中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009926006 姓名：游昆霖 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 格式保存在~/COD-Lab/reports 目录下。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。

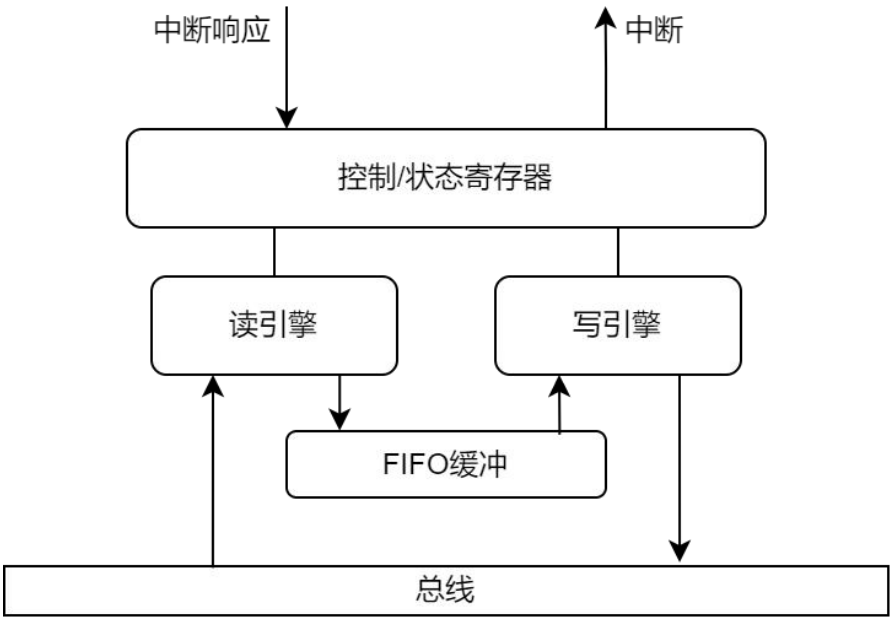
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。

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、相应信号的仿真波形和信号变化的说明等）

1、engine_core.v

逻辑结构图：



（1）控制/状态等寄存器处理

```
//Deal with input from CPU
        always @(posedge clk) begin
                if(rst) begin
                        src_base  <= 32'b0;
                        dest_base <= 32'b0;
                        tail_ptr  <= 32'b0;
                        head_ptr  <= 32'b0;
                        dma_size  <= 32'b0;
                        ctrl_stat <= 32'b0;
                end
                else if(EN & WR_cur_state[1] & RD_burst_cnt == burst_time & WR_burst_cnt ==
burst_time)
                begin
                        tail_ptr        <= tail_ptr +dma_size;
                        ctrl_stat[31]   <= 1'b1;
                end
                else begin
                        if(reg_wr_en[0])        src_base  <= reg_wr_data;
                        else if(reg_wr_en[1])   dest_base <= reg_wr_data;
                        else if(reg_wr_en[2])   tail_ptr  <= reg_wr_data;
                        else if(reg_wr_en[3])   head_ptr  <= reg_wr_data;
                        else if(reg_wr_en[4])   dma_size  <= reg_wr_data;
                        else if(reg_wr_en[5])   ctrl_stat <= reg_wr_data;
                end
        end

//Singals of DMA enable and intr valid
        assign intr = ctrl_stat[31];
        assign EN   = ctrl_stat[0];
```

初始复位后，根据外部所给信号将寄存器进行更新，从而获得对应使能信号及传输的位置、大小等相关信号。当 DMA 引擎处理完一个子缓冲区后，更新尾指针并拉高 intr 对应信号，以进行中断服务响应。

（2）读写引擎的状态转移

```
//State Machine of READ
        always @(posedge clk) begin
                if(rst)
                        RD_cur_state <= RD_idle;
                else
```

```
                        RD_cur_state <= RD_next_state;
        end

        always @(*) begin
                case(RD_cur_state)
                        RD_idle: begin
                                if(EN & WR_cur_state[0] & head_ptr!=tail_ptr)
                                        RD_next_state = RD_req;
                                else
                                        RD_next_state = RD_idle;
                        end
                        RD_req: begin
                                if(rd_req_ready&rd_req_valid)           //consider RD burst cnt !=
burst time
                                        RD_next_state = RD_work;
                                else if(RD_burst_cnt == burst_time)     //after last_burst
                                        RD_next_state = RD_idle;
                                else
                                        RD_next_state = RD_req;
                        end
                        RD_work: begin
                                //if(rd_valid & rd_last & ~fifo_is_full)
                                if(rd_ready & rd_valid & rd_last)
                                        RD_next_state = RD_req;
                                else
                                        RD_next_state = RD_work;
                        end
                        default:
                                RD_next_state = RD_idle;
                endcase
        end

//State machine of WRITE
        always @(posedge clk) begin
                if(rst)
                        WR_cur_state <= WR_idle;
                else
                        WR_cur_state <= WR_next_state;
        end

        always @(*) begin
                case(WR_cur_state)
                        WR_idle: begin
                                if(EN & head_ptr!=tail_ptr & !fifo_is_empty)
```

```
                                        WR_next_state = WR_req;
                                else
                                        WR_next_state = WR_idle;
                        end
                        WR_req: begin
                                if(wr_req_ready & wr_req_valid)
                                        WR_next_state = WR_work;
                                else if(WR_burst_cnt == burst_time)
                                        WR_next_state = WR_idle;
                                else
                                        WR_next_state = WR_req;
                        end
                        WR_work: begin
                                if(wr_ready & wr_valid & wr_last)
                                        WR_next_state = WR_req;
                                else
                                        WR_next_state = WR_work;
                        end
                        default:
                                WR_next_state = WR_idle;
                endcase
        end
```

读引擎在首尾指针不等，即子缓冲区非空时启动。写引擎则在此基础上还要求 fifo 非空，即读引擎已经向 fifo 填入了数据。读写引擎在请求信号握手后从请求状态进入工作状态，当数据握手且相应 last 信号拉高时再次进入请求态。当读写 burst 传输次数达到预先计算值时，进入相应 idle 态，表示本次子缓冲区传输的相应读写引擎工作完成。

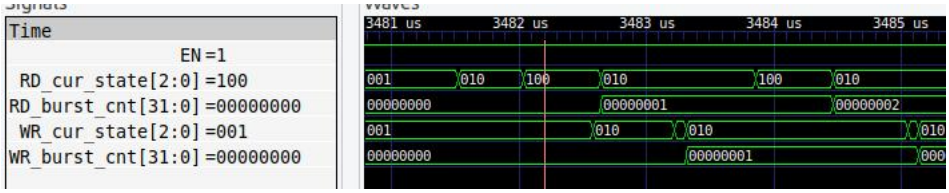
（3）burst 传输次数的计算和统计

```
//burst time and count
        assign burst_time_A[26:0]  = dma_size[31:5];
        assign burst_time_A[31:27] = 5'b0;
        assign burst_time_B[0]     = |dma_size[4:0];
        assign burst_time_B[31:1] = 31'b0;
        assign burst_time = burst_time_A + burst_time_B; //传输次数，burst_cnt 为 0 至 burst_time-1，
相等表示传输完毕
        assign last_burst_cnt = burst_time -1 ;


        always @(posedge clk) begin
```

```
            if(rst)
                    RD_burst_cnt <= 32'b0;
            else if(RD_cur_state[0] & WR_cur_state[0])
                    RD_burst_cnt <= 32'b0;
            else begin
                    if(RD_cur_state[2] & rd_ready & rd_valid & rd_last)     //the end of a burst
                            RD_burst_cnt <= RD_burst_cnt +32'b1;
                    else
                            RD_burst_cnt <= RD_burst_cnt;
            end
    end

    always @(posedge clk) begin
            if(rst)
                    WR_burst_cnt <= 32'b0;
            else if(RD_cur_state[0] & WR_cur_state[0])      //zero out after a transfer
                    WR_burst_cnt <= 32'b0;
            else begin
                    if(WR_cur_state[2] & wr_ready & wr_valid & wr_last)     //the end of a burst
                            WR_burst_cnt <= WR_burst_cnt +32'b1;
                    else
                            WR_burst_cnt <= WR_burst_cnt;
            end
    end
```

首先根据 dma_size 的大小计算出传输次数。当复位或一次子缓冲区处理完之后，将读写传输次数统计清零。当读写引擎每次 burst 传输结束后将统计值加 1，从而使请求状态时 burst_cnt 为 0 至 burst_time-1，当其到达 burst_time 时表示最后一次传输结束。便于请求时长度和地址等信号确定。

硬件仿真加速波形说明：



如上图所示，XX_burst_cnt 在一次请求和工作时保持稳定，表示该次 burst 传输请求和传输的序号（从 0 开始）。

（4）请求长度及地址信号的确定

```
//len of burst, transmission time of 4 byte data
        assign last_burst_len_A[2:0] = dma_size[4:2];
        assign last_burst_len_A[4:3] = 2'b0;
        assign last_burst_len_B[0]   = |dma_size[1:0];
        assign last_burst_len_B[4:1] = 4'b0;
        assign last_burst_len = last_burst_len_A + last_burst_len_B -1 ; //len is 0-7

        //when req, burst_cnt is 0 - burst_time-1
        assign rd_req_len = (burst_time_B[0] & RD_burst_cnt == last_burst_cnt) ? last_burst_len :
5'd7;
        assign wr_req_len = (burst_time_B[0] & WR_burst_cnt == last_burst_cnt) ? last_burst_len :
5'd7;

//addr of READ and WRITE
        always @(posedge clk) begin
                if(RD_cur_state[0] & WR_cur_state[0] & head_ptr != tail_ptr)
                        rd_req_addr <= src_base + tail_ptr;
                else if(RD_cur_state[2] & rd_ready & rd_valid & rd_last) begin
                        if(RD_burst_cnt == last_burst_cnt)
                                rd_req_addr <= rd_req_addr + dma_size[4:0];
                        else
                                rd_req_addr <= rd_req_addr + 32'd32;
                end
        end

        always @(posedge clk) begin
                if(RD_cur_state[0] & WR_cur_state[0] & head_ptr != tail_ptr)
                        wr_req_addr <= dest_base + tail_ptr;
                else if(WR_cur_state[2] & wr_ready & wr_valid & wr_last) begin
                        if(WR_burst_cnt == last_burst_cnt)
                                wr_req_addr <= wr_req_addr + dma_size[4:0];
                        else
                                wr_req_addr <= wr_req_addr + 32'd32;
                end
        end
```

首先根据 dma_size 计算最后一次传输长度，并根据 burst 统计值确定请求长度。在读写引擎开始工作前一拍根据基地址和指针确定读写请求地址的初值，并在每次 burst 传输后根据本次传输的大小更新请求地址。

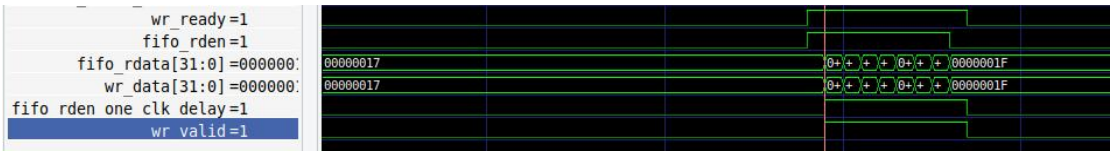
（5）控制信号

```
//handshake signal of read and write
```

```
        assign rd_req_valid = RD_cur_state[1] & ~fifo_is_full & RD_burst_cnt!=burst_time;
        assign wr_req_valid = WR_cur_state[1] & ~fifo_is_empty & WR_burst_cnt != burst_time;

        assign rd_ready = RD_cur_state[2] & ~fifo_is_full;
        assign wr_valid = fifo_rden_one_clk_delay;

        assign fifo_wen = rd_ready & rd_valid;
        assign fifo_wdata = rd_rdata;

        assign fifo_rden = WR_cur_state[2] & ~fifo_is_empty & wr_ready;

        assign wr_data = fifo_rdata;
        //考虑到 wr_ready 中间突然拉低几个周期，fifo_rden 有效时读出来的数据应当进行保存，
        //且应当在之后 wr_ready 拉高的第一个周期输出，因此此时 valid 应当为高，不可直接用 fifo_rden 延后一周期实现
        always @ (posedge clk) begin
                if(~wr_ready)
                        fifo_rden_one_clk_delay <= fifo_rden_one_clk_delay;
                else
                        fifo_rden_one_clk_delay <= fifo_rden;
        end
```

读写请求有效信号应当同时考虑 fifo 状态和 burst 统计值，最后一次请求状态已经完成所有传输，不必再拉高请求有效信号。由于 fifo 文件实现的是同步读同步写，因此写有效信号应当延后 fifo 读使能信号一拍拉高，同时考虑到 wr_ready 信号中间突然拉低几个周期的情形，应当在此情况下将 wr_valid 信号进行锁存。

硬件仿真波形说明：



如上图所示，在 fifo_rden 拉高的下一拍 fifo 才给出数据，因此应当延后一拍拉高写有效信号。

（6）wr_last 信号的产生

```
//last shifter for wr
```

```
        always @(posedge clk) begin
                if(wr_req_ready & wr_req_valid)
                        last_shifter <= (9'b1<<(wr_req_len+1));
                else if(fifo_rden)
                        last_shifter <= {1'b0,last_shifter[8:1]};
                else
                        last_shifter <= last_shifter;
        end

        assign wr_last = last_shifter[0];
```

在请求信号握手时根据请求长度放置一个移位器，wr_req_len+1 表示本次传输的数据数，由于 fifo_rden 信号恰在写数据前拉高，在每次 fifo 读使能信号拉高时将其右移，可以保证最后一个数据输出时，1 移至 last_shifter 的最低位。

2. Custom_cpu.v

该 CPU 基于 prj3 实现的 mips 型处理器修改，添加了中断和中断屏蔽的相关处理。以下只列出修改部分。

（1）状态机相关

```
assign Inst_Req_Valid  = current_state[1] & ~(~intr_mask & intr) ;     //IF

...

always @ (*)
begin
       case(current_state)
              ...
              IF: begin
                   if(~intr_mask & intr)
                         next_state = INTR;
                   else if ...
              end
              INTR: begin
                         next_state = IF;
              end
              ...
               EX: begin
                    if(... | inst_ERET )
```

```
                        next_state = IF;
                    else if...
                end


        ...
        endcase
end

...

assign inst_ERET = opcode[5:0] == 6'b010000 & func[5:0] == 6'b011000;
```

添加非中断屏蔽状态 IF 到 INTR 的状态转移，和 eret 指令时从 EX 到 IF 的状态转移。注意，当非屏蔽时 IF 接收到 intr 中断信号，不必拉高 Inst_Req_Valid 信号。

（2）中断屏蔽

```
always @(posedge clk) begin
        if(rst)
                intr_mask <= 1'b0;
        else if(current_state == INTR)
                intr_mask <= 1'b1;
        else if(current_state == EX & inst_ERET)
                intr_mask <= 1'b0;
        else
                intr_mask <= intr_mask;
end
```

进入 INTR 状态后将屏蔽信号拉高，接受到 ERET 指令后在 EX 状态将屏蔽解除。

（3）中断时 PC 的保存和恢复

```
always @ (posedge clk) begin
        if(current_state == INTR) begin
                EPC <= PC_tmp;
        end
end
...
always @ (posedge clk) begin
        ...
        else if(current_state[9])
                PC <= 32'h100;
```

```
        else if(current_state[4])begin
                if(inst_ERET)
                        PC   <= EPC;
                else if ...
        end
end
```

中断进入 INTR 状态时将 PC 值寄存至 EPC 寄存器，且 PC 跳转至 intr_handler 入口地址。接受到 eret 指令后，PC 在 EX 状态恢复为 EPC 中寄存值。

3. Intr_handler.S

```
intr_handler:
        # TODO: Please add your own interrupt handler for DMA engine
# Respond to DMA call
        la      $k0, 0x60020000         # base address of DMA MMIO register set
        lw      $k1, 0x14($k0)          # get ctrl_Stat, GPR[k1] = MEM [offset + GPR[k0]]
        li      $k0, 0x7fffffff         # Store imm temporarily, because imm out of range of andi
        and     $k1, $k1, $k0           # Set INTR(31bit) to 0
        la      $k0, 0x60020000         # Recover k0, as base addr
        sw      $k1, 0x14($k0)          # Write back ctrl_stat
# Mark sub buffer according to tail_ptr, last_tail_ptr and dma_size
  # count sub result between tail_ptr
        la      $k0, 0x60020000
        lw      $k0, 0x08($k0)          # GPR[k0] = cur_tail_ptr
        la      $k1, last_tail_ptr
        lw      $k1, 0($k1)             # GPR[k1] = last_tail_ptr
        beq     $k0, $k1, L1           # if DMA_engine work succesfully(change tail),change stat
        nop
  # dma_buf_stat -= sub_buf_num
        la      $k0, dma_buf_stat
        lw      $k1, 0($k0)             # GPR[k0] = dma_buf_stat(origin)
        addi    $k1, $k1, -1            # GPR[k0] = dma_buf_stat(refreshed)
        sw      $k1, 0($k0)             # Write back dma_buf_stat
  # store cur_tail_ptr to last_tail_ptr for next call
        la      $k0, last_tail_ptr      # k0 = addr of last_tail_ptr
        la      $k1, 0x60020000
        lw      $k1, 0x08($k1)          # GPR[k1] = cur_tail_ptr
        sw      $k1, 0($k0)             # store cur to last
# Return from interrupt
```

```
L1:
        eret
```

汇编程序依次需要完成，将 ctrl_stat 最高位(即 intr 信号)清零，取出当前尾指针和上一次中断尾指针的值并进行比较。若二者相等可直接中断返回，否则需要标记完成的子缓冲区，即 dma_buf_stat 减一（由于 dma_engine 实现搬移一个子缓冲区完毕响应一次），并将本次尾指针数据存入 last_tail_ptr 以供下次中断服务使用。

4. Data_mover.c

实现性能计数，与 prj5.1dnn 实验类似，不再赘述。

5. 性能对比

测试任务	时钟周期数
Data_mover_dma	51645054
Data_mover_no_dma	116953279

由上图可见，使用 DMA 外设时钟周期数约为 CPU 直接使用 load 和 store 指令进行数据搬移块时钟周期数的一半。考虑到 data_mover 在 generate_data 环节为 CPU 产生数据并填充进源缓冲区，同样会使用大量的 sw 指令，因此 DMA 实际的性能应当大于 CPU 搬移的两倍。

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）

问题 1：fifo 读数据和内存写有效信号异常

从 fifo 读取数据至内存时出现异常，wr_valid 信号和有效数据不同时拉高，查看框架代码 fifo.v 的实现逻辑发现，由于同步读，fifo 会在 fifo_rden 的后一拍才给出数据及更新 fifo 状态。因此设置 wr_valid 延后一拍拉高，fifo 取出数据不再使用时序逻辑进行存储。且 fifo_rden 信号考虑 fifo 非空，而 wr_valid 不需额外再次考虑。

问题 2：最后一次数据传输异常

上板运行时发现无法终止，使用硬件仿真加速工具查看波形时发现，最后一次传输请求长度异常，原因是当 dma_size 低 5 位时，最后一次传输长度同样为 7，针对低 5 位非零情况计算出的 last_burst_len 为-1。因此在请求长度选择时应当考虑 dma_size 第五位是否为零的情形。

问题 3：汇编程序立即数指令范围不足

开始时在 intr_handler 汇编程序中将 ctrl_stat 最高位清空时使用 andiu，发现立即数超出范围，先将立即数加载到某寄存器中，在使用寄存器指令完成相应操作即可。

问题 4：CPU 不支持中断服务程序中 DIV 指令

开始时使用除法计算子缓冲区完成传输数量，上板运行出错。查看反汇编文件及硬件仿真加速波形发现，CPU 无法支持 DIV 指令，考虑到 dma_engine 实现中为一次子缓冲区完成即中断一次，因此直接判断两次尾指针是否相等代替原本的子缓冲区计算。

三、 对讲义中思考题（如有）的理解和回答

本次实验无思考题。

四、 在课后，你花费了大约___20____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）

建议：

本次实验难度适中，但是在刚开始进行实验时容易对子缓冲区、fifo 等概念理解出错。建议在讲解实验时说明处理器和 DMA 的并行关系、DMA 中读写引擎的并行关系，以及 CPU、DMA 通过 data_mover 和 intr_handler 实现交互的过程。

致谢：

感谢常老师和陈欲晓在硬件仿真工具上的帮助，感谢芦溶民助教对 DMA 运行机制的介绍。