中山大学数据科学与计算机学院本科生实验报告

（2019 学年秋季学期）

课程名称：**计算机组成原理实验**　　　任课教师：　郭雪梅　　　助教：汪庭葳、刘洋旗

年级&班级	2018 级教务一班	专业(方向)	**计算机类**
学号	18340045	姓名	**关雅雯**
电话	13425512729	Email	18718093292@sina.cn
开始日期	2019/12/13	完成日期	2019/12/20

一、实验题目

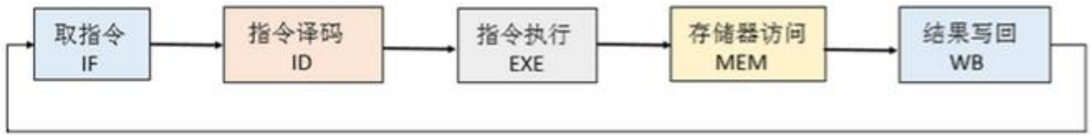
流水线 CPU

二、实验目的

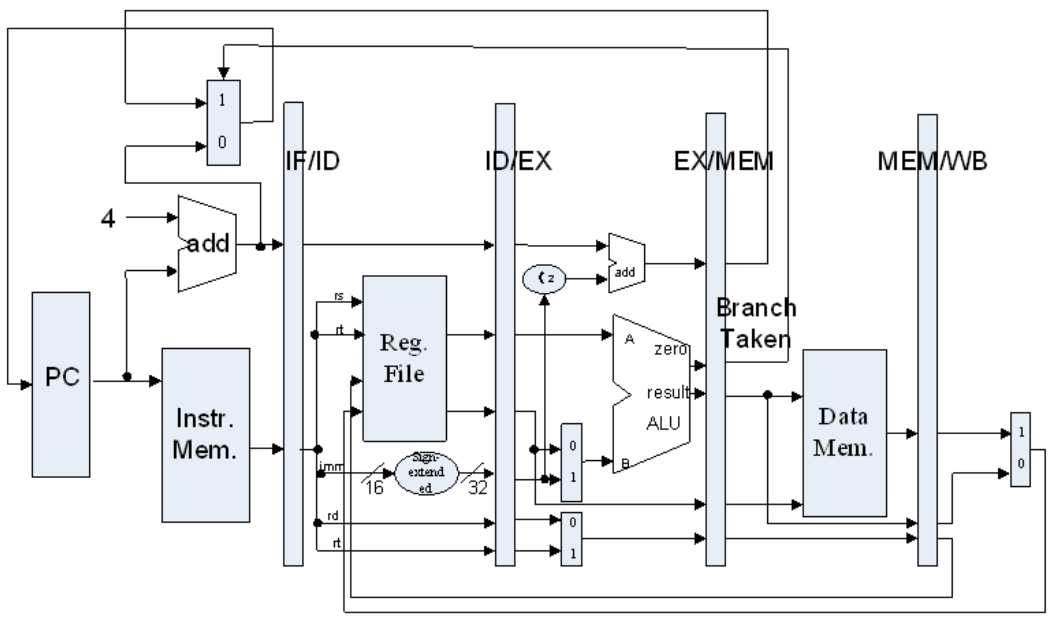
1) 了解流水线 CPU 基本功能部件的设计与实现方法，
2) 了解提高 CPU 性能的方法。
3) 掌握流水线 MIPS 微处理器的工作原理。
4) 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。
5) 掌握流水线 MIPS 微处理器的测试方法。

三、实验原理

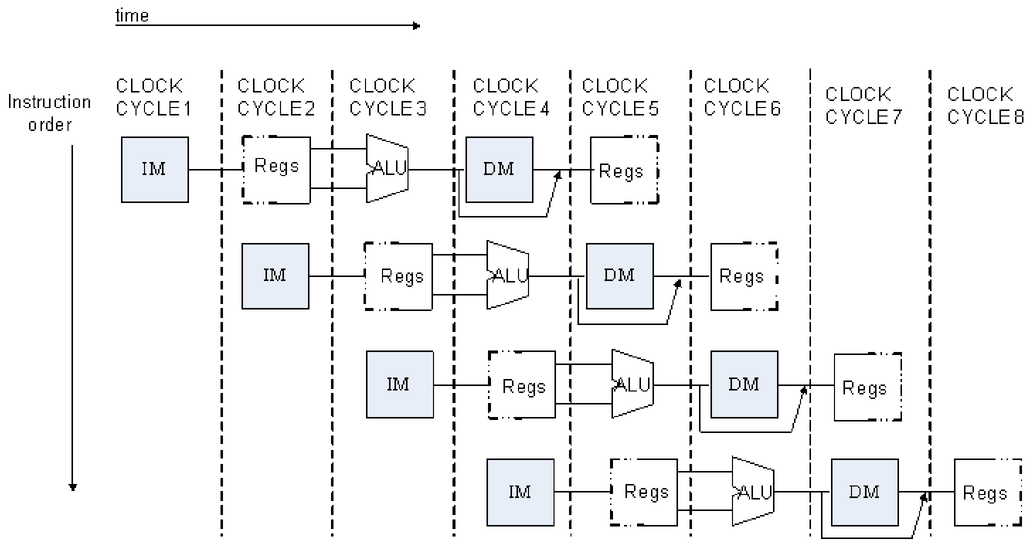
流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于现代 CPU 的架构中。根据 MIPS 处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器会写（WB）五级，对应多周期的五个处理阶段。一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。



流水线设计参考如下结构图:

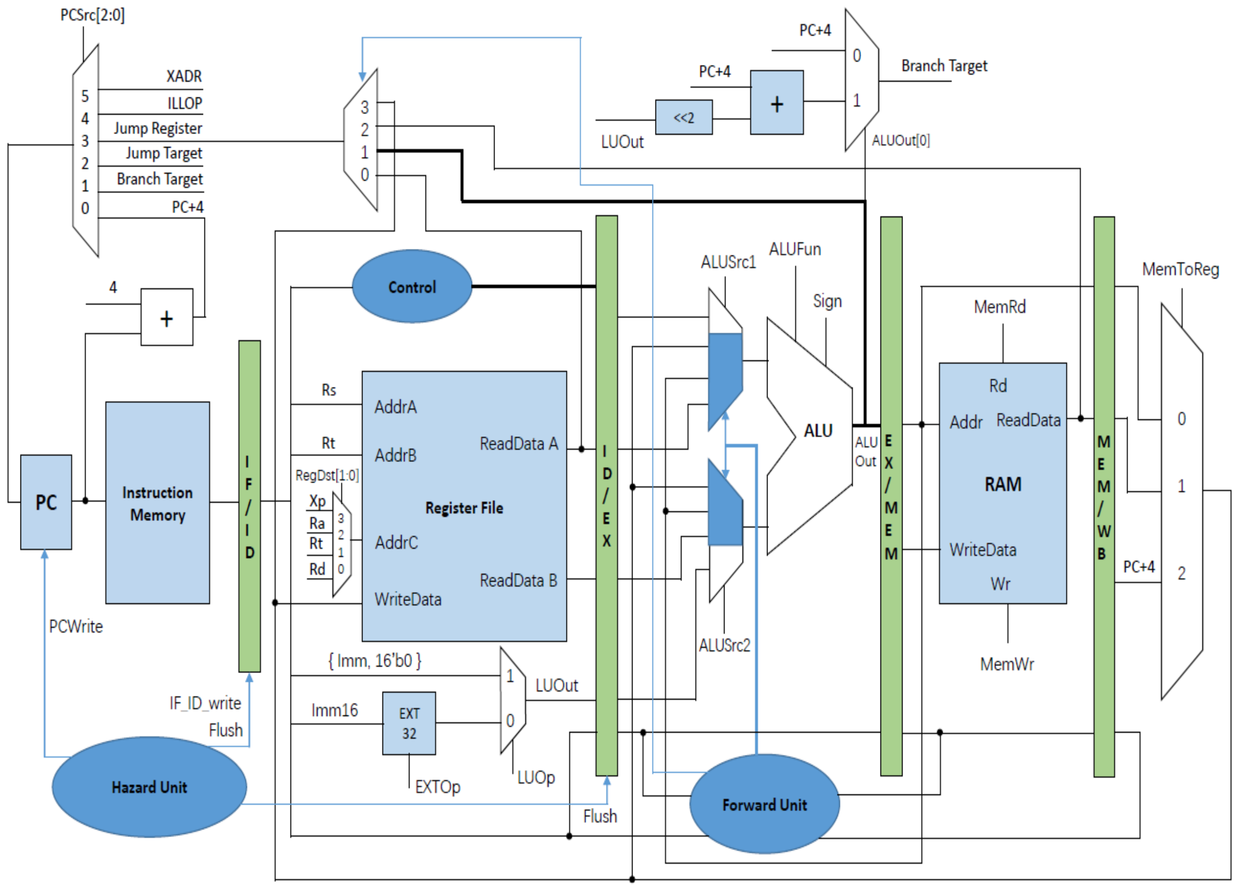


指令执行过程参考图:



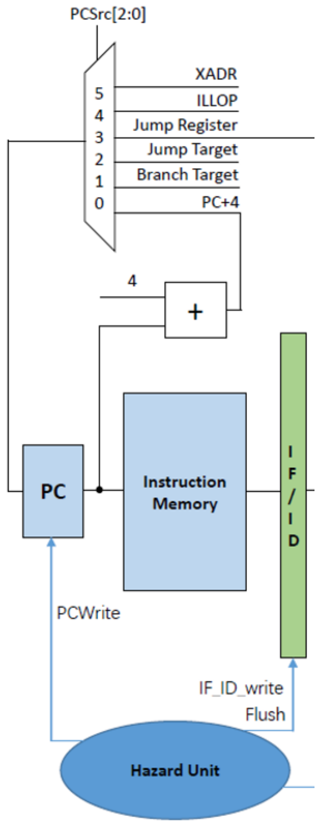
四、 实验内容

流水线实现的数据通路:



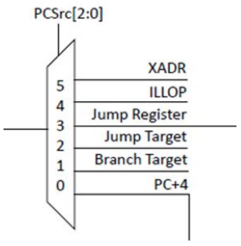
一、设计

1. 取指令部分（IF）



（1）PC 模块

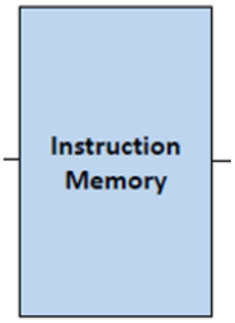
在 PCWrite（PC 的写使能信号）=1 的情况下，根据 PCSrc（PC 跳转控制信号）更新 PC。



（代码中未实现跳转到 ILLOP 与 XADR，不处理 exception 状况）

```verilog
assign IF_PCSrc = (EX_PCSrc == 3'b001 && EX_ALUOut[0]) ? 3'b001 : (ID_PCSrc == 3'b001
? 3'b000 : ID_PCSrc);

assign IF_PC_add = IF_PC + 32'h4;
assign IF_PC_add_4 = {IF_PC[31], IF_PC_add[30:0]};

always@(posedge clk or negedge reset)
    if(~reset)
        IF_PC <= 32'h0000_0000;         //PC begins at 0x0000_0000
    else if(PCWrite)
        case(IF_PCSrc)
            3'h0: IF_PC <= IF_PC_add_4; //pc+4
            3'h1: IF_PC <= EX_BT;       //branch target
            3'h2: IF_PC <= ID_JT;       //jump target
            3'h3: IF_PC <= ForwardJData;//jump register
            default: IF_PC <= 32'h0000_0000;
        endcase
```

（2）Instruction Memory（ROM）模块（指令寄存器模块）

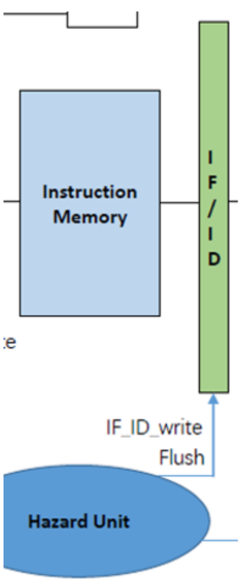


方便起见，此处直接将测试文件的指令对应的机器指令存储到代码中，使用 case 语句，将指令寄存器对应地址处的机器指令读出。

需要注意的是，代码模拟了 32 位的 ROM，而传入的地址是 32 位的位地址，要除以 4（也就是用高 30 位）作为 ROM 的字地址。
由于测试指令一共只有 26 条，故只采用了地址的第 8~2 位已足够。

```verilog
`timescale 1ns/1ps

module ROM (addr,data);
input [31:0] addr;
output [31:0] data;

reg [31:0] data;
localparam ROM_size = 32;
reg [31:0] ROM_data[ROM_size-1:0];

always@(*)
    case(addr[8:2]) //addr : byte address -> addr[:2] word beginning address
        0: data <= 32'b00100100000000010000000000001000;
        1: data <= 32'b00110100000000010000000000000010;
        2: data <= 32'b00000000010000010001100000100000;
        3: data <= 32'b00000000011000100010100000100010;
        4: data <= 32'b00000000101000100010000000100100;
        5: data <= 32'b00000000100000100100000000100101;
        6: data <= 32'b00000000000001000010000001000000;
        7: data <= 32'b00010101000000001111111111111110;
        8: data <= 32'b00101000010001100000000000000100;
        9: data <= 32'b00101100110001110000000000000000;
        10: data <= 32'b00100100111001110000000000001000;
        11: data <= 32'b00010000111000011111111111111110;
        12: data <= 32'b10101100001000100000000000000100;
        13: data <= 32'b10001100001010010000000000000100;
        14: data <= 32'b00100101001010010000000000000000;
        15: data <= 32'b00100100000010101111111111111110;
        16: data <= 32'b00100101010010100000000000000001;
        17: data <= 32'b00110000010010110000000000000010;
        18: data <= 32'b00000000000000100101000000100001;
        19: data <= 32'b00000001010000100101000000100011;
        20: data <= 32'b00000001000000100101000000100110;
        21: data <= 32'b00000001000000100101000000100111;
        22: data <= 32'b00000001000000100101000000101011;
        23: data <= 32'b00000000000010000101000001000010;
        24: data <= 32'b00100000000010101111111111111111;
        25: data <= 32'b00000000000010100101100001000011;
        26: data <= 32'b11111100000000000000000000000000;
      default: data <= 32'h0000_0000;
   endcase
endmodule
```

（3）IF_ID_Register 模块（IF_ID 寄存器模块）



输入信号：

clk（时钟信号）
reset（复位信号）
IF_ID_flush（IF_ID 寄存器清空的控制信号）
ID_EX_flush（ID_EX 寄存器清空的控制信号）
IF_ID_write(IF_ID 寄存器的写信号)
PC_add_4_in（输入的 PC+4 信号）
Instruction_in（输入的指令信号）

输出信号：

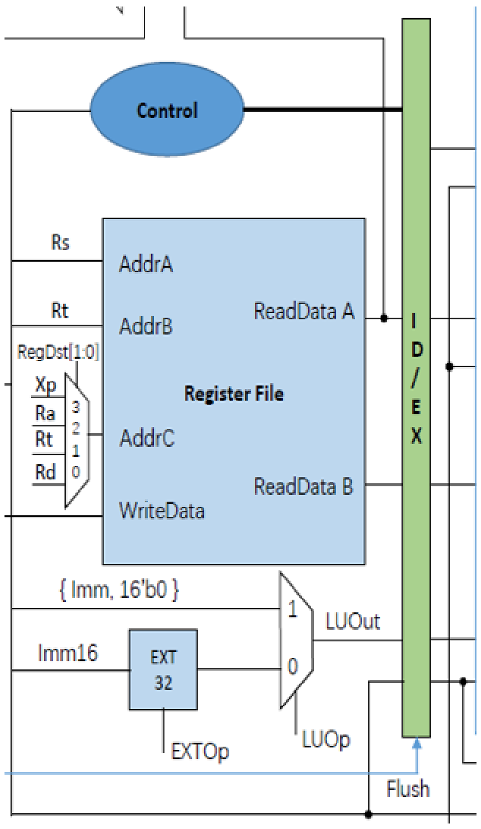
PC_add_4_out（输出的 PC+4 信号）
Instruction_out（输出的指令信号）

中断的实现是强行将 ID 段的执行指令的控制信号改为中断的控制信号，同时将当前指令的 PC+4 存入对应寄存器。而当中断恰好遇到一个 flush 的信号时，我们需要保存的不是当前这条指令对应的 PC+4(已经被 flush 了)，而是更前面的那条，对应的情况有三种。

1. ID/EX.flush == 1:   ID/EX.PCadd4 -= 4
2. IF/ID.flush == 1:   IF/ID.PCadd4 -= 4
3. ID/EX.flush == 1 && ID/EX.flush == 1:   IF/ID.PCadd4 -= 8 ID/EX.PCadd4 -= 4

```
module IF_ID_Reg(
    clk, reset, IF_ID_flush, ID_EX_flush, IF_ID_write,
    PC_add_4_in, Instruct_in,PC_add_4_out, Instruct_out);

//input
input clk;
input reset;
input IF_ID_flush;
input ID_EX_flush;
input IF_ID_write;
input [31:0] PC_add_4_in;
input [31:0] Instruct_in;
//output
output reg [31:0] PC_add_4_out;
output reg [31:0] Instruct_out;

always @(posedge clk or negedge reset) begin
    if (~reset) begin
        PC_add_4_out <= 32'h0000_0000;
        Instruct_out <= 32'h0000_0000;
    end
    else begin
        if(IF_ID_flush) begin
            if(ID_EX_flush) begin
                PC_add_4_out <= PC_add_4_in - 8;
            end
            else begin
                PC_add_4_out <= PC_add_4_in - 4;
            end
            Instruct_out <= 32'h0000_0000;
        end
        else begin
            if (IF_ID_write) begin
                PC_add_4_out <= PC_add_4_in;
                Instruct_out <= Instruct_in;
            end
        end
    end
end

endmodule
```

2. 指令译码部分（ID）



（1）控制模块（control）

输入信号：
Instruct（指令信号）

输出信号：
PCSrc（PC 跳转控制信号）
RegWr（寄存器写使能信号）
RegDst（Rd 控制信号）
MemRd（数据存储器读信号）
MemWr（数据存储器写信号）
MemToReg（数据存储器写到寄存器的控制信号）
ALUSrc1（ALU 第一个运算数控制信号）
ALUSrc2（ALU 第二个运算数控制信号）
EXTOp（立即数拓展控制信号）
LUOp（LUI 控制信号）
ALUFun（ALU 运算功能控制信号）
Sign（ALU 运算是否有符号数控制信号）

由于代码较长，分三栏显示如下：

```
module Control(Instruct, PCSrc, RegWr, R
egDst, MemRd, MemWr, MemToReg, ALUSrc1,
ALUSrc2, EXTOp, LUOp, ALUFun, Sign);
input [31:0] Instruct;
output reg [2:0] PCSrc;
output reg RegWr;
output reg [1:0] RegDst;
output reg MemRd;
output reg MemWr;
output reg [1:0] MemToReg;
output reg ALUSrc1;
output reg ALUSrc2;
output reg EXTOp;
output reg LUOp;
output reg [5:0] ALUFun;
output reg Sign;

always @(*) begin
    case(Instruct[31:26])
        6'b10_0011: begin   //lw
            PCSrc = 3'd0;
            RegWr = 1;
```

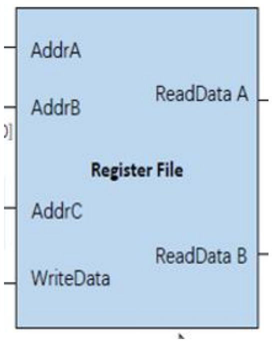
```
            RegDst = 2'd1;
            MemRd = 1;
            MemWr = 0;
            MemToReg = 2'd1;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b10_1011: begin   //sw
            PCSrc = 3'd0;
            RegWr = 0;
            RegDst = 2'd0;
            MemRd = 0;
            MemWr = 1;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b000000;
            Sign = 1;
```

```
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_1111: begin   //lui
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 1;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b011110;
            Sign = 1;
            EXTOp = 1;
            LUOp = 1;
        end
        6'b00_1000: begin   //addi
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_1001: begin   //addiu
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_1100: begin   //andi
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b011000;
            Sign = 1;
            EXTOp = 0;
            LUOp = 0;
        end

        6'b00_1101: begin   //ori
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b011110;
            Sign = 1;
            EXTOp = 0;
            LUOp = 0;
        end

        6'b00_1010: begin   //slti
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b110101;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_1011: begin   //sltiu
            PCSrc = 3'd0;
            RegWr = 1;
            RegDst = 2'd1;
            MemRd = 0;
```

```
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 1;
            ALUFun = 6'b110101;
            Sign = 0;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_0100: begin   //beq
            PCSrc = 3'd1;
            RegWr = 0;
            RegDst = 2'd0;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 0;
            ALUFun = 6'b110011;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_0101: begin   //bne
            PCSrc = 3'd1;
            RegWr = 0;
            RegDst = 2'd0;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 0;
            ALUFun = 6'b110001;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_0010: begin   //j
            PCSrc = 3'd2;
            RegWr = 0;
            RegDst = 2'd0;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 0;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_0011: begin   //jal
            PCSrc = 3'd2;
            RegWr = 1;
            RegDst = 2'd2;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd2;
            ALUSrc1 = 0;
            ALUSrc2 = 0;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
        6'b00_0000: begin   //R 型 0x00
            case(Instruct[5:0])
                6'b10_0000: begin   //ad
d
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
```

```
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0001: begin   //ad
du
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0010: begin   //su
b
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000001;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0011: begin   //su
bu
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000001;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0100: begin   //an
d
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b011000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0101: begin   //or
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b011110;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0110: begin   //xor
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b010110;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_0111: begin   //nor
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b010001;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b00_0000: begin   //sll
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 1;
                    ALUSrc2 = 0;
                    ALUFun = 6'b100000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b00_0010: begin   //srl
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 1;
                    ALUSrc2 = 0;
                    ALUFun = 6'b100001;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b00_0011: begin   //sra
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 1;
                    ALUSrc2 = 0;
                    ALUFun = 6'b100011;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_1010: begin   //slt
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b110101;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b10_1011: begin   //sltu
                    PCSrc = 3'd0;
                    RegWr = 1;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b110101;
                    Sign = 0;
                    EXTOp = 1;
                    LUOp = 0;
                end
                6'b00_1000: begin   //jr
                    PCSrc = 3'd3;
                    RegWr = 0;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;
                end
                default: begin
                    PCSrc = 3'd0;
                    RegWr = 0;
                    RegDst = 2'd0;
                    MemRd = 0;
                    MemWr = 0;
                    MemToReg = 2'd0;
                    ALUSrc1 = 0;
                    ALUSrc2 = 0;
                    ALUFun = 6'b000000;
                    Sign = 1;
                    EXTOp = 1;
                    LUOp = 0;

                end
            endcase
        end
        default: begin
            RegWr = 0;
            RegDst = 2'd0;
            MemRd = 0;
            MemWr = 0;
            MemToReg = 2'd0;
            ALUSrc1 = 0;
            ALUSrc2 = 0;
            ALUFun = 6'b000000;
            Sign = 1;
            EXTOp = 1;
            LUOp = 0;
        end
    endcase
end

endmodule
```

（2）寄存器堆模块（RegFile）



输入信号：

reset（复位信号）
clk（时钟信号）
wr（寄存器写使能信号）
addr1（rs 地址信号）
addr2（rt 地址信号）

addr3（rd 地址信号）
data3（要写入 rd 的数据，由 WB 提供）

输出信号：

data1（rs 数据信号）
data2（rt 数据信号）

值得注意的一点是，写使能信号为 1 时，还要判断是否写入 0 寄存器，0 寄存器的值永远只能为 0

```
`timescale 1ns/1ps

module RegFile (
    reset, clk, addr1, data1,
    addr2, data2, wr, addr3, data3);

input reset,clk;
input wr; // write enable
input [4:0] addr1,addr2,addr3;
output [31:0] data1,data2;
input [31:0] data3;

reg [31:0] RF_data[31:0];

assign data1 = RF_data[addr1];
assign data2 = RF_data[addr2];

integer i;

always@(negedge reset or negedge clk) begin
    if(~reset) begin
        for(i = 0; i < 32; i = i + 1)  RF_data[i] <= 32'b0;
    end
    else begin
        // write enable == 1  &&  addr3 != 0
        // $0 MUST be all zeros
        if(wr && (|addr3))
            RF_data[addr3] <= data3;
    end
end
endmodule
```

（3）ID_EX 寄存器堆模块（ID_EX_Reg）

输入信号：

```
input clk;//时钟信号
input reset;//复位信号
input ID_EX_flush;//ID_EX 寄存器清空信号
input [31:0] PC_add_4_in;//输入的 PC_add_4
input [31:0] DataBusA_in;//输入的 Rs 数据
input [31:0] DataBusB_in;//输入的 Rt 数据
input [31:0] LUOut_in;//输入的 LUOut
input [4:0] Rs_in;//Rs 地址
input [4:0] Rt_in;//Rt 地址
input [4:0] Rd_in;//Rd 地址
input [4:0] Shamt_in;//位移信号
input [1:0] RegDst_in;//Rt 或 Rd 的选择信号
input [2:0] PCSrc_in;//PC 跳转控制信号
input MemRead_in;//数据存储器读使能信号
input MemWrite_in;//数据存储器写使能信号
input [1:0] MemToReg_in;
input [5:0] ALUFun_in;
input ALUSrc1_in;
input ALUSrc2_in;
input RegWrite_in;
input Sign_in;
```

输出信号：

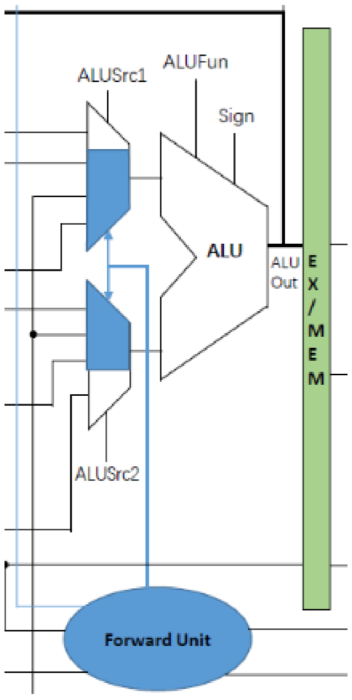
```
output reg [31:0] PC_add_4_out;
output reg [31:0] DataBusA_out;
output reg [31:0] DataBusB_out;
output reg [31:0] LUOut_out;

output reg [4:0] Rs_out;
output reg [4:0] Rt_out;
output reg [4:0] Rd_out;
output reg [4:0] Shamt_out;
output reg [1:0] RegDst_out;
output reg [2:0] PCSrc_out;
output reg MemRead_out;
output reg MemWrite_out;
output reg [1:0] MemToReg_out;
output reg [5:0] ALUFun_out;
output reg ALUSrc1_out;
output reg ALUSrc2_out;
output reg RegWrite_out;
output reg Sign_out;
```

同 IF_ID_Reg 的实现，若需要清空 ID_EX 寄存器，则 PC_add_4_out = PC_add_4_in – 4（返回上一条指令重新执行）

```
module ID_EX_Reg(
    clk,            reset,
    ID_EX_flush,    PC_add_4_in,
    DataBusA_in,    DataBusB_in,
    LUOut_in,       Rs_in,
    Rt_in,          Rd_in,
    Shamt_in,       RegDst_in,
    PCSrc_in,       MemRead_in,
    MemWrite_in,    MemToReg_in,
    ALUFun_in,      ALUSrc1_in,
    ALUSrc2_in,     RegWrite_in,
    Sign_in,        PC_add_4_out,
    DataBusA_out,   DataBusB_out,
    LUOut_out,      Rs_out,
    Rt_out,         Rd_out,
    Shamt_out,      RegDst_out,
    PCSrc_out,      MemRead_out,
    MemWrite_out,   MemToReg_out,
    ALUFun_out,     ALUSrc1_out,
    ALUSrc2_out,    RegWrite_out,
```

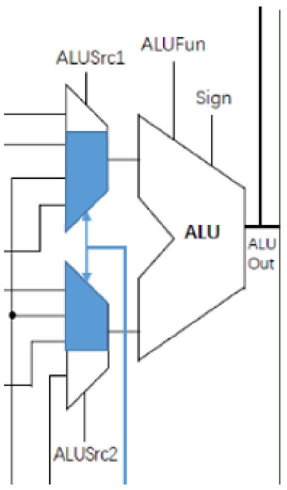
```
    Sign_out);

input clk;//时钟信号
input reset;//复位信号
input ID_EX_flush;//ID_EX 寄存器清空信号
input [31:0] PC_add_4_in;
input [31:0] DataBusA_in;//输入的 Rs 数据
input [31:0] DataBusB_in;//输入的 Rt 数据
input [31:0] LUOut_in;//输入的 LUOut

input [4:0] Rs_in;//Rs 地址
input [4:0] Rt_in;//Rt 地址
input [4:0] Rd_in;//Rd 地址
input [4:0] Shamt_in;//位移信号
input [1:0] RegDst_in;
input [2:0] PCSrc_in;//PC 跳转控制信号
input MemRead_in;//数据存储器读使能信号
input MemWrite_in;//数据存储器写使能信号
input [1:0] MemToReg_in;
input [5:0] ALUFun_in;
input ALUSrc1_in;
input ALUSrc2_in;
input RegWrite_in;
input Sign_in;

output reg [31:0] PC_add_4_out;
output reg [31:0] DataBusA_out;
output reg [31:0] DataBusB_out;
output reg [31:0] LUOut_out;

output reg [4:0] Rs_out;
output reg [4:0] Rt_out;
output reg [4:0] Rd_out;
output reg [4:0] Shamt_out;
output reg [1:0] RegDst_out;
output reg [2:0] PCSrc_out;
output reg MemRead_out;
output reg MemWrite_out;
output reg [1:0] MemToReg_out;
output reg [5:0] ALUFun_out;
output reg ALUSrc1_out;
output reg ALUSrc2_out;
output reg RegWrite_out;
output reg Sign_out;


always @(posedge clk or negedge reset) begin
    if (~reset) begin
            PC_add_4_out <= 32'h0000_0000;
            DataBusA_out <= 32'h0000_0000;
            DataBusB_out <= 32'h0000_0000;
            LUOut_out <= 32'h0000_0000;
            Rs_out <= 5'h00;
            Rt_out <= 5'h00;
            Rd_out <= 5'h00;
            Shamt_out <= 5'h00;
            RegDst_out <= 2'h0;
            PCSrc_out <= 3'h0;
            MemRead_out <= 1'h0;
            MemWrite_out <= 1'h0;
            MemToReg_out <= 2'h0;
            ALUFun_out <= 6'h00;
            ALUSrc1_out <= 1'h0;
            ALUSrc2_out <= 1'h0;
            RegWrite_out <= 1'h0;
            Sign_out <= 1'h0;
    end
    else begin
        if(ID_EX_flush) begin
            PC_add_4_out <= PC_add_4_in - 4;
            DataBusA_out <= 32'h0000_0000;
            DataBusB_out <= 32'h0000_0000;
            LUOut_out <= 32'h0000_0000;
            Rs_out <= 5'h00;
            Rt_out <= 5'h00;
            Rd_out <= 5'h00;
            Shamt_out <= 5'h00;
            RegDst_out <= 2'h0;
            PCSrc_out <= 3'h0;
            MemRead_out <= 1'h0;
            MemWrite_out <= 1'h0;
            MemToReg_out <= 2'h0;
            ALUFun_out <= 6'h00;
            ALUSrc1_out <= 1'h0;
            ALUSrc2_out <= 1'h0;
            RegWrite_out <= 1'h0;
            Sign_out <= 1'h0;
        end
        else begin
            PC_add_4_out <= PC_add_4_in;
            DataBusA_out <= DataBusA_in;
            DataBusB_out <= DataBusB_in;
            LUOut_out <= LUOut_in;
            Rs_out <= Rs_in;
            Rt_out <= Rt_in;
            Rd_out <= Rd_in;
            Shamt_out <= Shamt_in;
            RegDst_out <= RegDst_in;
            PCSrc_out <= PCSrc_in;
            MemRead_out <= MemRead_in;
            MemWrite_out <= MemWrite_in;
            MemToReg_out <= MemToReg_in;
            ALUFun_out <= ALUFun_in;
            ALUSrc1_out <= ALUSrc1_in;
            ALUSrc2_out <= ALUSrc2_in;
            RegWrite_out <= RegWrite_in;
            Sign_out <= Sign_in;
        end
    end
end

endmodule
```

3. 指令执行部分（EX）



（1）ALU 模块

ALU 模块直接沿用单周期 CPU 的 ALU 模块



其中，ALUFun 的编码如下：

000000: add	100000: sll
000001: sub	100001: srl
010001: nor	100011: sra
010110: xor	110101: slt
011110: or(lui)	110011: beq
011000: and	110001: bne

ALU 模块中使用了加减、比较、逻辑运算、位移操作。

```
module ALU(A, B, ALUFun, Sign, Z);

input Sign;
input [5:0] ALUFun;
input [31:0] A, B;
output [31:0] Z;//Z: zero label
wire zero,neg;
wire [31:0] S0, S1, S2, S3;

AddSub AddSub(.A(A), .B(B), .ALUFun(ALUFun), .Sign(Sign), .Z(zero), .N(neg), .S(S0));
Cmp Cmp(.ALUFun(ALUFun), .Sign(Sign), .Z(zero), .N(neg), .S(S1));
Logic Logic(.A(A), .B(B), .ALUFun(ALUFun), .S(S2));
Shift Shift(.A(A), .B(B), .ALUFun(ALUFun), .S(S3));

// Choose the type of calculation
assign Z =
(ALUFun[5:4] == 2'b00)? S0:
(ALUFun[5:4] == 2'b10)? S3:
(ALUFun[5:4] == 2'b01)? S2: S1;

endmodule
```

其中，加减实现如下：

```
// add or sub calculation
module AddSub(A, B, ALUFun, Sign, Z, N, S);
input Sign;
input [5:0] ALUFun;
input [31:0] A, B;
output Z, N;        //N = 1 : negative, sub; 0 : positive, add
output [32:0] S;    //S : result

// Choose which number to compare when determining Z
assign Z =
(ALUFun[3] && |A)? 0:
(~ALUFun[3] && |S)? 0: 1;

// Choose to perform add or sub
assign S =
(ALUFun[0])? ({1'b0, A} - {1'b0, B}): ({1'b0, A} + {1'b0, B});

// Determine N according to the Sign signal and carryin
assign N =
(ALUFun[3] && Sign && A[31])? 1:
(~ALUFun[3] && Sign && S[31])? 1:
(~ALUFun[3] && ~Sign && S[32])? 1: 0;

endmodule
```

比较运算实现如下：

```
// comparation calculation
module Cmp(ALUFun, Sign, Z, N, S);
input Sign, Z, N;
input [5:0] ALUFun;
output [31:0] S;

// Determine output according to N and Z
assign S[0] =
(ALUFun[3:1] == 3'b001)? Z:
(ALUFun[3:1] == 3'b000)? ~Z:
(ALUFun[3:1] == 3'b010)? N:
(ALUFun[3:1] == 3'b110)? (N || Z):
(ALUFun[3:1] == 3'b101)? N: (~N && ~Z);
```

```
assign S[31:1]=0;

endmodule
```

逻辑运算实现如下：

```
// logical calculation
module Logic(A, B, ALUFun, S);
input [5:0] ALUFun;
input [31:0] A, B;
output [31:0] S;

// result of logical calculation
assign S =
(ALUFun[3:0] == 4'b0001)? ~(A | B):
(ALUFun[3:0] == 4'b1110)? (A | B):
(ALUFun[3:0] == 4'b1000)? (A & B):
(ALUFun[3:0] == 4'b0110)? (A ^ B): A;

endmodule
```

位移操作运算实现如下：

```
// shift calculation
module Shift(A, B, ALUFun, S);
input [5:0] ALUFun;
input [31:0] A, B;
output [31:0] S;

assign S =

// srl or sra with the msb being '0'
(ALUFun[1:0] == 2'b01 || (ALUFun[1:0] == 2'b11 && B[31] == 0))? B >> A[4:0]:

// sra with the msb being '1'
(ALUFun[1:0] == 2'b11 && B[31] == 1)? ({32'hFFFFFFFF, B} >> A[4:0]):

// sll
(ALUFun[1:0] == 2'b00)? B << A[4:0]: 0;

endmodule
```

（2）EX_MEM 寄存器模块（EX_MEM_Reg）

EX_MEM_Reg 的实现类似于 ID_EX_Reg，由于没有清空信号，故少了需要清空信号的情况，又因为需要传递的信号数量减少了，所以实现起来较为简洁。

输入信号：

clk;
reset;
PC_add_4_in;
DataBusB_in;
ALUOut_in;
Rt_in;
Rd_in;
RegDst_in;
MemRead_in;
MemWrite_in;
MemToReg_in;
RegWrite_in;
AddrC_in;

输出信号：

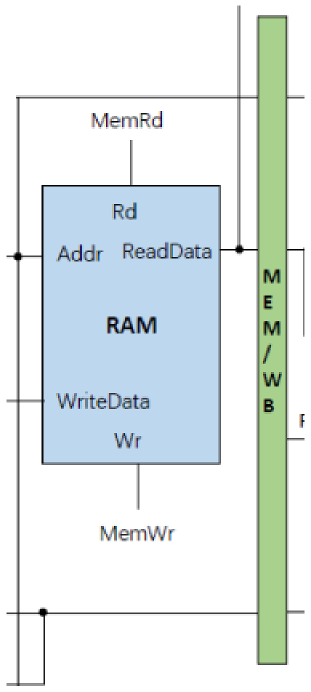
PC_add_4_out;
DataBusB_out;
ALUOut_out;

Rt_out;
Rd_out;
RegDst_out;
MemRead_out;
MemWrite_out;
MemToReg_out;
RegWrite_out;
AddrC_out;

每个信号的含义同 ID_EX_Reg 处的含义。

```
module EX_MEM_Reg(
    clk ,reset, PC_add_4_in,
    ALUOut_in, DataBusB_in,
    Rt_in, Rd_in, RegDst_in,
    MemRead_in, MemWrite_in, MemToReg_in,
    RegWrite_in, AddrC_in, PC_add_4_out,
    ALUOut_out, DataBusB_out,
    Rt_out, Rd_out, RegDst_out,
    MemRead_out, MemWrite_out, MemToReg_out,
    RegWrite_out, AddrC_out);

input clk;
input reset;
input [31:0] PC_add_4_in;
input [31:0] DataBusB_in;
input [31:0] ALUOut_in;

input [4:0] Rt_in;
input [4:0] Rd_in;
input [1:0] RegDst_in;
input MemRead_in;
input MemWrite_in;
input [1:0] MemToReg_in;
input RegWrite_in;
input [4:0] AddrC_in;

output reg [31:0] PC_add_4_out;
output reg [31:0] DataBusB_out;
output reg [31:0] ALUOut_out;

output reg [4:0] Rt_out;
output reg [4:0] Rd_out;
output reg [1:0] RegDst_out;
output reg MemRead_out;
output reg MemWrite_out;
output reg [1:0] MemToReg_out;
output reg RegWrite_out;
output reg [4:0] AddrC_out;

always @(posedge clk or negedge reset) begin
    if (~reset) begin
        PC_add_4_out <= 32'h0000_0000;
        DataBusB_out <= 32'h0000_0000;
        ALUOut_out <= 32'h0000_0000;
        Rt_out <= 5'h00;
        Rd_out <= 5'h00;
        RegDst_out <= 2'h0;
        MemRead_out <= 1'b0;
        MemWrite_out <= 1'b0;
        MemToReg_out <= 2'h0;
        RegWrite_out <= 1'b0;
        AddrC_out <= 5'h0;
    end
    else begin
        PC_add_4_out <= PC_add_4_in;
        DataBusB_out <= DataBusB_in;
        ALUOut_out <= ALUOut_in;
        Rt_out <= Rt_in;
        Rd_out <= Rd_in;
        RegDst_out <= RegDst_in;
        MemRead_out <= MemRead_in;
        MemWrite_out <= MemWrite_in;
        MemToReg_out <= MemToReg_in;
        RegWrite_out <= RegWrite_in;
        AddrC_out <= AddrC_in;
    end
end
endmodule
```

4. 存储器访问部分（MEM）



（1）数据存储器模块（RAM/DataMem）

输入信号：

reset（复位信号）
clk（时钟信号）
rd（读使能信号）
wr（写使能信号）
addr（地址）
wdata（写入数据）

输出信号：

rdata（读出数据）

```
`timescale 1ns/1ps

module DataMem (reset,clk,rd,wr,addr,wdata,rdata);
input reset,clk;
input rd,wr;
input [31:0] addr;  //word address
input [31:0] wdata;
output [31:0] rdata;

parameter RAM_SIZE = 256;
reg [31:0] RAMDATA [RAM_SIZE-1:0];

assign rdata = (rd && (addr[31:2] < RAM_SIZE)) ? RAMDATA[addr[31:2]] : 32'b0;

always@(posedge clk) begin
    if(wr && (addr[31:2] < RAM_SIZE))
        RAMDATA[addr[31:2]] <= wdata;
end

endmodule
```

（2）MEM_WB 寄存器模块（MEM_WB_Reg）

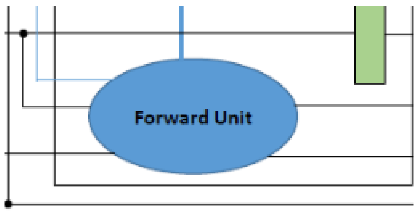
类似 EX_MEM_Reg，不再赘述。

```
module MEM_WB_Reg(clk, reset, PC_add_4_in, ALUOut_in, MemReadData_in, Rt_in, Rd_in, Re
gDst_in, MemToReg_in, RegWrite_in,
    AddrC_in, PC_add_4_out, ALUOut_out, MemReadData_out, Rt_out, Rd_out, RegDst_out, M
emToReg_out, RegWrite_out, AddrC_out);

input clk;
input reset;
input [31:0] PC_add_4_in;
input [31:0] ALUOut_in;
input [31:0] MemReadData_in;
input [4:0] Rt_in;
input [4:0] Rd_in;
input [1:0] RegDst_in;
input [1:0] MemToReg_in;
input RegWrite_in;
input [4:0] AddrC_in;

output reg [31:0] PC_add_4_out;
output reg [31:0] ALUOut_out;
output reg [31:0] MemReadData_out;
output reg [4:0] Rt_out;
output reg [4:0] Rd_out;
output reg [1:0] RegDst_out;
output reg [1:0] MemToReg_out;
output reg RegWrite_out;
output reg [4:0] AddrC_out;

always @(posedge clk or negedge reset) begin
    if (~reset) begin
        PC_add_4_out <= 32'h0000_0000;
        ALUOut_out <= 32'h0000_0000;
        MemReadData_out <= 32'h0000_0000;
        Rt_out <= 5'h00;
        Rd_out <= 5'h00;
        RegDst_out <= 2'h0;
        MemToReg_out <= 2'h0;
        RegWrite_out <= 1'h0;
        AddrC_out <= 5'h0;
    end
    else begin
        PC_add_4_out <= PC_add_4_in;
        ALUOut_out <= ALUOut_in;
        MemReadData_out <= MemReadData_in;
        Rt_out <= Rt_in;
        Rd_out <= Rd_in;
        RegDst_out <= RegDst_in;
        MemToReg_out <= MemToReg_in;
        RegWrite_out <= RegWrite_in;
        AddrC_out <= AddrC_in;
    end
end

endmodule
```

5. 结果写回部分（WB）

此部分在总体设计中完成。

6. 转发单元部分（Forward Unit）



最常见的冒险来自 EX 段，也就是 ALU 的操作数需要用到上一条或者倒数第二条的指令时可能产生的冒险，即有可能用到 EX/MEM 寄存器内的数据，或者是 MEM/WB 寄存器内的数据，判定伪码如下：

EX/MEM 段的转发需要判定的是前一条指令写入非$0 寄存器，且写入地址与需要读取的地址相等

```
if ( EX/MEM.RegWrite
and ( EX/MEM.RegisterRd != 0 )
and ( EX/MEM.RegisterRd == ID/EX.RegisterRs ))
ForwardA = 10
```

MEM/WB 段的转发需要增加一个判定，就是要求不能从 EX/MEM 就近转发

```
if ( MEM/WB.RegWrite
and ( MEM/WB.RegisterRd != 0 )
and not ( EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0 )
    and ( EX/MEM.RegisterRd != ID/EX.RegisterRd ))
and ( MEM/EX.RegisterRd == ID/EX.RegisterRs ))
ForwardA = 01
```

将上述的 Rs 全部改为 Rt 就得到了 ForwardB 的转发条件：

先获得控制信号

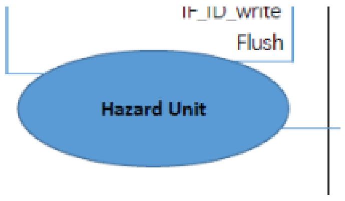
```
// ForwardA, strategy here same as the textbook
if(EX_MEM_RegWrite && EX_MEM_AddrC != 5'h00
    && EX_MEM_AddrC == ID_EX_Rs) begin
    ForwardA = 2'b10;
end
```

```
else if(MEM_WB_RegWrite && MEM_WB_AddrC != 5'h00
    && MEM_WB_AddrC == ID_EX_Rs) begin
    ForwardA = 2'b01;
end
else
    ForwardA = 2'b00;
// Only replace Rt with Rs of ForwardA and we get ForwardB
```
根据控制信号选择前传的数据

```
assign ForwardAData = (ForwardA==2'b00) ? EX_DataBusA :
    (ForwardA==2'b01) ? WB_DataBusC : MEM_ALUOut;
assign ForwardBData = (ForwardB==2'b00) ? EX_DataBusB :
    (ForwardB==2'b01) ? WB_DataBusC : MEM_ALUOut;
assign ForwardJData = (ForwardJ == 2'b00) ? ID_DataBusA :
    (ForwardJ == 2'b01) ? EX_ALUOut :
    (ForwardJ == 2'b10) ? WB_MemReadData : WB_DataBusC;
```

Forward Unit 总体设计代码附于文末。

7. 冒险单元部分（Hazard Unit）



冒险检测单元主要解决 load-use，beq 类指令和 jump 类指令的三种冒险。为了实现冒险控制，我们添加 PC.write 和 IF/ID.write 两个寄存器写入控制信号和 IF/ID.flush 和 ID/EX.flush 两个寄存器清空的控制信号。其中只有 write 信号为 1 时对应寄存器在上升沿才能被写入新值，一旦 flush 信号取 1，时钟上升沿时直接清空整个寄存器。我们通过这四个控制信号完成冒险控制。

load-use

当 ID/EX 段发现是 load 指令，且读取寄存器号与下一条指令的将要读取的寄存器号相同时，CPU 知道会触发 load-use 冒险，这意味着下一条必须 stall 一个周期，即我们要禁止 PC、IF/ID 寄存器的写入，同时下个上升沿到来后的 ID/EX 内的指令是不能用的，应该 flush 掉。伪码如下:

```
if ( ID/EX.MemRead && ( ID/EX.Rt == IF/ID.Rs || ID/EX.Rt == IF/ID.Rt))
    PC.write = 0
    IF/ID.write = 0
    IF/ID.flush = 0
    ID/EX.flush = 1
```
verilog 代码如下。

```
// Load use
```

```
if(ID_EX_MemRead && (ID_EX_Rt == IF_ID_Rs || ID_EX_Rt == IF_ID_Rt )) begin
    PCWrite_mark[2] = 1'b0;
    IF_ID_write_mark[2] = 1'b0;
    IF_ID_flush_mark[2] = 1'b0;
    ID_EX_flush_mark[2] = 1'b1;
end
```

beq 类冒险

这类冒险只有当判断到 Branch 结果时才能够知道，即在 EX 段才能被发现。一旦发现 Branch 将成功跳转，那么进入流水线的后两条指令都是不能用的，必须 flush 掉，伪码如下：

```
if ( EX.PCSrc == 1 && EX.ALUOut[0] == 1 )
    PC.write = 1
    IF/ID.write = 1
    IF/ID.flush = 1
    ID/EX.flush = 1
```

verilog 代码如下。

```
// Branch
if (EX_PCSrc == 3'd1 && EX_ALUOut_0) begin
    PCWrite_mark[0] = 1'b1;
    IF_ID_write_mark[0] = 1'b1;
    IF_ID_flush_mark[0] = 1'b1;
    ID_EX_flush_mark[0] = 1'b1;
end
```

jump 类冒险

这类冒险在 ID 段被发现，进入流水线后一条指令是没有用的，必须 flush 掉，伪码如下：

```
if ( ID.PCSrc != 0 )
    PC.write = 1
    IF/ID.write = 1
    IF/ID.flush = 1
    ID/EX.flush = 0
```

verilog 代码如下。

```
// Jump
if(ID_PCSrc[2:1]!=2'b00) begin
    PCWrite_mark[1] = 1'b1;
    IF_ID_write_mark[1] = 1'b1;
    IF_ID_flush_mark[1] = 1'b1;
    ID_EX_flush_mark[1] = 1'b0;
end
```

Hazard Unit 总体设计代码附于文末。

8. 顶端文件设计

将各个部分实例化，并实现各复用器的信号选择。
各复用器的实现如下:

```
//IF 部分
assign IF_PCSrc = (EX_PCSrc == 3'b001 && EX_ALUOut[0]) ? 3'b001 : (ID_PCSrc == 3'b001
? 3'b000 : ID_PCSrc);

assign IF_PC_add = IF_PC + 32'h4;
assign IF_PC_add_4 = {IF_PC[31], IF_PC_add[30:0]};

//WB 部分
assign WB_DataBusC = (WB_MemToReg == 2'b00)? WB_ALUOut: (WB_MemToReg == 2'b01)? WB_Mem
ReadData: WB_PC_add_4;

//ID 部分
//ID_EXTOp = 1 : sign extend; 0 : zero extend
assign ID_EXTOut = {ID_EXTOp? {16{ID_Immediate[15]}}: 16'h0000, ID_Immediate[15:0]};
//ID_LUOp = 1 : lui; 0 : else
//ID_LUOut = lui data
assign ID_LUOut = ID_LUOp? {ID_Immediate[15:0], 16'h0000}: ID_EXTOut;
//jump target
assign ID_JT = {IF_PC_add_4[31:28], ID_JumpAddr, 2'b00};

//EX 部分
// ALU Input
assign EX_ALU_A = EX_ALUSrc1? {17'h00000, EX_Shamt}: ForwardAData;
assign EX_ALU_B = EX_ALUSrc2? EX_LUOut: ForwardBData;
// Jump dst for branch instruction
assign EX_BT = (EX_ALUOut[0])? EX_PC_add_4 + {EX_LUOut[29:0], 2'b00}: EX_PC_add_4;
// AddrC will be used in WB, so it will flow to the next state
assign EX_AddrC = (EX_RegDst == 2'b00)? EX_Rd: (EX_RegDst == 2'b01)? EX_Rt: 5'd31;

//MEM 部分
assign DataMem_MemAddr = MEM_ALUOut;
assign DataMem_MemWrite = MEM_MemWrite;
assign DataMem_MemRead = MEM_MemRead;
assign MEM_MemReadData = DataMem_ReadData;
assign DataMem_WriteData = MEM_DataBusB;

//Forward Unit 部分
assign ForwardAData = (ForwardA==2'b00) ? EX_DataBusA : (ForwardA==2'b01) ? WB_DataBus
C : MEM_ALUOut;
assign ForwardBData = (ForwardB==2'b00) ? EX_DataBusB : (ForwardB==2'b01) ? WB_DataBus
C : MEM_ALUOut;
```

顶端文件总设计代码较长，附于文末。

二、仿真

测试程序：

测试程序段（汇编代码）

address	instruction	op	rs	rt	immediate	code	result
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	24010008	$1 = 8
0x00000004	ori $2,$0,2	001101	00000	00010	0000 0000 0000 0010	34020002	$2 = 2

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000008	add $3,$2,$1	000000	00010	00001	00011	00000	100000	00411820	$3 = 10
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101	00000	100010	00622822	$5 = 8
0x00000010	and $4,$5,$2	000000	00101	00010	00100	00000	100100	00a22024	$4 = 0
0x00000014	or $8,$4,$2	000000	00100	00010	01000	00000	100101	00824025	$8 = 2
0x00000018	sll $8,$8,1	000000	00000	01000	01000	00001	000000	00084040	$8 = 4 $8 = 8

address	instruction	op	rs	rt	immediate	code	result
0x0000001C	bne $8,$1,-2 (≠,转 18)	000101	00001	01000	1111 1111 1111 1110	1501fffe	
0x00000020	slti $6,$2,4	001010	00010	00110	0000 0000 0000 0100	28460004	$6 = 1
0x00000024	sltiu $7,$6,0	001011	00110	00111	0000 0000 0000 0000	2cc70000	$7 = 0
0x00000028	addiu $7,$7,8	001000	00111	00111	0000 0000 0000 1000	24e70008	$7 = 8 $7 = 16
0x0000002C	beq $7,$1,-2 (=,转 28)	000100	00111	00001	1111 1111 1111 1110	10e1fffe	
0x00000030	sw $2,4($1)	101011	00001	00010	0000 0000 0000 0100	ac220004	mem[12:15]<=2
0x00000034	lw $9,4($1)	100011	00001	01001	0000 0000 0000 0100	8c290004	$9 = 2
0x00000038	addiu $9,$9,0	001001	01001	01001	0000 0000 0000 0000	25290000	$9 = 2
0x0000003C	addiu $10,$0,-2	001001	00000	01010	1111 1111 1111 1110	240afffe	$10 = -2
0x00000040	addiu $10,$10,1	001001	01010	01010	0000 0000 0000 0001	254a0001	$10 = -1
0x00000044	andi $11,$2,2	001100	00010	01011	0000 0000 0000 0010	304b0002	$11 = 2

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000048	addu $10,$0,$2	000000	00000	00010	01010	00000	100001	00025021	$10 = 2
0x0000004C	subu $10,$10,$2	000000	01010	00010	01010	00000	100011	01425023	$10 = 0
0x00000050	xor $10,$8,$2	000000	01000	00010	01010	00000	100110	01025026	$10 =10
0x00000054	nor $10,$8,$2	000000	01000	00010	01010	00000	100111	01025027	$10= fffffff5
0x00000058	sltu $10,$8,$2	000000	01000	00010	01010	00000	101011	0102502b	$10 = 0
0x0000005C	srl $10,$8,1	000000	00000	01000	01010	00001	000010	00085042	$10 = 4

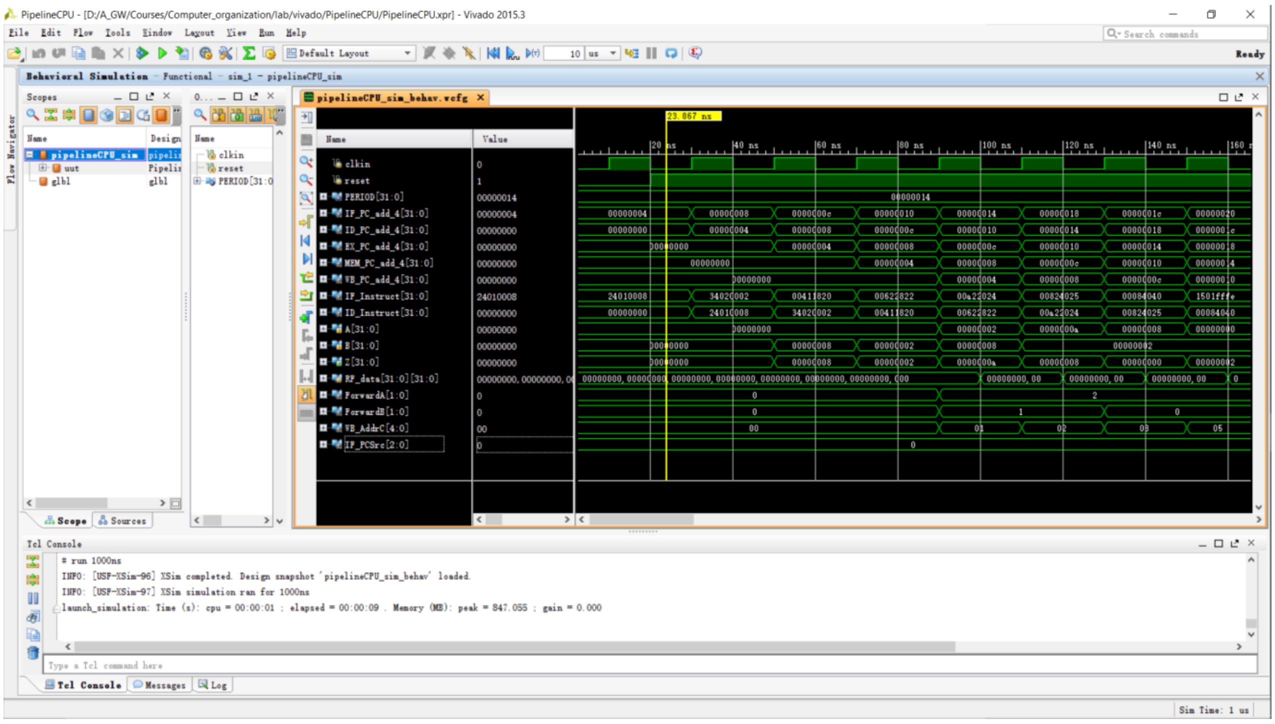
address	instruction	op	rs	rt	immediate	code	result
0x00000060	addi $10,$0,-1	001000	00000	01010	1111 1111 1111 1111	200affff	$10 = -1

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000064	sra $11,$10,1	000000	00000	01010	01011	00001	000011	000a5843	$11 = -1

address	instruction	op	rs	rt	immediate	code	result
0x00000068	haul	111111	00000	00000	0000000000000000	fc000000	haul

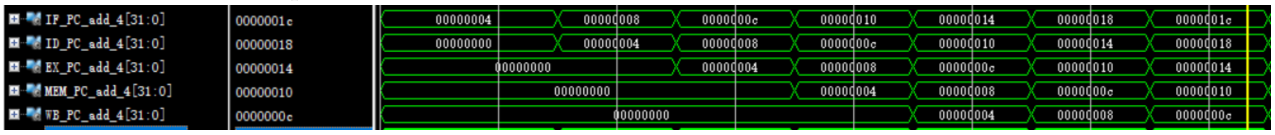
仿真模块： 文件附于文末

仿真结果：



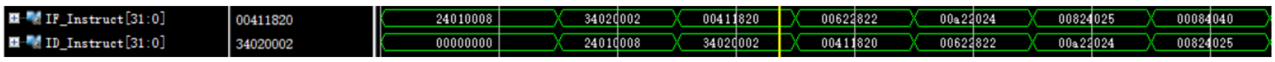
```
# run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'pipelineCPU_sim_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:09 . Memory (MB): peak = 847.055 ; gain = 0.000
```

首先可观察各个阶段的 pc_add_4:



可发现指令随流水线逐级往下执行。

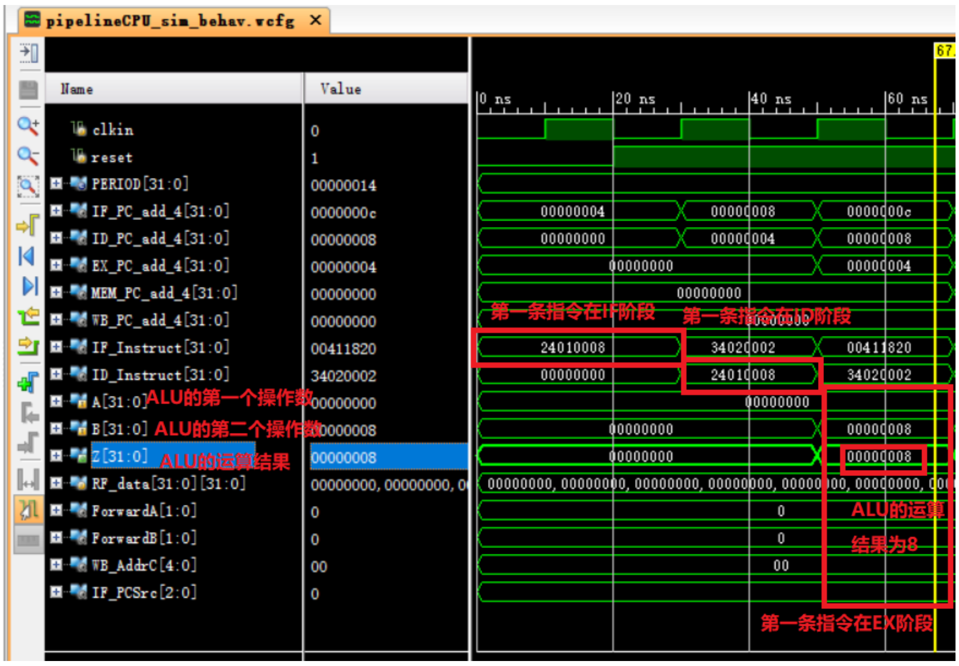
同理，可观察 IF_Instruction 和 ID_Instruction：

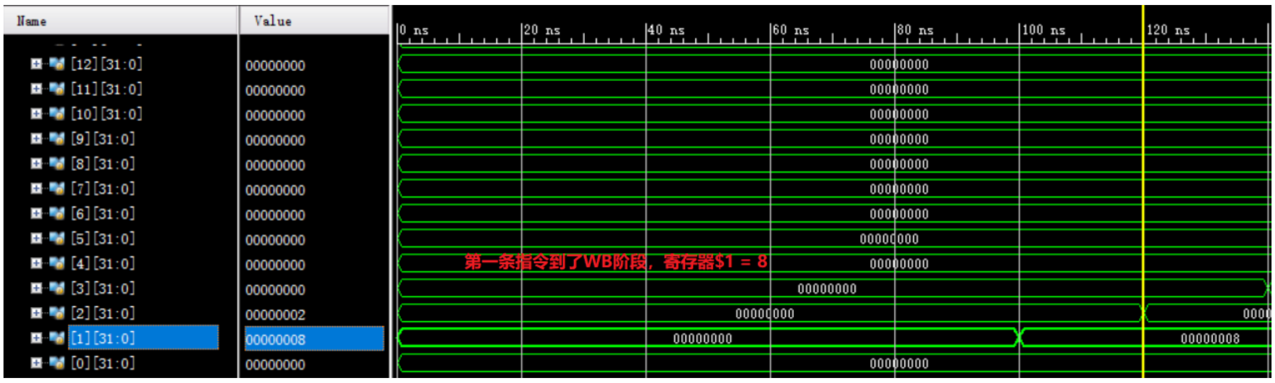


也可以观察到指令随流水线从 IF 到 ID 阶段的流动。

观察第一条指令的执行过程：

address	instruction	op	rs	rt	immediate	code	result
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	24010008	$1 = 8

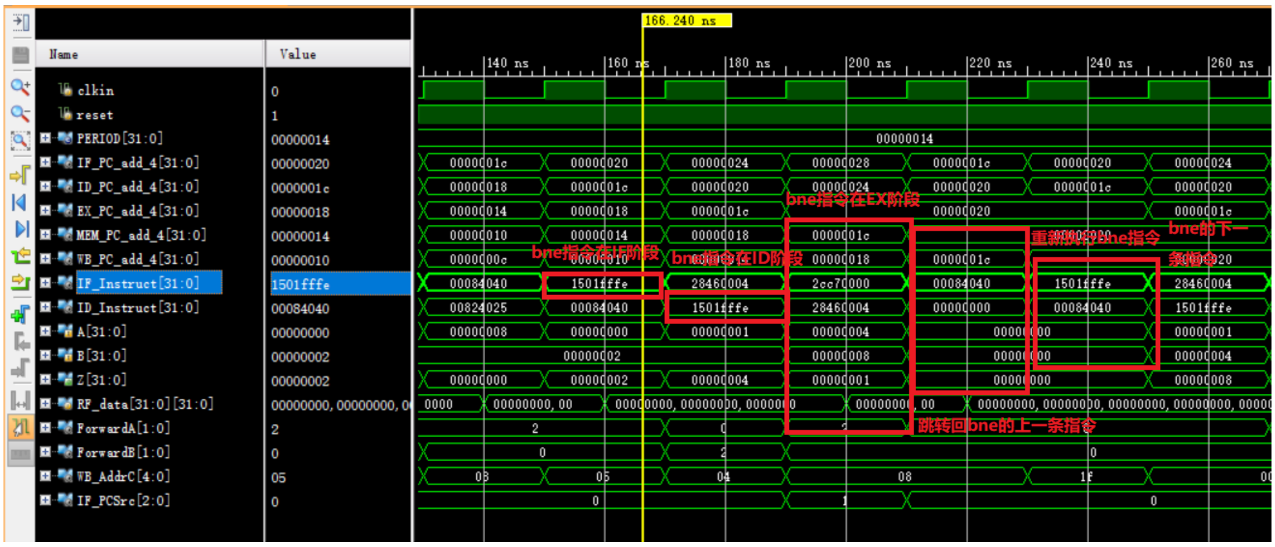




由于指令数较多，不一一细看，看几个比较有代表性的：

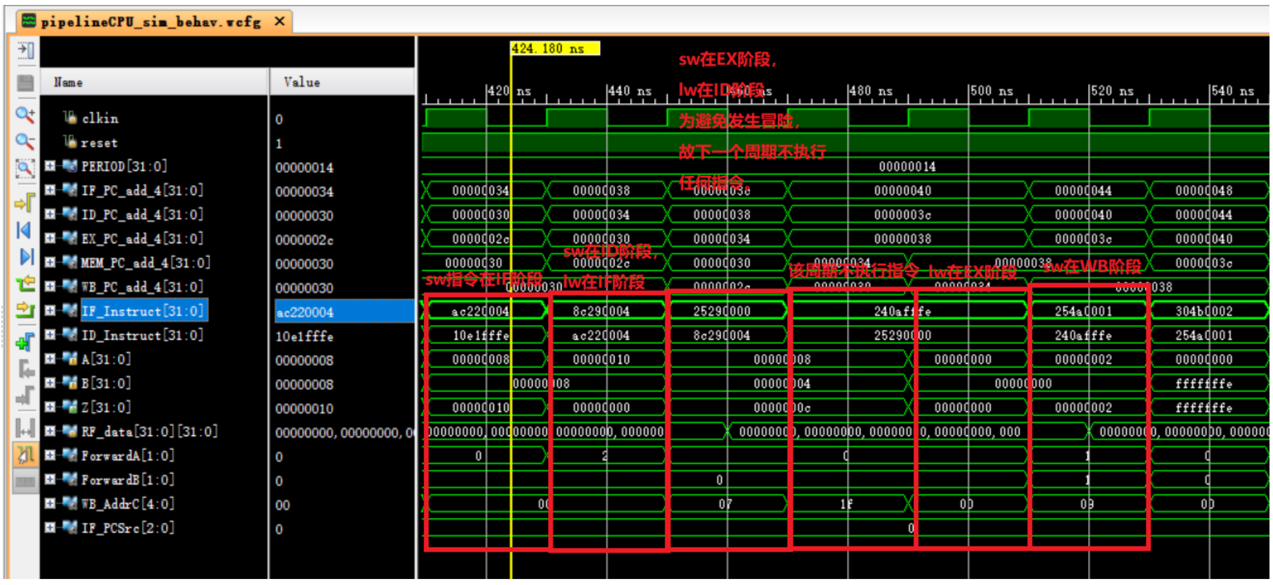
条件跳转指令：

address	instruction	op	rs	rt	immediate	code	result
0x0000001C	bne $8,$1,-2 (≠,转 18)	000101	00001	01000	1111 1111 1111 1110	1501fffe	

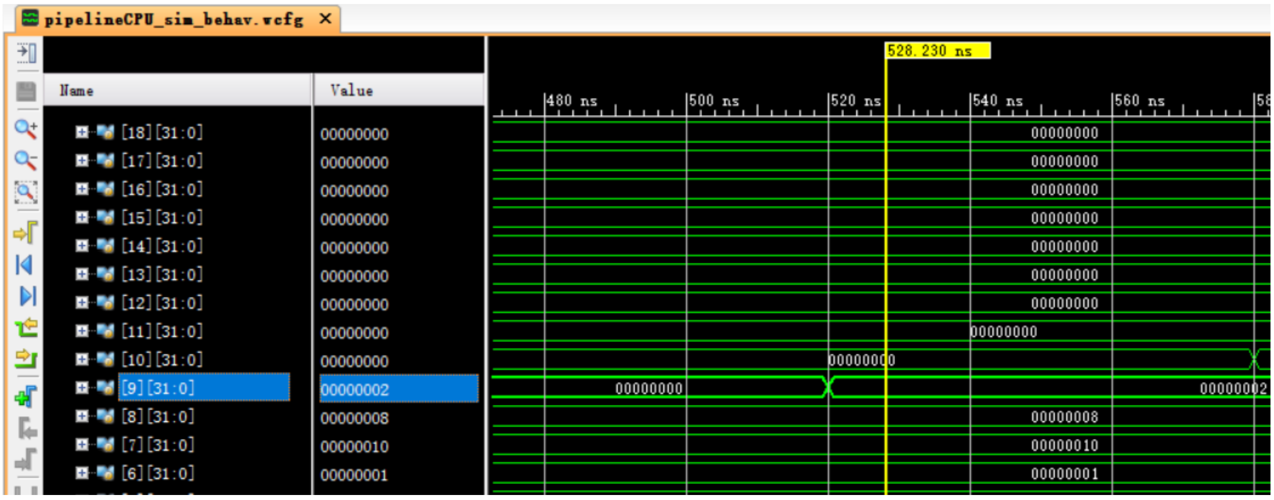


sw、lw 指令：

address	instruction	op	rs	rt	immediate	code	result
0x00000030	sw $2,4($1)	101011	00001	00010	0000 0000 0000 0100	ac220004	mem[12:15]<=2
0x00000034	lw $9,4($1)	100011	00001	01001	0000 0000 0000 0100	8c290004	$9 = 2

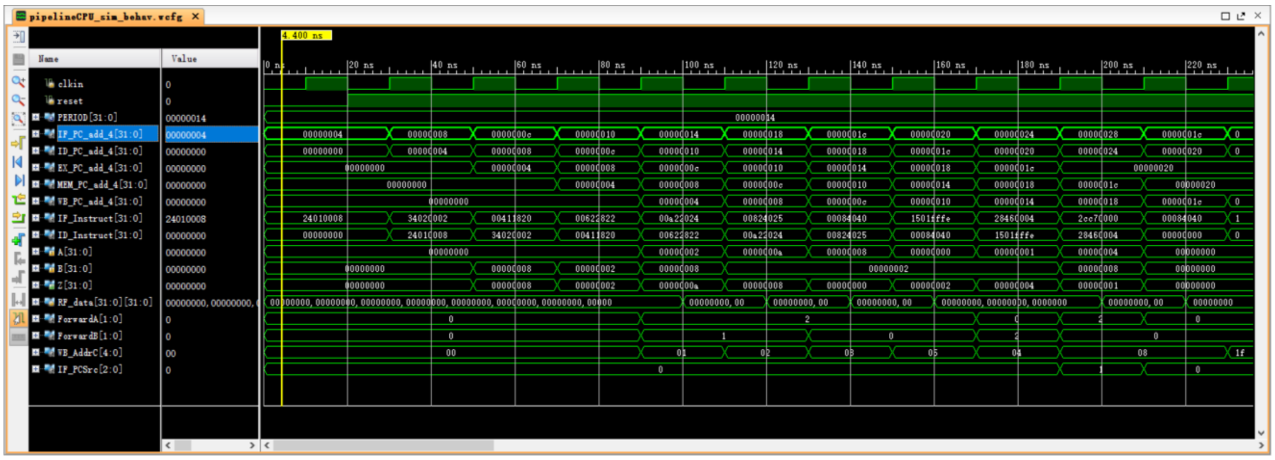


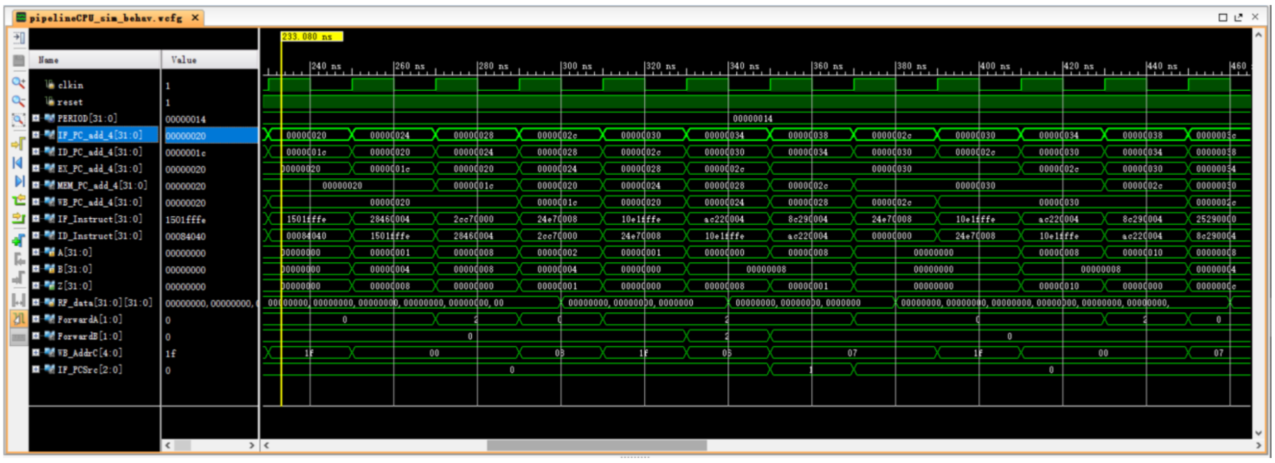
可以看到，最终$9 = 2，说明 sw、lw 两条指令都成功执行。

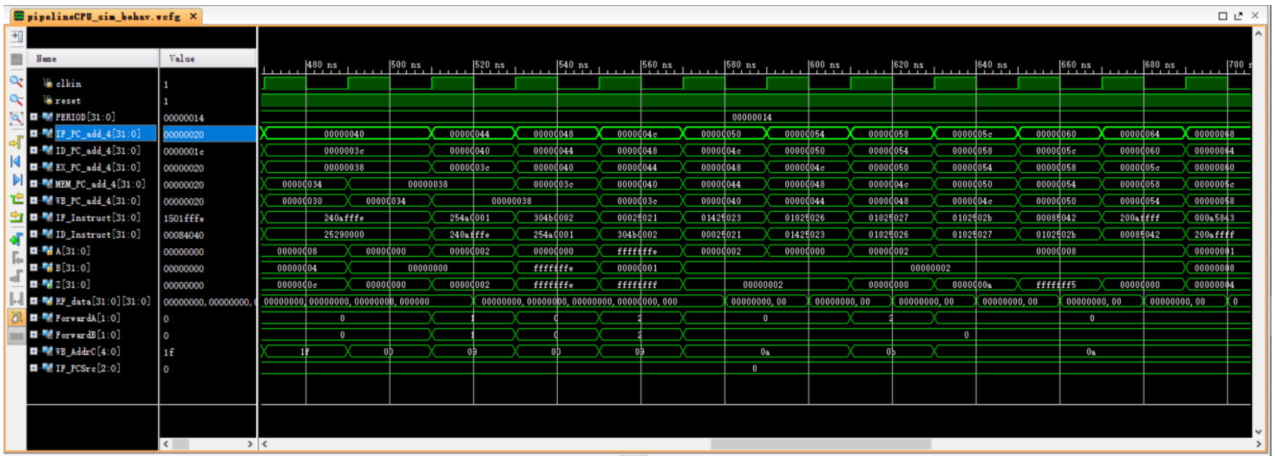


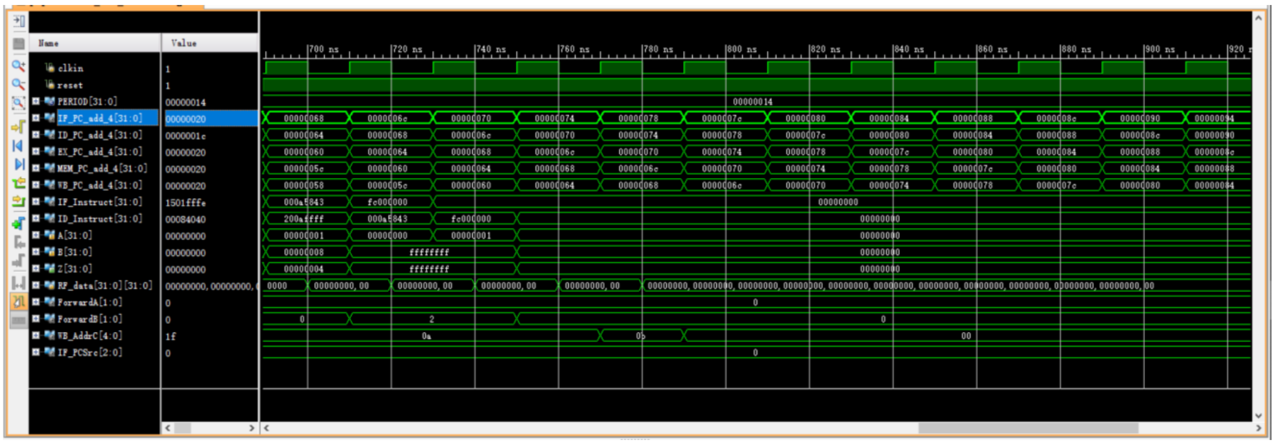
其他语句不一一赘述。

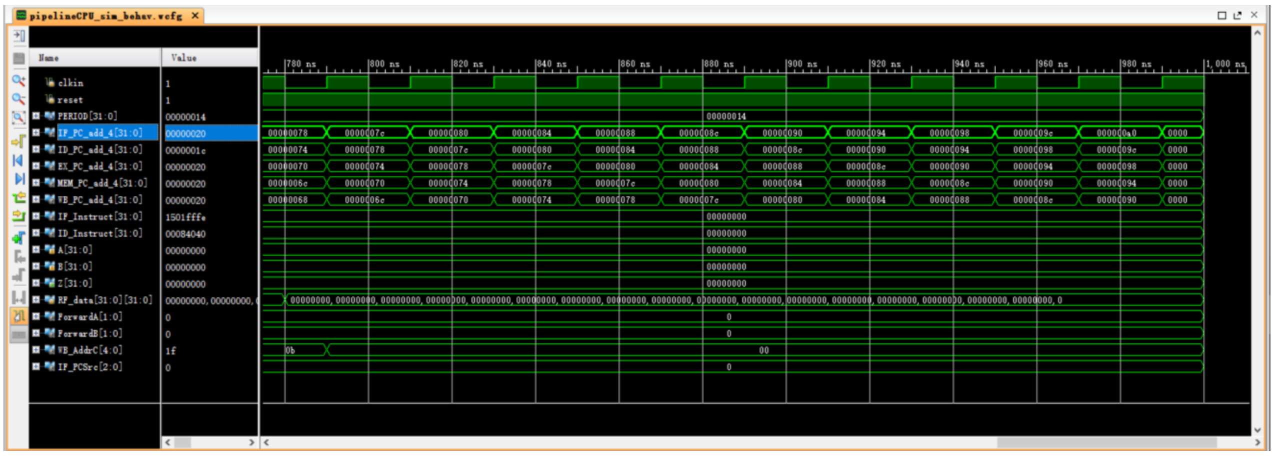
总结果：




pipelineCPU_sim_behav.wcfg
Name
Value
clkin 1
reset 1
PERIOD[31:0] 00000014
IF_PC_add_4[31:0] 00000020
ID_PC_add_4[31:0] 0000001c
EX_PC_add_4[31:0] 00000020
MEM_PC_add_4[31:0] 00000020
WB_PC_add_4[31:0] 00000020
IF_Instruct[31:0] 1501fffe
ID_Instruct[31:0] 00084040
A[31:0] 00000000
B[31:0] 00000000
Z[31:0] 00000000
RF_data[31:0][31:0] 00000000, 00000000,
ForwardA[1:0] 0
ForwardB[1:0] 0
WB_AddrC[4:0] 1f
IF_PCSrc[2:0] 0
233.080 ns


pipelineCPU_sim_behav.wcfg
Name
Value


Name
Value


pipelineCPU_sim_behav.wcfg
Name
Value

五、 实验感想

本次实验设计了一个流水线 CPU，在题目要求的基础上加入了转发和冒险控制功能，该 Forward Unit 与 Hazard Unit 实现起来还是比较复杂的，运用了很多上课讲到的知识，我在网上看了不少的资料以及别人的代码，最终选择了一个比较好的参照模板，按照它的标记方式来实现。

流水线 CPU 比单周期 CPU 复杂很多，为了效率的提升，牺牲的是实现的简易性。理想的五级流水线 CPU 的 CPI 是单周期 CPU 的五倍，但是把指令分成五级带来了许多问题。首先，由于每一个阶段所执行的指令并不一样，不能够共享数据，要增加四个寄存器来存储每一个阶段的数据；其次，流水线 CPU 存在着冒险的问题，为了解决冒险，一个朴素的想法是加入空指令或者流水线阻塞，但是这样又会大幅度降低流水线 CPU 的 CPI，于是人们就想到了转发和冒险控制。

在本次实验的过程中，我体会到了抽象的重要性，先把模块抽象化是非常重要的一步，能让我们不拘泥于细节，想清楚怎么安排功能，才能让代码组织起来更有条理。在单周期 CPU 的实验中我把移位功能完全内置到 ALU 内部，没有把它抽象成一个独立的模块，这次在流水线 CPU 中，在 ALU 内部定义了独立的 shift 模块，让代码组织更有条理也更容易拓展，而不是像是为了某些特殊指令而打补丁般的难拓展代码。

附录（流程图，注释过的代码）：

Forward Unit

```
module Forward_Unit(IF_ID_Rs, ID_EX_Rs, ID_EX_Rt, ID_PCSrc, ID_EX_RegWrite, ID_EX_Addr
C, EX_MEM_RegWrite, EX_MEM_AddrC,
    MEM_WB_RegWrite, MEM_WB_AddrC, ForwardA, ForwardB);

input [4:0] IF_ID_Rs;
input [4:0] ID_EX_Rs;
input [4:0] ID_EX_Rt;
input [2:0] ID_PCSrc;
input ID_EX_RegWrite;
input [4:0] ID_EX_AddrC;
input EX_MEM_RegWrite;
input [4:0] EX_MEM_AddrC;
input MEM_WB_RegWrite;
input [4:0] MEM_WB_AddrC;

output reg [1:0] ForwardA;
output reg [1:0] ForwardB;

always @(*) begin
    if(EX_MEM_RegWrite && EX_MEM_AddrC != 5'h00
        && EX_MEM_AddrC == ID_EX_Rs) begin
        ForwardA = 2'b10;
    end
    else if(MEM_WB_RegWrite && MEM_WB_AddrC != 5'h00
        && MEM_WB_AddrC == ID_EX_Rs) begin
        ForwardA = 2'b01;
    end
    else
        ForwardA = 2'b00;

    // Only replace Rt with Rs for ForwardA
    if(EX_MEM_RegWrite && EX_MEM_AddrC != 5'h00
        && EX_MEM_AddrC == ID_EX_Rt) begin
        ForwardB = 2'b10;
    end
    else if(MEM_WB_RegWrite && MEM_WB_AddrC != 5'h00
        && MEM_WB_AddrC == ID_EX_Rt) begin
        ForwardB = 2'b01;
    end
    else
        ForwardB = 2'b00;

end

endmodule
```

Hazard Unit

```
module Hazard_Unit(IF_ID_Rs, IF_ID_Rt, ID_PCSrc, EX_PCSrc, ID_EX_MemRead, ID_EX_Rt, EX
_ALUOut_0, PCWrite, IF_ID_write, IF_ID_flush, ID_EX_flush);
```

```verilog
input [4:0] IF_ID_Rs;
input [4:0] IF_ID_Rt;
input [2:0] ID_PCSrc;
input [2:0] EX_PCSrc;
input ID_EX_MemRead;
input [4:0] ID_EX_Rt;
input EX_ALUOut_0;

output PCWrite;
output IF_ID_write;
output IF_ID_flush;
output ID_EX_flush;

reg [2:0] PCWrite_mark;
reg [2:0] IF_ID_write_mark;
reg [2:0] IF_ID_flush_mark;
reg [2:0] ID_EX_flush_mark;

always @(*) begin
    // Branch
    if (EX_PCSrc == 3'd1 && EX_ALUOut_0) begin
        PCWrite_mark[0] = 1'b1;
        IF_ID_write_mark[0] = 1'b1;
        IF_ID_flush_mark[0] = 1'b1;
        ID_EX_flush_mark[0] = 1'b1;
    end
    else begin
        PCWrite_mark[0] = 1'b1;
        IF_ID_write_mark[0] = 1'b1;
        IF_ID_flush_mark[0] = 1'b0;
        ID_EX_flush_mark[0] = 1'b0;
    end
    // Jump
    if(ID_PCSrc[2:1]==2'b00) begin
        PCWrite_mark[1] = 1'b1;
        IF_ID_write_mark[1] = 1'b1;
        IF_ID_flush_mark[1] = 1'b0;
        ID_EX_flush_mark[1] = 1'b0;
    end
    else begin
        PCWrite_mark[1] = 1'b1;
        IF_ID_write_mark[1] = 1'b1;
        IF_ID_flush_mark[1] = 1'b1;
        ID_EX_flush_mark[1] = 1'b0;
    end
    // Load use
    if(ID_EX_MemRead && (ID_EX_Rt == IF_ID_Rs || ID_EX_Rt == IF_ID_Rt)) begin
        PCWrite_mark[2] = 1'b0;
        IF_ID_write_mark[2] = 1'b0;
        IF_ID_flush_mark[2] = 1'b0;
        ID_EX_flush_mark[2] = 1'b1;
    end
    else begin
        PCWrite_mark[2] = 1'b1;
        IF_ID_write_mark[2] = 1'b1;
        IF_ID_flush_mark[2] = 1'b0;
        ID_EX_flush_mark[2] = 1'b0;
    end
end

assign PCWrite = PCWrite_mark[0] & PCWrite_mark[1] & PCWrite_mark[2];
assign IF_ID_write = IF_ID_write_mark[0] & IF_ID_write_mark[1] & IF_ID_write_mark[2];
assign IF_ID_flush = IF_ID_flush_mark[0] | IF_ID_flush_mark[1] | IF_ID_flush_mark[2];
assign ID_EX_flush = ID_EX_flush_mark[0] | ID_EX_flush_mark[1] | ID_EX_flush_mark[2];

endmodule
```

顶端文件

```
module PipelineCPU(clk, reset);

input clk;
input reset;

//IF
wire [2:0] IF_PCSrc;
reg [31:0] IF_PC;
wire [31:0] IF_PC_add;
wire [31:0] IF_PC_add_4;
wire [31:0] IF_Instruct;

//ID
wire [31:0] ID_PC_add_4;
wire [31:0] ID_Instruct;
wire [4:0] ID_Rs;
wire [4:0] ID_Rt;
wire [4:0] ID_Rd;
wire [4:0] ID_Shamt;
wire [15:0] ID_Immediate;
wire [25:0] ID_JumpAddr;
wire [31:0] ID_JT;//jump target
wire [31:0] ID_DataBusA;
wire [31:0] ID_DataBusB;
wire [31:0] ID_EXTOut;
wire [31:0] ID_LUOut;

wire [1:0] ID_RegDst;
wire [2:0] ID_PCSrc;
wire ID_MemRead;
wire ID_MemWrite;
wire [1:0] ID_MemToReg;
wire [5:0] ID_ALUFun;
wire ID_EXTOp;
wire ID_LUOp;
wire ID_ALUSrc1;
wire ID_ALUSrc2;
wire ID_RegWrite;
wire ID_Sign;

//EX
wire [31:0] EX_PC_add_4;
wire [4:0] EX_Rs;
wire [4:0] EX_Rt;
wire [4:0] EX_Rd;
wire [4:0] EX_Shamt;
wire [4:0] EX_AddrC;
wire [31:0] EX_ALU_A;
wire [31:0] EX_ALU_B;
wire [31:0] EX_ALUOut;
wire [31:0] EX_DataBusA;
wire [31:0] EX_DataBusB;
wire [31:0] EX_LUOut;
wire [31:0] EX_BT;

wire EX_Sign;
wire [1:0] EX_RegDst;
wire [2:0] EX_PCSrc;
wire EX_MemRead;
wire EX_MemWrite;
wire [1:0] EX_MemToReg;
wire [5:0] EX_ALUFun;
wire EX_ALUSrc1;
wire EX_ALUSrc2;
wire EX_RegWrite;

//MEM
wire [31:0] MEM_PC_add_4;
wire [4:0] MEM_Rt;
wire [4:0] MEM_Rd;
wire [31:0] MEM_DataBusB;
wire [4:0] MEM_AddrC;
wire [31:0] MEM_MemReadData;
wire [31:0] MEM_ALUOut;

wire [1:0] MEM_RegDst;
wire MEM_MemRead;
wire MEM_MemWrite;
wire [1:0] MEM_MemToReg;
wire MEM_RegWrite;

//WB
wire [31:0] WB_PC_add_4;
wire [4:0] WB_Rt;
wire [4:0] WB_Rd;
wire [31:0] WB_DataBusC;
wire [31:0] WB_MemReadData;
wire [4:0] WB_AddrC;
wire [31:0] WB_ALUOut;

wire [1:0] WB_RegDst;
wire [1:0] WB_MemToReg;
wire WB_RegWrite;

//Data Memory
wire [31:0] DataMem_ReadData;
wire [31:0] DataMem_MemAddr;
wire [31:0] DataMem_WriteData;
wire DataMem_MemWrite;
wire DataMem_MemRead;

//Forward
wire [1:0] ForwardA;
wire [1:0] ForwardB;
wire [31:0] ForwardAData;
wire [31:0] ForwardBData;
wire [31:0] ForwardJData;

//Hazard
wire PCWrite;
wire IF_ID_write;
wire IF_ID_flush;
wire ID_EX_flush;

assign IF_PCSrc = (EX_PCSrc == 3'b001 && EX_ALUOut[0]) ? 3'b001 : (ID_PCSrc == 3'b001
? 3'b000 : ID_PCSrc);
```

```verilog
assign IF_PC_add = IF_PC + 32'h4;
assign IF_PC_add_4 = {IF_PC[31], IF_PC_add[30:0]};

always@(posedge clk or negedge reset)
    if(~reset)
        IF_PC <= 32'h0000_0000;         //PC begins at 0x0000_0000
    else if(PCWrite)
        case(IF_PCSrc)
            3'h0: IF_PC <= IF_PC_add_4; //pc+4
            3'h1: IF_PC <= EX_BT;       //branch target
            3'h2: IF_PC <= ID_JT;       //jump target
            3'h3: IF_PC <= ForwardJData;//jump register
            default: IF_PC <= 32'h0000_0000;
        endcase

ROM ROM(.addr(IF_PC), .data(IF_Instruct));

IF_ID_Reg IF_ID_Reg(
    .clk(clk), .reset(reset),   .IF_ID_flush(IF_ID_flush),
    .ID_EX_flush(ID_EX_flush),  .IF_ID_write(IF_ID_write),
    .PC_add_4_in(IF_PC_add_4),  .Instruct_in(IF_Instruct),
    .PC_add_4_out(ID_PC_add_4), .Instruct_out(ID_Instruct));

assign ID_Rs = ID_Instruct[25:21];
assign ID_Rt = ID_Instruct[20:16];
assign ID_Rd = ID_Instruct[15:11];
assign ID_Shamt = ID_Instruct[10:6];
assign ID_Immediate = ID_Instruct[15:0];
assign ID_JumpAddr = ID_Instruct[25:0];

assign WB_DataBusC = (WB_MemToReg == 2'b00)? WB_ALUOut: (WB_MemToReg == 2'b01)? WB_Mem
ReadData: WB_PC_add_4;

RegFile RegFile(
    .reset(reset), .clk(clk),
    .addr1(ID_Rs), .data1(ID_DataBusA),
    .addr2(ID_Rt), .data2(ID_DataBusB),
    .wr(WB_RegWrite),
    .addr3(WB_AddrC), .data3(WB_DataBusC));

//ID_EXTOp = 1 : sign extend; 0 : zero extend
assign ID_EXTOut = {ID_EXTOp? {16{ID_Immediate[15]}}: 16'h0000, ID_Immediate[15:0]};
//ID_LUOp = 1 : lui; 0 : else
//ID_LUOut = lui data
assign ID_LUOut = ID_LUOp? {ID_Immediate[15:0], 16'h0000}: ID_EXTOut;
```

```
//jump target
assign ID_JT = {IF_PC_add_4[31:28], ID_JumpAddr, 2'b00};

ID_EX_Reg ID_EX_Reg(
    .clk(clk), .reset(reset),   .ID_EX_flush(ID_EX_flush),
    .PC_add_4_in(ID_PC_add_4),  .DataBusA_in(ID_DataBusA),
    .DataBusB_in(ID_DataBusB),  .LUOut_in(ID_LUOut),
    .Rs_in(ID_Rs),              .Rt_in(ID_Rt),
    .Rd_in(ID_Rd),              .Shamt_in(ID_Shamt),
    .RegDst_in(ID_RegDst),      .PCSrc_in(ID_PCSrc),
    .MemRead_in(ID_MemRead),    .MemWrite_in(ID_MemWrite),
    .MemToReg_in(ID_MemToReg),  .ALUFun_in(ID_ALUFun),
    .ALUSrc1_in(ID_ALUSrc1),    .ALUSrc2_in(ID_ALUSrc2),
    .RegWrite_in(ID_RegWrite),  .Sign_in(ID_Sign),
    .PC_add_4_out(EX_PC_add_4), .DataBusA_out(EX_DataBusA),
    .DataBusB_out(EX_DataBusB), .LUOut_out(EX_LUOut),
    .Rs_out(EX_Rs),             .Rt_out(EX_Rt),
    .Rd_out(EX_Rd),             .Shamt_out(EX_Shamt),
    .RegDst_out(EX_RegDst),     .PCSrc_out(EX_PCSrc),
    .MemRead_out(EX_MemRead),   .MemWrite_out(EX_MemWrite),
    .MemToReg_out(EX_MemToReg), .ALUFun_out(EX_ALUFun),
    .ALUSrc1_out(EX_ALUSrc1),   .ALUSrc2_out(EX_ALUSrc2),
    .RegWrite_out(EX_RegWrite), .Sign_out(EX_Sign));

// ALU Input
assign EX_ALU_A = EX_ALUSrc1? {17'h00000, EX_Shamt}: ForwardAData;
assign EX_ALU_B = EX_ALUSrc2? EX_LUOut: ForwardBData;
// Jump dst for branch instruction
assign EX_BT = (EX_ALUOut[0])? EX_PC_add_4 + {EX_LUOut[29:0], 2'b00}: EX_PC_add_4;
// AddrC will be used in WB, so it will flow to the next state
assign EX_AddrC = (EX_RegDst == 2'b00)? EX_Rd: (EX_RegDst == 2'b01)? EX_Rt: 5'd31;

ALU ALU(
    .A(EX_ALU_A),       .B(EX_ALU_B),
    .ALUFun(EX_ALUFun), .Sign(EX_Sign),
    .Z(EX_ALUOut));

EX_MEM_Reg EX_MEM_Reg(
    .clk(clk),                  .reset(reset),
    .PC_add_4_in(EX_PC_add_4),  .ALUOut_in(EX_ALUOut),
    .DataBusB_in(ForwardBData), .Rt_in(EX_Rt),
    .Rd_in(EX_Rd),              .RegDst_in(EX_RegDst),
    .MemRead_in(EX_MemRead),    .MemWrite_in(EX_MemWrite),
    .MemToReg_in(EX_MemToReg),  .RegWrite_in(EX_RegWrite),
    .AddrC_in(EX_AddrC),        .PC_add_4_out(MEM_PC_add_4),
    .ALUOut_out(MEM_ALUOut),    .DataBusB_out(MEM_DataBusB),
```

```verilog
    .Rt_out(MEM_Rt),            .Rd_out(MEM_Rd),
    .RegDst_out(MEM_RegDst),    .MemRead_out(MEM_MemRead),
    .MemWrite_out(MEM_MemWrite),.MemToReg_out(MEM_MemToReg),
    .RegWrite_out(MEM_RegWrite),.AddrC_out(MEM_AddrC));

assign DataMem_MemAddr = MEM_ALUOut;
assign DataMem_MemWrite = MEM_MemWrite;
assign DataMem_MemRead = MEM_MemRead;
assign MEM_MemReadData = DataMem_ReadData;
assign DataMem_WriteData = MEM_DataBusB;

MEM_WB_Reg MEM_WB_Reg(
    .clk(clk), .reset(reset),       .PC_add_4_in(MEM_PC_add_4),
    .ALUOut_in(MEM_ALUOut),         .MemReadData_in(MEM_MemReadData),
    .Rt_in(MEM_Rt),                 .Rd_in(MEM_Rd),
    .RegDst_in(MEM_RegDst),         .MemToReg_in(MEM_MemToReg),
    .RegWrite_in(MEM_RegWrite),     .AddrC_in(MEM_AddrC),
    .PC_add_4_out(WB_PC_add_4),     .ALUOut_out(WB_ALUOut),
    .MemReadData_out(WB_MemReadData), .Rt_out(WB_Rt),
    .Rd_out(WB_Rd),                 .RegDst_out(WB_RegDst),
    .MemToReg_out(WB_MemToReg),     .RegWrite_out(WB_RegWrite),
    .AddrC_out(WB_AddrC));

Control Control(
    .Instruct(ID_Instruct),         .PCSrc(ID_PCSrc),
    .RegWr(ID_RegWrite),            .RegDst(ID_RegDst),
    .MemRd(ID_MemRead),             .MemWr(ID_MemWrite),
    .MemToReg(ID_MemToReg),         .ALUSrc1(ID_ALUSrc1),
    .ALUSrc2(ID_ALUSrc2),           .EXTOp(ID_EXTOp),
    .LUOp(ID_LUOp),                 .ALUFun(ID_ALUFun),
    .Sign(ID_Sign));

DataMem DataMem(
    .reset(reset),              .clk(clk),
    .rd(DataMem_MemRead),       .wr(DataMem_MemWrite),
    .addr(DataMem_MemAddr),     .wdata(DataMem_WriteData),
    .rdata(DataMem_ReadData));

Forward_Unit Forward_Unit(
    .IF_ID_Rs(ID_Rs),               .ID_EX_Rs(EX_Rs),
    .ID_EX_Rt(EX_Rt),               .ID_PCSrc(ID_PCSrc),
    .ID_EX_RegWrite(EX_RegWrite),   .ID_EX_AddrC(EX_AddrC),
    .EX_MEM_RegWrite(MEM_RegWrite), .EX_MEM_AddrC(MEM_AddrC),
    .MEM_WB_RegWrite(WB_RegWrite),  .MEM_WB_AddrC(WB_AddrC),
    .ForwardA(ForwardA),            .ForwardB(ForwardB));
```

```
assign ForwardAData = (ForwardA==2'b00) ? EX_DataBusA : (ForwardA==2'b01) ? WB_DataBus
C : MEM_ALUOut;
assign ForwardBData = (ForwardB==2'b00) ? EX_DataBusB : (ForwardB==2'b01) ? WB_DataBus
C : MEM_ALUOut;

Hazard_Unit Hazard_Unit(
    .IF_ID_Rs(ID_Rs),           .IF_ID_Rt(ID_Rt),
    .ID_PCSrc(ID_PCSrc),        .EX_PCSrc(EX_PCSrc),
    .ID_EX_MemRead(EX_MemRead), .ID_EX_Rt(EX_Rt),
    .EX_ALUOut_0(EX_ALUOut[0]), .PCWrite(PCWrite),
    .IF_ID_write(IF_ID_write),  .IF_ID_flush(IF_ID_flush),
    .ID_EX_flush(ID_EX_flush));

endmodule
```

仿真文件

```
`timescale 1ns / 1ps

module pipelineCPU_sim;

// Inputs
reg clkin;
reg reset;

// Instantiate the Unit Under Test (UUT)
PipelineCPU uut (
    .reset(reset),
    .clk(clkin)
);

initial
begin
    // Initialize Inputs
    clkin = 0;
    reset = 0;
    // Wait 20 ns for global reset to finish
    #20;
    reset = 1;
end

parameter PERIOD = 20;

always
begin
    clkin = 1'b0;
    #(PERIOD / 2);
    clkin = 1'b1;
    #(PERIOD / 2);
end
endmodule
```