实验 5-3 报告

学号：2016K8009929029
姓名：张丽玮

一、实验任务（10%）

第一阶段实现环境的变化，在虚拟 cpu 上完成一个类 SRAM 接口到 AXI 接口的转换桥，方便后续实验。

(1) 完全带握手的类 SRAM 接口到 AXI 接口的转换桥 RTL 代码编写。

(2) 通过简单的读写测试。

第二阶段改为在原有自己 cpu 基础上修改总线接口，进行试验。

(1) CPU 顶层修改为 AXI 接口。CPU 对外只有一个 AXI 接口，需内部完成取指和数据访问的仲裁。

(2) 集成到 SoC_AXI_Lite 系统中。

(3) 完成固定延迟的功能测试。

第三阶段完成随机延迟的功能测试。

二、实验设计（30%）

设计了类 SRAM 接口到 AXI 接口的转换桥。

类 SRAM 接口为 inst\data 双通道，AXI 接口为 r/w 双通道；转换桥内部采用 r/w 双通道独立状态机.

信号	位宽	方向	功能	备注
AXI 时钟与复位信号				
aclk	1	input	AXI 时钟	
aresetn	1	input	AXI 复位，低电平有效	
读请求地址通道，（以 ar 开头）				
arid	[3:0]	master—>slave	读请求的 ID 号	取指为 0 取数为 1
araddr	[31:0]	master—>slave	读请求的地址	
arlen	[7:0]	master—>slave	读请求控制信号，请求传输的长度(数据传输拍数)	固定为 0
arsize	[2:0]	master—>slave	读请求控制信号，请求传输的大小(数据传输每拍的字节数)	
arburst	[1:0]	master—>slave	读请求控制信号，传输类型	固定为 2'b01
arlock	[1:0]	master—>slave	读请求控制信号，原子锁	固定为 0
arcache	[3:0]	master—>slave	读请求控制信号，CACHE 属性	固定为 0
arprot	[2:0]	master—>slave	读请求控制信号，保护属性	固定为 0
arvalid	1	master—>slave	读请求地址握手信号，读请求地址有效	
arready	1	slave—>master	读请求地址握手信号，slave 端准备好接受地址传输	
读请求数据通道，（以 r 开头）				
rid	[3:0]	slave—>master	读请求的 ID 号，同一请求的 rid 应和 arid 一致	指令回来为 0

				数据回来为 1
rdata	[31:0]	slave—>master	读请求的读回数据	
rresp	[1:0]	slave—>master	读请求控制信号，本次读请求是否成功完成	可忽略
rlast	1	slave—>master	读请求控制信号，本次读请求的最后一拍数据的指示信号	可忽略
rvalid	1	slave—>master	读请求数据握手信号，读请求数据有效	
rready	1	master—>slave	读请求数据握手信号，master 端准备好接受数据传输	
写请求地址通道，（以 aw 开头）				
awid	[3:0]	master—>slave	写请求的 ID 号	固定为 1
awaddr	[31:0]	master—>slave	写请求的地址	
awlen	[7:0]	master—>slave	写请求控制信号，请求传输的长度(数据传输拍数)	固定为 0
awsize	[2:0]	master—>slave	写请求控制信号，请求传输的大小(数据传输每拍的字节数)	
awburst	[1:0]	master—>slave	写请求控制信号，传输类型	固定为 2'b01
awlock	[1:0]	master—>slave	写请求控制信号，原子锁	固定为 0
awcache	[3:0]	master—>slave	写请求控制信号，CACHE 属性	固定为 0
awprot	[2:0]	master—>slave	写请求控制信号，保护属性	固定为 0
awvalid	1	master—>slave	写请求地址握手信号，写请求地址有效	
awready	1	slave—>master	写请求地址握手信号，slave 端准备好接受地址传输	

写请求数据通道，（以 w 开头）				
wid	[3:0]	master—>slave	写请求的 ID 号	固定为 1
wdata	[31:0]	master—>slave	写请求的写数据	
wstrb	[3:0]	master—>slave	写请求控制信号，字节选通位	
wlast	1	master—>slave	写请求控制信号，本次写请求的最后一拍数据的指示信号	固定为 1
wvalid	1	master—>slave	写请求数据握手信号，写请求数据有效	
wready	1	slave—>master	写请求数据握手信号，slave 端准备好接受数据传输	
写请求响应通道，（以 b 开头）				
bid	[3:0]	slave—>master	写请求的 ID 号，同一请求的 bid、wid 和 awid 应一致	可忽略
bresp	[1:0]	slave—>master	写请求控制信号，本次写请求是否成功完成	可忽略
bvalid	1	slave—>master	写请求响应握手信号，写请求响应有效	
bready	1	master—>slave	写请求响应握手信号，master 端准备好接受写响应	

以上为这个转换桥的接口设置。

独立状态机一个 always 块控制读，一个 always 块控制写，这样实现读写独立，就可以同时执行。但是因为读写有相关——读后写相关、写后读相关，因此增加了一个阻塞延迟信号。

```
reg         wr_haz;
reg         rw_haz;
```

```
  wr_haz    <= ^w_status && data_addr[31:2] == w_addr[31:2];
```

写后读的阻塞延迟，根据 w_status 判断。

```
  rw_haz    <= ^r_status && data_addr[31:2] == r_addr[31:2];
```

读后写的阻塞延迟，根据 r_status 判断。

----------lab5-2 更新----------

• 转换桥更新

由于 swl 和 swr 是三字节输入，为了适应这个，将转换桥的 size 改用 3 字节，

```
  input  wire [ 2:0] inst_size    ,
```

其中最高位是用来表示这是一个三字节请求的，所以 arsize 的相关逻辑改动如下：

```
assign arsize  = {1'b0, r_size[2] ? 2'b10 : r_size[1:0]};
```

并且对于 size 的 3 字节判断：

```
always @(*) begin
    case({w_size, w_addr[1:0]})
    5'b000_00: wstrb = 4'b0001; // SB
    5'b000_01: wstrb = 4'b0010; // SB
    5'b000_10: wstrb = 4'b0100; // SB
    5'b000_11: wstrb = 4'b1000; // SB
    5'b001_00: wstrb = 4'b0011; // SH
    5'b001_10: wstrb = 4'b1100; // SH
    5'b010_00: wstrb = 4'b1111; // SW
    5'b100_00: wstrb = 4'b0001; // SWL
    5'b100_01: wstrb = 4'b0011; // SWL
    5'b100_10: wstrb = 4'b0111; // SWL
    5'b100_11: wstrb = 4'b1111; // SWL
    5'b101_00: wstrb = 4'b1111; // SWR
    5'b101_01: wstrb = 4'b1110; // SWR
    5'b101_10: wstrb = 4'b1100; // SWR
    5'b101_11: wstrb = 4'b1000; // SWR
    default :  wstrb = 4'b0000;
    endcase
end
```

后来发现给的框架不是 lab5-1 那样的……于是这个就暂时废弃……

更新对于下一条 pc 的跳转逻辑,增加了 pc_refresh 信号:

```
else if (PC_refresh) begin
    PC          <= inst_addr;               //To axi_araddr
    PC_next     <= inst_addr + 32'd4;       //To PC_mux For next value of PC
    PC_buffer   <= PC;                      //Wait to write into PC_IF_ID
    PC_AdEL     <= |PC[1:0] ? 1'b1 : 1'b0;  //For PC address excpetion
end
```

更新 fetch 逻辑，增加握手信号：

```
//AXI NEW
    input       [ 1:0] data_r_req,              //Doing data read request

    output             fetch_axi_rready ,
    input              fetch_axi_rvalid ,
    input       [31:0] fetch_axi_rdata  ,
    input       [ 3:0] fetch_axi_rid    ,

    input              fetch_axi_arready,

    input              decode_allowin,
    output             fe_to_de_valid,

    output             IR_buffer_valid
```

更新 decode 逻辑，增加握手信号、例外中断延迟信号

```
//AXI NEW
    input              ex_int_handling,
    input              eret_handling,

    output               de_to_exe_valid,
    output               decode_allowin,
    input                   exe_allowin,
    input                fe_to_de_valid,

    output               exe_refresh,
    output               decode_stage_valid
  );
```

```
assign decode_ready_go = !ID_EX_Stall;
assign decode_allowin = !decode_valid || exe_allowin && decode_ready_go;
assign de_to_exe_valid = decode_valid&&decode_ready_go;
```

以及我不得不承认老师刚开学的时候说的是对的……到了总线的时候，你总会用上 ready_go 的。

更新 execute 逻辑，增加握手信号、例外中断延迟信号

```
//AXI NEW
    output reg  [ 1:0]   s_vaddr_EXE_MEM,
    output reg  [ 2:0]    s_size_EXE_MEM,

    output reg              ex_int_handling,
    output reg              eret_handling,

    input                   mem_allowin,
    input                de_to_exe_valid,
    output                  exe_allowin,
    output               exe_to_mem_valid,

    output               exe_stage_valid,
    input                ID_EXE_Stall,

    output                  exe_ready_go,
    input       [31:0]      epc_value,
    input       [31:0]            PC
```

更新（可能是全部重构）memory 逻辑，输入各种信号，判断握手、延迟、数据相关（太多不贴了）

更新 writeback 逻辑，增加握手信号。

```
//AXI NEW
    output                  wb_allowin,
    input                mem_to_wb_valid,
    output                wb_stage_valid
```

更新 bypass 逻辑，增加 valid 信号，加在 haz 信号后用于判断延迟

```
//AXI NEW
    input   is_j_or_b,

    input   de_valid,
    input   wb_valid,
    input   exe_valid,
    input   mem_valid
```

更新 cp0reg 逻辑，增加 ready_go 信号，判断 epc 延迟

```
    input                   exe_ready_go,
    input                   exe_refresh
```

三、实验过程（60%）

（一）实验流水账

11 月 28 日，下午 4 点到晚上 10 点，阅读任务书，弄清楚新环境的框架

11 月 29 日，下午 4 点到第二天 2 点，写转换桥代码，测试 xxx

11 月 30 日，下午 6 点到晚上 12 点，debug，通过一部分测试，继续 debug

12 月 1 日，下午 6 点到晚上 10 点，通过测试

12 月 4 日，下午 6 点到晚上 12 点，阅读任务书准备开始 5-2 实验

12 月 5 日，下午 4 点到晚上 12 点，编写代码

12 月 6 日，下午 4 点到晚上 12 点，编写代码

12 月 7 日，下午 6 点到晚上 12 点，debug 调波形

12 月 8 日，下午 6 点到晚上 12 点，debug 调波形

12 月 9 日，下午 6 点到晚上 12 点，debug 调波形

12 月 10 日，下午 6 点到晚上 10 点，调出来了……PASS

（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

测试点 0pass 之后读取数据一直 xxx

（2）分析定位过程

有 Z 调 Z，有 X 调 X，发现 inst 和 data 的 ok 信号一直是 0

（3）错误原因

逻辑出了问题，握手了也没有接收数据

（4）修正效果

可以读入数据

（5）归纳总结（可选）

看波形真的很头秃。

2、错误 2

（1）错误现象

读 0x00008002 和 0x00008000 错误

（2）分析定位过程

0x00008000 是第一条请求，第一条请求未成功

（3）错误原因

valid 信号未初始化，

（4）修正效果

```
w_data      <= 32'd0;
awvalid_en  <= 1'b0;
wvalid_en   <= 1'b0;
```

增加了初始化。

（5）归纳总结（可选）

无。

3、错误 3

（1）错误现象

读 0x00008040 有误。

（2）分析定位过程

跟踪波形发现前一条的写指令没有执行，因此读到的是上一条指令的值。

（3）错误原因

只进行了读写独立处理，但是没有考虑读后写、写后读相关

（4）修正效果

增加了延迟阻塞信号，判断冲突，如果冲突进行阻塞。

（5）归纳总结（可选）

调试波形的时候不仅要考虑当前错误值，可能还要追溯前面几条找到根源。

4、错误 4

（1）错误现象

第一个测试点就有错，0000aaaa 而我是 00000000

（2）分析定位过程

第一条指令 00000000 没读进去

（3）错误原因

框架问题

（4）修正效果

重新下载 v0.03 就好了

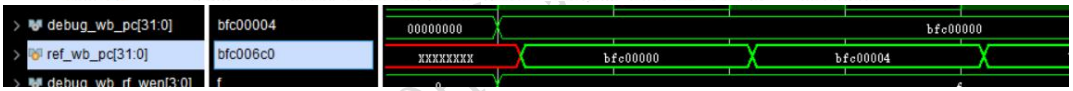
（5）归纳总结（可选）

多看 piazza……

5、错误 5

（1）错误现象

```
[   2107 ns] Error!!!
    reference: PC = 0xbfc00004, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff
    mycpu    : PC = 0xbfc00000, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff
```



（2）分析定位过程

猜测是 pc 跳转问题

（3）错误原因

```
end
 //多位选择?
MUX_4_32 PCS_MUX(
    .Src1   (         nextpc),
    .Src2   (  Jump_addr),
    .Src3   (    Br_target_DE),
    .Src4   (      32'd0),
    .op     (      PCSrc),
    .Result (  PC_mux)//new
);
//计算下一个PC
```

这里 nextpc 当时写成了 pc

```
assign  JSrc =  JSrc_DE & ~(ex_int_handling|eret_handling);
assign PCSrc = PCSrc_DE & {2{~(ex_int_handling|eret_handling)}};
```

这一条跳转信号逻辑出错

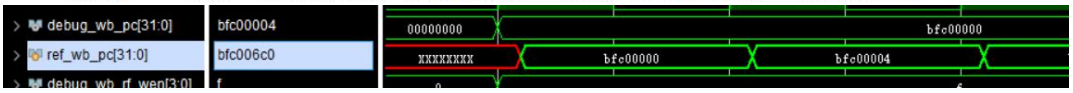
（4）修正效果

仍然有问题

6、错误 6

（1）错误现象

```
[   2107 ns] Error!!!
    reference: PC = 0xbfc00004, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff
    mycpu    : PC = 0xbfc00000, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff
```



（2）分析定位过程

猜测是 pc 跳转问题，但是改完之后仍然不能跑，猜测是写回问题

（3）错误原因

写回级没有把加入的信号融入逻辑中

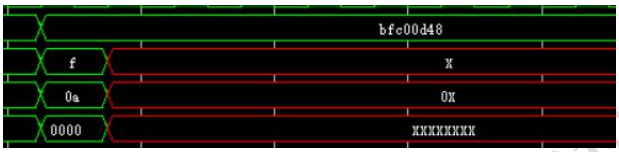
（4）修正效果

可以跑了

7、错误 7

（1）错误现象

```
[  62235 ns] Number 8'd01 Functional Test Point PASS!!!
[  72000 ns] Test is running, debug_wb_pc = 0xbfc00d48
[  82000 ns] Test is running, debug_wb_pc = 0xbfc00d48
[  92000 ns] Test is running, debug_wb_pc = 0xbfc00d48
```



第一个测试点 pass 之后时钟停在 0xbfc00d48

（2）分析定位过程

猜测是 bypass 的延迟问题

（3）错误原因

是 rs 和 rt 的传参错了……

（4）修正效果

可以跑了

四、实验总结（可选）

很长一段时间没有理解实验到底要我们做什么，可能因为换了环境，不是在原 cpu 基础上改。

一开始以为这个握手信号和上学期计组实现的差异不大，写一个简单的状态机判断读写就行，然后发现差得很大……听课的时候草草听过去了但是没实际概念，实际进行的时候就知道这个东西调起来很烦有很多坑……

现在对于接下来两周实验充满了畏惧。

我的畏惧是对的……我现在一点也不想见到我的辣鸡 cpu 了，事实证明早点领悟老师代码风格的真谛绝对是有裨益的……愿天堂没有总线。