中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929016　　姓名：李金明　专业：计算机科学与技术

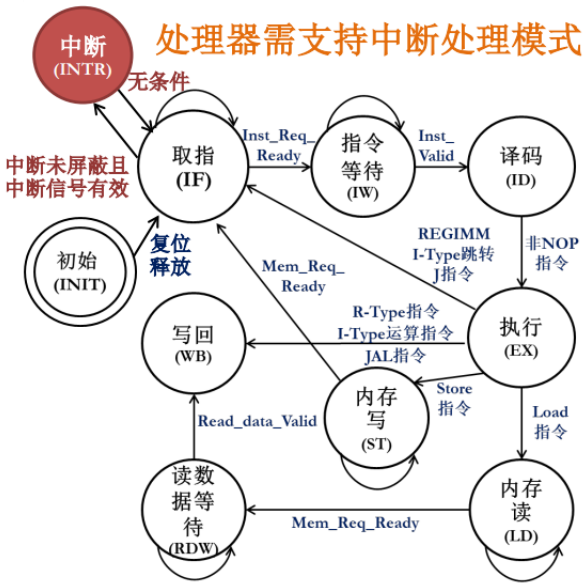
实验序号：5.2　　实验名称：DMA 引擎与中断处理

一、 关键代码及说明

1. custom_cpu 改动：在 MIPS 定制处理器中添加中断处理

a. 状态机的更改

Custom_cpu 首先需要实现对中断处理的支持，需要更改状态机，更改后的状态机如下：



更改的主要是增加中断状态，以及在 ERET 指令执行阶段更新 PC 值。状态转移更改部分由如下代码给出：

```
326             `S_IF: begin
327                 if (intr && ~shield) next_state = `S_INTR;
328                 else if (Inst_Req_Ready) next_state = `S_IW;
329                 else next_state = `S_IF;
330             end
331             `S_INTR: begin
332                 next_state = `S_IF;
333             end
```

当中断未屏蔽且中断信号有效，等待处理器在进入 IF 阶段时，需进入中断处理阶段（INTR）。

b. INTR 阶段的处理

在 INTR 阶段，处理器需要对中断信号加以屏蔽，并将此时的 PC 存入寄存器以备退出中断后使用，再讲 PC 设置为 0x100 从而进入中断程序，由以下代码给出：

```
367     always @ (posedge clk) begin
368         if (rst | state_EX & ERET) shield <= 1'b0;
369         else if (state_INTR) begin
370             shield <= 1'b1;
371             EPC <= PC;
372         end
373     end
```

处理器复位时需将屏蔽信号复位，屏蔽信号需要在 INTR 阶段拉高，并在 ERET 指令的指令执行阶段解除屏蔽。

c. PC 的更新

```
507     always @(posedge clk) begin
508         if (rst) PC <= 32'b0;
509         else if (state_IF) PC_4 <= ALU_Result;
510         else if (state_INTR) PC <= 32'h100;
511         else if (state_ID && Instruction_reg == 32'b0) PC <= PC_4;
512         else if (state_EX) begin
513             if (R_Type_jump) PC <= RF_rdata1;
514             else if (J_Type) PC <= {PC_4[31:28], instr_index[25:0], 2'b00};
515             else if (I_Type_branch && (~opcode[1] && (opcode[0] ^ Zero) ||
516                                        (opcode[1] && (opcode[0] ^ (ALU_Result == 32'b0)))) ||(REGIMM && (rt[0] ^ (ALU_Result == 32'b1))))
517                 PC <= ALUOut;
518             else if (ERET) PC <= EPC;
519             else PC <= PC_4;
520         end
521     end
```

更改在于在 INTR 状态，PC 值更新为 0x100 以进入中断处理程序，在退出中断程序时将 PC 值还原为原先的进入中断处理前的 PC 值。

d. Inst_Req_Valid 信号的更改

```
420     assign Inst_Req_Valid = state_IF & ~(intr & ~shield);
```

这一部分是经张士寅同学提醒而更改的，当处理器准备进入中断处理程序时，不应再拉高 Inst_Req_Valid 信号，不再试图与内存握手，否则可能导致冲突，从而在 FPGA 阶段发生错误。

2. engine_core 编写：设计实现 DMA 引擎硬件逻辑

a. DMA 引擎控制/状态寄存器

DMA 总共有 5 个控制/状态寄存器，它们的含义由下表给出：

偏移地址	寄存器名	访问方式	寄存器说明
0x0000	src_base	处理器可读写	DMA读引擎访问内存的基地址，初值为0
0x0004	dest_base	处理器可读写	DMA写引擎访问内存的基地址，初值为0
0x0008	tail_ptr	处理器可读写	DMA队列尾指针。在DMA完成操作后，由DMA硬件自动更新；处理器可设置该寄存器。初值为0
0x000C	head_ptr	处理器可读写	DMA队列头指针。处理器完成一个子缓冲区的准备工作后，更新该寄存器。初值为0
0x0010	dma_size	处理器可读写	DMA传输子缓冲区的大小（单位：字节） 由处理器设置，初值为0
0x0014	ctrl_stat	处理器可读写	DMA使能/中断标志寄存器(DMA_CTRL_STAT)

这些寄存器的赋值如下：

```
100     always @ (posedge clk) begin
101         if (reg_wr_en[0]) src_base <= reg_wr_data;
102     end
103     always @ (posedge clk) begin
104         if (reg_wr_en[1]) dest_base <= reg_wr_data;
105     end
106     always @ (posedge clk) begin
107         if (reg_wr_en[2]) tail_ptr <= reg_wr_data;
108         else if (rd_complete_burst & wr_complete_burst & state_rd_IDLE & state_wr_IDLE)
109             tail_ptr <= tail_ptr + dma_size;
110     end
111     always @ (posedge clk) begin
112         if (reg_wr_en[3]) head_ptr <= reg_wr_data;
113     end
114     always @ (posedge clk) begin
115         if (reg_wr_en[4]) dma_size <= reg_wr_data;
116     end
117     always @ (posedge clk) begin
118         if (reg_wr_en[5]) ctrl_stat <= reg_wr_data;
119         else if (EN & rd_complete_burst & wr_complete_burst & state_rd_IDLE & state_wr_IDLE)
120             ctrl_stat[31] = 1'b1;
121     end
122     assign intr = ctrl_stat[31];
123     assign EN = ctrl_stat[0];
```

首先依靠寄存器写使能信号 reg_wr_en 值这些寄存器的值。Reg_wr_en 采

用独热码编码，其 0-5 位分别控制表中 6 个寄存器从上到下的写入，当对应写使能拉高时，reg_wr_data 会存入相应的寄存器。

在其他情况需要写入的寄存器是 tail_ptr 和 ctrl_stat。当读写数据量达到 dma_size 时，本次 DMA 子缓冲区传输结束，需要发送中断请求。因此首先要将 tail_ptr 加上 dma_size，表示相应的子缓冲区已处理完毕，并将 ctrl_stat 的第 31 位拉高，即拉高 intr 值，使得处理器进入中断服务。

补充：ctrl_stat 信号的第 31 位代表中断标志位，第 0 位代表 DMA 使能位，当使用 DMA 引擎时在处理器初始化时拉高该位

b. burst 次数计数器及相关信号

```
126     assign last_burst = dma_size[4:2] + |dma_size[1:0];
127     assign total_burst_times = {5'b0, dma_size[31:5]} + |dma_size[4:0];
128     assign rd_complete_burst = rd_counter == total_burst_times;
129     assign wr_complete_burst = wr_counter == total_burst_times;
```

```
164 always @ (posedge clk) begin
165     if (rst | state_rd_IDLE & state_wr_IDLE & EN & ~equal & rd_complete_burst & wr_complete_burst)
166         rd_counter <= 32'b0;
167     else if (state_rd_RD & rd_valid & rd_last)
168         rd_counter <= rd_counter + 1;
169 end
```

```
213 always @ (posedge clk) begin
214     if (rst | state_rd_IDLE & state_wr_IDLE & EN & ~equal & ~intr & rd_complete_burst & wr_complete_burst)
215         wr_counter <= 32'b0;
216     else if (state_wr_WR & wr_ready & wr_last)
217         wr_counter <= wr_counter + 1;
218 end
```

DMA 引擎 Burst 传输长度小于等于 32Byte，故传输一个大小为 dma_size（单位为 Byte）的 DMA 子缓冲区需要的 Burst 传输次数为 dma_size/32 + (dma_size % 32 != 0)，最后一次 Burst 由于需要补齐到 4Byte，故最后一次 Burst 传输的长度为((dma_size % 32)/4) + ((dma_size%32)%4 != 0)。

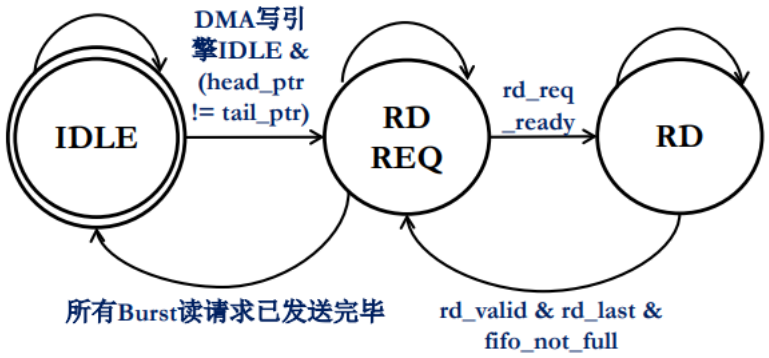
读（写）引擎计数器每次在完成一个 DMA 子缓冲区的传输后归零，当处于 RD（WR）状态且 rd_valid（wr_ready）和 last 信号有效时，表示该次 Burst 传输完成，计数器加一。

c. 读引擎

```
139     always @ (posedge clk) begin
140         if (rst) rd_current_state <= `S_IDLE;
141         else rd_current_state <= rd_next_state;
142     end
143
144     always @ (*) begin
145         case (rd_current_state)
146             `S_IDLE: begin
147                 if (EN & state_wr_IDLE & ~equal & ~rd_complete_burst & fifo_is_empty)
148                     rd_next_state = `S_REQ;
149                 else rd_next_state = `S_IDLE;
150             end
151             `S_REQ: begin
152                 if (fifo_is_full | rd_complete_burst) rd_next_state = `S_IDLE;
153                 else if (rd_req_ready) rd_next_state = `S_RD_WR;
154                 else rd_next_state = `S_REQ;
155             end
156             `S_RD_WR: begin
157                 if (rd_valid & rd_last & ~fifo_is_full) rd_next_state = `S_REQ;
158                 else rd_next_state = `S_RD_WR;
159             end
160             default: rd_next_state = `S_IDLE;
161         endcase
162     end
```

```
171     assign rd_req_addr = src_base + tail_ptr + (rd_counter << 5);
172     assign rd_req_len = rd_complete_burst? {2'b0, last_burst} : 5'b111;
173     assign rd_req_valid = state_rd_REQ & ~fifo_is_full & ~rd_complete_burst;
174     assign rd_ready = state_rd_RD;
```

读引擎状态机状态转移图由下图给出：



在 IDLE 状态，若写引擎处于 IDLE 状态、队列有任务（tail_ptr 值和 head_ptr 不相等）、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。

跳转至 REQ 状态时，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 Burst 传输完成，需跳转回 REQ 状态，等待下一次处理。

跳转回 REQ 状态时，若 FIFO 已满或本次子缓冲区传输完成，则需要跳转回 IDLE 状态（这与所给状态转移图有小区别，这是因为设计的 FIFO 空间大小不足以存储一个 DMA 子缓冲区所有的数据量），否则等待 rd_req_valid 信号拉高进入 RD 状态，开始下一次 Burst 传输。

Rd_req_addr 信号代表读地址，只需在每次 Burst 传输时传输一次，传递一个子缓冲区所读数据的基地址为 src_base + tail_ptr，每次 Burst 传输 32Byte，因此每次 Burst 传输的读地址为上式加 Burst 传输次数乘上 32。

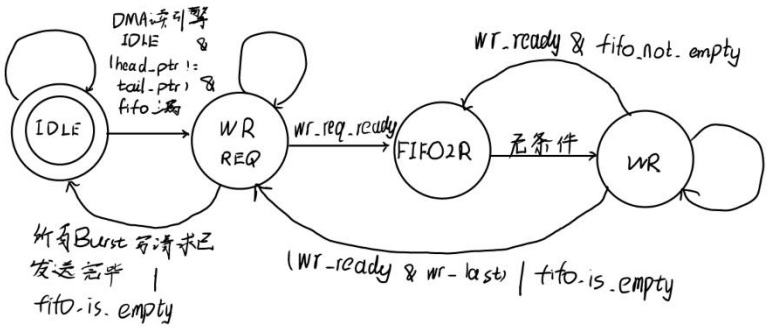
Rd_req_len 信号代表 Burst 传输长度，每次 Burst 传输 4 *(rd_req_len + 1)，若传输 32Byte，则取 7；若最后一次传输不为 32Byte，则 rd_req_len 值为 last_burst。

d. 写引擎

```
179     always @ (posedge clk) begin
180         if (rst) wr_current_state <= `S_IDLE;
181         else wr_current_state <= wr_next_state;
182     end
183 
184     always @ (*) begin
185         case (wr_current_state)
186             `S_IDLE: begin
187                 if (EN & state_rd_IDLE & ~equal & ~wr_complete_burst & fifo_is_full)
188                     wr_next_state = `S_REQ;
189                 else wr_next_state = `S_IDLE;
190             end
191             `S_REQ: begin
192                 if (wr_complete_burst | fifo_is_empty) wr_next_state = `S_IDLE;
193                 else if (wr_req_ready) wr_next_state = `S_FIFO2R;
194                 else wr_next_state = `S_REQ;
195             end
196             `S_FIFO2R: begin
197                 wr_next_state = `S_RD_WR;
198             end
199                 `S_RD_WR: begin
200                 if (wr_ready & wr_last | fifo_is_empty) wr_next_state = `S_REQ;
201                 else if (wr_ready & ~fifo_is_empty) wr_next_state = `S_FIFO2R;
202                 else wr_next_state = `S_RD_WR;
203             end
204             default: wr_next_state = `S_IDLE;
205         endcase
206     end
```

```
220     always @ (posedge clk) begin
221         if (rst | state_wr_REQ) wdata_counter <= 3'b0;
222         else if (state_wr_WR & wr_ready) wdata_counter <= wdata_counter + 1;
223     end
224
225     assign wr_req_valid = state_wr_REQ & ~fifo_is_empty;
226     assign wr_req_addr = dest_base + tail_ptr + (wr_counter << 5);
227     assign wr_req_len = (wr_counter == total_burst_times - 1 && |last_burst)? {2'b0, last_burst} : 5'b111;
228     assign wr_valid = state_wr_WR;
229     assign wr_data = fifo_data;
230     assign wr_last = wdata_counter == wr_req_len[2:0];
```

写引擎状态机由以下状态转移图给出：



当 FIFO 队列满后，写引擎开始工作。在 IDLE 状态，若读引擎处于 IDLE 状态，写引擎开始工作，跳转至 REQ 状态。

跳转至 REQ 状态后，若队列不为空，需要拉高 wr_req_valid 信号，等待请求应答，然后进入 FIFO2R 状态。

FIFO2R 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 FIFO2R 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（在 WR 状态且 wr_ready 拉高），计数器加一，当计数器值与 wr_req_len 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 传输已结束，准备下一步处理。

跳转回 REQ 状态后，若队列仍不为空，则再拉高 wr_req_valid 准备下一

次写入，若队列为空本次子缓冲区所有 Burst 请求已完成，则跳转回 IDLE 状态。

Wr_req_addr 信号与 wr_req_len 信号处理与读引擎相似，不再赘述。

向内存写入的数据 wr_data 是从 FIFO 中取到的数据，在下一部分中讲述。

e. FIFO 队列处理

```
176     assign fifo_wen = rd_ready & rd_valid & ~fifo_is_full;
177     assign fifo_wdata = rd_rdata;
```

```
208     assign fifo_rden = wr_next_state == `S_FIFO2R;
209     always @ (posedge clk) begin
210         if (state_wr_FIFO2R) fifo_data <= fifo_rdata;
211     end
```

读引擎操作的是有关 FIFO 写的操作，当读引擎在 RD 状态时（rd_ready 拉高)、rd_valid 拉高且 FIFO 不满时，说明 FIFO 可以写入数据且 DMA 已经握手成功，此时拉高 FIFO 写使能，写入数据是 DMA 读到的数据。

写引擎操作的是有关 FIFO 读的操作，当写引擎处于 FIFO2R 状态时，拉高 FIFO 读使能，并将读到的数据存入寄存器，准备写入内存。

3. intr_handler 编写：中断服务汇编程序

```
13      #base: 0x60020000
14      #offset:
15      #tail_ptr: 0x0008
16      #dma_size: 0x0010
17      #ctrl_stat: 0x0014
18
19      #获取tail_ptr的值存到k0寄存器，计算前后差值
20      lui     $k0, 0x6002
21      lw      $k0, 0x8 ($k0)
22      sub     $k1, $k0, $k1
23
24      #通过循环计算子缓冲区数并更新dam_buf_stat值
25  LOOP:
26      #每次dam_buf_stat减一
27      lw      $k0, 0x10($0)
28      addi    $k0, $k0, 0xffff
29      sw      $k0, 0x10($0)
30
31      #每次减去一个子缓冲区数
32      lui     $k0, 0x6002
33      lw      $k0, 0x10($k0)
34      sub     $k1, $k1, $k0
35
36      bgtz $k1, LOOP
37
38      #将DMA的ctrl_stat寄存器INTR标志位清0
39      lui     $k1, 0x6002
40      lw      $k0, 0x14 ($k1)
41      andi    $k0, $k0, 0x1
42      sw      $k0, 0x14 ($k1)
43
44      #记录tail_ptr的值到k1寄存器
45      lw      $k1, 0x8 ($k1)
46
47      #退出中断
48      eret
```

通过查看 dma.h 文件可以得知寄存器的基址时 0x60020000，需要使用的 DMA 引擎控制/状态机的偏移地址在 PPT 中给出。进入中断程序时，k1 寄存器存的时上一次进入中断程序时 tail_ptr 的值（若未进入过则为 0）。

首先将此时的 tail_ptr 值存入 k0 寄存器，将 k0 的值减去 k1 的值存入 k1，得到前后 tail_ptr 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。具体实现为在循环中，每次让差值

减去 dma_size 并存入 k1，并让 dam_buf_stat（代表待处理子缓冲区数量）减一，当 k1 所存值为 0 时退出循环。

处理完后将 ctrl_stat 与 0x1 作与操作并存回去，这是因为 ctrl_stat 只有第 0 位（EN）和第 31 位（intr）有意义，当处理器进行完中断操作后需将 intr 信号拉低。

最后处理器将选择的 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。

4. 性能计数器的结果

我在 data_mover.c 文件中加入了性能计数器的调用，以下是结果对比：

不使用 DMA：

```
51  Cycles: 190028043
52  benchmark finished
53  time 1905.82ms
```

使用 DMA：

```
51  Cycles: 75882742
52  benchmark finished
53  time 764.08ms
```

可以观察到,使用 DMA 引擎迁移数据比不使用 DMA 引擎所耗周期数降低了一半以上，这是由于 DMA 引擎与 CPU 是并行工作的，CPU 处理的只有中断服务程序，而不用使用大量的 lw/sw 指令来迁移数据。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1. 在使用仿真加速对程序进行测试时能通过，但实际上板测试无法通过。

解决方法：我首先在 custom_cpu 中增加了几个性能计数器，用来统计处理器进入中断服务程序的次数，发现 custom_cpu 从未进入过中断服务程序。由于 ctrl_stat 信号第 1 位到第 30 位是无用的且知道地址，我又向这几位赋值并打印出来，通过这样的方式实现了 bug 的剔除。另再次感谢张士寅同学提示我对 custom_cpu 中的 inst_req_valid 做出修改。

2. 仿真加速时在处理器发送 inst_req_valid 信号和 DMA 发送 rd_req_valid 后内存一直不返回对应的 ready 信号。

解决方法：由于这两个 ready 信号都是输入信号，导致我花了较长时间没有想明白为什么不会拉高。后来将各种 valid 信号和 ready 信号罗列出来，发现 wr_ready 信号一直在拉高，而 wr_valid 信号由于设计失误，在 fifo 读空前的最后一次 burst 传输不拉高，导致 DMA 和内存一直握手不成功。

3. 按照课上 PPT 给出的读引擎状态机设计不能正确工作。

解决方法：这是因为数据缓冲 FIFO 容量较小，仅能容纳 20（16 进制）次 Burst 传输的数据量，即 32×32Byte，而一个子缓冲区传输完需要 80（16 进制）次 Burst 传输，即 128×32Byte，故应在 FIFO 满时就将读引擎置于 IDLE 状态待机，开始写引擎的工作。

三、 在课后，你花费了大约 35 小时完成此次实验?

四、 对于此次实验的心得、感受和建议

这部分实验难度较大。很大程度上是由于寻找 bug 很不容易。

在实验开始阶段，可以使用仿真加速寻找 bug。这里难点在于寻找到 DMA 没有正常工作的时间周期。由于 DMA 不正常工作程序会陷入死循环，处理器不会完成工作直至超时，故在仿真加速时我提交了好几次以确定错误发生的周期。

在可以通过仿真加速测试后才是最头疼的：实际上板 fpga 无法通过。因此我又提交了好几次，借助 ctrl_stat 信号无用的信号位来输出一些值，看他们哪里出现了异常。这里加重自己实验难度的点在于某次改动导致逻辑出现了问题，连仿真加速都过不了，而我却过了很久才意识到这个问题。

总而言之这个实验的难度较大，主要就在于 debug 的过程过于艰辛。

感谢主讲老师及各位助教老师的讲解，让我深入了解到了 DMA 的工作原理、在数据搬移过程中的工作方式及其在提升效率方面的巨大作用。