



# SOC实验补充

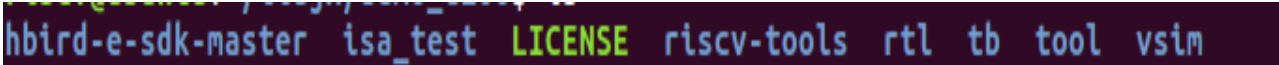
DMA控制器



上海交通大学  
SHANGHAI JIAO TONG UNIVERSITY

# 文件夹



- 文件夹 
- Hbird-e-sdk-master:
  - 软件开发工具包，基础软件，调试与编译软件
  - 操作系统，bsp，模块驱动，测试基准软件
- Riscv-tools:
  - 仿真过程中所需要的工具链
- RTL： SoC代码
- Tb： SoC testbench
- Tool： 编译和调试工具
- Vsim: 用户仿真文件，install时将其他所需文件拷贝/链接到本目录

# platform.h



- 文件夹路径

```
[jiangjf@DIC212 demo_e200]$ find -name "platform.h"
./hbird-e-sdk-master/software/FreeRTOSv9.0.0/Demo/RISCV_HiFive1_GCC/bsp/env/freedom-e300-hifive1/platform.h
./hbird-e-sdk-master/software/FreeRTOSv9.0.0/Demo/RISCV_E31Arty_GCC/bsp/env/coreplexip-e31-arty/platform.h
./hbird-e-sdk-master/bsp/hbird-e200/env/platform.h
./riscv-tools/riscv-isa-sim/softfloat/platform.h
```

```
/*
 * Platform definitions
 */

#define TRAPVEC_TABLE_CTRL_ADDR _AC(0x00001010,UL)
#define CLINT_CTRL_ADDR         _AC(0x02000000,UL)
#define PLIC_CTRL_ADDR          _AC(0x0C000000,UL)
// #define AON_CTRL_ADDR         _AC(0x10000000,UL)
#define GPIO_CTRL_ADDR          _AC(0x10012000,UL)
#define UART0_CTRL_ADDR         _AC(0x10013000,UL)
#define SPI0_CTRL_ADDR           _AC(0x10014000,UL)
#define PWM0_CTRL_ADDR          _AC(0x10015000,UL)
#define UART1_CTRL_ADDR         _AC(0x10023000,UL)
#define SPI1_CTRL_ADDR           _AC(0x10024000,UL)
#define PWM1_CTRL_ADDR          _AC(0x10025000,UL)
#define SPI2_CTRL_ADDR           _AC(0x10034000,UL)
#define PWM2_CTRL_ADDR          _AC(0x10035000,UL)
#define I2C_CTRL_ADDR           _AC(0x10042000,UL)
// add for dma
#define DMA_CTRL_ADDR           _AC(0x10000000,UL)
```

PS: AON\_CTRL\_ADDR 和 dma 的基地址重复了, 但是不影响。(不注释掉也不会影响)

# platform.h



- 增加定义

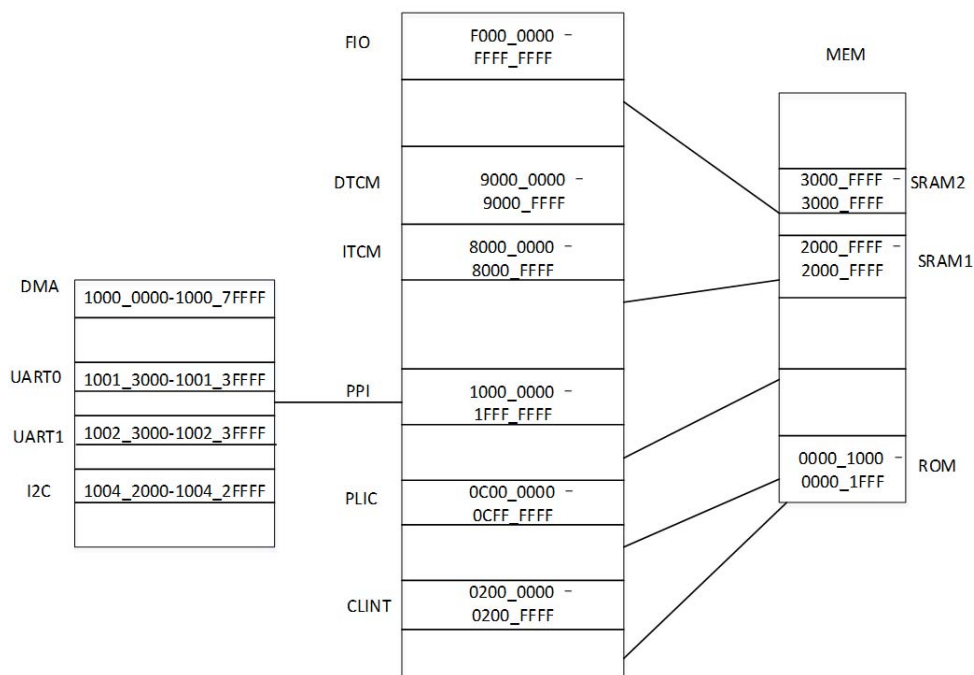
platform.h 里面增加两个关于 **DMA** 写寄存器的函数，**8bit** 和 **32bit** 均需要。（因为我们有两种长度的 reg）

```
#define I2C_REG(offset)          _REG8(I2C_CTRL_ADDR, offset)
// DMA
#define DMA_REG_8(offset)        _REG8(DMA_CTRL_ADDR, offset)
#define DMA_REG_32(offset)       _REG32(DMA_CTRL_ADDR, offset)
```

# 上电顺序

PC值从1000→1004→8000\_0000

ver	pc[31:0]	0	0	1000	1004	8000_0000	8000_0004	8
ver	pc_vld[31:0]	0	0	1	0			
ver	instr[31:0]	0	0	7fff_f297	2_8067	3004_7073	1000_1197	e



先跳到ROM，然后跳到ITCM，  
ITCM里面有编译好的二进制指令



# 上电顺序



```
module sirv_rom # (
    parameter AW = 12,
    parameter DW = 32,
    parameter DP = 1024
)(
    input [AW-1:2] rom_addr,
    output [DW-1:0] rom_dout
);

wire [31:0] mask_rom [0:DP-1]; // 4KB = 1KW

assign rom_dout = mask_rom[rom_addr];

genvar i;
generate
if(1) begin: jump_to_ram_gen
    // Just jump to the ITCM base address
    for (i=0; i<1024; i=i+1) begin: rom_gen
        if(i==0) begin: rom0_gen
            assign mask_rom[i] = 32'h7ffff297; //auipc    t0, 0x7ffff
        end
        else if(i==1) begin: rom1_gen
            assign mask_rom[i] = 32'h00028067; //jr      t0
        end
        else begin: rom_non01_gen
            assign mask_rom[i] = 32'h00000000;
        end
    end
end
else begin: jump_to_non_ram_gen
```

```
e203_cpu_top u_e203_cpu_top(
    .inspect_pc          (),
    .inspect_dbg_irq     (),
    .inspect_mem_cmd_valid (),
    .inspect_mem_cmd_ready (),
    .inspect_mem_rsp_valid (),
    .inspect_mem_rsp_ready (),
    .inspect_core_clk    (),

    .core_csr_clk        ( ),

    .tm_stop             ( ),
    .pc_rtvect            (32'h0000_1000),
```

ROM里面两条指令，一条auipc，一条jr指令。

# 上电顺序

```
reg [7:0] itcm_mem [0:(`E200_ITCM_RAM_DP*8)-1];
initial begin
    $readmemh({testcase, ".verilog"}, itcm_mem);

    for (i=0;i<(`E200_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]);
    $display("ITCM 0x07: %h", `ITCM.mem_r[8'h07]);
    $display("ITCM 0x16: %h", `ITCM.mem_r[8'h16]);
    $display("ITCM 0x20: %h", `ITCM.mem_r[8'h20]);
```

在tb里，通过readmemh系统函数将已经生成好的.verilog文件读到ITCM里面的数组里面去

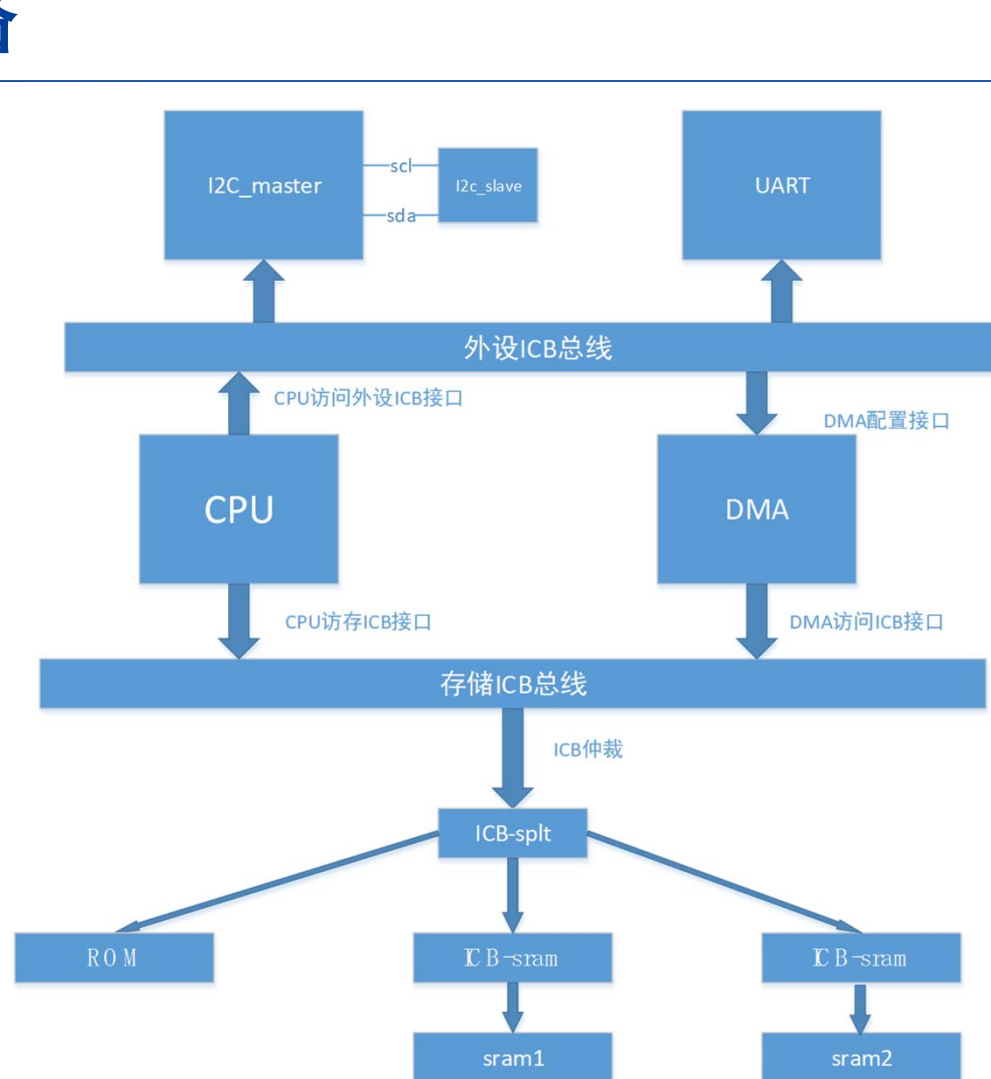
## 编译顺序



- Hbird目录是软件仿真，用它来将.c/.h文件编译成.verilog文件和.dump文件
- Vsim目录是RTL仿真，tb里面会将生成的.verilog文件读到ITCM，然后再执行
- 每次修改软件代码后需要先在hbird目录下编译，再到vsim目录下make
- 每次修改硬件代码后需要在vsim目录下make clean, make install, make run\_test



# 数据通路



## 数据通路-软件部分



指定i2c模块base\_addr

```
#define I2C_CTRL_ADDR    _AC(0x10042000,UL)
```

通过volatile对特定地址进行读写

```
#define _REG8(p, i)      (*(volatile uint8_t *) ((p) + (i)))
```

通过封装好的函数对I2C模块进行寄存器读写

```
#define I2C_REG(offset)  _REG8(I2C_CTRL_ADDR, offset)
```

# 数据通路-软件部分



指定中断服务程序名称

```
2 .weak handle_trap
3 handle_trap:
4 1:
5   j 1b
6
7 #endif
```

服务程序定义

```
9 uintptr_t handle_trap(uintptr_t mcause, uintptr_t epc)
10 {
11     /*
12     if (0){
13         // External Machine-Level interrupt from PLIC
14     } else if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_EXT)) {
15         handle_m_ext_interrupt();
16         // External Machine-Level interrupt from PLIC
17     } else if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_TIMER)){
18         handle_m_time_interrupt();
19     }
20     else {
21         write(1, "trap\n", 5);
22         _exit(1 + mcause);
23     }
24     */
25     handle_m_ext_interrupt();
26     return epc;
27 }
```

定义为weak函数，当出现重名函数时自动覆盖


```
__attribute__((weak)) void handle_m_ext_interrupt() {};
```

```
5 void handle_m_ext_interrupt(){
6     printf("irq handle\n");
7     I2C_REG(I2C_REG_CTR) = 0x80; //i2c
8 }
```

```
ext_irq_a      (dma_irqli2c_irq),
ext_irq_b      (1'b0)
```

# RTL目录

- Core (config.v)
- Fab: (总线)
- General: (通用模块)
- Mems (存储器, rom)
- Periphs: (外设)
- Soc (系统)
- Subsys (子系统, define.v)



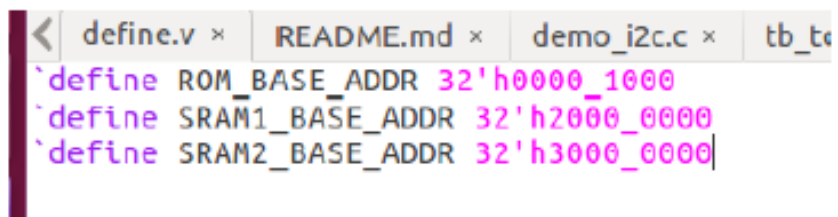
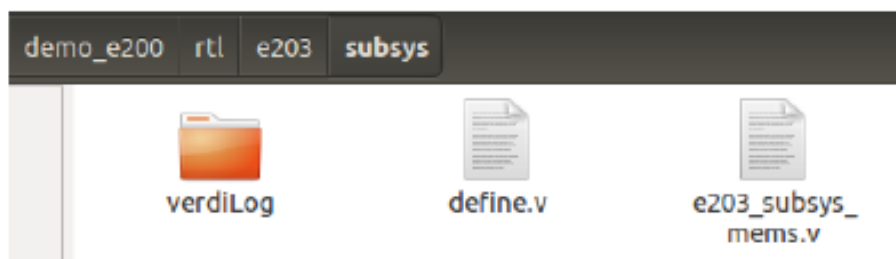
```
└─ sirv_icb1to16_bus
   └─ u_buf_icb_splt (sirv_gnrl_icb_splt)
      └─ u_sirv_gnrl_icb_buffer (sirv_gnrl_icb_buffer)
└─ sirv_icb1to2_bus
   └─ sirv_gnrl_cdc_rx
      └─ sirv_gnrl_cdc_tx
         └─ sirv_gnrl_ltcx
            └─ sirv_gnrl_icb_n2v
               └─ sirv_gnrl_icb2axi
                  └─ sirv_gnrl_icb2apb
                     └─ th_top
└─ u_e203_soc_top (e203_soc_top)
   └─ u_e203_subsys_top (e203_subsys_top)
      └─ i_periphs (e203_subsys_periphs)
         └─ u_e203_cpu_top (e203_cpu_top)
            └─ u_e203_subsys_mems (e203_subsys_mems)
               └─ 2 undefined modules
                  └─ u_men_icb_arbt (sirv_gnrl_icb_arbt)
                     └─ arbt_num_gt_1_gen
                        └─ u_sirv_mrom_top (sirv_mrom_top)
                           └─ u_sirv_sim_ram1 (sirv_sim_ram)
                              └─ u_sirv_sim_ram2 (sirv_sim_ram)
└─ u_main_ResetCatchAndSync_2_1 (sirv_ResetCatchAndSync_2)
```

# SRAM基地址定义

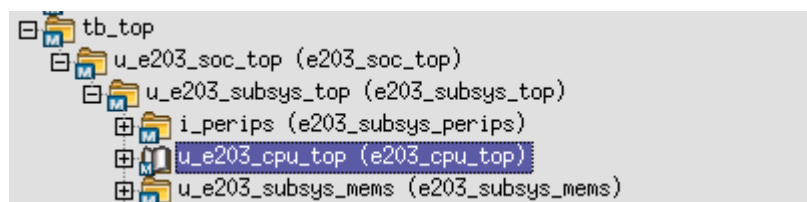


## 3. SRAM1 和 SRAM2 的基地址在 subsys/define.v 中有定义

这两个基地址也就是后续往 DMA 的源地址寄存器和目的地址寄存器写入的数据



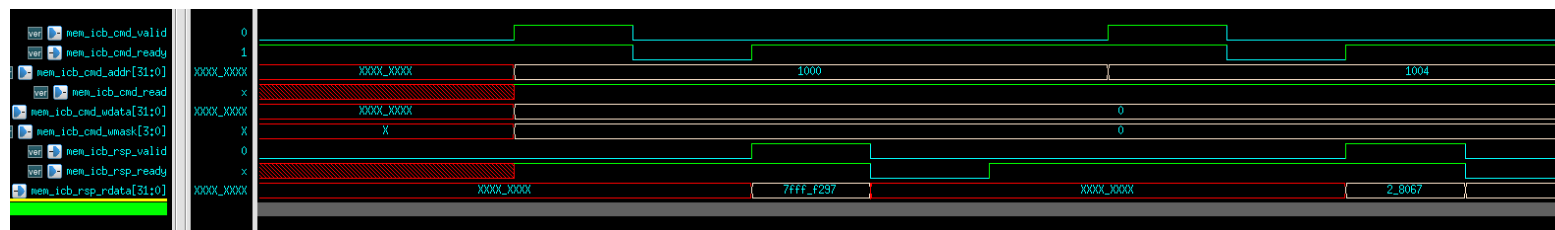
## 数据通路-硬件部分



Subsys\_perip是外设ICB总线,  
subsys\_mems是存储总线

```
//printf("i2c init\n"); //scl频率配置
I2C_REG(I2C_REG_PRERlo) = 0x7d ;//400KHZ
```

上电时, 由于PC定位到ROM, ROM挂在存储总线上, 相应的接口会发icb读时序





## 数据通路-硬件部分

```
// The total address range for the PPI is from/to
// *****0x1000 0000 -- 0x1FFF FFFF
// There are several slaves for PPI bus, including:
// * DMA      : 0x1000 0000 -- 0x1000 7FFF
// * Example-WishBone : 0x1004 2000 -- 0x1004 2FFF
// * UART0    : 0x1001 3000 -- 0x1001 3FFF
// * UART1    : 0x1002 3000 -- 0x1002 3FFF

sirv_icb1to8_bus # (
  .ICB_FIFO_DP      (2), // We add a ping-pong buffer here to cut down the timing path
  .ICB_FIFO_CUT_READY (1), // We configure it to cut down the back-pressure ready signal

  .AW      (32),
  .DW      (`E203_XLEN),
  .SPLT_FIFO_OUTS_NUM (1), // The peripherals only allow 1 outstanding
  .SPLT_FIFO_CUT_READY (1), // The peripherals always cut ready
  // * DMA      : 0x1000 0000 -- 0x1000 7FFF
  .OO_BASE_ADDR      (32'h1000_0000),
  .OO_BASE_REGION_LSB (15),
  // * Here is an I2C WishBone Peripheral
  .O1_BASE_ADDR      (32'h1004_2000),
```

```
////////// iic part//////////
// * Here is an example WishBone Peripheral
wire [`E203_ADDR_SIZE-1:0] i2c_wishb_adr; // lower address bits
wire [8-1:0] i2c_wishb_dat_w; // databus input
wire [8-1:0] i2c_wishb_dat_r; // databus output
wire i2c_wishb_we; // write enable input
wire i2c_wishb_stb; // stobe/core select signal
wire i2c_wishb_cyc; // valid bus cycle input
wire i2c_wishb_ack; // bus cycle acknowledge output

sirv_gnrl_icb32towishb8 # (
  .AW (`E203_ADDR_SIZE)
) u_i2c_wishb_icb32towishb8(
```

- 对于I2C\_REG函数，对应的会出现在ppi-icb接口
- 经过一个icb1to8\_bus，bus挂有DMA (s) ,i2c, uart
- 接icb转wishbone的桥后，接i2c外设

```
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),
```

## 数据通路-硬件部分



```
else
  if (wb_wacc)
    case (wb_adr_i) // synopsis parallel_case
      3'b000 : prer [ 7:0] <= #1 wb_dat_i;
      3'b001 : prer [15:8] <= #1 wb_dat_i;
      3'b010 : ctr      <= #1 wb_dat_i;
      3'b011 : txr      <= #1 wb_dat_i;
      //default: ; //Bob: here have a lint warning, so commented it out
    endcase
```

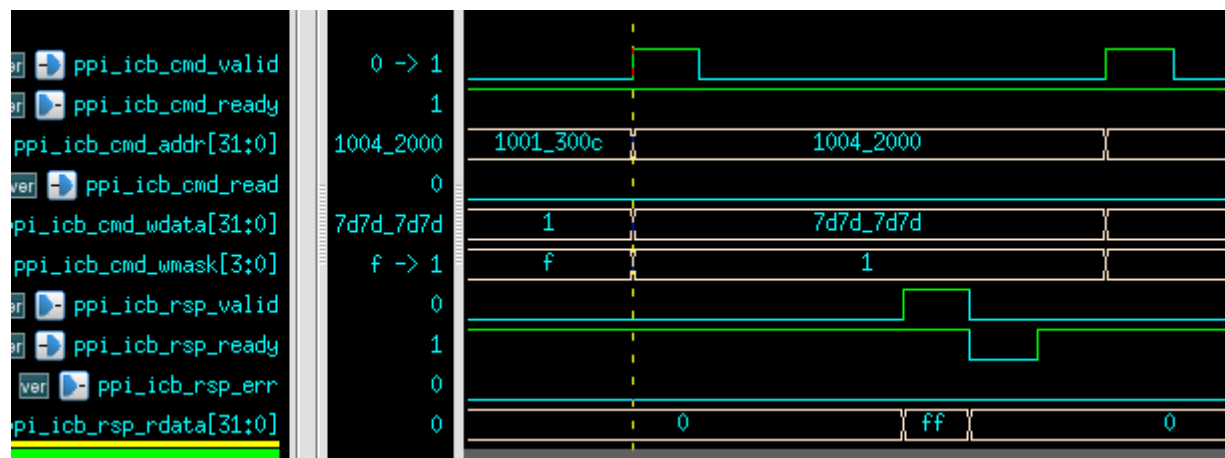
i2c模块内部会根据地址做判断，并维护相关寄存器

```
// decode command register
wire sta = cr[7];
wire sto = cr[6];
wire rd  = cr[5];
wire wr  = cr[4];
wire ack = cr[3];
wire iack = cr[0];
```

寄存器的相关位会作为i2c核心逻辑的控制信号

```
// hookup byte controller block
i2c_master_byte_ctrl byte_controller (
  .clk    ( wb_clk_i ),
  .rst    ( wb_rst_i ),
  .nReset ( rst_i    ),
  .ena    ( core_en  ),
  .clk_cnt ( prer    ),
  .start  ( sta      ),
  .stop   ( sto      ),
  .read   ( rd       ),
  .write  ( wr       ),
```

## 数据通路-硬件部分





## IIC端口问题



subsystem\_perip

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

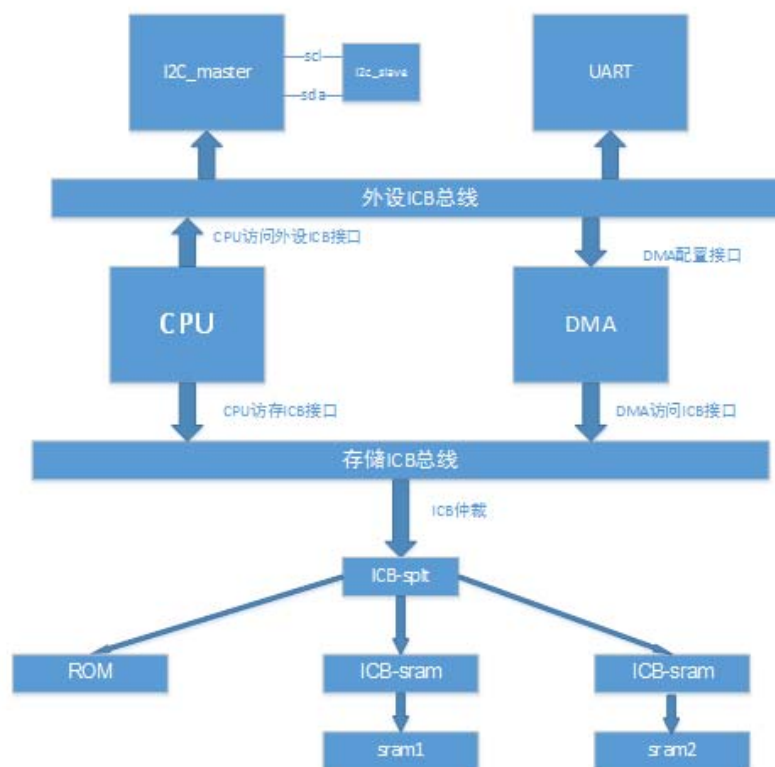
```
i2c_master_top u_i2c_master_top (  
    .wb_clk_i (clk),  
    .wb_rst_i (1'b0),  
    .arst_i    (rst_n),  
    .wb_adr_i (i2c_wishb_adr[2:0]),  
    .wb_dat_i (i2c_wishb_dat_w[7:0]),  
    .wb_dat_o (i2c_wishb_dat_r[7:0]),  
    .wb_we_i  (i2c_wishb_we),  
    .wb_stb_i (i2c_wishb_stb),  
    .wb_cyc_i (i2c_wishb_cyc),  
    .wb_ack_o (i2c_wishb_ack),  
  
    .scl_pad_i  (i2c_scl_pad_i),  
    .scl_pad_o  (i2c_scl_pad_o  ),
```

```
// assign scl and sda output (always gnd)  
assign scl_o = 1'b0;  
assign sda_o = 1'b0;
```

## 实验中的DMA模块



- 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。



# 实验介绍



按下面要求编写DMA控制器模块以及相应的软件驱动程序：

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）

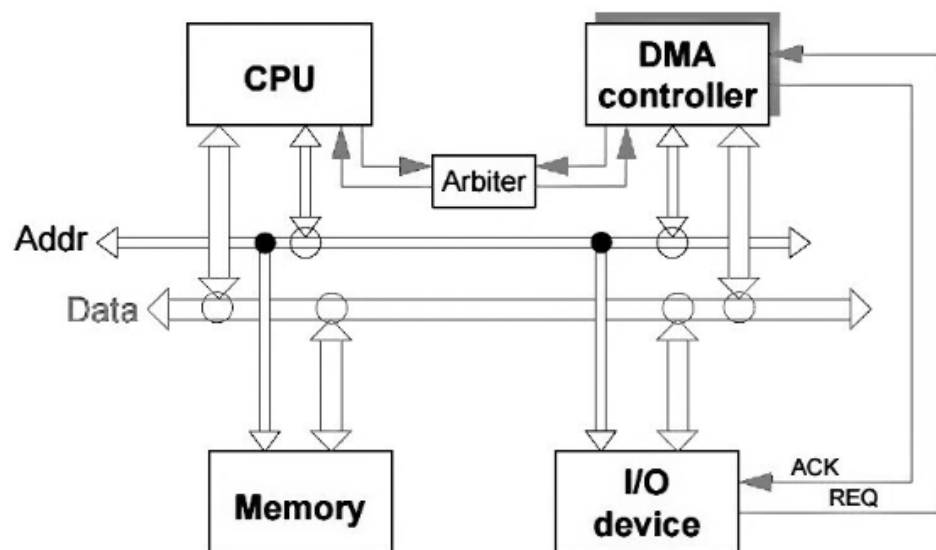
- 1，源地址寄存器，指示搬运的起始地址，可读可写
- 2，目的地址寄存器，指示搬运的目的地址，可读可写
- 3，数据长度寄存器，指示搬运的数据长度，可读可写
- 4，状态寄存器，只读，指示配置完成，搬运完成等

在软件代码中，配置DMA寄存器以及编写中断服务程序；

要求在报告文档里解释清除dma硬件模块的设计思路，需要有相应的波形介绍以及所维护的寄存器列表；并以附件形式附上具体的代码（硬件+软件）



# 模块设计顶层



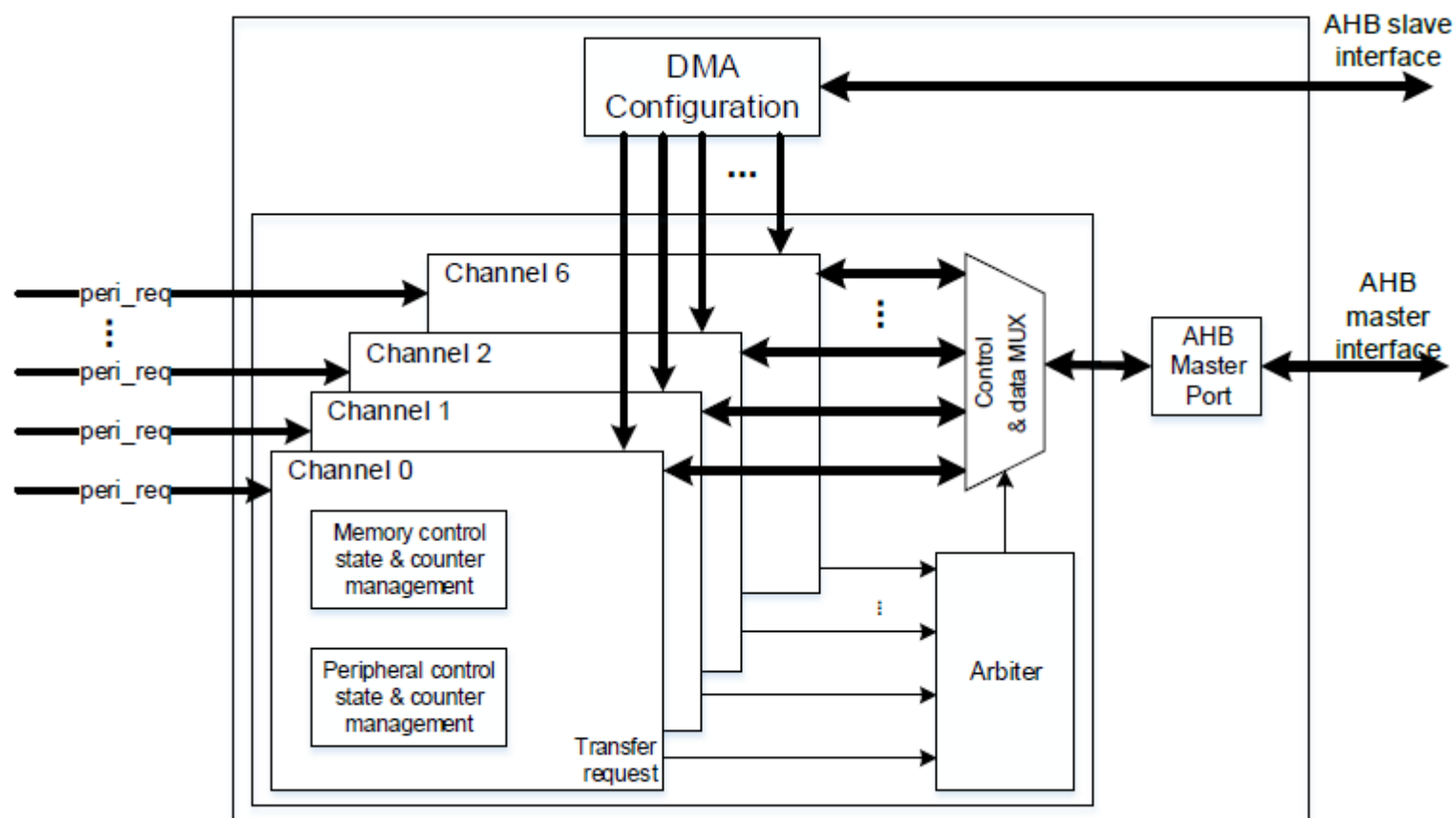
```
e203_dma i_dma(
    .clk(clk),
    .rst_n(rst_n),
    .dna_icb_cmd_valid (dna_icb_cmd_valid),
    .dna_icb_cmd_ready (dna_icb_cmd_ready),
    .dna_icb_cmd_addr (dna_icb_cmd_addr ),
    .dna_icb_cmd_read (dna_icb_cmd_read ),
    .dna_icb_cmd_wdata (dna_icb_cmd_wdata),
    .dna_icb_cmd_wmask (dna_icb_cmd_wmask),

    .dna_icb_rsp_valid (dna_icb_rsp_valid),
    .dna_icb_rsp_ready (dna_icb_rsp_ready),
    .dna_icb_rsp_err (dna_icb_rsp_err),
    .dna_icb_rsp_rdata (dna_icb_rsp_rdata),
    .dna_irq (dna_irq),

    .dna_cfg_icb_cmd_valid (dna_cfg_icb_cmd_valid),
    .dna_cfg_icb_cmd_ready (dna_cfg_icb_cmd_ready),
    .dna_cfg_icb_cmd_addr (dna_cfg_icb_cmd_addr ),
    .dna_cfg_icb_cmd_read (dna_cfg_icb_cmd_read ),
    .dna_cfg_icb_cmd_wdata (dna_cfg_icb_cmd_wdata),
    .dna_cfg_icb_cmd_wmask (dna_cfg_icb_cmd_wmask),

    .dna_cfg_icb_rsp_valid (dna_cfg_icb_rsp_valid),
    .dna_cfg_icb_rsp_ready (dna_cfg_icb_rsp_ready),
    .dna_cfg_icb_rsp_err (dna_cfg_icb_rsp_err ),
    .dna_cfg_icb_rsp_rdata (dna_cfg_icb_rsp_rdata)
);
```

# DMA结构



# 寄存器写时序

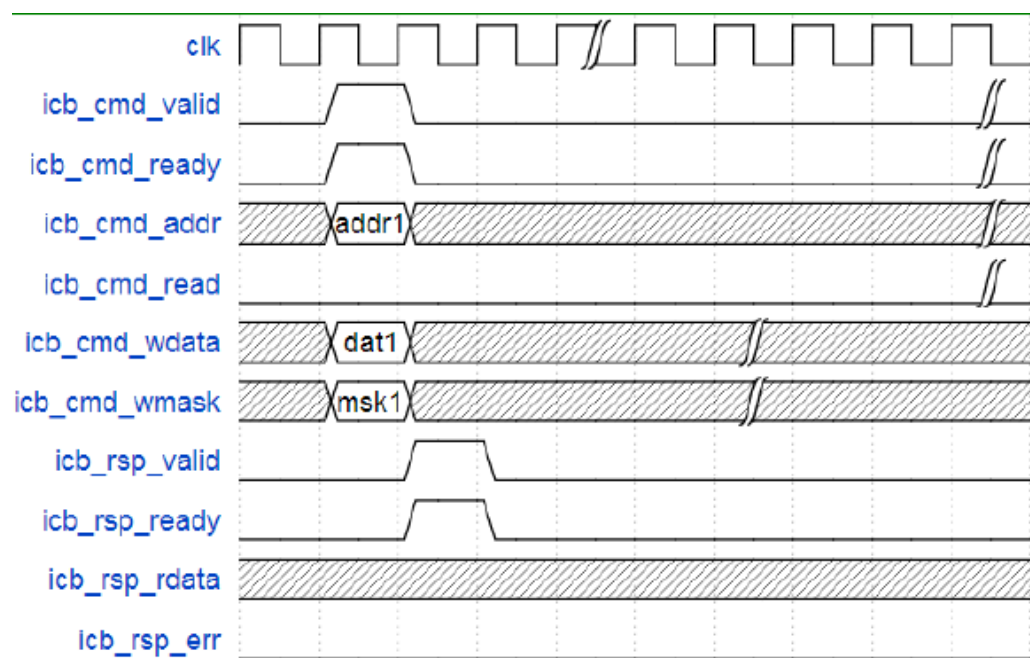


- 读写时序需参考doc/蜂鸟E203开源SoC简介.pdf文件
- 写时序
- 读时序

## 写时序1



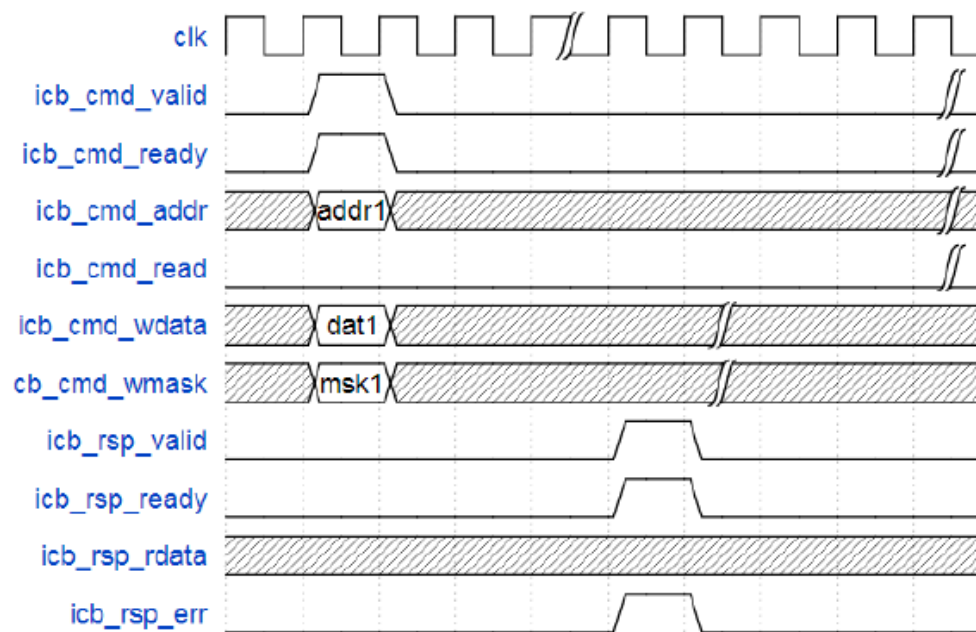
- 主设备向从设备通过ICB的Command Channel发送写操作请求（icb\_cmd\_read为低），从设备立即接收该请求（icb\_cmd\_ready为高）。从设备在下一个周期返回读结果且结果正确（icb\_rsp\_err为低），主设备立即接收该结果（icb\_rsp\_ready为高）。



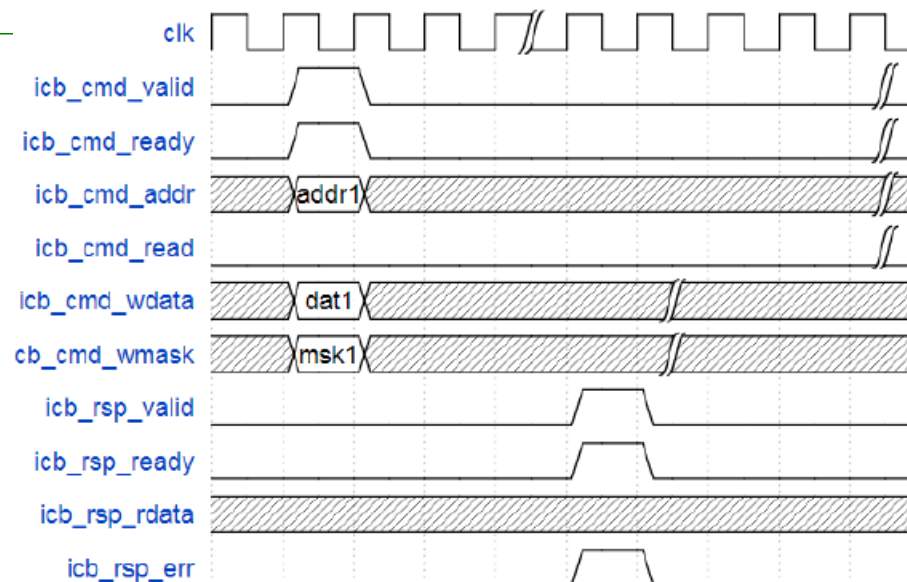
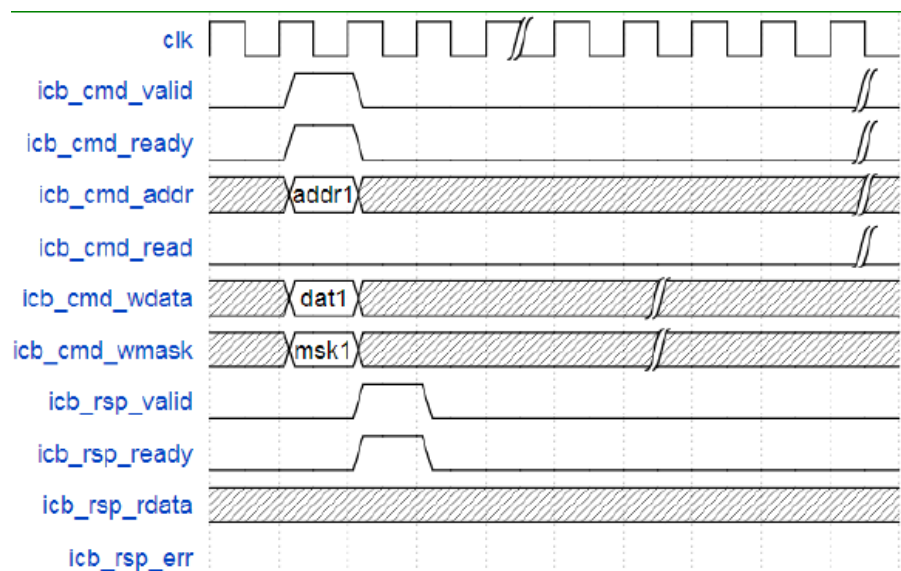
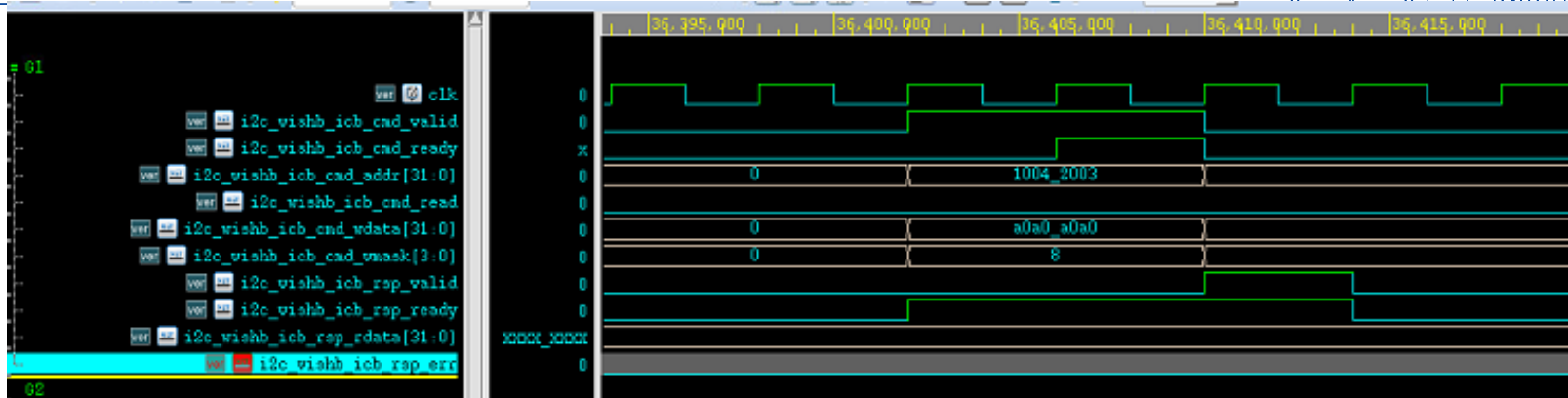
## 写时序2



- 主设备向从设备通过ICB的Command Channel发送写操作请求（icb\_cmd\_read为低），从设备立即接收该请求（icb\_cmd\_ready为高）。从设备在四个周期后返回结果且结果正确（icb\_rsp\_err为低），主设备立即接收该结果（icb\_rsp\_ready为高）。

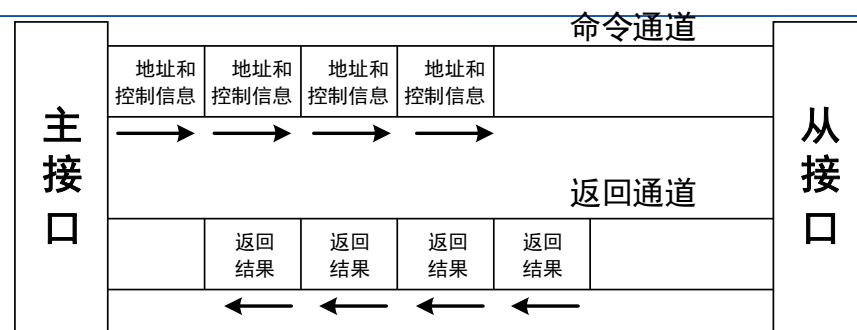


# I2C模块的写时序





## ICB写时序



通道	功能	方向	宽度	信号名	介绍
命令通道	主设备向从设备发起读写请求	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
			DW	icb cmd addr	读写地址
			1	icb cmd read	读或是写操作的指示
			DW	icb cmd wdata	写操作的数据
			DW/8	icb cmd wmask	写操作的字节掩码
反馈通道	从设备向主设备返回读写结果	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
			1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
			DW	icb_rsp_rdata	读反馈的数据
			1	icb_rsp_err	读或者写反馈的错误标志
		Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号

## VALID/READY握手机制

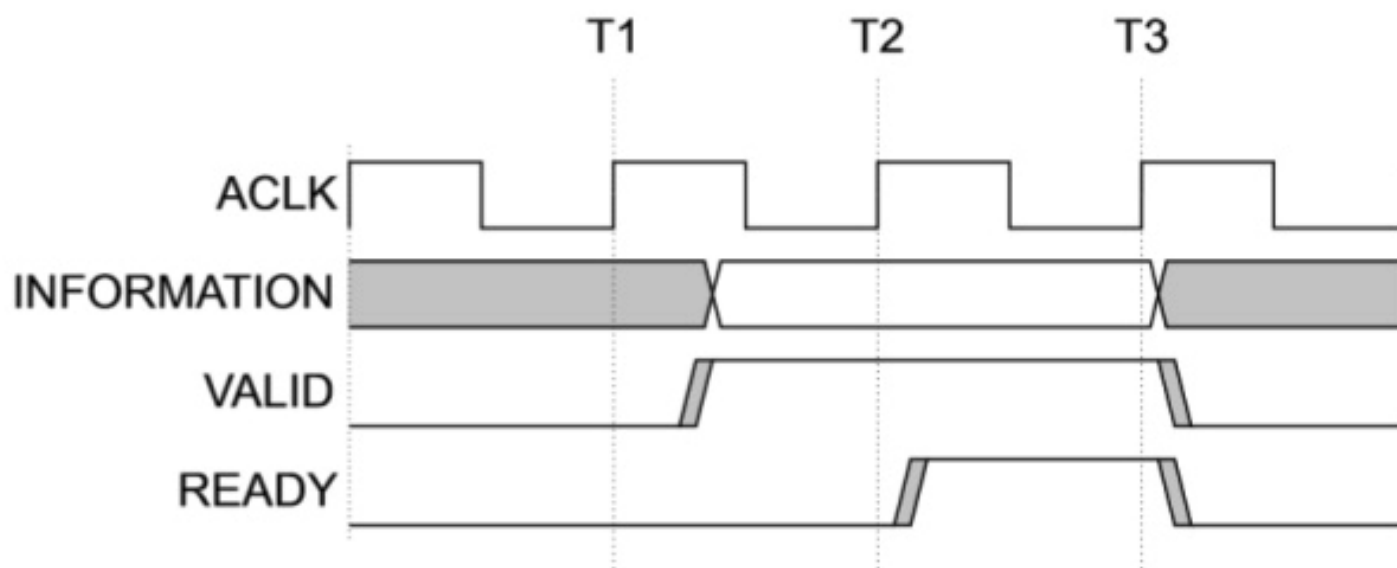


- 作为一种双向流控机制，VALID/READY 机制可以使发送接收双方都有能力控制传输速率。
- 发送方置高 VALID 信号表示发送方已经将数据，地址或者控制信息放到写的总线上，并保持。
- 接收方置高 READY 信号表示接收方已经做好接收的准备。
- 所谓的双向流控机制，指的是发送方通过 VALID 信号置起控制发送速度的同时，接收方也可以通过 READY 信号的置起与否控制接收速度，反压发送方的发送速度。
- 当双方的信息同时为高，时钟上升沿到达后，一次数据传输完成，在 1 到 n 次时钟上升沿后，双方传完了要传的信息后，两信号同时拉低。

## VALID 信号先到达



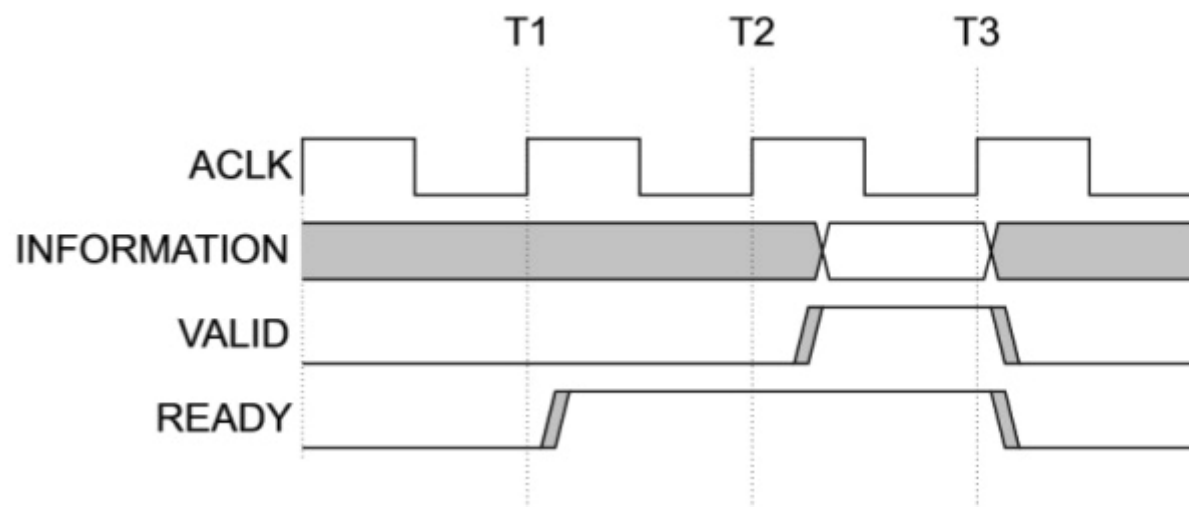
- 发送方 VALID 信号有效，接收方还处于忙状态，过了 T2 才到达，T3 时刻传输完成。VALID 信号一旦置起就不能拉低，直到此次传输完成。对于接收方来说，检测到 VALID 信号置起，如果系统正忙，可以让发送方等待，发送方在完成传输之前都不会置低 VALID 信号，



## READY 信号先到达



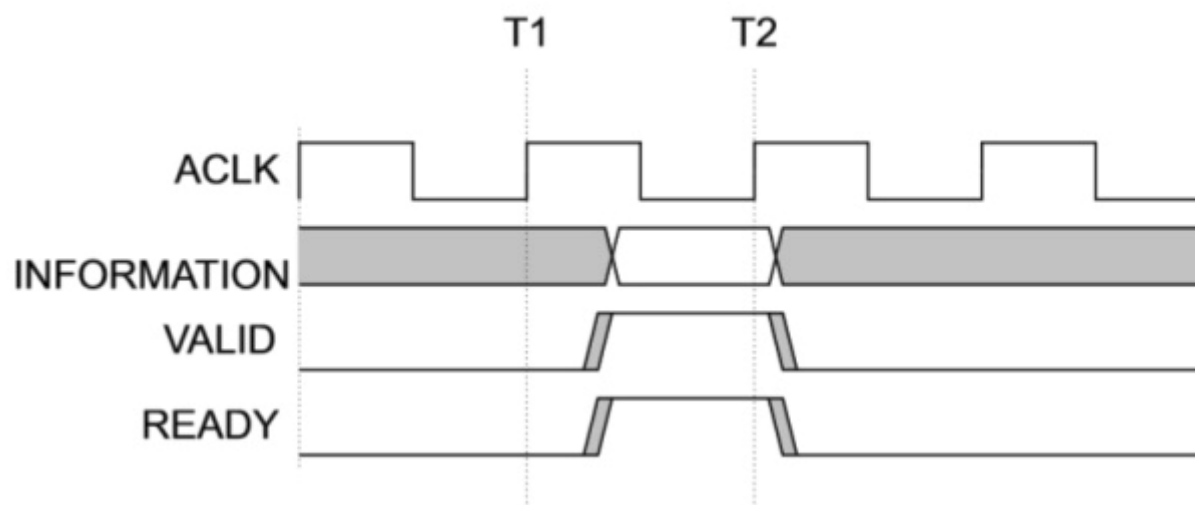
- READY 可以等待 VALID 信号到来再做响应，但也可以在 VALID 信号到来前就置高，表示接收端已经做好准备了。



## 同时到达



- 等到下一个时钟上升沿 T2，传输完成，一个时钟周期里就完成。



# Ready信号产生示例

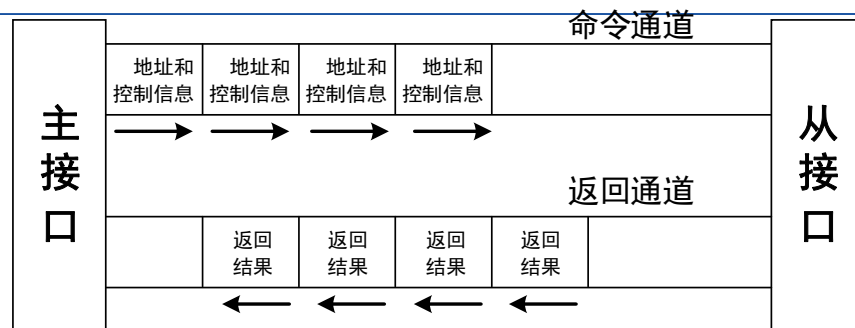


```
assign wb_stb          = (~i_icb_rsp_valid) & i_icb_cmd_valid;  
assign wb_cyc          = (~i_icb_rsp_valid) & i_icb_cmd_valid;  
assign i_icb_cmd_ready = (~i_icb_rsp_valid) & wb_ack;
```

```
always @(posedge wb_clk_i or negedge rst_i)  
//always @(posedge wb_clk_i) //Bob: Here the ack is X by default, so add the rst here  
if (!rst_i)  
    wb_ack_o <= #1 1'b0;  
else  
    wb_ack_o <= #1 wb_cyc_i & wb_stb_i & ~wb_ack_o; // because timing is always honored
```



## ICB写时序 (2)

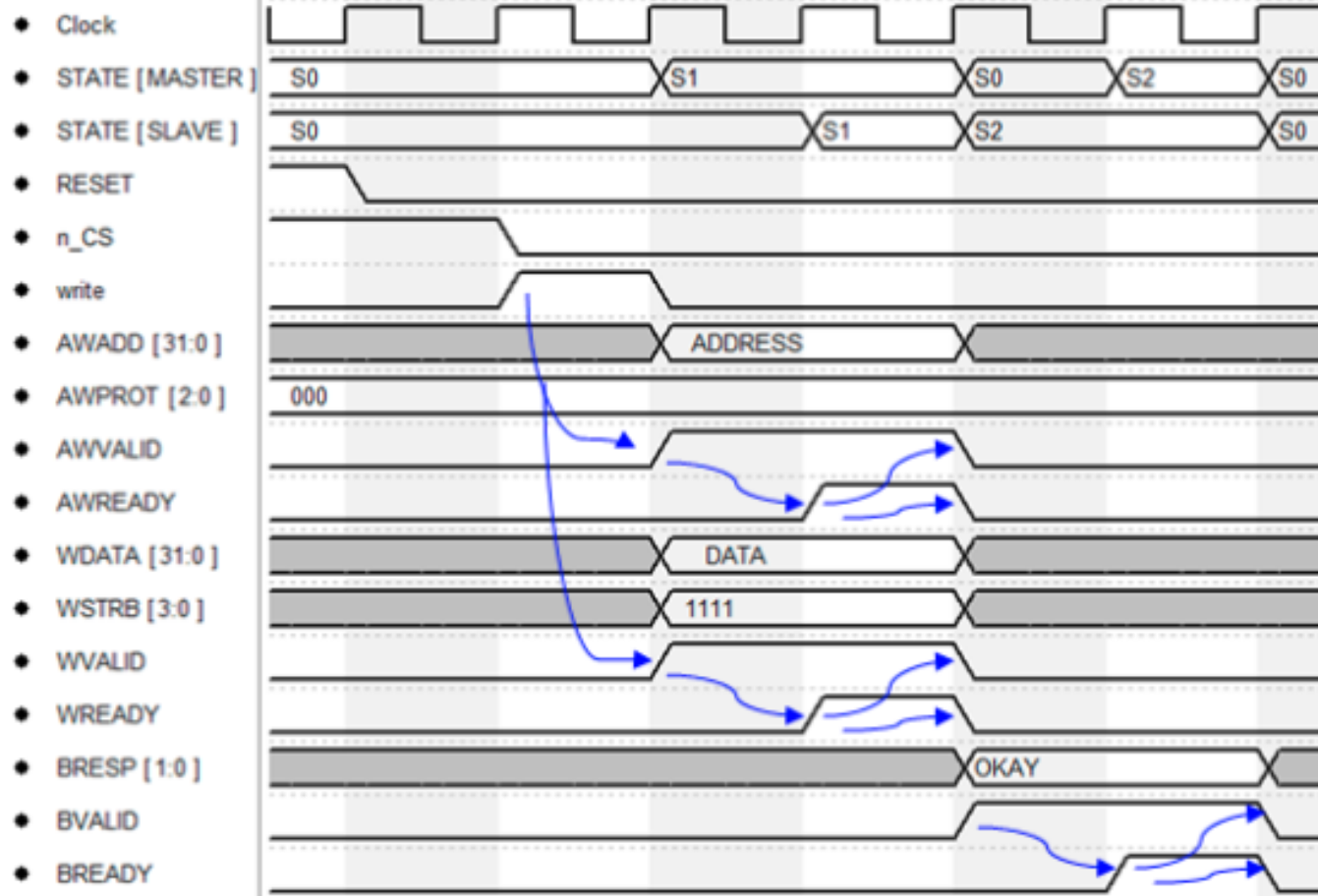


通道	功能	方向	宽度	信号名	介绍
命令通道	主设备向从设备发起读写请求	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
			DW	icb cmd addr	读写地址
			1	icb cmd read	读或是写操作的指示
			DW	icb cmd wdata	写操作的数据
			DW/8	icb cmd wmask	写操作的字节掩码
反馈通道	从设备向主设备返回读写结果	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
			1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
			DW	icb_rsp_rdata	读反馈的数据
			1	icb_rsp_err	读或者写反馈的错误标志
		Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号

# 完整的handshake



## Write transaction:



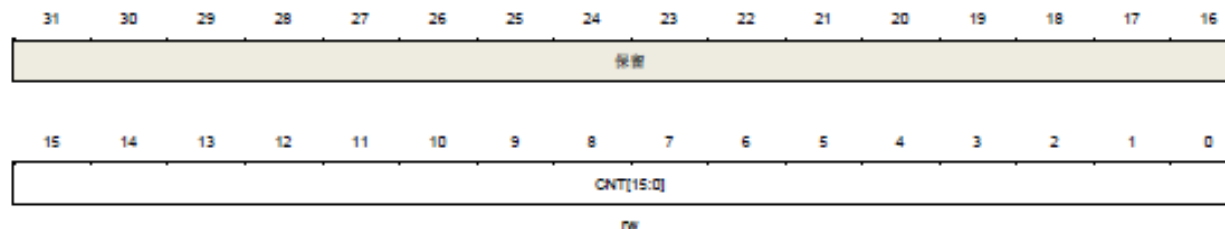
# 寄存器读写操作(CNN Register)



$x = 0 \dots 6$ ,  $x$  为通道序号

地址偏移:  $0x0C + 0x14 \times x$

复位值:  $0x0000\ 0000$



```
reg [?:0] cnt_reg
always @(posedge clk or negedge rst_n)
if(!rst_n)
...
else if(?)
...
else
```

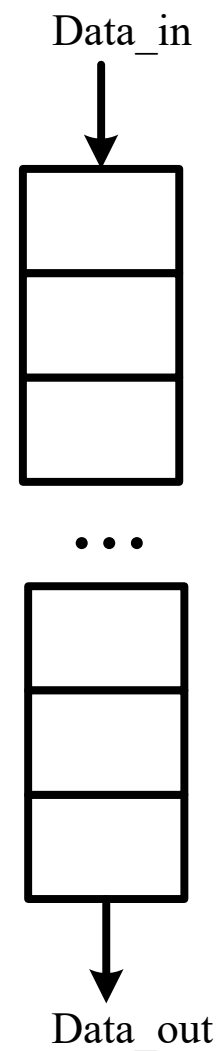
??



# FIFO



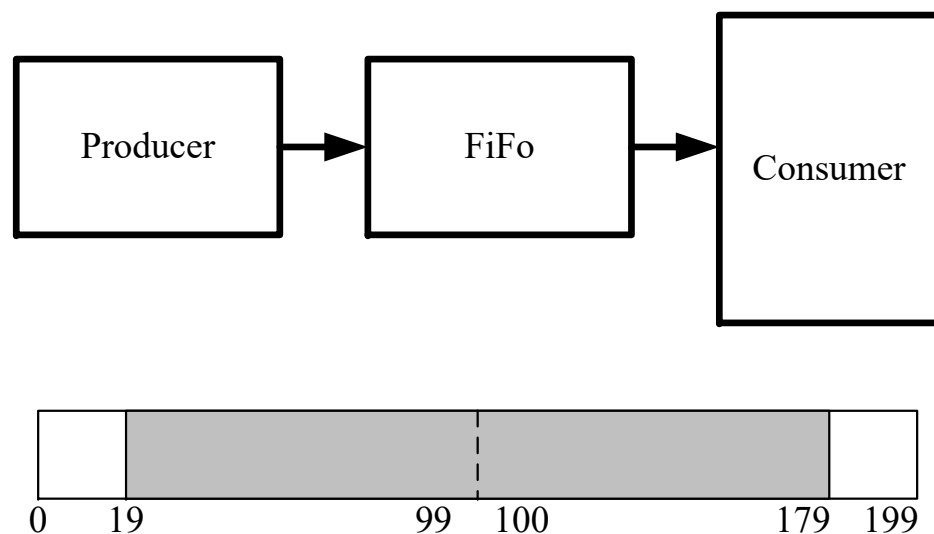
- FIFO(First IN First Out)先进先出电路，是一种实现数据先进先出的存储器件。
- 用途：用作数据缓冲器，广泛应用与模块之间，或者处理器之间的数据通信。
- 在异步通信中几乎不可避免地要使用FIFO。



# FIFO应用举例



- 一个模块中，如果数据产生模块，100个写时钟周期内，可以随机连续写入80个数据，而数据消费模块每10个读时钟可以读出8个数据。当 $wclk = rclk$  时，需要多大的fifo实现数据连续传输？



# FIFO建模步骤

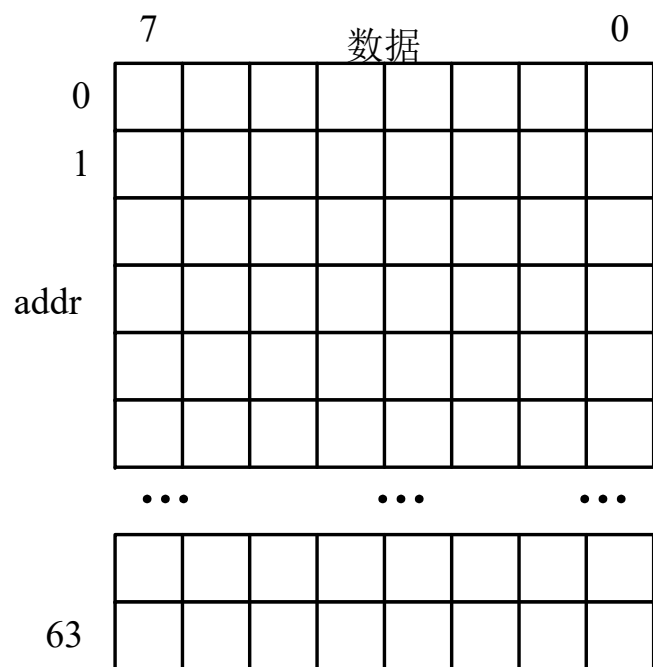
---



- 确定端口与功能
- 建立存储器模型
- 设计空满标志电路



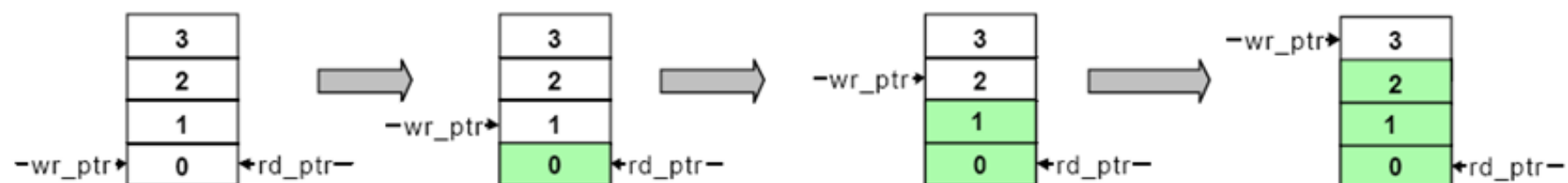
# 存储器



`reg [msb:lsb] memory [upper1:lower1];`

`reg [7:0] memory1 [0:63]; //64个8位的存储器`

# FIFO空满标志的产生



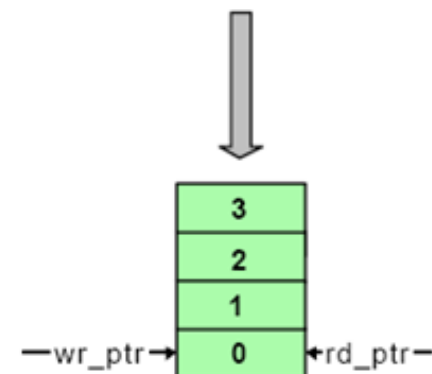
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

?	match	
---	-------	--

0	1	0
1	1	0

rd\_ptr

wr\_ptr



# Fifo 参考代码



```
always @(posedge clk)
if(w_en && !full)
    fifo_mem[wr_ptr[3:0]]<=data_w;
else
    fifo_mem[wr_ptr[3:0]]<=fifo_mem[wr_ptr[3:0]];

always @(posedge clk or negedge rst_n)
if(!rst_n)
    data_r<=16'b0;
else if(r_en && !empty)
    data_r<=fifo_mem[rd_ptr[3:0]];
else
    data_r<=data_r;

always @(posedge clk or negedge rst_n)
if(!rst_n)
    wr_ptr<=5'b0;
else if(w_en && !full)
    wr_ptr<=wr_ptr+1'b1;
else
    wr_ptr<=wr_ptr;

always @(posedge clk or negedge rst_n)
if(!rst_n)
    rd_ptr<=5'b0;
else if(r_en && !empty)
    rd_ptr<=rd_ptr+1'b1;
else
    rd_ptr<=rd_ptr;

assign full=(rd_ptr[3:0]==wr_ptr[3:0]) && (rd_ptr[4] != wr_ptr[4]);
wire empty=(rd_ptr[4:0] == wr_ptr[4:0]);
```

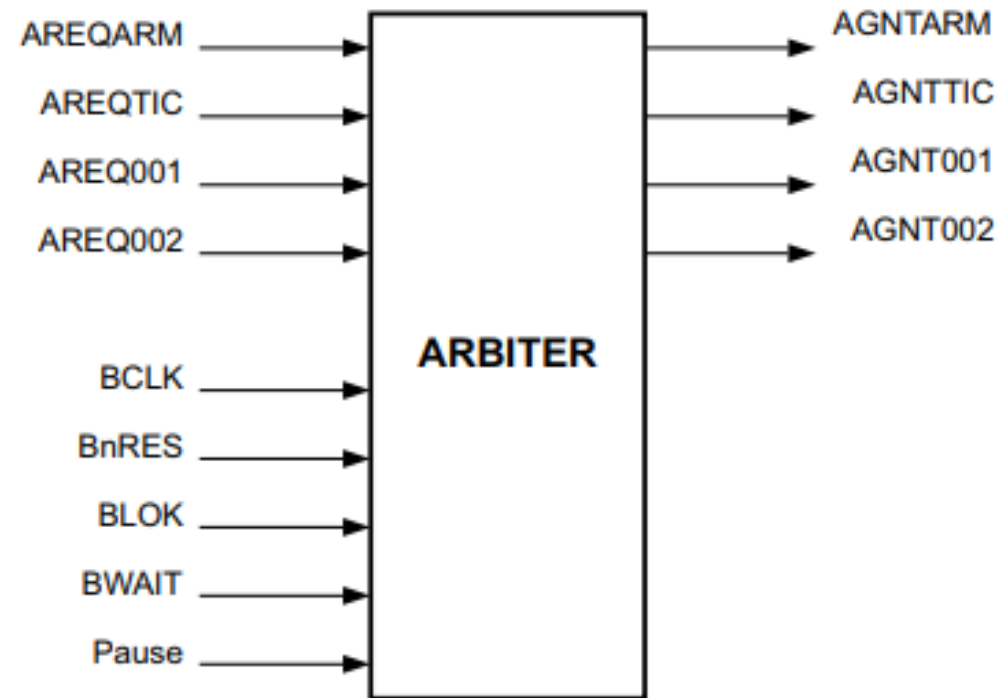
# Fifo其中描述方法



- 设置cnt
- 只读cnt-1
- 只写cnt+1



# arbiter



# 蜂鸟系统中的arbiter



```

module sirv_gnrl_icb_arbt # (
    parameter AW = 32,
    parameter DW = 64,
    parameter USR_W = 1,
    parameter ARBT_SCHEME = 0, // 0: priority based, 1: rrobin
    // The number of outstanding transactions supported
    parameter FIFO_OUTS_NUM = 1,
    parameter FIFO_OUT_READY = 0,
    // ARBT_NUM=4 ICB ports, so 2 bits for port id
    parameter ARBT_NUM = 4,
    parameter ALLOW_OCYCL_RSP = 1,
    parameter ARBT_PTR_W = 2
) (
    output                o_icb_cmd_valid,
    input                 o_icb_cmd_ready,
    output [1-1:0]        o_icb_cmd_read,
    output [AW-1:0]        o_icb_cmd_addr,
    output [DW-1:0]        o_icb_cmd_wdata,
    output [DW/8-1:0]      o_icb_cmd_wmask,
    output [2-1:0]         o_icb_cmd_burst,
    output [2-1:0]         o_icb_cmd_beat,
    output                o_icb_cmd_lock,
    output                o_icb_cmd_excl,
    output [1:0]           o_icb_cmd_size,
    output [USR_W-1:0]     o_icb_cmd_usr,

    input                 o_icb_rsp_valid,
    output                o_icb_rsp_ready,
    input                 o_icb_rsp_err,
    input                 o_icb_rsp_excl_ok,
    input [DW-1:0]        o_icb_rsp_rdata,
    input [USR_W-1:0]     o_icb_rsp_usr,

    output [ARBT_NUM+1-1:0] i_bus_icb_cmd_ready,
    input  [ARBT_NUM+1-1:0] i_bus_icb_cmd_valid,
    input  [ARBT_NUM+1-1:0] i_bus_icb_cmd_read,
    input  [ARBT_NUM+AW-1:0] i_bus_icb_cmd_addr,
    input  [ARBT_NUM+DW-1:0] i_bus_icb_cmd_wdata,
    input  [ARBT_NUM+DW/8-1:0] i_bus_icb_cmd_wmask,
    input  [ARBT_NUM+2-1:0] i_bus_icb_cmd_burst,
    input  [ARBT_NUM+2-1:0] i_bus_icb_cmd_beat,
    input  [ARBT_NUM+1-1:0] i_bus_icb_cmd_lock,
    input  [ARBT_NUM+1-1:0] i_bus_icb_cmd_excl,
    input  [ARBT_NUM+2-1:0] i_bus_icb_cmd_size,
    input  [ARBT_NUM+USR_W-1:0] i_bus_icb_cmd_usr,

    output [ARBT_NUM+1-1:0] i_bus_icb_rsp_valid,
    input  [ARBT_NUM+1-1:0] i_bus_icb_rsp_ready,
    output [ARBT_NUM+1-1:0] i_bus_icb_rsp_err,
    output [ARBT_NUM+1-1:0] i_bus_icb_rsp_excl_ok,
    output [ARBT_NUM+DW-1:0] i_bus_icb_rsp_rdata,
    output [ARBT_NUM+USR_W-1:0] i_bus_icb_rsp_usr,

    input clk,
    input rst_n
);
    
```

# burst



- dma有burst、burst size、transfer的概念
- Burst操作与单独的一次读写操作相比，burst只需要提供一个起始地址就行了，以后的地址依次加1，而非burst操作每次都要给出地址，以及需要中间的一些应答、等待状态等。
- 如果是对地址连续的读取，burst效率高得多，但如果地址是跳跃的，则无法采用burst操作。

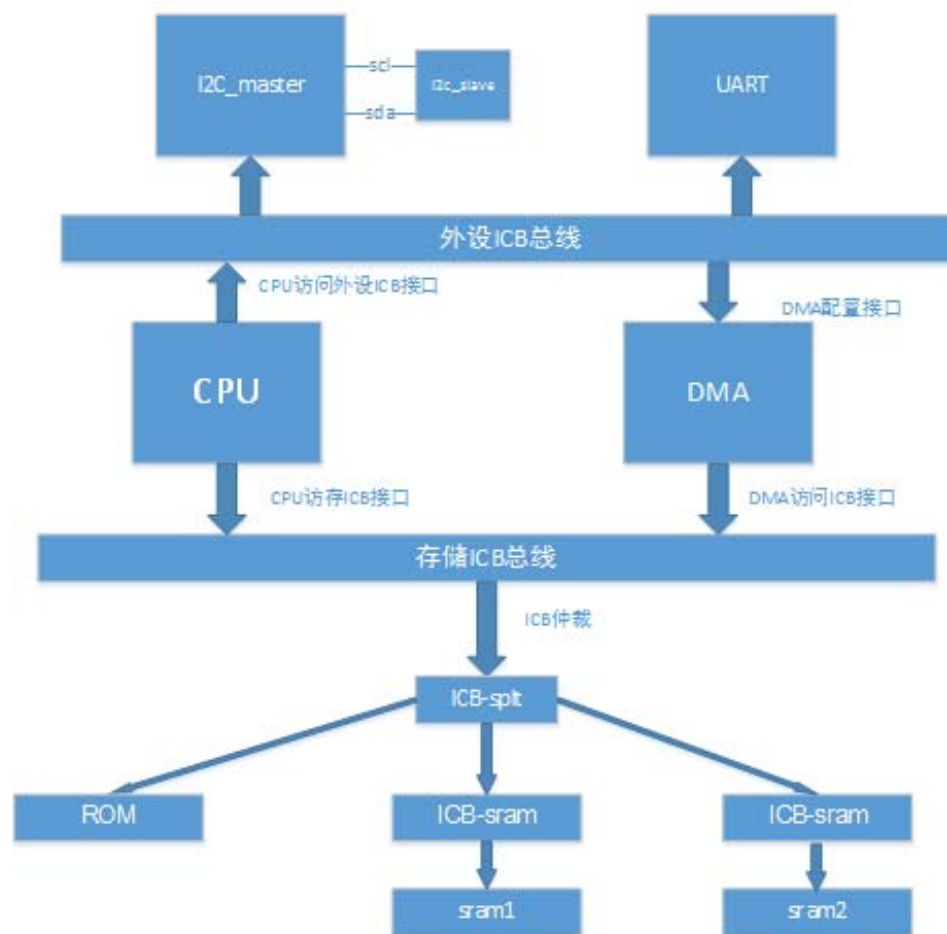


# Burst 概念



- dma实际上是一次一次的申请总线，把要传的数据总量分成一个一个小数据块。比如要传64个字节，那么dma内部可能分为2次，一次传 $64/2=32$ 个字节，这个2次呢，就叫做burst。这个burst是可以设置的。这32个字节又可以分为32位\*8或者16位\*16来传输。
- burst size: 就是一次传几个 transfer size.

# 集成



# 其他需要关注的代码



- ICB1to8bus

```
sirv_icb1to8_bus # (  
.ICB_FIFO_DP (2), // We add a ping-pong buffer here to cut down the timing path  
.ICB_FIFO_CUT_READY (1), // We configure it to cut down the back-pressure ready signal  
.AW (32),  
.DW ('E203_XLEN),  
.SPLT_FIFO_OUTS_NUM (1), // The Mem only allow 1 outstanding  
.SPLT_FIFO_CUT_READY (1), // The Mem always cut ready  
// * rom : 0x0000 1000 -- 0x0000 1FFF  
.00_BASE_ADDR ('ROM_BASE_ADDR),  
.00_BASE_REGION_LSB (12),  
// * sram1 : 0x2000_0000 -- 0x2000 FFFF  
.01_BASE_ADDR ('SRAM1_BASE_ADDR),  
.01_BASE_REGION_LSB (16),  
// sram2 : 0x3000 0000 -- 0x3000 FFFF  
.02_BASE_ADDR ('SRAM2_BASE_ADDR),  
.02_BASE_REGION_LSB (16),  
// not used  
.03_BASE_ADDR (32'h0000_0000),  
.03_BASE_REGION_LSB (0),  
// Not used  
.04_BASE_ADDR (32'h8000_0000),  
.04_BASE_REGION_LSB (0),  
  
// Not used  
.05_BASE_ADDR (32'h4000_0000),  
.05_BASE_REGION_LSB (0),  
  
// Not used  
.06_BASE_ADDR (32'h0000_0000),  
.06_BASE_REGION_LSB (0),  
  
// Not used  
.07_BASE_ADDR (32'h0000_0000),  
.07_BASE_REGION_LSB (0)  
)u_sirv_mem_fab(  
)
```

谢谢！

