Lab 7 报告

学号 2020K8009908024
姓名 陈卓
箱子号 69

一、实验任务（10%）

任务一：将原有 CPU 访问 SRAM 的接口调整为类 SRAM 总线接口。在取指令和取数据的总线接口上添加握手信号，实现随机延迟的数据获得。检测通过：在采用握手机制的 block RAM 的 SoC 验证环境中完成 exp14 对应 func 的随机延迟功能验证，通过仿真测试；上板验证时，要求“随意切换拨码开关后按复位键”，CPU 能通过对应 exp14 的 func 中 58 个功能点的验证。

任务二：将原有 CPU 访问 SRAM 的接口调整为 AXI 总线接口。在读通道和写通道的总线接口上添加握手信号，实现随机延迟的数据获得。检测通过：在采用握手机制的 block RAM 的 SoC 验证环境中完成 exp16 对应 func 的随机延迟功能验证，通过仿真测试；上板验证时，要求“随意切换拨码开关后按复位键”，CPU 能通过对应 exp16 的 func 中 58 个功能点的验证。

二、实验设计（40%）

（一）总体设计思路

类 SRAM 总线接口：

（1） top 模块：

首先更改模块接口中关于取指令和取数据的信号接口，主要分为 inst_sram 和 data_sram 的数据传输信号以及相应的握手信号。

对于取指令的操作，写相关的就没必要再设计到流水线的取指模块中，直接在 top 模块中定义好数值。

```
assign inst_sram_wr     = 1'b0;
assign inst_sram_size   = 2'h2;
assign inst_sram_wstrb  = 4'h0;
assign inst_sram_wdata  = 32'h0;
```

（2） PIF 级：

本次设计中仍然延续之前的设计，在 PIF 级发出取指请求，在 IF 级得到指令。

首先，给 PIF 级添加 valid、readygo 和 to_fs_vaid 信号。其中 ready_go 信号只在取指请求发送成功时有效，取指请求成功指的是 req 信号和 addr_ok 信号同时为 1 的时候。

```
assign pfs_valid     = ~reset;
assign to_fs_valid    = pfs_valid && to_fs_ready_go;
assign to_fs_ready_go = inst_sram_req & inst_sram_addr_ok ;
```

其次 PIF 级最重要的信号就是 nextpc，因为取指的地址仍然还是 nextpc。如下图所示，添加了两个 buffer 信号，一个是给中断和例外的，另一个是给 br 指令的。因为在这种随机时间取得指令和数据的情况下，不知道什么时候可以拿到数据。但是 br_taken 信号和 wb_ex|wb_ertn_flush 信号只能维持一拍就会消失，因此如果遇到了例外中断或者分支命中，但是这时候 IPF 级的 req 和 addr_ok 没办法握手成功的时候，信号就会丢失。因此添加了两个缓存来存储这两种信号，取指请求成功之后，也就是 pfs_ready_go 的时候，就可以把这两个缓存清空。

至于 inst_sram_req 信号在设计的时候采取了讲义上后一种方法，只在 fs_allowin 的时候才为 1.

```
assign next_pc = pc_buff_ex_valid ?        pc_buff_ex :
                 wb_ex|wb_ertn_flush ?     csr_pc  :
                 pc_buff_br_valid    ?     pc_buff_br:
                 !fs_allowin          ?    fs_pc:
                 br_taken            ?     br_target:
                                           seq_pc;

assign seq_pc = fs_pc+32'h4;

assign inst_sram_req   = pfs_valid & fs_allowin ;
assign inst_sram_addr  = next_pc;
```

```
always @ (posedge clk) begin
    if (reset) begin
        pc_buff_br_valid  <= 1'b0;
        pc_buff_br        <= 32'h0;
        pc_buff_ex_valid  <= 1'b0;
        pc_buff_ex        <= 32'h0;
    end else if ( (wb_ex|wb_ertn_flush) && !to_fs_ready_go) begin
        pc_buff_ex_valid  <= 1'b1;
        pc_buff_ex        <= csr_pc;
    end else if (br_taken && !to_fs_ready_go) begin
        pc_buff_br_valid  <= 1'b1;
        pc_buff_br        <= br_target;
    end else if (to_fs_ready_go) begin
        pc_buff_br_valid  <= 1'b0;
        pc_buff_br        <= 32'h0;
        pc_buff_ex_valid  <= 1'b0;
        pc_buff_ex        <= 32'h0;
    end
end
```

（3） IF 级：

IF 级最重要的设计就是获得返回的 inst，只有当 data_ok 信号有效的时候得到的 inst 才是有效的，但是 data_ok 信号是不确定时间有效，一次指令返回有效拍数只维持一拍。因此，如果出现了取到了有效指令，但是 ds_allowin 却无效的时候，这个时候无法流水到 ds 级。然而下一周期的时候仍然可能会得到一个有效的 inst，这个时候，上一个就会被覆盖掉，所以这里同样设计一个缓存，当出现 ds 无法 allowin 的时候，就存在缓存里面。待可以流通或者例外发生就直接清空。

```
assign fs_inst         = fs_inst_buff_valid ? fs_inst_buff : inst_sram_rdata;
always @ (posedge clk) begin
    if (reset) begin
        fs_inst_buff_valid  <= 1'b0;
        fs_inst_buff        <= 32'h0;
    end else if (!fs_inst_buff_valid && fs_valid && inst_sram_data_ok && !cancel && !ds_allowin) begin
        fs_inst_buff_valid  <= 1'b1;
        fs_inst_buff        <= inst_sram_rdata;
    end else if (ds_allowin || wb_ertn_flush || wb_ex) begin
        fs_inst_buff_valid  <= 1'b0;
        fs_inst_buff        <= 32'h0;
        end
end
```

接下来就是 fs_ready_go 信号的设计了，之后得到了有效的 inst 才能有效。这里会出现一种特殊的情况，当例外或者分支命中的时候，如果需要被取消指令的 inst 请求已经完成，这时候第一个到达的 data_ok 就是该指令的，需要被取消掉。但是这样就会出现另外一个问题，就是要被取消的 pc 如果被阻塞，就会拿到自己的 inst，这个时候再 cancel 的时候就会多取消一个 data_ok，因此设计 cancel_flag 信号来防止多取消。

```
assign fs_ready_go    = (!cancel && inst_sram_data_ok) | fs_inst_buff_valid|(wb_ex | wb_ertn_flush);
always @(posedge clk) begin
     if(reset) begin
        cancel_flag<=1'b0;
     end else if(fs_valid && !ds_allowin && inst_sram_data_ok) begin
        cancel_flag<=1'b1;
     end else if(ds_allowin) begin
        cancel_flag<=1'b0;
     end
end

always@(posedge clk) begin
     if(reset) begin
        cancel<=1'b0;
     end else if ((wb_ex | wb_ertn_flush|br_taken) && fs_valid && !inst_sram_data_ok && !cancel_flag)begin
        cancel<=1'b1;
     end else if(inst_sram_data_ok)begin
        cancel<=1'b0;
     end
end
```

（4） ID 级：

译码级要添加的东西不多，一个就是 MEM 的前递，不再是原来那样了。由于 MEM 级不知道什么时候才能拿到 load 的数据，会发生阻塞。原来只有 EXE 级才会阻塞，现在 MEM 级也要前递一个关于 load 指令的阻塞。

```
assign ds_ready_go    = !(
    es_blk_valid  && !wb_ertn_flush && !wb_ex && (es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
    ms_blk_valid  && !wb_ertn_flush && !wb_ex && (ms_rf_dest == rf_raddr1 || ms_rf_dest == rf_raddr2) ||
    es_csr_blk_valid && !wb_ertn_flush && !wb_ex && (es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
    ms_csr_blk_valid && !wb_ertn_flush && !wb_ex && (ms_rf_dest == rf_raddr1 || ms_rf_dest == rf_raddr2) ||
    ws_csr_blk_valid && !wb_ertn_flush && !wb_ex && (rf_waddr == rf_raddr1 || rf_waddr == rf_raddr2));
```

接下来就是 br 指令的相关问题，原来可以通过阻塞来不改变 nextpc 的值,但是现在随机延迟，可能相关了，但是 fs_allowin 照样有效。所以，基于先前 nextpc 的设计，这里只需要在发生相关的时候，令 br_taken 无效即可。

```
wire    br_stall;
assign  br_stall =  es_blk_valid  && (es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
                    ms_blk_valid  && (ms_rf_dest == rf_raddr1 || ms_rf_dest == rf_raddr2) ||
                    es_csr_blk_valid && (es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
                    ms_csr_blk_valid && (ms_rf_dest == rf_raddr1 || ms_rf_dest == rf_raddr2) ||
                    ws_csr_blk_valid && (rf_waddr == rf_raddr1 || rf_waddr == rf_raddr2);


assign br_taken = (   (inst_beq  &  rj_eq_rd)
                    | (inst_bne  & !rj_eq_rd)
                    | inst_jirl
                    | inst_bl
                    | inst_b
                    | (inst_blt & rj_lt_rd)
                    | (inst_bltu & rj_ltu_rd)
                    | (inst_bge & !rj_lt_rd)
                    | (inst_bgeu & !rj_ltu_rd)
                   )&&ds_valid && !br_stall;
```

（5） EXE 级：

这个阶段要添加的东西，就是 load/store 指令的请求、地址、以及 size 和 wstrb。

```
assign data_sram_req   =es_valid && ms_allowin && (es_load_op|es_st_op )&& !wb_ex && !wb_ertn_flush;
assign data_sram_wr    = es_load_op ? 1'b0 : 1'b1;
assign data_sram_size  = es_load_op ? 2'h2 : st_size;
assign data_sram_wstrb    = (es_valid & es_mem_we & !es_ex_ale & !wb_ex & !wb_ertn_flush & !ms_ex & !ms_ertn_flush) ? st_op : 4'h0;
assign data_sram_addr  = es_alu_result;
assign data_sram_wdata = st_data;
```

（6） MEM 级：

在这一级，需要得到 data_ok 才能 ready_go 有效

```
assign ms_ready_go    =  (wb_ex || wb_ertn_flush)  ? 1'b1:
                         (ms_load_op|ms_st_op) ? (data_sram_data_ok | ms_data_buff_valid):
                          1'b1;
```

同时需要更改前递相关的信号，因为在这一级也可能阻塞。添加了这一级前递的阻塞信号，更改了前递数据有效的信号，ms_valid 更改为 ms_to_ws_valid.

```
assign ms_blk_valid = ms_load_op && !ms_to_ws_valid && ms_valid;
assign ms_fwd_valid = ms_load_op ? {4{ ms_gr_we && ms_to_ws_valid }}:{4{ ms_valid && ms_gr_we}};
```

类 SRAM-AXI 转接桥

（1）工作原理

AXI 总线接口有五类通道。

与读相关的通道有读请求通道和读响应通道。当主设备的 arvalid 和从设备的 arready 同时为 1 时，读地址握手成功；主设备的 rvalid 和从设备的 rready 同时为 1 时，读响应成功，此时读的数据也被传送到主设备。

写相关的通道有写请求、写数据和写响应通道。当主设备的 awvalid 和从设备的 awready 同时为 1 时，写地址握手成功；主设备的 wvalid 和从设备的 wready 同时为 1 时，写数据握手成功；主设备的 bready 和 bvalid 同时为 1 时，写响应握手成功。

在转接桥中，地址握手成功时，对应的 addr_ok 拉高；响应握手成功时，对应 data_ok 拉高。

（2）接口定义

转接桥的接口中，连接 CPU 核的为类 SRAM 总线相关的接口信号，连接内存的为 AXI 总线相关的信号。

（3）读请求、读响应

读请求的状态机如下：

```
always @(posedge aclk)begin
if(areset)begin
    arvalid_r <= 1'b0;
    arid_r    <= 4'b0010;
    arsize_r  <= 3'h0;
    araddr_r  <= 32'h0;
end else if(!arvalid_r && (axi_rreq || ar_buff_valid) && write_finish)begin
    arvalid_r <= 1'b1;
    arid_r    <= rreq_id;
    arsize_r  <= rreq_size;
    araddr_r  <= rreq_addr;
end else if(arvalid_r && arready)begin
    arvalid_r <= 1'b0;
    arid_r    <= 4'b0010;
    arsize_r  <= 3'h0;
    araddr_r  <= 32'h0;
end
end
```

读请求设置了一个缓存。当主设备发来读请求、读请求缓存的 arvalid_r 为 0、所有写的行为均处理完毕（即写响应握手成功）后，读请求相关的信号被写入缓存，且置 arvalid_r 为 1。当读地址握手成功后，清空缓存并置 arvalid_r 为 0。

由于读行为有取指和读数据两种读请求，因此当二者同时发送请求时，需要进行仲裁。在我们的转接桥中，读内存的优先级比取指的优先级高。

本转接桥中没有设置 fifo_buffer，因为在我们的 cpu 中，只有在 br_taken、wb_ex 或 wb_ertn 拉高时，由于此时取指级可能还没拿到指令，同一时间会有两个没有得到读响应握手的读请求，其他情况下都最多只会有一个读响应没有拿到读请求。因此，我们没有设置 buffer 来缓存多个读请求——这同样不影响 br_taken、wb_ex 和 wb_ertn。当误取的指令还没地址握手时，跳转或异常的取指级的 req 会一直拉高，直至得到自己的读响应。

但注意到我们设置了一个 ar_buff_valid，它对应的是 ar_buff。因为当写回级的 wb_ex 拉高或者译码级是跳转指令，执行级的读请求还未得到地址握手时，取指级的读地址由于优先级较低，还未存入读请求缓存。当进入下一时钟周期时，例外入口的 pc 会覆盖原取指级的 pc。这就导致原取指级的读请求没有真正向内存发出，导致没有得到响应。这与 AXI 总线的有请求必须接响应的设计原理不符。因此，我们设置了一个 ar_buff，用于缓存这种情况下原取指级的读请求，让它也能得到响应。

对于读响应，我们没有设置专门的状态机。我们将 rready 一直拉高，当从设备的 rvalid 拉高时，数据握手成功。我们也没有设计读数据的缓存，因为 cpu 中已经设置了相应的读缓存（fs_inst_buff 和 mem_data_buff）。这样的设计对于这种简陋的 CPU 已经够用了。

（4）写请求、写数据、写响应

写请求和写数据设置了一个缓存。当主设备发来写请求且写请求缓存的 arvalid_r 为 0 时，写请求、写数据被写入缓存，且置 wrvalid_r、wvalid_r 为 1。当读地址握手成功后，清空缓存并置 wrvalid_r、wvalid_r 为 0。

```
always @(posedge aclk)begin
    if(areset)begin
        awvalid_r <= 1'b0;
        awsize_r  <= 3'h0;
        awaddr_r  <= 32'h0;
    end else if(!awvalid_r && axi_wreq && !wvalid_r)begin
        awvalid_r <= 1'b1;
        awsize_r  <= data_sram_size;
        awaddr_r  <= data_sram_addr;
    end else if(awvalid_r && awready)begin
        awvalid_r <= 1'b0;
        awsize_r  <= 3'h0;
```

```
        awaddr_r  <= 32'h0;
    end

    if(areset)begin
        wvalid_r <= 1'b0;
        wdata_r <= 32'b0;
        wstrb_r <= 4'b0;
    end else if(!awvalid_r && axi_wreq)begin
        wvalid_r <= 1'b1;
        wdata_r <= data_sram_wdata;
        wstrb_r <= data_sram_wstrb;
    end else if(wvalid_r && wready)begin
        wvalid_r <= 1'b0;
        wdata_r <= 32'b0;
        wstrb_r <= 4'b0;
    end
end
```

由于读请求需要在其之前所有写请求得到写响应后才会写入读请求缓存，因此我们通过 write_count 来统计未完成的写请求的数目。当 write_count 为 0 时，所有写请求处理完毕，拉高 write_finish，允许读请求缓存读入读请求。

三、实验过程（50%）

（一）实验流水账

第一周：周三周四看讲义，尝试写代码，结果一开始就跑不通。周五到周一，一直在 debug，一个逻辑可能要更改很多次。

第二周：周五看讲义，周六开始设计并 debug，周一完成实验。

第三周：周日开始 debug，于周一完成。

（二）错误记录

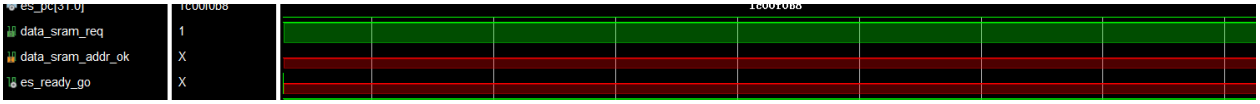
1、错误 1：data_sram_req 信号初始值为 x

（1）错误现象

PC 一直不更新，一直阻塞在 EX 阶段。

（2）分析定位过程

检查了一下 es_ready_go 发现它一直为 x，往前检查相关信号，发现是 data_sram_addr_ok 信号一直为 x，在 piazza 上面看到了同学提问的类似的问题，发现是 data_sram_req 信号初始值为 x 才会这样。然后就去检查 req 信号的组合逻辑代码，发现刚 reseten 的时候 req 信号确实为 x,与上了 es_valid 信号解决了初始值为 x 的情况。



（3）错误原因

data_sram_req 信号初始值为 x

（4）修正效果

改成如下代码后，问题解决。

```
assign data_sram_req    =es_valid && ms_allowin && (es_load_op|es_st_op )&& !wb_ex && !wb_ertn_flush;
```

（5）归纳总结（可选）

一定要考虑这些新添加信号的初始值，虽然之后都会改成正确的，但是有的可能刚开始的值也会影响后面的逻辑。

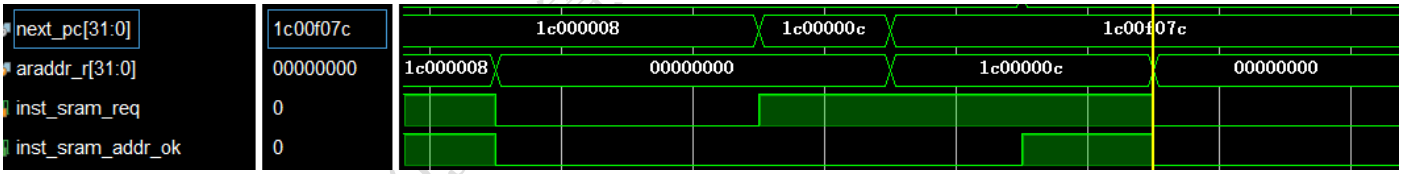
2、错误 2：跳转时误取指令后未取消

（1）错误现象

PC 一直不更新

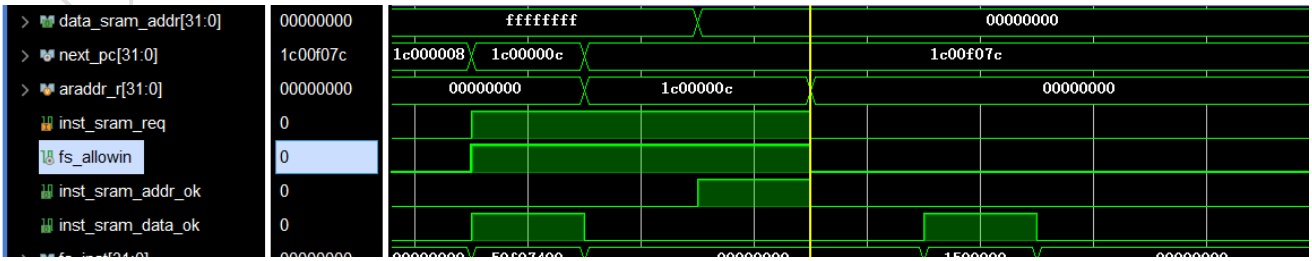
```
Test begin!
         [  22000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  32000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  42000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  52000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  62000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  72000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  82000 ns] Test is running, debug_wb_pc = 0x1c000008
         [  92000 ns] Test is running, debug_wb_pc = 0x1c000008
```

（2）分析定位过程



1c000008 为跳转指令，因此 1c00000c 为误取的。但取指阶段将 1c00000c 的地址握手误判成 1c00f07c（跳转地址）的地址握手。

由 inst_sram_req 的赋值，调出 fs_allowin 信号，发现其此时拉低了，不符合预期。而 fs_allowin 拉低的原因是 fs_valid 拉高了。注意到此时的指令是误取的，因此 fs_valid 应该为 0。



（3）错误原因

fs_valid 在跳转后未取消误取的指令。

（4）修正效果

修改 fs_valid 对应的状态机。

```
always @(posedge clk) begin
    if (reset) begin
        fs_valid <= 1'b0;
    end
    else if(br_taken && ds_allowin || cancel)begin
        fs_valid<=1'b0;
    end
    else if (fs_allowin) begin
        fs_valid <= to_fs_valid;
    end

end
```

增加 cancel 信号，取消误取的指令。

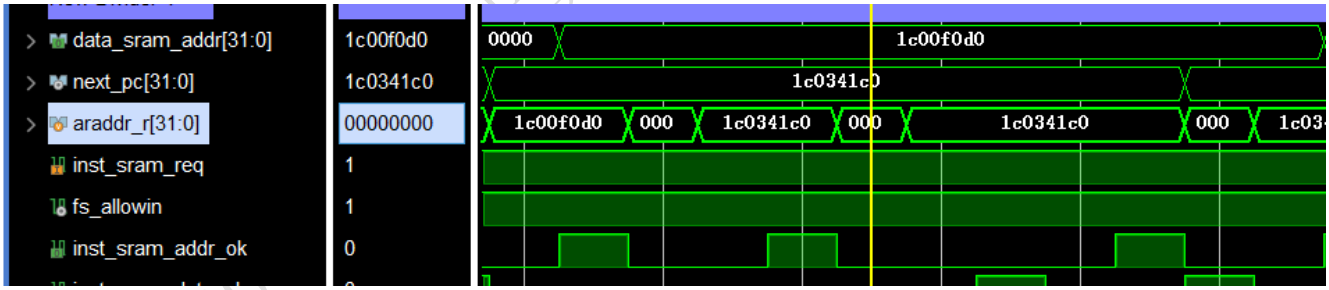
3、错误 3：跳转后重复取指

（1）错误现象

PC 一直不更新

```
Test begin!
----------------------------------------------------------------
[   4577 ns] Error!!!
    reference: PC = 0x1c0341c4, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c0341c4, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000002
```

（2）分析定位过程



1c341c0 取了两次指令，导致结果出错。此时 inst_sram_req 一直是 1，因为 fs_allowin 也是 1。

一开始想改动 fs_allowin。但检查后发现 fs_allowin、fs_valid 是符合逻辑的。因此考虑增加一个信号，来拉低 inst_sram_req。

观察发现，1c341c0 第一次取指时，已经有两两个 pc（包括误取的 pc）在等待指令。因此，增加信号，使得这两个 pc 都收到指令后，取指级再发送请求。

（3）错误原因

同一 pc 多次取指。

（4）修正效果

做如下修改。

```
always @(posedge clk)begin//------------
    if(reset)begin
        req_count <= 2'b00;
    end else if(inst_sram_addr_ok && !inst_sram_data_ok)begin
        req_count <= req_count + 2'b01;
    end else if(!inst_sram_addr_ok && inst_sram_data_ok)begin
        req_count <= req_count - 2'b01;
    end

    if(reset)begin
        cancel_req <= 1'b0;
    end else if(req_count == 2'b01 && inst_sram_addr_ok && !inst_sram_data_ok)begin
        cancel_req <= 1'b1;
    end else if(req_count == 2'b01 && !inst_sram_addr_ok && inst_sram_data_ok)begin
        cancel_req <= 1'b0;
    end
end
```

```
assign inst_sram_req    = pfs_valid & fs_allowin & !cancel_req;
```

这样，当跳转后，同一 pc 就不会重复取指了。