体系结构第一次实验报告

姓名：梁朝阳　**专业**：密码科学与技术

目录

摘要

CPU 结构部分: 我打算从相关性的级联控制开始谈我的理解，之后 CPU 最复杂的地方莫过于控制信号的设计，这部分内容我需要阅读 decode.v 文件来理解。还有就是我会讲下我对 CP0 寄存器的理解。最后我会说下我对延迟槽机制实现的理解，并做出我认为当前 CPU 的运行没有错误的判断。（当然 IP 核的配置不是按上学期照操作手册上描述的那样，需要取消 Primitive register 的使用）。

新增指令部分：我会逐一介绍新增的 9 条指令，大多数都是关于 CP0 寄存器的操作。还有一部分是乘法器指令有关的指令。

关键词：流水线 CPU，CP0 寄存器，延迟槽，乘法器

1 实验目的

1. 熟悉并掌握流水线 CPU 的原理和设计。
2. 检验运用 verilog 语言进行电路设计的能力。
3. 通过亲自设计实现静态 5 级流水线 CPU，加深对计算机组成原理和体系结构理论知识的理解。

2 实验要求

1. 修改原始代码 source code 中的 bug，保证指令运行正确
2. 针对五级流水线中新增加的指令，逐级分析指令执行过程和执行结果，梳理流水线知识点

3 实验过程

3.1 熟悉代码 |CPU 结构部分

3.1.1 相关性的级联控制

最重要的肯定是级联控制，与理论课上从相关性检测单元来控制的方法不同，这里的级联控制是基于两个阶段的握手信号来实现的：

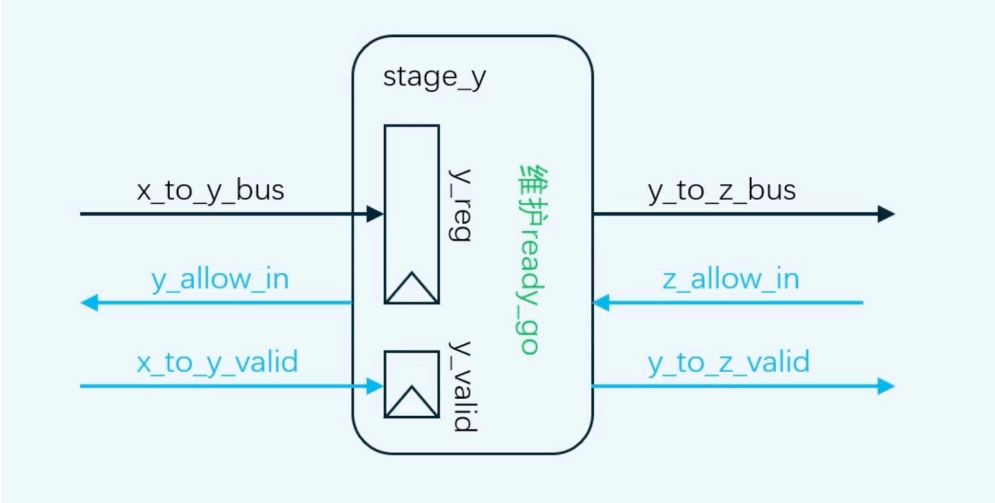

图 3.1: 相关性的级联控制

以最一般的 stage_y 举例，这里一共有这 7 个信号：

1. `x_to_y_bus`：x 阶段到 y 阶段的总线
2. `y_allow_in`：（本实验代码中实现为 `y_over` 信号）y 阶段允许 x 阶段的数据进入
3. `x_to_y_valid`：x 阶段到 y 阶段的数据有效（即 `x_to_y_bus` 有效）
4. `y_to_z_bus`：y 阶段到 z 阶段的总线
5. `z_allow_in`：（本实验代码中实现为 `z_over` 信号）z 阶段允许 y 阶段的数据进入

6. `y_to_z_valid`：y 阶段到 z 阶段的数据有效（即 `y_to_z_bus` 有效）

7. `ready_go`：是否为 1 取决于 y 阶段是否完成，这个在每个阶段的具体实现不同，例如本实验在乘法器的地方叫做 `mult_end`，在存储的时候比较复杂，没有给出显式的 ready_go 信号，而是给出了下面这个代码，这里的 `inst_load ? MEM_valid_r : MEM_valid` 实际上就是 ready_go 信号，从其能够影响（控制）over 信号可以看出，其中 over 本质上就是 to_next_stage 信号.

```
1       assign MEM_over = inst_load ? MEM_valid_r : MEM_valid;
```

以及如下两个寄存器：

1. `y_reg`：y 阶段的数据寄存器，用于接受 `x_to_y_bus` 的数据

2. `y_valid`：这个信号比较重要，具体地：

   (a) x 阶段的数据有效且 y 阶段可以接收数据，则为 1

   (b) x 阶段数据无效且 y 阶段可以接收数据，则为 0（这里要注意，有些运算并不能够一周期完成，因此如果 y 不可接受数据，那么 _valid 应当保持不变）

   (c) rest 或者 cancel 的时候为 0

   那么我总结一下这个信号本身意思是标记当前阶段数据是否有效，为 1 的情况有两种，一种是当前数据有效正在进行多周期运算/被阻塞，另一种是当前数据有效且下一个周期就要走了，这个时候 allow_in 也为 1，那么下一个周期仍然是有效的（需要赋值为 1）。

   具体地，这个信号这么实现：

```
always @(posedge clk) begin
  if (reset) begin
    y_valid <= 1'b0;
  end else if (y_allow_in && x_to_y_valid) begin
    y_valid <= 1'b1;
  end else if (y_allow_in && !x_to_y_valid) begin
    y_valid <= 1'b0;
  end
end
```

图 3.2: 级联控制 valid 信号

其中可以简化实现为将 y_valid 直接赋值为 x_to_y_valid，当 y_allow_in 为 1 的时候。

其余的信号也简单说下是怎么实现的，这里直接给出实验中的 CPU 的代码，我们这里统一使用 EXE 阶段的信号举例：

EXE_valid 这里说下实验中是如何实现 valid 信号的，以 EXE_valid 为例：

```
1   //EXE_valid
2   always @(posedge clk)
3   begin
4       if (!resetn || cancel)
```

```
        begin
            EXE_valid <= 1'b0;
        end
        else if (EXE_allow_in)
        begin
            EXE_valid <= ID_over;
        end
    end
```

这里是 ID 对 EXE 实现握手，ID_over 就是 ID_allow_in。这里的行为完全符合我刚才对图3.2的分析。

EXE_allow_in 还有 allow_in 信号，这个信号用于告诉前一个阶段自己是否可以接收数据：

```
    assign EXE_allow_in = ~EXE_valid | (EXE_over & MEM_allow_in);
```

可以看到，当本阶段没有有效数据（为空）的时候肯定是可以接受数据的。或者下一个阶段我这个阶段的数据就要走了，那么也可以接受。反之 stall。

锁存（总线传输） 当我这个阶段结束，且下一个阶段允许我流出的时候，我就可以把上一个给我的信号存到 reg 寄存器中。注意，这里我们采用的是永远让后一个 stage 维护 reg 的实现方式。

```
    always @(posedge clk)
    begin
        if(EXE_over && MEM_allow_in)
        begin
            EXE_MEM_bus_r <= EXE_MEM_bus;
        end
    end
```

到这里我认为级联控制部分就介绍完了。

3.1.2 decoder 的控制信号的设计

控制信号设计较为困难，我们分成多个小模块讲解：

解码 解码操作是将指令分解为各个部分，例如操作码、源操作数、目标操作数等。

```
    assign op   = inst[31:26]; // 操作码
    assign rs   = inst[25:21]; // 源操作数1
    ... \\ 手动省略几个，水字数没意义，就是把32位地址拆开了
    assign target = inst[25:0]; // 目标地址
    assign cp0r_sel= inst[2:0]; // cp0寄存器的select域
```

这里是解码器的一部分，用于把 inst 分解为各个部分。

接着是一对的 assign 语句，把满足某个指令条件的信号赋值为 1，举一个最简单的例子：

```
assign inst_MULT = op_zero & (rd==5'd0) & sa_zero & (funct == 6'b011000); //乘法
```

这个的意思是：当操作码为 0，特殊域为 0，功能码为 011000 的时候，inst_MULT 为 1。

ALU 的控制信号 接着是对 ALU 专门的信号控制，采用的是独热编码：

```
assign alu_control = {inst_add,
inst_sub,
inst_slt,
inst_sltu,
... \\ 这里我手动省略几个，总之就是打包成一个总线给ALU
inst_srl,
inst_sra,
inst_lui};
```

还有一个是 ALU 的两个操作数：

```
assign alu_operand1 = inst_j_link ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_j_link ? 32'd8 :
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ? {{16{imm[15]}}, imm} : rt_value;
```

这里为什么说一下，是因为实现旁路的时候这里的后面是需要动一下的。例如加两个 MUX：

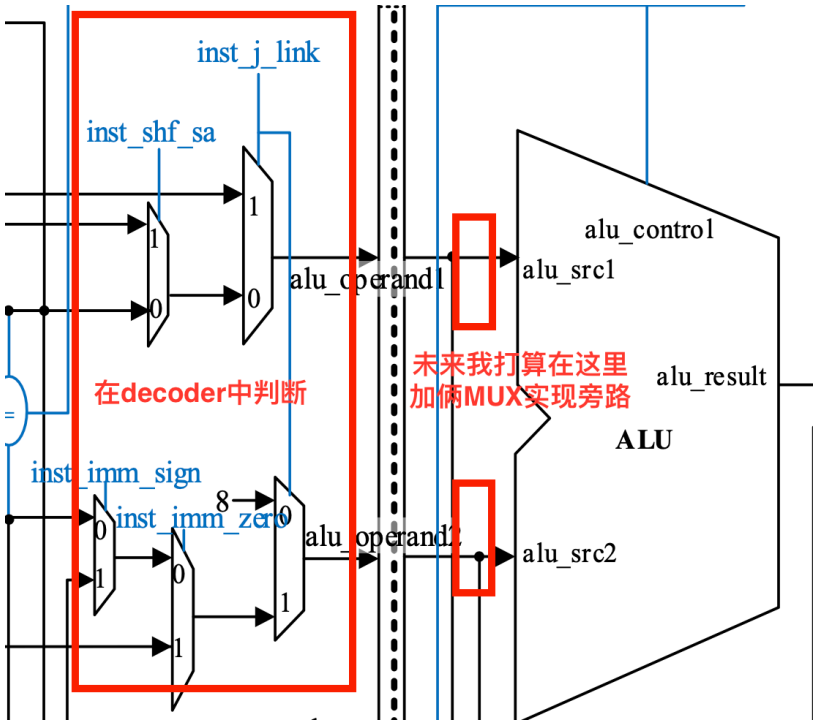

图 3.3: ALU 操作数

3.1.3 CP0 寄存器的理解（顺带 HI 和 LO 寄存器）

到了 CP0 寄存器，这个 CPO 是 Coprocessor 0 的缩写，表示协处理器 0。这个协处理器 0 是 MIPS 体系结构中的一个协处理器，用于实现一些特殊的功能，例如异常处理、中断处理等。

而 HI 和 LO 寄存器是用于存放乘法结果的，我打算单独一个地方说乘法的走线。而且在我的浅薄认知来看，一旦涉及这种某一个阶段可以跑多个周期，还是级联控制要简单很多。

WB 阶段的控制信号 要分析协处理器，就要看 WB 阶段：

```
module wb(                     // 写回级
    input         WB_valid,    // 写回级有效
    input [117:0] MEM_WB_bus_r, // MEM->WB总线
    output        rf_wen,      // 寄存器写使能
    output [ 4:0] rf_wdest,    // 寄存器写地址
    output [ 31:0] rf_wdata, // 寄存器写数据
    output        WB_over,     // WB模块执行完成

    // 省略常规信号如clk、resetn等
     output [ 32:0] exc_bus,  // Exception pc总线
     output [ 4:0] WB_wdest,  // WB级要写回寄存器堆的目标地址号
     output        cancel,     //
        syscall和eret到达写回级时会发出cancel信号，取消已经取出的正在其他流水级执行的指令
    //省略展示信号
);
```

上述代码片段中，主要涉及写回阶段寄存器堆相关的输出信号说明如下：

1. `rf_wen`：写回阶段寄存器堆写使能信号，高电平时表示允许写寄存器。
2. `rf_wdest`：写回阶段寄存器堆写地址，指定要写入的目标寄存器编号。
3. `rf_wdata`：写回阶段写入寄存器堆的数据。

这些信号共同完成了指令执行结果向寄存器堆的写回操作，是 WB（写回）阶段的核心输出。

CP0 寄存器的控制信号 由于目前设计的 CPU 并不完备，所用到的 cp0 寄存器也很少，故暂时只实现 STATUS(12.0),CAUSE(13.0),EPC(14.0) 这三个。

其中 mtc0 (move to coprocessor 0) 和 mfc0 (move from coprocessor 0) 是用于向 CP0 寄存器写入和读取数据，mfhi (move from hi) 和 mflo (move from lo) 是用于从 HI 和 LO 寄存器读取数据。

```
    assign status_wen = mtc0 & (cp0r_addr=={5'd12,3'd0});
    assign epc_wen  = mtc0 & (cp0r_addr=={5'd14,3'd0});
```

其中 5’d12,3’d0 的含义是：5’d12 表示 STATUS 寄存器，3’d0 表示 STATUS 寄存器的第 0 位。这里的编号（地址）是自己设计的。

CP0 寄存器读取　寄存器读取实现就是一个 MUX，根据地址选择读取哪个寄存器。说实话，我并不是很喜欢这种嵌套的实现，感觉不如一个大寄存器清晰，但是我猜测可能这样实现起来映射到硬件上更简单，毕竟这个是一堆的两个入口的 MUX。

```
assign cp0r_rdata = (cp0r_addr=={5'd12,3'd0}) ? cp0r_status :
                    (cp0r_addr=={5'd13,3'd0}) ? cp0r_cause :
                    (cp0r_addr=={5'd14,3'd0}) ? cp0r_epc : 32'd0;
```

但是，设计图中画的是一个大的 MUX，而代码是先判断两个，然后接着判断后面的：

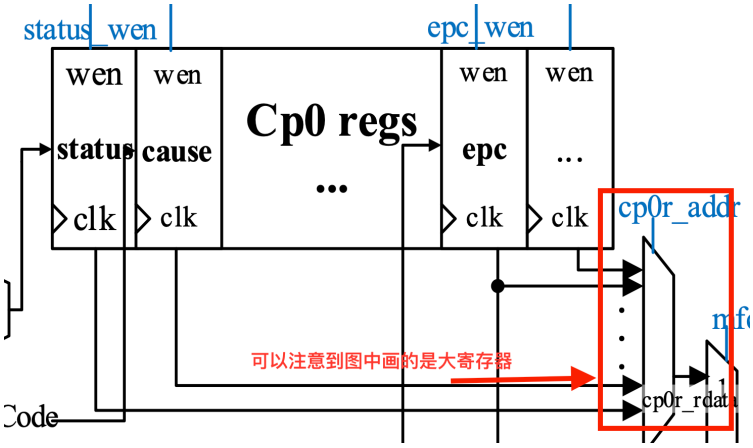

图 3.4: CP0 地址 MUX

STATUS 寄存器　STATUS 寄存器是用于存放异常状态的寄存器，其中 EXL（exception level）域是软件可读写的，故需要 status_wen 信号。

要理解这个寄存器，重要的是知道 syscall 指令执行的时候发生了什么。因此，有必要单开一个小的章节来说明 syscall 指令执行的时候发生了什么。

3.1.4　syscall 指令执行流程

SYSCALL 指令在译码阶段被识别，并通过流水线总线传播到写回阶段：

Listing 1: SYSCALL 指令识别

```
// 在decode.v中，SYSCALL指令的识别
assign inst_SYSCALL = (op == 6'b000000) & (funct == 6'b001100);
```

SYSCALL 指令在五级流水线中的执行时序如下：

Table 1: SYSCALL 指令流水线执行时序

周期	IF	ID	EXE	MEM	WB
1	SYSCALL	-	-	-	-
2	下一条指令	SYSCALL	-	-	-
3	下一条指令	下一条指令	SYSCALL	-	-
4	下一条指令	下一条指令	下一条指令	SYSCALL	-
5	下一条指令	下一条指令	下一条指令	下一条指令	SYSCALL
6	异常处理程序	下一条指令	下一条指令	下一条指令	下一条指令

当 SYSCALL 指令到达写回阶段时，执行以下关键操作：

Listing 2: STATUS 寄存器 EXL 域设置

```
always @(posedge clk)
begin
    if (!resetn || eret)
    begin
        status_exl_r <= 1'b0;
    end
    else if (syscall) // 如果遇到syscall指令，则将EXL域设置为1
    begin
        status_exl_r <= 1'b1;
    end
    else if (status_wen)
    begin
        status_exl_r <= mem_result[1];
    end
end
```

同时间，更新 CAUSE 寄存器：

Listing 3: CAUSE 寄存器异常编码设置

```
always @(posedge clk)
begin
    if (syscall)
    begin
        cause_exc_code_r <= 5'd8; // SYSCALL的异常编码为8
    end
end
```

这个寄存器用于存储发生异常的原因，这里设置为 8，即 SYSCALL 的异常编码。
同时更新 EPC 寄存器：

Listing 4: EPC 寄存器保存当前 PC

```
always @(posedge clk)
begin
    if (syscall)
    begin
        epc_r <= pc; // 保存产生异常的PC值
    end
    else if (epc_wen)
    begin
        epc_r <= mem_result;
```

```
    end
end
```

EPC 是 Exception Program Counter 的缩写，用于存储产生异常的 PC 值。
同时，发出 cancel 信号：

Listing 5: cancel 信号发出

```
assign cancel = (syscall | eret) & WB_over;
```

这里有个名词叫做流水线的冲刷。同事补充一下异常入口跳转操作：

Listing 6: 异常 PC 总线

```
wire        exc_valid;
wire [31:0] exc_pc;
assign exc_valid = (syscall | eret) & WB_valid;
assign exc_pc = syscall ? `EXC_ENTER_ADDR : cp0r_epc;
assign exc_bus = {exc_valid, exc_pc}; // 分别代表异常有效和EPC寄存器的值
```

总结一下 SYSCALL 指令执行流程：

1. **周期 1-4**：SYSCALL 指令在流水线中正常传播
2. **周期 5**：SYSCALL 指令到达写回阶段，执行以下操作：
   - STATUS[1] = 1（设置 EXL 域）
   - CAUSE[6:2] = 8（设置异常编码）
   - EPC = 当前 PC（保存返回地址）
   - cancel = 1（冲刷流水线）
   - exc_bus = {1, 0}（跳转到地址 0）
3. **周期 6**：IF 跳转到地址 0，开始执行异常处理程序

和所有的关键寄存器的变化过程：

Table 2: SYSCALL 执行前后关键寄存器状态

寄存器	执行前	执行后
STATUS[1] (EXL)	0	1
CAUSE[6:2] (ExcCode)	0	8
EPC	任意值	产生异常的 PC
PC	产生异常的 PC	0（异常处理入口）

3.1.5 延迟槽机制实现的理解

延迟槽机制在我看来就是编译器对指令排列优化后，然后 CPU 不用在对分支/跳转指令进行延时（stall）。最开始我是从 Vivado 的仿真中看出来这里有问题的：

图 3.5: 延迟槽导致提前执行 4CH 指令

但是经过我观察指令集的排布，我感觉这里并不是错误，这里本身是 48H 指令（其结果是跳转到 84H），那我们看一下到到底跳转过去要干嘛：

84H	lui $12,#12	[$12] = 000C_0000H
88H	srav $26,$12,$2	[$26] = 0000_000CH
8CH	addiu $27,$26,#68	[$27] = 0000_0050H
90H	jalr $27	跳转到 50H, [$31] = 0000_0098H

图 3.6: 84H 指令后续

可以看到他马上又跳转到到 50H 了，那么 50H 是在干嘛：

```
50H sw $5, #20($0) // 将$5寄存器的值存储到地址20处
```

回到这里的关键是多了一条 4CH 指令：

4CH	subu $5, $3,$4	[$5] = 0000_000DH
50H	sw $5, #20($0)	Mem[0000_0014H] = 0000_000DH

图 3.7: 4CH 指令

可以看到指令的解释，也就是第三列，这个手册对 50H 指令的解释是：把 0000_000DH 存储到 5 号寄存器中，而 0000_000DH 这个值是怎么来的呢？答案是从 4CH 减过来的。因此这么执行我并不认为是错误的。

上学期我在写报告的时候，我还不太懂延迟槽机制，因此我认为这里是错误的，增加了一个 stall。但是现在我认为我当时的修改是错误的。我认为这个 CPU 的整体工作状态是正确的。

3.2 配置 IP 核

最后 CPU 的 bug 应该只有一个，就是如果使用原始的 IP 核，那么会让 CPU 多停滞一个周期，但是 over 等信号并没有因为多增加的寄存器而改变，所以我们这里只需要正确配置 IP 即可，同时，由于我没看懂源代码的 IP 是怎么弄的，我自己写了一个。

```
module inst_rom(
    clka,
    addra,
    douta
);

// ENTITY inst_rom_ip IS 这里是赛灵思的IP接口说明
//  PORT (
//    clka : IN STD_LOGIC;
//    addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
//    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
//  );
// END inst_rom_ip;

input clka;
input [7 : 0] addra;
output [31 : 0] douta;

inst_rom_ip entity (
.clka(clka),
.addra(addra),
.douta(douta)
);

endmodule
```

其实我把赛灵思的 IP 接口说明拷贝过来注释就已经没有必要贴图片了，因为这个代码就是对 xci 文件/IP 核具体的描述，但是我这里还是贴上图片吧：

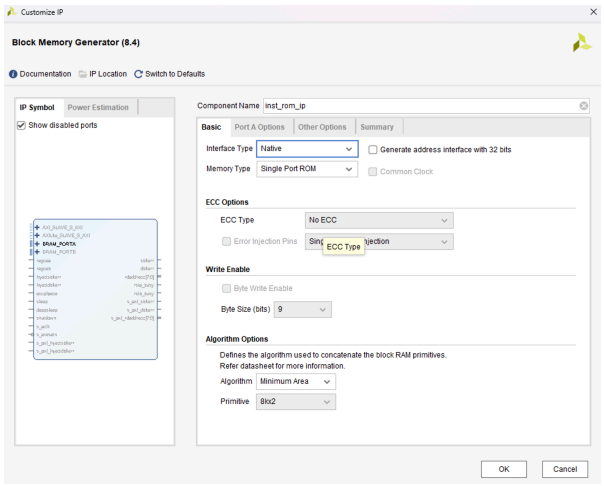

图 3.8: rom 选择

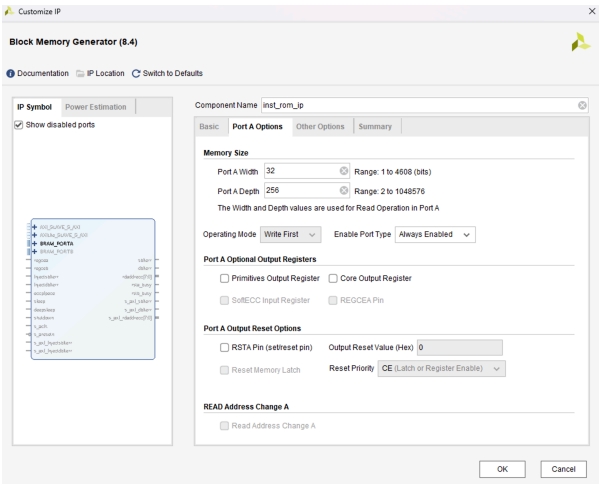

图 3.9: rom 配置

同理于 RAM，只是设置成了真双口。不过我看到网上有些对 RAM 的配置还选择了 No-change 模式，但是我的版本的 vivado 只有 Write-first 模式和 Read-First。对于这里我还是不太懂，但是其他的设置是什么意思我比较清楚。

这里还可以说一下 ROM 的特点，就是这个代码写 (或者说同步 ROM 导致) 的导致他会等一个周期才能拿到指令：

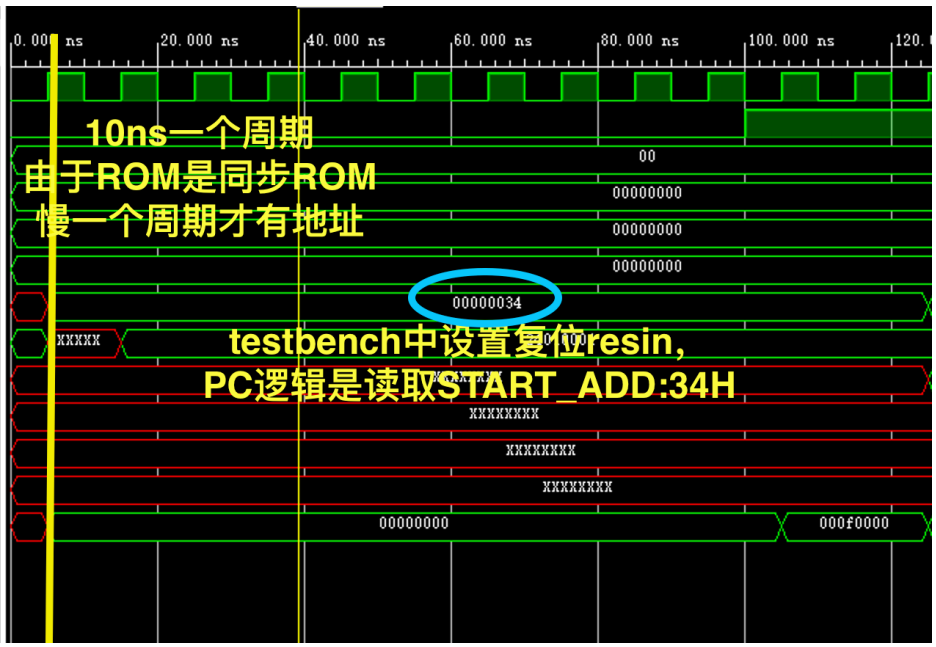

图 3.10: ROM 特点

4 新增指令说明

新增的指令有 9 条，分别如下表所示：

序号	指令名称	汇编指令
1	有符号乘法	mult rs, rt
2	从 LO 寄存器取值	mflo rd
3	从 HI 寄存器取值	mfhi rd
4	向 LO 寄存器存值	mtlo rs
5	向 HI 寄存器存值	mthi rs
6	从 cp0 寄存器取值	mfc0 rt, cs
7	向 cp0 寄存器存值	mtc0 rt, cd
8	系统调用	syscall
9	异常返回	eret

Table 3: 新增指令列表

其中，syscall、mt/mfc0 的作用在刚才为了讲解 CP0 寄存器的作用中已经讲过了，等下我他们直接分析在仿真中的行为。eret 指令刚才没有说，但是它是用于异常返回的，也就是 eret 指令执行后，会跳转到 EPC 寄存器的值。换句话说，他总是和 syscall 指令成对出现的。

我们可以看到在 coe 文件中，30H 的位置，也就是 Exception Handler 的结束位置，就是 eret 指令，他的作用是恢复现场，因此等下我也直接分析其仿真行为：

30H	eret	返回 108H	42000018
			CPU 复位地址 0000_0034H

图 4.11: Exception Handler 结束位置

4.1 乘法器指令讲解

乘法就是普通的乘法器实现的，没有做太多的优化。这里主要是关注 HI 和 LO 寄存器：

Listing 7: 乘法结果分配

```
1  //要写入HI的值放在exe_result里，包括MULT和MTHI指令
2  //要写入LO的值放在lo_result里，包括MULT和MTLO指令
3  assign exe_result = mthi ? alu_operand1 :
4                      mtc0    ? alu_operand2 :
5                      multiply ? product[63:32] : alu_result;
6  assign lo_result = mtlo ? alu_operand1 : product[31:0];
7  assign hi_write = multiply | mthi;
8  assign lo_write = multiply | mtlo;
```

可以看到，当出现 multiply 指令或者 mthi/mtlo 指令的时候，EXE 向下面的总线会包装上 hi_write/lo_write 信号，然后 WB 阶段会根据这个信号来写入 HI/LO 寄存器。

Table 4: 乘法相关指令

指令	功能	描述
MULT rs, rt	乘法运算	将 rs×rt 的结果存入 HI/LO
MFHI rd	从 HI 读取	将 HI 寄存器的值读入 rd
MFLO rd	从 LO 读取	将 LO 寄存器的值读入 rd
MTHI rs	向 HI 写入	将 rs 的值写入 HI 寄存器
MTLO rs	向 LO 写入	将 rs 的值写入 LO 寄存器

总结就是上面这个表格，那么具体地，我列出五个阶段的具体行为，等下仿真的时候我们验证：

1. **译码阶段**：识别 MULT 指令，设置 multiply 信号
2. **执行阶段**：
   - 启动乘法器，开始多周期运算
   - 等待 mult_end 信号，表示乘法完成
   - 将 64 位结果分为高 32 位和低 32 位
3. **访存阶段**：传递 HI/LO 写使能信号
4. **写回阶段**：
   - 将高 32 位写入 HI 寄存器
   - 将低 32 位写入 LO 寄存器

4.2 仿真行为

4.2.1 syscall 与 eret 指令仿真行为

首先我们分析 sys 和 eret 指令的仿真行为：

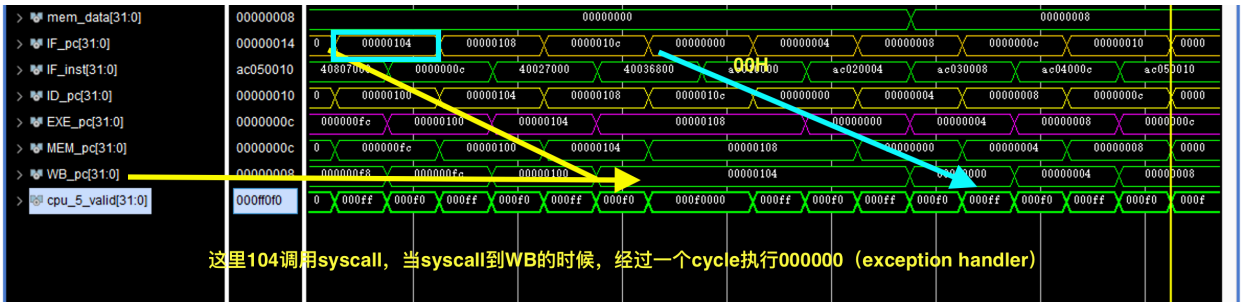

图 4.12: syscall 指令仿真行为

注意上面的执行，我挑选了 104H 这个位置（syscall 的位置），之后由于 syscall 需要等到 WB 阶段才能执行，因此会多读出来两个指令（当到 WB 阶段的时，且需要经过一周期后，00H 这个 Handler 才会进入到 IF 阶段，这个可以从设计图纸中看出，仿真图只是验证了其行为。

这里有必要插叙一个 cancle 信号：

Listing 8: cancel 信号

```
1   assign cancel = (syscall | eret) & WB_over;
```

这个信号是用于冲刷流水线的，也就是当 syscall 或 eret 指令执行完成后，会发出 cancel 信号，然后流水线会冲刷掉当前正在执行的指令，然后跳转到异常处理程序。

同时，在 pipeline_cpu.v 中，IF 允许进入的条件中也有 cancel 信号：

Listing 9: cancel 信号

```
1   assign IF_allow_in = (IF_over & ID_allow_in) | cancel;
```

这里时为了让 IF 可以立刻处理 syscall 的新指令，这点在 syscall 的仿真行为也得到验证，还是指刚过一周期的时候，假如 108H 或者 10CH 是乘法这种需要很多周期的指令，这个 cancle 信号依然能够保证 IF 可以立刻处理 syscall 的 Exception Handler。

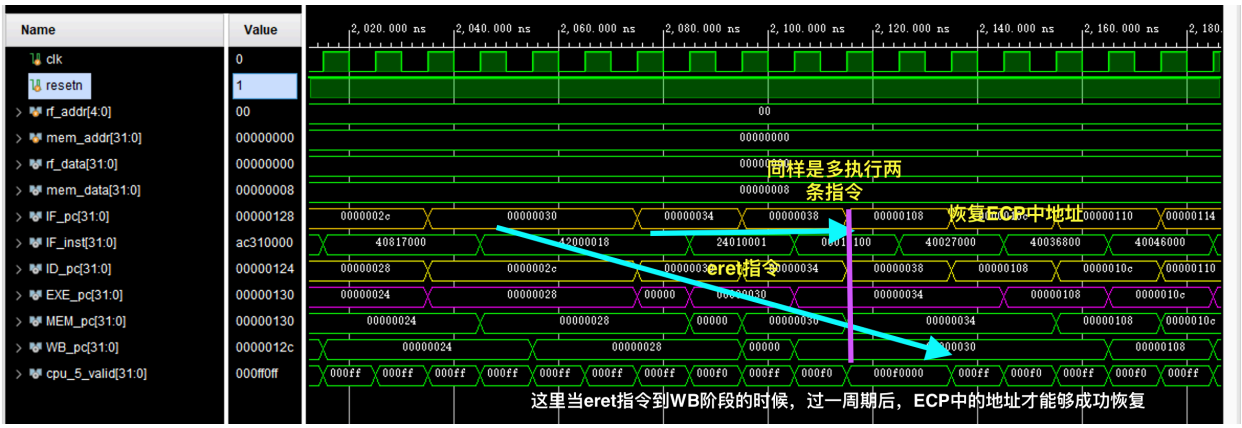

图 4.13: eret 指令仿真行为

上图是 eret 指令（在 30H）位置，执行完后（也就是在 WB 经过一周期后），108H 立即开始执行（因 IF 的 allow_in 信号由于 cancel 为 1）。验证了我们的分析。

注意，这里我突然想解释一下多拿出来的这两个指令为什么又重复执行了，这到底正不正确。我认为是正确的，因为上面的 syscall 取出来的两个指令在还没到 WB 阶段的时候就被 cancle 信号全给冲刷掉了，因此相当于没执行。所以这里 eret 要接着从 108H 开始执行才对，这样才能保证正确恢复现场。

4.2.2 有符号乘法

有符号乘法就是普通的乘法器实现的，没有做太多的优化。因此会消耗非常多的周期：

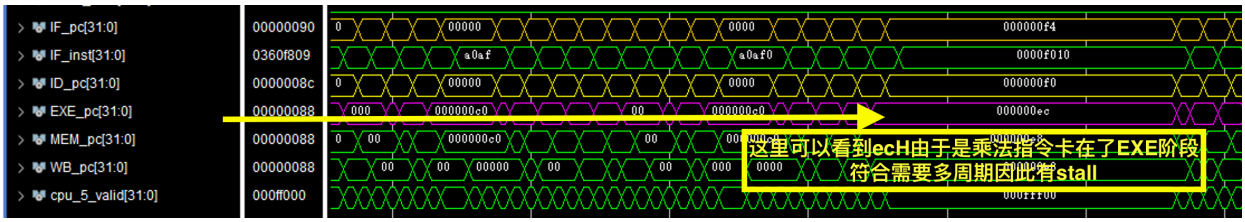

图 4.14: 有符号乘法仿真行为

这里可以观察到 ECH 位置的乘法指令卡在了 EXE 阶段，因为多周期乘法器需要消耗非常多的周期。

由于仿真是看不出来具体的寄存器值的（我没有增加显示接口），所以我们这里通过分析顶层 CPU 提供的 valid 信号来分析：

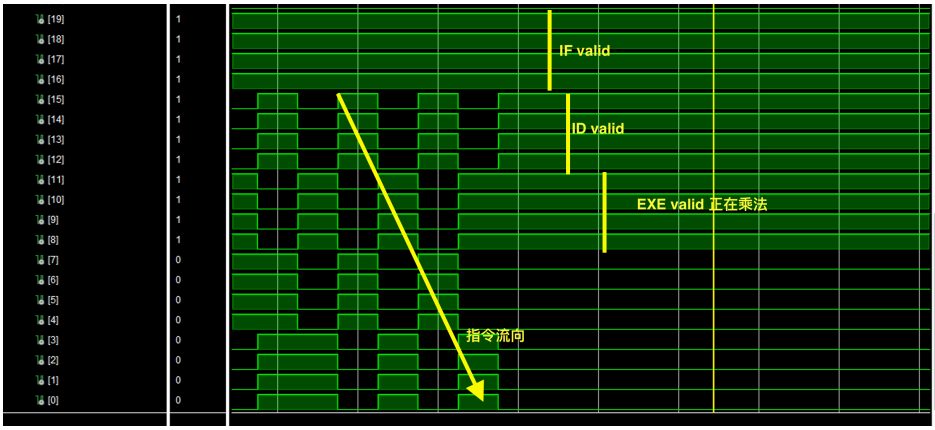

图 4.15: 有符号乘法 valid 行为 1

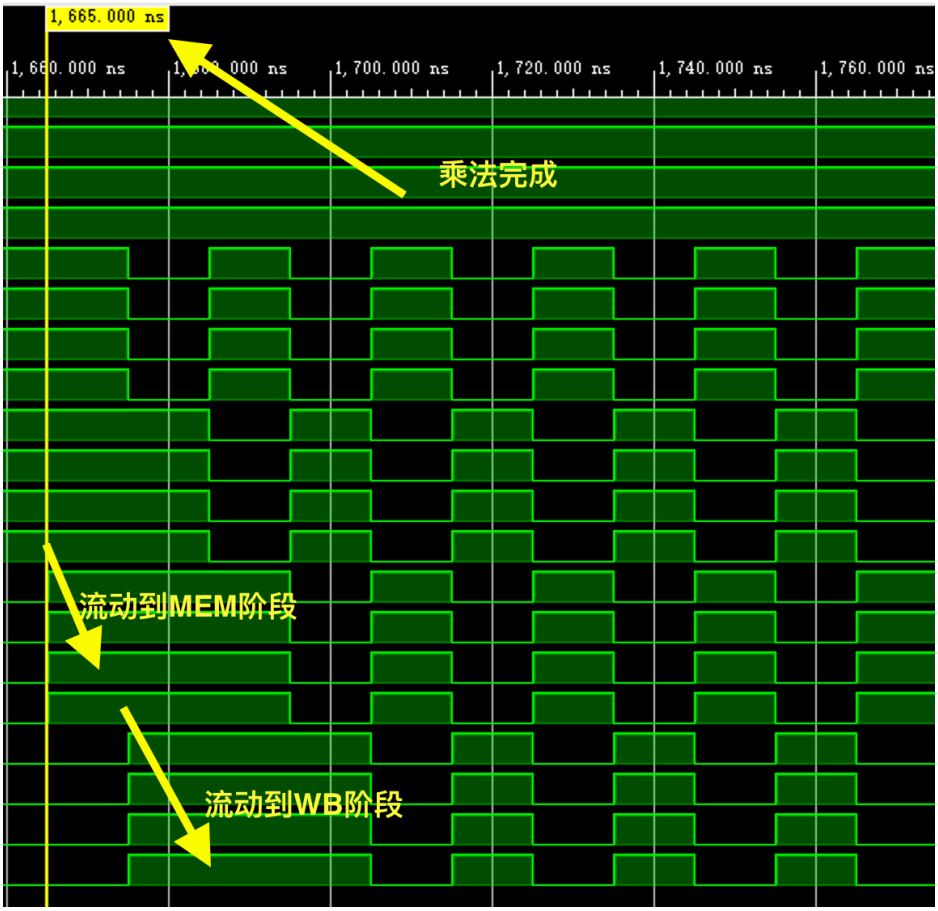

图 4.16: 有符号乘法 valid 行为 2

通过分析他们的 valid 信号，就可以更详细的查看乘法的行为。大概情况就是 valid 从卡住，后面阶段的 valid 过两个周期后都为 0 了（数据为空），代表流水线停滞，等乘法完成后，后面阶段的 valid 又过两个周期后都为 1 了（数据有效），代表流水线继续执行。

4.2.3 mfc0 与 mtc0 指令仿真行为

```
108H mfc0 $2, cp0(14.0)
10CH mfc0 $3, cp0 (13.0)
110H mfco $4, cp0 (12.0)
```

这三个指令是把 cp0 寄存器读取出来，然后写入到寄存器堆中。仿真图为：

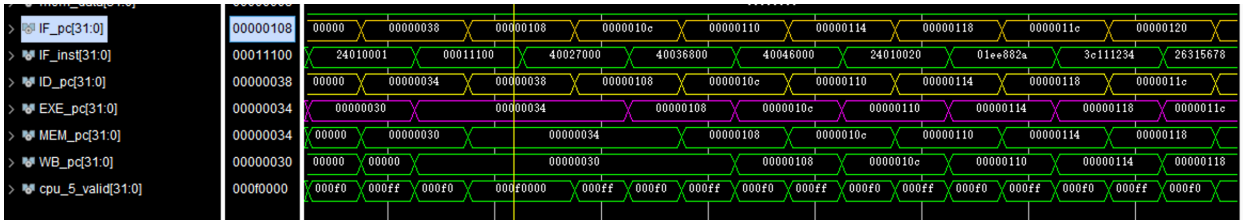

图 4.17: mfc0 与 mtc0 指令仿真行为

由于仿真确实不太能看到详细结果，除非手动再引出一些显示接口。这里我们给出一个逻辑上的分析，口述一下 mfc0 指令调用的时候发生了什么：

decode 阶段 在 decode.v 中，MFC0 指令的识别逻辑：

Listing 10: MFC0 指令识别

```
assign inst_MFC0 = (op == 6'b010000) & (rs==5'd0)
              & sa_zero & (funct[5:3] == 3'b000);
```

译码条件：

- 操作码为 010000（协处理器 0 指令）
- rs 域为 0（源寄存器为 0）
- sa 域为 0（特殊域为 0）
- funct[5:3] 为 000（协处理器 0 操作）

除此之外：

Listing 11: ID 阶段 MFC0 信号生成

```
assign mfc0 = inst_MFC0;
assign cp0r_addr = {rd, cp0r_sel}; // rd=14, cp0r_sel=0
```

关键信号：

- `mfc0 = 1`：标识为 MFC0 指令
- `cp0r_addr = 14`：目标 CP0 寄存器地址
- `rf_wen = 1`：需要写回寄存器堆
- `rf_wdest = 2`：目标寄存器为 $2

execute 阶段 在 exe 阶段：

Listing 12: EXE 阶段 MFC0 处理

```
// MFC0指令在EXE阶段不需要ALU运算
// 直接传递控制信号到后续阶段
assign EXE_MEM_bus = {..., mfc0, ..., rf_wen, rf_wdest, ...};
```

执行特点：

- 不需要 ALU 运算
- 直接传递控制信号
- 准备 CP0 寄存器读取

memory 阶段 在 MEM 阶段主要涉及这些代码：

Listing 13: MEM 阶段 MFC0 处理

```
    // MEM阶段继续传递MFC0信号
    assign MEM_WB_bus = {..., mfc0, ..., rf_wen, rf_wdest, ...};
```

处理过程：

- 继续传递 MFC0 控制信号
- 准备 CP0 寄存器数据读取
- 维护寄存器写回信息

write back 阶段 最后 WB 阶段会读取 CP0 寄存器，然后写入到寄存器堆中：

Listing 14: WB 阶段 CP0 寄存器读取

```
        // CP0寄存器读取逻辑
        wire [31:0] cp0r_rdata;
        assign cp0r_rdata = (cp0r_addr=={5'd12,3'd0}) ? cp0r_status :
                            (cp0r_addr=={5'd13,3'd0}) ? cp0r_cause :
                            (cp0r_addr=={5'd14,3'd0}) ? cp0r_epc : 32'd0;

        // 寄存器堆写回数据选择
        assign rf_wdata = mfhi ? hi :
                          mflo ? lo :
                          mfc0 ? cp0r_rdata : mem_result;
```

关键操作：

- 根据 cp0r_addr=14，读取 EPC 寄存器
- 将 EPC 寄存器的值作为 rf_wdata

- 写回到 $2 寄存器

到此为止，mfc0 指令的行为就分析完了。

5 实验总结

1. 我自行学习了级联控制，对延迟槽机制更加熟悉了。
2. 详细分析了 syscall、eret、mfc0、mtc0 指令的行为，并给出了逻辑上的分析。
3. 有一个地方最开始一直没看懂，就是为什么有些指令即使读进去也认为没问题（我指 syscall 的后面两条），后来想通了，我认为应该是由于这两条指令是没有机会走到 WB 阶段的，因此不会造成实质性影响。
4. 总的来说我更加熟悉 CPU 了，我打算国庆的时候把这学期要求的功能能实现多少实现多少。