000011 | WriteData=0000000f
```

```
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 0000000f to Register R 1
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  5] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000006
# KERNEL:     Next PC    : 00000007
# KERNEL:     Instr Raw  : 00083180
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000005
# KERNEL:     Decoding   : Op= 9 | Rs= 0 | Rt= 0 | Rd= 3 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000002) |
FwdD=01 (Val=00000022)
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 2 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000022 | WriteData=0000000a
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000011 to Register R 1
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  6] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000007
# KERNEL:     Next PC    : 00000008
# KERNEL:     Instr Raw  : 280a0000
# KERNEL:     Stall/Flush: PCWrite=0 | IF/ID_Write=0 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000006
# KERNEL:     Decoding   : Op= 0 | Rs= 3 | Rt= 3 | Rd= 4 | Rp= 0
# KERNEL:     Values Read: Reg[ 3]=00000003 | Reg[ 3]=00000003 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=1
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 3 | WriteEn=1 | MemRead=1
# KERNEL:   [MEM STAGE]
```

```
# KERNEL:     Access     : Addr=00000064 | WriteData=00000022
# KERNEL:     Control    : MemRead=0 | MemWrite=1
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000022 to Register R 2
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  7] ----------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000007
# KERNEL:     Next PC    : 00000008
# KERNEL:     Instr Raw  : 280a0000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000006
# KERNEL:     Decoding   : Op= 0 | Rs= 3 | Rt= 3 | Rd= 4 | Rp= 0
# KERNEL:     Values Read: Reg[ 3]=00000003 | Reg[ 3]=00000003 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000064 | WriteData=00000000
# KERNEL:     Control    : MemRead=1 | MemWrite=0
# KERNEL:     Read Result: xxxxxxxx (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000064 to Register R 2
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE  8] ----------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000008
# KERNEL:     Next PC    : 00000009
# KERNEL:     Instr Raw  : 280c0001
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000007
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 5 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=10 (Val=00000022) | FwdB=10 (Val=00000022) |
FwdD=00 (Val=00000003)
# KERNEL:     ALU Calc   : A=00000022 | B=00000022 | Result=00000044
# KERNEL:     Output     : Dest Reg=R 4 | WriteEn=1 | MemRead=0
```

```
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000022 to Register R 3
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE  9] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000009
# KERNEL:     Next PC    : 0000000a
# KERNEL:     Instr Raw  : 294e0064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000008
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 6 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 5 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000044 | WriteData=00000003
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE 10] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000a
# KERNEL:     Next PC    : 0000000b
# KERNEL:     Instr Raw  : 29900064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000009
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 7 | Rp= 5
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 5]=00000005
# KERNEL:     Control    : Predicate_Kill=1 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000001 | Result=00000001
```

```
# KERNEL:     Output     : Dest Reg=R 6 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000044 to Register R 4
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 11] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000b
# KERNEL:     Next PC    : 0000000c
# KERNEL:     Instr Raw  : 61800003
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 0000000a
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 8 | Rp= 6
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 6]=00000006
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 7 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000001 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 5
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 12] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000c
# KERNEL:     Next PC    : 0000000e
# KERNEL:     Instr Raw  : 281400ff
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=1
# KERNEL:   [ID STAGE] PC: 0000000b
# KERNEL:     Decoding   : Op=12 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 6
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 6]=00000001
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=1 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
```

```
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 8 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000064 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000001 to Register R 6
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 13] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000e
# KERNEL:     Next PC    : 0000000f
# KERNEL:     Instr Raw  : 28120011
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000000
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R31 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000064 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000064 to Register R 7
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE 14] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000f
# KERNEL:     Next PC    : 00000010
# KERNEL:     Instr Raw  : 69809000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 0000000e
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 9 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
```

```
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000064 to Register R 8
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 15] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000010
# KERNEL:     Next PC    : 00000009
# KERNEL:     Instr Raw  : 281400ff
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=1
# KERNEL:   [ID STAGE] PC: 0000000f
# KERNEL:     Decoding   : Op=13 | Rs= 9 | Rt= 0 | Rd= 0 | Rp= 6
# KERNEL:     Values Read: Reg[ 9]=00000009 | Reg[ 0]=00000000 | Reg[ 6]=00000001
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=1 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000011 | Result=00000011
# KERNEL:     Output     : Dest Reg=R 9 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 0000000c to Register R31
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 16] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000009
# KERNEL:     Next PC    : 0000000a
# KERNEL:     Instr Raw  : 294e0064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000000
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
```

```
# KERNEL:     Forwarding : FwdA=01 (Val=00000011) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000011 | B=00000000 | Result=00000011
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000011 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 17] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000a
# KERNEL:     Next PC    : 0000000b
# KERNEL:     Instr Raw  : 29900064
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000009
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 7 | Rp= 5
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 5]=00000000
# KERNEL:     Control    : Predicate_Kill=1 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000011 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000011 to Register R 9
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 18] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000b
# KERNEL:     Next PC    : 0000000c
# KERNEL:     Instr Raw  : 61800003
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 0000000a
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 8 | Rp= 6
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 6]=00000001
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
```

```
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 7 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000011 to Register R 0
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE 19] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000c
# KERNEL:     Next PC    : 0000000e
# KERNEL:     Instr Raw  : 281400ff
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=1
# KERNEL:   [ID STAGE] PC: 0000000b
# KERNEL:     Decoding   : Op=12 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 6
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 6]=00000001
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=1 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000064 | Result=00000064
# KERNEL:     Output     : Dest Reg=R 8 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000064 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 20] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000e
# KERNEL:     Next PC    : 0000000f
# KERNEL:     Instr Raw  : 28120011
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000000
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
```

```
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R31 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000064 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000064 to Register R 7
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE 21] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 0000000f
# KERNEL:     Next PC    : 00000010
# KERNEL:     Instr Raw  : 69809000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 0000000e
# KERNEL:     Decoding   : Op= 5 | Rs= 0 | Rt= 0 | Rd= 9 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000064 to Register R 8
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 22] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000010
# KERNEL:     Next PC    : 00000011
# KERNEL:     Instr Raw  : 281400ff
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=1
# KERNEL:   [ID STAGE] PC: 0000000f
# KERNEL:     Decoding   : Op=13 | Rs= 9 | Rt= 0 | Rd= 0 | Rp= 6
# KERNEL:     Values Read: Reg[ 9]=00000011 | Reg[ 0]=00000000 | Reg[ 6]=00000001
```

```
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=1 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000011 | Result=00000011
# KERNEL:     Output     : Dest Reg=R 9 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 0000000c to Register R31
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 23] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000011
# KERNEL:     Next PC    : 00000012
# KERNEL:     Instr Raw  : 2817f000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000000
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=01 (Val=00000011) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000011 | B=00000000 | Result=00000011
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=0 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000011 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 24] ----------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000012
# KERNEL:     Next PC    : 00000013
# KERNEL:     Instr Raw  : 00000000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000011
# KERNEL:     Decoding   : Op= 5 | Rs=31 | Rt= 0 | Rd=11 | Rp= 0
```

```
# KERNEL:     Values Read: Reg[31]=0000000c | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000011 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000011 to Register R 9
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 25] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000013
# KERNEL:     Next PC    : 00000014
# KERNEL:     Instr Raw  : 00000000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000012
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=1
# KERNEL:     Forwarding : FwdA=00 (Val=0000000c) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=0000000c | B=00000000 | Result=0000000c
# KERNEL:     Output     : Dest Reg=R11 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000011 to Register R 0
# KERNEL:     Enable     : 0
# KERNEL:
# KERNEL: [CYCLE 26] ------------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000014
# KERNEL:     Next PC    : 00000015
# KERNEL:     Instr Raw  : 00000000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000013
```

```
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=0000000c | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 27] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000015
# KERNEL:     Next PC    : 00000016
# KERNEL:     Instr Raw  : 00000000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
# KERNEL:   [ID STAGE] PC: 00000014
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 0000000c to Register R11
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: [CYCLE 28] ---------------------------------------------------------
# KERNEL:   [IF STAGE]
# KERNEL:     PC Current : 00000016
# KERNEL:     Next PC    : 00000017
# KERNEL:     Instr Raw  : 00000000
# KERNEL:     Stall/Flush: PCWrite=1 | IF/ID_Write=1 | Flush=0
```

```
# KERNEL:   [ID STAGE] PC: 00000015
# KERNEL:     Decoding   : Op= 0 | Rs= 0 | Rt= 0 | Rd= 0 | Rp= 0
# KERNEL:     Values Read: Reg[ 0]=00000000 | Reg[ 0]=00000000 | Reg[ 0]=00000000
# KERNEL:     Control    : Predicate_Kill=0 | JumpOccurred=0 | ID_EX_Flush=0
# KERNEL:   [EX STAGE]
# KERNEL:     ALU Ops    : OpCode=000 | ALUSrc=0
# KERNEL:     Forwarding : FwdA=00 (Val=00000000) | FwdB=00 (Val=00000000) |
FwdD=00 (Val=00000000)
# KERNEL:     ALU Calc   : A=00000000 | B=00000000 | Result=00000000
# KERNEL:     Output     : Dest Reg=R 0 | WriteEn=1 | MemRead=0
# KERNEL:   [MEM STAGE]
# KERNEL:     Access     : Addr=00000000 | WriteData=00000000
# KERNEL:     Control    : MemRead=0 | MemWrite=0
# KERNEL:     Read Result: 00000022 (from memory)
# KERNEL:   [WB STAGE]
# KERNEL:     WriteBack  : Writing Data 00000000 to Register R 0
# KERNEL:     Enable     : 1
# KERNEL:
# KERNEL: ================ FINAL REGISTER FILE STATUS ================
# KERNEL: Reg |  Value     |  Reg |  Value     |  Reg |  Value
# KERNEL: ----------------------------------------------------------
# KERNEL: R 0 : 00000000 | R 1 : 00000011 | R 2 : 00000022 | R 3 : 00000022
# KERNEL: R 4 : 00000044 | R 5 : 00000000 | R 6 : 00000001 | R 7 : 00000007
# KERNEL: R 8 : 00000064 | R 9 : 00000011 | R10 : 0000000a | R11 : 0000000c
# KERNEL: R12 : 0000000c | R13 : 0000000d | R14 : 0000000e | R15 : 0000000f
# KERNEL: R16 : 00000010 | R17 : 00000000 | R18 : 00000000 | R19 : 00000000
# KERNEL: R20 : 00000000 | R21 : 00000000 | R22 : 00000000 | R23 : 00000000
# KERNEL: R24 : 00000000 | R25 : 00000000 | R26 : 00000000 | R27 : 00000000
# KERNEL: R28 : 00000000 | R29 : 00000000 | R30 : 00000014 | R31 : 0000000c
# KERNEL: ==========================================================
# KERNEL:
# KERNEL:
# KERNEL: ================ FINAL DATA MEMORY STATUS ================
# KERNEL: Address | Value      | Address | Value
# KERNEL: ----------------------------------------------------------
# KERNEL: Addr 10 : 00000023 (        35)
# KERNEL: Addr 20 : 00000009 (         9)
# KERNEL: ==========================================================
# KERNEL:
```