通信IC设计笔记

介绍

第1章,主要介绍集成电路设计入门,重点讲述 Verilog HDL设计入门、各种设计思想以及设计规范。

第2章,主要面向FPGA开发者,讲述FPGA与ASC设计的不同,并详细讲述FPGA的各种特色应用,ASIC开发者也可以从中借鉴很多很好的FPGA设计理念。

1. 集成电路设计与HDL

本章的内容安排如下:

1.1节从总体上论述什么是IC设计,IC设计的流程以及整体规划的重要性,并强调面向验证和软硬件结合的设计原则。

1.2节简要介绍 Verilog语法,并通过8个例子讲述如何通过简单的HDL语句描述复杂的硬件电路。本节主要强调用HDL给出的描述需要有具体对应的基础电路原型,并明确时序的 概念。此外还给出各种常用的验证小技巧帮助读者快速进行验证设计。

1.3节则是对第2节设计方法的一个总结和提炼。本节主要强调必须按照结构化方式设计电路,并需要遵循常规思维模式对复杂模块进行解耦,按照控制与数据分离的原则简化设计。 此外,还讲述了芯片中控制流与数据流的设计方法。

1.4节则针对IC中的定点化进行论述,强调定点化的实质是人与机器的理解如何达到一致。

1.5节给出HDL语言描述规范,并强调遵循规范实质上是一种IC设计模式,能够有效地增加设计成功率。

1.6节则引入吞吐率、电路时序等关键性概念,给出各种电路思想,并介绍四种经典电路优化方法:重定时、折叠、展开与脉动阵列。

1.1 集成电路设计基础

1.1.1 集成电路概念

1.2 Verilog HDL 快速入门

1.2.1 Verilog HDL简介

Verilog HDL诞生于1983年，由Gateway Design Automation公司发明，由于其简洁易用且集成电路设计与HDL17与C语言语法类似，逐渐为公众所接受，并于1995年成为IEEE标准，被称为IEEEStdl364-1995，即Verilog-1995。

但在随后的使用中，设计人员发现Verilog-1995版本存在许多可以改进的地方。为了解决用户在使用Verilog-1995过程中遇到的问题，标准化组织对Verilog-1995进行了修正和扩展，并发展成为IEEE1364-2001标准，即Verilog-2001。Verilog-2001是针对Verilog-1995的一个重大改进版本，它具备很多新的实用功能，例如通用敏感列表、多维数组、生成语句块、命名端口连接等。目前，Verilog-2001是Verilog的主流版本，基本上所有的EDA工具均支持这一版本。

2005年，Verilog再次进行更新，即IEEE1364-2005标准。该版本只是对上一版本的细微修正。这个版本包括了一个相对独立的新部分，即Verilog-AMS。这个扩展使得传统的Verilog可以对模拟和混合信号系统进行建模。

2009年，IEEE 1364-2005和IEEE1800-2005两个部分合并为IEEE1800-2009，成为了一个新的、统一的SystemVerilog硬件描述验证语言（Hardware Description and Verification Language，HDVL）。该版本于2012年进行了修订，被称为SystemVerilog-2012。

2014年后的主流EDA商业软件（包括ASIC/FPGA）均支持本节所提到的所有Verilog版本，建议读者以Verilog-2001为基础进行学习，有条件的读者则可以从SystemVerilog-2012着手，结合UVM/VMM开展IC项目设计与验证。

1.2.2 Verilog的表达能力

Verilog能够支持以下不同级别的电路描述,读者可以根据应用场景的不同进行选择:

- 系统级(System):该级别基本属于电路行为级描述,通常用于仿真和高层建模,属于不可综合的范畴;但现在有部分高级综合工具在逐步支持系统级综合。此外,系统级 一般采用 Matlab或者C/C++进行描述。
- 算法级(Algorithm):这一级别重点关注各类复杂算法的实现,不关心具体的实现时序;而本书的描述重点就是将各种算法通过简单的 Verilog代码描述出来,并在此基础 上添加各种时序控制,最终完成算法级描述到RTL级描述的转换。目前很多 Verilog工具或者其他语言综合工具都能够支持基于算法级的描述,并能直接生成RTL代码或者最终的门级电路网表。同样,很多算法级描述采用 Matlab或C/C++实现。
- RTL级(Register Transfer Level):这一级主要描述电路中的寄存器行为,以及寄存器之 间的组合电路是如何实现的。RTL级的描述是现在HDL设计的主战场,基本上所有的 成熟数字电路设计都是以RTL代码形式存在的。
- 门级(ate-Level-):用于描述逻辑门以及逻辑门之间的连接模型,逻辑综合器的一个典型作用就是将RTL代码转换为门级网表,以形成基本的元器件电路。
- 开关级(Switch-Level-):用于描述器件中三极管和储存节点以及它们之间的连接模型,通常布局布线的结果就是形成开关级电路。

通常在进行大型芯片项目设计时，往往会完整经历系统级设计、算法级设计和RTL级、门级电路设计。通过系统级设计保证正确的软硬件划分；通过算法级的建模来保证设计的系统性能；在算法级验证通过后，再通过在算法中添加时序的方法，将高级语言转化为RTL代码；当RTL代码完成验证后，就进入门级电路设计。上述层次之间的关系如图1-12所示

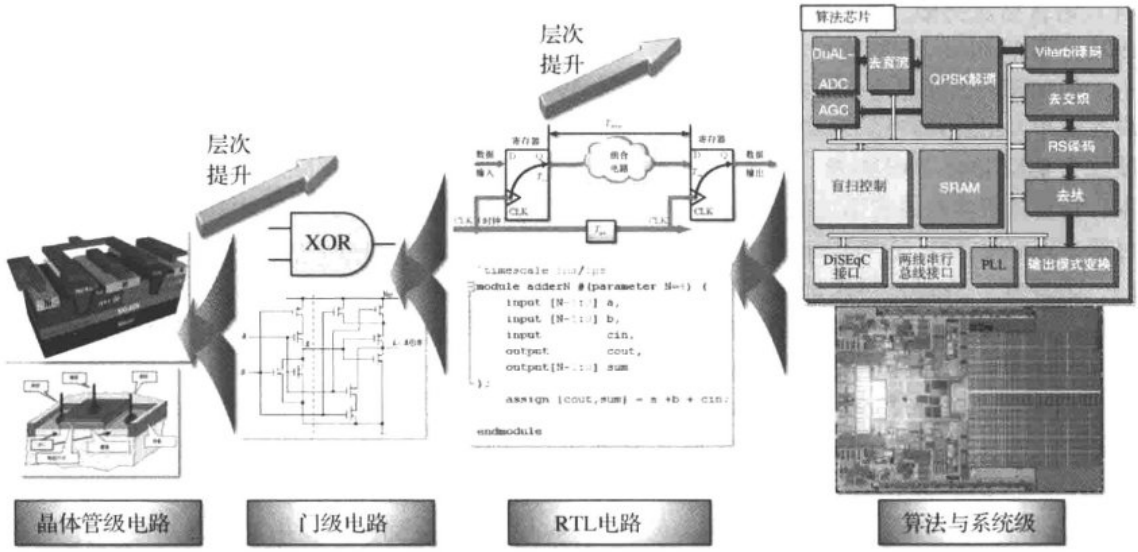


图 1-12　Verilog 描述层次之间的关系

1.2.3 第一个Verilog程序：通用加法器

Verilog作为一门硬件电路描述语言，一种快速熟练掌握它的方法就是不断练习，反复动手实践，通过例子掌握隐藏在语句背后的硬件电路。因此本书通过很多典型例子，由浅入深，快速引导读者掌握Verilog的精髓。

下面是第一个需要学习的Verilog例子，学习该例子的目的是让读者掌握电路模块module的概念。

```
module adderN #(parameter N=4)(
  input [N-1:0] a,
  input [N-1:0] b,
  input cin,
  output cout,
  output [N-1:0] sum
);
  assign {cout,sum} = a + b + cin;
endmodule
```

该例描述了一个4位加法器，从例子可看出整个模块是以module开始，endmodule结束。只有以module开始，以endmodule结束的描述才能被认为是一个完整的电路描述，其余都只能认为是电路片段。一个模块就是一个完整电路，如果有N个模块，这N个模块将通过某种机制结合起来，组成一个大的电路，但这N个模块都是独立运行的，而且是并行执行的。

上面代码的输入(定义为input)为a、b和cin，注意a、b前面有一个[N-1：0]，这表示a与b的位宽为N，最高位为第N-1位，最低位为0位；而cin前面没有位宽设定，则表示1位。对于任意有意义的逻辑变量（wire/reg，输入输出端口等），必须指定其位宽，如果没有显式指定，位宽就默认为1位。

该段代码的输出（定义为output)为sum和cout，其中，sum为标准加法和，位宽为N位，而cout则是1位变量，表示加法进位位。

除了单方向的输入input和输出output外，电路还存在双向输入输出，因此Verilog采用inout命名这类端口。

通用的module模块描述方法如图1-13所示。

图 1-13module 的基本语法结构

该模块隐含以下几个常用的语法点。

1. parameter N=4

parameter类型用来定义模块的全局符号常量，等效于C语言的参数设定。这是一种对模块进行抽象描述的重要手段，通过简单修改parameter设定，就能给出不同的电路模型。pa-rameter的主要目的是提高程序的可读性和可维护性，在对模型IP化时，会大量用到parame-ter型变量。另外，还有一个类似的符号常量localparam，该类型定义与parameter的区别在于，parameter可以被外部调用模块改变，相当于C/C++语言中传值的概念，而localparam仅在本模块内部有效，主要目的是为了提高本模块的可读性。

在本段代码中可以通过调整参数N实现任意位宽的加法器。

2. assign

assign语句是指连续赋值的意思。加法器是一个组合电路，当输入变化时，输出一定立刻响应，而且这个过程是持续不断的。因此Verilog专门设计assign语句以指定这类连续过程，而不是采用传统的赋值符号“=”。可以认为assign就是描述组合电路的专用语句。

3. {cout,sum}

上述描述方法是一种对两个变量合并赋值的简化描述，HDL语法解析器和综合器将按照如下的理解对上述描述进行解析，并生成最终电路。

```
wire[N:0] adder_temp;
assign cout = adder_temp[N];
assign sum[N-1:0] adder_temp[N-1:0];
```

4. a+b+cin的位宽匹配

这3个数位宽不同，但在Verilog语法中允许这三者直接相加。默认的规则为：算术表达式结果的长度由最长的操作数决定。在赋值语句下，算术操作结果的长度由操作符左端的目标长度决定。表达式中的所有中间结果应取最大操作数的长度，如果这个中间结果会赋值给左端的表达式，则最大操作数长度也包括左端目标长度。

因此a+b+cin的操作规则为：①如果不赋值给某个数，例如！（a+b+cin），则操作位宽在a/b/cin中选择最大值；②如果赋值给某个数dout，例如dout=(a+b+cin)，则操作位宽在a/b/cin/dout中选择最大值。

在这条语句中，a/b/cin的最大位宽为4，但由于cout，sum！表示5位的数据，所以加法器的代码等效于：

```
a+b+cin = {a[N-1], a[N-1:0]} + {b[N-1],b[N-1:0]} + cin
```

一般的加法器会产生1位的进位,因此加法器的输出结果通常需要比输入位宽多1位,这是算法电路设计中的一种标准写法。

例子小结

读者在理解 module 语法的基础上，需要理解 module 模块是整个电路描述中的最基本单元;任何复杂的 电路都是通过多个简单的 module 搭建而成的。任何一个模块都是独立的存在，各个模块都是并行运行的。

此外还需要理解，Verilog 的描述基础是 bit，任何一个运算或表达式都是基于 bit 进行表达的;在此基础上再去理解多个不同 bit 位宽的数据运算时，如何进行 bit 位对齐，中间结果的位宽如何取值，赋值的结果 位宽如何设定等。

为方便读者查阅 Verilog 语法，本书在附录中包含有标准的 Verilog 语法说明材料。欢迎读者在阅读例子 时对比参照，从而真正了解语法背后所要表达的思想。

1.2.4 第二个verilog程序：多路选择与运算操作

本节需要描述的电路结构如图 1-14 所示，该电路是一个标准的四选一选择器，输入 bit 的位宽通过参数 N 指定，默认为 2bit，由控制选择信号 sel 决定选择哪一个结果输出，并将结果赋予 mux_out。

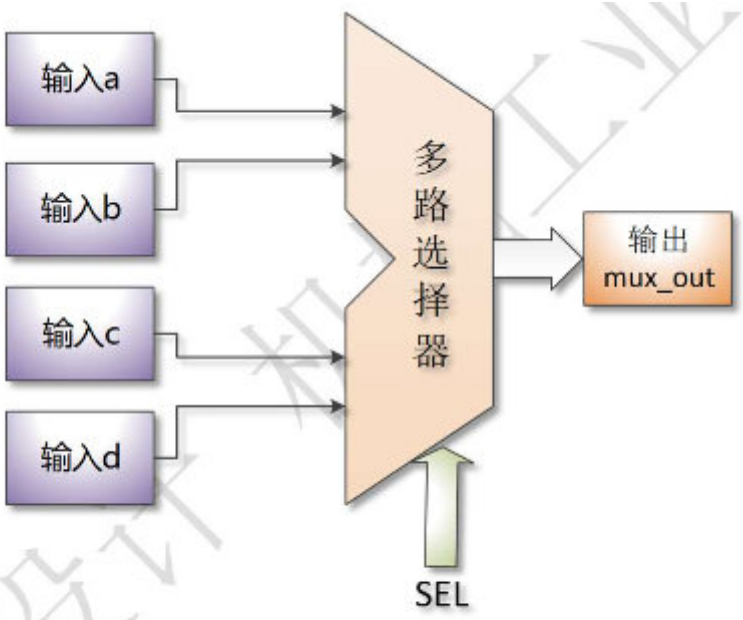


图 1-14 多路选择器与 Verilog 描述

该电路的 Verilog 描述代码如下：

```
module mux #(parameter N=2)(
  input [N-1:0] a, // sel=00时，选择该输入
  input [N-1:0] b, // sel=01时，选择该输入
  input [N-1:0] c, // sel=10时，选择该输入
  input [N-1:0] d, // sel=11时，选择该输入
  input [1:0] sel, //选择器
  output[N-1:0] mux_out);// 选择器结果输出
  reg [N-1:0] mux_temp; // 临时变量，用于防止其他调用者误认为输出锁存 assign
mux_out=mux_temp;

  //always_comb //该语句在systemverilog中可以替换下面的语句并检查 always @ (a or b or
c or d or sel)
  case (sel)
    0 : mux_temp = a;
    1 : mux_temp = b;
    2 : mux_temp = c;
    3 : mux_temp = d;
    default : $display("Error with sel signal");
endcase
endmodule
```

上述代码包含三个语法点:(1)$display 语句，该语句与 C 语言中的打印语句非常类似，能够在仿真时 输出各类中间变量信号，类似语句还有$fprintf 等; (2)利用 case 语句进行分支选择; (3)通过 always 语句描述组合电路实现。

1. Case语句语法

case 语句是一个多路条件分支选择的描述，其语法如下:

```
case (case_expr)
case_item_expr{ ,case_item_expr} : procedural_statement
  ...
    ...
[default:procedural_statement]
endcase
```

case 语句首先对条件表达式 case_expr 求值，然后依次对各分支项求值并进行比较，第一个与条件表达 式值相匹配的分支中的语句被执行。可以在 1 个分支中定义多个分支项;这些值不需要互斥。缺省分支覆 盖所有没有被分支表达式覆盖的其他分支。case 的缺省项必须给出写，这样能够防止产生锁存器电路。

利用 case 语句描述组合电路时，必须提供所有的选择结果情形，否则 HDL 综合器将根据描述，自动生成锁存电路。而锁存电路(latch)路，相对组合电路，会带来额外时钟延迟，并引入异步时序，不适合常规 电路设计。此外，Latch 在可测试性设计中，需要额外的逻辑实现。因此，建议读者在描述电路时，一定注 意不要编写不完全分支的电路语句，以免生成锁存 Latch。

2. always语句语法

在 Verilog 电路描述中，always 语句是表明所描述的电路行为将被重复执行，而执行机制是通过对一个称 为敏感变量表的事件驱动来实现。

可综合的 always 语句只有两种写法，第一种为电平事件驱动，当敏感事件列表中的信号发生变化，就执 行内部的语句块，代码书写方式如下:

```
always @ (敏感事件列表)
  //always  内部执行语句快
always @ (a or b or c or d or sel)//abcd sel均为敏感信号列表
```

第二种写法为边沿除法驱动，当敏感信号发生跳变时，就执行内部的语句块，典型的代码书写方式如下:

```
always @ (posedge clock )
always @ (posedge clock or posedge reset )
```

上述语句中的`posedge` 均可替换为`negedge` 。其中 表示`posedge`上升沿触发，而`negedge` 表示下降沿触发。该语句表达的含义就是当 clock 信号跳变瞬间，执行语句块的内容。

而对于没有提供敏感事件列表的语句，则是无条件执行，通常用于测试验证代码，属于不可综合。诸如下面的语句表示总是对 clock 信号取反，而这个取反过程会延迟 10ns，因此会产生一个 20ns 为周期的 clock 方波。

```
always #110 clock = ~clock;
```

初学者只需要记住，第一种`always`写法是组合电路的描述方法，而第二种`always`写法则是时序电路的描述方法。`Always` 语句既可以描述组合电路，也可以描述时序电路，而 reg 并不都是寄存器类型，也有 可能为连线。对于 reg 类型不能准确定义为寄存器，而是有可能为锁存器或者组合电路中的临时连线，这对 Verilog 语言设计而言，可以认为是一个较大缺憾，因此在后来的 SystemVerilog 语言中，通过引入 `always_comb` 、 `always_latch`、`always_ff` 来弥补 reg 以及 always 语句描述能力不足的问题。例如:

```
always_comb #1ns a = b & c; //组合电路写法
always_latch if(ck) q <= d; //锁存器电路写法
//下面内容为标准的时序可综合电路写法
always_ff@(posedge clock iff reset == 0 or posedge reset) begin
    r1 <= reset ? 0 : r2 + 1;
  ...
end
```

1. **always语句的并行性**

在逻辑电路中，各个硬件都是并行执行的。站在电路角度讲，所有的 D 触发器都是并行执行的，而驱动源就是时钟。因此在 HDL 描述中，每一个带时钟的 always 语句块都是并发执行的。例如下面的例子，out1 赋值和 out2 赋值是同时的，没有先后顺序。

```
always@(posedge clk)
  begin
        out1<=in1;
  end
always@(posedge clk)
  begin
        out2<=in2;
  end
```

但在 always 语句块内部，任何一个语句块(以 begin 开始，end 结束)都是串行执行的，只是存在赋值立刻生效还是事后生效的差异，即后面将要重点论述的阻塞赋值和非阻塞赋值两种区别。

```
always@(*)
  begin
    temp=b;
    b=a;
    a=temp;
  end
```

上面的例子，就是一个串行执行的例子，能够完成 a 与 b 的数值交换，如果不是串行执行，上述代码将很难完成类似各类程序控制。

2. **always 内部的控制流**

目前所有的控制结构可以概括为三种:(1)顺序结构;(2)分支选择结构;(3)循环结构。而这些控制 流在 always 语句内部均能实现。

顺序结构就是标准的顺序执行，这在 always 语句块内部已经天然实现。分支选择结构目前有两类语句支 持，if-else 语句和 case 语句。If-else 语句的语法为:

```
if(condition_1)
  procedural_statement_1
{else if(condition_2)
  procedural_statement_2}
{else
  procedural_statement_3}
```

如果对 condition_1 求值的结果为个非零值，procedural_statement_1 将被执，如果 condition_1 的值为 0、 x 或 z，则 procedural_statement_1 不被执行。如果存在一个 else 分支，那么这个分支被执行。具体例子如下:

```
if(a>b) sel_out<=in1;
else if(a==b) sel_out<=in2;
else sel_out<=in3;
```

注意，`if`语句与`case`语句一样，在描述组合电路时，一定要给出全部分支情况，否则容易生成锁存 Latch 电路。

对于循环语句，主要包含`for、while、forever、repeat`四类语句，但只有`for`语句才有可能具备可综合性，其余均为测试验证所准备。

循环语句 for 的语法为:

```
for(表达式 1;表达式 2;表达式 3)语句
```

其实可以将 for 语句理解为:

```
fox(循环变量赋初值:循环结束条件;循环变量增值)执行语句
```

for循环的例子如下，这是最原始的一个 8bit 乘法器实现，其中表示左移，等效于乘以 2 的移位次方:

```
parameter size = 8;
reg[size-1:0] opa, opb;
reg[2*size-1:0] result;
integer bindex;
result=0;
for( bindex=0; bindex<=size-1; bindex=bindex+1 )
  if(opb[bindex]) result = result + (opa<<(bindex));
end
```

读者在检查逻辑综合结果后很容易发现，该代码实现其实是一个全并行的加法器，for 语句等效于将以下语句完全展开:

```
if(opb[bindex]) result = result +(opa<<(bindex));
```

所以读者在编写 for 语句时，一定要理解这是一个并行完全展开语句，而不是串行多周期执行。

3. 算术逻辑单元例子

下面的代码，通过 case 语句给出了一个最简单 ALU(算术逻辑单元)的描述方法，方便大家加深印象:

```
module alu #(parameter N=8)(
    input [ 2:0] opcode, //操作码
    input [N-1:0] a,             //操作数
  input [N-1:0] b,           //操作数
    output reg[N-1:0] out  //操作结果
);
localparam add =3'd0;
localparam minus=3'd1;
localparam band =3'd2;
localparam bor =3'd3;
localparam bnot =3'd4;
always@(opcode or a or b)//电平敏感的 always 块
  case(opcode)
    add: out = a + b; //加操作
```

```
    minus: out = a - b; //减操作
    band: out = a & b; //求与
    bor: out = a | b; //求或
    bnot: out = ~a ; //求反
    default:out = 8'hx ; //未收到指令时，输出不定态
 endcase
endmodule
```

这个例子同样是一个组合电路，总共有 8 种选择，但电路只明确给出前 5 种选择的结果，其余都通过缺省语句设置为不定态结果。

这个例子引入了两个语法:(1)localparam 用于指定本地参数,主要是取代传统的 define 宏定义，因为 `define 一旦定义后，会全局有效，必须通过`undef 配套才能使宏定义失效。新版本的 verilog 所以推荐 localparam 用于设定局部宏参数。(2)引入多个运算操作符。

该例子还引入了基数表示法。我们知道，自然数通常是十进制表示，例如16，-32等等，Verilog支持给变量或信号赋值为自然数，但自然数通常都是默认32位整数，赋予某个变量后会产生各种截断处理，而且在操作时需要十进制转换为二进制，总之就是不够精确和灵活。因此，Verilog最常用的是基数表示法，例如`3'b011，4'd5，5'o27，6'h3f，`就分别表示3bit二进制011，4bit十进制5(为二进制101)，5bit八进制27(为二进制10111)，6bit十六进制3f(为二进制111111)。基数表示法的通用表示为:

```
[size ]'base value
```

其中，size 定义以 bit 位计的常量的位长;base 为 o 或 O(表示八进制)，b 或 B(表示二进制)，d 或 D (表示十进制)，h 或 H(表示十六进制)之一;value 是基于 base 的值的数字序列。值 x 和 z 以及十六进制 中的 a 到 f 不区分大小写。

基数格式的长度定义(size)是可选的。如果没有定义一个整数型的长度，数的长度为相应值中定义的位数。如果定义的长度比为常量指定的长度长，通常在左边填 0 补位。但是如果数最左边一位为 x 或 z，就相应地用 x 或 z 在左边补位。此外，如果长度定义得更小，那么最左边的位相应地被截断。下面是一系列的例子:

表 1-2　Verilog 数制举例

用例	说明
'hAE	8 位十六进制数
10'b10	左边添 0 占位，实际为 10'b0000000010
10'bx1x0	左边添 x 占位，实际为 10'bxxxxxxx1x0
3'b1001_0011	3'b011 相等

4. **运算操作符**

读者很容易发现前面的例子中的运算操作符与 C 语言中的定义完全一致。实际上，Verilog 正是从 C 中借 鉴了各种运算符。Verilog 的各类操作运算符定义如下:

表 1-3　Verilog 操作运算符定义

运算类别	符号	运算符含义
算术运算符	+	加法（二元运算符）
	-	减法（二元运算符）
	*	乘法（二元运算符）
	/	除法（二元运算符）
	%	取模（二元运算符）
关系运算符	>	大于
	<	小于
	>=	不小于
	<=	不大于
	==	逻辑相等
	! =	逻辑不等
逻辑运算符	&&	逻辑与
	||	逻辑或
	!	逻辑非
按位逻辑运算符	~	一元非，相当于非门运算
	&	二元与，相当于与门运算
	|	二元或，相当于或门运算
	^	二元异或，相当于异或门运算
	~^、^~	二元异或非即同或，相当于同或门运算
移位运算符	>>	右移
	<<	左移
赋值运算符	=	阻塞赋值，等效于立即生效
	<=	非阻塞赋值，等效于当前模块结束后赋值，或者下个时钟周期赋值生效
缩减运算符	&	一元与，相当于数据位逐个进行与操作
	|	一元或，相当于数据位逐个进行或操作
	^	一元异或，相当于数据位逐个进行异或操作
	~^	一元同或，相当于数据位逐个进行同或操作

上述表达式包含三类运算符：

1. 单元运算符:可以带一个操作数,操作数放在运算符的右边。
2. 二元运算符:可以带二个操作数,操作数放在运算符的两边。
3. 三元运算符:可以带三个操作数,这三个数用三目运算符分隔开。

例如:

```
clock=~clock;// ~是一个单元取反运算符,clock是操作数。
c=a&b; // &是一个二元按位与运算符,a和b是操作数。
r=sel ? a:b; // ?:是一个三元条件运算符,sel,a,b是操作数。
```

此外，为方便电路描述，使整体代码简洁，Verilog 相对 C 语言引入了具备电路特色的缩减运算符和连接 运算符。

缩减运算符是对单个操作数进行或与非递推运算,最后的运算结果是一位的二进制数。缩减运算符目前支持或与非三种操作。具体运算过程如下:第一步先将操作数的第一位与第二位进行或与非运算,第二步将运算结果与第三位进行或与非运算,依次类推,直至最后一位。例如:

```
wire[3:0] A;
wire B;
assign B =&A;
```

等效于以下语句:

```
assign B =((A[0]&A[1])& A[2])& A[3];
```

拼接运算符则与缩减运算符相反，主要目的是将两个或多个信号的某些位拼接起来进行运算操作。拼接运算不消耗任何逻辑资源，只是一个单纯的连线逻辑。其使用方法如下:

```
{信号1的某几位，信号2的某几位，..,..,信号n的某几位}
```

即把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。例如:

```
{a,b[3:0],c,3'b101}
```

也可以写成如下形式:

```
{a,b[3],b[2],b[1],b[0],c,1'b1,1'b0,1'b1}
```

在位拼接表达式中不允许存在没有指明位数的信号。这是因为在计算拼接信号的位宽的大小时必需知道其中每个信号的位宽。位拼接还可以用重复法来简化表达式，例如:

```
{6{a}}//这等同于{a,a,a,a,a,a,a}，a可为任意比特位宽
```

位拼接还可以用嵌套的方式来表达，例如:

```
{c,{3{a,b}}}//这等同于{c,a,b,a,b,a,b}
```

用于表示重复的表达式如上例中的 6 和 3，必须是常数表达式或者参数。

Verilog 在设计之初，就是考虑采用相同的 C 语言语法吸引大部分逻辑硬件设计者，同时在演进过程中也 同步引入 C 语言新版本的新特性，例如读者可以发现 verilog-2001 新的模块定义方法与 C99 语法非常类似。 而 SystemVerilog-2012 则大量引入 C++面向对象的特性，硬件接口和对象均具备抽象机制。

5. 运算符的优先级

所有的运算符都是有优先级的。逻辑综合器所理解的优先级如下:

表 1-4　逻辑综合器所理解的优先级

优先级（由高到低）	运算符
1	! ~
2	*/%
3	+-<<>>
4	<<=>>=
5	==! ====! ==
6	&
7	^ ^~
8	|
9	&&
10	||
11	?:

如果读者描述的优先级与综合器和仿真器理解的不一致，就会出现错误的结果。所以在进行描述复杂语句时，一定多次确认优先级，如果有疑惑可以通过括号()指定运算规则。

6. 例子小结

本例引入三个特性:(1)case 语句的特性;(2)数制的表示方法;(3)各类运算符。读者需要注意 case 必须添加 default 选项，并了解运算符的计算规则和优先级，尤其记住拼接运算符和缩减运算符的用法。

1.2.5 第三个 Verilog 程序:D 触发器和多路延迟

前面两个例子均为组合电路，下面将介绍时序电路的描述方法。

```
module DFF(
  input D,
  input CLK,
  output reg Q
);
    always@(posedge CLK)
        Q <= D;
endmodule
```

上述代码描述了一个最简单的 D 触发器:输入数据 D，下一个时钟周期就输出 Q。其中数据的存储时刻是在时钟 CLK 信号的上升沿(跳变瞬间)，即寄存器 Q 在时钟上升沿采样输入数据 D，并更新存储内容;除时钟 CLK 上升沿的其余时刻，无论 D 如何变化，都不会存储到 Q 寄存器中。只要理解这一点，就明白了数字电路进行时序电路控制的基础。

该代码包含了一个语法点:非阻塞赋值(<=)。阻塞赋值与非阻塞赋值是 Verilog 中专门针对电路行为而 特殊设计的赋值语句。因为对于组合电路而言，信号传递过来是立刻生效，而且生效的值也将继续传递，直 到某个寄存器或存储单元终结;而对 D 触发器或者寄存器而言，对其任何赋予数值，只要不是在触发沿(依 赖于定义的是上升沿还是下降沿触发)，都是无效的，而且输出的值也是一直保持稳定不变，直到新的时钟 沿到来并更新存储的数据。因此对于寄存器的赋值应当是非阻塞赋值(<=)，因为寄存器采样数据发生在当前 模块执行完毕(通常是 always 语句执行完毕)后。而对于组合电路赋值则应当采用阻塞赋值(=)。

在 Verilog 中，根据电路描述的行为特征，定义了两类数据类型:线网类型(net type)和寄存器类型(reg type)。

1. 线网类型

线网类型主要有 wire 和 tri 两种。线网类型用于对结构化器件之间的物理连线的建模。如器件的引脚，内部器件如与非门的输出等。

由于线网类型代表的是物理连接线，因此它不存贮逻辑值。必须由器件所驱动。通常由 assign 进行赋值。如 assign A = B + C;当一个 wire 类型的信号没有被驱动时，缺省值为 Z(高阻)。

当信号没有定义数据类型时，缺省为 wire 类型。因此前面的所有 module 端口没有定义数据类型的，全 部都默认为 wire 线网类型。所以通用加法器例子中的输出赋值采用 assign 语句赋值。

对于 tri 类型而言，主要用于定义三态的线网，由于该语句属于不可综合，因此只在仿真测试或功能模型 中使用。

2. 寄存器类型

reg 类型定义语法如下:

```
reg[msb:lsb] reg1, reg2, ...,regN;
```

msb 和 lsb 定义了范围，并且均为常数值表达式。范围定义省值为 1 位寄存器。例如:

```
reg[3:0] count;// 4 bit寄存器
reg flag;//1 位寄存器
```

此外还有一种专门针对存储器模型(RAM)的定义方法，例如:

```
(* ramstyle ="MLAB"*)reg[31:0] RegFile1[15:0];
(* ramstyle ="MLAB"*)reg[31:0] RegFile2[15:0];
```

在 ASIC 设计中，这种描述方式只会被识别为一系列的寄存器堆，并不会被识别为 RAM;ASIC 中应当用 RAM 单元库(IP)例化的方法描述 RAM。而在 FPGA 中，综合器首先将这种描述识别为 RAM 的声 明，并通过识别对象的行为确认描述对象是 RAM 还是寄存器堆。如果后续的描述行为满足 RAM 的特征， 就自动替换为 FPGA 内部内置的 RAM 单元库，否则将识别为寄存器堆。

上例的 RegFile1 与 RegFile2 对象在 Altera FPGA 中，将被识别为 16 个 32bit 位宽的 RAM，而且指定为 MLAB 类型。

需要注意的是，reg 类型的数据只能在 always 语句块中赋值。

3. 带复位的触发器

这个例子是带异步复位的触发器，其中 reset 是低电平有效

```
module DFF1(
  input clk,
  input reset,
  input d,
  output reg q
);
// negedge 表示复位信号低电平有,
  always@(posedge clk or negedge reset)
    if(!reset)
      q <= 0; //异步清 0，低电平有效
    else
      q <= d;
endmodule
```

如果是同步复位的触发器则应当修改为:

```
module DFF1(
  input clk,
  input reset,
  input d,
  output reg q
);
    always@(posedge clk) //同步复位无需添加到敏感列表中。
    if(!reset)
      q <= 0; //同步清 0，低电平有效
    else
      q <= d;
endmodule
```

读者应当很容易找出同步复位与异步复位的区别就是在于:always 语句的敏感列表中是否添加了复位信号。

4. 锁存器latch

前面已经反复谈到，锁存器在同步电路中应当尽量避免，除非做专门的特殊处理，但锁存器还是有一定的优点。因此在此处简单介绍一下锁存器，并描述一下锁存器是如何编码生成的。下面是一个电平敏感型锁存器结构:

```
module latch_N#(parameter N=1)(
    input clk;
  input [N-1:0] d;
    output [N-1:0] q;
); //电平敏感型的1位数据锁存器
  assign q = clk ? d : q;
endmodule
```

当时钟信号为高电平时，将输入端信号打入锁存器;当时钟信号为低电平时，锁存原来已打入的数据。通常锁存器数据有效，滞后于控制信号有效。图 1-15 显示了锁存器的工作时序状态:

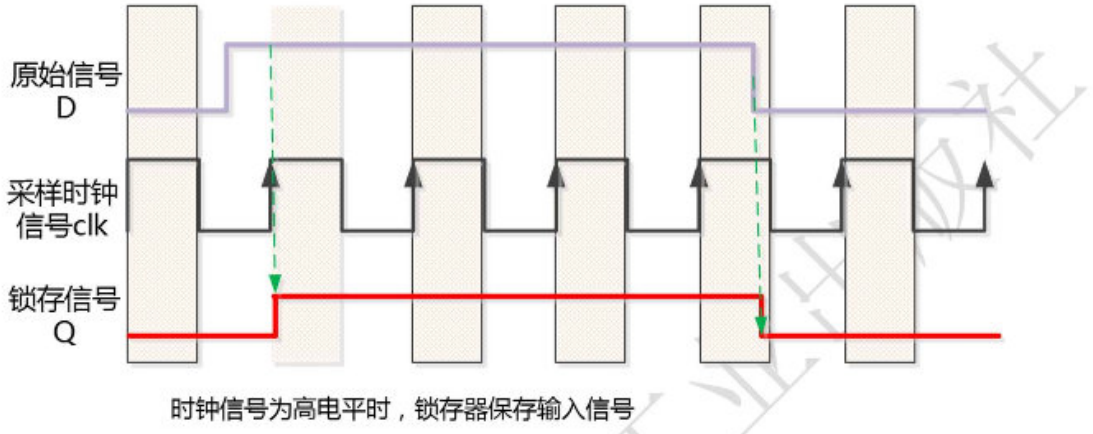


图 1-15 锁存器的工作波形示意图

Latch 结构也可以采用 always 语句描述，例如:

```
module latch_N#(parameter N=8)(
  input clk;
  input [N-1:0] d;
  output [N-1:0] q;
); //电平敏感型的8位数据锁存器
  always @(clk or d) //电平敏感
    if(clk) q=d;
endmodule
```

读者可以发现，正是由于 if 语句的不完整，缺少 else 分支，导致 q 变成锁存器。

5. 多级延迟的触发器

下面的代码是对输入信号进行多级延迟的代码:

```
module DFF_N #(parameter N=3)(
  input clk,
  input reset,
  input [N-1:0] D,
  output reg [N-1:0] Q
);
  reg [N-1:0]d0;
  reg [N-1:0]d1;
  always@(posedge clk or negedge reset)
```

```
    if(!reset)begin
      d0 <= 0; //异步清 0，低电平有效
      d1 <= 0; //异步清 0，低电平有效
      Q <= 0; //异步清 0，低电平有效
    end
        else begin
      d0 <= D;
      d1 <= d0;
      Q <= d1;
    end
endmodule
```

□其中，d0 是 D 的延迟一级采样或者称为 D 的延迟一拍信号，同样 d1 是 D 的延迟两拍信号，而 Q 则是 延迟三拍信号。该电路的主要用法: (1)如果 D 信号相对 clk 属于不稳定信号，则 Q 和 d1 输出相对 clk 属 于稳定信号; (2)可以通过 d1 和 Q 信号获取 D 信号的上升沿或者下降沿; (3)单纯的对 D 信号做延迟操作。

对于情况 1，会涉及时序和亚稳态的概念，后面的章节将会详细描述这种时序的处理方法。图 1-16 是一 个简单的不稳定时序概念示意图:

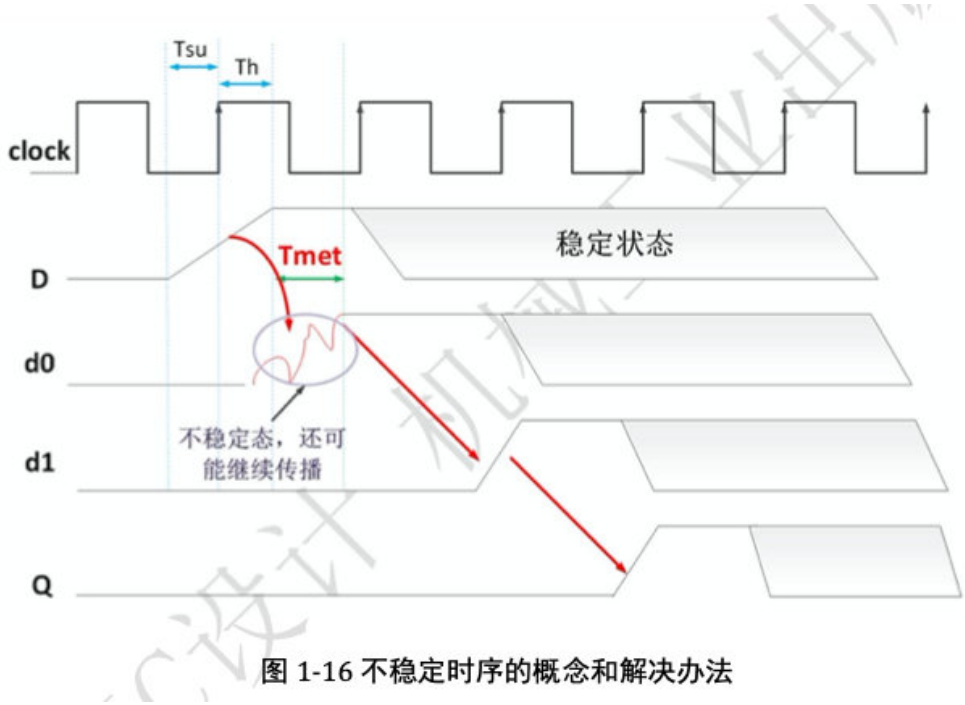


图 1-16 不稳定时序的概念和解决办法

对于情况二，先看如图 1-17 的时序图:

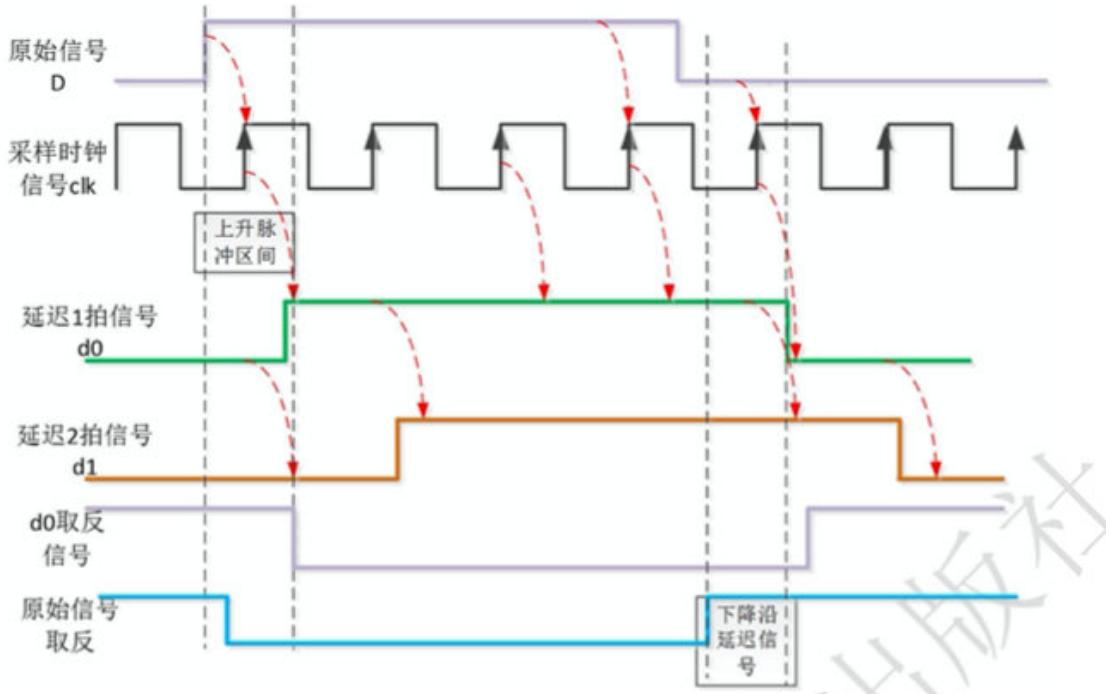


图 1-17 信号上升沿与下降沿检测

可以发现原始信号与延迟一拍的信号 d0 的反向信号相与，就是上升沿脉冲;而下降沿则是原始信号取反，然后与延迟一拍信号相与就是下降沿脉冲。如果担心采样不稳定，可以利用延迟两拍的 d1 信号进行相与。如果担心不定态，还可以将脉冲信号进行锁存。总之，信号延迟与信号本身做运算能够以最小代价生成边缘检测信号。实现参考代码如下:

```
module DFF_N #(parameter N=1)(
  input clk,
  input reset,
  input [N-1:0] D,
  output [N-1:0] D_rising_edge, //上升沿检测
    output [N-1:0] D_falling_edge //下降沿检测
  );
  reg [N-1:0]d0;
  reg [N-1:0]d1;
  reg [N-1:0]Q;
  always@(posedge clk or negedge reset)
    if(!reset)begin
      d0 <= 0;
        d1 <= 0;
        Q <= 0;
    end
    else begin
      d0 <= D;
      d1 <= d0;
      Q <= d1;
    end
  assign D_rising_edge = ~d0 & D;
  assign D_falling_edge= d0 & ~D;
endmodule
```

6 计数器

在 Verilog 中计数器是非常常见的电路，主要用途包括:(1)计数;(2)看门狗;(3)特殊的有限状态机描述。最简单的 N 位计数器，默认计数范围为[□0,2^{N} − 1□]的写法为:

```
module count#(parameter N=8)(
  input clk,
  input clear,
  output[N-1:0] cnt_Q
);
  reg[N-1:0] cnt;
  assign cnt_Q = cnt;
  always@(posedge clk)
    if(clear)
        cnt <= 'h0; //同步清 0，低电平有效
  else
    cnt <= cnt+1'b1; //减法计数
endmodule
```

通过 clear 信号清除计数器的数值，然后下一周期开始加 1 计数;当计数器计到能够存储的最大数值时，例如本例为 8 个 1，即 8'hff 就会自动回到 0，然后开始下一轮计数。

利用计数器进行状态指示，通常是将计数器的值与某一预设值比较，从而得出标志或状态指示，例如上面的计数器当计数到 8'h3f 就报警，可以通过如下的语句实现:

```
assign warning = (cnt == 8'h3f) ? 1'b1 : 1'b0;
```

如果想要实现□0, □□范围内计数，其中□ ≠ 2□，可以将 语句修改为:

```
always@(posedge clk)
  if(clear)
    cnt <= 'h0; //同步清 0，高电平有效 else if(cnt==K)
        cnt <= 'h0;
    else
        cnt <= cnt+1'b1; //减法计数
```

前面是累加计数，下面给出的是一个既可以递增也能递减，且具备初始值装载和复位的计数器。

```
module updown count #(parameter N = 8)(
    input clk,
    input clear,
    input load,
    input up_down,
    input [N-1:0] preset_D,
    output [N-1:0] cnt_Q
)
    reg[N-1:0] cnt;
    assign cnt_Q = cnt;
    always@(posedge clk)begin
        if(clear)
            cnt <= 'h0;
        else if(load)
            cnt <= present_D;
    else if(up_down)
        cnt <= cnt + 1;
    else
        cnt <= cnt -1;
    end
```

用过单片机或者 DSP 的读者都知道，看门狗定时器(WDT，Watch Dog Timer)是一种非常常见的电路，它实际上是一个计数器。在初始状态时，看门狗电路首先装载一个大数;当状态机或者程序开始运行后，看 门狗开始倒计数。如果状态机或程序运行正常，每隔一段时间应发出指令或信号让看门狗重新装载一个大 初始值，并再次开始倒计数。如果看门狗减到 0 就认为程序或状态机没有正常工作，就需要强制整个系统复位。

所以上面的计数器电路描述就是一个看门狗电路，只要加上 cnt==0 作为看门复位状态即可;而 load 信号则是状态机或软件给出的喂狗动作。

7. 分频器与门控使能信号

计数器实质上是对输入的驱动时钟进行计数，因此在某种意义上讲，计数器等同于对时钟进行分频。例如一个最大计数长度为N的计数器，将其最高位作为时钟输出（占空比不一定为1:1），则工作频率为输入时钟的$\frac{1}{2N}$。通常在ASIC和FPGA中，时钟都是全局信号，都需要通过PLL处理才能使用，但在某些简易场合，计数器输出时钟也是能够使用的，只是需要注意时序约束罢了。

下面的例子是一个基于计数器的通用时钟分频器，它能够支持任意整数倍分频，占空比尽量接近于1:1。

```
module clock_div_async #( parameter cfactor= 2)(
  input clk_in, //标准参考时钟输入
  input rst_x, //复位信号
  output clk_out//分频时钟输出
);
  reg clk_loc; //分频时钟的内部变量
  //reg [15:0] cnt;//最大支持65536分频
  reg [7:0] cnt;//最大支持256分频
  //根据分频因子选择输入等于输出还是分频
  assign clk_out = (cfactor==1)? clk_in : clk_loc;
  always@(posedge clk_in or negedge rst_x)
    if(!rst_x)begin
        cnt <= 'd0;
        clk_loc = 1; end
    else begin
        cnt <= cnt + 1'b1;
      if(cnt==cfactor/2-1) //分频时钟翻转
            clk_loc = 0;
        else if(cnt==cfactor-1) begin
            cnt <= 'd0;
            clk_loc = 1;
      end
    end
endmodule
```

此外，还可以通过对计数器状态判断，给出使能信号指示，从而间接达到门控时钟的目。例如，下面的 Q 寄存器，如果 enable 信号不为高，则无法更新内容，将一直保持现有工作状态。这与 Q 输入时钟 clk&enable 的组合信号没有太多区别。而 enable 信号可以通过 cnt 的某一个计数区间或状态来获得。

```
assign enable=(cnt[N-1]==1'b1)?1'b1:1'b0;
always@(posedge clk)
if(enable)
    Q<=D;
```

8. 例子小结

本节所讲的例子，主要是 D 触发器以及相关联的计数器以及分配时钟。需要注意两点: (1)D 触发器只 在指定时钟沿才会采样数据，并存储，而输出在整个时钟周期内维持不变。(2)明确阻塞赋值与非阻塞赋值的区别。

1.2.6 第四个Verilog程序：function与时序电路组合

前面给出的例子都是简单的赋值语句，而对于复杂的运算功能没有涉及，所以这里列举一个 function 的 例子，该函数的功能是计算有符号数的绝对值。

```
function [width-1:0] DWF_absval;
input [width-1:0] A;
begin
  DWF_absval = ((^(A^A) !== 1'b0)) ? {width{1'bx}}: (A[width - 1] == 1'b0) ? A :
(-A);
end
endfunction
```

该例子有两个语法点:(1) 函数的写法;(2)缩减运算符的用法，即自动缩减为 1bit，该语法能够自动判断 A 是否存在不定态或高阻态，主要用于仿真。

1. function的用法

Function 的标准写法如下:

```
function<返回值的类型或范围>(函数名);
  <端口说明语句>// input XXX;
  <变量类型说明语句>// reg YYY;
  begin
    <语句>
    ........
    函数名= ZZZ; //函数名就相当于输出变量;
  end
endfunction
```

函数的语法为:

- 定义函数时至少要有一个输入参量;可以按照 ANSI 和 module 形式直接定义输入端口;例如:

```
function[63:0] alu (input[63:0] a, b,input[7:0] opcode )
```

- 在函数的定义中必须有一条赋值语句给和函数名具备相同名字的变量赋值;
- 函数的定义不能包含有任何的时间控制语句，即任何用#、@、或 wait 来标识的语句;
- 函数不能启动任务;
- 如果描述语句是可综合的，则必须所有分支均赋值，不允许存在不赋值情况，只能按照组合逻辑方式描述。

2. function 与触发器电路结合

下面的例子是完整的 function 用法例子，function 本身描述的是组合电路，但赋值给某个触发器。

```
module MAC #(parameter N=8)(
  input clk,
  input reset,
  input [N-1:0] opa,
```

```
  input [N-1:0] opb,
  output reg[2*N-1:0] out
  );
function[2*N-1:0] mult; //函数定义，mult 函数完成乘法操作
  input[N-1:0] opa; //函数只能定义输入端
    input[N-1:0] opb; //输出端口为函数名本身
    reg[2*N-1:0] result;
  integer i;
  begin
    result = opa[0]? opb : 0;
    for(i=1;i<=N;i=i+1) begin
      if(opa[i]==1) result=result+(opb<<(i-1));
    end
    mult=result;
  end
  endfunction

  wire[2*N-1:0] sum;
    assign sum = mult(opa,opb) + out;
  always@(posedge clk or negedge reset)
    if(!reset)
        out<=0;
    else
        out<=sum;
endmodule
```

这段例子的含义是:将两个 bit 的数据相乘，并与当前已经缓存的数据相加，所得到的结果作为输出。这里的乘法电路实现是通过 bit 移位加实现的。实现流程图如图 1-16 所示:

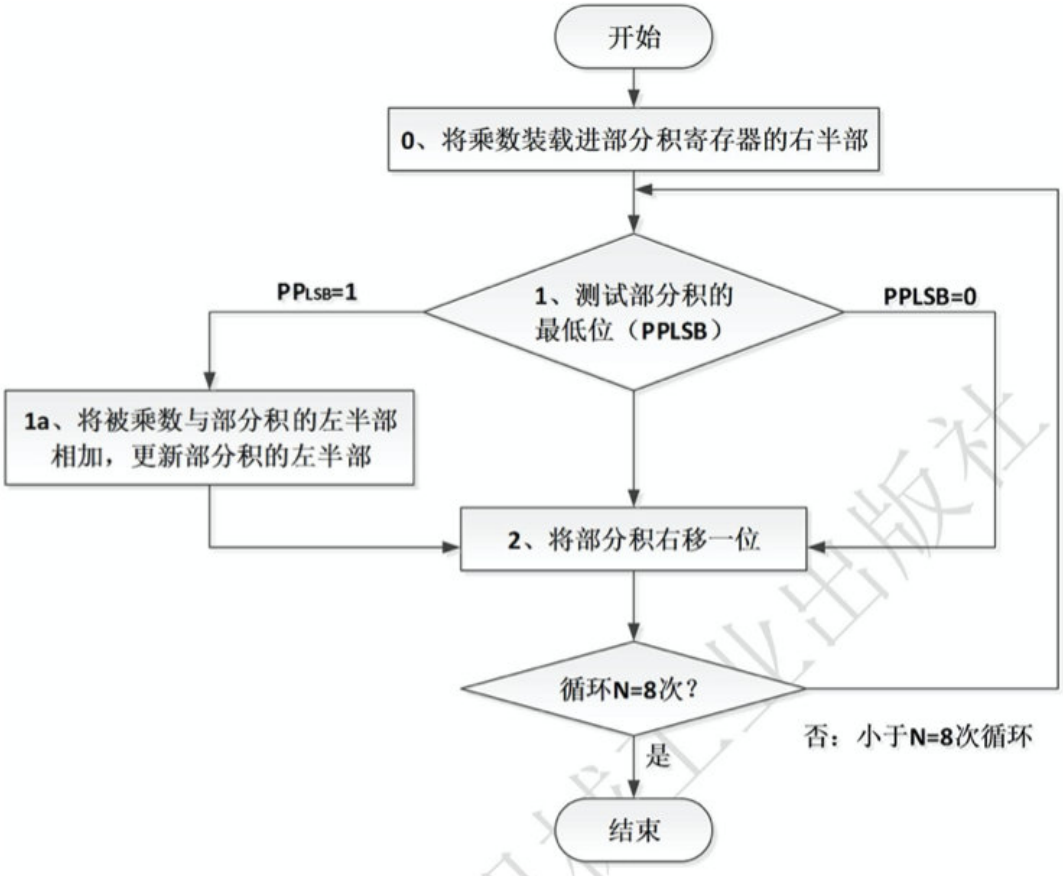


图 1-18 乘法实现流程图

上述代码包含电路实现原理图如图 1-19 所示:

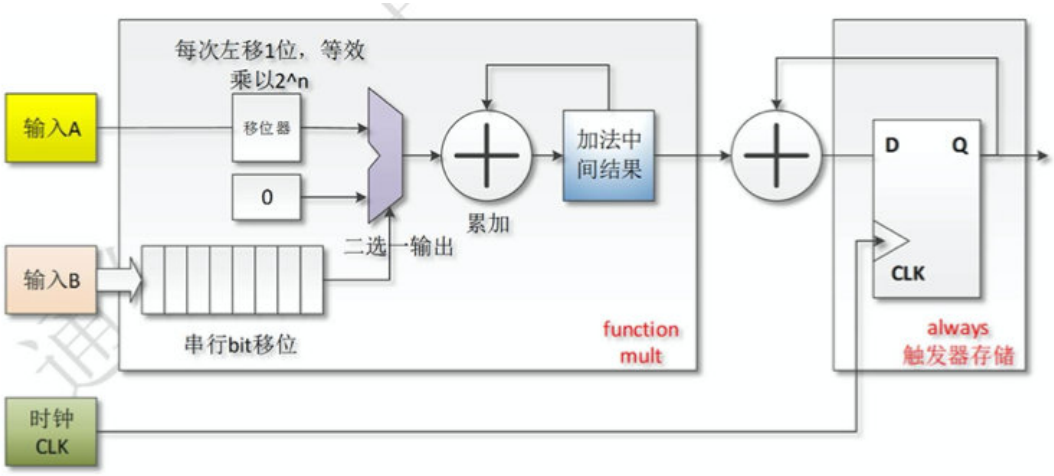


图 1-19 累加乘法原理图

整个电路图中 function 是对于移位累加部分，而 always 描述数据的存储过程。于是电路被很清晰的划分为两部分:(1)计算函数部分，这部分是纯组合电路实现;(2)D 触发存储部分。整体可抽象为:
$out = f_x(a, b, c..., n)$，描述伪代码为:

```
module abstract_circuit#(parameter N=1)(
  input clk,
  input reset,
  input a,
  input b,
  ... //省略部分输入描述
  input n,
  output reg[N-1:0] out
);
  function[N-1:0] fx; //函数定义，mult 函数完成乘法操作
    input a; //函数只能定义输入端
        input b; //输出端口为函数名本身
        ...
        input n; //输出端口为函数名本身
    begin
            fx=组合函数运算(a,b,....,n)
    end
    endfunction
  always@(posedge clk or negedge reset)
    if(!reset)
            out<=0;
    else
            out<=fx(a,b,....,n);
endmodule
```

这种方法描述出来的电路图如图 1-20 所示:

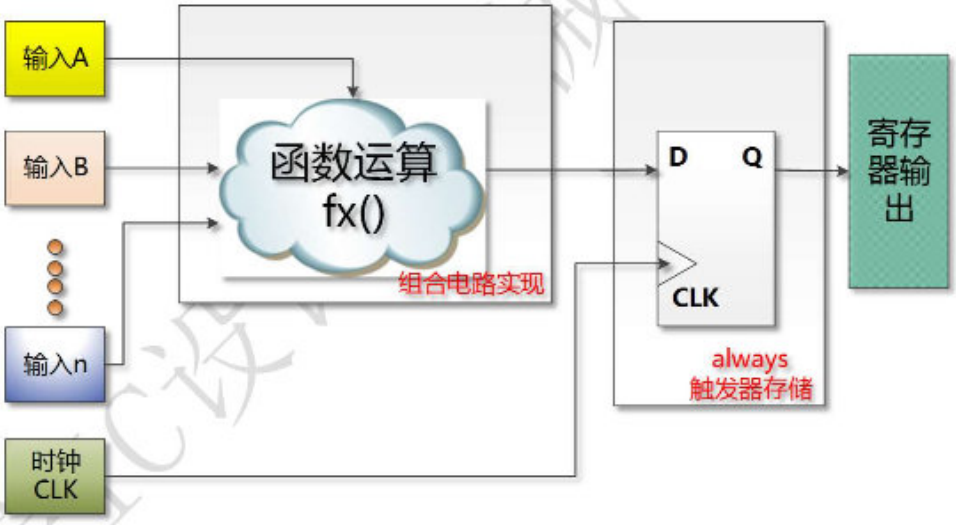


图 1-20 电路描述的原理

所以通过上述方式，如果不考虑时序或者最高工作频率的问题，任意一个通信算法，只要给出函数形式，就可以按照上面的模式形成电路。

但读者肯定会想到一个问题:如果函数 $fx()$是一个简单的加法减法，上述实现方法肯定没有问题，但是如果函数 $fx()$是许多个超越函数诸如□□□□□等的组合结果，则肯定会有疑惑，电路能够这样简单实现吗?

这种疑惑是非常正确的，这时候就需要建立一个时序的概念，即函数 $fx()$只能够承担有限的计算量，而 超出的计算量需要进行分解，变成多个小的$f\text{□□}_n(\text{□□})$实现。这种概念就是朴素的流水线电路设计。

3. Verilog电路时序模型的建立

众所周知，电磁波的传播速度是光速，IC 内部信号的传导速度也是光速，但这个的前提是携带信号不发生变化。如果一个信号由 0 变为 1，实际上是对后面的负载电路整个进行充放电，而这个是需要时间的。任何一个电路都可以按照如下的 RCL(阻容和电感)模型进行描述:

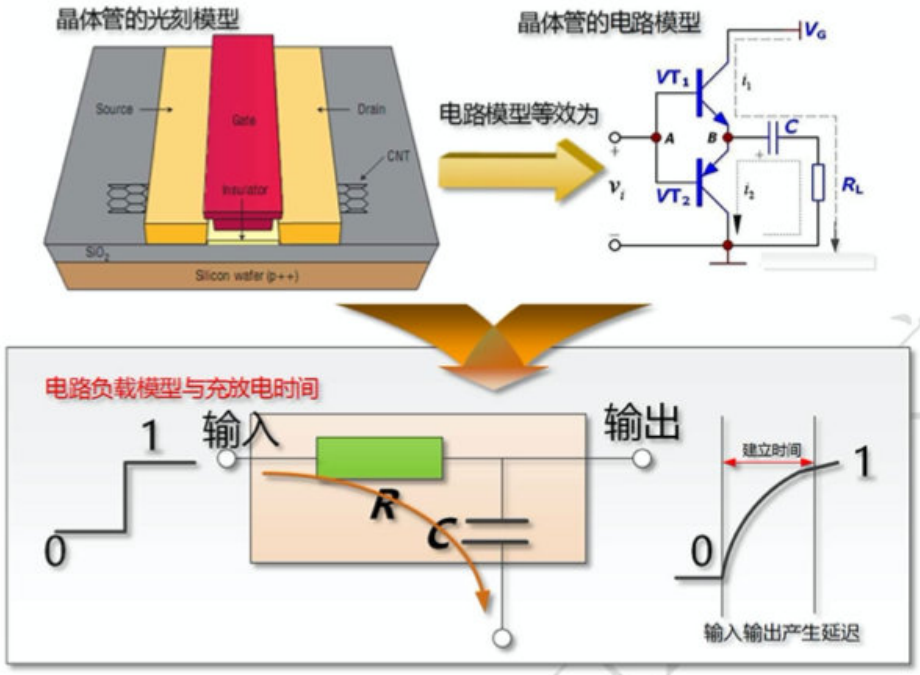


图 1-21 电路负载模型与时延关系

所以，任何的函数计算可以简化为一系列的 RC 电路组合，这些电路或串接或并联很多的负载。当输入发生 01 跳变时，等效于对后面的负载电路进行充放电，而每级负载都会有对应的延迟，如果串接延迟过多 就必然会导致末级响应过慢，导致无法在高时钟频率下工作。

因此站在电路角度，时序优化的核心工作，就是降低后级的 RC 负载、增强前级的驱动能力。

4. 基于算法视角的时序优化

前面提到了，如果设计的函数□(□)过大，就会需要进行函数拆解，从而使单个函数负载和时延控制在一定的范围内。经典的分解方法有三种:

$$
\begin{aligned}
&(1)\quad f(x) = f_1(x) + f_2(x)\\
&(2)\quad f(x) = g_1(x) \times g_2(x)\\
&(3)\quad f(x) = m(y), y = n(x)
\end{aligned}
$$

这三种拆解方法通常是混合应用，最终得到最佳的实现方案。

1. **累加拆解**

这类拆解是针对类似需求通过累加求和得出最终结果而设计的，例如对于 FFT 而言，计算过程为:

$$F[k] = \sum_{n=0}^{N-1} f[n] e^{-\mathrm{j}\frac{2\pi}{N}kn} \quad 0 \le k < N-1$$

当□ = 64或者更大时，一次性计算出□□□□，在逻辑芯片设计中可能性很低。所以此时就应当按照第一类方式进行拆解。即:

$$F[k] = f_0(x) + f_1(x) + f_2(x) + \cdots + f_{N-1}(x)$$

其中$f_i(x) = f[i]e^{-\mathrm{j}\frac{2\pi}{N}ki}$

此时的计算步骤就变为如下的步骤:

```
ACC_SUM=0
loop:
if(i<N)
  fi(x) = 计算fi(i)
  ACC_SUM = ACC_SUM + fi(x)
  i=i+1;
    goto loop
else
  输出最终结果=ACC_SUM//
```

可以发现，只需要单步计算分拆为 N 步计算，同时增添一个中间结果累加，即可在有限的时间内完成。对应的 Verilog 实现伪代码如下，此时会引入 Verilog 设计中的控制流或者控制状态机的写法。

```
module abstract_fx_add#(parameter BIT_W=8,parameter N=64 )(
    input clk,
    input reset,
    input start_fx,              //启动计算整个函数标志
  input [BIT_W-1:0]x_in, //每个节拍输入一个新的x，需要与start_fx配合
    output fx_finish,
    output [BIT_W-1:0] fx_out
);
reg [7:0] control_cnt;       //控制计数器，大于等于N就停止
always @(posedge clk or negedge reset)
 if(!reset)
    control_cnt<=0;
```

```
 else if(start_fx)
    control_cnt<=0;
 else if(control_cnt<N)
    control_cnt<=control_cnt+1'b1;
//根据控制计数器，判断计算是否结束，目前需要计算N次
assign fx_finish=(control_cnt==N) ? 1'b1 : 1'b0;

function[BIT_W-1:0] subfx;
  input [BIT_W-1:0] x;
  begin
    subfx=fx_i(x);
  end
endfunction

reg [2*BIT_W-1:0] sum_acc;
always @(posedge clk or negedge reset)
 if(!reset)
    sum_acc<=0;
 else if(control_cnt<N)
  sum_acc<=sum_acc+subfx(x_in);      //对分拆结果累加
//输出最终的结果，截位至有效位，后面有专门章节论述
assign fx_out=sum_acc[2*BIT_W-1:BIT_W];
endmodule
```

这类拆解的关键包含两点:(1)设计一个计数器，用于统计当前的累加计算次数，并给出状态指示。(3) 单独的子函数计算以及将单次子函数计算结果与中间缓存结果合并。

实现的原理图如图 1-22 所示:

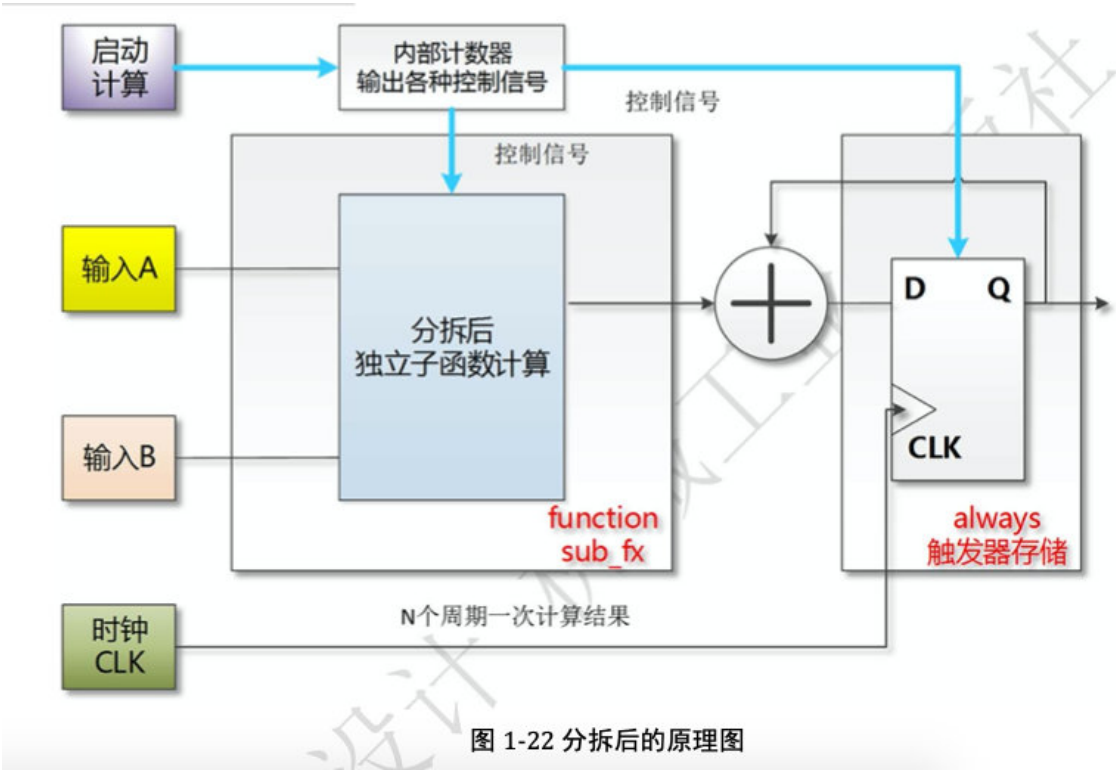


图 1-22 分拆后的原理图

第一种拆解实质是将一步完成的工作，分解为□个小的步骤完成，整体完成的时间拉长了。后面的章节将会提到，这种设计方法称为折叠(fold)。读者也会发现，这里面引入了控制流的概念。

2. **累乘拆解**

针对第一步拆解后的子函数□□$f_i(x) = f[i]e^{-j\frac{2\pi}{N}ki}$□□，读者很容易发现也不是简单一步就能够完成计算的。因为求□ $e^{-j\frac{2\pi}{N}ki}$肯定是一个非常耗时的算法，即使计算完毕，后面还有一个耗时的复数乘法在等待。整体电路描述如图 1-23 所示:

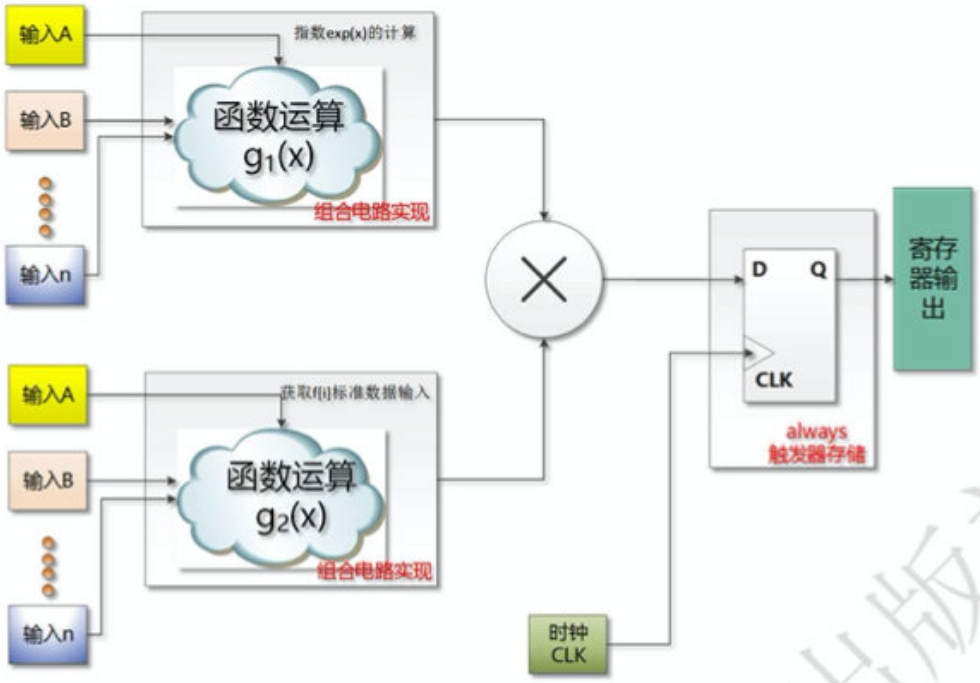


图 1-23 为拆解之前的电路实现

所以这就引出了第二类拆解，累乘拆解。将□□ (□)分解为如下步骤，每个步骤占用一个运算周期:

$$
\begin{aligned}
&\text{计算}\, g_2(x) = e^{-j\frac{2\pi}{N}ki} \\
&\text{令}\, g_1(x) = f[i] \\
&\text{计算}\, f_i(x) = g_1(x) \times g_2(x)
\end{aligned}
$$

所以，读者可以对比一下两者的计算过程，具体如图 1-24 所示:

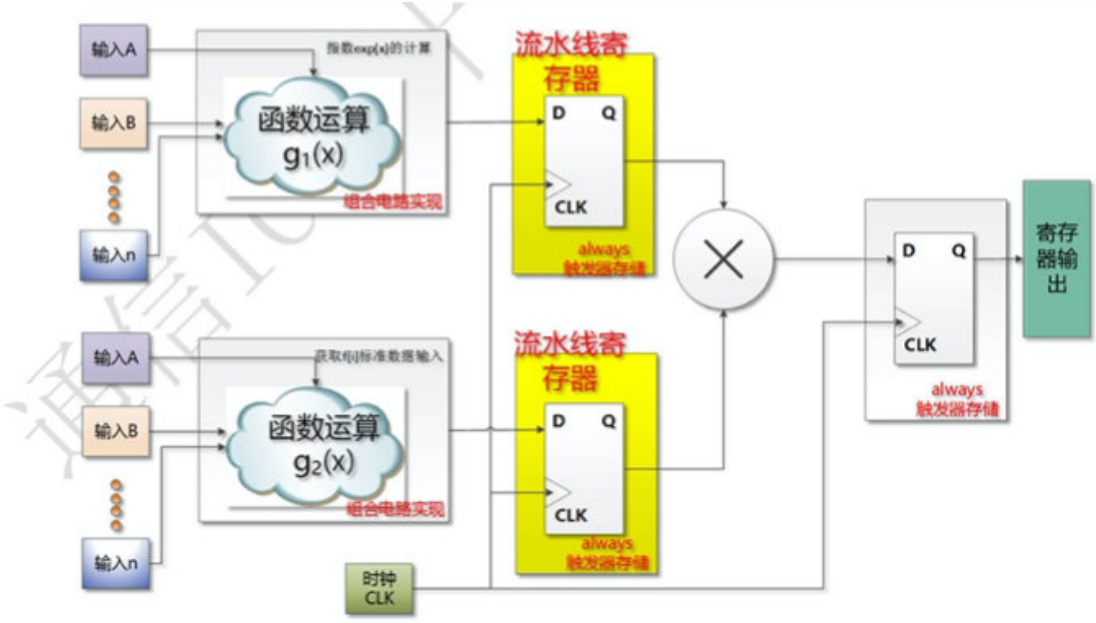


图 1-24 插入流水线寄存器后的电路优化

第二类拆解的本质是通过中间插入寄存器，将耗时较长的运算链打断，从而达到时序优化，电路在较高的工作频率工作的目的。

图 1-25 为整体拆解过程的抽象图:

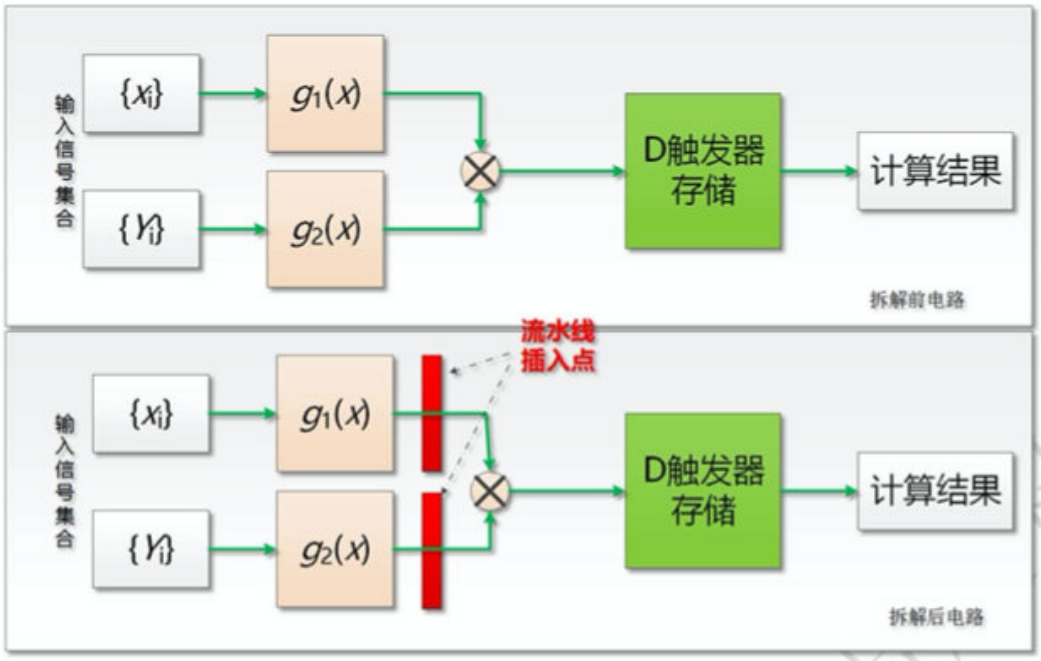


图 1-25 拆解前后的电路对比示意图

例如原始程序为:

```
module decompose2 #(
parameter A_width = 16,
parameter B_width = 16
)(
input                       CLK,
input [A_width - 1 : 0]   A,
input [B_width - 1 : 0]   B,
output reg[A_width+B_width-1:0] PRODUCT
);
localparam width = A_width + B_width;
reg [width - 1 : 0] temp_a;
reg [width - 1 : 0] temp_b;

parameter angle_width = A_width;
parameter sin_width = width;
parameter cos_width = width;
`include "DW02_cos_function.inc"
`include "DW02_sin_function.inc"

function[width-1:0] gx1;
  input [A_width-1:0] x;
  begin
    gx1=DWF_cos(x);// gx1的内核函数
  end
endfunction

function[width-1:0] gx2;
  input [B_width-1:0] x;
  begin
    gx2=DWF_sin(x);// gx2的内核函数
  end
endfunction
```

```
//Sign extending
always @( A or B) //组合电路
begin
  temp_a = gx1(A); //将fx拆解为gx1与gx2的乘积
  temp_b = gx2(B);
end

//Multiplier - product with a clock latency of one
always @ ( posedge CLK )
  PRODUCT <= temp_a * temp_b; //乘法完成后存储

endmodule
```

分拆后的程序就是将 与 缓存到 D 触发器中，而乘法则从稳定后的触发器输出结果。所以分拆代码相对原始代码，只需要加上寄存器缓存即可，即代码变更为:

```
always @ ( posedge CLK )//时序电路，插入流水线寄存器
  begin
     temp_a <= gx1(A); //将fx拆解为gx1与gx2的乘积
   temp_b <= gx2(B);
  end
```

3. **函数嵌套模式拆解**

继续针对前面的 FFT 表达式，可以发现$g_2(x) = e^{-j\frac{2\pi}{N}ki}$ □□，可以分解为两个步骤:

- 计算$\frac{2\pi}{N}ki$，得出旋转的角度$\theta = \frac{2\pi}{N}ki$
- 计算$e^{-j\theta} = \cos\theta + j\sin\theta$

也就是，$f(x) = e^{-j\theta}, \theta(i) = \frac{2\pi}{N}ki$这种函数嵌套模式。这类模式在计算诸如多项式求和中应用广泛:

$$f(x) = \sum_{i=0}^{N} c_i x^i$$

因为□$f(x)$可以化简为:$f(x) = \sum_{i=0}^{N} c_i x^i = \prod_{i=0}^{N} (a_i + x)$，即通过$(a_i + x)$个多项式相乘即可实现。按照标

准定义:□$f(x) = m(y), y = n(x)$，对应有

$$y_i = n(x) = (a_i + x)$$
$$f(x) = m(y) = \prod_{i=0}^{N} y_i$$

在矩阵变换和分组码纠错中，也存在大量的类似拆解方法。例如，对于 RS 等分组码在求伴随多项式时， 需要计算如下的矩阵:

$$S^T = HR^T = \begin{bmatrix} (a)^{n-1} & \cdots & (a)^{2t}\cdots & a & 1 \\ \vdots & \ddots & \cdots & \cdots & \vdots & \vdots \\ \left(a^{2t}\right)^{n-1} & \cdots & \left(a^{2t}\right)^{2t}\cdots & a^{2t} & 1 \end{bmatrix} \begin{bmatrix} c_{n-1} \\ \vdots \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} s_1 \\ \vdots \\ s_{2t-1} \\ s_{2t} \end{bmatrix}$$

其中，□□ c_i为输入，□s_i为需要计算的结果。

个矩阵算法如果按照常规算法，计算过程非常的繁琐，但如进行拆解后，就会发现:

s_{i-1}与s_i之间的单哥乘积项只相差$\begin{bmatrix} a^{n-1} & a^{n-2} & \cdots & 1 \end{bmatrix}$倍数。所以可以利用$si-1$的中间计算结果，计算出$s_i$中间结果。$s_{i-1}$和$s_i$之间等效相差如下的乘法系数：

$$H = \begin{bmatrix} a^{n-1} & a^{n-2} & \cdots & a^{2t} & \cdots & a & 1 \end{bmatrix}$$

第一次的数据输入为$\begin{bmatrix} c_{n-1} & \mathbf{c}_{n-2} & \cdots & c_{2t} & \cdots & c_1 & c_0 \end{bmatrix}$，而后续第□次滤波器输入为$\begin{bmatrix} c_{n-1}a^{(i-1)(n-1)} & \cdots & c_{2t}a^{(i-1)2t} & \cdots & c_1 a^{i-1} & c_0 \end{bmatrix}$，这个序列正是计算第□$i-1$次的中间结果。

这个例子的详情可以参见后续信道编解码中关于 RS 和 BCH 译码的章节，而实现的电路图如图 1-26 所示：

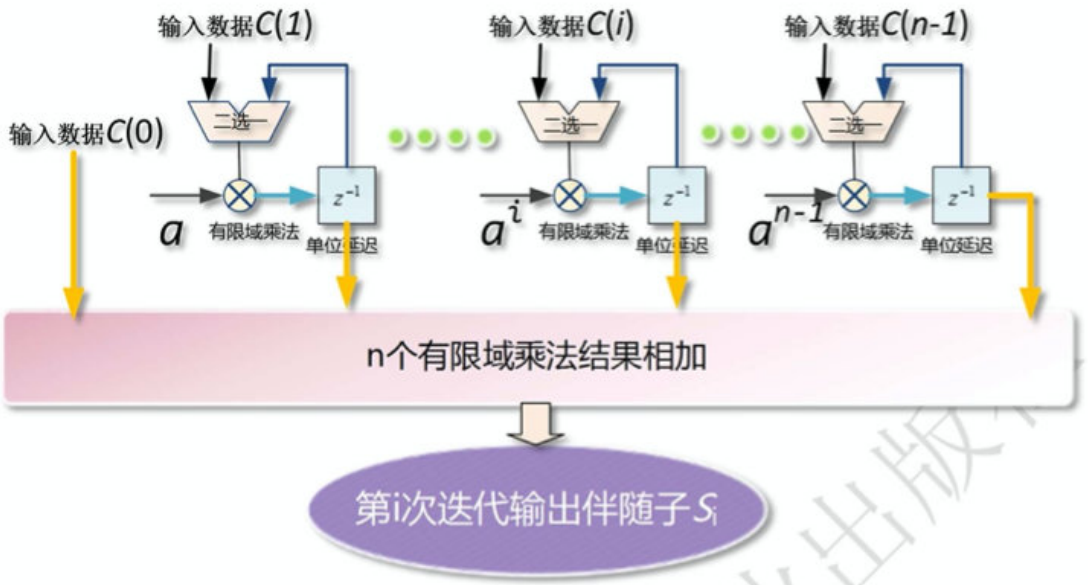


图 1-26 函数拆解的实现

第三种拆解的电路模型，如图 1-27 所示:

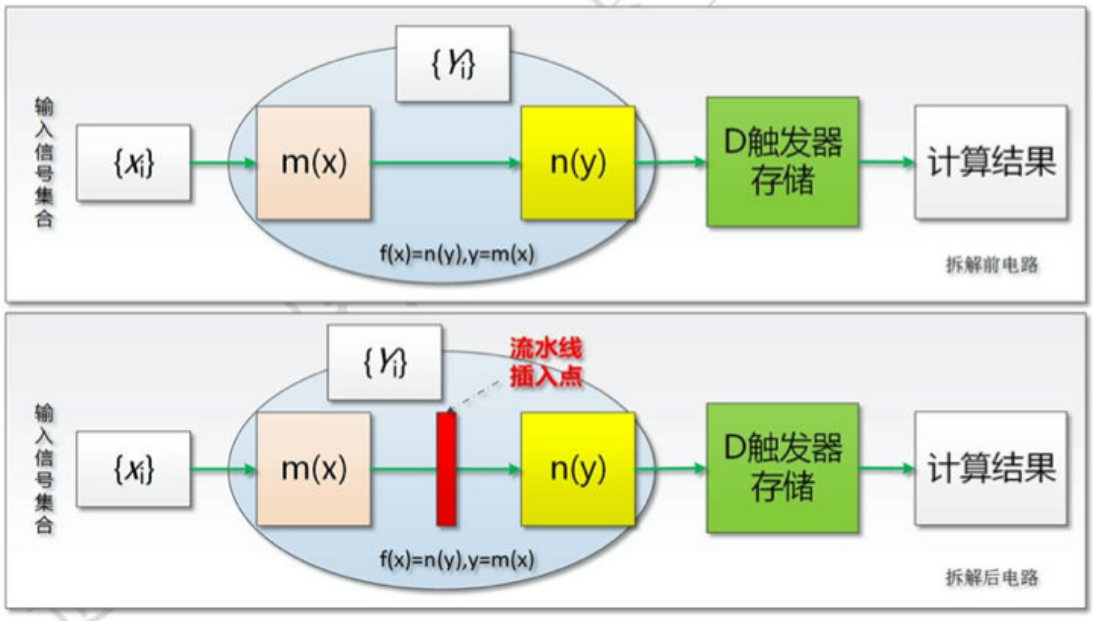


图 1-27 基于函数嵌套的拆解模型示意图

而对应的程序例子如下,其中原始代码如下:

```
module decompose3 #(parameter width = 16)(
input                CLK,
input [width-1:0]    A,
```

```
output reg[width-1:0] PRODUCT
);

reg [width - 1 : 0] temp_a;
reg [width - 1 : 0] temp_b;

parameter angle_width = width;
parameter sin_width = width;
parameter cos_width = width;
`include "DW_sqrt_function.inc"
`include "DW02_sin_function.inc"

function[width-1:0] nx;//nx=sin(x)
  input [width-1:0] x;
  begin
    nx=DWF_sin(x);// nx的内核函数,可替换任意函数
  end
endfunction

function[width-1:0] my;//my=sqrt(y)
  input [width-1:0] y;
  begin
    my=DWF_sqrt_tc(y);// my的内核函数,可替换任意函数
  end
endfunction

always @( A or temp_a )//组合电路，用于描述f(x)实现过程
begin
  temp_a <= nx(A); //fx可分解为函数嵌套f(x)=m(y),y=n(x)
  temp_b <= my(temp_a);
end

//Multiplier - product with a clock latency of one
always @ ( posedge CLK )
  PRODUCT <= temp_b; //存储temp_b

endmodule
```

进行函数嵌套修改后,对应的关键代码就是插入流水线寄存器,实现代码如下:

```
always @ ( posedge CLK )//时序电路，插入流水线寄存器
 begin
   temp_a <= nx(A); //将fx分解为函数嵌套f(x)=m(y),y=n(x)
   temp_b <= my(temp_a);
 end
```

4. **时许优化小节**

前面的三种拆解方法，核心都是在保证输入和输出结果不变的前提，通过插入寄存器或者改变输入输出的时间关系，实现时钟工作频率提高的方法。读者可以按照这种思路，拓展出更多的拆解方法。本章将在后面的内容详细讲述各种改变时序的技巧，包括折叠、展开、重定时和脉动。这些都是 IC 设计的基本功，在 基础单元设计时，就应当充分考虑这类实现方法。

5. 函数与通用描述的转换方法

本小节讲述了 function 的抽象方法，但利用 function 描述有时候稍显繁琐，所以通常采用等效直观方法 进行描述。所以读者需要学会如何将 function 转换为通用描述。以下的例子是本节 function 例子改编为直接 描述形式的代码:

```
module function2combine #(parameter N=8)(
  input clk,
  input reset,
  input [N-1:0] opa,
  input [N-1:0] opb,
  output reg[2*N-1:0] out
);
  reg [2*N-1:0]mult; //将function输出例化为变量
  reg [2*N-1:0]result;//function内部变量例化 integer i;
  always@ (*) //组合电路，替代function描述
    begin
        result = opa[0]? opb : 0;
        for(i= 1; i <= N; i = i+1)
        begin//注意此处为阻塞赋值，设计电路将完全展开
                if(opa[i]==1) result=result+(opb<<(i-1));
        end
      mult=result;
    end

  wire[2*N-1:0] sum;
  assign sum = mult + out;
  always@(posedge clk or negedge reset)
    if(!reset)
         out<=0;
    else
     out<=sum;
endmodule
```

读者可以发现，function 其实是一种抽象形式描述，是一种需要实例对象才能使功能生效的描述。如果将 function 的定义头替换为 always@(*)，然后定义必要的变量，就可以直接转换为实际的组合电路。所以 function 转为实际电路描述，可以采用 always@直接例化的方法实现。

标准的转换方法示意图如图 1-28 所示:

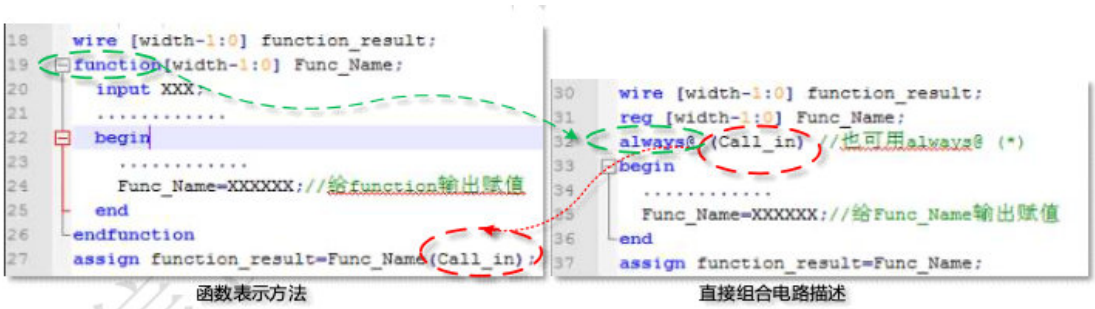


图 1-28 function 转为直接描述的对比

6. Function小结

站在 Verilog 综合器的角度，读者描述的任何一个组合电路块，或者组合 always 语句，以及所有赋值给 D 触发器之前的逻辑，都将抽象为多值函数:即多个输入与输出的组合。

前面论述的各种拆解方法，所有目的都是为了优化时序，并方便综合软件更快速、更灵活的实现 HDL 到 电路网表的转换。

1.3.7 第五个Verilog 程序：有限状态机

我们知道，任何事物都可以通过这样的模型描述:给定某个输入就有特定的输出，这个输出可能只与当前输入相关，也可能与以前的历史输入相关(数学上称为马尔可夫过程)。

对应到逻辑电路设计上，任何一个复杂的数字电路功能，都可以通过对其输入输出的行为进行准确描述。通常单纯的输入输出行为描述是不能够完全描述特定的复杂行为，因为经常出现相同的输入，但由于历史输入不同导致输出不同的情况，所以如果想要避免上述情况发生，就必须记忆历史记录。所以，通过存储器加上组合电路就能够完整的描述任何复杂的数字电路功能。

在数字逻辑设计中，这种建模方法被称为有限状态自动机(Finite State Machine，FSM)。有限状态机能够 将任意模型简化为如下的描述:将要输出的结果是当前状态以及当前输入的组合。

1. 有限状态机的设计思想

有限状态机 FSM 是由一组状态(states)、一个起始状态(start state)、一组输入和根据输入及现有状态转换 为下一个状态的转换函数(transition)组成的计算模型，具体如图 1-29 所示。

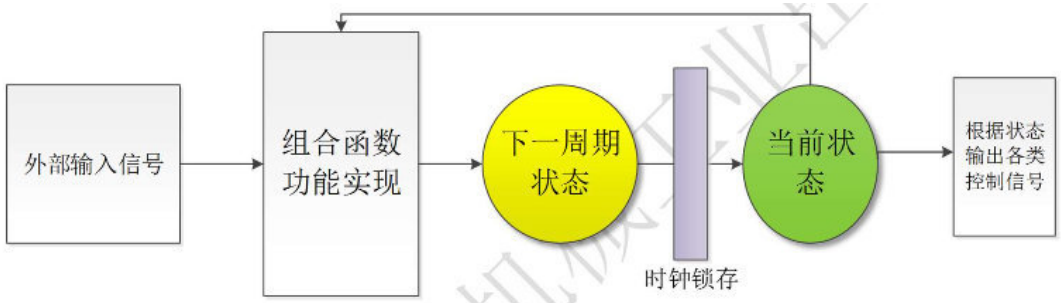


图 1-29 FSM 实现原理

有限状态机 FSM 思想广泛应用于硬件控制电路设计，也是软件上常用的一种处理方法(软件上称为 FMM-有限消息机)。它把复杂的控制逻辑分解成有限个稳定状态，但这并不意味着状态机只能进行有限次的处理; 相反，有限状态机是闭环系统，状态可以无限循环跳转，可以用有限的状态处理无穷的事务。

读者可以回忆一下，前面建立的逻辑电路基本模型(图 1-30)，正是一系列的寄存器(D 触发器)和寄存 器之间的组合电路。

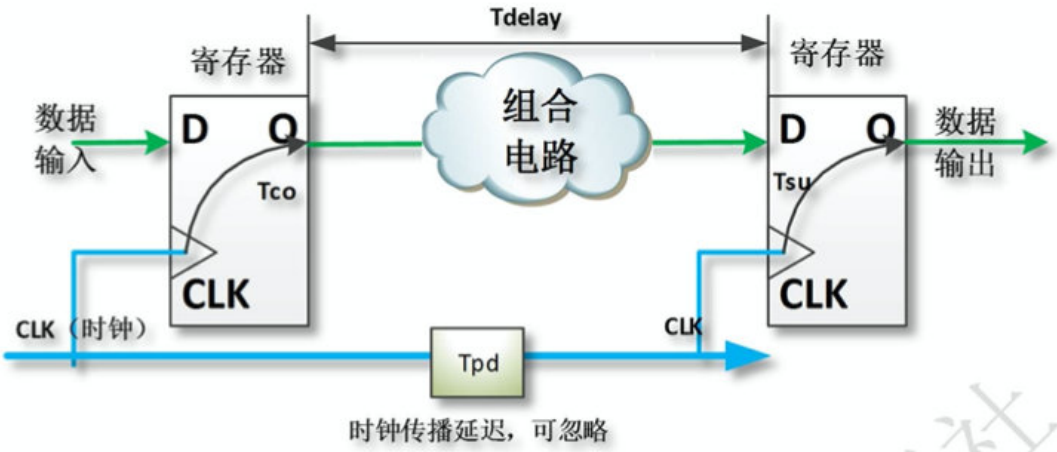


图 1-30 标准逻辑电路模型

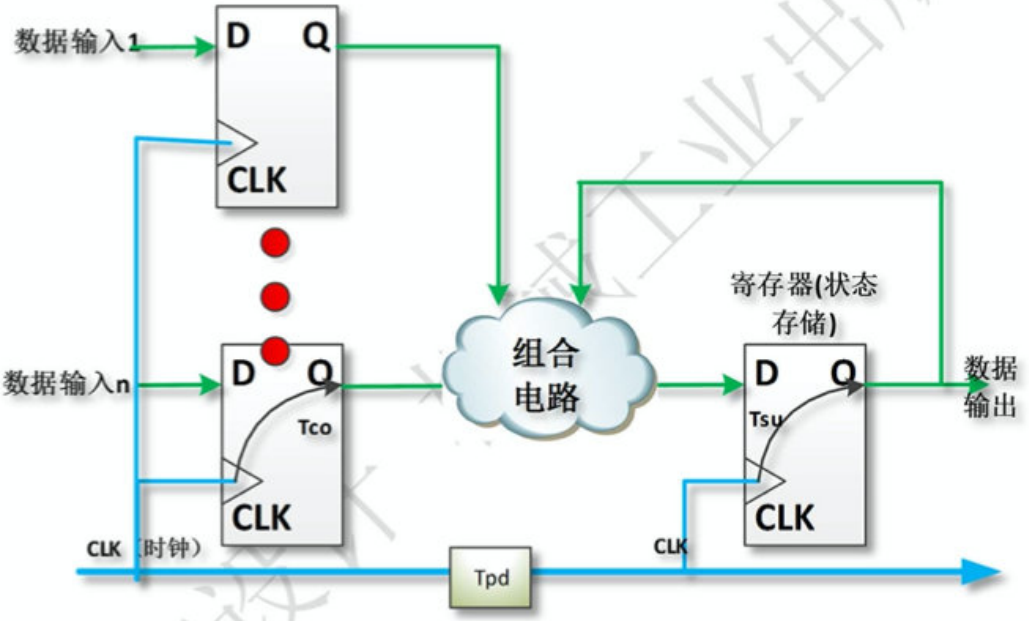


图 1-31 标准逻辑电路的 FSM 变形

图 1-30 中的寄存器可以存储关于过去的信息，它反映从系统开始到现在时刻输入的变化;而通过组合逻辑，可以实现对输入信号做特定的动作响应，例如输入某个规定的动作，则电路做出对应的响应，并记录这些动作指示到寄存器中。因此从状态描述的角度讲，标准逻辑电路模型经过简单变形就是一个标准有限状态机(图 1-31)，具备描述任何自然模型的能力。

2. 有限转态机设计

有限状态机的可以分成四大部分：状态机的编码、状态机的复位、状态机的条件转换和状态机的输出

1. **状态机的编码**

状态机编码的主要目的是为了定义参数，从而增强程序的可读性，状态编码有多种:顺序码(二进制码)，格雷码 Gray，独热编码 One-hot。此外还有 Johnson 码和 Nova 三态码。但常用的还是前面提到的三种。

采用格雷码，相邻状态可以减少瞬变的次数，有时不能在所有状态中采用格雷码，则应在状态矢量中增加触发器的数量以减少开关的次数。另一种方法是使用 One-hot 编码，该编码方式使用一组码元，每一个码元仅有 1bit 有效。虽然该编码使用的触发器较多，却可减少组合逻辑的使用，在带多个输出且每个输出是几个状态的函数的状态机更是如此。

2. **状态机的复位**

状态机的复位分成同步复位和异步复位。同步复位是指复位要与分频后的时钟信号同步，触发信号仅为分频后的时钟信号;而异步复位是指复位与分频后的时钟信号和复位信号都参与触发。归根到底就是在出发复位函数的敏感列表中是否把复位键信号作为触发信号。此外，由于电路外部干扰等情况，状态机存在进入未知状态的情况，此时需要对状态机自动复位，而添加看门狗电路就是最佳选择。

3. **状态机的条件跳转**

状态机的跳转可以说是状态机的核心部分，状态机的条件跳转是控制整个状态机状态之间的切换，从而决定输出的情况。通常需要列出类似如下的状态跳转表格，方便整理状态机的设计思路。

current state	control	next state
STO	--	ST1
ST1	1	ST2
	0	ST3
ST2	--	ST3
ST3	--	ST0

4. **状态机的输出**

设计有限状态机的核心就是为了准确描述输入与输出之间的关系。这一步目前有两种描述风格:Moore 型和 Mealy 型。

Moore 型的状态机输出只与当前状态有关，实现电路如图 1-32 所示:

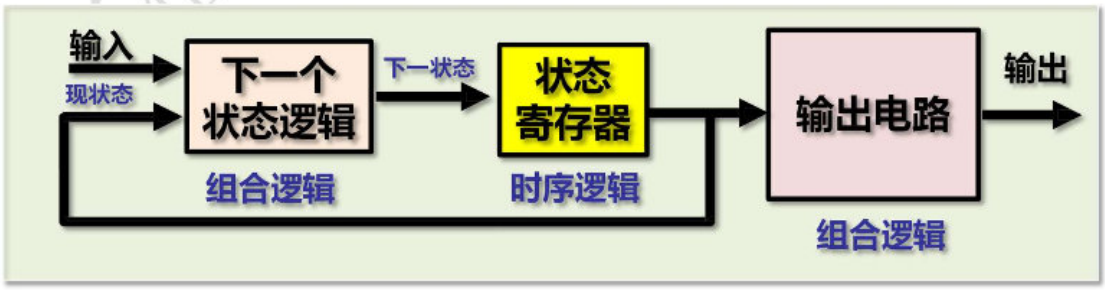


图 1-32 Moore 型电路

而 Mealy 型的状态机输出不仅与当前状态有关，还与输入有关，实现电路结构如图 1-33 所示:

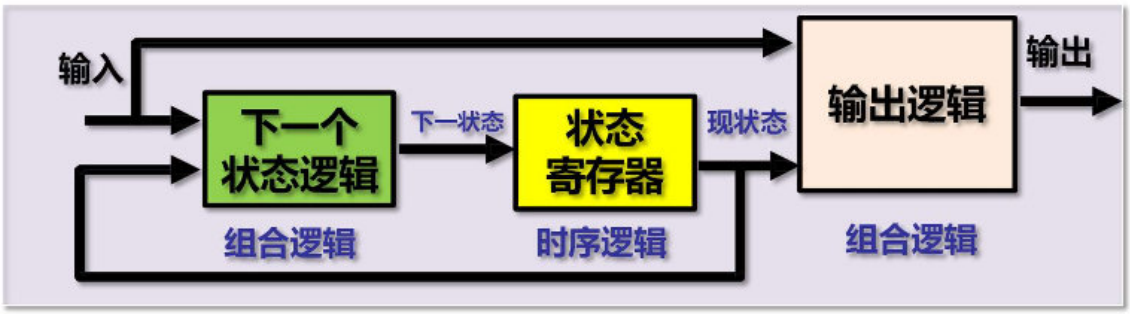


图 1-33Mealy 电路

由于 Mealy 型状态机的输出与输入有关，很容易导致输出信号出现毛刺，所以建议读者使用 Moore 型状态机进行状态机描述。

5. **转态机的设计步骤**

有限状态设计可以归纳为以下几个步骤:

第一步:对描述的逻辑进行抽象，得到状态转换图，实际就是将逻辑关系表示为时序逻辑函数的过程。通常包含以下工作。

- 分析给定的逻辑问题，确定输入变量、输出变量以及电路的状态数。通常是取原因(或条件)作为输入变量，取结果作为输出变量。
- 定义输入、输出逻辑状态的含意，并将电路状态顺序编号。
- 按照要求列出电路的状态转换表或画出状态转换图。

这样，就把给定的逻辑问题抽象到一个时序逻辑函数了。

第二步:状态化简，这一步能够有效降低电路实现的复杂度。电路的状态数越少，存储电路也就越简单。状态化简的目的在于将等价状态 合并，从而得到最简的状态转换图。

第三步:状态分配，就是对每个状态分配一个寄存器值。

第四步:选定触发器的类型并求出状态方程、驱动方程和输出方程。

第五步:按照方程得出逻辑图。

3. 状态机的三种风格描述

FSM 状态机可以有多种风格描述，包括一段式描述，两段式描述和三段式描述。每种都有其优缺点，都有适用的应用场合，但从芯片设计实践角度，最佳的描述方法是三段式描述。

下面的例子包含三个输入:in1、in2、in3，三个输出:out1、out2、out3。而状态转移顺序以及输出结果如图 1-34 所示。本书将提供三种不同描述风格的例子，方便对比各自差异。

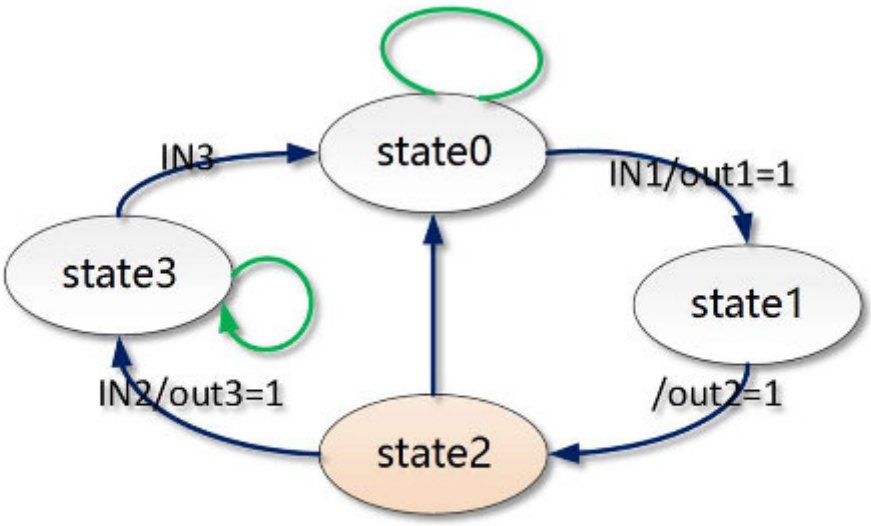


图 1-34 状态转移图

1. **一段式描述**

一段式的风格是将状态译码、状态寄存和输出放在一个 always 块，即在一个 always 块中完成整个状态机的设计。这种描述方式优点是代码简单，缺点是代码维护与修改非常困难。

上面的状态转移图(图 1-34 状态转移图)，采用一段式有限状态机进行描述的 Verilog 代码如下:

```
module FSM_style1(
input clk,
input rst_n,
input in1,
```

```
input in2,
input in3,
output reg out1,
output reg out2,
output reg out3
);
reg[3:0] state;
parameter state0 = 4'b0001,state1 = 4'b0010,
          state2 = 4'b0100,state3 = 4'b1000;
always @(posedge clk or negedge rst_n)
  if(!rst_n)
    state <= state0;
  else
  case(state)
    state0: begin
      if(in1) begin
        state<=state1;
        out1 <= 1;
      end
      else state<=state0;
    end
    state1: begin
      state<=state2;
      out2 <= 1;
    end
    state2: begin
      if(in2) begin
        state<=state3; out3 <= 1;
      end
      else state<=state0;
    end
    state3: begin
      if(in3) state<=state0;
      else begin
        state<=state0;out3 <= 1;
      end
    end
    default:
    begin
      state <= state0; out1 =0;out2=0;out3=0;
    end
  endcase

endmodule
```

2. **二段式与三段式描述**

两段式的代码风格由两个always 块构成，其中一个always块用于完成状态寄存(时序逻辑)，另一个always块即把状态译码和状态输出两个组合逻辑放在一起。

三段式的代码风格则将状态译码、状态寄存和输出分别放在三个 always 块，相对两段式逻辑层次更清晰。下面的代码是图 1-34 状态转移图的三段式代码描述。

```
module FSM_style3(
input clk,
input rst_n,
```

```
input in1,
input in2,
input in3,
output reg out1,
output reg out2,
output reg out3
);
reg[3:0] state,next_state;
parameter state0 = 4'b0001, state1 = 4'b0010,
          state2 = 4'b0100, state3 = 4'b1000;
//第一段 组合电路用于状态译码
always @(state or in1 or in2 or in3)
  case(state)
    state0:if(in1) //根据条件，选择目标跳转状态
        next_state<=state1;
      else
        next_state <= state0;
    state1: next_state<=state2;
    state2: if(in2)
        next_state<=state3;
      else
        next_state <= state0;
    state3: if(in3)
        next_state<=state0;
      else
        next_state <= state3;
    default:
      next_state <= state0;
  endcase

//第二段：更新状态寄存器
always @(posedge clk or posedge rst_n)
  if(!rst_n)
    state <= state0;
  else
    state <= next_state;
//第三段：利用状态寄存器输出控制结果
always @(state)
begin
  //首先产生默认值，后续再改写，防止锁存器产生
  {out1,out2,out3}=3'b000;
  case(state)
    state1: {out1,out2,out3}=3'b100;
    state2: {out1,out2,out3}=3'b110;
    state3: {out1,out2,out3}=3'b111;
  endcase
end
endmodule
```

3. **FSM编码风格小结**

对于有限状态机的描述风格，业内公认的结论如下:

1. 一段式的状态机从功能上说没有错误，但其可读性差，这种风格的状态机不能被综合工具很好的识别，同时很难对其进行优化。
2. 两段式的状态机的写法将组合逻辑和时序逻辑分开，具有较好的可读性，更有利于综合工具对状态机的优化。

3. 三段式的状态机的写法除了具有两段式的优点，还可以对输出进行寄存，可以有效地滤除毛刺，提高工作效率。

所以推荐读者在开始采用 HDL 编写 Verilog 代码时，就采用标准的三段式描述。

4. **有限状态机的判别标准**

一个好的状态机的标准很多,但最重要的几点如下：

第一，状态机必须是所有状态都能遍历，任何一个状态都能最终转移到缺省安全状态，状态机不会进入死循环，或者不会进入非预知的状态。这里面包含两层含义:(1)要求 FSM 要完备，即使状态机由于受到干扰而进入异常状态，也能很快恢复到正常状态;(2)状态机不会因为代码编写风格问题，导致跳转异常或 者出现临时状态(毛刺)。所以对于复杂的状态机，通常会设计一个看门狗电路(计数器定时清零电路)，当 状态机在规定的时间内没有跳转，就会复位到规定合法状态。

第二，状态机的设计要清晰易懂、易维护。每一个状态都有明确的转入状态和转出状态，而且转入转出条件均是可达的。对于复杂的状态描述，最好采用一个主状态机和若干个从状态机分开描述。

第三，状态机的设计要满足设计的面积和速度的要求。

此外，建议对状态机的各个状态利用 localparam 或 parameter，将状态定义为有意义的符号名。前面的程 序例子均采用类似规则命名。

4. 有限状态机举例

下面的例子是一个标准的 Moore 型有限状态机描述方法，也是逻辑设计中推荐的设计风格，属于三段式 描述:

```
module FSM(
  input clk,
  input reset,
  input start,
  input step2,
  input step3,
  output reg[2:0] fsm_out
);
  localparam state0=2'b00;
  localparam state1=2'b01;
  localparam state2=2'b11;
  localparam state3=2'b10;
  //标准三段式编码，每个周期更新当前状态
  reg[1:0] state;
  reg[1:0] next_state;
    always@(posedge clk or negedge reset)
      if(!reset)
            state <= state0;
   else
            state <= next_state;
 //根据当前状态和输入，确认下一个周期的状态
  always@(state or start or step2 or step3)
    begin
        case(state)
        state0: begin
            if(start) //如果接到start启动信号，就跳转到state1
             next_state <=state1;
          else
            next_state <=state0;
        end
        state1: begin //无条件跳转到state1
```

```
            next_state <= state2;
          end
        state2: begin
          if(step2) //如果输入step2有效，则跳转到state3
            next_state <=state3;
          else
            next_state <=state0;
          end
        state3: begin
          if(step3)
            next_state <=state0;
          else
            next_state <=state3;
          end
        default: next_state <=state0;//缺省状态
      endcase
  always @(state) //该进程定义FSM的输出
    case(state)
      state0: fsm_out=3'b001;
      state1: fsm_out=3'b010;
      state2: fsm_out=3'b100;
      state3: fsm_out=3'b111;
      default:fsm_out=3'b001; //default语句，避免锁存器的产生
    endcase
endmodule
```

1. **代码详解**

上述 Verilog 代码表达的是这样一个电路:(1)电路初始状态 state0 下输出 3'b001，在接收到启动信号 start 后，则进入 state1 状态,否则继续维持在本状态;(2)在 state1 状态下，直接输出一个周期的 3'b010，并立刻进入 state2 状态;(2)在 state2 状态下，输出 3'b100，并等待 step2 信号有效，如 果 step2 信号有效，则进入 state3 状态，否则继续维持在本状态;(3)在 state3 状态下，输出 3'b111， 并等待 step3 信号，如果 step3 信号有效，则回归到 state0，即初始状态，否则继续维持在本状态。

状态转移图如图 1-35 所示:

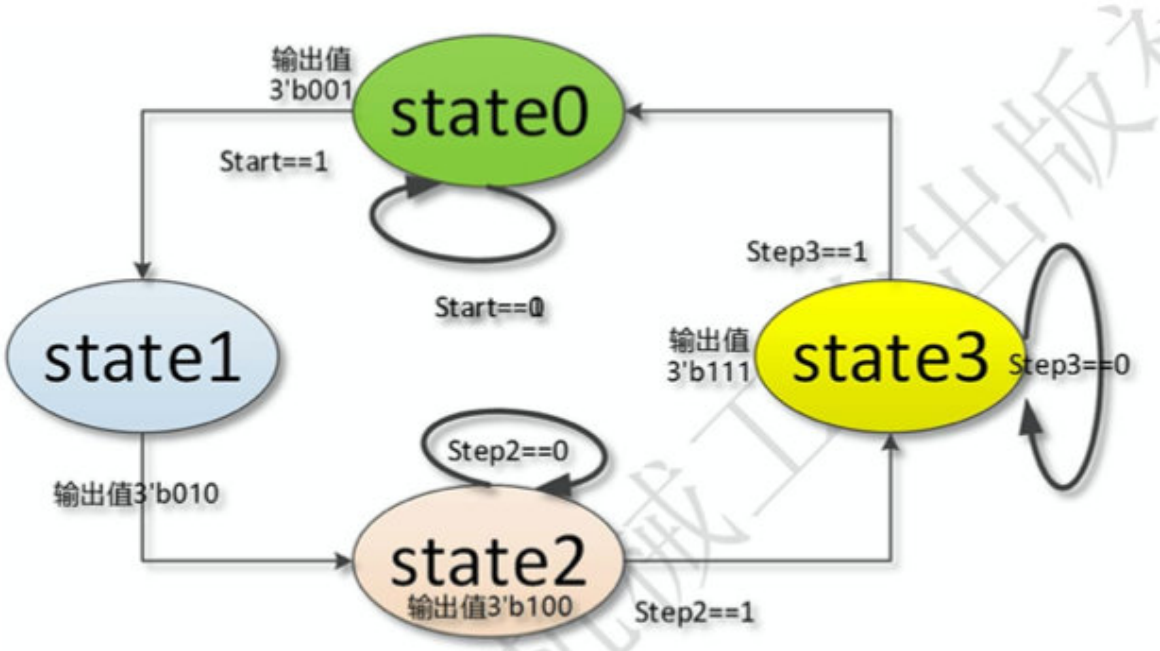


图 1-35 状态转移图

上述 state0 等的表述有点抽象，所以如果按照这种说法，读者可以更清楚的理解:这个状态机是描述 IT 码农的生活，state0 表示在家休息，而早上七点的起床闹钟就表示 start 信号，此时的状态输出值表示工作输 出强度。通常 IT 码农休息时的战斗力是较差的，所以输出为 3'b001。但闹钟响了，就必须立刻起床买早 餐挤地铁，所以跳转到 state1，即早餐状态。但早餐状态只能持续小小的一段时间，所以无条件的跳转到早 上工作状态，即 state2。通常早上的输出状态是“半血输出”，即输出为:3'b100。而上午不停的例会、评 审以及面谈，一直需要等待老板允许午饭，即 step2 信号。当 step2 信号来到后，经过中午或 tea break，就 需要进入下午的工作状态了，即 state3。下午的工作通常都是满血输出，即输出为 3'b111，而可以下班回 家的信号 step3 常常不能准时到来，有时候甚至需要通宵加班到象科比一样见到早上 5 点钟的太阳，然后回 家休息。

同样，读者可以将这个状态具体化到某个流程，例如通信接收信号的处理或者发射信号序列处理。

2. **有限状态机的状态添加**

上述例子仅包含 4 个状态，在实际应用中，状态数目可能远远超出 4 个，也经常会发生状态修改或添加的情形。所以下面给出一个如何由上面 4 个状态调整为 5 个状态的例子。该例子的改动核心包括:(1)在组合 always 块中添加一个状态，并加入如何跳转到该状态以及如何跳转出该状态即可。(2)调整状态寄存器 的位宽，并增加状态数量。

支持 5 个状态的 FSM 状态机描述代码如下:

```
module FSM(
  input clk,
  input reset,
  input start,
  input step2,
  input step3,
  input step4,
  output reg[2:0] fsm_out
);
  localparam fsm_width=3;
  localparam state0=3'b000;
  localparam state1=3'b001;
  localparam state2=3'b011;
  localparam state3=3'b010;
  localparam state4=3'b100;
  //标准三段式编码，每个周期更新当前状态
  reg[fsm_width-1:0] state;
  reg[fsm_width-1:0] next_state;
    always@(posedge clk or negedge reset)
      if(!reset)
            state <= state0;
   else
            state <= next_state;
 //根据当前状态和输入，确认下一个周期的状态
  always@(*)
    begin
        case(state)
        state0: begin
            if(start) //如果接到start启动信号，就跳转到state1
//省略部分与前面完全相同
          end
        state3: begin
          if(step3)
            next_state <=state4;
          else
            next_state <=state3;
```

```
            end
          state4: begin
            if(step4)
              next_state <=state0;
            else
              next_state <=state4;
            end
          default: next_state <=state0;//缺省状态
        endcase
    always @(state) //该进程定义FSM的输出
      case(state)
//省略部分与前面完全相同
        state4: fsm_out=3'b011;
        default:fsm_out=3'b001; //default语句，避免锁存器的产生
      endcase

endmodule
```

5. JTAG标准的状态机实现

提起 JTAG，大家的第一印象就是芯片的调试接口。JTAG 作为一项国际标准测试协议(IEEE 1149.1 兼容)，主要用于芯片内部测试和调试。目前的主流芯片均支持 JTAG 协议,如 DSP、FPGA、ARM、部分单片 机等。标准的 JTAG 接口是 20Pin，但 JATG 实际使用的只有 4 根信号线，再配合电源、地;目前常见的各 种接口形式(20pin、14pin、10pin)如下:

20 pin JTAG				14pin JTAG				10 pin JTAG			
名称	管脚号	管脚号	名称	名称	管脚号	管脚号	名称	名称	管脚号	管脚号	名称
Vcc	1	2	NC	nTRST	1	2	GND	TCK	1	2	Vcc
nTRST	3	4	GND	TDI	3	4	GND	TDI	3	4	Vcc
TDI	5	6	GND	TDO	5	6	GND	TDO	5	6	GND
TMS	7	8	GND	TMS	7	8	GND	TMS	7	8	GND
TCK	9	10	GND	TCK	9	10	GND	nTRST	9	10	GND
GND	11	12	GND	nSRST	11	12	n/a				
TDO	13	14	GND	DINT	13	14	Vcc				
NRESET	15	16	GND								
NC	17	18	GND	为防反接插头							
NC	19	20	GND								

芯片开发所用的 JTAG 口调试实物如图 1-36 所示:

图 1-36 IC 与 FPGA 的 JTAG 引脚

1. **JTAG 标准介绍**

JTAG 的基本原理是在器件内部定义一个 TAP(Test Access Port 测试访问口)通过专用的 JTAG 测试工具对内部节点进行测试。JTAG 测试允许多个器件通过 JTAG 接口串联在一起，形成一个 JTAG 链，能实现对各个器件分别测试。JTAG 引脚的定义如下:

信号名称	信号说明
TCK	TCK在IEEE1149.1标准里是强制要求的。TCK为TAP的操作提供了一个独立的、基本的时钟信号，TAP的所有操作都是通过这个时钟信号来驱动的
TMS	TMS在IEEE1149.1标准里是强制要求的。TMS信号在TCK的上升沿有效，用来控制TAP状态机的转换。通过TMS信号，可以控制TAP在不同的状态间相互转换
TDI	TDI在IEEE1149.1标准里是强制要求的。TDI是数据输入的接口,所有要输入到特定寄存器的数据都是通过TDI接口一位一位串行输入的(由TCK驱动)
TDO	TDO在IEEE1149.1标准里是强制要求的。TDO是数据输出的接口,所有要从特定的寄存器中输出的数据都是通过TDO接口一位一位串行输出的(由TCK驱动)
TRST	可选项，TRST可以用来对TAP控制器进行复位(初始化）。因为通过TMS也可以对TAP控制器进行复位(初始化），所以有四线JTAG与五线JTAG之分
RTCK	可选项,由目标端反馈给仿真器的时钟信号,用来同步TCK信号的产生,不使用时直接接地
nSRST	可选项，与目标板上的系统复位信号相连，可以直接对目标系统复位。同时可以检测目标系统的复位情况，为了防止误触发应在目标端上加上适当的上拉电阻

JTAG 标准的信号时序图如图 1-37 所示:

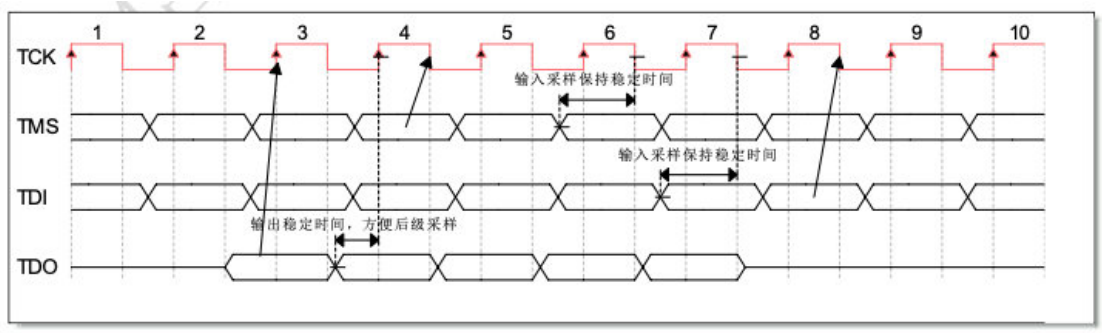


图 1-37 JTAG 时序

通过 JTAG 连接，可以完成如下的功能:

1. 对所有串接在一起的 IC 进行引脚连接性测试，确认 PCB 是否焊接正常;
2. 对 CPU、DSP、FPGA 等进行调试 debug;
3. 通过 JTAG 对 FPGA 进行编程。

进行引脚链接测试的 JTAG 用法如图 1-38 所示，各个芯片引脚联通状态可以依次串接通信到 PC 的 TDO 引脚中。

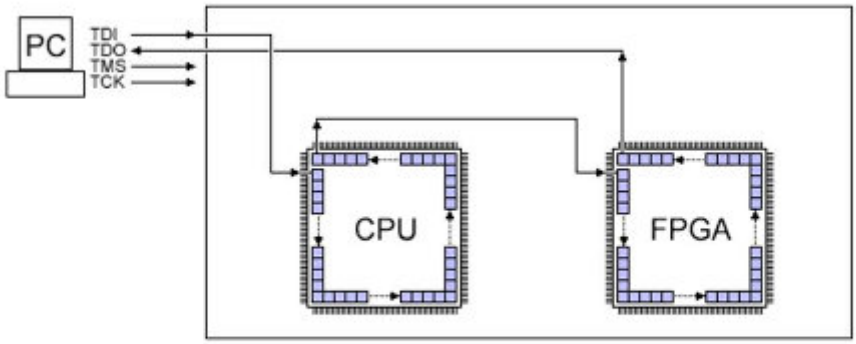


图 1-38 JTAG 的 Boundary 测试

按照菊花链方式串接调试的 JTAG 用法如图 1-39 所示，多个串接在一起的 CPU 和 FPGA 都能够在一起进行调试和测试:

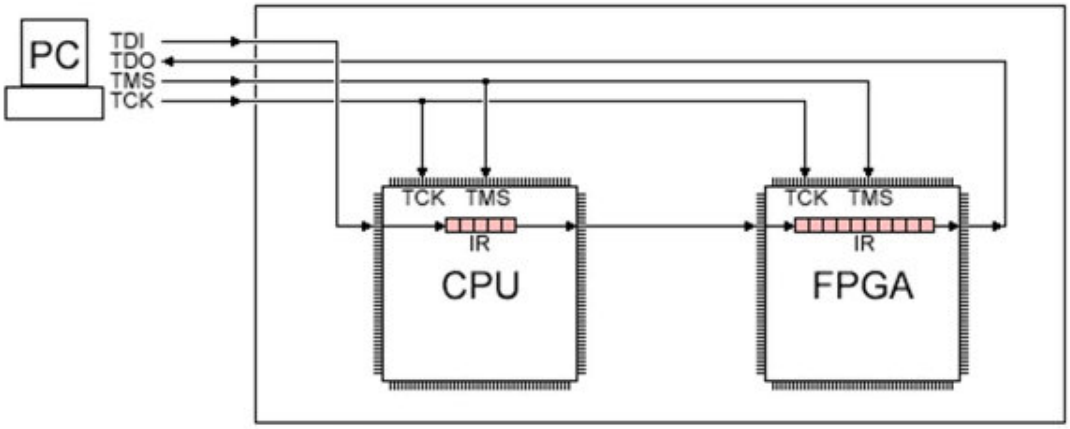


图 1-39 JTAG 的 Debug 过程

上图已经给出 JTAG 的调试原理:(1)所有的调试芯片的 IR 寄存器串接在一起，然后进行串行移位，最后所有数据都进入到 JTAG 口的 TDO 中;PC 通过对 TDO 数据串并转换，获得每个 CPU/DSP 或 FPGA 的内部寄存器状态。(2)PC 将需要写入 CPU/DSP 或 FPGA 的数据通过并串转换放置到 TDI 总线上，然后通 过状态移位到规定的 CPU 寄存器上，然后通过 TMS 指定生效时刻。(3)TCK 就是 TDI 和 TDS 的移位时 钟，而 TMS 则是控制指令。

所以 JTAG 通过一个标准状态机就能将 CPU/DSP/FPGA 的内部状态明确，也能改变内部寄存器内容，这也是一种状态控制原理。在 1990 年前，JTAG 的状态机基本是各个厂商自行定义，但后来出现的 IEEE1149.1 标准对状态转移过程进行了标准化。在某种意义上讲，JTAG 的状态机实现过程是最佳的 FSM 学习对象。

2. **JTAG状态机的设计**

JTAG 内部的状态转移图如图 1-40 所示，其中的状态值读者可以自行定义，但推荐采用 one-hot 或者格雷码进行编码:

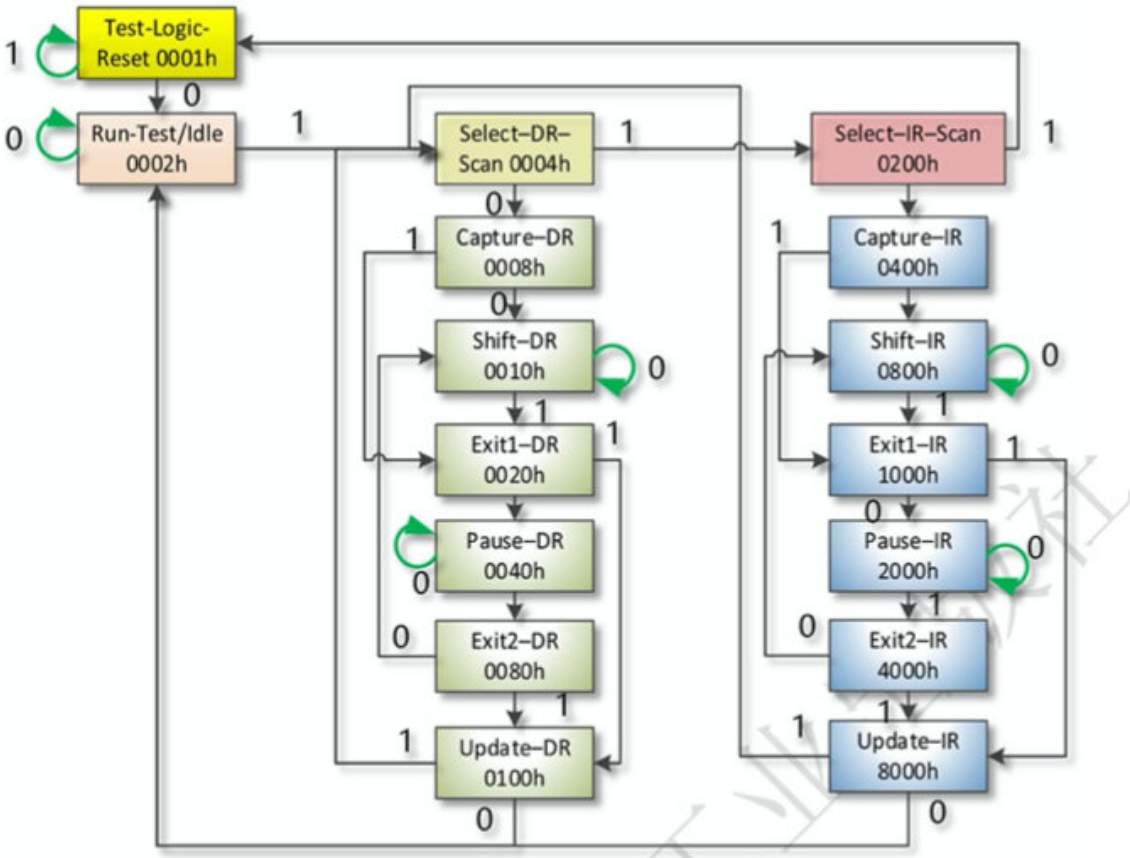


图 1-40 IEEE1149.1 规定的 JTAG 状态机

下面的代码是实现 JTAG 功能的 FSM 部分，该代码已经应用到多款 ASIC 芯片中，具有较高的研究价值。

```
//TAP FSM implementation
module tap_FSM #(parameter sync_mode = 1)(
input   tck,
input   trst_n,
input   tms,
input   tdi,
output  byp_out,
output updateIR,reset_n,

output reg  clockDR, updateDR, clockIR, tdo_en, shiftDR,shiftIR,
output  selectIR, sync_capture_en, sync_update_dr, flag,
output [15:0] tap_state
);
//Inter signal declaration
reg [15:0] state;
reg [15:0] next_s;
reg scan_out_a, scan_out_s, updateIR_a;

localparam TEST_LOGIC_RESET = 16'h0001, RUN_TEST_IDLE = 16'h0002, SELECT_DR_SCAN
= 16'H0004,
  CAPTURE_DR= 16'h0008, SHIFT_DR = 16'h0010, EXIT1_DR = 16'h0020,PAUSE_DR =
16'h0040,
  EXIT2_DR  = 16'h0080, UPDATE_DR= 16'h0100, SELECT_IR_SCAN = 16'h0200,
  CAPTURE_IR= 16'h0400, SHIFT_IR = 16'h0800, EXIT1_IR = 16'h1000,
  PAUSE_IR  = 16'h2000, EXIT2_IR = 16'h4000, UPDATE_IR= 16'h8000;

assign flag = state[10] || state[11];
```

```
wire updateIR_s = state == UPDATE_IR;
assign updateIR   = sync_mode ? updateIR_s : updateIR_a;
assign tap_state= state;

always @(posedge tck or negedge trst_n)
  if ( !trst_n )
    state<=TEST_LOGIC_RESET;
  else
    state<=next_s;

always @(*)
  case(state)
    TEST_LOGIC_RESET: if(tms)
                        next_s=TEST_LOGIC_RESET;
                      else
                        next_s=RUN_TEST_IDLE;
    RUN_TEST_IDLE: if( tms )
                     next_s=SELECT_DR_SCAN;
                   else
                     next_s=RUN_TEST_IDLE;
    SELECT_DR_SCAN: if(tms)
                      next_s=SELECT_IR_SCAN;
                    else
                      next_s=CAPTURE_DR;
    CAPTURE_DR: if(tms)
                  next_s=EXIT1_DR;
                else
                  next_s=SHIFT_DR;
    SHIFT_DR: if(tms)
                next_s=EXIT1_DR;
              else
                next_s=SHIFT_DR;
    EXIT1_DR: if(tms)
                next_s=UPDATE_DR;
              else
                next_s=PAUSE_DR;
    PAUSE_DR: if(tms)
                next_s=EXIT2_DR;
              else
                next_s=PAUSE_DR;
    EXIT2_DR: if(tms)
                next_s=UPDATE_DR;
              else
                next_s=SHIFT_DR;
    UPDATE_DR: if(tms)
                next_s=SELECT_DR_SCAN;
              else
                next_s=RUN_TEST_IDLE;
    SELECT_IR_SCAN:if(tms)
                     next_s=TEST_LOGIC_RESET;
                   else
                     next_s=CAPTURE_IR;
    CAPTURE_IR: if(tms)
                  next_s=EXIT1_IR;
                else
                  next_s=SHIFT_IR;
    SHIFT_IR: if(tms)
                next_s=EXIT1_IR;
```

```
                else
                  next_s=SHIFT_IR;
    EXIT1_IR: if(tms)
                  next_s=UPDATE_IR;
                else
                  next_s=PAUSE_IR;
    PAUSE_IR: if(tms)
                  next_s=EXIT2_IR;
                else
                  next_s=PAUSE_IR;
    EXIT2_IR: if(tms)
                  next_s=UPDATE_IR;
                else
                  next_s=SHIFT_IR;
    UPDATE_IR: if(tms)
                  next_s=SELECT_DR_SCAN;
                else
                  next_s=RUN_TEST_IDLE;
  endcase

//FSM outputs
reg rst_n;

//reg  clockDR, updateDR, clockIR, tdo_en, rst_n, shiftDR, shiftIR;
//ClockDR/ClockIR - posedge occurs at the posedge of tck
//updateDR/updateIR - posedge occurs at the negedge of tck
always @( tck or state )begin
    if ( !tck && ( state == CAPTURE_DR || state == SHIFT_DR ))
      clockDR = 0;
    else
      clockDR = 1;

    if ( !tck && ( state == UPDATE_DR ))
      updateDR = 1;
    else
      updateDR = 0;

    if ( !tck && ( state == CAPTURE_IR || state == SHIFT_IR ))
      clockIR = 0;
    else
      clockIR = 1;

    if ( !tck && ( state == UPDATE_IR ))
      updateIR_a = 1;
    else
      updateIR_a = 0;
  end

always  @( negedge tck )
  if ( state == SHIFT_IR || state == SHIFT_DR )
    tdo_en <= 1;
  else
    tdo_en <= 0;

always  @( negedge tck )
  if ( state == TEST_LOGIC_RESET )
    rst_n <= 0;
  else
```

```
    rst_n <= 1;

always @(negedge tck or negedge trst_n)
  if ( !trst_n )
    shiftDR <= 0;
  else if ( state == SHIFT_DR )
    shiftDR <= 1;
  else
    shiftDR <= 0;

always @(negedge tck or negedge trst_n)
  if ( !trst_n )
    shiftIR <= 0;
  else if ( state == SHIFT_IR )
    shiftIR <= 1;
  else
    shiftIR <= 0;

assign reset_n = rst_n & trst_n;
assign selectIR = state == SHIFT_IR;
assign sync_capture_en = ~(shiftDR | (state == CAPTURE_DR) | (state ==
SHIFT_DR));
assign sync_update_dr = state == UPDATE_DR;

always @( posedge clockDR )
  scan_out_a <= shiftDR & tdi & ~(state == CAPTURE_DR);

wire nxt_st_3 = (state == SELECT_DR_SCAN) & ~tms;
wire nxt_st_4 = ((state == CAPTURE_DR) & ~tms) || ( state == SHIFT_DR & ~tms);

reg sel;
always @(posedge tck or negedge trst_n)
  if(!trst_n )
    sel <= 0;
  else
    sel <= ~(nxt_st_3 | nxt_st_4);

wire scan_out = sel ? scan_out_s : shiftDR & tdi;
always @(posedge tck )
  scan_out_s <= scan_out & ~(state == CAPTURE_DR);

assign byp_out = sync_mode ? scan_out_s : scan_out_a;

endmodule
```

对于 JTAG 实现，还有一种非常简洁的代码描述，同样是三段式描述风格，但最重要的组合电路部分，通过手工推导，直接将 always 描述的组合电路化简为最小逻辑实现代码。

```
 module jtag_fsm1 (
 input   clk,    // Internal clock
 input   tdo_mux,// TDO before the negative edge flop
 input   bypass, // JTAG instruction=BYPASS
 input   tck,    // clock input
 input   trst_n, // optional async reset active low
 input   tms,    // Test Mode Select
 input   tdi,    // Test Data In
```

```
 output reg tdo, // Test Data Out
 output reg tdo_enb,//Test Data Out tristate enable

 output  tdi_r1,   // TDI flopped on TCK.
 output  tck_rise, // tck rate clock enable
 output  captureDR,// JTAG state=CAPTURE_DR
 output  shiftDR,  // JTAG state=SHIFT_DR
 output  updateDR, // JTAG state=UPDATE_DR
 output  captureIR,// JTAG state=CAPTURE_IR
 output  shiftIR,  // JTAG state=SHIFT_IR
 output  updateIR
 );
 reg     tck_r1,tck_r2,tck_r3;
 reg     tdi_f_local; //  local version
 wire    tdo_enb_nxt; //  D input to TDO_ENB flop
 wire    tdo_nxt; //  D input to TDO flop
 wire    itck_rise;
 wire    tck_fall;

 reg     [3:0] state; //  current state
 wire    a,b,c,d,a_nxt,b_nxt,c_nxt,d_nxt;
 assign {d,c,b,a} = state[3:0]; //a:0,b:1,c:2,d:3

 assign a_nxt=(~tms& ~c & a)|(tms &~b)|(tms & ~a)|(tms& d & c);
 assign b_nxt=(~tms& b & ~a)|(~tms&~c)|(~tms & ~d & b)|(~tms & ~d & ~a)|(tms & c
& ~b)|(tms & d & c & a);
 assign c_nxt=(c & ~b)|(c&a)|(tms &~b);
 assign d_nxt=(d & ~c)|(d&b)|(~tms&c&~b)|(~d & c & ~b &~a);

 assign tdo_enb_nxt = state == 4'b0010 | state == 4'b1010 ? 1'b1: 1'b0;
 assign captureIR = state == 4'b1110 ? 1'b1:1'b0;
 assign shiftIR = state == 4'b1010 ? 1'b1:1'b0;
 assign updateIR = state == 4'b1101 ? 1'b1:1'b0;
 assign captureDR = state == 4'b0110 ? 1'b1:1'b0;
 assign shiftDR = state == 4'b0010 ? 1'b1:1'b0;
 assign updateDR = state == 4'b0101 ? 1'b1:1'b0;
 assign tdo_nxt = (bypass ==1'b1&state == 4'b0010)?tdi_f_local:tdo_mux;
 assign tdi_r1 = tdi_f_local;

 always @(posedge clk) begin:rtck_proc
   tck_r3 <= tck_r2;
   tck_r2 <= tck_r1;  //synchronizers for edge detection
   tck_r1 <= tck;
 end
 assign tck_rise = itck_rise;
 assign itck_rise = tck_r2  &  ~tck_r3;
 assign tck_fall = ~tck_r2  &  tck_r3;

 always @(posedge clk)
   if (trst_n == 1'b0)
      state <= 4'b1111;
   else if (itck_rise == 1'b1)begin
      state <= {d_nxt, c_nxt, b_nxt, a_nxt};
   end

 always @(posedge clk)
    if (trst_n == 1'b0)
```

```
        tdi_f_local <= 1'b0;
    else if (itck_rise == 1'b1 ) begin
        tdi_f_local <= tdi;
    end

always @(posedge clk)
  if (trst_n == 1'b0)begin
        tdo <= 1'b0;
        tdo_enb <= 1'b0;
    end
    else if (tck_fall == 1'b1 ) begin
        tdo <= tdo_nxt;
        tdo_enb <= tdo_enb_nxt;
    end

endmodule // module vjtag
```

JTAG 接口除了标准 4 信号引脚外，TI 还定义了一种叫 SBW-JTAG 的接口，仅使用两根引脚(SBWTCK，SBWTDIO)即可实现 JTAG 功能，通常用于管脚受限的芯片上。ARM 的 Cortex-M 系列 CPU，均包含 SW- JTAG 定义标准。读者可以参考相关资料，编写对应的 JTAG 状态机代码。

6. 有限状态机小结

学习有限状态机的关键是理解有限状态机的运行原理，知道通过 FSM 能够将一个复杂的逻辑流程简化为有限的几个状态，所有的输出响应都是状态对应的输出。FSM 的编写风格应当采用三段式编码，并尽可能的采用 Moore 型状态机。

1.3.8 第六个Verilog程序：写testbench

前面已经给出了很多的 Verilog 电路实现例子，但如何证明上述代码实现正确?因此，本节将引入测试验证的概念。所谓测试验证，就检查编写的 RTL 是否真正完全遵守最初的设计规范(Design Specification)。 更进一步就是生成最终网表中的任何一个中间步骤是否完全满足了最初的设计规范和需求列表(feature list)。 所以，测试验证的目的是证明实现过程的正确性，而不是去证明这个 RTL 代码多么美妙或者代码多么的有 技巧。如果最初的需求设定就错误了，验证并不能纠正这个错误，而只是尽早的通过结果显示这个设定展现 的后果。

因此，验证过程是一个多次重复的过程，是一个不断向期望结果靠近的收敛过程。目前的验证主要包括功能验证和时序验证两种方法。功能验证主要是验证所设计的 RTL 代码是否符合系统规范，主要方法包括功能仿真和形式验证。而时序验证则验证综合和 Layout 后，含有延时信息的网表时序是否满足要求。通常功能仿真在 HDL 代码设计前期是最重要的，就是我们俗称的 TestBench 和各种 Case。

1. TestBench

在 Verilog 仿真验证中，设计模块(DUT)与测试模块之间的关系图如图 1-41 所示:

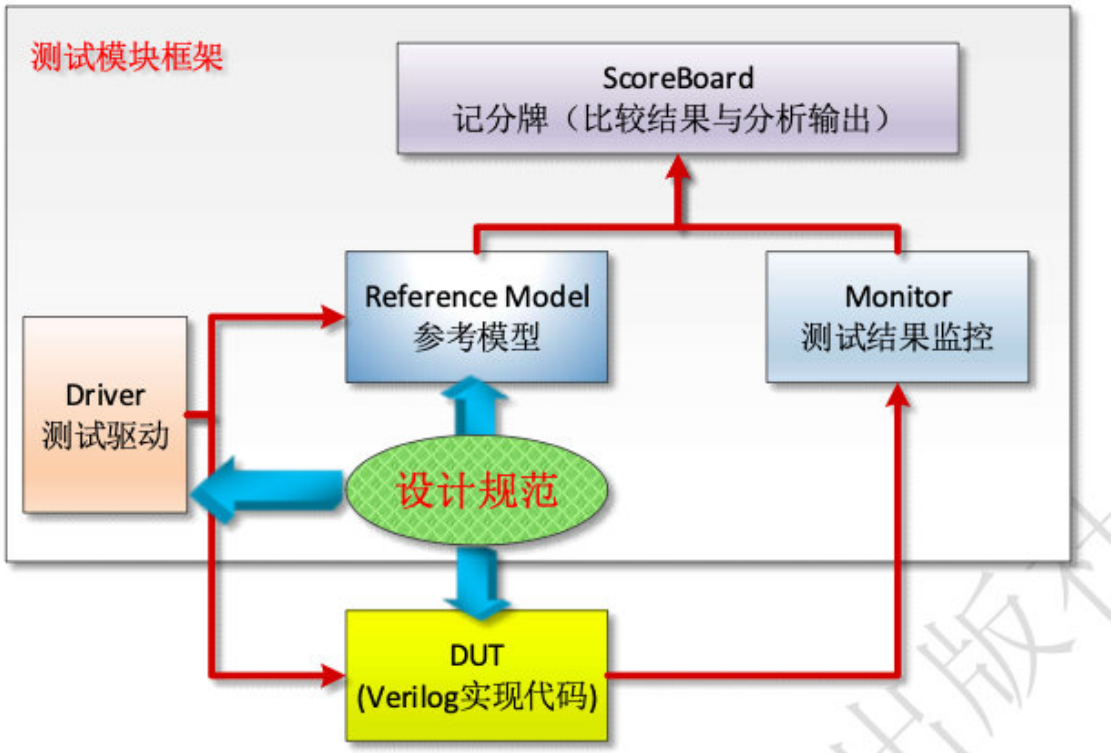


图 1-41 Testbench 的概念

上图中就是一个标准的测试验证框架。其中包含了测试驱动(driver)、理想参考模型(Golden Reference Model)、测试结果检测(Monitor)和记分牌(ScoreBoard)。

其中 Driver 将各种的激励施加给 Verilog 实现模型(DUT)和理想模型，两者将同时输出对应的激励响应。由于 HDL 获取激励响应的结果通常不太直观，所以大型设计通常都会设计一个专门的 monitor，用来监 测 DUT 的输出。此外，scoreboard，它专门的比较期望值与 monitor 监测到的 DUT 的输出，并对输出结果 做一定的处理，方便统计和得到更高层次的结果。

在读者开始接触编写 TestBench 时，一定要明确:测试的目的是为了验证所设计的 HDL 代码是否真正与设计规范一致，而不是以 RTL 代码可以运行为目的。所以写 TestBench 一定要满足完备性和充分性，所设 计的激励能够充分覆盖需求规范，并确定各种未知情况的结果。因此，Testbench 追求产生激励的便利性和 可验证性，而不关心是否能够转化为真实电路，所以 Testbench 往往采用有可重用、具备丰富组合能力的代 码，并以提高覆盖率为核心理念。

2. 计数器的测试例子

下面的例子是第三个例子中计数器的测试例子。

```
`timescale 1ns/1ns
module count_tb;
localparam DELY=100;
reg clk,reset;  //测试输入信号,定义为reg 型
wire[3:0] cnt;  //测试输出信号,定义为wire型

always#(DELY/2) clk = ~clk; //产生时钟波形
initial
begin                  //激励信号定义
  clk =0; reset=0;
  #DELY   reset=1;
  #DELY   reset=0;
  #(DELY*100) $finish(2);
end
```

```
//定义结果显示格式
initial $monitor($time," clk=%d reset=%d cnt=%d",clk, reset,cnt);

//调用测试对象
count#(.N(4))U_cnt(
 .clk   (clk  )
,.clear (reset)
,.cnt_Q (cnt  )
);

endmodule
```

其计数器代码为：

```
module count#(parameter N=8)(
input clk,
input clear,
output[N-1:0] cnt_Q
);
reg[N-1:0] cnt;
assign cnt_Q = cnt;

always@(posedge clk)
  if(clear)
    cnt <= 'h0;       //同步清 0，高电平有效
  else
    cnt <= cnt+1'b1; //减法计数

endmodule
```

程序的部分测试结果大致为:

```
clk=0 reset=0 cnt=1
clk=1 reset=0 cnt=2
clk=0 reset=0 cnt=2
clk=1 reset=0 cnt=3
clk=0 reset=0 cnt=3
```

上述测试程序有几个语法点：

1. 模块 module 的例化调用以及对应的参数设置。
2. initial 语句含义
3. 时钟波形的产生
4. 激励信号的简单产生方法
5. 测试结果的查看方法

3. module例化

上面的测试例子，有这样一条语句:

```
count#(.N(4))U_cnt(.clk(clk),.clear (reset),.cnt_Q(cnt));
```

该语句的含义是对 count 模块进行了例化，相当于在测试模块中将抽象的 count 模块变为实际的 4 位计 数器。模块例化的语法为:

```
模块名称<#(.参数名称 1(实例参数 1),...,.参数名称 n(实例参数 n))>[实例名]
    (<.端口名称 1(实例信号 1)> ,...,<.端口名称 n(实例信号 n)> );
```

如果调用多个相同模块，只需要例化多个模块，并将[实例名]修改为不同即可。例如如下用法:

```
<模块名称><#参数列表 1><实例名 1> (端口连接表 1),
<#参数列表 2><实例名 2> (端口连接表 2),
                    ......
<#参数列表 n><实例名 n> (端口连接表 n)
```

其实，模块例化是 Verilog 的三种逻辑功能描述方法之一。其余两点为:数据流描述方式(assign)和行 为描述方式。这三点的归纳小结如下:

1. **数据流描述方式**

数据流描述方式是根据信号(变量)之间的逻辑关系，采用持续赋值语句 assign 描述逻辑电路的方式， 描述数据流程的运动路径、运动方向和运动结果。assign 只能用于组合逻辑描述，而不能用于时序逻辑电路， 可以理解为一种连续赋值的方式。

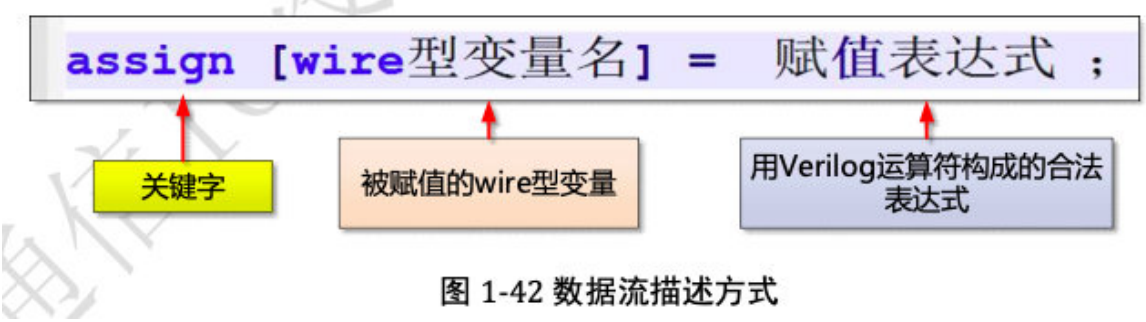


图 1-42 数据流描述方式

连续赋值的含义是:无论右边的操作数何时发生变化，右边表达式都重新计算。因此采用 assign 赋值后，只要赋值表达式中任一操作数发生变化立即对 wire 型变量进行更新操作，以保持对 wire 型变量的连续驱动，这就是组合逻辑电路的特征——任何输入的变化，立即影响输出。

只有 wire 型数据才能支持 assign 语句赋值，这是因为 wire 型变量没有数据保持能力，只有被连续驱动， 才能取得确定值。若一个连线型变量没有得到任何连续驱动时，它的取值将是不定态“x”。而寄存器型变量 只要在某时刻得到过一次过程赋值，就能一直保持该值，直到下一次过程赋值。

基于数据流描述方式的模块基本结构如下:

```
module 连续赋值模块名 ( // I/O端口列表说明
  input 输入端口列表
  output 输出端口列表
    // 数据类型说明
    wire 结果信号名;
    // 逻辑功能定义
    assign <结果信号名> =逻辑表达式 ; ...
  assign <结果信号名n>=逻辑表达式n;
endmodule
```

2. **行为描述方式**

行为描述方式关注逻辑电路输入、输出的因果关系(行为特性)，即直接描述在何种输入条件下，产生何种输出(操作)的方式，主要通过 always 和 initial 语句形式体现。行为描述相当软件设计过程中的流程图描述和算法描述，并不关心电路的内部实现结构，EDA 的综合工具能自动将行为描述转换成电路结构，形 成网表文件。行为描述的特点包括:

- 只有寄存器类型 reg 数据能够在这两种语句中被赋值。

- 行为描述方式主要用于描述时序逻辑，也可描述组合逻辑。

这里面，always 语句所描述的内容将不断地重复运行，而 initial 语句只执行一次，语句结束就结束。这正好对应了 always 语句可用于电路实现语句(可综合的)，而 initial 只能用于测试验证的特点。因为实际电 路在通电的情况下是不断运行的。

always 语句可以用如图 1-43 所示方式归纳:

图 1-43 always 的赋值语句

利用 always 进行模块行为级描述的语法为:

```
module 行为描述模块名 (
  // I/O端口列表说明
  input 输入端口列表
    output reg 输出端口列表
);
  // 数据类型说明
  reg 中间变量
  // 逻辑功能定义
  always @(敏感事件列表) //行为描述1
    begin
      if-else、case、for等行为语句
    end
  always @(敏感事件列表) //行为描述n
    begin
            .........
        if-else、case、for等行为语句
    end
endmodule
```

对于 initial 语句，其语法为:

```
initial
  begin
    语句1;

    语句2; ......
    语句n;
  end
```

所有的 initial 语句内的语句构成了一个 initial 块。initial 块从仿真 0 时刻开始执行，在整个仿真过程中只 执行一次。如果一个模块中包括了若干个 initial 块，则这些 initial 块从仿真 0 时刻开始并发执行，且每个块 的执行是各自独立的。如果在块内包含了多条行为语句，那么需要将这些语句组成一组，一般式使用关键字 begin 和 end 将他们组合在一个块语句;如果块内只有一条语句，则不必使用 begin 和 end.下面给出了 initial 语句的例子:

请注意，initial 里面的语句可以是任意合法的 Verilog 语句，并不是一些简单的初始化语句。例如以下语 句为永久执行代码，生成一个 50MHz 时钟。

```
parameter clk_period = 20;
reg clk=0; // 直接设定初始值
initial begin
  clk = 0; // 再次设定初始值，这个与reg定义初始化都有效
    forever #(clk_period/2) clk = ~clk;
end
```

3. **结构化描述方式**

通过创建多个实例，并给出这些实例之间的逻辑连接关系就是结构化描述。可以理解为:在已有很多的基本单元或元件的前提下，然后按照搭积木的方式，将各个单元之间的关系描述出来，就是结构化描述。也可以理解为如何将传统意义上的"逻辑原理图"转换为 Verilog HDL 的描述。例如如图 1-44 所示的算术逻 辑运算单元(ALU)就必须采用结构化方式描述，而组成的基本单元则可以采用前面的两种方式描述:

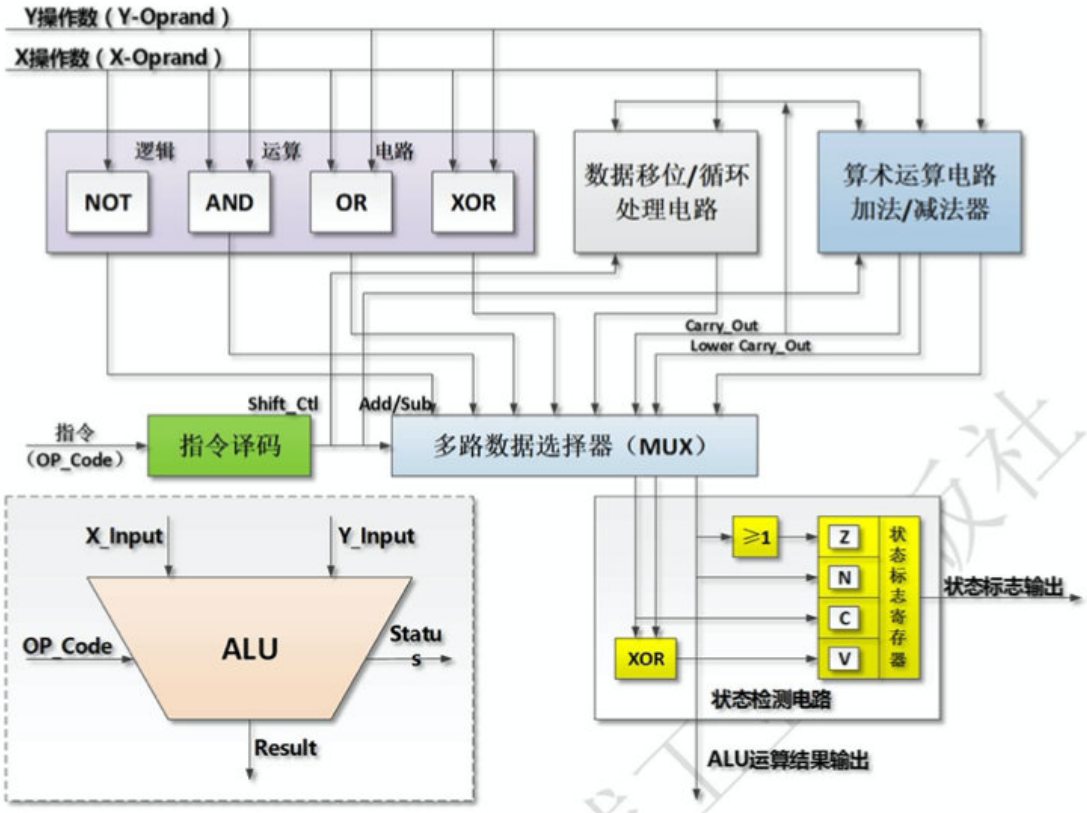


图 1-44 结构化电路描述的例子

结构化描述的基本元件包含两类:(1)基于真值表的原语(primitive);(2)模块实例。其中基于真值表的描述基本上用于仿真测试，因为原语实际是不可综合的，但读者还是需要了解原语，因为在门级仿真时会大量用到。Verilog 提供了 26 个内置基本元件，其中 12 个是门级元件，14 个为开关级元件，原语模型以结 构化方式描述逻辑。

原语类别	原语名称
多输入门	and,nand,or,nor,xor,xnor
多输出门	buf,not
三态门	bufif0,bufif1,notif0,notif1
双向开关	pullup,pulldown
MOS 开关	cmos,nmos,pmos,rcmos,rnmos,rpmos
双向开关	tran,tranif0,tranif1,rtran,rtranif0,rtranif1

利用原语进行结构化描述的例子:

```
module MUX2_1(
input a,b,sel,  //声明a,b,sel为输入端口
output out      //声明out为输出端口
);
  wire a1, b1, sel_n;  //定义内部节点信号（连线）
  not u1 (sel_n,sel ); //调用内置“非”门元件
  and u2 (a1,a,sel_n); //调用内置“与”门元件
  and u3 (b1,b,sel  ); //调用内置“与”门元件
  or u4  (out,a1,b1 ); //调用内置“或”门元件
endmodule
```

该例子表达的电路图如图 1-45 所示:

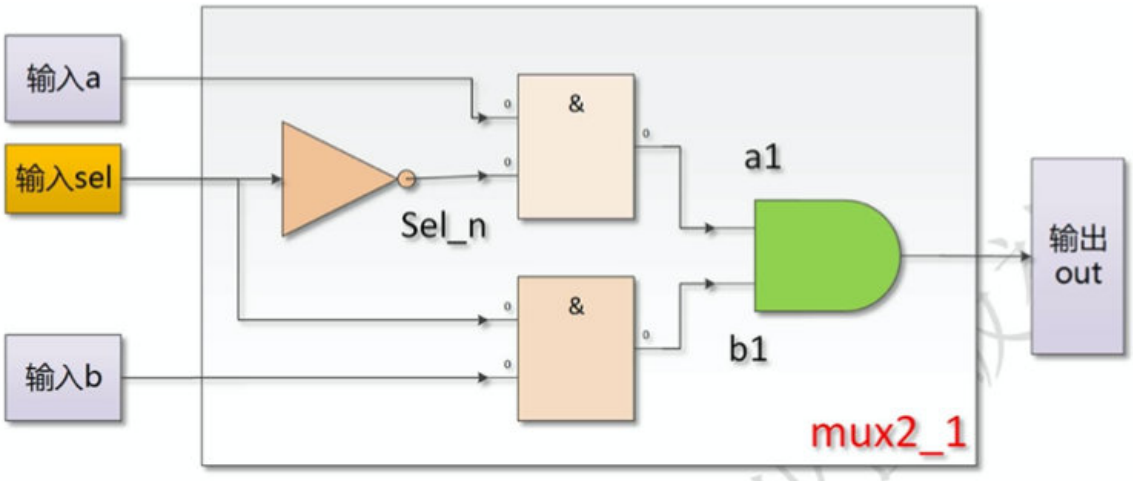


图 1-45 基于原语的结构化描述

而基于结构化的例子如下:

```
module ADD_1b(
    input a,b,cin,
    output sum,cout
);
```

```
    assign {cout,sum} = a+b+cin;
    endmodule

    //利用1bit加法器(ADD_1b)进行结构化描述
module adder_4bit(
    input [3:0] a,b,
    input cin,
    output [3:0] sum,
    output cout
);
    wire cout0; //第0位的进位信号;
    wire cout1; //第1位的进位信号;
    wire cout2; //第2位的进位信号;
    //描述逻辑功能,调用一位全加器的模块构建四位全加器
    ADD_1b A0(.a(a[0]),.b(b[0]),.cin(cin ),.sum(sum[0]),.cout(cout0));
    ADD_1b A1(.a(a[1]),.b(b[1]),.cin(cout0),.sum(sum[1]),.cout(cout1));
    ADD_1b A2(.a(a[2]),.b(b[2]),.cin(cout1),.sum(sum[2]),.cout(cout2));
    ADD_1b A3(.a(a[3]),.b(b[3]),.cin(cout2),.sum(sum[3]),.cout(cout ));
endmodule
```

该全加器的电路结构如图 1-46 所示:

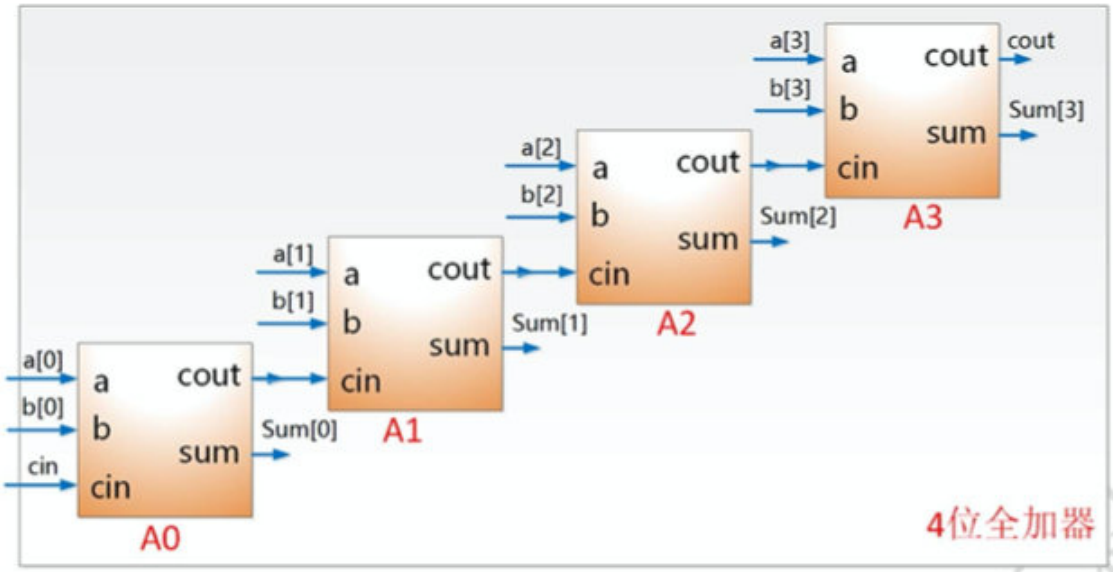


图 1-46 基于结构化的全加器

4. **三种描述方式的小结**

如果用 Verilog 模块实现一定的功能，首先应该清楚哪些是同时发生的,哪些是顺序发生的:

- assign 语句、always 块语句和元件、模块实例化描述的逻辑功能是同时执行的，也就是并发执行的，

  如果把这三项写到一个模块文件中去，它们的次序不会影响逻辑实现的功能。
- 在 always 模块(begin-end)块语句内，逻辑是按照指定的顺序执行的。always 块中的语句称为顺序 语句，而 FSM 的基础就是顺序执行。
- 两个或更多的“always”模块也是同时执行的，但是模块内部的语句是顺序执行的。

由于 always 块语句和 assign 语句是并发执行的，所以 assign 语句一定要放在 always 块语句之外。同样， initial 语句中也必然没有 assign 语句和 always 语句。

4. 激励信号的简单产生方法

激励信号通常包含两种，一类是有规律的信号，例如时钟与复位信号，或者是由某些信号逻辑生成;另外一类是无规律的信号。对于有规律的信号，只需要给出信号生成方式，就很容易在测试代码中生成。而无规律的信号则必须按照时间顺序，依次给出信号的工作行为。对于测试验证而言，有规律的信号能够验证常规的功能，而无规律的信号才能全面测试逻辑描述是否正确。

无规律的信号，通常有两种方式生成:利用随机约束生成，或者将时间行为存储在文件中，在执行时依次读出该时刻的值，然后赋予激励向量。这两点是激励信号生成的关键。

1. **有规律信号生成的例子**

下面的代码是一个标准的时钟与复位生成模块代码，其它的测试代码均通过引用方式(例如:force clk = system_init.masterclock)，获得时钟与复位。

```
//略 可参考通信IC设计  p73
```

2. **无规律信号的生成例子:文件读写**

无规律的信号，通常通过文件读写方式或随机函数获得，其实前面例子中的 syn_rst 也可以认为是一个无规律信号，只是直接在 initial 语句中，通过时间序列方式，描述了该时刻的 0/1 状态。下面的例子则是通过 文件读写方式实现文件无规律信号读取。当 enable=1 时，该模块每个时钟从文件中读取一个新的数值，并 将该值赋予输出信号 signal_out。当 enable=0 时，维持输出信号不变。

```
`timescale 1ns/1ps
`define LAST_TIME 3_000_000
`define DLY_1 1

module read_enable_signal #(
parameter signal_WIDTH=10,
parameter FILENAME="./pat/dfai.dat"
)(
input clk,
input enable,
output reg signed [signal_WIDTH-1:0] signal_out
);

integer signal_FILE;
reg signal_isNotFirstRise = 0;
reg signal_isSimulationEnd= 0;
reg signed [signal_WIDTH-1:0] tmp_sig_I;

initial begin
  signal_out=0;
#`DLY_1; signal_FILE = $fopen(FILENAME,"rb");
  if (signal_FILE ==0) begin
      $display("Error at opening file: %s",FILENAME);
      $stop;
  end else
   $display("Loading %s .........",FILENAME);
end

always @(posedge clk) begin
  signal_isNotFirstRise <=  #`DLY_1 1;
end

//-- Apply Input Vectors -----
always@(posedge clk)
  if(signal_isNotFirstRise) begin
```

```
    if ($feof(signal_FILE) != 0) begin
      signal_isSimulationEnd = 1;
      #`LAST_TIME;
      $finish(2);
    end else if(enable) begin
      if ($fscanf(signal_FILE, "%d\n", tmp_sig_I)<1) begin
        signal_isSimulationEnd = 1;
        #`LAST_TIME; $finish(2);
      end else begin
        `ifdef DATA_DEBUG
          $display("Data is %d",tmp_sig_I);
        `endif
        signal_out <=  #`DLY_1 tmp_sig_I;
      end
    end
  end
endmodule
```

3. **无规律信号的生成例子:基于随机函数**

Verilog HDL 中内置随机函数$random()。通过该函数能够生成各种符号测试要求的随机约束，从而对被测模块充分测试。$random 函数调用时返回一个 32 位的随机数，它是一个带符号的整形数

```
reg[23:0]rand;
rand=$random%50;
//产生一个在(-50,50)范围内的随机数
```

如果试图产生一个在[min, max]之间随机数，则可以采用如下的代码:

```
reg[23:0]rand;
rand = min +{$random}%(max-min+1);
```

如果产生一个随机间隔的随机数，则可以按照如下代码生成:

```
always begin
  t=$random;
  #(t) x =$random;
end
```

$random()函数仅用于测试验证，如果需要可综合的随机数，则需要研究 LSFR 等随机数生成方法，这部 分可以参见本书后面章节的内容。

4. **测试结果的存储**

下面的代码是标准的信号存储方法。当 enable=1 时，每个时钟周期对输入信号进行采样，并存储到制定 的数据文件中。当 enable=0 是，则不存储任何数据到文件中。如果需要存储多个数据，则请读者自行修改。

```
`define LAST_TIME 3_000_000
`define DLY_1 1
module write_enable_signal#(parameter signal_WIDTH=10,
parameter FILENAME="./pat/result.dat" )(
  input clk,
  input enable,
  input signed [signal_WIDTH-1:0] signal_out
```

```
);
integer signal_FILE;
reg signal_isNotFirstRise = 0;
reg signal_isSimulationEnd= 0;
reg signed [signal_WIDTH-1:0] tmp_sig_I;

initial begin
#`DLY_1;signal_FILE = $fopen(FILENAME,"wb");
if(signal_FILE ==0) begin
  $display("Error at opening file: %s",FILENAME);
  $stop;
end else
  $display("Loading %s .........",FILENAME);
end

always @(posedge clk) signal_isNotFirstRise <= #`DLY_1 1;
always@(posedge clk)
if(signal_isNotFirstRise) begin
  if(enable)begin
    if ($fwrite(signal_FILE, "%d\n", signal_out)<1)begin
      signal_isSimulationEnd = 1;
      #`LAST_TIME;
      $finish(2);
    end else begin `ifdef DATA_DEBUG
     $display("Data is %d",signal_out);`endif
    end
  end
end
endmodule
```

5. 测试结果的显示

前面的测试例子通过如下的语句完成中间仿真结果的监控:

```
initial $monitor($time," clk=%d reset=%d cnt=%d",clk, reset,cnt);
```

该例子调用了 函数，该函数具有监控和输出参数列表中的表达式或变量值的功能。每当函数参数列表中变量或表达式的值发生变化时，整个参数列表中变量或表达式的值都将输出显示，从而实现对变量的实时监控。Monitor 系列函数如下:

```
$monitor(p1,p2,...,pn);
$monitor;
$monitoron;
$monitoroff;
```

在*monitor*中，参数可以是time 系统函数，例如:

```
$monitor($time,,"rxd=%b txd=%b",rxd,txd);
```

*monitoron*和monitoroff 这两个任务的作用是通过打开和关闭监控标志来控制、监控任务$monitor 的启 动和停止，从而控制$monitor 何时发生。

6. 测试中的常见系统函数

Verilog 为测试提供了很多的函数，掌握这些函数有助于快速搭建测试框架，实现高效测试。常见函数如 下:

```
$bitstoreal,$rtoi,$display,$setup,$finish,$skew,$hold,
$setuphold,$itor,$strobe,$period,$time,$printtimescale,$time,
$timeformat,$real,$width,$realtobits,$write,$recovery
```

1. **打开文件open**

```
integer file_id;
file_id = $fopen("file_path/fiile_name")
```

2. **写入文件fwrite**

```
//$fmonitor只要有变化就一直记录
$fmonitor(file_id,"%format_char", parameter_list);
$fmonitor(file_id,"%m: %t in1=%d o1=%h",$time, in1, o1);
//$fwrite需要触发条件才记录
$fwrite(file_id,"%format_char", parameter_list);
//$fdisplay需要触发条件才记录
$fdisplay(file_id,"%format_char", parameter_list);
$fstrobe();
```

3. **读取文件fread**

```
integer file_id;
file_id = $fread("file_path/file_name","r");
```

4. **关闭文件fclose**

```
$fxlose(file_id);
```

5. **由文件设定存储器初值 readmemh 与 readmemb**

```
$readmemh("file_name", memory_name"); //初始化数据为十六进制
$readmemb("file_name", memory_name"); //初始化数据为二进制
```

6. **文件处理定位 fseek**

```
//$fseek，文件定位，可以从任意点对文件进行操作
`define SEEK_SET 0
`define SEEK_CUR 1
`define SEEK_END 2
integer file, offset, position, r;
r =$fseek(file,0,`SEEK_SET);
r =$fseek(file,0,`SEEK_CUR);
r =$fseek(file,0,`SEEK_END);
r =$fseek(file, position,`SEEK_SET);
```

7. **文件位置ftell**

```
integer file, position;
position =$ftell(file);
```

8. **文件格式化 sformat**

```
integer file, r, a, b;
reg[80*8:1]string;
file = $fopenw("output.log");
r =$sformat(string,"Formatted %d %x", a, b);
```

9. **常见函数的应用例子 1:读取数据到 memory**

```
//下面的例子是从文件中读数据到memory中，属于典型用法。
```

```
`define EOF 32'HFFFF_FFFF
`define MEM_SIZE 200_000
module load_mem;
  integer file, i;
  reg [7:0] mem[0:`MEM_SIZE];
  reg [80*8:1] file_name;
  initial begin
    file_name = "data.txt";
    file = $fopenr(file_name);
    i = $fread(file, mem[0]);
    $display("Loaded %0d entries \n", i);
    i = $fcloser(file);
    $stop;
  end
endmodule // load_mem
```

10. **常见函数的应用例子 2:自动比较结果**

```
`define EOF 32'hFFFF_FFFF
`define NULL 0
`define MAX_LINE_LENGTH 1000
module compare_result;
integer file, r;
reg a, b, expect_v, clock;
//reg [`MAX_LINE_LENGTH*8:1];
wire out;
parameter cycle = 20;
always #(cycle / 2) clock = !clock; // Clock generator

initial begin : file_block
  clock = 0;
  file = $fopen("compare.pat");
  if (file == `NULL) disable file_block;

  r = $fgets(line, MAX_LINE_LENGTH, file); // Skip comments
  r = $fgets(line, MAX_LINE_LENGTH, file);

  while (!$feof(file)) begin
    // Wait until rising clock, read stimulus
    @(posedge clock)
```

```
    r = $fscanf(file, " %b %b %b\n", a, b, expect_v);
    // Wait just before the end of cycle to do compare
    #(cycle-1)$display("%d %b %b %b %b", $stime, a, b, expect_v, out);
    $strobe_compare(expect_v, out);
  end // while not EOF

  r = $fcloser(file);
  $stop;
end // initial

DUT U_dut(.a(a),.b(b),.out(out)); //被测试的电路
endmodule // compare
```

7. Testbench 中的 task 与 fork-join

前面的测试例子只是一些基本的描述方法，但对于复杂的描述却存在一定的不足。这些不足体现为:(1)不能针对某些特定行为进行简单建模，做到“用完就扔，想用就捡起”;(2)不能显式的对某些行为指定并行化，例如让 A 和 B 并行执行。例如对 SPI/I2C/UART/PCI 等的验证时，往往只想发起一次读写过程，而且 针对不同的测试情况，读写方式还不一样，如果采用可综合的，类似过程描述的方法，实现是来非常繁琐， 甚至很难实现。

针对这些不足，verilog 中引入了一种新行为描述方法:task 以及并行化语句 fork-join。在新版的 SystemVerilog 中还针对 Verilog 这些不足，额外设计了新的 for-join 机制，也增强了 task 特性。可以认为掌 握 task、fork-join 和随机化约束，verilog 的测试就算入门了。

1. **task**

任务(task)就是一段封装在“task-endtask”之间的程序。任务是通过调用来执行的，而且只有在调用时才执 行，如果定义了任务，但是在整个过程中都没有调用它，则该任务不会被执行的。调用某个任务时可能需要 它处理某些数据并返回操作结果，所以 Verilog 中的 task 存在输入端和输出端。

1.1 **Task的定义**

```
task<任务名>; // <= task task_id;
    <端口及数据类型声明语句> // <=[declaration]
    <语句1>                                  // <= procedural_statement
    <语句2>                                // <=  procedural_statement
  .....                                    // <=
  <语句n>                                  // <=  procedural_statement
endtask                                    // <= endtask
```

其中，关键词 task 和 endtask 将它们之间的内容标志成一个任务定义，task 标志着一个任务定义结构的 开始;task_id 是任务名;可选项 declaration 是端口声明语句和变量声明语句，任务接收输入值和返回输出 值就是通过此处声明的端口进行的;procedural_statement 是一段用来完成这个任务操作的过程语句，如果过程语句多于一条，应将其放在语句块内;endtask 为任务定义结构体结束标志。

task 可以启动其他的 task，其他的 task 又可以启动别的 task，可以启动的 task 是没有限制的，只有当所 有的 task 都完成之后，控制才能返回。任务可以没有或者有多个输入输出类型的变量。

一个 task 的例子为:

```
task find_max; //任务定义结构开头，命名为 find_max
  input [15:0] x,y; //输入端口说明
    output [15:0] tmp; //输出端口说明
    if(x>y) //给出任务定义的描述语句
        tmp = x;
  else
        tmp = y;
endtask
```

task 编写的注意事项如下:

- 在第一行“task”语句中不能列出端口名称。
- 任务的输入、输出端口和双向端口数量不受限制，甚至可以没有输入、输出以及双向端口。
- 在任务定义的描述语句中，可以使用出现不可综合操作符合语句(使用最为频繁的就是延迟控制语句，例如#10ns)，但这样会造成该任务不可综合。
- 在任务中可以调用其他的任务或函数，也可以调用自身。
- 在任务定义结构内不能出现 initial 和 always 过程块。
- 在任务定义中可以出现“disable 中止语句”，将中断正在执行的任务，但其是不可综合的。当任务被中断后，程序流程将返回到调用任务的地方继续向下执行。

1.2 **Task的调用**

任务调用语句的语法形式如下:

```
task_id (端口1,端口2,...,端口n);
```

其中 task_id 是要调用的任务名，端口 1、端口 2，...是参数列表。参数列表给出传入任务的数据(进入任务的输入端)和接收返回结果的变量(从任务的输出端接收返回结果)。在调用任务时，需要注意以下几点:

- 任务调用语句只能出现在过程块内。
- 任务调用语句和一条普通的行为描述语句的处理方法一致。
- 当被调用输入、输出或双向端口时，任务调用语句必须包含端口名列表，且信号端口顺序和类型必须和任务定义结构中的顺序和类型一致。
- 任务的输出端口必须和寄存器类型的数据变量对应。
- 可综合的 task 只能实现组合逻辑，也就是说调用可综合任务的时间为 0。而在面向仿真的任务中可以带有时序控制，如多个时钟周期或时延，因此面向仿真的任务调用时间不为 0。

```
`define TIMESLICE 25
module bus_wr_rd_test();
reg clk,rd,wr,ce;
reg [7:0]  addr,data_wr,data_rd;
reg [7:0]  read_data;
// Clock Generator
initial begin : clock_Generator
   clk = 0;
   forever #(`TIMESLICE) clk = !clk;
end

initial begin
  read_data = 0;
  rd = 0;
  wr = 0;
  ce = 0;
  addr = 0;
```

```
  data_wr = 0;
  data_rd = 0;
  // Call the write and read tasks here
  #1 cpu_write(8'h55,8'hF0);
  #1 cpu_write(8'hAA,8'h0F);
  #1 cpu_write(8'hBB,8'hCC);
  #1 cpu_read (8'h55,read_data);
  #1 cpu_read (8'hAA,read_data);
  #1 cpu_read (8'hBB,read_data);
  repeat(10)@(posedge clk);
  $finish(2);
end

// CPU Read Task
task cpu_read;
input [7:0]  address;
output [7:0] data;
begin
  $display ("%g CPU Read @address : %h", $time, address);
  $display ("%g  -> Driving CE, RD and ADDRESS on to bus", $time);
  @ (posedge clk);
  addr = address; ce = 1; rd = 1;
  @ (negedge clk);
  data = data_rd;
  @ (posedge clk);
  addr = 0; ce = 0;  rd = 0;
  $display ("%g CPU Read  data              : %h", $time, data);
  $display ("======================");
end
endtask
// CU Write Task
task cpu_write;
input [7:0]  address;
input [7:0] data;
begin
  $display ("%g CPU Write @address : %h Data : %h",$time, address,data);
  $display ("%g -> Driving CE, WR, WR data and ADDRESS on to bus",$time);
  @ (posedge clk);
  addr = address; ce = 1; wr = 1;
  @ (posedge clk);
  data_wr = data;
  @ (posedge clk);
  addr = 0;  ce = 0;  wr = 0;
  $display ("======================");
end
endtask

// Memory model for checking tasks
reg [7:0] mem [0:255];
always @ (addr or ce or rd or wr or data_wr)
if (ce) begin
  if (wr) begin
    mem[addr] = data_wr;
  end
  if (rd) begin
    data_rd = mem[addr];
  end
end
```

```
endmodule
```

对于上面的 CPU 读写时序以及类似的 UART/SPI/I2C/PCI/USB 等总线接口时序，早期的验证方法都是提 供一些来的 task 进行总线交互，而现在则有一个专有名称:BFM(Bus Function Model)。

BFM 的作用是将低层总线的时序封装起来，对高层提供一个调用接口，使得高层不用关心低层的实现细 节，专注于 testcase 的设计。这一点类似 C++里面向对象的概念，在 C++里，对象相当于命令或调用，而对 象的成员函数实现具体的功能，外部无须关心类内部的细节。所以 Task 的核心也是尽量将底层细节封装， 对高层仅提供调用接口，实现验证的层次化与抽象化。

1.3 **Task与function的区别**

Task 与 function 都是两种常见的执行调用对象，初学者很容易混淆，所以将两者的区别列表如下:

差异项	task 特点	function 特点
1	任务可以有 input、output 和 inout，数量不限;	函数只有 input 参数，且至少有一个 input
2	任务可以包含时序控制(如延时等)	函数不能包含任何延迟,仿真时间为0
3	任务可以用 disable 中断;	函数不允许 disable、wait 语句;
4	任务可以通过 I/O 端口实现值传递;	函数名即输出变量名，通过函数返回值;
5	任务可以调用其他任务和函数;	函数只能调用其他函数，不能调用任务
6	任务可以定义自己的仿真时间单位;	函数只能与主模块共用一个仿真时间单位;
7	任务能支持多种目的，能计算多个结果 值，结果值只能通过被调用的任务的输 出端口输出或总线端口送出;	函数通过一个返回一个值来响应输入信号的值;
8		函数中不能有 wire 型变量。

2. **并行化的 fork-join**

Fork-join 与 begin-end 都属于块语句的一种，主要用于将两条或多条语句组合在一起，使其在格式上看更像一条语句。begin-end 内部的语句是串行执行的，而 fork-join 内部的语句是并行化执行的，当 fork 语句中所有分支进程中止后，该语句块结束。先看这样一个例子:

```
fork begin
    $display("First Block\n");
  #20ns;
  end
    begin
        $display("Second Block\n");
        #30ns;
    @eventA;
  end
join
```

这里面包含两个 begin-end 块，这两个块是并行执行的。而 begin-end 内部则是串行执行。整个语句执行 完毕需要等待@eventA 完成才能完成，因为第一个块在 20ns 就结束了。

fork-join 的语法如下:

```
par_block ::=
    fork [:block_identifier] {block_item_declaration} {statement_or_null}
    join_keyword [: block_identifier]
join_keyword ::= join //verilog-2001
join_keyword ::= join | join_any | join_none //systemverilog
```

join 的关键字在 systemverilog 中增加了两个 join_any join_none，主要用于并行进程的增强。这几个关 键字的区别在于:

关键字	含义
join	父进程会阻塞直到这个分支产生的所有进程结束。
join_any	父进程会阻塞直到这个分支产生的任意一个进程结束。
join_none	父进程会继续与这个分支产生的所有进程并发执行。在父线程执行一条阻塞语句之前，产生的进程不会启动执行。

下面这个例子是一个完整的 fork-join 例子，读者可以充分理解 fork 语句中的并行与串行。

```
`define TIMESLICE 20
module fork_join_test(output reg a, b, c, d, e, f);
reg clk=0;
initial begin : clock_Generator
  forever #(`TIMESLICE) clk = !clk;
end

initial $monitor($time,,,"a=%b,b=%b,c=%b,d=%b,e=%b,f=%b",a,b,c,d,e,f);
initial begin
  a = 0;
  b = 0;
  c = 0;
  d = 0;
  e = 0;
  f = 0;
```

```
  repeat(5)@(posedge clk);
  $finish(2);
end

always @(posedge clk)
fork
  #2 a = ~a;
  #2 b = ~b;
  begin
    #2 c = ~a;
    #2 d = ~b;
    #2 e = ~c;
  end
  #2 f = ~e;
join

endmodule
```

输出结果如下：

```
  0  a=0,b=0,c=0,d=0,e=0,f=0
 22  a=1,b=1,c=0,d=0,e=0,f=1
 26  a=1,b=1,c=0,d=0,e=1,f=1
 62  a=0,b=0,c=1,d=0,e=1,f=0
 64  a=0,b=0,c=1,d=1,e=1,f=0
 66  a=0,b=0,c=1,d=1,e=0,f=0
102  a=1,b=1,c=0,d=1,e=0,f=1
104  a=1,b=1,c=0,d=0,e=0,f=1
106  a=1,b=1,c=0,d=0,e=1,f=1
142  a=0,b=0,c=1,d=0,e=1,f=0
144  a=0,b=0,c=1,d=1,e=1,f=0
146  a=0,b=0,c=1,d=1,e=0,f=0
```

其实，fork-join 在验证中更多是这样的用法，请读者注意:

```
fork
  <task 1> // 并行任务1
  <task 2> // 并行任务2
  ....
  <task N> // 并行任务N
  begin
        #watchdogtime disable <task N> // 并行任务N监控
        Action XXXX //错误后续处理
  end
join
```

8. testbench中存储数据波形

前面论述的内容是如何输出关键信号，如何存储关键数据的方法，属于规模测试验证的手段，但对于 Verilog 的 debug 过程还不够直观，因为无法确切的了解 Verilog 代码仿真中各个关联信号是如何作用的。 Verilog 的确可以像 C/C++一样启动 Debug 模式，针对每行代码进行调试。但请注意，由于 Verilog 是并行执 行的，而仿真是采用 delta 时间逐步并行推进的，采用代码调试较为困难，所以常常需要存储全部或部分仿 真数据，这就是波形文件。

业界硬件将这种波形文件存储格式标准化了，各个 EDA 厂商也都支持各类开放数据标准，所以任何一个仿真过程只需要添加以下代码，就可以自动选择某种波形存储格式进行查看了整个工程的波形。

```
`define dump_level 10
module dump_task;
initial begin
  #1;
  `ifdef VCS_DUMP
    $display("Start Recording Waveform in VPD format!");
    $vcdpluson();
    $vcdplustraceon;
  `endif

  `ifdef FSDB_DUMP
    $display("Start Recording Waveform in FSDB format!");
    $fsdbDumpfile("dump.fsdb");
    $fsdbDumpvars(`dump_level);
  `endif

  `ifdef NC_DUMP
    $recordsetup("dump", "version=1", "run=1","directory=.");
    $recordvars("depth=6");
  `endif

  `ifdef VCD_DUMP
    $display("Start Recording Waveform in VCD format!");
    $dumpfile("dump.vcd");
    $dumpvars(`dump_level);
  `endif
end
endmodule
```

上述波形格式中，fsdb 格式最常用，VCD 则是任何一个仿真器默认支持的。

9. 图形方式验证与HDL描述验证

很多读者在开始设计测试程序时，喜欢采用直观的波形图输入方式对设计进行验证，尤其是 FPGA 设计人员。采用波形图方式具备一定的快捷优势，入门相对简单，但对于大型设计时无法想象的，因为这种方式 违背了测试验证的核心理念:以提高覆盖率为目的。波形图方式验证的缺点如下:

1. **波形图只能提供极低的功能覆盖率**

画波形图无法产生复杂的激励，因此它只是产生极其有限的输入，从而只能对电路的极少数功能进行测试;而 Testbench 以语言的方式描述激励源，容易进行高层次的抽象，以产生各种激励源，轻松提供很高的 功能覆盖率。

2. **画波形图无法实现验证自动化**

对于大规模的设计来说，仿真时间很长，长时间通过波形图来观察将几乎不可能检查出任何错误;而 Testbench 是以语言的方式进行描述的，能够很方便地实现对仿真结果地自动比较，并以文字的方式报告错 误。

3. **画波形图难以定位错误**

用画波形图进行仿真是一种原始的黑盒验证法，无法使用新的验证技术，例如基于 SystemVerilog 的 OVM/UVM;而 Testbench 可以通过在内部设置观测点，或者使用断言等技术，快速地定位问题

4. **画波形图的可重用性差**

由于波形数据属于手工设计，自动化输入存在很大困难，因此可重用性很差。而且基于波形的激励属于仿真工具依赖型输入，如果切换平台，原有波形图需要重新设计;但如使用 Testbench，只需进行一些小的 修改就可以完成一个新的测试平台，提高验证效率。

5. **画波形图的验证效率低**

基于波形图输入的验证过程，仿真速度相对基于 HDL 的测试验证要慢一个数量级。 所以，在设计中除了简单的设计外，建议读者采用基于 HDL 标准测试框架进行测试验证。

10. Testbench 小结

一个结构良好的 testbench 有助于提高验证代码的可重用性，减少 testbench 设计的工作量。所以读者在编 写测试代码时，一定注重测试激励的产生、如何存储最终结果、如何比较结果、如何判别验证充分这几点。 掌握如何写系统任务，如何利用 fork-join 并发启动，如何简单观测数据，并掌握如何存储数据波形。

1.3.9第七个 Verilog 程序: SPI总线

SPI(Serial Peripheral Interface,串行外围设备接口)总线，是一种基于时钟同步基础上的高速全双工的串行 通信总线。SPI 通常只包含四根信号线，芯片引脚占用数量很少，基本上所有的 SOC 芯片均包含 1 个或多 个 SPI 接口。而且 SPI 芯片能够让读者了解同步时钟在传输中的重要作用，并通过对后面异步通信 UART 的对比，能够进一步理解时钟同步与非同步下，串行通信应当如何设计。

1. SPI 总线的系统构架

SPI 总线采用主从模式(Master Slave)架构;支持多 slave 模式应用，但实际芯片通常仅支持单 Master。SPI 的总线结构如图 1-47 所示:

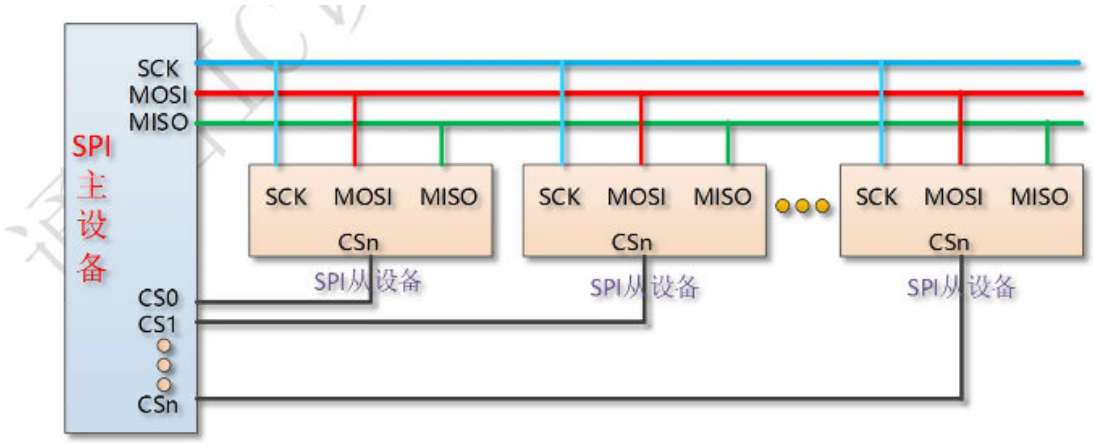


图 1-47 SPI 总线结构

2. SPI 总线信号

SPI 接口共有4根信号线，分别是:设备选择线、时钟线、串行输出数据线、串行输入数据线，具体见图 1-48:

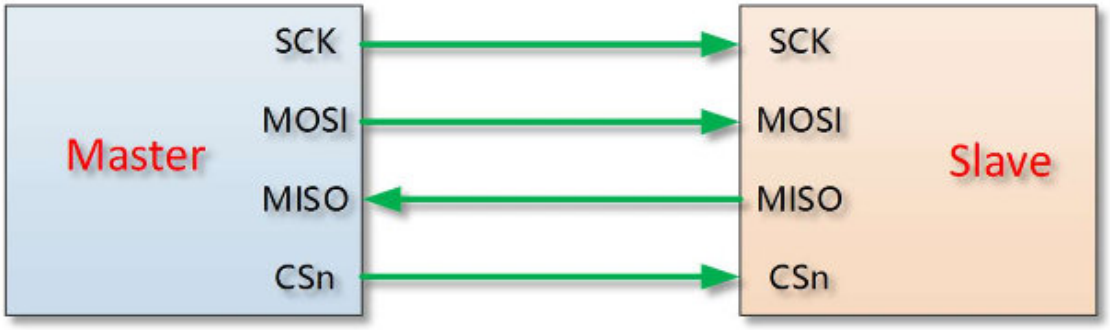


图 1-48 SPI 总线连接关系图

SPI 的引脚信号为:

1. SCLK(Serial Clock):时钟信号，SCK是串行时钟线，作用是Master向Slave传输时钟信号，控制数据交换的时机和速率;
2. MOSI(Master Out Slave in):在SPI Master上也被称为Tx-channel，作用是SPI主机给SPI从机发送数据;
3. MISO(Master In Slave Out):在SPI Master上也被称为Rx-channel，作用是SPI主机接收SPI从机传输过来的数据;
4. CSn:作用是SPI Master选择与哪一个SPI Slave通信，低电平表示从机被选中(低电平有效);

3. 时钟极性和时钟相位

在 SPI 操作中，最重要的两项设置就是时钟极性(CPOL 或 UCCKPL)和时钟相位(CPHA 或 UCCKPH)。 时钟极性设置时钟空闲时的电平(时钟为高电平还是低电平)，时钟相位设置读取数据和发送数据的时钟沿(上升沿还是下降沿)。这两项设置能够组成 4 种配置，具体组合示意如图 1-49 所示:

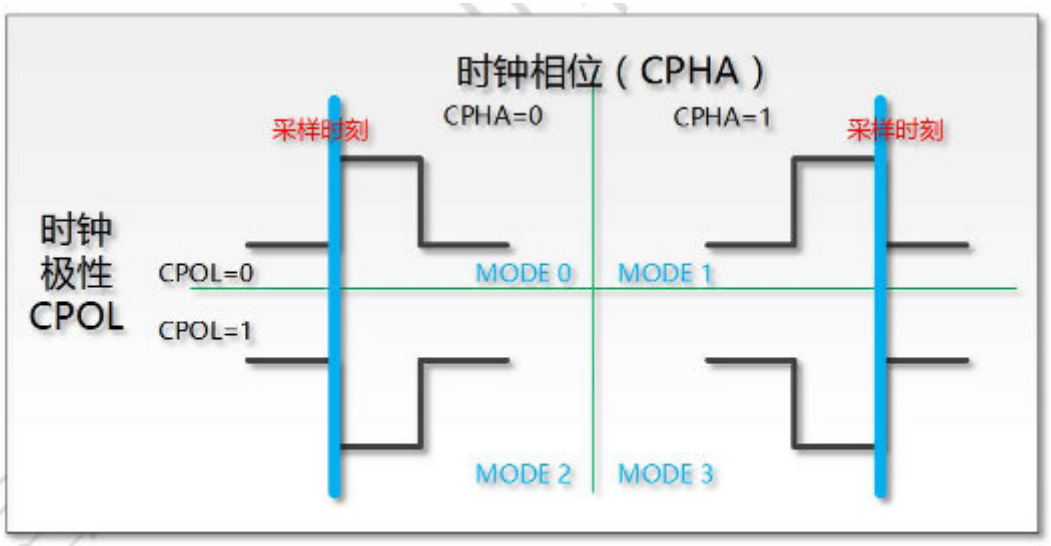
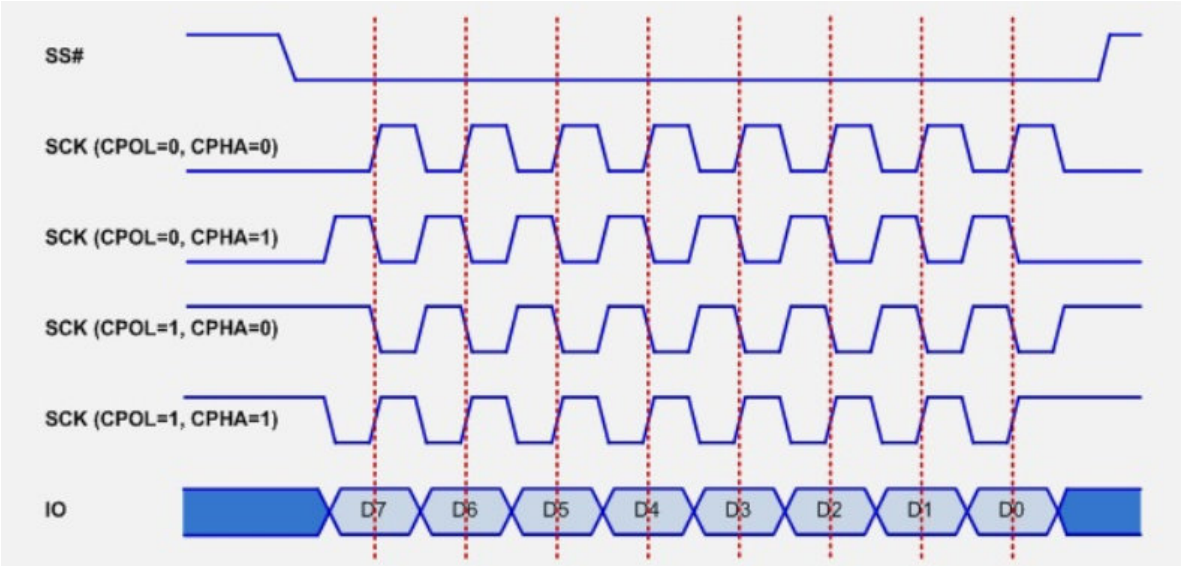


图 1-49 时钟相位与时钟极性的组合

对于 SPI 总线而言，主机和从机的发送数据是同时完成的，两者的接收数据也是同时完成的。所以为了保证主从机正确通信，应使得它们的 SPI 具有相同的时钟极性和时钟相位。下面将简单描述 SPI 的时序:

SPI总线传输一共有4中模式:



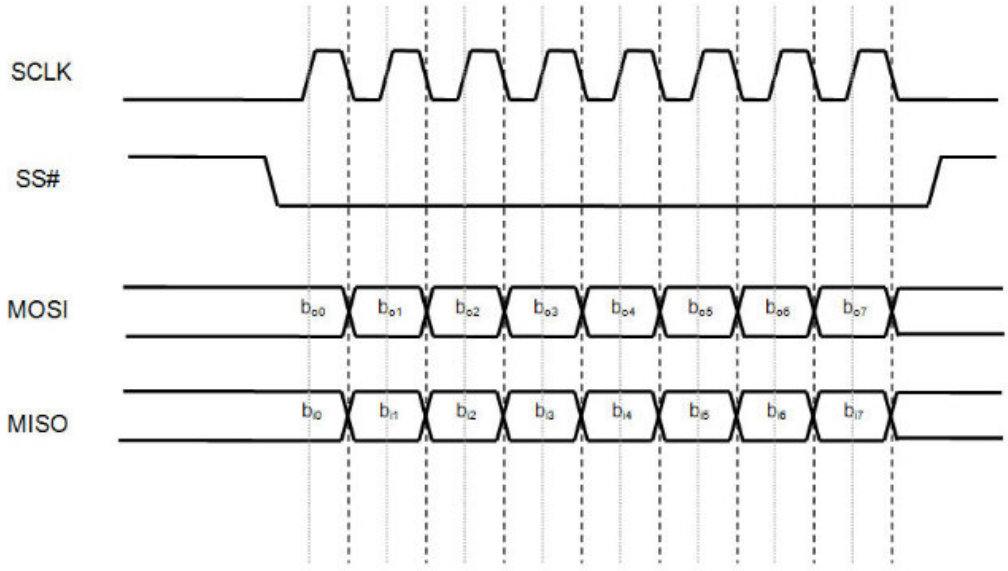
模式0：CPOL= 0，CPHA=0。SCK串行时钟线空闲是为低电平，数据在SCK时钟的上升沿被采样，数据在SCK时钟的下降沿切换

模式1：CPOL= 0，CPHA=1。SCK串行时钟线空闲是为低电平，数据在SCK时钟的下降沿被采样，数据在SCK时钟的上升沿切换

模式2：CPOL= 1，CPHA=0。SCK串行时钟线空闲是为高电平，数据在SCK时钟的下降沿被采样，数据在SCK时钟的上升沿切换

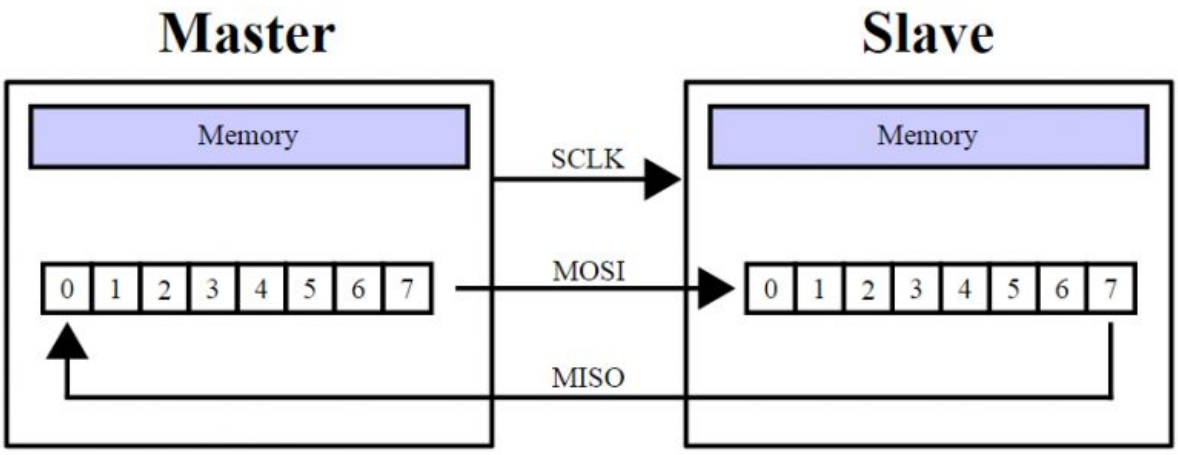
模式3：CPOL= 1，CPHA=1。SCK串行时钟线空闲是为高电平，数据在SCK时钟的上升沿被采样，数据在SCK时钟的下降沿切换

其中比较常用的模式是模式0和模式3。为了更清晰的描述SPI总线的时序，下面展现了模式0下的SPI时序图



4. SPI内部结构

从上面的时序可以看出，SPI 的主从设备内部均为一个 8bit 的移位寄存器。所以 SPI 的实现结构如图 1-54 所示:



5. SPI主设备代码

下面的代码是 SPI Master 设备的核心状态机代码，完整的代码请参考电子附件。SPI 每输入 1 个 bit，状 态机将跳转一个状态，读者可以发现 SPI 的 SCCK 时钟与模块的工作时钟，移位时钟是不同的信号。这段 代码的设计核心是根据SPI输入时钟的上升沿i_sck ==1'b1& d_sck ==1'b0，作为存储SPI总线数据的采 样时刻，即 CPHA 的相位为0。

```
 always @(posedge sysclk or negedge rst_x)
   if(!rst_x)
     state <= IDLE;
   else if (enable == 1'b1)
     state <= next_state;
   else
     state <= IDLE;
localparam[3:0]
IDLE=0,D7=1,D6=2,D5=3,D4=4,D3=5,D2=6,D1=7,D0=8,FINAL_CYCLE=9,LOAD_SHIFT_REG=10;

 always @(state or tx_data_ready or i_sck or d_sck)
 begin
   clear_tx_data_ready = 1'b0;
   shift_enable = 1'b0;
   tx_shift_reg_load = 1'b0;
   i_ss = 1'b1;
   sck_enable = 1'b0;
   load_rx_data_reg = 1'b0;
   next_state = IDLE;
   case (state)
     IDLE :if (tx_data_ready == 1'b1 & i_sck == 1'b1 & d_sck == 1'b0)
       begin
         clear_tx_data_ready = 1'b1;
         next_state = LOAD_SHIFT_REG;
       end
       else
         next_state = IDLE;
     LOAD_SHIFT_REG : begin
       tx_shift_reg_load = 1'b1;
       if (d_sck == 1'b0)begin
         i_ss = 1'b0;
         if (i_sck == 1'b1)
           next_state = D7;
         else
           next_state = LOAD_SHIFT_REG;
       end
       else
```

```
        next_state = LOAD_SHIFT_REG;
    end
    D7 :begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D6;
      end
      else
        next_state = D7;
    end
    D6 : begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D5;
      end
      else
          next_state = D6;
    end
    D5 :begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D4;
      end
      else
        next_state = D5;
    end
    D4 :begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D3;
      end
      else
        next_state = D4;
    end
    D3 :begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D2;
      end
      else
        next_state = D3;
    end
    D2 :begin
      i_ss = 1'b0;
      sck_enable = 1'b1;
      if (i_sck == 1'b1 & d_sck == 1'b0)begin
        shift_enable = 1'b1;
        next_state = D1;
```

```
        end
        else
          next_state = D2;
      end
      D1 :begin
        i_ss = 1'b0;
        sck_enable = 1'b1;
        if (i_sck == 1'b1 & d_sck == 1'b0)begin
          shift_enable = 1'b1;
          next_state = D0;
        end
        else
          next_state = D1;
      end
      D0 : begin
        i_ss = 1'b0;
        sck_enable = 1'b1;
        if (i_sck == 1'b1 & d_sck == 1'b0)begin
          shift_enable = 1'b1;
          next_state = FINAL_CYCLE;
        end
        else
          next_state = D0;
      end
      FINAL_CYCLE :if (d_sck == 1'b1)begin
          i_ss = 1'b0;
          next_state = FINAL_CYCLE;
        end
        else begin
          load_rx_data_reg = 1'b1;
          i_ss = 1'b1;
          if (tx_data_ready == 1'b1 & i_sck == 1'b1)begin
              clear_tx_data_ready = 1'b1;
              next_state = LOAD_SHIFT_REG;
          end
          else
            next_state = IDLE;
        end
endcase
```

6. SPI 从设备代码

SPI 的从设备可以按照主设备状态机的方法进行状态机设计，详情请参见电子附件。如果对 SPI 的工作频率要求不高，可以采用如下代码作为 SPI 从设备接口代码;在 ASIC 设计中建议使用电子附件中的代码， 因为下面的代码对很多异常情况未做处理，仅适合对可靠性要求不高的场合。该代码的核心是将 SPI 中的 主 SCK 作为时钟，利用该信号对总线数据进行采样或给出信号。

```
module mini_SPI_Slave(
  input clk,
  input SCK, SSEL, MOSI,
  output MISO,
  // SPI 数据接口
  input [7:0] tx_data,
  output reg[7:0] rx_data
);
// 对 SCK 进行3拍采样，消除不定态干扰
```

```
reg [2:0] SCK_d;
always @(posedge clk) SCK_d <= {SCK_d[1:0], SCK};
wire SCK_risingedge  = (SCK_d[2:1]==2'b01);  // now we can detect SCK rising
edges
wire SCK_fallingedge = (SCK_d[2:1]==2'b10);  // and falling edges
// 对 SSEL_d 进行3拍采样，消除不定态干扰
reg [2:0] SSEL_d;
always @(posedge clk) SSEL_d <= {SSEL_d[1:0], SSEL};
wire SSEL_active = ~SSEL_d[1];  // SSEL is active low
wire SSEL_startmessage = (SSEL_d[2:1]==2'b10);  // message starts at falling
edge
wire SSEL_endmessage = (SSEL_d[2:1]==2'b01);  // message stops at rising edge
// 对 MOSI 进行3拍采样，消除不定态干扰
reg [1:0] MOSI_d;
always @(posedge clk) MOSI_d <= {MOSI_d[0], MOSI};
wire MOSI_data = MOSI_d[1];
// SPI 中间变量，用于数据存储和计数
reg [2:0] bitcnt;
reg byte_received;  // high when a byte has been received
reg [7:0] byte_data_received;
reg [7:0] byte_data_tx;
always @(posedge clk)
begin
  if( SSEL_active) begin
    if(SSEL_startmessage) byte_data_tx <= tx_data;
  end
  if(~SSEL_active) begin
    bitcnt <= 3'b000;
    byte_data_tx <= 8'h00;
  end
  else if(SCK_risingedge)begin
    bitcnt <= bitcnt + 3'b001;
    byte_data_received <= {byte_data_received[6:0], MOSI_data};
    byte_data_tx <= {byte_data_tx[6:0], 1'b0};
  end
end

always @(posedge clk) byte_received <= SSEL_active && SCK_risingedge &&
(bitcnt==3'b111);

always @(posedge clk) if(byte_received) rx_data <= byte_data_received;

assign MISO = byte_data_tx[7];  // send MSB first

endmodule
```

1.3.10 第八个Verilog程序： 异步uart

UART(Universal Asynchronous Receiver/Transmitter)通用异步收发器，它是一种通用串行数据总线，用于异步通信，俗称串口通信。UART 最低采用两路信号(TX/RX)即可实现同时收发(全双工)，协议实现简单。 因此在嵌入式系统中，UART 应用广泛;而在低端 SOC 芯片中，UART 是标配实现，高端 SOC 芯片则基本 上用 USB(Universal Serial Bus)代替 UART。UART 的通信原理实际蕴含了无线通信的几个基本原则，读 者可以通过前后对比细心体会。

UART 在实际应用中的外观形态如图 1-55 所示，真正有用的信号是 RXD 与 TXD 两个引脚，其余只是作为辅助引脚，可有可无。

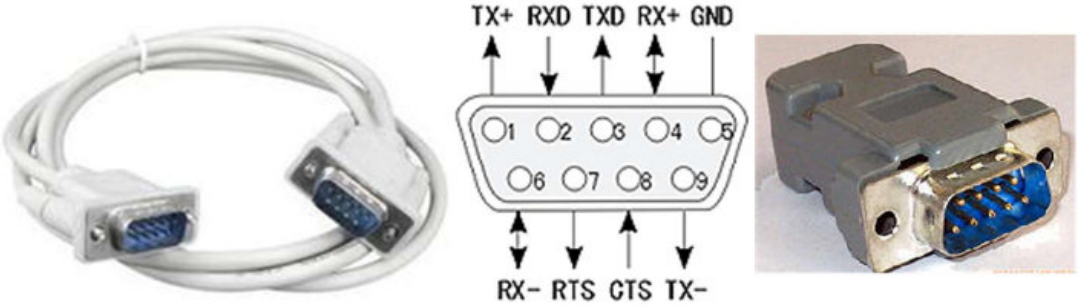


图 1-55 串口外观

1. UART传输格式

异步通信时，UART 发送/接收数据的传输格式如图 1-56 所示:

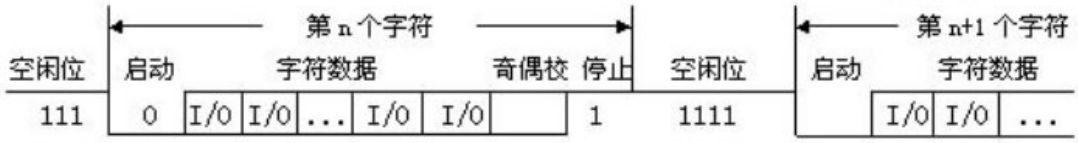


图 1-56 UART 工作时序

UART 设备在不发送数据时，数据信号线上总是呈现高电平状态，称为空闲状态(又称 MARK 状态)。当有数据发送时，信号线变成低电平，并持续一位的时间，用于表示发送字符的开始，该位称为起始位，也称 SPACE 状态。起始位之后，在信号线上依次出现待发送的每一位字符数据，并且按照先低位后高位的顺序逐位发送。待发送的 bit 数可以选择 5、6、7 或 8 位，而数据位的后面可以加上一位奇偶校验位，也可以选择不加。最后传送的是停止位，一般选择 1 位、1.5 位或 2 位。例如下面的信号将传输 0x55 信号。

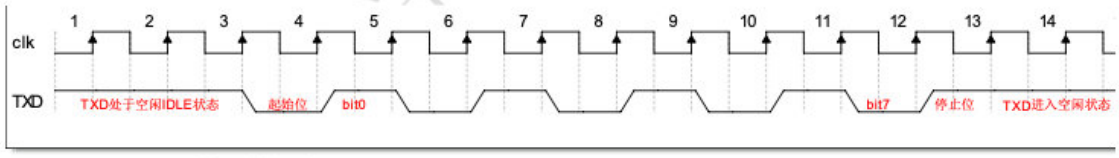


图 1-57 UART 的传输信号例子

图 1-57 是 UART 传输信号的例子，其中 0x55 的二进制数为 01010101，由于 UART 传输是采用 LSB 模式(bit 0 最先传输)，所以传输线信号的切换过程为:1-0-1-0-1-0-1-0。

2. UART 设计思想

UART 通信协议已经蕴含了最朴素的数字通信思想:通过协议规定收发双方的行为，通过约定动作获得 收发同步;通过冗余实现容错传输。具体实现思路如下:

1. 通过初始状态位的变换(下降沿)，通知接收方进入通信状态，收发双方能够有一个基本同步过程。
2. 通过预留 1bit 的起始位，接收方有充分的时间启动状态机，并能够准确判断上一次启动是否是误触发。
3. 后面通过事前约定，双方按照一定速率传输 5~8 个 bit 数据;读者可以设想如果一次传输 100bit 是否 可行?
4. 当传输完毕后，发送方将初始状态复位，从而通知接收方发送完毕，接收方也可以据此判断是否当前接收是否是误触发。
5. 通过选择停止位，保证接收方的状态机能够顺利复位到初始状态。

UART 最大一次传输 8bit 而不是更多的原因，在于收发双方并不是时钟同源的，并不像 SPI 系统有一个主时钟供主从共享。设想如果收发双方时钟偏差超过 5%，则双方对同一数据序列采样经过 20 个时钟周期 后，就会出现一个样点差异。这种情况必然导致数据统计错误。而 UART 设计标准就是允许收、发两端的 频率偏差在 10%以内都能正确通信，所以当接收 8 个 bit 数据后，收发双方的误差刚好控制在 1 个 bit 内，对数据的采样不会出错。

3. UAR的内部结构

根据前面的 UART 工作原理分析，可以将 UART 按照如图 1-58 所示的结构进行设计:

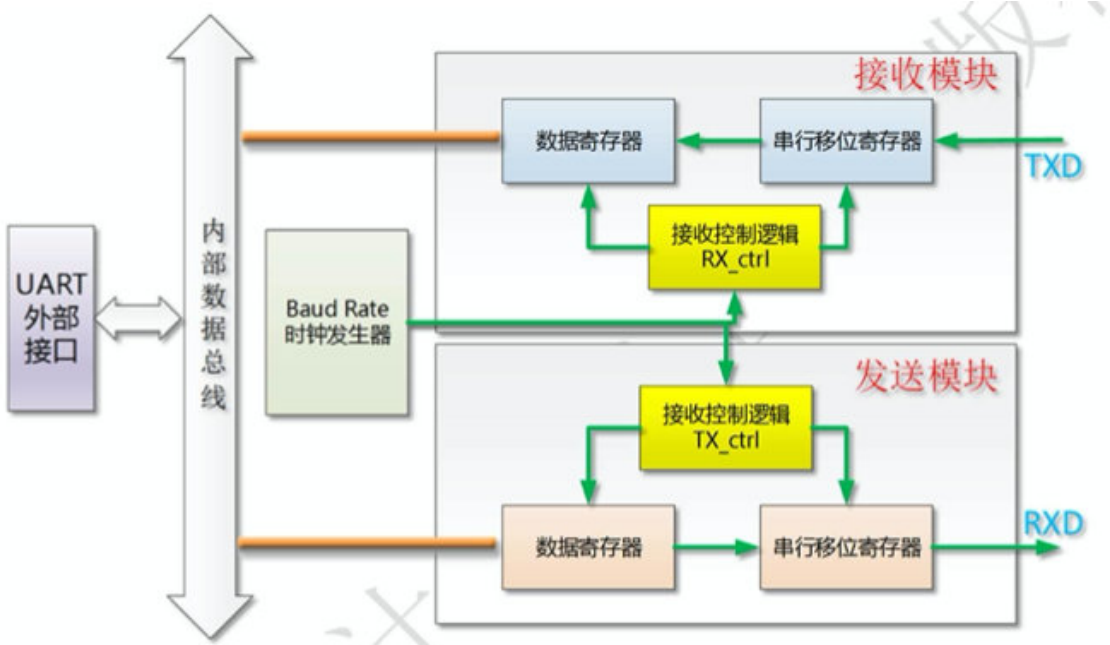


图 1-58 UART 内部实现结构

该方案中，UART 硬件由波特率时钟发生器、发送模块和接收模块组成。

1. **波特率寄存器**

波特率是单位时间内传送的二进制数据的位数，以位/秒(bps)表示，也称为比特率。目前最常用的波特率是 9600bps 和 115200bps，其余速率可以通过收发双方约定实现。假定当前时钟频率为□，而波特率为□□□□， 则相当于一个 bit，可以采样□ = □/□□□□个样点。

但由于收发双方是异步的，双方的 bit 对齐位置会逐渐漂移，所以接收方需要找到最佳采样(判断)位置。而实际上最佳采样位置就是发送数据的中心位置，即 N/2 处采样最佳。UART 的接收状态机就是通过判 断当前样点序列是否等于 N/2，作为存储当前接收值的依据。

图 1-59 是 8 倍采样的情形，可以发现最佳采样位置就是第 4 或第 5 个位置:

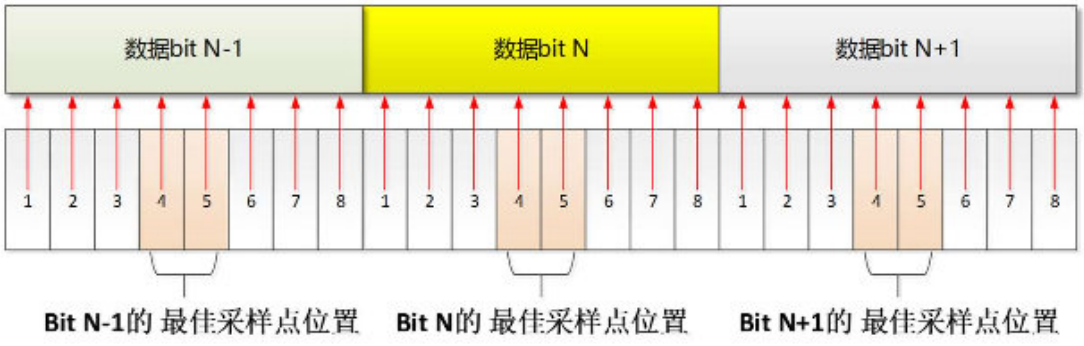


图 1-59 串口的最佳位置

在最早的 8086 芯片中，为了兼顾速度和稳定性，采样倍数一般取为 16。而在 IC 设计中，可以通过设定N，实现任意波特率的速率。而这个 N 寄存器就是波特率寄存器。而收发的通信基准时钟电路就可以认为是 一个 N 倍分频电路。

2. **发送模块**

UART 的发送原理为:

- 根据外部指令，将8bit数据加载到UART的发送FIFO或发送寄存器上。
- 如果当前发送进入空闲状态，则字符数据被加载到发送控制模块。
- 发送控制模块将字符数据被转换串行数据流输出，同时加入起始位，奇偶校验位，停止位。
- 每N个时钟，将串行数据流的1个bit发送到UART的发送引脚(TXD)上。

3. 接收模块

UART 的接收原理为:

- 在接收系统开始工作以后将启动接收进程，并直接进入空闲状态。在空闲状态始终检测起始位，即判断接收信号是否 1 跳变为 0。
- 当检测到有效起始位后，以约定波特率的时钟开始接收数据，并进入接收数据状态。
- 在数据接收状态，当计数器统计接收位数达到规定的bit数后，如果存在奇偶校验则进入奇偶校验状态，否则直接进入停止位状态
- 在奇偶校验状态下，如果校验位检测无误，则进入接收停止位状态。
- 停止位接收完毕后，将接收数据转存到数据寄存器中，并返回到初始空闲状态。

在 UART 接收过程中，有几个基本的电路状态判断技巧:

- 起始位的判断:在接收端开始接收数据位之前，通常需要搜索起始位。接收端通常以 N 倍波特率的速率读取线路状态，检测线路上出现低电平的时刻。因为异步传输的特点是以起始位为基准同步的，但通信线路噪声也极有可能使信号由“1”跳变到“0”。所以接收器以 N 倍的波特率对这种跳变进行 检测，直至在连续 N/2 个接收时钟以后采样值仍然是低电平，才认为是一个真正的起始位，而不是噪声引起的，其中若有一次采样得到的为高电平则认为起始信号无效，返回初始状态重新等待起始信号的到来。
- 最佳采样点:最可靠的接收应该是接收时钟的出现时刻正好对着数据位的中央，即前面提到的第 N/2 个采样位置。
- 合法 UART 接收状态的判断:当采样计数器计数结束后，所有数据位都已经输入完成。最后需要对停止位的高电平进行检测，若正确检测到高电平，说明本帧各 bit 位传输格式正确，可以将数据转存 到数据寄存器中;否则说明本帧接收有误，定时关系出错。

采用有限状态机模型可以更清晰明确地描述接收器的功能，便于代码实现。接收模块的状态转换图如下所示。接收器状态机由 5 个工作状态组成，分别是空闲状态、起始位确认、采样数据位、停止位确认和数据正确，触发状态转换的事件和在各个状态执行的动作见图 1-60 的文字说明。本例省略了奇偶校验的情况。

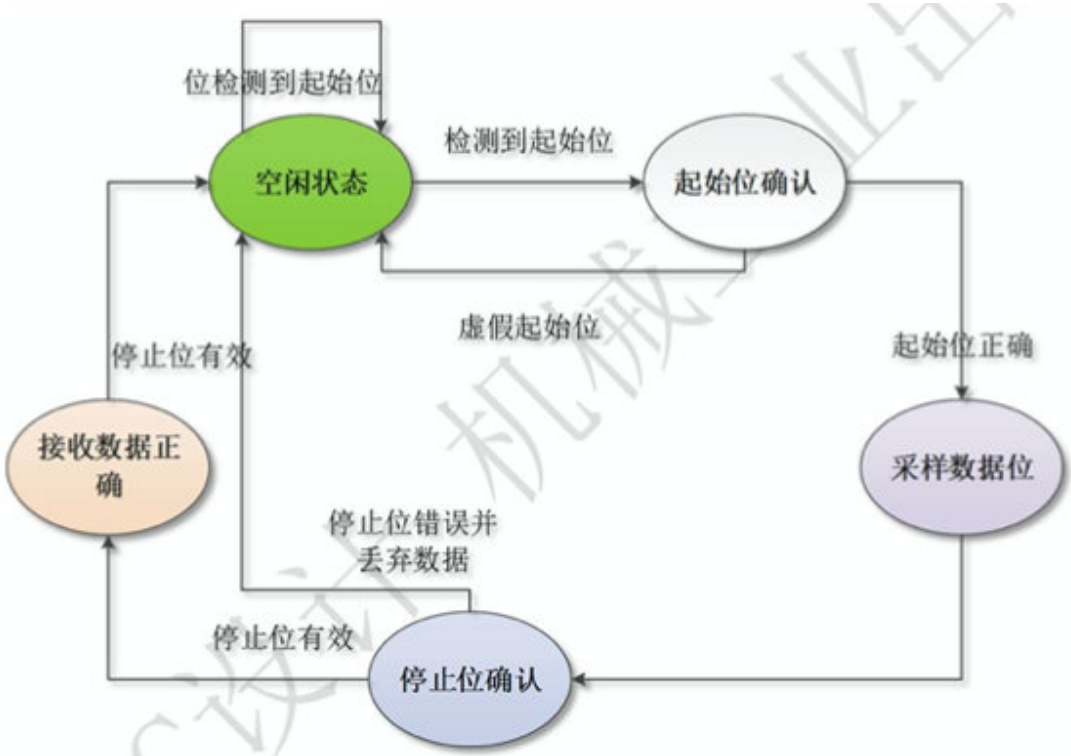


图 1-60 UART 的接收状态机

4. UART的接收实现代码

下面的代码是 UART 的接收实现代码，由于采用状态机描述代码长度较长，所以此处直接用一个计数器 rxnum 代替状态机状态。需要注意的是串行输入 i_clk_rx 需要在顶层进行 1 到 2 个节拍的缓存，以消除不定态。而 i_clk_rx 则是数据采样指示，应当在接收波特率计数器计数到 N/2 时，形成一个高脉冲。

```
`define P_EVEN 2'b00 //
`define P_ODD  2'b01 //
`define P_NONE 2'b10 //

module uart_rx(
  input i_clk,          //系统时钟输入
  input i_rst_n,        //复位输入
  input i_enable_n,     //允许传输输入
  input i_int_clrxn,    //清除中断输入
  input i_clk_rx,       //移位时钟输入，与波特率一致
  input i_rx_data,      //串行数据输入
  input [1:0] i_parity,//校验模式(奇校验、偶校验、无校验)
  output o_rx_int,      //接收中断输出
  output [7:0] o_data, //数据输出
  output o_err          //传输出错标志位
);

reg [7:0] rxdata;
reg [3:0] rxnum;

reg [1:0] rxparity;
reg rxerr;
reg rxstart;
wire rx_start = rxstart & ~i_enable_n ;
wire none_parity_finish=(rxnum==9&&rxparity == `P_NONE);
wire parity_finish=(rxnum==10&&(rxparity==`P_EVEN||rxparity==`P_ODD));
```

```
always @(posedge i_clk)
  if(!i_rst_n | rxerr) begin  //复位或传输发生错误则终止传输
    rxparity <= `P_NONE;
    rxstart <= 0;
  end else if(i_clk_rx && !i_rx_data && rxnum == 0) begin //收到起始位
    rxparity <= i_parity ;//每次开始传输的时候确认校验方式
    rxstart<= 1;
  end else if(i_rx_data&&(none_parity_finish|parity_finish))begin
    rxparity <= `P_NONE;
    rxstart <= 0;
  end


always @ (posedge i_clk)
  if(!i_rst_n) begin
    rxnum <= 0;
    rxdata <= 0;
    rxerr <= 0;
  end else if(rx_start && i_clk_rx) begin
    rxnum <= rxnum + 1;
    case (rxnum)
      0 : begin  rxdata[0] <= i_rx_data ; rxerr <= 0 ; end
      1 : rxdata[1] <= i_rx_data ;
      2 : rxdata[2] <= i_rx_data ;
      3 : rxdata[3] <= i_rx_data ;
      4 : rxdata[4] <= i_rx_data ;
      5 : rxdata[5] <= i_rx_data ;
      6 : rxdata[6] <= i_rx_data ;
      7 : rxdata[7] <= i_rx_data ;
      8 : case(rxparity)//核对校验位、若无校验则直接核对停止位
          `P_EVEN : rxerr <=  ^rxdata ^ i_rx_data ;
          `P_ODD  : rxerr <=  ^rxdata ^ i_rx_data ;
          `P_NONE : rxerr <= ~ i_rx_data ;
          default : rxerr <= ~ i_rx_data ;
        endcase
      9 : rxerr <= ~ i_rx_data | rxerr ;//检测停止位
    endcase
  end else if(none_parity_finish|parity_finish) begin
    rxnum <= 0;
  end

assign o_err = rxerr ;

reg [7:0] data;
reg rx_int;
always @ (posedge i_clk)
  if(!i_rst_n) begin
    data <= 0;
    rx_int <= 0;
  end else if(none_parity_finish|parity_finish) begin
    if(!rxerr) begin
      data <= rxdata;
      rx_int <= 1;
    end else begin
      data <= 'bz;
      rx_int <= 0;
    end
  end else if(!i_int_clrxn) begin //手动清除中断标志
```

```
    rx_int <= 0;
    data <= 'bz;
  end

assign o_rx_int = rx_int ;
assign o_data = data ;

endmodule
```

5. UART的发送实现代码

UART 的发送代码与接收代码类似，同样也是通过一个计数器计数依次给出每个 bit 的状态。i_clk_rx 是发送状态指示时钟，每启动一个新 bit 发送，该信号将产生一个高脉冲，时钟频率与发送波特率完全一致。 发送状态机由 txnum 控制，可以通过 txnum 得出当前该发送的位置。

```
`define P_EVEN 2'b00 //
`define P_ODD  2'b01 //
`define P_NONE 2'b10 //
module uart_tx(
input i_clk,          // 系统时钟输入
input i_rst_n,        // 复位输入
input i_clk_tx,       // 移位时钟输入，与波特率一致
input i_start_n,      // 启动uart发送
input [7:0] i_data,   // 数据输入，在i_start_n有效时载入
input [1:0] i_parity,// 校验模式(奇校验、偶校验、无校验)
output o_tx_data,     // 串行数据输出
output o_tx_int);     // 发送完成中断

reg [7:0] txdata;
reg [1:0] txparity;
reg [3:0] txnum;
reg txstart;
// 写内部寄存器进程
// Process generates start signal for transmission
always @(posedge i_clk) begin
  if(!i_rst_n) begin
    txdata <= 0;
    txstart<= 0;
    txparity <= 0;
  end else if(!i_start_n) begin
    txdata <= i_data;
    txparity <= i_parity;
    txstart <= 1;
  end else if(txnum == 11) begin
    txstart <= 0;
    txparity<= 'bx;
    txdata  <= 'bx;
  end
end

// 逻辑连接，在i_start_n 有效电平消失后才开始传输
wire tx_start = txstart &  i_start_n ;
reg txd;
always @(posedge i_clk)
  if(!i_rst_n) begin
    txd <= 1;
```

```
    txnum <= 0;
  end else if(i_clk_tx && tx_start) begin
    txnum <= txnum + 1;
    case (txnum)
      0 : txd <= 0;   // 发送起始位0
      1 : txd <= txdata[0];
      2 : txd <= txdata[1];
      3 : txd <= txdata[2];
      4 : txd <= txdata[3];
      5 : txd <= txdata[4];
      6 : txd <= txdata[5];
      7 : txd <= txdata[6];
      8 : txd <= txdata[7];
      9 : case(txparity)
          `P_EVEN : txd <= ~^ txdata ;// 偶校验 txdata各位同或输出
          `P_ODD  : txd <=  ^ txdata ;// 奇校验 txdata各位异或输出
          `P_NONE : txd <= 1;          // 无校验 提前发送停止位
          default : txd <= 1 ;
        endcase
      10: txd <= 1 ; // 发送停止位1
      default : txd <= 1;
    endcase
  end else if(txnum == 11)
    txnum <= 0;

assign o_tx_data = txd ;

// 传输中断产生进程
// Process generates interrupt signal for output
reg tx_int;
always @(posedge i_clk)
  if(!i_rst_n || (tx_start && txnum != 11))
    tx_int <= 0;
  else if( tx_start && txnum == 11)
    tx_int <= 1;

assign o_tx_int = tx_int ;

endmodule
```

1.2.11 一些有用的verilog程序

1. 跑马灯

下面的这个程序是跑马灯的程序,读者可以通过调节参数N设定灯的个数,通过TMEOUT设定跑马的快慢。

```
module led#(parameter N=4,parameter TIMEOUT=32'hfff_ffff)(
  input clk,
  input reset,
  output reg[N-1:0] led //
);
reg[31:0]cnt;
always@(posedge clk)
  if(reset==1'b1)begin
    cnt<='h0;
```

```
    led<='b1; //
  end else begin
    cnt<=cnt+1'b1;
    if(cnt==TIMEOUT)begin
      led = led<<1;                          // 右移跑马,但利用非阻塞赋值为后面判断提
供提前量
      if(led==4'b0000) led='b1;      //完成一轮跑马,从头再来
    end
  end

endmodule
```

2. 双向端口例子

当某个模块端口为 inout类型时,该端口同时具备输入和输出功能。通常在ASIC设计中， 只有在最顶层文件(IOPAD文件)中进行pin引脚设定时才会使用 inout类型;而对于FPGA, 则是直接体现在顶层文件中。使用双向端口的目的是减少引脚数量。

通常 inout需要通过三态门来实现,三态门的第三个状态就是高阻(Z)。当 inout端口不输出时,将三态门置高阻,这样信号就不会因为两端同时输出而出错了。要使用 inout类型数 据,可以用如下写法:

下面的代码则是一个完整的双向端口例子:

```
module bidir_data#(parameter N=8)(
  input en,
  input clk,
  inout[N-1:0] bidir
);
reg[N-1:0] temp;
assign bidir= en ? temp : 8'bz;
always@(posedge clk)
  if(en) temp=bidir;
  else   temp=temp+1;
endmodule
```

编写测试模块时,对于 inout类型的端口,需要定义成wire类型变量,而其他输入端口都定义成reg类型,这两者是有区别的。

方法一:在激励模块中生成一个与测试信号对应的反相 inout端口,这等效于在两个模块之间采用 inout双向口互连。

```
module test();
  wire bidir;
  reg data_reg;
  reg inout_ctr;
  initial begin
    .........
  end
  assign bidir = inout_ctr?data_reg:1'bz;
endmodule
```

方法二:使用 Verilog特有的 force和 release语句。其中, force语句能够强迫任意信号赋予规定的值,也等效于将该信号短接为指定的值;而 release则可释放该信号,让其恢复原有的赋值。 force与 release是编写测试程序的一大利器,能够很快捷地实现某些验证意图。

```
module test();
  wire bidir;
  reg clk, en;
  ......
  bidir_data DUT(en, clk, bidir);
  initial begin
    #xx
    force DUT.bidi =1'bx;
    #xx
    release DUT.bidir
  end
endmodule
```

3. JTAG的双向端口

下面这个例子是ARM cortex-M1的JTAG包装程序，因为原始的M1代码没有提供标准的JTAG引脚，只提供了多个单向引脚。

```
module CM1_JTAG(
  input          HCLK,  // AHB总线时钟
  input          rst_x, // 板级全局异步复位信号
  // JTAG inputs
  input          nTRST,       // JTAG reset
  input          TCK,         // JTAG clock
  input          TDI,         // JTAG data in
  inout          TMS,         // JTAG mode select
  // JTAG outputs
  output         TDO,         // JTAG data out
  output         nTDOEN,      // JTAG data enable
  output         RTCK,        // JTAG data out
  // Cortex-M1标准信号
  input          CM1_LOCKUP,
  input          CM1_HALTED,
  input          CM1_JTAGNSW,
  input          CM1_JTAGTOP,
  input          CM1_TDO,
  input          CM1_nTDOEN,
  input          CM1_SWDO,
  input          CM1_SWDOEN,
  input          CM1_SYSRESETREQ,
  output         CM1_SYSRESETn,
  input          CM1_DBGRESTARTED,
  output         CM1_EDBGRQ,
  output         CM1_DBGRESTART,
  output         CM1_DBGRESETn,
  output         CM1_nTRST,
  output         CM1_SWCLKTCK,
  output         CM1_SWDITMS,
  output         CM1_TDI
);
reg rst_x_d0,rst_x_d1,rst_x_d2;
always @(posedge HCLK)
begin
  rst_x_d0<=rst_x;
  rst_x_d1<=rst_x_d0;
  rst_x_d2<=rst_x_d1;
end
```

```
reg CM1_SYSRESETREQ_d0,CM1_SYSRESETREQ_d1,CM1_SYSRESETREQ_d2;
always @(posedge HCLK or negedge CM1_DBGRESETn)
  if(!CM1_DBGRESETn) begin
    CM1_SYSRESETREQ_d0<=0;
    CM1_SYSRESETREQ_d1<=0;
    CM1_SYSRESETREQ_d2<=0;
  end else begin
    CM1_SYSRESETREQ_d0<=CM1_SYSRESETREQ;
    CM1_SYSRESETREQ_d1<=CM1_SYSRESETREQ_d0;
    CM1_SYSRESETREQ_d2<=CM1_SYSRESETREQ_d1;
  end
assign CM1_SYSRESETn = rst_x_d1 & rst_x_d2
                     & ~CM1_SYSRESETREQ_d1 & ~CM1_SYSRESETREQ_d2;
assign CM1_DBGRESETn = rst_x_d1 & rst_x_d2 ;
assign CM1_EDBGRQ = 1'b0;
assign CM1_DBGRESTART = 1'b0;
////////////////////////////////////
assign TDO       = (CM1_nTDOEN==1'b0) ? CM1_TDO : 1'bz;
assign nTDOEN    = CM1_nTDOEN;
assign CM1_TDI   = TDI;
assign CM1_nTRST = nTRST;
assign TMS       = (CM1_SWDOEN==1'b1) ? CM1_SWDO : 1'bz;
assign CM1_SWCLKTCK = TCK;
assign CM1_SWDITMS  = TMS ;// | CM1_JTAGNSW;

reg tck1,tck2,tck3;
// debug clock synchroniser
always @ (posedge HCLK or negedge nTRST)
  if (~nTRST) begin
      tck1    <= 1'b0;
      tck2    <= 1'b0;
      tck3    <= 1'b0;
  end else begin
      tck1    <= TCK;
      tck2    <= tck1;
      tck3    <= tck2;
  end
assign RTCK = tck3;
endmodule
```

4. 通用1位寄存器的例子

通用1位移位寄存器的代码实现如下。

```
module Shifter#(parameter N=8)(
input clk,
input reset,
input left_in,
input right_in,
input [1:0] mod,
input [N-1:0] shift_in,
output reg[N-1:0] shift_out
);
always @(posedge clk or posedge reset)
    if(reset)
        shift_out <= 'b0; //清除移位寄存器内部状态
    else case(mod)
```

```
        2'b00:shift_out<={shift_out[N-2:0], right_in};//左移
        2'b01:shift_out<={left_in, shift_out[N-1:1]}; //右移
        2'b10:shift_out<= shift_in;             // 并行输入
    endcase
endmodule
```

5. 简单的中断控制器

```
module simple_vic#(parameter INT_NUM=6,parameter INT_WIDTH=8)(
  input clk,
  input rst_x,
  input [INT_NUM-1:0] int_source,
  input [INT_NUM-1:0] int_mask,
  input [INT_NUM-1:0] int_clr,
  output vic_int,
  output reg [INT_NUM-1:0] int_reg,
  output reg [INT_NUM-1:0] int_mask_reg
);
reg [INT_NUM-1:0] int_source_d0;
reg [INT_NUM-1:0] int_source_d1;
wire[INT_NUM-1:0] int_source_pulse;

always @(posedge clk or negedge rst_x)
  if(!rst_x)begin
    int_source_d0<=0;
    int_source_d1<=0;
  end else begin
    int_source_d0<=int_source;
    int_source_d1<=int_source_d0;
  end

assign int_source_pulse=(int_source_d0&~int_source_d1);
wire vic_int_pulse  =|(int_source_pulse[INT_NUM-1:0] & int_mask[INT_NUM-1:0]);

always @(posedge clk or negedge rst_x)
  if(!rst_x)
    int_reg<='d0;
  else if(|int_source_pulse)
    int_reg<=int_reg|int_source_pulse;
  else if(|int_clr)
    int_reg<=int_reg&(~int_clr);

always @(posedge clk or negedge rst_x)
  if(!rst_x)
    int_mask_reg<='d0;
  else if(vic_int_pulse)
    int_mask_reg<=int_mask_reg | {int_source_pulse&int_mask};
  else if(|int_clr)
    int_mask_reg<=int_mask_reg&(~int_clr);

reg [INT_WIDTH-1:0] clk_cnt;
reg int_state;
always @(posedge clk or negedge rst_x)
    if(!rst_x)begin
        clk_cnt<=0;
        int_state<=1'b0;
    end else if(int_state==0)   begin
```

```
        clk_cnt<=0;
        if(vic_int_pulse)
            int_state<=1'b1;
    end else begin
        if(~(|int_mask_reg))
            int_state<=1'b0;
        if(clk_cnt=={INT_WIDTH{1'b1}})
            int_state<=0;
        if(clk_cnt=={INT_WIDTH{1'b1}})
            clk_cnt<=0;
        else
            clk_cnt<=clk_cnt+1'b1;
    end
// int duration!
assign vic_int=(INT_WIDTH==0) ? vic_int_pulse:int_state;
endmodule
```

6. 4位转16位解码器

下面的这段代码在某国产CPU里面使用,读者可以思考一下为什么这样进行译码?

```
module decoder_4_16(
    input enable,
    input [3:0] in,
    output [15:0] out
);
//  wire enable;
//  wire [3:0] in;
//  wire [15:0] out;
    wire [3:0] high_d;
    wire [3:0] low_d;

    assign high_d[3]=( in[3])&( in[2])&enable;
    assign high_d[2]=( in[3])&(~in[2])&enable;
    assign high_d[1]=(~in[3])&( in[2])&enable;
    assign high_d[0]=(~in[3])&(~in[2])&enable;

    assign low_d[3]=( in[1])&( in[0]);
    assign low_d[2]=( in[1])&(~in[0]);
    assign low_d[1]=(~in[1])&( in[0]);
    assign low_d[0]=(~in[1])&(~in[0]);

    assign out[15]=high_d[3]&low_d[3];
    assign out[14]=high_d[3]&low_d[2];
    assign out[13]=high_d[3]&low_d[1];
    assign out[12]=high_d[3]&low_d[0];
    assign out[11]=high_d[2]&low_d[3];
    assign out[10]=high_d[2]&low_d[2];
    assign out[ 9]=high_d[2]&low_d[1];
    assign out[ 8]=high_d[2]&low_d[0];
    assign out[ 7]=high_d[1]&low_d[3];
    assign out[ 6]=high_d[1]&low_d[2];
    assign out[ 5]=high_d[1]&low_d[1];
    assign out[ 4]=high_d[1]&low_d[0];
    assign out[ 3]=high_d[0]&low_d[3];
    assign out[ 2]=high_d[0]&low_d[2];
```

```
    assign out[ 1]=high_d[0]&low_d[1];
    assign out[ 0]=high_d[0]&low_d[0];
endmodule
```

> 利用权重的关系，对应相等就可以，很巧妙，如$in[3]\&in[2]$ 代表了12的基数

7. FSM测试例子

在编写FSM的测试程序时，经常需要检查FSM的state，常规的做法是：

- 将state定义为输出端口，并引出，这种做法通常会浪费有限的硬件资源，而且多余。
- 采用打印语句，例如$display（）函数，在命令行中显示出来。若这样做，结果不是显示在波形里面，不直观。

一个比较好的做法是在testbench中直接引用该状态变量，并通过case 语句将state转换为自然语言表述。具体做法为：

1. 在testbench里定义一个reg型变量，用于存储ASCII字符串。例如，以下定义了一个最大长度为4的字母。

   ```
   reg [4*8-1: 0] current_state, next_state;
   ```

2. 利用模块变量的直接引用，并通过always语句将状态机的各个状态与自然语言一一对应起来：

   ```
   always@(tb.uut.state or tb.uut.next_state)begin
     case(tb.uut.state)
       Zero:current_state = "Zero";
       one:current_state = "one";
       two:current_state = "two";
       three:current_state = "three";
       four:current_state = "four";
       default:current_state = "Error";
     endcase
     case(tb.uut.next_state)
       Zero:next_state = "Zero";
       one:next_state = "one";
       two:next_state = "two";
       three:next_state = "three";
       four:next_state = "four";
       default:next_state = "Error";
     endcase
   end
   ```

3. 在仿真工具的波形窗口里，把前面定义的reg型state indicator的radix改成ASCII，就可以在波形窗口里面看到文本表示的state了。

8. Generate 生成语句

假定读者需要设计一个100×100的阵列乘法器，一定会思考有没有用几条语句就搞定这种描述的简单方法？Verilog-2001针对这类情况添加了generate生成语句与循环语句，通过该语句能够简化代码编写，梳理设计思路。

通过generate循环语句，允许产生module和primitive的多个实例化，同时也可以产生多个variable、wire、task、function、assign、always、initial。在generate 语句中可以引入for、if-else和case 语句，根据条件不同产生不同的实例。为此，Verilog-2001增加了以下关键字：generate、endgenerate、genvar、localparam。语法为：

1. genvar 后面的for，其变量必须是genvar变量。
2. for里必须有begin，哪怕只有一句。
3. begin必须有名称，因为该语句块中的变量必须要设定一个名称。
4. generate例化的模块名称是 inst [j].unit
5. generate 语句可以看作是标准化为块的综合指令。

1. for循环

for循环以 begin开始,end结束, begin后边必须有一个唯一的标识符,例如下面的8位加法器例化过程

```
generate
  genvar i;
  for(i=0;1<=7;i=1+1)
    begin: for_name
      adder U_add(a[8*i+7:8*i],b[8*i+7:8*i], ci[i], sum_for[8*i+7:8*i],
c0_or[i+1]);
    end
endgenerate
```

在for循环里使用always语句的例子如下：

```
generate
  genvar i;
  for(i = 0; i<11; i=i+1)begin:iq_data_gen
    always@(*)begin
      iq[i*2] = i_in[i];
      iq[i*2+1] = q_in[i];
    end
endgenerate
```

2. If-else 例化

这个例子基于数据宽度例化不同的乘法器：

```
generate
  if(IF_WIDTH < 10)
    begin:if_name
      multiplier_impl #(IF_WIDTH) U1(a,b,sum_if);
    end
  else
    begin:else_name
      multiplier_imp2 #(IF_WIDTH) u2(a,b,sum_if);
    end
endgenerate
```

3. Generate-case

这个例子与上面的例子基本一致，只是分支情况更多一些：

```
generate
  case(WIDTH)
    1:begin: case1_name
      adder #(WIDTH*8) x1(a, b, c, sum_case, c0_case);
    end
    2:begin: case2_name
      adder #(WIDTH*4) x2(a, b, c, sum_case, c0_case);
    end
    default:begin:d_case_name
      adder x3(a, b, c, sum_case, c0_case);
    end
  endcase
endgenerate
```

1.2.12 Verilog 不同版本差异

图1-61的内容是Verilog各个发展阶段的关键字列表。通过这个列表，应当体会到为什么要添加这些额外的关键字，以及背后隐藏的出发点。例如Verilog-2001中的signed types就是针对Verilog-1995不能自然表达有符号数的运算而添加的；而Verilog-2005中的$clog2系统函数就是为了方便计算数据位宽，避免位浪费，同时又保持IP的灵活性而设计的；而SystemVerilog中的always_comb/always_ff就是针对always不能清楚区分寄存器和组合电路而设计的。

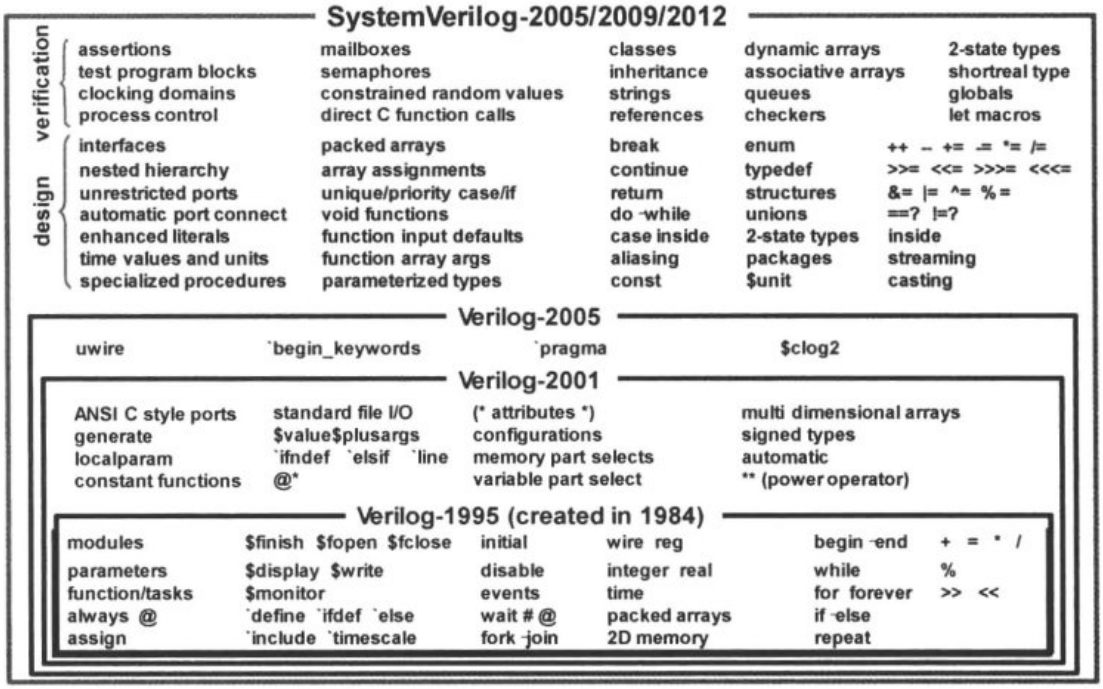


图 1-61　Verilog 语法发展脉络图

13. Verilog语法小结

1. 数据类型语法小结

1. wire型数据通常用assign关键字进行赋值。wire只能被assign连续赋值，reg只能在initial 和always中赋值。Input端口只能定义成wire型。如果端口没有声明，则默认是wire线网型，且输入端口只能是wire线网型。
2. reg是寄存器数据类型的关键字。寄存器是数据存储单元的抽象，通过赋值语句可以改变寄存器存储的值，相当于改变触发器存储的值。reg型常用来表示always模块内的指定信号，代表触发器。在always块内被赋值的每一个信号都必须定义为reg型，即赋值操作符的左端变量必须是reg型。reg型只能在initial和always中赋值。

3. 所谓的always必须用reg的意思是always里面的赋值语句中的被赋值变量为reg型，而不是说在always里面出现的变量都为reg型。
4. reg型与wire型的区别在于:reg型保持最后一次的赋值,而wire型需要持续的驱动。
5. memory型通过对reg型变量建立数组以对存储器建模。reg[n-1:0]的存储器名[m-1:0] reg[n-1:0]定义了存储器中每一个存储单元的大小,即该存储器单元是一个n位宽的寄存器,[m-1:0]代表了存储器的大小,即该存储器中有多少个这样的寄存器。
6. 一个n位寄存器可以在一条赋值语句中直接赋值,而一个完整的存储器则不行,如果要对 memory型存储单元进行读写,则必须要指定地址。
7. 用 parameter定义常量。

2. 数制表示语法小结

1. 下划线"_"可以随意用在整数和实数中，没有实际意义,只是为了提高可读性。如56等效于5_6。
2. 整数负数使用补码形式表示,而不能表示为诸如:16'h-9之类的形式。
3. 在进行基本算术运算时,如果某一操作数有不确定的值X,则运算结果也是不确定X。

3. 赋值过程语法小结

1. 连续赋值语句只能用来对线网型变量进行赋值,而不能对寄存器变量进行赋值。一个线网型变量一旦被连续赋值语句赋值之后,赋值语句右端的赋值表达式的值将持续对被赋值变量产生连续驱动。只要右端表达式中任一操作数的值发生变化,就会立即触发对被赋值变量的更新操作。其基本的语法格式如下:线网型变量类型[线网型变量位宽]线网型变量名；assign#(延时量)线网型变量名=赋值表达式。
2. 过程赋值主要用于两种结构化模块（initial模块和always模块）中的赋值语句。在过程块中只能使用过程赋值语句(不能在过程块中出现连续赋值语句），同时过程赋值语句也只能用在过程赋值模块中。
3. 过程赋值语句只能对寄存器类型的变量(reg、integer、real和time)进行操作，经过赋值后，上面这些变量的取值将保持不变，直到另一条赋值语句对变量重新赋值为止。
4. memory型只能对指定地址单元的整个字进行赋值，不能对其中的某些位单独赋值。
5. 在关系运算符中，若某个操作数的值不定，则关系是模糊的，返回的是不定值X。
6. 实例算子"==="和"!=="可以用于比较含有X、Z的操作数，在模块的功能仿真中有着广泛应用。
7. 对组合逻辑建模采用阻塞赋值；对时序逻辑建模采用非阻塞式赋值；用多个always块分别对组合和时序逻辑建模；尽量不要在同一个always块内混合使用阻塞赋值和非阻塞赋值，如果在同一个always块里面既为组合逻辑，又为时序逻辑建模，应使用非阻塞赋值；不要在多个always块中为同一个变量赋值。

4. 结构描述小结

1. 一个程序可以有多个initial模块、always模块、task模块和function模块。initial模块和always 模块都是同时并行执行的，区别在于initial模块只执行一次，而always模块则是不断重复运行，task模块和function模块能被多次调用。initial模块是面向仿真的，是不可综合的。
2. begin...end块定义语句中的语句是串行执行的，而fork...join块语句中的语句是并行执行的。
3. 利用always实现组合逻辑时，要将所有的信号放进敏感列表中，而实现时序逻辑时却不一定要将所有的结果放进敏感信号列表中。
4. 延时控制只能在仿真中使用，是不可综合的，在综合时，所有的延时控制都会被忽略。
5. if语句中的else不能省，case语句的default分支虽然可以默认，但是一般不要默认。
6. case语句的分支是并行执行的，各个分支没有优先级，而if语句的选择分支是串行执行的。
7. forever循环语句用于连续执行过程语句，必须写在initial模块中。
8. repeat循环语句执行指定循环数，如果循环技术表达式的值不确定，即为X或Z时，那么循环次数按0处理。

5. verilog 实现相关小结

1. 组合逻辑电路在逻辑功能上的特点是：任意时刻的输出仅仅取决于该时刻的输入,与电路原来的状态无关。而时序逻辑电路在逻辑功能上的特点是:任意时刻的输出不仅取决于当时的输入信号,而且

还取决于电路原来的状态,或者说,还与以前的输入有关。

2. 状态机一般包括组合逻辑和寄存器逻辑两部分。组合电路用于状态译码和产生输出信号,寄存器用于存储状态。
3. 所谓高阻是输出端属于浮空状态,只有很小的漏电流流动,其电平随外部电平高低而定,门电平放弃对输出电路的控制或者可以理解为输出与电路是断开的。双向引脚的输出就 是在不使能状态下,输出为高阻,而在使能状态下输出为正常。

1.3 复杂逻辑模的设计

前面章节详细介绍了Verilog的语法，并通过8个简单例子给出了常见的逻辑电路设计套路。本节将讲述如何利用简单的Verilog描述实现复杂的逻辑电路。

对于复杂的逻辑电路，可能有很多的答案，但有一点是一定的：电路描述的信息熵很高。逻辑电路的复杂性通常表现在以下几个方面：①内部结构组成复杂；②控制状态复杂且高度耦合；③接口复杂且信号数量很多；④模块数量巨大。

站在工程学的角度，内部结构组成的复杂可以通过树状结构或网状结构进行消解，即通常所说的结构化设计（Hierarchy)和总线设计；控制状态的复杂则可以通过控制流分解，按照主从状态细化的方法降低单个组成单元的复杂度；而任何复杂的接口都可以简化为流量控制处理，并通过成熟的通信协议设计降低难度；数量巨大的模块可以通过分类整理，实现条理化、清晰化。因此，任何复杂的逻辑设计最终可以演变为：①对基本单元的归类整理和层次化设计，即结构化设计；②基本功能的单元设计，即数据流设计；③将层次化的基本单元进行梳理，最终形成全局电路控制，即控制流设计。