硬十
硬件十万个为什么

硬十用户专享资料

之 逻辑技术

www.hw100k.com

前言——可编程逻辑

- 简介
- 可编程逻辑器件 英文全称为：programmable logic device 即 PLD。
- PLD是做为一种通用集成电路产生的，他的逻辑功能按照用户对器件编程来确定。一般的PLD的集成度很高，足以满足设计一般的数字系统的需要。这样就可以由设计人员自行编程而把一个数字系统“集成”在一片PLD上，而不必去请芯片制造厂商设计和制作专用的集成电路芯片了。
- 特点
- PLD与一般数字芯片不同的是：PLD内部的数字电路可以在出厂后才规划决定，有些类型的PLD也允许在规划决定后再次进行变更、改变，而一般数字芯片在出厂前就已经决定其内部电路，无法在出厂后再次改变，事实上一般的模拟芯片也都一样，都是在出厂后就无法再对其内部电路进行调修。

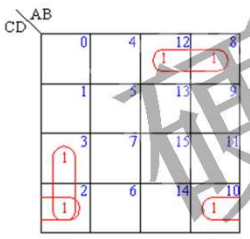
目录

- 章节简介
- 本章主要针对新入门的学习者，提供一些基础，和一些感性认识。讲解一些逻辑的基本知识。如果你已经具备一些CPLD、FPGA的基础知识，并会一些基本操作。可以跳过本章。

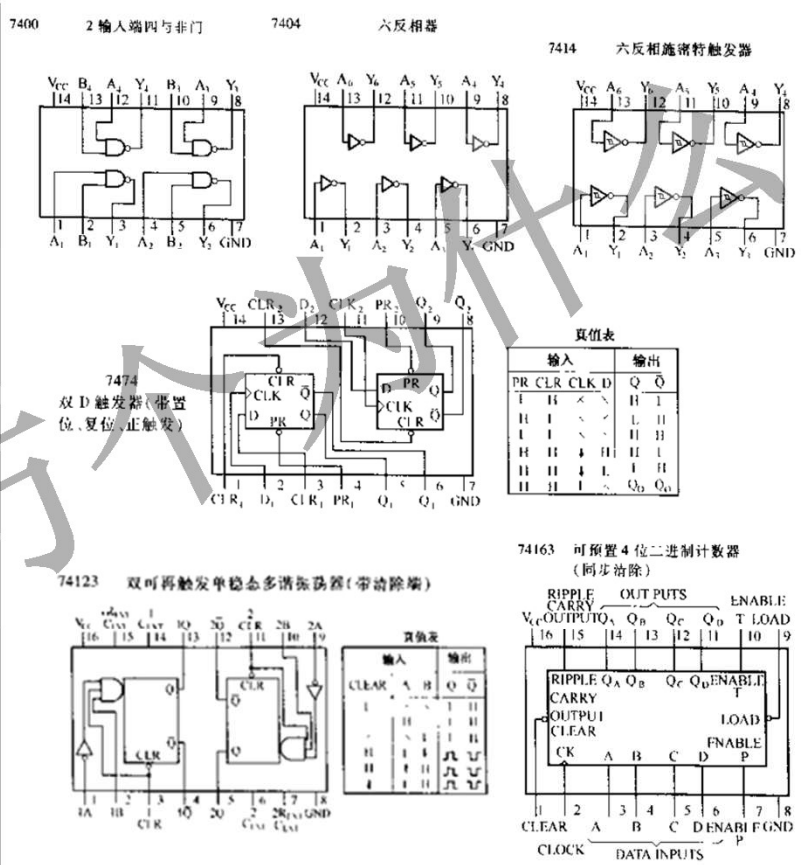
逻辑基础速成——传统数字电路设计

表 18-27　　数字集成电路系列

系列	子系列	名　称	国家标准型号
TTL	TTL	标准 TTL 系列	CT54/74- - -
	HTTL	高速 TTL 系列	CT54/74H- - -
	LTTL	低功耗 TTL 系列	CT54/74L- - -
	STTL	肖基特 TTL 系列	CT54S/74S- - -
	LSTTL	低功耗肖基特 TTL 系列	CT54LS/74LS- - -
	ALSTTL	先进低功耗肖基特 TTL 系列	CT54ALS/74ALS- - -
	ASTTL	先进肖基特 TTL 系列	CT54AS/74AS- - -
	FTTL	快速 TTL 系列	CT54F/74F- - -
MOS	PMOS	P 沟道场效应管系列	—
	NMOS	N 沟道场效应管系列	—
	CMOS	互补场效应管系列	CC4- - -（CC14- - -）
	HCMOS	高速 CMOS 系列	CC54HC/74HC- - -
	HCT	与 TTL 电平兼容的 HCMOS 系列	CC54HCT/74HCT- - -
	AC	先进的 CMOS 系列	—
	ACT	与 TTL 电平兼容的 AC 系列	—

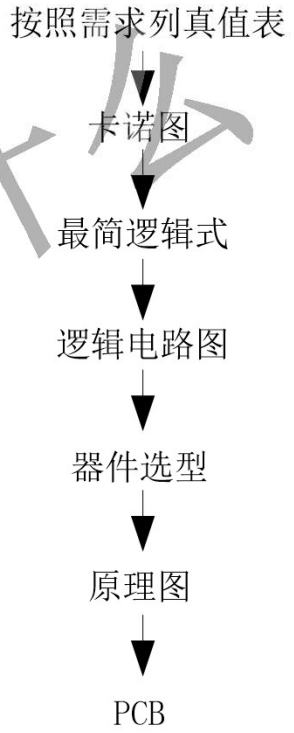


卡诺图是逻辑函数的一种图形表示。一个逻辑函数的卡诺图就是将此函数的最小项表达式中的各最小项相应地填入一个方格图内，此方格图称为卡诺图。卡诺图的构造特点使卡诺图具有一个重要性质：可以从图形上直观地找出相邻最小项。两个相邻最小项可以合并为一个与项并消去一个变量。



逻辑基础速成——传统数字电路设计

- 组合逻辑电路的一般设计步骤（四步法）
- 1 根据实际逻辑问题确定输入、输出变量，并定义逻辑状态的含义；
- 2 根据输入、输出的因果关系，列出真值表；
- 3 由真值表填写卡诺图。
- 4 根据卡诺图简化,逻辑表达式 根据需要简化和变换逻辑表达式。
- 5 画出逻辑图，原则：电路要最简（要求所用器件的种类和数量都尽可能少，且器件之间的连线也最少）
- 6 器件选型
- 7 原理图
- 8 PCB



逻辑基础速成——传统数字电路设计

硬十
硬件十万个为什么

1 逻辑抽象：用变量A、B、C、D表示输入，用L表示输出

2

输入				出	输入				出
A	B	C	D	L	A	B	C	D	L
0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	1	1
0	0	1	0	0	1	0	1	0	1
0	0	1	1	0	1	0	1	1	1
0	1	0	0	0	1	1	0	0	1
0	1	0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1

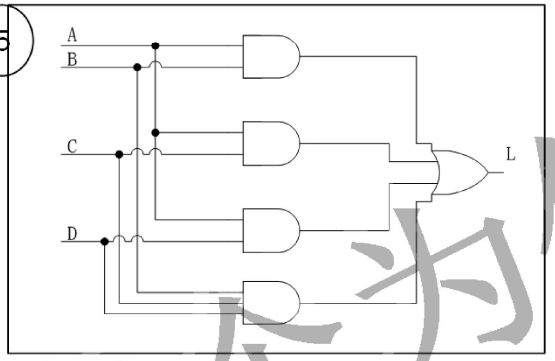
3

L = AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	1	1	1
10	0	1	1	1

BCD
AB
AC

4 $L = AB + AC + AD + BCD$

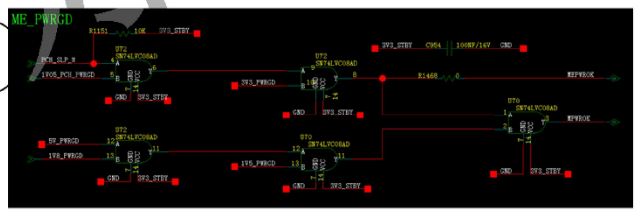
5



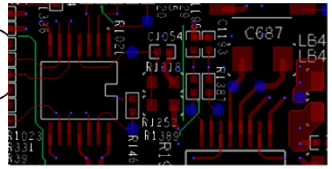
6

SN74HC08

7



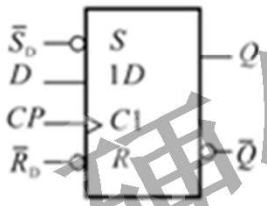
8



逻辑基础速成——D触发器

- D触发器是逻辑基础的非常重要的内容，是逻辑基础的基础。如果上学时没有理解透彻，建议重点学习。

- 定义：
- 电平触发的主从触发器工作时，必须在正跳沿前加入输入信号。如果在CP 高电平期间输入端出现干扰信号，那么就有可能使触发器的状态出错。而边沿触发器允许在CP 触发沿来到前一瞬间加入输入信号。这样，输入端受干扰的时间大大缩短，受干扰的可能性就降低了。边沿D触发器也称为维持-阻塞边沿D触发器。
- 功能描述：
- 功能框图　　逻辑功能表　　逻辑工作时序图

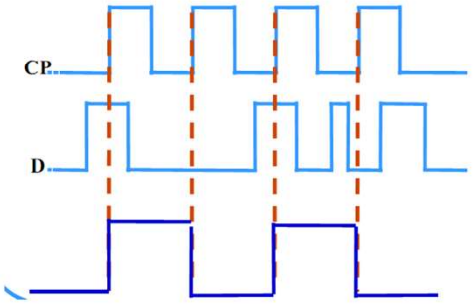


S即为右图中的/PRE

FUNCTION TABLE

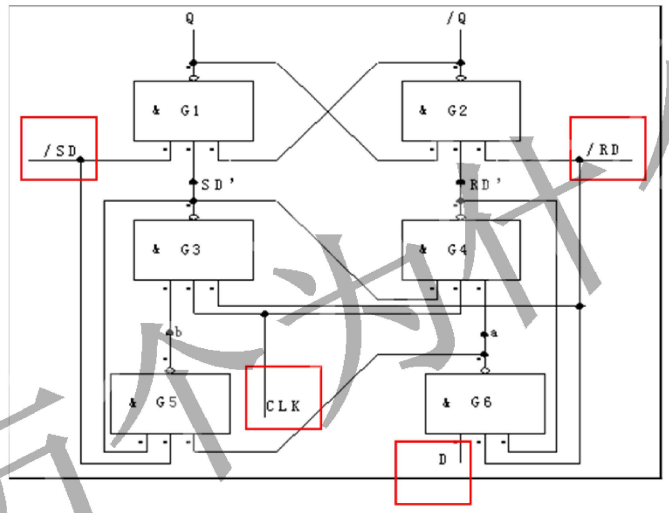
INPUTS				OUTPUTS	
PRE	CLR	CLK	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H†	H†
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

† This configuration is nonstable; that is, it does not persist when $\overline{PRE}$ or $\overline{CLR}$ returns to its inactive (high) level.



逻辑基础速成——D触发器

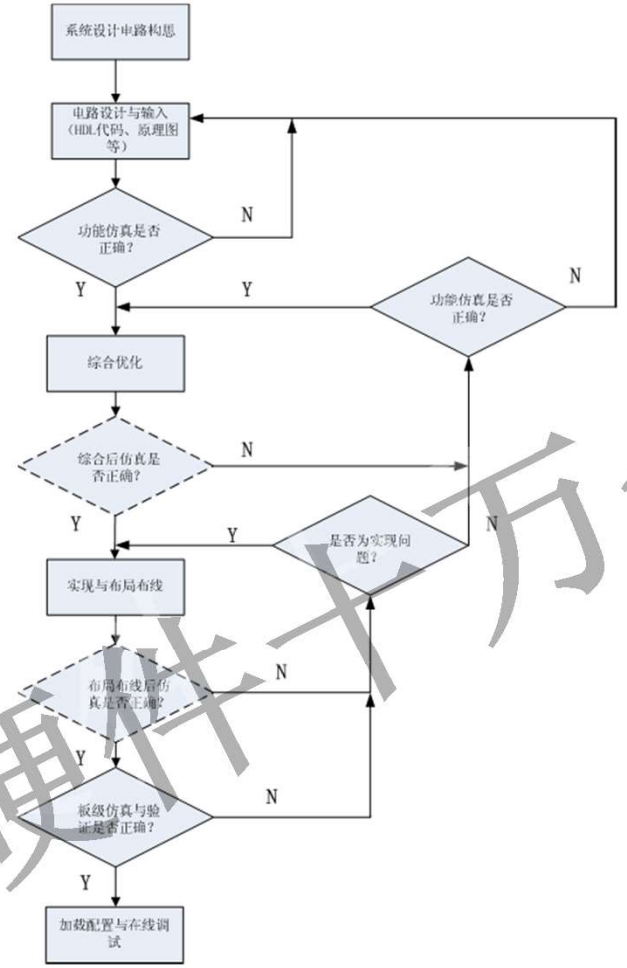
- D触发器工作原理
- 1、右图为维持阻塞结构的DFF，电路结构如右图所示。由6个与非门组成G1~G6。
- D触发器在CP为低电平的时候将输出与输入隔离，输出端不会因为输入端的变化而变化；
- 在CP为高电平的时候，D触发器依靠内部信号对输入端的反馈，保证输出端数据的稳定；
- 输出端的变化仅仅依赖于CP触发沿的一瞬间输入的数据。
- 2、/SD 和 /RD是直接置“1”和直接置“0”端；也称为异步置“1”和异步置“0”端。



D触发器为什么需要建立保持时间？

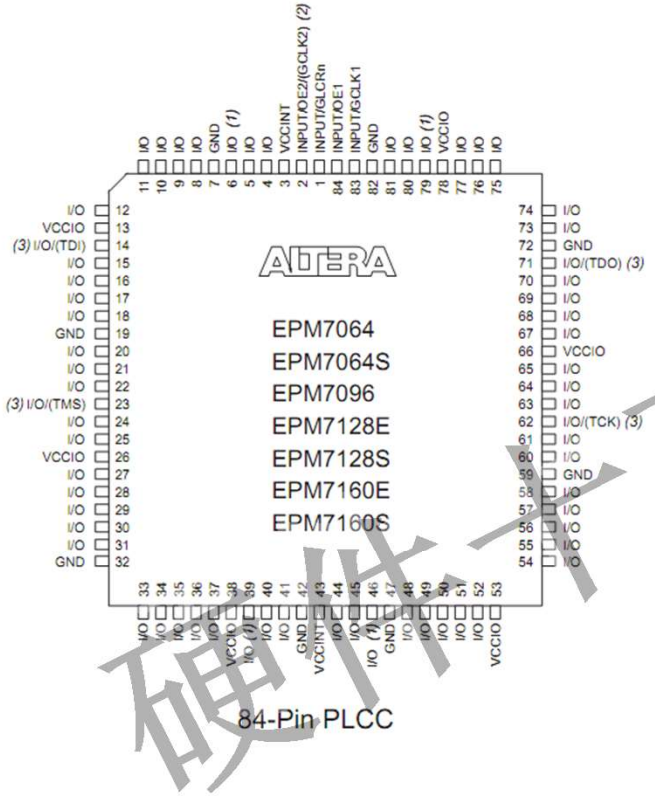
亚稳态论文.doc

逻辑基础速成——实现电路的基本过程



- 章节简介
- 本章介绍CPLD、FPGA的基本原理；可编程逻辑发展历程；主流器件厂家器件汇总；CPLD、FPGA的加载原理；FPGA演进分析。

逻辑器件原理——CPLD



电源信号

GND信号

通用I/O信号

JTAG加载信号

GCLK信号（全局时钟信号）

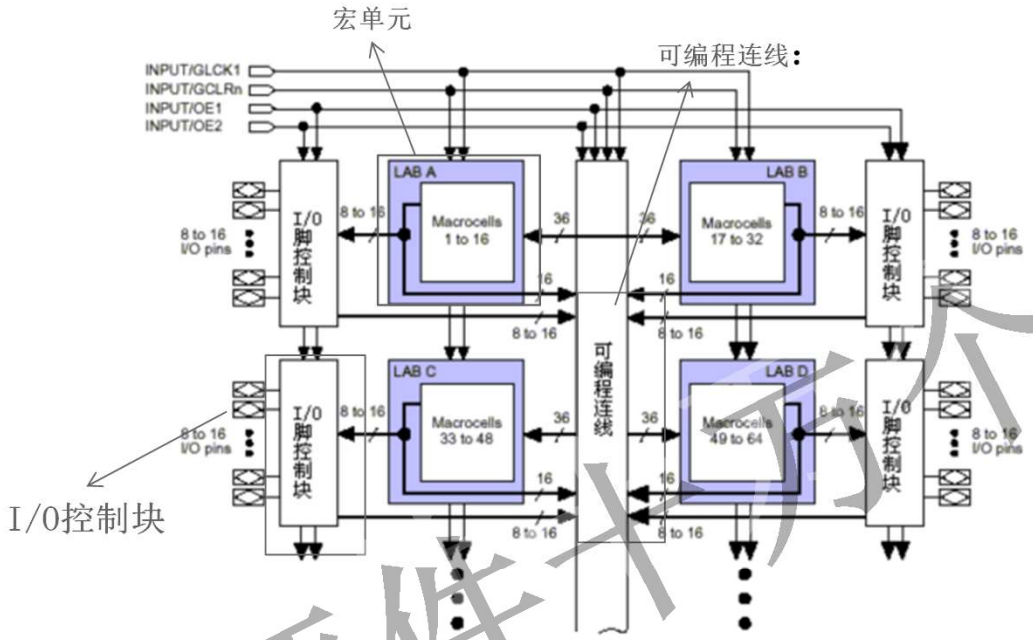
GLCR信号（全局复位信号）

OE信号（全局使能信号）

逻辑器件原理——CPLD结构

- CPLD结构一般有以下两种：
- 1、基于乘积项结构：基本结构为“与-或”阵列的器件，基于EEPROM或FLASH工艺，老的CPLD一般基于这种结构。这类结构的实现原理会在本章中详细介绍。
- 2、基于LUT查找表结构：基于SRAM工艺，CPLD上电时，由片内的FLASH对SRAM进行进行配置，新工艺的CPLD基于这种结构。这类结构的实现原理和FPGA一致，在FPGA结构章节中会详细说明。

逻辑器件原理——CPLD内部结构



基于乘积项结构的CPLD内部结构

CPLD主要是由可编程逻辑宏单元(MC，Macro Cell)围绕中心的可编程互连矩阵单元组成。其中MC结构较复杂，并具有复杂的I/O单元互连结构，可由用户根据需要生成特定的电路结构，完成一定的功能。

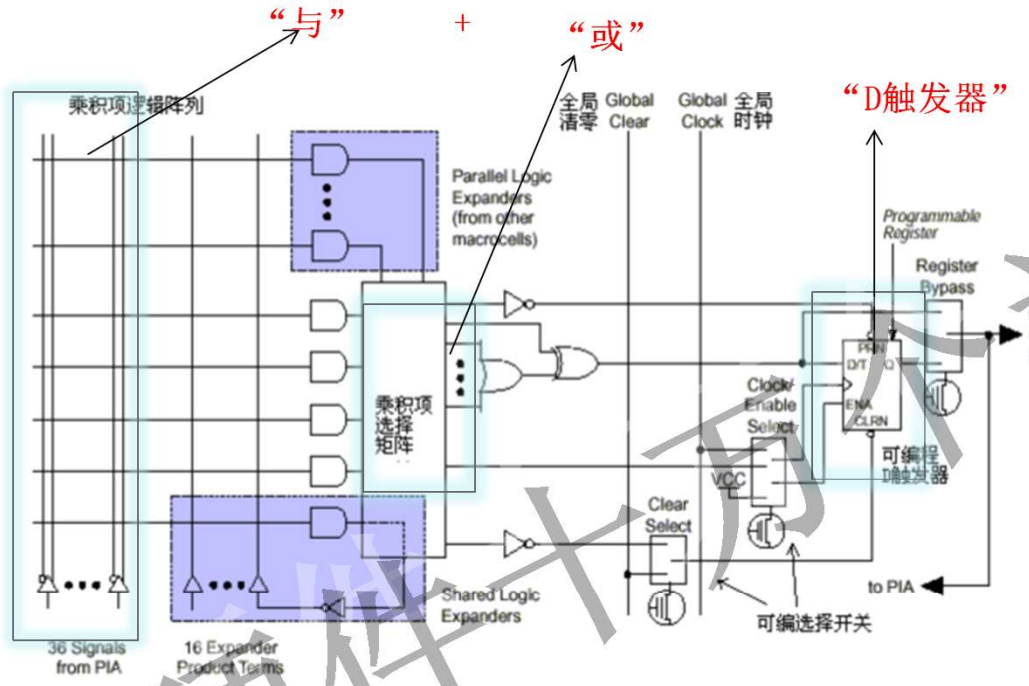
这种CPLD可分为三块结构：宏单元（Marocell），可编程连线（PIA)和I/O控制块。

宏单元：实现基本的逻辑功能。

可编程连线：负责信号传递，连接所有的宏单元。

I/O控制块负：责输入输出的电气特性控制，比如可以设定集电极开路输出，摆率控制，三态输出等。

逻辑器件原理——CPLD乘积项结构



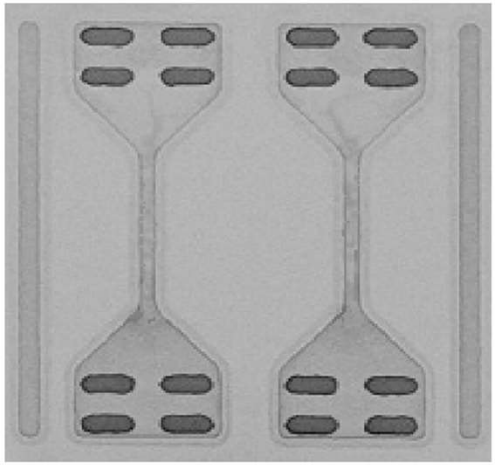
宏单元结构

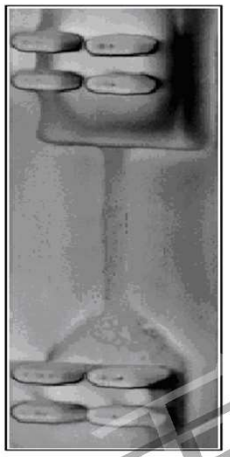
*任意一个组合逻辑都可以用“与-或”表达式来描述。

左侧为乘积项阵列结构，实际就是一个与-或阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。后面的乘积项选择矩阵是一个“或”阵列。两者一起完成组合逻辑。

图右侧是一个可编程D触发器，它的时钟，清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项阵列）产生的时钟和清零。

逻辑器件原理——CPLD乘积项结构



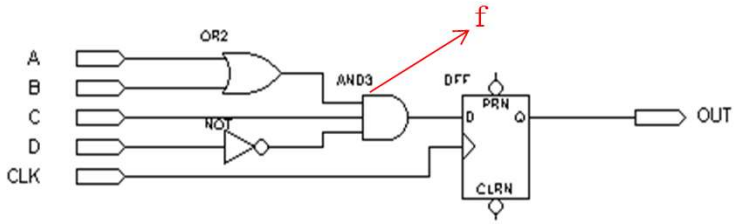


电子可编程熔丝efuse （electrically programmable fuse），通常又被称为多晶硅
熔丝，它是位于两个电极之间很短的一段最小宽度的多晶硅。

我们在两电极之间加以较高电流，这样在较高的电流密度的作用下，相关原子将会沿着电子运动方向进行迁移，形成空洞，最终断路，这种现象就是电迁移(EM)。它是由极高电流密度引起的慢性损耗现象，即移动载流子对静止金属原子的影响引起金属的逐渐移位元（displacement）。由于单个晶体（或晶粒）通常相互临接，电迁移引起金属原子逐渐移出晶粒间界，在相邻晶粒间形成空隙。这是联机的有效横介面积减小，引起联机的剩余部分的电流密度增大。新空隙形成并逐渐结合，最终切断联机。

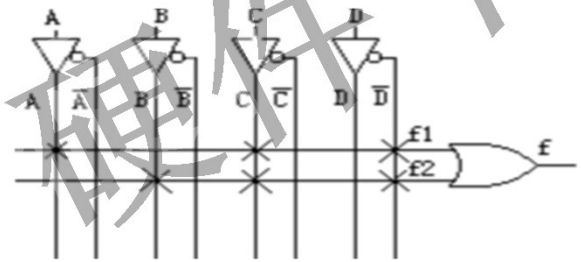
逻辑器件原理——CPLD乘积项结构功能实现举例

上面已经说过，任何一个组合逻辑都可以用“与-或”表达式来描述。
下面我们以一个简单的电路为例,具体说明CPLD是如何利用以上结构实现逻辑的，电路如下图：



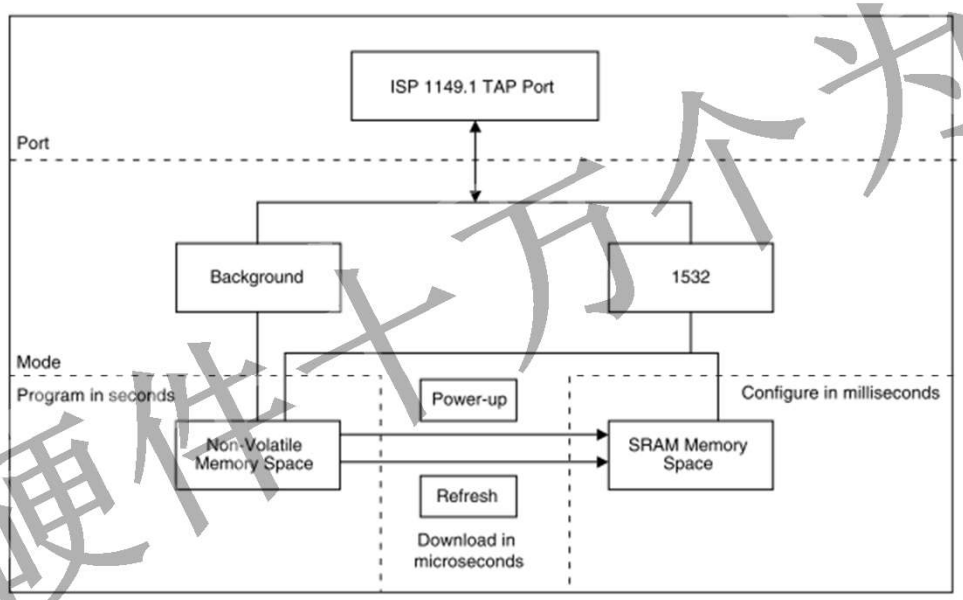
假设组合逻辑的输出(AND3的输出)为f，则f=(A+B)*C*(!D)=A*C*!D + B*C*!D（!D表示D的“非”）

CPLD将以下面的方式来实现组合逻辑f:



逻辑器件原理——CPLD加载原理

- CPLD一般用JTAG接口进行加载，内部有FLASH和SRAM，CPLD的配置文件可存在在内置的FLASH中，因此下电不会丢失，不需要每次上电的时候，额外对CPLD进行配置结构如下：



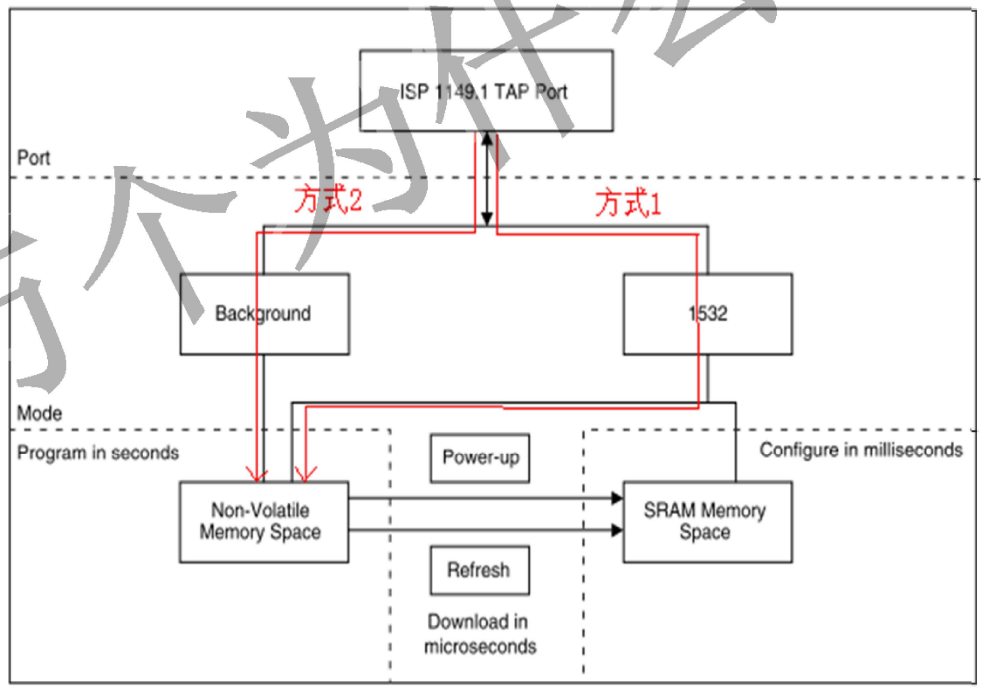
逻辑器件原理——CPLD加载原理

- on-chip Flash配置方式:
- CPLD内部的Flash 可以使用IEEE 1532通过IEEE 1149.1 ispJTAG port进行加载，加载方式有2种：

方式一：当SRAM为空时（CPLD一次都未加载过或者CPLD内部FLASH存储的配置文件有问题，不能加载到SRAM中），Flash 编程进入直接模式（如下方式一）此时CPLD的IO管脚状态由BSCAN registers（边界扫描寄存器）决定， BSCAN registers可以将IO设置成 high, low, tristate (default), or current value四种。**Mtca V1 SCU的CPLD将其中两个管脚设置成高输出。**

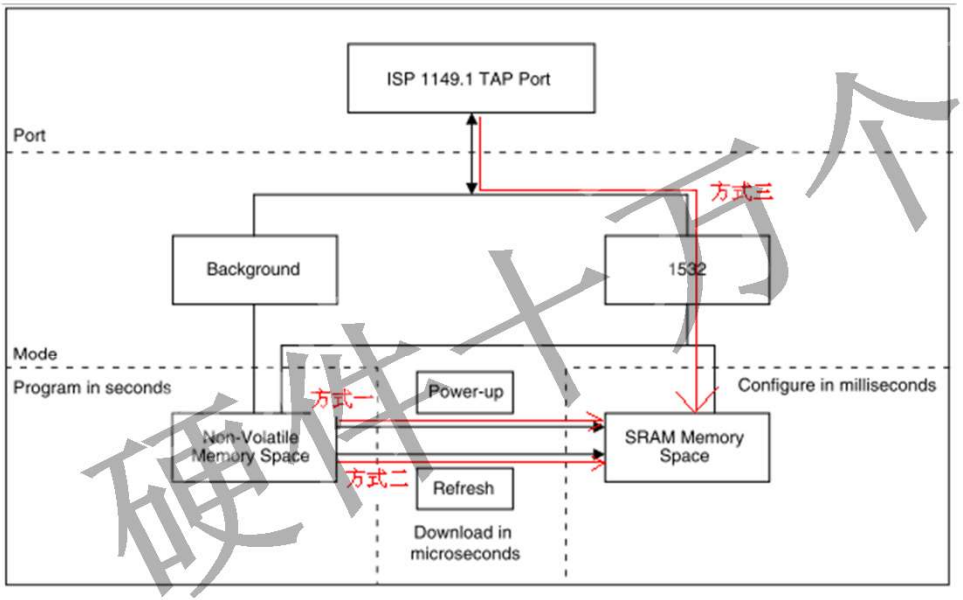
mTCA
MSCUA加载过程备用板异常告警.

方式二：当SRAM不为空的时候，Flash 可进行background编程 模式. 在此模式下，在加载on-chip Flash时，允许CPLD器件仍然维持在用户操作模式下（即CPLD可以正常工作）。



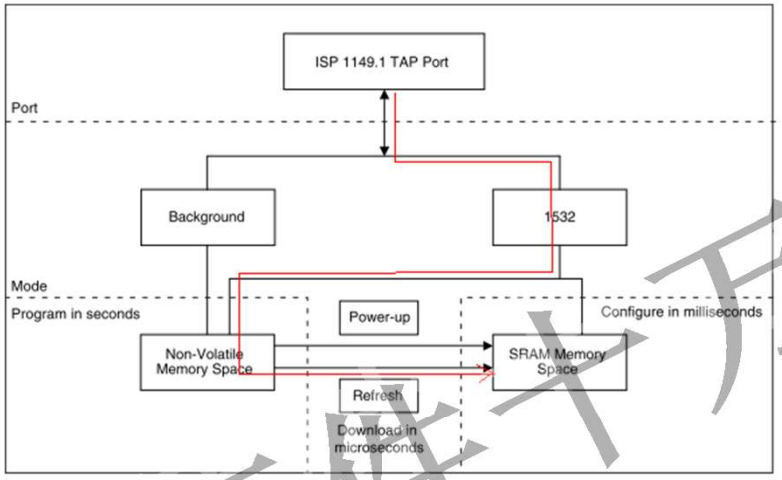
逻辑器件原理——CPLD加载原理

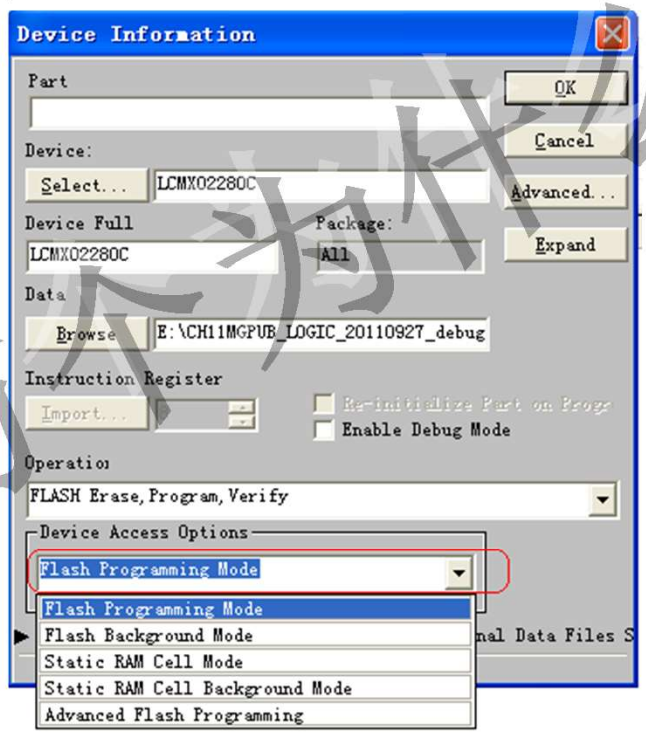
- SRAM 可以由如下二种方式进行配置:
- 一、通过FLASH进行配置：上电通过FLASH进行配置（下图方式一）或者通过IEEE 1149.1 port发送refresh命令（下图方式而）
- 二 、通过IEEE 1149.1 port 处于In IEEE 1532 mode 进行加载（下图方式三）



逻辑器件原理——CPLD加载原理

- 我们目前最常用的在线升级CPLD方式（路径）





CPLD加载方式选择

➢ IEEE 1532标准简介

➢IEEE 1532标准是一个基于IEEE 1149.1的在板编程的新标准，标准的名字为IEEE Standard for In-System Configuration of Programmable Devices。

➢ 在1993年，出现ISP（In-System Programming）的概念和应用。随之产生了应用IEEE1149.1进行ISP的需求。各个厂商提供了类似的不相同的基于JTAG的ISP工具。

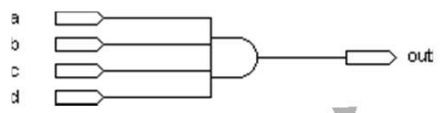
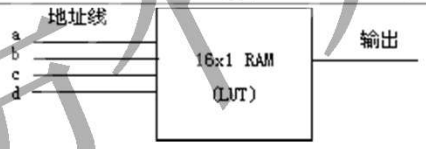
➢ 1996年4月，半导体厂商、ISP工具开发者、ATE开发商正式提出了IEEE 1532标准，旨在为JTAG器件的在板编程提供一系列标准的专门的寄存器和操作指令从而使得在板编程更为容易和高效。

➢ IEEE1532完全建立在IEEE1149.1标准之上，在IEEE 1532标准上可以开发通用的编程工具，为测试、编程和系统开发提供规范的接口和器件支持、促进了编程革新，开辟了边界扫描技术新的应用领域。

➢ IEEE1532 主要应用在CPLD、FPGA、PROM以及任意的支持IEEE 1532的可编程器件的在板编程。

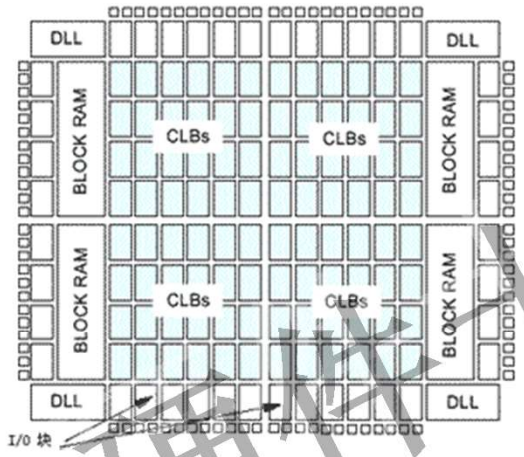
逻辑器件原理——FPGA基本原理

- 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。

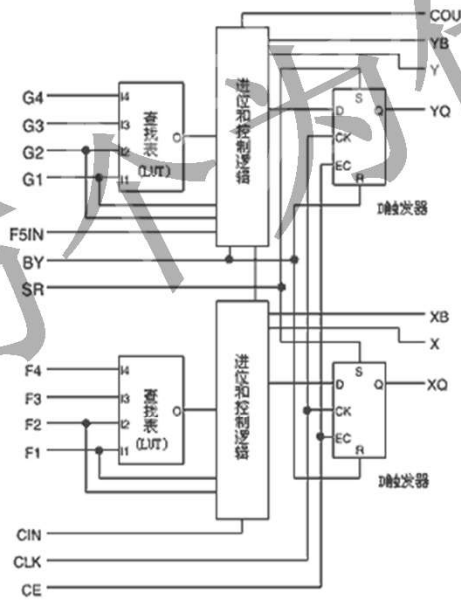
<table>
<tr><td colspan="2">实际逻辑电路</td><td colspan="2">LUT的实现方式</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2"> 
</td></tr>
<tr><td>a,b,c,d 输入</td><td>逻辑输出</td><td>地址</td><td>RAM中存储的内容</td></tr>
<tr><td>0000</td><td>0</td><td>0000</td><td>0</td></tr>
<tr><td>0001</td><td>0</td><td>0001</td><td>0</td></tr>
<tr><td>....</td><td>0</td><td>...</td><td>0</td></tr>
<tr><td>1111</td><td>1</td><td>1111</td><td>1</td></tr>
</table>

逻辑器件原理——FPGA基本原理

- Spartan-II主要包括CLBs，I/O块，RAM块和可编程连线（未表示出）。在spartan-II中，一个CLB包括2个Slices,每个slices包括两个LUT，两个触发器和相关逻辑。 Slices可以看成是SpartanII实现逻辑的最基本结构



xilinx Spartan-II 芯片内部结构



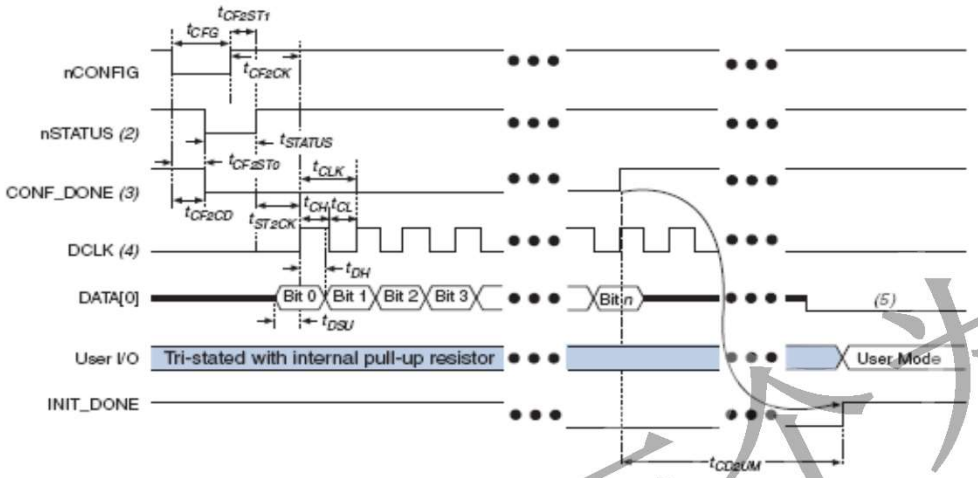
Slices结构

逻辑器件原理——FPGA加载原理

FPGA大部分是基于SRAM编程，编程信息在系统断电时丢失，每次上电时，需从器件外部将编程数据重新写入SRAM中。FPGA加载方式一般有一下几种：

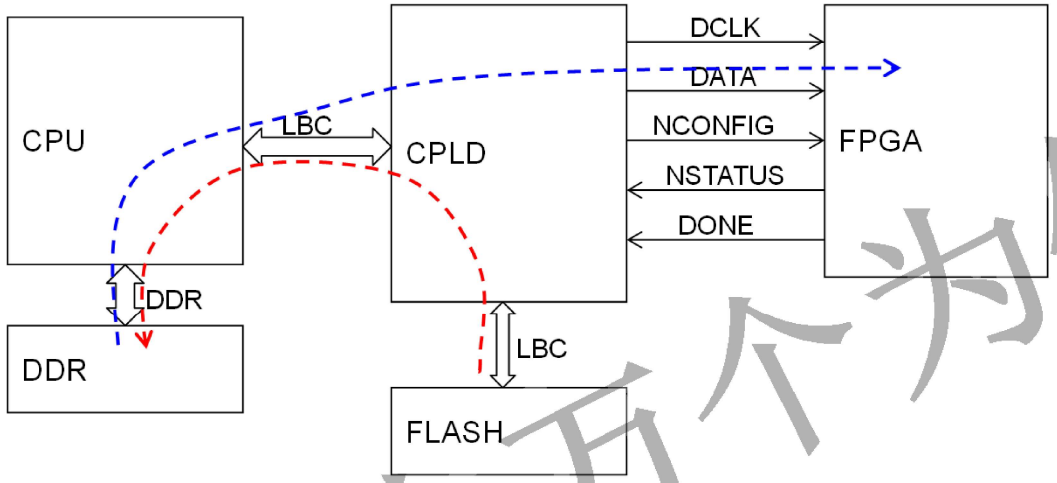
Configuration Scheme	MSEL3	MSEL2	MSEL1	MSEL0	POR Delay	Configuration Voltage Standard (V) (1)
FPP	0	0	0	0	Fast	3.3, 3.0, 2.5
	0	1	1	1	Fast	1.8
FPP with design security feature, decompression, or both enabled (2)	0	0	0	1	Fast	3.3, 3.0, 2.5
	1	0	0	0	Fast	1.8
PS	0	0	1	0	Fast	3.3, 3.0, 2.5
	1	0	0	1	Fast	1.8
	1	0	1	0	Standard	3.3, 3.0, 2.5
	1	0	1	1	Standard	1.8
AS with or without remote system upgrade	0	0	1	1	Fast	3.3
	1	1	0	1	Fast	3.0, 2.5
	1	1	1	0	Standard	3.3
	1	1	1	1	Standard	3.0, 2.5
JTAG-based configuration (3)	(4)	(4)	(4)	(4)	—	—

逻辑器件原理——FPGA加载之从串加载



Symbol	Parameter	Minimum	Maximum	Unit
t_{CF2CD}	nCONFIG low to CONF_DONE low	—	500	ns
t_{CF2ST0}	nCONFIG low to nSTATUS low	—	500	ns
t_{CFG}	nCONFIG low pulse width	500	—	ns
t_{STATUS}	nSTATUS low pulse width	45	230 *(2)*	µs
t_{CF2ST1}	nCONFIG high to nSTATUS high	—	230 *(2)*	µs
t_{CF2CK}	nCONFIG high to first rising edge on DCLK	230 *(2)*	—	µs
t_{ST2CK}	nSTATUS high to first rising edge of DCLK	2	—	µs
t_{DSU}	Data setup time before rising edge on DCLK	5	—	ns
t_{DH}	Data hold time after rising edge on DCLK	0	—	ns
t_{CH}	DCLK high time	3.2	—	ns
t_{CL}	DCLK low time	3.2	—	ns
t_{CLK}	DCLK period	7.5	—	ns
f_{MAX}	DCLK frequency	—	100 *(4)*	MHz
t_{CD2UM}	CONF_DONE high to user mode *(3)*	300	650	µs
t_{CD2CU}	CONF_DONE high to CLKUSR enabled	4 × maximum DCLK period	—	—

逻辑器件原理——FPGA加载之从串加载



参考文档：

CH51VSCUA单板_FP
\程序在线加载失败问

\FPGA从串加载不
成功案例分析.doc

FPGA从串加载提速
改进建议.ppt

逻辑器件原理——可编程逻辑发展历程

- 早期的可编程逻辑器件只有可编程只读存贮器(PROM)、紫外线可按除只读存贮器(EPROM)和电可擦除只读存贮器(EEPROM)三种。由于结构的限制，它们只能完成简单的数字逻辑功能。

  其后，出现了一类结构上稍复杂的可编程芯片，即可编程逻辑器件(PLD)，它能够完成各种数字逻辑功能。典型的PLD由一个“与”门和一个“或”门阵列组成，而任意一个组合逻辑都可以用“与一或”表达式来描述，所以， PLD能以乘积和的形式完成大量的组合逻辑功能，可以实现速度特性较好的逻辑功能，但其过于简单的结构也使它们只能实现规模较小的电路。

  为了弥补这一缺陷，20世纪80年代中期。Altera和Xilinx分别推出了类似于PAL (可编程阵列逻辑)结构的扩展型 CPLD(Complex Programmab1e Logic Dvice)和与标准门阵列类似的FPGA(Field Programmable Gate Array)，它们都具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。这两种器件兼容了PLD和通用门阵列GAL (Generic Array Logic)的优点，可实现较大规模的电路，编程也很灵活。与门阵列等其它ASIC(Application Specific IC)相比，它们又具有设计开发周期短、设计制造成本低、开发工具先进、标准产品无需测试、质量稳定以及可实时在线检验等优点，因此被广泛应用于产品的原型设计和产品生产(一般在10,000件以下)之中。几乎所有应用门阵列、PLD和中小规模通用数字集成电路的场合均可应用FPGA和CPLD器件。

硬件描述语言——Verilog

- Verilog HDL既是一种行为描述的语言也是一种结构描述的语言，有5种模型：
- 系统级（system）
- 算法级（algorithmic）
- RTL级（Register-Transfer-Level）
- 门级（gate-level）
- 开关级（switch-level）（一般情况，我们不涉及）

系统级、算法级和**RTL**级属于行为级，门级属于结构级。数字系统的逻辑设计工程师应熟练掌握。

```
module  fa_struct (a,b, cin, sum);
input    a,b,cin;
output   sum ;
wire     st ;

xor  u_x1 (st, a, b);
xor  u_x2 (sum, st, cin);

endmodule
```

结构级(门级)描述

```
module  fa_struct (a,b, cin, sum, cout);
input    a,b, cin;
output  sum , cout;
reg       cout,sum;

always @ (a or b or cin)
begin
    { cout,sum } = a + b + cin ;
end

endmodule
```

行为级描述

Verilog 语言基础——变量reg

- reg

- 寄存器是数据存储单元的抽象。Verilog HDL语言提供了控制结构用来描述硬件触发条件，使得寄存器的值发生改变，其作用和改变触发器储存的值相当。
- reg类型数据的默认初始值为不定值x
- reg型只表示被定义的信号将用在“always”模块内，常代表触发器或寄存器的输出。但当在always语句中实现组合逻辑时，reg表示组合逻辑的输出。
- 在“always”模块内被赋值的每一个信号都必须定义成reg型。

- reg型数据的格式如下：
- reg [n-1:0] 数据名1,数据名2,… 数据名i;
- 或 reg [n:1] 数据名1,数据名2,… 数据名i;
- 举例：
- reg [3:0] regb; //定义了一个四位的名为regb的reg型数据
- reg [4:1] regc, regd; //定义了两个四位的名为regc和regd的reg型数据

Verilog 语言基础——变量wire

- wire

- 网络数据类型，表示结构实体（例如门）之间的物理连接。不能进行变量的存储，必须收到驱动器的驱动。
- 如果没有驱动器连接到网络类型的变量上，其值应为高阻Z。
- 常用来表示用以assign关键字指定的组合逻辑信号。
- Verilog程序模块中输入输出信号类型默认时自动定义为wire型。

- wire型信号的格式同reg型信号的很类似。其格式如下：
- wire [n-1:0] 数据名1,数据名2,…数据名i;
- 或 wire [n:1] 数据名1,数据名2,…数据名i;

- 举例：
- wire [7:0] b; //定义了一个八位的wire型数据
- wire [4:1] c, d; //定义了二个四位的wire型数据

Verilog 语言基础——变量reg & wire

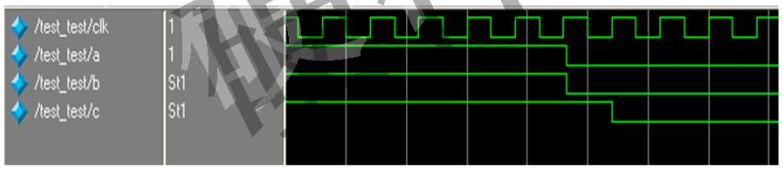
module endmodule input output inout wire reg assign always (四大法宝) posedge negedge （begin end）		
If_else if_ else	case_default endcase	casez_default endcase
parameter、`define 、`timescale、`resetall		

```
module  fa_struct (a,b,c,clk);
input    a,clk;
output   b ,c ;
wire b;
reg c;

assign b = a;

always@(posedge clk)
c <= a;

endmodule
```



- wire----结构化的器件之间的物理连线模型，不存储逻辑值的，须由器件驱动。
- 信号没有定义数据类型时缺省就是wire类型；
- reg----用于对存储单元的描述，如D触发器、T触发器;
- 使用assign语句赋值的信号应定义为wire类型
- 在always、initial等进程中进行描述的输出信号必须用reg类型定义
- reg类型最后综合的结果并不一定是寄存器。

Wire类型常用于组合逻辑，不依靠时钟而赋值；

Reg寄存器类型依靠时钟沿采样赋值，一般用于时序逻辑；

一、整数

- 整型常量有以下四种进制表示形式:
- 1) 二进制整数(b或B)
- 2) 十进制整数(d或D)
- 3) 十六进制整数(h或H)
- 4) 八进制整数(o或O)

- 数字表达方式有以下三种:
- 1) <位宽><进制><数字>这是一种全面的描述方式。
- 2) <进制><数字>在这种描述方式中,数字的位宽采用缺省位宽(这由具体的机器系统
- 决定,但至少32位)。
- 3) <数字>在这种描述方式中,采用缺省进制十进制。

- 举例:
- 8'b10101100 //位宽为8的数的二进制表示, 'b表示二进制
- 8'ha2 //位宽为8的数的十六进制， 'h表示十六进制。

二、 x和z值:

- 在数字电路中,x代表不定值,z代表高阻值。一个x或z可以用来定义十六进制数的四位,八进制数的三位,二进制数的一位。
- 4'b101z //位宽为4的二进制数从低位数起第一位为高阻值
- 8'h4x //位宽为8的十六进制数其低四位值为不定值

三、下划线：

- 下划线可以用来分隔开数的表达以提高程序可读性。但不可以用在位宽和进制处,只能用在具体的数字之间
- 16'b1010_1011_1111_1010 //合法格式
- 8'b_0011_1010 //非法格式

四、(Parameter)型

- parameter来定义常量，提高程序的可读性和可维护性。
- 定义方式： parameter 参数名1＝表达式，参数名2＝表达式, …， 参数名n＝表达式;
- 举例： parameter e=25, f=29; //定义二个常数参数

Verilog 语言基础——运算符

- 1) 算术运算符(+,－,×，/,%)
- 2) 赋值运算符(=,<=)
- 3) 关系运算符(>,<,>=,<=)
- 4) 逻辑运算符(&&,||,!)
- 5) 条件运算符(?:)
- 6) 位运算符(~,|,^,&,^~)
- 7) 移位运算符(<<,>>)
- 8) 拼接运算符({ })

优先级别	
! ~	最高优先级别
* / %	
+ -	
<< >>	
< <= > >=	
== != === !==	
&	
^ ^~	
|	
&&	
||	最低优先级别
?:	

Verilog 语言基础——运算符

- 算术运算符（+，-，*，/，%）

- %模运算符，要求%两侧均为整型数据。如9%2=1
- 在进行整数除法运算时，结果值要略去小数部分，只取整数部分；而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号为。如：-10%3 = -1；11%-3=2
- 在进行算术运算操作时，如果一个操作数为不定值X，则整个结果也为不定值X。

- 位运算符（~，|，^，&，^~）

- rega = 4'b1010 ; regb = 4'b1101;
- reg1 = ~rega; // rega的值进行按位取反运算后变为4'0101
- reg2 = rega | regb; //rega和regb按位或后职位4'1111
- reg3 = rega & regb; //rega和regb按位与后职位4'1000
- reg4 = rega ^ regb; //rega和regb按位异或后职位4'0111
- reg5 = rega ^~ regb; //rega和regb按位异或非后职位4'1000

Verilog 语言基础——运算符

- 关系运算符（<，>，<=，>=）
- 所有的关系运算符有着相同的优先级别，关系运算符优先级别低于算术运算符。
- a < size-1 ⇨ a <（size-1）
- size –（1 < a）⇨/ (size – 1) < a
- 为提高程序的可读性，明确表达个运算符间的优先级关系，建议使用括号。
- 位拼接运算符（{ }）
- {4{w}} ⇨ {w，w，w，w} {b，{3{a，b}}} ⇨ {b，a，b，a，b，a，b}
- 缩减运算符（&，|，~）
- 单目运算符，运算结果1位二进制数。运算过程：先将操作数的第1位与第2位进行或、与、非运算；再将运算结果与第3位进行或、与、非运算，依此类推，直至最后1位。
- eg: reg [3:0] B; reg C; C = &B // 相当于C=((B[0]& B[1]) &B[2]) &B[3]
- 条件运算符（？：）
- 逻辑运算符（&&，||，！）
- 此处区分逻辑与和按位与运算符；设 rega = 4`b1010 ; regb = 4`b1101;
- rega & regb = 4`b1000;
- rega && regb = 0
- 移位运算符（<<，>>）

Verilog 语言基础——阻塞&非阻塞赋值

- 非阻塞(Non-Blocking)赋值方式（如b<=a）

1. 在语句块中，上面语句所赋的变量值不能立即就为下面的语句所用；
2. 块结束后才能完成这次赋值操作，而所赋的变量值是上一次赋值得到的。
3. 在编写可综合的时序逻辑模块时，这是常用的赋值方法。
4. 对应的电路结构往往与触发沿有关系，只有在触发沿时才有可能发生赋值的情况。

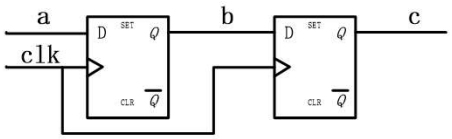
阻塞（Blocking）赋值方式（如b=a）

- 赋值语句执行完后，块才结束；
- 在赋值语句执行完后变量的值立刻就改变；
- 在时序逻辑中使用时，可能会产生意想不到的结果。
- 对应电路结构往往于触发沿没有关系，只与输入电平的变化有关系。

思考：这两类赋值所表示的电路有什么不同?

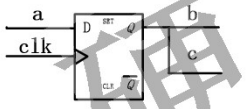
Verilog 语言基础——阻塞&非阻塞赋值

```
always @(posedge clk)
    begin
        b <= a;
        c <= b;    //非阻塞赋值
    end
```



非阻塞赋值对应电路图

```
always @(posedge clk)
    begin
        b = a;
        c = b;    //阻塞赋值
    end
```



阻塞赋值对应电路图

这个电路，一般不是我们想要。

- 总结：

1. 时序电路建模时，用非阻塞赋值；
2. 锁存器电路建模时，用非阻塞赋值；
3. 用always块建立组合逻辑模型时，用阻塞赋值；
4. 在同一个always块中建立时序和组合逻辑电路时，用非阻塞赋值；
5. 在同一个always块中不要既用非阻塞赋值又用阻塞赋值。
6. 不要在一个以上的always块中为同一个变量赋值。

Verilog 语言基础——If-else

```
//产生输入时钟无效告警，1告警，0正常
always @(posedge clk or posedge reset or posedge clk_lost)
    if (reset == 1'b1)
        err <= #`UDLY 1'b0;
    else if (clk_lost == 1'b1)                    //检测时钟丢失，置为告警
        err <= #`UDLY 1'b1;
    else if (signal_lost2 == 1'b1)                //检测时钟丢失，置为告警
        err <= #`UDLY 1'b1;
    else if (flag_reg == 3'b000)                  //移位寄存器都为0时判定输入时钟有效
        err <= #`UDLY 1'b0;
    else if (flag_reg == 3'b111)                  //移位寄存器都为1时判定输入时钟无效
        err <= #`UDLY 1'b1;
```

一个信号在进程中赋值，不允许出现多个else分支中对该信号赋值一样的设计，异步复位分支除外。

```
reg   din, dout;
always @(posedge clk  or posedge reset)
    begin
        if(reset ==1' b1)
            dout    <= #' UDLY 1' b0;
        else if (clr ==1' b1)
            dout    <= #' UDLY 1' b0;
        else
            dout    <= #' UDLY din;
    end
```

多个条件？条件存在优先级？

如果A B C 三个条件存在优先级：
从高到低分别为A B C
A 条件 –》赋值为 1
B 条件 –》赋值为 0
C 条件 –》赋值为 1
其他 –》赋值为 0

```
always @(posedge clk or posedge reset)
    if (reset ==1'b1)
        wr <= #UDLY 1'b0;
    else if (条件A |(条件C |(~条件B)))
        wr <= #UDLY 1'b1;
    else
        wr <= #UDLY 1'b0;
```

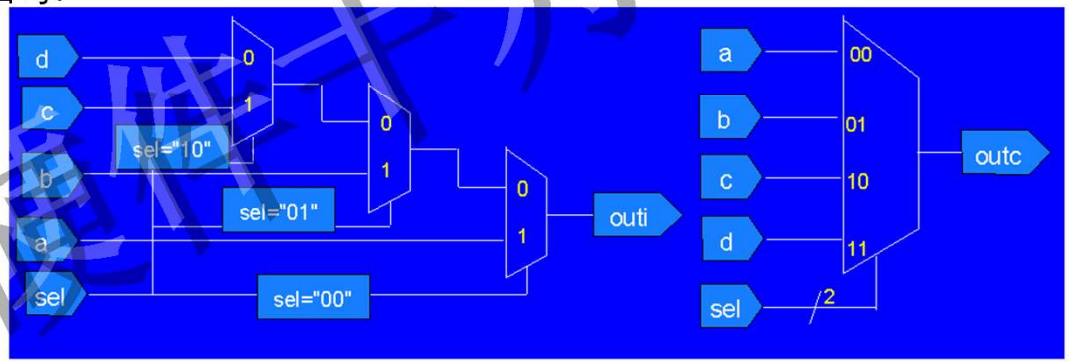
Verilog 语言基础——case

- case语句是一种多分支选择语句；
- Verilog HDL语句提供了三类case：

1. case—(default)—endcase;
2. casez—(default)—endcase;
3. casex—(default)—endcase;(规范中禁止使用)

```
case (code)
    3'b001 : led = 7'b1111001;      //1
    3'b010 : led = 7'b0100100;      //2
    3'b011 : led = 7'b0110000;      //3
    3'b100 : led = 7'b0011001;      //4
    3'b101 : led = 7'b0011010;      //5
    3'b110 : led = 7'b0000010;      //6
    3'b111 : led = 7'b1111000;      //7
    default : led = 7'b1000000;     //0
endcase
```

比较if-else和case电路，可见，左边else if所实现的电路的输入到输出的布线延迟是和else if 的级数相关。一般要求这个级数不要大于5级。条件分支之间没有优先级关系时，采用case语句。



Verilog 语言基础——锁存器

```
always @ (al or d)
  begin
    if(al)
     q<=d;
  end
```

有锁存器

```
always @ (al or d)
  begin
    if(al)  q<=d;
    else   q<=0;
  end
```

无锁存器

```
always @ (al[1:0] or
d or c)
  case (al[1:0] )
    2'b00 : q<=d;
    2'b11 : q<=c;
  endcase
```

有锁存器

```
always @ (al[1:0] or
d or c)
  case (al[1:0] )
    2'b00 : q<=d;
    2'b11 : q<=c;
    default:q<=1'b0;
  endcase
```

无锁存器

```
always @ (al[1:0] )
  case (al[1:0] )
    2'b00 : q<=d;
    2'b11 : q<=c;

default:q<=1'b0;
  endcase
```

有锁存器

```
always @ (al[1:0]
or d or c)
  case (al[1:0] )
    2'b00 : q<=d;
    2'b11 : q<=c;

default:q<=1'b0;
  endcase
```

无锁存器

- 总结
- 如果用到**if**语句，应该写上**else**项；
- 如果用到**case**语句，应该写上**default**项
- 所有的输入都必须在敏感列表中；
- 仔细检查综合器的综合报告，目前大多数的综合器对所综合出的**Latch**都会报“**warning**”，通过综合报告可以较为方便地找出无意中生成的**Latch**。

组合逻辑

- 组合逻辑特征：任何一个输入上有任何变化，输出将会立即根据电路输入来重新计算并驱动产生输出。
- 组合逻辑产生方法
- 连续赋值语句

```
wire      out;              //使用assign赋值的信号需定义为net（线网）类型
assign  out  =  a & b;      //将a和b相与输出给该信号
```

- always进程

```
reg   out;                  //每个always进程中被赋值的信号均需命名为reg类型
always @(a or b or sel)     //敏感信号列表（always进程的所有输入信号）
    begin
        if (sel == 1'b1)
            out = a;
        else
            out = b;
    end
```

总结：使用always进程产生组合逻辑的规则

a、在所有可能的控制路径里，输出都要被赋值；

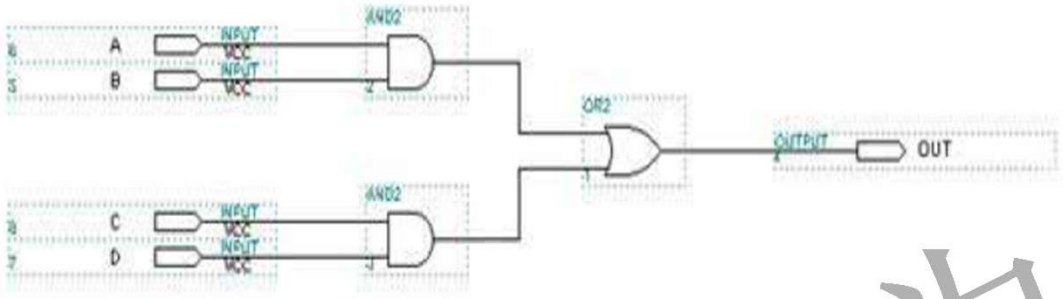
b、所有的输入都必须在敏感列表中；

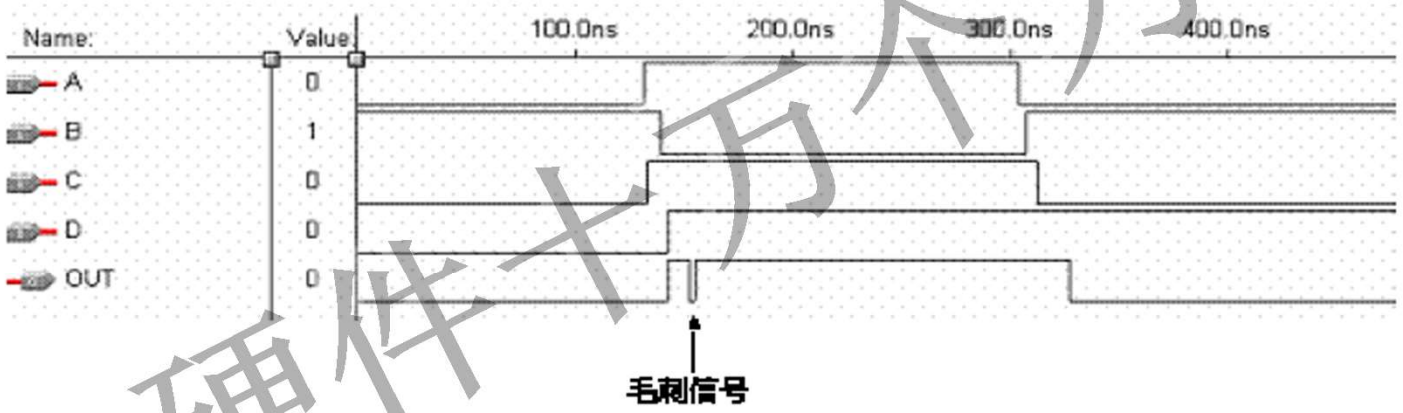
c、敏感列表中不能有边沿信号说明。

组合逻辑——竞争冒险

- 信号在逻辑器件内部通过连线和逻辑单元时，都有一定的延时。延时的大小与连线的长短和逻辑单元的数目有关，同时还受器件的制造工艺、工作电压、温度等条件的影响。信号的高低电平转换也需要一定的过渡时间。由于存在这两方面因素，多路信号的电平值发生变化时，在信号变化的瞬间，组合逻辑的输出有先后顺序，并不是同时变化，往往会出现一些不正确的尖峰信号，这些尖峰信号称为"毛刺"。如果一个组合逻辑电路中有"毛刺"出现，就说明该电路存在"冒险"。

组合逻辑——竞争冒险

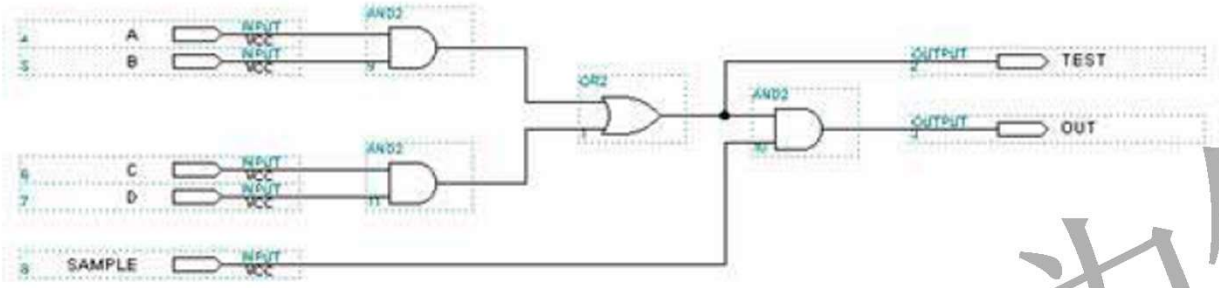


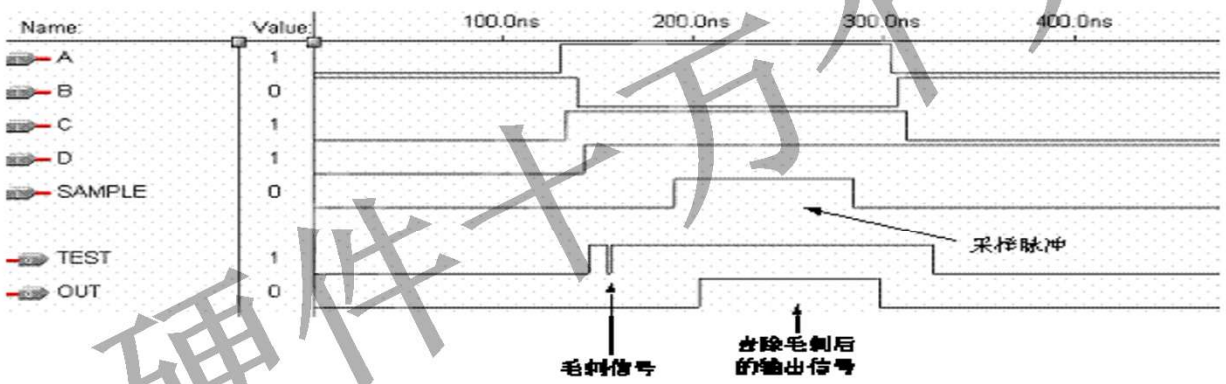


组合逻辑——竞争冒险危害

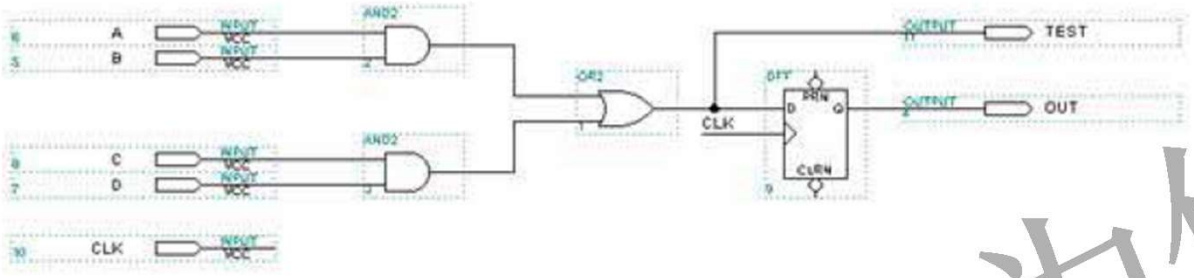
冒险往往会影响到逻辑电路的稳定性。尤其是时钟端口、清零和置位端口对毛刺信号十分敏感，任何一点毛刺都可能会使系统出错，因此判断逻辑电路中是否存在冒险以及如何避免冒险是设计人员必须要考虑的问题。

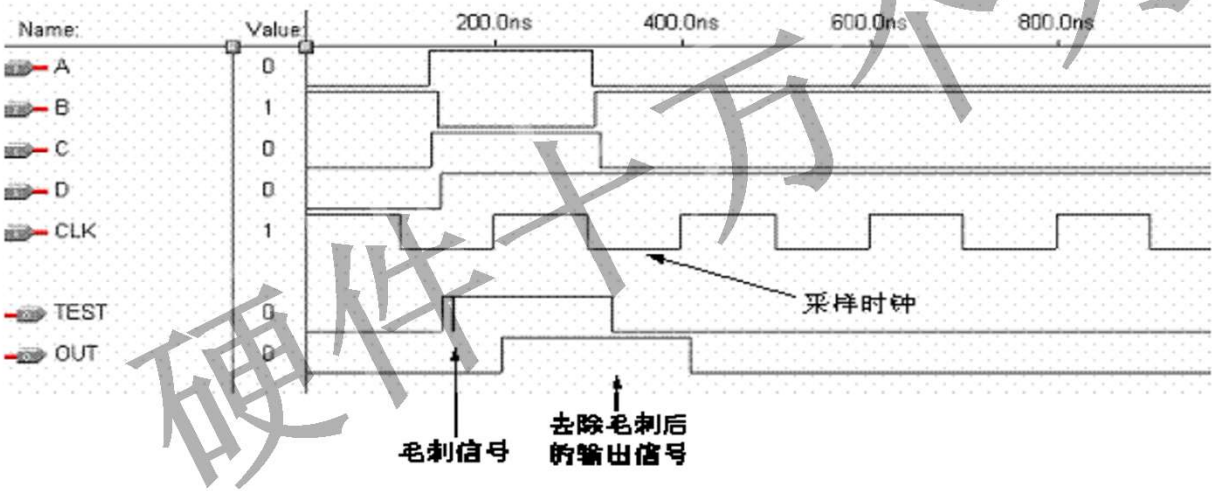
组合逻辑——竞争冒险消除之增加使能信号





组合逻辑——竞争冒险消除之D触发器采样





时序逻辑

- 时序电路基本特征：电路能够寄存信号值，信号在时钟沿被采样并进行重新计算输出值。
- 时序逻辑产生：使用always进程，产生触发器，实现时序逻辑

```
reg   [15:0] d;          //定义16位reg型变量，用于always模块内部被赋值
always @( posedge clk or posedge a_rst )
  if (a_rst == 1'b1)                    //异步复位
     d <=  #UDLY 16'h0000;
  else if (s_clr == 1'b1)               //同步清零
     d <=  #UDLY 16'h0000;
  else if (en == 1'b1)                  //每个使能en有效则计数
     d <=  #UDLY d + 1;
```

使用always进程产生时序逻辑，产生触发器设计：

a、 敏感列表中只存在边沿说明。

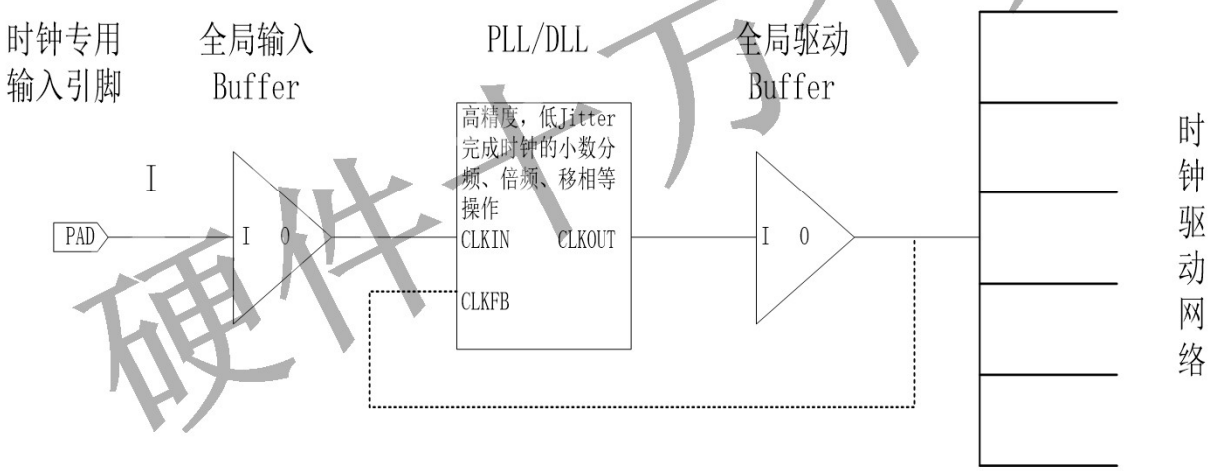
b、敏感列表中至少存在一个边沿信号说明，从该边沿说明中，工具推断出一个触发器；

c、 always块中的所有的寄存器都被声明的时钟边沿来定时。

d、至于控制路径的完整性对触发器的推断并不重要

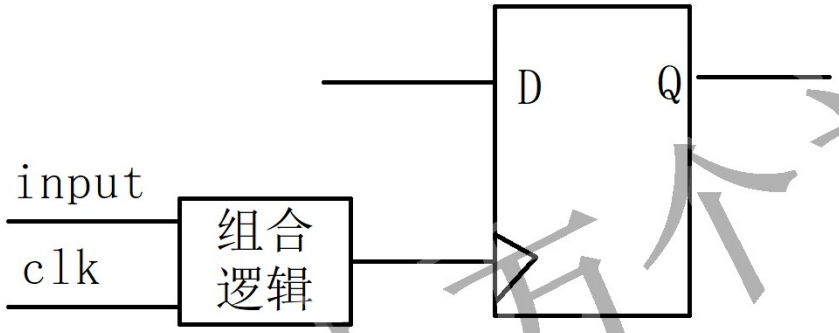
时钟——全局时钟

- 同步时序电路推荐的时钟使用方式为：
- 由全局时钟专用引脚输入，通过PAD的Skew和Jitter等都最小。而且专用全局时钟引脚到PLL和全局时钟驱动资源的路径最短。
- 使用PLL或DLL进行分频/倍频、移相等调整，附加Skew和Jitter都最小，而且操作简单，精度高。



时钟——门控时钟

门控时钟：使用一个控制门电路的使能信号控制时钟的打开和关闭，当时钟关闭时，相应的时钟域停止工作。目的是为了节能。

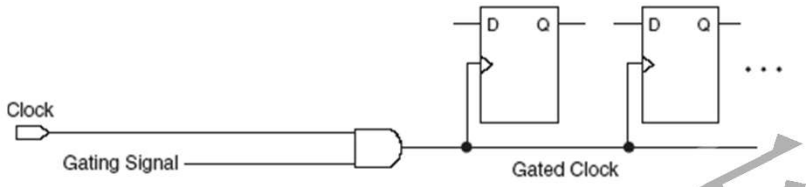


门控时钟是非常危险的，极易产生毛刺，使逻辑误动作，一般很少使用，为了降低功耗时可以使用门控时钟。

时钟——门控时钟

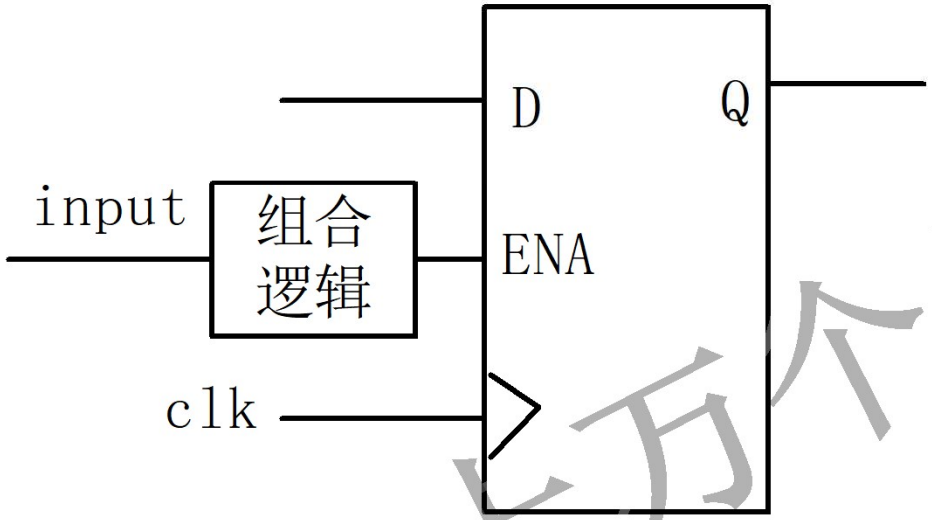
符合以下两个条件，门控时钟可以象全局时钟一样可靠地工作：

1. 驱动时钟的逻辑必须只包含一个“与”门或一个“或”门。如果采用任何附加逻辑；在某些工作状态下，会出现竞争产生的毛刺；



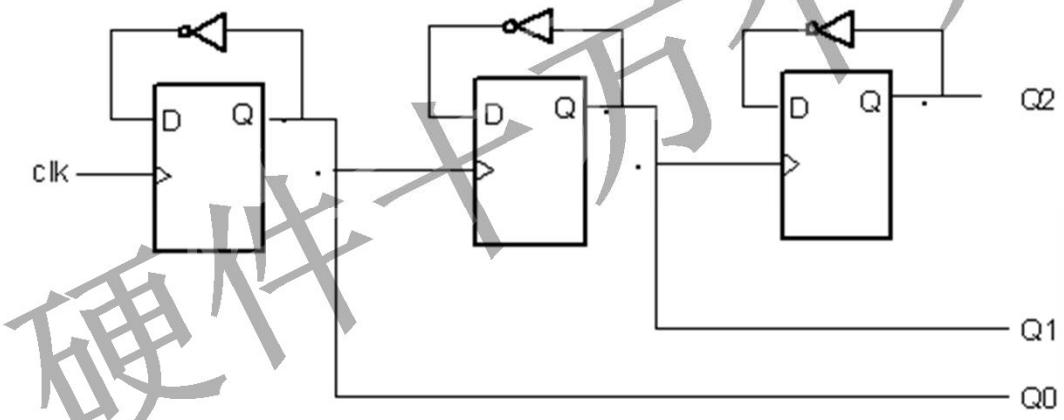
2. 逻辑门的一个输入作为实际的时钟，其它所有输入相对于时钟满足建立和保持时间的约束。

时钟——带使能同步电路



时钟——衍生时钟

- 所谓衍生时钟（行波时钟），即是用一个寄存器的输出作为另一个寄存器的时钟输入，理论上这种时钟没有毛刺，并且良好的设计将可以同全局时钟一样可靠，但是，如果设计是对时延非常敏感的设计，那么，这种时钟系统就不可避免的存在不可靠的因素，因为经过多级设计延时后，将难以估计这种时钟在链上各个触发器时钟之间产生的大量时间偏移，如果这种时间偏移引起建立时间、保持时间难以满足的话，那么设计就极有可能要失败了。

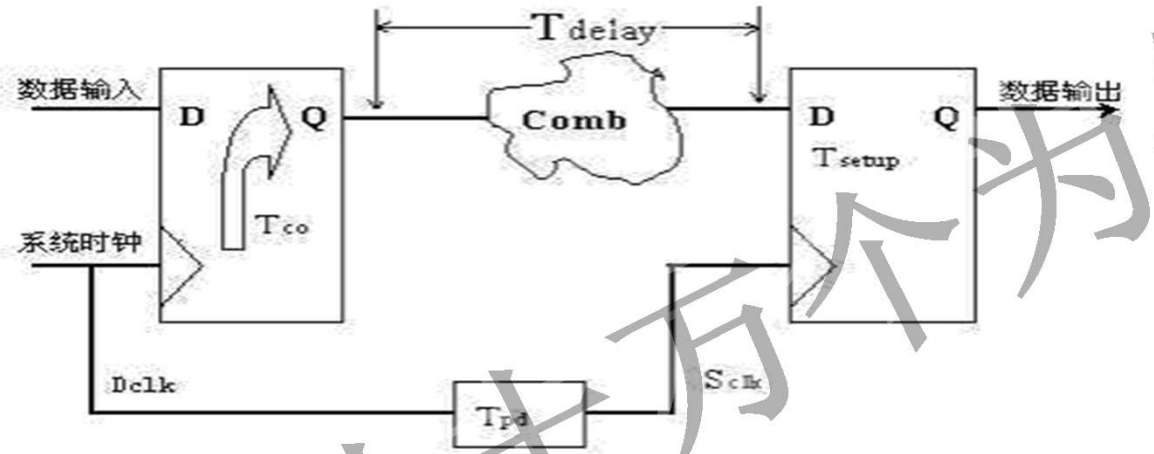


时钟使用建议

- 1、最优的使用时钟的方式为：整个工程的所有时钟均使用全局时钟，最好都在一个时钟域，以此来保证数据的同步性。
- 2、如果需要用到衍生时钟，将衍生时钟作为一个同步电路的使能控制端，时钟仍采用主时钟。如果不可避免的使用了衍生时钟，那么需要对衍生时钟进行约束至全局时钟，如果约束不到全局时钟，则需要对时钟进行hold margin设置，保证时钟的可靠工作。
- 3、如果需要用到门控时钟，则需要注意门控时钟的输入是否满足前面提到的两个条件。

时序逻辑——同步电路模型

基本模型



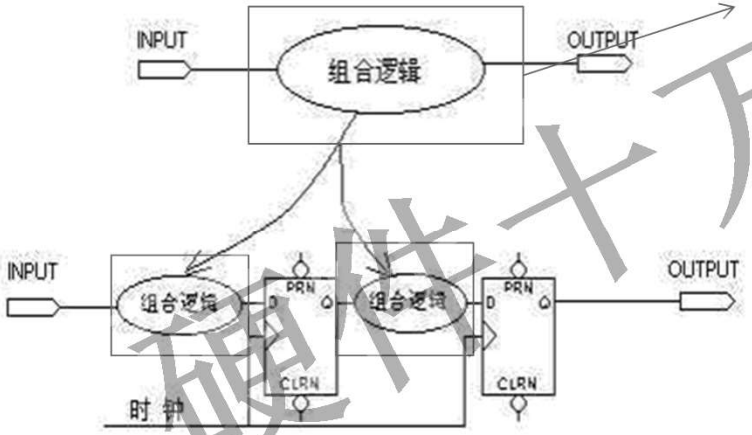
时序逻辑——同步电路速度问题

同步电路的速度是指同步时钟的速度。

同步时钟越快，电路处理数据的时间间隔越短，电路在单位时间处理的数据量就越大。

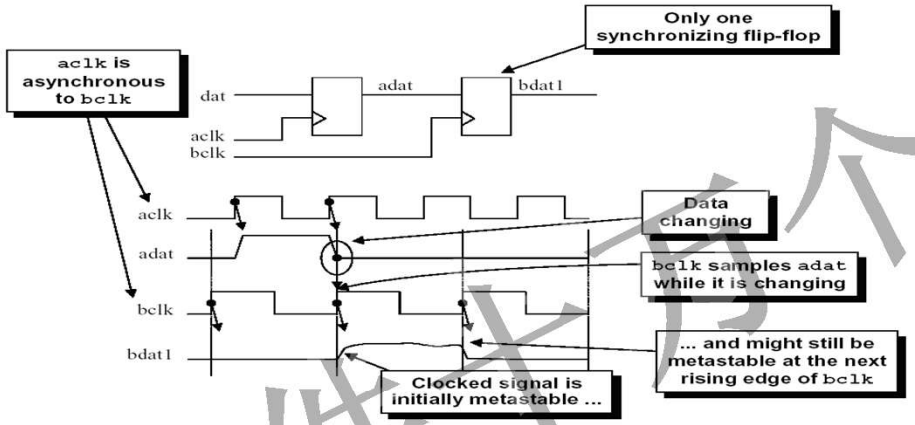
T=Tco + Tdelay + Tsetup – Tpd ， 最快时钟频率**F=1/T**

提升同步时钟的速率：

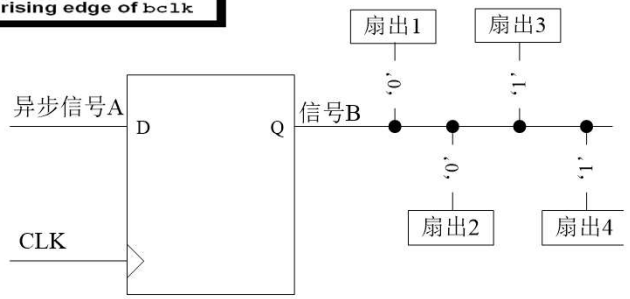


时序逻辑——异步信号问题

- **异步信号：** 异步信号对于本地时钟域系统都意味着一个不稳定的源，因为总是存在这种可能，时钟会在异步信号变化的时候进行了采样，导致建立/保持时间不稳定，D触发器进入亚稳定状态。

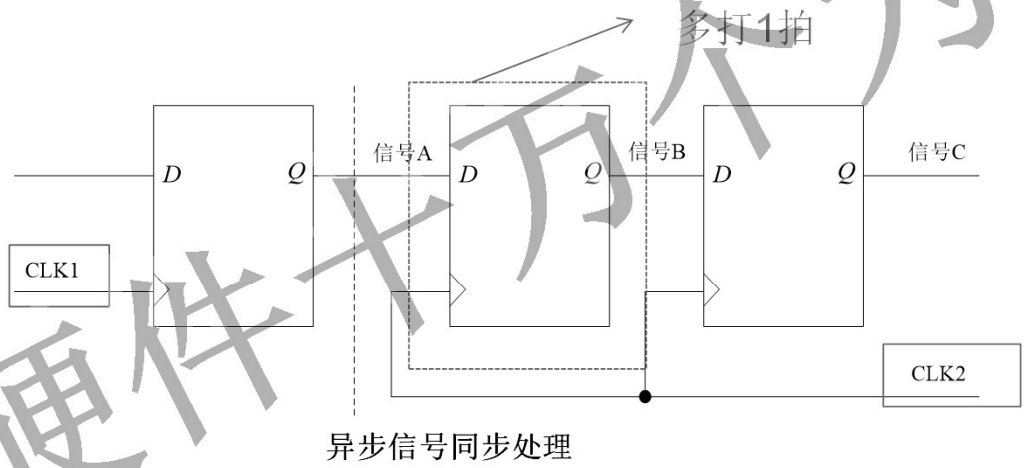


危害： 异步信号A的输入，可能导致亚稳态，引起信号B的输出状态不稳定。如果B信号有多个扇出，那么就不能保证所有的扇出将亚稳态的信号识别为相同的逻辑电平值，电路判断出现混乱。



时序逻辑——异步信号处理

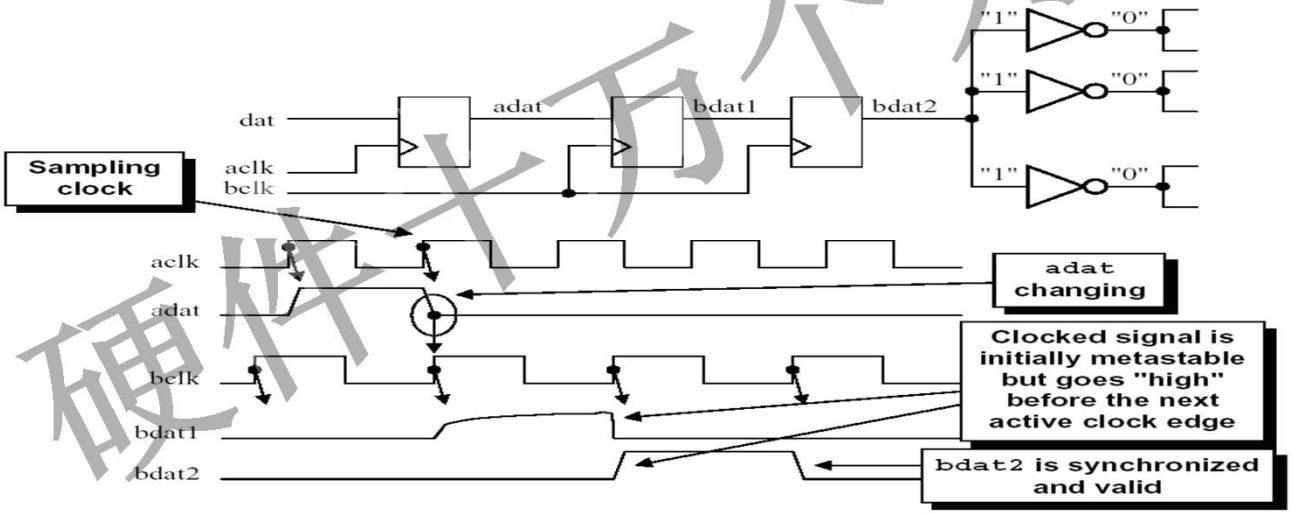
- 主要目的：实现两个时钟域数据的可靠交换
- 核心：保证下级时钟对上级数据采样的Setup时间和Hold时间，避免亚稳态。（注：如果触发器的Setup time或者Hold time不满足，就可能产生亚稳态，此时触发器输出端Q在有效时钟沿之后比较长的一段时间处于不确定的状态，在这段时间里Q端毛刺、振荡、固定的某一电压值，而不是等于数据输入端D的值。这段之间成为决断时间（Resolution time）。经过Resolution time之后Q端将稳定到0或1上，但是究竟是0还是1，这是随机的，与输入没有必然的关系）



异步信号同步处理

时序逻辑——亚稳态

- 使用两级寄存器采样可以有效的减少亚稳态继续传播的概率，但是无法纠正由于亚稳态导致的数据错误。在图中，左边为异步输入端，经过两级触发器采样，在右边的输出与bclk同步的，而且该输出基本不存在亚稳态。其原理是即使第一个触发器的输出端存在亚稳态，经过一个Clk周期后，第二个触发器D端的电平仍未稳定的概率非常小，因此第二个触发器Q端基本不会产生亚稳态。理论上如果再添加一级寄存器，使同步采样达到3级，则末级输出为亚稳态的概率几乎等于0
- **经过两级触发器采样能解决异步信号的亚稳态问题，但是不能保证输出的值是正确的。



时序逻辑——同步设计原则

- 同频异相问题

- 同频异相问题的简单解决方法是用后级时钟对前级数据采样2次，即通常所述的用寄存器打两次。这样的做法是有效的减少了亚稳态的传播，使后级电路数据都是有效电平值。这种方法适用于对少量错误不敏感的功能单元。
- 可靠的做法是用DPRAM、FIFO、或者一段寄存器Buffer完成异步时钟域的数据转换。把数据存放在DPRAM或FIFO的方法如下：将上级芯片提供的数据随路时钟作为写信号，将数据写入DPRAM或者FIFO，然后使用本级的采样时钟（一般是数据处理的主时钟），将数据读出来即可。由于时钟频率相同，所以DPRAM或FIFO两端的数据吞吐率一致，实现起来相对简单

- 异频问题

可靠完成异频问题的解决方法就是使用DPRAM或FIFO。其实现思路与前面所述一致，用上级随路时钟写上级数据，然后用本级时钟读出数据。但是由于时钟频率不同，所以两个端口的数据吞吐率不一致，所以设计时一定要开好缓冲区，并通过监控（Full、Half、Empty等指示）确保数据流不会溢出。

电路实现分析

- 个人认为，一个逻辑设计者一定要从电路思想去进行逻辑设计，如果认为Verilog跟C语言差不多，那么很可能走火入魔。
- 一定要建立起语句与逻辑电路的关联性，看到代码，但是出现的是电路。

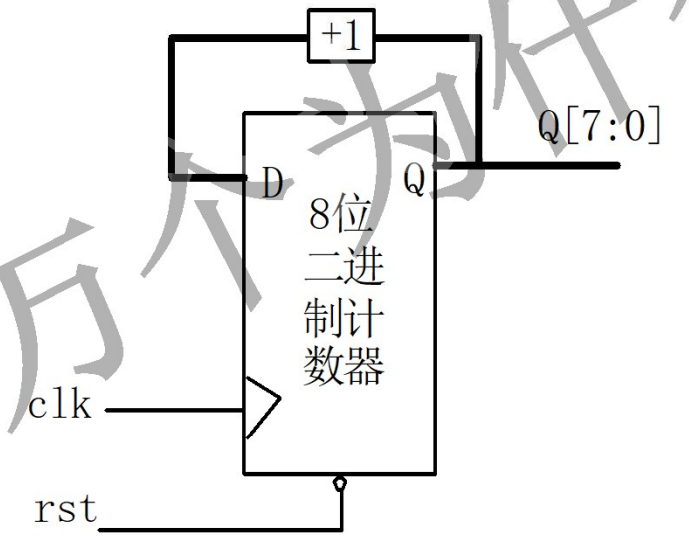
电路实现分析——常用电路设计

- 计数器
- MUX
- 译码器
- 上升下降沿检测电路

常用电路设计——计数器1

普通的8bit计数器，计满自动翻转

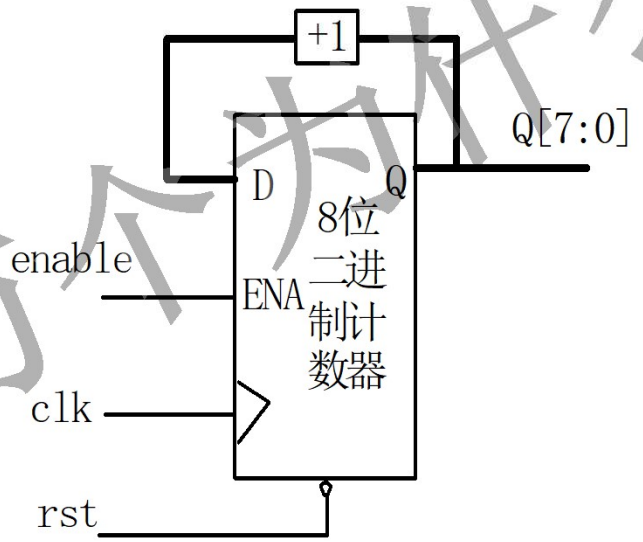
```
always@(posedge clock or negedge rst_n)
begin
    if (rst_n == 1'b0)
        count <= 8'd0;
    else
        count <= count + 1;
end
```



常用电路设计——计数器2

普通的带使能信号的8bit计数器

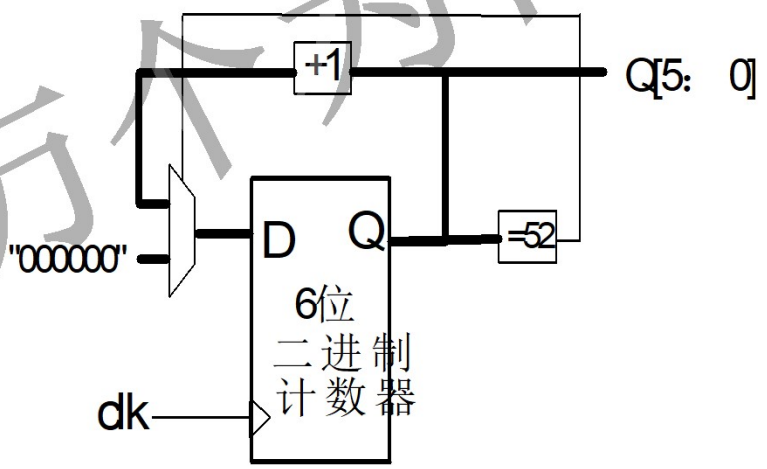
```
always@(posedge clock or negedge rst_n)
begin
    if (rst_n == 1'b0)
        count <= 8'd0;
    else if (enable == 1'b1)
        count <= count + 1;
end;
```



常用电路设计——计数器3

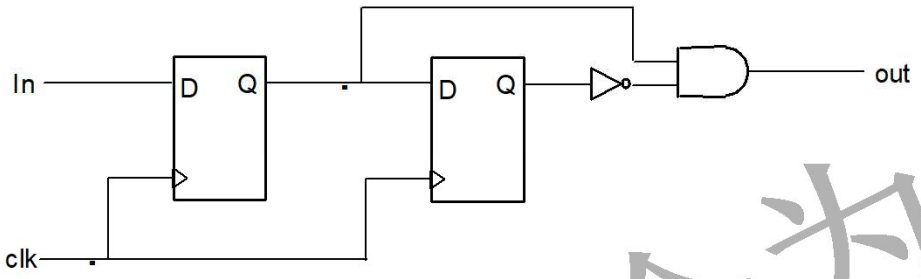
同步清0的不规则计数器

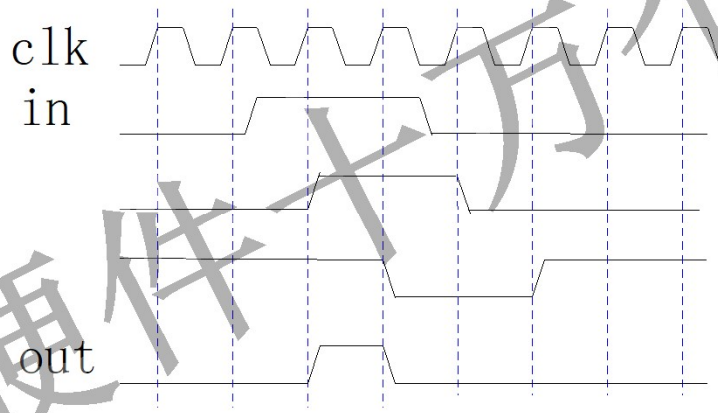
```
always@(posedge clock or negedge
reset)begin
    if (reset == 1'b1)
        count <= 6'd0;
    else if (count >= 6'd52)
        count <= 6'd0;
    else
     count <= count + 1;
end
```



```
always@(posedge clock or negedge rst_n)begin
    if (rst_n == 1'b0)
       mux_out <= 2'b00;
   else
       case(sel)
          2'b00:   mux_out <= 2'b00;
          2'b01:   mux_out <= 2'b01;
          2'b10:   mux_out <= 2'b10;
          2'b11:   mux_out <= 2'b11;
          default: mux_out <= 2'b00;
       endcase
end
```

常用电路设计——上升沿检测电路





电路工作的条件：
时钟周期小于输入信号的脉冲宽度。

常用电路设计——下降沿检测电路

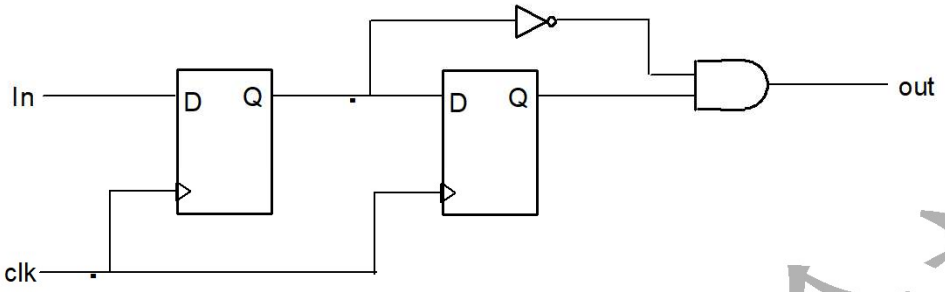
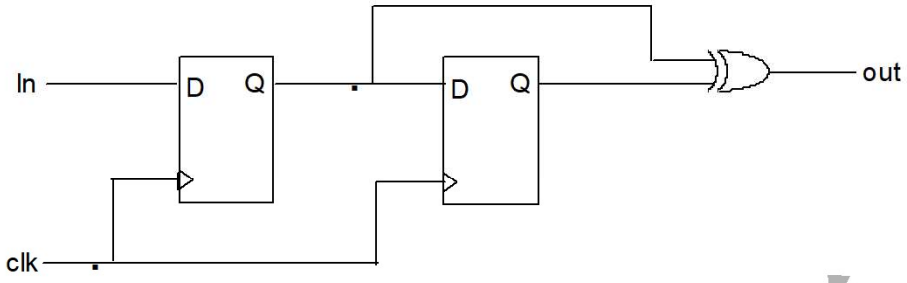


图4.16

电路工作的条件：
时钟周期小于输入信号的脉冲宽度。

常用电路设计——双沿检测电路



电路工作的条件：
时钟周期小于输入信号的脉冲宽度。

逻辑仿真验证

- 章节简介
- 本章介绍仿真工具的使用；通过结合实例讲解激励脚本编写；供初学者快速上手。

1.Modelsim使用简介

- Mentor公司的ModelSim是业界最优秀的HDL语言仿真软件，它能提供友好的仿真环境，是业界唯一的单内核支持VHDL和Verilog混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护IP核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是FPGA/ASIC设计的首选仿真软件。

2.如何编写激励脚本

- 光有仿真软件和源代码是不够的，要想看到输出波形，必须有对应的激励；逻辑代码的激励脚本就是一个.v(verilog)的文件。做好逻辑验证工作；与写好激励脚本是分不开的。

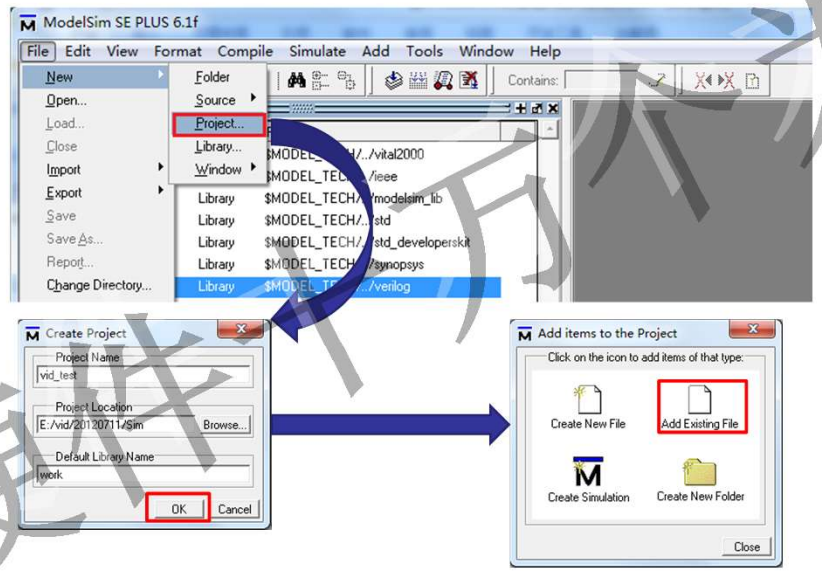
Modelsim使用简介

创建一个新的工程：

•点击File→ New→Create a Project按钮，新建一个工程，指定工程名和工程目录等。

•将产生一个.mpf文件和一个工作库在工程目录中，可以保留缺省库名为work，点击OK按钮。

•选择添加已有文件(也可以后续添加)，开始加入文件，工程可以引用或包括verilog HDL源代码和激励文件。



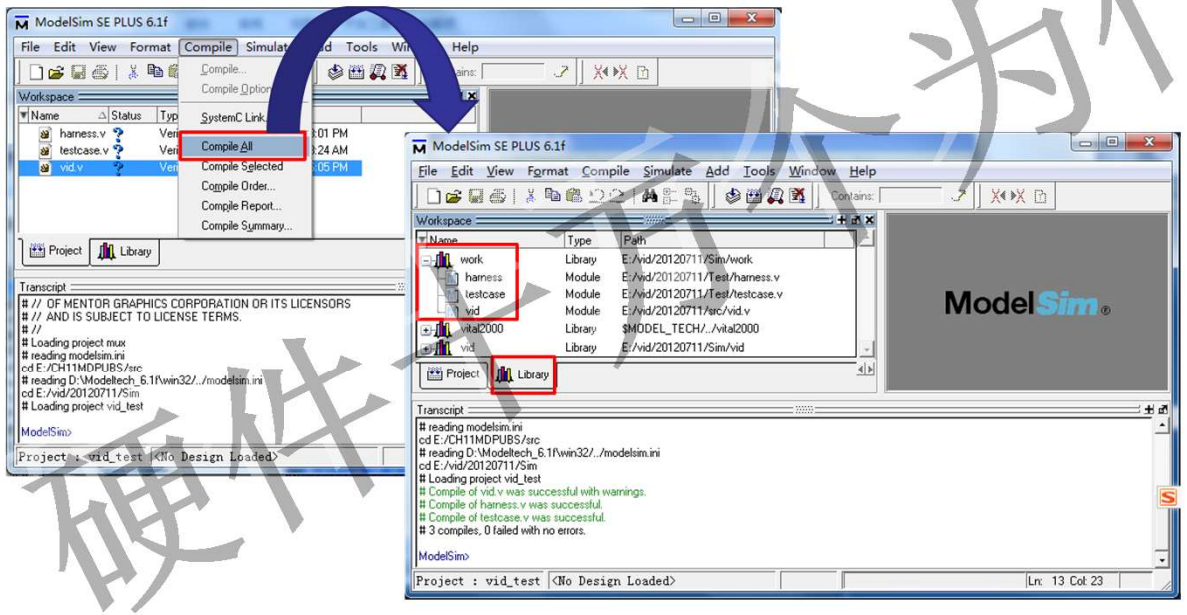
Modelsim使用简介

编译文件：

•点击Compile→ Compile All中点击鼠标右键或者按图示选择，选择Compile All;

•编译作用是检查语法和语义，并产生虚拟机器码，Modelsim用它来进行仿真;

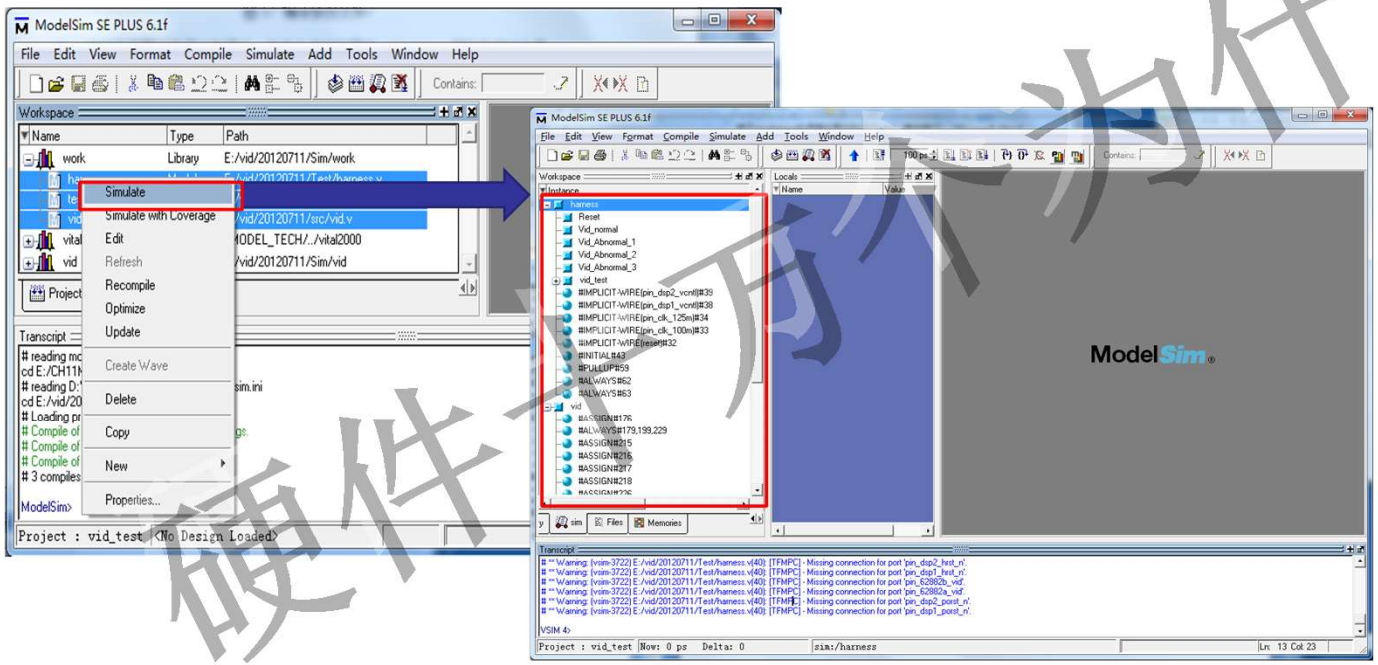
•编译完成后，点击Library tab，可以看到编译后的设计;



Modelsim使用简介

信号仿真：

•选中待仿真文件→右键→Simulate；运行后得到右边的列表；

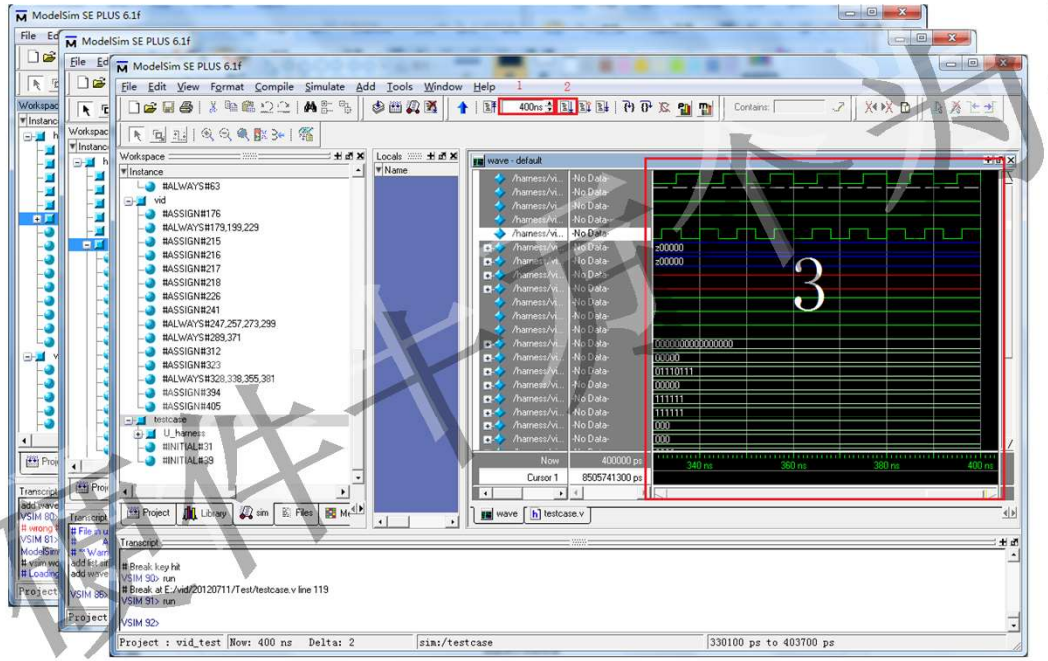


Modelsim使用简介

信号仿真：

•选中激励文件→add→add to wave；运行后得到右边的列表；

•设置仿真时间，单击后面的run；即可得到仿真波形，整个仿真过程结束。



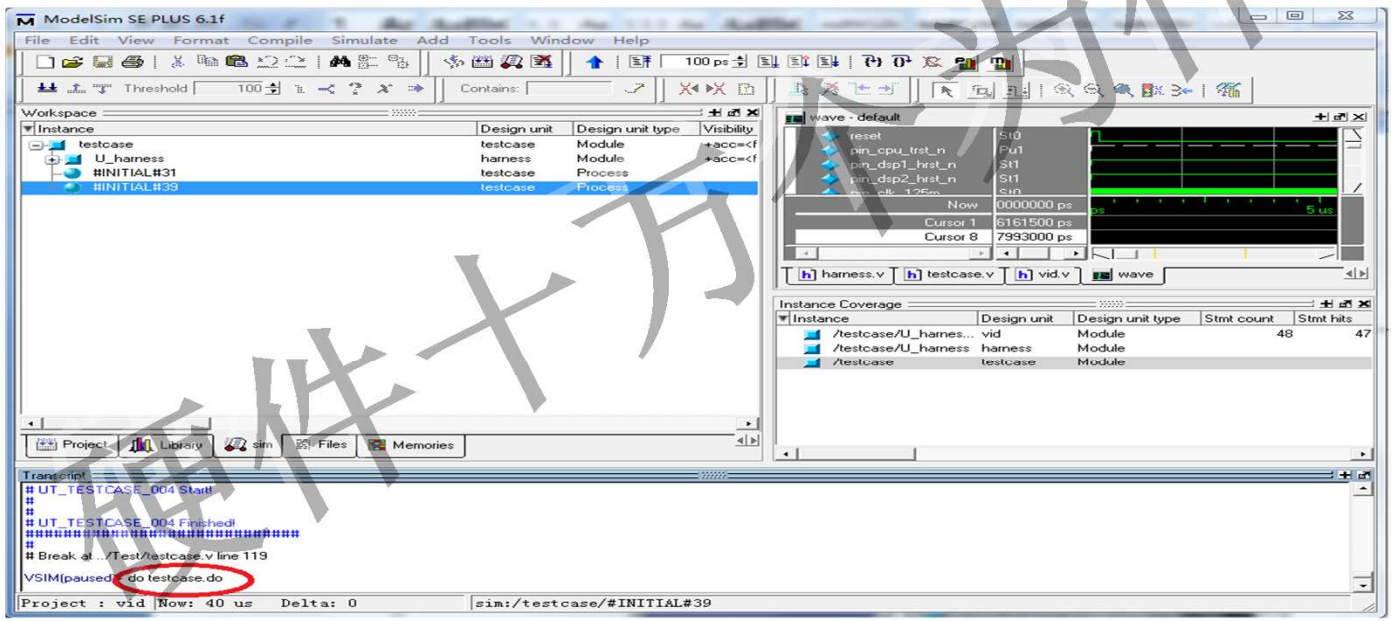
如何编写激励脚本

仿真文件在逻辑工程中的分布结构：

```
SIM     //仿真工程与脚本
    |--testcase.do
    |--wave.do
SRC     //逻辑顶层文件及子文件
    |--CHXXMXX.V
Syn     //逻辑工程
    |--XX.syn
Test    //激励文件与测试用例
    |--harness.v
    |--testcase.v
```

仿真脚本——testcase.do

仿真脚本的用于一键式完成仿真，我们将上述脚本文件与仿真工程置于同一个目录下，在ModelSim的Transcript中执行testcase.do文件，则软件会依据其中的代码自动执行仿真,如下：



仿真脚本说明

```
destroy .wave                                  //把之前的波形都干掉，不需要修改这句话
quit –sim                                      //退出之前的仿真，不需要修改这句话
# create a project
vlib work                                      //建库，不需要修改
vmap work work
# compile source file
vlog -cover bcesx -incr {../src/*.v}           //编译源代码，如果按照规范存放路径，不需要修改。
      Modelsim
# compile testbench                            //版本不同，可能需要修改。-cover为收集覆盖率
vlog -nocoverage -incr {../src/test_bench/*.v} //同上，对测试向量不收集覆盖率
# load design
vsim -novopt -coverage work.selectmap_tb       //仿真，根据需要修改模块名
#vsim -novopt work.selectmap_tb
add wave -r /*                                 //看所有波形
#do wave.do                    //若不想看所有波形，可把想看的波形存入wave.do，然后执行此语句
run 1000us                                     //run，时间自己修改
#run 5sec
//以下为测试覆盖率的结果存入文件，如果不关注可以不加。也可以在窗口查看结果。
```

设置波形文件——wave.do

仿真脚本只是将ModelSim中的一系列指令集成在其中，但要想得到我们想要的波形，我们还需设置另一个文件，他就是wave.do文件；同样，波形文件与仿真工程置于同一个目录下，前面有“add wave -”标识，输出波形分两种：单个信号，逻辑0或逻辑1，用“Logic”表示；总线信号（数据或地址），用Literal表示；其他保持不变。

```
onerror {resume}                                                    //打印错误消息而不停止宏的执行
quietly WaveActivateNextPane {} 0                                   //创建第二个平面包含3个信号
add wave -noupdate -format Logic   /testcase/U_harness/vid_test/clk_1562khz_edge //添加单个信号：Logic
add wave -noupdate -format Literal /testcase/U_harness/vid_test/pin_dsp1_vcntl   //添加总线信号： Literal
TreeUpdate [SetDefaultTree]                                         //更新所有的波形
WaveRestoreCursors {{Cursor 1} {5215718933 ps} 0}                   //恢复在原来仿真中设置的所有的指针
configure wave -namecolwidth 145                                    //指定wave窗口中name栏的宽度
update
WaveRestoreZoom {0 ps} {7995531264 ps}                              // 恢复所设置的放大区域。
```

wave.do

制作激励文件——harness.v

激励文件也是用Verilog语言编写；一般情形下我们每做一次仿真都要写不同的激励文件；在制作过程中未防止编译出错，可将顶层文件中的所有不涉及的输入信号赋初值，我们在此处将后面要用到的单元都定义不同的task；然后在测试用例中对这些文件做调用。

```
`resetall                                   //复位所有的综合指示到其默认值
`define   UDLY  1                          //定义延时参数
`timescale 1ns/1ns                         //定义最小时间单位与时间精度

initial                                    //结构说明语句，在仿真开始立即执行，一般用于产生激励波形
begin                                      //1个模块中可以有多个initial块，他们都是并行执行
   reset        = 1'b1;
   pin_clk_100m  = 1'b0;
   pin_clk_125m  = 1'b0;
end

always  #5    pin_clk_100m = ~pin_clk_100m ;    //生成时钟。 #5为两次翻转的单位时间间隔，仿真开始立即执行

task Reset;                                //定义task。 用于Testcase调用时执行
begin
   reset    = 1'b1;
   #200     reset    = 1'b0;
end
endtask
```

harness.v

制作测试用例——testcase.v

测试用例的作用是调用激励文件中的模块，通过函数将testcase.do与harness.v关联起来，来达到我们仿真时只需修改testcase.do中的一句话而完成对不同测试项调用的目的，只需一次将激励文件做全，以后做不同仿真项时就不必频繁修改该文件；减少没次编译所花的时间与出错的概率，使仿真高效完成。

```
initial
Begin
all = $test$plusargs("ALL_TESTCASE");
   if(all||$test$plusargs("UT_TESTCASE_001"))                    //用来确认执行哪个测试用例
    begin
      $display("##############################");
      $display("UT_TESTCASE_001 Start!");                        //显示Testcase的执行情况
      U_harness.Vid_normal;                                      //Testcase和harness中的task关联
      $display("UT_TESTCASE_001 Finished!");
      $display("##############################");
end
end
```

testcase.v

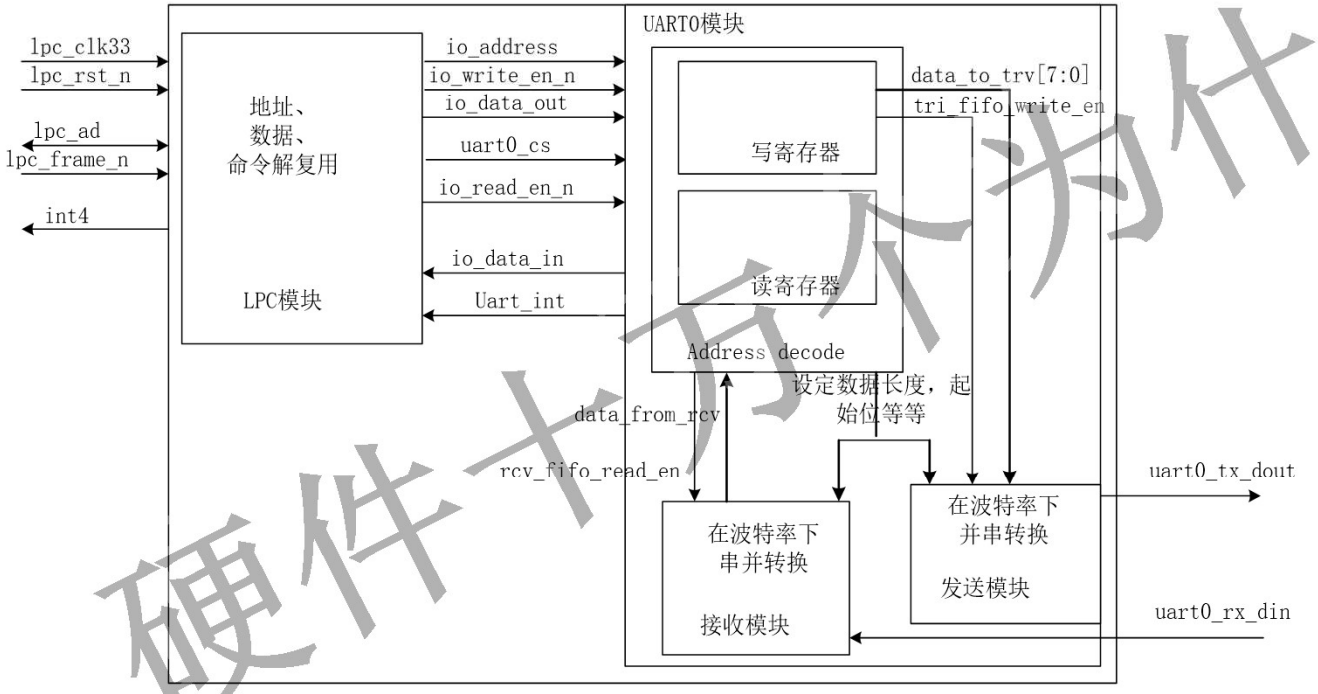
设计实例——LPC转串口

- LPC转串口模块包含两个二级模块，LPC模块将LPC总线转为localbus接口，串口模块uart_top包含三个模块，发送，接收和地址解码模块；实现LPC协议和UART协议的转换；

- --lpc_uart_top (LPC转UART模块)
- |--lpc (LPC控制器模块)
- |--uart_top (串口控制器模块)
- |--address_decode (地址分配模块)
- |--rcvandfifo (UART接收模块)
- |--fifo_lut_16bytes (接收FIFO)
- |--triandfifo (UART发送模块)
- |--fifo_16bytes (发送FIFO)

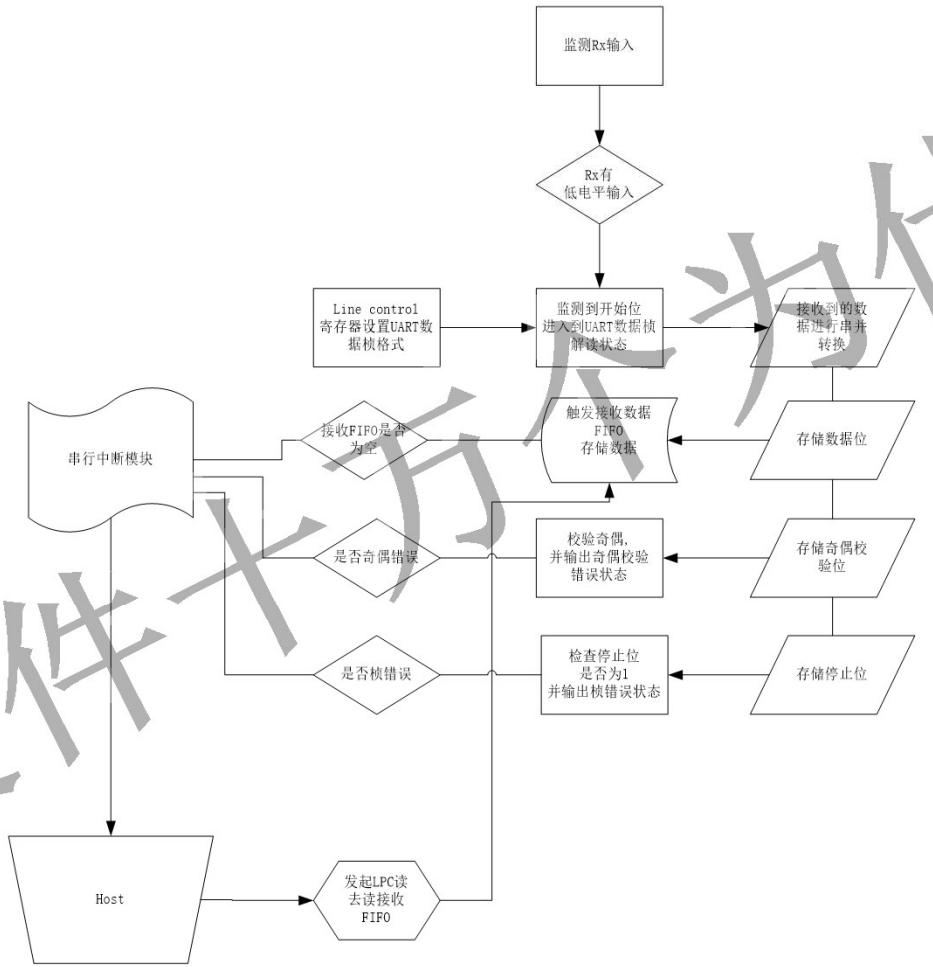
LPC转串口——总体设计方案框图

设计方案：

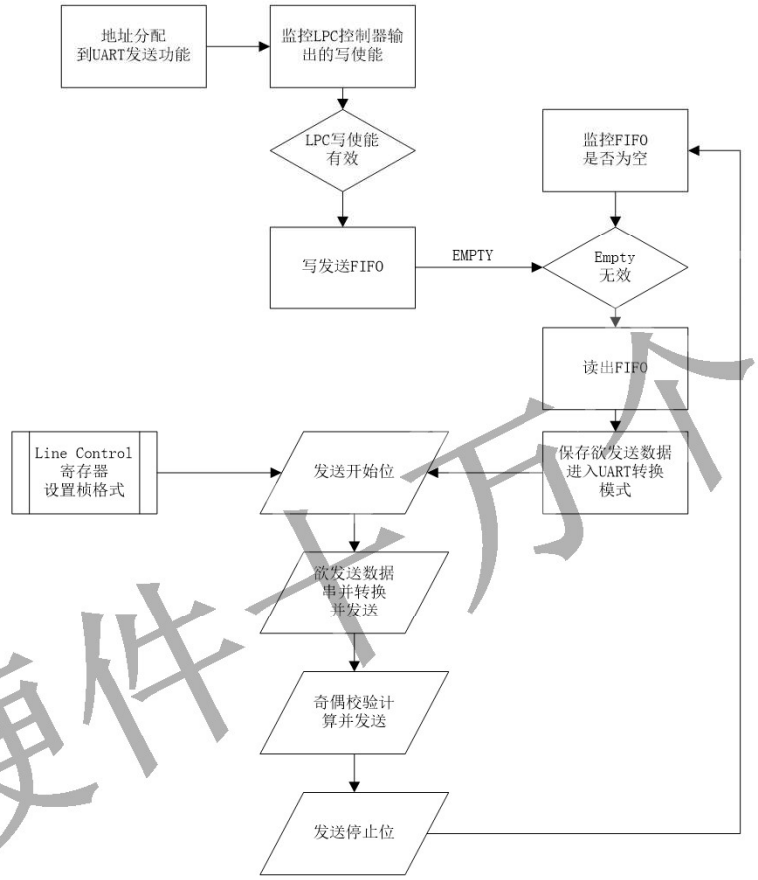


LPC转串口——串口接收模块原理框图

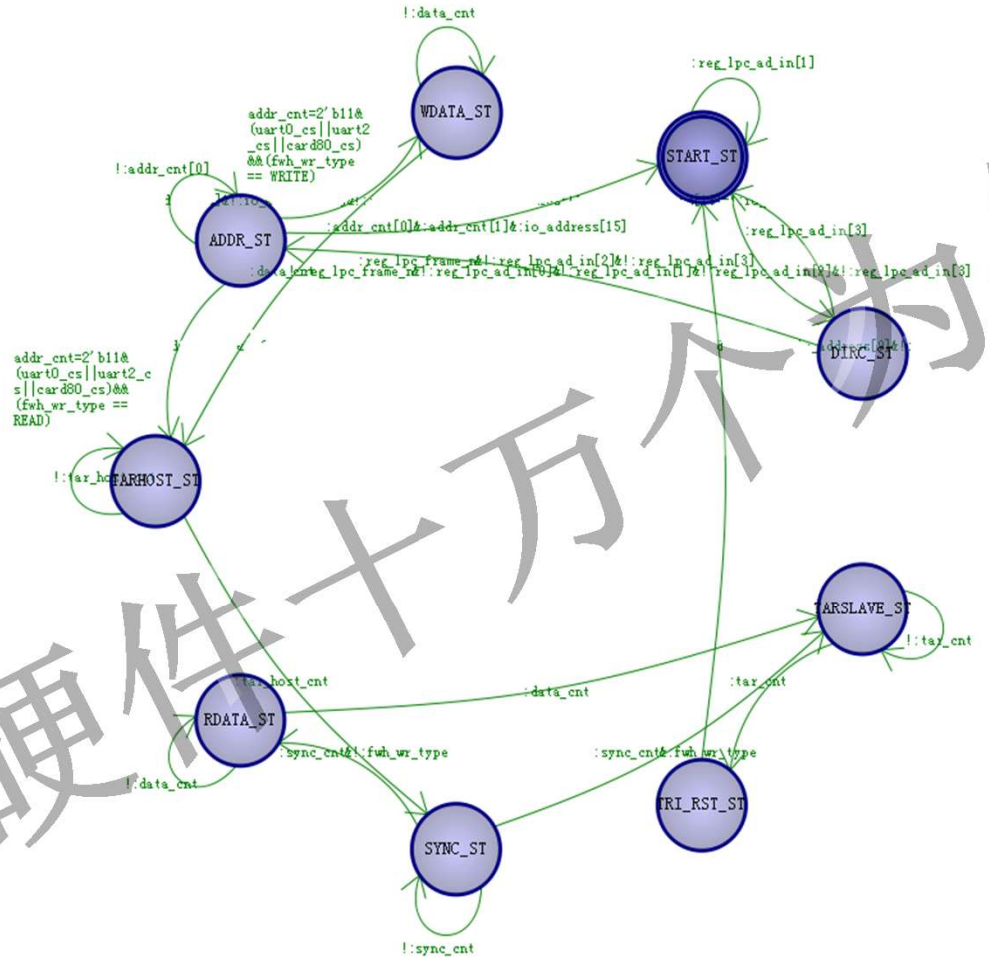
设计方案：



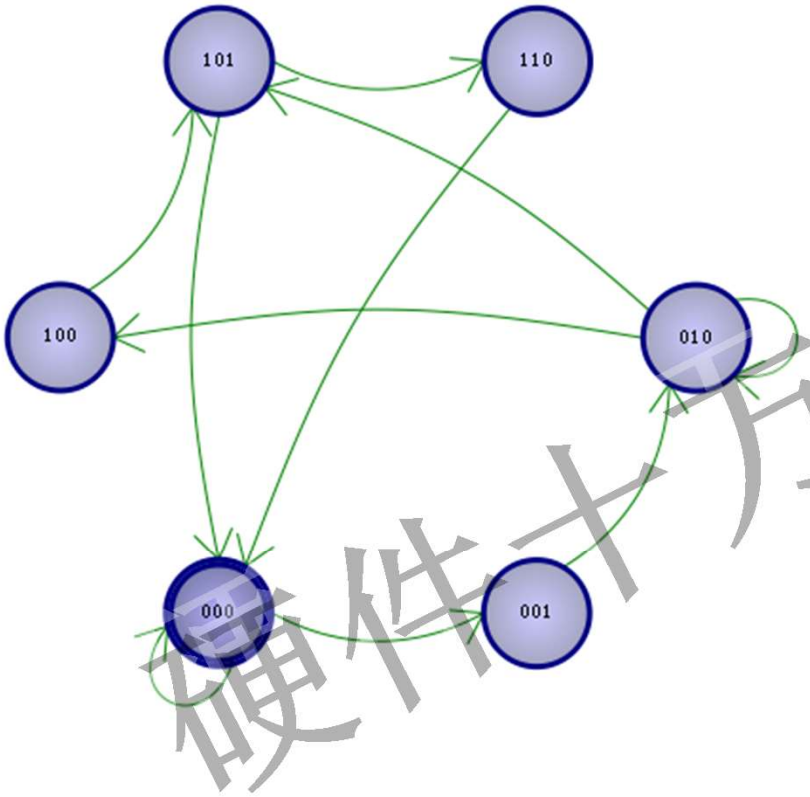
LPC转串口——串口发送模块原理框图



LPC转串口——LPC模块状态机切换



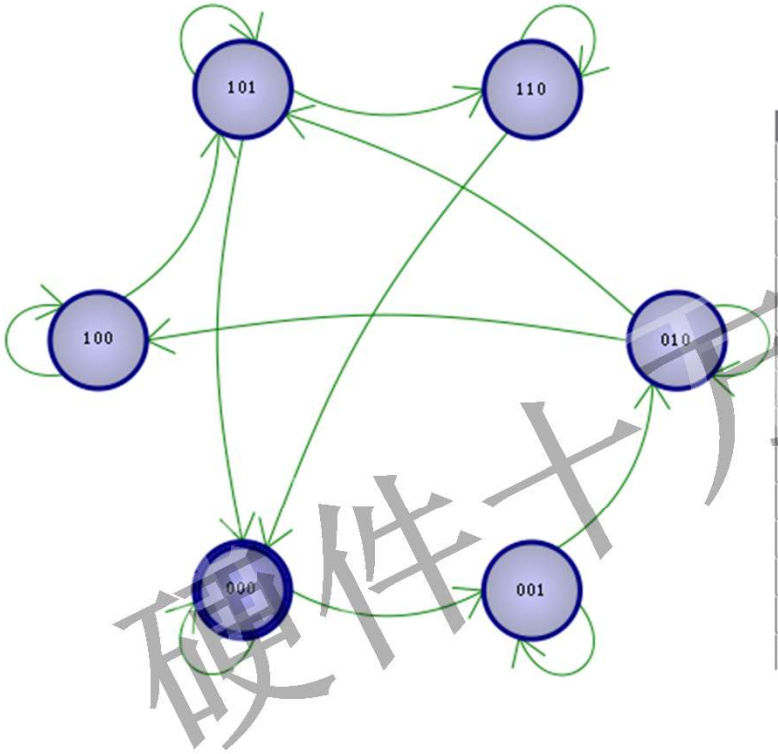
LPC转串口——串口发送模块状态机切换



	From State	To State	
1	101	110	:stop_length
2	100	101	
3	010	101	:state27&!:parity_en
4	010	100	:state27&:parity_en
5	010	010	!:state27
6	001	010	
7	000	001	!:empty_save
8	110	000	
9	101	000	!:stop_length
10	000	000	:empty_save

```
IDLE       = 0
START      = 1
SEND       = 2
STOP       = 3
PARITY_CHK = 4
STOP_BIT1  = 5
STOP_BIT2  = 6
```

LPC转串口——串口接收模块状态机切换

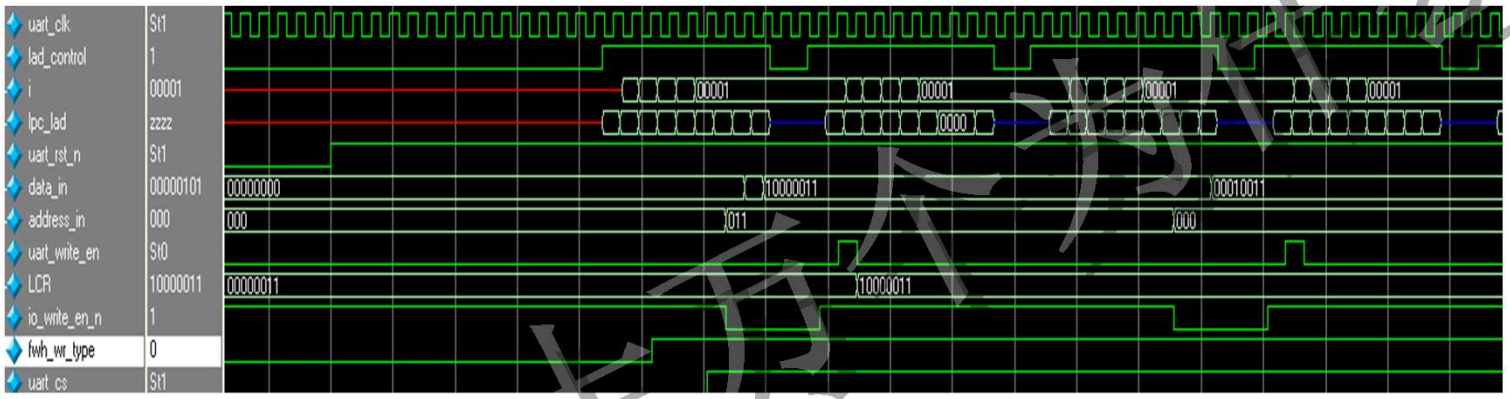


From State	To State	
110	110	!:clk_div_up
101	110	:stop_length&:clk_div_up
101	101	!:clk_div_up
100	101	:clk_div_up
010	101	:state33&!:parity_en
100	100	!:clk_div_up
010	100	:state33&:parity_en
010	010	!:state33
001	010	:clk_div_up
001	001	!:clk_div_up
000	001	:state10
110	000	:clk_div_up
101	000	!:stop_length&:clk_div_up
000	000	!:state10

```
IDLE       = 0
START      = 1
DATA       = 2
STOP       = 3
PARITY_CHK = 4
STOP_BIT1  = 5
STOP_BIT2  = 6
```

LPC转串口——功能仿真1

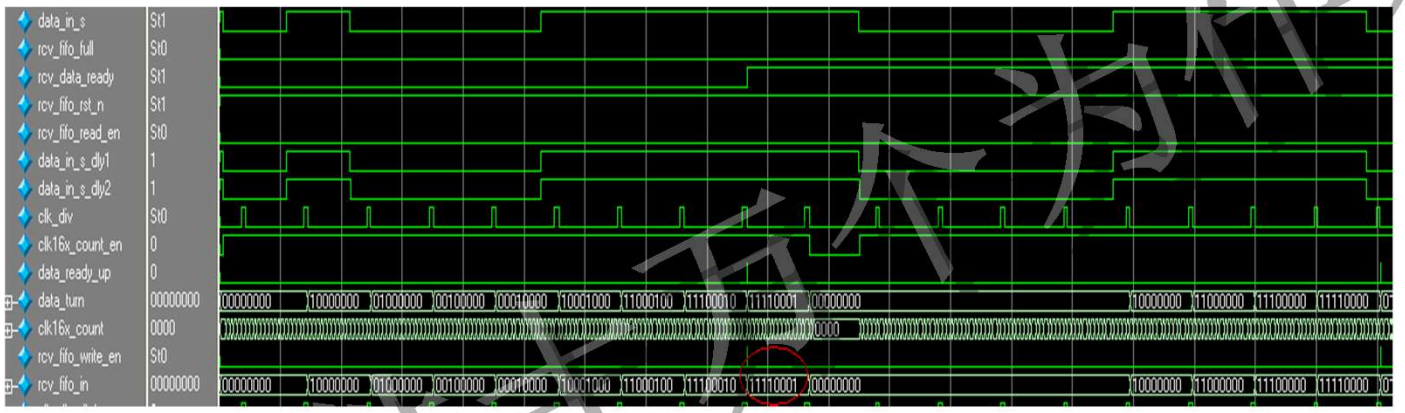
功能仿真1：测试 LPC 写寄存器；串口接收数据，从LPC总线上将接收数据读出来。



（1）LPC总线写LCR寄存器，正确

LPC转串口——功能仿真1

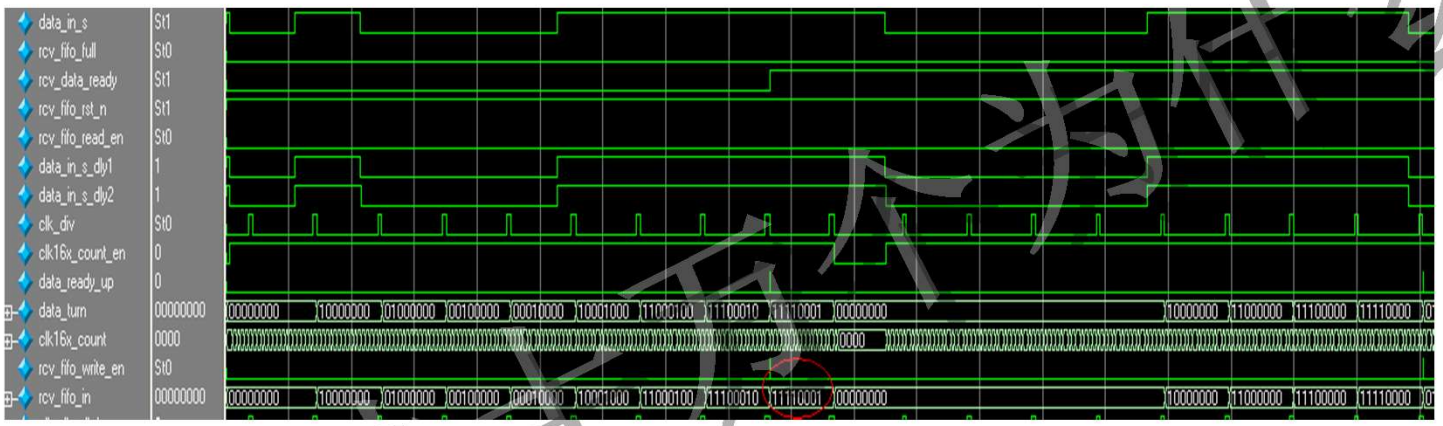
功能仿真1：测试 LPC 写寄存器；串口接收数据，从LPC总线上将接收数据读出来。



（2）波特率设置为 19200，能够正确写入数据到 fifo

LPC转串口——功能仿真1

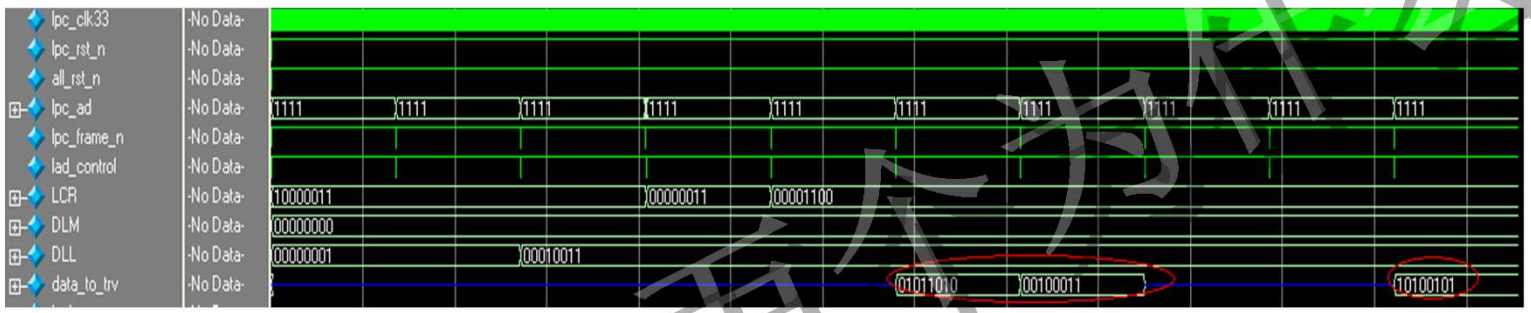
功能仿真1：测试 LPC 写寄存器；串口接收数据，从LPC总线上将接收数据读出来。



（3） LPC总线读SCR寄存器和接收缓存寄存器3F8,数据正确

LPC转串口——功能仿真2

功能仿真2：写串口寄存器，实现串口的低速发送

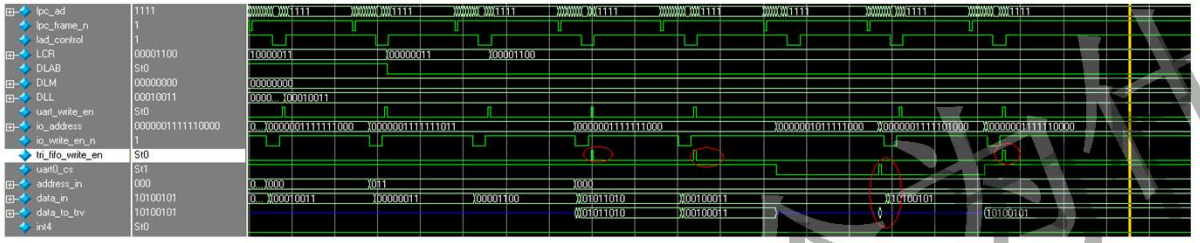


要发送的数据能正确发送，发送FIFO寄存器0x08,写到0x00 的数值也会在发送FIFO 寄存器里出现 0x08。

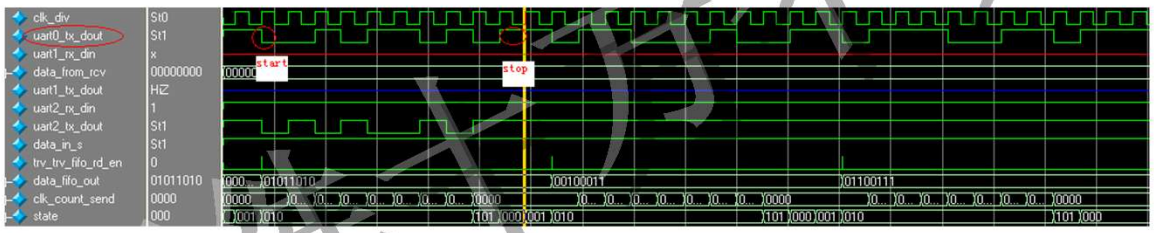
LPC转串口——功能仿真3

功能仿真3：快速发送数据

LPC总线上发送数据：



串口上出现数据：



异常：UART0_CS上有毛刺

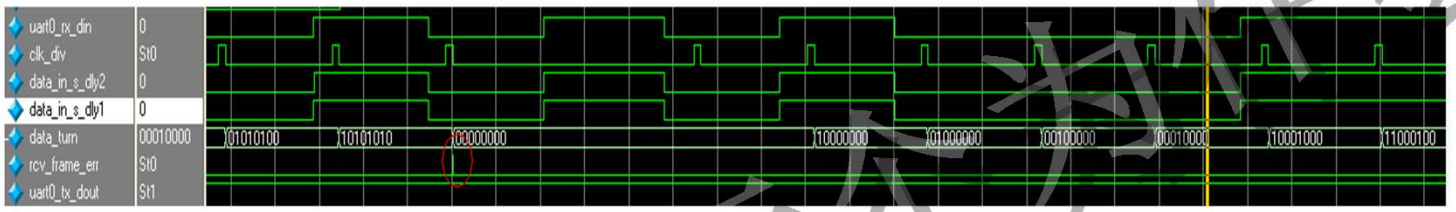
原因：和发送地址的数据构造有关，

```
U_harness.UART_Trf_high_Task ( 4'b0000,4'b0010,16'h02f8,8'h78 ) ;
//上一次发送地址2f8
U_harness.UART_Trf_high_Task ( 4'b0000,4'b0010,16'h03e8,8'ha5 ) ;
```

//下一次发送地址3e8,LPC总线上更新地址先更新高字节，导致达到短暂的3f8,UART0_CS有效，但这时写信号无效，无风险

LPC转串口——功能仿真4

功能仿真4：模拟帧错误，观察能否正确上报



接收一帧(8bit)数据后，又接收了一个bit的数据，能正确显示 帧错误。

硬十

硬件十万个为什么

THANK YOU!