FPGA&ASIC笔面试题船新版本

FPGA&ASIC基本开发流程

题目：简述ASIC设计流程，并列举出各部分用到的工具。

ASIC开发基本流程

题目：简述FPGA的开发流程。

FPGA开发基本流程

题目：名词解释：

数字电路基础

题目：bit, byte, word, dword, qword的区别

题目：什么是原码，反码，补码，符号-数值码。以8bit为例，给出各自表示的数值范围

题目：数制转换

题目：逻辑函数及其化简

题目：什么是冒险和竞争，如何消除?

题目：设计一个2-4译码器。

题目：设计BCD译码器，输入0~9。

题目：MOS逻辑门

题目：用D触发器带同步高置数和异步高复位端的二分频的电路，画出逻辑电路，Verilog描述。

题目：CMOS反相器的功耗主要包括哪几部分？分别与哪些因素相关?

题目：transition time, propagation delay等参数的定义

题目：ASIC中低功耗的设计方法和思路（不适用于FPGA）

题目：输入一个8bit数，输出其中1的个数。如果只能使用1bit全加器，最少需要几个?

时序逻辑电路基础

题目：简述建立时间和保持时间，作图说明

题目:说明D触发器与Latch的区别。

题目：什么是同步电路和异步电路。

题目：最小周期计算

题目：什么是Clock Jitter和Clock Skew，这两者有什么区别。

题目：什么是亚稳态，产生的原因，如何消除?

题目：同步和异步

题目：reg和wire的区别

题目：阻塞赋值与非阻塞赋值的区别

题目：localparam、parameter和define的区别

题目：task与function的区别

题目：谈谈对Retiming技术的理解

题目：什么是高阻态

题目：解释一下亚稳态。

RTL代码

题目：多时钟域设计中，如何处理跨时钟域

题目：编写Verilog代码描述跨时钟域信号传输，慢时钟域到快时钟域

题目：编写Verilog代码描述跨时钟域信号传输，快时钟域到慢时钟域

题目：用Verilog实现1bit信号边沿检测功能，输出一个周期宽度的脉冲信号。

题目： 用Verilog实现glitch free时钟切换电路。输入sel，clka，clkb，sel为1输出clka，sel为0输出clkb。

题目：用Verilog实现串并转换

题目：用verilog实现串并变换。

题目：用verilog实现一个4bit二进制计数器。

题目：用verilog实现4bit约翰逊(Johnson)计数器。

题目： 用verilog实现4bit环形计数器：复位有效时输出0001，复位释放后依次输出0010，0100，1000，0001，0010...

题目：用verilog实现PWM控制呼吸灯。呼吸周期2秒：1秒逐渐变亮，1秒逐渐变暗。系统时钟24MHz，pwm周期1ms，精度1us。

题目：序列检测器：有“101”序列输入时输出为1，其他输入情况下，输出为0。画出状态转移图，并用Verilog描述。

题目：用Verilog实现一个异步双端口ram，深度16，位宽8bit。A口读出，B口写入。支持片选，读写请求，要求代码可综合。

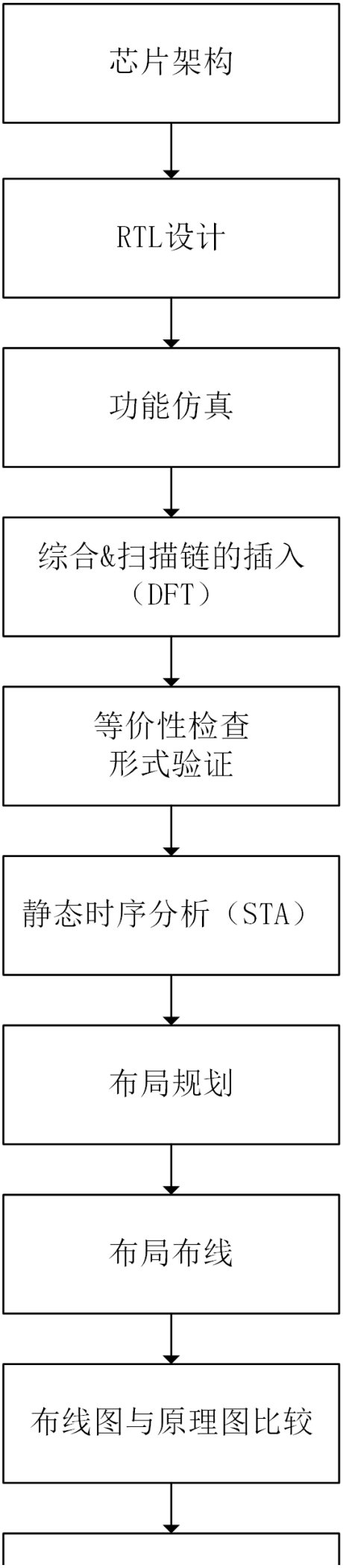
其中有错误或笔误的如果发现，请发送至邮箱

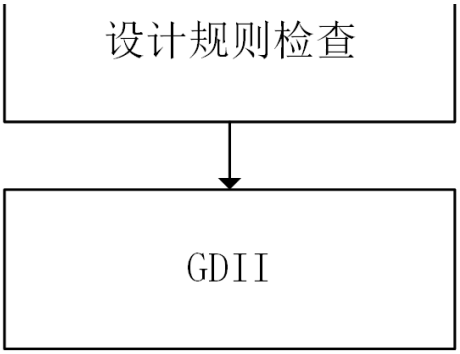
ninghechuan@foxmail.com，我会更新改正。

FPGA&ASIC笔面试题船新版本

FPGA&ASIC基本开发流程

题目：简述ASIC设计流程，并列举出各部分用到的工具。


芯片架构
Fabless
RTL设计
Verilog HDL
VHDL
System Verilog
Vim、Emacs
功能仿真
Modelsim\Questsim Mentor公司
VCS Synopsys公司
NC-Verilog Cadence公司
UVM
综合&扫描链的插入
（DFT）
Synplify Synopsys公司
Design Compile Synopsys公司
DFT Compile Synopsys公司
等价性检查
形式验证
Formality Synopsys公司
Conformal Cadence公司
静态时序分析（STA）
Prime Time Synopsys公司
布局规划
布局布线
IC Compile Synopsys公司
布线图与原理图比较
Dracula Cadence公司



ASIC开发基本流程

芯片架构，考虑芯片定义、工艺、封装

RTL设计，使用Verilog、System Verilog、VHDL进行描述

功能仿真，理想情况下的仿真

验证，UVM验证方法学、FPGA原型验证

综合，逻辑综合，将描述的RTL代码映射到基本逻辑单元门、触发器上

DFT技术，插入扫描链

等价性检查，使用形式验证技术

STA，静态时序分析

布局规划，保证没有太多的内部交互，避免布线上的拥堵和困扰

时钟树综合，均匀地分配时钟，减少设计中不同部分间的时钟偏移

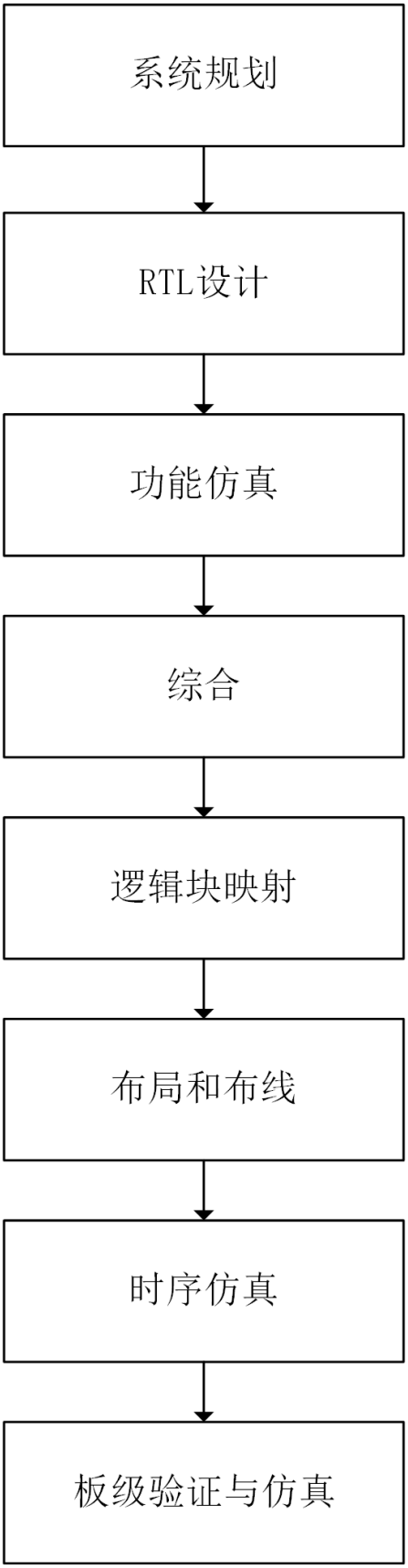
DRC，设计规则检查

LVS，布线图和原理图进行比较

生成GDSII

这整个流程称为RTL2GDSII，利用GDSII来生产芯片的过程称作流片（Tapeout），以上是一个Fabless公司的简易设计流程，最后将GDSII送至Foundry生产芯片。

题目：简述FPGA的开发流程。



Verilog HDL
VHDL
System Verilog
Vim、Notepad++

Modelsim\Questsim Mentor公司
Xsim\Isim Xilinx公司

ISE、Vivado Xilinx公司
QuartusII Intel公司

FPGA开发基本流程

系统规划，系统功能，功能模块划分

RTL设计，使用Verilog、System Verilog、VHDL进行描述

功能仿真，理想情况下的仿真

综合、编译、布局布线，FPGA厂商自带工具完成

时序仿真，时序分析约束

板级验证

题目：名词解释：

ROM：**R**ead **O**nly **M**emory，只读存储器，手机、计算机等设备的存储器，但现在的所说的ROM不只是Read Only了，也是可以写入的。

RAM：**R**andom **A**ccess **M**emory，随机存取存储器，手机、计算机的运行内存。

SRAM：**S**tatic **R**andom-**A**ccess **M**emory，静态随机存取存储器，只要供电数据就会保持，但断电数据就会消失，也被称为Volatile Memory

DRAM：**D**ynamic **R**andom **A**ccess **M**emory，动态随机存储器，主要原理是利用电容存储电荷的多少来代表一个bit是0还是1，由于晶体管的漏电电流现象，电容会放电，所以要周期性的给电容充电，叫刷新。SRAM不需要刷新也会保持数据丢失，但是两者断电后数据都会消失，称为Volatile Memory

SDRAM：**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory，同步动态随机存储器，同步写入和读出数据的DRAM。

EEPROM：**E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory，电可擦除可编程只读存储器，

DDR：**D**ouble **D**ata **S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory，双倍速率同步动态随机存储器，双倍速率传输的SDRAM，在时钟的上升沿和下降沿都可以进行数据传输。我们电脑的内存条都是DDR芯片。

FLASH： Flash Memory，闪存，非易失性固态存储，如制成内存卡或U盘。

数字电路基础

题目：bit, byte, word, dword, qword的区别

1byte = 8bit

1word = 2byte = 16bit

1dword = 2word = 4byte = 32bit

1qword = 2dword = 4word = 8byte = 64bit

题目：什么是原码，反码，补码，符号-数值码。以8bit为例，给出各自表示的数值范围

原码：符号位+真值，最高位表示符号位，以8bit为例。

[+3]原 = 0000_0011

[-3]原 = 1000_0011

表示范围：-127到+127

原码中0000和1000都表示0。

反码：正数的反码是它本身，负数的反码将原码除符号位外逐位取反。以8bit为例。

[+3]原 = [0000_0011]原 = [0000_0011]反

[-3]原 = [1000_0011]原 = [1111_1100]反

表示范围：-127到+127

反码中0000_0000和1111_1111都表示0。

补码：正数的补码是它本身，负数的补码将原码除符号位外逐位取反再加1。以8bit为例。

[+3]原 = [0000_0011]原 = [0000_0011]反 = [0000_0011]补

[-3]原 = [1000_0011]原 = [1111_1100]反 = [1111_1101]补

表示范围：-128到+127

补码中0的表示只有一种形式，即0000_0000，1000_0000表示-128。

以上是有符号数，对于无符号数来说都是来表示整数，其原码、反码、补码都是其本身。

更详细解释可参考维基百科。

https://zh.wikipedia.org/wiki/%E6%9C%89%E7%AC%A6%E8%99%9F%E6%95%B8%E8%99%95%E7%90%86

题目：数制转换

R进制数转换为十进制数：按权展开，相加

十进制数转化为R进制数：整数部分，除R取余法，除到商为0为止。小数部分，乘R取整法，乘到积为0为止。

二进制数转化八进制数：三位一组，整数部分左边补0，小数部分右边补0。反之亦然。

二进制数转化十六进制数：四位一组，整数部分左边补0，小数部分右边补0。反之亦然。

127 -127 127.375 -127.375 十进制数转化为R进制数：整数部分，除R取余法，除到商为0为止。小数部分，乘R取整法，乘到积为1为止。

127 = 0111_1111

-127 = 1111_1111

127.375 = 0111_1111.011

-127.375 = 1111_1111.011

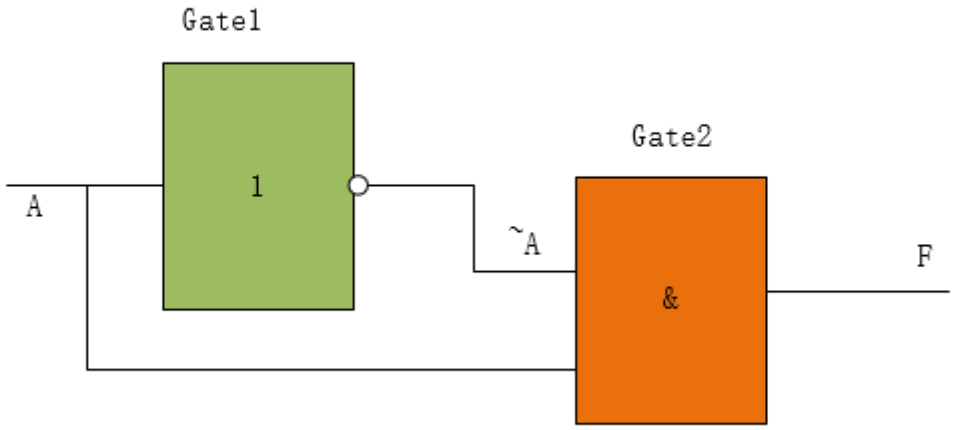
题目：逻辑函数及其化简

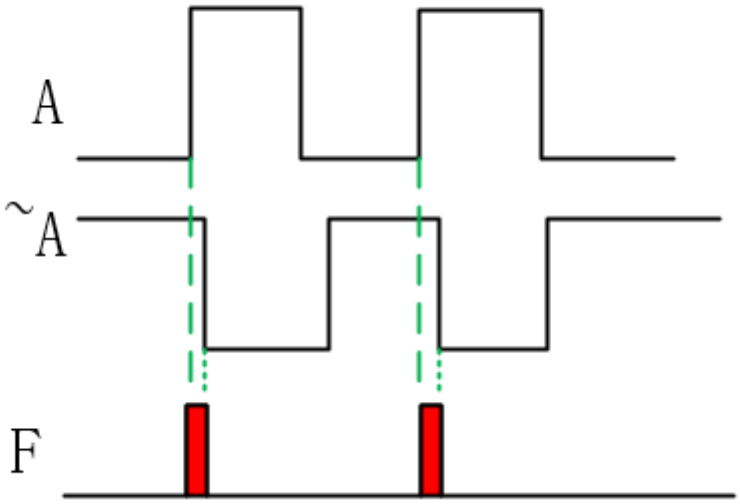
公式法

卡诺图法

题目：什么是冒险和竞争，如何消除?

下面这个电路，使用了两个逻辑门，一个非门和一个与门，本来在理想情况下F的输出应该是一直稳定的0输出，但是实际上每个门电路从输入到输出是一定会有时间延迟的，这个时间通常叫做电路的开关延迟。而且制作工艺、门的种类甚至制造时微小的工艺偏差，都会引起这个开关延迟时间的变化。





实际上如果算上逻辑门的延迟的话，那么F最后就会产生毛刺。信号由于经由不同路径传输达到某一汇合点的时间有先有后的现象，就称之为竞争，由于竞争现象所引起的电路输出发生瞬间错误的现象，就称之为冒险，FPGA设计中最简单的避免方法是尽量使用时序逻辑同步输入输出。

- 加滤波电容，消除毛刺的影响
- 加选通信号，避开毛刺
- 增加冗余项，消除逻辑冒险。

题目：设计一个2-4译码器。

```
module Decode_2_4(
    input       [1:0]   indata,
    input               enable_n,
    //output  reg [3:0]   outdata
    output      [3:0]   outdata
);

/*
always @(*)begin
    if(enable_n == 1'b1)
        outdata = 4'b1111;
    else begin
        case(indata)
```

```
        2'b00: outdata = 4'b1110;
        2'b01: outdata = 4'b1101;
        2'b10: outdata = 4'b1011;
        2'b11: outdata = 4'b0111;
        endcase
    end
end
*/

assign  outdata[3] = ~(indata[1] & indata[0] & ~enable_n);
assign  outdata[2] = ~(indata[1] & ~indata[0] & ~enable_n);
assign  outdata[1] = ~(~indata[1] & indata[0] & ~enable_n);
assign  outdata[0] = ~(~indata[1] & ~indata[0] & ~enable_n);

endmodule
```

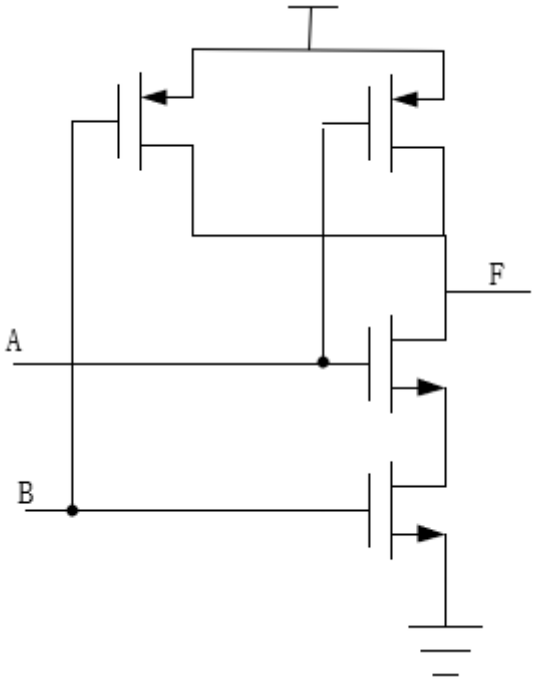
题目：设计BCD译码器，输入0~9。

BCD译码器也称为4-10线译码器

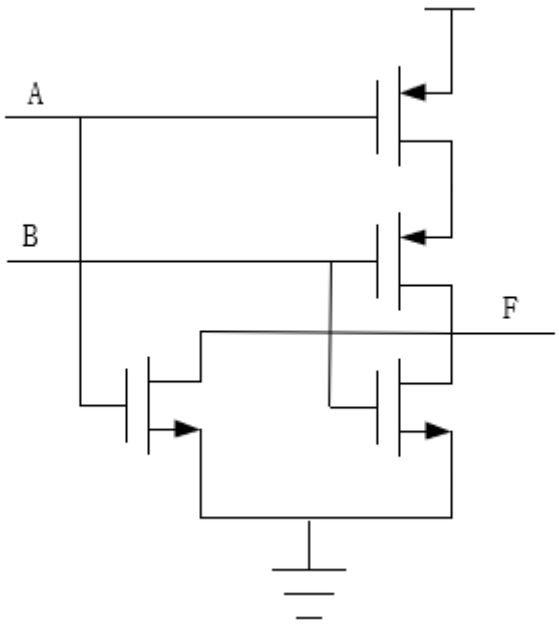
```
module Decode_4_10(
    input   [3:0]   indata,
    //output reg [9:0]   outdata
    output   [9:0]   outdata
    );
/*
always @(*)begin
    case(indata)
    4'b0000: outdata = 10'b1111_1111_10;
    4'b0001: outdata = 10'b1111_1111_01;
    4'b0010: outdata = 10'b1111_1110_11;
    4'b0011: outdata = 10'b1111_1101_11;
    4'b0100: outdata = 10'b1111_1011_11;
    4'b0101: outdata = 10'b1111_0111_11;
    4'b0110: outdata = 10'b1110_1111_11;
    4'b0111: outdata = 10'b1101_1111_11;
    4'b1000: outdata = 10'b1011_1111_11;
    4'b1001: outdata = 10'b0111_1111_11;
    default: outdata = 10'b1111_1111_11;
    endcase
end
*/
assign outdata[0] = ~(~indata[3] & ~indata[2] & ~indata[1] & ~indata[0]);
assign outdata[1] = ~(~indata[3] & ~indata[2] & ~indata[1] & indata[0]);
assign outdata[2] = ~(~indata[3] & ~indata[2] & indata[1]  & ~indata[0]);
assign outdata[3] = ~(~indata[3] & ~indata[2] & indata[1]  & indata[0]);
assign outdata[4] = ~(~indata[3] & indata[2]  & ~indata[1] & ~indata[0]);
assign outdata[5] = ~(~indata[3] & indata[2]  & ~indata[1] & indata[0]);
assign outdata[6] = ~(~indata[3] & indata[2]  & indata[1]  & ~indata[0]);
assign outdata[7] = ~(~indata[3] & indata[2]  & indata[1]  & indata[0]);
assign outdata[8] = ~(indata[3]  & ~indata[2] & ~indata[1] & ~indata[0]);
assign outdata[9] = ~(indata[3]  & ~indata[2] & ~indata[1] & indata[0]);

endmodule
```

题目：MOS逻辑门

与非门：上并下串（上为PMOS，下为NMOS）

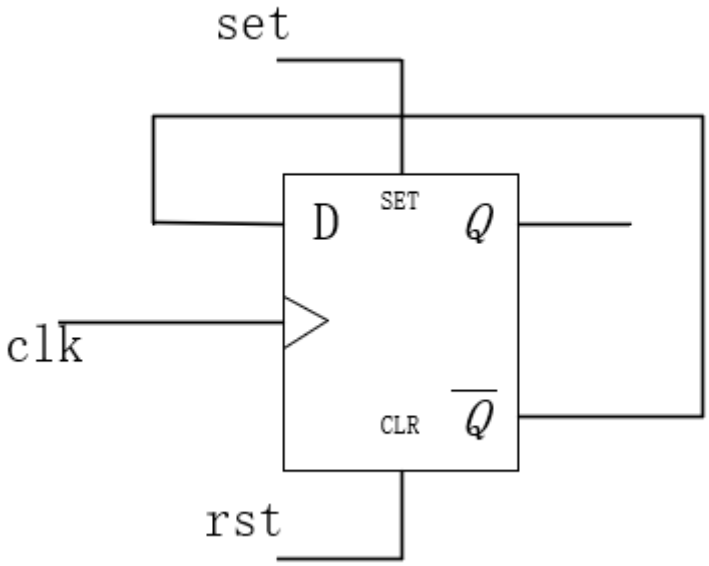


或非门：上串下并



反相器

题目：用D触发器带同步高置数和异步高复位端的二分频的电路，画出逻辑电路，Verilog描述。




```
reg     Q;
always @(posedge clk or posedge rst)begin
if(rst == 1'b1)
   Q <= 1'b0;

else if(set == 1'b1)
   Q <= 1'b1;

else
   Q <= ~Q;
end
```

题目：CMOS反相器的功耗主要包括哪几部分？分别与哪些因素相关?

$$P_{Total} = P_{dynamic} + P_{short} + P_{leakage}$$

- P_dynamic 是电路翻转产生的**动态功耗**
- P_short是P管和N管同时导通时产生的**短路功耗**
- P_leakage 是由扩散区和衬底之间的反向偏置漏电流引起的**静态功耗**

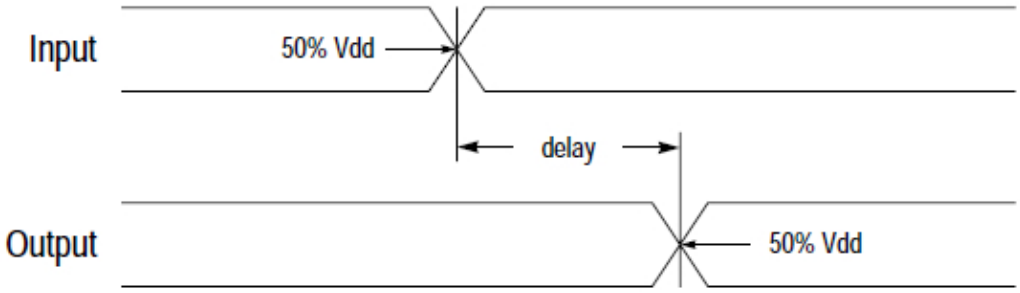
静态功耗：CMOS反相器在静态时，P、N管只有一个导通。由于没有Vdd到GND的直流桐庐，所以CMOS的静态功耗应该等于零。但实际上，由于扩散区和衬底的PN结上存在反向漏电流，所以会产生静态功耗。

短路功耗：CMOS电路在“0”和“1”的转换过程中，P、N管会同时导通，产生一个由Vdd到VSS窄脉冲电流，由此引起功耗

动态功耗：C_L 这个CMOS反相器的输出负载电容，由NMOS和PMOS晶体管的漏扩散电容、连线电容和扇出门的输入电容组成。

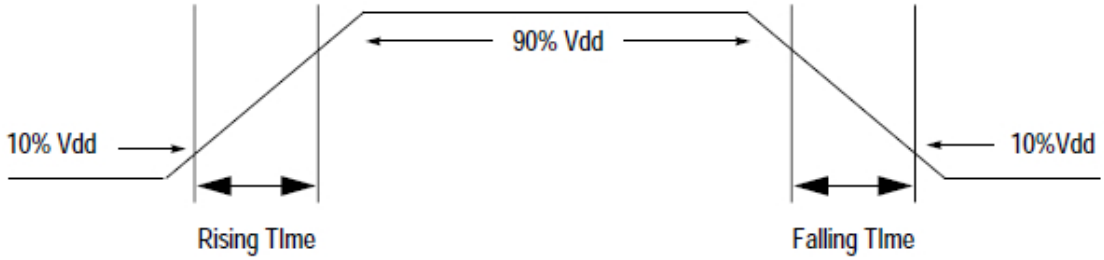
题目：transition time, propagation delay等参数的定义

Figure 1. Propagation Delay



Transition Time（转换时间）：输入和输出信号，上升时间：从10%Vdd上升到90%Vdd的时间，下降时间L从90%Vdd下降到10%Vdd的时间。上升时间和下降时间统称为Transition Time。

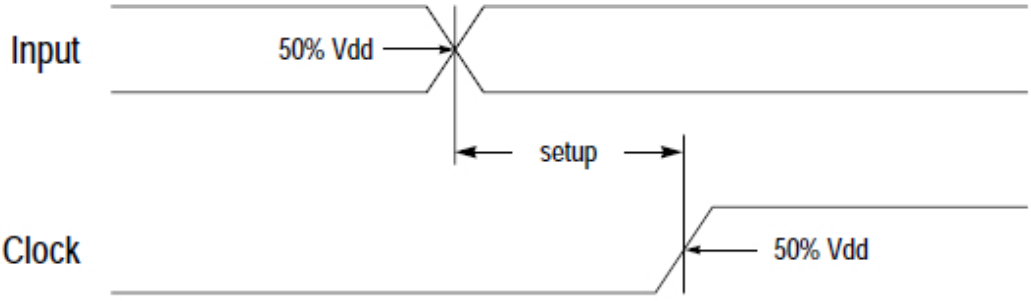
Figure 2. Transition Time



Propagation Delay（传播延时）：在输入信号变化到超过50%Vdd到输出信号变化到超过50%Vdd之间的时间。

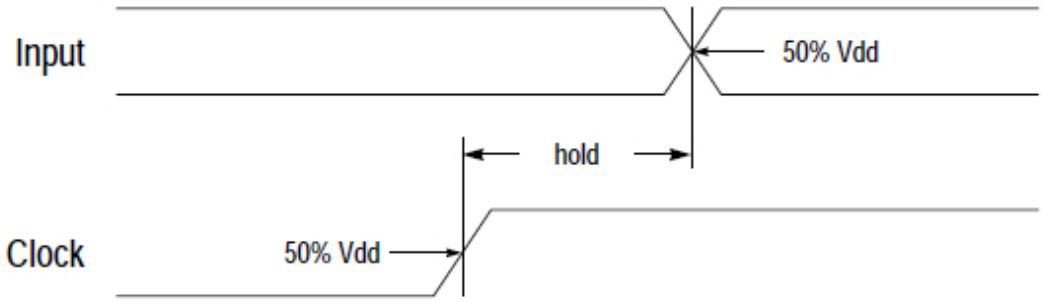
Timing constraints include: setup time, hold time, recovery time, and minimum pulse width.

Figure 3. Setup Time



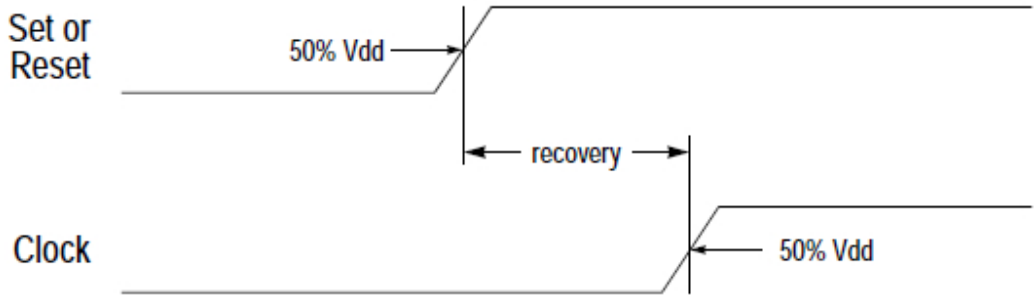
在时钟沿来临前，输入信号的变化超过50%Vdd的时间到时钟变化超过50%Vdd的时间中，输入信号保持稳定的最小时间。

Figure 4. Hold Time



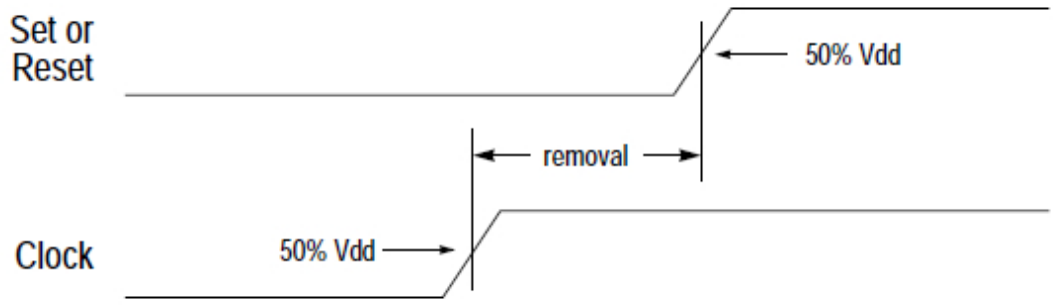
在时钟沿来临后，输入信号的变化超过50%Vdd的时间到时钟变化超过50%Vdd的时间中，输入信号保持稳定的最小时间。

Figure 5. Recovery Time



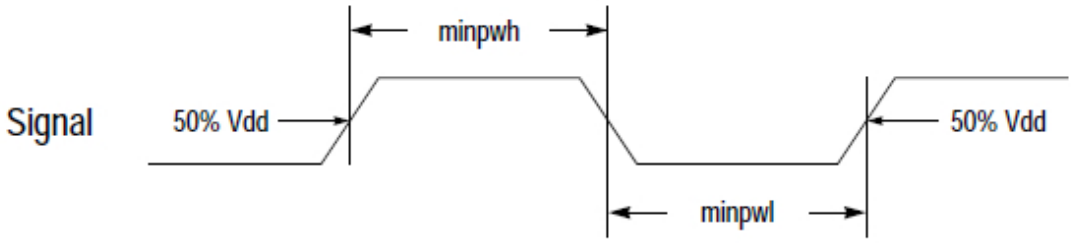
复位或者置位信号变化超过50%Vdd的时间到时钟变化超过50%Vdd的时间中，时钟沿来临的前最小时间，保证复位或置位完成。

Figure 6. Removal Time



复位或者置位信号变化超过50%Vdd的时间到时钟变化超过50%Vdd的时间中，时钟沿来临的后最小时间，保证置位或复位完成。

Figure 7. Minimum Pulse Width



最小脉冲宽度就是信号上升沿变化超过50%Vdd到下降沿变化低于50%Vdd时，测量高电平的最小脉冲宽度，低电平最小宽度同理。

个人认为不能保证各个时间参数，可能会产生亚稳态。

题目：ASIC中低功耗的设计方法和思路（不适用于FPGA）

- 合理规划芯片的工作模式，通过功耗管理模块控制芯片各模块的Clock，Reset起到控制功耗的目的。
- 门控时钟（Clockgateing）：有效降低动态功耗
- 多电压供电：通过控制模块的电压来降低功耗
- 多阈值电压

题目：输入一个8bit数，输出其中1的个数。如果只能使用1bit全加器，最少需要几个?

7个1bit全加器

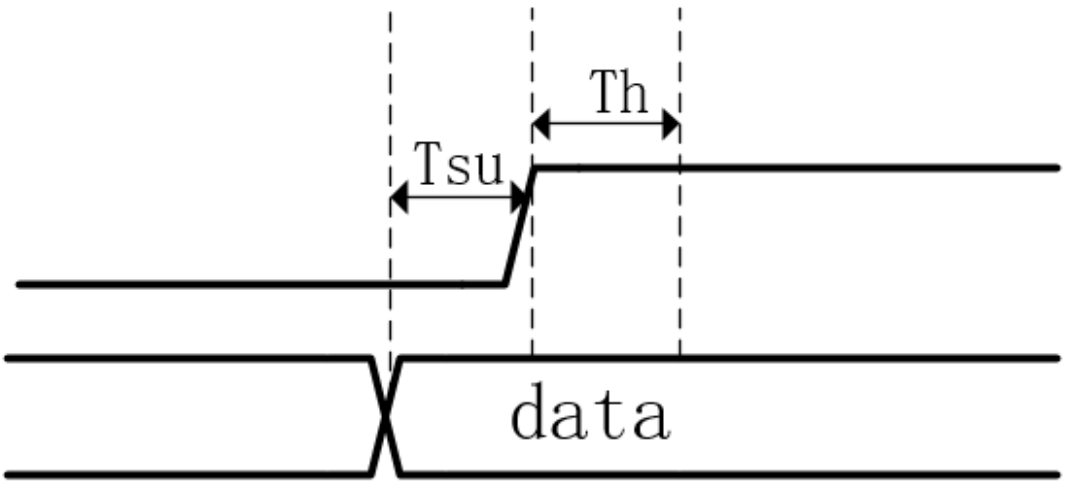
```
module number_one(
    input           clk,
    input           rst_n,
    input   [7:0]   din,
    output  [3:0]   num_one
);

wire [1:0]  sum0;
wire [1:0]  sum1;
wire [2:0]  sum2;

full_adder_one u0(
    .dina       (din[0]),
    .dinb       (din[1]),
    .cin        (din[2]),
    .sum        (sum0[0]),
    .cout       (sum0[1])
);

full_adder_one u1(
    .dina       (din[3]),
    .dinb       (din[4]),
    .cin        (din[5]),
    .sum        (sum1[0]),
```

```
    .cout       (sum1[1])
);

adder2 u3(
    .dina       (sum0),
    .dinb       (sum1),
    .cin        (din[6]),
    .sum        (sum2[1:0]),
    .cout       (sum2[2])
    );

adder3 u2(
    .dina       (sum2),
    .dinb       (0),
    .cin        (din[7]),
    .sum        (num_one[2:0]),
    .cout       (num_one[3])
    );

endmodule

module full_adder_one(
    input   dina,
    input   dinb,
    input   cin,
    output  sum,
    output  cout
);

assign {cout, sum} = dina + dinb + cin;

endmodule
    module adder2(
    input   [1:0]   dina,
    input   [1:0]   dinb,
    input           cin,
    output  [1:0]   sum,
    output          cout
);

wire co;

full_adder_one u0(
    .dina       (dina[0]),
    .dinb       (dinb[0]),
    .cin        (cin),
    .sum        (sum[0]),
    .cout       (co)
);

full_adder_one u1(
    .dina       (dina[1]),
    .dinb       (dinb[1]),
    .cin        (co),
    .sum        (sum[1]),
    .cout       (cout)
);

```

```
endmodule

module adder3(
    input   [2:0]   dina,
    input   [2:0]   dinb,
    input           cin,
    output  [2:0]   sum,
    output          cout
);

wire co;

full_adder_one u0(
    .dina       (dina[0]),
    .dinb       (dinb[0]),
    .cin        (cin),
    .sum        (sum[0]),
    .cout       (co)
);

adder2 u1(
    .dina       (dina[2:1]),
    .dinb       (dinb[2:1]),
    .cin        (co),
    .sum        (sum[2:1]),
    .cout       (cout)
    );

endmodule
```

时序逻辑电路基础

题目：简述建立时间和保持时间，作图说明



建立时间Tsu（setup）：触发器在时钟上升沿到来之前，其数据输入端的数据必须保持不变的最小时间。

保持时间Th（hold）：触发器在时钟上升沿到来之后，其数据输入端的数据必须保持不变的最小时间。

题目:说明D触发器与Latch的区别。

锁存器对电平信号敏感，在输入脉冲的电平作用下改变状态。

D触发器对时钟边沿敏感，检测到上升沿或下降沿触发瞬间改变状态。

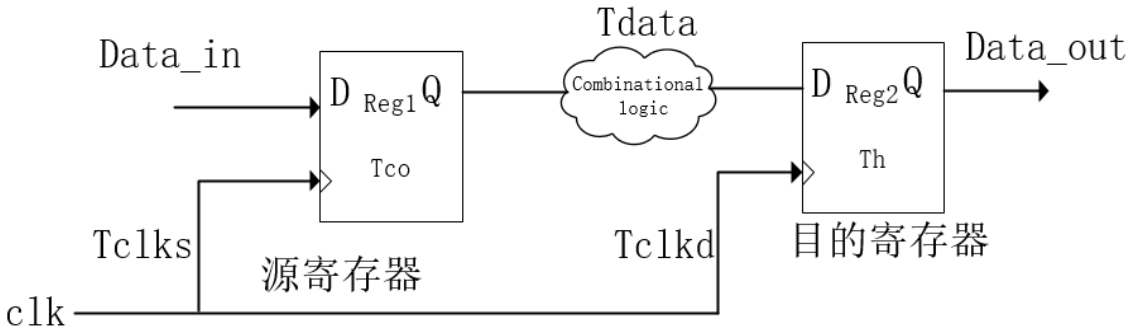
https://www.vlsifacts.com/difference-latch-flip-flop/

题目：什么是同步电路和异步电路。

同步逻辑是时钟之间有固定的因果关系。异步逻辑是各时钟之间没有固定的因果关系。

在电路中同一个时钟源的时钟分频出来的不同频率的时钟作用于两部分电路，这两部分电路也是同步的。反之，不同时钟源的电路就是异步电路。

题目：最小周期计算



Tco：寄存器更新延迟。clock output delay，时钟触发到数据输出的最大延迟时间

最小时钟周期：Tmin = Tco + Tdata + Tsu - Tskew。最快频率Fmax = 1/Tmin

Tskew = Tclkd – Tclks。

题目：什么是Clock Jitter和Clock Skew，这两者有什么区别。

时钟抖动（Clock Jitter）：指芯片的某一个给定点上时钟周期发生暂时性变化，使得时钟周期在不同的周期上可能加长或缩短。

时钟偏移（Clock Skew）：是由于布线长度及负载不同引起的，导致同一个时钟信号到达相邻两个时序单元的时间不一致。

区别：Jitter是在时钟发生器内部产生的，和晶振或者PLL内部电路有关，布线对其没有影响。Skew是由不同布线长度导致的不同路径的时钟上升沿到来的延时不同。

题目：什么是亚稳态，产生的原因，如何消除?

亚稳态：是指触发器无法在某个规定时间段内达到一个确定的状态。

原因：由于触发器的Tsu和Th不满足，当触发器进入亚稳态，使得无法预测该单元的输出，这种不稳定是会沿信号通道的各个触发器级联传播。

消除：两级或多级寄存器同步。理论上亚稳态不能完全消除，只能降低，一般采用两级触发器同步就可以大大降低亚稳态发生的概率，再加多级触发器改善不大。

```
reg data_d1;
reg data_d2;
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        data_d1 <= 1'd0;
        data_d2 <= 1'd0;
    end
    else begin
        data_d1 <= data_in;
        data_d2 <= data_d1;
    end
end
```

题目：同步和异步

同步复位和异步复位的区别

同步复位是复位信号随时钟边沿触发有效。异步复位是复位信号有效和时钟无关。

同步逻辑和异步逻辑的区别

同步逻辑是时钟之间有固定的因果关系。异步逻辑是各时钟之间没有固定的因果关系

同步电路和异步电路区别

同步电路有统一的时钟源，经过PLL分频后的时钟驱动的模块，因为是一个统一的时钟源驱动，所以还是同步电路。异步电路没有统一的时钟源。

跨时钟域处理

题目：reg和wire的区别

reg是寄存器类型可以存储数据，wire是线网型

reg型在always块和initial块中赋值，wire型用assign赋值

reg型可用于时序逻辑和组合逻辑赋值，wire型只能用于组合逻辑赋值

wire表示直通，即只要输入有变化，输出马上出现结果，reg表示一定要有触发，输出才会反映输入

题目：阻塞赋值与非阻塞赋值的区别

```
always @(posedge clk) begin
    b <= a;
    c <= b;
end

always @(posedge clk) begin
    b = a;
    c = b;
end
```

第一种赋值方式是非阻塞赋值，最后结果是b = a，c = b。

第二种赋值方式是阻塞赋值，最后的结果是b = a，c = a。

非阻塞赋值在触发调节满足是，两条语句是同时进行的，阻塞赋值是顺序执行的。

题目：localparam、parameter和define的区别

声明：

localparam xx = yy;

parameter xx = yy; `define XX YY

使用： xx `XX

localparam只能在当前Verilog文件中使用

parameter与define都可以用来定义常量

parameter写在模块中，可以被上一层模块调用时，进行参数传递。

parameter 作用于声明的那个文件；define 从编译器读到这条指令开始到编译结束都有效，或者遇到 `undef命令使之失效。

题目：task与function的区别

https://blog.csdn.net/kobesdu/article/details/39080571

题目：谈谈对Retiming技术的理解

Retiming就是重新调整时序，例如电路中遇到复杂的组合逻辑，延迟过大，电路时序不满足，这个时候采用流水线技术，在组合逻辑中插入寄存器加流水线，进行操作，面积换速度思想。

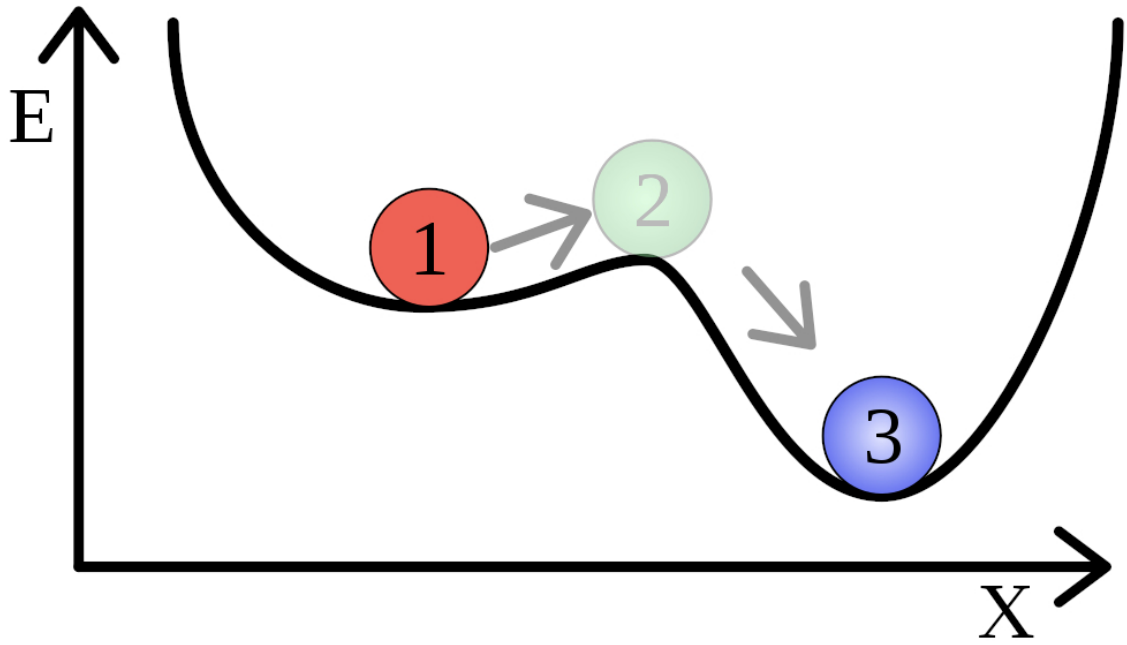
题目：什么是高阻态

高阻态：电路的一种输出状态，既不是高电平也不是低电平，如果高阻态再输入下一级电路的话，对下级电路无任何影响，可以理解为断路，不被任何东西所驱动，也不驱动任何东西

题目：解释一下亚稳态。

亚稳态指触发器的输出无法再某个规定时间段内达到一个可以确定的状态，介于0和1之间，如图中的2号小球既可能回到1状态，也可能达到3状态，亚稳态也是可以传输的，导致逻辑误判系统不稳定。亚稳态有恢复时间。解决亚稳态的方法

- 降低系统时钟
- 用更快的FF
- 引入同步机制，防止亚稳态传播
- 改善时钟质量



RTL代码

题目：多时钟域设计中，如何处理跨时钟域

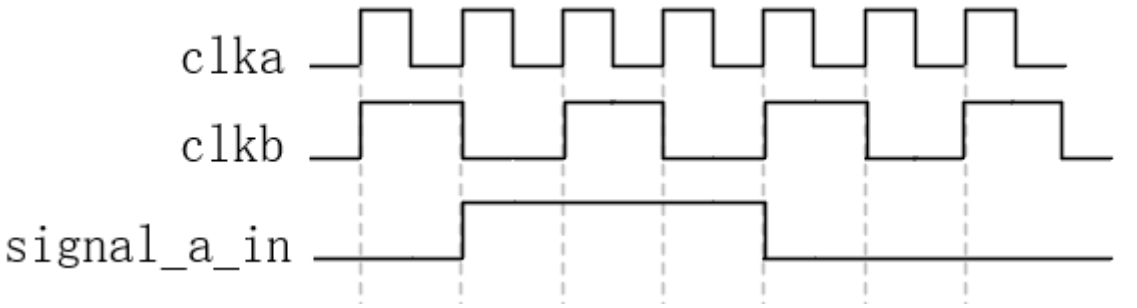
- 单bit：两级触发器同步（适用于慢到快）
- 多bit：采用异步FIFO，异步双口RAM
- 加握手信号
- 格雷码转换

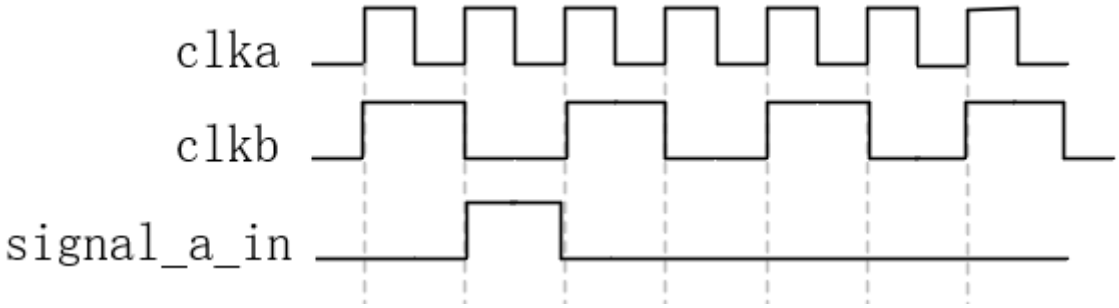
题目：编写Verilog代码描述跨时钟域信号传输，慢时钟域到快时钟域

```
reg     [1:0]   signal_r;
//----------------------------------------------------------
//
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_r <= 2'b00;
    end

else begin
       signal_r <= {signal_r[0], signal_in};
    end

end

assign  signal_out = signal_r[1];
```

题目：编写Verilog代码描述跨时钟域信号传输，快时钟域到慢时钟域

跨时钟域处理从快时钟域到慢时钟域，如果是下面第一个图，cklb则可以采样到signal_a_in，但是如果只有单脉冲，如第二个图，则不能确保采样掉signal_a_in。这个时候用两级触发器同步是没有用的。





代码如下：

```
//Synchronous
module Sync_Pulse(
    input           clka,
    input           clkb,
    input           rst_n,
    input           pulse_ina,
    output          pulse_outb,
    output          signal_outb
);

//------------------------------------------------------
reg             signal_a;
reg             signal_b;
reg     [1:0]   signal_b_r;
reg     [1:0]   signal_a_r;

//------------------------------------------------------
//在clka下，生成展宽信号signal_a
always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_a <= 1'b0;
    end
    else if(pulse_ina == 1'b1)begin
        signal_a <= 1'b1;
    end
    else if(signal_a_r[1] == 1'b1)
        signal_a <= 1'b0;
    else
       signal_a <= signal_a;
end

//------------------------------------------------------
//在clkb下同步signal_a
always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_b <= 1'b0;
    end
```

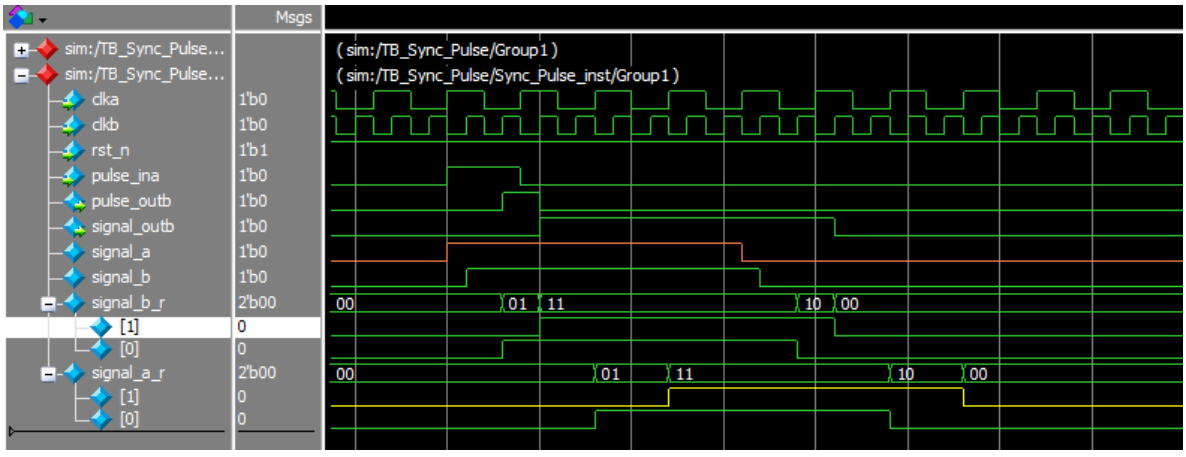
```
    else begin
        signal_b <= signal_a;
    end
end

//----------------------------------------------------------
//在clkb下生成脉冲信号和输出信号
always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
       signal_b_r <= 2'b00;
   end
   else begin
       signal_b_r <= {signal_b_r[0], signal_b};
    end
end

assign    pulse_outb = ~signal_b_r[1] & signal_b_r[0];
assign    signal_outb = signal_b_r[1];

//----------------------------------------------------------
//在clka下采集signal_b[1]，生成signal_a_r[1]用于反馈拉低signal_a
always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
       signal_a_r <= 2'b00;
    end
   else begin
       signal_a_r <= {signal_a_r[0], signal_b_r[1]};
    end

end

endmodule
```

这部分代码参考：

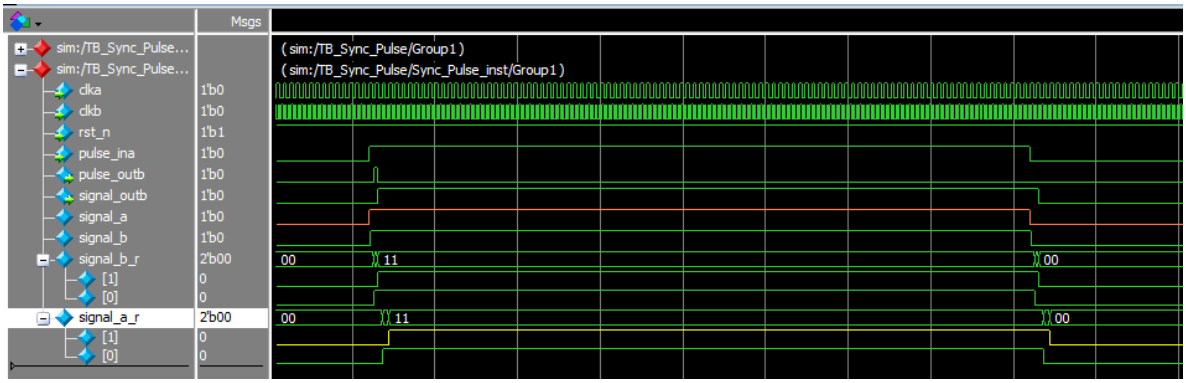
作者：肉娃娃

出处：https://home.cnblogs.com/u/rouwawa/

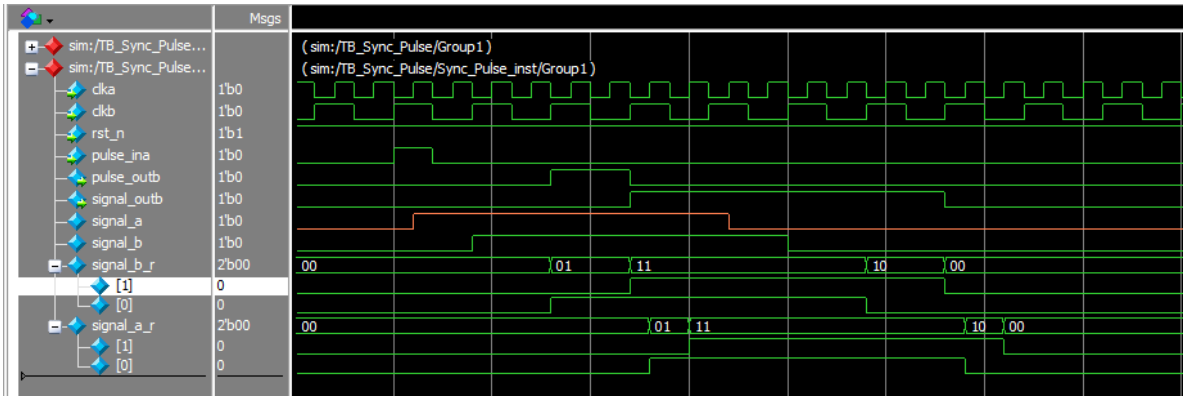
慢到快，单脉冲



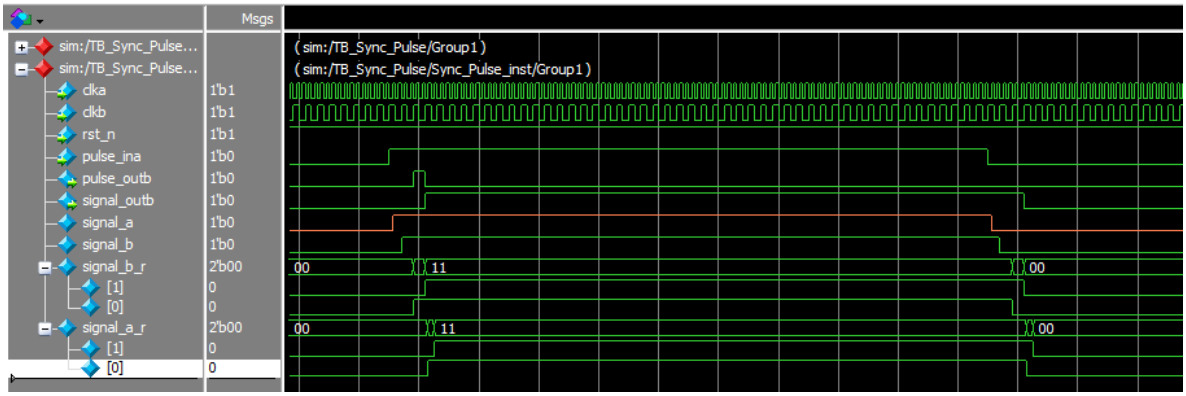
慢到快，长信号传递



快到慢，单脉冲



单脉冲，长信号传递



上述代码可以实现快到慢，慢到快时钟域任意转换，pulse_outb会输出单个脉冲，signal_outb输出信号时间长度最少为clkb的四个周期，当signal_a_in的信号长度大于clkb的四个周期，signal_outb输出与signal_a_in时间长度相同。

题目：用Verilog实现1bit信号边沿检测功能，输出一个周期宽度的脉冲信号。

- 上升沿
- 下降沿
- 上升沿或下降沿

```
input clk, rst_n, data;
output data_edge;
```

```
module Edge_Detect(
    input       clk,
    input       rst_n,
    input       data,
```

```
    output      pos_edge,
    output      neg_edge,
    output      data_edge
    );

reg     [1:0]   data_r;
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_r <= 2'b00;
    end
    else begin
        data_r <= {data_r[0], data};
    end
end

assign  pos_edge = ~data_r[1] & data_r[0];
assign  neg_edge = data_r[1] & ~data_r[0];
assign  data_edge = pos_edge | neg_edge;

endmodule
```

怎么记忆：上升沿之前是0，现在变成1，所以上个周期传输到的signal_r[1]是0所以取反。反之亦然。

题目： 用Verilog实现glitch free时钟切换电路。输入sel，clka，clkb，sel为1输出clka，sel为0输出clkb。

part1是比较垃圾的写法

part2 是两个时钟源是倍数的关系

part3是两个时钟源为异步时钟的关系

```
module Change_Clk_Source(
    input           clk1,
    input           clk0,
    input           select,
    input           rst_n,
    output          outclk
);

//------------------------------------------------------
//part 1
//assign outclk = (clk1 & select) | (~select & clk0);

//------------------------------------------------------
//part 2
reg     out1;
reg     out0;
always @(negedge clk1 or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        out1 <= 0;
    end
    else begin
        out1 <= ~out0 & select;
    end
end

```

```
26
27  always @(negedge clk0 or negedge rst_n)begin
28      if(rst_n == 1'b0)begin
29          out0 <= 0;
30      end
31      else begin
32          out0 <= ~select & ~out1;
33      end
34  end
35
36  assign outclk = (out1 & clk1) | (out0 & clk0);
37  /*
38  //------------------------------------------------------
39  //part 3
40  reg     out_r1;
41  reg     out1;
42  reg     out_r0;
43  reg     out0;
44
45  always @(posedge clk1 or negedge rst_n)begin
46      if(rst_n == 1'b0)begin
47          out_r1 <= 0;
48      end
49      else begin
50          out_r1 <= ~out0 & select;
51      end
52  end
53
54  always @(negedge clk1 or negedge rst_n)begin
55      if(rst_n == 1'b0)begin
56          out1 <= 0;
57      end
58      else begin
59          out1 <= out_r1;
60      end
61  end
62
63  always @(posedge clk0 or negedge rst_n)begin
64      if(rst_n == 1'b0)begin
65          out_r0 <= 0;
66      end
67      else begin
68          out_r0 <= ~select & ~out1;
69      end
70  end
71
72  always @(negedge clk0 or negedge rst_n)begin
73      if(rst_n == 1'b0)begin
74          out0 <= 0;
75      end
76      else begin
77          out0 <= out_r0;
78      end
79  end
80
81  assign outclk = (out1 & clk1) | (out0 & clk0);
82  */
83  endmodule
```

题目：用Verilog实现串并转换

- lsb优先
- msb优先

```
input clk, rst_n, data_i;
output [7:0] data_o;
```

```
module Deserialize(
    input           clk,
    input           rst_n,
    input           data_i,
    output   reg [7:0] data_o
    );

//lsb first
/*
reg     [2:0]   cnt;
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_o <= 8'b0;
        cnt <= 3'd0;
    end
    else begin
        data_o[cnt] <= data_i;
        cnt <= cnt + 1'b1;
    end
end
*/

//msb first
reg     [2:0]   cnt;
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_o <= 8'b0;
        cnt <= 3'd0;
    end
    else begin
        data_o[7 - cnt] <= data_i;
        cnt <= cnt + 1'b1;
    end
end

endmodule
```

题目：用verilog实现串并变换。

```
input [3:0] data_in;
output [3:0] data_out;
input [1:0] mode;
input clk;
input rst_n;
```

```
mode 0 ：串行输入data_in[0]，并行输出data_out[3:0]
mode 1 ：并行输入data_in[3:0]，串行输出data_out[0]
mode 2 ：并行输入data_in[3:0]，并行输出data_out[3:0]，延迟1个时钟周期
mode 3 ：并行输入data_in[3:0]，并行反序输出data_out[3:0]，延迟1个时钟周期并且交换bit顺序
data_out[3]=data_in[0];
data_out[2]=data_in[1]
data_out[1]=data_in[2]
data_out[0]=data_in[3]
```

附加要求【选做】 将输入输出的位宽做成参数化

可实现任意位宽设置

```
module Deserialize
#(
  parameter   DATA_WIDTH = 4,
  parameter   CNT_WIDTH  = log2(DATA_WIDTH)
  //parameter   CNT_WIDTH  = clog2(DATA_WIDTH-1)
)
(
    input           clk,
    input           rst_n,
    input    [1:0]  mode,
    input    [DATA_WIDTH-1:0]      data_i,
    output   reg [DATA_WIDTH-1:0]  data_o
    );

//mode 0
reg   [DATA_WIDTH-1:0]   data_r0;
reg            data_r1;
reg   [DATA_WIDTH-1:0]   data_r2;
reg   [DATA_WIDTH-1:0]   data_r3;

reg     [CNT_WIDTH-1:0]   cnt;
reg   [1:0] mode_r;
//mode change once cnt restart count
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
      mode_r <= 0;
    else
      mode_r <= mode;
end
assign change = (mode_r ^ mode)? 1'b1: 1'b0;

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
      cnt <= 0;
    else if(change == 1'b1)
      cnt <= 0;
    else
      cnt <= cnt + 1'b1;
end

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
```

```
      data_r0 <= 4'b0;
    else
      data_r0[cnt] <= data_i[0];
end

//mode 1
always @(posedge clk or negedge rst_n)begin
  if(rst_n == 0)
    data_r1 <= 0;
  else
    data_r1 <= data_i[cnt];
end

//mode 2
always @(posedge clk or negedge rst_n)begin
  if(rst_n == 0)
    data_r2 <= 0;
  else
    data_r2 <= data_i;
end

integer i;
reg [DATA_WIDTH -1:0] data_r;
always @(posedge clk)begin
for(i = 0; i <= DATA_WIDTH-1; i = i+1)begin
    data_r[DATA_WIDTH-1-i] <= data_i[i];
end
end

//mode 3
always @(posedge clk or negedge rst_n)begin
  if(rst_n == 0)
    data_r3 <= 0;
  else
    data_r3 <= data_r;
end

//mux4
always @(*)begin
  case(mode)
  2'b00: data_o = data_r0;
  2'b01: data_o = {data_o[3:1], data_r1};
  2'b10: data_o = data_r2;
  2'b11: data_o = data_r3;
  endcase
end

//-------------------------------------------------------
//以下两个函数任用一个
//求2的对数函数
function integer log2;
  input integer value;
  begin
    value = value-1;
    for (log2=0; value>0; log2=log2+1)
      value = value>>1;
  end
endfunction
```

```
101
102 //求2的对数函数
103 function integer clogb2 (input integer bit_depth);
104 begin
105     for(clogb2=0; bit_depth>0; clogb2=clogb2+1)
106         bit_depth = bit_depth>>1;
107 end
108 endfunction
109
110 endmodule
```

题目：用verilog实现一个4bit二进制计数器。

- 异步复位
- 同步复位

```
1 input clk, rst_n;
2 output [3:0] o_cnt;
```

```
1  module count_four_bit(
2      input           clk,
3      input           rst_n,
4      output  reg [3:0]   o_cnt
5      );
6
7  always @(posedge clk or negedge rst_n)begin
8      if(rst_n == 1'b0)begin
9          o_cnt <= 0;
10     end
11     else begin
12         o_cnt <= o_cnt + 1'b1;
13     end
14 end
15 /*
16 always @(posedge clk)begin
17     if(rst_n == 1'b0)begin
18         o_cnt <= 0;
19     end
20     else begin
21         o_cnt <= o_cnt + 1'b1;
22     end
23 end*/
24
25 endmodule
```

题目：用verilog实现4bit约翰逊(Johnson)计数器。

```
1  module Johnson_Counter(
2      input                   clk,
3      input                   rst_n,
4      output      reg [3:0]   johnson_cnt
5  );
6
7  //-------------------------------------------------
8  //johnson_cnt
```

```
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        johnson_cnt <= 4'b0000;
    else
        johnson_cnt <= {~johnson_cnt[0], johnson_cnt[3:1]};
end

endmodule
```

题目： 用verilog实现4bit环形计数器：复位有效时输出0001，复位释放后依次输出0010，0100，1000，0001，0010...

```
module cnt(
    input           clk,
    input           rst_n,
    output    reg  [3:0]   cnt
);

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt <= 4'b0001;
    end
    else begin
        cnt <= {cnt[2:0],cnt[3]};
    end
end

endmodule
```

题目：用verilog实现PWM控制呼吸灯。呼吸周期2秒：1秒逐渐变亮，1秒逐渐变暗。系统时钟24MHz，pwm周期1ms，精度1us。

led_out高电平占空比多，led较亮，反之，led较暗，实现呼吸灯效果

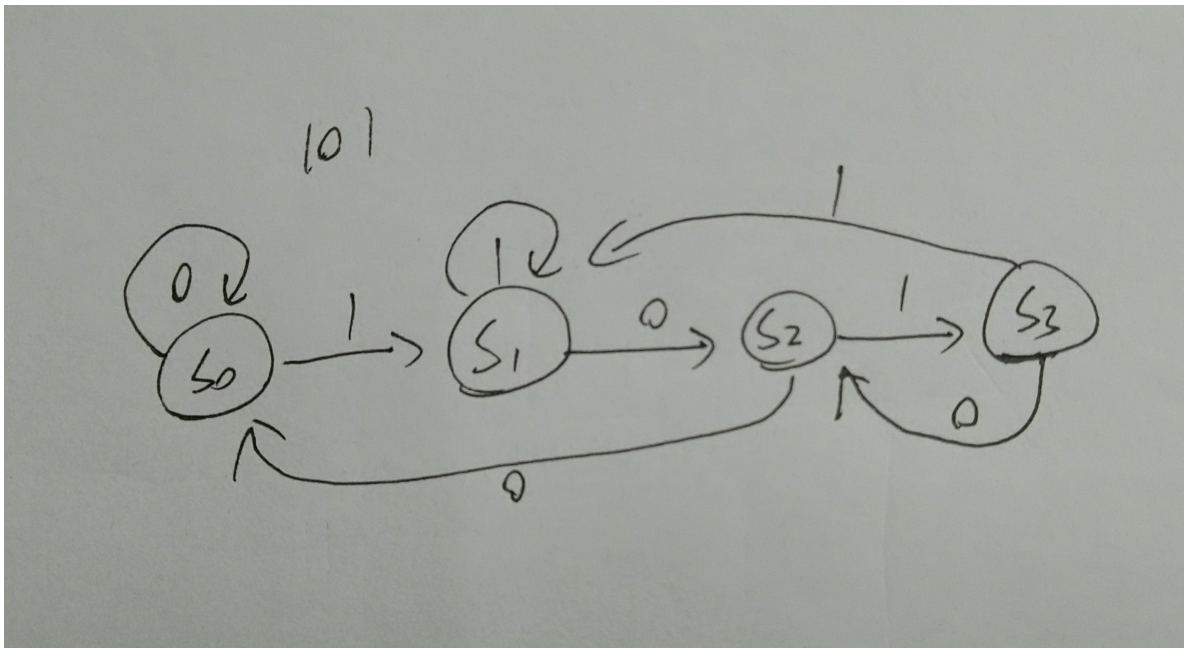
```
module Breath_LED(
    input           clk,    //24Mhz
    input           rst_n,
    output          led_out
);

parameter   DELAY24 = 24;
//parameter   DELAY1000 = 1000;
parameter   DELAY1000 = 10;//just test


wire            delay_1us;
wire            delay_1ms;
wire            delay_1s;
reg             pwm;
reg     [7:0]   cnt1;
reg     [10:0]  cnt2;
reg     [10:0]  cnt3;
reg             display_state;

//延时1us
```

```
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)
        cnt1 <= 6'b0;
    else if(cnt1 == DELAY24 - 1'b1)
        cnt1 <= 6'b0;
    else
        cnt1 <= cnt1 + 1'b1;
end

assign delay_1us = (cnt1 == DELAY24 - 1'b1)? 1'b1:1'b0;

//延时1ms
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)
        cnt2 <= 10'b0;
    else if(delay_1us == 1'b1)begin
        if(cnt2 == DELAY1000 - 1'b1)
            cnt2 <= 10'b0;
        else
            cnt2 <= cnt2 + 1'b1;
    end
    else
        cnt2 <= cnt2;
end
assign delay_1ms = ((delay_1us == 1'b1) && (cnt2 == DELAY1000 - 1'b1))?
1'b1:1'b0;

//延时1s
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)
        cnt3 <= 10'b0;
    else if(delay_1ms)
    begin
         if(cnt3 == DELAY1000 - 1'b1)
            cnt3 <= 10'b0;
        else
            cnt3 <= cnt3 + 1'b1;
     end
    else
        cnt3 <= cnt3;
end

assign delay_1s = ((delay_1ms == 1'b1) && (cnt3 == DELAY1000 - 1'b1))?
1'b1:1'b0;

//state change
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)
        display_state <= 1'b0;
    else if(delay_1s)//每一秒切换一次led灯显示状态
        display_state <= ~display_state;
    else
        display_state <= display_state;
end

//pwm信号的产生
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)
```

```
            pwm <= 1'b0;
    else
        case(display_state)
            1'b0: pwm <= (cnt2 < cnt3)? 1'b1:1'b0;
            1'b1: pwm <= (cnt2 < cnt3)? 1'b0:1'b1;
            default: pwm <= pwm;
        endcase
end

//位拼接使得输出八位led呼吸灯
assign led_out = pwm;

endmodule
```

题目：序列检测器：有“101”序列输入时输出为1，其他输入情况下，输出为0。画出状态转移图，并用Verilog描述。

```
input clk, rst_n, data;
output flag_101;
```




```
module Detect_101(
    input           clk,
    input           rst_n,
    input           data,
    output          flag_101
    );

parameter   S0 = 0,
            S1 = 1,
            S2 = 2,
            S3 = 3;

reg     [1:0]   state;

always @(posedge clk or negedge rst_n)begin
```

```
    if(rst_n == 1'b0)begin
        state <= S0;
    end
    else begin
        case(state)
        S0:
            if(data == 1)
                state <= S1;
            else
                state <= S0;
        S1:
            if(data == 0)
                state <= S2;
            else
                state <= S1;
        S2:
            if(data == 1)
                state <= S3;
            else
                state <= S0;
        S3:
            if(data == 1)
                state <= S1;
            else
                state <= S2;
        endcase
    end
end

assign  flag_101 = (state == S3)? 1'b1: 1'b0;

endmodule
```

题目：用Verilog实现一个异步双端口ram，深度16，位宽8bit。A口读出，B口写入。支持片选，读写请求，要求代码可综合。

```
module dpram_16x8 (
    input clk_a,
    input [3:0] addr_a,
    output [7:0] dout_a,
    ...
    input clk_b,
    input [7:0] din_b,
    input [3:0] addr_b,
    ...
);
...
endmodule
```

```
module Dual_Port_Sram
#(
    parameter           ADDR_WIDTH  =   4,
    parameter           DATA_WIDTH  =   8,
    parameter           DATA_DEPTH  =   1 << ADDR_WIDTH
)
```

```
(
    input               clka,
    input               clkb,
    input               rst_n,
    input               csen_n,
    //Port A Signal
    input       [ADDR_WIDTH-1:0]   addra,
    output   reg   [DATA_WIDTH-1:0]   data_a,
    input               rdena_n,
    //Port B Signal
    input       [ADDR_WIDTH-1:0]   addrb,
    input               wrenb_n,
    input       [DATA_WIDTH-1:0]    data_b
);

integer     i;
reg     [DATA_WIDTH-1:0]   register[DATA_DEPTH-1:0];

always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        for(i = 0; i < DATA_DEPTH; i = i + 1)
            register[i] <= 'b0000_1111;
    end
    else if(wrenb_n == 1'b0 && csen_n == 1'b0)
        register[addrb] <= data_b;
end

always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_a <= 0;
    end
    else if(rdena_n == 1'b0 && csen_n == 1'b0)
        data_a <= register[addra];
    else
        data_a <= data_a;
end

endmodule
```

题目：用Verilog实现三分频电路，要求输出50%占空比。

```
module Div_three(
    input       clk,
    input       rst_n,
    output      div_three
);

reg     [1:0]   cnt;
reg             div_clk1;
reg             div_clk2;
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt <= 0;
    end
    else if(cnt == 2)
        cnt <= 0;
    else begin
```

```
        cnt <= cnt + 1;
    end
end

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        div_clk1 <= 0;
    end
    else if(cnt == 0)begin
        div_clk1 <= ~div_clk1;
    end
    else
        div_clk1 <= div_clk1;
end

always @(negedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        div_clk2 <= 0;
    end
    else if(cnt == 2)begin
        div_clk2 <= ~div_clk2;
    end
    else
        div_clk2 <= div_clk2;
end

assign  div_three = div_clk2 ^ div_clk1;

endmodule
```

题目：用Verilog实现异步复位同步释放电路。

```
module Sys_Rst(
    input       clk,
    input       rst,
    output      sys_rst
);

reg     rst_r0;
reg     rst_r1;

always @(posedge clk or posedge rst)begin
    if(rst)begin
        rst_r0 <= 1'b1;
        rst_r1 <= 1'b1;
    end
    else begin
        rst_r0 <= 1'b0;
        rst_r1 <= rst_r0;
    end
end

assign  sys_rst = rst_r1;

endmodule
```

题目：用Verilog实现按键抖动消除电路，抖动小于15ms，输入时钟12MHz。

```
module debounce(
    input           clk,//12Mhz
    input           rst_n,
    input           key_in,
    output          key_flag
);

parameter   JITTER  =   240;//12Mhz / (1/20ms)

reg   [1:0]     key_r;
wire             change;
reg   [15:0]    delay_cnt;

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        key_r <= 0;
    end
    else begin
        key_r <= {key_r[0],key_in};
    end
end

assign  change = (~key_r[1] & key_r[0]) | (key_r[1] & ~key_r[0]);

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        delay_cnt <= 0;
    end
    else if(change == 1'b1)
        delay_cnt <= 0;
    else if(delay_cnt == JITTER)
        delay_cnt <= delay_cnt;
    else
        delay_cnt <= delay_cnt + 1;
end

assign  key_flag = ((delay_cnt == JITTER - 1) && (key_in == 1'b1))? 1'b1:
1'b0;

endmodule
```

题目：用Verilog实现一个同步FIFO，深度16，数据位宽8bit。

```
module Syn_fifo
#(
    parameter   DATA_WIDTH  =   8,
    parameter   ADDR_WIDTH  =   4,
    parameter   RAM_DEPTH   =   (1 << ADDR_WIDTH)
)
(
    input           clk,
    input           rst_n,
```

```
    input   [DATA_WIDTH-1:0]    data_in,
    input           wr_en,
    input           rd_en,
    output reg  [DATA_WIDTH-1:0]    data_out,
    output          empty,          //fifo empty
    output          full            //fifo full
);

reg     [ADDR_WIDTH-1:0]    wr_cnt;
reg     [ADDR_WIDTH-1:0]    rd_cnt;
reg     [ADDR_WIDTH-1:0]    status_cnt;
reg     [DATA_WIDTH-1:0]    data_ram;

//------------------------------------------------------
assign  full = (status_cnt == (RAM_DEPTH-1))? 1'b1: 1'b0;
assign  empty = (status_cnt == 0)? 1'b1: 1'b0;


//Syn
reg     rd_en_r;
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        rd_en_r <= 0;
    end
    else begin
        rd_en_r <= rd_en;
    end
end


//------------------------------------------------------
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        wr_cnt <= 0;
    end
    else if(wr_cnt == RAM_DEPTH-1)
        wr_cnt <= 0;
    else if(wr_en)begin
        wr_cnt <= wr_cnt + 1'b1;
    end
    else
        wr_cnt <= wr_cnt;
end

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        rd_cnt <= 0;
    end
    else if(rd_cnt == RAM_DEPTH-1)
        rd_cnt <= 0;
    else if(rd_en)begin
        rd_cnt <= rd_cnt + 1'b1;
    end
    else
        rd_cnt <= rd_cnt;
end

always @(posedge clk or negedge rst_n)begin
```

```
    if(rst_n == 1'b0)begin
        data_out <= 0;
    end
    else if(rd_en_r)begin
        data_out <= data_ram;
    end
end


always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        status_cnt <= 0;
    end
    else if(rd_en && !wr_en && (status_cnt != 0))begin
        status_cnt <= status_cnt - 1;
    end
    else if(wr_en && !rd_en && (status_cnt != RAM_DEPTH-1))
        status_cnt <= status_cnt + 1;
    else
        status_cnt <= status_cnt;
end

//-----------------------------------------------------
//Syn_Dual_Port_RAM
integer     i;
reg     [DATA_WIDTH-1:0]    register[RAM_DEPTH-1:0];

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        for(i = 0; i < RAM_DEPTH; i = i + 1)
            register[i] <= 0;
    end
    else if(wr_en == 1'b1)
        register[wr_cnt] <= data_in;
end

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_ram <= 0;
    end
    else if(rd_en == 1'b1)
        data_ram <= register[rd_cnt];
    else
        data_ram <= data_ram;
end

endmodule
```

Reference

http://www.asic-world.com/examples/verilog/syn_fifo.html

题目：IIC协议的RTL设计

https://zhuanlan.zhihu.com/p/34674402

https://www.cnblogs.com/ninghechuan/p/9534893.html

题目：Verilog设计异步FIFO

https://ninghechuan.com/2018/12/15/2018-12-15-Verilog%E8%AE%BE%E8%AE%A1%E5%BC%82%E6%AD%A5FIFO/

题目：FIFO深度计算

异步FIFO深度为17，如何设计地址格雷码

https://www.embedded.com/print/4015117

https://patents.google.com/patent/CN101930350A/zh

FIFO最小深度计算

你问我FIFO有多深?

Case-1：fA > fB 读写之间没有空闲周期

写速率fA = 80MHz

读速率fB = 50MHz

突发长度Burst Length = 120

读写之间没有空闲周期，是连续读写一个突发长度。

Sol：

写一个数据需要的时间 = 1 / 80MHz = 12.5ns

写一个突发需要的时间 = 120 * 12.5ns = 1500ns

读一个数据需要的时间 = 1 / 50MHz = 20ns

每1500ns，120个数据被写入FIFO，但读一个数据需要20ns的时间

可以计算出，1500ns内读出多少个数据，1500 / 20 = 75

剩下的没有读出，就存在FIFO中，则需要120 - 75 = 45

所以这种情况下，需要的FIFO最小深度为45

拿笔在纸上推导下更清楚。

Case-2：fA > fB 在两个连续读写之间有一个周期的延迟

Sol：

这个题目是制造了一些假象，这其实和Case-1的情况是一样的，因为两个连续的读写之间通常都会有延迟。解决方法，如同Case-1。

Case--3：fA > fB读写都有空闲周期（IDLE Cycles）

写速率fA = 80MHz

读速率fB = 50MHz

突发长度Burst Length = 120

两个连续写入之间的空闲周期为 = 1

两个连续读取之间的空闲周期为 = 3

Sol：

两个连续写入之间的空闲周期为1的意思是，每写入一个数据，要等待一个周期，再写入下一个数据。这也可以理解为每两个周期，写入一个数据。

两个连续读取之间的空闲周期为3的意思是，每读取一个数据，要等待三个周期，再读取下一个数据。这也可以理解为每四个周期，读取一个数据。

写一个数据需要的时间 = 2 * (1 / 80MHz) = 25ns

写一个突发需要的时间 = 120 * 25ns = 3000ns

读一个数据需要的时间 = 4 * (1 / 50MHz) = 80ns

每3000ns，120个数据被写入FIFO，但读一个数据需要80ns的时间

可以计算出，3000ns内读出可以多少个数据，3000 / 80 = 37.5

剩下的没有读出，就存在FIFO中，则需要120 - 37.5 = 82.5 约等于 83

所以这种情况下，需要的FIFO最小深度为83

Case-4：fA > fB并给出了读写使能的百分比

写速率fA = 80MHz

读速率fB = 50MHz

突发长度Burst Length = 120

写使能占得百分比为 = 50% = 1 / 2

读使能占得百分比为 = 25% = 1 / 4

Sol：

用你聪明的大脑想一想，这是不是和Case-3也是一模一样呢，写使能占得百分比为50%，即每两个周期写入一个数据。读使能占得百分比为25%，即每四个周期读取一个数据。

Case-5：fA < fB 读写操作无空闲周期（每两个连续读写之间有一个周期延迟）

Sol：

这类题目，因为读取速率大于写入速率，FIFO永远不会被写满，所以FIFO深度为1就够了。

Case-6：fA < fB 读写操作有空闲周期（读写使能占得百分比问题）

写速率fA = 30MHz

读速率fB = 50MHz

突发长度Burst Length = 120

两个连续写入之间的空闲周期为 = 1

两个连续读取之间的空闲周期为 = 3

Sol:

两个连续写入之间的空闲周期为1的意思是，每写入一个数据，要等待一个周期，再写入下一个数据。这也可以理解为每两个周期，写入一个数据。

两个连续读取之间的空闲周期为3的意思是，每读取一个数据，要等待三个周期，再读取下一个数据。这也可以理解为每四个周期，读取一个数据。

写一个数据需要的时间 = 2 * (1 / 30MHz) = 66.667ns

写一个突发需要的时间 = 120 * 66.667ns = 8000ns

读一个数据需要的时间 = 4 * (1 / 50MHz) = 80ns

每8000ns，120个数据被写入FIFO，但读一个数据需要80ns的时间

可以计算出，8000ns内读出可以多少个数据，8000 / 80 = 100

剩下的没有读出，就存在FIFO中，则需要120 - 100 = 20

所以这种情况下，需要的FIFO最小深度为20

Case-7：fA = fB 读写操作无空闲周期（每两个连续读写之间有一个周期延迟）

Sol：

很好理解。

如果读写时钟为同一个时钟，则可以不需要FIFO。

如果读写时钟存在相位差，FIFO深度为1，也是够了。

Case-8：fA = fB 读写操作有空闲周期（读写使能占得百分比问题）

写速率fA = 50MHz

读速率fB = 50MHz

突发长度Burst Length = 120

两个连续写入之间的空闲周期为 = 1

两个连续读取之间的空闲周期为 = 3

Sol：

同样的解题思路。

两个连续写入之间的空闲周期为1的意思是，每写入一个数据，要等待一个周期，再写入下一个数据。这也可以理解为每两个周期，写入一个数据。

两个连续读取之间的空闲周期为3的意思是，每读取一个数据，要等待三个周期，再读取下一个数据。这也可以理解为每四个周期，读取一个数据。

写一个数据需要的时间 = 2 * (1 / 50MHz) = 40ns

写一个突发需要的时间 = 120 * 40ns = 4800ns

读一个数据需要的时间 = 4 * (1 / 50MHz) = 80ns

每4800ns，120个数据被写入FIFO，但读一个数据需要80ns的时间

可以计算出，4800ns内读出可以多少个数据，4800 / 80 = 60

剩下的没有读出，就存在FIFO中，则需要120 - 60 = 60

所以这种情况下，需要的FIFO最小深度为60

Case-9 如果数据速率如下所示

读写速率相等

每100个时钟写入80个数据

每10个时钟读取8个数据

突发长度为160

Sol：

写速率的其他20个周期的位置是随机的。所以就有了下面几种情况。

Case	No. of cycles taken to complete write
1	200
2	200
3	180
4	160
5	200

为了保证数据的传输不丢失，我们考虑到最坏的情况。

考虑的最坏的情况，就是写数据速率和读数据速率之间的差别最大。即写数据速率最大，读数据速率最小。

写操作最坏得情况是Case-4 ，即两次连续的突发写入，又称“背靠背”的情况。

即为在160个周期内写入160个数据。

读数据速率读出每个数据的时间为 = 8 / 10

所以160个周期读出数据的个数为 160 * （8 * 10） = 128

剩下的没有读出，就存在FIFO中，则需要160 - 128 = 32

所以这种情况下，需要的FIFO最小深度为32。

Case-10：如下所示

写入时钟20MHz

读出时钟40MHz

每1000个时钟周期写入500个数据

每4个时钟周期读出1个数据

读写数据位宽一致。

Sol:

考虑到“背靠背”的情况突发长度则为500 * 2 = 1000

则为每1000个时钟周期写入1000个数据

每4个周期，读取一个数据。

写一个数据需要的时间 = 1 / 20MHz = 50ns

写一个突发需要的时间 = 1000 * 50ns = 50000ns

读一个数据需要的时间 = 4 * (1 / 40MHz) = 100ns

每50000ns，120个数据被写入FIFO，但读一个数据需要100ns的时间

可以计算出，50000ns内读出可以多少个数据，50000 /100 = 500

剩下的没有读出，就存在FIFO中，则需要1000- 500 = 500

所以这种情况下，需要的FIFO最小深度为500

Reference

CALCULATION OF FIFO DEPTH - MADE EASY —— Putta Satish

http://comm.chinaaet.com/adi/blogdetail/37555.html

https://blog.csdn.net/u011412586/article/details/10241585/

参考文献中诸多实例，可以参考点击查看。参考文献的文档关注**硅农**订阅号，后台回复**FIFO Depth Cal**即可获得。这份文档的写的太好了，解决所有FIFO深度计算类笔面试题都不再话下。

例子

FIFO深度 /（写入速率 - 读出速率）= FIFO被填满时间 > 数据包传送时间 = 写入数据量 / 写入速率

确保对FIFO写数据时不存在overflow,从FIFO读出数据时不存在underflow。

即：FIFO深度 = (写入速率 - 读出速率) * (写入数据量 / 写入速率)

例：A/D采样率50MHz，dsp读A/D读的速率40MHz，要不丢失地将10万个采样数据送入DSP，在A/D在和DSP之间至少加多大容量（深度）的FIFO才行?

100,000 / 50MHz = 1/500 s = 2ms (50MHz - 40MHz) * 1/500 = 20k就是FIFO深度。

例子

FIFO深度计算： 写入时钟20MHz，读出时钟40MHz，每1000个时钟周期写入500个数据，每4个时钟周期读出1个数据，读写数据位宽一致。

写时钟频率w_clk, 读时钟频率 r_clk, 写时钟周期里，每B个时钟周期会有A个数据写入FIFO 读时钟周期里，每Y个时钟周期会有X个数据读出FIFO 则，FIFO的最小深度是?

fifo_depth = burst_length - burst_length * X / Y * r_clk / w_clk

首先认为写操作是Burst（突发）进行的，但是写操作的效率并不是100%，而是A/B，因此实际的写速率为Fwr = (A/B)*w_clk。同理，实际的读速率为Frd = (X/Y) * r_clk。

而且这类题也没有约束突发长度场景，正常情况下应该是这样的：

空闲——Burst——空闲——Burst——空闲——Burst

但是在计算时要考虑极端的情况，即"背靠背"的情况。

空闲——Burst——Burst——空闲——Burst——Burst——空闲

这就是"背靠背"的极端情况。

所以这里的burst_length = (A+A) / w_clk

此题得解。

fifo_depth = 1000-1000(1/4)*(40/20) = 500

例子

再来一个例子。

例：一个8bit宽的AFIFO，输入时钟为100MHz，输出时钟为95MHz，设一个package为4Kbit，且两个package之间的发送间距足够大。求AFIFO的深度？

公式：fifo_depth = burst_length - burst_length * (X / Y) * (r_clk/w_clk)

burst_length = 4Kbit/8bit ，有两种结果

其一，根据存储厂商的惯用算法，4Kbit=4000bit，burst_length=500；

其二，用一般二进制算法，4Kbit=4*1024=4096bit，burst_length=512。

因为X和Y的值没有给出，所以默认为1.

其一，fifo_depth = 500 - 500*（95/100）= 25 ，所以fifo_depth最小取值是25 。

其二，fifo_depth = 512 - 512*（95/100）= 25.6 ，所以fifo_depth最小取值是26 。

SDRAM中应用

在SDRAM的应用中，我们通常使用的读写FIFO是突发长度的2倍，比如突发长度为256，那FIFO的深度设置为512，使得FIFO始终保持半满的状态。可以保证数据的传输。

题目：Verilog设计一个RAM

SRAM

Verilog设计一个sram，供仿真使用，实际工程中，FPGA直接例化工具自带的IP Core，ASIC由后端做专门的memory。 csen_n为低，wren_n为高写数据 csen_n为低，wren_n为高读数据 csen_n为高，memory不工作 。

```
reg     [DATA_WIDTH-1:0]   register[2**ADDR_WIDTH-1:0];

always @(posedge clk)begin
    if(!wren_n && !csen_n)
        register[addr] <= data_i;
end

always @(posedge clk)begin
    if(wren_n && !csen_n)
        data_o <= register[addr];
end
```

双口RAM

双口ram是单一时钟，支持一个读地址和一个写地址。本设计是同步读数ram，异步读数ram去掉时钟即可。

```
reg     [DATA_WIDTH-1:0]   register[2**ADDR_WIDTH-1:0];
reg     [ADDR_WIDTH-1:0]    addrb_r;

always @(posedge clk)begin
    if(!wrena_n && !csen_n)
        register[addra] <= data_a;
end

always @(posedge clk)begin
    if(!rdenb_n && !csen_n)
        data_b <= register[addrb];//read old data
        //addrb_r <= addrb//read new data
end

//assign data_b = register[addrb_r];//read new data
```

真双口RAM

真双口ram是两个时钟，支持两套独立完整的读写。

```
reg     [DATA_WIDTH-1:0]   register[2**ADDR_WIDTH-1:0];

always @(posedge clka)begin
        if(!wrena_n && !csen_n)begin
        register[addra] <= dina;
        douta <= dina;
end
    else if(!csen_n)
        douta <= register[addra];
end

always @(posedge clkb)begin
        if(!wrenb_n && !csen_n)begin
        register[addrb] <= dinb;
        doutb <= dinb;
end
    else if (!csen_n)begin
        doutb <= register[addrb];
    end
```

```
20 end
```

Register File

在我的认知中Register File的写法其实和上述大同小异，在FPGA中写成Register File，工具貌似会综合成BRAM资源，在ASIC设计中RAM Memory需要专门去做存储块。本文大致了解RAM的原理，只可供仿真使用。

使用Reg File在存储为小规模使用时有优势，面积小速度快，当存储的规模大于256个单元时，提倡使用sram的标准单元库。

SRAM和DRAM

SRAM：Static Random-Access Memory，静态随机存取存储器，只要供电数据就会保持，但断电数据就会消失，也被称为Volatile Memory

DRAM：Dynamic Random Access Memory，动态随机存储器，主要原理是利用电容存储电荷的多少来代表一个bit是0还是1，由于晶体管的漏电电流现象，电容会放电，所以要周期性的给电容充电，叫刷新。SRAM不需要刷新也会保持数据丢失，但是两者断电后数据都会消失，称为Volatile Memory

SDRAM和DDR

SDRAM：Synchronous Dynamic Random Access Memory，同步动态随机存储器，同步写入和读出数据的DRAM。

DDR：Double Data Synchronous Dynamic Random Access Memory，双倍速率同步动态随机存储器，就是DDR SDRAM双倍速率传输的SDRAM，在时钟的上升沿和下降沿都可以进行数据传输。我们电脑的内存条都是DDR芯片。

留一个练习

设计一个双口RAM，要求可读可写，可以同时读写，两种处理方式

- 将输入输出参数化，可重复调用
- 读取原来存储的值，然后立即写入新值
- 不能读取原来的值，而是直接获得正在写入的值

题目：Verilog设计一个ROM

FPGA中有专门的ROM IP Core，如果按照规范用Verilog编写的ROM文件可以被工具综合成RAM资源，而ASIC在需要后端去做专门Memory，前端仿真可以自己编写RAM/ROM/FIFO/RegFile IP。为了方便仿真这样写个ROM，方便初始化。

```
1  module single_port_rom(/*autoarg*/
2     // Outputs
3     q,
4     // Inputs
5     addr, clk
6     );
7
8  parameter DATA_WIDTH = 8;
9  parameter ADDR_WIDTH = 8;
10
11 input   [ADDR_WIDTH-1:0] addr;
12 input   clk;
13 output reg [DATA_WIDTH-1:0] q;
14
```

```
reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

initial begins

 $readmemh("/home/IC/Digital_Front_End_Verilog/ip_lib/rtl/DDS/triangular.txt", rom);
    //$readmemb("sin.txt", rom);
end

always @ (posedge clk)begin
    q <= rom[addr];
end

endmodule
```

系统函数$readmemh和$readmemb分别用来读取十六进制文件和二进制文件。貌似没有读十进制的。txt中的数据每行一个不需要逗号和最后一个数据后面的分号，数据格式对应。更多使用可以查询IEEE的Verilog语法手册。

例化方式和rom IP一样可参数化配置任意大小

```
single_port_rom
#(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH)
)
u_sin(/*autoinst*/
      // Outputs
      .q                                (dout[DATA_WIDTH-1:0]),
      // Inputs
      .addr                             (addra[ADDR_WIDTH-1:0]),
      .clk                              (clk));
```

题目：数的操作

数的表示

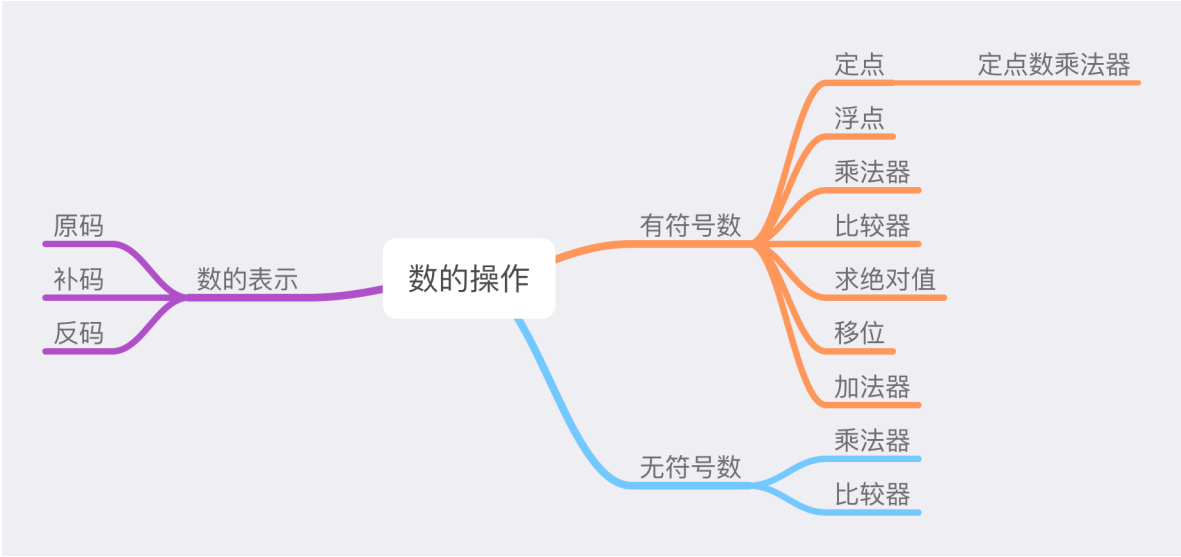
数有有符号数和无符号数

表示形式有原码、反码、补码。

无符号数处理相对来说比较简单。

有符号数运算中负数通常用补码表示。

浮点数和定点数相比不常用。定点数的运算电路和普通数据无差别，要注意小数点的位置。有符号数处理需要注意数据的符号位。



有符号数比较器

```
function    [DATA_WIDTH-1:0] max_op;
input   [DATA_WIDTH-1:0]   dat_op0;
input   [DATA_WIDTH-1:0]   dat_op1;

reg    cmp_flag;
if(dat_op0[DATA_WIDTH-1] != dat_op1[DATA_WIDTH-1])begin
    cmp_flag = dat_op0;
end
else begin
    cmp_flag = (dat_op0[DATA_WIDTH-1])? ((-dat_op0[DATA_WIDTH-1:0]) > (-dat_op1[DATA_WIDTH-1:0])): (dat_op0[DATA_WIDTH-1:0] < dat_op1[DATA_WIDTH-1:0]);
end

max_op[DATA_WIDTH-1:0] = cmp_flag? dat_op1[DATA_WIDTH-1:0]: dat_op0[DATA_WIDTH-1:0];

endfunction
```

还可以直接定义为signed，工具自动进行符号位扩展。

```
assign max_op[DATA_WIDTH-1:0] = ($signed(dat_op1) >= $signed(dat_op0)? dat_op0[DATA_WIDTH-1:0]: dat_op1[DATA_WIDTH-1:0];
```

有符号数求绝对值

```
function [DATA_WIDTH-1:0] dat_abs;
    input [DATA_WIDTH-1:0] dat_in;

    assign dat_abs[DATA_WIDTH-1:0] = (dat_in[DATA_WIDTH-1])? (-dat_in): dat_in;
endfunction
```

有符号数加法器

进行符号位扩展。

```
assign dout[DATA_WIDTH:0] = {add_a[DATA_WIDTH-1], add_a} +
{add_b[DATA_WIDTH-1], add_b};
```

有符号数乘法器

```
always @(*)begin
        if(tc)begin
            dat_a_tmp[A_WIDTH-1:0] = dat_a[A_WIDTH-1]? (-dat_a): dat_a;
            dat_b_tmp[B_WIDTH-1:0] = dat_b[B_WIDTH-1]? (-dat_b): dat_b;
            product_tmp[PRODUCT_WIDTH-1:0] = dat_a_tmp * dat_b_tmp;
            product[PRODUCT_WIDTH-1:0] = (dat_a[A_WIDTH-1] ^ dat_b[B_WIDTH-
1])? (-product_tmp): product_tmp;
        end
        else begin
            product[PRODUCT_WIDTH-1:0] = dat_a * dat_b;
        end
end
```

无符号数乘以常数，也直接用*号，例如a * 2'd3，工具会帮你优化成 a << 2'd1 + a。甚至可能优化得更好。（杠：不要过度依赖工具）。

上面对于无符号数，有符号数应该注意符号位扩展。

```
assign product[A_WIDTH+1:0] = {a[A_WIDTH-1], a[A_WIDTH-1:0],1'd0} +
{{2{a[A_WIDTH-1]}}, a[A_WIDTH-1:0]};
```

还可以直接定义为signed，工具自动进行符号位扩展。

```
input   signed  [A_WIDTH - 1:0]     multa;
input   signed  [B_WIDTH - 1:0]     multb;
output  signed  [C_WIDTH - 1:0]     product;

assign product = multa * multb;

input     [A_WIDTH - 1:0]     multa;
input     [B_WIDTH - 1:0]     multb;
output signed [C_WIDTH - 1:0]     product;

assign product = $signed(multa) * $sigend(multb);
```

哪种写法好？习惯了就行，最重要的是代码风格统一。不应该混用。

扫描关注我的微信订阅号，硅农，分享更多FPGA和ASIC设计相关知识。

硅农

微信扫描二维码，关注我的公众号

开源Verilog是一个免费知识星球，主要分享Verilog基础知识，定位于Verilog初学者。注意本星球不是主要讨论语法，而是讨论如何描述电路。

Verilog是用来描述数字电路的，不是设计电路的。再写代码前要心中有电路。当然这是要经过积累和经验才能慢慢体会到的。我们分为以下四个阶段练习。

心中无电路，代码无电路

心中有电路，代码无电路

心中有电路，代码有电路

心中无电路，代码无电路

第四个阶段可能就是精通阶段了。不过能达到第三个阶段你已经是高手级别了。

。。。博主还在努力想从第一个阶段爬出来。来一起练习吧。目前开源Verilog星球用户已经超过**700名硅农**。

硅农小灶是一个付费知识星球，主要是分享我个人学习过程中做的一些实验，主要面向Verilog和FPGA初学者。目前硅农小灶星球已经有超过**300****名硅农**加入，沉淀了大量的关于FPGA图像处理，ASIC/FPGA笔面试常考题目，持续更新Verilog小练习。还有本订阅号过去及未来涉及的一切资源。遇到问题先在星球搜索或题目，更精准且及时反馈。

硅农小灶持续分享，也欢迎有朋友进来分享和交流吧。

其中有错误或笔误的如果发现，请发送至邮箱

ninghechuan@foxmail.com，我会更新改正。