

# Verilog HDL数字系统设计基础

实验与创新实践教育中心

# 主要内容

- 数字系统设计
- Verilog HDL基本知识
- Verilog HDL语法
- Verilog HDL数字系统设计
  
- 方法：
  - 讲义 ➡ 自学 ➡ 实践

# 参考书目

- Verilog数字系统设计教程（第3版）  
夏宇闻，北京航空航天大学出版社
- 搭建你的数字积木-数字电路与逻辑世界  
（VerilogHDL&Vivado版）  
汤勇明等，清华大学出版社
- Verilog HDL数字设计与综合（第2版）  
Sanir Palnitkar编 电子工业出版社

# 数字系统设计

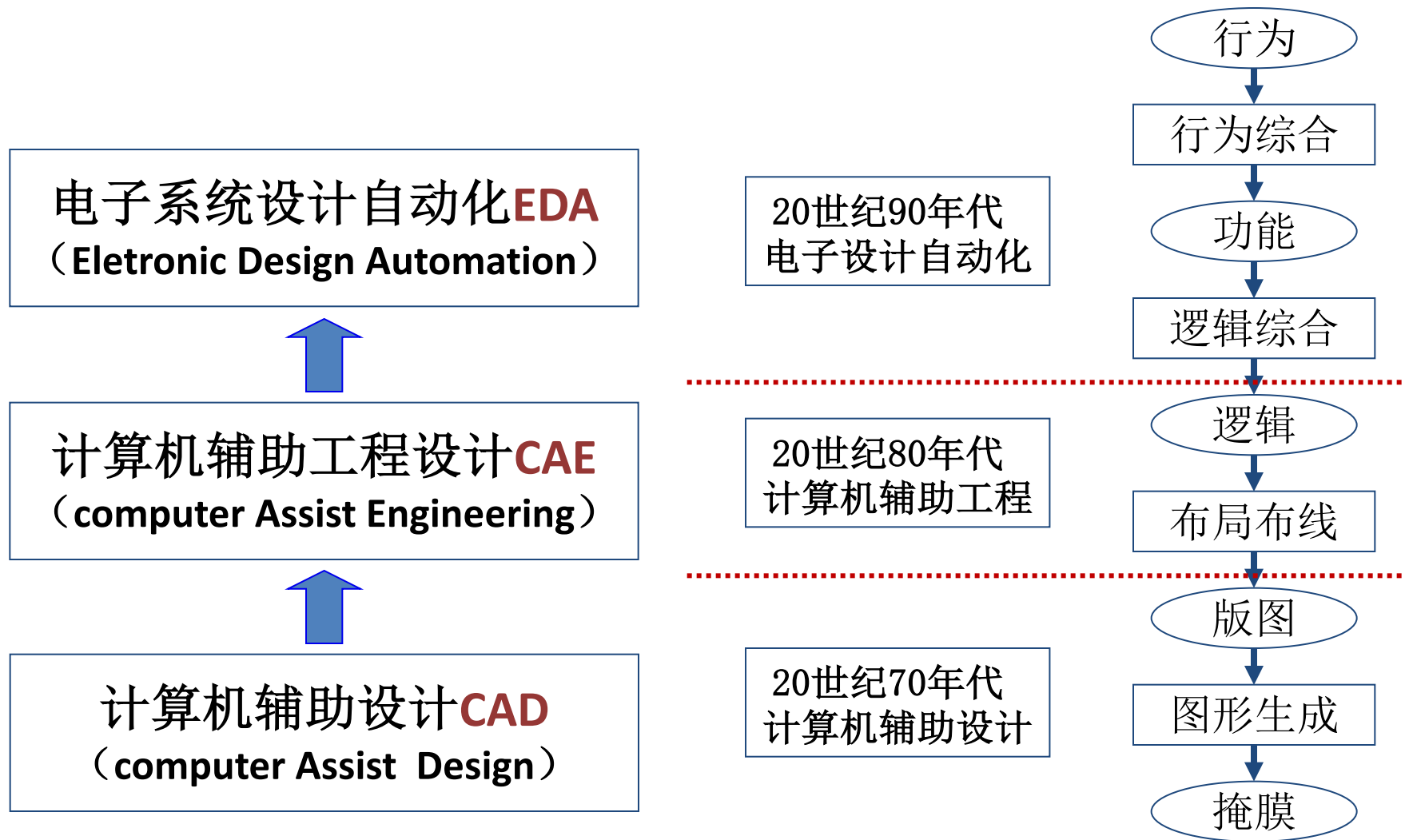
# 数字系统概念

- **数字系统**：是指对数字信息进行存储、传输、处理的电子系统。它的输入和输出都是数字量。
- 通常把门电路、触发器等称为**逻辑器件**；
- 将由逻辑器件构成，能执行某单一功能的电路，如计数器、译码器、加法器等，称为**逻辑功能部件**；
- 把由逻辑功能部件组成的能实现复杂功能的数字电路称**数字系统**。

# 复杂的数字系统

- 嵌入式微处理机系统
- 数字信号处理系统
- 高速并行计算逻辑
- 高速通信协议电路
- 高速编码/解码、加密/解密电路
- 复杂的多功能智能接口
- 门逻辑总数超过几万门达到几百甚至达几千万门的数字系统

# 数字系统设计的发展



# EDA技术

- **EDA (Electronic Design Automation)** 电子设计自动化
- 以计算机为工具，在EDA软件平台上，用硬件描述语言完成设计文件，然后由计算机自动地完成逻辑编译、化简、分割、综合、优化、布局、布线和仿真，直至对于特定目标芯片的适配编译、逻辑映射和编程下载等工作。
- 在可编程逻辑器件（如CPLD、FPGA）的应用
  - 设计者完成的事情：
    - 从概念、算法、协议等设计电子系统
  - 计算机完成的事情：
    - 电路设计、性能分析到设计出IC版图或PCB版图

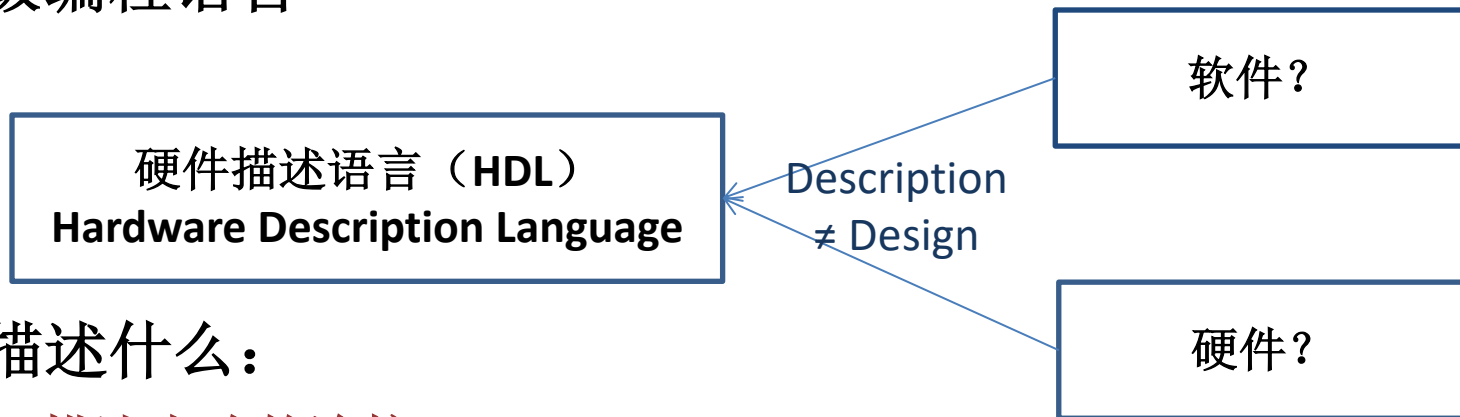


# Verilog HDL基本知识

# 什么是硬件描述语言

## ■ HDL(Hardware Description Language)

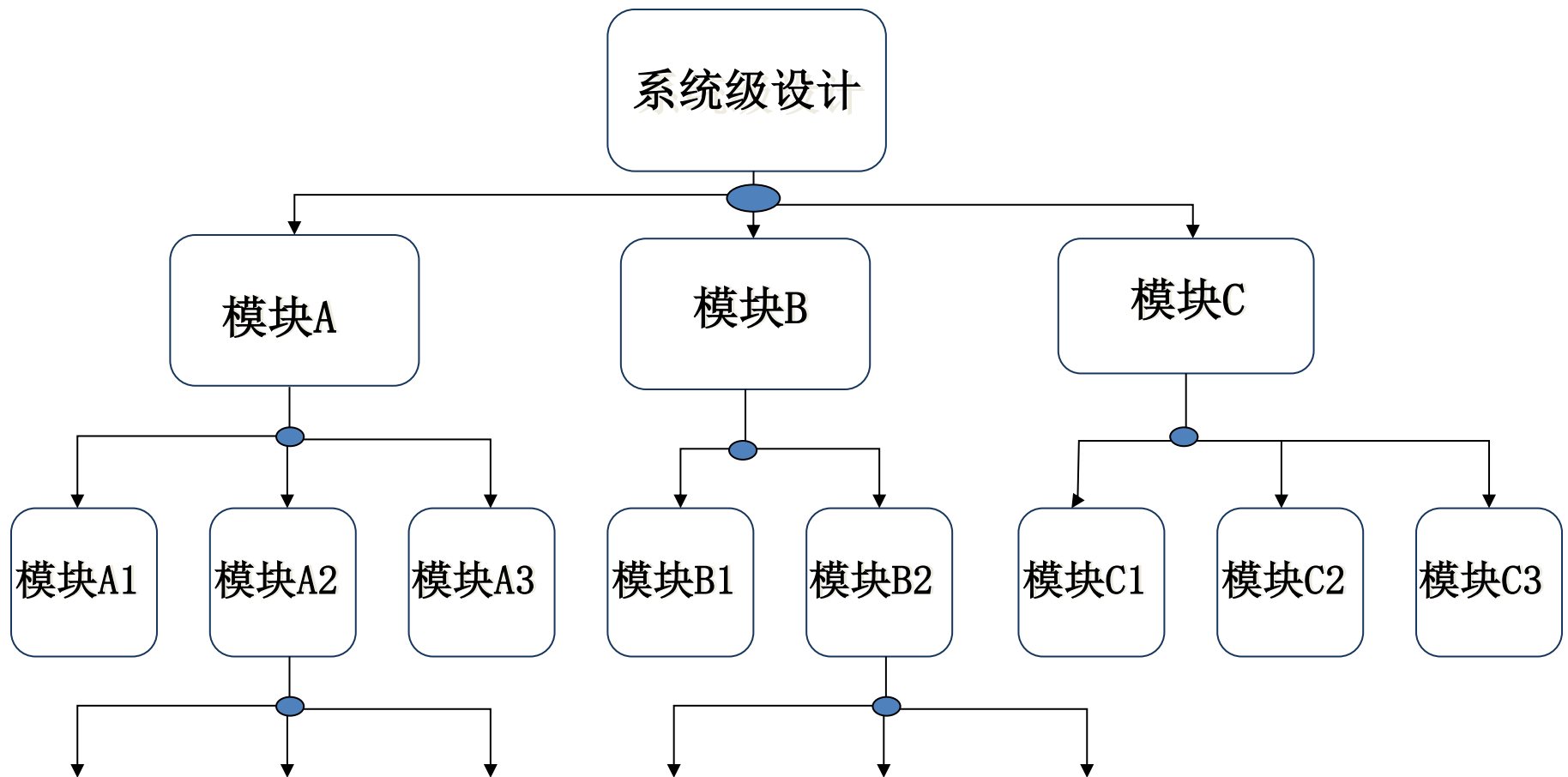
- 具有特殊结构能够对硬件逻辑电路的功能进行描述的一种高级编程语言



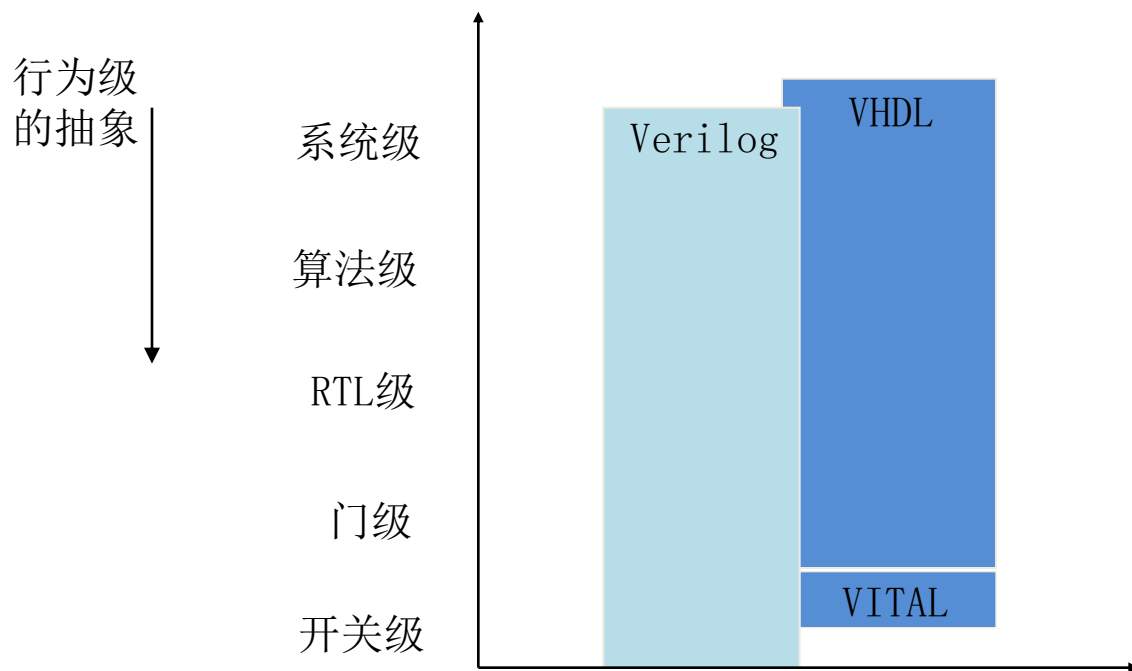
- 描述什么：
  - 描述电路的连接
  - 描述电路的功能
  - 在不同抽象级上描述电路
  - 描述电路的时序
  - 表达具有并行性

从语法结构，推想电路结构。  
从电路结构，总结语法结构。

# Top-Down设计思想



# VHDL vs. Verilog



Verilog 与 VHDL 建模能力的比较

**Verilog 应用较广泛、起步更容易！**

- VHDL:

- 起源于ADA语言
- 侧重于系统级描述，含有大量的内置数据类型和用户自定义类型

- Verilog:

- 起源于C语言
- 侧重于电路级描述，数据类型由语言本身定义，含有专门描述连线等的类型

# Verilog HDL的特点

- 语法结构上的主要特点：
  - 形式化地表示电路的行为和结构；
  - 从C编程语言中继承了多种操作符和结构；
  - 可在多个层次上对所设计的系统加以描述，语言对设计规模不加任何限制；
  - 具有混合建模能力：一个设计中的各子模块可用不同级别的抽象模型来描述；
  - 基本逻辑门、开关级结构模型均内置于语言中，可直接调用。
- 易学易用，功能强

# Verilog HDL的抽象级别

## • 行为描述语言&结构描述语言

### 行为级描述

系统级：

用高级语言结构实现设计模块的外部性能模型；

算法级：

用高级语言结构实现设计算法的模型；

RTL级（寄存器传输级）：

描述数据在寄存器之间流动和如何处理这些数据的模型；

侧重对模块**行为功能**的抽象描述

### 结构级描述

门级：

描述逻辑门以及逻辑门之间的连接的模型；

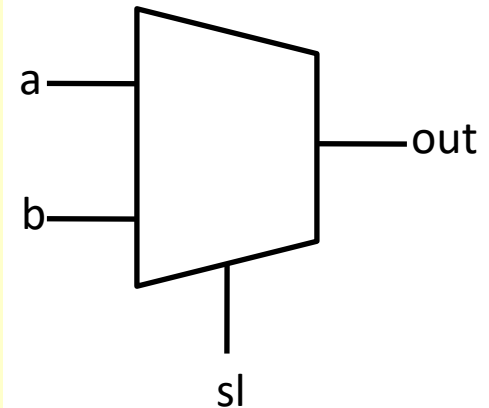
开关级：

描述器件中三极管和存储节点以及它们之间连接的模型。

侧重对模块**内部结构**实现的具体描述

# 行为级Verilog HDL

```
module muxtwo (out, a, b, sl); //二选一多路选择器
    input a, b, sl;           //输入信号名
    output out;                //输出信号名
    reg out;
    always @( sl or a or b)
        if (! sl) out = a;
        //控制信号sl为非, 输出与输入信号a一致
        else out = b;
        //控制信号sl为真, 输出与输入信号b一致
endmodule
```

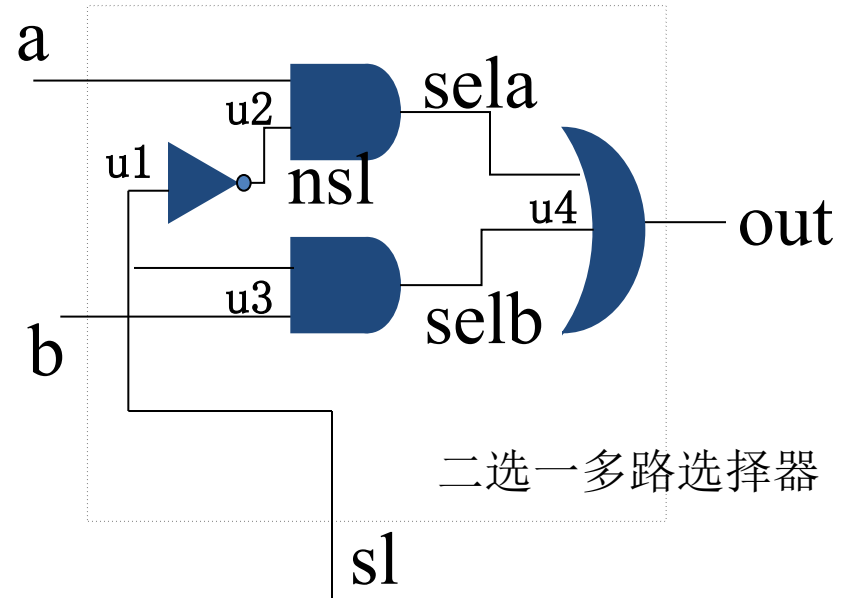


二选一多路选择器

- 在行为级模型中, 逻辑功能描述采用高级语言结构, 如@、while、if、case。
- RTL模块是可综合的, 它是行为模块的一个子集合。
- 行为级Verilog

# 结构级Verilog HDL

```
module muxtwo (out, a, b, s1); //二选一多路选择器
    input a, b, s1;
    output out;
    not u1 (ns1, s1 );
    //ns1=~s1
    and #1 u2 (sela, a, ns1);
    //sela=a&ns1
    and #1 u3 (selb, b, s1);
    //selb=b&s1
    or #2 u4 (out, sela, selb);
    //out=sela|selb
endmodule
```



- Verilog内部带有描述基本逻辑功能的基本单元(primitive), 如and门。
- 综合产生的结果网表通常是结构级的。
- 结构级Verilog适合开发小规模元件



# 学习Verilog HDL的要点

- 编写Verilog代码的目的是生成实际硬件电路。而电路，一般都不是串行执行的，很多时候都是并行工作的。所以在Verilog中，你一定要对电路图和电路的时序图有深刻的认识！
- 很多情况下，使用其他程序语言编写的代码越简洁越好。但是在Verilog中绝不是这样！衡量Verilog代码的唯一标准，就是在代码正确与清晰的前提下，可以生成结构尽可能简单、功能却非常强大的电路！
- 不是所有的Verilog代码都能够转换成实际电路的，学习语法时要分辨清楚。那些可以转换成实际电路的，我们称为“可综合”！另外，即使你使用的可综合的代码去编写，如果你描述的电路实际上无法实现，也是无法综合的！

# Verilog HDL与C语言

- Verilog 有许多语法规则与 C 语言一致。
- 但与 C 语言有根本的区别：
- 时序
- 并行性
- 块的含义： always块 和 initial块
- 两种赋值语句：阻塞赋值 “=”
- 非阻塞赋值 “<=”
- .....

硬件描述语言

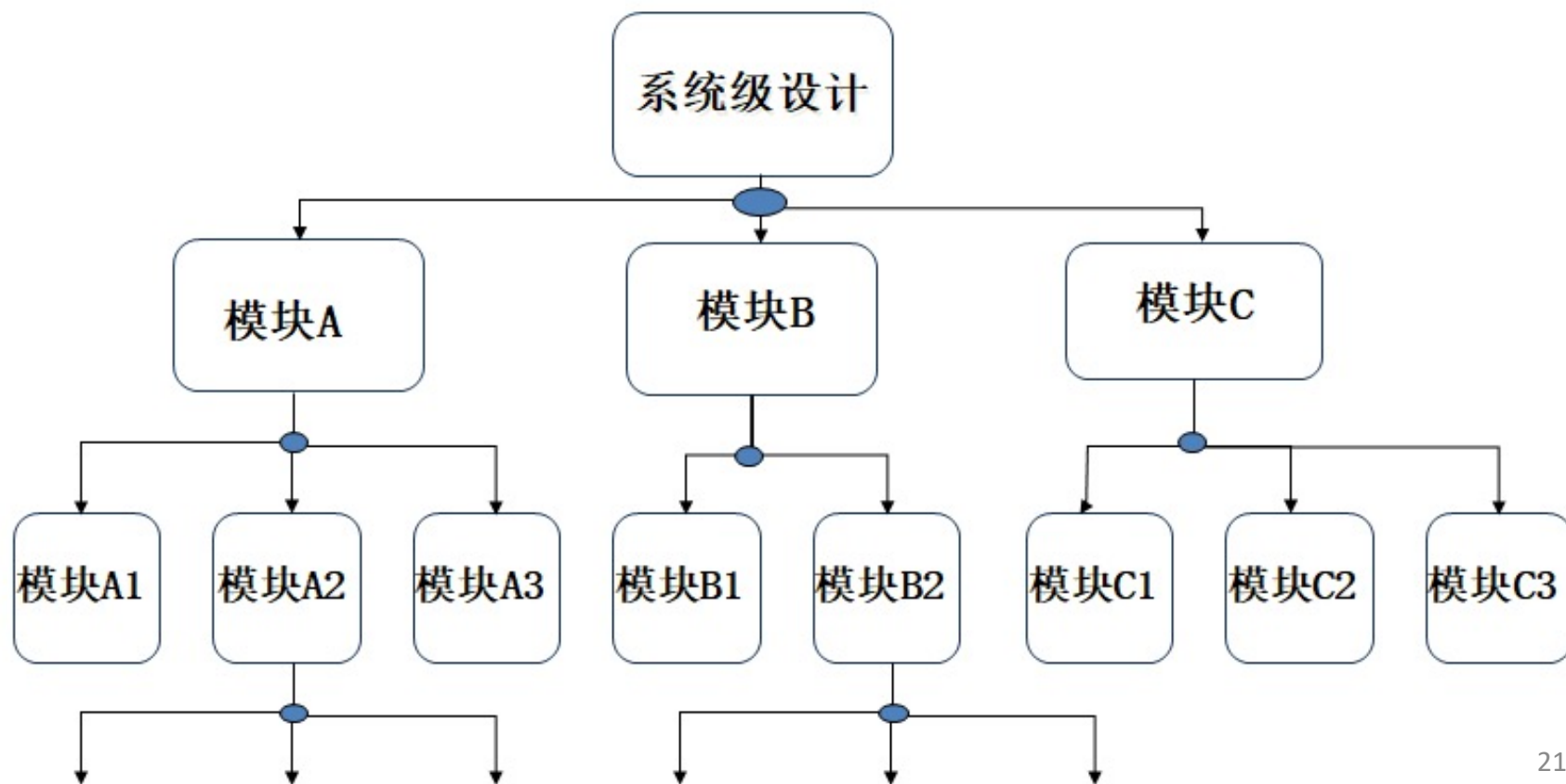
# Verilog HDL语法

# Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# Verilog 的模块

- Verilog 的基本设计单元是“**模块**”，模块的实际意义是代表硬件电路上的逻辑实体，模块之间是并行运行的。



# 模块基本结构

模块声明:

```
module module_name (port_list) ; //模块名 (端口声明列表)
```

端口定义:

```
input[信号位宽];    //输入声明
output [信号位宽];  //输出声明
...
```

数据类型说明:

```
reg [信号位宽];      //寄存器类型声明
wire [信号位宽];     //线网类型声明
parameter;           //参数声明
...
```

功能描述: //主程序代码

```
assign a=b+c
always@(posedge clk or negedge reset)
function
...
endmodule
```

# 模块声明

**module** module\_name (port\_list) ; //模块名（端口声明列表）

**module** muxtwo (out, a, b, sl);

```
module muxtwo (out, a, b, sl); //二选一
    多路选择器
    input a, b, sl;
    output out;
    reg out;
    always @( sl or a or b)
        if (! sl) out = a;
        else out = b;
endmodule
```

- “模块名”是模块唯一的标识符，区分大小写。
- “端口列表”是由模块各个输入、输出和双向端口组成的列表。(input,output,inout)
- 端口用来与其它模块进行连接，括号中的列表以“,”来区分，列表的顺序没有规定，先后自由。

# 端口定义

input[信号位宽]

output [信号位宽]

inout[信号位宽]

input a, b, sl; //输入信号名  
output out; //输出信号名  
endmodule

致

致

endmodule

```
module muxtwo (out, a, b, sl); //二选一多路选择器
    input a, b, sl; //输入信号名
    output out; //输出信号名
    reg out;

    always @( sl or a or b)
        if (! sl) out = a;
        //控制信号sl为非, 输出与输入信号a一致

    else out = b;
    //控制信号sl为真, 输出与输入信号b一致
endmodule
```

- 输入端口：模块从外界读取数据的接口，是连线类型
- 输出端口：模块向外界传输数据的接口，是连线或寄存器型
- 输入输出端口：可读取数据也可接收数据的端口，数据是双向的，是连线型
- 端口定义也可以写在端口声明的位置：

```
module module_name(input port1,input port2,...output port1,...);
```



# 数据类型说

```
reg [信号位宽];  
wire [信号位宽]  
parameter;
```

```
reg out; //输出
```

```
// reg [3:0] a4,b4,c4;
```

```
parameter[3:0] s0=4'h0,s1=4'h5....
```

```
module muxtwo (out, a, b, s1); //二选一多路选择器  
    input a, b, s1; //输入信号名  
    output out; //输出信号名  
    reg out;  
    always @( s1 or a or b)  
        if (! s1) out = a;  
        //控制信号s1为非, 输出与输入信号a一致  
        else out = b;  
        //控制信号s1为真, 输出与输入信号b一致  
endmodule
```

- 模块中用到的所有信号都必须进行数据类型的定义。
- 声明变量的数据类型后, 不能再进行更改
- 在VerilogHDL中只要在使用前声明即可
- 声明后的变量、参数不能再次重新声明
- 声明后的数据使用时的配对数据必须和声明的数据类型一致

# 模块基本结构

模块声明:

```
module module_name (port_list) ; //模块名 (端口声明列表)
```

端口定义:

```
input[信号位宽];    //输入声明
output [信号位宽];  //输出声明
...
```

数据类型说明:

```
reg [信号位宽];    //寄存器类型声明
wire [信号位宽];   //线网类型声明
parameter;         //参数声明
...
```

功能描述:                      //主程序代码

```
assign  a=b+c
always@(posedge clk or negedge reset)
function
...
endmodule
```

# Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# 标识符

- 赋给对象的唯一名称，可以是字母、数字、下划线和符号“\$”的组合，且首字符只能是字母或下划线。
- 大小写敏感。
- 关键词用小写字母定义，如：always,and,assign等
- 注释有两种：
  - 以“/\*”开头，以“\*/”结束。
  - 以“//”开头到本行结束。

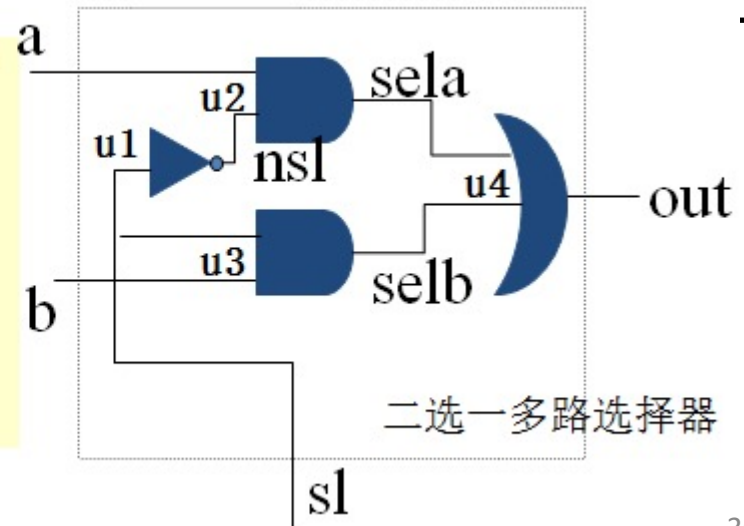
# 数据类型

- 共有**19**种数据类型，分为物理数据类型和抽象数据类型
- 物理数据类型：
  - **线网型wire**、**寄存器型reg**、存储器型memory 等
- 抽象数据类型：
  - **参数型parameter**、整型integer、时间型time、实型real等
- 物理数据类型的抽象数据程度比较低，与实际硬件电路的映射关系明显。
- 抽象数据类型是用于进行辅助设计和验证的数据类型。
- 数据类型分为常量和变量，分别属于以上类型。
  - 常量：数字、参数型parameter
  - 变量：连线型wire、寄存器型reg、存储器型memory 等

# 线网类型（wire）

- 硬件电路中元件之间实际连线的抽象。（如器件的管脚，内部器件如与非门的输出等）。
- 不存储逻辑值，必须由器件驱动。通常由assign 进行赋值。
- 如 assign  $Y = \sim (A \& C)$  ;

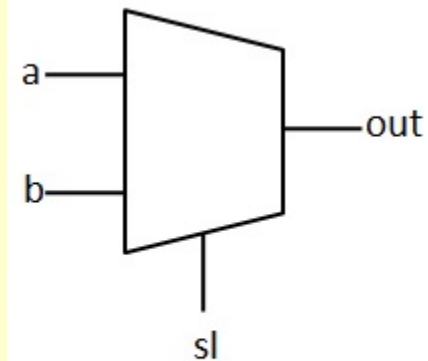
```
module muxtwo (out, a, b, sl); //二选一多路选择器
input a, b, sl;
output out;
    not u1 (nsl, sl);           //nsl=~sl
    and #1 u2 (sela, a, nsl);   //sela=a&nsl
    and #1 u3 (selb, b, sl);    //selb=b&sl
    or #2 u4 (out, sela, selb); //out=sela|selb
endmodule
```



# 寄存器类型（reg）

- reg型的变量具有状态保存的作用。综合后常常是寄存器或触发器的输出，但不一定总是这样。
- 在过程块“always”块内被赋值的每一个信号都必须定义成reg型。
- reg型的变量只能在initial或always过程语句的内部被赋值。

```
module muxtwo (out, a, b, sl); //二选一多路选择器
    input a, b, sl; //输入信号名
    output out; //输出信号名
    reg out;
    always @( sl or a or b)
        if (! sl) out = a;
            //控制信号sl为非，输出与输入信号a一致
        else out = b;
            //控制信号sl为非，输出与输入信号b一致
endmodule
```



二选一多路选择器

# 参数型 ( parameter )

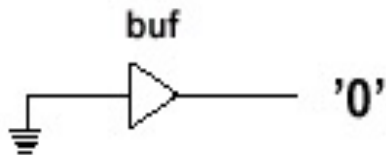
- 定义格式: `parameter 参数名1=表达式1, ..., 参数名n=表达式n;`
  - 其中, 表达式既可以是常数, 也可以是表达式。参数定义完以后, 程序中出现的所有的参数名都将被替换为相对应的表达式。
- `parameter length=32,weight=16;`//定义了两个参数
- 属于常量, 常用来定义延迟时间和变量的位宽。
- 在模块或实例引用时, 可通过参数传递改变在被引用模块或实例中已定义的参数。



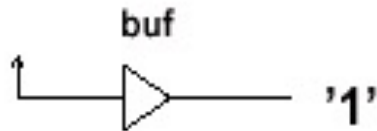
# 数字

- 整数：
  - 二进制（b或B）、十进制（d或D）、十六进制（h或H）、八进制（o或O）
  - 表达方式：
    - <位宽><进制><数字>，这是一种全面的描述方式
    - 在<进制><数字>这种描述方式中，数字的尾款采用默认位宽（这又具体的机器系统决定，但至少32位）
    - 在<数字>这种描述方式中，采用默认进制（十进制）
    - 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的数的十六进制表示，'h表示十六进制
- 负数
  - 在位宽表达式前加一个减号，减号必须写在数字定义表达式的最前面。
    - -8'd5//5的补码

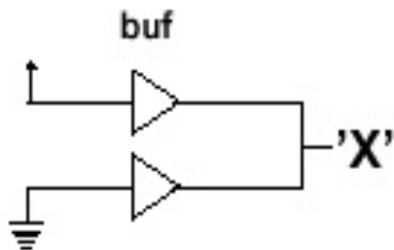
# 四种逻辑值



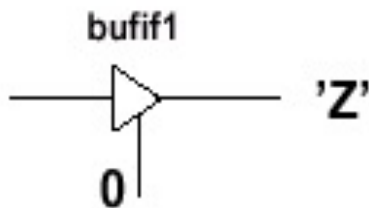
低电平、逻辑0、“假”、接地



高电平，逻辑1、“真”



不确定或未知的逻辑状态



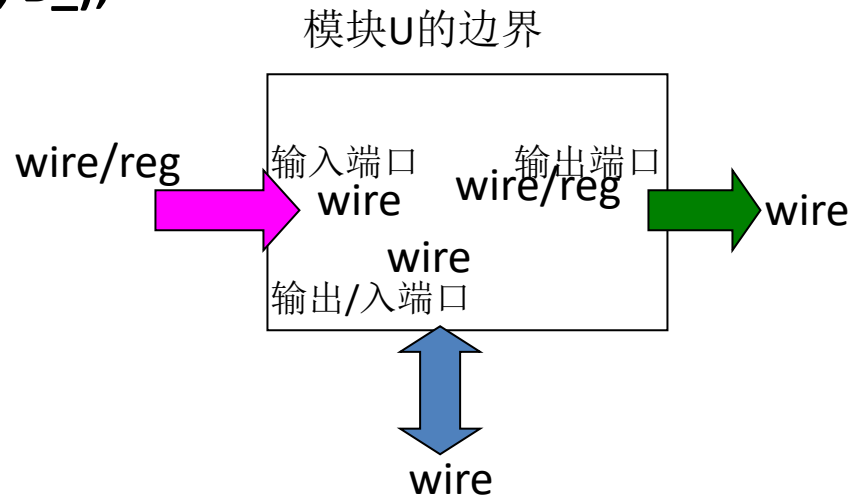
高阻态

# 如何选择正确的数据类型？

- 输入端口（**input**）：可以由寄存器或线网连接驱动，但它本身只能驱动线网连接。
- 输出端口（**output**）：可以由寄存器或线网连接驱动，但它本身只能驱动线网连接。
- 输入/输出端口（**in/out**）：只可以由线网连接驱动，它本身只能驱动线网连接。
- 如果信号变量是在过程块（**initial**块 或 **always**块）中被赋值的，必须把它声明为寄存器类型变量

# 如何选择正确的数据类型？

```
module U(Y, A, B_);  
output Y;  
input A,B;  
wire Y, A, B;  
and (Y, A, B);  
endmodule
```



```
module top;  
wire y;  
reg a, b;  
U u1(y,a,b);  
initial  
begin  
a = 0; b = 0;  
#10 a =1; ....  
end  
endmodule
```

# Verilog HDL语法

- 模块的结构与实例化
- 标识符和数据类型
- **运算符及表达式**
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# 运算符

- 按功能分为以下几类：

- ① 算术运算符+，-，\*，/，%
- ② 逻辑运算符：&&，||，!
- ③ 位运算符：~，|，^，&，^~
- ④ 赋值运算符：=，<=
- ⑤ 关系运算符：>，<，>=，<=
- ⑥ 条件运算符：?:
- ⑦ 移位运算符：<<,>>
- ⑧ 拼接运算符：{}
- ⑨ 等式运算符：==，!=，===，!==
- ⑩ 其他

# 运算符

- 算术运算符：+，-，\*，/，%
  - 在进行整数的除法运算时，结果要略去小数部分，只取整数部分；而进行取模运算时（%，亦称作求余运算符）结果的符号位采用模运算符中第一个操作数的符号。
  - 在进行算术运算时，如果某一个操作数有不确定的值x，则整个结果也为不确定值x。
- 逻辑运算符：&&，||，!
  - 其中&&和||是双目运算符，其优先级别低于关系运算符，而！高于算术运算符。
- 位运算符：~，|，^，&，^~
  - 在不同长度的数据进行位运算时，系统会自动的将两个数右端对齐，位数少的操作数会在相应的高位补0，两个操作数按位进行操作。
  - 缩减运算 `wire[3:0] a; wire y;`  
`assign y=a[3]|a[2]|a[1]|a[0]`

# 运算符

- 关系运算符：  $>$  ,  $<$  ,  $>=$  ,  $<=$ 
  - 如果关系运算是假的，则返回值是0，如果关系是真的，则返回值是1。
  - 关系运算符的优先级别低于算数运算符。如：  $a < \text{size}-1$  等同于  $a < (\text{size}-1)$
  - 如果某个操作数值不定，则关系是模糊的，返回值是不定值。
- 移位运算符：  $<<$  ,  $>>$ 
  - $a >> n$  其中  $a$  代表要进行移位的操作数，  $n$  代表要移几位。这两种移位运算都用0来填补移出的空位。如果操作数已经定义了位宽，则进行移位后操作数改变，但是其位宽不变。
- 拼接运算符：  $\{ \}$ 
  - {信号1的某几位， 信号2的某几位， .....信号n的某几位} 将某些信号的某些为列出来，中间用逗号分开，最后用大括号括起来表示一个整体的信号。在位拼接的表达式中不允许存在没有指明位数的信号。

wire [7:0] b8,c8,d8

assign d8={b8[3:0],c8[7:6],c8[1:0]}




# 运算符

- 等式运算符：==, !=, ===, !==
  - ==, !=: X和Z进行比较时为X
  - ===, !==: 操作数相同结果为1, 常用于case表达式的判别。

===	0	1	x	z		==	0	1	x	z
0	1	0	0	0		0	1	0	x	x
1	0	1	0	0		1	0	1	x	x
x	0	0	1	0		x	x	x	x	x
z	0	0	0	1		z	x	x	x	x

# 运算符优先级别表

优 先 级 别	
<div>! ~</div> <div>* / %</div> <div>+ -</div> <div>&lt;&lt; &gt;&gt;</div> <div>&lt; &lt;= &gt; &gt;=</div> <div>== != === !==</div> <div>&amp;</div> <div>^ ^~</div> <div> </div> <div>&amp;&amp;</div> <div>  </div> <div>?:</div>	<div>高 优 先 级 别</div> <div></div> <div>低 优 先 级 别</div>

# 逻辑门的描述

	Verilog描述	逻辑表达式
与门	$F = A \& B;$	$F=AB$
或门	$F = A \mid B;$	$F=A+B$
非门	$F = \sim A;$	$F=A'$
与非门	$F = \sim (A \& B);$	$F=(AB)'$
或非门	$F = \sim (A \mid B);$	$F=(A+B)'$
与或非门	$F = \sim((A \& B) \mid (C \& D));$	$F=(AB+CD)'$
异或门	$F = (A \wedge B);$ $F = (\sim A \& B) \mid (A \& \sim B);$	$F=A \oplus B$ $F=A'B+AB'$
同或门	$F = (A \sim \wedge B);$ $F = \sim (A \wedge B);$ $F = (\sim A \& \sim B) \mid (A \& B);$	$F=(A \oplus B)'$ $F=A'B'+AB$

# Verilog HDL语法

- 模块的结构与实例化
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# 赋值语句

- 在Verilog中，变量是不能随意赋值的，需要使用连续赋值语句和过程赋值语句。
- assign称为**连续赋值**，对应于wire类型变量；
- initial或always称为**过程赋值**，对应于寄存器类型变量reg。

# 连续赋值语句

```
wire a;
```

```
assign a=1'b1;
```

- 语法格式： `assign` 线网型变量名=赋值表达式;
- 等号右端赋值表达式的值会持续对被赋值变量产生连续驱动，而且只要等号右端赋值表达式的值改变，左端被赋值变量的值就会立即改变;
- 对应到电路，就是导线。

# 连续赋值语句assign

- 左侧数据类型必须是线网型数据（**wire**）；
- 所有右值都是敏感信号，右侧任何信号的变化都会激活该语句，使其被立即执行一次；
- 每条**assign**赋值语句相当于一个逻辑单元，等价于门级描述；
- 各个**assign**赋值语句之间是并发的关系；
- 在过程块（**initial/always**）外面；
- 描述组合电路。

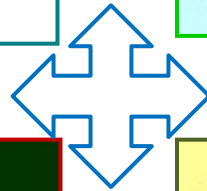
# 连续赋值语句

## ●变量（标量）

```
wire a , b ;  
assign a = b ;
```

## ●向量中某一位

```
wire [7:0] a , b ;  
assign a[3] = b[3] ;
```



## ●向量

```
wire [7:0] a , b ;  
assign a = b ;
```

## ●向量中某几位

```
wire [7:0] a , b ;  
assign a[3:2] = b[3:2] ;
```



# 过程赋值语句

- 例子:

```
reg a,c;
```

```
always@(a) begin
```

```
    c<=c+a;
```

```
end
```

```
reg c;
```

```
initial begin
```

```
    c=1'b0;
```

```
end
```

- 在always/initial块中使用过程赋值语句;
- always不断执行, 即每一次a的值改变时, c都会被重新赋值。
- initial只会执行一次, 即只执行一次把c赋零的操作;

# 过程赋值语句

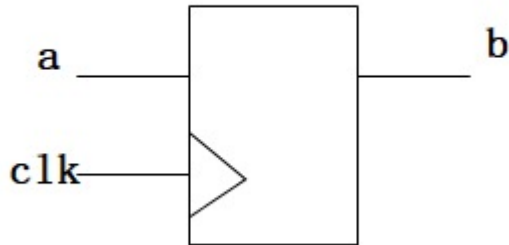
- 只能出现在过程块中（always/initial），主要描述时序电路；
- 过程赋值语句中：没有关键词“assign”；
- 左侧数据类型必须是reg类型的变量；
- 包括阻塞赋值（运算符=）和非阻塞赋值（运算符<=）

# 阻塞赋值与非阻塞赋值

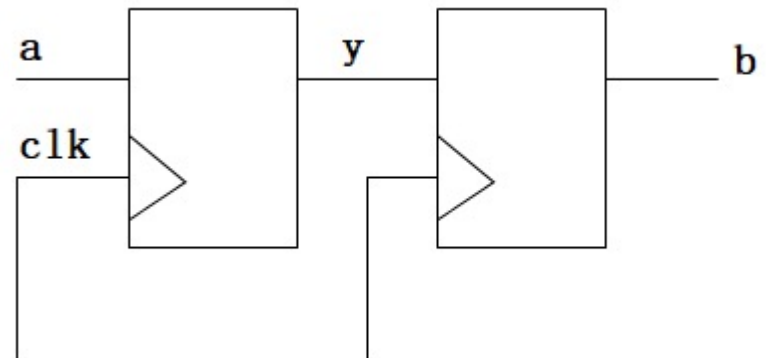
- 阻塞（**Blocking**）赋值方式（如 **$b = a$  ;**）
  - 赋值语句执行完后，块才结束；
  - **b**的值在赋值语句执行完后立刻就改变；
  - 可能产生意想不到的结果。
  - 组合电路中常用
- 非阻塞（**Non\_Blocking**）赋值方式（如 **$b <= a$ ;**）
  - 块结束后才完成赋值操作；
  - **b**的值并不是立刻就改变；
  - 时序电路中常用的赋值方法。（特别在编写可综合模块时）

# 阻塞赋值与非阻塞赋值

```
module bloc(clk, a, b);  
input clk, a;  
output b; reg b;  
reg y;  
always @(posedge clk)  
begin  
    y=a;  
    b=y;  
end  
endmodule
```



```
module nonbloc(clk, a, b);  
input clk, a;  
output b; reg b;  
reg y;  
always @(posedge clk)  
begin  
    y<=a;  
    b<=y;  
end  
endmodule
```



# 非阻塞赋值

块内的赋值语句同时进行：先同时采样，最后一起更新

## ■ 非阻塞赋值

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

结果：  
a=2, b=5  
c=3, d=2

结果与书写的顺序无关  
(原因：同步更新)

时序电路特点：输出不会随输入变化而立即变化

初值：  
a=5, b=3  
c=10, d=2

## ■ 非阻塞赋值

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

结果：  
a=2, b=5  
c=3, d=2

本质上，在一个时钟沿触发里，a得到d的值，但b得到的永远是a的旧值，c得到的永远是b的旧值（原因：同步更新）。

# 如何区分阻塞赋值与非阻塞赋值？

- 时序逻辑
  - 一定用非阻塞赋值 “<=”, 只要看到敏感列表有posedge就用 “<=”。
- 组合逻辑
  - 一定用 “=”, 只要敏感列表没有posedge就用 “=”。
- 时序逻辑和组合逻辑分成不同的模块
  - 即一个always模块里面只能出现非阻塞赋值 “<=” 或者 “=”。

# 连续赋值与过程赋值的比较

	过程赋值	连续赋值
assign	无assign (过程性连续赋值除外)	有assign
符号	使用 “=”， “<=”	只使用 “=”
位置	在always语句或initial语句中均可出现	不可出现于always语句和initial语句
执行条件	与周围其他语句有关	等号右端操作数的值发生变化时
用途	驱动寄存器	驱动线网

# 过程块

- 过程块是行为模型的基础。
- 过程块有两种：
  - initial块
    - 只能执行一次。
    - Initial语句不带触发条件。
    - 它通常用于仿真模块。
  - always块
    - 循环执行。
    - always语句通常带触发条件，满足触发条件则执行。
    - 一个模块中有多个always块时，可以并行进行。

```
reg c;  
initial begin  
    c=1'b0;  
end
```

```
reg a,c;  
always@(a) begin  
    c<=c+a;  
end
```



# initial块

- 格式如下：

Initial

begin

语句1;

语句2;

.....

语句n;

end

initial

begin

#20 begin a = 0;b = 0; cin= 1;end

#20 begin a = 0;b = 1; cin= 0;end

#20 begin a = 0;b = 1; cin= 1;end

#20 begin a = 1;b = 0; cin= 0;end

#20 begin a = 1;b = 0; cin= 1;end

#20 begin a = 1;b = 1; cin= 0;end

end

- begin\_end为顺序块，用来标识顺序执行的语句。

# always块

- 格式如下：

```
always @ (<敏感信号列表>)  
begin  
    //过程赋值  
    //if-else、case选择语句  
    //for、while等循环块  
end
```

- **always**语句通常带触发条件，触发条件被写在敏感信号列表中，只有当触发条件满足条件或发生变化时，其后的”**begin-end**”块语句才能被执行。
- 敏感信号列表中可有多个信号，用关键字**or**连接；
- 敏感信号可分为两种：电平敏感、边沿敏感；
- 用关键字**posedge**和**negedge**限定信号敏感边沿。

# always块

- 电平触发的always块通常用于描述组合逻辑和带锁存器的组合逻辑；
- 边沿触发的always块通常用于描述时序逻辑。

```
module reg_adder (out, a, b, clk);  
    input clk;  
    input [2: 0] a, b;  
    output [3: 0] out;  
    reg [3: 0] out;  
    reg [3: 0] sum;  
    always @( a or b) // 若a或b发生任何变化，执行  
        sum = a + b;  
    always @( negedge clk) // 在clk下降沿执行  
        out = sum;  
endmodule
```

# 顺序块和并行块

- 顺序块（**begin——end**）
  - 语句顺序执行（带有内嵌延迟控制的非阻塞赋值语句除外）
  - 如果语句包括延迟或事件控制，延迟总是相对于前面那条语句执行完成的时刻
- 并发块（**fork——join**）
  - 语句并发执行
  - 语句执行的顺序由各自语句内延迟或者时间控制决定的
  - 语句中的延迟或事件控制是相对于块语句开始执行的时刻而言的。

# 示例代码

```
reg x,y;  
reg [1:0] z,w;  
initial  
    begin  
        x=1'b0;        //0  
        #5 y=1'b1;     //5  
        #10 z={x,y}    //15  
        #20 w={y,x}    //35  
    end
```

```
reg x,y;  
reg [1:0] z,w;  
initial  
    fork  
        x=1'b0;        //0  
        #5 y=1'b1;     //5  
        #10 z={x,y}    //10  
        #20 w={y,x}    //20  
    join
```

# Verilog HDL语法

- 模块的结构与实例化
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# 条件语句和循环语句

- 都必须在过程块中使用
- 条件语句：
  - if语句、case语句(case,casex,casexz)
- 循环语句：
  - forever语句、repeat语句、while语句、for语句

# 条件语句-if语句

- **if(表达式) 语句**                      例: **if(a>b)    out1=int1;**
- **If(表达式)**                      例: **if(a>b)**  
    **语句1**                                      **out1=int1;**  
    **else**                                      **else**  
    **语句2**                                      **out1=int2;**

```
always @ ( some_event )  
    begin  
        if (a>b)                      out1=int1;  
            else if(a==b)              out1=int2;  
                else  
out1=int3;  
    end
```

- 即使用不到else分支, 语句中else分支也最好加上, 否则电路有可能生成不稳定的电路, 造成结果的错误。



# 条件语句-case语句

- **case** (条件表达式)

分支1: 语句块1;

分支2: 语句块2;

.....

**default:** 语句块n;

**endcase**

例: **reg [2:0] cnt;**

**case(cnt)**

**3'b000:q=q+1;**

**3'b001:q=q+2;**

**default:q=q;**

**endcase**

- case语句的所有表达式值的位宽必须相等
- 语句中default一般不要缺省。在"always"块内, 如果给定条件下变量没有赋值, 这个变量将保持原值(生成一个锁存器)
- 分支表达式中可以存在不定值x和高阻值z(case x, case z)
  - 如2'b0x, 或2'b0z

# 条件语句if与case的区别

- if生成的电路是串行，有优先级的编码逻辑；
- case生成的电路是并行的，各种判定情况的优先级相同。
- if生成的电路延时较大，占用硬件资源少；
- case生成的电路延时短，但占用硬件资源多。

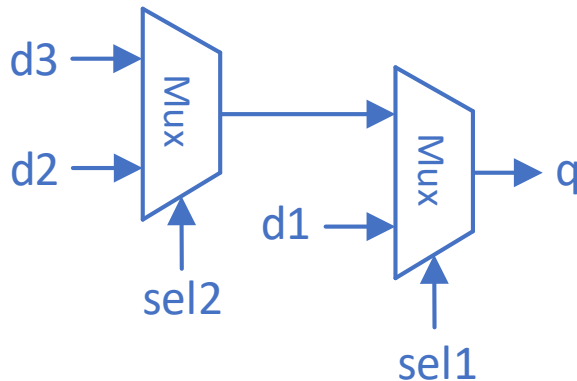
# if...else...语句与case语句的区别

- if...else...语句：表达的电路逻辑语义具有串行性，也就是说生成的数字逻辑电路要在逻辑上满足if...else...所表达的先后判断优先性语义。
- case语句：有并行的含义，会生成multiplexer电路，但是同时要注意，如果case没有完全覆盖所有情况，那么暗含着保持原来值的语义，会生成latch，可以加入default语句来避免这种情况。

# if...else...语句与case语句的区别

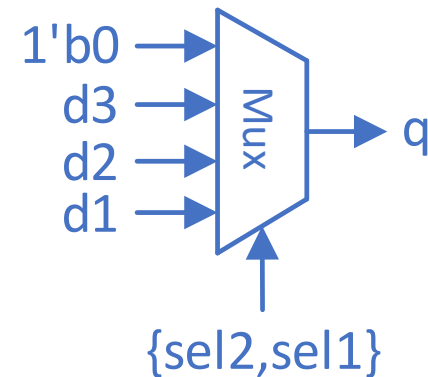
## if...else...

```
always @ (*) begin
    if (sel1)      q = d1;
    else if (sel2) q = d2;
    else          q = d3;
end
```



## case

```
always @ (*) begin
    case ({sel2,sel1})
        2'b00 : q = d1;
        2'b01 : q = d2;
        2'b10 : q = d3;
        default : q = 1'b0;
    endcase
end
```



# 循环语句

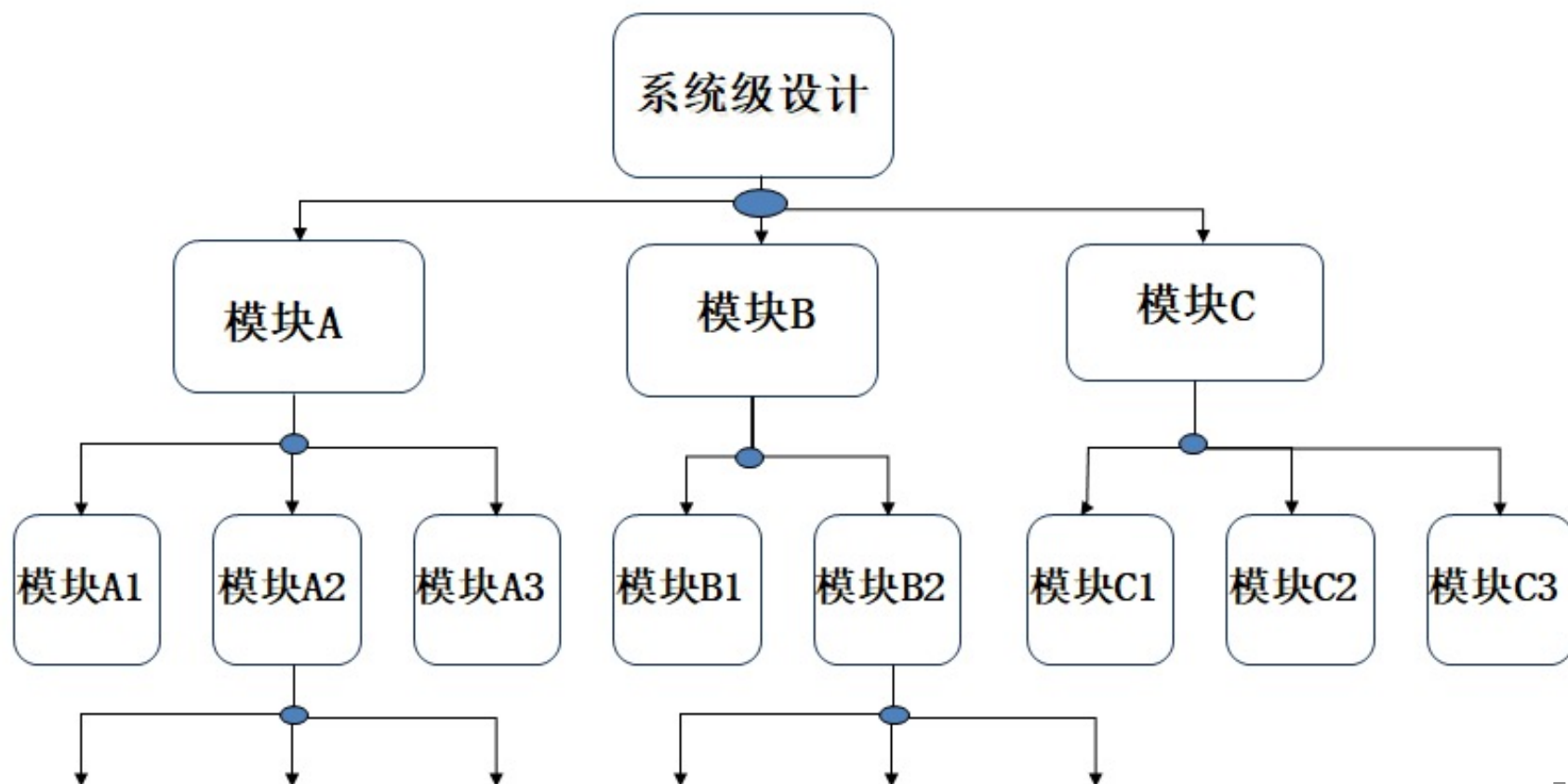
- Verilog的循环语句是依靠电路的重复生成实现的。
- 4种循环语句：
  - for 循环：执行给定的循环次数（不能用++）；
  - while 循环：执行语句直到某个条件不满足；
  - repeat 循环：连续执行语句N次；
  - forever 循环：连续执行某条语句。
- for、while是可综合的，但循环次数需要在编译之前就确定，动态改变循环次数的语句则是不可综合的
- repeat在有些工具中可综合，有些不可综合
- forever是不可综合的，常用于产生各类仿真激励

# Verilog HDL语法

- 模块的结构与实例化
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- **模块的调用**
- 模块的测试

# 模块的调用

- 无论多么复杂的系统，总能划分成多个小的功能模块。系统的设计可以按照下面三个步骤进行：



# 模块基本结构

模块声明:

```
module module_name (port_list) ; //模块名 (端口声明列表)
```

端口定义:

```
input[信号位宽];    //输入声明
output [信号位宽];  //输出声明
...
```

数据类型说明:

```
reg [信号位宽];      //寄存器类型声明
wire [信号位宽];     //线网类型声明
parameter;           //参数声明
...
```

功能描述: //主程序代码

```
assign a=b+c
always@(posedge clk or negedge reset)
function
...
endmodule
```



# 模块实例化方法

- 调用模块实例的一般形式为：
  - <模块名><参数列表><实例名>（<端口列表>）；
  - `module_name instance_name(port_associations) ;`
- 其中参数列表是传递到子模块的参数值。
- 信号端口可以通过位置或名称关联，但是关联方式不能够混合使用。
- 定义模块：`module Design(端口1, 端口2, 端口3.....);`
  1. 引用时，严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名。
    - `Design u_1(u_1的端口1, u_1的端口2, .....);` //和Design对应
  2. 引用时，用"."符号，标明原模块定义时规定的端口名。
    - `Design u_2( .(端口1(u_1的端口1), .(端口2(u_1的端口2),..... )`);

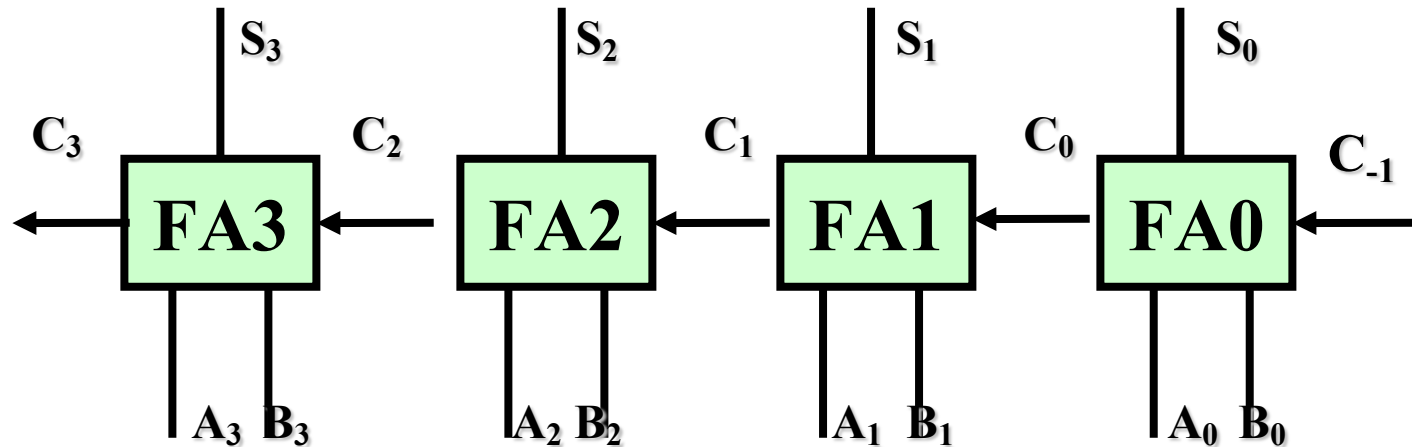
# 模块实例化

- 例:

```
module  and1 (C,A,B);  
input A,B;  
output C;  
...
```

- `and1 A1 (T3,A,B);` //实例化时采用位置关联。
- `and1 A2 (.C(T3), .A(A),.B(B) );` //实例化时采用名字关联
- `port_expr`可以是以下的任何类型:
  - 标识符（`reg` 或 `net`）如 `.C（T3）`，`T3` 为 `wire` 型标识符。
  - 位选择，如 `.C（D[0]）`，`C` 端口接到 `D` 信号的第0bit 位。
  - 部分选择，如 `.Bus（Din[5: 4]）`。
  - 上述类型的合并，如 `.Addr（{ A1, A2[1: 0]}）`。
  - 表达式（只适用于输入端口），如 `.A（wire Zire = 0 ）`。
- 例化名不能为元模块名或关键字。

# 4位全加器设计实例



- 半加器模块：半加器由两个一位输入相加，输出一个结果位和进位。
- 1位全加器模块：由两个半加器和一个或门实现。
- 4位全加器模块：由4个1位全加器串联形成。

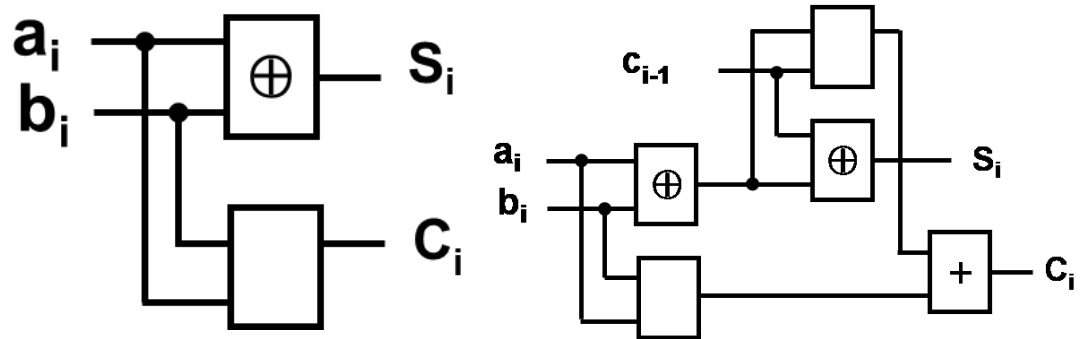
# 4位全加器设计实例

**module half\_adder(input a,input b, output sum,output c\_out); //半加器**

**assign sum = a^b;**

**assign c\_out = a&b;**

**endmodule**



**module full\_adder(input a,input b,input c\_in,output sum,output c\_out);**

**wire sum1;**

**//1 位全加器**

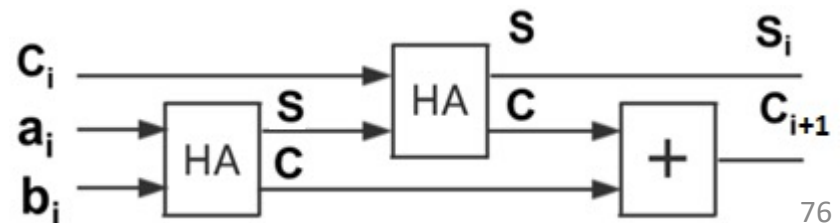
**wire c\_out1,c\_out2;**

**half\_adder half\_adder1(.a(a),.b(b),.sum(sum1),.c\_out(c\_out1));**

**half\_adder half\_adder2(.a(c\_in),.b(sum1),.sum(sum),.c\_out(c\_out2));**

**assign c\_out = c\_out1|c\_out2;**

**endmodule**



# 4位全加器设计实例

```
module add_4 ( input [3:0] a, input [3:0]b, input c_in,  
               output [3:0] sum, output c_out ); //4位全加器
```

```
    wire [3:0] c_tmp;
```

```
    full_adder i0 ( a[0], b[0], c_in, sum[0], c_tmp[0]);
```

```
    full_adder i1 ( a[1], b[1], c_tmp[0], sum[1], c_tmp[1] );
```

```
    full_adder i2 ( a[2], b[2], c_tmp[1], sum[2], c_tmp[2] );
```

```
    full_adder i3 ( a[3], b[3], c_tmp[2], sum[3], c_tmp[3] );
```

```
    assign c_out = c_tmp[3];
```

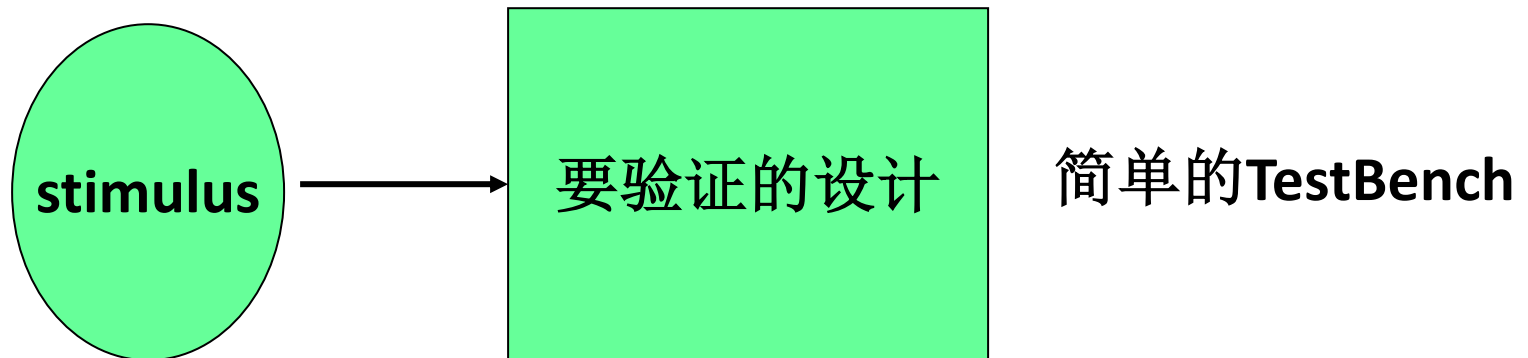
```
endmodule
```

# Verilog HDL语法

- 模块的结构与实例化
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- 模块的测试

# 测试平台

- 测试平台（**test bench**）是一个无输入，有输出的顶层调用模块。
- 一个简单的测试平台包括：
  - 产生激励信号；
  - 实例化待测模块，并将激励信号加入到待测模块中。



# TestBench模块

- 激励模块通常是顶层模块
- 激励信号数据类型要求为reg，以便保持激励值不变，直至执行到下一条激励语句为止
- 输出信号的数据类型要求为wire，以便能随时跟踪激励信号的变化

```
'timescale 1ns/1ns    //时间单位为1ns，精度为1ns
module module_name_sim();    //模块名（无端口声明列表）
    reg [信号位宽];        //激励信号声明
    wire [信号位宽];        //输出信号声明
    module_name instance_name (port_associations);    //实例化设计模块
    initial
        begin    //激励信号
            PS=1'b0;PD1=1'b1;    //语句1
            #5 PS=1'b0;PD1=1'b1;    //语句2
            .....
        end
    endmodule
```

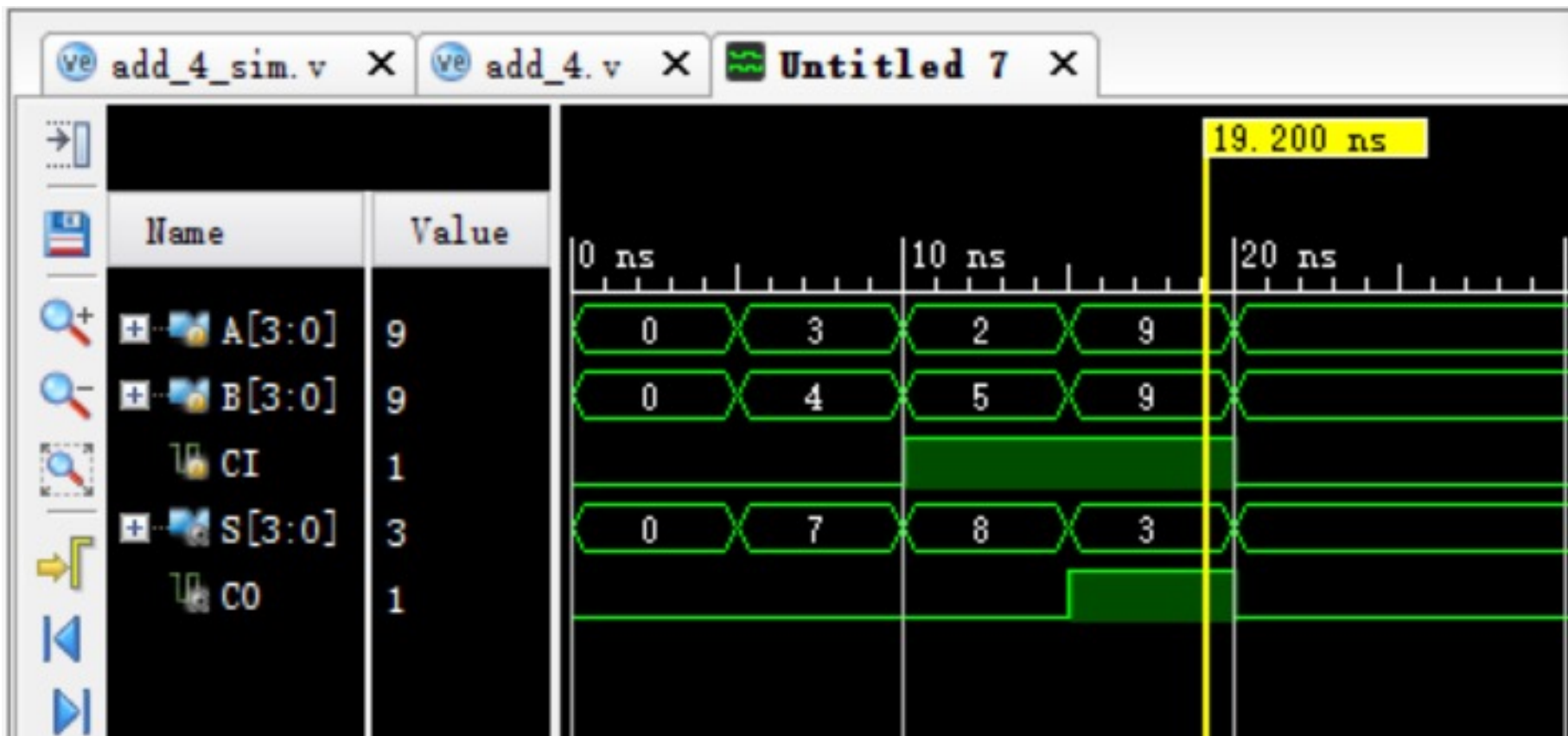


# 4位全加器模块测试

```
`timescale 1ns / 1ps
module add_4_sim();
    reg [3:0] A,B;
    reg Cl;
    wire [3:0] S;
    wire CO;
    add_4 A1(A,B,Cl,S,CO);
    initial
        begin
            A=4'd0;B=4'd0;Cl=1'b0;
            #5 A=4'd3;B=4'd4;Cl=1'B0;
            #5 A=4'd2;B=4'd5;Cl=1'b1;
            #5 A=4'd9;B=4'd9;Cl=1'b1;
            #5 A=4'd0;B=4'd0;Cl=1'b0;
        end
endmodule
```

```
module add_4 ( input [3:0] a, input [3:0]b,
               input c_in, output [3:0] sum,
               output c_out );

    wire [3:0] c_tmp;
    full_adder i0 ( a[0], b[0], c_in, sum[0],
                   c_tmp[0]);
    full_adder i1 ( a[1], b[1], c_tmp[0],
                   sum[1], c_tmp[1] );
    full_adder i2 ( a[2], b[2], c_tmp[1],
                   sum[2], c_tmp[2] );
    full_adder i3 ( a[3], b[3], c_tmp[2],
                   sum[3], c_tmp[3] );
    assign c_out = c_tmp[3];
endmodule
```



```

A=4'd0;B=4'd0;CI=1'b0;
#5 A=4'd3;B=4'd4;CI=1'B0;
#5 A=4'd2;B=4'd5;CI=1'b1;
#5 A=4'd9;B=4'd9;CI=1'b1;
#5 A=4'd0;B=4'd0;CI=1'b0;

```

# Verilog HDL数字系统设计

# Verilog HDL数字系统设计

- 组合逻辑与时序逻辑的区别
- 建立时间与保持时间
- 锁存器
- 组合逻辑环
- 异步信号处理

# 组合逻辑与时序逻辑的区别

- 概念上的区别

- **组合逻辑**：任意时刻的输出**仅取决于该时刻的输入**，与电路原本的状态无关，逻辑中不牵涉跳变沿信号的处理
- **时序逻辑**：任意时刻的输出不仅取决于该时刻的输入，而且还和电路原来的状态有关。电路里面有存储元件（触发器）用于记忆信息，不管输入如何变化，**仅当时钟的沿（上升沿或下降沿）到达时**，输出才发生变化。

# 组合逻辑与时序逻辑的区别

- Verilog描述的区别

- 组合逻辑:

```
reg q;  
always @ (*) begin  
    if (d_en) q2 = d;  
    else      q2 = 1'h0;  
end
```

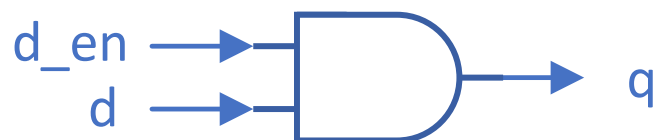
- 时序逻辑:

```
reg q;  
always @ (posedge clk or negedge rst_n) begin  
    if (~rst_n) q1 <= 1'h0;  
    else if (d_en) q1 <= d;  
end
```

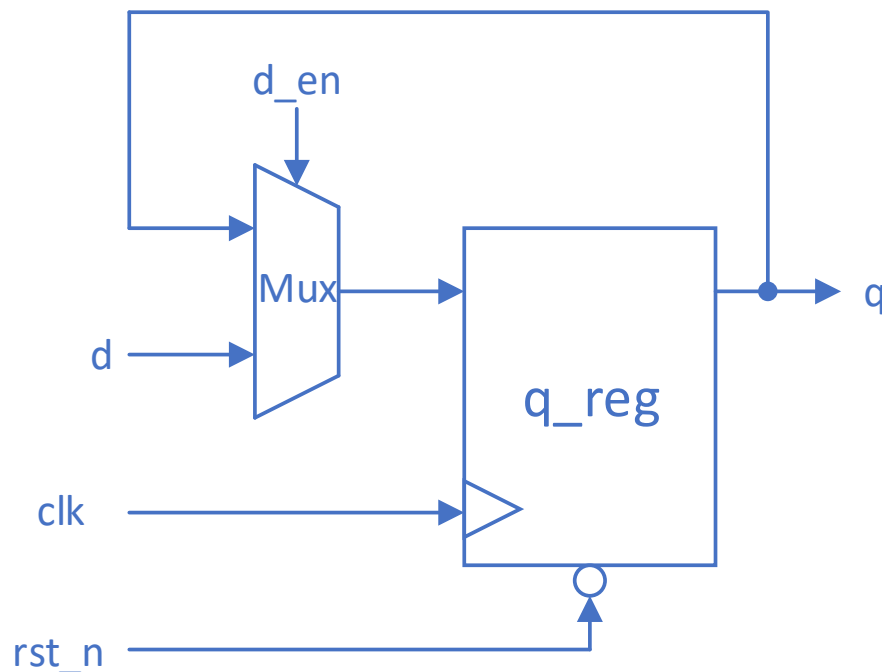
# 组合逻辑与时序逻辑的区别

- 生成电路的区别

组合逻辑：



时序逻辑：



# Verilog HDL数字系统设计

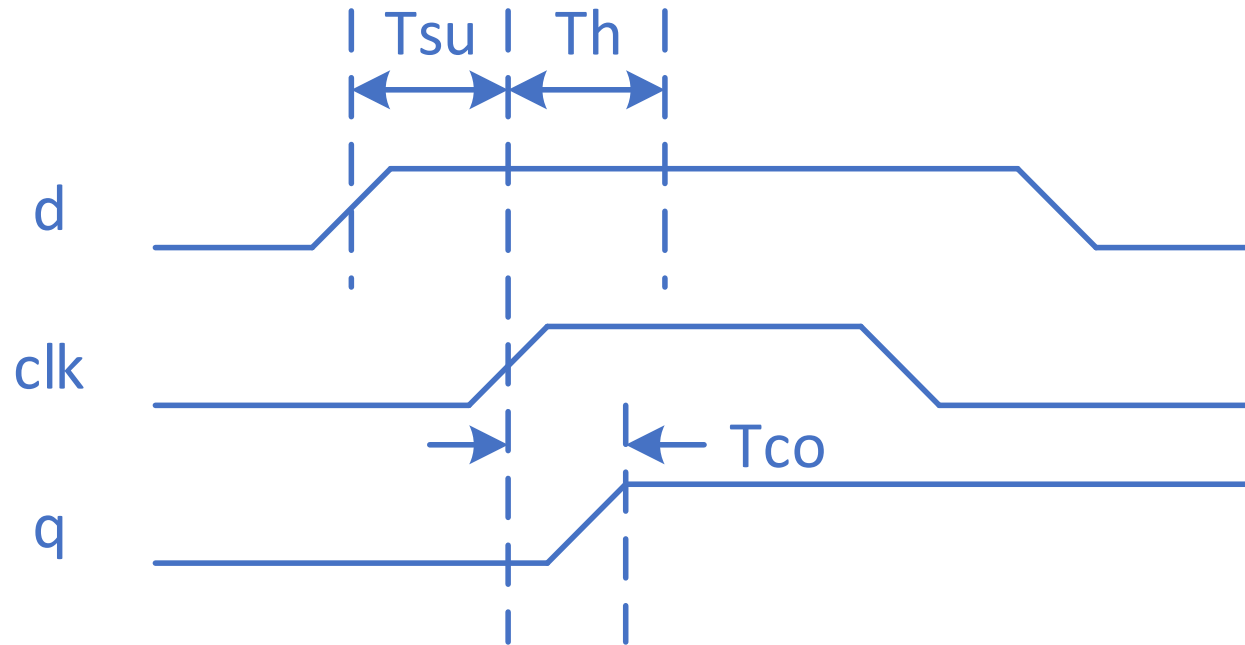
- 组合逻辑与时序逻辑的区别
- 建立时间与保持时间
- 锁存器
- 组合逻辑环
- 异步信号处理



# 建立时间与保持时间

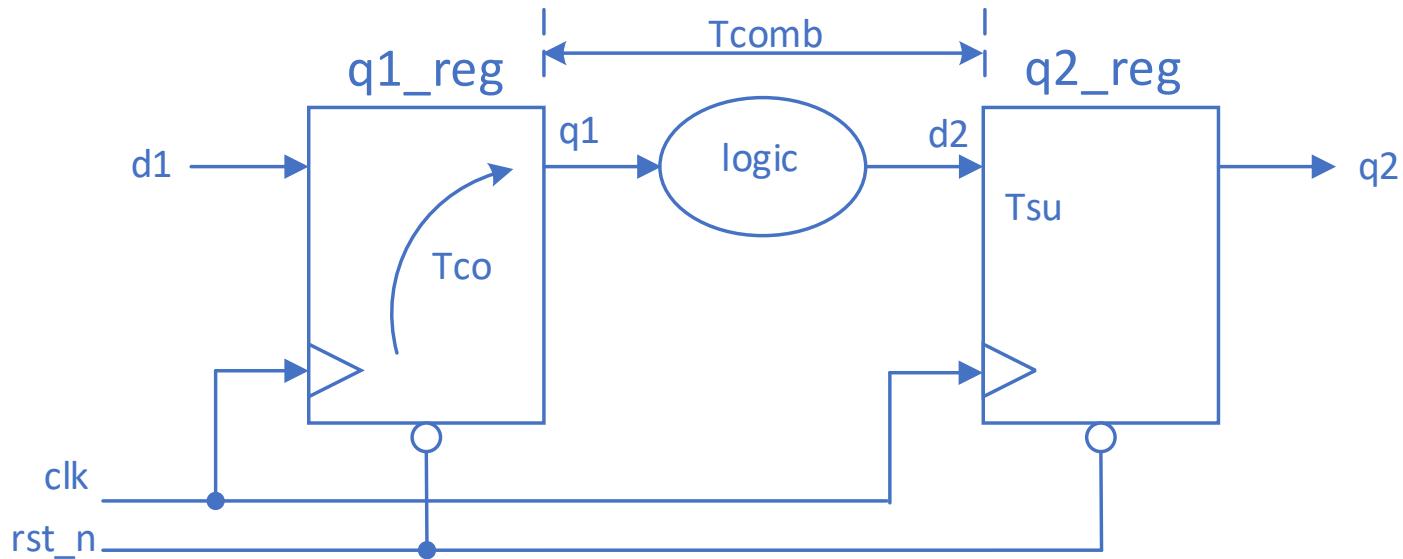
- 建立时间和保持时间都是针对**触发器**的特性说的。
  - **建立时间（Tsu: set up time）**：是指在触发器的时钟信号上升沿到来以前，数据稳定不变的时间，如果建立时间不够，数据将不能在这个时钟上升沿被稳定的打入触发器，Tsu就是指这个最小的稳定时间。
  - **保持时间（Th: hold time）**：是指在触发器的时钟信号上升沿到来以后，数据稳定不变的时间，如果保持时间不够，数据同样不能被稳定的打入触发器，Th就是指这个最小的保持时间。

# 建立时间与保持时间



**输出相应时间 ( $T_{co}$ )**：触发器的输出在clk时钟上升沿到来之后多长的时间内发生变化，即触发器的输出延时。

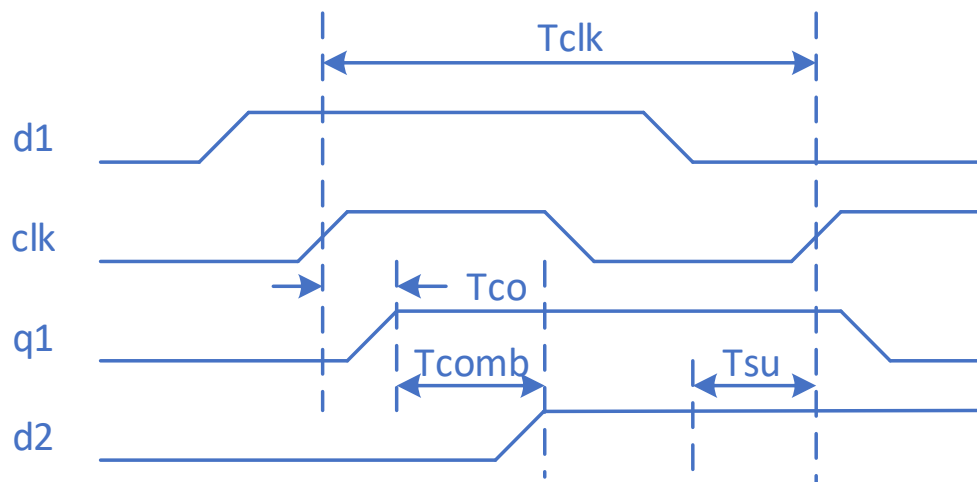
# 建立时间与保持时间



**问题：** 如何保证数据能够正确的在这两个触发器上进行传输，由此确定中间组合逻辑电路的传输延时 $T_{comb}$ 的范围。

# 建立时间与保持时间

## 建立时间的约束条件

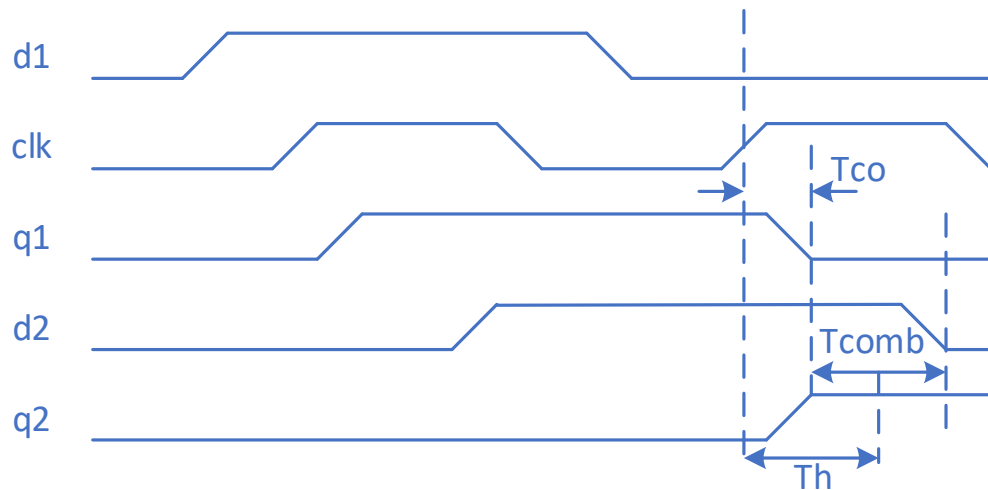


在第一个时钟上升沿，前边的触发器采集D1信号，将高电平打入触发器，经过Tco的触发器输出延时到达组合逻辑电路。又经过组合逻辑电路的延时Tcomb，送到了D2接口上。在第二个时钟上升沿到来之前，D2数据线上的信号要满足稳定时间>触发器的建立时间Tsu。

$$T_{clk} - T_{co} - T_{comb} > T_{su}$$

# 建立时间与保持时间

## 保持时间的约束条件



在第二个时钟上升沿前边触发器采集到D1上的低电平，经过Tco的延时在Q1上得到表达。这个低电平在经过组合电路延时Tcomb到达D2。D2上原本的高电平在第二个时钟上升沿到来之后的稳定时间 > 第二个触发器的保持时间。

$$T_{co} + T_{comb} > T_h$$

# 建立时间与保持时间

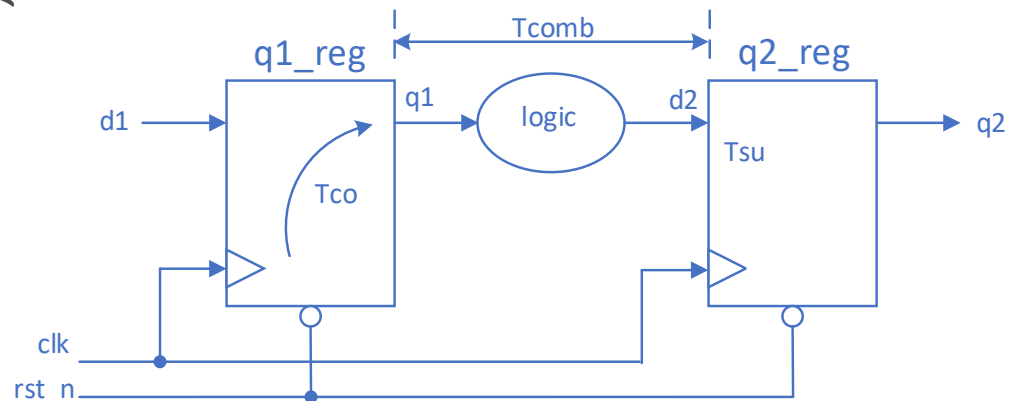
处理方法:

保持时间: 无需特别处理, 后端工具可以解决;

建立时间:

$$T_{clk} - T_{co} - T_{comb} > T_{su}$$

对于给定的时钟频率和工艺,  
 $T_{clk}$ 、 $T_{co}$ 和 $T_{su}$ 都是固定的,  
所以欲满足上述公式, 需要  
减少 $T_{comb}$ , 即优化两级寄  
存器之间的组合逻辑;



# Verilog HDL数字系统设计

- 组合逻辑与时序逻辑的区别
- 建立时间与保持时间
- 锁存器
- 组合逻辑环
- 异步信号处理

# 锁存器

- 什么样的Verilog描述会生成锁存器
  - if...else...语句没有else

```
always @ (*) begin
    if (d_en) q = d;
end
```

```
always @ (*) begin
    if (d_en) q = d;
    else      q = q;
end
```

- case语句没有default

```
always @ (*) begin
    case (cnt[1:0])
        2'b00 : q = d1;
        2'b01 : q = d2;
        2'b10 : q = d3;
    endcase
end
```



# 锁存器

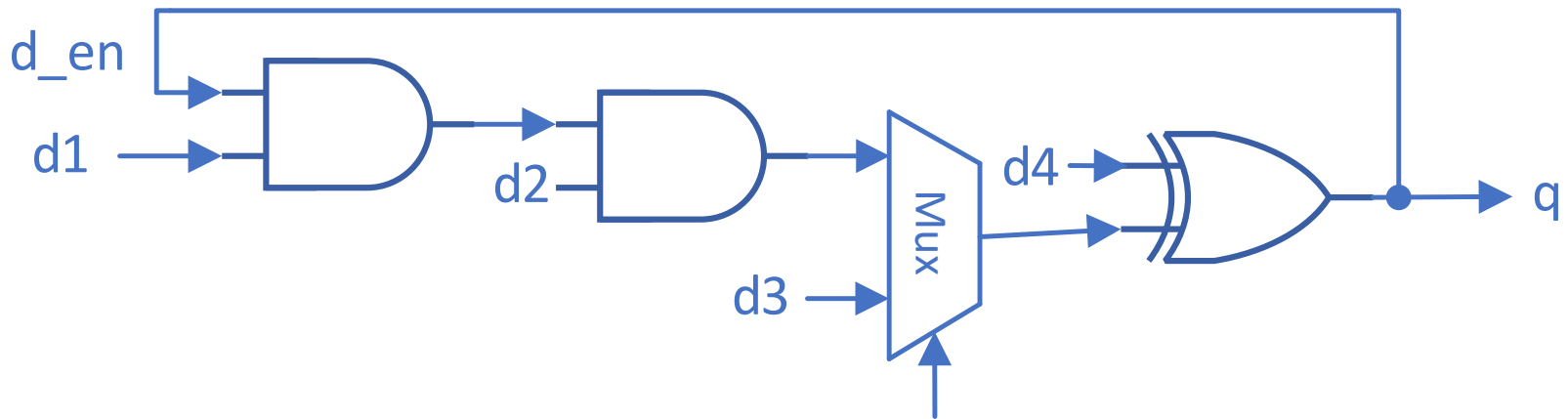
- 锁存器的危害
  - 对毛刺敏感，不能异步复位，所以上电以后处于不确定的状态；
  - Latch会使静态时序分析变得非常复杂；

# Verilog HDL数字系统设计

- 组合逻辑与时序逻辑的区别
- 建立时间与保持时间
- 锁存器
- 组合逻辑环
- 异步信号处理

# 组合逻辑环

- 组合逻辑环（**Combinational loop**）：起始于某个组合逻辑单元经过一串组合逻辑又回到起始组合逻辑单元的逻辑环路，称为组合逻辑环。



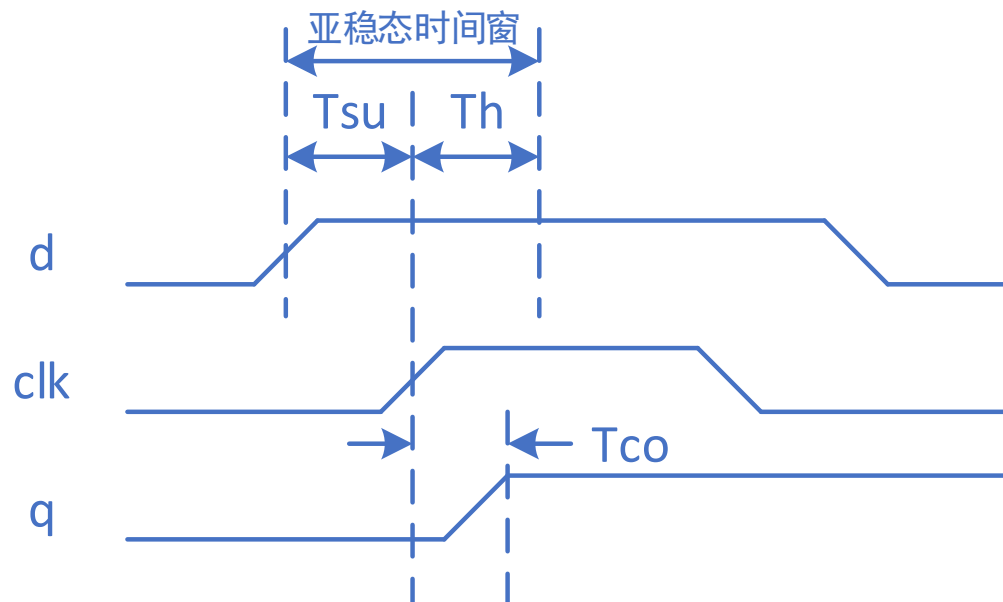
解决方法：插入寄存器，打断组合逻辑环；

# Verilog HDL数字系统设计

- 组合逻辑与时序逻辑的区别
- 建立时间与保持时间
- 锁存器
- 组合逻辑环
- 异步信号处理

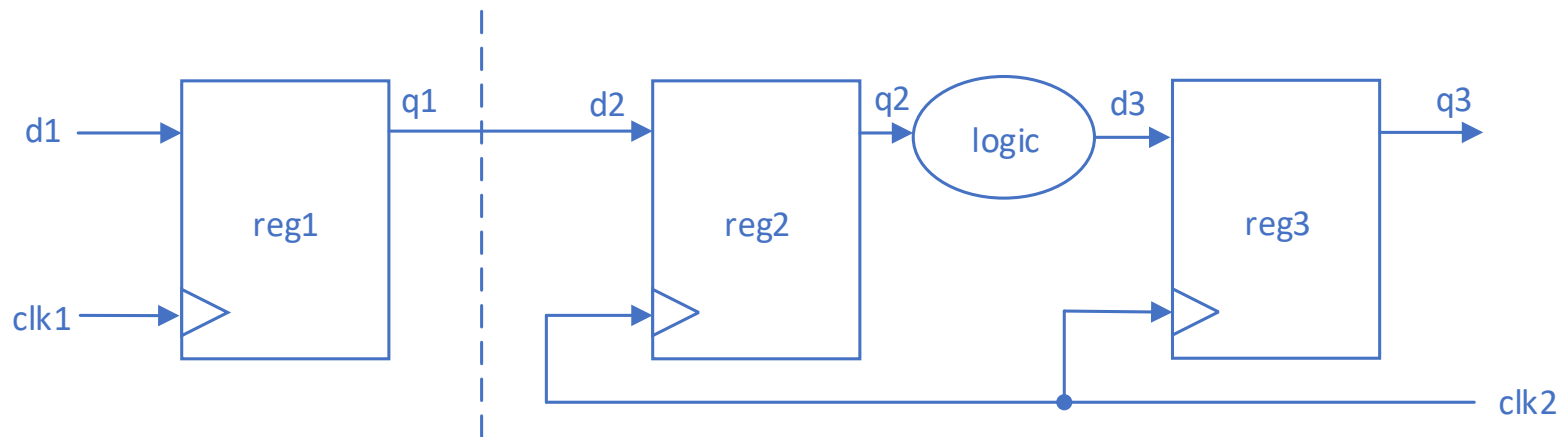
# 异步信号处理

- **亚稳态：**指触发器无法在亚稳态时间窗内达到一个可确认的状态。但触发器进入亚稳态时，既无法预测该单元输出电平，也无法预测何时才能稳定在某个正确电平上。



# 异步信号处理

- 异步数据传输可能导致亚稳态



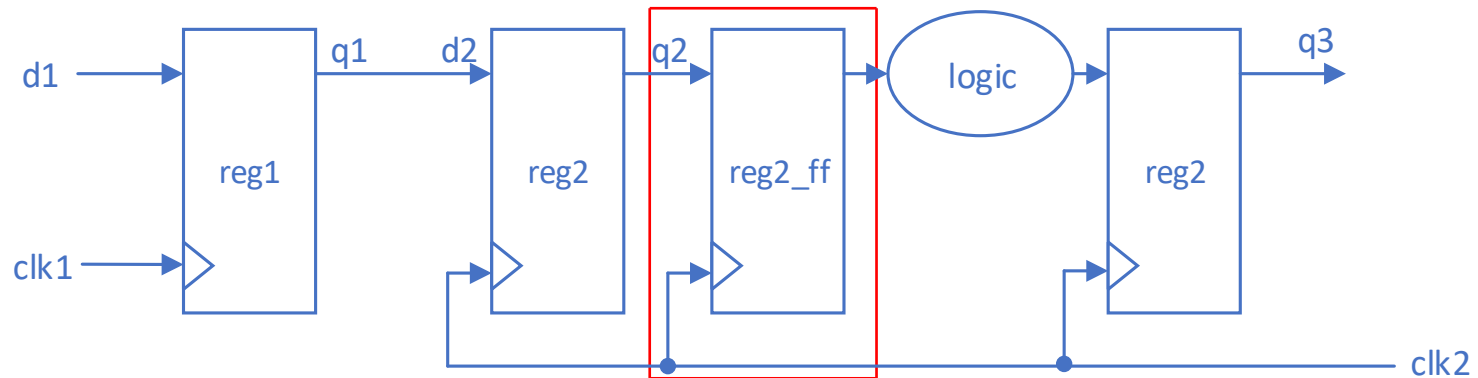
两个时钟`clk1`和`clk2`没有相关性，无法保证`reg1`的输出在`reg2`上能满足建立保持时间。

# 异步信号处理

## 异步数据处理-单bit信号

处理方法：

加入多级寄存器，降低亚稳态概率；



注意：

若是脉冲信号，可能需要脉冲宽度展宽；

# 异步信号处理

## 异步数据处理-多bit信号

处理方法：

使用握手协议或者异步FIFO。

注意：

在握手协议中，异步的REQ/ACK需要使用上述同步技术进行同步处理，异步FIFO也是如此。