# 概述

随着集成电路制造技术的发展，电子系统变得越来越复杂，人们已经可以把复杂的电子系统集成到一个芯片上，就是所谓的片上系统<soc>。片上系统集成的东西越来越多，越来越复杂，这样复杂的一个系统对设计自动化的要求越来越高。

为了提高soc时代集成电路的设计效率，人们提出了基于IP复用和基于高层次综合的设计方法。IP复用技术，人们希望像搭积木一样将一个复杂的集成电路构建起来；基于高层次综合的设计，首先对电路进行比寄存器传输级<RTL>更抽象的描述，然后利用高层次综合工具完成到RTL电路的转换。这两中方法有各自的长处，也有各自的缺点。

无论采用什么样的设计方法学，人们都需要对soc时代的复杂电子系统进行描述，以选择合适的系统架构，进行软硬件划分、算法仿真等等。描述的级别越低，细节问题就越突出，对实际系统的仿真就越精准，完成建模所消耗的时间、仿真和验证时间就越长。相反，描述的抽象级别越高，完成建模所消耗的时间就越短，但对目标系统的描述也就越不精确。计算机的运行速度是有限的，设计人员必须在速度和精确性之间做出折中。

随着系统越来越复杂和相比较之下市场时间越来越紧迫，人们迫切需要一种语言单一完成全部设计。这种语言必须能够用于描述各种不同的抽象级别，能够胜任软硬件的协同设计和验证，并且仿真速度快。总之，人们对系统级描述语言的要求是：高仿真速度和建模效率、时序和行为可分开建模、支持基于接口的设计、支持软硬件混合建模、支持从系统级到门级的无缝过度、支持系统级调试和系统性能分析等。

System verilog、superlog 和 systemc；二者各有优点，systemc开放，得到了较广泛支持和发展。

# systemc基本语法

## 模块

模块化的管理允许设计者把复杂的电子系统划分成更易管理的模块。每个模块独立进行设计和验证，然后像搭积木一样搭成更大的模块。模块能够帮助在一个有许多不同设计者的设计组里分解复杂的设计。模块允许隐藏内部设计细节，这就使设计者必须在各个模块中使用公共的接口，使整个系统容易修改和维护，从而有利于设计复用。

一个模块可以包含一些其他的systemc基本元素如端口、内部信号、内部数据、子模块、进程、构造函数和析构函数等。这写元素共同定义模块所表达的功能。

### 模块定义

有三种：

SC\_MODULE(MMU)

{

}

class mmu : public sc\_module 该声明所有成员都是私有的

{

}

struct mmu : public sc\_module 该声明所有成员都是公有的

{

}

### 模块端口

模块之间通过端口链接。端口分为in、out和inout三种类型。可以指定端口的数据类型、允许的数据类型包括c++基本数据类型，如bool、int、short、char等；或者是systemc专有数据类型，如：sc\_int、sc\_unit、sc\_logic等；或者是用户定义类型中任何的数据类型。例如：

SC\_MODULE(fifo)

{

sc\_in\_clk clk;

sc\_in<bool> rst;

sc\_in<bool> wr\_en;

sc\_in<bool> rd\_en;

sc\_out<bool> full, almost\_full;

sc\_out<bool> empty,almost\_empty;

sc\_out<unsigned short> data\_count;

sc\_out<int> dout;

}

systemc的类库中预先定义的端口包括sc\_in, sc\_out, sc\_inout。同时systemc允许通过对基本端口类型sc\_port<IF, N>扩展生成更复杂的端口。

systemc的端口定义了与特定端口类型相关的方法，如read()和write()都是基本的端口类型sc\_in, sc\_out和sc\_inout可以使用的方法。下面都是允许的：

dout=din;

dout=din.read();

dout.write(din);

dout.write(din.read()); 推荐使用read()和write()方法读写，可减少类型不匹配带来的错误。

### 模块信号

一个顶层模块可能由几个字模块组成，这些子模块需要通过信号相互链接。信号不能用in, out或inout来声明，信号的传输方向取决于链接部分的端口状态。verilog中使用wire来定义信号。在systemc中使用sc\_signal<T>来定义信号。

### 位置关联

位置关联中，所有端口都是按照声明的顺序位置进行一一对应的。例如：

sc\_signal<int> a, b, result;

sc\_in\_clk clk;

alu\_p = new ALU(“ALU”);

(\*alu\_p)(clk, a, b, result); → 按照位置一一对应；

在位置关联中，已经初始化的模块的关联里每一个信号都有一个匹配的端口。关联表中的第一个信号对应第一个端口，第二个信号对应第二个端口，以此类推。在这种类型的关联中，端口的声明顺序是非常重要的，如果没有完全按照顺序，就会把一个信号错误地关联到不该关联的端口上，从而导致关联错误。

位置关联在一个只有少量端口的模块的初始化是非常适用的，它能够是描述简单化。对于有大量端口的模块使用位置关联比较叫危险。

### 名字关联

名字关联就是利用名字的一一对应来链接信号和模块的端口。如下：

sc\_signal<int> a, b, result;

sc\_in\_clk clk;

alu\_p = new ALU(“ALU”);

alu\_p->clk(clk);

alu\_p->op\_a(a);

alu\_p->op\_b(b);

alu\_p->out(result);

### 模块内部数据

设计者可以声明内部变量来保存模块内部数据。内部数据存储类型可以是任何合法的systemc类型、c++类型和用户定义类型。一般地，在模块外部是看不到模块内部数据的，一个模块只可以通过端口来访问另一个模块。实际上，如果使用SC\_MODULE()来定义模块，则默认情况下所有数据成员都是公有的，其他模块可以通过直接访问的方式来读写一个模块的内部数据。除非是极特殊情况，这种方法最好不要使用。

### 模块的构造函数

模块的另外一个基本组成部分就是它的构造函数，熟悉C++的读者对此并不陌生。模块的构造函数完成创建和初始化一个模块的最初工作，他在一个模块的实例被创建时就被执行。C++中的构造函数创建模块内部数据结构，并把这些数据结构初始化为已知值。在systemc中，除了完成C++中所要求的基本功能外，构造函数还用于初始化进程的类型并创建进程的敏感表。

### 模块的析构函数

析构函数之在特殊的情况下才被使用，systemc没有专门的类似SC\_CTOR的宏来处理析构函数，析构函数还是按照传统的C++的模式声明。析构函数的作用是清除构造函数申请的内存单元。

## 端口和信号

### 基本概念

芯片通过管脚与电路版上其他芯片或者元器件通信，管脚有输入、输出和双向管脚。在systemc中，模块通过端口<port>与其他模块通信，端口也分为输出、输出和双向端口。一个模块的端口通过信号与其他模块的端口相连。

**systemc的信号与VHDL的信号一样引入了▲延迟的概念，即在同一个时间步▲内信号的值不变，只有下一个时间步到来的时候信号的值才可能更新。**

正常情况下，**端口总是与一个且仅有一个信号相连，也可以说端口总是与一个信号绑定的**。只有一个情况是特殊的，**就是当一个模块被实例化时，它的端口可以直接绑定到高一层模块的端口上。**

Systemc端口的定义方法如下：

1、输入端口：sc\_in<端口数据类型> 端口名；

2、输出端口：sc\_out<端口数据类型> 端口名；

3、双向端口：sc\_inout<端口数据类型> 端口名；

4、特殊情况：时钟的定义可以如“sc\_in\_clk 始终名称”

bind后，write调用绑定的write；read调用绑定的read；

有时候需要声明一个端口向量，如计算机数据和地址总线，这在systemc中也是被允许的

sc\_in<sc\_logic> a[32];

信号也可使用类似的声明办法

sc\_signal<sc\_logic> abus[16];

### 端口和信号的读写

在systemc中端口的读写可以是与verilog一样的赋值，如果定义了：

sc\_in<bool> data\_in;

那么，在进程中可以进行如下读写操作；

if (data\_in == TRUE) { }

另外，systemc中对端口还定义了read()和write()助手函数，以帮助完成可能需要的隐式类型转换。

If (data\_in.read() == TRUE) { }

对于需要隐式类型转换的场合，只能用助手函数read和write进行操作。

如下，直接赋值是非法的，使用助手函数是允许的

sc\_in<int> data\_out;

data\_out = data\_in; //非法

data\_out.write(data\_in.read()); //合法

对于信号，上面端口赋值的方法完全适用。

### 端口和信号类型

端口数据类型：

1、常见C++数据类型，如long、int、char、short、float、double。

2、systemc专有数据类型，如sc\_int<n>, sc\_uint<n>, sc\_bigint<n>, sc\_biguint<n>, sc\_bit, sc\_logic, sc\_bv<n> , sc\_lv<n>, sc\_fixed, sc\_ufixed, sc\_fix, sc\_ufix。

3、用户自定义结构

4、抽象端口： sc\_port<虚拟继承的类> arbiter\_port;

### 端口和信号的多驱动处理

systemc引入了解析逻辑向量信号来解决多驱动问题。例如：

sc\_in\_rv<n> x; //x被定义为n比特宽的解析逻辑向量输入端口

sc\_out\_rv<n> y; //y被定义为n比特宽的解析逻辑向量型输出端口

sc\_inout\_rv<n> z; //z被定义为n比特宽的解析逻辑向量型双向端口

类似地，解析型信号的定义方法如下：

sc\_signal\_rv<n> x; //宽度为n比特的解析型向量信号x

可以如下使用：

sc\_out\_rv<1> out\_1;

高阻态的时候可-> out\_1.write(“Z”); // SC\_LOGIC\_Z

解析型的端口比普通的端口需要更多的仿真时间，如果不是必须的，最好不要使用。

### 端口和信号的绑定

端口和信号的绑定方法分为位置关联和名字关联两种。在systemc中，端口到端口的直接绑定也是允许的。.....

### systemc时钟模型

时钟是同步数字逻辑中必不可少的基本元件，在systemc中，时钟被作为一个特殊的对象处理，它就是sc\_clock类。

在systemc中，sc\_clock一共有6个重载的构造函数，其中最后一个是为了与1.0版本的时钟模型兼容。name是时钟的名字，period\_ 和 period\_v\_是时钟周期，period\_tu\_是时钟的时间单位， duty\_cycle\_是占空比， start\_time\_v\_和start\_time\_是时钟初始第一逻辑值的持续时间， posedge\_first\_表示第一个逻辑值是高电平还是低电平。

具体函数实现参见代码

假设clk1的周期为50MHz，即时间单位应该为那秒ns，合适的定义方式：

sc\_clock clk1(“clk1”, 20, SC\_NS, 0.5, 5, SC\_NS, true);

名称 周期20个单位 单位 站空比 第一个逻辑时间 单位 逻辑高

常见时间单位：

SC\_SEC Second 1

SC\_MS Millisecond

SC\_US Microsecond

SC\_NS Nanosecond

SC\_PS Picosecond

SC\_FS Femtosecond

### systemc时间模型

2.01版本基于实数的时间模型修改为基于整数的时间模型，系统时间用一个64位无符号整数表示。使用整数的时间模型克服实数时间模型溢出问题。

时间分辨率是仿真系统能够处理的时间的最小精度，比时间分辨率更精细的时间将被四舍五入到时间分辨率所定义的精度。假设系统的时间分辨率10ps，则

wait(33.667, SC\_NS); == wait(33.67, SC\_NS);

systemc缺省的时间分辨率是1ps，同时提供了

sc\_set\_time\_resolution(double, sc\_time\_unit);函数，允许修改系统的时间分辨率。如下代码将系统的时间分辨率设置为10ps：sc\_set\_time\_resolution(10, SC\_PS)；

**systemc对时间分辨率的设置有以下要求：**

1、时间分辨率必须是10的幂

2、只能在仿真开始之前设置

3、只能设置1次

4、必须在任何的非零的sc\_time声明之前设置

systemc缺省的时间单位是SC\_NS， 同时允许通过调用sc\_set\_default\_time\_unit(double, sc\_time\_unit)来修改缺省的时间单位。如下代码将时间单位设置为100ps：

sc\_set\_default\_time\_unit(100, SC\_PS);

在时间单位设置为100ps的情况下，下面clk1的周期为10x100ps=1ns

sc\_clock clk1(“clk1”, 10);

**systemc对缺省时间单位的设置有以下要求：**

1、必须是10的幂

2、时间单位必须大于等于时间分辨率

3、只能设置一次

4、只能在仿真开始之前设置

### 基本数据类型

作为c++的一个硬件库，systemc支持所有c++数据类型，除此之外，systemc还定义了专有数据类型：

sc\_bit 2值单比特数据类型

sc\_logic 4值单比特数据类型

sc\_int 1~64比特有符号整形数据类型

sc\_uint 1~64比特无符号整型数据类型

sc\_bigtint 任意宽度的有符号整型数据类型

sc\_biguint 任意宽度的无符号整型数据类型

sc\_bv 任意宽度的2值比特向量数据类型

sc\_lv 任意宽度的4值比特向量数据类型

sc\_fixed 模版类有符号定点数据类型

sc\_ufixed 模版类无符号定点数据类型

sc\_fix 非模版类有符号定点数据类型

sc\_ufix 非模版类无符号定点数据类型

### sc\_bit和sc\_logic数据类型

sc\_logic比sc\_bit多‘X’和‘Z’，实现起来必然更复杂一些，仿真速度也要慢一些。虽然多数情况下可以使用sc\_logic代替sc\_bit， 但推荐做法是在能够使用sc\_bit的时候尽量不要使用sc\_logic。

### 固定精度整型数据类型sc\_int和sc\_uint

在数字系统建模中常常需要对固定宽度的整型数据进行操作。c++的固有数据类型int的宽度是与所使用的机器有关的，通常是32位。使用c++固有数据类型显然可以提高仿真速度，但只能实现对8/16/32位的数据操作。因此systemc中引入了sc\_int和sc\_uint来实现1~64比特中任意宽度的整型数据类型，又引入了sc\_bigint和sc\_biguint来实现任意宽度的整型操作。显然使用sc\_int和sc\_uint的仿真速度要比使用sc\_biting和sc\_binguint快，因此推荐的方法是，在能够使用sc\_int和sc\_uint的时候尽量不要使用sc\_bigint和sc\_binguint。

例如：

sc\_int<43> a; //定义了一个43位的有符号整型数

sc\_int<64> b; //定义了一个64位的有符号整型数

**不要超过64**

***sc\_int和sc\_uint可以相互赋值，系统自动完成转换，规则如下：***

***1、无符号整数uint1(sc\_uint<M>)被赋值给有符号整数int1(sc\_int<N>)时，uint1首先被扩展为64位，然后从低位开始取N位赋值给int1。当int1被赋值给uint1时， 系统首先将它按照符号扩展64位，然后从低位开始取M位赋值给uint1。***

### 任意精度整型数据类型sc\_bigint和sc\_biguint

在实际设计中，有时候需要宽度大于64的整型数据类型，systemc引入了sc\_bigint和sc\_biguint，其宽度可一从1到任意值，与sc\_int和sc\_uint相比仿真速度要慢得多，因此如果不是必要，就不应该使用sc\_bigint和sc\_biguint，而尽量使用sc\_int和sc\_uint。

由于要考虑任意精度，所以sc\_bigint和sc\_biguint仿真速度大大下降。在sc\_constants.h中，用户可以根据自己的需要定义SC\_MAX\_NBITS，以约束sc\_bigint和sc\_biguint的最大宽度，这样可以增加2~3倍仿真速度，这个常量在默认情况下是没有定义的，用户可以根据自己的修要进行修改，systemc推荐的SC\_MAX\_NBITS值应该为BITS\_PER\_DIGIT的整数倍。

### 任意长度比特和逻辑向量

为支持任意宽度的比特操作，systemc定义了支持任意宽度的比特向量类型sc\_bv<W>，其中W为比特向量的宽度。比特向量适用于之需要二值逻辑向量而不需要算术运算的场合。sc\_bv面向比特操作优化，但不支持算术运算，它的仿真速度高于systemc整数类型。

比特向量的赋值需要使用‘0’和‘1’构成的字符串进行赋值，例如：

sc\_bv<6> val;

val = “101101”;

为了支持宽度大于1的三态总线建模，systemc定义了sc\_lv<W>，它与sc\_bv<W>的区别是，它是sc\_logic类型的向量，因此支持‘0’，‘1’， ‘X’， ‘Z’四值逻辑。

### 用户自定义类型

除C++固有数据类型和systemc专有数据类型外，systemc作为c++的扩展，还支持用户自定义数据结构。自定义的数据结构为了能够对它进行操作，必须定义运算符重载和方法函数。

### 定点类型

参见文档和代码

## 进程

使用普通的编程语言可以很容易地描述系统的顺序行为，描述系统的并行行为是困难的，所以在硬件描述语言中，为了描述硬件系统的并发行为也引入了进程概念，与操作系统中的进程相比它更简单，进程类型和进程状态也是有限的。

在systemc中，进程是一个基本执行单位，它被调用来仿真目标系统的行为。Systemc的基本进程有三种：

1、SC\_METHOD

2、SC\_THREAD

3、SC\_CTHREAD

systemc master-slave库中还定义了第四中进程SC\_SLAVE。

### 进程基础

进程的行为是多样的。有些进程的行为像函数一样被调用后立刻返回，有些进程在仿真开始以后只运行一次，一直运行到仿真结束，有些进程在运行过程中可能被挂起直到某一个条件满足。引起进程运行或者解除挂起的条件可能是时钟的边沿或信号值的变化。

systemc中，进程不是层次化的。一个进程中不能包含或者直接调用其他进程，但进程可以调用非进程的函数和方法。

进程通常会有一个敏感表。在当敏感表中的信号上有事件发生时，进程就会被激活。信号上的事件是指信号值的变化，如时钟的上升沿就是时钟信号从0变为1。当信号上的时间发生，所有对该事件敏感的进程都会被激活。

### 方法进程SC\_METHOD

在systemc中，方法进程SC\_METHOD是唯一的可以综合的寄存器传输级进程。它的特点是当敏感表上有事件发生，它就会被调用，调用后应该立刻返回。只有该类进程返回后仿真系统的时间才有可能前进，因此该类进程中不能使用wait()类语句。如果该类进程内部有一个无穷循环，可以想像仿真系统时间将会停止前进。

SC\_METHOD进程的敏感表在模块的构造函数内设定。

### 线程进程SC\_THREAD

线程进程的特点是它能够被挂起和重新激活。线程进程使用wait()挂起，当敏感表中有事件发生，线程进程被重新激活运行到遇到新的wait()语句再重新挂起。

由于线程进程的这个特点，从理论上讲，用它可以描述几乎一切系统的行为。

线程进程不是寄存器传输级进程，一个方便的用途就是用来描述验证平台的输入输出。

***可以使用SC\_THREAD进程来完成Monitor的设计，有时候使用SC\_METHOD进程实现会更简单些，有时候适合SC\_THREAD，有时候适合SC\_METHOD***。

### 钟控线程进程

钟控线程进程是一种特殊的线程进程，它继承于线程进程，但只能在时钟的上升沿或者下降边沿触发或者激活，这种行为更加接近实际硬件的行为。引入钟控线程进程的目的是产生更好的行为综合的结果。

钟控线程进程最适合来描述隐式有限状态机。所谓隐式有限状态机是指编程中并不显示地定义状态机的状态，而是通过程序中的wait()语句和wait()语句中间的赋值语句来完成对状态机的描述。显示的有限状态机通常要明确定义系统的状态，要用case语句来实现状态转移。

### wait\_until()、wait()和next\_trigger()

wait\_unit将进程挂起直到指定的表达式的值为真，它只能用于线程进程和钟控线程进程，例如：

wait\_until(布尔表达式)

wait()函数有以下几种参数形式：

1、wait() //等待敏感表中有事件发生

2、wait(const sc\_event &) //等待事件发生

3、wait(sc\_event\_or\_list &) //等待事件之一发生

4、wait(sc\_event\_and\_list &) //等待事件全部发生

5、wait(double v, sc\_time\_unit tu) //等待由v和tu决定的一段时间

6、wait(double v, sc\_time\_unit tu, const sc\_event &e) //在由v和tu决定的时间内等待e发生

7、wait(double v, sc\_time\_unit tu, const sc\_event\_list &el) //在时间内等待e列表发生

8、前面所有的(dobule v, sc\_time\_unit tu)两个参数可以用一个sc\_time型参数替代

9、wait(0, SC\_NS)或者wait(SC\_ZERO\_TIME) //等待一个▲时间

next\_trigger()只能用于SC\_METHOD类进程。nex\_trigger()的参数与wait()的参数相同，区别是他们分别用于不同类型的进程。next\_trigger()的特点是调用后立刻返回，在一个SC\_METHOD进程中可以多次调用next\_trigger(),但由最后一个调用决定进程的下一次激活时间。

### watching结构

SC\_CTHREAD进程通常只有一个死循环，但有时候需要初始化一些变量和信号，或者当某些条件满足的时候能够让进程从循环中跳出来。使用watching结构可以跳出循环。watching结构会不停地监视某一个条件，一旦该条件发生，则SC\_CTHREAD进程就会跳出循环从进程的开始处重新执行。

***watching(布尔表达式)***

### 局部watching

有时候需要局部代码重新运行，这时候需要使用局部watching，其框架如下：

W\_BEGIN

// watching的条件声明放在这里

watching(...);

watching(...);

W\_DO

//正常执行的代码

W\_ESCAPE

//当watching的条件成立时执行的代码放在这里

W\_END

说明：

1、放在W\_BEGIN和W\_DO之间的观察条件的优先级是相等的。

2、允许嵌套使用局部watching以实现不同的优先级

3、全局的观察条件的优先级要大于非全局的观察条件的优先级

4、观察条件只在SC\_CTHREAD进程所工作的时钟的边沿被采样

## 仿真与波形跟踪

### 仿真控制

调用sc\_start()函数，仿真开始。sc\_start()函数控制所有时钟的产生并在适当的时候激活systemc调度器。systemc调度器控制整个仿真过程中的调度工作，包括激活进程、产生▲延迟、计算和更新变量和信号的指等。一般来讲，sc\_start()只在sc\_main()中调用。

sc\_start函数原型参见代码

sc\_stop函数控制仿真的停止。一旦遇到sc\_stop()，systemc调度器就会停止运行。

sc\_simulation\_time()，在仿真中，经常需要知道当前的仿真时间以分析仿真结果是否与语气相符，

该函数返回当前仿真时间的双整型值。

### systemc波形跟踪概述

Systemc允许将仿真结果保存为以下三种标准的波形：

VCD

WIF

ISDB

systemc的波形跟踪有以下特点：

1、只有在整个仿真期间都存在的信号和变量才能被跟踪。能够保证模块中的所有信号和数据成员都被

跟踪，函数的本地变量只有在函数被调用期间才存在，所以不能跟踪。

2、任何类型的信号和变量包括标量、数组和其他聚合结构都能被跟踪。

3、不同格式的波形文件可以在同一次仿真过程中同时产生。

### 创建和关闭波形跟踪文件

波形跟踪一般在sc\_main()函数中进行，并且放在所有的信号和模块的例化之后。第一步是产生波形跟踪文件，相应的函数为sc\_create\_xxx\_trace\_file()，这里的xxx可以是vcd、wif和isdb。

在仿真结束的时候应该将波形跟踪文件关闭，否则将会出现或者文件为空或者文件大不开等错误。

sc\_close\_xxx\_file(my\_trace\_file);

### 跟踪标量型变量和信号

创建波形文件后，还必须告诉systemc调度器到底要跟踪那些信号和变量，以及被跟踪的信号和变量在波形文件中保存的名字。这时需要调用sc\_trace()函数：

sc\_trace( sc\_trace\_file\* tf, const tp& a, const sc\_string& name )

tf ：为指向波形跟踪文件的指针；

object ：是对要跟踪的信号或者变量的引用，这里object只能是基本的c++/systemc标准数据类型；

name ：跟踪信号或者变量在波形文件中保存的名字；

### 跟踪聚合型变量和信号

sc\_trace()函数只能跟踪标准类型的信号和变量，为了跟踪聚合类型的变量和信号，需要重载sc\_trace函数。例如：

void sc\_trace( sc\_trace\_file\* tf, const pkt& a, const std::string& name )

{

sc\_trace( tf, a.data, name + ".data" );

sc\_trace( tf, a.id, name + ".id" );

sc\_trace( tf, a.dest0, name + ".dest0" );

sc\_trace( tf, a.dest1, name + ".dest1" );

sc\_trace( tf, a.dest2, name + ".dest2" );

sc\_trace( tf, a.dest3, name + ".dest3" );

}

# 寄存器传输级systemc设计

RTL设计的重要特性是可以被综合器综合为门级电路，而门级电路最终可以实现为专用集成电路或者下载到FPGA/CPLD。

## systemc寄存器传输级设计和综合

### 综合

寄存器传输级设计的重要特征是可以综合。综合是指将RTL或者行为级的硬件描述语言的描述转换为满足约束条件的网表的过程。

对一个电子系统的描述可以有不同的层次，主要的层次按照其抽象程度由高到底，可分为系统级、算法级，也称为行为级、寄存器传输级、门级、和开关级。

**系统级** ：对整个系统功能的定义和结构的探索。

**算法级** ：确保一些核心算法和系统行为的正确。

**寄存器传输级** ：利用组合逻辑和时序逻辑来描述电路。

***门级*** ：偏重工艺描述

***开关级***  ：

由于寄存器传输级描述了电路的结构，但又不像门级描述一样需要太多的细节，从而保持了适当的抽象，成为逻辑综合的起点。

逻辑综合工具使用两个主要阶段对寄存器传输级设计进行处理。第一个阶段是将设计读入并且进行分析并化简。第二个阶段是工艺映射，将第一阶段读入的设计转换为与元件库中的元件相匹配的形式。

为了让综合器能够正确的把我们的设计综合为需要的电路结构，我们必须遵守一定的设计规范。因为有些systemc的语言结构，综合器并不支持，有时候用不同的代码描述的同一设计，仿真行为相同，但综合后的电路结构大不一样，效率差别也大。

**1、verilog建模方式分为：行为级和结构级**

**2、行为级建模包括系统级、算法级和RTL级**

**3、结构级也称为“门级和开关级”，包含模块实例和基本元件实例**

systemc也一样

### 为什么要使用systemc进行RTL建模

前提：系统设计者已经创建了一个基于systemc的SOC系统级结构模型，在这种情况下开发一个周边设备。会有如下问题：

1、表述问题

如果使用verilog的综合工具，需要重写systemc的代码。这会花很多时间，而且容易出错。也可以使用systemc来进行周边设备的综合，这样就可以直接利用系统级设计已经完成的工作，把抽象的、不可综合的代码利用一定规则，重新修改为可综合代码。这个过程大大减少出错的可能。

2、验证重用

使用verilog的流程，还要重写测试平台来验证代码是否正确，而在给予systemc的综合设计流程中，可以直接利用systemc的可执行设计规范，作为系统级验证环境，来验证重写的代码。

3、工作效率

修改比重写快，验证修改的设计比验证重写的设计快，所以效率高

## RTL风格的systemc编程

描述systemc和c++的RTL风格语言要素

### 定义模块和进程

模块是systemc的基本构件，一个典型模块可能包含：

1、单个或多个RTL进程，用来描述组合或者时序逻辑

2、多个RTL模块，描述层次化

3、一个或多个成员函数，执行具体硬件动作

进程的触发是靠敏感表来进行。敏感表规定了能够触发进程的输入和输出端口。我们可以定义电平触发的敏感输入信号，用来对组合逻辑电路进行建模，也可以定义边沿触发的敏感信号，用来对时序逻辑电路进行建模。

进程间主要利用信号来相互通信，一个进程对链接它和另外一个进程的信号赋新知值，能够触发另外一个进程的执行。

进程间也可以利用变量来进行通信，但实际设计中应尽量不要使用变量来进行进程之间的通信。因为进程是以随即顺序来执行的，而变量的交换是与执行顺序有关的。这样就会有不确定因素。

systemc提供三种进程类型，但对于RTL，我们只能使用SC\_METHOD。

### 创建模块

对于一个模块，可以声明任意数量的sc\_in, sc\_out和sc\_inout型端口。为了从一个输出端口都去数据，我们可以把它声明为一个sc\_inout型的端口，而不仅仅是一个sc\_out型端口。也就是说sc\_inout型端口可能描述的不是一个双向端口，而仅仅是一个内部模块也能访问的输出端口。

端口与信号相互链接，数据类型必须相同。为了能够综合，每个端口的数据类型必须要被声明为可综合的数据类型之一。

可综合风格的systemc代码只能用SC\_METHOD类型的进程。该进程的返回类型必须被声明为void型，而且不带入口参数。可以在模块里声明不是进程的成员函数，它可以被进程调用。非进程的成员函数可返回任意的可综合的数据类型。

每个模块都需要一个构造器，他的作用：

1、注册进程

2、定义SC\_METHOD的敏感表

对于可综合代码，除了上述两种语句外，其余的语句不允许出现在构造器中。

### 定义敏感表

能够与一个SC\_METHOD进程交互作用的信号集合被称为这个进程的敏感表。在敏感表声明中，可以使用sensitive()，sensitive\_pos()或者sensitive\_neg()函数，也可以使用字符串形式的声明sensitive, sensitive\_pos或者sensitive\_net。

可综合代码的敏感表必须是完备的，不完备的敏感表综合以后会产生不应该出现的锁存器，从而导致综合结果不够优化、综合前后的仿真结果可能不一致，并且不利于静态时序分析。这一点对组合逻辑非常重要。对于组合逻辑，我们需要把组合逻辑进程的所有输入全部放到进程的敏感表中。

定义敏感表时，需要注意一下两点：

1、不能在要综合的同一进程中同时指定边沿触发和电平触发输入。

2、不能把时钟声明为sc\_logic类型，只能将其声明为sc\_in<bool>或者sc\_in\_clk型。

### 信号和变量的读写

读写端口的时候，推荐的编程风格是使用read()和write()方法。读写变量的时候，推荐的编程风格是使用赋值算子。

我们只能对端口和信号的所有位进行读写，而不能对一些特定的位进行读写。要对端口或者信号进行特定比特位的选择，首先把值读入一个中间值，然后对这个中间值进行位选择。

当我们给一个变量赋值时，赋值语句等号右端的值会立刻被送到左端。这和对信号和端口赋值明显不同。

# SystemC行为建模

## 行为级建模的目的

几乎所有的硬件描述语言都支持行为级建模。对于systemc，行为级建模似乎更加重要。行为建模的主要目的是从更高层次描述系统的行为，从而减少仿真时间以加速设计收敛。特别的，有些行为描述可以通过行为综合工具进行综合而得到RTL代码。这往往比设计者自己完成寄存器传输级设计效率更高、性能更好。

当要建模的目标系统十分复杂时，利用行为模型迅速建模，使得设计者对目标设计有更清楚的理解，则有利于软硬件划分、减少系统功耗。给予行为模型的软硬件系统验证更使得设计初期就可以对软件进行初步调试，从而节省设计周期，这也是当前进行行为建模的主要目的之一。

SystemC行为级建模中一个很重要的概念被称为交易级建模<transaction level modeling>。

## 接口、端口和通道的基本概念

为了支持进程同步和通信细化，systemc定义了接口<interface>、端口<port>和通道<channel>。

**接口：**

在systemc中，接口是一个C++抽象类。这种抽象类的特点是它定义了一组抽象方法，但不定义这些方法的具体实现，这些方法都以纯虚函数的方式出现。

**通道：**

通道实现一个或者多个接口。也就是说，通道必须继承一个或者多个接口，这些接口中定义的抽象方法必须在通道中实现。通过端口，模块中的进程可以链接到通道并使用通道提供的方法。端口总是与一定的接口类型相关联，端口只能链接到实现了该类接口的通道上。

通道分为基本通道和分层通道。基本通道是非结构化的，它不包含进程，也不能直接读写其他基本通道。而分层通道是一个特殊模块，它可以包含字模块和进程，当然也可以直接读写其他通道。

这种接口的定义和实现分开的方法是面向对象中的一个重要概念，称为接口方法调用。一个进程通过端口调用通道提供的方法，但它不必知道具体的方法如何实现。通道自己实现了具体的方法，但自己却不能调用。

**端口:**

systemc的基本类型包括sc\_in<T>、sc\_out<T>、sc\_inout<T>。利用它们，我们就可以完成RTL级设计。为了满足行为建模的需要，systemc允许用户自己定义端口类型。

### 自定义端口

端口与特定的通道接口相连，或者同父模块的端口相连。进程通过特定的端口调用通道的接口提供的方法。对于基本的端口类型，可以调用的接口方法有write()和read()，但这些在有些情况下是不能够满足我们的要求的。比如：

1、当端口与总线接口或者存储器接口相连的时候，需要同时提供地址和数据

2、进程可能需要了解通道的状态信息。

3、高层次建模中与数据读写无关的一些操作，如复位操作

4、高层次建模中的进程的动态敏感，如wait\_for\_response()、wait\_for\_request()。

端口定义方法：

sc\_port<interfaceType, ChannelNumber = 1>

interfacetype是端口所要链接的通道的接口类型， channelnumber代表端口所要链接的最大通道数，默认为1，当链接数目未定时可以使用0，代表可以一个端口链接无穷多个通道。

### 端口基类sc\_port<IF, N>

该类是所有端口的基类，是一个模版类。IF是接口类型，N是所链接的同一类型的通道数目，也就是接口数，缺省是1。

**重载操作符operator()(IF & interface) 用于将端口与接口绑定起来，实现端口与通道的接口的互通，这是端口的最基本操作**。操作符operator()用于将端口与父模块的端口绑定起来，使端口支持分层的模块化设计。IF \*operator->()和const IF \*operator->() const实现接口方法的调用。利用size()可以查询当前端口所链接的同一类型的接口数。**当多个接口链接到同一个端口上时，可以利用operator[](int index)来调用不同的接口的方法**。

### 直接通道调用

对于很多HDL设计经验的读者，端口的存在似乎是自然的和必须的；但对于长期从事软件设计的设计者来说，直接调用通道的方法而不使用端口才似乎是自然的。

在systemc中并不允许这样做。systemc规定了不同的模块之间进行通信必须通过端口。通过端口，有利于设计重用，而且通过端口还可以进行静态的或者动态的设计规则检查，或者附加属性，如优先级、建立和保持时间、在FPGA中是否允许使用I/O逻辑块的寄存器等。

在同一模块内，各个进程之间也需要通信，他们可以通过共享变量、握手信号、模块内通道等方式通信。如果它们直接的通信是通过模块内通道，则此时不需要通过端口，即所谓直接通道调用。

systemc中与定义了许多通道如：sc\_fifo, sc\_mutex等。

## 通道基础

系统抽象的三个关键是行为、时序和通信。从广义上来讲，模块是系统行为的主要载体，通道则是通信的主要载体。而通道本身也可能是模块，也表现出一定的行为。

在systemc中，接口本身只是定义了一组通信方法，而不具体负责这些方法如何实现。通道才是这些接口方法的实现者。通道可以实现一个或者多个接口。同时通道也可以链接两个或者多个模块。systemc允许用户自己定义通道，以实现多样的抽象系统模型。

systemc中的通道分为两中，基本通道和分层通道。基本通道不包含任何进程，也不对外展现任何可见结构，他们也不能够直接的或者间接的调用其他基本通道，像sc\_mutex。而分层的通道本身是一个模块，当然可以包含进程、子模块。也可以包含和调用其他通道。

### 通道的同步规则

systemc的通道允许并行的动作，这就涉及到同步的问题。比如：在同一时钟的上升沿既读又写，则读的结果应该是写入之前的值。通用的做法是将通道的操作分为两个部分进行，即所谓的**“求值-更新”**过程。在求值阶段，新的结果被记录，同时保存原有的结果。如果是读操作，则在求值阶段将原有的结果返回，如果是写操作，则接着执行更新过程，真正将新的数据写入。可以看出，更新过程并不是必须的，比如读操作和状态查询操作。

所有的基本通道都有sc\_prim\_channel继承而来，sc\_prim\_channel完成的是对**“求值-更新”**过程的基本支持。

### 静态规则检查

设计过程中有一些不能违反的设计规则，如一个sc\_signal<T>只能被一个进程驱动，一个sc\_fifo<T>只能链接一个读端口和一个写端口等。在设计中进行规则检查能够尽早发现设计中产生的一些不合理性。

设计规则检查分为静态规则检查和动态规则检查。静态规则检查是指在系统运行前进行的静态检查，比如一个sc\_fifo<T>只能链接一个读端口和一个写端口的检查。但是只有静态规则检查是不够的，有些规则检查必须在运行时间进行，这就是动态检查。比如，一个sc\_signal<T>只能被一个进程驱动的规则只能在运行时动态地进行检查。

systemc约定：当一个端口与一个通道相链接，这时相应通道的：register\_port()函数就会被调用。用户可以自己重载这个函数进行专门的静态设计规则检查。

### 动态规则检查

### 通道的属性

通道的属性描述的是通道的一些特征，比如：所链接到通道的端口的优先级，由通道制定各个端口的优先级显然比各个端口自己来指定自己的优先级要好些，因为这样的端口可以不管自己的优先级，而由通道做全局的设置，这样也有助于接口与实现分开。对于通道属性的概念，后面详细讨论。

## 基本通道

基本通道不包含任何进程，也不对外展现出任何可见结构，他们也不能够直接的或者间接的调用其他基本通道。

sc\_signal<T>, sc\_signal\_rv<N>

sc\_mutex

sc\_fifo<T>

sc\_semaphore

sc\_buffer<T>

### sc\_signal<T>、sc\_signal\_rv<T>和sc\_buffer<T>

sc\_signal<T>是最基本通道，用于链接模块的基本端口sc\_in<T>, sc\_out<T>, sc\_inout<T>。sc\_signal<T>的设计规则最多只有一个sc\_out<T>或者sc\_inout<T>可以链接到sc\_signal<T>,否则就会产生典型的多驱动情况， sc\_signal<T>可能会在同一时刻被赋予不同的值，将导致系统行为的不确定性。可以有多个sc\_in<T>同时链接到sc\_signal<T>，这将导致一个信号的值被赋给多个信号，这是允许的，也是应该的。

sc\_signal\_rv<T>是所谓解析的信号通道，它允许同时有多个端口链接到其上并进行写操作。

sc\_buffer<T>继承于sc\_signal<T>：sc\_buffer<T>不管write()写的数据是否与原数据相同，都要求进行数据更新；而sc\_signal<T>首先要检查新数据是否与原数据相同，如果不相同才进行更新。这是唯一区别。

### sc\_mutex

互斥。。。。。

### sc\_fifo<T>

systemc库已经实现好的FIFO通道。

### sc\_semphore

信号量。。。。

## 分层通道

### 分层通道定义

基本通道只实现了一个或者多个特定的接口，但不具有任何的可见结构。而分层通道则具有可见结构，包含进程，可以直接调用其他通道，它是一个实现了一个或者多个接口的模块。

分层通道能够建模复杂的硬件模块。一个典型的可以用分层通道建模的例子就是片上总线。目前流行的片上总线有OCB，AMBA，OPB等。一个利用分层通道实现片上总线是一个抽象的行为模型，它与所采用的具体总线架构无关，具有很好的重用性。

**一般，一下情况可以考虑使用基本通道：**

1、当需要使用“求值-更新”的策略时。

2、当通道的功能特别基本，很难划分成独立的功能块时。

3、仿真速度对于设计十分关键，使用基本通道能够减少▲周期，节省仿真时间。

4、当使用进程没有明显的意义时。

**建议使用分层通道：**

1、当通道的结构层次很明显，用户需要了解通道的底层结构时。

2、当通道应该包含进程时。

### 一般分层通道

凡是通道中例化其他通道的情况都可以归入一般分层通道之列。

### 特殊分层通道

合成通道称为特殊分层通道，它间接的体现了通道的层次行。通过合成通道，一个模块可以利用端口穿越了一个以上的通道进行间接通道调用。

合成通道的可复用性和可读性较差，在实际建模中很少使用。

## 系统建模中的分层模型

交易级建模和通信细化是systemc中的重要概念，

### 系统建模中通信的抽象层次

系统建模是将底层细节屏蔽的抽象过程。如下列出了通信体系结构的抽象层次和每个层次屏蔽掉的细节。

*消息层(Message Layer) L3 屏蔽了资源共享，时序*

*交易层(TransactionLayer) L2 屏蔽了时钟，协议*

*传输层(Transfer Layer) L1 屏蔽了连线，寄存器*

*寄存器传输层(RTL Layer) L0 屏蔽门，门/连线延时*

每个层次包含三种组建：发起者、通道和目标设备。发起者通过接口，利用通道链接到目标设备上。接口给发起者和目标设备提供了通道所提供的服务。这样在保证提供基本的服务需求下就可以方便的修改通道，而无需修改发起者和目标设备。通信分层结构能够支持基于端口的设计方法。来自不同抽象层次的发起者和目标设备间的通信操作能够利用一些适配器实现，这些适配器实现对多个抽象层的包装或者能够适应于多个抽象层次。

### 寄存器传输层

寄存器传输层描述了存储单元和组合逻辑块之间的互连关系，可以直接进行逻辑综合得到门级网表<netlist>。有关RTL的建模可参考上面章节描述。

寄存器传输层的描述主要用来进行Soc的最终硬件实现。特征：

1、精确到管脚和比特

2、形成最终可综合的VHDL，Verilog或者SystemC代码

3、能够反向标注估计的传播延迟

通信抽象模型与集成电路设计中的分层抽象模型不同。在集成电路设计中的分层抽象模型中，RTL层下面还有门级和晶体管级，而通信抽象中则一般不谈这两个级别，这也说明通信抽象更高级。

### 传输层

传输层在寄存器传输层之上，它的特点是这个抽象层中的系统是由精确到周期的行为组成，寄存器传输层也是精确到时钟的。传输层的系统建模一般对应着一定的总线协议。这一层的系统模型是有时钟的，主设备和从设备之间的一次交易在时钟事件上完成。一次交易代表了一次数据传输和相应的总线控制信号的握手。

这一层的模型并不需要写成可综合的实现代码，而只需要提供一个精确到时钟周期、符合协议要求的模块互连机制和一个快速仿真的测试平台。L1的模块可以通过适当的转换与L0的RTL模块互连，也可以与L2的通道互连。

传输层模型的功能可以由寄存器传输层模型完成，但是与寄存器传输层相比，**传输层的特点在于**：

1、网表简单，整个通信接口之需要一跟线即可完成，网表不随着参数改变

2、 仿真速度快。通信信道的实现不需要信号，而通过事件、变量和函数完成，这比RTL层的仿真快。

3、功能块和通信信道之间的接口代码更为简单，功能块仅需要得到通道的函数调用，不需要知道接口信号。

**该层主要用途**：

1、是L3和L2模块和精确到周期的模型之间的桥梁。

2、为一些IP模块<eg. 嵌入式处理器>的抽象仿真模型提供精确到周期的接口。

3、进行精确到周期的测试平台建模

4、进行精确到周期的性能仿真。

**主要特点**：

1、将精确到周期的协议映射到给定的硬件接口和总线结构上

2、隐藏了接口的管脚

3、精确到字节的数据

4、交易具有内部结构

5、交易直接映射到总线周期

6、可以通过改变参数来对不同的总线协议和信号端口进行建模

### 交易层

交易层是L2，该层的系统模型带有时间信息，但并不精确到时钟周期，系统是时间驱动执行的。发起者设备和目标设备之间的一个交易可能包含几个数据的传输(也就是一个突发型传输)。执行这个交易所需要的时间由主设备和从设备来估计。通常的交易级系统模型与总线协议()无关，因为总线协议总是与精确到时钟周期的系统有关。

**交易级建模的主要用途为**：

1、硬件体系结构的性能分析和行为分析

2、软硬件划分和协同设计

3、周期性能分析

4、作为底层设备驱动和硬件仿真模型的接口

5、集成操作系统仿真器和硬件模拟器

6、作为精确到周期的模型和交易级模型的仿真平台

7、作为测试图样，或者模拟设计的系统环境

**特点**：

1、交易层是将理想的结构映射到需要考虑资源分配和设计约束的结构中

2、存储器和寄存器的映射是精确的

3、允许多线程通信

4、可以通过对数据类型的约束来对总线的突发传输或者突发传输片段进行建模

5、事件驱动的仿真机制，带有时间估计

6、能够基于传输量和延时约束进行延时性能估计。性能估计能够采取基于报告的方式。也就是说，

时间驱动的通道能够根据总线宽度和总线协议计算并报告一个交易需要多少个时钟周期，或者

通过插入延时来仿真时间。

7、能够利用参数来精细的调整设计规范，以解决约束过紧或者过松的情形。

8、利用参数来对不同的总线协议和信号接口进行建模。

### 消息层

消息层是L3，L3的系统没有任何时间信息，系统执行是事件驱动的。主设备和从设备之间的一次交易包含了多个数据的传输，这些数据可能处于非常抽象的层次。

消息层系统模型主要用来证明系统概念，制定可执行的设计规范，功能划分和高层的设计折中等，**包括**：

1、对数据通道和控制单元进行功能划分

2、系统通信定义和最初的概念型系统的测试平台

3、与高层的系统仿真(比如SDL， UML)继承

4、算法性能、行为和控制探索

**特点**：

1、实现结构的抽象

2、事件驱动的仿真语法

3、原语支持类似RTOS的通信和同步方法。利用复制和指针操作来进行数据传输。

4、点对点的主从设备互连。

5、支持抽象数据类型

6、不使用地址映射，而使用进程映射

7、支持阻塞型和非阻塞型的消息传递端口，不要求实现消息队列

8、某些系统中，如果从设备等不到事件发生，就会忽略它们，而主设备必须做出处理

9、支持非流水线风格和时间顺序的流水线风格，非分离的或分离的端口等

## systemc的交易级建模

### 交易的概念

在片上系统建模中，交易(transaction)是一个十分重要的概念。一般来说，交易可以理解为系统模型中两个组建之间的一次数据交换。这个交换与所采用的协议无关，因为交易级的模型通常不牵扯到具体的总线时序等细节。一个数据交易可能是在系统组件之间传输的单个字，一列字或者整个数据结构。例如：一个DMA主设备能够发出请求，从寄存器中读入一个数据。它首先要发出一个读交易，制定目标存储器的地址。另外一种情况，当嵌入式的软件要向DMA控制器中写入控制字时，就会发出一个写交易。为了保证SoC模块之间的同步所做的操作可以被看作是一个事件交易；而组建之间的中断也可以被视为一个交易。

### 嵌入式软件开发与交易级建模

交易级建模提供了一个非常重要的用途就是可以在整个设计的比较早的阶段就开始进行嵌入式软件的开发。实际上，大多数的SoC设计都包含至少一个可编程的处理器，软件开发是soc设计中重要的一部分。TLM模型能够提供非常快速的硬件仿真速度，从而提高开发的效率。工程实践中的TLM模型仿真的速度要比RTL模型仿真的速度快100~1000倍，每秒钟能够仿真至少100k个交易。

虽然一些与底层硬件实现密切相关的软件开发，要等到实际的RTL模型开发完成才能进行，上述进程仍然能够节省很多时间。

### 交易级建模用于系统结构探索

TLM模型包含了时间之间的正确顺序，而没有底层的物理延时，从而为软件设计者提供了一些初步的性能分析。而系统设计者对含有时间信息的TLM模型更感兴趣，因为这种模型可以进行系统结构的一些评价，从而进行系统结构的探索和优化。虽然RTL模型会提供更为精确的时间信息以用于分析，但是它也有缺点。首先，由于考虑的细节增多，它的开发比TLM开发要困难。其次，当进行结构优化时，带有物理细节的模型要比TLM模型更难以更改。

### systemc交易级建模的特点

进行交易级建模的soc设计一般都是需要软硬件协同设计的混合系统。大多数的系统带有一个或者多个嵌入式的微处理器和相应的片上总线通信结构。完成各个功能模块之间的相互通信。交易级建模的主要任务是利用systemc进行相应的通信抽象，实现通信机制。

Systemc的通信机制有两个特点：

1、功能与通信分离。也就是实现具体算法的部分与实现数据和事件传输的部分分离。功能由systemc的模块(sc\_module)来实现，而通信由通道(channel)来实现。

2、接口方法调用(interface Method call, IMC)。一组给定的通信方法(method)被称为接口(interface),包括数据接口和控制接口。通道(channel)是由一个或者多个接口来实现。模块能够使用他们的端口(port)与实现相应接口的通道进行互连。

## 通信细化

### 通信细化的概念

当最佳系统体系结构被选定，TLM模型被逐步的细化，转换为可以综合的寄存器传输级模型。

所谓通信细化是指，在保持接口不变的情况下具体实现的细节可以由粗糙的功能模型向更低抽象级别的模型和具体的实现细节转换，单独修改或者替换某一个单元不会影响其他单元，一般需要引入适配模块以保持到其他模块的接口不变。

通信细化本身包含了很多内容，在各个层次的模型向下一个层次的模型进行转换的时候都要进行通信细化。实际上在系统综合等过程中通信细化是很关键的技术。狭义上，可以将通信细化简单的理解为将TLM模型转化为RTL模型的过程。

# systemc的master-slave通信库

## Systemc master-slave通信库综述

systemc可以支持业界的很多设计方法学。OSCI的思路是定义一个小的语言子集作为核心语言，以完成最基本的语言架构。具体的设计方法学和具体的应用是由一些专用库实现的。比如master-slvae通信库和verification库。

当今集成电路的发展方向是片上系统集成，通常一个片上系统包括一个或者多个处理器、DSP、周边设备和用户定制集成电路，这些模块利用一个总线体系相互通信。为了对这样的体系高校建模，systemc引入了master-slave库，对片上通信的系统进行抽象。master是发起传输的设备，slave响应主设备发出的请求。他们之间的通信由通道完成连接。master-slave通信库还引入了进程之间的内嵌通信机制，适用于对软件-软件通信、硬件-软件接口和硬件-硬件接口之间通信的建模。他还在功能级定义了master-slave的通信协议，当它被细化到master-slave总线协议时，保留了功能级定义的通信和执行顺序。利用这个库，复杂的系统模型能够利用顺序通信的功能模块之间的互联来构建，从而屏蔽无用的实现细节。

master-slave通信库还使得通信与模块的行为能够在一定程度上分离。这样的好处，首先在于可以方便地向soc中集成IP模块。封装成为抽象通信模型的IP模块能够使用接口综合工具自动嵌入到系统中，设计者可以不去考虑具体的总线时序和管脚映射等细节，能够迅速更换类似的IP模块。另外的一个好处是，可以进行模块之间的通信细化，把模块之间的抽象通道利用一个更加详细、更加底层的模型来替换，而不影响模块本身。例如，我们可以在功能级定义一个抽象的通信协议，确保无误后，将其细化为一个FIFO通信链路或者一个总线周期精确级的通信链路。这种设计方法学可以很容易地实现模块化设计和设计重用。

### 利用总线协议进行通信细化

当把一个抽象功能级的设计细化到总线周期精确或者周期精确硬件抽象级时，需要做以下转化：

1、功能级的串行通信信道利用总线协议细化到并行的通信信道。

2、功能级的进程通过引入时钟和置位信号细化为同步的并行进程。

信道细化不仅仅是利用总线协议信道替换一个功能级的信道。对于简单的点对点通信，可以使用简单的替换。但复杂的多点信道并不适用。复杂信道的细化需要进行一些设计，可以手工进行也可以使用接口综合工具来设计地址译码器、多路器、桥和协议适配器等。一般有两种通道细化的方式：

1、模块细化：互相通信的模块通过引入总线协议而完成细化

2、通道细化：信道通过引入总线协议完成细化，而模块保持不变

### 模块细化

功能级的通信被细化为一个全握手总线协议的主要内容如下：

1、主端口和从端口被细化为总线端口，细化的方法是给他们指定一个总线协议模版参数

2、producer和consumer进程细化为带时钟的进程，它们实现了全握手总线协议

当端口指定了总线协议后，它就被称为总线端口(bus port)，因为它具有端口终端，例如request、acknowledge和data等物理层的总线信号。这些信号成为终端(terminal)，这些终端实现了总线协议信号关系。他们与具体协议有关，在总线类中定义。

### 通道细化

在通道中插入协议转换模块完成所有的细化工作。这种方法允许通信和行为的分离。

# TLM设计实例

在片上系统尤其是嵌入式系统中，片上总线几乎是必须的。典型的片上总线包括AMBA等

## 片上总线系统概念

对于包括总线、总线仲裁器、数字信号处理器、微处理器、存储器和其他专用集成电路这样一个系统。如果集成到一个片内，那么总线就变成了片上总线。对于这样一个复杂系统同时包括软件和硬件，一个传统的办法是全部用C/C++进行描述以进行系统级验证，然后将硬件部分的描述手工翻译为硬件描述语言进行描述。等硬件全部实现后再进行软件实现。

使用systemc作为建模语言的情况下，整个系统可以方便地用一种语言描述，然后细化到最终实现。

一个片上总线系统从逻辑上看包括总线主设备、从设备、仲裁器和总线本身。典型的主设备包括处理器、DMA控制器等，典型的从设备包括存储器、中断控制器、GPIO和UART等。

# Systemc与传统硬件描述语言VHDL/Verilog HDL的比较

## systemc与传统硬件描述语言的关系

随着集成电路制造技术按照摩尔定律的快速发展，人们越来越感觉到传统的硬件描述语言VHDL/Verilog HDL已经不能够满足越来越复杂的设计需求，就像当初的小规模集成电路向大规模集成电路过度一样，当时产生了VHDL和VerilogHDL。如今，超大规模集成电路已经不能够表达集成电路的复杂度。

关于门级和寄存器传输级，多数人比较清楚，但对于系统级这个概念可能比较模糊，可以统一地理解为比RTL更抽象的描述级别。

一般认为，Verilog HDL的RTL描述能力最强，VHDL和SystemC次之，所以对于设计不是很复杂的小规模电路，SystemC不是最佳选择；而对于系统描述能力，SystemC最佳，VHDL次之，Verilog HDL再次之，因此要设计复杂的片上系统，systemc是最佳选择。一个典型的片上系统可能包括嵌入式处理器、存储管理单元、通用串口、I/O、其他外设、片上总线以及操作系统。

对于这样一个系统，真正复杂的不是如何完成它的硬件部分的RTL描述，而是如何确定系统的资源如存储器和FIFO大小等，以及如何实现软硬件并行设计和协同仿真，以减小开发周期。

# 基于systemc的验证方法学

## systemc验证标准

现代集成电路制造工艺技术的发展和设计复杂度的不断提高对集成电路设计的验证提出了更高的要求。在典型的设计中，验证通常只会占用整个设计周期的三分之二左右的时间。片上系统集成使得统一块芯片上可能集成非常复杂的系统。从而要求验证的级别从传统的电路级和RTL级提高到系统级以加速验证收敛。在集成电路相关学科中，验证和测试是两个涵义不同的专有名词，验证指的是流片以前的基于软件的仿真，而测试专指流片以后对裸片或者封装后的芯片的性能的检查。

通常的设计流程一般包括系统结构设计、子模块设计、子模块验证和系统级验证。系统级的验证在设计的最后完成，因为只有各个系统子模块的结构和功能设计完成后，整个系统的验证才能进行。验证通常针对设计的关键路径进行，如果达不到设计要求，就需要对设计进行部分修改。此时如果再对系统结构进行修改，就要重新进行子模块设计等后续工作，通常工作量巨大。

采用systemc的设计流程的系统级验证可以在整个项目的生命周期内进行。同一个验证组件能够在模块验证和结构探索、优化中进行重用。这个流程中子模块的验证开始较早，可以在系统结构设计的进行中就开始。采用systemc的系统设计流程可以提高代码的利用率，保证产品的上市时间。

systemc成立了systemc验证工作组(VWG)，提出systemc的验证标准(SCV)。SCV的主要内容是systemc验证库，它包括支持验证的一系列类库。

SCV为构建高级的systemc可重用的验证IP提供了主要的语言功能。包括：

1、数据内查

2、随机数产生和随即种子管理

3、约束的随机数产生

4、带权重的随机数产生

5、交易监视和记录

6、稀疏矩阵支持

除了上述各项，SCV还包括了一些对编写验证平台非常有用的语言功能，其中有的对验证平台建模是必须的：

1、到其他硬件描述语言的接口

2、常见数据结构的集合

3、异常处理

4、调试

## systemc验证方法学相关术语

验证平台(testbench) ：对设计进行验证的代码或者模块，凡是为了验证某一设计模块而编写的所有相

关 代码都可以归为验证平台。

交易(transaction) ：交易代表了一组信息，用来表示模块间的一些特定行为。一个交易具有开始时

间，结束时间和一组属性。

交易器(transactor) ：交易级的验证和被验证模块(他们通常在不同抽象级别)之间的适配器，基于交

易的systemc验证模型也被称为交易验证模型(transaction verification

model, TVM)

交易器接口 ：用于验证的模块和交易器的接口

端口接口 ：交易器和被验证设计进行通信的端口

交易器方法 ：在交易器中定义的方法

交易记录 ：将交易信息存储至数据库，用作仿真后分析的进程

自动交易记录 ：一种交易器建模的方法，交易器能够自动地将交易记录下来，而不需要在交易

器的代码中嵌入交易记录的代码

交易流，

交易级测试，

数据内查，

接口内查，

约束，

声明，

暂态约束

## systemc的验证标准

systemc验证标准提供一些应用程序接口(API)，增强了验证的能力。这些API支持基于交易的验证，可约束的随机数产生，异常处理等验证任务。通过引入普通的数据内查能力，SCV能够以统一的方式操作任意的数据类型，包括C/C++内建类型，systemc内建类型、用户自定义的组合类型和枚举类型。这样在变量记录、交易记录、约束和其余功能中可以使用任意的数据类型。

### 交易级建模的风格

基于交易的验证是利用systemc进行功能验证的主要策略。交易级的验证平台的好处是，既能够对用户定义的通道进行TLM仿真，也能够进行RTL仿真，从而体现了“设计复用”的基本思想。

### 动态并发性建模

通常，一个设计利用静态的并发性行为来对硬件的并发性进行建模，然而验证平台并不需要静态并发性，而采用动态并发性。测试器能够利用动态线程来产生并发行为。虽然动态线程操作的API并不是SCV标准的一部分，但它对于验证平台的建模非常重要。