中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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。

1 电路设计及说明

本次实验的核心任务是设计并实现一个复杂的 I/O 外设——直接内存访问(DMA)控制器,并对自行设计的多周期 MIPS CPU 进行改造,使其能够支持外部中断,最终实现 CPU 与 DMA 引擎的高效协同工作。实验主要分为三个部分:DMA 引擎核心(`engine_core.v`)的设计、MIPS 中断服务程序(`intr_handler.S`)的编写,以及对 CPU(`custom_cpu.v`)的硬件改造。

1.1 DMA 引擎核心 (engine_core) 设计

DMA 引擎的设计旨在将 CPU 从繁重的内存数据搬移任务中解放出来。本次设计的 DMA 引擎基于队列结构, 遵循生产者-消费者模型。CPU 作为生产者, 通过更新 `head_ptr` 指针来提交数据搬移任务 (子缓冲区); DMA 引擎作为消费者,通过更新 `tail_ptr` 指针来完成任务。DMA 引擎的核心功能是,当检测到 `head_ptr != tail_ptr` 时, 自动从源地址(`src_base + tail_ptr`)读取数据, 并写入目标地址(`dest_base + tail_ptr`), 完成一个大小为 `dma_size` 的数据块搬运后,通过中断通知 CPU。

为实现高效的并发读写,DMA 引擎内部设计了两个独立的状态机(FSM)分别控制读和写操作,并通过一个内部 FIFO 进行解耦。

1.1.1 控制与状态寄存器 (CSR) 单元

CSR 是 CPU 与 DMA 引擎交互的接口,通过内存映射 I/O 方式访问。我设计了 6 个 32 位寄存器,其更新逻辑在一个 `always` 块中统一管理, CPU 的写请求具有最高优先级, 其次是 DMA 硬件在任务完成时的自动更新。

Listing 1: DMA 控制与状态寄存器(CSR)更新逻辑

```
always @(posedge clk) begin
    if (rst) begin
        // Reset all registers to their initial states.
        src_base_reg <= 32'h0;
        dest_base_reg <= 32'h0;
        tail_ptr_reg <= 32'h0;
```

```
        head_ptr_reg <= 32'h0;
        dma_size_reg <= 32'h0;
        ctrl_stat_reg <= 32'h1; // Enable DMA by default.
    end else if (|reg_wr_en[5:0]) begin
        // CPU write request takes precedence.
        if (reg_wr_en[0]) src_base_reg <= reg_wr_data;
        if (reg_wr_en[1]) dest_base_reg <= reg_wr_data;
        if (reg_wr_en[2]) tail_ptr_reg <= reg_wr_data; // CPU can set initial tail.
        if (reg_wr_en[3]) head_ptr_reg <= reg_wr_data; // CPU signals new data.
        if (reg_wr_en[4]) dma_size_reg <= reg_wr_data;
        if (reg_wr_en[5]) ctrl_stat_reg <= reg_wr_data; // CPU can clear interrupt.
    end else if ((current_state_WR == WR_REQ) && wr_complete) begin
        // DMA hardware automatically updates upon task completion.
        ctrl_stat_reg <= {1'b1, ctrl_stat_reg[30:0]}; // Set interrupt flag.
        tail_ptr_reg <= tail_ptr_reg + dma_size_reg; // Advance tail pointer.
        // ... retain other registers
    end else begin
        // ... retain all registers
    end
end
```

上述代码清晰地定义了寄存器更新的优先级:复位 > CPU 写 > DMA 硬件自动更新 > 保持。特别地,在写任务完成 (wr_complete) 且状态机处于 WR_REQ 时,硬件会自动更新 tail_ptr 并置位中断标志。

1.1.2 双状态机 (Dual FSM) 控制逻辑

我采用了读、写分离的双状态机设计，它们分别管理与内存的读、写交互。两个状态机都包含 IDLE、REQ (请求)、OP (操作) 三个核心状态。次态逻辑在一个组合逻辑块 always @(*) 中实现，并通过块首的默认赋值 next_state = current_state 来防止意外生成锁存器。

Listing 2: 读状态机(FSM-RD)次态逻辑

```
always @(*) begin
    next_state_RD = current_state_RD; // Default assignment to prevent latches.

    case (current_state_RD)
        IDLE: begin
            if (EN && (tail_ptr != head_ptr) && (current_state_WR == IDLE)) begin
                next_state_RD = RD_REQ; // Start read if task is available.
            end
        end
        RD_REQ: begin
            if (rd_req_ready && rd_req_valid) begin
                next_state_RD = RD;  // Move to RD state after request is accepted.
            end else if (rd_complete) begin
                next_state_RD = IDLE; // Return to IDLE if read task is finished.
            end
        end
        RD: begin
            if (rd_valid && rd_last && !fifo_is_full) begin
                next_state_RD = RD_REQ; // Go for next burst after current one finishes.
```

```
            end
        end
        default: next_state_RD = IDLE;
    endcase
    // ... Write FSM logic is similar
end
```

两个状态机之间存在一定的耦合(例如启动时互相检查对方是否为 IDLE),这是一种保守的设计,确保了启动序列的稳定。

1.1.3 Burst 传输与任务进度跟踪

为了将一个大的 dma_size 任务分解成多个小的 Burst 传输,我设计了 rd_counter 和 wr_counter 来跟踪已完成的 Burst 数量。这两个计数器只在一个新任务开始的瞬间被清零,并在每次 Burst 成功完成后递增,保证了任务进度的正确跟踪。

Listing 3: Burst 计数器更新逻辑

```
// A signal indicating a new task should start.
wire start_new_task = (current_state_RD == IDLE) && (current_state_WR == IDLE) && EN && (head_ptr
    != tail_ptr);

always @(posedge clk) begin
    if (rst) begin
        rd_counter <= 28'h0;
        wr_counter <= 28'h0;
    end else if (start_new_task) begin
        // Reset counters only at the very beginning of a new task.
        rd_counter <= 28'h0;
        wr_counter <= 28'h0;
    end else begin
        // Increment on successful completion of a read burst.
        if (current_state_RD == RD && rd_valid && rd_last && !fifo_is_full && rd_ready) begin
            rd_counter <= rd_counter + 1;
        end
        // Increment on successful completion of a write burst.
        if (current_state_WR == WR && wr_valid && wr_last && wr_ready) begin
            wr_counter <= wr_counter + 1;
        end
    end
end
```

同时,我实现了对非 32 字节对齐的尾部数据的处理逻辑,通过计算 total_burst_num 和 last_burst_len 来动态调整最后一次 Burst 的传输长度。

1.2 MIPS CPU 中断支持改造

为使 CPU 能够响应 DMA 的中断请求,我对其进行了五个关键部分的改造:

1. **增加 INTR 状态**: 在 FSM 中添加了 INTR 状态,并修改了 IF 状态的跳转逻辑,使其在接收到中断信号 intr 且中断未被屏蔽时,能够跳转到 INTR 状态。

2. **实现 EPC 寄存器**: 添加了 32 位的异常程序计数器 `EPC`,用于在进入 `INTR` 状态时保存当前的 PC 值,作为中断返回的地址。

Listing 4: EPC 寄存器锁存逻辑

```
// -- EPC Register Logic --
reg [31:0] EPC;     // Exception Program Counter (EPC) register
always @(posedge clk) begin
    if (rst) begin
        EPC <= 32'b0;    // Reset EPC to 0
    end else if (current_state == INTR) begin
        EPC <= current_pc; // Store current PC in EPC on interrupt
    end else begin
        EPC <= EPC;      // Keep EPC unchanged otherwise
    end
end
```

3. **实现中断屏蔽逻辑**: 添加了 `intr_mask` 寄存器。当 CPU 进入 `INTR` 状态时,该位置 1,屏蔽后续中断;当 CPU 在 `ID` 阶段译码到 `ERET` 指令时,该位被清零,重新使能中断。

Listing 5: 中断屏蔽(Interrupt Mask)逻辑

```
// -- interrupt mask logic --
always @(posedge clk) begin
    if (rst || (current_state == ID && is_eret)) begin
        intr_mask <= 1'b0; // Enable interrupts on reset or ERET.
    end else if (current_state == INTR) begin
        intr_mask <= 1'b1; // Disable interrupts during interrupt handling.
    end else begin
        intr_mask <= intr_mask; // Retain current mask state.
    end
end
```

4. **修改 PC 更新逻辑**: PC 的更新逻辑被修改以处理两种特殊跳转。首先,在 `pc` 模块内部,当检测到 CPU 进入 `INTR` 状态时,PC 被强制设置为中断服务程序的入口地址 `0x100`。其次,修改了 `next_pc` 的计算逻辑,使其在检测到 `ERET` 指令时,选择 `EPC` 作为下一个 PC 值。

5. **修改 PC 写使能**: 为了让 `ERET` 指令能够成功触发 PC 更新,我扩展了 `pc_write_enable` 信号的生成逻辑,加入了 `((current_state == ID) && is_eret)` 这一条件,确保了从中断返回的路径是通畅的。

1.3 中断服务程序 (`intr_handler.S`) 实现

我编写了一个 MIPS 汇编中断服务程序,严格按照讲义的要求,分为三个部分:

1. **响应中断**: 首先,通过读-改-写操作,将 DMA 的 `ctrl_stat` 寄存器(地址 `0x60020014`)中的中断标志位(bit 31)清零,以防止 CPU 重复进入中断。

2. **计算增量与更新指针**: 读取 DMA 硬件更新后的 `tail_ptr`,并与软件保存的上一次的 `last_tail_ptr` 作差,得到本次中断期间处理的数据字节数 `delta`。然后,将新的 `tail_ptr` 写回 `last_tail_ptr` 变量,为下一次中断做准备。

3. **更新软件状态**: 根据 `delta` 和 `dma_size`,通过一个循环,用减法实现除法,计算出本次处理掉的子缓冲区数量,并相应地递减全局变量 `dma_buf_stat` 的值。

最后,程序通过 `eret` 指令从中断中返回。整个程序只使用了 `$k0` 和 `$k1` 两个内核保留寄存器,避免了保护和恢复通用寄存器的开销。

Listing 6: MIPS 中断服务程序核心逻辑

```
intr_handler:
    # Part 1: Clear interrupt flag in ctrl_stat register.
    lui  $k0, 0x6002
    lw   $k0, 0x14($k0)
    lui  $k1, 0x7FFF
    ori  $k1, $k1, 0xFFFF
    and  $k1, $k0, $k1
    lui  $k0, 0x6002
    sw   $k1, 0x14($k0)

    # Part 2: Calculate processed bytes (delta) and update last_tail_ptr.
    lw   $k1, 0x08($k0)    # k1 = new_tail_ptr
    lw   $k0, last_tail_ptr # k0 = old_tail_ptr (pseudo-instruction)
    sub  $k0, $k1, $k0     # k0 = delta
    sw   $k1, last_tail_ptr # Update last_tail_ptr (pseudo-instruction)
    blez $k0, L2           # Skip if no data processed.
    NOP

L1:
    # Part 3: Decrement dma_buf_stat based on delta and dma_size.
    # Assumption: dma_buf_stat is at address 0x10.
    lw   $k1, 0x10($0)     # k1 = dma_buf_stat
    addi $k1, $k1, -1
    sw   $k1, 0x10($0)
    lui  $k1, 0x6002
    lw   $k1, 0x10($k1)    # k1 = dma_size
    sub  $k0, $k0, $k1     # delta -= dma_size
    bgtz $k0, L1           # Loop if more buffers were processed.
    NOP

L2:
    eret                   # Return from interrupt.
```

2 实验过程中遇到的问题、对问题的思考过程及解决方法

在本次 DMA 与中断的综合实验中,我遇到了诸多涉及硬件、软件和两者交互的复杂问题。调试过程不仅是对我设计能力的考验,更是一次宝贵的系统级问题排查经验。

1. **问题:FIFO 数据的"单周期有效"特性导致的数据丢失**

   - **现象与思考:** 我最初试图直接将 FIFO 的输出 `fifo_rdata` 连接到写引擎的数据端口 `wr_data`。但我很快意识到,标准的同步 FIFO 在读使能 `fifo_rden` 拉高后的下一个周期,其输出数据仅有效一

拍。如果此时内存接口的 `wr_ready` 信号为低，即发生 stall，那么这一拍的数据就会丢失，因为无法在下一拍重新从 FIFO 中读出相同的数据。

- **解决方法：**为了解决这个问题，我设计并实现了一个 1 级深度的“skid buffer”。该缓冲由一个数据寄存器 `wr_data_reg` 和一个有效位寄存器 `wr_valid_reg` 构成。通过引入一个延迟一拍的 `last_fifo_rden` 信号，我得以在 `fifo_rdata` 有效的那个精确周期，将其锁存到 `wr_data_reg` 中。最终的 `wr_data` 输出由一个多路选择器决定：如果当前有新的数据从 FIFO 流出，则直接转发；否则，从 `wr_data_reg` 中取出之前保存的数据进行“重播”。这个设计确保了在任何 stall 情况下，写数据都不会丢失。

Listing 7: Skid Buffer 关键实现逻辑

```
// A 1-cycle delayed version of fifo_rden to sample fifo_rdata correctly.
always @(posedge clk) begin
    last_fifo_rden <= fifo_rden;
end

// Register to hold the data word during a stall.
always @(posedge clk) begin
    if (last_fifo_rden) begin
        wr_data_reg <= fifo_rdata; // Latch data on the cycle it becomes valid.
    end
end

// Data multiplexer: Choose fresh data or stalled data.
assign wr_data = (last_fifo_rden) ? fifo_rdata : wr_data_reg;
```

2. **问题：DMA 状态机死锁——取指请求与中断处理的冲突**

- **现象：**在初次集成后，运行测试程序时，系统在启动 DMA 后很快就卡死，并报告超时。输出日志如下：

```
time 1000994.20ms
custom cpu running time out
./software/workload/ucas-cod/benchmark/simple_test/dma_test/mips/elf/data_mover_dma failed
Hit bad trap
```

- **思考与分析：**通过仔细分析 FSM 的跳转逻辑，我定位到了一个严重的设计缺陷。我的 CPU 在 `IF` 状态会无条件地发出取指请求信号 `Inst_Req_Valid`。当中断信号 `intr` 在 `IF` 阶段到来时，状态机会决定下一周期跳转到 `INTR` 状态。然而，在当前这个 `IF` 周期，取指请求已经发出去了，但内存尚未响应。CPU 进入 `INTR` 状态后，会修改 PC 到 `0x100` 并跳转回 `IF` 状态，试图获取中断服务程序的第一条指令。但此时，内存接口仍然在等待处理上一个（被打断的）取指请求，无法响应新的请求，导致 CPU 永远卡在等待 `0x100` 处指令的状态，造成死锁。
- **解决方法：**必须阻止在即将进入中断时发出那个多余的取指请求。我修改了 `Inst_Req_Valid` 的生成逻辑，为其增加了一个条件：只有在当前不准备响应中断时，取指请求才有效。

Listing 8: 修正后的 Inst_Req_Valid 逻辑避免死锁

```
// Original problematic logic: assign Inst_Req_Valid = (current_state == IF);
// Corrected logic:
assign Inst_Req_Valid = (current_state == IF) && !(intr && !intr_mask);
```

这个修改确保了在 CPU 决定要处理中断的那个周期，不会向内存系统发出任何新的取指请求，从而避免了握手协议的冲突和死锁。

3. **问题：对复杂组合逻辑信号的理解偏差**

   - **思考：**在设计过程中，我对一些复杂信号的生成逻辑，如 `fifo_rden`，最初的理解不够深入。我曾尝试简化其逻辑，但发现这会破坏“skid buffer”的正常工作。例如，`fifo_rden` 的拉高时机，不仅与写请求的启动有关，还与写操作过程中 `wr_valid` 和 `wr_ready` 的握手状态紧密相连，以实现背靠背的数据流。
   - **体会：**这让我深刻体会到，在复杂的数字系统中，每一个控制信号的生成都必须经过严密的逻辑推演，考虑所有的边界条件和状态组合。一个看似微小的改动，都可能在某些特定时序下引发意想不到的错误。编写清晰、健壮且意图明确的控制逻辑，比追求最简化的表达式更为重要。

3 实验结果

3.1 仿真结果与性能对比

在修复了上述关键问题后，我的 DMA 引擎和支持中断的 MIPS CPU 成功通过了所有测试。为了验证 DMA 的有效性，我对比了使用 DMA（`data_mover_dma`）和不使用 DMA（即纯 CPU 内存拷贝，`data_mover_no_dma`）两种方式下，完成相同数据搬移任务的性能。

线上 FPGA 评测系统的输出结果清晰地展示了性能差异：

Listing 9: 使用 DMA 与纯 CPU 拷贝的性能对比结果

```
# --- Test with DMA enabled ---
RUNNER_CNT = 1
...
Prepare DMA engine
Prepare SW data mover
benchmark finished
time 765.41ms
...
./software/workload/ucas-cod/benchmark/simple_test/dma_test/mips/elf/data_mover_dma passed
Hit good trap
pass 1 / 1
Job succeeded

# --- Test with DMA disabled (CPU copy) ---
RUNNER_CNT = 2
...
Prepare SW data mover
benchmark finished
time 1883.84ms
...
./software/workload/ucas-cod/benchmark/simple_test/dma_test/mips/elf/data_mover_no_dma passed
Hit good trap
pass 1 / 1
Job succeeded
```

从结果可以看出，使用 DMA 引擎完成数据搬移耗时约 **765.41ms**，而完全由 CPU 执行相同任务耗时约 **1883.84ms**。通过硬件 DMA 加速，性能提升了约 **2.46 倍**。这一显著提升证明了 DMA 设计的正确性和其在分担 CPU 访存任务、提高系统并行度方面的巨大价值。

4 对讲义中思考题的理解和回答

本实验讲义中未明确指定思考题，但结合 DMA 引擎的设计实践，以及计组理论课所讲解的一些内容，可以思考以下相关问题：

1. **为何 DMA 需要通过中断与 CPU 交互，而不是让 CPU 轮询 DMA 的状态？**

   - 让 CPU 通过轮询（即反复读取 DMA 的 `ctrl_stat` 寄存器）来检查任务是否完成，虽然在逻辑上可行，但效率极其低下。在 DMA 进行数据搬运的漫长时间里，CPU 将把所有时间都浪费在一个紧密的“读-比较-跳转”循环中，无法执行任何其他有用的计算任务。这完全违背了使用 DMA 将 CPU 解放出来的初衷。
   - 中断机制则是一种“事件驱动”的模式。CPU 可以启动 DMA 后，立即切换去执行其他程序。只有当 DMA 这个“事件”真正完成时，它才会通过中断信号来“打断”CPU，请求 CPU 进行后续处理（如准备下一个 DMA 任务）。这种方式极大地提高了系统的并行性和 CPU 的利用率，是高效操作系统和驱动程序工作的基石。

2. **中断服务程序(ISR)的设计为何要尽可能快且短小？**

   - ISR 的执行具有两个关键特性：一是它会打断当前正在执行的程序；二是在 ISR 执行期间，通常会屏蔽掉同级或低级的中断，以防止中断嵌套带来的混乱。
   - 如果 ISR 执行时间过长，会导致：1）被中断的程序响应延迟增加，影响系统的实时性。2）长时间的中断屏蔽会使系统无法及时响应其他重要的外部事件，可能导致数据丢失或系统错误。因此，ISR 的设计原则是“快进快出”。它只应该做最核心、最紧急的工作，例如清除中断标志、从硬件读取少量状态、更新关键的软件标志位等。更复杂的、耗时的数据处理应该在 ISR 中设置一个标志，然后交由正常的、非中断的上下文（如主程序的循环或操作系统的任务调度）来完成。

5 实验所耗时间

在课后，我花费了大约 30 小时完成本次 DMA 引擎的设计、CPU 中断改造、软件编写、以及所有问题的调试和报告撰写工作。其中，大部分时间消耗在对 DMA 与 CPU、内存之间复杂时序和握手协议的调试上。

6 实验总结与心得体会

在本次实验中，我成功地设计并实现了一个基于队列的 DMA 硬件引擎，并对已有的 MIPS 处理器进行了改造，使其能够正确响应和处理外部中断。这个过程不仅是一次复杂的硬件设计挑战，更是一次深刻的软硬件协同设计实践。

设计的核心在于 DMA 引擎的实现。我采用了读写双状态机加内部 FIFO 的解耦架构，以实现内存读写操作的流水化和并行化，最大化数据吞吐率。在实现过程中，我遇到了并解决了几个关键的技术难题。其中最具挑战性的是处理内存接口的 `stall` 情况。我意识到 FIFO 输出数据的“单周期有效”特性，设计了“skid buffer”来暂存数据，确保在 `wr_ready` 信号为低时数据不会丢失。此外，我还实现了对非 32 字节对齐传输的支持，通过动态计算最后一个 Burst 的长度，增强了 DMA 的通用性。

对 CPU 的改造同样至关重要。我为其增加了 `INTR` 状态、`EPC` 寄存器以及 `intr_mask` 中断屏蔽逻辑。调试中最关键的突破，是解决了在 `IF` 阶段响应中断时，因发出多余的取指请求而导致的系统死锁问题。通过为取指

有效信号 `Inst_Req_Valid` 增加`!(intr && !intr_mask)` 的条件,我确保了 CPU 在决定处理中断的瞬间,不会与内存发生新的握手,从而保证了状态转换的原子性和正确性。

通过编写 MIPS 中断服务程序,我将硬件行为与软件控制联系了起来。ISR 的设计必须高效且精简,它作为硬件和应用程序之间的桥梁,负责清除中断、更新硬件指针和软件状态。这个过程让我深刻理解了中断驱动 I/O 的工作原理,以及软硬件接口设计的规范和重要性。

最终的性能测试结果令人振奋:使用我设计的 DMA 引擎进行数据搬移,相比纯 CPU 软件拷贝,性能提升了约 2.46 倍。这个量化的结果直观地证明了 DMA 在卸载 CPU 任务、提升系统并行处理能力方面的巨大优势,也验证了我整个设计的正确性和有效性。

总而言之,本次实验让我从一个系统级的视角,完整地经历了一次复杂 I/O 外设的设计、CPU 功能的扩展、以及底层软件驱动的编写。从状态机的时序到软硬件的握手协议,从性能的瓶颈分析到疑难 BUG 的定位,每一个环节都极大地锻炼了我的硬件设计能力、系统调试能力和对计算机组成原理的深入理解。

7 致谢

感谢宁彦祯同学和宋俊仪同学,他们的真知灼见给了我很大的启发;感谢朱徐塬学长及各位助教,您们的帮助对我有非常大的指导意义。