中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 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

注1：撰写此Word格式实验报告后以PDF格式保存在~/COD-Lab/reports目录下。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. 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}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）
2. engine\_core.v  
    逻辑结构图：   
    

（1）控制/状态等寄存器处理

|  |
| --- |
| //Deal with input from CPU          always @(posedge clk) begin                  if(rst) begin                          src\_base  <= 32'b0;                          dest\_base <= 32'b0;                          tail\_ptr  <= 32'b0;                          head\_ptr  <= 32'b0;                          dma\_size  <= 32'b0;                          ctrl\_stat <= 32'b0;                  end                  else if(EN & WR\_cur\_state[1] & RD\_burst\_cnt == burst\_time & WR\_burst\_cnt == burst\_time)                  begin                          tail\_ptr        <= tail\_ptr +dma\_size;                          ctrl\_stat[31]   <= 1'b1;                  end                  else begin                          if(reg\_wr\_en[0])        src\_base  <= reg\_wr\_data;                          else if(reg\_wr\_en[1])   dest\_base <= reg\_wr\_data;                          else if(reg\_wr\_en[2])   tail\_ptr  <= reg\_wr\_data;                          else if(reg\_wr\_en[3])   head\_ptr  <= reg\_wr\_data;                          else if(reg\_wr\_en[4])   dma\_size  <= reg\_wr\_data;                          else if(reg\_wr\_en[5])   ctrl\_stat <= reg\_wr\_data;                  end          end  //Singals of DMA enable and intr valid          assign intr = ctrl\_stat[31];          assign EN   = ctrl\_stat[0]; |

初始复位后，根据外部所给信号将寄存器进行更新，从而获得对应使能信号及传输的位置、大小等相关信号。当DMA引擎处理完一个子缓冲区后，更新尾指针并拉高intr对应信号，以进行中断服务响应。  
 （2）读写引擎的状态转移

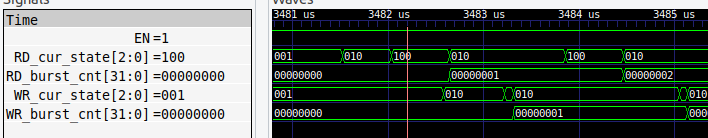
|  |
| --- |
| //State Machine of READ          always @(posedge clk) begin                  if(rst)                          RD\_cur\_state <= RD\_idle;                  else                          RD\_cur\_state <= RD\_next\_state;          end          always @(\*) begin                  case(RD\_cur\_state)                          RD\_idle: begin                                  if(EN & WR\_cur\_state[0] & head\_ptr!=tail\_ptr)                                          RD\_next\_state = RD\_req;                                  else                                          RD\_next\_state = RD\_idle;                          end                          RD\_req: begin                                  if(rd\_req\_ready&rd\_req\_valid)           //consider RD burst cnt != burst time                                          RD\_next\_state = RD\_work;                                  else if(RD\_burst\_cnt == burst\_time)     //after last\_burst                                          RD\_next\_state = RD\_idle;                                  else                                          RD\_next\_state = RD\_req;                          end                          RD\_work: begin                                  //if(rd\_valid & rd\_last & ~fifo\_is\_full)                                  if(rd\_ready & rd\_valid & rd\_last)                                          RD\_next\_state = RD\_req;                                  else                                          RD\_next\_state = RD\_work;                          end                          default:                                  RD\_next\_state = RD\_idle;                  endcase          end  //State machine of WRITE          always @(posedge clk) begin                  if(rst)                          WR\_cur\_state <= WR\_idle;                  else                          WR\_cur\_state <= WR\_next\_state;          end          always @(\*) begin                  case(WR\_cur\_state)                          WR\_idle: begin                                  if(EN & head\_ptr!=tail\_ptr & !fifo\_is\_empty)                                          WR\_next\_state = WR\_req;                                  else                                          WR\_next\_state = WR\_idle;                          end                          WR\_req: begin                                  if(wr\_req\_ready & wr\_req\_valid)                                          WR\_next\_state = WR\_work;                                  else if(WR\_burst\_cnt == burst\_time)                                          WR\_next\_state = WR\_idle;                                  else                                          WR\_next\_state = WR\_req;                          end                          WR\_work: begin                                  if(wr\_ready & wr\_valid & wr\_last)                                          WR\_next\_state = WR\_req;                                  else                                          WR\_next\_state = WR\_work;                          end                          default:                                  WR\_next\_state = WR\_idle;                  endcase          end |

读引擎在首尾指针不等，即子缓冲区非空时启动。写引擎则在此基础上还要求fifo非空，即读引擎已经向fifo填入了数据。读写引擎在请求信号握手后从请求状态进入工作状态，当数据握手且相应last信号拉高时再次进入请求态。当读写burst传输次数达到预先计算值时，进入相应idle态，表示本次子缓冲区传输的相应读写引擎工作完成。  
 （3）burst传输次数的计算和统计

|  |
| --- |
| //burst time and count          assign burst\_time\_A[26:0]  = dma\_size[31:5];          assign burst\_time\_A[31:27] = 5'b0;          assign burst\_time\_B[0]    = |dma\_size[4:0];          assign burst\_time\_B[31:1] = 31'b0;          assign burst\_time = burst\_time\_A + burst\_time\_B; //传输次数，burst\_cnt为0至burst\_time-1，相等表示传输完毕          assign last\_burst\_cnt = burst\_time -1 ;          always @(posedge clk) begin                  if(rst)                          RD\_burst\_cnt <= 32'b0;                  else if(RD\_cur\_state[0] & WR\_cur\_state[0])                          RD\_burst\_cnt <= 32'b0;                  else begin                          if(RD\_cur\_state[2] & rd\_ready & rd\_valid & rd\_last)     //the end of a burst                                  RD\_burst\_cnt <= RD\_burst\_cnt +32'b1;                          else                                  RD\_burst\_cnt <= RD\_burst\_cnt;                  end          end          always @(posedge clk) begin                  if(rst)                          WR\_burst\_cnt <= 32'b0;                  else if(RD\_cur\_state[0] & WR\_cur\_state[0])      //zero out after a transfer                          WR\_burst\_cnt <= 32'b0;                  else begin                          if(WR\_cur\_state[2] & wr\_ready & wr\_valid & wr\_last)     //the end of a burst                                  WR\_burst\_cnt <= WR\_burst\_cnt +32'b1;                          else                                  WR\_burst\_cnt <= WR\_burst\_cnt;                  end          end |

首先根据dma\_size的大小计算出传输次数。当复位或一次子缓冲区处理完之后，将读写传输次数统计清零。当读写引擎每次burst传输结束后将统计值加1，从而使请求状态时burst\_cnt为0至burst\_time-1，当其到达burst\_time时表示最后一次传输结束。便于请求时长度和地址等信号确定。

硬件仿真加速波形说明：



如上图所示，XX\_burst\_cnt在一次请求和工作时保持稳定，表示该次burst传输请求和传输的序号（从0开始）。

1. 请求长度及地址信号的确定

|  |
| --- |
| //len of burst, transmission time of 4 byte data          assign last\_burst\_len\_A[2:0] = dma\_size[4:2];          assign last\_burst\_len\_A[4:3] = 2'b0;          assign last\_burst\_len\_B[0]   = |dma\_size[1:0];          assign last\_burst\_len\_B[4:1] = 4'b0;          assign last\_burst\_len = last\_burst\_len\_A + last\_burst\_len\_B -1 ; //len is 0-7          //when req, burst\_cnt is 0 - burst\_time-1          assign rd\_req\_len = (burst\_time\_B[0] & RD\_burst\_cnt == last\_burst\_cnt) ? last\_burst\_len : 5'd7;          assign wr\_req\_len = (burst\_time\_B[0] & WR\_burst\_cnt == last\_burst\_cnt) ? last\_burst\_len : 5'd7;  //addr of READ and WRITE          always @(posedge clk) begin                  if(RD\_cur\_state[0] & WR\_cur\_state[0] & head\_ptr != tail\_ptr)                          rd\_req\_addr <= src\_base + tail\_ptr;                  else if(RD\_cur\_state[2] & rd\_ready & rd\_valid & rd\_last) begin                          if(RD\_burst\_cnt == last\_burst\_cnt)                                  rd\_req\_addr <= rd\_req\_addr + dma\_size[4:0];                          else                                  rd\_req\_addr <= rd\_req\_addr + 32'd32;                  end          end          always @(posedge clk) begin                  if(RD\_cur\_state[0] & WR\_cur\_state[0] & head\_ptr != tail\_ptr)                          wr\_req\_addr <= dest\_base + tail\_ptr;                  else if(WR\_cur\_state[2] & wr\_ready & wr\_valid & wr\_last) begin                          if(WR\_burst\_cnt == last\_burst\_cnt)                                  wr\_req\_addr <= wr\_req\_addr + dma\_size[4:0];                          else                                  wr\_req\_addr <= wr\_req\_addr + 32'd32;                  end          end |

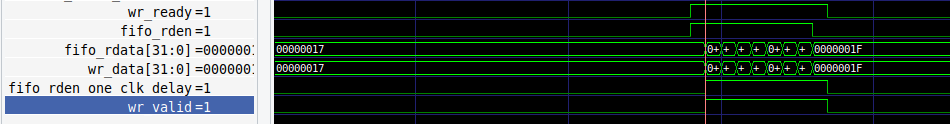
首先根据dma\_size计算最后一次传输长度，并根据burst统计值确定请求长度。在读写引擎开始工作前一拍根据基地址和指针确定读写请求地址的初值，并在每次burst传输后根据本次传输的大小更新请求地址。

（5）控制信号

|  |
| --- |
| //handshake signal of read and write          assign rd\_req\_valid = RD\_cur\_state[1] & ~fifo\_is\_full & RD\_burst\_cnt!=burst\_time;          assign wr\_req\_valid = WR\_cur\_state[1] & ~fifo\_is\_empty & WR\_burst\_cnt != burst\_time;          assign rd\_ready = RD\_cur\_state[2] & ~fifo\_is\_full;          assign wr\_valid = fifo\_rden\_one\_clk\_delay;          assign fifo\_wen = rd\_ready & rd\_valid;          assign fifo\_wdata = rd\_rdata;          assign fifo\_rden = WR\_cur\_state[2] & ~fifo\_is\_empty & wr\_ready;            assign wr\_data = fifo\_rdata;          //考虑到wr\_ready中间突然拉低几个周期，fifo\_rden有效时读出来的数据应当进行保存，          //且应当在之后wr\_ready拉高的第一个周期输出，因此此时valid应当为高，不可直接用fifo\_rden延后一周期实现          always @ (posedge clk) begin                  if(~wr\_ready)                          fifo\_rden\_one\_clk\_delay <= fifo\_rden\_one\_clk\_delay;                  else                          fifo\_rden\_one\_clk\_delay <= fifo\_rden;          end |

读写请求有效信号应当同时考虑fifo状态和burst统计值，最后一次请求状态已经完成所有传输，不必再拉高请求有效信号。由于fifo文件实现的是同步读同步写，因此写有效信号应当延后fifo读使能信号一拍拉高，同时考虑到wr\_ready信号中间突然拉低几个周期的情形，应当在此情况下将wr\_valid信号进行锁存。

硬件仿真波形说明：



如上图所示，在fifo\_rden拉高的下一拍fifo才给出数据，因此应当延后一拍拉高写有效信号。

（6）wr\_last信号的产生

|  |
| --- |
| //last shifter for wr          always @(posedge clk) begin                  if(wr\_req\_ready & wr\_req\_valid)                          last\_shifter <= (9'b1<<(wr\_req\_len+1));                  else if(fifo\_rden)                          last\_shifter <= {1'b0,last\_shifter[8:1]};                  else                          last\_shifter <= last\_shifter;          end          assign wr\_last = last\_shifter[0]; |

在请求信号握手时根据请求长度放置一个移位器，wr\_req\_len+1表示本次传输的数据数，由于fifo\_rden信号恰在写数据前拉高，在每次fifo读使能信号拉高时将其右移，可以保证最后一个数据输出时，1移至last\_shifter的最低位。

1. Custom\_cpu.v

该CPU基于prj3实现的mips型处理器修改，添加了中断和中断屏蔽的相关处理。以下只列出修改部分。  
 （1）状态机相关

|  |
| --- |
| assign Inst\_Req\_Valid  = current\_state[1] & ~(~intr\_mask & intr) ;     //IF  ...  always @ (\*)  begin        case(current\_state)                ...                IF: begin                     if(~intr\_mask & intr)                           next\_state = INTR;                     else if ...                end                INTR: begin                           next\_state = IF;                end  ...  EX: begin                      if(... | inst\_ERET )                           next\_state = IF;                      else if...                end  ...          endcase  end  ...  assign inst\_ERET = opcode[5:0] == 6'b010000 & func[5:0] == 6'b011000; |

添加非中断屏蔽状态IF到INTR的状态转移，和eret指令时从EX到IF的状态转移。注意，当非屏蔽时IF接收到intr中断信号，不必拉高Inst\_Req\_Valid信号。

1. 中断屏蔽

|  |
| --- |
| always @(posedge clk) begin          if(rst)                  intr\_mask <= 1'b0;          else if(current\_state == INTR)                  intr\_mask <= 1'b1;          else if(current\_state == EX & inst\_ERET)                  intr\_mask <= 1'b0;          else                  intr\_mask <= intr\_mask;  end |

进入INTR状态后将屏蔽信号拉高，接受到ERET指令后在EX状态将屏蔽解除。

1. 中断时PC的保存和恢复

|  |
| --- |
| always @ (posedge clk) begin          if(current\_state == INTR) begin                  EPC <= PC\_tmp;          end  end  ...  always @ (posedge clk) begin          ...          else if(current\_state[9])                  PC <= 32'h100;          else if(current\_state[4])begin                  if(inst\_ERET)                          PC   <= EPC;                  else if ...          end  end |

中断进入INTR状态时将PC值寄存至EPC寄存器，且PC跳转至intr\_handler入口地址。接受到eret指令后，PC在EX状态恢复为EPC中寄存值。

1. Intr\_handler.S

|  |
| --- |
| intr\_handler:          # TODO: Please add your own interrupt handler for DMA engine  # Respond to DMA call          la      $k0, 0x60020000         # base address of DMA MMIO register set          lw      $k1, 0x14($k0)          # get ctrl\_Stat, GPR[k1] = MEM [offset + GPR[k0]]          li      $k0, 0x7fffffff         # Store imm temporarily, because imm out of range of andi          and     $k1, $k1, $k0           # Set INTR(31bit) to 0          la      $k0, 0x60020000         # Recover k0, as base addr          sw      $k1, 0x14($k0)          # Write back ctrl\_stat  # Mark sub buffer according to tail\_ptr, last\_tail\_ptr and dma\_size    # count sub result between tail\_ptr          la      $k0, 0x60020000          lw      $k0, 0x08($k0)          # GPR[k0] = cur\_tail\_ptr          la      $k1, last\_tail\_ptr          lw      $k1, 0($k1)             # GPR[k1] = last\_tail\_ptr          beq     $k0, $k1, L1            # if DMA\_engine work succesfully(change tail),change stat          nop    # dma\_buf\_stat -= sub\_buf\_num          la      $k0, dma\_buf\_stat          lw      $k1, 0($k0)             # GPR[k0] = dma\_buf\_stat(origin)          addi    $k1, $k1, -1            # GPR[k0] = dma\_buf\_stat(refreshed)          sw      $k1, 0($k0)             # Write back dma\_buf\_stat    # store cur\_tail\_ptr to last\_tail\_ptr for next call          la      $k0, last\_tail\_ptr      # k0 = addr of last\_tail\_ptr          la      $k1, 0x60020000          lw      $k1, 0x08($k1)          # GPR[k1] = cur\_tail\_ptr          sw      $k1, 0($k0)             # store cur to last  # Return from interrupt  L1:          eret |

汇编程序依次需要完成，将ctrl\_stat最高位(即intr信号)清零，取出当前尾指针和上一次中断尾指针的值并进行比较。若二者相等可直接中断返回，否则需要标记完成的子缓冲区，即dma\_buf\_stat减一（由于dma\_engine实现搬移一个子缓冲区完毕响应一次），并将本次尾指针数据存入last\_tail\_ptr以供下次中断服务使用。

1. Data\_mover.c

实现性能计数，与prj5.1dnn实验类似，不再赘述。

1. 性能对比

|  |  |
| --- | --- |
| 测试任务 | 时钟周期数 |
| Data\_mover\_dma | 51645054 |
| Data\_mover\_no\_dma | 116953279 |

由上图可见，使用DMA外设时钟周期数约为CPU直接使用load和store指令进行数据搬移块时钟周期数的一半。考虑到data\_mover在generate\_data环节为CPU产生数据并填充进源缓冲区，同样会使用大量的sw指令，因此DMA实际的性能应当大于CPU搬移的两倍。

1. 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）

问题1：fifo读数据和内存写有效信号异常

从fifo读取数据至内存时出现异常，wr\_valid信号和有效数据不同时拉高，查看框架代码fifo.v的实现逻辑发现，由于同步读，fifo会在fifo\_rden的后一拍才给出数据及更新fifo状态。因此设置wr\_valid延后一拍拉高，fifo取出数据不再使用时序逻辑进行存储。且fifo\_rden信号考虑fifo非空，而wr\_valid不需额外再次考虑。

问题2：最后一次数据传输异常  
 上板运行时发现无法终止，使用硬件仿真加速工具查看波形时发现，最后一次传输请求长度异常，原因是当dma\_size低5位时，最后一次传输长度同样为7，针对低5位非零情况计算出的last\_burst\_len为-1。因此在请求长度选择时应当考虑dma\_size第五位是否为零的情形。

问题3：汇编程序立即数指令范围不足

开始时在intr\_handler汇编程序中将ctrl\_stat最高位清空时使用andiu，发现立即数超出范围，先将立即数加载到某寄存器中，在使用寄存器指令完成相应操作即可。

问题4：CPU不支持中断服务程序中DIV指令

开始时使用除法计算子缓冲区完成传输数量，上板运行出错。查看反汇编文件及硬件仿真加速波形发现，CPU无法支持DIV指令，考虑到dma\_engine实现中为一次子缓冲区完成即中断一次，因此直接判断两次尾指针是否相等代替原本的子缓冲区计算。

1. 对讲义中思考题（如有）的理解和回答  
   本次实验无思考题。
2. 在课后，你花费了大约\_\_\_20\_\_\_\_小时完成此次实验。
3. 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

建议：

本次实验难度适中，但是在刚开始进行实验时容易对子缓冲区、fifo等概念理解出错。建议在讲解实验时说明处理器和DMA的并行关系、DMA中读写引擎的并行关系，以及CPU、DMA通过data\_mover和intr\_handler实现交互的过程。

致谢：

感谢常老师和陈欲晓在硬件仿真工具上的帮助，感谢芦溶民助教对DMA运行机制的介绍。