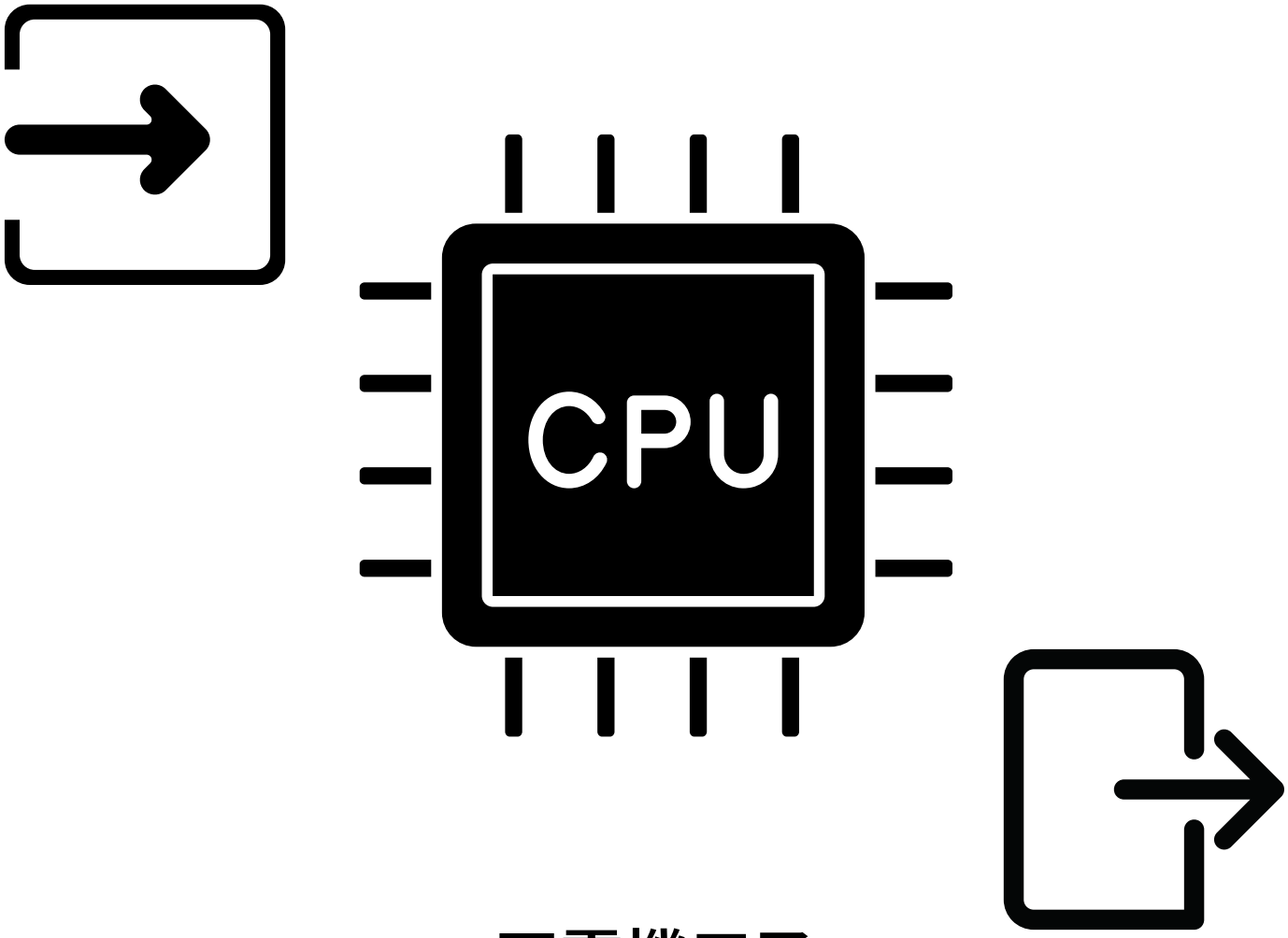
PA2

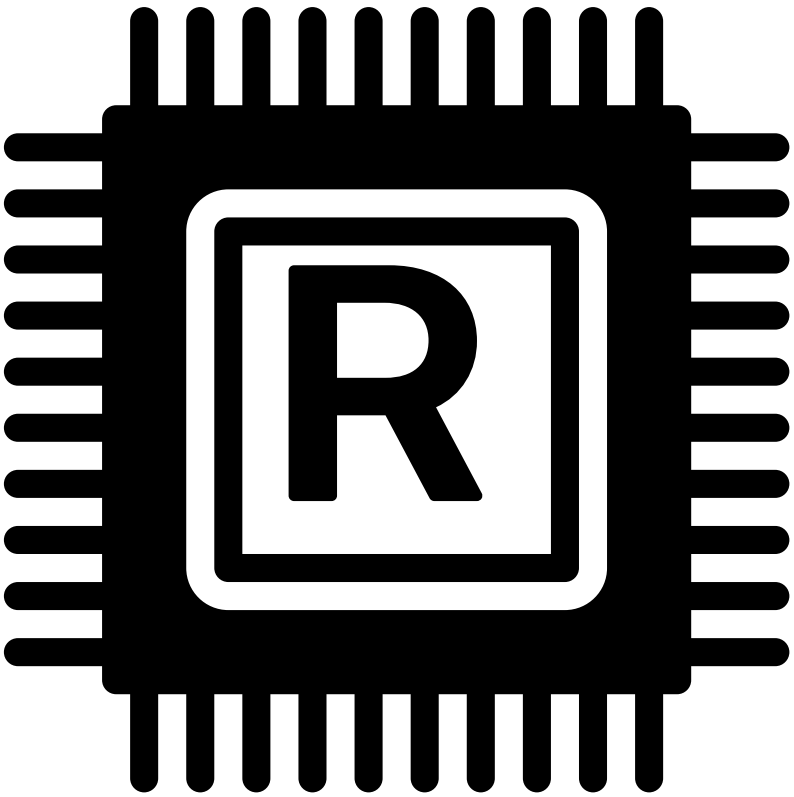
Simple CPU

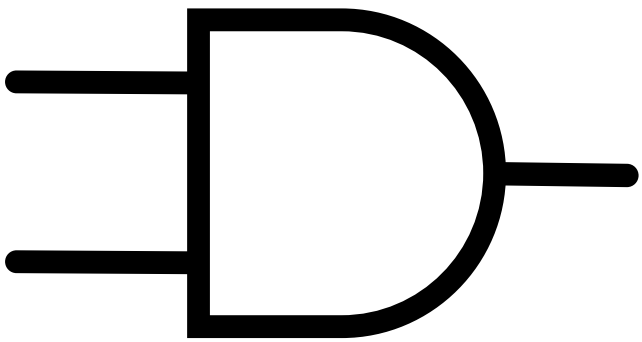


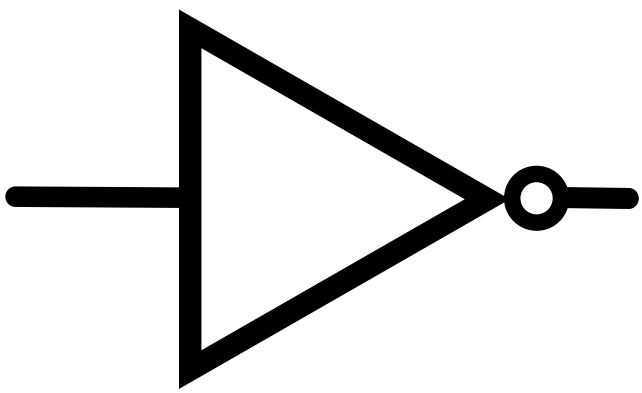
四電機三乙

B10932010 游兆暄

Part 1
R_FormatCPU







IM、RF、ALU、Control

ALU_Control、Adder、R_formatCPU

(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
```

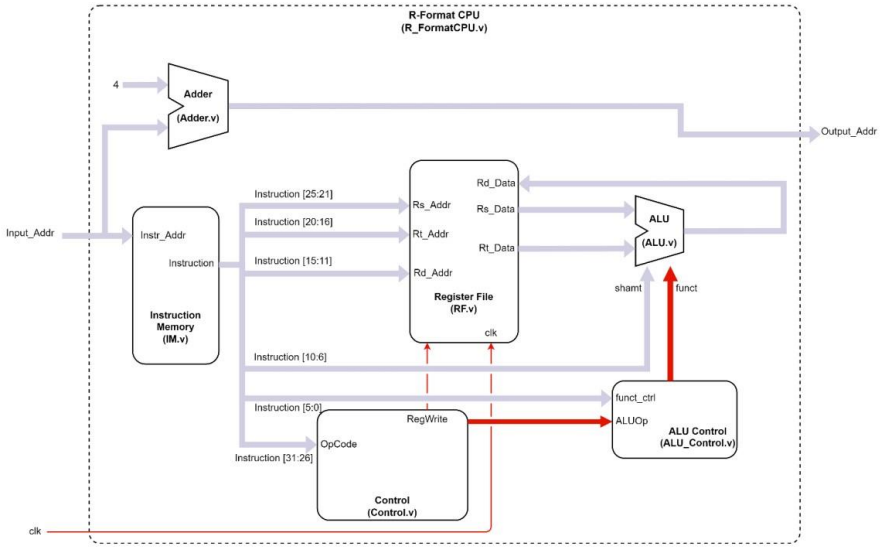
有輸入訊號 6bits 的 OpCode 以及輸出訊號 RegWrite 和 ALUOp。先判斷 OpCode 的值，當 OpCode 為 6'b000000 (R_format 指令) 時，設定 RegWrite 為 1，讓資料可以寫入 RF，並且將 ALUOp 設為 2'b10。

```
`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 R_FormatCPU(
    // Outputs
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);

wire [31:0] Instruction;
wire RegWrite;
wire [1:0] ALUOp;
wire [5:0] funct;
wire [31:0]Rs_Data;
wire [31:0]Rt_Data;
wire [31:0]Rd_Data;

    /*
     * Declaration of Instruction Memory.
     * CAUTION: DONT MODIFY THE NAME.
     */
    IM Instr_Memory(
        // Outputs
        .Instruction(Instruction),
        // Inputs
        .Instr_Addr(Input_Addr)
    );
```

```
RF Register_File(
    // Outputs
    .Rs_Data(Rs_Data),
    .Rt_Data(Rt_Data),
    // Inputs
    .Rd_Data(Rd_Data),
    .Rs_Addr(Instruction[25:21]),
    .Rt_Addr(Instruction[20:16]),
    .Rd_Addr(Instruction[15:11]),
    .RegWrite(RegWrite),
    .clk(clk)
);

ALU Arithmetic(
    // Outputs
    .Result(Rd_Data),
    // Inputs
    .Src1(Rs_Data),
    .Src2(Rt_Data),
    .shamt(Instruction[10:6]),
    .funct(funct)
);

Control Controller(
    // Outputs
    .ALUOp(ALUOp),
    .RegWrite(RegWrite),
    // Inputs
    .OpCode(Instruction[31:26])
);
```

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

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

將 IM、RF、ALU、Control、ALU_Control、Adder、R_formatCPU 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_formatCPU

(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
```

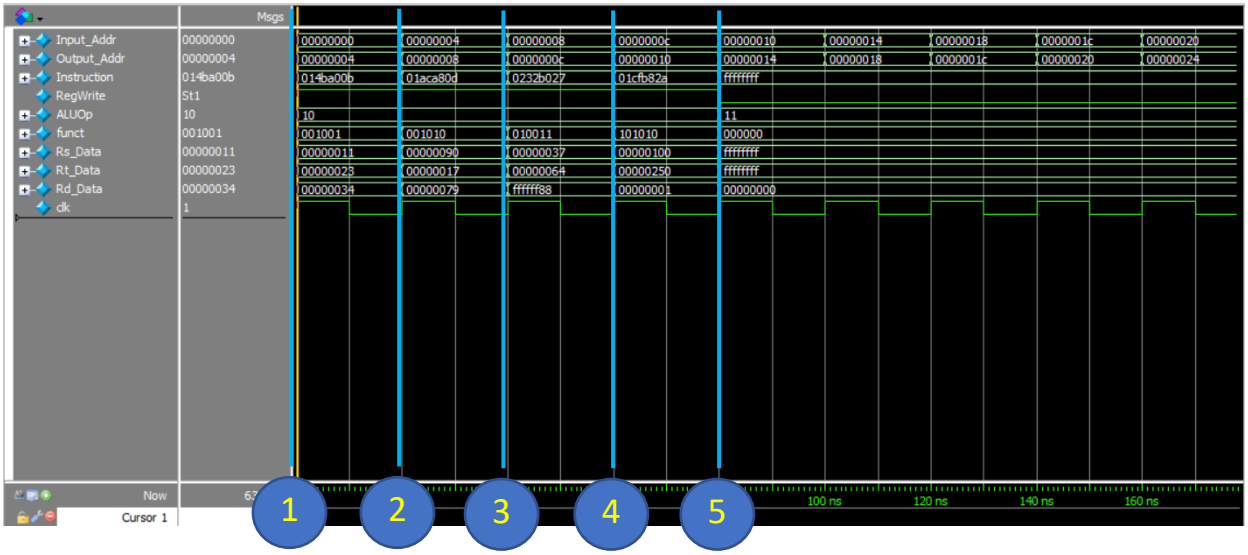
IM

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

RF.out

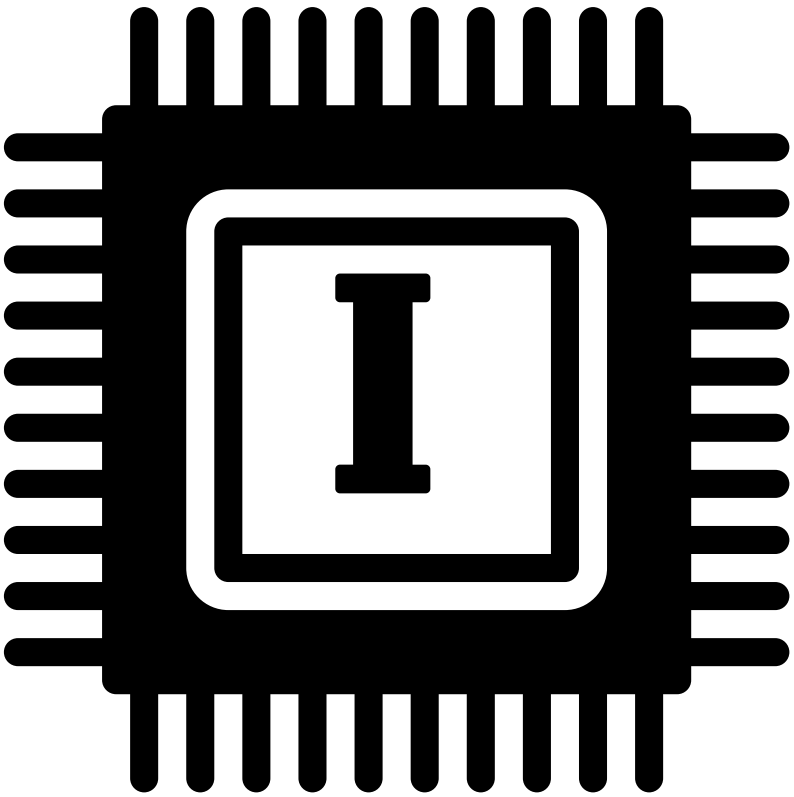
```
0000_0011   // R[10]
0000_0023   // R[11]
0000_0017   // R[12]
0000_0090   // R[13]
0000_0100   // R[14]
0000_0250   // R[15]
0000_0300   // R[16]
0000_0037   // R[17]
0000_0064   // R[18]
0000_0030   // R[19]
0000_0000   // R[20]
```

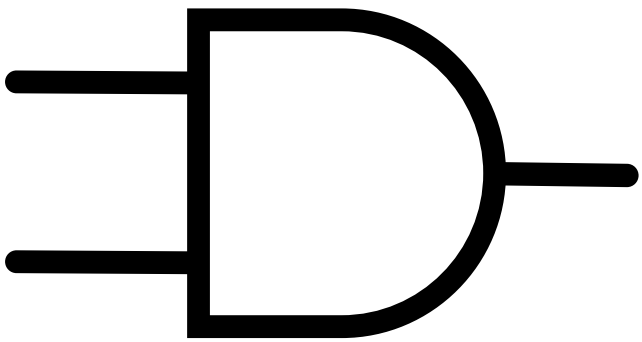
RF

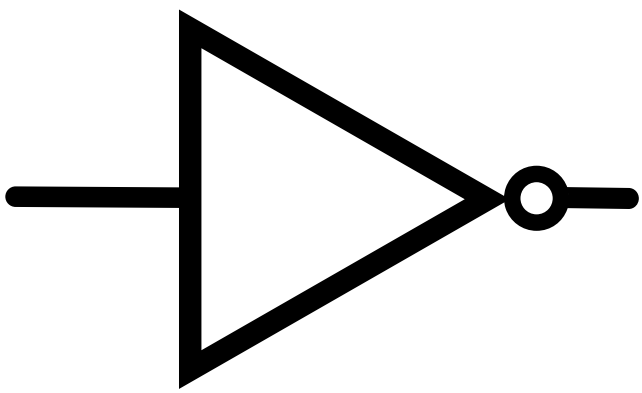


Instruction(Hex)	Instruction_Type	Instruction(Binary)						Meaning(Dec)	Data(Hex)		Result(Hex)	IM_Addr(Hex)
	R_format	OP(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Funct(6)	Ins $Rd, $Rs, $Rt	Rs	Rt		
014BA00B	R_format	000000	01010	01011	10100	00000	001011	Addu $20, $10, $11	0000_0011	0000_0023	0000_0034	00
01ACA80D	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_FormatCPU







IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

I_formatCPU

(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 I_FormatCPU(
    // Outputs
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);

wire [31:0] Instruction;
wire RegWrite;
wire MemWrite;
wire MemRead;
wire MemtoReg;
wire RegDst;
wire ALUSrc;
wire [1:0] ALUOp;
wire [5:0] funct;
wire [31:0] Rs_Data;
wire [31:0] Rt_Data;
wire [31:0] Rd_Data;
wire [4:0] Bits5_Mux_out;
wire [31:0]ALU_out;
wire [31:0] Bits32_Mux_out;
wire [31:0] SignExtend_out;
wire [31:0] MemReadData;
```

```
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[25:21]),
.Rt_Addr(Instruction[20:16]),
.Rd_Addr(Bits5_Mux_out),
.RegWrite(RegWrite),
.clk(clk)
);
```

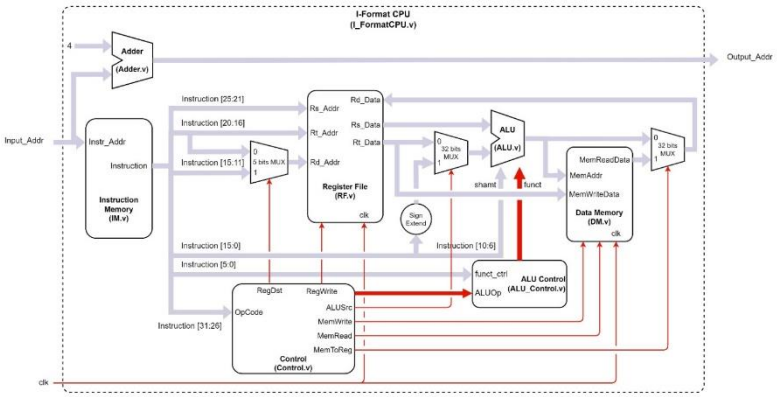
```
DM Data_Memory(
// Outputs
.MemReadData(MemReadData),
// Inputs
.MemAddr(ALU_out),
.MemWriteData(Rt_Data),
.MemWrite(MemWrite),
.MemRead(MemRead),
.clk(clk)
);

Control Controller(
    //output
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    //input
    .OpCode(Instruction[31:26])
);
```

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

ALU Arithmetic(
    // Outputs
    .Result(ALU_out),
    // Inputs
    .Src1(Rs_Data),
    .Src2(Bits32_Mux_out),
    .shamt(Instruction[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[15:0])
);

Bits5_Mux Bits5_Mux(
    //output
    .Mux_out(Bits5_Mux_out),
    //input
    .Mux_in_0(Instruction[20:16]),
    .Mux_in_1(Instruction[15:11]),
    .sel(RegDst)
);

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

```
    Bits32_Mux Bits32_Mux_Mem(
        //output
        .Mux_out(Rd_Data),
        //input
        .Mux_in_0(ALU_out),
        .Mux_in_1(MemReadData),
        .sel(MemtoReg)
    );
endmodule
```

將 IM、RF、DM、Control、ALU_Control、ALU、Adder、Sign Extension、Bits5_Mux 和 Bits32_Mux 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_formatCPU

(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
98      // Addr = 0x19
00      // Addr = 0x1A
03      // Addr = 0x1B
45      // Addr = 0x1C
9A      // Addr = 0x1D
00      // Addr = 0x1E
03      // Addr = 0x1F
FF      // Addr = 0x20
FF      // Addr = 0x21
FF      // Addr = 0x22
FF      // Addr = 0x23
```

IM

```
00000034    // R[20]
00000079    // R[21]
ffffff88    // R[22]
00000001    // R[23]
0000001b    // R[24]
00000008    // R[25]
0000001b    // R[26]
```

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]
```

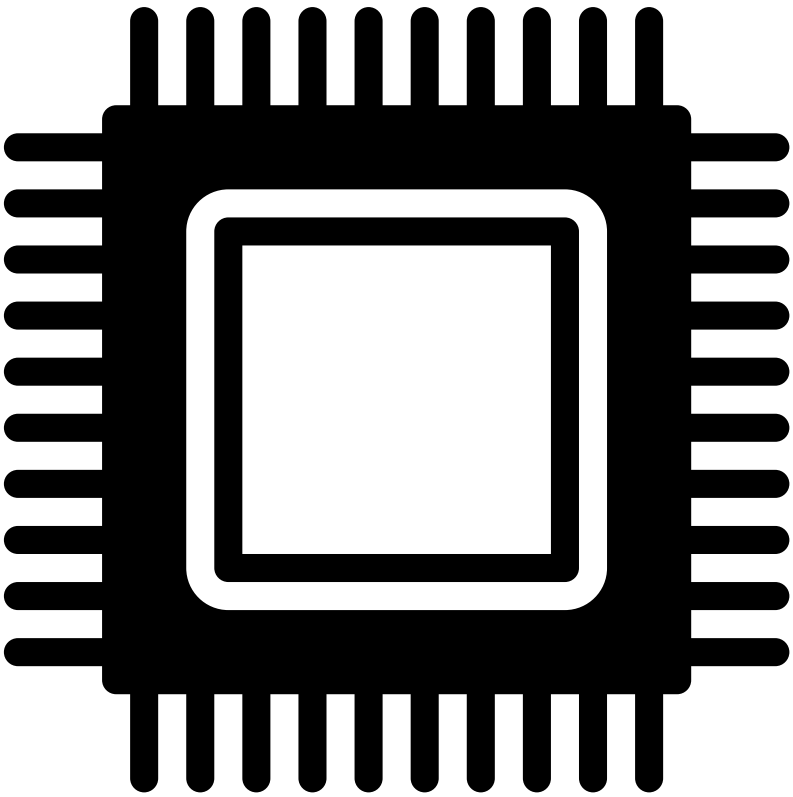
RF

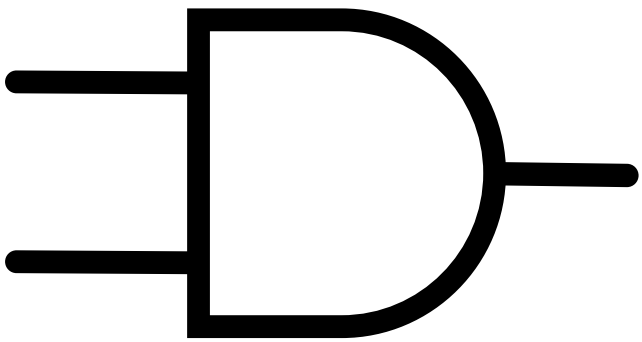
```
00  // Addr = 0x1A
00  // Addr = 0x1B
00  // Addr = 0x1C
1b  // Addr = 0x1D
```

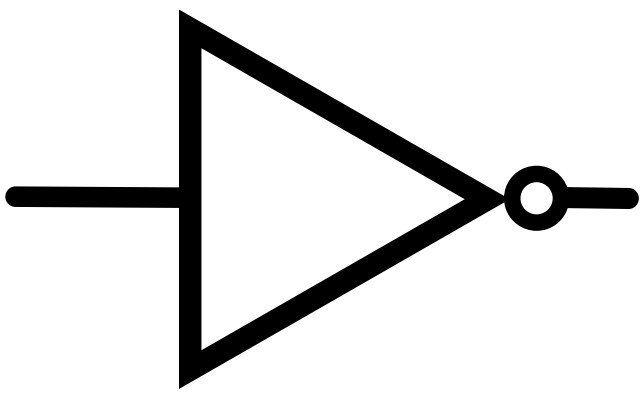
DM.out

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)**			
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
3158000A	I_format	001100	01010	11000	0000000000001010			Addiu $24, $10, 10	0000_0011		0000_001B	10
35590009	I_format	001101	01010	11001	0000000000001001			Subiu $25, $10, 9	0000_0011		0000_0008	14
41980003	I_format	010000	01100	11000	0000000000000011			Sw $24, 3($12)	0000_0017		Mem[26](Dec)	18
459A0003	I_format	010001	01100	11010	0000000000000011			Lw $26, 3($12)	0000_0017		Mem[26](Dec)	1C
FFFFFFFF	Undefined	111111	11111	11111	11111	11111	111111	Undefined			Fixed	20

Part 3
Simple CPU







IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

Shifter、SimpleCPU

(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;
            Jump <= 0;
            Branch <= 0;
            ALUOp <= 2'b10;
        end
        `Add_imm_unsigned: begin
            RegWrite <= 1;
            MemWrite <= 0;
            MemRead <= 0;
            RegDst <= 0;
            ALUSrc <= 1;
            MemtoReg <= 0;
            Jump <= 0;
            Branch <= 0;
            ALUOp <= 2'b00;
        end
```

```
`Load_word: begin
    RegWrite <= 1;
    MemWrite <= 0;
    MemRead <= 1;
    RegDst <= 0;
    ALUSrc <= 1;
    Jump <= 0;
    Branch <= 0;
    MemtoReg <= 1;
    ALUOp <= 2'b00;
end
`Jump:begin
    Jump <= 1;
    Branch <= 0;
    RegWrite <= 0;
    MemWrite <= 0;
    MemRead <= 0;
    ALUOp <= 2'b01;
end
`Branch_on_equal:begin
    ALUSrc <= 0;
    Branch <= 1;
    Jump <= 0;
    RegWrite <= 0;
    MemWrite <= 0;
    MemRead <= 0;
    ALUOp <= 2'b01;
end
```

```
`Sub_imm_unsigned: begin
    RegWrite <= 1;
    MemWrite <= 0;
    MemRead <= 0;
    RegDst <= 0;
    ALUSrc <= 1;
    MemtoReg <= 0;
    ALUOp <= 2'b01;
end
```

```
        default:begin
            RegWrite <= 0;
            Jump <= 0;
            Branch <= 0;
            RegWrite <= 0;
            MemWrite <= 0;
            MemRead <= 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)，若為 J 指令時，則將 ALUOp 設為 2'b01，並分別依照其指令之功能改變 RegWrite、RegDst、ALUSrc、MemWrite、MemRead、MemtoReg、Jump、Branch 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 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 Shift(
    //output
    output [31:0]out,
    //input
    input [31:0]in
);
assign out = in << 2;
endmodule
```

對輸入資料 in 進行左移 2 bits 後輸出到 out。

```
module And_Gate(
    //output
    output out,
    //input
    input Src1,
    input Src2
);
assign out = Src1 && Src2;
endmodule
```

對 Src1 和 Src2 進行 And 並輸出到 out。

```
module SimpleCPU(
    // Outputs
    output  wire    [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);
wire [31:0] Instruction;
wire RegWrite;
wire MemWrite;
wire MemRead;
wire MemtoReg;
wire RegDst;
wire ALUSrc;
wire Branch;
wire Zero;
wire Mux_Branch_sel;
wire [1:0] ALUOp;
wire [5:0] funct;
wire [31:0] Rs_Data;
wire [31:0] Rt_Data;
wire [31:0] Rd_Data;
wire [4:0] Bits5_Mux_out;
wire [31:0]ALU_out;
wire [31:0] Bits32_Mux_out;
wire [31:0] SignExtend_out;
wire [31:0] MemReadData;
wire [31:0] Adder_InAddr_out;
wire [31:0] Adder_Branch_in;
wire [31:0] Adder_Branch_out;
wire [31:0] Mux_Branch_out;
```

```
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[25:21]),
.Rt_Addr(Instruction[20:16]),
.Rd_Addr(Bits5_Mux_out),
.RegWrite(RegWrite),
.clk(clk)
);
```

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

Control Controller(
    //output
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .Branch(Branch),
    .Jump(Jump),
    //input
    .OpCode(Instruction[31:26])
);
```

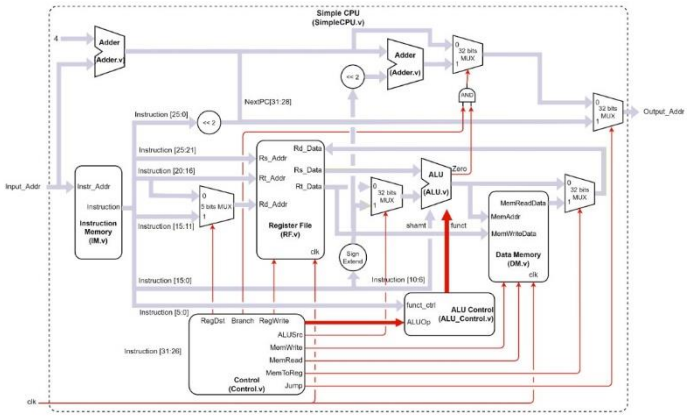
```
ALU_Control ALU_Controller(
    //Outputs
    .funct(funct),
    //Inputs
    .funct_ctrl(Instruction[5:0]),
    .ALUOp(ALUOp)
);

ALU Arithmetic(
    // Outputs
    .Result(ALU_out),
    .zero(Zero),
    // Inputs
    .Src1(Rs_Data),
    .Src2(Bits32_Mux_out),
    .shamt(Instruction[10:6]),
    .funct(funct)
);

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

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

Bits5_Mux Bits5_Mux(
    //output
    .Mux_out(Bits5_Mux_out),
    //input
    .Mux_in_0(Instruction[20:16]),
    .Mux_in_1(Instruction[15:11]),
    .sel(RegDst)
);

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

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

Adder Adder_Branch(
    //Outputs
    .Output_Addr(Adder_Branch_out),
    //Inputs
    .Src1(Adder_Branch_in),
    .Src2(Adder_InAddr_out)
);

Bits32_Mux Bits32_Mux_Branch(
    //output
    .Mux_out(Mux_Branch_out),
    //input
    .Mux_in_0(Adder_InAddr_out),
    .Mux_in_1(Adder_Branch_out),
    .sel(Mux_Branch_sel)
);

Bits32_Mux Bits32_Mux_OutAddr(
    //output
    .Mux_out(Output_Addr),
    //input
    .Mux_in_0(Mux_Branch_out),
    .Mux_in_1({Adder_InAddr_out[31:28], Instruction[25:0], 2'b00}),
    .sel(Jump)
);
```

```
    Shift Shifter_Branch(
        //output
        .out(Adder_Branch_in),
        //input
        .in(SignExtend_out)
    );

    And_Gate And_Gate_Branch(
        //output
        .out(Mux_Branch_sel),
        //input
        .Src1(Branch),
        .Src2(Zero)
    );


endmodule
```

將 IM、RF、DM、Control、ALU_Control、ALU、Adder、Sign Extension、Shifter、Bits5_Mux 和 Bits32_Mux 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。另外 Output_Addr 的 Mux 輸入資料時因為需要位移，但是與 Shifter 的位元數不相符，因此在輸入時直接輸入相應的位元數而不使用 Shifter。



IM、RF、ALU、Control、MUX

DM、Sign_Extend、ALU_Control、Adder

Shifter、SimpleCPU

(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
98      // Addr = 0x19
00      // Addr = 0x1A
03      // Addr = 0x1B
45      // Addr = 0x1C
9A      // Addr = 0x1D
00      // Addr = 0x1E
03      // Addr = 0x1F
4F      // Addr = 0x20
DF      // Addr = 0x21
00      // Addr = 0x22
12      // Addr = 0x23
FF      // Addr = 0x24
FF      // Addr = 0x25
FF      // Addr = 0x26
FF      // Addr = 0x27
01      // Addr = 0x28
EB      // Addr = 0x29
E0      // Addr = 0x2A
0B      // Addr = 0x2B
70      // Addr = 0x2C
00      // Addr = 0x2D
00      // Addr = 0x2E
1D      // Addr = 0x2F
FF      // Addr = 0x30
FF      // Addr = 0x31
FF      // Addr = 0x32
FF      // Addr = 0x33
```

IM

```
00000034    // R[20]
00000079    // R[21]
ffffff88    // R[22]
00000001    // R[23]
0000001b    // R[24]
00000008    // R[25]
0000001b    // R[26]
0000002e    // R[27]
00000273    // R[28]
```

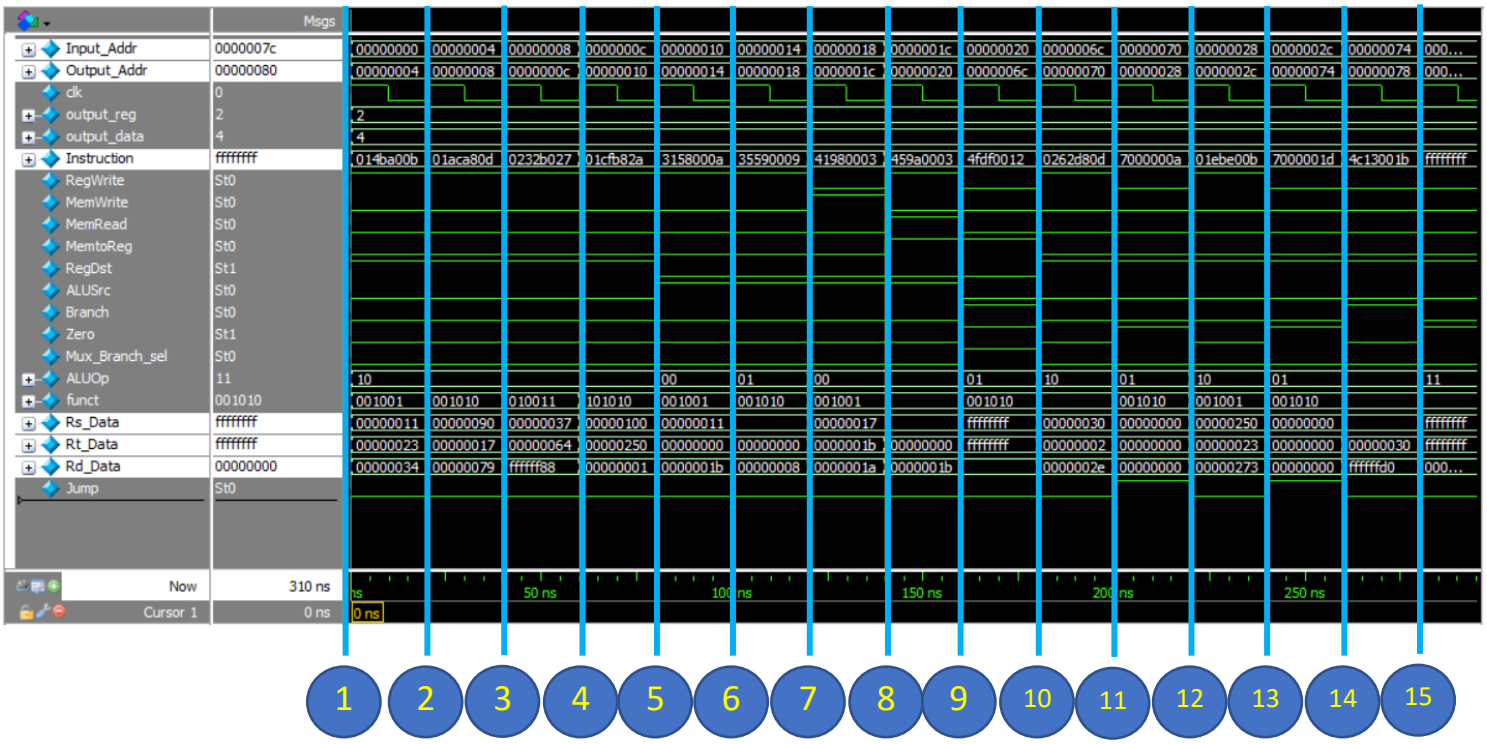
RF.out

```
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]
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

```
00  // Addr = 0x1A
00  // Addr = 0x1B
00  // Addr = 0x1C
1b  // Addr = 0x1D
```

DM.out



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)			
	J_format	OP(6)	Immediate(26)					Ins Imm				
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
3158000A	I_format	001100	01010	11000	0000000000001010			Addiu $24, $10, 10	0000_0011		0000_001B	10
35590009	I_format	001101	01010	11001	0000000000001001			Subiu $25, $10, 9	0000_0011		0000_0008	14
41980003	I_format	010000	01100	11000	0000000000000011			Sw $24, 3($12)	0000_0017		Mem[26](Dec)	18
459A0003	I_format	010001	01100	11010	0000000000000011			Lw $26, 3($12)	0000_0017		Mem[26](Dec)	1C
4FDF0012	I_format	010011	11110	11111	0000000000010010			Beq $30, $31, 18	FFFF_FFFF	FFFF_FFFF	eq jump to 6C	20
0262D80D	R_format	000000	10011	00010	11011	00000	001101	Subu $27, $19, $2	0000_0030	0000_0002	0000_002E	6C
7000000A	J_format	011100	00000000000000000000001010					J 10			Jump to 28	70
01EBE00B	R_format	000000	01111	01011	11100	00000	001011	Addu $28, $15, $11	0000_0250	0000_0023	0000_0273	28
7000001D	J_format	011100	00000000000000000000011101					J 29			Jump to 74	2C
4C13001E	I_format	010011	00000	10011	0000000000011110			Beq $0, $19, 30	0000_0000	0000_0030	No Branch	74
FFFFFFFF	Undefined	111111	11111	11111	11111	11111	111111	Undefined			Fixed	78

Conclusion and insight

這次的作業是要讓我們做出簡單的 CPU。基本上在做的時候沒有問題，需要比較注意的就是 Controller 在執行不同指令時訊號一定要給對，不然會因為資料走到不對的路徑，造成錯誤的結果。另外需要注意的是設計指令時，J-type 的情況有可能會出現一個問題，就是 Jump 指令如果跳到前面的地址，且沒有其他指令可以直接跳過此 Jump 指令，那就會造成無限迴圈，一直在執行一樣的指令。因此可以在前面加上 Beq 或者 Jump 讓指令可以繼續執行下去。另外也要注意 Jump 指令不要跳到自己身上，不然一樣會造成無限迴圈的問題。