Verilog HDL
硬件描述语言

刘 鹏

浙江大学信息与电子工程学院

liupeng@zju.edu.cn

HDLs

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - *Structural descriptions* similar to hierarchical netlist.
  - *Behavioral descriptions* use higher-level constructs (similar to conventional programming).
- Originally designed to help in abstraction and simulation.
  - Now “logic synthesis” tools exist to automatically convert from behavioral descriptions to gate netlist.
  - Greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs!

- “Structural” example:

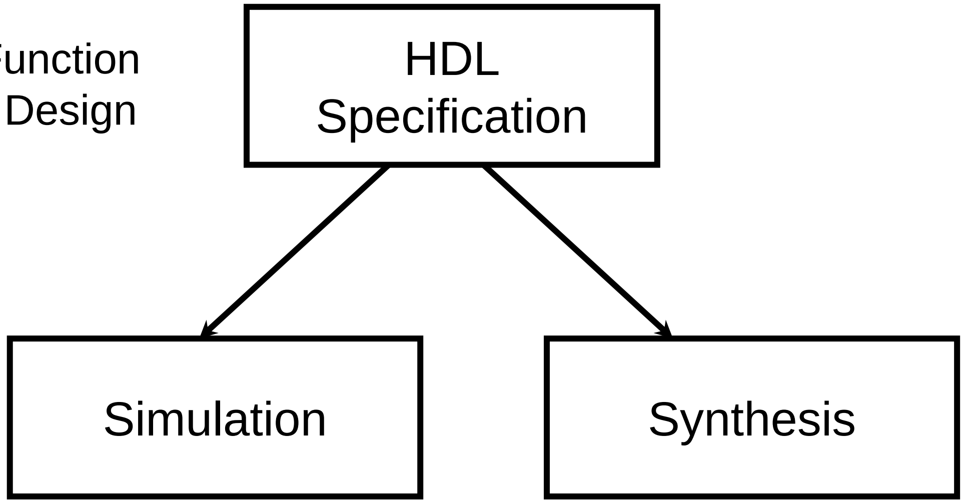
```
Decoder(output x0,x1,x2,x3;
            inputs a,b)
{
  wire abar, bbar;
  inv(bbar, b);
  inv(abar, a);
  nand(x0, abar, bbar);
  nand(x1, abar, b   );
  nand(x2, a,    bbar);
  nand(x3, a,    b   );
}
```

- “Behavioral” example:

```
Decoder(output x0,x1,x2,x3;
            inputs a,b)
{
   case [a b]
    00: [x0 x1 x2 x3] = 0x0;
    01: [x0 x1 x2 x3] = 0x2;
    10: [x0 x1 x2 x3] = 0x4;
    11: [x0 x1 x2 x3] = 0x8;
   endcase;
}
```

Design Methodology

Structure and Function
(Behavior) of a Design



Verification: Design
Behave as Required?
Functional: I/O Behavior
Register-Level (Architectural)
Logic-Level (Gates)
Transistor-Level (Electrical)
Timing: Waveform Behavior

Generation: Map
Specification to
Implementation

Quick History of HDLs

- ISP (circa 1977) - research project at CMU
  - Simulation, but no synthesis
- Abel (circa 1983) - developed by Data-I/O
  - Targeted to programmable logic devices
  - Not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now Cadence)
  - Similar to Pascal and C, originally developed for simulation
  - Fairly efficient and easy to write
  - 80s Berkeley develops synthesis tools
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - Similar to Ada (emphasis on re-use and maintainability)
  - Simulation semantics visible
  - Very general but verbose
  - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
  - Explicit structure of the circuit
  - How a module is composed as an interconnection of more primitive modules/components
  - E.g., each logic gate instantiated and connected to others
- Behavioral
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior
  - E.g., different implementations of one Boolean function

Verilog Introduction

- the `module` describes a component in the circuit
- Two ways to describe:
  - Structural Verilog
    - List of components and how they are connected
    - Just like schematics, but using text
      - A net list
    - tedious to write, hard to decode
    - Essential without integrated design tools
  - Behavioral Verilog
    - Describe *what* a component does, not *how* it does it
    - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools
- Build up a hierarchy of modules

Structural Model - XOR

module name

port list

```
module xor_gate ( out, a, b );
   input     a, b;
   output    out;
   wire      abar, bbar, t1, t2;

   inverter invA (abar, a);
   inverter invB (bbar, b);
   and_gate and1 (t1, a, bbar);
   and_gate and2 (t2, b, abar);
   or_gate  or1 (out, t1, t2);

endmodule
```

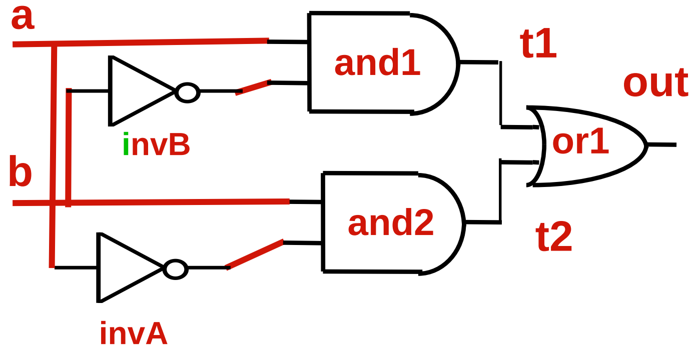
declarations

Built-in gates

statements

interconnections

Instance name



- Composition of primitive gates to form more complex module
- Note use of `wire` declaration!

By default, identifiers are wires

Structural Model: 2-to1 mux

```
//2-input multiplexor in gates
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);

endmodule // mux2
```

- Notes:
  - comments
  - “module”
  - port list
  - declarations
  - wire type
  - primitive gates
  - Instance names?
  - List per type

Simple Behavioral Model

- Combinational logic
  - Describe output as a function of inputs
  - Note use of `assign` keyword: *continuous* assignment

```
module and_gate (out, in1, in2);
   input          in1, in2;
   output         out;

   assign out = in1 & in2;

endmodule
```

Output port of a *primitive* must be first in the list of ports

Restriction does not apply to modules

When is this evaluated?

2-to-1 mux Behavioral Description

```
// Behavioral model of 2-to-1
//  multiplexor.
module mux2 (in0,in1,select,out);
   input in0,in1,select;
   output out;
   //
   reg out;
   always @ (in0 or in1 or select)
     if (select) out=in1;
     else out=in0;
endmodule // mux2
```

Sensitivity list

- Notes:
  - behavioral descriptions use the keyword **`always`** followed by blocking *procedural* assignments
  - Target output of procedural assignments must of type `reg` **(not a real register)**
  - Unlike `wire` types where the target output of an assignment may be continuously updated, a `reg` type retains it value until a new value is assigned (the assigning statement is executed).
  - Optional initial statement

Behavioral 4-to1 mux

```
//Does not assume that we have
// defined a 2-input mux.

//4-input mux behavioral description
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
      case (select)
             2'b00: out=in0;
             2'b01: out=in1;
             2'b10: out=in2;
             2'b11: out=in3;
      endcase
endmodule // mux4
```

- Notes:
  - No instantiation
  - Case construct equivalent to nested if constructs.
  - *Definition:* A structural description is one where the function of the module is defined by the instantiation and interconnection of sub-modules.
  - A behavioral description uses higher level language constructs and operators.
  - Verilog allows modules to mix both behavioral constructs and sub-module instantiation.

Mixed Structural/Behavioral Model

- Example 4-bit ripple adder

```
module full_addr (S, Cout, A, B, Cin);
  input     A, B, Cin;
  output    S, Cout;

  assign {Cout, S} = A + B + Cin;
endmodule

module adder4 (S, Cout, A, B, Cin);
  input  [3:0] A, B;
  input        Cin;
  output [3:0] S;
  output       Cout;
  wire         C1, C2, C3;

  full_addr fa0 (S[0], C1,  A[0], B[0], Cin);
  full_addr fa1 (S[1], C2,  A[1], B[1], C1);
  full_addr fa2 (S[2], C3,  A[2], B[2], C2);
  full_addr fa3 (S[3], Cout, A[3], B[3], C3);
endmodule
```

Behavior

Structural

Order of ports?

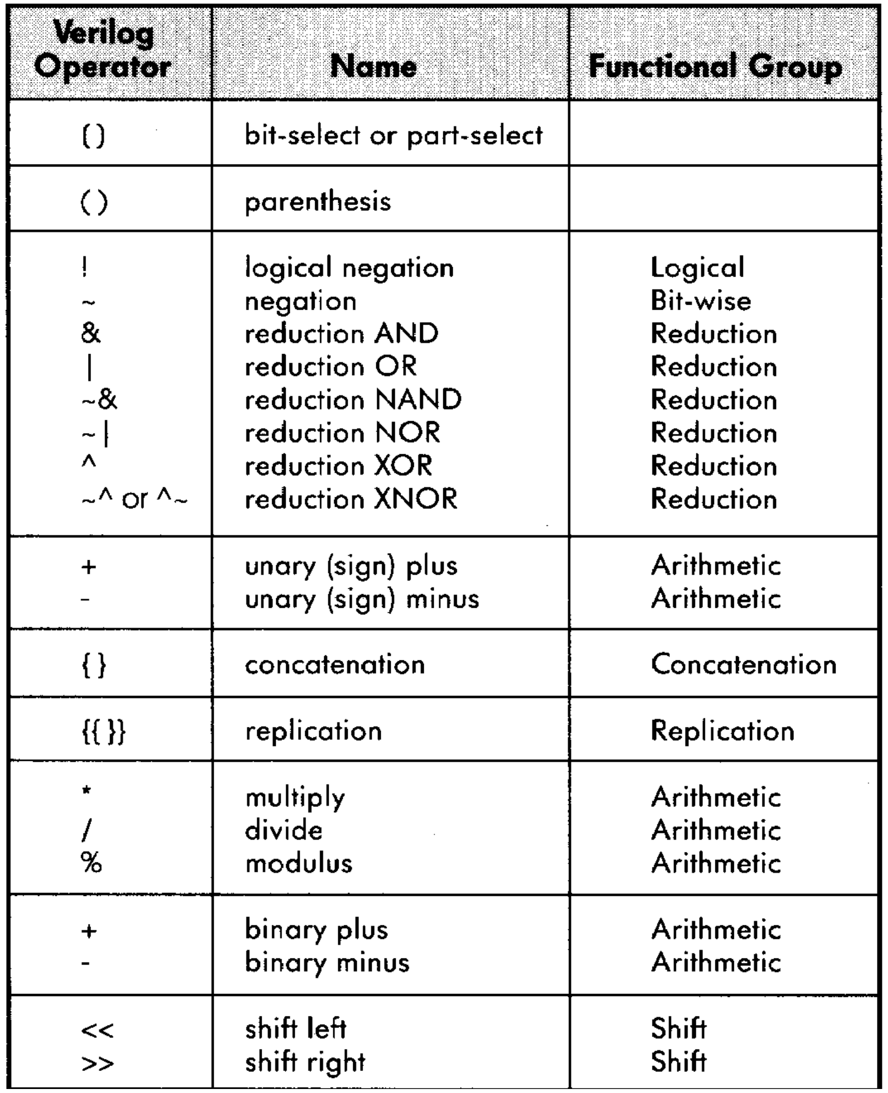
Verilog Data Types and Values

- Bits - value on a wire
  - `0, 1`
  - `X` - don't care/don't know
  - `Z` - undriven, tri-state
- Vectors of bits
  - **`A[3:0]` - vector of 4 bits: `A[3], A[2], A[1], A[0]`**
  - Treated as an *unsigned* integer value
    - e.g. , `A < 0` ??
  - Concatenating bits/vectors into a vector
    - e.g., sign extend
    - `B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};`
    - `B[7:0] = {4{A[3]}, A[3:0]};`
  - Style: Use `a[7:0] = b[7:0] + c;`
    Not: `a = b + c;` // need to look at declaration

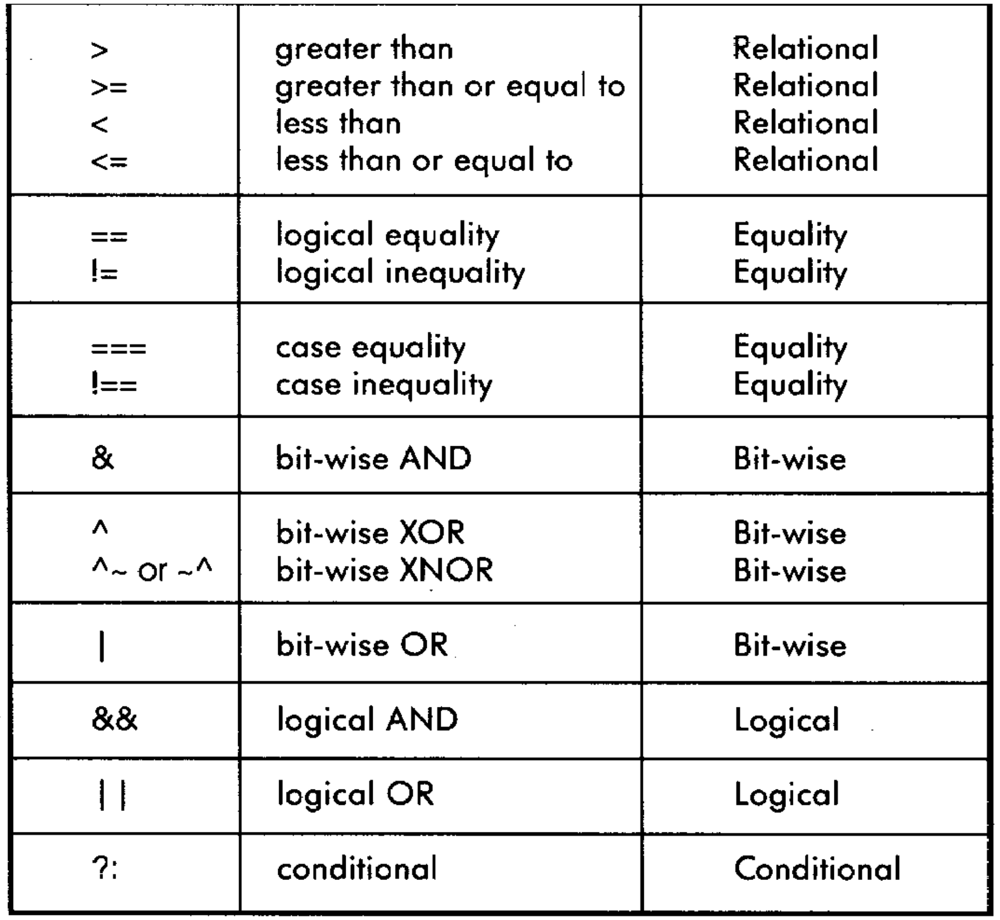
Verilog Numbers

- `14` - ordinary decimal number
- `-14` - 2’s complement representation
- `12’b0000_0100_0110` - binary number with 12 bits (_ is ignored)
- `12’h046` - hexadecimal number with 12 bits
- Verilog values are *unsigned*
  - e.g., `C[4:0] = A[3:0] + B[3:0];`
  - if A = 0110 (6) and B = 1010(-6)
    C = 10000 not 00000
    i.e., B is zero-padded, not sign-extended

Verilog Operators



Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
|	reduction OR	Reduction
~&	reduction NAND	Reduction
~|	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift



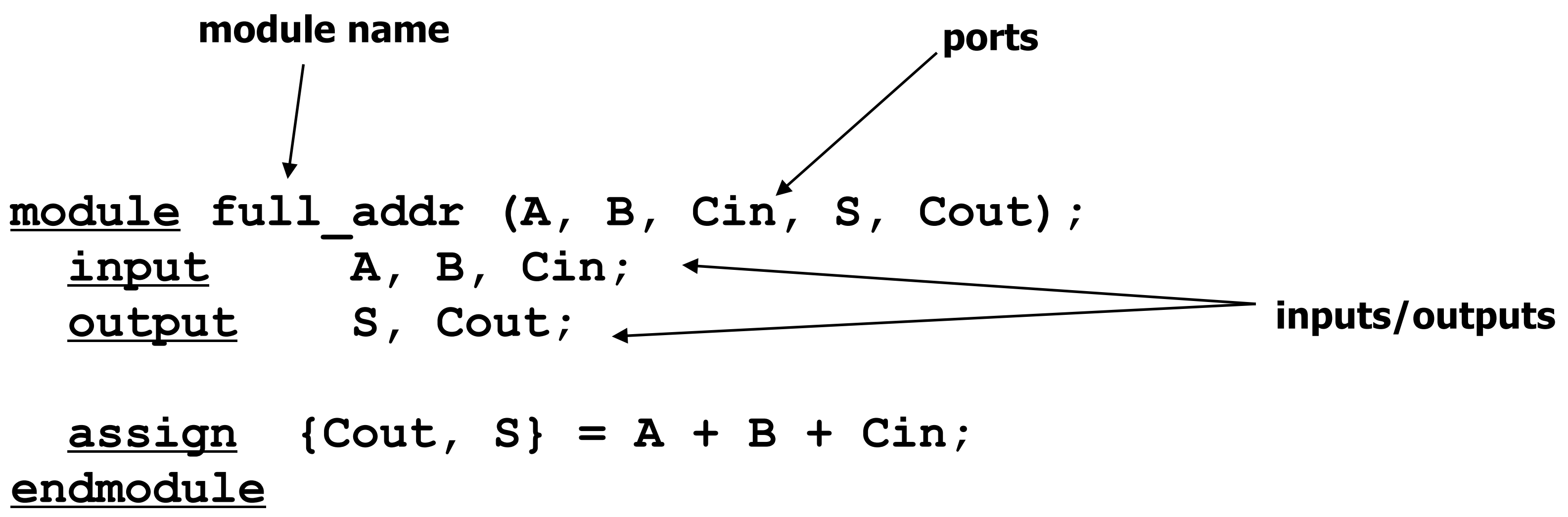
>	greater than	Relational
>=	greater than or equal to	Relational
<	less than	Relational
<=	less than or equal to	Relational
==	logical equality	Equality
!=	logical inequality	Equality
===	case equality	Equality
!==	case inequality	Equality
&	bit-wise AND	Bit-wise
^	bit-wise XOR	Bit-wise
^~ or ~^	bit-wise XNOR	Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
| |	logical OR	Logical
?:	conditional	Conditional

Verilog Variables

- wire
  - Variable used simply to connect components together
- reg
  - Variable that saves a value as part of a behavioral description
  - Usually corresponds to a wire in the circuit
  - Is *NOT* necessarily a register in the circuit
- usage:
  - Don’t confuse reg assignments with the combinational continuous `assign` statement!
  - `Reg` should only be used with `always` blocks (sequential logic, to be presented …)

Verilog Module

- Corresponds to a circuit component
  - “Parameter list” is the list of external connections, aka “ports”
  - Ports are declared “input”, “output” or “inout”
    - inout ports used on tri-state buses
  - Port declarations imply that the variables are wires



```
module full_addr (A, B, Cin, S, Cout);
  input     A, B, Cin;
  output    S, Cout;

  assign  {Cout, S} = A + B + Cin;
endmodule
```

Verilog Continuous Assignment

- Assignment is continuously evaluated
- `assign` corresponds to a connection or a simple component with the described function
- Target is *NEVER* a reg variable
- Dataflow style

```
assign A = X | (Y & ~Z);

assign B[3:0] = 4'b01XX;

assign C[15:0] = 4'h00ff;

assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;
```

use of Boolean operators
(~ for bit-wise, ! for logical negation)

bits can take on four values
(0, 1, X, Z)

variables can be n-bits wide
(MSB:LSB)

use of arithmetic operator

multiple assignment (concatenation)

delay of performing computation, only used by simulator, not synthesis

Comparator Example

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign Equal = (A & B) | (~A & ~B);
  assign Alarger = (A & ~B);
  assign Blarger = (~A & B);
endmodule
```

Comparator Example

```
// Make a 4-bit comparator from 4 1-bit comparators

module Compare4(A4, B4, Equal, Alarger, Blarger);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire e0, e1, e2, e3, Al0, Al1, Al2, Al3, B10, Bl1, Bl2, Bl3;

  Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
  Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
  Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
  Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);

  assign Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (Al3 | (Al2 & e3) |
                   (Al1 & e3 & e2) |
                   (Al0 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule
```

可综合的组合逻辑电路设计

组合逻辑电路

- ❑ 任何时刻电路的输出仅与该时刻的输入有关的数字电路被称为组合逻辑电路
- ❑ 组合电路不含有反馈，不含有记忆元件，仅仅通过若干门电路连接实现的
- ❑ 组合逻辑的表达一般有3种方式：
  - ❑ 真值表、逻辑表达式、电路原理图

可综合的组合逻辑电路设计

使用Verilog HDL描述组合电路有多种方式

(1) 与真值表对应地是用户自定义原语

(2) 与电路图对应地是门级建模即结构描述

(3) 与逻辑表达式对应地称为行为描述

可综合的组合逻辑电路设计

用Verilog语言可以有多种方式描述组合电路，但是有些综合工具不支持每一种描述方式

大多数综合工具可综合下面三种形式描述的组合电路

✓ **门级模型、数据流模型、行为级模型**

（1）**门级模型**

- ❑ 门级模型是描述逻辑门以及逻辑门之间连接关系的模型，通过门原语描述电路

可综合的组合逻辑电路设计

Verilog HDL中提供了丰富的门类型关键字，用于电路的门级描述。

Verilog HDL有关类型的关键字比较常用的有：

not	非门	**nor**	或非门
and	与门	**xor**	异或门
nand	与非门	**xnor**	异或非门
or	或门	**buf**	缓冲器

可综合的组合逻辑电路设计

调用门原语的句法如下：
门类型关键字　实例化的门名称(端口列表)
端口列表按下列顺序列出：
(输出，输入1，输入2，输入3 ，　……)；
对于三态门，则按如下顺序列出端口
(输出，输入，使能控制端)；
比如：

```
and      myand(out, in1, in2, in3);  //三输入与门
and      amd1(out, in1, in2);        //二输入与门
buffif1 mytri(out, in, enable);      //高电平使能的三态门
```

可综合的组合逻辑电路设计

例: 2-4译码器

```
 module decoder(D, A, B, enable)
 Output  [ 3:0]  D;
 input            A, B;
 Input            enable;
 wire             A_not, B_not, enable_not;
 not  G1(A_not, A) ,
      G2(B_not, B),
      G3(enable, enable_not);
nand  G4(D[0], A_not, B_not, enable_not) ,
       G5(D[1], A_not, B, enable_not) ,
       G6(D[2], A, B_not, enable_not) ,
       G7(D[3], A, B, enable_not) ;
endmodule
```

可综合的组合逻辑电路设计

数据流模型

- 利用连续赋值语句assign来描述一个组合电路
- 综合工具可以把用连续赋值描述的电路翻译成布尔等式并优化
- 它对操作数进行不同的操作以得到需要的结果，提供了多种对操作数的操作

可综合的组合逻辑电路设计

symbol	**operation**
+	binary addition
–	binary subtraction
&	bitwise AND
^	bitwise XOR
|	bitwise OR
~	bitwise not
==	equality
>	greater than
<	less than
{ }	concatenation
? :	conditional

可综合的组合逻辑电路设计

例: 4位加法器

```
module adder4(cout, sum, ina, inb, cin);
      output [3: 0]  sum;
      output cout;
      input [3: 0]  ina, inb;
      input cin;
      assign {cout, sum}  =  ina + inb + cin;
 endmodule
```

可综合的组合逻辑电路设计

例: 4位比较器

```
module mag_compare
    ( output A_it_B, A_eq_B, A_gt_B,
      input [3:0]  A, B
    );
      assign A_it_B = (A<B);
      assign A_gt_B = (A>B);
      assign A_eq_B = (A=B);
endmodule
```

可综合的组合逻辑电路设计

行为级模型

- 行为级模型通常用来描述时序逻辑电路，但有时候也可以用来描述组合逻辑电路
- 行为级模型使用关键词always来进行描述，其声明格式如下：
- always <时序控制 or 事件控制> <语句>
- always语句在仿真过程中是不断活动的
- always语句后面跟着的过程快是否执行，则要看它的触发条件是否满足

可综合的组合逻辑电路设计

例: 2-to-1选择器

```
module mux(m_out, A, B, select);
      output  m_out;
      input A, B, select;
      reg  m_out;
      always @(A or B or select);
      if (select == 1) m_out = A;
      else m_out = B;
endmodule
```

当信号A，B，select中的任何一个发生变化时，将会触发下面语句执行

可综合的组合逻辑电路设计

例: 4-to-1选择器

```
module mux
     ( output  reg m_out, input  in_0, in_1, in_2, in_3， input [1:0] select);

 always @(in_0, in_1, in_2, in_3, select)
     case(select)
          2’b00:     m_out = in_0;
          2’b01:     m_out = in_1;
          2’b10:     m_out = in_2;
          2’b11:     m_out = in_3;
     endcase

endmodule
```

可综合的组合逻辑电路设计

Test Bench

- 编写test bench 的目的是验证设计的正确性。通过模拟各种可能的情况查看输入输结果是否符合设计的要求。简单的test bench要向设计提供向量，人工验证输出

- 除了使用always语句外，它还使用initial来产生激励给被测试的模块

可综合的组合逻辑电路设计

initial 模块如下：

```
initial
 begin
    A = 0;
    B = 0;
 #10 A = 1;
 #20 A = 0; B = 1;
 end
```

在这个例子中用initial语句在仿真开始时对变量A，B进行初始化，这个初始化的过程不需要任何仿真时间，在10ns之后，变量A变1,20ns之后变量A为0，B为1。

可综合的组合逻辑电路设计

Verilog中常用的系统任务

$display(p1,p2,…,pn);将参数p2到pn按p1给定的格式输出，自动的在输出后换行

$write（p1p2 ,…,pn ； 与$display相同，只是在输出后不换行

$monitor 提供了监视和输出参数列表中的表达式或变量值的功能

$time 返回一个64位的整数来表示当前仿真时刻

$finish 退出仿真

可综合的组合逻辑电路设计

激励模块的格式通常如下所示：

Module test_module_name

  //Declare local **reg** and **wire** identifiers（标识符）

  //Instantiate(实例化) the design module under test

  //Specify a stopwatch, using $finish to terminate the simulation

  //Generate stimulus ,using **initial** and **always** statements

  //Display the output response

endmodule

可综合的组合逻辑电路设计

例: 2-1选择器的test bench

```
 module mux;
   wire t_mux_out;
   reg  t_A,t_B;
   reg  t_select;
   parameter   stop_time=50;
mux  M1(t_mux_out, t_A,t_B, t_select);
initial # stop_time $finish;
initial begin
         t_select = 1; t_A = 0; t_B = 1;
#10   t_A = 1; t_B = 0;
```

可综合的组合逻辑电路设计

```
     #10 t_select = 0;
     #10 t_A = 0; t_B = 1;
end
initial begin
$monitor(“time=“”, $time, “select=%b A=%b B=%b OUT=%b,
 t_select, t_A,t_B, t_mux_out);
end
endmodule
```

测试

- 测试是一个专门用来给电路的HDL模型施加的一个激励的HDL程序，目的是测试和观察其在激励下的响应
- 典型的测试模块没有输入和输出，加到设计模块用于模拟的输入信号在激励模块中定义为局部**reg型**数据。显示设计模块的测试输出在激励模块中定义为局部**wire型**数据。用局部的标识来例化测试的模块
- 测试模块的HDL格式：

```
module test_module_name;
  //Declare local reg and wire identifiers
  //Instantiate the design module under test
  //Specify a stopwatch, using $finish to terminate the simulation
  //Generate stimulus, using initial and always statements
  //Display the output response (text or graphics (or both))
endmodule
```

最终思考

- Verilog类似于C语言，Verilog描述硬件
  - 多个物理元素的并行性
  - 时序关系
  - 仿真和综合
  - 画出电路图，采用Verilog语言描述
- 理解语言的元素
  - 模块、端口、线、寄存器、原语、连续赋值、阻塞赋值、敏感列表、层次化
  - 最好方式：实践动手
- 行为级结构隐藏了电路细节
- 设计人员需要管理结构、数据通信、并行性和设计的时序