A detailed close-up of a green printed circuit board (PCB) populated with various electronic components. Three miniature construction worker figurines are positioned on a large, square, purple component. The worker on the left wears a blue jumpsuit and a yellow hard hat, with one arm raised. The worker in the center wears a light blue shirt, brown trousers, and red suspenders, holding a long, thin wooden pole vertically. The worker on the right wears a white jumpsuit and a red hard hat, with both arms raised in a celebratory gesture. The background is a blurred view of the rest of the PCB, showing numerous integrated circuits, resistors, and other electronic parts.

数字逻辑和处理器基础实验 Verilog语言进阶、仿真和静态时 序分析

数字逻辑与处理器基础实验

Verilog硬件描述语言进阶内容



Verilog可综合和不可综合

- Verilog是硬件描述语言，进行逻辑综合是其中重要但不唯一的一个功能
- 另一部分重要的功能是进行仿真验证和时序分析等，这些部分不必实现成电路，可以使用不可综合的语法



常量

其他类型的常量（不可综合的）

整形常量中的X

实数：1.34，1.3e2（130）

字符串：“FourValue”

参数型常量（可综合的）：*parameter, localparam*

```
parameter time_delay=5,time_count=10;
```

```
localparam PERIOD_VALUE=20;
```



常量 参数型常量: *parameter*, *localparam*

```
parameter time_delay=5;
```

```
localparam PERIOD_VALUE=20;
```

◆ **parameter**

定义标志符，可在外部改变

◆ **localparam**

定义本地标志符，不可在外部改变

```
module shift_registers_0 (clk, clken, SI, SO);  
    parameter WIDTH = 32;  
    input clk, clken, SI;  
    output SO;  
  
    reg [WIDTH-1:0] shreg;  
  
    always @(posedge clk)  
        begin  
            if (clken)  
                shreg = {shreg[WIDTH-2:0], SI};  
        end  
  
    assign SO = shreg[WIDTH-1];  
  
endmodule
```



参数定义

不要求掌握

```
`timescale 1ns/1ns
```

```
module demo_para(...);  
...  
parameter p1=8,p2=9,p3=10;  
...  
endmodule
```

定义参数

```
`timescale 1ns/1ns
```

```
module demo_call(...);  
...  
demo_para #(4,5,6) g1(...);  
...  
endmodule
```

嵌入隐式调用

```
demo_para #(.p1(4),.p2(5),.p3(6)) g1(...);
```

嵌入显式调用

```
demo_para g1(...);  
defparam g1.p1=4,g1.p2=5,g1.p3=6;
```

显式调用



Reg型变量

- Reg型变量需要被明确地赋值，在设计中变量必须放在过程块语句中（如initial或always），**通过过程赋值语句赋值**，而且在过程块内被赋值的每一个变量必须定义成该类型
- reg: 常用寄存器型变量
- integer: 32位带符号整数型变量
- real: 64位带符号实数型变量（不可综合）
- time: 无符号时间变量（不可综合）

```
reg a,b; //定义了两个一位宽reg变量a,b;  
reg [7:0] data; //定义8位宽reg向量  
reg a=0; //初始化操作
```

Reg型变量不一定对应寄存器，仅代表可以在过程语句中被赋值的变量



延时表示（不可综合）

过程赋值语句的延时表示

外部模式: `#delay a=b;`

内部模式: `a= #delay b;`

经过由delay所确定的时间单位的延时后，过程赋值语句右端的表达式才被求值并被赋给左端的寄存器变量

右端表达式的值经由delay所确定的延时后，其值被赋给左端的寄存器变量

一般使用外部模式建模

`temp_a=b; #delay a=temp_a;`

两种延时模式的区别在于右端表达式求值的时间段的区别，外部模式中，求值是在延时时间到达后进行，而内部模式则是先求值，再等待延时到达后赋值



基本门延时和连线延时

- 门级延时表示从基本门的输入端发生变化到输出端发生变化的门传输延时
- assign语句延时表达了赋值表达式右端的变化反映到左端连线上的信号发生变化的延时时间
- 连线延时直接体现了信号在连线上的传输延时
- 缺省延时为0，延时一般表示方法 $\#(d1, d2, d3)$ ，其中d1表示转移到1状态的延时，即上升沿延时；d2表示转移到0状态的延时，即下降沿延时；d3表示转移到z状态的延时，也称关断延时。d1、d2、d3地单位是由`timescale语句确定的



基本门延时和连线延时

- 如果延时项中只有一个值，此值表示所有状态转移的延时；如果给出两个延时值，则表示d1和d2，d3为d1和d2中的最小值；如果给出三个延时值，则表示d1、d2和d3。如果未指定延时值，那么默认延时为零
- 一种更复杂的延时表示是对d1，d2，d3的三种转移延时都定义了最小值、典型值和最大值（：分隔），共九个延时值。模拟器通过选择不同情况的延时值进行仿真，表示最好情况、典型情况和最差情况

不要求掌握



基本门延时和连线延时

```
not #5 (ndata, data);
```

```
nand #(3,5) d(wa,data,clock),nd(wb,ndata,clock);
```

```
nand #(12,15) qQ(q,nq,wa), nqQ(nq,q,wb);
```

```
bufif1 #(3,7,13) qdrive(qout,q,enable), nqdrive(nqout,nq,enable);
```

```
assign #(7,8,12) a=(ctrl==1)?b&c:z;
```

```
assign #(1:2:3,4:5:6) b=d^f;
```

不要求掌握



过程语句

always语句

```
always @(sensitive expr)
begin
    //过程赋值语句、if语句、case语句、循环语句、
    // task、function
end
```

当敏感表达式的值改变时，
就会执行一遍块内语句

initial语句

```
initial
begin
    clause 1;
    clause 2;
    ...
end
```

在仿真时刻‘0’启动，
只执行一次，主要用于
编写测试向量和对某些
变量赋初值



过程语句

always语句和initial语句的区别

◆ initial过程语句不带触发条件，因而必定从模拟0时刻开始执行它后面地城语句，沿时间轴只执行一次。而always过程语句通常带有敏感条件，只有条件被激活才执行，如果执行条件缺省，则认为始终满足

◆ initial语句最经常应用于测试模块中对激励向量的描述，而在对硬件功能模块的行为描述中，仅在必要时给寄存器变量赋以初值。这是一条主要面向仿真的过程语句，通常不为综合器所接受。例外：在利用Vivado进行FPGA设计时，由于FPGA的特性，可以赋初值。

◆ always语句一般可用于测试模块中的时钟描述，以及硬件功能模块的功能描述

```
initial begin
    clk1=0;
    clk2=0;
end
```

```
initial begin
    clk1=0;
    forever
        #5 clk1=~clk1;
end
```

```
always
    #50 clk2=~clk2;
```



块语句

块语句是由块标识符begin-end或fork-join界定的一组行为描述语句。块内的语句按照既定次序顺序或并行执行

```
begin
    #10 b=a;
    #10 c=a;
end
```

begin-end包含的块内语句按照串行方式顺序执行，块中每条语句给出的延时都是相对于前一条语句执行结束的相对时间

```
fork
    #10 b=a;
    #10 c=a;
join
```

fork-join包含的块内语句按照并行方式执行，他们同时开始执行，与排列顺序无关，块中每条语句给出的延时都是相对于并行块开始执行的时间而言的；并行块结束的时间是所有语句中最后执行完的语句的结束时间



循环语句

用于仿真
在本课程中不建议用于综合

```
count=0;  
for(i=0;i<var_size;i=i+1)  
    if (var[i])  
        count=count+1;
```

for循环语句

```
initial  
begin  
    clock=0;  
    forever  
        #50 clock=~clock;  
    end
```

forever循环语句,永远循环

```
parameter size=8;  
input [7:0] var;  
integer count;  
.....  
always @(var)  
begin  
    count=0;  
    repeat(size)  
    begin  
        count=count+var[i];  
    end  
end
```

repeat语句: 先计算出循环次数
计算表达式的值, 并将他作为过程重复执行的次数的基值保存起来; 然后每执行一次循环体则减1, 直至为0退出循环

```
parameter size=8;  
input [7:0] var;  
integer count;  
integer i;  
.....  
always @(var)  
begin  
    i=0;  
    count=0;  
    while(i<size)  
    begin  
        count=count+var[i];  
        i=i+1;  
    end  
end
```

while语句: 先判断循环执行条件表达式是否为真, 只要是真, 则执行循环体, 否则退出循环



系统函数/任务和编译向导

- 为了方便设计者对仿真过程的控制与对仿真结果的分析，Verilog HDL提供了一系列的系统功能调用，统称为系统函数/任务。在程序中，系统函数/任务以符号\$表明
- 编译向导是一类特殊的Verilog HDL语句，他的目的是对Verilog HDL的编译过程实现控制。在程序中，编译向导以符号`标识



系统函数/任务

系统任务Display & Write

```
module demo_display;  
reg [16:1] var;  
wire temp_wire;  
  
initial  
begin  
    var='h8a';  
    $display("line1:var=%d(D)=%h(H)=%o(O)=%b(B)",var,var,var,var);  
    $display("line2:var=%b(B)=%0b(B)",var,var);  
    $display("line3:var=",var);  
end  
endmodule
```

%h, %d, %o, %b: 以16进制、十进制、8进制、二进制格式输出
%c: 以ASCII字符形式输出
%s: 以字符串方式输出
%t: 输出模拟系统所使用的时间单位
%e: 将实型量以指数方式输出...
,: 打印一个空格

\$display("格式控制字符串", 输出变量名表项);
\$write("格式控制字符串", 输出变量名表项);

Line1: var=138(D)=008a(H)=000212(O)=0000000010001010(B)

Line2: var=0000000010001010(B)=10001010(B)

Line3: var=138

区别: display输出结束后能自动换行, 而write不会自动换行



系统函数/任务

系统任务monitor

```
module demo_monitor;  
reg var;  
parameter delay=10;
```

```
initial  
begin
```

```
    $display("time          value");  
    $monitor($time,,, "var=%b", var);  
    #delay var=1;  
    #delay var=0; $monitoroff;  
    #delay var=1; $monitoron;  
    #delay var=0;  
    #delay var=1;  
    #1000 $finish;
```

```
end  
endmodule
```

`$monitor("格式控制字符串",输出变量名表项);`

`$monitoron;`
`$monitoroff;`

display与monitor的区别

`display`每调用一次，则执行一次；而
`monitor`一旦被调用，则相当于启动了一个后台进程，将随时对输出变量名表项列出的各项进行监测，一旦发现其中任何一个在某一时刻有变化，则输出一次变量结果；而`monitoron`与`monitoroff`则是对后台进程的开关管理



系统函数/任务

- 系统函数\$time与\$realttime

- 调用形式: \$time, \$realttime, 分别返回一个64位整数或实型数, 表示被调用的时刻的时间值

- 系统任务\$finish和\$stop

- \$finish的作用为中止仿真器的运行, 结束仿真过程
- \$stop的作用为暂停仿真器的运行, 此时设计人员可以输入相应的命令, 对模拟过程进行交互控制, 然后接着运行

- 系统函数\$random

- \$random用于产生一个随机数, 其调用方式为\$random %b, 他将产生一个从(-b+1)到(b-1)之间的随机数

```
a=$random % 70;           //-69~69  
b={$random} % 70;         //0~69
```



编译向导

- ``define: `define A 100;`
 - 用于为一个复杂的名字和字符串定义一个简单的名字或标志符。在引用这个标志符时，注意在前面加上`符号
- ``timescale; `timescale 1ns/1ps`
 - 用于定义模块的时间单位与时间精度，调用方式`timescale <时间单位>/<时间精度>



编译向导

- 条件编译: ``ifdef`, ``else`, ``endif`

```
`ifdef 宏名  
    一组描述语句  
`else  
    另一组描述语句  
`endif
```

- ``resetall`: 取消前面所有编译向导引入的定义, 并恢复其原始的缺省态



常见问题

某些C语法在
verilog中没有对应

```
for(i=0;i<8;i++)  
begin  
...  
end
```

```
initial  
begin  
a=0,b=0,c=0;  
#10 a=1,b=1;  
#10 a=0,c=1;  
end
```

```
module mux_beh(out,a,b,sel);  
output out;  
input a,b,sel;
```

```
assign out=(sel==0)?a:b;  
endmodule
```

```
module test_for_mux;
```

```
mux_beh mux(out,a,b,s);
```

```
initial begin  
$monitor(...);
```

```
a=1;b=0;...  
#10 a=1;...  
#10 b=0;...
```

```
end  
endmodule
```

过程（initial、
always）中的被赋值
变量必须是寄存器型

数字逻辑与处理器基础实验

仿真

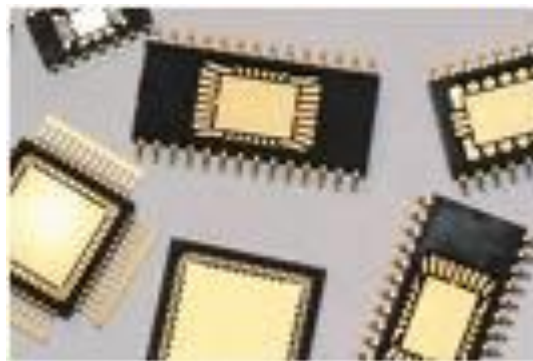
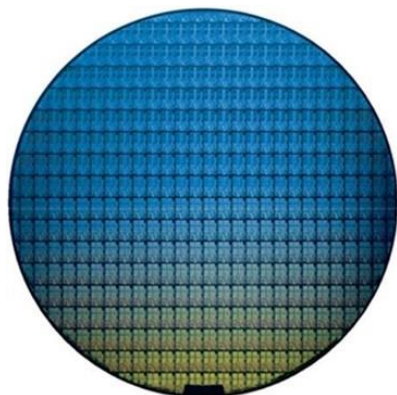
验证与测试(Verification & Test)

- 验证

- 检查设计实现是否能够满足设计需求，发现设计错误的过程，是设计过程的反过程，验证发现的问题可以通过重新设计进行更正，在很多情况下，**一个设计的70%以上的时间都被应用到验证上**

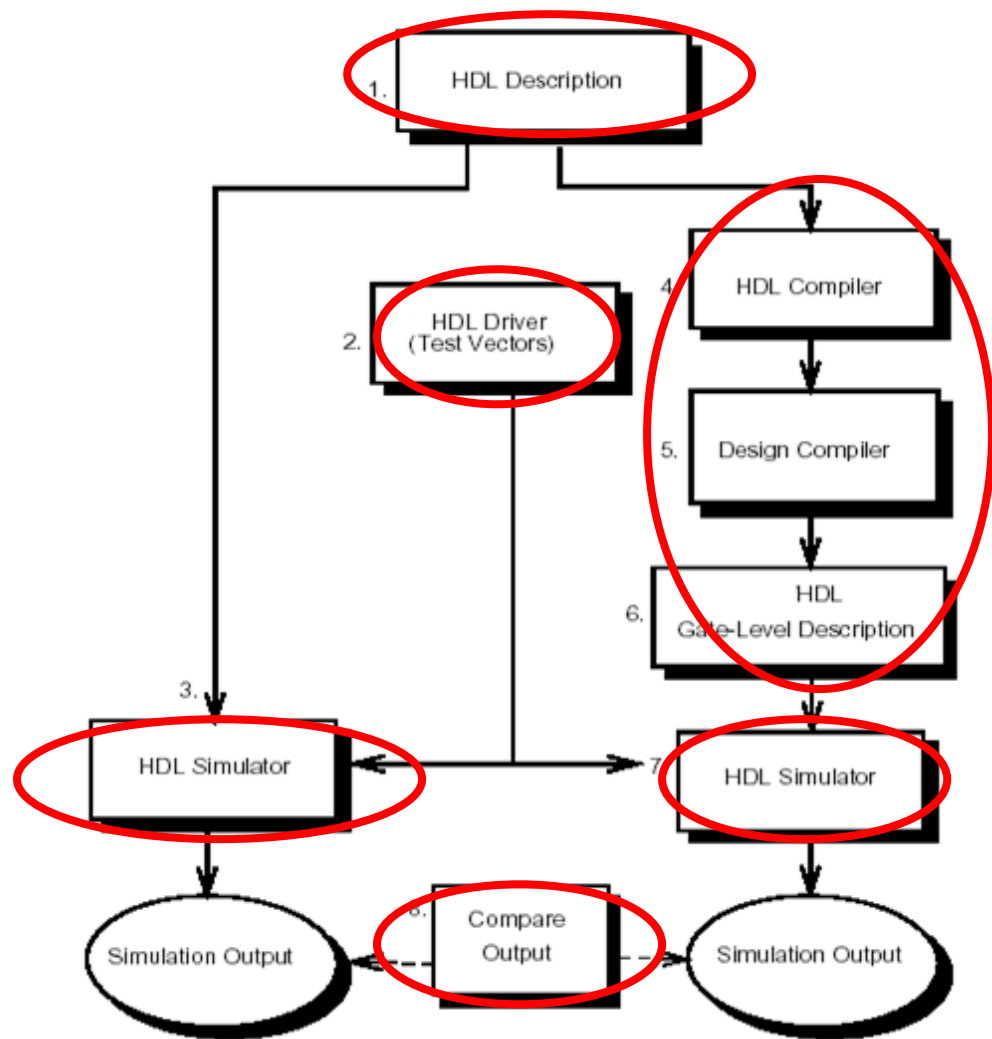
- 测试

- 检验芯片是否在生产和加工环节存在缺陷，这些缺陷往往是由于材料、工艺等因素造成的，不能依靠设计过程加以避免



芯片裸片的成本一般远小于封装成本，对芯片裸片进行测试可以降低芯片的制造成本

逻辑设计流程



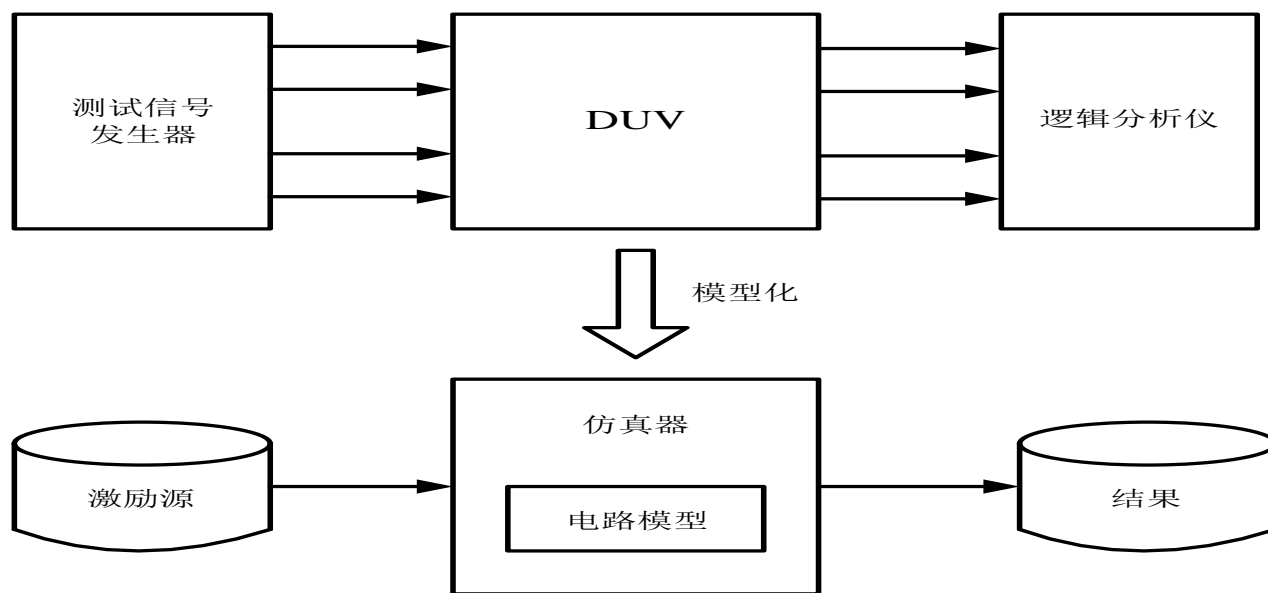


验证主要方法

- 分类
 - 功能验证：等价性验证、模型验证
 - 时序验证
- 主要方法
 - 仿真方法 (Simulation)
 - 形式化方法 (Formal method)
 - 静态时序分析方法 (Static Timing Analysis)

验证-仿真 (Simulation)

- 什么是仿真？
 - 仿真是最常见的一种验证方式
 - 基于等价性验证方法
 - 可以对设计进行功能验证和时序验证





常见仿真类型

- 行为与功能仿真

以行为算法和结构的混合描述为对象，一般采用VHDL或Verilog HDL语言描述，主要着眼于系统功能和内部运行过程、基本元素的操作和过程，各操作之间主要考虑数据传输、时序配合、操作流程和状态转换。高层次仿真的方法一般是对描述的解释执行，仿真时观察运行结果的数据及其时序配合关系、状态转换关系，来判断描述的正确性

- 门级仿真

逻辑仿真或者门级仿真通常是用来检查ASIC功能及时序性能。对于门级仿真器，每一个逻辑或者逻辑单元（比如NAND和NOR等）用黑盒子来模拟，一个逻辑门或逻辑单元的函数变量是输入信号。该函数也可以通过逻辑单元模拟其延迟。将所有延迟设定为单位值是等效的行为与功能仿真



测试平台设计要求

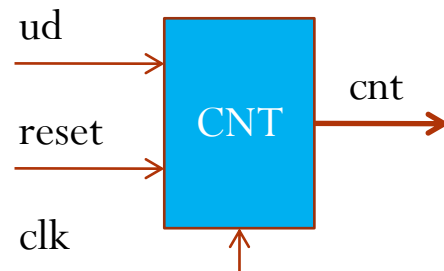
- 测试平台程序可以采用任何符合语法规则的语句
- 在系统级仿真时，主要验证系统的数学模型、算法结构的行为是否正确，应尽可能采用抽象程度较高的语句，以使程序简洁明了。此时，电路的各种延时都不予考虑，所有的语句及数据类型均可使用
- 在RTL级仿真时，主要验证被测模块是否符合逻辑工具的要求，能否生成相应的正确的门级电路，此时测试平台程序设计应注意：不能使用浮点数；必要时需要考虑门延时，但连线延时可以不考虑
- 门级仿真时，主要验证系统在不同延时条件下的功能和时序是否正确，测试平台程序应充分考虑到这一点



测试平台结构

```
`timescale 1ns/1ns
module cnt_dec(reset,clk,ud,cnt);
input reset,clk,ud;
output cnt;
reg [3:0] cnt;

always @(negedge reset or posedge clk) begin
    if(~reset) cnt <= 0;
    else begin
        if(ud) begin
            if(cnt==4'b1001) cnt <= 0;
            else cnt <= cnt + 1;
        end
        else begin
            if(cnt==0) cnt <= 4'b1001;
            else cnt <= cnt - 1;
        end
    end
end
endmodule
```



十进制计数器模式
ud:1,递增计数
ud:0,递减计数

递增计数

递减计数



测试平台结构

```
module cnt_dec_tb;  
  reg reset,clk,ud;  
  wire [3:0] cnt;  
  cnt_dec uut(.reset(reset),.clk(clk),.ud(ud),.cnt(cnt));
```

实例化待测模块

```
initial begin  
  reset <= 0;  
  clk <= 0;  
  ud <= 1;  
end
```

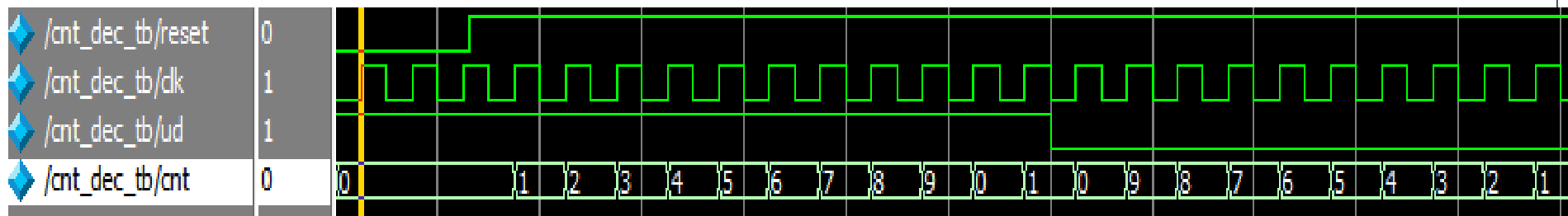
Initial初始化变量

```
initial fork  
  #130 reset <= 1;  
  forever #25 clk <= ~clk;  
  #700 ud <= 0;  
join  
endmodule
```

Fork...join并行块描述
多个激励信号



测试平台结构





```
`timescale 1ns/1ps
`define PERIOD 10

module adder_tb_simple();

    reg clk    ;
    reg rst_n;

    reg [3:0] data_in1;
    reg [3:0] data_in2;
    wire [3:0] data_out;

    // Generate the clock
    initial begin
        forever
            #(`PERIOD/2) clk = ~clk;
    end
    // Adder Instance
    adder i_adder (
        .clk      (clk      ),
        .rst_n     (rst_n    ),
        .data_in1  (data_in1),
        .data_in2  (data_in2),
        .data_out  (data_out)
    );
```

```
initial begin
    // Initialization
    rst_n = 1'b1;
    clk = 1'b0;
    data_in1 = 4'b0;
    data_in2 = 4'b0;

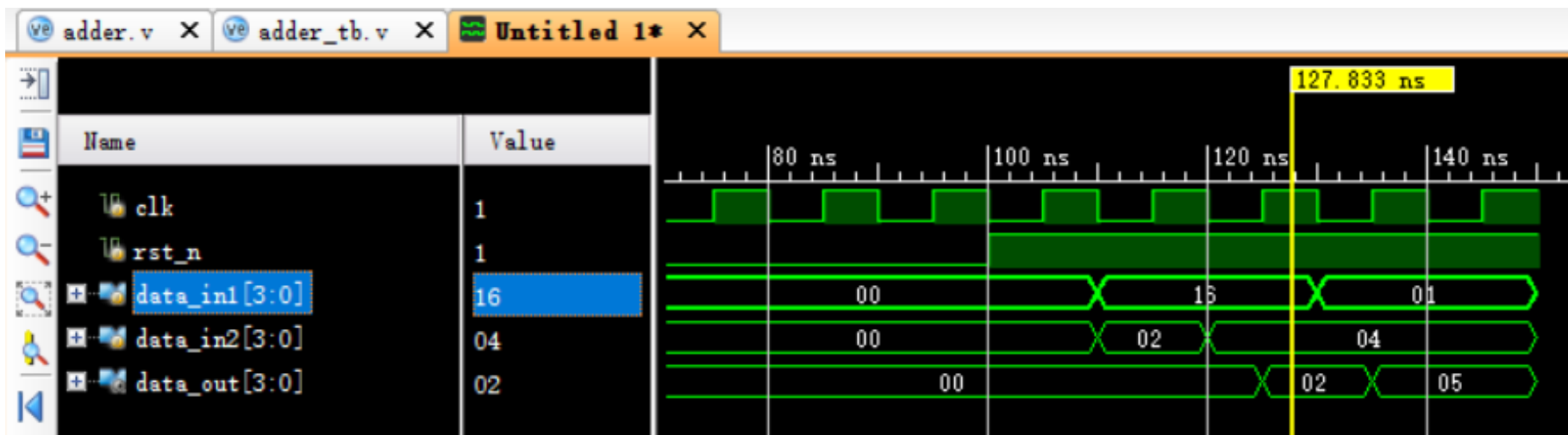
    // Reset the DUT
    #(`PERIOD*5) rst_n = 1'b0;
    #(`PERIOD*5) rst_n = 1'b1;

    #(`PERIOD)
    data_in1 = 4'b1110;
    data_in2 = 4'b0010;

    #(`PERIOD)
    data_in2 = 4'b0100;

    #(`PERIOD)
    data_in1 = 4'b0001;

    #(`PERIOD*2);
    $finish;
end
endmodule
```





测试平台基本原则

- 尽可能模拟实际情况

构建测试矢量应该尽可能模拟被测设计单元实际工作下的情况。

- 增加激励信号的随机性，比如可以随机调整数据包发生器发出数据包的长度。
- 在激励信号中人工插入错误（可以是随机错误，也可以是固定模式错误），检验被测设计能否识别错误并正确处理情况

- 灵活编写测试矢量

测试矢量的目的是用来测试验证被测设计的正确性，测试矢量部分本身不必考虑可综合和可实现性，因此编写测试矢量相对可以比较灵活。



仿真软件

- Vivado
- VCS
 - VCS是Verilog仿真器，适用于大型ASIC设计的仿真，稳定性好，仿真速度快，但需要较大容量内存
- Verilog-XL及NC
 - Verilog-XL是Cadence早期版本的verilog仿真器，仿真速度较慢；NC系列仿真器是新推出的vhdl、verilog仿真工具，仿真速度快，所需内存容量不是很大，可以支持vhdl和verilog的混合仿真，是一种较理想仿真工具
- Modelsim
 - 是一种支持VHDL及Verilog两种语言及混合语言的仿真工具，具有较好的稳定性和很好的操作界面，易于学习和掌握，是windows平台下较常用的仿真软件

Vivado中仿真波形的查看

- 可以在scopes中看到各模块中的各信号

Behavioral Simulation - Functional - sim_1 - signinput_tb

Scopes

Name	Design Unit	Block
signinput_tb	signinput_tb	Veri
signinput_uut	signinput	Veri
glbl	glbl	Veri

Objects

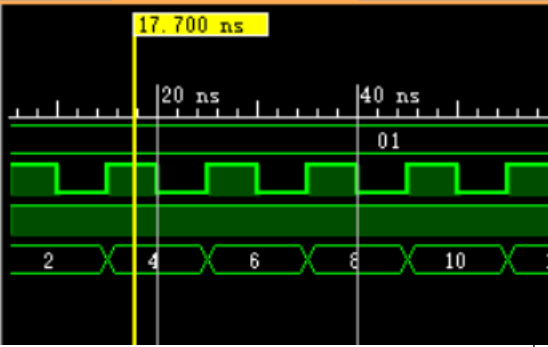
Name	Value
testmode[1:0]	00
sysclk	0
signinl	1
state[20:0]	16000
divide[20:0]	32000

signinput.v * x signinput_tb.v x Untitled 5* x

Name	Value
testmode[1:0]	01
sysclk	1
signinl	1
state[20:0]	4

这里展开各层级的模块

右键将需要观察的信号添加到波形窗口



- 不关心的信号，如sysclk，可以在波形图中删掉再仿真，提高一些仿真速度
- 可在Tcl console 窗口输入命令
restart, run 100ms 等

数字逻辑与处理器基础实验

静态时序分析



静态时序分析 (STA)

- 什么是静态时序分析？

- 静态时序分析是时序电路检查分析方法，可以分析由于电路时延而产生的时序裕度，从而判定逻辑电路是否能可靠工作

- 使用静态时序分析的必要性

- 随着IC设计的发展，芯片的规模、速度和加工工艺水平都飞速发展，这使得设计重用已经逐渐成为必然趋势
- 各种软、硬核的使用，规模达到数百万门的设计以及不断提高的时钟频率也在对电路的设计、验证方法提出更高的要求
- 传统的动态仿真方式存在很多问题：
 - 门级逻辑仿真对于验证电路时序的正确性在很大程度上依赖于验证向量的完备性，这一点也是最令设计者感到棘手的；
 - 为了得到较高的向量覆盖率，需要大量的验证向量；
 - 由于设计本身及验证向量的复杂性，使得仿真消耗大量的时间；
 - 随着深亚微米工艺的发展，通常需要在不同阶段进行多次仿真



静态时序分析 (STA)

- 静态时序分析的优缺点

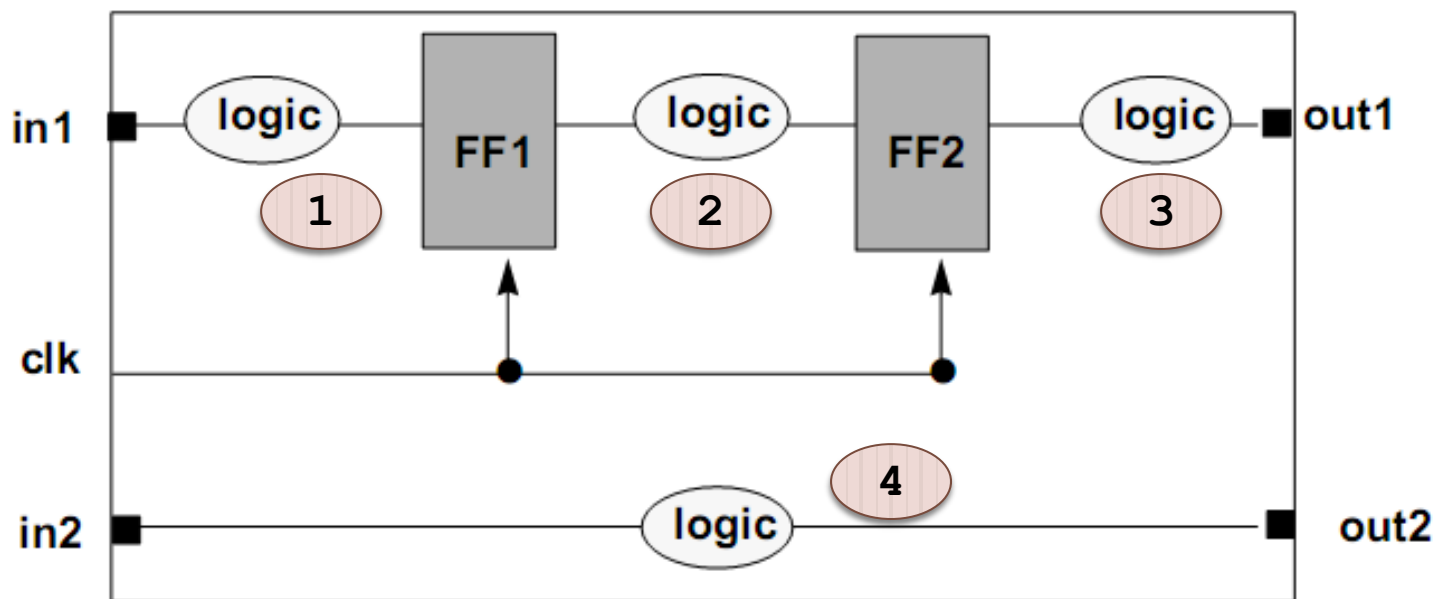
- 可以检查电路设计中所有路径的时序特性，验证路径的覆盖率可以达到100%，从而保证验证的完备性
- 不需要验证向量，所以STA验证时间远小于基于事件驱动的仿真
- 无法验证电路功能是否正确，对于大部分电路中所包含的异步时序逻辑的验证则需要使用动态仿真来完成

- 静态时序分析的步骤

- 将设计分解为不同路径的集合；
- 计算每条路径的延时信息；
- 检查所有路径的延时，分析时序约束是否满足

静态时序分析 (STA)

- 静态时序分析的路径



1 输入端口->寄存器

2 寄存器->寄存器

3 寄存器->输出端口

4 输入端口->输出端口



静态时序分析 (STA)

- 静态时序分析的时序检查

- 信号到达时间 (AT: arrival time)

表示计算所得信号到达逻辑电路中某一点的绝对时间，等于信号到达某条路径起点的时间加上信号在该条路径上的逻辑单元间传递延时的总和

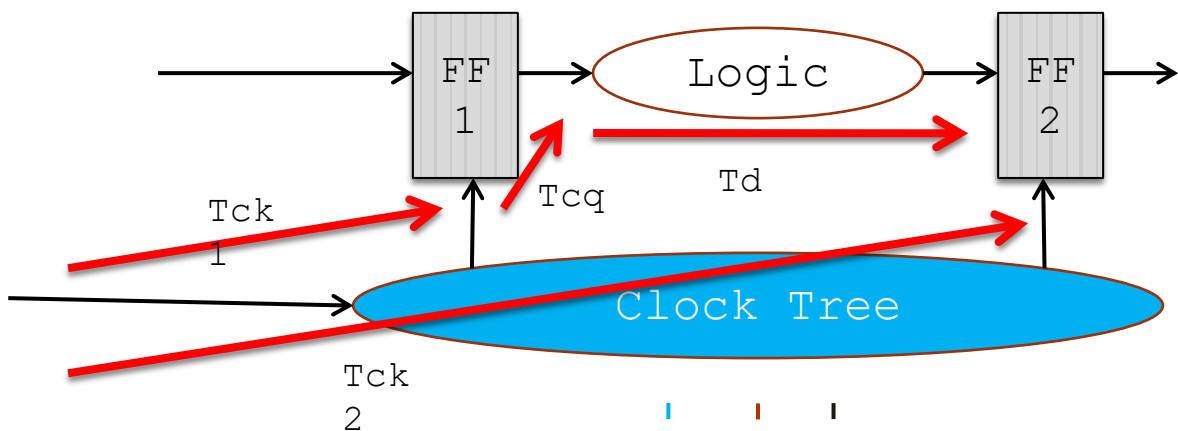
- 要求到达时间 (RAT: required arrival time)

表示要求信号在逻辑电路的某一特定点处的到达时间

- Slack

表示在逻辑电路的某个特定点处要求到达时间与实际到达时间之间的差，slack的值表示该信号到达是否太晚或者太早

静态时序分析 (STA)

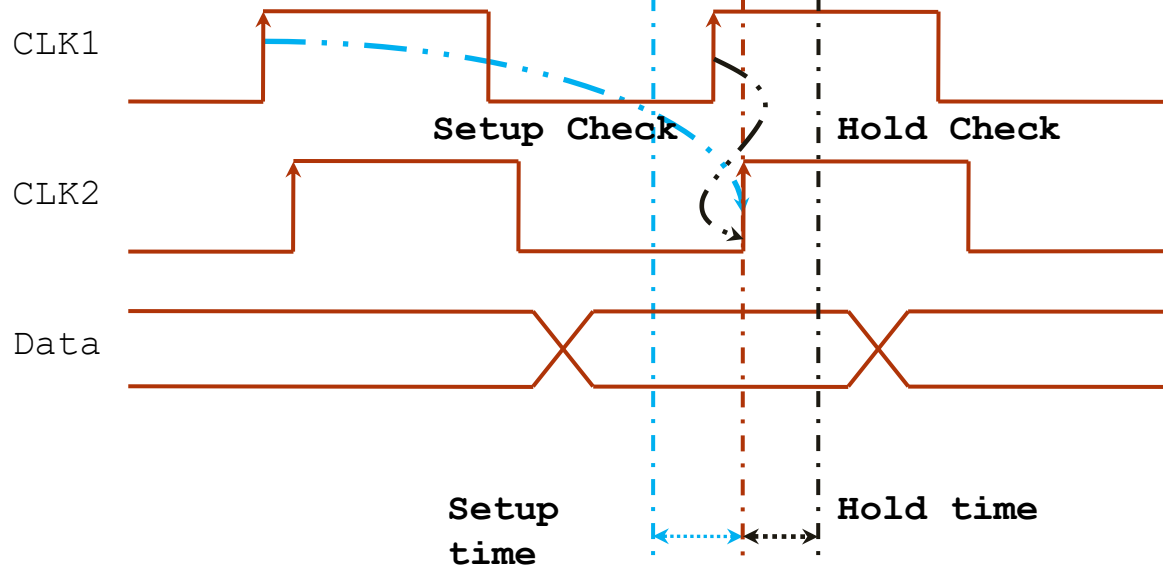


Late Data AT \leq
Early AT of clock
leading edge - Setup
time

Slack = RAT - AT =
(Early AT of clock
leading edge - Setup
time) - Late data AT

Early Data AT \geq
Late AT of clock
leading edge + Hold
time

Slack = AT - RAT =
Early data AT -
(Late AT of clock
leading edge + Hold
time)





时钟约束 (XDC文件相关)

- XDC约束文件

- XDC 是 Xilinx Design Constraints 的简写，其基础语法来源于业界统一的约束规范 SDC（最早由 Synopsys 公司提出，故名 Synopsys Design Constraints）
- 包括外部时钟约束、输入输出延时、管脚分配等。
- 在XDC文件中，每个完整的约束指令不应当跨行，必须在一行之内表达完毕。同样，在一行之内也只允许存在一个约束指令，不可以把多个约束放在同一行。
- XDC后面输入的约束在有冲突的情况下会覆盖之前输入的约束
- XDC文件的注释以“#”开头。唯一的限制是，在有效的XDC约束指令行末，不可以添加“#”开头的注释，否则Vivado会理解错误



时钟约束 (XDC文件相关)

- 实例

外部时钟输入约束

```
create_clock -period (clock period) -name  
(clock name) -waveform {(Traise), (Tfall)}  
[get_ports (clock port name)]
```

举例说明

```
create_clock -period 10.000 -name CLK -  
waveform {0.000 5.000} [get_ports sysclk]
```

input/output delay设置

```
set_input_delay -clock [get_clocks (clock name)]  
(delay time ns) [all inputs]
```

```
set_output_delay -clock [get_clocks (clock name)]  
(delay time ns) [all outputs]
```



时钟约束 (XDC文件相关)

管脚分配

```
set_property PACKAGE_PIN (pin location)  
[get_ports (port name)]
```

```
set_property IOSTANDARD (level: LVDS, LVCMOS18,  
LVCMOS33 etc.) [get_ports (port name)]
```

举例说明

```
set_property PACKAGE_PIN T18 [get_ports rst]  
set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

也可以将上面两行合并

```
set_property {PACKAGE_PIN T18 IOSTANDARD  
LVCMOS33} [get_ports rst]
```

将非时钟专用管脚用作时钟输入

```
set_property CLOCK_DEDICATED_ROUTE FALSE  
[get_nets (port_name)]
```