实验 2-2 报告

学号：2016K8009929029
姓名：张丽玮

一、实验任务（10%）

这个实验是在原有多周期 CPU 的基础上增加静态五级流水的功能，并实现延迟槽。延迟槽不只是原来的 nop 指令，可以是任意指令。五级流水的指令之间可不考虑相关性。

在实验 2-1 基础上新增 19 条指令，其中包括乘法除法指令；考虑数据相关，做一个旁路延迟设计；进行数据的前递处理；实现 booth+wallace 的乘法器和迭算法的除法器。

二、实验设计（30%）

总的来说，从 mips_cpu 顶层结构出发，直接跳过先前冗余繁杂的数据处理，而是将数据处理放入每个状态和模块中。因此五级流水就单独 IF、DE、EX_ID、MEM 和 WB 每个状态设计一个模块，而每个模块承接上一个模块的输出进行输入，从而达到五级流水的控制效果。

设计一个 bypass 的旁路模块，从而来判定是否有指令相关，而要延迟。一个 booth 算法完成乘法操作和 wallace 算法完成加法操作的高速乘法器，利用迭代辗转相除的除法器，从而实现新增的 19 条指令。

1、fetch 模块

fetch 模块主要完成一个取指操作，因此实际输入进来的下一条 PC 指令以及读入数据 rdata。而模块中用了两个 always 是为了实现一个两拍操作，达到下一条指令在下一拍取到的效果。

增加了一个 IRwrite 接口，用以判定指令是否出现相关，需要延迟取指。

IRwrite 接口在 bypass 中获值。

2、decode 模块

decode 模块是一个译码操作模块，接口过多不予列举，但这么多接口实际上就是完成了从指令出发，通过 control 模块获取指令应该进行的操作，后缀 ID 代表从译码阶段传去 PC 跳转判定的指令，而 ID_EXE 则代表从译码阶段传去执行阶段的相关数据。

译码阶段由于新指令的加入，增加了几个接口，主要与乘除法相关。

```
output reg  [ 1:0]      MULT_DE_EX, //new
output reg  [ 1:0]       DIV_DE_EX, //new
output reg  [ 1:0]      MFHL_DE_EX, //new
output reg  [ 1:0]      MTHL_DE_EX, //new
```

这几个均是从译码送往执行阶段的接口，用以判断 DIV、DIVU、MULT、MULTU、MFHI、MFLO、MTHI、MTLO 这八条指令。

如果不需要延迟，就将将这四个接口数据进行前递。而实际上 decode 模块的关键在于 control 模块，即对指令的读入判定，这里也加入了相关接口，从而输出 MULT_DE 等值。如果并非除法相关指令则初始化寄存器，若除法指令已经完成，啧初始化其他寄存器后，DIV 信号和两个 RegData 寄存器仍为本身。

还有一个判断 HI\LO 的逻辑，

```
wire [31:0] MULT_HI_LO = {32{MFHL_DE_EX_1[1]}}  & MULT_Result[63:32] | {32{MFHL_DE_EX_1[0]}}  & MULT_Result[
wire [31:0]  EX_HI_LO = {32{MFHL_DE_EX_1[1]}}  & HI                | {32{MFHL_DE_EX_1[0]}}  & LO;
wire [31:0]  MEM_HI_LO = {32{MFHL_EX_MEM[1]}}   & HI                | {32{MFHL_EX_MEM[0]}}   & LO;
wire [31:0]   WB_HI_LO = {32{MFHL_MEM_WB[1]}}    & HI                 | {32{MFHL_MEM_WB[0]}}    & LO;
```

实际上就是判断是何种乘法，两位寄存器 0 位低位取数，1 为高位取数。

3、execute 模块

这是一个执行模块，完成的主要操作是调用运算器进行执行。接口过多不予列举，不过接口命名可以看出是从哪个模块往哪个模块的数据。主体部分是三个多路选择器与一个运算器执行运算操作。

4、memory 模块

简单明了的存指操作，接口命名可以看出数据来源和去向。主体部分就是一个时钟节拍控制的赋值操作。

5、writeback 模块

一个写回模块。不需要时钟信号控制，直接赋值即可。

6、control 模块

新增添了 19 条指令的判定，重点在于这八条。

```
wire inst_div    = (op == 6'd0) && (func == 6'b011010);
wire inst_divu   = (op == 6'd0) && (func == 6'b011011);
wire inst_mult   = (op == 6'd0) && (func == 6'b011000);
wire inst_multu  = (op == 6'd0) && (func == 6'b011001);
wire inst_mfhi   = (op == 6'd0) && (func == 6'b010000);
wire inst_mflo   = (op == 6'd0) && (func == 6'b010010);
wire inst_mthi   = (op == 6'd0) && (func == 6'b010001);
wire inst_mtlo   = (op == 6'd0) && (func == 6'b010011);//八个乘除法指令
```

```
assign MULT[0] = inst_mult;
assign MULT[1] = inst_multu;//两位是为了判断是有符号乘法还是无符号乘法

assign DIV[0] = inst_div;
assign DIV[1] = inst_divu;

assign MFHL[0] = inst_mflo;
assign MFHL[1] = inst_mfhi;

assign MTHL[0] = inst_mtlo;
assign MTHL[1] = inst_mthi;
//乘除的八条指令
```

这个即是前面所说的低位和高位取数时的信号获取来源，根据 0 位与 1 位进行逻辑判断。

7、Bypass 模块

这是一个为了实现指令相关时延迟取指的旁路模块，

先看数据相关。数据相关可以根据冲突访问读和写的次序分为 3 种。第 1 种是写后读相关（Read After Write，RAW），就是后面指令要用到前面指令所写的数据，这是最常见的类型也称为真相关。第 2 种是写后写相关（Write After Write，WAW），也称为输出相关，即两条指令写同一个单元，在乱序执行的结构中如果后面的指令先写，前面的指令后写，就会产生错误的结果。第 3 种是读后写相关（Write After Read，WAR），在乱序执行的结构或者读写指令流水级不一样时，如果后面的写指令执行得快，在前面的读指令读数之前就把目标单元原来的值覆盖掉了，导致读数指令读到了该单元“未来”的值，从而引起错误。

其实逻辑并不复杂，根据传递过来的 MEM_WB 信号，判断这个读写之间是否会产生冲突，需要延迟（即 HAZ 信号），最后 stall 信号用几个或逻辑判断了四种等待情况，最后一种为迭代除法未做完的情况。

8、divider 模块

一个运用迭代的除法器。busy 和 done 信号分别表示除法是否正在运行和是否已经做完，从而在除法进行时对于其他操作阻塞处理。

实际上是一个辗转相除法的处理，利用时钟周期，和 count，完成一个 busy 和 done 的判断从而达到阻塞效果。

9、Multiplier 模块

此模块原本实现了 booth 算法的乘法器，但是由于是和 div 除法器一样利用了时钟周期，进行了一个状态机的操作，而导致在测试 mul_tb 的时候由于时钟延迟，没有办法获得正确结果。后来只是简单采用了乘号。

（具体代码可见 multiply.v 文件）。

等之后时间富裕之后再添加信号 busy 和 done 等尝试完成这个乘法器。

10、其他模块

实际上还有一个 pc 跳转模块，是为了根据是否有跳转指令来判断下一个 pc 是+4 还是跳转。control 模块较上次进行了一些改动，但是实现方式和结果相差不大。alu 模块增加了四条指令，为了代码风格，重新写了一下逻辑。reg_file 模块基本未改。decode 模块在 tools 文件中，基本就是龙芯的 tools 复制过来。

三、实验过程（60%）

（一）实验流水账

9 月 19 日，晚六点到半夜不知道几点，重构了 cpu

9 月 20 日，晚八点到一点，cpu 的 debug

9 月 21 日，下午两点到晚十点，阅读五级流水讲义，改写 pipeline 模块

9 月 22 日，下午两点到四点，尝试嵌入 pipeline 模块失败

9 月 24 日，下午两点到第二天早上，重新构建流水线结构，debug，调试测试通过

9 月 25 日，晚 6 点到晚 10 点，增加旁路 bypass，未做完

9 月 26 日，晚 10 点到 10 点半，继续旁路 bypass

9 月 27-30 日，阅读手册和讲义，考虑实验要求和逻辑处理。

10 月 1-3 日，查阅相关资料

10 月 4 日，下午 3 点到晚 9 点，完成 bypass 设计，增加 control 模块指令

10 月 5 日，下午三点到晚 6 点，写乘除法器，仿真测试出错

10 月 6 日，下午 3 点到晚 9 点，debug 无果

10 月 7 日，下午 3 点到 9 点，找到了 bug 原因，尝试修改乘法器

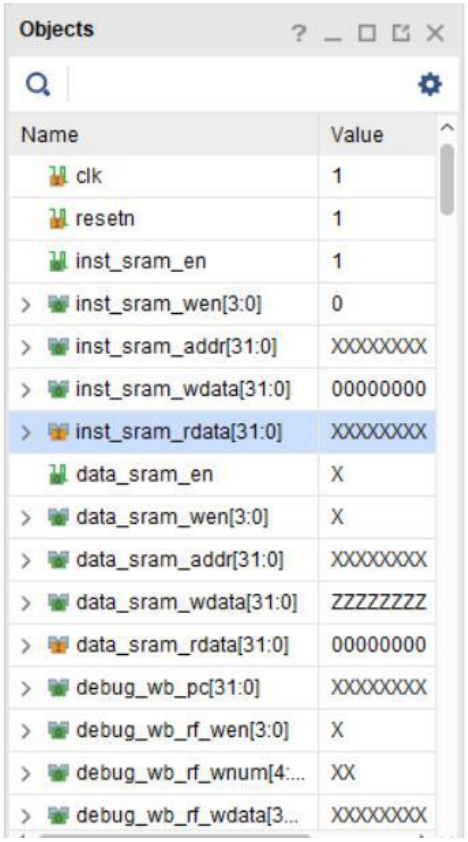
10 月 8 日，下午 3 点到第二天 3 点，出现新的 bug，修改 alu，测试通过，写报告。

（二）错误记录

具体描述实验过程中的错误，环境问题、仿真阶段、上板阶段的都可以记录。

1、错误 1

（1）错误现象



全线飘红，wdata 为高

（2）分析定位过程

有 Z 调 Z，实际上是 wdata 接口接错。

（3）错误原因

wdatas 所接接口接口名写错，该接口未定义。

（4）修正效果

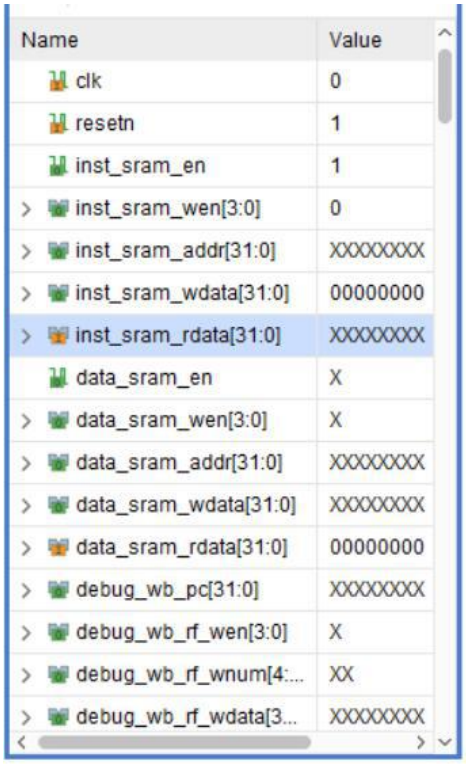
更正接口名，成功便成 XXXX

（5）归纳总结（可选）

属于写代码和重构时的粗心。善用查找和替换。

2、错误 2

（1）错误现象



全线飘红，输入也为 XXXXX

（2）分析定位过程

按照逻辑关系寻找 raddr，与 pc 有关，猜想是 pc 的问题。

（3）错误原因

先前用的 pc 跳转逻辑在此流水线中不再适用，不再是简单多周期的跳转方式，需要另外判定。

（4）修正效果

重新设计一个 pc 跳转模块，成功解决，

（5）归纳总结（可选）

要从简单多周期的思维方式转变为流水线的思维方式。

3、错误 3

（1）错误现象

```
[   2177 ns] Error!!!
    reference: PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
    mycpu    : PC = 0xbfc00008, wb_rf_wnum = 0x08, wb_rf_wdata = 0xxxxxXxxx
```

（2）分析定位过程

判断是 pc 出错，外部逻辑无误，定位到 alu。

（3）错误原因

之前重构的 cpu，alu 采取 12 位 op 进行 control，后来因为不知道如何实现示例模块的 pipeline 调用而重新返回原本思路，但外部调用时忘记了 alu 此时是 12 位的 op，仍然当做四位来调用。

（4）修正效果

重新采用 lab1 的 alu，成功解决。

（5）归纳总结（可选）

tcl 的 warning 同样很重要，有些接口不符警告能够帮你发现问题。

3、错误 4

（1）错误现象

```
[   2137 ns] Error!!!
    reference: PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
    mycpu    : PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xffffbfb0
```

Re...	04	Array
AL...	XXXXXXXX	Array
Me...	XXXXXXXX	Array
PC...	bfc00390	Array
AL...	00000000	Array
AL...	ffffbfb0	Array
AL...	ffffbfb0	Array
Re...	05	Array

（2）分析定位过程

定位到 regfile 写回，但是发现逻辑并无问题，然后询问同学发现

是 inst_sram 出了问题

（3）错误原因

Inst_sram 的 generate 有问题

（4）修正效果

自己重试了多次未果，包括重下环境，多次重新载入，之后求助同学。

（5）归纳总结（可选）

5、错误 5

（1）错误现象

```
[   2157 ns] Error!!!
    reference: PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
    mycpu    : PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfaf0000
```

（2）分析定位过程

在 piazza 上看到了老师写的注意事项，这个错误是由于 ori 等几个指令未添加造成的。

（3）错误原因

指令添加出错，增加了 alu 的可处理指令数

（4）修正效果

添加之后可以正常通过

6、错误 6

（1）错误现象

```
——[1553655 ns] Number 8'd34 Functional Test Point PASS!!!
--------------------------------------------------------------

[1554237 ns] Error!!!
    reference: PC = 0xbfc2982c, wb_rf_wnum = 0x15, wb_rf_wdata = 0x00000002
    mycpu    : PC = 0xbfc29834, wb_rf_wnum = 0x02, wb_rf_wdata = 0x00000002
--------------------------------------------------------------
```

```
45807    bfc2982c:    0000a812    mflo    s5
45808    bfc29830:    0000b010    mfhi    s6
```

```
        [1542000 ns] Test is running, debug_wb_pc = 0xbfc11800
        [1552000 ns] Test is running, debug_wb_pc = 0xbfc11fd0
——[1553655 ns] Number 8'd34 Functional Test Point PASS!!!
        [1562000 ns] Test is running, debug_wb_pc = 0x00000000
        [1572000 ns] Test is running, debug_wb_pc = 0x00000000
        [1582000 ns] Test is running, debug_wb_pc = 0x00000000
        [1592000 ns] Test is running, debug_wb_pc = 0x00000000
        [1602000 ns] Test is running, debug_wb_pc = 0xbfc2a384
        [1612000 ns] Test is running, debug_wb_pc = 0xbfc2a5d4
        [1622000 ns] Test is running, debug_wb_pc = 0xbfc2a824
        [1632000 ns] Test is running, debug_wb_pc = 0x00000000
        [1642000 ns] Test is running, debug_wb_pc = 0x00000000
        [1652000 ns] Test is running, debug_wb_pc = 0x00000000
        [1662000 ns] Test is running, debug_wb_pc = 0x00000000
——[1666335 ns] Number 8'd35 Functional Test Point PASS!!!
        [1672000 ns] Test is running, debug_wb_pc = 0x00000000
        [1682000 ns] Test is running, debug_wb_pc = 0x00000000
        [1692000 ns] Test is running, debug_wb_pc = 0x00000000
        [1702000 ns] Test is running, debug_wb_pc = 0x00000000
        [1712000 ns] Test is running, debug_wb_pc = 0x00000000
        [1722000 ns] Test is running, debug_wb_pc = 0xbfc1d65c
        [1732000 ns] Test is running, debug_wb_pc = 0xbfc1d8ac
        [1742000 ns] Test is running, debug_wb_pc = 0x00000000
        [1752000 ns] Test is running, debug_wb_pc = 0x00000000
        [1762000 ns] Test is running, debug_wb_pc = 0x00000000
        [1772000 ns] Test is running, debug_wb_pc = 0x00000000
        [1782000 ns] Test is running, debug_wb_pc = 0x00000000
        [1792000 ns] Test is running, debug_wb_pc = 0x00000000
        [1802000 ns] Test is running, debug_wb_pc = 0xbfc1e95c
        [1812000 ns] Test is running, debug_wb_pc = 0xbfc1ebac
        [1822000 ns] Test is running, debug_wb_pc = 0x00000000
```

（2）分析定位过程

分析找到波形，错误原因在于并没有取到乘法指令，而导致没有进入乘法运算流程

困扰很久，不明白为何取指错误。但是之后看到 piazza 上老师回到另一个同学的问题时提到往 error 前找错误，发现实际上在之前就已经有了很多 0，是不正常现象，而这导致了后面的取指错误。

（3）错误原因

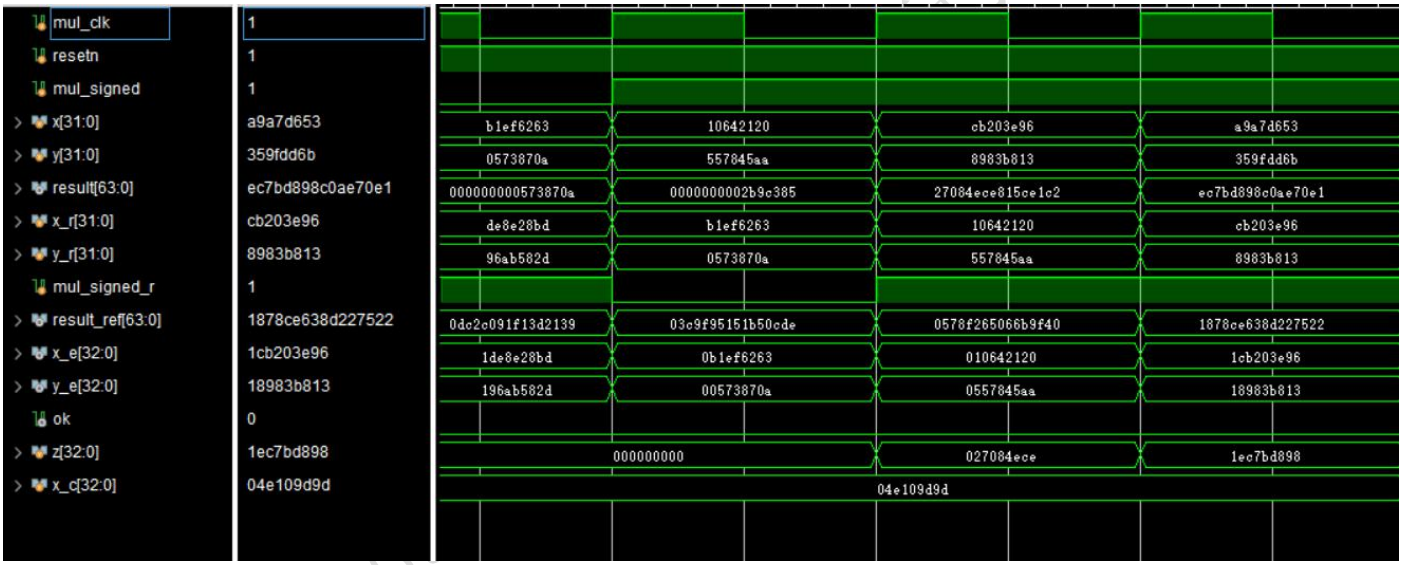
最后发现是我的乘法器有问题，乘法器问题在下个错误再说。但在测试时接入 IP 核测试时，接错了端口位数，导致最后同样出错，并误以为并非自己乘法器的问题。

（4）修正效果

后来直接用乘号运算，可以正常通过。

7、错误 7

（1）错误现象



（2）分析定位过程

这是检查乘法器，运用 mul_tb 时出现的错误，仔细观察发现和时钟延迟有关，一开始高位都为 0，是因为时钟并没跳到高位处理的阶段。

（3）错误原因

采用了状态机设计，但是并没有做阻塞操作，因此乘法并不能在一个时钟节拍完成，这样在 CPU 中会导致大量 result 的高位为 0，这也就是为什么我当时 ERROR 会取指错误。

（4）修正效果

其实并没有来得及修正这个乘法器加入原本 cpu 中，仍然在考虑状态机是否可行。

四、实验总结（可选）

我可能要重写好多遍 cpu 才能完全理解一个实验。原本有一个根据示例代码改出来的很漂亮的 cpu 想要在上面直接调用 Pipeline 模块改出五级流水，并且以为这个比较简单就稍微拖了点时间，结果发现自己并没能完全理解老师的思路，不知道该如何合理调用这个 pipeline，而 cpu 虽然看着漂亮但是思路不是我的习惯思路，很难下手，最写五级流水基本又相当于重写了一遍。

祝大家中秋快乐已经过时了，以后就祝大家中秋写完代码吧。

//

ERROR 的时候不能紧盯这个 ERROR，因为测试用例并不会在测试点外把某些错误暴露出来，很多时候需要向前找，实际上可能之前的结果已经出了查错，才导致之后的取指等操作读入失误。

国庆真的没放假，OS 和 CPU 两大实验压着根本不可能出去玩的，然后其他系的都一边浪着一边看着我们说：“也不看看自己什么系。”

piazza 是个好东西，考虑以后稍微晚一点做，可以早点借鉴别人的弯路经验，同时也有利于 debug。