VERILOG explanation - MIPS Pipeline - Processor Workflow Explanation

Yashvardhan Singh

1] Introduction

1.1 Background Information

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a classic example of simplicity meeting efficiency in processor design. Built on the RISC (Reduced Instruction Set Computing) philosophy, MIPS was developed to counter the growing complexity of CISC (Complex Instruction Set Computing) architectures like x86 in the 1980s. By focusing on a streamlined, predictable execution model, MIPS prioritized performance, scalability, and ease of implementation – qualities that continue to make it relevant in modern computing.
At its core, MIPS follows a five-stage pipeline (Fetch, Decode, Execute, Memory Access, and Write-back), ensuring one instruction per cycle execution. Unlike other architectures that rely on intricate hardware mechanisms for handling data hazards, early MIPS processors placed this responsibility on the compiler, reducing hardware overhead and improving efficiency. Even though modern MIPS designs now incorporate interlocked pipeline stages, the architecture remains true to its original philosophy of simplicity and performance. Beyond its historical significance, MIPS is widely used today in embedded systems, network-ing devices, and academic environments. Its fixed 32-bit instruction length, load/store architecture, and efficient pipelining make it an excellent platform for learning ISA design, processor architecture, and hardware implementation. Through this project, we aim to deepen
our understanding of Verilog HDL, FPGA-based design, and reconfigurable computing by implementing a 5-stage pipelined MIPS processor, bridging the gap between theoretical concepts and real-world applications.

1.2 Objectives and Scope

• Design a 5-stage pipelined MIPS processor in Verilog
• Implement core instruction set (arithmetic, logical, memory, control)
• Validate through simulation using Xilinx Vivado
• Analyze performance characteristics under various test cases

1.3 Verilog Design Rationale

Verilog was chosen for its:
• Hardware description capabilities
• Compatibility with Xilinx Vivado toolchain
• Support for hierarchical design
• Industry-standard status in digital design

1.4 Problem Motivation

Modern computing demands efficient processor architectures. Understanding Instruction Set Architectures (ISAs) is fundamental to computer architecture. Its structured design allows enthusiasts to explore processor implementation without excessive hardware complexity.
Our team, driven by a deep interest in computer architecture, aims to explore ISAs and processor design by implementing a 32-bit, 5-stage pipelined MIPS processor. MIPS serves as an ideal starting point due to its simplicity and structured design, making it an excellent learning platform for ISA exploration.
Beyond processor architecture, this project also enhances our understanding of Verilog HDL and FPGA-based design. By simulating our implementation in Xilinx Vivado, we gain hands-on experience with hardware description languages.

---

2 System Architecture

The processor implements classic MIPS pipeline stages:

1. Instruction Fetch (IF)
   The processor retrieves the instruction from memory at the address specified by the Program
   Counter (PC). The PC is then incremented to point to the next instruction. This stage involves
   interaction with the instruction memory and the fetching of the instruction bits.
2. Instruction Decode (ID)
   The fetched instruction is decoded to determine the operation and operand registers. The control
   unit generates control signals based on the opcode. Register values are read from the register file
   if needed. If the instruction involves immediate values, they are sign-extended or zero-extended as
   required.
3. Execute (EX)
   The Arithmetic Logic Unit (ALU) performs the necessary computation (e.g., arithmetic

operations,
logical operations, shifts, or branch target calculations). The ALU takes operands from the register
file or the sign-extended immediate value. If it's a branch instruction, the branch target address is
computed, and conditions are checked.

4. Memory Access (MEM)
For load instructions, data is fetched from memory. For store instructions, data is written to
memory at the calculated address. Other instructions (like arithmetic ones) bypass this stage
without accessing memory.

5. Write Back (WB)
The final result (from the ALU or memory) is written back to the destination register in the register
file. This stage ensures that computed values are stored for future instructions to use.
Key Modules in a MIPS Processor
Instruction Memory (imem)
• Stores program instructions in a 32-bit wide memory array.
• Instructions are fetched during the Instruction Fetch (IF) stage using the Program Counter (PC).
Data Memory (dmem)
• Stores data for load (‘lw‘) and store (‘sw‘) instructions.
• Supports word-aligned 32-bit memory access during the Memory Access (MEM) stage.
Register File (regfile)
• Contains 32 general-purpose 32-bit registers, with register ‘0‘ hardwired to zero.
• Supports dual read ports and one write port for efficient operand access and result storage.
Arithmetic Logic Unit (ALU)
• Performs arithmetic and logical operations such as addition, subtraction, AND, OR, and compar-
isons.
• Operates on two operands provided by the register file or an immediate value.
ALU Control Unit
• Decodes control signals to determine the operation performed by the ALU.
• For R-type instructions, it uses the ‘funct‘ field; for I-type instructions, it directly uses the ‘ALUop‘.
Sign Extension Unit
• Extends 16-bit immediate values to 32 bits for arithmetic or logical operations.

• Supports both sign extension (for arithmetic) and zero extension (for logical operations).
Multiplexers (MUXes)
• Used throughout the pipeline to select between multiple inputs:

6. ALUSrc MUX: Selects between a register operand or an immediate value for ALU input.
7. MemToReg MUX: Selects between memory output or ALU result for write-back to registers.
8. PC MUX: Chooses between sequential PC (‘PC+4‘) or branch/jump target address.

2.3 Design Choices

Decision Rationale
5-stage pipeline: Balance between complexity & performance
Verilog implementation: FPGA compatibility & design flexibility
No Hazard Handling: Design simplicity and Focus on pipeline fundamentals
NOP(No Operation) based: Testbench Introduce delays and avoid hazard based testcases

---

3 Results

3.1 Simulated Waveforms

Figure 2: Non-Pipelined: (Top: Waveform) (Bottom Left: Power) (Bottom Right: Utilisation)
Figure 3: Pipelined processor - GTKWave output

3.2 Final Register State

The RTL Verilog design was compiled using opensource tool Icarus Verilog (iverilog) with -g2012 on Ubuntu 24.04 LTS, running on our personal laptops. This setup allowed us to extensively prototype within our hostel room. The compiled output was executed using vvp, and waveform analysis was performed using GTKWave with a generated .vcd file.

```
$s0 = 5 (expected 5)
$s1 = 10 (expected 10)
$t0 = 15 (expected 15)
$s2 = 00000005 (expected 5)
$s3 = 000000fa (expected fa)
$s4 = 0 (expected 0)
$t1 = 15 (expected 15)
Mem[4] = 15 (expected 15)
$s6 = 000000ff (expected ff)
```

```
mips2 tb.v:94: $finish called at 850000 (1ps)
```

4] Conclusion

The implemented 5-stage pipelined MIPS processor successfully demonstrates:
• Functional 5-Stage MIPS RISC Pipeline Implementation
• Hazard-Free Execution via NOP Insertion
• Memory and Register File Functionality - validated LOAD/STORE operations
• Future work: Hazard Detection - forwarding/stalling, Extended Instruction Set Architecture, CISC implementations.
This project bridges theoretical computer architecture concepts with practical implementation of digital design, providing a foundation for advanced processor design studies.
All the codes, and other files provided by authors can be accessed on the following github repository: https://github.com/yashv373/MIPS-Microprocessor-Pipelined

References

[1] John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, 5th
Edition, Morgan Kaufmann, 2012.
[2] David Harris and Sarah Harris, Digital Design and Computer Architecture, Morgan Kaufmann, 2010.

APPENDIX: DETAILED EXPLANATION - CODE, TESTBENCH AND VVP OUTPUTS

ISA (Instruction Set Architecture)

Opcode / Funct	Instruction	Instruction Function	Type	Working Path in Code
6'b000000 / 6'b100000	ADD	Adds the two source registers and writes the result to rd	R-type	In the ID stage, when opcode = 000000 the control signals are set (ALUop = 4'b1010). The ALU control module decodes funct = 100000 to produce alu_control = 0000. The ALU then adds the

Opcode / Funct	Instruction	Instruction Function	Type	Working Path in Code
				value in ID_EX_rs and the second operand from mux_alu_src (ID_EX_rt) and sends the result down the pipeline.
6'b000000 / 6'b100010	SUB	Subtracts the second source register from the first	R-type	Similar to ADD: ID stage sets ALUop = 1010; alu_control decodes funct = 100010 to 0001. The ALU subtracts and passes the result along to EX_MEM_alu and later to the WB stage.
6'b000000 / 6'b100100	AND	Bitwise AND of the two source registers	R-type	ID stage: opcode 000000 → ALUop=1010; alu_control decodes funct = 100100 to 0010. The ALU performs an AND between ID_EX_rs and ID_EX_rt.
6'b000000 / 6'b100101	OR	Bitwise OR of the two source registers	R-type	In the ID stage, with opcode 000000 and ALUop=1010; alu_control decodes funct = 100101 to 0011. The ALU then performs an OR between the source register values.
6'b000000 / 6'b101010	SLT	Sets rd to 1 if rs < rt; otherwise, sets rd to 0	R-type	Again, opcode 000000 → ALUop=1010; alu_control decodes funct = 101010 to 0100. The ALU compares ID_EX_rs and ID_EX_rt, outputting 1 or 0.
6'b001000	ADDI	Adds an immediate value to a source register	I-type	In the ID stage, for opcode 001000 control signals are set with ALUop = 0000, and ALUSrc selects the immediate. The immediate value is sign-extended, and the ALU adds ID_EX_rs with the immediate (via mux_alu_src). The result is routed through the pipeline for write-back.
6'b001100	ANDI	Bitwise AND of a register and an	I-type	For opcode 001100, control is set with ALUop = 0010 and zero extension enabled. The

Opcode / Funct	Instruction	Instruction Function	Type	Working Path in Code
		immediate (treated as unsigned)		immediate is zero-extended, and mux_alu_src selects it to perform a bitwise AND with the source register value.
6'b001101	ORI	Bitwise OR of a register and an immediate (treated as unsigned)	I-type	Opcode 001101 sets ALUop = 0011 with zero extension; the immediate is zero-extended and, via mux_alu_src, used by the ALU to OR with the register value.
6'b001010	SLTI	Sets the destination to 1 if the register is less than the immediate	I-type	With opcode 001010, control sets ALUop = 0100 and selects the immediate (after sign extension) as the second operand (via mux_alu_src). The ALU performs the less-than comparison and the result is passed along.
6'b100011	LW	Loads a word from data memory into a register	I-type	Opcode 100011 sets control signals for a load (RegWrite = 1, MemToReg = 1, ALUop = 0000). The ALU computes an address using the base register and sign-extended immediate, then the MEM stage reads dmem at that address and the result is written back in the WB stage.
6'b101011	SW	Stores a word from a register into data memory	I-type	For opcode 101011, control signals indicate a store (MemWrite = 1, RegWrite = 0, ALUop = 0000). The ALU computes the effective address, and in the MEM stage, data from the second source register is written to dmem at that computed address.
6'b000010	J	Jumps to a target address	J-type	Opcode 000010 sets the jump control signal (ID_EX_jump = 1). In the IF stage, jump_address is computed from IF_ID_pc and the 26-bit address field of the instruction. A multiplexer

Opcode / Funct	Instruction	Instruction Function	Type	Working Path in Code
				(mux_jump) then selects this jump_address as the new PC (pc_next), causing an unconditional jump.

A Few Notes:

- **R-type Instructions:**
  The opcode is always 000000. The ALUop is set to a special value (4'b1010) to tell the ALU control module that the actual operation should be determined by the 6-bit funct field.
- **I-type Instructions:**
  They include an immediate value. The immediate is processed (via sign or zero extension) and selected as the ALU operand when needed.
- **J-type Instruction:**
  The jump instruction (J) uses a different format and bypasses the ALU for arithmetic; instead, it forms a new PC value directly.

Timescale

we first see a timescale command which help us have a defined timescale to avoid issues when we simulate regarding timing axis.

Module Instantiation

then we can see our module being instantiated
"module mips(input clk, input reset);"
as it clearly shows the module is called mips, and there are 2 inputs, a clock and a reset signal.

Registers

below this module instantiation we can see a lot of registers being declared with the help of "reg-command". These are actually the on-chip (on cpu) registers! lets explore them one by one.

1. pc - program counter - 32 bitwidth, is present at the beginning of the pipeline (if stage) and basically holds the address of the next instruction to be executed.
2. IF_ID_instr - 32 bits, Instruction Register (IF/ID Stage) - Stores the fetched instruction from memory. Between the Instruction Fetch (IF) and Instruction Decode (ID) stages. it holds the instruction which can then be decoded and executed and it can be imagined as todo checkpoint to know what to do
3. IF_ID_pc - 32 bit Program Counter (IF/ID Stage) , same function as 1
4. ID_EX_PC - 32 bit Program Counter (ID/EX Stage) , same function as 1
5. ID_EX_rs - source register 1 , a1 - decode stage - rd1, decode stage reads the value of rs
6. ID_EX_rt - source register 2, a2 - decode stage - rd2 , decode stage reads the value of rt
   5,6 are source registers. when adding something u take 2 values and then sum them. but u need to store these somewhere right, and this where the source registers help. they help store the operands. The names **rs** and **rt** come from the MIPS instruction format, which has historically used these names to indicate "source register" fields in the binary encoding.
7. ID_EX_imm - immediate value for I-type instructions - decode - sign extend - 15:0 initially then extended to 32 bits, The **imm** register holds an immediate value — a constant encoded within the instruction. Immediate values are "immediately" available within the instruction rather than coming from a register. They are used when you need a constant operand (like adding 5, not the value from a register). Many I-type instructions (e.g., `ADDI`, `ANDI`) use an immediate value. The processor extracts these bits from the instruction, extends them to 32 bits (with sign or zero extension), and then uses them as a second operand in the ALU.
8. ID_EX_rd - desitination register for r-type instructions- a3 in decode - The **rd** field in R-type instructions specifies **which register will receive the result** of an operation. It doesn't hold the result itself; instead, it stores the register number (or address) where the result should be written later in the write-back stage
9. ID_EX_funct - present in the execute stage - has the function code for alu operations for r-type instructions and this comes as 6 bit instruction sent to alu. The **funct** field is part of the R-type instruction format. Since the opcode for R-type is always 0, the specific operation (like ADD, SUB, etc.) is determined by this 6-bit field. It's like a secret handshake that tells the ALU exactly what to do when the opcode isn't enough.
10. ID_EX_aluop - ALU's control signals,which are found in the execute stage and it determines the alu operations to be done line whether to add or subtract or and or OR etc. the **aluop** is more like a shorthand for the control logic. In R-type, it signals "check the funct field" while for I-type it directly indicates the operation.

11. ID_EX_regwrite - register write enable signal, which is found in the writeback stage, and it controls whether the result is written back to a register, **RegWrite** is a control signal that tells the processor whether the current instruction should write a result back to the register file.
12. ID_EX_memtoret - Memory to Register Control Signal, Determines if the data to write back comes from memory (as in a load instruction) or the ALU. Directs the data path for write-back operations.
13. ID_EX_memwrite - Memory Write Control Signal, Signals whether a memory write (store) operation should be performed. Enables writing data from the register to the data memory during a store instruction.
14. ID_EX_zero_extend - Zero Extension Control Signal, Decides whether an immediate should be zero-extended (as opposed to sign-extended). Ensures that immediate values are correctly processed depending on the instruction type (useful for logical operations).
15. ID_EX_alusrc - ALU Source Selector, Determines whether the second operand for the ALU comes from a register or the immediate value. Chooses the proper data source for the ALU, which is essential for handling both register-based and immediate operations.
16. EX_MEM_alu - ALU Output Result, Holds the result of the ALU computation. Passes the computed value to memory stage or write-back, depending on the instruction.
17. EX_MEM_writedata - 32 bit register to Write Data for Memory Operations, it will hold the data that will be written to data memory during store operation. Prepares the data for a store instruction by transferring the correct value from the register file.
18. EX_MEM_rd - 5 bit destination register identifier, Ensures that the write-back stage knows exactly which register to update with the result.
19. EX_MEM_regwrite - Register Write Control Signal
20. EX_MEM_memtoret - Memory-to-Register Control Signal
21. EX_MEM_memwrit - Memory Write Control Signal
22. MEM_WB_readdata - 32b register for Data Read from Memory, Holds the data fetched from memory for load instructions. - - Delivers the memory contents to be written back into the register file.
23. MEM_WB_alu - 32b ALU Result (MEM/WB Stage)
24. MEM_WB_rd - Destination Register identifier (MEM/WB Stage)
25. MEM_WB_regwrite - MEM/WB Register Write Control Signal
26. MEM_WB_memtoret - MEM/WB Memory-to-Register Control Signal
27. imem - Instruction Memory, stores program instructions, used during IF stage , this memory array is 32x1024 memory block. where 1024 entries can be stored of 32bit entry each.
28. dmem - Data memory, it will hold the data for operations like LOAD and STORE used in MEM stage, same size i.e. 32x1024. Provides a storage space for data used and

generated by the processor during execution.

29. regfile - Register File, Contains the processor's general-purpose registers (like R0–R31) [GENERAL PURPOSE REGISTER COMPUTER ORGANIZATION!]. Accessed during the Instruction Decode (and sometimes Execute) stage. Each register is 32 bits; there are 32 registers in total. Serves as the fast, on-chip scratchpad for data—almost like your mental notepad for quick computations.

that covers all the registers we've declared, moving ahead lets discuss the data paths.

Datapath Wires

data path wires—they're the highways that carry signals (or "data packets") between various pipeline stages. Each wire is declared with a bitwidth and they connect submodules and registers in the processor.

```
wire [31:0] pc_plus4, pc_next, jump_address; wire [31:0] sign_extended,
zero_extended, imm_value; wire [31:0] alu_src_b, alu_result; wire [3:0]
alu_control;
```

- **pc_plus4 (32 bits)**
  - **Location:** Just after the PC in the IF stage.
  - **What:** The result of adding 4 to the current PC.
  - **Why:** Because instructions are word-aligned (4 bytes each), so the next sequential instruction is at PC+4.
  - **How:** An adder module computes this sum to move the instruction pointer forward.
- **pc_next (32 bits)**
  - **Location:** Input to the PC update.
  - **What:** The next PC value, which could be either pc_plus4 or a jump target.
  - **Why:** It determines which instruction to fetch next, based on whether there's a jump.
  - **How:** A multiplexer selects between the normal next address (pc_plus4) and a jump address.
- **jump_address (32 bits)**
  - **Location:** Calculated in the IF stage.
  - **What:** The full jump target address.
  - **Why:** J-type instructions require forming an address from parts of the instruction.

    - **How:** It combines the upper bits from IF_ID_pc with the 26-bit jump offset (and appends two zero bits) to form a 32-bit address.
- **sign_extended (32 bits)**
    - **Location:** In the decode stage.
    - **What:** A 16-bit immediate value extended to 32 bits using sign extension.
    - **Why:** To properly handle negative numbers in arithmetic operations.
    - **How:** A sign-extension module copies the sign bit to the upper 16 bits.
- **zero_extended (32 bits)**
    - **Location:** Also in the decode stage.
    - **What:** A 16-bit immediate value extended to 32 bits with zeros.
    - **Why:** Some instructions (like ANDI or ORI) treat immediate values as unsigned.
    - **How:** It pads the upper 16 bits with zeros.
- **imm_value (32 bits)**
    - **Location:** In the decode stage.
    - **What:** The final immediate value used by the ALU.
    - **Why:** It provides the second operand for immediate-type instructions.
    - **How:** A simple assignment chooses between sign_extended and zero_extended based on a control signal (ID_EX_zero_extend).
- **alu_src_b (32 bits)**
    - **Location:** In the execute stage.
    - **What:** The second operand input to the ALU.
    - **Why:** Some operations need an immediate value, others need a register value.
    - **How:** A multiplexer selects between the register (ID_EX_rt) and the immediate (ID_EX_imm) based on the ID_EX_alusrc signal.
- **alu_result (32 bits)**
    - **Location:** Output of the ALU in the execute stage.
    - **What:** The computed result from the ALU (e.g., sum, difference, bitwise operations).
    - **Why:** It's the main outcome of the ALU's processing used later in the pipeline.
    - **How:** The ALU uses the operands and alu_control to produce this result.
- **alu_control (4 bits)**
    - **Location:** Generated for the ALU.
    - **What:** A control signal that specifies the exact operation the ALU should perform.
    - **Why:** It decodes high-level instructions (via aluop and, if necessary, the funct field) into specific operations.
    - **How:** The alu_control module outputs this based on the aluop (from the decode stage) and, for R-type instructions, the funct field.

COMPUTATIONAL COMPONENTS

1. PC Increment and Jump Target Calculation

Adder for PC Increment

```
adder pc_adder(.a(PC), .b(32'd4), .sum(pc_plus4));
```

- **What It Is:**
  An instance of an adder module that computes `pc_plus4`.
- **Bitwidth:**
  32 bits for inputs and output.
- **Location in Pipeline:**
  This is used in the Instruction Fetch (IF) stage.
- **Functionality:**
  It adds a constant value (4) to the current Program Counter (PC). Since MIPS instructions are word-aligned (each instruction is 4 bytes), adding 4 moves the PC to the next sequential instruction.
- **Why It's Needed:**
  To ensure the processor always knows the address of the next instruction to fetch, unless altered by a jump.
- **How It Works:**
  The module takes the current PC and the constant `32'd4` (a 32-bit representation of the number 4) and outputs their sum as `pc_plus4`.

Jump Address Calculation

```
assign jump_address = {IF_ID_pc[31:28], IF_ID_instr[25:0], 2'b00};
```

- **What It Is:**
  A combinational assignment that calculates the jump target address.
- **Bitwidth:**
  32 bits.
- **Location in Pipeline:**
  Used in the IF stage (with IF/ID pipeline registers).
- **Functionality:**
  It constructs a complete jump address by combining:
  - **Upper 4 bits:** Taken from the current PC (`IF_ID_pc[31:28]`). This preserves the high-order bits of the PC.

    - **Middle 26 bits:** Taken from the instruction (`IF_ID_instr[25:0]`). This is the jump offset provided by the instruction.
    - **Lower 2 bits:** Hardcoded as `2'b00` to ensure the address is word-aligned.
- **Why It's Needed:**
  For J-type (jump) instructions, the processor must form a full 32-bit address from a 26-bit immediate. This concatenation ensures proper alignment and correct address formation.
- **How It Works:**
  The Verilog concatenation operator `{}` stitches these bit slices together into a valid 32-bit jump address.

2. ALU and ALU Control

Main ALU Module

```
alu main_alu(.a(ID_EX_rs), .b(alu_src_b), .alu_control(alu_control),
.result(alu_result));
```

- **What It Is:**
  The Arithmetic Logic Unit (ALU) that performs operations like addition, subtraction, bitwise AND/OR, and comparisons.
- **Bitwidth:**
  Operates on 32-bit operands; control signal is 4 bits.
- **Location in Pipeline:**
  Execute (EX) stage.
- **Functionality:**
    - **Inputs:**
        - **Operand A:** `ID_EX_rs` (the first source register's value).
        - **Operand B:** `alu_src_b`, which is selected by a multiplexer (see below).
        - **Control Signal:** `alu_control`, which tells the ALU which operation to perform.
    - **Output:**
        - **Result:** The computed value (`alu_result`), passed to later pipeline stages.
- **Why It's Needed:**
  To perform the arithmetic or logic operation specified by the instruction.
- **How It Works:**
  Depending on the `alu_control` signal, it selects the appropriate operation (e.g., addition for ADD, subtraction for SUB, etc.) and outputs the result.

ALU Control Module

```
alu_control alu_ctrl(.funct(ID_EX_funct), .aluop(ID_EX_aluop),
.alu_control(alu_control));
```

- **What It Is:**
  A module that decodes the high-level ALU operation request into a specific 4-bit control signal.
- **Bitwidth:**
  Inputs: `funct` is 6 bits, `aluop` is 4 bits; output is 4 bits.
- **Location in Pipeline:**
  Stays in the Execute stage control logic.
- **Functionality:**
  - **For R-type instructions:**
    When the opcode is `6'b000000`, `aluop` is set to a special value (e.g., `4'b1010`) indicating that the actual operation should be decoded using the 6-bit `funct` field.
  - **For I-type instructions:**
    The `aluop` directly represents the ALU function.
- **Why It's Needed:**
  It translates the instruction's intended operation (derived from the opcode and funct field) into a precise control code the ALU understands.
- **How It Works:**
  It checks `aluop` and, if necessary, further decodes the `funct` field (for R-type instructions) to produce the final `alu_control` signal.

3. Immediate Extension

Sign Extension

```
sign_extend #(16,32) se(.in(IF_ID_instr[15:0]), .out(sign_extended));
```

- **What It Is:**
  A module that extends a 16-bit immediate value to 32 bits by replicating the sign bit.
- **Bitwidth:**
  Input: 16 bits; Output: 32 bits.
- **Location in Pipeline:**
  Decode (ID) stage.
- **Functionality:**
  Converts a 16-bit signed immediate into a 32-bit signed value.

- **Why It's Needed:**
  To properly represent signed immediate values (like in ADDI) in the full 32-bit data path.
- **How It Works:**
  The module replicates the most significant bit (the sign bit) of the 16-bit input to fill the upper 16 bits of the 32-bit output.

Zero Extension

```
assign zero_extended = {16'b0, IF_ID_instr[15:0]};
```

- **What It Is:**
  A concatenation operation that extends the 16-bit immediate by prefixing it with 16 zeros.
- **Bitwidth:**
  32 bits.
- **Location in Pipeline:**
  Decode (ID) stage.
- **Functionality:**
  Provides a 32-bit representation of the immediate value without sign extension.
- **Why It's Needed:**
  Certain instructions (like ANDI, ORI) treat the immediate as an unsigned value, so no sign extension is desired.
- **How It Works:**
  It directly concatenates 16 zeros to the upper bits of the immediate value.

Selecting the Correct Immediate

```
assign imm_value = ID_EX_zero_extend ? zero_extended : sign_extended;
```

- **What It Is:**
  A conditional assignment that selects either the zero-extended or sign-extended immediate value.
- **Bitwidth:**
  32 bits.
- **Location in Pipeline:**
  Decode (ID) stage, with the value carried into the ID/EX register.
- **Functionality:**
  Chooses the correct version of the immediate based on the `ID_EX_zero_extend` control signal.

- **Why It's Needed:**
  Different instructions require different extensions. This selection ensures that the immediate value is correctly formatted for the operation.
- **How It Works:**
  If `ID_EX_zero_extend` is high (true), the zero-extended immediate is used; otherwise, the sign-extended version is chosen.

4. Multiplexers: For ALU Source and Jump

Multiplexer for ALU Source

```
mux2 #(32) mux_alu_src(.d0(ID_EX_rt), .d1(ID_EX_imm), .sel(ID_EX_alusrc),
.y(alu_src_b));
```

- **What It Is:**
  A 2-to-1 multiplexer that selects the second operand for the ALU.
- **Bitwidth:**
  32 bits.
- **Location in Pipeline:**
  Execute (EX) stage.
- **Functionality:**
  - **Option 0 (d0):**
    If `ID_EX_alusrc` is 0, the multiplexer passes the value from `ID_EX_rt` (the register value).
  - **Option 1 (d1):**
    If `ID_EX_alusrc` is 1, it selects `ID_EX_imm` (the immediate value).
- **Why It's Needed:**
  Many instructions come in two flavors:
  - **Register-Register Operations (R-type):** Use two register values.
  - **Immediate Operations (I-type):** Use one register value and one immediate.
- **How It Works:**
  The `ID_EX_alusrc` control signal determines which source is fed into the ALU as the second operand. This allows the same ALU to handle both kinds of operations.

Multiplexer for Jump Target Selection

```
mux2 #(32) mux_jump(.d0(pc_plus4), .d1(jump_address), .sel(ID_EX_jump),
.y(pc_next));
```

- **What It Is:**
  Another 2-to-1 multiplexer, this time used to select the next Program Counter (PC) value.
- **Bitwidth:**
  32 bits.
- **Location in Pipeline:**
  At the boundary between the IF stage and the next instruction fetch.
- **Functionality:**
  - **Option 0 (d0):**
    If `ID_EX_jump` is 0, it selects `pc_plus4` – the normal sequential flow.
  - **Option 1 (d1):**
    If `ID_EX_jump` is 1, it selects `jump_address` – redirecting the PC for a jump instruction.
- **Why It's Needed:**
  To handle jump instructions (J-type) where the next instruction is not sequentially located but at a target address provided by the instruction.
- **How It Works:**
  The control signal `ID_EX_jump` indicates when a jump is to occur. When set, the multiplexer chooses `jump_address` as the new PC; otherwise, it defaults to the normal incremented address (`pc_plus4`).

Summary

- **PC Increment and Jump:**
  The adder computes `PC + 4`, while the jump address is built from parts of the current PC and instruction. These feed into a multiplexer that selects the appropriate next PC value.
- **ALU and ALU Control:**
  The ALU performs operations on two operands (one from a register, one from a multiplexer that chooses between a register value or an immediate). The ALU control module decodes instructions (using the opcode and funct fields) to generate the correct control signals for the ALU.
- **Immediate Extension:**
  Immediate values from instructions are extended to 32 bits, either preserving their sign or zero-padding them, depending on the instruction's requirement.
- **Multiplexers:**
  They enable the dynamic selection of inputs – one for the ALU's second operand and one for determining the next instruction address (PC). Control signals determine which input is chosen based on the instruction type.

Each of these components is crucial for routing the correct data and control signals through the processor pipeline, ensuring that every instruction is executed as intended.

PIPELINE

Pipeline Stages Overview

Our design divides instruction execution into several stages. Each stage does its part and passes the relevant data and control signals to the next stage via pipeline registers. This organization is like an assembly line (supply line): one stage fetches the instruction, the next decodes it, and so on.

---

1. Instruction Fetch (IF) Stage

```
// IF Stage: Fetch instruction and update PC.
always @(posedge clk) begin
  if (reset) begin
    PC <= 0;
    IF_ID_instr <= 0;
    IF_ID_pc <= 0;
  end else begin
    IF_ID_instr <= imem[PC >> 2];
    IF_ID_pc <= PC;
    PC <= pc_next;
  end
end
```

What's Happening Here:

- **Clock Edge Trigger:**
  This `always` block triggers on the rising edge of the clock. Think of it as a “tick” in the processor's heartbeat.
- **Reset Condition:**
  - When `reset` is true, the processor sets the Program Counter (PC) to 0.
  - It clears the IF/ID pipeline registers (`IF_ID_instr` and `IF_ID_pc`) to ensure no garbage instruction or address is passed along.
- **Normal Operation:**
  - **Instruction Fetch:**

    - `IF_ID_instr <= imem[PC >> 2];`
      The instruction memory (`imem`) is indexed by the PC (dividing by 4 since instructions are word-aligned, hence `PC >> 2`).
      This fetches the instruction from memory and stores it in `IF_ID_instr`.
  - **PC Propagation:**
    - `IF_ID_pc <= PC;`
      The current PC value is stored in the IF/ID register for later stages. This helps later when we need to compute jump targets or simply know where the instruction came from.
  - **PC Update:**
    - `PC <= pc_next;`
      The next PC value, calculated by earlier logic (either sequential `pc_plus4` or a jump target), is loaded into the PC for the next cycle.

Intuitive Summary:

The IF stage acts like a “mailroom” that picks up the next instruction from the memory based on the PC and then updates the PC for the next cycle. It also passes along the instruction and its address to the decode stage.

---

2. Instruction Decode (ID) Stage

```
// ID Stage: Decode instruction, read registers, extend immediate, and set
control signals.
always @(posedge clk) begin
  if (reset) begin
    ID_EX_pc <= 0;
    ID_EX_rs <= 0;
    ID_EX_rt <= 0;
    ID_EX_imm  <= 0;
    ID_EX_rd <= 0;
    ID_EX_funct <= 0;
    ID_EX_aluop <= 0;
    {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite, ID_EX_jump,
ID_EX_zero_extend, ID_EX_alusrc} <= 0;
  end else begin
    // Pass the fetched PC into the next stage.
    ID_EX_pc <= IF_ID_pc;
    // Extract source register values using the instruction fields.
    ID_EX_rs <= regfile[IF_ID_instr[25:21]];
    ID_EX_rt <= regfile[IF_ID_instr[20:16]];
    // Process the immediate value (using sign or zero extension).
    ID_EX_imm <= imm_value;
    // Extract the funct field (for R-type instructions).
    ID_EX_funct <= IF_ID_instr[5:0];
    // Select the destination register: R-type uses bits [15:11], I-type
uses bits [20:16].
    ID_EX_rd <= (IF_ID_instr[31:26] == 6'b000000) ? IF_ID_instr[15:11] :
IF_ID_instr[20:16];

    // Set control signals based on opcode (bits [31:26] of the
instruction).
    case (IF_ID_instr[31:26])
      6'b000000: begin // R-type instructions (like ADD, SUB, etc.)
        // For R-type, set:
        //   - regwrite = 1 (result should be written back)
        //   - memtoret = 0 (result comes from ALU, not memory)
        //   - memwrite = 0 (no memory write operation)
        //   - aluop = 4'b1010 (special value to indicate decoding using the
funct field)
        //   - jump = 0 (not a jump instruction)
        //   - zero_extend = 0 (immediate is not used; if it were, it would
be sign-extended)
        //   - alusrc = 0 (the second ALU operand comes from the register,
not immediate)
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b0, 1'b0, 4'b1010, 1'b0, 1'b0, 1'b0};
      end
      6'b001000: begin // ADDI
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b0, 1'b0, 4'b0000, 1'b0, 1'b0, 1'b1};
      end
      6'b001100: begin // ANDI
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b0, 1'b0, 4'b0010, 1'b0, 1'b1, 1'b1};
      end
      6'b001101: begin // ORI
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b0, 1'b0, 4'b0011, 1'b0, 1'b1, 1'b1};
      end
      6'b001010: begin // SLTI
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b0, 1'b0, 4'b0100, 1'b0, 1'b0, 1'b1};
      end
      6'b100011: begin // LW (Load Word)
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b1, 1'b1, 1'b0, 4'b0000, 1'b0, 1'b0, 1'b1};
      end
      6'b101011: begin // SW (Store Word)
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b0, 1'b0, 1'b1, 4'b0000, 1'b0, 1'b0, 1'b1};
      end
      6'b000010: begin // J (Jump)
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc}
          <= {1'b0, 1'b0, 1'b0, 4'b0000, 1'b1, 1'b0, 1'b0};
      end
      default: begin
        {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite,
         ID_EX_aluop, ID_EX_jump, ID_EX_zero_extend, ID_EX_alusrc} <= 0;
      end
    endcase
  end
end
```

Detailed Breakdown:

- **Reset Block:**
  - When reset is active, every signal in the ID/EX pipeline register is set to 0. This is to ensure that no old or invalid data is passed along after a reset.
- **Register Propagation and Instruction Field Extraction:**
  - `ID_EX_pc <= IF_ID_pc;`
    The PC from the fetch stage is forwarded to maintain the context of where the instruction came from.
  - `ID_EX_rs <= regfile[IF_ID_instr[25:21]];`
    Extracts the first source register field (`rs`) from the instruction. It then reads that register's value from the register file.
  - `ID_EX_rt <= regfile[IF_ID_instr[20:16]];`
    Similarly, extracts the second source register field (`rt`) and reads its value.
  - `ID_EX_imm <= imm_value;`
    Uses the precomputed immediate value (resulting from sign or zero extension) and passes it along.
  - `ID_EX_funct <= IF_ID_instr[5:0];`
    For R-type instructions, the last 6 bits (`funct`) are used to determine the specific

ALU operation.

- `ID_EX_rd <= ...`
  Chooses the destination register field.
  - If the opcode is 0 (R-type), use bits [15:11].
  - Otherwise (for I-type instructions), use bits [20:16].
    This determines where the result should be written later.

- **Control Signal Assignment (via `case` statement):**
  - Based on the opcode (the upper 6 bits of the instruction), the control signals are set for each instruction type:
    - **R-type (opcode 6'b000000):**
      - `regwrite` is enabled (1) because the result should be written back.
      - `memtoret` is 0 because the result comes from the ALU, not memory.
      - `memwrite` is 0 as no data is written to memory.
      - `aluop` is set to `4'b1010` to indicate that the ALU control should use the funct field to decide the exact operation.
      - `jump`, `zero_extend`, and `alusrc` are set appropriately (in this case, jump is 0, and alusrc is 0 meaning the second ALU operand comes from the register).
    - **ADDI (opcode 6'b001000):**
      - Here, `alusrc` is set to 1, meaning the second ALU operand is the immediate value.
      - `aluop` is `4'b0000`, typically representing an add operation.
    - **ANDI, ORI, SLTI, LW, SW, J:**
      - Similar patterns follow for these instructions.
      - For example, LW (load word) sets `memtoret` to 1 to indicate that data should come from memory, while SW (store word) disables `regwrite` and enables `memwrite` to perform a store.
    - The `case` structure makes it clear how each instruction is configured for proper data path control.

Intuitive Summary of ID Stage:

The ID stage acts like a “decoder” and “dispatcher.” It reads the instruction from the previous stage, extracts the fields (register numbers, immediate, function code), reads the register file for operand values, and sets the control signals that tell subsequent stages how to process the instruction. Think of it as the brain that says, “I know what this instruction wants to do, so let’s set the stage for the ALU, memory, or jump operations!”

---

3. Execute (EX) Stage

```
// EX Stage: Execute ALU operation and propagate control signals.
always @(posedge clk) begin
  if (reset) begin
    EX_MEM_alu <= 0;
    EX_MEM_writedata <= 0;
    EX_MEM_rd <= 0;
    {EX_MEM_regwrite, EX_MEM_memtoret, EX_MEM_memwrite} <= 0;
  end else begin
    EX_MEM_alu <= alu_result;
    EX_MEM_writedata <= ID_EX_rt;
    EX_MEM_rd <= ID_EX_rd;
    {EX_MEM_regwrite, EX_MEM_memtoret, EX_MEM_memwrite}
       <= {ID_EX_regwrite, ID_EX_memtoret, ID_EX_memwrite};
  end
end
```

Detailed Explanation:

- **Reset Condition:**
  - On reset, the pipeline registers for the EX/MEM stage are cleared to avoid passing invalid data.
- **Normal Operation:**
  - **ALU Result Propagation:**
    - `EX_MEM_alu <= alu_result;`
      The result computed by the ALU (from the EX stage) is passed to the EX/MEM register. This result might be an arithmetic result, a computed address, or a logical output.
  - **Data for Memory Write:**
    - `EX_MEM_writedata <= ID_EX_rt;`
      For store instructions, the value to be written to memory comes from the second source register (`rt`), which was read during the ID stage.
  - **Destination Register Propagation:**
    - `EX_MEM_rd <= ID_EX_rd;`
      The chosen destination register (either from the R-type field or the I-type field) is forwarded so the WB stage knows where to write the result.
  - **Control Signals:**
    - `{EX_MEM_regwrite, EX_MEM_memtoret, EX_MEM_memwrite}` are simply passed along from the ID/EX stage to control later actions in the MEM and WB stages.

Intuitive Summary of EX Stage:

The EX stage is where the heavy lifting happens – the ALU performs the required calculation or logical operation. Once computed, the result, along with necessary control information and data, is forwarded to the next stage for potential memory access or write-back.

---

4. Memory Access (MEM) Stage

```verilog
// MEM Stage: Memory access
always @(posedge clk) begin
  if (reset) begin
    MEM_WB_readdata <= 0;
    MEM_WB_alu <= 0;
    MEM_WB_rd <= 0;
    {MEM_WB_regwrite, MEM_WB_memtoret} <= 0;
  end else begin
    if (EX_MEM_memwrite)
      dmem[EX_MEM_alu >> 2] <= EX_MEM_writedata;
    MEM_WB_readdata <= dmem[EX_MEM_alu >> 2];
    MEM_WB_alu <= EX_MEM_alu;
    MEM_WB_rd <= EX_MEM_rd;
    {MEM_WB_regwrite, MEM_WB_memtoret} <= {EX_MEM_regwrite,
EX_MEM_memtoret};
  end
end
```

Detailed Explanation:

- **Reset Condition:**
  - When reset is active, all MEM/WB registers are cleared.
- **Memory Write Operation (Store Word):**
  - **Conditional Write:**
    - `if (EX_MEM_memwrite)` checks if the control signal indicates a memory write.
    - `dmem[EX_MEM_alu >> 2] <= EX_MEM_writedata;`
      If true, the data (`EX_MEM_writedata`) is written into data memory (`dmem`). The memory address is calculated using `EX_MEM_alu >> 2` (dividing by 4 since addresses are word-aligned).
- **Memory Read Operation (Load Word):**
  - Regardless of whether a store occurred, the processor reads from memory:

- `MEM_WB_readdata <= dmem[EX_MEM_alu >> 2];`
  For load instructions, this is the value read from memory at the computed address.

- **Pass Through of ALU Result and Register Info:**
  - `MEM_WB_alu <= EX_MEM_alu;`
    The ALU result is passed along because, for non-load instructions, this is the value that will be written back.
  - `MEM_WB_rd <= EX_MEM_rd;`
    The destination register remains unchanged.
- **Control Signals Propagation:**
  - `{MEM_WB_regwrite, MEM_WB_memtoret} <= {EX_MEM_regwrite, EX_MEM_memtoret};`
    The control signals that determine if a register should be written back and whether that data comes from memory or the ALU are also forwarded.

Intuitive Summary of MEM Stage:

In the MEM stage, the processor accesses the data memory. For store instructions, it writes data into memory. For load instructions, it reads data from memory. It also forwards the ALU result and control signals so the next stage knows what to do.

---

5. Write-Back (WB) Stage

```
// WB Stage: Write back to register file.
always @(posedge clk) begin
  if (reset) begin
    for (integer i = 0; i < 32; i = i + 1)
      regfile[i] <= 0;
  end else if (MEM_WB_regwrite && MEM_WB_rd != 0) begin
    regfile[MEM_WB_rd] <= MEM_WB_memtoret ? MEM_WB_readdata : MEM_WB_alu;
  end
end
```

Detailed Explanation:

- **Reset Condition:**
  - On reset, the entire register file is cleared.
  - The loop goes through all 32 registers, setting them to 0. This ensures that the processor starts with a clean slate.
- **Write Back Operation:**

- **Condition Check:**
    - `if (MEM_WB_regwrite && MEM_WB_rd ≠ 0)`
      The processor checks if the `regwrite` control signal is active and ensures that the destination register is not register 0 (which is hardwired to 0 in MIPS).
- **Choosing the Data Source:**
    - `regfile[MEM_WB_rd] ≤ MEM_WB_memtoret ? MEM_WB_readdata : MEM_WB_alu;`
      This statement writes the final result back into the register file:
        - If `MEM_WB_memtoret` is true, it means the data should come from memory (as in a load instruction), so it writes `MEM_WB_readdata`.
        - Otherwise, it writes the ALU result (`MEM_WB_alu`), as is typical for arithmetic or logical operations.

Intuitive Summary of WB Stage:

The WB stage is the final step in the instruction's journey. Here, the computed result (whether from memory or the ALU) is written back into the processor's register file. It's like updating your "notebook" with the new results so that future instructions can use them.

Overall Picture

- **IF Stage:**
  Fetch the instruction and update the program counter.
- **ID Stage:**
  Decode the instruction, read the necessary registers, extend immediates, and set up control signals.
- **EX Stage:**
  Execute the instruction using the ALU and prepare data for memory or write-back.
- **MEM Stage:**
  Access memory for load or store instructions, and forward results.
- **WB Stage:**
  Write the final result back to the register file.

Each stage is carefully synchronized by the clock, and the pipeline registers ensure that data flows smoothly from one stage to the next. This modular approach is what makes pipelining both powerful and challenging – it allows for multiple instructions to be in different stages of execution simultaneously, increasing the overall throughput of the processor.

Submodules

1. Adder Module

```
module adder(input [31:0] a, b, output [31:0] sum);
  assign sum = a + b;
endmodule
```

What It Does:

- **Purpose:** This module takes two 32-bit inputs, `a` and `b`, and produces their sum.
- **How It Works:**
  - The `assign` statement continuously evaluates `a + b` and assigns the result to `sum`.
- **Why It's Needed:**
  - In our processor, it is used to compute the next sequential Program Counter (PC) value by adding 4 (since each instruction is 4 bytes).
- **Bitwidth:**
  - All signals (a, b, and sum) are 32 bits wide, matching the word size of the processor.
- **Intuitive Analogy:**
  - Think of the adder as a simple calculator that always adds two numbers. Here, it's primarily used to "step" the PC forward.

---

2. ALU (Arithmetic Logic Unit) Module

```
module alu(input [31:0] a, b, input [3:0] alu_control, output reg [31:0]
result);
  always @(*) begin
    case (alu_control)
      4'b0000: result = a + b;
      4'b0001: result = a - b;
      4'b0010: result = a & b;
      4'b0011: result = a | b;
      4'b0100: result = (a < b) ? 1 : 0;
      default: result = 0;
    endcase
  end
endmodule
```

What It Does:

- **Purpose:** The ALU performs arithmetic and logical operations based on a control signal.
- **How It Works:**
  - The `always @(*)` block means it continuously checks its inputs.
  - The `case` statement uses the 4-bit `alu_control` signal to decide which operation to perform:
    - **0000:** Addition (a + b)
    - **0001:** Subtraction (a - b)
    - **0010:** Bitwise AND (a & b)
    - **0011:** Bitwise OR (a | b)
    - **0100:** Set Less Than – outputs 1 if a is less than b; else outputs 0.
  - The `default` case sets the result to 0 if none of the cases match.
- **Why It's Needed:**
  - It's the "workhorse" of the processor. The ALU performs calculations for arithmetic operations, logical operations, and comparisons.
- **Bitwidth:**
  - Operands `a` and `b`, and the `result` are all 32 bits, ensuring compatibility with our 32-bit architecture.
- **Intuitive Analogy:**
  - Imagine the ALU as a multi-tool that can both calculate numbers and make decisions (like comparing values) based on instructions given by the control signal.

---

3. ALU Control Module

```
module alu_control(input [5:0] funct, input [3:0] aluop, output reg [3:0]
alu_control);
  always @(*) begin
    case (aluop)
      4'b1010: begin // R-type: decode using funct field
        case (funct)
          6'b100000: alu_control = 4'b0000; // ADD
          6'b100010: alu_control = 4'b0001; // SUB
          6'b100100: alu_control = 4'b0010; // AND
          6'b100101: alu_control = 4'b0011; // OR
          6'b101010: alu_control = 4'b0100; // SLT
          default:   alu_control = 4'b0000;
        endcase
      end
      default: begin // I-type: pass ALUop directly
        alu_control = aluop;
      end
    endcase
  end
endmodule
```

What It Does:

- **Purpose:** This module converts high-level ALU control information into a 4-bit code that the ALU understands.
- **How It Works:**
  - It receives two inputs:
    - `aluop` **(4 bits):** A signal from the control logic in the decode stage.
    - `funct` **(6 bits):** A field from R-type instructions that specifies the operation.
  - **For R-type instructions:**
    - When `aluop` equals `4'b1010`, the module enters a nested `case` block that checks the `funct` field to determine the exact operation:
      - For example, `funct` equal to `6'b100000` sets `alu_control` to `4'b0000` (for ADD).
  - **For I-type instructions:**
    - The `default` block simply passes `aluop` to `alu_control`, because the opcode directly dictates the operation.
- **Why It's Needed:**
  - R-type instructions have a generic opcode (0), so the detailed operation is specified in the `funct` field. This module disambiguates the operation.
- **Bitwidth:**
  - The outputs and control signals match the bitwidth required by the ALU (4 bits for `alu_control`).
- **Intuitive Analogy:**
  - Picture it as a translator that takes high-level instructions (like "do R-type thing") and then reads the fine print (`funct`) to tell the ALU exactly what to do.

---

4. Sign Extend Module

```
module sign_extend #(parameter IN = 16, parameter OUT = 32)
  (input [IN-1:0] in, output [OUT-1:0] out);
  assign out = {{OUT-IN{in[IN-1]}}, in};
endmodule
```

What It Does:

- **Purpose:** This module converts a 16-bit number into a 32-bit number while preserving its sign.
- **How It Works:**
  - It uses Verilog's concatenation operator to extend the input.
  - `{{OUT-IN{in[IN-1]}}, in}` means:
    - Replicate the most significant bit (`in[IN-1]`, which is the sign bit) enough times to fill the upper bits.
    - Then concatenate it with the original 16-bit number.
- **Why It's Needed:**
  - Many instructions use a 16-bit immediate value. For arithmetic operations, we need to extend these to 32 bits while preserving the sign (for example, -5 remains negative).
- **Parameters:**
  - The parameters allow for flexibility; by default, the module extends a 16-bit input to a 32-bit output.
- **Intuitive Analogy:**
  - Think of it like "stretching" a short number to fit a bigger frame while keeping its "personality" (sign) intact.

---

5. 2-to-1 Multiplexer (mux2) Module

```
module mux2 #(parameter WIDTH = 32)
  (input [WIDTH-1:0] d0, d1, input sel, output [WIDTH-1:0] y);
  assign y = sel ? d1 : d0;
endmodule
```

What It Does:

- **Purpose:** The multiplexer selects one of two 32-bit inputs to pass to the output based on a single control signal (`sel`).
- **How It Works:**
  - The conditional operator `? :` checks the value of `sel`.
    - If `sel` is true (1), `y` is assigned `d1`.
    - If `sel` is false (0), `y` is assigned `d0`.
- **Why It's Needed:**
  - In the processor, multiplexers are used to choose between different data sources:

        - For example, selecting whether the second ALU operand comes from a register or the immediate value.
        - Another multiplexer selects whether the next PC is `pc_plus4` (sequential) or `jump_address` (for jumps).
- **Bitwidth:**
    - The `WIDTH` parameter is set to 32 by default, so the inputs and output are 32 bits.
- **Intuitive Analogy:**
    - Imagine a light switch that chooses between two different inputs (like picking between two TV channels). The `sel` signal is the switch that decides which input "channel" goes through to the output.

Overall Picture

- **Adder:**
    - Simply adds two 32-bit numbers – essential for PC updating.
- **ALU:**
    - Performs the main calculations and logical operations as dictated by the instruction.
- **ALU Control:**
    - Acts as a translator, decoding the finer details of the operation (especially for R-type instructions) to set up the ALU.
- **Sign Extend:**
    - Makes sure that a 16-bit immediate value is properly extended to 32 bits, preserving its sign.
- **Mux2:**
    - A basic selector that picks one of two 32-bit inputs based on a control signal. It's used in multiple places to choose between different data paths.

TESTBENCH

Remember, the testbench is our "laboratory" where we simulate our processor design. In this case, we specifically choose test cases where hazards (conflicts between instructions) don't occur by inserting NOPs between instructions.

Overall Testbench Structure

The testbench is written in Verilog and is used to simulate the MIPS processor design (named `mips` in our module). It performs three main tasks:

1. **Clock Generation:**
   It creates a clock signal that drives the simulation.
2. **Memory and Register Initialization:**
   It sets up the instruction memory (`imem`), data memory (`dmem`), and the register file with predetermined values.
3. **Simulation Control and Output:**
   It applies reset, releases it after a short period, waits for the design to run, and then displays the final states of registers and memory.

Detailed Walk-Through

Module Declaration

```
module tb;
  reg clk, reset;
  mips cpu(clk, reset);
```

- `module tb;`
  This declares the testbench module, named `tb` (short for "testbench").
- `reg clk, reset;`
  Two registers (variables) are declared:
  - `clk`: the clock signal.
  - `reset`: the reset signal.
- `mips cpu(clk, reset);`
  This instantiates the MIPS processor (our design under test) as `cpu` and connects it to the testbench clock and reset signals. Think of this as "plugging in" our processor into the simulation environment.

Clock Generation

```
  // Clock generation: period = 10 ns
  always #5 clk = ~clk;
```

- `always #5 clk = ~clk;`
  This line creates a clock signal:
  - The keyword `always` means the statement executes repeatedly.
  - `#5` indicates a delay of 5 nanoseconds.
  - `clk = ~clk;` toggles the clock (if it's 0, it becomes 1; if it's 1, it becomes 0).

- **What It Means:**
  Every 5 ns the clock flips, so one full clock cycle is 10 ns. This is our heartbeat that synchronizes the processor's operations.

Testbench Initialization Block

```
  // Testbench: Instruction and Data Memory Initialization
  initial begin
    clk = 0;
    reset = 1;
```

- `initial begin ... end`
  An `initial` block runs once at the start of the simulation.
- `clk = 0; reset = 1;`
  Initially, the clock is set to 0 and reset is asserted (set to 1). This ensures the processor starts in a known state.

Instruction Memory Initialization

The next section sets up the instructions in the instruction memory (`imem`) of the processor. Each instruction is 32 bits wide and stored in an array. Notice the use of NOPs (No Operation instructions, encoded as `32'h00000000`) inserted between real instructions to avoid hazards.

Example: First Instruction – ADDI

```
    //-----------------------------------------------------------------------
---
    // Instruction Memory Initialization - NOPs inserted to ensure proper
WB.
    //-----------------------------------------------------------------------
---
    // Address 0: addi $s0, $zero, 5   ⟶ $s0 = 5 (dest: reg16)
    cpu.imem[0] = 32'h20100005;
    cpu.imem[1] = 32'h00000000; // NOP
    cpu.imem[2] = 32'h00000000; // NOP
```

- `cpu.imem[0] = 32'h20100005;`
  This instruction is an ADDI (add immediate) operation.
  - Opcode `001000` is encoded here as `20` (in hexadecimal) in the upper bits.
  - It tells the processor to add the immediate value 5 to register `$zero` (always 0) and store the result in `$s0` (which is register 16 in MIPS convention).

- **NOPs:**
  The next two memory locations (addresses 1 and 2) are filled with NOP instructions.
  - **Why NOPs?**
    They ensure that the write-back stage has enough time to complete before the next instruction that might depend on `$s0` is executed. This avoids hazards that would require forwarding or hazard detection (which our design does not implement).

Subsequent Instructions

The testbench initializes several other instructions in a similar fashion. Here's a quick rundown:

- **Address 3:**
  `addi $s1, $zero, 10` – Sets `$s1` (register 17) to 10.
- **Address 7:**
  `add $t0, $s1, $s0` – Adds `$s1` and `$s0` and stores the result (expected 15) in `$t0` (register 8).
- **Address 10:**
  `andi $s2, $s0, 0xF` – Performs a bitwise AND between `$s0` and `0xF` and stores the result in `$s2` (register 18). Expected result is 5.
- **Address 13:**
  `ori $s3, $s1, 0xF0` – Performs a bitwise OR between `$s1` and `0xF0` and stores the result in `$s3` (register 19). Expected result is 0xFA (or 250 in decimal).
- **Address 16:**
  `slti $s4, $s1, 10` – Sets `$s4` (register 20) to 1 if `$s1` is less than 10; otherwise, 0. Expected result is 0.
- **Address 19:**
  `sw $t0, 16($zero)` – Stores the value of `$t0` (15) into data memory at an address calculated as `$zero + 16`. Since `$zero` is 0, the value is stored at address 16 (which maps to index 4, as addresses are word-aligned).
- **Address 22:**
  `lw $t1, 16($zero)` – Loads a word from data memory at address 16 into `$t1` (register 9). Expected value is 15.
- **Address 25:**
  `addi $s6, $zero, 0xFF` – Sets `$s6` (register 22) to 0xFF (255 in decimal).

Each instruction is followed by two NOPs to prevent hazards, ensuring that write-back completes before a new instruction might try to use the result.

Filling Remaining Instruction Memory

```
    // Fill the rest of imem with NOPs.
    for (integer i = 28; i < 1024; i = i + 1)
      cpu.imem[i] = 32'h00000000;
```

- **What It Does:**
  This loop fills all remaining memory locations (from address 28 to 1023) with NOPs.
- **Why:**
  It avoids having random data in the instruction memory that might be erroneously executed.

Register File and Data Memory Initialization

```
    //-----------------------------------------------------------------------
---
    // Initialize register file and data memory to zero.
    //-----------------------------------------------------------------------
---
    for (integer i = 0; i < 32; i = i + 1)
      cpu.regfile[i] = 0;
    for (integer i = 0; i < 1024; i = i + 1)
      cpu.dmem[i] = 0;
```

- **Register File Initialization:**
  - The loop sets all 32 registers in the processor's register file to 0.
  - This ensures that no register contains a garbage value at the start.
- **Data Memory Initialization:**
  - Similarly, this loop clears all 1024 words of the data memory.
  - This is important to ensure predictable behavior when instructions like `lw` (load word) are executed.

Reset Release and Simulation Time Control

```
    // Release reset after an extended interval, then wait for pipeline
flush.
    #50 reset = 0;
    #800;
```

- `#50 reset = 0;`
  After 50 nanoseconds, the reset signal is de-asserted (set to 0).

  - **Why?**
    This gives the processor enough time to initialize and ensures that the reset state persists long enough for the entire design to clear out any unintended data.
- `#800;`
  The simulation then waits for an additional 800 nanoseconds.
  - **Purpose:**
    This delay lets the processor run through enough clock cycles so that all instructions (and the NOPs) propagate through the pipeline and finish executing.
  - Essentially, we're giving the pipeline time to "flush" – that is, to process all instructions from fetch through write-back.

Final State Display

```
    // Display final states.
    $display("\nFinal Register/Memory State at ~350 ns:");
    $display("$s0 = %d (expected 5)",  cpu.regfile[16]);
    $display("$s1 = %d (expected 10)", cpu.regfile[17]);
    $display("$t0 = %d (expected 15)", cpu.regfile[8]);
    $display("$s2 = %h (expected 5)",  cpu.regfile[18]);
    $display("$s3 = %h (expected fa)", cpu.regfile[19]);
    $display("$s4 = %d (expected 0)",  cpu.regfile[20]);
    $display("$t1 = %d (expected 15)", cpu.regfile[9]);
    $display("Mem[4] = %d (expected 15)", cpu.dmem[4]);
    $display("$s6 = %h (expected ff)", cpu.regfile[22]);

    $finish;
```

- `$display` **Statements:**
  These commands print out the final values of certain registers and data memory locations to the console.
  - For example, it checks that register `$s0` (register 16) is 5, `$t0` (register 8) is 15, etc.
  - The expected values are commented beside each display command.
- `$finish;`
  This tells the simulator to end the simulation after displaying the results.

Waveform Dumping and Monitoring

```
  initial begin
    $dumpfile("waves.vcd");
    $dumpvars(0, tb);
    $monitor("Time=%0t: PC=%h, Instr=%h", $time, cpu.PC, cpu.IF_ID_instr);
  end
endmodule
```

- **Waveform Dump:**
  - `$dumpfile("waves.vcd");`
    This command tells the simulator to write signal changes into a file named `waves.vcd`. This file can be viewed later with a waveform viewer to see how signals change over time.
  - `$dumpvars(0, tb);`
    This specifies which variables to dump. Here, all variables in the testbench (`tb`) are recorded.
- **Monitoring:**
  - `$monitor("Time=%0t: PC=%h, Instr=%h", $time, cpu.PC, cpu.IF_ID_instr);`
    This command continuously prints out the current simulation time, the Program Counter (PC), and the current instruction in the IF/ID pipeline register.
    - **Purpose:**
      It provides real-time feedback during simulation, helping to verify that the processor is fetching and processing instructions correctly.

Why This Testbench Avoids Hazards

- **NOP Insertion:**
  The testbench intentionally inserts NOPs (No Operation instructions) between critical instructions. This spacing ensures that:
  - The pipeline stages have enough time to complete a write-back before a new instruction that depends on that result is fetched.
  - Data hazards (when one instruction depends on the result of a previous instruction) are avoided since the NOPs give the processor time to flush the pipeline.
- **Controlled Execution:**
  By initializing the memories and registers to known values, and by clearly defining the instruction sequence, the testbench ensures that hazards (like data or control hazards) don't occur, making it easier to verify that the design works correctly under "ideal" conditions.

This testbench is a carefully crafted simulation environment:

- **Clock Generation** creates a regular heartbeat.
- **Memory and Register Initialization** ensures a clean start.

- **Instruction Sequencing with NOPs** guarantees that our pipeline doesn't run into hazards that the design isn't equipped to handle.
- **Monitoring and Waveform Dumping** provide visibility into the internal workings of the processor during simulation.

VVP Output analysis :

VCD Output Initialization

```
VCD info: dumpfile waves.vcd opened for output.
```

- **What It Means:**
  The simulator has opened a VCD (Value Change Dump) file named `waves.vcd` where all signal transitions will be recorded.
- **Why It's Important:**
  This file lets you later view waveforms with a waveform viewer to trace signal changes over time, a common debugging tool in digital design.

Monitor Output – Early Time

```
Time=0: PC=xxxxxxxx, Instr=xxxxxxxx
```

- **What It Means:**
  At time zero, the monitor shows unknown values (`xxxxxxxx`) for both the Program Counter (PC) and the Instruction (Instr).
- **Why It Happens:**
  Before the simulation begins proper (or before reset has fully initialized the design), the signals may be undefined.
- **Relation to Our Design:**
  We start with a reset condition that later forces these values to known states.

After Reset – First Few Cycles

```
Time=5000: PC=00000000, Instr=00000000
```

- **At Time=5000 (5,000 time units):**
  - **PC=00000000:** The processor's PC is zero, as expected during reset.

    - **Instr=00000000:** The IF/ID instruction register holds a NOP (No Operation), meaning no meaningful instruction is being executed yet.
- **Why It's Correct:**
  The testbench initially asserts reset. During this period, our design clears pipeline registers. Hence, the PC and fetched instruction are zero.

Instruction Fetching – Beginning of Execution

```
Time=55000: PC=00000004, Instr=20100005
```

- **At Time=55,000:**
    - **PC=00000004:** The PC has advanced from 0 to 4 (recall our adder adds 4 every cycle, moving to the next word-aligned address).
    - **Instr=20100005:**
        - This 32-bit code is the first instruction (at address 0) stored in the instruction memory.
        - It encodes `addi $s0, $zero, 5` (opcode 001000, setting `$s0` to 5).
- **Why It's Correct:**
  After reset is released, the processor fetches the instruction at address 0. Our testbench placed `20100005` at `imem[0]`, so the processor correctly reads and displays it.

NOP Insertion – Ensuring Proper Pipeline Timing

```
Time=65000: PC=00000008, Instr=00000000
Time=75000: PC=0000000c, Instr=00000000
```

- **At Times 65,000 and 75,000:**
    - The PC increments by 4 each time.
    - The instructions at these addresses are NOPs (`00000000`), inserted by the testbench.
- **Why It's Correct:**
  These NOPs create a delay that allows the result of the previous instruction (the addi that sets `$s0`) to propagate fully through the pipeline. This avoids hazards since our design doesn't have forwarding.

Fetching the Next ADDI Instruction

```
Time=85000: PC=00000010, Instr=2011000a
```

- **At Time=85,000:**
  - **PC=00000010:** The processor has now advanced to address 4 (PC value 0x10).
  - **Instr=2011000a:**
    - This encodes `addi $s1, $zero, 10` (setting `$s1` to 10).
- **Why It's Correct:**
  Our testbench initialized `imem[3]` with this instruction. Note that NOPs inserted before ensured that `$s1` is set correctly without interference.

Further PC Increments with NOPs

```
Time=95000: PC=00000014, Instr=00000000
Time=105000: PC=00000018, Instr=00000000
Time=115000: PC=0000001c, Instr=00000000
```

- **What's Happening:**
  The processor is still fetching NOP instructions.
- **Why It's Correct:**
  These NOPs are inserted to create delays in the pipeline, ensuring that the results from the previous ADDI instructions are completely written back before they are used by subsequent instructions.

Fetching the ADD Instruction

```
Time=125000: PC=00000020, Instr=02304020
```

- **At Time=125,000:**
  - **PC=00000020:** The PC is at address 8 (0x20 in hexadecimal).
  - **Instr=02304020:**
    - This is the encoding for the R-type instruction `add $t0, $s1, $s0`.
    - It adds the values in registers `$s1` (10) and `$s0` (5) and writes the result (15) into `$t0`.
- **Why It's Correct:**
  Our testbench placed this instruction at `imem[7]`. The fetched instruction matches the expected encoding for the add operation.

More NOPs – Pipeline Pauses

```
Time=135000: PC=00000024, Instr=00000000
Time=145000: PC=00000028, Instr=00000000
```

- **Explanation:**
  Again, NOPs are being fetched, giving the pipeline time to complete the ADD instruction's write-back stage before the next instruction.
- **Why It's Correct:**
  These NOPs prevent any hazards by ensuring there's no immediate dependency on the previous instruction's result.

Fetching the ANDI Instruction

```
Time=155000: PC=0000002c, Instr=3212000f
```

- **At Time=155,000:**
  - **PC=0000002c:** Address is now at 0x2c.
  - **Instr=3212000f:**
    - This encodes `andi $s2, $s0, 0xF`.
    - The operation performs a bitwise AND between the value in `$s0` (which is 5) and `0xF` (15), which yields 5.
- **Why It's Correct:**
  The instruction was correctly placed at `imem[10]` by the testbench. The encoding and operation match our design expectations.

Fetching the ORI Instruction

```
Time=185000: PC=00000038, Instr=363300f0
```

- **At Time=185,000:**
  - **PC=00000038:** The PC now points to the ORI instruction's location.
  - **Instr=363300f0:**
    - This encodes `ori $s3, $s1, 0xF0`.
    - The operation computes `$s1 | 0xF0`. With `$s1` equal to 10, ORing with 0xF0 yields 0xFA (250 decimal).
- **Why It's Correct:**
  Our testbench set up this instruction at `imem[13]`. The fetched value matches the expected ORI operation.

Fetching the SLTI Instruction

```
Time=215000: PC=00000044, Instr=2a34000a
```

- **At Time=215,000:**
  - **PC=00000044:** Points to the SLTI instruction.
  - **Instr=2a34000a:**
    - This encodes `slti $s4, $s1, 10`.
    - Since `$s1` is 10, the condition `$s1 < 10` is false, so `$s4` should be set to 0.
- **Why It's Correct:**
  The testbench placed this instruction at `imem[16]`. The fetched instruction confirms the SLTI operation is in progress.

Fetching the SW Instruction

```
Time=245000: PC=00000050, Instr=ac080010
```

- **At Time=245,000:**
  - **PC=00000050:** Now we see the store word (SW) instruction.
  - **Instr=ac080010:**
    - This encodes `sw $t0, 16($zero)`.
    - It means store the value in `$t0` (which should be 15 from the earlier ADD) into data memory at address 16.
- **Why It's Correct:**
  Our testbench assigned this instruction at `imem[19]`. The SW operation is triggered as expected.

Fetching the LW Instruction

```
Time=275000: PC=0000005c, Instr=8c090010
```

- **At Time=275,000:**
  - **PC=0000005c:** The PC has moved to the load word (LW) instruction.
  - **Instr=8c090010:**
    - This encodes `lw $t1, 16($zero)`.
    - It loads the word from memory at address 16 into `$t1`. Since we previously stored 15 there, `$t1` should be 15.

- **Why It's Correct:**
  The instruction was placed at `imem[22]`, and the fetched encoding matches the LW operation.

Fetching the Last ADDI Instruction

```
Time=305000: PC=00000068, Instr=201600ff
```

- **At Time=305,000:**
  - **PC=00000068:** This instruction is the last in our programmed sequence.
  - **Instr=201600ff:**
    - This encodes `addi $s6, $zero, 0xFF` (setting `$s6` to 255 or 0xFF).
- **Why It's Correct:**
  The testbench placed this instruction at `imem[25]`. The encoding shows that the immediate value (0xFF) is correctly loaded into `$s6`.

The Rest of the Output – Filling with NOPs

```
Time=315000: PC=0000006c, Instr=00000000
...
Time=845000: PC=00000140, Instr=00000000
```

- **What It Means:**
  From this point onward, the PC keeps incrementing and the fetched instruction remains `00000000` (NOP).
- **Why It's Correct:**
  The remainder of the instruction memory was filled with NOPs by our testbench loop. This ensures no unintended instructions execute once our programmed sequence is complete.

Final Register/Memory State Report

```
Final Register/Memory State at ~350 ns:
$s0 =          5 (expected 5)
$s1 =         10 (expected 10)
$t0 =         15 (expected 15)
$s2 = 00000005 (expected 5)
$s3 = 000000fa (expected fa)
$s4 =          0 (expected 0)
```

```
$t1 =          15 (expected 15)
Mem[4] =          15 (expected 15)
$s6 = 000000ff (expected ff)
```

- **What It Means:**
  The testbench prints out the contents of specific registers and one memory location:
  - **$s0 (reg16):** 5, as set by the first ADDI instruction.
  - **$s1 (reg17):** 10, as set by the second ADDI.
  - **$t0 (reg8):** 15, the result of adding `$s1` and `$s0`.
  - **$s2 (reg18):** 5, result of `andi $s2, $s0, 0xF` (5 AND 15).
  - **$s3 (reg19):** 0xFA, result of `ori $s3, $s1, 0xF0` (10 OR 240).
  - **$s4 (reg20):** 0, result of `slti $s4, $s1, 10` (10 is not less than 10).
  - **$t1 (reg9):** 15, result of loading the word stored by SW.
  - **Mem[4]:** 15, the data memory location where `$t0` was stored.
  - **$s6 (reg22):** 0xFF, as set by the final ADDI.
- **Why It's Correct:**
  Each of these results is exactly what our instructions were designed to produce. The careful insertion of NOPs ensured that write-back occurred properly, so our final register and memory values match our expectations.

Simulation End Message

```
mips2_tb.v:94: $finish called at 850000 (1ps)
```

- **What It Means:**
  The simulation has reached the point where `$finish` is executed (line 94 in our testbench file), and the simulation terminates at time 850,000 time units.
- **Why It's Correct:**
  Our testbench was programmed to wait a sufficient amount of time (allowing all instructions to propagate through the pipeline) before calling `$finish`. This confirms that the entire instruction sequence was simulated, and our final states are valid.

Final Summary

- **Timing & Instruction Flow:**
  The output shows the PC advancing in 4-byte increments, and instructions are fetched in the correct order as we programmed them in `imem`. NOPs appear exactly where expected.

- **Register & Memory Updates:**
  The final state of the registers and memory confirms that the ari s
  thmetic, logic, load, and store operations have executed correctly. The outputs match the expected values derived from our instruction sequence.
- **Overall Correctness:**
  Every line of the simulation output aligns with our design's behavior. The control signals, data path operations, and pipeline delays (implemented via NOPs) all work together to produce the correct final state.

This detailed analysis proves that our processor design and testbench are functioning exactly as intended, with every instruction processed correctly and hazards avoided by our careful sequencing.