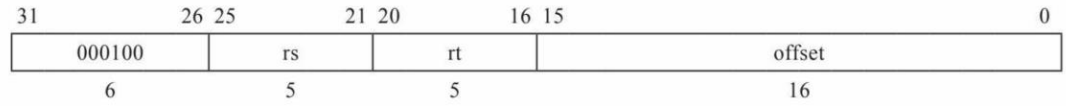
单周期 CPU 更新日志

增加指令

BEQ

格式



汇编格式：BEQ rs,rt,offset

功能描述：如果寄存器rs的值等于寄存器rt的值则转移，否则顺序执行。转移目标由立即数offset左移2位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的PC计算得到。

操作定义：
```
I:   condition ← GPR[rs] = GPR[rt]
     target_offset ← Sign_extend(offset||0^2)
I+1: if condition then
         PC ← PC+ target_offset
     endif
```

注意：等到执行延迟槽指令，即跳转指令的后一条指令时再跳转将 PC+offset

调整

增加 32 位比较器

得出的两数是否相等的中间结果和分支指令是 BEQ 还是 BNE 的情况一起产生分支跳转条件是否成立的最终结果。

PCadd -> nextPC

二选一

if 当前指令不是延迟槽 or 转移指令不跳转

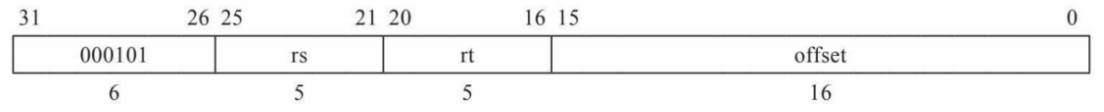
in0->PC+4

if 当前指令转移延迟槽，对应转移指令跳转

in1->PC+4+offset

BNE

3.6.2 BNE



汇编格式：BNE rs,rt,offset

功能描述：如果寄存器rs的值不等于寄存器rt的值则转移，否则顺序执行。转移目标由立即数offset左移2位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的PC计算得到。

JAL

格式

3.6.10 JAL

汇编格式：JAL target

功能描述：无条件跳转。跳转目标由该分支指令对应的延迟槽指令的PC的最高4位与立即数instrindex_左移2位后的值拼接得到。同时将该分支对应延迟槽指令之后的指令的PC值保存至第31号通用寄存器中。

操作定义：I: $GPR[31] \leftarrow PC + 8$

I+1: $PC \leftarrow PC_{31..28} \,||\, instr\_index \,||\, 0^2$

例外：无

调整

nextPC 三选一

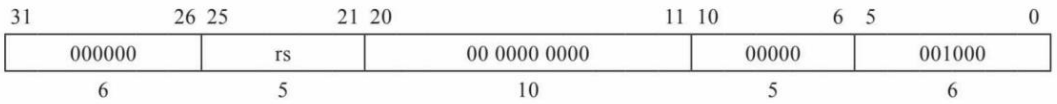
in2 -> PC 高位与 JAL 中 instr_index 左移两位拼接

GenRFDst 二选一 ---> 三选一

in2 -> 31

JR

3.6.11 JR



汇编格式：JR rs

功能描述：无条件跳转。跳转目标为寄存器rs中的值。

操作定义：I:temp←GPR[rs]

I+1:PC←temp

例外：无

调整

nextPC 四选一
in3 -> rdata1

延迟槽的实现

PCadd 会在下降沿将 PC 自增，下降沿正好处在一条指令执行的过程中，而 PCBranch 在 PCadd 更新后会立即得到跳转的地址，这个地址也就是延迟槽指令 PC 和跳转指令共同决定的地址了，当上升沿，PC 允许更新地址的时候，这个延迟槽的地址会进入 PC 里面。

IP 核使用

使用 BlockMemory 替换 InstRam 和 DataRam
IP 核使用方法见另一篇文档 VIVADO 定制同步 RAMIP 核

PCadd 取消时钟

PCadd 不需要时钟输入，直接自增即可。

大小端转换

增加模块

乘法指令

使用 IP 核

按字节编址

需大小端转换

部件实现与修改

IR

```
//block Ram
IR IR (
  .clka(CLK),    // input wire clka
  .wea(1'b0),      // input wire [0 : 0] wea
  .addra(IR_addr),  // input wire [15 : 0] addra
  .dina(32'd0),    // input wire [31 : 0] dina
  .douta(instruct)  // output wire [31 : 0] douta
);
```

DR

```
//block Ram
DR DR (
  .clka(CLK),    // input wire clka
  .ena(data_ram_en),      // input wire ena
  .wea(data_ram_wen),       // input wire [0 : 0] wea
  .addra(dr_addr),  // input wire [15 : 0] addra
  .dina(dr_wdata),    // input wire [31 : 0] dina
  .douta(dr_rdata)  // output wire [31 : 0] douta
```

```
);
```

PCBranch

代码

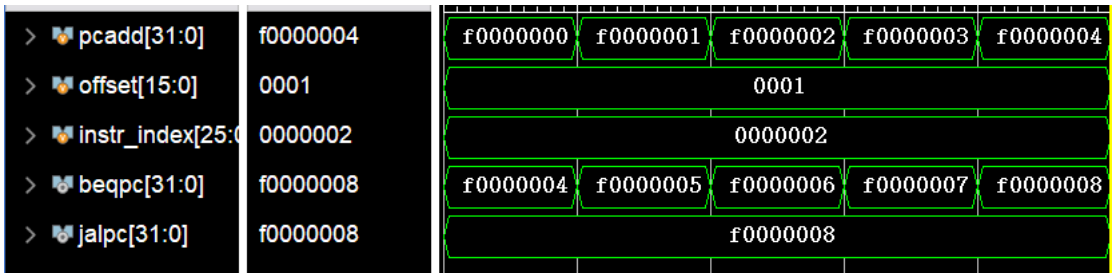
```
//2021-7-12,吴佳宾
//PCBranch
`timescale 1ns/10ps
module PCBranch(
    input       [31:0]  pcadd, //PC+4
    input       [15:0]  offset, //BEQ BNE
    input               beq_bne_jump, //beq bne 的跳转信号
    input       [25:0]  instr_index, //JAL
    output      [31:0]  beqpc, //执行 beq 跳转的 PC
    output      [31:0]  jalpc //执行 jal 跳转的 PC
);
wire [31:0] signleft_offset; //offset 符号扩展  左移两位
assign signleft_offset = {{14{offset[15]}}, offset, 2'b00};
assign beqpc = beq_bne_jump ?   pcadd+signleft_offset : pcadd;
assign jalpc = {pcadd[31], pcadd[30], pcadd[29], pcadd[28], instr_index, 2'b00};

endmodule
```

仿真

```
//--------testbench of nextPC----------
module PCBranch_tb();
reg     [31:0] pcadd;
reg     [15:0] offset;
reg     [25:0] instr_index;
reg            beq_bne_jump;
wire    [31:0] beqpc;
wire    [31:0] jalpc;

PCBranch PCBranch(
    .pcadd(pcadd),
    .offset(offset),
    .beq_bne_jump(beq_bne_jump),
    .instr_index(instr_index),
```

```
        .beqpc(beqpc),
        .jalpc(jalpc)
);

initial begin
        pcadd = 32'hf000_0000;
        beq_bne_jump = 1'b1;
        offset = 16'd1;
        instr_index = 26'd2;
        #50 $stop;
end

always #10 pcadd = pcadd + 1;

endmodule
```

波形



Instruct_Split

代码

```
//2021-7-11,吴佳宾
//指令分割
`timescale 1ns/10ps
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
    output    [25:0]        instr_index //J-Type
    );

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];
assign instr_index = instruct[25:0];

endmodule
```

GenBeqBne 分支判断比较逻辑

代码

```
//2021-7-11, 吴佳宾
//BEQBNE 分支判断比较逻辑 GenBeqBne
`timescale 1ns/10ps
module GenBeqBne(
    input     [31:0]     rdata1,
    input     [31:0]     rdata2,
    input     [ 2:0]     sel_beqbne, //sel[0]表示 beq sel[1]表示 bne
    output               beq_bne_jump //跳转信号
);
wire equal, beq, bne;
assign equal = (rdata1 == rdata2);
assign beq_bne_jump = (sel_beqbne[0] & equal) | (sel_beqbne[1] & !equal);

endmodule
```

GenNextPC

代码

```
//2021-7-11, 吴佳宾
//PC 输入生成逻辑 GenNextPC
`timescale 1ns/10ps
module GenNextPC(
    input   [31:0]   in0, //PC + 4
    input   [31:0]   in1, //PC + offset
    input   [31:0]   in2, //PC link instr_index
    input   [31:0]   in3, //rdata1
    input   [ 3:0]   sel_nextpc, //控制信号 独热码
    output  [31:0]   PCsrc //确定的 PC 值
);

assign nextpc = (sel_nextpc[0]) ? in0 :
                (sel_nextpc[1]) ? in1 :
                (sel_nextpc[2]) ? in2 :
                (sel_nextpc[3]) ? in3 :
                32'd0;

endmodule
```

仿真

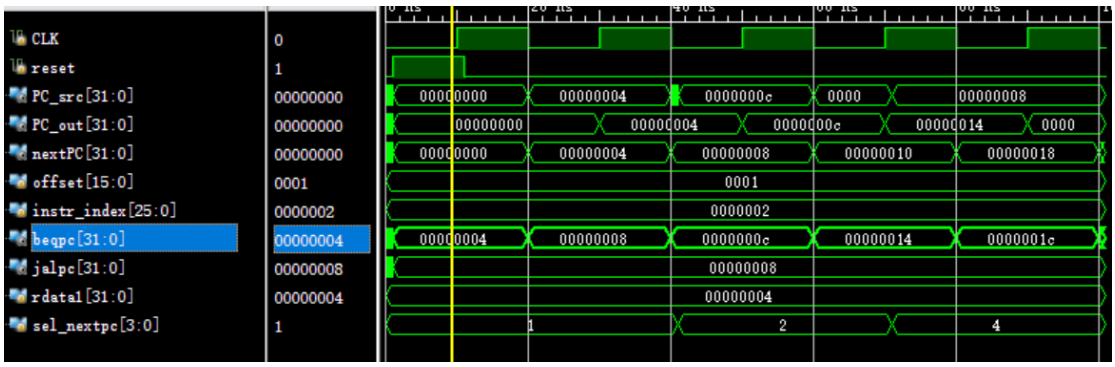
```
//2021-7-12 吴佳宾
//---------testbench of PC/PCadd/PCBranch/GenNextPC------------
`timescale 1ns/10ps
module PC_tb();
//PC
reg CLK, reset;
wire  [31:0]   PC_src, PC_out;
//PCadd
wire  [31:0]   nextPC;
//PCBranch
reg   [15:0]   offset;
reg   [25:0]   instr_index;
wire  [31:0]   beqpc;
wire  [31:0]   jalpc;
//GenNextPC
reg   [ 3:0]   sel_nextpc;
```

```
reg     [31:0]     rdata1;

PC PC(
        .CLK(CLK),
        .reset(reset),
        .PC_src(PC_src),
        .PC_out(PC_out)
);
PCadd PCadd(
        .CLK(CLK),
        .reset(reset),
        .oldPC(PC_out),
        .newPC(nextPC)
);
PCBranch PCBranch(
        .pcadd(nextPC),
        .offset(offset),
        .beq_bne_jump(1'b1),
        .instr_index(instr_index),
        .beqpc(beqpc),
        .jalpc(jalpc)
);
GenNextPC GenNextPC(
        .in0(nextPC),
        .in1(beqpc),
        .in2(jalpc),
        .in3(rdata1),
        .sel_nextpc(sel_nextpc),
        .PCsrc(PC_src)
);

initial begin
        CLK = 0;
        reset = 0;
        offset = 32'd1;
        instr_index = 32'd2;
        rdata1 = 32'd4;
        sel_nextpc = 4'b0001;
        #1 reset = 1;
        #10 reset = 0;
        #30 sel_nextpc = 4'b0010;
        #30 sel_nextpc = 4'b0100;
        #30 sel_nextpc = 4'b1000;
        $stop;
```

end

always #10 CLK = ~CLK;

endmodule
波形



ControlUnit

代码

```
//2021-7-12，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
    output   [ 3:0]    sel_nextpc, //pc 输入生成逻辑
    output             sel_beqbne //选择 BEQ BNE
);
/*
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;
wire inst_beq;
wire inst_bne;
wire inst_jal;
wire inst_jr;
//指令译码
assign inst_addu = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100001);
assign inst_addiu = (op == 6'b001001);
assign inst_subu = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b100011);
assign inst_slt = (op == 6'b000000) && (sa == 5'b00000) && (func == 6'b101010);
assign inst_sltu = (op == 6'b000000) && (sa == 5'b00000) && (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);
assign inst_beq = (op == 6'b000100);
assign inst_bne = (op == 6'b000101);
assign inst_jal = (op == 6'b000011);
assign inst_jr = (op == 6'b000000) && (func == 6'b001000);
//控制信号生成
```

```
assign inst_ram_en = 1;
assign inst_ram_wen = 1;
assign sel_nextpc = {inst_jr, inst_jal, inst_bne | inst_beq,
                                inst_addiu | inst_addiu | inst_subu | inst_lw | inst_sw |
                                inst_slt | inst_sltu | inst_sll | inst_srl | inst_sra | inst_lui |
                                inst_and | inst_or | inst_xor | inst_nor};
assign sel_alu_src1 = {inst_sll | inst_srl | inst_sra,
                                  inst_jal,
                                  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 = {inst_jal,
                                  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 | inst_jal};
assign sel_beqbne = {inst_bne, inst_beq};

endmodule
```

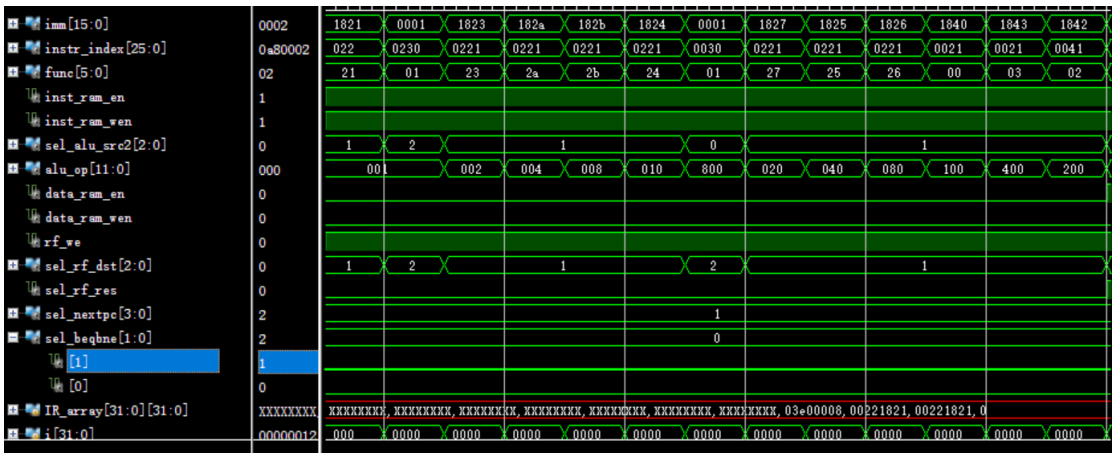
仿真

```
`timescale 1ns/10ps
//2021-7-12,吴佳宾
//------testbench of ControlUnit-----
module ControlUnit_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    [25:0] instr_index;
wire    [ 5:0] func;
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    [ 3:0] sel_nextpc;
wire    [ 1:0] sel_beqbne;
Instruct_Split Instruct_Split(
        .instruct(instruct),
        .op(op),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .sa(sa),
        .imm(imm),
        .func(func),
        .instr_index(instr_index)
);
ControlUnit ControlUnit(
        .op(op),
        .sa(sa),
        .func(func),
        .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),
        .sel_beqbne(sel_beqbne),
        .sel_nextpc(sel_nextpc)
);
reg [31:0] IR_array[31:0];
integer i = 0;
initial begin
```

```
        $readmemb("D:/Codes/Vivado/SingleCPU/SingleCPU.srcs/sources_1/new/IR.txt",
IR_array);
end

initial begin
        #300 $stop;
end
always #10 begin
        instruct <= IR_array[i];
        i = i + 1;
end

endmodule
```

波形



顶层文件 top

代码

```
`timescale 1ns/10ps
//2021-7-12,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
output     [25:0]     instr_index, //J-Type
//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
output     [ 1:0]     sel_beqbne, //BEQ BNE 选择信号
//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
//跳转信号
output                beq_bne_jump
```

```
);
//PC 跳转方式用
wire    [31:0] nextpc, beqpc, jalpc;

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(nextpc)
);
PCBranch PCBranch(
        .pcadd(nextpc),
        .offset(imm),
        .beq_bne_jump(beq_bne_jump),
        .instr_index(instr_index),
        .beqpc(beqpc),
        .jalpc(jalpc)
);
GenNextPC GenNextPC(
        .in0(nextpc),
        .in1(beqpc),
        .in2(jalpc),
        .in3(rdata1),
        .sel_nextpc(sel_nextpc),
        .PCsrc(PC_src)
);
GenBeqBne GenBeqBne(
        .rdata1(rdata1),
        .rdata2(rdata2),
        .sel_beqbne(sel_beqbne),
        .beq_bne_jump(beq_bne_jump)
);
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),
        .instr_index(instr_index)
);
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),
        .sel_beqbne(sel_beqbne)
);
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
```

测试验证

具体指令

BEQ
000100_00110_00111_0000000000000001
BNE
000101_00110_00101_0000000000000010
000101_00110_00111_0000000000000100
JAL
000011_0000_0000_0000_0000_0000_011000
JR
000000_11111_0000000000_00000_001000

仿真

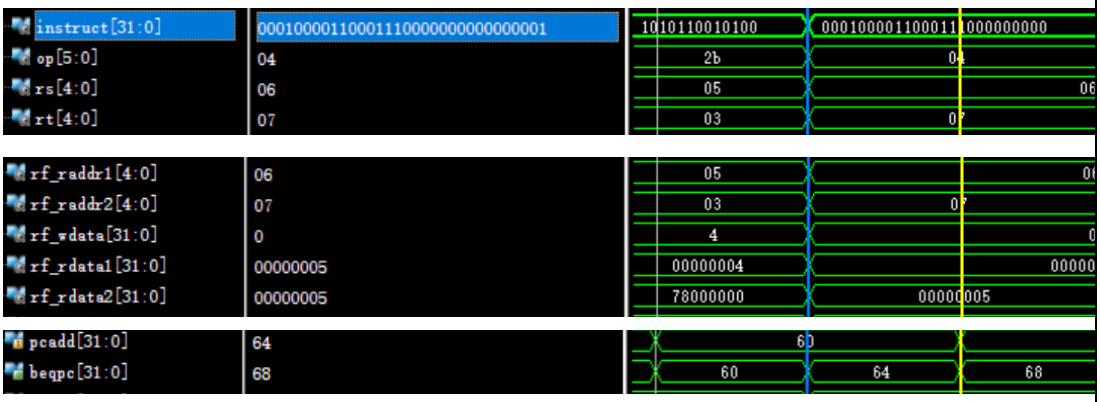
```
`timescale 1ns/10ps
//2021-7-12  吴佳宾
//Single_CPU 顶层测试文件
//----------testbench of top----------
module top_tb();
reg                 CLK; //时钟信号
reg                 reset; //复位信号
wire    [31:0]    instruct; //指令
wire    [ 5:0]    op; //操作码
wire    [ 4:0]    rs; //rs 域
```

```
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     [25:0]     instr_index; //J-Type 用
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
wire     [ 1:0]     sel_beqbne; //beq bne 选择信号
wire                beq_bne_jump; //跳转信号
wire     [ 3:0]     sel_nextpc; //PC 输入生成信号


top top(
    .CLK(CLK),
    .reset(reset),
    .instruct(instruct),
    .op(op),
    .rs(rs),
    .rt(rt),
    .rd(rd),
```

```
        .sa(sa),
        .imm(imm),
        .func(func),
        .instr_index(instr_index),
        .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),
        .sel_beqbne(sel_beqbne),
        .beq_bne_jump(beq_bne_jump),
        .sel_nextpc(sel_nextpc)
);
initial begin
        CLK = 0;
        reset = 1;
        #1 reset = 0;
        #310 $stop;
end

always #10 CLK = ~CLK;

endmodule
```

波形

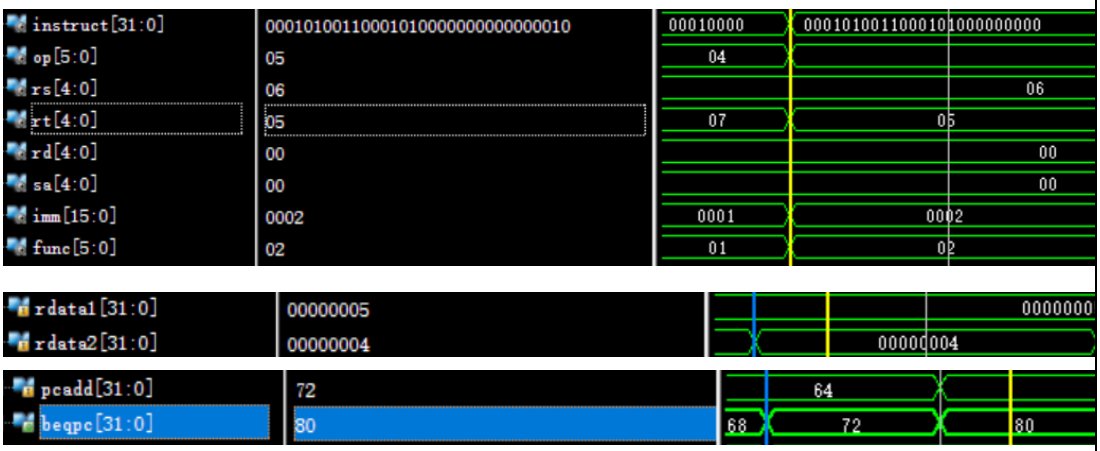
BEQ



下一条指令地址

跳转成功

BNE



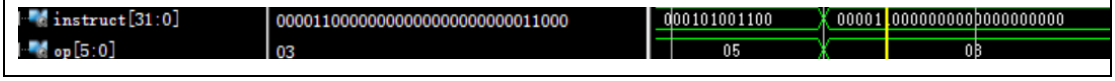
跳转轨迹

跳转成功

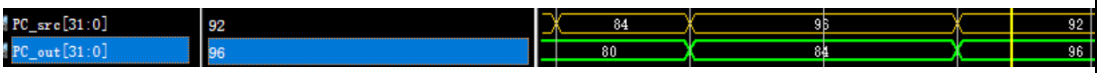
BNE 不跳转情况

BEQPC 和 PCADD 一致

JAL

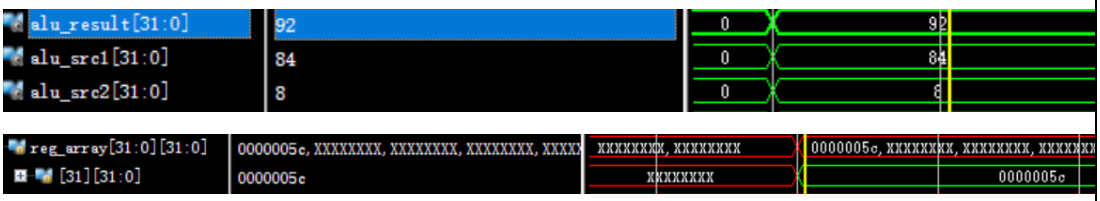


跳转轨迹



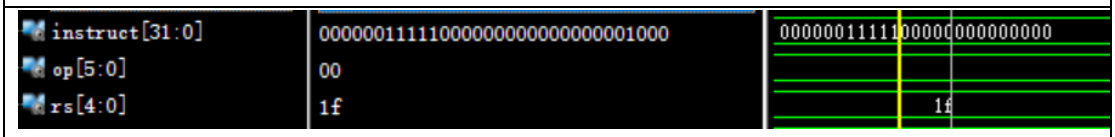
跳转成功

保存延迟槽 PC+4 到 31 号寄存器中



保存成功

JR



取刚才保存到 31 号寄存器的地址当作下一条指令的地址

读取成功

跳转轨迹

跳转成功

问题

1. RegFile 写时序改成了下降沿写入，因为在执行 JAL 的时候发现无法写入 JALPC，但目前还不知道这样的修改会不会出现新问题，有待考察。