**INTRODUCTION AND OVERVIEW**

In this project, a MIPS32 RISC processor core with a conventional 5-stage pipeline (IF, ID, EX, MEM, and WB) that can handle the majority of integer instructions is created using Verilog. With pipeline registers, a complete network for transmitting control signals, and modules for the datapath and control path, the design is entirely new. Pipeline hazards are addressed by a stall mechanism when necessary, data-forwarding paths, and a specialised hazard detection unit. For the implementation to be reusable, expandable, and rigorously tested, it prioritises accuracy, modularity, and clarity.

# 

# 

# **OBJECTIVES OF THE PROJECT**

* **Architecture and Specifications:**
* **Define Instruction Set Architecture:** A defined subset of the MIPS32 integer instruction set.
* **Set Core Specifications:** Make the processor a 32-bit, little-endian machine that follows standard word alignment for memory accesses.
* **Define Memory Model:** Build a memory system with single-cycle interfaces for both data and instructions, and a method to deal with misaligned accesses.

**Making the design:**

* Build a **5-Stage Pipeline:** Design and build a classic 5-stage (IF, ID, EX, MEM, WB), single-issue, in-order execution microarchitecture in Verilog.
* **Build the Datapath:** Make all the parts of the datapath, such as the PC logic, the 32x32 register file, the ALU, the immediate generator, and all the multiplexers and routing that are needed.
* **Make Pipeline Registers:** Create the four pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB) that can hold data and control signals between stages and have the ability to flush and stall.
* **Develop the Control Path:** Create the main control decoder and the logic to send out all the control signals needed to carry out instructions.
* **Make a Hazard Control Unit:** Create a full hazard detection unit that uses data forwarding, pipeline stalls, and instruction flushes to deal with all data and control hazards.
* **Forwarding and Stalls**: Create and build a forwarding unit to handle data dependencies, and a stall unit to deal with load-use hazards by adding bubbles.
* **Verification:**
* **Set up a Verification Environment:** Make a full testbench architecture that can generate stimuli, check responses against a reference model, and automatically report whether the test passed or failed.
* **Make a Complete Test Suite:** Write a set of directed assembly-level test programs to make sure that all of the implemented instructions work correctly.
* **Achieve Functional Correctness:** Check the design carefully by looking at simulation waveforms and trace logs to make sure that all of the forwarding, stalling, and flushing mechanisms work as they should.

# 

# **DETAILED LIST OF COMPONENTS IN THE MIPS32 PROCESSOR**

* Top-level processor module (clock, reset, global buses, top-level I/O)
* 5 pipeline stages: IF, ID, EX, MEM, WB, organized for integer-only MIPS32 ISA execution

## **Pipeline registers and interstage control/data**

* IF/ID pipeline register: holds fetched instruction, PC+4, and ancillary bits (valid/flush)
* ID/EX pipeline register: register operands, sign-extended immediate, shamt, funct, control bundle for EX/MEM/WB stages
* EX/MEM pipeline register: ALU result, branch target, zero/compare flags, store data, control bundle for MEM/WB
* MEM/WB pipeline register: memory read data, ALU result, destination register id, WB control
* Per-register control: write-enable, flush, stall etc.

## **Instruction Fetch (IF) stage**

* Program Counter (PC) register with write-enable
* PC update MUX: selects next PC from PC+4, branch target, jump target
* PC incrementer (adder for PC+4)
* Instruction memory/read port (Harvard-style)
* Branch/jump feedback inputs from later stages
* IF stage control: always-read instruction memory, implicit PC write each cycle (unless stalled/flushed)
* IF hazard/flush controls interfacing (from branch resolution and hazard units)

## **Instruction Decode (ID) stage**

* Instruction field extraction: opcode(6), rs(5), rt(5), rd(5), shamt(5), funct(6), imm(16), target(26)
* Main control unit (decode): generates RegDst, RegWrite, ALUSrc, ALUOp, Branch, MemRead, MemWrite, MemToReg, Jump, and other related control signals. It also generated ALU control signals and operands from opcode and function fields in the instruction
* Register file: 32×32 integer registers, two asynchronous (to model single cycle simultaneous synchronous read) read ports (rs, rt), one synchronous write port (rd/rt), with write enable and write-back data path

REGISTERS IN MIPS32

| Register Number | Register Name | Purpose / Function | Speciality / Notes |
| --- | --- | --- | --- |
| 0 | $zero | Constant zero | Always holds 0, writes ignored |
| 1 | $at | Assembler temporary | Reserved for assembler use |
| 2-3 | $v0-$v1 | Function return values | Used to store return values from functions or for expr evaluation |
| 4-7 | $a0-$a3 | Function arguments | Used to pass the first 4 arguments to functions |
| 8-15 | $t0-$t7 | Temporary registers | Caller-saved, not preserved across calls |
| 16-23 | $s0-$s7 | Saved registers | Callee-saved, preserved across calls |
| 24-25 | $t8-$t9 | Temporary registers | Additional temps, caller-saved |
| 26-27 | $k0-$k1 | Reserved for OS kernel | Used by OS |
| 28 | $gp | Global pointer | Points to the middle of the 64KB data block |
| 29 | $sp | Stack pointer | Points to the last used location on the stack |
| 30 | $fp | Frame pointer | Used to reference the function frame |
| 31 | $ra | Return address | Holds return address for jump-and-link and branch-and-link instructions |

**Special-purpose registers (not part of the 32 general-purpose registers)**

* **Program Counter (PC)**
* Sign extender (16→32) for immediates
* Branch address generation (PC+4 + sign-extended immediate<<2) path inputs forwarded to EX
* Jump target formation support (upper PC bits concatenated with target field<<2)
* Hazard-controlled ID control signal zeroing for bubble insertion
* ID outputs packaged into ID/EX (including control bits destined for EX/MEM/WB)

## **Execute (EX) stage**

* ALU: 32-bit integer unit supporting ADD/SUB, AND/OR/XOR/NOR, shifts (sll/srl/sra), comparisons/zero flag
* Operand A MUX: selects between register operand, forwarded EX/MEM/MEM/WB data
* Operand B MUX: selects between register operand, sign-extended immediate, or forwarded data
* Destination register MUX
* Comparison logic/Zero flag for branches (equality/inequality)
* EX outputs to EX/MEM register: ALU result, write-data (rt) for stores, branch-taken condition, branch target

## **Memory Access (MEM) stage**

* Data memory interface: address, read data, write data, MemRead, MemWrite
* Store data selection
* Branch resolution and PCSrc computation (Branch AND condition) with flush request
* MEM outputs to MEM/WB: read data (for loads), ALU result (for ALU ops), destination register id, WB control

## **Write Back (WB) stage**

* Write-back MUX (MemToReg: memory data vs ALU result)
* Register file write port logic: destination register selection and RegWrite
* Timing alignment so WB occurs for the correct instruction each cycle

## **Hazard detection and resolution**

* Hazard Detection Unit:
  + Detects load-use RAW hazards (ID using result of a load in EX/MEM), asserts stall(s), and inserts bubble into EX
  + Generates control to freeze PC and IF/ID, and zero ID/EX control bundle when needed
* Forwarding (Bypass) Unit:
  + Compares EX stage source regs vs EX/MEM and MEM/WB destination regs
  + Selects forwarding sources for ALU operands (EX/MEM ALU result, MEM/WB write-back data)
* Control Hazard handling:
  + Branch decision in EX/MEM and PCSrc generation
  + Flush logic for IF/ID (and optionally ID/EX) on the taken branch/jump

## **Control path (distributed and pipelined)**

* Main instruction decoder (ID stage)
* Pipelined control bundles carried in ID/EX, EX/MEM, MEM/WB
* Branch/jump control generation and routing back to IF
* Stall/flush controller integrating hazard, branch decision, etc.

## **Memory system assumptions (integer-only)**

* Separate instruction memory/read port for IF
* Data memory with byte/half-word/word addressing as required for loads/stores

**DESCRIPTION OF INSTRUCTION SET ARCHITECTURE**

**R-type (Register-type) arithmetic and logical instructions**

All R-type instructions have the same OP code of 6'b000000 as per MIPS convention, and the FUNCTION code differentiates instructions internally. The ALU control code is used to control the ALU operation accordingly in hardware implementation.

| Mnemonic | FUNCTION Code (6-bit) | OP Code (6-bit) | ALU Control Code (5-bit) | Description |
| --- | --- | --- | --- | --- |
| ADD | 100000 | 000000 | 00100 | Signed Add |
| ADDU | 100001 | 000000 | 00101 | Unsigned Add |
| SUB | 100010 | 000000 | 00110 | Signed Subtract |
| SUBU | 100011 | 000000 | 00111 | Unsigned Subtract |
| AND | 100100 | 000000 | 10000 | Bitwise AND |
| OR | 100101 | 000000 | 00001 | Bitwise OR |
| XOR | 100110 | 000000 | 00010 | Bitwise XOR |
| NOR | 100111 | 000000 | 00011 | Bitwise NOR |
| SLT | 101010 | 000000 | 01000 | Set on Less Than (signed) |
| SLTU | 101011 | 000000 | 01001 | Set on Less Than (unsigned) |
| SLL | 000000 | 000000 | 01010 | Shift Left Logical |
| SLLV | 000100 | 000000 | 01101 | Shift Left Logical Variable |
| SRL | 000010 | 000000 | 01011 | Shift Right Logical |
| SRLV | 000110 | 000000 | 01110 | Shift Right Logical Variable |
| SRA | 000011 | 000000 | 01100 | Shift Right Arithmetic |
| SRAV | 000111 | 000000 | 01111 | Shift Right Arithmetic Variable |

**ALU-BASED I–TYPE INSTRUCTIONS with UNIQUE OPCODES**

| Mnemonic | OP Code (6-bit) | Description |
| --- | --- | --- |
| ADDI | 001000 | Add Immediate (signed) |
| ADDIU | 001001 | Add Immediate Unsigned |
| SLTI | 001010 | Set Less Than Immediate (signed) |
| SLTIU | 001011 | Set Less Than Immediate Unsigned |
| ANDI | 001100 | Bitwise AND Immediate |
| ORI | 001101 | Bitwise OR Immediate |
| XORI | 001110 | Bitwise XOR Immediate |
| LUI | 001111 | Load Upper Immediate |

**LW/SW BASED INSTRUCTIONS with UNIQUE OPCODES**

| Mnemonic | OP Code (6-bit) | Description |
| --- | --- | --- |
| LB | 100000 | Load Byte |
| LBU | 100100 | Load Byte Unsigned |
| LH | 100001 | Load Halfword |
| LHU | 100101 | Load Halfword Unsigned |
| LW | 100011 | Load Word |
| SB | 101000 | Store Byte |
| SH | 101001 | Store Halfword |
| SW | 101011 | Store Word |

**JUMP/BRANCH TYPE INSTRUCTIONS with UNIQUE OPCODES**

| Mnemonic | Opcode (6-bit) | Description |
| --- | --- | --- |
| BEQ | 000100 | Branch on Equal (PC-relative, compares two registers) |
| BNE | 000101 | Branch on Not Equal (PC-relative, compares two registers) |
| J | 000010 | Jump (Unconditional jump within a 256 MB region) |
| JAL | 000011 | Jump and Link (Unconditional jump and store return address) |

**32-bit MACHINE CODE ENCODING**

**R-Type Instruction Format**

R-type instructions are used for register-to-register operations and have the following encoding:

| Bit Range | Field | Description |
| --- | --- | --- |
| 31–26 | opcode | 6-bit operation code (always 000000 for standard R-type) |
| 25–21 | rs | Source register 1 |
| 20–16 | rt | Source register 2 |
| 15–11 | rd | Destination register |
| 10–6 | shamt | Shift amount (used for shift instructions) |
| 5–0 | funct | Function code (specific operation) |

Example: Instruction: add $s1, $s2, $s3

Encoding fields: opcode = 000000, rs = 10010, rt = 10011, rd = 10001, shamt = 00000, funct = 100000

**I-Type Instruction Format**

I-type instructions perform operations with immediate values or for memory access:

| Bit Range | Field | Description |
| --- | --- | --- |
| 31–26 | opcode | 6-bit operation code |
| 25–21 | rs | Source register |
| 20–16 | rt | Destination register |
| 15–0 | Immediate Data | 16-bit immediate value |

Example: Instruction: lw $s1, 48($s1)

Encoding fields: opcode = 100011, rs = 10001, rt = 10001, immediate = 0000000000110000

**J-Type Instruction Format**

J-type instructions are used for jump operations:

| Bit Range | Field | Description |
| --- | --- | --- |
| 31–26 | opcode | 6-bit operation code |
| 25–0 | Immediate Data | 26-bit jump address (extended to 28 bits by appending two zeros) |

Example: Instruction: j Label

Encoding fields: opcode (6 bits), immediate (26-bit address)

**DESCRIPTION OF IMPORTANT CONTROL SIGNALS**

| Signal Name | Width | Description |
| --- | --- | --- |
| RegDst | 1 bit | Selects the **destination register address** for the write-back operation.  • 1: For R-type, the destination is the rd field (instr[15:11])  • 0: For I-type (lw, addi), the destination is the rt field (instr[20:16]). |
| PC\_source | 2 bit | 00: PC + 4 selected |
| ALUSrc | 1 bit | Selects the **second operand for the ALU**  1: The operand is the sign-extended 16-bit immediate  0: The operand is from the register file (ReadData2). |
| MemtoReg | 2 bit | Selects the **data source for the write-back**  01: Data comes from the Data Memory (for lw)  00: Data comes from the ALU result (for R-type/addi). |
| RegWrite | 1 bit | Enables the write operation on the register file in the WB stage  Asserted for lw, R-type, addi, jal. |
| MemRead | 1 bit | Enables the read operation on the Data Memory in the MEM stage. Asserted for lw. |
| MemWrite | 1 bit | Enables the write operation on the Data Memory in the MEM stage. Asserted for sw. |
| Branch | 1 bit | Asserts if the instruction is a conditional branch (beq, bne)  Used by the Hazard Unit and PC logic. |
| Jump | 2 bits | (Jump Control) Encodes the type of unconditional jump  • 2'b01: j instruction.  • 2'b10: jal instruction.  • 2'b11: jr instruction.  • 2'b00: Not a jump. |
| ALUControl | 5 bits | The final, specific 5-bit control signal sent directly to the ALU in the EX stage. |

**OUTPUT VERIFICATION**

**ALU-BASED INSTRUCTIONS WITHOUT HAZARD**

| Hex code | Mnemonic |
| --- | --- |
| 32'h02114020 | ADD $t0, $s0, $s1 |
| 32'h02534821 | ADDU $t1, $s2, $s3 |
| 32'h02945022 | SUB $t2, $s4, $s5 |
| 32'h02D75823 | SUBU $t3, $s6, $s7 |
| 32'h00856024 | AND $t4, $a0, $a1 |
| 32'h00C76825 | OR $t5, $a2, $a3 |
| 32'h00437026 | XOR $t6, $v0, $v1 |
| 32'h035B7827 | NOR $t7, $k0, $k1 |
| 32'h031EC02A | SLT $t8, $t8, $fp |
| 32'h03BFC82B | SLTU $t9, $sp, $ra |
| 32'h00008040 | SLL $s0, $zero, 1 |
| 32'h00018882 | SRL $s1, $at, 2 |
| 32'h00139083 | SRA $s2, $s3, 2 |
| 32'h02D5A004 | SLLV $s4, $s5, $s6 |
| 32'h0128B806 | SRLV $s7, $t0, $t1 |
| 32'h016A2007 | SRAV $a0, $t2, $t3 |
| 32'h21850064 | ADDI $a1, $t4, 100 |
| 32'h25A600C8 | ADDIU $a2, $t5, 200 |
| 32'h29C70032 | SLTI $a3, $t6, 50 |
| 32'h2DE2004B | SLTIU $v0, $t7, 75 |
| 32'h330300FF | ANDI $v1, $t8, 255 |
| 32'h373A00F0 | ORI $k0, $t9, 240 |
| 32'h3A1B00AA | XORI $k1, $s0, 170 |
| 32'h3C1C1000 | LUI $gp, 4096 |

**ALU INSTRUCTION TEST WITH HAZARD**

| Hex code | Mnemonic |
| --- | --- |
| 32'h02114020 | ADD $t0, $s0, $s1 |
| 32'h01134821 | ADDU $t1, $t0, $s3 |
| 32'h02945022 | SUB $t2, $s4, $s5 |
| 32'h01575823 | SUBU $t3, $t2, $s7 |
| 32'h00018882 | SRL $s1, $at, 2 |
| 32'h00139083 | SRA $s2, $s3, 2 |
| 32'h02D5A004 | SLLV $s4, $s5, $s6 |

**LOAD INSTRUCTION TEST**

| Hex instruction | Mnemonic |
| --- | --- |
| 32'h81860004 | LB $a2, 4($t4) |
| 32'h85870006 | LH $a3, 6($t4) |
| 32'h8D840010 | LW $a4, 16($t4) |
| 32'h9149000A | LBU $t1, 10($t2) |
| 32'h9550000C | LHU $s0, 12($t2) |

**STORE INSTRUCTION TEST**

| Hex instruction | Mnemonic |
| --- | --- |
| 32'hA1850018 | SB $a1, 24($t4) |
| 32'hA586001A | SH $a2, 26($t4) |
| 32'hAD87001C | SW $a3, 28($t4) |

**CONTROL INSTRUCTION TEST**

| Hex instruction | Mnemonic |
| --- | --- |
| 32'h02114020 | ADD $t0, $s0, $s1 |
| 32'h02534821 | ADDU $t1, $s2, $s3 |
| 32'h11280004 | BEQ $t1, $t2, +2 |
| 32'h152A0002 | BNE $t1, $t2, +2 |
| 32'h02945022 | SUB $t2, $s4, $s5 |
| 32'h02D75823 | SUBU $t3, $s6, $s7 |
| 32'h00856024 | AND $t4, $a0, $a1 |
| 32'h00C76825 | OR $t5, $a2, $a3 |
| 32'h00437026 | XOR $t6, $v0, $v1 |
| 32'h035B7827 | NOR $t7, $k0, $k1 |
| 32'h031EC02A | SLT $t8, $t8, $fp |
| 32'h08000001 | J <LABEL> |
| 32'h0C000001 | JAL <LABEL> |