一、Verilog HDL 语法实践（12）

1. 32 位四选一多路选择器（4）

1.1 代码（1）

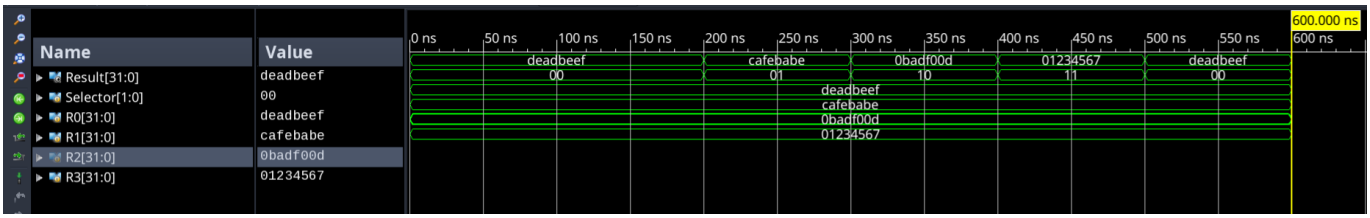
```
module mux4to1(
    input [1:0] Selector,
    input [31:0] R0, R1, R2, R3,
    output reg [31:0] Result // reg
);
    always @(*) begin // always
        case (Selector)
            2'b00: Result = R0;
            2'b01: Result = R1;
            2'b10: Result = R2;
            2'b11: Result = R3;
            default: Result = 32'b0; // Selector00, 01, 10, 11
        endcase
    end
endmodule
```

1.2 激励文件（1）

```
`timescale 1ns / 1ps
module mux4to1_tb;
    // Inputs
    reg [1:0] Selector;
    reg [31:0] R0;
    reg [31:0] R1;
    reg [31:0] R2;
    reg [31:0] R3;
    // Outputs
    wire [31:0] Result;
    // Instantiate the Unit Under Test (UUT)
    mux4to1 uut (
        .Selector(Selector),
        .R0(R0),
        .R1(R1),
        .R2(R2),
        .R3(R3),
        .Result(Result)
    );
    initial begin
        $dumpfile("mux4to1_tb.vcd");
        $dumpvars(0, mux4to1_tb);
        // Initialize Inputs
        Selector = 0;
        R0 = 32'hDEADBEEF;
        R1 = 32'hCAFEBABE;
        R2 = 32'hBADF00D;
        R3 = 32'h01234567;

        // Wait
        #100;
        // Add stimulus here
        Selector = 2'b00;
        #100;
        Selector = 2'b01;
        #100;
        Selector = 2'b10;
        #100;
        Selector = 2'b11;
        #100;
        Selector = 2'b00;
        #100;
        $finish; // End the simulation
    end
endmodule
```

1.3 仿真波形（1）



1.4 结果分析（1）

可以看到，随着 Selector 的值的改变，Result 的值也变为了 R(Selector)

Selector = 00，Result = R0 = 32'hDEADBEEF

Selector = 01，Result = R1 = 32'hCAFEBABE

Selector = 10，Result = R2 = 32'hBADF00D

Selector = 11，Result = R3 = 32'h01234567

四选一多路选择器实现无误

2. ALU 的设计与实现（8）

2.1 代码（2）

```
module ALU (X, Y, ALUctr, R, Overflow, Zero);
    input [31:0] X, Y;                //32
    input [2:0] ALUctr;                         //ALU
    output [31:0] R;                //32
    output Overflow, Zero;    //Overflow-Zero-

    wire SUBctr, OVctr, SIGctr;
    wire [1:0] OPctr;

    assign SUBctr = ALUctr[2];
    assign OVctr   = ~ALUctr[1] & ALUctr[0];
    assign SIGctr  = ALUctr[0];
    assign OPctr[1] = ALUctr[2] & ALUctr[1];
    assign OPctr[0] = ~ALUctr[2] & ALUctr[1] & ~ALUctr[0];

    wire [31:0] Data_Y;
    wire [31:0] one32, zero32;
    assign one32 = 32'h1;
    assign zero32 = 32'h0;
    assign Data_Y = (SUBctr ? one32 ^ Y: Y);

    wire [31:0] F;
    wire OF, SF, CF;

    Adder32 adder32(
        .A(X),
        .B(Data_Y),
        .F(F),                        // R0
        .Cin(SUBctr),
        .Cout(),
        .OF(OF),
        .SF(SF),
        .ZF(Zero),                // result of Zero
        .CF(CF)
    );

    assign Overflow = OVctr & OF;        // result of Overflow
    wire [31:0] Result;
    mux4to1 mux (
        .Selector(OPctr),
        .R0(F),
        .R1(X | Y),
        .R2((SIGctr ? (OF ^ SF) : CF) ? one32 : zero32),
        .R3(32'h0),
        .Result(Result)                // result of R
    );
```

```
    assign R = Result;
endmodule
```

2.2 激励文件（1）

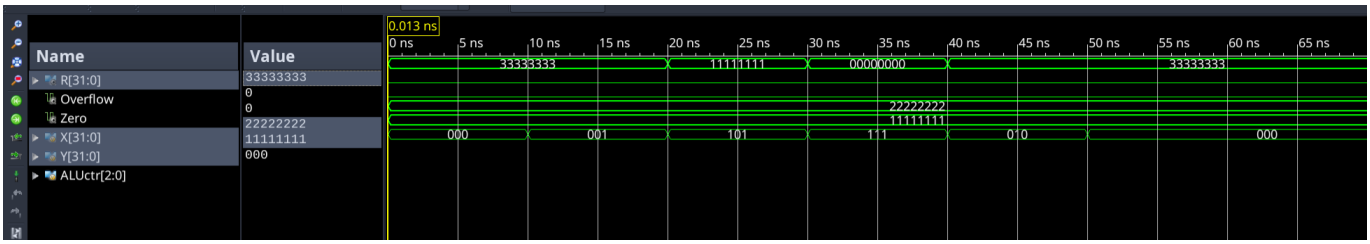
```
module ALU_tb;
    reg [31:0] X, Y;
    reg [2:0] ALUctr;
    wire [31:0] R;
    wire Overflow, Zero;
    // Instantiate the ALU module
    ALU uut (
        .X(X),
        .Y(Y),
        .ALUctr(ALUctr),
        .R(R),
        .Overflow(Overflow),
        .Zero(Zero)
    );
    initial begin
        $dumpfile("ALU_tb.vcd");
        $dumpvars(0, ALU_tb);
        // Initial values
        X = 32'h22222222;
        Y = 32'h11111111;
        ALUctr = 3'b000; // ADD operation
        #10;

        // Test ADD
        ALUctr = 3'b001;
        #10;
        // Test SUB
        ALUctr = 3'b101;
        #10;
        // Test SLT
        ALUctr = 3'b111;
        #10;
        // Test ORI
        ALUctr = 3'b010;
        #10;
        // Test ADDIU
        ALUctr = 3'b000;
        #10;
        // Test LW
        ALUctr = 3'b000;
        #10;
        // Test SW
        ALUctr = 3'b000;
        #10;
        // Test BEQ
        ALUctr = 3'b101;
        #10;
```

```
        $finish;
    end

endmodule
```

2.3 仿真波形（1）



2.4 结果分析（4）

ADDIU，LW，SW 的输入，SUB，BEQ 的输入是一致的，而 ADDU 与 ADD 在 X,Y 的取值下没有差异，故而这里实际上只有三种情况：

1. 3‘b000, 001：可以看到输出的结果是 33333333，为二者值和
2. 3'b111：这是 slt 指令，因为 X 比 Y 大，故而结果为 0，正确
3. 3‘b101：这是 sub 指令，可以看到结果为 11111111

综上所述，从波形图来看，结果无误

二、单周期 CPU 的设计与实现（16）

1. 单周期 CPU 设计与仿真（16）

1.1 代码（4）

```
module CPU(Clk, PC, Inst, R);
    input Clk;                          //
    output [31:0] PC;         //
    output [31:0] Inst;     //
    output [31:0] R;         //ALU

    wire Jump, Branch, Zero, Overflow;
    wire RegWr, ALUSrc, RegDst, MemtoReg, MemWr, ExtOp, WE;
    wire [2:0] ALUctr;

    wire [5:0] OP;
    wire [4:0] Rs, Rt, Rd, Rw;
    wire [15:0] immd;
    wire [5:0] func;
    wire [31:0] extImmd, busA, busB, aluB, busW, Dout, Result;
    wire [31:0] Inst_wire;

    assign OP = Inst_wire[31:26];
    assign Rs = Inst_wire[25:21];
    assign Rt = Inst_wire[20:16];
    assign Rd = Inst_wire[15:11];
    assign immd = Inst_wire[15:0];
    assign func = Inst_wire[5:0];

    assign R = Result;
    assign Inst = Inst_wire;

    assign WE = (~Overflow) & RegWr;

    MUX32_2_1 mux32_1(
        .X1(extImmd),
        .X0(busB),
        .S(ALUSrc),
        .Y(aluB)
    );

    MUX32_2_1 mux32_2(
        .X1(Dout),
        .X0(Result),
        .S(MemtoReg),
        .Y(busW)
    );

    MUX5_2_1 mux5(
        .X1(Rd),
        .X0(Rt),
        .S(RegDst),
```

```
        .Y(Rw)
    );

    ALU cpu_alu(
        .X(busA),
        .Y(aluB),
        .ALUctr(ALUctr),
        .R(Result),
        .Overflow(Overflow),
        .Zero(Zero)
    );

    Ifetch ifetch(
        .Clk(Clk),
        .Jump(Jump),
        .Branch(Branch),
        .Z(Zero),
        .Inst(Inst_wire),
        .PC(PC)
    );

    ControlUnit controlUnit(
        .OP(OP),
        .func(func),
        .RegWr(RegWr),
        .ALUSrc(ALUSrc),
        .RegDst(RegDst),
        .MemtoReg(MemtoReg),
        .MemWr(MemWr),
        .Branch(Branch),
        .Jump(Jump),
        .ExtOp(ExtOp),
        .ALUctr(ALUctr)
    );

    RegFiles regs(
        .CLK(Clk),
        .busW(busW),
        .WE(WE),
        .Rw(Rw),
        .Ra(Rs),
        .Rb(Rt),
        .busA(busA),
        .busB(busB)
    );


    Ext ext(
```

```
        .imm16(immd),
        .ExtOp(ExtOp),
        .Extout(extImmd)
    );

    DataRAM RAM(
        .CLK(Clk),
        .DataIn(busB),
        .WE(MemWr),
        .Address(Result),
        .DataOut(Dout)
    );

endmodule
```

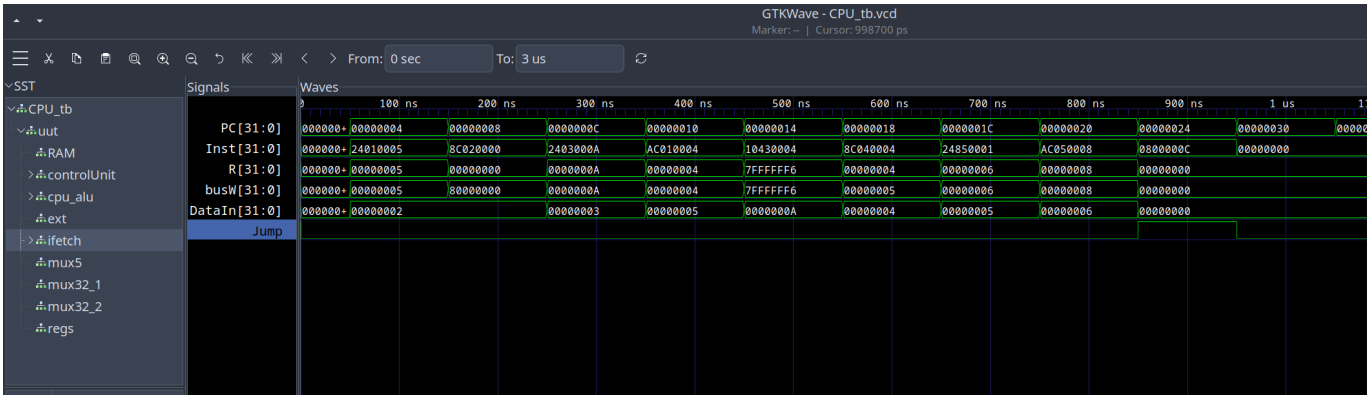
1.2 设计指令序列（4）

新建 MyInstROM_CPU.v:

```
    initial
      begin
        for(i = 0; i < 256; i = i + 1)
          InstROM[i] = 32'h00000000;

          // addi $1,$0, 5   // $1 = 5
        InstROM[1] = {OP_addiu, 5'b00000, 5'b00001, 16'h0005};
        // lw $2, 0($0)     // $2 = M[0]
        InstROM[2] = {OP_lw, 5'b00000, 5'b00010, 16'h0000};
        // addi $3,$0, 10  // $3 = 10
        InstROM[3] = {OP_addiu, 5'b00000, 5'b00011, 16'h000A};
        // sw $1, 4($0)     // M[4] = $1
        InstROM[4] = {OP_sw, 5'b00000, 5'b00001, 16'h0004};
        // beq $2,$3, 4    // if ($2 ==$3) PC += 4
        InstROM[5] = {OP_beq, 5'b00010, 5'b00011, 16'h0004};
        // lw $4, 4($0)     // $4 = M[4]
        InstROM[6] = {OP_lw, 5'b00000, 5'b00100, 16'h0004};
        // addi $5,$4, 1   // $5 =$4 + 1
        InstROM[7] = {OP_addiu, 5'b00100, 5'b00101, 16'h0001};
        // sw $5, 8($0)     // M[8] = $5
        InstROM[8] = {OP_sw, 5'b00000, 5'b00101, 16'h0008};
        // j 0x0000000C     // 无条件跳转到地址 0x0000000C
        InstROM[9] = {OP_jump, 26'h000000C};
     end
```

1.3 设计指令序列仿真波形（2）



1.4 设计指令序列结果分析（6）

指令	结果分析
addi $1,$0, 5	结果的 R 和 busW 都为 5，$1 = 5
lw $2, 0($0)	R 为 0，busW 为 80000000，$2 = 80000000
addi $3,$0, 10	R 与 busW 都为 A，$3 = A
sw $1, 4($0)	R 为 4，DataIn 为 5，是 $1 的值
beq $2,$3, 4	80000000 - A = 7ffffff6，正确结果
lw $4, 4($0)	busW 为 5，是 $1 在之前存放此处的值
addi $5,$4, 1	R 结果为 6 = 5 + 1
sw $5, 8($0)	DataIn 为 6，即 $5 的值
j 0x0000000C	指令从 0x24 跳到了 0x30

CPU 正常运行了这一指令序列

三、流水线 CPU 的设计与实现（18）

1. 流水线寄存器的设计与实现(8)

1.1 IF_ID 流水线寄存器代码（2）

```
module IF_ID(Clk, I_PC4, I_PC, I_Inst, PC4, PC, Inst
    );
     input Clk;
     input [31:0] I_PC4, I_PC, I_Inst;         //从IF流水段输入的信号；
     output reg [31:0] PC4, PC, Inst;         //输出至ID流水段的信号；

     always @ (negedge Clk)
       begin
          PC4  <= I_PC4;
          PC   <= I_PC;
          Inst <= I_Inst;
        end

     initial begin                             //对流水线寄存器进行初始化；
       PC4 = 32'h0; PC = 32'h0; Inst = 32'h0;
     end
endmodule
```

1.2 ID_Ex 流水线寄存器代码（2）

```
module ID_Ex(
     input Clk,
     input [31:0] PC4,
     input [31:0] Jtarg,                 //ID
     input [31:0] busA,
     input [31:0] busB,
     input [5:0] func,
     input [15:0] immd,
     input [4:0] Rd,
     input [4:0] Rt,
     output reg [31:0] E_PC4,
     output reg [31:0] E_Jtarg,        //Ex
     output reg [31:0]  E_busA,
     output reg [31:0] E_busB,
     output reg [5:0] E_func,
     output reg [15:0] E_immd,
     output reg [4:0] E_Rd,
     output reg [4:0] E_Rt,

     input ExtOp,
     input RegDst,
     input ALUSrc,
     input [2:0] ALUctr,
     output reg E_ExtOp,
     output reg E_RegDst,
     output reg E_ALUSrc,
     output reg [2:0] E_ALUctr,

     input Jump,
     input Branch,
     input MemWr,
     output reg E_Jump,
     output reg E_Branch,
     output reg E_MemWr,

     input RegWr,
     input MemtoReg,
     output reg E_RegWr,
     output reg E_MemtoReg


     );

     always @ (negedge Clk)
       begin
          E_PC4 <= PC4;
          E_Jtarg <= Jtarg;
```

```
            E_busA <= busA;
            E_busB <= busB;
            E_func <= func;
            E_immd <= immd;
            E_Rd <= Rd;
            E_Rt <= Rt;

            E_ExtOp <= ExtOp;
            E_RegDst <= RegDst;
            E_ALUSrc <= ALUSrc;
            E_ALUctr <= ALUctr;

            E_Jump <= Jump;
            E_Branch <= Branch;
            E_MemWr <= MemWr;

            E_RegWr <= RegWr;
            E_MemtoReg <= MemtoReg;
         end

      initial begin
         E_PC4 = 32'h0; E_Jtarg = 32'h0; E_busA = 32'h0; E_busB = 32'h0;
         E_func = 5'h0; E_immd = 16'h0; E_Rd = 4'h0; E_Rt = 4'h0;
          E_ExtOp = 1'b0; E_RegDst = 1'b0; E_ALUSrc = 1'b0; E_ALUctr = 3'h0;
         E_Jump = 1'b0; E_Branch = 1'b0; E_MemWr = 1'b0;
         E_RegWr = 1'b0;    E_MemtoReg = 1'b0;
      end


   endmodule
```

1.3 Ex_Mem 流水线寄存器代码（2）

```
module Ex_Mem(
            input Clk,
            input [31:0] E_Jtarg,
            input [31:0] E_Btarg,
            input E_Zero,
            input E_Overflow,
            input [31:0] E_ALUout,
            input [31:0] E_busB,
            input [4:0] E_Rw,
            output reg [31:0] M_Jtarg,
            output reg [31:0] M_Btarg,
            output reg M_Zero,
            output reg M_Overflow,
            output reg [31:0] M_ALUout,
            output reg [31:0] M_busB,
            output reg [4:0] M_Rw,

            input E_Jump,
             input E_Branch,
             input E_MemWr,
             output reg M_Jump,
             output reg M_Branch,
             output reg M_MemWr,

             input E_RegWr,
             input E_MemtoReg,
             output reg M_RegWr,
             output reg M_MemtoReg
    );

     always @ (negedge Clk)
       begin
            M_Jtarg <= E_Jtarg;
            M_Btarg <= E_Btarg;
            M_Zero <= E_Zero;
            M_Overflow <= E_Overflow;
            M_ALUout <= E_ALUout;
            M_busB <= E_busB;
            M_Rw <= E_Rw;

              M_Jump <= E_Jump;
            M_Branch <= E_Branch;
            M_MemWr <= E_MemWr;

            M_RegWr <= E_RegWr;
            M_MemtoReg <= E_MemtoReg;
        end
```

```
    initial begin                           //
        M_Jtarg = 32'h0;
        M_Btarg = 32'h0;
        M_Zero = 0;
        M_Overflow = 0;
        M_ALUout = 32'h0;
        M_busB = 32'h0;
        M_Rw = 4'h0;
        M_Jump = 1'b0; M_Branch = 1'b0; M_MemWr = 1'b0;
        M_RegWr = 1'b0; M_MemtoReg = 1'b0;
    end
endmodule
```

1.4 Mem_Wr 流水线寄存器代码（2）

```
module Mem_Wr(
            input Clk,
            input [31:0] M_Dout,
            input [31:0] M_ALUout,
            input M_Overflow,
            input [4:0] M_Rw,
            output reg [31:0] W_Dout,
            output reg [31:0] W_ALUout,
            output reg W_Overflow,
            output reg [4:0] W_Rw,

            input M_RegWr,
             input M_MemtoReg,
             output reg W_RegWr,
             output reg W_MemtoReg
    );

     always @ (negedge Clk)
       begin
          W_Dout  <= M_Dout;
          W_ALUout   <= M_ALUout;
          W_Overflow <= M_Overflow;
          W_Rw  <=  M_Rw;
          W_RegWr <= M_RegWr;
          W_MemtoReg <= M_MemtoReg;
        end

     initial begin                                //
       W_Dout = 32'h0; W_ALUout = 32'h0; W_Overflow = 0; W_Rw = 0;
       W_RegWr = 0; W_MemtoReg = 0;
     end

endmodule
```

2. 五级流水线结构处理器设计与仿真（10）

2.1 代码（4）

```
module PPCPU (Clk, I_PC, I_Inst, Inst, E_ALUout, M_ALUout, W_RegDin);
        input Clk;            //
        output [31:0] I_PC;        //IF
        output [31:0] I_Inst;        //IF
        output [31:0] Inst;        //ID
        output [31:0] E_ALUout;//ExALU
        output [31:0] M_ALUout;    //MemALU
        output [31:0] W_RegDin;    //Wr


          // IF
          wire [31:0] PCin, PCout, I_PC4;
          wire [31:0] I_PC_wire, I_Inst_wire;  // for output

          // ID
          wire [31:0] PC4, PC, busA, busB, Jtarg;
          // wire [25:0] target;
          wire [4:0] Rs, Rt, Rd;
          wire [5:0] func;
          wire [15:0] immd;
          wire [31:0] Inst_wire; // for output

          //Ex
          wire [31:0] E_Jtarg, E_Btarg, E_PC4, E_busA, E_busB, E_immd_ext;
          wire [15:0] E_immd;
          wire [5:0] E_func;
          wire [4:0] E_Rt, E_Rd, E_Rw;
          //wire [2:0] /*E_ALUop, E_func_CU,*/ ALUctr;
          wire E_RegDst, /*E_R_type,*/ E_Z, E_Overflow, E_ALUSrc, E_ExtOp;
          wire [31:0] E_ALUout_wire; // for output

          //Mem
          wire [31:0] M_Btarg, M_Jtarg, M_busB,M_Dout, E_Btarg_or_Jtarg;
          wire [4:0] M_Rw;
          wire M_Z, M_Overflow, M_MemWr, M_Branch, M_Jump, E_PCSrc;
          wire [31:0] M_ALUout_wire; // for output

          // Wr
          wire [31:0] W_Dout, W_ALUout;
          wire [4:0] W_Rw;
          wire W_Overflow, W_RegWr, W_RegWE, W_MemtoReg;
          wire [31:0] W_RegDin_wire; // for output

          assign I_PC = I_PC_wire;
          assign I_Inst = I_Inst_wire;
          assign Inst = Inst_wire;
          assign E_ALUout = E_ALUout_wire;
```

```
assign M_ALUout = M_ALUout_wire;
assign W_RegDin = W_RegDin_wire;



/************
      IF
  ***********/

MUX32_2_1 mux32_3(
      .X1(E_Btarg_or_Jtarg),
      .X0(I_PC4),
      .S(E_PCSrc),
      .Y(PCin)
  );

PC pc (
      .Clk(Clk),
      .PCin(PCin),
      .PCout(I_PC_wire)
  );

  assign I_PC4 = I_PC_wire + 4;

  MyInstROM instROM (
          .Addr(I_PC_wire),
          .INST(I_Inst_wire)
  );

  IF_ID if_id (
          .Clk(Clk),
          .I_PC4(I_PC4), .I_PC(I_PC_wire), .I_Inst(I_Inst_wire),
          .PC4(PC4), .PC(PC), .Inst(Inst_wire)
  );



/************
      ID
  ***********/

  wire [5:0] OP;
  assign OP = Inst[31:26];
  assign Rs = Inst[25:21];
  assign Rt = Inst[20:16];
  assign Rd = Inst[15:11];
  assign immd = Inst[15:0];
  assign func = Inst[5:0];
```

```
wire ExtOp, ALUSrc, RegDst, Jump, Branch, MemWr, RegWr, MemtoReg;
wire [2:0] ALUctr;

ControlUnit CU (
        .OP(OP), .func(func),
        .RegWr(RegWr),
        .ALUSrc(ALUSrc),
        .RegDst(RegDst),
        .MemtoReg(MemtoReg),
        .MemWr(MemWr),
        .Branch(Branch),
        .Jump(Jump),
        .ExtOp(ExtOp),
        .ALUctr(ALUctr)
 );

 RegFiles regfiles (
        .CLK(Clk),
        .busW(W_RegDin),
        .WE(W_RegWE),
        .Rw(W_Rw),
        .Ra(Rs),
        .Rb(Rt),
        .busA(busA),
        .busB(busB)
    );

wire E_Jump, E_Branch, E_MemWr, E_RegWr, E_MemtoReg;
wire [2:0] E_ALUctr;
assign Jtarg = {PC[31:28], Inst[25:0], 2'b00};
ID_Ex id_ex (
        .Clk(Clk),
        .PC4(PC4), .Jtarg(Jtarg),
        .busA(busA), .busB(busB),
        .func(func), .immd(immd),
        .Rd(Rd), .Rt(Rt),
        .E_PC4(E_PC4), .E_Jtarg(E_Jtarg) ,
        .E_busA(E_busA), .E_busB(E_busB),
        .E_func(E_func), .E_immd(E_immd),
        .E_Rd(E_Rd), .E_Rt(E_Rt),

        .ExtOp(ExtOp), .RegDst(RegDst),
        .ALUSrc(ALUSrc), .ALUctr(ALUctr),
        .E_ExtOp(E_ExtOp), .E_RegDst(E_RegDst),
        .E_ALUSrc(E_ALUSrc), .E_ALUctr(E_ALUctr),

        .Jump(Jump), .Branch(Branch), .MemWr(MemWr),
```

```
            .E_Jump(E_Jump), .E_Branch(E_Branch), .E_MemWr(E_MemWr),

            .RegWr(RegWr), .MemtoReg(MemtoReg),
            .E_RegWr(E_RegWr), .E_MemtoReg(E_MemtoReg)
    );



/************
      Ex
  ***********/

    assign E_Rw = (E_RegDst ? E_Rd : E_Rt);

    Ext ext1 (
            .imm16(E_immd),
            .ExtOp(E_ExtOp),
            .Extout(E_immd_ext)
    );

    wire [31:0] AluB;
    MUX32_2_1 mux32_4(
            .X1(E_immd_ext),
            .X0(E_busB),
            .S(E_ALUSrc),
            .Y(AluB)
    );

    ALU alu_ppcpu (
            .X(E_busA),
            .Y(AluB),
            .ALUctr(E_ALUctr),
            .R(E_ALUout_wire),
            .Overflow(E_Overflow),
            .Zero(E_Z)
    );

    assign E_Btarg = E_PC4 + (E_immd_ext << 2);

    wire M_RegWr, M_MemtoReg;
    Ex_Mem ex_mem (
            .Clk(Clk),
            .E_Jtarg(E_Jtarg), .E_Btarg(E_Btarg),
            .E_Zero(E_Z), .E_Overflow(E_Overflow), .E_ALUout(E_ALUout),
            .E_busB(E_busB), .E_Rw(E_Rw),
            .M_Jtarg(M_Jtarg), .M_Btarg(M_Btarg),
            .M_Zero(M_Z), .M_Overflow(M_Overflow),
            .M_ALUout(M_ALUout), .M_busB(M_busB), .M_Rw(M_Rw),
```

```
        .E_Jump(E_Jump), .E_Branch(E_Branch), .E_MemWr(E_MemWr),
        .M_Jump(M_Jump), .M_Branch(M_Branch), .M_MemWr(M_MemWr),

        .E_RegWr(E_RegWr), .E_MemtoReg(E_MemtoReg),
        .M_RegWr(M_RegWr), .M_MemtoReg(M_MemtoReg)
);



/************
    Mem
***********/

MUX32_2_1 mux32_5 (
        .X0(M_Jtarg),
        .X1(M_Btarg),
        .S(M_Branch),
        .Y(E_Btarg_or_Jtarg)
);

assign E_PCSrc = M_Jump | (M_Branch & M_Z);

DataRAM RAM_1 (
        .CLK(Clk),
        .DataIn(M_busB), .DataOut(M_Dout),
        .WE(M_MemWr), .Address(M_ALUout)
);

Mem_Wr mem_wr (
        .Clk(Clk),
        .M_Dout(M_Dout), .M_ALUout(M_ALUout),
        .M_Overflow(M_Overflow), .M_Rw(M_Rw),
        .W_Dout(W_Dout), .W_ALUout(W_ALUout),
        .W_Overflow(W_Overflow), .W_Rw(W_Rw),

        .M_RegWr(M_RegWr), .M_MemtoReg(M_MemtoReg),
        .W_RegWr(W_RegWr), .W_MemtoReg(W_MemtoReg)
);



/************
    Wr
***********/

MUX32_2_1 mux32_6 (
        .X0(W_ALUout),
```

```
                .X1(W_Dout),
                .S(W_MemtoReg),
                .Y(W_RegDin)
            );

            assign W_RegWE = (W_RegWr & (~W_Overflow));
endmodule
```

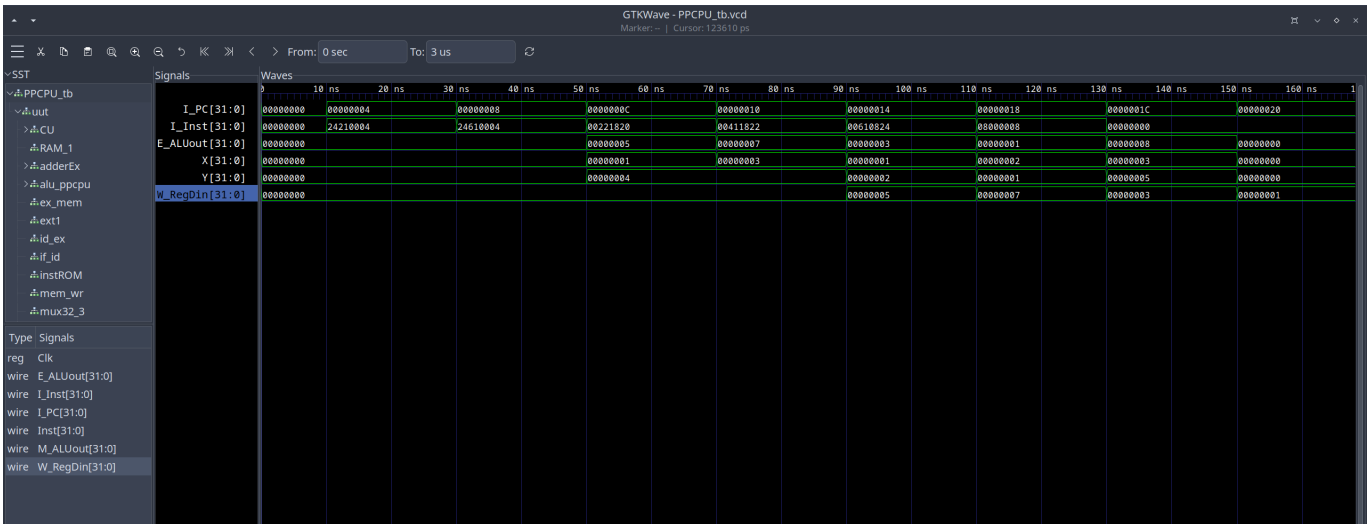
2.2 指令序列（1）

新建 MyInstROM.v

```
InstROM[8'h01] = {OP_addiu, 5'b00001, 5'b00001, 16'h0004};
InstROM[8'h02] = {OP_addiu, 5b00011, 5'b00001, 16'h0004};
InstROM[8'h03] = {OP_R_type, 5'b00001, 5'b00010, 5'b00011, shamt, FUNC_add};
InstROM[8'h04] = {OP_R_type, 5'b00010, 5'b00001, 5'b00011, shamt, FUNC_sub};
InstROM[8'h05] = {OP_R_type, 5'b00011, 5'b00001, 5'b00001, shamt, FUNC_and};
```

```
addi    $1, $1, 4
addi    $1, $3, 4
add     $3, $1, $2
sub     $3, $2, $1
and     $1, $3, $1
```

2.3 仿真波形（1）



2.4 结果分析（4）

在读出第一条指令（IF），计算出第一条指令结果（Ex），和将计算结果写回（Wr）之间都间隔了两个时钟周期，分别对应五级流水线的 1,3,5 段

可以看到，在执行第三条指令时，由于 $1 的结果尚未写回，所以 ALU 得到的输入 X 仍然为 1，直到产生 W_RegDin 后的一个周期后，即在执行第 5 条指令的 Ex 段时，才看到所取出的 $1 的值变为第一条指令的结果 5

PPCPU 成功地像流水线一般依次执行各个指令的各个阶段，在指令见相互不影响的情况之下可以正确地执行，然而存在诸如数据冒险的问题

四、流水线冒险及处理 - 数据冒险（12）

1. 普通数据冒险问题处理（6）

1.1 数据前推通路关键代码段及文字说明（2）

a) 为 ID_Ex 新增 Rs 输入与 E_Rs 输出

b) 使用 DetUnit 生成 ALUSrcA 与 ALUSrcB 信号

```
    wire [1:0] ALUSrcA, ALUSrcB;
    DetUnit detunit (
        .E_Rs(E_Rs),
        .E_Rt(E_Rt),
        .E_ALUSrc(E_ALUSrc),
        .M_Rw(M_Rw),
        .W_Rw(W_Rw),
        .M_RegWr(M_RegWr),
        .W_RegWr(W_RegWr),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB)
    );
```

c) 使用 ALUSrcA 与 ALUSrcB 选择 AluA 与 AluB

```
    wire [31:0] AluA, AluB;
    mux4to1 mux1 (
        .Selector(ALUSrcA),
        .R0(E_busA),
        .R1(M_ALUout),
        .R2(W_RegDin),
        .Result(AluA)
    );

    mux4to1 mux2 (
        .Selector(ALUSrcB),
        .R0(E_busB),
        .R1(M_ALUout),
        .R2(W_RegDin),
        .R3(E_immd_ext),
        .Result(AluB)
    );
```

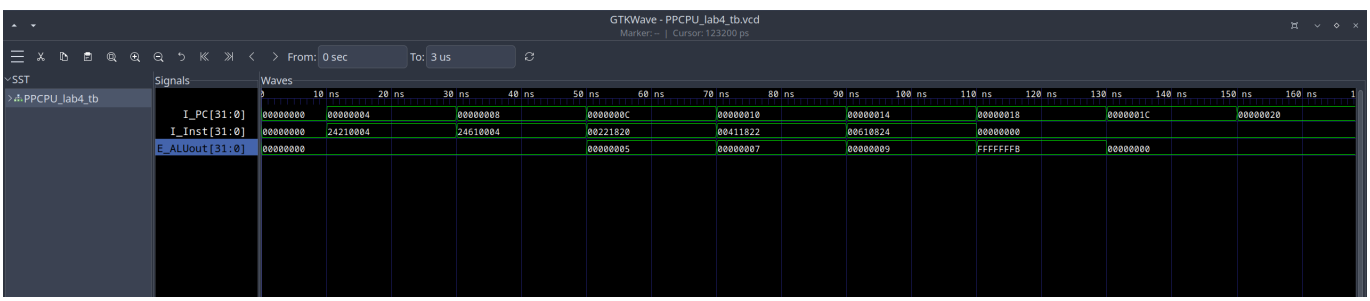
1.2 仿真指令序列（1）

使用 MyInstROM.v

```
InstROM[8'h01] = {OP_addiu, 5'b00001, 5'b00001, 16'h0004};
InstROM[8'h02] = {OP_addiu, 5b00011, 5'b00001, 16'h0004};
InstROM[8'h03] = {OP_R_type, 5'b00001, 5'b00010, 5'b00011, shamt, FUNC_add};
InstROM[8'h04] = {OP_R_type, 5'b00010, 5'b00001, 5'b00011, shamt, FUNC_sub};
```

```
addi    $1, $1, 4
addi    $1, $3, 4
add     $3, $1, $2
sub     $3, $2, $1
```

1.3 仿真波形（1）



1.4 结果分析（2）

第一条指令成功将寄存器 $1 的值初始化为 5，紧随其后的第二条指令则将其值更新为 7。通过观察波形的 E_ALUout 部分，我们可以确认运算结果正确无误。

在第三条指令中，我们对寄存器 $1 和 $2 的值进行相加操作，并将结果存储于寄存器 $3。根据计算，7 加 2 应等于 9，波形显示的结果与此相符，验证了运算的正确性。

第四条指令则是对寄存器 $1 和 $2 的值进行相减，并将结果赋给寄存器 $3。计算结果为 2 减 7，即 -5，波形显示的十六进制数值 FFFFFFFB 与预期一致。

可以看到，PPCPU 成功地处理了第二条与第三条指令之间的数据冒险问题，确保了指令执行的连续性和正确性。

2. load-use 数据冒险问题处理（6）

2.1 流水线关键代码段及文字说明（2）

a) 为 PC，IF_ID 寄存器添加写使能 EN

```
module PC_lab4(Clk, EN, PCin, PCout);
     input Clk;         //CLK-ĥ���źţ�
     input EN;
     input [31:0] PCin;    //PCin-����������������
     output [31:0] PCout;    //PCout-������������������
     reg [31:0] PC = 32'h00000000;
     always @ (negedge Clk)
         if (EN) begin
               PC <= PCin;
        end
     assign PCout = PC;
endmodule
```

```
module IF_ID_lab4(Clk, EN, I_PC4, I_PC, I_Inst, PC4, PC, Inst
    );
     input Clk;
     input EN;
     input [31:0] I_PC4, I_PC, I_Inst;         //从IF流水段输入的信号；
     output reg [31:0] PC4, PC, Inst;         //输出至ID流水段的信号；

     always @ (negedge Clk)
       begin
            if (EN) begin
                PC4  <= I_PC4;
                PC   <= I_PC;
                Inst <= I_Inst;
            end
        end

     initial begin                              //对流水线寄存器进行初始化；
       PC4 = 32'h0; PC = 32'h0; Inst = 32'h0;
     end
endmodule
```

c) 为 ID_Ex 加入清零信号

```
always @ (negedge Clk)
       begin
            if (!Clrn) begin
                E_PC4 <= 32'h0; E_Jtarg <= 32'h0; E_busA <= 32'h0; E_busB <= 32'h0;
                E_func <= 5'h0; E_immd <= 16'h0; E_Rd <= 4'h0; E_Rt <= 4'h0; E_Rs <= 4'h0;
                 E_ExtOp <= 1'b0; E_RegDst <= 1'b0; E_ALUSrc <= 1'b0; E_ALUctr <= 3'h0;
                E_Jump <= 1'b0; E_Branch <= 1'b0; E_MemWr <= 1'b0;
                E_RegWr <= 1'b0;    E_MemtoReg <= 1'b0;
            end else begin
                E_PC4 <= PC4;
                E_Jtarg <= Jtarg;
                ......
```

d) 连线，在 ID 段使用 DetUnit_load 产生的 load_use 信号作为使能或清零信号传给寄存器

```
    PC_lab4 pc (
        .Clk(Clk),
        .EN(not_load_use),
        ......
    IF_ID_lab4 if_id (
        .Clk(Clk),
        .EN(not_load_use),
        ......

    DetUnit_load detunit_load (
        .E_MemtoReg(E_MemtoReg),
        .Rs(Rs), .Rt(Rt), .E_Rt(E_Rt),
        .load_use(load_use)
    );
    assign not_load_use = ~load_use;

    ID_Ex_lab4_2 id_ex (
        .Clk(Clk),
        .Clrn(not_load_use),
        ......
```

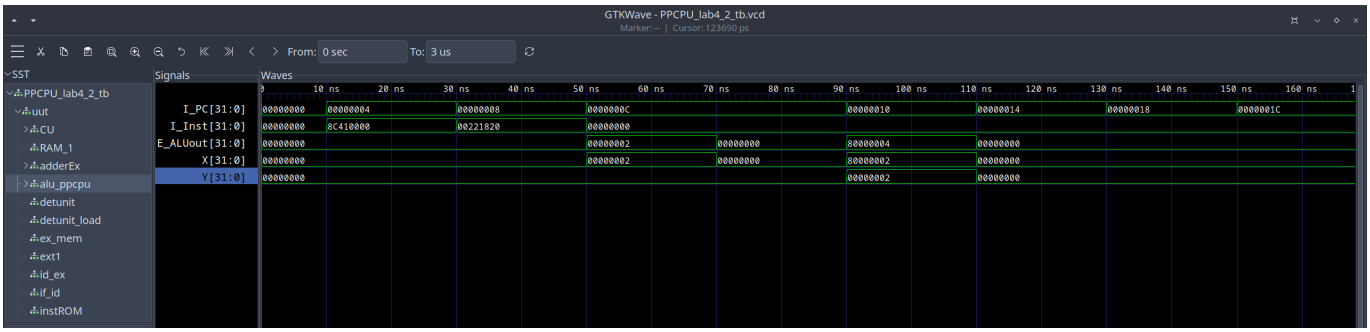
2.2 仿真指令序列（1）

新建了 MyInstROM2.v

```
InstROM[8'h01] = {OP_lw, 5'b00010, 5'b00001, 16'h0000};
InstROM[8'h02] = {OP_R_type, 5'b00001, 5'b00010, 5'b00011, shamt, FUNC_add};
```

```
lw     $1, 0($2)
add    $3, $1, $2
```

2.3 仿真波形（1）



2.4 结果分析（2）

这里我设置了一个简单的 load-use 冒险。第一条指令将 $2 寄存器所存储的地址所指向的值赋给 $1。按照默认，$2 的值为 2，DataRAM[i] 的值为 0x80000000 + i

第二条指令在未写回的时候，将 $1，$2 寄存器的值相加。可以看到，这个时候 $1 的值为 80000002，成功解决了 load-use 数据冒险

五、流水线冒险及处理 - 控制冒险（12）

1. 控制冒险问题处理（12）

1.1 修改关键代码段及文字说明（2）

a) 在 Mem 段生成 M_PCSrc 信号，并为 Mem_Wr 添加 M_PCSrc 输入与 W_PCSrc 输出

```
    input M_PCSrc,
    output reg W_PCSrc,
    ......
    always @ (negedge Clk)
       begin
          ........
          W_PCSrc <= M_PCSrc;
       end
    initial begin                              //
        ........
        W_PCSrc = 0;
     end
```

```
assign M_PCSrc = (M_Branch & M_Z) | M_Jump;
```

b) 为 Ex_Mem 寄存器添加清零信号 Clrn

```
always @ (negedge Clk)
   begin
    if (Clrn) begin
          M_Jtarg <= E_Jtarg;
          M_Btarg <= E_Btarg;
          M_Zero <= E_Zero;
          M_Overflow <= E_Overflow;
          M_ALUout <= E_ALUout;
          M_busB <= E_busB;
          M_Rw <= E_Rw;
        M_Jump <= E_Jump;
        M_Branch <= E_Branch;
        M_MemWr <= E_MemWr;
        M_RegWr <= E_RegWr;
        M_MemtoReg <= E_MemtoReg;
    end else begin
        M_Jtarg <= 32'h0;
          M_Btarg <= 32'h0;
          M_Zero <= 0;
          M_Overflow <= 0;
          M_ALUout <= 32'h0;
          M_busB <= 32'h0;
          M_Rw <= 4'h0;
          M_Jump <= 1'b0; M_Branch <= 1'b0; M_MemWr <= 1'b0;
          M_RegWr <= 1'b0; M_MemtoReg <= 1'b0;
    end
   end
```

c) 连线，设置 ID_Ex，Ex_Mem 的 Clrn 信号

```
    Ex_Mem_lab5 ex_mem (
            .Clk(Clk),
            .Clrn(~M_PCSrc),
    .......

    ID_Ex_lab4_2 id_ex (
            .Clk(Clk),
            .Clrn(~(load_use | M_PCSrc | W_PCSrc)),
```

1.2 仿真指令序列（2）

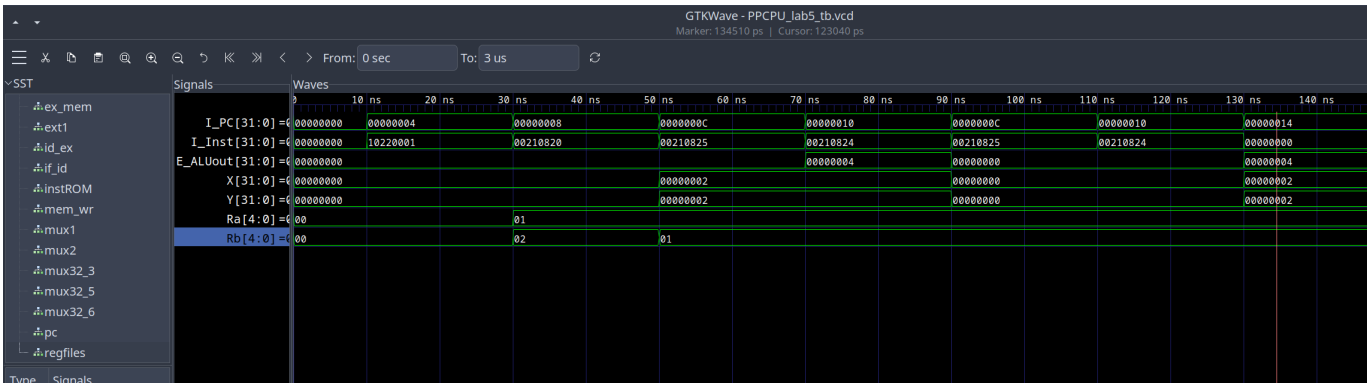
新建 MyInstROM3.v，初始化序列：

```
InstROM[8'h01] = {OP_beq, 5'b00001, 5'b00010, 16'h0001};
InstROM[8'h02] = {OP_R_type, 5'b00001, 5'b00001, 5'b00001, shamt, FUNC_add};
InstROM[8'h03] = {OP_R_type, 5'b00001, 5'b00001, 5'b00001, shamt, FUNC_or};
InstROM[8'h04] = {OP_R_type, 5'b00001, 5'b00001, 5'b00001, shamt, FUNC_and};
```

```
    beq    $1, $2, jump
    add    $1, $1, $1
jump:
    or     $1, $1, $1
    and    $1, $1, $1
```

本次初始化寄存器 $1 的值为 2，与 $2 相同

1.3 仿真波形（2）



1.4 结果分析（6）

观察可见，在首个 ALU 操作中，输入的 X 和 Y 均为 2，根据前一个时钟周期中 Ra 和 Rb 的值，我们可以推断出这是来自寄存器 $1 和 $2 的数据，且二者数值相同。尽管如此，CPU 并未因此跳转，而是继续推进至下一条指令。当预测错误被检测到时，我们可以观察到系统状态被清零的现象，紧接着程序计数器（PC）的值回退至正确的跳转位置。

根据我所设计的指令逻辑，若寄存器 $1 和 $2 的值不相等，则 $1 的值应翻倍。在执行跳转操作之后，我们可以注意到，尽管确实执行了第二条指令，但 $1 的值并未变为 4，而是依旧保持为 2。这一现象表明，成功实现了简单预测的策略。

六、课后作业（30）

1. 实验2（10）

1.1 指令序列分析（4）

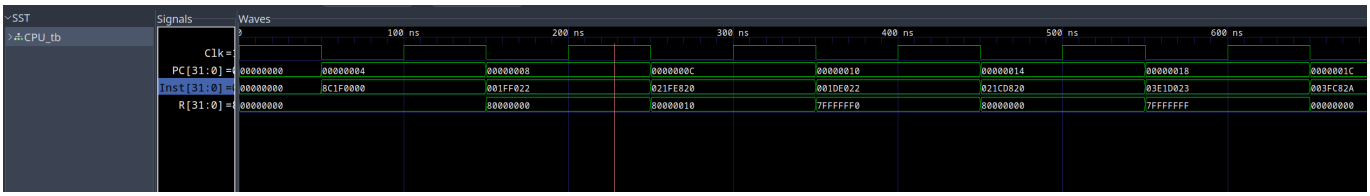
InstROM 源代码为：

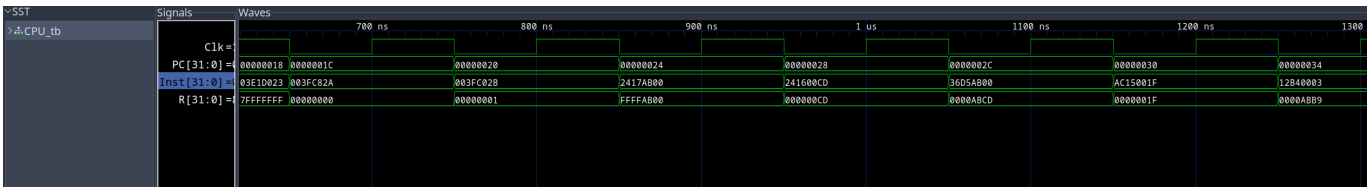
```
InstROM[8'h01] = {OP_lw, 5'b00000, 5'b11111, 16'h0000};
InstROM[8'h02] = {OP_R_type, 5'b00000, 5'b11111, 5'b11110, shamt, FUNC_sub};
InstROM[8'h03] = {OP_R_type, 5'b10000, 5'b11111, 5'b11101, shamt, FUNC_add};
InstROM[8'h04] = {OP_R_type, 5'b00000, 5'b11101, 5'b11100, shamt, FUNC_sub};
InstROM[8'h05] = {OP_R_type, 5'b10000, 5'b11100, 5'b11011, shamt, FUNC_add};
InstROM[8'h06] = {OP_R_type, 5'b11111, 5'b00001, 5'b11010, shamt, FUNC_subu};
InstROM[8'h07] = {OP_R_type, 5'b00001, 5'b11111, 5'b11001, shamt, FUNC_slt};
InstROM[8'h08] = {OP_R_type, 5'b00001, 5'b11111, 5'b11000, shamt, FUNC_sltu};
InstROM[8'h09] = {OP_addiu, 5'b00000, 5'b10111, 16'hab00};
InstROM[8'h0a] = {OP_addiu, 5'b00000, 5'b10110, 16'h00cd};
InstROM[8'h0b] = {OP_ori, 5'b10110, 5'b10101, 16'hab00};
InstROM[8'h0c] = {OP_sw, 5'b00000, 5'b10101, 16'h001f};
InstROM[8'h0d] = {OP_beq, 5'b10101, 5'b10100, 16'h0003};
InstROM[8'h0e] = {OP_lw, 5'b00000, 5'b10100, 16'h001f};
InstROM[8'h0f] = {OP_beq, 5'b10101, 5'b10100, 16'h0003};
InstROM[8'h13] = {OP_jump, 26'h000000f};
```

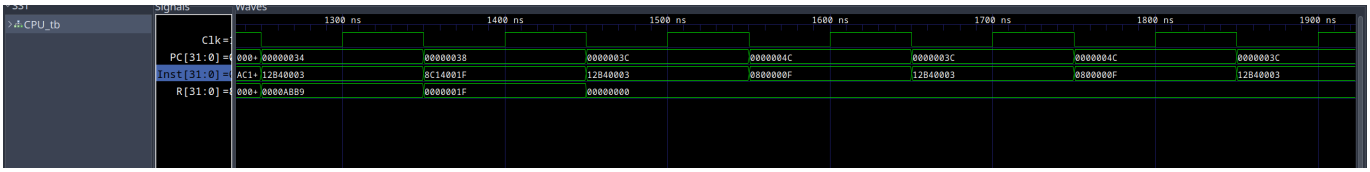
其 MIPS 汇编代码为：

```
lw     $31, 0($zero)
sub    $30, $zero, $31
add    $29, $16, $31
sub    $28, $zero, $29
add    $27, $16, $28
subu   $26, $31, $1
slt    $25, $1, $31
sltu   $24, $1, $31
addiu  $23, $zero, 0xab00
addiu  $22, $zero, 0x00cd
ori    $21, $22, 0xab00
sw     $21, 0x1f($zero)
beq    $21, $20, 0x3
lw     $20, 0x1f($zero)
beq    $21, $20, 0x3
j      0x0000000f
```

1.2 仿真波形（2）







1.3 结果分析（4）

我们可以从指令序列中初步分析结果

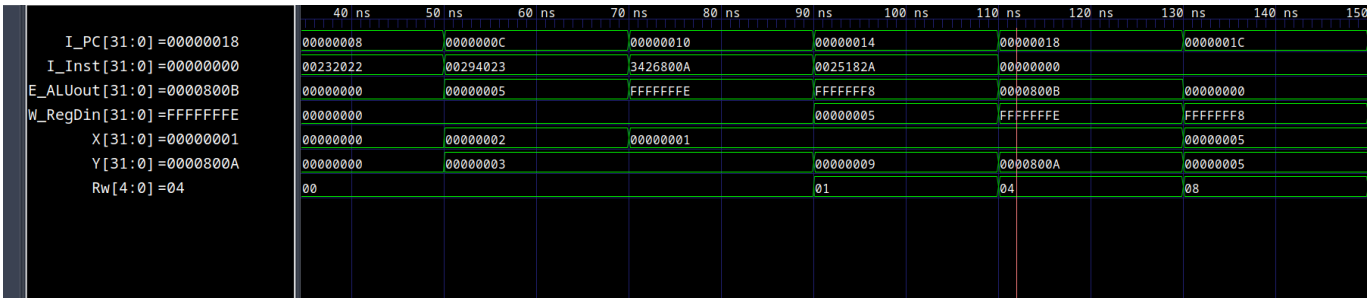
```
lw      $31, 0($zero)        // R = 0, busW = 80000000  $31
sub     $30, $zero, $31      // R = 80000000    $30
add     $29, $16, $31        // R = 80000010    $29
sub     $28, $zero, $29      // R = 7ffffff0    $28
add     $27, $16, $28        // R = 80000000    $27
subu    $26, $31, $1         // R = 7fffffff    $26
slt     $25, $1, $31         // R = 0           $25
sltu    $24, $1, $31         // R = 1           $24
addiu   $23, $zero, 0xab00   // R = ffffab00    $23
addiu   $22, $zero, 0x00cd   // R = 000000cd    $22
ori     $21, $22, 0xab00     // R = 0000abcd    $21
sw      $21, 0x1f($zero)     // R = 1f
beq     $21, $20, 0x3        // not equals
lw      $20, 0x1f($zero)     // $20 = $21
beq     $21, $20, 0x3        // jump to 0x4c
j       0x0000000f           // jump to 0x3c
```

可以看到，上面的仿真波形结果完美符合 R 的预估值。在最后，地址也如期地在 0x3c 和 0x4c 之间来回跳转

2. 实验3（10）

2.1. 修改关键代码及文字说明（2）

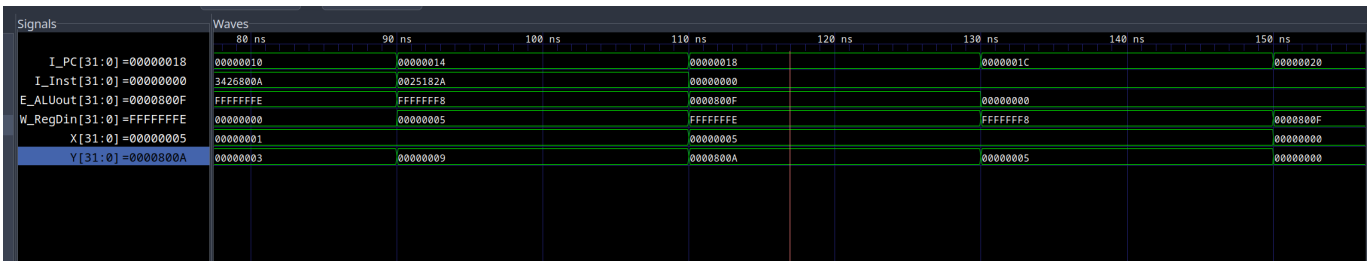
初步仿真可以看到，在第四条指令时，$1 的结果尚未写回

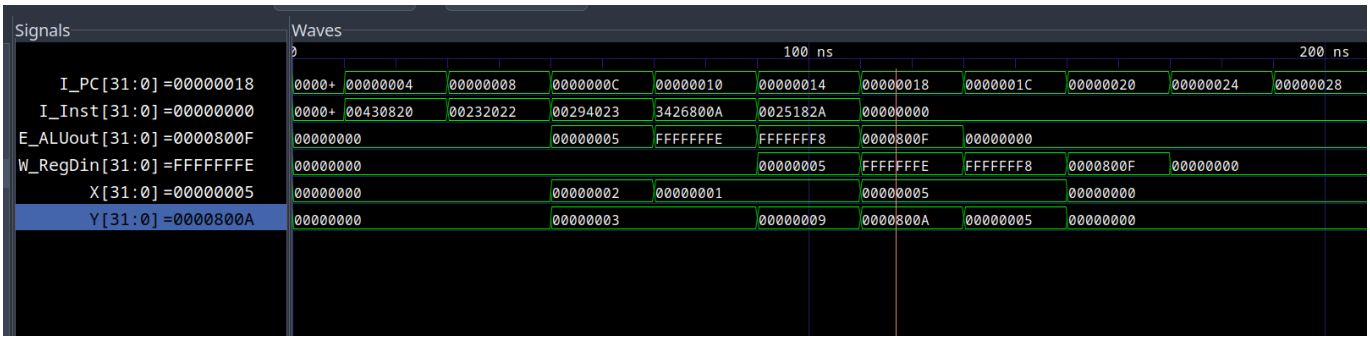


于是我们可以在 ID 段获取 busA 时，检测正在写回的寄存器 Rw，若是所取的 Ra 与 Rw 相等，那么可以将 busA 修改为 busW 的值。这个修改可以在传入 ID_Ex 寄存器时完成：

```
    ID_Ex id_ex (
        .Clk(Clk),
        .PC4(PC4), .Jtarg(Jtarg),
        .busA(Rs == W_Rw ? W_RegDin : busA), .busB(busB),
        .func(func), .immd(immd),
        ......
```

2.2 仿真波形（2）





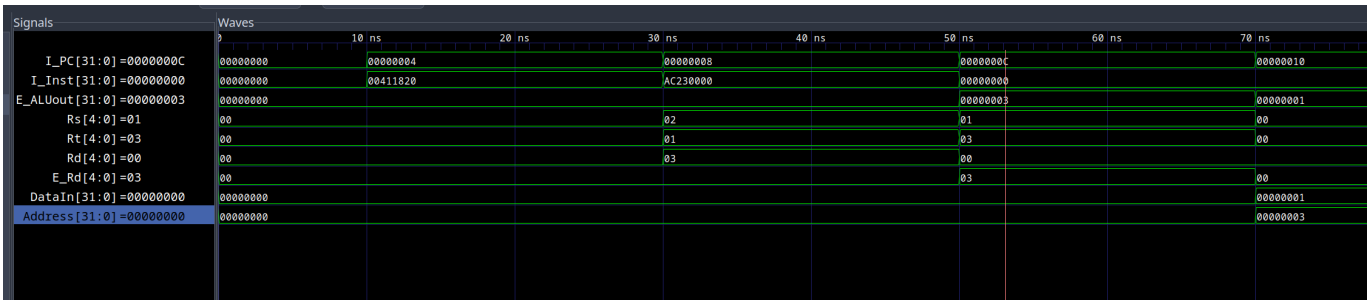
2.3 结果分析（6）

通过使用这种策略，可以看到在相同的位置，传入 ALU 的 X 值由 $1 原本的值 1 变更为了即将写入的 5。说明了在 ID 段检测取出的寄存器与即将写入的寄存器，并根据其结果修改传入 ID_Ex 寄存器的值的方法，可以简单并有效地处理第一条与第四条指令之间的数据冒险

3.实验4 - 选择习题 a（10）

3.1 修改关键代码段及文字说明（4）

还是先仿真看一看

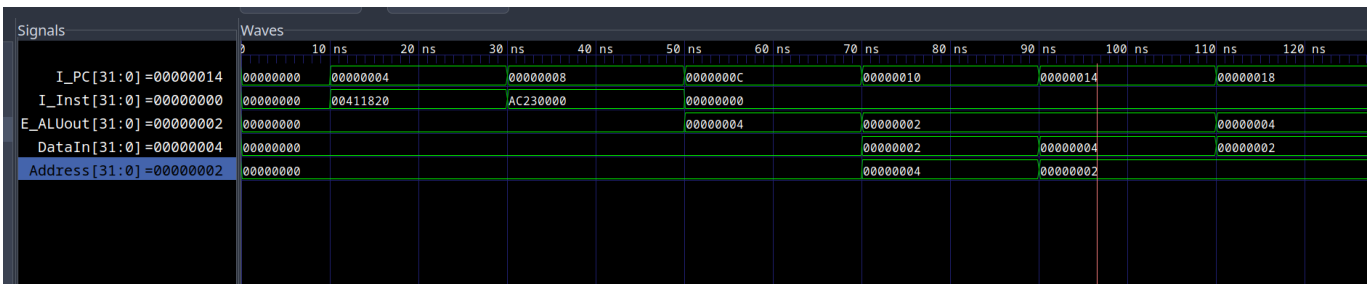


可以看到，在执行 sw 的 ID 阶段的时候，我们可以利用上一条指令的 ID 段所传递的 E_Rd 来判断是否需要进行数据转发。此时，上一条指令正在执行 Ex 段，可以产生出运算结果 E_ALUout，若上一条指令的 E_Rd 和 sw 的目的寄存器的 Rt 相等，那么我们可以将 ALU 的结果转发

```
    ID_Ex_lab4_2 id_ex (
        .Clk(Clk),
        .Clrn(not_load_use),
        .PC4(PC4), .Jtarg(Jtarg),
        .busA(busA), .busB((Rt == E_Rd) ? E_ALUout : busB),
        .func(func), .immd(immd),
```

3.2 仿真波形（2）

因为 $1 + $2 的结果与 $3 的原值一样，所以这里将 $1 初始化为 2



3.3 结果分析（4）

可以看到，在 sw 指令的 Mem 段，DataIn 的值为 4，是为第一条指令 $1 + $2 的值，地址值为 2，是 0($1) 的值。可以看到，这样的硬件结构的改变成功解决了给定指令序列的数据冒险问题