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 的 SoC 设计

项目类型：综合设计型

课程学时：56 地点：T2612

学生班级：21 级计算机 3 班

学生学号：210110315

学生姓名：吕弋卿

评阅教师：

报告成绩：

实验与创新实践教育中心制

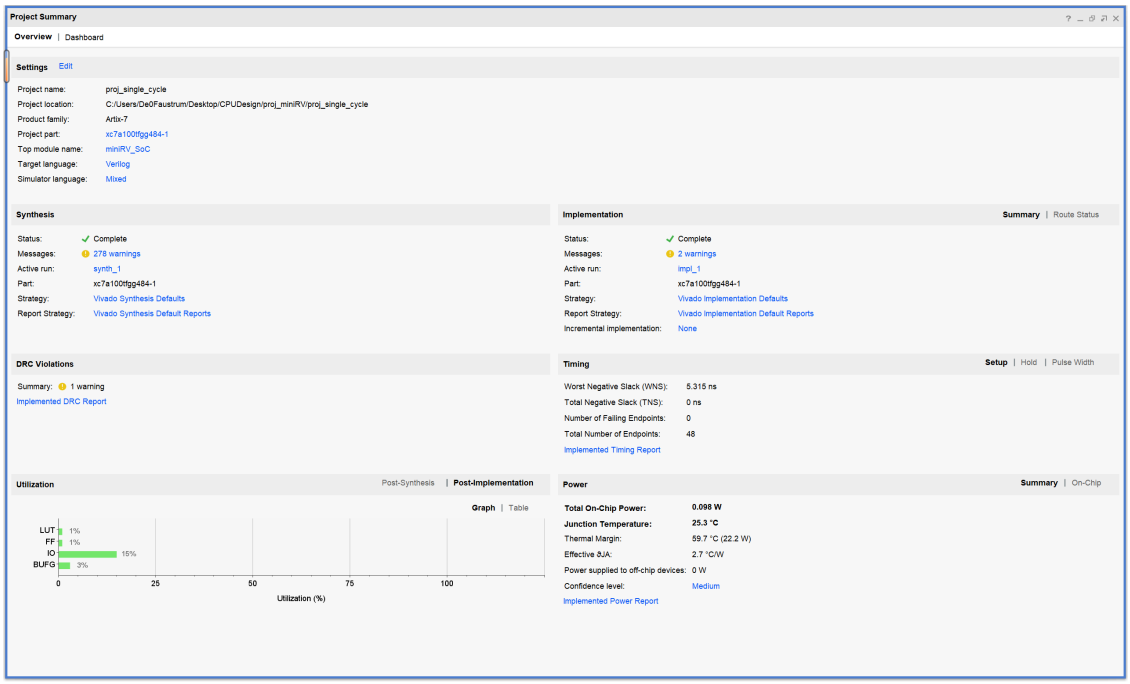
2023 年 7 月

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）。

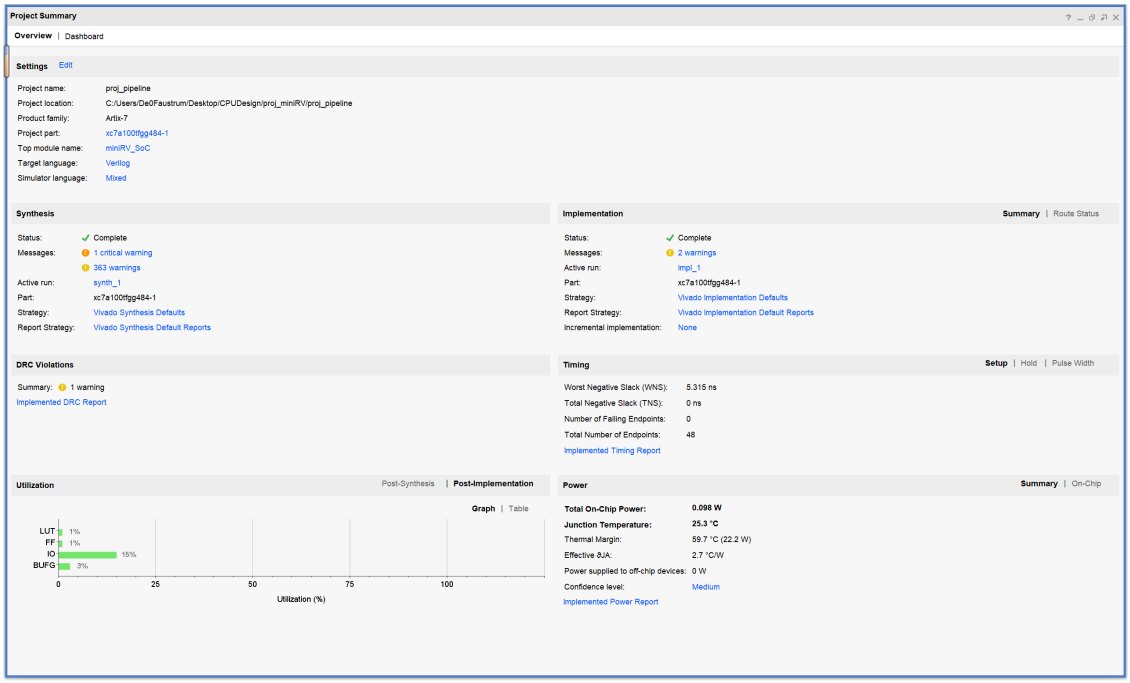
设计概述（罗列出所有实现的指令，以及单周期/流水线 CPU 频率）
设计实现了单周期和流水线的必做 24 条指令，包括： Type-R: and, sub, and, or, xor, sll, srl, sra Type-I: addi, andi, ori, xori, slli, srli, srai, lw, jalr Type-S: sw Type-B: beq, bne, blt, bge Type-U: lui Type-J: jal 单周期频率为 25MHZ，流水线频率为 50MHZ
设计的主要特色（除基本要求以外的设计）
1.I/O 总线 在单周期 CPU 和流水线 CPU 中，使用 I/O 总线来连接 CPU 和外设。对于 CPU 来说，外设和内存被统一成一个整体，可以使用一组地址线、数据线和使能信号进行管理。 2.数据前递 通过采用前传机制来解决三种数据冲突。同时，当这三种数据冲突同时发生时，可以按照正确的优先级进行转发处理。 3.流水线暂停 采用暂停操作来解决加载-使用冲突和分支控制。当我们在执行阶段检测到加载使用冲突时，通过暂停取值和译码阶段，使得译码指令延迟到访存结束后执行，从而解决加载使用冲突。 4 分支处理 (1) 对于 B 型指令，始终预测不跳转。直到执行阶段检测到发生跳转时，通过清除访存和译码阶段的指令，并重新写入 PC 来消除影响。如果未检测到跳转，则继续执行。 (2) 对于跳转指令，我们在执行阶段计算出跳转地址后，清空访存和译码阶段的指令，最后跳转到新计算出的 PC 值。
资源使用、功耗数据截图（Post Implementation；含单周期、流水线 2 个截图）

以下是示例，请贴自己的图。

单周期：



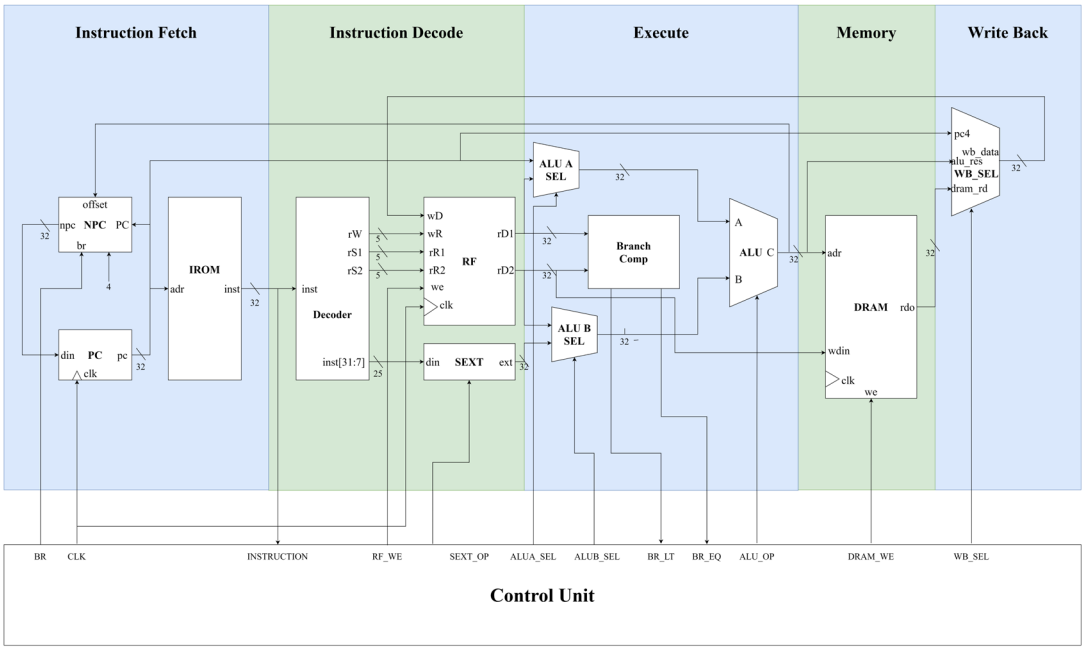
流水线：



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。



模块描述：

1.PC：指令寄存器，用于寄存和输出当前指令的地址
2.NPC：用于更新 PC 值
3.IROM：指令存储器
4.Decoder：译码器，用于解释指令，并送入后续模块
5.RF：寄存器堆，由 32 个 32 位寄存器组成
6.SEXT：立即数扩展生成器
7.ALU A/B SEL：选择 ALU 操作数的多路复用器
8.Branch Comp：用于分子跳转指令中分支的判断
9.ALU：运算器
10.DRAM：数据存储器
11.WBSel：写回多路复用器
12.ControlUnit：控制单元，输入指令产生控制信号

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。

Module PC:

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	rst	复位
Input	wire	1	clk	时钟
Input	wire	32	Din	输入（下一条 PC 值）
output	reg	32	Pc	输出（当前 PC 值）

Core Code:

```
always @(posedge clk or posedge rst) begin
    if(rst) Pc <= 1'B0;
    else Pc <= Din;
end
```

Module NPC:

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	2	Operation	用于选择与指令相对应的下一条 PC 来源
Input	wire	1	Branch	是否分支 Flag
Input	wire	32	Offset	PC 偏移量
Input	wire	32	NpcImmediate	立即数值
Input	wire	32	Pc	输入（当前 PC 值）
Input	reg	32	Npc	输出（下一条 PC 值）
output	wire	32	Pc4	输出（Pc+4 用于 Jalr 等指令）

Core Code:

```
assign Pc4 = Pc+4;

always @(*) begin
    case(Operation)
    `NPC_PC4: Npc = Pc + 4;
    `NPC_JALR: Npc = NpcImmediate;
    `NPC_B: case(Branch) 1'B1: Npc = Pc + Offset; 1'B0: Npc = Pc+4; endcase
    `NPC_JAL: Npc = Pc + Offset;
    endcase
end
```

Module ControlUnit

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	32	Instruction	当前指令码
output	reg	2	NpcOperation	控制信号：NPC 选择
output	reg	1	RegisterFileWriteEnable	控制：寄存器堆写使能
output	reg	2	RegisterFileWriteSelect	控制：写回阶段选择
output	reg	3	SextOperation	控制：立即数生成模式
output	reg	0	AluBselect	控制：Alu 操作数选择
output	reg	0	DramWriteEnable	控制：DRAM 写使能
output	reg	4	AluOperation	控制：ALU 操作控制

Core Code Segment 1 – Generate NpcOperation:

```
always @(*) begin
    case(OperationCode)
        OP_R:   NpcOperation = `NPC_PC4;
        OP_I:   NpcOperation = `NPC_PC4;
        OP_LOAD:NpcOperation = `NPC_PC4;
        OP_LUI: NpcOperation = `NPC_PC4;
        OP_S:   NpcOperation = `NPC_PC4;
        OP_B:   NpcOperation = `NPC_B;
        OP_JAL: NpcOperation = `NPC_JAL;
        OP_JALR:NpcOperation = `NPC_JALR;
        default:NpcOperation = `NPC_PC4;
  endcase
end
```

Core Code Segment 2 – Generate RegisterFileWriteEnable:

```
always @(*) begin
    if(OperationCode == OP_B || OperationCode == OP_S) RegisterFileWriteEnable = 0;
    else RegisterFileWriteEnable  = 1;
end
```

Core Code Segment 3 – Generate RegisterFileWriteSelect:

```
always @(*) begin
    case(OperationCode)
        OP_R:   RegisterFileWriteSelect = `WB_ALU;
        OP_I:   RegisterFileWriteSelect = `WB_ALU;
        OP_LOAD:RegisterFileWriteSelect = `WB_DRAM;
        OP_JALR:RegisterFileWriteSelect = `WB_PC4;
        OP_JAL: RegisterFileWriteSelect = `WB_PC4;
        OP_LUI: RegisterFileWriteSelect = `WB_SEXT;
        default:RegisterFileWriteSelect = `WB_ALU;
  endcase
end
```

Core Code Segment 4 – Generate SextOperation:

```
always @(*) begin
    case(OperationCode)
        OP_B:   SextOperation = `SEXT_B;
        OP_S:   SextOperation = `SEXT_S;
        OP_R:   SextOperation = `SEXT_R;
        OP_LOAD:SextOperation = `SEXT_I;
        OP_JALR:SextOperation = `SEXT_I;
        OP_LUI: SextOperation = `SEXT_U;
        OP_JAL: SextOperation = `SEXT_J;
        OP_I:
            case(Function3)
                3'b001: SextOperation = `SEXT_MOVE;
                3'b101: SextOperation = `SEXT_MOVE;
                default:SextOperation = `SEXT_I;
            endcase
        default:SextOperation = `SEXT_R;
  endcase
end
```

Core Code Segment 5 – Generate AluBSelect:

```
always @ (*) begin
    case(OperationCode)
        OP_I:   AluBSelect = 1;
        OP_LOAD:AluBSelect = 1;
        OP_S:   AluBSelect = 1;
        OP_JALR:AluBSelect = 1;
        default:AluBSelect = 0;
    endcase
end
```

Core Code Segment 6 – Generate DramWriteEnable:

```
always @(*) begin
    if(OperationCode == OP_S) DramWriteEnable = 1;
    else DramWriteEnable = 0;
end
```

Core Code Segment 7 – Generate AluOperation

```
always@(*) begin
    case({OperationCode,Function3})
        {OP_R,3'B000}: AluOperation = Function7[5] ? `ALU_SUB : `ALU_ADD;
        {OP_R,3'B001}: AluOperation = `ALU_SLL;
        {OP_R,3'B010}: AluOperation = `ALU_AND;
        {OP_R,3'B011}: AluOperation = `ALU_AND;
        {OP_R,3'B100}: AluOperation = `ALU_XOR;
        {OP_R,3'B101}: AluOperation = Function7[5] ? `ALU_SRA : `ALU_SRL;
        {OP_R,3'B110}: AluOperation = `ALU_OR;
        {OP_R,3'B111}: AluOperation = `ALU_AND;
        {OP_I,3'B000}: AluOperation = `ALU_ADD;
        {OP_I,3'B001}: AluOperation = `ALU_SLL;
        {OP_I,3'B010}: AluOperation = `ALU_AND;
        {OP_I,3'B011}: AluOperation = `ALU_AND;
        {OP_I,3'B100}: AluOperation = `ALU_XOR;
        {OP_I,3'B101}: AluOperation = Function7[5] ? `ALU_SRA : `ALU_SRL;
        {OP_I,3'B110}: AluOperation = `ALU_OR;
        {OP_I,3'B111}: AluOperation = `ALU_AND;
        {OP_I,3'B000}: AluOperation = `ALU_BEQ;
        {OP_I,3'B001}: AluOperation = `ALU_BNE;
        {OP_I,3'B010}: AluOperation = `ALU_BEQ;
        {OP_I,3'B011}: AluOperation = `ALU_BEQ;
        {OP_I,3'B100}: AluOperation = `ALU_BLT;
        {OP_I,3'B101}: AluOperation = `ALU_BGE;
        {OP_I,3'B110}: AluOperation = `ALU_BEQ;
        {OP_I,3'B111}: AluOperation = `ALU_BEQ;
        {OP_S,3'B???}:   AluOperation = `ALU_ADD;
        {OP_LOAD,3'B???}:AluOperation = `ALU_ADD;
        {OP_JALR,3'B???}:AluOperation = `ALU_ADD;
        default: AluOperation = `ALU_AND;
    endcase
end
```

Module Sext

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	32	Din	当前指令
Input	wire	3	SextOperation	Sext 生成模式
output	reg	32	Ext	立即数输出

Core Code:

```
always @(*) begin
    case(SextOperation)
        `SEXT_R: Ext = 32'H00000000;
        `SEXT_I: Ext = {{20{Din[31]}},{Din[31:20]}};
        `SEXT_S: Ext = {{20{Din[31]}},{Din[31:25]},{Din[11:7]}};
        `SEXT_U: Ext = {{Din[31:12]},{12{1'B0}}};
        `SEXT_B: Ext = {{19{Din[31]}},{Din[31]},{Din[7]},{Din[30:25]},{Din[11:8]},{1'B0}};
        `SEXT_J: Ext = {{11{Din[31]}},{Din[31]},{Din[19:12]},{Din[20]},{Din[30:21]},{1'B0}};
        `SEXT_MOVE: Ext =  {{27{1'B0}},{Din[24:20]}};
    endcase
end
```

Module RF

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	clk	时钟
Input	wire	5	ReadRegister1	RS1
Input	wire	5	ReadRegister2	RS2
Input	wire	5	WriteRegister	RD
Input	wire	1	WriteEnable	写使能
Input	wire	2	RegisterFileWriteSelect	写选择
Input	wire	32	Pc4	PC4，来自 NPC
Input	wire	32	Ext	立即数
Input	wire	32	AluResult	ALU 运算结果
Input	wire	32	DramData	DRAM 中读出的数据
output	wire	32	RegisterData1	rD1
output	wire	32	RegisterData2	rD2
output	reg	32	WriteData	写数据

Core Code:

```
reg [31:0] RegisterFile[31:0];

assign RegisterData1 = (ReadRegister1 == 5'B00000)? 32'H00000000 : RegisterFile[ReadRegister1];
assign RegisterData2 = (ReadRegister2 == 5'B00000)? 32'H00000000 : RegisterFile[ReadRegister2];

always @(*) begin
  case(RegisterFileWriteSelect)
    `WB_ALU : WriteData = AluResult;
    `WB_DRAM: WriteData = DramData;
    `WB_PC4 : WriteData = Pc4;
    `WB_SEXT: WriteData = Ext;
    default : WriteData = Pc4;
  endcase
end

always @(posedge clk) begin
    if(WriteEnable && (WriteRegister != 5'B00000)) RegisterFile[WriteRegister] <= WriteData;
end
```

Moulde ALU:

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	32	Resource1	操作数 1
Input	wire	32	Resource2	操作数 2
Input	wire	32	Immediate	立即数
Input	wire	1	AluBSelect	操作数 2 选择信号（rD2 或立即数）
Input	wire	4	AluOperation	操作符
output	wire	32	AluResult	运算结果
output	wire	1	AluStatus	运算状态，如分支状态

Core Code:

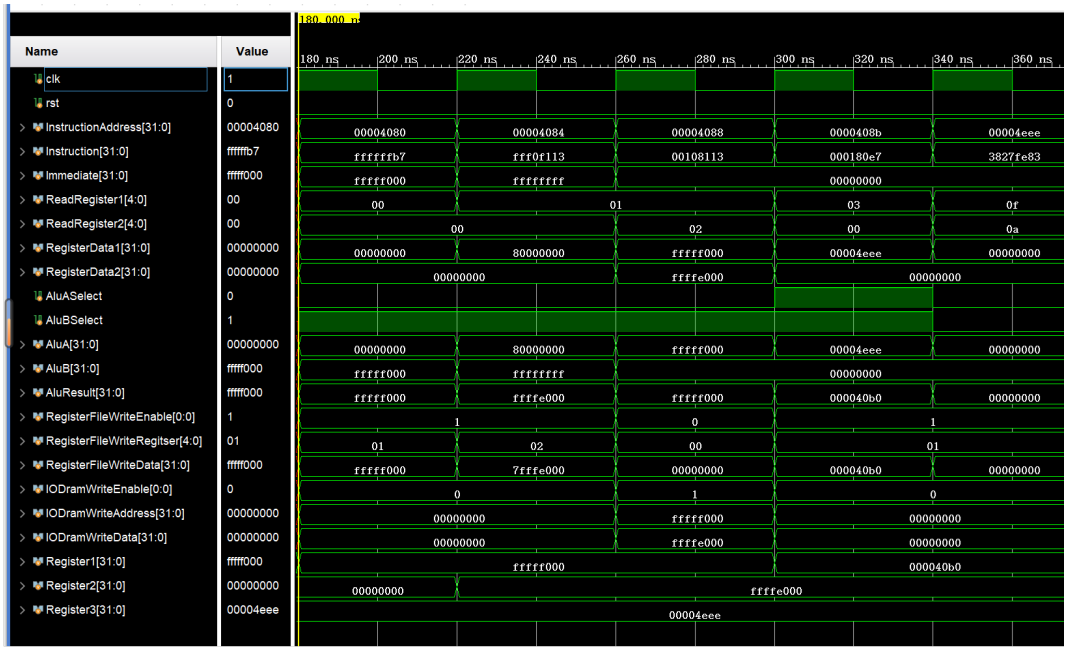
```
always @(*) begin
    case(AluOperation)
        `ALU_ADD: Result = DataA + DataB;
        `ALU_SUB: Result = DataA - DataB;
        `ALU_AND: Result = DataA & DataB;
        `ALU_OR : Result = DataA | DataB;
        `ALU_XOR: Result = DataA ^ DataB;
        `ALU_SLL: Result = DataA << DataB[4:0];
        `ALU_SRL: Result = DataA >> DataB[4:0];
        `ALU_SRA: Result = ($signed(DataA)) >>> DataB[4:0];
        default : Result = 0;
    endcase
end

assign AluResult = Result;

wire[31:0] Discrepancy = DataA - DataB;
reg [0:0]Status;
always @(*) begin
    if(AluOperation == `ALU_BEQ && Discrepancy == 0) Status = 1;
    else if(AluOperation == `ALU_BNE && Discrepancy != 0) Status = 1;
    else if(AluOperation == `ALU_BLT && Discrepancy[31]) Status = 1;
    else if(AluOperation == `ALU_BGE && Discrepancy[31] == 0) Status = 1;
    else Status = 0;
end

assign AluStatus = Status;
```

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。

Simulation Result



Program Segment:

```
4080 lui x1, FFFFF;
4084 addi x2, x1, -1;
4088 sw x2, 0(x1);
408B jalr x1, 0(x3);
.....
4EEE ... ...
```

Analysis:

[180ns] PC 地址为 0x00004080，从 IROM 中读取的指令为 0xFFFFFFB7，译码得指令为 lui x1, FFFFF;将 x1 的值装载为 0xFFFFF000;

[220ns] PC 地址为 0x00004084，从 IROM 中读取的指令为 0xFFF0F113，译码得指令为 addi x2, x1, -1(逻辑运算指令);下面分析该条指令：Immediate 输出为 0xFFFFFFFF，即-1，ReadRegister1 为 00001，表示 RS1=x1；RegisterData1 为 0xFFFFF000，与上一步中的数值符合；AluASelect 为 0，表示 ALU 操作数 1 来自 rD1；AluBSelect 为 1，表示 ALU 操作数 2 来自 Sext 生成的 Immediate；AluResult 为 0xFFFFE000，即为 0xFFFFF000 + (-1)的值，符合运算结果；该指令有写回，因此 RegisterFileWriteEnable（即 RD）为 1，RegisterFileWriteRegister 为 00010，即 x2，RegisterFileWriteData 为 ALU 运算结果 0xFFFFE000；该指令无需访存，因此 IODramWriteEnable 为低电平；Register2 的值被修改为 0xFFFFE000;该条指令符合预期结果。

[260ns] PC 地址为 0x00004088，从 IROM 中读取的指令为 0x00108113，译码得指令为 sw x2, 0(x1)(访存指令);下面分析该条指令：Immediate 输出

为 0x00000000，与偏移量 0 符合，ReadRegister1 为 00001，表示 RS1=x1；RegisterData1 为 0xFFFFF00；ReadRegister2 为 00010，表示 RS2=x2；RegisterData2 为 0xFFFFE000，与上一条指令结果符合；AluASelect 为 0，表示 ALU 操作数 1 来自 rD1；AluBSelect 为 1，表示 ALU 操作数 2 来自 Sext 生成的 Immediate；AluResult 为 0xFFFFF000，即为 0xFFFFF000 + 0 的值，符合运算结果；该指令无写回，因此 RegisterFileWriteEnable 为 0；该指令有访存，因此 IODramWriteEnable 为高电平，IODramWriteAddress 来自 ALU 的运算结果 0xFFFFF000，IODramWriteData 来自 RS2，即 0xFFFFE0000；即该条指令执行了向地址 0xFFFFF000 写入数据 0xFFFFE000;该条指令符合预期结果。

[300ns] PC 地址为 0x0000408B，从 IROM 中读取的指令为 0x0000180E7，译码得指令为 jalr x1, 0(x3)(跳转指令);下面分析该条指令：Immediate 输出为 0x00000000，与偏移量 0 符合，ReadRegister1 为 00003，表示 RS1=x3；RegisterData1 为 0x00004EEE，即寄存器堆中 x3 的值；AluASelect 为 1，表示 ALU 操作数 1 来自 PC4；AluBSelect 为 1，表示 ALU 操作数 2 来自 Sext 生成的 Immediate；AluResult 为 0x000040B0，即为 0x000040B0 + 0 的值，符合运算结果；该指令有写回，RegisterFileWriteEnable 为 1，RegisterFileWriteRegister 为 00001,即 RD=x1，RegisterFileWriteData 为来自 ALU 的 0x000040B0；该指令无访存，因此 IODramWriteEnable 为低电平；Register1 的值被修改为 0x000040B0.

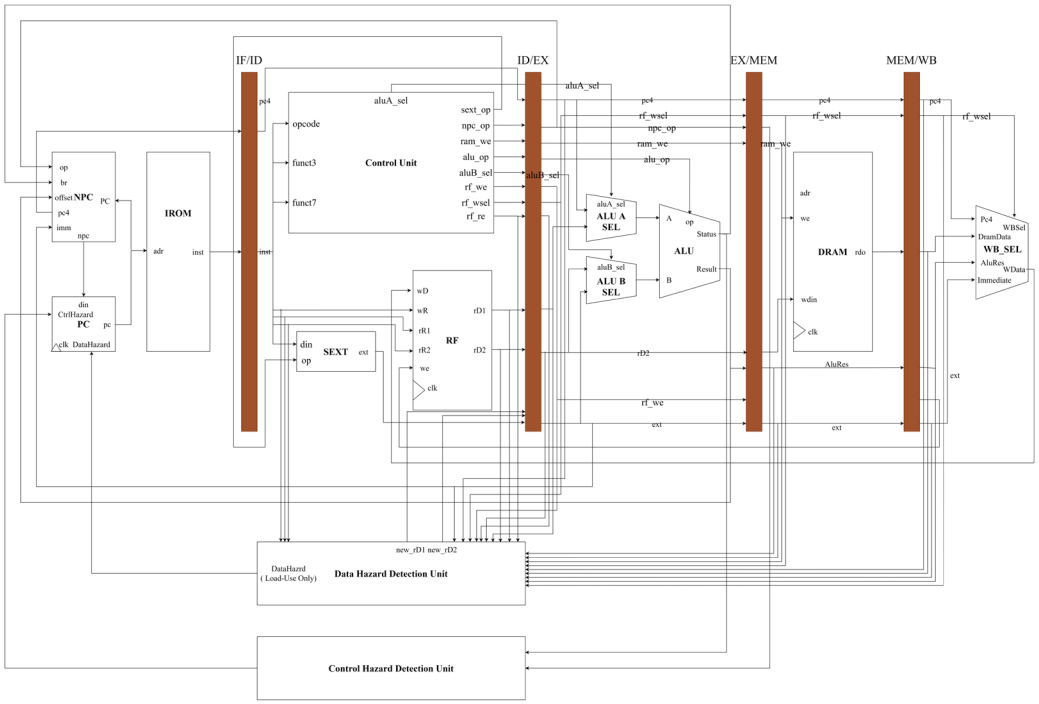
[340ns] PC 地址被修改为 0x00004EEE，即完成了跳转功能，因此该条指令符合预期结果。

综上，分析了 addi,sw,jalr，即运算、访存、跳转指令，得出结论 CPU 功能正常。

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。



一、各模块名称及作用

1.PC：指令寄存器，用于寄存和输出当前指令的地址
2.NPC：用于更新 PC 值
3.IROM：指令存储器
4.Decoder：译码器，用于解释指令，并送入后续模块
5.RF：寄存器堆，由 32 个 32 位寄存器组成
6.SEXT：立即数扩展生成器
7.ALU A/B SEL：选择 ALU 操作数的多路复用器
8.ALU：运算器
9.DRAM：数据存储器
10.WBSel：写回多路复用器
11.ControlUnit：控制单元，输入指令产生控制信号
12.IF/ID：流水线寄存器
13.ID/EX：流水线寄存器
14.EX/MEM：流水线寄存器
15.MEM/WB：流水线寄存器

16.DataHazardDetectionUnit：数据冒险检测模块
16.ControlHazardDetectionUnit：控制冒险检测模块

二、流水线各阶段功能：

（1）IF：取指阶段。在此阶段，保存 PC 的值，并在每个上升沿更新它。通过输出线将 PC 传递到 IROM，获得指令，分析指令类型，将指令划分为不同的部分，并将其传递到解码阶段。
（2）ID：译码阶段。在这个阶段，我们分析访问和写回寄存器号的指令，并生成各种控制信号。同时，我们还读取了寄存器的值。在这个阶段，我们还可以检测数据风险，并通过数据转发进行处理。
（3）EXE：执行阶段。该阶段主要包括两个模块：分支选择模块和 ALU 计算模块。分支选择模块比较从寄存器读取的数据（在转发校正之后），并基于指令类型计算分支选择信号。基于 ALU_OP 的 ALU 计算模块对输入的操作数进行相应的运算，得到运算结果。当然，传递到 ALU 算术单元的操作数是由多路复用器选择的操作数。
（4）MEM：访存阶段。这个阶段是我们的 CPU 访问数据存储器和外围设备的通信平台。存储器访问级的功能不是以模块化的方式实现的，其主要逻辑是连接 CPU 的内部和外部电路。
（5）WB：写回阶段。顾名思义，我们需要将在执行阶段或内存访问阶段获得的值写回寄存器文件。在这一点上，我们可以确保所有必要的数据都已经通过 wd_Sel 选择要写回寄存器的数据。

在这个实验中，我们设计了一个标准的五级流水线 CPU，它由四个寄存器分隔。当时钟的每个上升沿到达时，这四个寄存器同时更新其内容，以便在每个阶段实现有序的数据流。

通过在管道中添加正向路由和控制模块，我们的 CPU 可以高效、正确地处理数据风险和控制风险。

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。

Module PC

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	rst	复位
Input	wire	1	clk	时钟
Input	wire	32	Din	输入（下一条 PC 值）
Input	wire	1	DataHazard	有无数据冒险
Input	wire	1	ControlHazard	有无控制冒险
output	reg	32	Pc	输出（当前 PC 值）

Core Code:

```
module pc(
    input wire [0:0] rst,
    input wire [0:0] clk,
    input wire [31:0] Din,
    input wire DataHazard,
    input wire ControlHazard,
    output reg [31:0] Pc
);

always @(posedge clk or posedge rst) begin
  if(rst) Pc <= 0;
  else if(ControlHazard) Pc<=Din;
  else if(DataHazard) Pc<=Pc;
  else Pc <= Din;
end

endmodule
```

Module NPC – Same As SingleCycle CPU (Skip)

Module Decoder – Same As SingleCycle CPU (Skip)

Module RF – Same As SingleCycle CPU (Skip)

Module Sext – Same As SingleCycle CPU (Skip)

Module ALU – Same As SingleCycle CPU (Skip)

Module WBSel – Same As SingleCycle CPU (Skip)

Module IF/ID

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	clk	时钟
Input	wire	1	rst	复位
Input	wire	32	IFInstruction	IF 阶段的指令
Input	wire	32	IFPc4	IF 阶段的 PC4
Input	wire	1	DataHazard	有无数据冒险
Input	wire	1	ControlHazard	有无控制冒险
output	reg	1	IDInstruction	ID 阶段的指令
output	reg	1	IDPc4	ID 阶段的 PC4

Core Code:

```
always @(posedge clk or posedge rst) begin
    if(rst) IDInstruction <= 0;
    else if(ControlHazard) IDInstruction <= 0;
    else if(DataHazard) IDInstruction <= IDInstruction;
    else IDInstruction <= IFInstruction;
end

always @(posedge clk or posedge rst) begin
    if(rst) IDPc4<=0;
    else if(ControlHazard) IDPc4 <= 0;
    else if(DataHazard) IDPc4 <= IDPc4;
    else IDPc4 <= IFPc4;
end
```

Module ID/EX

Input Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	clk	时钟
Input	wire	1	rst	复位
Input	wire	1	ControlHazard	有无控制冒险
Input	wire	1	DataHazard	有无数据冒险
Input	wire	2	IDNpcOperation	ID 阶段的 NPC 控制符
Input	wire	1	IDDramWriteEnable	ID 阶段的内存写使能
Input	wire	4	IDAluOperation	ID 阶段的 ALU 运算符
Input	wire	1	IDAluBSelect	ID 阶段的 ALU 操作数选择符号
Input	wire	1	IDRegisterFileWriteEnable	ID 阶段的寄存器写使能
Input	wire	2	IDRegisterFileWriteSelect	ID 阶段的寄存器写选择
Input	wire	4	IDWriteRegister	ID 阶段的写寄存器
Input	wire	32	IDPc4	ID 阶段的 PC4
Input	wire	32	IDRegisterData1	ID 阶段的寄存器数据 1
Input	wire	32	IDRegisterData2	ID 阶段的寄存器数据 2
Input	wire	32	IDExt	ID 阶段的立即数
output	reg	2	EXNpcOperation	EX 阶段的 NPC 控制符
output	reg	1	EXDramWriteEnable	EX 阶段的内存写使能
output	reg	4	EXAluOperation	EX 阶段的 ALU 运算符
output	reg	1	EXAluBSelect	EX 阶段的 ALU 操作数选择符号
output	reg	1	EXRegisterFileWriteEnable	EX 阶段的寄存器写使能
output	reg	2	EXRegisterFileWriteSelect	EX 阶段的寄存器写选择
output	reg	5	EXWriteRegister	EX 阶段的写寄存器
output	reg	32	EXPc4	EX 阶段的 PC4
output	reg	32	EXRegisterData1	EX 阶段的寄存器数据 1
output	reg	32	EXRegisterData2	EX 阶段的寄存器数据 2
output	reg	32	EXExt	EX 阶段的立即数

Core Code Segment:

```
always @(posedge clk or posedge rst) begin
    if(rst) EXNpcOperation <= 0;
    else if(ControlHazard | DataHazard) EXNpcOperation <= 0;
    else EXNpcOperation <= IDNpcOperation;
end

always @(posedge clk or posedge rst) begin
    if(rst) EXDramWriteEnable <= 0;
    else if(ControlHazard | DataHazard) EXDramWriteEnable <= 0;
    else EXDramWriteEnable <= IDDramWriteEnable;
end

always @(posedge clk or posedge rst) begin
    if(rst) EXAluOperation <= 0;
    else if(ControlHazard | DataHazard) EXAluOperation <= 0;
    else EXAluOperation <= IDAluOperation;
end
```

Module EX/MEM

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	clk	时钟
Input	wire	1	rst	复位
Input	wire	1	EXDramWriteEnable	EX 阶段内存写使能
Input	wire	1	EXRegisterFileWriteEnable	EX 阶段寄存器写使能
Input	wire	2	EXRegisterFileWriteSelect	EX 阶段寄存器写选择
Input	wire	5	EXWriteRegister	EX 阶段的写寄存器
Input	wire	32	EXPc4	EX 阶段的 PC4
Input	wire	32	EXAluResult	EX 阶段 ALU 运算结果
Input	wire	32	EXRegisterData2	EX 阶段寄存器数据 2
Input	wire	32	EXExt	EX 阶段立即数
output	reg	32	MEMDramWriteEnable	MEM 阶段内存写使能
output	reg	1	MEMRegisterFileWriteEnable	MEM 阶段寄存器写使能
output	reg	1	MEMRegisterFileWriteSelect	MEM 阶段寄存器写选择
output	reg	2	MEMWriteRegister	MEM 阶段写寄存器
output	reg	5	MEMPc4	MEM 阶段的 PC4
output	reg	32	MEMAluResult	MEM 阶段 ALU 运算结果
output	reg	32	MEMRegisterData2	MEM 阶段寄存器数据 2
output	reg	32	MEMExt	MEM 阶段的立即数

Core Code Segment:

```
always @(posedge clk or posedge rst) begin
    if(rst) MEMRegisterFileWriteEnable <= 0;
    else MEMRegisterFileWriteEnable <= EXRegisterFileWriteEnable;
end

always @(posedge clk or posedge rst) begin
    if(rst) MEMRegisterFileWriteSelect <= 0;
    else MEMRegisterFileWriteSelect <= EXRegisterFileWriteSelect;
end

always @(posedge clk or posedge rst) begin
    if(rst) MEMWriteRegister <= 0;
    else MEMWriteRegister <= EXWriteRegister;
end

always @(posedge clk or posedge rst) begin
    if(rst) MEMPc4 <= 0;
    else MEMPc4 <= EXPc4;
end
```

Module MEM/WB

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	1	clk	时钟
Input	wire	1	rst	复位
Input	wire	1	MEMRegisterFileWriteEnable	MEM 阶段寄存器写使能
Input	wire	2	MEMRegisterFileWriteSelect	MEM 阶段寄存器写选择
Input	wire	5	MEMWriteRegister	MEM 阶段写寄存器
Input	wire	32	MEMPc4	MEM 阶段的 PC4
Input	wire	32	MEMAluResult	MEM 阶段 ALU 运算结果
Input	wire	32	MEMDramData	MEM 阶段内存读数据
Input	wire	32	MEMExt	MEM 阶段立即数
output	reg	1	WBRegisterFileWriteEnable	WB 阶段寄存器写使能
output	reg	2	WBRegisterFileWriteSelect	WB 阶段寄存器写选择
output	reg	5	WBWriteRegister	WB 阶段写寄存器
output	reg	32	WBPc4	WB 阶段 PC4
output	reg	32	WBAluResult	WB 阶段 ALU 运算结果
output	reg	32	WBDramData	WB 阶段内存读数据
output	reg	32	WBExt	WB 阶段立即数

Module DataHazardDetector

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	5	IDReadRegister1	ID 阶段的读寄存器 1
Input	wire	5	IDReadRegister2	ID 阶段的读寄存器 2
Input	wire	2	IDRegisterFileReadEnable	ID 阶段的寄存器写使能
Input	wire	32	IDRegisterData1	ID 阶段的寄存器读数据 1
Input	wire	32	IDRegisterData2	ID 阶段的寄存器读数据 2
Input	wire	5	EXWriteRegister	EX 阶段的写寄存器
Input	wire	1	EXRegisterFileReadEnable	EX 阶段的寄存器写使能
Input	wire	2	EXRegisterFileReadSelect	EX 阶段的寄存器写选择
Input	wire	32	EXPc4	EX 阶段的 PC4
Input	wire	32	EXExt	EX 阶段的立即数
Input	wire	32	EXAluResult	EX 阶段的 ALU 运算结果
Input	wire	5	MEMWriteRegister	MEM 阶段的写寄存器
Input	wire	1	MEMRegisterFileReadEnable	MEM 阶段的寄存器写使能
Input	wire	2	MEMRegisterFileReadSelect	MEM 阶段的寄存器写选择
Input	wire	32	MEMPc4	MEM 阶段的 PC4
Input	wire	32	MEMExt	MEM 阶段的立即数
Input	wire	32	MEMAluResult	MEM 阶段的 ALU 运算结果
Input	wire	32	MEMDramData	MEM 阶段的内存读数据
Input	wire	5	WBWriteRegister	WB 阶段的写寄存器
Input	wire	1	WBRegisterFileReadEnable	WB 阶段的寄存器写使能
Input	wire	2	WBRegisterFileReadSelect	WB 阶段的寄存器写选择
Input	wire	32	WBPc4	WB 阶段的 PC4
Input	wire	32	WBExt	WB 阶段的立即数
Input	wire	32	WBAluResult	WB 阶段的 ALU 运算结果
Input	wire	32	WBDramData	WB 阶段的内存读数据
output	reg	32	NewRegisterData1	更新后的寄存器数据 1
output	reg	32	NewRegisterData2	更新后的寄存器数据 2
output	wire	1	DataHazard	有无 Load-Use 型数据冒险

Core Code Segement 1:

```
//A,ID and EX
wire ReadRegisterTypeA1 = (IDReadRegister1 == EXWriteRegister) & EXRegisterFileWriteEnable &
    IDRegisterFileReadEnable[0] & (IDReadRegister1 != 5'B00000);
wire ReadRegisterTypeA2 = (IDReadRegister2 == EXWriteRegister) & EXRegisterFileWriteEnable &
    IDRegisterFileReadEnable[1] & (IDReadRegister2 != 5'B00000);
//B,ID and MEM
wire ReadRegisterTypeB1 = (IDReadRegister1 == MEMWriteRegister) & MEMRegisterFileWriteEnable &
    IDRegisterFileReadEnable[0] & (IDReadRegister1 != 5'B00000);
wire ReadRegisterTypeB2 = (IDReadRegister2 == MEMWriteRegister) & MEMRegisterFileWriteEnable &
    IDRegisterFileReadEnable[1] & (IDReadRegister2 != 5'B00000);
//C,ID and WB
wire ReadRegisterTypeC1 = (IDReadRegister1 == WBWriteRegister) & WBRegisterFileWriteEnable &
    IDRegisterFileReadEnable[0] & (IDReadRegister1 != 5'B00000);
wire ReadRegisterTypeC2 = (IDReadRegister2 == WBWriteRegister) & WBRegisterFileWriteEnable &
    IDRegisterFileReadEnable[1] & (IDReadRegister2 != 5'B00000);

//Only for load-use
assign DataHazard = (ReadRegisterTypeA1 && EXRegisterFileWriteSelect == `WB_DRAM) ||
    (ReadRegisterTypeA2 && EXRegisterFileWriteSelect == `WB_DRAM);
```

Core Code Segement 2 – Update RegisterData1 (Similar To 2, Which Is Skipped)

```
always @(*) begin
    if(ReadRegisterTypeA1) begin
        case(EXRegisterFileWriteSelect)
            `WB_PC4: NewRegisterData1 = EXPc4;
            `WB_SEXT:NewRegisterData1 = EXExt;
            `WB_ALU: NewRegisterData1 = EXAluResult;
            default: NewRegisterData1 = EXAluResult;
        endcase
    end
    else if(ReadRegisterTypeB1) begin
        case(MEMRegisterFileWriteSelect)
            `WB_PC4: NewRegisterData1 = MEMPc4;
            `WB_SEXT:NewRegisterData1 = MEMExt;
            `WB_ALU: NewRegisterData1 = MEMAluResult;
            `WB_DRAM:NewRegisterData1 = MEMDramData;
            default: NewRegisterData1 = MEMAluResult;
        endcase
    end
    else if(ReadRegisterTypeC1) begin
        case(WBRegisterFileWriteSelect)
            `WB_PC4: NewRegisterData1 = WBPc4;
            `WB_SEXT:NewRegisterData1 = WBExt;
            `WB_ALU: NewRegisterData1 = WBAluResult;
            `WB_DRAM:NewRegisterData1 = WBDramData;
            default: NewRegisterData1 = WBAluResult;
        endcase
    end
    else NewRegisterData1 = IDRegisterData1;
end
```

Module ControlHazardDetector

Input/ Output	Signal Type	Bit Width	Signal Name	Discription
Input	wire	2	EXNpcOperation	EX 阶段的 NPC 操作符
Input	wire	1	AluStatus	ALU 运算状态（有无分支跳转）
output	reg	1	ControlHazard	有无控制冒险

Core Code:

```
always @(*) begin
    if(EXNpcOperation == `NPC_JALR || EXNpcOperation == `NPC_JAL) ControlHazard = 1'B1;
    else if(EXNpcOperation == `NPC_B && AluStatus == 1) ControlHazard = 1'B1;
    else ControlHazard = 1'B0;
end
```

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。

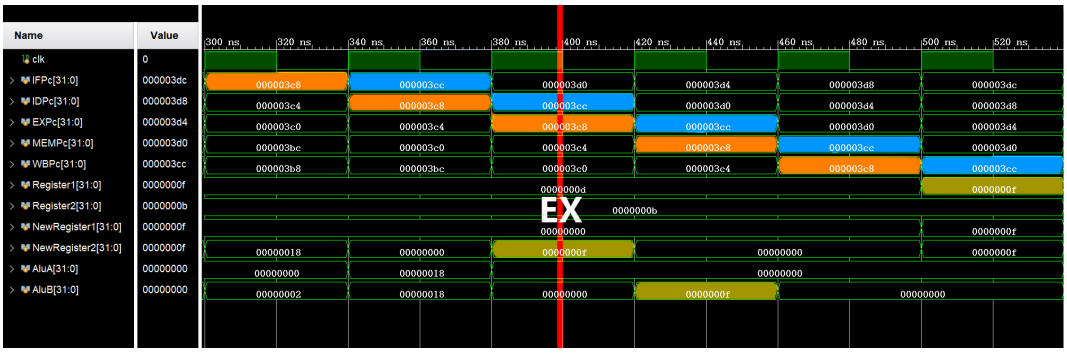
一、数据冒险

1.Raw-A 型

测试代码：

```
000003c8 <test_1035>:
     3c8: 00f00093          addi  x1,x0,15
     3cc: 00100133          add   x2,x0,x1
     3d0: 00f00393          addi  x7,x0,15
     3d4: 40b00193          addi  x3,x0,1035
     3d8: e8711ce3          bne   x2,x7,270
     3dc: 00140413          addi  x8,x8,1
     3e0: fffff0b7          lui   x1,0xfffff
     3e4: 0080a023          sw x8,0(x1)
     3e8: 000f80e7          jalr  x1,0(x31)
```

测试波形



波形分析：

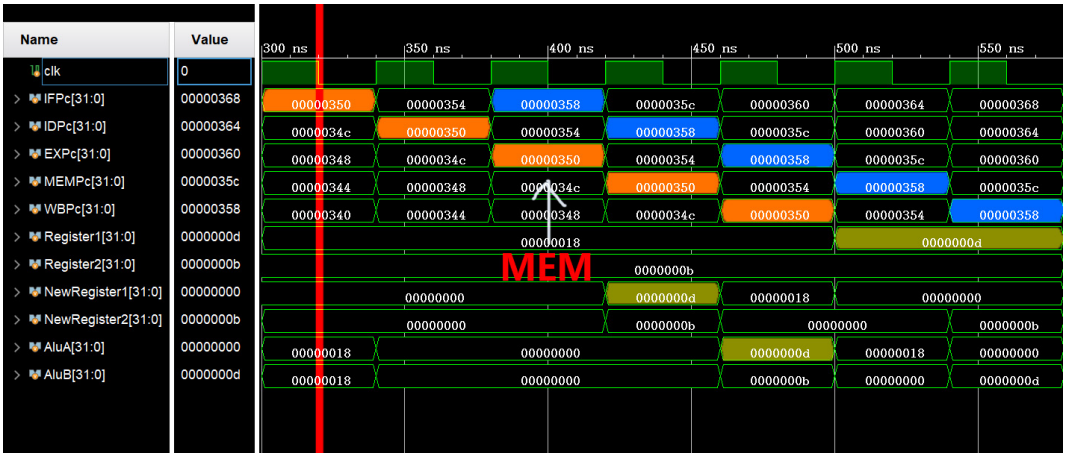
在指令 pc=0x000003C8 的 WB 阶段结束后，x1 的值变为 0x0000000F。然而，在 pc=0x000003C8 的指令的 EX 阶段需要使用 x1，而此时 x1 的值仍然是原来的 0x0000000D，因此需要使用数据前递来解决问题。指令 pc=0x000003C8 在 EX 阶段时，指令 pc=0x000003C8 在 ID 阶段；此时，Data Hazard Detector 检测到数据冒险发生，并对 RegisterData2 进行前递：将前一指令的新值 NewRegister2=0x0000000F 作为 ALU 操作数 2 传递给 ALU。因此，在指令 pc=0x000003CC 到达 EX 阶段时，从 ID/EX 寄存器获取的 RegisterData2 为 0x0000000F，成功地进行了数据前递，解决了数据冒险。

2.Raw-B 型

测试代码：

```
0000034c <test_1020>:
     34c: 00000213          addi   x4,x0,0
     350: 00d00093          addi   x1,x0,13
     354: 00b00113          addi   x2,x0,11
     358: 00208733          add    x14,x1,x2
     35c: 00070313          addi   x6,x14,0
     360: 00120213          addi   x4,x4,1
     364: 00200293          addi   x5,x0,2
     368: fe5214e3          bne    x4,x5,350
     36c: 01800393          addi   x7,x0,24
     370: 3fc00193          addi   x3,x0,1020
     374: ee731ee3          bne    x6,x7,270
```

测试波形：



波形分析：

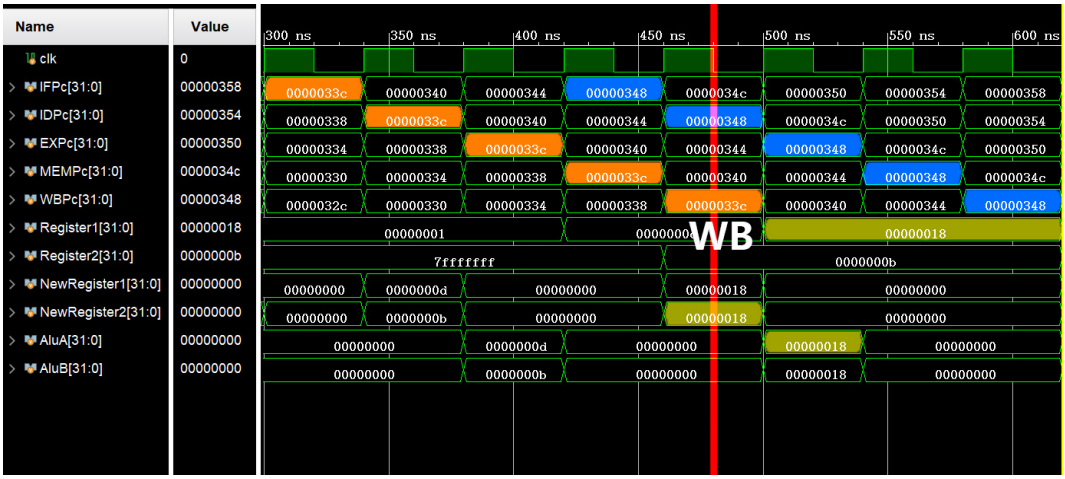
在指令 pc=0x00000350 的 WB 阶段结束后，x1 的值为 0x0000000D。然而，在 pc=0x00000358 的指令的 EX 阶段需要使用 x1，而此时 x1 的值仍为 0x00000018，所以需要进行数据前递。指令 pc=0x00000350 在 MEM 阶段时，指令 pc=0x00000358 在 ID 阶段。此时，Data Hazard Detector 检测到数据冒险发生，并进行了数据前递：将前一指令的新值 NewRegister=0x0000000D 作为 ALU 操作数 1 传递给 ALU。当指令 pc=0x00000358 到达 EX 阶段时，从 ID/EX 寄存器获取的 RegisterData1 为 0x0000000D，成功地进行了数据前递，解决了数据冒险。

3.Raw-C 型

测试代码：

```
00000334 <test_1017>:
     334: 00d00093          addi   x1,x0,13
     338: 00b00113          addi   x2,x0,11
     33c: 002080b3          add    x1,x1,x2
     340: 01800393          addi   x7,x0,24
     344: 3f900193          addi   x3,x0,1017
     348: f27094e3          bne    x1,x7,270
```

测试波形：



波形分析：

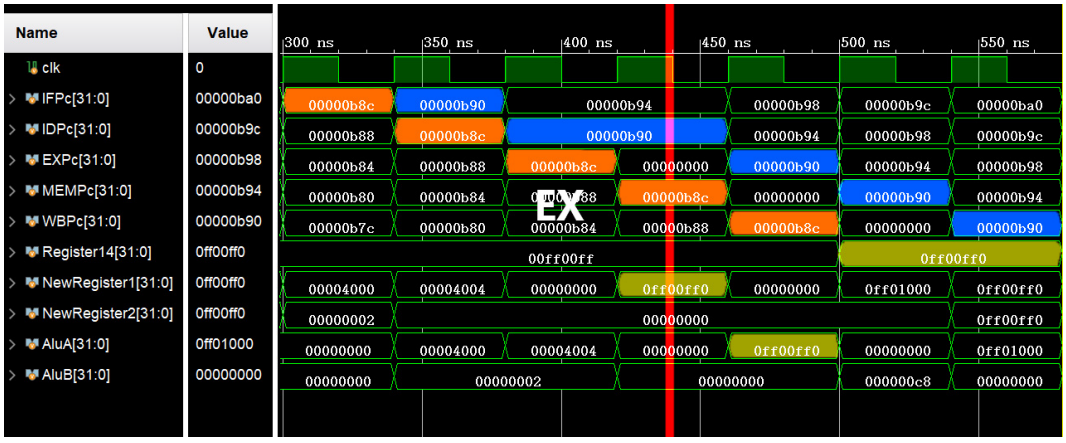
在指令 pc=0x0000033C 的 WB 阶段结束后，x1 的值为 0x00000018。然而，在 pc=0x00000348 的指令的 EX 阶段需要使用 x1，而此时 x1 的值仍为 0x0000000D，所以需要进行数据前递。指令 pc=0x0000033C 在 WB 阶段时，指令 pc=0x00000348 在 ID 阶段。此时，Data Hazard Detector 检测到数据冒险发生，并进行了数据前递：将前一指令的新值 NewRegister=0x00000018 作为 ALU 操作数 1 传递给 ALU。当指令 pc=0x00000348 到达 EX 阶段时，从 ID/EX 寄存器获取的 RegisterData1 为 0x00000018，成功地进行了数据前递，解决了数据冒险。

4.Load-Use 型

测试代码：

```
00000b78 <test_12012>:
    b78:   000031b7          lui  x3,0x3
    b7c:   eec18193          addi    x3,x3,-276
    b80:   00000213          addi    x4,x0,0
    b84:   000040b7          lui  x1,0x4
    b88:   00408093          addi    x1,x1,4 # 4004
    b8c:   0040a703          lw   x14,4(x1)
    b90:   00070313          addi    x6,x14,0
    b94:   0ff013b7          lui  x7,0xff01
    b98:   ff038393          addi    x7,x7,-16
    b9c:   ec731a63          bne x6,x7,270
    ba0:   00120213          addi    x4,x4,1
    ba4:   00200293          addi    x5,x0,2
    ba8:   fc521ee3          bne x4,x5,b84
```

测试波形：



波形分析：

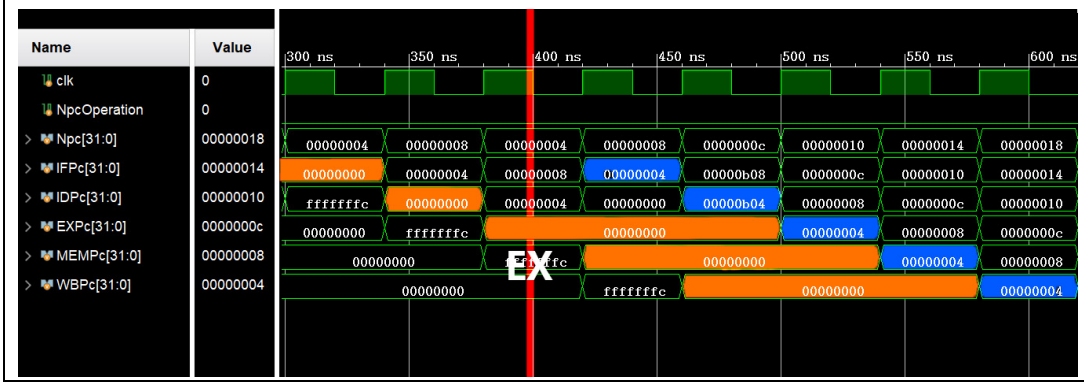
在 pc=0x00000B8C 的指令的写回阶段结束后，x14 的值为 0x0FF00FF0。然而，要从 DRAM 内读取这个值，需要等到 MEM 阶段。而 pc=0x00000B90 的指令在执行阶段需要使用 x14，但 x14 的值仍然是 0x00FF00FF，因此需要进行前向传递。在 pc=0x00000B8C 的指令执行阶段时，pc=0x00000B90 的指令在译码阶段。此时，数据冒险检测器检测到发生了 Load-Use 型数据冒险，于是插入了一个气泡，导致流水线停顿，并通过前向传递解决了对 RegisterData1 的依赖：将前向传递过来的 NewRegister 值 0x0FF00FF0 作为 ALU 操作数 1 送入 ALU。当 pc=0x00000B90 的指令到达执行阶段时，从 ID/EX 寄存器获取的 RegisterData1 值是 0x0FF00FF0，成功完成了前向传递，解决了 Load-Use 型数据冒险。

二、控制冒险

测试代码：

```
00000000 <_start>:
       0: 0040006f          jal   x0,4

00000004 <reset_vector>:
       4: 02500413          addi  x8,x0,37
       8: 01841413          slli  x8,x8,0x18
       c: fffff0b7          lui   x1,0xfffff
      10: 0080a023          sw x8,0(x1) # fffff000
      14: 27c00fef          jal   x31,290
      18: 00000013          addi  x0,x0,0
```

测试波形：



波形分析：

使用静态分支预测，总是预测不跳转。为了解决控制冒险，清除了后两条指令，即 flush 了 IF/ID 和 ID/EX 阶段的指令。当 pc=0 的 jal 指令移动到 EX 阶段时，由于是 jal 指令，控制信号 npc_op 被设置为 1，表示发生跳转。此时，冒险检测器检测到控制冒险的发生，清除了后两条指令，即 flush 了 IF/ID 和 ID/EX 阶段的指令。于是，在下一个周期中，pc_EX 和 pc_ID 被清零。此外，NPC 的 npc 也被修正为 4。因此，在下一个周期中，PC 的值被修正为 4。

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。

在设计和实现单周期和流水线 CPU 时，我遇到并解决了以下问题：

1.外设接口的实现。

在写入外围设备时，由于 sw 指令只有一个时钟周期，因此其信号也只有一个钟周期。在之前的设计中，数码管和 LED 的数据只显示一个周期，这对人眼来说太快了，所以数码管总是显示全零。后来的解决方案是使用时态逻辑来实现外围写入。几个寄存器用于外部存储需要显示的值。当每个时钟上升沿到达时，通过检查写入使能和写入地址来更新显示数据。也就是说，外围设备的读取使用组合逻辑来实现，而写入使用时间逻辑来实现。

2.时钟约束问题。

当第一次实现流水线 CPU 时，存在无法满足时钟要求的情况。这是因为 CPU 的逻辑电路太长，导致了太多的延迟。经过检查，发现问题在于解码模块的逻辑电路太长。发现问题后，我对一些逻辑进行了修改，将一些逻辑电路推进到提取阶段，并将一些逻辑回路推迟到执行阶段，以在不同阶段均匀分配线路延迟。最后满足了时钟要求。

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。

在参与了“计算机设计与实践”课程后，我对 Verilog 语言的运用和简单 CPU 的设计有了深入的了解。通过课程，我学会了如何使用 Verilog 语言来实现基于 miniRV 指令集的简单 CPU。在设计过程中，我掌握了 CPU 的基本原理和各个模块的功能。通过编写和调试代码，我能够独立完成 CPU 的设计和实现，并且能够解决在设计过程中遇到的问题。通过实践，我不仅提高了我的编程能力和逻辑思维能力，还加深了对计算机体系结构的理解。这门课程为我今后深入学习计算机体系结构和硬件设计打下了坚实的基础。