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

课程报告

开课学期：2025 夏季

课程名称：计算机设计与实践

项目名称：基于 miniLA 的 SoC 设计

项目类型：综合设计型

课程学时：48 地点：T2507

学生班级：计信 8 班

学生学号：2023311819

学生姓名：彭雯婷

评阅教师：

报告成绩：

实验与创新实践教育中心制

2025 年 6 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.w sub.w and or xor sll.w srl.w sra.w slt sltu slli.w srli.w srai.w addi.w andi ori xori slti sltui ld.b ld.bu ld.h ld.hu ld.w st.b st.h st.w lu12i.w pcaddu12i beq bne blt bltu bge bgeu jirl b bl

单周期 CPU 频率：25MHz

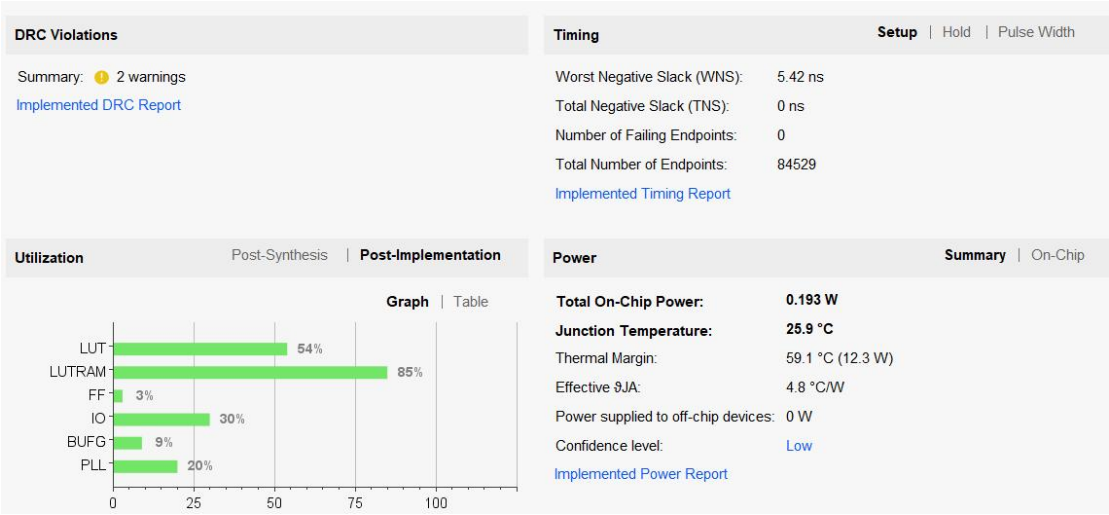
流水线 CPU 频率：100MHz

设计的主要特色（除基本要求以外的设计）

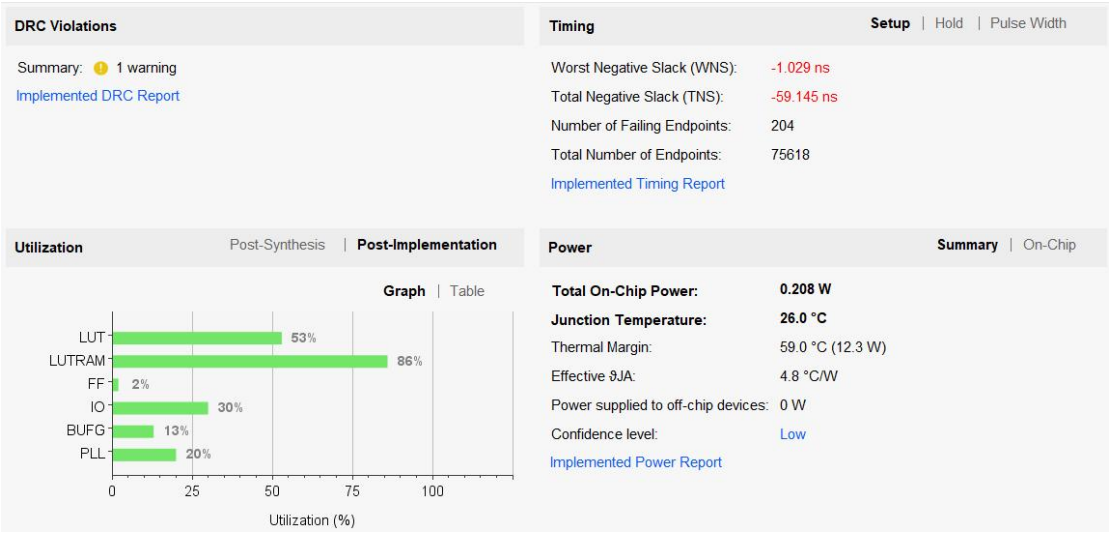
实现了所有选做指令，解决了控制冒险

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：



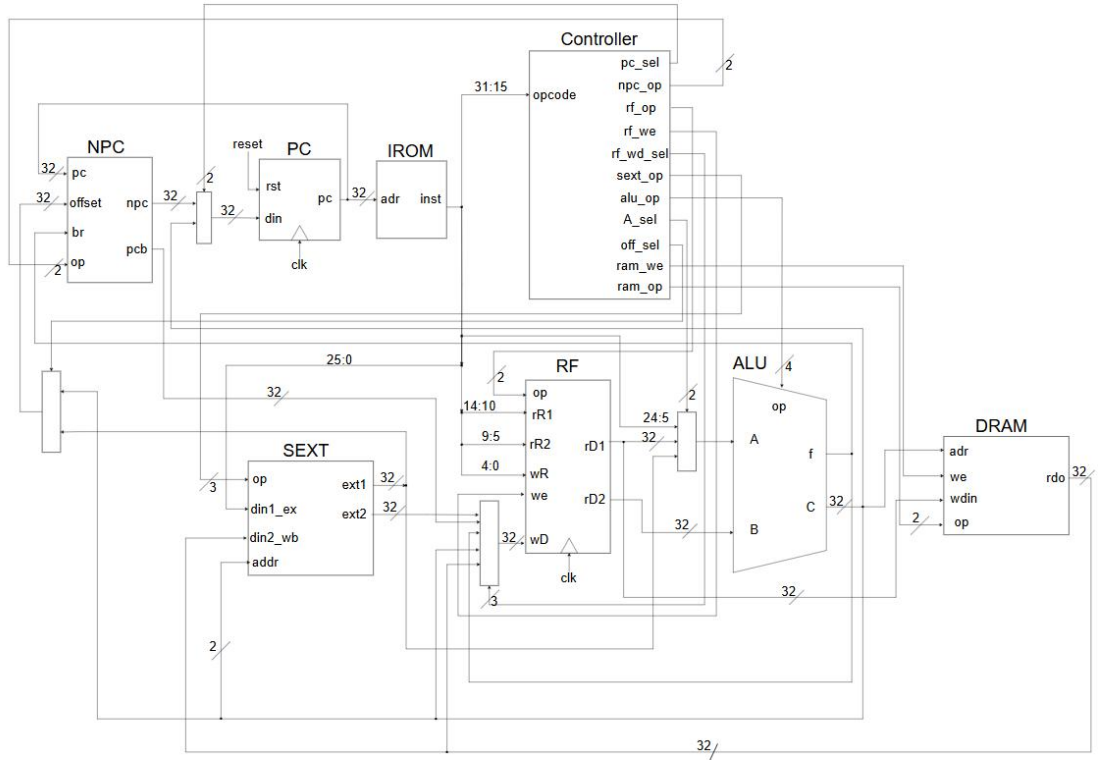
流水线：



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。



PC 模块：存储着当前指令的地址

NPC 模块：用于计算下一个 PC 值 npc，更新 PC 值。同时生成用于需要写回的和 PC 有关的值。比如，对于 pcaddu12i 指令，NPC 计算写回的数据 PC + (si20[31:12]<<12)

IROM 模块：用接收到的 PC 取出指令机器码

Controller 模块：根据当前指令机器码指令类型有关的数据段生成各种控制信号，指示其他模块的操作和控制数据选择器

SEXT 模块：数据拓展模块，完成对立即数的拓展和补全，对要求字节、半字写回数据结果进行整合

RF 模块：寄存器堆文件，存储 32 个寄存器的值。可以异步一次性读 2 个寄存器的值，可以同步写回一个寄存器。

ALU 模块：用于计算和比较操作，生成计算结果和比较结果

DRAM 模块：用地址从主存里异步读取数据和同步写数据

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。

PC 模块：

接口信号	rst	clk	din	pc
位宽	1	1	32	32
功能描述	复位信号	时钟信号	当前周期 PC 输入（大部分由 NPC 计算得到	输出当前周期的 PC

遇到复位信号，复位为 0。时钟上升沿时更新 PC 值。

```
wire [31:0] PC_din;
assign PC_din = (Ctrl_pc_sel)? ALU_C : NPC_npc;
```

特例：jirl 的下一周期 PC 即 rj + sext(imm, 2'b0)不通过 NPC 计算，由数据选择器控制，选择 ALU 计算的结果 ALU.C 得到）

NPC 模块：

接口信号	pc	op	offset	br	npc	pcb
位宽	32	2	32	1	32	32
功能描述	当前周期指令的 PC	操作选择信号	和 PC 做加法的操作数，相对当前 PC 的偏移量（大部分情况用于计算下一周期的 PC）	决定分支指令是否跳转的标志位	输出下一个周期的指令 PC	当前指令需要写回的数据（和 pc 有关

```
always @ (*) begin
    if(npc_op == `NPC_PC4)begin
        npc = pc + 32'h4;
        pcb = pc + 32'h4; //not used
    end
    else if(npc_op == `NPC_BRC)begin
        if(br) npc = pc + offset;
        else npc = pc + 32'h4;
        pcb = pc + 32'h4; //not used
    end
    else if(npc_op == `NPC_JMP)begin
        npc = pc + offset;
        pcb = pc + 32'h4;
    end
    else begin //npc_op == NPC_PC4_ADD (for pcaddu12i)
        npc = pc + 32'h4;
        pcb = pc + offset;
    end
```

共有 4 种操作：

1）加 4 型：针对 3R 型、2RI5 型、2RI2 型以及 lu12i.w 指令。这类指令下一周期的 PC 就是当前 PC+4。不会写回 NPC 生成的 pcb 信号，默认和 npc 值一样。

2）分支型：针对分支跳转指令。这类指令下一周期的 PC 要通过比较后得到的标志位来判断下一指令跳还是不跳到目标地址。同样不会写回 NPC 生成的 pcb 信号，默认设置为当前周期 PC+4。

3）直接跳转型：针对指令 b, bl, jirl。这类指令下一周期无需判断，

直接跳到目标地址。其中 bl 和 jirl 会写回 NPC 生成的 pcb 信号，值为当前周期 PC+4。
4）Pcaddu12i 指令：要写回的数据是 PC + (si20[31:12]<<12)，其中 offset 是来自 ALU 处理过的 (si20[31:12]<<12)

IROM 模块：

接口信号	addr	inst
位宽	32	32
功能描述	当前 PC	当前指令的机器码

生成 IP 核，在顶层模块实例化

Controller 模块：

接口信号	opcode	Pc_sel	Npc_op	Rf_op
位宽	32	1	2	2
功能描述	当前指令的机器码	选择下一周期 PC 是来自 NPC 的计算还是 ALU.C	控制 NPC 操作。共 4 种：PC+4 型、分支型、直接跳转型、pcaddu 型	控制 RF 操作。判断输出 rD1 选择取出 rR1 还是 wR 输入的寄存器值。 判断写回寄存器是采用 wR 还是指定 r1(bl 指令)
接口信号	Rf_we	Rf_wd_sel	Sext_op	Alu_op
位宽	1	3	3	4
功能描述	写使能信号,寄存器堆当前周期是否允许写	选择寄存器堆写回的数据来源: ALU.C ALU.f SEXT.ext2 DRAM.rdo NPC.pcb	控制符号拓展单元的操作。选择对访存阶段前的立即数做和主存输出做的操作，如符号拓展 5 位或者 12 位立即数，零拓展等	控制 ALU 的操作。选择对输入操作数做加、减、移位、逻辑运算、比较等操作
接口信号	A_sel	Off_sel	Ram_we	Ram_op
位宽	2	1	1	2
功能描述	选择 ALU 操作数 A 的数据来源: RF.rD1 SEXT.ext1 Inst[24:5]	选择 NPC 的输入 offset 的数据来源: ALU.C, SEXT.ext1	写使能信号，主存当前周期是否允许写	控制主存 DRAM 的操作。选择读出的数据是字、半字还是字节

根据 LA 指令机器码的特点，也把 inst 其中几部分拆成和 RV 指令类似的数字段来判断当前指令是哪一条。通过 OPCODE、SIG、FUNC3 和 FUNC7 字段来生成信号

```
wire [5:0] OPCODE;
assign OPCODE = inst[16:11];
wire SIG = inst[10];
wire [2:0] FUNC3 = inst[9:7];
wire [6:0] FUNC7 = inst[6:0];

//3R型
wire ADD_W = (OPCODE == 6'b000000) & (SIG == 1'b0) & (FUNC3 == 3'b000) & (FUNC7 == 7'b0100000);
wire SUB_W = (OPCODE == 6'b000000) & (SIG == 1'b0) & (FUNC3 == 3'b000) & (FUNC7 == 7'b0100010);
wire AND   = (OPCODE == 6'b000000) & (SIG == 1'b0) & (FUNC3 == 3'b000) & (FUNC7 == 7'b0101001);
```

通过如下方式编写，当某一信号有效时，生成对应的控制信号

```
assign pc_sel = JIRL ? 1 : 0;

wire npc_op_brc = BEQ | BNE | BLT | BLTU | BGE | BGEU;
wire npc_op_jmp = JIRL | B | BL;
wire npc_op_pc4_add = PCADDU;
wire npc_op_pc4 = !(npc_op_brc | npc_op_jmp | npc_op_pc4_add);
assign npc_op = {2{npc_op_pc4}} & `NPC_PC4
              | {2{npc_op_brc}} & `NPC_BRC
              | {2{npc_op_jmp}} & `NPC_JMP
              | {2{npc_op_pc4_add}} & `NPC_PC4_ADD;
```

SEXT 模块：

接口信号	op	din1_ex	din2_wb	addr	ext1	ext2
位宽	3	16	32	2	32	32
功能描述	控制符号拓展单元的操作。	执行或者计算 PC 阶段需要拓展的立即数	写回前需要拓展的数据	访存的地址低 2 位，用于加载字节和半字对数据的取用	din1_ex 拓展后的数据	din2_wb 拓展后的数据

用 case 语句选择进行的操作。

对于字节加载和半字加载，通过 addr 选择 32 位数据里要选取的位置，再对字节和半字进行符号拓展。

```
`SEXT_I12_b: begin
    //imm[21:10]
    ext1 = {{20{din1_ex[21]}}, din1_ex[21:10]};
    //DRAM.rdo[7:0]
    case(addr)
        2'b00:  ext2 = {{24{din2_wb[7]}}, din2_wb[7:0]};
        2'b01:  ext2 = {{24{din2_wb[15]}}, din2_wb[15:8]};
        2'b10:  ext2 = {{24{din2_wb[23]}}, din2_wb[23:16]};
        2'b11:  ext2 = {{24{din2_wb[31]}}, din2_wb[31:24]};
    endcase
end
```

RF 模块：

接口信号	clk	op	rR1	rR2
位宽	1	2	5	5
功能描述	时钟信号	控制 RF 操作	输入 rk	输入 rj
接口信号	wR	wD	rD1	rD2
位宽	5	32	32	32
功能描述	输入 rd	写回的数据	读出的寄存器值	读出的寄存器值

保证第 0 号寄存器存的值始终为 0。按 rf_op 的值选择不同的读取和写入操作：

```
always @(posedge clk) begin
    if (we) begin
        if(rf_op == `WR_1)  Regs[1] <= wD;
        else if(wR != 5'b00000)  Regs[wR] <= wD;
    end
end

assign rD1 = ((rR1 | wR) == 5'b0) ? 32'b0 : ((rf_op == `RD1_3R)? Regs[rR1] : Regs[wR]);
assign rD2 = (rR2 == 5'b0) ? 32'b0 : Regs[rR2];
```

ALU 模块：

接口信号	op	A	B	C	f
位宽	4	32	32	32	1
功能描述	控制 ALU 的操作。	操作数 1	操作数 2	操作结果	比较后得到的标志位

根据操作控制信号对操作数做不同的运算和比较，并赋值

```
// 计算标志位f
always @(*) begin
    case (alu_op)
        `OP_BEQ:    f = (B == A);
        `OP_BNE:    f = (B != A);
        `OP_BLT:    f = ($signed(B) < $signed(A));
        `OP_BLTU:   f = (B < A);
        `OP_BGE:    f = ($signed(B) >= $signed(A));
        `OP_BGEU:   f = (B >= A);
        default:   f = 1'b0;
    endcase
end
```

```
always @(*) begin
    case (alu_op)
        `OP_ADD:    C = A + B;
        `OP_SUB:    C = B + (~A) + 1;
        `OP_AND:    C = A & B;
        `OP_OR:     C = A | B;
        `OP_XOR:    C = A ^ B;
        `OP_SLL:    C = B << A[4:0];
        `OP_SRL:    C = B >> A[4:0];
        `OP_SRA:    C = $signed(B) >>> A[4:0];  // 算术右
        `OP_SLL_12: C = A << 4'hc;  // 固定移位12位
        default:   C = 32'b00000000;
    endcase
end
```

DRAM 模块：

接口信号	adr	we	wdin	op	rdo
位宽	32	1	32	2	32
功能描述	访存地址	写使能	写入的数据	控制主存 DRAM 的操作。选择读出的数据是字、半字还是字节	从主存读出的数据

生成 DRAM 的 IP 核，在顶层模块实例化。读写数据通过桥来实现连接。对读出数据的操作在 CPU 的模块文件中完成：

```
assign Bus_addr = ALU_C;
always @ (*) begin
    case (Ctrl_ram_op)
        `RAM_B: begin
            //wdin[7:0]
            case(Bus_addr[1:0])
                2'b00:  Bus_wdata = {Bus_rdata[31:8], RF_rD1[7:0]};
                2'b01:  Bus_wdata = {Bus_rdata[31:16], RF_rD1[7:0], Bus_rdata[7:0]};
                2'b10:  Bus_wdata = {Bus_rdata[31:24], RF_rD1[7:0], Bus_rdata[15:0]};
                2'b11:  Bus_wdata = {RF_rD1[7:0], Bus_rdata[23:0]};
            endcase
        end
```

1.3 单周期 CPU 仿真及结果分析

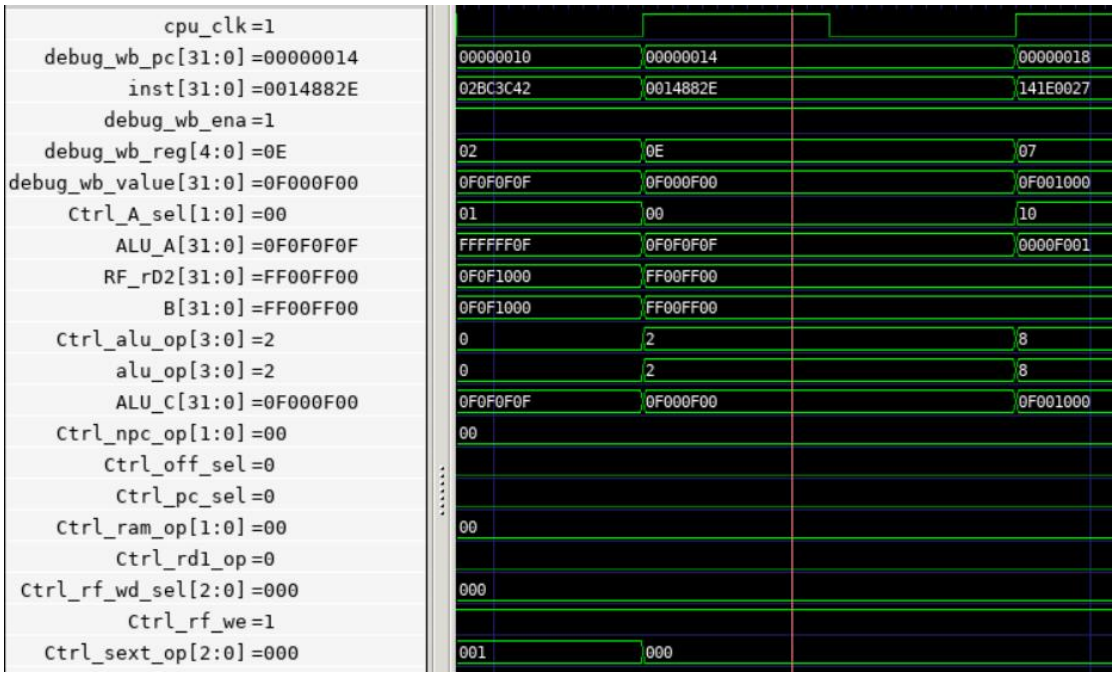
要求：包含**逻辑运算**、**访存**、**分支跳转**三类指令的仿真截图及波形分析；每类指令的截图和分析中，至少包含 1 条具体指令；截图需包含信号名和关键信号。

逻辑运算：

示例：and 指令

```
  34: 0283c042            addi.w  r2,r2,240 # f0f0f0f0 <_end
+0xf0f0d0f0>
  38: 0014882e            and     r14,r1,r2
  3c: 1401e007            lu12i.w r7,0xf00
```

仿真截图：



分析：

时钟周期上升沿来临，由上一个周期得到的 PC.din 输入，PC 更新为 00000014，输出给 IROM，IROM 取值出来得到机器码 inst（0x0014882E），机器码的高 17 位输入 Ctrl 模块，生成控制信号如下：

pc_sel = 0,　下一个周期的信号 PC 更新还是使用这个周期 NPC 的 npc 输出

npc_op = 0,　NPC 计算 npc = pc+4, pcb = pc+4

off_sel = 0,　and 指令的 NPC 用不到 offset，默认为 0

rf_op = 0, (trace 和原先设计文件里的信号名没统一，这个信号就是图上的 rd1_op)　and 指令属于 3R 型，使用 3 个寄存器。rD1 输出 rR1 指定的(rk)寄存器值，而不像分支跳转比较的是 rd 和 rj，要使用 wR 端口输入的寄存器号读数。写回寄存器也并非 bl 指令那样特殊使用 r1，要写回的寄存器号即输入的 wR。

rf_we = 1,　and 指令需要写回寄存器

rf_wd_sel = 0, 写回的数据来自 ALU.C

sext_op = 0,　但是 and 指令不需要拓展立即数，默认为 0

alu_op = 4'h2,　对应按位与操作

A_sel = 0,　ALU 操作数 A 来自 RF 的输出 rD1

ram_we = 0,　ram_op = 0　and 指令不需要访问 DRAM，写使能无效，ram_op 默认为 0

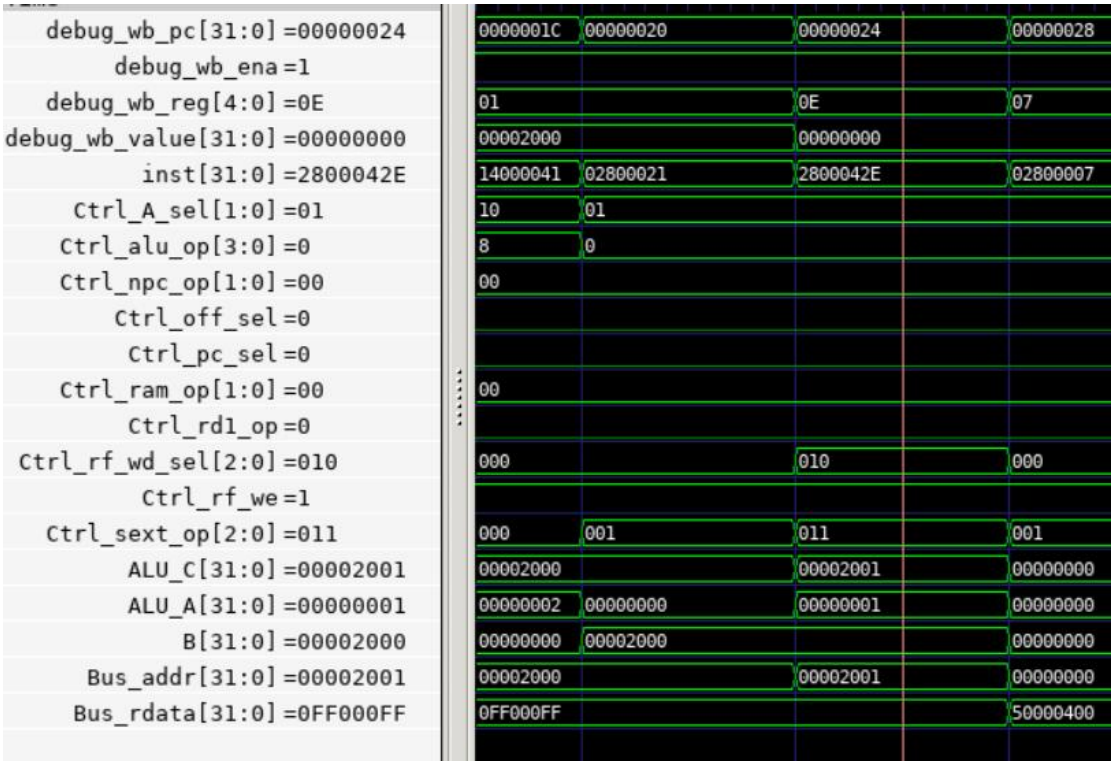
生成控制多路选择器和多功能模块的信号之后，依据 inst 的低 15 位得到会用到

的 3 个寄存器号（对于这条具体指令来说，rR1 = 1, rR2 = 2），从 RF 中取出寄存器值 rD1, rD2，并指定写回的寄存器 14。
ALU 操作数 A 由 A_sel 选择来自 rD1、操作数 B 始终来自 rD2，ALU 由 alu_op 选择计算 A&B，输出计算结果 C，rf_wd_sel 选择写回数据来自 ALU.C，输送回 RF 写回寄存器 r14。

访存：
示例：ld.b 指令

```
0000001c <test_3>:
  1c: 14000041            lu12i.w r1,0x2
  20: 02800021            addi.w  r1,r1,0 # 2000 <begin_signature>
  24: 2800042e            ld.b    r14,r1,1
```

仿真截图：



分析：
时钟周期上升沿来临，由上一个周期得到的 PC.din 输入，PC 更新为 00000024，输出给 IROM，IROM 取值出来得到机器码 inst（0x2800042E），机器码的高 17 位输入 Ctrl 模块，生成控制信号如下：
pc_sel = 0,　下一个周期的信号 PC 更新还是使用这个周期 NPC 的 npc 输出
npc_op = 0,　NPC 计算 npc = pc+4, pcb = pc+4
off_sel = 0,　ld.b 指令的 NPC 用不到 offset，默认为 0
rf_op = 0, (trace 和原先设计文件里的信号名没统一，这个信号就是图上的 rd1_op)　ld.b 指令只用到了 2 个寄存器。只读一个寄存器 r1（rj）, rD2 输出 rR2 指定的寄存器值。写回寄存器为 wR 输入指定的 r14。
rf_we = 1,　ld.b 指令需要写回寄存器
rf_wd_sel = 3'b010,　写回的数据来自字节处理过后的 SEXT.ext2

sext_op = 3'b011,　ld.b 指令需要拓展用于 ALU 运算的立即数，ld.b 需要拓展 12 位立即数。并且对需要写回的字节数据再次拓展为 32 位。
alu_op = 4'h0,　对应加法操作
A_sel = 1,　ALU 操作数 A 来自 SEXT 的输出 ext1
ram_we = 0,　ram_op = 0　ld.b 指令只读 DRAM。写使能无效，ram_op 为 0

生成控制多路选择器和多功能模块的信号之后，依据 inst 的低 10 位得到会用到的 2 个寄存器号(对于这条 ld.b 指令来说，rR2 = 1)，从 RF 中取出寄存器值 rD2，并指定写回的寄存器 14。
SEXT 符号拓展单元负责将立即数 1（12 位）根据操作控制信号从机器码段中截取出，并且拓展为 32 位。
ALU 操作数 A 由 A_sel 选择来自 SEXT.ext1、操作数 B 始终来自 rD2，ALU 由 alu_op 选择计算 A+B，输出计算结果 C。
ld.b 指令需要访问主存，访问地址始终来自 ALU.C。从主存读出数据后，将该 32 位数据输送给 SEXT 单元做字节加载处理，根据访问地址 0x2001 的低二位来选择加载的字节是取出的 32 位数据的（从 0 开始）低 9 至 16 位。最后写回选择器根据选择信号 rf_wd_sel = 010 选择 EXT.ext2，写回 r14。

分支跳转：
示例：bne 指令。选取 PC 为 0x30 的 bne 指令进行分析，bne 判断正确，要跳转至 0x3c。

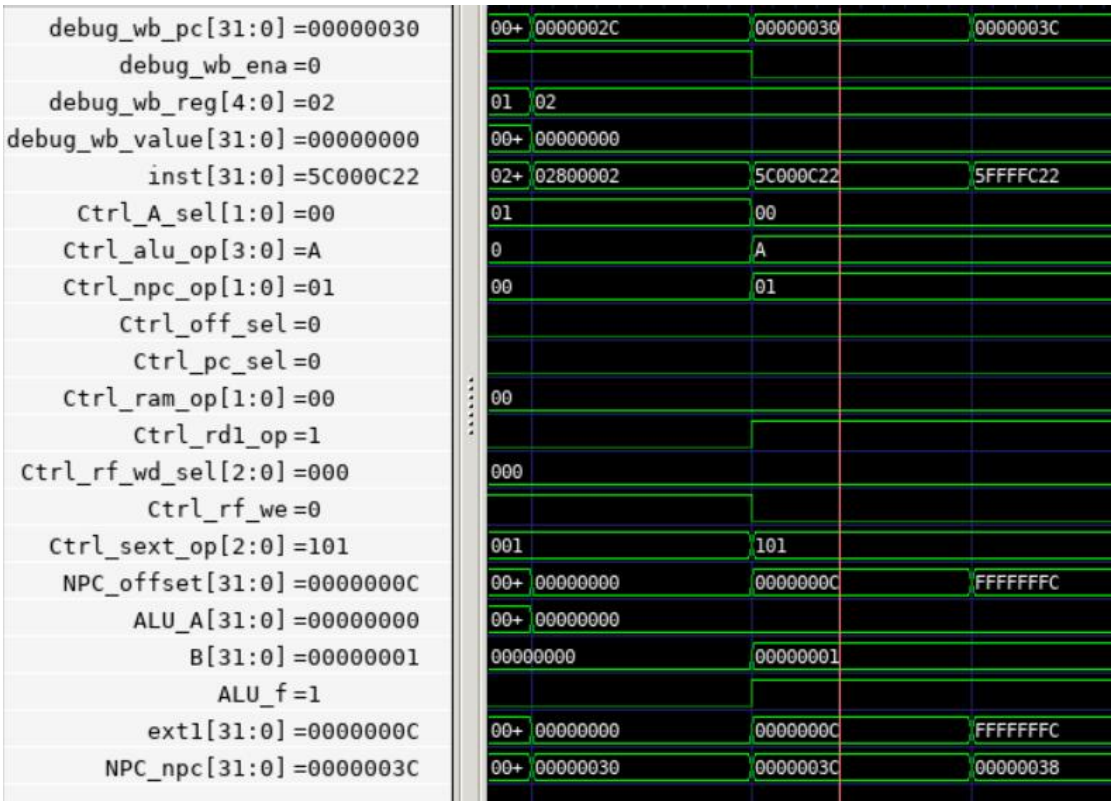
```
00000024 <test_3>:
  24: 02800c03          addi.w  r3,r0,3
  28: 02800401          addi.w  r1,r0,1
  2c: 02800002          addi.w  r2,r0,0
  30: 5c000c22          bne r1,r2,0x3c <test_3+0x18>
  34: 5c029403          bne r0,r3,0x2c8 <fail>
```

分析：
时钟周期上升沿来临，由上一个周期得到的 PC.din 输入，PC 更新为 00000030，输出给 IROM，IROM 取值出来得到机器码 inst（0x5C000C22），机器码的高 17 位输入 Ctrl 模块，生成控制信号如下：
pc_sel = 0,　下一个周期的信号 PC 更新还是使用这个周期 NPC 的 npc 输出
npc_op = 1,　NPC 根据 br 标志位选择输出下一周期跳转或者不跳转的 PC 值
off_sel = 0,　bne 指令的 NPC 的 offset 信号来自 SEXT.ext1
rf_op = 1, (trace 和原先设计文件里的信号名没统一，这个信号就是图上的 rd1_op)　bne 指令使用 2 个寄存器。但是 rD1 要输出 wR 指定的(rd)寄存器值，因为分支跳转比较的是 rd 和 rj 的值，要使用 wR 端口输入的寄存器号读数。写回寄存器号即输入的 wR。
rf_we = 0,　bne 指令不需要写回寄存器
rf_wd_sel = 0,　bne 指令不需要写回寄存器，默认为 0
sext_op = 3'b101,　bne 指令需要拓展立即数，将立即数（偏移量）低 2 位 0 补全再做符号拓展
alu_op = 4'hA,　对应不等比较操作

A_sel = 0,　　ALU 操作数 A 来自 RF 的输出 rD1
ram_we = 0,　ram_op = 0　bne 指令不需要访问 DRAM，写使能无效，ram_op 默认为 0。

生成控制多路选择器和多功能模块的信号之后，依据 inst 的低 10 位得到会用到的 2 个寄存器号（对于这条 bne 指令来说，wR = 1, rR2 = 2），从 RF 中取出寄存器值 rD1, rD2。
ALU 操作数 A 由 A_sel 选择来自 rD1、操作数 B 始终来自 rD2，ALU 由 alu_op 选择生成标志位 ALU.f = 1（因为 rD1 的值为 1，rD2 的值为 0，两者确实不相等），输出标志位，输送给 NPC 模块。
与此同时，inst 的低 11 位到 26 位（从 1 开始）输送给 SEXT.ext1，补全低 2 位 0 并且符号拓展后，输送给 NPC 模块。
NPC 计算下一周期跳转地址 0x3c。

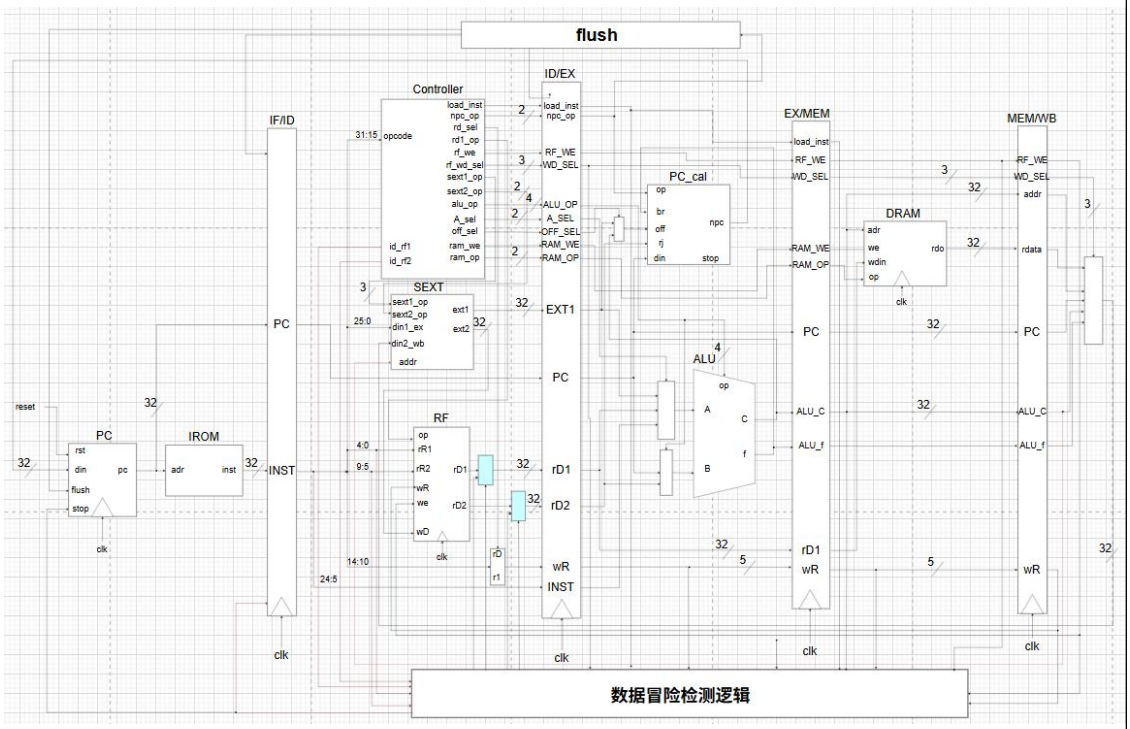
仿真截图：



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。



数据冒险检测模块：检测冒险的存在，输出冒险的类型。根据 RAW 冒险类型进行数据前递。如果检测到加载-使用型冒险，生成 stop 信号插入一条空指令——暂停 PC 更新和 IF/ID 流水线寄存器，清空 ID_EX 流水线寄存器（防止下一条指令还是 load，出现新的加载-使用型冒险）

flush 模块：检测到当前指令 PC 更新逻辑不一定是 PC+4 时，生成 flush 信号，以便清空 IF/ID 流水线寄存器和 ID_EX 流水线寄存器，插入两个气泡，等待当前指令进行到执行阶段，计算好下一个 PC 值。

IF/ID 模块、ID/EX 模块、EX/MEM 模块、MEM/WB 模块：均为流水线寄存器，分隔流水线各个阶段。存储着上一阶段的值，为每个阶段提供前几个阶段执行后得到的值，以便当前阶段的进行，防止数据混乱。

PC 模块：存储着取值阶段指令的地址

IROM 模块：用接收到的 PC 取出指令机器码

Controller 模块：根据译码阶段的指令机器码指令类型有关的数据段生成各种控制信号，指示其他模块的操作和控制数据选择器。其中 npc_op 信号还作为 flush 信号生成的依据。同时 Controller 模块还会生成寄存器是否读取数据的信号，以及指示当前指令是否是加载指令的信号，作为数据冒险检测的依据。

SEXT 模块：数据拓展模块，完成对执行阶段需要用到的立即数的拓展和补全，以及对要求字节、半字写回的数据结果进行整合
RF 模块：寄存器堆文件，存储 32 个寄存器的值。可以异步一次性读 2 个寄存器的值，可以同步写回一个寄存器。
ALU 模块：用于计算和比较操作，生成计算结果和比较结果
PC_cal 模块：用于计算 npc 非 PC+4 类型指令的下一个 PC 值，用于更新。
DRAM 模块：用地址从主存里异步读取数据和同步写数据

流水线分为五级：取值、译码、执行、访存、写回
取值：更新 PC，根据 PC 取出即将进入流水线后续阶段的指令机器码
译码：根据 IF/ID 流水线寄存器的机器码，生成各种控制信号，拓展立即数，取出寄存器的值
执行：ALU 执行计算、比较等操作，同时如果有控制冒险，PC_cal 生成跳转的目标地址转发给阻塞暂停了两个周期的 PC
访存：访问主存，按指令要求读写数据
写回：选择写回数据源，如果有字节或者是半字加载，将要写回的数据进行处理，再写回

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，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 模块：

接口信号	rst	clk	stop	flush	din	pc
位宽	1	1	1	1	32	32
功能描述	复位信号	时钟信号	暂停信号	是否使用 din 的信号	PC_cal 计算得到的 PC	输出当前阶段的 PC

默认 pc 更新为 pc+4。如果接收到 stop 暂停信号，pc 保持不动。如果接收到 flush 信号，说明检测到控制冒险，插入了两个周期的空指令，pc 更新为执行阶段计算好的跳转地址

```
    always @ (posedge clk or posedge rst) begin
        if (rst) begin
//            pc <= 32'hfffffffC;      //-4
            pc <= 0;
        end else if(stop)begin
            pc <= pc;
        end else if(flush)begin
            pc <= din;
        end else  pc <= pc + 4;
    end
```

PC_cal 模块：

接口信号	clk	din	op	offset	br	rj	npc
位宽	1	32	2	32	1	32	32
功能描述	时钟信号	当前执行阶段的指令 PC	操作选择信号	相对当前执行阶段 PC 的偏移量	决定分支指令是否跳转的标志位	jirl 指令跳转地址计算会用到的操作数	下一个 PC 值

根据 npc_op 信号选择操作，计算可能会用到的 npc

```
else if(npc_op == `NPC_BRC)begin
    if(br) npc = pc + offset;
    else npc = pc + 32'h4;
end
else if(npc_op == `NPC_OFF)begin
    npc = pc + offset;
end
else begin //for JIRL)
    npc = rj + offset;
end
```

冒险检测模块（数据冒险和控制冒险一起检测）：

接口信号	Cpu_clk	Cpu_rst	ID_EX_wR	ID_EX_rf_we
位宽	1	1	5	1
功能描述	时钟信号	复位信号	当前时钟周期执行阶段的指令要写回的寄存器	当前时钟周期执行阶段的指令是否要写回（写使能
接口信号	EX_MEM_wR	EX_MEM_rf_we	MEM_WB_wR	MEM_WB_rf_we
位宽	5	1	5	1
功能描述	当前时钟周期访存阶段的指令要写回的寄存器	当前时钟周期访存阶段的指令是否要写回（写使能	当前时钟周期写回阶段的指令要写回的寄存器	当前时钟周期写回阶段的指令是否要写回（写使能
接口信号	ID_rs1	ID_rs2	Ctrl_id_rf1	Ctrl_id_rf2
位宽	5	5	1	1
功能	当前时钟周期	当前时钟周期	当前时钟周期	当前时钟周期

描述	译码阶段的指令要读的寄存器 1	译码阶段的指令要读的寄存器 2	译码阶段的指令是否需要读寄存器 1	译码阶段的指令是否要读寄存器 2
接口信号	ID_EX_load	ID_EX_npc_op	Rs1_id_ex_hazard	Rs2_id_ex_hazard
位宽	1	2	1	1
功能描述	指示当前时钟周期执行阶段的指令是否是 load 指令	指示当前时钟周期执行阶段的指令是否是需要 PC_cal 计算 NPC 的指令	当前周期是否出现 RAW 情形 A 的冒险：取指阶段指令要读的 rs1 和执行阶段指令后续要写回的寄存器相同	当前周期是否出现 RAW 情形 A 的冒险：取指阶段指令要读的 rs2 执行阶段指令后续要写回的寄存器相同
接口信号	rs1_id_mem_hazard	rs2_id_mem_hazard	rs1_id_wb_hazard	rs2_id_wb_hazard
位宽	1	1	1	1
功能描述	当前周期是否出现 RAW 情形 B 的冒险：取指阶段指令要读的 rs1 和访存阶段指令后续要写回的寄存器相同	当前周期是否出现 RAW 情形 B 的冒险：取指阶段指令要读的 rs2 和执行阶段指令后续要写回的寄存器相同	当前周期是否出现 RAW 情形 C 的冒险：取指阶段指令要读的 rs1 和执行阶段指令后续要写回的寄存器相同	当前周期是否出现 RAW 情形 C 的冒险：取指阶段指令要读的 rs2 和执行阶段指令后续要写回的寄存器相同
接口信号	pipeline_stop	flush_if_id	flush_id_ex	/
位宽	1	1	1	/
功能描述	出现了加载-使用型冒险的标志，作为信号暂停或清空流水线寄存器	出现了控制冒险的标志，作为信号清空 IF/ID 流水线寄存器	出现了控制冒险的标志，作为信号清空 ID/EX 流水线寄存器	/

判断 RAW 冒险：当前是否真的需要读 rs1/rs2，rs1/rs2 是否和前几条（1 至 3）指令的写回寄存器相同，写回寄存器不是 r0，前几条（1 至 3）指令需要写回。四个条件同时满足才说明出现了冒险。

```
//RAW冒险情形A：
assign rs1_id_ex_hazard = (ID_EX_wR != 5'b0) & (ID_EX_wR == ID_rs1) & ID_EX_rf_we & Ctrl_id_rf1;
assign rs2_id_ex_hazard = (ID_EX_wR != 5'b0) & (ID_EX_wR == ID_rs2) & ID_EX_rf_we & Ctrl_id_rf2;
```

普通 RAW 冒险使用前递解决。

加载-使用型冒险无法单靠前递数据解决。需要插入一条空指令，使加载指令到达访存阶段取出数据后再前递。

故要特别判断是否是加载-使用型冒险：

检测到 RAW 冒险情形 A 且执行阶段的指令是 load 指令，pipeline_stop 信号有效。

```
// 必须暂停的Load-Use冒险
wire rs1_load_use;
wire rs2_load_use;
assign rs1_load_use = rs1_id_ex_hazard & ID_EX_load;
assign rs2_load_use = rs2_id_ex_hazard & ID_EX_load;
assign pipeline_stop = (rs1_load_use | rs2_load_use);

assign flush_if_id = (ID_EX_npc_op != `NPC_PC4);
assign flush_id_ex = flush_if_id;
```

任何下一条地址不是当前 pc+4 而是需要计算的指令，都会生成两个寄存器清空信号，插入两条空指令，等待当前指令到执行阶段，从而计算出下一条地址。

前递模块：

接口信号	Cpu_clk	Cpu_rst	ID_EX_rd1 ID_EX_rd2	RF_rD1 RF_rD2
位宽	1	1	32	32
功能描述	时钟信号	复位信号	当前时钟周期执行阶段指令需要从 RF 读出的寄存器值	当前时钟周期译码阶段指令已经从 RF 读出的寄存器值（未前递）
接口信号	Rs1_id_ex_hazard rs1_id_mem_hazard rs1_id_wb_hazard	Rs2_id_ex_hazard rs2_id_mem_hazard rs2_id_wb_hazard	ALU_C EX_MEM_alu_c MEM_WB_alu_c	ALU_f EX_MEM_alu_f MEM_WB_alu_f
位宽	1	1	32	1
功能描述	当前 rs1 是否需要从执行/访存/写回阶段前递数据	当前 rs2 是否需要执行/访存/写回阶段前递数据	执行/访存/写回阶段指令的 ALU 计算结果，用于前递	执行/访存/写回阶段指令的 ALU 生成的标志位，用于写回比较结果的前递
接口信号	ID_EX_pc EX_MEM_pc MEM_WB_pc	Bus_addr MEM_WB_raddr	EX_MEM_sext2_op MEM_WB_sext2_op	ID_EX_rf_wd_sel EX_MEM_rf_wd_sel MEM_WB_rf_wd_sel
位宽	32	32	2	3

功能描述	执行/访存/写回阶段的 pc，用于写回 pc+4 的前递	访存/写回阶段指令的访存地址，用于半字或是字节加载数据的整合	访存/写回阶段指令的符号拓展控制信号，用于半字或是字节加载数据的整合	执行/访存/写回阶段指令的写回数据选择信号
接口信号	Bus_rdata MEM_WB_rdata	/	/	/
位宽	32			
功能描述	访存/写回阶段指令的访存读取数据，用于前递			

根据冒险情况的不同，分情况前递。Id_ex 冒险不涉及访存数据，加载-使用型冒险会使用插入空指令的方法使其后续变为 id_mem 冒险。前递时，根据前递数据来源指令的写回选择信号来选择前递要写回的数据。

```
always @(posedge cpu_clk or posedge cpu_rst) begin
    if(cpu_rst)begin
        ID_EX_rd1       <= 0;
    end
    else if(rs1_id_ex_hazard)begin
        if(ID_EX_rf_wd_sel == `WD_C)       ID_EX_rd1 <= ALU_C;
        else if(ID_EX_rf_wd_sel == `WD_f)  ID_EX_rd1 <= {31'h00000000, ALU_f};
        else if(ID_EX_rf_wd_sel == `WD_PCB) ID_EX_rd1 <= ID_EX_pc + 4;

    end
    else if(rs1_id_mem_hazard)begin
        if(EX_MEM_rf_wd_sel == `WD_C)       ID_EX_rd1 <= EX_MEM_alu_c;
        else if(EX_MEM_rf_wd_sel == `WD_f)  ID_EX_rd1 <= {31'h00000000, EX_MEM_alu_f};
        else if(EX_MEM_rf_wd_sel == `WD_PCB) ID_EX_rd1 <= EX_MEM_pc + 4;
        else if(EX_MEM_rf_wd_sel == `WD_RDO) ID_EX_rd1 <= Bus_rdata;
        else if(EX_MEM_rf_wd_sel == `WD_RDOB)begin
            case(Bus_addr[1:0])
                2'b00:  ID_EX_rd1 <= {24'b0, Bus_rdata[7:0]};
                2'b01:  ID_EX_rd1 <= {24'b0, Bus_rdata[15:8]};
                2'b10:  ID_EX_rd1 <= {24'b0, Bus_rdata[23:16]};
                2'b11:  ID_EX_rd1 <= {24'b0, Bus_rdata[31:24]};
            endcase
        end
```

4 个流水线寄存器模块：

接口信号：一定有时钟信号和复位信号。对于每个阶段需要保存到下一阶段的数据，流水线寄存器的接口信号会有这个阶段的输入和当前流水线寄存器的输出。此外，ID/EX 和 IF/ID 流水线寄存器还有 pipeline_stop 信号和 flush 信号输入，

分别用于解决载入-使用型冒险和控制冒险。

```
always @(posedge cpu_clk or posedge cpu_rst) begin
    if (cpu_rst) begin
        IF_ID_pc   <= 32'h0;
        IF_ID_inst <= 32'h0;
    end else if(flush_if_id)begin
        IF_ID_pc   <= 32'h0;
        IF_ID_inst <= 32'h0;
    end else if (pipeline_stop) begin
        IF_ID_pc   <= IF_ID_pc;
        IF_ID_inst <= IF_ID_inst;
    end else begin
        IF_ID_pc   <= PC_pc;
        IF_ID_inst <= inst;
    end
end
```

```
else if(flush_id_ex || pipeline_stop)begin
    ID_EX_rf_we     <= 0;
    ID_EX_rf_wd_sel <= 0;
    ID_EX_alu_op    <= 0;
    ID_EX_A_sel     <= 0;
    ID_EX_off_sel   <= 0;
    ID_EX_ram_we    <= 0;
    ID_EX_ram_op    <= 0;
    ID_EX_load      <= 0;
    ID_EX_ext1      <= 0;
    ID_EX_pc        <= 0;
    ID_EX_wR        <= 0;
    ID_EX_inst      <= 0;
```

左图为 IF/ID 流水线寄存器的所有更新逻辑，右图只截取了 ID/EX 流水线寄存器受 flush 和 stop 信号控制的部分逻辑。

stop 信号暂停 IF/ID 流水线寄存器，清空 ID/EX 流水线寄存器。flush 信号清空 IF/ID 流水线寄存器和 ID/EX 流水线寄存器。复位信号将四个流水线寄存器清空。

Ctrl 模块的微调：

增加了说明当前执行阶段是 load 指令的信号；增加了译码阶段是否读寄存器的信号；sext_op 变为 sext1_op 和 sext2_op；rf_op 拆分为 rd1_op 和 rd_sel；删去了 pc_sel 信号。

SEXT 模块的微调：

接口信号变化：sext_op 变为 sext1_op 和 sext2_op。考虑到流水线通过寄存器传递信号，两个输出使用的阶段不用，控制信号变为两个信号分别指示 ext1 和 ext2 的生成。

RF 模块的微调：

接口信号变化：rf_op 变为 rd1_op。bl 写回寄存器指定 r1 放在多路选择器完成。

ALU 模块的微调：

接口信号不变，但是在流水线中对 pcaddu12i 的指令的数据通路有所变更。ALU 操作增加一项：

```
always @(*) begin
    case (alu_op)
        `OP_ADD:    C = A + B;
        `OP_SUB:    C = B + (~A) + 1;
        `OP_AND:    C = A & B;
        `OP_OR:     C = A | B;
        `OP_XOR:    C = A ^ B;
        `OP_SLL:    C = B << A[4:0];
        `OP_SRL:    C = B >> A[4:0];
        `OP_SRA:    C = $signed(B) >>> A[4:0];  // 算术右移
        `OP_SLL_12: C = A << 4'hc;  // 固定移位12位
        `OP_PCADDU: C = (A << 4'hc) + B;  // 固定移位12位
        default:   C = 32'b00000000;
    endcase
end
```

2.3 流水线 CPU 仿真及结果分析

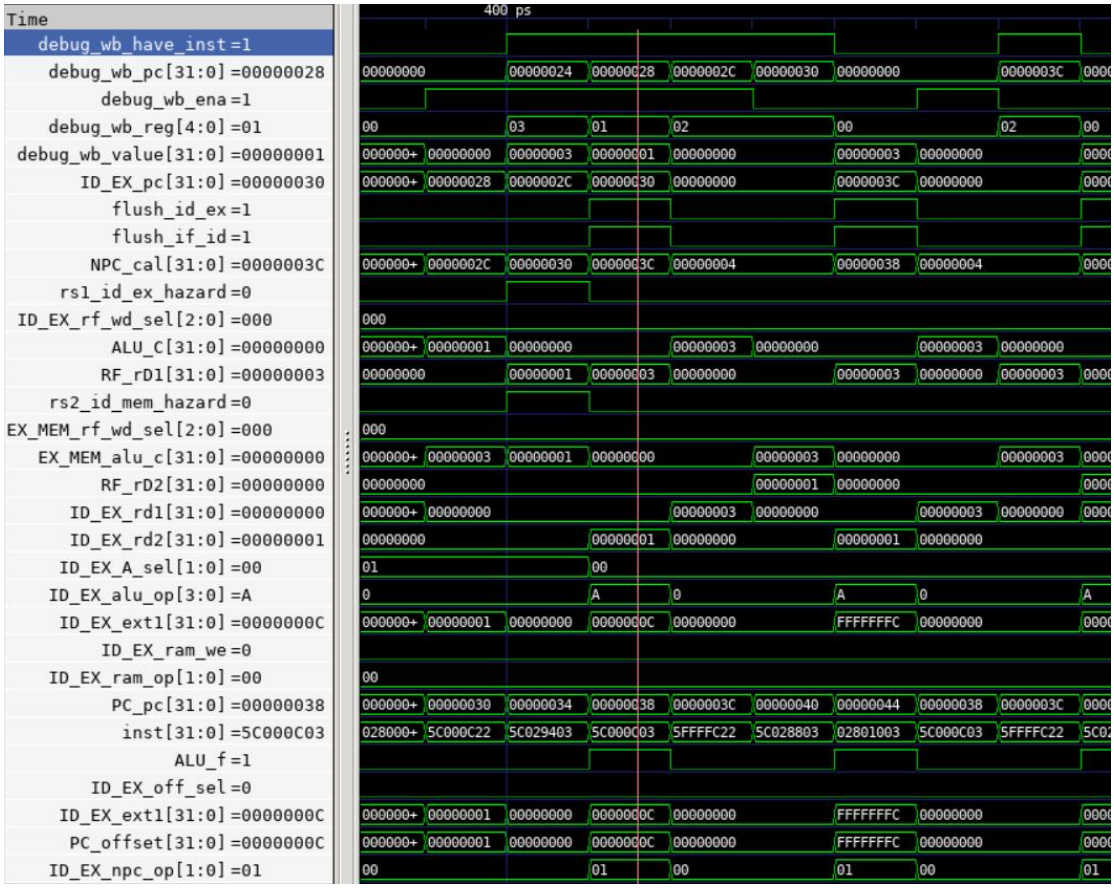
要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。若仅实现了理想流水，则此处贴上理想流水的仿真截图及详细的波形分析。

控制冒险和数据冒险情形 A、B：

示例：bne 指令。选取 PC 为 0x30 的 bne 指令进行分析。bne 要比对的两个寄存器 r1 和 r2 和前两条指令均有相关性。bne 判断正确，要跳转至 0x3c。

```
00000024 <test_3>:
   24: 02800c03            addi.w  r3,r0,3
   28: 02800401            addi.w  r1,r0,1
   2c: 02800002            addi.w  r2,r0,0
   30: 5c000c22            bne r1,r2,0x3c <test_3+0x18>
   34: 5c029403            bne r0,r3,0x2c8 <fail>
```

仿真截图：



分析：

Debug_wb_pc = 0x24 时，addi.w r3, r0, 3 这条指令正在写回。bne 指令在时钟上升沿结束取值阶段，PC 更新，IROM 取出指令机器码 0x5C000C22，传给了 IF/ID 寄存器。bne 正在译码阶段，生成各种控制信号，并且将跳转地址偏移量做符号拓展生成 SEXT.ext1。在这一阶段，读取寄存器数据 RF_rD1 和 RF_rD2。冒险检测模块检测到了 rs1（r2）有情形 A 的数据冒险，rs2（r1）有情形 B 的数据冒险，准备前递。

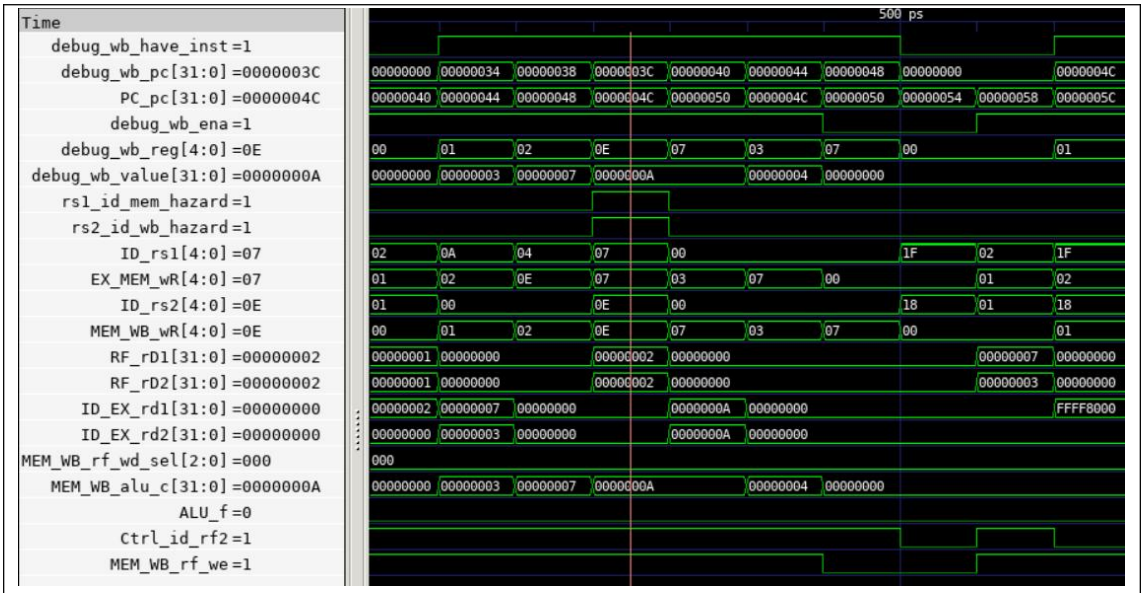
Debug_wb_pc = 0x28 时，bne 指令已经到达了执行阶段，上一阶段译码得到的

所需值通过 ID_EX 寄存器保存。此时可以看到上一阶段取出的 RF_rD1 和 RF_rD2 数据都是滞后的，不是 bne 要用来比对的数据。根据 ID_EX_rf_wd_sel 和 EX_MEM_rf_wd_sel 前递得到正确的寄存器值 RF_rD1 和 RF_rD2。由 ID_EX_a_sel = 0，执行阶段 ALU 操作数 A 来自 RF_rD1。ALU 根据 ID_EX_alu_op = 4'hA，进行不等比较操作，生成标志位 ALU_f = 1。
又由冒险检测模块检测到控制冒险。
生成 flush_if_id 和 flush_id_ex 信号，插入两个空指令。可以看到 Debug_wb_pc = 0x30 那个周期后是两个周期的无效指令。
与此同时，flush 指令输送给 PC 模块，PC 在两条空指令后选择 PC_cal 的值作为输入来更新 PC 地址。NPC_cal 根据 ALU_f、ID_EX_npc_op 和 ID_EX_off_sel 计算得到下一周期跳转地址 0x3c。
bne 指令无访存和写回阶段。待 bne 指令结束两个周期后成功跳转 0x3c。

数据冒险情形 C：
示例：add.w 指令。PC 为 0x3c 的 add.w 指令和 PC 为 0x48 的 bne 指令有数据相关性，是情形 C 的数据冒险。同时 PC 为 0x40 的 addi.w 指令和 PC 为 0x48 的 bne 指令也有数据相关性，是情形 B 的数据冒险，由于上一个分析中有涵盖，这个示例中主要分析情形 C 的数据冒险的前递。

```
00000034 <test_4>:
  34: 02800c01            addi.w  r1,r0,3
  38: 02801c02            addi.w  r2,r0,7
  3c: 0010082e            add.w   r14,r1,r2
  40: 02802807            addi.w  r7,r0,10
  44: 02801003            addi.w  r3,r0,4
  48: 5c049dc7            bne r14,r7,0x4e4 <fail>
```

分析：
Debug_wb_pc = 0x3C 时，add.w r14, r1, r2 这条指令正在写回，寄存器 r14 的值还没有更新。而此时 0x48 的 bne 指令正在译码阶段。MEM_WB_wR = 0xE = ID_rs2，且 MEM_WB_we 和 Ctrl_id_rf2 均有效。冒险检测模块检测到了 rs2（r14）有情形 C 的数据冒险，准备前递。
Debug_wb_pc = 0x40 时，bne 指令已经到达了执行阶段。此时可以看到上一阶段取出的 RF_rD1 和 RF_rD2 数据都是滞后的，不是 bne 要用来比对的数据。根据 MEM_WB_rf_wd_sel 和 EX_MEM_rf_wd_sel 前递得到正确的寄存器值 RF_rD1 和 RF_rD2。由 ID_EX_a_sel = 0，执行阶段 ALU 操作数 A 来自 RF_rD1。ALU 根据 ID_EX_alu_op = 4'hA，进行不等比较操作，生成标志位 ALU_f =0。
仿真截图如下：


Time
debug_wb_have_inst =1
debug_wb_pc[31:0] =0000003C
PC_pc[31:0] =0000004C
debug_wb_ena =1
debug_wb_reg[4:0] =0E
debug_wb_value[31:0] =0000000A
rs1_id_mem_hazard =1
rs2_id_wb_hazard =1
ID_rs1[4:0] =07
EX_MEM_wR[4:0] =07
ID_rs2[4:0] =0E
MEM_WB_wR[4:0] =0E
RF_rD1[31:0] =00000002
RF_rD2[31:0] =00000002
ID_EX_rd1[31:0] =00000000
ID_EX_rd2[31:0] =00000000
MEM_WB_rf_wd_sel[2:0] =000
MEM_WB_alu_c[31:0] =0000000A
ALU_f =0
Ctrl_id_rf2 =1
MEM_WB_rf_we =1
500 ps

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。

遇到的问题：

问题一：单周期我发现下板后最后得数码管显示排好序得 8 个数字并不会出现，而是从头开始显示计数器的值。

解决方法：经排查和思考后，找到问题是原先我的汇编程序没有 end 结束的标识或是指令，而 logisim 跑电路仿真时遇到最后一条指令并不会再接着执行下去。但下板和仿真不同，只显示一个时钟周期很显然人眼看不出来。于是我在汇编程序最后一行加了 b 指令，强制最后循环执行显示排好序后的数字。于是我明白仿真并不能代替现实的下板验证，有的问题在现实必须考虑到。

问题二：流水线我发现下板后按拨码开关并没有用，数码管不会现实计数器的值，始终为 8 个 0，而 led[0]始终亮着。

解决方法：下班前我的 trace 已经全部正确通过，修改汇编程序下板发现理想情况下我的 CPU 能够正常运行。于是我也想过是否是我的冒险处理还有问题。但使用重排和插入空指令解决控制冒险下板发现仍然没用。最后是写了一个仿真文件，想通过仿真波形查看是哪个信号或者哪个阶段出问题了。在仿真过程中，注意到一个 critical warning，上网查询，发现原来是因为我的 ID_IF 流水线寄存器模块 always 块里 if-else 语句判断条件中异步复位信号和同步清空信号写在一起了，导致混乱。修改完这个问题后，下板成功了。于是我明白了规范地写代码有多么重要。需要特别注意 vivado 给出的报错信息，把 critical warnings 都尽量解决，是很有好处的。以及，仿真波形查 bug 是有力的工具。

问题三：trace 时发现我的数据冒险处理得不对，打开波形比对，发现前递的数据根本不对

解决方法：发现原来是写代码时，前递数据没有考虑具有数据相关性写回的数据来源，最开始的前递代码如下图所示。当我修改完这部分逻辑之后，我对前递操作有了更深的理解。

```
//前递
always @(posedge cpu_clk or posedge cpu_rst) begin
    if(cpu_rst)begin
        ID_EX_rd1       <= 0;
    end
    else if(rs1_id_ex_hazard)begin
        ID_EX_rd1       <= ALU_C;
    end
    else if(rs1_id_mem_hazard)begin
        ID_EX_rd1       <= EX_MEM_alu_c;
    end
    else if(rs1_id_wb_hazard)begin
        ID_EX_rd1       <= WB_RF_wD;
    end
    else ID_EX_rd1      <= RF_rD1;
end
```

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。

而且对外设和总线桥的理解也比上学期更深刻。

在学习完本课程后，我对 CPU 的体系结构、工作原理以及硬件实现有了更加深入的理解。通过从零开始搭建电路，我不仅巩固了计算机组成原理的相关知识，还掌握了如何将理论转化为实际的硬件设计。相比在课堂上对着电路图学习数据通路，看 PPT 一步一步添加冒险逻辑，经过实践，我对 CPU 的数据通路、控制单元、总线通信等核心模块有了更直观的认识。通过设计 ALU、寄存器堆、指令译码器等模块，我更加清晰地认识到 CPU 如何执行指令、处理数据以及协调各部件的工作。

在完成设计的过程中，及时的提问和求助也很重要。最开始，由于不熟悉龙芯指令集，指令和指令之间的操作看起来很不同，我的模块设计有些别扭和冗余复杂。多亏老师在前期数据通路设计的时候指出了我的设计不合理的地方，后续代码实现要轻松不少。

比起更快地动手，提前设计模块接口与连线、控制信号表和数据通路图是实现顺利的前提。我在做单周期 CPU 时，由于疏忽未注意到 bl 指令的写回寄存器要求，导致设计有所遗漏。缺少提前全面的考量，导致后续发现问题时修改的工作量很大。

本课程还让我复习和熟练了硬件描述语言 Verilog 的使用，对仿真调试波形的技巧掌握更进一层。在调试代码时，对着波形分析各信号的生成是最快最高效排查错误的方法。

此外，我对外设与总线通信的理解也更深了。上学期学习计算机组成原理时，我对外设与总线这章一知半解。在实验中，我通过修改总线桥，编写外设代码，理解了 CPU 如何与外设交互。这让我对计算机系统的整体架构有了更全面的认识，弥补了理论学习中的不足。

总的来说，本课程让我受益匪浅，不仅提升了我的硬件设计水平，也让我对计算机体系结构有了更深刻的理解。

意见与建议：

学校自主研发的龙芯汇编器一点开“打开文件”的图标，当前文件即使未保存也会直接清空。希望能完善这一点，退出文件时弹出询问是否要保存的选项。或者将“打开文件”和“保存”两个图标区分得更明显，可以加上文字说明。