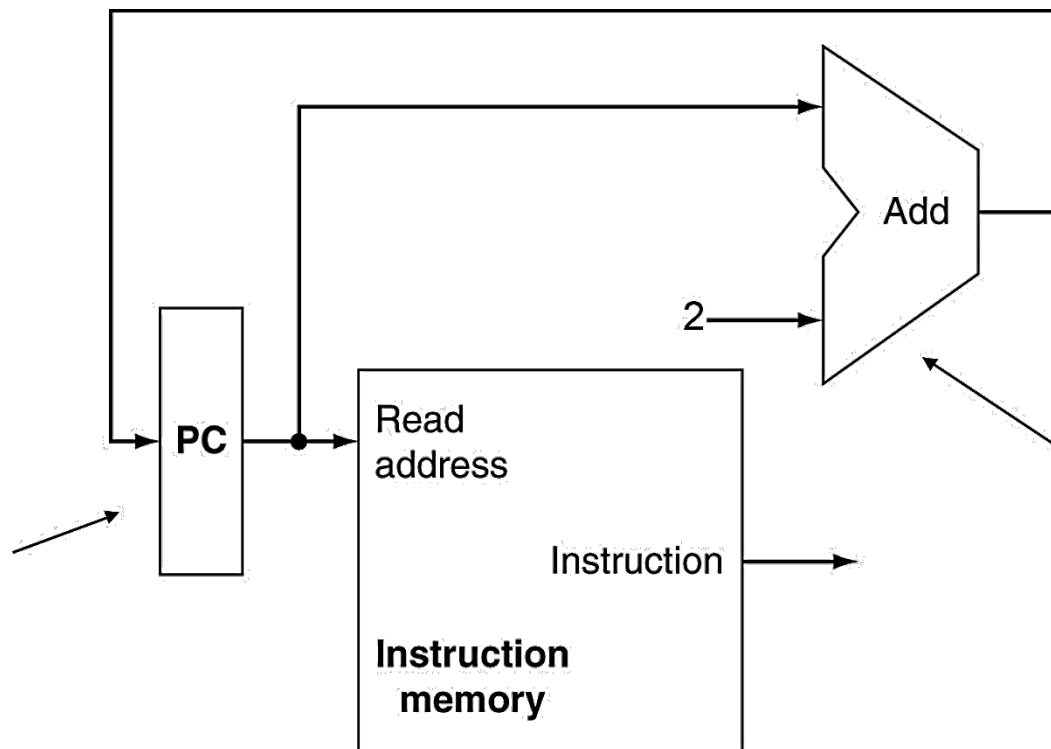


Introduction:

A MIPS (Microprocessor without Interlocked Pipeline Stages) processor is a type of reduced instruction set computer (RISC) microprocessor architecture. It is widely used in various applications, including embedded systems, consumer electronics, and networking devices. The MIPS architecture is known for its simplicity and efficiency, making it popular in both academic and industrial settings.

A MIPS processor typically consists of the following major components:

Instruction Fetch (IF) Unit: Fetches instructions from memory based on the program counter (PC) and provides them to the subsequent stages for execution.



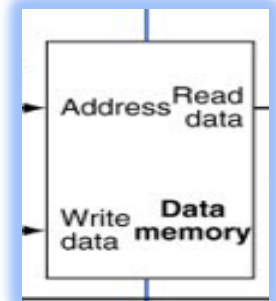
Instruction Decode (ID) Unit: Decodes the fetched instructions and extracts relevant information, such as the operation code (opcode) and operands.

Execution Unit: Performs arithmetic and logical operations, such as addition, subtraction, AND, OR, etc., on the operands. It also handles control flow instructions like branches and jumps.

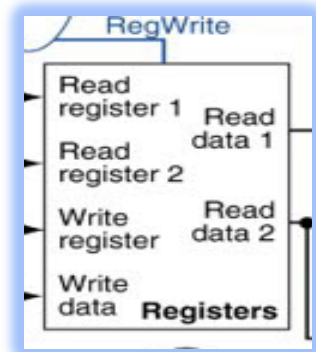
Memory Unit: Interfaces with the memory system to perform load (LW) and store (SW) operations, allowing data transfer between the processor and memory.

```
*ram.txt
1 0000_0000_0000_0000 //
2 0000_0000_0000_0000 //
3 0000_0000_0000_0000 //
4 0000_0000_0000_0000 //
5 0000_0000_0000_1100 // 6
6 0000_0000_0000_0010 // 2
7 0000_0000_0000_0000 //
8 0000_0000_0000_0000
9 0000_0000_0000_0000 //
10 0000_0000_0000_0000
11 0000_0000_0000_0000 //
12 0000_0000_0000_0000
13 0000_0000_0000_0000 // I
14 0000_0000_0000_0000
15 0000_0000_0000_0000 //
16 0000_0000_0000_0000

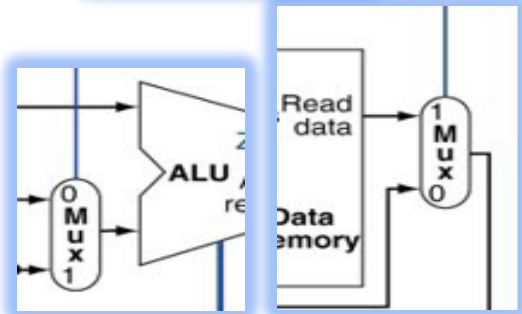
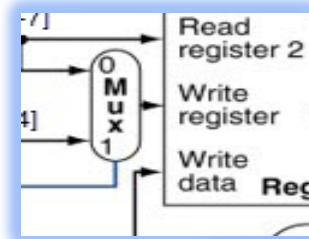
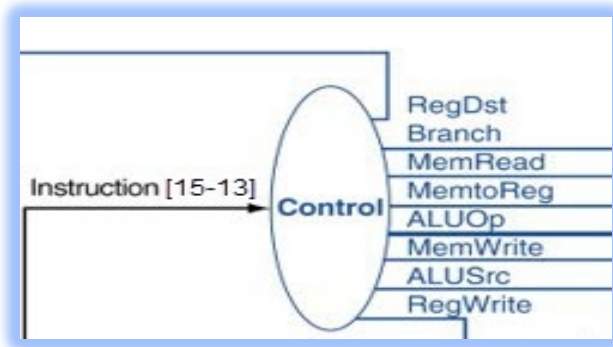
I0 0000_0000_0000_0000
I2 0000_0000_0000_0000 //
I4 0000_0000_0000_0000
I3 0000_0000_0000_0000 // I
I5 0000_0000_0000_0000
// 0000_0000_0000_0000 //
```



Registers: MIPS processors typically have a set of 32 general-purpose registers (R0 to R31), along with special-purpose registers such as the program counter (PC), instruction register (IR), and others.



Control Unit: Manages the flow of instructions and data between different components of the processor, controls the execution of instructions, and handles control signals for various operations.

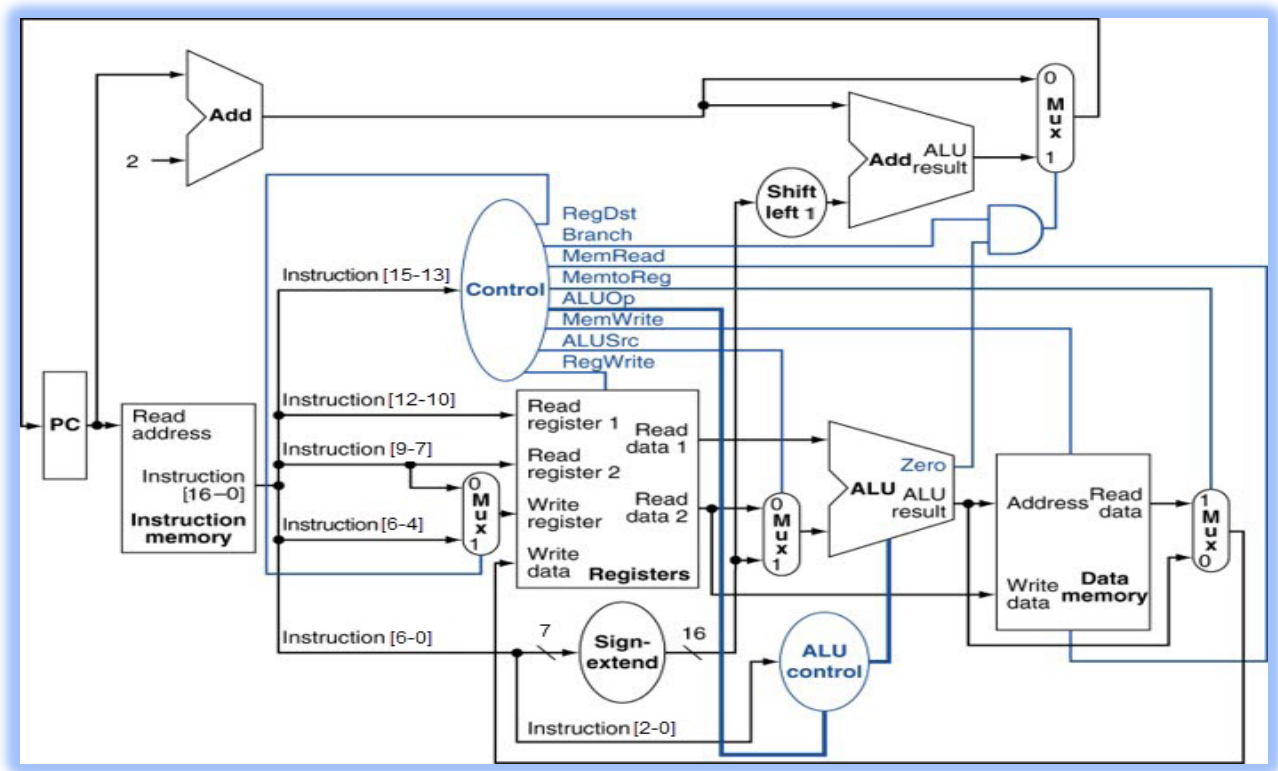


Control signals									
Instruction	Reg Dst	ALUSrc	Memto Reg	Reg Write	MemRead	Mem Write	Branch	ALUOp	Jump
R-type(add,sub..)	1	0	0	1	0	0	0	00	0
LW	0	1	1	1	1	0	0	11	0
SW	0	1	0	0	0	1	0	11	0
addi	0	1	0	1	0	0	0	11	0
beq	0	0	0	0	0	0	1	01	0

In this table, we define for each instruction which pin select's muxes should be high and which ones to be low as we need for instructions. we use this table in control_unit.v.

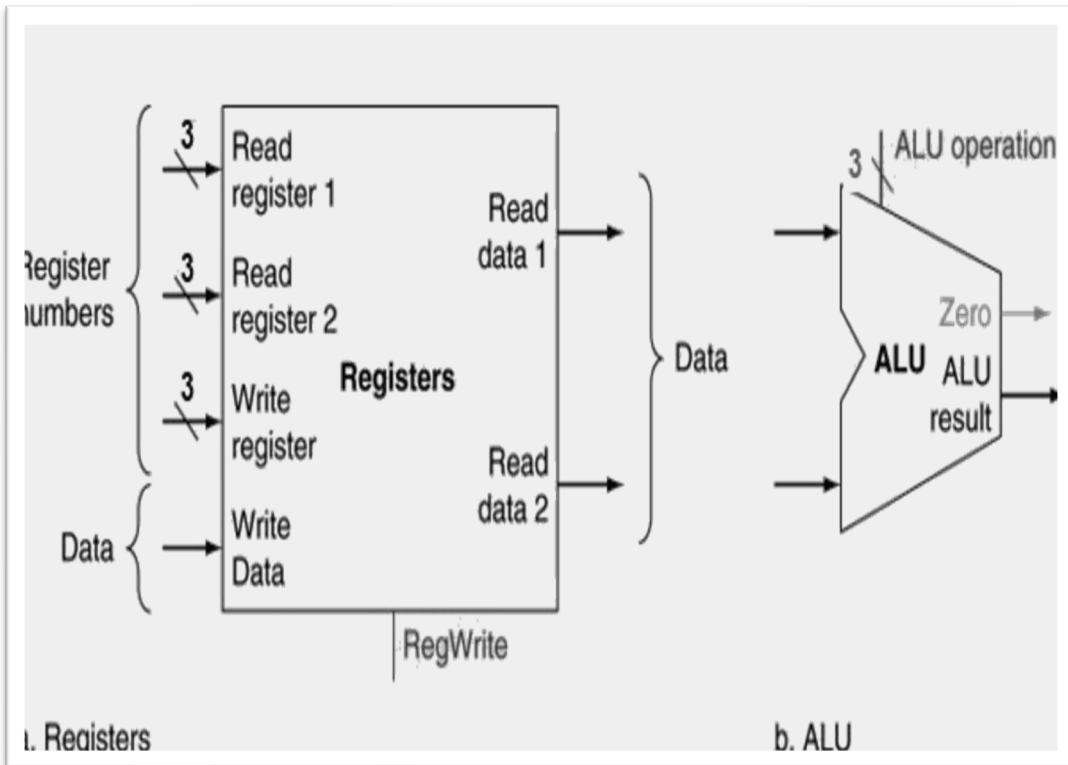
The MIPS instruction set architecture (ISA) is known for its simplicity and regularity. It includes a fixed-length instruction format and a limited number of instruction formats. Instructions are typically 32 bits long and can perform operations on registers, immediate values, or memory locations.

Full Datapath:



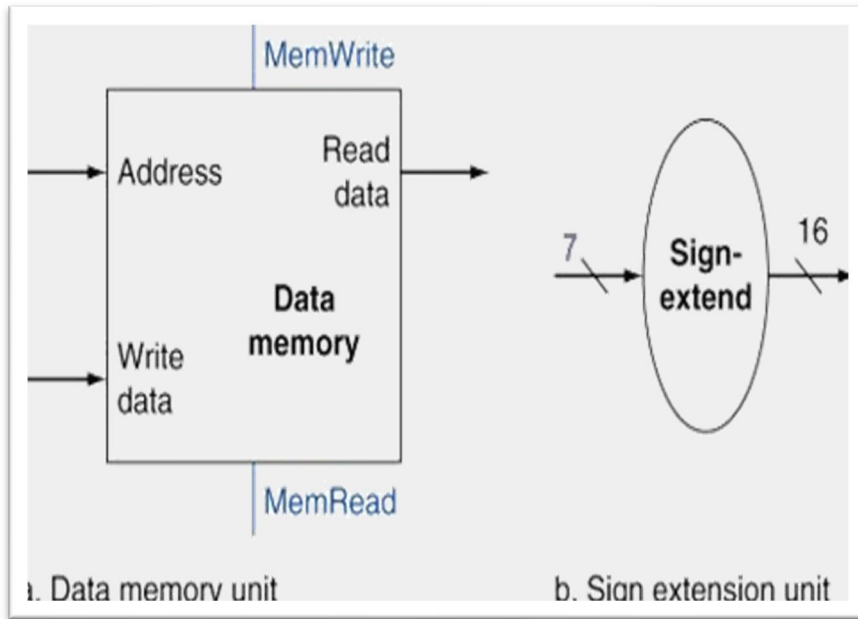
R-Format Instructions:

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Load/Store Instructions:

- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



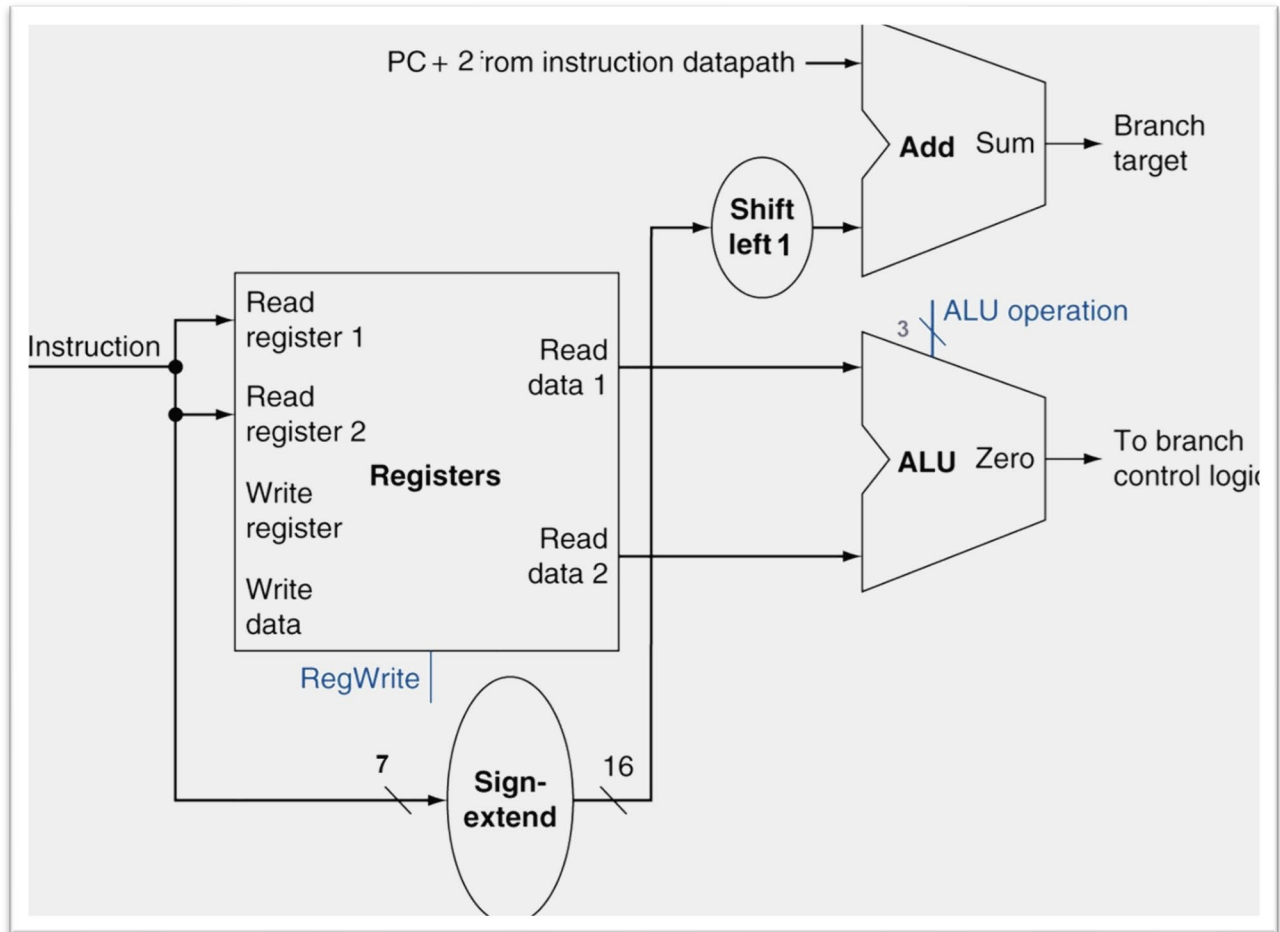
Branch Instructions:

- ❖ Read register
- ❖ Compare
- ❖ Use ALU, subtract and check Zero output
- ❖ Calculate the target address

Sign-extend displacement

Shift left 1 places Add to PC + 2

Already calculated by instruction fetch



Test bench :

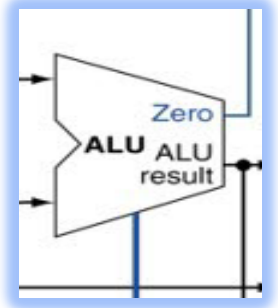
These are the instructions we are going to run for the test processor. we also can run "jump" and "and" instructions on this processor. Since the processor does only understand the machine code we converted each line to binary and hex number. we need to follow the table below to convert each instruction.

rom.txt					
1	1000_0000_1000_0100	// lw \$1, 4(\$zero)	⇒	#0x8084	// pc=0000
2	1000_0001_0000_0101	// lw \$2, 5(\$zero)	⇒	#0x8105	// pc=0010
3	1100_0101_0000_0010	// beq \$1,\$2,jump	⇒	#0xC502	// pc=0100
4	0000_0101_0011_0001	// sub \$3,\$1,\$2	⇒	#0x531	// pc=0110
5	1010_0001_1000_0111	// sw \$3, 7(\$zero)	⇒	#0xA187	// pc=1000
6	0000_0101_0011_0000	// jump:add \$3, \$1, \$2	⇒	#0x530	// pc=1010
7	1010_0001_1000_0111	// sw \$3, 7(\$zero)	⇒	#0xA187	// pc=1100
8	0000_0000_0000_0000	//			
9	0000_0000_0000_0000	//			
10	0000_0000_0000_0000	//			
11	0000_0000_0000_0000	//			
12	0000_0000_0000_0000	//			
13	0000_0000_0000_0000	//			
14	0000_0000_0000_0000	//			
15	0000_0000_0000_0000	//			

					immediate		comments
Comment Name	format	Opcode (3bits)	Rs (3bits)	Rt (3bits)	Rd (3bits)	Func (4bits)	
lw	I	100	000	001	0000100		lw \$1, 4(\$zero)
lw	I	100	000	010	0000101		lw \$2, 5(\$zero)
beq	I	110	001	010	0000010		beq \$1,\$2,jump
sub	R	000	001	010	011	0001	sub \$3,\$1,\$2
sw	I	101	000	011	0000111		sw \$3, 7(\$zero)
add	R	000	001	010	011	0000	jump: add \$3, \$1, \$2
sw	I	101	000	011	0000111		sw \$3, 7(\$zero)

We define each comment with a specific opcode for example lw opcode is 4(100) or beq opcode is (110).for add and sub we have the same opcode but a different function code.so we need to check func code after we got add or sub or and..... we use this table in control_unit.v.

Alu:



```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan,  Spring,2023      ***
***                                                                ***
***                                                                ***
***                                                                ***
****
*** Filename//  alu.v Created by Fred Ryan, 12 May 2023            ***
*** --- revision history, if any, goes here ---                    ***
*****/
`timescale 1 ns / 1 ps

module alu(
    input      [15:0]    a,           // Input signal a
    input      [15:0]    b,           // Input signal b
    input      [2:0]     alu_ctrl,     // Function select control
    signal for ALU operation
    output reg  [15:0]    result,      // Output signal for the ALU
    result
    output zero           // Output signal indicating
    if the result is zero
);
    always @(*)
    begin
        case(alu_ctrl) // Perform different
operations based on alu_ctrl value
            3'b000: result = a + b; // Add operation
            3'b001: result = a - b; // Subtract operation
            3'b010: result = a & b; // Bitwise AND operation
            3'b011: result = a | b; // Bitwise OR operation
            3'b100: begin if (a<b) result = 16'd1; // Compare operation: if a
< b, set result to 1; otherwise, set result to 0
end
end

```

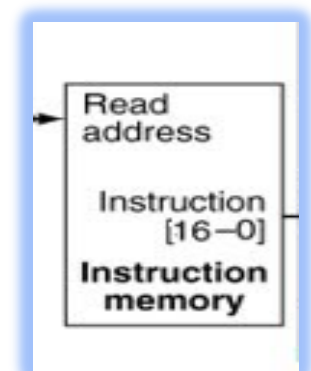
```

        else result = 16'd0;
        end
    default: result = a + b;           // Default operation is
    addition                           // addition
    endcase
end
assign zero = (result==16'd0) ? 1'b1: 1'b0; // Assign zero output
based on the result value
endmodule

```

The comments provided above explain the purpose and functionality of each line in a Verilog module for an ALU (Arithmetic Logic Unit). The module takes two input signals **a** and **b**, performs the specified ALU operation based on the **alu_ctrl** control signal, and produces the output **result**. It also assigns the output signal **zero** based on whether the result is zero or not. The ALU supports addition, subtraction, bitwise AND, bitwise OR, and comparison operations.

Instruction Memory:



```

/*****
*** EE 526 L 16-bit single cycle MIPS Fred Ryan, Spring, 2023 ***
***
***
***
****
*** Filename// instr_mem.v Created by Fred Ryan, 12 May 2023 ***
*** --- revision history, if any, goes here --- ***
*****/
`timescale 1 ns / 1 ps // Set the timescale for simulation

module instr_mem
(
    input [15:0] pc, // Input signal: program
    counter
    output wire [15:0] instruction // Output signal: instruction
    fetched from instruction memory
);

wire [3:0] rom_addr = pc[4:1]; // Create a new 4-bit signal
called rom_addr and assign it the value of the 4 bits of the pc signal from
bit 4 down to bit 1

/*
    lw $1, 4($zero)
    lw $2, 5($zero)
    beq $1,$2,jump
    sub $3,$1,$2
    sw $3, 7($zero)
    jump: add $3, $1, $2
    sw $3, 7($zero)
*/

reg [15:0] rom[15:0]; // Declare a register array
called rom to store instructions

initial
begin
    $readmemb("./test/rom.txt", rom, 0, 14); // Read the contents of the
"rom.txt" file and initialize the rom array with the instruction values
end

assign instruction = rom[rom_addr[3:0]]; // Assign the instruction
output based on the value of the rom_addr signal

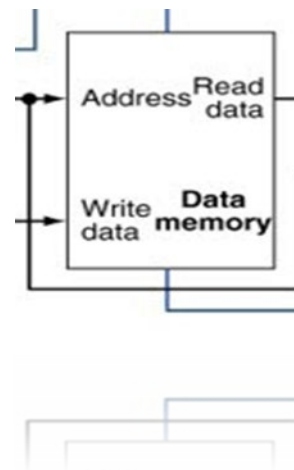
endmodule

```

The comments provided above explain the purpose and functionality of each line in a Verilog module for instruction memory. The module takes a program counter (**pc**) as input and outputs the corresponding instruction stored in the memory. It uses a register array (**rom**) to store

the instructions. The **rom_addr** signal is derived from the **pc** to determine the address of the instruction to be fetched. The instructions are read from a file (**rom.txt**) using the **\$readmemb** system function. The selected instruction is assigned to the **instruction** output signal. The commented section indicates the instructions stored in memory (not directly used in the code).

Data memory (ram.txt):



```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan,  Spring,2023      ***
***                                                                ***
***                                                                ***
***                                                                ***
*****/
*** Filename//  data_memory.v Created by Fred Ryan, 12 May 2023    ***
*** --- revision history, if any, goes here ---                    ***
*****/
`timescale 1 ns / 1 ps

module data_memory
    input clk,
    input [15:0] mem_addr,
    input [15:0] mem_write,
    input mem_write_en,
    input mem_read,
    // read port
    output [15:0] mem_read_data,
    output wire [15:0] mem
);

```

```

reg [15:0] ram[15:0]; // Declaration of a register array named "ram" with 16
elements, each of 16 bits.
integer i; // Declaration of an integer variable "i".

wire [7:0] ram_addr = mem_addr[7:0]; // Declaration of a wire "ram_addr"
which takes the lower 8 bits of "mem_addr".

initial begin
    $readmemb("./test/ram.txt", ram, 0, 15); // Reads the contents of a
memory initialization file into the "ram" array.
    for (i = 0; i < 16; i = i + 1)
        $display("ram%d = %0d", i, ram[i]); // Displays the values of the
"ram" array elements.
end

always @(posedge clk) begin
    if (mem_write_en)
        ram[ram_addr] <= mem_write; // Updates the value in "ram" at the
specified address when mem_write_en is asserted.
end

assign mem_read_data = (mem_read == 1'b1) ? ram[ram_addr] : 16'd0; //
Assigns the value from "ram" at the specified address to mem_read_data if
mem_read is asserted, else assigns 16'd0.
assign mem = ram[7]; // Assigns the value from "ram" at address 7 to the
"mem" wire.

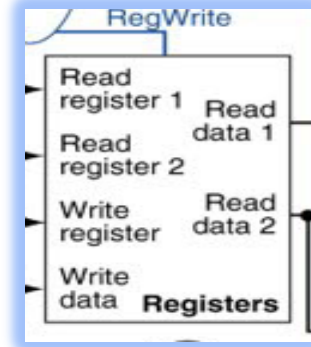
initial begin
    #250;
    $display("\nram7 = %d \n", mem); // Displays the value in "ram" at
address 7.
end

endmodule

```

This Verilog code defines a data memory module (**data_memory**) with read and write ports. It uses a register array (**ram**) to store and retrieve data. The module reads the initial contents of the memory from a file, performs write operations based on the input signals (**mem_write**, **mem_write_en**), and provides the read data (**mem_read_data**) based on the input address (**mem_addr**) and read control (**mem_read**). The **mem** wire holds the value from **ram** at address 7. The code also includes some initial blocks for displaying the values of **ram** elements and the value at address 7 after a delay of 250 units of time.

Registers:



```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan, Spring,2023 ***
*** ***
*** ***
*** ***
*****/
*** Filename// register_file.v Created by Fred Ryan, 12 May 2023 ***
*** --- revision history, if any, goes here --- ***
*****/
`timescale 1 ns / 1 ps

module register_file (
    input clk,
    input rst,
    // write port
    input reg_write,
    input [2:0] write_dest,
    input [15:0] write_data,
    // read port 1
    input [2:0] read_addr_1,
    output [15:0] read_data_1,
    // read port 2
    input [2:0] read_addr_2,
    output [15:0] read_data_2,
    output wire [15:0] reg1, reg2, reg3
);

reg [15:0] reg_file[7:0]; // Declaration of a register array named
"reg_file" with 8 elements, each of 16 bits.

integer i; // Declaration of an integer variable "i".

always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < 8; i = i + 1)
            reg_file[i] <= 16'd0; // Resetting all elements of "reg_file" to
16'd0 when "rst" is asserted.
        end
    else begin
        if (reg_write) begin
            reg_file[write_dest] <= write_data; // Writing "write_data" to
the specified destination address in "reg_file" when "reg_write" is asserted.
        end
    end
end

```

```

        end
    end

    assign read_data_1 = reg_file[read_addr_1]; // Assigning the value from
    "reg_file" at "read_addr_1" to "read_data_1".
    assign read_data_2 = reg_file[read_addr_2]; // Assigning the value from
    "reg_file" at "read_addr_2" to "read_data_2".
    assign reg1 = reg_file[1]; // Assigning the value from "reg_file" at address
    1 to "reg1".
    assign reg2 = reg_file[2]; // Assigning the value from "reg_file" at address
    2 to "reg2".
    assign reg3 = reg_file[3]; // Assigning the value from "reg_file" at address
    3 to "reg3".

endmodule

```

ALU Control:

```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan, Spring, 2023 ***
*** ***
*** ***
*** ***
*****/
*****/
*** Filename// control_unit.v Created by Fred Ryan, 12 May 2023 ***
*** --- revision history, if any, goes here --- ***
*****/

`timescale 1 ns / 1 ps
module control_unit( input[2:0] opcode,
                    input reset,
                    output reg[1:0] alu_op,
                    output reg
jump,beq,bne,mem_read,mem_write,alu_src,reg_dst,mem_to_reg,reg_write
);
    always @(*)
    begin
        if(reset == 1'b1) begin
            reg_dst = 1'b0;
            mem_to_reg = 1'b0;
            alu_op = 2'b00;
            jump = 1'b0;
            beq = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            reg_write = 1'b0;
            bne = 1'b0;
        end
        else begin
            case(opcode)
            3'b000: begin // R type (add,sub,and,or,slt,jr)
                reg_dst = 1'b1;
            end

```



```

mem_to_reg = 1'b0;
alu_op = 2'b00;
jump = 1'b0;
beq = 1'b0;
mem_read = 1'b0;
mem_write = 1'b0;
alu_src = 1'b0;
reg_write = 1'b1;
bne = 1'b0;
end
3'b001: begin // I type (slti)
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b10;
    jump = 1'b0;
    beq = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b1;
    reg_write = 1'b1;
    bne = 1'b0;
end
3'b010: begin // j type(j)
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b00;
    jump = 1'b1;
    beq = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b0;
    bne = 1'b0;
end
3'b011: begin// I type (bne)
    reg_dst = 1'b0;
    alu_src = 1'b0;
    mem_to_reg = 1'b0;
    reg_write = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    beq = 1'b0;
    bne = 1'b1;
    alu_op = 2'b01;
    jump = 1'b0;
end
3'b100: begin // I type (lw)
    reg_dst = 1'b0;
    mem_to_reg = 1'b1;
    alu_op = 2'b11;
    jump = 1'b0;
    beq = 1'b0;
    mem_read = 1'b1;
    mem_write = 1'b0;
    alu_src = 1'b1;
    reg_write = 1'b1;

```

```

        bne = 1'b0;
    end
3'b101: begin // I type( sw)
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b11;
    jump = 1'b0;
    beq = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b1;
    alu_src = 1'b1;
    reg_write = 1'b0;
    bne = 1'b0;
end
3'b110: begin // I type (beq)
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b01;
    jump = 1'b0;
    beq = 1'b1;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b0;
    bne = 1'b0;
end
3'b111: begin // I type (addi)
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b11;
    jump = 1'b0;
    beq = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b1;
    reg_write = 1'b1;
    bne = 1'b0;
end
default: begin
    reg_dst = 1'b1;
    mem_to_reg = 1'b0;
    alu_op = 2'b00;
    jump = 1'b0;
    beq = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b1;
    bne = 1'b0;
end
endcase
end
end
endmodule

module ALUControl( ALU_Control, ALUOp, Function);
output reg[2:0] ALU_Control;

```

```

input [1:0] ALUOp;
input [3:0] Function;
wire [5:0] ALUControlIn;
assign ALUControlIn = {ALUOp,Function};
always @(ALUControlIn)
case (ALUControlIn)
6'b11xxxx: ALU_Control=3'b000; // Addi, lw, sw
6'b10xxxx: ALU_Control=3'b100; //i-type: slti if (a<b) result = 16'd1;
6'b01xxxx: ALU_Control=3'b001; // sub
6'b000000: ALU_Control=3'b000; // R-type: ADD
6'b000001: ALU_Control=3'b001; // R-type: sub
6'b000010: ALU_Control=3'b010; // R-type: AND
6'b000011: ALU_Control=3'b011; // R-type: OR
6'b000100: ALU_Control=3'b100; //R-type: slt if (a<b) result = 16'd1;
default: ALU_Control=3'b000;
endcase
endmodule

```

This code represents a control unit and an ALU control module in a digital system. Here's a breakdown of what the code does:

1. The **control_unit** module takes an opcode and reset signal as inputs and provides various control signals as outputs.
2. Inside the **control_unit** module, there is an **always** block that triggers whenever there is a change in the inputs.
3. If the reset signal is active (**reset == 1'b1**), all the control signals are set to their initial values (usually 0) to initialize the system.
4. If the reset signal is not active (**reset == 1'b0**), a case statement (**case(opcode)**) is used to determine the appropriate control signals based on the opcode.
5. The case statement checks the value of the opcode and assigns specific values to the control signals accordingly.
6. Each case represents a specific type of instruction (R-type, I-type, J-type) or specific operations (add, sub, lw, sw, etc.).
7. The default case is used to assign default values to the control signals in case the opcode does not match any of the defined cases.
8. The **ALUControl** module is a separate module responsible for generating the appropriate ALU control signal based on the ALUOp and Function inputs.

9. It uses a **case** statement to match specific bit patterns of the combined ALUOp and Function inputs (**ALUControlIn**).
10. Based on the matched bit patterns, specific values are assigned to the **ALU_Control** output, representing different ALU operations (add, sub, AND, OR, slt, etc.).
11. The default case assigns a default value to the **ALU_Control** output in case none of the defined patterns match.

We created another module in this file and we called module `ALUControl`, which is responsible for generating the control signal `ALU_Control` based on input signals `ALUOp` and `Function`. Here's an explanation of what the code does:

The `ALUControl` module has three output ports: `ALU_Control` (a 3-bit reg) and two input ports: `ALUOp` (a 2-bit input) and `Function` (a 4-bit input).

Inside the module, there is a wire `ALUControlIn` declared, which is a concatenation of `ALUOp` and `Function`.

The `ALUControlIn` wire combines the two input signals into a 6-bit value, where the most significant 2 bits come from `ALUOp`, and the remaining 4 bits come from `Function`.

The `ALUControlIn` wire is assigned the concatenated value of `ALUOp` and `Function`.

There is an `always` block triggered whenever there is a change in `ALUControlIn`.

Inside the `always` block, a `case` statement is used to match specific bit patterns of `ALUControlIn`.

The `case` statement checks the value of `ALUControlIn` and assigns a specific value to `ALU_Control` based on the matched bit patterns.

Each case represents a specific bit pattern of `ALUControlIn` and assigns a corresponding value to `ALU_Control`.

For example, if `ALUControlIn` matches the pattern `6'b11xxxx`, `ALU_Control` is assigned the value `3'b000`.

There are specific cases defined for different operations, such as addition (ADD), subtraction (sub), logical AND (AND), logical OR (OR), and set less than (slt).

The default case is used to assign a default value of 3'b000 to ALU_Control in case none of the defined patterns match ALUControlIn.

The Main Control Unit(mips_16):

```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan, Spring,2023 ***
*** ***
*** ***
*** ***
****
*** Filename// mips_16.v Created by Fred Ryan, 12 May 2023 ***
*** --- revision history, if any, goes here --- ***
*****/
`timescale 1 ns / 1 ps // Sets the timescale for the module

module mips_16( input clk,reset, output[15:0] pc_out,instruction,
alu_result,reg1,reg2,reg3 ); // Defines a module named "mips_16" with input
and output ports

wire [15:0] instr; // Declares a 16-bit wire "instr"
wire[1:0] alu_op; // Declares a 2-bit wire "alu_op"
wire [2:0] write_dest; // Declares a 3-bit wire "write_dest"
wire [15:0] write_data; // Declares a 16-bit wire "write_data"
wire [2:0] read_addr_1; // Declares a 3-bit wire "read_addr_1"
wire [15:0] read_data_1; // Declares a 16-bit wire "read_data_1"
wire [2:0] read_addr_2; // Declares a 3-bit wire "read_addr_2"
wire [15:0] read_data_2; // Declares a 16-bit wire "read_data_2"
wire jump,beq,mem_read,mem_write,alu_src,reg_dst,mem_to_reg,reg_write,bne;
// Declares several wires for control signals
wire [15:0] read_data2,imm_ext; // Declares several 16-bit wires
wire [2:0] ALU_Control; // Declares a 3-bit wire "ALU_Control"
wire [15:0] ALU_out; // Declares a 16-bit wire "ALU_out"
wire zero_flag; // Declares a wire "zero_flag"
reg[15:0] pc_current; // Declares a 16-bit register "pc_current"
wire signed[15:0] pc_next,pc_2; // Declares two 16-bit wires "pc_next" and
"pc_2"
wire signed[15:0] im_shift_1, PC_j, PC_beq, PC_4beq,PC_bne,PC_4bne; //
Declares several 16-bit wires
wire beq_control,bne_control; // Declares wires for control signals
wire [14:0] jump_shift_1; // Declares a 15-bit wire "jump_shift_1"
wire [15:0] mem_read_data; // Declares a 16-bit wire "mem_read_data"

// PC
always @(posedge clk or posedge reset) // Executes the following block on
the positive edge of the clock or the positive edge of the reset signal
begin
    if(reset) // If the reset signal is active
        pc_current <= 16'd0; // Assigns 0 to the register "pc_current"

```

```

        else
            pc_current <= pc_next; // Assigns the value of "pc_next" to the
register "pc_current"
        end

// PC + 2
assign pc_2 = pc_current + 16'd2; // Adds 2 to "pc_current" and assigns the
result to "pc_2"
// instruction memory
// instruction memory
instr_mem instrucion_memory(.pc(pc_current),.instruction(instr));

// jump shift left 1
assign jump_shift_1 = {instr[12:0],1'b0};

// control unit
control_unit control(.reset(reset),.opcode(instr[15:13]),.reg_dst(reg_dst),

.mem_to_reg(mem_to_reg),.alu_op(alu_op),.jump(jump),.beq(beq),.mem_read(mem_r
ead),

.mem_write(mem_write),.alu_src(alu_src),.reg_write(reg_write),.bne(bne));

// multiplexer regdest
assign write_dest = (reg_dst==1'b1) ? instr[6:4] : instr[9:7];

// register file
assign read_addr_1 = instr[12:10];
assign read_addr_2 = instr[9:7];
register_file reg_file(.clk(clk),.rst(reset),.reg_write(reg_write),
.write_dest(write_dest),
.write_data(write_data),
.read_addr_1(read_addr_1),
.read_data_1(read_data_1),
.read_addr_2(read_addr_2),
.read_data_2(read_data_2),
.reg1(reg1),
.reg2(reg2),
.reg3(reg3));

// sign extend
assign imm_ext = {{9{instr[6]}},instr[6:0]}; //9{instr[6]}: This expression
creates a 9-bit signal where all the bits are the same as the 6th bit of the
instr signal

// ALU control unit
ALUControl
ALU_Control_unit(.ALUOp(alu_op),.Function(instr[3:0]),.ALU_Control(ALU_Contro
l));

// multiplexer alu_src
assign read_data2 = (alu_src==1'b1) ? imm_ext : read_data_2;

// ALU
alu
alu_unit(.a(read_data_1),.b(read_data2),.alu_ctrl(ALU_Control),.result(ALU_ou
t),.zero(zero_flag));

```

```

// immediate shift 1
assign im_shift_1 = {imm_ext[14:0],1'b0};

// PC beq&bne add
assign PC_beq = pc_2 + im_shift_1 ;
assign PC_bne = pc_2 + im_shift_1 ;

// beq control
assign beq_control = beq & zero_flag;
assign bne_control = bne & (~zero_flag);

// PC_beq
assign PC_4beq = (beq_control==1'b1) ? PC_beq : pc_2;

// PC_bne
assign PC_4bne = (bne_control==1'b1) ? PC_bne : PC_4beq;

// PC_j
assign PC_j = {pc_2[15],jump_shift_1};

// PC_next
assign pc_next = (jump == 1'b1) ? PC_j : PC_4beq;

// data memory
data_memory datamem(.clk(clk),.mem_addr(ALU_out),
.mem_write(read_data_2),.mem_write_en(mem_write),.mem_read(mem_read),
.mem_read_data(mem_read_data));

// write back
assign write_data = (mem_to_reg == 1'b0) ? pc_2:((mem_to_reg == 1'b1)?
mem_read_data: ALU_out);

// output
assign pc_out = pc_current;
assign alu_result = ALU_out;
assign instruction = instr;
endmodule

```

The provided code represents a lot of calculators and definitions and instantiates for example we define how to calculate the next “pc” and “pc” for” bne”,”beq”,”jump”, “immediate extension “.for one bite left shift we use the `assign im_shift_1 = {imm_ext[14:0],1'b0};` fore the sign extend

`assign imm_ext = {{9{instr[6]}},instr[6:0]};` This expression creates a 9-bit signal where all the bits are the same as the 6th bit of the instr signal with instr bit zero to six .

test bench(tb_mips_16.v):

```

/*****
*** EE 526 L 16-bit syngle cycle MIPS Fred Ryan,   Spring,2023   ***
***                                                         ***
***                                                         ***
***                                                         ***
*****/
*** Filename//   tb_mips_16.v Created by Fred Ryan, 12 May 2023   ***
*** --- revision history, if any, goes here ---                 ***
*****/
// Define the timescale for the simulation
timescale 1 ns / 1 ps

// Declare the testbench module
module tb_mips16;

    // Inputs
    reg clk;
    reg reset;

    // Outputs
    wire [15:0] pc_out, instr;
    wire [15:0] alu_result, reg1, reg2, reg3;

    // Instantiate the Unit Under Test (UUT)
    mips_16 uut(
        .clk(clk),
        .reset(reset),
        .pc_out(pc_out),
        .instruction(instr),
        .alu_result(alu_result),
        .reg1(reg1),
        .reg2(reg2),
        .reg3(reg3)
    );

    // Initialize the clock signal
    initial begin
        clk = 0;
        forever #10 clk = ~clk;
    end

    // Initialize the testbench
    initial begin
        // Enable VCD waveform dumping
        $vcdpluson;

        // Monitor the values of the instruction, register 1, register 2, and
        register 3
        $monitor("\ninstruction =%h, register 1=%d, register 2=%d, register
        3=%d", instr, reg1, reg2, reg3);
    end
endmodule
```



```

// Set the initial value of the reset signal
reset = 1;

// Wait for 100 ns for the global reset to finish
#100;

// Deassert the reset signal
reset = 0;

// Wait for additional 250 ns
#250;

// Finish the simulation
$finish;
end
endmodule

```

The provided code represents a testbench for the **mips_16** module. It defines the inputs (**clk** and **reset**) and the outputs (**pc_out**, **instr**, **alu_result**, **reg1**, **reg2**, and **reg3**) of the **mips_16** module. The testbench initializes the clock signal, enables VCD waveform dumping, monitors the values of the instruction, register 1, register 2, and register 3, asserts and deasserts the reset signal, waits for specific time intervals, and then finishes the simulation.

demonstrate:

Ram4=2,ram5=2

rom.txt					
1	1000_0000_1000_0100	// lw \$1, 4(\$zero)	⇒	#0x8084	// pc=0000
2	1000_0001_0000_0101	// lw \$2, 5(\$zero)	⇒	#0x8105	// pc=0010
3	1100_0101_0000_0010	// beq \$1,\$2,jump	⇒	#0xC502	// pc=0100
4	0000_0101_0011_0001	// sub \$3,\$1,\$2	⇒	#0x531	// pc=0110
5	1010_0001_1000_0111	// sw \$3, 7(\$zero)	⇒	#0xA187	// pc=1000
6	0000_0101_0011_0000	// jump:add \$3, \$1, \$2	⇒	#0x530	// pc=1010
7	1010_0001_1000_0111	// sw \$3, 7(\$zero)	⇒	#0xA187	// pc=1100
8	0000_0000_0000_0000	//			
9	0000_0000_0000_0000	//			
10	0000_0000_0000_0000	//			
11	0000_0000_0000_0000	//			
12	0000_0000_0000_0000	//			
13	0000_0000_0000_0000	//			
14	0000_0000_0000_0000	//			
15	0000_0000_0000_0000	//			

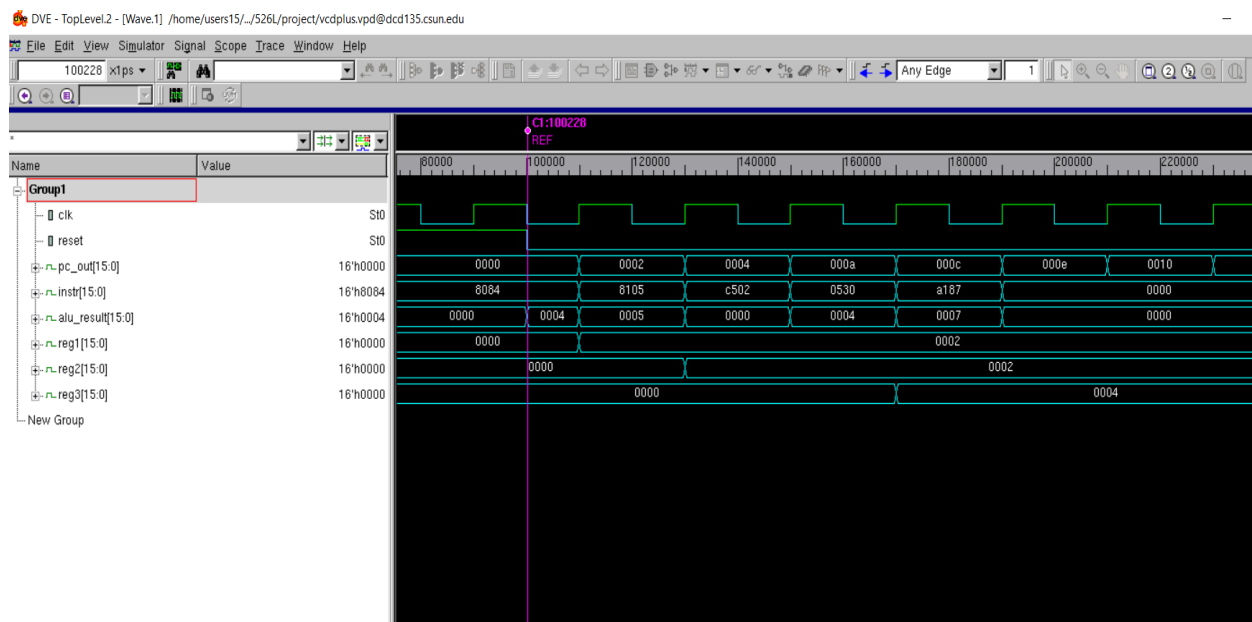
```

ram0= 0
ram1= 0
ram2= 0
ram3= 0
ram4= 2
ram5= 2
ram6= 0
ram7= 0
ram8= 0
ram9= 0
ram10= 0
ram11= 0
ram12= 0
ram13= 0
ram14= 0
ram15= 0

instruction =8084,register 1= 0, register 2= 0,register 3= 0
instruction =8105,register 1= 2, register 2= 0,register 3= 0
instruction =c502,register 1= 2, register 2= 2,register 3= 0
instruction =0530,register 1= 2, register 2= 2,register 3= 0
instruction =a187,register 1= 2, register 2= 2,register 3= 4
instruction =0000,register 1= 2, register 2= 2,register 3= 4

ram7= 4

```



As you can see each instruction is converted to the machine code and loaded into instruction memory. when we run the program(with simv) it shows us whatever is loaded in data memory for this scenario we put equal values (2)in to demonstrate that beq is working well.

lw \$1, 4(\$zero) ==> #0x8084 //pc=0000: This instruction loads a word from memory into register \$1. It retrieves the value (2) from the memory address 4 plus the value of register \$zero (which always holds the value 0).

lw \$2, 5(\$zero) ==> #0x8105 //pc=0010: This instruction loads a word from memory into register \$2. It retrieves the value (2) from the memory address 5 plus the value of register \$zero.

beq \$1,\$2,jump ==> #0xC502 //pc=0100 : This instruction is a branch equal (beq) instruction. It compares the values (2,2) in registers \$1 and \$2. If they are equal, it jumps to the label or address specified by jump if not next instruction will be run.

bren

the way we calculate the “jump” address in the mips_16.v file

//Jump address=>pc=1010: since we need to jump 2 instructions then we put 2(010) in the address branch after ex_sign(7 bites to 16 bites) we shift it by one (0000000000000100) then we add to pc_next(0110)=0000 0000 0000 0000 1010

```
assign im_shift_1 = {imm_ext[14:0],1'b0};
```

```
// PC beq&bne add
```

```
assign PC_beq = pc_2 + im_shift_1 ;
```

```
assign PC_bne = pc_2 + im_shift_1 ; // beq control
```

```
assign beq_control = beq & zero_flag;
```

```
assign bne_control = bne & (~zero_flag);
```

jump:add \$3, \$1, \$2==> #0x530 //pc=1010: since we have equal value in register \$1, \$2 then this instruction will be executed next and add the value register \$1 and \$2 and store the result(4) in register \$3

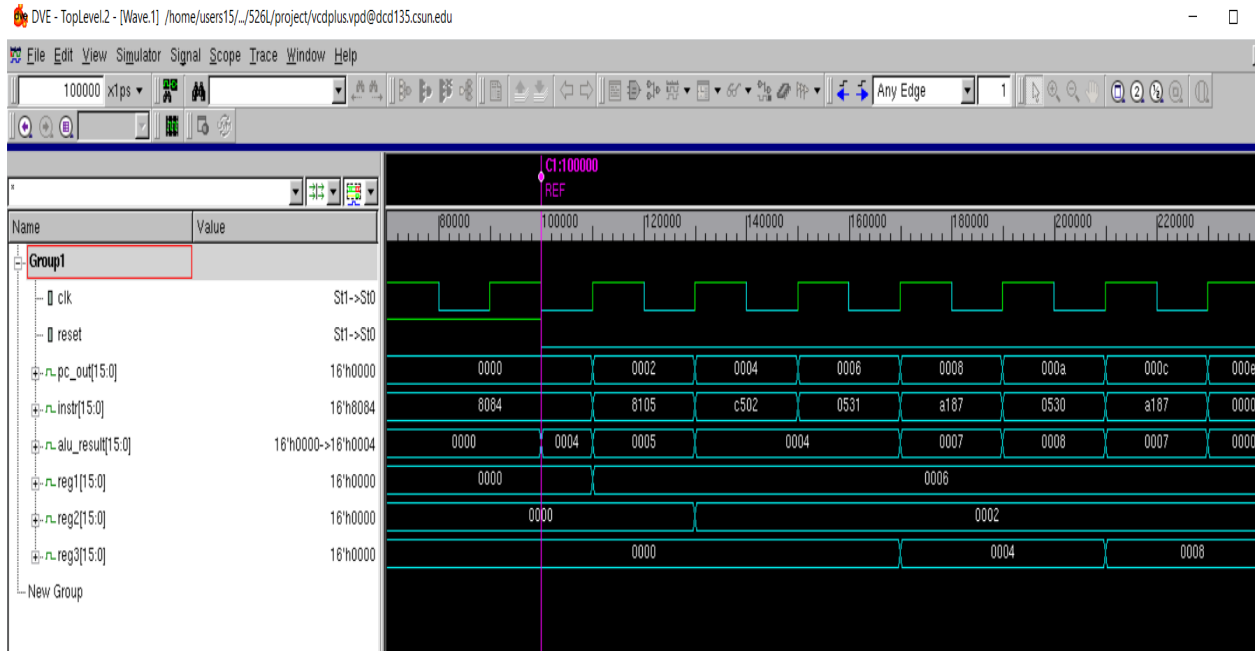
sw \$3, 7(\$zero) ==> #0xA187 //pc=1100: This instruction stores the value (4) from register \$3 into memory. It stores the value at the memory address 7 plus the value of register \$zero.

Ram4=6,ram5=2

```
rom.txt
1 1000_0000_1000_0100 // lw $1, 4($zero) ==> #0x8084 //pc=0000
2 1000_0001_0000_0101 // lw $2, 5($zero) ==> #0x8105 //pc=0010
3 1100_0101_0000_0010 // beq $1,$2,jump ==> #0xC502 //pc=0100
4 0000_0101_0011_0001 // sub $3,$1,$2 ==> #0x531 //pc=0110
5 1010_0001_1000_0111 // sw $3, 7($zero) ==> #0xA187 //pc=1000
6 0000_0101_0011_0000 // jump:add $3, $1, $2 ==> #0x530 //pc=1010
7 1010_0001_1000_0111 // sw $3, 7($zero) ==> #0xA187 //pc=1100
8 0000_0000_0000_0000 //
9 0000_0000_0000_0000 //
10 0000_0000_0000_0000 //
11 0000_0000_0000_0000 //
12 0000_0000_0000_0000 //
13 0000_0000_0000_0000 //
14 0000_0000_0000_0000 //
15 0000_0000_0000_0000 //
```

```
ram0= 0
ram1= 0
ram2= 0
ram3= 0
ram4= 6
ram5= 2
ram6= 0
ram7= 0
ram8= 0
ram9= 0
ram10= 0
ram11= 0
ram12= 0
ram13= 0
ram14= 0
ram15= 0

instruction =8084,register 1= 0, register 2= 0,register 3= 0
instruction =8105,register 1= 6, register 2= 0,register 3= 0
instruction =c502,register 1= 6, register 2= 2,register 3= 0
instruction =0531,register 1= 6, register 2= 2,register 3= 0
instruction =a187,register 1= 6, register 2= 2,register 3= 4
instruction =0530,register 1= 6, register 2= 2,register 3= 4
instruction =a187,register 1= 6, register 2= 2,register 3= 8
instruction =0000,register 1= 6, register 2= 2,register 3= 8
ram7= 8
```



lw \$1, 4(\$zero) ==> #0x8084 //pc=0000: This instruction loads a word from memory into register \$1. It retrieves the value (6) from the memory address 4 plus the value of register \$zero (which always holds the value 0).

lw \$2, 5(\$zero) ==> #0x8105 //pc=0010: This instruction loads a word from memory into register \$2. It retrieves the value (2) from the memory address 5 plus the value of register \$zero.

beq \$1,\$2,jump ==> #0xC502 //pc=0100: This instruction is a branch equal (beq) instruction. It compares the values (6,2) in registers \$1 and \$2. If they are equal, it jumps to the label or address specified by jump if not next instruction will be run.

sub \$3,\$1,\$2 ==> #0x531 //pc=0110: since (6,2) are not equal this instruction will be run. This instruction subtracts the value in register \$2(6) from the value in register \$1(2) and stores the result (4) in register \$3.

sw \$3, 7(\$zero) ==> #0xA187 //pc=1000: This instruction is similar to the previous store instruction. It stores the value(4) from register \$3 into memory at the memory address 7 plus the value of register \$zero.

jump: add \$3, \$1, \$2 ==> #0x530 //pc=1010: add the value register \$1 and \$2 and store the result(8) in register \$3


sw \$3, 7(\$zero) ==> #0xA187 //pc=1100: This instruction stores the value (8) from register \$3 into memory. It stores the value at the memory address 7 plus the value of register \$zero.

I hereby attest that this lab report is entirely my own work. I have not copied either code or text from anyone, nor have I allowed or will I allow anyone to copy my work.

Name (printed)

Fred Ryan

Name (signed)



Date

5,17,2023