

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 专业 计算机科学与技术
实验项目编号 5.4 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 推送到实验课 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 Verilog HDL 关键代码段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)

• 基于队列结构的 DMA 引擎设计。

DMA 是一种特殊的中断传输方式, 其不同于传统传输方式在于, 只进行一个周期的中断而不需要整个流水线去等待中断传输的完成。

读/写引擎可主动发起突发(Burst)内存访问请求, 然后处理器通过 Load/Store 指令访问 DMA 引擎的控制/状态 I/O 寄存器。

本实验的要求是将数据从 BUFF0 中传输到 BUFF1 中, 在该过程中需要将 BUFF0 划分成多个大小相等的子缓冲区, 子缓冲区大小为 4KB。CPU 正常的功能是填充子缓冲区, 而 DMA 的功能则是需要清空子缓冲区。将这些子缓冲区进一步抽象成任务队列, 并通过队列头(head)/尾(tail)指针对各子缓冲区读/写进行控制。

❑ 将Buffer 0划分成多个大小相等的子缓冲区

- 如本次实验项目子缓冲区大小为4KB
- 处理器: **填充(写)**子缓冲区(**生产者**)
- DMA: **清空(读)**子缓冲区(**消费者**)

❑ 将这些子缓冲区进一步抽象成任务队列, 并通过队列头(head)/尾(tail)指针对各子缓冲区读/写进行控制

- 处理器填充头指针head指向的子缓冲区, 每完整填充一个子缓冲区, head指针加“1”
- DMA从尾指针tail指向的子缓冲区读出数据, 每完成一个子缓冲区读取并全部写入Buffer 1, tail指针加“1”
- 处理器和DMA**并行处理**, 二者互不干扰

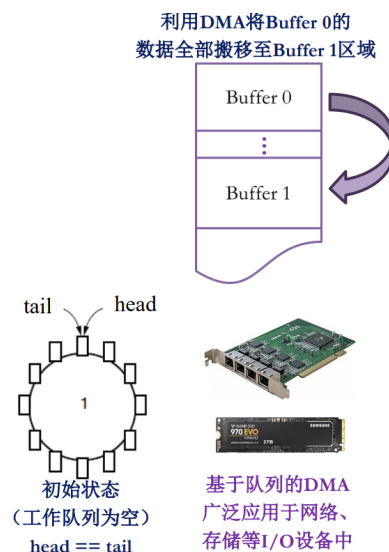


图 1: 队列处理的具体要求

只要子缓冲区还未被填满, CPU 就可以向里面写入数据, 直到写满。只要子缓冲区已被填满, DMA 就可以从里面读出数据, 直至全部读出。

这里我们要将读和写分开处理, 分别设计对应的状态机。PPT 中已经给出了读引擎的状态转移图:

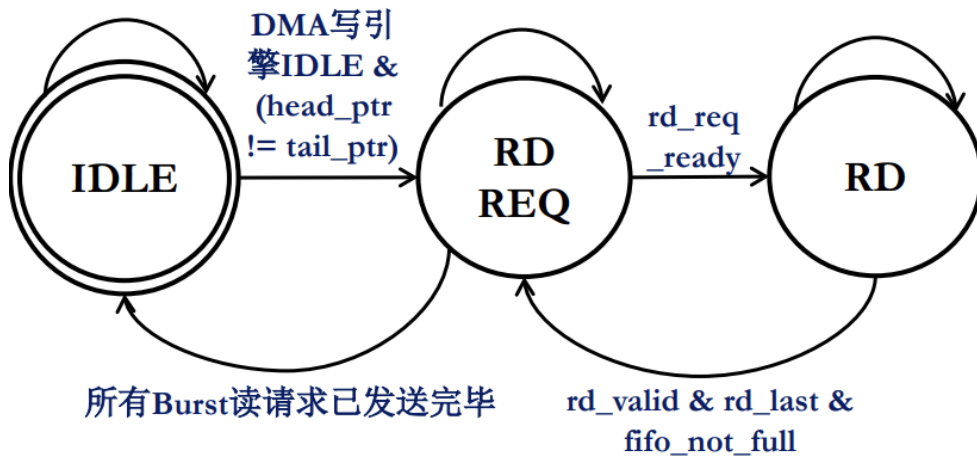


图 2: 读引擎状态转移图

然后我们设计的对应代码为:

```

1  always @ (*) begin
2      case (rd_current_state)
3          `IDLE: begin
4              if (EN && (wr_current_state == `IDLE) && ~(head_ptr == tail_ptr) && ~(rd_counter == burst_times) && fifo_is_empty)
5                  rd_next_state = `RD_WR_REQ;
6              else
7                  rd_next_state = `IDLE;
8              end
9          `RD_WR_REQ: begin
10             if (fifo_is_full || (rd_counter == burst_times))
11                 rd_next_state = `IDLE;
12             else if (rd_req_ready)
13                 rd_next_state = `RD_WR;
14             else
15                 rd_next_state = `RD_WR_REQ;
16             end
17          `RD_WR: begin
18             if (rd_valid && rd_last && ~fifo_is_full)
19                 rd_next_state = `RD_WR_REQ;
20             else
21                 rd_next_state = `RD_WR;
22             end
23          default: rd_next_state = `IDLE;
24      endcase
25  end
  
```

图 3: 读引擎状态转移代码

在 IDLE 状态, 若写引擎处于 IDLE 状态、队列有任务、本次子缓冲区传输读操作未进行完毕和 FIFO 为空时, 读引擎开始工作, 跳转至 REQ 状态。若 FIFO 不满且本次子缓冲区传输读操作未完成时, 需拉高 rd_req_valid 信号并等待请求应答, 当请求应答时跳转至 RD 状态准备接收读数据。跳转至 RD 阶段后, 需拉高 rd_ready 信号并等待请求应答, 当 rd_valid 和 rd_last 信号同时拉高时, 说明本次突发传输完成, 需跳转回 REQ 状态, 等待下一次处理。若 FIFO 已满或本次子缓冲区传输完成, 则需要跳转回 IDLE 状态。

写引擎和读引擎类似, 但是需要注意的是读数据每次读取一个 32bit 数据即可写入队列, 而写数据则需要当数据 FIFO 缓冲有足够一次 Burst 的传输数据时, 再拉起写数据请求, 并在请求应答后逐个发出写数据。因此

我在设计写引擎的状态转移图的时候设计了四个状态,除了读引擎的三个状态外,还有一个 FIFO 状态,当每次 FIFO 有足够一次突发传输数据的数量时才会一起写入。

```
1  always @ (*) begin
2      case (wr_current_state)
3          `IDLE: begin
4              if (EN && (rd_current_state == `IDLE) && ~(head_ptr == tail_ptr) && ~(wr_counter == burst_times) && fifo_is_full)
5                  wr_next_state = `RD_WR_REQ;
6              else
7                  wr_next_state = `IDLE;
8          end
9          `RD_WR_REQ: begin
10             if ((wr_counter == burst_times) || fifo_is_empty)
11                 wr_next_state = `IDLE;
12             else if (wr_req_ready)
13                 wr_next_state = `FIFO;
14             else
15                 wr_next_state = `RD_WR_REQ;
16          end
17          `FIFO: begin
18             wr_next_state = `RD_WR;
19          end
20          `RD_WR: begin
21             if (wr_ready && wr_last || fifo_is_empty)
22                 wr_next_state = `RD_WR_REQ;
23             else if (wr_ready && ~fifo_is_empty)
24                 wr_next_state = `FIFO;
25             else
26                 wr_next_state = `RD_WR;
27          end
28          default: wr_next_state = `IDLE;
29      endcase
30  end
```

图 4: 写引擎状态转移代码

FIFO 状态将 FIFO 中的数据存入寄存器,准备写入内存。然后跳转至 WR 状态,WR 状态需要将拿到的数据存入内存,等待 wr_ready 拉高写入数据,若此时 FIFO 不空,则跳回 FIFO 状态准备下一次写入。使用计量 WR 状态写入数据次数的计数器,每写一次数据计数器加一,当计数器值与需要的数据数量相等时,说明本次 Burst 写操作已完成,需要拉高 wr_last 信号。若 wr_last 信号拉高或 FIFO 为空,则说明本次 Burst 传输完成,准备下一步处理。

对应的 FIFO 状态计数器实现如下图:

```

1  always @ (posedge clk) begin
2      if ((wr_current_state == `FIFO))
3          fifo_data <= fifo_rdata;
4  end
5  always @ (posedge clk) begin
6      if (rst || (wr_current_state == `RD_WR_REQ))
7          fifo_data_counter <= 3'b0;
8      else if ((wr_current_state == `RD_WR) && wr_ready)
9          fifo_data_counter <= fifo_data_counter + 1;
10 end

```

图 5: FIFO 数据传输和计数器实现

然后还有整体的读写计数器需要我们去实现,以判断每次突发传输所需要的传输次数:

```

1  //计数器操作: 每次Burst传输完成, 计数器加一。每次在完成一个DMA子缓冲区的传输后归零。
2  always @ (posedge clk) begin
3      if (rst || (rd_current_state == `IDLE) && (wr_current_state == `IDLE) && EN &&
4          ~(head_ptr == tail_ptr) && (rd_counter == burst_times) && (wr_counter == burst_times))
5          rd_counter <= 32'b0;
6      else if ((rd_current_state == `RD_WR) && rd_valid && rd_last)
7          rd_counter <= rd_counter + 1;
8  end
9  always @ (posedge clk) begin
10     if (rst || (rd_current_state == `IDLE) && (wr_current_state == `IDLE) && EN &&
11         ~(head_ptr == tail_ptr) && ~intr && (rd_counter == burst_times) && (wr_counter == burst_times))
12         wr_counter <= 32'b0;
13     else if ((wr_current_state == `RD_WR) && wr_ready && wr_last)
14         wr_counter <= wr_counter + 1;
15 end

```

图 6: 读写计数器实现

读写引擎计数器每次在完成一个 DMA 子缓冲区的传输后归零,当处于 RD/WR 状态且 rd_valid/wr_ready 和 last 信号有效时,表示该次突发传输完成,计数器加一。

需要注意的是,最后一次 Burst 传输的长度为 $(N\%32)$ Byte,需要补齐到 4-byte 对齐的大小。

然后需要编写的是各个输出的赋值:

```

1  /*读引擎与内存控制器的互连端口输出赋值*/
2  assign rd_req_addr = src_base + tail_ptr + (rd_counter << 5); //每次传输32bit, 所以左移5位
3  //传输长度为4*(rd_req_len+1)字节,若传输32字节, 则取7, 若最后一次传输不为字节, 则取last_burst。
4  assign rd_req_len = ((rd_counter == burst_times - 1) && |last_burst)? {2'b0, {3{last_burst - 1'b1}}} : 5'b111;
5  assign rd_req_valid = (rd_current_state == `RD_WR_REQ) & ~fifo_is_full & ~(rd_counter == burst_times);
6  assign rd_ready = (rd_current_state == `RD_WR);
7
8  /*写引擎与内存控制器的互连端口输出赋值*/
9  assign wr_req_addr = dest_base + tail_ptr + (wr_counter << 5);
10 assign wr_req_len = ((wr_counter == burst_times - 1) && |last_burst)? {2'b0, {3{last_burst - 1'b1}}} : 5'b111;
11 assign wr_req_valid = (wr_current_state == `RD_WR_REQ) & ~fifo_is_empty;
12 assign wr_valid = (wr_current_state == `RD_WR);
13 assign wr_data = fifo_data;
14 assign wr_last = (fifo_data_counter == wr_req_len[2:0]);
15
16 /*读写引擎与FIFO队列接口信号输出赋值*/
17 assign fifo_rden = (wr_next_state == `FIFO);
18 assign fifo_wen = rd_ready & rd_valid & ~fifo_is_full;
19 assign fifo_wdata = rd_rdata;
20
21 //突发传输次数计算
22 assign last_burst = dma_size[4:2] + |dma_size[1:0]; //最后一次传输长度int((N2)/4) + ( ((N % 32) % 4) != 0
23 assign burst_times = {5'b0, dma_size[31:5]} + |dma_size[4:0]; //总共需要传输的次数

```

图 7: 各个输出的赋值

上图为 engine_core 模块中部分输出的赋值, 该部分输出只需要用组合逻辑即可实现, 不需要时序逻辑。然后有一部分输出寄存器/指针的值, 需要 we 根据对应的时序进行实现的:

```

1 //控制/状态寄存器与处理器的互连端口的赋值
2 always @ (posedge clk) begin
3     if (reg_wr_en[0])
4         src_base <= reg_wr_data;
5 end
6 always @ (posedge clk) begin
7     if (reg_wr_en[1])
8         dest_base <= reg_wr_data;
9 end
10 always @ (posedge clk) begin
11     if (reg_wr_en[2])
12         tail_ptr <= reg_wr_data;
13     //读写数据量全部达到dma_size时，本次DMA子缓冲区传输结束，发送中断请求
14     else if ((rd_counter == burst_times) && (wr_counter == burst_times) &&
15             (rd_current_state == `IDLE) && (wr_current_state == `IDLE))
16         tail_ptr <= tail_ptr + dma_size;
17 end
18 always @ (posedge clk) begin
19     if (reg_wr_en[3])
20         head_ptr <= reg_wr_data;
21 end
22 always @ (posedge clk) begin
23     if (reg_wr_en[4])
24         dma_size <= reg_wr_data;
25 end
26 always @ (posedge clk) begin
27     if (reg_wr_en[5])
28         ctrl_stat <= reg_wr_data;
29     //读写数据量全部达到dma_size时，本次DMA子缓冲区传输结束，发送中断请求
30     else if (EN && (rd_counter == burst_times) && (wr_counter == burst_times) &&
31             (rd_current_state == `IDLE) && (wr_current_state == `IDLE))
32         ctrl_stat[31] = 1'b1;
33 end

```

图 8: 输出寄存器/指针的赋值

- custom_cpu 中断处理设计。

该部分要实现的内容较少,主要是修改状态机,增加一个中断状态,对应的状态转移图如下:

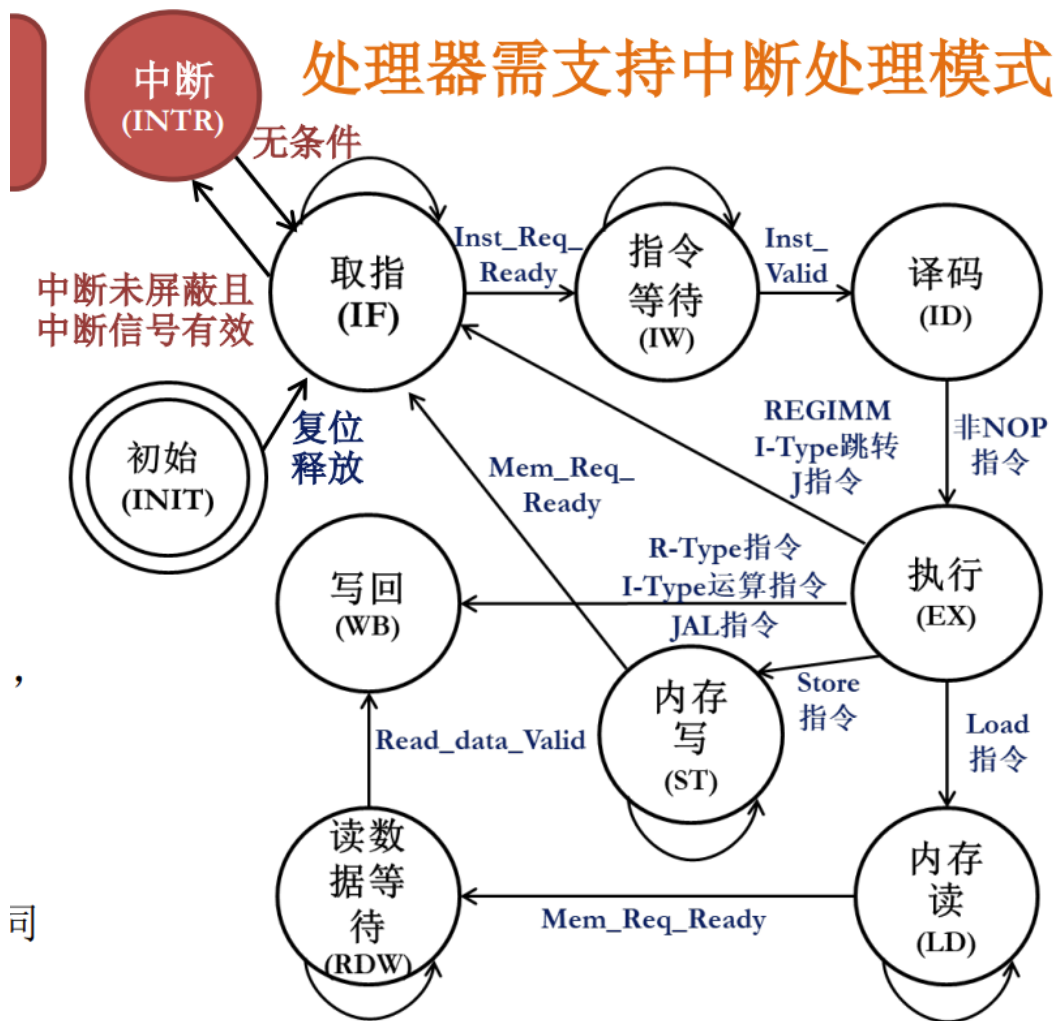


图 9: cpu 状态转移图

当处于取指状态时需要进行判断, 如果中断信号有效并且中断未屏蔽, 则需要转移到中断状态, 然后在下一周期无条件跳转回取指状态, 进行中断周期。在中断状态, 需要屏蔽外部中断信号, 保存现场后, 处理器跳回 IF 状态, 开始执行中断服务程序。

该部分具体要求是需要保护 PC 值, 将此时的 PC 存入寄存器以备退出中断后使用, 再将 PC 设置为 0x100 从而进入中断程序。

```

1  `IF: begin
2      if (intr && ~shield)
3          next_state = `INTR;
4      else if (Inst_Req_Ready)
5          next_state = `IW;
6      else
7          next_state = `IF;
8  end
9  `INTR: begin
10     next_state = `IF;
11 end

```

图 10: cpu 状态机修改

```

1  always @ (posedge clk) begin
2      if (rst | EX_state & ERET)
3          shield <= 1'b0;
4      else if (INTR_state) begin
5          shield <= 1'b1;
6          EPC <= PC;
7      end
8  end

```

图 11: 中断请求和屏蔽的更新

中断服务程序的最后一条指令为 ERET 指令,该指令在执行阶段需用 EPC 值重新设置 PC,同时解除中断屏蔽,处理器退出中断模式。

- 中断服务程序设计。

该部分的具体要求及实现思路如下:

- 中断服务程序的入口地址为0x100
 - 如果全部写在0x100开始的位置，程序大小需要不超过256 Byte（64条指令）
- 在中断服务程序中仅能使用K0、K1两个寄存器
- 中断服务程序需完成如下功能
 - ① 响应中断：将DMA的ctrl_stat寄存器INTR标志位清0
 - ② 根据DMA引擎tail_ptr寄存器的内容，标记已完成传输的子缓冲区
 - ③ 中断返回ERET（一定不能遗忘）

图 12: 中断服务程序编写要求

然后通过查看头文件 dma.h,找到对应的入口地址如下：



图 13: 中断服务程序各寄存器地址及偏移

然后我们根据上图需要完成的功能,先写出对应的 C 语言代码：

```

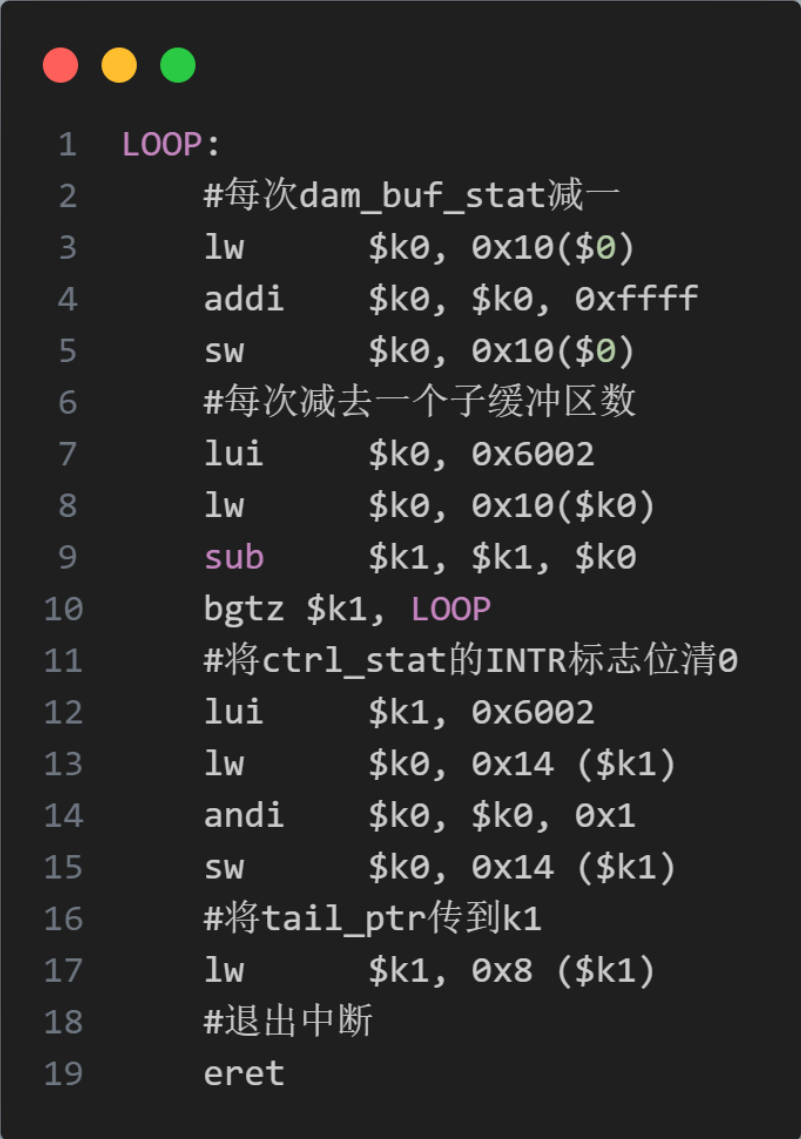
1 volatile uint32_t* get_address(uint32_t base, uint32_t offset) {
2     return (volatile uint32_t*)(base + offset);
3 }
4
5 void intr_handler() {
6     // 定义指向内存映射寄存器的指针
7     volatile uint32_t* tail_ptr = get_address(DMA_BASE, TAIL_PTR_OFFSET);
8     volatile uint32_t* dma_size = get_address(0, DMA_SIZE_OFFSET);
9     volatile uint32_t* ctrl_stat = get_address(DMA_BASE, CTRL_STAT_OFFSET);
10
11     // 加载tail_ptr的值
12     uint32_t k0 = *tail_ptr;
13     uint32_t k1 = 0;
14
15     // 计算差值
16     k1 = k0 - k1;
17
18     // 循环更新dam_buf_stat
19     do {
20         // 减少dam_buf_stat
21         uint32_t buf_stat = *dma_size;
22         buf_stat -= 1;
23         *dma_size = buf_stat;
24
25         // 从k1中减去一个子缓冲区大小
26         k0 = *get_address(DMA_BASE, DMA_SIZE_OFFSET);
27         k1 -= k0;
28     } while (k1 > 0);
29
30     // 清除ctrl_stat寄存器中的INTR标志位
31     uint32_t ctrl_stat_val = *ctrl_stat;
32     ctrl_stat_val &= 0xFFFFF0; // 清除INTR标志
33     *ctrl_stat = ctrl_stat_val;
34
35     // 记录新的tail_ptr值
36     k1 = *tail_ptr;
37 }

```

图 14: 中断服务程序 C 语言代码

首先将 tail_ptr 值存入 k0 寄存器,将 k0 的值减去 k1 的值存入 k1,得到 tail_ptr 的差值,使用循环计算该差值除以 dma_size 的值,从而得到前后两次进入中断程序所处理的子缓冲区数目。当处理器进行完中断操作后需将 intr 信号拉低。最后处理器将 tail_ptr 值存入 k1 寄存器以备下次进入中断使用,并执行 eret 指令退出中断服务。

对应的汇编语言如下:



```

1  LOOP:
2      #每次dam_buf_stat减一
3      lw      $k0, 0x10($0)
4      addi    $k0, $k0, 0xffff
5      sw      $k0, 0x10($0)
6      #每次减去一个子缓冲区数
7      lui     $k0, 0x6002
8      lw      $k0, 0x10($k0)
9      sub     $k1, $k1, $k0
10     bgtz    $k1, LOOP
11     #将ctrl_stat的INTR标志位清0
12     lui     $k1, 0x6002
13     lw      $k0, 0x14 ($k1)
14     andi    $k0, $k0, 0x1
15     sw      $k0, 0x14 ($k1)
16     #将tail_ptr传到k1
17     lw      $k1, 0x8 ($k1)
18     #退出中断
19     eret

```

图 15: 中断服务程序汇编语言代码

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)

• 实验设计问题。

本实验中遇到了一个很大的问题就是 fpga 运行超时, 然后我通过加速仿真也无法完成, 在我采用控制加速仿真区间的方法得到了波形之后, 发现 intr 信号始终为 0, 但是这又和本实验本身实际情况不符, 在询问助教之后也没有得到实质性的解释。所以只能放弃这种 debug 的方式。

本实验最后我采用的方式是逐段修改, 然后反复上板测试, 查看究竟是哪个部分出现了问题, 一开始发现我的 engine_core.v 和 custom_cpu.v 都存在问题。然后又分别去查看, 发现 engine_core.v 是对六个计数器或指针

赋值的时序逻辑出现了问题,在参考了 github 上学长学姐们的仓库之后进行了修改,最终解决了该部分的问题。

```
1 //控制/状态寄存器与处理器的互连端口的赋值
2 always @ (posedge clk) begin
3     if (reg_wr_en[0])
4         src_base <= reg_wr_data;
5 end
6 always @ (posedge clk) begin
7     if (reg_wr_en[1])
8         dest_base <= reg_wr_data;
9 end
10 always @ (posedge clk) begin
11     if (reg_wr_en[2])
12         tail_ptr <= reg_wr_data;
13     //读写数据量全部达到dma_size时,本次DMA子缓冲区传输结束,发送中断请求
14     else if ((rd_counter == burst_times) && (wr_counter == burst_times) &&
15             (rd_current_state == `IDLE) && (wr_current_state == `IDLE))
16         tail_ptr <= tail_ptr + dma_size;
17 end
18 always @ (posedge clk) begin
19     if (reg_wr_en[3])
20         head_ptr <= reg_wr_data;
21 end
22 always @ (posedge clk) begin
23     if (reg_wr_en[4])
24         dma_size <= reg_wr_data;
25 end
26 always @ (posedge clk) begin
27     if (reg_wr_en[5])
28         ctrl_stat <= reg_wr_data;
29     //读写数据量全部达到dma_size时,本次DMA子缓冲区传输结束,发送中断请求
30     else if (EN && (rd_counter == burst_times) && (wr_counter == burst_times) &&
31             (rd_current_state == `IDLE) && (wr_current_state == `IDLE))
32         ctrl_stat[31] = 1'b1;
33 end
```

图 16: engine_core 中对寄存器和指针赋值的修改

然后是 cpu 的修改,也采用了同样的方法,但是并没有能够精确的具体是那一部分出现了问题,这个和 dnm 部分的实验一样,应该是我之前的 cpu 写的有些问题我没有发现,之前的 cpu 的仿真测试以及 fpga 上板都没有检测到问题。

三、实验所耗时间

在课后,你花费了大约 20 小时完成此次实验。

四、实验思考与心得

关于是否使用 dma 的性能差异对比,从最后的 fpga 结果中可以看出,使用 dma 的方式花费的时间仅为未使用 dma 方式的不到一半,可以看出 dma 方式对于中断传输的提升巨大。

```
47 benchmark finished
48 time 823.06ms
49 reset: before MMIO access...
46 benchmark finished
47 time 1948.28ms
48 reset: before MMIO access...
```

图 17: 性能对比:使用 dma(上), 未使用 dma(下)

关于本实验的心得, 该部分在理解上稍有些难度, 不过是在期末考完之后再开始写该部分实验, 理论课程已经学完了该部分内容, 故不会有之前没有学习理论知识就写实验的困难。

该部分和 dnn 部分不提供行为仿真, 这给同学们 debug 的过程增加了很多困难, 而且我在本实验和 dnn 实验中都使用 fpga 加速仿真来帮助我 debug, 但是作用都不是很大, 希望老师和助教们能够在以后给同学们提供一些更有效更方便的 debug 方法。