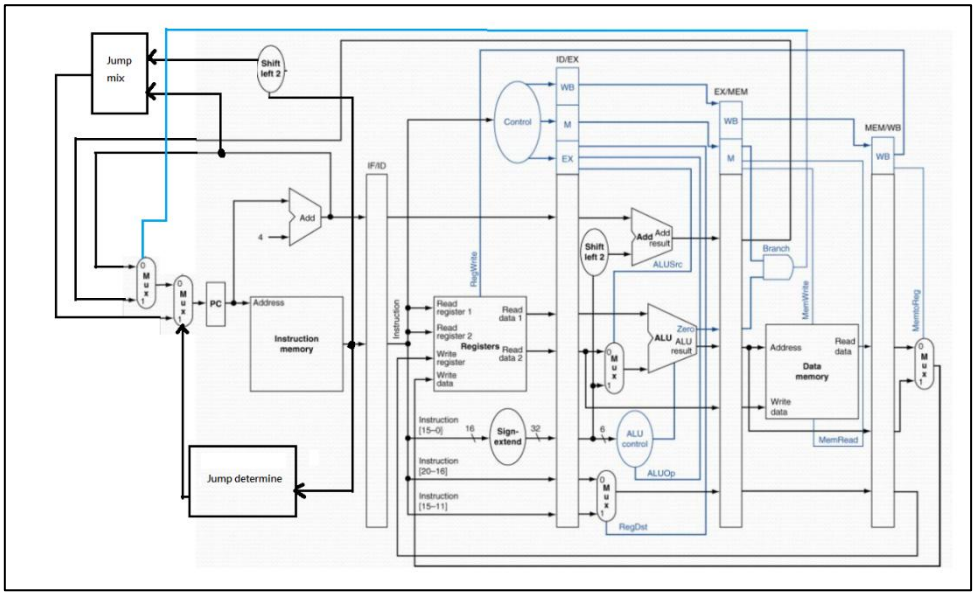
B. Pipeline implementation

1. Introduction

Pipeline implementation differs from single-cycle implementation by dividing the CPU into 5 stages. In each stage, pipeline registers are introduced to store values passed down from the previous stage. As a result, in pipeline implementation, multiple instructions can be executed simultaneously.

2. Block diagram



3. Example code

Due to the potential for hazards in pipeline implementation, hazard handling requires both a Hazard Detection Unit and a Forwarding Unit. To avoid the occurrence of hazards, we design the example code in a way that eliminates the possibility of hazards. However, to execute jump instructions, we design a block in the IF Stage that detects jumps, which helps prevent jump hazards.

```
Loop:
    lw $t1 ,0($t3)
    addi $t5 ,$t2 , 4
    sub $t6 ,$t2 ,$t3
    or $t7 ,$t1 ,$t4
    and $t3,$t1,$t5
    beq $t0 ,$t2,LOOP
    add $t2 ,$t2 ,$t4
    sw $t1 ,20($t3)
    slt $t4,$t0,$t5
    j LOOP
```

4. Main code

(1) PC

The address of the **PC** is determined by three possible scenarios: **Branch**, **Jump**, and the default case, which is **PC + 4**. Below is an explanation of how to calculate the address for each scenario, followed by an explanation of how to select the next **PC** address.

I. PC+4

If no **Branch** or **Jump** occurs, the next instruction is executed, and the **PC** is incremented by 4.

```
module PC_Add4(PC, PC_4);

input [31:0]PC;
output [31:0]PC_4;
assign PC_4 = PC + 4;

endmodule
```

II. Branch

The Branch address is composed of (PC + 4) + address, forming a 32-bit address. The original 16-bit address is sign-extended to a 32-bit word address, and then shifted left by 2 to become a 32-bit byte address.

- Branch address

```
module Branch_Add(branch_byte, PC_4, branch_out);

input [31:0]branch_byte, PC_4;
output [31:0]branch_out;

assign branch_out = branch_byte + PC_4;

endmodule
```

- Shift-left-2

```
module shift_left_2_I_format(address_word,address_byte);

input [31:0] address_word;
output [31:0] address_byte;

assign address_byte = address_word << 2;

endmodule
```

III. Jump

The Jump address is composed of {PC + 4 [31:28], address[27:0]}. The original 26-bit address (word address) is shifted left by 2 to become a 28-bit byte address, resulting in a total of 32 bits.

- Jump address

```
module jump_mix (pc_4_3128,address_in,jump_address);

input [3:0] pc_4_3128;
input [31:0] address_in;
output reg [31:0] jump_address;

always@(*)
    begin
        jump_address <= {pc_4_3128,address_in};
    end

endmodule
```

- shift-left-2

```
module shift_left_2_jump(address_word,address_byte);

input [31:0] address_word;
output [31:0] address_byte;

assign address_byte = address_word << 2;

endmodule
```

IV. The selection of the PC

The next instruction's address is determined through Branch_MUX and Jump_MUX.

Branch_MUX is controlled by Branch_AND, which uses the Branch and Zero signals to decide whether a branch will occur.

Jump_MUX is controlled by Jump_determine, which uses the instruction's opcode (opcode = 000010) to determine whether a jump will occur.

- Branch_MUX

```
module branch_mux(pc_4,branch_add_result,branch_and,branch_mux_out);

input [31:0]pc_4,branch_add_result;
input branch_and;
output reg [31:0] branch_mux_out;
```

```
always@(*)
	begin
		if(branch_and)
			branch_mux_out <= branch_add_result;
		else
			branch_mux_out <= pc_4;
		end
endmodule
```

- Branch_AND

```
module Branch_AND (a,b,branch_and_out);

input a,b;
output branch_and_out;

assign branch_and_out = a & b;

endmodule
```

- Jump_MUX

```
module jump_mux(jump,jump_address,branch_mux_out,pc_next);

input[31:0] jump_address,branch_mux_out;
input jump;
output    reg [31:0] pc_next;

always@(*)
	begin
		if(jump)
			pc_next <= jump_address;
		else
			pc_next <= branch_mux_out;
	end

endmodule
```

- Jump_determine

```
module jump_determine(instructions,jump_or_not);

input [5:0] instructions;
output reg jump_or_not;
```

```
always@(instructions)
    begin
        if(instructions == 6'b000010)
            jump_or_not <= 1'b1;
        else
            jump_or_not <= 1'b0;
    end

endmodule
```

(2) Register file

What differs in the Register file is WriteReg and RegWrite. In a pipeline implementation, since multiple instructions are executed simultaneously, these two signals need to be stored in the pipeline registers. In the WB stage, these signals are used to write WriteData into WriteReg. During the write-back process, the instruction decoding process is not affected because the first half of the Register file handles the WB stage, while the second half handles the ID stage.

(3) Pipeline register

The results computed by each stage are temporarily stored in the pipeline registers.

- IF/ID pipeline register

```
module IF_ID_pipeline_register(clk,pc_4_in,instruction_in,pc_4_out,instruction_out);

input clk;

input [31:0] pc_4_in;
input [31:0] instruction_in;

output reg [31:0] pc_4_out;
output reg [31:0] instruction_out;

always@(posedge clk)
    begin
        pc_4_out <= pc_4_in;
        instruction_out <= instruction_in;
    end

endmodule
```

- ID/EX pipeline register

```
module
ID_EX_pipeline_register(clk,ID_EX_RegDst_in,ID_EX_ALUsrc_in,ID_EX_MemtoReg_in,ID_EX_RegWrite
```

```
_in,ID_EX_MemRead_in,ID_EX_MemWrite_in,ID_EX_Branch_in,ID_EX_ALUop_in,

ID_EX_RegDst_out,ID_EX_ALUsrc_out,ID_EX_MemtoReg_out,ID_EX_RegWrite_out,ID_EX_MemRead_
out,ID_EX_MemWrite_out,ID_EX_Branch_out,ID_EX_ALUop_out,
                                        read_data1_in,read_data2_in,read_data1_out,read_data2_out,
                                      ID_EX_RegisterRt_in,ID_EX_RegisterRd_in,
                                      ID_EX_RegisterRt_out,ID_EX_RegisterRd_out,
                                        sign_extend_in,sign_extend_out,
                                        pc_in,pc_out
                                    );

input clk;

// EX
input ID_EX_RegDst_in,ID_EX_ALUsrc_in;
input [1:0] ID_EX_ALUop_in;
// MEM
input ID_EX_MemRead_in,ID_EX_MemWrite_in,ID_EX_Branch_in;
//WB
input ID_EX_MemtoReg_in,ID_EX_RegWrite_in;

output reg
ID_EX_RegDst_out,ID_EX_ALUsrc_out,ID_EX_MemtoReg_out,ID_EX_RegWrite_out,ID_EX_MemRead_
out,ID_EX_MemWrite_out,ID_EX_Branch_out;
output reg [1:0] ID_EX_ALUop_out;

input [31:0] read_data1_in,read_data2_in;
output reg [31:0] read_data1_out,read_data2_out;

input [4:0] ID_EX_RegisterRt_in,ID_EX_RegisterRd_in;
output reg [4:0] ID_EX_RegisterRt_out,ID_EX_RegisterRd_out;

input [31:0] sign_extend_in;
output reg [31:0] sign_extend_out;

input [31:0] pc_in;
output reg[31:0] pc_out;

always@(posedge clk)
    begin
        // Control //
        // EX
```

```
        ID_EX_RegDst_out <= ID_EX_RegDst_in;
        ID_EX_ALUsrc_out <= ID_EX_ALUsrc_in;
        ID_EX_ALUop_out <= ID_EX_ALUop_in;
        // MEM
        ID_EX_MemRead_out <= ID_EX_MemRead_in;
        ID_EX_MemWrite_out <= ID_EX_MemWrite_in;
        ID_EX_Branch_out <= ID_EX_Branch_in;
        // WB
        ID_EX_MemtoReg_out <= ID_EX_MemtoReg_in;
        ID_EX_RegWrite_out <= ID_EX_RegWrite_in;
        ////////////////////////////////////////////////////////////////////////////////
        //ID_EX_RegisterRs_out <= ID_EX_RegisterRs_in;
        ID_EX_RegisterRt_out <= ID_EX_RegisterRt_in;
        ID_EX_RegisterRd_out <= ID_EX_RegisterRd_in;
        ////////////////////////////////////////////////////////////////////////////////
        read_data1_out <= read_data1_in;
        read_data2_out <= read_data2_in;
        sign_extend_out <= sign_extend_in;
        pc_out <= pc_in;
    end

endmodule
```

- EX/MEM pipeline register

```
module EX_MEM_pipeline_register(clk, Regwrite_in, MemtoReg_in, Branch_in, MemRead_in,
MemWrite_in, Add_branch_in, Zero_in, ALU_result_in, Write_mem_in, Write_reg_in
, Regwrite_out, MemtoReg_out, Branch_out, MemRead_out, MemWrite_out, Add_branch_out,
Zero_out, ALU_result_out, Write_mem_out, Write_reg_out);

input clk, Regwrite_in, MemtoReg_in, Branch_in, MemRead_in, MemWrite_in, Zero_in;
input [31:0]Add_branch_in, ALU_result_in, Write_mem_in;
input [4:0]Write_reg_in;
output reg Regwrite_out, MemtoReg_out, Branch_out, MemRead_out, MemWrite_out, Zero_out;
output reg [31:0] Add_branch_out, ALU_result_out, Write_mem_out;
output reg [4:0]Write_reg_out;

always @(posedge clk)
    begin
        Regwrite_out <= Regwrite_in;
        MemtoReg_out <=MemtoReg_in;
        Branch_out <= Branch_in;
        MemRead_out <= MemRead_in;
```

```
        MemWrite_out <= MemWrite_in;
        Add_branch_out <= Add_branch_in;
        Zero_out <= Zero_in;
        ALU_result_out <= ALU_result_in;
        Write_mem_out <= Write_mem_in;
        Write_reg_out <= Write_reg_in;
    end

endmodule
```

- MEM/WB pipeline register

```
module MEM_WB_pipeline_register(clk, Regwrite_in, MemtoReg_in, Read_data_in, ALU_result_in,
Write_reg_in,
Regwrite_out, MemtoReg_out, Read_data_out, ALU_result_out, Write_reg_out);

input clk, Regwrite_in, MemtoReg_in;
input [31:0]Read_data_in, ALU_result_in;
input [4:0]Write_reg_in;
output reg Regwrite_out, MemtoReg_out;
output reg[31:0] Read_data_out, ALU_result_out;
output reg[4:0]Write_reg_out;

always @(posedge clk)
    begin
        Regwrite_out <= Regwrite_in;
        MemtoReg_out <= MemtoReg_in;
        Read_data_out <= Read_data_in;
        ALU_result_out <= ALU_result_in;
        Write_reg_out <= Write_reg_in;
    end

endmodule
```

5. <u>Testbench</u>

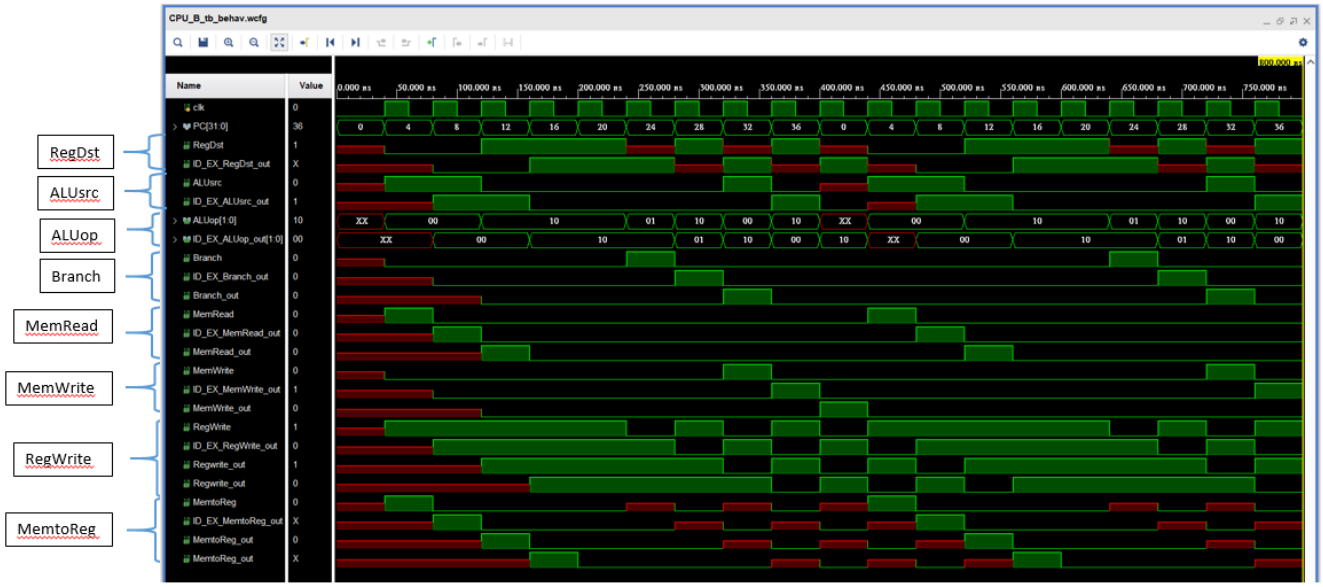
```
`timescale 1ns / 1ps

module CPU_B_tb;

reg clk;
wire [31:0] PC;

CPU_B test( .clk(clk) , .PC(PC));
```

```
initial
	begin
		clk = 1'b0;
		#20;
		forever #20 clk = ~clk;
	end

initial
	begin
		#800; $finish;
	end

endmodule
```

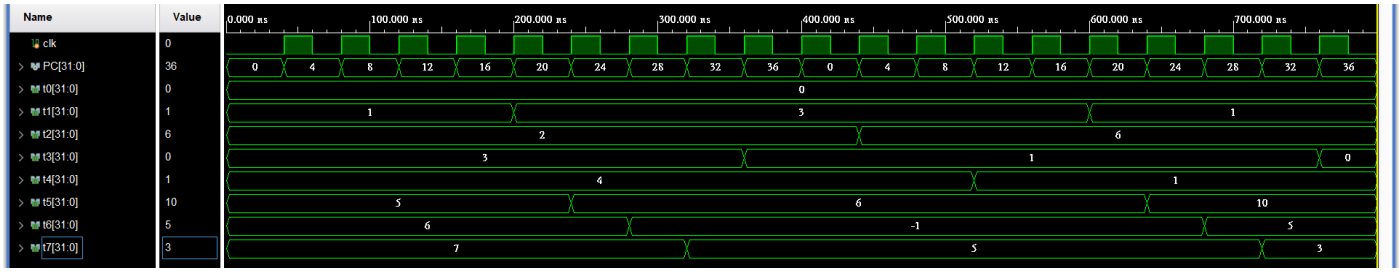
6. Waveform

In a Pipeline Implementation, it is possible to observe whether the Control signals are passed to the next stage for decision-making.

- Control



- Changes in Register($t0-$t7)



- Changes in DataMemory