中国科学院大学
《计算机体系结构(研讨课)》实验报告

姓名 裴晨皓 竹彦博 纪弘璐 学号 2023K8009916003 2023K8009916001 2023K8009916002

专业 计算机科学与技术 实验项目编号 Project 5 实验名称 AXI 总线接口设计专题实验

1 实验简介与总体思路

本实验中，我们首先对流水线 CPU 进行改造，把访存接口改为类 SRAM 接口，并将五级流水线改为七级以实现请求和数据响应分离 (利于并行处理提高性能)，然后按照教材提示设计了具有四个状态机的类 SRAM-AXI 转接桥，最终完成了本次实验。

2 设计方案介绍

下图是完成本次实验后的处理器结构框图：

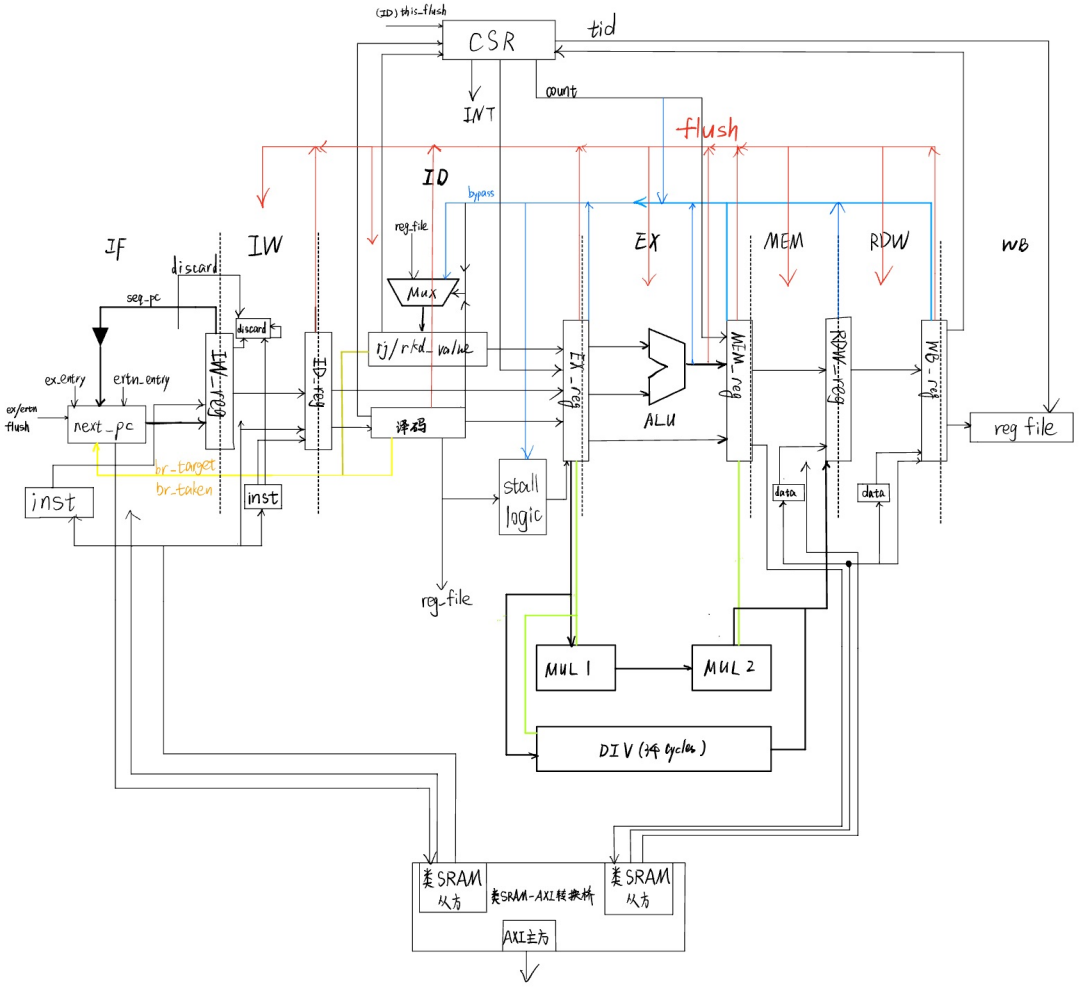

图 1: 处理器结构框图

2.1 流水线处理器本身的进一步改进

2.1.1 五级流水线拆分为七级流水线

进入本次实验，访存操作引入了握手机制，一次访存操作要分为发送请求和等待数据这两次握手 (实际上用类 SRAM 接口的 CPU 的第 2 次“握手”为单向的 data_ok，并非严格意义的握手)。为了提高流水线效率，我们将访存的流水级都拆成两级，一级用于请求，一级用于等待数据，具体表现为：IF 级拆分为 IF(相当于讲义的 pre-IF) 和 IW(Instruction Waiting，相当于讲义的 IF)，MEM 级拆分为 MEM 和 RDW(Read Data Waiting，这是组成原理研讨课时使用的名称，实际上由于 load/store 操作都要等待 data_ok，故不仅是 Read Data，Write Data 时也要在这一级等待)。

2.1.2 IF 阶段的取指请求设计

这部分要重点关注 req 信号的问题：对同一个 PC 而言，在没完成取指令请求握手时，req 要一直拉高；完成握手后，req 要拉低，避免重复取指。为此，我们设置了 handshake_done 信号，用于记录当前 IF 阶段的指令是否已经完成取指请求握手：

```
    always @(posedge clk) begin
        if(rst) begin
            handshake_done <= 1'b0;
        end
        else if(ready_go) begin
            handshake_done <= !out_ready;
        end
        else if(ex_flush || ertn_flush || br_taken) begin
            handshake_done <= 1'b0;
        end
    end
```

Listing 1: handshake_done 信号的定义

考虑以下情况：

1. 在 IF 阶段工作完成 (完成取指，ready_go 为 1) 后，若 IW 不允许流入 (out_ready 为 0)，则要把 handshake_done 拉高，记录当前指令的取指请求握手已完成；若 IW 允许流入 (out_ready 为 1)，则接下来应当发下一条指令的取指请求，故 handshake_done 应该重置为 0。
2. 出现冲刷情况时，由于 IF 在当拍就把 PC 改为冲刷后的新 PC，故 handshake_done 也应重置为 0。

到上面这步还不够，因为若出现冲刷，那么 IF 会在同一周期改为新的 PC，它不能被冲刷前 PC 的握手情况干扰；此外，若冲刷前的 PC 对应请求已经握手成功，在执行流的角度看，这条指令将不会在后续的阶段被执行——根据这两点可以想到：这时的 handshake_done 是无效的。故结合三类冲刷情况，我们另设 handshake_done_effective 信号：

```
    wire handshake_done_effective = handshake_done && !ex_flush && !ertn_flush && !br_taken;
```

Listing 2: handshake_done_effective 信号

此外，对于分支指令跳转冲刷的情况，由于 ID 阶段分支指令可能存在 load-to-branch 情况 (br_stall，在 ID 根据指令类型和已有的 load_use_stall 信号得出)，分支判定结果无法立即得出，所以这时需要暂停取指请求，直到得到分支结果。

进一步地，req 信号也就能得到定义：

```
    assign req = !handshake_done_effective && !(br_stall && ID_in_valid);
```

Listing 3: req 信号定义

从自然语言角度讲，即：当前 PC 没有有效的握手完成行为，且没有未完成的分支判断，就可以发取指请求。

除了 req，取指 PC 也需要更多考虑。原先不存在握手和延迟的取指请求一拍就能完成，但现在大多数时候都需要多拍；又因为取指有时依赖冲刷信号，而 WB 阶段报出的异常冲刷和 ID 给出的分支跳转冲刷有时只能维持一拍，所以要把报出的异常信号暂存下来。以异常冲刷信号为例：

```
    always @(posedge clk) begin
        if(rst) begin
            ex_flush_reg <= 1'b0;
        end
        else if(in_valid && ready_go && out_ready) begin
            ex_flush_reg <= 1'b0;
        end
        else if(ex_flush) begin
            ex_flush_reg <= 1'b1;
        end
    end
```

Listing 4: 异常冲刷信号暂存

在当前指令流入 IW 时，重置 ex_flush_reg 为 0；在 WB 报出异常时，把 ex_flush_reg 拉高，即暂存异常信号。这样一来，在报出冲刷的当拍可以依赖 ex_flush 本身，下一拍则借助 ex_flush_reg，把二者结合得到：

```
    wire ex_flush_preserved = ex_flush | ex_flush_reg;
```

Listing 5: IF 最终使用的异常冲刷信号

下面是异常冲刷情况下，对异常处理入口 PC 的生成 (也是按照上面的逻辑设计)：

```
    wire [31:0] ex_entry_preserved = ex_flush ? ex_entry : ex_entry_reg;
```

Listing 6: IF 异常处理入口 PC

其他信号也与上面类似——把 nextpc 生成逻辑中涉及的冲刷信号和冲刷后重新取指的 PC，都暂存为 xxx_preserved 信号，也就保证了取指 PC 在发请求的过程中始终保持正确：

```
    assign nextpc       = ex_flush_preserved ? ex_entry_preserved :
                          ertn_flush_preserved ? ertn_entry_preserved :
                          br_taken_preserved ? br_target_preserved : seq_pc;
```

Listing 7: nextpc

2.1.3 指令的接收与传递

正常情况应为位于 IW 的 PC 收到 data_ok 和指令，这与“Instruction Waiting”的含义一致。若 IW 收到指令后被阻塞，根据教材的提示：若选择暂停 IF 的取指请求的方式，则性能较差，故我们选择另一种方式——IF 正常发取指请求，若 IF 的 PC 在进入 IW 前就已经收到 data_ok，则把它暂存下来，待 IW 允许流入时再传递下去。

因此，IW 向 ID 传递的指令可分为三种情况 (在流入 ID 的那个上升沿把下面的 wire 信号赋给 ID 的流水寄存器)：

```
    wire [31:0] inst_out_wire = inst_valid_from_IF ? inst_from_IF :
                                inst_valid ? inst :
                                data_ok ? rdata :
                                32'd0;
```

Listing 8: IW 向 ID 传递的指令

1. IW 的 PC 对应的指令在 IF 阶段就已经收到 (inst_valid_from_IF)，后来它又被传给 IW，故 IW 要把它 (inst_from_IF) 继续传递;

2. IW 之前已经收到过 data_ok 并把指令暂存了下来 (inst_valid)，则传递这条暂存的指令 (inst)。

3. 当前这一拍收到 data_ok，则直接把当前访存得到的指令 (rdata) 传递下去。

IF 阶段暂存指令的逻辑如下：

```
always @(posedge clk) begin
    if(rst) begin
        inst_valid <= 1'b0;
        inst <= 32'd0;
    end
    else if(ex_flush || ertn_flush || br_taken) begin
        inst_valid <= 1'b0;
        inst <= 32'd0;
    end
    else if(in_valid && ready_go && out_ready) begin
        inst_valid <= 1'b0;
        inst <= 32'd0;
    end
    else if(data_ok && !out_ready && (inst_valid_out || IW_inst_valid) && (~(|discard)))
        begin
        inst_valid <= 1'b1;
        inst <= rdata;
    end
end
```

Listing 9: IF 对指令的暂存

1. 有冲刷时，要把暂存的指令清空；当前流水级向后流动时，也应如此。

2. 当 IF 被阻塞 (!out_ready)，且 IW 已经收到过来自 IF 或是直接来自内存的指令 (inst_valid_out || IW_inst_valid) 时，若没有因冲刷而需要丢弃的指令 (~(|discard))，则说明马上/现在来的 data_ok 对应于当前 IF 的 PC，故把指令暂存下来。

然后在流入 IW 时，把 inst 传递下去即可。

IW 阶段保存取回指令的逻辑如下：

```
always @(posedge clk) begin
    if(rst) begin
        inst_valid <= 1'b0;
        inst <= 32'd0;
    end
    else if(ex_flush || ertn_flush || br_flush) begin
        inst_valid <= 1'b0;
        inst <= 32'd0;
    end
    else if(data_ok && out_ready && (inst_valid_from_IF || inst_valid) && (~(|discard)))
        begin
        inst_valid <= 1'b1;
        inst <= rdata;
    end
```

```
        else if(data_ok && !out_ready && !(inst_valid_from_IF || inst_valid) && (~(|discard)))
            begin
            inst_valid <= 1'b1;
            inst <= rdata;
        end
        else if(in_valid && ready_go && out_ready) begin
            inst_valid <= 1'b0;
            inst <= 32'd0;
        end
    end
```

Listing 10: IW 阶段保存取回指令

除了冲刷和指令流动后进行重置以外，可以分成以下两种情况来看：

1. IW 的 PC 对应的指令已经收到 (inst_valid_from_IF || inst_valid)，且这一拍能够流入 ID(out_ready)：若这时 data_ok 为 1，说明这是当前 IF 的 PC 对应的指令，则在下一个上升沿保存下来，同时 IF 的 PC 进入 IW，正好对应这条指令。

2. IW 的 PC 对应指令尚未收到 (!(inst_valid_from_IF || inst_valid))，且这一拍不能流入 ID(!out_ready)：若这时 data_ok 为 1，说明这是 IW 的 PC 对应的指令，则在下一个上升沿保存下来，等待流入 ID。

(注意：上述情况都是在没有需要因冲刷而丢弃指令的情况下 (~(|discard)) 描述的。)

2.1.4 冲刷流水线时，对异常前已握手成功的取指请求的处理

我们的设计中，冲刷行为报出时，IF 级 PC 会在同一个周期被设为冲刷后的新值，但此时冲刷前 IF 可能已经取指请求握手成功，IW 可能还未等到冲刷前的请求的 data_ok 返回——也就是说，接下来会有 0 到 2 条应该被冲刷的指令返回，我们需要把它们丢弃掉。discard 计数器用于记录还要丢弃的指令数目：

```
    always @(posedge clk) begin
        if(rst) begin
            discard <= 2'd0;
        end
        else if((|discard) && data_ok) begin
            discard <= discard - 2'b01;
        end
        else if(discard_from_IF ^ discard_from_IW) begin
            discard <= discard + 2'b01;
        end
        else if(discard_from_IF && discard_from_IW) begin
            discard <= discard + 2'b10;
        end
    end
```

Listing 11: discard 计数器更新逻辑

1. 若当前还有需要丢弃的指令 ((|discard))，且收到 data_ok，则把 discard 减 1。

2. 若 IF 和 IW 中只有一个阶段报出需要丢弃指令 (discard_from_IF ^ discard_from_IW)，则把 discard 加 1。

3. 若 IF 和 IW 都报出需要丢弃指令 (discard_from_IF && discard_from_IW)，则把 discard 加 2。

IF 遇到冲刷时，如果之前的请求已经握手成功 (handshake_done)，且没有返回指令 (!inst_valid)，则要等这条指令返回后丢弃：

```
assign discard_out_wire = (ex_flush || ertn_flush || br_taken) && handshake_done && !
    inst_valid;
```

Listing 12: IF 阶段-冲刷丢弃指令-控制信号

IW 遇到冲刷时，如果冲刷前的 PC 对应的指令不是从 IF 传来的 (inst_valid_from_IF)，也不是之前暂存下来的 (inst_valid)，并且也没有在报出冲刷的这个周期收到当前 PC 的 data_ok(data_ok && (~(|discard))，其中 ~(|discard) 为 1 表示这个 data_ok 不是需要丢弃的，而是恰好对应当前 IW 的 PC)，那么这条指令也应该被丢弃：

```
wire discard_from_IW = (ex_flush || ertn_flush || br_flush) && in_valid && !(
    inst_valid_from_IF || (data_ok && (~(|discard))) || inst_valid);
```

Listing 13: IW 阶段-冲刷丢弃指令-控制信号

2.1.5 MEM 级和 RDW 级访存设计

与 IF 和 IW 类似，MEM 为了防止重复发取指请求，也引入了 handshake_done 信号，逻辑与 IF 类似，不再赘述。

req 信号与 IF 略有不同：

```
assign req = in_valid && !handshake_done && !this_flush && (res_from_mem || mem_we);
```

Listing 14: MEM 阶段的访存请求 IF 信号

在 MEM 级是有效的访存指令时，若请求没有握手成功 (!handshake_done)，并且 MEM 及以后的阶段不存在异常指令 (!this_flush)，则应当发出访存请求。与 IF 的区别在于：无论后面的阶段有没有异常/跳转指令，IF 都应该持续发取指请求，收到冲刷信号时利用 handshake_done_effective 信号处理“无效握手”；而 MEM 及以后的阶段如果存在异常指令，this_flush 会被拉高，抑制 MEM 级访存指令发挥作用，即不允许 req 拉高、不发请求，所以也就不存在握手成功后收到冲刷信号导致“无效握手”的情况，所以这里也就不必引入 handshake_done_effective 信号。

也正是因为上述原因，MEM 和 RDW 级不存在“丢弃”操作。

对于 load 得到的数据的接受与传递，与取指时类似，如果在 MEM 级收到数据且 MEM 的指令下一个上升沿无法流入 RDW，就在 MEM 暂存，待 RDW 允许时流入 RDW；RDW 也同样设置寄存器保存取回的数据——这方面的设计都与 IF/IW 类似，故不展示具体代码也不再赘述。

2.1.6 流水控制信号的修改

引入类 SRAM 接口并增加了两个流水级后，取指和 load/store 访存相关流水级的流水控制信号都有所修改 (主要是用于表明当前阶段任务完成与否、是否准备好流入下一级的 ready_go 信号)。

在 load/store 访存方面：

1. MEM 级要额外考虑在访存请求握手成功后 (当前周期握手成功 (req && addr_ok) 或是之前已经握手成功 (handshake_done))，指令才准备好流入 RDW：

```
assign ready_go = !in_valid  ||
                  this_flush ||
                  !(res_from_mul && !(to_mul_resp_ready && from_mul_resp_valid)) &&
                  !(res_from_div && !(to_div_resp_ready && from_div_resp_valid)) &&
                  !((res_from_mem || mem_we) && !(req && addr_ok || handshake_done));
```

Listing 15: MEM 级 ready_go 信号

2. RDW 级除了考虑无效和冲刷情况，主要应该考虑的是流水级内“没有还未收到 data_ok 的访存指令”时才算准备好流入 WB：

```
assign ready_go =!in_valid ||
                 this_flush||
                 !((res_from_mem || mem_we) && !(data_valid_from_MEM || data_ok ||
                     data_valid));
```

Listing 16: RDW 级 ready_go 信号

在取指方面，与 load/store 访存的不同之处在于前面提到的：就算后面流水级有异常/跳转指令，也不会暂停取指，所以与标记异常的 this_flush 等信号无关。

1. IF 级只有在当前 PC 取指请求握手成功 (当前周期或者是以前的某个周期) 时才算准备好流入 IW。特别地，在冲刷报出的那个周期，PC 已经修改为冲刷后的值，所以之前就算是有过握手成功且未流入 IW 的指令，那也是“无效握手”，所以这时 IF 是否准备好进 IW 就取决于 handshake_done_effective，而非 handshake_done：

```
assign ready_go = req && addr_ok || handshake_done_effective;
```

Listing 17: IF 级 ready_go 信号

2. IW 级除了无效时一定能流入下一级外，还要考虑：

   - 当前收到的指令不是需要丢弃的指令，确保指令与 PC 一致，才可流入 ID；
   - 当前周期有冲刷信号时，等待指令返回 (data_ok) 的操作已经交给前面提及的另设的 discard 相关逻辑进行处理，所以 IW 流水级相当于“无事可做”，可视为任务完成并准备好流动，故直接拉高 ready_go。

```
assign ready_go = !in_valid ||
                   ex_flush || ertn_flush ||
                   br_flush ||
                  (~(|discard)) && (inst_valid_from_IF || data_ok || inst_valid);
```

Listing 18: IW 级 ready_go 信号

2.2 类 SRAM-AXI 的转接桥设计

2.2.1 顶层模块设计

由于 CPU 对外只有一个 AXI 接口，所以转接桥顶层模块要对取指和 load/store 访存进行仲裁。

我们设立了独热码 ar/aw_id，根据优先级表示当前应该与哪个阶段的请求握手 (00 为无请求，01 为取指请求，10 为 load/store 请求)，并用于控制状态机的行为。根据教材的提示，load 访存请求的优先级大于取指请求，即两者同时存在时，优先与 load 握手：

```
assign ar_id = (~sram_wr_2 && sram_req_2) ? 2'b10 : (~sram_wr_1 && sram_req_1) ? 2'b01 : 2'
    b00;
assign aw_id = (sram_wr_2 && sram_req_2) ? 2'b10 : 2'b00;
```

Listing 19: CPU 读内存请求独热码

对于读请求而言，利用 ar_id 可以选择读请求的 size 和 addr 来自 IF 还是 MEM。

特别地，写请求只有 store 访存会发出，所以 aw_id 只有 00 和 10 两种情况，虽然不用区分对哪一流水级进行握手，但是为了对状态机进行驱动 (在 00 时保持空闲不动，10 时才开始工作)，仍保留类似 ar_id 的形式。

此外，为了处理“写后读”相关，引入 writing 信号，表明当前还有几个 store 写操作没有处理完：

```
    always @(posedge clk) begin
        if (!resetn) begin
            writing <= 5'b0;
        end
        else if (sram_wr_2 && aw_addr_ok && sram_req_2 && b_data_ok) begin
            writing <= writing;
        end
        else if (sram_wr_2 && aw_addr_ok && sram_req_2) begin
            writing <= writing + 1;
        end
        else if (b_data_ok) begin
            writing <= writing - 1;
        end
    end
```

Listing 20: writing 计数器

结合以上两方面，就可以对取指和 load/store 访存的请求进行握手 (addr_ok)：

```
    assign sram_addr_ok_1 = ar_id[0] ? ar_addr_ok : 1'b0;
    assign sram_addr_ok_2 = sram_wr_2 ? aw_addr_ok : (writing != 0) ? 1'b0 : ar_id[1] ?
        ar_addr_ok : 1'b0;
```

Listing 21: 顶层模块-CPU 读写内存请求-addr_ok(1 为取指，2 为 load/store)

1. aw_addr_ok 和 aw_addr_ok 由状态机发出，表示他们“能够”处理 CPU 发来的请求。
2. 但是否向 CPU 给出 addr_ok，还要依赖于这时有无请求 (ar_id 或 aw_id)，以及先处理哪一个请求 (ar_id，取指或 load)。
3. 此外，对于 load/store 访存请求，要在写请求全处理完后 (writing 为 0) 才能与读请求握手，防止写后读。

与请求类似，对于读数据响应，要根据 AXI 接口返回的 rid 信号生成独热码 r_id，选择向取指还是 load 访存进行响应 (data_ok)，对于写数据响应则不存在选择：

```
    assign sram_data_ok_1 = r_id[0] && r_data_ok;
    assign sram_data_ok_2 = r_id[1] && r_data_ok || b_data_ok;
```

Listing 22: 顶层模块-CPU 数据响应-data_ok(1 为取指，2 为 load/store)

2.2.2 读请求通道 AR

状态机的转移图如下：

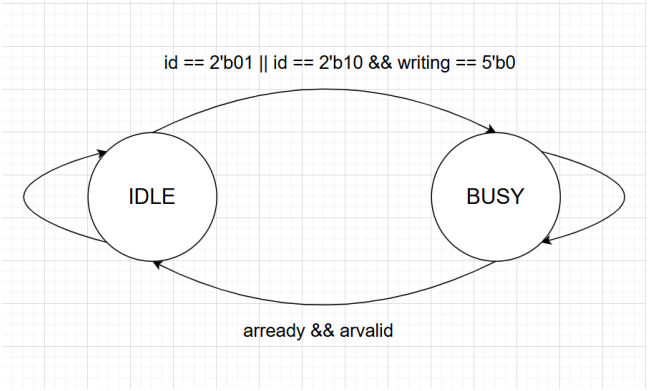

图 2: 读请求通道状态机

读请求通道的任务是，响应 CPU 用类 SRAM 接口发来的读请求，并通过 AXI 接口发送给内存。

1. 空闲状态 IDLE——响应 CPU 发来的读请求：

   在空闲时，拉高 addr_ok，告知顶层模块可以处理读请求 (进一步判定交给顶层，上文已经描述)：

```
        assign addr_ok = current_state == IDLE;
```

Listing 23: AR 状态机-addr_ok

由状态图，在收到要马上处理的请求时，IDLE 态变为 BUSY 态，开始处理读请求。

此外，由于后续通过 AXI 接口向内存发送读请求时，可能需要持续用到请求信息，但 CPU 相应流水级收到 addr_ok 后可能会处理其他指令，导致请求信息丢失，所以读请求通道要用寄存器保存它们：

```
        always @(posedge clk) begin
        if (!resetn) begin
            id_reg <= 2'b0;
            addr_reg <= 32'b0;
            size_reg <= 2'b0;
        end
        else if (current_state == IDLE) begin
            id_reg <= id;
            addr_reg <= addr;
            size_reg <= size;
        end
    end
```

Listing 24: AR 暂存请求信息

2. BUSY 状态——通过 AXI 接口把收到的读请求发给内存：

   这时只需拉高 arvalid 信号，并把保存下来的请求信息通过 AXI 接口发送：

```
        assign arid = {3'b0, id_reg[1]};
        assign araddr = addr_reg;
        assign arsize = {1'b0, size_reg};
        assign arvalid = current_state == BUSY;
```

Listing 25: AXI 接口读通道信号

由状态图，当用 AXI 接口发送的请求与内存握手成功后 (arready && arvalid)，本次任务完成，回到 IDLE，等待 CPU 发来的下一个读请求。

2.2.3 读响应通道 R

此通道的作用是接收内存通过 AXI 接口返回的读数据相关信号，然后通过类 SRAM 接口传递给 CPU。

由于类 SRAM 接口的读数据没有握手，直接返回数据和 data_ok 即可，所以面向 AXI 接口一侧的 rready 可以在解除复位后一直拉高：

```
1    assign rready = resetn;
```

Listing 26: rready 信号

要做的工作十分简单，除了接受数据，再把 rid 转换成顶层模块使用的独热码 (id)，并把握手成功的信号转化成 data_ok 即可：

```
1    always @(posedge clk) begin
2        if (!resetn) begin
3            id <= 2'b0;
4            data_ok <= 1'b0;
5            data <= 32'b0;
6        end
7        else begin
8            id <= {rid[0], ~rid[0]};
9            data_ok <= rvalid && rready;
10           data <= rdata;
11       end
12   end
```

Listing 27: 读响应通道工作

2.2.4 写请求 AW 与写数据 W 通道

写请求与写数据利用同一个状态机实现，状态转移图如下：

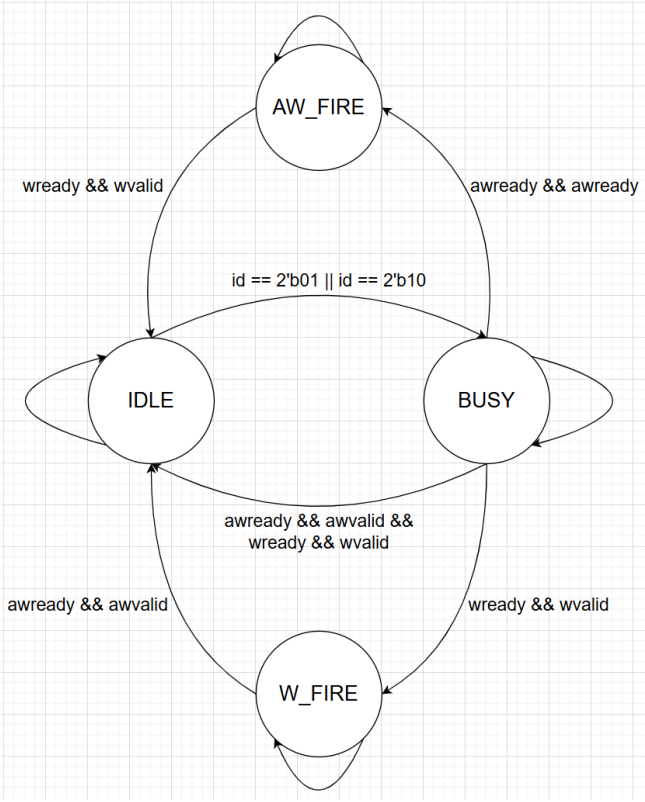

图 3: 写请求与写数据通道状态机

这一模块的任务是，响应 CPU 发的写请求和数据，并把它通过 AXI 接口发送给内存。

1. 初始化后，处于 IDLE。根据顶层模块发来的写请求独热码 (id)，检测到写请求后便启动，转到 BUSY 态。
2. 在 BUSY 态，同时拉高 awvalid 和 wvalid 信号，通过 AXI 接口发送写请求和写数据。
   (a) 若二者同时握手成功，则本次任务完成，回到 IDLE，等待下一个写请求和数据。
   (b) 若 aw 通道握手成功，则跳转到 AW_FIRE 态，不再拉高 awvalid(防止重复发请求)，但保持 wvalid 拉高，继续等待 w 通道握手成功，成功后回到 IDLE。
   (c) 类似地，若 w 通道握手成功，则跳转到 W_FIRE 态，不再拉高 wvalid(防止重复发请求)，但保持 awvalid 拉高，继续等待 aw 通道握手成功，成功后回到 IDLE。

结合上面的阐释，awvalid 和 wvalid 信号定义如下：

```
1  assign awvalid = current_state == BUSY || current_state == W_FIRE;
2  assign wvalid = current_state == BUSY || current_state == AW_FIRE;
```

Listing 28: awvalid 和 wvalid 信号

与读请求状态机类似，本状态机也要把发给 AXI 接口的请求信息保存于寄存器中。

2.2.5 写响应通道 B

与读响应通道类似，在解除复位后拉高 bready，一直能够接收内存从 AXI 接口发回的响应信号。data_ok 的设置类似读响应通道，不再赘述。

3 Debug 记录

3.1 转接桥写请求与写数据状态机逻辑错误

按照教材要求，我们把写请求和写数据通道写在一个状态机中，想当然地认为写请求和写数据会同时得到 ready 信号，于是只设计了 IDLE 和 BUSY 态，在 BUSY 态完成握手后回到 IDLE，但仿真时出现了下图错误：

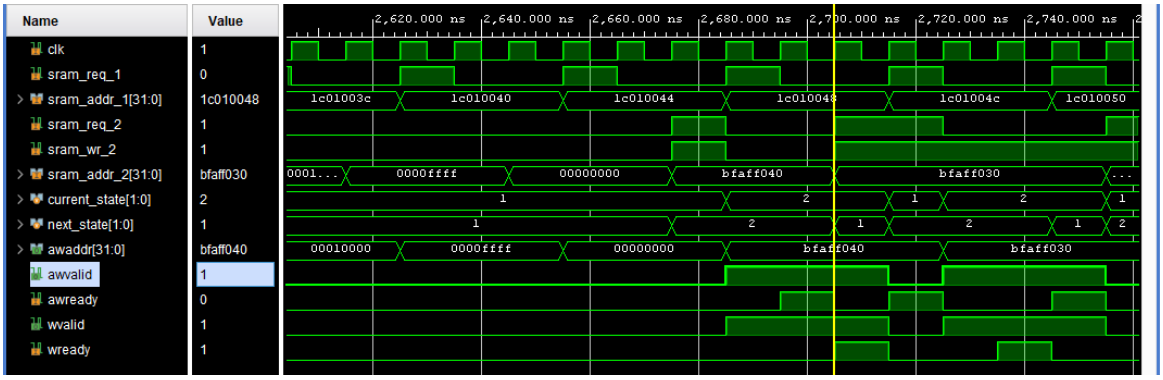

图 4: 转接桥写请求与写数据状态机逻辑错误-导致请求重复发送-波形

在光标位置附近的几个周期内，状态机处于 BUSY，持续拉高 awvalid 和 wvalid 信号，但 awready 在光标左侧位置就已经发来，wready 在光标右侧位置才发来。在这一版设计中，由于没有完成两个通道的握手，所以状态机一直处于 BUSY，一直在发请求。在光标右侧的那个周期依旧拉高的 awvalid 又被内存接收到一次，导致下一个周期又回来一次 awready——于是，同一条指令的内存写请求做了两次握手，写数据做了一次握手，最终导致后面继续运行时出现卡死。

所以我们后来增加了这一状态机的状态数，如果只完成了一个通道的握手，就转到 W_FIRE 或 AW_FIRE，拉低已经握手成功的 valid 信号，保留没成功的那个通道的 valid，等其握手成功后才能回到 IDLE。

3.2 writing 信号提前拉低导致写后读处理错误

最初我们在做设计时，只考虑了在收到 store 写请求后，拉高 writing，写响应发回后，拉低 writing 的情况，然后在收到 load 读请求时如果 writing 为 1 就不与 CPU 握手。然而，仿真时遇到如下问题：

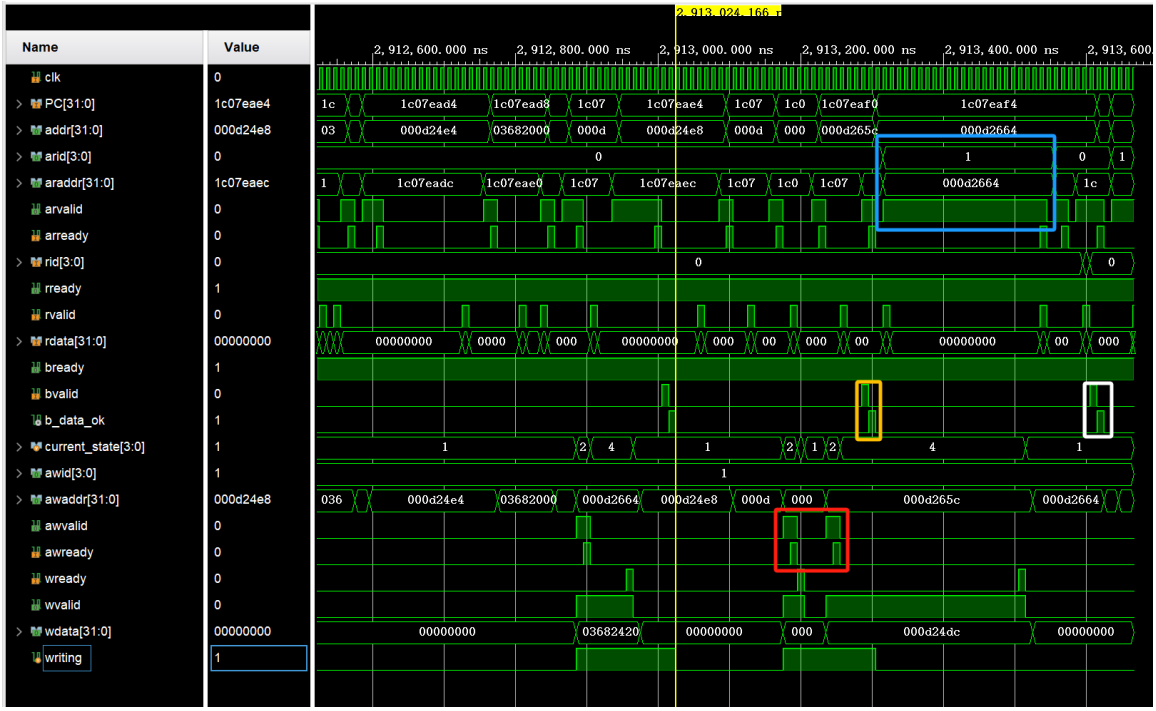

图 5: writing 信号提前拉低导致写后读处理错误-波形

如上图所示，图中有连续的三条访存指令——store-store-load，在反汇编文件中是下面这部分：

```
1c07eaec:       29860084        st.w    $r4,$r4,384(0x180)
1c07eaf0:       298600a5        st.w    $r5,$r5,384(0x180)
1c07eaf4:       2886018a        ld.w    $r10,$r12,384(0x180)
```

图 6: store-store-load 访存指令序列

在波形图的红框部分，两条 store 指令请求握手成功，在发第一条请求时 writing 被拉高，行为正确；黄框和白框部分分别收到两条指令的写响应。然而，第一次响应时 writing 就被拉低了，其无法正确反应当前还有一条握手成功写请求没有得到响应；进一步地，由于 writing 拉低，蓝框部分的 load 读请求向内存发了出去，导致写后读相关变成了先读后写，最终引发错误。

修改方法上文已经提及，把 writing 由标志位改为计数器，每多一个 store 写请求就加 1，收到响应就减 1，减到 0 时才能允许 load 读请求发给内存。

3.3 this_flush 信号设计错误

在上一个 project 中，由于访存没有延迟，所以出现异常的指令与前面流水级的指令之间不会有空泡，于是我们设计的 this_flush 信号如下 (这一信号用于标记当前流水级的指令是否在接下来的某一刻被冲刷，阻止这些会被冲刷的指令改变体系状态)：

```
1    this_flush = in_valid && (has_exception || next_flush || SYSCALL || BRK || INE || INT ||
         inst_ertn);
```

Listing 29: 上一个 Project 的 this_flush 信号 (以 ID 为例)

需要考虑当前级指令在流经前面流水级时有没有异常 (has_exception)，当前流水级有没有异常 (SYSCALL 等信号)，以及下一流水级的指令有没有异常 (next_flush)——由于没有空泡，这是合理的。

然而，在本 Project 中，访存具有延迟，就可能出现两个有效流水级之间存在无效流水级 (空泡)，无效流水级自然也不会有 this_flush。这时只用 next_flush 就会出错，比如：MEM 级存在异常指令，EX 级是空泡，ID 级的指令之后会被冲刷 (this_flush 应为 1)，但 MEM 的 this_flush 无法通过 next_flush 链条经 EX 传到 ID，导致 ID 的 this_flush 实际不为 1，遂出错。

修改后，我们把每一级 this_flush 逻辑中的 next_flush 改为了其后所有流水级的 this_flush 之“或”：

```
assign this_flush = in_valid && (has_exception || EX_flush || MEM_flush || RDW_flush ||
    WB_flush || SYSCALL || BRK || INE || INT || inst_ertn);
```

Listing 30: 本 Project 修改后的 this_flush 信号 (以 ID 为例)

3.4 冲刷时指令丢弃数目统计错误

我们在 CPU 设计时犯了一个与上面的 writing 信号类似的错误：

图 7: 指令丢弃数目错误-波形

如上图所示，红框部分出现 ertn 冲刷，IF 和 IW 都判断出自身流水级内部还有冲刷前请求成功但没收到的指令 (需要扔掉)，两个 discard_from 都拉高，即要丢掉两条指令，但 (蓝框所示的)discard 最初实现为标志位，导致第一个 data_ok 到来时 (黄框所示) 就被拉低。从而使得后面第二个 (要扔掉的)data_ok 到来时因 discard 为 0 而被误视为是当前 IW 阶段的 PC 对应的正确指令，从而出现指令和 PC 错位，最终引发错误。

修改方法类似 writing，改为计数器即可。

3.5 discard_from_IW 信号逻辑不完备

在最初的设计中，discard_from_IW 信号设计如下：

```
wire discard_from_IW = (ex_flush || ertn_flush || br_flush) && in_valid && !(
    inst_valid_from_IF || data_ok || inst_valid);
```

Listing 31: 最初的 discard_from_IW

然后出现了下图所示的错误波形：

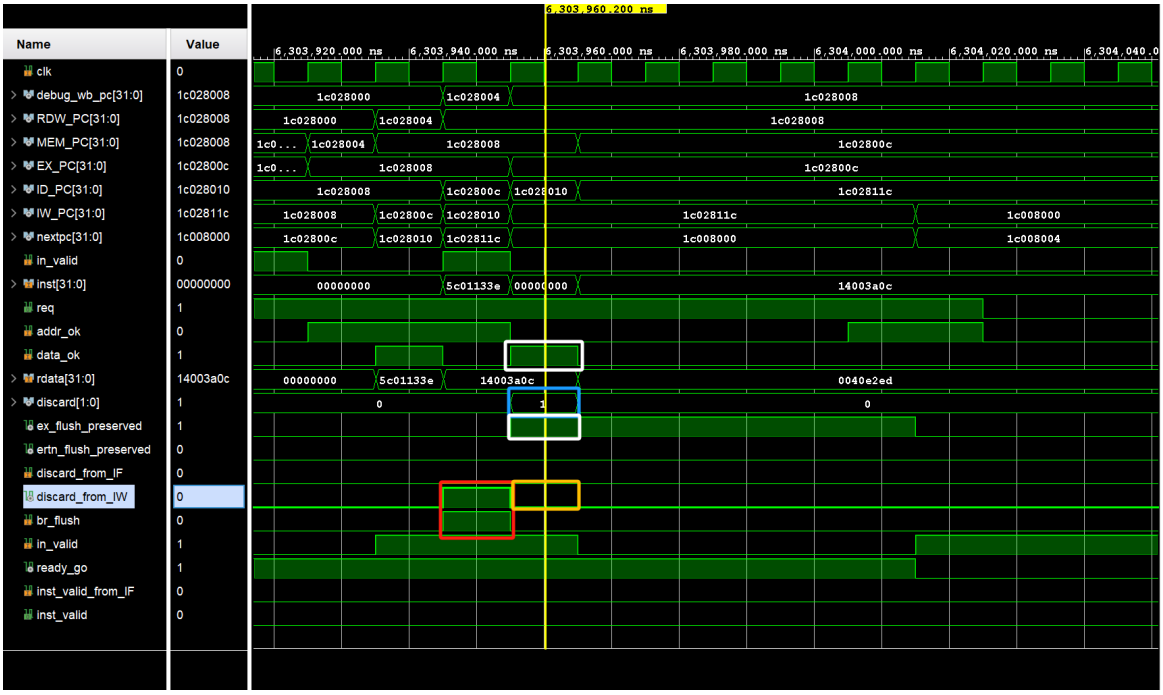

图 8: discard_from_IW 信号逻辑不完备导致的错误-波形

在红框所示位置，报出跳转冲刷 br_flush，由于 IW 当前 PC 还未收到 data_ok，故拉高 discard_from_IW，又由于 discard_from_IF 为 0，故上升沿到来时 discard 计数器从 0 更新为 1(如蓝框所示)。在上升沿到来后的这一周期，WB 报出 ertn 冲刷，同时收到一个 data_ok(白框所示)，这时我们的设计会误认为 data_ok 是当前 IW 的 PC 对应的指令 (但由于 discard 为 1，实则并不对应)，相当于判定为 IW 阶段没有没收到的 data_ok，故黄框位置处 discard_from_IW 为 0(应该为 1)，导致少丢弃一条指令，出现后续指令与 PC 对应不上的错误。

于是，我们把计数器 discard 纳入了 discard_from_IW 的逻辑以避免上述误判：

```
wire discard_from_IW = (ex_flush || ertn_flush || br_flush) && in_valid && !(
    inst_valid_from_IF || (data_ok && (~(|discard))) || inst_valid);
```

Listing 32: 修改后的 discard_from_IW

4 合作说明

本实验由本组成员共同合作完成，组内同学同等贡献。