

笔试题

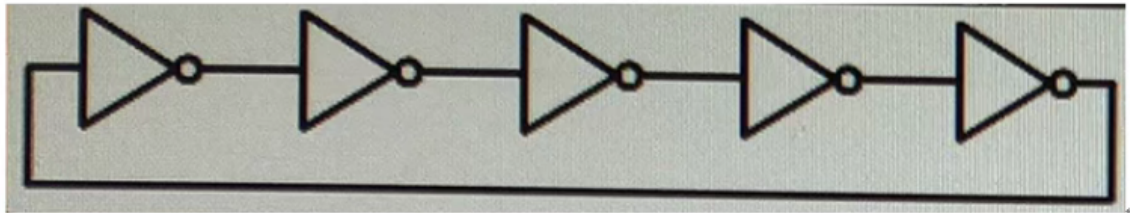
Q1: 有一个读端口和一个写端口的memory类型为?

- A. one port RAM
- B. ROM
- C. simple dual port RAM
- D. two port RAM

单口RAM读写一个地址；简单双口，一个口写，一个口读，时钟可以不同；真双口，两个端口都可以分别读写。选C

Q2: 环形振荡器

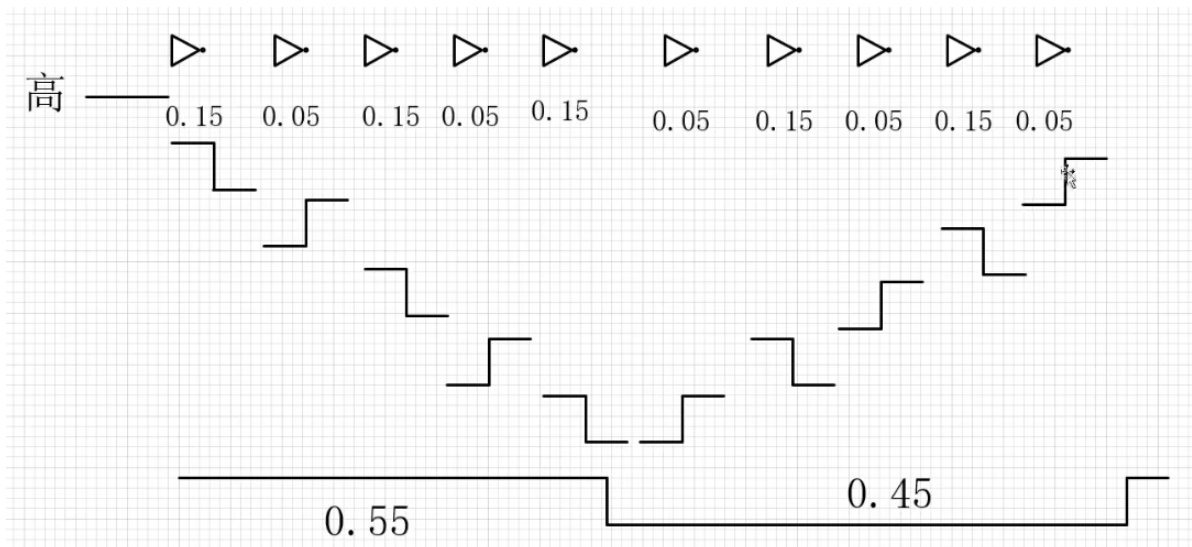
题 2 如下图环形振荡器 $TP_{LH}=0.05\text{ns}$ ， $TP_{HL}=0.15\text{ns}$ ，震荡周期 T 为多少？



- A 1ns
- B 0.25ns
- C 0.75ns
- D 0.5ns

假设以最左边的输入为起始点，两个反相器的延时为 $0.05+0.15\text{ ns}$ ，以两个为一组，数五组后正好回到起始点，那么总延时为 $5*0.2=1\text{ns}$ ，即周期。

方法2：假设高电平输入，经过五个反相器后翻转，高电平时间为 0.55ns ，变成低电平输入，经过 0.45ns 后翻转，总周期为 $0.55+0.45=1\text{ns}$ 。



Q3: 以下不是串行总线的?

A. SDIO B.SPI C.IIS D.AHB

SDIO用于SD卡接口传输总线, 4Bit data 串行的

SPI用于adc dac读出写入接口或者一些芯片控制寄存器接口, 或者SPI FLASH, 分为3线制, 4线制, 是串行的

IIS用于传输音频, 是串行

AHB AMBA总线一部分, 32bit位宽, 地址总线 and 数据总线, 并行的

Q4: 假设在CRC校验中生成多项式是 $G(x)=x^3+x+1$, 四位原始报文为1010, 求编码后的报文

生成多项式 $1011=x^3+x+1$

最高次幂表示有3位校验位

利用异或除生成多项式, 直到余数小于等于3位 得到校验位

原始报文1010添加校验位后是1010 000

1010 000

1011

0001 000

1 011

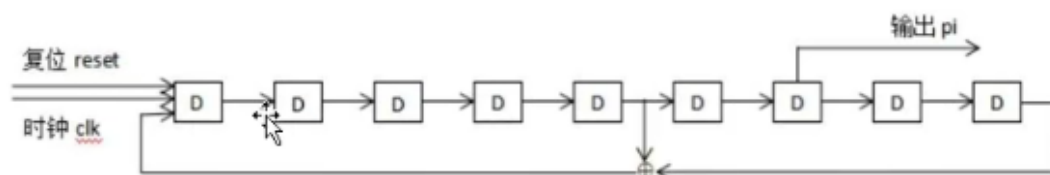
0000 011

校验位为011

所以编码后结果是1010 011

Q5: 编程

3. 初始值都是全 1



请用verilog或VHDL 写出完整代码。如以上两种语言不熟悉, 可用C语言实现。

```
`timescale 1ns/1ps
```

```

module shift_ex4(
    input wire sclk,
    input wire rst_n,

    output wire o_pi
);
reg [8:0] shift_reg;

always @(posedge sclk or negedge rst_n) begin
    if(!rst_n) begin
        shift_reg <= 9'b11111_1111;
    end
    else begin
        shift_reg <= {shift_reg[7:0], shift_reg[4]^shift_reg[8]}
    end
end

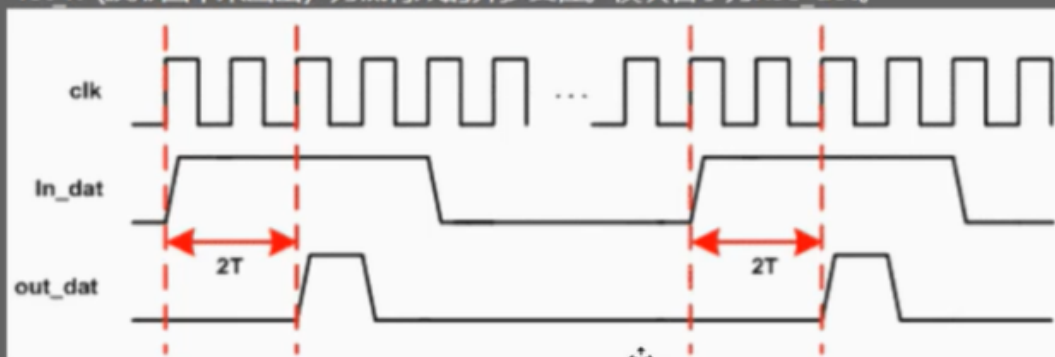
assign o_pi = shift_reg[6];
endmodule

```

Q6: 编程

问答题 | 10.0分

用HDL语言实现下面波形：时钟clk上升沿触发，reset信号
"rst_n"(波形图中未画出) 为低有效的异步复位。模块名字为rise_det。



```

6 // File : rise_det.v
7 // Create : 2021-07-22 22:15:21
8 // Revise : 2021-07-22 22:24:51
9 // Editor : sublime text3, tab size (4)
10 // -----
11 module rise_det(
12     input wire clk,
13     input wire rst_n,
14     input wire in_dat,
15     output wire out_dat
16 );
17 reg in_dat_dly1,in_dat_dly2;
18 reg out_dat_reg;
19
20 always @(posedge clk or negedge rst_n) begin
21     if(rst_n == 1'b0) begin
22         in_dat_dly1<=1'b0;
23         in_dat_dly2<=1'b0;
24     end
25     else begin
26         in_dat_dly1<=in_dat;
27         in_dat_dly2<=in_dat_dly1;
28     end
29 end
30
31 always @(posedge clk or negedge rst_n) begin
32     if(rst_n == 1'b0) begin
33         out_dat_reg <= 1'b0;
34     end
35     else if(in_dat_dly1== 1'b1 && in_dat_dly2 == 1'b0) begin
36         out_dat_reg <= 1'b1;
37     end
38     else begin
39         out_dat_reg <= 1'b0;
40     end
41 end
42
43 assign out_dat = out_dat_reg;
44

```

Q7: 问答题 脉冲产生

FPGA产生两个输出脉冲，要求这两个脉冲之间延迟为0.5ns，请描述实现方案。

IOB

Verilog中有符号与无符号的加法和乘法运算

对于无符号的乘法和加法，没有什么好说的，就是直接用'*'和'+'。

verilog里如果有符号数和无符号数做运算，会强制当作无符号运算

这就涉及verilog处理运算时的法则：

例如 $c = a + b$; 其中a和b都是四位数，c是五位。在计算时，**verilog会将a和b都扩展到5位，然后再做加法**，而如果a和b中有无符号数，则位宽扩展就按照无符号数来，也就是高位补0。所以如果a和b中既有无符号又有有符号，结果就不正确了。

解决方法是用 *signed* () 来修饰: $c = a + \text{signed}(b)$ 这样在 $c = a + b$, 这个运算开始的扩位就会按照有符号数的方式进行扩位, 在高位补符号位, 加法得出的结果就是a、b视为有符号数的结果。

1、无符号二进制数 (unsigned) 乘法运算

对于无符号数的乘法运算相对简单, 直接通过移位后相加得到。这里我随意举个无符号数相乘的例子来说明运算法则, 比如 $(1001)_2$ 和 $(0101)_2$ 相乘可以写成如下形式:

$$\begin{array}{r} 1001 9 \\ * 0101 5 \\ \hline 1001 \\ 0000 \\ 1001 \\ + 0000 \\ \hline 0101101 45 \end{array}$$

从上式可以看出, 该运算和我们十进制的运算相差无几, 在二进制运算中乘数 (0101) 从低位开始与被乘数 (1001) 相乘。在乘数为1时就相当于对被乘数进行相应的移位操作, 乘数使用的是第n位, 则被乘数相应的右移 $(n-1)$ 位, 最后对中间结果进行相加, 得到运算结果。

原理相当简单, 但是在verilog代码编写中, 在规定了乘数与被乘数的数据位宽后 (如例子的4bits数据), 结果数据位宽范围: 4bits~7bits $(4+4-1)$, 为防止运算结果溢出则需要预先给结果数据给定7bits的位宽。

根据最近的Verilog的学习, 我总结了如下表的数据位宽分配规律 (当然是针对无符号数而言)

数据名	数据位宽
乘数	a
被乘数	b
乘积	$a+b-1$

2有符号定点数乘法补位运算

对于有符号数的乘法运算, 对其乘数和被乘数进行位宽扩展。

这里我总结了一下, 扩展后的位数宽度为正常乘积位数 + 符号位, 即:

扩展后位宽 = 被乘数位数 a + 乘数位数 b - 1 + 符号位宽 (1 bit) = a + b
扩展后位宽 = 被乘数位数 a + 乘数位数 b - 1 + 符号位宽 (1 bit) = a + b

扩展时, 对高位进行补充的方式和无符号数的运算不一样, 需要依据符号进行补充, 符号位为'1'则高位补'1', 符号位为'0'则高位补'0'。同样的以 $(1001)_2$ 和 $(0101)_2$ 为例。乘数与被乘数均是4bits, 扩展后不至于溢出的位数为 $4+4=8$ bits

有符号数是以补码 (正数的补码为其本身, 负数的补码为其反码+1) 的形式来表示的, 则该数据的结果为:

补码: 1001 => 反码: 0110 => 原码: 0111 (-7) 补码: 1001 => 反码: 0110 => 原码: 0111 (-7)

补码: 1001 => 反码: 0110 => 原码: 0111 (-7)

补码：0101 => 原码：0101 (5) 补码：0101 => 原码：0101 (5)

补码：0101 => 原码：0101 (5)

乘法运算：-7 * 5 = -35 乘法运算：-7 * 5 = -35

乘法运算：-7 * 5 = -35

应用补位截断的方法进行有符号定点数乘法如下图所示

$$\begin{array}{r} * \begin{array}{l} 11111001 \\ 00000101 \end{array} \begin{array}{l} -7 \\ 5 \end{array} \\ \hline \begin{array}{l} \text{截断, 舍弃} \\ \end{array} \begin{array}{l} 11111001 \\ 00000000 \\ 11111001 \\ 00000000 \end{array} \\ \hline \begin{array}{l} + \\ \end{array} \begin{array}{l} 00000000 \\ 00000000 \end{array} \\ \hline \begin{array}{l} \\ \end{array} \begin{array}{l} 10011011101 \end{array} \end{array}$$

https://blog.csdn.net/qq_44626493

高于扩展位数的数据进行截断舍弃，最后结果如下

补码：11011101 => 反码：00100010 => 原码：00100011 补码：11011101 => 反码：

00100010 => 原码：00100011

补码：11011101 => 反码：00100010 => 原码：00100011

结果为：(-1) * (1 * 2^5 + 1 * 2^1 + 1 * 2^0) = -35 结果为：(-1) * (12^5 + 12^1 + 12^0) = -35

商汤

1. 芯片中降低功耗的方法有哪些，各有什么优缺点？

1、系统级的低功耗设计方法

(1) 多电压设计：压与功耗有着密切的联系。因此功耗的降低可以考虑使用低一点的电压。

(2) 软硬件协同：

1) 系统中的功耗都是硬件单元消耗的，但是软件组织对硬件的功耗有很大的影响。在满足系统应用的基础上，速度应尽可能慢、电压尽可能低、尽可能满足时间要求。应尽可能达到性能和功耗的平衡。

2) DVFS(动态电压频率技术)。将不同电路模块的工作电压以及工作频率降低到恰到好处满足系统最低要求，从而降低系统中不同电路模块的功耗。

3) 低功耗软件的动态功耗管理

(3) 系统时钟分配：

时钟是系统中频率最高的信号，其功耗之高是不容置疑的。在系统设计层面，可以考虑应用要求，将系统设置为不同的工作模式，加入时钟控制模块，在不同的工作模式下选用不同频率的时钟，并且将一些不需要的模块时钟关闭。

(4) 使用异步设计：

在异步设计里面，不需要全局时钟，两个模块通过握手信号进行交互，这时候就可以减少功耗。因此异步设计也是降低功耗的一种方法。

2、RTL级的低功耗设计方法：

(1) 并行设计：

对于某一个功能模块，我们可以通过并行的方式进行实现，也可以通过流水线的方式进行实现，这两种方法都是面积换速度，不过在一定的场合下可以降低功耗。

并行处理常用于数字信号处理部分。采用并行处理，可以降低系统工作频率，从而可能降低功耗；

(2)流水线的设计:

流水线技术可以将一个较长的组合路径分成M级流水线, 路径长度缩短为原始路径长度的 $1/M$ 。在相同的速度要求下, 可以采用较低的电源电压来驱动系统, 系统的整体功耗可能会降低。

(3)源共享:

对于设计比较多算术运算的设计, 如果有同样的操作在多处使用, 那我们就可以避免相应的运算逻辑在多个位置重复出现。

(4)使用低功耗的状态编码:

对于一些变化非常频繁的信号, 我们利用数据编码来降低开关活动(例如, 用格雷码比用二进制码翻转更少, 功耗更低)。

(5)操作数隔离:

如果在某一段时间内, 数据通路的输出是无用的, 则将其输入置成个固定值, 这样, 数据通路部分没有翻转, 功耗就会降低。

2.对于全局Bus, 如何提升频率?

3.异步信号进行同步的常用方法。

- [348] 如何处理跨时钟域信号?

跨时钟域处理有很多方法, 具体取决于我们需要在不同的时钟域之间传递1位还是多位。假设以下情况: 多个信号从一个时钟域传输到另一时钟域, 所有信号同时变化, 并且源和目标活动时钟沿彼此接近。在这种情况下, 这些信号中的某些信号可能在目标时钟域的一个时钟周期中被捕获, 而另一些信号在目标时钟域中的另一个时钟周期中被捕获, 从而导致数据不一致性。可以使用下面方法在两个时钟域之间同步信号。

对于单bit跨时钟域:

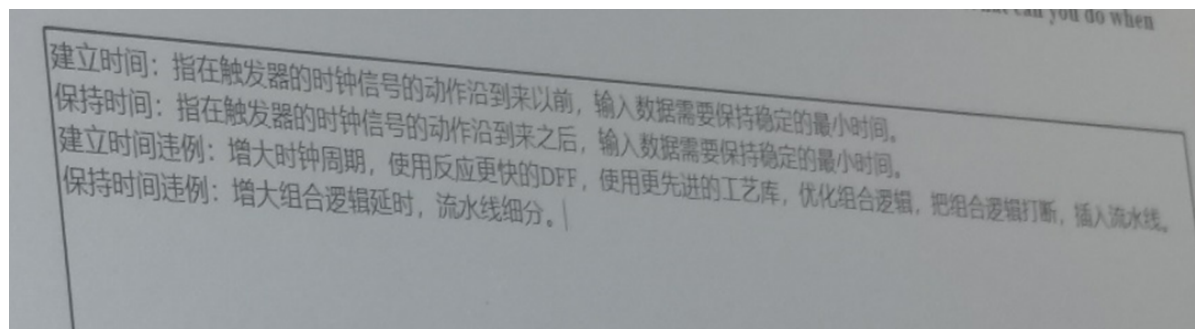
- 两级或者三级同步器
- 使用握手信号进行同步

对于多bit跨时钟域:

- 使用多周期路径的方法进行同步, 将未经同步的信号和同步控制信号一起发射到目标时钟域
- 对信号进行格雷码编码, 由于相邻的格雷码计数只会变化1bit, 亚稳态的发生会大大减小
- 使用异步FIFO
- 将多比特信号合并成1bit, 然后再通过多级同步器进行传输

澜起

1.建立时间与保持时间的概念, 出现违例后, 该如何修补?



2.latch与DFF的区别, 用Verilog实现latch的设计 (e为使能, i为输入, o为输出)。

- latch由电平触发, 非同步控制(没有时钟端, 不受系统同步时钟的控制, 无法实现同步操作)。在使能信号有效时latch相当于通路, 在使能信号无效时latch保持输出状态。DFF由时钟沿触发, 同步控制。
- latch容易产生毛刺 (glitch), DFF则不易产生毛刺
- 如果使用门电路来搭建latch和DFF, 则latch消耗的門资源比DFF要少, 这是latch比DFF优越的地方。所以, 在ASIC中使用latch的集成度比DFF高, 但在FPGA中正好相反, 因为FPGA中没有标准的latch单元, 但有DFF单元, 一个LATCH需要多个LE才能实现。
- latch将静态时序分析变得极为复杂, DFF则会时序分析变得容易。

方法1

```
module top_module (
    input d,
    input ena,
    output q);
    assign q=(ena)?d;q;
endmodule
```

方法2

```
module top_module (
    input d,
    input ena,
    output reg q);

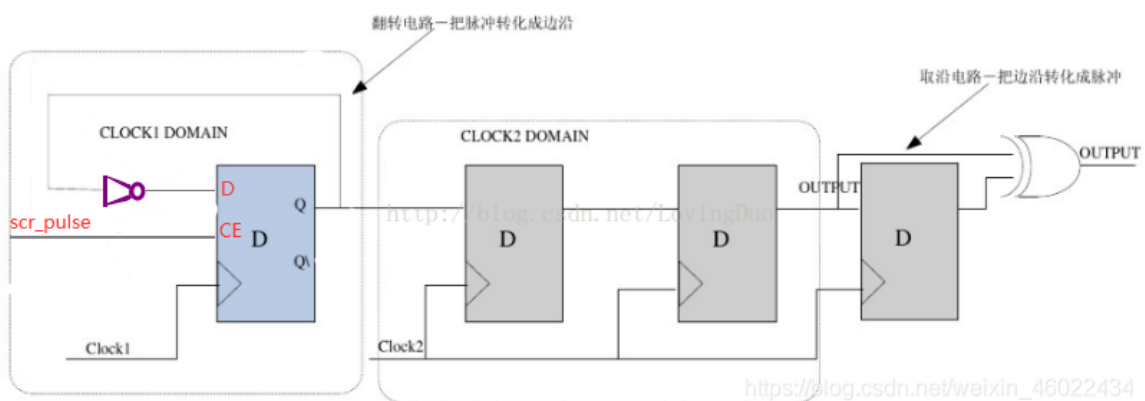
    always @(*)begin
        q <= (ena) ? d : q;
    end
endmodule
```

方法3

```
module top_module (
    input d,
    input ena,
    output reg q);

    always @(*)begin
        q = (ena) ? d : q;
    end
endmodule
```

3.clka同步到clkb时钟，clka比clkb时钟要快，用verilog实现。



```
module pulse_sync(src_clk, src_rst_n, src_pulse, dst_clk, dst_rst_n, dst_pulse);
    input      src_clk;           //source clock
    input      src_rst_n;         //source reset
    input      src_pulse;         //source pulse in

    input      dst_clk;           //destination clock
    input      dst_rst_n;         //destination reset
    output     dst_pulse;         //destination pulse out

    //Internal singles
    reg        src_state;
    reg        state_delay1;
```



```

    reg        state_delay2;
    reg        dst_state;
    wire       dst_puase;

//=====MODULE MAIN CODE=====
//1.输入脉冲转成电平信号，确保时钟B可以采到
    always@(posedge src_clk or negedge src_rst_n)begin
        if(src_rst_n==0)
            src_state <= 0;
        else if(src_pulse)
            src_state <= ~src_state;
    end

//2.//源时钟域的src时钟下电平信号转成时钟dst下的脉冲信号
    always@(posedge dst_clk or negedge dst_rst_n)begin
        if(dst_rst_n)begin
            state_delay1 <= 0;
            state_delay2 <= 0;
            dst_state <= 0;
        end
        else begin
            state_delay1 <= src_state;
            state_delay2 <= state_delay1;
            dst_state <= state_delay2;
        end
    end

    assign dst_state = dst_state^state_delay2;

endmodule

```

```

//另一版本
1 //
*****
*
2 // Project Name : fast2low
3 // Email       :
4 // Create Time  : 2020/07/15 :
5 // Module Name  :
6 // editor       :
7 // Version      : Rev1.0.0
8 //
*****
*
9
10 module fast2low(
11     input          clk_a, // fast clk
12     input          clk_b,
13     input          s_rst_n,
14
15     input          pulse_a,
16
17     output         pulse_outb,
18     output         signal_outb
19 );
20

```

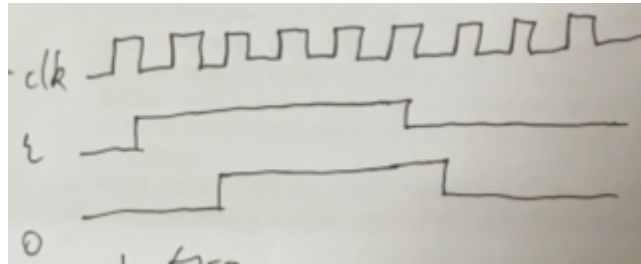
```

21 reg                                signal_a    ;
22 reg                                signal_b    ;
23
24 reg                                signal_b_r1  ;
25 reg                                signal_b_r2  ;
26
27 reg                                signal_b_a1  ;
28 reg                                signal_b_a2  ;
29
30 always @(posedge clka or negedge s_rst_n) begin // 在时钟 clka 下，生成展宽信号
signal_a
31     if(s_rst_n == 1'b0)
32         signal_a <= 1'b0;
33     else if(pulse_a == 1'b1)                // 检测到输入信号 pulse_a_in被拉高，则拉高
signal_a
34         signal_a <= 1'b1;
35     else if(signal_b_a2 == 1'b1)            // 检测到 signal_b_a2被拉高，则拉低signal_a
36         signal_a <= 1'b0;
37     else
38         signal_a <= signal_a;
39 end
40
41 always @ (posedge clkb or negedge s_rst_n) begin // 在clkb下同步signal_a
42     if(s_rst_n == 1'b0)
43         signal_b <= 0;
44     else
45         signal_b <= signal_a;
46 end
47
48 always @ (posedge clkb or negedge s_rst_n) begin
49     if(s_rst_n == 1'b0) begin
50         signal_b_r1 <= 1'b0;
51         signal_b_r2 <= 1'b0;
52     end
53     else begin
54         signal_b_r1 <= signal_b;
55         signal_b_r2 <= signal_b_r1;
56     end
57 end
58
59 always @ (posedge clka or negedge s_rst_n) begin // 在时钟域clk_a下，采集
signal_b_r1, 用于反馈来拉低展宽信号signal_a
60     if(s_rst_n == 1'b0) begin
61         signal_b_a1 <= 1'b0;
62         signal_b_a2 <= 1'b0;
63     end
64     else begin
65         signal_b_a1 <= signal_b_r1;
66         signal_b_a2 <= signal_b_a1;
67     end
68 end
69
70 assign signal_outb = signal_b_r1;
71 assign pulse_outb = signal_b_r1 & (~signal_b_r2);
72
73 endmodule

```

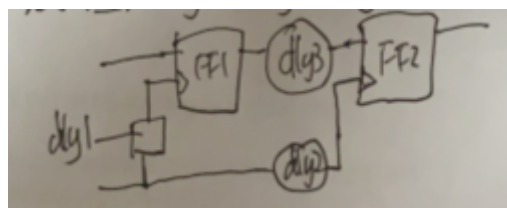
4.描述digital front设计流程与用到的对应的EDA工具。

5.用Verilog实现上下述波形图。



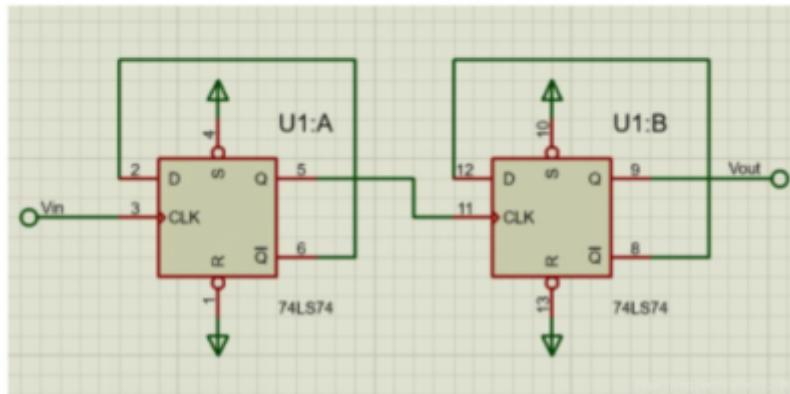
```
module rise_det(  
    input clk,  
    input rstn,  
    input signal_in,  
    output signal_out  
);  
  
reg reg_a, reg_b;  
  
always @(posedge clk or negedge rstn) begin  
    if(!rstn) begin  
        reg_a <= 1'b0;  
        reg_b <= 1'b0;  
    end  
    else begin  
        reg_a <= signal_in;  
        reg_b <= reg_a;  
    end  
end  
  
assign signal_out = reg_a & reg_b;  
  
endmodule
```

6.修复setup与hold, 图如下

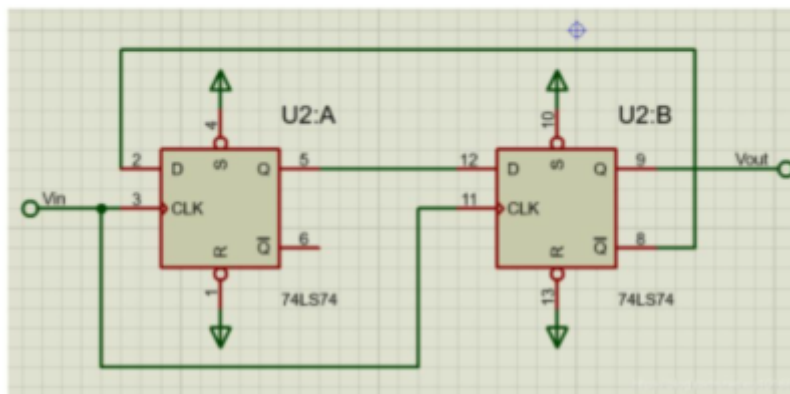


7.四分频电路, 分别用同步电路实现和异步电路实现,

方法一：先接成两个二分频电路，再相连就是四分频电路，依次类推可以做成八分频、十六分频.....



方法二：直接四分频电路



```
// 同步
module divide4_sync(
    input clk_in,
    input rstn,
    output clk_out
);

    reg reg_a, reg_b;
    wire reg_b_n;
    assign reg_b_n = ~reg_b;

    always @(posedge clk_in or negedge rstn) begin
        if(!rstn) begin
            reg_a <= 1'b0;
            reg_b <= 1'b0;
        end
        else begin
            reg_a <= reg_b_n;
            reg_b <= reg_a;
        end
    end

    assign clk_out = reg_b;
end
```

```

endmodule

//异步
module divide4_asyn(
    input clk_in,
    input rstn,
    output clk_out
);

    reg reg_a, reg_b;
    wire reg_a_n, reg_b_n;
    assign reg_a_n = ~reg_a;
    assign reg_b_n = ~reg_b;

    always @(posedge clk_in or negedge rstn) begin
        if(!rstn) begin
            reg_a <= 1'b0;
        end
        else begin
            reg_a <= reg_a_n;
        end
    end

    wire clk_b_in;
    assign clk_b_in = reg_a;

    always @(posedge clk_b_in or negedge rstn) begin
        if(!rstn) begin
            reg_b <= 1'b0;
        end
        else begin
            reg_b <= reg_b_n;
        end
    end

    assign clk_out = reg_b;

endmodule

```

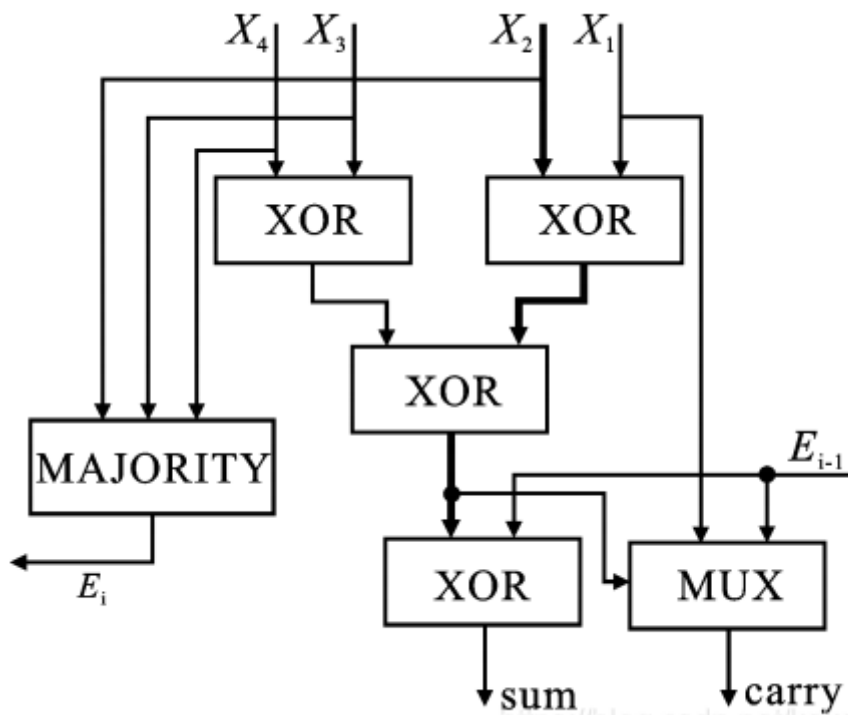
8.计算 $A+B+C+D+E$ ，其中A、B、C、D是32bit，E为1bit。

2) 两种实现

第一种实现方式是利用全加器FA来实现的，其结构图如下：

这里是用两个FA串联构成的，这种4-2压缩器有一个缺点就是它的进位信号产生要经过4个异或门的延迟（参看全加器的结构），当我们部署4-2压缩器的时候就会使电路的时序延长，造成不必要的性能损失，因此不推荐使用。

第二种实现方式的结构图如下：



改进型的4-2压缩器产生进位信号只需要3个异或门的时延，这缩短了结果产生的时间，我认为是一种好的设计。

$$E_i = (X_4 \& X_3) \mid \mid (X_3 \& X_2) \mid \mid (X_4 \& X_2)$$

$$\text{sum} = X_4 \oplus X_3 \oplus X_2 \oplus X_1 \oplus E_{i-1}$$

$$\text{carry} = (X_1 \& (\overline{X_4 \oplus X_3 \oplus X_2 \oplus X_1})) \mid \mid (E_{i-1} \& (X_4 \oplus X_3 \oplus X_2 \oplus X_1))$$

$$X_1 + X_2 + X_3 + X_4 + E_{i-1}$$

$$= 2 \times (E_i + \text{carry}) + \text{sum}$$

9. verilog 实现 3×3 矩阵A 乘 3×1 vector B. 3×1 vector C 为结果
面积小, 耗周期短.

10.异步FIFO使用格雷码的原因，Verilog实现格雷码到二进制码的转换

在异步的FIFO中，采用格雷码进行计数，相邻的数据仅仅只有1bit变化，这样在两个时钟域同步的时候仅仅可能只有1bit产生亚稳态，通过同步以后，亚稳态可以消除，最坏的情况是这1bit采错，但是即使是采错地址也只是相差1个，这对判断空满标志不会产生很大影响。

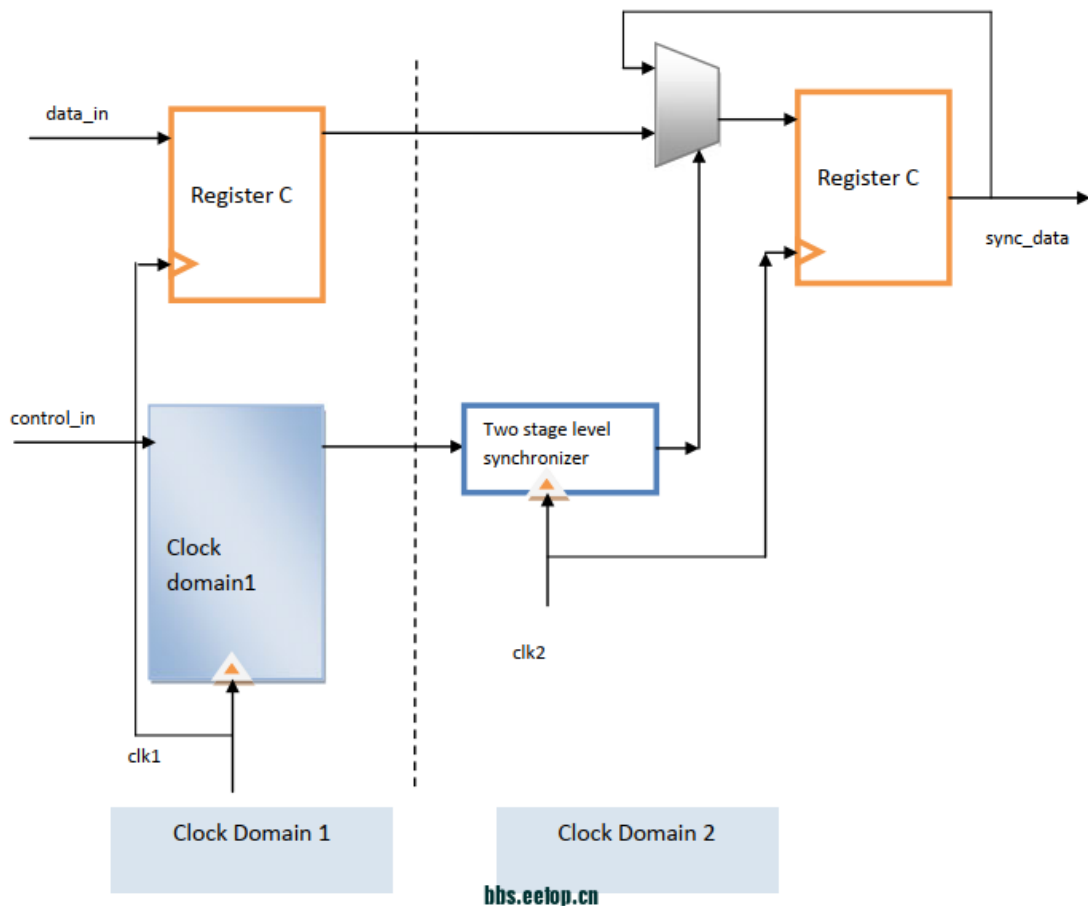
```
module gry2bin(Gry,Bin);
    parameter length = 8;
    input [length-1:0] Gry;
    output [length-1:0] Bin;
    reg [length-1:0] Bin;

    integer i;
    always @ (Gry)begin
        Bin[length-1]=Gry[length-1];           //最高位保留
        for(i=length-2;i>=0;i=i-1)
            Bin[i]=Bin[i+1]^Gry[i];
        end
    endmodule

module binary2grey #(parameter N = 4)(
    input [N-1: 0] binary ,
    output wire [N-1: 0] grey
);

    assign grey = (binary>>1)^binary;           //{1'b0,binary[N-1:1]}^binary;
endmodule
```

DMUX跨时钟域



Dmux实现多位宽信号同步原理：将多Bit数据信号和单bit控制信号成对的从发送时钟域发往接受时钟域，控制信号在接收时钟域以两级同步器（打两拍）的形式接收，多Bit数据信号由同步后的单Bit控制信号控制的MUX决定数据是否通过。

MUX的作用：判断源时钟域的单比特信号是否在目的时钟域成功同步，如果是，那么这个时间长度下多比特信号也可以同步过来，而不违背建立时间

异步复位同步释放

```
`timescale 1ns / 1ps

module test(
    input clk,
    input rst_n,
    input data_in,
    output reg data_out
);

    reg rst_n_reg_a, rst_n_reg_b;
    wire rst_n_sync;

    always @ (posedge clk or negedge rst_n)
        if (!rst_n)
            begin
                rst_n_reg_a <= 1'b0;
                rst_n_reg_b <= 1'b0;
            end
end
```

```

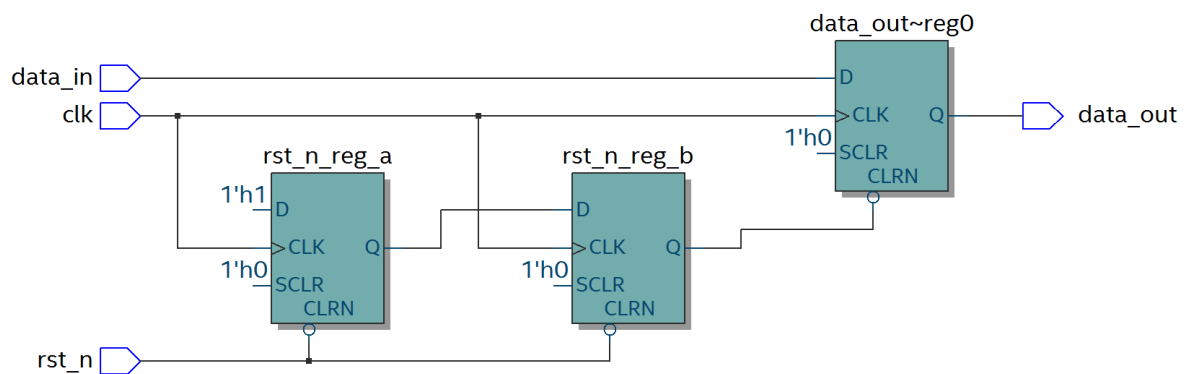
else
begin
    rst_n_reg_a <= 1'b1;
    rst_n_reg_b <= rst_n_reg_a;
end

assign rst_n_sync = rst_n_reg_b;

always @ (posedge clk or negedge rst_n_sync)
if (!rst_n_sync)
    data_out <= 1'b0;
else
    data_out <= data_in;

endmodule

```



https://blog.csdn.net/q_35579390

矩阵乘法

Row	Column	Bit's Position in 1-D array
0	0	7:0
0	1	15:8
0	2	23:16
1	0	31:24
1	1	39:32
1	2	47:40
2	0	55:48
2	1	63:56
2	2	71:64

```

//3 by 3 matrix multiplier. Each element of the matrix is 8 bit wide.
//Inputs are named A and B and output is named as C.
//Each matrix has 9 elements each of which is 8 bit wide. So the inputs is 9*8=72
bit long.

```

```

module matrix_mult
(
    input Clock,
    input reset, //active high reset
    input Enable, //This should be High throughout the matrix
multiplication process.
    input [71:0] A,
    input [71:0] B,
    output reg [71:0] C,
    output reg done //A High indicates that multiplication is done and
result is availble at C.
);

//temperory registers.
reg signed [7:0] matA [2:0][2:0];
reg signed [7:0] matB [2:0][2:0];
reg signed [7:0] matC [2:0][2:0];
integer i,j,k; //loop indices
reg first_cycle; //indicates its the first clock cycle after Enable went
High.
reg end_of_mult; //indicates multiplication has ended.
reg signed [15:0] temp; //a temeporary register to hold the product of two
elements.

//Matrix multiplication.
always @(posedge Clock or posedge reset)
begin
    if(reset == 1) begin //Active high reset
        i = 0;
        j = 0;
        k = 0;
        temp = 0;
        first_cycle = 1;
        end_of_mult = 0;
        done = 0;
        //Initialize all the matrix register elements to zero.
        for(i=0;i<=2;i=i+1) begin
            for(j=0;j<=2;j=j+1) begin
                matA[i][j] = 8'd0;
                matB[i][j] = 8'd0;
                matC[i][j] = 8'd0;
            end
        end
    end
    else begin //for the positive edge of Clock.
        if(Enable == 1) //Any action happens only when Enable is High.
            if(first_cycle == 1) begin //the very first cycle after Enable
is high.
                //the matrices which are in a 1-D array are converted to 2-D
matrices first.
                for(i=0;i<=2;i=i+1) begin
                    for(j=0;j<=2;j=j+1) begin
                        matA[i][j] = A[(i*3+j)*8+:8];
                        matB[i][j] = B[(i*3+j)*8+:8];
                        matC[i][j] = 8'd0;
                    end
                end
                //re-initialize registers before the start of multiplication.
                first_cycle = 0;
            end
        end
    end
end
end

```

```

        end_of_mult = 0;
        temp = 0;
        i = 0;
        j = 0;
        k = 0;
    end
    else if(end_of_mult == 0) begin        //multiplication hasnt ended.
        Keep multiplying.
        //Actual matrix multiplication starts from now on.
        temp = matA[i][k]*matB[k][j];
        matC[i][j] = matC[i][j] + temp[7:0];    //Lower half of the
        product is accumulatively added to form the result.
        if(k == 2) begin
            k = 0;
            if(j == 2) begin
                j = 0;
                if (i == 2) begin
                    i = 0;
                    end_of_mult = 1;
                end
            else
                i = i + 1;
            end
        end
        else
            j = j+1;
        end
    else
        k = k+1;
    end
    else if(end_of_mult == 1) begin        //End of multiplication has
        reached
        //convert 3 by 3 matrix into a 1-D matrix.
        for(i=0;i<=2;i=i+1) begin    //run through the rows
            for(j=0;j<=2;j=j+1) begin    //run through the columns
                C[(i*3+j)*8+:8] = matC[i][j];
            end
        end
        done = 1;    //Set this output High, to say that C has the final
        result.
    end
end
end
endmodule

```

格雷码快速写法

格雷码的bit n ($n = 0, 1, 2, \dots$), 以 $[2^n \text{个} 0, 2^n \text{个} 1, 2^n \text{个} 1, 2^n \text{个} 0]$ 为周期变化

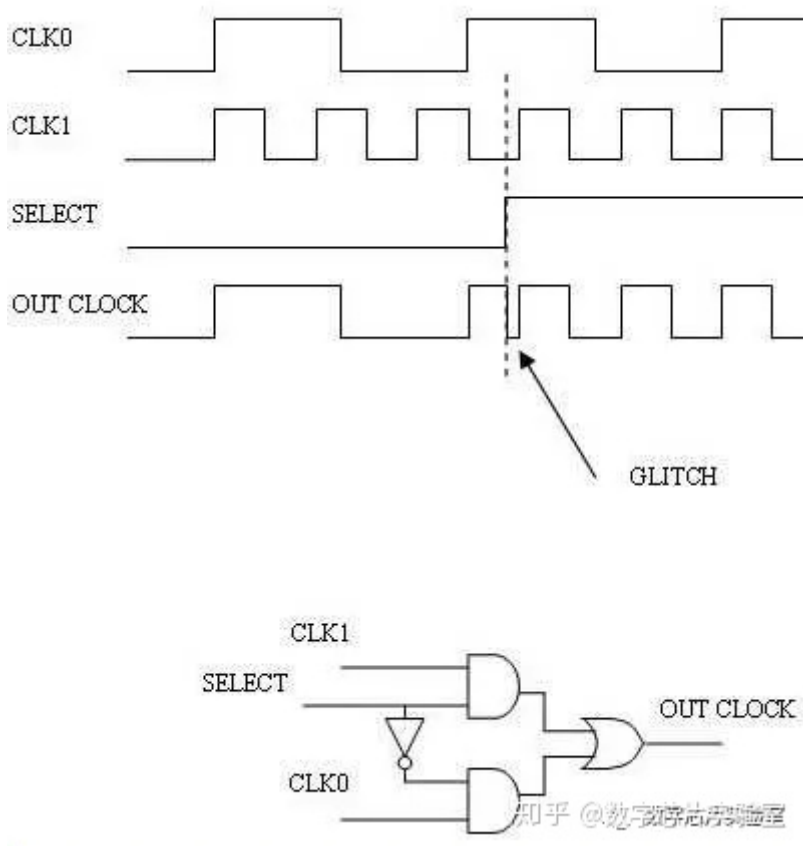
当FIFO深度是非2次幂的偶数时, 使用掐头去尾法, 但是要设计新的地址译码逻辑

Glitch Free 时钟切换电路

随着当今芯片中使用越来越多的时钟，通常在芯片运行时需要对时钟进行切换。

我们可以在芯片中复用两个不同频率的时钟源，由内部逻辑控制多路复用器的选择信号。

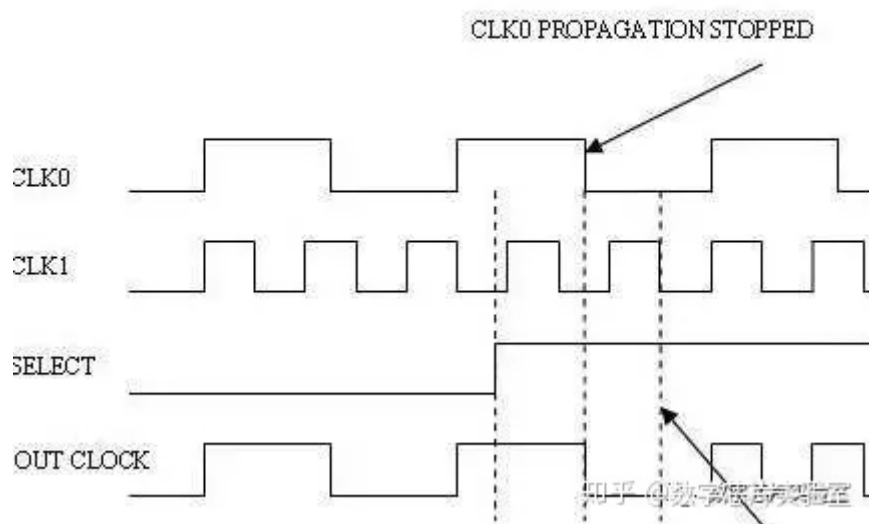
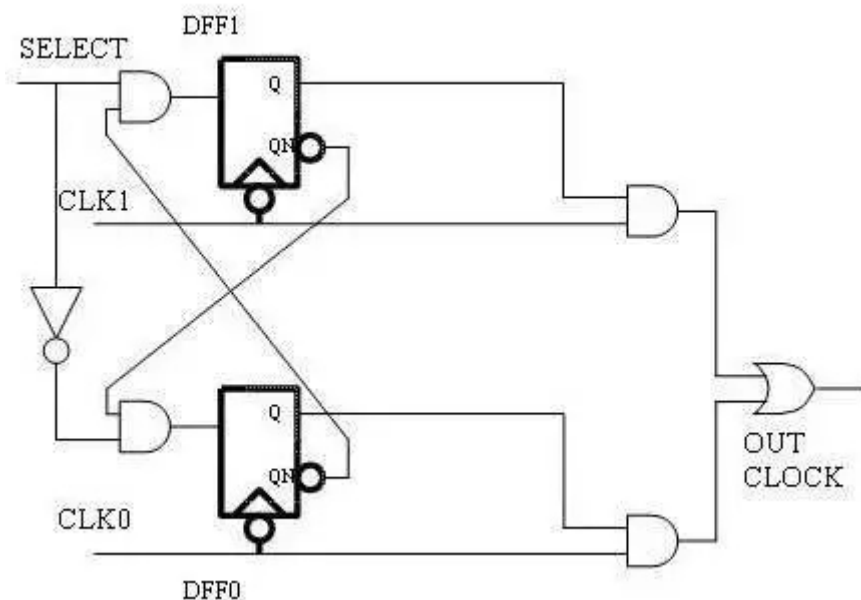
所复用的两个时钟频率可以完全无关，也可以是彼此的倍数。在任何一种情况下，都有可能时钟切换时产生毛刺。时钟上的毛刺会造成整个系统功能的故障，因为它可能被某些寄存器认为是时钟边沿而被其他寄存器忽略。



上图显示了使用AND-OR型多路复用器逻辑的时钟切换的简单实现。

多路复用器有一个名为SELECT的控制信号，

当设置为“SELECT= 1'b0”时，它将CLK0传播到输出，当设置为“SELECT = 1'b1”时将CLK1传播到输出。当SELECT值改变时，可能会产生毛刺。



上图中的时序图显示了当SELECT控制信号发生变化时，输出OUT CLOCK如何产生毛刺。

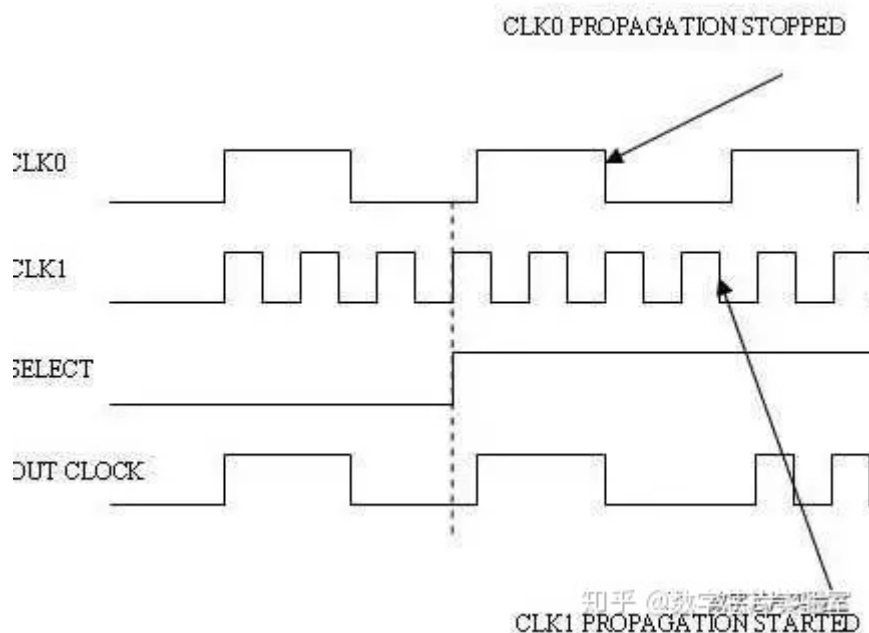
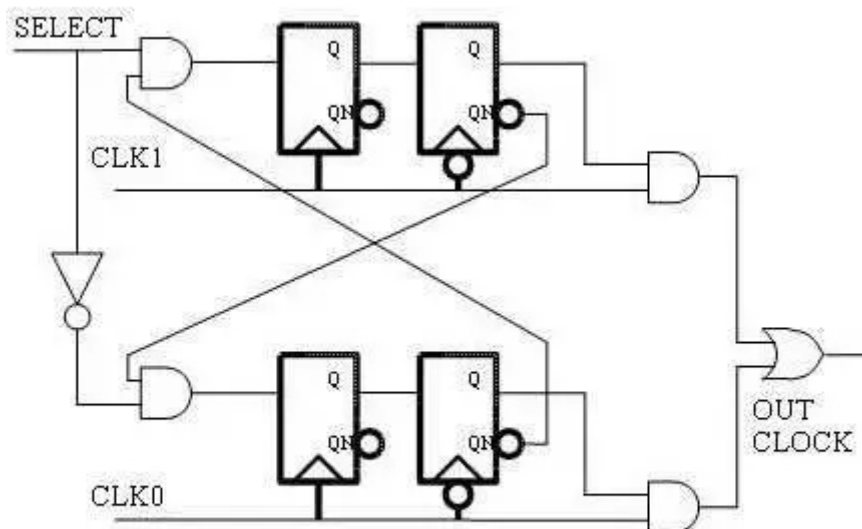
上图显示了防止时钟切换出现毛刺的解决方案。在每个时钟源的选择路径中插入一个负边沿触发的D触发器。在其他时钟不使能后，在时钟的下降沿寄存选择控制信号，可以防止输出信号出现毛刺。

在时钟的下降沿寄存选择信号可确保在时钟处于高电平时输出端不会发生变化。

在上面的时序图中，SELECT信号从0到1的转变，在CLK0的下降沿停止CLK0传播到输出，然后在CLK1的后续下降沿开始CLK1传播到输出。

该电路中有三个时序路径需要特别考虑 - SELECT控制信号到两个负边沿触发触发器，DFF0输出到DFF1的输入，DFF1的输出到DFF0的输入。

如果这三条路径中的任何一条路径上的信号与触发器时钟边沿靠近，则该寄存器的输出很可能变为亚稳态。



如上图所示，通过为每个时钟源添加一个额外级的正边沿触发器，稳定数据来防止亚稳态的传播。

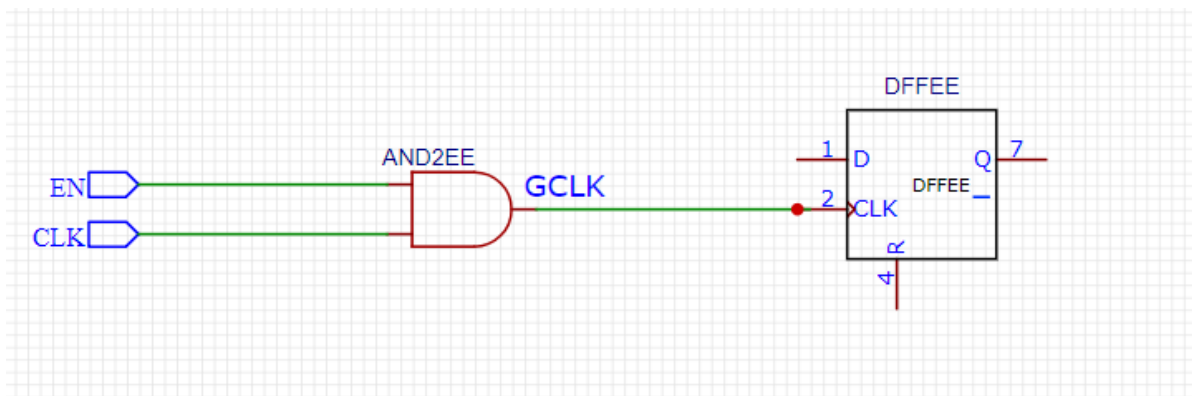
门控时钟

from <https://zhuanlan.zhihu.com/p/139363948>

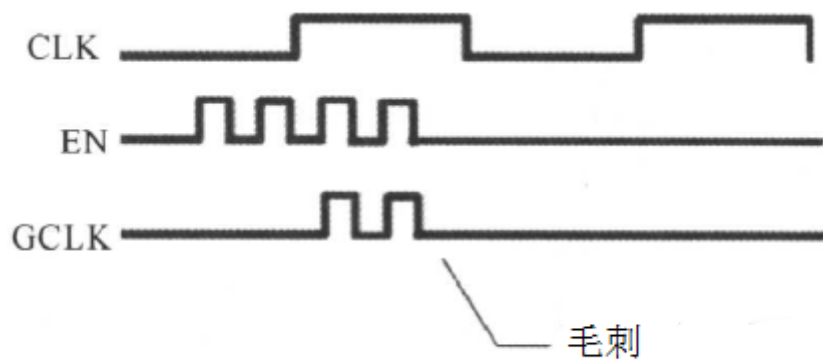
1 门控时钟的结构

1.1 与门门控

如果让我们设计一个门控时钟的电路，我们会怎么设计呢？最直接的方法，不需要时钟的时候关掉时钟，这就是与操作，我们只需要把enable和CLK进行“与”操作不就行了么，电路图如下：



这种直接将控制EN信号和时钟CLK进行与操作完成门控的方式，可以完成EN为0时，时钟被关掉。但是同时带来另外一个很大的问题：毛刺



如上图所示，EN是不受控制的，随时可能跳变，这样纯组合输出GCLK就完全可能会有毛刺产生。时钟信号上产生毛刺是很危险的。实际中，这种直接与门的方式基本不会被采样。

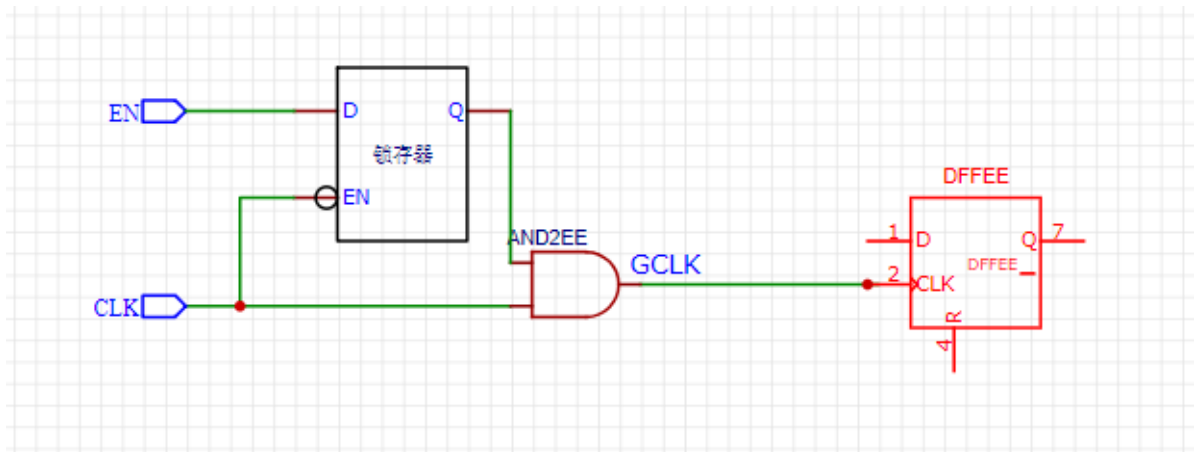
所以我们需要改进电路，为了使门控时钟不产生毛刺，我们必须对EN信号进行处理，使其在CLK的高低电平期间保持不变，或者说EN的变化就是以CLK为基准的。

1 很自然的我们会想到触发器，只要把EN用CLK寄存一下，那么输出就是以CLK为基准的；

2 其实还有一种办法是锁存器，把EN用锁存器锁存的输出，也是以CLK为基准的。

1.2 锁存门控

我们先看一下第二种电路，增加锁存器的电路如下：



对应的时序如下：



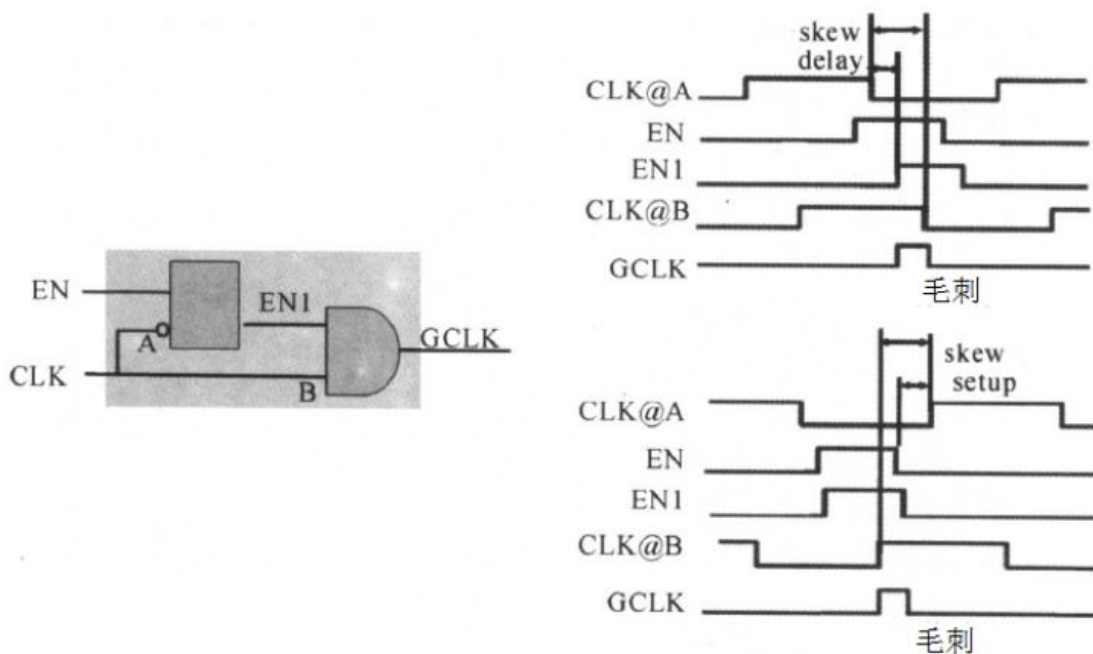
可以看到，只有在CLK为高的时候，GCLK才可能会输出高，这样就能消除EN带来的毛刺。这是因为D锁存器是电平触发，在clk=1时，数据通过D锁存器流到了Q；在Clk=0时，Q保持原来的值不变。

虽然达到了我们消除毛刺的目的，但是这个电路还有两个缺点：

1如果在电路中，锁存器与与门相隔很远，到达锁存器的时钟与到达与门的时钟有较大的延迟差别，则仍会出现毛刺。

2 如果在电路中，时钟使能信号距离锁存器很近，可能会不满足锁存器的建立时间，会造成锁存器输出出现亚稳态。

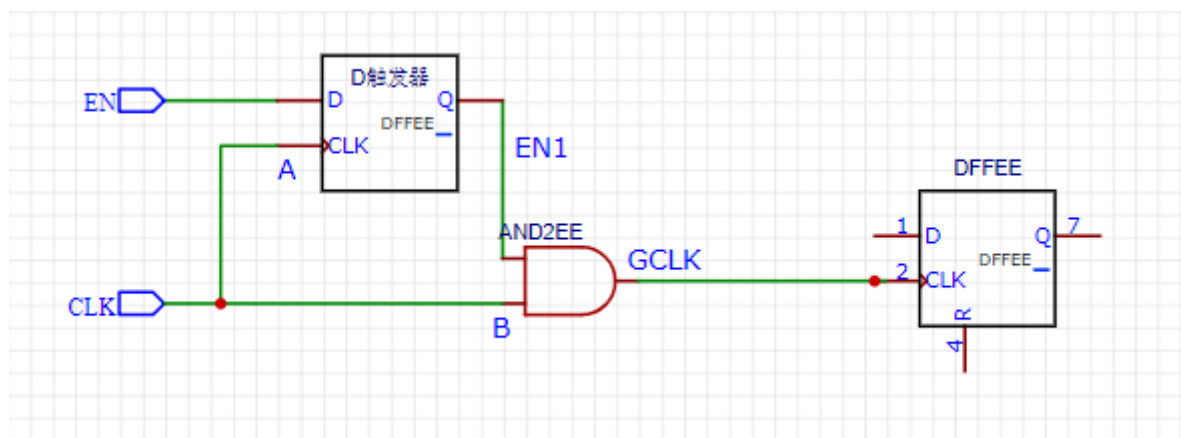
如下图分析所示：



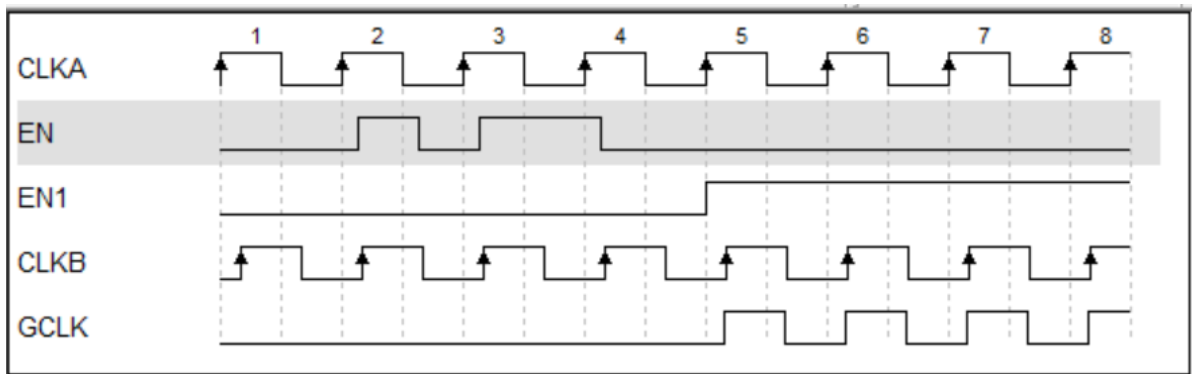
上述的右上图中，B点的时钟比A时钟迟到，并且 $Skew > delay$ ，这种情况下，产生了毛刺。为了消除毛刺，要控制Clock Skew，使它满足 $Skew > Latch\ delay$ （也就是锁存器的clk-q的延时）。上述的右下图 中，B点的时钟比A时钟早到，并且 $|Skew| > ENsetup - (D \rightarrow Q)$ ，这种情况下，也产生了毛刺。为了消除毛刺，要控制Clock Skew，使它满足 $|Skew| < ENsetup - (D \rightarrow Q)$ 。

1.3 寄存门控

如1.1中提到的，我们还有另外的解决办法，就是用寄存器来寄存EN信号再与上CLK得到GCLK，电路图如下所示：



时序如下所示：



由于DFF输出会delay一个周期，所以除非CLKB上升沿提前CLKA很多，快半个周期，才会出现毛刺，而这种情况一般很难发生。但是，这种情况CLKB比CLKA迟到，是不会出现毛刺的。

当然，如果第一个D触发器不能满足setup时间，还是有可能产生亚稳态。

1.4 门控时钟结构选择

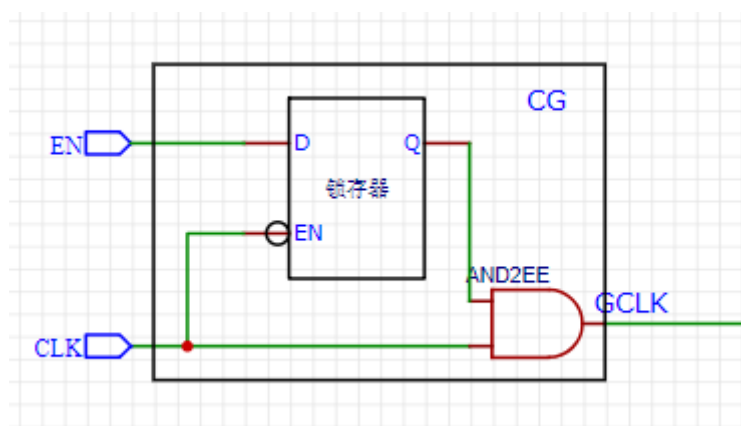
那么到底采用哪一种门控时钟的结构呢？是锁存结构还是寄存器结构呢？通过分析，我们大概会选择寄存器结构的门控时钟，这种结构比锁存器结构的问题要少，只需要满足寄存器的建立时间就不会出现问题。

那么实际中是这样么？答案恰恰相反，SOC芯片设计中使用最多的却是锁存结构的门控时钟。

原因是:在实际的SOC芯片中，要使用大量的门控时钟单元。所以通常会把门控时钟做出一个标准单元，有工艺厂商提供。那么锁存器结构中线延时带来的问题就不存在了，因为是做成一个单元，线延时是可控和不变的。而且也可以通过挑选锁存器和增加延时，总是能满足锁存器的建立时间，这样通过工艺厂预先把门控时钟做出标准单元，这些问题都解决了。

那么用寄存器结构也可以达到这种效果，为什么不用寄存器结构呢？那是因为面积！一个DFF是由两个D锁存器组成的，采样D锁存器组成门控时钟单元，可以节省一个锁存器的面积。当大量的门控时钟插入到SOC芯片中时，这个节省的面积就相当可观了。

所以，我们在工艺库中看到的标准门控时钟单元就是锁存结构了：



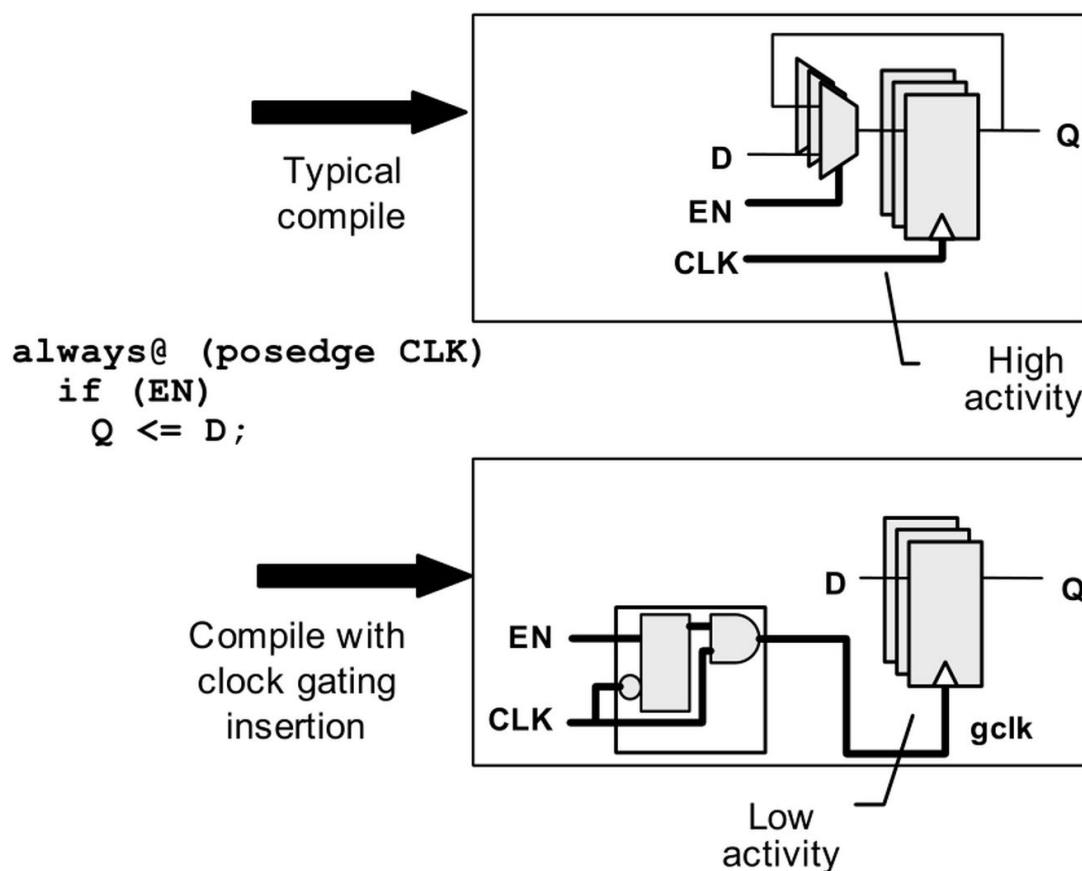
当然，这里说的是SOC芯片中使用的标准库单元。如果是FPGA或者用RTL实现，个人认为还是用寄存器门控加上setup约束来实现比较稳妥。

2 RTL中的门控时钟

通常情况下，时钟树由大量的缓冲器和反相器组成，时钟信号为设计中翻转率最高的信号，时钟树的功耗可能高达整个设计功耗40%。

加入门控时钟电路后，由于减少了时钟树的翻转，节省了翻转功耗。同时，由于减少了寄存器时钟引脚的翻转行为，寄存器的内部功耗也减少了。采用门控时钟，可以非常有效地降低设计的功耗，一般情况下能够节省20%~60%的功耗。

那么RTL中怎么才能实现门控时钟呢？答案是不用实现。现在的综合工具比如DC会自动插入门控时钟。如下图所示：



这里有两点需要注意：

1. 插入门控时钟单元后，上面电路中的MUX就不需要了，如果数据D是多bit的（一般都是如此），插入CG后的面积可能反而会减少；
2. 如果D是单bit信号，节省的功耗就比较少，但是如果D是一个32bit的信号，那么插入CG后节省的功耗就比较多。

这里的决定因素就是D的位宽了，如果D的位宽很小，那么可能插入的CG面积比原来的MUX大很多，而且节省的功耗又很少，这样得不偿失。只有D位宽超过了一定的bit数后，插入CG的收益就比较大。

那么这个临界值是多少呢？不同的工艺可能不一样，但是DC给的默认值是3。

也就是说，如果D的位宽超过了3bit，那么DC就会默认插入CG，这样综合考虑就会有收益。

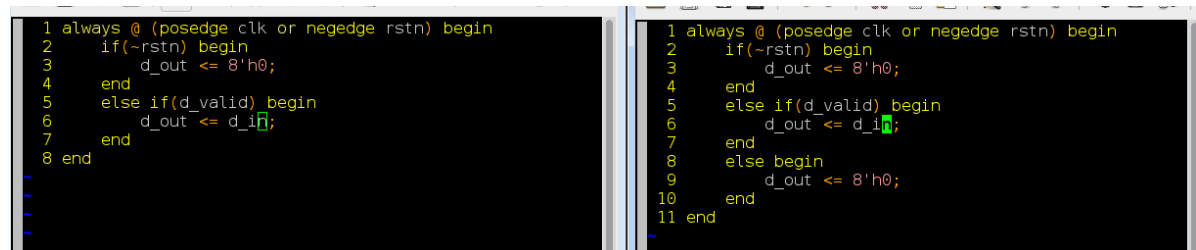
我们可以通过DC命令：

```
set_clock_gating_style -minimum_bitwidth 4
```

来控制芯片中，对不同位宽的寄存器是否自动插入CG。一般情况都不会去修改它。

2.1 RTL 门控时钟编码风格

虽然现在综合工具可以自动插入门控时钟，但是如果编码风格不好，也不能达到自动插入CG的目的。比较下面两种RTL写法：



```
1 always @(posedge clk or negedge rstn) begin
2     if(~rstn) begin
3         d_out <= 8'h0;
4     end
5     else if(d_valid) begin
6         d_out <= d_in;
7     end
8 end

1 always @(posedge clk or negedge rstn) begin
2     if(~rstn) begin
3         d_out <= 8'h0;
4     end
5     else if(d_valid) begin
6         d_out <= d_in;
7     end
8     else begin
9         d_out <= 8'h0;
10    end
11 end
```

左边的RTL代码能够成功的综合成自动插入CG的电路；

右边的RTL不能综合成插入CG的电路；

右边电路在d_valid为低时，d_out也会一直变化，其实没有真正的数据有效的指示信号，所以综合不出来插入CG的电路。

需要注意的是，有的前端设计人员，为了仿真的时候看的比较清楚，很容易会写成右边的代码，这样不仅不能在综合的时候自动插入CG来减少功耗；而且增加了d_out的翻转率，进一步增加了功耗。

在不用的时候把数据设成0并不能减少功耗，保持数据不变化才能减少toggle，降低功耗！

所以我们在RTL编写的时候一定要注意。

作为前端设计者，了解这些知识就足够了，如果想深入了解综合的控制，可以去了解

set_clock_gating_style 这个核心控制命令。

AHB、AXI中的Burst地址计算

AHB

❖根据HSIZE和HBURST来计算地址

❖例：起始地址是0x48，HSIZE=010(32bits)

HBURST	Type	Address
000	SINGLE	0x48
001	INCR	0x48, 0x4C, 0x50,... The most useful
010	WRAP4	0x48, 0x4C, 0x40, 0x44
011	INCR4	0x48, 0x4C, 0x50, 0x54
100	WRAP8	0x48, 0x4C, 0x50, 0x54, 0x58, 0x5c, 0x40, 0x44
101	INCR8	0x48, 0x4C, 0x50, 0x54, 0x58, 0x5c, 0x60, 0x64
110	WRAP16	0x48, 0x4C,..., 0x7c, 0x40, 0x44
111	INCR16	0x48, 0x4C,..., 0x7c, 0x80, 0x84

如WRAP4，增量为4，范围为4(beat)*4byte=16，则遇到16（0x10）的整数倍时回卷，而WRAP8，增量为4，范围为32(0x20)，则遇到0x20的整数倍时回环（即0x5c后，0x60-0x20=0x40）。

注意

Burst传输不能超过1k（1K=2¹⁰=0x400byte）边界

AXI

包装式突发读写有两个限制：

1 起始地址必须以传输的 size 对齐（不一定？见下面分析）。

2 突发式读写的长度必须是 2、4、8 或者 16。

关于一些地址的计算公式。

Start_Address 主机发送的起始地址

Number_Bytes 每一次数据传输所能传输的数据 byte 的最大数量

Data_Bus_Bytes 数据总线上面 byte lanes 的数量

Aligned_Address 对齐版本的起始地址

Burst_Length 一次突发式读写所传输的数据的个数

Address_N 每一次突发式读写所传输的地址数量，范围是 2-16

Wrap_Boundary 包装式突发读写的最低地址

Lower_Byte_Lane 传输的最低地址的 byte lane

Upper_Byte_Lane 传输的最高地址的 byte lane

INT(x) 对 x 进行向下取整

Start_Address = ADDR

Number_Bytes = 2^{SIZE}

Burst_Length = LEN + 1

$\text{Aligned_Address} = (\text{INT}(\text{Start_Address} / \text{Number_Bytes})) \times \text{Number_Bytes}$

$\text{Address_1} = \text{Start_Address}$

$\text{Address_N} = \text{Aligned_Address} + (\text{N} - 1) \times \text{Number_Bytes}$

地址下界 $\text{Wrap_Boundary} = (\text{INT}(\text{Start_Address} / (\text{Number_Bytes} \times \text{Burst_Length}))) \times (\text{Number_Bytes} \times \text{Burst_Length})$

如果第N个地址大于等于地址上界 $\text{Address_N} = \text{Wrap_Boundary} + (\text{Number_Bytes} \times \text{Burst_Length})$

则 $\text{Address_N} = \text{Wrap_Boundary}$ (回卷)。

第一次突发式读写：

$\text{Lower_Byte_Lane} = \text{Start_Address} - (\text{INT}(\text{Start_Address} / \text{Data_Bus_Bytes})) \times \text{Data_Bus_Bytes}$

$\text{Upper_Byte_Lane} = \text{Aligned_Address} + (\text{Number_Bytes} - 1) - (\text{INT}(\text{Start_Address} / \text{Data_Bus_Bytes})) \times \text{Data_Bus_Bytes}$

除了第一次读写之后的读写：

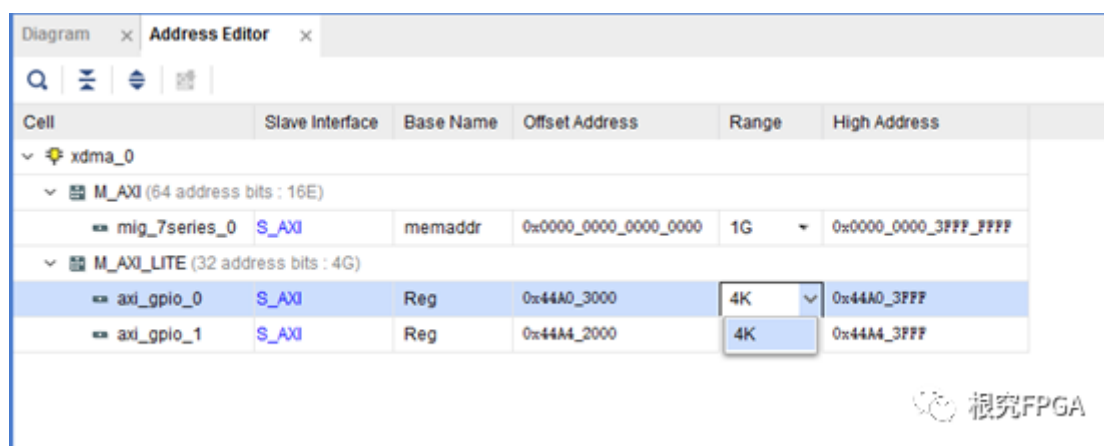
$\text{Lower_Byte_Lane} = \text{Address_N} - (\text{INT}(\text{Address_N} / \text{Data_Bus_Bytes})) \times \text{Data_Bus_Bytes}$

$\text{Upper_Byte_Lane} = \text{Lower_Byte_Lane} + \text{Number_Bytes} - 1$

$\text{DATA}[(8 \times \text{Upper_Byte_Lane}) + 7 : (8 \times \text{Lower_Byte_Lane})]$ 。

非对齐传输

最后说明在对从设备进行地址分配时，每个从设备的地址最小对齐边界为4K，即地址的低12位全为0，这样表示地址范围大小为 $2^{12}=4\text{K}$ ，4K对齐最大原因是系统中定义一个page大小是4K。所以，为了更好的设定每个slave的访问attribue，就给一个slave划分4K空间：



AXI 协议支持地址非对齐的传输，允许突发传输的首字节地址，即起始地址与突发传输位宽不对齐。举个例子，总线位宽为 32bit 时，如果起始地址为 0x1002，则产生了非对齐现象。与 32bit 位宽总线对齐的地址需要能被 4 整除，即 $\text{ADDR}[1:0] = 2'b0$ 。

对于非对齐写传输，主机会进行两项操作：

- (1) 即使起始地址非对齐，也保证所有传输是对齐的
- (2) 在首个 transfer 中增加填充数据，将首次传输填充至对齐，填充数据使用 WSTRB 信号标记为无效 (此处需要说明 TKEEP 和 TSTRB 了，在写传输中，对于填充数据字节，TKEEP 对应的位 1，对应的 WSTRB 为 0，表示该字节数据无效，仅用于数据填充，实现地址对齐)。

FPGA时序约束

1、时序约束的含义：时序约束的含义很简单，就是对设计的电路提出时序上的要求，并且检验电路是否满足要求或者尽量去满足这个要求。一般来说其可以细分为内部时钟约束、IO口时序约束、偏移约束、静态路径约束和例外路径约束等等，这些约束有相互重叠的部分。

2、时序约束的意义：我认为时序约束有2个意义。第一对已经完成了排版布线的电路，加入时序约束的静态时序分析可以检验生成的电路是否满足设计要求，并且生成时序分析报告便于用户分析关键路径以便修改电路。第二，时序约束是用户与EDA工具之间的一个界面。通过附加约束可以控制逻辑的综合、映射、布局和布线，以减小逻辑和布线延时，从而提高工作频率（如你所说）。

3、时序约束的原因：原因当然是电路的设计要求吧。

4、FPGA时序约束的本质：FPGA时序约束的本质是基于估计延时值分析路径的裕量（slack）并且基于这个裕量优化FPGA的综合、映射、布局和布线。

作者：知乎用户

链接：<https://www.zhihu.com/question/61569416/answer/190741951>

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

英文自我介绍

I received the B.S. degree in microelectronics science and engineering from the Xidian University, China, in 2019. I'm currently pursuing the master's degree in the School of Microelectronics, Xidian University. My research interests during the postgraduate period include image processing, and the application of machine learning in wireless communication. In addition, I also systematically learned the knowledge of digital ic design through elective courses and practice.

兴趣爱好 I like photography or playing football in my leisure time to relax myself and enjoy the colorful life.

企业文化 I agree with Veirisilicon's corporate culture, like fair, care, share and cheer. It's perfect for me to achieve a balance between work and life.

I grew up in Jiangsu, so I want to find a job in Jiangsu Province so that i can share more time with my parents and old friends.

优点 As for my strengths, I think I am a person with self-drive and self-discipline. Besides, I have strong self-learning ability and can adapt to the new job in a short time.

挑战 The biggest challenge I have ever encountered is our robot competition. First of all, this is my first time to participate in a competition in a new field and turn theoretical knowledge into practice. From debugging to parameter adjustment in actual competition, it is a great challenge to our mental and physical strength.

OFDM的原理、优点

OFDM是一种多载波的传输方法，它将频带划分为多个子信道并行传输数据，将高速数据流分成多个并行的低速数据流，然后调制到每个信道的子载波上进行传输。由于它将非平坦衰落无线信道转化成多个正交平坦衰落的子信道，从而可消除信道波形间的干扰，达到对抗多径衰落的目的。

正交频分复用（OFDM）是对多载波调制（MCM）的一种改进，在。它的特点是：**各子载波相互正交，所以扩频调制后的频谱可以相互重叠，不但减少了子载波间的相互干扰，还大大提高了频谱利用率。**

选择OFDM的一个很大的原因是该系统能够很好的对抗频率选择性衰落和窄带干扰。在单载波系统中，一次衰落或者干扰会导致整个链路失效，但是在多载波系统中，某一时刻只会有少部分的子信道受到深衰落的影响。

发送端将被传输的**数字信号**转换成子载波幅度和相位的映射，并进行离散傅里叶变换（IDFT），将数据的频谱表达式变到时域上。IFFT和IDFT变换的作用相同，只是有更高的计算效率，所以适用于所有的应用系统。接收端进行与发送端相反的操作，用FFT变换分解，子载波的幅度和相位最终转换回数字信号。