

# 单周期 CPU

## 目录

一．指令.....	2
算术运算指令.....	2
逻辑运算指令.....	2
移位指令.....	3
访存指令.....	3
二．数据通路.....	4
三．部件实现.....	4
虚实地址转换部件.....	4
PCadd.....	5
PC.....	6
指令 RAM.....	8
符号扩展.....	10
通用寄存器堆写数据生成逻辑.....	10
通用寄存器堆写地址生成逻辑.....	11
通用寄存器堆.....	12
数据 RAM.....	14
ALUsrc1 生成逻辑.....	15
ALUsrc2 生成逻辑.....	16
ALU.....	16
指令分割.....	19
控制模块.....	20
四．顶层设计.....	23
五．测试.....	27
测试文件.....	27
仿真激励文件.....	28
波形.....	30

## 一. 指令

### 算术运算指令

ADDU

000000	rs	rt	rd	00000	1000001
--------	----	----	----	-------	---------

操作定义:  $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

ADDIU

001001	rs	rt	imm		
--------	----	----	-----	--	--

操作定义:  $GPR[rt] \leftarrow GPR[rs] + \text{sign\_extend}(imm)$

SUBU

000000	rs	rt	rd	00000	1000011
--------	----	----	----	-------	---------

操作定义:  $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

SLT

000000	rs	rt	rd	00000	101010
--------	----	----	----	-------	--------

操作定义: if  $GPR[rs] < GPR[rt]$  then  $GPR[rd] \leftarrow 1$   
else  $GPR[rd] \leftarrow 0$  endif

SLTU

000000	rs	rt	rd	00000	101011
--------	----	----	----	-------	--------

操作定义: if  $(0 \| GPR[rs]_{31..0}) < (0 \| GPR[rt]_{31..0})$  then  
     $GPR[rd] \leftarrow 1$   
else  
     $GPR[rd] \leftarrow 0$   
endif

### 逻辑运算指令

AND

000000	rs	rt	rd	00000	100100
--------	----	----	----	-------	--------

操作定义:  $GPR[rd] \leftarrow GPR[rs] \& GPR[rt]$

LUI

001111	00000	rt	imm		
--------	-------	----	-----	--	--

操作定义:  $GPR[rt] \leftarrow (imm \| 0^{16})$

NOR

000000	rs	rt	rd	00000	100111
--------	----	----	----	-------	--------

操作定义:  $GPR[rd] \leftarrow GPR[rs] \text{nor } GPR[rt]$

OR

000000	rs	rt	rd	00000	100101
--------	----	----	----	-------	--------

操作定义:  $GPR[rd] \leftarrow GPR[rs] \text{or } GPR[rt]$

XOR

000000	rs	rt	rd	00000	100110
--------	----	----	----	-------	--------

操作定义： $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \text{ xor } \text{GPR}[\text{rt}]$

## 移位指令

SLL 逻辑左移

000000	00000	rt	rd	sa	000000
--------	-------	----	----	----	--------

操作定义： $s \leftarrow sa$

$$\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}]_{(31-s)..0} \parallel 0^s$$

SRA 算术右移

000000	00000	rt	rd	sa	000011
--------	-------	----	----	----	--------

操作定义： $s \leftarrow sa$

$$\text{GPR}[\text{rd}] \leftarrow (\text{GPR}[\text{rt}]_{31})^s \parallel \text{GPR}[\text{rt}]_{31..s}$$

SRL 逻辑右移

000000	00000	rt	rd	sa	000010
--------	-------	----	----	----	--------

操作定义： $s \leftarrow sa$

$$\text{GPR}[\text{rd}] \leftarrow 0^s \parallel \text{GPR}[\text{rt}]_{31..s}$$

## 访存指令

LW

100011	base	rt	offset
--------	------	----	--------

汇编格式：LW rt,offset(base)

SW

101011	base	rt	offset
--------	------	----	--------

操作定义： $vAddr \leftarrow \text{GPR}[\text{base}] + \text{sign\_extend}(\text{offset})$

if  $vAddr_{1..0} \neq 0$  then

    SignalException(AddressError)

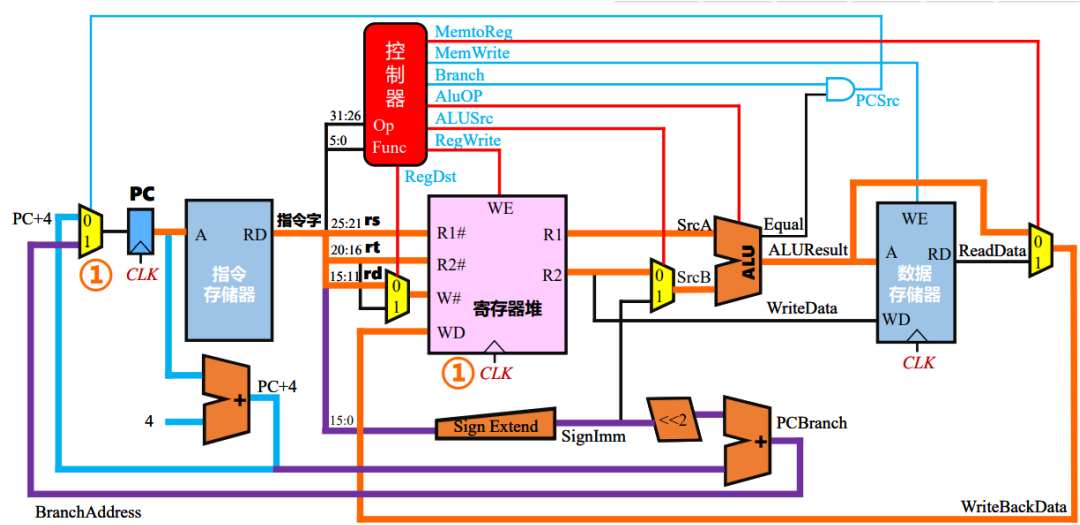
endif

$(pAddr, CCA) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE})$

$\text{dataword} \leftarrow \text{GPR}[\text{rt}]_{31..0}$

StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)

## 二 . 数据通路



## 三 . 部件实现

### 虚实地址转换部件

代码:

//2021-5-24

//单周期 CPU

//虚实地址转换部件

`timescale 1ns/10ps

module Addr\_transform(

input [31:0] tran\_in,

output [31:0] tran\_out

);

assign tran\_out = (tran\_in[31:29] == 3'b100) ? {1'b0,tran\_in[30:0]} :

(tran\_in[31:29] == 3'b101) ? {3'b000,tran\_in[26:0]} :

tran\_in;

endmodule

仿真:

//-----testbench of Addr\_tansform-----

module Addr\_transform\_tb();

reg [31:0] tran\_in;

wire [31:0] tran\_out;

```

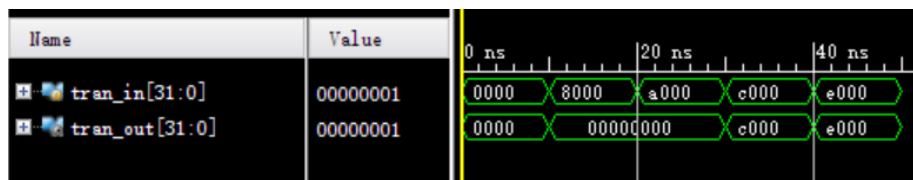
Addr_transform Addr_transform(
    .tran_in(tran_in),
    .tran_out(tran_out)
);

initial begin
    tran_in = 32'h0000_0001;
    #10 tran_in = 32'h8000_0000;
    #10 tran_in = 32'hA000_0000;
    #10 tran_in = 32'hC000_0000;
    #10 tran_in = 32'hE000_0000;
end

```

endmodule

波形：



## PCadd

代码：

//2021-6-7,吴佳宾

//PC+4

`timescale 1ns/10ps

module PCadd(

input CLK, //时钟信号

input reset, //复位信号

input [31:0] oldPC, //当前指令地址

output reg [31:0] newPC //下一条指令地址

);

//这里设为下降沿触发是为了让 PC 在上升沿前准备好下一条指令的地址

always@(negedge CLK or posedge reset) begin

if(reset) begin

newPC <= 32'd0;

end

else begin

newPC <= oldPC + 4;

end

end

```
endmodule
```

仿真：

```
//-----testbench of PCadd-----
```

```
module PCadd_tb();
```

```
reg CLK;
```

```
reg reset;
```

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

```
wire [31:0] newPC;
```

```
PCadd PCadd(
```

```
    .CLK(CLK),
```

```
    .reset(reset),
```

```
    .oldPC(oldPC),
```

```
    .newPC(newPC)
```

```
);
```

```
initial begin
```

```
    CLK = 1'b0;
```

```
    reset = 1'b0;
```

```
    oldPC = 32'h0000_0000;
```

```
    #32 reset = 1;
```

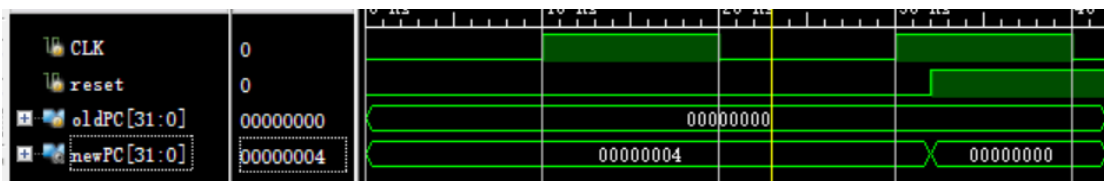
```
    #10 $stop;
```

```
end
```

```
always #10 CLK = ~CLK;
```

```
endmodule
```

波形：



## PC

代码：

```
//2021-5-23,吴佳宾
```

```
//PC
```

```
`timescale 1ns/10ps
```

```
module PC(
```

```
    input                CLK, //时钟信号
```

```
    input                reset, //是否复位
```

```

        input    [31:0]      PC_src, //PC+4 得到的值
        output   reg [31:0]  PC_out //当前指令的地址
    );

always@(posedge CLK or posedge reset) begin
    if(reset) PC_out <= 32'hBFC00000; //复位值
    else      PC_out <= PC_src;
end
测试:
//2021-6-11
//PC 功能测试
`timescale 1ns/10ps
module PCtest(
    input          CLK, //时钟信号
    input          reset, //是否复位
    output [31:0]  PC_src, //PC+4 得到的值
    output [31:0]  PC_out //当前指令的地址
);

    PC PC(
        .CLK(CLK),
        .reset(reset),
        .PC_src(PC_src),
        .PC_out(PC_out)
    );

    PCadd PCadd(
        .CLK(CLK),
        .reset(reset),
        .oldPC(PC_out),
        .newPC(PC_src)
    );

endmodule

//-----testbench of PCtest-----
module PCtest_tb();
    reg CLK, reset;
    wire [31:0] PC_src;
    wire [31:0] PC_out;
    PCtest PCtest(
        .CLK(CLK),
        .reset(reset),
        .PC_src(PC_src),
        .PC_out(PC_out)
    );

```

```

);
initial begin
    CLK = 0;
    reset = 0;
    #5 reset = 1;
    #7 reset = 0;
    #55 reset = 1;
end
always #10 CLK = ~CLK;
endmodule
波形：

```



## 指令 RAM

```

代码：
//2021-5-25, 吴佳宾
//单周期 CPU
//指令 RAM
`timescale 1ns/10ps
module IR(
    input          inst_ram_en, //片选信号, 恒为 1
    input          inst_ram_wen, //写使能信号, 恒为 0
    input  [31:0]  IR_addr, //指令地址
    output [31:0]  IR_out //IR 输出
);
wire [31:0]Byte_addr; //指令 RAM 中按字节寻址
reg [31:0]IR_array[31:0]; //IR 数据宽度 32 位 大小 64B

//加载数据到存储器, 必须使用绝对路径
initial begin
    $readmemb("D:\\Code\\Verilog\\Single_CPU\\IR.txt", IR_array);
end

assign Byte_addr = IR_addr >> 2; //将地址转换为字节地址
assign IR_out = inst_ram_en ? IR_array[Byte_addr] : 32'd0; //片选信号有效时读取数据

endmodule

//-----testbench of IR-----
module IR_tb();
reg inst_ram_en, inst_ram_wen;

```



```

reg [31:0] IR_addr;
wire [31:0] IR_out;
IR IR_inst(
    .inst_ram_en(inst_ram_en), //片选信号, 恒为 1
    .inst_ram_wen(inst_ram_wen), //写使能信号, 恒为 0
    .IR_addr(IR_addr), //指令地址
    .IR_out(IR_out) //IR 输出
);
initial begin
    inst_ram_en <= 1;
    inst_ram_wen <= 0;
    IR_addr <= 32'd0;
    $stop;
end
endmodule

```

仿真:

//-----testbench of IR-----

```

module IR_tb();
reg inst_ram_en, inst_ram_wen;
reg [31:0] IR_addr;
wire [31:0] IR_out;

IR IR(
    .inst_ram_en(inst_ram_en),
    .inst_ram_wen(inst_ram_wen),
    .IR_addr(IR_addr),
    .IR_out(IR_out)
);

```

```

initial begin
    inst_ram_en = 1;
    inst_ram_wen = 1;
    IR_addr = 32'd0;
    #10 IR_addr = 32'd4;
    #10 IR_addr = 32'd8;
    #10 IR_addr = 32'd12;
    #10 IR_addr = 32'd16;
    #10 IR_addr = 32'd20;
    #10 IR_addr = 32'd24;
    $stop;
end
endmodule

```

波形:

inst_ram_en	1						
inst_ram_wen	1						
IR_addr[31:0]	00000000	00000000	00000008	0000000c	00000010	00000014	
IR_out[31:0]	00221821	24230001	00221823	0022182a	0022182b	00221824	

## 符号扩展

代码：

//2021-6-8,吴佳宾

//符号扩展

`timescale 1ns/10ps

module SignExtend(

input [15:0] extin, //符号扩展输入

output [31:0] extout //符号扩展输出

);

//符号扩展

assign extout = {{16{extin[15]}}, extin};

endmodule

`timescale 1ns/10ps

//2021-6-15,吴佳宾

//SignExtend\_sa

module SignExtend\_sa(

input [4:0] extin, //符号扩展输入

output [31:0] extout //符号扩展输出

);

//符号扩展

assign extout = {{27{extin[4]}}, extin};

endmodule

## 通用寄存器堆写数据生成逻辑

代码：

//2021-6-1, 吴佳宾

//通用寄存器堆写数据生成逻辑

`timescale 1ns/10ps

module GenRFRes(

input [31:0] in0, //in0 对应 ALU 计算结果 alu\_res

input [31:0] in1, //in1 对应从 RAM 读出的 load 操作返回值 ld\_res

input sel\_rf\_res, //选择信号 1 选择 DRAM 输出 0 对应 ALU 输出

output [31:0] rf\_res //数据输出

```

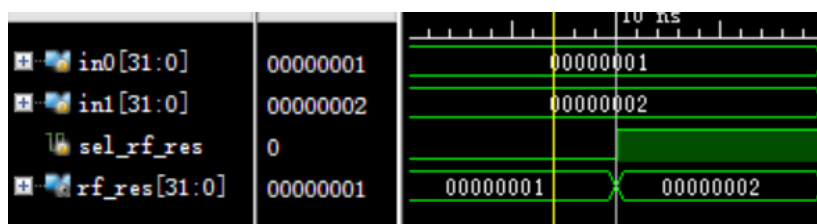
    );

    assign rf_res = sel_rf_res ? in1 : in0;

endmodule;
仿真：
`timescale 1ns/10ps
//-----testbench of GenRFRes-----
module GenRFRes_tb();
    reg [31:0] in0;
    reg [31:0] in1;
    reg sel_rf_res;
    wire [31:0] rf_res;
    GenRFRes GenRFRes(
        .in0(in0),
        .in1(in1),
        .sel_rf_res(sel_rf_res),
        .rf_res(rf_res)
    );
    initial begin
        in0 = 32'h0000_0001;
        in1 = 32'h0000_0002;
        sel_rf_res = 0;
        #10 sel_rf_res = 1;
        #10 $stop;
    end
endmodule

```

波形：



## 通用寄存器堆写地址生成逻辑

代码：

```

//2021-6-1,吴佳宾
//通用寄存器堆写地址生成逻辑
`timescale 1ns/10ps
module GenRFDst(
    input [4:0] in0, //对应指令的 rd 域

```

```

        input [4:0] in1, //对应指令的 rt 域
        input [2:0] sel_rf_dst, //选择信号
        output [4:0] rf_dst //寄存器堆写地址
    );

    assign rf_dst = sel_rf_dst[0] ? in0 :
                    sel_rf_dst[1] ? in1 :
                    sel_rf_dst[2] ? 5'd31 : 5'd0;

endmodule

```

## 通用寄存器堆

代码：

```

//2021-6-1, 吴佳宾
//通用寄存器堆 RegFile
`timescale 1ns/10ps
module RegFile(
    input          CLK, //时钟信号
    input          we, //写使能信号
    input  [4:0] waddr, //寄存器堆写地址
    input  [4:0] raddr1, //读地址 1
    input  [4:0] raddr2, //读地址 2
    input  [31:0] wdata, //写数据
    output [31:0] rdata1, //读数据 1
    output [31:0] rdata2 //读数据 2
);

reg [31:0] reg_array[31:0]; //32 个 32 位宽的数据
//加载数据到存储器，必须使用绝对路径
initial begin
    $readmemh("D:\\Code\\Verilog\\Single_CPU\\RegFile.txt", reg_array);
end
//Write
always@(posedge CLK) begin
    if(we) reg_array[waddr] <= wdata;
end
//Read out 1
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 : reg_array[raddr1];
//Read out 2
assign rdata2 = (raddr2 == 5'b0) ? 32'b0 : reg_array[raddr2];

endmodule

```

仿真：

```
`timescale 1ns/10ps
//-----testbench of RegFile-----
module RegFile_tb();
reg CLK, we;
reg [ 4:0] waddr;
reg [ 4:0] raddr1;
reg [ 4:0] raddr2;
reg [31:0] wdata;
wire [31:0] rdata1;
wire [31:0] rdata2;
RegFile RegFile(
    .CLK(CLK),
    .we(we),
    .waddr(waddr),
    .raddr1(raddr1),
    .raddr2(raddr2),
    .wdata(wdata),
    .rdata1(rdata1),
    .rdata2(rdata2)
);
initial begin
    CLK = 0;
    we = 1;
    waddr = 5'd31;
    wdata = 32'h0000_0001;
    raddr1 = 5'd0;
    raddr2 = 5'd1;
    #10 raddr2 = 5'd2;
    #10 raddr2 = 5'd3;
    #10 raddr2 = 5'd4;
    #10 raddr2 = 5'd30;
    #10 raddr2 = 5'd31;
    #10 $stop;
end

always #10 CLK = ~CLK;

endmodule
波形：
```

CLK	0						
we	1						
waddr[4:0]	1f				1f		
raddr1[4:0]	00				00		
raddr2[4:0]	01	01	02	03	04	1e	1f
wdata[31:0]	00000001				00000001		
rdata1[31:0]	00000000				00000000		
rdata2[31:0]	00000001	00000001	00000002	00000003	f0000000	XXXXXXXX	00000001

## 数据 RAM

代码：

//2021-6-2,吴佳宾

//数据 RAM

`timescale 1ns/10ps

module DR(

```

    input      CLK, //时钟
    input      en, //片选信号
    input      we, //写使能
    input [4:0] addr, //读、写地址
    input [31:0] wdata, //写数据
    output [31:0] rdata //读数据

```

);

reg [31:0] reg\_array [31:0];

//加载数据到存储器，必须使用绝对路径

initial begin

```

    $readmemh("D:/Codes/Vivado/Single_CPU/Single_CPU.srcs/sources_1/new/DR.txt",
    reg_array);

```

end

//Write

always@(negedge CLK) begin

```

    if(en && we) reg_array[addr] <= wdata;

```

end

//Read

```

assign rdata = en ? reg_array[addr] : 32'd0;

```

endmodule

仿真：

`timescale 1ns/10ps

//2021-6-14,吴佳宾

//-----testbench of DR-----

module DR\_tb();

reg CLK, we;

reg [4:0] addr;

```

reg [31:0] wdata;
wire [31:0] rdata;
DR DR(
    .CLK(CLK),
    .we(we),
    .addr(addr),
    .wdata(wdata),
    .rdata(rdata)
);
initial begin
    CLK = 0;
    we = 0;
    addr = 5'd0;
    #10 addr = 5'd1; we = 1; wdata = 32'hffff_ffff;
    #10 $stop;
end

always #10 CLK = ~CLK;

endmodule

```

## ALUsrc1 生成逻辑

代码：

```

`timescale 1ns/10ps
//2021-6-15,吴佳宾
//ALUsrc1 生成逻辑
module GenALUSrc1(
    input [31:0] in0, //rf_rdata1
    input [31:0] in1, //PC
    input [31:0] in2, //signext(sa)
    input [2:0] sel_alu_src1, //选择信号
    output [31:0] alu_src1 //ALU 源操作数 1
);

assign alu_src1 = sel_alu_src1[0] ? in0 :
                  sel_alu_src1[1] ? in1 :
                  sel_alu_src1[2] ? in2 :
                  32'd0;

endmodule

```

## ALUSrc2 生成逻辑

代码:

//2021-6-8,吴佳宾

//ALU 源操作数输入 alu\_src2 生成逻辑

```
module GenALUSrc2(
    input [31:0] in0, //对应寄存器堆读端口 2 数据读出
    input [31:0] in1, //对应指令码的 imm 域扩展至 32 位
    input [31:0] in2, //对应常值 32'd8
    input [2:0] sel_alu_src2, //选择信号
    output [31:0] alu_src2 //alu_src2
);

assign alu_src2 = (sel_alu_src2[0]) ? in0 :
                  (sel_alu_src2[1]) ? in1 :
                  (sel_alu_src2[2]) ? in2 :
                  32'd0;

endmodule
```

## ALU

代码:

//2021-6-1,吴佳宾

//ALU

`timescale 1ns/10ps

```
module ALU(
    input [11:0] alu_control,
    input [31:0] alu_src1,
    input [31:0] alu_src2,
    output [31:0] alu_result
);

wire op_add; //加法操作
wire op_sub; //减法操作
wire op_slt; //有符号比较, 小于置位
wire op_sltu; //无符号数比较, 小于置位
wire op_and; //按位与
wire op_nor; //按位或非
wire op_or; //按位或
wire op_xor; //按位异或
wire op_sll; //逻辑左移
```



```
wire op_srl;    //逻辑右移
wire op_sra;    //算术右移
wire op_lui;    //高位加载
```

```
assign op_add    = alu_control[ 0];
assign op_sub    = alu_control[ 1];
assign op_slt   = alu_control[ 2];
assign op_sltu  = alu_control[ 3];
assign op_and    = alu_control[ 4];
assign op_nor   = alu_control[ 5];
assign op_or    = alu_control[ 6];
assign op_xor    = alu_control[ 7];
assign op_sll   = alu_control[ 8];
assign op_srl   = alu_control[ 9];
assign op_sra   = alu_control[10];
assign op_lui   = alu_control[11];
```

```
wire [31:0] add_sub_result;
wire [31:0] slt_result;
wire [31:0] sltu_result;
wire [31:0] and_result;
wire [31:0] nor_result;
wire [31:0] or_result;
wire [31:0] xor_result;
wire [31:0] sll_result;
wire [31:0] srl_result;
wire [31:0] sra_result;
wire [31:0] lui_result;
```

```
assign and_result = alu_src1 & alu_src2;
assign or_result  = alu_src1 | alu_src2;
assign nor_result = ~or_result;
assign xor_result = alu_src1 ^ alu_src2;
assign lui_result = {alu_src2[15:0], 16'b0};
```

```
wire [31:0] adder_a;
wire [31:0] adder_b;
wire      adder_cin;
wire [31:0] adder_result;
wire      adder_cout;
```

```
assign adder_a    = alu_src1;
assign adder_b    = (op_sub | op_slt | op_sltu) ? ~alu_src2 : alu_src2;
assign adder_cin  = (op_sub | op_slt | op_sltu) ?      1'b1 :      1'b0;
```

```

assign {adder_cout, adder_result} = adder_a + adder_b + adder_cin;

assign add_sub_result = adder_result;

assign slt_result[31:1] = 31'b0;
//带符号数比较
//1.如果 src1 为正数, src2 为负数, src1>src2, 返回 0;
//2.如果 src1 与 src2 同号, 则看相减结果的符号位, 为正则 src1 > src2, 返回 0; 为负则 src1
< src2, 返回 1
assign slt_result[0] = (alu_src1[31] & ~alu_src2[31])
                      | (~(alu_src1[31] ^ alu_src2[31]) & adder_result[31]);
//无符号数比较
assign sltu_result[31:1] = 31'b0;
assign sltu_result[0]    = ~adder_cout;
//逻辑左移
assign sll_result = alu_src2 << alu_src1[4:0];
//逻辑右移
assign srl_result = alu_src2 >> alu_src1[4:0];
//算术右移
assign sra_result = ($signed(alu_src2) >>> alu_src1[4:0]);

assign  alu_result = ({32{op_add|op_sub }} & add_sub_result)
                    | ({32{op_sltu   }} & slt_result)
                    | ({32{op_and     }} & and_result)
                    | ({32{op_nor     }} & nor_result)
                    | ({32{op_or      }} & or_result)
                    | ({32{op_xor     }} & xor_result)
                    | ({32{op_sll     }} & sll_result)
                    | ({32{op_srl     }} & srl_result)
                    | ({32{op_sra     }} & sra_result)
                    | ({32{op_lui     }} & lui_result);

endmodule
仿真:
//-----testbench of ALU-----
`timescale 1ns/10ps
module ALU_tb();
reg [11:0]alu_control;
reg [31:0]alu_src1, alu_src2;
wire [31:0]alu_result;
ALU ALU(
    .alu_control(alu_control),
    .alu_src1(alu_src1),
    .alu_src2(alu_src2),

```

```

        .alu_result(alu_result)
    );

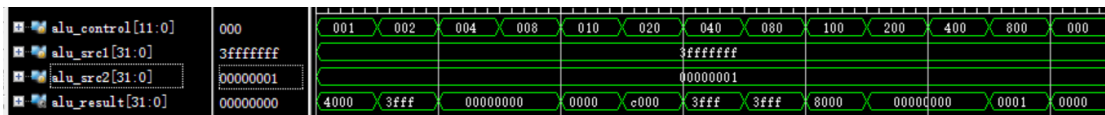
    initial
    begin
        alu_control = 12'd1;
        alu_src1 = 32'h3fffffff;
        alu_src2 = 32'd1;
        #130 $stop;
    end

    always #10 alu_control = alu_control << 1;

endmodule

```

波形：



## 指令分割

代码：

```

`timescale 1ns/10ps
//2021-6-1,吴佳宾
//指令分割
module Instruct_Split(
    input    [31:0]    instruct, //指令
    output   [5:0]     op, //操作码
    output   [4:0]     rs, //rs 域
    output   [4:0]     rt, //rt 域
    output   [4:0]     rd, //rd 域
    output   [4:0]     sa, //sa 域
    output   [15:0]    imm, //imm 立即数
    output   [5:0]     func //ALU 操作码
);

    assign op = instruct[31:26];
    assign rs = instruct[25:21];
    assign rt = instruct[20:16];
    assign rd = instruct[15:11];
    assign sa = instruct[10: 6];
    assign imm = instruct[15:0];
    assign func = instruct[5:0];

```

endmodule

仿真：

//-----testbench of Instruct\_Split-----

```
module Instruct_Split_tb();
```

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

```
wire [5:0] op;
```

```
wire [4:0] rs;
```

```
wire [4:0] rt;
```

```
wire [4:0] rd;
```

```
wire [4:0] sa;
```

```
wire [15:0] imm;
```

```
wire [5:0] func;
```

```
Instruct_Split Instruct_Split(
```

```
    .instruct(instruct),
```

```
    .op(op),
```

```
    .rs(rs),
```

```
    .rt(rt),
```

```
    .rd(rd),
```

```
    .sa(sa),
```

```
    .imm(imm),
```

```
    .func(func)
```

```
);
```

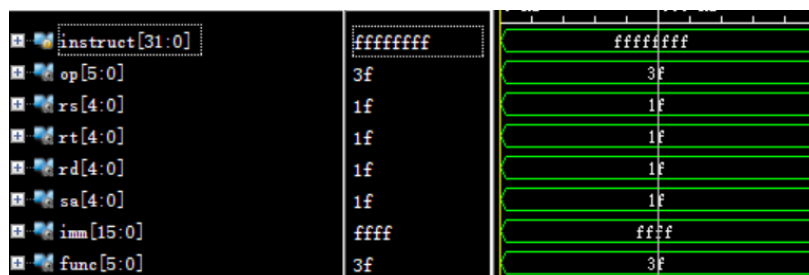
```
initial begin
```

```
    instruct = 32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
end
```

endmodule

波形：



## 控制模块

代码：

//2021-6-1, 吴佳宾

//指令译码

`timescale 1ns/10ps

```

module ControlUnit(
    input  [ 5:0]  op, //操作
    input  [ 4:0]  sa, //sa 域
    input  [ 5:0]  func, //ALU 操作码
    output          inst_ram_en, //指令 RAM 片选
    output          inst_ram_wen, //指令 RAM 读使能
    output [ 2:0]   sel_alu_src1, //ALU 源操作数 1
    output [ 2:0]   sel_alu_src2, //ALU 源操作数 2
    output [11:0]   alu_op, //ALU 操作码
    output          data_ram_en, //DataRam 片选
    output          data_ram_wen, //DataRam 写使能
    output          rf_we, //寄存器堆写使能
    output [ 2:0]   sel_rf_dst, //寄存器堆写地址生成逻辑
    output          sel_rf_res //寄存器堆读地址生成逻辑
);

/*
op 对应指令
000000 ADDU ADD SUB SLT SLTU AND
001001 ADDIU
001010 SLTI

*/
wire inst_addu;
wire inst_addiu;
wire inst_subu;
wire inst_lw;
wire inst_sw;
wire inst_slt;
wire inst_sltu;
wire inst_sll;
wire inst_srl;
wire inst_sra;
wire inst_lui;
wire inst_and;
wire inst_or;
wire inst_xor;
wire inst_nor;
//指令译码
assign inst_addu = (op == 6'b000000) && (sa == 5'b000000) && (func == 6'b100001);
assign inst_addiu = (op == 6'b001001);
assign inst_subu = (op == 6'b000000) && (sa == 5'b000000) && (func == 6'b100011);
assign inst_slt = (op == 6'b000000) && (sa == 5'b000000) && (func == 6'b101010);
assign inst_sltu = (op == 6'b000000) && (sa == 5'b000000) && (func == 6'b101011);
assign inst_sll = (op == 6'b000000) && sa && (func == 6'b000000);


















```

```

assign inst_srl = (op == 6'b000000) && sa && (func == 6'b000010);
assign inst_sra = (op == 6'b000000) && sa && (func == 6'b000011);
assign inst_lui = (op == 6'b001111);
assign inst_and = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100100);
assign inst_or = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100101);
assign inst_xor = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100110);
assign inst_nor = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100111);
assign inst_lw = (op == 6'b100011);
assign inst_sw = (op == 6'b101011);
//控制信号生成
assign inst_ram_en = 1;
assign inst_ram_wen = 1;
assign sel_alu_src1 = {inst_sll | inst_srl | inst_sra, 1'b0, inst_addu | inst_addiu | inst_subu |
                    inst_lw | inst_sw | inst_slt | inst_sltu | inst_and | inst_or | inst_xor |
                    inst_nor};
assign sel_alu_src2 = {1'b0, inst_addiu | inst_lw | inst_sw, inst_addu | inst_subu | inst_slt |
inst_sltu | inst_sll |
                    inst_srl | inst_sra | inst_and | inst_or | inst_xor | inst_nor};
assign data_ram_en = inst_lw | inst_sw;
assign data_ram_wen = inst_sw;
assign rf_we = inst_addu | inst_addiu | inst_subu | inst_lw | inst_sltu | inst_slt | inst_sll | inst_srl
            | inst_sra | inst_lui | inst_and | inst_or | inst_xor | inst_nor;
assign sel_rf_dst = {1'b0, inst_addiu | inst_lw | inst_lui, inst_addu | inst_subu | inst_slt | inst_sltu
| inst_sll
            | inst_srl | inst_sra | inst_and | inst_or | inst_xor | inst_nor};
assign sel_rf_res = inst_lw;
assign alu_op = {inst_lui, inst_sra, inst_srl, inst_sll, inst_xor, inst_or, inst_nor,
                inst_and, inst_sltu, inst_slt, inst_subu, inst_addu | inst_addiu | inst_lw |
inst_sw};

endmodule
波形:

```

 instruct[31:0]	00000000001	0000000000100010000110000
 op[5:0]	000000	000000
 rs[4:0]	00001	00001
 rt[4:0]	00010	00010
 rd[4:0]	00011	00011
 sa[4:0]	00000	00000
 imm[15:0]	00011000001	0001100000100001
 func[5:0]	100001	100001
 inst_ram_en	1	
 inst_ram_wen	1	
 sel_alu_src2[2:0]	001	001
 alu_op[11:0]	00000000000	00000000001
 data_ram_en	0	
 data_ram_wen	0	
 rf_we	1	
 sel_rf_dst[2:0]	001	001
 sel_rf_res	0	

## 四．顶层设计

代码：

```
`timescale 1ns/10ps
```

```
//2021-6-8,吴佳宾
```

```
//Single_CPU 顶层文件
```

```
module top(
```

```
    input                CLK, //时钟信号
```

```
    input                reset, //复位信号
```

```
    //指令分割
```

```
    output [31:0]        instruct, //指令
```

```
    output [ 5:0]        op, //操作码
```

```
    output [ 4:0]        rs, //rs 域
```

```
    output [ 4:0]        rt, //rt 域
```

```
    output [ 4:0]        rd, //rd 域
```

```
    output [ 4:0]        sa, //sa 域
```

```
    output [15:0]        imm, //imm 立即数
```

```
    output [ 5:0]        func, //ALU 操作码
```

```
    //PC
```

```
    output [31:0]        PC_src, //PC+4
```

```
    output [31:0]        PC_out, //当前指令
```

```
    //IR
```

```
    output [31:0]        IR_addr, //IR 地址输入
```

```
    //Control_Unit
```

```
    output                inst_ram_en, //指令 RAM 片选
```

```
    output                inst_ram_wen, //指令 RAM 读使能
```

```
    output [ 2:0]        sel_alu_src1, //ALU 源操作数 1
```

```
    output [ 2:0]        sel_alu_src2, //ALU 源操作数 2
```

```
    output [11:0]        alu_op, //ALU 操作码
```

```

output          data_ram_en, //DataRam 片选
output          data_ram_wen, //DataRam 读使能
output          rf_we, //寄存器堆写使能
output [ 2:0]    sel_rf_dst, //寄存器堆写地址生成逻辑
output          sel_rf_res, //寄存器堆读地址生成逻辑
//ALU
output [31:0]    alu_result, //ALU 运算结果
output [31:0]    alu_src1, //ALU 操作数 1
output [31:0]    alu_src2, //ALU 操作数 2
//Signextend
output [31:0]    imm_ext, //指令码 imm 扩展 32 位
output [31:0]    sa_ext, //移位 sa 扩展 32 位
//RegFile
output [ 4:0]    rf_waddr, //寄存器堆写地址
output [ 4:0]    rf_raddr1, //regfile 读地址 1
output [ 4:0]    rf_raddr2, //regfile 读地址 2
output [31:0]    rf_wdata, //regfile 写数据
output [31:0]    rf_rdata1, //regfile 读数据 1
output [31:0]    rf_rdata2, //regfile 读数据 2
//DataRam
output [ 4:0]    dr_addr, //DR 读、写地址
output [31:0]    dr_wdata, //DR 写数据
output [31:0]    dr_rdata //DR 读数据
);
assign rf_raddr1 = rs;
assign rf_raddr2 = rt;
assign dr_addr = alu_result;
assign dr_wdata = rf_rdata2;
PC PC(
    .CLK(CLK),
    .reset(reset),
    .PC_src(PC_src),
    .PC_out(PC_out)
);
PCadd PCadd(
    .CLK(CLK),
    .reset(reset),
    .oldPC(PC_out),
    .newPC(PC_src)
);
Addr_transform Addr_transform_1(
    .tran_in(PC_out),
    .tran_out(IR_addr)
);

```



```

IR IR(
    .inst_ram_en(inst_ram_en),
    .inst_ram_wen(inst_ram_wen),
    .IR_addr(IR_addr),
    .IR_out(instruct)
);
Instruct_Split Instruct_Split(
    .instruct(instruct),
    .op(op),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .sa(sa),
    .imm(imm),
    .func(func)
);
ControlUnit ControlUnit(
    .op(op),
    .sa(sa),
    .func(func),
    .inst_ram_en(inst_ram_en),
    .inst_ram_wen(inst_ram_wen),
    .sel_alu_src1(sel_alu_src1),
    .sel_alu_src2(sel_alu_src2),
    .alu_op(alu_op),
    .data_ram_en(data_ram_en),
    .data_ram_wen(data_ram_wen),
    .rf_we(rf_we),
    .sel_rf_dst(sel_rf_dst),
    .sel_rf_res(sel_rf_res)
);
Addr_transform Addr_transform_2(
    .tran_in(alu_result),
    .tran_out(dr_addr)
);
DR DR(
    .CLK(CLK),
    .en(data_ram_en),
    .we(data_ram_wen),
    .addr(dr_addr),
    .wdata(dr_wdata),
    .rdata(dr_rdata)
);
GenRFDst GenRFDst(

```

```

        .in0(rd),
        .in1(rt),
        .sel_rf_dst(sel_rf_dst),
        .rf_dst(rf_waddr)
    );
    GenRFRes GenRFRes(
        .in0(alu_result),
        .in1(dr_rdata),
        .sel_rf_res(sel_rf_res),
        .rf_res(rf_wdata)
    );
    SignExtend SignExtend(
        .extin(imm),
        .extout(imm_ext)
    );
    SignExtend_sa SignExtend_sa(
        .extin(sa),
        .extout(sa_ext)
    );
    GenALUSrc1 GenALUSrc1(
        .in0(rf_rdata1),
        .in1(PC_out),
        .in2(sa_ext),
        .sel_alu_src1(sel_alu_src1),
        .alu_src1(alu_src1)
    );
    GenALUSrc2 GenALUSrc2(
        .in0(rf_rdata2),
        .in1(imm_ext),
        .in2(32'd8),
        .sel_alu_src2(sel_alu_src2),
        .alu_src2(alu_src2)
    );
    ALU ALU(
        .alu_control(alu_op),
        .alu_src1(alu_src1),
        .alu_src2(alu_src2),
        .alu_result(alu_result)
    );
    RegFile RegFile(
        .CLK(CLK),
        .we(we),
        .waddr(rf_waddr),
        .raddr1(rf_raddr1),

```

```

        .raddr2(rf_raddr2),
        .wdata(rf_wdata),
        .rdata1(rf_rdata1),
        .rdata2(rf_rdata2)
    );

endmodule;

```

## 五．测试

### 测试文件

IR.txt
000000_00001_00010_00011_00000_100001 001001_00001_00011_00000000000000001 000000_00001_00010_00011_00000_100011 000000_00001_00010_00011_00000_101010 000000_00001_00010_00011_00000_101011 000000_00001_00010_00011_00000_100100 001111_00000_00011_00000000000000001 000000_00001_00010_00011_00000_100111 000000_00001_00010_00011_00000_100101 000000_00001_00010_00011_00000_100110 000000_00000_00010_00011_00001_000000 000000_00000_00010_00011_00001_000011 000000_00000_00100_00011_00001_000010 100011_00101_00011_00000000000000000 101011_00101_00011_00000000000000000
RegFile.txt
0000_0000 0000_0001 0000_0002 0000_0003 F000_0000 0000_0004
DR.txt
0000_0000 0000_0001 0000_0002 0000_0003 0000_0004

## 仿真激励文件

```
//2021-6-14;吴佳宾
//Single_CPU 顶层测试文件
//-----testbench of top-----
module top_tb();
reg          CLK; //时钟信号
reg          reset; //复位信号
wire  [31:0] instruct; //指令
wire  [ 5:0] op; //操作码
wire  [ 4:0] rs; //rs 域
wire  [ 4:0] rt; //rt 域
wire  [ 4:0] rd; //rd 域
wire  [ 4:0] sa; //sa 域
wire  [15:0] imm; //imm 立即数
wire  [ 5:0] func; //ALU 操作码
wire  [31:0] PC_src; //PC+4
wire  [31:0] PC_out; //当前指令
wire  [31:0] IR_addr; //IR 地址输入
wire          inst_ram_en; //指令 RAM 片选
wire          inst_ram_wen; //指令 RAM 读使能
wire  [ 2:0] sel_alu_src2; //ALU 源操作数 2
wire  [11:0] alu_op; //ALU 操作码
wire          data_ram_en; //DataRam 片选
wire          data_ram_wen; //DataRam 读使能
wire          rf_we; //寄存器堆写使能
wire  [ 2:0] sel_rf_dst; //寄存器堆写地址生成逻辑
wire          sel_rf_res; //寄存器堆读地址生成逻辑
wire  [31:0] alu_result; //ALU 运算结果
wire  [31:0] alu_src1; //ALU 操作数 1
wire  [31:0] alu_src2; //ALU 操作数 2
wire  [31:0] imm_ext; //指令码 imm 扩展 32 位
wire  [ 4:0] rf_waddr; //寄存器堆写地址
wire  [ 4:0] rf_raddr1; //regfile 读地址 1
wire  [ 4:0] rf_raddr2; //regfile 读地址 2
wire  [31:0] rf_wdata; //regfile 写数据
wire  [31:0] rf_rdata1; //regfile 读数据 1
wire  [31:0] rf_rdata2; //regfile 读数据 2
wire  [ 4:0] dr_addr; //DR 读、写地址
wire  [31:0] dr_wdata; //DR 写数据
wire  [31:0] dr_rdata; //DR 读数据

top top(
```

```

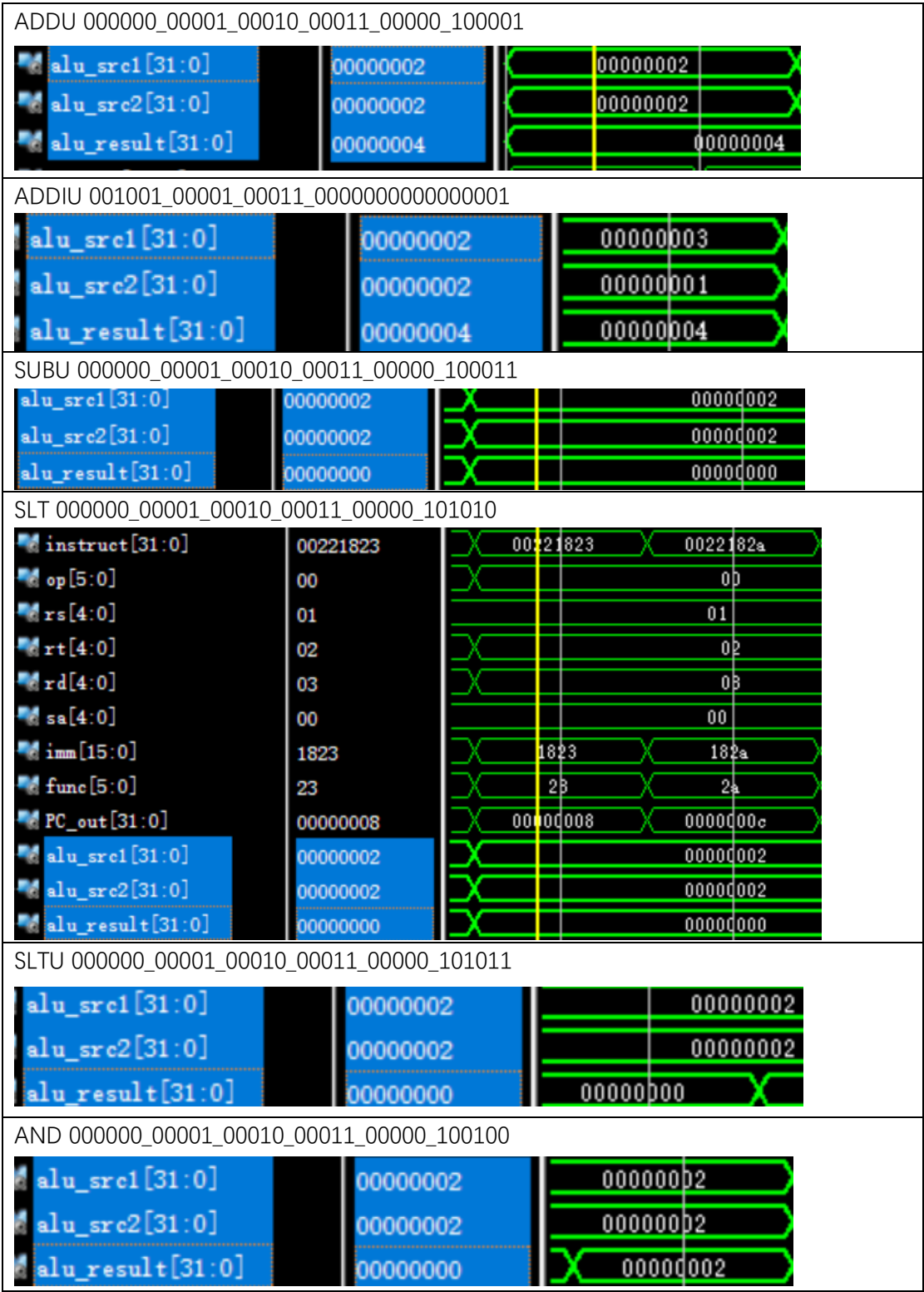
        .CLK(CLK),
        .reset(reset),
        .instruct(instruct),
        .op(op),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .sa(sa),
        .imm(imm),
        .func(func),
        .PC_src(PC_src),
        .PC_out(PC_out),
        .IR_addr(IR_addr),
        .inst_ram_en(inst_ram_en),
        .inst_ram_wen(inst_ram_wen),
        .sel_alu_src2(sel_alu_src2),
        .alu_op(alu_op),
        .data_ram_en(data_ram_en),
        .data_ram_wen(data_ram_wen),
        .rf_we(rf_we),
        .sel_rf_dst(sel_rf_dst),
        .sel_rf_res(sel_rf_res),
        .alu_result(alu_result),
        .alu_src1(alu_src1),
        .alu_src2(alu_src2),
        .imm_ext(imm_ext),
        .rf_waddr(rf_waddr),
        .rf_raddr1(rf_raddr1),
        .rf_raddr2(rf_raddr2),
        .rf_wdata(rf_wdata),
        .rf_rdata1(rf_rdata1),
        .rf_rdata2(rf_rdata2),
        .dr_addr(dr_addr),
        .dr_wdata(dr_wdata),
        .dr_rdata(dr_rdata)
    );
    initial begin
        CLK = 0;
        reset = 1;
        #1 reset = 0;
    end

    always #10 CLK = ~CLK;

```

endmodule

波形



LUI 001111\_00000\_00011\_0000000000000001

alu_src1[31:0]	00000002	X	00000003
alu_src2[31:0]	00000002	X	00000000
alu_result[31:0]	00000000	X	00000000

NOR 000000\_00001\_00010\_00011\_00000\_100111

alu_src1[31:0]	00000002	000	
alu_src2[31:0]	00000002	000	
alu_result[31:0]	fffffffd	000	fffffffd

OR 000000\_00001\_00010\_00011\_00000\_100101

alu_src1[31:0]	00000002		
alu_src2[31:0]	00000002		
alu_result[31:0]	00000002	X	00000002

XOR 000000\_00001\_00010\_00011\_00000\_100110

alu_src1[31:0]	00000002		
alu_src2[31:0]	00000002		
alu_result[31:0]	00000000	X	00000000

SLL 000000\_00000\_00010\_00011\_00001\_000000

instruct[31:0]	00021840	X	00021840	X
sa[4:0]	01	X		01
alu_src2[31:0]	00000002			00000002
alu_result[31:0]	00000004	X	00000004	X

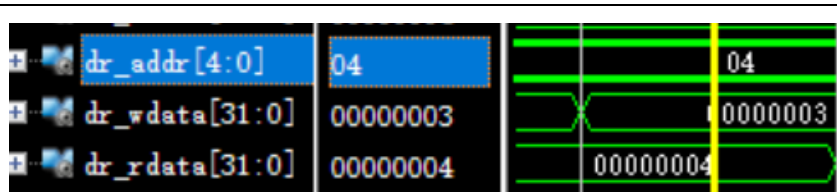
SRA 000000\_00000\_00010\_00011\_00001\_000011

instruct[31:0]	00021843	0002	X	00021843	X
sa[4:0]	01			01	
alu_src2[31:0]	00000002			00000002	X
alu_result[31:0]	00000001	0000	X	00000001	X

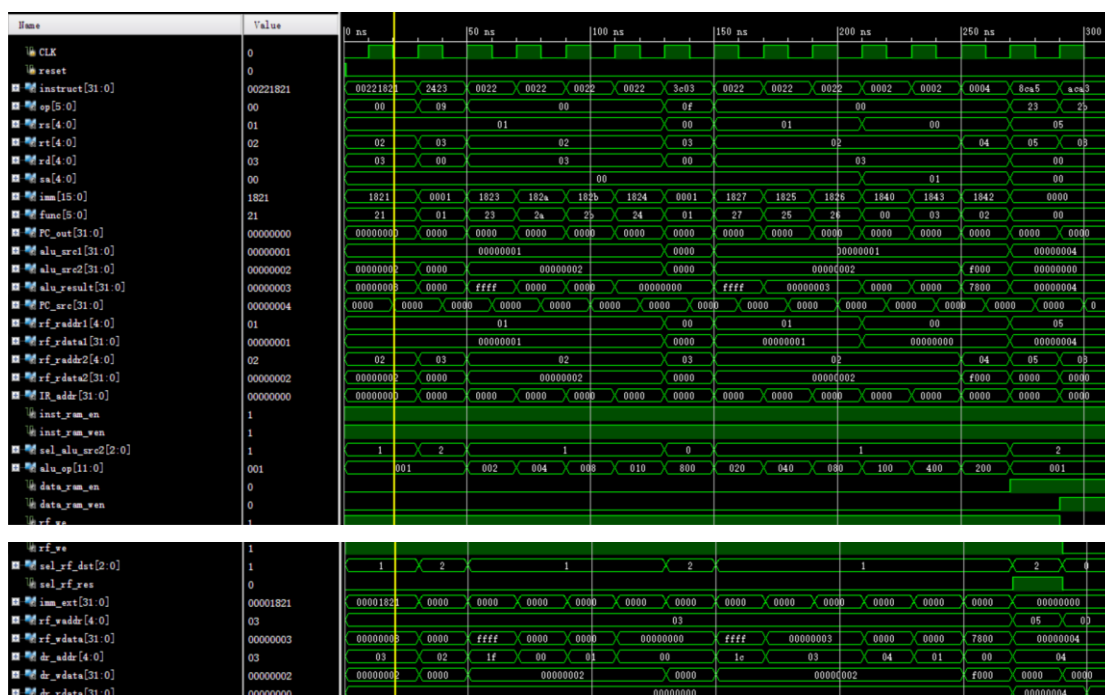
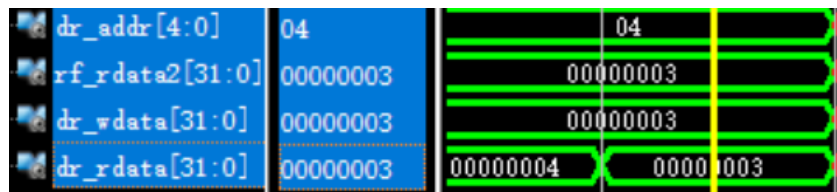
SRL 000000\_00000\_00100\_00011\_00001\_000010

instruct[31:0]	00041842	X	00041842	X
sa[4:0]	01		01	
alu_src2[31:0]	f0000000	X	f0000000	X
alu_result[31:0]	78000000	X	78000000	X

LW 10011\_00101\_00101\_0000000000000000



SW 101011\_00101\_00011\_0000000000000000



注：运行代码时需要修改的地方：

IR.v:第 16 行 readmemb 路径 DR.v:第 15 行 readmemh 路径

RegFile.v:第 18 行 readmemh 路径