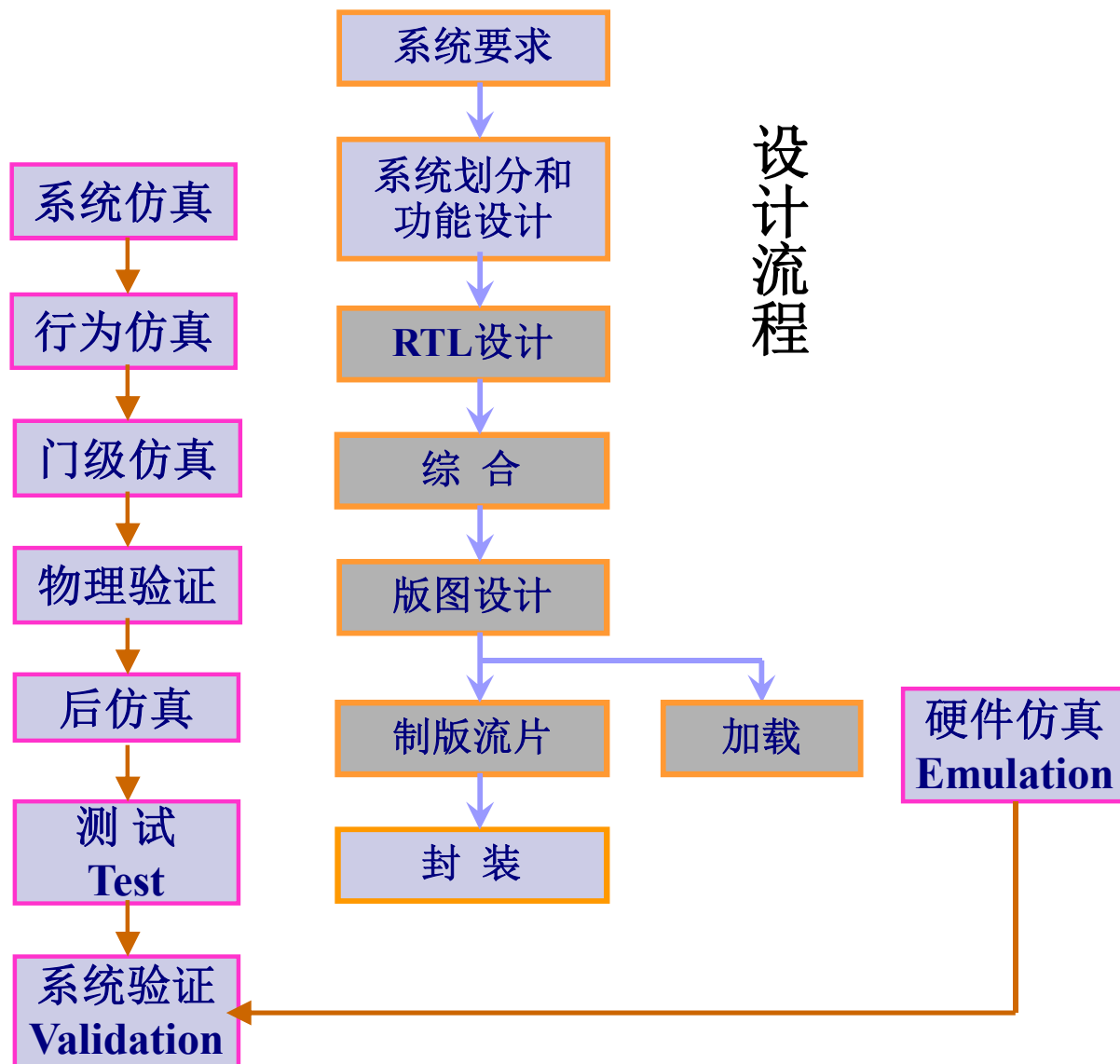


# 目 录

第一章	绪论
第二章	系统级设计
第三章	Verilog HDL硬件描述语言
第四章	逻辑综合
第五章	可编程逻辑器件
第六章	物理版图设计基础
第七章	仿真验证
第八章	集成电路设计发展趋势

# 重要性一：设计的每个环节都需要验证

验证流程



设计流程

集成电路设计 II

设计创意

+ 验证



## 重要二：验证/设计 $\approx 70\%$

- 资料表明，集成电路设计领域，无论是ASIC设计，或FPGA设计，还是单个**知识产权**（IP），或者**片上系统**（SOC）的设计，在整个设计周期中，50%~80%的精力花在验证上。
- 一个设计团队中，验证工程师的数目通常是RTL级编码工程师数目的两倍
- 当一个项目完成以后，所有的代码中，**用于实现测试平台**（Testbench）的代码占80%以上。
- 验证工程师（Verification Engineer）和编码工程师（Coding Engineer）一样是必不可少的
  - 验证效率的提高成为整个设计效率提高的关键
  - 验证也成为EDA设计工具和设计方法（Methodology）的主攻目标



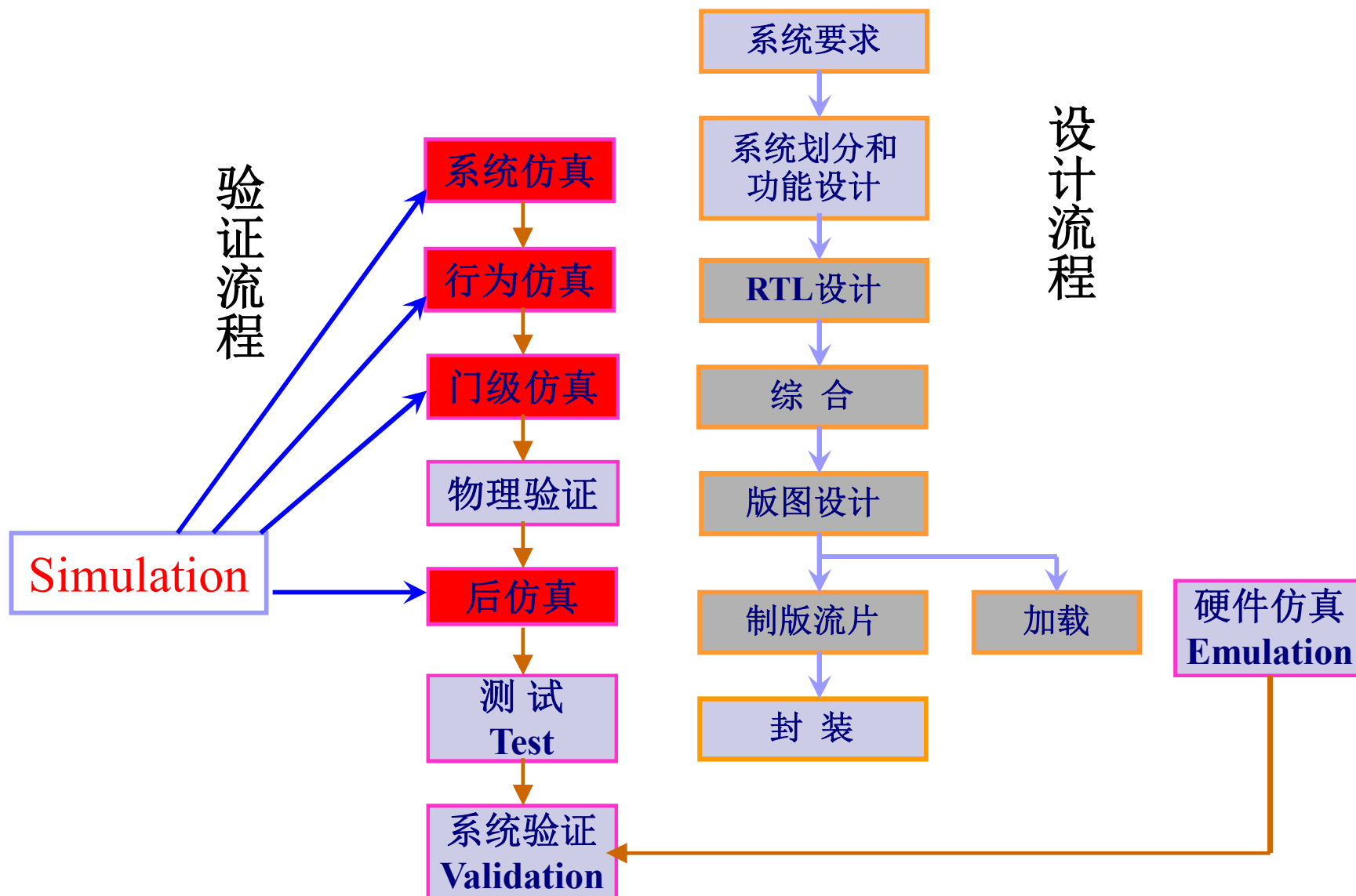
# 验证方法的分类

- 验证方法种类繁多，按照不同的标准可以形成不同的分类方法。
  - 根据验证参照系的不同，验证可以分为目的性验证和等价性验证；
    - 目的性验证 (Intent Verification)：判定一个设计方案是否同其行为规范相符的验证过程。目的性验证最典型的情况是在最高抽象层次完成，其最终结果是建立一套“黄金模型 (Golden File)”，它可以在整个设计过程中作为后续验证的参考模型。
    - 等价性验证 (Equivalence Verification)：判定设计方案的不同层次或不同实现形式之间是否功能一致。等价性验证主要用于验证在设计演化过程中新的设计实现的功能是否符合“黄金模型”。
  - 根据待验证设计的不同，验证可以分为IP模块的验证和集成系统的验证；
    - IP验证：是指对某个IP的功能进行验证的过程。IP验证的关键是对IP模块的内部逻辑进行详细的验证以确保IP模块功能的正确性。
    - 集成系统的验证：是指对包含一个或多个IP的SoC进行集成环境下功能验证的过程。集成系统的验证重点关注系统芯片内部IP模块间接口方式和接口通讯行为的验证。
  - 根据验证方法的技术特点不同，可以分为基于模拟的验证方法、静态验证方法、形式化验证方法、物理验证方法和硬件仿真方法。



# 验证方法

- 基于模拟的验证方法（Simulation）
- 形式化验证方法（Formal Verification）
- 静态验证方法（Static Verification）
- 物理验证（Physical Verification）
- 硬件仿真（Emulation）





# Simulation的基本要素

## ■ 激励生成

- 激励是指驱动待验证设计使之运行起来的输入信号序列。激励可以有多种不同的抽象层次和形式，包括，信号级激励、事务级激励、波形激励等。
- 不同的**Simulation**方法采用的激励形式会有很大差别，如确定型仿真输入、随机型仿真输入、手动编写仿真向量、自动生成仿真激励、端口信号激励、指令代码输入等。

## ■ 结果检验

- 通过检查待验证设计在激励的作用下运行后的输出结果来判断设计是否能正确运行的过程。
- 结果检验方法也多种多样，包括：采用确定型激励自动检查、基于黄金模型的参考模型比较检查、内置监视模块、人工分析仿真波形等



# Simulation的基本要素 (续)

## ■ 效果评估

- 对Simulation的完备程度的检验和估计。效果评估的度量指标称为模拟覆盖率。最常用的覆盖率度量指标为代码覆盖率。
- 由于Simulation非完备性的特点，随着待验证设计的复杂度增加，就更需要重视覆盖率信息的获取以增强对验证过程的掌握和控制。

## ■ 验证环境建模

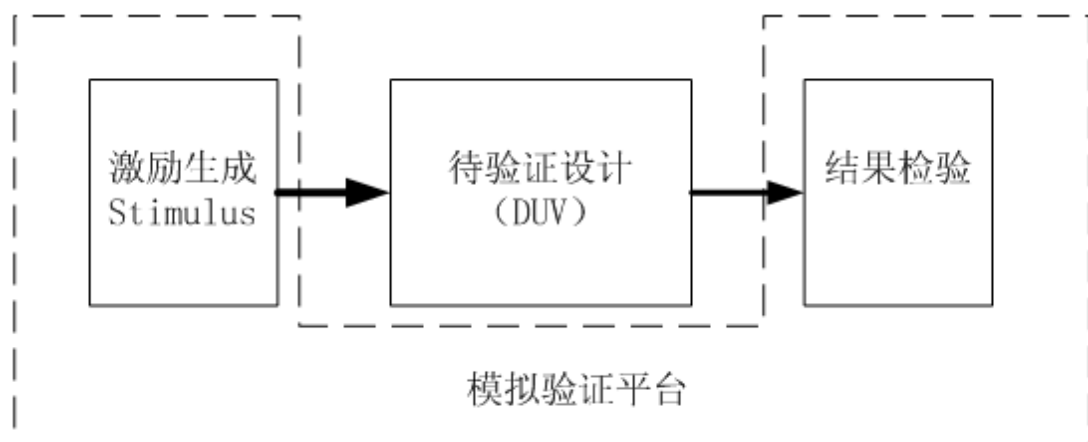
- 由验证激励生成组件、结果检验组件、覆盖率分析组件和待验证设计模型共同组成的模拟验证系统。
- 模拟验证环境的建模方法种类很多，常用的有：基于硬件描述语言的验证平台、基于编程语言接口（PLI: Programmable Language Interface）的验证平台、有些验证工具支持波形输入的验证平台。随着近年来一些先进的验证建模语言被相继提出，还出现了基于事务的验证平台和基于规范的验证平台等。



# 模拟覆盖率

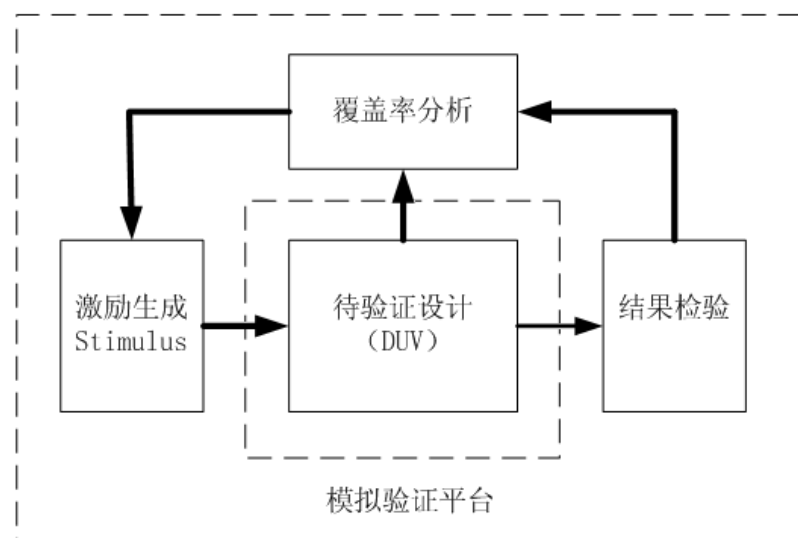
- **代码覆盖率**，表示模拟验证中待验证设计（通常为RTL模型）与某属性要求相关的部分代码被激活的百分比。代码覆盖率可以细分为：**语句覆盖率、信号触发覆盖率、有限状态机覆盖率、触发覆盖率、分支覆盖率、表达式覆盖率、路径覆盖率、信号覆盖率**（对确定信号）
- **器件覆盖率**
- **状态机覆盖率**，这种度量方法来源于形式化验证方法。它的基本思想是把设计看成一个有限状态机，通过统计这个有限状态机的状态遍历状况和状态跃迁状况来获得覆盖率信息。
- **错误模型覆盖率**，这种覆盖率度量方法的思想来源于芯片测试中的故障模型方法。故障模型的提出对于芯片测试起到了至关重要的作用，它使得一系列可测性方法的研究成为可能。类似于芯片测试中使用的故障模型，该方法试图建立功能验证的错误模型，在此基础上统计模拟验证中错误模型的覆盖率情况。
- **标记覆盖率**，这种覆盖率度量方法通过统计设计中描述的信号在模拟过程中的翻转状况来获得度量指标。它通过在模拟过程中对信号施加标记，并分析统计标记的传播情况来获得覆盖率信息。该方法注重模拟中激活的错误在设计端口的可观测性。

# 验证环境建模



传统的模拟验证平台

包含覆盖率分析组件的模拟验证平台





# Simulation

不同阶段仿真的特点不同：

系统仿真：输入输出关系；

RTL仿真：RTL描述；

门级仿真：电路特性的信息（如器件延时，*线延时*）；

后仿真：版图的物理参数（如线延时，寄生电阻，电容等）；

可以采用同样的激励在不同阶段进行仿真。



# RTL级仿真

- 通过施加外部激励，观察信号在寄存器之间的传输情况来判断寄存器之间的连接所表现逻辑是否实现了系统的行为功能；
- 仿真对象：HDL代码；
- 仿真目的：验证HDL代码逻辑功能的正确性；
- 特点：没有任何工艺库方面的信息，只验证理想状态下逻辑功能是否正确，时序安排是否合理，因此仿真输出中不包括器件延时和路径延时，波形很规整，没有过渡态和毛刺没有任何时序方面的信息。



## 门级仿真（Gate Level Simulation）

- 仿真对象：综合后生成的网表；
- 仿真目的：验证在各种环境下，带有器件延迟信息的设计功能的正确性；
- 局限性：网表仅仅含有器件延迟，对于深亚微米（**DSM**）阶段的设计，需要知道更多的时序信息，如连线延迟的信息。



# 门级仿真

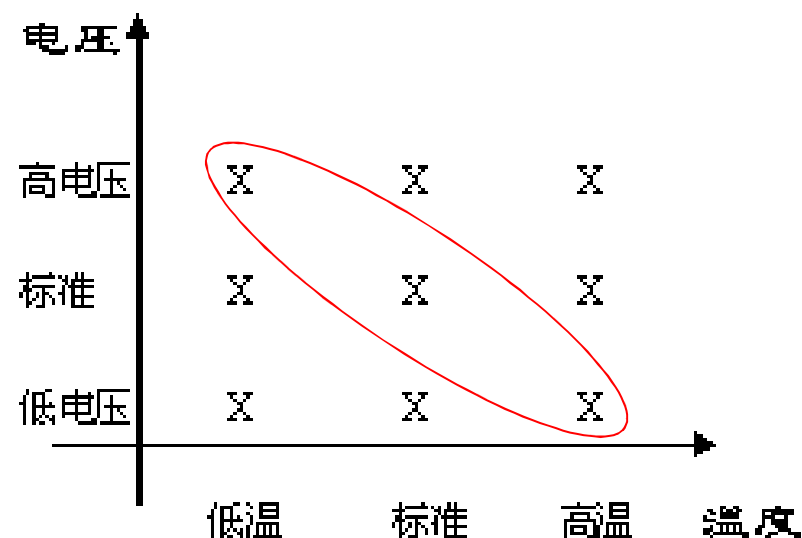
门级仿真是设计流程中第一次加入工业参数的仿真。网表中的所有器件都包含有必须的工业参数，如固有门延时、输入阻抗、驱动能力、温度特性、电压特性、上升时间、下降时间、面积等。正是由于这些工业参数的存在，门级仿真（**gate level**）十分接近于真实芯片的物理测试。

门级仿真的重点是：

- 加入工业参数后的时序重新调整；
- 温度、电压的组合测试。

# 组合测试 (corner test)

门级仿真中的组合测试 (corner test) 验证芯片在不同工作环境中的性能。工作电压允许 $\pm 10\%$ 的偏差，工作温度也有最低温度和最高温度的规定（随芯片的等级和种类而异）。不同的温度和电压可以组合出9个“角落”，故名之。





# 后仿真 (Post Simulation)

- 仿真对象：通过参数提取程序提取出实际版图参数和寄生电阻、寄生电容等寄生参数，进一步生成带寄生参数的器件级网表，对此进行模拟分析。
- 仿真目的：验证在各种环境下，带有器件延迟、连线延迟信息的设计功能的正确性；
- 局限性：高度依赖于库模型建模的准确性；对信号的完整性分析信息不够详细。





# 软件仿真的特点

- 优点：通过大量的测试向量对DUV进行仿真；
- 缺点：
  - 依赖于EDA软件模型的准确性；
  - 依赖于芯片制造商的仿真模型。



# 验证方法

- 基于模拟的验证方法（Simulation）
- 形式化验证方法（Formal Verification）
- 静态验证方法（Static Verification）
- 物理验证（Physical Verification）
- 硬件仿真（Emulation）



# 形式化验证的概念

- 形式化验证就是采用形式化的方法验证设计的正确性。
- 形式化方法是一个很复杂的概念，一般来说，它是用具有形式语义的记号和工具明确地表述所要设计的计算机系统的设计要求，即给出系统规范（**Specification**），并根据系统规范利用上述记号和工具对系统具有的性质和最终实现的正确性进行严格的证明。
- 形式化方法应用广泛，这里所涉及的形式化验证则仅限于用形式化方法进行硬件设计的功能验证范畴。



# 形式化验证的特点

- 形式化验证对设计进行数学分析，是一种数学证明的方法，而不是进行模拟，因而不需要模拟激励，也不需要建立验证环境。
- 形式化验证方法最大的优点是，对于它所证明的任何属性，都可以保证百分之百的正确。



# 形式化验证的方向

- 定理证明
- 模型检验
- 等价性验证



# 定理证明

- 定理证明的典型做法是将系统及其性质用某种形式化逻辑公式表示，以相应的公理和推理规则为基础，逐步推导出表达系统性质的公式，从而证明目标设计具有该性质。一些工具可以辅助推理过程的进行，称为定理证明器。
- 由于使用定理证明器需要较高的数学能力并要经过相当长时间的训练，因而这种验证方法的代价较高。定理证明方法主要掌握在一些学术研究人员的手中，普通的验证工程师难于掌握。
- 作为学术研究的成功案例，已经有很多的定理证明系统在大型的设计中得以成功地运用，如浮点计算单元和复杂流水控制等。



# 模型检验

- 模型检验运用公式化的数学技巧来检验设计的功能属性。模型检验工具搜索一个设计所有可能的状态空间，寻找通过仿真很难发现的缺陷。当模型检验发现设计中的错误时，会给出一个反例，以帮助分析；若没有给出反例，则可以保证目标设计的该属性是百分之百正确的。
- 随着设计复杂度的提高，设计中的状态数目呈指数增长，模型检验对设计状态空间进行遍历的做法很容易遇到状态爆炸。随着一些化简技术和状态表示、操作方法的提出，模型检验方法的处理规模已经有了很大进步。
- 模型验证通常对控制密集型设计的验证比对数据通道密集型设计的验证更加有效。数据通道中大量的存储型单元造成了很大的状态空间，需要花费昂贵的存储空间和大量的处理时间。

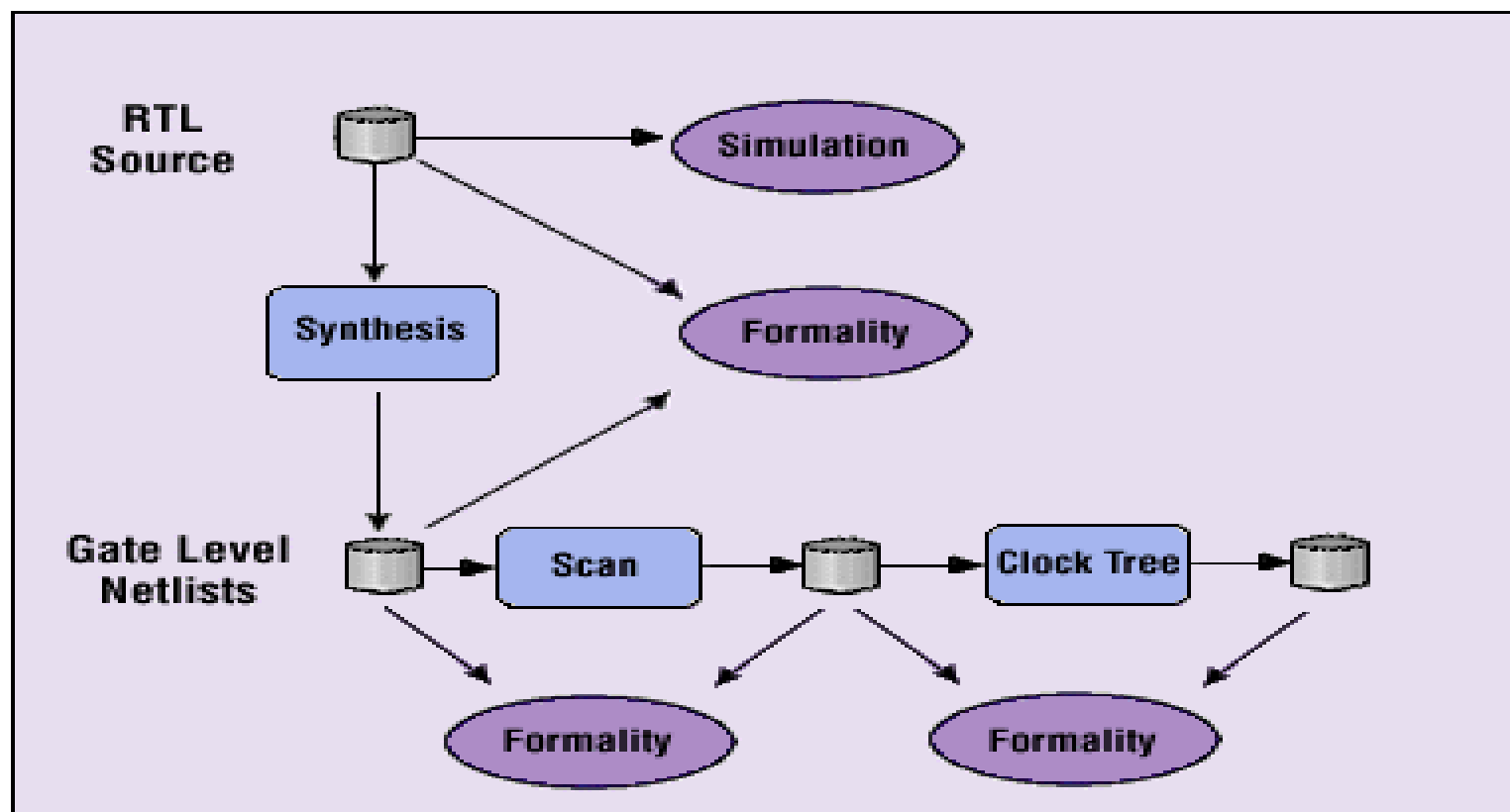


# 等价性验证

- 等价性检验验证一个设计的不同实现层次之间的等价性，如RTL实现与门级实现之间、插入时钟树前后的设计实现之间、扫描链重排前后的设计实现之间等。等价性检验的基本原理是建立两个被比较模型之间的关系。检验的依据是数学的定理和公理，以及设计实现所利用的标准单元库的精确描述。
- 等价性检验方法的自动化程度很高，不需要用户的干预，而且处理规模比模型检验要大许多，是最成熟的形式化验证方法。许多大的EDA公司都有等价性检验工具推出，如Cadence的Affirma和Synopsys的Formality等。
- 等价性检验的不足之处是不能发现两个设计模型中同时存在的错误。



# 形式化验证的位置





# 形式化验证面临的问题

- 形式化方法所倡导的利用严格的数学工具形式化地表示研究对象，并按照数学分析、推理方法进行证明的做法对验证工程师来说是非常难于接受和掌握的，因而也一定程度上制约了形式化验证方法的推广。
- 形式化的证明方法依赖于设计规范的形式化描述，而客观上设计规范来源于非形式化的用户要求。目前仍然缺少可以将设计规范完全形式化的语言，而且对复杂设计而言，将设计规范完全形式化的成本也是不能接受的，因而形式化方法无法独立完成功能验证。
- 目前的形式化验证工具仍然面临规模制约的问题，对于复杂设计而言，自动工具支持的形式化验证方法容易遇到状态空间爆炸等容量溢出问题。尽管近些年来一些商用的纯形式化工具被推出，但距离设计规模快速增长的需求，仍有相当距离。



# 验证方法

- 基于模拟的验证方法（Simulation）
- 形式化验证方法（Formal Verification）
- 静态验证方法（Static Verification）
- 物理验证（Physical Verification）
- 硬件仿真（Emulation）



# 静态验证方法

## ■ 语法检查

用于在设计初始阶段检查一些代码中的语法错误，是一种规则性检查。

## ■ 静态时序分析（**Static Timing Analysis, STA**）

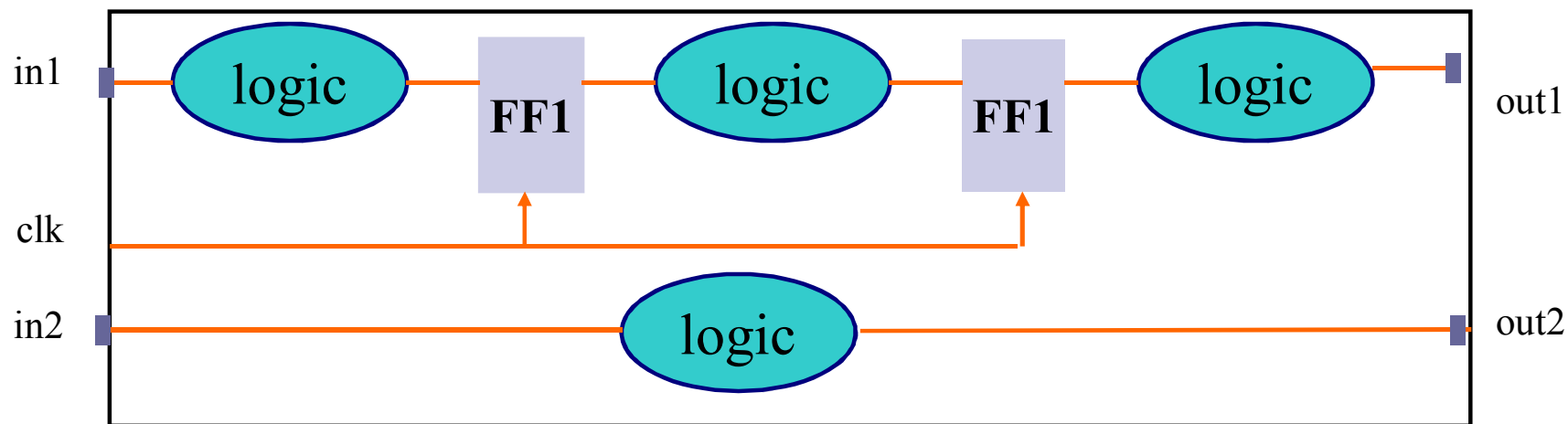
通过计算设计中所有存储型单元之间的路径延时是否在设计频率所允许的范围内来判断设计中是否有时序问题。



# 静态时序分析

- 目的：检查电路是否满足设计规定的时序要求；  
能否减少延迟，提高频率。
- 特点：不使用测试向量；  
不检查电路功能；  
只能用于同步电路。
- 工具：PrimeTime (Synopsys)  
Pearl (Cadence)

# 静态时序分析的约束



- 四种路径：
1. input port-->register
  2. register-->register
  3. register-->output port
  4. input port-->output port



# 验证方法

- 基于模拟的验证方法（Simulation）
- 形式化验证方法（Formal Verification）
- 静态验证方法（Static Verification）
- 物理验证（Physical Verification）
- 硬件仿真（Emulation）



# 物理验证 (1)

- 特点：对版图的检查。版图的正确性在于它首先要符合工艺规则，其次它必须与其对应电路结构保持一致性。
- 种类：
  - 几何设计规则检查（DRC）
  - 电学设计规则检查（ERC）
  - 网表一致性检查（LVS）





## 物理验证 (2)

**几何设计规则检查 (DRC, Design Rule Check)**，是以给定的设计规则为标准，对最小线宽，最小图形间距，最小接孔尺寸、栅和源漏区的最小交叠等工艺限制进行检查。

**电学规则检查 (ERC, Electronic Rule Check)** 介于几何规则检查和行为级分析之间，其主要作用是在功能和性能检查前对提取出的电路网表检测出没有电路意义的连接错误。

**网表一致性检查 (LVS, Layout Versus Schematic)** 是指通过网表提取工具对版图作电路连接复原，然后将提取出的电路网表与原理图得到的网表进行比较，检查两者是否一致。



# 验证方法

- 基于模拟的验证方法（Simulation）
- 形式化验证方法（Formal Verification）
- 静态验证方法（Static Verification）
- 物理验证（Physical Verification）
- 硬件仿真（Emulation）



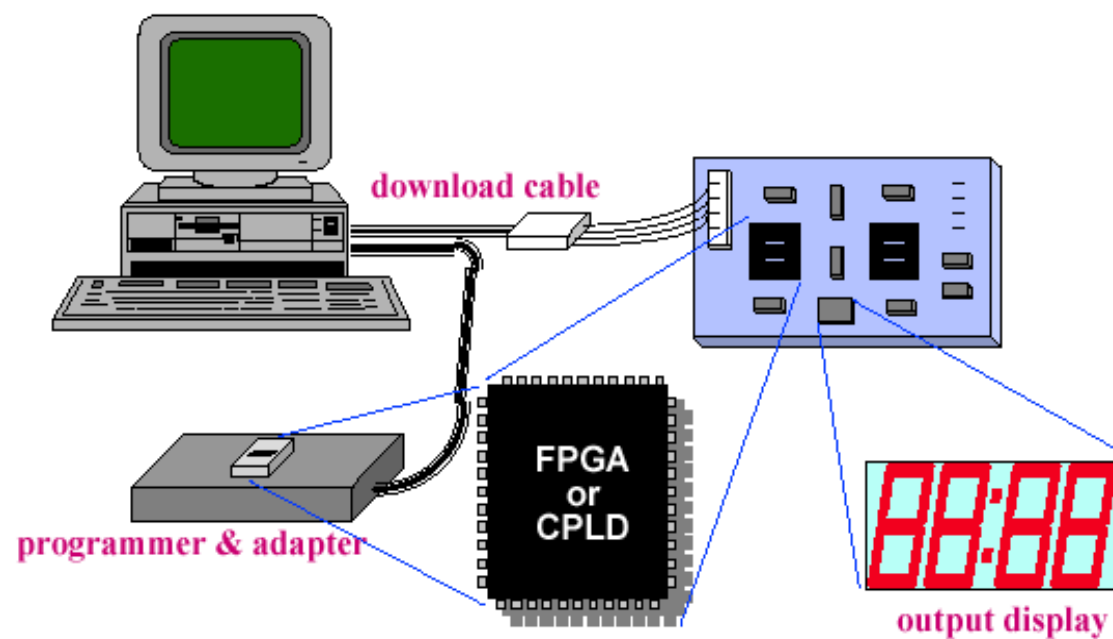
## 硬件仿真 (Emulation)

**特点：**将设计的样片或原型（可以用FPGA实现）放在硬件环境中进行功能的验证。

**优点：**速度非常快。

**缺点：**对设计要求高（可综合）  
难于调试

# 硬件仿真环境示意





## 小结：

- 仿真与验证的关系

- ☐ 仿真是验证，但验证不仅仅是仿真
- ☐ 仿真是验证的一种形式，如
  - RTL级仿真
  - 门级仿真
  - ...

- 按照技术特点，验证可有以下分类

- ☐ Simulation——系统仿真、RTL仿真，门级仿真、后仿真
- ☐ 形式化验证
- ☐ 静态验证 ——STA
- ☐ 物理验证 ——DRC, ERC, LVS
- ☐ Emulation——FPGA



# 示例1

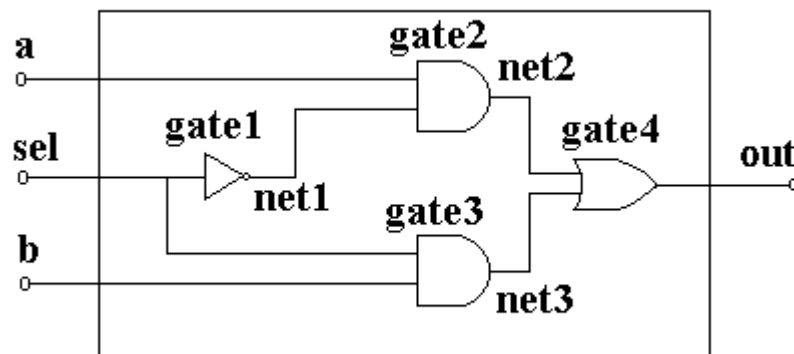
## ◆ 二选一 (MUX)

- 有两路输入，一路输出，外加一个控制信号；
- 通过控制信号选取其中一路输入作为输出；

## ◆ 具体的设计实现

- a,b 为输入信号，out为输出信号，sel为控制信号；
- sel为低电平时，选择a作为输出；反之则选择b作为输出。

# Design Under Verification



```
module mux_str(out, a, b, sel);  
  
    //端口说明  
    output out;  
    input a, b, sel;  
  
    assign out= (sel ==0) ? a : b;  
endmodule
```



# 测试计划

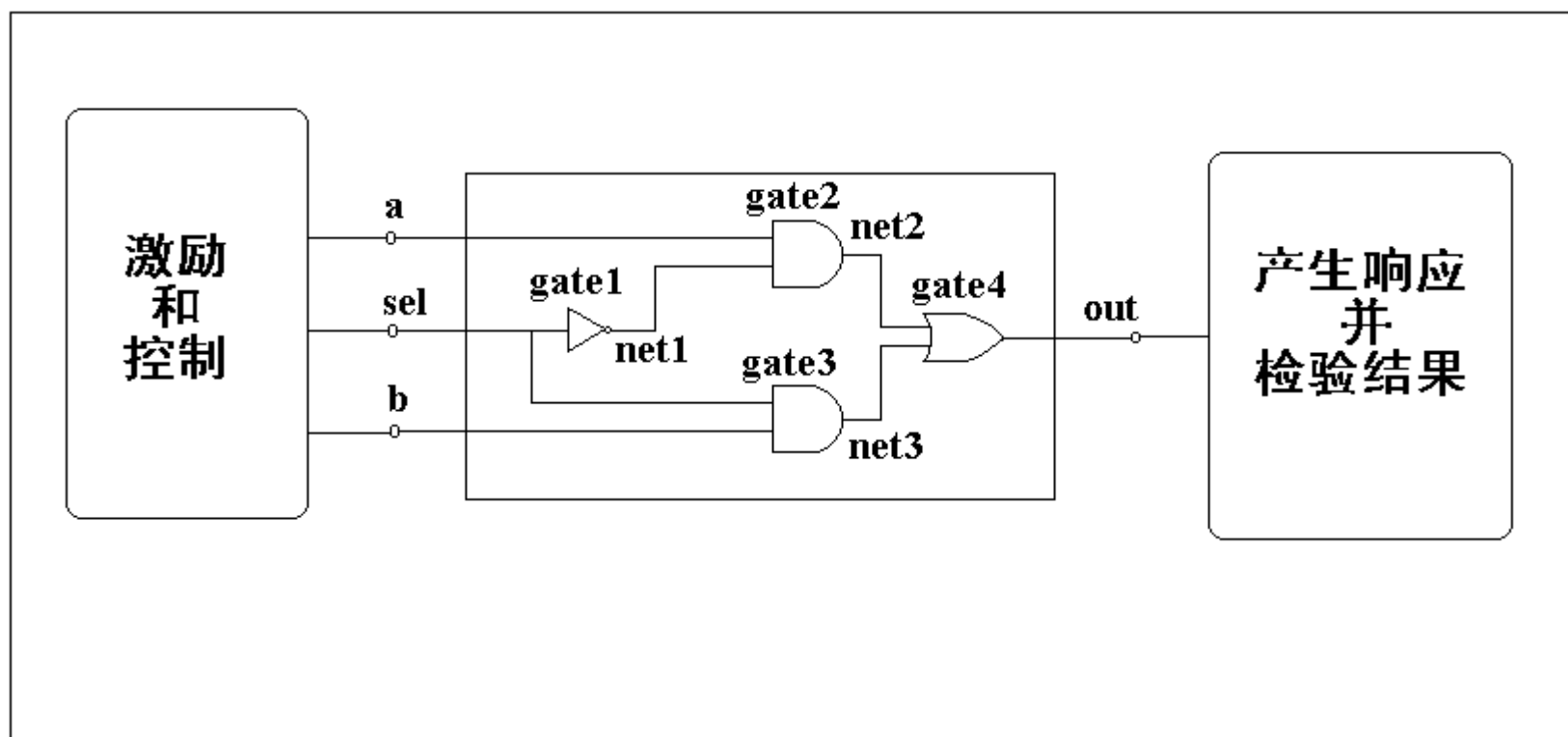
- sel为低电平时，选择a作为输出；
- sel为高电平时，则选择b作为输出；

## 具体过程

- 赋初值a=0;b=1;sel=0;观测out输出（应该为0）
- 置a=1; b=0;                      观测out输出（应该为1）
- 置sel=1;                          观测out输出（应该为0）
- 置b=1;a=0;                      观测out输出（应该为1）



# 测试环境





# 测试环境模板

```
module module_name();
```

```
    //数据类型声明
```

```
    //调用被测模块
```

```
    //施加激励
```


```
    //显示结果
```

```
endmodule
```

**请注意：**  
在测试程序中没有  
端口声明。



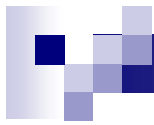
- initial 过程语句块
- always 过程语句块

	a   ways	c
	c	_____
	c	_____
	c	_____
	c	_____
	c	_____
	c	_____

# 激励的施加

```
module test_for_mux;  
//数据类型声明  
reg a, b, sel;  
//调用被测模块  
mux_str mux1(out, a, b, sel);  
//施加激励  
initial  
begin  
    a=0;b=1;sel=0;  
    #10 a=1;  
    #10 b=0;  
    #10 sel=1;  
    #10 b=1;  
    #10 a=0;  
    #100 $finish;  
end  
//显示结果  
endmodule
```

时间	取值		
	a	b	sel
0	0	1	0
10	1	1	0
20	1	0	0
30	1	0	1
50	1	1	1
150	0	1	1



# 响应的产生和观察

## ■两个系统函数

`$shm_open("waves.shm")`

打开与波形数据库的连接

`$shm_probe()`

选择信号，并将其数值的变化导入波形文件中

`initial`

`begin`

`$shm_open("mux_wave.shm");`

`$shm_probe("AS");`

`#100 $stop;` //在第100个时钟单位停止仿真

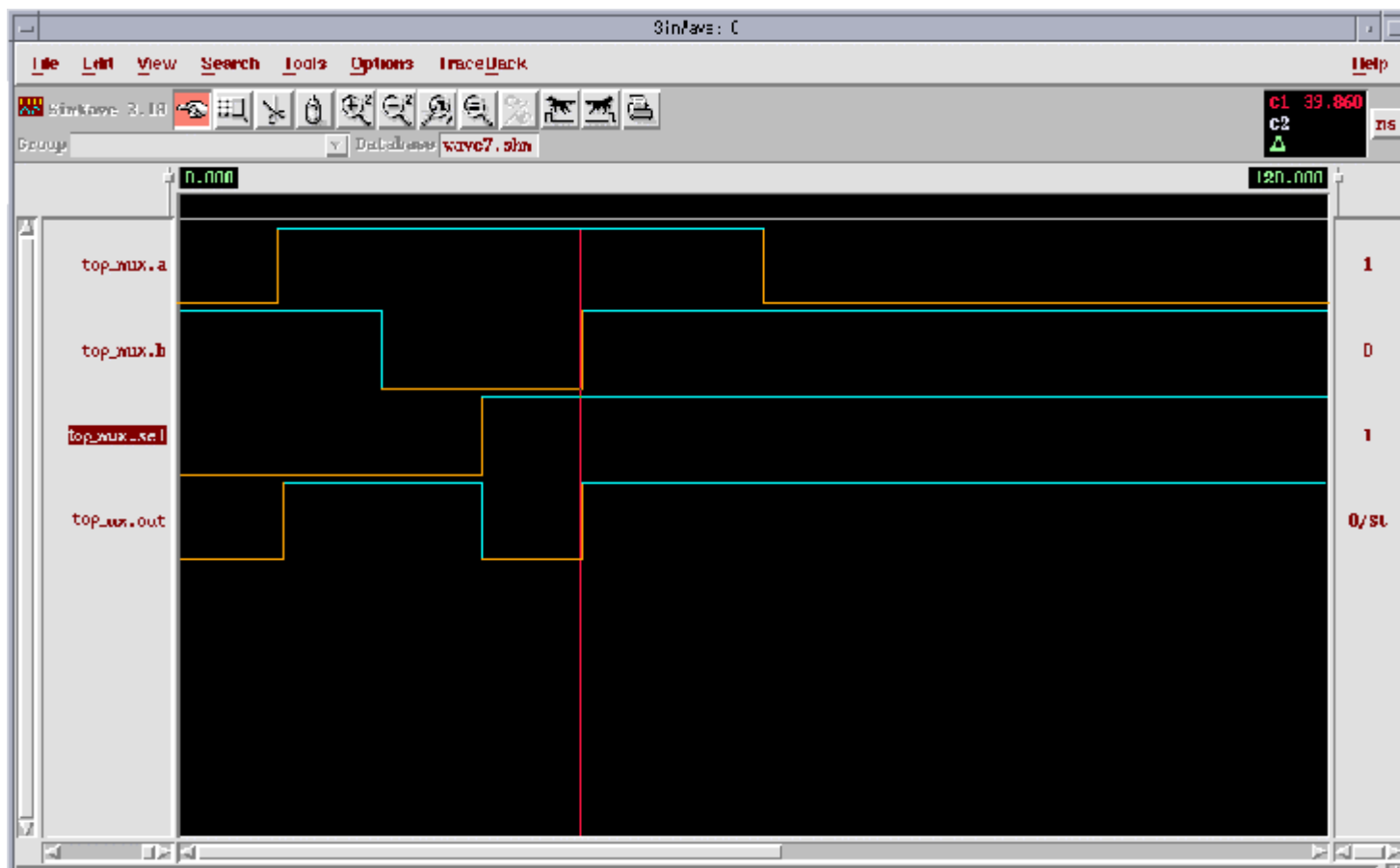
`end`



# 完整的测试文件

```
module test_for_mux;
reg a, b, sel ;
//调用被测模块
mux_str mux1(out , a, b, sel);
//施加激励
initial
begin
    a=0;b=1;sel=0;
    #10 a=1;
    #10 b=0;
    #10 sel=1;
    #10 b=1;
    #10 a=0;
//显示结果
    $shm_open("mux_wave. shm") ;
    $shm_probe("AS") ;//
    #100 $stop;
    #100 $finish ;
end
endmodule
```

# 观察波形



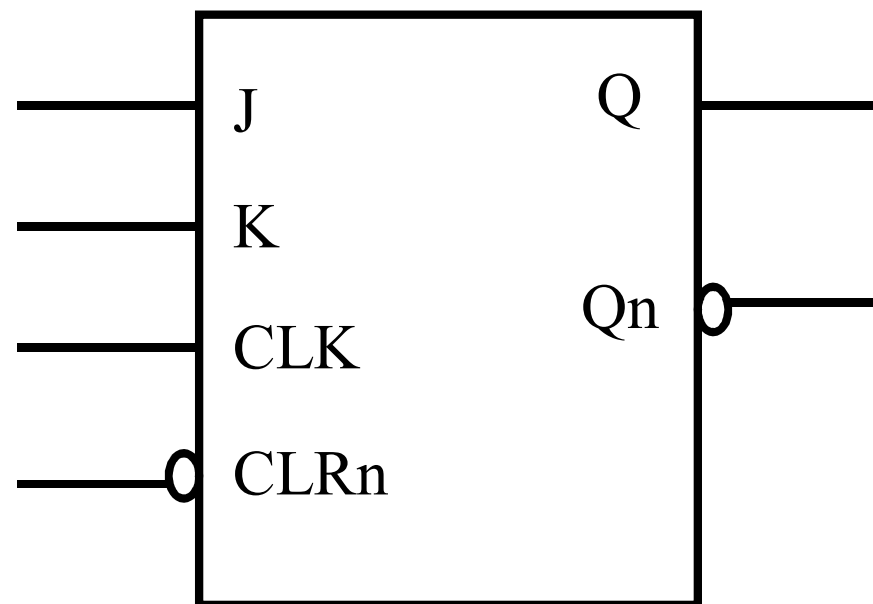
## 示例2

### ■ J k 触发器





# JK触发器外特性





# 真值表

输入				输出	
CLRn	CLK	J	K	Q	Qn
0	x	x	x	0	1
1	↓	0	0	不变	
1	↓	1	0	1	0
1	↓	0	1	0	1
1	↓	1	1	翻转	



# DUV——RTL代码

## 文件头

```
/*  
// MODULE:          jk flip-flop  
// FILE NAME:       jk_rtl.v  
// DATE:            1.0  
// AUTHOR:          xxx  
//  
// CODE TYPE:       Register Transfer Level  
//  
// DESCRIPTION:      This module defines a J-K flip-flop with an asynchronous,  
//                  active low reset.  
//  
*/
```

# DUV (续)

```
// DEFINES
`define DEL 1 // Clock-to-output
               // delay. Zero time delays can
               // be confusing and sometimes
               // cause problems.

// TOP MODULES
module jk_flipflop(
               clk,
               clr_n,
               j,
               k,
               q,
               q_n);

// INPUTS
input clk;      // clock
input clr_n;    // Active low, asynchronous reset
input j;        // J Input
input k;        // K input
```

```
// OUTPUTS
output q;
output q_n;

// signal Declarations
wire clk;
wire clr_n;
wire j;
wire k;
wire q;
wire q_n;

// PARAMETERS
// Define the J-K input combinations as parameters
parameter [1:0] HOLD    = 0,
                RESET    = 1,
                SET       = 2,
                TOGGLE    = 3;
```

# DUV (续)

```
// ASSIGN STATEMENTS
assign #`DEL q_n = ~q;

// MAIN CODE
// Look at the falling edge of clock for state transitions
always @(negedge clk or negedge clr_n)
begin
    if (~clr_n)
    begin
        // This is the reset condition. Most synthesis tools require an asynchronous reset to be defined
        // this way
        q <= #`DEL 1`b0;
    end
    else begin
        case ({j,k})
            RESET      q<= #`DEL 1`b0;
            SET         q<= #`DEL 1`b1;
            TOGGLE      q<= #`DEL ~q;
        endcase
    end
end
endmodule // jk_FlipFlop
```



# JK触发器的测试计划

- 需要测试的特征
  - J-K的四种组合
  - 复位功能是否正确

# 仿真代码的设计

- 利用变量cycle\_count 来计算时钟周期的数目
- 用case语句来检查每个时钟的输出是否正确，同时为下一个时钟周期的输入建立入口

例如

```
case (cycle_count)
4: begin
    // Test output
    if ((out==0)&&(out_n=1))
        $display ("Hold is working");
    else begin
        $display ("\n ERROR at time%0t:",$time)
        $display ("Hold is not working");
        $display (" out=%h, out_n=%h \n",out, out_n);
    // use stop for debugging
        $stop;
    end
    {j_in, k_in} = SET;
endcase
```



# 仿真代码 (1)

```

/*****/
// MODULE:          jk flip-flop simulation
// FILE NAME:       jk_sim.v
// DATE:            1.0
// AUTHOR:          xxx
//
// CODE TYPE:       Simulation
//
// DESCRIPTION:      This module provides stimuli for simulating a J-K flip-flop .
// It tests each combination of J and K inputs, and the asynchronous clear input.
// The hold function is tested twice to ensure that the flip-flop correctly holds both
// 0 and 1 states of the output.
/*****/
```





## 仿真代码 (2)

### **//DEFINES**

**`define DEL 1**

**//Clock-to-output delay. Zero time  
//delays can be confusing and  
//sometimes cause problems.**

### **//TOP MODULE**

**module jk\_sim();**

### **//INPUTS**

### **//OUTPUTS**

### **//INOUTS**

### **//SIGNAL DECLARATIONS**

**reg clock;**

**reg clear\_n;**

**reg j\_in ;**

**reg k\_in;**

**wire out;**

**wire out\_n;**

**integer cycle\_count; //Clock count variable**

## 仿真代码 (3)

```
//PARAMETERS
//Define the J-K input combinations as
parameters
parameter[1:0] HOLD =0,
                RESET =1,
                SET =2,
                TOGGLE =3;

//ASSIGN STATEMENTS
//MAIN CODE

//Instantiate the flip-flop
jk_flipflop jk(
.clk(clock),
.clr_n(clear_n),
.j(j_in),
.k(k_in),
.q(out),
.q_n(out_n));
```

```
//Initialize inputs
initial
begin
clock=0;
clear_n=1;
cycle_count=0;
end

//Generate the clock
always #100 clock =~clock;

//Simulate
always @(posedge clock)
begin
case (cycle_count)
0: begin
clear_n=0;
```



## 仿真代码 (4)

```
//Wait for the outputs to change
asynchronously
#`DEL
#`DEL
//Test outputs
if ((out===0)&&(out_n===1))
    $display ("Clear is working");
else
begin
$display("\nERROR at time %0t:",
    $time);
$display("Clear is not working");
$display( "out =%h, out_n =%h\n",
    out, out_n);

$stop; //Use $stop for debugging
end
end
```

```
1: begin
//Deassert the clear signal
clear_n=1;
{j_in, k_in}= TOGGLE;
end

2: begin
//Test outputs
if ((out===1)&& (out_n=== 0))
    $display ("Toggle is working");
else
begin
$display ("\nERROR at time %0t:", $time);
$display("Toggle is not working ");
$display("  out = %h, out_n =%h\n ", out,
    out_n);
//Use $stop for debugging
$stop;
end
{j_in ,k_in} =RESET;
end
```



## 仿真代码 (5)

```
3: begin
//Test outputs
if ((out === 0)&& (out_n ===1))
$display ("Reset is working");
else
begin
$display("\nERROR at time %0t:",
$time);
$display ("Reset is not working");
$display(" out= %h, out_n=%h\n",
out, out_n);
//Use $stop for debugging
$stop;
end
{j_in, k_in}= HOLD;
end
```

```
4: begin
//Test outputs
if ((out ===0) && (out_n===1))
$display ("Hold is working");
else
begin
$display("\nERROR at time %0t:",
$time);
$display("Hold is not working ");
$display( "out =%h, out_n=%h\n",
out, out_n);
//Use $stop for debugging
$stop;
end
{j_in , k_in}=SET;
end
```



## 仿真代码 (6)

```
5: begin
//Test outputs
  if ((out===1)&& (out_n=== 0))
    $display ("Set is working");
  else
    begin
      $display ("/nERROR at time
        %0t:", $time);
      $display("Set is not working ");
      $display("  out = %h, out_n
        =%h\n ", out, out_n);
      //Use $stop for debugging
      $stop;
    end
    {j_in ,k_in} =HOLD;
  end
```

```
6: begin
//Test outputs
  if ((out===1)&& (out_n=== 0))
    $display ("Hold is working");
  else
    begin
      $display ("/nERROR at time %0t:",
        $time);
      $display("Hold is not working ");
      $display("  out = %h, out_n =%h\n
        ", out, out_n);
      //Use $stop for debugging
      $stop;
    end
    {j_in ,k_in} =SET;
  end
```



## 仿真代码 (7)

```
7: begin
$display("/nSimulation complete - no errors \n");
$finish;
end
endcase

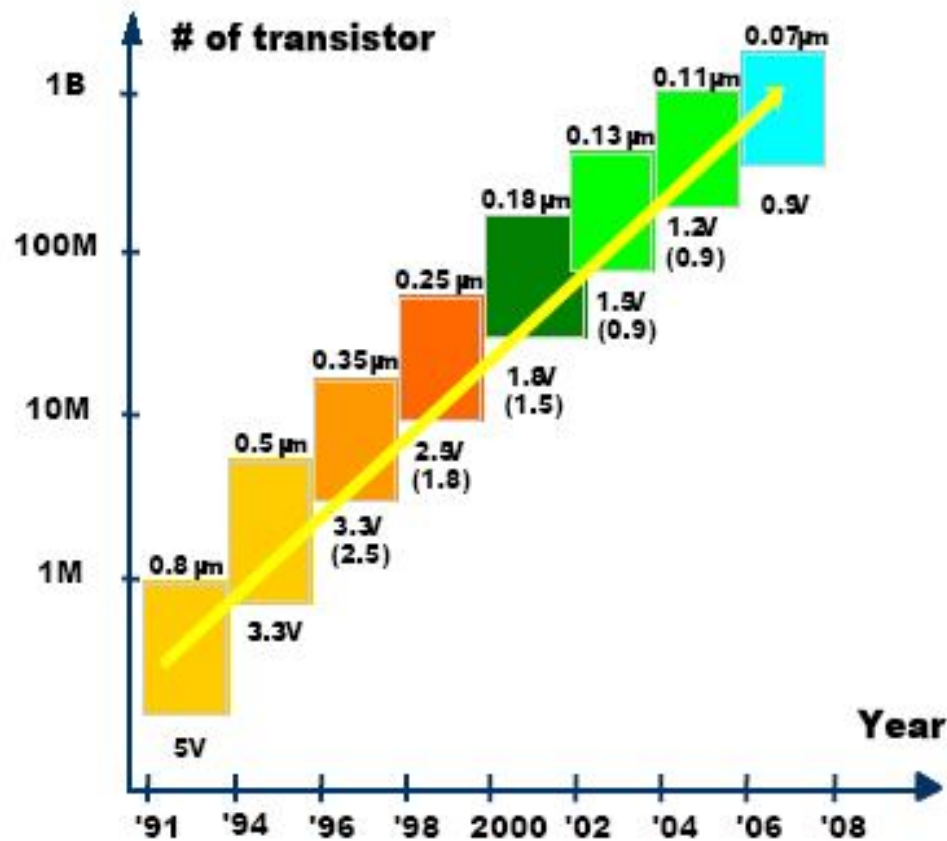
cycle_count = cycle_count + 1;
end
endmodule //jk_sim
```



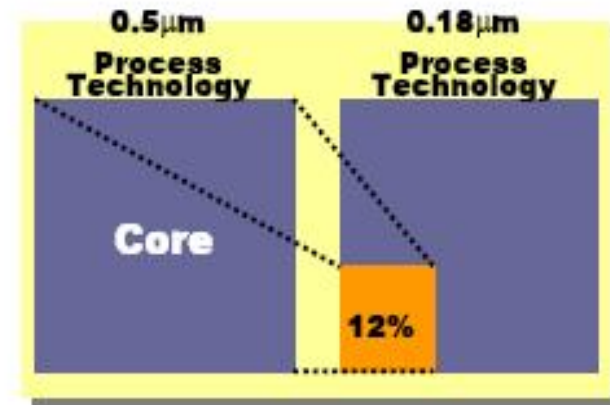
## 小结\_编码风格

- Comment
- Define & Parameter
- Message

# Trends of Process Technology

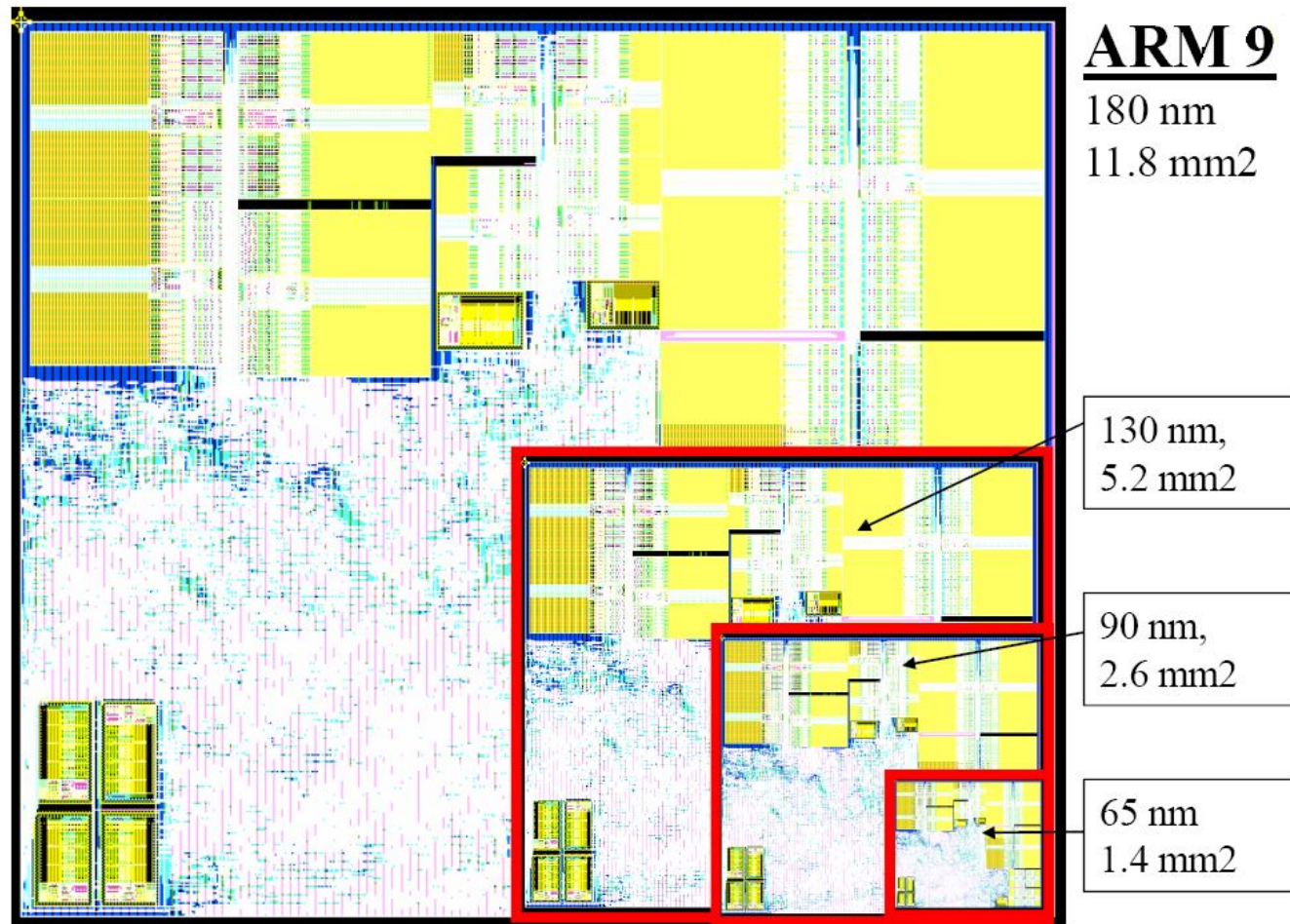


■ Moore's law





# Scaling ARM9

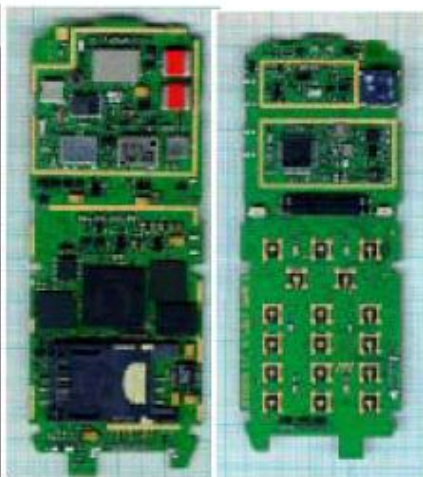




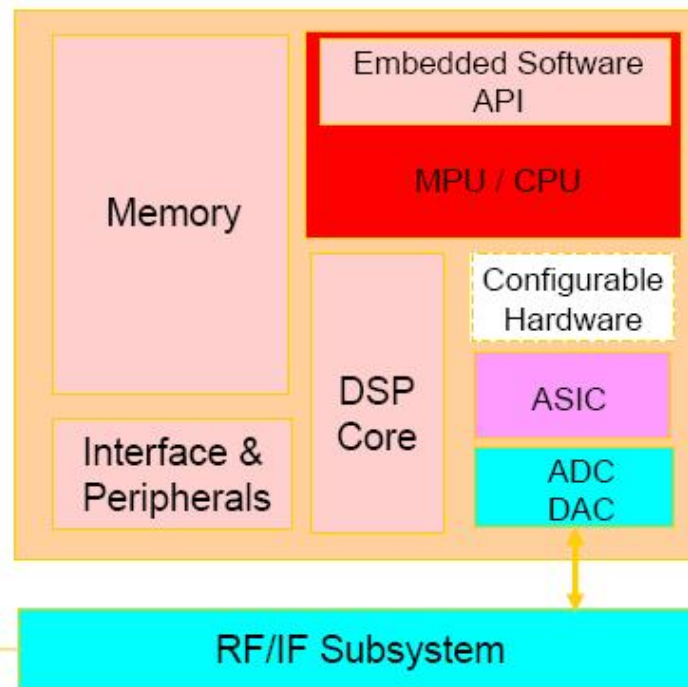
# Moore's Law drives the development of System-in-Chip Architectures

- Technological Advances
  - today's chip can contains 100M transistors .
  - transistor gate lengths are now in term of nano meters .
  - approximately every 18 months the number of transistors on a chip doubles – **Moore's law** .
- The Consequences
  - components connected on a Printed Circuit Board can now be integrated onto single chip .
  - hence the development of **System-On-Chip** design .

# Illustration



System-on-a-Board  
(PCB)



System-on-a-Chip  
(SoC)

# 微电子领域经历的三次重大变革

IC的速度很高、功耗很小，但由于PCB板中的连线延时、噪声、可靠性以及重量等因素的限制，已无法满足性能日益提高的整机系统的要求

分立元件

集成电路

在需求牵引和技术推动的双重作用下

系统芯片

System-on-a-Chip  
(简称SoC)

IC设计  
IC规划  
芯片

系统芯片(SOC)与集成电路(IC)的设计思想是不同的，它是微电子技术领域的一场革命。

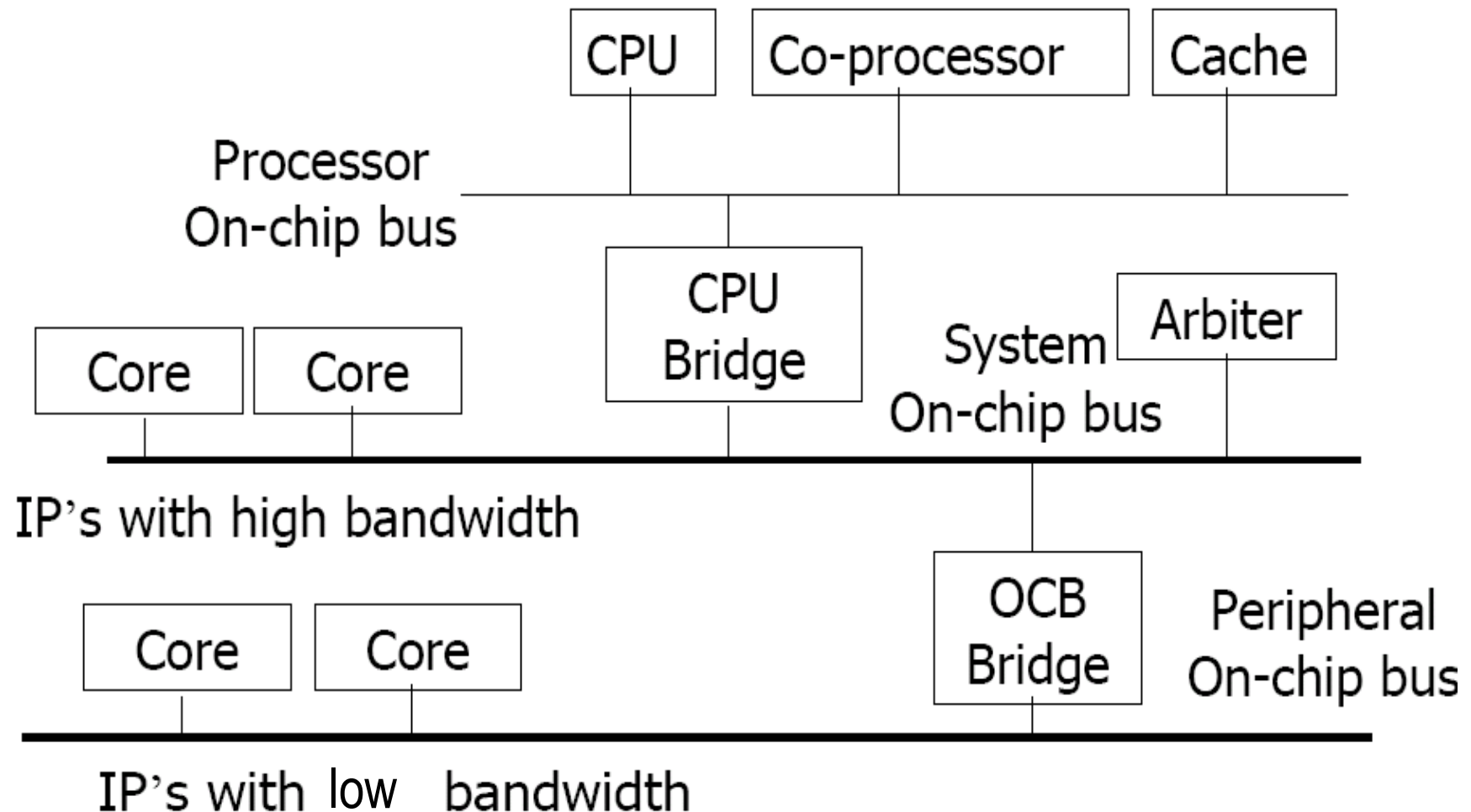
将整个系统集成在一个微电子芯片上



# What is SoC ?

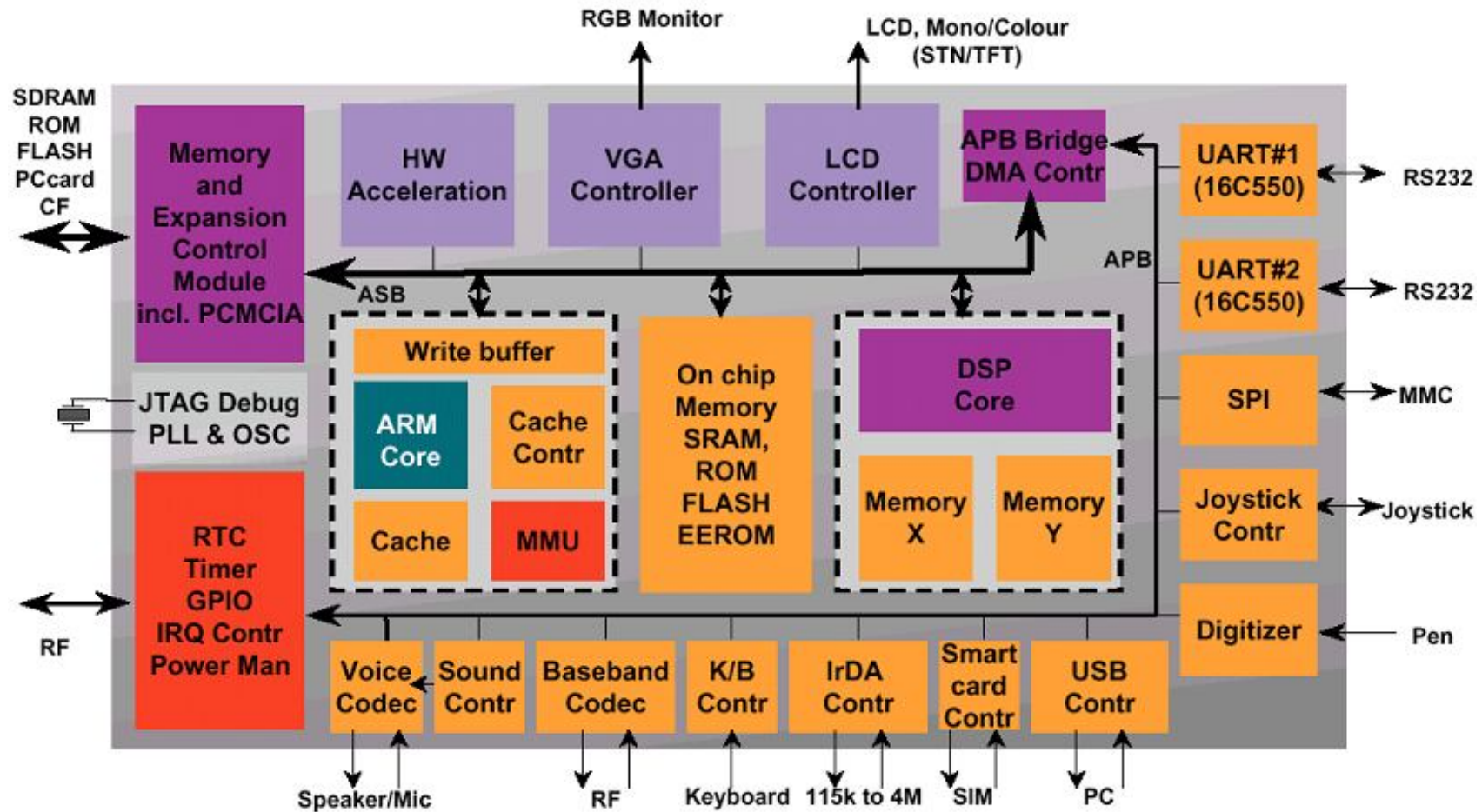
- 系统芯片(System-on-a-Chip, SoC)通常指在单一芯片上实现的计算系统。该系统应包含两个基本部分：**硬件部分**和**软件部分**。硬件部分包括 $\mu$ P, Bus, ROM/RAM, I/O Port等计算机系统的基本部件；软件部分主要指操作系统，也可以包括重要的应用软件。
- 系统芯片有时会包含模拟电路和混合电路，在有些应用场合中还会含有微传感器和微机械。

# A Typical SOC Architecture





# SoC Example





# Benefits of Using SOC

- Reduced size
- Reduced overall system cost
- Lower power consumption
- Faster circuit operation
- More reliable implementation
- Greater design security
- ...





# Challenges in SoC Era (1/2)

- Time-to-market
  - Process roadmap acceleration
  - Consumerization of electronic devices
- Silicon Complexity
  - Heterogeneous processes
  - Billion Transistors
  - Deep submicron effects : crosstalk, wire delays, electromigration
  - Mask costs




# Challenges in SoC Era (2/2)

- Design Complexity

- ☐  $\mu$ Cs, DSPs, HW/SW, SW protocol stacks, RTOS's, digital/analog IPs, On-chips buses
- ☐ System-level architecture
- ☐ Increased verification requirements


- Time-in-market

- ☐ Performance/Energy/Cost tradeoff
- ☐ Scalable architecture with unified design environment



# Challenges for CAD Tools in SoC Design (1/2)

- Designing at higher levels of abstraction
- Verification
  - Better and faster verification
- Timing & Power
  - Better physical design tools and tool integration, for instance 3D modeling
- Testing
  - Different testing schemes
- Capacity
  - To support high number of gate counts



# Challenges for CAD Tools in SoC Design (2/2)

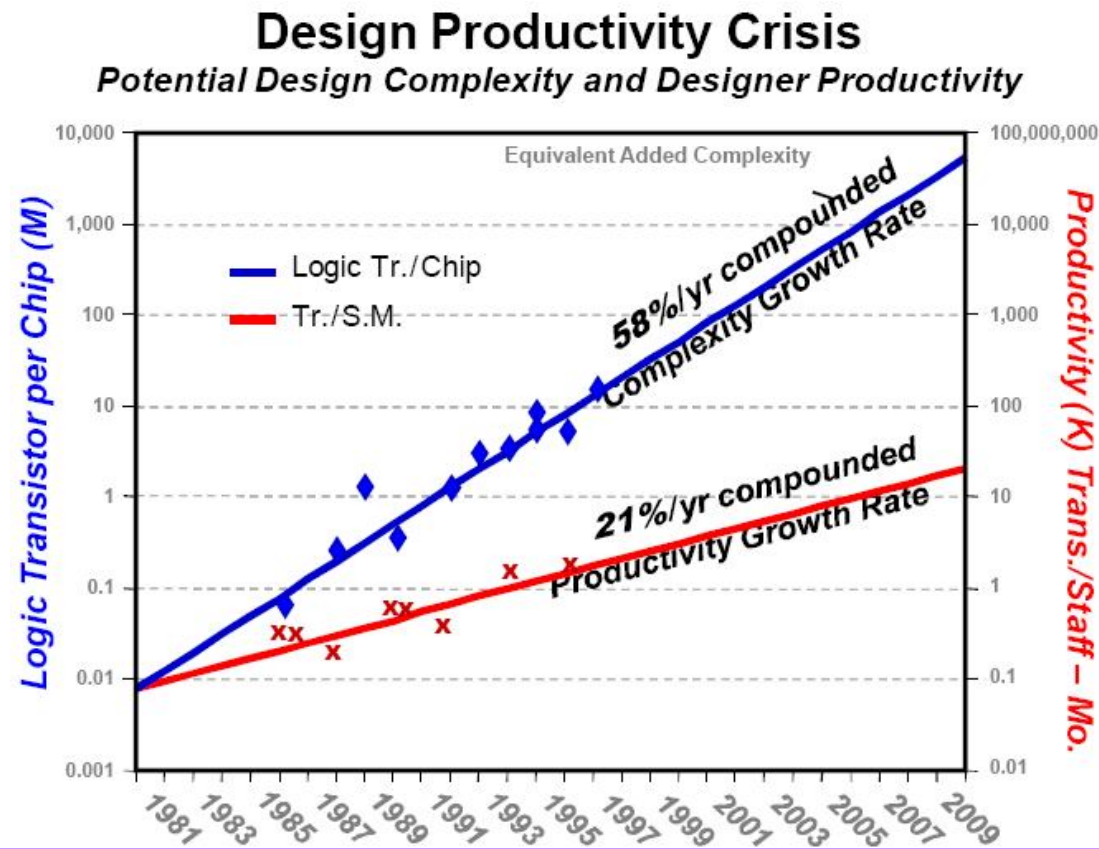
- IP Integration
  - To support use of commercial IP
- Hard IP Transition
  - Better physical design tool
- IP Standards
  - To facilitate use of IP from multiple sources
- IP security
  - To support various business model



# How to Conquer the Complexity?

- Use a known real entity
  - A pre-designed component (IP reuse)
  - A platform (architecture reuse)
- Partition
  - Based on functionality
  - Hardware and software partitioning
- Modeling
  - At different levels (system, architecture, circuit, and layout)
  - Consistent and accurate (formal verification)

# Growing Design-Productivity Gap



- Designs do not only get more complex, but also much more expensive!



# Solution is Design Re-use

- Overcome complexity and verification issues by designing **Intellectual Property** (IP) to be **re-usable** .
- Done on such a scale that a new industry has been developed.
- Design activity is split into two groups:
  - IP Authors – **producers** .
  - IP Integrators – **consumers** .
- IP Authors produce fully verified IP libraries
  - Thus making overall verification task more manageable
- IP Integrators select, evaluate, integrate IP from multiple vendors
  - IP integrated onto Integration Platform designed with specific application in mind



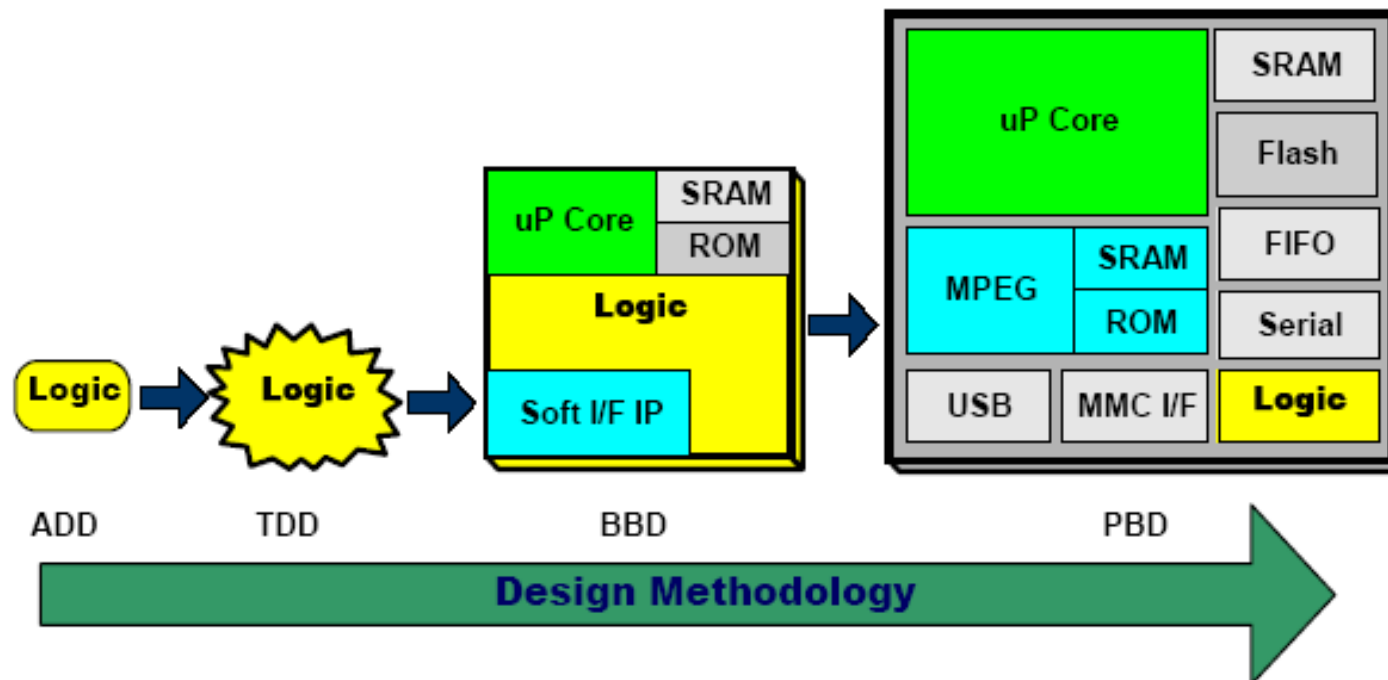
# IP

- A predefined, designed/verified, reusable building block for System-on-Chip
- Software IP, Silicon IP (Soft IP, Hard IP, ...)
- IP types
  - Foundation IP (cell library, gate array)
  - Standard IP (MPEG2/4, JPEG, USB, IEEE 1394, PCI...)
  - Star IP (ARM, MIPS, Rambus, ...)
- Ancillary characteristics
  - Deliverable at certain level, software/hardware interfaces
  - Modeling at different levels
  - Customizable, Configurable, Parameterizable



# Transition of Design Methodology


- From Area-Driven To Timing-Driven Design
- From Block-Based To Platform-Based Design





# Platform

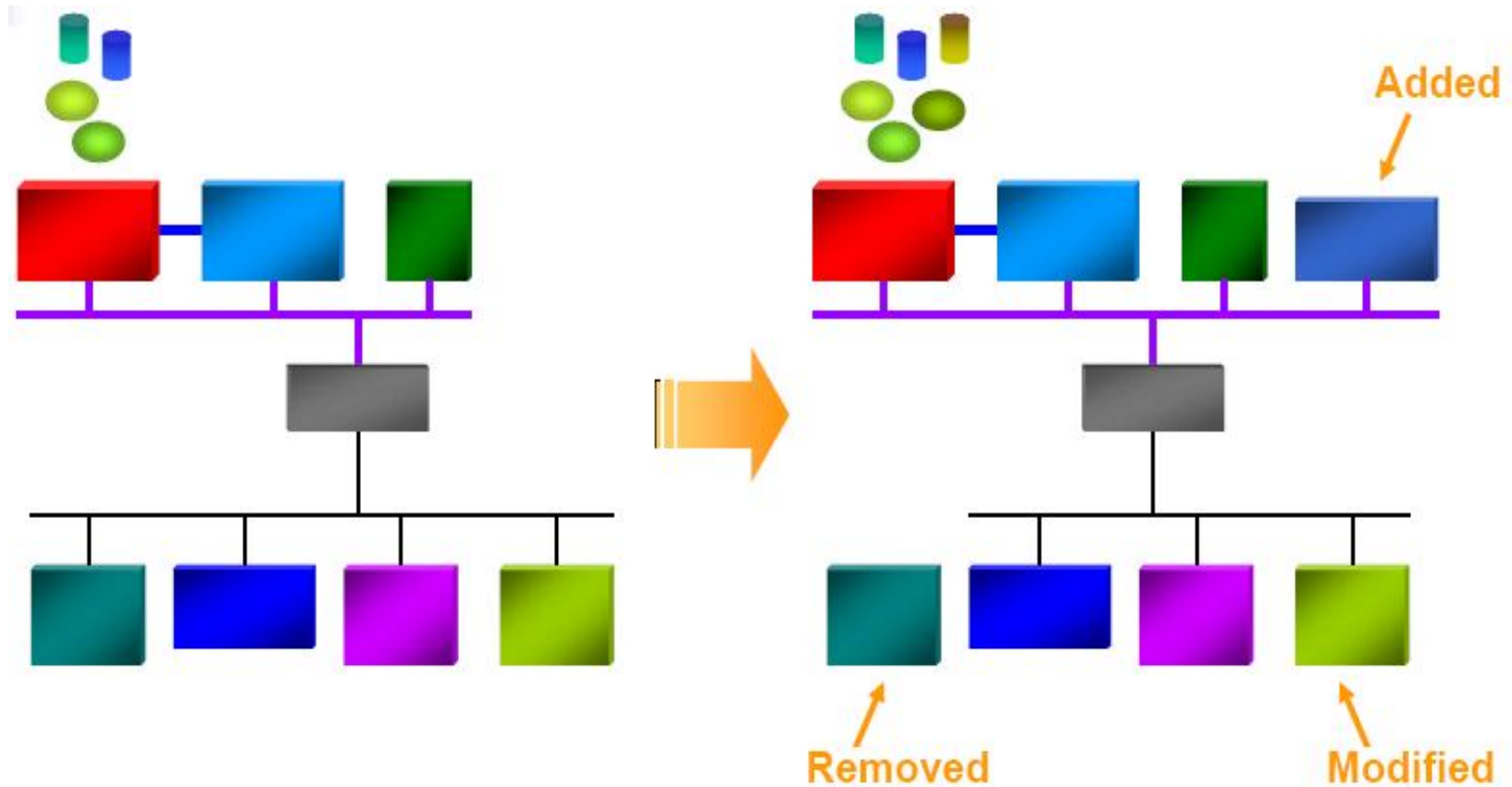
- A fully defined bus structure and a collection of IP blocks
- A design methodology to support the feature of “Plugging and Playing”
- The definition of a platform is the result of a trade-off process involving reusability (programmability and configurability), cost and performance optimization.
- Enhance the differentiation



# Derivative Design from an Integration Platform

- Change peripherals depending on the applications
- Add optional accelerating hardware
- Move hardware design to software, relying on new, faster embedded processors
- Limited IP blocks
- Significantly change software, tailoring a global product for particular markets or adding special user interface capabilities
- Derivative design is possible on integration platform

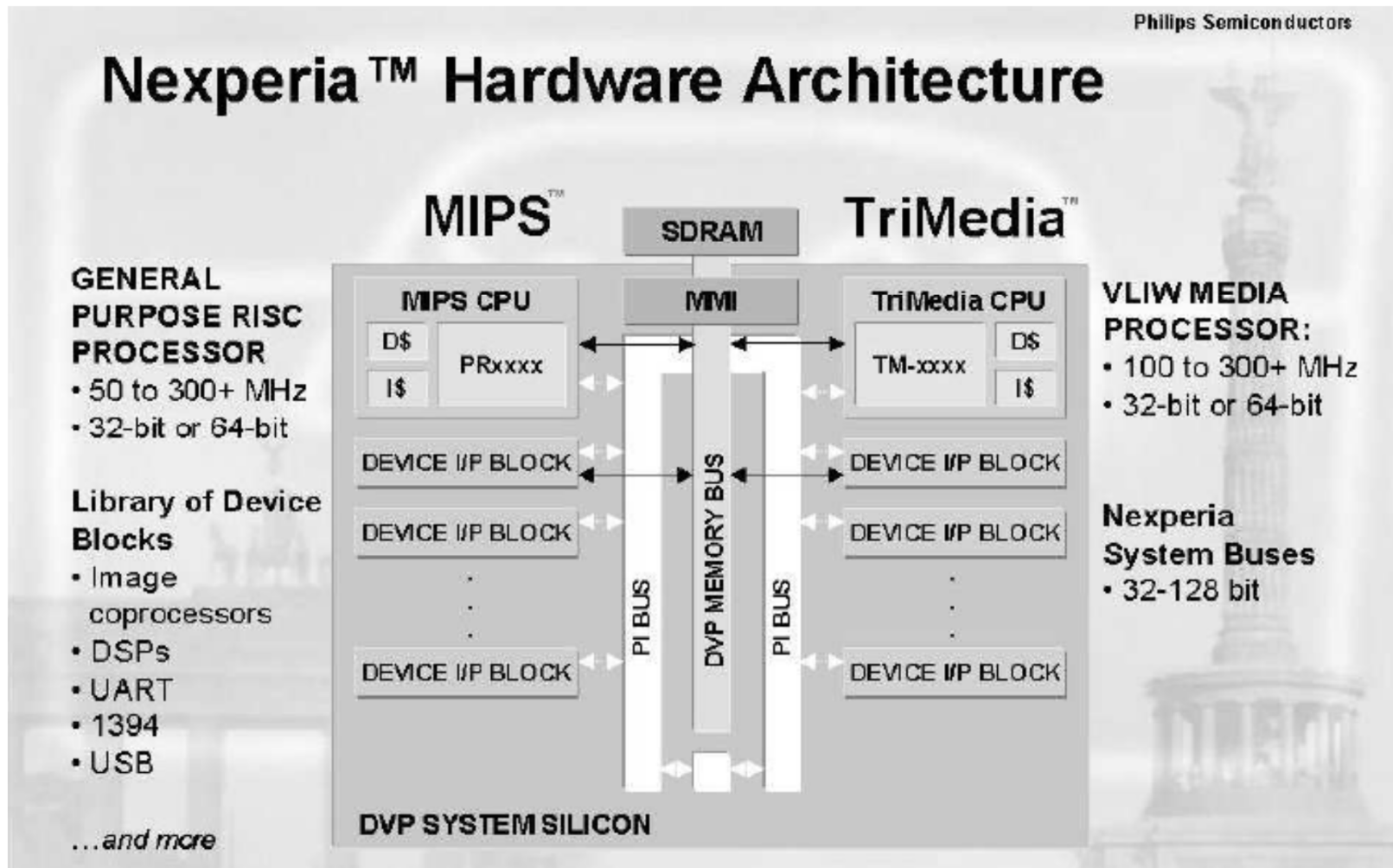
# Platform-Based Design, PBD

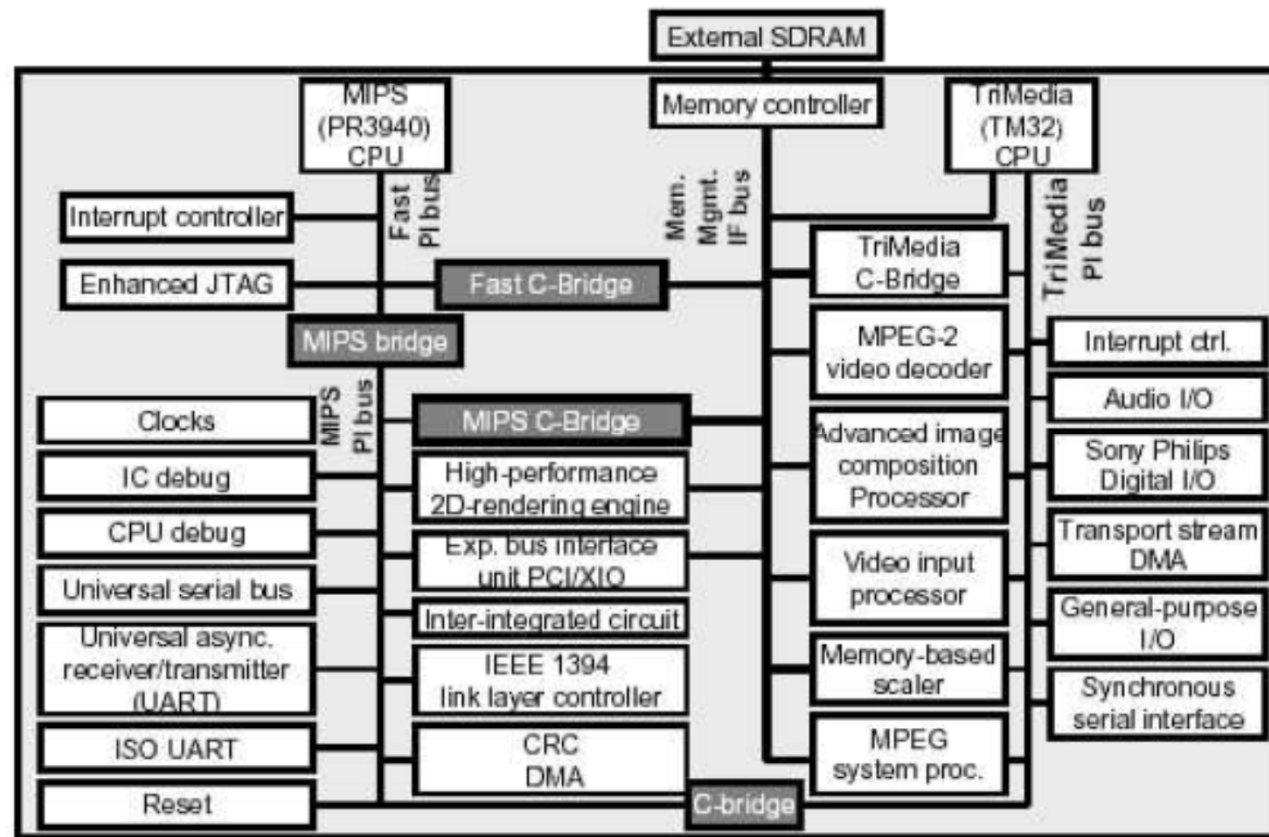


Reference design

Derivative design

# Platform Example: Nexperia







# Core Technologies

- System Architecture
- IP Development
- SoC Verification
- Embedded Software
- High Speed/Low Power Design

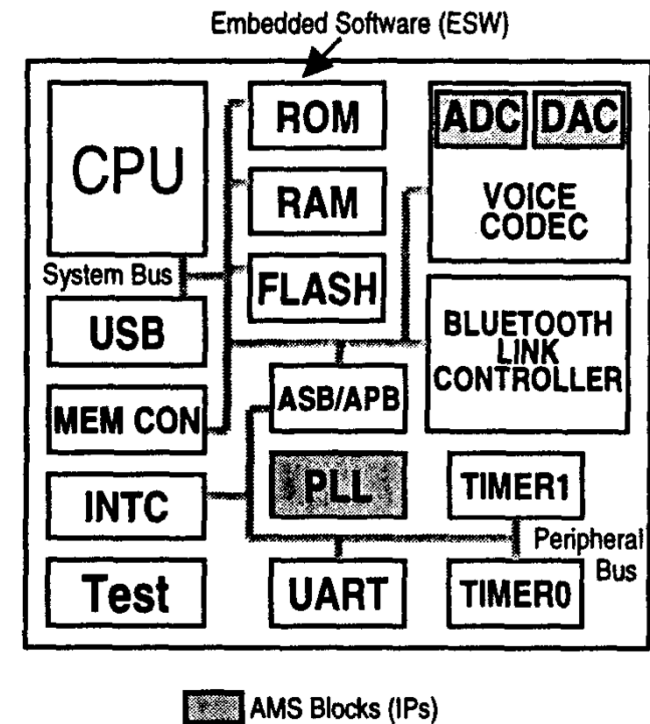
# System-on-Chip Verification Challenges

## ■ Verification goals

- functionality, timing, performance, power, physical issues in DSM

## ■ Design complexity

- MPUs, MCUs, DSPs, AMS IPs, ESW, clock/power distribution, test structures, interface, telecom, multimedia





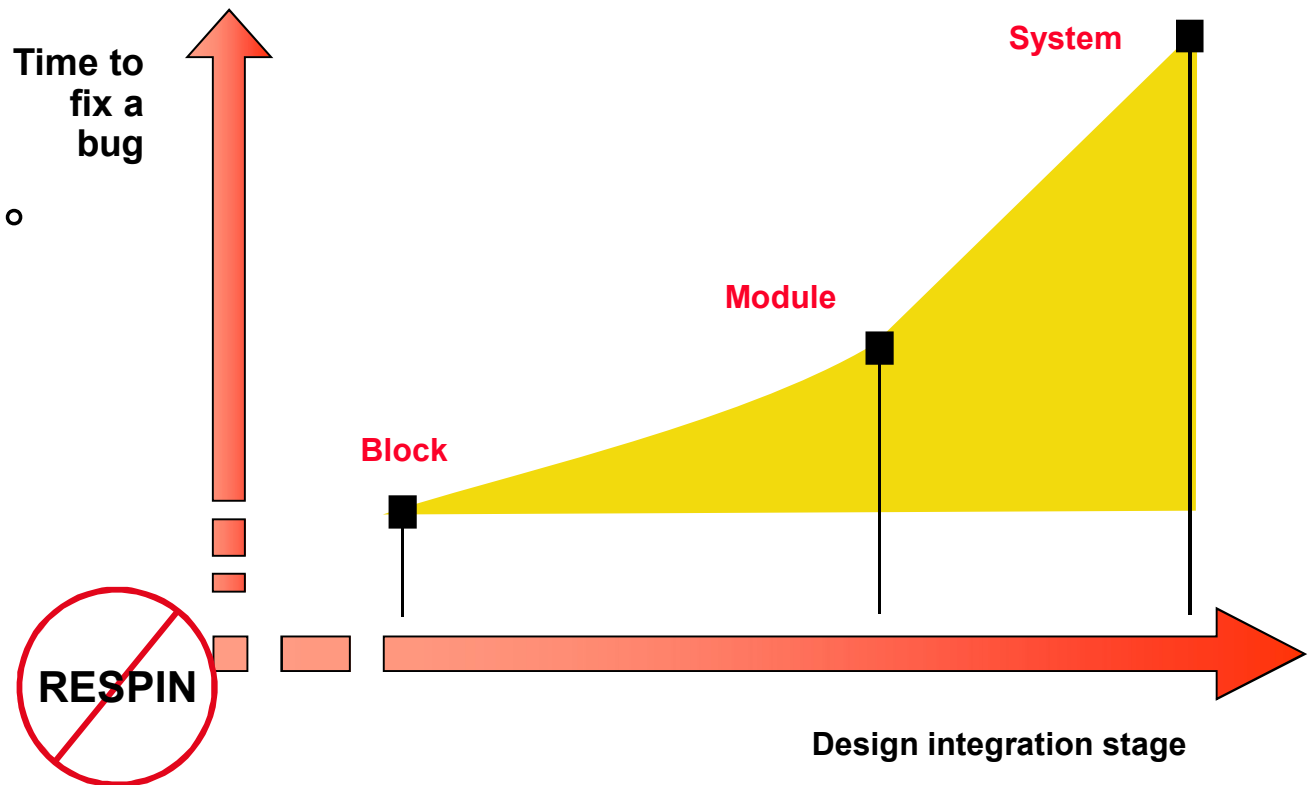


# System-on-Chip Verification Challenges

- Diversity of blocks (IPs/Cores)
  - different vendors
  - soft, firm, hard
  - digital, analog, synchronous, asynchronous
  - different modeling and description languages
    - C, Verilog, VHDL
  - software, firmware, hardware
- Different phases in system design flow
  - specification validation, algorithmic, architectural, hw/sw, full timing, prototype

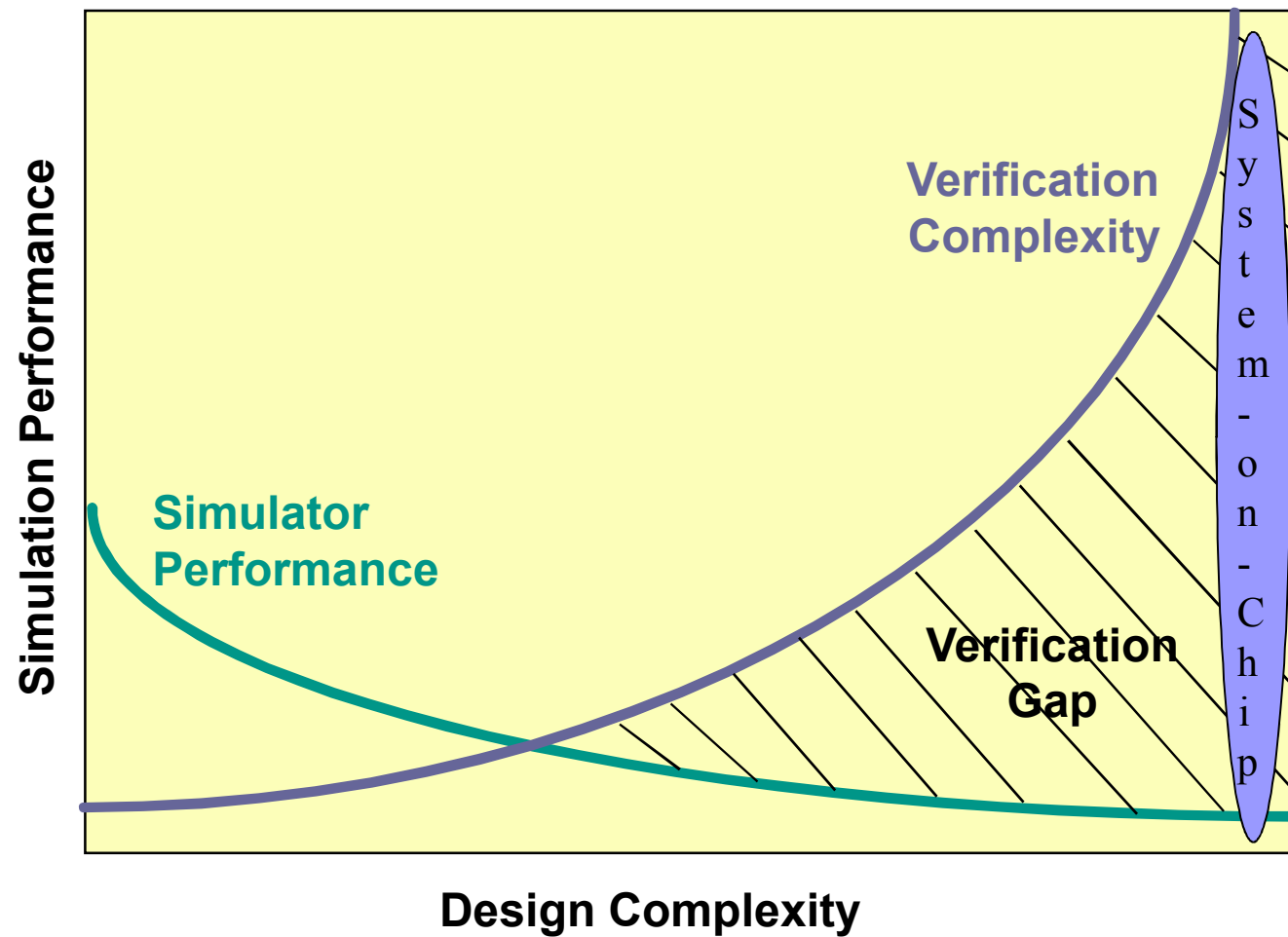
# Finding/fixing bugs costs in the verification process

设计规模的扩大使得工程师花费更多的时间来进行验证。虽然工程师们花了大量的时间和精力对产品进行验证，但是根据美国Collett公司的调查显示，集成电路设计的一次通过率还是从之前的50%下降到了35%。



- Increase in chip NREs make respins an unaffordable proposition
  - Average ASIC NRE ~\$122,000
  - SoC NREs range from \$300,000 to \$1,000,000
- NRE=non-recurring engineering

# SoC Design/Verification Gap



Source: Cadence



# IP verification strategy

- Three major phases
  - Subblock verification
    - Through and exhaustive **functionality** verification
    - Simulation, code coverage, test bench (TB) automation
  - Macro verification
    - **Interface** verification between subblocks
    - Simulation, hardware accelerator, TB automation, code coverage
  - Prototyping
    - Real prototype runs real software in the real application
    - FPGA, emulation, test chip
- Bottom-up approach
  - Locality
  - Catching bugs is **easier** and **faster** in the lower level



# Types of verification tests

- Compliance testing
- Corner case testing
  - Scenarios are created by designers
- Random testing
  - Create scenarios that engineers do not anticipate
  - Constrained random pattern generation
- Real code testing
  - Avoid misunderstanding the specification



# Testbench design

- Definition of testbench
  - A verification environment containing a set of components such as bus functional models (BFMs), bus monitors, memory models – and the interconnect of such components with the design-under-test
- Auto or semi-auto stimulus generation is highly preferred
- Automatic response checking is a must



# Code coverage

- Indicate how much of design has been exercised
- Point out what areas need additional verification
- Quantitative stopping criterion
- Types
  - statement      - branch
  - condition      - path
  - toggle      - FSM

100% coverage in statement, branch, condition



# Timing Verification

- STA is the fastest and most complete approach for timing verification of synchronous designs
  - Avoiding any timing exceptions is extremely important
- Gate-level simulation
  - Most useful in verifying timing of asynchronous logic, multi-cycle paths and false paths
  - Much slower run-time performance





# Level of confidence

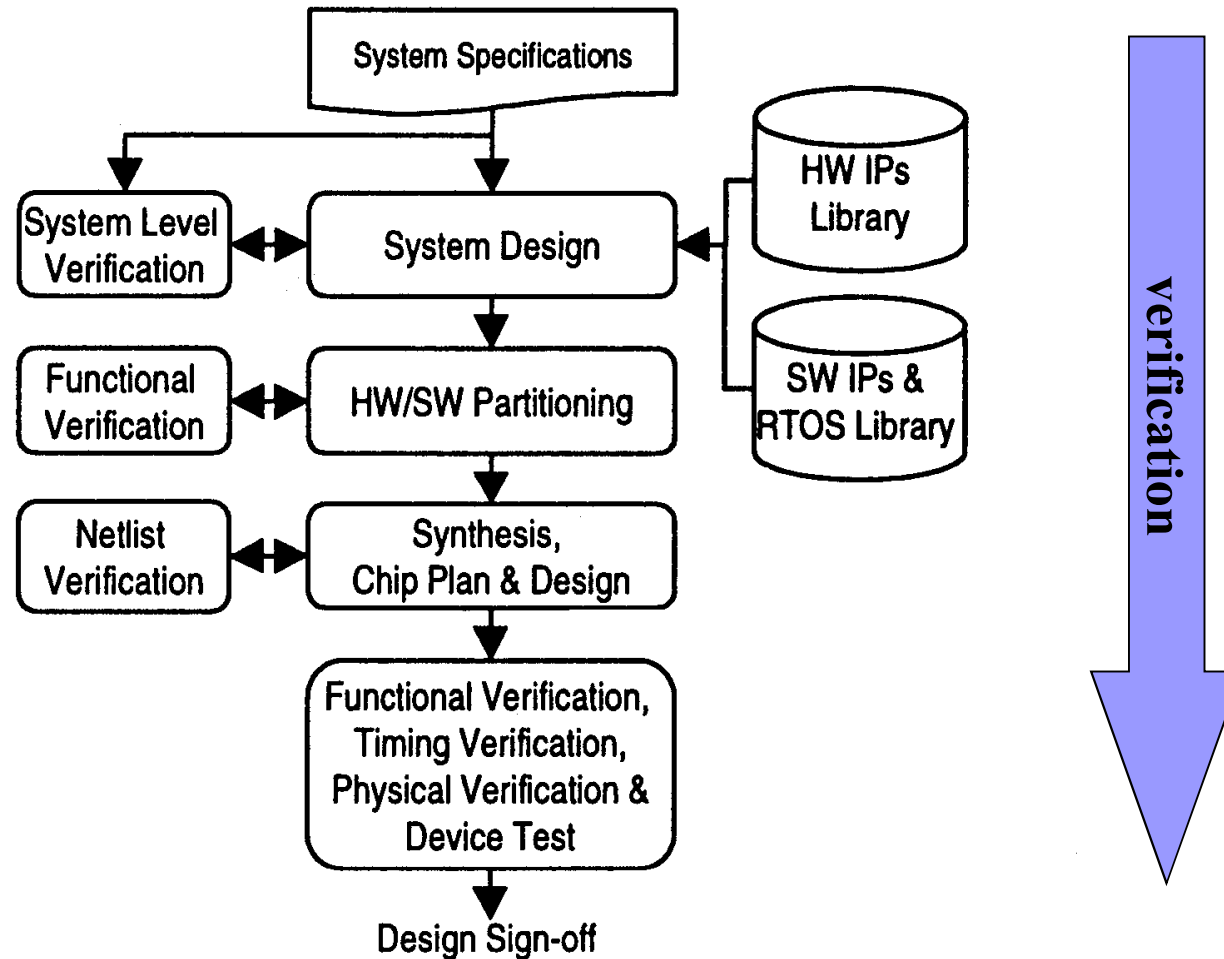
- Simulation pass
- High code coverage
- Thorough functional verification
  - Both deterministic and random
- FPGA prototyping
  - Simple demo system
- Chip prototyping
  - Sophisticated demo system



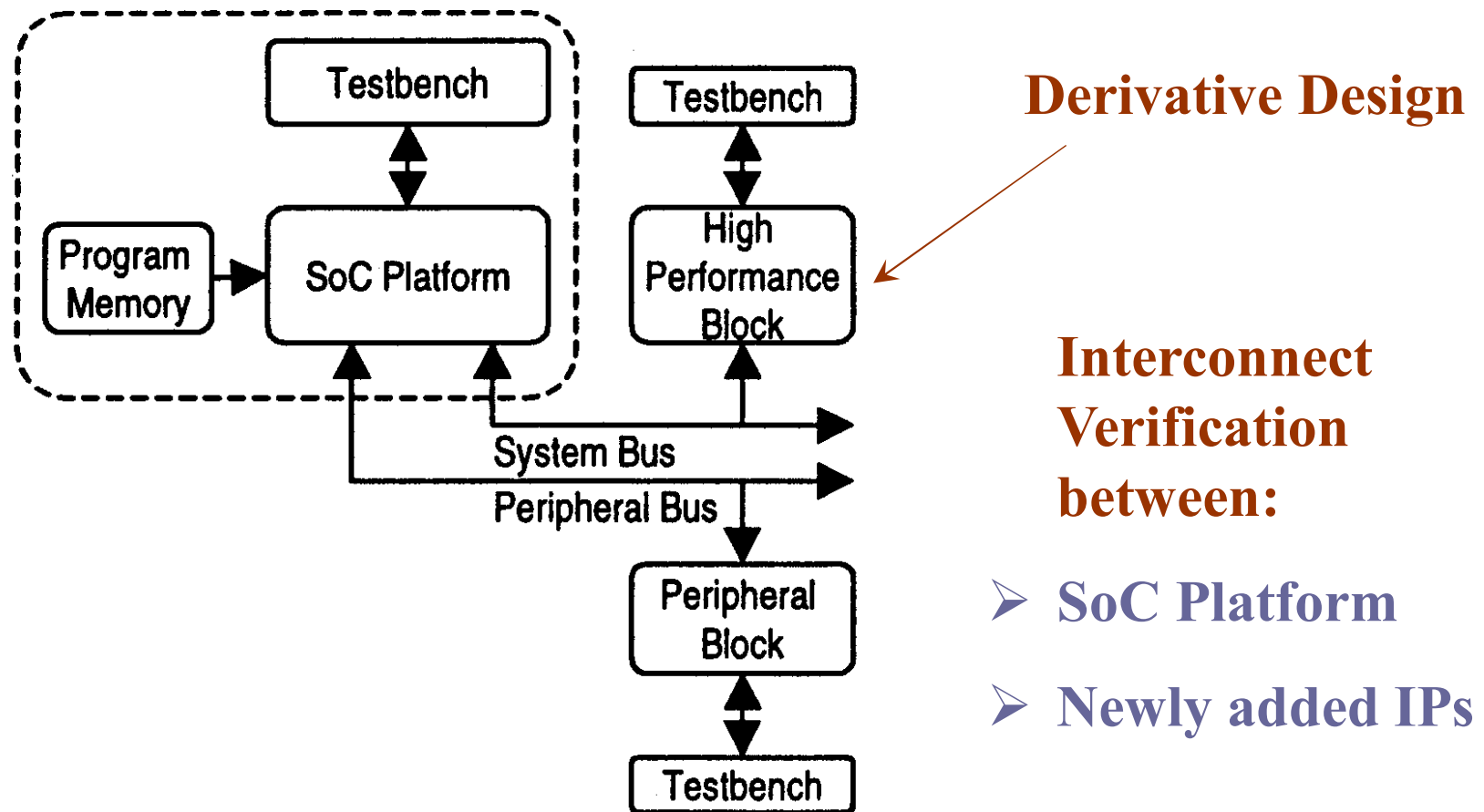
# SoC Verification Approaches

- Top-Down Verification
- Platform-Based Verification
- System Interface-Driven Verification

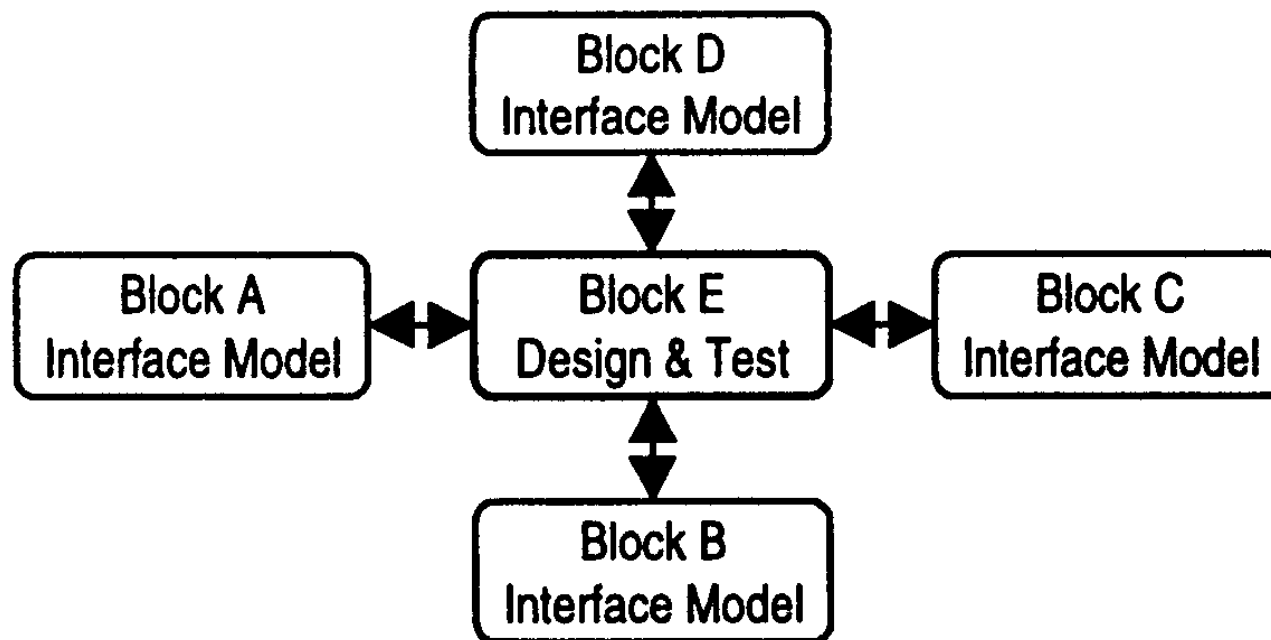
# Top-Down SoC Verification



# Platform Based SoC Verification



# System Interface-driven SoC Verification



**Besides Design-Under-Test,  
all others are interface models**



# Conclusions & Future works

- Verification productivity level increases lags all other aspects of design!
  - New **Techniques** and **methodology** are required for SoC Verification.
- Formal verification of SoC is definitely **required**! But, it should be used in **conjunction** with other verification techniques.
  - **Capacity** of formal verification must be enlarged for its wide-spread adoption
  - How do we **integrate** formal verification with conventional simulation and testing?
- How to come up with a **verification framework** which involves different verification methods cooperating in a single environment?
  - What should be the data structure?
- Design for Verifiability (**DFV**) -- is it viable and practical?