中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034 姓名：侯汝垚 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

1. engine_core 设计

6 个 dma 内 cpu 可写的寄存器写法类似，以 src_base 为例，当对应的写使能信号有效时就将 reg_wr_data 写入。

```
// six registers controlled by CPU
reg [31:0] src_base_reg;
always @(posedge clk) begin
        if (rst)
                src_base_reg <= 32'b0;
        else if (reg_wr_en[0])
                src_base_reg <= reg_wr_data;
        else
                src_base_reg <= src_base_reg;
end
assign src_base = src_base_reg;
```

读写引擎采用类似的写法，都为三个状态，状态机也几乎相同。但是因为读取 fifo 数据的时候 valid 信号的下一个周期读取的数据才有效，这里会和读引擎有所区别。

```
localparam IDLE   = 3'b001;
localparam RD_REQ = 3'b010;
localparam RD     = 3'b100;
localparam WR_REQ = 3'b010;
localparam WR     = 3'b100;
```

```
// FSM 2
always @(*) begin
        case(RD_current_state)
                IDLE : begin
                        if (EN && head_ptr != tail_ptr && WR_current_state[0])
                                RD_next_state = RD_REQ;
                        else
                                RD_next_state = IDLE;
                end
                RD_REQ : begin
                        // if there is only rd_req_ready, memory will always waiting for valid signal
                        if (rd_req_ready && rd_req_valid)
                                RD_next_state = RD;
                        else if (rd_complete)
                                RD_next_state = IDLE;
                        else
                                RD_next_state = RD_REQ;
                end
                RD: begin
                        if (rd_valid && rd_last && !fifo_is_full)
                                RD_next_state = RD_REQ;
                        else
                                RD_next_state = RD;
                end
                default :
                        RD_next_state = IDLE;
        endcase
end
```

```
always @(*) begin
        case(WR_current_state)
                IDLE : begin
                        if (EN && RD_current_state[0] && head_ptr != tail_ptr)
                                WR_next_state = WR_REQ;
                        else
                                WR_next_state = IDLE;
                end
                WR_REQ : begin
                        // if there is only wr_req_ready, memory will always waiting for valid signal
                        if (wr_req_ready && wr_req_valid)
                                WR_next_state = WR;
                        else if (wr_complete)
                                WR_next_state = IDLE;
                        else
                                WR_next_state = WR_REQ;
                end
                WR: begin
                        if (wr_last)
                                WR_next_state = WR_REQ;
                        else
                                WR_next_state = WR;
                end
                default :
                        WR_next_state = IDLE;
        endcase
end
```

用两个计数器分别计数读引擎和写引擎 burst 传输的次数。

```
// record the burst num
reg [27:0] rd_counter;

// to judge the last burst
wire [27:0] rd_counter_plus = rd_counter + 1;
always @(posedge clk) begin
        if (start || rst)
                rd_counter <= 0;
        else if (rd_valid && rd_last && !fifo_is_full)
                rd_counter <= rd_counter_plus;
        else
                rd_counter <= rd_counter;
end

reg [27:0] wr_counter;
wire [27:0] wr_counter_plus = wr_counter + 1;
always @(posedge clk) begin
        if (start || rst)
                wr_counter <= 0;
        else if (wr_last)
                wr_counter <= wr_counter_plus;
        else
                wr_counter <= wr_counter;
end
```

计算需要的 burst 传输的次数以及最后一次传输的长度。其中加和减用来处理非对齐的情况。

```
// calculate the tatal burst times and the last burst length
wire [27:0] burst_num = dma_size[31:5] + (|dma_size[4:0]);
wire [ 2:0] burst_last_len = dma_size[4:2] - {2'b0, !(|dma_size[2:0])};

wire rd_complete = (rd_counter == burst_num) && (rd_counter != 0);
wire wr_complete = (wr_counter == burst_num) && (wr_counter != 0);
```

写引擎相关控制信号比较直接，读地址直接根据基址、尾指针以及计数器相加得出，读请求信号只有当 fifo 为空时才发出，保证缓冲区有够一次突发传输的存储空间。写使能信号在握手成功时拉高。

```
assign rd_req_addr = src_base + tail_ptr + {rd_counter, 5'b0};
assign rd_req_valid = RD_current_state[1] && fifo_is_empty && !rd_complete;
assign rd_req_len = (rd_counter_plus == burst_num) ? {2'b0, burst_last_len} : 5'b00111;
assign rd_ready = RD_current_state[2] && !fifo_is_full;
assign fifo_wdata = rd_rdata;
assign fifo_wen = RD_current_state[2] && rd_valid && rd_ready;
```

读引擎需要稍微特殊处理。

信号名	类型	数据位宽	说明
fifo_rden	Output	1-bit	读使能 （下一时钟上升沿读数据有效）

因为读使能有效后的下一周期读数据才有效，然而这时内存不一定 ready，因此需要用一个寄存器来存储 fifo 中读出的数据，同时记录上一周期的 fifo_rden，写回数据有效当且仅当 last_fifo_rden 为 1 并且 fifo 非空，每次握手成功或者 fifo 为空时无效，用一个寄存器表示写回内存数据是否有效。根据这个信号来连接 dma 与内存写回部分的连接。

```
assign wr_req_addr = dest_base + tail_ptr + {wr_counter, 5'b0};
assign wr_req_len = (wr_counter_plus == burst_num) ? {2'b0, burst_last_len} : 5'b00111;
assign wr_req_valid = WR_current_state[1] && !fifo_is_empty && !wr_complete;
assign wr_data = (last_fifo_rden) ? fifo_rdata : wr_data_reg;
assign wr_valid = wr_valid_reg && WR_current_state[2];
assign wr_last = WR_current_state[2] && (wr_last_counter == 0) && wr_valid;

// get data from fifo only if the data in reg now is invalid or it is valid and will be written
assign fifo_rden = wr_burst_start || WR_current_state[2] && ( ~wr_valid || wr_valid && wr_ready && !wr_last);
```

```
// get data from fifo only if the data in reg now is invalid or it is valid and will be written
assign fifo_rden = wr_burst_start || WR_current_state[2] && ( ~wr_valid || wr_valid && wr_ready && !wr_last);

reg wr_valid_reg;
always @(posedge clk) begin
        if (rst)
                wr_valid_reg <= 0;
        // 上一周期拉高fifo读并且非空，读数据才有效
        else if (fifo_rden && !fifo_is_empty)
                wr_valid_reg <= 1;
        // 上一周期数据被写走并且fifo读为低以及fifo空的时候无效
        else if (wr_valid && wr_ready && !fifo_rden || fifo_rden && fifo_is_empty)
                wr_valid_reg <= 0;
        else
                wr_valid_reg <= wr_valid_reg;
end

// record last_fifo_rden to judge whether fifo_rdata is valid
reg last_fifo_rden;
always @(posedge clk) begin
        last_fifo_rden <= fifo_rden;
end
```

2. CPU 中断功能

在原先的多周期 CPU 基础上增加一个 INTR 状态并修改当前状态为 IF 时的状态转移关系。其中 no_intr 信号用于响应中断后的屏蔽中断。要注意中断和发送指令读请求的优先级，否则可能无法正常进入中断。

```
reg [9:0] current_state;
reg [9:0] next_state;
localparam INIT = 10'b0000000001,
           IF   = 10'b0000000010,
           IW   = 10'b0000000100,
           ID   = 10'b0000001000,
           EX   = 10'b0000010000,
           ST   = 10'b0000100000,
           LD   = 10'b0001000000,
           RDW  = 10'b0010000000,
           WB   = 10'b0100000000,
           INTR = 10'b1000000000;
```

```
        IF    : begin
                if (intr && !no_intr)
                        next_state = INTR;
                else if (Inst_Req_Ready)
                        next_state = IW;
                else
                        next_state = IF;
        end
```

增加 EPC 寄存器，触发中断后保存 PC。同时修改 PC 寄存器，当指令为 ERET 时将返回地址写回 PC。

```
// EPC
reg [31:0] EPC;
always @(posedge clk) begin
        if (current_state[9])
                EPC <= PC;
        else
                EPC <= EPC;
end
```

```
assign PC = PC_reg;
assign PC_next = {32{~ERET && PCsource == 2'b01}} & ALUOut
               | {32{~ERET && PCsource == 2'b00}} & ALU_result
               | {32{~ERET && PCsource == 2'b10}} & PC_jmp
               | {32{ ERET                     }} & EPC;
```

3. 中断服务程序

```
intr_handler:
        # TODO: Please add your own interrupt handler for DMA engine

        # set INTR to 0
        lui     $k1, 0x7fff
        ori     $k1, $k1, 0xffff
        lui     $k0, 0x6002
        lw      $k0, 0x14($k0)
        and     $k1, $k1, $k0
        lui     $k0, 0x6002
        sw      $k1, 0x14($k0)

        # store new tail_ptr, calculate new_tail_ptr - last_tail_ptr in $k0
        lw      $k1, 0x08($k0)
        lw      $k0, last_tail_ptr
        sub     $k0, $k1, $k0
        sw      $k1, last_tail_ptr
        blez    $k0, L2

        # (dma_buf_stat - $k0) / dma_buf_size
L1:
        lw      $k1, 0x10($0)
        addi    $k1, $k1, -1
        sw      $k1, 0x10($0)
        lui     $k1, 0x6002
        lw      $k1, 0x10($k1)
        sub     $k0, $k0, $k1
        bgtz    $k0, L1
        NOP

L2:
        eret
```

中断服务程序为 mips 汇编，主要分为三段，第一段将 ctrl_stat 寄存器中的 INTR 位置为 0，第二段读取旧的 tail_ptr 的值，求出新旧 tail_ptr 的差后存储当前 tail_ptr，第三段根据上述的差值和 dma_size 的大小循环递减 dma_buf_stat。

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）

1. fpga_emu 可以跑完，但是 fpga_run 却卡死。

因为本实验没有行为仿真，只能通过仿真加速功能进行调试，然而因为仿真加速不能完全模拟真实的 fpga 环境，出现了仿真加速可以正确跑完并向 0xC 写 0，而 fpga_run 却卡死。开始的时候怀疑可能是实验环境问题，但是因为无法捕获实际 fpga 上跑的信号，所以一直找不到问题所在，某天早上突发奇想 ctrl_stat 寄存器中间三十位保留位接入一些信号即可通过 cpu 利用 data_mover.c 程序打印一些 dma 的内部信号。将多个信号一一排查后发现问题较多。

首先是状态一直保持在 IDLE 状态。必须把状态转移条件改为 valid&&ready 才能正确转移，微信群内多位同学出现了相同的问题。其次是没有正确进入中断服务程序，经多次排查发现依然在 dma 内部没有发现问题。研究很久才发现问题其实出在 cpu，cpu 中中断的 IF 周期内依然拉高了指令请求，但是没有相应的 ready 信号接收指令，因此一旦进入中断，cpu 就会卡死。

2. 读 fifo 的使能信号和数据并非同时有效，如何处理?

最开始没有认真阅读 ppt，以为拉高 fifo_rden 后读数据同周期有效，因此导致读数据时握手信号卡死，为了解决这个问题需要记录上一周期的 fifo_rden，同时也需要一个寄存器存读出来的数据，然后根据 last_fifo_rden 定义 wr_valid 信号。

3. 虽然通过平台测试，但是 dma 没有正确搬运数据

在缺乏调试手段下经过多天排查终于通过 fpga_run，但是我发现 data_mover.c 中只对 dma_buf_stat 进行检查而不检查数据搬运是否正确，因此我加了一个循环来检测数据是否搬运成功，即 src 和 dest 数组是否全部相等。结果第一个数就不对，并且搬运的数据非常眼熟，仔细一样全是指令，也就是说搬运的 src_base 是从 0 开始，采用将 src_base 接入 ctrl_stat 中间 30 位的方法调试后发现问题果然在于 src_base 没有正确写入,但是 fpga_emu 的波形中 src_base 是真实写入的，其他寄存器的写入都没有问题。该问题目前还不知道原因何在。

三、 在课后，你花费了大约 20 小时完成此次实验。

四、 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）

本次实验可能因为课堂讲解可能与理论课进度不一，导致大伙对 dma 有些不太理解，根据我和一些同学的交流，这次实验虽然难度不算大，代码量不多，但是难在理解。ppt 中存在一定的误导，让我在一开始的时候混淆了 dma 中缓冲区的队列和搬运时内存中头指针尾指针对应的这个队列。根据群里面同学的交流，多位同学都出现了仿真加速能过但是 fpga_run 卡死的情况，大多都是访存信号的问题，希望之后实验框架的仿真的访存模型可以更为真实。有本次实验的经历，我建议可以在一些外部器件上加一些与 cpu 互连的自定义寄存器，这样可以结合软件打印方便查看内部信号，这样可以获取 fpga_run 时某些关键信号的情况，解决仿真与真实运行不一致的问题。