Lab6 Report

实验原理

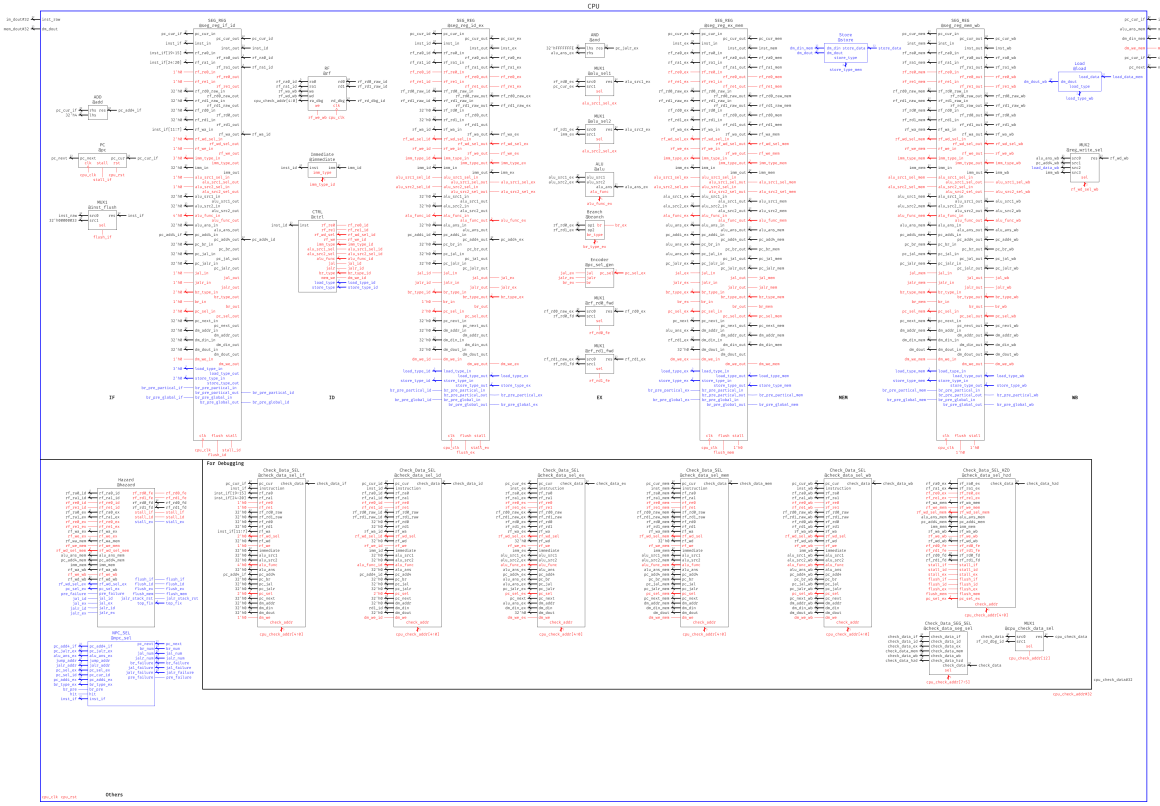
实验内容

1. 必做：RV32-I 指令子集扩展，在流水线CPU中实现所有非系统 RV32-I 指令(共 27 条)
2. 选做：2bits 感知机局部历史分支预测；2bits 感知机全局历史分支预测

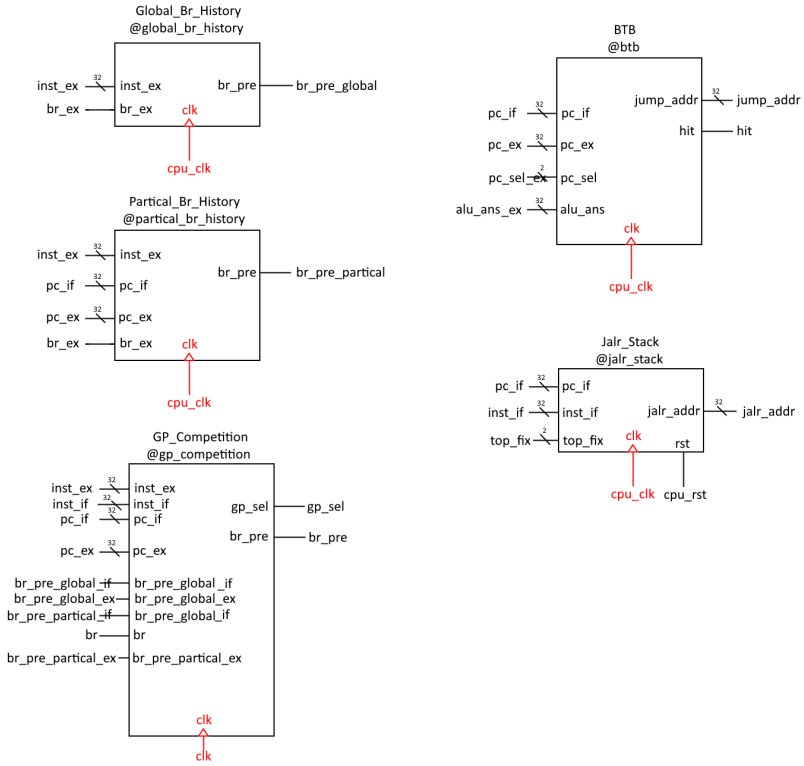
设计流程

数据通路

除分支预测模块之外的数据通路与Lab5的必做部分并无太多不同，图中蓝色部分为改动的部分，包含`L-Type` `S-Type`指令的实现和分支预测的部分实现



分支预测关键模块的数据通路



关键模块设计

CTRL模块

发送不同指令对应的不同控制信号，具体含义在其他模块中解释

```
module CTRL (
    input [31:0] inst,
    output reg jal,
    output reg jalr,
    output reg [2:0] br_type,
    output reg alu_src1_sel,
    output reg alu_src2_sel,
    output reg [3:0] alu_func,
    output reg mem_we,
    output reg [2:0] imm_type,
    output reg rf_re0,
    output reg rf_re1,
    output reg [1:0] rf_wd_sel,
    output reg rf_we,
    output reg [2:0] load_type,
    output reg [1:0] store_type
);
    always @(*) begin
        jal=0; jalr=0; br_type=0;
        alu_src1_sel=0; alu_src2_sel=0;
        alu_func=0; mem_we=0; imm_type=0; load_type=0; store_type=0;
        rf_re0=0;rf_re1=0;rf_wd_sel=0;rf_we=0;
        case (inst[6:0])
            7'b0010011:begin
                alu_src1_sel=0;
                alu_src2_sel=1;
                imm_type=1;
                rf_wd_sel=0;
```

```
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                case (inst[14:12])
                    3'b000: alu_func=4'b0000;   //addi
                    3'b001: alu_func=4'b1001;   //slli
                    3'b010: alu_func=4'b0100;   //slti
                    3'b011: alu_func=4'b0011;   //sltiu
                    3'b100: alu_func=4'b0111;   //xori
                    3'b101:begin
                        case (inst[31:25])
                            7'b0000000: alu_func=4'b1000;   //srli
                            7'b0100000: alu_func=4'b1010;   //srai
                        endcase
                    end
                    3'b110: alu_func=4'b0110;   //ori
                    3'b111: alu_func=4'b0101;   //andi
                endcase
            end
            7'b0110011:begin
                alu_src1_sel=0;
                alu_src2_sel=0;
                rf_wd_sel=0;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                rf_re1= (inst[24:20]==5'b00000) ? 0 : 1;
                case (inst[14:12])
                    3'b000:begin
                        case (inst[31:25])
                            7'b0000000: alu_func=4'b0000;   //add
                            7'b0100000: alu_func=4'b0001;   //sub
                        endcase
                    end
                    3'b001: alu_func=4'b1001;   //sll
                    3'b010: alu_func=4'b0100;   //slt
                    3'b011: alu_func=4'b0011;   //sltu
                    3'b100: alu_func=4'b0111;   //xor
                    3'b101:begin
                        case (inst[31:25])
                            7'b0000000: alu_func=4'b1000;   //srl
                            7'b0100000: alu_func=4'b1010;   //sra
                        endcase
                    end
                    3'b110: alu_func=4'b0110;   //or
                    3'b111: alu_func=4'b0101;   //and

                endcase
            end
            7'b0110111:begin
                alu_src2_sel=1;
                imm_type=4;
                rf_wd_sel=3;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                alu_func=4'b0000;   //lui
            end
            7'b0010111:begin
```

```
                alu_src1_sel=1;
                alu_src2_sel=1;
                imm_type=4;
                rf_wd_sel=0;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                alu_func=4'b0000;   //auipc
            end
            7'b1101111:begin
                jal=1;
                alu_src1_sel=1;
                alu_src2_sel=1;
                imm_type=5;
                rf_wd_sel=1;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                alu_func=4'b0000;   //jal
            end
            7'b1100111:begin
                jalr=1;
                alu_src1_sel=0;
                alu_src2_sel=1;
                imm_type=1;
                rf_wd_sel=1;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                alu_func=4'b0000;   //jalr
            end
            7'b1100011:begin
                alu_src1_sel=1;
                alu_src2_sel=1;
                imm_type=3;
                alu_func=4'b0000;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                rf_re1= (inst[24:20]==5'b00000) ? 0 : 1;
                case (inst[14:12])
                    3'b000: br_type=1;  //beq
                    3'b001: br_type=2;  //bne
                    3'b100: br_type=3;  //blt
                    3'b101: br_type=4;  //bge
                    3'b110: br_type=5;  //bltu
                    3'b111: br_type=6;  //bgeu
                endcase
            end
            7'b0000011:begin
                alu_src1_sel=0;
                alu_src2_sel=1;
                imm_type=1;
                alu_func=4'b0000;
                rf_wd_sel=2;
                rf_we= (inst[11:7]==5'b00000) ? 0 : 1;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                case (inst[14:12])
                    3'b000: load_type=1;    //lb
                    3'b001: load_type=2;    //lh
                    3'b010: load_type=3;    //lw
                    3'b100: load_type=4;    //lbu
```

```
                    3'b101: load_type=5;    //lhu
                endcase
            end
            7'b0100011:begin
                alu_src1_sel=0;
                alu_src2_sel=1;
                imm_type=2;
                alu_func=4'b0000;
                rf_re0= (inst[19:15]==5'b00000) ? 0 : 1;
                rf_re1= (inst[24:20]==5'b00000) ? 0 : 1;
                mem_we=1;
                case (inst[14:12])
                    3'b000: store_type=1;   //sb
                    3'b001: store_type=2;   //sh
                    3'b010: store_type=3;   //sw
                endcase
            end
        endcase
    end
endmodule
```

Load模块

`lb` `lh`指令使用符号位扩展，`lbu` `lhu`指令使用`0`扩展

```
module Load (
    input [31:0] dm_dout,
    input [2:0] load_type,
    output reg [31:0] load_data
);
    always @(*) begin
        case (load_type)
            3'h1: load_data={{24{dm_dout[31]}}, dm_dout[31:24]};    //lb
            3'h2: load_data={{16{dm_dout[31]}}, dm_dout[31:16]};    //lh
            3'h3: load_data=dm_dout;                                //lw
            3'h4: load_data={24'h0, dm_dout[31:24]};                //lbu
            3'h5: load_data={16'h0, dm_dout[31:16]};                //lhu
            default: load_data=32'h0;
        endcase
    end
endmodule
```

Store模块

遇到`sb` `sh`指令时，同步将数据存储器中对应位置的数据读取出来，通过位拼接，将要写入的值和原值拼接，然后写入数据存储器中，如此不需要改变数据存储器的结构

```
module Store (
    input [31:0] dm_din,
    input [31:0] dm_dout,
    input [1:0] store_type,
    output reg [31:0] store_data
);
    always @(*) begin
        case (store_type)
```

```
            2'h1: store_data={dm_din[7:0], dm_dout[23:0]};  //sb
            2'h2: store_data={dm_din[15:0], dm_dout[15:0]}; //sh
            2'h3: store_data=dm_din;                        //sw
            default: store_data=0;
        endcase
    end
endmodule
```

其他指令大多只需要改变ALU模块模式信号，或Branch模块模式信号等等，不再赘述

2bits 感知机局部历史分支预测

接入流水线的模块为`Partical_Br_History`。在该模块中，`BHT`模块通过`pc_if`的哈希，找到对应的局部历史寄存器；`PHT_Partical`模块通过刚刚取得的局部历史，找到对应的 2bits 寄存器（感知机），读取此时的感知机状态，并根据此输出是否跳转的预测信号

除此之外，当`ex`阶段的指令为分支指令时，`BHT`模块通过`pc_ex`的哈希，找到对应的局部历史寄存器；`PHT_Partical`模块通过刚刚取得的局部历史，找到对应的 2bits 寄存器（感知机），并根据`ex`阶段计算出的`br`信号修改感知机状态值。

当`if``ex`阶段的指令为同一条分支指令时，通过写优先读出`ex`阶段写入的值

```
module BHT (
    input clk,
    input [31:0] pc_if,
    input [31:0] pc_ex,
    input br_inst,  //ex阶段是否是br指令
    input br,
    output [3:0] br_history_if,
    output [3:0] br_history_ex
);
    reg [3:0] bht [0:63];
    wire [5:0] pc_if_hash, pc_ex_hash;
    integer i=0;

    assign pc_if_hash=pc_if[15:10]^pc_if[7:2];
    assign pc_ex_hash=pc_ex[15:10]^pc_ex[7:2];
    assign br_history_if= (br_inst && pc_if==pc_ex) ? {bht[pc_ex_hash][2:0],br} : bht[pc_if_hash];    //写优先
    assign br_history_ex= (br_inst) ? {bht[pc_ex_hash][2:0],br} : bht[pc_ex_hash];  //写优先

    initial begin
        for (i=0; i<64; i=i+1) begin
            bht[i]=4'h0;    //所有BHR初值为零
        end
    end

    always @(posedge clk) begin
        if(br_inst)
            bht[pc_ex_hash]<={bht[pc_ex_hash][2:0],br};  //移位寄存器
    end
endmodule

module PHT_Partical (
```

```
    input clk,
    input [31:0] pc_if,
    input [31:0] pc_ex,
    input br_inst,  //是否是br指令
    input br,
    input [3:0] br_history_if,
    input [3:0] br_history_ex,
    output reg br_pre
);
    reg [1:0] pht [0:63][0:15];
    wire [5:0] pc_if_hash, pc_ex_hash;
    integer i=0, j=0;

    assign pc_if_hash=pc_if[15:10]^pc_if[7:2];
    assign pc_ex_hash=pc_ex[15:10]^pc_ex[7:2];

    always @(*) begin
        if(br_inst && pc_if==pc_ex)begin    //写优先
            case (pht[pc_ex_hash][br_history_ex])
                2'b00: br_pre=0;
                2'b01: br_pre= (br) ? 1 : 0;
                2'b10: br_pre= (br) ? 1 : 0;
                2'b11: br_pre=1;
            endcase
        end
        else begin
            case (pht[pc_if_hash][br_history_if])
                2'b00: br_pre=0;
                2'b01: br_pre=0;
                2'b10: br_pre=1;
                2'b11: br_pre=1;
            endcase
        end
    end

    initial begin
        for (i=0; i<64; i=i+1) begin
            for (j=0; j<16; j=j+1) begin
                pht[i][j]=2'b01;    //弱不跳转
            end
        end
    end

    always @(posedge clk) begin
        if (br_inst) begin
            case (pht[pc_ex_hash][br_history_ex])
                2'b00: pht[pc_ex_hash][br_history_ex]<= (br) ? 2'b01 :
2'b00;
                2'b01: pht[pc_ex_hash][br_history_ex]<= (br) ? 2'b10 :
2'b00;
                2'b10: pht[pc_ex_hash][br_history_ex]<= (br) ? 2'b11 :
2'b01;
                2'b11: pht[pc_ex_hash][br_history_ex]<= (br) ? 2'b11 :
2'b10;
            endcase
```

```
        end
    end
endmodule

module Partical_Br_History (
    input clk,
    input [31:0] pc_if,
    input [31:0] pc_ex,
    input [31:0] inst_ex,
    input br_ex,
    output br_pre
);
    wire br_inst;
    wire [3:0] br_history_if;
    wire [3:0] br_history_ex;
    assign br_inst= (inst_ex[6:0]==7'b1100011) ? 1 : 0;

    BHT bht(
        .clk(clk),
        .pc_if(pc_if),
        .pc_ex(pc_ex),
        .br_inst(br_inst),
        .br(br_ex),
        .br_history_if(br_history_if),
        .br_history_ex(br_history_ex)
    );

    PHT_Partical pht_partical(
        .clk(clk),
        .pc_if(pc_if),
        .pc_ex(pc_ex),
        .br_inst(br_inst),
        .br(br_ex),
        .br_history_if(br_history_if),
        .br_history_ex(br_history_ex),
        .br_pre(br_pre)
    );
endmodule
```

2bits 感知机全局历史分支预测

与局部历史分支预测原理类似。接入流水线的模块为`Global_Br_History`。在该模块中，`GHR`模块存储当前的全局分支指令跳转历史；`PHT_Global`模块通过刚刚取得的全局历史，找到对应的 2bits 寄存器（感知机），读取此时的感知机状态，并根据此输出是否跳转的预测信号

除此之外，当`ex`阶段的指令为分支指令时，`GHR`模块修改当前的全局分支指令跳转历史；`PHT_Global`模块通过刚刚取得的全局历史，找到对应的 2bits 寄存器（感知机），并根据`ex`阶段计算出的`br`信号修改感知机状态值。

当`if` `ex`阶段的指令为同一条分支指令时，通过写优先读出`ex`阶段写入的值

```
module GHR (
    input clk,
    input br_inst,  //ex阶段是否是br指令
    input br,
```

```
    output [7:0] br_history
);
    reg [7:0] ghr;

    assign br_history= (br_inst) ? {ghr[6:0],br} : ghr; //写优先

    initial begin
        ghr=8'h0;
    end

    always @(posedge clk) begin
        if(br_inst)
            ghr<={ghr[6:0],br}; //移位寄存器
    end
endmodule

module PHT_Global (
    input clk,
    input br_inst,  //是否是br指令
    input br,
    input [7:0] br_history,
    output reg br_pre
);
    reg [1:0] pht [0:255];
    integer i=0;

    always @(*) begin
        if(br_inst)begin    //写优先
            case (pht[br_history])
                2'b00: br_pre=0;
                2'b01: br_pre= (br) ? 1 : 0;
                2'b10: br_pre= (br) ? 1 : 0;
                2'b11: br_pre=1;
            endcase
        end
        else begin
            case (pht[br_history])
                2'b00: br_pre=0;
                2'b01: br_pre=0;
                2'b10: br_pre=1;
                2'b11: br_pre=1;
            endcase
        end
    end

    initial begin
        for (i=0; i<256; i=i+1) begin
                pht[i]=2'b01;    //弱不跳转
        end
    end

    always @(posedge clk) begin
        if (br_inst) begin
            case (pht[br_history])
                2'b00: pht[br_history]<= (br) ? 2'b01 : 2'b00; //强不跳转
```

```
                2'b01: pht[br_history]<= (br) ? 2'b10 : 2'b00; //弱不跳转
                2'b10: pht[br_history]<= (br) ? 2'b11 : 2'b01; //弱跳转
                2'b11: pht[br_history]<= (br) ? 2'b11 : 2'b10; //强跳转
            endcase
        end
    end
endmodule

module Global_Br_History (
    input clk,
    input [31:0] inst_ex,
    input br_ex,
    output br_pre
);
    wire br_inst;
    wire [7:0] br_history;
    assign br_inst= (inst_ex[6:0]==7'b1100011) ? 1 : 0;

    GHR ghr(
        .clk(clk),
        .br_inst(br_inst),
        .br(br_ex),
        .br_history(br_history)
    );

    PHT_Global pht_global(
        .clk(clk),
        .br_inst(br_inst),
        .br(br_ex),
        .br_history(br_history),
        .br_pre(br_pre)
    );
endmodule
```

全局历史预测与局部历史预测的竞争

在该模块中，根据 `pc_if` 的哈希找到对应的 2bits 寄存器（感知机），根据其中的值，决定选择全局历史预测还是局部历史预测

除此之外，当 `ex` 阶段的指令为分支指令时，该模块根据 `pc_ex` 的哈希找到对应的 2bits 寄存器（感知机），并根据 `ex` 阶段计算出的 `br` 信号修改感知机状态值。当全局预测正确时，修改 2bits 寄存器向全局方向转化；当全局预测错误时，修改 2bits 寄存器向局部方向转化

```
module GP_Competition (
    input clk,
    input [31:0] pc_if,
    input [31:0] pc_ex,
    input [31:0] inst_ex,
    input [31:0] inst_if,
    input br_pre_global_if,
    input br_pre_partical_if,
    input br_pre_global_ex,
    input br_pre_partical_ex,
    input br,
    output reg gp_sel, //使用全局(1)/分支(0)历史预测
```

```
    output br_pre
);
    wire br_inst, gp_sel_ex;
    wire [5:0] pc_if_hash, pc_ex_hash;
    reg [1:0] gpht[0:63];
    integer i=0;

    assign br_inst= (inst_ex[6:0]==7'b1100011) ? 1 : 0;
    assign pc_if_hash=pc_if[15:10]^pc_if[7:2];
    assign pc_ex_hash=pc_ex[15:10]^pc_ex[7:2];
    assign gp_sel_ex= (br==br_pre_global_ex) ? 1 : 0;   //全局历史预测优先，全局正确时认为是全局
    assign br_pre= (inst_if[6:0]!=7'b1100011) ? 0 :
        (gp_sel) ? br_pre_global_if : br_pre_partical_if;

    always @(*) begin
        if(br_inst && pc_if==pc_ex)begin    //写优先
            case (gpht[pc_ex_hash])
                2'b00: gp_sel=0;
                2'b01: gp_sel= (gp_sel_ex) ? 1 : 0;
                2'b10: gp_sel= (gp_sel_ex) ? 1 : 0;
                2'b11: gp_sel=1;
            endcase
        end
        case (gpht[pc_if_hash])
            2'b00: gp_sel=0;
            2'b01: gp_sel=0;
            2'b10: gp_sel=1;
            2'b11: gp_sel=1;
        endcase
    end

    initial begin
        for (i=0; i<256; i=i+1) begin
            gpht[i]=2'h0;    //所有GBHR初值为零
        end
    end

    always @(posedge clk) begin
        if(br_inst)
            case (gpht[pc_ex_hash])
                2'b00: gpht[pc_ex_hash]<= (gp_sel_ex) ? 2'b01 : 2'b00; //强局部
                2'b01: gpht[pc_ex_hash]<= (gp_sel_ex) ? 2'b10 : 2'b00; //弱局部
                2'b10: gpht[pc_ex_hash]<= (gp_sel_ex) ? 2'b11 : 2'b01; //弱全局
                2'b11: gpht[pc_ex_hash]<= (gp_sel_ex) ? 2'b11 : 2'b10; //强全局
            endcase
    end

endmodule
```

跳转地址预测

在该模块中，根据`pc_if`的哈希找到对应的`jat`寄存器，若对应的`tag`寄存器中的值当前`pc_if`的[5:2]位相等时，认为命中，使用对应`jat`寄存器中的值为预测的地址

除此之外，当`ex`阶段的指令为分支指令或跳转指令时，该模块根据`pc_ex`的哈希找到对应的`jat`寄存器，并根据`ex`阶段计算出的`alu_ans`信号修改对应的`jat`寄存器的值和对应的`tag`寄存器的值

```
module BTB (
    input clk,
    input [31:0] pc_if,
    input [31:0] pc_ex,
    input [1:0] pc_sel,
    input [31:0] alu_ans,
    output reg hit,
    output reg [31:0] jump_addr
);
    wire [5:0] pc_if_hash, pc_ex_hash;
    wire [3:0] tag_if, tag_ex;
    reg [3:0] tag [0:63];
    reg [31:0] jat [0:63];  //jump addr table
    integer i=0;

    assign tag_if=pc_if[5:2];
    assign tag_ex=pc_ex[5:2];
    assign pc_if_hash=pc_if[15:10]^pc_if[7:2];
    assign pc_ex_hash=pc_ex[15:10]^pc_ex[7:2];

    initial begin
        for(i=0; i<64; i=i+1)begin
            tag[i]=4'h0;
            jat[i]=32'h0;
        end
    end

    always @(*) begin
        if(pc_if==pc_ex)begin
            hit=1;
            jump_addr=alu_ans;
        end
        else begin
            if(tag_if==tag[pc_if_hash])begin
                hit=1;
                jump_addr=jat[pc_if_hash];
            end
            else begin
                hit=0;
                jump_addr=0;
            end
        end

    end
    always @(posedge clk) begin
        if(pc_sel==2 || pc_sel==3)begin //jal, br
            if(tag[pc_ex_hash]!=tag_ex)begin
```

```
                jat[pc_ex_hash]<=alu_ans;
                tag[pc_ex_hash]<=tag_ex;
            end
        end
    end
endmodule
```

`jalr`指令地址栈

当`if`阶段遇到`jal`指令时，存储当前`pc_if + 4`到栈中

当`if`阶段遇到`jalr`指令时，出栈，并将出栈的值作为预测的跳转地址

```
module Jalr_Stack (
    input clk,
    input rst,
    input [31:0] pc_if,
    input [31:0] inst_if,
    input [1:0] top_fix,
    output reg [31:0] jalr_addr
);
    reg [31:0] jas[0:63];  //jalr addr stack
    reg [5:0] top;

    integer i=0;

    initial begin
        top=1;
        for(i=0; i<64; i=i+1)begin
            jas[i]=0;
        end
    end

    always @(*) begin
        jalr_addr=jas[top-1];
    end

    always @(posedge clk) begin
        if(rst)begin
            top<=1;
        end
        else begin
            top<=top+top_fix;
            if(inst_if[6:0]==7'b1100111)begin //jalr, 出栈
                top<=top+top_fix-1;
            end
            if(inst_if[6:0]==7'b1101111)begin //jal, 入栈
                jas[top]=pc_if+4;
                top<=top+top_fix+1;
            end
        end
    end
endmodule
```

Hazard模块的修改

预测失败时，插入气泡，冲刷`id` `ex`阶段

若被冲刷的指令中包含`jal` `jalr`指令，则需要将被改变的 jalr 栈指针修改回去，修改的值即为`top_fix`

```
if(pre_failure)begin
    case (pc_sel_ex)
        2'h0: begin //ex阶段不应该跳转，但跳转了，
            top_fix=jalr_id+jalr_ex-jal_id-jal_ex;
            flush_id=1;
            flush_ex=1;
        end
        2'h1: begin //jalr预测失败，说明jalr不是用于函数调用的返回，为了安全清空jalr栈
            jalr_stack_rst=1;
            flush_id=1;
            flush_ex=1;
        end
        2'h2: begin //br
            top_fix=jalr_id+jalr_ex-jal_id-jal_ex;
            flush_id=1;
            flush_ex=1;
        end
        2'h3: begin//jal
            top_fix=jalr_id+jalr_ex;
            flush_id=1;
            flush_ex=1;
        end
    endcase
end
```

NPC_SEL模块的修改

根据`ex`阶段的结果来分类，同时统计分支跳转指令的次数和预测错误次数

```
module NPC_SEL (
    input clk,
    input [31:0] pc_add4_if,
    input [31:0] pc_jalr_ex,
    input [31:0] alu_ans_ex,
    input [31:0] jump_addr,
    input [31:0] jalr_addr,
    input [1:0] pc_sel_ex,
    input [31:0] pc_cur_id,
    input [31:0] pc_add4_ex,
    input [2:0] br_type_ex,
    input br_pre,
    input hit,
    input [31:0] inst_if,
    output reg [31:0] pc_next,
    output reg pre_failure, //当预测失败时为1
    output reg [31:0] br_num,
    output reg [31:0] jal_num,
```

```
    output reg [31:0] jalr_num,
    output reg [31:0] br_fail_num,  //预测失败的次数
    output reg [31:0] jal_fail_num,
    output reg [31:0] jalr_fail_num
);
    initial begin
        br_num=0; jal_num=0; jalr_num=0;
        br_fail_num=0; jal_fail_num=0; jalr_fail_num=0;
    end

    always @(*) begin
        pc_next=pc_add4_if;
        pre_failure=0;
        case (pc_sel_ex)
            2'h0: begin //ex阶段是不应该跳转，根据if阶段预测pc_next
                if(pc_cur_id!=pc_add4_ex && br_type_ex!=0)begin //ex阶段是br指令，且本不应该跳转
                    pre_failure=1;  //预测失败，插入气泡
                    pc_next=pc_add4_ex;
                end
                else begin
                    case (inst_if[6:0])
                        7'b1101111: begin   //jal
                            if(hit)
                                pc_next=jump_addr;
                        end
                        7'b1100111: begin   //jalr
                            pc_next=jalr_addr;
                        end
                        7'b1100011: begin   //br
                            if(br_pre)
                                if(hit)
                                    pc_next=jump_addr;
                        end
                    endcase
                end

            end
            2'h1: begin //ex阶段为jalr指令，jalr指令"在99%的情况下"一定被正确预测，但在助教的测试程序中不行
                if(pc_cur_id!=pc_jalr_ex)begin
                    pre_failure=1;  //预测失败，插入气泡
                    pc_next=pc_jalr_ex;
                end
                else begin
                    case (inst_if[6:0])
                        7'b1101111: begin   //jal
                            if(hit)
                                pc_next=jump_addr;
                        end
                        7'b1100111: begin   //jalr
                            pc_next=jalr_addr;
                        end
                        7'b1100011: begin   //br
                            if(br_pre)
```

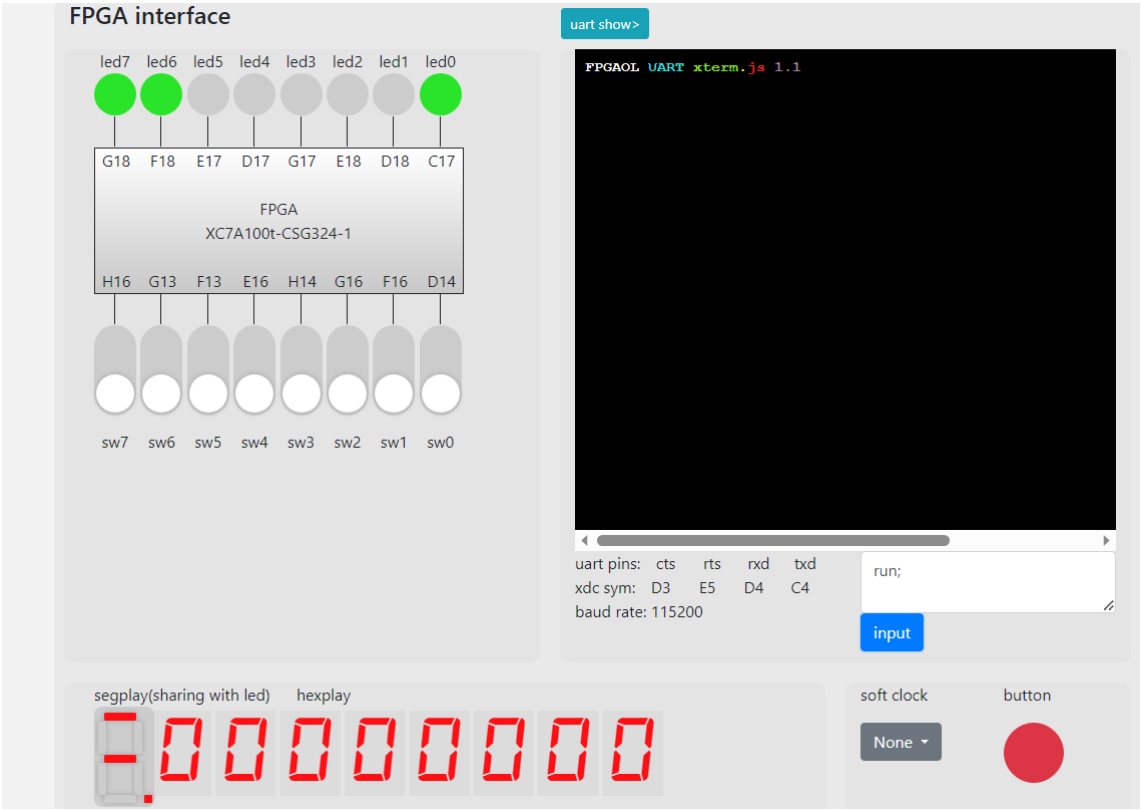
```
                                if(hit)
                                    pc_next=jump_addr;
                        end
                    endcase
                end
            end
            2'h2: begin //ex阶段为br指令，需要检测br指令ex阶段预测是否正确，若不正确则pc_next=alu_ans_ex
                if(pc_cur_id!=alu_ans_ex)begin
                    pre_failure=1;  //预测失败，插入气泡
                    pc_next=alu_ans_ex;
                end
                else begin  //ex阶段预测成功，根据if阶段预测pc_next
                    case (inst_if[6:0])
                        7'b1101111: begin   //jal
                            if(hit)
                                pc_next=jump_addr;
                        end
                        7'b1100111: begin   //jalr
                            pc_next=jalr_addr;
                        end
                        7'b1100011: begin   //br
                            if(br_pre)
                                if(hit)
                                    pc_next=jump_addr;
                        end
                    endcase
                end
            end
            2'h3: begin //ex阶段为jal指令，需要检测jal指令ex阶段预测是否正确，若不正确则pc_next=pc_jalr_ex
                if(pc_cur_id!=pc_jalr_ex)begin
                    pre_failure=1;  //预测失败，插入气泡
                    pc_next=pc_jalr_ex;
                end
                else begin  //ex阶段预测成功，根据if阶段预测pc_next
                    case (inst_if[6:0])
                        7'b1101111: begin   //jal
                            if(hit)
                                pc_next=jump_addr;
                        end
                        7'b1100111: begin   //jalr
                            pc_next=jalr_addr;
                        end
                        7'b1100011: begin   //br
                            if(br_pre)
                                if(hit)
                                    pc_next=jump_addr;
                        end
                    endcase
                end
            end
        endcase
    end

```

```
125     always @(posedge clk) begin
126         case (pc_sel_ex)
127             2'h0: begin //ex阶段是不应该跳转，根据if阶段预测pc_next
128                 if(pc_cur_id!=pc_add4_ex && br_type_ex!=0)begin //ex阶段是br指令，且本不应该跳转
129                     br_num<=br_num+1;
130                     br_fail_num<=br_fail_num+1;
131                 end
132                 else if(br_type_ex!=0)
133                     br_num<=br_num+1;
134             end
135             2'h1: begin //ex阶段为jalr指令，jalr指令"在99%的情况下"一定被正确预测，但在助教的全指令测试程序中不行
136                 if(pc_cur_id!=pc_jalr_ex)begin
137                     jalr_num<=jalr_num+1;
138                     jalr_fail_num<=jalr_fail_num+1;
139                 end
140                 else
141                     jalr_num<=jalr_num+1;
142             end
143             2'h2: begin //ex阶段为br指令，需要检测br指令ex阶段预测是否正确，若不正确则pc_next=alu_ans_ex
144                 if(pc_cur_id!=alu_ans_ex)begin
145                     br_num<=br_num+1;
146                     br_fail_num<=br_fail_num+1;
147                 end
148                 else
149                     br_num<=br_num+1;
150             end
151             2'h3: begin //ex阶段为jal指令，需要检测jal指令ex阶段预测是否正确，若不正确则pc_next=pc_jalr_ex
152                 if(pc_cur_id!=pc_jalr_ex)begin
153                     jal_num<=jal_num+1;
154                     jal_fail_num<=jal_fail_num+1;
155                 end
156                 else
157                     jal_num<=jal_num+1;
158             end
159         endcase
160     end
161 endmodule
```

实验结果

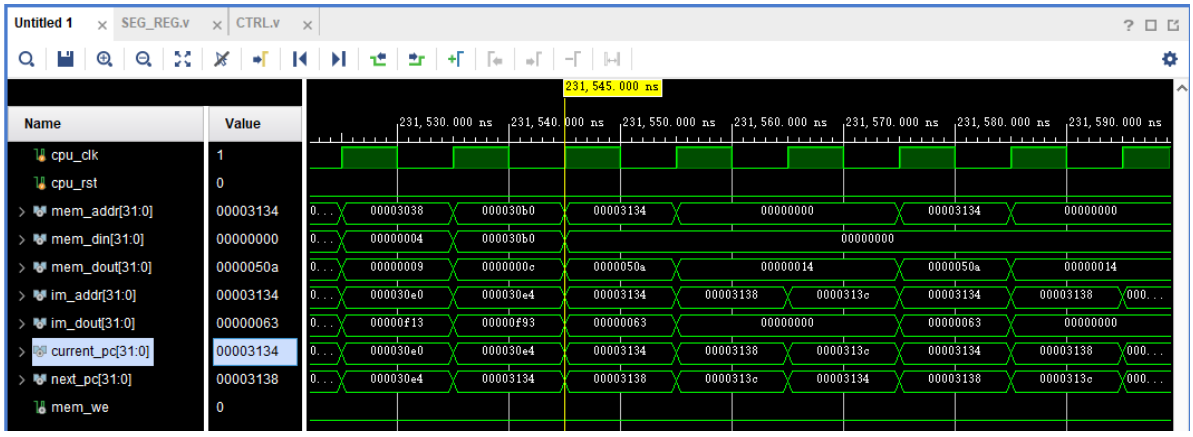
全指令测试



矩阵乘法（两个矩阵大小分别为4*5，5*6，其中数值按顺序为1~20和30~1）

无分支预测的流水线处理器（Lab5）

周期数：23154



仅局部历史分支预测的流水线处理器

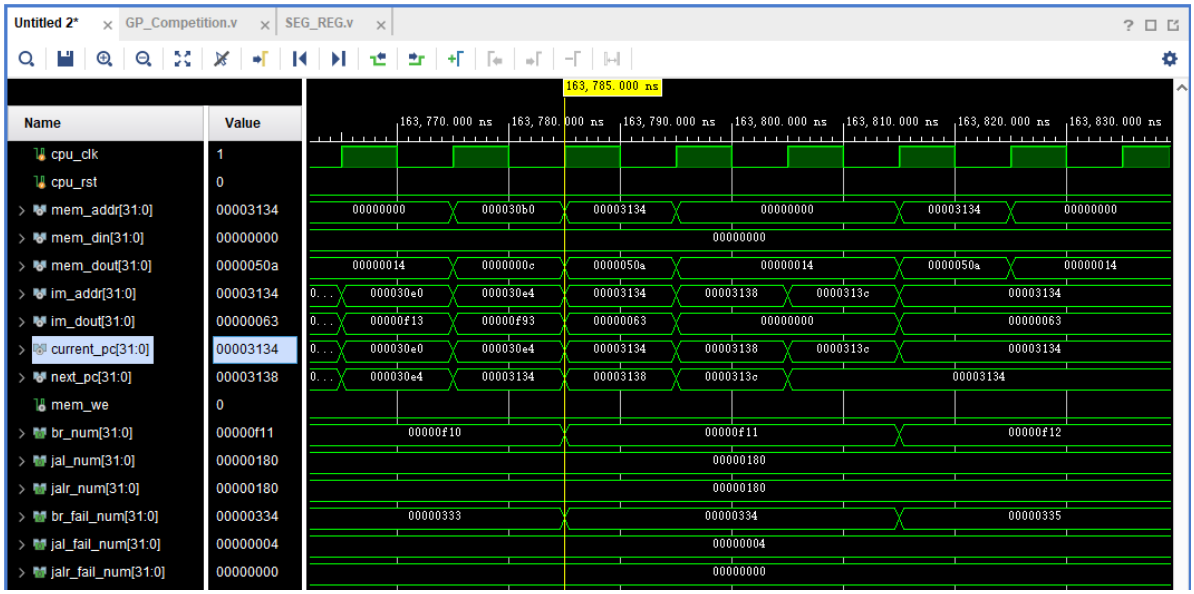
周期数：16378

分支指令数：3'bf10 (3856)

分支指令预测失败数：3'b333 (819)

分支预测成功率：78.76%

比无分支预测的处理器快：41.37%



仅全局历史分支预测的流水线处理器

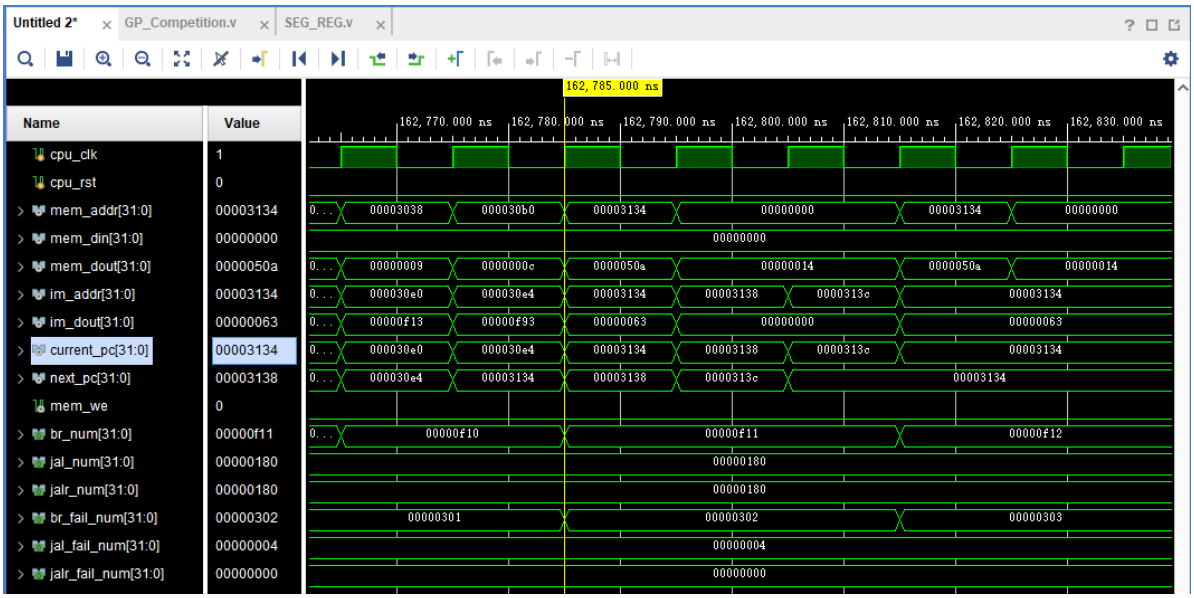
周期数：16278

分支指令数：3'bf10 (3856)

分支指令预测失败数：3'b302 (770)

分支预测成功率：80.03%

比无分支预测的处理器快：42.24%



包含全局历史、局部历史分支预测的流水线处理器

周期数：15636

分支指令数：3'bf10 (3856)

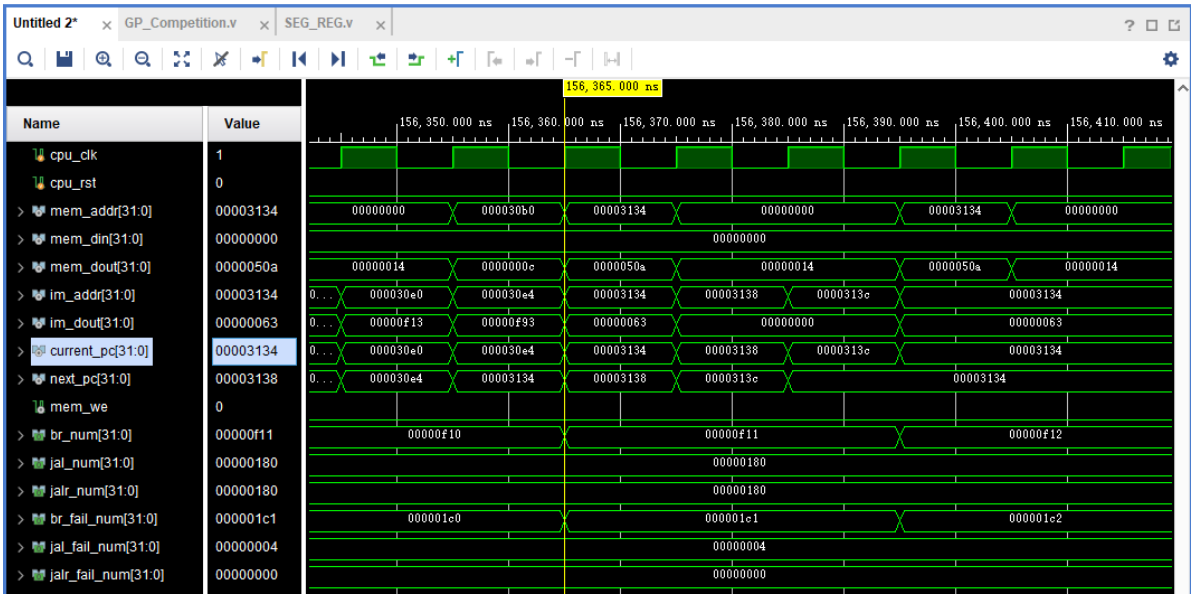
分支指令预测失败数：3'b1c0 (448)

分支预测成功率：88.38%

比无分支预测的处理器快：48.08%

比仅局部历史分支预测的处理器快：4.745%

比仅全局历史分支预测的处理器快：4.106%



体会

分支预测的各种冲突非常多，现代处理器这么快真是人类科技的结晶。期待早日超到10GHz