LAB2实验报告-DMA搬运数据

张翼鹏

518030910072

一、实验要求

DMA，Direct Memory Access，直接内存访问，是一种不经过CPU而直接从内存存取数据的数据交换模式。在DMA模式下，CPU只需要向DMA控制器下达指令（配置DMA寄存器），传输数据由DMA来完成，数据传送完再把信息反馈给CPU，这样能够减少CPU的资源占有率。

在开始传输前，DMA控制器接收CPU对于源地址，目的地址，搬运数据长度的配置信息，当这些寄存器信息更新后，DMA开始自行进行数据搬运。只需要实现sram1的数据搬运到sram2中即可（或者反过来）。搬运结束后，将DMA中断拉高。要求至少实现下面寄存器的维护：（如果需要可自行维护其它寄存器）

1，源地址寄存器，指示搬运的起始地址，可读可写

2，目的地址寄存器，指示搬运的目的地址，可读可写

3，数据长度寄存器，指示搬运的数据长度，可读可写

4，状态寄存器，只读，指示配置完成，搬运完成等

在软件代码中，配置DMA寄存器以及编写中断服务程序；

二、实验思路

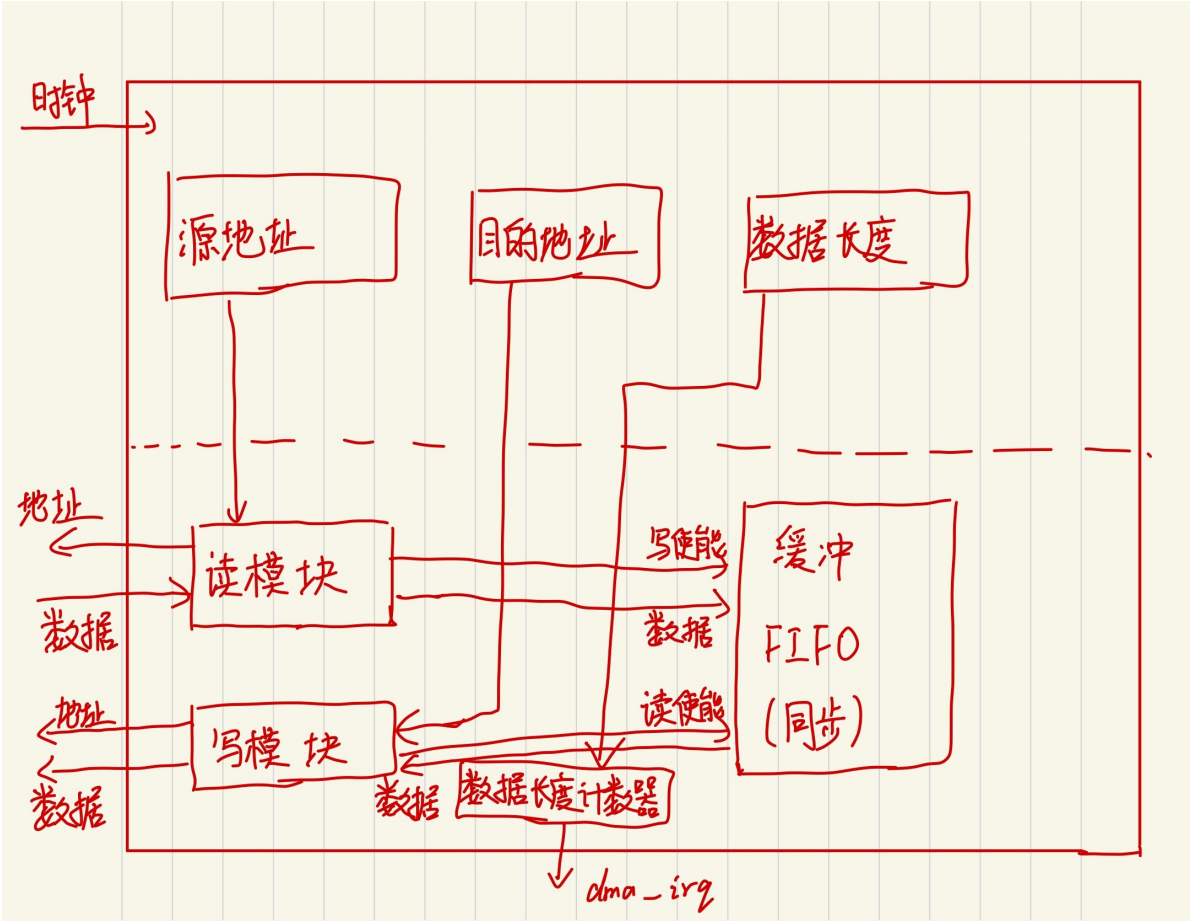
实验时，并不考虑当DMA工作时CPU的工作状态，因此只需负责对DMA进行控制和调试。

1、ICB总线信号

表 12-1 ICB 总线信号

通　道	方向	宽度	信 号 名	介　绍
命令通道	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
	Output	DW	icb_cmd_addr	读写地址
	Output	1	icb_cmd_read	读或是写操作的指示
	Output	DW	icb_cmd_wdata	写操作的数据
	Output	DW/8	icb_cmd_wmask	写操作的字节掩码
反馈通道	Input	1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号
	Input	DW	icb_rsp_rdata	读反馈的数据
	Input	1	icb_rsp_err	读或者写反馈的错误标志

2、DMA结构图



3、DMA工作过程

3.1 DMA寄存器的配置

根据需要，我认为实现最基本的四个寄存器已经足够。刚开始，DMA作为CPU的从设备，接受CPU对其的配置，CPU通过地址线和数据线对DMA的寄存器实现读写。首先配置源地址寄存器，因为SRAM1的地址范围是2000_0000-2000_FFFF，因此源地址为2000_0000。因为SRAM2的地址为3000_0000-3000_FFFF，故目的地址寄存器被配置为3000_0000、数据长度寄存器表示了将要搬运的数据个数，每个数据是32位，即当数据长度被配置为625时，表示将要搬移625个32位数据。其中状态寄存器是DMA根据其他三个寄存器的配置情况由DMA自己进行配置，源地址寄存器、目的地址寄存器和长度寄存器都为32位，状态寄存器为3位，每一位都代表其他三个寄存器的状态，还未配置寄存器时，状态寄存器的初始值为3个1，如果配置源地址寄存器，那么状态寄存器的第一位置为0，如果配置目的地址寄存器，那么将状态寄存器的第二位置为0，如果配置数据长度寄存器，那么将状态寄存器的第三位置为0。当状态寄存器的三位均为0时可以开始工作。考虑到没有读寄存器值的需要，并没有实现读寄存器值的功能。

在配置寄存器时，正确配置是非常重要的。在实践中发现，对于32位寄存器，只有当所配置的寄存器的地址为4的倍数时，才能正确配置。这是因为32是4的倍数，如果所要配置的寄存器的地址不为4的倍数，数据掩码信号会根据地址而相应置0，即覆盖相应的4位数据。因此源地址寄存器、目的地址寄存器、数据长度寄存器的地址分别是：1000_0004,1000_0008,1000_0020。在进行进行寄存器的配置时，地址需要匹配，即发送给DMA的地址线的值需要与内部的寄存器的地址匹配；还需要检查此寄存器是否已经被配置，即查看状态寄存器的对应位是否为0，只有对应位为1的情况下，才允许配置；同时，还要满足dma_cfg_icb_cmd_valid信号和dma_cfg_icb_cmd_read信号为高电平，此时表示CPU发出了配置DMA寄存器的指令，而DMA也可以接受数据，对相应的寄存器进行配置，即把数据线上的数据写入相应寄存器。

当CPU对DMA进行配置时会由valid信号，此时如果DMA发出ready的高电平信号，CPU才认为可以配置，对于DMA而言，ready的判定条件是CPU是否发出valid信号以及相应寄存器是否已经被写入。当DMA端配置完成相应寄存器后，需要发出rsp_valid高电平信号，此信号的逻辑与状态寄存器一致。

3.2 缓冲单元

在DMA从SRAM1读入数据再将数据写入SRAM2的过程中，DMA应该有一部分作为数据缓冲区域。如果是读一次写一次，那么总线的利用率太低，因此实现了深度为16的数据缓冲区，即每读入16个数据之后再对SRAM2进行数据的写入。对缓冲区的读写需要加以控制，缓冲区的读写需要用读写使能信号进行控制，读写使能信号由读写模块发出，缓冲区内部有读地址和写地址。读写地址的最大范围是0-15，考虑到每次都是写入16个再读出16个，并不会有两地址之间的冲突，并且每次写入都是写满，缓冲区的满状态对整个DMA也并没有影响，因此并不考虑缓冲区的空状态信号和满状态信号。当读写使能并且时钟上升沿时，读写地址加一，前一个数值为15时则下一个数值变为0。

3.3 读写模块

当DMA单元的寄存器配置完成之后，读写模块开始工作，DMA作为主设备，而SRAM1和SRAM2作为从设备。读写模块中有一个计数器count，最高可以到31，当计数器count小于等于15时为从SRAM1读取数据，当count大于15而小于等于31时为对SRAM2写数据，此计数器在dma_icb_rsp_valid为高电平且时钟上升沿才可以加一，这样保证了数据正确地读出或者写入。同时，还有一个已写数据长度寄存器，每写入一个数据就加一，当此寄存器的值与数据长度寄存器的值相等时，将完成信号done拉高，dma的中断信号dma_irq也与此信号一致。读写模块还有读地址寄存器和写地址寄存器，读写模块接受DMA的配置寄存器的信息，读地址寄存器和写地址寄存器都从0开始，DMA作为主设备对外发出读写信号时，读或写与count有关，同样，当DMA读入数据时，读地址寄存器加一，写出数据时，写地址寄存器加一。

读写模块还负责当DMA作为主设备时，控制发送dma_icb_cmd信号，读写信号已经说明。dma_icb_cmd_valid信号在状态寄存器为3'b000且dma_icb_rsp_valid信号不为1时为高电平。dma_icb_rsp_ready与dma_icb_rsp_valid信号同步，这是因为可以确保DMA每次都能有效的将数据读入或者写出。因为只有一根地址线，因此是读地址还是写地址需要根据dma_icb_cmd_read信号进行选择。

4、存储器

在本次实验中，存储器已经写好，但是仍然存在问题。比如，在原有的存储器中，sram1和sram2公用一个源文件进行例化，这就造成了读写存储器的冲突，即sram1作为被读的存储器，会有$\$readmemb$函数将数据文档读入sram中的存储器单元。如果sram1和sram2共用一个源文件例化，sram2也将读入数据。这是不愿看到的，因此，在原有文件的基础上，复制之后生成另一个文件$sifv_sim1_ram$，ram1作为被读的存储器，用$sifv_sim1_ram$进行例化，而ram2作为被写的存储器，用$sifv_sim_ram$进行例化。

在原有的基础上，$sifv_sim1_ram$中加入$\$readmemb$函数，读入数据，如下图所示：

```
initial
begin
$readmemb("/home/riscv/demo_e200/vsim/install/rtl/general/data.txt",mem_r);
end
```

在原有的基础上，$sifv_sim_ram$中加入$\$fwrite$函数，将数据写出，但并不是将数据都存入存储器之后统一写入文件，而是每一个数据到来并且写有效信号为高时，将每一次的数据都在当时写入目标文件中。在实验中，我还发现了，如果$\$fwrite$函数中，将输出数据格式设置为二进制，即参数设置为$\%b$，那么将会有多行数据为错误数据，这一点可以通过把错误的数据转换为图片发现，经过多次尝试，偶然发现，如果将此参数设置为十六进制，即将其设置为$\%h$，那么数据可以正确地写入文件中，如下图所示。

```
integer fd;
initial begin
    count <= 0;
    fd = $fopen("/home/riscv/demo_e200/vsim/install/rtl/general/mem_r.txt","w");
end
always @(posedge clk) begin
    if (wen==4'hf)begin
        count <= count + 1'd1;
        $fwrite(fd,"%h\n",din);
    end
    else if (count == 625) begin
        count <= 0;
    end
end
```

另一问题出现在地址信号$addr$上，经过查看信号我们发现，存储器每次接收到的地址信号包含了存储器的基地址，而在文件中，存储器的每次索引也将32位的数据全部用到，这样是不正确的，以ram1为例，如果读地址为2000_0012，那么如果将此地址完全用于存储器的索引，将会出现存储器范围不能满足，因此，我们每次需要将地址信号进行切片，只是用32位地址信号中的后16位。可是，如果将这样的切片用于ram2中，通过观察波形可以发现，数据无法写入ram2存储器中。通过阅读终端报出的$warning$可以知道，即使将地址进行切片，存储器的索引范围依然不能满足。那么将存储器扩大，原本为512，通过尝试发现，将存储器深度调得很大时才能将数据完全写入，最终为了保险，设置深度为4096。

除此之外需要说明的是，使用助教提供的matlab文件进行图像转二进制时，发现每一行只有末八位的数据有用，在存储到存储器中时，考虑到每次的32位数据只有末八位有用有些浪费，因此自己用Python重新写了图像转换代码，转换后每一行为32位且32位均有效，则50*50的灰色图像转为625行32位的数据，即DMA所需搬运的数据长度为625。另外，因为我使用的$\$fwrite$函数中，数据以十六进制的格式写入文件，因为我matlab学的不好，所以用Python重新写了数据文件转图像的代码。上述两个PY文件以及改写过的$sifv_sim_ram.v$、$sifv_sim1_ram.v$文件都会在附件中。

三、具体实现

上述已经将具体的思路、整体的工作时序等讲得比较清楚，接下来将结合具体代码说一下具体的实现步骤。

1、DMA的实现

DMA主要实现三大部分：维护寄存器、DMA对ram的读写请求、数据缓冲。对应文件分别为$e203_dma.v$，$FIFO.v$，$write\_or\_read.v$。其中，$FIFO.v$在$write_or_read.v$中例化，$write_or_read.v$在$e203_dma.v$中例化。

1.1维护寄存器

首先要分配寄存器地址，用到了源地址寄存器、目的地址寄存器、数据长度寄存器和状态寄存器，因此共有四个寄存器地址：

```
parameter source_reg_addr = 32'h1000_0004;
parameter purpose_reg_addr = 32'h1000_0008;
parameter burst_length_addr = 32'h1000_0020;
parameter state_reg_addr = 32'h1000_0016;
```

分配寄存器地址时需要注意，每个寄存器地址后16位应该可以被4整除，因为每个寄存器为32位，即4个字节。否则会出现数据掩膜，即cfg_mask信号不为$4'b1111$。虽然最后一个状态寄存器的地址并不能被4整除，但因为读写寄存器时并没有用到，因此并没有改正。

只有当所要写的寄存器的地址正确且此寄存器还未被写过时，即地址正确且状态寄存器中对应位为1时，$dma_cfg_icb_cmd_ready$信号才为高电平，表示DMA告知CPU可以接受此配置数据，如下图所示。

```
//ready应该是在需要写的寄存器中可以写以及valid有效
assign dma_cfg_icb_cmd_ready =(dma_cfg_icb_cmd_valid && ((state_reg[0] == 1 &&dma_cfg_icb_cmd_addr == source_reg_addr)
                            || (state_reg[1] == 1 && dma_cfg_icb_cmd_addr == purpose_reg_addr)
                            || (state_reg[2] == 1 && dma_cfg_icb_cmd_addr == burst_length_addr)||(dma_cfg_icb_cmd_read && dma_cfg_icb_cmd_addr == state_reg_addr) ))?1'b1:1'b0;
```

目的地址寄存器、源地址寄存器、数据长度寄存器的写操作都是在时钟上升沿且状态寄存器对应位满足1，地址对应正确时进行数据写入，如下代码所示：

```
//源地址寄存器的维护
//当valid信号和state_reg[0]信号都为高是才能接收信号
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        source_reg <= 0;
    else if ((state_reg[0] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == source_reg_addr)
        source_reg <= dma_cfg_icb_cmd_wdata;
    else if (dma_irq)//dma工作完成就将除了状态寄存器外的各个寄存器清0，包括源地址寄存器
        source_reg <= 0;
    else
        source_reg <= source_reg;
end


//目的寄存器的维护
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        purpose_reg <= 0;
    else if ((state_reg[1] == 1) && (dma_cfg_icb_cmd_valid == 1) && dma_cfg_icb_rsp_ready == 1 && dma_cfg_icb_cmd_addr == purpose_reg_addr)
        purpose_reg <= dma_cfg_icb_cmd_wdata;
    else if (dma_irq)
        purpose_reg <= 0;
    else
        purpose_reg <= purpose_reg;
end

//数据长度的维护
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        burst_length <= 0;
    else if ((state_reg[2] == 1) && (dma_cfg_icb_cmd_valid == 1) && dma_cfg_icb_rsp_ready == 1 && dma_cfg_icb_cmd_addr == burst_length_addr)
        burst_length <= dma_cfg_icb_cmd_wdata;
    else if (dma_irq)
        burst_length <= 0;
    else
        burst_length <= burst_length;
end
```

状态寄存器则与之类似，当写入某个寄存器时，如果地址正确且状态寄存器对应位为1，状态寄存器更新对应位的值为0，如下代码所示。

```
//状态寄存器的维护
```

```
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        state_reg <= 3'b111;
    else if ((state_reg[0] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == source_reg_addr)
        state_reg[0] <= 0;
    else if ((state_reg[1] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == purpose_reg_addr)
        state_reg[1] <= 0;
    else if ((state_reg[2] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == burst_length_addr)
        state_reg[2] <= 0;
    else if (dma_irq)
        state_reg <= 3'b111;
    else
        state_reg <= state_reg;
end
```

反馈有效信号在成功写入数据时为高电平，其余情况为低电平。反馈错误信号在写入某一个已经写过的寄存器后，会将错误信号拉高，如下代码所示。

```
reg dma_cfg_icb_rsp_err;
reg dma_cfg_icb_rsp_valid;
//rsp_valid寄存器
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        dma_cfg_icb_rsp_valid <= 0;
    else if ((state_reg[0] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == source_reg_addr)
        dma_cfg_icb_rsp_valid <= 1;
    else if ((state_reg[1] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == purpose_reg_addr)
        dma_cfg_icb_rsp_valid <= 1;
    else if ((state_reg[2] == 1) && (dma_cfg_icb_cmd_valid == 1) && (dma_cfg_icb_rsp_ready == 1) && dma_cfg_icb_cmd_addr == burst_length_addr)
        dma_cfg_icb_rsp_valid <= 1;
    else if (dma_cfg_icb_cmd_addr == state_reg_addr)
        dma_cfg_icb_rsp_valid <= 1;
    else
        dma_cfg_icb_rsp_valid <= 0;
end

//错误反馈寄存器
always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        dma_cfg_icb_rsp_err <= 1'b0;
    else if ((state_reg[0] == 0) && (dma_cfg_icb_cmd_valid == 1) && dma_cfg_icb_cmd_addr == source_reg_addr)
        dma_cfg_icb_rsp_err <= 1'b1;
    else if ((state_reg[1] == 0) && (dma_cfg_icb_cmd_valid == 1) && dma_cfg_icb_cmd_addr == purpose_reg_addr)
        dma_cfg_icb_rsp_err <= 1'b1;
    else if ((state_reg[2] == 0) && (dma_cfg_icb_cmd_valid == 1) && dma_cfg_icb_cmd_addr == burst_length_addr)
        dma_cfg_icb_rsp_err <= 1'b1;
    else
        dma_cfg_icb_rsp_err <= 1'b0;
```

```
31  end
```

综上所述，已经把寄存器的维护部分完成。当DMA完成搬运后，会将状态寄存器置为3'b111，而三个需要维护的寄存器也会清零。

1.2DMA搬运数据时的信号控制

在本次实验中，每次DMA搬运16个数据进行缓冲，时序控制还是比较简单的，只需要在状态寄存器为3'b000后开始工作，需要源地址计数器和目的地址计数器，每次在读写存储器时，将基地址与对应的计数器相加即为所需要的地址。需要注意的是，并不是每次搬运数据都能保证数据长度为16的整数倍，以625为例，625并不是16的整数倍，但是，读数据时所读的多余数据只要不在写入ram2的时候用到即可。因此还需要一个计数器以记录已经写入的数据个数，在此计数器的大小达到数据长度寄存器的大小时，将dma_irq信号拉高，与此同时，将状态寄存器变为3'b111，即表示已经完成搬运，同时将内部的计数器都清空，将$FIFO$的读写地址都置为0，这样下次的新一轮读写数据在写入FIFO进行缓冲时，将覆盖掉老的数据，而不用担心旧数据残留的影响。

$write_or_read$模块在$e203_dma$中进行例化。其中，将源地址寄存器、目的地址寄存器、状态寄存器、数据长度寄存器都读入$write_or_read$中。

```
write_or_read u_write_or_read(
    .clk(clk),
    .rst_n(rst_n),

    .read_source_addr(source_reg),
    .write_source_addr(purpose_reg),
    .data_length(burst_length),
    .state(state_reg),
    .dma_icb_cmd_valid(dma_icb_cmd_valid),
    .dma_icb_cmd_ready(dma_icb_cmd_ready),
    .dma_icb_cmd_addr(dma_icb_cmd_addr),
    .dma_icb_cmd_read(dma_icb_cmd_read),
    .dma_icb_cmd_wdata(dma_icb_cmd_wdata),
    .dma_icb_cmd_wmask(dma_icb_cmd_wmask),
    //
    .dma_icb_rsp_valid(dma_icb_rsp_valid),
    .dma_icb_rsp_ready(dma_icb_rsp_ready),
    .dma_icb_rsp_err(dma_icb_rsp_err),
    .dma_icb_rsp_rdata(dma_icb_rsp_rdata),
    .dma_irq(dma_irq)

);
```

需要注意，将数据掩膜信号始终置为4'b1111，即32位数据均有效，同时，每次读写地址计数器在累加时，都为加4。还需要添加额外计数器$count$来控制是读ram还是写ram，cmd_read信号在$count$小于等于15时为读，在大于15小于等于31时为写。当为31时转为0，反馈接受信号与反馈信号始终一致，FIFO的读写信号也与cmd_read信号一致，其中写使能还需要与ram的反馈信号一致，读使能还需要与ram的ready信号保持一致，如下代码所示。

```
1  assign buffer_write_en = (dma_icb_rsp_valid && dma_icb_cmd_read) ?
   1'b1:1'b0;//buffer的写使能对应的时dma从sram读数据。
2  assign buffer_read_en = (dma_icb_cmd_ready &&
   (~dma_icb_cmd_read)&&dma_icb_cmd_valid) ? 1'b1:1'b0;
3
4  assign dma_irq = (length_count == data_length)?1'b1:1'b0;
5
6  assign dma_icb_cmd_wmask = 32'hFFFF_FFFF;
```

```
assign dma_icb_rsp_ready = dma_icb_rsp_valid;
assign dma_icb_cmd_addr = (dma_icb_cmd_read)? read_addr + read_source_addr:write_addr + write_source_addr;
assign dma_icb_cmd_read = ((count <= 15)&&(length_count != data_length)&&(state == 3'b000))?1'b1:1'b0;     //1代表读，0代表写
assign dma_icb_cmd_valid = (state == 3'b000 && dma_icb_rsp_valid != 1)?1'b1:1'b0;
assign dma_irq = ((length_count != 32'd0)&&(length_count == data_length))?1'b1:1'b0;

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        count <= 0;
    else if (state == 3'b000 && count < 31 && dma_icb_rsp_valid)
        count <= count + 1'b1;
    else if (state == 3'b000 && count == 31 && dma_icb_rsp_valid)
        count <= 0;
    else if (state == 3'b111)
        count <= 0;
end

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        read_addr <= 0;
    else if (state == 3'b000 && count <= 15 && dma_icb_cmd_ready && dma_icb_cmd_valid)
        read_addr <= read_addr + 1;
    else if (state == 3'b111)
        read_addr <= 0;
end

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        write_addr <= 0;
    else if (state == 3'b000 && count > 15 && dma_icb_cmd_ready && dma_icb_cmd_valid)
        write_addr <= write_addr + 1;
    else if (state == 3'b111)
        write_addr <= 0;
end

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)
        length_count <= 0;
    else if (dma_icb_cmd_ready && dma_icb_cmd_valid && state == 3'b000 && count > 15 && length_count != data_length)
        length_count <= length_count + 1;
    else if (dma_icb_cmd_ready && dma_icb_cmd_valid && state == 3'b000 && count > 15 && length_count == data_length)
        length_count <= length_count;
    else if (state == 3'b111)
        length_count <= 0;
end
```

1.3FIFO的具体实现

FIFO的实现和控制是比较简单的，即当使能有效时，将数据写入或读出对应地址数据并且将地址加一，当地址为15时将其转为0。

因为其控制逻辑及代码成分比较简单故不再展示。

2、存储器的实现

存储器部分的改动已经在第二部分的末尾说得非常清楚，这里也不再赘述。

3、软件代码

软件代码中，中断处理函数为空函数，需要注意的是，需要在$platform.h$文件中加入相应的函数名称。而各个寄存器的地址在demo_dma.c文件中固定。demo.c的作用只是配置各个寄存器的值，具体如下图所示。

```
#define DMA_CTRL_ADDR           _AC(0x10000000,UL)
// IOF Mappings
#define DMA_REG(offset)         _REG32(DMA_CTRL_ADDR, offset)
```

```
void dma_init(uint32_t *regDat);
#define DMA_REG_SOURCE_ADDR 0x04
#define DMA_REG_PURPOSE_ADDR 0x08
#define DMA_REG_BURST_ADDR 0x20
#define DMA_REG_STATE 0x16
// #define DMA_REG_SOURCE_ADDR 0x04
// #define DMA_REG_PURPOSE_ADDR 0x08
// #define DMA_REG_BURST_ADDR 0x12
void handle_m_ext_interrupt(){
    //printf("irq_handle");
}

void dma_init(uint32_t *regDat)
{
    DMA_REG(DMA_REG_SOURCE_ADDR) = regDat[0];
    //while (DMA_REG(DMA_REG_STATE) != 0x06){}
    //printf("done");

    DMA_REG(DMA_REG_PURPOSE_ADDR) = regDat[1];
    //while (DMA_REG(DMA_REG_STATE) != 0x04){}
    //printf("done");

    DMA_REG(DMA_REG_BURST_ADDR) = regDat[2];
}

int main(int argc,char **argv)
{
    //set_csr(mie,MIP_MEIP);
    //set_csr(mstatus,MSTATUS_MIE);

    uint32_t data_list[3] = {0x20000000,0x30000000,0x00000271};
    dma_init(data_list);
    return 0;
}
```

可以看到，与i2c不同的是，main函数中的前两行被注释了，这是因为，再加上去后，通过观察波形会发现，整个SOC进入死循环，而无法进入dma_init函数，最终原因也并不清楚，因此做了注释。

四、最终结果

1、波形

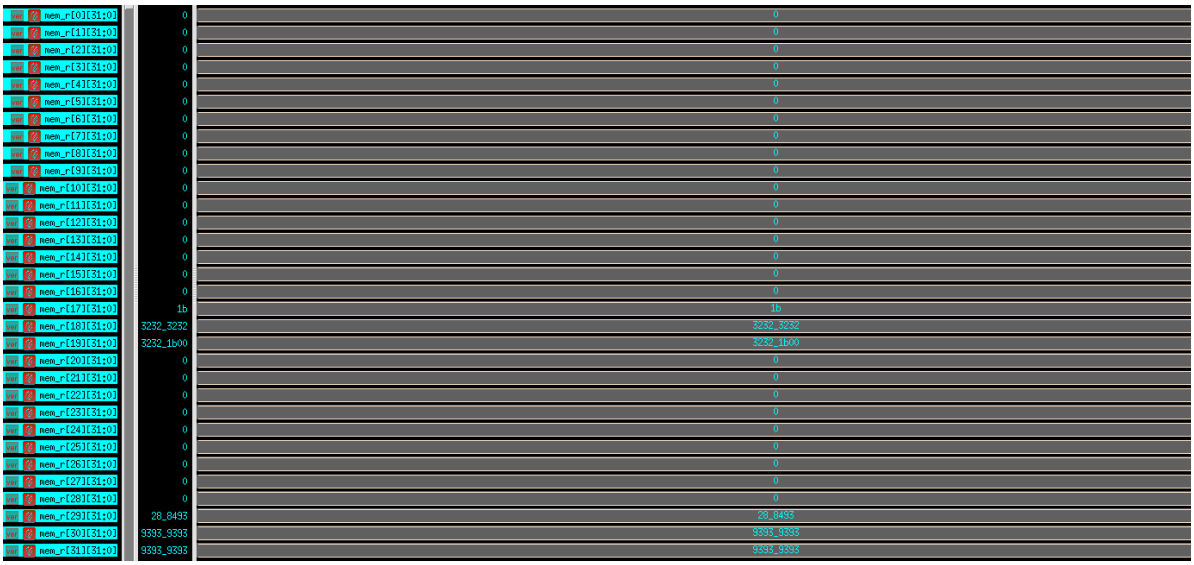
在$vsim$文件夹下输入$make\ wave$命令，打开$verdi$软件查看波形。

在刚开始，可以看到CPU对DMA寄存器的配置：

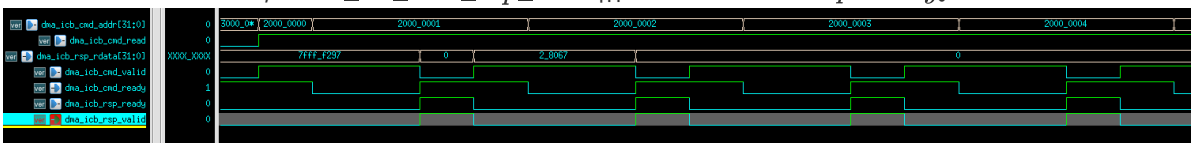
可以看到，$dma_cfg_icb_cmd_addr$依次对源地址寄存器、目的地址寄存器和数据长度寄存器进行配置，$dma_cfg_icb_cmd_wdata$分别为软件程序中写入的数据，其中，最后的数据长度寄存器配置数值为十六进制的271，对应位十进制的625，配置后各寄存器的值如下图所示。

至此寄存器配置完成，而此时，状态寄存器$state_reg$的变化如下图所示，可以看到状态寄存器的数值由七逐渐变为0，这是从3'b111变为3'b000。

全部寄存器配置完成且状态寄存器表示工作状态后，DMA开始工作。而在这之前，可以看到ram1已经将数据文档中的内容读入ram1的存储器中，存储器中部分内容如下图所示：



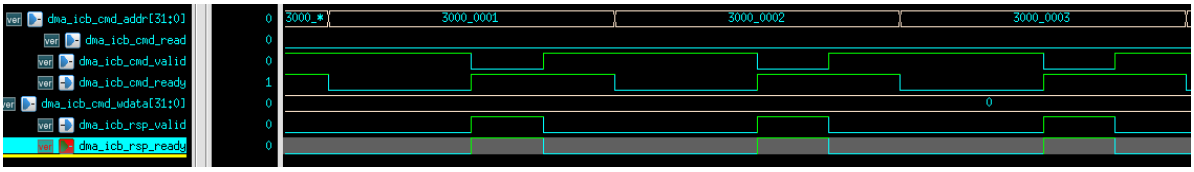
当DMA开始工作之后，首先查看$dma_icb_cmd_addr$，$dma\_icb\_cmd\_valid$，$dma\_icb\_cmd\_rdata$，$dma_icb_cmd_rsp_valid$和$dma_icb_cmd_rsp_ready$。



可以看到，第一次读数据时，发出的地址为2000_0000，对应于ram1的第一个数据，读写信号$read$为高电平表示读，$valid$和$ready$信号在一段时间同时为高，在这一部分，addr的内容才是有效的，之后，addr加一，当数据被读出时，rsp_valid信号和rsp_ready信号为高，表示已经读出数据并且有效，此时数据线上数据才是有效数据。

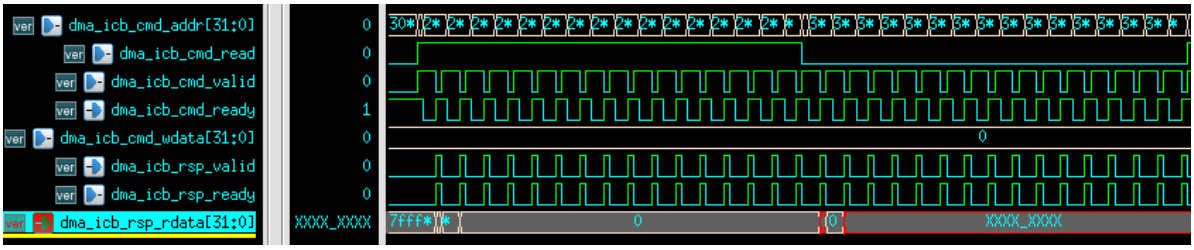
与此同时，ram1的地址线和数据线如下图所示：

每一轮为十六次读和十六次写，在读完数据之后，开始写数据，此时观察$dma_icb_cmd_addr$，$dma\_icb\_cmd\_valid$，$dma\_icb\_cmd\_wdata$，$dma_icb_cmd_rsp_valid$和$dma_icb_cmd_rsp_ready$。



可以看到，第一个写地址为3000_0000，写入的数据为FIFO中暂存的数据。

每一大轮读写信号如下：

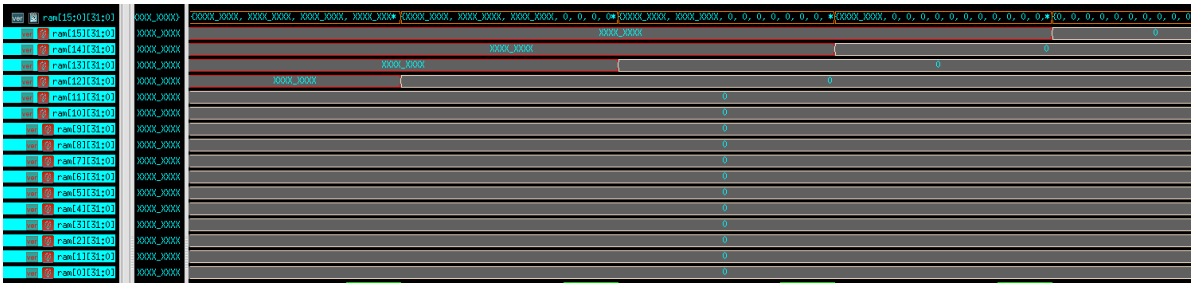


在写数据时，ram2的信号如下所示：

可以看到，每写入一个数据就使count加一。写使能有效是wen信号为4'hf时。

当全部数据写完之后，即写数据长度达到数据长度寄存器中的值时，dma_irq信号拉高。

在此过程中，FIFO作为数据缓冲，根据读写使能更新内部的值，当DMA读取数据时，FIFO被写入，部分信号如下图所示：



2、读数据文件和写数据文件

图像转为二进制的文件，部分内容如下图所示：

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000011011
00110010001100100011001000110010
00110010001100100001101100000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000010100010000100100010011
10010011100100111001001110010011
10010011100100111001001110010011
10010011100100111001001110000100
00101000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000010100101001001110010011
10010011100100111001001110010011
10010011100100111001001110010011
```

DMA搬运后ram2中的数据写出后如下所示：

```
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
0000001b
32323232
32321b00
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00288493
93939393
93939393
93939384
28000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00529393
93939393
93939393
```

3、搬运后的数据转换为图像

利用txt2img.py文件转换后的图像如图所示：

利用img2txt.py文件将原来的彩色图片转为二进制文件，我还加入了将转换的灰色图像显示出来，如下图所示：

可以看到，两者基本一致。

两个python文件内容如下所示：

img2txt.py

```
import cv2 as cv
img = cv.imread("img02.png")
img_gray = cv.cvtColor(img,cv.COLOR_RGB2GRAY)
img_new = cv.resize(img_gray,(50,50))
size = img_new.shape
cv.imwrite("new_02.jpg",img_new)
with open ("data.txt","w") as f:
    count = 0
    for i in range(50):
        for j in range(50):

            if count % 4 == 0 and count != 0:
                f.write("\n")
                print(j)
            f.write((8-len(bin(img_new[i][j])[2:]))*"0"+bin(img_new[i][j])[2:])
            count += 1
            # f.write(bin(img_new[i][j])[2:]+"\n")

    f.close()
```

txt2img.py

```
import cv2 as cv
import numpy as np

img_array = []

def binary_to_dec(bin_str):
    for j in range(4):
        num = int(bin_str[j*2:j*2 + 2],16)
        print(bin_str[j*2:j*2 + 2])
        img_array.append(num)

with open ("mem_r.txt","r") as file :
    line = file.readlines()
    # print(line)
    for i in range(len(line)):
        print(line[i][:-1])
        binary_to_dec(line[i][:-1])
    file.close()

# print(img_array)
img_array = np.array(img_array).reshape((50,50))
print(img_array)
cv.imwrite("trans.jpg",img_array)
```

五、总结

通过本次实验，我更加详细的了解了Soc中，总线设备之间的通信，通过DMA对ram的访问和控制，更加了解了这一系统的基本工作原理。除此之外，在进行调试和不断更改的过程中，锻炼了自己的思维。

因为我新添了一个sifv_sim1_ram.v文件和两个python文件以及读入数据和写出数据时在原来的代码上有些更改，所以如果检验代码需要使用附件中的文件进行替代。

感谢老师和助教的悉心指导。