LAB-1 I2C实验报告

张翼鹏 518030910072

一、例程如何编译

1、首先使用$cd../hbird-e-sdk-master$命令进入文件夹，执行命令$make\ dasm\ PROGRAM=demo_i2c$，通过查看$Makefile$文件：

```
dasm: software
    $(RISCV_OBJDUMP) -D $(PROGRAM_ELF) >& $(PROGRAM_ELF).dump
    $(RISCV_OBJCOPY) $(PROGRAM_ELF) -O verilog $(PROGRAM_ELF).verilog
    sed -i 's/@800/@000/g' $(PROGRAM_ELF).verilog
```

从上边所示文件的一部分可以看出，$dasm$是将多个命令集成在一起，第一行是执行$dump$文件，$dump$文件对C文件进行编译，$dump$文件的名称已经在$Makefile$文件的开头给出：

```
PROGRAM_ELF = software/$(PROGRAM)/$(PROGRAM)
```

第二行是执行$verilog$文件。

其中$PROGRAM=demo_i2c$文件指定了编译文件的文件名。

二、程序如何加载

进入到$vsim$文件夹后，可以看到$install$文件夹，其中的文件夹可以通过$make\ install$命令，通过$Makefile$中的指定位置，将其中的文件夹复制进$install$文件夹中，打开$install$文件夹中的$tb_top.v$文件。

$tb_top.v$文件部分内容如下所示：

```
reg [7:0] itcm_mem [0:(`E203_ITCM_RAM_DP*8)-1];
     initial begin
       $readmemh({testcase, ".verilog"}, itcm_mem);

       for (i=0;i<(`E203_ITCM_RAM_DP);i=i+1) begin
           `ITCM.mem_r[i][00+7:00] = itcm_mem[i*8+0];
           `ITCM.mem_r[i][08+7:08] = itcm_mem[i*8+1];
           `ITCM.mem_r[i][16+7:16] = itcm_mem[i*8+2];
           `ITCM.mem_r[i][24+7:24] = itcm_mem[i*8+3];
           `ITCM.mem_r[i][32+7:32] = itcm_mem[i*8+4];
           `ITCM.mem_r[i][40+7:40] = itcm_mem[i*8+5];
           `ITCM.mem_r[i][48+7:48] = itcm_mem[i*8+6];
           `ITCM.mem_r[i][56+7:56] = itcm_mem[i*8+7];
       end

         $display("ITCM 0x00: %h", `ITCM.mem_r[8'h00]);
         $display("ITCM 0x01: %h", `ITCM.mem_r[8'h01]);
         $display("ITCM 0x02: %h", `ITCM.mem_r[8'h02]);
         $display("ITCM 0x03: %h", `ITCM.mem_r[8'h03]);
         $display("ITCM 0x04: %h", `ITCM.mem_r[8'h04]);
         $display("ITCM 0x05: %h", `ITCM.mem_r[8'h05]);
         $display("ITCM 0x06: %h", `ITCM.mem_r[8'h06]);
```

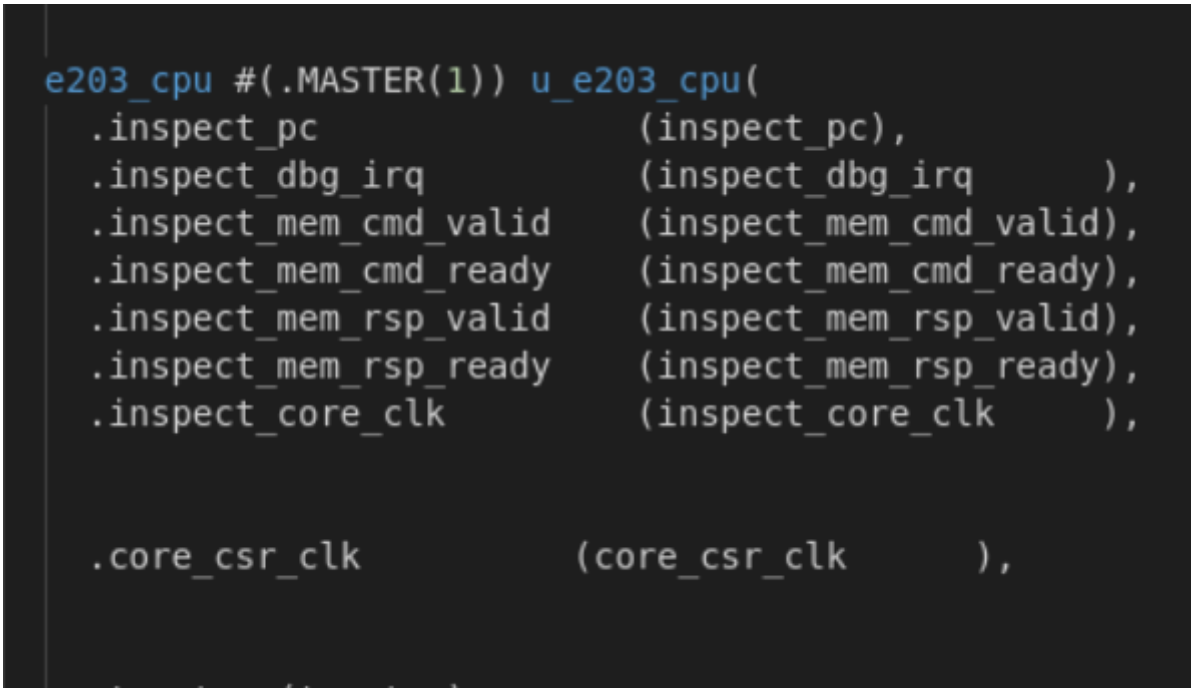
```
23          $display("ITCM 0x07: %h", `ITCM.mem_r[8'h07]);
24          $display("ITCM 0x16: %h", `ITCM.mem_r[8'h16]);
25          $display("ITCM 0x20: %h", `ITCM.mem_r[8'h20]);
```

其中，将指令读入$itcm_mem$寄存器中，再将其中的指令转入‘$ITCM.mem_r$中，其中的$ITCM$在文件开头通过$define$定义：

```
1 `define ITCM
  `CPU_TOP.u_e203_srams.u_e203_itcm_ram.u_e203_itcm_gnrl_ram.u_sirv_sim_ram
```

我对此的理解是，将已经实例化的指令存储器与$ITCM$变量进行关联。从而可以在顶层文件中使用更加底层的实例化模块。

接着阅读$vsim$文件夹下的rtl文件夹下的$core/e203_cpu_top.v$文件，可以看到如下图所示的部分代码：



再看$demo_i2c$文件夹下，可以看到$demo_i2c.dump$文件，部分内容如下图所示：

```
Disassembly of section .init:

80000000 <_start>:
80000000:     30047073            csrci  mstatus,8
80000004:     10001197            auipc  gp,0x10001
80000008:     ef418193            addi   gp,gp,-268 # 90000ef8 <__global_pointer$>
8000000c:     10010117            auipc  sp,0x10010
80000010:     ff410113            addi   sp,sp,-12 # 90010000 <_sp>
80000014:     00000517            auipc  a0,0x0
80000018:     08450513            addi   a0,a0,132 # 80000098 <_itcm>
8000001c:     00000597            auipc  a1,0x0
80000020:     07c58593            addi   a1,a1,124 # 80000098 <_itcm>
80000024:     02b50063            beq    a0,a1,80000044 <_start+0x44>
80000028:     00005617            auipc  a2,0x5
8000002c:     47060613            addi   a2,a2,1136 # 80005498 <__fini_array_end>
80000030:     00c5fa63            bleu   a2,a1,80000044 <_start+0x44>
80000034:     00052283            lw     t0,0(a0)
80000038:     0055a023            sw     t0,0(a1)
8000003c:     0511                addi   a0,a0,4
8000003e:     0591                addi   a1,a1,4
80000040:     fec5eae3            bltu   a1,a2,80000034 <_start+0x34>
80000044:     00005517            auipc  a0,0x5
80000048:     45450513            addi   a0,a0,1108 # 80005498 <__fini_array_end>
8000004c:     10000597            auipc  a1,0x10000
80000050:     fb458593            addi   a1,a1,-76 # 90000000 <_data>
80000054:     81818613            addi   a2,gp,-2024 # 90000710 <_global_atexit>
80000058:     00c5fa63            bleu   a2,a1,8000006c <_start+0x6c>
8000005c:     00052283            lw     t0,0(a0)
80000060:     0055a023            sw     t0,0(a1)
80000064:     0511                addi   a0,a0,4
80000066:     0591                addi   a1,a1,4
80000068:     fec5eae3            bltu   a1,a2,8000005c <_start+0x5c>
8000006c:     81818513            addi   a0,gp,-2024 # 90000710 <_global_atexit>
80000070:     8b818593            addi   a1,gp,-1864 # 900007b0 <_end>
80000074:     00b57763            bleu   a1,a0,80000082 <_start+0x82>
```

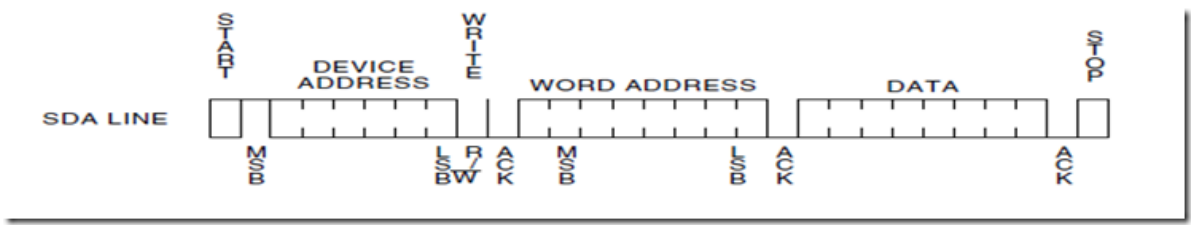
可以看到每一行的开头都为此行代码对应的地址，从8000_0000起步，32位指令，地址每次递增4(如8000_0004)；16位指令，地址每次递增2(如8000_0064-8000_0066)。结合波形。

可以看到，代码中$inspect_pc$即为指令计数器。可以从$verdi$中看到波形，部分波形如下图所示：

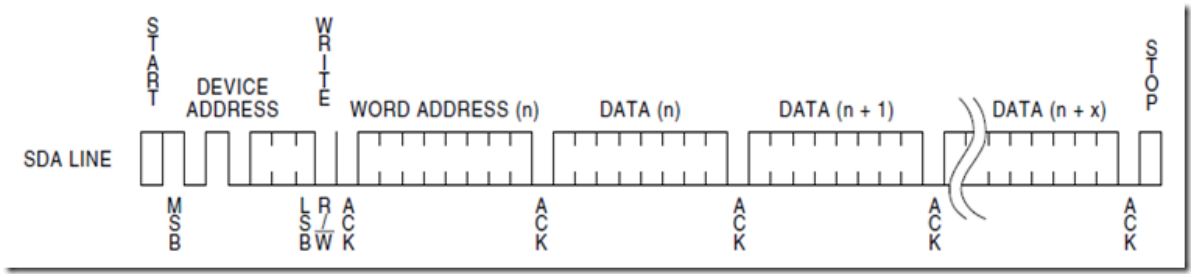
通过波形，可以看到PC值刚开始为依次递增的变化，即每次都从指令存储器中取出指令。

三、I2C的基本工作原理

通过查找网上的资料，我了解到$I2C$的工作原理如图所示：



写时序



读时序

写入数据：发送一个起始信号（$master$设备会将SCL置为高电平「当总线空闲时，SDA和SCL都处于高电平状态」，然后将SDA拉低，这样，所有$slave$设备就会知道传输即将开始。）接着是主设备会发送从设备的地址加一个读写标志位，其中写标志是0，如果有某个从设备的地址与之匹配，从设备会返回一个应答信号ACK，那么接下来的通信就在主设备和此从设备之间进行。主设备会继续发送一个要写入数据的寄存器地址，加一个从设备的应答信号，再紧接着发送8位数据，从设备再进行一次应答，如果数据通信就此结束，那么终止信号是在SDA置于低电平时，将SCL拉高并保持高电平，然后将SDA拉高。

读出数据：与写入数据相同，发送一个起始信号。接着主设备发送从设备地址加一个读写标志位，其中读标志是1。主设备发送一个寄存器地址，接下来从设备向主设备发送8位数据。

读写数据过程中，都是SCL为跳跃的高电平时有效，如下图所示：

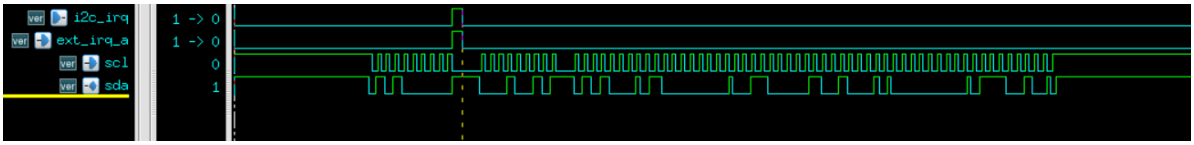
结合本次项目中所用到的蜂鸟CPU以及I2C模块，在$e203_subsys_perips$模块下看到对i2c_master和对i2c_slave模块的例化，打开文件，可以看到如下图所示的信号连接:

```
i2c_master_top u_i2c_master_top (
      .wb_clk_i (clk),
      .wb_rst_i (1'b0),
      .arst_i   (rst_n),
      .wb_adr_i (i2c_wishb_adr[2:0]),
      .wb_dat_i (i2c_wishb_dat_w[7:0]),
      .wb_dat_o (i2c_wishb_dat_r[7:0]),
      .wb_we_i  (i2c_wishb_we),
      .wb_stb_i (i2c_wishb_stb),
      .wb_cyc_i (i2c_wishb_cyc),
      .wb_ack_o (i2c_wishb_ack),

      .scl_pad_i   (i2c_scl_pad_i),
      .scl_pad_o   (i2c_scl_pad_o   ),
      .scl_padoen_o(i2c_scl_padoen_o),

      .sda_pad_i   (i2c_sda_pad_i),
      .sda_pad_o   (i2c_sda_pad_o   ),
      .sda_padoen_o(i2c_sda_padoen_o),

      .wb_inta_o(i2c_irq)
);

i2c_slave_model u_i2c_slave_model(
    .scl(i2c_scl_pad_i),
    .sda(i2c_sda_pad_i)
  );
```

上图所示的代码片段中，只有i2c_req信号是向CPU发送的，i2c的从设备中的scl信号和sda信号是如下图所示产生的：

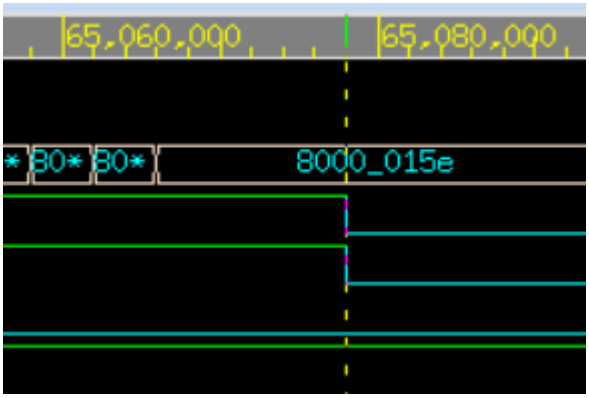
```
assign i2c_scl_pad_i = i2c_scl_padoen_o? 1'bz:i2c_scl_pad_o;
pullup p1(i2c_scl_pad_i);//shangla
assign i2c_sda_pad_i = i2c_sda_padoen_o? 1'bz:i2c_sda_pad_o;
pullup p2(i2c_sda_pad_i);
```

可以看到，$i2c_scl_pad_i$和$i2c_sda_pad_i$信号会分别根据$i2c_scl_padoen_o$ $i2c_sda_padoen_o$信号进行选择，如果为高阻态，那么会有$pullup$进行拉高。这一点可以根据代码结合网上查找到的$I2C$协议分析得出。为了分析整个I2C信号是如何从CPU到I2C主从设备之间如何变化的。波形如下图所示：



可以看到，当I2C传递完从设备地址信号之后，会将i2c_irq信号拉高，做i2c的中断请求。

放大之后，可以看到代码中在$i2c_irq$信号拉高后，PC值如下图所示为8000_015e：



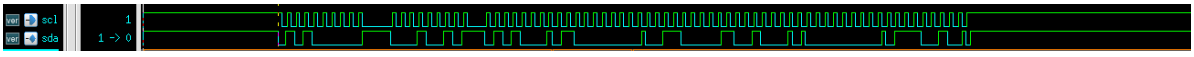
再结合之前所说的demo_i2c,dump文件，找到PC值为8000_015e对应的那一行如下图所示：

```
8000015e:     0141                addi    sp,sp,16
80000160:     8082                ret

80000162 <OV_WriteReg>:
80000162:     1101                addi    sp,sp,-32
80000164:     ca26                sw      s1,20(sp)
80000166:     c452                sw      s4,8(sp)
80000168:     84ae                mv      s1,a1
8000016a:     8a2a                mv      s4,a0
8000016c:     85aa                mv      a1,a0
8000016e:     90000537            lui     a0,0x90000
80000172:     c84a                sw      s2,16(sp)
80000174:     00050513            mv      a0,a0
80000178:     8932                mv      s2,a2
8000017a:     8626                mv      a2,s1
8000017c:     cc22                sw      s0,24(sp)
8000017e:     ce06                sw      ra,28(sp)
80000180:     c64e                sw      s3,12(sp)
80000182:     5b7000ef            jal     ra,80000f38 <iprintf>
80000186:     100427b7            lui     a5,0x10042
8000018a:     fa000713            li      a4,-96
8000018e:     00e781a3            sb      a4,3(a5) # 10042003 <__stack_size+0x10041803>
80000192:     f9000713            li      a4,-112
80000196:     00e78223            sb      a4,4(a5)
8000019a:     10042437            lui     s0,0x10042
8000019e:     00444783            lbu     a5,4(s0) # 10042004 <__stack_size+0x10041804>
800001a2:     8b89                andi    a5,a5,2
800001a4:     ffed                bnez    a5,8000019e <OV_WriteReg+0x3c>
800001a6:     90000537            lui     a0,0x90000
800001aa:     02450513            addi    a0,a0,36 # 90000024 <_sp+0xffff0024>
800001ae:     6ab000ef            jal     ra,80001058 <puts>
800001b2:     47c1                li      a5,16
800001b4:     014401a3            sb      s4,3(s0)
800001b8:     00f40223            sb      a5,4(s0)
```

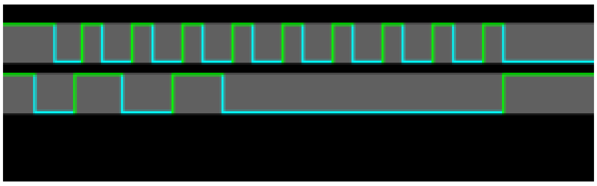
从上图中的红色部分可以看出，这些部分对应的是进入了C文件中的$OV_WriteReg$函数，向I2C发送寄存器地址和数据。

四、波形截图验证输出

整体波形如下图所示：

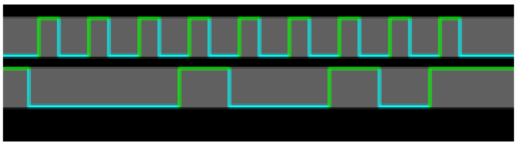


起始信号加从设备地址加读写标志加应答信号：



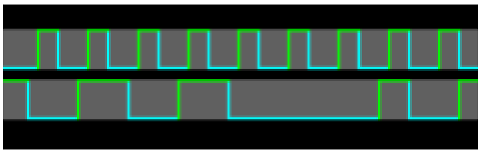
（从设备地址<<1+写标志）

寄存器地址：



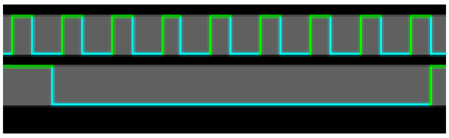
（首地址0x12加应答信号）

我的学号为518030910072，第一个数据为0x51：



（01010001）对应0x51

第二个数据为0x80：



（10000000）对应0x80

以此类推。

五、C文件需要更改的地方

如果不修改C文件中的写函数，那么每次写数据都要重新发送一遍从设备地址和寄存器地址。

修改后的函数为(注释有乱码)：

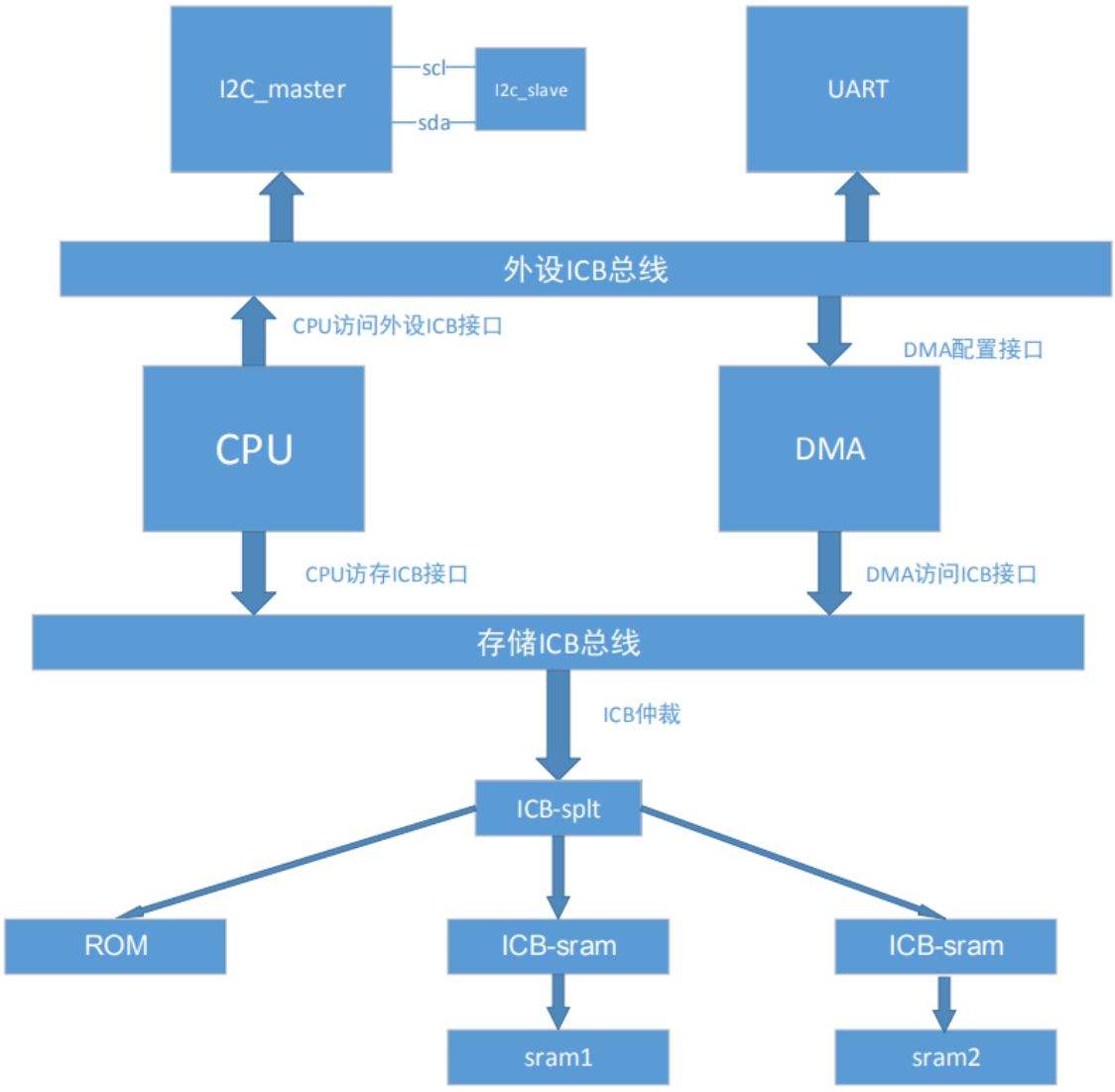
```
uint8_t OV_WriteReg(uint8_t regID, uint8_t *regDat,int length)
{
        printf("write addr : %x ,write data :%x \n",regID,regDat);
        //phase 1
        I2C_REG(I2C_REG_TXR)= SLAVE_ADDR<<1 +0 ;// 0 = write
        I2C_REG(I2C_REG_CR) = 0x90;     //1001 0000   // start bit and WR bit
        while ((I2C_REG(I2C_REG_SR)&0x02)!=0x00)
        {

        }//纭□晶鍛戒护宸茬粡鍙戦€
        printf("phase 1 cmd sent\n");

        //phase 2
        I2C_REG(I2C_REG_TXR)= regID ;//灏嗚□鍙戦€佺殑ID瑁呰繘瀵勫瓨鍣?
        I2C_REG(I2C_REG_CR) = 0x10;//鍙戦€佷粠璁惧□瀛愬湴鍧€  // WR bit
        while ((I2C_REG(I2C_REG_SR)&0x02)!=0x00)
        { }//纭□晶鍛戒护宸茬粡鍙戦€
        printf("phase 2 cmd sent\n");
```

```c


        for (int i = 1;i <= length;i++){
                I2C_REG(I2C_REG_TXR)=regDat[i-1]; //灏嗚□鍙戦€佺殑鏁版嵁鍐欏叆鍙戦€佸瘎瀛樺櫒? I2C_REG(I2C_REG_CR) = 0x10;// WR bit
                while ((I2C_REG(I2C_REG_SR)&0x02)!=0x00)
                { }//绛□晣鍛戒护宸茬粡鍙戦€?
        }
        //phase 3
        // I2C_REG(I2C_REG_TXR)=regDat; //灏嗚□鍙戦€佺殑鏁版嵁鍐欏叆鍙戦€佸瘎瀛樺櫒?    // I2C_REG(I2C_REG_CR) = 0x10;// WR bit
        // while ((I2C_REG(I2C_REG_SR)&0x02)!=0x00)
        // { }//绛□晣鍛戒护宸茬粡鍙戦€?

        // phase 4
        I2C_REG(I2C_REG_CR)=0x40; // Stop bit
        printf("stop \n");

        return 0;
}
```

相比之前函数修改的地方在于，第三部分发送数据用for循环重复发送数据，而函数的参数列表之前为一个数据，现在用指针接收一个数组，再多一个数组长度参数$length$。

六、SOC结构

引用附件中的一张图片：



具体的从处理器到I2C的通路在I2C工作原理中已经说明。