中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 专业：计算机科学与技术

实验序号：5.2 实验名称：DMA 引擎与中断处理

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

1） 中断服务汇编程序 intr_handler.S 的实现

```
    #==== dma_buf_stat -= (new_tail_ptr - old_tail_ptr) / dma_size ====
    # k0: old_tail_ptr
    #get the value of new_tail_ptr
    lui   $k1, 0x6002
    lw    $k1, 0x8($k1) # k1: new_tail_ptr

    sub   $k0, $k1, $k0 # k0: new_tail_ptr - old_tail_ptr

L1:
    # do {
    lw    $k1, 0x10($0) # k1: dma_buf_stat
    addi  $k1, $k1, -1   # dma_buf_stat --;
    sw    $k1, 0x10($0)

    lui   $k1, 0x6002    # k1: base
    lw    $k1, 0x10($k1) # k1: dma_size
    sub   $k0, $k0, $k1  # k0 -= dma_size;}

    bgtz  $k0, L1        # while (k0 > 0);

    # ===== respond intr: ctrl_stat INTR = 0 =====
    lui   $k0, 0x6002       # k0: base
    lw    $k1, 0x14($k0)    # k1: ctrl_stat
    andi  $k1, $k1, 0x1
    sw    $k1, 0x14($k0)


    # ==== record tail_ptr ====
    lui   $k0, 0x6002
    lw    $k0, 0x8($k0)

    eret
```

中断服务程序可以大致分为 3 个部分：

I. 更新dma_buf_stat：首先从0x60020008取得dma的尾指针值，与之前保存的尾巴指针值相减，得到 dma 处理的空间大小；然后根据 dma 处理的空间的大小，更新 dma_buf_stat。

(dma_buf_stat -= (new_tail_ptr - old_tail_ptr) / dma_size)

II. 将 ctrl_stat 的最高位置为 0

III. 将取得到的尾指针保存在寄存器中，方便下一次的处理

2） custom_cpu 的变化

I. 状态机的变化

新增了一个 INTR 的状态，代表 CPU 需要进行中断处理，其转移逻辑如下所示：

```
IF: begin
        if (intr & !intr_en) begin //intr为1，且不处于中断处理状态，则转移到INTR状态
                next_state <= INTR;
        end
        else if (Inst_Req_Ready) begin
                next_state <= IW;
        end
        else begin
                next_state <= IF;
        end
```

```
end
EX: begin
        if (ERET) begin //ERET也会转移到IF(没有执行阶段)
                next_state <= IF;
        end
```

```
end
WB:      next_state <= IF;
INTR:    next_state <= IF;
default: next_state <= INIT;
```

在 IF 阶段时，如果 intr 为 1，且目前不处于中断处理状态，则转移到 INTR 状态，进行中断处理，否则正常取指。

而 INTR 的下一个状态便是 IF，读取中断服务程序，进行处理。

（同时新添对 ERET 指令的考虑）

II. 新增寄存器 EPC

```
//EPC(保存PC值)
always @(posedge clk) begin
        if (current_state == INTR) begin //进入中断处理，保存PC值
                EPC <= PC;
        end
end
```

其作用是保存进入中断处理前的 PC 值

III. 中断屏蔽信号

```
// INTR
always @(posedge clk) begin
        if (rst) begin
                intr_en <= 1'b0;
        end
        else if ((current_state == EX) & ERET) begin //完成ERET指令，则intr_en赋值0
                intr_en <= 1'b0;
        end
        else if (current_state == INTR) begin //进入中断处理，将intr_en赋值1
                intr_en <= 1'b1;
        end
end
```

其目的是保证在进入中断处理后，CPU 不再接受中断信号

IV. PC 逻辑的变化

```
always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == INTR) begin  // 进入中断处理，PC改为0x100(中断处理程序的入口)
                PC <= 32'h100;
        end
        else if (current_state == IW && Inst_Valid) begin
                PC <= ALUResult;
        end
        else if (current_state == EX) begin
                if(ERET) begin
                        PC <= EPC; //ERET PC更新到保存的值
                end
                else if (R_Type_jump) begin
                        PC <= RF_rdata1;
                end
                else if (J_Type) begin
                        PC <= {PC_normal[31:28], reg_Instruction[25:0], 2'b00};
                end
                else if ((Zero ^ (REGIMM & ~reg_Instruction[16] |
                        I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1)))) & Branch) begin //branch
                        PC <= Result; // ID阶段算出的PC
                end
        end
end
```

当状态机状态为 INTR 时，PC 值变为 0x100（中断服务程序的地址），当接受到 ERET 指令后，PC 更新为 EPC 的值（进入中断处理前的 PC 值）。

V. 握手信号的变化

```
//组合逻辑
  //Handshake Signal
  assign Inst_Req_Valid          = current_state == IF & ~(intr & ~intr_en); //保证intr拉高的那一个clk内Inst_Req_Valid没有效
  assign Inst_Ready              = current_state == IW || current_state == INIT;
  assign MemWrite                = current_state == ST;
  assign MemRead                 = current_state == LD;
  assign Read_data_Ready         = current_state == RDW || current_state == INIT;

  //Instruction Decode
```

Inst_Req_Valid 逻辑发生变化，目的是保证 intr 拉高的那一个 clk 内 Inst_Req_Valid 没有效，不会与访存冲突。（否则在 fpga 时候会与访存冲突而卡死）

3) Engine_core.v

I. 读引擎状态机

```
// --- read状态机 ---
// 第一段
always @(posedge clk) begin
        if (rst) begin
                rd_current_state <= IDLE;
        end
        else begin
                rd_current_state <= rd_next_state;
        end
end
```

```
// 第二段
always @(*) begin
        case (rd_current_state)
                IDLE: begin
                        if (en & wr_current_state[isIDLE] & (head_ptr != tail_ptr) & !rd_dma_finished & fifo_is_empty) begin
                                rd_next_state = REQ;  //tail_ptr != head_ptr时，DMA引擎自动启动
                        end
                        else begin
                                rd_next_state = IDLE;
                        end
                end
                REQ: begin
                        if (fifo_is_full) begin
                                rd_next_state = IDLE;  //fifo满了停止写入，读引擎停止工作，写引擎开始工作!!!!
                        end
                        else if (rd_req_ready & rd_req_valid) begin //握手成功
                                rd_next_state = RW;
                        end
                        else begin
                                rd_next_state = REQ;
                        end
                end
                RW: begin
                        if (rd_valid & rd_ready & rd_last) begin //握手成功+本轮burst的最后一次读完成
                                rd_next_state = REQ;
                        end
                        else begin
                                rd_next_state = RW;
                        end
                end
                default: rd_next_state = IDLE;
        endcase
end
```

定义了三个状态，分别代表闲置，请求和执行。

IDLE 代表读引擎没有工作，而其开始工作的条件（即转移到 REQ 状态）的条件为头尾指针不相等，且 fifo 为空，同时还要求写引擎完成一轮写内存的工作（即把 fifo 读空），处于闲置状态，且本轮 DMA 子缓冲区的搬运工作没有全部完成。

当开始一轮读内存工作后，读引擎会在 REQ 和 RW 状态之间切换。REQ 表示读引擎发送读内存请求，当握手成功后，进入 RW 状态，当 FIFO 写满后，完成本轮工作，进入 IDLE 状态。

RW 表示读引擎进行读内存工作，当握手完成，且本轮要读的数据全部完成，进入 REQ 状态。

II. 写引擎状态机

```
// --- write状态机 ---
// 第一段
always @(posedge clk) begin
        if (rst) begin
                wr_current_state <= IDLE;
        end
        else begin
                wr_current_state <= wr_next_state;
        end
end
```

```
always @(*) begin
        case (wr_current_state)
                IDLE: begin
                        if (en & rd_current_state[isIDLE] & !(fifo_is_full & wr_dma_finished) & !wr_dma_finished & fifo_is_full) begin
                                wr_next_state = REQ; //tail_ptr != head_ptr时，DMA引擎自动启动
                        end
                        else begin
                                wr_next_state = IDLE;
                        end
                end
                REQ: begin
                        if (wr_dma_finished | fifo_is_empty) begin //fifo读空后 or 一次DMA子缓冲区的全部读写完成
                                wr_next_state = IDLE;
                        end
                        else if (wr_req_ready & wr_req_valid) begin //握手完成
                                wr_next_state = FIFO;
                        end
                        else begin
                                wr_next_state = REQ;
                        end
                end
```

```
                RW: begin
                        if (wr_ready & wr_last | fifo_is_empty) begin //fifo读空后 or 握手完成且本次写完成
                                wr_next_state = REQ;
                        end
                        else if (wr_ready & !fifo_is_empty) begin //握手完成 但本次写未完成(fifo不空)
                                wr_next_state = FIFO;
                        end
                        else begin
                                wr_next_state = RW;
                        end
                end
                FIFO: begin
                        wr_next_state = RW; //FIFO阶段寄存器存进数据，下一个状态为RW，进行写操作(4个字节)
                end
                default: begin
                        wr_next_state = IDLE;
                end
        endcase
end
```

定义了四个状态，IDLE 和 REQ 分别代表闲置，请求，RW 和 FIFO 用来执行写操作。

IDLE 代表读引擎没有工作，而其开始工作的条件（即转移到 REQ 状态）的条件为头尾指针不相等，且 fifo 为满，同时还要求读引擎完成一轮读内存的工作（即把 fifo 写满），处于闲置状态，且本轮 DMA 子缓冲区的搬运工作没

有全部完成。

当开始一轮读内存工作后，读引擎会在 REQ 和 RW、FIFO 状态之间切换。REQ 表示写引擎发送写内存请求，当握手成功后，进入 RW 状态，当 fifo 读空后或 dma 子缓冲区搬运操作完成，完成本轮工作，进入 IDLE 状态。

FIFO 表示把 fifo 中的数据搬 4byte 搬到寄存器中，下一状态为 RW，进行这 4byte 数据的写。

RW 表示读引擎进行读内存工作，当握手完成且本轮 burst 要写的数据全部完成或者 fifo 读空后，进入 REQ 状态。

III. dma 寄存器的更新

```
always @(posedge clk) begin
        if (reg_wr_en[0]) begin
                src_base <= reg_wr_data;
        end
end
always @(posedge clk) begin
        if (reg_wr_en[1]) begin
                dest_base <= reg_wr_data;
        end
end
always @(posedge clk) begin
        if (reg_wr_en[2]) begin
                tail_ptr <= reg_wr_data;
        end
        else if (rd_dma_finished & wr_dma_finished & wr_current_state[isIDLE] & rd_current_state[isIDLE]) begin
                tail_ptr <= tail_ptr + dma_size; //一次DMA子缓冲区的读写全部完成，更新尾指针
        end
end
always @(posedge clk) begin
        if (reg_wr_en[3]) begin
                head_ptr <= reg_wr_data;
        end
end
always @(posedge clk) begin
        if (reg_wr_en[4]) begin
                dma_size <= reg_wr_data;
        end
end
always @(posedge clk) begin
        if (reg_wr_en[5]) begin
                ctrl_stat <= reg_wr_data;
        end
        else if (en & rd_dma_finished & wr_dma_finished & wr_current_state[isIDLE] & rd_current_state[isIDLE]) begin
                ctrl_stat[31] <= 1'b1;  //一次DMA子缓冲区的读写全部完成，更新ctrl_stat[31](中断标志位)
        end
end
```

当完成一次 dma 子缓冲区的数据的搬运，尾指针加 dma_size，ctrl_stat 的最高位置 1

IV. 判断本轮 DMA 子缓冲区的读写是否完成

```
//判断本轮DMA子缓冲区的读写是否完成
assign rd_dma_finished = (rd_burst_num == burst_total_num); //完成rd的burst的总数等于 需要完成的burst的总数
assign wr_dma_finished = (wr_burst_num == burst_total_num); //完成wr的burst的总数等于 需要完成的burst的总数
```

```
// read mem
always @(posedge clk) begin
        if (rst) begin  // rst复位
                rd_burst_num <= 0;
        end
        else if (rd_current_state[isIDLE] & wr_current_state[isIDLE] & en & (head_ptr != tail_ptr) & rd_dma_finished & wr_dma_finished) begin
                rd_burst_num <= 0;  //当一次DMA子缓冲区的读写全部完成后，开启下一轮DMA子缓冲区读写时，burst计数器清0
        end
        else if (rd_current_state[isRW] & rd_valid & rd_last) begin  // 一次burst的rd完成
                rd_burst_num <= rd_burst_num + 1;
        end
end
```

```
// write memory
always @(posedge clk) begin
        if (rst) begin  // rst复位
                wr_burst_num <= 0;
        end
        else if(wr_current_state[isIDLE] & rd_current_state[isIDLE] & en & ~intr & (head_ptr != tail_ptr) & rd_dma_finished & wr_dma_finished) begin
                wr_burst_num <= 0; //当一次DMA子缓冲区的读写全部完成后，开启下一轮DMA子缓冲区读写时，burst计数器清0
        end
        else if (wr_current_state[isRW] & wr_ready & wr_last) begin  // 一次burst的wr完成
                wr_burst_num <= wr_burst_num + 1;
        end
end
```

```
//DMA引擎一轮共需发起( int(N/32) + (N % 32 != 0) )次Burst传输
assign last_burst       = dma_size[4:0]; //(N % 32)
assign burst_normal_num = {5'b0, dma_size[31:5]}; //前(int(N / 32))次Burst
assign burst_total_num  = {5'b0, dma_size[31:5]} + |last_burst; //Burst总数

assign last_burst_len = last_burst[4:2] + |(last_burst[1:0]); //((N % 32) / 4) + ( ((N % 32) % 4) != 0)
```

首先，定义了两个计数器，用来判断已经完成的 burst 的工作数（分为读写两部分），然后判断其是否等于总共需要的 burst 总数即可。

DMA 引擎一轮共需发起(int(N/32) + (N % 32 != 0))次 Burst 传输，所以其硬件逻辑如上面第四张图所示。

两个计数器均是在一轮 dma 子缓冲区搬运工作全部完成后，下一轮开始

时，清 0，当完成一次 burst 后加一。

V. 读写内存的握手信号

```
// Handshake Signal
assign rd_req_valid = rd_current_state[isREQ] & !fifo_is_full & !rd_dma_finished;
assign rd_ready = rd_current_state[isRW];
```

```
assign wr_req_valid = wr_current_state[isREQ] & !fifo_is_empty;
assign wr_valid = wr_current_state[isRW];
```

当读引擎为 REQ 且 fifo 没满的时候，rd_req_valid 拉高；当读引擎为 RW（暗含 fifo 没满），rd_ready 拉高。

当写引擎为 REQ 且 fifo 不空的时候，wr_req_valid 拉高；当写引擎为 RW（暗含 fifo 不空），wr_valid 拉高。

VI. 读写引擎其它信号

```
//rd的addr & len
assign rd_req_addr = src_base + tail_ptr + {rd_burst_num, 5'b0}; //DMA读操作的
assign rd_req_len = rd_dma_finished ? {2'b0, (last_burst_len)} : 5'b111;//不
```

```
assign wr_req_addr = dest_base + tail_ptr + {wr_burst_num, 5'b0}; // DMA
assign wr_req_len = wr_dma_finished ? {2'b0, (last_burst_len)} : 5'b111;
assign wr_data = reg_fifo_rdata;
```

DMA 读操作的首地址为(src_base + tail_ptr),每次 burst 读入 32 位，所以逻辑表达式如图一所示。读数据的长度，如果不是最后一次长度为 32Byte（一次 fifo 的数据交换是 4Byte，所以为 8 次，用 111 表示），否则是((N%32)/4) + (((N%32)%4)!=0)次。

写地址和数据长度与读类似：DMA 写操作的首地址为(dest_base + tail_ptr),每次 burst 读入 32 位，写数据的长度，如果不是最后一次长度为 32Byte（一次 fifo 的数据交换是 4Byte，所以为 8 次，用 111 表示），否则是((N%32)/4) + (((N%32)%4)!=0)次。

VII. fifo 的相关信号

```
// write to fifo
assign fifo_wen = rd_ready & rd_valid & !fifo_is_full; //读内存握手完成
assign fifo_wdata = rd_rdata;


// read from fifo
assign fifo_rden = wr_next_state[isFIFO];
```

当读内存握手完成且 fifo 没满，fifo_wen 拉高。

写入的数据就是从内存读出的数据。

而当写引擎下一状态为 FIFO 状态时，表示能从 fifo 读出数据，fifo_rden 拉高

VIII. 一次 burst 的写结束信号：wr_last

```
//最后一次wr完成
assign wr_last = (wr_size == wr_req_len[2:0]); //本次burst写的次数等于需要的次数
```

```
//wr_size
always @(posedge clk) begin
        if (rst) begin  // rst复位
                wr_size <= 3'b0;
        end
        else if(wr_current_state[isREQ]) begin //write state为REQ时重新计数(下一轮burst的wr)
                wr_size <= 3'b0;
        end
        else if (wr_current_state[isRW] & wr_ready) begin  // 每轮传输写入4byte后，wr_size加一
                wr_size <= wr_size + 1;
        end
end
```

wr_last 是表示一次 burst 写结束的信号。首先我定义了一个计数器 wr_size，当下一轮 burst 开始时候，其初始化为 0，当完成一次写入 4Byte 后，其次数加一，而当其等于需要的 burst 数目的时候，wr_last 拉高。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，逻辑仿真和 FPGA 调试过程中的难点等）

本次实验的硬件部分代码 debug 特别麻烦，因为缺少特别有效的手段，只能对着汇编代码和仿真加速的波形图慢慢去 debug，这浪费了本人特别多的时间。对于本次实验中遇到的 bug，主要有以下几个：

遇到的重要 bug 以及解决过程：

1. 对读写引擎的工作逻辑理解出错：

本人的最开始以为。读引擎和写引擎可以同时工作，即只要 fifo 没满，就能从内存里读数据，写入 fifo；只要 fifo 非空，就能从 fifo 里读数据，写入内存，最初的状态机代码如下：

```
always @(*) begin
        case (rd_current_state)
                IDLE: begin
                        if (en & wr_current_state[isIDLE] & (head_ptr != tail_ptr) & ~rd_dma_finished) begin
                                rd_next_state = REQ;  //tail_ptr != head_ptr时，DMA引擎自动启动
                        end
                        else begin
                                rd_next_state = IDLE;
                        end
                end
                REQ: begin
                        if (rd_dma_finished) begin
                                rd_next_state = IDLE;  //一次DMA子缓冲区的全部读写完成
                        end
                        else if (rd_req_ready) begin
                                rd_next_state = RW;
                        end
                        else begin
                                rd_next_state = REQ;
                        end
                end
                RW: begin
                        if (rd_dma_finished) begin
                                rd_next_state = IDLE;  //一次DMA子缓冲区的全部读写完成
                        end
                        else if (rd_valid & rd_last & !fifo_is_full) begin //握手成功+本轮burst的最后一次读+fifo不满
                                rd_next_state = REQ;
                        end
                        else begin
                                rd_next_state = RW;
                        end
                end
                default: rd_next_state = IDLE;
        endcase
```

```
always @(*) begin
        case (wr_current_state)
                IDLE: begin
                        if (en & rd_current_state[isIDLE] & head_ptr != tail_ptr & ~wr_dma_finished) begin
                                wr_next_state = REQ; //tail_ptr != head_ptr时，DMA引擎自动启动
                        end
                        else begin
                                wr_next_state = IDLE;
                        end
                end
                REQ: begin
                        if (wr_dma_finished) begin
                                wr_next_state = IDLE; //一次DMA子缓冲区的全部读写完成
                        end
                        else if (wr_req_ready & !fifo_is_empty) begin //握手完成且FIFO非空
                                wr_next_state = FIFO;
                        end
                        else begin
                                wr_next_state = REQ;
                        end
                end
```

```
                end
                RW: begin
                        if (wr_dma_finished) begin
                                wr_next_state = IDLE; //一次DMA子缓冲区的全部读写完成
                        end
                        else if (wr_ready & wr_last) begin //握手完成且本次写完成
                                wr_next_state = REQ;
                        end
                        else if (wr_ready & !fifo_is_empty) begin //握手完成 但本次写未完成且fifo不空
                                wr_next_state = FIFO;
                        end
                        else begin
                                wr_next_state = RW;
                        end
                end
                FIFO: begin
                        wr_next_state = RW; //FIFO阶段寄存器存进数据，下一个状态为RW，进行写操作
                end
                default: begin
                        wr_next_state = IDLE;
                end
        endcase
end
```

但实际上，这种做法会导致 fpga 卡住，进入死循环，即不能让读引擎和写引擎并发工作。

解决过程：这个错误是理解问题，所以在经过很多次提交仍然不知道为什么会卡住后，我问了老师，才知道是自己这个地方的理解出了问题，然后才对状态机进行修改，修改后的代码在第一部分已经展示了，这里不再重复展示。

2. 对 burst 的计数器清零逻辑出错：

由于第一点的错误，所以本人最开始计数器清零逻辑如下所示：

```
// read mem
        always @(posedge clk) begin
                if (rst) begin  // rst复位
                        rd_burst_num <= 0;
                end
                else if (rd_current_state[isIDLE] & wr_current_state[isIDLE] & en & head_ptr != tail_ptr) begin
                        rd_burst_num <= 0;
                end
                else if (rd_current_state[isRW] & rd_valid & rd_last & !fifo_is_full) begin  // 一次burst的rd完成
                        rd_burst_num <= rd_burst_num + 1;
                end
        end


        // write memory
        always @(posedge clk) begin
                if (rst) begin  // rst复位
                        wr_burst_num <= 0;
                end
                else if(wr_current_state[isIDLE] & rd_current_state[isIDLE] & en & head_ptr != tail_ptr) begin /
                        wr_burst_num <= 0;
                end
                else if (wr_current_state[isRW] & wr_ready & wr_last) begin  // 一次burst的wr完成
                        wr_burst_num <= wr_burst_num + 1;
                end
        end
```

但实际上，由于一次读并不能把需要读的数据读完，所以这一计数器的值永远等不了需要的 burst 数目，从而导致死循环。

解决过程： 通过仿真加速，找到死循环的点，发现原因是一次 fifo 只能接受 20*32Byte 的数据，而一次 dma 子缓冲区的大小为 80*32Byte 的数据，找到出错点，并进行修改，最后代码如下：

```
// read mem
always @(posedge clk) begin
        if (rst) begin  // rst复位
                rd_burst_num <= 0;
        end
        else if (rd_current_state[isIDLE] & wr_current_state[isIDLE] & en & (head_ptr != tail_ptr) & rd_dma_finished & wr_dma_finished) begin
                rd_burst_num <= 0;  //当一次DMA子缓冲区的读写全部完成后，开启下一轮DMA子缓冲区读写时，burst计数器清0
        end
        else if (rd_current_state[isRW] & rd_valid & rd_last) begin  // 一次burst的rd完成
                rd_burst_num <= rd_burst_num + 1;
        end
end
```

```
// write memory
always @(posedge clk) begin
        if (rst) begin  // rst复位
                wr_burst_num <= 0;
        end
        else if(wr_current_state[isIDLE] & rd_current_state[isIDLE] & en & ~intr & (head_ptr != tail_ptr) & rd_dma_finished & wr_dma_finished) begin
                wr_burst_num <= 0; //当一次DMA子缓冲区的读写全部完成后，开启下一轮DMA子缓冲区读写时，burst计数器清0
        end
        else if (wr_current_state[isRW] & wr_ready & wr_last) begin  // 一次burst的wr完成
                wr_burst_num <= wr_burst_num + 1;
        end
end
```

3. 握手信号（Inst_Req_Valid）没有修改而出错

本人最开始对握手信号没有修改，如下所示：

```
//Handshake Signal
assign Inst_Req_Valid           = current_state == IF;
assign Inst_Ready               = current_state == IW || current_state == INIT;
assign MemWrite                 = current_state == ST;
assign MemRead                  = current_state == LD;
assign Read_data_Ready          = current_state == RDW || current_state == INIT;
```

然而，这样其实是存在问题的。这会导致在 intr 拉高的那一个周期内，同时进行对内存的访问，导致 fpga 的时候卡死。

解决过程：这个错误极其难发现，由于这个错误，导致仿真加速能过，但在 fpga 正式跑的时候却会卡死，在打算放弃的时候，张士寅同学告知我了这个可能会出错的地方（他也卡在这个地方很久），特别感谢张士寅同学的帮助，最后修改的代码如下：

```
//Handshake Signal
assign Inst_Req_Valid           = current_state == IF & ~(intr & ~intr_en); //保证ir
assign Inst_Ready               = current_state == IW || current_state == INIT;
assign MemWrite                 = current_state == ST;
assign MemRead                  = current_state == LD;
assign Read_data_Ready          = current_state == RDW || current_state == INIT;
```

三、 对讲义中思考题（如有）的理解和回答

性能对比如下：

I. 使用 dma 的性能计数器结果：

```
==============Hardware Performance Counter Result==============
        Cycles: at least 80233793
        Instruction Count: at least 1058987
        Memory Read Instruction Count: at least 2626
        Memory Write Instruction Count: at least 263340
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 74325095
        Memory Read Request Delay Cycles: at least 2630
        Read Data Delay Cycles: at least 81418
        Memory Write Request Delay Cycles: at least 263340
        Branch Instruction Count: at least 263528
        Jump Instruction Count: at least 12
============================================================
```

II. 用 Load 和 Store 指令代替 dma 的性能计数器的结果：

```
==============Hardware Performance Counter Result==============
        Cycles: at least 196138925
        Instruction Count: at least 2362355
        Memory Read Instruction Count: at least 262641
        Memory Write Instruction Count: at least 524638
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 163421500
        Memory Read Request Delay Cycles: at least 262645
        Read Data Delay Cycles: at least 19070848
        Memory Write Request Delay Cycles: at least 524638
        Branch Instruction Count: at least 524801
        Jump Instruction Count: at least 8
============================================================
```

可以看出，未使用 dma（用 Store 和 Load 指令来代替 dma 的作用）的周期数是使用 dma 的周期数的 2.44 倍，可见使用 dma 的效果还是很显著的。

四、 在课后，你花费了大约___40__小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

这次实验初看感觉还好，但实际实现过程中发现 fpga 总是卡死。最开始，本人以为只是因为自己的实现逻辑出现了问题，但后面经过与老师和同学的讨论，发现是自己对 PPT 的关于读写引擎的描述产生了误解。然后后面又发现仿真加速能过，但是 fpga 却过不了，最后发现是进入中断处理的那一个周期，将访存的握手信号拉高，导致了冲突，最终卡死，在修改了这一点后，才完成本次实验。

所以本人有一些建议，一个是可以在 PPT 中对读写引擎的逻辑做更加详细的描述，如必须等写引擎把 fifo 写满后，读引擎才能工作，等读引擎把 fifo 读空后，写引擎才能继续工作，不然就会卡死，这里真的卡了本人很久；二是对握手信号的变化能做一点提示，不然 debug 真的很难想到这一点（毕竟仿真加速能过，但是 fpga 过不了，真的很难搞）。

再就是，本次实验的硬件部分代码 debug 特别麻烦，因为缺少特别有效的手段，只能对着汇编代码和仿真加速的波形图慢慢去 debug，这浪费

了本人特别多的时间,希望下届的学弟学妹们能有更方便的 debug 手段吧（虽然客观来说，确实很难有）。

最后特别感谢老师，助教们和张士寅，李金明同学对本人提出的问题的帮助与解答。