笔面试题集与知识点

1. 相关知识点总结

1.1 同步与异步

1.1.1 什么是同步逻辑与异步逻辑？

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。

同步时序逻辑电路的特点：各触发器的时钟端全部连接在一起，并接在系统时钟端，只有当时钟脉冲到时电路状态才能改变，其状态维持到下一个时钟脉冲到来。

异步时序逻辑电路的特点：电路中除可以使用带时钟的触发器外，还可以使用不带时钟的触发器和延迟元件作为存储单元，电路中没有统一的时钟，电路状态可以由外部输入的变化直接改变。

1.1.2 同步电路与异步电路区别

同步电路：存储电路中所有触发器的时钟输入端都接同一个时钟脉冲源，因而所有触发器的状态都与所加的时钟同步。

异步电路：电路中没有统一的时钟，有些触发器时钟与时钟脉冲相连，有一些没有。

1.1.3 异步信号同步（跨时钟域同步）

..\..\word\异步 FIFO 设计与仿真.docx

1.2 时序设计

1.2.1 时序设计实质

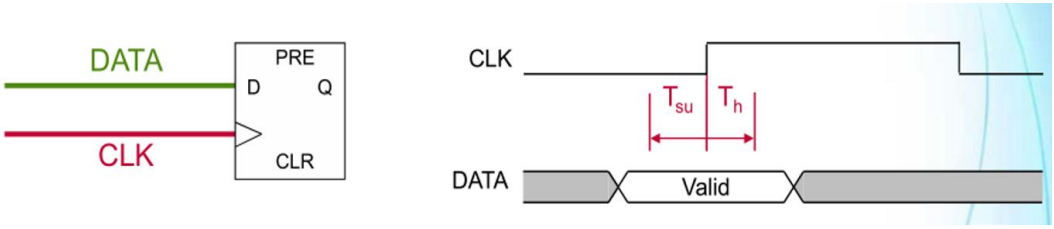
时序设计的实质就是满足每一个触发器的建立/保持时间的要求。

1.2.2 建立保持时间概念

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

保持时间：触发器在时钟沿到来之后，其数据输入端的数据必须保持不变的

最小时间。



1.2.3 为什么触发器要满足建立时间和保持时间

这是因为触发器内部数据的形成是需要一定的时间的，如果不满足建立和保持时间，触发器将进入亚稳态，这会导致触发器输出将不稳定，在 0-1 之间变化，这时需要一个恢复时间，其输出才能稳定，不仅大大增加延迟，而且输出也不一定正确。

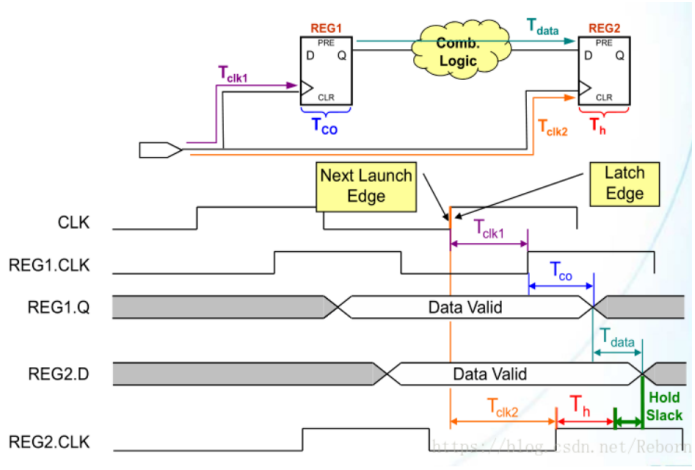
1.2.4 亚稳态的产生

亚稳态：指触发器无法在某个规定的时间段内达到一个可以确认的状态，原因就是因为无法满足触发器的建立保持时间。一位同步器有效条件：第一级触发器进入亚稳态后的恢复时间+第二季触发器的建立时间<=时钟周期。更确切地说，输入脉冲宽度必须大于同步时钟周期与第一级触发器所需的保持时间之和。最保险的脉冲宽度是两倍同步时钟周期。 所以，这样的同步电路对于从较慢的时钟域来的异步信号进入较快的时钟域比较有效，对于进入一个较慢的时钟域，则没有作用。具体亚稳态详细信息与如何解决请看：亚稳态的世界.docx

1.2.5 系统最高时钟频率计算

同步电路的速度指的是同步系统的时钟速度，时钟速度越快，单位时间处理的有效数据越大。如图所示为触发器时序延迟图。其中 Tco 是触发器输入数据到触发器输出数据的延迟时间（Tco＝Tsetup+Thold），Tdata 是组合逻辑延迟（路径延迟），两寄存器之间时钟偏移 Tth= clk2-clk1，因此，可以计算出想要满足第二个寄存器的建立时间，就必须要时钟周期 **T>=Tco+Tdata+Tsetup-Tth**。因此看出来当 Tth 为正时钟偏斜时，可以改善时序，有利于满足建立时间。不过，建立时间要满足上述式子，保持时间呢？如图所示，要满足数据在 hold 中保持不变，就必须满足 Tclk1+Tco+Tdata>=Tclk2+Thold，即 **Thold<=Tco+Tdata-Tth**。这个公式可以看出，一般 Tco 与 Thold 都是固定的，Tth 若是正时钟偏斜，要满足式子

就必须 Tdata 足够大，而前面要提高时钟频率就必须 Tdata 足够小，两者要兼顾，Tdata 既不能太大也不能太小。



1.2.6 时序约束的概念和基本策略

时序约束主要包括周期约束，偏移约束，静态时序路径约束三种。通过附加时序约束可以综合布线工具调整映射和布局布线，使设计达到时序要求。

附加时序约束的一般策略是先附加全局约束，然后对快速和慢速例外路径附加专门约束。附加全局约束时首先定义设计的所有时钟，对各时钟域的同步元件进行分组，对分组附加周期约束，然后对 FPGA 输入输出 PAD 附加偏移约束、对组合逻辑 PAD 路径附加约束。附加专门约束时，首先约束分组之间的路径，然后约束快慢例外路径和多周期路径以及其他特殊路径。

1.2.7 附加约束的作用

提高设计的工作频率（减少了逻辑和布线延迟）；获得正确的时序分析报告（静态时序分析工具以约束作为判断时序是否满足设计要求的标准，因此要求设计者正确输入约束，以便静态时序分析工具可以正确的输出时序报告）；指定 FPGA 的电气标准和引脚位置。

1.2.8 全局时钟概念

详情请看：全局时钟.docx

1.2.9 时序约束设计

详情请看：时序约束.docx

要去写指令，写 tcl XDC 约束指令，这样才能记忆深刻，会考指令

1.2.10 静态时序与动态时序分析

静态时序分析是采用穷尽分析方法来提取出整个电路存在的所有时序路径，计算信号在这些路径上的传播延时，检查信号的建立和保持时间是否满足时序要求，通过对最大路径延时和最小路径延时的分析，找出违背时序约束的错误。它不需要输入向量就能穷尽所有的路径，且运行速度很快、占用内存较少，不仅可以对芯片设计进行全面的时序功能检查，而且还可利用时序分析的结果来优化设计，因此静态时序分析已经越来越多地被用到数字集成电路设计的验证中。

动态时序模拟就是通常的仿真，因为不可能产生完备的测试向量，覆盖门级网表中的每一条路径。因此在动态时序分析中，无法暴露一些路径上可能存在的时序问题；

1.2.11 建立时间裕量和保持时间裕量 0514

上述知识点已经说过了建立时间与保持时间和相关延迟的关系：

T+Tkew >= Tco + Tdata + Tsteup ,因此建立裕量 = T + Tkew – (Tco + Tdata + Tsteup)

Thold + Tkew <= Tco + Tdata + Tsteup,因此保持裕量 = Thold + Tkew - (Tco + Tdata + Tsteup)

答案减反了，必须是正的，保持裕量不要Tsteup

1.3. FPGA 内部资源分析

FPGA 由 6 部分组成，分别为可编程输入/输出单元（IOB）、基本可编程逻辑单元（CLB）、嵌入式块 RAM（block ram）、丰富的布线资源、底层嵌入功能单元和内嵌专用硬核等

1.3.1 锁存器与触发器区别

锁存器作用是在新的值到来之前保持原来信号的值，设计中应该避免所有可能位置使用锁存器，而应该用触发器代替。锁存器电平触发，是连通模式，即在数据输入和输出之间是存在直接通路的，输入端的毛刺会传递到输出端。如图 1.7 所示，X,Y 信号同时变高时，锁存器电平触发，同时开启会使电路产生震荡。

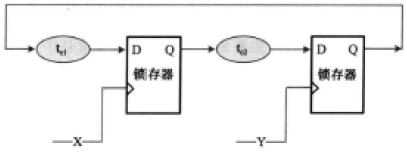


图 1.7 锁存器的竞争条件

锁存器通常使电路不可测，而且 FPGA 是寄存器密集型，使用锁存器回占用更多的逻辑。什么情况会产生设计之外的锁存器呢？例如：代码中有不完整的“if”或“else”语句，或者忽略 else 分句也会产生，如图 1.8 所示，由于 a 信号只在”“if”语句中赋值而在“else”中却忽略了，这会使综合器 自动形成一个锁存器。（主要其实还是因为为了时序分析，寄存器更适合分析）

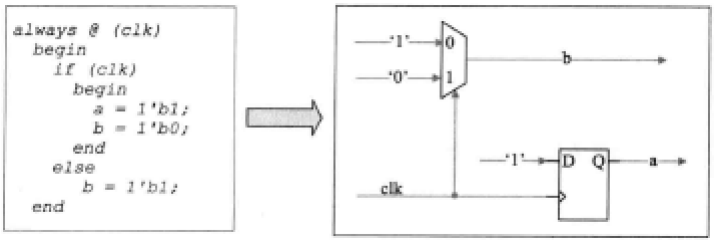


图 1.8 不完整“if else”描述产生触发器

如图 1.9 所示的代码，在异步复位边沿触发的 always 块中综合却产生了锁存器，原因是因为在异步复位模块里信号的输出仍然由外接输入决定，此时就会产生锁存器，故一般异步复位端的赋值不能由未知信号输入。若异步复位端一定要输入可以更改为图 1.10 所示，这将 always 条件的异步复位端更改为一个使能信号。

```
module Decode(input A,input B,input C,output reg[31:0] edata,output reg[31:0] eCapData,input bCap,output reg CapSt,input n

reg[1:0] state;

always@(posedge clk or negedge n_rst)
begin
    if(!n_rst)
    begin
        edata <= rstVal;
        state <= {A,C};
    end
    else
    begin
        state <= {A,B};
        edata <= {31'd0,A};
  *****此处省略一万*****
```

图 1.9 边沿触发的 always 块中产生锁存器原因

```
module Decode(input A,input B,input C,output reg[31:0] edata,output reg[31:0] eCapData,input bCap,output reg CapSt,input n 复制

reg[1:0] state;

always@(posedge clk)
begin
    if(!n_rst)
    begin
        edata <= rstVal;
        state <= {A,C};
    end
    else
    begin
        state <= {A,B};
        edata <= {31'd0,A};
  *****此处省略一万*****
```

图 1.10 更改锁存器不改变代码原意

结论：推断锁存器的一般规则是：组合逻辑中，如果一个变量未能在 always 语句的所有可能执行的条件下赋值，就可能形成锁存器。若还有问题，可以查看下面浏览器中的知识：http://xilinx.eetrend.com/content/2020/100047135.html

1.3.2 FPGA 内部存储器资源有哪两种

FPGA 内部结构详解.docx

FPGA 芯片内有两种存储器资源：一种叫 BLOCK RAM,另一种是由 LUT 配置成的内部存储器（也就是分布式 RAM）。BLOCK RAM 由一定数量固定大小的存储块构成的，使用 BLOCK RAM 资源不占用额外的逻辑资源，并且速度快。但是使用的时候消耗的 BLOCK RAM 资源是其块大小的整数倍。

1.3.3 LUT 原理与结构

查找表（look-up-table）简称为 LUT，LUT 本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成一个有 4 位地址线的 16x1 的 RAM。当用户通过原理图或 HDL 语言描述了一个逻辑电路以后，PLD/FPGA 开发软件会自动计算逻辑电路的所有可能的结果，并把结果事先写入 RAM,这样，每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。

1.3.4 逻辑表达式

同一律	$A \cdot A = A$	$A + A = A$
德·摩根定理	$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$
还原律	$\overline{\overline{A}} = A$	

1.3.5 SRAM、DRAM 与 SDRAM、Flash 的区别

SRAM 是静态随机访问存储器，由晶体管存储数据，无需刷新，读写速度快。DRAM 是动态随机访问存储器，由电容存储数据，由于电容漏电需要动态刷新，电容充放电导致读写速度较 SRAM 低。但 DRAM 成本较低，适合做大容量片外缓存。

SDRAM：同步动态 RAM，需要刷新，速度较快，容量大。它比一般 DRAM 和 EDO RAM 速度都快，它已经逐渐成为 PC 机的标准内存配置。

DDR SDRAM：双通道同步动态 RAM（朱老师讲的双倍率，不是太重要），需要刷新，速度快，容量大

flash：闪存，存取速度慢，容量大，掉电后数据不会丢失

1.4 时钟与复位

时钟与复位.docx

1.4.1 时钟抖动是什么

流水线 艺术.docx

时钟抖动是指芯片的某一个给定点上时钟周期发生暂时性变化，也就是说时钟周期在不同的周期上可能加长或缩短。它是一个平均值为 0 的平均变量。

主要受外界干扰引起，通过各种抗干扰手段可以避免

1.4.2 时钟偏斜是什么

流水线 艺术.docx

时钟偏斜 Skew:指全局时钟产生的各个子时钟信号到达不同触发器的时间点不同，是时钟相位的不一致，skew 由数字电路内部各路径布局布线长度和负载不同导致，利用全局时钟网络可尽量将其消除。

1.4.3 如何产生时钟延迟

首先说说异步电路的延时实现：异步电路一半是通过加 buffer、两级与非门等来实现延时（我还没用过所以也不是很清楚），但这是不适合同步电路实现延时的。在同步电路中，对于比较大的和特殊要求的延时，一半通过高速时钟产生计数器，通过计数器来控制延时；对于比较小的延时，可以通过触发器打一拍，不过这样只能延迟一个时钟周期。

1.4.4 时钟分频

时钟分频技术.docx

1.4.5 跨时钟同步

多时钟处理.docx

不同的时钟域之间信号通信时需要进行同步处理，这样可以防止新时钟域中第一级触发器的亚稳态信号对下级逻辑造成影响。

信号跨时钟域同步：当**单个信号**跨时钟域时，可以采用两级触发器来同步（**注意此处分为电平和脉冲信号 ，快慢时钟领域，详情看上述文档**），因此，对于异步信号，不要使用两个同步电路同步后，分别给到 clkb 下的不同地方去做逻辑，这与 signal_a 不经过同步直接用在 clkb 下的多个地方做逻辑犯的错误后果是一样的！快时钟与慢时钟差别，一定要保证慢时钟能够检测到来自快时钟信号的两个点（慢时钟两个上升沿都可以）

数据或地址总线跨时钟域时可以采用异步 FIFO 来实现时钟同步；第三种方法就是采用握手信号。

1.4.6 同步、异步复位

时钟与复位.docx

1.4.7 时钟切换方法

时钟切换分成两种方式，普通切换和去毛刺无缝切换。

普通切换，就是不关心切出的时钟是否存在毛刺，这种方式电路成本小。如果时钟切换时，使用此时钟的模块电路处于非工作状态，或者模块内电路被全局复位信号 reset 住的，即使切出毛刺也不会导致 DFF 误触发，这样的模块可以选择用此种切换方式。

写法很简单 assign clk_o = sel_clkb ? clkb : clka ,当 sel_clkb 为 1 时选择 clkb，否则选择 clka。不过在实际设计中，建议直接调用库里的 MUX 单元并 set_dont_touch，不要采用这里的 assign 写法，因为这种写法最后综合得到的可能不是 MUX 而是复杂组合逻辑，给前后端流程的时钟约束和分析带来不便。

无缝切换，就是切换时无毛刺时钟平稳过渡。在时钟切换中，只要出现比 clka 或者 clkb 频率更高的窄脉冲，不论是窄的高电平还是窄的低电平，都叫时钟毛刺。工作在切换后时钟 clk_o 下的电路模块，综合约束是在 max{clka,clkb}频率下的，也就是说设计最后 signoff 的时候，只保证电路可以稳定工作的最高频率是 max{clka,clkb},如果切换中出现更高频的时钟毛刺，电路可能出现无法预知的结果而出错。无缝切换，一般用在处于工作状态的模块需要调频或者切换时钟源，比如内部系统总线，cpu 等。你刚用手机打完游戏后马上关屏听音乐，这两种场景中，CPU 在满足性能前提下为了控制功耗，其工作频率会动态地从很高调至较低，此时就可能是在 CPU 一直处于工作状态下，通过无缝切换时钟源头实现的。

无缝切换需要解决两个问题，一是异步切换信号的跨时钟域同步问题，这里需要使用《verilog 基本电路设计之一》里的同步电路原理消除亚稳态；二是同步好了的切换信号与时钟信号如何做逻辑，才能实现无毛刺。

具体详情请看：Verilog 基本电路设计之二（时钟无缝切换）

http://bbs.eetop.cn/forum.php?mod=viewthread&tid=605514&fromuid=1753025

(出处: EETOP 创芯网论坛)

1.4.8 去抖滤波

Verilog 基本电路设计之四（去抖滤波）

http://bbs.eetop.cn/forum.php?mod=viewthread&tid=605729&fromuid=1753025

根据希望滤除的宽度相关，换算到 clk 下是多少个 cycle 数，从而决定使用多少级 DFF。如果希望滤除的宽度相对 cycle 数而言较大，可以先在 clk 下做一个计数器，产生固定间隔的脉冲，再在脉冲信号有效时使用多级 DFF 去抓 signal_i；或者直接将 clk 分频后再使用。

1.4.9 时钟域与时钟树的使用

时钟域与时钟树.docx

1.4.10 Xilinx 时钟原语

全局时钟.docx

1.5 IC 相关

1.5.1 IC 设计前端到后端的流程和 EDA 工具？

设计前端也称逻辑设计，后端设计也称物理设计，两者并没有严格的界限，一般涉及到与工艺有关的设计就是后端设计。

1：规格制定：客户向芯片设计公司提出设计要求。

2：详细设计：芯片设计公司（Fabless）根据客户提出的规格要求，拿出设计解决方案和具体实现架构，划分模块功能。目前架构的验证一般基于 systemC 语言，对价后模型的仿真可以使用 systemC 的仿真工具。例如：CoCentric 和 Visual Elite 等。

3：HDL 编码：设计输入工具：ultra ， visual VHDL 等

4：仿真验证：modelsim

5：逻辑综合：synplify

6：静态时序分析：synopsys 的 Prime Time

7：插入扫描链 DFT

8：形式验证：Synopsys 的 Formality.

9：布局规划

10：布局布线

11：后仿

12：流片，测试

1.5.2 寄生效应在 IC 设计中怎样加以克服和利用

所谓寄生效应就是那些溜进你的 PCB 并在电路中大施破坏、令人头痛、原因不明的小故障。它们就是渗入高速电路中隐藏的寄生电容和寄生电感。其中包括由封装引脚和印制线过长形成的寄生电感；焊盘到地、焊盘到电源平面和焊盘到印制线之间形成的寄生电容；通孔之间的相互影响，以及许多其它可能的寄生效应。

理想状态下，导线是没有电阻，电容和电感的。而在实际中，导线用到了金属铜，它有一定的电阻率，如果导线足够长，积累的电阻也相当可观。两条平行的导线，如果互相之间有电压差异，就相当于形成了一个平行板电容器（你想象一下）。通电的导线周围会形成磁场（特别是电流变化时），磁场会产生感生电场，会对电子的移动产生影响，可以说每条实际的导线包括元器件的管脚都会产生感生电动势，这也就是寄生电感。

在直流或者低频情况下，这种寄生效应看不太出来。而在交流特别是高频交流条件下，影响就非常巨大了。根据复阻抗公式，电容、电感会在交流情况下会对电流的移动产生巨大阻碍，也就可以折算成阻抗。这种寄生效应很难克服，也难摸到。只能通过优化线路，尽量使用管脚短的 SMT 元器件来减少其影响，要完全消除是不可能的。

1.5.3 IC 设计流程

IC 设计分为前端和后端。前端设计主要将 HDL 语言-->网表，后端设计是网表-->芯片版图。

前端主要有需求分析与架构设计、RTL 设计、仿真验证、逻辑综合、STA、

形式验证。后端主要包括 DFT、布局规划、布线以及版图物理验证。

1.5.4 低功耗技术

功耗可用公式描述：Power＝KFCV^2，即功率等于常数系数*工作频率*负载电容值*电压的平方。

故从以下几个方面降低功耗方式：

a.控制工作频率：降低频率增大数据路径宽度，动态频率调整，门控时钟（时钟使能有效时钟才进入寄存器时钟输入引脚）

b.减少电容负载：使用几何尺寸更小的逻辑门，其电容负载较小，功率也随之减少。

c.降低工作电压：动态改变工作电压、零操作电压（直接关闭系统中一部分的电源）。

1.6 状态机

状态机设计.docx

1.6.1 moore 与 mealy 状态机区别

Moore 状态机的输出仅与当前状态值有关，且只在时钟边沿到来时才会有状态变化。

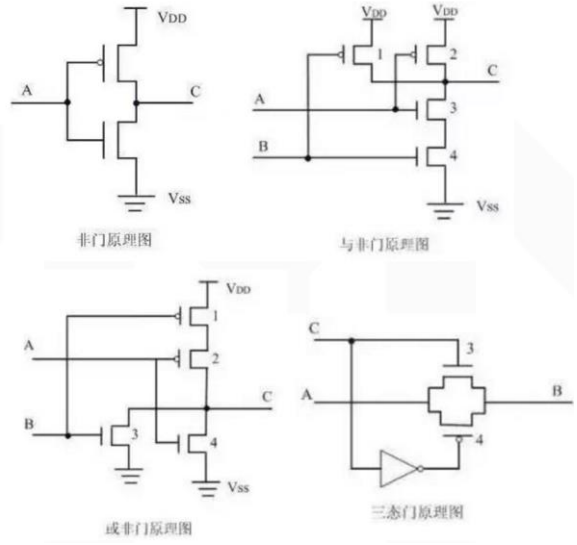
Mealy 状态机的输出不仅与当前状态值有关，而且与当前输入值有关。

1.7 数电知识

1.7.1 MOS 管基本概念与画图

MOS 中文意思是金属氧化物半导体场效应管，由栅极（G）、漏级（D）、源级（S）组成。分为 PMOS 和 NMOS 两种类型，区别在于 G 级高电平时，N 型管导通，P 型管截止。两者往往是成对出现的，即 CMOS。只要一只导通，另一只则不导通，现代单片机主要是采用 CMOS 工艺制成的。

画图一般需要根据一个简单的逻辑表达式，画出 CMOS 电路图结构。需要掌握常用逻辑门的实现方式。总体来看还是挺好记的，与非门和或非门都是上下各两个 MOS 管，且上面是 PMOS，下面是 NMOS。不同之处在于与非是“上并下串”，或非是“上串下并”。



1.7.2 N 位 bit 位宽表示范围

有符号数：（补码）-2^(N-1) ~ 2^(N-1)-1 如 N = 8，则表示范围是：-128 ~ 127.

无符号数：0~2^N-1 如 N = 8，则表示范围是：0~255.

定点数：3Q13 范围是：-4~4-2^(-13) 精度是：2^(-13)

相加相乘后需要的数据位宽若无已知数据范围，按照基本规律：相加位宽+1，相乘位宽*2.若已知操作数范围，根据运算操作数所能表示数的绝对值最大值求出运算结果极限值。

问：例如两个 8bit 有符号数相乘，其结果需要的位宽是多少？

答：8bit 有符号数补码所能表示的数据范围是：-128 到 127，具有最大绝对值的数是-128，所以极限情况下是-128*（-128） = 16384 = 2^14. 15bit 有符号数所能表示范围是：-2^14 到 2^14-1，并不能表示 2^14。综上，需要 16bit 位宽。正好位宽扩大一倍。

1.7.3 竞争冒险如何识别消除

竞争：在组合逻辑电路中，信号经过多条路径到达输出端，每条路径经过的逻辑门不同存在时差，在信号变化的瞬间存在先后顺序。这种现象叫竞争。

冒险：由于竞争而引起电路输出信号中出现了非预期信号，产生瞬间错误的现象称为冒险。表现为输出端出现了原设计中没有的窄脉冲，即毛刺。

常见的逻辑代数法判断是否有竞争冒险存在：只要输出逻辑表达式中含有某个信号的原变量 A 和反变量/A 之间的“与”或者“或”关系，且 A 和/A 经过不

同的传播路径，则存在竞争。解决办法一是修改逻辑表达式避免以上情况，二是采样时序逻辑，仅在时钟边沿采样，三是在芯片外部并联电容消除窄脉冲。

1.7.5 CMOS 与 TTL 电路区别

两者区别主要体现在三个方面：

a.结构：CMOS 电路由场效应管构成，TTL 由双极性晶体管构成。

b.电平范围：CMOS 逻辑电平范围大（5~15V），TTL 只工作在 5V 以下，因此 CMOS 噪声容限比 TTL 大，抗干扰能力强。

c.功耗与速率：CMOS 的功耗比 TTL 小，但工作频率低于 TTL。

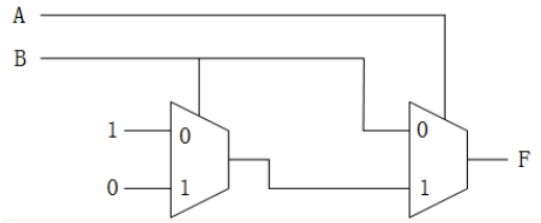
1.7.6 线与逻辑

线与逻辑是两个输出信号相连可以实现与的功能。在硬件上,要用 oc 门来实现,由于不用 oc 门可能使灌电流过大,而烧坏逻辑门. 同时在输出端口应加一个上拉电阻。Oc 门就是集电极开路门。

1.7.7 引脚电平知识

常用逻辑电平：TTL、CMOS、LVTTL、LVCMOS、ECL（Emitter Coupled Logic）、PECL（Pseudo/Positive Emitter Coupled Logic）、LVDS（Low Voltage Differential Signaling）、GTL（Gunning Transceiver Logic）、BTL（Backplane Transceiver Logic）、ETL（enhanced transceiver logic）、GTLP（Gunning Transceiver Logic Plus）；RS232、RS422、RS485（12V，5V，3.3V）；也有一种答案是：常用逻辑电平：12V，5V，3.3V。TTL和CMOS 不可以直接互连，由于TTL是在0.3-3.6V之间，而CMOS则是有在12V的有在5V的。CMOS输出接到TTL是可以直接互连。TTL接到 CMOS需要在输出端口加一上拉电阻接到5V或者12V。
cmos的高低电平分别为:Vih>=0.7VDD,Vil<=0.3VDD;Voh>=0.9VDD,Vol<=0.1VDD.
ttl的为:Vih>=2.0v,Vil<=0.8v;Voh>=2.4v,Vol<=0.4v.
用cmos可直接驱动ttl;加上拉电阻后,ttl可驱动cmos.
1、当TTL电路驱动COMS电路时，如果TTL电路输出的高电平低于COMS电路的最低高电平（一般为3.5V），这时就需要在TTL的输出端接上拉电阻，以提高输出高电平的值。
2、OC门电路必须加上拉电阻，以提高输出的搞电平值。
3、为加大输出引脚的驱动能力，有的单片机管脚上也常使用上拉电阻。
4、在COMS芯片上，为了防止静电造成损坏，不用的管脚不能悬空，一般接上拉电阻产生降低输入阻抗，提供泄荷通路。
5、芯片的管脚加上拉电阻来提高输出电平，从而提高芯片输入信号的噪声容限增强抗干扰能力。
6、提高总线的抗电磁干扰能力。管脚悬空就比较容易接受外界的电磁干扰。
7、长线传输中电阻不匹配容易引起反射波干扰，加上下拉电阻是电阻匹配，有效的抑制反射波干扰。
上拉电阻阻值的选择原则包括：
1、从节约功耗及芯片的灌电流能力考虑应当足够大；电阻大，电流小。
2、从确保足够的驱动电流考虑应当足够小；电阻小，电流大。
3、对于高速电路，过大的上拉电阻可能边沿变平缓。综合考虑以上三点,通常在1k到10k之间选取。对下拉电阻也有类似道理。
OC门电路必须加上拉电阻，以提高输出的搞电平值。
OC门电路要输出"1"时才需要加上拉电阻不加根本就没有高电平
在有时我们用OC门作驱动（例如控制一个 LED）灌电流工作时就可以不加上拉电阻
OC门可以实现"线与"运算
OC门就是　集电极 开路 输出
总之加上拉电阻能够提高驱动能力。

1.7.8 使用 2 选 1mux 完成异或逻辑，至少需要几个 mux

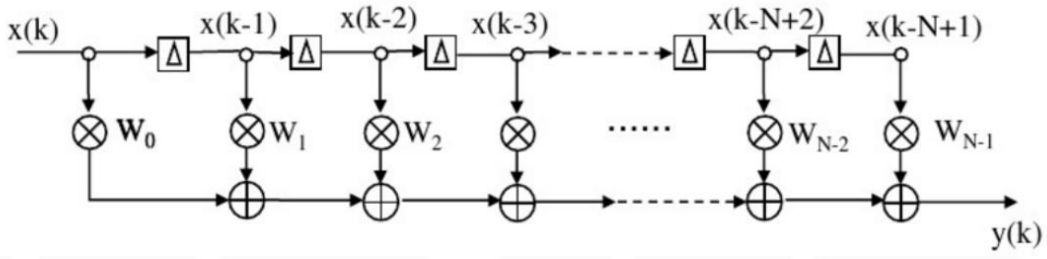


1.8 信号处理知识

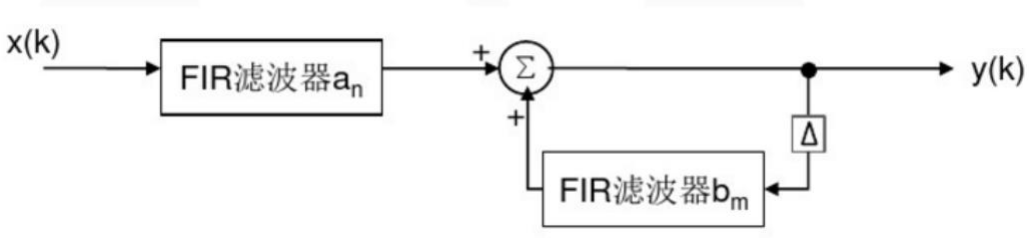
1.8.1 FIR 与 IIR 区别

FIR（有限冲激响应）滤波器：非递归，具有线性相位。IIR（无限冲激响应）滤波器：递归结构，非线性相位。相同阶数 FIR 和 IIR 滤波器，IIR 滤波器滤波效果较好，但会产生相位失真。

FIR 滤波器：对 N 个采样数据进行加权和平均处理。



IIR 滤波器：包含递归部分也包含非递归部分。表达式：（具有 N 个前馈系数和 M-1 个反馈系数）



1.9 FPGA 设计优化

1.9.1 FPGA 设计流程

与数字 IC 设计流程类似，以 xilinx vivado 工具为例，主要有以下步骤：系统规划、RTL 输入、行为仿真、逻辑综合、综合后仿真（可选）、综合后设计分析（时序及资源）、设计实现（包括布局布线及优化）、实现后设计分析（时序及资源）、板级调试、bitstream 固化。

1.9.2 面积速度优化类型

面积优化：资源共享，压缩数据，串行化，ram 适当空间

速度优化：流水线，关键路径法，并行处理，逻辑复制，寄存器配平

1.9.3 FIFO 最小深度计算

https://blog.csdn.net/balanx/article/details/54861677

异步 FIFO 最小深度计算原理

FIFO 用于缓冲块数据流，一般用在写快读慢突发传输的情况，遵循的规则如下：

$$\frac{FIFO深度}{(写入速率-读出速率)}=FIFO被填满时间>数据包传送时间=\frac{写入最大突发数据量}{写入速率}$$

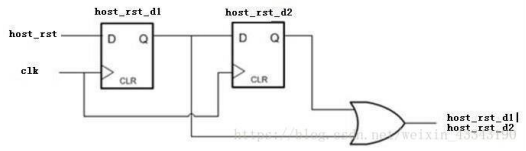
这里假设读写 FIFO 是可以同时进行的，
写时钟频率 w_clk，
读时钟频率 r_clk，
写时钟周期里，每 B 个时钟周期会有 A 个数据写入 FIFO，
读时钟周期里，每 Y 个时钟周期会有 X 个数据读出 FIFO，
则 FIFO 的最小深度的计算公式如下：

$$fifo\_depth = burst\_length - burst\_length * (X/Y) * (r\_clk/w\_clk)$$

例2：
如果100个写时钟周期可以写入80个数据，10个读时钟可以读出8个数据。令w_clk=r_clk，考虑背靠背 (20个clk不发数据 + 80clk发数据 + 80clk发数据 + 20个clk不发数据的200个clk)，代入公式可计算 FIFO 的深度，
fifo_depth = 160-160*80% = 160 - 128 = 32

如果令w_clk = 200MHz，改为100个w_clk里写入40个，r_clk = 100MHz，10个r_clk里读出8个，那么 FIFO 深度为48。计算如下，
fifo_depth = 80 - 80 * 80% * (100/200) = 80 - 32 = 48

1.9.4 glitch 消除




```
module digital_filter_(clk_in,rst,host_rst,host_rst_filter);
input  clk_in;
input  rst;
input  host_rst;
output host_rst_filter;
reg host_rst_d1;
reg host_rst_d2;

always@(posedge clk_in or negedge rst)
  begin
    if(~rst)
    begin
        host_rst_d1 <= 1'b1;
        host_rst_d2 <= 1'b1;
       end
    else
      begin
        host_rst_d1 <= host_rst;
        host_rst_d2 <= host_rst_d1;
      end
  end

assign host_rst_filter = host_rst_d1 | host_rst_d2;
endmodule
```

1.9.5 边沿检测设计

```
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  singal_posedge = ~signal_r[1] & signal_r[0];//检测上升沿
assign  singal_negedge = signal_r[1] & ~signal_r[0];//检测下降沿
```

1.9.6 always @()触发时钟个数

注意：在 always@()模块中时钟触发只能有一个时钟且只能是一个边沿，不可双边沿同时触发，如 always @(posedge clk or negedge clk)，这是错误的。因为一个触发器只有一个触发端口，没有多个触发端口的元件。

甚至这种也是不对的：

always@(posedge clk or negedge rst_n) begin

a <= 1'b0;

end

为什么 DDR 可以双边沿呢？

这是先产生奇偶数据链，然后通过 assign dout = clk ? dout_o : dout_e;

1.10 Verilog 语法

1.10.1 parameter 与 localparam、define 区别

Localparam：局部定义参数化，只能在模块中使用。

```
module adder_carry
    (

    input [3:0] a,
    input [3:0] b,
    output [3:0] sum,
    input cout
    );
    localparam N = 4;
    localparam N1 = N-1;
    wire [N:0] sum_ext;
    assign sum_ext = {1'b0,a} + {1'b0,b};
    assign sum = sum_ext[N1:0];
    assign cout = sum_ext[N];
endmodule
```

Parameter：可在模块间传递。现在一般常用 parameter 定义常数

```
module adder_carry
     #(parameter N = 4)
    (

    input [N-1:0] a,
    input [N-1:0] b,
    output [N-1:0] sum,
    input cout
    );
    localparam N1 = N-1;
    wire [N:0] sum_ext;
    assign sum_ext = {1'b0,a} + {1'b0,b};
    assign sum = sum_ext[N1:0];
    assign cout = sum_ext[N];
endmodule
```

```
module adder_instant(
    input [3:0] a4,
    input [3:0] b4,
    output [3:0] sum4,
    output cout4,
    input [7:0] a8,
    input [7:0] b8,
    output [7:0] sum8,
    output cout8
    );
    //实例化8位加法器
    adder_carry #(.N(8)) uu1(.a(a8),.b(b8),.sum(sum8),.cout(cout8));

    //实例化4位加法器
    adder_carry uu2(.a(a4),.b(b4),.sum(sum4),.cout(cout4));
endmodule
```

Define：`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。一旦`define 指令被编译，其在整个编译过程中都有效。例如，通过另一个文件中的`define 指令，定义的常量可以被其他文件中被调用。直到遇到`undef；parameter 只在定义的文件中有效，在其它文件中无效。

1.10.2 阻塞与非阻塞

非阻塞（Non_Blocking）赋值方式（如 b <= a;）

1) 块结束后才完成赋值操作；
2) 值并不是立刻就改变；
3) 这是一种比较常用的赋值方法。（特别在编写可综合模块时）

阻塞（Blocking）赋值方式（如： b = a;）

1) 赋值语句执行完后，块才结束；
2) 值在赋值语句执行完后立刻就改变；
3) 可能产生意想不到的结果。

非阻塞赋值方式和阻塞赋值方式的区别常给设计人员带来问题。问题主要是给

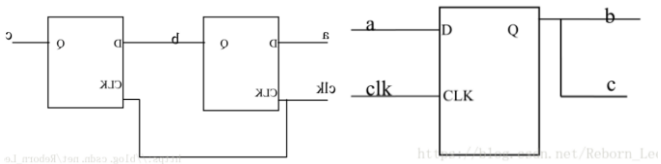
“always”块内的 reg 型信号的赋值方式不易把握。

非阻塞赋值

```
always @(posedge clk)
begin

    b <= a;
    c <= b;

end
```

阻塞赋值

```
always @(posedge clk)
begin

    b = a;
    c = b;

end
```



1.10.3 generate 使用

生成语句可以动态的生成 verilog 代码，当对矢量中的多个位进行重复操作时，或者当进行多个模块的实例引用的重复操作时，或者根据参数的定义来确定程序中是否应该包含某段 Verilog 代码的时候，使用生成语句能大大简化程序的编写过程。

生成语句生成的实例范围，关键字 generate-endgenerate 用来指定该范围。生成实例可以是以下的一个或多个类型：

（1）模块；（2）用户定义原语；（3）门级语句；（4）连续赋值语句；（5）initial 和 always 块。

generate 语句有 generate-for，generate-if，generate-case 三种语句。

```
module top_module(
    input [7:0] in,
    output [7:0] out
);
    genvar i;
    generate
        for(i = 0; i < 8; i = i + 1) begin : bit_reverse
            assign out[i] = in[7 - i];
        end

    endgenerate

endmodule
```

好了，大概怎么写应该知道了吧，那么需要注意什么呢?

(1) 必须有genvar关键字定义for语句的变量。

(2) for语句的内容必须加begin和end（即使就一句）。

(3) for语句必须有个名字。

1.10.4 逻辑符号与按位符号区别

注意，运算符| & ~ ^都是按位进行运算 ，运算符|| && ！都是整体进行运算，当运算对象为单 bit 时，无区别。

1.10.5 task 与 function 区别

1) 函数可以返回一个值而任务可以返回多个值
2) 函数一经调用必须立即执行，里面不能包含任何的时序控制，而 task 中可以有时序控制
3) 函数可以调用函数，但不可以调用任务，任务既可以调用函数也可以调用任务
4) 函数必须要有一个输入参数，而任务可以没有参数输入。
5) 任务输出的信号，在模块中必须定义为 reg 信号

1.10.6 逻辑移位与算术移位

>>>（算术右移）与>>（逻辑右移）的区别：

逻辑右移就是不考虑符号位，右移一位，左边补零即可。

算术右移需要考虑符号位，右移一位，若符号位为 1，就在左边补 1,；否则，就补 0。所以算术右移也可以进行有符号位的除法，右移,n 位就等于除 2 的 n 次方。

例如，8 位二进制数 11001101 分别右移一位。逻辑右移就是[0]1100110

算术右移就是[1]1100110

左移的话都是直接左移，右边补 0。

1.10.7 多位复用器 MUX 与逻辑门换用方法

如果一个标准单元库只有三个 cell：2 输入 mux(o = s ？ a ： b;)，TIEH(输出常数 1)，TIEL(输出常数 0)，如何实现以下功能？

1) 反相器 inv　assign out = in ？ TIEL ： TIEH;
2) 缓冲器 buffer　assign out = in ？ TIEH ： TIEL;
3) 两输入与门 and2　assign out = in1 ？ in2 ： TIEL;
4) 两输入或门 or2　assign out = in1 ？ TIEH ： in2;
5) 两输入异或 xor　assign out= a ? (b? TIEL : TIEH) : (b? TIEH: TIEL);
6) 四输入的 mux4　assign out = in2 ? (in1?d3:d2) : (in1?d1:d0);
7) 一位全加器 fa　**需要 9 个 MUX2**

一位全加器：

一位全加器的表达式如下：

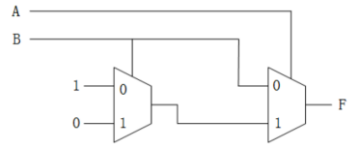
Si=Ai⊕Bi⊕Ci-1

$C_i = A_iB_i + C_{i-1}(A_i + B_i)$

第二个表达式也可用一个异或门来代替或门对其中两个输入信号进行求和：

$C_i = A_iB_i + C_{i-1}(A_i \oplus B_i)$

第一个式子：a^b : out1 = a ? (b? TIEL : TIEH) : (b? TIEH: TIEL);得到结果 out 与 C_{i-1} 异或即可。Out = out1 ? (C_{i-1}? TIEL : TIEH) : (C_{i-1}? TIEH: TIEL);

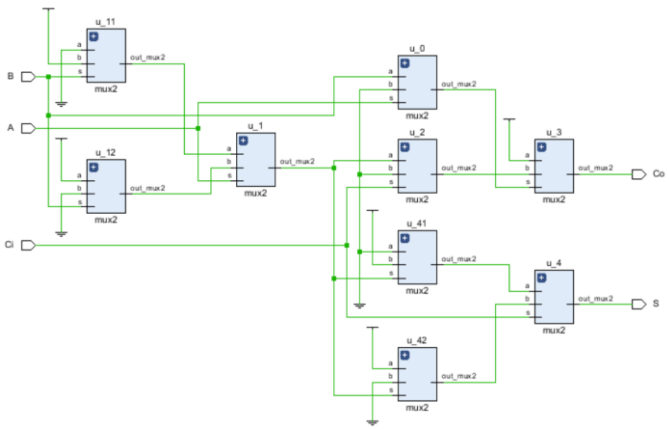


A^B

C_i 求法：

```
out1为Ai与Bi：

out1 = Ai ? Bi : TIEL;

out2为：Ai与Bi的异或：

out2 = Ai ? (Bi ? TIEL:TIEH) : (Bi ? TIEH:TIEL)

out3为Ci-1与out2 ：

out3 = Ci-1 ? out2 : TIEL;

out4 为out1或out3，也就是Ci：

out4 = out1 ? TIEH：out3

分别代入：

Ci = (Ai ? Bi : TIEL)? TIEH : (Ci-1 ? ( Ai ? (Bi ? TIEL:TIEH) : (Bi ? TIEH:TIEL)) : TIEL)
```

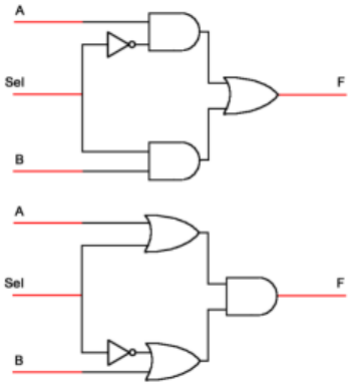
全加器综合的 RTL 电路



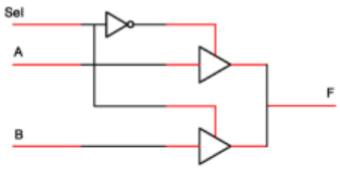
1.10.8 逻辑门换用 MUX2 多位复用器

多路复用器属于小规模集成组合逻辑单元，它的实现方式很多，以 MUX2IN1 为

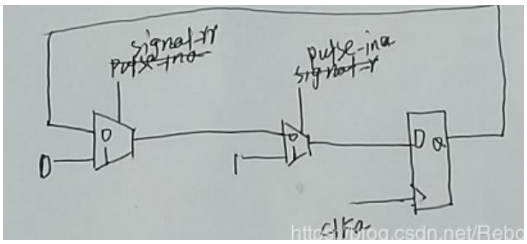
例，可以利用与、或、非门实现如下



也可以用三态门实现：

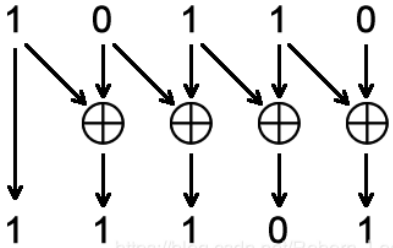



```
//在clka下，生成展宽信号signal_a
always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_a <= 1'b0;
    end
    else if(pulse_ina == 1'b1)begin
        signal_a <= 1'b1;
    end
    else if(signal_a_rr == 1'b1)
        signal_a <= 1'b0;
    else
        signal_a <= signal_a;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



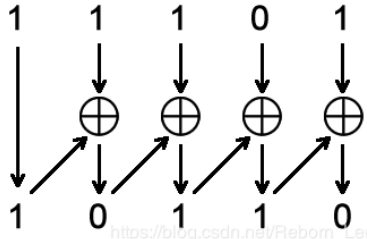
1.10.9 二进制与格雷码转换与独热码区别

二进制向格雷码转换图：



Verilog 对应语句：assign gray_value = binary_value ^ (binary_value>>1);

格雷码向二进制转换：



Verilog 对应语句：

```
module gray2bin #(
        parameter N = 4
)(
        input [N-1:0] gray,
        output [N-1:0] bin

    );

        assign bin[N-1] = gray[N-1];

        genvar i;
        generate
        for(i = N-2; i >= 0; i = i - 1) begin: gray_2_bin
                assign bin[i] = bin[i + 1] ^ gray[i];
        end
        endgenerate


endmodule
```

格雷码：相邻之间只变 1bit，编码密度高。

独热码：任何状态只有 1bit 为 1，其余皆为 0，编码密度低。**相当于已经译码后**

比如说，表示 4 个状态，那么状态机寄存器采用格雷码编码只需要 2bit：00(S0),01(S1),11(S2),10(S3)；

采用独热码需要 4bit：0001(S0),0010(S1),0100(S2),1000(S3)。所以很明显采用格雷码可以省 2bit 寄存器。

难理解的是，为什么独热码更节省组合逻辑：

例一：

假如我们要在代码中判断状态机是否处于某状态 S1，对于格雷码的状态机来说，代码是这样的：assign S1 = (STATUS==2'b01);对于独热码来说，代码是这样的就行：assign S1=STATUS[1];所以独热码的译码非常简单。

例二：

考虑最简单的跳变，当 A 为 1 时，状态机会从 S0 跳到 S1:。采用格雷码写：

STATUS[1:0] <= (STATUS==2'h00) & A ? 2'h01 : 2'h00;采用独热码写:

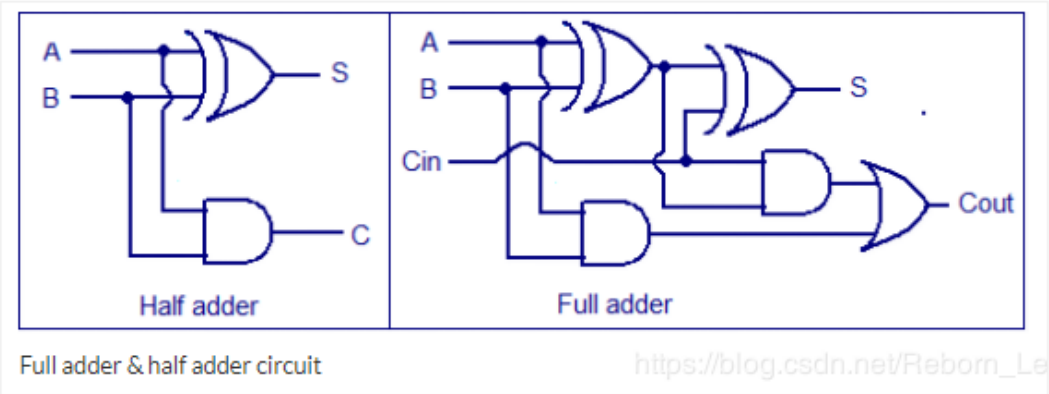
STATUS[1] <= STATUS[0] & A;

独热码适合写条件复杂但是状态少的状态机；

格雷码适合写条件不复杂但是状态多的状态机。

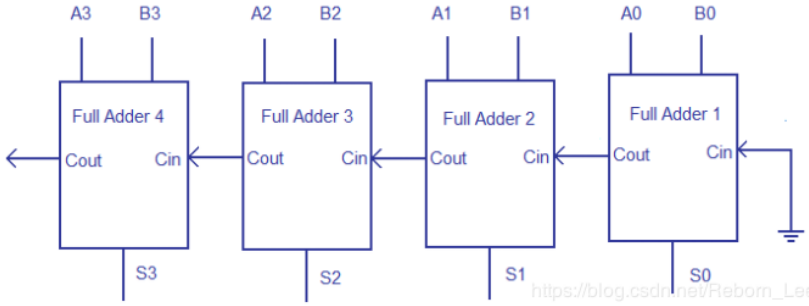
1.10.10 全加器与半加器和加法器组合

这是半加器与全加器的电路 组合图，具体知识可以查看上述描述。



Full adder & half adder circuit

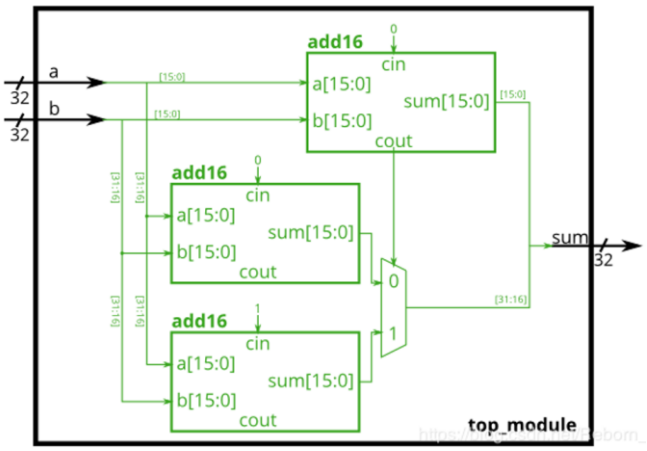
1) 等波纹四位加法器（1bit 全加器组成）RCA



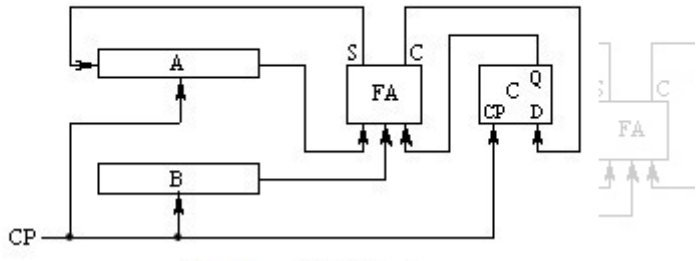
从图可以看出，每个全加器都是由组合逻辑组成，而且整个加法器串联，加法器越大，整个加法器的延迟就越大。

2) 进位选择加法器

如图所示，32 位的加法器，如果按照上述 1 中进行，需要 32 级的延迟，但是如果将 32 位拆分为高 16 和低 16 的话，并将高 16 复制一份，分别计算有进位和无进位两种情况，最后选择，这样高 16 位与低 16 位加法器并行运算，时间比 1 减少一半，资源消耗多了 16bit 加法器资源。这是典型的**以面积换取速度**。



3) 串行加法器（只用一位全加器）



如图所示，FA 为全加器，AB 分别为两个加数的一个移位器，FA 的和重新送入 A 中，进位触发器对有进位的进行触发。真值表如图所示。

串行加法器

A	B	Cin	S	Cout
1	**0**	**0**	**1**	**0**
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
1	1	0	0	1

4) 超前进位加法器 CLA

$$C_{i+1} = (A_i \cdot B_i) + (A_i \cdot C_i) + (B_i \cdot C_i)$$
$$= (A_i \cdot B_i) + (A_i + B_i) \cdot C_i$$

设：

- 生成（Generate）信号：$G_i = A_i \cdot B_i$
- 传播（Propagate）信号：$P_i = A_i + B_i$

则：$C_{i+1} = G_i + P_i \cdot C_i$

因此可以得到最高位扩展出来的计算等式：

$$C_1 = G_0 + P_0 \cdot C_0$$

$$\mathbf{C_2} = G_1 + P_1 \cdot C_1$$
$$= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$$
$$\mathbf{= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0}$$

$$\boxed{C_{i+1} = G_i + P_i \cdot C_i}$$

$$\mathbf{C_3} = G_2 + P_2 \cdot C_2$$
$$= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$$
$$\mathbf{= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0}$$

$$\mathbf{C_4} = G_3 + P_3 \cdot C_3$$
$$= G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0)$$
$$\mathbf{= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0}$$

由于计算 C1,C2,C3,C4 的等式值都是已知的，可以直接计算，并不需要等待。其门延迟大大低于等纹波加法器，不过逻辑门扩展的很复杂。

1.10.11 有符号与无符号数计算

如果定义 reg 与 wire 类型时，加上 signed 标注，这表示此数据类型为有符号 ，否则为无符号类型。定义了 signed 的数据进行运算时，是采用补码形式进行的，这一点要注意 ，但是如果此符号数为正数，其本身就是补码（数电知识点）。此处移位要用算术移位符>>>和<<<。其余加减乘还是一样运算，会自动将符号添加。工程：..\..\..\..\..\project_all\project\digit_module_function\signed

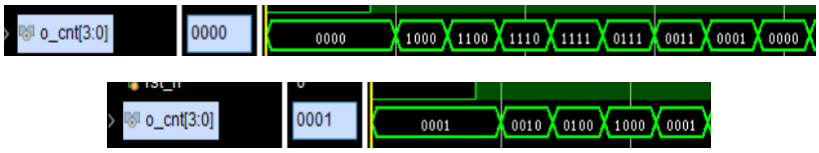
1.10.12 二进制、Johnson、环形计数器

假设都是 N 位的计数器，二进制计数器有 2^N 个状态，Johnson 计数器有 2N 个状态，而环形计数器有 N 个状态；相对于二进制计数器，Johnson 计数器相邻两组代码只可能有一位二进制代码不同，环形计数器也是如此。

```
end
else if(o_cnt == 4'b1111) begin
    o_cnt <= 4'b0000;
end
else begin
    o_cnt <= o_cnt + 4'b0001;
end
```

```
else if(!q[0])
    q <= {1'b1,q[N-1:1]};
else
    q <= {1'b0,q[N-1:1]};
```

```
end
else begin
        o_cnt <= {o_cnt[2:0],o_cnt[3]};
end
```



1.11 存储使用与仿真

存储单元与模块.docx

1.12 通信方式与仿真

通信方式与仿真.docx

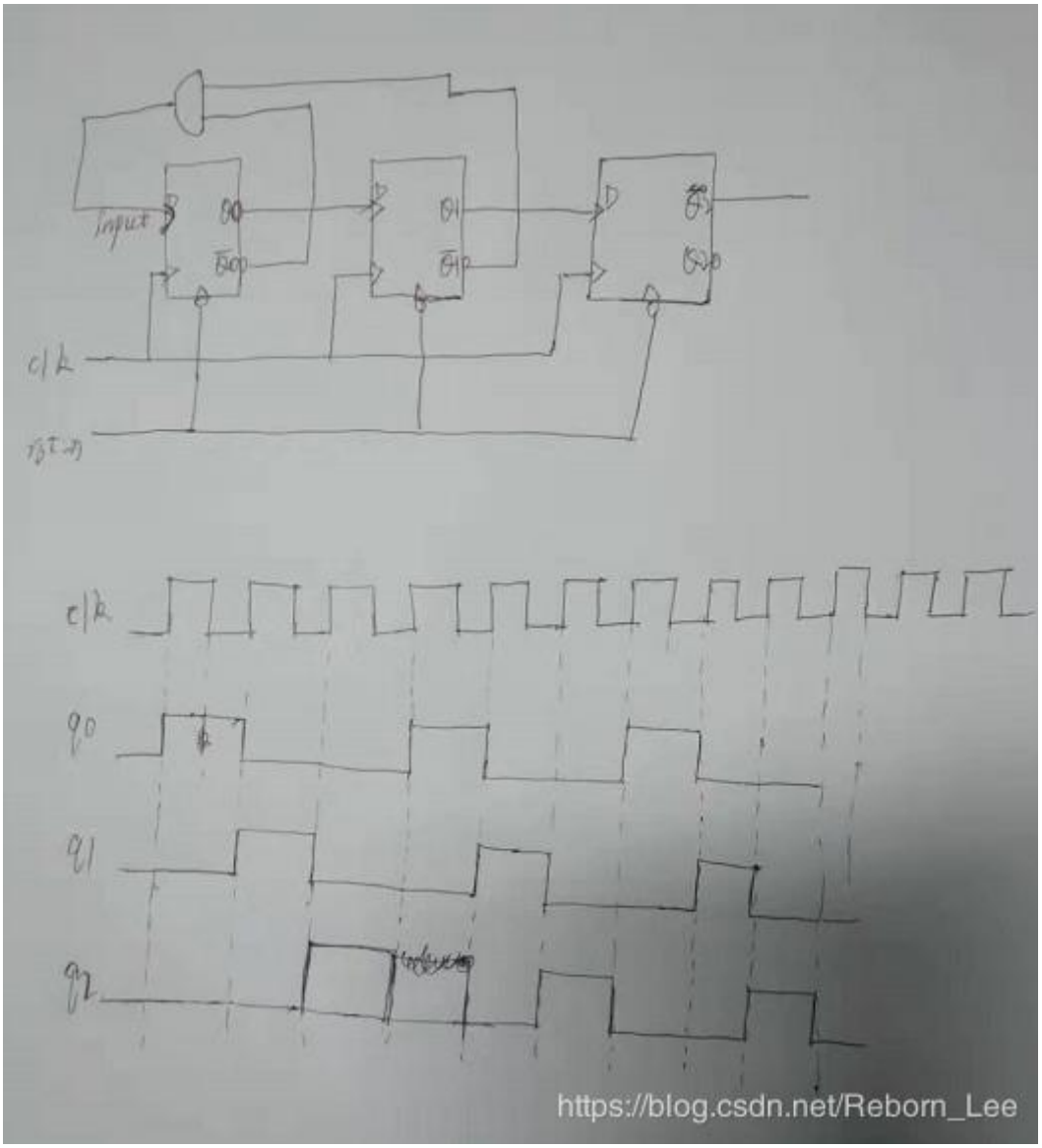
2. 笔面试题集分析

笔试题：http://www.mdy-edu.com/bishiti/

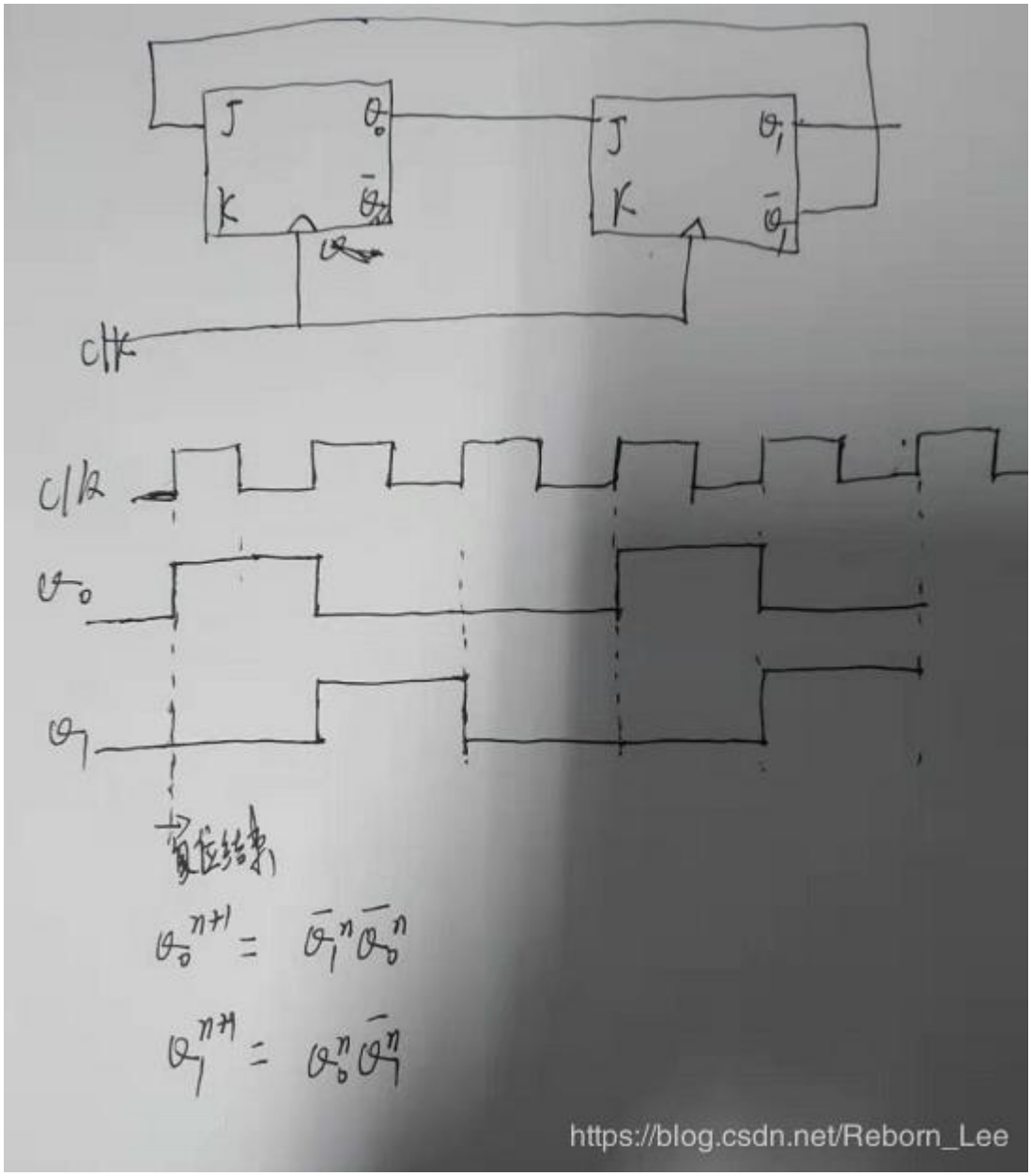
2.1 看电路画波形

2.2.1 无输入波形

切记：有时候波形图并不需要强制要有输入，只要复位有初始值就可以，比如此处 ，复位初始值 Q1,Q2,Q3 假设为 0 就可以有输出，输出功能为 3 分频。

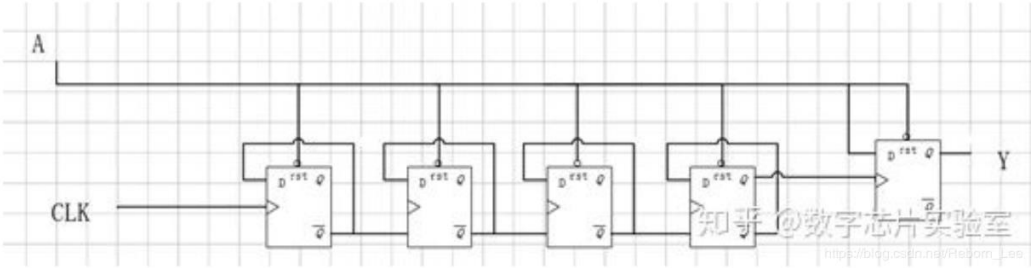


2.1.2 驱动方程



2.1.3 分频滤波电路

分析以下电路功能?



根据画图可以看 Q1,Q2,Q3,Q4 是分频电路，分别 2/4/8/16 分频，但 Y 作用是对

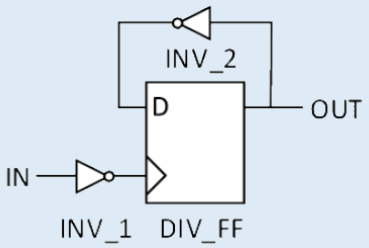
A 进行检测，当 A 的脉冲小于 7 个时钟周期时，当做毛刺处理，滤波作用。

2.2. 代码设计

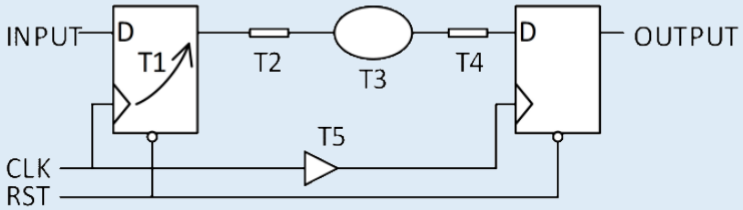
2.2.1 任意时钟分频 1-8 分频

2.2.2 最小周期计算 （每条路径都要计算，求解关键路径延迟即为最小周期）

1. 如下分频电路，触发器DIV_F的建立时间2ns，保持时间2ns，逻辑延时6ns，反相器INV_1，INV_2的逻辑延时为2ns，连线延时为0，该电路正常工作的最高频率？

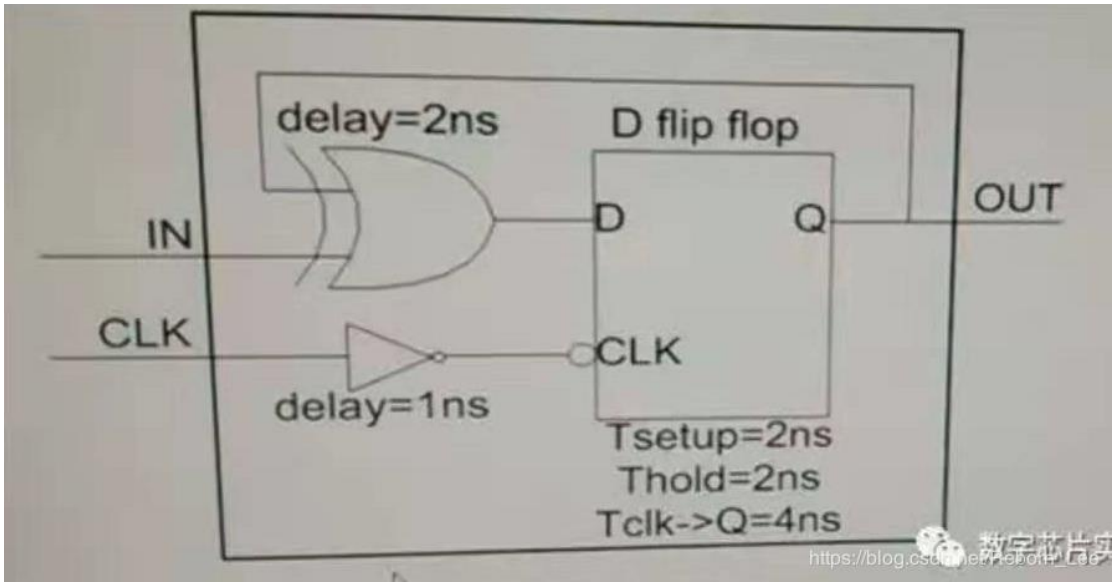


4. 在同步电路设计中，电路时序模型如下：T1为触发器时钟到输出延时，T2和T4为连线延时，T3为组合路径延时，T5为时钟网络延时，假设时钟周期为Tcycle，Tsetup，Thold分别为触发器建立保持时间，为保证？？正确采样（改路径为multi-cycle路径），下列必须满足的是：



① T1+T2+T3+T4<Tcycle-Tsetup+T5，T1+T2+T3+T4>Thold
② T1+T2+T3+T4<Tcycle-Tsetup，T1+T2+T3+T4+T5>Thold
③ T1+T2+T3+T4+T5<Tcycle-Tsetup，T1+T2+T3+T4>Thold
④ T1+T2+T3+T4<Tcycle-Tsetup+T5，T1+T2+T3+T4> Thold+T5

https://blog.csdn.net/Reborn_Lee/article/details/100049997



有效建立时间分析：

假设电路的有效Setup为Tsetup_valid：

对于D触发器而言，其本身的建立时间是2ns，也就是说数据必须在时钟有效沿到达之前2ns保持稳定，这样到达D端后就一定是稳定的数据了。

这个电路的数据来自于IN，时钟来自于CLK；

考虑时钟路径延迟影响：

时钟CLK要早于触发器的时钟1ns到达，因此对于D触发器建立时间的满足是有害的，电路有效建立时间

Tsetup_valid = Tsetup - 1ns = 1ns（因为数据需要提前1ns稳定下来）

考虑数据路径延迟影响：

Tsetup_valid = Tsetup - 1ns + 2ns = 3ns；（经过组合逻辑后的数据需要在时钟有效沿之前Tsetup时间稳定下来）

有效保持时间分析：

和建立时间分析套路一致，对于D触发器而言，数据需要在时钟有效沿到来之后保持Thold时间。

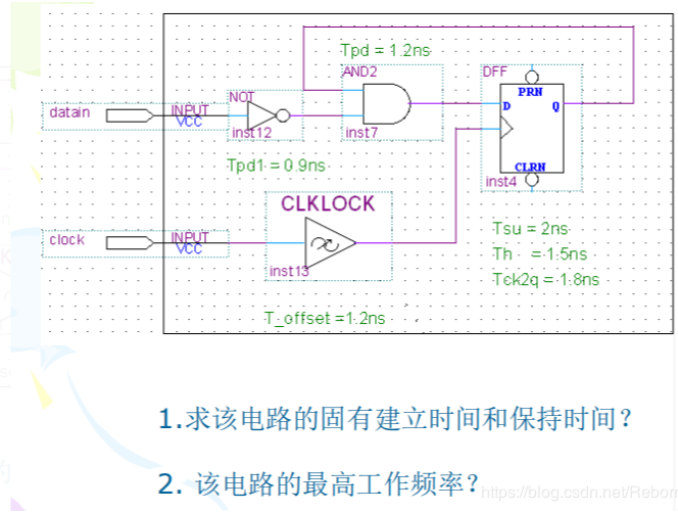
考虑时钟延迟的影响：

考虑到电路时钟对于触发器时钟早到1ns，所以电路有效保持时间Thold_valid = Thold + 1ns = 3ns;

考虑路径延迟影响：

数据需要经过一段组合逻辑之后才能保持稳定，因此电路的有效保持时间为：

Thold_valid = Thold + 1ns - 2ns = 1ns。



求电路的固有建立时间和保持时间和上一题的有效建立时间和保持时间是一个意思的。

因此固有建立时间为Tsu_valid = Tsu - T_offset + Tpd + Tpd1 = 2 - 1.2 + 1.2 + 0.9 = 2.9ns

固有保持时间为：Th_valid = Th + T_offset - Tpd - Tpd1 = 1.5 + 1.2 - 1.2 - 0.9 = 0.6ns.

而系统的最高频率呢？

先求系统的最小周期，考虑两个触发器之间的路径：

Tmin = Tco + Tpd + Tsu = 1.8 + 1.2 + 2 = 5ns,那么系统最高频率为200MHz。

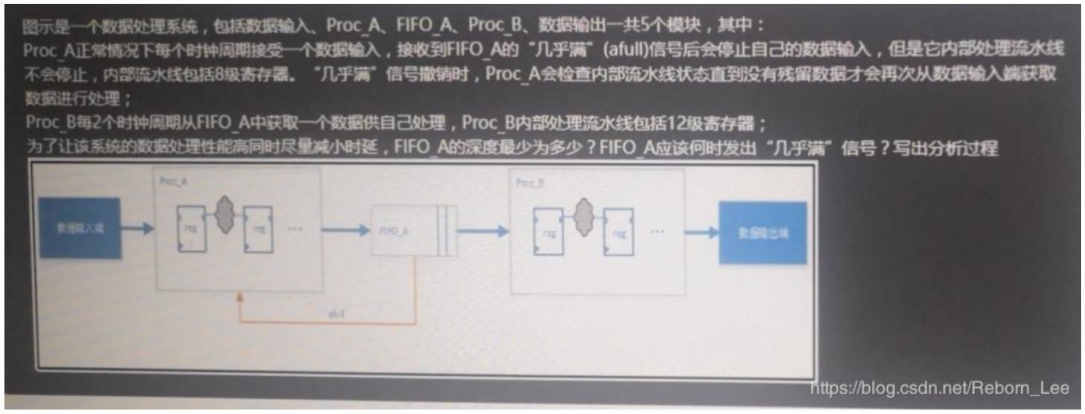
2.2.3 最小 FIFO 深度

https://blog.csdn.net/Reborn_Lee/article/details/100127937

算出读写时钟周期，计算突发写数据量，算出突发写数据时间，计算该时间内读数据量，两者相减即可。

Given the following FIFO and rules, how deep does the FIFO need to be to prevent underflow or overflow? ( )
RULES:
1) frequency(clk_A) = frequency(clk_B) / 4
2) period(en_B) = period(clk_A) * 100
3) duty_cycle(en_B) = 25%

- 25 entries
- 50 entries
- 75 entries
- 100 entries



看描述，对于Proc_A，接收到FIFO_A的几乎满信号后，会停止信号输入，但是内部有8级流水线，需要全部流入到FIFO后才能停止写入FIFO_A，由于发出几乎满信号，一定是FIFO内部有数据了（最小为1个数据，就产生afull信号），所以afull之后的突发写数据个数为8，需要8个时钟.而8个时钟bproc已经读出4个数据了，所以fifo的最小深度应该为8-4+1 = 5；

为什么会加1呢？按理说一般计算的套路应该为8-4 = 4就够了。但是我们还要考虑我们考虑的仅仅是afull之后的fifo最小深度，afull之所以为1，是因为fifo_A内部有数据了，而数据最小为1产生了afull信号，所以需要加1构成最终的fifo最小深度。

最后给出讨论群里的其他大佬的答案（第一个给出答案的）：

fifo深度应该是1+4为5，前面一个时钟写一个，两个时钟读一个，也就是两个时钟fifo里面会有一个数写进去，实际上a端已经写了两个数了，只不过被读出去了一个，为什么要再加一呢，afull是为了让fifo可以写进去数据，所以afull为1

2.2.4 至简设计

1) 状态机编写，序列检测，售货机
2) 计数器编写，时钟
3) 同步 FIFO 设计

2.3 功耗

低功耗设计.docx

2.3.1 降低峰值功耗

选择 1，峰值功耗是短路功耗

功耗分为动态功耗（开关功耗、短路功耗）和静态功耗

动态功耗：主要由于信号翻转导致功耗：降低办法有：降低电压、减低时钟频率、减少翻转率（门控时钟等）。

短路功耗：因此一般提高 HVT 比例

短路功耗（P_{short}）：也称为直通功耗，由于输入电压波形并不是理想的阶跃输入信号，而是以正弦波的形式。输入波形在上升与下降转换的短暂时间过程中，某个电压输入范围内，NMOS和PMOS都导通，这时就会出现电源到地的直流导通电流，即引起开关过程中的短路功耗。计算公式为：

$P_{short}=\tau A_{short} V_{dd}=\tau A\beta(V_{dd}-V_{th})^3$，其中，$I_{short}$表示短路电流，$\tau$表示电平信号从开始上升或开始下降，直到稳定所需时间，β是工艺参数，V_{dd}表示供电的电源电压，V_{th}表示器件阈值电压，因此减少开关功耗可从降低器件阈值电压、改善电路工

静态功耗：主要是漏电流造成，通过高阈值电压元件降低，多阈值处理等

下列降低功耗的措施，哪个可以降低峰值功耗

① 大幅度提高HVT比例

② 静态模块级clock gating

③ power gating

④ Memory shut down

2.4 数电题

2.4.1 CMOS 门电路

https://blog.csdn.net/Reborn_Lee/article/details/100659089

2.4.2 定点小数与浮点小数设计

1101.101　= 13.625；

FPGA 中定点与浮点设计：

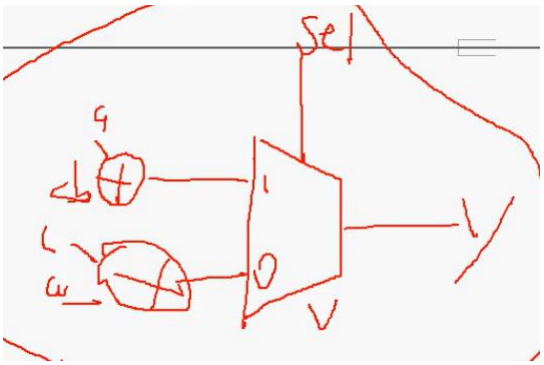
2.5 设计优化简化

2.5.1 电路简化

化简代码使硬件尽可能少[Trident]

```
always@ (sel or a or b or c)
    if(sel)
        y = a + b;
    else
        y = a + c;
```

要有电路的思维，它会综合出什么电路呢，如图所示：耗费两个加法器一个选择器



其实，由于两个加法器都有 a，我们可以通过选出是 b,还是 c，再来加 a。如图所示。注意，并不是代码越少越好，而是硬件越简单越好。

```
always  @(*)begin
    if(sel)
        x = b;
    else
        x = c;
end

always  @(*)begin
    y = a  + x;
end
```

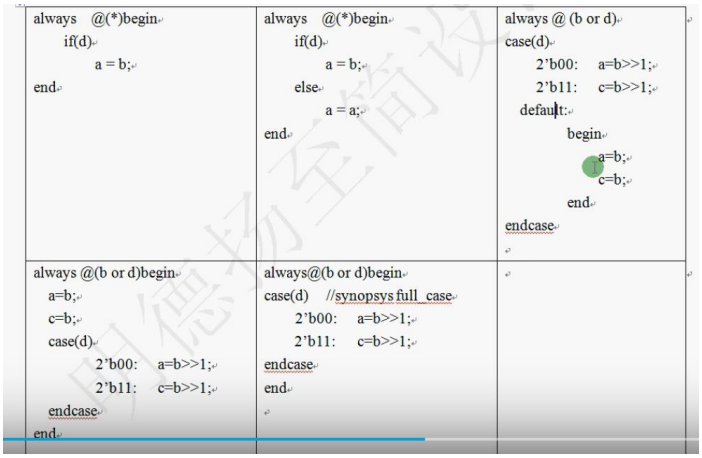
2.5.2 产生锁存器 latch 的代码

组合逻辑设计中 if else 或者 case 语句没有写完整，由于组合逻辑不能回环，因此综合后会导致产生锁存器。**上述这句话不对，并不是 if else 没写全就不会，而是有没有锁存这个动作，比如你的 else 里面是 a=a 这个语句，仍然会锁存。**

如图所示，下图前面两个会生成锁存器，后面两个不会。

原因：第三个是因为当状态为 00 时，a 是不会生成锁存器，但是 c 是保持的，所以会。第四个因为每个状态都会重新赋值，所以不会（不建议如此编写，综合器

不同可能不同），第五个有一句//synopsysfull case 这不是注释，是综合命令语句，会帮助综合，因此不会有锁存器。



```
always   @(*)begin
    if(d)
        a = b;
end
```

```
always   @(*)begin
    if(d)
        a = b;
    else
        a = a;
end
```

```
always @ (b or d)
case(d)
    2'b00:   a=b>>1;
    2'b11:   c=b>>1;
  default:
        begin
            a=b;
            c=b;
        end
endcase
```

```
always @(b or d)begin
  a=b;
  c=b;
  case(d)
        2'b00:   a=b>>1;
        2'b11:   c=b>>1;
  endcase
end
```

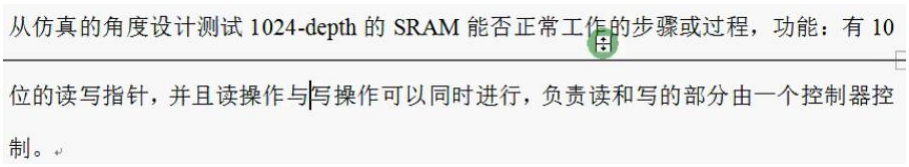
```
always@(b or d)begin
case(d)   //synopsys full_case
    2'b00:   a=b>>1;
    2'b11:   c=b>>1;
endcase
end
```

2.6 仿真测试

2.6.1 测试激励的完备性

仿真测试 32bit*32bit 乘法器的测试代码应该如何设计呢？

我们要考虑测试激励的完备性，我们要考虑边界以及极端的情况，比如 0 或者最大值，这要在设计之前就要列出来，一一测试。这其实与验证有一定连通，以前的定向测试就是如此，不过现在基本都是用 UVM 进行随机测试。



从仿真的角度设计测试 1024-depth 的 SRAM 能否正常工作的步骤或过程，功能：有 10 位的读写指针，并且读操作与写操作可以同时进行，负责读和写的部分由一个控制器控制。

如图，要测试 SDRAM 的功能，有深度 1024，地址指针 10 位，可同时读写三个功能；考虑地址为 0 和 1023 的时候，考虑读写同时与不同时，考虑读写同一地址与不同地址；前面两两组合进行测试，比如同时读 1023，写 0 的情况，同时读 1023 和写 1023 的情况等，看看能否工作，一一检测之后才能算完备。

2.7 综合

2.7.1 时序设计报文替换

https://www.bilibili.com/video/av64287046?p=13 第 13 讲，考点为时序设计以及计数器设计，ram 设计，跨时钟设计等。

如图所示题目，这道题主要是要注意 ram 读数据有一个时钟的延迟，这里需要进行时序对齐，因此可以用一个计数器对报文字节计数，产生相应的读使能信号，

对 ram 进行操作，读出相应的 ID，并对齐 data_i 数据替换输出。最后 RAM 读写时钟不一样，因此有跨时钟的问题，加入异步 fifo 即可。