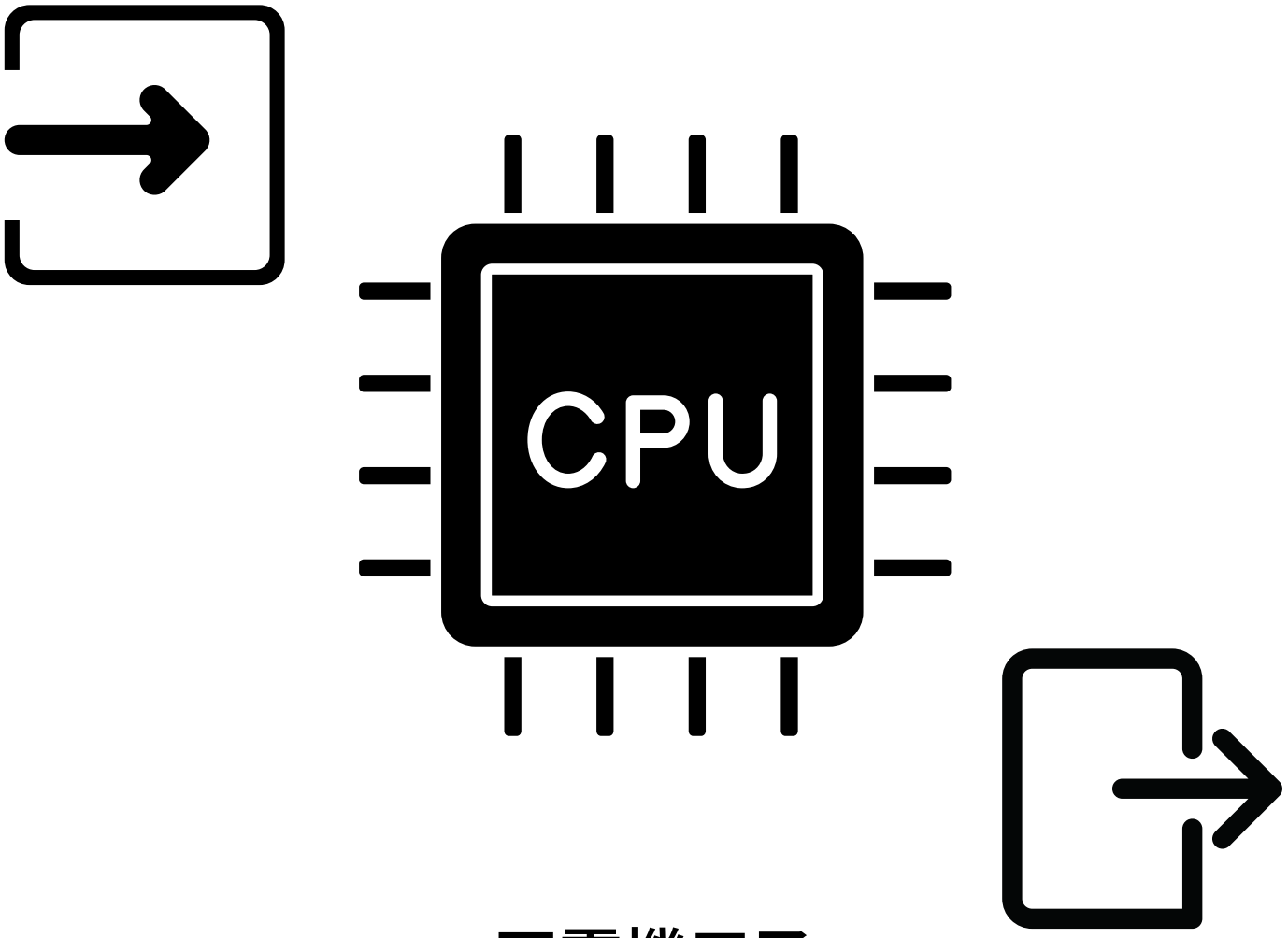
PA5

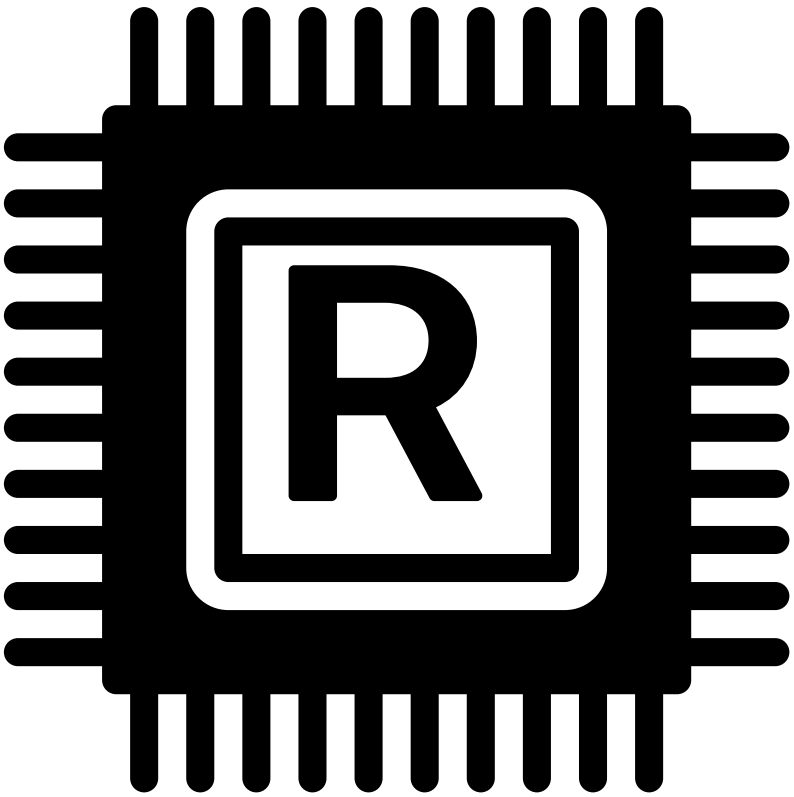
CPU with pipelines

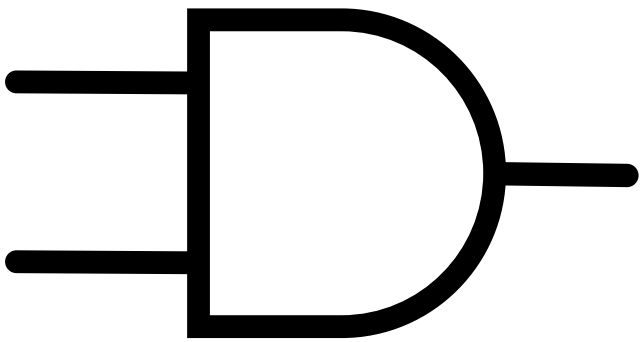


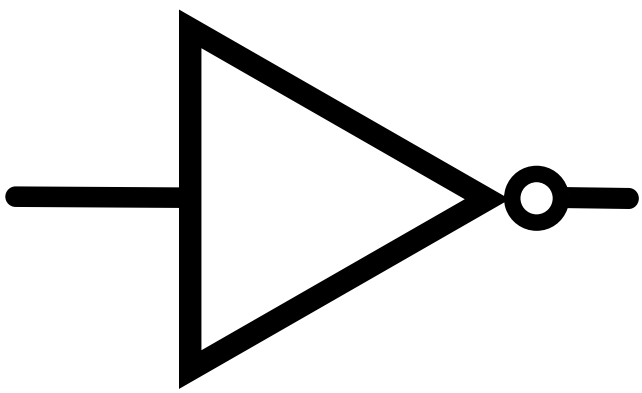
四電機三乙

B10932010 游兆暄

Part 1
R_PipelineCPU







IM、RF、ALU、Control

ALU_Control、Adder、R_PipelineCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Screenshots of each program(.v), and describe how you implement it.)

```
module IM(
    // Outputs
    output [31:0]Instruction,
    // Inputs
    input [31:0]Instr_Addr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign Instruction = {InstrMem[Instr_Addr], InstrMem[Instr_Addr+1], InstrMem[Instr_Addr+2], InstrMem[Instr_Addr+3]};
endmodule
```

有一個 32bit 的 Instr_Addr，會從 InstrMem 的 Instr_Addr、Instr_Addr+1、Instr_Addr+2、Instr_Addr+3 位置中取出值放到 Instruction 中。

```
module RF(
    // Outputs
    output [31:0] Rs_Data,
    output [31:0] Rt_Data,
    // Inputs
    input [31:0] Rd_Data,
    input [4:0] Rs_Addr,
    input [4:0] Rt_Addr,
    input [4:0] Rd_Addr,
    input RegWrite,
    input clk
);

    /*
     * Declaration of inner register.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [31:0]R[0:`REG_MEM_SIZE - 1];

    //Register MUXs
    assign  Rs_Data = R[Rs_Addr];
    assign  Rt_Data = R[Rt_Addr];
    always@(posedge clk)begin
        if(RegWrite)begin
            R[Rd_Addr] <= Rd_Data;
        end
        else R[Rd_Addr] <= R[Rd_Addr];
    end
endmodule
```

使用輸入訊號 Rs_Addr 和 Rt_Addr 來選擇讀取的 register。在 clk 正緣觸發時，當 RegWrite 信號為 1 時，代表要將 Rd_Data 寫入到 Rd_Addr 中，否則不進行任何寫入操作。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU(
    input [31:0] Src1,
    input [31:0] Src2,
    input [4:0] shamt,
    input [5:0] funct,

    output reg [31:0] Result
);

    always @(Src1, Src2, shamt, funct)begin
        case(funct)
            `Addu:  Result <= Src1 + Src2;
            `Subu:  Result <= Src1 - Src2;
            `Nor:   Result <= ~(Src1 | Src2);
            `Sltu:begin
                if(Src1 < Src2) Result <= 1;
                else Result <= 0;
            end
            default: Result = Result;
        endcase
    end
endmodule
```

定義每個功能的 Function code，在判斷 funct 中的值後，對於資料 Src_1、Src_2 進行相對應的運算，最後運算完的資料傳到 ALUResult。

```
module Control(
    output reg RegWrite,
    output reg[1:0]ALUOp,

    input [5:0]OpCode
);

always@(OpCode)begin
    case(OpCode)
        6'b000000: begin
            RegWrite <= 1;
            ALUOp <= 2'b10;
        end
        default:begin
            RegWrite <= 0;
            ALUOp <= 2'b11;
        end
    endcase
end
endmodule
```

有輸入訊號 6bits 的 OpCode 以及輸出訊號 RegWrite 和 ALUOp。先判斷 OpCode 的值，當 OpCode 為 6'b000000 (R_format 指令) 時，設定 RegWrite 為 1，讓資料可以寫入 RF，並且將 ALUOp 設為 2'b10。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU_Control(
    output reg[5:0] funct,

    input [1:0] ALUOp,
    input [5:0] funct_ctrl
);
always@(ALUOp, funct_ctrl)begin
    case(ALUOp)
        2'b10:begin
            case(funct_ctrl)
                6'b001011: funct <= `Addu;
                6'b001101: funct <= `Subu;
                6'b100111: funct <= `Nor;
                6'b101010: funct <= `Sltu;
                default: funct <= 6'b0;
            endcase
        end
    endcase
end
endmodule
```

根據 ALUOp 的值來設定 funct 的值。當 ALUOp 的值為 2'b10 時，判斷 funct_ctrl 的值來設定 funct(Addu[6'b001011]、Subu[6'b001101]、Nor[6'b100111]、Sltu[6'b101010])。如果 funct_ctrl 的沒有在 case 裡面，則設定 funct 為 6'b0。

```
module Adder(
    output [31:0]Output_Addr,
    input [31:0]Src1,
    input [31:0]Src2
);
assign Output_Addr = Src1 + Src2;
endmodule
```

將 Src1 和 Src2 的值相加輸出到 Output_Addr，用來計算下一個 Address。

```
module IF_ID_Pipeline(
    output reg[31:0] out,
    input [31:0] Instr,
    input clk
);
always@(posedge clk)begin
    out <= Instr;
end
endmodule
```

在 clk 正緣觸發時，Instr 會將資料傳到 out 中輸出。

```
module ID_EX_Pipeline(
    //output
    output reg[31:0] ID_EX_RsData,
    output reg[31:0] ID_EX_RtData,
    output reg[4:0] ID_EX_RdAddr,
    output reg[15:0] ID_EX_imm,
    output reg [1:0] ID_EX_M,
    output reg [1:0] ID_EX_EX,
    output reg ID_EX_WB,
    //input
    input [31:0] RsData,
    input [31:0] RtData,
    input [4:0] IF_ID_RdAddr,
    input [15:0] imm,
    input [1:0] M,
    input [1:0] EX,
    input WB,
    input clk
);
```

```
always@(posedge clk)begin
    ID_EX_RsData <= RsData;
    ID_EX_RtData <= RtData;
    ID_EX_RdAddr <= IF_ID_RdAddr;
    ID_EX_imm <= imm;
    ID_EX_WB <= WB;
    ID_EX_M <= M;
    ID_EX_EX <= EX;
end
endmodule
```

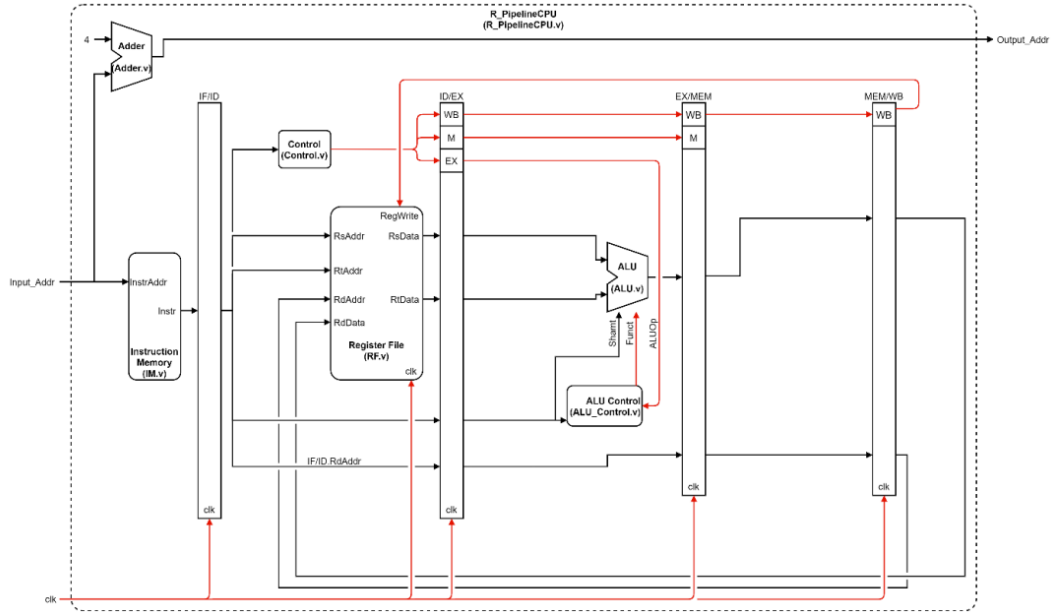
在 clk 正緣觸發時，會將 RsData、RtData、IF_ID_RdAddr、imm、WB、M、EX 中的資料輸出。

```
module EX_MEM_Pipeline(
    //output
    output reg[31:0] EX_MEM_ALU_Result,
    output reg[4:0] EX_MEM_RdAddr,
    output reg [1:0] EX_MEM_M,
    output reg EX_MEM_WB,
    //input
    input [31:0] ALU_Result,
    input [4:0] ID_EX_RdAddr,
    input [1:0] M,
    input WB,
    input clk
);
always@(posedge clk)begin
    EX_MEM_ALU_Result <= ALU_Result;
    EX_MEM_RdAddr <= ID_EX_RdAddr;
    EX_MEM_WB <= WB;
    EX_MEM_M <= M;
end
endmodule
```

在 clk 正緣觸發時，會將 ALU_Result、ID_EX_RdAddr、M、WB 中的資料輸出。

```
module MEM_WB_Pipeline(
    //output
    output reg[31:0] MEM_WB_ALU_Result,
    output reg[4:0] IF_ID_RdAddr_out,
    output reg WB_out,
    //input
    input [31:0] EX_MEM_ALU_Result,
    input [4:0] EX_MEM_RdAddr,
    input WB,
    input clk
);
always@(posedge clk)begin
    MEM_WB_ALU_Result <= EX_MEM_ALU_Result;
    IF_ID_RdAddr_out <= EX_MEM_RdAddr;
    WB_out <= WB;
end
endmodule
```

在 clk 正緣觸發時，會將 EX_MEM_ALU_Result、EX_MEM_RdAddr、WB 中的資料輸出。




```v
module R_PipelineCPU(
    // Outputs
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);
wire [31:0] Instruction;
wire [31:0] Rs_Data;
wire [31:0] Rt_Data;
wire [5:0] funct;
wire [31:0] ALU_Result;
//  Control
wire WB;
wire [1:0] M;
wire [1:0] EX;
//  IF/ID
wire [31:0] Instruction_out;
wire [4:0] IF_ID_RdAddr;
//  ID/EX
wire ID_EX_WB;
wire [1:0] ALUOp;
wire [1:0] ID_EX_M;
wire [31:0] ID_EX_RsData;
wire [31:0] ID_EX_RtData;
wire [15:0] ID_EX_imm;
wire [4:0] ID_EX_RdAddr;
//  EX/MEM
wire EX_MEM_WB;
wire [31:0] EX_MEM_ALU_Result;
wire [4:0] EX_MEM_RdAddr;
```

```v
//  MEM/WB
wire RegWrite;
wire [31:0] Rd_Data;
wire [4:0] Rd_Addr;

IM Instr_Memory(
    // Outputs
    .Instruction(Instruction),
    // Inputs
    .Instr_Addr(Input_Addr)
);

/*
 * Declaration of Register File.
 * CAUTION: DONT MODIFY THE NAME.
 */
RF Register_File(
    // Outputs
    .Rs_Data(Rs_Data),
    .Rt_Data(Rt_Data),
    // Inputs
    .Rd_Data(Rd_Data),
    .Rs_Addr(Instruction_out[25:21]),
    .Rt_Addr(Instruction_out[20:16]),
    .Rd_Addr(Rd_Addr),
    .RegWrite(RegWrite),
    .clk(clk)
);
```

```v
ALU Arithmetic(
    // Outputs
    .Result(ALU_Result),
    // Inputs
    .Src1(ID_EX_RsData),
    .Src2(ID_EX_RtData),
    .shamt(ID_EX_imm[10:6]),
    .funct(funct)
);

Control Controller(
    // Outputs
    .ALUOp(EX),
    .RegWrite(WB),
    .MemRead(M[1]),
    .MemWrite(M[0]),
    // Inputs
    .OpCode(Instruction_out[31:26])
);

ALU_Control ALU_Controller(
    //Outputs
    .funct(funct),
    //Inputs
    .funct_ctrl(ID_EX_imm[5:0]),
    .ALUOp(ALUOp)
);
```

```v
Adder Addr_Adder(
    //Outputs
    .Output_Addr(Output_Addr),
    //Inputs
    .Src1(Input_Addr),
    .Src2(32'd4)
);

IF_ID_Pipeline IF_ID_Pipeline(
    //output
    .out(Instruction_out),
    //input
    .Instr(Instruction),
    .clk(clk)
);
```

```v
ID_EX_Pipeline ID_EX_Pipeline(
    //output
    .ID_EX_RsData(ID_EX_RsData),
    .ID_EX_RtData(ID_EX_RtData),
    .ID_EX_RdAddr(ID_EX_RdAddr),
    .ID_EX_imm(ID_EX_imm),
    .ID_EX_WB(ID_EX_WB),
    .ID_EX_M(ID_EX_M),
    .ID_EX_EX(ALUOp),
    //input
    .RsData(Rs_Data),
    .RtData(Rt_Data),
    .IF_ID_RdAddr(Instruction_out[15:11]),
    .imm(Instruction_out[15:0]),
    .WB(WB),
    .M(M),
    .EX(EX),
    .clk(clk)
);
```

```v
EX_MEM_Pipeline EX_MEM_Pipeline(
    //output
    .EX_MEM_ALU_Result(EX_MEM_ALU_Result),
    .EX_MEM_RdAddr(EX_MEM_RdAddr),
    .EX_MEM_M(),
    .EX_MEM_WB(EX_MEM_WB),
    //input
    .ALU_Result(ALU_Result),
    .ID_EX_RdAddr(ID_EX_RdAddr),
    .M(ID_EX_M),
    .WB(ID_EX_WB),
    .clk(clk)
);

MEM_WB_Pipeline MEM_WB_Pipeline(
    //output
    .MEM_WB_ALU_Result(Rd_Data),
    .IF_ID_RdAddr_out(Rd_Addr),
    .WB_out(RegWrite),
    //input
    .EX_MEM_ALU_Result(EX_MEM_ALU_Result),
    .EX_MEM_RdAddr(EX_MEM_RdAddr),
    .WB(EX_MEM_WB),
    .clk(clk)
);
endmodule
```

將 IM、RF、ALU、Control、ALU_Control、Adder、R_PipelineCPU、IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline 組合成一個大的模組，並且使用 wire 將所有的元件串接在一起，注意每個元件的輸出及輸入，有些指輸入 32bits 中的 5bits 或 6bits，分別放入正確的輸出入位置。

IM、RF、ALU、Control

ALU_Control、Adder、R_PipelineCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Test each part with your testbench and explain the results.)

```
// Instruction Memory in Hex
01      // Addr = 0x00
4B      // Addr = 0x01
A0      // Addr = 0x02
0B      // Addr = 0x03
01      // Addr = 0x04
AC      // Addr = 0x05
A8      // Addr = 0x06
0D      // Addr = 0x07
02      // Addr = 0x08
32      // Addr = 0x09
B0      // Addr = 0x0A
27      // Addr = 0x0B
01      // Addr = 0x0C
CF      // Addr = 0x0D
B8      // Addr = 0x0E
2A      // Addr = 0x0F
FF      // Addr = 0x10
FF      // Addr = 0x11
FF      // Addr = 0x12
FF      // Addr = 0x13
```

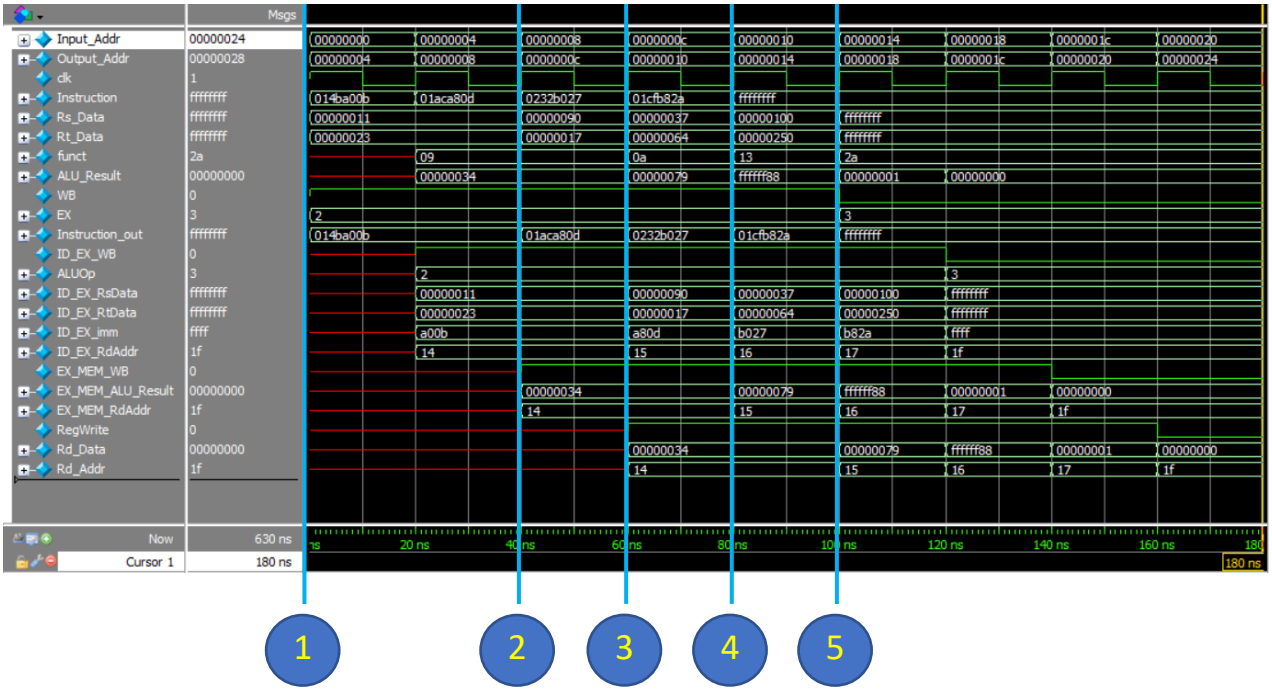
IM

```
21    00000034 // R[20]
22    00000079 // R[21]
23    ffffff88 // R[22]
24    00000001 // R[23]
```

RF.out

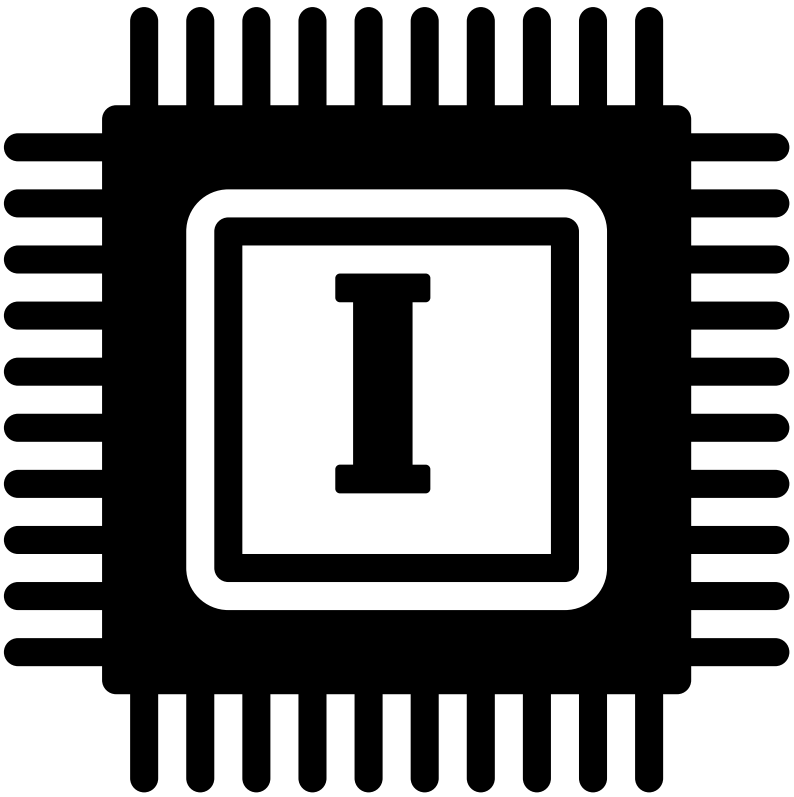
```
0000_0011   // R[10]
0000_0023   // R[11]
0000_0017   // R[12]
0000_0090   // R[13]
0000_0100   // R[14]
0000_0250   // R[15]
0000_0300   // R[16]
0000_0037   // R[17]
0000_0064   // R[18]
0000_0030   // R[19]
0000_0000   // R[20]
```

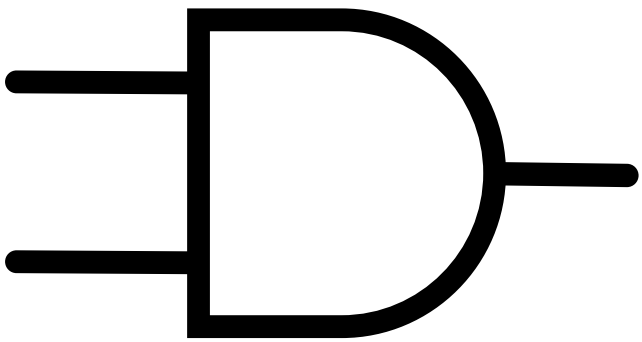
RF

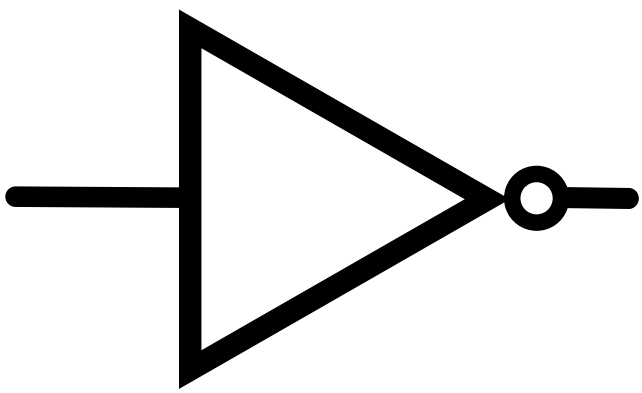


Instruction(Hex)	Instruction_Type	Instruction(Binary)						Meaning(Dec)	Data(Hex)		Result(Hex)	IM_Addr(Hex)
	R_format	OP(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Funct(6)	Ins $Rd, $Rs, $Rt	Rs	Rt		
014BA00B	R_format	000000	01010	01011	10100	00000	001011	Addu $20, $10, $11	0000_0011	0000_0023	0000_0034	00
01ACA80D	R_format	000000	01101	01100	10101	00000	001101	Subu $21, $13, $12	0000_0090	0000_0017	0000_0079	04
0232B027	R_format	000000	10001	10010	10110	00000	100111	Nor $22, $17, $18	0000_0037	0000_0064	FFFF_FF88	08
01CFB82A	R_format	000000	01110	01111	10111	00000	101010	Sltu $23, $14, $15	0000_0100	0000_0250	0000_0001	0C
FFFFFFFF	Undefined	111111	11111	11111	11111	11111	111111	Undefined			Fixed	10

Part 2
I_PipelineCPU







IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

I_PipelineCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Screenshots of each program, and description of the process.)

```
module IM(
    // Outputs
    output [31:0]Instruction,
    // Inputs
    input [31:0]Instr_Addr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign Instruction = {InstrMem[Instr_Addr], InstrMem[Instr_Addr+1], InstrMem[Instr_Addr+2], InstrMem[Instr_Addr+3]};
endmodule
```

有一個 32bit 的 Instr_Addr，會從 InstrMem 的 Instr_Addr、Instr_Addr+1、Instr_Addr+2、Instr_Addr+3 位置中取出值放到 Instruction 中。

```
module RF(
    // Outputs
    output [31:0] Rs_Data,
    output [31:0] Rt_Data,
    // Inputs
    input [31:0] Rd_Data,
    input [4:0] Rs_Addr,
    input [4:0] Rt_Addr,
    input [4:0] Rd_Addr,
    input RegWrite,
    input clk
);

    /*
     * Declaration of inner register.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [31:0]R[0:`REG_MEM_SIZE - 1];

    //Register MUXs
    assign  Rs_Data = R[Rs_Addr];
    assign  Rt_Data = R[Rt_Addr];
    always@(posedge clk)begin
        if(RegWrite)begin
            R[Rd_Addr] <= Rd_Data;
        end
        else R[Rd_Addr] <= R[Rd_Addr];
    end
endmodule
```

使用輸入訊號 Rs_Addr 和 Rt_Addr 來選擇讀取的 register。在 clk 正緣觸發時，當 RegWrite 信號為 1 時，代表要將 Rd_Data 寫入到 Rd_Addr 中，否則不進行任何寫入操作。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU(
    input [31:0] Src1,
    input [31:0] Src2,
    input [4:0] shamt,
    input [5:0] funct,

    output reg [31:0] Result
);

    always @(Src1, Src2, shamt, funct)begin
        case(funct)
            `Addu:  Result <= Src1 + Src2;
            `Subu:  Result <= Src1 - Src2;
            `Nor:   Result <= ~(Src1 | Src2);
            `Sltu:begin
                if(Src1 < Src2) Result <= 1;
                else Result <= 0;
            end
            default: Result = Result;
        endcase
    end
endmodule
```

定義每個功能的 Function code，在判斷 funct 中的值後，對於資料 Src_1、Src_2 進行相對應的運算，最後運算完的資料傳到 ALUResult。

```
`define R_format            6'b000000
`define Add_imm_unsigned    6'b001100
`define Sub_imm_unsigned    6'b001101
`define Store_word          6'b010000
`define Load_word           6'b010001

module Control(
    //output
    output reg RegWrite,
    output reg[1:0]ALUOp,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemWrite,
    output reg MemRead,
    output reg MemtoReg,
    //input
    input [5:0]OpCode
);
```

```
always@(OpCode)begin
    case(OpCode)
        `R_format: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            ALUSrc <= 0;
            RegDst <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b10;
        end
        `Add_imm_unsigned: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b00;
        end
        `Sub_imm_unsigned: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b01;
        end
```

```
        `Store_word: begin
            RegWrite <= 0;
            MemWrite <= 1;
            MemRead <= 0;
            ALUSrc <= 1;
            ALUOp <= 2'b00;
        end
        `Load_word: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 1;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 1;
            ALUOp <= 2'b00;
        end
        default:begin
            RegWrite <= 0;
            ALUOp <= 2'b11;
        end
    endcase
end
endmodule
```

有輸入訊號 6bits 的 OpCode 以及輸出訊號 RegWrite 和 ALUOp。先判斷 OpCode 的值，當 OpCode 為 6'b000000 (R_format 指令) 時，設定 RegWrite 為 1，讓資料可以寫入 RF，並且將 ALUOp 設為 2'b10。若為 I 指令時，則將 ALUOp 設為 2'b00(Subiu 為 2'b01)，並分別依照其指令之功能改變 RegWrite、RegDst、ALUSrc、MemWrite、MemRead、MemtoReg 的值，使資料透過正確的路徑達到正確的元件執行相對應的功能。

```
module Bits5_Mux(
    //output
    output [4:0]Mux_out,
    //input
    input [4:0]Mux_in_0,
    input [4:0]Mux_in_1,
    input sel
);

assign Mux_out = (sel)? Mux_in_1: Mux_in_0;
endmodule
```

```
module Bits32_Mux(
    //output
    output [31:0]Mux_out,
    //input
    input [31:0]Mux_in_0,
    input [31:0]Mux_in_1,
    input sel
);

assign Mux_out = (sel)? Mux_in_1: Mux_in_0;
endmodule
```

根據 sel 的值，若為 1，Mux_out 中放入 Mux_in_1。若為 0，Mux_out 中放入 Mux_in_0。

```
module DM(
    // Outputs
    output [31:0] MemReadData,
    // Inputs
    input [31:0] MemAddr,
    input [31:0] MemWriteData,
    input MemWrite,
    input MemRead,
    input clk
);

    /*
     * Declaration of data memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
    assign  MemReadData =(MemRead)?
    {DataMem[MemAddr], DataMem[MemAddr+1], DataMem[MemAddr+2], DataMem[MemAddr+3]}: MemReadData;
    always@(posedge clk)begin
        if(MemWrite)begin
            {DataMem[MemAddr], DataMem[MemAddr+1], DataMem[MemAddr+2], DataMem[MemAddr+3]} <= MemWriteData;
        end
    end
endmodule
```

有四個輸入信號 MemAddr、MemWriteData、MemWrite、MemRead。有一個 MemAddr 32bits 的 Memory 位置輸入，表示要 Write 或 Read 的地址。當 clk 正緣觸發時，判斷 MemWrite 是否為 1，若是，MemWriteData 將被寫入到地址的位置。當 MemRead 被設為 1 時，模組將讀取位址所指示的資料，並輸出到 MemReadData。

```
module SignExtend(
    //output
    output [31:0]out,
    //input
    input [15:0]in
);
assign out = (in[15])? {16'hFFFF, in}: {16'h0000, in};
endmodule
```

對輸入資料 in 進行 Sign Extension 至 32 bits 後輸出到 out。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU_Control(
    output reg[5:0] funct,

    input [1:0] ALUOp,
    input [5:0] funct_ctrl
);
always@(ALUOp, funct_ctrl)begin
    case(ALUOp)
        2'b10:begin
            case(funct_ctrl)
                6'b001011: funct <= `Addu;
                6'b001101: funct <= `Subu;
                6'b100111: funct <= `Nor;
                6'b101010: funct <= `Sltu;
                default: funct <= 6'b0;
            endcase
        end
        2'b01:begin
            funct <= `Subu;
        end
        2'b00:begin
            funct <= `Addu;
        end
    endcase
end
endmodule
```

根據 ALUOp 的值來設定 funct 的值。當 ALUOp 的值為 2'b10 時，判斷 funct_ctrl 的值來設定 funct(Addu[6'b001011]、Subu[6'b001101]、Nor[6'b100111]、Sltu[6'b101010])。當 ALUOp 的值為 2'b10 時，設定 funct 為 Subu[6'b001101]。當 ALUOp 的值為 2'b00 時，設定 funct 為 Addu[6'b001011]。

```
module Adder(
    output [31:0]Output_Addr,
    input [31:0]Src1,
    input [31:0]Src2
);
assign Output_Addr = Src1 + Src2;
endmodule
```

將 Src1 和 Src2 的值相加輸出到 Output_Addr，用來計算下一個 Address。

```
module IF_ID_Pipeline(
    output reg[31:0] out,
    input [31:0] Instr,
    input clk
);
always@(posedge clk)begin
    out <= Instr;
end
endmodule
```

在 clk 正緣觸發時，Instr 會將資料傳到 out 中輸出。

```
module ID_EX_Pipeline(
    //output
    output reg[31:0] ID_EX_RsData,
    output reg[31:0] ID_EX_RtData,
    output reg[4:0] ID_EX_RdAddr,
    output reg[4:0] ID_EX_RtAddr,
    output reg[31:0] ID_EX_SignExtend,
    output reg [1:0] ID_EX_M,
    output reg [3:0] ID_EX_EX,
    output reg [1:0]ID_EX_WB,
    //input
    input [31:0] RsData,
    input [31:0] RtData,
    input [4:0] IF_ID_RdAddr,
    input [4:0] IF_ID_RtAddr,
    input [31:0] SignExtend,
    input [1:0] M,
    input [3:0] EX,
    input [1:0] WB,
    input clk
);
```

```
always@(posedge clk)begin
    ID_EX_RsData <= RsData;
    ID_EX_RtData <= RtData;
    ID_EX_RdAddr <= IF_ID_RdAddr;
    ID_EX_RtAddr <= IF_ID_RtAddr;
    ID_EX_SignExtend <= SignExtend;
    ID_EX_WB <= WB;
    ID_EX_M <= M;
    ID_EX_EX <= EX;
end
endmodule
```

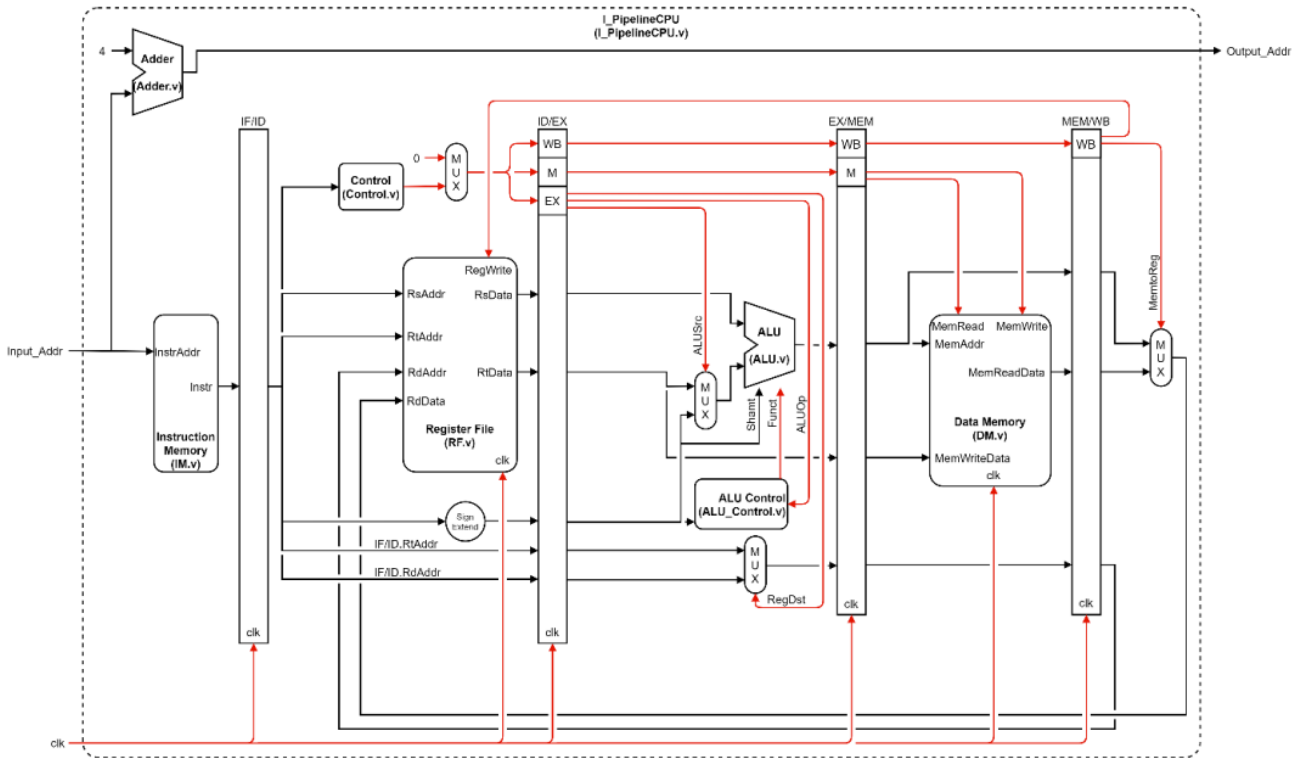
在 clk 正緣觸發時，會將 RsData、RtData、IF_ID_RdAddr、IF_ID_RtAddr、SignExtend、WB、M、EX 中的資料輸出。

```
module EX_MEM_Pipeline(
    //output
    output reg[31:0] EX_MEM_ALU_Result,
    output reg[31:0] EX_MEM_RtData,
    output reg[4:0] EX_MEM_Addr,
    output reg [1:0] EX_MEM_M,
    output reg [1:0]EX_MEM_WB,
    //input
    input [31:0] ALU_Result,
    input [31:0] Rt_Data,
    input [4:0] IF_ID_Addr,
    input [1:0] M,
    input [1:0]WB,
    input clk
);
always@(posedge clk)begin
    EX_MEM_ALU_Result <= ALU_Result;
    EX_MEM_Addr <= IF_ID_Addr;
    EX_MEM_RtData <= Rt_Data;
    EX_MEM_WB <= WB;
    EX_MEM_M <= M;
end
endmodule
```

在 clk 正緣觸發時，會將 ALU_Result、Rt_Data、IF_ID_RdAddr、M、WB 中的資料輸出。

```
module MEM_WB_Pipeline(
    //output
    output reg[31:0] MEM_WB_ALU_Result,
    output reg[31:0] MemReadData_out,
    output reg[4:0] IF_ID_Addr_out,
    output reg [1:0]WB_out,
    //input
    input [31:0] ALU_Result,
    input [31:0] MemReadData,
    input [4:0] EX_MEM_Addr,
    input [1:0]WB,
    input clk
);
always@(posedge clk)begin
    MEM_WB_ALU_Result <= ALU_Result;
    MemReadData_out <= MemReadData;
    IF_ID_Addr_out <= EX_MEM_Addr;
    WB_out <= WB;
end
endmodule
```

在 clk 正緣觸發時，將 ALU_Result、MemReadData、EX_MEM_Addr、WB 中的資料輸出。



```
module I_PipelineCPU(
    // Outputs
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire        clk
);
wire [31:0] Instruction;
wire [31:0] Rs_Data;
wire [31:0] Rt_Data;
wire [31:0] Rd_Data;
wire [5:0] funct;
wire [31:0] ALU_Result;
wire [31:0] SignExtend_out;
wire [31:0] Bits32_Mux_out;
wire [4:0] IF_ID_Addr;
//  Control
wire [1:0] WB;
wire [1:0] M;
wire [3:0] EX;
//  DM
wire [31:0] MemReadData;
//  IF/ID
wire [31:0] Instruction_out;
```

```
//  ID/EX
wire [1:0]ID_EX_WB;
wire RegDst;
wire ALUSrc;
wire [1:0] ALUOp;
wire [1:0] ID_EX_M;
wire [31:0] ID_EX_RsData;
wire [31:0] ID_EX_RtData;
wire [31:0] ID_EX_SignExtend;
wire [4:0] ID_EX_RdAddr;
wire [4:0] ID_EX_RtAddr;
//  EX/MEM
wire [1:0]EX_MEM_WB;
wire [31:0] EX_MEM_ALU_Result;
wire [31:0] EX_MEM_RtData;
wire [4:0] EX_MEM_Addr;
wire MemRead;
wire MemWrite;
//  MEM/WB
wire RegWrite;
wire MemtoReg;
wire [31:0] MEM_WB_ALU_Result;
wire [31:0] MemReadData_out;
wire [4:0] Rd_Addr;
```

```
IM Instr_Memory(
    // Outputs
    .Instruction(Instruction),
    // Inputs
    .Instr_Addr(Input_Addr)
);

/*
 * Declaration of Register File.
 * CAUTION: DONT MODIFY THE NAME.
 */
RF Register_File(
// Outputs
.Rs_Data(Rs_Data),
.Rt_Data(Rt_Data),
// Inputs
.Rd_Data(Rd_Data),
.Rs_Addr(Instruction_out[25:21]),
.Rt_Addr(Instruction_out[20:16]),
.Rd_Addr(Rd_Addr),
.RegWrite(RegWrite),
.clk(clk)
);
```

```
DM Data_Memory(
// Outputs
.MemReadData(MemReadData),
// Inputs
.MemAddr(EX_MEM_ALU_Result),
.MemWriteData(EX_MEM_RtData),
.MemWrite(MemWrite),
.MemRead(MemRead),
.clk(clk)
);

Control Controller(
    //output
    .RegWrite(WB[1]),
    .ALUOp(EX[2:1]),
    .RegDst(EX[0]),
    .ALUSrc(EX[3]),
    .MemWrite(M[0]),
    .MemRead(M[1]),
    .MemtoReg(WB[0]),
    //input
    .OpCode(Instruction_out[31:26])
);

ALU_Control ALU_Controller(
    //Outputs
    .funct(funct),
    //Inputs
    .funct_ctrl(ID_EX_SignExtend[5:0]),
    .ALUOp(ALUOp)
);
```

```
ALU Arithmetic(
    // Outputs
    .Result(ALU_Result),
    // Inputs
    .Src1(ID_EX_RsData),
    .Src2(Bits32_Mux_out),
    .shamt(ID_EX_SignExtend[10:6]),
    .funct(funct)
);

Adder Addr_Adder(
    //Outputs
    .Output_Addr(Output_Addr),
    //Inputs
    .Src1(Input_Addr),
    .Src2(32'd4)
);

SignExtend SignExtension(
    //Outputs
    .out(SignExtend_out),
    //Inputs
    .in(Instruction_out[15:0])
);
```

```
Bits5_Mux Bits5_Mux(
    //output
    .Mux_out(IF_ID_Addr),
    //input
    .Mux_in_0(ID_EX_RtAddr),
    .Mux_in_1(ID_EX_RdAddr),
    .sel(RegDst)
);

Bits32_Mux Bits32_Mux_ALU(
    //output
    .Mux_out(Bits32_Mux_out),
    //input
    .Mux_in_0(ID_EX_RtData),
    .Mux_in_1(ID_EX_SignExtend),
    .sel(ALUSrc)
);

Bits32_Mux Bits32_Mux_Mem(
    //output
    .Mux_out(Rd_Data),
    //input
    .Mux_in_0(MEM_WB_ALU_Result),
    .Mux_in_1(MemReadData_out),
    .sel(MemtoReg)
);
```

```
IF_ID_Pipeline IF_ID_Pipeline(
    //output
    .out(Instruction_out),
    //input
    .Instr(Instruction),
    .clk(clk)
);

ID_EX_Pipeline ID_EX_Pipeline(
    //output
    .ID_EX_RsData(ID_EX_RsData),
    .ID_EX_RtData(ID_EX_RtData),
    .ID_EX_RdAddr(ID_EX_RdAddr),
    .ID_EX_RtAddr(ID_EX_RtAddr),
    .ID_EX_SignExtend(ID_EX_SignExtend),
    .ID_EX_WB(ID_EX_WB),
    .ID_EX_M(ID_EX_M),
    .ID_EX_EX({ALUSrc, ALUOp, RegDst}),
    //input
    .RsData(Rs_Data),
    .RtData(Rt_Data),
    .IF_ID_RdAddr(Instruction_out[15:11]),
    .IF_ID_RtAddr(Instruction_out[20:16]),
    .SignExtend(SignExtend_out),
    .WB(WB),
    .M(M),
    .EX(EX),
    .clk(clk)
);
```

```
    EX_MEM_Pipeline EX_MEM_Pipeline(
        //output
        .EX_MEM_ALU_Result(EX_MEM_ALU_Result),
        .EX_MEM_Addr(EX_MEM_Addr),
        .EX_MEM_M({MemRead, MemWrite}),
        .EX_MEM_RtData(EX_MEM_RtData),
        .EX_MEM_WB(EX_MEM_WB),
        //input
        .ALU_Result(ALU_Result),
        .IF_ID_Addr(IF_ID_Addr),
        .Rt_Data(ID_EX_RtData),
        .M(ID_EX_M),
        .WB(ID_EX_WB),
        .clk(clk)
    );

    MEM_WB_Pipeline MEM_WB_Pipeline(
        //output
        .MEM_WB_ALU_Result(MEM_WB_ALU_Result),
        .IF_ID_Addr_out(Rd_Addr),
        .MemReadData_out(MemReadData_out),
        .WB_out({RegWrite, MemtoReg}),
        //input
        .ALU_Result(EX_MEM_ALU_Result),
        .MemReadData(MemReadData),
        .EX_MEM_Addr(EX_MEM_Addr),
        .WB(EX_MEM_WB),
        .clk(clk)
    );
endmodule
```

將 IM、RF、DM、Control、ALU_Control、ALU、Adder、Sign Extension、Bits5_Mux、Bits32_Mux、IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline 組合成一個大的模組，並且使用 wire 將所有的元件串接在一起，注意每個元件的輸出及輸入，有些指輸入 32bits 中的 5bits 或 6bits，分別放入正確的輸出入位置。

IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

I_PipelineCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Test each part with your testbench and explain the results.)

```
// Instruction Memory in Hex
01      // Addr = 0x00
4B      // Addr = 0x01
A0      // Addr = 0x02
0B      // Addr = 0x03
01      // Addr = 0x04
AC      // Addr = 0x05
A8      // Addr = 0x06
0D      // Addr = 0x07
02      // Addr = 0x08
32      // Addr = 0x09
B0      // Addr = 0x0A
27      // Addr = 0x0B
01      // Addr = 0x0C
CF      // Addr = 0x0D
B8      // Addr = 0x0E
2A      // Addr = 0x0F
31      // Addr = 0x10
58      // Addr = 0x11
00      // Addr = 0x12
0A      // Addr = 0x13
35      // Addr = 0x14
```

```
59      // Addr = 0x15
00      // Addr = 0x16
09      // Addr = 0x17
41      // Addr = 0x18
87      // Addr = 0x19
00      // Addr = 0x1A
08      // Addr = 0x1B
45      // Addr = 0x1C
9A      // Addr = 0x1D
00      // Addr = 0x1E
04      // Addr = 0x1F
FF      // Addr = 0x20
FF      // Addr = 0x21
FF      // Addr = 0x22
FF      // Addr = 0x23
```

IM

```
80000000    // R[7]
```

```
0000_0011   // R[10]
0000_0023   // R[11]
0000_0016   // R[12]
0000_0090   // R[13]
0000_0100   // R[14]
0000_0250   // R[15]
0000_0300   // R[16]
0000_0037   // R[17]
0000_0064   // R[18]
0000_0030   // R[19]
0000_0000   // R[20]
```

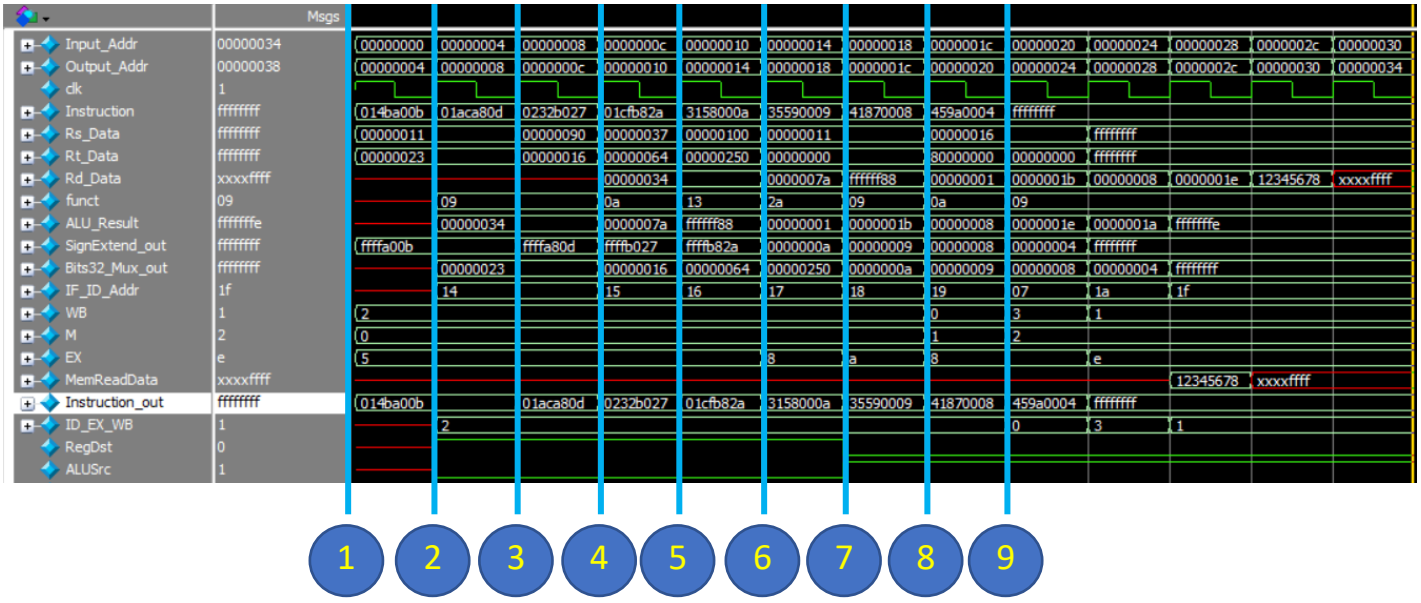
```
00000034    // R[20]
0000007a    // R[21]
ffffff88    // R[22]
00000001    // R[23]
0000001b    // R[24]
00000008    // R[25]
12345678    // R[26]
```

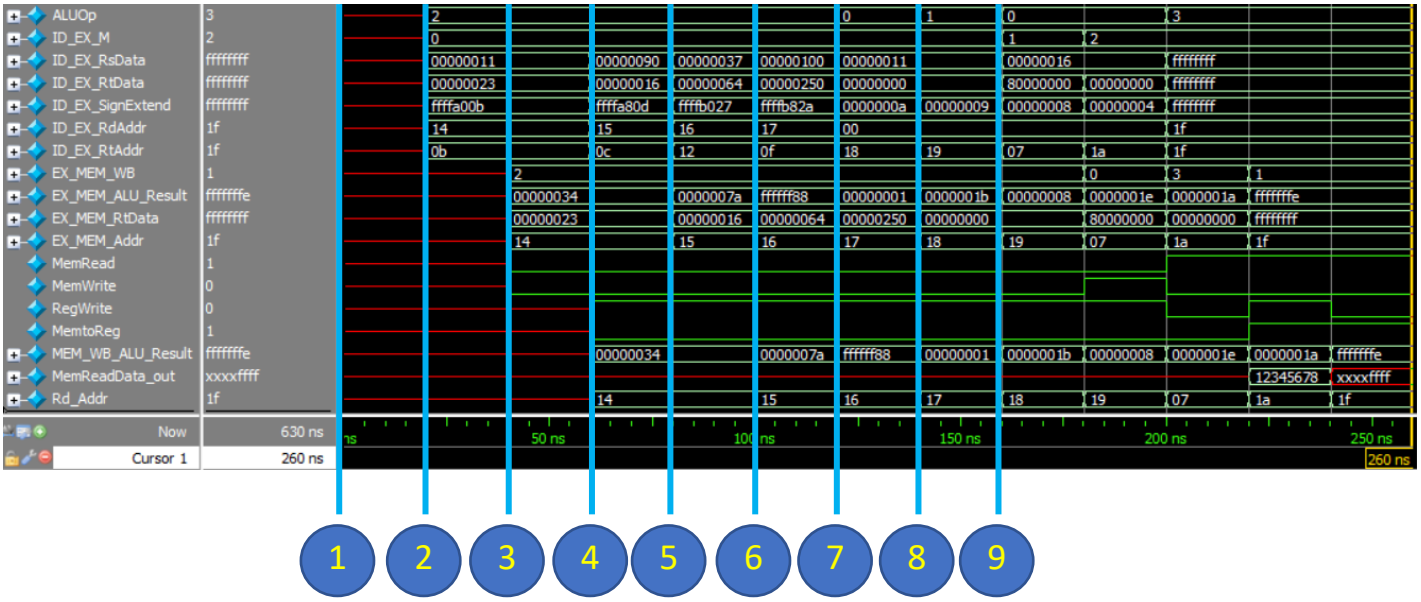
RF.out

RF

```
12  // Addr = 0x1A
34  // Addr = 0x1B
56  // Addr = 0x1C
78  // Addr = 0x1D
80  // Addr = 0x1E
00  // Addr = 0x1F
00  // Addr = 0x20
00  // Addr = 0x21
```

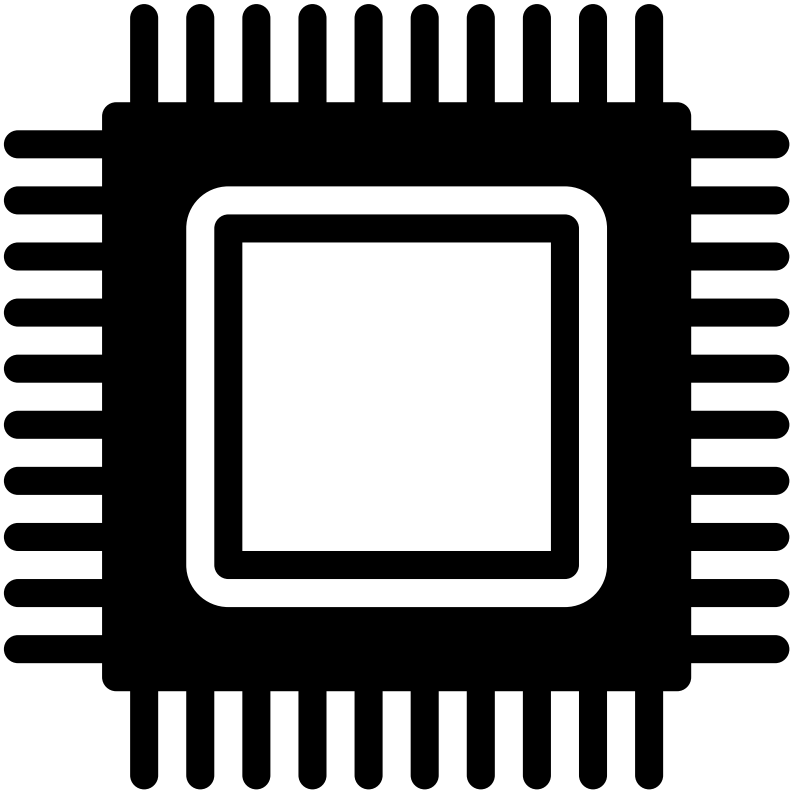
DM.out

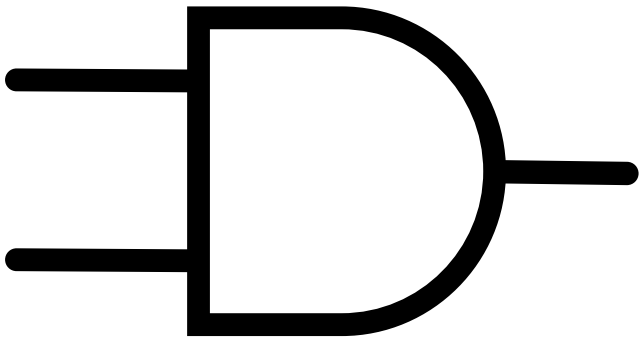


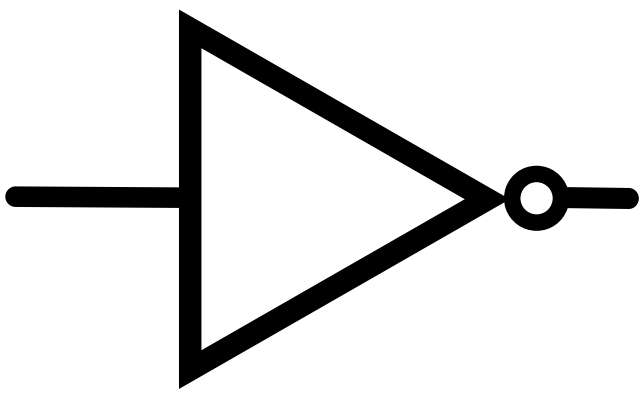


Instruction_Type	Instruction(Binary)						Meaning(Dec)	Data(Hex)		Result(Hex)	IM_Addr(Hex)
R_format	**OP(6)**	**Rs(5)**	**Rt(5)**	**Rd(5)**	**Shamt(5)**	**Funct(6)**	**Ins $Rd, $Rs, $Rt**	**Rs**	**Rt**		
I_format	**OP(6)**	**Rs(5)**	**Rt(5)**	**Immediate(16)**			**Ins $Rt, $Rs, Imm**	**Rs(Beq Rs, Rt)**			
R_format	000000	01010	01011	10100	00000	001011	Addu $20, $10, $11	0000_0011	0000_0023	0000_0034	00
R_format	000000	01101	01100	10101	00000	001101	Subu $21, $13, $12	0000_0090	0000_0016	0000_007A	04
R_format	000000	10001	10010	10110	00000	100111	Nor $22, $17, $18	0000_0037	0000_0064	FFFF_FF88	08
R_format	000000	01110	01111	10111	00000	101010	Sltu $23, $14, $15	0000_0100	0000_0250	0000_0001	0C
I_format	001100	01010	11000	0000000000001010			Addiu $24, $10, 10	0000_0011		0000_001B	10
I_format	001101	01010	11001	0000000000001001			Subiu $25, $10, 9	0000_0011		0000_0008	14
I_format	010000	01100	00111	0000000000001000			Sw $7, 8($12)	0000_0000		Mem[30](Dec)	18
I_format	010001	01100	11010	0000000000000100			Lw $26, 3($12)	1234_5678		Mem[25](Dec)	1C
Undefined	111111	11111	11111	11111	11111	111111	Undefined			Fixed	20

Final CPU







IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

Forwarding、HazardDetection、FinalCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Screenshots of each program, and description of the process.)

```
module IM(
    // Outputs
    output [31:0]Instruction,
    // Inputs
    input [31:0]Instr_Addr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign Instruction = {InstrMem[Instr_Addr], InstrMem[Instr_Addr+1], InstrMem[Instr_Addr+2], InstrMem[Instr_Addr+3]};
endmodule
```

有一個 32bit 的 Instr_Addr，會從 InstrMem 的 Instr_Addr、Instr_Addr+1、Instr_Addr+2、Instr_Addr+3 位置中取出值放到 Instruction 中。

```
module RF(
    // Outputs
    output [31:0] Rs_Data,
    output [31:0] Rt_Data,
    // Inputs
    input [31:0] Rd_Data,
    input [4:0] Rs_Addr,
    input [4:0] Rt_Addr,
    input [4:0] Rd_Addr,
    input RegWrite,
    input clk
);

    /*
     * Declaration of inner register.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [31:0]R[0:`REG_MEM_SIZE - 1];

    //Register MUXs
    assign  Rs_Data = R[Rs_Addr];
    assign  Rt_Data = R[Rt_Addr];
    always@(posedge clk)begin
        if(RegWrite)begin
            R[Rd_Addr] <= Rd_Data;
        end
        else R[Rd_Addr] <= R[Rd_Addr];
    end
endmodule
```

使用輸入訊號 Rs_Addr 和 Rt_Addr 來選擇讀取的 register。在 clk 正緣觸發時，當 RegWrite 信號為 1 時，代表要將 Rd_Data 寫入到 Rd_Addr 中，否則不進行任何寫入操作。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU(
    input [31:0] Src1,
    input [31:0] Src2,
    input [4:0] shamt,
    input [5:0] funct,

    output reg [31:0] Result
);

    always @(Src1, Src2, shamt, funct)begin
        case(funct)
            `Addu:  Result <= Src1 + Src2;
            `Subu:  Result <= Src1 - Src2;
            `Nor:   Result <= ~(Src1 | Src2);
            `Sltu:begin
                if(Src1 < Src2) Result <= 1;
                else Result <= 0;
            end
            default: Result = Result;
        endcase
    end
endmodule
```

定義每個功能的 Function code，在判斷 funct 中的值後，對於資料 Src_1、Src_2 進行相對應的運算，最後運算完的資料傳到 ALUResult。

```
`define R_format            6'b000000
`define Add_imm_unsigned    6'b001100
`define Sub_imm_unsigned    6'b001101
`define Store_word          6'b010000
`define Load_word           6'b010001

module Control(
    //output
    output reg RegWrite,
    output reg[1:0]ALUOp,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemWrite,
    output reg MemRead,
    output reg MemtoReg,
    //input
    input [5:0]OpCode
);
```

```
always@(OpCode)begin
    case(OpCode)
        `R_format: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            ALUSrc <= 0;
            RegDst <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b10;
        end
        `Add_imm_unsigned: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b00;
        end
        `Sub_imm_unsigned: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 0;
            ALUOp <= 2'b01;
        end
```

```
        `Store_word: begin
            RegWrite <= 0;
            MemWrite <= 1;
            MemRead <= 0;
            ALUSrc <= 1;
            ALUOp <= 2'b00;
        end
        `Load_word: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 1;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 1;
            ALUOp <= 2'b00;
        end
        default:begin
            RegWrite <= 0;
            ALUOp <= 2'b11;
        end
    endcase
end
endmodule
```

有輸入訊號 6bits 的 OpCode 以及輸出訊號 RegWrite 和 ALUOp。先判斷 OpCode 的值，當 OpCode 為 6'b000000 (R_format 指令) 時，設定 RegWrite 為 1，讓資料可以寫入 RF，並且將 ALUOp 設為 2'b10。若為 I 指令時，則將 ALUOp 設為 2'b00(Subiu 為 2'b01)，並分別依照其指令之功能改變 RegWrite、RegDst、ALUSrc、MemWrite、MemRead、MemtoReg 的值，使資料透過正確的路徑達到正確的元件執行相對應的功能。

```
module Bits5_Mux(
    //output
    output [4:0]Mux_out,
    //input
    input [4:0]Mux_in_0,
    input [4:0]Mux_in_1,
    input sel
);

assign Mux_out = (sel)? Mux_in_1: Mux_in_0;
endmodule
```

```
module Bits32_Mux(
    //output
    output [31:0]Mux_out,
    //input
    input [31:0]Mux_in_0,
    input [31:0]Mux_in_1,
    input sel
);

assign Mux_out = (sel)? Mux_in_1: Mux_in_0;
endmodule
```

根據 sel 的值，若為 1，Mux_out 中放入 Mux_in_1。若為 0，Mux_out 中放入 Mux_in_0。

```
module Bits32_3input_Mux(
    //output
    output [31:0]Mux_out,
    //input
    input [31:0]Mux_in_0,
    input [31:0]Mux_in_1,
    input [31:0]Mux_in_2,
    input [1:0]sel
);

assign Mux_out = (sel == 2'b00)? Mux_in_0: (sel == 2'b01)? Mux_in_1: (sel == 2'b10)? Mux_in_2: Mux_out;
endmodule
```

根據 sel 的值，若為 2'b 00，Mux_out 中放入 Mux_in_0。若為 2'b 01，Mux_out 中放入 Mux_in_1。若為 2'b 10，Mux_out 中放入 Mux_in_2。

```
module DM(
    // Outputs
    output [31:0] MemReadData,
    // Inputs
    input [31:0] MemAddr,
    input [31:0] MemWriteData,
    input MemWrite,
    input MemRead,
    input clk
);

    /*
     * Declaration of data memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
    assign  MemReadData =(MemRead)?
    {DataMem[MemAddr], DataMem[MemAddr+1], DataMem[MemAddr+2], DataMem[MemAddr+3]}: MemReadData;
    always@(posedge clk)begin
        if(MemWrite)begin
            {DataMem[MemAddr], DataMem[MemAddr+1], DataMem[MemAddr+2], DataMem[MemAddr+3]} <= MemWriteData;
        end
    end
endmodule
```

有四個輸入信號 MemAddr、MemWriteData、MemWrite、MemRead。有一個 MemAddr 32bits 的 Memory 位置輸入，表示要 Write 或 Read 的地址。當 clk 正緣觸發時，判斷 MemWrite 是否為 1，若是，MemWriteData 將被寫入到地址的位置。當 MemRead 被設為 1 時，模組將讀取位址所指示的資料，並輸出到 MemReadData。

```
module SignExtend(
    //output
    output [31:0]out,
    //input
    input [15:0]in
);
assign out = (in[15])? {16'hFFFF, in}: {16'h0000, in};
endmodule
```

對輸入資料 in 進行 Sign Extension 至 32 bits 後輸出到 out。

```
`define Addu    6'b001001
`define Subu    6'b001010
`define Nor     6'b010011
`define Sltu    6'b101010

module ALU_Control(
    output reg[5:0] funct,

    input [1:0] ALUOp,
    input [5:0] funct_ctrl
);
always@(ALUOp, funct_ctrl)begin
    case(ALUOp)
        2'b10:begin
            case(funct_ctrl)
                6'b001011: funct <= `Addu;
                6'b001101: funct <= `Subu;
                6'b100111: funct <= `Nor;
                6'b101010: funct <= `Sltu;
                default: funct <= 6'b0;
            endcase
        end
        2'b01:begin
            funct <= `Subu;
        end
        2'b00:begin
            funct <= `Addu;
        end
    endcase
end
endmodule
```

根據 ALUOp 的值來設定 funct 的值。當 ALUOp 的值為 2'b10 時，判斷 funct_ctrl 的值來設定 funct(Addu[6'b001011]、Subu[6'b001101]、Nor[6'b100111]、Sltu[6'b101010])。當 ALUOp 的值為 2'b10 時，設定 funct 為 Subu[6'b001101]。當 ALUOp 的值為 2'b00 時，設定 funct 為 Addu[6'b001011]。

```
module Adder(
    output [31:0]Output_Addr,
    input [31:0]Src1,
    input [31:0]Src2
);
assign Output_Addr = Src1 + Src2;
endmodule
```

將 Src1 和 Src2 的值相加輸出到 Output_Addr，用來計算下一個 Address。

```
module Forwarding(
    //output
    output reg[1:0] ForwardA = 2'b00,
    output reg[1:0] ForwardB = 2'b00,
    //input
    input [4:0] ID_EX_RtAddr,
    input [4:0] ID_EX_RsAddr,
    input [4:0] EX_MEM_RdAddr,
    input [4:0] MEM_WB_RdAddr,
    input EX_MEM_RegWrite,
    input MEM_WB_RegWrite
);
    always@(*)begin
        if (EX_MEM_RegWrite
            && EX_MEM_RdAddr != 0
            && EX_MEM_RdAddr == ID_EX_RsAddr)
        begin
            ForwardA <= 2'b10;
        end
        else if (MEM_WB_RegWrite
            && MEM_WB_RdAddr != 0
            // && EX_MEM_RdAddr != ID_EX_RsAddr
            && MEM_WB_RdAddr == ID_EX_RsAddr)
        begin
            ForwardA <= 2'b01;
        end
        else ForwardA = 2'b00;
        if (EX_MEM_RegWrite
            && EX_MEM_RdAddr != 0
            && EX_MEM_RdAddr == ID_EX_RtAddr)
        begin
            ForwardB <= 2'b10;
        end
        else if (MEM_WB_RegWrite
            && MEM_WB_RdAddr != 0
            // && EX_MEM_RdAddr != ID_EX_RtAddr
            && MEM_WB_RdAddr == ID_EX_RtAddr)
        begin
            ForwardB <= 2'b01;
        end
        else ForwardB = 2'b00;
    end
endmodule
```

透過判斷 ID_EX_RtAddr、ID_EX_RsAddr、EX_MEM_RdAddr、MEM_WB_RdAddr、EX_MEM_RegWrite、MEM_WB_RegWrite 來控制 ForwardA 及 ForwardB 訊號來選擇需要 Forwarding 的資料。

```
module HazardDetection(
    //output
    output reg IF_ID_Write = 1'b1,
    output reg Stall_Sel = 1'b1,
    output reg PCWrite = 1'b1,
    //input
    input [4:0] ID_EX_RtAddr,
    input [4:0] IF_ID_RsAddr,
    input [4:0] IF_ID_RtAddr,
    input ID_EX_MemRead
);
    always@(*)begin
        if (ID_EX_MemRead && ((ID_EX_RtAddr == IF_ID_RsAddr) || (ID_EX_RtAddr == IF_ID_RtAddr)))begin
            {IF_ID_Write, Stall_Sel, PCWrite} = 3'd0;
        end
        else {IF_ID_Write, Stall_Sel, PCWrite} = 3'b111;
    end
endmodule
```

判斷 ID_EX_RtAddr、IF_ID_RsAddr、IF_ID_RtAddr、ID_EX_MemRead 來控制 IF_ID_Write、Stall_sel、PCWrite 訊號來停止所有寫入的動作一個 clock。

```
module IF_ID_Pipeline(
    //output
    output reg[31:0] out,
    //input
    input [31:0] Instr,
    input IF_ID_Write,
    input clk
);
always@(posedge clk)begin
    if(IF_ID_Write)begin
        out <= Instr;
    end
end
endmodule
```

在 clk 正緣觸發時，並且 IF_ID_Write 為 1 時，Instr 會將資料傳到 out 中輸出。

```
module ID_EX_Pipeline(
    //output
    output reg[31:0] ID_EX_RsData,
    output reg[31:0] ID_EX_RtData,
    output reg[4:0] ID_EX_RdAddr,
    output reg[4:0] ID_EX_RtAddr,
    output reg[4:0] ID_EX_RsAddr,
    output reg[31:0] ID_EX_SignExtend,
    output reg [1:0] ID_EX_M,
    output reg [3:0] ID_EX_EX,
    output reg [1:0]ID_EX_WB,
    //input
    input [31:0] RsData,
    input [31:0] RtData,
    input [4:0] IF_ID_RdAddr,
    input [4:0] IF_ID_RtAddr,
    input [4:0] IF_ID_RsAddr,
    input [31:0] SignExtend,
    input [1:0] M,
    input [3:0] EX,
    input [1:0] WB,
    input clk
);
```

```
always@(posedge clk)begin
    ID_EX_RsData <= RsData;
    ID_EX_RtData <= RtData;
    ID_EX_RdAddr <= IF_ID_RdAddr;
    ID_EX_RtAddr <= IF_ID_RtAddr;
    ID_EX_RsAddr <= IF_ID_RsAddr;
    ID_EX_SignExtend <= SignExtend;
    ID_EX_WB <= WB;
    ID_EX_M <= M;
    ID_EX_EX <= EX;
end
endmodule
```

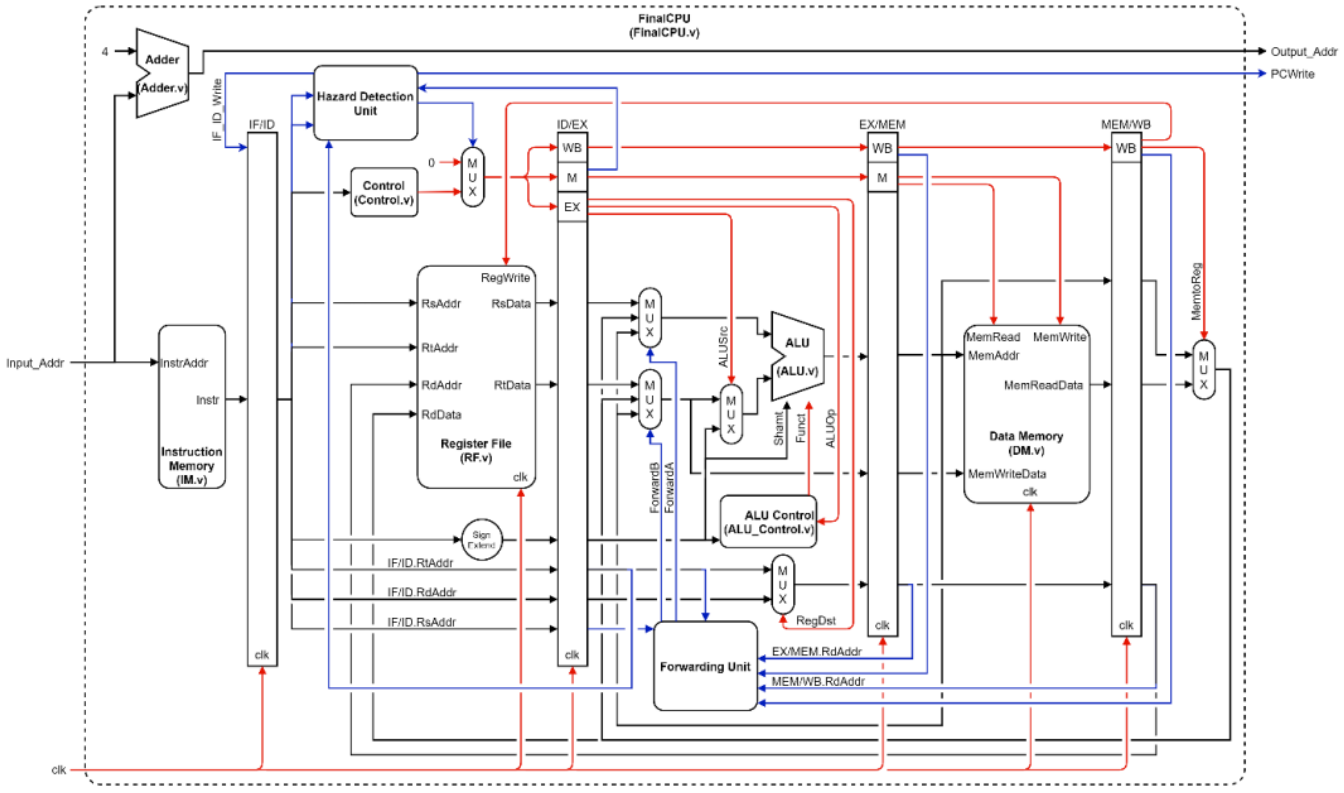
在 clk 正緣觸發時，會將 RsData、RtData、IF_ID_RdAddr、IF_ID_RtAddr、IF_ID_RsAddr、SignExtend、WB、M、EX 中的資料輸出。

```
module EX_MEM_Pipeline(
    //output
    output reg[31:0] EX_MEM_ALU_Result,
    output reg[31:0] EX_MEM_RtData,
    output reg[4:0] EX_MEM_Addr,
    output reg [1:0] EX_MEM_M,
    output reg [1:0]EX_MEM_WB,
    //input
    input [31:0] ALU_Result,
    input [31:0] Rt_Data,
    input [4:0] IF_ID_Addr,
    input [1:0] M,
    input [1:0]WB,
    input clk
);
always@(posedge clk)begin
    EX_MEM_ALU_Result <= ALU_Result;
    EX_MEM_Addr <= IF_ID_Addr;
    EX_MEM_RtData <= Rt_Data;
    EX_MEM_WB <= WB;
    EX_MEM_M <= M;
end
endmodule
```

在 clk 正緣觸發時，會將 ALU_Result、Rt_Data、IF_ID_RdAddr、M、WB 中的資料輸出。

```
module MEM_WB_Pipeline(
    //output
    output reg[31:0] MEM_WB_ALU_Result,
    output reg[31:0] MemReadData_out,
    output reg[4:0] IF_ID_Addr_out,
    output reg [1:0]WB_out,
    //input
    input [31:0] ALU_Result,
    input [31:0] MemReadData,
    input [4:0] EX_MEM_Addr,
    input [1:0]WB,
    input clk
);
always@(posedge clk)begin
    MEM_WB_ALU_Result <= ALU_Result;
    MemReadData_out <= MemReadData;
    IF_ID_Addr_out <= EX_MEM_Addr;
    WB_out <= WB;
end
endmodule
```

在 clk 正緣觸發時，將 ALU_Result、MemReadData、EX_MEM_Addr、WB 中的資料輸出。



```
module FinalCPU(
    // Outputs
    output  wire            PCWrite,
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);
wire [31:0] Instruction;
wire [31:0] Rs_Data;
wire [31:0] Rt_Data;
wire [31:0] Rd_Data;
wire [5:0] funct;
wire [31:0] ALU_Result;
wire [31:0] SignExtend_out;
wire [31:0] Bits32_Mux_out;
wire [4:0] IF_ID_Addr;
//  Control
wire [7:0] Controller_out;
//  Mux_Control
wire [7:0] Mux_Controller_out;
//  DM
wire [31:0] MemReadData;
//  IF/ID
wire [31:0] Instruction_out;
//  HazardDetection
wire Mux_Controller_sel;
wire IF_ID_Write;
```

```
//  ID/EX
wire [1:0]ID_EX_WB;
wire RegDst;
wire ALUSrc;
wire [1:0] ALUOp;
wire [1:0] ID_EX_M;
wire [31:0] ID_EX_RsData;
wire [31:0] ID_EX_RtData;
wire [31:0] ID_EX_SignExtend;
wire [4:0] ID_EX_RdAddr;
wire [4:0] ID_EX_RtAddr;
wire [4:0] ID_EX_RsAddr;
//Forwarding_Unit
wire [1:0] ForwardA;
wire [1:0] ForwardB;
//Mux_ForwardA
wire [31:0] Mux_ForwardA_out;
//Mux_ForwardB
wire [31:0] Mux_ForwardB_out;
//  EX/MEM
wire [1:0]EX_MEM_WB;
wire [31:0] EX_MEM_ALU_Result;
wire [31:0] EX_MEM_RtData;
wire [4:0] EX_MEM_Addr;
wire MemRead;
wire MemWrite;
wire RegWrite;
wire MemtoReg;
wire [31:0] MEM_WB_ALU_Result;
wire [31:0] MemReadData_out;
wire [4:0] Rd_Addr;
```

```
IM Instr_Memory(
    // Outputs
    .Instruction(Instruction),
    // Inputs
    .Instr_Addr(Input_Addr)
);

/*
 * Declaration of Register File.
 * CAUTION: DONT MODIFY THE NAME.
 */
RF Register_File(
// Outputs
.Rs_Data(Rs_Data),
.Rt_Data(Rt_Data),
// Inputs
.Rd_Data(Rd_Data),
.Rs_Addr(Instruction_out[25:21]),
.Rt_Addr(Instruction_out[20:16]),
.Rd_Addr(Rd_Addr),
.RegWrite(RegWrite),
.clk(clk)
);
```

```
DM Data_Memory(
// Outputs
.MemReadData(MemReadData),
// Inputs
.MemAddr(EX_MEM_ALU_Result),
.MemWriteData(EX_MEM_RtData),
.MemWrite(MemWrite),
.MemRead(MemRead),
.clk(clk)
);

Control Controller(
    //output
    .RegWrite(Controller_out[7]),    //WB[1]
    .ALUOp(Controller_out[2:1]),     //EX[2:1]
    .RegDst(Controller_out[0]),      //EX[0]
    .ALUSrc(Controller_out[3]),      //EX[3]
    .MemWrite(Controller_out[4]),    //M[0]
    .MemRead(Controller_out[5]),     //M[1]
    .MemtoReg(Controller_out[6]),    //WB[0]
    //input
    .OpCode(Instruction_out[31:26])
);

ALU_Control ALU_Controller(
    //Outputs
    .funct(funct),
    //Inputs
    .funct_ctrl(ID_EX_SignExtend[5:0]),
    .ALUOp(ALUOp)
);
```

```
ALU Arithmetic(
    // Outputs
    .Result(ALU_Result),
    // Inputs
    .Src1(Mux_ForwardA_out),
    .Src2(Bits32_Mux_out),
    .shamt(ID_EX_SignExtend[10:6]),
    .funct(funct)
);

Adder Addr_Adder(
    //Outputs
    .Output_Addr(Output_Addr),
    //Inputs
    .Src1(Input_Addr),
    .Src2(32'd4)
);

SignExtend SignExtension(
    //Outputs
    .out(SignExtend_out),
    //Inputs
    .in(Instruction_out[15:0])
);
```

```
Bits8_Mux Mux_Controller(
    //output
    .Mux_out(Mux_Controller_out),
    //input
    .Mux_in_0(8'd0),
    .Mux_in_1(Controller_out),
    .sel(Mux_Controller_sel)
);

Bits5_Mux Bits5_Mux(
    //output
    .Mux_out(IF_ID_Addr),
    //input
    .Mux_in_0(ID_EX_RtAddr),
    .Mux_in_1(ID_EX_RdAddr),
    .sel(RegDst)
);

Bits32_Mux Bits32_Mux_ALU(
    //output
    .Mux_out(Bits32_Mux_out),
    //input
    .Mux_in_0(Mux_ForwardB_out),
    .Mux_in_1(ID_EX_SignExtend),
    .sel(ALUSrc)
);
```

```
Bits32_3input_Mux Mux_ForwardA(
    //output
    .Mux_out(Mux_ForwardA_out),
    //input
    .Mux_in_0(ID_EX_RsData),
    .Mux_in_1(Rd_Data),
    .Mux_in_2(EX_MEM_ALU_Result),
    .sel(ForwardA)
);

Bits32_3input_Mux Mux_ForwardB(
    //output
    .Mux_out(Mux_ForwardB_out),
    //input
    .Mux_in_0(ID_EX_RtData),
    .Mux_in_1(Rd_Data),
    .Mux_in_2(EX_MEM_ALU_Result),
    .sel(ForwardB)
);

Bits32_Mux Bits32_Mux_Mem(
    //output
    .Mux_out(Rd_Data),
    //input
    .Mux_in_0(MEM_WB_ALU_Result),
    .Mux_in_1(MemReadData_out),
    .sel(MemtoReg)
);
```

```
Forwarding Forwarding_Unit(
    //output
    .ForwardA(ForwardA),
    .ForwardB(ForwardB),
    //input
    .ID_EX_RtAddr(ID_EX_RtAddr),
    .ID_EX_RsAddr(ID_EX_RsAddr),
    .EX_MEM_RdAddr(EX_MEM_Addr),
    .MEM_WB_RdAddr(Rd_Addr),
    .EX_MEM_RegWrite(EX_MEM_WB[1]),
    .MEM_WB_RegWrite(RegWrite)
);

HazardDetection HazardDetection_Unit(
    //output
    .IF_ID_Write(IF_ID_Write),
    .Stall_Sel(Mux_Controller_sel),
    .PCWrite(PCWrite),
    //input
    .ID_EX_RtAddr(ID_EX_RtAddr),
    .IF_ID_RsAddr(Instruction_out[25:21]),
    .IF_ID_RtAddr(Instruction_out[20:16]),
    .ID_EX_MemRead(ID_EX_M[1])
);
```

```
IF_ID_Pipeline IF_ID_Pipeline(
    //output
    .out(Instruction_out),
    //input
    .Instr(Instruction),
    .IF_ID_Write(IF_ID_Write),
    .clk(clk)
);
ID_EX_Pipeline ID_EX_Pipeline(
    //output
    .ID_EX_RsData(ID_EX_RsData),
    .ID_EX_RtData(ID_EX_RtData),
    .ID_EX_RdAddr(ID_EX_RdAddr),
    .ID_EX_RtAddr(ID_EX_RtAddr),
    .ID_EX_RsAddr(ID_EX_RsAddr),
    .ID_EX_SignExtend(ID_EX_SignExtend),
    .ID_EX_WB(ID_EX_WB),
    .ID_EX_M(ID_EX_M),
    .ID_EX_EX({ALUSrc, ALUOp, RegDst}),
    //input
    .RsData(Rs_Data),
    .RtData(Rt_Data),
    .IF_ID_RdAddr(Instruction_out[15:11]),
    .IF_ID_RtAddr(Instruction_out[20:16]),
    .IF_ID_RsAddr(Instruction_out[25:21]),
    .SignExtend(SignExtend_out),
    .WB(Mux_Controller_out[7:6]),
    .M(Mux_Controller_out[5:4]),
    .EX(Mux_Controller_out[3:0]),
    .clk(clk)
);
```

```
EX_MEM_Pipeline EX_MEM_Pipeline(
    //output
    .EX_MEM_ALU_Result(EX_MEM_ALU_Result),
    .EX_MEM_Addr(EX_MEM_Addr),
    .EX_MEM_M({MemRead, MemWrite}),
    .EX_MEM_RtData(EX_MEM_RtData),
    .EX_MEM_WB(EX_MEM_WB),
    //input
    .ALU_Result(ALU_Result),
    .IF_ID_Addr(IF_ID_Addr),
    .Rt_Data(Mux_ForwardB_out),
    .M(ID_EX_M),
    .WB(ID_EX_WB),
    .clk(clk)
);

MEM_WB_Pipeline MEM_WB_Pipeline(
    //output
    .MEM_WB_ALU_Result(MEM_WB_ALU_Result),
    .IF_ID_Addr_out(Rd_Addr),
    .MemReadData_out(MemReadData_out),
    .WB_out({RegWrite, MemtoReg}),
    //input
    .ALU_Result(EX_MEM_ALU_Result),
    .MemReadData(MemReadData),
    .EX_MEM_Addr(EX_MEM_Addr),
    .WB(EX_MEM_WB),
    .clk(clk)
);

endmodule
```

將 IM、RF、DM、Control、ALU_Control、ALU、Adder、Sign Extension、Bits5_Mux、Bits32_Mux、Bits32_3input_Mux、Forwarding、HazardDetection、IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline 組合成一個大的模組，並且使用 wire 將所有的元件串接在一起，注意每個元件的輸出及輸入，有些指輸入 32bits 中的 5bits 或 6bits，分別放入正確的輸出入位置。

IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

Forwarding、HazardDetection、FinalCPU

IF_ID_Pipeline、ID_EX_Pipeline、EX_MEM_Pipeline、MEM_WB_Pipeline

(Test each part with your testbench and explain the results.)

```
// Instruction Memory in Hex
00      // Addr = 0x00
43      // Addr = 0x01
D0      // Addr = 0x02
0B      // Addr = 0x03
03      // Addr = 0x04
41      // Addr = 0x05
D8      // Addr = 0x06
0D      // Addr = 0x07
45      // Addr = 0x08
9C      // Addr = 0x09
00      // Addr = 0x0A
04      // Addr = 0x0B
41      // Addr = 0x0C
9C      // Addr = 0x0D
00      // Addr = 0x0E
08      // Addr = 0x0F
45      // Addr = 0x10
9D      // Addr = 0x11
00      // Addr = 0x12
08      // Addr = 0x13
03      // Addr = 0x14
AA      // Addr = 0x15
F0      // Addr = 0x16
0B      // Addr = 0x17
33      // Addr = 0x18
DF      // Addr = 0x19
00      // Addr = 0x1A
0A      // Addr = 0x1B
37      // Addr = 0x1C
99      // Addr = 0x1D
00      // Addr = 0x1E
09      // Addr = 0x1F
FF      // Addr = 0x20
FF      // Addr = 0x21
FF      // Addr = 0x22
FF      // Addr = 0x23
FF      // Addr = 0x24
FF      // Addr = 0x25
FF      // Addr = 0x26
FF      // Addr = 0x27
FF      // Addr = 0x28
```

IM

```
12  // Addr = 0x1A
34  // Addr = 0x1B
56  // Addr = 0x1C
78  // Addr = 0x1D
12  // Addr = 0x1E
34  // Addr = 0x1F
56  // Addr = 0x20
78  // Addr = 0x21
```

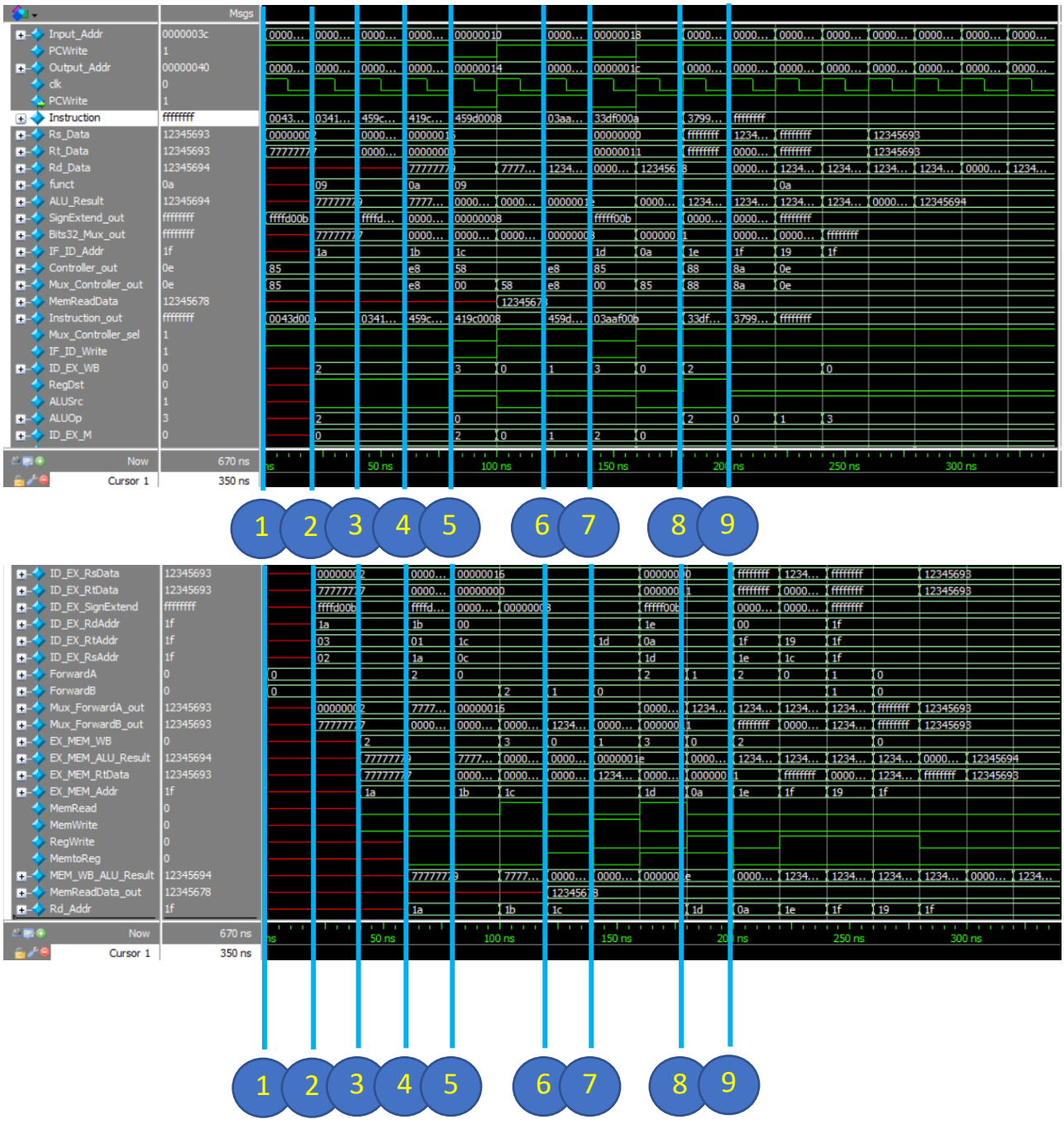
DM

```
1234566f    // R[25]
77777779    // R[26]
77777778    // R[27]
12345678    // R[28]
12345678    // R[29]
12345689    // R[30]
12345693    // R[31]
```

RF.out

```
0000_0000   // R[0]
0000_0001   // R[1]
0000_0002   // R[2]
7777_7777   // R[3]
7F7F_7F7F   // R[4]
F7F7_F7F7   // R[5]
7FFF_FFFF   // R[6]
8000_0000   // R[7]
FFFF_0000   // R[8]
0000_FFFF   // R[9]
0000_0011   // R[10]
0000_0023   // R[11]
0000_0016   // R[12]
0000_0090   // R[13]
0000_0100   // R[14]
0000_0250   // R[15]
0000_0300   // R[16]
0000_0037   // R[17]
0000_0064   // R[18]
0000_0030   // R[19]
0000_0000   // R[20]
0000_0000   // R[21]
0000_0000   // R[22]
0000_0000   // R[23]
0000_0000   // R[24]
0000_0000   // R[25]
0000_0000   // R[26]
0000_0000   // R[27]
0000_0000   // R[28]
0000_0000   // R[29]
FFFF_FFFF   // R[30]
FFFF_FFFF   // R[31]
```

RF



Instruction(Hex)	Instruction_Type	Instruction(Binary)						Meaning(Dec)	Data(Hex)		Result(Hex)	IM_Addr(Hex)
	R_format	OP(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Funct(6)	Ins $Rd, $Rs, $Rt	Rs	Rt		
	I_format	OP(6)	Rs(5)	Rt(5)	Immediate(16)			Ins $Rt, $Rs, Imm	Rs(Beq Rs, Rt)			
0043D00B	R_format	000000	00010	00011	11010	00000	001011	Addu $26, $2, $3	0000_0002	7777_7777	7777_7779	00
0341D80D	R_format	000000	11010	00001	11011	00000	001101	Subu $27, $26, $1	7777_7779	0000_0001	7777_7778	04
459C0004	I_format	010001	01100	11100	0000000000000100			Lw $28, 4($12)	1234_5678		Mem[26](Dec)	08
419C0008	I_format	010000	01100	11100	0000000000001000			Sw $28, 8($12)	1234_5678		Mem[30](Dec)	0C
459D0008	I_format	010001	01100	11101	0000000000001000			Lw $29, 4($12)	1234_5678		Mem[26](Dec)	10
03AAF00B	I_format	000000	11101	01010	11110	00000	001011	Addu $30, $29, $10	1234_5678	0000_0011	1234_5689	14
33DF000A	I_format	001100	11110	11111	0000000000001010			Addiu $31, $30, 10	1234_5689		1234_5693	18
37990009	I_format	001101	11100	11001	0000000000001001			Subiu $25, $28, 9	1234_5678		1234_566F	1C
FFFFFFFF	Undefined	111111	11111	11111	11111	11111	111111	Undefined			Fixed	20

Conclusion and insight

這次的作業是要讓我們做出簡單的有 pipeline 的 CPU。基本上跟上次的 PA 差不多，只是多了 pipeline 的部分。但是在做 Part3 的時候我遇到一個問題就是 HazardDetection 的部分因為在一開始執行的時候都還沒有任何的輸入訊號，所以所有的控制線都會是錯誤的狀態，因此讓我想了很久。後來我才發現可以在宣告輸出的時候就先初始化，這樣就算一開始判斷條件還沒有訊號輸入時，就可以先輸出初始化的值，讓程式順利地執行下去。另外因為在上課的時候，Forwarding 的部分我有一點聽不懂，但是透過這次的 PA 實際操作過一次，我現在已經對於這個步驟比較清楚了，也知道實際運作上要如何去控制。