中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 朱夏楠 学号 2022K8009929031 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 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

• **DMA 控制引擎设计**。DMA 引擎通过读引擎从缓冲区不断读出数据填充 DMA 的队列, 与此同时, 写引擎不断的从队列中读出数据送到内存中相应的位置。读引擎和写引擎何时开始工作, 传输多少数据, 这些则都由状态寄存器来确定。本次实验中 DMA 引擎部分要实现的就是这三个部分。状态寄存器的含义以及何时修改都在讲义上给出, 读引擎的状态转移图也在讲义中给出, 我们需要设计的主要是写引擎的状态。

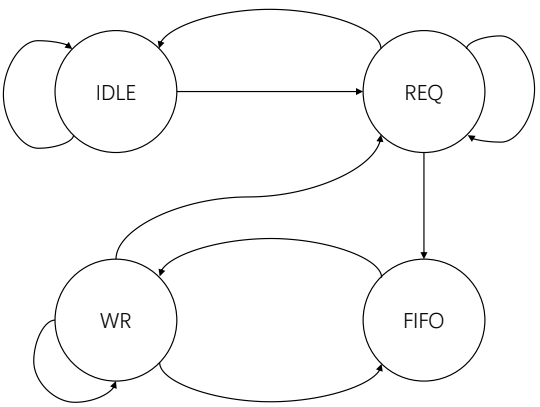


图 1: 写引擎状态转移图

可以看到，与读引擎的状态转移图不同的是，写引擎多出了一个 FIFO 状态，这是由于 DMA 引擎的队列要求先将队列读使能拉高，然后在下一个时钟周期才会将数据传送给写引擎，因此我额外引入了 FIFO 状态，当写请求被内存接收后或者是内存可写数据之后，就从 REQ 或者是 WR 状态转移到 FIFO 状态，在该状态下将队列读使能信号拉高，然后在下一拍无条件跳转到 WR。

写引擎和读引擎实现关键是硬件能够知道什么时候需要进行状态的转换，为此，最关键的是需要计算每一次 DMA 引擎启动需要进行多少次突发传输，以及每次突发传输的长度是多少。我选择使用计数器，记录每一次 DMA 引擎启动所进行的突发读和突发写次数。

```
    /*记录一次DMA引擎启动过程中突发读的次数*/
always @ (posedge clk)
    begin
  if (rst | rd_current_state[0] & wr_current_state[0] & ctrl_stat[0] & ~equal & rd_burst_finish
      & wr_burst_finish)
     rd_cnt <= 32'b0;
  else if (rd_current_state[2] & rd_valid & rd_last)
     rd_cnt <= rd_cnt + 32'b1;
end

    /*记录一次DMA引擎启动过程中突发写的次数*/
always @ (posedge clk)
    begin
    //burst_finish:读/写引擎是否已经完成一次完整的启动
  if(rst | (rd_current_state[0] & wr_current_state[0] & ctrl_stat[0] & ~equal & ~intr &
      rd_burst_finish & wr_burst_finish))
     wr_cnt <= 32'b0;
  else if(wr_current_state[2] & wr_ready & wr_last)
     wr_cnt <= wr_cnt + 32'b1;
end

    /*突发写的过程中已经读的数据数*/
always @ (posedge clk)
    begin
  if(rst | wr_current_state[1])
                wdata_counter <= 3'b0;
  else if(wr_current_state[2] & wr_ready)
                wdata_counter <= wdata_counter + 3'b1;
end

    assign last_burst = dma_size[4:2] + |dma_size[1:0]; //计算最后一次突发传输需要传送的字节数
    //前几次突发传输长度都是8，只有最后一次可能不为8
    assign rd_req_len = ((rd_cnt == total_burst_num - 1) & |last_burst)? {2'b0, (last_burst -
        3'b1)} : 5'b111;
    assign wr_req_len = ((wr_cnt == total_burst_num - 1) & |last_burst)? {2'b0, (last_burst -
        3'b1)} : 5'b111;
```

由此，可以描述读/写引擎的状态机。后续只需要根据不同状态对状态寄存器和控制信号赋值即可

```
    /*RD ENG*/
always @ (posedge clk)
    begin
```

```
   if (rst)
                  rd_current_state <= IDLE;
   else
                  rd_current_state <= rd_next_state;
end

always @ (*)
    begin
   case (rd_current_state)
     IDLE:
                  begin
       if (ctrl_stat[0] & wr_current_state[0] & ~equal & ~rd_burst_finish& & fifo_is_empty)
         rd_next_state = REQ;
       else
                              rd_next_state = IDLE;
     end
     REQ:
                  begin
       if (rd_burst_finish)
                              rd_next_state = IDLE;
       else if(rd_req_ready)
                              rd_next_state = RW;
       else
                              rd_next_state = REQ;
     end
     RW:
                  begin
       if (rd_valid & rd_last & ~fifo_is_full)
                              rd_next_state = REQ;
       else
                              rd_next_state = RW;
     end
     default:
                         rd_next_state = IDLE;
   endcase
end

    /*WR ENG*/
    always @ (posedge clk)
    begin
   if (rst)
                  wr_current_state <= IDLE;
   else
                  wr_current_state <= wr_next_state;
end

always @ (*)
    begin
   case (wr_current_state)
     IDLE:
```

```
                begin
                //比较读传输和写传输的次数，若写小于读，则说明队列中有足够写引擎传输一次的数据
        if (ctrl_stat[0] & ~equal & ~wr_burst_finish & (wr_cnt < rd_cnt))
          wr_next_state = REQ;
        else
                            wr_next_state = IDLE;
      end
      REQ:
                begin
        if (wr_burst_finish)
                            wr_next_state = IDLE;
        else if(wr_req_ready & (wr_cnt < rd_cnt))
                            wr_next_state = FIFO;
        else
                            wr_next_state = REQ;
      end
      FIFO:
                begin
        wr_next_state = RW;
      end
      RW:
                begin
        if (wr_ready & wr_last)
                            wr_next_state = REQ;
        else if (wr_ready & ~fifo_is_empty)
                            wr_next_state = FIFO;
        else
                            wr_next_state = RW;
      end
      default:
                        wr_next_state = IDLE;
    endcase
  end
```

- **MIPS 处理器中断处理设计**。

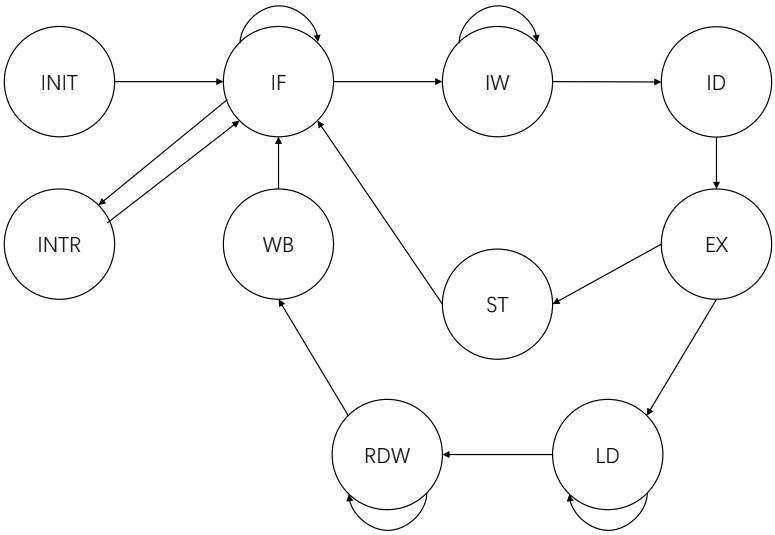


图 2: 添加中断之后的 MIPS 多周期处理器状态转移图

DMA 引擎在完成一次数据搬运之后，会发出一次中断请求，CPU 接收到中断请求之后，需要陷入中断状态，执行中断服务程序。为了实现这个功能，需要在原来的多周期处理器的基础上机型改动，添加一个 INTR 状态，每次处理器返回 INIT 状态，就要检查 DMA 是否发出了中断请求，若有中断请求，且当且不处于中断服务程序中，则进入 INTR，在 INTR 状态，需要保存当前的 PC 值到 EPC 寄存器，并且将 PC 值替换为中断程序入口，同时将 EINT 寄存器置 0，代表进入了中断服务程序，不再处理新的中断请求。

```
assign iseret = opcode[4]; //根据eret指令的操作码特点判断指令是否为eret
EPC epcregister(clk,rst,EPCchange,PC,EPC); 在INTR状态, EPCchange拉高
assign nextpc = (((E_regA & {32{PCsrc[1] & PCsrc[0]}})
    | ({PC[31:28],taddr,2'b0} & {32{PCsrc[1] & ~PCsrc[0]}})
    | (MW_ExcutionResult & {32{~PCsrc[1] & PCsrc[0]}})
    | (addedpc & {32{~PCsrc[1] & ~PCsrc[0]}})) & {32{~iseret & ~EPCchange}})
    | (EPC & {32{iseret}})      //指令为eret, 恢复PC值
    | (32'h100 & {32{EPCchange}}); //在INTR状态将, 中断程序入口地址写入PC
```

同时，处理器还需要能处理一条新的指令 eret，该指令用于跳出中断服务程序，返回 CPU 的正常工作，同时将 EINT 重新置 1。

- **MIPS 汇编中断服务程序设计**。

中断服务程序主要负责更新一个全局变量 dma_buf_stat 的值，每当 CPU 完成一个子缓冲区的填充，就会将该值加 1；而 DMA 每取出一个子缓冲区，就会将其减 1，这个过程就是由处理器执行中断处理程序完成的。主要的思路是根据上一次中断处理程序结束后的缓冲区队列尾指针的值和当前的值比较，两者之间的差值可以说明距离上一次中断请求 DMA 总共搬运了多少个子缓冲区，并以此更新 dma_buf_stat 的值。需要注意的是，程序只能使用 k0 和 k1 两个寄存器，程序结束时要用 eret 返回。

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)

- **DMA 实验缺少错误检测手段。**

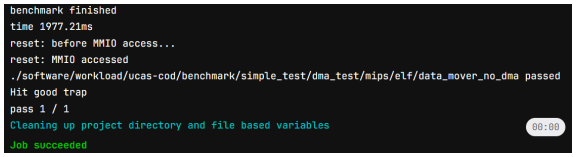
本次 DMA 实验,包括之前的 DNN 实验,都不存在行为仿真阶段,而 FPGA 阶段的仿真加速对于一部分错误很难检查出来。我在修改处理器之后,无法仿真成功,通过 FPGA 仿真加速也无法截取到我想要查看的波形。最终我决定检查处理器本身是否出现问题,在本地进行了一次针对 MIPS 处理器的行为仿真之后,verilator 指出来我在添加了 INTR 状态之后,未将 current_state 和 next_state 寄存器的位数改变。而在云平台的进行代码检查时,虽然发现了这个问题,仍进行到下一阶段,导致我一直没有注意到这个问题。

这给我的教训是在云上仿真时虽然某个阶段出现了错误,但并不代表之前的阶段没有问题,需要仔细的审查每个阶段给出的警告信息。

三、 性能对比

DMA 引擎实验一次性对使用 DMA 引擎和 Load/Store 搬运数据两种方式进行了仿真,根据框架提供的时间统计功能可知,使用 DMA 引擎搬运数据时,完成测试程序花了 860.83ms,而在使用 Load/Store 搬运时,花费了 1977.21ms,可见 DMA 引擎在进行数据块迁移时对程序的提升是比较大的。

(a) 使用 DMA 搬运数据的结果



(b) 不适用 DMA 搬运数据的结果

四、 实验所耗时间

在课后,你花费了大约____20____ 小时完成此次实验。