**实验二报告**

学号 2017K8009929044

姓名 李昊宸

箱子号 33

一、实验任务（10%）

1. 加深对指令集的理解。

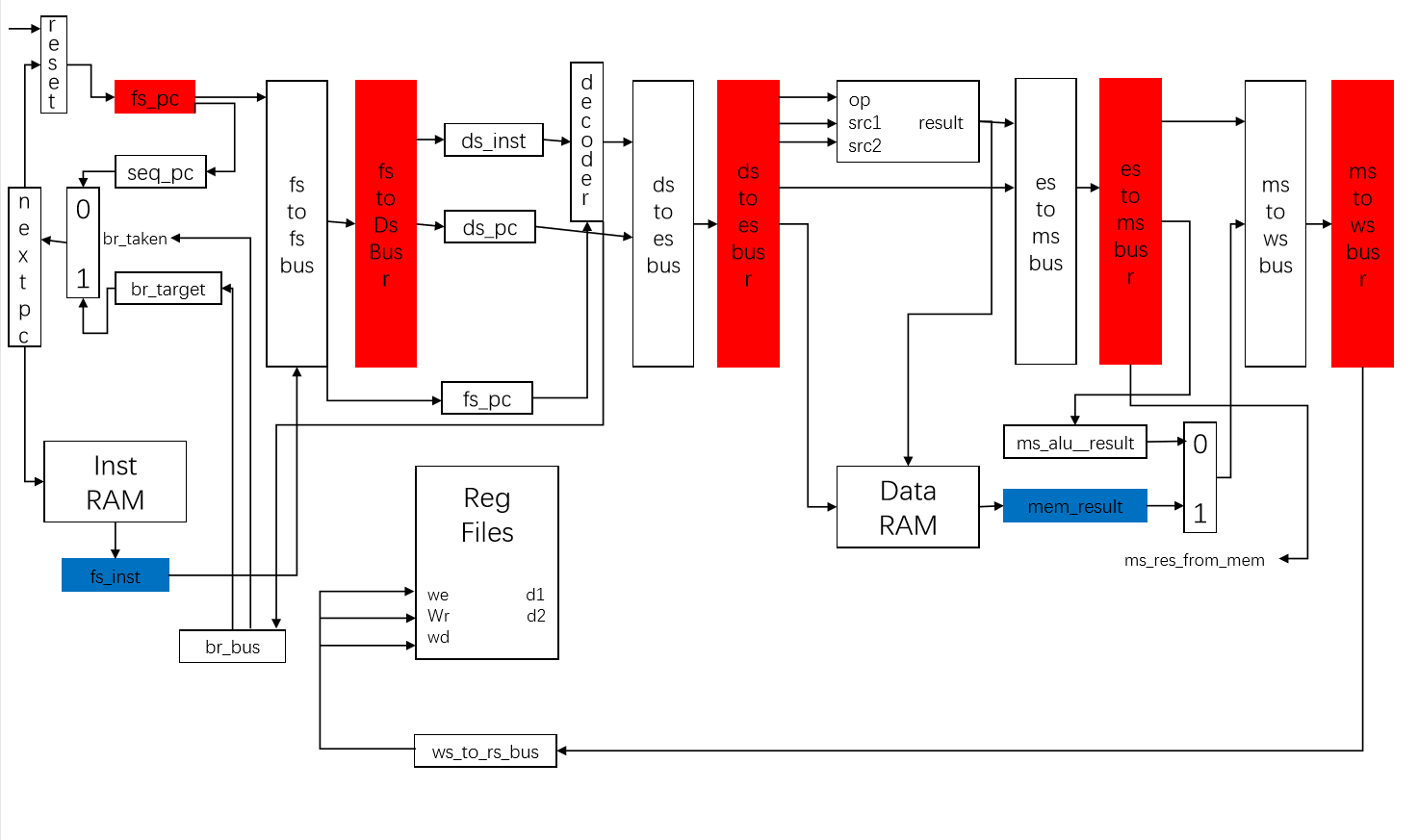
2. 了解 CPU设计中流水线切分的原理。

3. 学会识别常见波形异常，理解其产生原因，并能修正。

4. 熟悉并运用 verilog语言进行电路设计。

二、实验设计（40%）

（一）总体设计思路



图一 mycpu模块结构图

mycpu实验结构主要分为五个部分：IF取值模块，ID译码模块，EX执行模块，MEM访存模块和WB写回模块。

每一个模块流水线的控制核心均为一个触发器：IF为fs\_pc，ID为fs\_to\_ds\_bus\_r，EX为ds\_to\_es\_bus\_r，MEM为es\_to\_ms\_bus\_r，WB为ms\_to\_ws\_bus\_r。四个触发器的值由流水条件中的valid与allowin控制，当握手信号握手成功时，数据被传输到下一个寄存器。

另外，在模块之间被传输的数据主要有这些：因为目前的cpu为五级流水cpu，在不同模块中被执行的指令都是不同的指令，所以一般来讲在整个cpu中有五条指令同时在执行，所以每一个模块向下一个模块传输的数据都至少包括当前指令的PC值，当前指令的具体信息。此外，每一个模块也需要向下一个模块传递控制信号，比如IF向ID传输指令的编码，ID向EX传输将指令译码之后得到的指令名和alu的操作码、操作数等等，EX向MEM传输alu的计算结果，以及最终数据的来源是alu的计算结果还是访存的取出数据，MEM向WB传输最终将写回Regfiles中的寄存器编号和写回数据。

除此之外，在ID到IF，WB到ID之间还有由组合逻辑搭成的反向总线，分别用于传输下一条指令地址和向寄存器写回的地址和数值。有人可能会问为什么不直接把寄存器堆放在WB模块中方便写回，这是因为在ID阶段涉及到一些指令需要从寄存器堆中取出数据，如果将其放在WB模块中运用起来并没有那么直观。

（二）重要模块1设计：IF取指模块

1. 工作原理

以fs\_pc作为操作核心，记录每个时钟周期处于IF操作模块内部正在执行的PC，并伴随着同步RAM（数据RAM）不断取出对应PC地址处存放的指令，发送给下个模块ID。同时，PC的跳转逻辑是由与ID模块相连接的数据通路br\_bus决定。

1. 接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟 |
| reset | IN | 1 | 复位信号，拉高为复位 |
| ds\_allowin | IN | 1 | 允许数据流水进入ID |
| br\_bus | IN | 33 | 从ID模块返回的下一步PC跳转信号 |
| fs\_to\_ds\_valid | OUT | 1 | 从IF向ID的数据传输是否可以进行 |
| fs\_to\_ds\_bus | OUT | 64 | IF向ID的总线 |
| inst\_sram\_en | OUT | 1 | ram使能信号，高电平有效 |
| inst\_sram\_wen | OUT | 4 | ram字节写使能信号，高电平有效 |
| inst\_sram\_addr | OUT | 32 | ram读写地址，字节寻址 |
| inst\_sram\_wdata | OUT | 32 | ram写数据 |
| isnt\_sram\_rdata | IN | 32 | ram读数据 |

表一 IF模块接口定义

1. 功能描述

此模块的作用主要为PC的转换和指令的取出。首先，fs\_pc信号由reset信号置于0xhbfbffffc，此时nextpc被赋值为起始地址0xbfc00000，发送取值信号到指令RAM中，这一个阶段被称为pre-IF阶段。因为同步RAM从发起读命令后，需要经历一个上升沿之后才能取出指令，所以在fs\_pc等于0xbfbffffc时，发出取0xbfc00000地址的指令，直到下一个上升沿，PC更新到0xbfc00000时，位于0xbfc00000的指令恰好被返回到fs\_inst。随后，nextpc被组合逻辑立即赋值到新的PC值，并被发送到数据RAM中，下一个上升沿重复以上循环。另外，PC的下一步更新与从ID模块返回的br\_bus信号有关。

（三）重要模块2设计：ID译码模块

1. 工作原理

从IF传入的信号，经过译码逻辑后，被分解成每个部分的信号后，和PC有关的信号被返回ID，其余和执行有关的信号被传递到EX模块。

1. 接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟 |
| reset | IN | 1 | 复位信号，拉高为复位 |
| es\_allowin | IN | 1 | 允许数据流水进入EX |
| ds\_allowin | OUT | 1 | 允许数据流水进入DS |
| fs\_to\_ds\_valid | IN | 1 | 从IF向ID的数据传输是否可以进行 |
| fs\_to\_ds\_bus | IN | 64 | IF向ID的总线 |
| ds\_to\_es\_valid | OUT | 1 | 从ID向EX的数据传输是否可以进行 |
| ds\_to\_es\_bus | OUT | 136 | ID向ES的总线 |
| br\_bus | OUT | 33 | 从ID模块返回的下一步PC跳转信号 |
| ws\_to\_rf\_bus | IN | 38 | 从WB模块返回的存入regfile的数据 |

表二 ID模块接口定义

1. 功能描述

从fs\_pc传来的PC值经过一拍上升沿，被更新到ds\_pc中。fs\_inst也被传递到ds\_inst中。指令的跳转信号br\_taken和将要跳转到的PC地址br\_target被通过br\_bus信号传递回IF模块。其他和执行有关的信号（如ALU控制信号，从regfile中取出的数据等）被放置于ds\_to\_es\_bus中，传递到EX模块。此外，regfile被放置在此模块中，由rf\_we,rf\_waddr和rf\_wdata信号控制写使能，写地址和写数据。

（四）重要模块3设计：EX执行模块

1. 工作原理

从ID传入的信号，被一一取出，一部分进如alu中进行计算，另一部分作为数据RAM的使能和控制信号被输出到数据RAM处，等待下一拍上升沿被更新到MEM阶段。

1. 接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟 |
| reset | IN | 1 | 复位信号，拉高为复位 |
| ms\_allowin | IN | 1 | 允许数据流水进入MEM |
| es\_allowin | OUT | 1 | 允许数据流水进入EX |
| ds\_to\_es\_valid | IN | 1 | 从ID向EX的数据传输是否可以进行 |
| ds\_to\_es\_bus | IN | 136 | ID向ES的总线 |
| es\_to\_ms\_valid | OUT | 1 | 从EX向MEM的数据传输是否可以进行 |
| es\_to\_ms\_bus | OUT | 71 | EX向MEM的总线 |
| data\_sram\_en | OUT | 1 | ram使能信号，高电平有效 |
| data\_sram\_wen | OUT | 4 | ram字节写使能信号，高电平有效 |
| data\_sram\_addr | OUT | 32 | ram读写地址，字节寻址 |
| data\_sram\_wdata | OUT | 32 | ram写数据 |

表三 EX模块接口定义

1. 功能描述

从fs\_pc传来的PC值经过一拍上升沿，被更新到ds\_pc中。fs\_inst也被传递到ds\_inst中。指令的跳转信号br\_taken和将要跳转到的PC地址br\_target被通过br\_bus信号传递回IF模块。其他和执行有关的信号（如ALU控制信号，从regfile中取出的数据等）被放置于ds\_to\_es\_bus中，传递到EX模块。此外，regfile被放置在此模块中，由rf\_we,rf\_waddr和rf\_wdata信号控制写使能，写地址和写数据。

（五）重要模块4设计：MEM访存模块

1. 工作原理

从EX传入的信号主要被保存之后，等待下一拍将其传递到下一个WB写回阶段。另外，最关键的部分是上个阶段从EX提出的访存申请被这一拍上升沿应答之后，返回到该模块中。

1. 接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟 |
| reset | IN | 1 | 复位信号，拉高为复位 |
| ws\_allowin | IN | 1 | 允许数据流水进入WB |
| ms\_allowin | OUT | 1 | 允许数据流水进入MEM |
| es\_to\_ms\_valid | IN | 1 | 从EX向MEM的数据传输是否可以进行 |
| es\_to\_ms\_bus | IN | 64 | EX向MEM的总线 |
| ms\_to\_ws\_valid | OUT | 1 | 从MEM向WB的数据传输是否可以进行 |
| ms\_to\_ws\_bus | OUT | 70 | MEM向WB的总线 |
| data\_sram\_rdata | IN | 32 | ram读数据 |

表四 MEM模块接口定义

1. 功能描述

从EX阶段传来的PC,指令和寄存器地址，被保存后通过ms\_to\_ws\_bus传递到WB写回模块。在这个模块，上个周期向数据RAM提交的请求将返回对应的内存数据到data\_sram\_rdata，并且从上个模块传过来的alu\_result在从ID传递的信号下，二者被选择出对应指令的那个后传递到WB模块。

（六）重要模块5设计：WB写回模块

1. 工作原理

从MEM传来的写回寄存器堆的数据在从ID一路传递过来的指令控制下，被写入到regfile中。

1. 接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟 |
| reset | IN | 1 | 复位信号，拉高为复位 |
| ws\_allowin | OUT | 1 | 允许数据流水进入WB |
| ms\_to\_ws\_valid | IN | 1 | 从MEM向WB的数据传输是否可以进行 |
| ms\_to\_ws\_bus | IN | 70 | MEM向WB的总线 |
| ws\_to\_rf\_bus | OUT | 38 | WB向ID中regfile的总线 |
| debug\_wb\_pc | OUT | 32 | 写回级（多周期最后一级）的 PC，需要 mycpu 里将 PC 一路带到 写回级 |
| debug\_wb\_rf\_wen | OUT | 4 | 写回级写寄存器堆(regfiles)的写使能，为字节写使能，如果 mycpu 写 regfiles为单字节写使能，则将写使能扩展成 4位即可 |
| debug\_wb\_rf\_wnum | OUT | 5 | 写回级写 regfiles的目的寄存器号 |
| debug\_wb\_rf\_wdata | OUT | 32 | 写回级写 regfiles的写数据 |

表五 WB模块接口定义

1. 功能描述

从MEM传递来的写回寄存器的数据和写回的地址被传递到ws\_to\_rf\_bus总线中，被返回到ID模块中执行写回操作。

三、实验过程（50%）

（一）实验流水账

2019/9/12 8:00 – 11:00 对照代码研究单周期流水cpu的结构，并画出框图

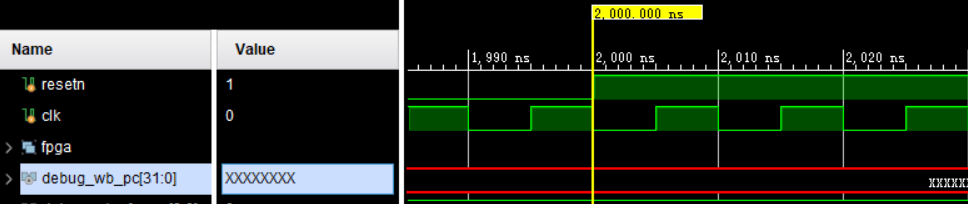
2019/9/12 14:00 – 20:00 修改程序中的bug

2019/9/15 13:00 –19:00 书写实验报告

2019/9/16 00:00 – 1:00 绘制框图

（二）错误记录

1、错误1：信号为X && always块内reg型变量使用阻塞赋值&&信号宽度不足

（1）错误现象

图二 debug\_wb\_pc信号为X的波形图

debug\_wb\_pc信号出现X波形。

（2）分析定位过程

X信号表示出现不定值，这种问题的出现主要有两种情况：1）声明为wire型的变量从未被赋值 2）写成了多驱动的代码。

debug\_wb\_pc是从IF阶段一步一步传递到WB阶段得到的变量，途径中没有出现多驱动的情况，于是问题可能出在没有对其赋值。倒序查找赋值逻辑，语句如下：

*assign debug\_wb\_pc = ws\_pc;*

*assign {ws\_gr\_we , ws\_dest , ws\_final\_result, ws\_pc} = ms\_to\_ws\_bus\_r;*

*if (ms\_to\_ws\_valid && ws\_allowin) begin*

*ms\_to\_ws\_bus\_r <= ms\_to\_ws\_bus;*

*end*

*assign ms\_to\_ws\_bus = {ms\_gr\_we，ms\_dest ，ms\_final\_result, ms\_pc };*

*assign {ms\_res\_from\_mem, ms\_gr\_we，ms\_dest，ms\_alu\_result，ms\_pc} = es\_to\_ms\_bus\_r;*

*if (es\_to\_ms\_valid && ms\_allowin) begin*

*es\_to\_ms\_bus\_r = es\_to\_ms\_bus; //发现 阻塞赋值*

*end*

*assign es\_to\_ms\_bus = {es\_res\_from\_mem, es\_gr\_we ，es\_dest ，es\_alu\_result ，es\_pc };*

*assign {es\_alu\_op，es\_load\_op，es\_src1\_is\_sa， es\_src1\_is\_pc，es\_src2\_is\_imm ，es\_src2\_is\_8，es\_gr\_we，es\_mem\_we，es\_dest，es\_imm ，es\_rs\_value ，es\_rt\_value，es\_pc } = ds\_to\_es\_bus\_r;*

*if (ds\_to\_es\_valid && es\_allowin) begin*

*ds\_to\_es\_bus\_r <= ds\_to\_es\_bus;*

*end*

*assign ds\_to\_es\_bus = {alu\_op，load\_op ，src1\_is\_sa，src1\_is\_pc，src2\_is\_imm，src2\_is\_8，gr\_we，mem\_we ，dest ，imm，rs\_value ，rt\_value，ds\_pc };*

*assign {ds\_inst, ds\_pc } = fs\_to\_ds\_bus\_r;*

*if (fs\_to\_ds\_valid && ds\_allowin) begin*

*fs\_to\_ds\_bus\_r <= fs\_to\_ds\_bus;*

*end*

*assign fs\_to\_ds\_bus = {fs\_inst , fs\_pc };*

*if (reset) begin*

*fs\_pc <= 32'hbfbffffc; //trick: to make nextpc be 0xbfc00000 during reset*

*end*

*else if (to\_fs\_valid && fs\_allowin) begin*

*fs\_pc <= nextpc;*

*end*

*assign nextpc = br\_taken ? br\_target : seq\_pc;*

*assign seq\_pc = fs\_pc + 3'h4;*

*assign {br\_taken,br\_target} = br\_bus; //（在IF中）*

*assign br\_bus = {br\_taken,br\_target}; //（在ID中）*

*assign br\_taken = ( inst\_beq && rs\_eq\_rt || inst\_bne && !rs\_eq\_rt || inst\_jal || inst\_jr ) && ds\_valid;*

*assign br\_target = (inst\_beq || inst\_bne) ? (fs\_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :*

*(inst\_jr) ? rs\_value : {fs\_pc[31:28], jidx[25:0], 2'b0};*

除此之外，在定义br\_bus的宽度时也出现了这样的语句：

*`define BR\_BUS\_WD 32*

*output [`BR\_BUS\_WD -1:0] br\_bus，*

*wire br\_taken;*

*wire [31:0] br\_target;*

（3）错误原因

在RESET为1时，fs\_pc被设置为32'hbfbffffc，seq\_pc为32’hbc00000。在RESET为0后，在满足条件（to\_fs\_valid && fs\_allowin）为1，但是还没有迎来上升沿时，IF向指令RAM发起读指令请求，地址为seq\_pc；迎来上升沿时，fs\_pc此时将迎来更新，由于指令RAM是同步读写，从而此时读出的指令才进入fs\_inst，紧接着进入fs\_to\_ds\_bus，但是由于上升沿已过，无法进入fs\_to\_ds\_bus\_r，从而导致br\_taken与br\_target的值为X，于是fs\_pc无法被赋值。从而导致之后所有模块的pc均为不定态。另外，由于br\_bud信号的宽度被错误的定义为了32位，这导致原来处于第32位（从第0位开始计算）的br\_taken信号无法被传递，于是fs\_pc无法正确取值。

（4）修正效果

将赋值语句进行如下修改：

*assign nextpc = (fs\_pc == 32'hbfbffffc)?seq\_pc:br\_taken ? br\_target : seq\_pc;*

这样保证了第一个上升沿到来时fs\_pc可以正常的跳转至32’ hbc00000。

另外，将阻塞赋值修改为非阻塞赋值：*es\_to\_ms\_bus\_r <= es\_to\_ms\_bus;*

将信号宽度的定义进行修改：*`define BR\_BUS\_WD 33*

修正后，fs\_pc信号开始正确跳转。

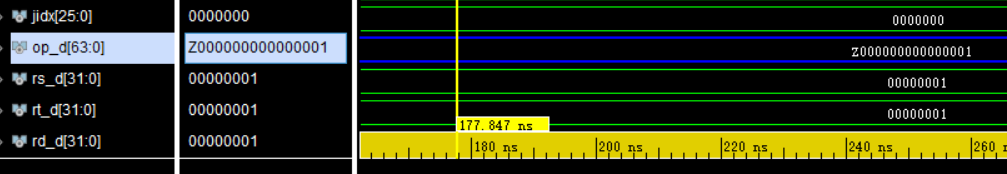
（5）归纳总结（可选）

流水线中从后面时序模块发出的控制前面时序模块的语句要注意，在启动时可能会失去判断条件。

另外，一定要检查赋值语句前后的信号宽度是否一致。

2、错误2：信号为Z

（1）错误现象



图三 op\_d的最高位信号为Z的波形图

op\_d信号的第63位出现Z波形



图四 load\_op信号为Z的波形图

load\_op信号出现Z波形

（2）分析定位过程

Z信号表示出现高阻。

op\_d信号为64位宽信号，类型为wire类型。正常情况下wire类型出现Z信号有两种可能：1）未用assign语句赋初值 2）调用模块时没有连接到端口。

寻找针对op\_d信号的赋值语句：decoder\_6\_64 u\_dec0(.in(op ), .out(op\_d )); 猜想问题出在decoder\_6\_64这个模块中。*module decoder\_6\_64( input [ 5:0] in, output [63:0] out);*

*genvar i;*

*generate for (i=0; i<63; i=i+1) begin : gen\_for\_dec\_6\_64*

*assign out[i] = (in == i);*

*end endgenerate*

load\_op信号为1位宽信号，类型为wire类型。正常情况下wire类型出现Z信号有两种可能：1）未用assign语句赋初值 2）调用模块时没有连接到端口。

尝试着去找load\_op信号的赋值语句，没有找到。

（3）错误原因

op\_d信号：对其赋值的decoder\_6\_64模块中的第63位没有被赋值操作。

load\_op信号：没有被赋值。

（4）修正效果

将decoder\_6\_64中的generate语句进行如下修改：

*generate for (i=0; i<64; i=i+1)*

为了搞清load\_op信号的作用，我查看了load\_op信号给其他信号的赋值语句：

*assign ds\_to\_es\_bus = {alu\_op，load\_op ，src1\_is\_sa，src1\_is\_pc，src2\_is\_imm，src2\_is\_8，gr\_we，mem\_we ，dest ，imm，rs\_value ，rt\_value，ds\_pc };*

*if (ds\_to\_es\_valid && es\_allowin) begin*

*ds\_to\_es\_bus\_r <= ds\_to\_es\_bus;*

*end*

*assign {es\_alu\_op，es\_load\_op，es\_src1\_is\_sa， es\_src1\_is\_pc，es\_src2\_is\_imm ，es\_src2\_is\_8，es\_gr\_we，es\_mem\_we，es\_dest，es\_imm ，es\_rs\_value ，es\_rt\_value，es\_pc } = ds\_to\_es\_bus\_r;*

*assign es\_res\_from\_mem = es\_load\_op;*

*assign es\_to\_ms\_bus = {es\_res\_from\_mem, es\_gr\_we ，es\_dest ，es\_alu\_result ，es\_pc };*

*if (es\_to\_ms\_valid && ms\_allowin) begin*

*es\_to\_ms\_bus\_r <= es\_to\_ms\_bus;*

*end*

*assign {ms\_res\_from\_mem, ms\_gr\_we，ms\_dest，ms\_alu\_result，ms\_pc} = es\_to\_ms\_bus\_r;*

*assign ms\_final\_result = ms\_res\_from\_mem ? mem\_result: ms\_alu\_result;*

也就是说，load\_op相当于在译码之后向ms阶段传递一个选择信号，用于ms将把alu的计算结果传递给wb阶段还是把从内存中读出来的数据传递给wb阶段。所以，定义load\_op的赋值语句如下：

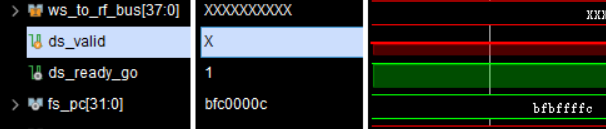
*assign load\_op = res\_from\_mem;*

至此，问题解决。op\_d和load\_op都有了正确的波形。

（5）归纳总结（可选）

对于Z信号，最常见的就是没有被赋值的wire信号。检查的时候只要一步一步按照赋值逻辑找到源头就能解决问题。

3、错误3：信号为X

（1）错误现象

图五 ds\_valid信号为X的波形图

修正前面的错误后，ds\_valid信号出现X波形。

（2）分析定位过程

X信号表示出现不定值，这种问题的出现主要有两种情况：1）声明为reg型的变量从未被赋值 2）写成了多驱动的代码。

尝试去寻找ds\_valid的赋值语句，结果没有找到。

（3）错误原因

ds\_valid是用来控制流水的信号。没有对该信号的赋值，流水就无法正确进行。

（4）修正效果

补充以下赋值语句：

*always @(posedge clk) begin*

*if (reset) begin*

*ds\_valid <= 1'b0;*

*end*

*else if (ds\_allowin) begin*

*ds\_valid <= fs\_to\_ds\_valid;*

*end*

*……*

*end*

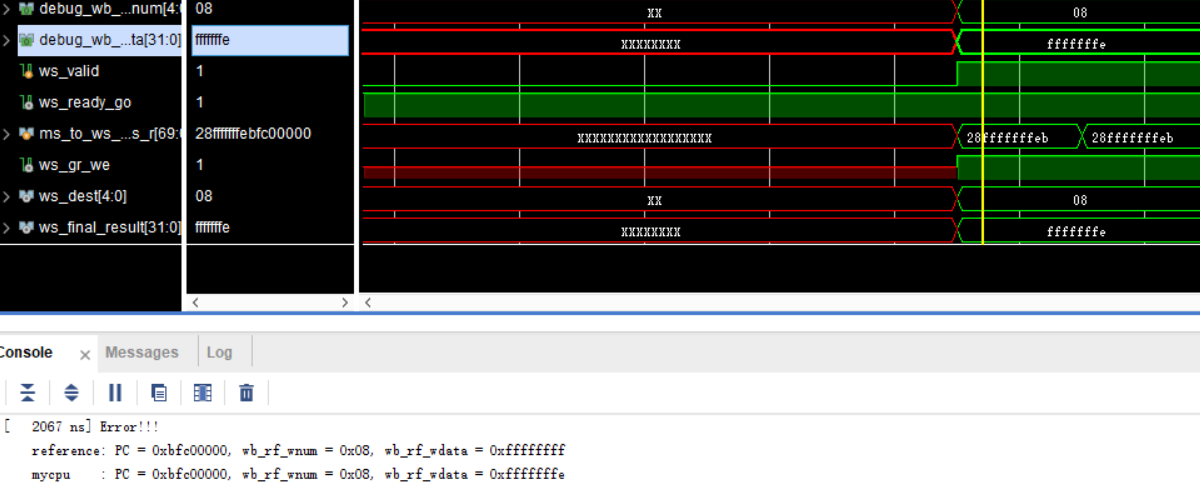
修正后，ds\_valid信号开始正确跳转，后续流水开始正常进行。

（5）归纳总结（可选）

reg类型的变量出现X信号，往往是出现了多驱动赋值，使其值处于不定态，或者根本没有被赋过值。检查的时候只要一步一步按照赋值逻辑找到源头就能解决问题。

4、错误4：写入数据比对错误

（1）错误现象

图六 alu计算结果错误的波形图

PC执行的指令应该为addiu，两个加数一个是0x00000000，一个是0xffffffff，结果应该为0xffffffff，但是得到了0xfffffffe的结果。

（2）分析定位过程

addiu指令表示一个寄存器内的数与一个有符号数相加，0与-1的和应该是-1，但是却得到了-2的结果。由此猜想有可能是在调用alu模块时两个入口数据都写为了同一个数据：（-1）+（-1） = -2。

（3）错误原因

*alu u\_alu(*

*.alu\_op (es\_alu\_op ),*

*.alu\_src1 (es\_alu\_src2 ),*

*.alu\_src2 (es\_alu\_src2 ),*

*.alu\_result (es\_alu\_result)*

*);*

此处alu的src1和src2都写成了es\_alu\_src2。

（4）修正效果

将调用模块修改为以下：

*alu u\_alu(*

*.alu\_op (es\_alu\_op ),*

*.alu\_src1 (es\_alu\_src1 ),*

*.alu\_src2 (es\_alu\_src2 ),*

*.alu\_result (es\_alu\_result)*

*);*

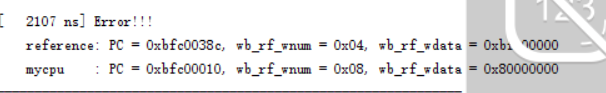
修改后，test function得以通过。

（5）归纳总结（可选）

哼其实这个错误我早就猜到了！因为助教老师在课上一直强调他的模块调用出现了笔误。

5、错误5：写入数据比对错误

（1）错误现象



图七 branch指令跳转错误的波形图

从这里可以看出，问题大概出现在参考的pc值应该通过branch指令从0xbfc0000bc跳转到0xbfc0038c，但是却向下直接执行0xbfc00010了。

（2）分析定位过程

这个错误的出现时间是在我刚刚修改完对next\_pc的逻辑赋值之后。于是可能是我的修改逻辑有错误。

（3）错误原因

起初我把nextpc修改为了*assign nextpc = (！reset)?seq\_pc:br\_taken ? br\_target : seq\_pc;*

后来发现是我对reset信号的定义不太了解导致错误的发生，使在复位信号关闭之后nextpc始终等于fs\_pc+4。

（4）修正效果

将赋值语句修改为以下：

*assign nextpc = (fs\_pc == 32'hbfbffffc)?seq\_pc:br\_taken ? br\_target : seq\_pc;*

但是，事实上，在经过原先的修改之后，ds\_valid有了其确定值，从而nextpc也可以取到其正常值，于是最后可以再修改回原来的*assign nextpc = br\_taken ? br\_target : seq\_pc;*

（5）归纳总结（可选）

reset信号其实是由resetn信号取反得到的，波形图中体现的是resetn。

一定要检查赋值语句前后的信号宽度是否一致。（改了一圈bug又改回来了）

6、错误6：组合环路

（1）错误现象

在结束第9个测试点，进入第10个测试点时，波形停止。

（2）分析定位过程

查看测试函数集，在汇编文件start中，找到指令*jal n10\_nor\_test*。找到调用的函数n10\_nor，打开，入口地址LEAF(n10\_nor\_test)下执行了如下函数：*TEST\_NOR(0xad9cc00e, 0x38b7ec24, 0x424013d1)；*

继续查找头文件inst\_test.h，找到定义

*#define TEST\_NOR(in\_a, in\_b, ref) \*

*LI (t0, in\_a); \*

*LI (t1, in\_b); \*

*LI (v1, ref); \*

*NOP4; \*

*nor v0, t0, t1; \*

*NOP4; \*

*bne v0, v1, inst\_error; \*

*nop*

于是确认关键位置是nor指令中出现了组合环。

（3）错误原因

查看波形停止的位置，在*assign conf\_addr <= cpu\_data\_addr；*处出现循环赋值的错误。继续查找剩余文件，找到如下赋值链：*data\_sram\_addr <= es\_alu\_result <= alu\_result <= nor\_result <= ~or\_result*

*assign or\_result = alu\_src1 | alu\_src2 | alu\_result;*在这里出现组合环。

（4）修正效果

将赋值语句修改为以下：

*assign or\_result = alu\_src1 | alu\_src2；*

随后测试程序通过。

（5）归纳总结（可选）

组合环太烦人了。。。一定不要把纯组合逻辑中的两个变量以任何方式接起来。

7、错误7：写入数据比对错误

（1）错误现象

错误波形忘记截图了……但是问题出现在执行到第12个测试点时。在一个执行逻辑右移0位的指令时，原数据最高位本来为1，输出的alu结果中最高位却变成了0。

（2）分析定位过程

猜想为alu中的srl指令出现了问题，毕竟其他位的数据都十分正确，并且pc停止在了执行srl的第一个指令处。

（3）错误原因

来看看srl的定义指令！

*assign sr\_result = sr64\_result[30:0];*

*assign sr64\_result = {{32{op\_sra & alu\_src2[31]}}, alu\_src2[31:0]} >> alu\_src1[4:0];*

*assign alu\_result = ({32{op\_add|op\_sub}} & add\_sub\_result)*

*| ({32{op\_slt }} & slt\_result)*

*| ({32{op\_sltu }} & sltu\_result)*

*| ({32{op\_and }} & and\_result)*

*| ({32{op\_nor }} & nor\_result)*

*| ({32{op\_or }} & or\_result)*

*| ({32{op\_xor }} & xor\_result)*

*| ({32{op\_lui }} & lui\_result)*

*| ({32{op\_sll }} & sll\_result)*

*| ({32{op\_srl|op\_sra}} & sr\_result);*

很显然，sr\_result的第31位又被忽略了。

（4）修正效果

将赋值语句修改为以下：

*assign sr\_result = sr64\_result[31:0];*

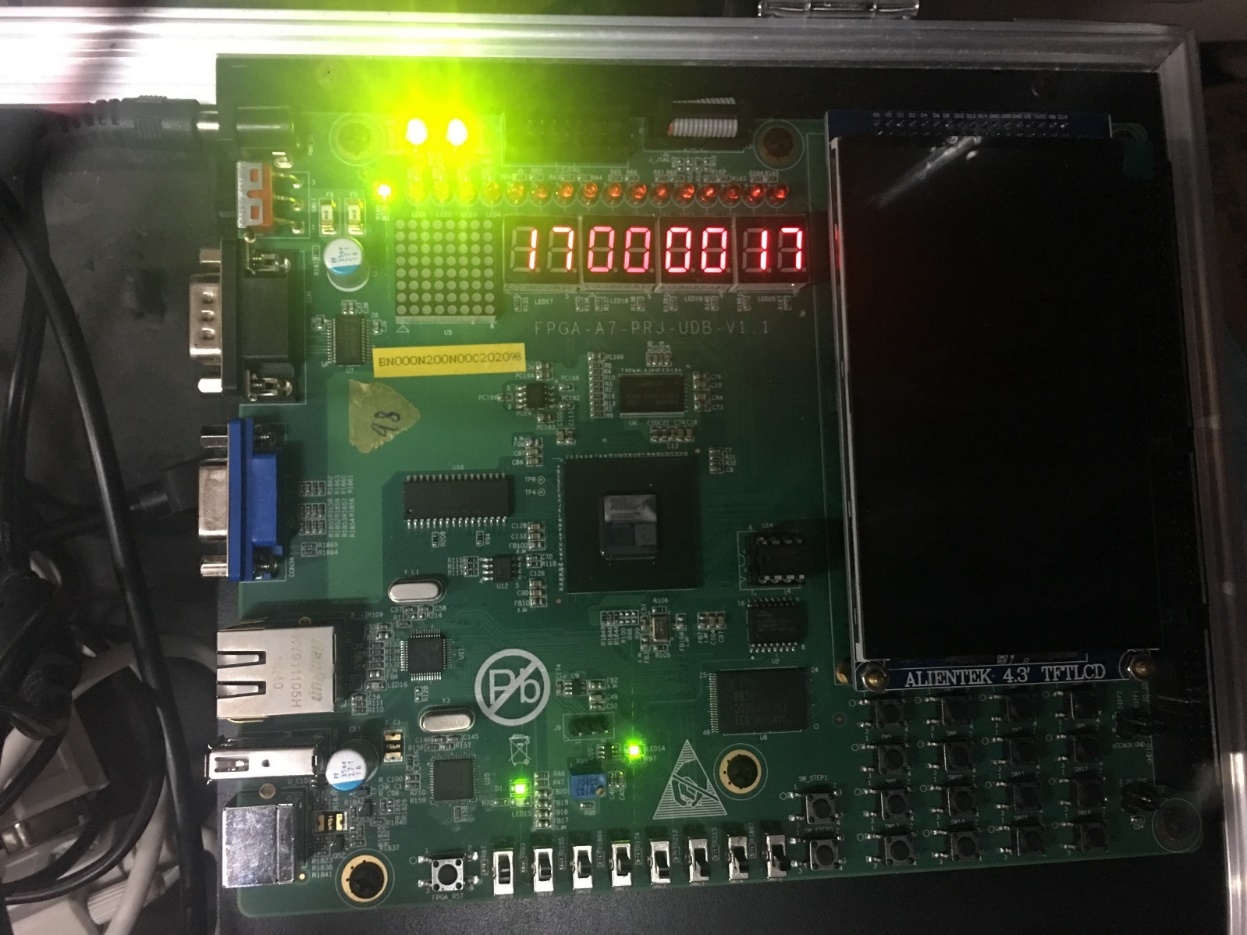
修改后测试程序通过。

（5）归纳总结（可选）

一定要检查赋值语句前后的信号宽度是否一致。

不过，信号位宽不够的bug也太多了吧。。。

四、实验总结（可选）



图八 上板成功拍照留念

看着着两个绿灯，心里的愉悦不可言说。（啊我的肝好疼）

其实起初被这个17吓得不轻（因为记得有23个测试点）。后来查看了led灯的源代码，发现它以16进制作为输出，才缓了一口气。