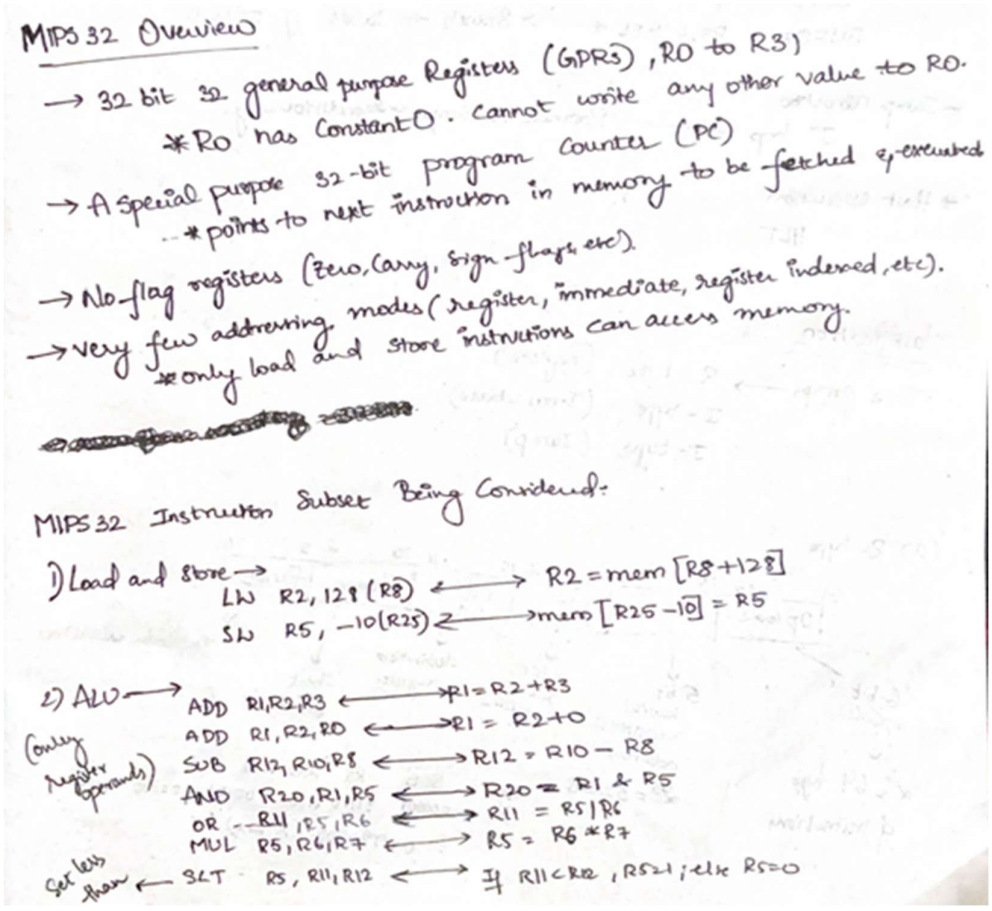
SINGLE CYCLE MIPS32 PROCESSOR

BY
YANDARAPU JAYA KUSHAL
GITHUB: jayakushal24

*This project was undertaken as a **personal learning initiative** to deepen my understanding of processor architecture, digital design, and the MIPS32 instruction set. The goal was to implement a functional **single-cycle processor** entirely in Verilog, simulating core components such as instruction decoding, register file handling, memory operations, and ALU computations.*

Through this project, I successfully developed a working MIPS32 processor capable of executing a variety of basic R-type and I-type instructions. The processor was verified using simulation waveforms and test programs written in hexadecimal format. This hands-on experience significantly enhanced my practical knowledge of computer architecture, RTL design, and the functional flow of instruction execution in a CPU.

OVERVIEW



MIPS 32 Overview

→ 32 bit 32 general purpose Registers (GPRS), R0 to R31)
  * R0 has Constant 0. Cannot write any other value to R0.

→ A special purpose 32-bit program counter (PC)
  * points to next instruction in memory to be fetched & executed

→ No flag registers (zero, carry, sign-flags, etc).

→ very few addressing modes (register, immediate, register indexed, etc).
  * only load and store instructions can access memory.

MIPS 32 Instruction Subset Being Considered:

1) Load and Store →
   LW R2, 128(R8) ⟷ R2 = mem[R8+128]
   SW R5, -10(R25) ⟷ mem[R25-10] = R5

2) ALU → (only register operands)
   ADD R1,R2,R3 ⟷ R1 = R2+R3
   ADD R1,R2,R0 ⟷ R1 = R2+0
   SUB R12,R10,R8 ⟷ R12 = R10 − R8
   AND R20,R1,R5 ⟷ R20 = R1 & R5
   OR R11,R5,R6 ⟷ R11 = R5|R6
   MUL R5,R6,R7 ⟷ R5 = R6 * R7
   Set less than ← SLT R5, R11, R12 ⟷ If R11<R12, R5=1; else R5=0

* ALU (immediate operand.

   ADDI R1, R2, 25 ⟷ R1 = R2 +25
   SUBI R5, R1, 150 ⟷ R5 = R1 − 150
   SLTI R2, R10, 10 ⟷ If R10<10, R2=1, else R2=0;

*Branch instructions.

   BEQZ R1, LOOP ⟷ Branch to loop if R1=0
   BNEQZ R5, LABEL ⟷ Branch to label if R5!=0

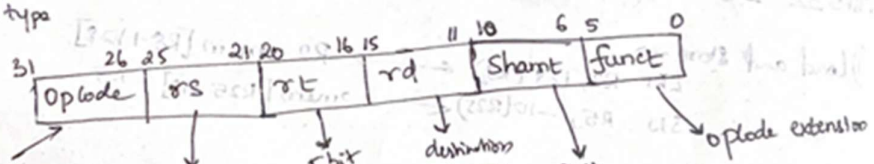
* Jump instruction
   J loop ⟷ Branch to loop unconditionally.

* Halt execution
   HLT

Classification

Three groups → R-type (register)
I-type (Immediate)
J-type (Jump)

(a) R-type

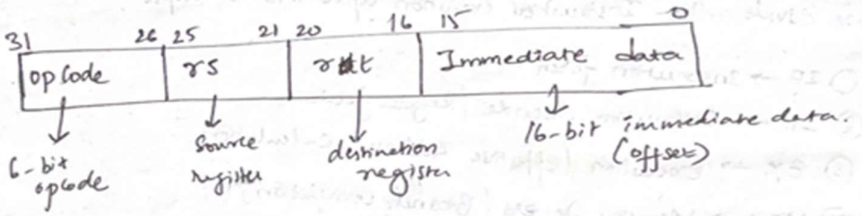


2^6 = 64 type of instructions.

2^5 = 32 bits

eg → SUB R5 R12 R25
000001 01100 11001 00101 00000 000000
= 059932800 (in hex)

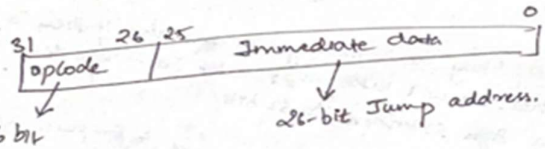
(b) I type



LW (load), SW (store), ADDI, SUBI, SLTI, BNEQ, BEQZ.

(opcode can be chosen).

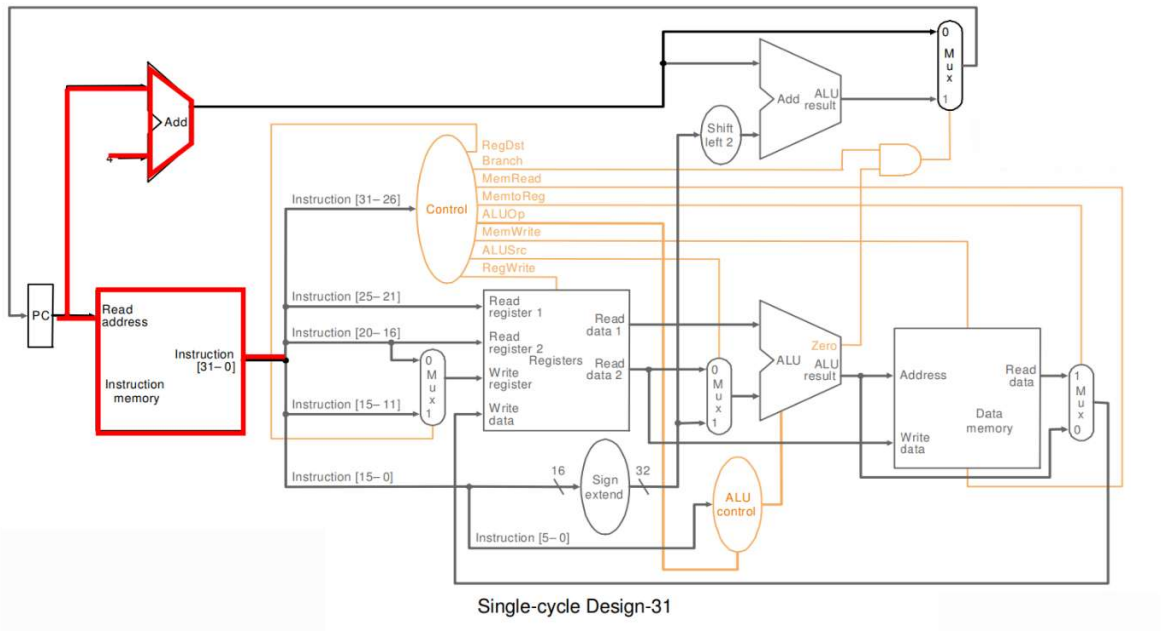
(c) J type



*Only after decoding the instruction, we know if both registers rs & rt are req. or just rs.

So, while decoding is going on, we can prefetch the registers in parallel.

MIPS32 ARCHITECTURE



Single-cycle Design-31

Source: https://www.cs.nthu.edu.tw/~king/courses/cs4100/P51-proc.pdf

NOTE: Code was written based on this architecture

ARCHITECTURE FUNCTION

1. Program Counter (PC)

- **Function:** Holds the address of the current instruction
- **Operation:** Updates every cycle to point to the next instruction address
- **Input:** New PC value (either PC + 4 or branch target)
- **Output:** Instruction memory read address

2. Instruction Memory

- **Function:** Stores all program instructions.
- **Operation:** Outputs the 32-bit instruction located at the address from the PC
- **Input:** 32-bit read address
- **Output:** 32-bit instruction

3. Instruction Fields

- **Function:** Breaks down the 32-bit instruction into individual fields for decoding and execution
- **Fields information:**
    - **Opcode (bits 31–26):** Specifies the instruction type (R-type, I-type, or J-type). This field determines the general operation category
    - **rs (bits 25–21):** Source register 1. This register supplies the first operand for the instruction
    - **rt (bits 20–16):** Source register 2 for R-type instructions or destination register for I-type instructions. For example, in an I-type load/store, this specifies the register involved
    - **rd (bits 15–11):** Destination register for R-type instructions. This is where the result of the computation is stored
    - **shamt (bits 10–6):** Shift amount field. Used **only** in shift instructions (like sll, srl, sra) to specify how many bits positions the bits in a register should be shifted. It is a 5-bit unsigned value allowing shifts from 0 to 31 bits
    - **funct (bits 5–0):** Function field. Used **only** in R-type instructions to specify the exact ALU operation to perform (such as add, subtract, AND, OR, set on less than, etc.). Since the opcode for all R-type instructions is the same, the funct field differentiates among various R-type operations
-

4. Control Unit

- **Function:** Generates control signals based on the opcode. //decides if the opcode is for R type /I type /Lw/ Sw
- **Operation:** Decides what each datapath element should do for the current instruction
- **Input:** Opcode (Instruction [31:26]).
- **Output:** Control signals:
    - RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite

5. Register memory

- **Function:** Stores all 32 general-purpose registers
- **Operation:**
    - Reads two source registers
    - Writes one destination register if RegWrite is enabled
- **Inputs:**
    - Read register 1, Read register 2
    - Write register, Write data, and RegWrite
- **Outputs:**
    - Read data 1, Read data 2

6. Sign-Extend module

- **Function:** Extends 16-bit immediate to 32-bit signed value
- **Input:** Instruction [15:0] (immediate)
- **Output:** 32-bit sign-extended immediate

7. ALU Control

- **Function:** Generates specific ALU operation code
- **Input:** ALUOp (from Control Unit) and function field (Instruction [5:0])
- **Output:** 4-bit control signal to ALU

8. ALU

- **Function:** Performs arithmetic and logic operations
- **Inputs:**
  - Operand A: Read data 1
  - Operand B: Either Read data 2 or sign-extended immediate (based on ALUSrc)
  - Control signal from ALU Control
- **Output:**
  - Result of the operation
  - Zero flag (used for branches)

9. Data Memory

- **Function:** Stores and retrieves data (for load/store instructions)
- **Inputs:**
  - Address (from ALU)
  - Write data (from register file)
  - Control: MemRead, MemWrite
- **Output:** Read data (used in load instructions)

10. MUX (Multiplexers)

- **MUX 1 (RegDst):** Chooses between rt and rd as destination register
- **MUX 2 (ALUSrc):** Chooses second ALU operand between Read data 2 and sign-extended immediate
- **MUX 3 (MemtoReg):** Chooses between ALU result and Data Memory output for writing back to register
- **MUX 4 (PCSrc):** Chooses next PC value: PC + 4 or branch address

11. Shift Left 2

- **Function:** Shifts the sign-extended immediate left by 2 bits (×4), used for branch address calculation
- **Input:** Sign-extended immediate
- **Output:** Shifted value to be added to PC + 4

12. Adder (PC + 4 and Branch Target)

- **Adder 1:** Computes PC + 4.
- **Adder 2:** Computes branch target address (PC + 4 + shifted immediate).

13. Branch Logic

- **Function:** Determines if a branch should be taken.
- **Inputs:**
  - Zero flag from ALU.
  - Branch control signal.
- **Output:** Enables MUX 4 to select branch address if condition is met.

CODE:

PC Block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/program_counter.v

```
1  `timescale 1ns / 1ps
2
3  module program_counter(
4      input clk,
5      input reset,
6      input [31:0]pc,
7      output reg [31:0]next_pc
8  );
9      always @(posedge clk or posedge reset) begin
10         if (reset)
11             next_pc<=32'b0;
12         else
13             next_pc<=pc;//pc is not directly incremented..as there might be branch and jump instructions
14     end
15 endmodule
```

PC_adder Block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/pc_add.v

```
1  `timescale 1ns / 1ps
2
3
4  module pc_add(
5  input [31:0]pc_in,
6  output [31:0]pc_out
7      );
8      assign pc_out =  pc_in+4;
9  endmodule
10
```

MUX block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/mux2x1.v

```
1  `timescale 1ns / 1ps
2
3  ///used between registers memory(after) and ALU(before)
4
5  ///used between data memory (after) and registers memory(before)
6
7  module mux2x1 (
8      input [31:0]in1,
9      input [31:0]in0,
10     input sel,
11     output [31:0]out
12 );
13
14     assign out= sel? in1:in0;
15
16 endmodule
17
18
19 //used between  instruction memroy(after) and register memory(before)
20 module mux2x1_5b(
21     input [4:0]in1,
22     input [4:0]in0,
23     input sel,
24     output [4:0]out
25 );
26
27     assign out= sel? in1:in0;
28
29 endmodule
30
```

Instruction memory Block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/instruction_memory.v

```
1  `timescale 1ns / 1ps
2
3  module instruction_memory(
4      input clk, reset,
5      input [31:0]read_address,
6      output reg [31:0]instruction
7  );
8
9      reg [31:0]mem[63:0];
10     integer i;
11
12     initial begin
13         for (i=0; i<64; i=i+1)
14             mem[i] = 32'h00000000;
15
16         mem[0]  = 32'h20080005; // addi $t0, $zero, 4
17         mem[1]  = 32'h20090003; // addi $t1, $zero, 3
18         mem[2]  = 32'h01095020; // add $t2, $t0, $t1
19         mem[3]  = 32'h012A5822; // sub $t3, $t1, $t2
20         mem[4]  = 32'hAC0A0004; // sw $t2, 4($zero)
21         mem[5]  = 32'h8C0C0004; // lw $t4, 4($zero)
22         mem[6]  = 32'h110C0002; // beq $t0, $t4, label (2 offset)///branching inst branch=1 zero =0..next inst not skipped
23         mem[7]  = 32'h200D0001; // addi $t5, $zero, 1
24         mem[8]  = 32'h200D0002; // addi $t5, $zero, 2
25         mem[9]  = 32'h11AD0001; // beq $t5, $t5, label (1 offset)///branching branch=1 and zero=1
26         mem[10] = 32'h200E00FF; // addi $t6, $zero, 255/////////////////Skipped Instruction
27         mem[11] = 32'h200E0001; // addi $t6, $zero, 1
28         mem[12] = 32'h200E0002; // addi $t6, $zero, 2
29         mem[13] = 32'h200E0003; // addi $t6, $zero, 3
30     end
31
32     always @(*) begin
33         instruction=mem[read_address>>2]; //might have to divide  by 4 here...depending onn value initialization
34     end
35
36 endmodule
37
```

Register memory Block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/registerts.v

```
1  `timescale 1ns / 1ps
2
3
4  module registerts(
5  input clk,
6  input [31:0]write_data,
7  input [4:0]write_register,read_register1,read_register2,
8  input reg_write,
9  output [31:0]read_data1,read_data2
10     );
11
12  reg [31:0] register_mem[31:0];
13
14  integer i;
15     initial begin
16         for (i=0; i<32; i=i+1)
17             register_mem[i] = 32'd0;
18     end
19  always @(posedge clk)begin
20
21      if (reg_write && write_register!=0)begin //1st register is 0(fixed in MIPS)
22         register_mem[write_register]=write_data;
23      end
24   end
25      assign read_data1=register_mem[read_register1];
26      assign read_data2=register_mem[read_register2];
27
28
29 endmodule
30
```

Sign extend block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/sign_extend.v

```
1  `timescale 1ns / 1ps
2
3
4  module sign_extend(
5  input [15:0]in,
6  output [31:0]out
7      );
8
9      assign out ={{16{in[15]}},in};
10 endmodule
11
```

CONTROL BLOCK

D:/Vivado_projects/M32/MIPS32/MIPS32.srcs/sources_1/new/control.v

```
1  `timescale 1ns / 1ps
2
3  module control(
4      input [5:0]opcode,
5      output reg RegWrite,
6      output reg MemRead,
7      output reg MemWrite,
8      output reg MemToReg,
9      output reg ALUSrc,
10     output reg Branch,
11     output reg RegDst,
12     output reg [1:0] ALUOp
13 );
14
15     always @(*) begin
16         case (opcode)
17             6'b000000: //R type
18             begin
19                 RegDst= 1'b1;ALUSrc= 1'b0;MemToReg= 1'b0;RegWrite= 1'b1;
20                 MemRead= 1'b0;MemWrite= 1'b0;Branch= 1'b0;ALUOp= 2'b10;
21             end
22
23             6'b100011: // lw
24             begin
25                 RegDst= 1'b0;ALUSrc= 1'b1;MemToReg= 1'b1;RegWrite= 1'b1;
26                 MemRead= 1'b1;MemWrite= 1'b0;Branch= 1'b0;ALUOp= 2'b00;
27             end
28
29             6'b101011: // sw
30             begin
31                 RegDst= 1'bx;ALUSrc= 1'b1;MemToReg= 1'bx;RegWrite= 1'b0;
32                 MemRead= 1'b0;MemWrite= 1'b1;Branch= 1'b0;ALUOp= 2'b00;  // ALU performs add to cal address
33             end
34
35             6'b000100: // beq
36             begin
37                 RegDst= 1'bx;ALUSrc= 1'b0;MemToReg= 1'bx;RegWrite= 1'b0;
38                 MemRead= 1'b0;MemWrite= 1'b0;Branch= 1'b1;ALUOp= 2'b01;
39             end
40
41             6'b001000: // addi (I-type ALU immediate)
42             begin
43                 RegDst= 1'b0;ALUSrc= 1'b1;MemToReg= 1'b0;RegWrite= 1'b1;
44                 MemRead= 1'b0;MemWrite= 1'b0;Branch= 1'b0;ALUOp= 2'b00;  //ADD
45             end
46
47             default:
48             begin
49                 RegDst= 1'b0;ALUSrc= 1'b0;MemToReg= 1'b0;RegWrite= 1'b0;
50                 MemRead= 1'b0;MemWrite= 1'b0;Branch= 1'b0;ALUOp= 2'b00;
51             end
52         endcase
53     end
54
55
56
57 endmodule
58
```

ALU_CONTROL BLOCK

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/ALU_control.v

```
1  `timescale 1ns / 1ps
2
3  module ALUcontrol(
4      input [1:0] ALUOp,           //output from Control
5      input [5:0] funct,           //from instruction[5:0] for r type
6      output reg [3:0] operation
7  );
8
9      always @(*) begin
10         case (ALUOp)
11             2'b00: operation = 4'b0000; //lw,sw,addi..add
12             2'b01: operation = 4'b0001; //beq => SUBTRACT
13
14             2'b10: begin  //for r type ...decode funct field
15                 case (funct)
16                     6'b100000: operation = 4'b0000; //add
17                     6'b100010: operation = 4'b0001; //sub
18                     6'b100100: operation = 4'b0010; //and
19                     6'b100101: operation = 4'b0011; //or
20                     6'b101010: operation = 4'b0111; //slt
21                     6'b100110: operation = 4'b0100; //xor
22                     6'b000000: operation = 4'b0101; //Shift left logical
23                     6'b000010: operation = 4'b0110; //shift right logical
24                     default:   operation = 4'b0000;
25                 endcase
26             end
27
28             default: operation = 4'b0000;
29         endcase
30     end
31
32 endmodule
```

ALU BLOCK

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/ALU.v

```
1  `timescale 1ns / 1ps
2
3  module ALU(
4      input [31:0] A,
5      input [31:0] B,
6      input [3:0] operation,
7      output reg [31:0] Result,
8      output reg Zero
9  );
10
11     always @(A or B or operation) begin
12         case (operation)
13             4'b0000: Result = A + B;
14             4'b0001: Result = A - B;
15             4'b0010: Result = A & B;   //and
16             4'b0011: Result = A | B; //or
17             4'b0100: Result = A ^ B;  //xor
18             4'b0101: Result = B << A[4:0];  //logical shift left
19             4'b0110: Result = B >> A[4:0];  //logical shift right
20             4'b0111: Result = (A < B) ? 32'd1 : 32'd0;
21             default: Result = 32'b0;
22         endcase
23
24         Zero = (Result == 32'b0) ? 1 : 0;//to be used for branching op
25     end
26
27 endmodule
28
```

Data Memory BLOCK

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/data_memory.v

```
1  `timescale 1ns / 1ps
2
3
4  module data_memory(
5  input clk,
6      input rst,
7      input MemRead,
8      input MemWrite,
9      input [31:0]address,
10     input [31:0]write_data,
11     output[31:0]read_data
12 );
13   reg [31:0]data_memory[31:0];
14
15   integer i;
16   assign read_data=(MemRead)? data_memory[address>>2]:32'b00;
17
18
19   always @(posedge clk or posedge rst) begin
20     if (rst ) begin
21       for (i=0; i<32; i=i+1) begin
22         data_memory[i] = 32'b00;
23       end
24     end else if (MemWrite) begin
25       data_memory[address>>2]=write_data;
26     end
27   end
28
29 endmodule
30
```

Adder Branch Block

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sources_1/new/adder_branch.v

```
1  `timescale 1ns / 1ps
2
3
4  module adder_branch(
5  input [31:0]pc,
6      input [31:0]branch_jump,
7      output [31:0]destination
8  );
9
10 assign destination = pc+ branch_jump;
11
12 endmodule
```

TOP MODULE

D:/Vivado_projects/M32/MIPS32/MIPS32.srcs/sources_1/new/MIPS32_TOP.v

```
`timescale 1ns / 1ps

module MIPS32_TOP(
    input clk,
    input rst
);

// Wires
wire [31:0] pc_wire,pc_out_wire,pc_next_wire,decode_wire,read_data1,read_data2,reg_mux_alu_out,sign_extension_out,ALU_result,read_data,write_data,Adder_result;
wire [4:0]write_reg_mux;
wire [1:0]ALUOp;
wire [3:0]operation;
wire RegDst,RegWrite,ALUSrc,MemRead,MemWrite, MemToReg,Branch,Zero;

////////////////////////////////////Program Counter
program_counter PC(
    .clk(clk),.reset(rst),.pc(pc_wire),.next_pc(pc_out_wire)
);

////////////////////////////////PC Adder
pc_add PC_Adder(
    .pc_in(pc_out_wire),.pc_out(pc_next_wire)
);

////////////////////////////////PC mux
mux2x1 pc_mux(
    .in0(pc_next_wire),.in1(Adder_result),.sel(Branch & Zero),.out(pc_wire)
);

////////////////////////////////instruction memory
instruction_memory Instr_Mem(
    .reset(rst),.clk(clk),.read_address(pc_out_wire),.instruction(decode_wire)
);

////////////////////////////////Write register mux
mux2x1_5b wr_reg_mux(
    .in0(decode_wire[20:16]),.in1(decode_wire[15:11]),.sel(RegDst),.out(write_reg_mux)
);

////////////////////////////////Register memory
registerts Reg_mem(
    .clk(clk),.reg_write(RegWrite),.write_data(write_data),.write_register(write_reg_mux),.read_register1(decode_wire[25:21]),
    .read_register2(decode_wire[20:16]),.read_data1(read_data1),.read_data2(read_data2)
);

////////////////////////////////Sign extend
sign_extend sign_ext(
    .in(decode_wire[15:0]),.out(sign_extension_out)
);

////////////////////////////ALU input mux
mux2x1 reg_mux_alu(
    .in0(read_data2),.in1(sign_extension_out),.sel(ALUSrc),.out(reg_mux_alu_out)
);

//////////////////////////// Control unit
control control(
    .opcode(decode_wire[31:26]),.RegWrite(RegWrite),.MemRead(MemRead),.MemWrite(MemWrite),
    .MemToReg(MemToReg),.ALUSrc(ALUSrc),.Branch(Branch),.RegDst(RegDst),.ALUOp(ALUOp)
);

//////////////////////////// ALU control
ALUControl alu_control(
    .funct(decode_wire[5:0]),.ALUOp(ALUOp),.operation(operation)
);

//////////////////////////// ALU
ALU alu_block(
    .A(read_data1),.B(reg_mux_alu_out), .operation(operation),.Result(ALU_result),.Zero(Zero)
);

//////////////////////// data memory
data_memory dat_mem(
    .clk(clk),.rst(rst),.MemRead(MemRead),.MemWrite(MemWrite),.address(ALU_result),
    .write_data(read_data2),.read_data(read_data)
);

////////////////////Mem to Reg Mux
mux2x1 datmem_mux_regmem(
    .in0(ALU_result),.in1(read_data),
    .sel(MemToReg),.out(write_data)
);

////////////////////branch address adder
adder_branch adder(
    .pc(pc_next_wire),.branch_jump(sign_extension_out << 2),.destination(Adder_result)
);

endmodule
```

VERIFICATION

TESTBENCH

D:/Vivado_projects/Project_M32/MIPS32/MIPS32.srcs/sim_1/new/MIPS32_tb.v

```
1  `timescale 1ns / 1ps
2
3
4  module MIPS32_tb();
5
6      reg clk, rst;
7      MIPS32_TOP DUT (.clk(clk), .rst(rst));
8
9      initial begin
10         clk = 0;
11     end
12     always #25 clk = ~clk; //clk gen
13
14     initial begin
15         rst = 1'b1;
16         #50;
17         rst = 1'b0;
18         #2000;
19         $finish;
20     end
21
22 endmodule
```

TABULATION:

PC (HEX)	INSTRUCTION	OPERATION DESCRIPTION	RESULT/VALUE STORED
0X00	addi $t0, $zero, 4	Load 4 into $t0	$t0 = 4
0X04	addi $t1, $zero, 3	Load 3 into $t1	$t1 = 3
0X08	add $t2, $t0, $t1	Add $t0 + $t1 → $t2	$t2 = 4 + 3 = 7
0X0C	sub $t3, $t1, $t2	Subtract $t1 - $t2 → $t3	$t3 = 3 - 7 = -4 (2's complement)
0X10	sw $t2, 4($zero)	Store $t2 to memory address 4	mem[4] = 7
0X14	lw $t4, 4($zero)	Load value from memory address 4 into $t4	$t4 = 7
0X18	beq $t0, $t4, 2	Branch if $t0 == $t4 → Not taken	No branch, continue
0X1C	addi $t5, $zero, 1	Load 1 into $t5	$t5 = 1
0X20	addi $t5, $zero, 2	Load 2 into $t5 (overwrites previous value)	$t5 = 2
0X24	beq $t5, $t5, 1	Branch taken (since $t5 == $t5)	PC → 0x2C
0X28	addi $t6, $zero, 255	Skipped due to branch from 0x24	—
0X2C	addi $t6, $zero, 1	Load 1 into $t6	$t6 = 1
0X30	addi $t6, $zero, 2	Load 2 into $t6	$t6 = 2
0X34	addi $t6, $zero, 3	Load 3 into $t6	$t6 = 3

Instruction Explanation:

Instruction @ 0x00 Assembly: addi $t0, $zero, 5
Hex: 20080005
Binary: 001000 00000 01000 0000 0000 0000 0101
Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01000 → $t0
- immediate: 0000 0000 0000 0101 → 5

Instruction @ 0x04 Assembly: addi $t1, $zero, 3
Hex: 20090003
Binary: 001000 00000 01001 0000 0000 0000 0011
Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01001 → $t1
- immediate: 0000 0000 0000 0011 → 3

Instruction @ 0x08 Assembly: add $t2, $t0, $t1
Hex: 01095020
Binary: 000000 01000 01001 01010 00000 100000
Type: R-type

- opcode: 000000
- rs: 01000 → $t0
- rt: 01001 → $t1
- rd: 01010 → $t2
- shamt: 00000
- funct: 100000 → add

Instruction @ 0x0C Assembly: sub $t3, $t1, $t2
Hex: 012A5822
Binary: 000000 01001 01010 01011 00000 100010
Type: R-type

- opcode: 000000
- rs: 01001 → $t1
- rt: 01010 → $t2
- rd: 01011 → $t3

- shamt: 00000
- funct: 100010 → sub

Instruction @ 0x10 Assembly: sw $t2, 4($zero)
Hex: AC0A0004
Binary: 101011 00000 01010 0000 0000 0000 0100
Type: I-type
- opcode: 101011 → sw
- rs: 00000 → $zero
- rt: 01010 → $t2
- immediate: 0000 0000 0000 0100 → 4

Instruction @ 0x14 Assembly: lw $t4, 4($zero)
Hex: 8C0C0004
Binary: 100011 00000 01100 0000 0000 0000 0100
Type: I-type
- opcode: 100011 → lw
- rs: 00000 → $zero
- rt: 01100 → $t4
- immediate: 0000 0000 0000 0100 → 4

Instruction @ 0x18 Assembly: beq $t0, $t4, 2
Hex: 110C0002
Binary: 000100 01000 01100 0000 0000 0000 0010
Type: I-type
- opcode: 000100 → beq
- rs: 01000 → $t0
- rt: 01100 → $t4
- immediate: 0000 0000 0000 0010 → 2

Instruction @ 0x1C Assembly: addi $t5, $zero, 1
Hex: 200D0001
Binary: 001000 00000 01101 0000 0000 0000 0001
Type: I-type
- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01101 → $t5
- immediate: 0000 0000 0000 0001 → 1

Instruction @ 0x20 Assembly: addi $t5, $zero, 2
Hex: 200D0002
Binary: 001000 00000 01101 0000 0000 0000 0010
Type: I-type
- opcode: 001000 → addi

- rs: 00000 → $zero
- rt: 01101 → $t5
- immediate: 0000 0000 0000 0010 → 2

Instruction @ 0x24 Assembly: beq $t5, $t5, 1
Hex: 11AD0001
Binary: 000100 01101 01101 0000 0000 0000 0001
Type: I-type

- opcode: 000100 → beq
- rs: 01101 → $t5
- rt: 01101 → $t5
- immediate: 0000 0000 0000 0001 → 1

Instruction @ 0x28 Assembly: addi $t6, $zero, 255
Hex: 200E00FF
Binary: 001000 00000 01110 0000 0000 1111 1111
Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01110 → $t6
- immediate: 0000 0000 1111 1111 → 255

(Skipped if branch taken)

Instruction @ 0x2C Assembly: addi $t6, $zero, 1
Hex: 200E0001
Binary: 001000 00000 01110 0000 0000 0000 0001
Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01110 → $t6
- immediate: 0000 0000 0000 0001 → 1

Instruction @ 0x30 Assembly: addi $t6, $zero, 2
Hex: 200E0002
Binary: 001000 00000 01110 0000 0000 0000 0010
Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01110 → $t6
- immediate: 0000 0000 0000 0010 → 2

Instruction @ 0x34 Assembly: addi $t6, $zero, 3

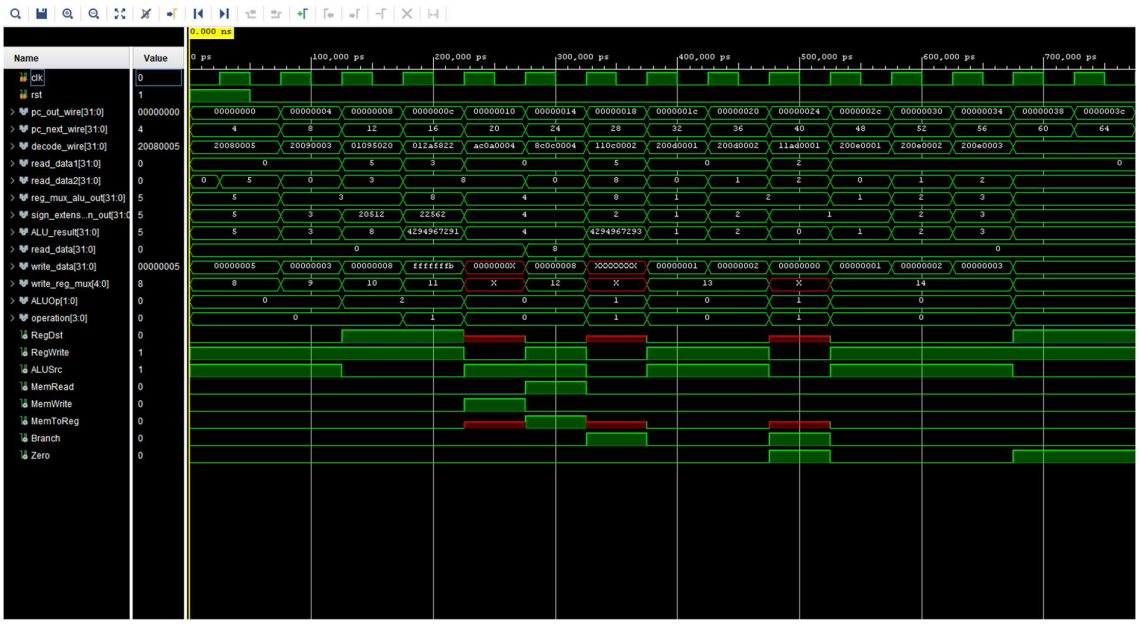
Hex: 200E0003

Binary: 001000 00000 01110 0000 0000 0000 0011

Type: I-type

- opcode: 001000 → addi
- rs: 00000 → $zero
- rt: 01110 → $t6
- immediate: 0000 0000 0000 0011 → 3

Waveform:



Result:

Single Cycle MIPS32 ISA Processor for my custom instruction (R type, I type, LW, SW) is **Successfully built**. The design is **verified.**

FUTURE WORK:

Need to modify the code so that it supports Jump instructions.
Idea to Implement:

1. Add a Jump reg to the control block to Identify opcode (6'b000010) and set it to 1.
2. Create a module for this:
   jump address = {PC+4[31:28], decode_wire [25:0], 2'b00};
3. *In1 of mux is jump address and in0 is (pc+4) pc adder. And select line is jump wire.*
4. *The output of this mux is connected to in0 of mux.*

the best place to insert the Jump MUX is just before the top-right MUX.

Theory for JUMP Instruction Execution:

1. Why Shift Left by 2?

*MIPS instructions are **word-aligned** — each instruction starts at an address divisible by 4 (e.g., 0x00000000, 0x00000004, 0x00000008, etc.).*
*So the actual memory address of any instruction will **always end in 00 in binary**.*

2.Why Take PC+4[31:28]?

*This is because MIPS jump instructions **only jump within the current 256MB memory block**.*
***Each 256MB block** has the same top 4 bits in a 32-bit address:*

- *e.g., 0x00400000 to 0x004FFFFF all have top 4 bits as 0x0.*

So, we just copy those bits from the current PC + 4.
Jump stays within the same region of the memory, which is how MIPS J-type instructions are defined.

Term	***Value***	***Why?***
instruction [25:0]	*26 bits*	*Jump instruction field*
After shift <<2	*28 bits*	*Word to byte address*
Top 4 bits from PC+4[31:28]	*4 bits*	*To complete full 32-bit address*
Max jumpable range	*256 MB*	*Because 2^28 = 256 MB*

3.The offset is 26 bit.It specifies what address it shd jump to.If you modify it by left shift twice. Wont it change the address?

*No, it doesn't "change" the address. In fact, **left-shifting by 2 is necessary**, because the 26-bit value is **not a byte address**, but a **word address**.*

Example:

*If the jump target is **instruction 100**, it lives at **address 100 × 4 = 400 bytes**.*

Addressing is based on Bytes.. it shd jump to 400th location.

the 26-bit field in the instruction is Word address

But the actual memory address is Word Address × 4 -> which means shift left by 2 bits
Example:
instruction [25:0] = 26'b000000_000000_000000_000000_000100 (decimal 4)
*It refers to **instruction number 4**, located at byte address = 4 × 4 = 16 = 0x10*

4.256 MB region concept?? why the restriction for jumpable range?

*The upper 4 bits PC+4[31:28] define which **256 MB "region"** you're in.*
That is:

- *0x0--- ---- to 0x0FFF FFFF → First 256 MB*
- *….*
- *0xF--- ---- to 0xFFFF FFFF → Last 256 MB*

*So, when jumping, you're **stuck in the same 256 MB region** as where you came from. The top 4 bits from PC+4 are **not changeable** by the jump instruction.*
Restriction: *this goes into **ISA design trade-offs** and hardware simplicity.*
A jump (j) instruction format: | opcode (6) | target address (26) |
*That leaves only **26 bits** for the target address.*
Byte Addressing + Word Alignment: 26 bits + shift by 2 = 28 bits = 256 MB range
jump address = {PC+4[31:28], instruction [25:0], 2'b00};
This means:

- *Jump can only **replace the lower 28 bits** of PC.*
- ***Top 4 bits stay the same**, so the jump stays **within the current 256 MB segment**.*

REFERENCES:

ARCHITECTURE DESIGN:

https://www.cs.nthu.edu.tw/~king/courses/cs4100/P51-proc.pdf

https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec07.pdf

https://docslib.org/download/952472/mips32-architecture-for-programmers-volume-i-introduction-to-the-mips32-architecture

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html