Lab 4 报告

学号 2021K8009929005
姓名 饶嘉怡
箱子号 5

一、实验任务（10%）

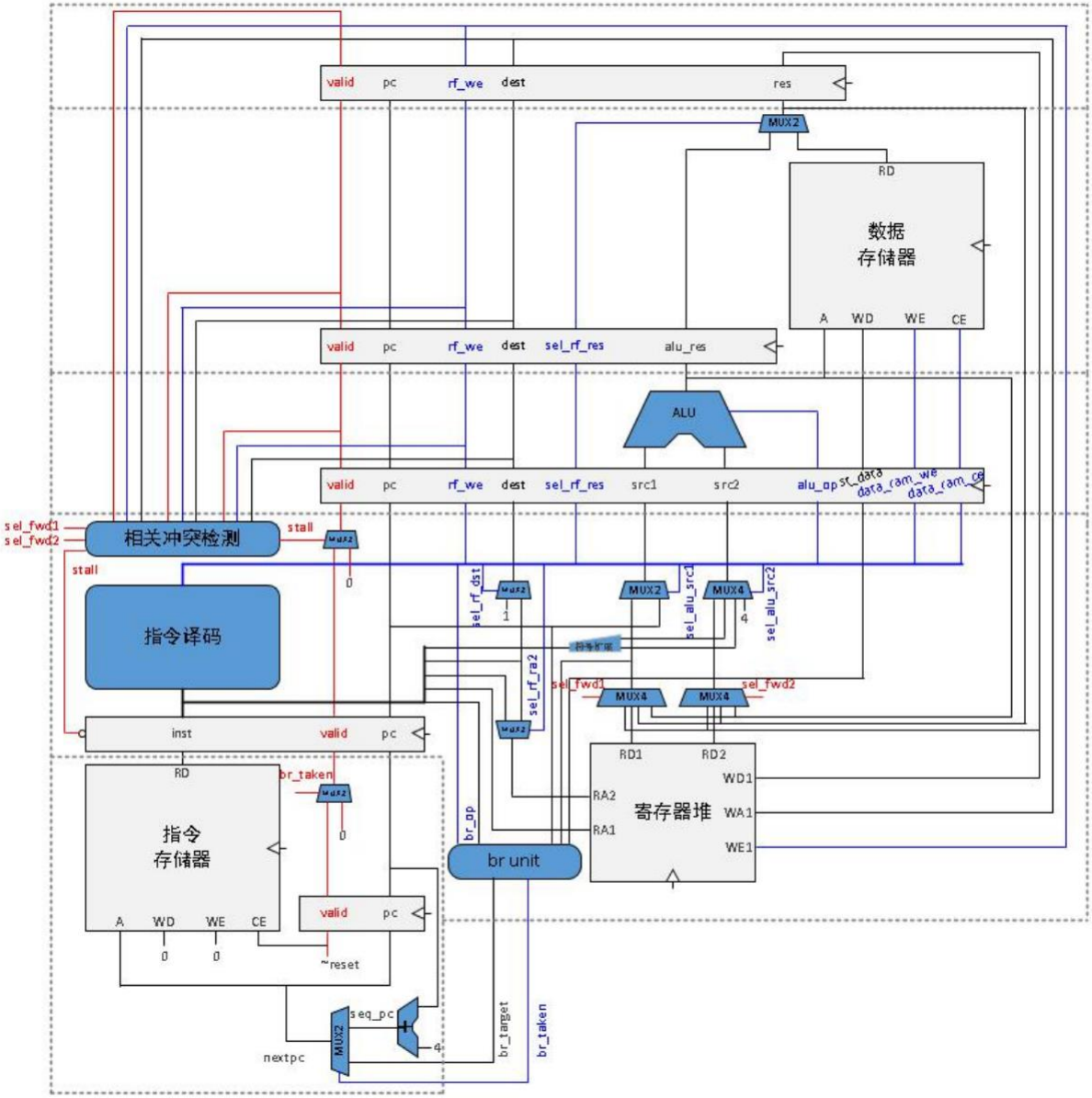
1.在单周期 CPU 的基础上引入流水线，但暂时不考虑处理各类相关引发的冲突；

2.在此基础上使用阻塞技术解决相关引发的冲突；

3.在此基础上使用前递技术解决相关引发的冲突。

二、实验设计（40%）

（一）总体设计思路



添加流水级的初衷是缩短时序期间之间组合逻辑关键路径的时延，在不降低电路处理吞吐率的情况下提升电路的时钟频率。该实验直接采用经典的但发生五级流水线，将之前设计的单发生 CPU 的数据通路拆分为五段，并在各段之间加入触发器作为流水线缓存。

而考虑到 nextpc 是在当前处于 IF 阶段的指令的 PC+4 和当前处于 ID 阶段的指令的跳转地址之间二选一，这有点属于某种“猜测”，当 br_taken=0 时，地址一直+4，猜测正确，而当 br_taken=1 时，猜测错误，但错误的那条指令已经进入了流水线 IF 阶段，这是控制相关引发的流水线冲突。解决方法是，阻止这条错误的指令从 IF 进入 ID，同时发射新的指令进入流水线，覆盖掉 IF 阶段这条错误的指令。整体效果就是，从 ID 开始后面每个阶段依次空一拍。

进一步考虑到，某一条指令在进入流水线时，前面的若干条指令可能尚未离开流水线，这样倘若这条指令需要用到前面若干条指令的写回结果，就会发生写后读冲突，这是数据相关引发的流水线冲突。由于结果在 WB 阶

段才会写回寄存器，但 ID 阶段就已经需要从寄存器中取数并准备好所有的源操作数。因此若该指令执行到 ID 阶段时，如果当前任意处在 EXE、MEM、WB 阶段的指令的写回寄存器号与该指令的读数寄存器号相同，意味着发生冲突。第一种处理思路是阻塞：阻止新的指令进入流水线，阻塞住 IF 和 ID 阶段往下流的通路，当这条引发冲突的指令离开流水线之后再继续正常流水。但是，实际上所有指令的写回数据在 EXE 阶段（需要用 ALU 计算结果的指令）和 MEM 阶段（ld 指令）就能够计算出来，只不过尚未统一写回。因此这样阻塞会浪费很多时间，不如设计一些数据通路，把这些“虽然已经算好但尚未写回”的数据前递给后续需要用到该数据的指令。这样除了一种情况——ID 阶段的指令与 EXE 阶段的指令冲突且 EXE 阶段是 ld 指令——无法前递数据，仍然需要阻塞一拍之外，其他的冲突情况均不需要阻塞。

为了实现所谓的阻塞功能，需要在流水级之间添加握手信号。信号是否能从流水级 A 传递给流水级 B 取决于两个信号：流水级 A 的 A2B_valid 信号和流水级 B 的 B_allowin 信号。前者表示 A 传递给 B 的内容是有效的，后者表示 B 允许 A 的数据流入。两者都是有必要的。当发生控制相关冲突时，IF 流向 ID 的内容无效，当发生数据相关冲突时，ID 流向 EXE 的内容无效，这些都需要 A2B_valid 信号实现。当发生数据相关冲突时，IF 和 ID 因为需要保存内容留以后用，不允许前面的流水级将数据流入，这需要 B_allowin 信号实现。除此之外，还有模块内部信号：A_valid 信号表示流水级 A 当前工作有效，A_ready_go 表示流水级 A 的内容下一拍可以往下流。

（二）重要模块 1 设计：IF 阶段

1、工作原理

由于该实验使用同步 RAM，也就是说在这一拍发送使能和读地址，到下一拍才能把指令读出来。理论上我们可以在 IF 阶段请求，ID 阶段读出进行译码。这在不考虑相关的情况下是非常容易实现的，？？？？？因此我们采用在 IF 阶段之前的 preIF 阶段发送使能和读地址，在 IF 阶段读出指令。

计算 nextpc，在复位拉低之后每一拍都将 nextpc 非阻塞赋值给 pc。从指令 RAM 中读取信号，取回指令。

2、接口定义

```
module IFstage (
  input wire clk,
  input wire resetn,
  input wire reset,

  output reg fs_valid,
  input wire ds_allowin,
  output wire fs2ds_valid,

  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,

  input wire [32:0] br_zip,

  output wire [63:0] fs2ds_bus
);
```

3、功能描述

从指令 RAM 中取指。

（三）重要模块 2 设计：ID 阶段

4、工作原理

ID 阶段将指令进行译码，从寄存器中读数并准备好源操作数，这一部分设计思路与单周期 CPU 相同，不再赘述。

而需要仔细讲述的是，如何在 ID 阶段判断是否发生冲突，以及进行阻塞。

下图是只使用阻塞技术时，ds_ready_go 的赋值逻辑。当 ID 阶段有效时，如果 EXE 阶段（或 MEM 或 WB）也有效且需要写回数据，写回的寄存器号与当前 ID 阶段读数的任何一个寄存器号相同，那么要把当前 ID 阶段的指令阻塞住，等那条与之冲突的指令离开流水线，ds_ready_go 自然就会拉高。/////

```
assign ds_ready_go    =
~(ds_valid && ( (es_valid && exe_gr_we && (exe_dest == rf_raddr1 || exe_dest == rf_raddr2))    ||
                (ms_valid && mem_gr_we && (mem_dest == rf_raddr1 || mem_dest == rf_raddr2))    ||
                (ws_valid && gr_we_reg  && (dest_reg  == rf_raddr1 || dest_reg  == rf_raddr2))      )   );
```

下图是加入前递技术时，ds_ready_go 的赋值逻辑。当 ID 阶段有效时，只有当 EXE 阶段有效且是 ld 指令，需要写回数据，写回的寄存器号与当前 ID 阶段读数的任何一个寄存器号相同，才需要将当前 ID 阶段的指令阻塞住。

并且 rj_value 和 rkd_value 的赋值逻辑也需要修改，与哪个阶段冲突就让哪个阶段前递回响应的结果。

```
assign ds_ready_go    =
~(ds_valid && ((es_valid && exe_gr_we && es_inst_is_ld_w &&  (exe_dest == rf_raddr1 || exe_dest == rf_raddr2))));
```

```
assign rj_value =
    (exe_gr_we && es_valid && exe_dest == rf_raddr1)? alu_result :
    (mem_gr_we && ms_valid && mem_dest == rf_raddr1)? final_result :
    (gr_we_reg && ws_valid  && dest_reg   == rf_raddr1)? rf_wdata   :
    rf_rdata1;
assign rkd_value =
    (exe_gr_we && es_valid && exe_dest == rf_raddr2)? alu_result :
    (mem_gr_we && ms_valid && mem_dest == rf_raddr2)? final_result :
    (gr_we_reg && ws_valid  && dest_reg   == rf_raddr2)? rf_wdata   :
    rf_rdata2;
```

同时，用 br_taken 信号判断并完成对控制相关的指令的冲突的阻塞。

```
always @(posedge clk) begin
  if (reset || br_taken) begin
    ds_valid <= 1'b0;
  end else if (ds_allowin) begin
    ds_valid <= fs2ds_valid;
  end
```

5、接口定义

```
module IDstage (
  input wire clk,
  input wire resetn,
  input wire reset,

  input wire fs_valid,
  output reg ds_valid,
  input wire es_allowin,
  output wire ds_allowin,
  input wire fs2ds_valid,
  output wire ds2es_valid,

  output wire [32:0] br_zip,
  output wire [178:0] ds2es_bus,
  input [37:0] ws2ds_bus,
  input wire [63:0] fs2ds_bus,
  input [38:0] exe_forward,
  input [38:0] mem_forward,
  input [6:0] wb_forward,

  input wire es_inst_is_ld_w,
  output wire inst_ld_w

);
```

6、功能描述

ID 阶段将指令进行译码，从寄存器中读数并准备好源操作数；判断是否冲突，进行阻塞。

（四）重要模块 3 设计：EXE 阶段

7、工作原理

EXE 阶段要向数据 RAM 发送使能和地址。

由于前递技术需要判断 EXE 阶段是否执行的是 ld 指令，此处要添加如下时序逻辑。

```
always @(posedge clk)begin
    if(ds2es_valid && es_allowin)begin
        if(inst_ld_w)begin
            es_inst_is_ld_w <= 1'b1;
        end
        else begin
            es_inst_is_ld_w <= 1'b0;
        end
    end
end
```

8、接口定义

```
module EXEstage (
  input wire clk,
  input wire resetn,
  input wire reset,

  input wire ds_valid,
  output reg es_valid,
  input wire ms_allowin,
  output wire es_allowin,
  input wire ds2es_valid,
  output wire es2ms_valid,

  output wire data_sram_en,
  output wire [3:0] data_sram_we,
  output wire [31:0] data_sram_addr,
  output wire [31:0] data_sram_wdata,
  input wire [31:0] data_sram_rdata,

  input wire [178:0] ds2es_bus,
  output wire [101:0] es2ms_bus,

  output [38:0] exe_forward,

  output reg es_inst_is_ld_w,
  input wire inst_ld_w
);
```

9、功能描述

判断当前是否为 ld 指令并告知 ID 阶段

（五）重要模块 4 设计：MEM 阶段

10、 工作原理

从数据 RAM 读出结果，与 ALU 的结果作一个二选一，得到最后要写回的结果。

11、 接口定义

```
module MEMstage (
  input wire clk,
  input wire resetn,
  input wire reset,

  input wire es_valid,
  output reg ms_valid,
  input wire ws_allowin,
  output wire ms_allowin,
  input wire es2ms_valid,
  output wire ms2ws_valid,

  input wire [101:0] es2ms_bus,
  output wire [165:0] ms2ws_bus,

  output [38:0] mem_forward

);
```

12、 功能描述

从数据 RAM 读出结果，与 ALU 的结果作一个二选一，得到最后要写回的结果

（六）重要模块 5 设计：WB 阶段

13、 工作原理

将结果写回寄存器。

14、 接口定义

```
module WBstage (
    input wire clk,
    input wire resetn,
    input wire reset,

    input wire ms_valid,
    output reg [31:0] ws_pc,
    output reg ws_valid,
    input wire ms_allowin,
    output wire ws_allowin,
    input wire es2ms_valid,
    output wire ms2ws_valid,



    input wire [165:0] ms2ws_bus,

    output [37:0] ws2ds_bus,

    output wire [31:0] debug_wb_pc,
    output wire [3:0] debug_wb_rf_we,
    output wire [4:0] debug_wb_rf_wnum,
    output wire [31:0] debug_wb_rf_wdata,

    output [6:0] wb_forward

);
```

15、 功能描述

将结果写回寄存器。

三、实验过程（50%）

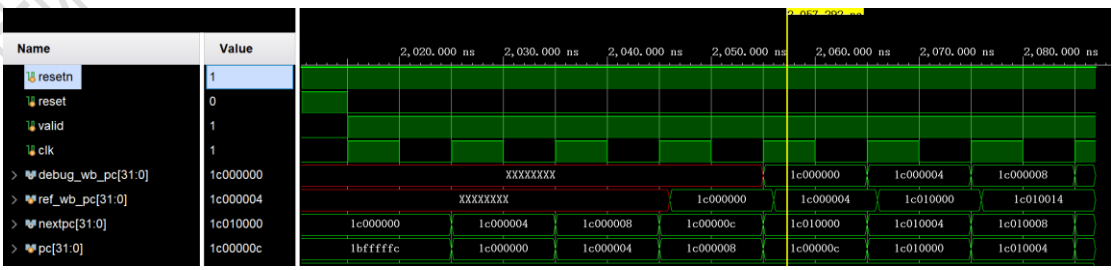
（一）实验流水账

略。

（二）错误记录

1、错误 1：金标准与指令错位

（1）错误现象



如图所示，金标准的 PC 信号 ref_wb_pc 比 debug_wb_pc 早了一拍，导致出错。

（2）分析定位过程

首先，确认到底是金标准晚了一拍，还是我的流水线整体早了一拍。由于 fs_valid 应当是与 valid 通过组合逻辑赋值保持相同的，因此我的流水线看起来是对的，那就是金标准早进了一拍。金标准何时读取是 debug_wb_rf_we 决定的，而 debug_wb_rf_we 又是通过 rf_we 赋值的。回看代码，发现 rf_we 用的是 MEM 阶段的 mem_gr_we 赋值的。由于 mem_gr_we 没有再流一拍变成 WB 阶段的 gr_we_reg，导致金标准在 WB 阶段率先使用了 MEM 阶段的写使能，因此早一拍进入流水线。

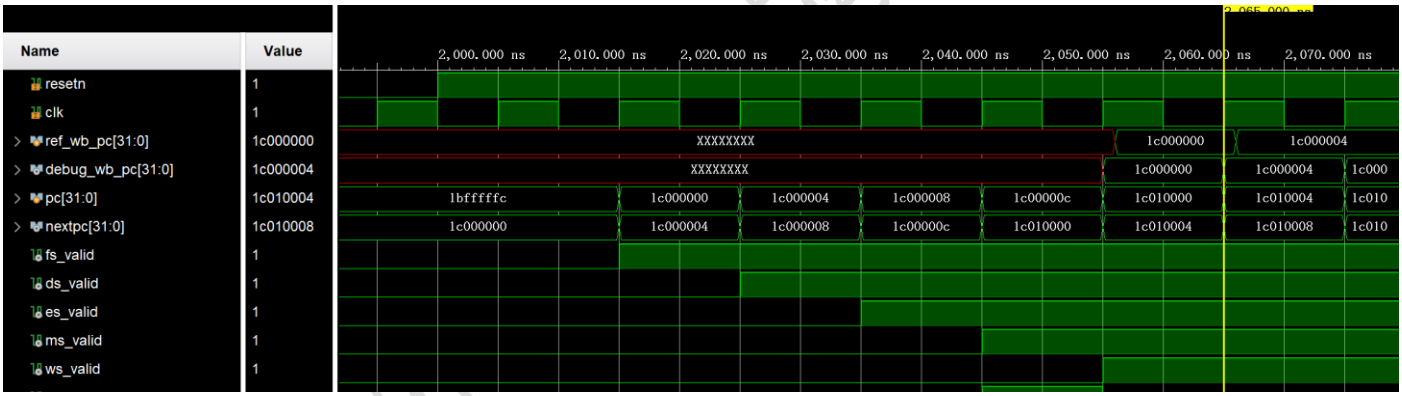
（3）错误原因

rf_we 是用 MEM 阶段的 mem_gr_we 赋值的，导致金标准在 WB 阶段率先使用了 MEM 阶段的写使能，因此早进入一拍。

（4）修正效果

将 rf_we 的赋值逻辑改为通过 gr_we_reg 赋值，得到了正确的与我的流水线同步的金标准。

```
assign rf_we = gr_we_reg && ws_valid;
assign debug_wb_rf_we = {4{rf_we}};
```



2、错误 2：从同步数据 RAM 中读数出错

（1）错误现象

在 PC=0x1c010588 时写回寄存器的数据出错。

```
[  35887 ns] Error!!!
    reference: PC = 0x1c010588, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
    mycpu    : PC = 0x1c010588, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
```

（2）分析定位过程

写回出错在 WB 阶段，向前回溯到 ID 阶段，发现 inst_ld_w 拉高，得知是 LD.W 指令，那么写回寄存器的数据应当是在 MEM 阶段访存，在 WB 阶段读到（由于是单周期 CPU，访存的数据何时取回尚且影响不大，因此我最开始是在 MEM 阶段请求访问数据 RAM，在 WB 阶段才取回）。rf_wdata 由 final_result_reg，也就是 MEM 阶段得到的 final_result 流了一拍后的结果赋值，final_result 在这条指令中由 mem_result 赋值，mem_result 是 data_sram_rdata 赋值得到的。仔细思考就很容易发现问题，因为访存的数据只有 WB 阶段才能读到，因此 mem_result 和 final_result 应该在 WB 得到而不是 MEM，也就是说应该直接用 final_result 给

rf_wdata 赋值，删除多余的流水。

（3）错误原因

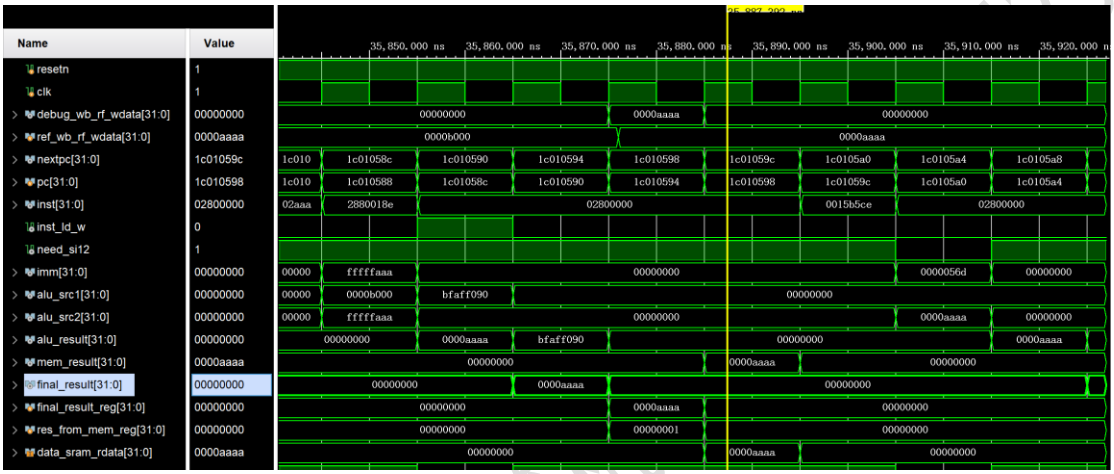
rf_wdata 由 final_result_reg 而不是 final_result 赋值，导致写回的其实是上一拍从内存中取回的数据。并且之所以会出现这样的错误，还因为我没有给 data_sram_en 正确的赋值逻辑，而是让其一直拉高，导致随时都能从内存读回数据。

（4）修正效果

修改了 rf_wdata 和 debug_rf_wdata，以及实现了 data_sram_en 的正确赋值逻辑，得到正确的波形。

```
assign rf_wdata = final_result;

assign data_sram_en    = mem_we_reg || exe_res_from_mem;
```



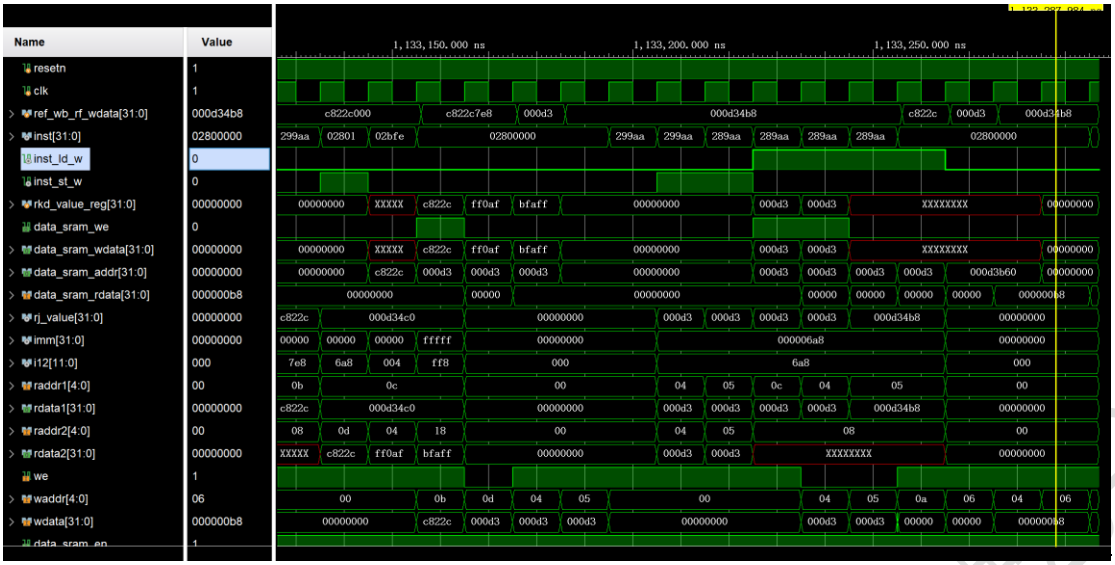
3、错误 3：RAM 写使能位宽错误

（1）错误现象

```
[1133257 ns] Error!!!
    reference: PC = 0x1c03128c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xc822c7e8
    mycpu    : PC = 0x1c03128c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x000000e8
```

在 PC=0x1c03128c 的时候写回寄存器的指令出错。

（2）分析定位过程



出错的指令是 LD.W，回看对应位置的 data_sram_rdata，发现从内存中读出来的数就是错的。于是向前追溯，在很近的地方就有 ST.W 指令。并且通过对比 ST.W 指令和 LD.W 指令的 data_sram_addr，发现二者相同，也就是说 LD.W 读出来的数就应该是 LD.W 写进去的数。我去查看 ST.W 指令对应的 data_sram_wdata，发现就是 0xc822c7e8，而读出来的数却是 0x000000e8。这个数错得很奇怪，给我一种我要么只写进去了最后一个字节，要么只读出来了最后一个字节，而前面的字节被抹掉了的感觉。为了印证这个猜想，我又看了相邻几个 LD.W 指令，发现它们与金标准相比，也只剩最后一个字节。金标准如下图所示，这几条指令都是 LD.W，但读到的数据分别是 000000e8、000000c4、000000b8、000000b8。

```
1 1c03128c 0a c822c7e8
1 1c031290 06 000d34c4
1 1c031294 04 000d34b8
1 1c031298 06 000d34b8
```

```
1c03128c:       289aa18a        ld.w    $r10,$r12,1704(0x6a8)
1c031290:       289aa086        ld.w    $r6,$r4,1704(0x6a8)
1c031294:       289aa0a4        ld.w    $r4,$r5,1704(0x6a8)
1c031298:       289aa0a6        ld.w    $r6,$r5,1704(0x6a8)
```

于是我推测，是在写入或读出的过程中，某个地方的位宽错误导致出错。翻看讲义，发现如下要求，于是狼狈地去改了两个 RAM 的写使能位宽。

2. 调整 CPU 顶层接口，将 `inst_sram_we` 和 `data_sram_we` 都从 1 比特的写使能调整为 4 比特的字节写使能。

（3）错误原因

没有把 data_sram_we 信号改成四位，导致在只有一位的情况下只能写进去最后一个字节，从而造成访存读数的时候只能读出来最后一个字节。

（4）修正效果

```
output wire [3:0] data_sram_we,
```

将 data_sram_we 和 inst_sram_we 的位宽修改为 4，不考虑相关冲突处理的流水线 CPU 实验 pass。

```
----[2023145 ns] Number 8'd20 Functional Test Point PASS!!!
==============================================================
Test end!
----PASS!!!
$finish called at time : 2024455 ns : File "D:/CA_local_depository/exp6-10/exp7/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
) run: Time (s): cpu = 00:01:02 ; elapsed = 00:01:29 . Memory (MB): peak = 1035.598 ; gain = 95.887
```

（5）归纳总结

一定要认真阅读讲义上的要求！可以少浪费很多不必要的时间！

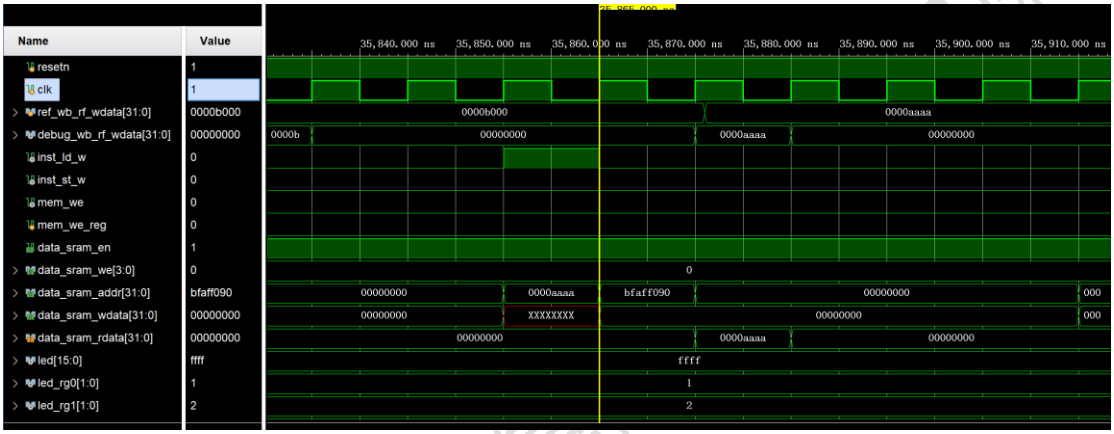
4、错误 4：修改访存时的错误

（1）错误现象

```
[  35887 ns] Error!!!
    reference: PC = 0x1c010588, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x0000aaaa
    mycpu    : PC = 0x1c010588, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
```

在 PC=0x1c10588 的时候写回寄存器的数据错误。

（2）分析定位过程



可以看到又是 LD.W 指令出错。data_sram_rdata 提前一拍写回了我想要的数。有了之前错误 2 的经验，我很快意识到这又是一个从数据 RAM 中读数的错误，是由于我将请求访存和读回数据都提前了一拍导致的。由于请求访存和读回数据都提前了一拍，在 WB 阶段的 mem_result 拿到的应该是 MEM 阶段从内存取出的数据在流一级水之后的结果。而我直接拿了 MEM 阶段从内存中取出的数据，导致错误。

（3）错误原因

给 mem_result 赋值时，直接拿了 MEM 阶段从内存中取出的数据，而不是 MEM 阶段从内存取出的数据在流一级水之后的结果。

（4）修正效果

```
461   assign mem_result    = data_sram_rdata_reg;
```

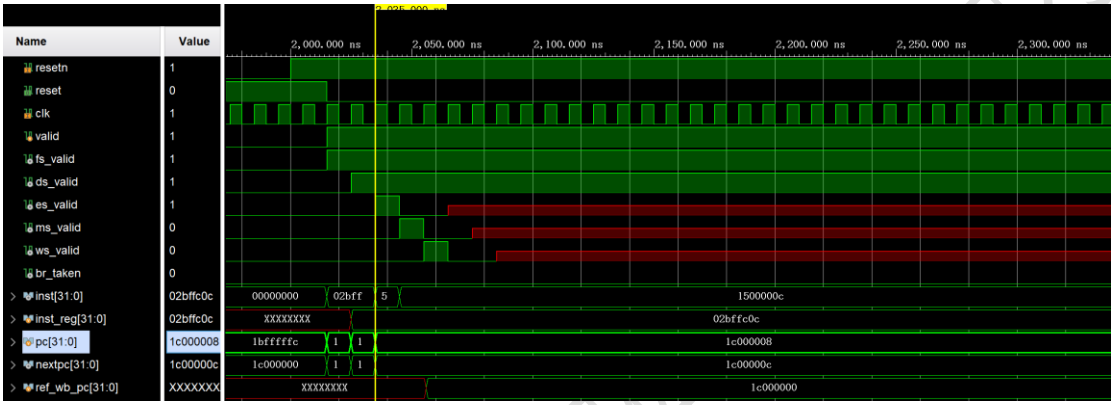
修改 mem_result 的赋值逻辑后，不再出错。

5、错误 5：阻塞之后无法恢复流水/////

（1）错误现象

```
Test begin!
[  22000 ns] Test is running, debug_wb_pc = 0x1c000000
[  32000 ns] Test is running, debug_wb_pc = 0x1c000000
[  42000 ns] Test is running, debug_wb_pc = 0x1c000000
[  52000 ns] Test is running, debug_wb_pc = 0x1c000000
[  62000 ns] Test is running, debug_wb_pc = 0x1c000000
[  72000 ns] Test is running, debug_wb_pc = 0x1c000000
[  82000 ns] Test is running, debug_wb_pc = 0x1c000000
[  92000 ns] Test is running, debug_wb_pc = 0x1c000000
[ 102000 ns] Test is running, debug_wb_pc = 0x1c000000
[ 112000 ns] Test is running, debug_wb_pc = 0x1c000000
```

具体表现为，仿真的时候 test 一直跑个不停，但 debug_wb_pc 从不变化，反映在波形上就是，阻塞之后恢复不了，流水线不再流水。



（2）分析定位过程

es_valid 是不定态，推测是给 es_valid 赋值的时候出错。但由于 debug 的时候只记录了错误，没有将错误原因记录下来，写实验报告的时候已经记不清了。

（3）归纳总结

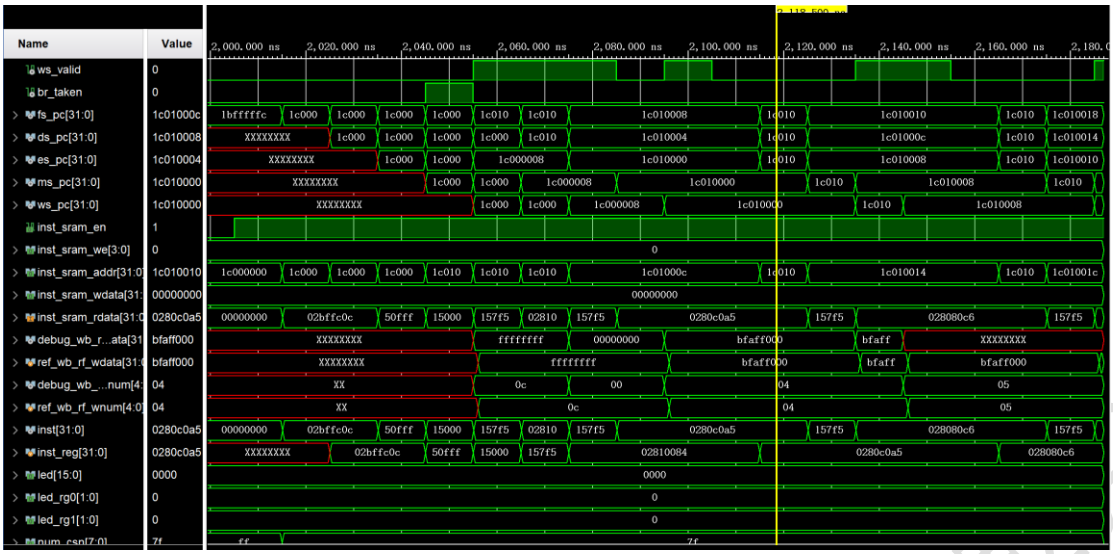
一定要养成 debug 的时候随手记录的习惯！

6、错误 6：没有通过对 inst_sram_en 的控制在阻塞时阻止新指令进入流水线

（1）错误现象

当 PC=157f5fe5 时写回寄存器的数据错误。

（2）分析定位过程



向前追溯可以看到这条指令是 LU12I.W，不需要 rj_value 和 rdk_value（而且确实都是不定态），立即数 bfaff000 写回 0101（5）号寄存器。验算后，在这条指令处没有发现错误，但写回的数据就是和金标准不同。LU12I.W 既不从寄存器里读数也不访存，写回的数据全从指令里得来，就不存在之前的历史错误导致的这一拍错误的情况。那就只可能是这一拍读出来的指令有错，本身就不应该是 LU12I.W。

于是我拉出了 inst（IF 阶段的指令）和 inst_reg（ID 阶段的指令）信号，发现 inst_reg 并不总比 inst 滞后一拍（理想中的场景应该是一直滞后一拍的）。我猜测可能是冲突的时候，虽然 IF 向 ID 的流水被阻塞住了，但仍然在向流水线发射指令，也就是说 IF 中会有新的指令，但不会流向 ID。这样是错误的，因为会导致 IF 中原来的指令被新指令覆盖掉，从而消失不再执行，流水线上就少执行了一条指令，又早发射了一条指令，从而导致混乱的错误。回看代码发现 inst_sram_en 没有受 fs_allowin 控制，这就导致了哪怕 IF 向 ID 的流水被阻塞，新的指令也会进入 IF 阶段。

（3）错误原因

发生冲突时，inst_sram_en 没有受 fs_allowin 控制，导致哪怕 IF IF 向 ID 的流水被阻塞，新的指令也会进入 IF 阶段。

（4）修正效果

```
57 |   assign inst_sram_en = resetn && fs_allowin; // 1'b1;
```

修改 inst_sram_en 的赋值逻辑后，exp8 通过。

```
----[1340705 ns] Number 8'd20 Functional Test Point PASS!!!
==============================================================
Test end!
----PASS!!!
$finish called at time : 1341695 ns : File "D:/CA_local_depository/exp6-10/exp8
```

7、错误 7：晚了一拍得到 final_result

（1）错误现象

写回寄存器的数据错误。

（2）分析定位过程

往前追溯到 ID，是 add.w 指令，拉取信号，此时 rf_raddr1=mem_dest，rj_value=final_result，也就是说该指令的 ID 阶段和此时的 MEM 阶段冲突。我突然意识到 final_result 好像并不是 MEM 阶段得到的结果。

（3）错误原因

```
assign rj_value =
    (exe_gr_we && es_valid && exe_dest == rf_raddr1)? alu_result :
    (mem_gr_we && ms_valid && mem_dest == rf_raddr1)? final_result :
    (gr_we_reg && ws_valid  && dest_reg   == rf_raddr1)? rf_wdata   :
    rf_rdata1;
assign rkd_value =
    (exe_gr_we && es_valid && exe_dest == rf_raddr2)? alu_result :
    (mem_gr_we && ms_valid && mem_dest == rf_raddr2)? final_result :
    (gr_we_reg && ws_valid  && dest_reg   == rf_raddr2)? rf_wdata   :
    rf_rdata2;
```

我的 rj_value 和 rkd_value 的赋值是这样实现的，如果和 MEM 阶段冲突，就将其写为 final_result 的值。可是由于我最开始是在 WB 阶段取回的访存结果，因此哪怕后来将取回访存结果提前到 MEM 阶段了，我仍然将 MEM 阶段的 mem_result 流了一级成为 mem_result_reg，在 WB 阶段从 mem_result_reg 和 alu_result_reg 中选择得到 final_result。这就导致我在和 MEM 阶段冲突时，给 rj_value 或 rkd_value 赋的值根本就不是 MEM 阶段的结果。并且此时 MEM 阶段根本就没有一个结果。因此可以将 final_result 提前一拍，在 MEM 阶段得到，同时也可以减少没必要的流水。

（4）修正效果

```
assign mem_result = data_sram_rdata;
assign final_result = mem_res_from_mem ? mem_result : mem_alu_result;
```

在 MEM 阶段得到 final_result，exp9 通过。

```
----[ 604125 ns] Number 8'd20 Functional Test Point PASS!!!
==============================================================
Test end!
----PASS!!!
$finish called at time : 604575 ns : File "D:/CA_local_depository/exp6-10/exp9/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:25 ; elapsed = 00:00:32 . Memory (MB): peak = 2507.809 ; gain = 0.000
```

（5）归纳总结

这个错其实很有意思，关于 final_result 的错误贯穿了 exp7、8、9。最开始是 MEM 请求 WB 读写，因为提前了一拍拿读出的数据出过错，于是就把 mem_result 和 final_result 写在了 WB 阶段；后来由于将访存提前了一拍，EXE 请求 MEM 读写，因为没有把 mem_result 流一级给 WB 又出了错；最后，由于采用前递技术，ld 指令需要在 MEM 阶段拿到结果才能够提高 xiaol ，因此又因为没把 final_result 写在 MEM 阶段出了错。

这给我的教训是，写代码之前一定设计好，磨刀不误砍柴工。

四、实验总结

exp8 和 exp9 的测试文件相同，对比测试完成的用时：只用阻塞技术时，用时 1341695ns，加入前递技术之后，用时 604575ns，用时几乎少了一半，只用了原来的 45%。说明前递技术对流水线的效率的提高非常有用。通过前递技术的原理分析，这种结果也是合理的。设理想状况下一拍一条指令，整体用时为 t。假设 50%的指令会发生数据相关的冲突，只使用阻塞技术时平均每条指令被阻塞两拍，那么整体用时会为(1+0.5*2)*t=2t；加入前递技术之后平均每条指令被阻塞半拍，那么整体用时会为(1+0.5*0.5)t=1.25t 拍，差不多会达到用时缩短一半的结果。当然这只是粗略计算，不过足以表明前递技术对于提高流水线效率的重要性。

```
Test end!
----PASS!!!
$finish called at time : 1341695 ns : File "D:/CA_local_depository/exp6-10/exp8/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:25 ; elapsed = 00:00:38 . Memory (MB): peak = 1106.812 ; gain = 142.965
```

```
Test end!
----PASS!!!
$finish called at time : 604575 ns : File "D:/CA_local_depository/exp6-10/exp9/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:25 ; elapsed = 00:00:32 . Memory (MB): peak = 2507.809 ; gain = 0.000
```

这次实验的进行虽然一波三折，但我收获了很多。

最开始写无相关冲突的流水线的时候，由于代码还比较简单，讲义里也没有提醒，我没有实现每一阶段的握手信号、valid 信号的流水，也没有将每个阶段分文件写，甚至在 MEM 阶段请求数据 WB 阶段读写数据。这就导致了我虽然 exp7 通过了，但代码仍然“千疮百孔”，在 exp8 之前经历了艰难的修改访存、添加 valid 信号流水、分文件、添加握手信号、打包模块间的数据。这也提醒我在之后的实验中一定先设计好再动手。

好在最后还是顺利完成了。