计算机组成原理实验五——单周期 MIPS_CPU 设计

一、**实验目的**：设计一个单周期 CPU，完成一个程序代码的执行

二、**实验内容**：设计 CPU，完成以下程序代码的执行，其功能是起始数为 3 和 3 的斐波拉契数列的计算，只计算 20 个数。

```
.data
fibs: .word  0 : 20     # "array" of 20 words to contain fib values
size: .word 20          # size of "array"
temp: .word 3 3
    .text
    la   $t0, fibs      # load address of array
    la   $t5, size      # load address of size variable
    lw   $t5, 0($t5)    # load array size
    la   $t3, temp                  # load
    lw   $t3, 0($t3)
    la   $t4, temp
    lw   $t4, 4($t4)

    sw   $t3, 0($t0)    # F[0] = $t3
    sw   $t4, 4($t0)    # F[1] = $t4
    addi $t1, $t5, -2   # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)    # Get value from array F[n]
    lw   $t4, 4($t0)    # Get value from array F[n+1]
    add  $t2, $t3, $t4   # $t2 = F[n] + F[n+1]
    sw   $t2, 8($t0)    # Store F[n+2] = F[n] + F[n+1] in array
    addi $t0, $t0, 4    # increment address of Fib. number source
    addi $t1, $t1, -1   # decrement loop counter
    bgtz $t1, loop      # repeat if not finished yet.
out:
            j out
```

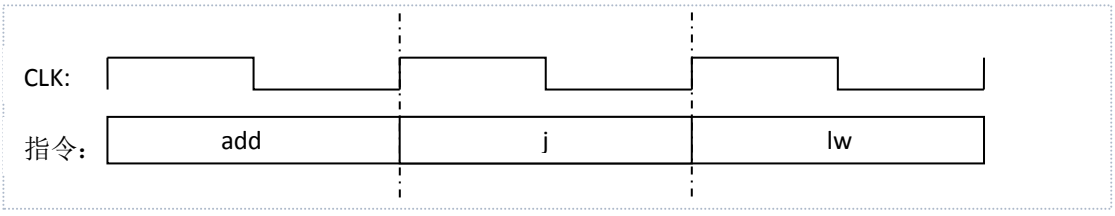
说明：需要例化两个 mem，一个存放指令，一个存放数据,首地址均为 0，两个 ram 和一个 regfile 均要求是异步读，同步写。

基本思路是依据给定过的指令集（6 条），设计核心的控制信号。依据前面给定的数据通路和控制单元信号进行设计。

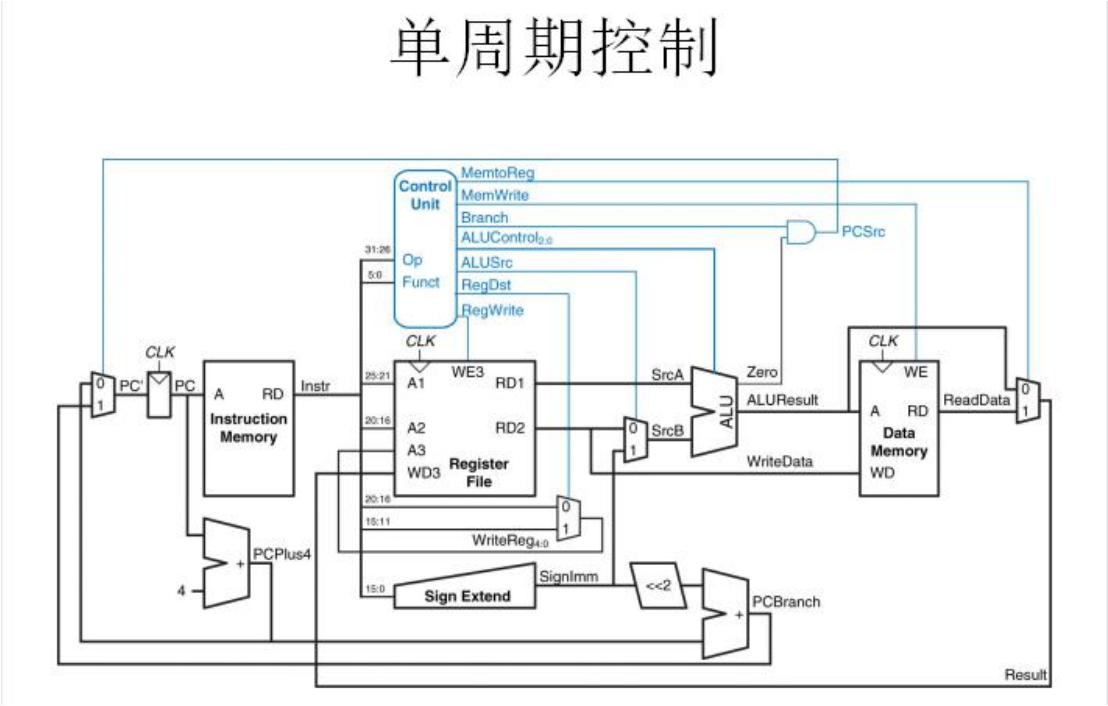
三、**实验分析**：

1.<u>实验原理</u>

单周期 CPU：单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令，在这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。



单周期 CPU 的顶层结构控制为：



其中，控制单元(Ctrl Unit)定义如下：

（1）JUMP：为 1 时，选择跳转目标地址；为 0 时，选择由 Branch 选出的地址；

（2）MemToReg：为 1 时，选择存储器数据；为 0 时，选择 ALU 输出的数据；

（3）Branch：为 1 时，选择转移目标地址；为 0 时，选择 PC +4（图中的 NextPC）；

（4）MemWrite：为 1 时写入存储器。存储器地址由 ALU 的输出决定，写入数据为寄存器 rt 的内容；

（5）ALUOP：ALU 控制码；

（6）ALUSrc：ALU 操作数 B 的选择，为 1 时，选择扩展的立即数；

为 0 时，选择寄存器数据；

（7）RegWrite：为 1 时写入寄存器堆，目的寄存器号是由 REGDES 选出的 rt 或 rd，写入数据是由 MemToReg 选出的存储器数据或 ALU 的输出结果；

（8）RegDst：目的地址，为 1 时，选择 rd；为 0 时，选择 rt。

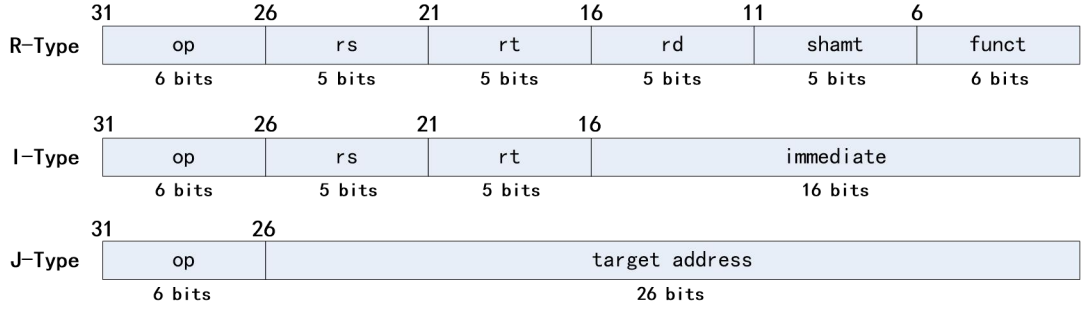
2.MIPS 指令集

本次实验共涉及三种类型的 MIPS 指令，分别为 R 型、I 型和 J 型，三种类型的 MIPS 指令格式定义如下：

R（register）类型的指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆；

I（imm）类型的指令使用一个 16 位的立即数作为一个源操作数；

J（jump）类型的指令使用一个 26 位立即数作为跳转的目标地址。



3.本次实验中的六条指令

_ op

（1）add 000000 rs rt rd shamt funct=100000

rs+rt->rd

（2）addi 001000 rs rt immediate

rs+(signextend)immediate -> rt

（3）sw　　101011　rs　rt　immediate

rt -> memory[rs + (signextend)immediate]

（4）lw　　100011　rs　rt　immediate

memory[rs + (signextend)immediate] -> rt

（5）bgtz　000111　rs　rt　offset

rs 中的值大于 0 时跳转，跳转值为 offset

（6）j　　000010　target address

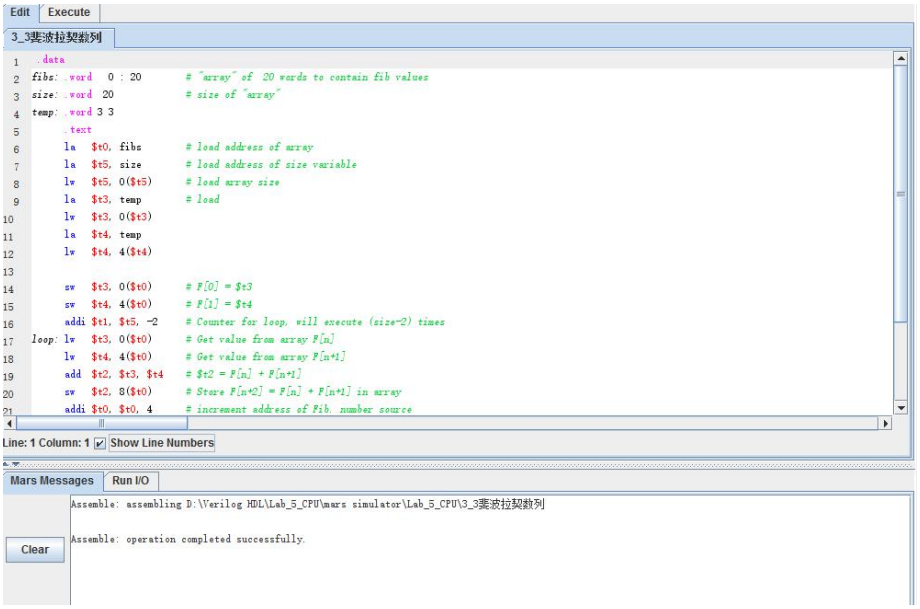
(PC+4)[31:28],address,0,0 -> PC　(26 位扩展为 32 位)

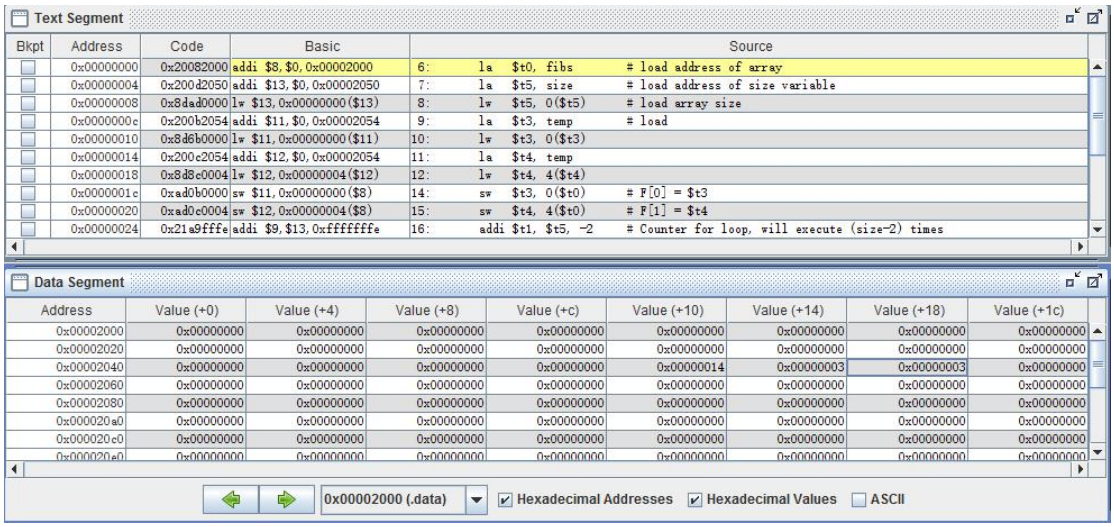
4.实验过程分析

（1）对之前实验的 ALU 进行修改，完成 ALUDecoder 的编写

指令	add	addi	lw	sw	bgtz	j
ALUop[1:0]	10	00	00	00	11	00

（2）a.下载好 Java 开发环境，打开 Mars4_4，将实验所给程序写入进行汇编，导出分别写入相应的 coe 文件中，其中指令部分地址作修改，使访问的起始地址为 0；





b.建立 ram/rom，选择 distributed memory generator,使用相应的 coe 文件进行初始化；

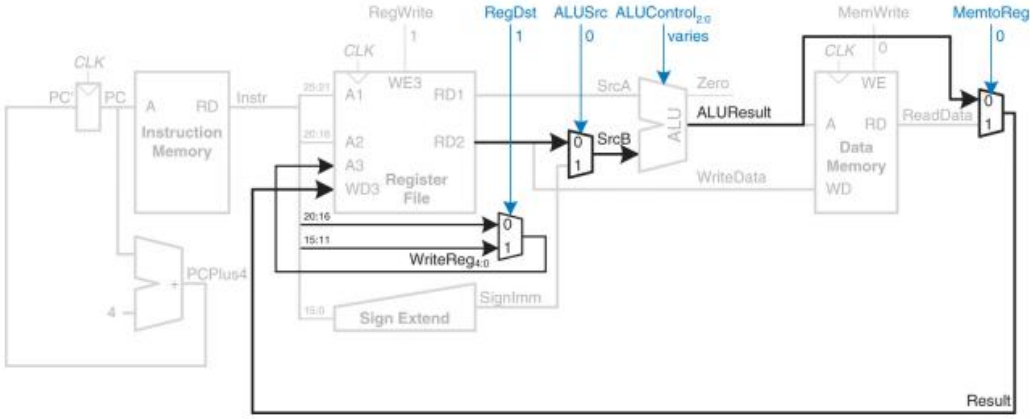
c.完成 pc、选择器 MUX（分为 5bit 和 32bit）、译码器（已完成 ALUDecoder,还需完成 Control Unit 部分）、signextend 的模块编写

d.根据数据通路进行连线

```
53  pc pc(clk,rst,next_pc,curr_pc);
54  InstructionMemory InstructionMemory(curr_pc[6:2],IR);
55  MainDecoder MainDecoder(IR[31:26],MemtoReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump,ALUOp);
56  ALUDecoder ALUDecoder(IR[5:0],ALUOp,ALUControl);
57  regfile regfile(clk,rst,IR[25:21],IR[20:16],WriteReg,Result,RegWrite,data1,data2);
58  Mux5b Mux1(IR[15:11],IR[20:16],RegDst,WriteReg);          //5bit
59  SignExtend SignExtend(IR[15:0],signimm);
60  Mux Mux2(signimm,data2,ALUSrc,SrcB);                    //32bit
61  alu ALU(data1,SrcB,ALUControl,Zero,ALUResult);
62  DataMemory DataMemory(ALUResult[6:2],data2,clk,MemWrite,ReadData);
63  Mux Mux3(ReadData,ALUResult,MemtoReg,Result);           //32bit
64
65  assign PCPlus4=curr_pc+4;
66  assign PCBranch=(signimm<<2)+PCPlus4;
67  assign PCSrc=Branch&Zero;
68  Mux Mux4(PCBranch,PCPlus4,PCSrc,nonjump);               //32bit
69  assign Jumpaddr[27:0]=IR[25:0]<<2;
70  assign Jumpaddr[31:28]=PCPlus4[31:28];
71
72  Mux Mux5(Jumpaddr,nonjump,Jump,next_pc);
```

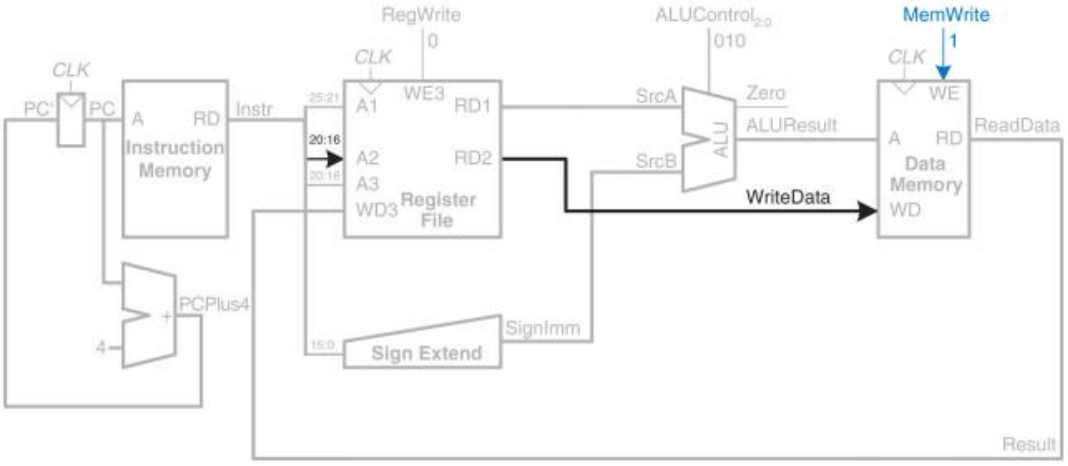
✧ 部分指令执行说明：

（1）add



（2）lw

（3）sw



（4）bgtz

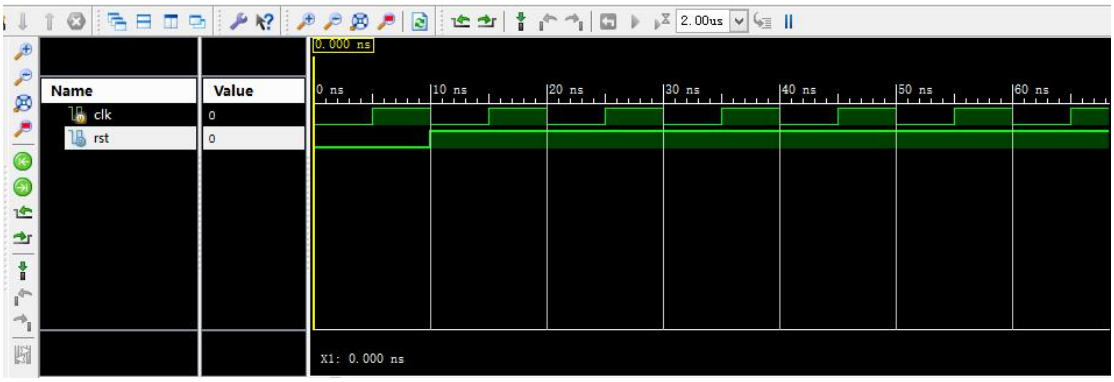
执行 bgtz 指令时，Branch=1，ALUresult=0，从而 Zero=1，PCSrc=Branch&Zero=1,所以 MUX4 选择器结果 nonjump=PCBranch，若此时不执行无条件跳转指令，则进行跳转

e.建立测试文件，对时钟进行设置，复位值由 0 变为 1，进行仿真检测

四、**实验结果：**

成功将代码编译并存入 rom/ram，成功完成六个 MIPS 指令的编

写，仿真正确，数据 ram 内容正确。



指令：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x1F	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	08000011	1D20FFF9
0xF	2129FFFF	21080004	AD0A0008	016C5020	8D0C0004	8D0B0000	21A9FFFE	AD0C0004	AD0B0000	8D8C0004	200C0054	8D6B0000	200B0054	8DAD0000	200D0050	20080000

仿真后 ram 里的结果：

	0	1	2	3	4	5	6	7
0x1F	0	0	0	0	0	0	0	0
0x17	0	3	3	20	20295	12543	7752	4791
0xF	2961	1830	1131	699	432	267	165	102
0x7	63	39	24	15	9	6	3	3

五、附录：

1.各模块代码

（1）alu

```
module alu(
  input [31:0]SrcA,
  input [31:0]SrcB,
  input [2:0]ALUControl,
  output reg Zero,
  output reg [31:0]ALUResult
    );
```

```
parameter     A_AND = 3'b000;
parameter     A_OR   = 3'b001;
parameter     A_ADD = 3'b010;
parameter     A_SUB = 3'b110;
parameter     A_SOLT = 3'b111;

always@(*)
begin
   case(ALUControl)
        A_AND:     ALUResult<=SrcA&SrcB;
        A_OR:      ALUResult<=SrcA|SrcB;
        A_ADD:     ALUResult<=SrcA+SrcB;
        A_SUB:     ALUResult<=SrcA-SrcB;
        A_SOLT:
        begin
                   if(SrcA<=SrcB)
                        ALUResult<=1;
                   else ALUResult<=0;
        end
        default: ALUResult<=0;
   endcase
   if(ALUResult==0)
```

```
            Zero<=1;
        else Zero<=0;
      end


      endmodule
      （2）ALUDecoder
      module ALUDecoder(//ALUcontrol
    input [5:0]Funct,
    input [1:0]ALUOp,
    output reg [2:0]ALUControl
      );

always@(*)
begin
    case(ALUOp)
        2'b00:  ALUControl<=3'b010;//LW、SW、addi、j     add
        2'b01:  ALUControl<=3'b110;//beg      sub
        2'b10://RType
        begin
            case(Funct)
                6'b100000: ALUControl<=3'b010;//add rs+rt->rd
                6'b100010: ALUControl<=3'b110;//sub rs-rt—>rd
```

```
                6'b100100: ALUControl<=3'b000;//add
                6'b100101: ALUControl<=3'b001;//or
                6'b101010: ALUControl<=3'b111;//slt (rs<rt)->rd,有符号
比较
                default:        ALUControl<=3'b000;
            endcase
        end
        2'b11:  ALUControl<=3'b111;//bgtz    solt
        default: ALUControl<=3'b000;
    endcase
end
endmodule
```

（3）MainDecoder

```
module MainDecoder(//CU
input [5:0]opcode,
output reg MemtoReg,
output reg MemWrite,
output reg Branch,
output reg ALUSrc,
output reg RegDst,
output reg RegWrite,
output reg Jump,
```

```
    output reg[1:0]ALUOp
      );

parameter Rtype = 6'b000000;//add
parameter lw = 6'b100011;
parameter sw = 6'b101011;
parameter beq = 6'b000100;
parameter bgtz = 6'b000111;
parameter addi = 6'b001000;
parameter j = 6'b000010;

always@(opcode)
begin
    case(opcode)
        Rtype:
            begin
            RegDst<=1;
            ALUSrc<=0;
            MemtoReg<=0;
            RegWrite<=1;
            MemWrite<=0;
            Branch<=0;
```

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

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

```
        ALUOp<=2'b11;
        Jump<=0;
        end
    addi:
        begin
        RegDst<=0;
        ALUSrc<=1;
        MemtoReg<=0;
        RegWrite<=1;
        MemWrite<=0;
        Branch<=0;
        ALUOp<=2'b00;
        Jump<=0;
        end
    j:
        begin
        RegDst<=0;
        ALUSrc<=0;
        MemtoReg<=0;
        RegWrite<=0;
        MemWrite<=0;
        Branch<=0;
```

```
            ALUOp<=2'b00;
            Jump<=1;
            end
        default:
            begin
            RegDst<=0;
            ALUSrc<=0;
            MemtoReg<=0;
            RegWrite<=0;
            MemWrite<=0;
            Branch<=0;
            ALUOp<=2'b00;
            Jump<=0;
            end
        endcase
end

endmodule
```

（4）MUX_5

```
module Mux5b(
  input [4:0]a,
  input [4:0]b,
```

```
        input sw,
        output [4:0]out
        );
assign out=sw?a:b;
endmodule
```

（5）MUX_32

```
        module Mux(
        input [31:0]a,
        input [31:0]b,
        input sw,
        output [31:0]out
        );

assign out=sw?a:b;
endmodule
```

（6）PC

```
      module pc(
      input clk,
      input rst,
      input [31:0]next_pc,
      output reg [31:0]curr_pc
        );
```

```
always@(posedge clk or negedge rst)
begin
    if(~rst)curr_pc<=0;
    else
    curr_pc<=next_pc;
end
endmodule
```

（7）Regfile

```
    module regfile(
    input clk,
    input rst,
    input [4:0]a1,
    input [4:0]a2,
    input [4:0]a3,//写地址
    input [31:0]wd,
    input we,
    output [31:0]rd1,
    output [31:0]rd2
    );

reg [31:0]r[31:0];
```

```
assign rd1=r[a1];
assign rd2=r[a2];
integer i;
always@(posedge clk or negedge rst)
begin
    if(~rst)
    begin
        for(i=0;i<32;i=i+1)
            r[i]<=0;
    end
    else if(we)
        r[a3]<=wd;
end

endmodule
```

（8）Signextend

```
module SignExtend(
    input [15:0]din,
    output [31:0]dout
    );
assign dout[31:0]={{16{din[15]}},din[15:0]};
```

endmodule

（9）DataMemory

memory_initialization_radix=16;

memory_initialization_vector=

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000000,

00000014,

00000003,

00000003,

00000000

;

（10）InsteuctionMemory

MEMORY_INITIALIZATION_RADIX=16;

MEMORY_INITIALIZATION_VECTOR=

20080000,

200d0050,

8dad0000,

200b0054,

8d6b0000,

200c0054,

8d8c0004,

ad0b0000,

ad0c0004,

21a9fffe,

8d0b0000,

```
8d0c0004,
016c5020,
ad0a0008,
21080004,
2129ffff,
1d20fff9,
08000011;
```

（11）Main_CPU

```
module main_CPU(
    input clk,
    input rst
      );


wire [31:0]IR;
wire [31:0]next_pc;
wire [31:0]curr_pc;
wire MemtoReg;
wire MemWrite;
wire Branch;
wire ALUSrc;
wire RegDst;
```

```
wire RegWrite;
wire [1:0]ALUOp;
wire [2:0]ALUControl;
wire [31:0]Result;
wire [4:0]WriteReg;
wire [31:0]data1;
wire [31:0]data2;
wire [31:0]signimm;
wire [31:0]SrcB;
wire Zerp;
wire [31:0]ALUResult;
wire [31:0]ReadData;
wire [31:0]PCPlus4;
wire [31:0]PCBranch;
wire PCSrc;
wire Jump;
wire [31:0]Jumpaddr;
wire [31:0]nonjump;
pc pc(clk,rst,next_pc,curr_pc);
InstructionMemory InstructionMemory(curr_pc[6:2],IR);
MainDecoder
MainDecoder(IR[31:26],MemtoReg,MemWrite,Branch,ALUSrc,RegDst,Re
```

```
gWrite,Jump,ALUOp);
ALUDecoder ALUDecoder(IR[5:0],ALUOp,ALUControl);
regfile
regfile(clk,rst,IR[25:21],IR[20:16],WriteReg,Result,RegWrite,data1,data2);
Mux5b Mux1(IR[15:11],IR[20:16],RegDst,WriteReg);          //5bit
SignExtend SignExtend(IR[15:0],signimm);
Mux Mux2(signimm,data2,ALUSrc,SrcB);                      //32bit
alu ALU(data1,SrcB,ALUControl,Zero,ALUResult);
DataMemory
DataMemory(ALUResult[6:2],data2,clk,MemWrite,ReadData);
Mux Mux3(ReadData,ALUResult,MemtoReg,Result);             //32bit

assign PCPlus4=curr_pc+4;
assign PCBranch=(signimm<<2)+PCPlus4;
assign PCSrc=Branch&Zero;
Mux Mux4(PCBranch,PCPlus4,PCSrc,nonjump);                 //32bit
assign Jumpaddr[27:0]=IR[25:0]<<2;
assign Jumpaddr[31:28]=PCPlus4[31:28];

Mux Mux5(Jumpaddr,nonjump,Jump,next_pc);

endmodule
```

2.仿真：

```
module maintest;


    // Inputs
    reg clk;
    reg rst;
    // Instantiate the Unit Under Test (UUT)
    main_CPU uut (
        .clk(clk),
        .rst(rst)
    );
     initial
    begin
        clk = 0;
        forever
        begin
        #5;
        clk=~clk;
        end
    end
    initial begin
        // Initialize Inputs
```

```
        clk = 0;
        rst = 0;
        // Wait 100 ns for global reset to finish
        #10;
          rst=1;
        // Add stimulus here
    end
    endmodule
```