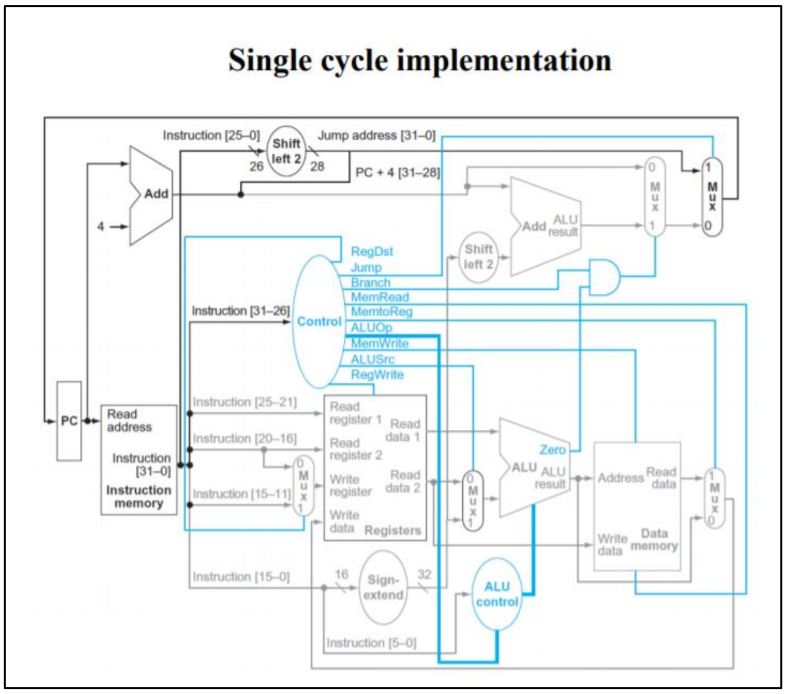
## A. Single cycle implementation

### 1. Introduction

The characteristic of a single-cycle implementation is that each instruction is fully executed before the next instruction begins.

### 2. Block diagram



### 3. Example code

```
Loop:
        lw $t1 ,0($t3)
        addi $t5 ,$t2 , 4
        sub $t6 ,$t1 ,$t3
        or $t7 ,$t5 ,$t6
        and $t3,$t1,$t5
        beq $t6 ,$t7,LOOP2
        j LOOP2
Loop2:
        add $t2 ,$t2 ,$t4
        sw $t2 ,20($t3)
        slt $t4,$t0,$t5
        j LOOP
```

### 4. Main Code

(1) CPU & PC(Main module)

CPU_A is the main module that calls other modules, and the Program Counter (PC) is also defined within this module. The PC address is initially set to 0 (i.e., the first instruction).

When a jump occurs, the jump address is calculated as {PC+4[31:28], address}. The original 26-bit address (a word address) is shifted left by 2 bits to become a 28-bit byte address, resulting in a complete 32-bit jump address.

When a branch occurs (determined based on the Branch and Zero signals), the branch address is calculated as (PC + 4) + offset. The original 16-bit address is first sign-extended to a 32-bit word address, then shifted left by 2 bits to become a 32-bit byte address.

In all other cases, the PC is simply updated to PC + 4.

```
module CPU_A(clk,PC);

input clk;
wire [31:0]ins_out;
wire RegDst ,Jump, ALUsrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch;
wire [1:0] ALUop;
wire [4:0]write_reg;
wire [31:0]write_data, read_data1, read_data2, address_out, alu_mux_out;
wire [31:0]ALU_out;
wire zero;
wire [3:0]ALU_ctrl;
wire [31:0]DataMem_read_data;
wire [31:0] t0,t1,t2,t3,t4,t5,t6,t7;

output reg [31:0] PC;

initial
    begin
        PC <= 0;
    end

always@(posedge clk)
    begin
        if(Jump)
            begin
                PC <= PC + 4;
                PC <= { PC [31:28] , ins_out [25:0] , 2'b00 } ;
            end
        else
            if(Branch && zero)
                begin
                    PC <= PC + 4;
```

```
                              PC <= PC +    (address_out << 2);
                         end
                    else
                         PC <= PC + 4;
       end


ins_memory U2(.clk(clk), .read_address(PC), .instruction(ins_out));

control U3(.signal(ins_out[31:26]), .RegDst(RegDst), .Jump(Jump), .ALUsrc(ALUsrc),
.MemtoReg(MemtoReg), .RegWrite(RegWrite), .MemRead(MemRead), .MemWrite(MemWrite), .Branc
h(Branch), .ALUop(ALUop));

register_file U4(.clk(clk) , .RegWrite(RegWrite), .read_reg1(ins_out[25:21]),
                              .read_reg2(ins_out[20:16]), .write_reg(write_reg), .write_data(write_data),
                              .read_data1(read_data1), .read_data2(read_data2),
                              .t0(t0),.t1(t1),.t2(t2),.t3(t3),.t4(t4),.t5(t5),.t6(t6),.t7(t7)
                              );

write_register_mux U5
(.RegDst(RegDst), .rd(ins_out[15:11]), .rt(ins_out[20:16]), .write_reg(write_reg));

sign_extend U6(.address_in(ins_out[15:0]), .address_out(address_out));

reg2alu_mux U7(.alusrc(ALUsrc), .reg2(read_data2), .extended(address_out), .muxo(alu_mux_out));

alu U8(.alu_ctr(ALU_ctrl), .a(read_data1), .b(alu_mux_out), .alu_out(ALU_out), .zero(zero));

alu_control U9(.func(ins_out[5:0]), .aluop(ALUop), .alu_Ctr(ALU_ctrl));

data_memoryU10(.clk(clk), .MemWrite(MemWrite), .MemRead(MemRead), .address(ALU_out), .write
_data(read_data2), .read_data(DataMem_read_data)
                         );
write_data_muxU11(.MemtoReg(MemtoReg), .read_data(DataMem_read_data), .ALU_result(ALU_out
), .write_data(write_data));

endmodule
```

(2) Instruction memory

First, the example code is written into the instruction memory. Then, the instruction to be executed is obtained through the value of read_address(PC).

Since the PC uses a byte address while the instruction memory is indexed by instruction number, the read_address must be divided by 4 to retrieve the correct instruction. ◦

```
module ins_memory(clk,read_address,instruction);

input clk;
input [31:0] read_address;
output reg [31:0] instruction;

reg [31:0] instruction_memoroy [31:0];

initial
	begin
		// Loop
		instruction_memoroy[0] = 32'b100011_01011_01001_0000000000000000;
// lw $t1 ,0($t3)
		instruction_memoroy[1] = 32'b001000_01010_01101_0000000000000100;
// addi $t5 ,$t2 , 4
		instruction_memoroy[2] = 32'b000000_01001_01011_01110_00000_100010;
// sub $t6 ,$t1 ,$t3
		instruction_memoroy[3] = 32'b000000_01101_01110_01111_00000_100101;
// or $t7 ,$t5 ,$t6
		instruction_memoroy[4] = 32'b000000_01001_01101_01011_00000_100100;
// and $t3,$t1,$t5
		instruction_memoroy[5] = 32'b000100_01110_01111_0000000000000001;
// beq $t6 ,$t7,LOOP2
		instruction_memoroy[6] = 32'b000010_00000000000000000000000111;
// j LOOP2
		// Loop2
		instruction_memoroy[7] = 32'b000000_01010_01100_01010_00000_100000;
// add $t2 ,$t2 ,$t4
		instruction_memoroy[8] = 32'b101011_01011_01010_0000000000010100;
// sw $t2 ,20($t3)
		instruction_memoroy[9] = 32'b000000_01000_01101_01100_00000_101010;
// slt $t4,$t0,$t5
		instruction_memoroy[10] = 32'b000010_00000000000000000000000000;
// j LOOP
	end

always@(*)
```

```
        begin
                instruction <= instruction_memoroy[read_address/4];
        end

endmodule
```

(3) Register file & WriteReg_MUX & WriteData_MUX

- Register file
  First, initialize the variables that will be used (i.e., registers $t0 to $t7). Then, based on read_reg1 and read_reg2, the corresponding values read_data1 and read_data2 are retrieved.
  The write_reg is determined by the result of the write_reg_mux, and the write_data obtained from the write_data_mux is written into the specified register.

```
module
register_file(clk,RegWrite,read_reg1,read_reg2,write_reg,write_data,read_data1,read_data2,t0,t1,t2,t
3,t4,t5,t6,t7);

input clk;

input RegWrite; // Control

input [4:0] read_reg1;
input [4:0] read_reg2;
input [4:0] write_reg;

input [31:0] write_data;
output reg [31:0] read_data1;
output reg [31:0] read_data2;

output reg [31:0] t0,t1,t2,t3,t4,t5,t6,t7;

initial
        begin
                t0 = 0;
                t1 = 1;
                t2 = 2;
                t3 = 3;
                t4 = 4;
                t5 = 5;
                t6 = 6;
                t7 = 7;
        end
```

```
always@(*)
	begin
		case(read_reg1)
		8 : read_data1 = t0;
		9 : read_data1 = t1;
		10 : read_data1 = t2;
		11 : read_data1 = t3;
		12 : read_data1 = t4;
		13 : read_data1 = t5;
		14 : read_data1 = t6;
		15 : read_data1 = t7;
		default : read_data1 = 32'dX;
		endcase
	end

always@(*)
	begin
		case(read_reg2)
		8 : read_data2 = t0;
		9 : read_data2 = t1;
		10 : read_data2 = t2;
		11 : read_data2 = t3;
		12 : read_data2 = t4;
		13 : read_data2 = t5;
		14 : read_data2 = t6;
		15 : read_data2 = t7;
		default : read_data2 = 32'dX;
		endcase
	end

always@(posedge clk)
	begin
		if(RegWrite)
			begin
				case(write_reg)
					8 : t0 <= write_data;
					9 : t1 <= write_data;
					10 : t2 <= write_data;
					11 : t3 <= write_data;
					12 : t4 <= write_data;
					13 : t5 <= write_data;
					14 : t6 <= write_data;
```

```
                    15 : t7 <= write_data;
                endcase
            end
        else
            begin
                t0 <= t0;
                t1 <= t1;
                t2 <= t2;
                t3 <= t3;
                t4 <= t4;
                t5 <= t5;
                t6 <= t6;
                t7 <= t7;
            end
    end

endmodule
```

- WriteReg_MUX

  The value of WriteReg is determined based on the control signal RegDst.

```
module write_register_mux(RegDst,rd,rt,write_reg);

input RegDst;
input [4:0] rd; // R-type destination
input [4:0] rt;

output reg [4:0] write_reg;

always@(*)
    begin
        case(RegDst)
            0 : write_reg = rt;
            1 : write_reg = rd; // R-type
            default : write_reg = 5'dX;
        endcase
    end

endmodule
```

- WriteData_MUX

  The value of write_data is determined based on the control signal MemtoReg.

```
module write_data_mux(MemtoReg,read_data,ALU_result,write_data);

input MemtoReg;

input [31:0] read_data; // lw
input [31:0] ALU_result; // R-format

output reg [31:0] write_data;

always@(*)
	begin
		case(MemtoReg)
			0 : write_data = ALU_result; // R-format
			1 : write_data = read_data; // lw
			default : write_data = 32'dX;
		endcase
	end

endmodule
```

(4) Control

Based on the opcode, the values of the various control signals in the Control unit are output.

```
module control(signal,
RegDst,Jump,ALUsrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUop);

input [5:0] signal;
output reg RegDst,ALUsrc,Jump,MemtoReg,RegWrite,MemRead,MemWrite,Branch;
output reg[1:0] ALUop;

always@(signal)
	begin
		case(signal)
			6'b000000:begin //R format
						RegDst=1'b1;
						ALUsrc=1'b0;
						Jump=1'b0;
						MemtoReg=1'b0;
						RegWrite=1'b1;
```

```
                MemRead=1'b0;
                MemWrite=1'b0;
                Branch=1'b0;
                ALUop=2'b10;
            end
6'b100011:begin // lw
                RegDst=1'b0;
                ALUsrc=1'b1;
                Jump=1'b0;
                MemtoReg=1'b1;
                RegWrite=1'b1;
                MemRead=1'b1;
                MemWrite=1'b0;
                Branch=1'b0;
                ALUop=2'b00;
            end
6'b101011:begin // sw
                RegDst=1'bX;
                ALUsrc=1'b1;
                Jump=1'b0;
                MemtoReg=1'bX;
                RegWrite=1'b0;
                MemRead=1'b0;
                MemWrite=1'b1;
                Branch=1'b0;
                ALUop=2'b00;
            end
6'b000100:begin // beq
                RegDst=1'bX;
                ALUsrc=1'b0;
                Jump=1'b0;
                MemtoReg=1'bX;
                RegWrite=1'b0;
                MemRead=1'b0;
                MemWrite=1'b0;
                Branch=1'b1;
                ALUop=2'b01;
            end
6'b001000:begin // addi
                RegDst=1'b0;
                ALUsrc=1'b1;
                Jump=1'b0;
```

```
                        MemtoReg=1'b0;
                        RegWrite=1'b1;
                        MemRead=1'b0;
                        MemWrite=1'b0;
                        Branch=1'b0;
                        ALUop=2'b00;
                    end
            6'b000010:begin // jump
                        RegDst=1'bX;
                        ALUsrc=1'bX;
                        Jump=1'b1;
                        MemtoReg=1'bX;
                        RegWrite=1'b0;
                        MemRead=1'b0;
                        MemWrite=1'b0;
                        Branch=1'b0;
                        ALUop=2'bXX;
                    end
            default: begin
                        RegDst=1'bX;
                        ALUsrc=1'bX;
                        MemtoReg=1'bX;
                        RegWrite=1'bX;
                        MemRead=1'bX;
                        MemWrite=1'bX;
                        Branch=1'bX;
                        ALUop=2'bXX;
                    end
        endcase
    end

endmodule
```

(5) Sign-extend

The 16-bit address or immediate value is sign-extended to 32 bits before it can be used in the ALU operation.

```
module sign_extend(address_in,address_out);

input [15:0] address_in;
output [31:0] address_out;

assign address_out = { {16{address_in[15]}} , address_in}; //  正數補 0  負數補 1
```

```

endmodule
```

(6) ALU & ALU_Control & ALU_MUX

- ALU

  The ALU_Control signal determines the type of operation that the ALU will perform.

```
module alu( alu_ctr,a,b,alu_out,zero);


	input [31:0] a,b;
	input[3:0] alu_ctr;
	output reg [31:0] alu_out;
	output reg zero;

	always@(a or b or alu_ctr)
		begin
			case (alu_ctr)
				4'b0000: alu_out = a & b;
				4'b0001: alu_out = a | b;
				4'b0010: alu_out = a + b;
				4'b0110: alu_out = a - b;
				4'b0111: alu_out = a < b;
				4'b1100: alu_out = a ~^ b;
				default: alu_out = 32'dX;
			endcase
	end

	always@(alu_out)
		begin
			if (alu_out==32'd0)
				zero=1'b1;
			else
				zero=1'b0;
			end


endmodule
```

- ALU_Control

  The ALU_Control signal determines the type of operation that the ALU will perform.

```
module alu_control( func,aluop,alu_Ctr);
input [5:0] func;
input [1:0] aluop;
output reg [3:0] alu_Ctr;
```

```
always@(func or aluop)
	begin
		case (aluop)
			2'b00: alu_Ctr=4'b0010; // lw,sw
			2'b01: alu_Ctr=4'b0110; // beq
			2'b10:
				begin
					if(func==6'b100000) // add
						alu_Ctr=4'b0010;
					else if (func==6'b100010) // sub
						alu_Ctr=4'b0110;
					else if (func==6'b100100) // AND
						alu_Ctr=4'b0000;
					else if (func==6'b100101) // OR
						alu_Ctr=4'b0001;
					else if (func==6'b101010) // slt
						alu_Ctr=4'b0111;
					else
						alu_Ctr=4'b1111;
				end
			default: alu_Ctr=4'b1111;
		endcase
	end

endmodule
```

- ALU_MUX

  The value of ALUSrc determines the input to the ALU.

```
module reg2alu_mux(alusrc,reg2,extended,muxo);

input alusrc;
input [31:0]reg2,extended;
output reg [31:0] muxo;

always@(alusrc or reg2 or extended)
	begin
		if(alusrc)
			muxo = extended;
		else
			muxo = reg2;
```

```
        end

endmodule
```

(7) Data memory

First, set up a data_memory array containing 32 entries, each of which is 32 bits wide, and assign initial values to each data entry. Then, based on the result of the ALU operation and the values of MemWrite and MemRead, determine whether to read data from memory and write it to the register, or write data from the register to memory.

```
module data_memory(clk,MemWrite,MemRead,address,write_data,read_data);

input clk;

input MemWrite; // Control for sw
input MemRead; // Control for lw

input [31:0] address; //    rt address
input [31:0] write_data; // sw
output reg [31:0] read_data; // lw

reg [31:0] memory_data [31:0]; // 32 個 register 的 32-bit data

initial
        begin
                memory_data[0] = 0;
                memory_data[1] = 1;
                memory_data[2] = 2;
                memory_data[3] = 3;
                memory_data[4] = 4;
                memory_data[5] = 5;
                memory_data[6] = 6;
                memory_data[7] = 7;
                memory_data[8] = 0;
                memory_data[9] = 0;
                memory_data[10] = 0;
                memory_data[11] = 0;
                memory_data[12] = 0;
                memory_data[13] = 0;
                memory_data[14] = 0;
                memory_data[15] = 0;
                memory_data[16] = 0;
                memory_data[17] = 0;
                memory_data[18] = 0;
                memory_data[19] = 0;
                memory_data[20] = 0;
                memory_data[21] = 0;
                memory_data[22] = 0;
                memory_data[23] = 0;
                memory_data[24] = 0;
                memory_data[25] = 0;
                memory_data[26] = 0;
                memory_data[27] = 0;
                memory_data[28] = 0;
                memory_data[29] = 0;
                memory_data[30] = 0;
                memory_data[31] = 0;
        end

always@(negedge clk)
        begin
                if(MemWrite && ! MemRead) // sw
                     memory_data[address] <= write_data;
                else
                     memory_data[address] <= memory_data[address];
        end

always@(*)
        begin
                if(MemRead && !MemWrite) // lw
                    read_data = memory_data[address];
                else
                    read_data = 32'dX;
        end

endmodule
```

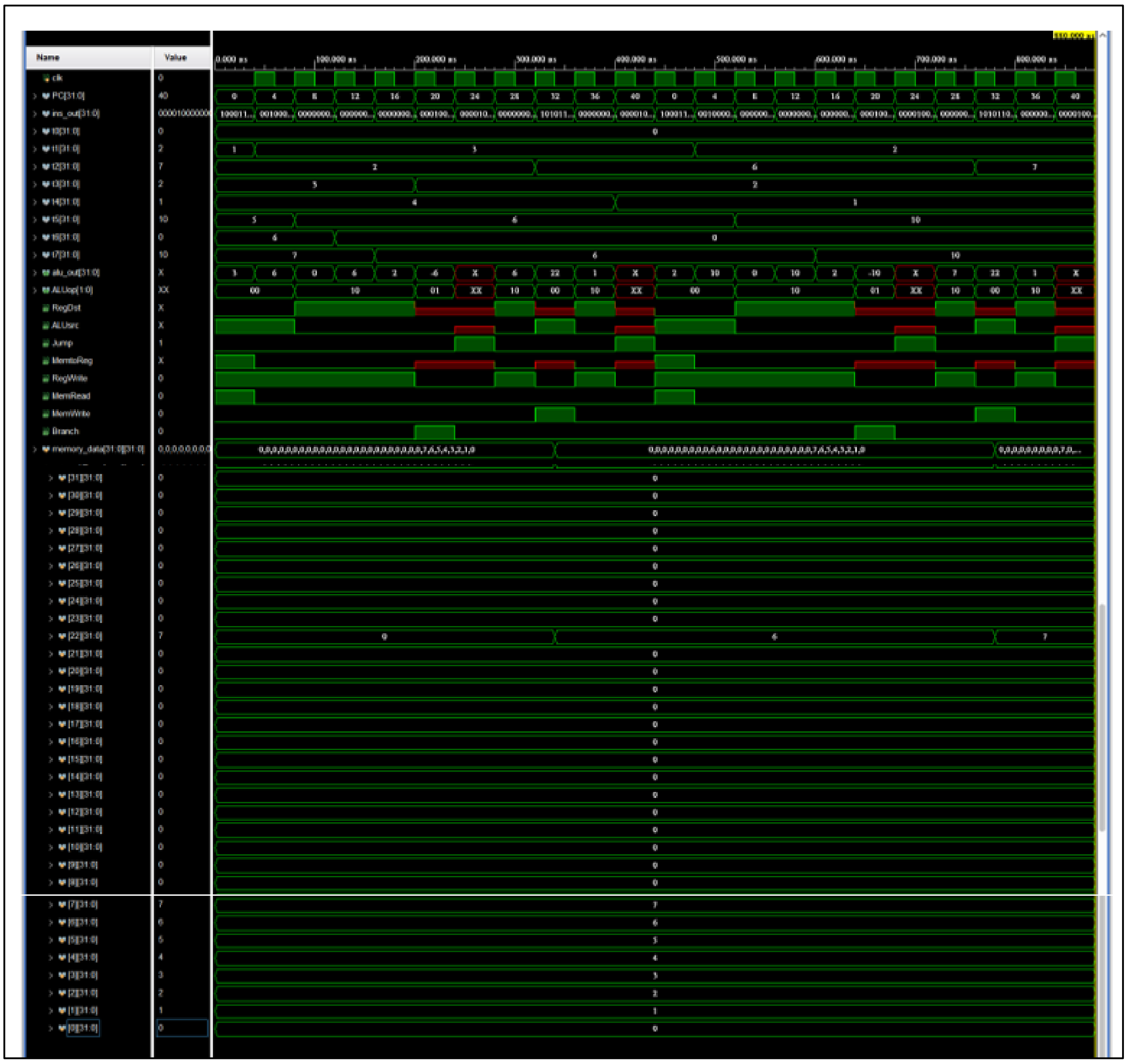
**5. Testbench**

```
`timescale 1ns / 1ps


module CPU_A_tb;


reg clk;
wire [31:0] PC;


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

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

initial
        begin
                #880; $finish;
        end

endmodule
```

## 6. <u>**Waveform**</u>

- Changes in Control, registers ($t0-$t7), and Data memory

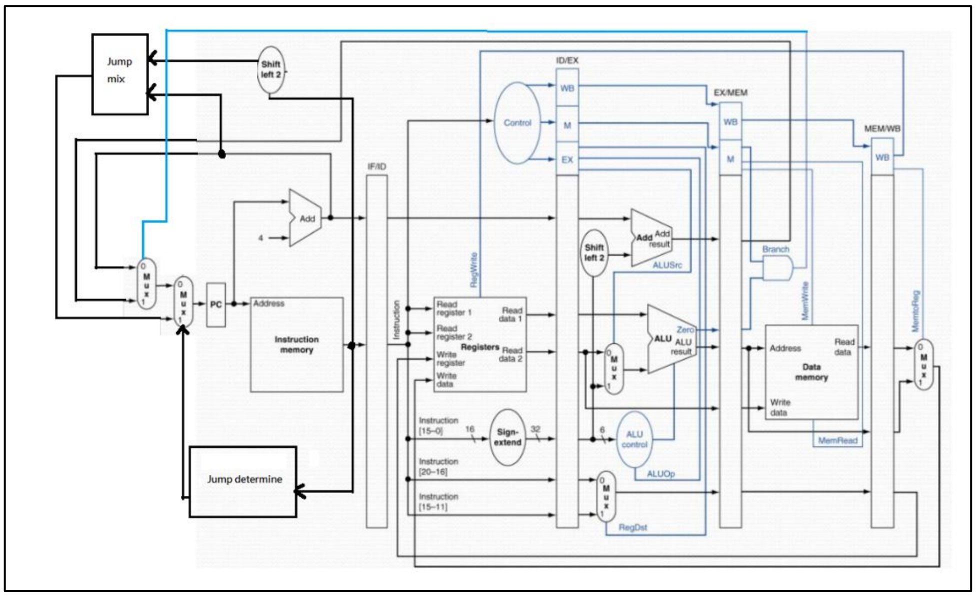


## B. Pipeline implementation

### 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>**

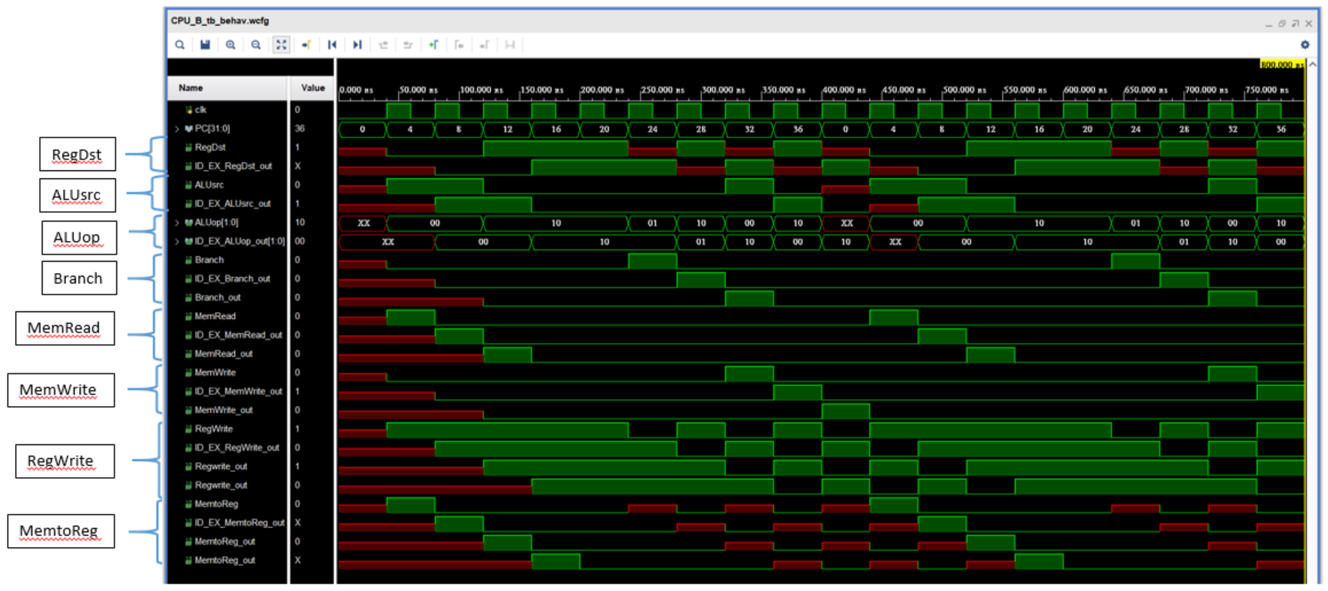
```
`timescale 1ns / 1ps


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
```

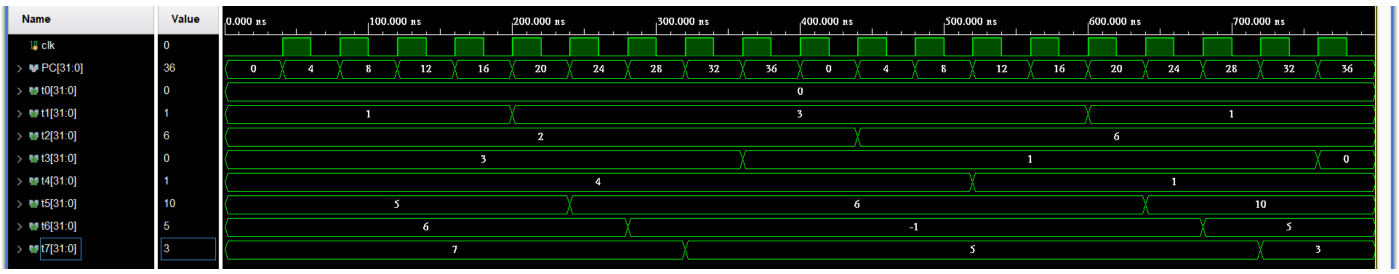
## 6. <u>Waveform</u>

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

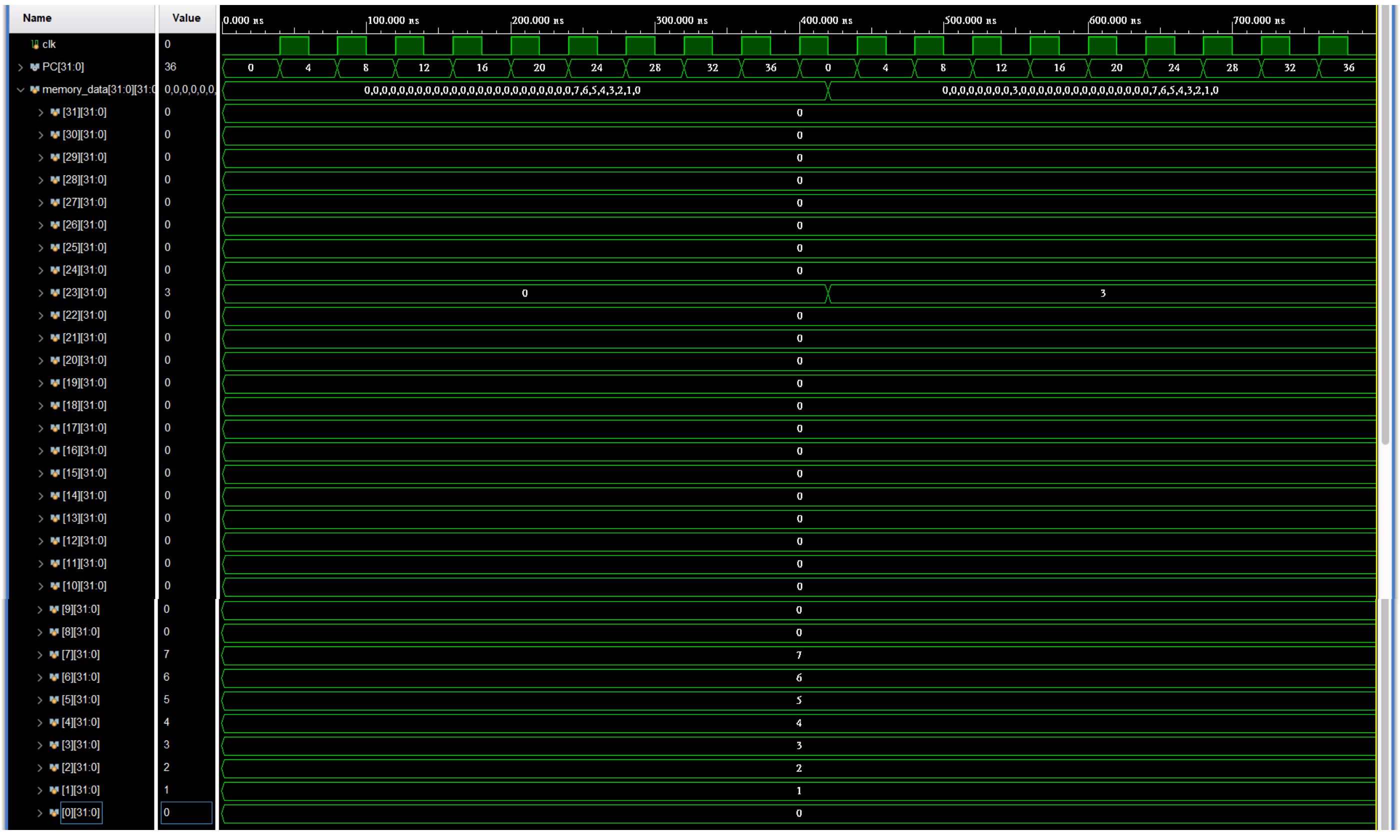
- Control



- Changes in Register($t0-$t7)



- Changes in DataMemory



| Name | Value |
|---|---|
| clk | 0 |
| PC[31:0] | 36 |
| memory_data[31:0][31:0 | 0,0,0,0,0,0, |
| [31][31:0] | 0 |
| [30][31:0] | 0 |
| [29][31:0] | 0 |
| [28][31:0] | 0 |
| [27][31:0] | 0 |
| [26][31:0] | 0 |
| [25][31:0] | 0 |
| [24][31:0] | 0 |
| [23][31:0] | 3 |
| [22][31:0] | 0 |
| [21][31:0] | 0 |
| [20][31:0] | 0 |
| [19][31:0] | 0 |
| [18][31:0] | 0 |
| [17][31:0] | 0 |
| [16][31:0] | 0 |
| [15][31:0] | 0 |
| [14][31:0] | 0 |
| [13][31:0] | 0 |
| [12][31:0] | 0 |
| [11][31:0] | 0 |
| [10][31:0] | 0 |
| [9][31:0] | 0 |
| [8][31:0] | 0 |
| [7][31:0] | 7 |
| [6][31:0] | 6 |
| [5][31:0] | 5 |
| [4][31:0] | 4 |
| [3][31:0] | 3 |
| [2][31:0] | 2 |
| [1][31:0] | 1 |
| [0][31:0] | 0 |

## C. Datapath with Hazard Detection

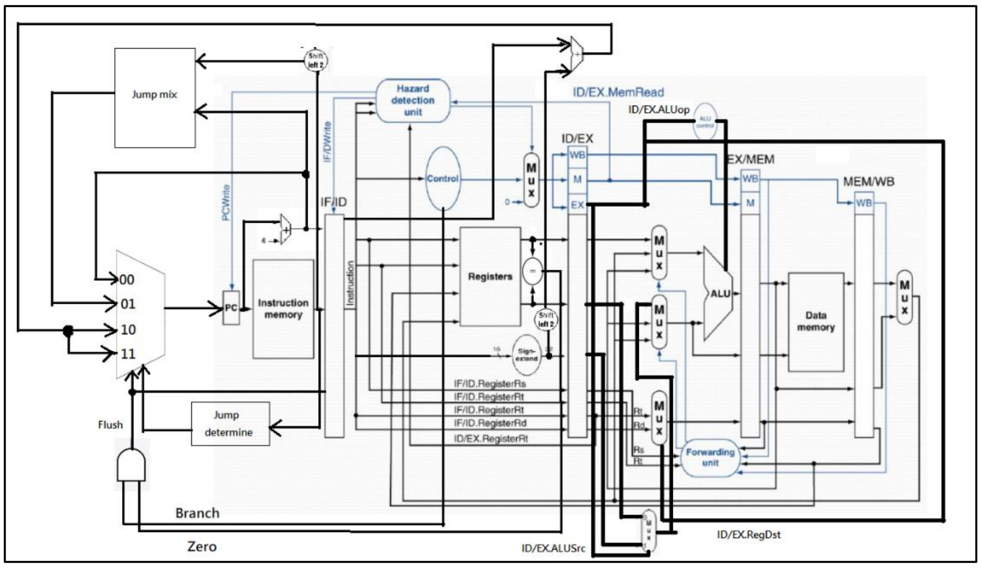
### 1. <u>Introduction</u>

In Pipeline Implementation, hazards may occur, so we use a Forwarding unit to handle them. However, since the Forwarding unit can only forward values within the same clock cycle, it can only address certain types of Data Hazards. Therefore, we also need to add a Hazard detection unit in the ID stage to handle other Data Hazards.

Branch Hazards cannot be solved by the Forwarding unit or Hazard detection unit, and when a Branch Hazard occurs, a stall will always happen. To reduce the number of stalls, we design a block in the ID stage to predict Branch Hazards in advance.

Jump Hazards also cannot be solved by the Forwarding unit or Hazard detection unit, so we design a block to address the hazards caused by jumps.

### 2. <u>Block diagram</u>



### 3. <u>Example code</u>

The Forwarding unit can resolve some types of Data Hazards, so in Loop2, we design code where a Double Data Hazard occurs. However, the Load-use Hazard requires the Hazard detection unit to be addressed, so we also design a situation in the loop where a Load-use Hazard occurs. As for Branch Hazards, the result of the branch is predicted and determined in the ID stage, so we also design code that causes at least one Branch Hazard to occur.

```
Loop
    lw $t1 ,0($t3)
    sub $t6 ,$t1 ,$t3
    addi $t5 ,$t1 , 4
    sw $t2 ,20($t3)
    beq $t6 ,$t0,LOOP2
    slt $t4,$t0,$t5
    j LOOP
Loop2
    add $t2 ,$t2 ,$t4
    and $t2,$t2,$t0
    or $t2 ,$t2 ,$t7
    addi $t3 ,$t2 , 4
    j LOOP
```

4. **Main code**

(1) PC

The PC address is the same as previously described, with the key difference being that the PC will use the PcWrite signal to decide whether or not to write the new PC address.

```
always@(posedge clk)
    begin
        if(!pc_write)
            PC <= PC;
        else
            PC <= PC_next;
    end
```

(2) Register file

When writing WriteData into WriteReg, in the earlier levels of the CPU, we used the posedge clk (positive edge of the clock) to perform the write. However, due to Data Hazards, we need to change this to negedge clk (negative edge of the clock) to avoid reading a value during ReadData that hasn't been written back yet.

(3) Forwarding unit

The Forwarding unit can resolve some Data Hazards by forwarding the results of the ALU computation to the next instruction's data usage, without needing to wait until the WB (Write-Back) stage to retrieve the computed value. The situations where hazards occur are shown in the code below. It is important to note that the MEM Hazard must also include conditions where EX Hazards do not occur. This is because, when a Double Data Hazard occurs, the most recent value needs to be forwarded to the register that is currently being operated on.

```
module
forwarding_unit(ForwardA,ForwardB,ID_EX_RegisterRt,ID_EX_RegisterRs,EX_MEM_RegisterRd,MEM_
WB_RegisterRd,EX_MEM_RegWrite,MEM_WB_RegWrite);

input [4:0] ID_EX_RegisterRt;
input [4:0] ID_EX_RegisterRs;
input [4:0] EX_MEM_RegisterRd;
input [4:0] MEM_WB_RegisterRd;

input EX_MEM_RegWrite; // Data memory
input MEM_WB_RegWrite; // Register file
```

```
output reg [1:0] ForwardA;
output reg [1:0] ForwardB;

reg EX_hazard;

always@(*)
	begin
		if( EX_MEM_RegWrite && (EX_MEM_RegisterRd!=0) && (EX_MEM_RegisterRd ==
ID_EX_RegisterRs) )
			EX_hazard = 1;
		else if(	EX_MEM_RegWrite && (EX_MEM_RegisterRd!=0) && (EX_MEM_RegisterRd ==
ID_EX_RegisterRt) )
			EX_hazard = 1;
		else
			EX_hazard = 0;
	end

always@(*)
	begin
		// EX hazard
		if( EX_MEM_RegWrite && (EX_MEM_RegisterRd!=0) && (EX_MEM_RegisterRd ==
ID_EX_RegisterRs) )
			ForwardA = 2'b10;
		else if(	EX_MEM_RegWrite && (EX_MEM_RegisterRd!=0) && (EX_MEM_RegisterRd ==
ID_EX_RegisterRt) )
			ForwardB = 2'b10;
		// MEM hazard
		else if( MEM_WB_RegWrite && (MEM_WB_RegisterRd!=0) && (!EX_hazard) &&
(MEM_WB_RegisterRd == ID_EX_RegisterRs) )
			ForwardA = 2'b01;
		else if( MEM_WB_RegWrite && (MEM_WB_RegisterRd!=0) && (!EX_hazard) &&
(MEM_WB_RegisterRd == ID_EX_RegisterRt) )
			ForwardB = 2'b01;
		else
			begin
				ForwardA = 2'b00;
				ForwardB = 2'b00;
			end
	end

endmodule
```

(4) Hazard detection unit

Since the Forwarding unit can only resolve some Data Hazards, a Hazard detection unit is added to handle the remaining data hazards. The function of the Hazard detection unit is to detect hazards and, when a hazard is detected, it performs the following actions:

Set PCWrite = 0 to prevent the PC from writing a new address.

Set IF/ID_Write = 0 to hold the instruction that is currently being executed.

Set Control_mux_select = 1 to zero out the control signals that are to be passed to the next stage.

```
module
hazard_detectio_unit(ID_EX_MemRead,ID_EX_RegisterRt,IF_ID_RegisterRs,IF_ID_RegisterRt,IF_ID_Wri
te,Control_mux_select,PcWrite);

input ID_EX_MemRead;
input [4:0] ID_EX_RegisterRt;
input [4:0] IF_ID_RegisterRs;
input [4:0] IF_ID_RegisterRt;

output reg IF_ID_Write;
output reg Control_mux_select;
output reg PcWrite;

always@(*)
    begin
        if( ID_EX_MemRead && ( (ID_EX_RegisterRt == IF_ID_RegisterRs) || (ID_EX_RegisterRt ==
IF_ID_RegisterRt) ) )
            begin
                IF_ID_Write = 0; // IF_ID_register keeps the current instruction
                Control_mux_select = 1; //    Flush
                PcWrite = 0;      // don't write in new instructions
            end
        else
            begin
                IF_ID_Write = 1;
                Control_mux_select = 0;
                PcWrite = 1;
            end
    end

endmodule
```

(5) Control

In Pipeline Implementation, the Control signals are passed to the next stage through the

Pipeline registers. When the Hazard detection unit detects a hazard, it uses the Control_MUX to set the control signals that are to be passed to the next stage to 0 (mainly RegWrite and MemWrite). This prevents incorrect data from being written to the register or memory.

- Control_MUX

```
module control_mux
(Control_mux_select,RegDst,MemtoReg,RegWrite,MemRead,MemWrite,ALUop,ALUsrc,
ID_EX_RegDst_in,ID_EX_MemtoReg_in,ID_EX_RegWrite_in,ID_EX_MemRead_in,ID_EX_MemWrite_in,
ID_EX_ALUop_in,ID_EX_ALUsrc_in);

input Control_mux_select;
input RegDst,MemtoReg,RegWrite,MemRead,MemWrite,ALUsrc;
input [1:0]ALUop;
output
ID_EX_RegDst_in,ID_EX_MemtoReg_in,ID_EX_RegWrite_in,ID_EX_MemRead_in,ID_EX_MemWrite_in,
ID_EX_ALUop_in,ID_EX_ALUsrc_in;
reg
ID_EX_RegDst_in,ID_EX_MemtoReg_in,ID_EX_RegWrite_in,ID_EX_MemRead_in,ID_EX_MemWrite_in,
ID_EX_ALUsrc_in;
reg[1:0] ID_EX_ALUop_in;
always@(*)
    begin
        if(Control_mux_select)
        begin
        ID_EX_RegDst_in=1'b0;
        ID_EX_ALUsrc_in<=1'b0;
        ID_EX_MemtoReg_in=1'b0;
        ID_EX_RegWrite_in=1'b0;
        ID_EX_MemRead_in=1'b0;
        ID_EX_MemWrite_in=1'b0;
        ID_EX_ALUop_in=2'b00;
        end
        else
        begin
        ID_EX_RegDst_in=RegDst;
        ID_EX_ALUsrc_in<=ALUsrc;
        ID_EX_MemtoReg_in=MemtoReg;
        ID_EX_RegWrite_in=RegWrite;
        ID_EX_MemRead_in=MemRead;
        ID_EX_MemWrite_in=MemWrite;
        ID_EX_ALUop_in=ALUop;
        end
    end

endmodule
```

(6) Pipeline register

In the Pipeline register, aside from slight differences in the input and output, the most notable distinction occurs in the IF/ID pipeline register. In the IF/ID pipeline register, additional control signals such as IF/ID.Write and IF_ID.flush are included to control whether the instruction is passed to the next stage for execution.

The purpose of IF/ID.Write is to decide whether to write a new PC. When a Load-use Data Hazard occurs, the value from lw (load word) is available only in the MEM stage, so a stall is required for one cycle to avoid fetching incorrect data. In this case, IF/ID.Write = 0 will hold the current instruction.

The purpose of IF_flush is to decide whether to flush unused instructions. When a Branch Hazard occurs, the result of the branch decision is known only in the ID stage, so a stall for one cycle is required. In this case, IF_flush = 1 will flush out the unused instruction.

- IF/ID Pipeline register

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

input clk;

input IF_ID_Write,IF_flush;

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
        if(IF_flush)
        begin
        pc_4_out <= 32'b0;
        instruction_out <= 32'b0;
        end
      else   if(!IF_ID_Write) // 沒有 Hazard detection 所以寫入新的 PC+4
            begin
                pc_4_out <= pc_4_out;
                instruction_out <= instruction_out;
            end
        else
            begin
                pc_4_out <= pc_4_in;
                instruction_out <= instruction_in;
            end
    end
```

(7) Branch

In Pipeline Implementation, the branch decision is typically made in the EX stage. However, this can cause problems when a branch occurs, as the branch result is obtained too late, leading to the fetching of many unnecessary instructions. To reduce the flushing of instructions, the branch-related block is moved to the ID stage, minimizing the number of flushed instructions.

The branch decision is made using the Branch signal from the Control unit and the Zero signal, which is originally computed by the ALU. In this implementation, a new block called zero_determine is added in the ID stage to compute the zero value. When a branch occurs, the instruction stored in the IF/ID Register is flushed via the IF/ID.flush signal.

- zero_determine

```
module zero_determine(data1,data2,zero_or_not);
input [31:0] data1,data2;
output reg zero_or_not;
wire [31:0]sub;
assign sub=data1-data2;

always@(*)
    begin
        if(sub==32'b0)
            zero_or_not=1'b1;
        else
            zero_or_not=1'b0;
    end

endmodule
```

- IF/ID.flush

```
module Branch_AND (a,b,branch_and_out);
input a,b;
output branch_and_out;
wire branch_and_out;
```

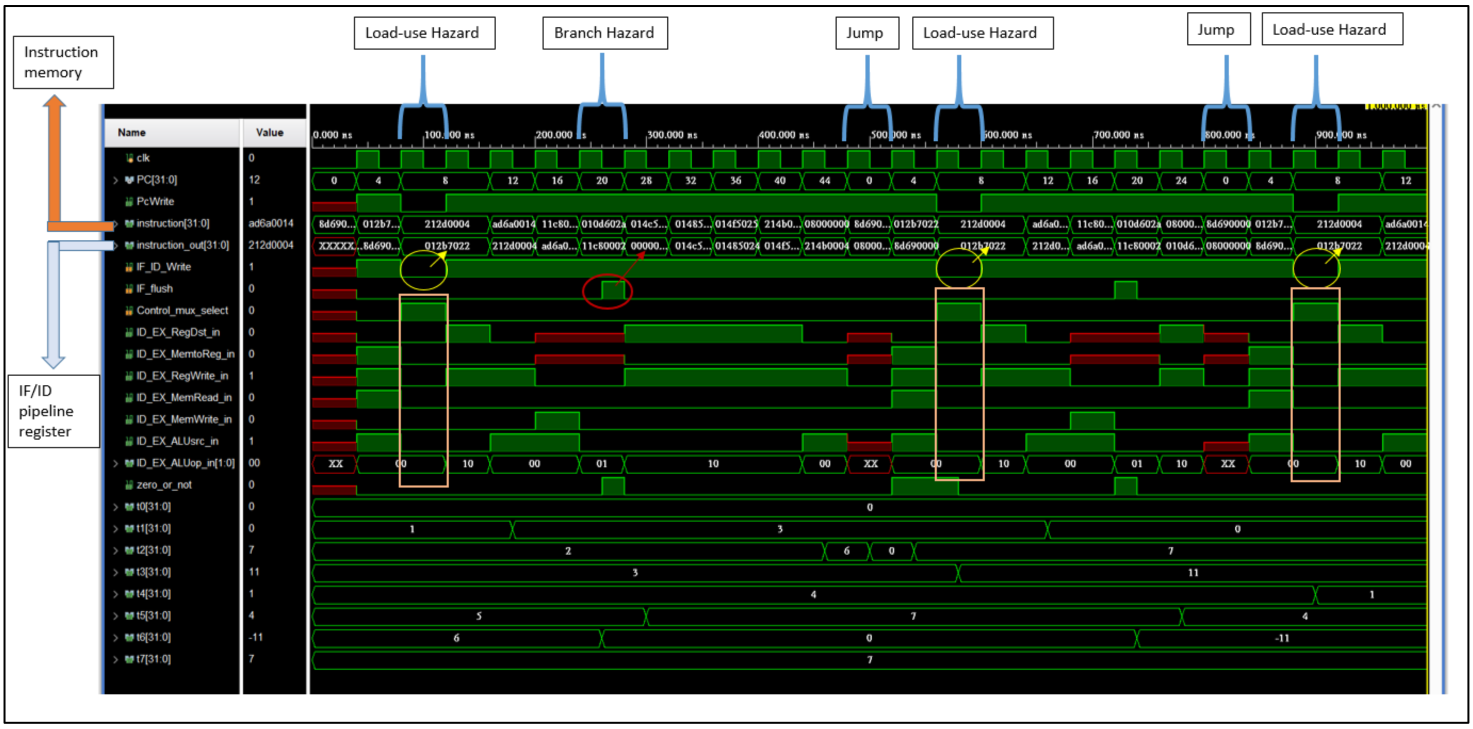
```
assign branch_and_out=a&b;

endmodule
```

**5.	Testbench**

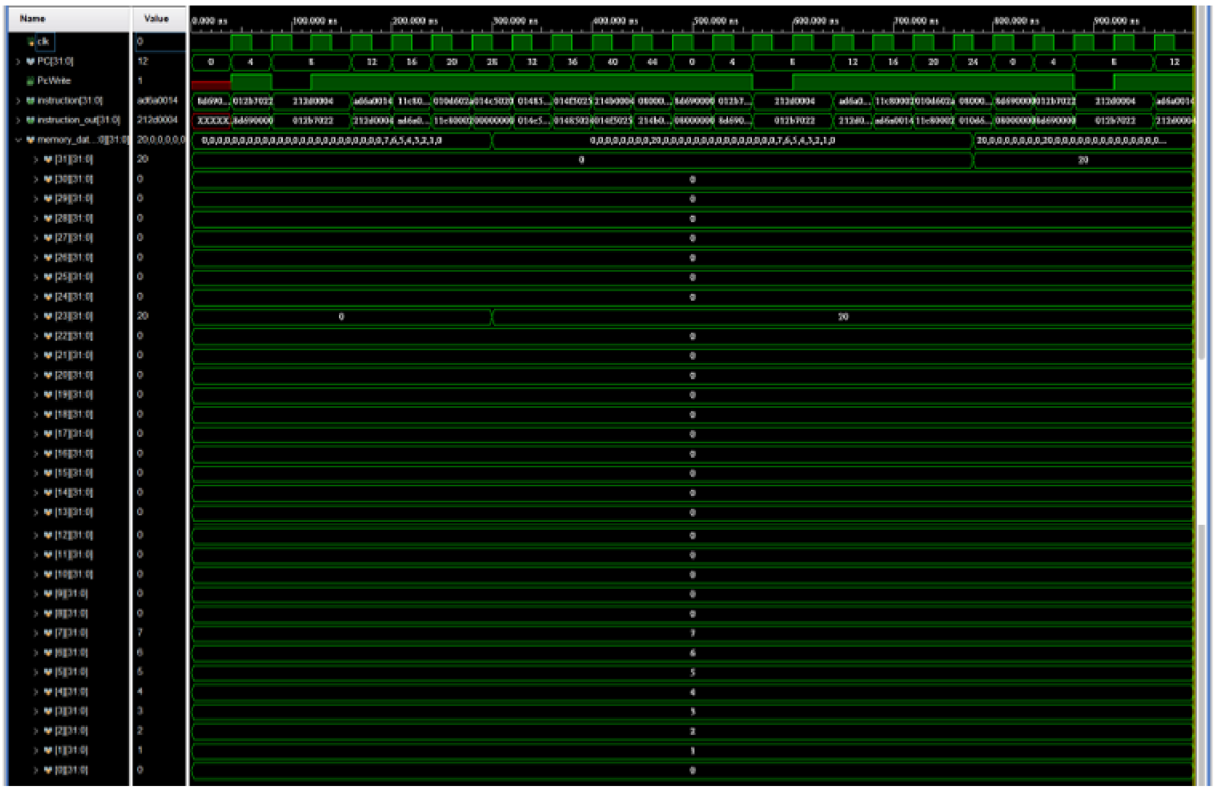
```
`timescale 1ns / 1ps

module CPU_C_tb;

reg clk;
wire [31:0] PC;

CPU_C test( .clk(clk) , .PC(PC));

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

initial
	begin
		#1000; $finish;
	end

endmodule
```

## 6. Waveform

- Hazard Occurrence & Changes in Registers ($t0-$t7).



- Changes in Data memory



◆ Reference

1. Computer Organization And Design 5th