

# LAB 4

学号:2020K8009915008

姓名:林孟颖 箱子号:79

## LAB 4

### 一.实验任务概览

### 二.实验设计

#### (一) 总体设计思路

1. 流水线CPU的设计理念
2. 各流水级交互方式
3. myCPU总框架示意图

#### (二) 重要模块1设计: IF阶段

1. 工作原理
2. 接口定义
3. 功能描述

#### (三) 重要模块1设计: ID阶段

1. 工作原理
2. 接口定义
3. 功能描述

#### (四) 重要模块1设计: EXE阶段

1. 工作原理
2. 接口定义
3. 功能描述

#### (五) 重要模块1设计: MEM阶段

1. 工作原理
2. 接口定义
3. 功能描述

#### (六) 重要模块1设计: WB阶段

1. 工作原理
2. 接口定义
3. 功能描述

### 三.实验过程

#### (一) 实验流水

#### (二) 实践任务7: 不考虑相关冲突处理的简单流水线CPU

##### 7.1 多周期CPU

- 错误1: 指令寄存器读使能设置不合理
- 错误2: 分支有效信号br\_taken设置不合理
- 错误3: PC更新时间有误
- 错误4: ID→IF的条件未考虑需写回的分支指令

##### 7.2 简单流水线CPU

- 错误1: 信号未定义&类型定义有误
- 错误2: 有序端口和命名端口连接混合
- 错误3: PC无法正常更新
- 错误4: 信号位宽定义有误
- 错误5: 分支取消处理不当
- 错误6: 隐式变量定义
- 错误7: 未提前申请取数据
- 错误8: 信号滞后导致的数据相关

#### (三) 实践任务9: 前递技术解决相关引发的冲突

##### 9.1 初版思路: ALU计算提前

- 错误1: 信号位宽定义有误
- 错误2: 未进行数据预取&提前计算读数地址

## 一.实验任务概览

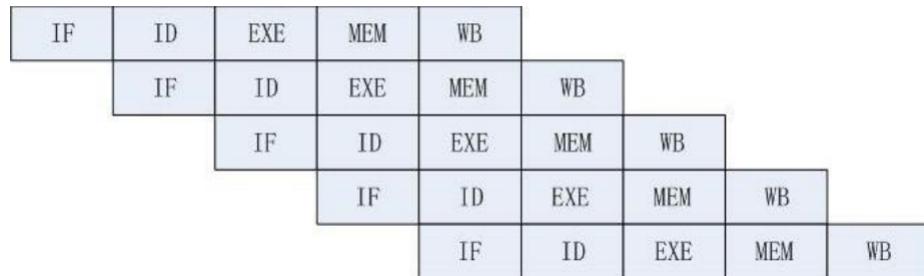
- 完成不考虑数据冲突的五级流水线的简单CPU设计；
- 了解流水线冲突原因以及冲突的类型（结构相关、数据相关、控制相关）；
- 了解流水线冲突的解决方式（流水线阻塞和数据前递），最终实现数据前递以减少阻塞带来的性能损失。

## 二.实验设计

### (一) 总体设计思路

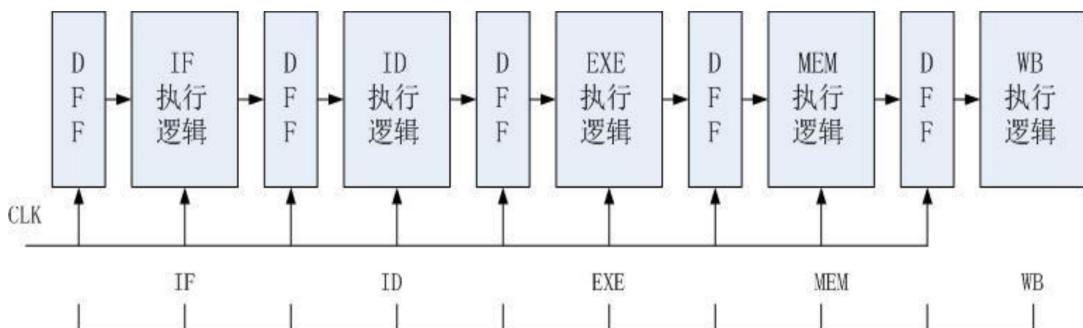
#### 1. 流水线CPU的设计理念

流水线的核心思想是把多条指令的不同执行阶段重叠起来，使得 CPU 能同时处理多条指令。处于不同流水级的指令使用不同的物理器件，以求更高效地利用硬件资源。在流水线 CPU 中，每条指令的执行过程被分成若干个执行阶段，本实验中将之切分为五个流水级，分别为 IF、ID、EXE、MEM、WB 阶段，流水线时空图如下：



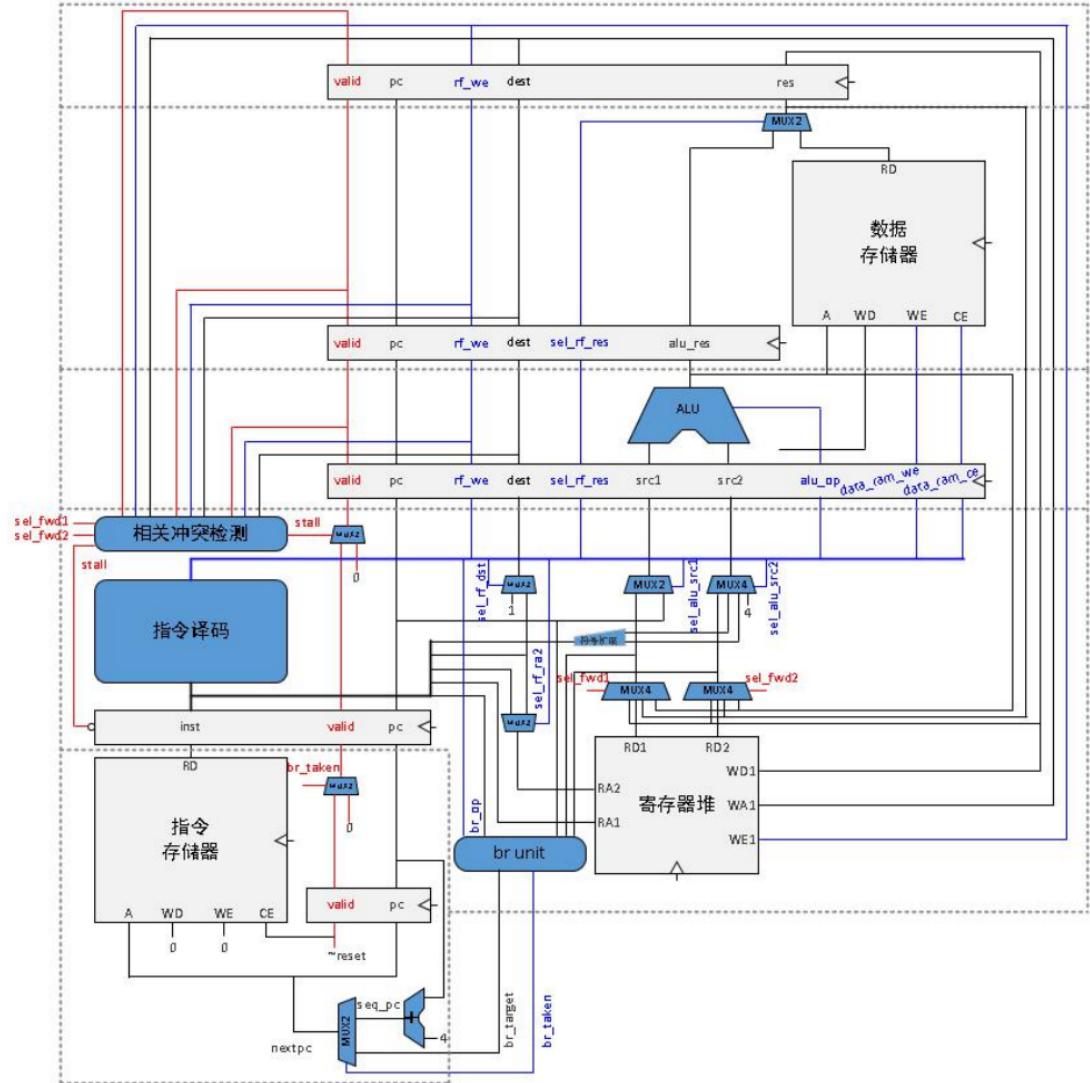
#### 2. 各流水级交互方式

两级流水级间需设有一组寄存器，暂存处理结果，如下图所示（DFF的D型触发器）：



信号是否从流水级A传递给流水级B取决于两个信号：流水级A的 `A2B_valid` 和流水级B的 `allow_in`，前者标志着A传递给B的指令是否是有效的，而后者标志着B是否允许A的信号流入，通过控制这两个信号可以控制流水的流动与阻塞。

### 3. myCPU总框架示意图

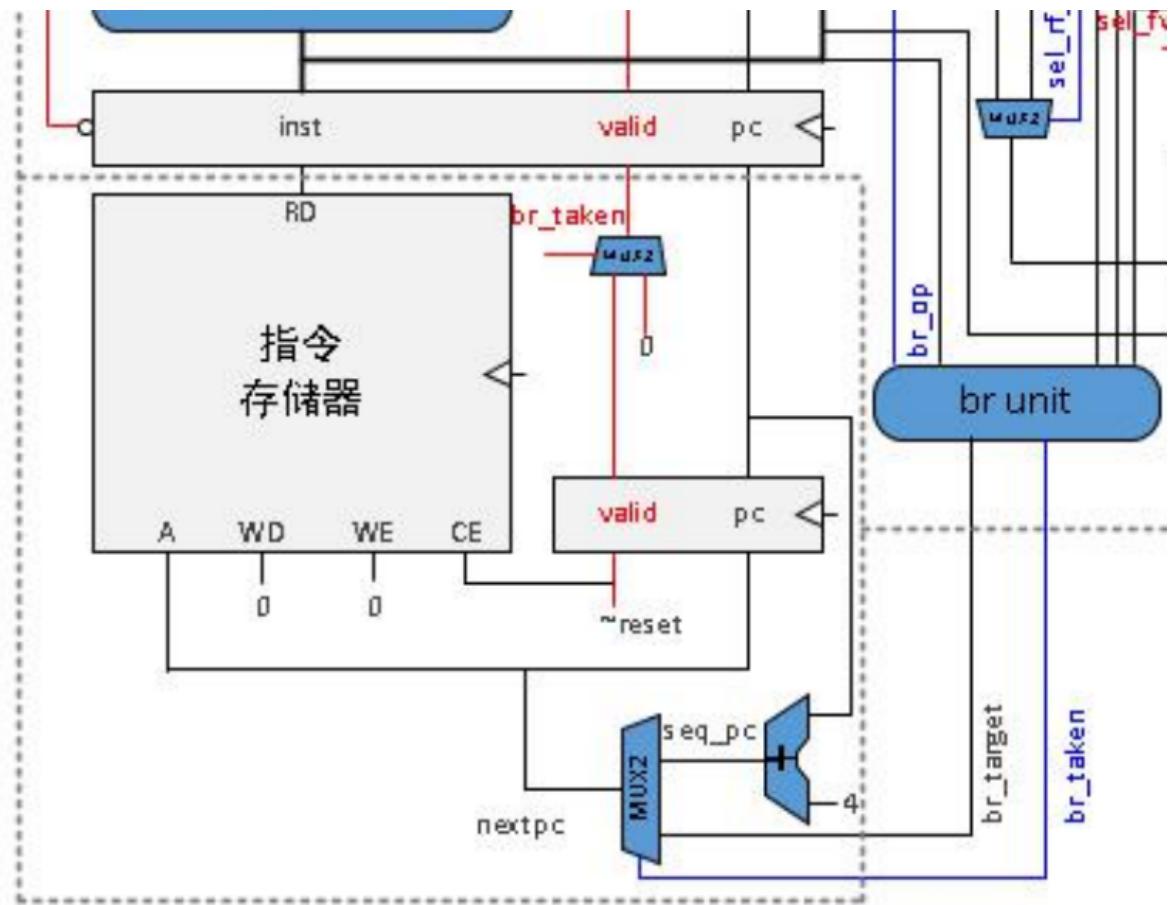


## (二) 重要模块1设计：IF阶段

### 1. 工作原理

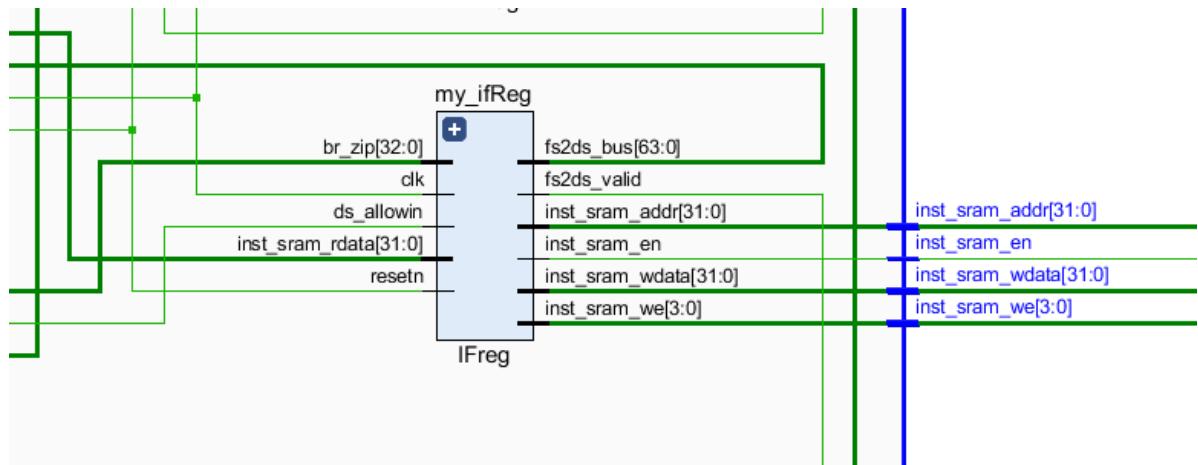
IF阶段根据后续流水级传递而来的信号（如br\_taken和后续实验的例外信号等）判断下一条指令的地址nextpc（现阶段nextpc的来源仅有顺序执行的pc以及分支指令的跳转地址，br\_taken拉低则选取顺序执行pc），并将之送往指令寄存器。

值得一提的是指令寄存器是异步读写，其读出指令的时机为给出指令地址的下一个时钟上升沿，故应该提前给出指令地址（pre-IF预取指）。



## 2. 接口定义

注：接口信号只列出重要功能信号，时钟信号和复位信号未单独列出。



| Name            | I/O | Func                                  |
|-----------------|-----|---------------------------------------|
| br_zip          | IN  | ID阶段传递来的分支指令相关信号，包含br_taken和br_target |
| ds_allowin      | IN  | ID阶段传递来的准入信号                          |
| inst_sram_rdata | IN  | 指令寄存器中读出的数据                           |
| inst_sram_addr  | OUT | CPU传递给指令寄存器的指令地址                      |
| inst_sram_en    | OUT | CPU传递给指令寄存器的读使能信号                     |
| inst_sram_wdata | OUT | CPU传递给指令寄存器的写数据                       |
| inst_sram_we    | OUT | CPU传递给指令寄存器的写使能信号（始终为0）               |
| fs2ds_bus       | OUT | IF传递给ID的数据zip，包含fs_inst和fs_pc         |

### 3. 功能描述

IF：取指令阶段，从指令存储器中获取指令存入IR寄存器（传递给IR流水级），并更新PC寄存器值。

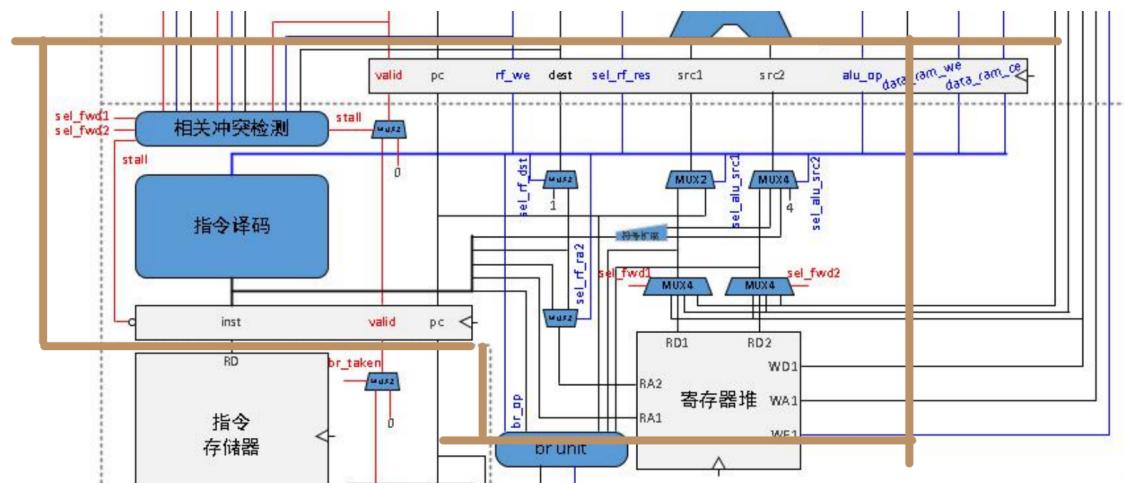
## (三) 重要模块1设计：ID阶段

### 1. 工作原理

ID阶段对IF送来的指令进行译码操作，确定当前指令类型并生成后续流水级所需控制信号，如ALU的操作码、操作数等。

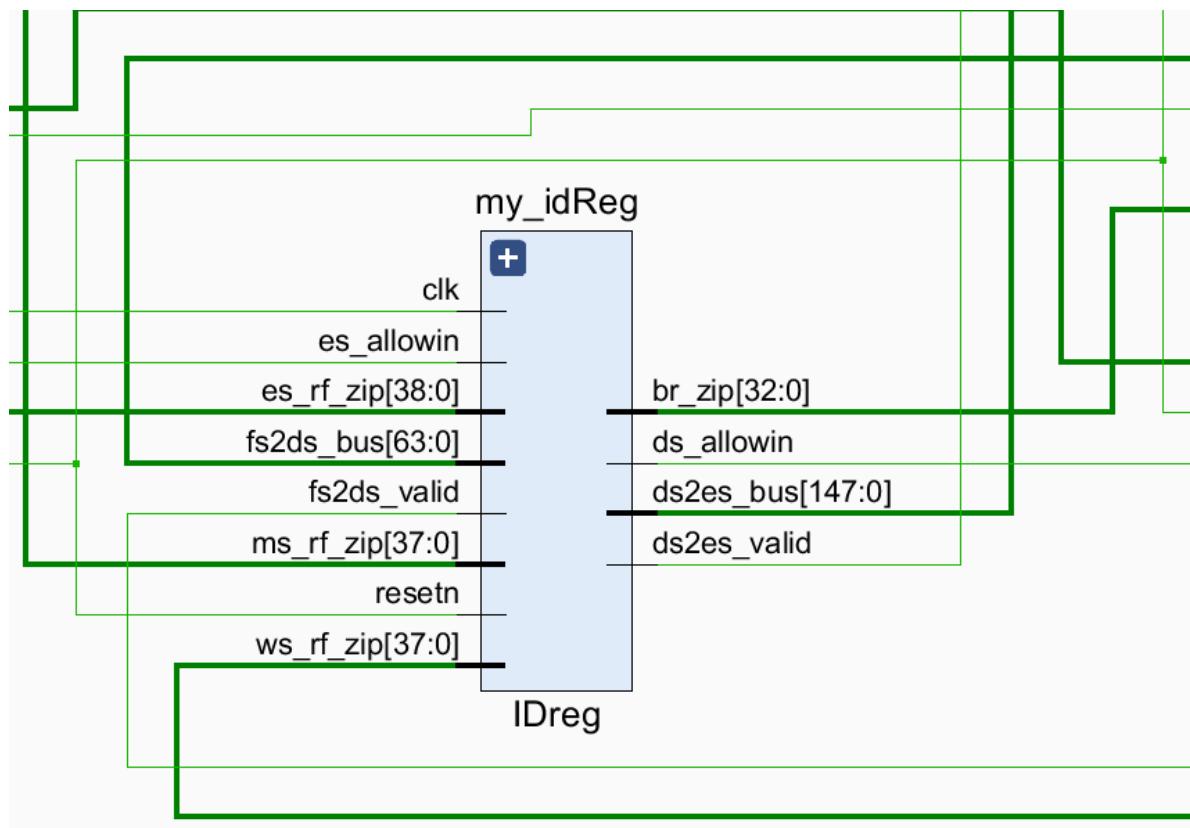
同时，ID级在判断ALU操作数时还需注意是否存在数据相关的情况，根据后续流水级传递来的寄存器写地址、写有效信号、写数据判断最终有效的数据；

需要注意的是，遇到访存指令+寄存器指令的数据相关时，需要阻塞一周期。阻塞是通过控制ds\_ready\_go的信号实现的，其与ds2es\_valid和ds\_allowin相关，其拉低可确保流水线被阻塞。



框线给出的是ID阶段的模块

## 2. 接口定义



| Name                    | I/O | Func   |
|-------------------------|-----|--|
| <code>br_zip</code>     | OUT | ID阶段传递给IF阶段的分支指令相关信号，包含br_taken和br_target  |
| <code>ds_allowin</code> | OUT | ID阶段传递给IF阶段的准入信号   |
| <code>ds2es_bus</code>  | OUT | ID传递给EXE阶段的数据zip，包含ds_alu_op, ds_res_from_mem, ds_alu_src1, ds_alu_src2, ds_mem_we, ds_rf_we, ds_rf_waddr, ds_rkd_value, ds_pc |
| <code>fs2ds_bus</code>  | IN  | IF传递给ID的数据zip，包含fs_inst和fs_pc  |
| <code>es_allowin</code> | IN  | EXE阶段传递给ID阶段的准入信号  |
| <code>ws_rf_zip</code>  | IN  | WB阶段传递给ID阶段的写寄存器相关信号，用于判断是否存在数据冲突，若存在，使用前递技术解决。包含ws_rf_we, ws_rf_waddr, ws_rf_wdata  |
| <code>es_rf_zip</code>  | IN  | EXE阶段传递给ID阶段的写寄存器相关信号，用于判断是否存在数据冲突，若存在，使用前递技术解决。包含es_res_from_mem, es_rf_we, es_rf_waddr, es_alu_result。                       |
| <code>ms_rf_zip</code>  | IN  | MEM阶段传递给ID阶段的写寄存器相关信号，用于判断是否存在数据冲突，若存在，使用前递技术解决。包含ms_rf_we, ms_rf_waddr, ms_rf_wdata   |

### 3. 功能描述

ID：指令译码阶段，根据 IR 寄存器中的指令译码出各个阶段需要的控制信号，同时确定 EXE 阶段需要的操作数。操作数的来源有两个，从寄存器堆 Regfile 中获取或指令的立即数进行扩展。

从寄存器堆中读取数据时可能存在冲突的情况，故ID阶段还肩负着判断是否存在数据相关、使用前递技术选择出正确操作数的功能。注意操作数的选取具有一定优先级关系，若存在来自 EXE、MEM、WB 的寄存器写使能信号多个拉高的情况，应遵从 EXE > MEM > WB 的原则，确保使用的数据是最新写入寄存器的。

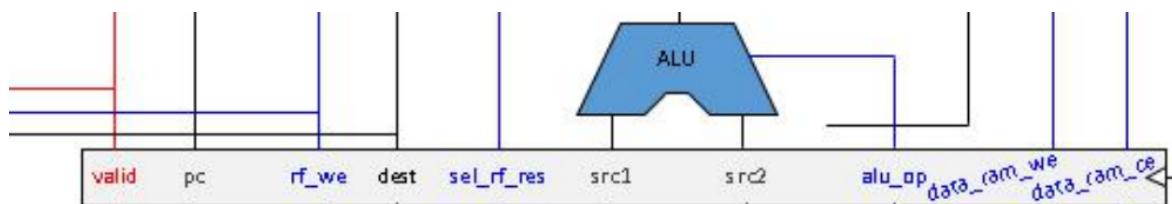
## (四) 重要模块1设计：EXE阶段

### 1. 工作原理

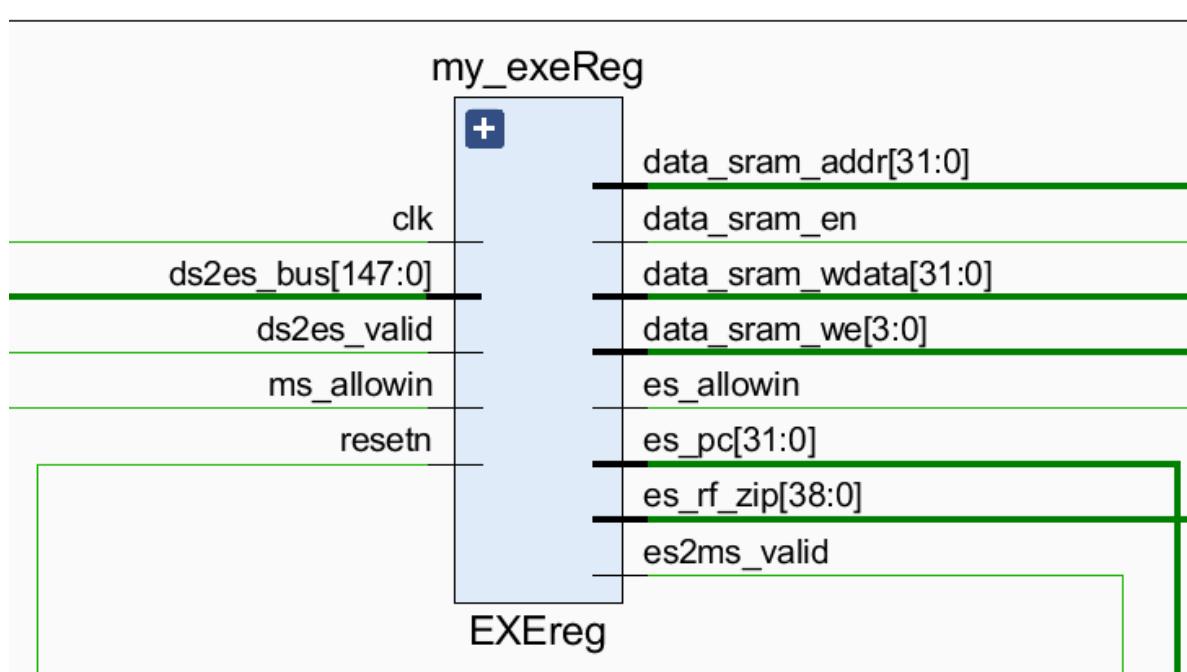
EXE阶段从ID阶段取得有效操作数和操作码后，进行算术逻辑运算。

同时EXE需根据ID传来的 res\_from\_mem 和 mem\_we 信号判断是否需要拉高数据寄存器的使能信号和写使能信号，其中读写地址有alu计算结果给出，写数据由 rkd\_value 给出。其读使能信号的拉高是为后续 MEM阶段做准备。

此外，其需将写寄存器相关信号传递给ID阶段用于判断是否存在数据相关。



### 2. 接口定义



| Name            | I/O | Func   |
|-----------------|-----|--|
| data_sram_en    | OUT | EXE阶段传递给数据RAM的使能信号   |
| data_sram_we    | OUT | EXE阶段传递给数据RAM的写使能信号  |
| data_sram_addr  | OUT | EXE阶段传递给数据RAM的数据地址   |
| data_sram_wdata | OUT | EXE阶段传递给数据RAM的写数据  |
| es_allowin      | OUT | EXE阶段传递给ID阶段的准入信号  |
| es_rf_zip       | OUT | EXE传递给MEM阶段的数据zip，相当于es2ms_bus，由于其还将传递给ID阶段判断数据相关，为功能明确，故命名为es_rf-zip。<br>含es_res_from_mem, es_rf_we, es_rf_waddr, es_alu_result |
| es_pc           | IN  | EXE阶段的pc信号，单独拆出的原因是后续中断异常实验还需传递给其他流水级。   |
| es2ms_valid     | OUT | EXE阶段传递给MEM阶段的有效信号   |
| ds2es_bus       | IN  | ID传递给EXE的数据zip，内容见ID级介绍  |
| ms_allowin      | IN  | MEM阶段传递给EXE阶段的准入信号   |

### 3. 功能描述

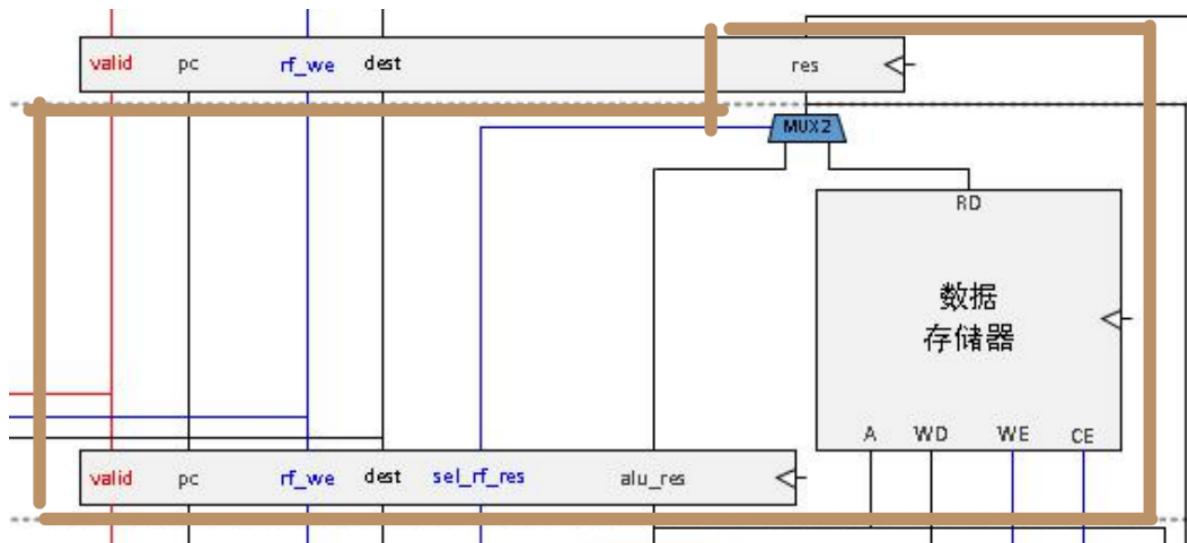
EXE：指令执行阶段，ALU 根据操作数和操作码完成算数逻辑运算，产生数据RAM的读写信号、写地址与写数据。

## (五) 重要模块1设计：MEM阶段

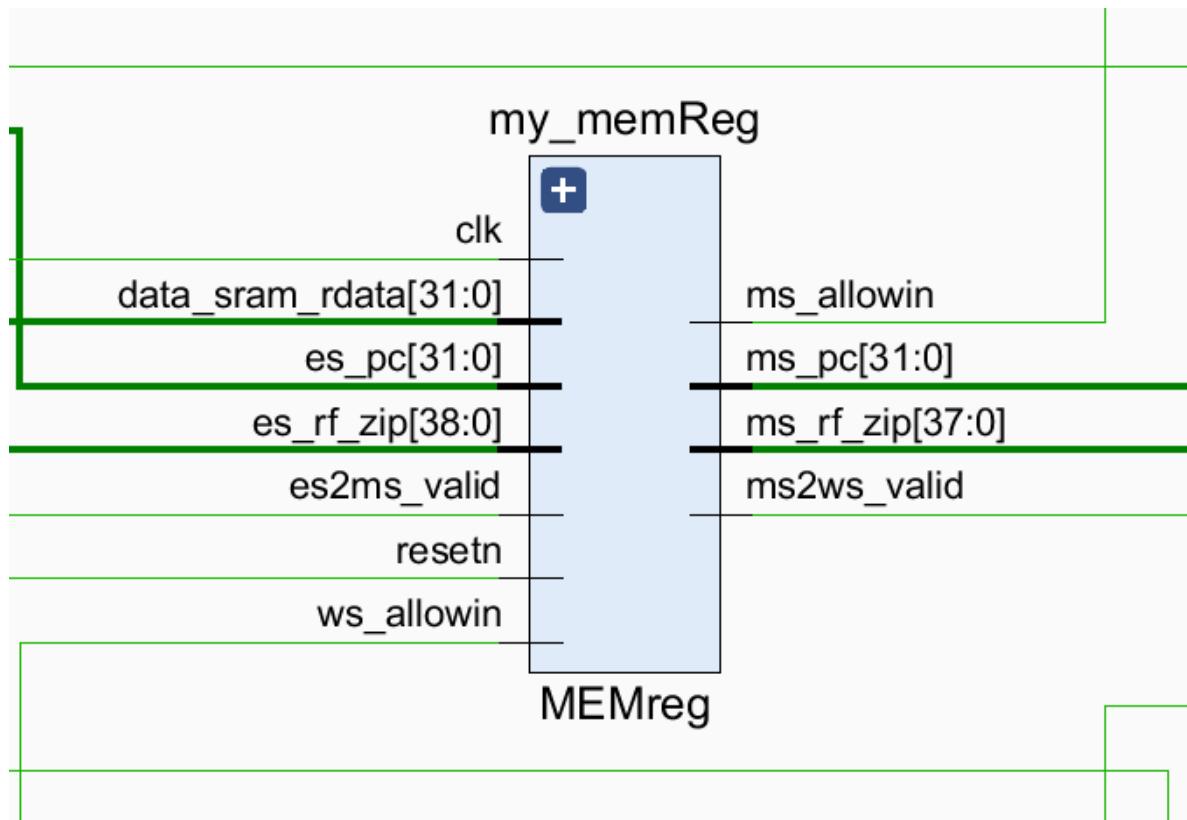
### 1. 工作原理

MEM阶段取到数据存储器读出的数据（若EXE中拉高了数据存储器的使能信号），其将根据res\_from\_mem信号判断写回数据的来源是数据存储器还是alu计算结果，并生成写回的wdata。

其需将写寄存器相关信号传递给ID阶段用于判断是否存在数据相关。



## 2. 接口定义



| Name                         | I/O | Func   |
|------------------------------|-----|--|
| <code>data_sram_rdata</code> | IN  | EXE阶段传递给数据RAM的使能信号   |
| <code>es_pc</code>           | IN  | EXE阶段的pc信号   |
| <code>es_rf_zip</code>       | IN  | EXE阶段传递来的寄存器相关信号   |
| <code>es2ms_valid</code>     | IN  | EXE阶段传递给MEM的有效信号   |
| <code>ws_allowin</code>      | IN  | WB阶段传递给EXE阶段的准入信号  |
| <code>es_rf_zip</code>       | OUT | EXE传递给MEM阶段的数据zip，相当于 <code>es2ms_bus</code> ，由于其还将传递给ID阶段判断数据相关，为功能明确，故命名为 <code>es_rf-zip</code> 。 |
| <code>ms2ws_valid</code>     | OUT | MEM阶段传递给WB阶段的有效信号  |
| <code>ms_pc</code>           | OUT | MEM阶段的PC   |
| <code>ms_rf_zip</code>       | OUT | MEM阶段寄存器相关信号。含 <code>ms_rf_we</code> , <code>ms_rf_waddr</code> , <code>ms_rf_wdata</code> 。         |
| <code>ms_allowin</code>      | OUT | MEM阶段传递给EXE阶段的准入信号   |

## 3. 功能描述

`MEM`：访问存储器阶段，EXE阶段计算的结果作为数据存储器的地址输入端读取数据（地址低2位应为0），如果数据SDRAM写使能信号拉高，还需要把rt寄存器的数据写入数据存储器对应存储单元。

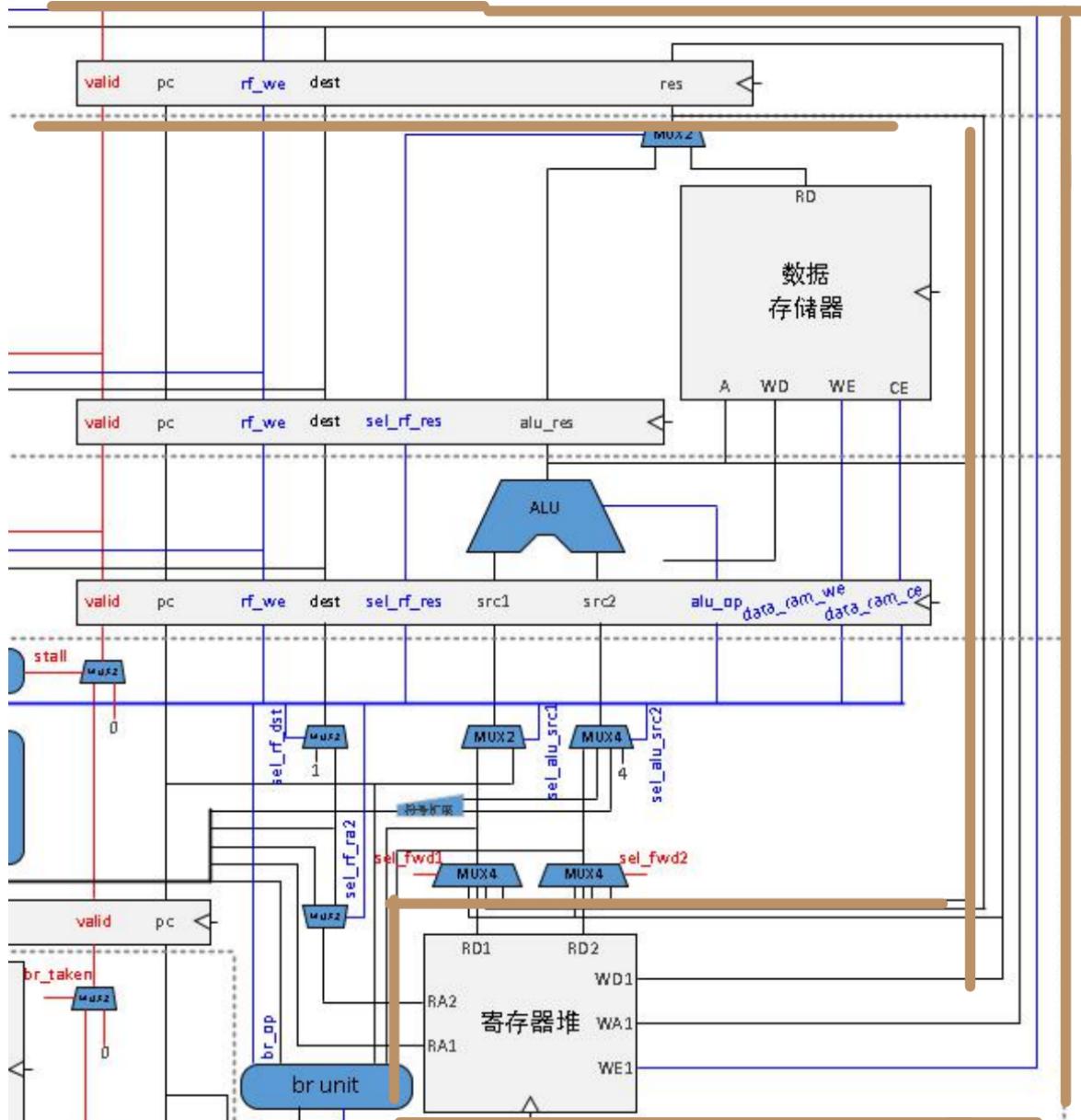
## (六) 重要模块1设计：WB阶段

### 1.工作原理

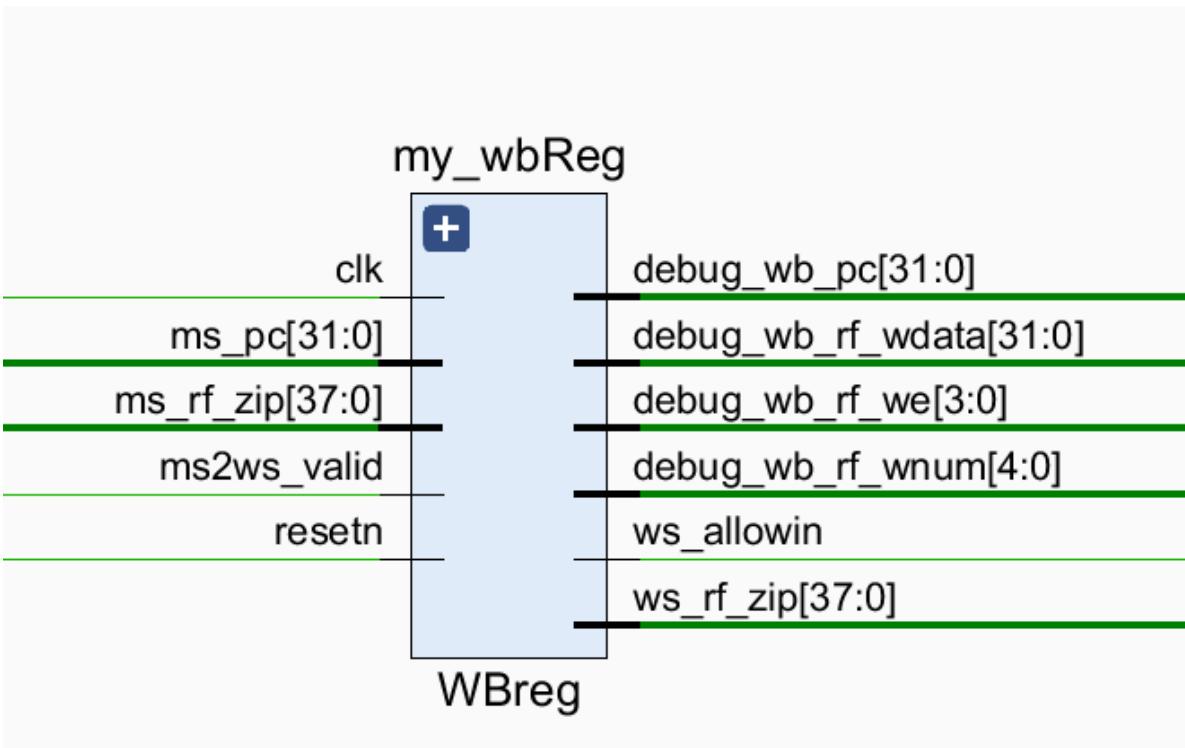
WB阶段根据MEM阶段传递来的寄存器写使能信号判断是否需要写回，并根据写地址、写数据将正确的数据写入目标寄存器。其需将写寄存器相关信号传递给ID阶段用于判断是否存在数据相关。

同时，其与debug模块进行交互，若当前写寄存器信号拉高，会驱动debug模块更新PC、寄存器写地址以及写数据，据此可判断是否写回有误。

图中框线给出的即为



### 2.接口定义



| Name                        | I/O | Func   |
|-----------------------------|-----|--|
| <code>ms2ws_valid</code>    | IN  | MEM阶段传递给WB阶段的有效信号  |
| <code>ms_pc</code>          | IN  | MEM阶段的PC   |
| <code>ms_rf_zip</code>      | IN  | MEM阶段寄存器相关信号。含 <code>ms_rf_we</code> , <code>ms_rf_waddr</code> , <code>ms_rf_wdata</code> 。 |
| <code>ws_allowin</code>     | OUT | WB阶段传递给MEM阶段的准入信号  |
| <code>ws_rf_zip</code>      | OUT | WB阶段写寄存器相关信号, 传递给ID判断数据冲突  |
| <code>debug_wb_we</code>    | OUT | WB传递给debug模块的写使能信号, 拉高则驱动debug模块更新   |
| <code>debug_wb_pc</code>    | OUT | WB传递给debug模块的PC  |
| <code>debug_wb_wdata</code> | OUT | WB传递给debug模块的寄存器写数据  |
| <code>debug_wb_wnum</code>  | OUT | WB传递给debug模块的写寄存器号   |

### 3. 功能描述

WB：数据写回阶段，选择写回寄存器堆的数据（来自ALU计算结果或访存数据），根据前面的流水段传递来的写使能控制信号和目的寄存器号将数据写回。

## 三. 实验过程

### (一) 实验流水

- 2022.09.05 19:00- 24:00 阅读讲义&进行多周期CPU设计
- 2022.09.06 8:00-9:40 22:00-次日1:00 多周期CPU改为流水线CPU
- 2022.09.07 8:00- 9:40 13:30-14:30 使用前递技术解决数据相关（同时采用ALU计算前提至ID阶段的策略）
- 2022.09.17 20:25- 23:00 据PIAZZA上建议将ALU重新改至EXE阶段，并将模块接口规范化命名。

## (二) 实践任务7：不考虑相关冲突处理的简单流水线CPU

### 7.1 多周期CPU

按照讲义所言，先划分为多个阶段，再让其执行过程“流动起来”。

讲义的意思应该是先理解多周期划分的原理，再在此基础上写流水线，而我起初理解有误，老老实实地又写了一个多周期CPU，在此基础上再改成流水线，以为此后轻而易举了，但是由于我设计的多周期CPU中，未必所有指令都需经历五个周期（如部分指令无需访存等，这些可由状态机给出），后续划分模块、加上流水时（纯寄存器指令也需经历访存阶段）还是de了很久的bug……此部分先记录写多周期CPU时的bug们。

#### 错误1：指令寄存器读使能设置不合理

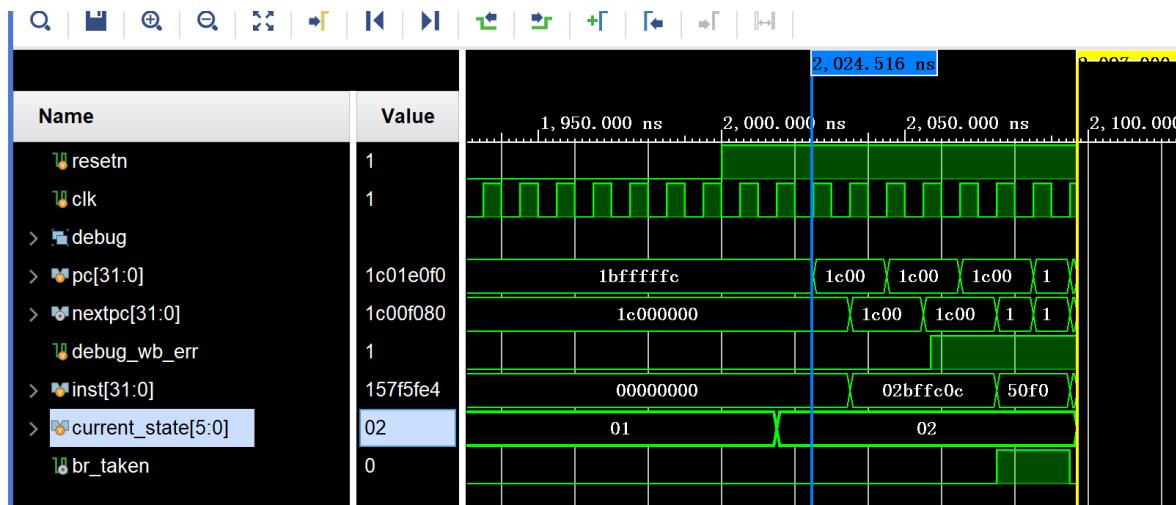
##### (1) 错误现象

PC和参考值不相符：

```
=====
Test begin!
=====
[ 2057 ns] Error!!!
    reference: PC = 0x1c00f07c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfffff000
    mycpu   : PC = 0x1c000004, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
=====
```

##### (2) 分析定位过程

查看波形：



发现指令未及时更新：原定于IF阶段取指（current\_state=0x02），则指令寄存器inst应该在蓝色光标处更新。查看cpu和指令寄存器的交互：

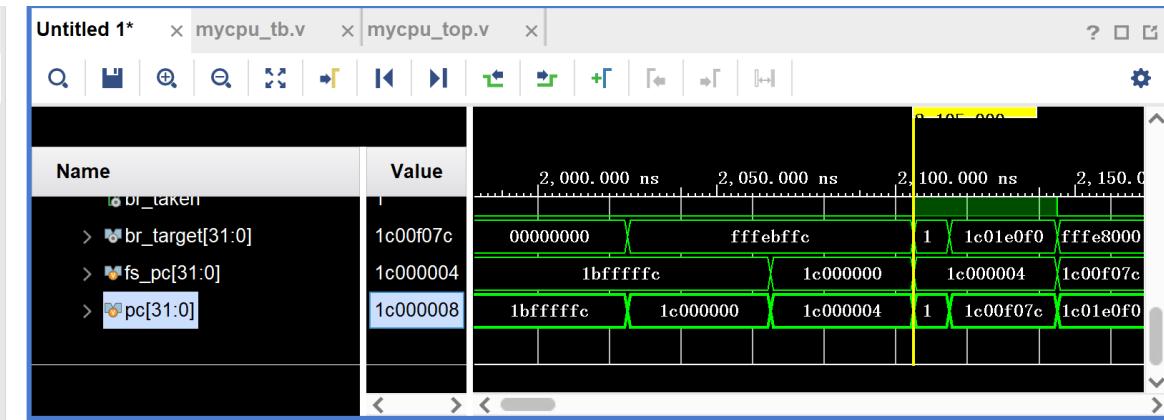
```
assign inst_sram_en      = current_state[1]; //IF阶段读指令寄存器
```

##### (3) 错误原因

将 `inst_sram_en` 如上赋值，将导致指令寄存器在下一个时钟上升沿才被使能，由于每条指令的处理都需要经过取指阶段（即访问指令寄存器），可将之改为始终赋值为 `resetn`。

##### (4) 修正效果

波形可继续往后跑：

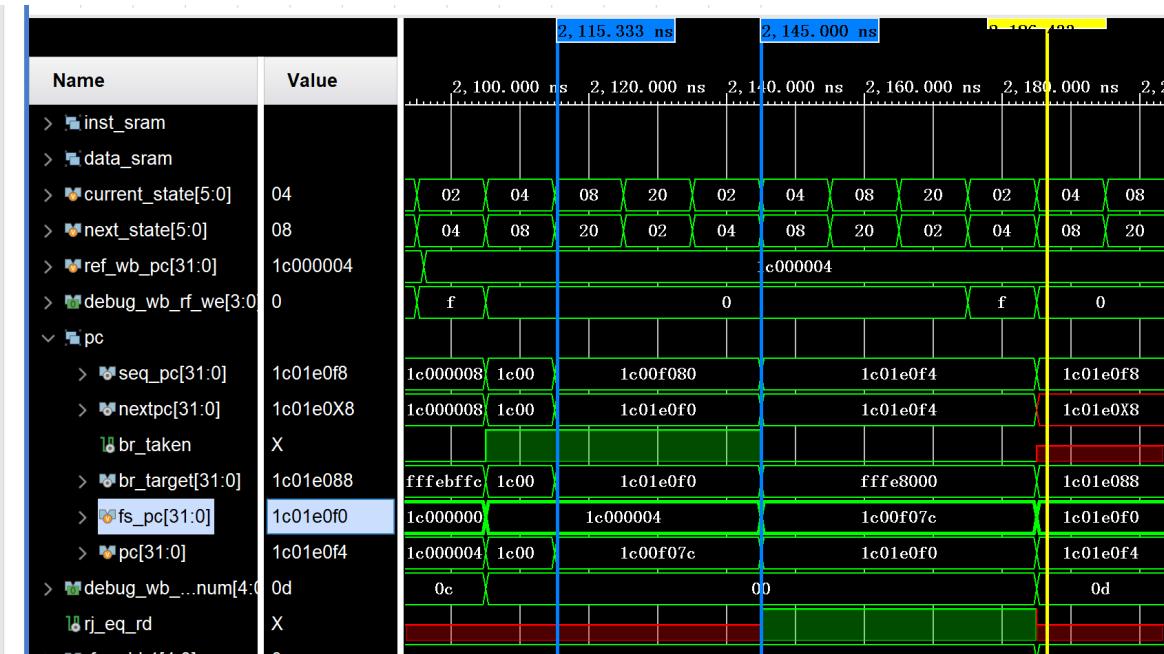


## (5) 归纳总结 (可选)

### 错误2：分支有效信号br\_taken设置不合理

#### (1) 错误现象

pc 突然崩了，开始乱跑，图中两个蓝色光标处分别是两次PC跳变：



#### (2) 分析定位过程

PC异常跳变是由分支有效信号异常导致的，查看原 br\_taken 的定义：

```
assign br_taken = (  inst_beq  &&  rj_eq_rd
                  || inst_bne  && !rj_eq_rd
                  || inst_jirl
                  || inst_b1
                  || inst_b
                );
```

#### (3) 错误原因

br\_taken 受指令寄存器inst决定，而inst只会在取指阶段的下一个上升沿才发生更新，未归零，在发生首次branch后进入IF阶段，而此时指令寄存器中存的仍是上一个指令，导致nextpc的值选择为br\_target，进而出错。

#### (4) 修正效果

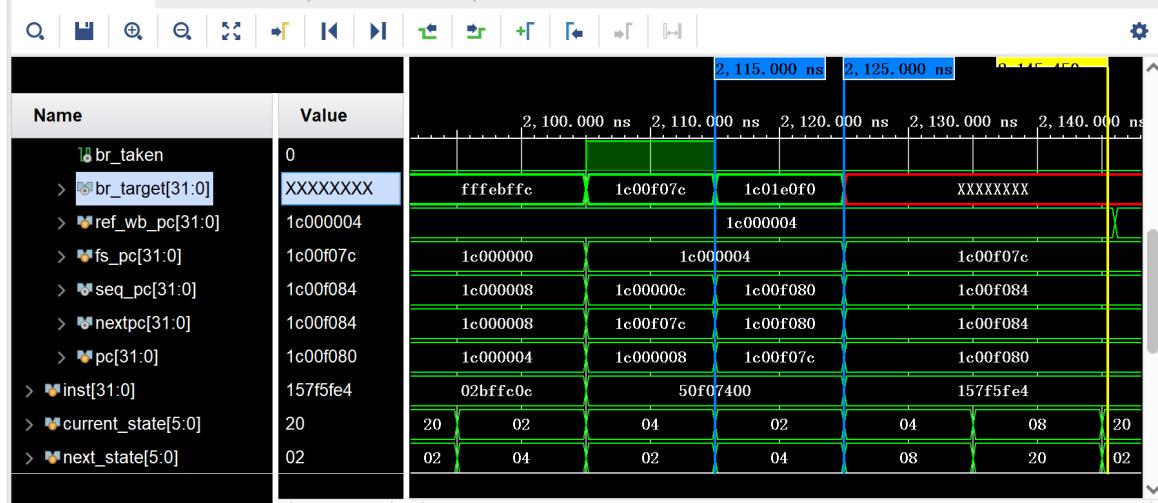
需确保 br\_taken 仅在ID阶段有效：

```

assign br_taken = ( inst_beq && rj_eq_rd
                   || inst_bne && !rj_eq_rd
                   || inst_jirl
                   || inst_b1
                   || inst_b
                   ) & current_state[2];

```

修改后PC不会二次分支：



(5) 归纳总结（可选）

### 错误3：PC更新时间有误

(1) 错误现象

PC提早更新，与参考PC未对上：

```

[ 2147 ns] Error!!!
reference: PC = 0x1c00f07c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfaaff000
mycpu     : PC = 0x1c00f080, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfaaff000

```

(2) 分析定位过程

注意到错误2的修正图中PC虽然没有二次分支，但是转移后的IF阶段又对PC做了一次更新 (+4)。观察pc更新逻辑：

```

//PC更新
always @(posedge clk) begin
    if (current_state[0]) begin
        pc <= 32'h1BFF_FFFC;
    end
    else if(current_state[1] | current_state[2] & br_taken) begin //IF阶段PC
        更新或ID阶段判断出需要转移
        pc <= nextpc;
    end
end

```

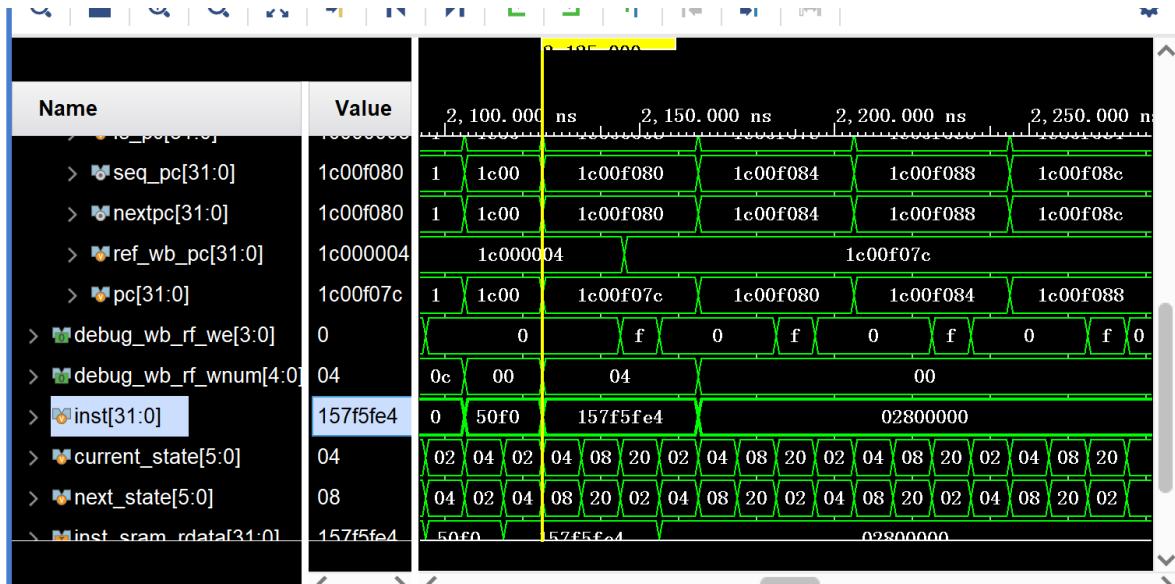
(3) 错误原因

pc只能在IF阶段更新，上述多做了一次更新，应改为

```
//PC更新
always @(posedge clk) begin
    if (current_state[0]) begin
        pc <= 32'h1BFF_FFFC;
    end
    else if(current_state[1]) begin //IF阶段PC更新
        pc <= nextpc;
    end
end
```

#### (4) 修正效果

此处pc和参考值可正常对上。



#### (5) 归纳总结 (可选)

未理清逻辑，**此处PC只应在取指阶段更新**，虽然讲义建议跳转转移PC更新时间安排在译码阶段，但鉴于**我的多周期CPU设计中未必所有指令都包含五个流水级**，即对无需写回的转移指令ID阶段后紧接着就是下一条指令的IF阶段，故若按照讲义所述在ID阶段再做一次更新将导致一条指令被跳过。

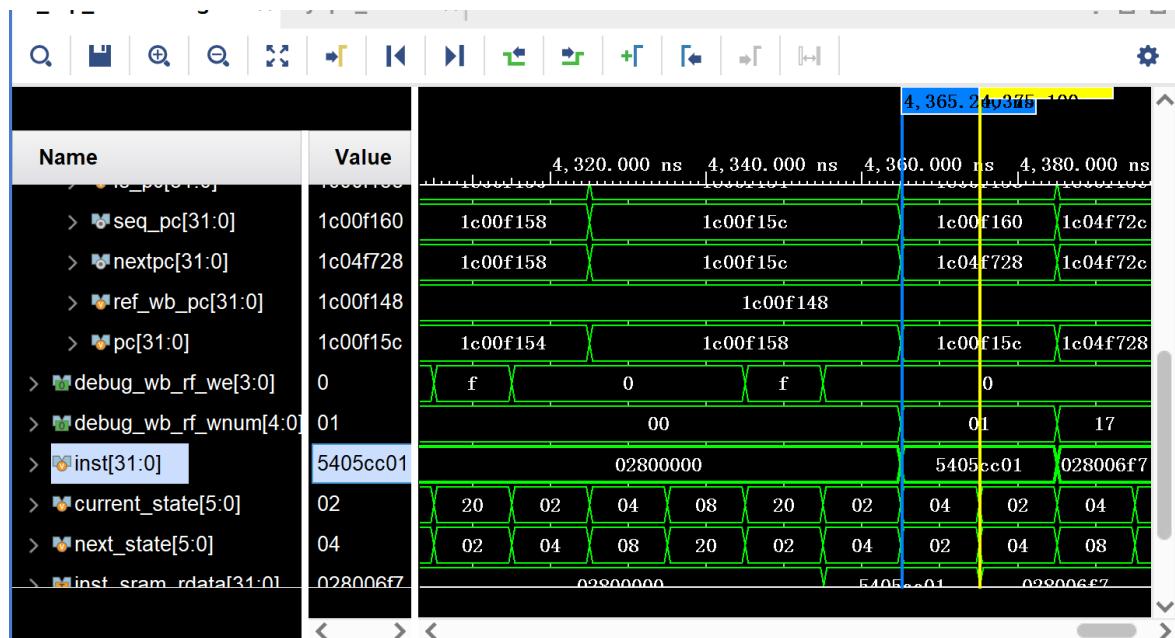
**错误4：ID→IF的条件未考虑需写回的分支指令**

### (1) 错误现象

PC不匹配

## (2) 分析定位过程

查看PC=0x1c00f15c时的波形：



结合汇编代码：

```
1c00f15c: 5405cc01 b1 263628(0x405cc) # 1c04f728 <n1_lu12i_w_test>
```

意识到未对需写回的分支指令作特殊处理。

### (3) 错误原因

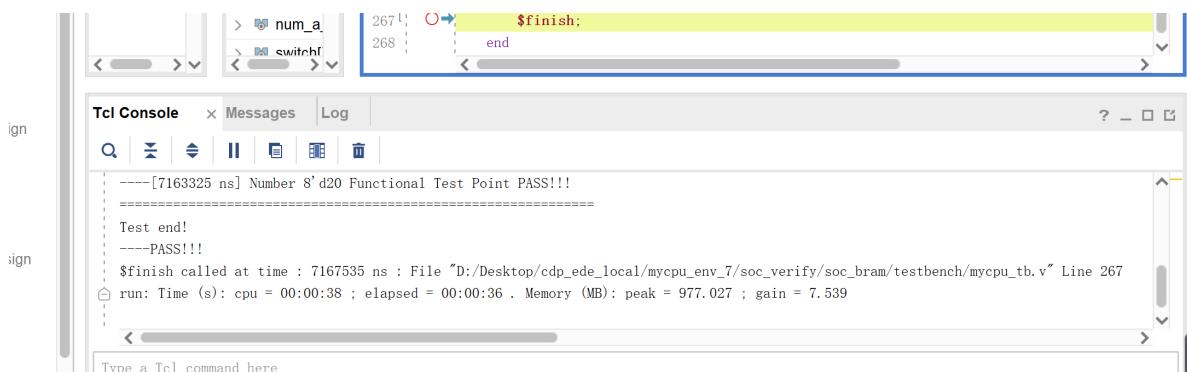
`b1`会使`br_taken`有效，下一周期直接进入取指周期，跳过写回阶段，因而标准PC的更新滞后，导致不匹配。同理还有`jrl`，此处一并修改。

### (4) 修正效果

对ID跳转至IF的条件作如下修改：

```
ID:begin
    if(~|inst | br_taken & ~inst_b1 & ~inst_jrl) //nop指令或者需要跳转
        next_state = IF;
    else if(id_ready_go)
        next_state = EXE;
    else
        next_state = ID;
end
```

幸福来得太突然！！！ PASS！！！



### (5) 归纳总结 (可选)

需要注意分支类指令中需要写寄存器的特例。

## 7.2 简单流水线CPU

较之多周期CPU，流水线CPU是多条指令并行的，部分硬件资源（alu、regfile等）在每个周期都会被充分使用（除了初始几个周期），简单地把不同指令的相关信号全部堆在一起十分的混乱，同时某些信号在传输到特定阶段后就无需再保存（如指令在译码后就无需再传递给EXE阶段）。在阅读《自己动手写CPU》后，考虑将各个状态模块化，将之隔离开来，各模块间只传递必要的信息。

起初我将所有信号拆分传递，以IF和ID阶段为例：

```
module IFreg(
    input wire          clk,
    input wire          resetn,
    // inst sram interface
    output wire         inst_sram_en,
    output wire [3:0]   inst_sram_we,
    output wire [31:0]  inst_sram_addr,
    output wire [31:0]  inst_sram_wdata,
    input  wire [31:0]  inst_sram_rdata,
    // if and id state interface
    input  wire          id_allowin,
    input  wire          br_taken,
    input  wire [31:0]  br_target,
    output wire         if_to_id_valid,
    output wire [31:0]  if_inst,
    output reg [31:0]   if_pc
);
```

```
module IDreg(
    input wire          clk,
    input wire          resetn,
    // if and id state interface
    output wire         id_allowin,
    output wire          br_taken,
    output wire [31:0]  br_target,
    input  wire          if_to_id_valid,
    input  wire [31:0]  if_inst,
    input  wire [31:0]  if_pc,
    // id and exe state interface
    input  wire          exe_allowin,
    output wire [5:0]   id_rf_zip, // {id_rf_we, id_rf_waddr}
    output wire          id_to_exe_valid,
    output reg [31:0]   id_pc,
    output wire [75:0]  alu_data_zip, // {alu_op, alu_src1, alu_src2}
    output wire          res_from_mem, // res_from_mem
    output wire          mem_we,
    output wire [31:0]  rkd_value;
    // id and wb state interface
    input  wire [37:0]  wb_data_zip // {wb_rf_we, wb_rf_waddr, wb_rf_wdata}
);
```

但这样在设计顶层模块时候就需要声明非常多的变量来连接各个模块，同时十分容易出现增添信号时端口漏接的情况。

注意到上述设计我在ID和WB阶段交互的界面处将regfile相关的信号（`wb_data_zip`）打包传输、在ID和EXE交互的界面将alu相关的信号（`alu_data_zip`）也打包传输，这样可以在次顶层模块中将若干引线合为一股，信号的功能更加明确，且不易出现漏接的现象。

后续老师在讲课时也建议划分模块，同时老师还提及建议将各个流水级交互的信号全部打包，而我的实验记录在老师讲课前完成，课后我根据老师的讲解又将代码规范化了，故此处部分代码和最终代码不完全一致。

老师讲解后，我又将所有信号封装成bus，并将交互信号命名改为老师建议的模式，以ID阶段的模块为例：

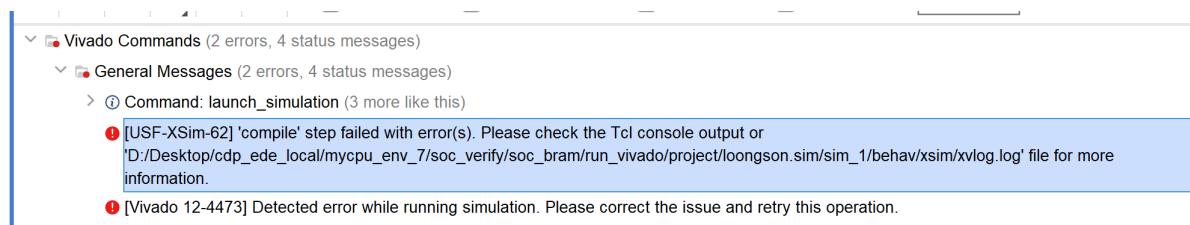
```
module IDreg(
    input wire          clk,
    input wire          resetn,
    // fs and ds interface
    input wire          fs2ds_valid,
    output wire         ds_allowin,
    output wire [32:0]   br_zip,
    input wire [`FS2DS_LEN -1:0] fs2ds_bus,
    // ds and es interface
    input wire          es_allowin,
    output wire         ds2es_valid,
    output wire [`DS2ES_LEN -1:0] ds2es_bus,
    // signals to determine whether conflict occurs
    input wire [37:0]   ws_rf_zip, // {ws_rf_we, ws_rf_waddr, ws_rf_wdata}
    input wire [37:0]   ms_rf_zip, // {ms_rf_we, ms_rf_waddr, ms_rf_wdata}
    input wire [38:0]   es_rf_zip // {es_res_from_mem, es_rf_we, es_rf_waddr,
es_alu_result}
);
```

以下记录将多周期CPU改为流水线CPU过程：

### 错误1：信号未定义&类型定义有误

#### (1) 错误现象

跑仿真时console报错：



#### (2) 分析定位过程

上述报错告知错误被重定位到xvlog.log，查看该文件：

```
xvlog.log - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
INFO: [VRFC 10-2263] Analyzing Verilog file
"D:/Desktop/cdp_edc_local/mycpu_env_7/soc_verify/soc_bram/rtl/xilinx_ip/data_ram/sim/data_ram.v" into library xil_defaultlib
INFO: [VRFC 10-311] analyzing module data_ram
INFO: [VRFC 10-2263] Analyzing Verilog file
"D:/Desktop/cdp_edc_local/mycpu_env_7/soc_verify/soc_bram/rtl/xilinx_ip/inst_ram/sim/inst_ram.v" into library xil_defaultlib
INFO: [VRFC 10-311] analyzing module inst_ram
INFO: [VRFC 10-2263] Analyzing Verilog file
"D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/EXEreg.v" into library xil_defaultlib
INFO: [VRFC 10-311] analyzing module EXEreg
ERROR: [VRFC 10-2989] 'alu_data_zip' is not declared
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/EXEreg.v:47]
ERROR: [VRFC 10-1280] procedural assignment to a non-register exe_rf_zip is not
permitted, left-hand side should be reg/integer/time/genvar
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/EXEreg.v:48]
ERROR: [VRFC 10-818] illegal expression in target
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/EXEreg.v:48]
ERROR: [VRFC 10-2865] module 'EXEreg' ignored due to previous errors
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/EXEreg.v:1]
```

### (3) 错误原因

由xvlog.log中信息得知 alu\_data\_zip 未定义（可能是将多周期CPU拆分时改名改漏了），同时注意到 exe\_rf\_zip 误被定义为wire类型。

### (4) 修正效果

console依旧报错，但xvlog.log的报错内容不再是信号定义问题。

```
"D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/mycpu_top.v" into library xil_defau
INFO: [VRFC 10-311] analyzing module mycpu_top
ERROR: [VRFC 10-1247] port connections cannot be mixed ordered and named
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/mycpu_top.v:58]
ERROR: [VRFC 10-2865] module 'mycpu_top' ignored due to previous errors
[D:/Desktop/cdp_edc_local/mycpu_env_7/myCPU/mycpu_top.v:1]
```

### (5) 归纳总结（可选）

模块拆分时需要尤其注意在每个分模块中补齐信号定义，否则有可能遇到变量隐式声明的情况（默认为宽度为1的wire类型），此类bug非常难发现。

## 错误2：有序端口和命名端口连接混合

### (1) 错误现象

xvlog.log报错如错误1中修改后的图片。

### (2) 分析定位过程

没见过此类错误，上网搜索回答如下：

[Synth 8-2543] port connections cannot be mixed ordered and named

说明例化时最后一个信号添加了一个逗号。

反观代码：

wb\_rf\_zip

```
IFreg my_ifReg(
    .clk(clk),
    .resetn(resetn),
    .inst_sram_we(inst_sram_we),
    .inst_sram_addr(inst_sram_addr),
    .inst_sram_wdata(inst_sram_wdata),
    .inst_sram_rdata(inst_sram_rdata),
    .id_allowin(id_allowin),
    .br_taken(br_taken),
    .br_target(br_target),
    .if_to_id_valid(if_to_id_valid),
    .if_inst(if_inst),
    .if_pc(if_pc),
);
```

#### (3) 错误原因

IF模块例化时最后一个端口加了逗号

#### (4) 修正效果

仿真可以进行。

### 错误3：PC无法正常更新

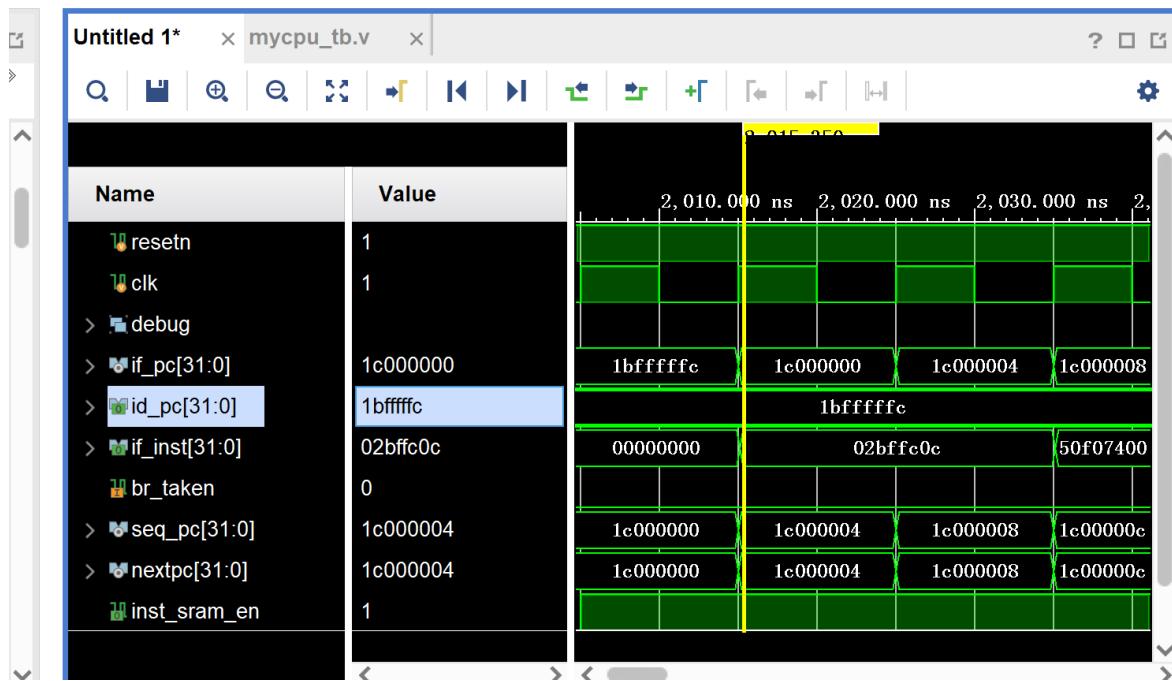
#### (1) 错误现象

如图，PC始终未更新

```
[11282000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11292000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11302000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11312000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11322000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11332000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11342000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11352000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11362000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11372000 ns] Test is running, debug_wb_pc = 0x1bfffffc
[11382000 ns] Test is running, debug_wb_pc = 0x1bfffffc
```

#### (2) 分析定位过程

查看波形得：if\_pc 可正常更新，而 id\_pc 未变化



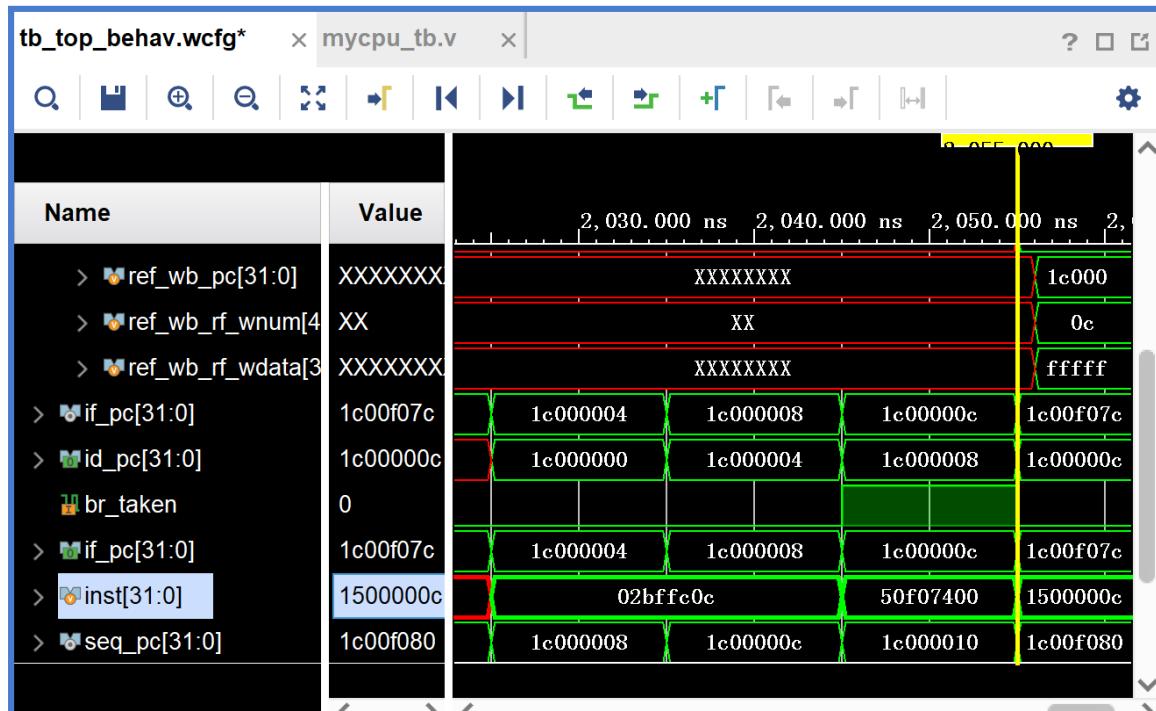
查看 id\_pc 赋值逻辑：

```
always @(~posedge clk) begin
    if(~if_to_id_valid & id_allowin) begin
        id_pc <= if_pc;
    end
end
```

### (3) 错误原因

错把 if\_to\_id\_valid 写成了 ~if\_to\_id\_valid ...

### (4) 修正效果



id\_pc可随之更新。

#### 错误4：信号位宽定义有误

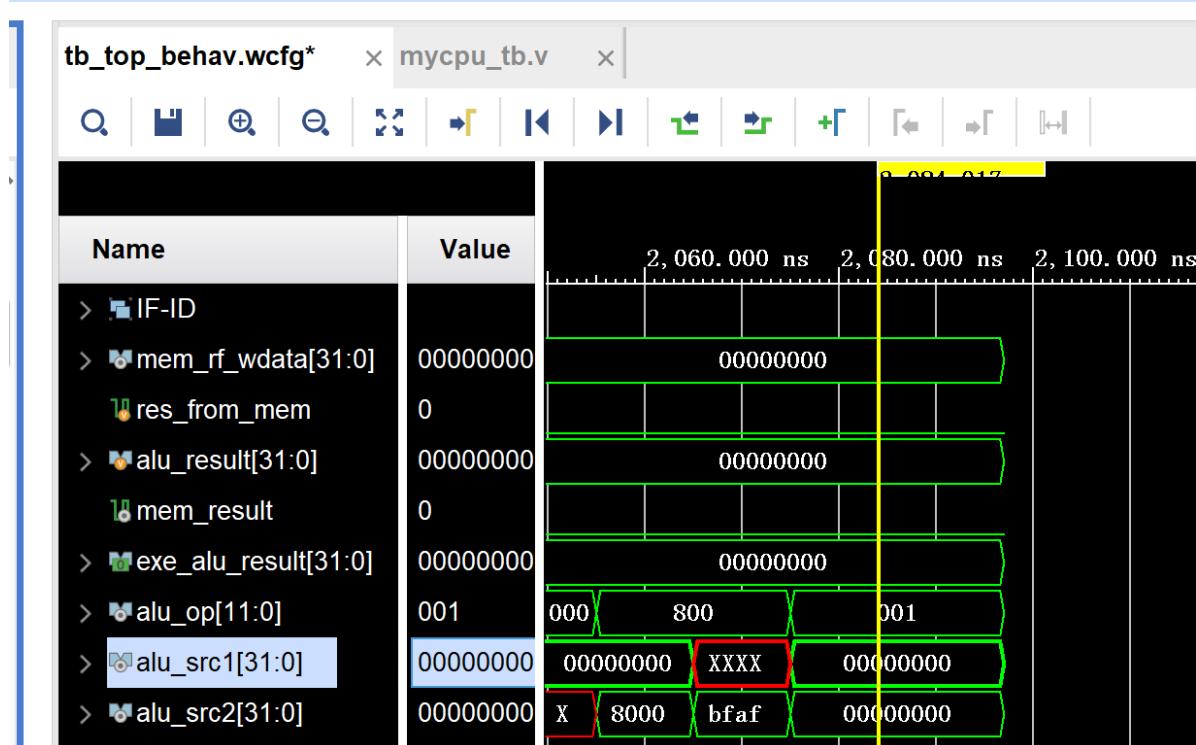
##### (1) 错误现象

写回数据不对

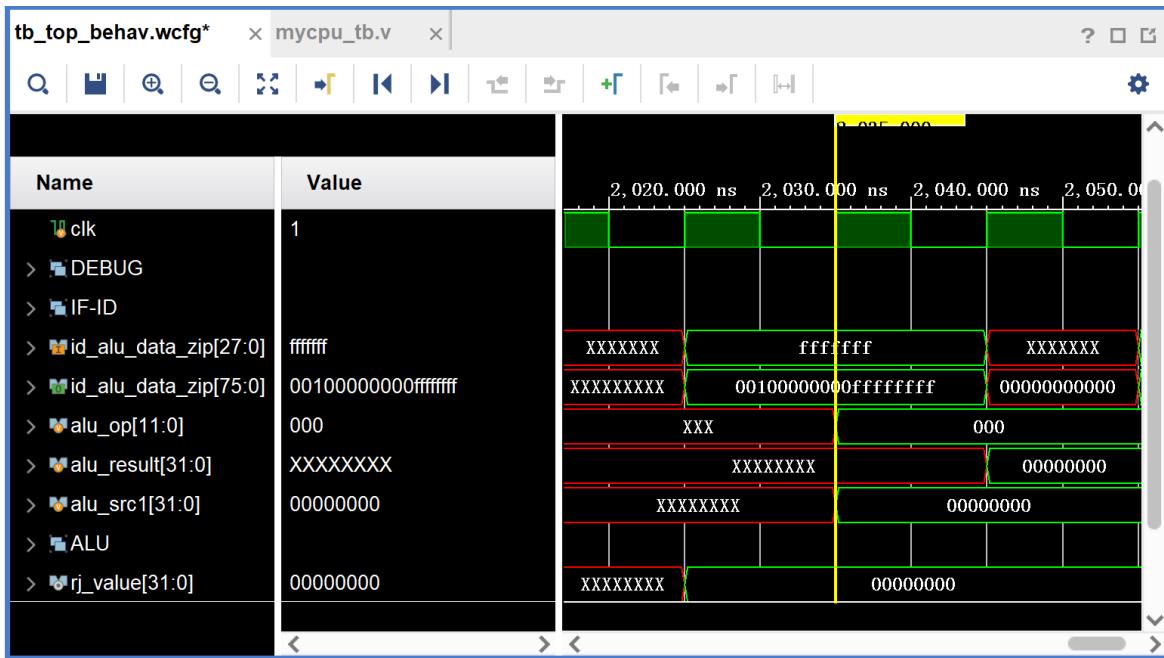
```
-----  
Test begin!  
-----  
[ 2057 ns] Error!!!  
    reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff  
    mycpu      : PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000  
-----
```

##### (2) 分析定位过程

回溯到MEM阶段传给WB阶段的写回数据：



res\_from\_mem为0，结果来自ALU，而ALU的两个源操作数都为0，定位到ID阶段对ALU操作数的确定：



### (3) 错误原因

EXE阶段接受ID阶段的ALU相关data位宽不对。

### (4) 修正效果

来到下一个bug：

```
Tcl Console  x Messages  Log  ?
Q  H  D  I  S  B  M  W  ?
```

```
[ 2667 ns] Error!!!
reference: PC = 0x1c04f728, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000001
mycpu : PC = 0x1c00f15c, wb_rf_wnum = 0x01, wb_rf_wdata = 0x1c00f160

$finish called at time : 2707 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_7/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 169
```

### (5) 归纳总结（可选）

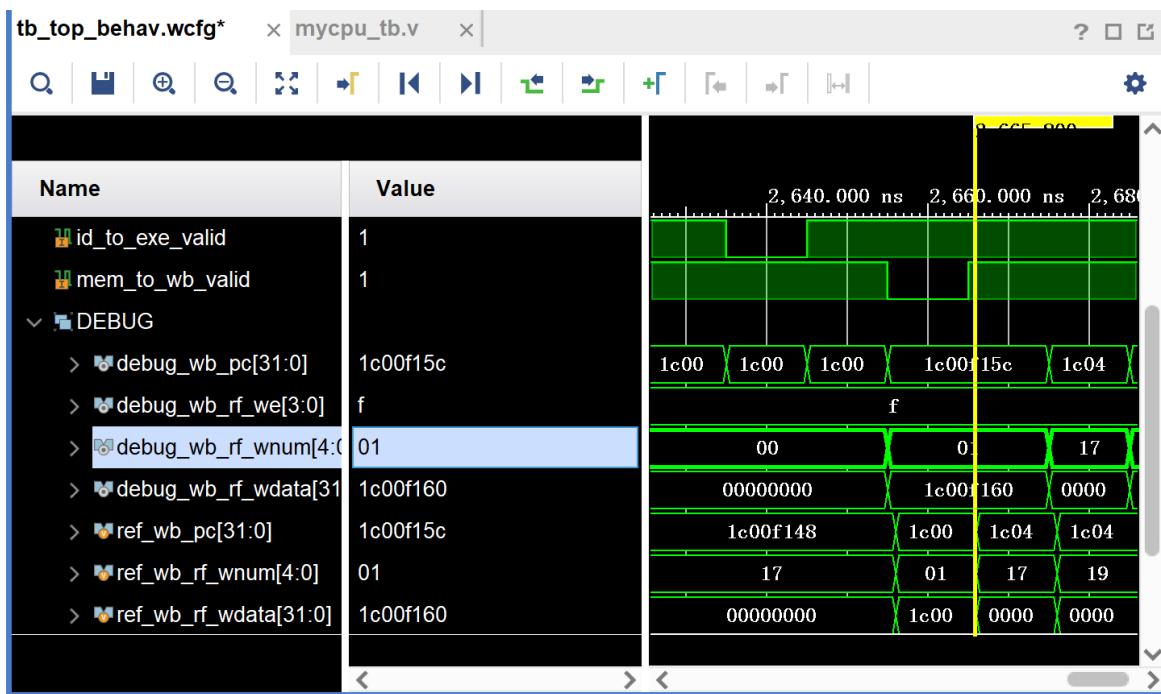
## 错误5：分支取消处理不当

### (1) 错误现象

如上，PC未在跳转指令到来时正常分支

### (2) 分析定位过程

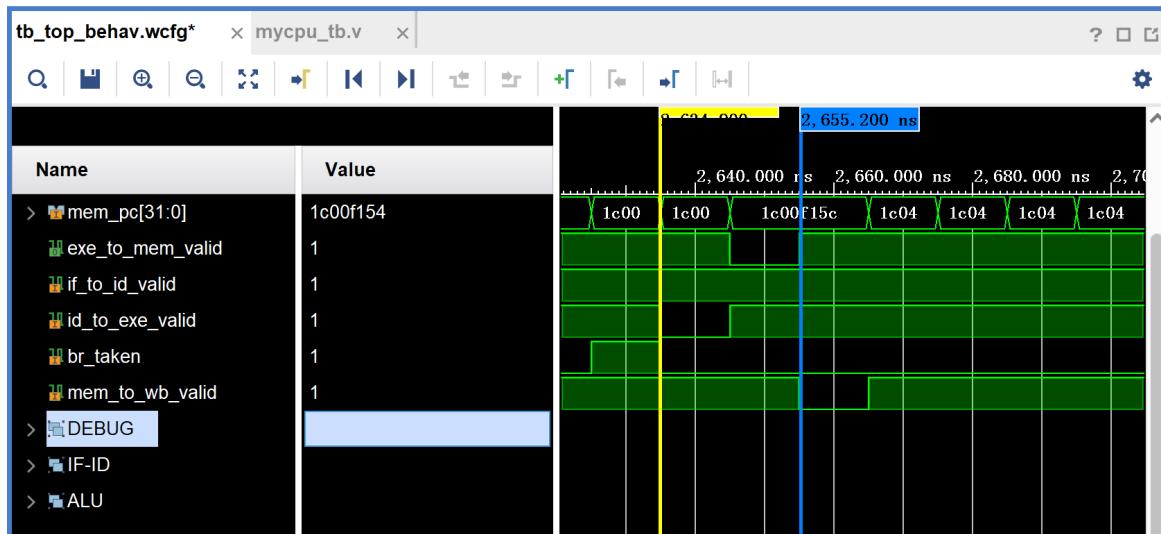
检查debug相关信号：



发现光标所示处写回的PC值未发生改变，同时写使能拉高，导致金标准提前更新。检查exe\_pc，发现其在marker处未更新：



PC值的更新与前一个阶段传来的valid信号有关，定位到id阶段存在一次取指取消的情况：



```

assign id_ready_go      = 1'b1;
assign id_allowin       = ~id_valid | id_ready_go & exe_allowin;
assign id_to_exe_valid = id_valid & id_ready_go;
always @(posedge clk) begin
    if(~resetn)
        id_valid <= 1'b0;
    else
        // 有效条件: if向id输入有效、非转移指令、id允许接收数据
        id_valid <= if_to_id_valid & ~br_taken & id_allowin;
end

```

### (3) 错误原因

当前流水线CPU遇到分支取消的情况会停滞一个周期，此时 debug\_wb\_rf\_we 应该被取消。

### (4) 修正效果

此种错误有两种修改方法：

- 将 debug\_wb\_rf\_we 改为寄存器类型，并在拉高后的下一个时钟上升沿将之拉低；
- 将 debug\_wb\_rf\_we 改为：

```
assign debug_wb_rf_we = {4{rf_we & wb_valid}};
```

后者更简易，采取后者，修改后pass第一个测试点：

```

Test begin!
[ 22000 ns] Test is running, debug_wb_pc = 0x1c051558
----[ 27925 ns] Number 8'd01 Functional Test Point PASS!!!
-----
[ 28127 ns] Error!!!
    reference: PC = 0x1c00f35c, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
    mycpu   : PC = 0x1c00f35c, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
-----
$finish called at time : 28167 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_7/soc_verify/soc_bram/testbench/mycpu_tb.v" L

```

## 错误6：隐式变量定义

### (1) 错误现象

如图，上述写回数据有误。

### (2) 分析定位过程

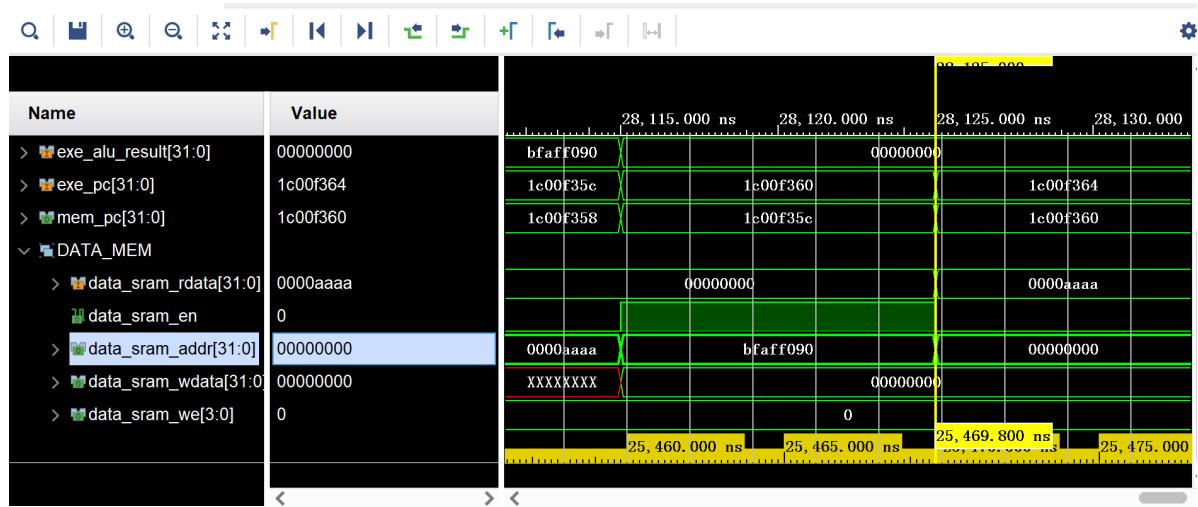
查看反汇编代码：

```

1c00f350: 02800000 addi.w $r0,$r0,0
1c00f354: 02800000 addi.w $r0,$r0,0
1c00f358: 02aaa9ad addi.w $r13,$r13,-1366(0xaa)
1c00f35c: 2880018e ld.w   $r14,$r12,0
1c00f360: 02800000 addi.w $r0,$r0,0

```

是load类指令，观察读数据过程：



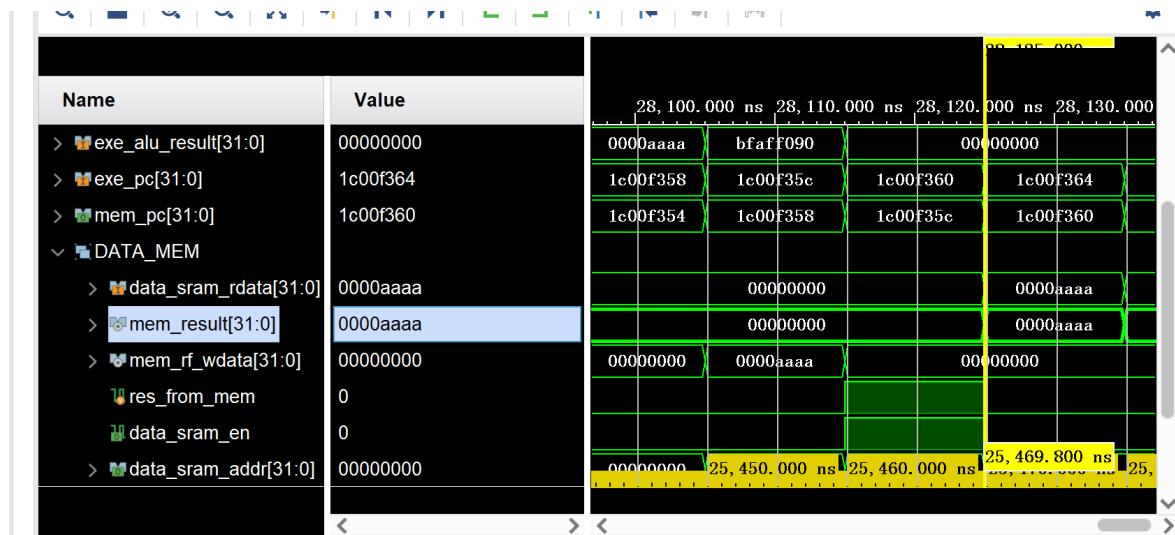
发现数据正确读出，但未能正确传给下一流水级。查看交互部分：

```
assign mem_rf_wdata      = res_from_mem ? mem_result : alu_result;
assign mem_result      = data_sram_rdata;
```

### (3) 错误原因

mem\_result未定义，默认置为1位宽的wire类型。

### (4) 修正效果



## 错误7：未提前申请取数据

### (1) 错误现象

由上述图可知，此次位宽正确，但产生的写回数据（mem\_rf\_data）依旧不正确，

### (2) 分析定位过程&错误原因

此次的CPU设计是基于bram，其读数据也是同步与于时钟的，因而在下一个周期才可以拿到正确的读数据，而下一周期res\_from\_mem已经拉低，导致信号错开。

### (4) 修正方法&效果

类比于pre-IF阶段，也可设置一个预取数的环节，当还处在EXE阶段时就发出读数要求，相应的其余内存相关的信号也都应该提前给出：

```

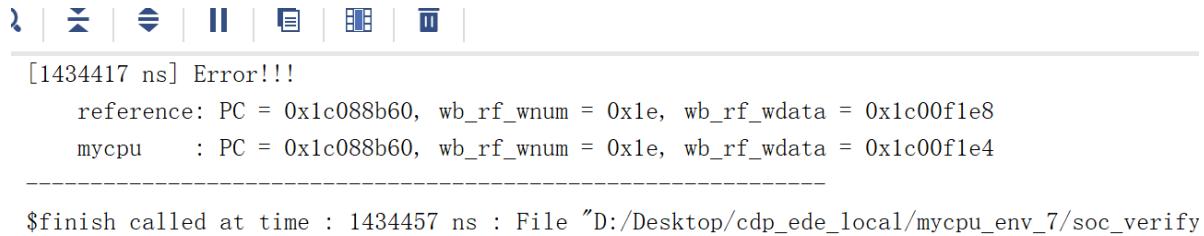
assign data_sram_en      = exe_res_from_mem || exe_mem_we;
assign data_sram_we      = {4{exe_mem_we}};
assign data_sram_addr    = exe_alu_result;
assign data_sram_wdata   = exe_rkd_value;

```

## 错误8：信号滞后导致的数据相关

### (1) 错误现象

写回数据有误：



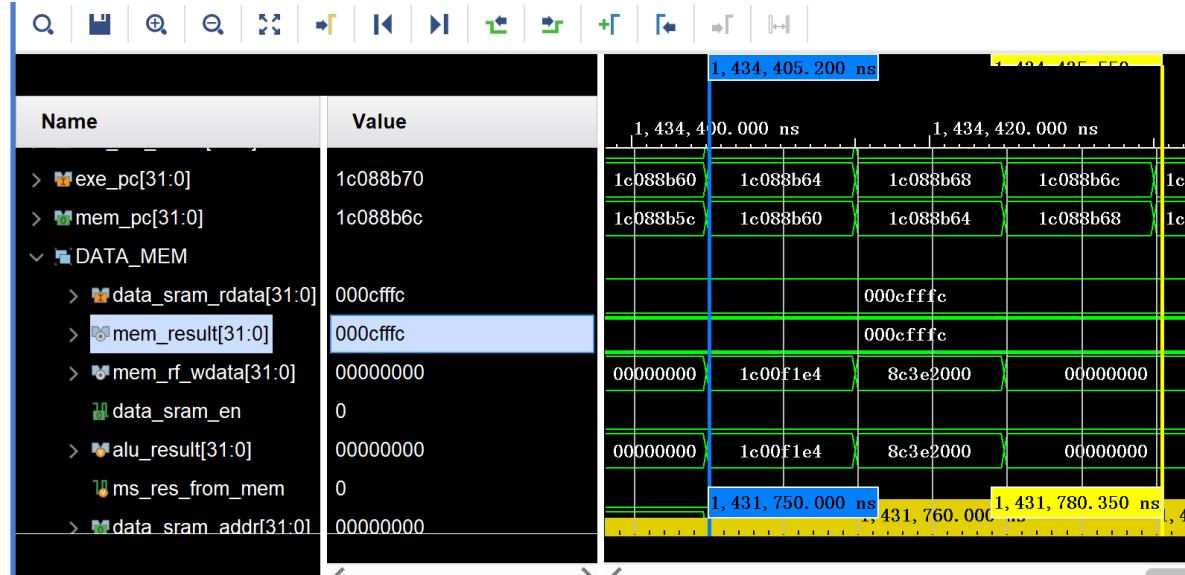
```

[1434417 ns] Error!!!
reference: PC = 0x1c088b60, wb_rf_wnum = 0x1e, wb_rf_wdata = 0x1c00f1e8
mycpu    : PC = 0x1c088b60, wb_rf_wnum = 0x1e, wb_rf_wdata = 0x1c00f1e4
-----
$finish called at time : 1434457 ns : File "D:/Desktop/cdp_ebe_local/mycpu_env_7/soc_verify

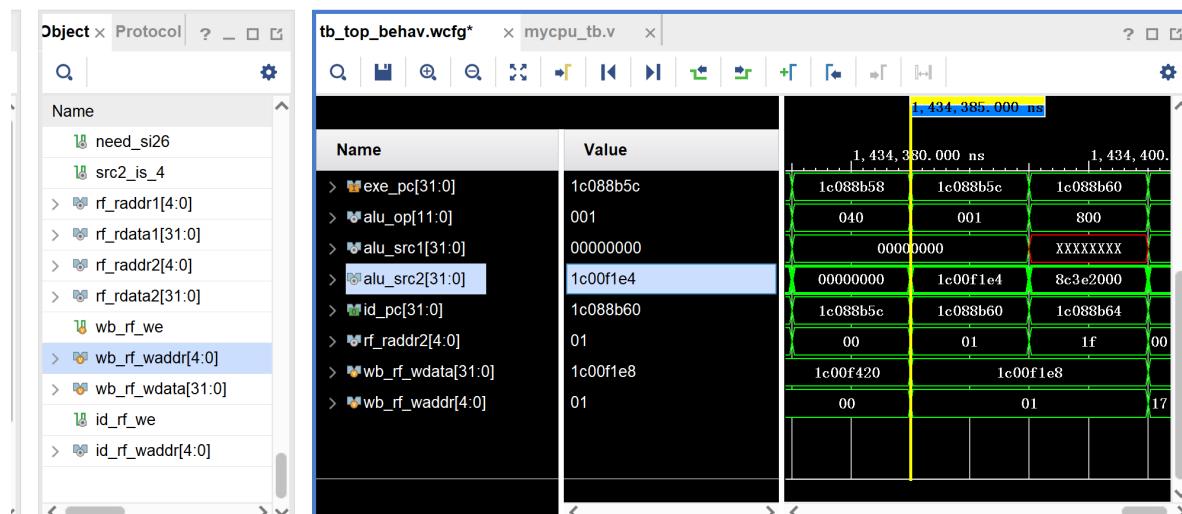
```

### (2) 分析定位过程

查看写回数据来源，得知结果来自alu，如图中marker所示：



查看ID阶段解析出的ALU的相关信号：



### (3) 错误原因

如图所示，此时在写回地址恰为01号寄存器且读数地址也在01号寄存器时，出现了写后读相关的问题

跟踪一下指令流：

```
5// 1c00f1e4: 548a1400 b1 355/2(0x8a14) # 1c01/cd0 <n1/_one_test>
578 1c00f1e0: 54013c00 b1 316(0x13c) # 1c00f31c <idle_1s>
579 1c00f1e4: 57997401 b1 498036(0x79974) # 1c088b58 <n18_bl_test>
580 1c00f1e8: 54013400 b1 308(0x134) # 1c00f31c <idle_1s>
581 1c00f1ec: 542cbc01 b1 273596(0x42cbc) # 1c051ea8 <n19_jirl_test>
```

```
1c088b58 <n18_bl_test>:
n18_bl_test():
1c088b58: 028006f7 addi.w $r23,$r23,1(0x1)
1c088b5c: 00150019 move $r25,$r0
1c088b60: 0010041e add.w $r30,$r0,$r1
```

从 1c00f1e4 (bl写r1寄存器) 跳转到 1c088b58，之后add.w指令使用了r1寄存器中的数据。

后在piazza提问后意识到，在bl处于译码阶段时，理应block了流水线，使得后续流入的add.w指令与之的间隔恰足以使得数据写回。反观wb阶段和id阶段的交互：

ID阶段：

```
//写回阶段传回数据处理
always @(posedge clk) begin
    {wb_rf_we, wb_rf_waddr, wb_rf_wdata} <= wb_rf_zip;
end
```

WB阶段：

```
assign wb_rf_zip = {rf_we, rf_waddr, rf_wdata};
```

在ID阶段应该以组合逻辑接受WB发来的数据。

#### (4) 修正方法&效果

HAPPY ENDING：



```
[1752000 ns] Test is running, debug_wb_pc = 0x1c078b68
[1762000 ns] Test is running, debug_wb_pc = 0x1c079ca0
[1772000 ns] Test is running, debug_wb_pc = 0x1c07ada8
[1782000 ns] Test is running, debug_wb_pc = 0x1c07bec8
[1792000 ns] Test is running, debug_wb_pc = 0x1c07cf0
[1802000 ns] Test is running, debug_wb_pc = 0x1c07e0c0
[1812000 ns] Test is running, debug_wb_pc = 0x1c07f1f8
[1822000 ns] Test is running, debug_wb_pc = 0x1c080300
[1832000 ns] Test is running, debug_wb_pc = 0x1c0813f0
----[1838305 ns] Number 8'd20 Functional Test Point PASS!!!
=====
Test end!
----PASS!!!
$finish called at time : 1839455 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_7/soc_verify/soc_bram/test.sv"
run: Time (s): cpu = 00:00:39 ; elapsed = 00:00:58 . Memory (MB): peak = 1112.215 ; gain = 0.000
```

### (三) 实践任务9：前递技术解决相关引发的冲突

在实验8、9中需考虑数据相关的情况，需要将EXE、MEM、WB阶段的写寄存器相关数据传回ID阶段，于此有两种设计思路：

- 将CPU和数据RAM交互的部分置于MEM阶段，这将导致CPU在EXE阶段无法提前读到数据RAM中的值，需在某些指令并排时阻塞一周期（如LD紧跟其余寄存器相关指令，在EXE阶段LD指令暂无法拿到内存中数据，还需等待至MEM阶段取到内存操作数）
  - 将数据RAM的模块转移至EXE部分，并提前发送读请求，做数据预取，此番在EXE阶段即可拿到内存中操作。

我在最初设计时采取了后者，这样MEM可与EXE阶段合并，但后续发现此种做法会导致ID阶段的逻辑级数较高，对后续设计并不友好，故又写了一版第一种处理方式，在9.2中也给出了性能比对。

## 9.1 初版思路：ALU计算提前

将ALU计算前提的思路并非我一下就想出的，而是在后续发现数据需提前给出时思考出的一种处理方式。

这种方式处理非常简单，完全无需考虑阻塞的情况，写的时候也是一马平川，没有遇到非常多的bug。

### 错误1：信号位宽定义有误

### (1) 错误现象

写回数据有误：

```
run all
=====
Test begin!
-----
[ 12197 ns] Error!!!
    reference: PC = 0xlc00f1e4, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
    mycpu     : PC = 0xlc00f1e4, wb_rf_wnum = 0x0e, wb_rf_wdata = 0xbfaaff090
-----
```

## (2) 分析定位过程

查看EXE阶段对应pc生成的写回数据，发现EXE的数据变化与ID传来的值不一致：

The timing diagram illustrates the logic flow between three signals:

- id\_rf\_zip[5:0]**: Address signal, starting at 2f and transitioning to 00.
- exe\_rf\_zip\_tmp**: Control signal, starting at 1 and transitioning to 0.
- id\_to\_exe\_valid**: Output signal, which is asserted (high) during the transition period from 2f to 00.

| Time Step | id_rf_zip[5:0] | exe_rf_zip_tmp | id_to_exe_valid |
|-----------|----------------|----------------|-----------------|
| 1         | 2f             | 1              | 0               |
| 2         | 2d             | 0              | 1               |
| 3         | 2e             | 0              | 1               |
| 4         | 2f             | 0              | 0               |
| 5         | 00             | 0              | 0               |

exe rf zip tmp = {exe rf we, exe rf waddr}

查看exe阶段相关信号：

```
reg [31:0] exe_rkd_value ;
reg          exe_res_from_mem;
reg          exe_mem_we      ;
reg          exe_rf_we      ;
reg [5 :0]  exe_rf_waddr  ;
```

### (3) 错误原因

exe\_rf\_waddr位宽定义错误。

#### (4) 修正效果

修正其位宽为4后，来到新bug：

```
| Test begin!
| ----[ 12105 ns] Number 8'd01 Functional Test Point PASS!!!
| -----
| [ 12197 ns] Error!!!
|   reference: PC = 0x1c00f1e4, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
|   mycpu    : PC = 0x1c00f1e4, wb_rf_wnum = 0x0e, wb_rf_wdata = Oxxxxxxxxx
| -----
| $finish called at time : 12237 ns : File "D:/Desktop/cdp_ede_local/mycpu_env_8/soc_verify/soc_bram/testbench/mycpu_tb.v" L
```

#### (5) 归纳总结（可选）

在将信号打包处理后需要格外留意各个信号的位宽是否定义正确，否则在后续拆包的时候可能导致信号连环错误，较难定位出错误原因。

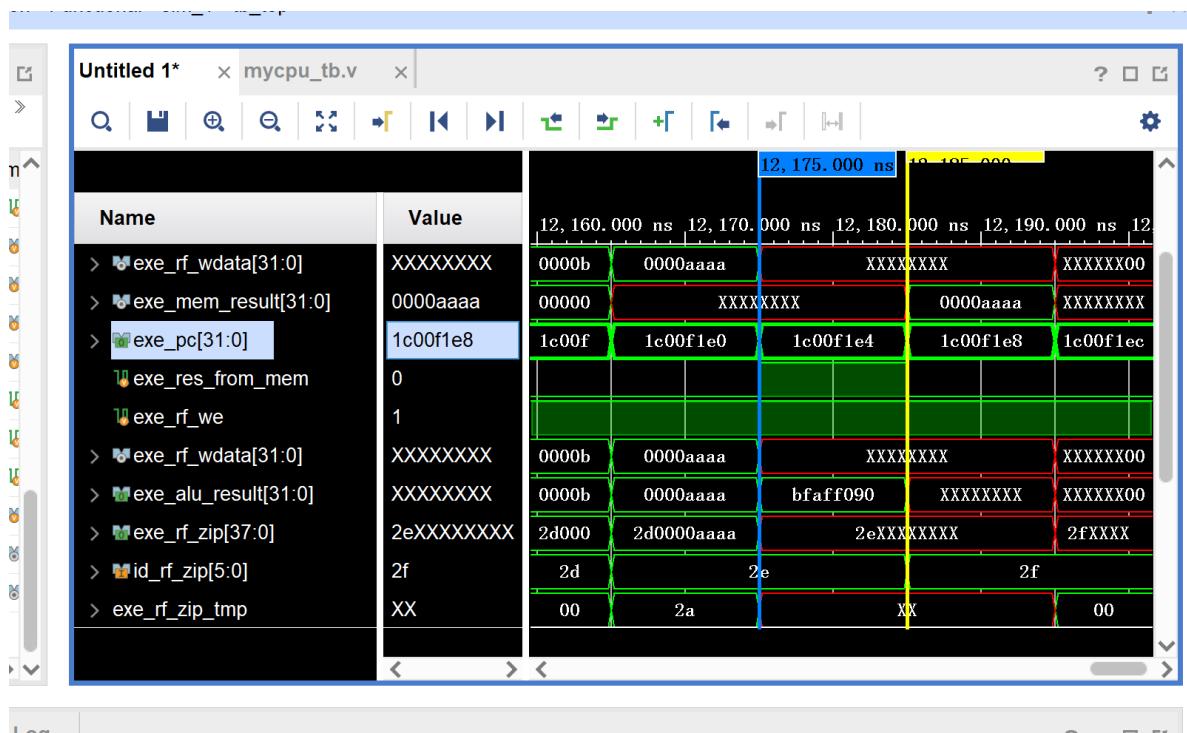
### 错误2：未进行数据预取&提前计算读数地址

#### (1) 错误现象

如上，写回数据未定义。

#### (2) 分析定位过程

查看此时EXE阶段PC=0x1c00f1e4的时刻（图中蓝色marker处）：



#### (3) 错误原因

原本的EXE接受信号逻辑：

```

always @(posedge clk) begin
    if(id_to_exe_valid & exe_allowin)
        {exe_alu_op, exe_alu_src1, exe_alu_src2} <= id_alu_data_zip;
end
always @(posedge clk) begin
    if(id_to_exe_valid & exe_allowin)
        {exe_res_from_mem, exe_mem_we, exe_rkd_value, exe_rf_we,
exe_rf_waddr} <= {id_res_from_mem, id_mem_we, id_rkd_value, id_rf_zip};
end

assign data_sram_en      = (exe_res_from_mem || exe_mem_we) & id_to_exe_valid
& exe_allowin;
assign data_sram_we      = {4{exe_mem_we & id_to_exe_valid & exe_allowin}};
assign data_sram_addr    = exe_alu_result;
assign data_sram_wdata   = exe_rkd_value;

```

由于在蓝色marker处才发出了读信号，导致数据在下一个时钟上升沿才到达（黄色光标处），导致选择数据错开，故应收到来自ID阶段的读使能时就向数据RAM发送读请求，同时给出读地址（相应的ALU的计算也应该提前），注意此时还需保证ID阶段有效且EXE阶段允许流入：

```

alu u_alu(
    .alu_op      (id_alu_op      ),
    .alu_src1    (id_alu_src1    ),
    .alu_src2    (id_alu_src2    ),
    .alu_result  (id_alu_result)
);

//-----data sram interface-----
-----

assign data_sram_en      = (id_res_from_mem|| exe_mem_we);
assign data_sram_we      = {4{id_mem_we}};
assign data_sram_addr    = id_alu_result;
assign data_sram_wdata   = id_rkd_value;
assign exe_mem_result    = data_sram_rdata;
assign exe_rf_wdata      = exe_res_from_mem ? exe_mem_result :
exe_alu_result;
assign exe_rf_zip        = {exe_rf_we, exe_rf_waddr, exe_rf_wdata};

```

同时注意到此时ALU所使用的信号都来自ID阶段，可把ALU模块也转移至ID模块内。

#### (4) 修正效果

修正后可正确工作：

```

[ 592000 ns] Test is running, debug_wb_pc = 0x1c031414
[ 602000 ns] Test is running, debug_wb_pc = 0x1c032300
----[ 603615 ns] Number 8'd20 Functional Test Point PASS!!!
=====
Test end!
----PASS!!!
$finish called at time : 604055 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_8/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:14 ; elapsed = 00:00:21 . Memory (MB): peak = 2450.906 ; gain = 0.000

```

## 9.2 前递+阻塞 vs ALU计算提前

在Piazza询问后，我意识到了上述处理方法有其不当之处

Piazza的Q&A: <https://piazza.com/class/l742qgidoxl54g/post/153>

原本采取ALU计算前提的时序报告：

| Design Timing Summary                          |             |            |   |             |            |   |                    |              |  |  |
|--|-------------|------------|---|-------------|------------|---|--------------------|--------------|--|--|
| General Information                            |             |            | Setup                                       |             |            | Hold                                    |                    |              | Pulse Width                              |  |
| Timer Settings                                 |             |            | Worst Negative Slack (WNS): <b>0.730 ns</b> |             |            | Worst Hold Slack (WHS): <b>0.113 ns</b> |                    |              | Worst Pulse Width Slack (WPW)            |  |
| Design Timing Summary                          |             |            | Total Negative Slack (TNS): <b>0.000 ns</b> |             |            | Total Hold Slack (THS): <b>0.000 ns</b> |                    |              | Total Pulse Width Negative Slack (TPWNS) |  |
| Clock Summary (4)                              |             |            | Number of Failing Endpoints: <b>0</b>       |             |            | Number of Failing Endpoints: <b>0</b>   |                    |              | Number of Failing Endpoints: <b>0</b>    |  |
| > Check Timing (86)                            |             |            | Total Number of Endpoints: <b>7787</b>      |             |            | Total Number of Endpoints: <b>7787</b>  |                    |              | Total Number of Endpoints: <b>7787</b>   |  |
| All user specified timing constraints are met. |             |            |   |             |            |   |                    |              |  |  |
| Paths  | Requirement | Path Delay | Logic Delay                                 | Net Delay   | Clock Skew | Slack                                   | Clock Relationship | Logic Levels | Routes                                   |  |
| Path #1  | 20.000      | 14.484     | 3.813(27%)                                  | 10.671(73%) | -0.145     | 4.841                                   | Safely Timed       | 19           | C  |  |
| Path #2  | 20.000      | 14.484     | 3.813(27%)                                  | 10.671(73%) | -0.145     | 4.841                                   | Safely Timed       | 19           | C  |  |
| Path #3  | 20.000      | 14.484     | 3.813(27%)                                  | 10.671(73%) | -0.145     | 4.841                                   | Safely Timed       | 19           | C  |  |
| Path #4  | 20.000      | 14.484     | 3.813(27%)                                  | 10.671(73%) | -0.145     | 4.841                                   | Safely Timed       | 19           | C  |  |
| Path #5  | 20.000      | 14.484     | 3.813(27%)                                  | 10.671(73%) | -0.145     | 4.841                                   | Safely Timed       | 19           | C  |  |

可见暂时未出现违约的情况，逻辑级数为19，Path Delay为14.484ns。目前是时序还算看得过去，但后续我再加上乘除法指令就会出现违例的情况，故考虑尝试讲义给出的划分五级流水的方法。

在原有基础上，按照讲义给出的设计方法修改源代码，在此种设计方式下，大部分的冲突可以通过前递解决，只有ld类指令相关的指令才需阻塞一周期。

修改过程中并没有遇到太多的bug，主要是信号未定义以及各流水级接口未修改好的问题，接着xvlogi.log的提示修改基本可以解决，此处不再赘述。修改后时序报告如下：

| Design Timing Summary                          |             |            |   |            |            |  |                    |              |        |
|--|-------------|------------|---|------------|------------|--|--------------------|--------------|--------|
| Setup  |             |            | Hold                                    |            |            | Pulse Width                              |                    |              |        |
| Worst Negative Slack (WNS): <b>6.684 ns</b>    |             |            | Worst Hold Slack (WHS): <b>0.053 ns</b> |            |            | Worst Pulse Width Slack (WPW)            |                    |              |        |
| Total Negative Slack (TNS): <b>0.000 ns</b>    |             |            | Total Hold Slack (THS): <b>0.000 ns</b> |            |            | Total Pulse Width Negative Slack (TPWNS) |                    |              |        |
| Number of Failing Endpoints: <b>0</b>          |             |            | Number of Failing Endpoints: <b>0</b>   |            |            | Number of Failing Endpoints: <b>0</b>    |                    |              |        |
| Total Number of Endpoints: <b>7933</b>         |             |            | Total Number of Endpoints: <b>7933</b>  |            |            | Total Number of Endpoints: <b>7933</b>   |                    |              |        |
| All user specified timing constraints are met. |             |            |   |            |            |  |                    |              |        |
| Paths  | Requirement | Path Delay | Logic Delay                             | Net Delay  | Clock Skew | Slack                                    | Clock Relationship | Logic Levels | Routes |
| Path #1  | 10.000      | 3.156      | 2.333(74%)                              | 0.823(26%) | -0.145     | 6.684                                    | Safely Timed       | 9            | 0   FD |
| Path #2  | 10.000      | 3.137      | 2.314(74%)                              | 0.823(26%) | -0.145     | 6.703                                    | Safely Timed       | 9            | 0   FD |
| Path #3  | 10.000      | 3.064      | 2.241(74%)                              | 0.823(26%) | -0.145     | 6.776                                    | Safely Timed       | 9            | 0   FD |
| Path #4  | 10.000      | 3.043      | 2.220(73%)                              | 0.823(27%) | -0.145     | 6.797                                    | Safely Timed       | 9            | 0   FD |
| Path #5  | 10.000      | 3.042      | 2.219(73%)                              | 0.823(27%) | -0.145     | 6.798                                    | Safely Timed       | 8            | 0   FD |
| Path #6  | 10.000      | 3.023      | 2.200(73%)                              | 0.823(27%) | -0.145     | 6.817                                    | Safely Timed       | 8            | 0   FD |
| Path #7  | 10.000      | 3.050      | 2.197(73%)                              | 0.822(27%) | -0.145     | 6.800                                    | Safely Timed       | 8            | 0   FD |

逻辑级数仅为9，Path Delay为3.156ns，时序果真好了很多。

## 四.实验总结

本次实验将单周期CPU改造为流水线CPU，同时学习了数据相关的产生原因与解决方法，如阻塞技术和前递技术等。

此次实验还让我更进一步理解合理流水级划分的重要性，尝试了四流水划分（将ALU计算提前至ID阶段，导致流水线划分不是很均匀）和五流水划分，可以很明显看到流水级划分得好可以使逻辑级数大大降低、延迟较短。

此外，我还进一步认识到模块化以及规范化命名的好处。老师在课堂讲解时建议将各个模块拆分写，同时建议我们采取规范化命名，并将各模块交互信号尽可能封装成bus，此种处理方法使得逻辑更为清晰，也方便调试。唯一需要注意的是拆分模块时尤其需要注意在分模块中补全信号定义，否则遇到隐式变量声明时调试较为困难。