哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

课程报告

开课学期：2023 夏季

课程名称：计算机设计与实践

项目名称：基于 miniRV/LA 的 SoC 设计

项目类型：综合设计型

课程学时：56　地点：T2506

学生班级：计算机 8 班

学生学号：210110831

学生姓名：王静茹

评阅教师：

报告成绩：

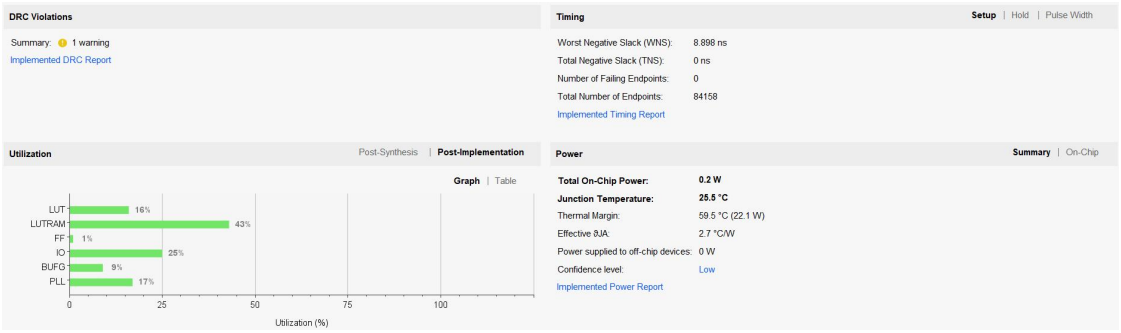
实验与创新实践教育中心制

2023 年 7 月

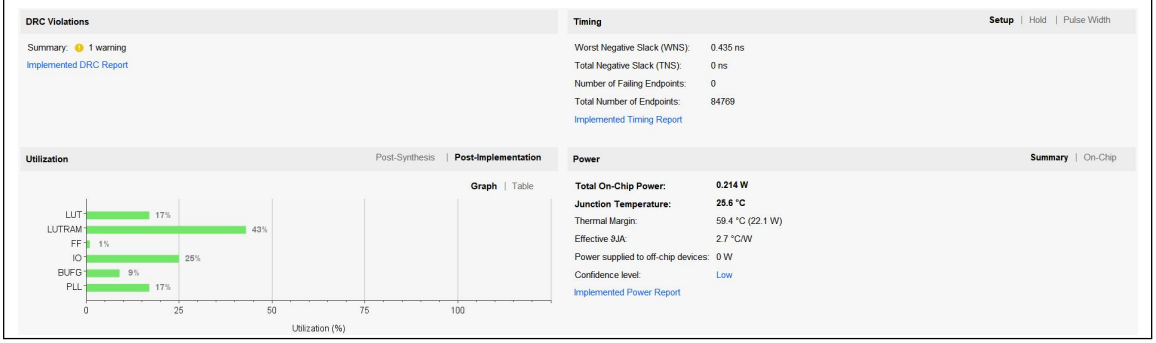
注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

设计概述（罗列出所有实现的指令，以及单周期/流水线 CPU 频率）
实现的指令：add、sub、and、or、xor、sll、srl、sra、slt、sltu、addi、andi、ori、xori、slli、srli、srai、slti、sltiu、lb、lbu、lh、lhu、lw、jalr、sb、sh、sw、beq、bne、blt、bltu、bge、bgeu、lui、auipc、jal 单周期 CPU 频率：25MHz 流水线 CPU 频率：75MHz
设计的主要特色（除基本要求以外的设计）
1. 实现了 37 条指令 2. 使用分支预测解决了流水线 CPU 的控制冒险
资源使用、功耗数据截图（Post Implementation；含单周期、流水线 2 个截图）

单周期：



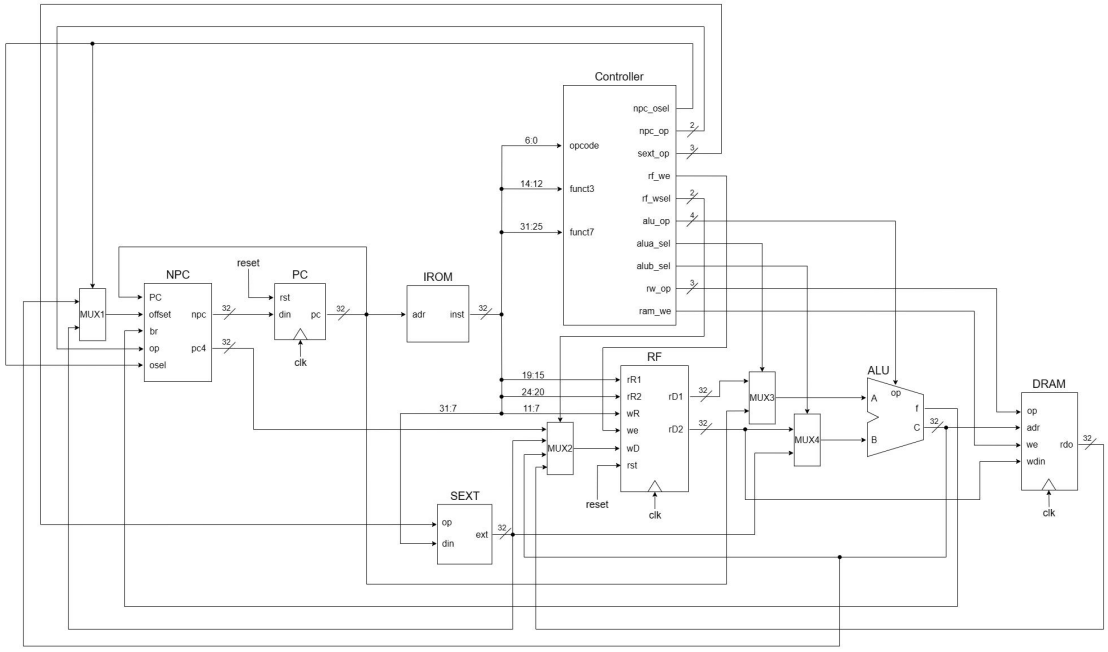
流水线：



1 单周期 CPU 设计与实现

1.1 单周期 CPU 数据通路设计

要求：贴出完整的单周期数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。



功能：

NPC：确定下一条指令的地址并输出 PC+4 的值；

PC：存储当前指令的地址；

IROM：指令存储器（只读）；

NPC、PC、IROM 这三个功能部件及必要的多路选择器共同组成了单周期 CPU 的取指单元。

SEXT：根据指令类型对指令中的立即数进行扩展；

RF：寄存器堆，用于完成读/写寄存器操作；

SEXT、RF 这两个功能部件及必要的多路选择器共同组成了单周期 CPU 的译码单元。

ALU：对源操作数 A 和 B 进行算术运算/逻辑运算/移位运算/大小判断等运算；

ALU 及两个针对源操作数的多路选择器共同组成了单周期 CPU 的执行单元。

DRAM：数据存储器，可读可写；

DRAM 及相关数据处理逻辑组成了单周期 CPU 的访存单元。

Controller：控制单元，根据指令类型产生相应的控制信号。

1.2 单周期 CPU 模块详细设计

要求：以表格的形式列出各个部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等形象化工具和手段，详细描述各个部件的关键实现。

NPC：

部件	接口信号	属性	位宽	功能/释义
NPC	PC	输入	32	当前指令地址
	offset	输入	32	指令跳转的偏移量
	br	输入	1	条件分支的跳转标志
	op	输入	2	控制 npc 的产生
	osel	输入	1	控制 npc 的产生（仅在面对无条件跳转指令时起作用）
	npc	输出	32	下一条指令地址
	pc4	输出	32	PC+4 的值

关键实现：

```
assign pc4 = PC + 4;

always @(*) begin
    case (op)
        `NPC_PC4 : npc = PC + 4;
        `NPC_BRN : npc = br ? (PC + offset) : (PC + 4);
        `NPC_JMP : npc = osel ? offset : (PC + offset);
        default: npc = 32'b0;
    endcase
end
```

综合考虑 op、br、osel 这三个控制信号，根据其值的不同，为 npc 选出不同的数据来源。

PC：

部件	接口信号	属性	位宽	功能/释义
PC	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	din	输入	32	下一条指令地址
	pc	输出	32	当前指令地址

关键实现：

```
always @(posedge clk or posedge rst) begin
    if (rst)
        pc <= 32'b0;
    else
        pc <= din;
end
```

正常执行时，pc 在时钟上升沿处更新；CPU 复位时，pc 清零。

SEXT：

部件	接口信号	属性	位宽	功能/释义
SEXT	op	输入	3	控制立即数的扩展方式
	din	输入	25	指令中的立即数
	ext	输出	32	扩展后的立即数

关键实现：

```
always @(*) begin
    case (op)
        `EXT_I : ext = {{20{din[24]}}, din[24:13]};
        `EXT_S : ext = {{20{din[24]}}, din[24:18], din[4:0]};
        `EXT_B : ext = {{20{din[24]}}, din[0], din[23:18], din[4:1], 1'b0};
        `EXT_U : ext = {din[24:5], 12'b0};
        `EXT_J : ext = {{12{din[24]}}, din[12:5], din[13], din[23:14], 1'b0};
        default: ext = 32'b0;
    endcase
end
```

根据 op 的不同，为立即数选择不同的扩展方式。

RF：

部件	接口信号	属性	位宽	功能/释义
RF	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	rR1	输入	5	源寄存器 1 的编号
	rR2	输入	5	源寄存器 2 的编号
	wR	输入	5	目的寄存器的编号
	we	输入	1	写使能信号
	wD	输入	32	要写回目的寄存器的数据
	rD1	输出	32	源寄存器 1 中存的数据
	rD2	输出	32	源寄存器 2 中存的数据

关键实现：

```
//读操作
assign rD1 = (rst || rR1 == 5'b0) ? 32'b0 : regs[rR1];
assign rD2 = (rst || rR2 == 5'b0) ? 32'b0 : regs[rR2];

//写操作
always @(posedge clk or posedge rst) begin
    if (rst) begin…
    end
    else if (we && wR != 5'b0)
        regs[wR] <= wD;
end
```

利用数组实现寄存器堆；利用 we 控制寄存器堆的写操作；异步读、同步写。

ALU：

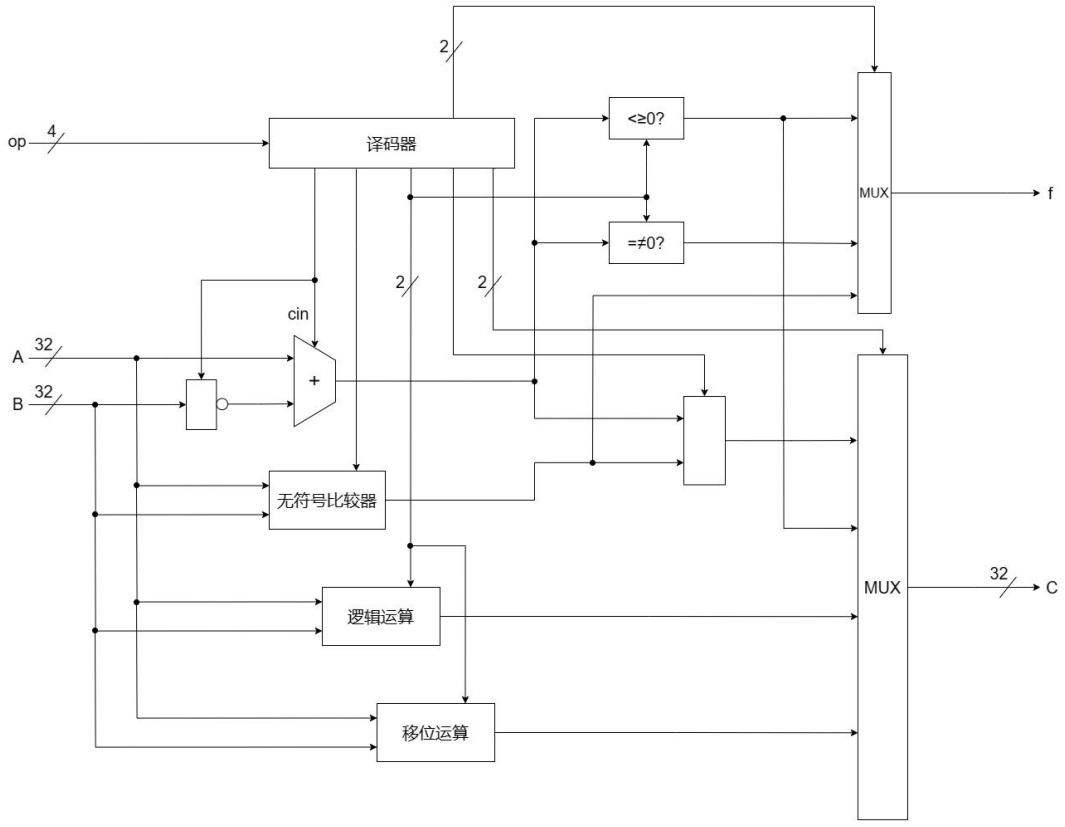
部件	接口信号	属性	位宽	功能/释义

ALU	op	输入	4	运算控制信号
	A	输入	32	源操作数 A
	B	输入	32	源操作数 B
	f	输出	1	条件分支跳转标志
	C	输出	32	运算结果

关键实现：

```
assign op_A = alua_sel ? pc : rD1;
assign op_B = alub_sel ? imm_ext : rD2;

ALU u_ALU (
    .A (op_A),
    .B (op_B),
    .op (alu_op),
    .C (ALU_C),
    .f (ALU_f)
);
```



首先，根据控制信号确定源操作数 A、B 的数据来源；然后，分别对源操作数 A、B 进行算术运算、比较运算、逻辑运算和移位运算（其中，算术运算通过加法器和补码逻辑实现，有符号比较通过减法运算实现，无符号比较、逻辑运算和移位运算都直接通过 Verilog 运算符实现）；最后，根据从译码器得到的选择信号，分别确定 f 和 C 的数据来源，得到最终结果。

memory（访存模块）：

部件	接口信号	属性	位宽	功能/释义

memory	op	输入	3	控制读/写存储器的位宽
	we	输入	1	写使能信号
	adr	输入	32	写地址
	wdin	输入	32	写数据（原始）
	rdata	输入	32	读数据（原始）
	wdo	输出	32	最终要写入 DRAM 的数据
	rdo	输出	32	最终读出的数据

关键实现：

写：

```
always @(*) begin
    if (we && op[1:0] == 2'b00) begin
        case (adr[1:0])
            2'b00 : wdo = {rdata[31:8], wdin[7:0]};
            2'b01 : wdo = {rdata[31:16], wdin[7:0], rdata[7:0]};
            2'b10 : wdo = {rdata[31:24], wdin[7:0], rdata[15:0]};
            2'b11 : wdo = {wdin[7:0], rdata[23:0]};
            default: wdo = 32'b0;
        endcase
    end
    else if (we && op[1:0] == 2'b01)
        wdo = adr[1] ? {wdin[15:0], rdata[15:0]} : {rdata[31:16], wdin[15:0]};
    else
        wdo = wdin;
end
```

op[1:0]代表着要写入 DRAM 的数据的位宽，adr[1:0]代表着写入时的字内地址，综合考虑二者即可拼接出最终要写入 DRAM 的 32 位数据。

读：

```
always @(*) begin
    case (adr[1:0])
        2'b00 : rdata_b = rdata[7:0];
        2'b01 : rdata_b = rdata[15:8];
        2'b10 : rdata_b = rdata[23:16];
        2'b11 : rdata_b = rdata[31:24];
        default: rdata_b = 8'b0;
    endcase
end

always @(*) begin
    rdata_h = adr[1] ? rdata[31:16] : rdata[15:0];
end

always @(*) begin
    case (op)
        `RW_SB : rdo = {{24{rdata_b[7]}}, rdata_b};
        `RW_UB : rdo = {24'b0, rdata_b};
        `RW_SH : rdo = {{16{rdata_h[15]}}, rdata_h};
        `RW_UH : rdo = {16'b0, rdata_h};
        default: rdo = rdata;
    endcase
end
```

op 代表着读数据时的位宽和符号要求，adr[1:0]代表着读数据在 DRAM 中的字内

地址。首先，我们可以根据 adr[1:0]确定在按字节读和按半字读的情况下读数据分别应该是多少，然后根据 op 来确定最终的读数据和对该数据的扩展方式。

Controller：

部件	接口信号	属性	位宽	功能/释义
Controller	opcode	输入	7	指令的操作码
	funct3	输入	3	指令的 3 位功能码
	funct7	输入	7	指令的 7 位功能码
	npc_op	输出	2	控制 npc 的产生
	npc_osel	输出	1	控制 npc 的产生（仅在面对无条件跳转指令时起作用）
	rf_we	输出	1	寄存器堆的写使能信号
	rf_wsel	输出	2	选择要写回寄存器的数据
	sext_op	输出	3	控制立即数的扩展
	alu_op	输出	4	控制 ALU 的运算类型
	alua_sel	输出	1	选择源操作数 A 的数据来源
	alub_sel	输出	1	选择源操作数 B 的数据来源
	rw_op	输出	3	控制读/写存储器的位宽
	ram_we	输出	1	DRAM 的写使能信号

关键实现：（以 alu_op 的产生为例）

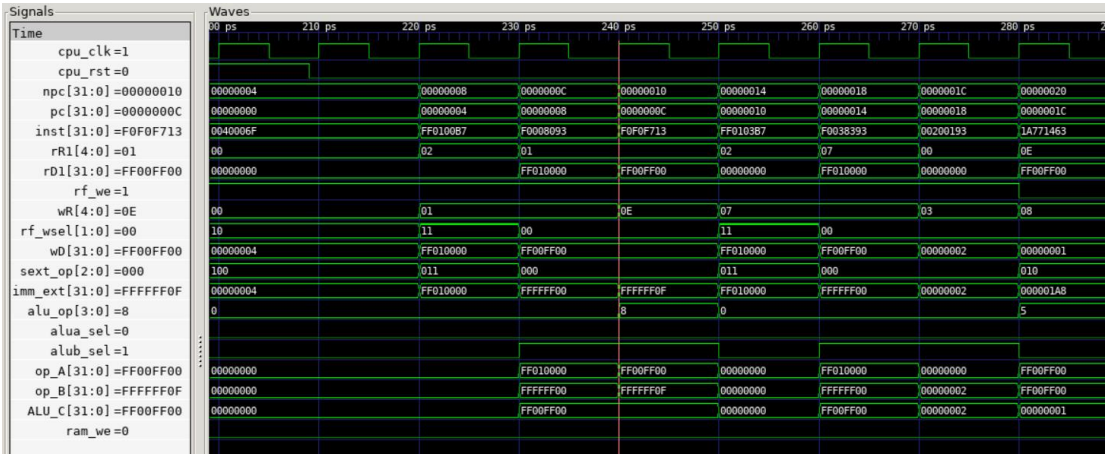
```
always @(*) begin
    if (opcode == 7'b0110011 && funct3 == 3'b000 && funct7 == 7'b0100000)
        alu_op = `ALU_SUB;
    else if (opcode[4:0] == 5'b10011) begin
        case (funct3)
            3'b111 : alu_op = `ALU_AND;
            3'b110 : alu_op = `ALU_OR;
            3'b100 : alu_op = `ALU_XOR;
            3'b001 : alu_op = `ALU_SLL;
            3'b101 : alu_op = (funct7 == 7'b0) ? `ALU_SRL : `ALU_SRA;
            3'b010 : alu_op = `ALU_LT;
            3'b011 : alu_op = `ALU_LTU;
            default: alu_op = `ALU_ADD;
        endcase
    end
    else if (opcode == 7'b1100011) begin
        case (funct3)
            3'b000 : alu_op = `ALU_EQ;
            3'b001 : alu_op = `ALU_NE;
            3'b100 : alu_op = `ALU_LT;
            3'b110 : alu_op = `ALU_LTU;
            3'b101 : alu_op = `ALU_GE;
            3'b111 : alu_op = `ALU_GEU;
            default: alu_op = `ALU_ADD;
        endcase
    end
    else
        alu_op = `ALU_ADD;
end
```

根据指令的操作码以及功能码，确定其对应的各控制信号的值。

1.3 单周期 CPU 仿真及结果分析

要求：包含逻辑运算、访存、分支跳转三类指令的仿真截图及波形分析；每类指令的截图和分析中，至少包含 1 条具体指令；截图需包含信号名和关键信号。

逻辑运算指令：（以 andi 指令为例）

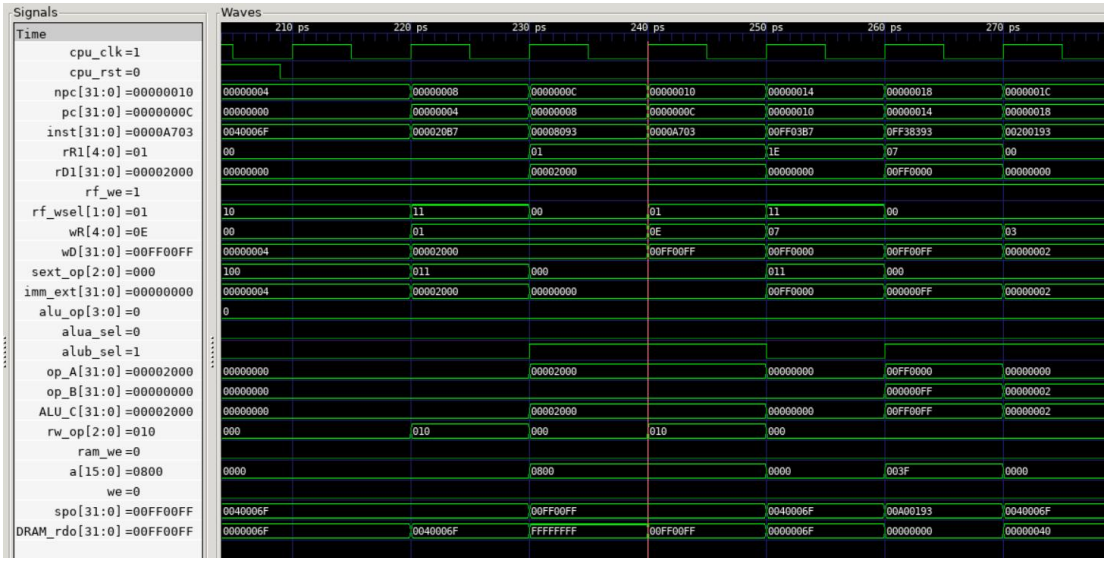


由波形可知：

从 210ps 开始，CPU 复位信号 cpu_rst 为 0，pc 每逢时钟上升沿更新一次，而 npc 总是领先 pc 一个时钟周期，符合预期。

240ps 时，pc 在时钟上升沿处变为 0000000C，同时 inst 变为 F0F0F713（即 11110000111100001110111100010011），rR1 变为 00001，rf_we 变为 1，wR 变为 01110，rf_wsel 变为 00，sext_op 变为 000，imm_ext 变为 FFFFFF0F（即 11111111111111111111111100001111），alu_op 变为 1000，alua_sel 变为 0，alub_sel 变为 1，ram_we 变为 0，符合预期；此时，npc=00000010=pc+4，op_A=rD1=FF00FF00，op_B=imm_ext=FFFFFF0F，wD=ALU_C=FF00FF00=op_A&op_B，亦符合预期。

访存指令：（以 lw 指令为例）

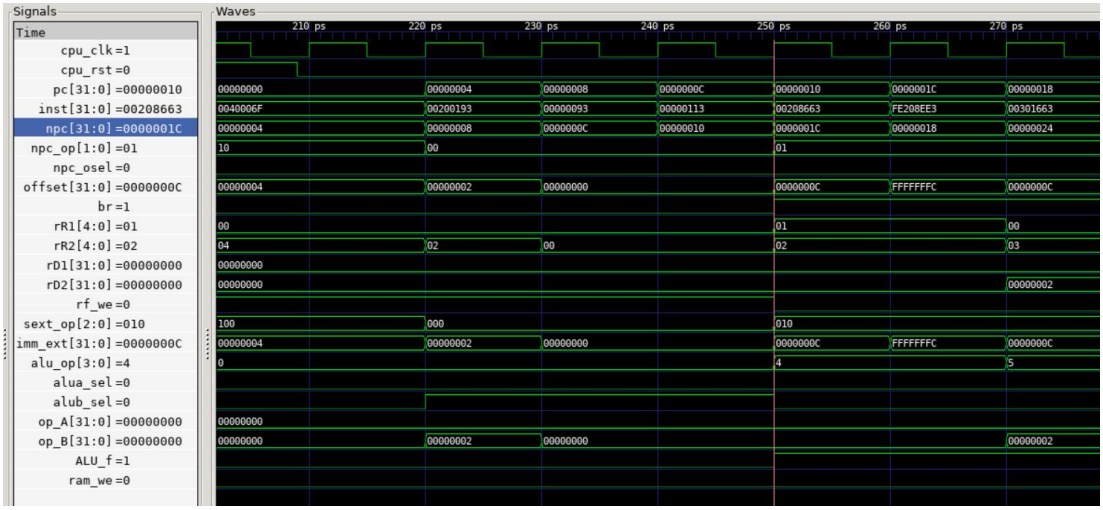


由波形可知：

从 210ps 开始，CPU 复位信号 cpu_rst 为 0，pc 每逢时钟上升沿更新一次，而

npc 总是领先 pc 一个时钟周期，符合预期。
240ps 时，pc 在时钟上升沿处变为 0000000C，同时 inst 变为 0000A703（即 00000000000000001010011100000011），rR1 变为 00001，rf_we 变为 1，rf_wsel 变为 01，wR 变为 01110，sext_op 变为 000，imm_ext 变为 00000000（即 00000000000000000000000000000000），alu_op 变为 0（即 0000），alua_sel 变为 0，alub_sel 变为 1，rw_op 变为 010，ram_we 变为 0，符合预期；此时，npc=00000010=pc+4，op_A=rD1=00002000，op_B=imm_ext=00000000，ALU_C=00002000=op_A+op_B，we=ram_we=0，a={2'b0, ALU_C[15:2]}=0800，wD=DRAM_rdo=spo=00FF00FF，亦符合预期。

分支跳转指令：（以 beq 指令为例）

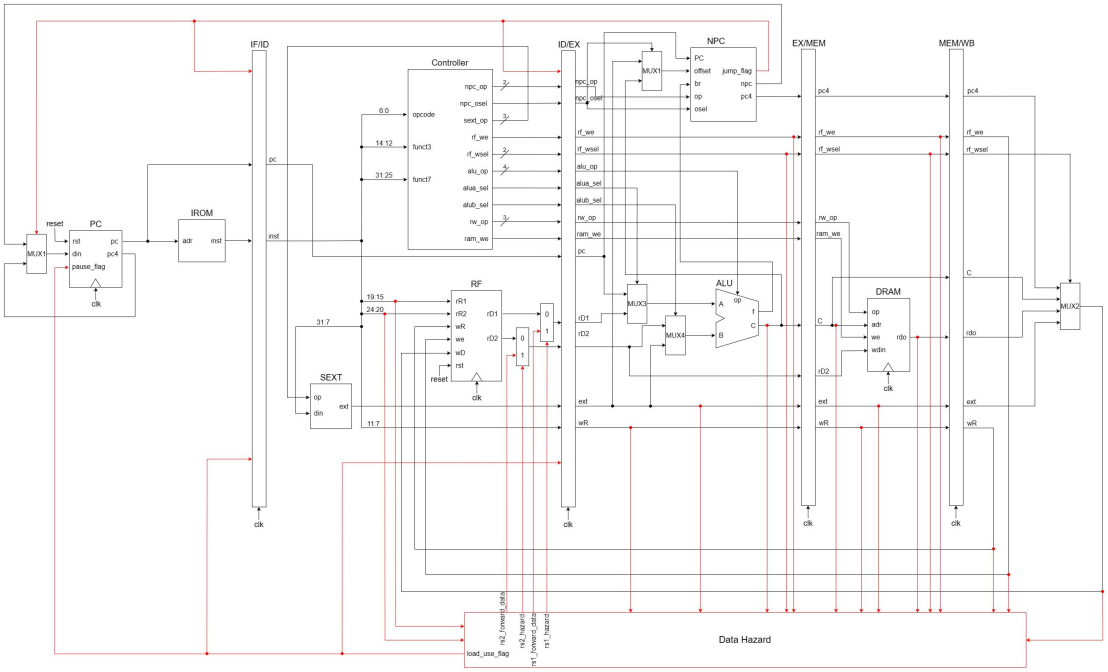


由波形可知：
从 210ps 开始，CPU 复位信号 cpu_rst 为 0，pc 每逢时钟上升沿更新一次，而 npc 总是领先 pc 一个时钟周期，符合预期。
250ps 时，pc 在时钟上升沿处变为 00000010，同时 inst 变为 00208663（即 00000000001000001000011001100011），npc_op 变为 01，npc_osel 变为 0，rR1 变为 00001，rR2 变为 00010，rf_we 变为 0，sext_op 变为 010，imm_ext 变为 0000000C（即 00000000000000000000000000001100），alu_op 变为 4（即 0100），alua_sel 变为 0，alub_sel 变为 0，ram_we 变为 0，符合预期；
与此同时，offset=imm_ext=0000000C，op_A=rD1=00000000，op_B=rD2=00000000，br=ALU_f=1，npc=0000001C=pc+offset，显然，该指令在源操作数 A、B 相等的情况下发生了分支跳转，符合预期。

2 流水线 CPU 设计与实现

2.1 流水线 CPU 数据通路

要求：贴出完整的流水线数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。此外，数据通路图应当能体现出流水线是如何划分的，并用文字阐述每个流水级具备什么功能、需要完成哪些操作。



功能：
PC：存储当前指令的地址；
IROM：指令存储器（只读）；
SEXT：根据指令类型对指令中的立即数进行扩展；
RF：寄存器堆，用于完成读/写寄存器操作；
Controller：控制单元，根据指令类型产生相应的控制信号；
NPC：确定下一条指令的地址并输出 PC+4 的值；
ALU：对源操作数 A 和 B 进行算术运算/逻辑运算/移位运算/大小判断等运算；
DRAM：数据存储器，可读可写；
Data Hazard：数据冒险处理器；
IF/ID：IF/ID 流水线寄存器，用于保存 IF 流水级输出的数据信息，并将其交给 ID 流水级处理，同时共享给其他指令；
ID/EX：ID/EX 流水线寄存器，用于保存 ID 流水级输出的数据信息，并将其交给 EX 流水级处理，同时共享给其他指令；
EX/MEM：EX/MEM 流水线寄存器，用于保存 EX 流水级输出的数据信息，并将其交给 MEM 流水级处理，同时共享给其他指令；
MEM/WB：MEM/WB 流水线寄存器，用于保存 MEM 流水级输出的数据信息，

并将其交给 WB 流水级处理，同时共享给其他指令。
流水线：
IF 流水级：包括 PC 模块和 IROM 模块，其功能为取指，需要完成的操作有选择 PC 模块的输入源、更新当前指令地址、根据当前指令地址从 IROM 中取出指令以及将当前指令及其地址传给下一流水级。
ID 流水级：包括 SEXT 模块、RF 模块和 Controller 模块，其功能为译码，需要完成的操作有对指令中的立即数进行扩展、读寄存器、生成控制信号以及将之后会用到的数据与信号传给下一流水级。
EX 流水级：包括 NPC 模块和 ALU 模块，其功能为执行，需要完成的操作有根据相关控制信号生成下一条指令的地址并将其传回 IF 流水级、选择 ALU 模块两操作数的数据来源、对操作数进行一系列的运算、将相关数据及信号传给下一流水级。
MEM 流水级：包括 DRAM 模块，其功能为访存，需要完成的操作有按位宽要求处理读/写数据、将处理后的写数据写入 DRAM、将处理后的读数据传给下一流水级、继续传递那些后面会用到的数据及信号。
WB 流水级：其功能为写回，需要完成的操作有根据相关控制信号从多个数据来源中选出要写回寄存器的数据、将写使能信号、写寄存器编号、写数据等传回 ID 流水级。

2.2 流水线 CPU 模块详细设计

要求：以表格的形式列出所有与单周期不同的部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等形象化工具和手段，详细描述这些部件的关键实现。此外，如果实现了冒险控制，必须结合数据通路图，详细说明数据冒险、控制冒险的解决方法。

PC：

部件	接口信号	属性	位宽	功能/释义
PC	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	din	输入	32	下一条指令地址
	pause_flag	输入	1	暂停标志
	pc	输出	32	当前指令地址
	pc4	输出	32	pc+4 的值

关键实现：

```
assign if_npc = jump_flag ? ex_npc : if_pc4;

PC u_PC (
    .clk (clk),
    .rst (rst),
    .din (if_npc),
    .pause_flag (pause_flag),
    .pc (if_pc),
    .pc4 (if_pc4)
);
```

```
always @(posedge clk or posedge rst) begin
    if (rst)
        pc <= 32'b0;
    else if (pause_flag)
        pc <= pc;
    else
        pc <= din;
end
```

正常执行时，pc 在时钟上升沿处更新（一般情况下，pc←pc+4；指令发生跳转时，pc←ex_npc）；CPU 暂停时，pc 保持不变；CPU 复位时，pc 清零。

IF/ID：

部件	接口信号	属性	位宽	功能/释义
IF/ID	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	if_pc	输入	32	IF 阶段输出的 pc（当前指令地址）
	if_inst	输入	32	IF 阶段输出的 inst（指令）
	pause_flag	输入	1	暂停标志
	flush_flag	输入	1	寄存器清空标志
	id_pc	输出	32	传到 ID 阶段的 pc
	id_inst	输出	32	传到 ID 阶段的 inst

关键实现：

```
always @(posedge clk or posedge rst) begin
    if (rst)
        id_pc <= 32'b0;
    else if (pause_flag)
        id_pc <= id_pc;
    else
        id_pc <= if_pc;
end

always @(posedge clk or posedge rst) begin
    if (rst)
        id_inst <= 32'b0;
    else if (flush_flag)
        id_inst <= 32'b0;
    else if (pause_flag)
        id_inst <= id_inst;
    else
        id_inst <= if_inst;
end
```

正常执行时，id_pc 和 id_inst 在时钟上升沿处更新；CPU 复位时，id_pc 和 id_inst 清零；寄存器清空时，id_inst 在时钟上升沿处清零；CPU 暂停时，id_pc 和 id_inst 保持不变。

ID/EX：

部件	接口信号	属性	位宽	功能/释义
ID/EX	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	id_npc_op	输入	2	ID 阶段输出的 npc_op
	id_npc_osel	输入	1	ID 阶段输出的 npc_osel
	id_rf_we	输入	1	ID 阶段输出的 rf_we
	id_rf_wsel	输入	2	ID 阶段输出的 rf_wsel
	id_alu_op	输入	4	ID 阶段输出的 alu_op
	id_alua_sel	输入	1	ID 阶段输出的 alua_sel
	id_alub_sel	输入	1	ID 阶段输出的 alub_sel
	id_rw_op	输入	3	ID 阶段输出的 rw_op
	id_ram_we	输入	1	ID 阶段输出的 ram_we
	id_pc	输入	32	ID 阶段输出的 pc
	id_rD1	输入	32	ID 阶段输出的 rD1
	id_rD2	输入	32	ID 阶段输出的 rD2
	id_wR	输入	5	ID 阶段输出的 wR
	id_imm_ext	输入	32	ID 阶段输出的 imm_ext
	pause_flag	输入	1	暂停标志
	flush_flag	输入	1	寄存器清空标志
	ex_npc_op	输出	2	传到 EX 阶段的 npc_op
	ex_npc_osel	输出	1	传到 EX 阶段的 npc_osel
	ex_rf_we	输出	1	传到 EX 阶段的 rf_we

	ex_rf_wsel	输出	2	传到 EX 阶段的 rf_wsel
	ex_alu_op	输出	4	传到 EX 阶段的 alu_op
	ex_alua_sel	输出	1	传到 EX 阶段的 alua_sel
	ex_alub_sel	输出	1	传到 EX 阶段的 alub_sel
	ex_rw_op	输出	3	传到 EX 阶段的 rw_op
	ex_ram_we	输出	1	传到 EX 阶段的 ram_we
	ex_pc	输出	32	传到 EX 阶段的 pc
	ex_rD1	输出	32	传到 EX 阶段的 rD1
	ex_rD2	输出	32	传到 EX 阶段的 rD2
	ex_wR	输出	5	传到 EX 阶段的 wR
	ex_imm_ext	输出	32	传到 EX 阶段的 imm_ext

关键实现：（以 npc_op 为例）

```
always @(posedge clk or posedge rst) begin
    if (rst)
        ex_npc_op <= 2'b0;
    else if (pause_flag || flush_flag)
        ex_npc_op <= 2'b0;
    else
        ex_npc_op <= id_npc_op;
end
```

正常执行时，ex_npc_op 在时钟上升沿处更新；CPU 复位时，ex_npc_op 清零；CPU 暂停或寄存器清空时，ex_npc_op 在时钟上升沿处清零。

NPC：

部件	接口信号	属性	位宽	功能/释义
NPC	PC	输入	32	当前指令地址
	offset	输入	32	指令跳转的偏移量
	br	输入	1	条件分支的跳转标志
	op	输入	2	控制 npc 的产生
	osel	输入	1	控制 npc 的产生（仅在面对无条件跳转指令时起作用）
	jump_flag	输出	1	指令跳转标志
	npc	输出	32	下一条指令地址
	pc4	输出	32	PC+4 的值

关键实现：

```
assign pc4 = PC + 4;
assign jump_flag = ((op == `NPC_BRN) & br) | (op == `NPC_JMP);

always @(*) begin
    case (op)
        `NPC_PC4 : npc = PC + 4;
        `NPC_BRN : npc = br ? (PC + offset) : (PC + 4);
        `NPC_JMP : npc = osel ? offset : (PC + offset);
        default: npc = 32'b0;
    endcase
end
```

综合考虑 op、br、osel 这三个控制信号，根据其值的不同，为 npc 选出不同的数据来源。
当且仅当（1）当前指令为条件分支指令，且条件成立；或者（2）当前指令为无条件跳转指令时，jump_flag 才会为 1，代表指令发生跳转。

EX/MEM：

部件	接口信号	属性	位宽	功能/释义
EX/MEM	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	ex_pc4	输入	32	EX 阶段输出的 pc4
	ex_rf_we	输入	1	EX 阶段输出的 rf_we
	ex_rf_wsel	输入	2	EX 阶段输出的 rf_wsel
	ex_rw_op	输入	3	EX 阶段输出的 rw_op
	ex_ram_we	输入	1	EX 阶段输出的 ram_we
	ex_ALU_C	输入	32	EX 阶段输出的 ALU_C
	ex_rD2	输入	32	EX 阶段输出的 rD2
	ex_wR	输入	5	EX 阶段输出的 wR
	ex_imm_ext	输入	32	EX 阶段输出的 imm_ext
	mem_pc4	输出	32	传到 MEM 阶段的 pc4
	mem_rf_we	输出	1	传到 MEM 阶段的 rf_we
	mem_rf_wsel	输出	2	传到 MEM 阶段的 rf_wsel
	mem_rw_op	输出	3	传到 MEM 阶段的 rw_op
	mem_ram_we	输出	1	传到 MEM 阶段的 ram_we
	mem_ALU_C	输出	32	传到 MEM 阶段的 ALU_C
	mem_rD2	输出	32	传到 MEM 阶段的 rD2
	mem_wR	输出	5	传到 MEM 阶段的 wR
	mem_imm_ext	输出	32	传到 MEM 阶段的 imm_ext

关键实现：（以 pc4 为例）

```
always @(posedge clk or posedge rst) begin
    if (rst)
        mem_pc4 <= 32'b0;
    else
        mem_pc4 <= ex_pc4;
end
```

正常执行时，mem_pc4 在时钟上升沿处更新；CPU 复位时，mem_pc4 清零。

MEM/WB：

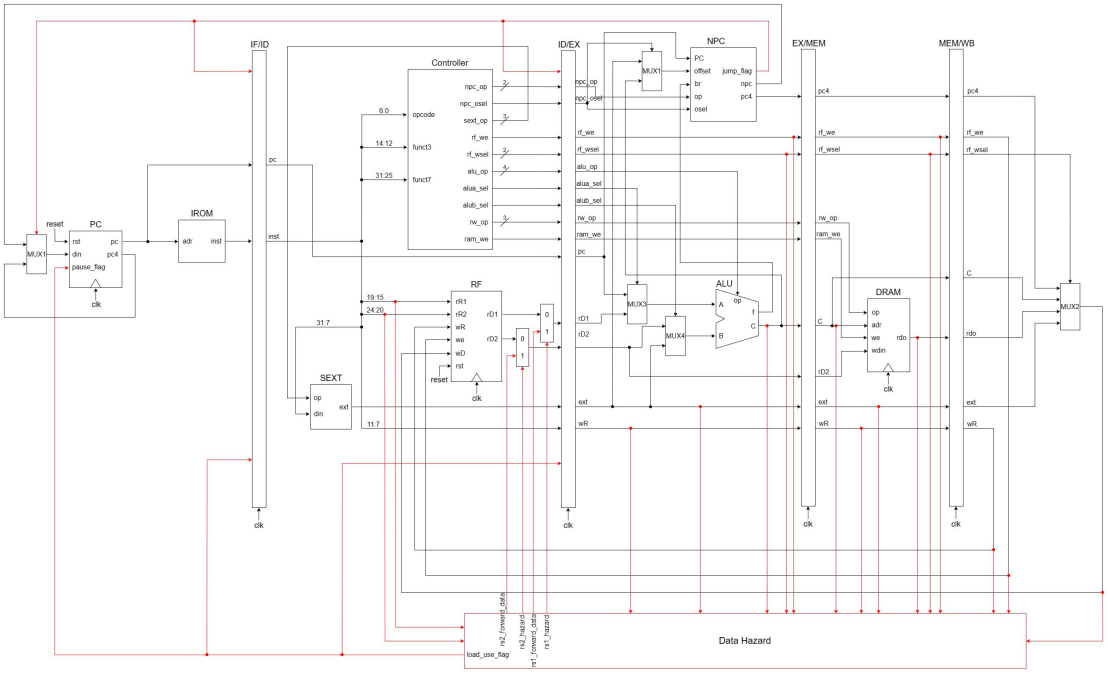
部件	接口信号	属性	位宽	功能/释义
MEM/WB	clk	输入	1	时钟信号
	rst	输入	1	复位信号
	mem_pc4	输入	32	MEM 阶段输出的 pc4
	mem_rf_we	输入	1	MEM 阶段输出的 rf_we
	mem_rf_wsel	输入	2	MEM 阶段输出的 rf_wsel

	mem_ALU_C	输入	32	MEM 阶段输出的 ALU_C
	mem_DRAM_rdo	输入	32	MEM 阶段输出的 DRAM_rdo
	mem_wR	输入	5	MEM 阶段输出的 wR
	mem_imm_ext	输入	32	MEM 阶段输出的 imm_ext
	wb_pc4	输出	32	传到 WB 阶段的 pc4
	wb_rf_we	输出	1	传到 WB 阶段的 rf_we
	wb_rf_wsel	输出	2	传到 WB 阶段的 rf_wsel
	wb_ALU_C	输出	32	传到 WB 阶段的 ALU_C
	wb_DRAM_rdo	输出	32	传到 WB 阶段的 rD2
	wb_wR	输出	5	传到 WB 阶段的 wR
	wb_imm_ext	输出	32	传到 WB 阶段的 imm_ext

关键实现：（以 pc4 为例）

```
always @(posedge clk or posedge rst) begin
    if (rst)
        wb_pc4 <= 32'b0;
    else
        wb_pc4 <= mem_pc4;
end
```

正常执行时，wb_pc4 在时钟上升沿处更新；CPU 复位时，wb_pc4 清零。



数据冒险的解决：

EX 型数据冒险、MEM 型数据冒险、WB 型数据冒险：前递

载入-使用型数据冒险：暂停+前递

dataHazard：

部件	接口信号	属性	位宽	功能/释义
dataHazard	if_id_rs1	输入	5	IF/ID 寄存器输出的源寄存器 1 的编号
	if_id_rs2	输入	5	IF/ID 寄存器输出的源寄存器 2 的编号
	id_ex_rf_we	输入	1	ID/EX 寄存器输出的 rf_we
	id_ex_rf_wsel	输入	2	ID/EX 寄存器输出的 rf_wsel
	id_ex_wR	输入	5	ID/EX 寄存器输出的 wR
	ex_ALU_C	输入	32	EX 阶段输出的 ALU_C
	id_ex_imm_ext	输入	32	ID/EX 寄存器输出的 imm_ext
	ex_mem_rf_we	输入	1	EX/MEM 寄存器输出的 rf_we
	ex_mem_rf_wsel	输入	2	EX/MEM 寄存器输出的 rf_wsel
	ex_mem_wR	输入	5	EX/MEM 寄存器输出的 wR
	ex_mem_ALU_C	输入	32	EX/MEM 寄存器输出的 ALU_C
	mem_DRAM_rdo	输入	32	MEM 阶段输出的 DRAM_rdo
	ex_mem_imm_ext	输入	32	EX/MEM 寄存器输出的 imm_ext
	mem_wb_rf_we	输入	1	MEM/WB 寄存器输出的 rf_we
	mem_wb_wR	输入	5	MEM/WB 寄存器输出的 wR
	wb_wD	输入	32	WB 阶段输出的 wD
	rs1_hazard	输出	1	源寄存器 1 的数据冒险标志
	rs2_hazard	输出	1	源寄存器 2 的数据冒险标志
	rs1_forward_data	输出	32	前递给源寄存器 1 的数据
	rs2_forward_data	输出	32	前递给源寄存器 2 的数据
	load_use_flag	输出	1	加载-使用型冒险标志

实现：

```
assign rs1_id_ex_hazard = (id_ex_wR != 5'b0) & (if_id_rs1 == id_ex_wR) & id_ex_rf_we;
assign rs2_id_ex_hazard = (id_ex_wR != 5'b0) & (if_id_rs2 == id_ex_wR) & id_ex_rf_we;
assign rs1_id_mem_hazard = (ex_mem_wR != 5'b0) & (if_id_rs1 == ex_mem_wR) & ex_mem_rf_we;
assign rs2_id_mem_hazard = (ex_mem_wR != 5'b0) & (if_id_rs2 == ex_mem_wR) & ex_mem_rf_we;
assign rs1_id_wb_hazard = (mem_wb_wR != 5'b0) & (if_id_rs1 == mem_wb_wR) & mem_wb_rf_we;
assign rs2_id_wb_hazard = (mem_wb_wR != 5'b0) & (if_id_rs2 == mem_wb_wR) & mem_wb_rf_we;
assign rs1_hazard = rs1_id_ex_hazard | rs1_id_mem_hazard | rs1_id_wb_hazard;
assign rs2_hazard = rs2_id_ex_hazard | rs2_id_mem_hazard | rs2_id_wb_hazard;
```

```
assign load_use_flag = (id_ex_rf_wsel == `WB_RAM) & (id_ex_wR != 5'b0) & ((if_id_rs1 == id_ex_wR) | (if_id_rs2 == id_ex_wR)) & id_ex_rf_we;
```

```
always @(*) begin
    if (rs1_id_ex_hazard) begin
        if (id_ex_rf_wsel == `WB_ALU)
            rs1_forward_data = ex_ALU_C;
        else
            rs1_forward_data = id_ex_imm_ext;
    end
    else if (rs1_id_mem_hazard) begin
        case (ex_mem_rf_wsel)
            `WB_ALU : rs1_forward_data = ex_mem_ALU_C;
            `WB_RAM : rs1_forward_data = mem_DRAM_rdo;
            default: rs1_forward_data = ex_mem_imm_ext;
        endcase
    end
    else if (rs1_id_wb_hazard)
        rs1_forward_data = wb_wD;
    else
        rs1_forward_data = 32'b0;
end
```

```
always @(*) begin
    if (rs2_id_ex_hazard) begin
        if (id_ex_rf_wsel == `WB_ALU)
            rs2_forward_data = ex_ALU_C;
        else
            rs2_forward_data = id_ex_imm_ext;
    end
    else if (rs2_id_mem_hazard) begin
        case (ex_mem_rf_wsel)
            `WB_ALU : rs2_forward_data = ex_mem_ALU_C;
            `WB_RAM : rs2_forward_data = mem_DRAM_rdo;
            default: rs2_forward_data = ex_mem_imm_ext;
        endcase
    end
    else if (rs2_id_wb_hazard)
        rs2_forward_data = wb_wD;
    else
        rs2_forward_data = 32'b0;
end
```

首先，通过比对 ID 流水级的源寄存器编号与 EX/MEM/WB 流水级的目的寄存器编号可以判断出源寄存器是否存在数据冒险；然后，对于存在数据冒险的源寄存器，我们需要根据数据冒险发生的阶段和相关控制信号确定要向其前递的数据；最后，如果发生的是 EX 型数据冒险，且前一条指令要写回目的寄存器的数据来源于 DRAM，那就说明发生了载入-使用型冒险，要把 load_use_flag 置 1。

```
assign id_rD1 = rs1_hazard ? rs1_forward_data : rD1;
assign id_rD2 = rs2_hazard ? rs2_forward_data : rD2;
```

此外，在 dataHazard 模块的外部，数据冒险标志及要前递的数据会被传入 IF 流水级中。在此，前递数据会与源寄存器中存储的数据进行“竞争”——如果未产生数据冒险，源寄存器中存储的数据就会作为真正的 rD 被传入下一流水级中；反之，前递数据就会作为真正的 rD 被传到后面。而在面对加载-使用型冒险时，由于单纯的前递无法解决问题，所以我们必须先将后续指令暂停一个时钟周期，也就是说，当 load_use_flag 为 1 时，各流水级寄存器和 PC 模块中的 pause_flag 信号也应该为 1，因而，我们应该把 load_use_flag 信号和 pause_flag 信号连在一起，这样一来，当 load_use_flag 变为 1 之后，紧接着在下一个时钟上升沿处，pc 就会保持不变，传到 EX 阶段的各控制信号也会清零（控制信号一清零，load_use_flag 就会恢复低电平，那么从下一个时钟周期开始，后续指令就会恢复正常执行），从而达到暂停指令一个时钟周期的目的。

控制冒险的解决：

分支预测：总是预测不跳转；如果跳转（预测失败），则需清空流水线中分支指令之后的所有指令，并重新取指。

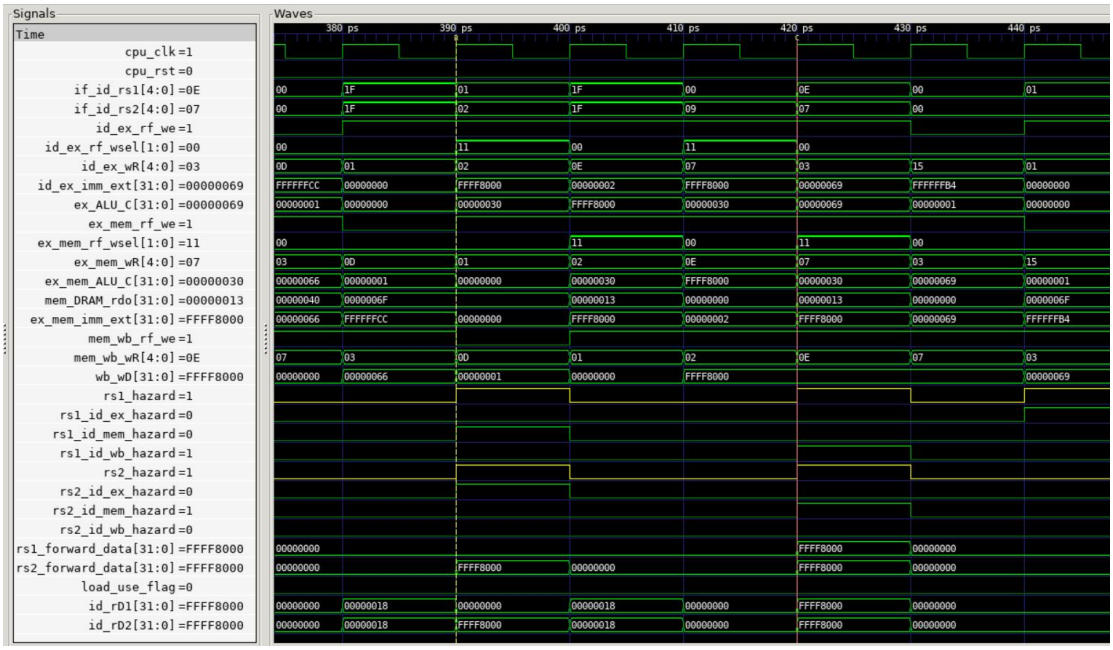
具体实现：

在每个时钟上升沿处，默认 pc←pc+4；而 NPC 模块中产生的指令跳转信号 jump_flag 实际上就是预测失败标志，把该信号接到各个模块中的 flush_flag 信号上即可实现 CPU 在指令发生分支跳转（即预测失败）时清空 IF/ID、ID/EX 流水线寄存器及重置 pc 的功能，进而达到清空指令、重新取指的效果。

2.3 流水线 CPU 仿真及结果分析

要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。若仅实现了理想流水，则此处贴上理想流水的仿真截图及详细的波形分析。

数据冒险：



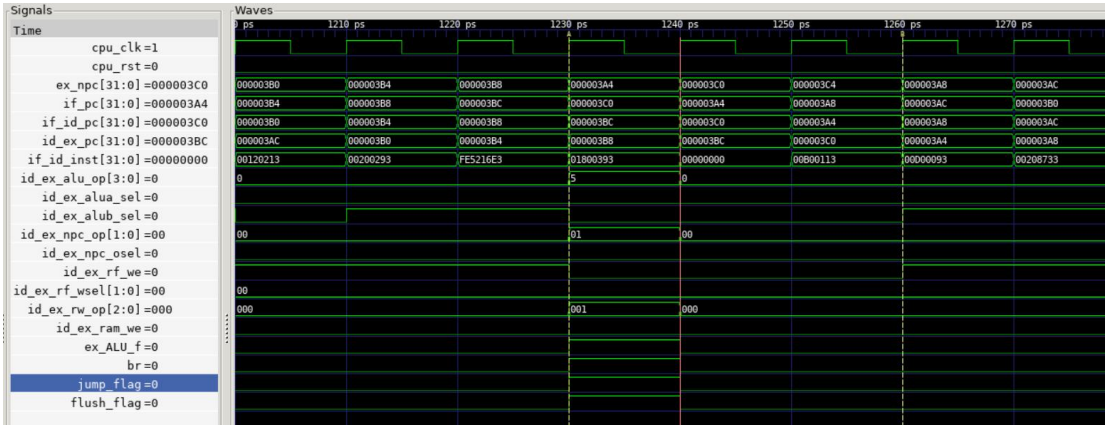
由波形可知：

390ps 时，if_id_rs1 为 00001，if_id_rs2 为 00010，id_ex_rf_we 为 1，id_ex_wR 为 00010，ex_mem_rf_we 为 1，ex_mem_wR 为 00001，mem_wb_rf_we 为 1，mem_wb_wR 为 01101，显然，rs1 存在 MEM 型数据冒险，rs2 存在 EX 型数据冒险，而此时 rs1_id_ex_hazard 为 0，rs1_id_mem_hazard 为 1，rs1_id_wb_hazard 为 0，rs1_hazard 为 1，rs2_id_ex_hazard 为 1，rs2_id_mem_hazard 为 0，rs2_id_wb_hazard 为 0，rs2_hazard 为 1，符合预期；与此同时，ex_mem_rf_wsel 为 00，id_ex_rf_wsel 为 11，意味着 rs2 面临的不是载入-使用型数据冒险，且要前递给 rs1 的数据来源于 ex_mem_ALU_C，要前递给 rs2 的数据来源于 id_ex_imm_ext，此时，load_use_flag 为 0，rs1_forward_data=00000000=ex_mem_ALU_C，rs2_forward_data=FFFF8000=id_ex_imm_ext，符合预期；此外，该时刻，id_rD1 变为 00000000，id_rD2 变为 FFFF8000，分别与 rs1_forward_data、rs2_forward_data 相等，符合预期。

420ps 时，if_id_rs1 为 01110，if_id_rs2 为 00111，id_ex_rf_we 为 1，id_ex_wR 为 00011，ex_mem_rf_we 为 1，ex_mem_wR 为 00111，mem_wb_rf_we 为 1，mem_wb_wR 为 01110，显然，rs1 存在 WB 型数据冒险，rs2 存在 MEM 型数据冒险，而此时 rs1_id_ex_hazard 为 0，rs1_id_mem_hazard 为 0，rs1_id_wb_hazard 为 1，rs1_hazard 为 1，rs2_id_ex_hazard 为 0，rs2_id_mem_hazard 为 1，rs2_id_wb_hazard 为 0，rs2_hazard 为 1，load_use_flag 为 0，符合预期；与此同时，ex_mem_rf_wsel 为 11，意味着要前递给 rs2 的数据来源于 ex_mem_imm_ext，此时，rs1_forward_data=FFFF8000=wb_wD（发生 WB 型冒险时要前递的数据只可能是 wb_wD），rs2_forward_data=FFFF8000=ex_mem_imm_ext，符合预

期；此外，该时刻，id_rD1 变为 FFFF8000，id_rD2 变为 FFFF8000，分别与 rs1_forward_data、rs2_forward_data 相等，符合预期。

控制冒险：



由波形可知：

1230ps 时，id_ex_npc_op 为 01，br=ex_ALU_f=0，意味着出现了条件分支指令且条件为真，此时，jump_flag 变为 1，flush_flag 也变为 1，符合预期。

1240ps 时，在时钟上升沿处，if_pc 变为 000003A4，if_id_inst 变为 00000000，id_ex_alu_op 等由 ID/EX 流水线寄存器输出的控制信号都变为 0，jump_flag 与 flush_flag 都恢复低电平，表明预测失败后，寄存器被清空，pc 被重置，符合预期。

1240ps 到 1260ps 期间，if_pc 与 if_id_pc 依次变为 000003A4，表明跳转后的指令在沿着流水线正常流动，符合预期；与此同时，ID/EX 流水线寄存器输出的各控制信号始终为 0，表明这个时间段正处在 EX 阶段的两条指令（其地址分别为 000003BC、000003C0）实为空指令，符合预期。

1260ps 时，id_ex_pc 变为 000003A4，EX 阶段的各控制信号恢复正常，表明跳转后的指令已经沿着流水线到达了 EX 级，符合预期。

3 设计过程中遇到的问题及解决方法

要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。

1. 在设计数据冒险处理模块时，最初，我并没有对前递到 rs1 和 rs2 的数据进行区分（也就是说我每次前递到两个源寄存器的数据都是一样的），其原因主要在于我没有考虑到源寄存器 rs1 和 rs2 同时发生数据冒险的情况。对此，只需设置两个数据寄存器，让其分别存储要前递到两个源寄存器的数据即可。此外，我在一开始对前递数据寄存器进行赋值的时候，对前递数据的数据来源考虑得不够全面，忽略了发生 MEM 型数据冒险时要前递的数据也可能来源于 ALU_C 等情况，造成了一些错误。对此，只需传入各流水级的 rf_wsel 信号及数据信号，并在必要时根据相应的 rf_wsel 信号对前递数据的数据来源作出选择即可。
2. 在进行 trace 测试时，我经常会遇到一些比较奇怪的数据错误，而在经过波形分析和代码检查后，我发现，这些错误往往来源于模块连接时的接线错误或者接口信号的位宽错误，这就突出了细心和命名规范的重要性。

4 总结

要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。

在本次计算机设计与实践课程期间，我先是从零开始一步一步地完成了单周期 CPU 的设计，然后又在单周期 CPU 的基础上实现了五级流水线 CPU，在这个过程中，我不仅重新掌握了 Verilog 的相关知识与自顶向下的结构化设计方法，还加深了自己对于汇编、CPU、总线、外设、SoC 相关知识的理解，提升了自己分析波形与解决问题的能力，受益匪浅。除此之外，由于本次实验的工程量较大，耗费时间也较多，所以当我通过了 trace 测试，并在开发板上成功运行了汇编程序之后，我体会到了巨大的成就感，也正是这一点让我深刻感受到了计算机硬件设计与数字逻辑的独特魅力。