数字设计实践与 Verilog硬件描述语言

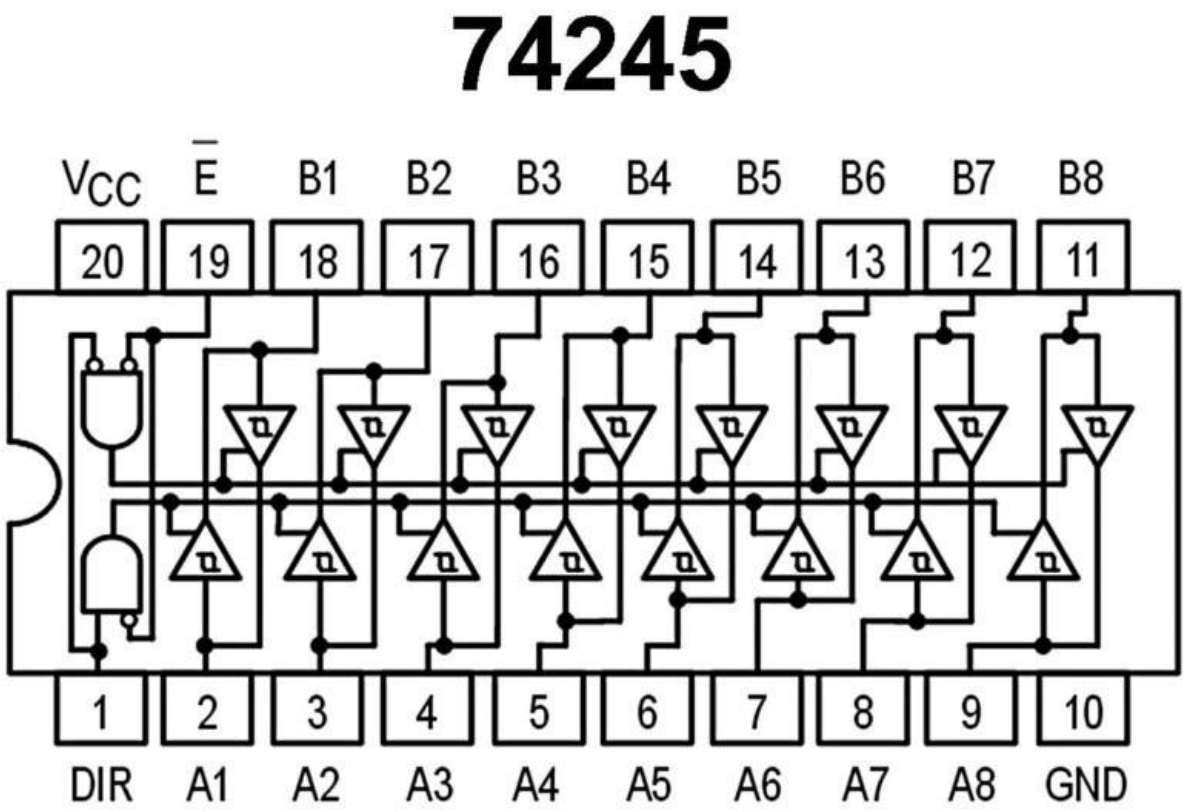
高翠芸
School of Computer Science
gaocuiyun@hit.edu.cn

Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- **赋值语句和块语句**
- 条件语句和循环语句
- 模块的调用
- 模块的测试

用Verilog实现三态输出

- Verilog为高阻态内设了位数据值”z”，所以它很容易指定三态输出




```
module Vr74x245(G_L, DIR, A, B);
  input G_L, DIR;
  inout [1:8] A, B;              //输出端口可作输入

  assign A = (~G_L & ~DIR) ? B : 8'bz;
  assign B = (~G_L &  DIR) ? A : 8'bz;
endmodule
```

与74*245类似的8位三态驱动器的Verilog模块

模块基本结构

模块声明：

module module_name（port_list）；//模块名（端口声明列表）

端口定义：

input[信号位宽];　　//输入声明

output [信号位宽] ;　//输出声明

…

数据类型说明：

reg [信号位宽] ;　　//寄存器类型声明

wire [信号位宽] ;　　//线网类型声明

parameter;　　//参数声明

…

功能描述：　　//主程序代码

assign *a=b+c*

always@(posedge clk or negedge reset)

function

task

…

endmodule

Verilog HDL语法

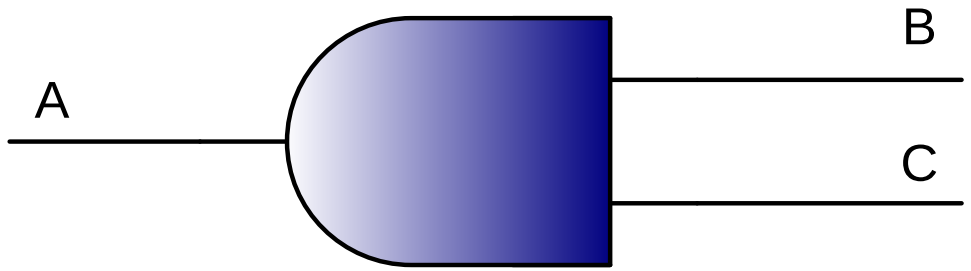
- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- **赋值语句**和块语句
  - **连续赋值语句**
  - **过程赋值语句**
- 条件语句和循环语句
- 模块的调用
- 模块的测试

赋值语句

- 在Verilog中，变量是不能随意赋值的，需要使用**连续赋值**语句和**过程赋值**语句。
- assign称为**连续赋值**，对应于线网类型变量wire；
- initial或always称为**过程赋值**，对应于寄存器类型变量reg。

连续赋值语句

- 连续赋值语句用于把值赋给**线网型变量**(**不能为寄存器型变量赋值**)
- 语句形式为： assign　A = B & C;
  - 只要在右端表达式的操作数上有事件(事件为值的变化)发生时，表达式即被计算；
  - 如果计算的结果值有变化，新结果就赋给左边的线网。



连续赋值语句

例：

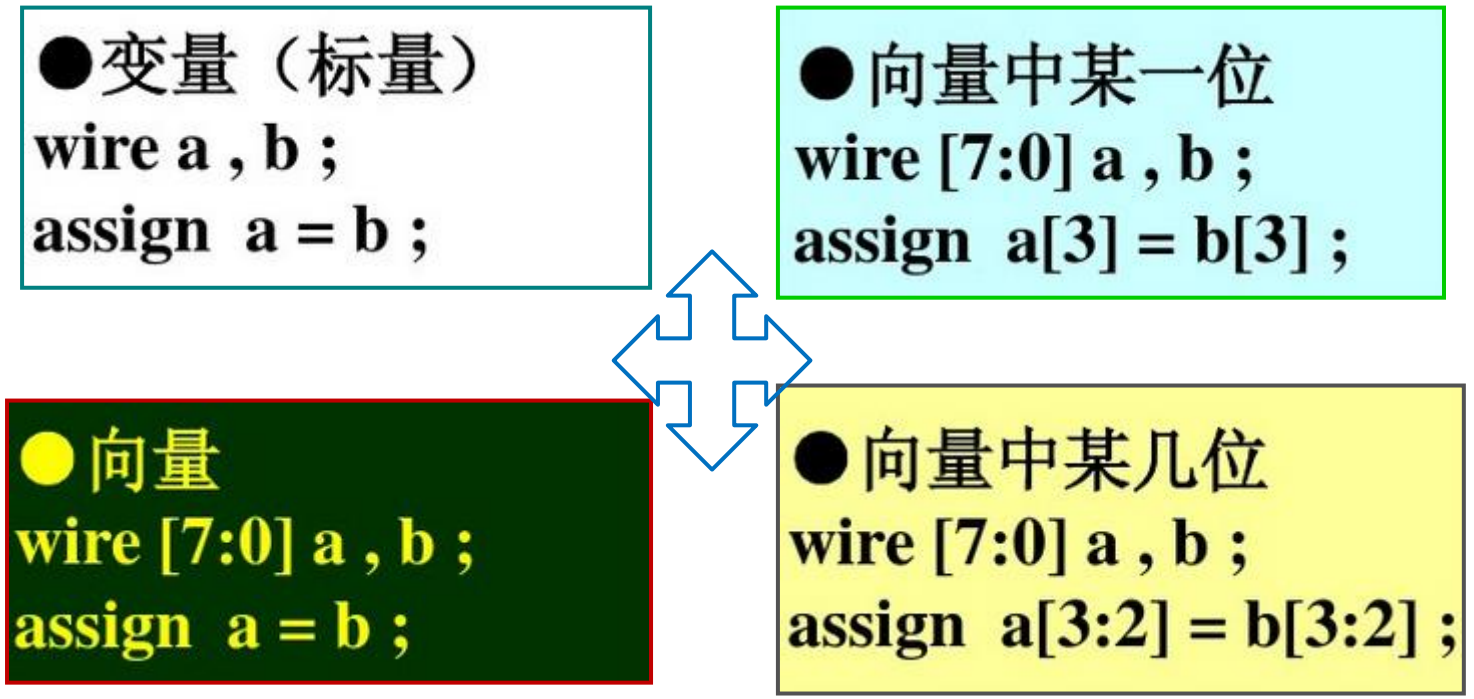
```
wire  a;
assign a=1'b1;
```

- 语法格式：　assign 线网型变量名=赋值表达式;
- 等号右端赋值表达式的值会持续对被赋值变量产生连续驱动，而且只要等号右端赋值表达式的值改变，左端被赋值变量的值就会**立即改变**;
- 对应到电路中去，**就是导线**。

连续赋值语句assign

- 左侧数据类型必须是线网型数据（wire）；
- 所有右值都是敏感信号，右侧任何信号的变化都会激活该语句，使其被立即执行一次；
- 每条assign赋值语句相当于一个逻辑单元，等价于门级描述；
- 各个assign赋值语句之间是并发的关系；
- 在过程块（initial/always）外面；
- 描述**组合电路**。

连续赋值语句


●变量（标量）
wire a , b ;
assign a = b ;
●向量中某一位
wire [7:0] a , b ;
assign a[3] = b[3] ;
●向量
wire [7:0] a , b ;
assign a = b ;
●向量中某几位
wire [7:0] a , b ;
assign a[3:2] = b[3:2] ;

连续赋值语句举例

```
module   FA _Df (A, B, Cin, Sum, Cout ) ;
input   A, B, Cin;
output   Sum, Cout ;

assign   Sum = A ^B ^Cin;
assign   Cout = (A & Cin) | (B & Cin) | (A & B) ;
endmodule
```

在本例中，有两个连续赋值语句。这些赋值语句是并发的，与其书写的顺序无关

过程赋值语句

- 首先，举两个例子：

```
reg c;
initial begin
    c=1'b0;
end
```

```
reg a,c;
always@(a) begin
    c<=c+a;
end
```

- 在initial/always块中使用**过程赋值**语句；
- initial只会执行一次，即只执行一次把c赋零的行为；
- 而always会不断执行，即每一次a的值改变时，c都会被重新赋值。

过程赋值语句

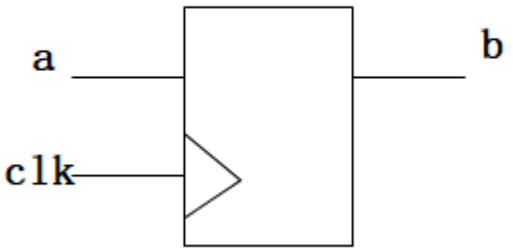
- 只能出现在**过程块**中（initial/always），主要描述**时序电路**；
- 过程赋值语句中： 没有关键词“assign”；
- 左侧数据类型必须是**reg**类型的变量；
- 每条过程赋值语句之间是**顺序执行**的关系。
- **阻塞赋值**（运算符=）和**非阻塞赋值**（运算符<=）

阻塞赋值与非阻塞赋值

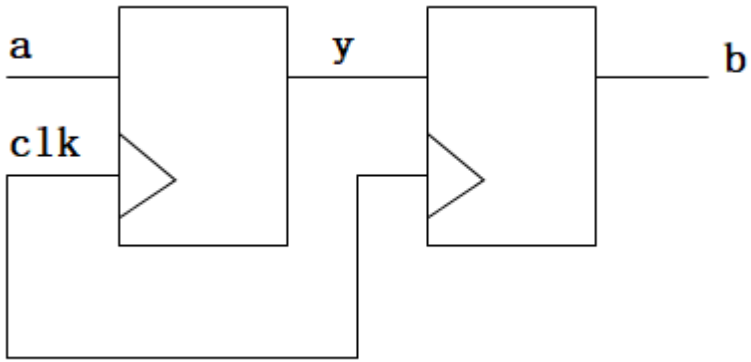
- 阻塞（Blocking）赋值方式（如b = a ; ）
  - 赋值语句执行完后，块才结束；
  - b的值在赋值语句执行完后**立刻就改变**；
  - 可能产生意想不到的结果。
- 非阻塞（Non_Blocking）赋值方式（如b <= a;）
  - 块结束后才完成赋值操作；
  - b的值**并不是立刻就改变**；
  - 这是一种**比较常用的赋值方法**。（特别在**编写可综合模块**）

阻塞赋值与非阻塞赋值

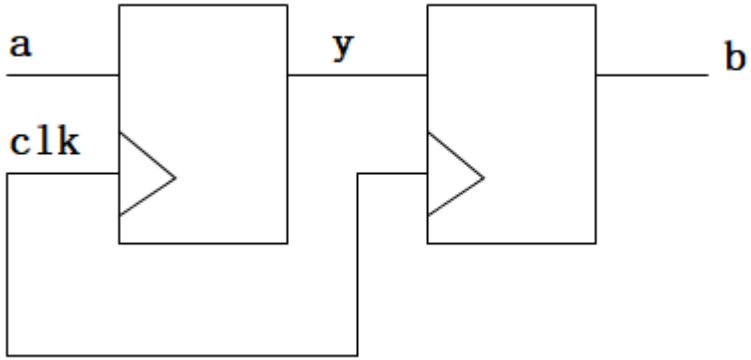
```
module bloc(clk,a,b);
input clk, a;
output b; reg b;
reg y;
always @(posedge clk)
begin
   y=a;
   b=y;
end
endmodule
```

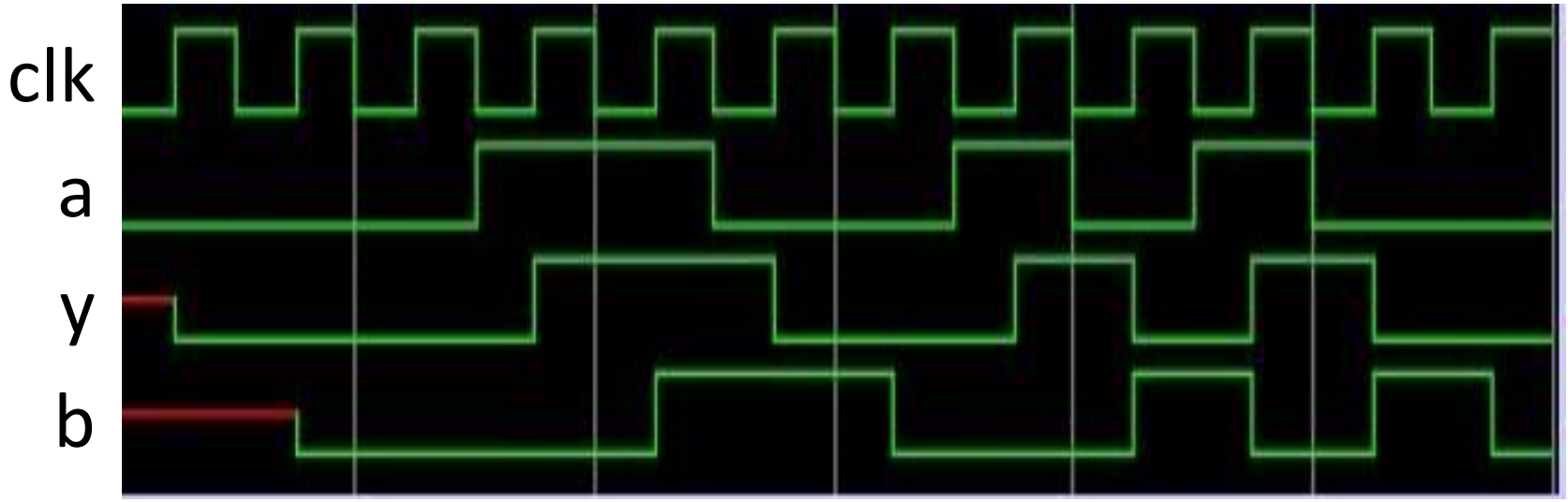



```
module nonbloc(clk,a,b);
input clk, a;
output b; reg b;
reg y;
always @(posedge clk)
begin
   y<=a;
   b<=y;
end
endmodule
```



非阻塞赋值示例





非阻塞赋值

块内的赋值语句同时进行：先同时采样，最后一起更新

非阻塞赋值

```
always @ ( posedge clk )
 begin
      c<= b;
      b<= a;
      a<= d;
 End
```

结果：
a=2，b=5
c=3，d=2

结果与书写的顺序无关（原因：同步更新）

时序电路特点：输出不会随输入变化而立即变化

初值：
a=5，b=3
c=10, d=2

非阻塞赋值

```
always @ ( posedge clk )
 begin
      a<= d;
      b<= a;
      c<= b;
 End
```

结果：
a=2，b=5
c=3，d=2

本质上，在一个时钟沿触发里，a得到d 的值，但b得到的永远是a的旧值，c得到的永远是b的旧值（原因：同步更新）。

如何区分阻塞赋值与非阻塞赋值？

- 时序逻辑
  - 一定用非阻塞赋值“<=”,只要看到敏感列表有posedge就用“<=”。
- 组合逻辑
  - 一定用“=”，只要敏感列表没有posedge就用“=”。
- 时序逻辑和组合逻辑分成不同的模块
  - 即一个always模块里面只能出现非阻塞赋值“<=”或者“=”。

连续赋值与过程赋值的比较

	过程赋值	连续赋值
assign	无assign （过程性连续赋值除外）	有assign
符号	使用“=”，“<=”	只使用“=”
位置	在always语句或initial语句中均可出现	不可出现于always语句和initial语句
执行条件	与周围其他语句有关	等号右端操作数的值发生变化时
用途	驱动寄存器	驱动线网

Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和**块语句**
  - **initial块语句**
  - **always语句**
- 条件语句和循环语句
- 模块的调用
- 模块的测试

过程块

- 过程块是**行为模型**的基础。
- 过程块有两种：
  - initial块
    - 只能执行一次。
    - 不带触发条件。
    - 它通常用于**仿真**模块。
  - always块
    - **循环**执行。
    - 通常**带触发条件**，满足触发条件则执行。
    - 一个模块中有多个always块时，可以**并行**进行。

```
reg c;
initial begin
    c=1'b0;
end
```

```
reg a,c;
always@(a) begin
    c<=c+a;
end
```

initial块

- 格式如下：

```
Initial
    begin
            语句1;
            语句2;
            ……
            语句n;
    end
```

```
initial
    begin
        #20 begin a = 0;b = 0; cin= 1;end
        #20 begin a = 0;b = 1; cin= 0;end
        #20 begin a = 0;b = 1; cin= 1;end
        #20 begin a = 1;b = 0; cin= 0;end
        #20 begin a = 1;b = 0; cin= 1;end
        #20 begin a = 1;b = 1; cin= 0;end
    end
```

- begin_end为顺序块，用来标识顺序执行的语句。

always块

- 格式如下：

```
always @（<敏感信号列表>）
        begin
                //过程赋值
                //if-else、case选择语句
                        //for、while等循环块
        end
```

- always语句通常**带触发条件**，触发条件被写在敏感信号列表中，只有当触发条件满足条件或发生变化时，其后的”begin-end”块语句才能被执行。
- 敏感信号列表中可以有多个信号，用关键字or连接；
- 敏感信号可分为两种：电平敏感、边沿敏感；
- 用关键字*posedge*和*negedge*限定信号敏感边沿。

always块

- 电平触发的always块通常用于描述组合逻辑和带锁存器的组合逻辑；
- 边沿触发的always块通常用于描述时序逻辑。

```
module reg_ adder (out, a, b, clk);
    input clk;
    input [2: 0] a, b;
    output [3: 0] out;
    reg [3: 0] out;
    reg [3: 0] sum;
  always @( a or b) // 若a或b发生任何变化，执行
        sum = a + b;
  always @( negedge clk) // 在clk下降沿执行
        out = sum;
endmodule
```

Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- **条件语句和循环语句**
- 模块的调用
- 模块的测试

条件语句和循环语句

- 都必须在过程块中使用
- **条件语句**：
  - if、case
- **循环语句**：
  - forever、repeat、while、for

条件语句-if语句

- 即使用不到**else分支**，语句中else分支也最好加上，否则电路有可能生成不稳定的电路，**造成**结果的错误。
- 编译器认为条件不满足时，会引进触发器保持原值。
- 时序电路可利用上述特性来保持状态。

```
always @ ( some_event )
    begin
        if (a>b)
out1=int1;
        else
            if(a==b)
out1=int2;
            else
```

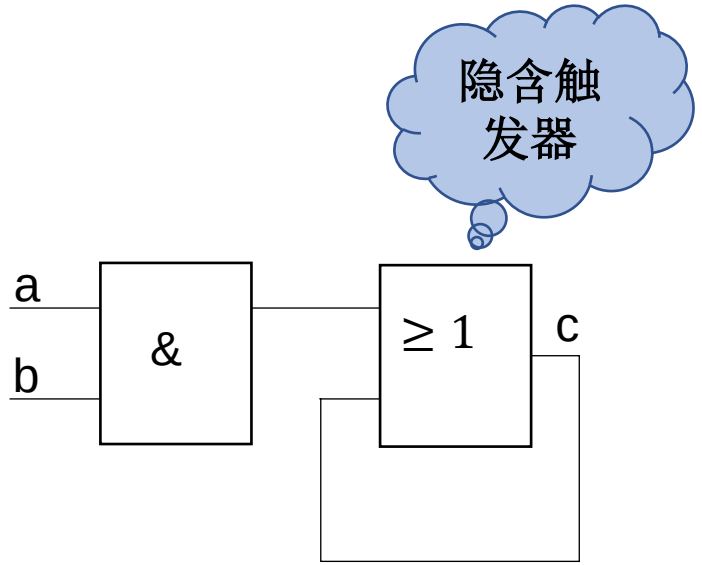
- **if(表达式)　语句**　　例：**if(a>b)　out1=int1;**
- **If(表达式)**
  语句1
  else
  语句2

例：**if(a>b)**
out1=int1;
else
out1=int2;

条件语句-if语句

```
module   buried_ff(c,b,a) ;
input   a,b;
output   c;
reg c;

always @(a or b)
    begin
      if ((b=1)&&(a==1))   c=1;
    end
endmodule
```



如何改正错误？

条件语句-case语句

- **case（条件表达式）**

  分支1：语句块1；

  分支2：语句块2；

  ……

  default：语句块n；

  endcase

例：reg [2:0] cnt;

```
case(cnt)
    3'b000:q=q+1;
    3'b001:q=q+2;
    default:q=q;
endcase
```

- case语句的所有表达式值的**位宽必须相等**
- 语句中**default一般不要缺省**。在”always”块内，如果给定条件下变量没有赋值，这个变量将保持原值（生成一个**锁存器**）
- 分支表达式中可以存在**不定值x**和**高阻值z**
  - 如2**‘b0x，或2’b0z**

条件语句if与case的区别

- if　生成的电路是**串行**的，是有优先级的编码逻辑；
- case生成的电路是**并行**的，各种判定情况的优先级相同。

- if　生成的电路**延时较大**，占用硬件资源少；
- case生成的电路**延时较短**，占用硬件资源多。

循环语句

- Verilog的循环语句是依靠**电路的重复生成**实现的。
- 4种循环语句：
  - for 循环：执行给定的循环次数；
  - while 循环：执行语句直到某个条件不满足；
  - repeat 循环：连续执行语句N次；
  - forever 循环：连续执行某条语句，多余于initial块中，以生成周期性波形。
- for、while是**可综合**的，但循环的次数需要在编译之前就确定，动态改变循环次数的语句则是不可综合的
- repeat语句在有些工具中可综合，有些不可综合
- forever语句是不可综合的，常用于产生各类仿真激励

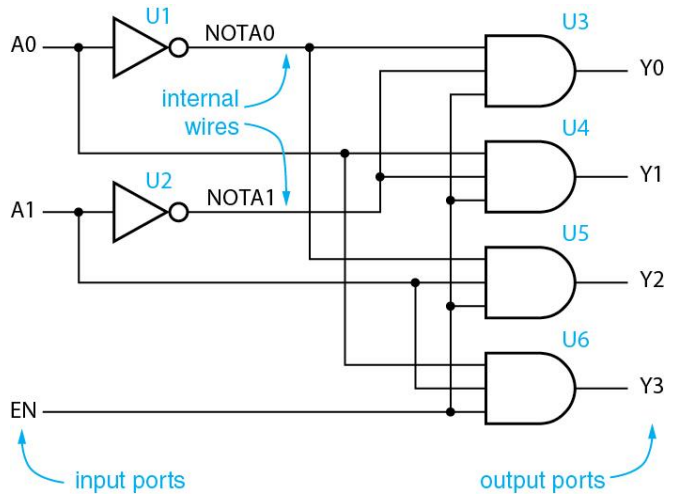
用Verilog实现2-4译码器——1

- Verilog设计译码器方法有很多
- 最原始的方法是编写等效于译码器逻辑电路的**结构性程序**。

```
module Vr2to4dec_s(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output Y0, Y1, Y2, Y3;
  wire NOTA0, NOTA1;

  not U1 (NOTA0, A0);
  not U2 (NOTA1, A1);
  and U3 (Y0, NOTA0, NOTA1, EN);
  and U4 (Y1,   A0, NOTA1, EN);
  and U5 (Y2, NOTA0,   A1, EN);
  and U6 (Y3,   A0,   A1, EN);
endmodule
```

缺点：不易于理解和维护



Inputs			Outputs			
EN	A1	A0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

用Verilog实现2-4译码器——2

- 使用Verilog的行为化描述。
- 采用了一个always语句，其敏感信号列表包括译码器所有输入。
- 将输出变量声明为reg类型，故而可以在过程块中对输出变量赋值。
- 用一个if语句来测试使能输入。
  - 如果EN为0，所有输出都设置为0；
  - 当EN有效时，译码器的功能。

```
module Vr2to4dec_b1(A0, A1, EN, Y0,
Y1, Y2, Y3);
  input A0, A1, EN;
  output reg Y0, Y1, Y2, Y3;

  always @ (A0, A1, EN)
    if (EN==0) {Y3,Y2,Y1,Y0} = 4'b0000;
    else
      case ({A1,A0})
        2'b00: {Y3,Y2,Y1,Y0} = 4'b0001;
        2'b01: {Y3,Y2,Y1,Y0} = 4'b0010;
        2'b10: {Y3,Y2,Y1,Y0} = 4'b0100;
        2'b11: {Y3,Y2,Y1,Y0} = 4'b1000;
        default: {Y3,Y2,Y1,Y0} = 4'b0000;
      endcase
endmodule
```

用Verilog实现2-4译码器——3

- 使用2-4二进制译码器的行为化风格Verilog模块
  - 能够能更好地捕捉到译码器的行为特性。

```
module Vr2to4dec_b2(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output reg Y0, Y1, Y2, Y3;
  reg [3:0] IY;
  integer i;
  always @ (A0 or A1 or EN) begin
    IY = 4'b0000;        // Default outputs all 0
    if (EN==1)           // If enabled...
      for (i=0; i<=3; i=i+1)   // set out bit i where i equals {A1,A0}
        if (i == {A1,A0}) IY[i] = 1;
      {Y3,Y2,Y1,Y0} = IY;  // Copy internal bit-vector variable to outputs
  end
endmodule
```

2输入8位多路复用器的数据流型Verilog模块

- 多路复用器在**数据流形式**中，可以使用一系列条件操作符(?:)来提供所要求的功能。

```
module Vrmux2in8b_d(EN_L, S, D0, D1, Y);
  input EN_L, S;
  input [1:8] D0, D1;
  output [1:8] Y;

  assign Y = (~EN_L == 1'b0) ? 8'b0 : (
             (S == 1'd0) ? D0 : (
               (S == 1'd1) ? D1 : 8'bx));
endmodule
```

采用多重if语句实现多路复用器

- 多路复用器的行为化描述。
- 采用一系列多重if语句，一个if语句对应一个选择输入值

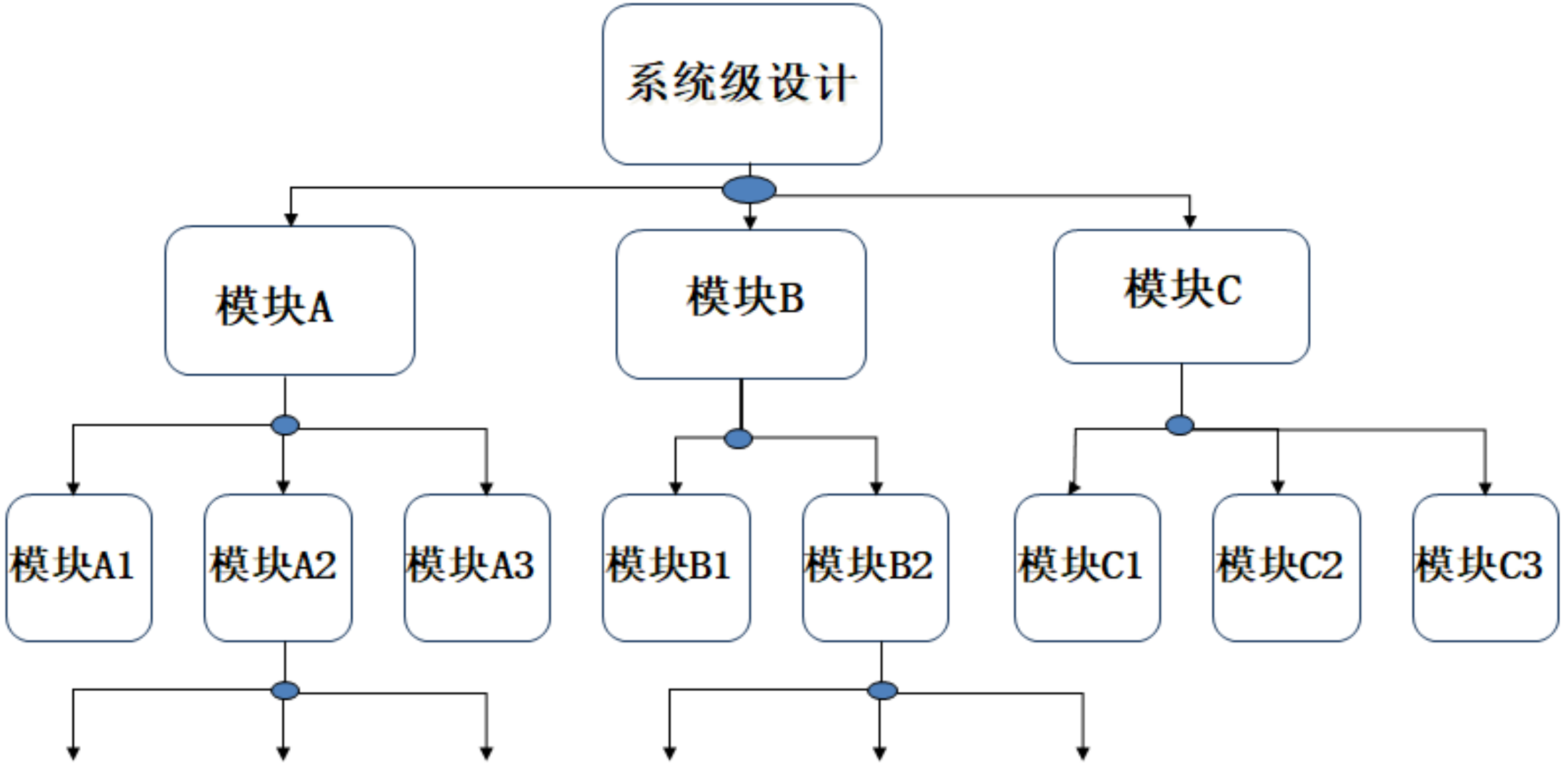
```
module Vrmux2in8b_b(EN_L, S, D0, D1, Y);
  input EN_L, S;
  input [1:8] D0, D1;
  output reg [1:8] Y;

  always @ (*) begin
    if (~EN_L == 1'b0) Y = 8'b0;
    else if (S == 1'b0) Y = D0;
      else if (S == 1'b1) Y = D1;
        else Y = 8'bx;
  end
endmodule
```

Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- **模块的调用**
- 模块的测试

模块的调用



模块基本结构

模块声明：

module module_name（port_list）；//模块名（端口声明列表）

端口定义：

input[信号位宽];　　//输入声明
output [信号位宽] ;　//输出声明
…

数据类型说明：

reg [信号位宽] ;　　//寄存器类型声明
wire [信号位宽] ;　　//线网类型声明
parameter;　　//参数声明
…

功能描述：　　//主程序代码

assign *a=b+c*
always@(posedge clk or negedge reset)
function
task
…
endmodule

模块实例化方法

- **调用模块实例**的一般形式为：
  - <模块名><参数列表><实例名>（<端口列表>）；
  - module_name　instance_name(port_associations) ;
- 其中**参数列表**是传递到子模块的参数值。
- 信号端口可以通过位置或名称关联，但是**关联方式不能够混合使用**。
- **定义模块**：module Design(端口1，端口2，端口3……);

1. 引用时，严格按照**模块定义的端口顺序**来连接，不用标明原模块定义时规定的端口名。
   - Design u_1(u_1的端口1，u_1的端口2，……)；//和Design对应
2. 引用时，用”.”符号，标明原模块定义时规定的端口名。
   - Design u_2(　.(端口1(u_1的端口1), .(端口2(u_1的端口2),……　);

模块实例化

- 例：

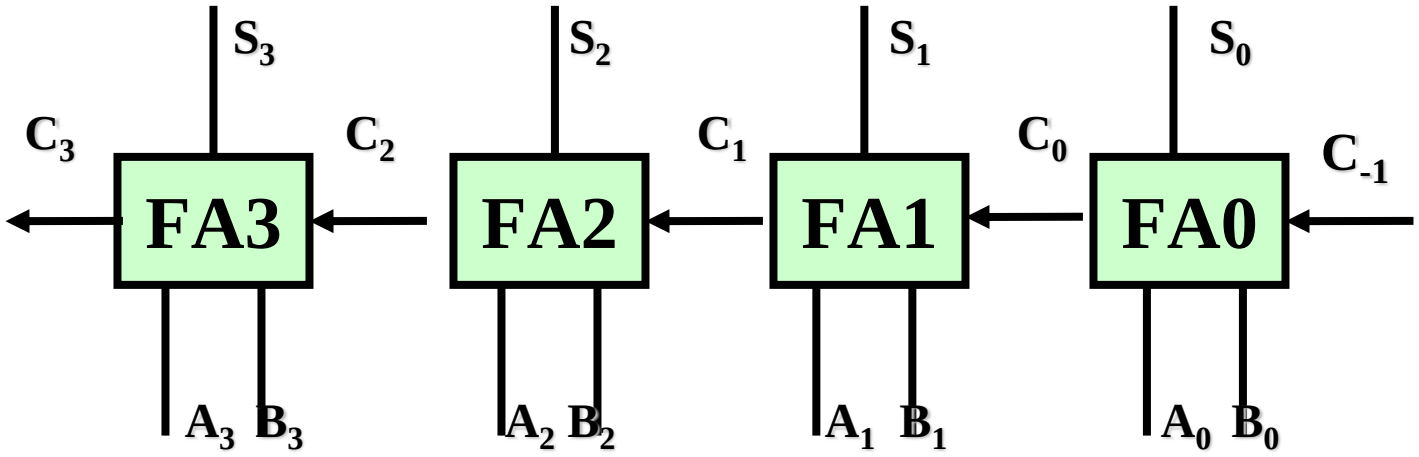
```
module    and1 (C,A,B);
input A,B;
output C;
...
```

- and1 A1 (T3,A,B); //实例化时采用**位置关联**
- and1 A2 (.C(T3), .A(A),.B(B))； //实例化时采用**名字关联**

- port_expr可以是以下的任何类型：
  - 标识符（reg 或net ）如 .C（T3），T3 为wire 型标识符。
  - 位选择，如 .C（D[0]），C 端口接到D 信号的第0bit 位。
  - 部分选择，如 .Bus（Din[5：4]）。
  - 上述类型的合并，如 .Addr（{ A1，A2[1：0]}。
  - 表达式（只适用于输入端口），如 .A（wire Zire = 0 ）。
- 例化名不能为**元模块名**或**关键字**。

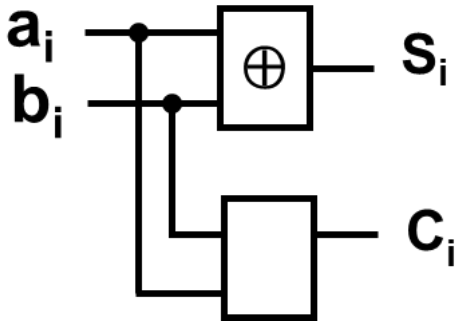
4位全加器设计实例

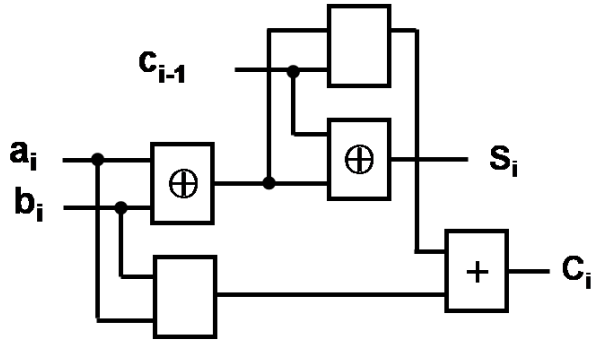
- 半加器模块：半加器由两个一位输入相加，输出一个结果位和进位。
- 1位全加器模块：由两个半加器和一个或门实现。
- 4位全加器模块：由4个1位全加器串联形成。



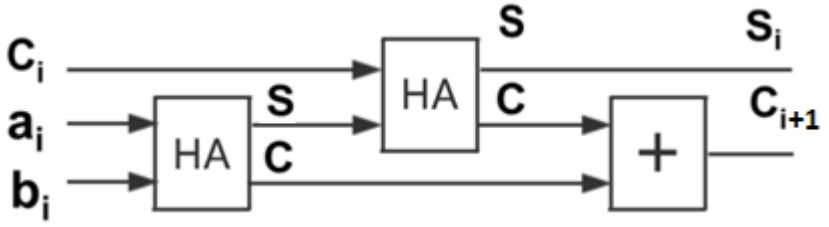
4位全加器设计实例

```
module half_adder(input  a,input  b, output  sum,output  c_out); //半加器
    assign sum = a^b;
    assign c_out = a&b;
endmodule
```






```
module full_adder(input  a,input  b,input  c_in,output  sum,output  c_out);
    wire  sum1;                                          //1 位全加器
    wire  c_out1,c_out2;
    half_adder  half_adder1(.a(a),.b(b),.sum(sum1),.c_out(c_out1));
    half_adder  half_adder2(.a(c_in),.b(sum1),.sum(sum),.c_out(c_out2));
    assign c_out =  c_out1|c_out2;
endmodule
```



4位全加器设计实例

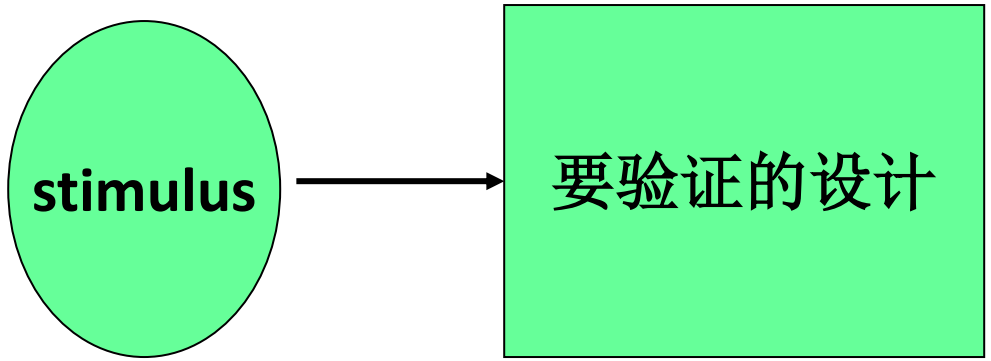
```
module add_4 ( input [3:0] a, input [3:0]b, input c_in,
                        output [3:0] sum, output c_out ); //4位全加器
    wire [3:0] c_tmp;
     full_adder i0 ( a[0], b[0], c_in, sum[0], c_tmp[0]);
     full_adder i1 ( a[1], b[1], c_tmp[0], sum[1], c_tmp[1] );
     full_adder i2 ( a[2], b[2], c_tmp[1], sum[2], c_tmp[2] );
     full_adder i3 ( a[3], b[3], c_tmp[2], sum[3], c_tmp[3] );
     assign c_out = c_tmp[3];
endmodule
```

Verilog HDL语法

- 模块的结构
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和块语句
- 条件语句和循环语句
- 模块的调用
- **模块的测试**

测试平台（test bench）

- 测试平台是一个**无输入**，**有输出**的顶层调用模块。
- 一个简单的测试平台包括：
  - 产生激励信号；
  - 实例化待测模块，并将激励信号加入到待测模块中。



简单的TestBench

TestBench模块

激励模块通常是**顶层模块**

- 激励信号数据类型要求为**reg**，以便保持**激励值**不变，直至执行到下一跳激励语句为止
- 输出信号的数据类型要求为wire，以便能随时跟踪激励信号的变化

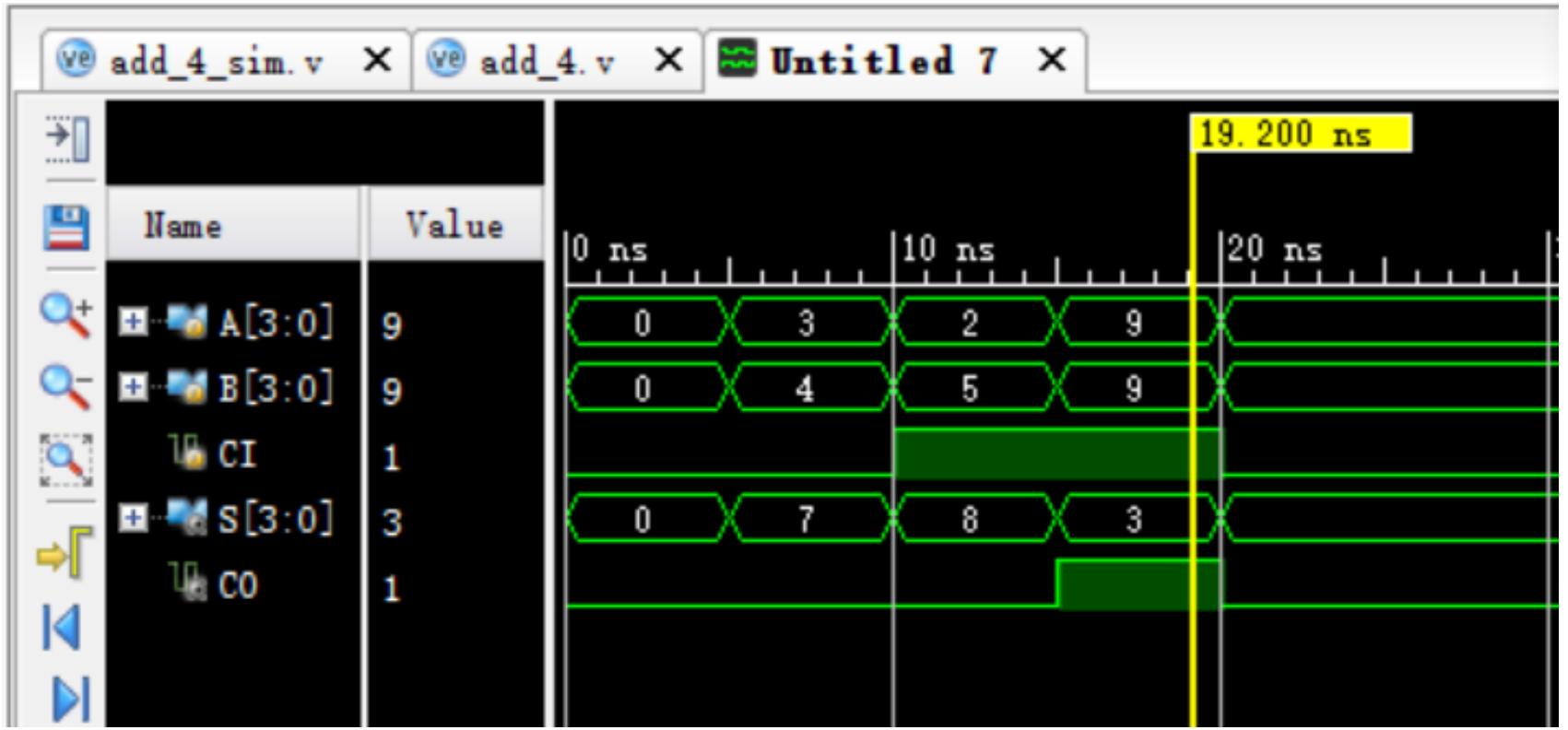
```
'timescale  1ns/1ns      //时间单位为1ns，精度为1ns
module module_name_sim();       //模块名（无端口声明列表）
      reg [信号位宽] ;          //激励信号声明
      wire [信号位宽] ;         //输出信号声明
      module_name    instance_name (port_associations) ;    //实例化设计模块
     initial
          begin      //激励信号
             PS=1’b0;PD1=1’b1;     //语句1
             #5 PS=1’b0;PD1=1’b1;     //语句2
                      ……
          end
endmodule
```

4位全加器模块测试

```
`timescale 1ns / 1ps
module add_4_sim();
        reg [3:0] A,B;
        reg CI;
        wire [3:0] S;
        wire CO;
        add_4 A1(A,B,CI,S,CO);
        initial
            begin
                A=4'd0;B=4'd0;CI=1'b0;
                #5 A=4'd3;B=4'd4;CI=1'B0;
                #5 A=4'd2;B=4'd5;CI=1'b1;
                #5 A=4'd9;B=4'd9;CI=1'b1;
                #5 A=4'd0;B=4'd0;CI=1'b0;
            end
endmodule
```

```
module add_4 ( input [3:0] a, input [3:0]b,
                    input c_in, output [3:0] sum,
                    output c_out );
    wire [3:0] c_tmp;
    full_adder i0 ( a[0], b[0], c_in, sum[0],
                    c_tmp[0]);
    full_adder i1 ( a[1], b[1], c_tmp[0],
                    sum[1], c_tmp[1] );
    full_adder i2 ( a[2], b[2], c_tmp[1],
                    sum[2], c_tmp[2] );
    full_adder i3 ( a[3], b[3], c_tmp[2],
                    sum[3], c_tmp[3] );
    assign c_out = c_tmp[3];
endmodule
```

4位全加器结果


add_4_sim.v
add_4.v
Untitled 7
Name
Value
A[3:0]
9
B[3:0]
9
CI
1
S[3:0]
3
CO
1
19.200 ns
0 ns
10 ns
20 ns
0
3
2
9
0
4
5
9
0
7
8
3

Verilog HDL与C语言的最大区别

- Verilog HDL语言是并行的，即具有在同一时刻执行多任务的能力，因为在实际硬件中许多操作都是在同一时刻发生的。一般来讲，计算机编程语言是非并行的。

- Verilog HDL语言有时序的概念，因为在硬件电路中从输入到输出总是有延迟存在的。