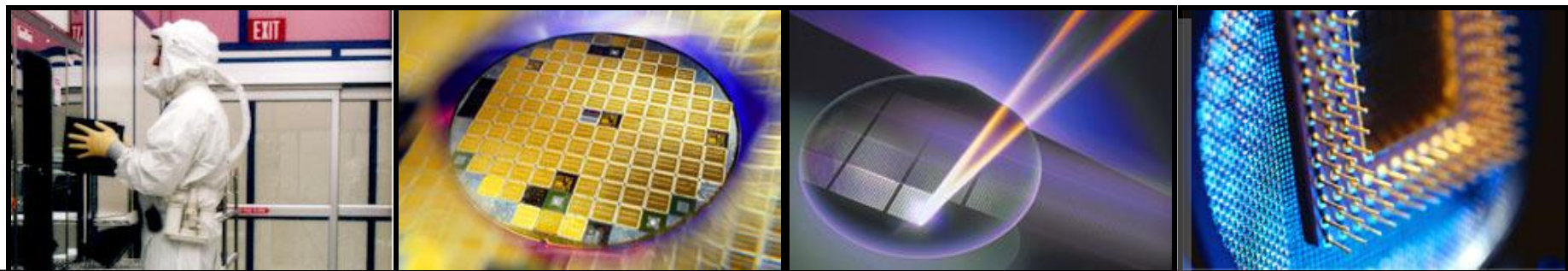




FPGA 系统设计和验证



课程安排

- FPGA设计基本流程
- FPGA设计规范
- FPGA的验证方法

课程安排

- FPGA设计基本流程
- FPGA设计规范
- FPGA的验证方法

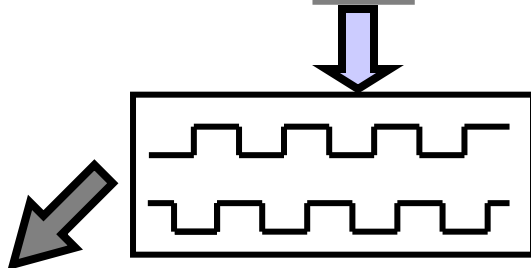
FPGA 设计流程

设计规范



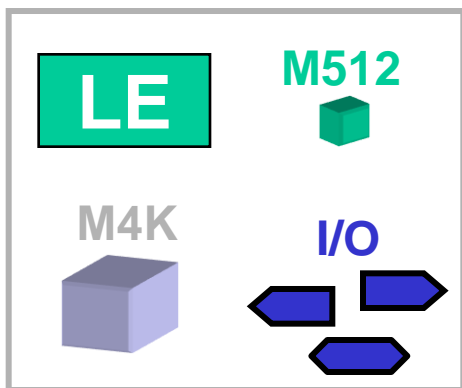
设计导入/RTL实现

- 面向结构或功能的描述



RTL 仿真

- 功能仿真
- 确认逻辑功能不考虑延时等因素



综合

- 将设计转化成和器件相关的描述
- 优化以达到面积和性能的要求

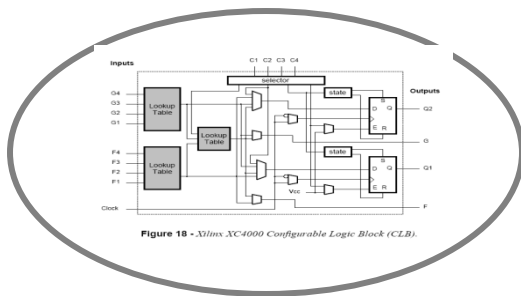
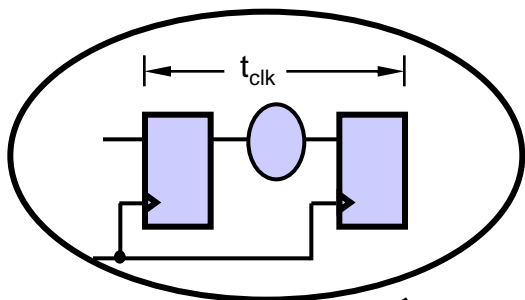


Figure 18 • Xilinx XC4000 Configurable Logic Block (CLB)

布局布线

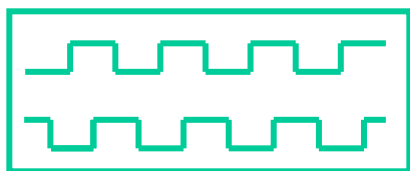
- 将逻辑实体映射到目标器件中去，并使其满足面积和性能的要求。
- 确定布线需要的资源

FPGA 设计流程



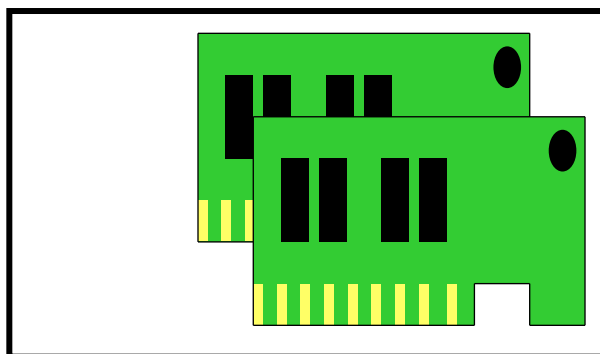
时序分析

- 确认性能满足要求
- 静态时序分析



门级模拟

- 时序仿真
- 确认设计在目标工艺中的正常工作



板级模拟&测试

- 仿真版设计
- 板上程序&测试

系统规划和预算

- 系统功能的总体规划：
 - 功能集的定义；
 - 端口的定义；
- 模块的基本划分和功能定义：
 - 每个模块应该完成的功能；
 - 模块之间的接口定义；
 - 模块间通讯的问题一定要考虑好，硬件通信的成本一般比较大。

设计的整体规划

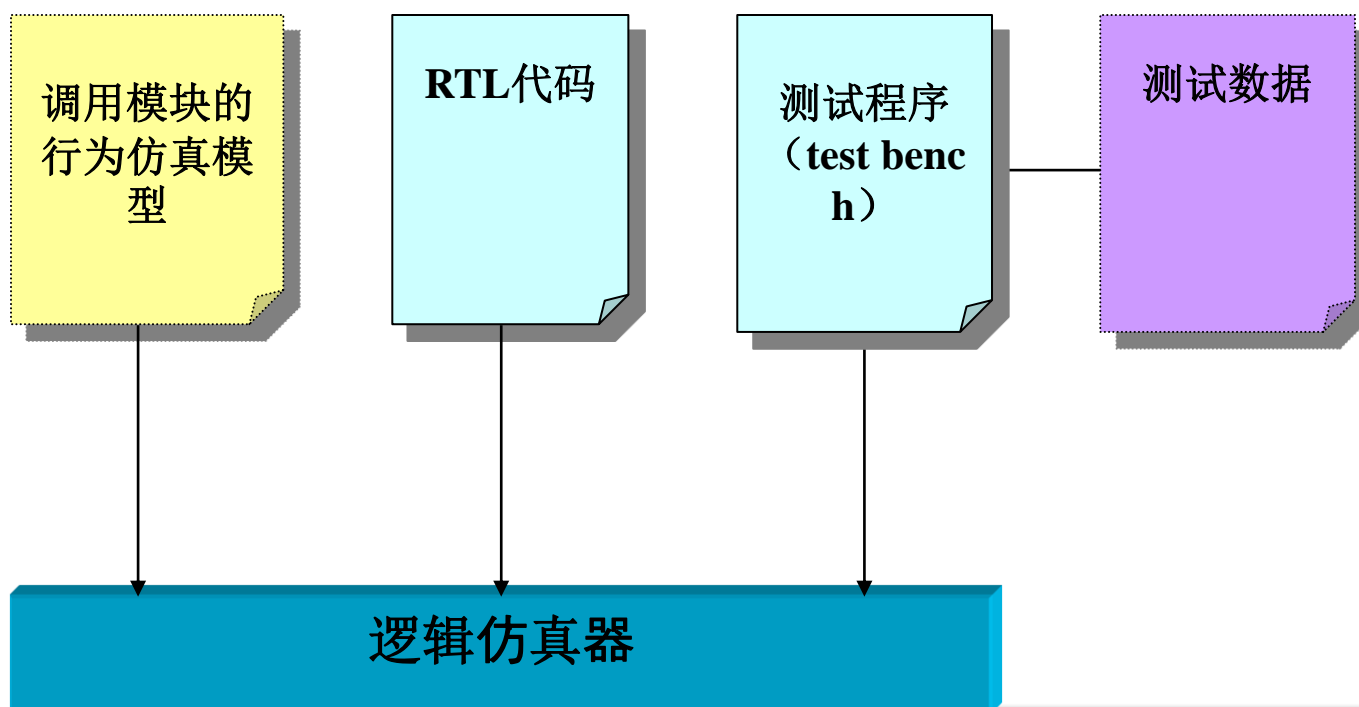
- 设计规模的初步估计，大致应该选择哪一层次的芯片；
- 设计时序的宏观规划：
 - 频率和时钟结构；
 - 可能的关键路径，着重优化；
- 模块的进一步细化，考虑可重用性等的规划：
 - 可以考虑基本单元，比如加法、乘法器和寄存器等。

设计实现

- 用电路框图或者HDL描述实现自己的设计：
 - 简单的设计可以用电路框图；
 - 大型复杂的一般倾向于用HDL描述；
 - HDL描述和计算机编程中的高级语言描述有很大不同，每一个描述都要考虑硬件的实现能力，是不是可以综合的等等，目前HDL语言标准中仍然有不能被综合的语法，这些要尤其注意。

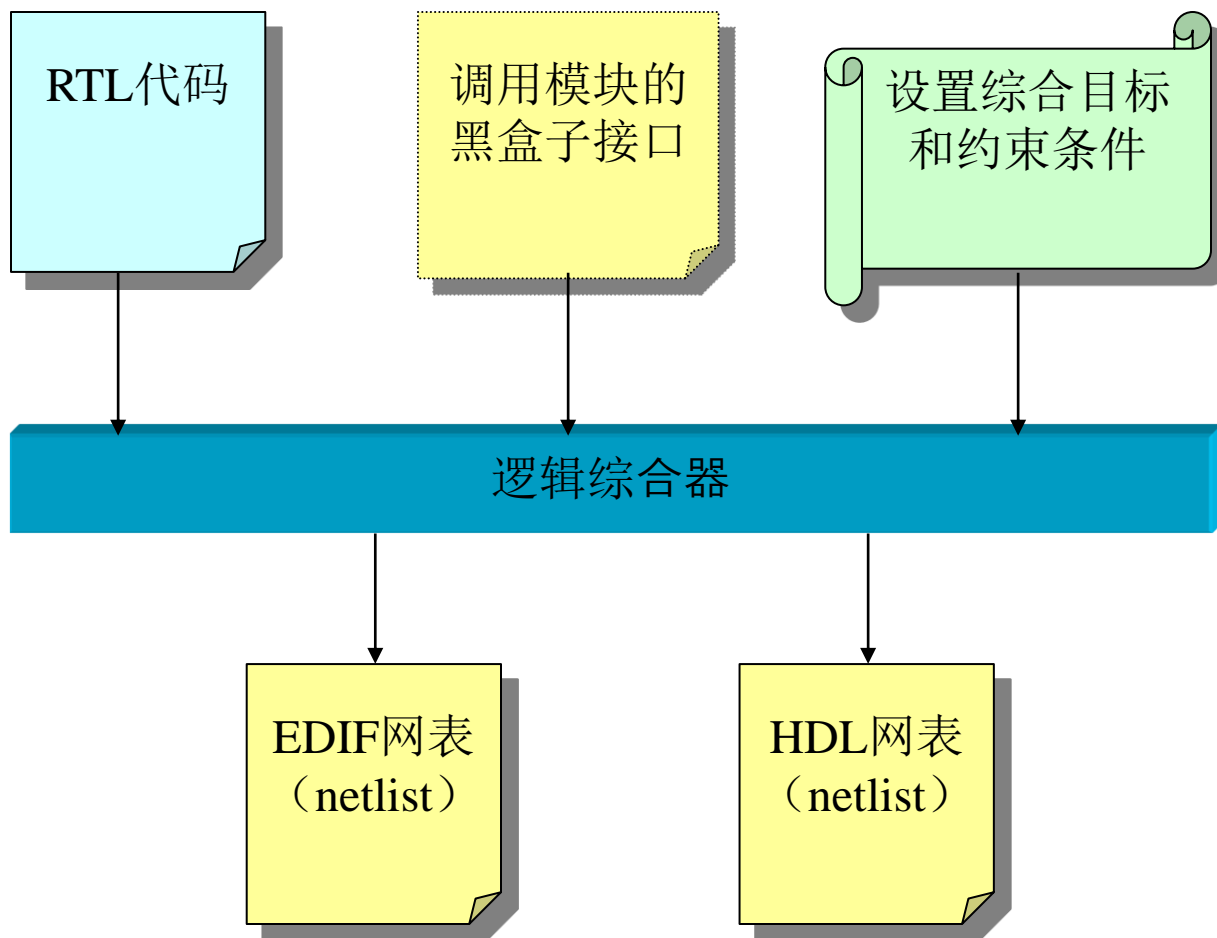
功能仿真

- 对逻辑功能进行验证：
 - 不考虑时序问题，认为门都是理想门，没有延时；
 - 详细一些的可以认为门延时都是一样，而忽略互连线的延时。

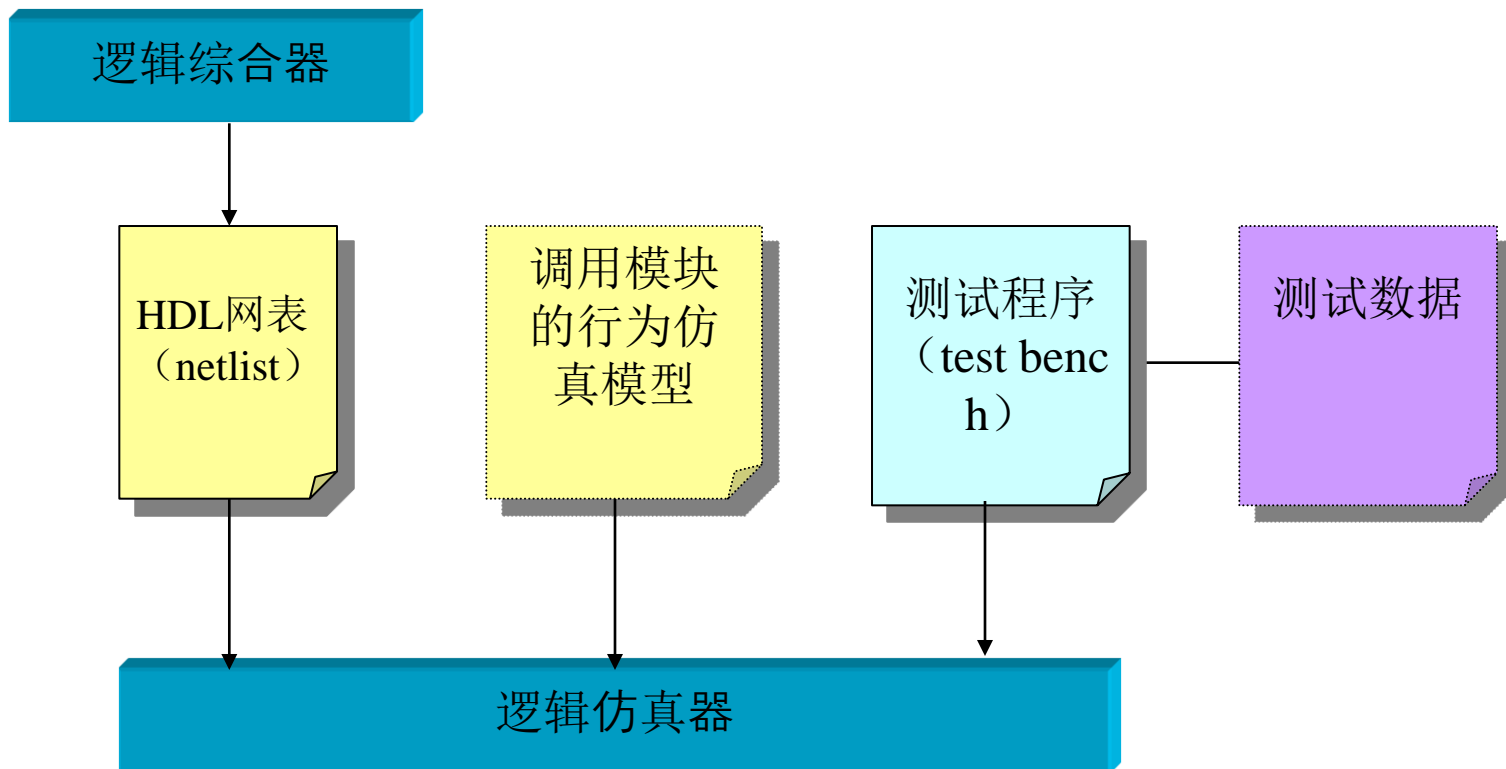


逻辑综合

- 通过映射和优化，把逻辑设计描述转换为和物理实现密切相关的工艺网表：



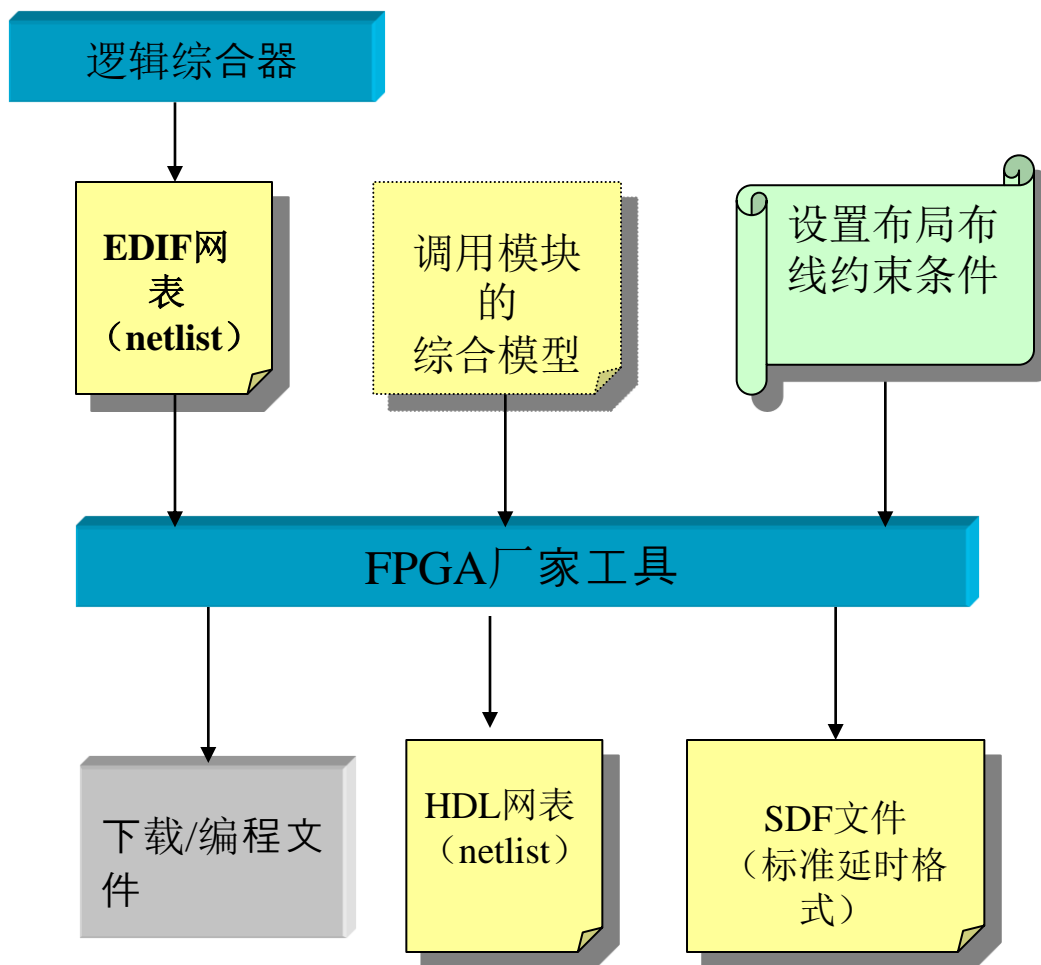
门级仿真



一般来说，对FPGA设计这一步可以跳过不做，
但可用于debug综合有无问题

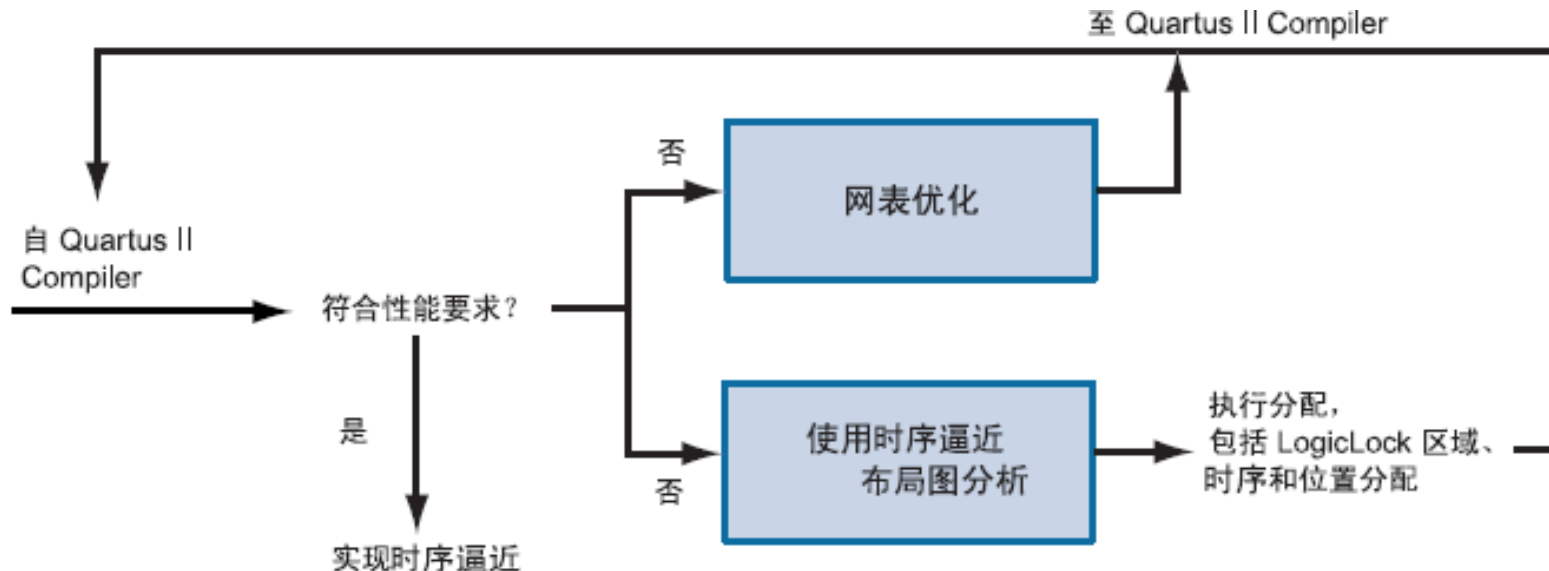
布局布线

- 将综合生成的网表，在FPGA内部进行布局布线的设计，并最终生成用于下载的二进制配置文件；

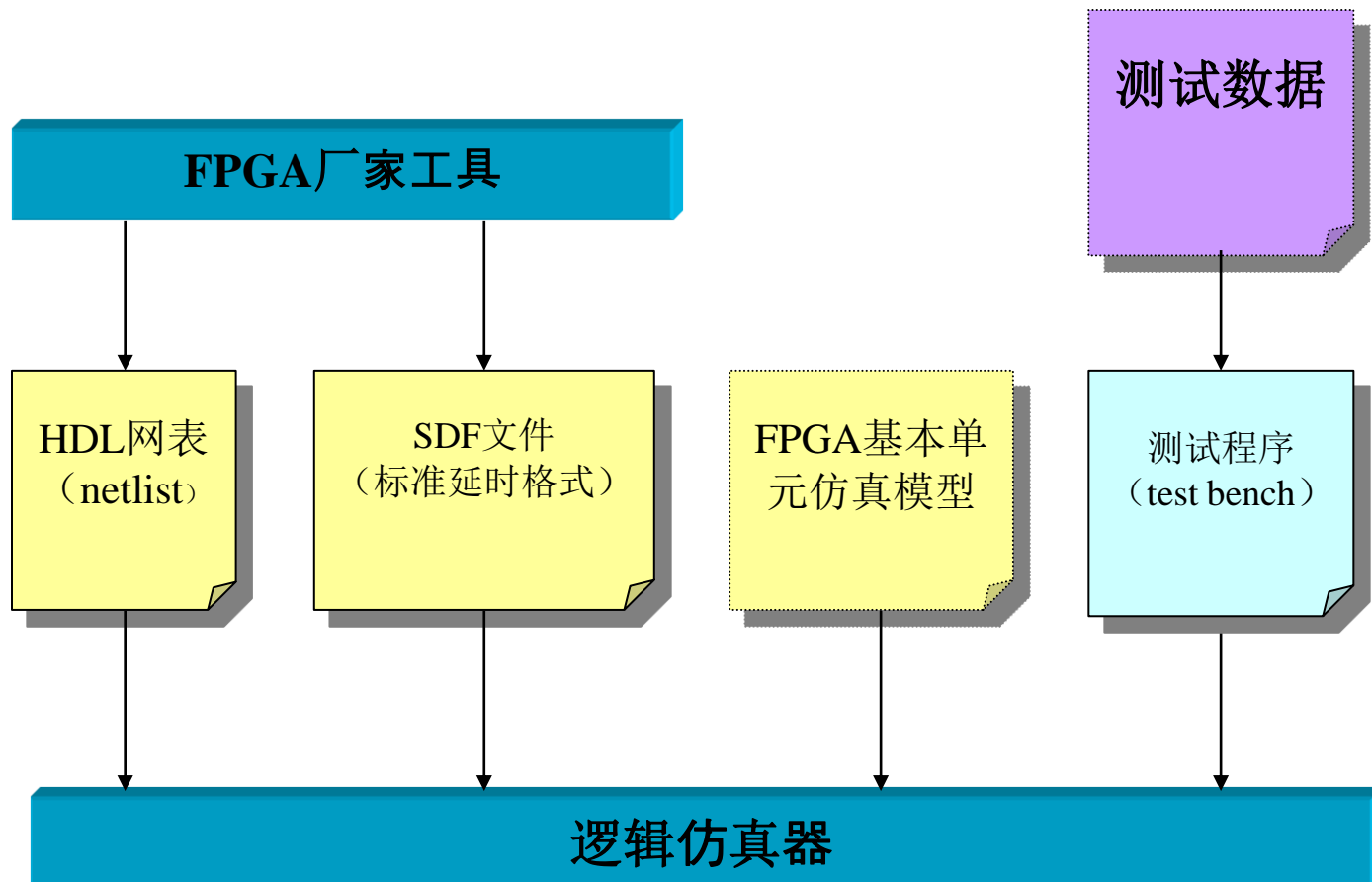


时序逼近

- 时序逼近流程是一个推荐的设计方法可以帮助设计满足它们的时序目标



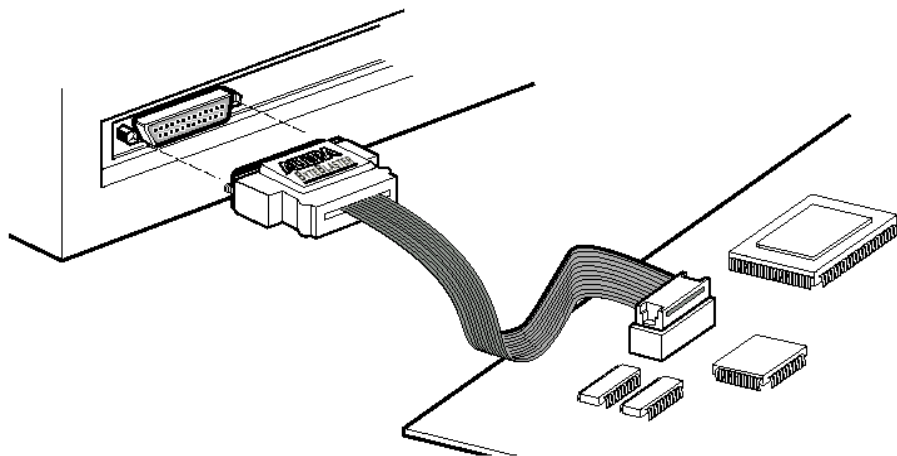
时序仿真



物理验证

- 将生成的二进制配置文件下载到FPGA上，进行实际的功能和时序的测试；
- 由于FPGA常常是作为整个系统一部分，因此还应该将FPGA放到整个系统中进行验证，整个系统工作正常，才算完成了开发过程。

Figure 1. ByteBlaster Parallel Port Download Cable



课程安排

- FPGA设计基本流程
- **FPGA设计规范**
- FPGA的验证方法

设计文档

一个完整的软件是由程序、数据和文档三部分组成的。在FPGA电路设计中，撰写完善的设计文档是非常重要的。对于一个比较复杂的设计来说，各个子单元的功能各不相同，实现的方法也不一样，各子单元之间信号时序和逻辑关系也是纷繁复杂的。因此，在设计文档中对整个设计进行详细的描述，可以保证使用者能够在较短时间内理解和掌握整个设计方案，同时设计人员在对设计进行维护和升级时，完善的设计文档也是非常有用的。

设计文档的内容

- (1) 设计所要实现的功能；
- (2) 设计所采用的基本思想；
- (3) 整个设计的组织结构；
- (4) 各个子单元的设计思路；
- (5) 各个子单元之间的接口关系；
- (6) 关键节点的位置、作用及其测试波形的描述；
- (7) I/O引脚的名称、作用及其测试波形的描述；
- (8) 采用的FPGA器件的型号；
- (9) 片内各种资源的使用情况；
- (10) 该设计与其它设计的接口方式等。

软件思维→硬件思维 的转变

- 在电路描述时，必须摒弃软件思维方式，一切从硬件的角度去思考代码的描述。
- 在具体的项目实践中，必须先画好模块的接口时序图，然后画出或者在脑子里形成模块的内部原理框图，最后才是代码实现。
- 企图一开始就依靠“软件算法”思维进行代码实现，最后才分析时序和电路图，是非常不可取的。
- 硬件思维的形成，需要一定的硬件设计训练才能达到，熟练了之后才可能科学地在初始阶段完成模块划分和时序设计。

时序的设计

- 时序是设计出来的，不是仿出来的，更不是凑出来的。
- 先写总体设计方案和逻辑详细设计方案
 - 总体方案主要是涉及模块划分，模块之间的接口信号和时序
- Logic Design的难点在于系统结构设计和仿真验证
 - 提高代码覆盖率

设计规范化

- 设计文档化
 - 设计思路，详细实现等写入文档
- 代码规范化

```
-----
-- Title      : hdbne
-- Project    : hdbn
-----
-- File       : hdbne.vhd
-- Author     : xxx
-- Organization : xxx
-- Created    : 9 Aug 2011
-- Platform   :
-- Simulators  : Any VHDL '87, '93 or '00 compliant simulator will work. Tested with several versions of Modelsim and Simili.
-- Synthesizers : Any VHDL compliant synthesiser will work (tested with Synplify Pro and Leonardo).
-- Targets     : Anything (contains no target dependent features except combinatorial logic and D flip flops with async – reset or set).
-----
-- Description : HDB3 or HDB2 (B3ZS) encoder.
-- P and N outputs are full width by default. Half width pulses can be created by using a double rate clock and strobing ClkEnable and OutputEnable appropriately (high every second clock).
-- HDB3 is typically used to encode data at 2.048, 8.448 and 34.368Mb/s. B3ZS is typically used to encode data at 44.736Mb/s
-- The outputs will require pulse shaping if used to drive the line.
-- These encodings are polarity insensitive, so the P and N outputs may be used interchangeably (swapped).
--
-- Reference   : ITU-T G.703
--
```

文件的头信息

- 所有源文件中都应包含头信息
- 内容
 - 作者信息
 - 修改记录
 - 目标描述
 - 可用参数
 - 复位机制和时钟
 - 关键时序、异步接口
 - 测试方法
- 应该有一个标准模板

端口

- 顺序
 - 每行一个端口，并准确注释
 - 先列输入信号，再列输出信号
 - 参考顺序：时钟、复位、使能、其它控制信号、地址总线、数据总线……

设计中注意事项 — 软件设计

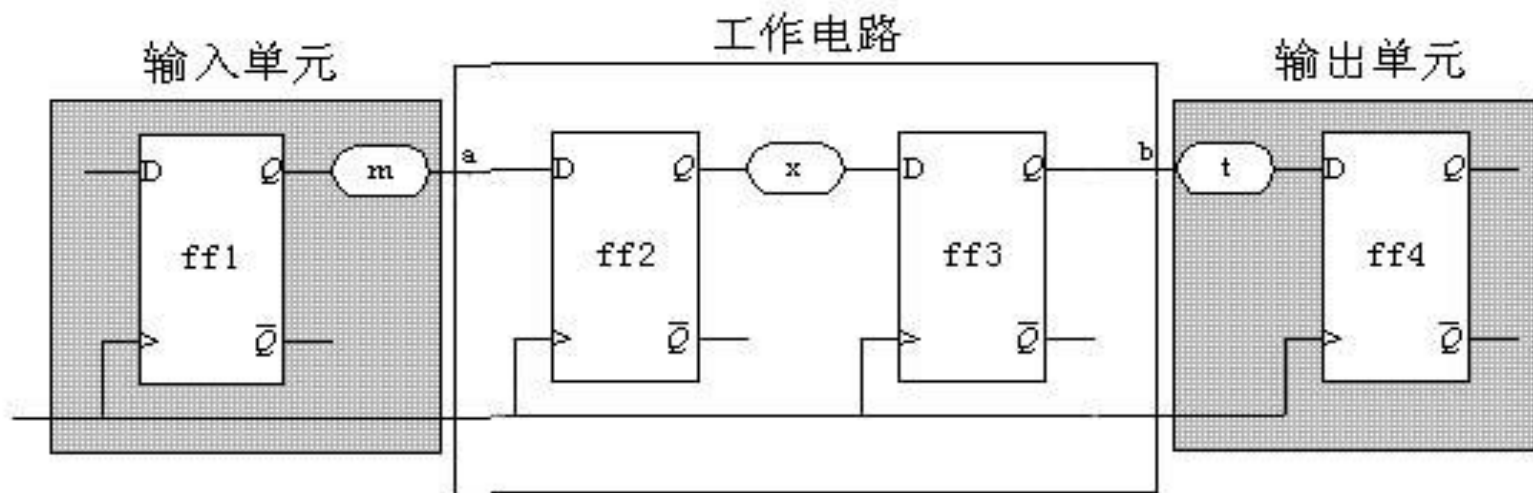
- 合理规划设计
- 敏感信号的选取
- 一个模块尽量只用一个时钟
- 尽量在底层模块上做逻辑，在高层尽量做例化，顶层模块只能做例化，禁止出现任何胶连逻辑（glue logic）
- 进入FPGA的信号先同步
- 避免使用latch
- 多看RTL门级电路
- 多用同步电路，少用异步电路（reset全局异步，本地同步）
- 多用全局时钟，少用门控时钟

设计中注意事项－软件设计

- FSM
 - FSM 与non-FSM 逻辑的分割
 - 组合逻辑与时序逻辑的分割
 - 状态向量的编码方法
 - 缺省（复位）状态的分配
- 避免在端口映射中使用表达式
- 所有模块的输出都要寄存器化，以提高工作频率，这对设计做到时序收敛也是极有好处的

关于寄存输入和寄存输出

- 数字系统中，各模块应采取（寄存输入和）寄存输出，这样做有如下优点：
 1. 模块化清晰(特别是寄存输出)；
 2. 提高系统最高工作速率；
 3. 有利于整个系统和单个模块分别进行静态时序分析。



HDL描述方式(管脚输入信号处理)

- 输入电路
- `always @(negedge rst or posedge clk)`
- `if(!rst)`
- `calc <= 0;`
- `else`
- `calc <= dina + dinb;`
- `dina, dinb` 对应于芯片的输入管脚

- 输入电路
- `always @(negedge rst or posedge clk)`
- `if(!rst) begin`
- `in_rega <= 0;`
- `in_regb <=0;`
- `end`
- `else begin`
- `in_rega <= dina;`
- `in_regb <= dinb;`
- `end`
- `always @(negedge rst or posedge clk)`
- `if(!rst) in_regb <=0;`
- `else calc <= in_rega + in_regb;`

HDL描述方式(管脚输出信号处理)

- always @(tempa or tempb)
- case(tempa)
- 0: dout <= tempb+1;
- 1: dout <= tempb+3;
-
- Default : dout <= 0;
- endcase
- tempa,tempb 对应于芯片的输出管脚

- always @(negedge rst or posedge clk)
- if(!rst)
- dout <= 0;
- else
- case(tempa)
- 0: dout <= tempb+1;
- 1: dout <= tempb+3;
-
- Default : dout <= 0;
- endcase

设计中注意事项－硬件设计

- 下载配置方式的选取
- 供电电压 VCCINT VCCIO
- 电源的滤波
- 空闲I/O的处理
- 时钟的走线
- 输出调试信号
- 器件选取

选型指南

特殊要求

逻辑单元

PLL

I/O个数

DSP模块

驱动能力

RAM 大小

工作环境

硬件乘法器（DSP48E1）个
数

课程安排

- FPGA设计基本流程
- FPGA设计规范
- **FPGA的验证方法**

设计验证

- 随着集成度的提高，系统的规模日益庞大且复杂。
- 强壮的系统应有完备的验证作保障。
- 验证工作量远大于系统设计工作量。
- 验证工作的基本技术和方法。

设计验证的概念

- 检验ASIC、FPGA等设计的功能和时序是否满足要求，以保证功能的正确实现
- 验证方式
 验证方法 + 验证工具
- 一般采用逻辑仿真的方式来验证功能和时序
 需要仿真工具, 需要输入向量作为激励

验证技术

局限性很大

- 1、设计规模越来越大复杂
- 2、模型检验所描述的特性有限

目前主流的两类“验证技术”：

(1) 基于形式化的验证-----通过数学的方法，证明设计的功能是否与规范一致。

- ◆ 等价性检验：比较两个设计是否完全等价。

两个网表比较，网表与RTL代码比较

- ◆ 模型检验：根据设计的RTL代码，提取有限状态机并穷举搜索设计状态空间，验证设计特性。

模型检验工具：Cadence的FormalCheck、IBM的Sugar和
Sypopsys的Formality

验证技术

(2) 基于TestBench的验证(目前主要的验证方式)

Testbench——测试平台

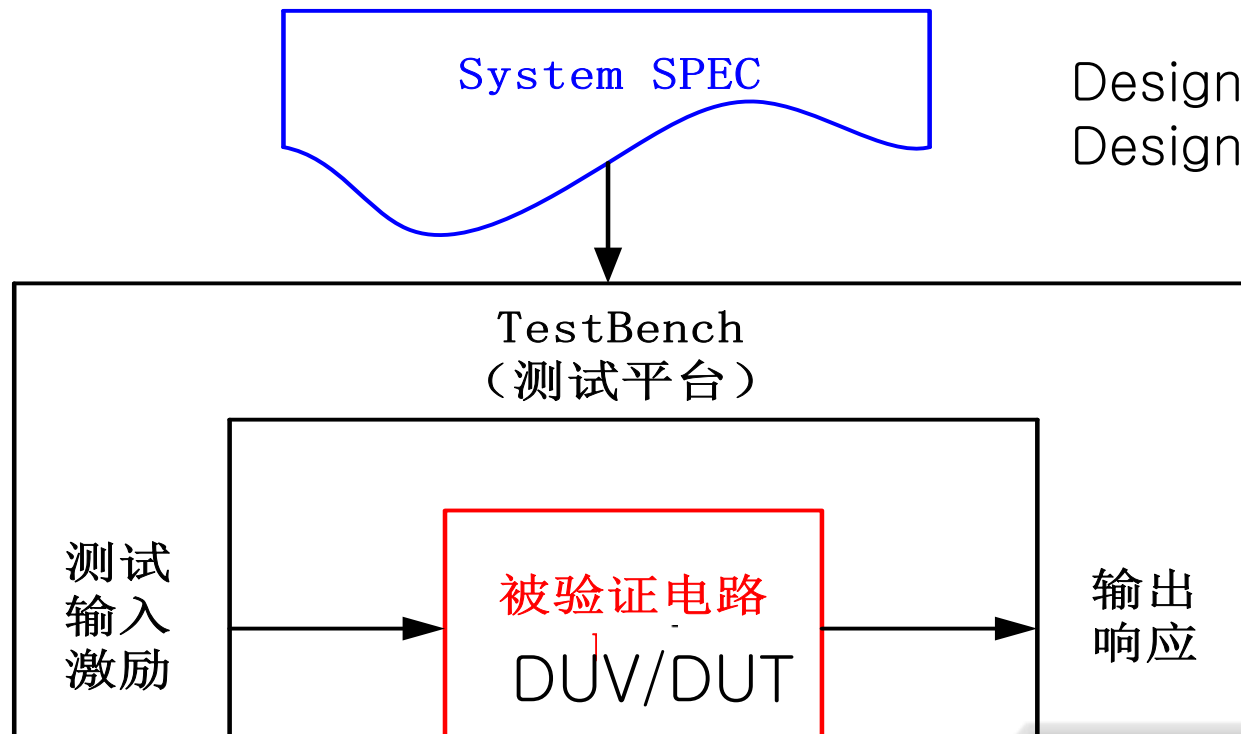
即： Testbench产生激励给被验证设计（DUV）或待测设计（DUT），同时检查DUV/DUT输出是否满足要求

验证技术

解释：

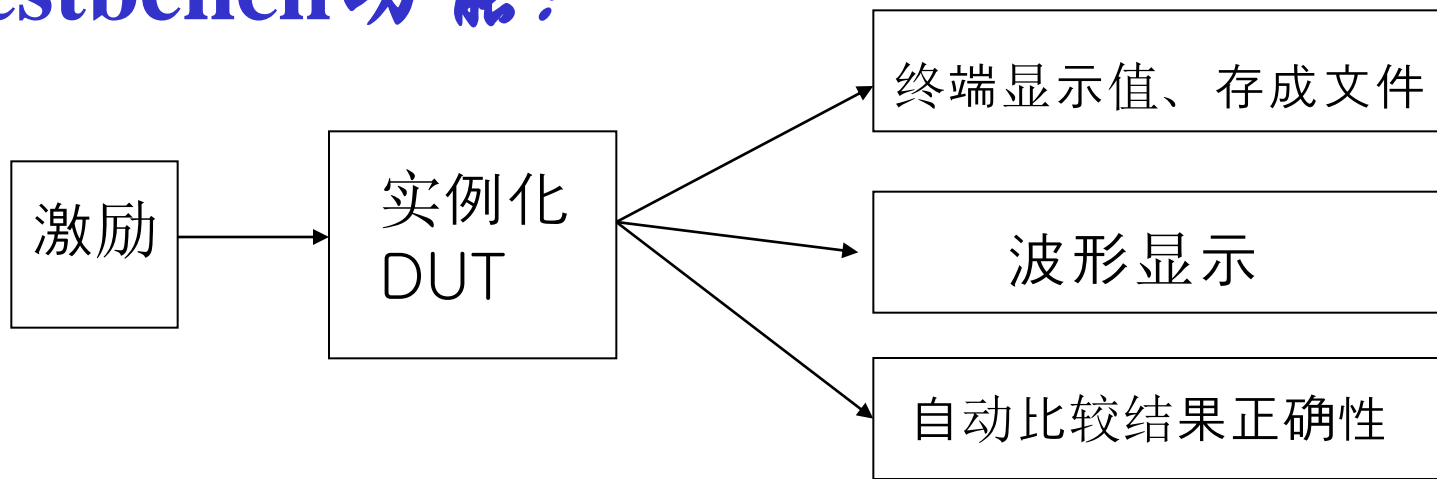
1. 黑盒验证法
2. 白盒验证法
3. 灰盒验证法

TB 结构模型



Design Under Verification
Design Under Test

Testbench 功能：



- 为DUT/DUV提供激励信号
- 正确实例化DUT/DUV
- 将仿真数据显示在终端或者存为文件，也可以显示在波形窗口用于分析
- 复杂设计可以使用EDA工具，自动比较仿真结果与理想值。

仿真工具

语句覆盖

触发覆盖

路径覆盖

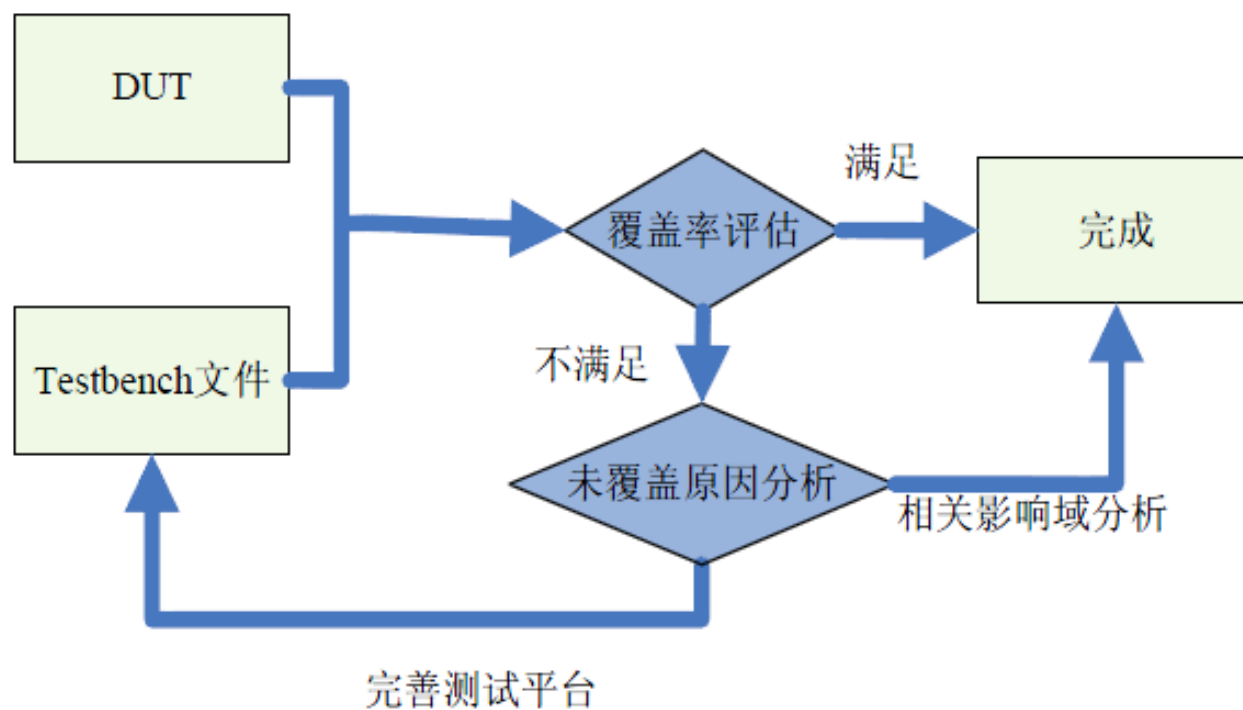
表达式覆盖

使用代码覆盖技术必须非常了解设计细节，通过代码覆盖分析工具了解哪些语句、路径已经被执行，那些表达式已经被执行，那些过程没有被触发等等，然后修改测试程序，提高代码覆盖率。

提高覆盖率可以提高测试的完备性。

代码覆盖率评估方法

基于覆盖率统计的仿真验证



代码覆盖率评估方法

常用代码覆盖率

- 语句覆盖（Statement coverage）
- 分支覆盖（Branch coverage）
- 条件覆盖（Condition coverage）
- 表达式覆盖（Expression coverage）

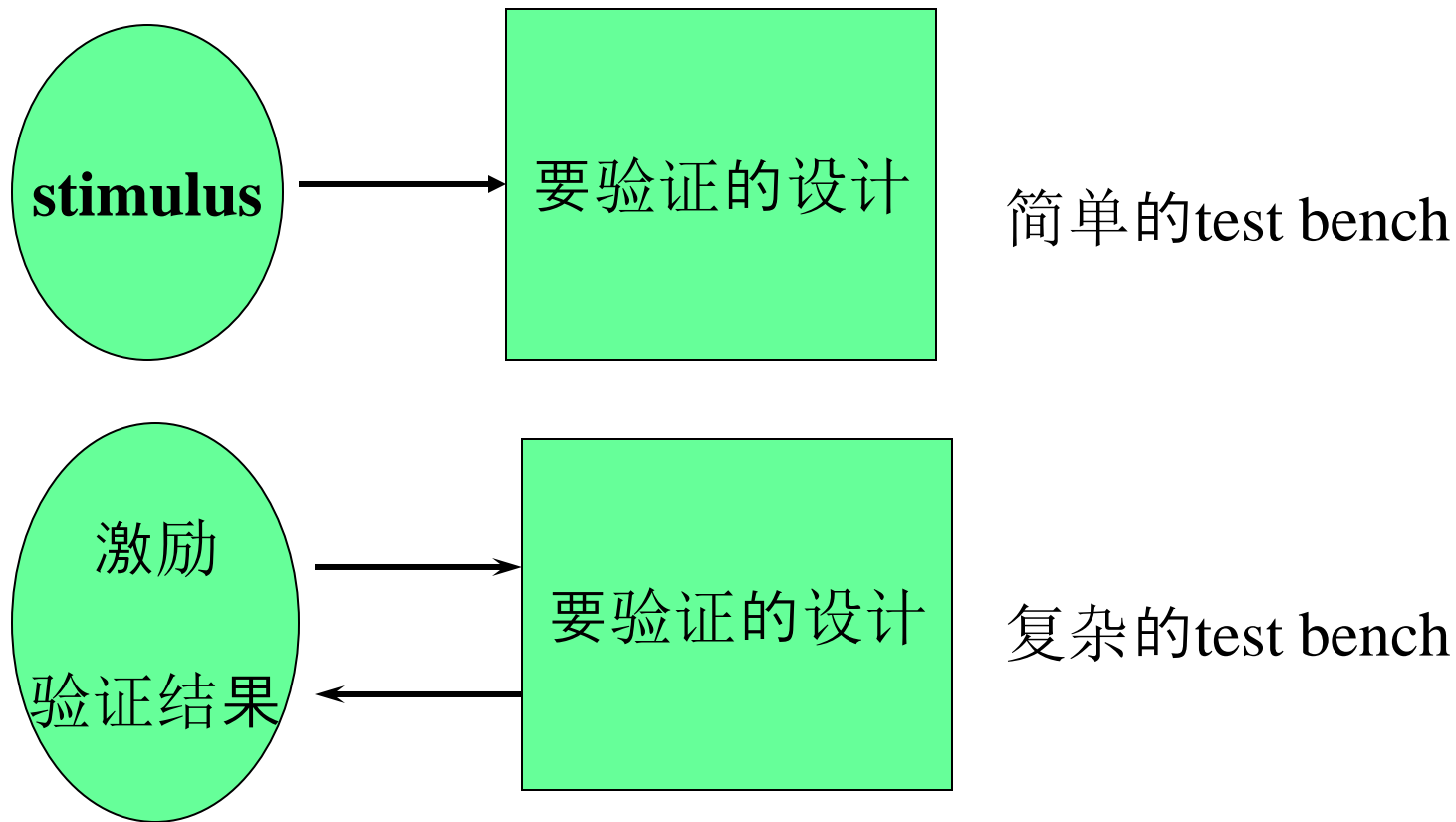
为什么要写Testbench

- 画波形图只能提供极低的功能覆盖率
- 画波形图无法实现验证自动化
- 画波形图难以定位错误
- 画波形图的可重用性和平台移植性极差
- 通过画波形图的验证速度极慢

•RTL功能仿真；

•FPGA功能及时序仿真；

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。

怎么写好testbench

- 基础：熟练掌握行为级语法
- 提高：深刻理解层次结构在testbench的使用



行为级语法

Testbench和RTL的区别

- RTL：平面化的电路结构描述
- Testbench：时序化的过程描述
- RTL：主要由状态机来描述，
- 测试向量：软件化的时序过程，可用流程图的方式描述。

RTL建模 VS. 行为级建模

- RTL级建模是面向物理电路的描述，语句需要有相应的物理电路与之对应，应避免不可综合的语句；RTL级建模时要严格遵从代码规范，如避免产生锁存器、模块输出信号要寄存、对跨时钟域的控制信号要用D触发器打两拍等；RTL级建模的关键是“thinking in hardware”。
- 行为级建模则是面向电路行为的描述，它比RTL级建模更加灵活：它可以通过各种方法描述电路行为，而不需要有物理电路与之对应；行为级的描述没有严格的代码规范，更多要求在于提高抽象层次、规划testbench结构等方面；它带有很强的软件设计思想；行为级的语法包括了RTL级的语法，覆盖了Verilog语法中的大部分，甚至可以说，行为级语法才是真正反映了Verilog语法的实质；行为级建模的关键是“thinking in behavior”。

数据类型

- 在RTL级，我们使用的数据类型主要有wire、reg及integer;
- 而在行为级中，则要广泛的多，所有Verilog语法中定义的数据类型都可以使用

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

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
```

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。

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

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
```

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(               //模块例化
    .a(a_input)
    ,.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
```



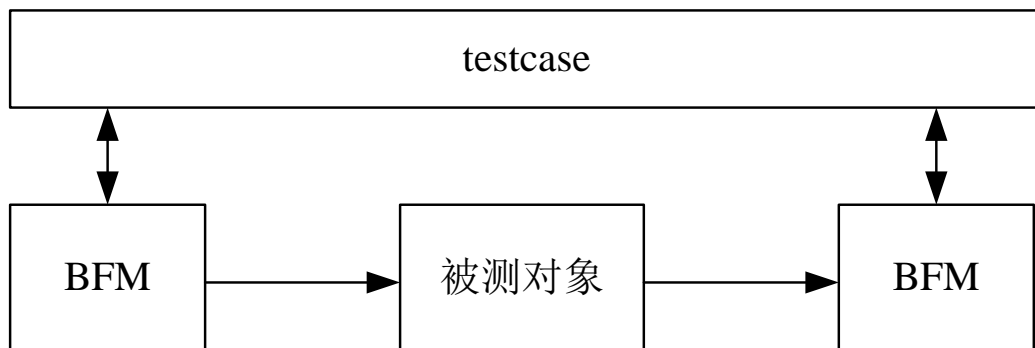
Testbench的结构

Testbench的结构

- 一个好的testbench结构应该包含两点要求：第一，层次清晰；第二，具备较好的重用性。这两点要求之间也有很重要联系：只有层次清晰的testbench才是可重用的，重用性也将使testbench层次更加清晰。

Testbench的层次

- 对于简单的testbench，一般可分为两层：底层是BFM，上层是testcase。BFM中不能出现与具体应用有关的功能，它只负责屏蔽底层的接口时序，或者说它只提供一个通用的传输平台，而不去关心它“承载”的是什么数据；具体应用由上层的testcase实现

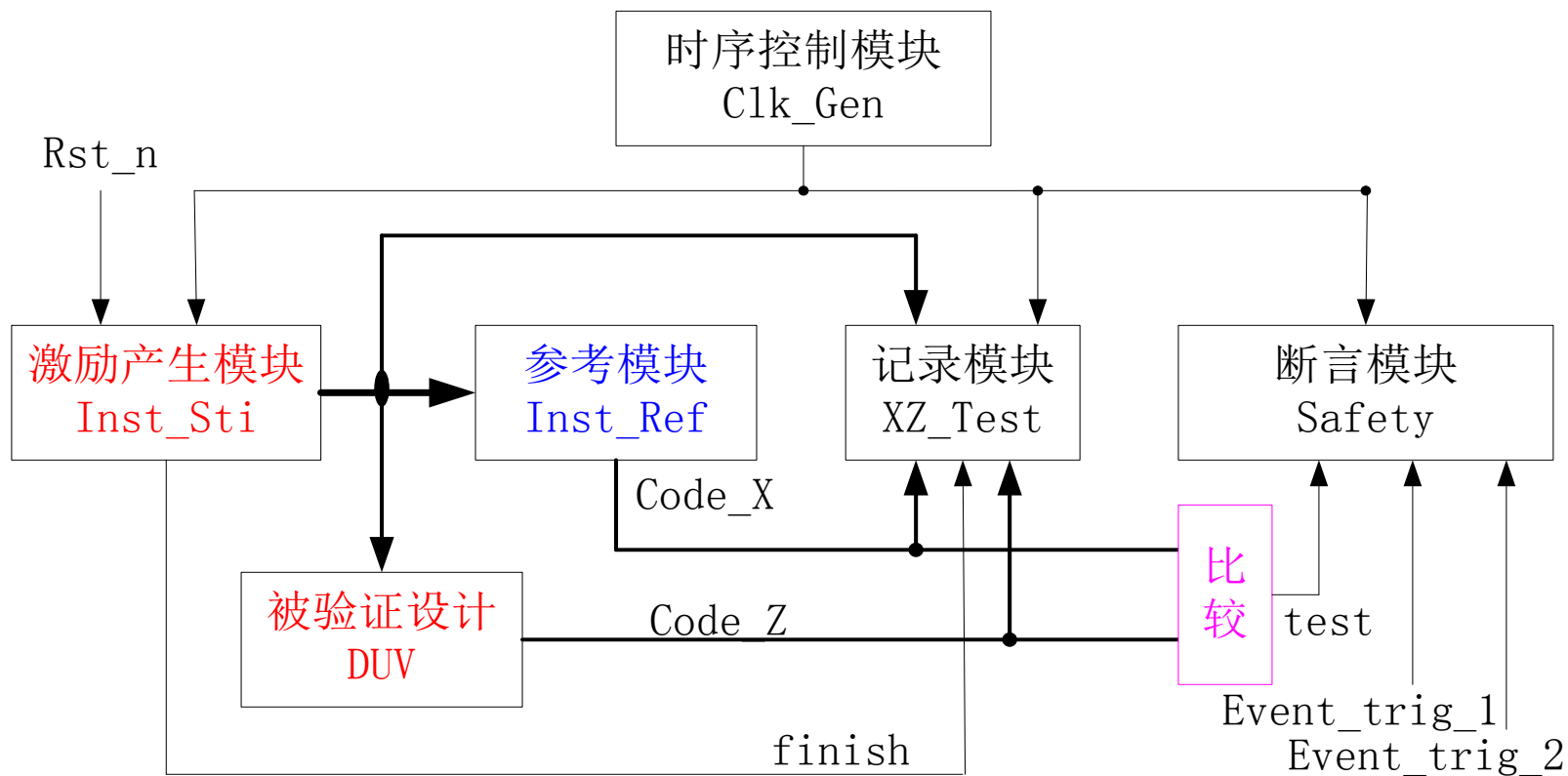


验证程序(TestBench)的组成

一个典型的Testbench的六个组成部分：

1. **DUV** (被验证的设计)--- Design Under Verification, 可能是RTL设计, 也可以是网表。
2. **输入激励** --- 产生DUV需要的各种输入信号。
3. **时序控制模块** --- 产生TB和DUV所需的时钟信号。
4. **参考模块**--- 产生预期信号 (行为级编码模块/以验证过正确的设计) 。
5. **诊断记录** --- 相关信号变化情况的记录。
6. **断言检查器** --- 对特定的信号形式检测。

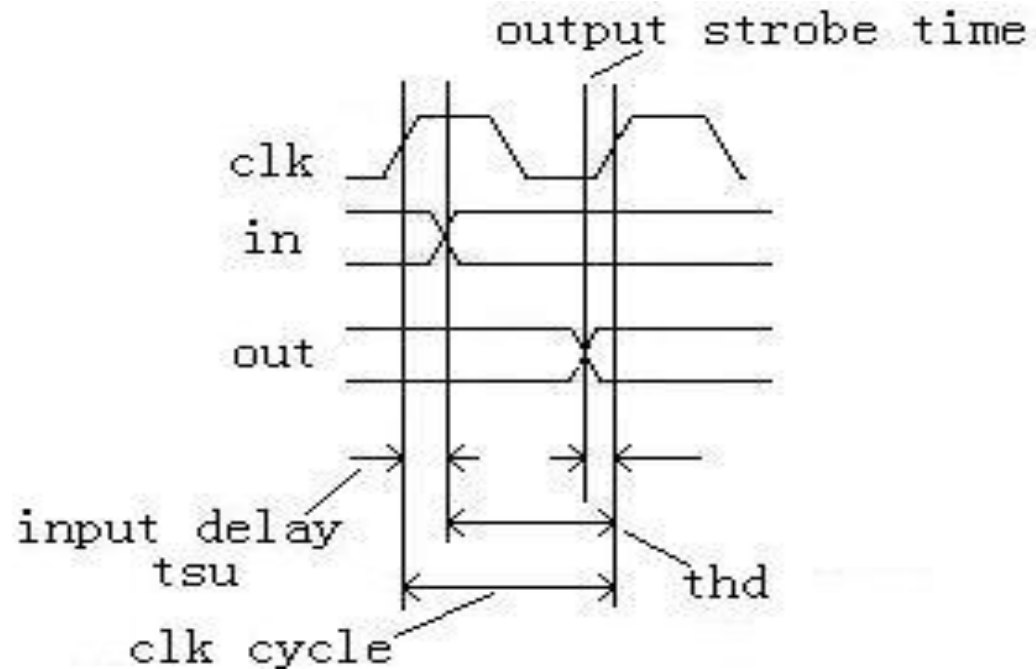
验证程序 (TestBench) 的组成



一、FPGA Testbench通用格式

- 要求有输出检测；
- 测试向量中只能有一个主控时钟；
- 以主控时钟的CYCLE宽度为测试周期，一个周期内输入信号只能改变一次；
- 输入信号(包括双向口处于非输出模式时)必须确定，即不能为0或1以外的值；
- 输出信号尽量控制使其不要处在除0和1以外的其它状态；

二、推荐 Testbench 格式



Input delay和 output strobe time的值均可调节

三、Testbench时钟设计

1、要求：

- ❖ 因测试向量中只能有一个主控时钟，所以要求RTL代码中时钟数(树)尽可能少，不同时钟之间最好要有同相，同频，倍频的关系，以免X状态的传递导致整个仿真的崩溃。
- ❖ 时钟以高电平开始第一个CYCLE，这样形成的第一个CYCLE是一个完整的CYCLE。

2、实例：

```
parameter PCY = 40, PHC = 20;  
initial  
begin  
    RX_ZT_PCLK <= 1;  
    forever  
        #PHC RX_ZT_PCLK <= ~RX_ZT_PCLK;  
end
```

四、输入信号处理

要求：

- 基于时钟CYCLE，每个CYCLE仅可变化一次，
- 按INPUT DELAY来调节输入延迟，以达到在不同测试要求测试TSU，THD的需要，尽量不要用动态的数值来控制输入延迟。
- 例：

```
parameter PCY = 40, PHC = 20, indelay = 2;
#indelay;           //initial input delay;
RTX_ZT_PFRAME_tb <= 0;
RTX_ZT_PIRDY_tb   <= 0;
RX_ZT_PIDSEL      <= 1;
#PCY;
RTX_ZT_PFRAME_tb <= 1;
RX_ZT_PIDSEL      <= 0;
RTX_ZT_PCBE_tb    <= 4'b0000;    //ALL BYTE ENABLED;
if(i==0) RTX_ZT_PAD_tb <= 32'h55555555; //write;
else if(i==2) RTX_ZT_PAD_tb <= 32'haaaaaaaa; //write;
      else RTX_ZT_PAD_tb <= 32'hZZZZZZZZ; //read;
#(PCY*3-indelay);           //right at the rising edge of clk;and wait for target ack;
while(!PCI_DATA_TRANSFER) #PCY;
#indelay;                   //2ns after the rising edge of clk;
RTX_ZT_PIRDY_tb <= 1;
#(PCY*10);
```

五、仿真结果保留(二进制格式)

- 测试向量稳定后，可以将输入向量、输出结果保留，以进行对比测试，ASIC流程各阶段的仿真，并监测仿真结果。
- 因为输入和输出信号在一个CYCLE内只变化一次，所以可在一个CYCLE的末期采样输入、输出、三态使能信号存入文件中。
- 时钟信号无需保留，方便后续仿真自由决定时钟频率。

例：

```
#define strobe = 1;
initial
begin
    desc = $fopen ("zx2701_ptarget.dat") ;
    #(SCY- strobe ); //1ns before clk rising edge,
    forever
    begin
        $fwrite(desc,"%d",din);
        $fwrite(desc,"%d",dout );
        $fwrite(desc,"%d",dinout );
        $fwrite(desc,"%d",U_ENTITY.dinout_oen);
        $fwrite(desc,"\n");
        #(SCY);
    end
end
```

记得在仿真结束之前关闭文件。

```
.
#(PCY*20)
    $fclose (desc) ;
    $stop;
    $finish;
end
```

小结：

- 测试数据从文件中读取
- 输入信号按文件中读出值在INPUT_DELAY的控制下加入
- 文件中读出的输出信号用于和仿真出的输出结果对照比较
- 从文件中读出的双向信号值，在Out_en无效时作为输入信号值加入，反之作为输出对照值和仿真出的双向口的输出结果对照比较。

例：输入部分

```
initial //txt testbench reader;
begin
    $readmemb("zx2701_gports_fc.dat",mem_a);
    for(i=0;i<filelength;i++)
    begin
        one_line = mem_a[i];
        #indelay;
        RX_ZT_PRST_N          <= one_line[0];
        TX_ZT_PREQ_N_tb        <= one_line[1];
        RTX_ZT_PFRAME_tb       <= one_line[2];
        TX_ZT_FRAME_OEN_N_tb   <= one_line[3];
        #(SCY-indelay); //right to clk rising_edge;
    end
    $stop;
    $finish;
end
assign RTX_ZT_PFRAME = (TX_ZT_FRAME_OEN_N_tb)?R
TX_ZT_PFRAME_tb:1'bz;
```

文件输入操作

- Verilog中有两个系统任务可以将数据文件读入寄存器组。一个读取二进制数据（\$readmemb），另一个读取十六进制数据（\$readmemh）。
- ```
$readmemb (" file_name", <memory_name>);

$readmemh (" file_name", <memory_name>);
```

filename指定要调入的文件。  
mem\_name指定存储器名。

# 指定地址的文件读操作

- \$readmemb

\$readmemb ("file\_name", <memory\_name>, <start\_addr>);

\$readmemb ("file\_name", <memory\_name>, <start\_addr>, <finish\_addr>);

- \$readmemh

\$readmemh ("file\_name", <memory\_name>, <start\_addr>);

\$readmemh ("file\_name", <memory\_name>, <start\_addr>, <finish\_addr>);

start和finish决定存储器将被装载的地址。start为开始地址，finish为结束地址。如果不指定开始和结束地址，\$readmem开始读入数据，从存储器的最低地址开始存放。

# 文件读操作举例 (1)

```
`timescale 1ns/10ps
module myrom(read_data,addr,read_en_);
input read_en_;
input [3:0] addr;
output [3:0] read_data;

reg [3:0] read_data;
reg [3:0] mem [0:15];

initial
 $readmemb("my_rom_data.txt",mem,4'd2,4'd4);

always @ (addr or read_en_ or read_en_)
 if(!read_en_)
 read_data=mem[addr];
endmodule
```

|      |      |
|------|------|
| xxxx | 0000 |
| xxxx | 0101 |
| 0000 | 1100 |
| 0101 | 0011 |
| 1100 | 1101 |
| xxxx | 0010 |
| xxxx | 0011 |
| xxxx | 1111 |
| xxxx | 1000 |
| xxxx | 1001 |
| xxxx | 1000 |
| xxxx | 0001 |
| xxxx | 1101 |
| xxxx | 1010 |
| xxxx | 0001 |
| xxxx | 1101 |

mem

my\_rom\_data.txt

# 文件格式说明

- 可以指定二进制（b）或十六进制（h）数
- 用下划线（\_）提高可读性。
- 可以包含单行或多行注释。
- 可以用空格和换行区分存储器字。
- 可以给后面的值设定一个特定的地址，格式为 @(hex\_address)
  - 十六进制地址的大小写不敏感。
  - 在@和数字之间不允许有空格。

## 文件读操作举例(2)

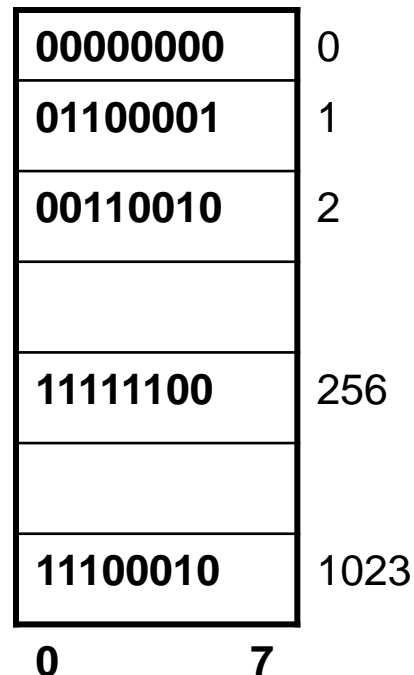
```
$readmemb("mem_file. txt", mema);
```

文本文件：mem\_file.txt

```
0000_0000
0110_0001 0011_0010
// 地址3~255没有定义
@100 // hex
1111_1100
//地址257~1022没有定义
@3FF
1110_0010
```

声明的存储器组

```
reg [0:7] mema[0:1023]
```



```
module readmem;
reg [0:7] mema [0:1023];
initial
$readmemb("mem_file.txt",
mema);
endmodule
```

# 例：输出检测

```
initial
begin
 result = 1;
 #(SCY-tstrobe); //2NS BEFORE CLK RISING EDGE;
 forever
 begin
 result = (TX_ZT_PREQ_N_tb == TX_ZT_PREQ_N)
 && (TX_ZT_FRAME_OEN_N_tb ||
 (RTX_ZT_PFRAME_N == RTX_ZT_PFRAME_t
b))
 #SCY;
 if(!result)
 $write('result error,your simulation may meet some error,at c
ycle: %d\n',i);
 end
end
```

# 文件输出

- `$monitor`, `$display` 等系统任务可以将结果输出到标准输出设备，相似的系统任务（`$fmonitor`, `$fdisplay`）可以将结果输出到文件中。
- `$fopen` 打开一个文件并返回一个多通道描述符（MCD）。
  - MCD是与文件唯一对应的32位无符号整数。
  - 如果文件不能打开并进行写操作，MCD将等于0。
  - 如果文件成功打开，MCD中的一位将被置位。
- 以`$f`开始的显示系统任务将输出写入与MCD相对应的文件中。



# 文件输出举例

...

integer **MCD1**;

...

**MCD1** = \$fopen("<name\_of\_file>");

\$fdisplay( **MCD1**, P1, P2, ..., Pn);

\$fwrite( **MCD1**, P1, P2, ..., Pn); \$fstrobe( **MCD1**, P1, P2, ..., Pn);

\$fmonitor( **MCD1**, P1, P2, ..., Pn);

\$fclose( **MCD1**);

```
`timescale 1ns/100ps
```

```
module inv_tb;
```

```
reg a;
```

```
wire b;
```

```
not inv(b,a);
```

```
initial a=0;
```

```
always #5 a=~a;
```

```
integer file_handler;
```

```
initial
```

```
file_handler=$fopen("data_out.txt");
```

```
initial
```

```
$monitor("%t,%b,%b",$time,a,b);
```

```
initial
```

```
$fmonitor(file_handler,"%t,%b,%b",
$time,a,b);
```

```
//$fclose(file_handler);
```

```
endmodule
```

## 可以执行写文件的系统任务

- \$fdisplay,\$fdisplayb, \$fdisplayh,\$fdisplayo
- \$fwrite, \$fwriteb, \$fwriteh, \$fwriteo
- \$fstrobe, \$fstrobeb, \$fstrobeh, \$fstrobeo
- \$fmonitor, \$fmonitorb, \$fmonitorh, \$fmonitro

## 任务(task)与函数(function)

- 通过把代码分成小的模块或者使用任务和函数，可把电路功能分成许多较小的、易于管理的部分，从而提高代码的可读性、可维护性和可重用性。
- 任务：一般用于编写测试模块，或者行为描述的模块。
  -
- 函数：一般用于计算，或者用来代替组合逻辑。



# 输出验证

# 输出验证

输出验证的手段

- 响应的可视检查
- 产生仿真结果
- 减小采样
- 观察波形

产生激励只完成了一半的工作，实际上，只有差不多30%的工作，另外一部分工作就是验证输出是否期待的结果，而后者是更加耗时的和容易产生错误的。

# 响应的可视检查

- 可视检查的手段

## ❖ 按照一定的时间间隔打印信息

```
parameter INTERVAL = 10
always
begin
 #(INTERVAL);
 $write(...);
end
```

# 响应的可视检查

- 可视检查的手段

- ❖ 按照一定的时间间隔打印信息

- ❖ 基于参考信号打印信息

```
always @(posedge clk)
 $write(...);
```

# 响应的可视检查

- 可视检查的手段
  - ❖ 按照一定的时间间隔打印信息
  - ❖ 基于参考信号打印信息
  - ❖ 基于信号的变换打印信息



```
initial
begin
 $monitor(..., rst, d0, d1);
end
```



# 响应的可视检查

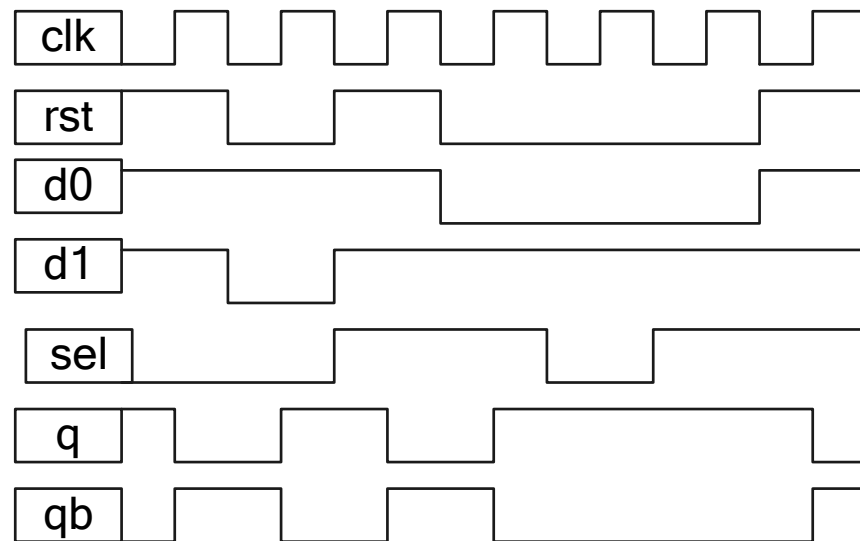
- 可视检查的手段
  - ❖ 按照一定的时间间隔打印信息
  - ❖ 基于参考信号打印信息
  - ❖ 基于信号的变换打印信息
  - ❖ 减小采样，加速仿真

```
initial
begin
 $monitor("...", rst, d0, d1);
 $monitoroff;
 sync_reset;
 load_do(1'b1);
 sync_reset;
 $monitoron;
 $load_d1(1'b1);
 $load_d0(1'b0);
 $sync_reset;
 $monitoroff;
end
```



# 响应的可视检查

- 可视检查的手段
  - ❖ 按照一定的时间间隔打印信息
  - ❖ 基于参考信号打印信息
  - ❖ 基于信号的变换打印信息
  - ❖ 减小采样，加速仿真
  - ❖ 观察波形



# 当模型错误出现错误的时候，如何加速定位过程？

对于一个简单的设计，一个较短的仿真周期，较少的信号变量，可视化的检查非常有效！

对于一个复杂点的设计，成百上千的时钟周期，数百的输入输出信号变量，可视化的检查方法是否还是适用

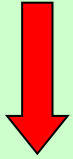


# 输入输出向量

```
Task apply_vector;
 input [...] in_data;
 output [...] out_data;
begin
 inputs <= in_data;
 @ (posedge clk)
 fork
 begin
 #(Thold) ;
 inputs <= ...'bx;
 end
 begin
 #(Td) ;
 if(outputs != out_data) ...
 end
 join
end
```

优点?

```
initial
begin
 // In: rst, d0, d1, sel
 // Out : q, qb
 apply_vector(4'b1110 , 2'b00);
 apply_vector(4'b0100 , 2'b10);
 apply_vector(4'b1111 , 2'b00);
 apply_vector(4'b0011 , 2'b10);
 apply_vector(4'b0010 , 2'b01);
 apply_vector(4'b0011 , 2'b10);
 apply_vector(4'b1111 , 2'b00);
end
```



输出结果与期待值比较，只对不满足条件的结果检查

# 自动检查

```
task rev;
 input rx;
 input [7:0] expected;
 integer period;
 reg [7:0] data;
begin
 period = 100;
 wait(rx == 1'b1);
 #(period / 2);
 data[7] = 0;
 for(i = 0; i < 8; i = i + 1) begin
 #(100) |
 data[i] = rx;
 end
 #period
 rx = 0;
 if (expected != data) $display
 (...);85
end
endtask
```

## Self-checking

将任务封装起来，同时输入期待的值与实际值比较

## 2-4译码器

```
`timescale 1ns/100ps
module dec2x4 (A , B , Enable , Z) ;
Input A , B , Enable ;
output [3:0] Z ;
always @(A or B or Enable)
begin
 if (Enable == 1'b0)
 Z = 4'b1111 ;
 else
 case (A , B)
 2'b00 : Z = 4'b1110 ;
 2'b01 : Z = 4'b1101 ;
 2'b10 : Z = 4'b1011 ;
 2'b11 : Z = 4'b0111 ;
 default : Z = 4'b1111 ;
 endcase
 end
endmodule
```

## 2-4译码器 - Testbench

```
module testbench ;
reg a , b , en ;
wire [3:0] z ;

parameter DLY = 10 ;
// 例化被测测试模块 ;
dec2x4 DUT (
. A (a),
. B (b),
. Enable (en),
. Z (z)
);
//产生输入激励 :
initial
begin
en = 0 ;
a = 0 ;
b = 0 ;

#DLY en = 1 ;
#DLY b = 1 ;
#DLY a = 1 ;
#DLY b = 0 ;
#DLY a = 0 ;
#DLY $stop ;
end
//显示输出结果
always @(en or a or b or z)
begin
$display ("At time %t , input is %b%b%b% , output is
%b", $time , a , b , en , z) ;
endmodule
```

下面是测试模块执行时产生的输出。

```
At time 0, input is 000, output is 1111
At time 10, input is 001, output is 1110
At time 20, input is 011, output is 1101
At time 30, input is 111, output is 0111
At time 40, input is 101, output is 1011
At time 50, input is 001, output is 1110
```

# 时序检测器—“101”

```
module test ;
reg Data , Clock , Detect ;
integer Out_File ;
parameter PCY = 10, PHC = 5;

//待测试模块的应用实例。
Count3_ls DUT (Data, Clock, Detect) ;
initial
begin
Clock = 0 ;
forever
#PHC Clock = ~ Clock ;
end
initial
begin
Data = 0 ;
#PHC Data = 1;
#PCY*10 Data = 0;
#PCY Data = 1;
#PCY*10 Data = 0;
#PCY*2 $stop; // 仿真结束。
end
```

```
// 创建一个记录文件 ;
initial
Out_file = $fopen ("results.txt");
//在文件中保存监控信息 ;
always @(posedge Clock)
begin
if (Detect == 1'b1)
$fwrite (Out_file , "At time %t , Detect out is 1\n"
");
end
endmodule
```