中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　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．实验目的

1. 理解复杂 I/O 外设的设计原理：设计实现基于队列结构的 DMA 硬件引擎；
2. 理解 CPU-外设协同工作原理：在 MIPS 处理器中，添加中断处理逻辑并编写中断服务程序；并支持中断屏蔽，实现中断返回指令（ERET）。

二．实验背景知识

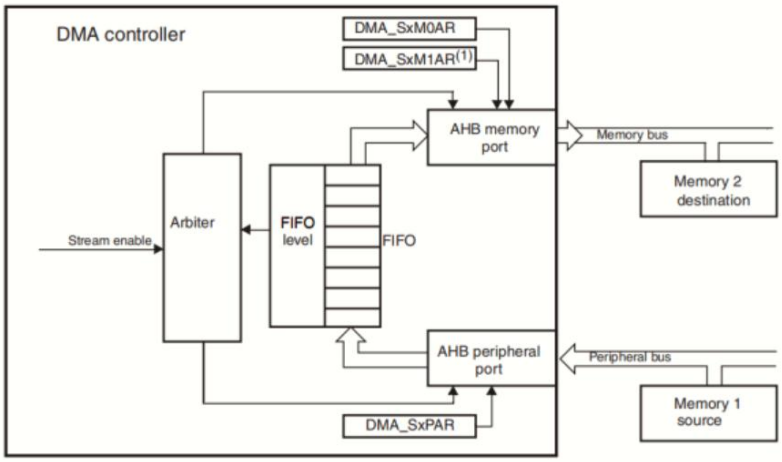
我们在理论课上已经学过 DMA 传输，它是网卡、磁盘、显卡等**高速外设**特有的 I/O 方式。I/O 设备与内存进行数据传输有三种基本方式：程序查询方式（最简单的 I/O 方式）、I/O Interrupt (中断 I/O 方式)、Direct Memory Access (DMA 方式)。课上有一道例题还计算了使用程序查询方式时，不同 I/O 工作时占用 CPU 的时间，结果是对于鼠标这样的低速 I/O，CPU 对它的程序查询基本不会影响性能；但是对于硬盘这样的大数据量、高速的 I/O，即使将全部的 CPU 时间都用于查询硬盘也不能满足传输要求。

可以看出，程序查询方式的 I/O 传输对于 CPU 的依赖是很大的，同样，I/O 中断方式也需要通过 CPU 的访存指令来访问内存，只是将程序查询软件通过 I/O 中断指令来完成。它们在传输过程中都需要占用 CPU 的时间，而且数据量大时，需要 CPU 很多次访存才能完成。

而 DMA 方式在数据传输时是不需要占用 CPU 时间的，它对于 CPU 时间的占用则（理论上）不受传输数据量的大小影响。DMA 方式下 I/O 接口与内存传输数据的过程如下：

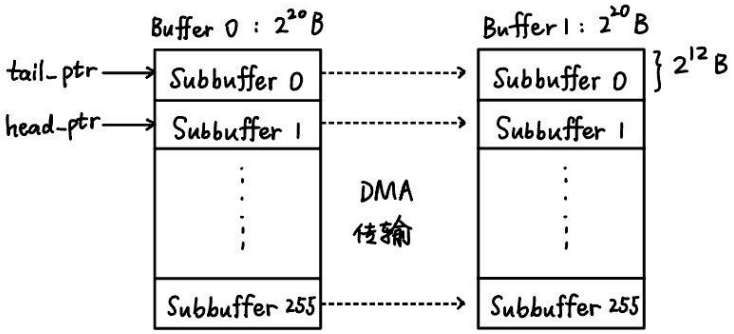
1. 预处理。CPU 将目标起始地址、传输数据长度、I/O 设备编号等信息发送给 DMA 控制器；
2. 数据传送。DMA 控制器再向 CPU 发总线请求，CPU **让出总线**后，由 DMA 控制器控制总线进行外设和主存的数据传输，无需 CPU 干涉。
3. 后处理。传输完毕后，由 DMA 控制器向 CPU 发出 DMA 中断，通知其进行 DMA 后处理。

实际上，DMA 传输方式不仅可以用于 I/O 与内存的数据传输，它也可以用于从内存到内存的数据搬移。下图为 DMA 控制器在内存数据搬移时的工作原理。

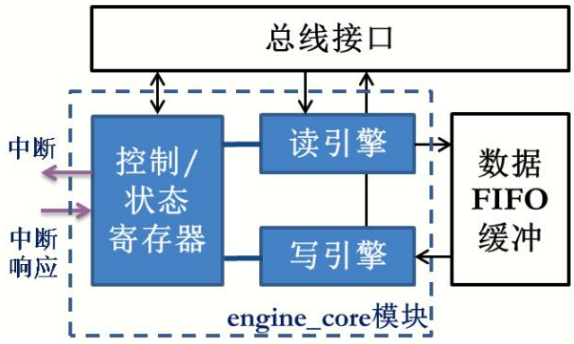


本次实验中我们要实现的 DMA 引擎就是要实现内存数据的搬移。测试程序要利用 DMA 引擎将内存中的数据块 Buffer0 搬到 Buffer1 处。Buffer0 处的数据量为 2^20B。将 Buffer0 划分成 256 个子缓冲区（Sub-buffer），每个子缓冲区的大小为 2^12B（4KB），CPU 给 DMA 引擎分配的一次读写量（dma_size）就是一个子缓冲区的大小。对于每一个子缓冲区，CPU 要先填入数据，并将起始地址、数据长度等信息传给 DMA 引擎（预处理），然后 DMA 引擎才能开始搬运。

这些子缓冲区可以进一步抽象为一个队列，并使用头尾指针来控制队列。CPU 每填充完一个子缓冲区的数据，头指针就“+1”（加上 dma_size），DMA 引擎每搬移完一个子缓冲区的数据，尾指针就“+1”。当头尾指针相等时，表示队列为空，即 DMA 引擎没有可搬移的数据，DMA 就不工作；只要头指针不等于尾指针，DMA 就会开始搬移尾指针所指向的那一个子缓冲区。



在搬移一个子缓冲区的数据时，DMA 读写引擎是通过多次的突发传输向内存读写数据的。其中，读写引擎分别负责从内存中读数据和向内存中写数据。同时 DMA 引擎中还有一个 FIFO 缓冲器，用来暂存从内存中读出来的数据。其结构图在课件上已经画出来了，如下图。



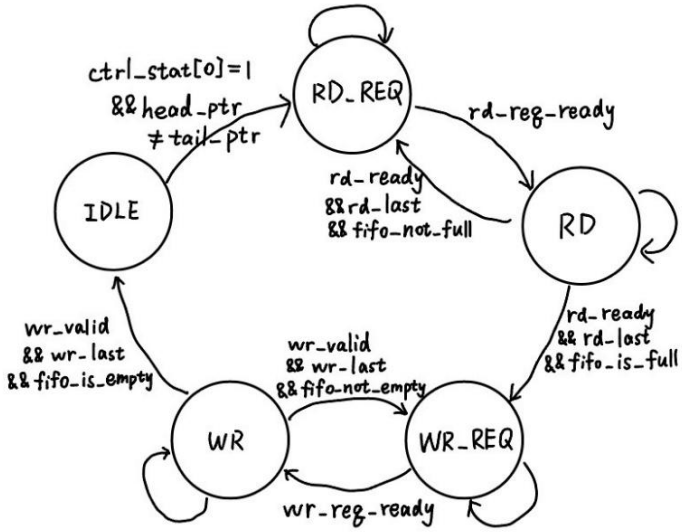
一次突发传输（burst）中，DMA 引擎首先需要向内存发送读/写请求信号（req_ready），等待内存的 valid 信号与之握手，同时 DAM 引擎还要将读/写的起始地址与读/写长度传给内存。握手成功后，便开始数据的传输，而且在传输过程中，每一个数据的传输都需要一次 ready 和 valid 的握手。在最后一个数据传输时还需要拉高 last 信号（内存读时由内存拉高）。

每一个子缓冲区搬移完成后，DMA 引擎需要向 CPU 发送中断请求，触发 CPU 的中断，进入后处理阶段。此时 CPU 需要运行 I/O 中断服务程序，标记搬移完成的子缓冲区（将 dma_buf_stat 减一）。结束中断服务程序后，通过 eret 指令返回中断现场，继续执行中断前的程序。

三．实验过程

1. DAM 引擎的设计

DMA 与内存的数据交换我没有采用读、写同时进行的方案，而是读、写轮流进行。因此，读写引擎的状态控制只要用一个状态机就可以控制了。下面是我的 DMA 引擎状态机的状态转移图。



当头尾指针不相等，并且 DMA 的使能信号拉高时，DMA 离开 IDLE 状态开始工作。DMA 读引擎首先开始，进入 RD_REQ 状态发送读内存请求，然后进入 RD 状态读内存，并将读到的数据存在 fifo 中；一次突发读完成后，回到 RD_REQ 状态再开始下一次突发读……直到 fifo 被写满。fifo 被写满后，写引擎开始工作，进入 WR_REQ 状态发送内存写请求，然后进入 WR 状态，读出 fifo 中的数据写入内存；一次突发写完成后，回到 WR_REQ 状态再开始下一次突发写……直到 fifo 被读空。fifo 被读空后，DMA 引擎回到 IDLE 状态，开始下一次读写循环。

因此，DMA 引擎状态循环一次读写的数据量就是 fifo 的空间大小。实验中写好的 fifo 大小为 256B，因此要搬移一个子缓冲区（4KB），就要循环读写 16 次。DMA 引擎的状态机实现如下所示：

```
// dma读写引擎状态机
always @(posedge clk)begin
    if(rst == 1'b1)
        current_state <= IDLE;
    else
        current_state <= next_state;
end

always @(*)begin
    if(rst == 1'b1)
        next_state = IDLE;
    else begin
        case(current_state)
        IDLE:begin
            if((head_ptr_reg != tail_ptr_reg) && (ctrl_stat_reg[0] == 1'b1))begin
                next_state = RD_REQ;
            end
            else begin
                next_state = IDLE;
            end
            end
        RD_REQ:begin
            if(rd_req_ready)begin
                next_state = RD;
            end
            else begin
                next_state = RD_REQ;
            end
            end
```

```
        RD:begin
            if(rd_ready && rd_valid && rd_last && (~fifo_is_full))begin
                next_state = RD_REQ;
            end
            else if(rd_ready && rd_valid && rd_last && fifo_is_full)begin
                next_state = WR_REQ;
            end
            else begin
                next_state = RD;
            end
            end
        WR_REQ:begin
            if(wr_req_ready)begin
                next_state = WR;
            end
            else begin
                next_state = WR_REQ;
            end
            end
        WR:begin
            if(wr_valid && wr_ready && wr_last && (~fifo_is_empty))begin
                next_state = WR_REQ;
            end
            else if(wr_valid && wr_ready && wr_last && fifo_is_empty)begin
                next_state = IDLE;
            end
            else begin
                next_state = WR;
            end
            end
        default:begin
            next_state = current_state;
            end
        endcase
    end
end
```

为了判断读写的数据量是否达到一个子缓冲区的大小，还需要设置两个计数器，分别统计读、写的数据量，同时也可以作为每次突发传输时读、写地址的偏移量。当读写的数据量都到达 dma_size 时，就将状态寄存器中的中断请求位拉高，触发 CPU 中断。计数器的实现如下：

```
reg [31:0] rd_counter; // dma读引擎传输数据量统计,以字节为单位
reg        rd_finished; // dma子缓冲区读内存完成标志
always @(posedge clk)begin
    if(rst == 1'b1)begin
        rd_counter <= 32'b0;
    end
    else if(rd_ready && rd_valid && rd_last)begin
        rd_counter <= rd_counter + 32'b100000;// 每次dma突发读完成后rd_counter += 32
    end
    else if(rd_counter == dma_size_reg)begin
        rd_counter <= 32'b0;
    end
end
```

```
always @(posedge clk)begin
    if(rst == 1'b1)begin
        rd_finished <= 1'b0;
    end
    else if(rd_counter == dma_size_reg)begin
        rd_finished <= 1'b1;
    end
    else if(subbuffer_finished)begin
        rd_finished <= 1'b0;
    end
end

reg [31:0] wr_counter;  // dma写引擎传输数据量统计,以字节为单位
reg        wr_finished; // dma子缓冲区写内存完成标志

always @(posedge clk)begin
    if(rst == 1'b1)begin
        wr_counter <= 32'b0;
    end
    else if(wr_valid && wr_ready && wr_last)begin
        wr_counter <= wr_counter + 32'b100000;// 每次dma突发写完成后wr_counter += 32
    end
    else if(wr_counter == dma_size_reg)begin
        wr_counter <= 32'b0;
    end
end

always @(posedge clk)begin
    if(rst == 1'b1)begin
        wr_finished <= 1'b0;
    end
    else if(wr_counter == dma_size_reg)begin
        wr_finished <= 1'b1;
    end
    else if(subbuffer_finished)begin
        wr_finished <= 1'b0;
    end
end

wire subbuffer_finished = rd_finished && wr_finished;
```

因为 CPU 向 DMA 发送的信息（如 buffer0 的基地址 src_base，buffer1 的基地址 dest_base，头尾指针 head_ptr，tail_ptr 等）并不一定能在 DMA 引擎工作时始终保持，所以 DMA 中最好还是需要用寄存器及时存下这些输入，因为这些寄存器的值是 CPU 和 DMA 控制器都可以读写的，所以它们的更新需要同时考虑 CPU 和 DMA 控制器，如下图所示：

```
reg [31:0] src_base_reg;
reg [31:0] dest_base_reg;
reg [31:0] head_ptr_reg;
reg [31:0] tail_ptr_reg;
reg [31:0] dma_size_reg;
reg [31:0] ctrl_stat_reg;
```

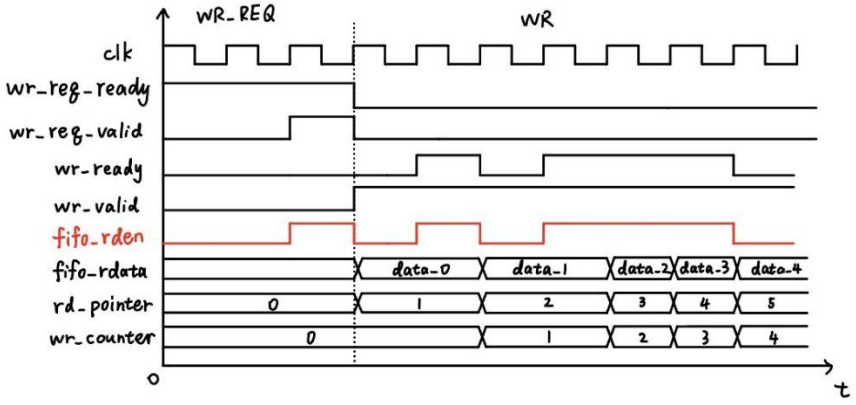
```
// CPU 和 dma 对I/O寄存器的赋值
always @(posedge clk)begin
    if(rst == 1'b1)begin
        src_base_reg <= 32'b0;
        dest_base_reg <= 32'b0;
        head_ptr_reg <= 32'b0;
        tail_ptr_reg <= 32'b0;
        dma_size_reg <= 32'b1;
        // dma_size的初值不能取为0，否则初始时会导致subbuffer_finished=1，直接开始中断
        ctrl_stat_reg <= 32'b0;
    end
    else if(subbuffer_finished)begin
        tail_ptr_reg <= tail_ptr_reg + dma_size_reg; // 尾指针+1
        ctrl_stat_reg[31] <= 1'b1;                   // 中断位拉高
    end
    else begin
        case(reg_wr_en)
        6'b000001:begin
            src_base_reg <= reg_wr_data;
        end
        6'b000010:begin
            dest_base_reg <= reg_wr_data;
        end
        6'b000100:begin
            tail_ptr_reg <= reg_wr_data;
        end
        6'b001000:begin
            head_ptr_reg <= reg_wr_data;
        end
        6'b010000:begin
            dma_size_reg <= reg_wr_data;
        end
        6'b100000:begin
            ctrl_stat_reg <= reg_wr_data;
        end
        endcase
    end
end
```

这里还有一个值得注意的地方是 fifo 读使能信号的赋值。因为 fifo 的读写都是时序控制的，fifo_rden 每拉高一个时钟周期，fifo 中的读指针 rd_pointer 就会+1，同时会将 fifo_rdata 更新为读指针更新前指向的数据（注意时序逻辑使用的是非阻塞赋值）。因此为了在 wr_ready=1 时的时钟上升沿到来之前就将数据从 fifo 中读出来，需要提前将读使能信号 fifo_rden 拉高，同时更新读数据 fifo_rdata 和读指针 rd_pointer。感觉上就是读使能信号总是需要早一拍拉高，这样在将数据写入内存的时钟上升沿，fifo_rdata 都会更新为下一次写入的数据，而将旧的数据写入内存。

所以读使能信号的赋值需要在内存写请求握手时就第一次拉高，而在最后一个数据握手时不拉高（即 last 信号拉高时，否则一次 burst 写就会读出 9 个数据）。fifo_rden 的赋值如下：

```
assign fifo_rden = ((current_state == WR_REQ) && wr_req_ready) ||
                   ((current_state == WR) && wr_ready && ~wr_last);
```

在仿真时一次突发写的波形示意图如下所示。

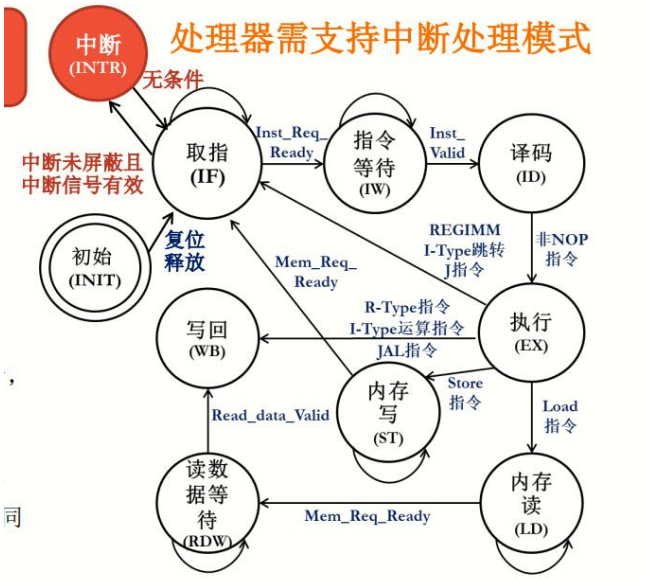


2. CPU 中断处理

我们还需要在 MIPS 多周期处理器中增加对中断的支持。CPU 在每条指令的取指阶段检测是否有中断请求（即 intr 信号是否拉高），如果有中断请求，且没有中断屏蔽，则进入中断状态 INTR。在中断状态需要保存中断现场（即将 PC 存入 EPC 中），然后将中断服务程序的起始地址 0x100 赋给 PC，同时开启中断屏蔽信号（将 intr_mask 置为 0），防止在中断服务程序中继续进入中断。这几个操作可以同时在一个时钟周期内完成，因此中断状态只需要持续一个时钟周期，然后无条件回到取指状态，开始执行中断服务程序。

中断服务程序的最后一条指令为 eret 指令，我们之前实现的 45 条 MIPS 指令中还没有它，所以需要对它增加译码、执行等阶段操作的支持。eret 指令在译码后进入执行阶段。在执行阶段将 EPC 的值（中断现场的 PC）传入 PC，还要将中断屏蔽信号清除（将 intr_mask 置为 1）。执行阶段后，eret 指令像 jump 指令一样直接进入下一次取指阶段。

下面的两张图分别为加入中断状态后 MIPS 处理器的状态转移图，以及 MIPS32 中 eret 指令的指令格式。



31 26	25	24 6	5 0
COP0	CO	0	ERET
010000	1	000 0000 0000 0000 0000	011000
6	1	19	6

对 eret 指令的译码只要看它的 opcode 即可，之前实现的所有指令的 opcode[4]都是 0。

```
wire ERET               = opcode[4];
```

新加入的 EPC 寄存器和中断屏蔽 intr_mask 寄存器的赋值如下。

```
// EPC 寄存器的赋值
```

```
    always @(posedge clk)begin
        if(current_state == INIT)
            EPC_reg <= 32'b0;
        else if(current_state == INTR)
            EPC_reg <= PC_reg;
    end
// intr_mask 寄存器的赋值
    always @(posedge clk)begin
        if(current_state == INIT)
            intr_mask <= 1'b1;
        else if(current_state == INTR)
            intr_mask <= 1'b0;
        else if(ERET)
            intr_mask <= 1'b1;
    end
```

3. MIPS 处理器中断服务程序

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；

因为在 DMA 控制器中，DMA 控制器向 CPU 发出的中断信号的赋值就来自于 ctrl_stat 的中断标志位，因此将此位清 0 后中断请求也会停止。所谓“标记已完成的子缓冲区”，就是计算出两次相邻中断响应之间，DMA 引擎完成处理的子缓冲区数（实际上在此次实验中大概率是 1，因为每一个子缓冲区搬移完成都会触发中断），然后将 dma_buf_stat 变量减去计算出的子缓冲区数。这个 dma_buf_stat 记录的就是当前等待 DMA 处理的子缓冲区个数。

计算两次相邻中断相应之间的子缓冲区个数，可以通过一个 last_tail_ptr 变量记录上一次中断时尾指针的大小，然后子缓冲区个数就等于(tail_ptr - last_tail_ptr)/dma_size。

中断服务程序需用 MIPS 汇编语言来写，写起来还是有点门槛的……

```
    lw  $k1, 0($k0)                 #将ctrl_stat寄存器的值加载到k1寄存器中
    li  $k0, 0x7fffffff             #将011...11存入k0寄存器中
    and $k1, $k1, $k0               #将k0中的值与k1中的值按位与，存入k1中
    li  $k0, 0x60020014             #将ctrl_stat寄存器的地址加载到k0中
    sw  $k1, 0($k0)                 #将k0寄存器中的值存入ctrl_stat寄存器中，至此完成了将ctrl_stat寄存器INTR标志位清0

cycle:
    li  $k0, 0x60020008             #将tail_ptr寄存器的地址加载到k0中
    lw  $k0, 0($k0)                 #将tail_ptr寄存器的值加载到k0寄存器中

    la  $k1, last_tail_ptr          #将last_tail_ptr的地址存入k1寄存器中
    lw  $k1, 0($k1)                 #将last_tail_ptr的值存入k1寄存器中

    beq $k1, $k0, last              #比较tail_ptr-last_tail_ptr的值，如果相等，则退出循环

    li  $k0, 0x60020010             #将dma_size寄存器的地址加载到k0中
    lw  $k0, 0($k0)                 #将dma_size寄存器的值加载到k0中

    addu $k1, $k0, $k1              #将last_tail_ptr+dma_size，存入k1寄存器中
    la  $k0, last_tail_ptr          #加载last_tail_ptr的地址
    sw  $k1, 0($k0)

    la  $k0, dma_buf_stat           #加载dma_buf_stat的地址
    lw  $k1, 0($k0)                 #加载dma_buf_stat的值
    addiu $k1, $k1, -1              #将k1的值减1            #这里用addi反而会出错（变成+1），很奇怪！
    sw  $k1, 0($k0)                 #将k1的值移到dma_buf_stat中

    j   cycle                       #跳转回到循环开始

last:
    eret
```

四．实验总结

1. 性能分析

在 fpga_run 阶段的测试中，用两种方式跑 data_mover 程序，一种是不使用 DMA，一种是使用 DMA。实际跑下来，不使用 DMA 时用时 2493.87ms，使用 DMA 时用时 1242.65ms。使用 DMA 的用时差不多是不使用 DMA 的一半，还要考虑到程序中除了搬移数据外其他辅助指令的影响（如打印语句 printf），DMA 对于内存数据搬移速度的提升还是很显著的（>=2 倍）。

2. 调试过程总结

其实我写下来感觉 DMA 引擎的设计相比于 cache 是更清晰更简单的。它的状态数只有 5 个，在状态控制上就要比 dcache 的 13 个状态好上不少。写的过程中思路也算是比较清晰流畅的，先写状态机，再写寄存器时序更新，最后是输入输出信号的组合逻辑赋值。反而是中断服务的汇编程序我不太会写，因为没有接触过汇编，需要查一些 MIPS 汇编指令，然后再将程序的功能拆解成一条条汇编指令。虽然要实现的中断服务非常简单，但就是这么简单的中断服务，写成汇编语言也用了 22 条汇编指令，中间还弄错了好多次。

这个实验调试过程中最让人头疼的就是出现超时。我跑的大多数 pipeline 在最后一个阶段都会超时，而且超时之后没有任何信息可以查看，所以一开始就毫无头绪，根本不知道哪里出错。后来我尝试了 fpga 加速仿真，以及打印程序执行时的中间过程等方法，才查出了那几个错误。一个是 fifo 写使能信号要超前拉高的问题，还有几个基本上就是中断信号的问题，中断信号一直没有拉高或者一直被拉高，dma_buf_stat 的值没减下去，都会导致程序一直在等待 DMA 传输完成，最终超时。

我还有一个没弄明白的问题是 dma_buf_stat – 1 操作时，如果使用 addi 指令，dma_buf_stat 反而会加 1，而使用 addiu 指令却能够正常减 1。但问题是-1 是带符号数，它适用的不应该是 addi 指令吗？addiu 指令为什么可以将立即数置为-1？而更奇怪的是，addi 指令反而会导致 dma_buf_stat+1。我不太了解汇编，这个问题在网上查了一下也没有查到太多有用的信息。希望之后学了汇编语言，再来看这里的时候能够理解吧阿巴阿巴。

最后，我的 DMA 引擎没有采用内存读写同时进行的方案。我们的内存读写的数据接口是分开的，因此内存应该是支持同时读写的。我觉得如果使用读写引擎同时工作的话（只要 fifo 不空，写引擎就开始向内存写；只要 fifo 不满，读引擎就开始内存读），DMA 的搬移速度就能做到更快，性能也应该会更好，之后有时间可以尝试实现一下。