C. Datapath with Hazard Detection

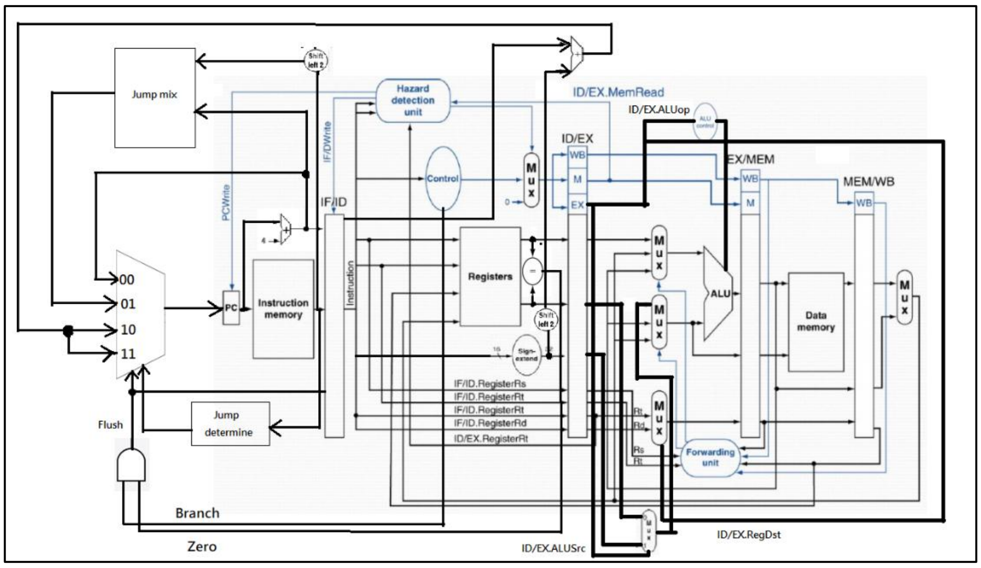
1. Introduction

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. Block diagram



3. Example code

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;
```

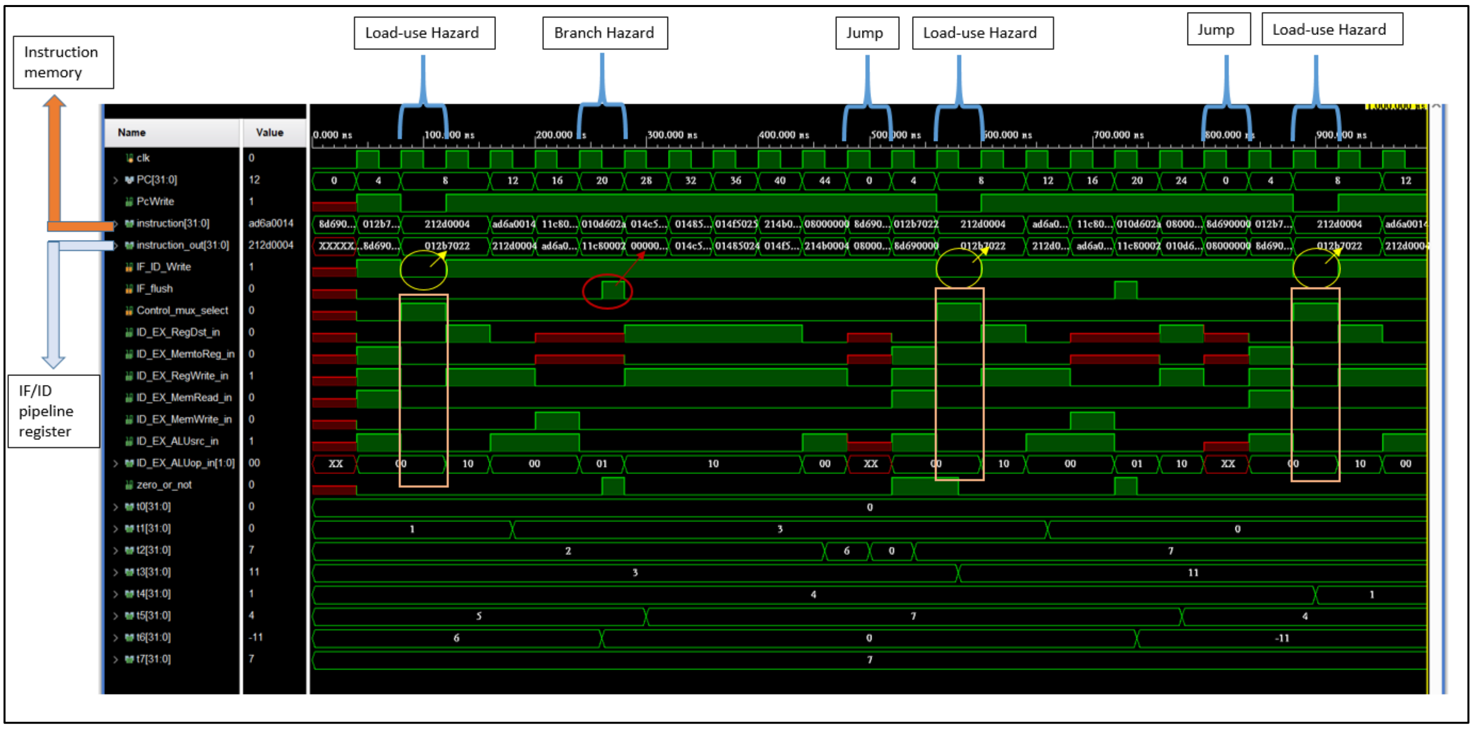
```
assign branch_and_out=a&b;

endmodule
```

5. Testbench

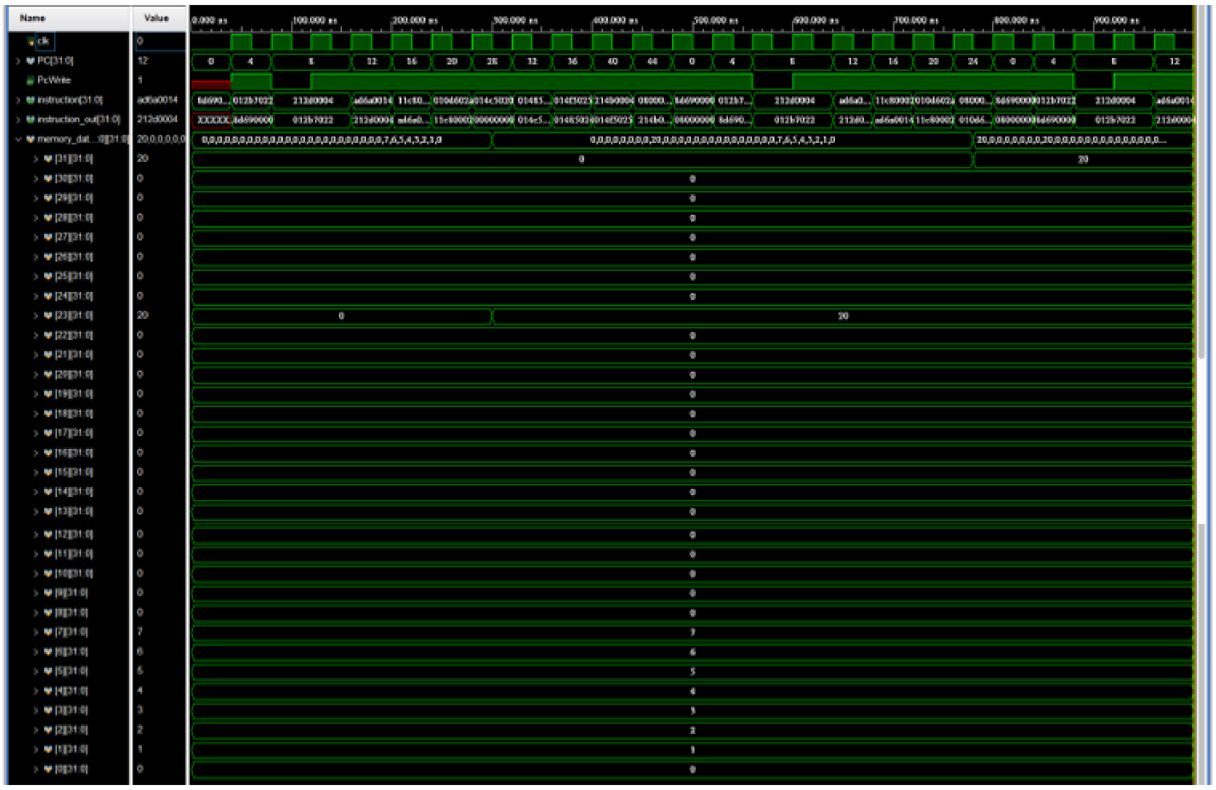
```
`timescale 1ns / 1ps

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