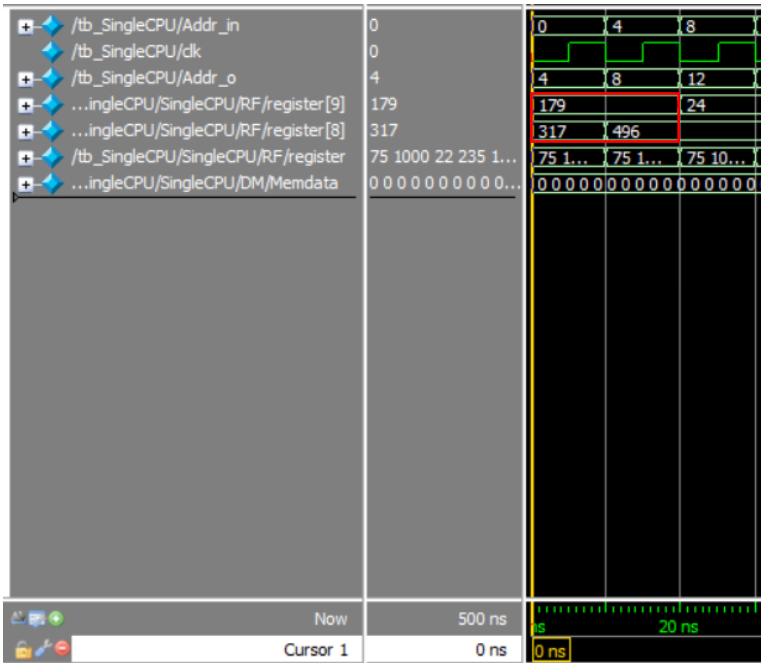
這次報告將會先展示模擬成果，最後再討論各項 module 程式碼內容

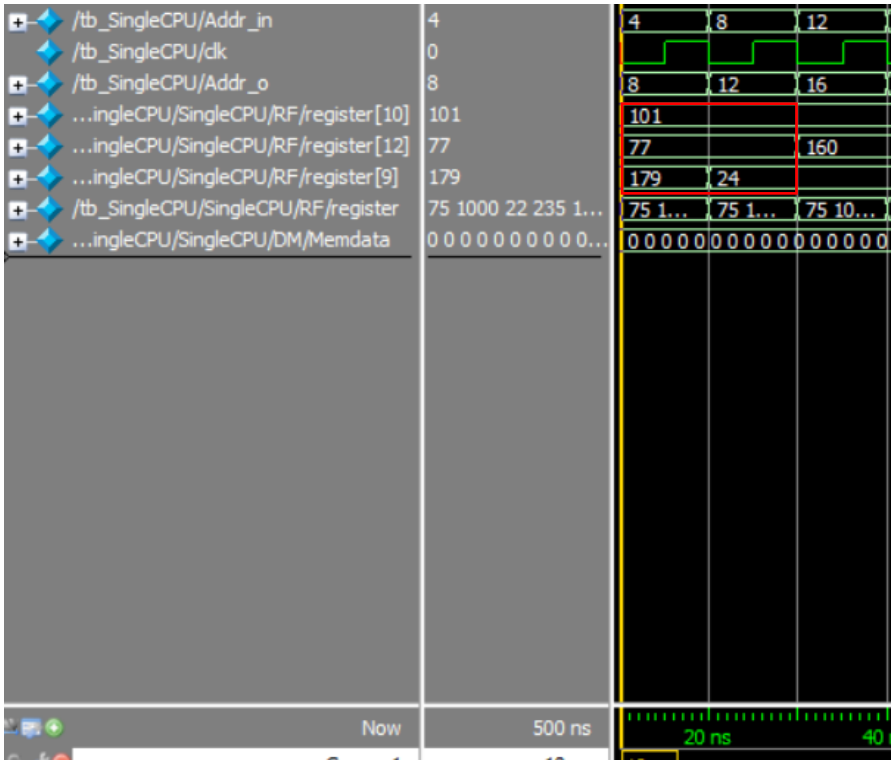
Part1: R-Format

```
module IM(
        input [31:0] Addr_in,//the value is the address of running instruction
        output reg [31:0] Instruction//run instruction
);
        reg [31:0]Instr[99:0];//Creat 100 Instruction address each is 32-bit


always@(Addr_in)begin
        Instruction=Instr[Addr_in/4];//the address of instruction is 4times
end

initial begin
        Instr[0]=32'b010100_01000_01001_01000_00000_010101;//add $t0, $t0, $t1
        Instr[1]=32'b010100_01010_01100_01001_00000_010110;//sub $t1, $t2, $t4
        Instr[2]=32'b010100_01101_00000_01100_00001_010111;//srl $t4, $t5, 1
        Instr[3]=32'b010100_01111_00000_01110_00100_011000;//sll $t6, $t7, 4
        Instr[4]=32'b010100_01001_01010_01011_00000_011001;//xor $t3, $t1, $t2
        Instr[5]=32'b010100_01010_01100_01101_00000_011010;//and $t5, $t4, $t2

        Instr[6]=32'b101011_01111_01000_0000000000000010;//sw $t0, 2($t7)
        Instr[7]=32'b100011_01111_10001_0000000000000010;//lw $s1, 2($t7)
        Instr[8]=32'b100011_01111_10010_0000000000000100;//lw $s2, 4($t7)
        Instr[9]=32'b101011_01010_01000_0000000000000010;//sw $t0, 2($t2)
        Instr[10]=32'b101011_01001_10011_0000000000000100;//sw $s3, 4($t1)
        Instr[11]=32'b001000_10011_10100_0000000001101111;//addi $s4, $s3, 111
        Instr[12]=32'b001000_10101_10110_0000000000011011;//addi $s6, $s5, 27
        Instr[13]=32'b001001_10110_10001_0000000000001001;//subi $s1, $s6, 9
        Instr[14]=32'b001001_10001_10111_0000000000000101;//subi $s7, $s1, 5
```

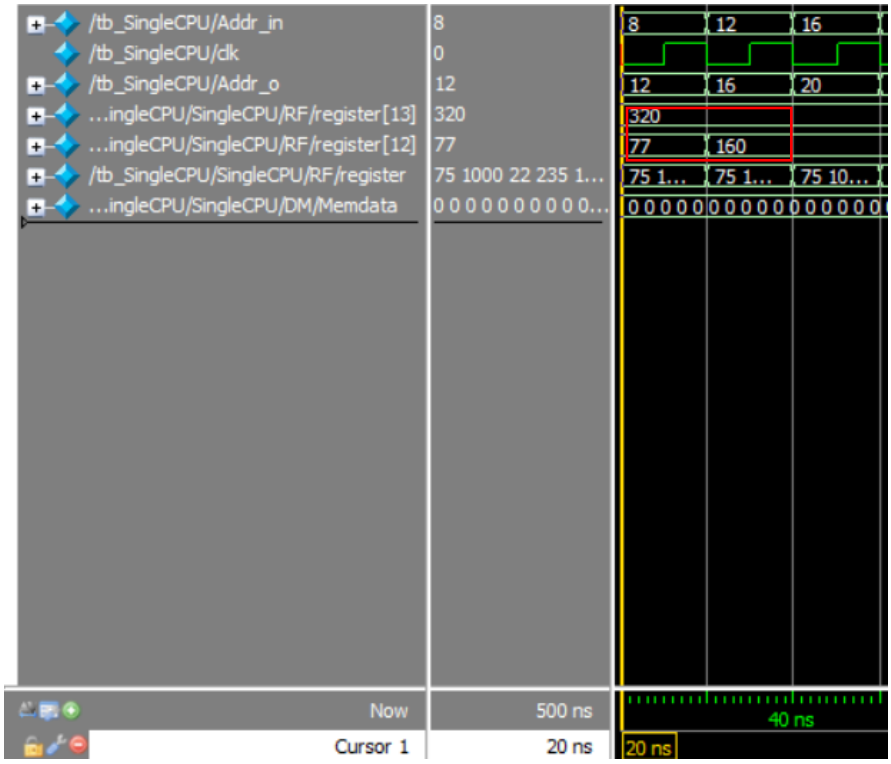
1. Instr[0]=>$t0($8)=496=$t0(317)+$t1(179)



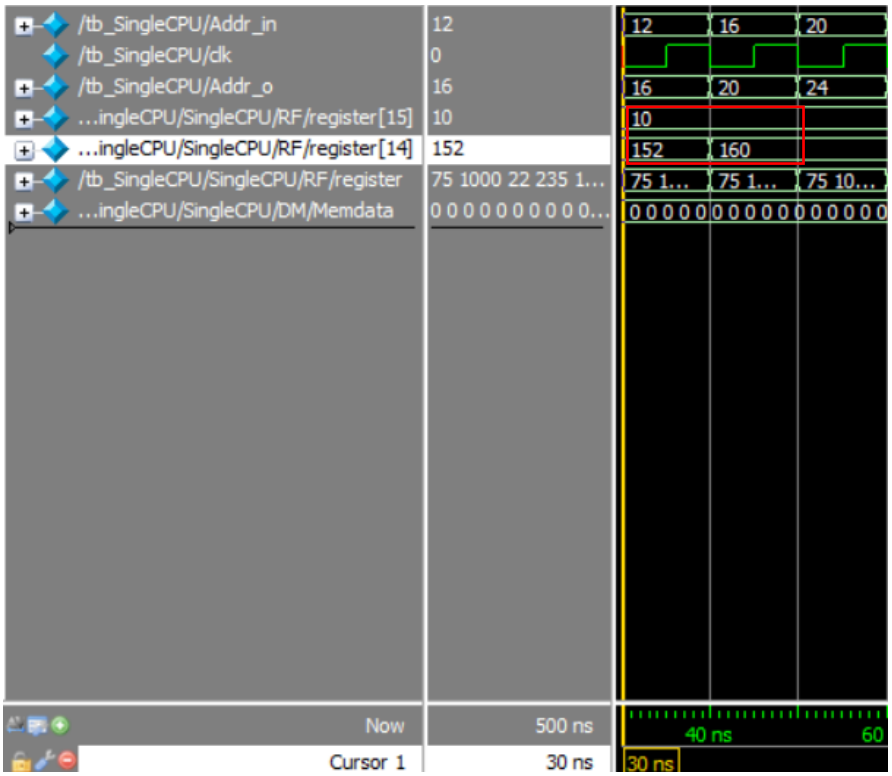
2. Instr[1]=>$t1($9)=24=$t2(101)-$t4(77)



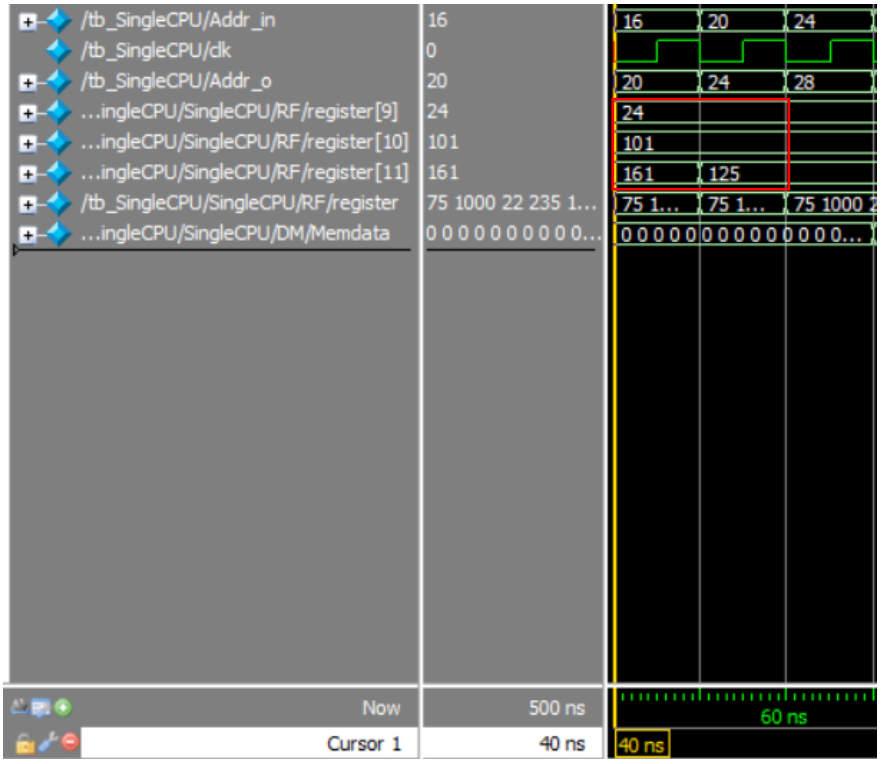
3. Instr[2]=>$t4($12)=160=$t5(320)/2



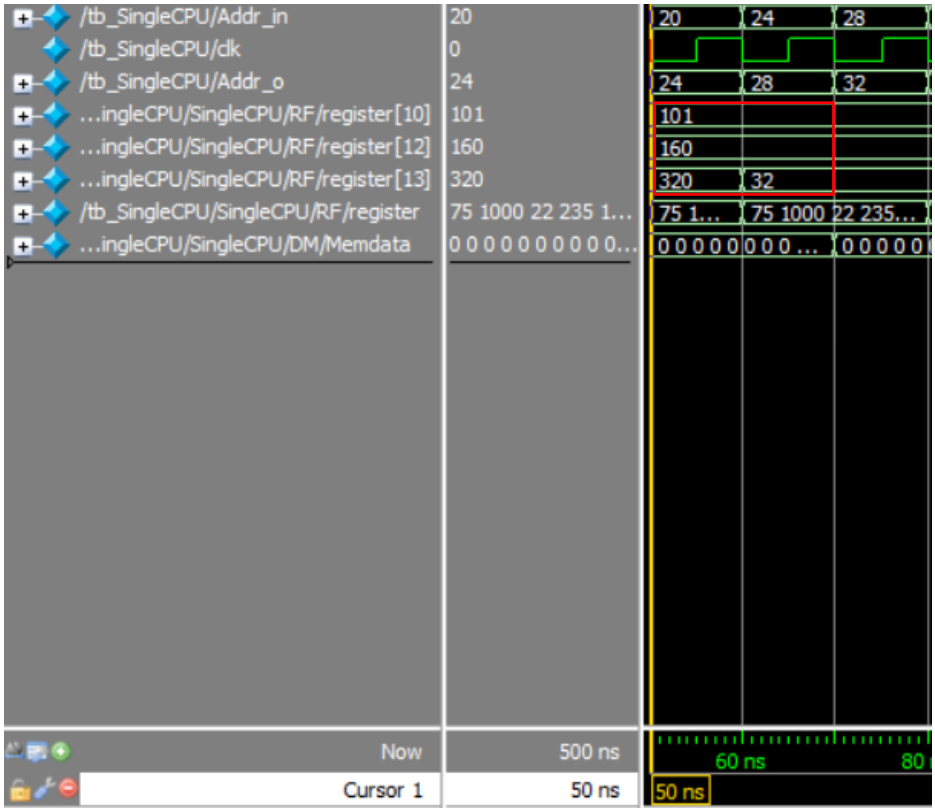
4. Instr[3]=>$t6($14)=160=$t7(10)*16



5. Instr[4]=>$t3($11)=125(01111101)=$t1(00011000)XOR$t2(01100101)



6. Instr[5]=>$t5($13)=32(00100000)=$t4(10100000)AND$t2(01100101)

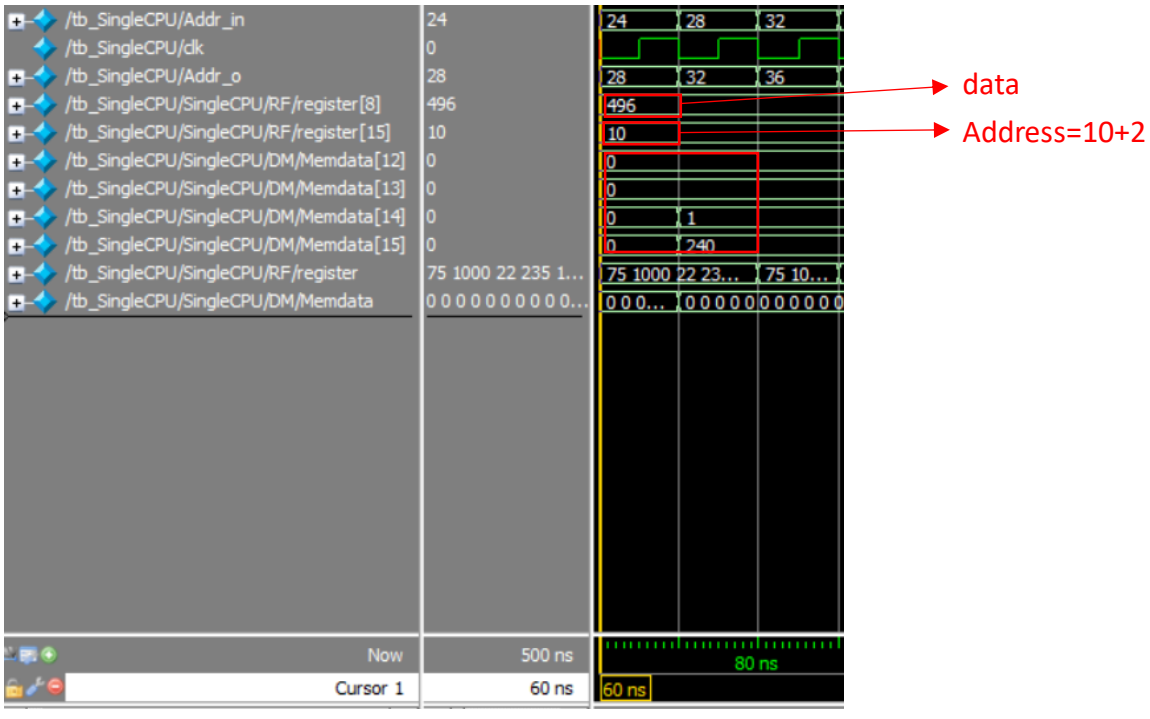


Part2: I-Format

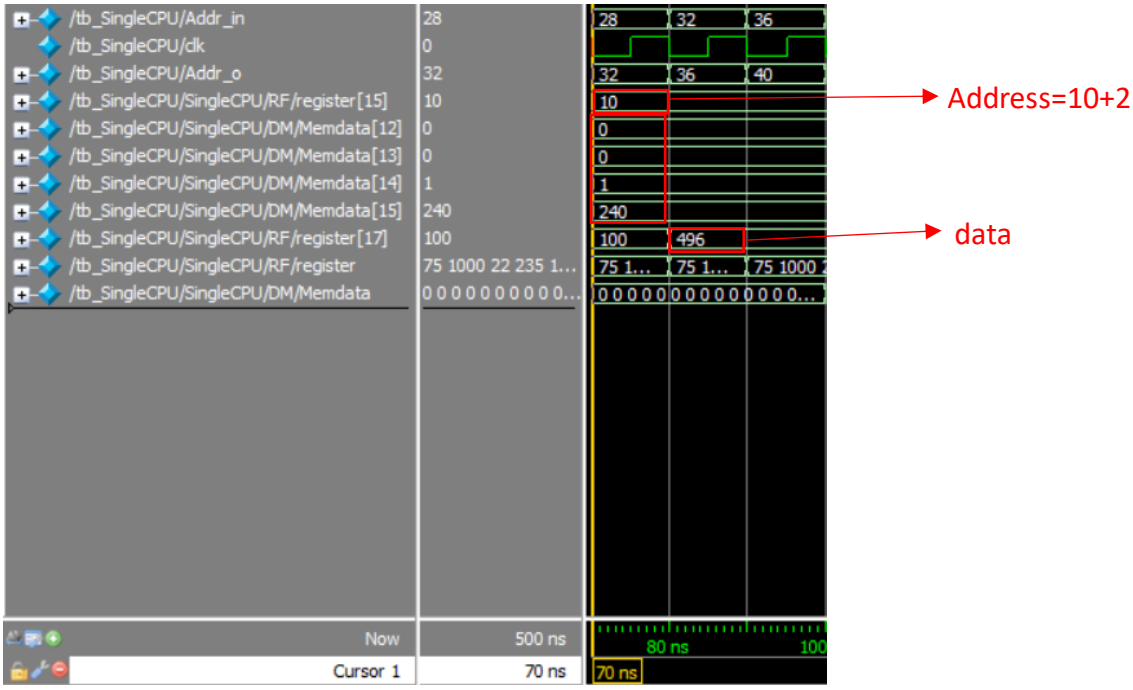
```
5         reg [31:0]Instr[99:0];//Creat 100 Instruction address each is 32-bit
6
7
8   always@(Addr_in)begin
9         Instruction=Instr[Addr_in/4];//the address of instruction is 4times
10  end
11
12  initial begin
13        Instr[0]=32'b010100_01000_01001_01000_00000_010101;//add $t0, $t0, $t1
14        Instr[1]=32'b010100_01010_01100_01001_00000_010110;//sub $t1, $t2, $t4
15        Instr[2]=32'b010100_01101_00000_01100_00001_010111;//srl $t4, $t5, 1
16        Instr[3]=32'b010100_01111_00000_01110_00100_011000;//sll $t6, $t7, 4
17        Instr[4]=32'b010100_01001_01010_01011_00000_011001;//xor $t3, $t1, $t2
18        Instr[5]=32'b010100_01010_01100_01101_00000_011010;//and $t5, $t4, $t2
19
20        Instr[6]=32'b101011_01111_01000_0000000000000010;//sw $t0, 2($t7)
21        Instr[7]=32'b100011_01111_10001_0000000000000010;//lw $s1, 2($t7)
22        Instr[8]=32'b100011_01111_10010_0000000000000100;//lw $s2, 4($t7)
23        Instr[9]=32'b101011_01010_01000_0000000000000010;//sw $t0, 2($t2)
24        Instr[10]=32'b101011_01001_10011_0000000000000100;//sw $s3, 4($t1)
25        Instr[11]=32'b001000_10011_10100_0000000001101111;//addi $s4, $s3, 111
26        Instr[12]=32'b001000_10101_10110_0000000000011011;//addi $s6, $s5, 27
27        Instr[13]=32'b001001_10110_10001_0000000000001001;//subi $s1, $s6, 9
28        Instr[14]=32'b001001_10001_10111_0000000000000101;//subi $s7, $s1, 5
29
30        Instr[15]=32'b000100_11000_11001_0000000000000100;//beq $t8, $t9, 4
31        Instr[16]=32'b000100_01100_11000_0000000000000001;//beq $t4, $t8, 1
32        Instr[17]=32'b000100_10011_10101_0000000000000100;//beq $s3, $s5, 4
33        Instr[18]=32'b000100_01100_01110_0000000000000001;//beq $t4, $t6, 1
```

1.Instr[6]=>{Memdata[12],Memdata[13],Memdata[14],Memdata[15]}

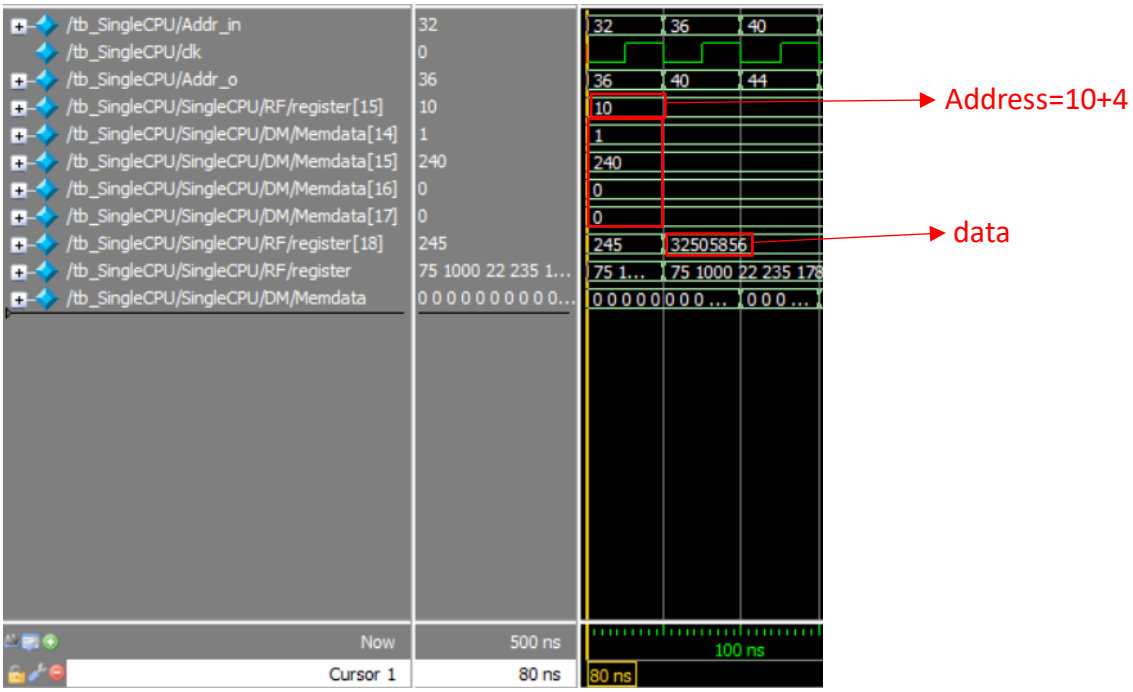
={8'd0,8'd0,8'b00000001,8'b11110000}(496)



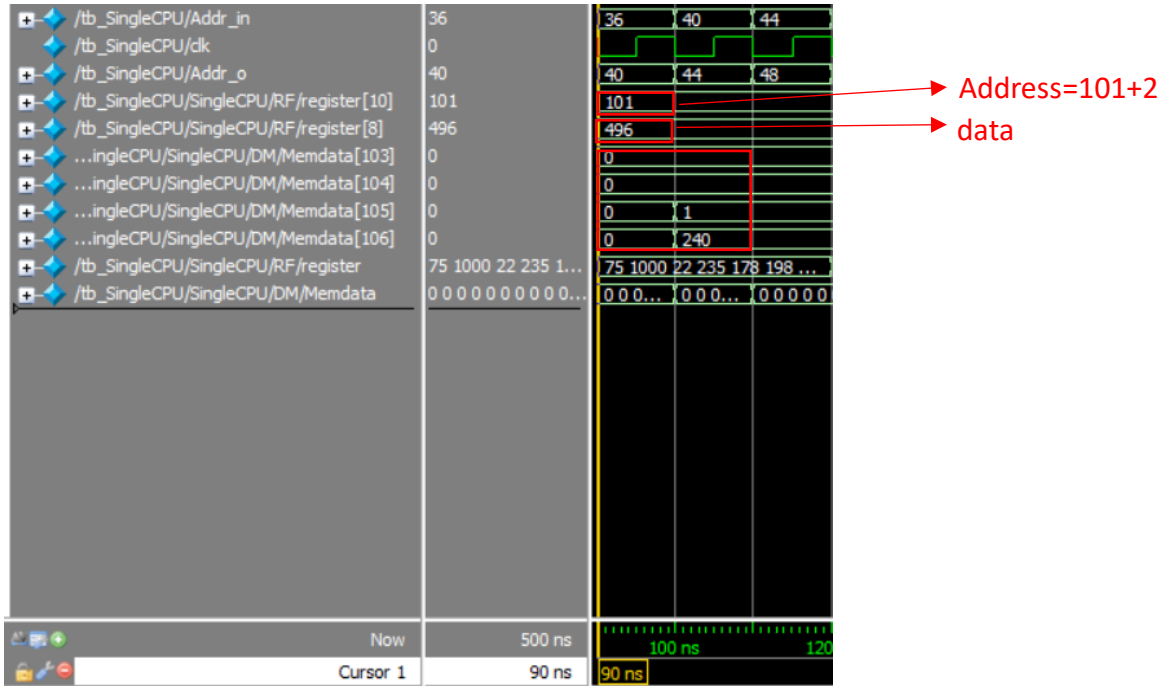
2.Instr[7]=>$s1($17)={Memdata[12],Memdata[13],Memdata[14],Memdata[15]}



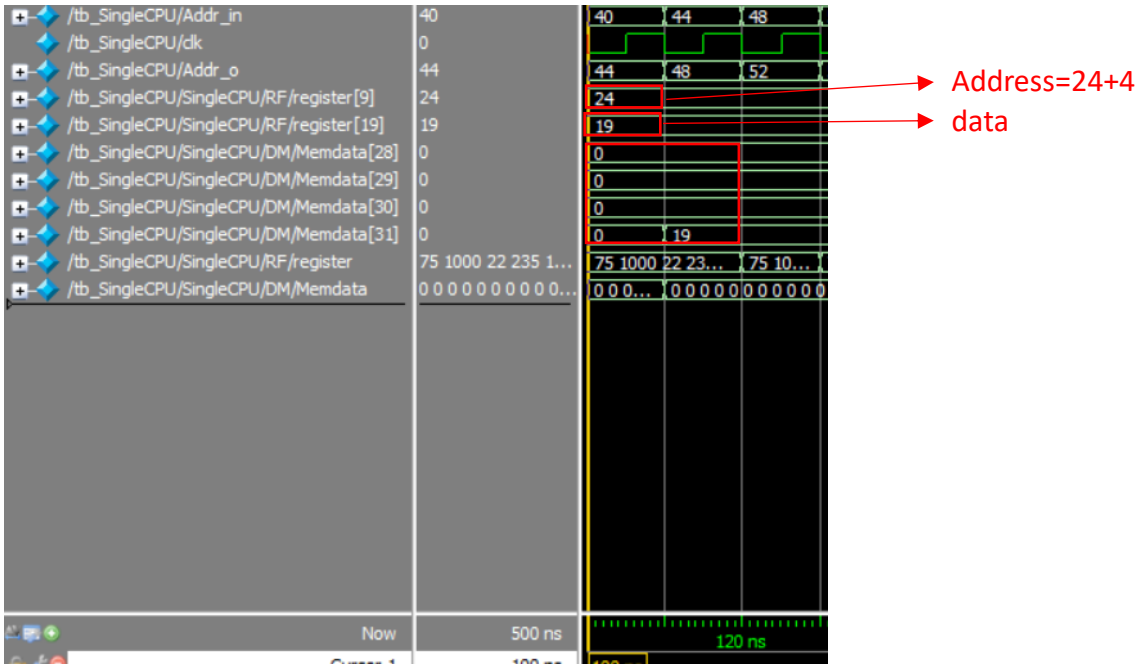
3.Instr[8]=> $s2($18)={Memdata[14],Memdata[15],Memdata[16],Memdata[17]}



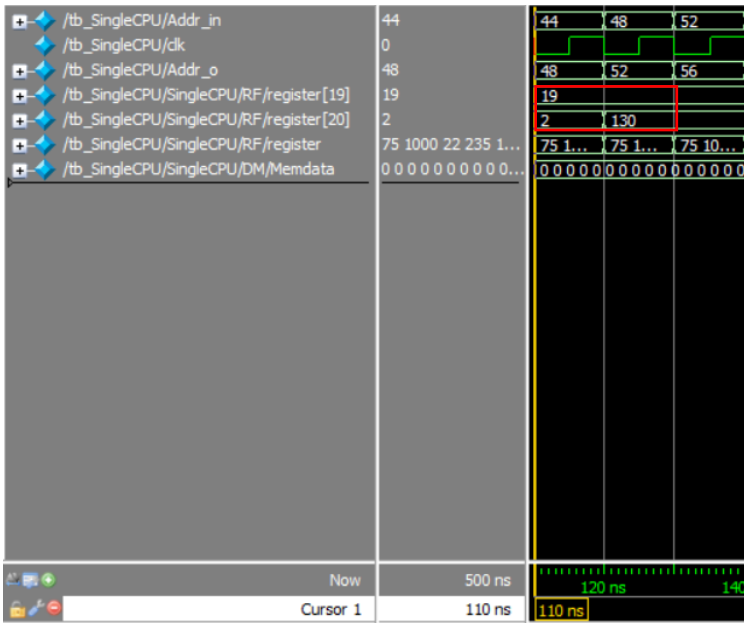
4.Instr[9]=> {Memdata[103],Memdata[104],Memdata[105],Memdata[106]}
={8'd0,8'd0,8'b00000001,8'b11110000}(496)



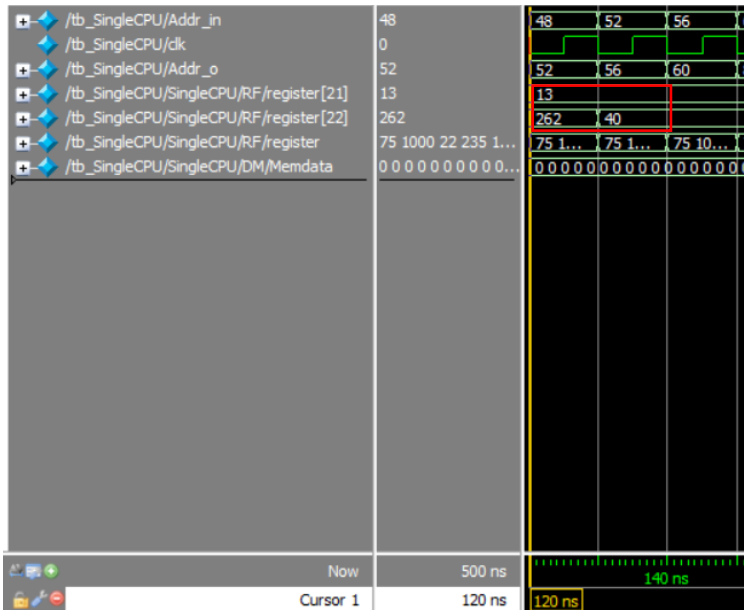
5.Instr[10]=> {Memdata[28],Memdata[29],Memdata[30],Memdata[31]}
={8'd0,8'd0,8'd0,8'b00010011}(19)



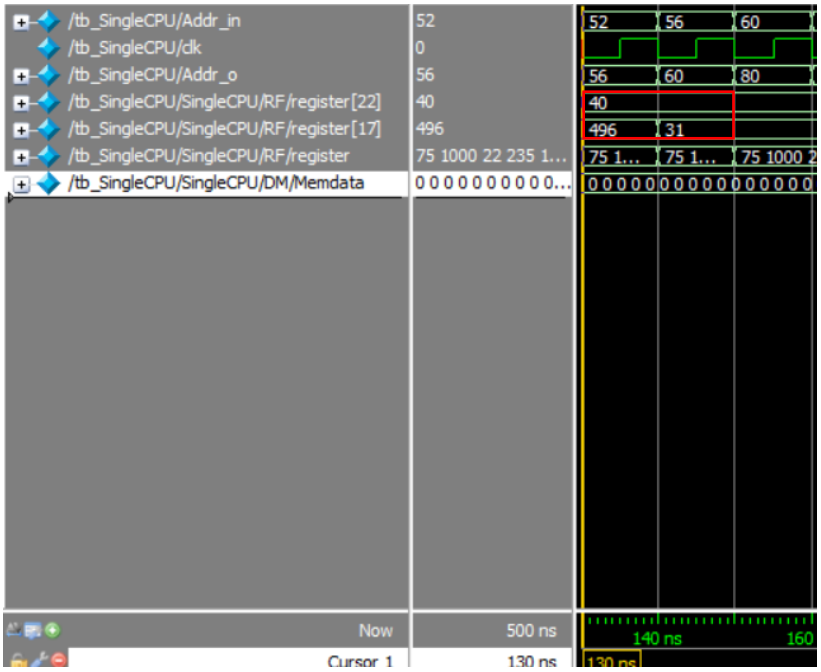
6.Instr[11]=>$s4($20)=130=$s3(19)+111



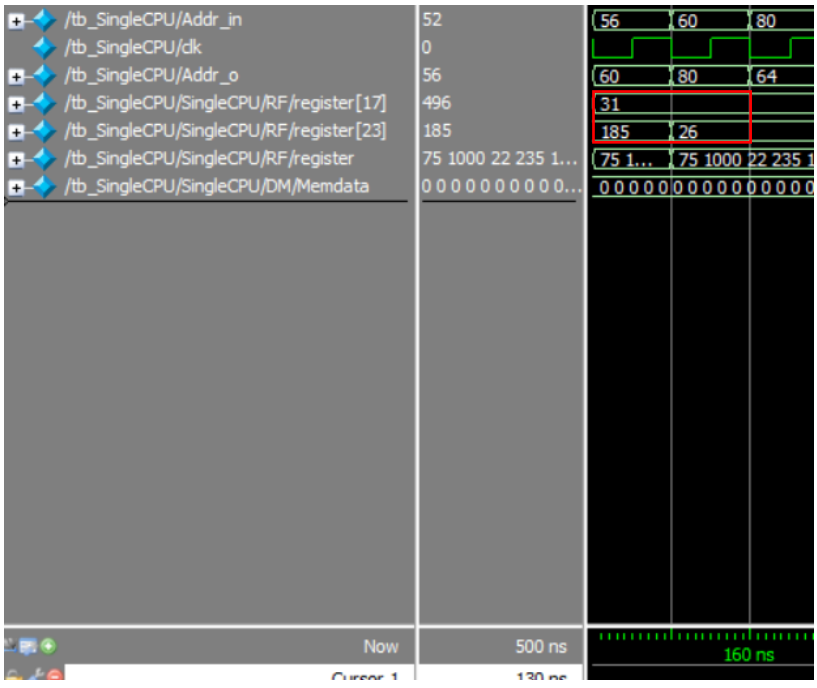
7.Instr[12]=>$s6($22)=40=$s5(13)+27



8.Instr[13]=>$s1($17)=31=$s6(40)-9

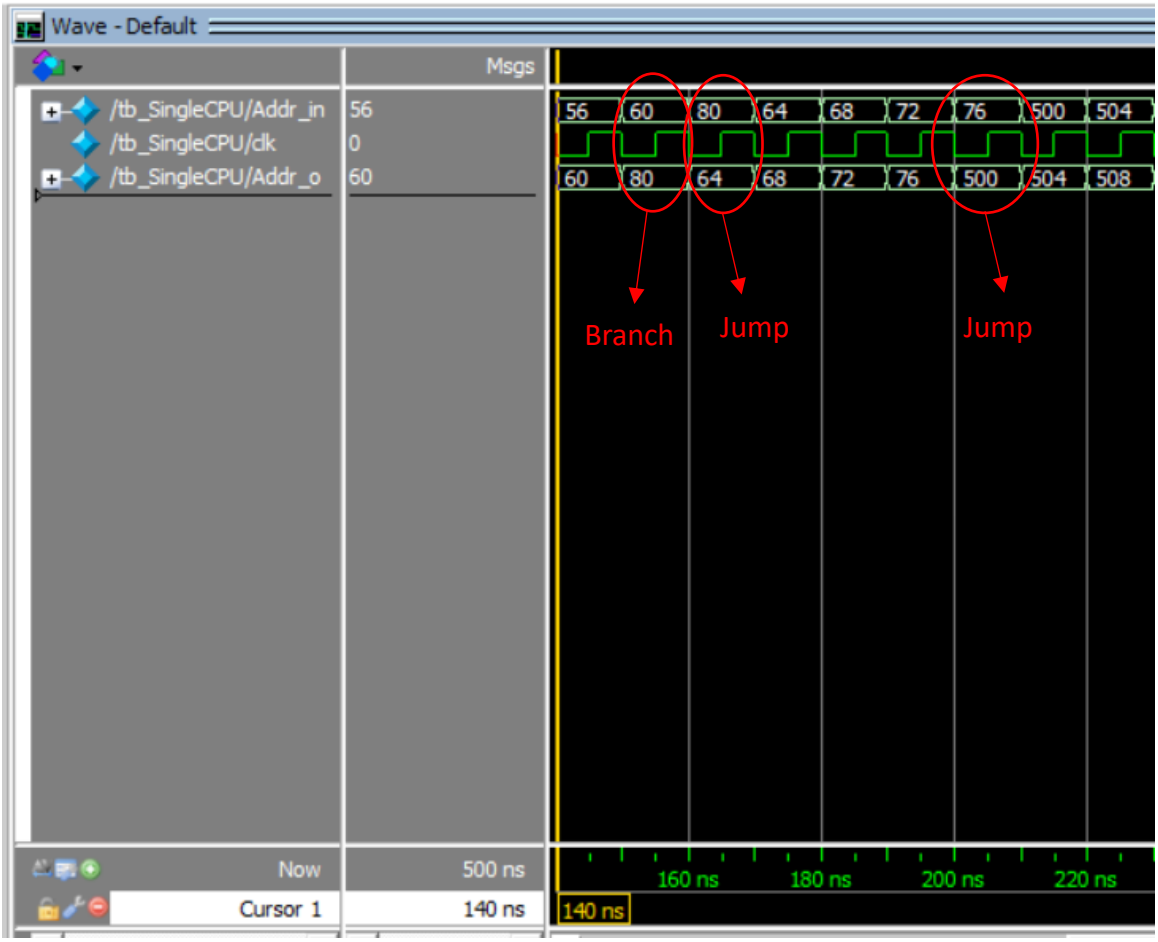


9.Instr[14]=>$s7($23)=26=$s1(31)-5



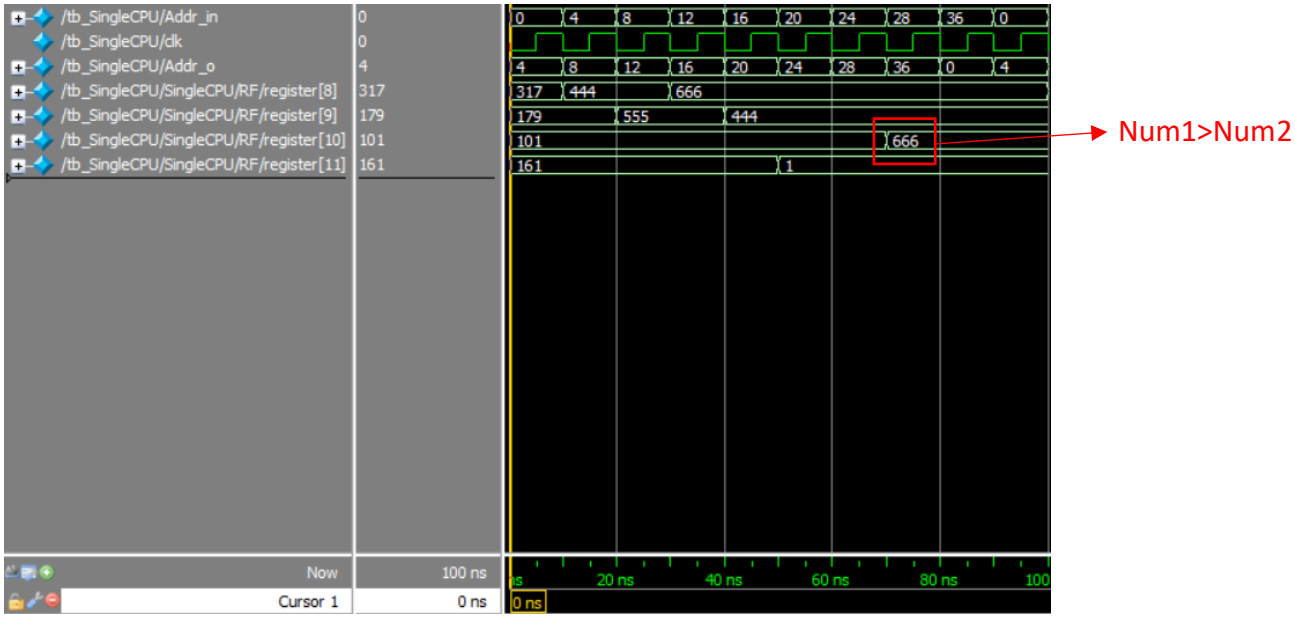
Part3:Beq&Jump

```
8   always@(Addr_in)begin
9           Instruction=Instr[Addr_in/4];//the address of instruction is 4times
10  end
11
12  initial begin
13          Instr[0]=32'b010100_01000_01001_01000_00000_010101;//add $t0, $t0, $t1
14          Instr[1]=32'b010100_01010_01100_01001_00000_010110;//sub $t1, $t2, $t4
15          Instr[2]=32'b010100_01101_00000_01100_00001_010111;//srl $t4, $t5, 1
16          Instr[3]=32'b010100_01111_00000_01110_00100_011000;//sll $t6, $t7, 4
17          Instr[4]=32'b010100_01001_01010_01011_00000_011001;//xor $t3, $t1, $t2
18          Instr[5]=32'b010100_01010_01100_01101_00000_011010;//and $t5, $t4, $t2
19
20          Instr[6]=32'b101011_01111_01000_0000000000000010;//sw $t0, 2($t7)
21          Instr[7]=32'b100011_01111_10001_0000000000000010;//lw $s1, 2($t7)
22          Instr[8]=32'b100011_01111_10010_0000000000000100;//lw $s2, 4($t7)
23          Instr[9]=32'b101011_01010_01000_0000000000000010;//sw $t0, 2($t2)
24          Instr[10]=32'b101011_01001_10011_0000000000000100;//sw $s3, 4($t1)
25          Instr[11]=32'b001000_10011_10100_0000000001101111;//addi $s4, $s3, 111
26          Instr[12]=32'b001000_10101_10110_0000000000011011;//addi $s6, $s5, 27
27          Instr[13]=32'b001001_10110_10001_0000000000001001;//subi $s1, $s6, 9
28          Instr[14]=32'b001001_10001_10111_0000000000000101;//subi $s7, $s1, 5
29
30          Instr[15]=32'b000100_11000_11001_0000000000000100;//beq $t8, $t9, 4
31          Instr[16]=32'b000100_01100_11000_0000000000000001;//beq $t4, $t8, 1
32          Instr[17]=32'b000100_10011_10101_0000000000000100;//beq $s3, $s5, 4
33          Instr[18]=32'b000100_01100_01110_0000000000000001;//beq $t4, $t6, 1
34          Instr[19]=32'b000010_00000000000000000001111101;//j 125
35          Instr[20]=32'b000010_00000000000000000000010000;//j 16
36
```



Part3: Bonus

```
42        Instr[0]=32'b001000_00000_01000_0000000110111100;//addi $t0, $zero, 444
43        Instr[1]=32'b001000_00000_01001_0000001000101011;//addi $t1, $zero, 555
44        Instr[2]=32'b001000_01000_01000_0000000011011110;//addi $t0, $t0, 222
45        Instr[3]=32'b001001_01001_01001_0000000001101111;//subi $t1, $t1, 111
46        Instr[4]=32'b010100_01001_01000_01011_00000_011011;//slt $t3, $t1, $t0
47        Instr[5]=32'b000100_00000_01011_0000000000000010;//beq $zero, $t3, 2
48        Instr[6]=32'b001000_00000_01010_0000001010011010;//addi $t2, $zero, 666
49        Instr[7]=32'b000010_00000000000000000000001001;//j 9
50        Instr[8]=32'b001000_00000_01010_0000001100001001;//addi $t2, $zero, 777
51        Instr[9]=32'b000010_00000000000000000000000000;//j 0
```



令 Number1=$8　Number2=$9　Number3=$10

先將值丟進暫存器

在依照題目做加減

之後到了 if 比較的地方

我是利用指令 slt+beq 來完成這部分

slt 會比較 Source1 有沒有小於 Source2

如果有他會將 Result 設為 1

之後利用 beq 去做比較

就可以完成這部分程式

比較完後會跳回重新執行一次

Module 程式碼

a. Adder

```
module Adder(
        input [31:0] data1,//Number1
        input [31:0] data2,//Number2
        output [31:0] data_o//Result
);

assign data_o=data1+data2;//Function

endmodule
```

這部分非常簡單，因為只是要做一個簡單的加法器，可以使用 Verilog 提供的方式寫會簡單很多(assign 加法)

b.ALU

```
module ALU(
        input [31:0] Source1, //Register1 input
        input [31:0] Source2, //Register2 input
        input [5:0] operation, //Operation code
        input [4:0] shamt, //Shift amount
        output reg [31:0] result, //Result output
        output zero,//Zero flag
        output reg carry //Carry flag
);

assign zero=(result==0)?1:0;//If the result is zero,the zero flag is 1

always@(Source1 or Source2 or operation or shamt)begin
        case(operation[5:0])//Identify function code
                6'd27: {carry,result}<=Source1+Source2;//Function ADD
                6'd28: result<=Source1-Source2;//Function SUB
                6'd29: result<=Source1>>shamt;//Function SRL
                6'd30: result<=Source1<<shamt;//Function SLL
                6'd31: result<=Source1^Source2;//Function XOR
                6'd32: result<=Source1&Source2;//Function AND
                6'd33: result<=(Source1<Source2)?32'd1:32'd0;//Function slt
                default: result<=result;//If function code no match,maintain the result
        endcase
end

initial begin//initial value
        result=32'd0;
        carry=0;
end
```

這部分跟上一次的大同小異，只有多出了 slt 這個運算而已，而這個運算方式也很簡單，只要比較 Source1 有沒有小於 Source2，只要有小於 Result 寫入 1，沒有的話寫入 0，因為等等要跟$zero 做比較，所以只要寫入非 0 的數都可以做比較

c.ALUctrl

```
module ALUctrl(
        input [5:0] funct,//the instruction last 6 bits
        input [2:0] ALUOp,//the signal from controller
        output reg [5:0] operation//the signal to the ALU operation
 );

always@(funct or ALUOp)begin
        case(ALUOp)//identify the signal of controller is what type
                3'b010:begin//R-Type
                        case(funct)
                                6'd21:operation=6'd27;//ADD
                                6'd22:operation=6'd28;//SUB
                                6'd23:operation=6'd29;//SRL
                                6'd24:operation=6'd30;//SLL
                                6'd25:operation=6'd31;//XOR
                                6'd26:operation=6'd32;//AND
                                6'd27:operation=6'd33;//SLT
                        endcase
                end
                3'b000:operation=6'd27;//Type of LW SW addi
                3'b001:operation=6'd28;//Type of subi
                3'b101:operation=6'd28;//type of beq
        endcase
end
initial begin//initial value
        operation=6'd0;
end
endmodule
```

這部分因為要先判斷 Control 給的訊號去做判斷我要讓 ALU 收到甚麼 Operation，而如果輸入 R-type 這個種類的判斷訊號，我就要再判斷 funct 這個訊號代表哪個 operation

d. Control

```
module Control(
        input [5:0] Op,//the instruction first 6bits
        output reg [2:0] ALUOp,//the signal for ALUctrl
        output reg RegDst,//the signal for RD_address
        output reg MemRead,//the signal for memory read or not
        output reg MemtoReg,//the signal for which data(ALU result or memory data) is write data in RF
        output reg MemWrite,//the signal for memory write or not
        output reg ALUSrc,//the signal for comfirm ALU source2
        output reg RegWrite,//the signal for register write or not
        output reg Jump,//the signal for jump or not
        output reg Branch//the signal for branch or not
);
always@(Op)begin
        ALUOp=3'd0;
        RegDst=0;
        MemRead=0;
        MemWrite=0;
        ALUSrc=0;
        RegWrite=0;
        Jump=0;
        Branch=0;
        case(Op)
                6'd2:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b000_0_0_0_0_0_0_1_0;//setting of Jump
                6'd4:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b101_0_0_0_0_0_0_0_1;//setting of branch
                6'd8:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b000_0_0_0_0_0_1_1_0;//setting of addi
                6'd9:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b001_0_0_0_0_0_1_1_0;//setting of subi
                6'd20:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b010_1_0_0_0_0_1_0_0;//setting of R-type
                6'd35:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b000_0_1_1_0_1_1_0_0;//setting of LW
                6'd43:{ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch}=11'b000_0_0_0_1_1_0_0_0;//setting of SW
        endcase
end
```

```
initial begin//initial value
        ALUOp=3'd0;
        RegDst=0;
        MemRead=0;
        MemWrite=0;
        ALUSrc=0;
        RegWrite=0;
        Jump=0;
        Branch=0;
end
endmodule
```

這個部分只要依照 datapath 去控制訊號就可以，因此如果有上課照著處理就好

e.DM

```
module DM(
        input clk,//Clock
        input [31:0] Address,//Memory address
        input [31:0] data,//Memory data by address
        input MemRead,//Control output data
        input MemWrite,//Control input data
        output reg [31:0] DM_data//output data
);
        integer i;//For initial
        reg[7:0]Memdata[127:0];//Creat 128 memory address each address is 8-bit

always@(posedge clk)begin//Read out data
        if(MemRead) DM_data <= {Memdata[Address], Memdata[Address+1], Memdata[Address+2], Memdata[Address+3]};
end
always@(negedge clk)begin//Write in data
        if(MemWrite) {Memdata[Address], Memdata[Address+1], Memdata[Address+2], Memdata[Address+3]} <= data;
end

initial begin//initial value
        DM_data=32'd0;
        for(i=0;i<=127;i=i+1)begin
                Memdata[i] = 8'b0;
        end
end
endmodule
```

這部分是 data memory，難易程度跟等等的 RF 差不多，因為要考慮到寫入讀出的時機還有觸發條件，但其實也不難只要搞清楚觸發時機跟 data 是否準備好這些就可以寫出來

f.IM

```
module IM(
        input [31:0] Addr_in,//the value is the address of running instruction
        output reg [31:0] Instruction//run instruction
);
        integer i;
        reg [31:0]Instr[199:0];//Creat 200 Instruction address each is 32-bit


always@(Addr_in)begin
        Instruction=Instr[Addr_in/4];//the address of instruction is 4times
end

initial begin
        for(i=0;i<200;i=i+1)begin
                Instr[i]=32'd0;
        end

        Instr[0]=32'b010100_01000_01001_01000_00000_010101;//add $t0, $t0, $t1
        Instr[1]=32'b010100_01010_01100_01001_00000_010110;//sub $t1, $t2, $t4
        Instr[2]=32'b010100_01101_00000_01100_00001_010111;//srl $t4, $t5, 1
        Instr[3]=32'b010100_01111_00000_01110_00100_011000;//sll $t6, $t7, 4
        Instr[4]=32'b010100_01001_01010_01011_00000_011001;//xor $t3, $t1, $t2
        Instr[5]=32'b010100_01010_01100_01101_00000_011010;//and $t5, $t4, $t2

        Instr[6]=32'b101011_01111_01000_0000000000000010;//sw $t0, 2($t7)
        Instr[7]=32'b100011_01111_10001_0000000000000010;//lw $s1, 2($t7)
        Instr[8]=32'b100011_01111_10010_0000000000000100;//lw $s2, 4($t7)
        Instr[9]=32'b101011_01010_01000_0000000000000010;//sw $t0, 2($t2)
        Instr[10]=32'b101011_01001_10011_0000000000000100;//sw $s3, 4($t1)
        Instr[11]=32'b001000_10011_10100_0000000001101111;//addi $s4, $s3, 111
        Instr[12]=32'b001000_10101_10110_0000000000011011;//addi $s6, $s5, 27
        Instr[13]=32'b001001_10110_10001_0000000000001001;//subi $s1, $s6, 9
        Instr[14]=32'b001001_10001_10111_0000000000000101;//subi $s7, $s1, 5
```

這部分就是將 Instruction 寫入的地方，因為是利用 Addr_in 去跑 Instruction，所以要除以 4 才能表示是要執行第幾個 Instruction

g.MUX5b

```
module MUX5b(
        input [4:0] data1,//value1
        input [4:0] data2,//value2
        input select,//control
        output [4:0] data_o//output
);
assign data_o=(select)?data1:data2;//if select is 1,output is data1.if select is 0,output is data2.


endmodule
```

5-bit 的多工器只要利用 assign 的方式然後去判斷選擇線調整輸出即可

h.MUX32b

```
module MUX32b(
        input [31:0] data1,//value1
        input [31:0] data2,//value2
        input select,//control
        output [31:0] data_o//output
 );
assign  data_o=(select)? data1:data2;//if select is 1,output is data1.if select is 0,output is data2.

endmodule
```

32-bit 的多工器也是只要利用 assign 的方式然後去判斷選擇線調整輸出即可

i.RF

```
module RF(
        input clk,//Clock
        input RegWrite,//The signal of write in register or not
        input [4:0] RS_Address,//The address of register1
        input [4:0] RT_Address,//The address of register2
        input [4:0] RD_Address,//The address of register write in
        output reg [31:0] RSdata,//The data of register1
        output reg [31:0] RTdata,//The data of register2
        input [31:0] RDdata//The data of register write in
);

reg [31:0] register[31:0];//Creat 32 registers

always@(RS_Address or RT_Address)begin
        RSdata<=register[RS_Address];//Read the data of register1 at the address1
        RTdata<=register[RT_Address];//Read the data of register2 at the address2
end
always@(negedge clk)begin
        if(RegWrite) register[RD_Address]=RDdata;//Write data in the register at the address
end

initial begin//initial RF
        register [0]= 32'd0;
        register [1]= 32'd11;
        register [2]= 32'd370;
        register [3]= 32'd183;
        register [4]= 32'd91;
```

Register File 跟剛剛的 Data Memory 一樣困難，因為要考慮寫入讀出的問題，但其實跟 DM 一樣只要仔細想好 latch 的問題就不會有太大的困難點

j.SE

```
module SE(
        input [15:0] data_i,//input 16-bit
        output [31:0] data_o//output 32-bit
);

assign  data_o={16'd0,data_i};

endmodule
```

SE 的話就只是讓我的 immediate 值從 16-bit 擴充到 32-bit，所以只要利用 assign 的方式在[31:16]這幾個位置補 0 就可以了

k. SingleCPU

```
module SingleCPU(
        input [31:0] Addr_in,//run instruction address
        input clk,//clock
        output [31:0] Addr_o//next instruction address
);
wire [31:0]NextPC;//run instruction address+4
wire [31:0]BranchPC;//branch address
wire [31:0]NBPC;//BranchPC or NextPC
wire [31:0]JumpPC;//jump address
wire [31:0]RSdata;//the data of Register1
wire [31:0]RTdata;//the data of Register2
wire [31:0]RDdata;//input data to RF
wire [31:0]ALUSrc2;//inpur ALU source2
wire [31:0]ALUResult;//Result of ALU
wire [31:0]Instruction;//Instruction
wire [31:0]DM_data;//Memory output data
wire [5:0]operation;//ALU operation
wire [4:0]WriteRegister;//Register input data
wire [31:0]immediate;//SE output
wire [2:0]ALUOp;//comfirm ALU operation
wire zero;//zero flag
wire carry;//carry flag
wire RegDst;//the signal for RD_address
wire MemRead;//the signal for memory read or not
wire MemtoReg;//the signal for which data(ALU result or memory data) is write data in RF
wire MemWrite;//the signal for memory write or not
wire ALUSrc;//the signal for comfirm ALU source2
wire RegWrite;//the signal for register write or not
wire Jump;//the signal for jump or not
wire Branch;//the signal for branch or not

assign JumpPC={NextPC[31:28],Instruction[25:0],2'd0};//Jump address 32-bit
```

```
Adder add1(32'd4,Addr_in,NextPC);
IM IM(Addr_in,Instruction);
MUX5b MUX1(Instruction[15:11],Instruction[20:16],RegDst,WriteRegister);
SE SE(Instruction[15:0],immediate);
Control Ctrl(Instruction[31:26],ALUOp,RegDst,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump,Branch);
RF RF(clk,RegWrite,Instruction[25:21],Instruction[20:16],WriteRegister,RSdata,RTdata,RDdata);
Adder add2(immediate<<2,NextPC,BranchPC);
MUX32b MUX2(immediate,RTdata,ALUSrc,ALUSrc2);
ALUctrl ALUctrl(Instruction[5:0],ALUOp,operation);
ALU ALU(RSdata,ALUSrc2,operation,Instruction[10:6],ALUResult,zero,carry);
MUX32b MUX3(BranchPC,NextPC,Branch&zero,NBPC);
MUX32b MUX4(JumpPC,NBPC,Jump,Addr_o);
DM DM(clk,ALUResult,RTdata,MemRead,MemWrite,DM_data);
MUX32b MUX5(DM_data,ALUResult,MemtoReg,RDdata);

endmodule
```

最後 single cycle processor 就做出來了，只要按造接線圖將以上的 Module 接起來，還有一些線路的定義做好，這部分就只剩接線要接對，其他沒甚麼大問題

l. tb_SingleCPU

```
`timescale 1ns/1ns
module tb_SingleCPU();

reg [31:0] Addr_in;
reg clk;
wire [31:0] Addr_o;

SingleCPU SingleCPU(Addr_in,clk,Addr_o);

initial begin
        clk=0;
        Addr_in=32'd0;
#500    $finish;
end
always begin//Creat a clock which the period is 10ns and the duty cycle is 50%
#5      clk=~clk;
end
always begin
#10 Addr_in=Addr_o;
end
endmodule
```

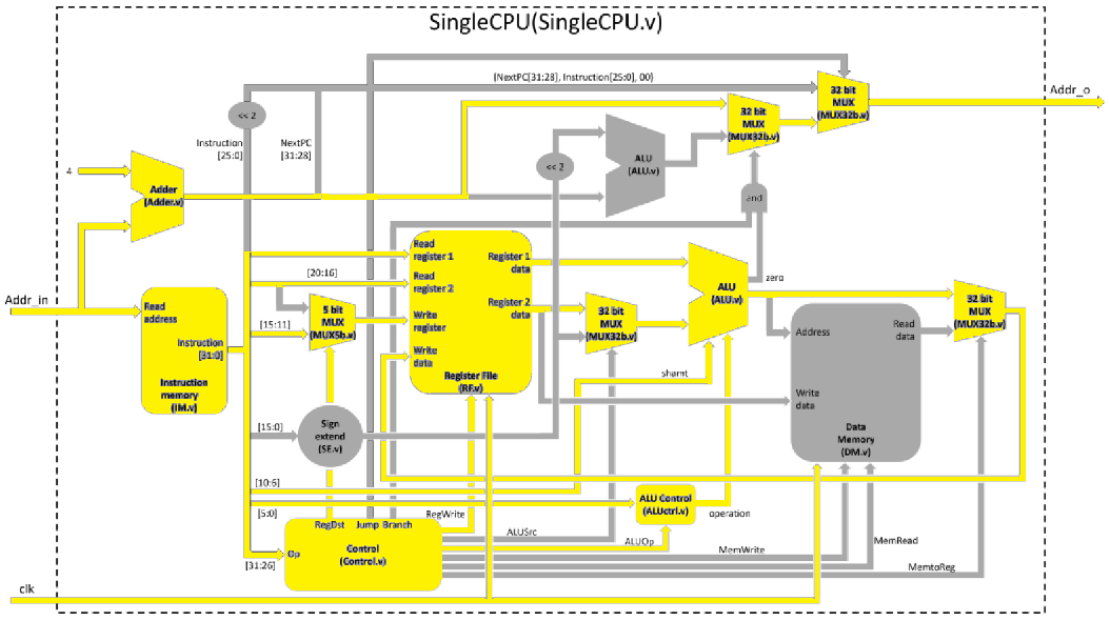
這是最後測試的 testbench，而寫法也很簡單，先將 input output 定義好，接下來去製造出 clock，週期是 10ns，最後再讓我的 Addr_in 從 0 開始，然後每 10ns 從 Addr_o 讀值進來，這樣 testbench 就完成了

Data Path

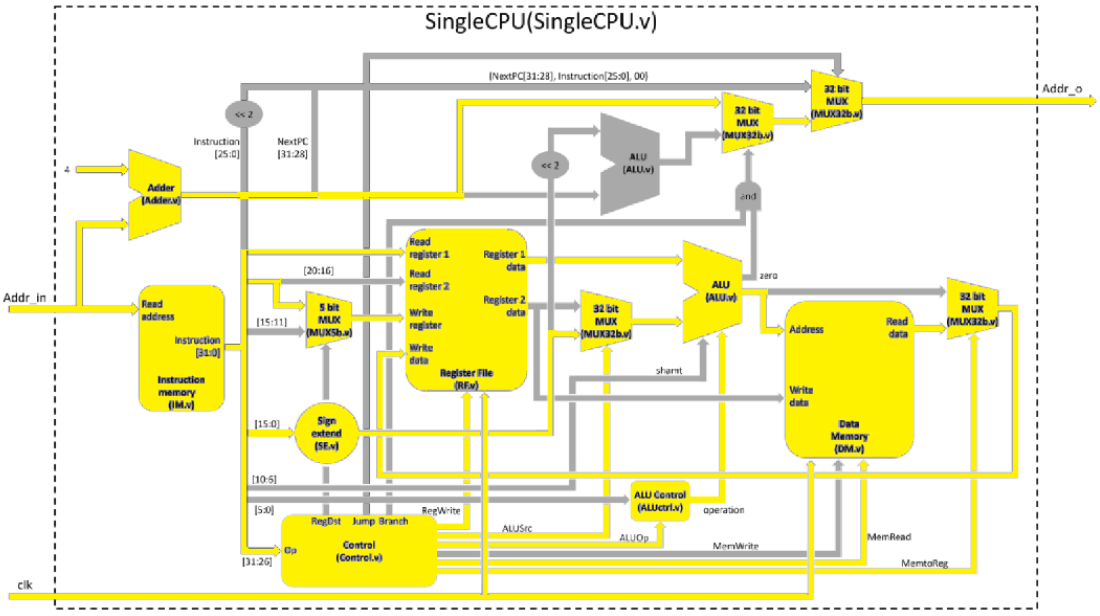
Control 的 Output 訊號若為 1 會標記有顏色

若為 0 則不標記顏色

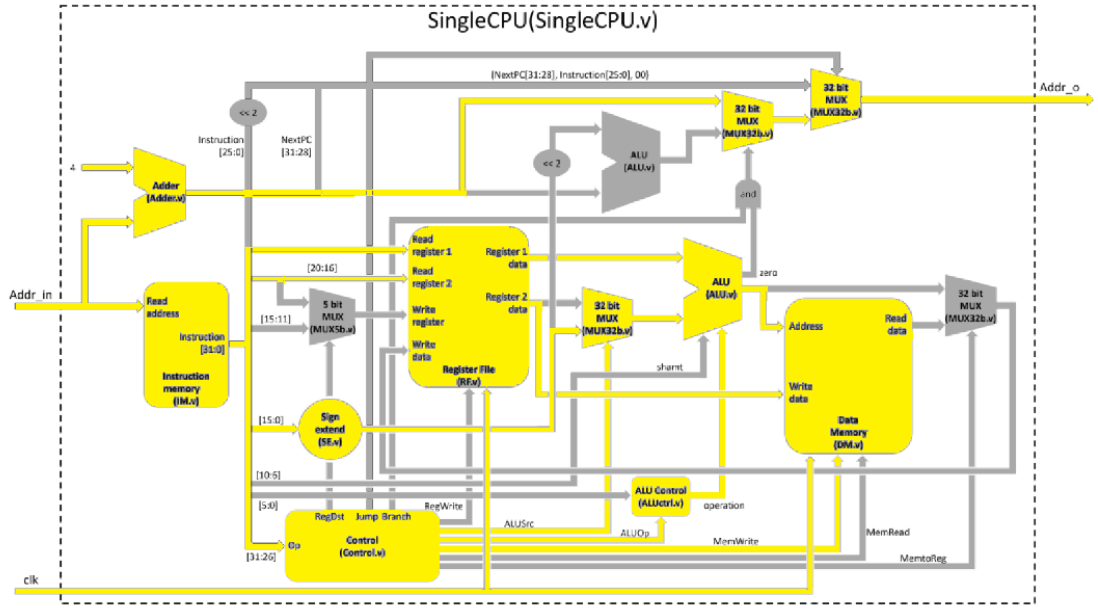
R-Type



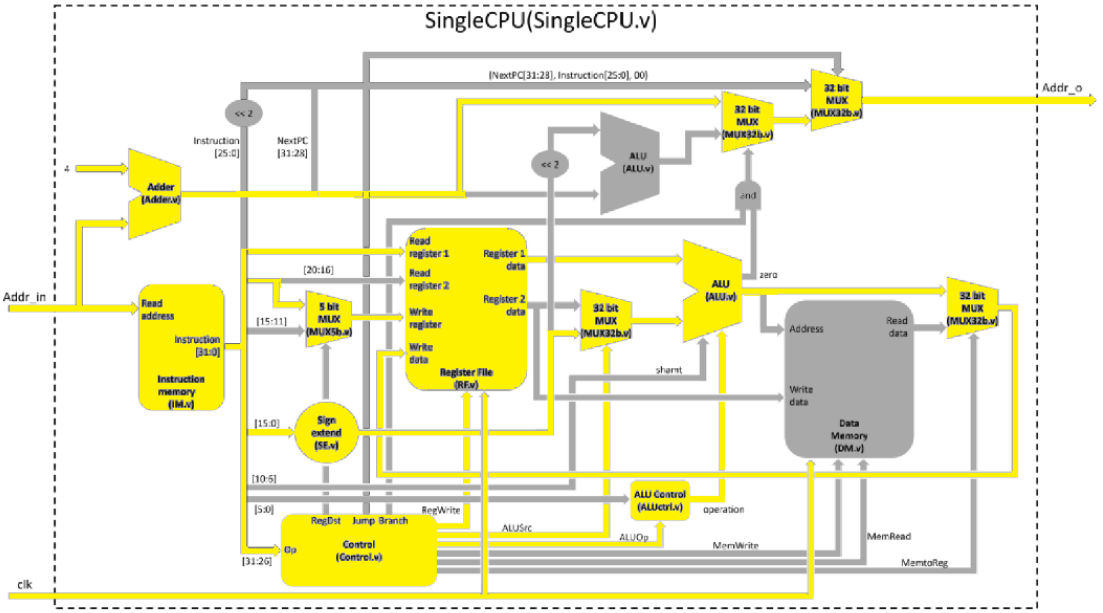
LW



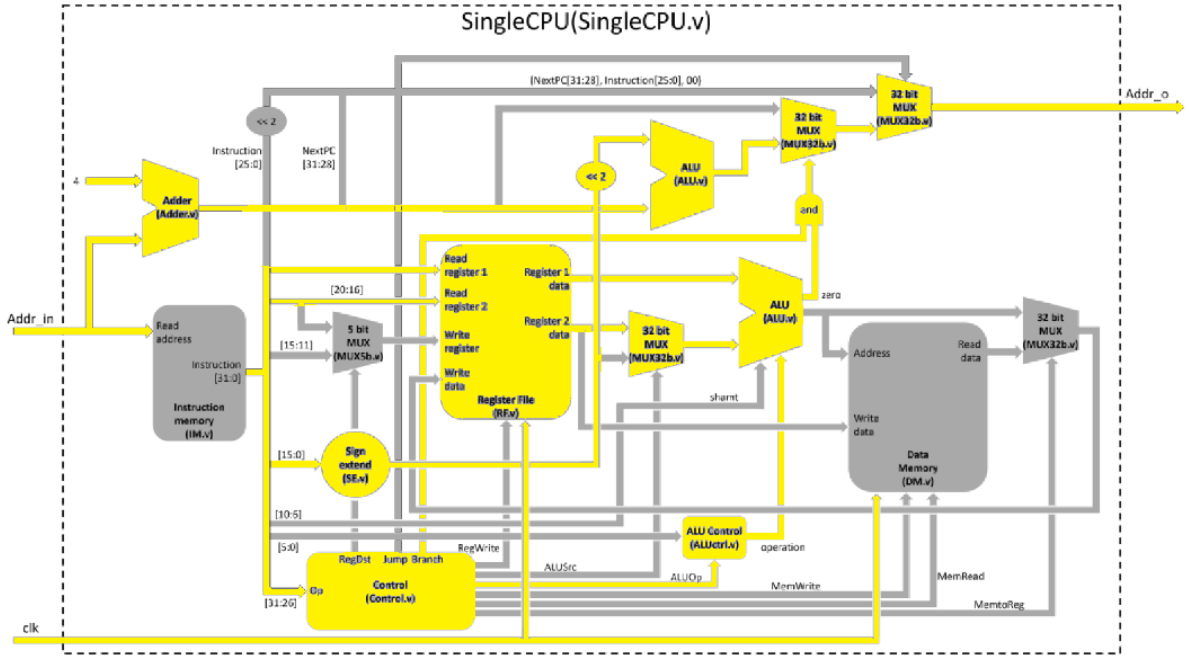
SW


SingleCPU(SingleCPU.v)
{NextPC[31:28], Instruction[25:0], 00}
32 bit MUX (MUX32b.v)
Addr_o
<< 2
Instruction [25:0]
NextPC [31:28]
4
Adder (Adder.v)
ALU (ALU.v)
and
Addr_in
Read address
Instruction [31:0]
Instruction memory (IM.v)
[20:16]
[15:11]
5 bit MUX (MUX5b.v)
Read register 1
Read register 2
Write register
Write data
Register 1 data
Register 2 data
Register File (RF.v)
zero
shamt
Address
Read data
Write data
Data Memory (DM.v)
[15:0]
Sign extend (SE.v)
[10:6]
[5:0]
[31:26]
RegDst Jump Branch
Op
Control (Control.v)
RegWrite
ALUSrc
ALUOp
ALU Control (ALUctrl.v)
operation
MemWrite
MemRead
MemtoReg
clk

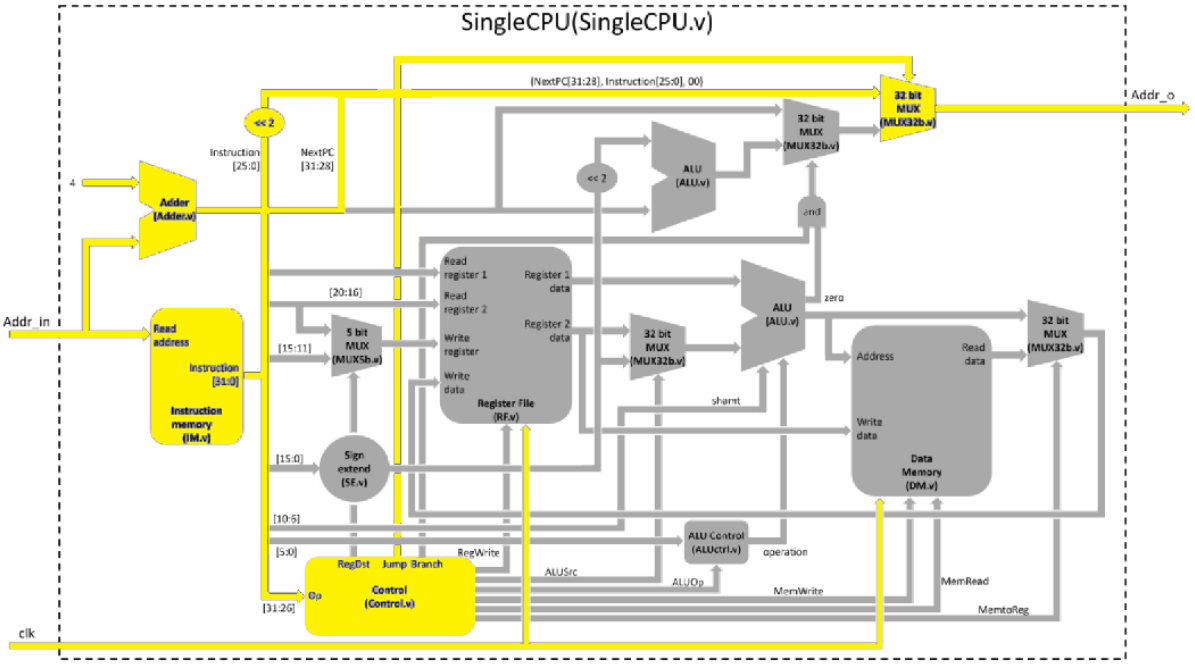
addi/subi


SingleCPU(SingleCPU.v)
{NextPC[31:28], Instruction[25:0], 00}
32 bit MUX (MUX32b.v)
Addr_o
<< 2
Instruction [25:0]
NextPC [31:28]
4
Adder (Adder.v)
ALU (ALU.v)
and
Addr_in
Read address
Instruction [31:0]
Instruction memory (IM.v)
[20:16]
[15:11]
5 bit MUX (MUX5b.v)
Read register 1
Read register 2
Write register
Write data
Register 1 data
Register 2 data
Register File (RF.v)
zero
shamt
Address
Read data
Write data
Data Memory (DM.v)
[15:0]
Sign extend (SE.v)
[10:6]
[5:0]
[31:26]
RegDst Jump Branch
Op
Control (Control.v)
RegWrite
ALUSrc
ALUOp
ALU Control (ALUctrl.v)
operation
MemWrite
MemRead
MemtoReg
clk

branch


SingleCPU(SingleCPU.v)
{NextPC[31:28], Instruction[25:0], 00}
Addr_o
32 bit MUX (MUX32b.v)
<< 2
Instruction [25:0]
NextPC [31:28]
4
Adder (Adder.v)
ALU (ALU.v)
and
Addr_in
Read address
Instruction [31:0]
Instruction memory (IM.v)
[20:16]
[15:11]
5 bit MUX (MUX5b.v)
Read register 1
Read register 2
Write register
Write data
Register 1 data
Register 2 data
Register File (RF.v)
zero
sharmt
Address
Read data
Write data
Data Memory (DM.v)
[15:0]
Sign extend (SE.v)
[10:6]
[5:0]
ALU Control (ALUctrl.v)
operation
RegDst
Jump
Branch
RegWrite
Op
Control (Control.v)
[31:26]
ALUSrc
ALUOp
MemWrite
MemRead
MemtoReg
clk

jump


SingleCPU(SingleCPU.v)
{NextPC[31:28], Instruction[25:0], 00}
Addr_o
32 bit MUX (MUX32b.v)
<< 2
Instruction [25:0]
NextPC [31:28]
4
Adder (Adder.v)
ALU (ALU.v)
and
Addr_in
Read address
Instruction [31:0]
Instruction memory (IM.v)
[20:16]
[15:11]
5 bit MUX (MUX5b.v)
Read register 1
Read register 2
Write register
Write data
Register 1 data
Register 2 data
Register File (RF.v)
zero
sharmt
Address
Read data
Write data
Data Memory (DM.v)
[15:0]
Sign extend (SE.v)
[10:6]
[5:0]
ALU Control (ALUctrl.v)
operation
RegDst
Jump
Branch
RegWrite
Op
Control (Control.v)
[31:26]
ALUSrc
ALUOp
MemWrite
MemRead
MemtoReg
clk