



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

课程报告

开课学期: 2024 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: 56 地点: T2506

学生班级: 22 计科 6 班

学生学号: 220110609

学生姓名: 李子韬

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）

实现指令(miniLA 指令集必做+选做指令)：

3R 型：

add.w sub.w and or xor sll.w srl.w sra.w slt sltu

2RI5 型：

slli.w srli.w srai.w

2RI12 型：

addi.w andi ori xori slti sltui ld.b ld.bu ld.h ld.hu ld.w st.b
st.h st.w

1RI20 型：

lu12i.w pcaddu12i

2RI16 型：

beq bne blt bltu bge bgeu jirl

I26 型：

b bl

单周期频率：25MHz

流水线频率：50MHz

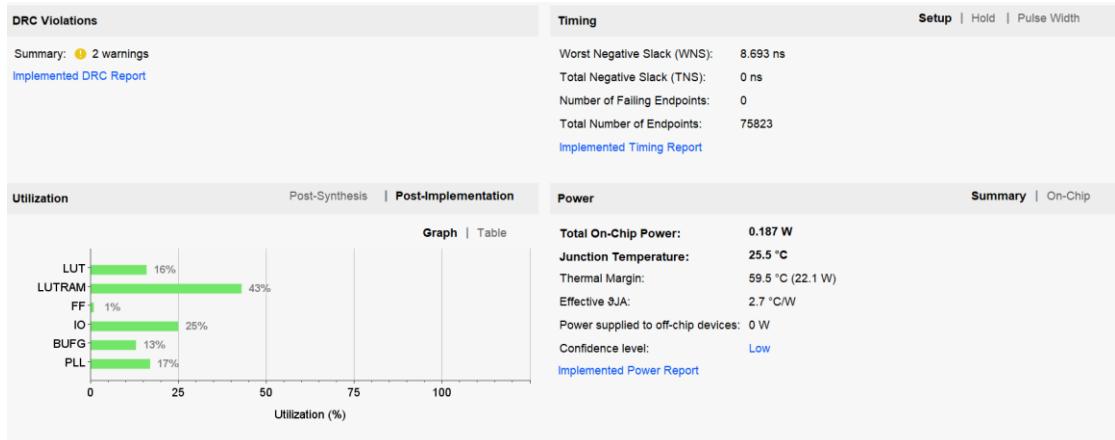
设计的主要特色（除基本要求以外的设计）

- 1、实现了龙芯指令集；
- 2、实现了 miniLA 选做指令
- 3、实现了静态分支预测

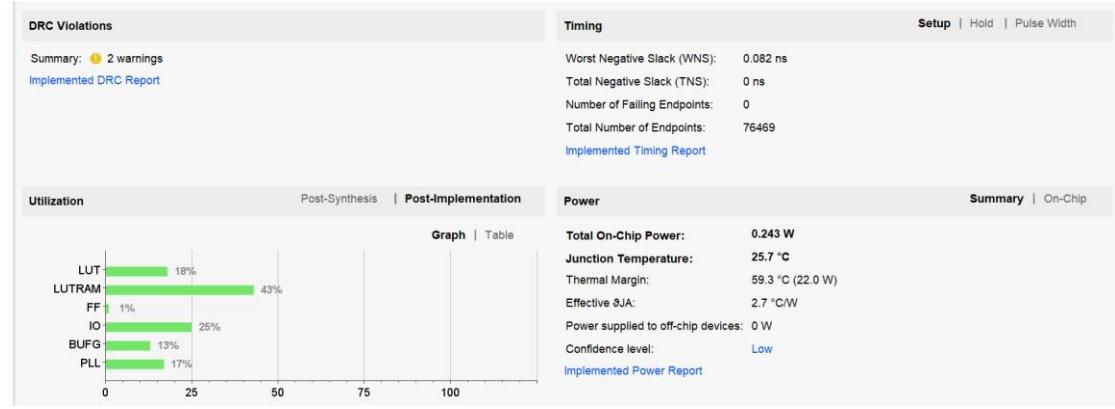
资源使用、功耗数据截图（Post Implementation；含 <u>单周期</u> 、 <u>流水线</u> 2个截图）

以下是示例，请贴自己的图。

单周期：



流水线：

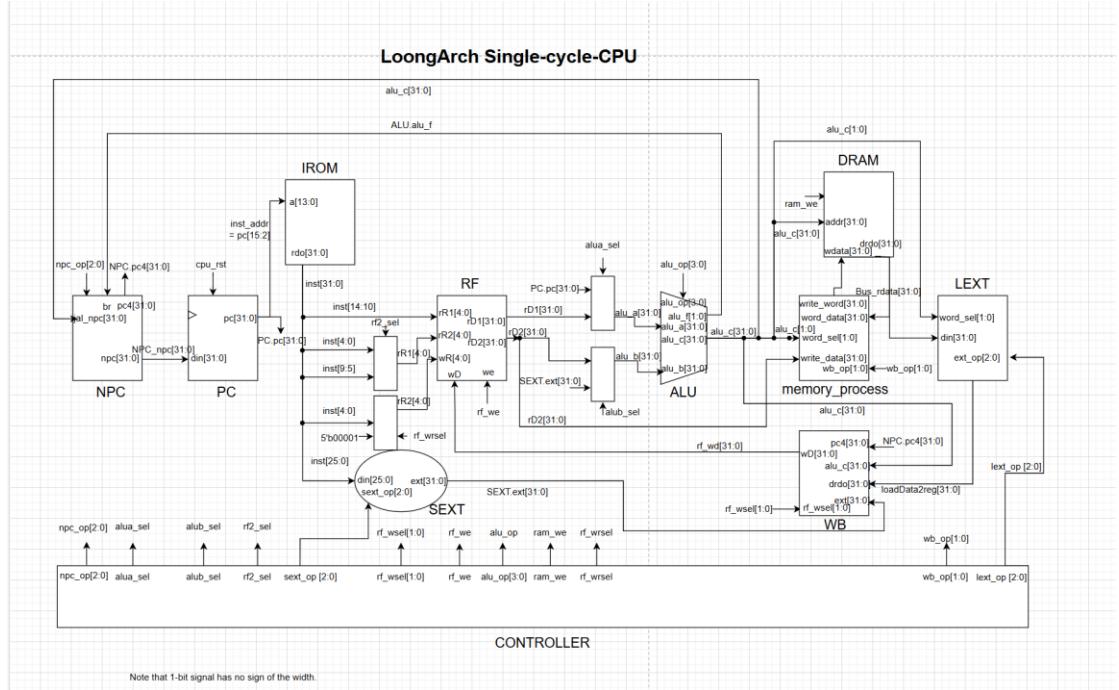


1 单周期 CPU 设计与实现

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。

数据通路图 (单周期):



(为使图像更清晰, 未标位宽的默认为 1-bit 数据)

模块定义和功能:

- **NPC:** 用于计算下一条要执行的指令地址, 传输给 PC;
- **PC:** 时序部件, 一个 32 位的地址寄存器, 记录当前指令的地址, npc 接口用于输入下一个执行的指令, 并在下一个时钟上升沿更新 PC;
- **IROM:** 指令只读存储器, 用于存储程序指令, 通过 PC 此刻存储的指令地址给出对应的执行指令 inst;
- **RF:** 时序部件堆, 存储 32 个 32 位寄存器的数据, 时钟上升沿写入选中的寄存器的数据, 其中第 1 个寄存器 x0 接地, 为 zero;
- **ALU:** 运算单元, 用于对两个输入操作数执行加法、减法、乘法、除法、与、或、非、逻辑移位(左右)、算术移位(右)、有符号比较、无符号比较运算, 并输出运算结果和标志位(判 0 或第一位, 根据具体运算而定);
- **DRAM:** 数据随机存取存储器, 用于写入和读出数据, 存储空间大, 起到主存作用, 每次读出和写入均以一个字(32-bit)为单位;
- **SEXT:** (指令) 符号拓展单元。用于根据指令低 26 位(minILA 指令集分布)生成 32 位立即数(有符号/无符号, minILA 中一共五种立即数生成类型), 传输到 EX 部分的 ALU 模块参与运算或 WB 部分写入目标寄存器(视具体指令);
- **LEXT:** (数据加载 Load) 符号拓展单元。根据指令取得所读取字的切片, 并做

符号拓展（五种），形成写入寄存器的 32 位数据；

- **memory_process:** 面向 DRAM 和 LEXT 的主存读取和写入数据综合处理器。可以将需要写入的数据嫁接入数据所在的字（写一个字节（有无符号）、半个字（有无符号）、一个字五种），再生成需要写入主存的字，将主存各种写入指令整合逻辑与主存分离；
- **WB:** 写回模块，根据 controller 的控制信号 rf_wsel[1:0] 处理四种可能写入寄存器的数据，形成写回的数据到 RF 模块，完成寄存器的写回操作；
- **controller:** 控制器模块，根据输入的指令生成各个模块的控制信号，指导各个模块完成指令需要的工作；
- 与此同时，在 RF 的 rR2 和 wR 接口前，在 ALU 两个操作数接口前，均设有多路选择器（二选一）分别通过 rf2_sel、rf_wrsel、alua_sel、alub_sel 控制，前二者用于不同指令分割指令的不同部分作为 RF 模块输入，后二者用于在 RF 读出数据和 pc、立即数数据之间选择成为 ALU 的操作数 A 和 B；

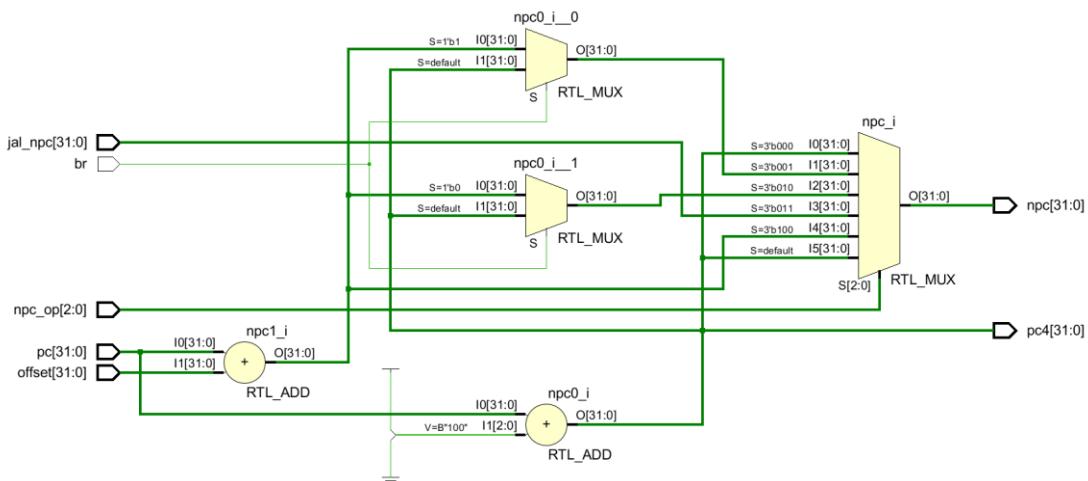
1.2 单周期 CPU 模块详细设计

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: 计算下一条执行指令地址的模块

接口类型	接口信号	位宽	功能描述
input wire	pc	32	当前指令地址
input wire	offset	32	跳转的指令偏移量
input wire	br	1	跳转校验位，来自 ALU 标志位输出
input wire	jal_npc	32	jirl 指令的无条件跳转指令地址（通过 ALU 算出）
input wire	npc_op	3	NPC 模块控制信号
output reg	npc	32	输出下一条执行指令的地址
output reg	pc4	32	输出当前指令的下一条顺位指令地址 (PC+4)

RTL 分析图：



核心代码：

```

case (npc_op)
`NPC_PC4: npc = pc + 4;
`NPC_BEQ: npc = (br == 1) ? (pc + offset) : (pc + 4);
`NPC_BNE: npc = (br == 0) ? (pc + offset) : (pc + 4);
`NPC_JAL: npc = jal_npc;
`NPC_BLT: npc = pc + offset;
default: npc = pc + 4; // Default case to handle undefined npc_op
values
endcase

```

关键实现的描述：

- 首先从数据通路图可知, br 接的是 alu_f 的 1-bit 信号在进行分支跳转判断时, beq 和 bne 采用 ALU_SUB 信号, alu_f 为 ALU 输出结果判零标志 (全 0 为 1);
- 设计时注意到按照 ALU 的设计, 在 NPC 模块中, blt 和 bge 可以复用 alu_f 的信号设计, 具体表现为: 当执行 blt 和 bge 指令时, ALU 收到 ALU_SLT 控制信号, 执行小于比较判断, 若 alu_c 为 1 则 blt 跳转而 bge 不跳转,

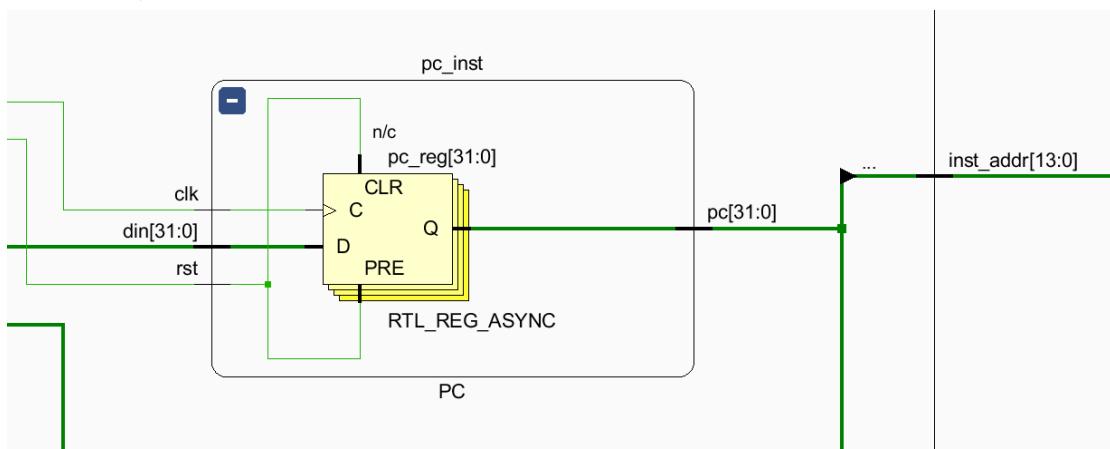
- 反之则反过来，此时将 alu_c 信号最低位同步到标志位 alu_f，就可以省下 NPC 的特判逻辑；
- 3、除此之外，设计时考虑了无条件跳转指令 jirl 和 bl，由于 jirl 需要读取寄存器信息执行加法计算，故经过 ALU，对此二条指令，采用特判逻辑，分别生成不同信号，对 jirl，将 jal_npc（即接通 alu_c[31:0]）输出为 npc，对 bl，将 offset 加在 pc 上输出为 npc；
 - 4、从 RTL 分析图来看，首先通过 br 的信号高低生成对应 BEQ 和 BNE 的输出结果，然后通过 npc_op 进行多路选择；（在代码中，还生成了 pc+4 作为 pc4 输出，会写入寄存器，故有两个加法器）

- PC：程序计数器，记录当前指令地址

接口类型	接口信号	位宽	功能描述
input wire	clk	1	时钟信号
input wire	rst	1	复位信号
input wire	din	32	输入下周期存储数据，来自 NPC 输出 (NPC_npc[31:0])
output reg	pc	32	当前记录的指令地址，输出到 IROM 读取指令

本质上是一个 32bit 的 D-触发器，在时钟上升沿更新数据；

RTL 分析图：



核心代码：

```
always @(posedge clk or posedge rst) begin
    if (rst == 1'b1) begin
        pc <= 32'hfffffc;
    end else begin
        pc <= din;
    end
end
```

关键实现的描述：

每个时钟上上升沿，可见用非阻塞赋值更新 pc 的值为 din，其次，在复位信号为高电平时，会重置 pc 为 32'hfffffc (trace 特性)；

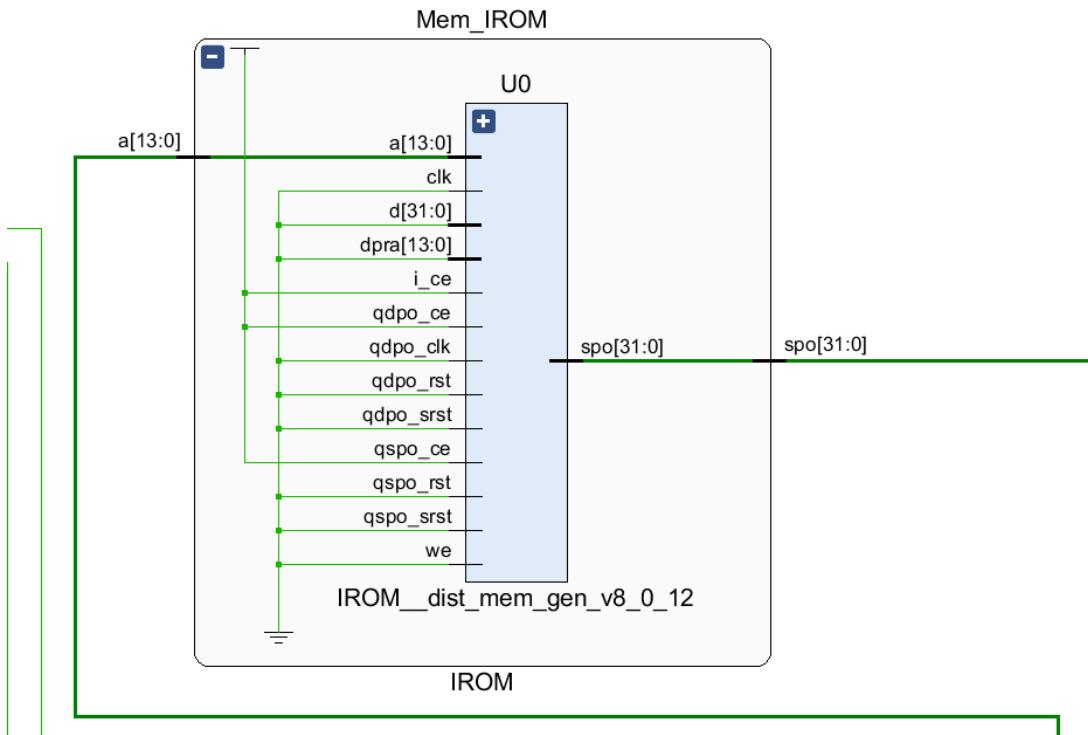
总而言之，实现了 32 位的 D 触发器，用于存储当前指令地址 pc，实现了程序计数器 PC；

• **IROM: 指令只读存储器**

宏观来看，以下是外接信号：

接口类型	接口信号	位宽	功能描述
input wire	a	14	指令的字地址 (inst[13:0]=pc[15:2])
output reg	spo	32	输出，从 IROM 读出的指令字

RTL 分析图：



核心代码：

这是 IF 模块的部分，指示如何分离指令字地址给 IROM

```
assign inst_addr = pc[15:2];
```

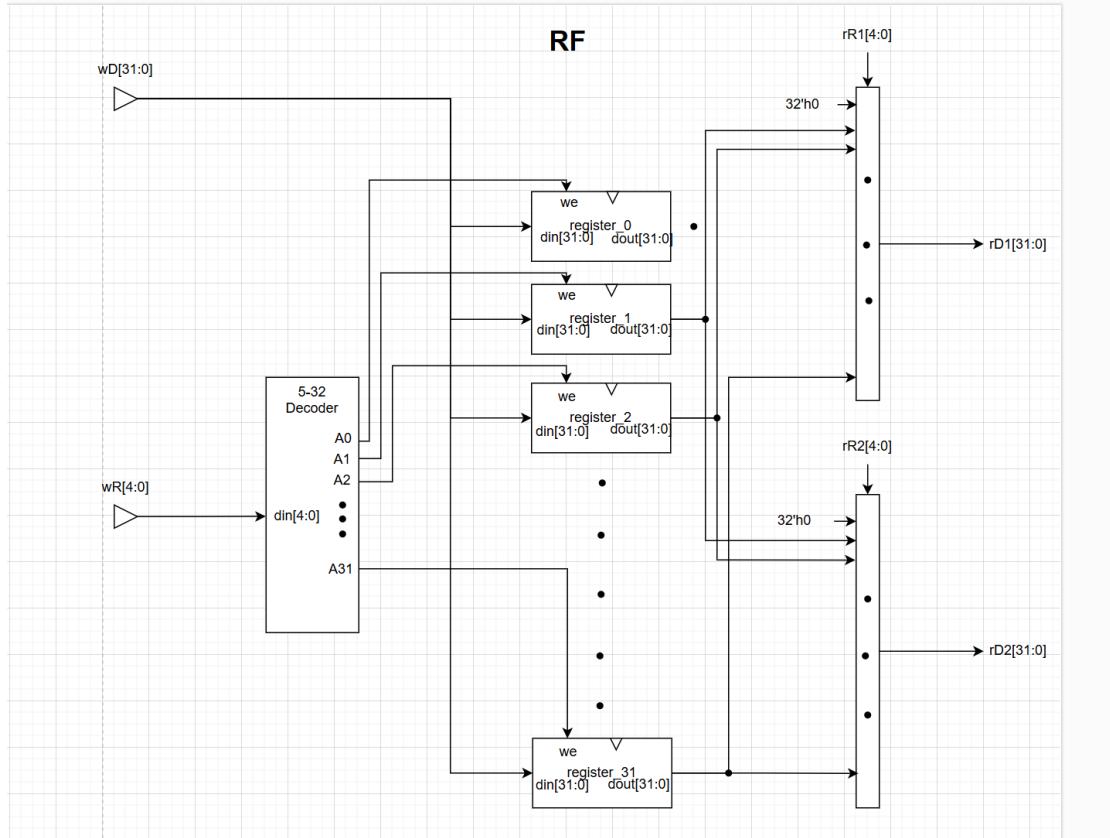
关键实现的描述：

使用的 ip catalog，存储空间为 16K*32bits；使用字寻址；通过分离 pc 的[15:2]部分输入 IROM 读取，读取是组合逻辑；

• **RF: Register Files 寄存器堆，用于存储寄存器数据，共 32 个 32 位寄存器**

接口类型	接口信号	位宽	功能描述
input wire	clk	1	时钟信号
input wire	rst	1	复位信号
input wire	rR1	5	读寄存器 1 编号
input wire	rR2	5	读寄存器 2 编号
input wire	wR	5	写寄存器编号
input wire	we	1	写使能（高电平有效）
input wire	wD	32	写寄存器写入数据
output reg	rD1	32	读寄存器 1 数据
output reg	rD2	32	读寄存器 2 数据

硬件框图：



核心代码：

修改部分：当 `rst` 高电平则重置所有为 0，每个时钟上升沿根据写寄存器编号更新寄存器数据为 `wD`；

```
always @(posedge rst or posedge clk) begin
    if (rst == 1'b1) begin
        registers[0]  <= 32'h0;
        (...)
        registers[31] <= 32'h0;
    end else if (we && (wR != 5'b0)) begin
        registers[wR] <= wD;
    end
end
```

读取部分：特判寄存器编号为 0 的部分，强制读出为 0，实现\$zero 寄存器

```
always @(*) begin
    rD1 = (rR1 == 5'b0) ? 32'h0 : registers[rR1];
    rD2 = (rR2 == 5'b0) ? 32'h0 : registers[rR2];
end
```

关键实现的描述：

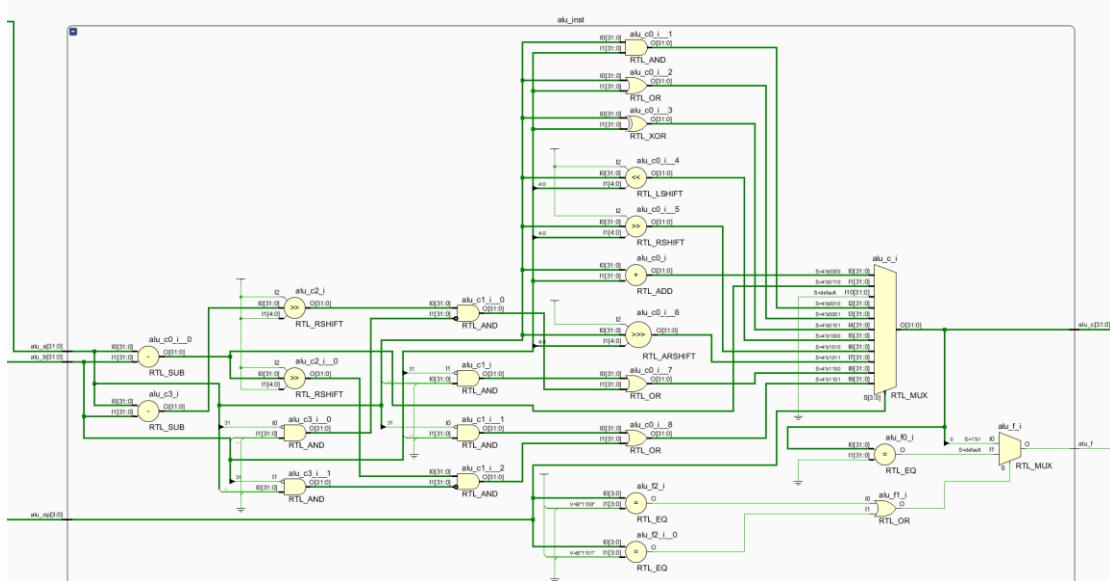
- 1、一是在模块内部定义了 `reg [31:0] registers [31:0]` 用于储存寄存器数据；
- 2、二是读出是组合逻辑，而写入是时序逻辑，这是为了让单周期 cpu 数据快速读出而且不受写回阶段数据突变的影响（可能会读到准备写回的数据参与后续的组合逻辑运算导致再生成新的写回数据）

3、三是对\$zero 寄存器的特判，可以修改，但读出强连通 0，实现强接 0 的效果；

- **ALU：运算逻辑单元**

接口类型	接口信号	位宽	功能描述
input wire	alu_a	32	操作数 A
input wire	alu_b	32	操作数 B
input wire	alu_op	4	控制信号 alu_op
output reg	alu_c	32	运算结果
output reg	alu_f	1	标志位

RTL 分析图：



核心代码：

```

always @(*) begin
    case (alu_op)
        `ALU_ADD: alu_c = alu_a + alu_b;
        `ALU_SUB: alu_c = alu_a - alu_b;
        `ALU_AND: alu_c = alu_a & alu_b;
        `ALU_OR: alu_c = alu_a | alu_b;
        `ALU_XOR: alu_c = alu_a ^ alu_b;
        `ALU_SLL: alu_c = alu_a << alu_b[4:0];
        `ALU_SRL: alu_c = alu_a >> alu_b[4:0];
        `ALU_SRA: alu_c = signed_a >>> alu_b[4:0];
        `ALU_SLT: alu_c = (signed_a[31] & ~signed_b[31]) |
                     ((signed_a - signed_b) >> 31 &
                     (~signed_a[31] & signed_b[31])); // Signed comparison
        `ALU_SLTU: alu_c = (~alu_a[31] & alu_b[31]) |
                     ((alu_a - alu_b) >> 31 & ~(alu_a[31] &
                     ~alu_b[31])); // Unsigned comparison
        default: alu_c = 32'd0;
    endcase

```

```

// Signature flag
if (alu_op == `ALU_SLT | alu_op == `ALU_SLTU) alu_f = alu_c[0];
else alu_f = (alu_c == 32'd0);
end

```

总体而言，ALU 通过不同控制信号实现不同的运算和比较功能，具体定义如下：
关键实现：

ALU 模块根据不同的 alu_op 控制信号(来自 controller)对操作数 alu_a 和 alu_b 作不同的运算：

不同的 alu_op 值代表不同的操作：定义如下

- ALU_ADD: 执行加法运算, $alu_c = alu_a + alu_b$
- ALU_SUB: 执行减法运算, $alu_c = alu_a - alu_b$
- ALU_AND: 执行按位与运算, $alu_c = alu_a \& alu_b$
- ALU_OR: 执行按位或运算, $alu_c = alu_a | alu_b$
- ALU_XOR: 执行按位异或运算, $alu_c = alu_a ^ alu_b$
- ALU_SLL: 执行逻辑左移运算, $alu_c = alu_a << alu_b[4:0]$
- ALU_SRL: 执行逻辑右移运算, $alu_c = alu_a >> alu_b[4:0]$
- ALU_SRA: 执行算术右移运算, $alu_c = alu_a >>> alu_b[4:0]$
- ALU_SLT: 有符号数比较, $alu_c = (alu_a < alu_b)$
- ALU_SLTU: 无符号数比较, $alu_c = (alu_a < alu_b)$
- 对于标志位 alu_f:
如果操作是比较运算 (ALU_SLT 或 ALU_SLTU), alu_f 会直接跟随 alu_c[0],
用于分支比较跳转的标志，在先前 NPC 部分有提到。
否则，如果结果为 0，则 alu_f 为 1，否则为 0 (判零逻辑);
- 从实现上来说，ALU 的框图总体上有一个多路选择器，上面接上各种运算单元
产出的结果，并通过 alu_op 来决定 alu_c 输出哪一个，alu_f 亦然；
- 还有一点较为关键的，就是无符号数和有符号数的问题，使用 wire signed 来
区分输入的有符号和无符号数，确保移位和比较运算正确处理有符号数。

• SEXT: 符号拓展单元 (指令分割)

接口类型	接口信号	位宽	功能描述
input wire	din	26	需要拓展的立即数片段
input wire	sext_op	3	符号拓展单元 SEXT 控制信号
output reg	ext	32	生成的 32 位立即数

核心代码：

根据 sext_op 选择拓展方式：

```

always @(*) begin
    case (sext_op)
        `EXT_I5: begin
            // Unsigned 5-bit immediate
            ext = {27'b0, din[14:10]};
        end
        `EXT_I12: begin

```

```

        // Signed 12-bit immediate
        ext = {{20{din[21]}}, din[21:10]};
    end
    `EXT_I12U: begin
        // Zero extended 12-bit immediate
        ext = {20'b0, din[21:10]};
    end
    `EXT_I20: begin
        // Signed 20-bit immediate shifted left by 12 bits
        ext = {din[24:5], 12'b0};
    end
    `EXT_I16: begin
        // Signed 16-bit immediate shifted left by 2 bits
        ext = {{14{din[25]}}, din[25:10], 2'b0};
    end
    `EXT_I26: begin
        ext = {{4{din[9]}}, din[9:0], din[25:10], 2'b0};
    end
    default: begin
        // Default case to handle unexpected sext_op values
        ext = 32'b0;
    end
endcase
end

```

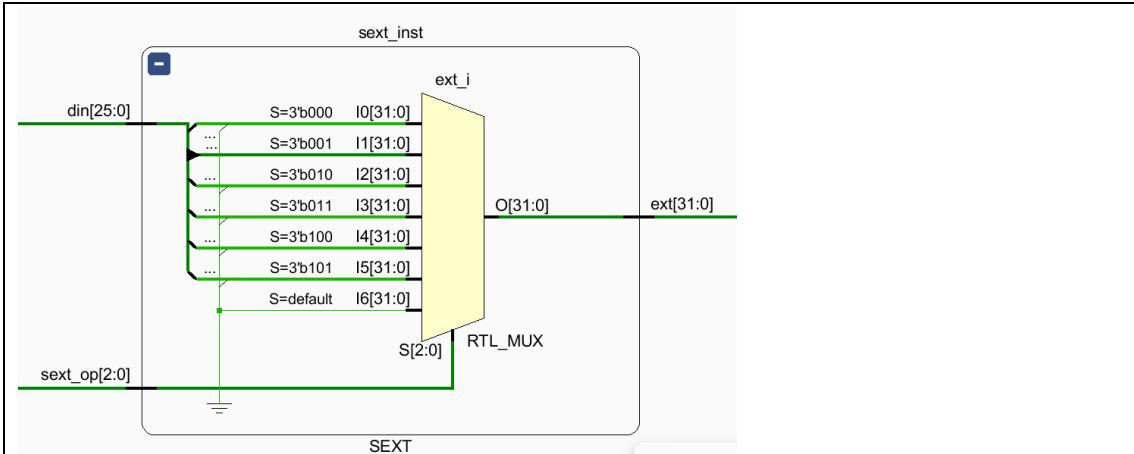
控制信号对应的操作如下：

- EXT_I5: ui5 = din[14:10], 并得到 ext = {27'b0, ui5}, 用于 2RI5 指令
- EXT_I12: si12 = din[21:10], 并得到 ext = sext(si12), 有符号拓展, 用于部分 2RI12 指令
- EXT_I12U: zi12 = din[21:10], 并得到 ext = zext(si12), 无符号拓展, 用于部分 2RI12 指令
- EXT_I20: si20 = din[24:5], 并得到 ext = {si20, 12'b0}, 用于 1RI20 指令
- EXT_I16: offs = din[25:10], 并得到 ext = sext({offs, 2'b0}), 用于 2RI16 指令

关于 EXT_I12U, 只适用于 andi, xori, ori;
sext 表示符号扩展, 而 zext 表示零扩展

RTL 分析图:

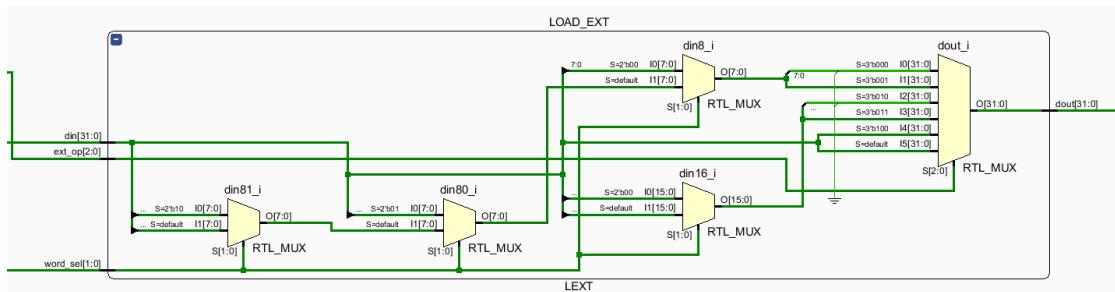
首先生成各种形式的拓展立即数, 然后通过 sext_op 进行多路选择确定输出结果



• LEXT: 符号拓展单元 (数据读取)

接口类型	接口信号	位宽	功能描述
input wire	din	32	主存中读取的需要解析的字
input wire	ext_op	3	符号拓展单元 LEXT 控制信号
input wire	word_sel	2	字内取片段偏置，用于解析
output reg	ext	32	生成的 32 位立即数

RTL 分析图：



核心代码和关键实现：

字段切割方面：

```
wire [7:0] din8 = (word_sel == 2'b00) ? din[7:0] :  
                      (word_sel == 2'b01) ? din[15:8] :  
                      (word_sel == 2'b10) ? din[23:16] :  
                      din[31:24];  
  
wire [15:0] din16 = (word_sel == 2'b00) ? din[15:0]:  
                      din[31:16];
```

根据主存提供的地址最低两位，可以得到所需取到的字中偏置的半字或字节，分别放入 din16 和 din8 中，并提供给后续操作；

字段拓展方面：

```
always @(*) begin  
    case(ext_op)  
        `LEXT_8U : dout = {{24{1'b0}}, din8}; // Zero extend 8-bit  
        `LEXT_8  : dout = {{24{din8[7]}}, din8}; // Sign extend 8-bit  
        `LEXT_16U: dout = {{16{1'b0}}, din16}; // Zero extend 16-bit  
        `LEXT_16 : dout = {{16{din16[15]}}, din16}; // Sign extend 16-bit
```

```

`LEXT_32 : dout = din;                                // Output 32-bit
original data
    default: dout = din;
endcase
end

```

可见，ext_op 有五种控制信号（来自 controller），分别代表五种操作：

- LEXT_8U：将截取的字节作无符号拓展到 32 位
- LEXT_8：将截取的字节作有符号拓展到 32 位
- LEXT_16U：将截取的半字作无符号拓展到 32 位
- LEXT_16：将截取的半字作有符号拓展到 32 位
- LEXT_32：不拓展，直接输出整个字

从 RTL 分析图上看，其先通过多路选择器 MUX 分离字节和半字，再形成多种拓展的结果，最后通过多路选择器用 ext_op 选择输出；

DRAM：数据随机存取器

宏观来看，以下是外接信号：

接口类型	接口信号	位宽	功能描述
input wire	clk	1	时钟信号（写）
input wire	a	14	数据的字地址 (a [13:0]=Bus_addr[15:2])
output reg	spo	32	从 DRAM 读出的数据字
input wire	we	1	写使能
input wire	d	32	写入地址 a 的字

关键设计：

```

DRAM Mem_DRAM (
    .clk      (clk_bridge2dram),
    .a        (addr_bridge2dram[15:2]),
    .spo      (rdata_dram2bridge),
    .we       (we_bridge2dram),
    .d        (wdata_bridge2dram)
);

```

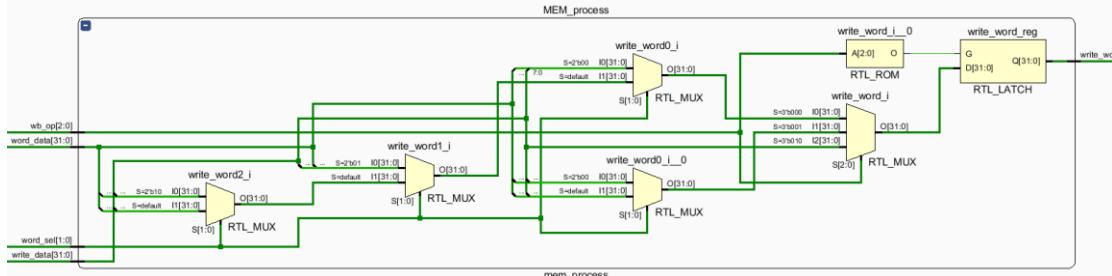
- 使用的 ip catalog，存储空间为 16K*32bits；使用字寻址；通过分离 Bus_addr 的[15:2]部分输入 DRAM 读取，读取是组合逻辑，通过 Bridge 连接 CPU；
- 对于写入部分，当 we=1 的时钟上升沿时，将 d 写入 a 所指向的字，写入的前处理交由 memory_process 模块完成（为了完成半字和字节写入），主要是根据指令拼接字接到 d 接口（有些像单体多字的思想，但是这是单字），写入是时序逻辑；

- **memory_process：**写入主存前处理模块

接口类型	接口信号	位宽	功能描述
input wire	wb_op	2	前处理指令
input wire	word_sel	32	写入数据在字的偏移量
input wire	word_data	32	读出的字

input wire	write_data	32	写入的数据, 可能是后八位和后十六位
output reg	write word	32	拼接的字, 输出给 DRAM 的 d 接口写入

RTL 分析图:



核心代码:

```

wire [7:0] data_byte = write_data[7:0];
wire [15:0] data_hex = write_data[15:0];

always@(*) begin
    case(wb_op)
        `WB_BYTE: write_word = (word_sel == 2'b00) ? {word_data[31:8], data_byte} :
                    (word_sel == 2'b01) ? {word_data[31:16], data_byte, word_data[7:0]} :
                    (word_sel == 2'b10) ? {word_data[31:24], data_byte, word_data[15:0]} :
                                         {data_byte, word_data[23:0]};
        `WB_HEX:  write_word = (word_sel == 2'b00) ? {word_data[31:16], data_hex} :
                    (word_sel == 2'b01) ? {word_data[31:24], data_hex, word_data[7:0]} :
                                         {data_hex, word_data[15:0]};
        `WB_WORD: write_word = write_data; // write entire word
    endcase
end

```

详细设计:

先将需要写的数据分离字节和半字，再根据写入的类型（字节、半字、字）来决定替代字的多少部分，再通过字的偏移量来确定替代哪一部分，在实现上，仍是生成多个结果，再用多路选择器选择，存到 reg 内；

控制信号代表的含义：

- **WB_BYTE**: 改变一个字节，有四个字节可能被改变；
- **WB_HEX**: 改变一个半字，可能是前半，中半，后半；
- **WB_WORD**: 无需拼接，整字替代；

• WB: 写回取值模块

接口类型	接口信号	位宽	功能描述
input wire	inst	32	当前指令
input wire	aluc	32	ALU 运算结果 C
output reg	pc4	32	pc+4, 来自 npc
input wire	drdo	32	DRAM 读出拓展数据
input wire	ext	32	SEXT 立即数
input wire	rf_wrsel	1	写寄存器编号选择控制信号
input wire	rf_wsel	2	写寄存器数据选择控制信号
output reg	wR	5	写寄存器编号
output reg	wD	32	写寄存器数据

核心代码:

```

always@(*) begin
    case(rf_wsel)
        `RFW_ALUC: wD = aluc;
        `RFW_DRAM: wD = drdo;
        `RFW_SEXT: wD = ext;
        `RFW_NPC:  wD = pc4;
        default:   wD = 32'h0;
    endcase
    wR  = (rf_wrsel == `RFWR_N) ? inst[4:0] : 5'b00001;
end

```

实现描述：实质上是两个多路选择器，分别受 rf_wrsel 和 rf_wsel 控制；

- **controller:**

这是生成控制信号的集成器件，负责解析指令，从而对所有控制信号赋值，是组合逻辑；

接口类型	接口信号	位宽	功能描述
input wire	inst	32	当前指令
output reg	npc_op	3	NPC 控制信号
output reg	rf_wsel	2	写寄存器数据选择信号
output reg	rf_wrsel	1	写寄存器编号选择信号
output reg	rf_we	1	写寄存器使能
output reg	rf2_sel	1	读寄存器 2 编号选择
output reg	sext_op	3	SEXT 立即数拓展控制信号
output reg	lext_op	3	LEXT 立即数拓展控制信号
output reg	alu_op	4	ALU 操作控制信号
output reg	alua_sel	1	ALU 操作数 A 选择信号
output reg	alub_sel	1	ALU 操作数 B 选择信号
output reg	ram_we	1	主存写使能
output reg	wb_op	2	memory_process 控制信号

关键实现：

先根据 miniLA 指令集进行不同指令段的划分，再对这四个分段进行判断生成控制信号，每个 always 块生成一个控制信号；

```

// Opcode and function extraction
wire [5:0] opcode = inst[31:26];
wire      funct1 = inst[25];
wire [2:0] funct3 = inst[24:22];
wire [6:0] funct7 = inst[21:15];

```

例如，对 npc_op：有如下生成：

```
// NPC_OP
always @(*) begin
    case (opcode)
        6'b000000: npc_op = `NPC_PC4;
        6'b001010: npc_op = `NPC_PC4;
        6'b000101: npc_op = `NPC_PC4;
        6'b000111: npc_op = `NPC_PC4; // Normal
        6'b010110: npc_op = `NPC_BEQ; // beq
        6'b010111: npc_op = `NPC_BNE; // bne
        6'b011000: npc_op = `NPC_BEQ; // blt
        6'b011010: npc_op = `NPC_BEQ; // bltu
        6'b011001: npc_op = `NPC_BNE; // bge
        6'b011011: npc_op = `NPC_BNE; // bgeu
        6'b010011: npc_op = `NPC_JAL; // jirl
        6'b010100: npc_op = `NPC_BL; // b
        6'b010101: npc_op = `NPC_BL; // bl
    default: npc_op = 2'b00;
    endcase
end
```

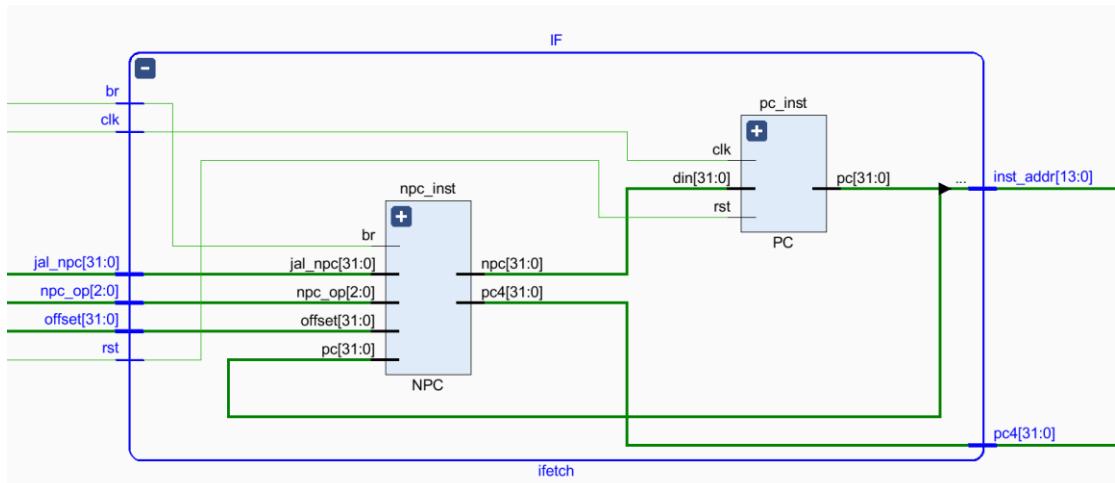
分别对应了不同指令，这也是对 npc 控制信号的一种诠释，其他略；

- 其他集成模块部分：

- IF 模块：整合 NPC 和 PC，作为流水线的一级 (instruction fetch)

接口类型	接口信号	位宽	功能描述
input wire	clk	1	时钟信号
input wire	rst	1	复位信号
input wire	offset	32	跳转的指令偏移量
input wire	br	1	跳转校验位，来自 ALU 标志位输出
input wire	jal_npc	32	jirl 的无条件跳转指令地址
input wire	npc_op	3	NPC 模块控制信号
output wire	pc4	32	输出当前指令的下一条顺位指令地址 (PC+4)
output wire	inst_addr	14	当前指令字地址，给 IROM

只是实例化了 NPC 和 PC 模块，并将输出给 IROM 的指令地址取出；



核心代码：实例化

```
NPC npc_inst (
    .pc(pc),
```

```

    .offset(offset),
    .jal_npc(jal_npc),
    .br(br),
    .npc_op(npc_op),
    .npc(NPC_npc),
    .pc4(pc4)
);

PC pc_inst (
    .clk(clk),
    .rst(rst),
    .din(NPC_npc),
    .pc(pc)
);

```

关键实现：整合 NPC 和 PC 为一个模块，并且提取 inst_addr(在 IROM 描述)

- ID 模块：集成了 RF 和 SEXT，作为流水线的一级

```

assign rR1 = inst[9:5];                                // rj
assign rR2 = (rf2_sel == `RF2_RK) ? inst[14:10] : inst[4:0]; // rk or rd

```

这是 rf2_sel 的控制部分，解析解决 miniLA 不同的指令的操作数位置不一样的问题；

- EX 模块：包括 ALU 和 ALU 操作数预处理，作为流水线的一级
ALU 预处理选择逻辑：

```

// Selection logic for ALU A
assign alu_a = (alu_a_sel == `ALUA_PC) ? pc : rf_rD1;

// Selection logic for ALU B
assign alu_b = (alu_b_sel == `ALUB_SEXT) ? ext : rf_rD2;

```

具体与数据通路表一致；

- MEM 模块：集成 memory_process 和 LEXT；
- WB 模块，即前面提到的 WB 模块；

接口总表：

部分	部件和功能	接口类型	接口信号	位宽	功能描述
IF	NPC：确定下一个执行的指令地址	input wire	pc	32	当前指令地址
		input wire	offset	32	跳转的指令偏移量
		input wire	br	1	跳转校验位，来自 ALU 标志位输出
		input	jal_npc	32	jirl 指令的无条件跳转

		wire			指令地址 (通过 ALU 算出)
	NPC	input wire	npc_op	3	NPC 模块控制信号
		output reg	npc	32	输出下一条执行指令的地址
		output reg	pc4	32	输出当前指令的下一条顺位指令地址 (PC+4)
	PC：存储当前指令地址	input wire	clk	1	时钟信号
		input wire	rst	1	复位信号
		input wire	din	32	输入下周期存储数据，来自 NPC 输出 (NPC_npc[31:0])
		output reg	pc	32	当前记录的指令地址，输出到 IROM 读取指令
ID	RF：负责存储寄存器数据和处理寄存器的读出、写入	input wire	clk	1	时钟信号
		input wire	rst	1	复位信号
		input wire	rR1	5	读寄存器 1 编号
		input wire	rR2	5	读寄存器 2 编号
		input wire	wR	5	写寄存器编号
		input wire	we	1	写使能 (高电平有效)
		input wire	wD	32	写寄存器写入数据
		output reg	rD1	32	读寄存器 1 数据
		output reg	rD2	32	读寄存器 2 数据
EX	ALU：对操作数执行计算和比较，输出结果	input wire	alu_a	32	操作数 A
		input wire	alu_b	32	操作数 B
		input wire	alu_op	4	控制信号 alu_op

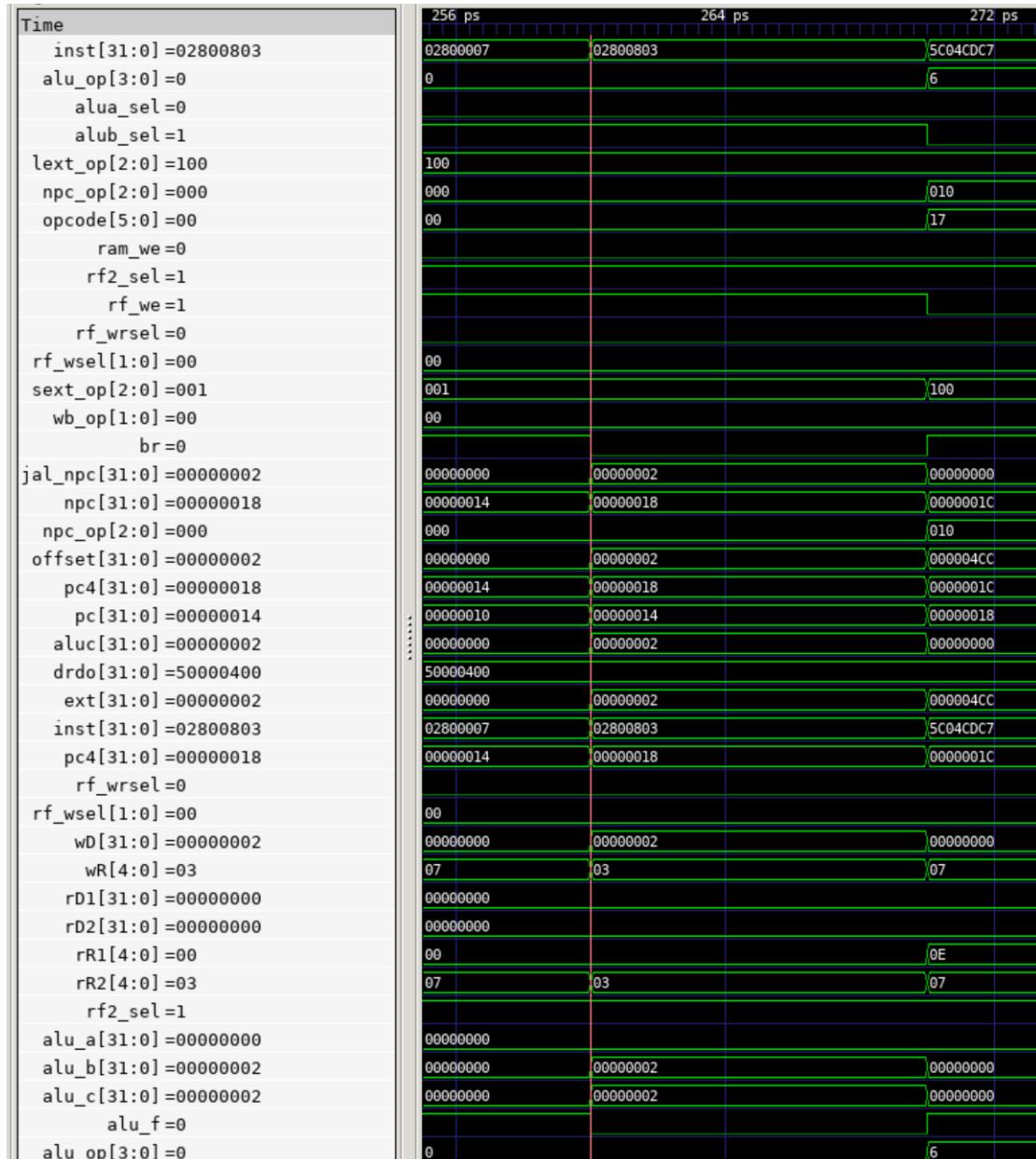
		output reg	alu_c	32	运算结果
		output reg	alu_f	1	标志位
ID	SEXT: 根据指令和控制信号生成 32 位立即数	input wire	din	26	需要拓展的立即数片段
		input wire	sextr_op	3	符号拓展单元 SEXT 控制信号
		output reg	ext	32	生成的 32 位立即数
MEM	LEXT: 根据读出数据和指令类型生成 32 位主存读出数据	input wire	din	32	主存中读取的需要解析的字
		input wire	ext_op	3	符号拓展单元 LEXT 控制信号
		input wire	word_sel	2	字内取片段偏置，用于解析
		output reg	ext	32	生成的 32 位立即数
memory_process:	生成主存写入的最终字	input wire	wb_op	2	前处理指令
		input wire	word_sel	32	写入数据在字的偏移量
		input wire	word_data	32	读出的字
		input wire	write_data	32	写入的数据，可能是后八位和后十六位
		output reg	write_word	32	拼接的字，输出给 DRAM 的 d 接口写入
WB	WB: 根据指令控制信号生成写入寄存器的数据和写入寄存器的编号	input wire	inst	32	当前指令
		input wire	aluc	32	ALU 运算结果 C
		output reg	pc4	32	pc+4, 来自 npc
		input wire	drdo	32	DRAM 读出拓展数据
		input wire	ext	32	SEXT 立即数
		input wire	rf_wrsel	1	写寄存器编号选择控制信号
		input wire	rf_wsel	2	写寄存器数据选择控制信号
		output	wR	5	写寄存器编号

		reg			
		output reg	wD	32	写寄存器数据
controller 根据分割 指令生成 各个部件 的控制信 号	controller 根据分割 指令生成 各个部件 的控制信 号	input wire	inst	32	当前指令
		output reg	npc_op	3	NPC 控制信号
		output reg	rf_wsel	2	写寄存器数据选择信 号
		output reg	rf_wrsel	1	写寄存器编号选择信 号
		output reg	rf_we	1	写寄存器使能
		output reg	rf2_sel	1	读寄存器 2 编号选择
		output reg	sext_op	3	SEXT 立即数拓展控 制信号
		output reg	lext_op	3	LEXT 立即数拓展控 制信号
		output reg	alu_op	4	ALU 操作控制信号
		output reg	alua_sel	1	ALU 操作数 A 选择信 号
		output reg	alub_sel	1	ALU 操作数 B 选择信 号
		output reg	ram_we	1	主存写使能
		output reg	wb_op	2	memory_process 控制 信号

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：



当前时间：260ns

当前指令：0x02800803

二进制：0000 0010 1000 0000 0000 1000 0000 0011

miniLA 指令集翻译：addi.w \$x3, \$x0, 2

期望结果：x3 赋值为 2

过程分析：

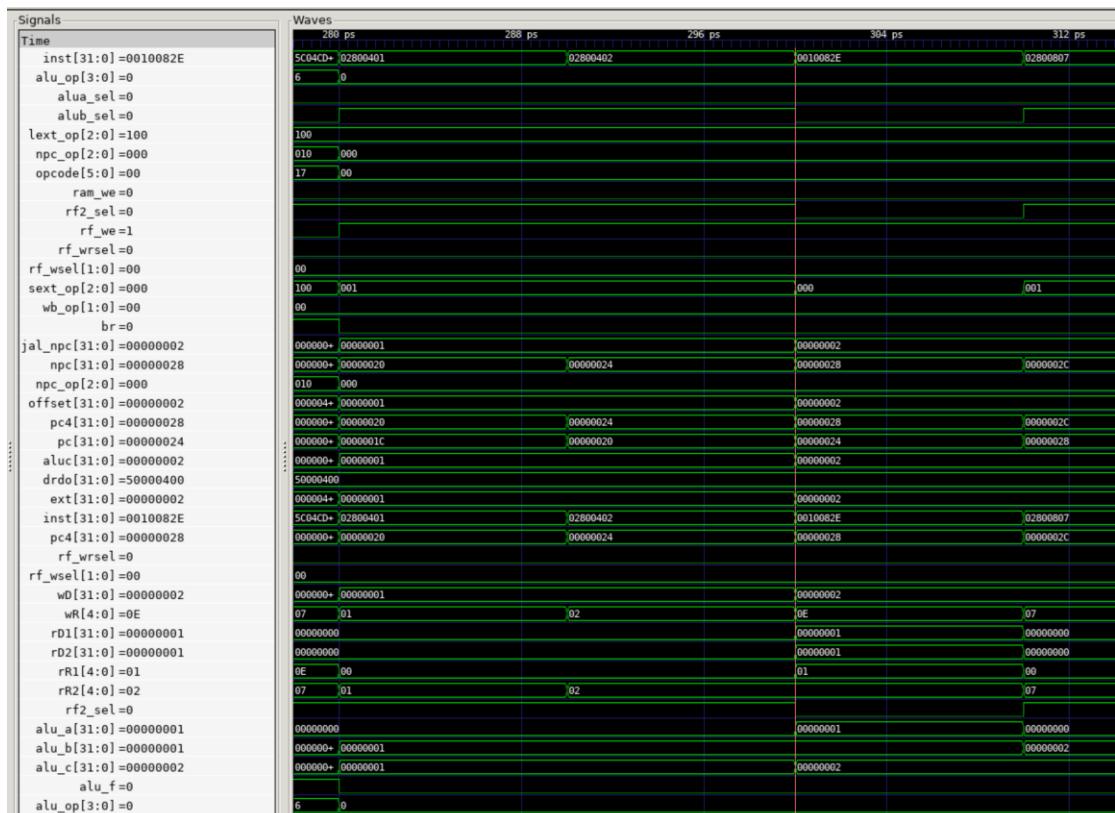
- npc_op 为 0， npc = pc+4， 可见 pc=0x00000014， npc=0x00000018， 符合预期；

- pc=0x00000014， 读出 inst=02800803， controller 生成 SEXT 控制信号

- sex=001, 解析 2RI12 指令, 符号拓展, 生成 ext=0x00000002, 符合预期;
- x0 强连通 0, rR1 = 0, rD1 = 0, 符合预期;
 - wR 是写寄存器, rf_wrsel = 0, 选择 x3, wR=3 符合预期;
 - rD1 和 ext 将进入 ALU 作为两个操作数, alua_sel=0, alub_sel=1, 对操作数 A 选择 rD1, 对操作数 B 选择 ext 而非 rD2; 可见 alu_a = 0, alu_b = 2; alu_op = ALU_ADD(即 0000), 符合预期; 输出 alu_c = 2, 符合预期;
 - 无访存, 只有写回, 写回部分 rf_wsel=00, 采用 ALU 结果 alu_c 写回, 可见 wD=0x00000002, 符合预期;



后续, x3 被写入 0x00000002, 符合预期;



当前时间: 300ns

当前指令: 0x0010082E

二进制: 0000 0000 0001 0000 0000 1000 0010 1110

miniLA 指令集翻译: add.w \$x14, \$x1, \$x2

此时 x1 = 1, x2 = 1, 预计 x14 = 2, 从结果而言符合预期;



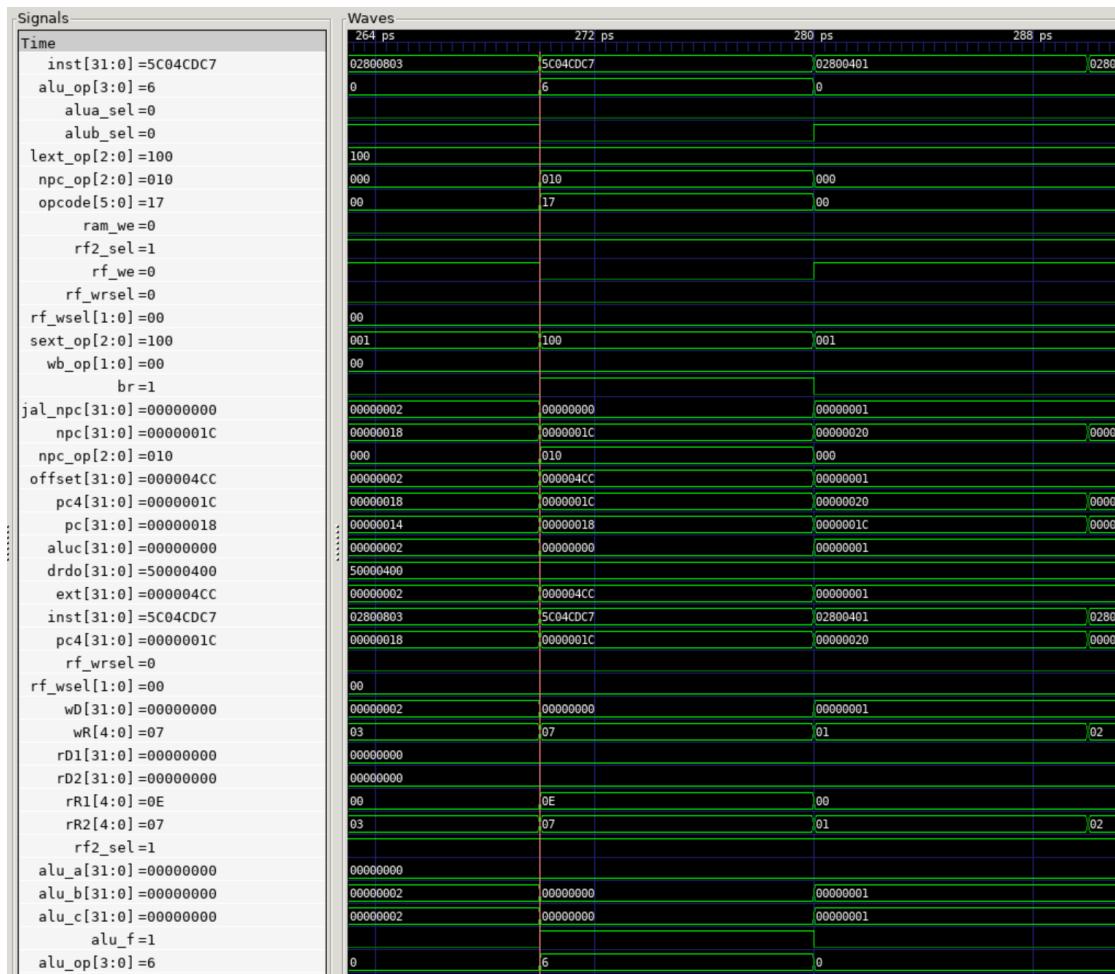
过程分析:

- npc_op 为 0, npc = pc+4, 可见 pc=0x00000028, npc=0x0000002C, 符合预期;

- $x1 = 1$, 读寄存器编号 $rR1 = 1$, 读出 $rD1 = 1$, 符合预期;
 - $x2 = 1$, $rf2_sel = 0$, 选取 $inst[14:10]$ 为读寄存器编号 $rR2 = 2$, 读出 $rD2 = 1$, 符合预期;
 - wR 是写寄存器, $rf_wrsel = 0$, 选择 $x14$, $wR=0E$ 符合预期;
 - $rD1$ 和 $rD2$ 将进入 ALU 作为两个操作数, $alua_sel=0$, $alub_sel=0$, 对操作数 A 选择 $rD1$, 对操作数 B 选择 $rD2$; 可见 $alu_a = 1$, $alu_b = 1$; $alu_op = ALU_ADD$ (即 0000), 符合预期; 输出 $alu_c = 2$, 符合预期;
 - 无访存, 只有写回, 写回部分 $rf_wsel=00$, 采用 ALU 结果 alu_c 写回, 可见 $wD=0x00000002$, 符合预期;
- 可见在这之后 $x14$ 变为 0x00000002, 符合预期, 功能正常;

访存:

分支跳转:

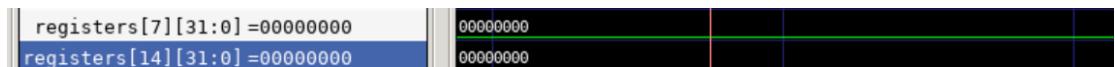


当前时间: 270ns

当前指令: 0x5C04CDC7

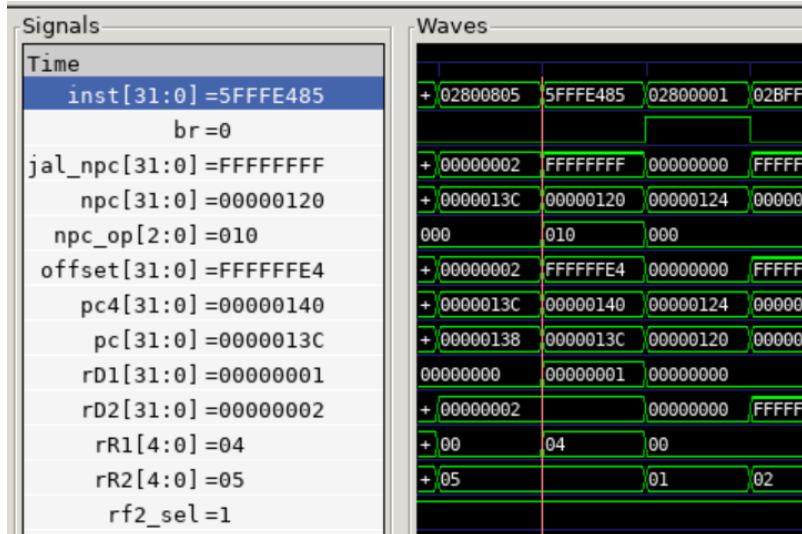
二进制: 0101 1100 0000 0100 1100 1101 1100 0111

miniLA 指令翻译: bne \$x15, \$x7, 0x812C



此时 $x14 = x7 = 0$; 理论上不跳转;

- $npc_op = 010$ (NPC_BNE), 当 $br = 0$ 时跳转, 我们来观察 br ;
- 首先, $x14 = 0$, 读寄存器编号 $rR1 = 0E$, 读出 $rD1 = 0$, 符合预期;
- $x7 = 0$, $rf2_sel = 1$, 选取 $inst[4:0]$ 为读寄存器编号 $rR2 = 07$, 读出 $rD2 = 0$, 符合预期;
- 之后 $alu_op = 0110$ (ALU_SUB), $alua_sel=0$, $alub_sel=0$, 对操作数 A 选择 $rD1$, 对操作数 B 选择 $rD2$, 相减得 $alu_c = 0$; 判 0 标志 $alu_f = 1$; 符合预期;
- $br = alu_f = 1$; 可知不跳转, 此时 $npc = pc+4 = 0x0000001C$, 不跳转, 符合预期, 功能正常;

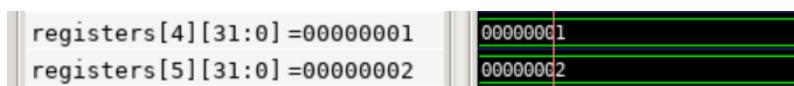


当前时间: 1030ns

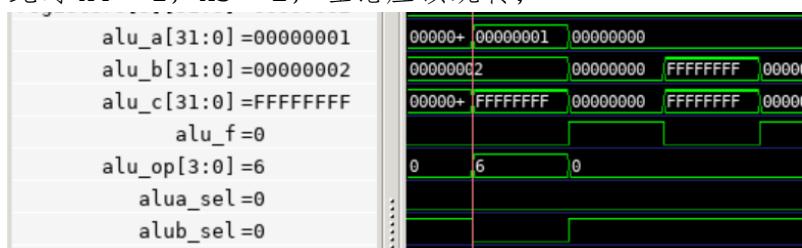
当前指令: 0x5FFE485

二进制: 0101 1111 1111 1111 1110 0100 1000 0101

miniLA 指令翻译: bne \$x4, \$x5, 0xFFFF9



此时 $x4 = 1$, $x5 = 2$, 理论应该跳转;



- $npc_op = 010$ (NPC_BNE), 当 $br = 0$ 时跳转, 我们来观察 br ;
- 首先, $x4 = 1$, 读寄存器编号 $rR1 = 04$, 读出 $rD1 = 1$, 符合预期;
- $x5 = 2$, $rf2_sel = 1$, 选取 $inst[4:0]$ 为读寄存器编号 $rR2 = 05$, 读出 $rD2 = 2$, 符合预期;
- 之后 $alu_op = 0110$ (ALU_SUB), $alua_sel=0$, $alub_sel=0$, 对操作数 A 选择 $rD1$, 对操作数 B 选择 $rD2$, 相减得 $alu_c = FFFFFFFF$; 判 0 标志 $alu_f = 0$; 符合预期;
- $br = alu_f = 0$; 可知跳转, 此时 $npc = pc+offset = 0x00000120$, 跳转, 符合预期, 功能正常;

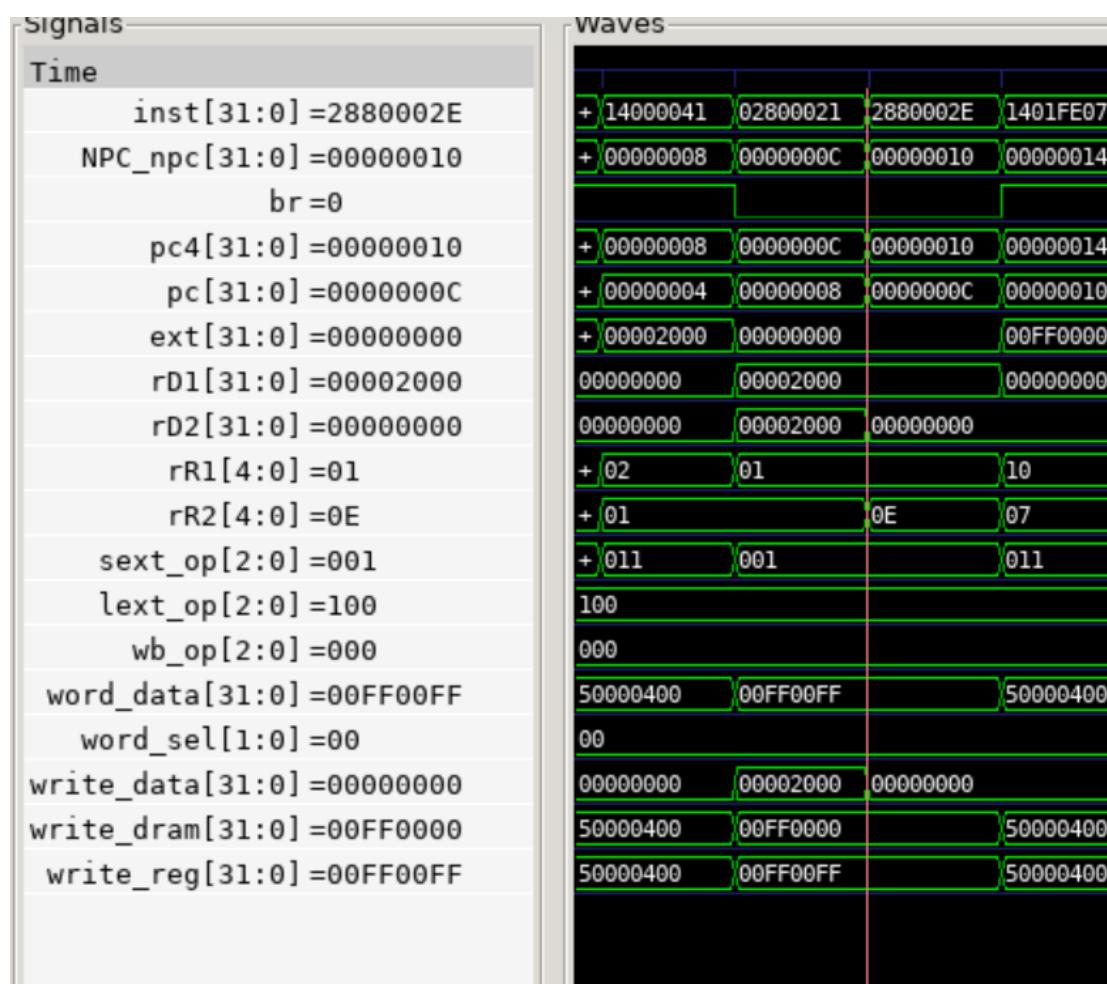
- 我们检验一下 SEXT 生成的 offset 对不对:

sext_op[2:0] =100	001	100	001
ext[31:0] =FFFFFFE4	00000+	FFFFFFE4	000000
din[25:0] =3FFE485	28008+	3FFE485	280000

sext_op = 100, 表示解析 2RI16, 输入指令后 26 位为 11 1111 1111 1110 0100 1000 0101, 符合指令原貌, 提取[25:10]作符号拓展(后面先补两个 0), 提取得 FFF9, 补 0 得 11 1111 1111 1110 0100, 符号拓展后为 FFFFFFFE4, 符合预期, 功能正常;

综上, 分支跳转功能正常;

访存:



当前时间: 220ns

当前指令: 0x2880002E

二进制: 0010 1000 1000 0000 0000 0000 0010 1110

miniLA 指令翻译: ld.w \$x14, \$x1, 0

registers[1][31:0] =00002000	00000000	00002000	
registers[14][31:0] =00000000	00000000		00FF00FF

理论上应该读取 0x00002000 的一个字到 x14;

alu_a[31:0] = 00002000	00000000	00002000	00000000
alu_b[31:0] = 00000000	00000000		
alu_c[31:0] = 00002000	00000000	00002000	00000000
alu_f = 0			
alu_op[3:0] = 0	0		
alua_sel = 0			
alub_sel = 1			

首先 sext_op = 001, 解析 2RI12 指令, 符合预期, 生成立即数为 00000000, 符合预期 (符号拓展);

- rD1 和 ext 将进入 ALU 作为两个操作数, alua_sel=0, alub_sel=1, 对操作数 A 选择 rD1, 对操作数 B 选择 ext 而非 rD2; 可见 alu_a = 00002000, alu_b = 0; alu_op = ALU_ADD(即 0000), 符合预期; 输出 alu_c = 00002000, 符合预期; alu_c 作为目标地址对主存进行访存, 可知读出数据 word_data = 0x00ff00ff, 在这之后写入了 x14, 符合预期, 功能正常;

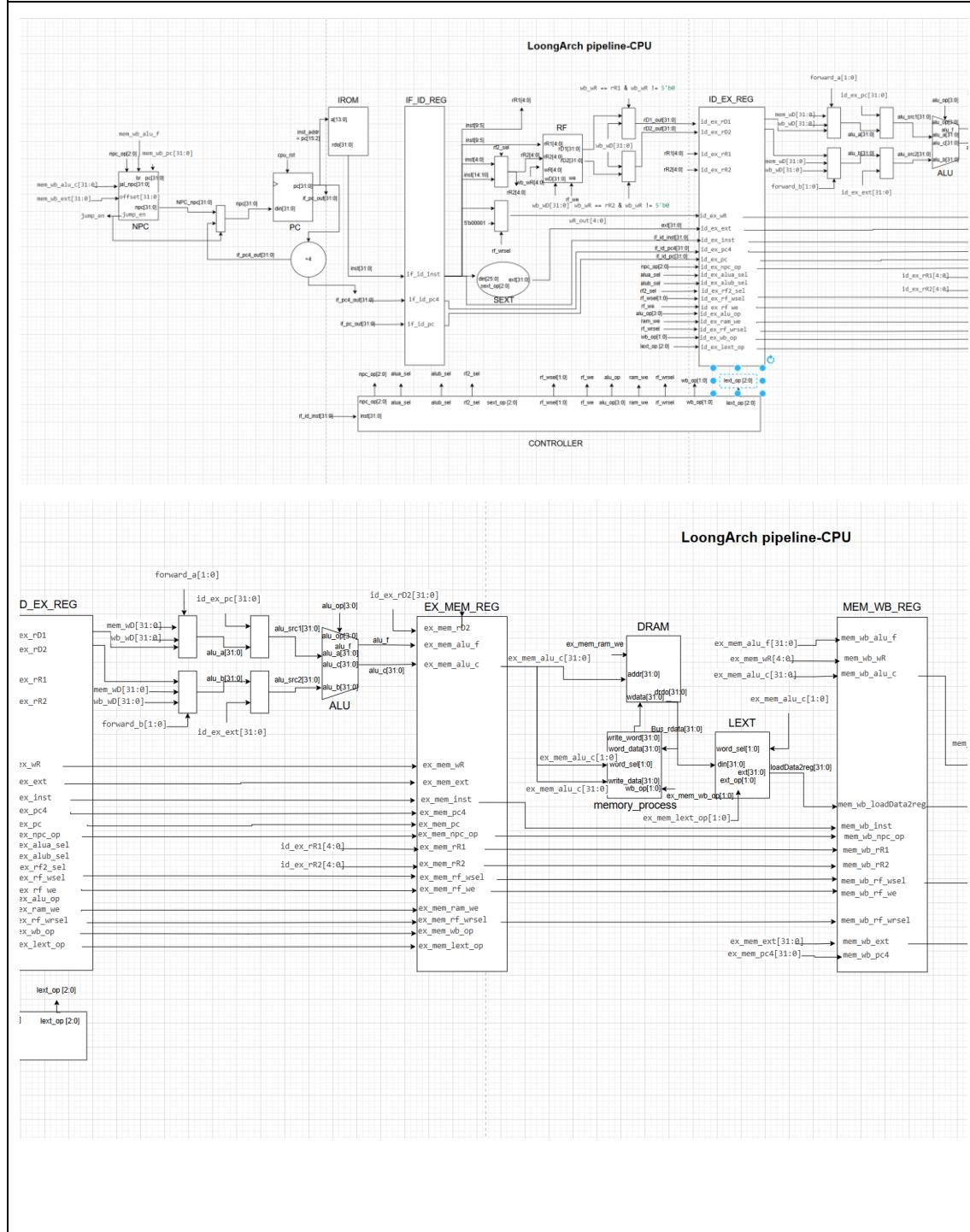
aluc[31:0] = 00002000	00000000	00002000	00
drdo[31:0] = 00FF00FF	50000400	00FF00FF	50
ext[31:0] = 00000000	+ 00002000	00000000	00
inst[31:0] = 2880002E	+ 14000041	02800021	2880002E 14
pc4[31:0] = 00000010	+ 00000008	0000000C	00000010 00
rf_wrsel = 0			
rf_wsel[1:0] = 01	+ 10	00	01 10
wD[31:0] = 00FF00FF	+ 00002000	00FF00FF	00FF00FF
wR[4:0] = 0E	+ 01	0E	07

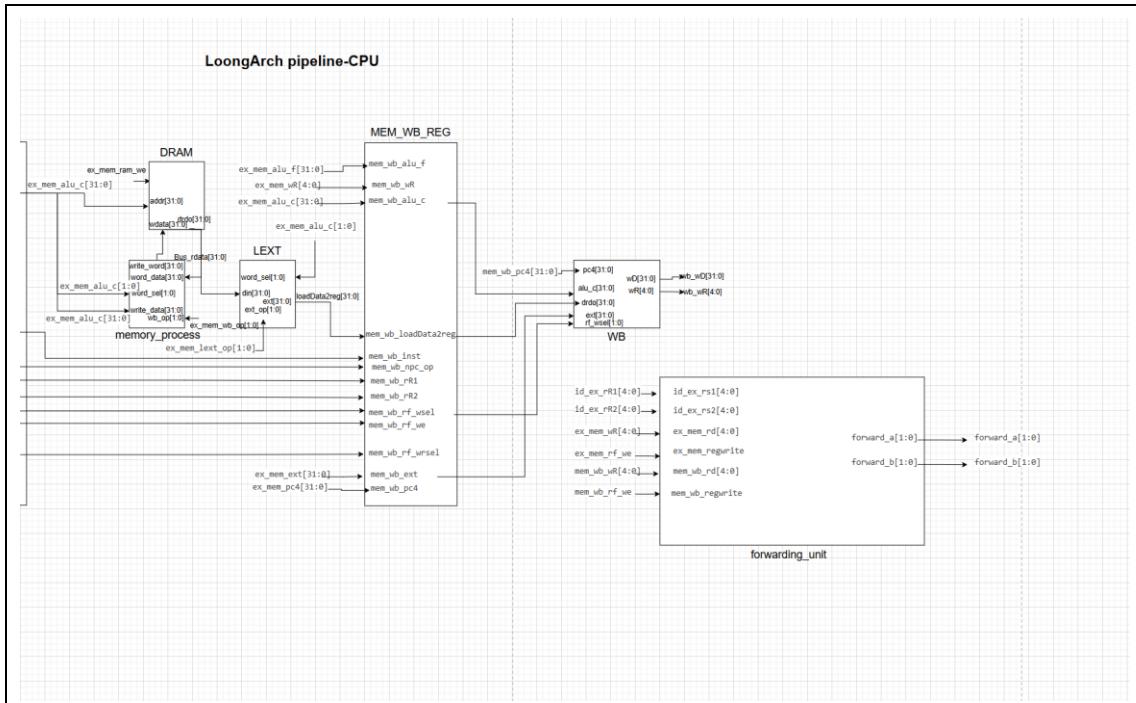
再观察写回模块, 可知写入寄存器数据 wD = 00FF00FF, 符合预期, 选择了 word_data;

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。





描述功能：

模块定义和功能：

- **NPC:** 用于计算下一条要执行的指令地址，传输给 PC；注意：这和单周期不同，他计算的是 wb 阶段指令的 npc，在输入 pc 前，会执行跳转判断，输出 jump_en，jump_en=1 会让所有流水级归零，然后将 npc 的输出输入 pc，否则 $pc=pc+4$ （静态分支预测处理控制冒险）
- **PC:** 时序部件，一个 32 位的地址寄存器，记录当前指令的地址，npc 接口用于输入下一个执行的指令，并在下一个时钟上升沿更新 PC；
- **IROM:** 指令只读存储器，用于存储程序指令，通过 PC 此刻存储的指令地址给出对应的执行指令 inst；
- **RF:** 时序部件，寄存器堆，存储 32 个 32 位寄存器的数据，时钟上升沿写入选中的寄存器的数据，其中第 1 个寄存器 x0 接地，为 zero；
- **ALU:** 运算单元，用于对两个输入操作数执行加法、减法、乘法、除法、与、或、非、逻辑移位（左右）、算术移位（右）、有符号比较、无符号比较运算，并输出运算结果和标志位（判 0 或第一位，根据具体运算而定）；
- **DRAM:** 数据随机存取存储器，用于写入和读出数据，存储空间大，起到主存作用，每次读出和写入均以一个字（32-bit）为单位；
- **SEXT:**（指令）符号拓展单元。用于根据指令低 26 位（miniLA 指令集分布）生成 32 位立即数（有符号/无符号，miniLA 中一共五种立即数生成类型），传输到 EX 部分的 ALU 模块参与运算或 WB 部分写入目标寄存器（视具体指令）；
- **LEXT:**（数据加载 Load）符号拓展单元。根据指令取得所读取字的切片，并做符号拓展（五种），形成写入寄存器的 32 位数据；
- **memory_process:** 面向 DRAM 和 LEXT 的主存读取和写入数据综合处理器。可以将需要写入的数据嫁接入数据所在的字（写一个字节（有无符号）、半个字（有无符号）、一个字五种），再生成需要写入主存的字，将主存各种写入指令整合逻辑与主存分离；
- **WB:** 写回模块，根据 controller 的控制信号 rf_wsel[1:0] 处理四种可能写入寄

存器的数据，形成写回的数据到 RF 模块，完成寄存器的写回操作；

- **controller:** 控制器模块，根据输入的指令生成各个模块的控制信号，指导各个模块完成指令需要的工作；
- 与此同时，在 RF 的 rR2 和 wR 接口前，在 ALU 两个操作数接口前，均设有多路选择器（二选一）分别通过 rf2_sel、rf_wrsel、alua_sel、alub_sel 控制，前二者用于不同指令分割指令的不同部分作为 RF 模块输入，后二者用于在 RF 读出数据和 pc、立即数数据之间选择成为 ALU 的操作数 A 和 B；

流水线的特殊功能：

可以看见在 ALU 前方增加了两个多路选择器，这是为了将传统的 ALU 当前命令读取数据与前递数据作取舍，通过 forward_a/b 进行判断选择；

这两个信号通过 forwarding_unit 产生：

- **forwarding_unit:** 输入 ex 阶段、mem 阶段和 wb 阶段的流水线寄存器信号，判断是否有数据冒险，如果有，则生成前递信号 forward_a 和 forward_b；

在 ID 模块，为了实现先读后写，假如写入寄存器和当前需要读的寄存器相同，则将写入数据覆盖到读出数据，下一个周期再更新寄存器数据；

流水线级划分：

IF: 包括 NPC 和 PC 和 PC+4，实现指令地址的更新和分支跳转的处理，获取指令；

ID: 包括 SEXT、controller、RF，实现指令的解码，生成控制信号和读取寄存器数据；

EX: 包括 ALU 和前递后处理模块，实现运算功能

MEM: 包括 DRAM、memory_process 和 LEXT 模块，实现对写入数据字拼接处理、从主存读数据、从主存写数据、对主存读的数据进行分离和拓展、加工；

WB: 确定写入寄存器的编号和数据，并处理前递，写入寄存器；（包括 RF 一部分）

对于每一级流水线寄存器，需要实现的功能有更新和重置，当 jump_en=1 时需要重置寄存器的值，每个时钟周期更新寄存器为上一级数据；

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。

流水线寄存器部分：

IF/ID	if_id_pc	32 位
	if_id_pc4	32 位
	if_id_inst	32 位
ID/EX	id_ex_rf_wsel	2 位
	id_ex_rf_we	1 位
	id_ex_rf2_sel	1 位
	id_ex_rf_wrsel	1 位
	id_ex_sext_op	3 位
	id_ex_lext_op	3 位
	id_ex_alu_op	4 位
	id_ex_npc_op	3 位
	id_ex_wb_op	2 位
	id_ex_alua_sel	1 位
	id_ex_alub_sel	1 位
	id_ex_ram_we	1 位
	id_ex_pc	32 位
	id_ex_pc4	32 位
	id_ex_inst	32 位
	id_ex_rD1	32 位
	id_ex_rD2	32 位
	id_ex_ext	32 位
	id_ex_rR1	5 位
	id_ex_rR2	5 位
	id_ex_wR	5 位
EX/MEM	ex_mem_rf_wsel	2 位
	ex_mem_rf_we	1 位
	ex_mem_rf_wrsel	1 位
	ex_mem_sext_op	3 位
	ex_mem_lext_op	3 位
	ex_mem_alu_op	4 位

	ex_mem_npc_op	3 位	
	ex_mem_wb_op	2 位	
	ex_mem_alua_sel	1 位	
	ex_mem_alub_sel	1 位	
	ex_mem_ram_we	1 位	
	ex_mem_pc	32 位	
	ex_mem_pc4	32 位	
	ex_mem_inst	32 位	
	ex_mem_alu_c	32 位	
	ex_mem_alu_f	32 位	
	ex_mem_ext	32 位	
	ex_mem_rD2	32 位	
	ex_mem_rR1	5 位	
	ex_mem_rR2	5 位	
	ex_mem_wR	5 位	
MEM/WB	mem_wb_rf_wsel	2 位	
	mem_wb_rf_we	1 位	
	mem_wb_rf_wrsel	1 位	
	mem_wb_sext_op	3 位	
	mem_wb_lext_op	3 位	
	mem_wb_alu_op	4 位	
	mem_wb_npc_op	3 位	
	mem_wb_wb_op	2 位	
	mem_wb_alua_sel	1 位	
	mem_wb_alub_sel	1 位	
	mem_wb_ram_we	1 位	

	mem_wb_pc	32 位
	mem_wb_pc4	32 位
	mem_wb_inst	32 位
	mem_wb_ext	32 位
	mem_wb_alu_c	32 位
	mem_wb_alu_f	32 位
	mem_wb_loadData2reg	32 位
	mem_wb_rR1	5 位
	mem_wb_rR2	5 位
	mem_wb_wR	5 位
	mem_wb_wD	32 位

其他部分：

forwarding_unit:

信号名称	方向	位宽
id_ex_rs1	输入	5 位
id_ex_rs2	输入	5 位
ex_mem_rd	输入	5 位
ex_mem_regwrite	输入	1 位
mem_wb_rd	输入	5 位
mem_wb_regwrite	输入	1 位
forward_a	输出	2 位
forward_b	输出	2 位

NPC:

接口类型	接口信号	位宽	功能描述
input wire	pc	32	当前指令地址
input wire	offset	32	跳转的指令偏移量
input wire	br	1	跳转校验位，来自 ALU 标志位输出
input wire	jal_npc	32	jirl 指令的无条件跳转指令地址（通过 ALU 算出）
input wire	npc_op	3	NPC 模块控制信号
output reg	npc	32	输出下一条执行指令的地址
output reg	pc4	32	输出当前指令的下一条顺位

			指令地址 (PC+4)
output reg	jump_en	1	是否跳转

关键设计的描述：

所有流水线寄存器：

核心代码：

```
if (cpu_RST | jump_en) begin
end else begin end
```

在每个时钟上升沿，检测是否跳转，如果 jump_en = 1，则清空所有寄存器，实现静态分支预测错误的处理；

forwarding_unit：

关键代码：

```
always @(*) begin
    // Default forwarding controls
    forward_a = 2'b00;
    forward_b = 2'b00;

    // EX hazard
    if (ex_mem_regwrite && (ex_mem_rd != 0) && (ex_mem_rd == id_ex_rs1)) begin
        forward_a = 2'b10;
    end
    if (ex_mem_regwrite && (ex_mem_rd != 0) && (ex_mem_rd == id_ex_rs2)) begin
        forward_b = 2'b10;
    end

    // MEM hazard
    if (mem_wb_regwrite && (mem_wb_rd != 0) && (mem_wb_rd == id_ex_rs1) &&
        !(ex_mem_regwrite && (ex_mem_rd != 0) && (ex_mem_rd == id_ex_rs1))) begin
        forward_a = 2'b01;
    end
    if (mem_wb_regwrite && (mem_wb_rd != 0) && (mem_wb_rd == id_ex_rs2) &&
        !(ex_mem_regwrite && (ex_mem_rd != 0) && (ex_mem_rd == id_ex_rs2))) begin
        forward_b = 2'b01;
    end
end
```

设计解释：

- 关键是判断在 ex 阶段的需要使用的寄存器数据是否与 mem 和 wb 阶段的写寄存器编号相同，若相同，则有数据冒险，此时要考虑用前递处理；
- forward_a 和 forward_b 分别代表 rs1 和 rs2 的前递信号，假如为 00 则没有前递，01 代表 wb 前递，10 代表 mem 前递；
- 前递采用新数据优先的原则，假如 mem 阶段有前递需求，则不考虑 wb 阶段的前递，这在第二段代码可以见到；

NPC 和 IF 的关键设计：

```

    always @(*) begin
        case (npc_op)
            `NPC_PC4: jump_en = 1'b0;
            `NPC_BEQ: jump_en = (br == 1);
            `NPC_BNE: jump_en = (br == 0);
            `NPC_JAL: jump_en = 1'b1;
            `NPC_BL:  jump_en = 1'b1;
            default: jump_en = 1'b0; // Default case to handle undefined npc_op values
        endcase
    end

```

这是 NPC 的代码，简单地将前面确定 npc 的代码重复利用在 jump_en，在需要跳转时输出 jump_en=1，平时默认不跳转，为 0；

对于 IF：

```
assign npc = jump_en ? NPC_npc : pc4;
```

npc 是给到 PC 的 din 的，默认不跳转，当跳转时将 npc 设置为 NPC 特殊的跳转；

- RF 的后处理有改变：

```

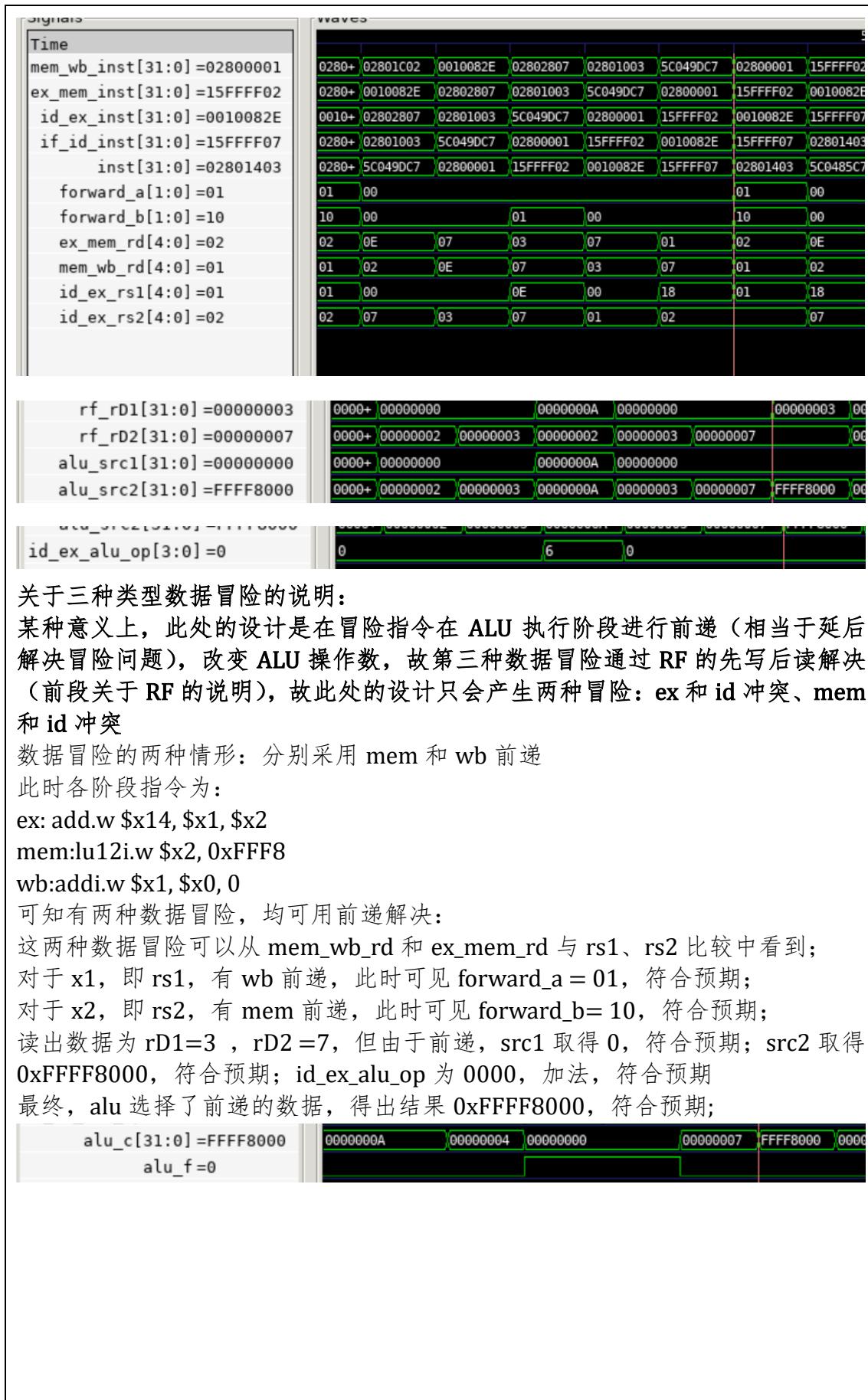
assign rD1_out = (wb_wR == rR1 & wb_wR != 5'b0) ? wb_wD : rD1;
assign rD2_out = (wb_wR == rR2 & wb_wR != 5'b0) ? wb_wD : rD2;

```

此处实现了先写后读（先把将要写的数据读出，再实际上写入寄存器，宏观上实现了先写后读），此时本来就需要把写回的部分传到 ID 模块内，一举两得，顺便解决了数据冒险的第三种：wb 和 id 冲突

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。



在此也简单说一下第三种数据冒险的实际分析处理：

Time	700 ps					
mem_wb_inst[31:0] = 14000107	02BF+	0010082E	14000107	02BFFCE7	02802003	5C0435C7 15000001
ex_mem_inst[31:0] = 02BFFCE7	0010+	14000107	02BFFCE7	02802003	5C0435C7 15000001	02BFFC21
id_ex_inst[31:0] = 02802003	1400+	02BFFCE7	02802003	5C0435C7 15000001	02BFFC21	02800002
if_id_inst[31:0] = 5C0435C7	02BF+	02802003	5C0435C7 15000001	02BFFC21	02800002	0010082E
inst[31:0] = 15000001	0280+	5C0435C7 15000001	02BFFC21	02800002	0010082E	15000007
forward_a[1:0] = 00	00	10	00	10	00	
forward_b[1:0] = 00	00	10	01	00	10	00
ex_mem_rd[4:0] = 07	0E	07	03	07	01	
mem_wb_rd[4:0] = 07	02	0E	07	03	07	01
rR1[4:0] = 0E	07	00	0E	00	01	00
rR2[4:0] = 07	07	03	07	01	02	
id_ex_rsl[4:0] = 00	08	07	00	0E	00	01
id_ex_rs2[4:0] = 03	07	03	07	01	02	
rf_rD1[31:0] = 00000000	0000+	7FFF8000	00000000	00007FFF	00000000	
rf_rD2[31:0] = 00000007	7FFF8000	00000007	00008000	00000000	00007FFF	
alu_src1[31:0] = 00000000	0000+	00008000	00000000	00007FFF	00000000	80000000 00000000
alu_src2[31:0] = 00000007	7FFF+	00008000	00000007	00007FFF	00000000	80000000 00007FFF
id_ex_alu_op[3:0] = 0	0		6	0		
alu_c[31:0] = 00000008	7FFF+	00007FFF	00000008	00000000	7FFFFFFF	00000000
alu_f = 0						
wb_wD[31:0] = 00008000	00007FFF	00008000	00007FFF	00000008	00000000	80000000
wb_wR[4:0] = 07	02	0E	07	03	07	01
rD2_out[31:0] = 00008000	7FFF+	00000007	00008000	00000000	00007FFF	
registers[7][31:0] = 7FFF8000	7FFF8000		00008000	00007FFF		

可以看到，此处 id 阶段指令为：0x5C0435C7

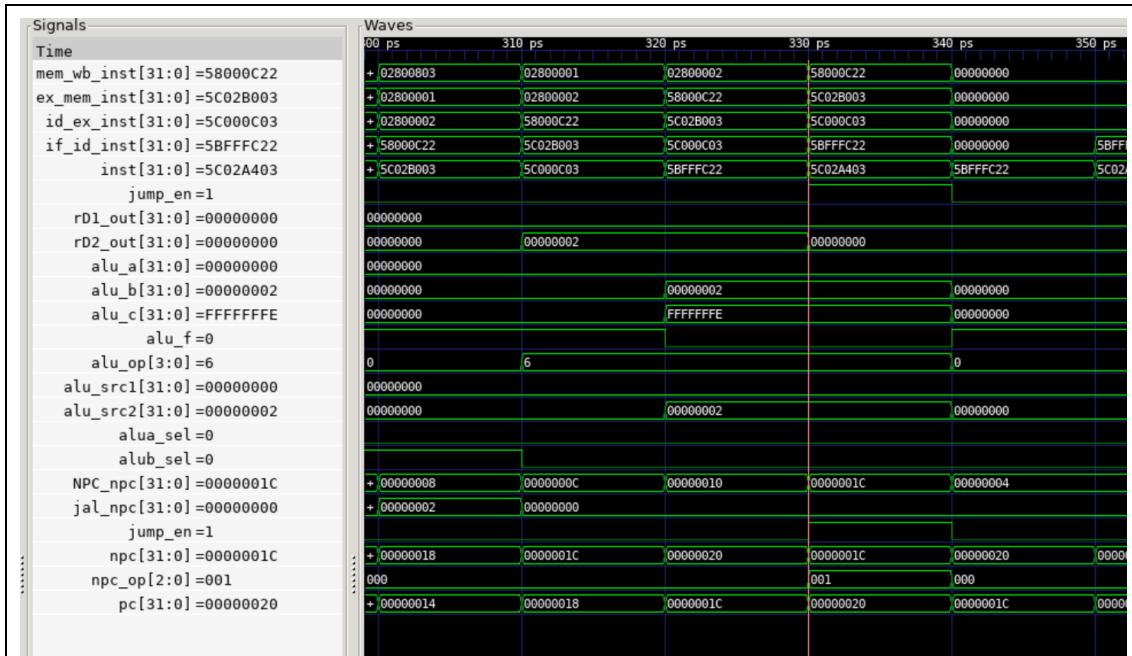
0101 1100 0000 0100 0011 0101 1100 0111

bne \$x14, \$x7, 0x010D

此时 rR2 = 7；但是此时 wR = 7 = rR2，根据 RF 的设计，是先读后写，此处就构成了第三种数据冒险，但是我们有：

```
assign rD1_out = (wb_wR == rR1 & wb_wR != 5'b0) ? wb_wD : rD1;
assign rD2_out = (wb_wR == rR2 & wb_wR != 5'b0) ? wb_wD : rD2;
```

故可以看见 rD2_out 取出的并不是此时 x7 的值，而是 wb_wD 的值，也就是 0x00008000 而非 0x00007FFF，而且在进入 ex 阶段后才将 x7 的值更新为 0x00008000，可见这种“前递”解决了第三种数据冒险，符合预期（但实际上在这种设计中构不成冒险）；



控制冒险：

此时 wb 指令: 0x580000C22

即: 0101 1000 0000 0000 0000 1100 0010 0010

即: beq \$x1, \$x2, 0x3

- 包含所有的前递影响，在两个周期前，应是此指令经过 EX 阶段，此时有 $\text{src1} = \text{src2} = 0$ ，会跳转，直到 WB 阶段跳转，此时 $\text{jump_en} = 1$ 符合预期；在下一个周期，可见所有流水线寄存器的指令都被清空，之后新加载了指令 5BFFC22，符合预期；
- 观察 NPC 内信号，可知 $\text{mem_wb_alu_f} = 1$ ，结合 $\text{npc_op} = 001$ ，可知 npc 为 NPC_BEQ 模式， $\text{br} = 1$ 时跳转，符合预期， jump_en 拉到高电平，此时， $\text{npc} = \text{NPC_npc}(0000001CH)$ 而不是 $\text{pc}+4(00000024H)$ ，控制冒险处理功能正常；



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。

单周期:

- 在编写单周期 cpu 时,我碰到访存的错误,波形十分难查,最后是通过反复查看波形前半段,将前面一段的指令导出,然后分析此时应该读出什么(最难搞的一个就是字的切片,这个折磨了我很久),最终让我学会了如何去根据波形来判断访存的错误,尤其是切片和字不匹配的错误,最终发现是切片模块写错了个判断条件;
- 可以说这极大的提升了我的波形分析能力,尤其是碰到一些访存(根本不知道主存里有什么)的验证,而且也充分体会到了模块设计时单元测试的重要性,每个模块如果按照每个单元都编写一个仿真程序,可能就不会出现这么愚蠢的问题;

流水线:

- 在编写流水线 cpu 时,这位更是比单周期难调,但是经过单周期的功能验证,各个小模块出问题的概率并不大;相比而言,出问题的大多是前递模块。前递模块我编写了单元测试,模块内部是不会出问题的,但是 trace 就是过不了,于是开始找原因;
- 我分析了这条不能通过的指令的前后指令,发现这就是前递出的问题,但是我的 forwarding_unit 不应该出事,百思不得其解;
- 于是没有思路,我便下去吃了个饭,在路上,我用手机翻找理论课的 ppt,试图寻找一些灵感,其实晚上在外边十分有助于灵感迸发,这是一个可以注意的点(以后想不出来就下去逛),发现是我沿用了理论课的思路,就是 RF 是先写后读,仔细思考我写的 RF 的时序,更新得在这个周期之后,不符合先写后读,那么这个前递就可能失效了,因为可能要处理 st.w 这些指令的前递,而我的 ALU 又没有写直接把操作数放到输出结果这个选项(我认为这不优雅),于是就使用了 cache 的思路,假如 rR2 读的数据是需要前递的,那么就直接把输出接到前递,间接实现了先写后读;
- 这某种意义上再次体现了做好单元测试的重要性,RF 这部分的单元测试的这个问题并没有在单周期出现(因为不需要),但是会在流水线中爆发,以后设计时一定要做好充分考虑,做好单元测试,理论和实践需要对齐每一个细节。

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。

学完这门课程后，我有以下收获和体会：

- 理论与实践的结合:**通过设计单周期和流水线的 CPU，我不仅加深了对计算机体系结构理论的理解，还在实际操作中掌握了硬件设计和验证的基本技能。这让我认识到理论知识只有在实践中才能真正内化。
- 解决问题的能力:**在项目开发过程中，我遇到过许多技术难题，例如数据通路中的冲突和数据冒险问题。通过不断调试和优化，我学会了如何系统地分析问题并寻找解决方案，这对我以后处理复杂的工程问题非常有帮助。
- 代码优化的重要性:**在实现 CPU 的过程中，我逐渐意识到代码的优化对性能的影响是巨大的。在流水线 CPU 设计中，如何减少冒险、提高执行效率，都是通过优化代码来实现的。这让我在以后的代码编写中更加注重效率和规范。

对本课程的建议和意见：

- 加强实践指导:**虽然课程中涉及了很多实际操作，但在一些关键问题的解决上，可能会遇到较大困难。希望课程能增加一些针对复杂问题的指导，例如提供更多的调试案例或解决方案参考。
- 增加课后讨论:**建议增加一些课后讨论环节，方便同学们交流心得体会，分享遇到的问题和解决方案。这不仅能促进学习，也有助于提升团队合作能力。
- 更新教学材料:**课程的部分教学材料可能需要更新，以跟上技术发展的步伐。例如，加入更多关于现代处理器设计的新技术和新趋势的介绍，能够让学生更好地理解当前行业的动态。

总体来说，这门课程对我提升硬件设计能力和解决复杂问题的能力有很大的帮助，我非常感谢老师的指导和课程的设置。