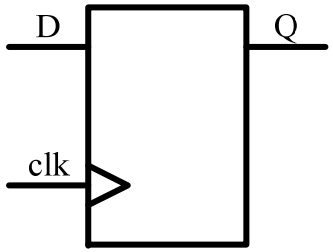
上海交通大学
Shanghai Jiao Tong University

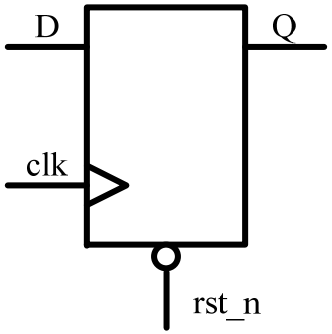
Verilog HDL 硬件描述语言（3）
时序电路与Testbench设计

上海交通大学微电子学院
蒋剑飞

典型的时序逻辑——DFF

- D flip-flop



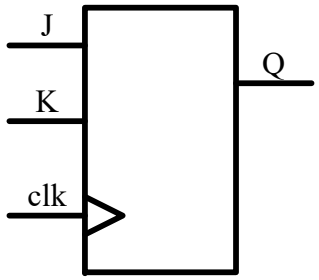



```
reg q;
always @(posedge clk)

    q<=d;
```

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

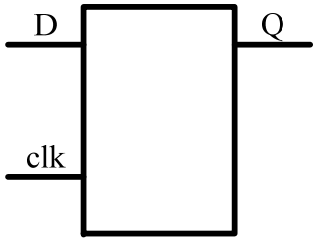
典型的时序逻辑——JKFF/TLAT

- JK flip-flop




```
reg q;
always @(posedge clk)
if(j&k)
      q<=!q;
else if (j&!k)
       q<=1'b1;
else if (!j&k)
       q<=1'b0;
else
       q<=q;
```

- 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

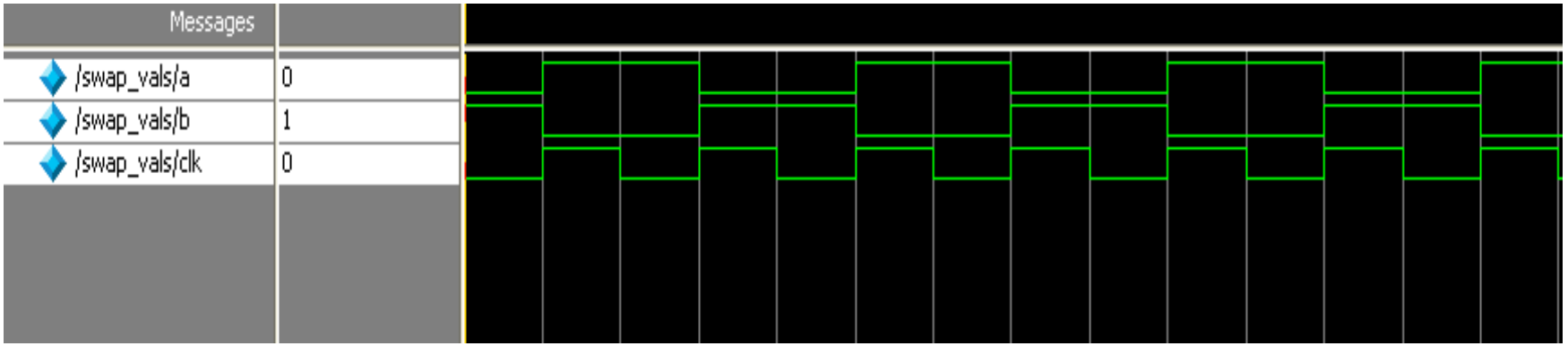
- 阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。
- 非阻塞过程赋值对于目标的赋值是非阻塞的，在某个时间步同步发生。

```
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
```

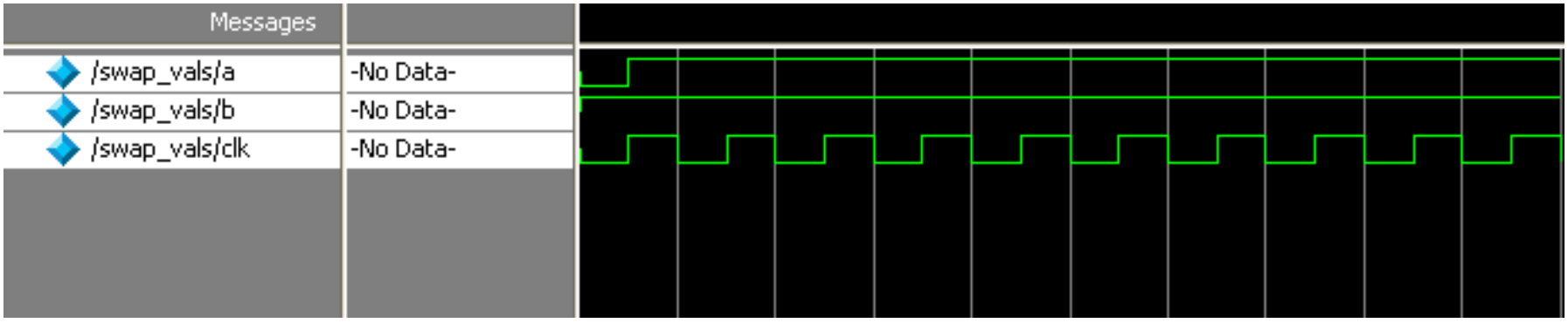
```
module swap_vals;
reg a, b, clk;
initial begin
a = 0; b = 1; clk = 0;
end
always #5 clk = ~clk;
always @( posedge clk)
begin
a = b; // 阻塞过程赋值
b = a; //
end
endmodule
```

阻塞与非阻塞波形比较

非阻塞



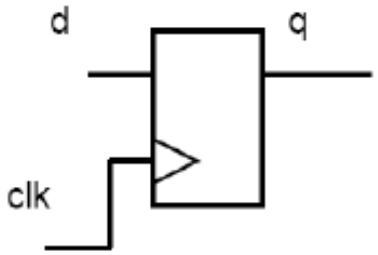
阻塞



阻塞与非阻塞电路举例

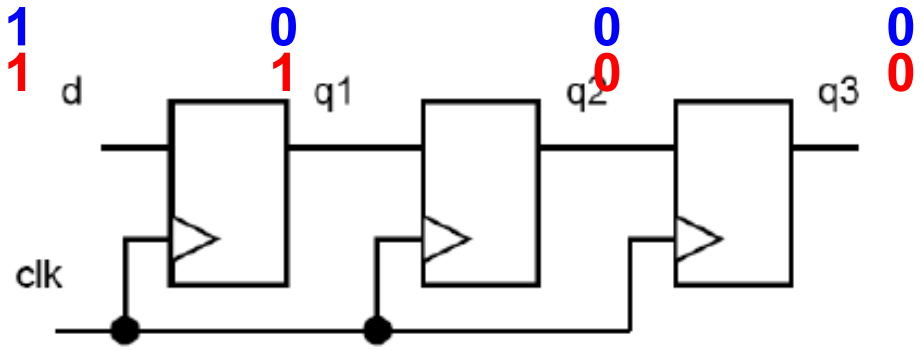
```
module pipeline1(clk,d,q);
input clk,d;
output q;

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
```



Blocking VS Non-Blocking

```
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
```

```
module pipeline2(clk,d,q);
input clk,d;
output q;

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

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

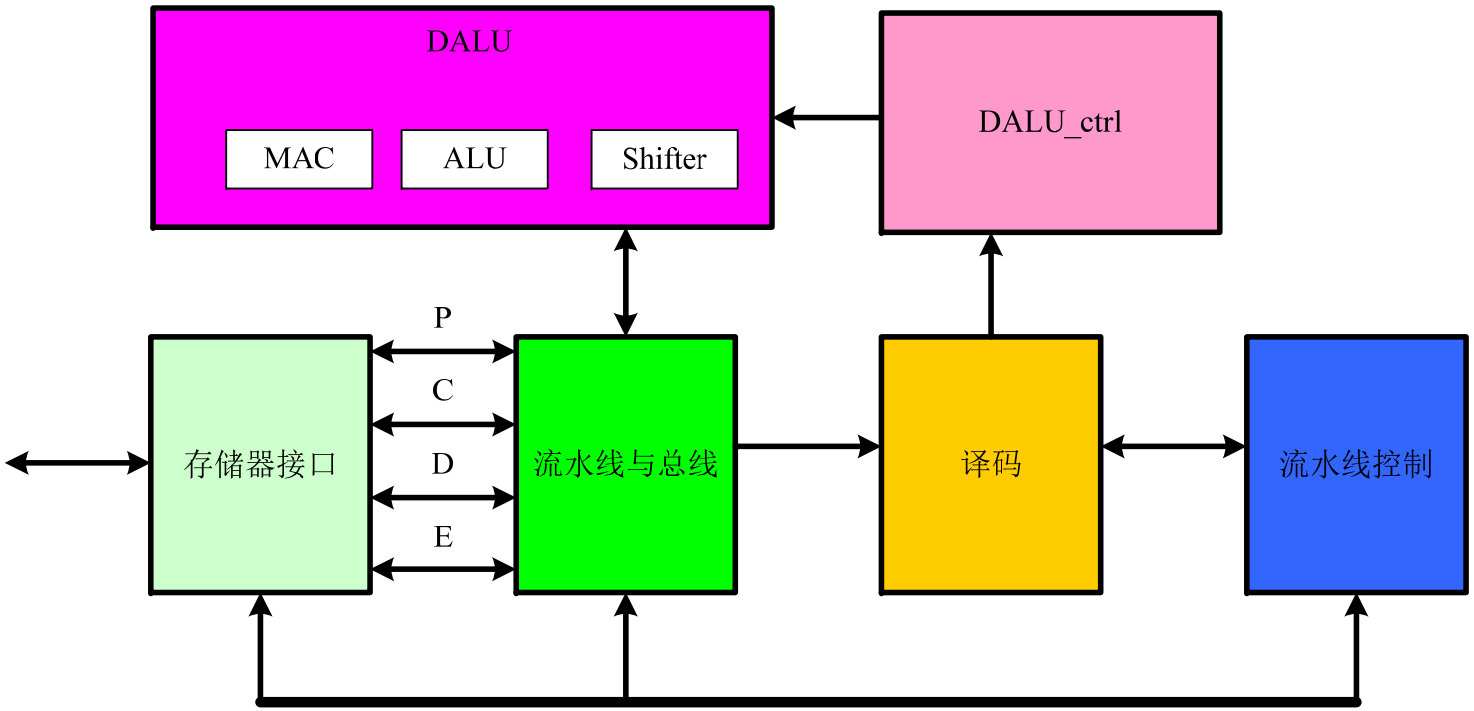
```
module pipeline2(clk,d,q);
input clk,d;
output q;

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

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

Pipeline1 VS pipeline2 VS pipeline3

代码准则

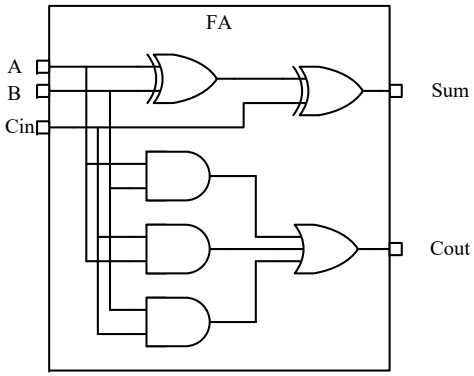
- 不要在一个always块中同时使用阻塞和非阻塞。
- 建模时序逻辑时使用非阻塞赋值。
- 使用always块描述组合逻辑，使用阻塞赋值。
- 不能在不同的always块中多次赋值一个信号。

结构化设计

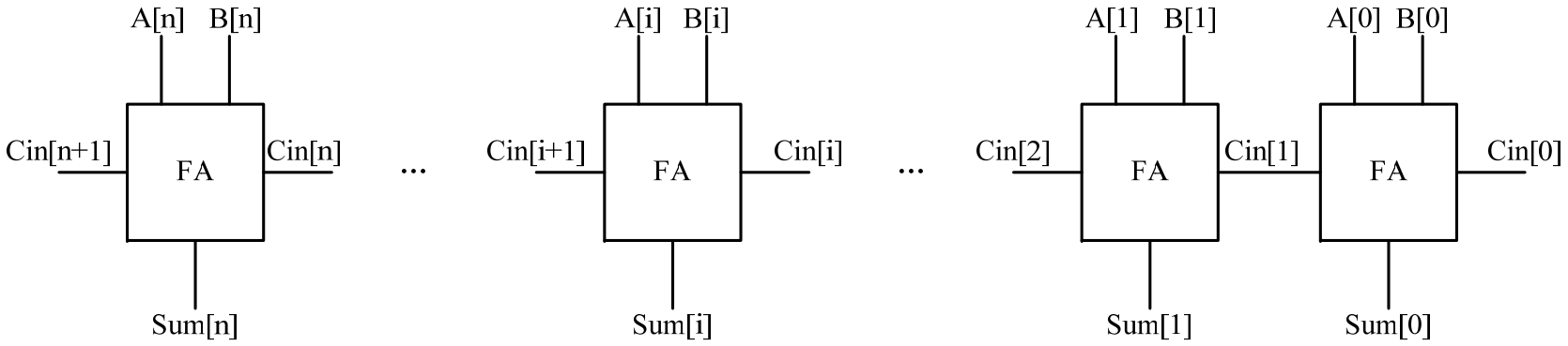


结构化设计举例

- 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

内部端口与模块端口

```
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

内部端口，指定类型，宽度等

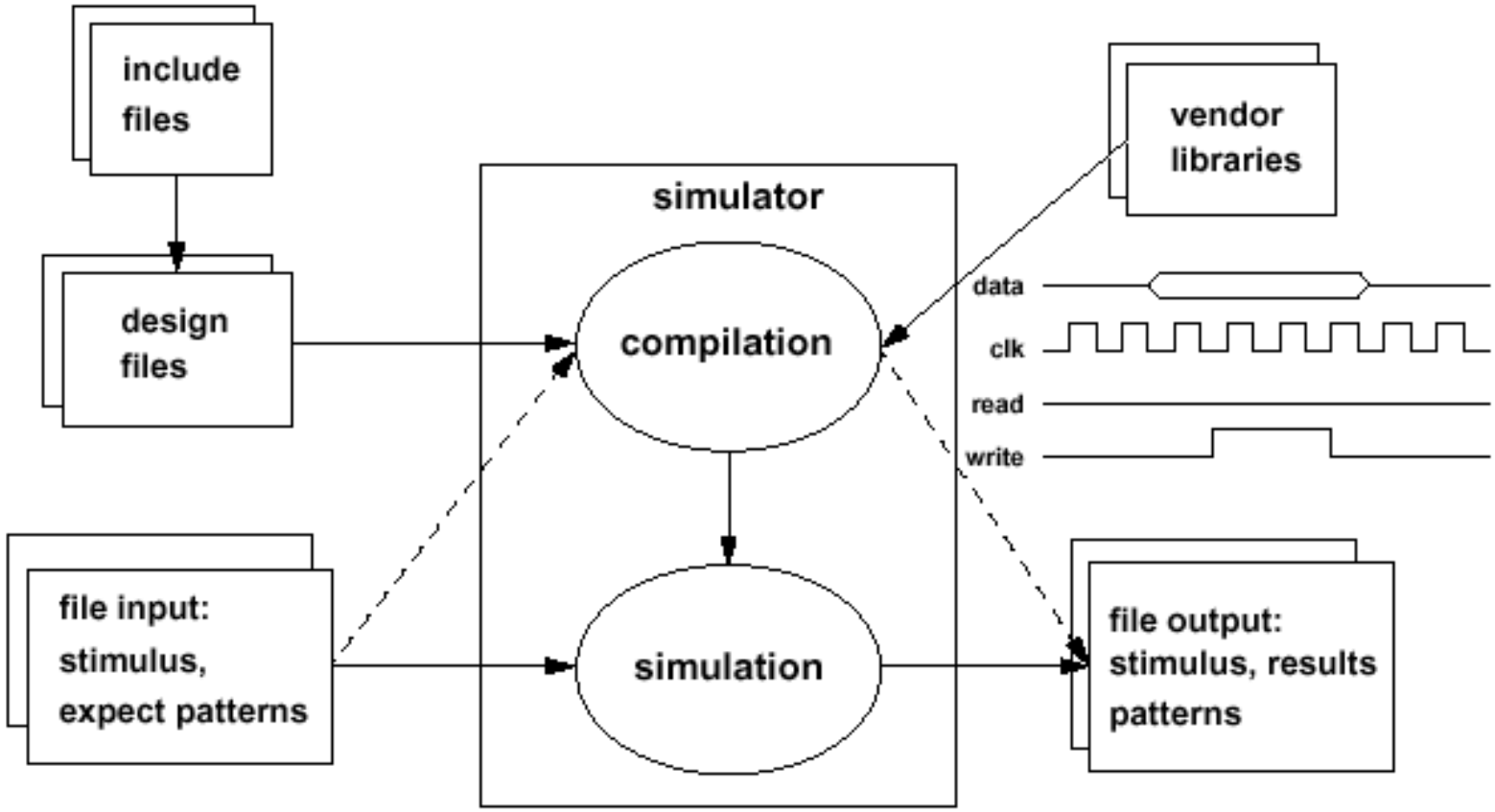
结构化设计举例

```
`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

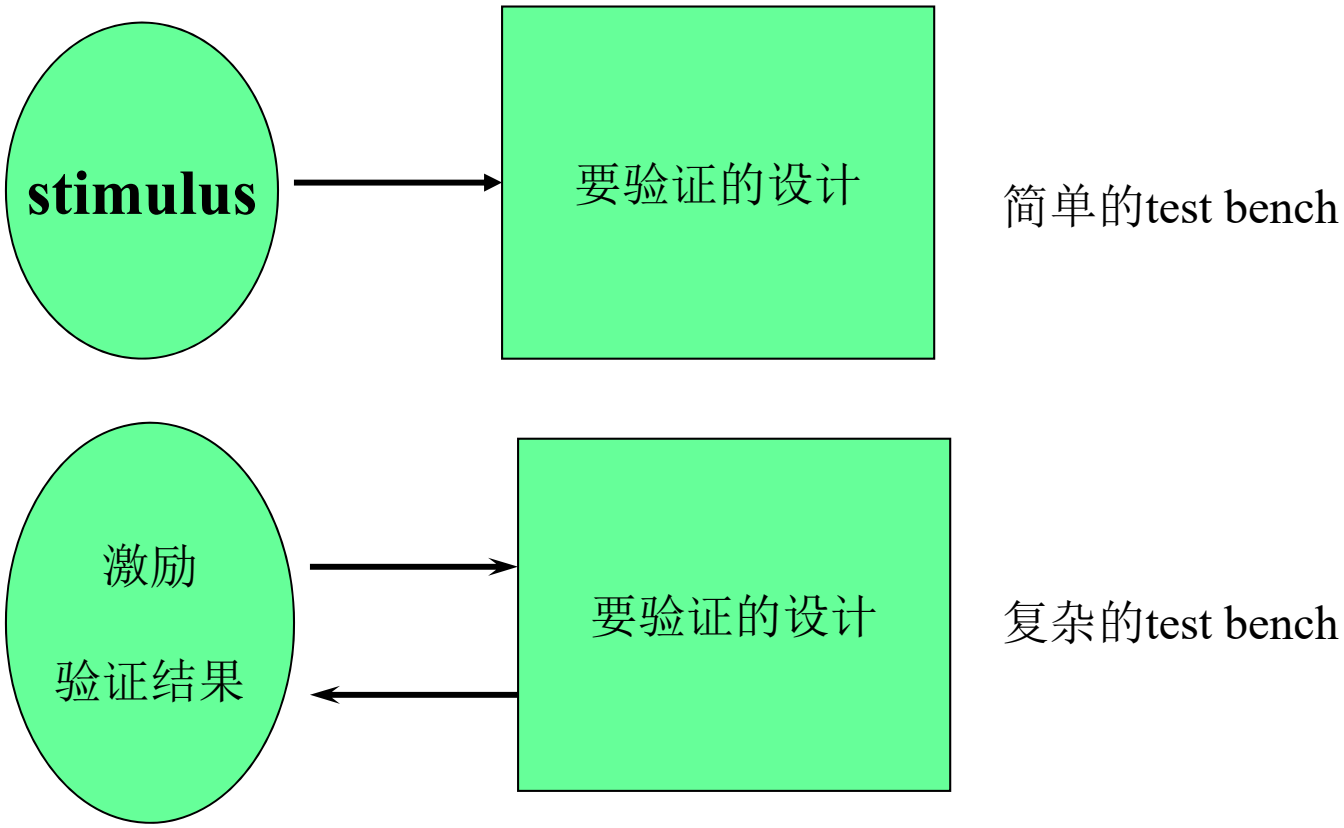
- 尽量使用名字关联
- 注意端口位宽的一致性
- 检查端口连接规则
- 不要出现没有连接的端口
- 一般不需要显式声明模块端口

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。

Testbench组织



- 简单的test bench向要验证的设计提供向量，人工验证输出。
- 复杂的test bench是自检测的，其结果自动验证。

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 fork                                //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
join
endmodule
```

编译指导

- 编译指导以（`）开头，指导仿真编译器进行一些特定的操作，编译指导被设置后一直有效，直到被替换或者撤销。

`celldefine
`default_nettype
`define
`else
`endcelldefine
`endif
`ifdef
`include
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef

`timescale 说明时间单位及精度
格式：

`timescale <time_unit> /<time_precision>

如：**`timescale 1 ns / 100 ps**

time_unit: 时间单位

time_precision: 时间精度

`timescale必须在模块之前出现

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：

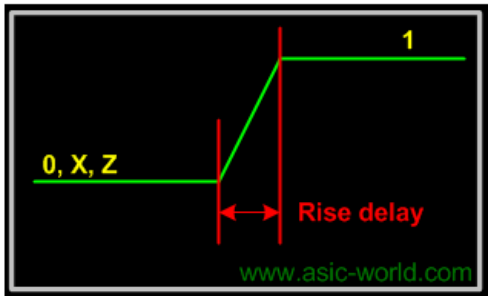
.内部信号（外部信号）

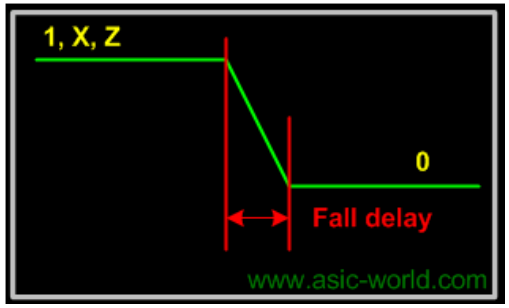
施加激励

- 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

时延

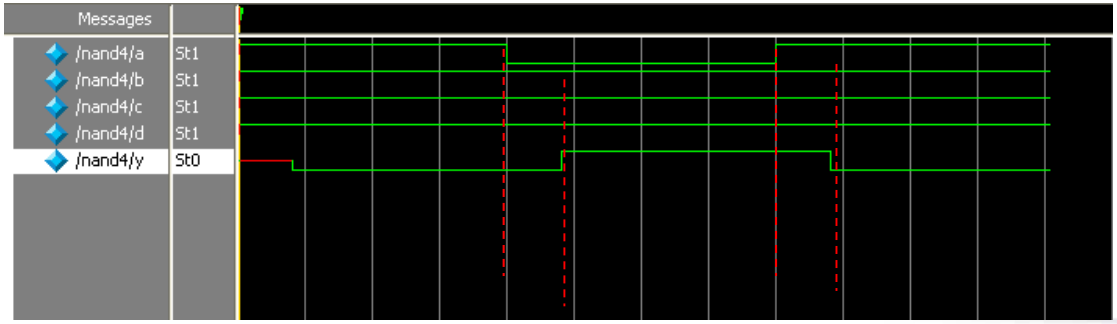
- 延时的来源





- 考虑时延的数据流模型。

```
`timescale 1ns/10ps
module nand4(a,b,c,d,y);
input a,b,c,d;
output y;
wire a,b,c,d;
assign #2  y=!(a&b&c&d);
endmodule
```



fork…join

fork...join块在测试文件中很常用。他们的并行特性使用户可以说明绝对时间，并且可以并行的执行复杂的过程结构，如循环或任务

```
module inline_ tb;
  reg [7: 0] data_ bus;
  // instance of DUT
  initial    fork
          data_bus = 8'b00;
    #10 data_bus = 8'h45;
    #20 repeat (10) #10 data_bus = data_bus + 1;
    #25 repeat (5) #20 data_bus = data_bus << 1;
    #140 data_bus = 8'h0f;
  join
endmodule
```

Time	data_ bus
0	8'b0000_0000
10	8'b0100_0101
30	8'b0100_0110
40	8'b0100_0111
45	8'b1000_1110
50	8'b1000_1111
60	8'b1001_0000
65	8'b0010_0000
70	8'b0010_0001
80	8'b0010_0010
85	8'b0100_0100
90	8'b0100_0101
100	8'b0100_0110
105	8'b1000_1100
110	8'b1000_1101
120	8'b1000_1110
125	8'b0001_1100
140	8'b0000_1111

上面的两个**repeat**循环从不同时间开始，并行执行。象这样的特殊的激励集在单个的**begin…end**块中将很难实现。

线性激励

线性激励有以下特性：

- ✓ 只有变量的值改变时才列出
- ✓ 易于定义复杂的时序关系
- ✓ 对一个复杂的测试，测试基准(test bench)代码可能非常大

```
module inline_tb ();
    reg [7: 0] data_bus, addr;
    wire [7: 0] results;
    DUT u1 (results, data_ bus,
addr);
    initial
        fork
                data_bus = 8'h00;
                addr = 8'h3f;
          #10 data_ bus = 8'h45;
          #15 addr = 8'hf0;
          #40 data_ bus = 8'h0f;
          #60 $finish;
    join
endmodule
```

循环激励

从循环产生激励有以下特性：

✓在每一次循环
　修改同一组激励变量

✓时序关系规则

✓代码紧凑

```
module loop_tb;
    reg clk;
    reg [7:0] stimulus;
    wire [7:0] results;
    integer i;
    DUT u1 (results, stimulus);
    always begin
    #5 clk = 1;  #5  clk = 0;
    end
    initial begin
        for (i = 0; i < 256; i = i + 1)
       @( negedge clk)  stimulus = i;
       #20 $finish;
   end
endmodule
```

存储器激励

从存储器产生激励有以下特性：

- ✓ 在每次反复中，修改同一组激励变量
- ✓ 激励向量可以直接从文件中读取

```
module array_ tb;
   reg [7: 0] data_ bus, stim_ array[ 0: 15]; //
数组
   integer i;
   DUT u1 (results, stimulus);
   initial begin
      // 从数组读入数据
      #20 stimulus = stim_array[0];
      #30 stimulus = stim_array[15]; // 线激励
      #20 stimulus = stim_array[1];
      for (i = 14; i > 1; i = i - 1) // 循环
         #50 stimulus = stim_array[i] ;
      #30 $finish;
   end
endmodule
```

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

显示信号值 — $display

- $display输出参数列表中信号的当前值到标准输出设备，并且带有行结束符。

语法：$display (format_specifiers，argument_ list)

```
$display (“%b,%d”,signal1,signal2);
$display (“%h,%o”, signal3,signal4);
```

格式符

%h	%o	%d	%b	%c	%s	%v	%m	%t
hex	octal	decimal	binary	ASCII	string	strength	module	time

转义符

\t	\n	\\	\"	\< 1-3 digit octal number>	%0d
tab	换行	反斜杠	双引号	上述的ASCII表示	无前导0的十进制数

监视信号任务—$monitor

- $monitor持续监视参数列表中的变量，$display任务在返回值之后就结束。
- 在一个时间片中，参数表中任何信号发生变化，$monitor将显示参数表的信号值。
- 后面的$monitor将覆盖前面的$monitor。
- 可以用系统任务$monitoron和$monitoroff控制持续监视。
- $monitor支持多种基数。与$display任务一样缺省为十进制。

$monitor举例

```
module monitor;
  reg a,b,c;
  initial
  begin
     #0    a=1;b=0;c=1;
     #10   a=0;b=0;c=1;
           a=1;b=0;c=0;
           a=1;b=1;c=1;
     #10   a=1;b=0;c=1;
     #10   a=1;b=0;c=1;
     #10   a=1;b=1;c=1;
  end
initial begin
$monitor ("%t,%b,%b,%b",$time,a,b,c);
end
endmodule
```

下面是模块仿真的输出：

输出：

0 1 0 1

10 1 1 1

20 1 0 1

40 1 1 1

$display举例

```
module display ();
  reg a,b,c;
  initial
  begin
     #0  a=1;b=0;c=1;
     $display ("%t,%b,%b,%b",$time,a,b,c);
     #10 a=0;b=0;c=1;
        a=1;b=0;c=0;
        a=1;b=1;c=1;
     $display ("%t,%b,%b,%b",$time,a,b,c);
     #10 a=1;b=0;c=1;
     #10 a=1;b=0;c=1;
     #10 a=1;b=1;c=1;
     $display ("%t,%b,%b,%b",$time,a,b,c);
  end

endmodule
```

下面是模块仿真的输出：

输出：

0 1 0 1

10 1 1 1

40 1 1 1

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 fork
      begin   a_input=0;b_input=0;cin=0; end
#10 begin   a_input=0;b_input=0;cin=1; end
#15 begin   a_input=1;b_input=1;cin=0; end
join
initial begin
$monitor("%b,%b,%b,%b,%b",a_input,b_input,cin,sum_out,cout);
#30 $finish;
end
endmodule
```

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)

仿真器工作模式

- 仿真器分两步实现仿真功能。
- 第一步为编译阶段，生成一个宿主机可执行的仿真程序。
- 第二个阶段为执行阶段，运行可执行仿真程序即可产生波形文件或者输出信息。

Verilog的执行顺序

- 由于verilog是一种并行语言，因此需要确定哪一些执行顺序是确定的，哪些执行顺序是不确定的。

- 确定的调度顺序
  - 在begin...end定义的范围内，只要这个begin...end块被执行，必定以顺序的方式执行

- 不确定的调度顺序
  - 多个事件有效时，它们的执行顺序是不定的。
  - Verilog语言是并行语言，允许多个同时发生的动作，但执行这些操作时要求进行序列化，因为计算机与其程序可能是不并行的。
  - 虽然每个仿真器对于相同的电路和输入、输出给出相同的结果，但不同的仿真器版本或者不同的仿真器之间可能是不同。

example

```
module test();
initial
fork begin
$display("ev state is observed at block1");
end
 begin
$display("ev state is observed at block2");
end
begin
$display("ev state is observed at block3");
end
begin
$display("ev state is observed at block4");
end
join
endmodule
```

```
VSIM 14> run
# ev state is observed at block1
# ev state is observed at block2
# ev state is observed at block3
# ev state is observed at block4
```

Thank you!