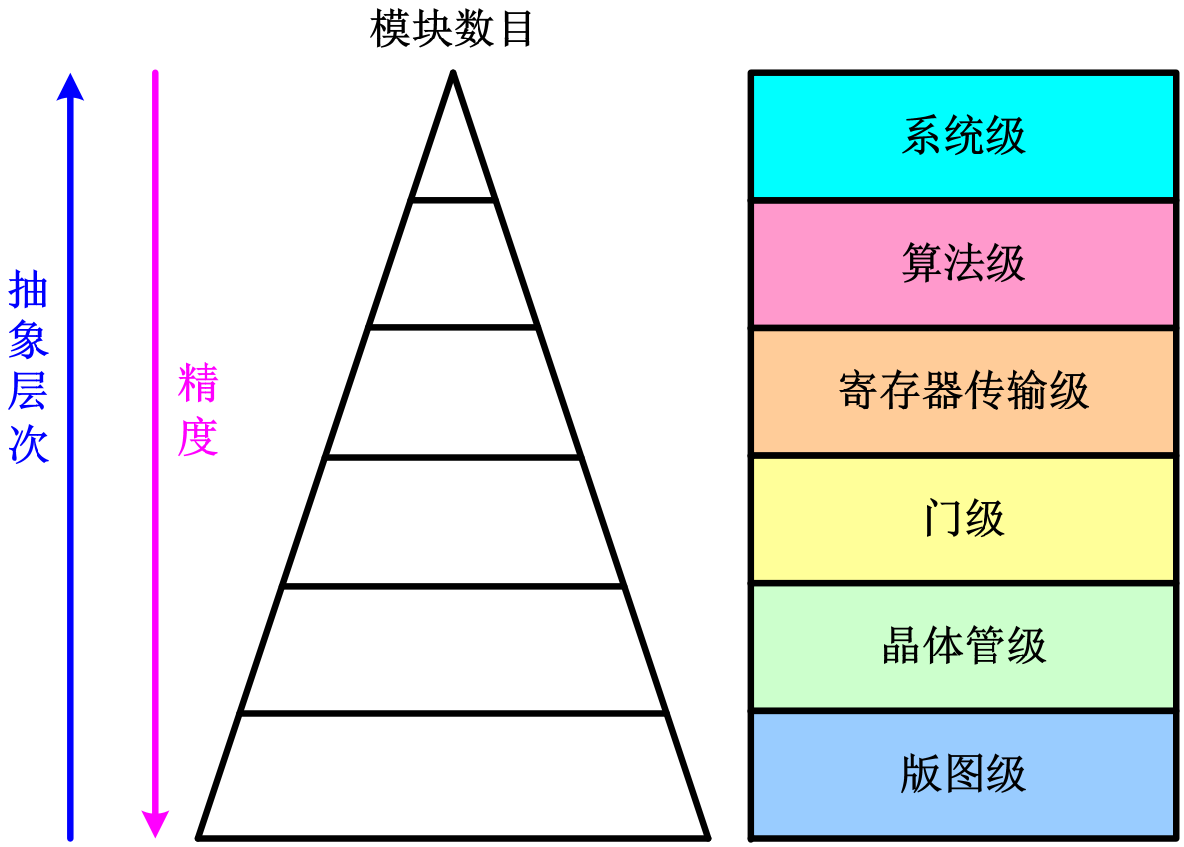
上海交通大學
Shanghai Jiao Tong University

Digital Circuits and Verilog HDL

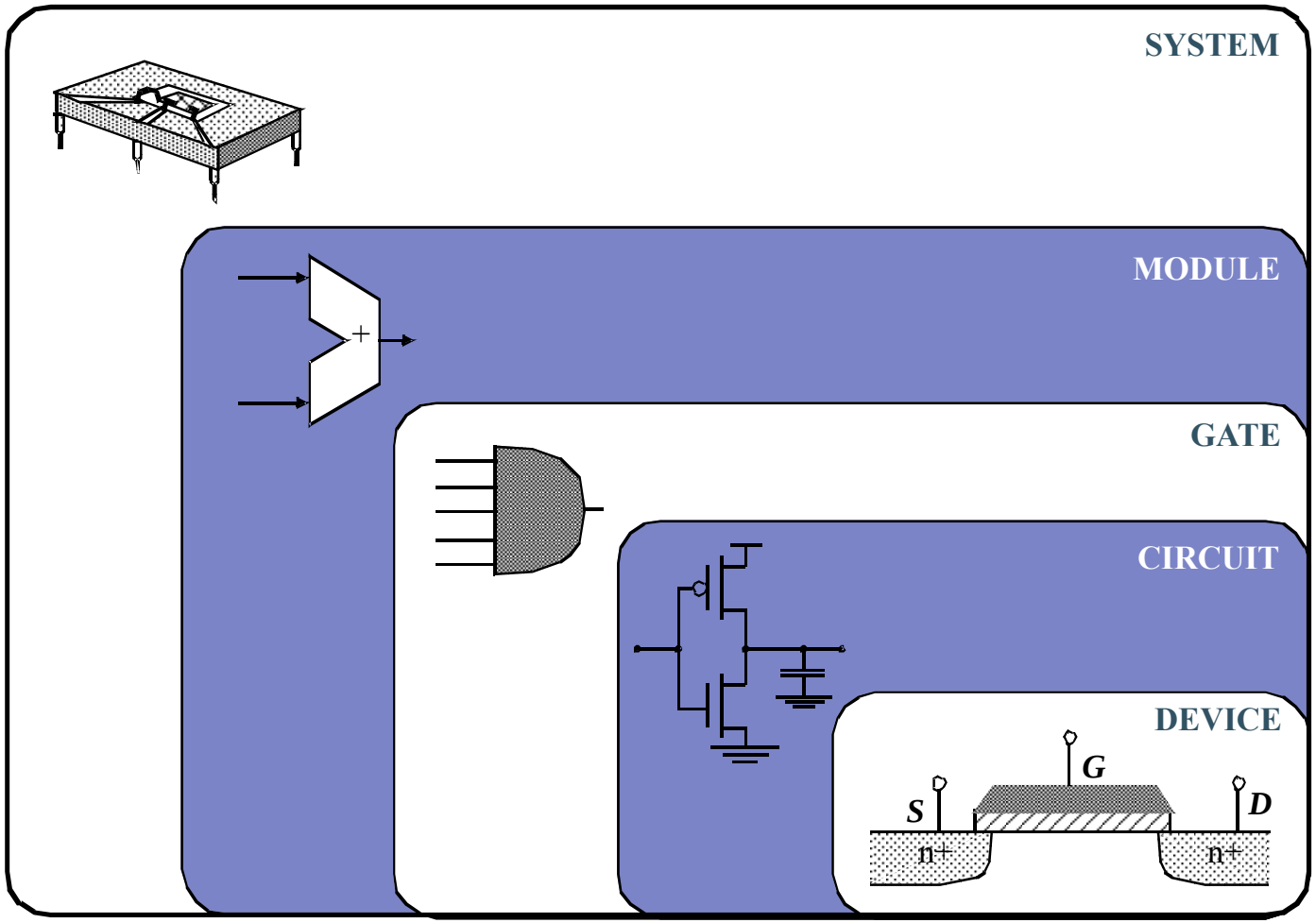
景乃锋

2015.3.20

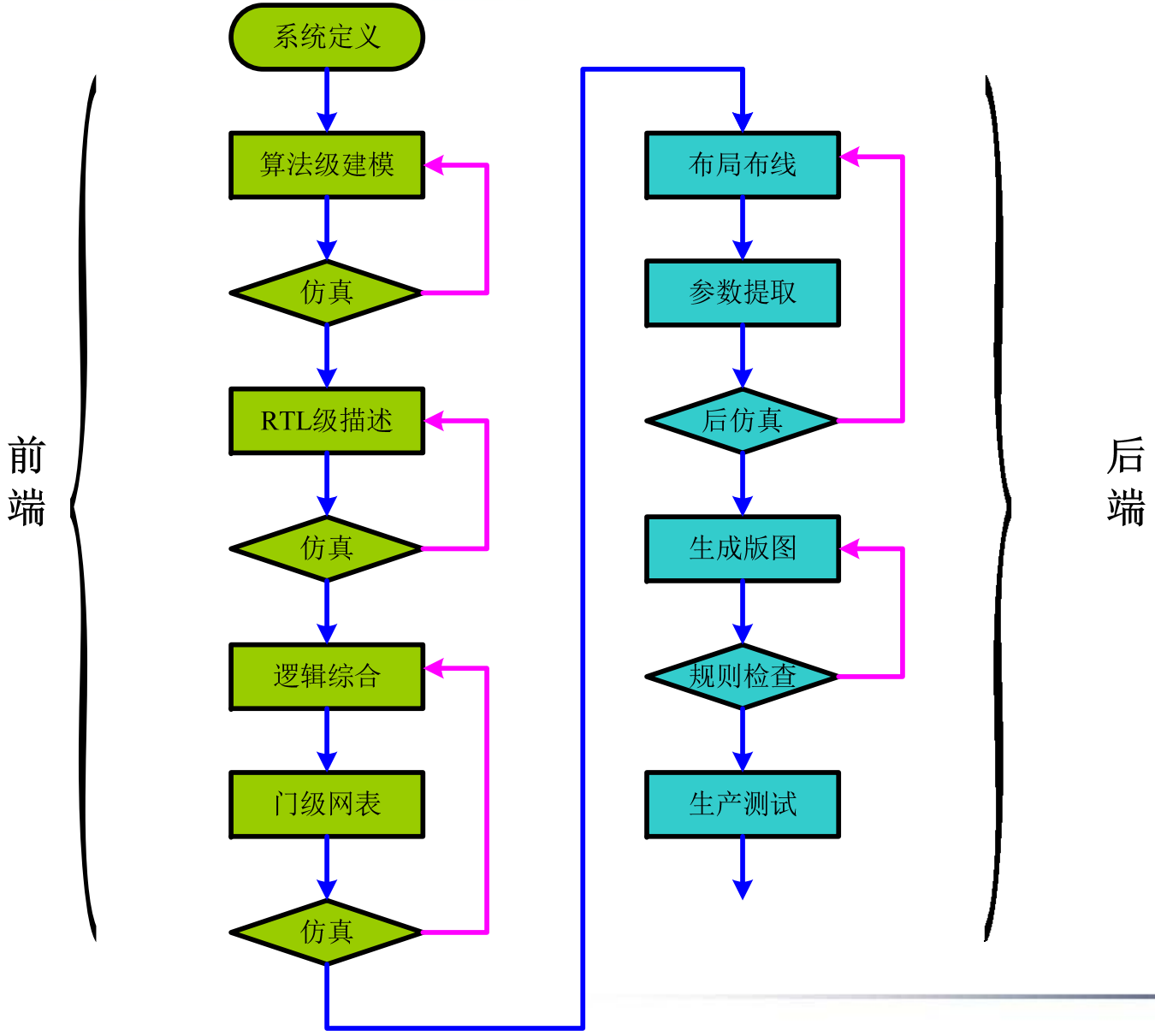
电路抽象层次



电路抽象层次



IC设计基本流程


系统定义
算法级建模
仿真
RTL级描述
仿真
逻辑综合
门级网表
仿真
布局布线
参数提取
后仿真
生成版图
规则检查
生产测试
前端
后端

Verilog HDL发展历史

- 1985年在英国由Automated Integrated Design Systems（Gateway Design Automation，1990年被Cadence收购）公司发明，作为仿真语言。
- 1995年成为IEEE（Institute of Electrical and Electronics Engineers ，美国电气及电子工程师学会）标准IEEE Standard 1364-1995（Verilog-95）。
- Verilog-2001（ IEEE Standard 1364-2001）对于Verilog-95的扩展。
- Verilog-2005（ IEEE Standard 1364-2005）最新的Verilog标准。

其他电路设计主要描述语言

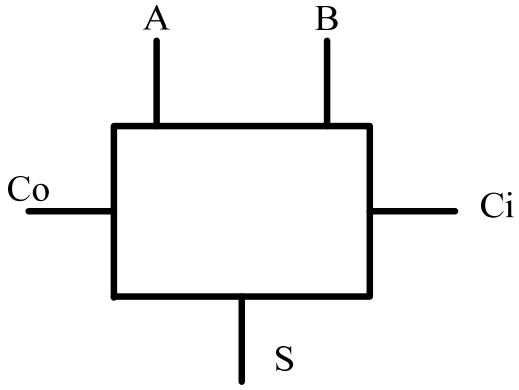
- VHDL（VHSIC HDL），美国国防部发起，1987年成为IEEE标准，在欧洲被较多使用。
- System Verilog（Verilog语言的扩展，主要用于验证），2005年成为IEEE标准。
- SystemC（C语言的扩展，主要用于高层次的硬件系统表述）2005年成为IEEE标准。

HDL在设计流程中的作用

- **使用HDL描述设计具有下列优点：**
  - 设计在高层次进行，提高设计的效率
  - 设计不涉及到制造工艺，可以灵活地复用
  - 可以利用EDA软件，自动的将高级描述映射到更低层次的抽象

全加器

全加器表述



真值表

A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器的Verilog描述

$$P = A \oplus B .$$

$$S = P \oplus Ci$$

$$Co = PCi + P'A$$

模块名

模块

```
module full_adder ( a, b, ci, s, co);

input a,b,ci;
output s;
output co;

wire a,b,ci,s,co;
wire p;

assign    p=a^b;
assign    s=p^ci;
assign    co=(p & ci) | (!p  &  a);

endmodule
```

有分号

端口声明

类型定义

表达式

没有分号

module（模块）

- 模块是Verilog的基本设计描述单位，用于描述某个设计的功能或者结构，以及他们与其他模块通信的外部端口。
- module和endmodule总是成对出现，是系统的保留字。
- Verilog文件的后缀是“.v”,如full_adder.v。一个verilog文件中可以包括多个module，一般建议一个module对应一个文件。

标识符

- Verilog HDL中的标识符（identifier)可以是任意一组字母、数字、$符（$)、下划线（_)的组合。
  标识符区分大小写。
  标识符的第一个字母必须是字母或者是下划线。

- 系统标识符（关键字、保留字）
  Verilog HDL定义了一系列的保留字，保留字都是小写的。
  例：module，endmodule, input, ouput, wire, assign等

- 用户自定义标识符需符合上面规定

注释

- 注释
  - 单行注释:
    //本行被注释
  - 多行注释:
    /* 本段被注释 ...*/
  - 养成良好的注释习惯

```
//version:xxxx
//date:xxx
//author:xxx
/*=========================
          a        b
          |        |
         ----------------
  co__ |      FA      |__ ci
         |              |
         ----------------
                | S
==========================*/
module full_adder ( a, b, ci, s, co);
input a,b,ci;                //c is a carry_in。
ouput s;                      //
output co;                   //
wire a,b,c,s,co;
wire p;
assign     p=a^b;
assign     s=p^ci;
assign     co=(p & ci) | (!p  &  a);
endmodule
```

格式

- 格式
  - 除区分大小写外，Verilog HDL是自由格式的，可以跨越多行，也可以在一行内编写。

```
module full_adder ( a, b, ci, s, co);
input a,b,ci;
output s;
output co;
wire
a,b,ci,s,co;
wire p;
assign     p=a^b;
assign     s=p^ci;
assign     co=(p & ci) | (!p  &  a);
endmodule
```

  - Verilog中三种符号认为是空白，没有特殊意义
    空格、制表符、换行符

端口

- 模块的端口是模块与外部模块的通信接口，端口可以分为三种类型：
  - 输入端口 （input）
  - 输出端口 (output)
  - 输入输出端口（inout）

```
module test (address,ctrl,data);

input  [1:0] address;
output [2:0] ctrl;
inout  [8:0]data;
```

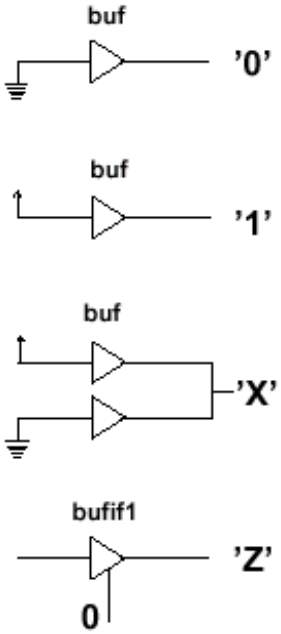
端口定义中需要指定类型、宽度

```
input  a,b,c;
wire  a,b,c;
```

值集合与数据类型

值集合

- **逻辑0，false**
- **逻辑1，true**
- **未知值 " x"**
- **高阻 " z"**

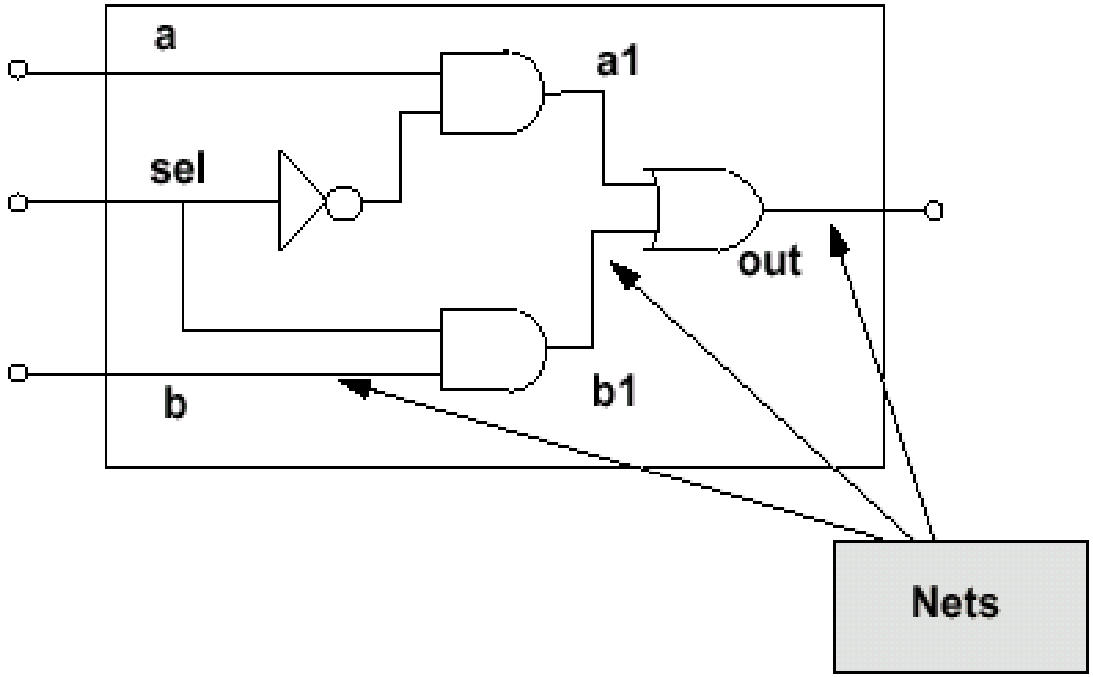


数据类型

- 线网类型（nets）
- 寄存器类型（registers）

线网(1)

- 线网类型表示单元之间的物理连线，线网不存储值，它的值由驱动单元的值决定，比如连续的赋值或者门的输出，如果没有驱动，线网的缺省值是Z（trireg类型除外）。



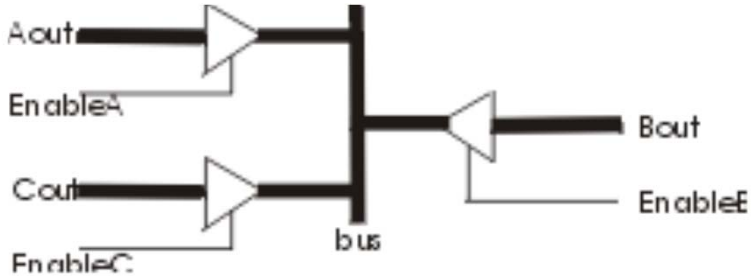
线网（2）

- 线网类型包括以下11种子类型

net类型	功 能
wire, tri	标准内部连接线(缺省)
supply1, supply0	电源和地
wor, trior	多驱动源线或
wand, triand	多驱动源线与
trireg	能保存电荷的net，用于电容节点建模
tri1, tri0	无驱动时上拉/下拉

wire/tri类型

- 它们是最常见的线网类型，两者的语法与功能一致，tri本意用于描述多个驱动源同时驱动一根线的线网类型，而wire本意用于描述一个驱动源的驱动。实际上应用中，没有区别，为了可读性在多个驱动源时建议使用tri类型。



- Wire/tri在多个驱动源时值的判断

Table 3-2—Truth table for wire and tri nets

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

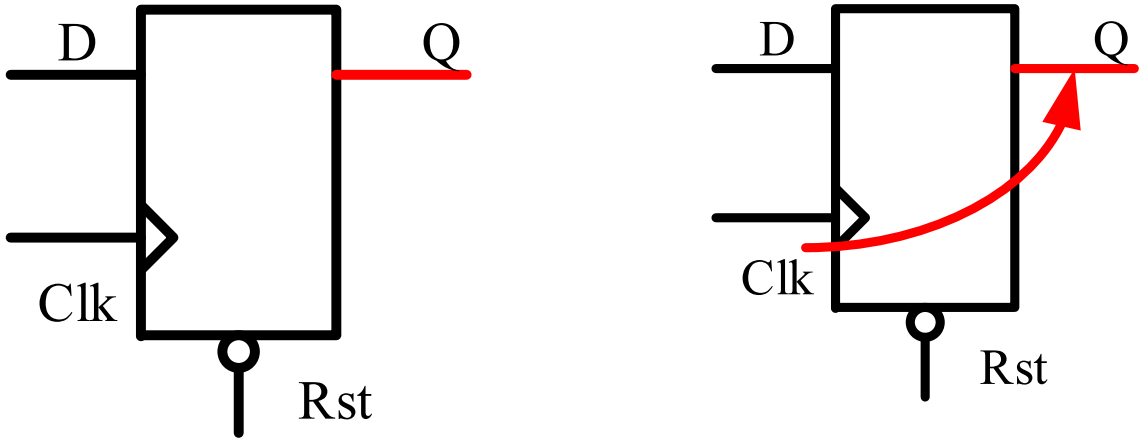
系统缺省的数据类型是1比特的wire类型。

寄存器类型（register)

- 寄存器类型(register)分为5种子类型
  - reg类型
  - integer类型
  - real类型
  - time类型（testbench中介绍）
  - realtime类型（testbench中介绍）

reg类型

- reg类型是数据存储单元的抽象。



- reg类型与wire一起类型是RTL级描述的基本数据类型。

在硬件描述语言中，有时候数据类型并不一定与硬件电路相关，
在组合逻辑的行为描述中，也会定义reg类型。

操作符

Verilog中的操作符

分类	操作符
连接复制符	{} , {{}}
算术运算符	* , / , + , - , %
关系操作符	<, <=, >, >=
相等操作符	==,!=, ===,!==
逻辑操作符	!,&&,||
比特操作符	~, &, |, ^, ^~(~^)
归约操作符	&, ~&, |, ~|, ^, ~^(^~)
移位操作符	<<, >>
条件操作符	? :
事件操作符	or

连接复制操作符

- 连接复制操作符{}，{{}}

```
wire [7:0]  a,b,c,d,e;
wire [15:0] f;
assign e={a[7:6], b[5:4], c[3:2], a[1:0]};  //连接
assign f={a,b};
…
data1={3'b101,2'b11,3'd7};                  //连接
…
data2={128{3'b101}};                        //复制
data2={100{1'b1},8{a},8{f[15:7]}};   //复制，连接
```

次数

注意：连接的数之间用逗号分开
连接的数必须指定位数

可以从不同的矢量中选择位
并用它们组成一个新的矢量。
用于位的重组和矢量构造

在级联和复制时，必须指定位数，否则将产生错误。
下面是类似错误的例子：

```
a[7:0] = {4{ ´b10}};
b[7:0] = {2{ 5}};
c[3:0] = {3´b011, ´b0};
```

算术运算

- 算术运算有* ，/ ， + ，- ，%

逻辑操作符

逻辑操作符 &&, ||, !

```
…
a = 4‘b0011;     //逻辑值为“1”
b = 4‘b10xz;     //逻辑值为“1”
c = 4‘b0z0x;     //逻辑值为“x”
e = a && b;      // e = 1
f =  a && c;     // f = x
g = ！ a;        // g=0
```

- 逻辑操作符的结果为一位1，0或x。
- 逻辑操作符只对逻辑值运算。
- 若操作数只包含0、x、z，则逻辑值为x
- 逻辑反操作符将操作数的逻辑值取反。

相等操作符

- 逻辑相等==， ！=

  操作数中只要含有x或者z，结果是x

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

- Case等===， ！==

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

```
…
a = 4‘b101z;
b = 4‘b10xz;
c=  4‘b10xz;
d =（a==b);        //d=x
e = (a===b);       // e=0
f  = (b===c);      //  f=1
```

比特操作符

- 比特操作符:~，&，|，^，~^(^~)

```
…
a=4’b1011;
b=4’b1010;
h=4’b1x0x;
c=~a;          //c=4’b0100;
d=a&b;         //d=4’b1010;
e=a^b;         //e=4’b0001;
f=a|h;         //f=4’b1x11;
```

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

~	
0	1
1	0
x	x
z	x

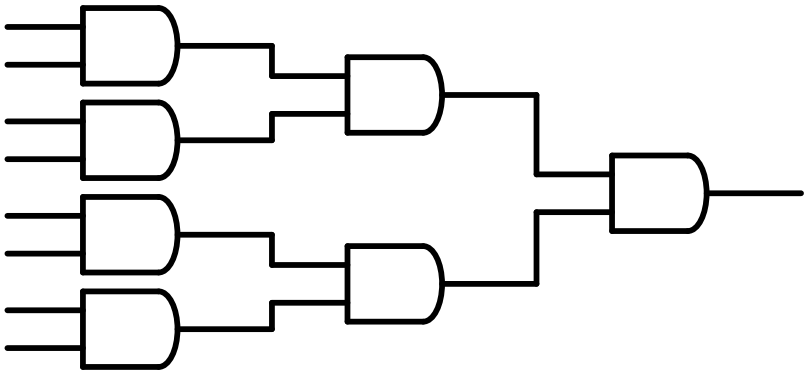
^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

^~ ~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

归约操作符

- 归约操作：&, ~&, |, ~|, ^, ^~(~^)

```
…
a=4’b1011;
b=4’b1010;
h=4’b1x0x;
c=&a;          //c=1’b0;
d=|b;          //d=1’b1;
e=&h;          //e=1’b0;
f=^a;          //f=1’b1;
```



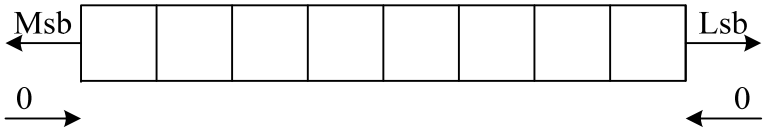
- 归约操作符的操作数只有一个。
- 对操作数的所有位进行位操作。
- 结果只有一位，可以是0, 1, X。

```
wire [7:0]  data;
full_p1=(data==8’hff);

full_p2=&data;
```

移位操作符

- 移位操作符: <<, >>

```
…
wire [7:0] data,result;

assign result = data <<3;

assign result ={data[4:0],3'b0};
```



- 在移位操作中，右边的操作数总是被当成无符号数，因此在需要位扩展时总是填0。
- 移入的位只能是0.

条件操作符

条件操作符：？：

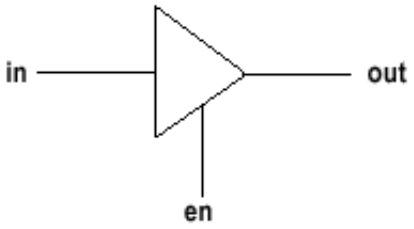
result=expression1 ? expression2:expression3

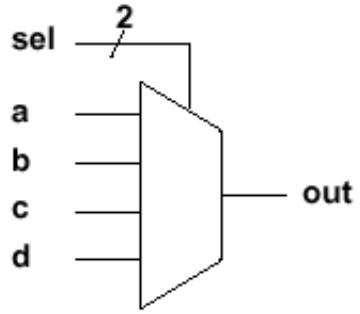
expression1 为真执行 **expression2**

否则执行**expression3**

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

```
…
out=en？ in: 1’bz;      //三态门

out=(sel==2’b00)?a:
      (sel==2’b01)?b:
      (sel==2’b10)?c:d;
```





事件操作符

- 事件操作符： or

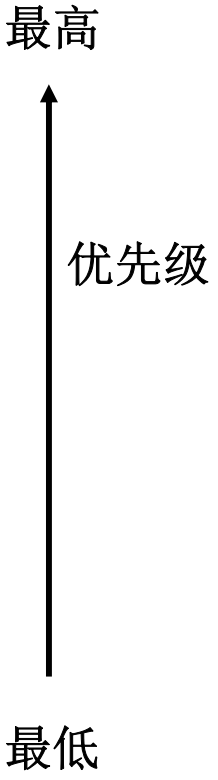
在行为级描述中使用

```
always @(a or b or c)


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

操作符的优先级

优先级

操作符类型	符号
连接及复制操作符	{}，　{{}}
一元操作符	+，-，！，~，&，|，^
算术操作符	*，　/，%
	+，　-
逻辑移位操作符	<<，　>>
关系操作符	>，　<，　>=，<=
相等操作符	= =，　= = =，!=，!= =
按位操作符	&，
	^，　~^，
	|
逻辑操作符	&&，
	||
条件操作符	?:



操作符的组织

- 除条件操作以外，操作符按照从左到右的方式组织。
- 优先级高的操作符先组织。
- 用”（ ）”可以改变操作符的优先级。

上海交通大學
Shanghai Jiao Tong University

组合逻辑与行为建模

组合逻辑的特点

- 组合逻辑在任何时刻，其输出由该时刻的输入决定，在组合逻辑中输入的改变经过一定的内部延时将传递到输出。

组合逻辑中的数据类型：　线网？寄存器？

组合逻辑中的赋值：　持续赋值

持续赋值与组合逻辑

- 持续赋值操作（assign）中，当表达式中的值改变时，立即更新目标值，与组合逻辑的特点符合，常常用来描述组合逻辑。

```
wire out,in1,in2;

assign out=in1 & in2;   //显式持续赋值

wire out2=in1 &in2;     //隐式持续赋值
```

常用组合电路的描述

反相器

```
module inverter (a,b);

input a;
output b;

wire a,b;

assign b=~a;

endmodule
```

定义线网类型

采用持续赋值

四输入与门

```
module and4 (a,b,c,d,y);

input a,b,c,d;
output y;
wire a,b,c,d,y,m1,m2;

assign y=m1 &m2;
assign m1=a&b;
assign m2=c&d;

endmodule
```

持续赋值语句是并发的，与其书写的顺序无关。

数据流模型

- 数据流模型主要用于组合逻辑的描述，采用assign语句进行赋值，通过布尔表达式能清楚地反映电路的功能结构与组合逻辑的数据特性。

```
module mux2 (f, a, b, sel );
output f;
input a,b,sel;

assign f = (a & ~sel) | (b & sel);
endmodule
```

```
module full_adder2
(a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```

行为模型

- 行为模型是一个模块怎样工作的一种抽象，更加关心模块的工作行为，而不是其具体实现，属于比较高的抽象层次。

- 行为模型通过过程语句来描述。

行为模型的基础——过程结构

- 两种过程结构：

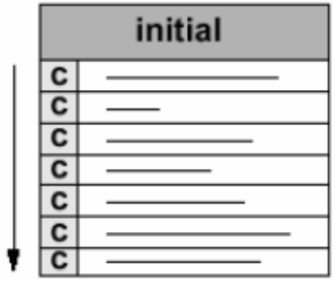
  initial结构（块）——单次执行

  always结构（块）——重复执行

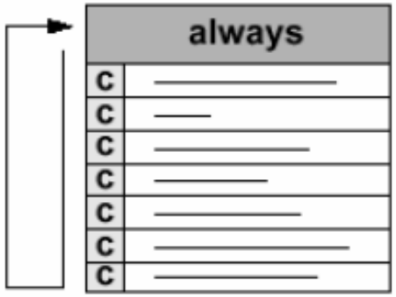
begin…end结构中语句顺序执行，只有一条语句时begin…end可以省略

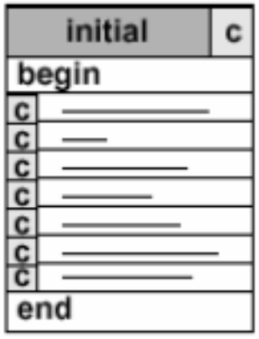
fork…join结构中语句的并行执行。

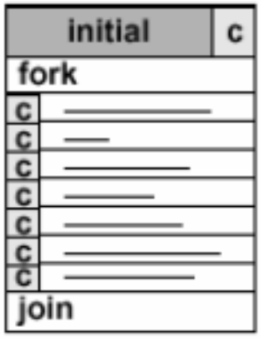
单次触发

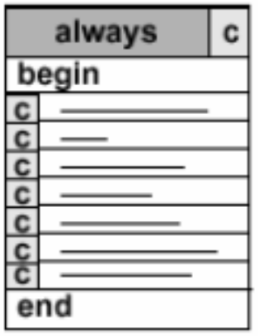


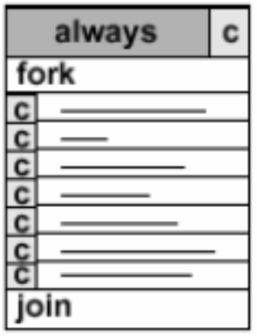
重复触发











过程块的语法介绍

- initial/always块的语法介绍（敏感列表...）

```
initial begin
a=1'b1;          //过程赋值
b=1'b0;
c=2'b1;
#2
a=1'b1;
b=1'b0;
c=2'b1;
end
```

```
module full_adder1( a, b, cin, sum,
cout);
input a, b, cin;
output sum, cout;
wire a, b, cin;
reg sum, cout;

always @(a or b or cin)      //敏感列表
begin
 {cout, sum} = a + b+ cin;     //过程赋值
end

endmodule
```

过程赋值

- 在过程块中的赋值称为过程赋值，在过程赋值语句中表达式左边的信号必须是寄存器类型（如**reg**类型），表达式右边类型没有限制（表达式左边信号没有声明会怎样？）。

```
module full_adder (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;
wire a,b,cin;
reg sum,cout;

always @(a or b or cin)
begin
   sum=a^b^cin;
   cout=(a&cin) | (b&cin) | (a&b);
   middle=a^b;
end
endmodule
```

如果一个信号没有声明则缺省为**wire**类型。使用过程赋值语句给**wire**类型变量赋值会产生错误。

过程赋值与持续赋值比较

过程赋值	持续赋值
在always或者initial语句中出现	在一个模块内出现
执行与周围其他语句有关	与其它语句并行执行，在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
使用”=“或者”<= “	使用“=“赋值
无assign关键词（过程性持续赋值中除外）	有assign关键词

过程块的时序控制

- 过程块时序控制有两类：
  - 时延控制

```
always #5 clk=~clk;
```

```
initial begin
clk=0;
#5 clk=1;
end
```

  - 事件控制
    - 边沿触发事件控制@(<signal>)

```
always @(a or b or c)      //在信号发生翻转时执行
…
always @(posedge clk or negedge rst_n)   //可以指定翻转的沿
…
```

    - 电平敏感事件控制

```
always wait(a)
…
always wait(clk)
…
```

过程块的分支控制（if..else)

- If...else...控制

描述方式：
if (表达式)
begin
.....
end
else if (表达式)
begin
.....
end
else
begin
...
end

```
always @(a or b)
begin
if(a)
c=1'b1;
if(b)
c=1'b0;
end
```

要避免的写法

```
always @(a or b or c)
begin
if({a,b}==2'b10)
c=1'b1;
else if({a,b}==2'b01)
c=1'b0;
else if({a,b}==2'b11)
c=1'b0;
else
c=1'b1;
end
```

语法上和其他高级语言类似，编写代码时要养成良好的习惯：

考虑所有的分支情况；

尽量少地使用嵌套；

过程块的分支控制（case）

- case语句

```
case （操作数）
条件1：表达式1；
条件2：表达式2；
条件3：表达式3；
条件4：表达式4；
…
default: 表达式n
endcase
```

```
……
always @(a or b)
begin
case({a,b})
2’b10:c=1’b1;
2’b01:c=1’b0;
default:c=1’b1;
endcase
end
……
```

依次对于分支项求值，第一个与条件表达式匹配的分支项被执行：

考虑所有的分支情况，分支不全时要使用default；

Verilog主要建模层次比较

- 行为级模型
  - 高层次的抽象
  - 与硬件的关系最远，
  - 和人的想法最接近
- 数据流模型
  - 用布尔表达式描述的逻辑
- 门级模型
  - 内置基本门（或者用户定义原语UDP）实现的电路

全加器的三种描述——行为模型

行为模型

```
module full_adder1( a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
wire a, b, cin;
reg sum, cout;
always @(a or b or cin)
 {cout, sum} = a + b+ cin;

endmodule
```

数据流模型

```
module full_adder2 (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```

```
module full_adder3 (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;
wire p,m1,m2,m3;

xor (p,a,b);
xor (sum,p,cin);
and (m1,a,cin);
and (m2,b,cin);
and (m3,a,b);
or (cout,m1,m2,m3);

endmodule
```

门级模型

四选一mux的两种模型

行为模型

```
module MUX_4_1 (a,b,c,d,sel,op);
input a,b,c,d;
input [1:0] sel;
output op;
wire a,b,c,d;
wire [1:0] sel;
reg op;
always @(sel or a or b or c or d)
begin
case(sel)
2'b00: op = a;
2'b01: op = b;
2'b10: op = c;
2'b11: op = d;
endcase
end
endmodule
```

数据流模型

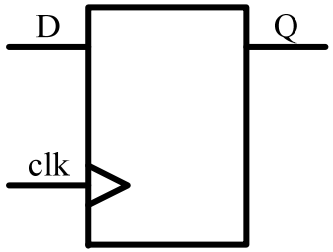
```
module MUX_4_1 (a,b,c,d,sel,op);

input a,b,c,d;
input [1:0] sel;

output op;

wire a,b,c,d,op;

wire [1:0] sel;

assign op =(sel[1]==1)
            ?(sel[0]==1?d:c)
            :(sel[0]==1?b:a);


endmodule
```

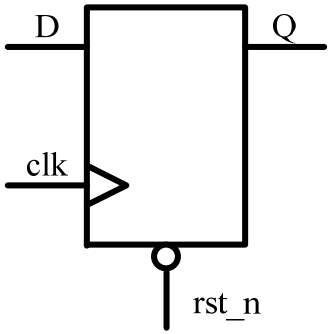
上海交通大學
Shanghai Jiao Tong University

Verilog HDL 硬件描述语言
时序电路与Testbench设计

典型的时序逻辑——DFF

- D flip-flop



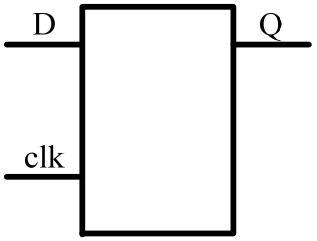



```
reg q;
always @(posedge clk)

    q<=d;
```

```
reg q;
always @(posedge clk or negedge rst_n)
if(!rst_n)
    q<=1'b0;
else
    q<=d;
```

典型的时序逻辑——TLAT

- D latch




```
reg q;
always @(clk or d)
if(clk)
      q<=d;
else
       q<=q;
```

阻塞赋值与非阻塞过程赋值

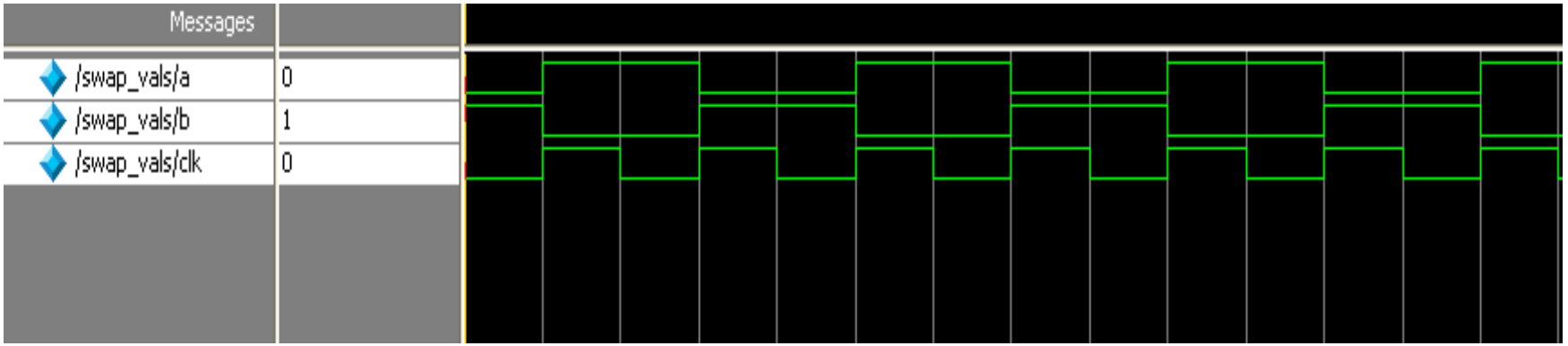
- 阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。
- 非阻塞过程赋值对于目标的赋值是非阻塞的，在某个时间步同步发生。

```
module swap_vals;
reg a, b, clk;
initial begin
a = 0; b = 1; clk = 0;
end
always #5 clk = ~clk;
always @( posedge clk)
begin
a <= b; // 非阻塞过程赋值
b <= a; //
end
endmodule
```

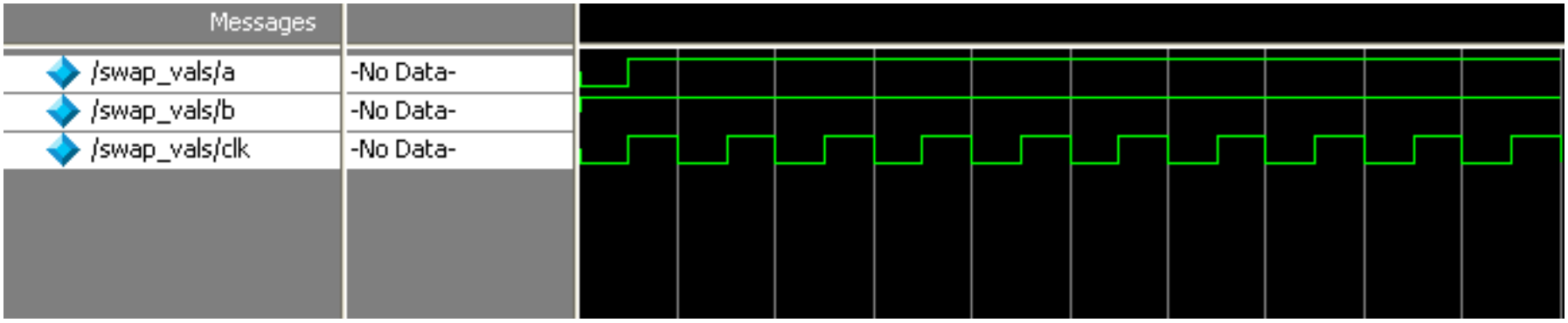
```
module swap_vals;
reg a, b, clk;
initial begin
a = 0; b = 1; clk = 0;
end
always #5 clk = ~clk;
always @( posedge clk)
begin
a = b; // 阻塞过程赋值
b = a; //
end
endmodule
```

阻塞与非阻塞波形比较

非阻塞



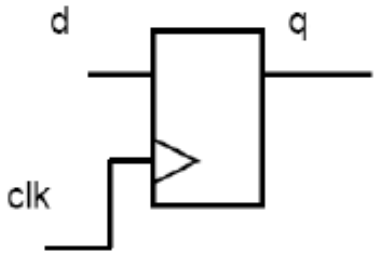
阻塞



阻塞与非阻塞电路举例

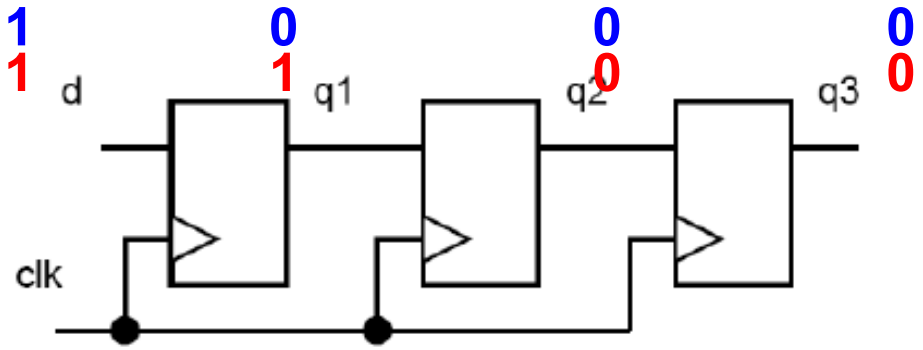
```
module pipeline1(clk,d,q);
input clk,d;
output q;

reg q1,q2,q3;
wire clk,d,q;

always @(posedge clk)
begin
q1=d;
q2=q1;
q3=q2;
end
assign q=q3;
endmodule
```

?



推荐的写法

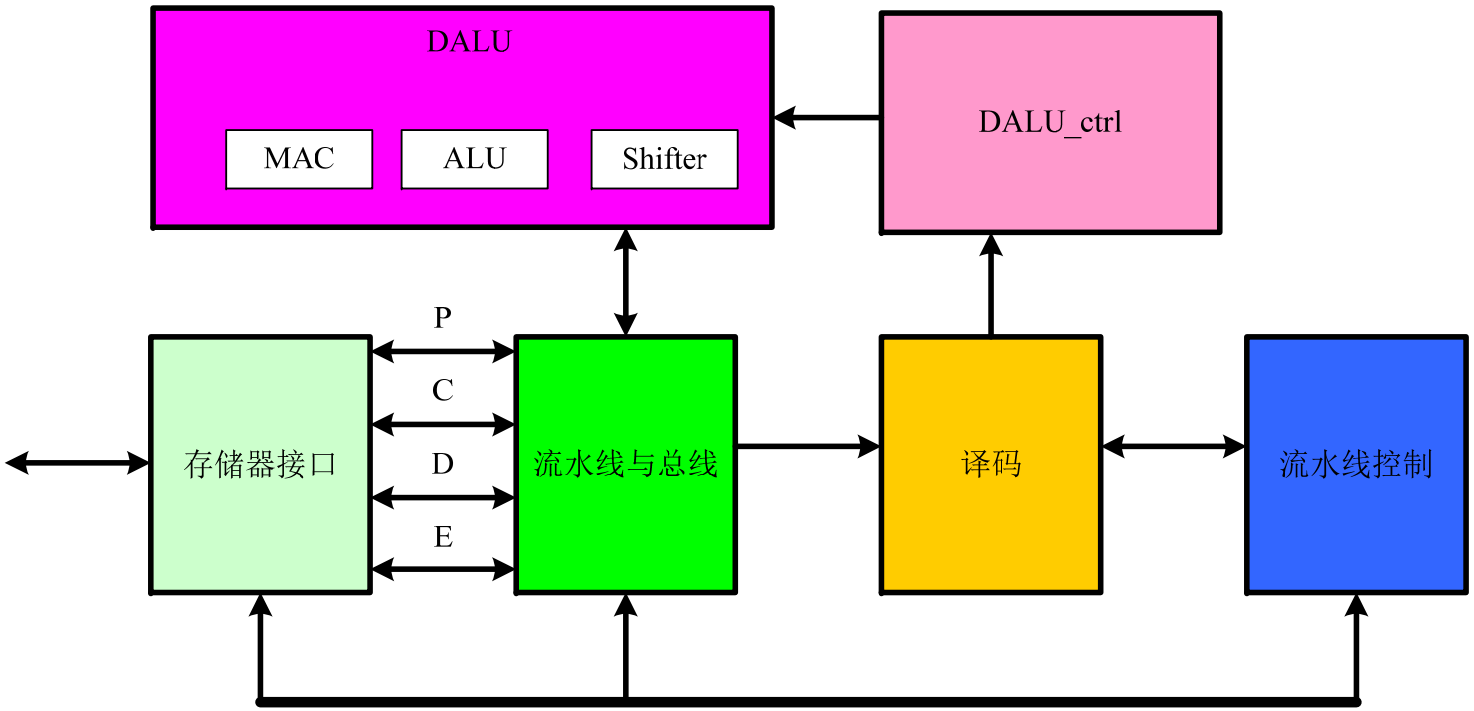
```
module pipeline1(clk,d,q);
input clk,d;
output q;

reg q1,q2,q3;
wire clk,d,q;

always @(posedge clk)
begin
q1<=d;
q2<=q1;
q3<=q2;
end
assign q=q3;
endmodule
```



代码准则

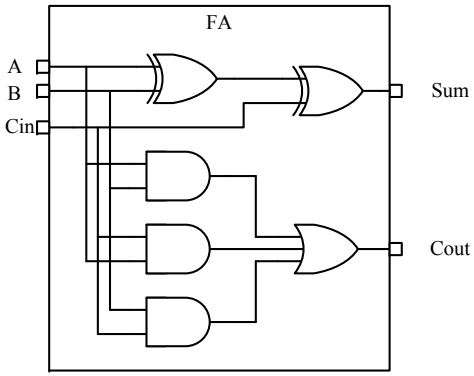
- 建模时序逻辑时使用非阻塞赋值。
- 使用always块描述组合逻辑，使用阻塞赋值。
- 不要在一个always块中同时使用阻塞和非阻塞。

结构化设计

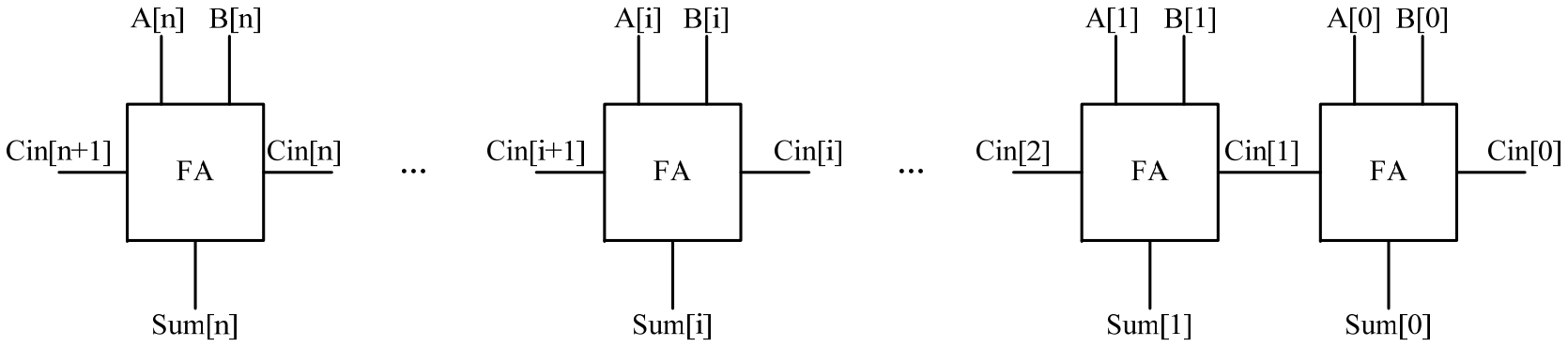


结构化设计举例

- 模块的例化




```
module full_adder (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```



模块例化

- 例化过程是一个模块在另一个模块的引用过程，通过例化建立层次结构的描述方式。

```
//模块名        例化名（端口连接）
full_adder fa0(a[0],b[0],c0,sum[0],cin[1]);//位置关联


full_adder fa0(   .a(a[0])
                  ,.b(b[0])
                  ,.cin(c0)
                  ,.sum(sum[0])
                  ,.cout(cin[1]));              //名称关联
```

同一个模块可以有多个不同的例化单元，它们之间通过例化名区分。

端口连接可以通过位置关联和名称关联来实现。

例化过程中应当注意端口的定义与连接

模块端口与连接规则

- 输入端口（input）
  ---输入端口的类型不能是reg类型。

```
reg d1,d2,d3,d4,d5;
full_adder fa1(    .a(d1)
                  ,.b(d2)
                  ,.cin(d3)
                  ,.sum(d4)
                  ,.cout(d5));
```

左边连接正确与否???

- 输出端口（output）
  ---输出端口可以是reg类型或wire类型

- 输入输出端口（inout）
  ---输入输出端口不能是reg类型

模块端口

```
module full_adder (a,b,cin,sum,cout);

input a,b,cin;
output sum,cout;

wire a,b,cin,sum,cout;

assign sum=a^b^cin;
assign cout=(a&cin) | (b&cin) | (a&b);

endmodule
```

模块端口，用于外部连接

模块端口

内部端口，指定类型，宽度等

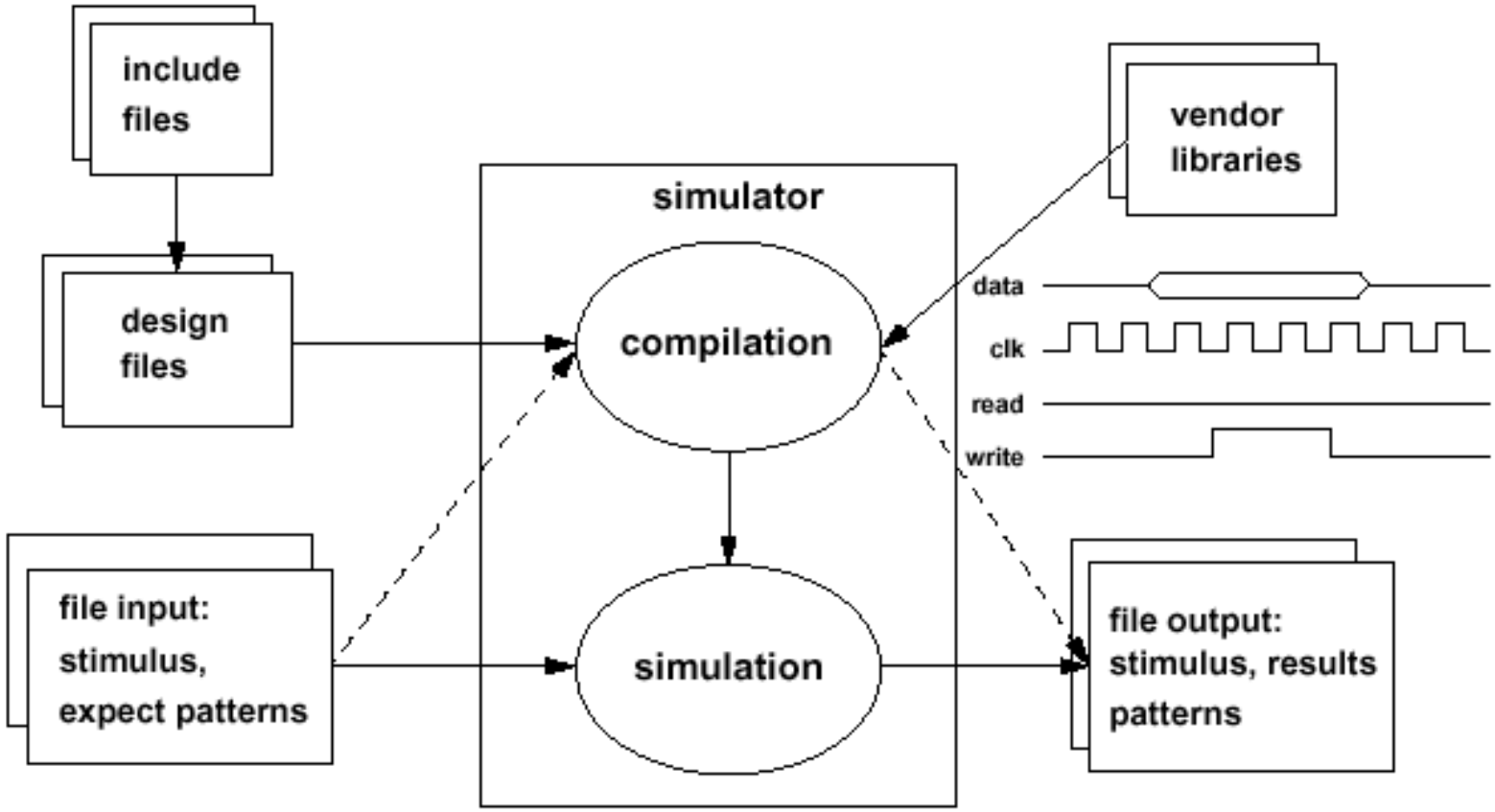
结构化设计举例

```
`include "full_adder.v"
module adder_4(a,b,c0,sum,carry);
input [3:0]   a,b;
input         c0;
output        carry;
output [3:0] sum;
wire[3:0]     a,b,sum;
wire[3:1]     cin;
wire          c0,carry;
full_adder fa0(a[0],b[0],c0,sum[0],cin[1]);
full_adder fa1(a[1],b[1],cin[1],sum[1],cin[2]);
full_adder fa2(a[2],b[2],cin[2],sum[2],cin[3]);
full_adder fa3(a[3],b[3],cin[3],sum[3],carry);
//full_adder fa0(.a(a[0]),.b(b[0]),.cin(c0),.sum(sum[0]),.cout(cin[1]));
//full_adder fa1(.a(a[1]),.b(b[1]),.cin(cin[1]),.sum(sum[1]),.cout(cin[2]));
//full_adder fa2(.a(a[2]),.b(b[2]),.cin(cin[2]),.sum(sum[2]),.cout(cin[3]));
//full_adder fa3(.a(a[3]),.b(b[3]),.cin(cin[3]),.sum(sum[3]),.cout(carry));
endmodule
```

模块例化时的注意事项

- 尽量使用名字关联
- 注意端口位宽的一致性
- 检查端口连接规则
- 不要出现没有连接的端口
- 一般不需要显式声明模块端口

Testbench设计



虚线表示编译时检测输入文件是否存在及可读并允许生成输出文件。

简单Testbench设计

```
`timescale 1ns/10ps          //编译指导，规定时间单位与精度
`include “full_addder.v”     //编译指导，包含所需文件
module testbench ();         //testbench
wire sum_out;
reg a_input, b_input, cin;                  //信号定义
full_adder fa1(    .a(a_input)              //模块例化
                  ,.b(b_input)
                  ,.cin(cin)
                  ,.sum(sum_out)
                  ,.cout(cout));
initial begin                                        //激励产生
       a_input=0;b_input=0;cin=0;
#10  a_input=0;b_input=0;cin=1;
#15  a_input=1;b_input=1;cin=0;
#25   $finish;                                       //结束仿真
end
endmodule
```

模块实例化(module instantiation)

- 模块实例化时实例必须有一个名字。
- 使用位置映射时，端口次序与模块的说明相同。
- 使用名称映射时，端口次序与位置无关
- 没有连接的输入端口初始化值为x。

```
module comp (o1, o2, i1, i2);
      output   o1, o2;
      input   i1, i2;
      . . .
endmodule

module test;
      comp c1 (Q, R, J, K); // Positional mapping
      comp c2 (.i2(K),   .o1(Q),  .o2(R),  .i1(J)); //
Named mapping
      comp c3 (Q,   ,  J,  K);        // One port left
unconnected
      comp c4 (.i1(J),  .o1(Q));   // Named, two
unconnected ports
endmodule
```

没有连接时通常会产生警告

名称映射的语法：

.内部信号（外部信号）

施加激励

- 产生激励并加到设计有很多 种方法。一些常用的方法有:
  - 从一个initial块中施加线激励
  - 从一个循环或always块施加激励
  - 从一个向量或存储器施加激励

建立时钟

有启动初始值的对称时钟

```
reg clk;
initial  begin
        clk = 0;
forever  #( period/2) clk
= !clk;
end
```

```
reg clk
always #( period/2) clk=~clk;
initial
begin
            clk = 0;
end
```

产生的波形（假定**period**为20）

Testbench中常用的系统函数

- Verilog提供了内置的系统任务与函数
  - 显示任务（$display, $monitor)
  - 文件输入输出
  - 时间函数($time,返回当前的系统仿真时间)
  - 其他验证与分析高级任务

输出格式化时间

```
`timescale 1 ns / 10 ps
module top;
      reg in1;
      not #9.53 n1 (o1, in1);
      initial
      begin
          $display("time realtime stime \t in1 \t o1 ");
          $timeformat(-9, 2, "ns", 10);
          $monitor("%d %t %d \t %b \t %b", $time,
$realtime, $stime, in1, o1);
              in1 = 0;
          #10 in1 = 1;
          #10 $finish;
      end
endmodule
```

```
time realtime stime in1  o1
0      0.00ns    0    0    x
10     9.53ns    10   0    1
10     10.00ns   10   1    1
20     19.53ns   20   1    0
```

结束仿真与记录波形

```
initial
begin
$dumpfile(“file_name.vcd”);          //生成文件
$dumpvas(0);                         //记录信号
#time_delay $finish;                 //结束仿真
end
```

在**modelsim**中**$finish** 会关闭软件，**window**环境中谨慎使用

完整的testbench

```
`timescale 1ns/10ps
`include "C:/Modeltech_6.3c/examples/full_adder.v"
module testbench ();
wire sum_out;
reg a_input, b_input, cin;
full_adder fa1(    .a(a_input)
                  ,.b(b_input)
                  ,.cin(cin)
                  ,.sum(sum_out)
                  ,.cout(cout));
initial begin
    a_input=0;b_input=0;cin=0;
#10  a_input=0;b_input=0;cin=1;
#15  a_input=1;b_input=1;cin=0;
end
initial begin
$monitor("%b,%b,%b,%b,%b",a_input,b_input,cin,sum_out,cout);
#30 $finish;
end
endmodule
```

上海交通大學
Shanghai Jiao Tong University

Verilog HDL 硬件描述语言
高级设计话题

可综合RTL代码设计

- 可综合设计语法是Verilog语言的一个子集，使不同的RTL综合工具可以一致地实现相同的结果，并保证与仿真结果的相同。

- IEEE制定了专门的标准来规范Verilog语言的可综合设计，IEEE Standard for Verilog® Register Transfer Level Synthesis（IEEE1364.1-2002）（www.ieee.org)

可综合语法结构

Construct Type	Keyword or Description	Notes
ports	input, inout, output	inout应该在顶层的IO使用
parameters	parameter	设计更有继承性
module definition	module	
signals and variables	wire, reg, tri	支持向量
instantiation	module instances	避免使用基本门
function and tasks	function , task	忽略延时信息
procedural	always, if, else, case, casex, casez	不支持initial
procedural blocks	begin,end,namedblocks, disable	支持命名块的disable操作
data flow	assign	延时信息被忽略
named Blocks	disable	支持disable
loops	for, while, forever	While 和 forever 必须包含 @(posedge clk) 或者 @(negedge clk)

不可综合的语法结构

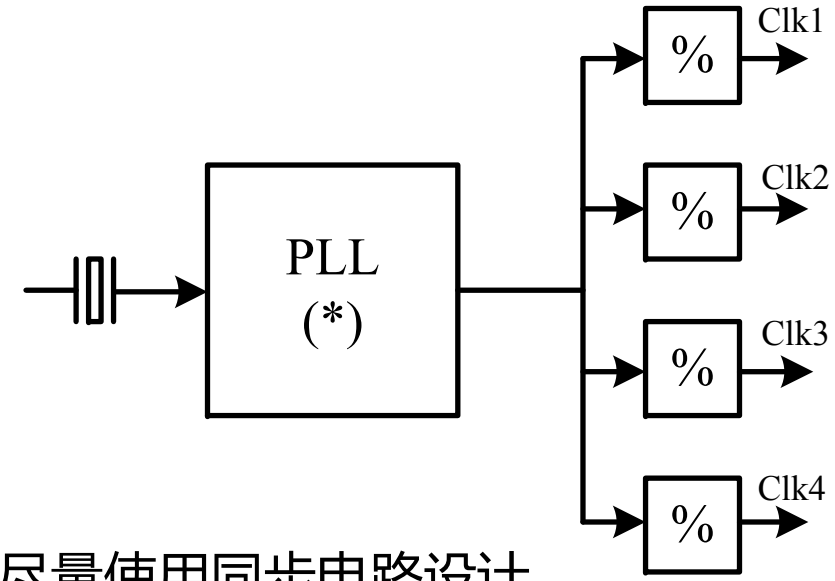
Construct Type	Notes
initial	只可用在Testbench设计中
events	用在Testbench设计中
real	不支持real类型
time	不支持time类型
force and release	不支持强制操作
assign deassign	不支持对于寄存器的强制操作
fork join	不支持，可以用非阻塞赋值实现相同的功能
primitives	不支持，多用于库单元描述
table	不支持，多用于库单元描述

不支持以下基本单元的描述： nmos， pmos，cmos，rnmos，rpmos，rcmos， pullup， pulldown， rtran， tranif0，tranif1， rtranif0， rtranif1

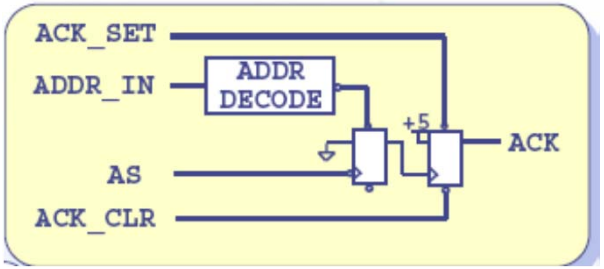
不支持===与！==操作

使用同步设计

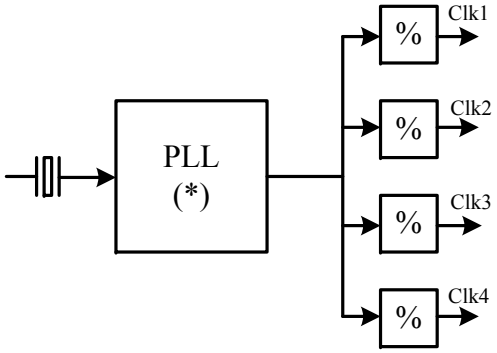
- 尽量使用一个时钟，或者同一个时钟源的时钟。

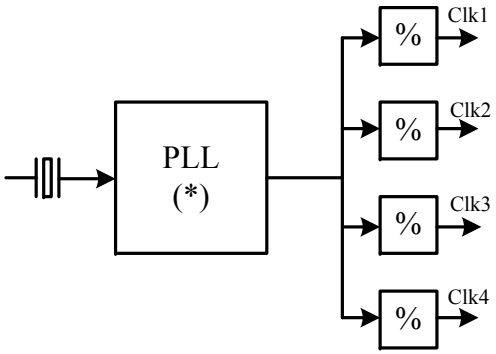


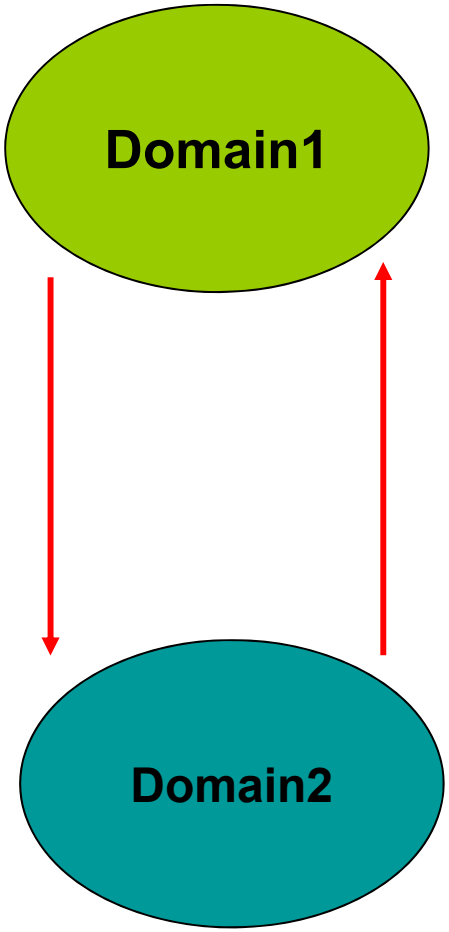
同步电路可以稳定的工作，尽量使用同步电路设计。



什么是异步设计







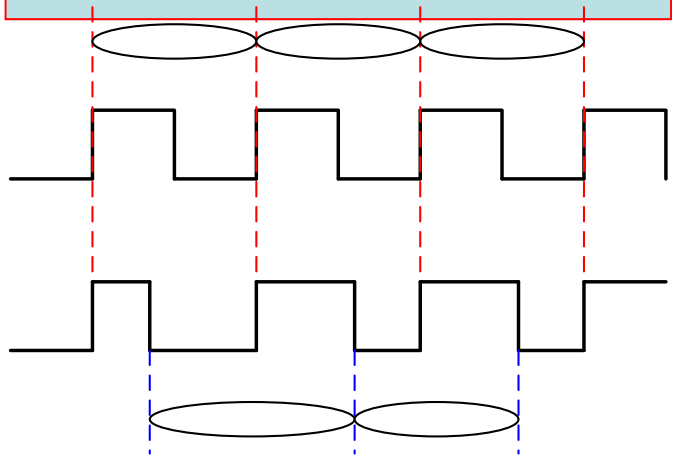
异步设计一般需要特殊的电路，应尽量避免。

使用上升沿设计

- 在设计中，使用一个时钟沿工作。

好的风格

```
…
always @(posedge clk)
begin
…
end
```



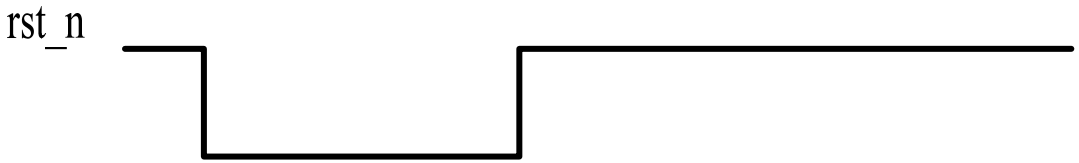
无法保证duty cycle，要避免的设计

```
…
always @(posedge clk)
begin
…
end

always @(negedge clk)
begin
…
end
```

电路reset

- 在电路设计中要求有一个初始态，在电路工作前初始化所有的寄存器（组合逻辑的输入可以溯源自寄存器的输出）。

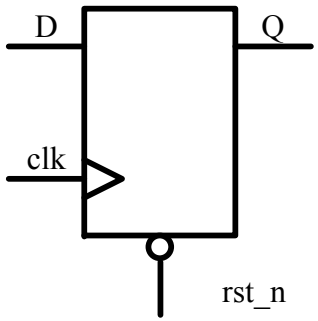
- 异步reset



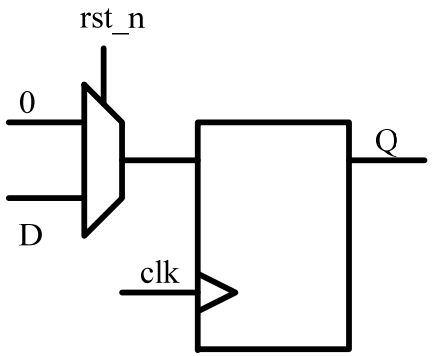
- 同步reset

异步与同步reset

```
…
always @(posedge clk or negedge rst_n)
If(~rst_n)
…
else
…
```




```
…
always @(posedge clk)
If(~rst_n)
…
else
…
```



ModelSim的简单使用

HDL 仿真器

- HDL仿真器模拟硬件电路的行为，对于HDL描述的硬件电路在软件上进行仿真调试。
- 桌面仿真软件
  - Modelsim (mentor graphic)
  - Questasim(mentor graphic)
- 企业级的仿真软件
  - NC-verilog(cadence)
  - VCS(synopsys)

ModelSim简介

- 由Model技术公司开发
- 工业上最通用的仿真器之一

- 交互式的命令行 (Cmd)
  - 唯一的界面是控制台的命令行, 没有用户界面
- 用户界面 (UI)
  - 能接受菜单输入和命令行输入
  - 课程主要讨论
- 批处理模式
  - 从DOS或UNIX命令行运行批处理文件
  - 不讨论

基本步骤

1 ⇨ 建立库

2 ⇨ 映射库到物理目录

3 ⇨ 编译源代码

- 所有的HDL代码必须被编译
- Verilog和VHDL是不同的

4 ⇨ 启动仿真器

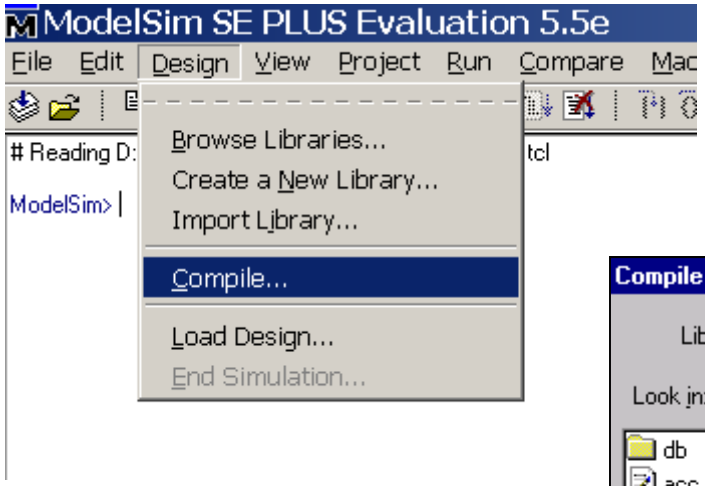
5 ⇨ 执行仿真

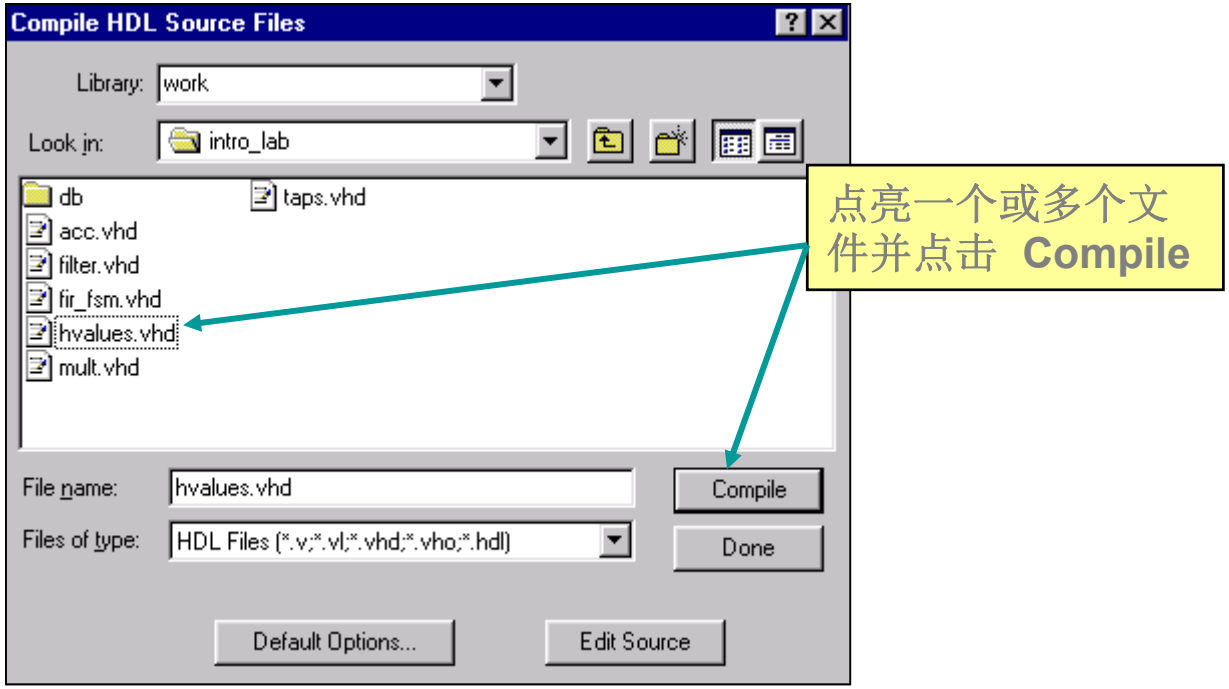
3 ⇨ 编译源代码(Verilog)

- UI) ***Design -> Compile***
- Cmd) *vlog* -work <library_name> <*file1*>.v <*file2*>.v
  - 文件按出现的顺序被编译
  - 文件的顺序或者编辑的顺序不重要
- 支持增量式编译(只有被改动的设计单元被编译)
- 缺省编译到work库
  - 例如. *vlog* my_design.v

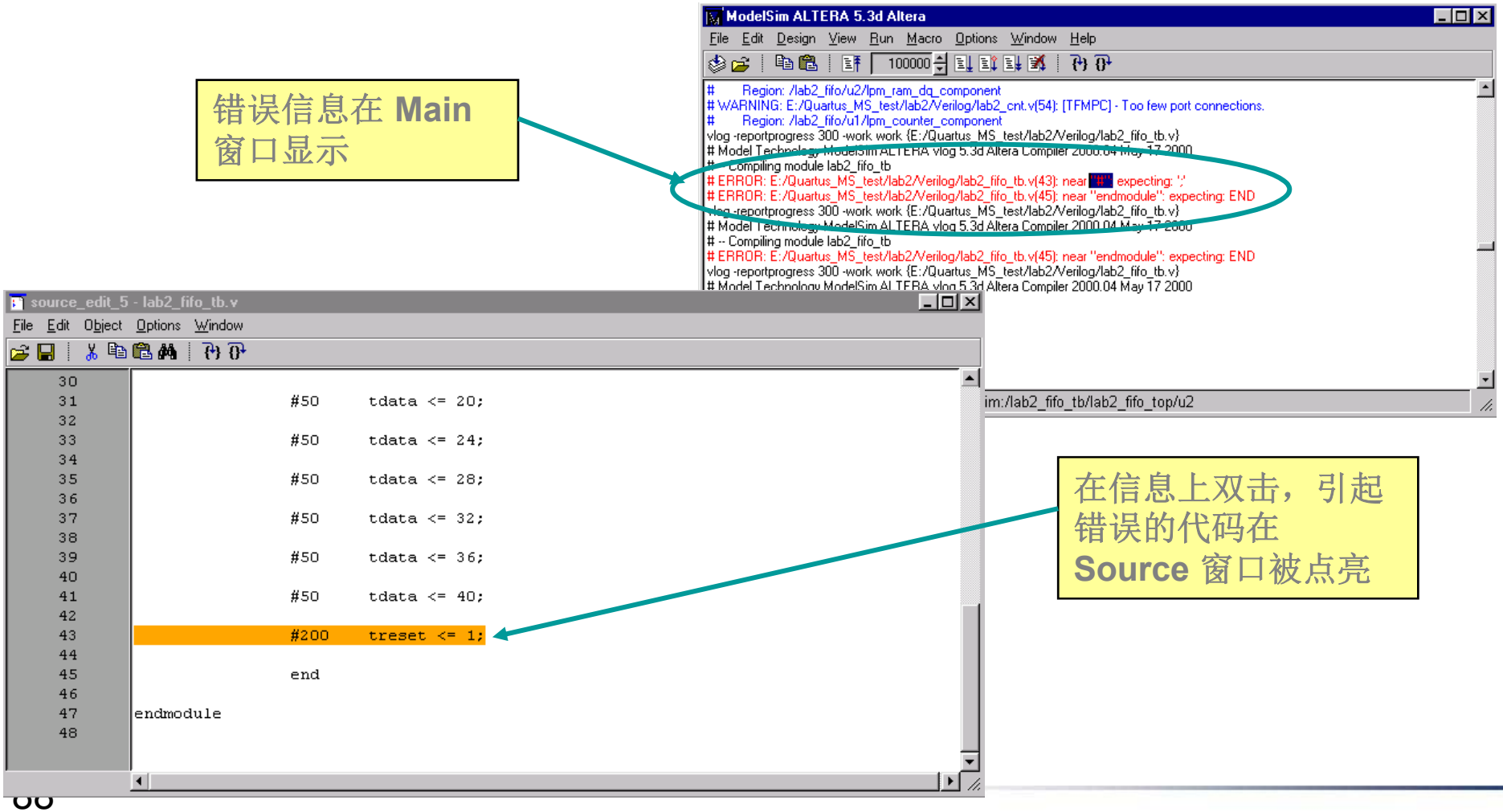
用ModelSim作功能仿真

3 ⇨ 编译源代码





3 ⇨ 编译源代码----错误信息


错误信息在 Main 窗口显示
ModelSim ALTERA 5.3d Altera
File Edit Design View Run Macro Options Window Help
100000
Region: /lab2_fifo/u2/lpm_ram_dq_component
WARNING: E:/Quartus_MS_test/lab2/Verilog/lab2_cnt.v(54): [TFMPC] - Too few port connections.
Region: /lab2_fifo/u1/lpm_counter_component
vlog -reportprogress 300 -work work {E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v}
Model Technology ModelSim ALTERA vlog 5.3d Altera Compiler 2000.04 May 17 2000
Compiling module lab2_fifo_tb
ERROR: E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v(43): near "#": expecting: ';'
ERROR: E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v(45): near "endmodule": expecting: END
vlog -reportprogress 300 -work work {E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v}
Model Technology ModelSim ALTERA vlog 5.3d Altera Compiler 2000.04 May 17 2000
-- Compiling module lab2_fifo_tb
ERROR: E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v(45): near "endmodule": expecting: END
vlog -reportprogress 300 -work work {E:/Quartus_MS_test/lab2/Verilog/lab2_fifo_tb.v}
Model Technology ModelSim ALTERA vlog 5.3d Altera Compiler 2000.04 May 17 2000
im:/lab2_fifo_tb/lab2_fifo_top/u2
source_edit_5 - lab2_fifo_tb.v
File Edit Object Options Window
30
31 #50 tdata <= 20;
32
33 #50 tdata <= 24;
34
35 #50 tdata <= 28;
36
37 #50 tdata <= 32;
38
39 #50 tdata <= 36;
40
41 #50 tdata <= 40;
42
43 #200 treset <= 1;
44
45 end
46
47 endmodule
48
在信息上双击，引起错误的代码在 Source 窗口被点亮

用ModelSim作功能仿真

4 ⇨ 启动仿真器

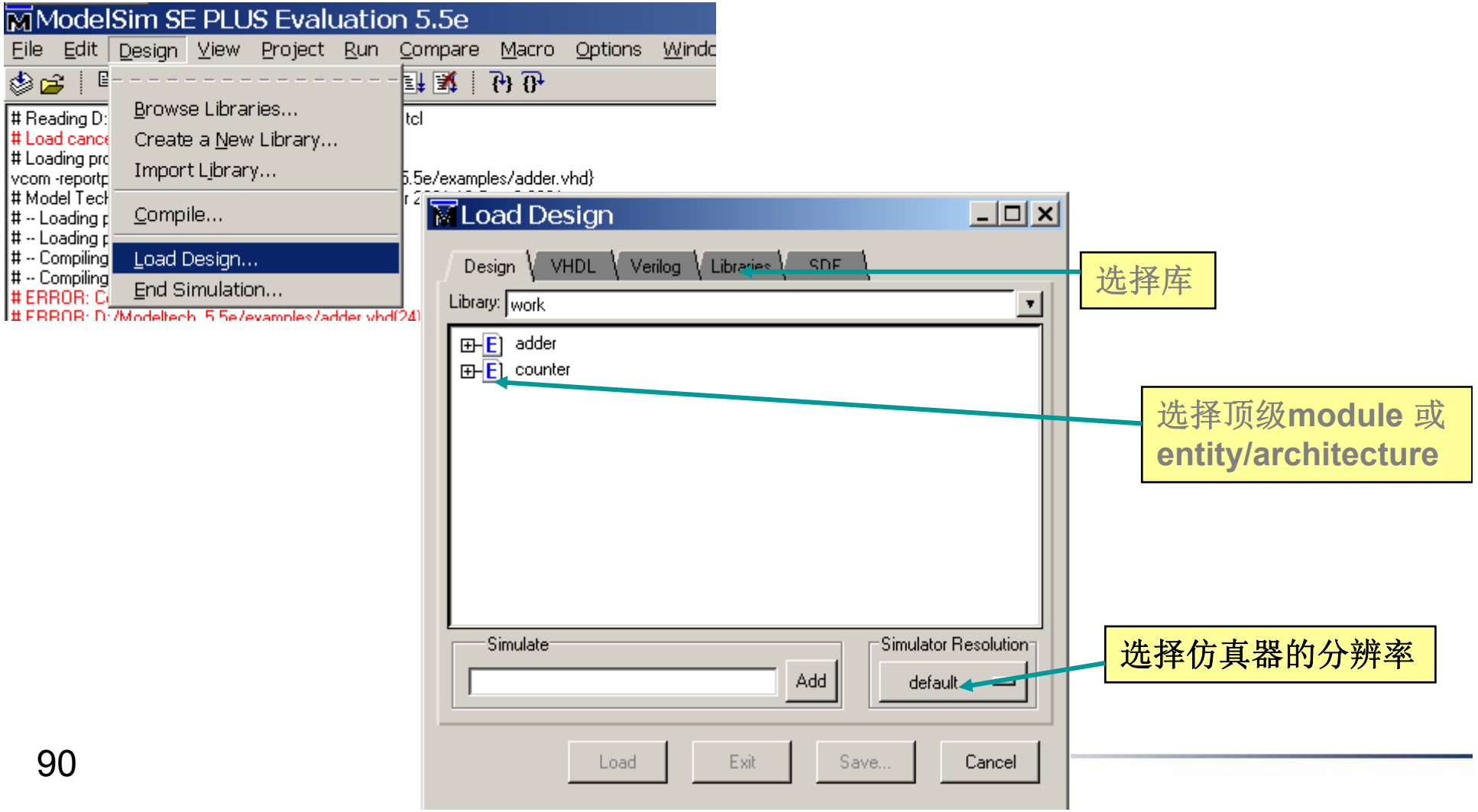
- UI) ***Design -> Load New Design***
  Cmd) *vsim* -lib <library_name> <top_level_design>

- VHDL
  - *vsim* top_entity top_architecture

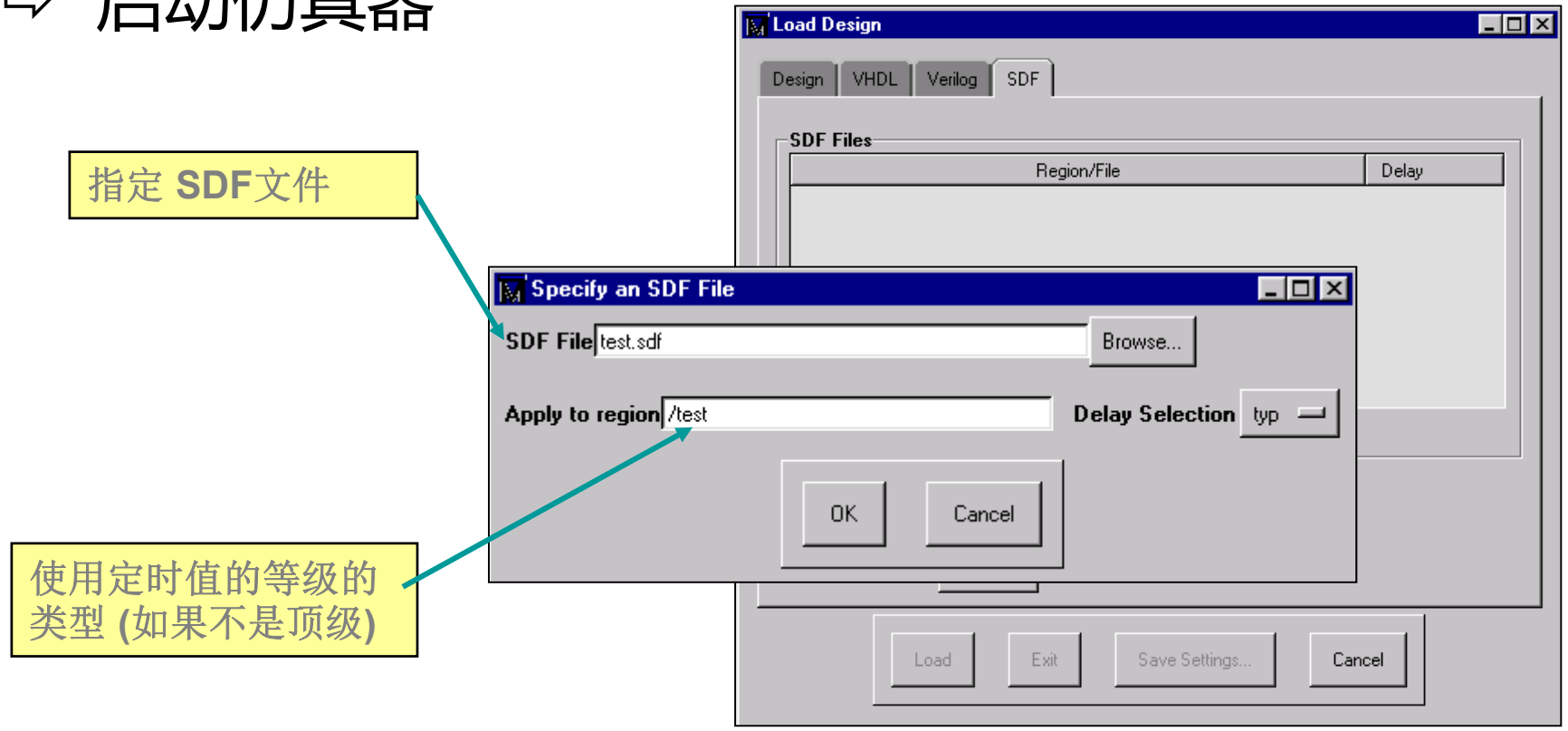
- Verilog
  - *vsim* top_level1 top_level2
    - 仿真多个top级模块

用ModelSim作功能仿真

4 ⇨ 启动仿真器



4 ⇨ 启动仿真器


指定 SDF文件
使用定时值的等级的类型 (如果不是顶级)
Load Design
Design
VHDL
Verilog
SDF
SDF Files
Region/File
Delay
Specify an SDF File
SDF File
test.sdf
Browse...
Apply to region
/test
Delay Selection
typ
OK
Cancel
Load
Exit
Save Settings...
Cancel

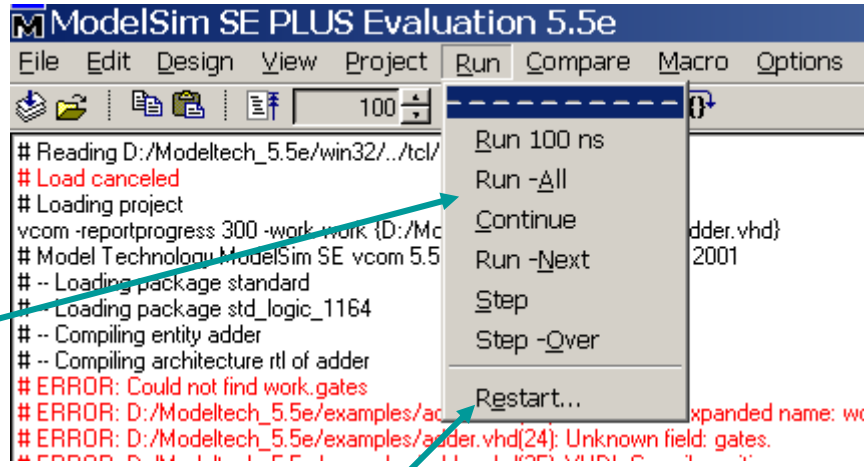
5 ⇨ 执行仿真

❖ UI) ***Run***

CMD) *run* <time_step> <time_units>

用ModelSim作功能仿真

5 ⇨ 执行仿真(UI)



选择 **timesteps**数量就可以执行仿真

Restart **–** 重装任何已改动的设计元素并把仿真时间设为零

COM) restart

5 ⇨ 执行仿真----仿真器激励

- 测试台
  - Verilog 或 VHDL代码
  - 非常复杂的仿真（交互式仿真、数据量大的仿真）
- force命令
- DO 文件 (宏文件)

5 ⇨ 执行仿真----仿真器激励

Force命令

- 允许用户给VHDL的信号和Verilog的线网予以激励
- 常规语法:
  - *force* <item_name> <value> <time>, <value> <time>

5 ⇨ 执行仿真----仿真器激励

.do文件

- 自动完成仿真步骤的宏文件
  - 库设置
  - 编译
  - 仿真
  - 强制仿真激励
- 能在所有的ModelSim 模式里被调用
  - UI) ***Macro -> Execute***
  - COM) *do* <filename>.do
- 能调用其他的DO文件

5 ⇨ 执行仿真----仿真器激励

.do文件举例

```
cd c:\mydir
vlib work
vcom counter.vhd
vsim counter
view *
add wave /*
add list /*
do run.do
```

my_sim.do

```
cd c:\mydir
vlib work
vcom counter.vhd
vsim counter
view *
do stimulus.do
```

stimulus.do

```
add wave /clk
add wave /clr
add wave /load
add wave -hex /data
add wave /q
force /clk 0 0, 1 50 -repeat 100
force /clr 0 0, 1 100
run 500
force /load 1 0, 0 100
force /data 16#A5 0
force /clk 0 0, 1 50 -repeat 100
run 1000
```

5 ⇨ 执行仿真----仿真器激励

测试台文件(test bench)

- 针对复杂的仿真
- VHDL文件或者Verilog
- 在测试台文件中将设计模块实例化

  --将测试台文件置于TOP层,调用设计模块

  --在测试台文件中加载时钟激励信号,以及给部分信号赋初值
- 测试台文件的写法与设计模块写法有区别

  -- 一些符合语法但又无法被综合的语句(根据具体的综合工具而定),可以在测试台文件中使用

modelsim.ini文件

- ModelSim使用 ASCII文件, 由用户控制
- 在ModelSim的安装目录一个缺省文件被提供
- modelsim.ini被编译器和仿真器使用

  存有初始信息
  库定位
  启动文件的定位
  ModelSim其他缺省设定
  [Library]：逻辑上的LIBRARY与实际硬盘驱动目录的连接
  [vcom]：COMPILE时的一些选项的默认值，0=OFF，1=ON
  [vsim]：仿真时参数的设定

- modelsim.ini缺省为只读属性，故编译库文件时应该将只读属性去掉。

Thanks