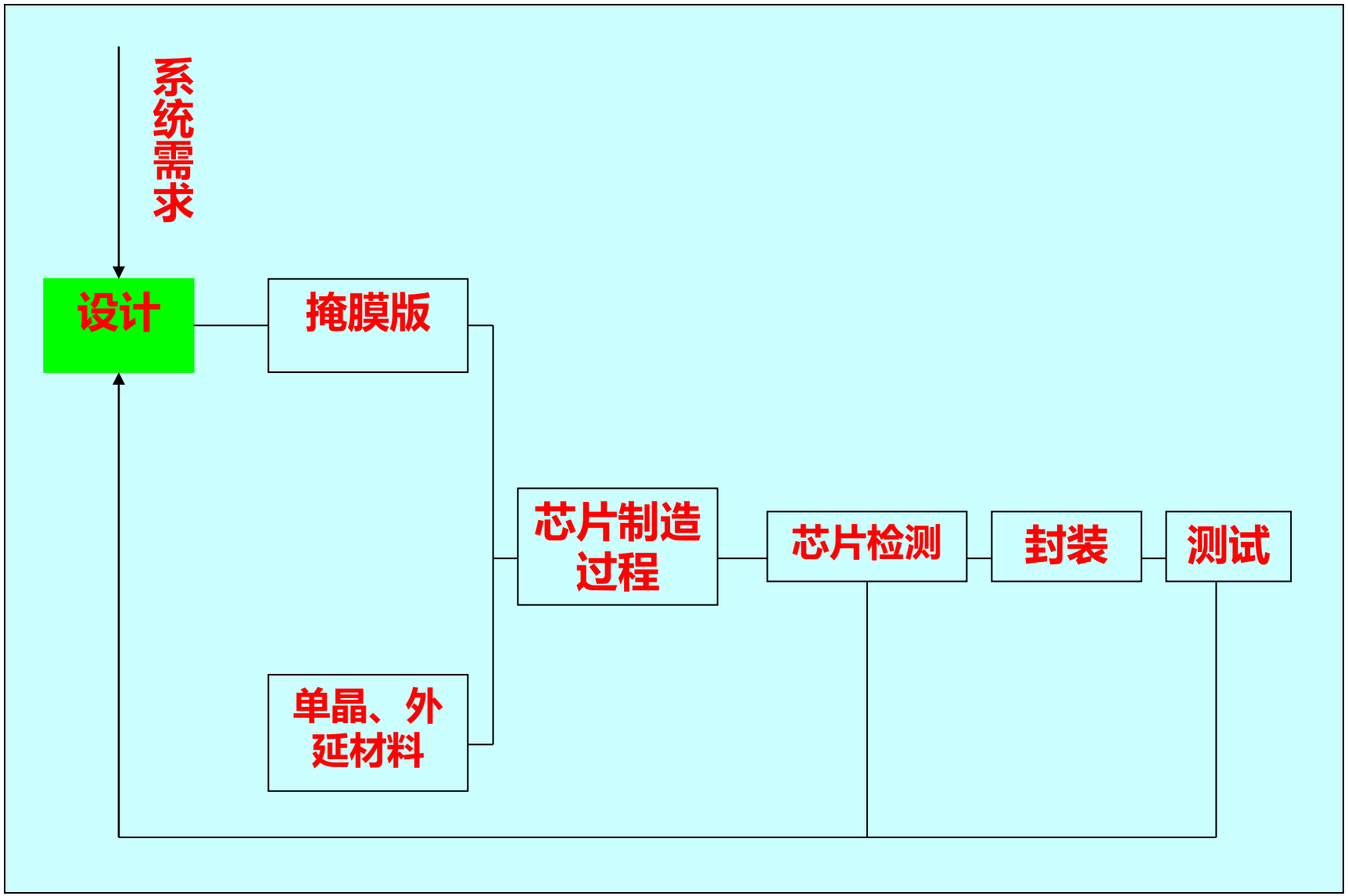
西安电子科技大学

集成电路导论

任课教师：齐佩汉
菁英教授/博导
phqi@xidian.edu.cn
15829723431（微信同号）

集成电路设计与制造的主要流程框架

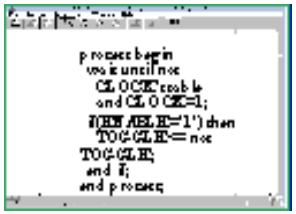


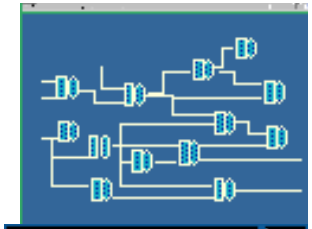
集成电路的设计过程

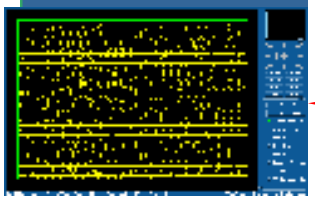
设计创意

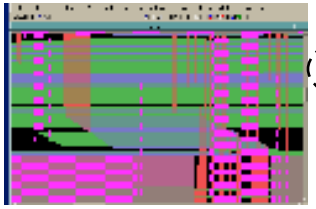
+

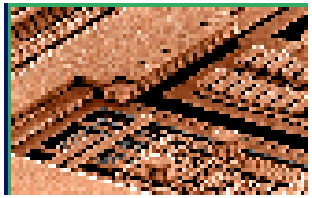
仿真验证

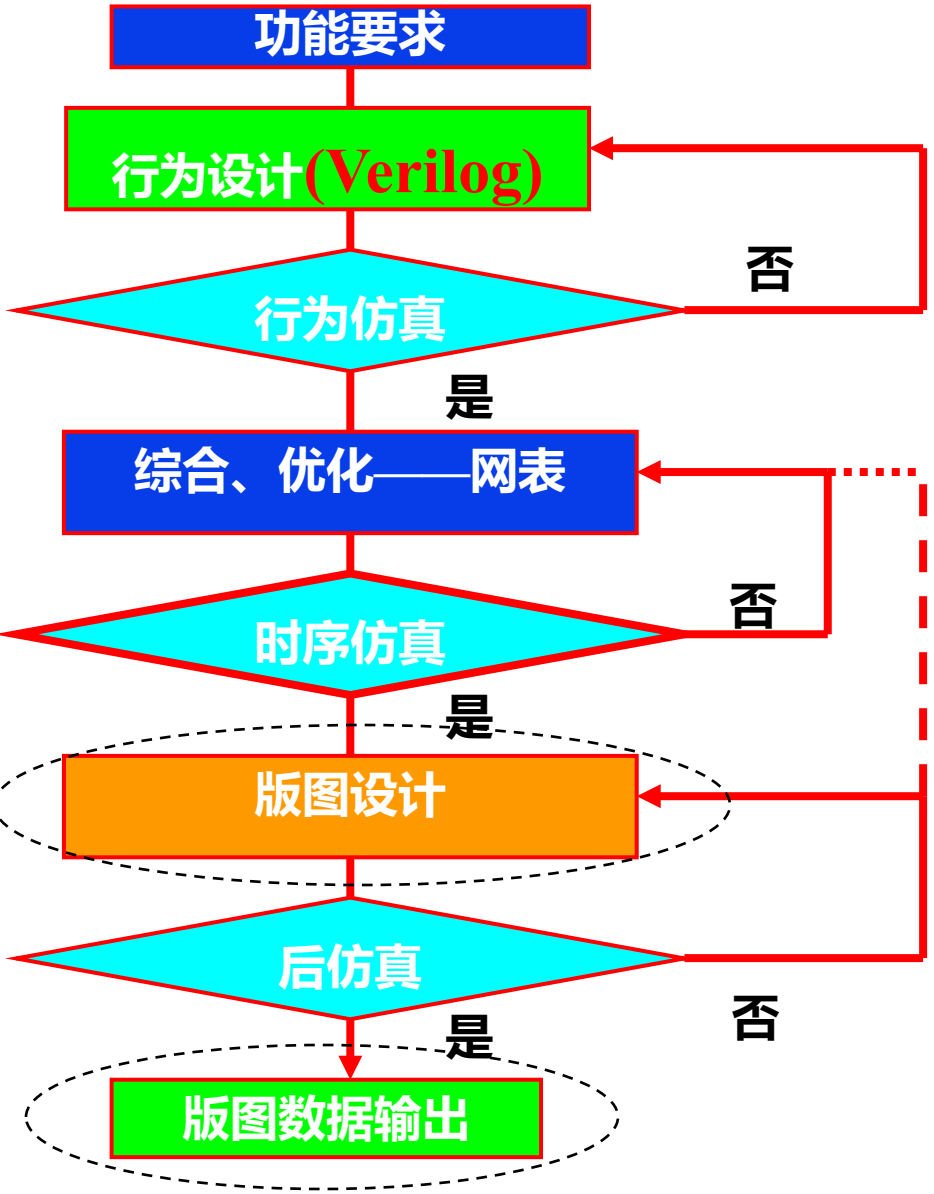






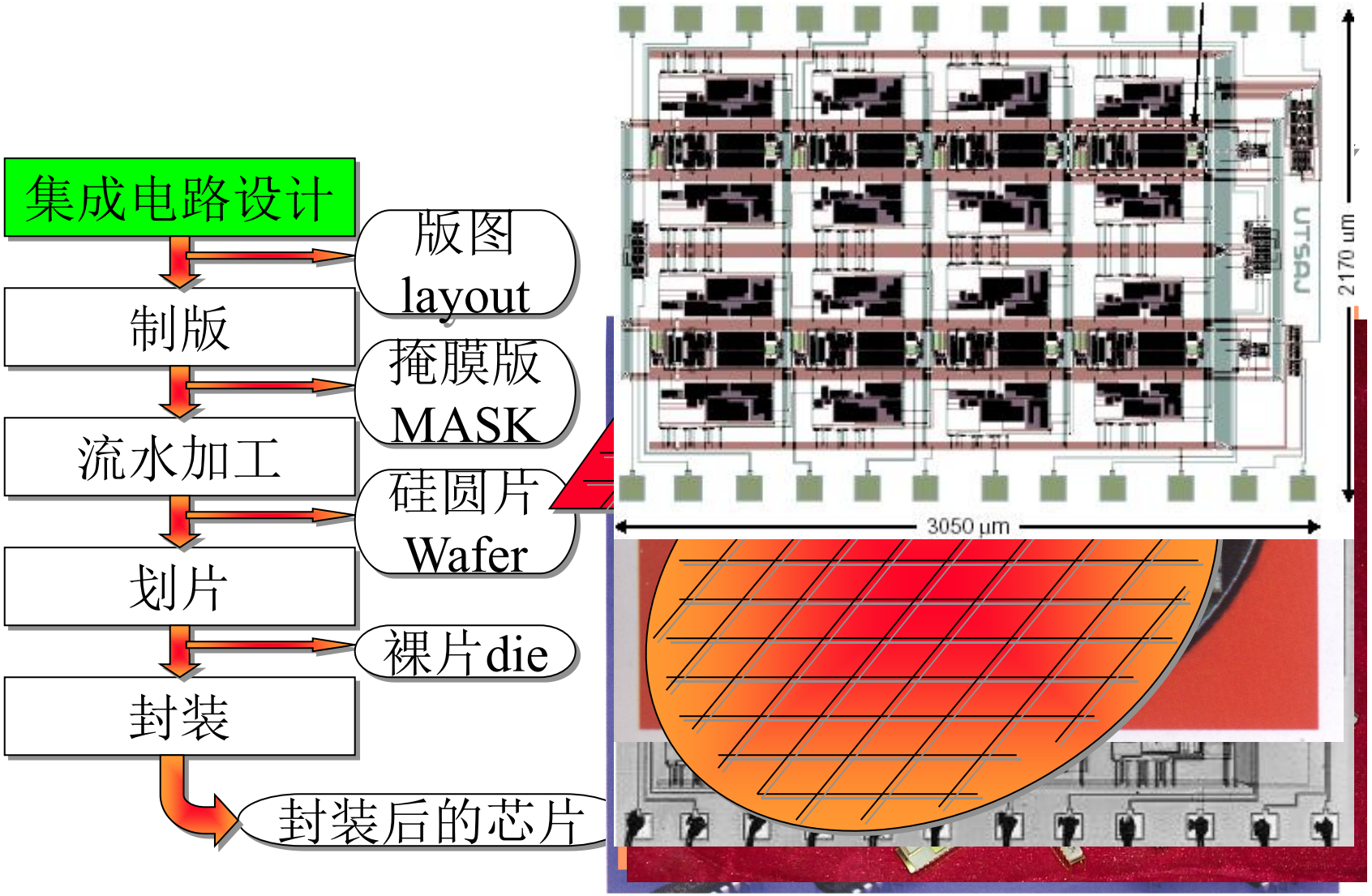






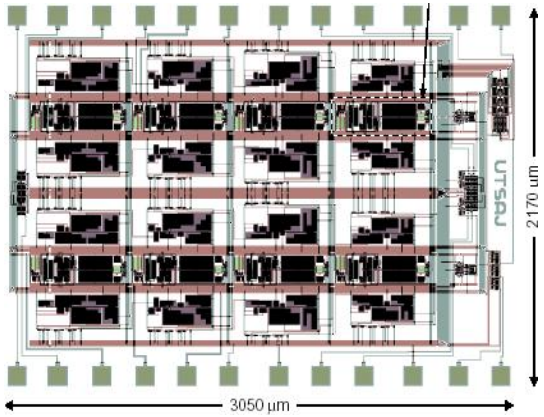
✸集成电路设计的最终输出是掩膜版图，通过制版和工艺流片可以得到所需的集成电路。

集成电路的制造过程

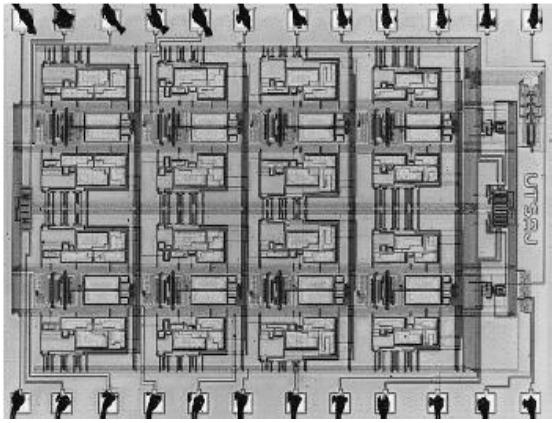

集成电路设计
制版
流水加工
划片
封装
封装后的芯片
版图
layout
掩膜版
MASK
硅圆片
Wafer
裸片die
UTSAJ
2170 µm
3050 µm

集成电路版图的作用

芯片加工：从版图到裸片



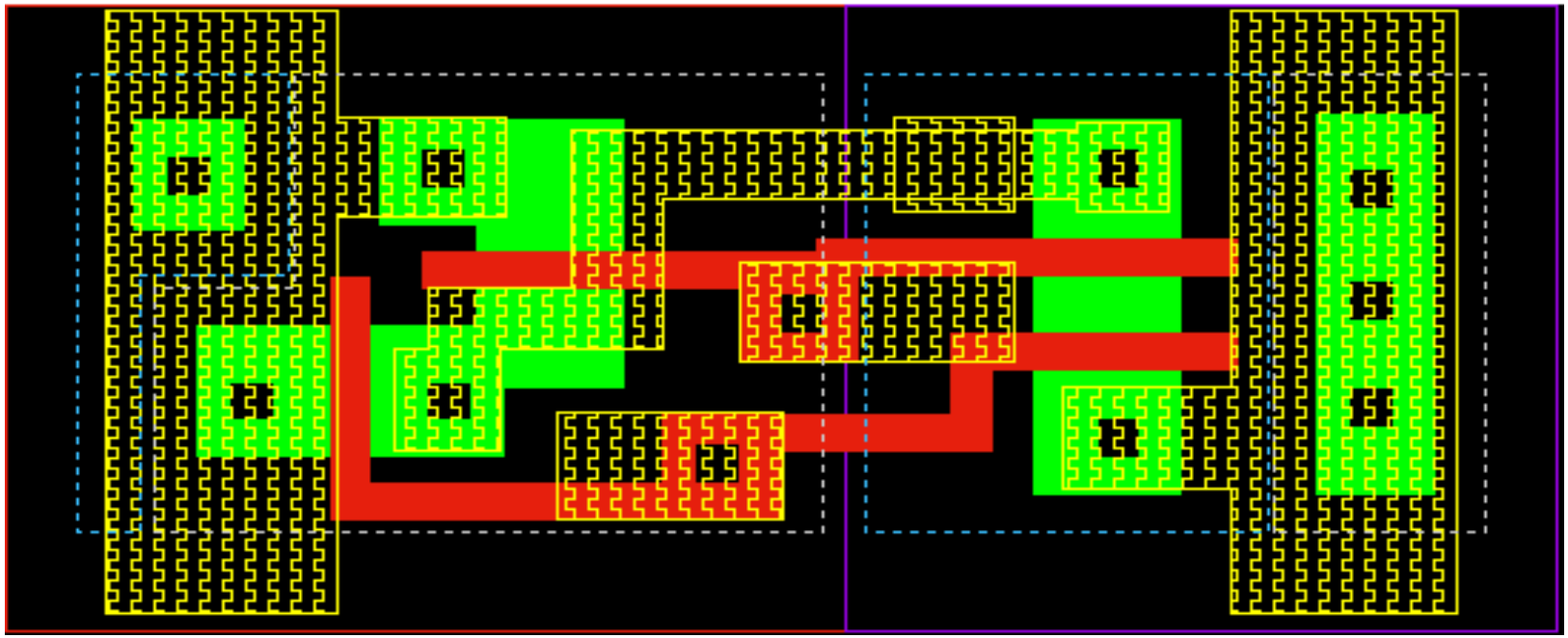
制版 → 加工 →



是一种多层平面“印刷”和叠加过程。

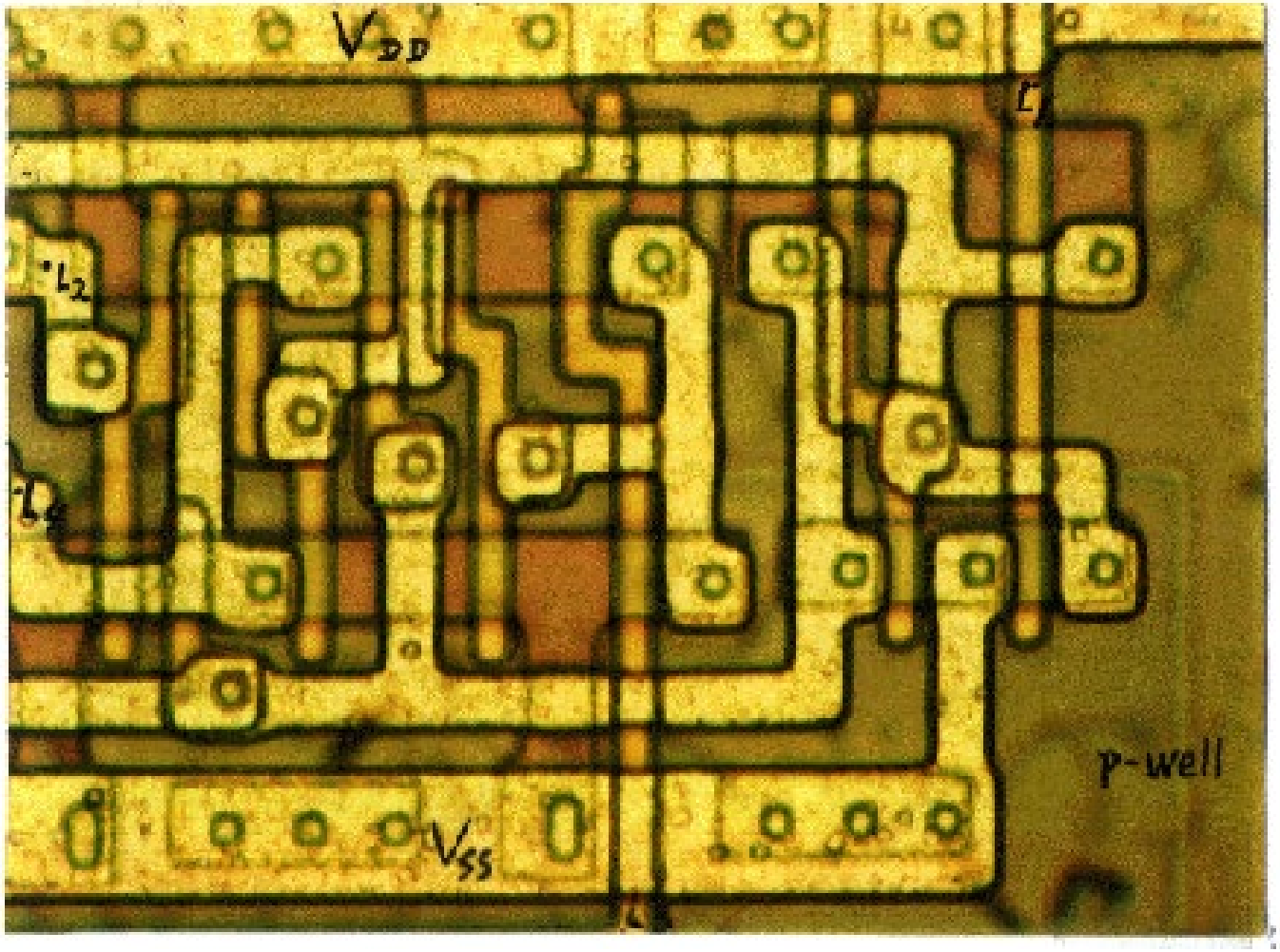
集成电路版图的作用

所设计的版图



集成电路版图的作用

加工后得到的实际芯片版图



西安电子科技大学

第三章

硬件描述语言简介

硬件描述语言简介

所谓硬件描述语言(HDL, Hardware Description Language)，就是可以描述**硬件电路功能**、**信号连接关系**及**定时关系**语言。它能比电原理图更有效地表示硬件电路的特性。

- 最具代表性的HDL
  - VHDL
  - Verilog HDL

VHDL与Verilog HDL的发展史

- VHDL的发展史

美国国防部在上个世纪70年代末和80年代初提出的VHSIC (Very High Speed Integrated Circuit)计划产物。1981年提出了一种新的硬件描述语言，简称为VHDL (VHSIC Hardware Description Language)。

- Verilog HDL的发展史

Gateway Design Automation公司于1983年创建的仿真与验证工具，之后又陆续开发了相关的故障仿真与时序分析工具，这是在C语言基础上发展起来的一种硬件描述语言。1989年Cadence公司收购GDA公司并促进了Verilog HDL的发展。

VHDL与Verilog HDL的标准化进程

- VHDL的标准化
- 1987年12月VHDL被接纳为IEEE std-1076-1987标准，一般称为 VHDL’87。
- 1993年进一步修订，形成IEEE std-1076-1993标准，称为VHDL’93。
- 随后又经过陆续修订，形成IEEE std-1076-2002、IEEE std-1076-2008等标准。

- Verilog HDL的标准化
- 1990年Cadence公司公开发表Verilog HDL，并成立OVI组织促进其发展。
- 1995年Verilog HDL成为IEEE标准，即IEEE std-1364-1995。
- 随后又经过陆续修订，形成IEEE std-1364-2001、IEEE std-1364-2005等标准。

VHDL与Verilog HDL的比较

1. 从推出过程来看，VHDL偏重于标准化方面的考虑，而Verilog HDL与EDA工具的结合更为紧密。
2. 与VHDL相比，Verilog HDL的编程风格更加简洁明了、高效便捷。如果从描述结构上考察，两者的代码比为3:1。
3. 目前市场上的EDA工具绝大部分支持这两种语言，而在ASIC设计领域，Verilog HDL占有优势。

推荐书目

- [美] Michael D Ciletti著,李广军 林水生 阎波 等译, Verilog HDL高级数字设计(第二版), 北京: 电子工业出版社, 2014.
- [美] Samir Palnitkar著, 夏宇闻 胡燕祥 刁岚松等译, Verilog HDL数字设计与综合(第二版)(本科教学版), 北京: 电子工业出版社, 2015.

3.1 Verilog HDL概述

采用Verilog的硬件电路设计方法

采用Verilog设计数字系统一般采用自上而下(Top Down)的分层设计方法，所谓自上而下的设计方法，就是从系统总体出发，自上而下地逐步将设计内容细化，最后完成系统硬件的整体设计。

1. 第一层次行为描述－就是对整个系统的数学模型的描述。
2. 第二层次RTL方式描述－即寄存器传输级描述，也称为数据流描述。采用RTL方式描述，才能导出系统的逻辑表达式，才能进行逻辑综合。
3. 第三层次是逻辑综合－就是利用逻辑综合工具，将RTL方式描述的程序转换成用基本元件表示的文件(门级网表)。

采用Verilog设计硬件电路的优点

1. 设计技术齐全、方法灵活、支持广泛。Verilog语言可以支持自上而下和基于库的设计方法，而且支持同步电路、异步电路、FPGA以及其它随机电路的设计。
2. 系统硬件描述能力强，能支持硬件的设计、验证、综合和测试，是一种多层次的硬件描述语言。
3. Verilog语言可以与工艺无关编程。当门级或门级以上的描述通过仿真验证后，再利用相应的工具将设计映射成不同的工艺(如MOS、CMOS等)。这样，在工艺更新时，就无须修改原设计程序，只要改变相应的映射工具就行了。
4. Verilog语言标准、规范，易于共享和重复利用。

西安电子科技大学

3.2 Verilog的基本语法规则

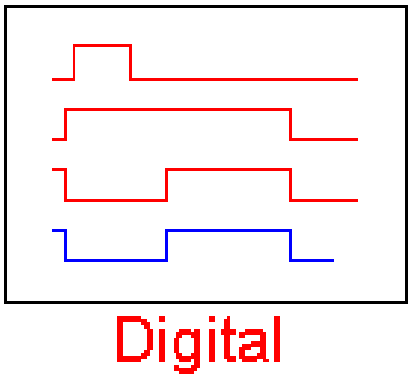
Verilog HDL的历史

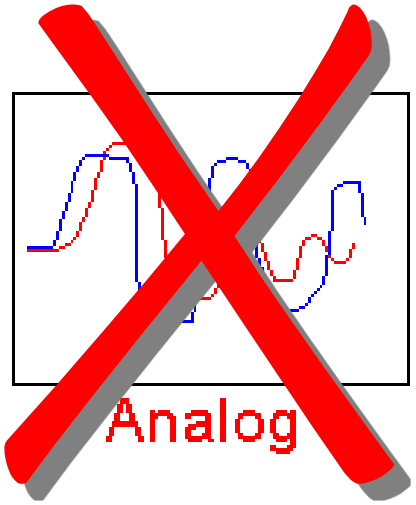
- Verilog HDL是在1983年由GDA(GateWay Design Automation)公司的Phil Moorby所创。Phil Moorby后来成为Verilog-XL的主要设计者和Cadence公司的第一个合伙人；
- 在1984~1985年间，Moorby设计出了第一个Verilog-XL的仿真器；
- 1987年，Synopsys公司的综合软件开始接受Verilog输入；
- 1989年，Cadence公司收购GDA，进一步扩大Verilog的影响；
- 1991年，Cadence公司公开发表Verilog语言，成立了OVI(Open Verilog International)组织来负责Verilog HDL语言的发展；
- 1993年，OVI推出Verilog2.0，作为IEEE提案提出申请；
- 1995年，IEEE(Institute of Electrical and Electronics Engineers)通过Verilog HDL标准IEEE 1364-1995；
- 2001年，IEEE 发布了Verilog IEEE 1364-2001标准；
- 2005年，推出IEEE 1364-2005标准。
- 2005年，IEEE发布System Verilog(IEEE 1800)，主要用于复杂半导体设计。

Verilog HDL的现状

- Verilog HDL是最广泛使用的、具有国际标准支持的硬件描述语言，绝大多数的EDA厂商都支持；
- 在工业界和ASIC设计领域，Verilog HDL应用更加广泛。

Verilog的基本限制：只能描述数字系统





Verilog HDL与VHDL

- 建模层次(抽象描述层次)
- 系统级(System): 用高级语言结构实现设计模块的外部性能的模型。
- 算法级(Algorithmic): 用高级语言结构实现设计算法的模型。
- RTL级(Register Transfer Level): 描述数据在寄存器之间流动和如何处理这些数据的模型。
- 门级(Gate-Level): 描述逻辑门以及逻辑门之间的连接的模型。
- 开关级(Switch-Level): 描述器件中三极管和储存节点以及它们之间连接的模型。

Verilog HDL　　VHDL

Verilog HDL与VHDL的比较

- 相同点：
  - 都能形式化抽象表示电路行为和结构；
  - 支持逻辑设计中层次与范围的描述；
  - 具有电路仿真和验证机制；
  - 与工艺无关。不专门面向FPGA或ASIC代工厂设计
- 不同点：
  - Verilog与C语言相似，语法灵活；VHDL源于Ada语言，语法严格；
  - Verilog更适合ASIC设计。

Verilog HDL的主要应用

- ASIC和FPGA工程师编写可综合的RTL代码
- 使用高抽象级描述仿真系统，进行系统结构开发
- 测试工程师用于编写各种层次的测试程序
- 用于ASIC和FPGA单元或更高层次的模块的模型开发

抽象级(Levels of Abstraction)

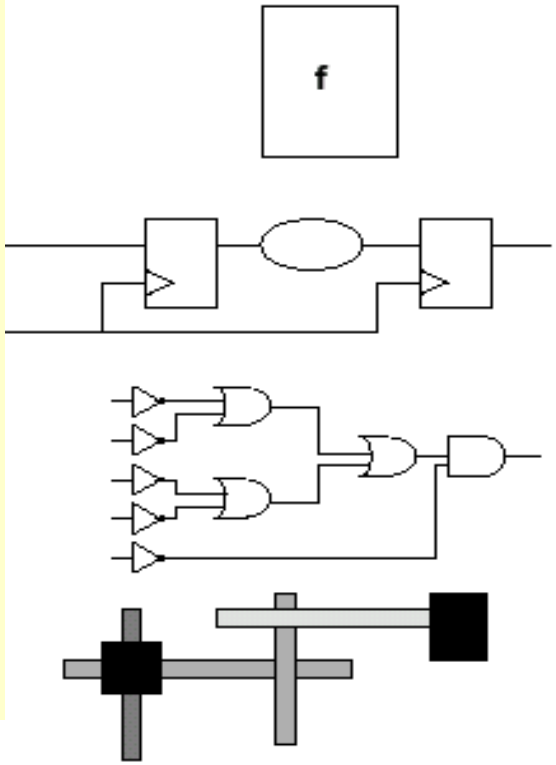
- Verilog既是一种行为描述的语言也是一种结构描述语言。Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别包括：

行为级(系统说明)
-设计文档/算法描述

RTL/功能级
-Verilog

门级/结构级
-Verilog

版图/物理级
-几何图形



行为综合
↓
综合前仿真
↓
逻辑综合
↓
综合后仿真
↓
版图

抽象级(Levels of Abstraction)

- Verilog可以在三种抽象级上进行描述

行为级(Behavioral level)

- 用功能块之间的数据流对系统进行描述
- 在需要时在函数块之间进行调度赋值。

RTL级/功能级(RTL level or Functional level)

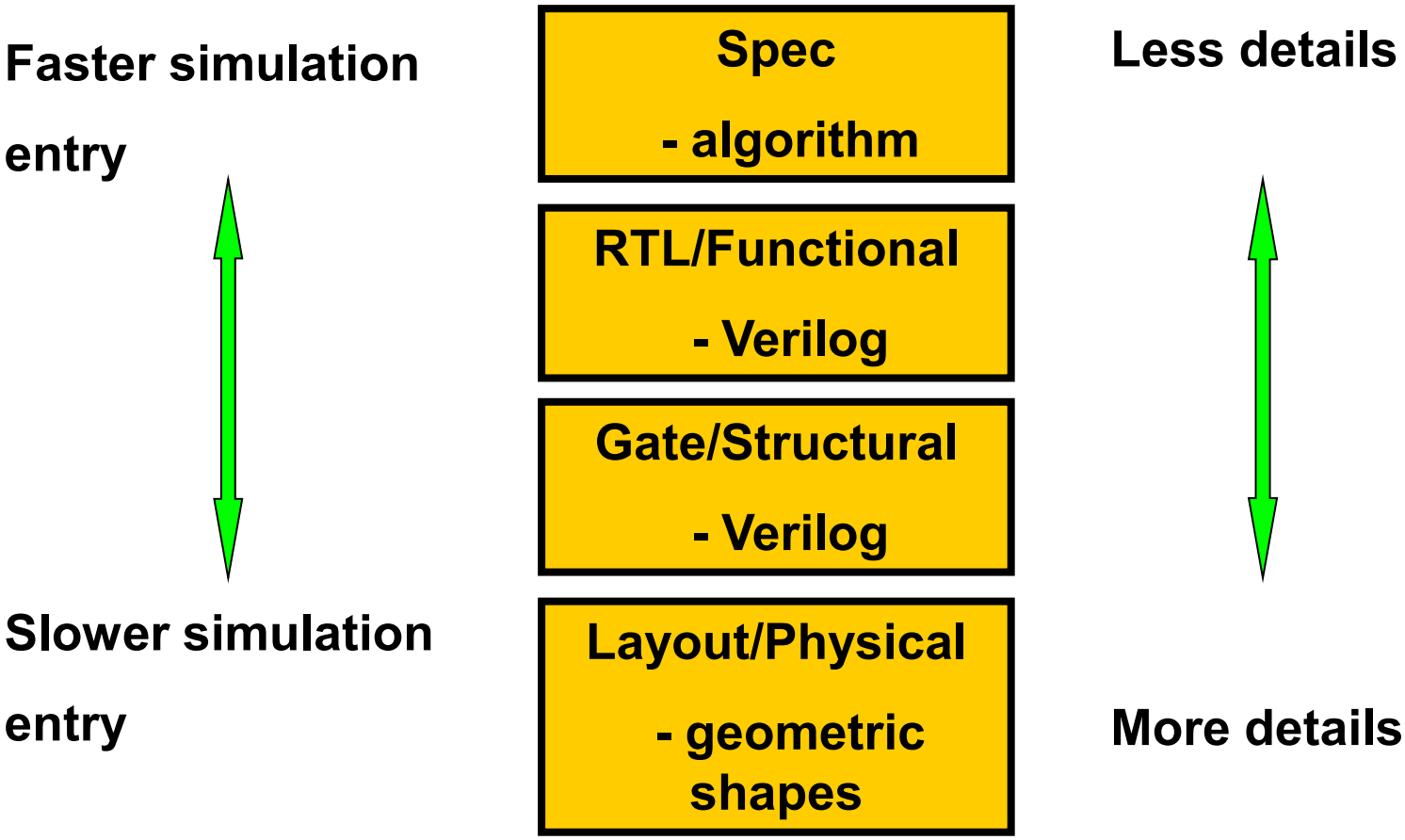
- 用功能块内部或功能块之间的数据流和控制信号描述系统
- 基于一个已定义的时钟的周期来定义系统模型

结构级/门级(Structural level or Gate level)

- 用基本单元(primitive)或底层元件(component)的连接来描述系统以得到更高的精确性，特别是时序方面。
- 在综合时用特定工艺和底层元件将RTL描述映射到门级网表

抽象级(Levels of Abstraction)

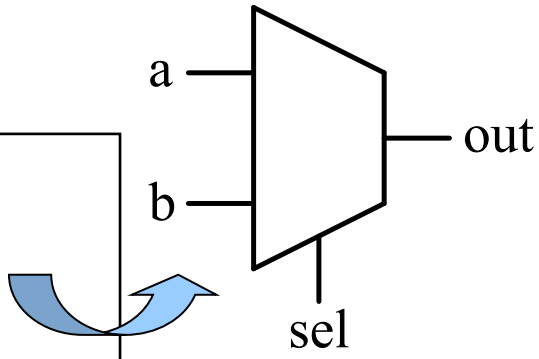
- 各级描述风格之间的Trade-offs

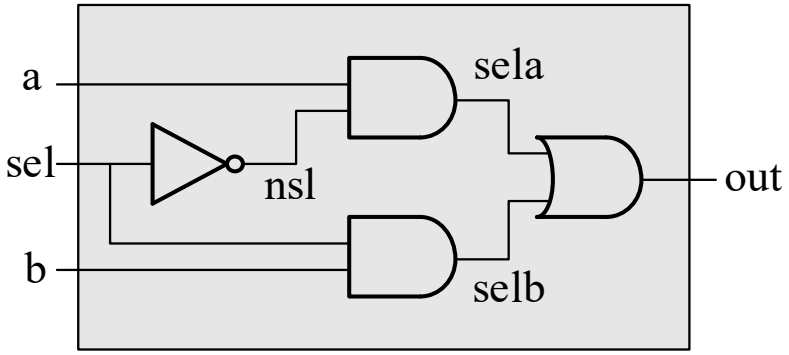


抽象级(Levels of Abstraction)

Behavioral & RTL

```
module muxtwo(out,a,b,sel);
input a, b, sel;
output out;
reg out;
always@(sel or a or b)
  if (!sel)
          out = a;
  else
          out = b;
endmodule
```





Structural

```
module muxtwo(out,a,b,sel);
input  a, b, sl;
output out;
wire nsl, sela, selb;
not          U1 (nsl, sel);
and      #1 U2 (sela, a,    nsl);
and      #1 U3 (selb, b,    sel);
or       #2 U4 (out,  sela, selb);
endmodule
```

抽象级(Levels of Abstraction)

- 混合设计方式的1位全加器实例

真值表

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

逻辑表达式

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + BC_{in} + AC_{in}$$

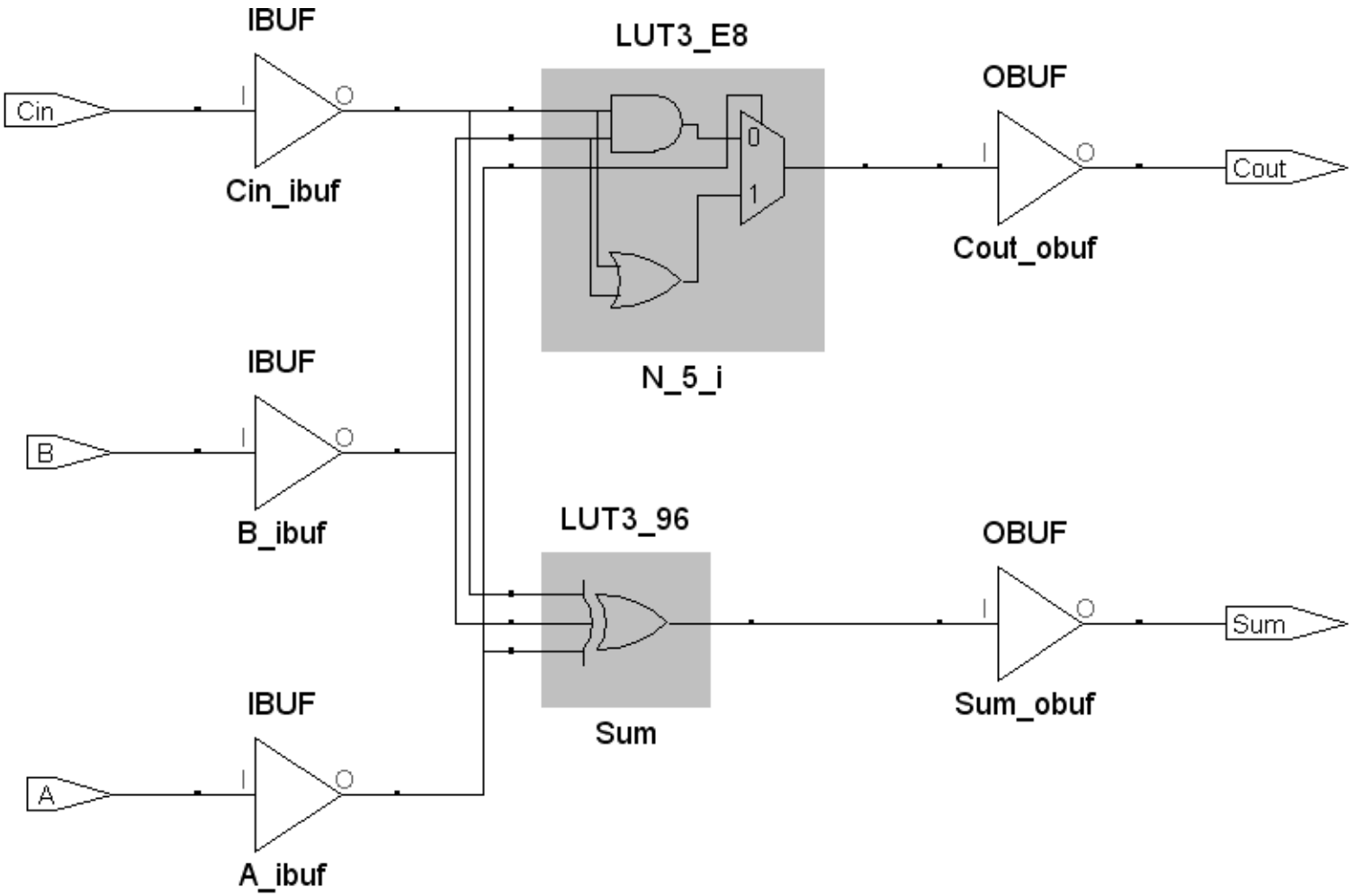
抽象级(Levels of Abstraction)

- 混合设计方式的1位全加器实例

```
module FA_Mix (A, B, Cin, Sum, Cout);
    input  A, B, Cin;
    output Sum, Cout;
    reg    Cout;
    reg    T1, T2, T3;
    wire   S1;
    xor X1(S1, A, B);                    // 门实例语句。
    always@(A or B or Cin)               // always语句。
    begin
          T1   = A & Cin;
          T2   = B & Cin;
          T3   = A & B;
          Cout = (T1 | T2) | T3;
    end
    assign Sum = S1 ^ Cin;               // 连续赋值语句。
endmodule
```

抽象级(Levels of Abstraction)

- 混合设计方式的1位全加器实例(综合)



Verilog语言结构

module是层次化设计的基本构件

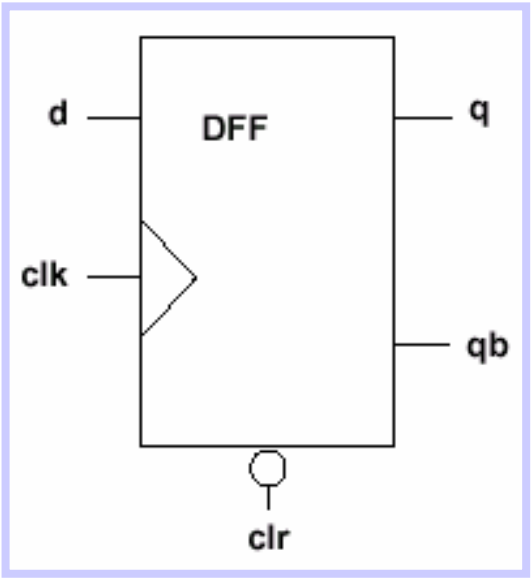
端口及内部信号
信号类型：
net
register
parameter

```
module DFF (q, qb, d, clk, clr);
 // 端口说明
   output q, qb;
   input d,  // input data
         clk, /*input clock */ clr;
   reg q;
   wire qb, d, clk, clr;
/*
  clk is posedge and clr is active low
*/
   assign qb = !q;
   always @(posedge clk or negedge clr)
      if(!clr)
            q <= 0;
      else
            q <= d;
endmodule
```

端口在模块名字后的括号中列出

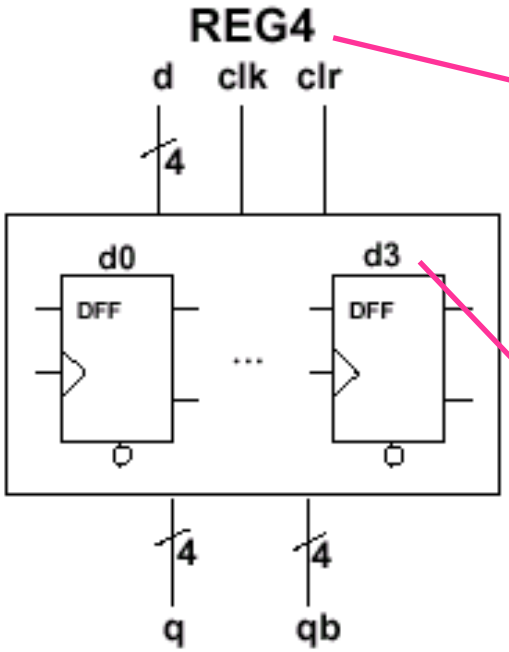
端口可以说明为*input*, *output*及*inout*

功能描述放在module内部



模块实例化(构成层次体系)

模块实例化(module instances)




```
module DFF (d, clk, clr, q, qb);
    ....
endmodule

module REG4(d, clk, clr, q, qb);
    output [3: 0] q, qb;
    input [3: 0] d;
    input clk, clr;
    DFF d0 (d[0], clk, clr, q[0], qb[0]);
    DFF d1 (d[1], clk, clr, q[1], qb[1]);
    DFF d2 (d[2], clk, clr, q[2], qb[2]);
    DFF d3 (d[3], clk, clr, q[3], qb[3]);
endmodule
```

空白符和注释

```
module DFF (q, qb, d, clk, clr);
 // 端口说明
   output q, qb;
   input d,  // input data
         clk, /*input clock */ clr;
   reg q;
   wire qb, d, clk, clr;
/*
  clk is posedge and
  clr is active low
*/
   assign qb = !q;
   always @(posedge clk or negedge clr)
      if(!clr)
            q <= 0;
      else
            q <= d;
endmodule
```

单行注释“//”到行末结束

多行注释，在/* */内

格式自由

使用空白符(空格、换行符)提高可读性及代码组织。Verilog忽略空白符除非用于分开其它的语言标记。

语言要素：标识符

- 所谓标识符就是用户为程序描述中的Verilog 对象所起的名字。
- 模块名、变量名、常量名、函数名、任务名
- 标识符必须以英语字母(a-z, A-Z)起头，或者用下横线符(_)起头。其中可以包含数字、$符和下划线符。
- 标识符最长可以达到1023个字符。
- 模块名、端口名和实例名都是标识符。
- Verilog语言大小写敏感
- sel 和 SEL 是两个不同的标识符。
- 所有的关键词都是小写的。

语言要素：系统任务和函数

- 以$字符开始的标识符表示系统任务或系统函数。
- 任务可以返回0个或多个值，函数除只能返回一个值以外与任务相同。
- 函数在0时刻执行，即不允许延迟，而任务可以带有延迟。
- 常用于测试模拟，一般不用于源代码设计。

$符号指示这是系统任务和函数；
系统函数有很多，如：

- 返回当前仿真时间$time
- 显示/监视信号值($display, $monitor)
- 停止仿真$stop
- 结束仿真$finish

```
$display ("Hi, you have reached LT today");
    /*$display系统任务在新的一行中显示。* /
$time  //该系统任务返回当前的模拟时间。
```

```
$monitor ($time, “a = %b, b = %h”, a, b);
```

语言要素：编译指令

- 以`(反引号)开始的某些标识符是编译器指令
  - ` define 和`undef，很像C语言中的宏定义指令
  - ` ifdef、`else 和`endif，用于条件编译
  - ` include 文件既可以用相对路径名定义，也可以绝对路径
  - ` timescale 编译器指令将时间单位与实际时间相关联。该指令用于定义时延的单位和时延精度。

文本替换(substitution) - `define

编译指令`define提供了一种简单的文本替换的功能

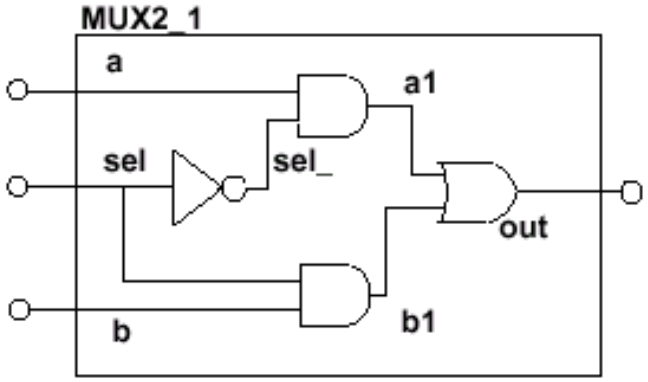
`define <macro_name> <macro_text>

在编译时<macro_text>替换<macro_name>，可提高描述的可读性

```
`define not_delay   #1
`define and_delay  #2
`define or_delay    #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay  not1( sel_, sel);
    and `and_delay and1( a1, a, sel_);
    and `and_delay and2( b1, b, sel);
    or   `or_delay    or1( out, a1, b1);
endmodule
```

定义not_delay

使用not_delay



文本包含(inclusion) - `include

- 编译指令`include在当前内容中插入一个文件

格式： `include “<file_name>”

如 `include "global.v"

`include "parts/count.v"

`include "../../library/mux.v”

可以是相对路径或绝对路径

- `include可用于：
  - include在文件中保存全局的或经常用到的一些定义，如文本宏
  - 在模块内部include一些任务(tasks)，提高代码的可维护性

时延：`timescale

- `timescale 说明时间单位及精度

  格式：

  `timescale <time_unit> / <time_precision>

  如：`timescale 1 ns / 100 ps

  time_unit: 延时或时间的测量单位

  time_precision: 延时值超出精度要先舍入后使用

- `timescale必须在模块之前出现

  // 所有时间都是时间单位的倍数

```
`timescale 1 ns / 10 ps
module DFF (q, qb, d, clk, clr);
 // 端口说明
   output q, qb;
   input d,  // input data
         clk, /*input clock */ clr;
   reg q;
   wire qb, d, clk, clr;
/*
  clk is posedge and
  clr is active low
*/
   assign #2 qb = !q;
   always @(posedge clk or negedge clr)
      if(!clr)
            q <= 0;
      else
            q <= d;
endmodule
```

时延：`timescale

- time_precision不能大于time_unit
- time_precision和time_unit的表示方法：integer unit_string
  - integer：可以是1、10、100
  - unit_string: 可以是s(second), ms(millisecond), us(microsecond), ns(nanosecond), ps(picosecond), fs(femtosecond)
  - 以上integer和unit_string可任意组合
- precision的时间单位应尽量与设计的实际精度相同
  - precision是仿真器的仿真时间步
  - 若time_unit与precision_unit差别很大将严重影响仿真速度
  - 如说明一个`timescale 1s / 1ps，则仿真器在1秒内要扫描其事件序列$10^{12}$次；而`timescale 1s/1ms则只需扫描10^{3}次
- 如果没有timescale说明将使用缺省值，一般是s

时延：`timescale

所有timescale中的最小值决定仿真时的最小时间单位
这是因为仿真器必须对整个设计进行精确仿真
在下面的例子中，仿真时间单位(STU)为100fs

```
`timescale 1ns/100fs
module1 (. . .);
        not #1.23 (. . .) // 1.23ns or 12300 STUs
        . . .
endmodule
`timescale 100ns/100fs
module2 (. . .);
        not #1.23 (. . .) // 123ns or 1230000 STUs
        . . .
endmodule
`timescale 1ps/100fs
module3 (. . .);
        not #1.23 (. . .) // 1.23ps or 12 STUs (rounded off)
        . . .
endmodule
```

模块与端口

- 模块：基本单元定义成模块形式

```
module module_name (port _list) ;
    Declarations_and_Statements
endmodule
```

端口队列port_list列出了该模块通过哪些端口与外部模块通信。

- 三种端口：
  - input (输入端口)
  - output (输出端口)
  - inout (双向端口)
- 三类(class)数据类型：
  - net(连线): 表示器件之间的物理连接
  - register(寄存器)：表示抽象存储元件
  - parameter(参数): 运行时的常数 (run-time constants)

物理意义和行为都有差别

```
module DFF (q, qb, d, clk, clr);
  // 端口说明
    output q, qb;
    input d,  // input data
          clk, /*input clock */ clr;
    reg q;
    wire qb, d, clk, clr;
/*
  clk is posedge and
  clr is active low
*/
    assign qb = !q;
    always @(posedge clk or negedge clr)
       if(!clr)
              q <= 0;
       else
              q <= d;
endmodule
```

模块与端口

端口

- 模块的端口可以是
  input(输入端口)、output (输出端口) 或者inout (双向端口)；
- 缺省的端口类型为wire型；
- output或inout能够被重新声明为reg型，但是input不可以；
- 线网或寄存器必须与端口说明中指定的长度相同。

例：

```
module Micro (PC, Instr, NextAddr);
//端口说明
        input [3:1] PC;
        output [1:8] Instr;
        inout [16:1] NextAddr;
        //重新说明端口类型：
        wire [16:1] NextAddr;
        //该说明是可选的，但如果指定了，就必须与它的端口说明保持相同长度。
        reg [1:8] Instr;
        //Instr已被重新说明为reg型，因此能在always语句或在initial语句中赋值。
...
endmodule
```

模块实例化

模块实例化语句

- 一个模块能够在另外一个模块中被引用，这样就建立了描述的层次。
  模块实例语句形式

  module_name instance_name (port _associations) ;

- 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。
  端口关联形式

  port_expr　　//通过位置，隐式关联

  .PortName (port_expr)　　//通过名称，显示关联，强烈推荐！

- **port_expr可以是以下的任何类型：**

1) 标识符（reg型或wire型）
2) 位选择
3) 部分选择
4) 上述类型的合并
5) 表达式(只适用于input型信号)

```
Micro M1 (UdIn[3:0], {WrN, RdN}, Status[0], Status[1] ,
                &UdOut[0:7], TxData ) ;
```

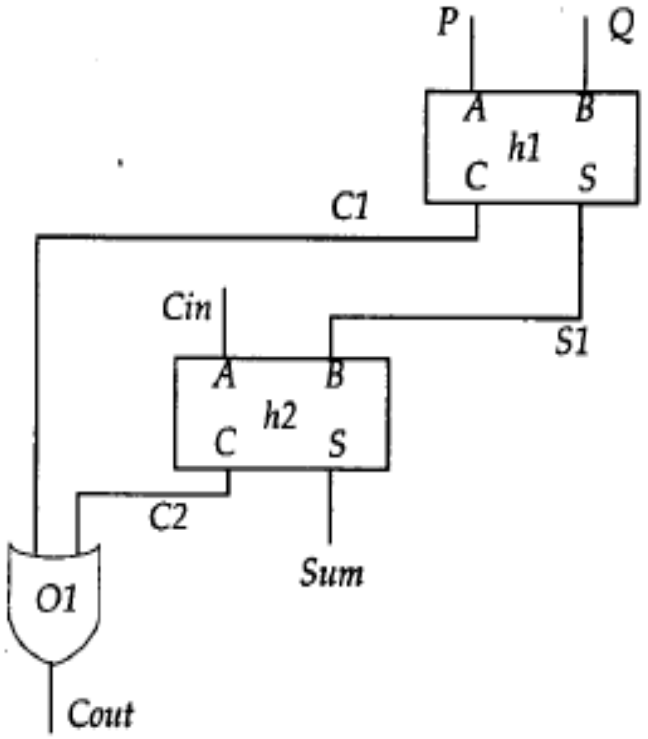
模块实例化：示例

- 使用两个半加器模块构造全加器

```
module HA (A , B , S , C);
      input A , B;
      output S, C;
      assign S = A ^ B;
      assign C = A & B;
endmodule

module FA (P, Q, Cin, Sum, Cout) ;
      input P, Q, Cin;
      output Sum, Cout;
      wire S1, C1, C2;
      HA h1 (P, Q, S1, C1);                    //通过位置关联。
      HA h2 (.A(Cin), .S(Sum), .B(S1), .C(C2));   //通过端口与信号的名字关联。
      or O1 (Cout, C1, C2) ;                   //或门实例语句
endmodule
```

考虑如何模块参数化？

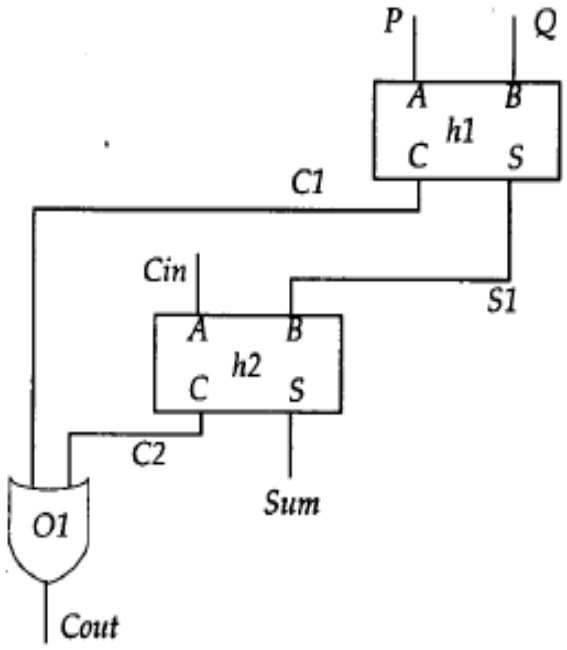


模块实例化：示例

使用两个半加器模块构造全加器(模块参数化)

```
module HA (A , B , S , C);
      input A , B;
      output S, C;
      parameter AND_DELAY = 1, XOR_DELAY = 2;
      assign #XOR_DELAY S = A ^ B;
      assign #AND_DELAY C = A & B;
endmodule

module FA (P, Q, Cin, Sum, Cout) ;
      input P, Q, Cin;
      output Sum, Cout;
      parameter OR_DELAY = 1;
      wire S1, C1, C2;
      HA (.AND_DELAY(3), .XOR_DELAY(4)) h1 (P, Q, S1, C1); //通过位置关联。
      HA #(2, 5) h2 (.A(Cin), .S(Sum), .B(S1), .C(C2));    //通过端口与信号的名字关
          联。
      or #OR_DELAY O1 (Cout, C1, C2) ;       //或门实例语句
endmodule
```



模块实例化：模块参数

- 模块参数值改变

1)参数定义语句(defparam)

```
module TOP (NewA , NewB , NewS , NewC) ;
    input New A , New B;
    output New S , New C;
    defparam Ha1. XOR_DELAY = 5,
    //实例Ha1中的参数XOR_DELAY。
    Ha1. AND_DELAY = 2;
    //实例Ha1中参数的AND_DELAY。
    HA Ha1 (NewA, NewB, NewS, NewC) ;
endmodule
```

模块实例化：模块参数

- 模块参数值改变
- 2) 带参数值的模块引用

```
module TOP (NewA , NewB , NewS , NewC) ;
    input New A , New B;
    output New S , New C;
    HA #(5,2) Ha1(NewA , NewB , NewS , NewC) ;
    //第1个值5赋给参数AND_DELAY，该参数在模块HA
      中说明。
    //第2个值2赋给参数XOR_DELAY，该参数在模块HA
      中说明。
endmodule
```

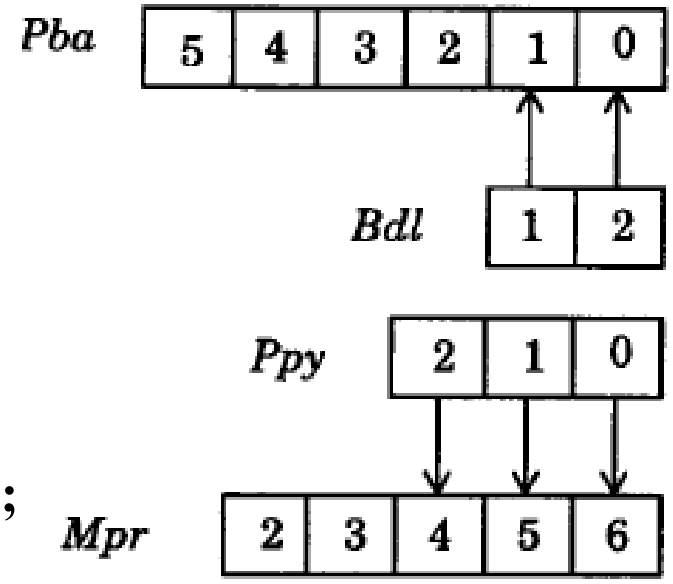
模块实例化

- 悬空端口
  - 通过将端口表达式表示为空白来指定为悬空端口

```
DFF d1(.Q(QS), .Qbar(), .Data(D),.Preset(), .Clock(CK));
```

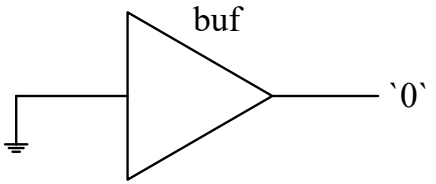
- 端口长度不同
  - 通过无符号数的右对齐或截断方式进行匹配

```
module Child(Pba, Ppy) ;
    input [5:0] Pba;
    output [2:0] Ppy;
    . . .
endmodule
module Top;
    wire [1:2] Bdl;
    wire [2:6] Mpr;
    Child C1 (.Pba(Bdl), .Ppy(Mpr));
endmodule
```

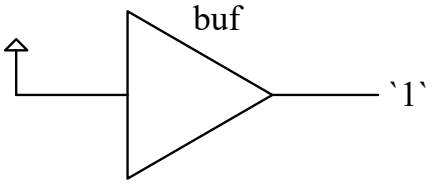


4值逻辑

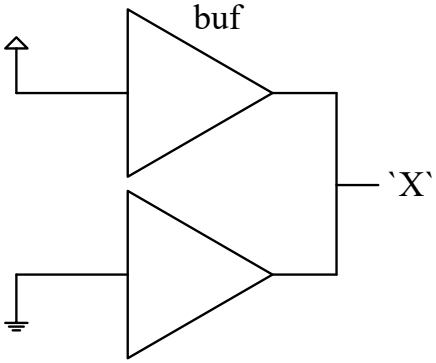
Verilog采用4值逻辑 0，1，X，Z



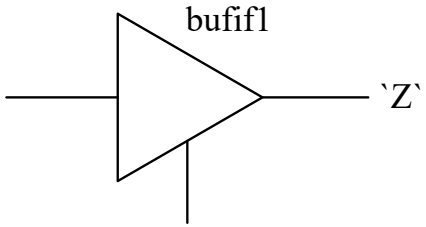
‘0’表示Zero，Low，Logic Low，False，Ground，VSS，Negative Assertion



‘1’表示One，High，Logic High，True，Power，VDD，VCC，Positive Assertion



‘X’表示信号状态时表示未知；当表示条件判断时(casex或casez中)表示不关心；



‘Z’表示高阻状态，也就是没有任何驱动；

net类的分类

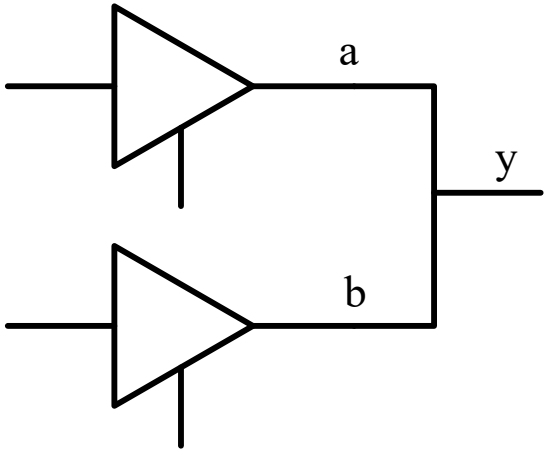
- 有多种net类型用于设计(design-specific)建模和工艺(technology-specific)建模

net类型	功 能
wire, tri	标准内部连接线(缺省)
supply1, supply0	电源和地
wor, trior	多驱动源线或
wand, triand	多驱动源线与
trireg	能保存电荷的net
tri1, tri0	无驱动时上拉/下拉

- 没有声明的net的缺省类型为 1 位(标量)wire类型，但这个缺省类型可由下面的编译指令改变：

  `default_nettype <nettype>

net数据类型中逻辑冲突的解决



wire/tri

a \ b	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand/triand

a \ b	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor/trior

a \ b	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

register类的类型

- 寄存器类有四种数据类型

寄存器类型	功能
reg	可定义的无符号整数变量，可以是标量(1位)或矢量，是最常用的寄存器类型
integer	32位有符号整数变量，算术操作产生二进制补码形式的结果。通常用作不会由硬件实现的的数据处理
real	双精度的带符号浮点变量，用法与integer相同
time	64位无符号整数变量，用于仿真时间的保存与处理
realtime	与real内容一致，但可以用作实数仿真时间的保存与处理

不要混淆寄存器数据类型与结构级存储元件，如udp_dff

Verilog中net和register声明语法

net声明

```
<net_type> [range] [delay] <net_name>[, net_name];
```

net_type： net类型

range: 矢量范围，以[MSB：LSB]格式

delay： 定义与net相关的延时(注意是固有延时)

net_name: net名称，一次可定义多个net，用逗号分开

寄存器声明

signed / unsigned

```
<reg_type> [range] <reg_name>[, reg_name];
```

reg_type： 寄存器类型

range： 矢量范围，以[MSB：LSB]格式，只对reg类型有效

reg_name ： 寄存器名称，一次可定义多个寄存器，用逗号分开

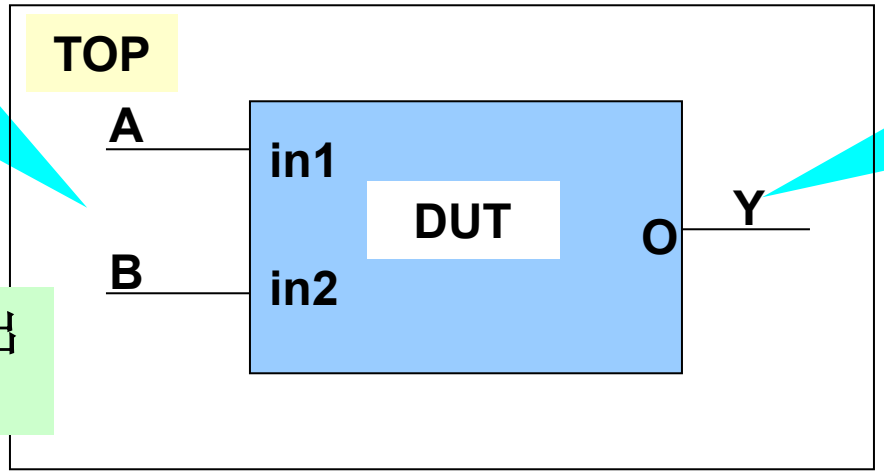
Verilog中net和register声明语法

- 举例：

```
reg a;                    //一个标量寄存器
wand w;                   // 一个标量wand类型net
reg [3 : 0] v;            // 从MSB到LSB的4位寄存器向量
reg [7 : 0] m, n;         // 两个8位寄存器
tri [15 : 0] busa;        // 16位三态总线
wire [0 : 31] w1, w2;     // 两个32位wire，MSB为bit0
```

选择正确的数据类型

输入端口可以由net/register驱动，但输入端口只能是net



输出端口可以是net/register类型，输出端口只能驱动net

双向端口输入/输出只能是net类型

```
module top;
wire Y;
reg A, B;
    DUT u1 (Y, A, B) ;
    initial begin
            A = 0; B = 0;
        #5 A = 1;
     end
endmodule
```

在过程块中只能给register类型赋值

```
module DUT (O, in1, in2);
output O;
input in1, in2;
wire O, in1, in2;
    and (O, in1, in2);
endmodule
```

若O，in1，in2说明为reg则会产生错误

Register数组(Register Arrays)

寄存器数组声明语法：

reg [MSB:LSB] <memory_name> [first_addr:last_addr];

- 声明一个寄存器数组：

```
integer  NUMS  [7:0];  //8个整数变量的数组
time  t_vals  [3:0];  //4个time变量的数组
```

- 数据类型为reg的数组通常称为一个memory:

```
reg [15:0] MEM [0:1023];  //1K x 16-bit memory array
reg [7:0] PREP ['hFFFE:'hFFFF];  //2 x 8-bit memory array
```

- 可以使用parameters建模memory size:

```
parameter wordsize = 16;
parameter memsize = 1024;
reg [wordsize-1:0] MEM3 [memsize-1:0];
```

存储器寻址(Memory Addressing)

存储器可以通过索引寻址一个存储器数组

```
module mems;
    reg [8:1] mema [0:255]; //声明一个叫做mema的存储器
    reg [8:1] mem_word;     //叫做mem_word的内部寄存器
    . . .

    initial
      begin
        // 显示第6个存储器地址的内容
       $displayb(mema[5]);
        // 显示第6个存储器的最高位
       mem_word = mema[5];
       $displayb(mem_word[8]);
      end
endmodule
```

功能描述

持续赋值
- *assign*

过程块
- initial
- always

```
module DFF (q, qb, d, clk, clr);
    output q, qb;
    input d,  // input data
          clk, /*input clock */ clr;
    reg q;
    wire qb, d, clk, clr;

    assign qb = !q;

    always @(posedge clk or negedge clr)
       if(!clr)
            q <= 0;
       else
            q <= d;

endmodule
```

持续赋值(continuous assignment)

- 描述的是组合逻辑
- 在过程块外部使用
- 用于net驱动
- 在等式左边可以有一个简单延时说明
  - 只限于在表达式左边用#delay形式
- 可以是显式或隐含

语法：

<assign>[#delay][strength]<net_name>=<expressions>；

```
wire out;
assign out = a & b; // 显式
wire inv = ~in; // 隐含
```

持续赋值(continuous assignment)

- 连续赋值语句在什么时候执行呢？

  只要在右端表达式的操作数上有事件发生(值变化)，表达式立即被计算，新结果就赋给左边的线网。

- 连续赋值的目标类型(左侧操作数类型)

  - 1) 标量线网　　assign Z1 = … ;
  - 2) 向量线网　　assign data_tmp = … ;
  - 3) 向量的常数型位选择　　assign data_tmp[2] = … ;
  - 4) 向量的常数型部分选择　　assign data_tmp[7:0] = … ;
  - 5) 上述类型的任意的拼接运算结果

    assign {Z1, data_tmp[15]} = 2'b10;

持续赋值： 数据流建模

- 例：数据流描述的一位全加器

```
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
```

1）assign语句之间是并发的，与其书写的顺序无关；

2）线网的赋值可以在声明时赋值，例如

```
    wire Sum = A^B ^Cin;
```

持续赋值： 数据流建模

- 数据流建模的时延

```
assign  #2  Sum = A ^ B ^ Cin;
```

- #2表示右侧表达式的值延迟两个时间单位赋给Sum；
- 时间单位是多少？由谁来决定？

```
`timescale 1ns/100ps
```

- FPGA设计中的时延仅在功能仿真时有效，不影响实际电路生成。

持续赋值： 数据流建模

- 数据流建模注意事项：
  - 1）wire型变量如果不赋值，默认值为z；
  - 2）数据流建模没有存储功能，不能保存数据；
  - 3）wire型变量只能在声明时赋值或者assign语句赋值；
  - 4）assign语句并发执行，实际的延迟由物理芯片的布线结果决定。
  - 5）最基本的FPGA设计源代码描述语句之一，用于生成组合逻辑，定制LUT的逻辑功能。常作为中间信号的描述用于控制寄存器的输入输出。

块语句

- 块语句用来将多个语句组织在一起，使得他们在语法上如同一个语句
- 块语句分为两类：
  - 顺序块：语句置于关键字begin和end之间，块中的语句以顺序方式执行
  - 并行块：关键字fork和join之间的是并行块语句，块中的语句并行执行

always	c
begin	
c	———
c	———
c	———
c	———
c	———
c	———
c	———
end	

always	c
fork	
c	———
c	———
c	———
c	———
c	———
c	———
c	———
join	

initial	c
begin	
c	———
c	———
c	———
c	———
c	———
c	———
c	———
end	

initial	c
fork	
c	———
c	———
c	———
c	———
c	———
c	———
c	———
join	

fork和join语句常用于test bench描述，这是因为可以一起给出矢量及其绝对时间，而不必描述所有先前事件的时间

块语句(续)

顺序语句将被顺序执行，也就是逐条执行；

并行语句在同一时间步内被调度，但经过相关延迟后被执行；

```
begin
    #5 a = 3;
    #5 a = 5;
    #5 a = 4;
end
```

```
fork
    #5   a = 3;
    #15 a = 4;
    #10 a = 5;
join
```

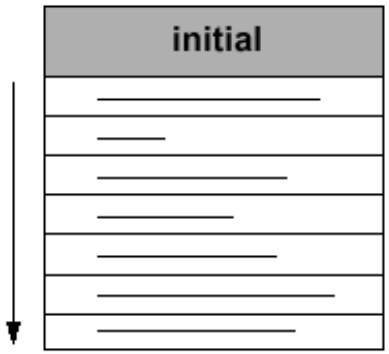
上面2个例子在功能上是等价的

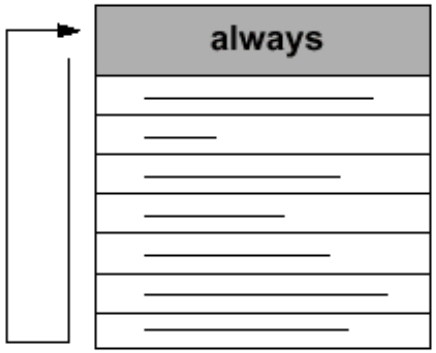
但是fork/join模块综合器不支持！！

过程块(procedural block)

过程语句有两种：

- *initial* ： 只执行一次
- *always* ： 循环执行





所有过程在时间0执行一次

过程块之间

assign语句之间

过程块与assign语句

} 并行执行

```
module DFF (q, qb, d, clk, clr);
   output q, qb;
   input d,  // input data
         clk, /*input clock */ clr;
   reg q;
   wire qb, d, clk, clr;

   assign qb = !q;

   always (时序控制)
   begin/fork
      过程赋值语句;
      系统任务和函数;
      高级描述语句;
      {if语句;
       case语句;
       循环语句;
   end/join
endmodule
```

行为建模： initial语句

- initial 语句只执行一次;
- 在模拟开始时执行，即在0时刻开始执行;
- 不能嵌套使用。

initial [timing_control] procedural_statement

procedural_statement可以是：

procedural_continuous_assignment		过程赋值（阻塞或者非阻塞）
conditional_statement	->	if
case_statement	->	case
loop_statement	->	for, forever, repeat, while
wait_statement	->	wait
disable_statement	->	disable（相当于C中的break）
event_trigger	->	@ (event)
sequential_block	->	begin ... end
parallel_block	->	fork ... join
task_enable (user or system)		

行为建模： initial语句

- 例：

```
reg Curt;
. . .
initial #2 Curt = 1;
```

- 例：

```
parameter SIZE = 1024;
reg [7:0] RAM [0 : SIZE-1] ;
reg RibReg;

initial
begin: SEQ_BLK_A        //顺序过程的标记，如果没有局部声明，则不需要
      integer Index;
      RibReg = 0;
      for (Index = 0; Index < SIZE; Index = Index + 1)
         RAM [Index] = 0;
end
```

行为建模： initial语句

- initial语句在仿真文件产生时钟和构造数据简单示例

```
parameter APPLY_DELAY = 5;
reg [0 : 7] port_A;
reg clk;
. . .
initial
begin
      Port_A = 'h20 ;
      #APPLY_DELAY Port_A= 'hF2;
      #APPLY_DELAY Port_A= 'h41;
      #APPLY_DELAY Port_A= 'h0A;
end
initial
begin
     clk = 0;
     while(1)                          //或者 forever
         clk = #5 ~clk;                //或者 #5 clk = ~clk;
end
```

行为建模： always语句

- always语句重复执行，语法和initial语句相同：

  always [timing_control] procedural_statement

  procedural_statement可以是：

  procedural_continuous_assignment 过程赋值（阻塞或者非阻塞）

  conditional_statement -> if

  case_statement -> case

  loop_statement -> for , forever, repeat, while

  wait_statement -> wait

  disable_statement -> disable（相当于C中的break）

  event_trigger -> @ (event)

  sequential_block -> begin ... end

  parallel_block -> fork ... join

  task_enable (user or system)

行为建模： always语句

- 两种典型的always语句
  - 1) 组合逻辑(电平触发)

```
reg c;
always @ ( a or b or sel )
   c = sel ? a : b;
```

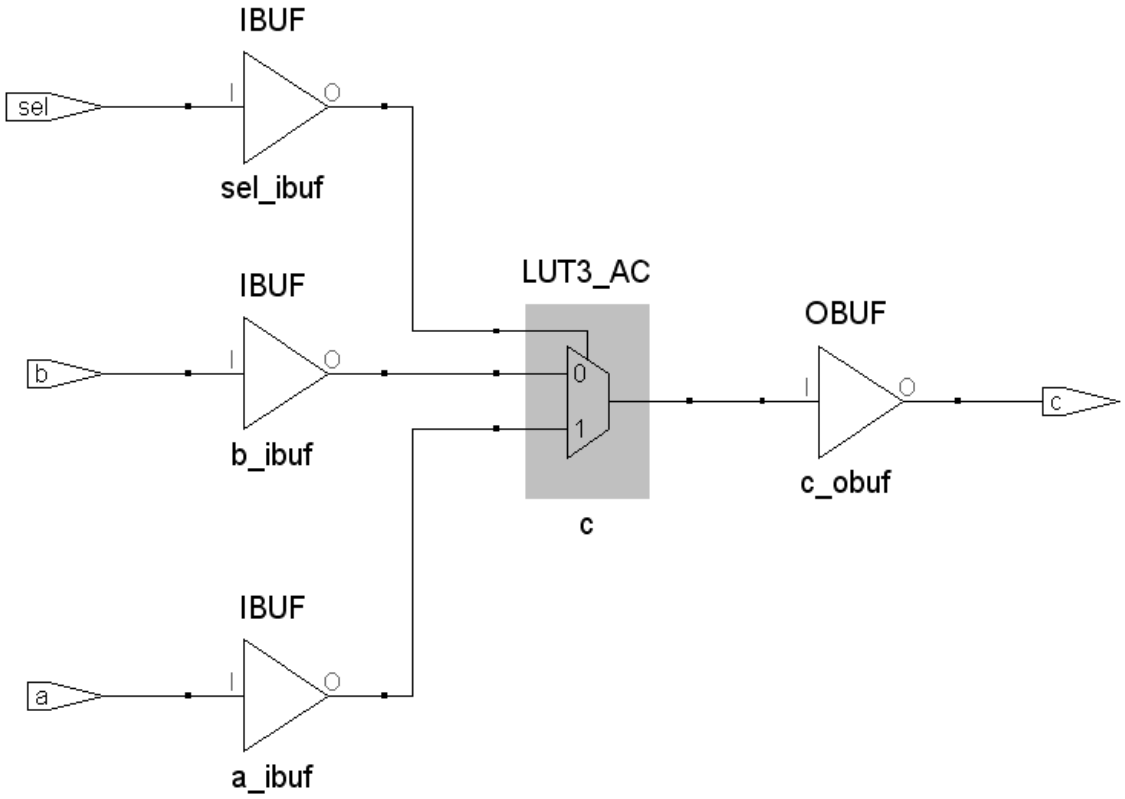
说明:

(1)虽然c是reg型，但综合的结果是组合电路;

(2)等同于数据流描述

```
wire  c;
assign c = sel ? a : b;
```

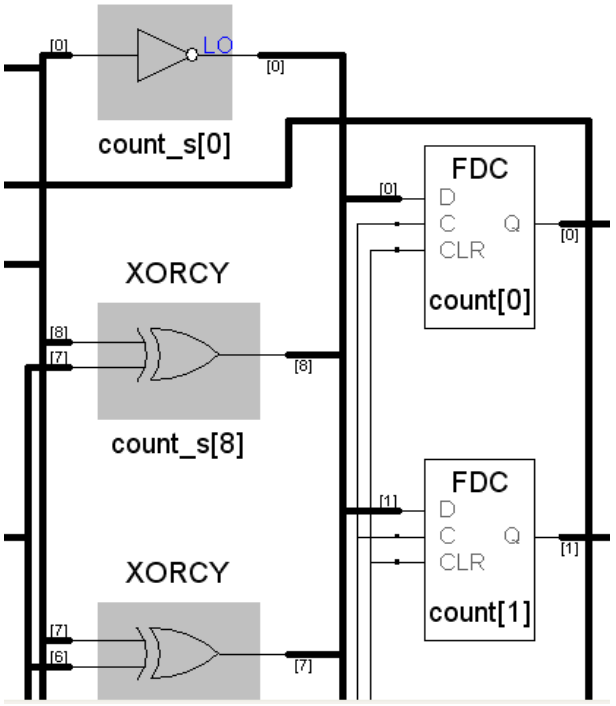
(3) FPGA设计中不建议使用此外，容易产生锁存器



行为建模： always语句

- 两种典型的always语句
  - 2) 时序逻辑（时钟沿触发）

```
reg [8:0]count;
always @ (posedge clk or  negedge
    reset)
begin
    if (~reset)
          count <= 0;
    else
        begin
         if (count == 511)
             count <= 0;
         else
             count <= count + 1;
        end
end
```



- 说明：

(1) 在always语句中所有被赋值的信号必须是reg型；

(2) 综合为触发器，推荐使用；

(3) 异步时序逻辑

过程赋值(procedural assignment)

- 在过程块中的赋值称为过程赋值，分为阻塞过程赋值和非阻塞过程赋值两种
- 表达式左边的信号必须是寄存器类型(如reg类型)
- 等式右边可以是任何有效的表达式，数据类型也没有限制
- 如果信号没有声明则缺省为wire类型，使用过程赋值语句给wire赋值会产生错误

```
module adder (out, a, b, cin);
    input a, b, cin;
    output [1:0] out;
    reg half_sum;
    reg [1: 0] out;
  always @( a or b or cin)
  begin
    half_sum = a ^ b ^ cin ; // OK
    half_carry = a & b | a & !b & cin | !a & b & cin ; // ERROR!
    out = {half_carry, half_ sum} ;
  end
endmodule
```

half_carry
没有声明

非阻塞过程赋值

```
module swap_vals;
  reg a, b, clk;
  initial begin
    a = 0; b = 1; clk = 0;
  end
  always #5 clk = ~clk;
  always @( posedge clk)
    begin
      a <= b; // 非阻塞过程赋值
      b <= a; // 交换a和b值
    end
endmodule
```

过程赋值有两类

阻塞过程赋值

非阻塞过程赋值

阻塞赋值执行完成后再执行顺序块内的下一条语句；

非阻塞赋值不阻塞过程流，仿真器读入一条赋值语句并对它进行调度之后，就可以处理下一条赋值语句；

若过程块中的所有赋值都是非阻塞的，赋值按两步进行：

1. 仿真器计算所有RHS表达式的值，保存结果，并进行调度，在时序控制指定的时间赋值；
2. 在经过相应的延迟后，仿真器通过将保存的值赋给LHS表达式，完成赋值；

非阻塞过程赋值(续)

阻塞与非阻塞赋值语句行为差别举例1

```
module non_block1;
  reg a, b, c, d, e, f;
  initial begin // 阻塞赋值
    a = #10 1; // time 10
    b = #2  0; // time 12
    c = #4  1; // time 16
  end
  initial begin // 非阻塞赋值
    d <= #10 1; // time 10
    e <= #2  0; // time 2
    f <= #4  1; // time 4
  end
  initial begin
    $monitor( $time,," a= %b b= %b c= %b d= %b e= %b f= %b", a, b, c, d, e, f);
    #100 $finish;
  end
endmodule
```

输出结果：

```
 0   a= x b= x c= x d= x e= x f = x
 2   a= x b= x c= x d= x e= 0 f = x
 4   a= x b= x c= x d= x e= 0 f = 1
10   a= 1 b= x c= x d= 1 e= 0 f = 1
12   a= 1 b= 0 c= x d= 1 e= 0 f = 1
16   a= 1 b= 0 c= 1 d= 1 e= 0 f = 1
```

过程语句：条件语句 (if分支语句)

if 和 *if-else* 语句：

```
always #20
    if (index > 0) // 开始外层 if
        if (rega > regb) // 开始内层第一层 if
                result = rega;
        else
                result = 0; // 结束内层第一层 if
    else
        if (index == 0)
              begin
                $display(" Note : Index is zero");
                result = regb;
              end
        else
                $display(" Note : Index is negative");
```

描述方式：

```
if (表达式）
    begin
        ……
    end
else
    begin
        ……
    end
```

1. 条件语句必须在过程块语句中使用，不能单独使用；
2. 可以多层嵌套，在嵌套 *if* 序列中，*else*和前面最近的*if* 相关；
3. 为提高可读性及确保正确关联，使用**begin...end**块语句指定其作用域；

过程语句： 条件语句(case分支语句)

case 语句：

```
module compute (result, rega, regb, opcode);
input [7: 0] rega, regb;
input [2: 0] opcode;
output [7: 0] result;
reg [7: 0] result;
always @( rega or regb or opcode)
    case (opcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3‘b010 , // 多个case有同一个结果
        3'b100 : result = rega / regb;
        default : begin
                      result = 'bx;
                      $display ("no match");
                  end
    endcase
endmodule
```

过程语句： 条件语句(case分支语句)

case语句是测试表达式与另外一系列表达式分支是否匹配的多路条件语句

- case语句进行逐位比较以求完全匹配(包括x和z)；
- default语句可选，在没有任何条件成立时执行，此时如果未说明default，Verilog不执行任何动作；
- 多个default语句是非法的；

重要内容：

使用**default**语句是一个很好的编程习惯，特别是用于检测**x**和**z**；

casez和casex为case语句的变体，允许比较无关(don't-care)值：

- case表达式的任何位为无关值时，在比较过程中该位不予考虑；
- 在casez语句中，**?** 和 **z** 被当作无关值；
- 在casex语句中，**?**、**z** 和 **x** 被当作无关值；

case语法：

```
case <表达式>
        <表达式>, <表达式>：赋值语句或空语句；
        <表达式>, <表达式>：赋值语句或空语句；
                        default：赋值语句或空语句；
endcase
```

过程语句：循环语句

有四种循环语句：

repeat：将一块语句循环执行确定次数

repeat (次数表达式) <语句>

while：在条件表达式为真时一直循环执行

while（条件表达式）〈语句〉

forever：重复执行直到仿真结束

forever〈语句〉

for：在执行过程中对变量进行计算和判断，在条件满足时执行

for（赋初值；条件表达式；计算）〈语句〉

综合工具还不支持

循环(looping)语句-repeat

repeat：将一块语句循环执行确定次数

```
module multiplier( result, op_a, op_b);
    parameter size = 8;
    input [size:1] op_a, op_b;
    output [2* size:1] result;
    reg [2*size:1] shift_opa, result;
    reg [size:1] shift_opb;
    always @( op_a or op_b) begin
        result = 0;
        shift_opa = op_a; // 零扩展至16位
        shift_opb = op_b;
        repeat (size) begin
            #10 if (shift_opb[1]) result = result + shift_opa;
            shift_opa = shift_opa << 1; // Shift left
            shift_opb = shift_opb >> 1; // Shift right
        end
    end
endmodule
```

为什么要说明一个shift_opb变量？

循环(looping)语句-while

while：只要表达式为真(不为0)，则重复执行一条语句(或语句块)

```
. . .
reg [7: 0] tempreg;
reg [3: 0] count;
. . .
    count = 0;
    while (tempreg) // 统计tempreg中 1 的个数
    begin
        if (tempreg[0]) count = count + 1;
        tempreg = tempreg >> 1; // 右移
    end
end
. . .
```

循环(looping)语句-forever

forever：一直执行到仿真结束

forever应该是过程块中最后一条语句，其后的语句将永远不会执行。

forever语句不可综合，通常用于test bench描述。

```
...
reg clk;
initial
    begin
    clk = 0;
    forever
        begin
            #10 clk = 1;
            #10 clk = 0;
    end
end
...
```

这种行为描述方式可以非常灵活的描述时钟，可以控制时钟的开始时间及周期占空比。仿真效率也高。

循环(looping)语句-for

for：只要条件为真就一直执行

条件表达式若是简单的与0比较通常处理得更快一些，但综合工具可能不支持与0的比较。

```
// X检测
for (index = 0; index < size; index = index + 1)
      if (val[index] === 1'bx)
            $display (" found an X");

// 存储器初始化; “!= 0”仿真效率高
for (i = size; i != 0; i = i - 1)
      memory[i-1] = 0;

// 阶乘序列
factorial = 1;
for (j = num; j != 0; j = j - 1)
      factorial = factorial * j;
```

常量

1. 整数

- 表达方式：
  - <位宽>’<进制><数字>：标准方式
  - ’<进制><数字>：默认位宽，与机器类型有关
  - <数字>：不指明进制默认为十进制
- 进制
  - 二进制(b或B) : 8’b10101100， ’b1010
  - 十进制(d或D) : 4’d1543，512
  - 十六进制(h或H) : 8’ha2
  - 八进制(o或O) : 6’o41
- x和z值
  - x：不确定：4’b100x
  - z：高阻：16’hzzzz，没有驱动元件连接到线网，线网的缺省值为z。

常量

- 负数：
  - 在位宽表达式前加一个减号，如 -8’d5
  - 减号不可以放在位宽和进制之间，也不可以放在进制和具体的数之间，如8’d-5
- 下划线：
  - 只能用在具体的数字之间，如16’b1010_1111_1010_1101
  - 位数指的是二进制位数。
- 数位扩展：(定义的长度比为常量指定的长度长)
  - 最高位是0、1，高位用0扩展：8’b1111 等于 8’b00001111
  - 最高位是z、x，高位自动扩展：4’bz 等于 4’bzzzz
- 数位截断：
  - 如果长度定义得更小，最左边的位被截断，如：
  - 3’b1001_0011 等于 3’b011，5'H0FFF等于 5'H1F

常量

- 2. 实数
  - 十进制计数法；例如

    2.0

    5.68
  - 科学计数法；

    23_5.1e2　其值为23510.0，忽略下划线

    3.6E2　其值为360.0 (e与E相同)

实数通常不用于FPGA源代码的常量

常量

- 3. 字符串
  - 字符串是双引号内的字符序列。字符串不能分成多行书写。例如:

    "INTERNAL ERROR"

    "REACHED—>HERE"
  - 用8位ASCII值表示的字符可看作是无符号整数。

    为存储字符串“INTERNAL ERROR”，变量需要8 *14位。

    reg [1 : 8*14] *Message*;

    (*Message* = “INTERNAL ERROR”)

字符串较少用于FPGA源代码的常量

表达式

- 表达式由操作数和操作符组成；
- 表达式可以在出现数值的任何地方使用；
- 表达式是数据流描述的基础。

  - A & B
  - Addr1[3:0] +Addr2[3:0]
  - Count + 1
  - (a[0] ^ b[0]) | (a[1] & ~b[1])

表达式：操作数

- 操作数可以是以下类型中的一种：
  - 常数
  - 参数
  - 线网
  - 寄存器
  - 位选择
  - 部分选择
  - 存储器单元
  - 函数调用

表达式：操作数

常数

表达式中的整数值可被解释为有符号数或无符号数；
如果整数是基数型整数，作为无符号数对待。

- ✓ 12　　01100的5位向量形式　　(有符号)
- ✓ -12　　10100的5位向量形式　　(有符号)
- ✓ 5‘b01100　十进制数12　　(无符号)

参数

参数类似于常量，并且使用参数声明进行说明。例如

parameter *LOAD* = 4'd12, *STORE* = 4'd10;

***LOAD*和*STORE*为参数，值分别被声明为12和10。**

表达式：操作数

线网

线网中的值被解释为无符号数，

表达式中可使用：标量线网(1位)和向量线网(多位)。

- ✓ **wire [3:0] led;** //4位向量线网。
- ✓ **wire line;** //标量线网。
- ✓ **assign led = 4‘ha;** //被赋于位向量1010，为十进制10。

表达式：操作数

- 寄存器

  integer型的值被解释为有符号的二进制补码数，
  reg型或time型的值被解释为无符号数，
  real型和realtime的值被解释为有符号浮点数。

  - ✓ reg [4:0] state;
  - ✓ state = 5'b01011;　　// 值为位向量01011，十进制值11。
  - ✓ state = 9;　　// 值为位向量01001，十进制值9。

表达式：操作数

位选择

位选择从向量中抽取特定的位。形式如下：

net_or_reg_vector[bit_select_expr]

- ✓ **state[1] && state[4]** //寄存器位选择。
- ✓ **led[0] | line** //线网位选择。

如果选择表达式的值为**x**、**z**或越界，则位选择的值为**state[x]**值为**x**。(**FPGA设计中禁用**)

表达式：操作数

- 部分选择

net_or_reg_vector[msb_const_expr:lsb_const_expr]

state [4:1] //寄存器部分选择。　　reg [4:0] state;

led [2:0]　 //线网部分选择。　　wire [3:0] led;

选择范围越界或为x、z时，部分选择的值为x。

(FPGA设计中禁用越界)

表达式：操作数

存储器单元

存储器单元从存储器中选择一个memory[word_address]

```
reg [7 : 0] Dram [63 : 0];
Dram [60];          //存储器的第61个单元。
```

不允许对存储器变量值部分选择或位选择。

函数调用

表达式中可使用函数调用。

```
$time + SumOfEvents (A, B)
```

/* $time是系统函数，并且SumOfEvents是在别处定义的用户自定义函数。*/

表达式：操作符

- Verilog HDL中的操作符可以分为下述类型：
  - 算术操作符
  - 关系操作符
  - 相等操作符
  - 逻辑操作符
  - 按位操作符
  - 归约操作符
  - 移位操作符
  - 条件操作符
  - 连接和复制操作符

算术操作符

+	加
-	减
*	乘
/	除
%	模

- 将负数赋值给reg或其它无符号变量时，使用2的补码表示
- 如果操作数的某一位是x或z，则结果为x
- 在整数除法中，余数舍弃
- 模运算中使用第一个操作数的符号

```
module arithops ();
    parameter five = 5;
    integer ans, int;
    reg [3: 0] rega, regb;
    reg [3: 0] num;
  initial begin
    rega = 3;
    regb = 4'b1010;
    int = -3;     //int = 1111……1111_1101
  end
  initial fork
    #10 ans = five * int;        // ans = -15
    #20 ans = (int + 5) / 2;     // ans = 1
    #30 ans = five / int;        // ans = -1
    #40 num = rega + regb;  // num = 1101
    #50 num = rega + 1;          // num = 0100
    #60 num = int;               // num = 1101
    #70 num = regb % rega; // num = 1
    #80 $finish;
  join
endmodule
```

注意：integer和reg类型在算术运算时的差别

integer是有符号数，而reg是无符号数

关系操作符

>	大于
<	小于
>=	大于等于
<=	小于等于

其结果是

1’b1

1’b0

或 1’bx。

什么时候出现 x 值?

```
module relationals ();
    reg [3: 0] rega, regb, regc;
    reg val;
  initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b0x10;
  end
  initial fork
    #10 val = regc > rega ;   // val = x
    #20 val = regb < rega ;   // val = 0
    #30 val = regb >= rega ; // val = 1
    #40 val = regb > regc ;   // val = 1
    #50 $finish;
  join
endmodule
```

rega和regc的关系取决于x

无论x为何值，regb>regc

相等操作符

==	逻辑等
!=	逻辑不等

- 其结果是1'b1、1'b0或1'bx
- 如果左边及右边为确定值并且相等，则结果为1
- 如果左边及右边为确定值并且不相等，则结果为0
- 如果左边及右边有值不能确定，但值确定的位相等，则结果为x
- !=的结果与= =相反

值确定是指所有的位为0或1

不确定值是有值为x或z的位

```
module equalities1();
    reg [3: 0] rega, regb, regc;
    reg val;
  initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b1x10;
  end
  initial fork
    #10 val = rega == regb ;  // val = 0
    #20 val = rega != regc;    // val = 1
    #30 val = regb != regc;    // val = x
    #40 val = regc == regc;   // val = x
    #50 $finish;
  join
endmodule
```

相同操作符

=== 相同(case等)
! == 不相同(case不等)

- 其结果是1’b1、1’b0或1’bx
- 如果左边及右边的值相同(包括x、z)，则结果为1
- 如果左边及右边的值不相同，则结果为0
- !==的结果与 === 相反

综合工具不支持

```
module equalities2();
    reg [3: 0] rega, regb, regc;
    reg val;
  initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b1x10;
  end
  initial fork
    #10 val = rega === regb ; // val = 0
    #20 val = rega !== regc;   // val = 1
    #30 val = regb === regc;  // val = 0
    #40 val = regc === regc;  // val = 1
    #50 $finish;
  join
endmodule
```

相等操作符

= 赋值操作符，将等式右边表达式的值拷贝到左边

注意逻辑等与case等的差别

== 逻辑等

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
a = 2'b1x;
b = 2'b1x;
if (a == b)
    $display(" a is equal to b");
else
    $display(" a is not equal to b");
```

2’b1x==2’b0x

值为0，因为不相等

2’b1x==2’b1x

值为x，因为可能不相等，也可能相等

=== case等

==	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 2'b1x;
b = 2'b1x;
if (a === b)
    $display(" a is identical to b");
else
    $display(" a is not identical to b");
```

2’b1x===2’b0x

值为0，因为不相同

2’b1x==2’b1x

值为1，因为相同

逻辑操作符

!	not
&&	and
||	or

- 逻辑操作符的结果为一位1，0或x
- 逻辑操作符只对逻辑值运算
- 如操作数为全0，则其逻辑值为false
- 如操作数有一位为1，则其逻辑值为true
- 若操作数只包含0、x、z，则逻辑值为x

逻辑反操作符将操作数的逻辑值取反。例如，若操作数为全0，则其逻辑值为0，逻辑反操作值为1。

```
module logical ();
    parameter five = 5;
    reg ans;
    reg [3: 0] rega, regb, regc;
  initial
    begin
      rega = 4'b0011;     //逻辑值为“1”
      regb = 4'b10xz;     //逻辑值为“1”
      regc = 4'b0z0x;     //逻辑值为“x”
  end
  initial fork
    #10 ans = rega && 0;        // ans = 0
    #20 ans = rega || 0;        // ans = 1
    #30 ans = rega && five;     // ans = 1
    #40 ans = regb && rega;     // ans = 1
    #50 ans = regc || 0;        // ans = x
    #60 $finish;
  join
endmodule
```

按位操作符

~	not
&	and
|	or
^	xor
~ ^	xnor
^ ~	xnor

按位操作符对矢量中相对应位运算

regb = 4’b1010;

regc = 4’b1x10;

num = regb & regc = 4’b1010;

位值为x时不一定产生x结果，如#50时的or计算

当两个操作数位数不同时，位数少的操作数零扩展到相同位数

```
a = 4'b1011;
b = 8'b01010011;
c = a | b; // a零扩展为 8'b0000_1011
```

```
module bitwise ();
    reg [3: 0] rega, regb, regc;
    reg [3: 0] num;
  initial begin
    rega = 4'b1001;
    regb = 4'b1010;
    regc = 4'b11x0;
  end
  initial fork
    #10 num = rega & 0;       // num = 0000
    #20 num = rega & regb; // num = 1000
    #30 num = rega | regb;    // num = 1011
    #40 num = regb & regc; // num = 10x0
    #50 num = regb | regc;    // num = 1110
    #60 $finish;
  join
endmodule
```

逻辑反与位反的对比

!	logical not	逻辑反
~	bit-wise not	位反

- 逻辑反的结果为一位1，0或x
- 位反的结果与操作数的位数相同

逻辑反操作符将操作数的逻辑值取反。例如，若操作数为全0，则其逻辑值为0，逻辑反操作值为1。

```
module negation();
    reg [3: 0] rega, regb;
    reg [3: 0] bit;
    reg log;

  initial begin
    rega = 4'b1011;
    regb = 4'b0000;
  end

  initial fork
    #10 bit = ~rega;    // num = 0100
    #20 bit = ~regb;    // num = 1111
    #30 log = !rega;    // num = 0
    #40 log = !regb;    // num = 1
    #50 $finish;
  join
endmodule
```

一元归约操作符

&	and
|	or
^	xor
~ ^	xnor
^ ~	xnor

- 归约操作符的操作数只有一个
- 对操作数的所有位进行位操作
- 结果只有一位，可以是0, 1, X

```
module reduction();
    reg val;
    reg [3: 0] rega, regb;
    initial begin
        rega = 4'b0100;
        regb = 4'b1111;
    end
    initial fork
        #10 val = & rega ;    // val = 0
        #20 val = | rega ;    // val = 1
        #30 val = & regb ;    // val = 1
        #40 val = | regb ;    // val = 1
        #50 val = ^ rega ;    // val = 1
        #60 val = ^ regb ;    // val = 0
        #70 val = ~| rega;    // (nor) val = 0
        #80 val = ~& rega;    // (nand) val = 1
        #90 val = ^rega && &regb; // val = 1
      $finish;
    join
Endmodule.
```

移位操作符

>>	逻辑右移
<<	逻辑左移

- 移位操作符对其左边的操作数进行向左或向右的位移操作
- 第二个操作数(移位位数)是无符号数
- 若第二个操作数是x或z则结果为x

<< 将左边的操作数左移右边操作数指定的位数

>> 将左边的操作数右移右边操作数指定的位数

在赋值语句中，如果右边(RHS)的结果:

位宽大于左边，则把最高位截去

位宽小于左边，则零扩展

```
module shift ();
    reg [9: 0] num, num1;
    reg [7: 0] rega, regb;
  initial     rega = 8'b0000_1100;
  initial fork
    #10 num <= rega << 5 ;  // num = 01_1000_0000
    #10 regb  <= rega << 5 ;  // regb =     1000_0000
    #20 num <= rega >> 3;   // num = 00_0000_0001
    #20 regb  <= rega >> 3 ;  // regb =      0000_0001
    #30 num <= 10'b11_1111_0000;
    #40 rega <= num << 2;   //rega =       1100_0000
    #40  num1 <= num << 2;//num1=11_1100_0000
    #50 rega <= num >> 2;   //rega =       1111_1100
    #50  num1 <= num >> 2;//num1=00_1111_1100
    #60 $finish;
  join
endmodule
```

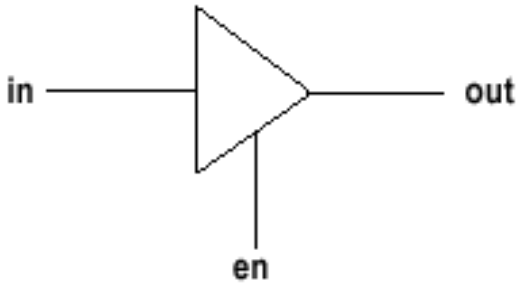
先补后移

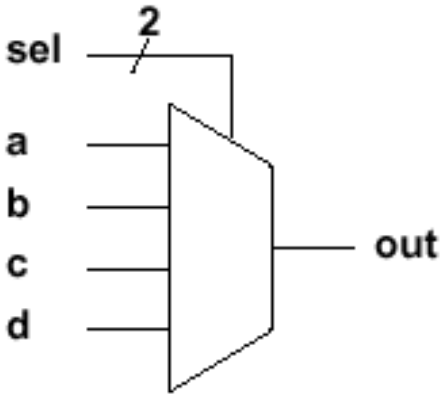
先移后截

建议：表达式左右位数一致

条件操作符

? : 条件





如果条件值为x或z，

则结果可能为x或z

```
module likebufif( in, en, out);
      input in;
      input en;
      output out;
   assign out = (en == 1) ? in : 'bz;
endmodule

module like4to1( a, b, c, d, sel, out);
      input a, b, c, d;
      input [1: 0] sel;
      output out;
   assign out = sel == 2'b00 ? a :
                sel == 2'b01 ? b :
                sel == 2'b10 ? c : d;
endmodule
```

条件操作符

条件操作符的语法为：

<LHS> = <condition> ? <true_expression> : <false_expression> ;

其意为：if condition为真, 则 LHS=true_expression, 否则 LHS = false_expression；

每个条件操作符必须有三个参数，缺少任何一个都会产生错误！

register = condition ? true_value : false_value;

上式中，若condition为真则register等于true_value；若condition为假则register等于false_value。一个很有意思的地方是，如果条件值不确定，且true_value和false_value不相等，则输出不确定值。

例如：assign out = (sel == 0) ? a : b;

若sel为0则out =a；若sel为1则out = b。如果sel为x或z，若a = b =0，则out = 0；若a≠b，则out值不确定。

级联操作符

级联或称为拼接

- 可以从不同的矢量中选择位并用它们组成一个新的矢量
- 用于位的重组和矢量构造

在级联和复制时，必须指定位数，否则将产生错误！

下面是类似错误的例子：

```
a[7:0] = {4{ `b10}};
b[7:0] = {2{ 5}};
c[3:0] = {3`b011, `b0};
```

级联时不限定操作数的数目。在级联符号{ }中，用逗号将操作数分开。例如：

```
{A, B, C, D}
```

```
module concatenation;
    reg [7: 0] rega, regb, regc, regd;
    reg [7: 0] new;
  initial begin
    rega = 8'b0000_0011;
    regb = 8'b0000_0100;
    regc = 8'b0001_1000;
    regd = 8'b1110_0000;
  end
  initial fork
    #10 new = {regc[4:3], regd[7:5],
                regb[2], rega[1:0]};
    // new = 8'b1111_1111
    #20 $finish;
  join
endmodule
```

复制操作符

{{}} 复制

复制一个变量或在{ }中的值

前两个{ 符号之间的正整数指定复制次数

```
module replicate ();
    reg [3: 0] rega;
    reg [1: 0] regb, regc;
    reg [7: 0] bus;
  initial begin
    rega = 4'b1001;
    regb = 2'b11;
    regc = 2'b00;
  end
  initial fork
      #10 bus <= {4{regb}}; //bus=8`b11111111
   // regb 复制4次.
      #20 bus <= { {2{regb}}, {2{regc}} };
   // regc 和 regb 复制后的结果级联
   // bus = 11110000
      #30 bus <= { {4{rega[1]}}, rega };
   // bus = 00001001
      #40 $finish;
  join
endmodule
```

西安电子科技大学

7.3 基本逻辑电路设计

- 组合逻辑电路设计
- 时序逻辑电路设计
- 有限状态机设计

组合逻辑电路设计

- 组合电路通常包括门电路、多路选择器、编码器、译码器等电路

下面以3-8译码器为例说明组合电路的设计方法：

```
module decode (Ain, En, Yout);
    input [2:0] Ain;
    input En;
    output [7:0] Yout;
    reg [7:0] Yout;

always@(En, Ain)
begin
    if (!En)
          Yout = 8'b0;
    else
     case (Ain)
       3'b000:  Yout = 8'b00000001;
       3'b001:  Yout = 8'b00000010;
       3'b010:  Yout = 8'b00000100;
       3'b011:  Yout = 8'b00001000;
       3'b100:  Yout = 8'b00010000;
       3'b101:  Yout = 8'b00100000;
       3'b110:  Yout = 8'b01000000;
       3'b111:  Yout = 8'b10000000;
       default: Yout = 8'b00000000;
     endcase
end
endmodule
```

时序逻辑电路设计

- 时序逻辑电路就是具有记忆(或内部状态)的电路，即：时序逻辑电路的输出不但与当前的输入状态有关，而且与以前的输入状态有关。时序电路的内部状态元件可以由**边沿敏感的触发器或**由**电平敏感的锁存器**实现，但大多数时序电路采用**触发器**来实现。
- 时序电路又可分为**同步时序**电路和**异步时序**电路两种。

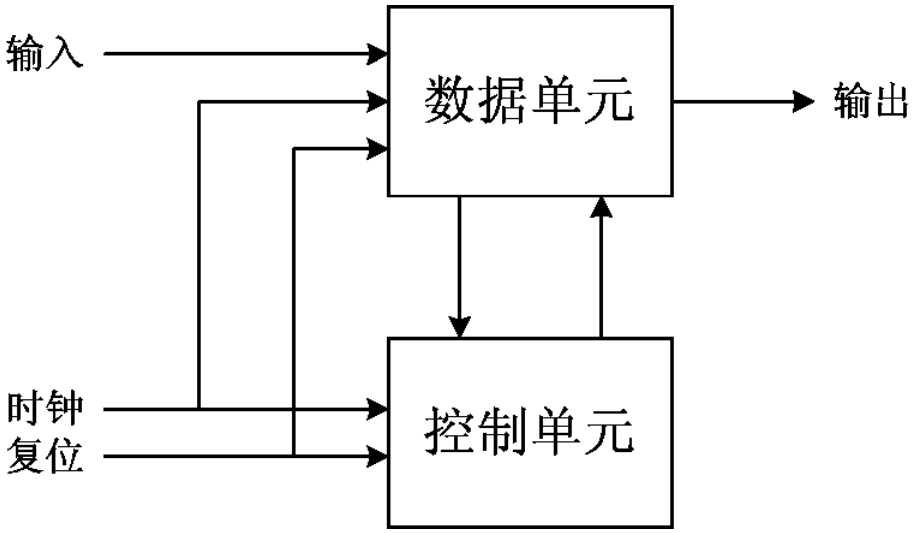
时序逻辑电路设计

- 同步计数器的设计

下面以**8-bit异步复位、同步预置、带进位输出的同步加法计数器**为例说明同步计数器的设计方法

```
module counter_load (out, cout, data, load, clk, en, reset);
parameter Width = 8;
    input load, clk, en, reset;
    input [Width-1:0] data;
    output cout;
    output [Width-1:0] out;
    reg [Width-1:0] out;
always@(posedge clk or negedge reset)
    if(!reset)
        out <= 8'b0;
    else if (load)
        out <= data;
    else if (en)
        out <= out + 1;
    else
        out <= out;
assign cout = &out;
endmodule
```

有限状态机的设计



数字系统组成框图

- 数据单元：保存运算数据和运算结果的寄存器及完成运算的组合电路。
- 控制单元：产生控制信号序列。
  - 微程序控制单元
  - 硬件实现的控制单元(有限状态机)

有限状态机的设计

- 有限状态机由三部分组成：

1. 当前状态寄存器：是用来保存当前状态矢量的一组n比特的触发器，这组触发器由一个时钟信号驱动。长度为n比特的状态矢量具有2^n个可能状态，称为状态编码。通常并不是所有的2^n个都需要，所以在正常操作时就不能出现未使用的状态。或者说，具有m个状态的有限状态机至少需要$\log_2(m)$个状态触发器。
2. 下一个状态逻辑：有限状态机在任何给定时刻只能处于一个状态，只有在时钟的有效沿上才从当前状态转移到下一个状态。这个转换过程是由“下一个状态逻辑”确定的，下一个状态是状态机的输入和当前状态的函数。在Verilog HDL设计有限状态机时一般由一个进程实现，进程的敏感信号为当前状态和状态机的输入。

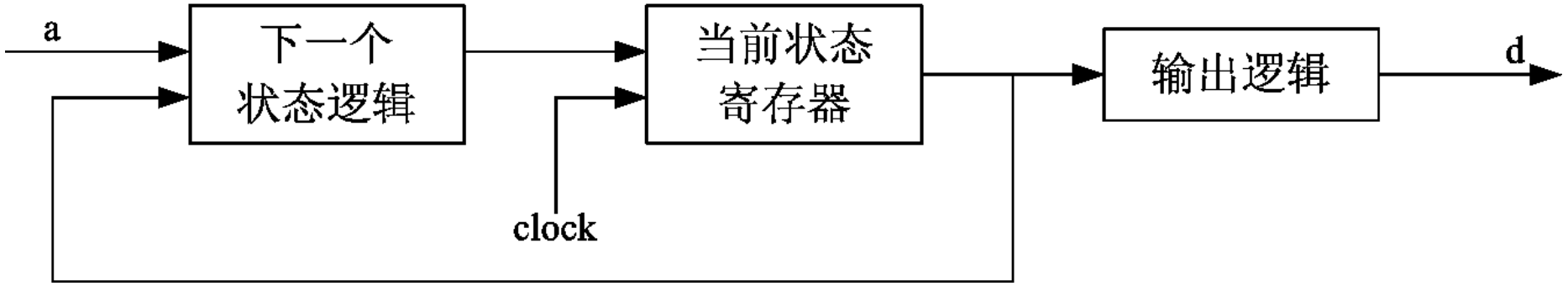
有限状态机的设计

3. 输出逻辑：有限状态机的输出通常是当前状态的函数(Moore状态机)或者当前状态和状态机输入的函数(Mealy状态机)。在Verilog设计有限状态机时一般由一个进程实现，进程的敏感信号为当前状态或者当前状态和状态机的输入。

- 同步时序系统一般可用两种模型来描述，即：Moore状态机和Mealy状态机

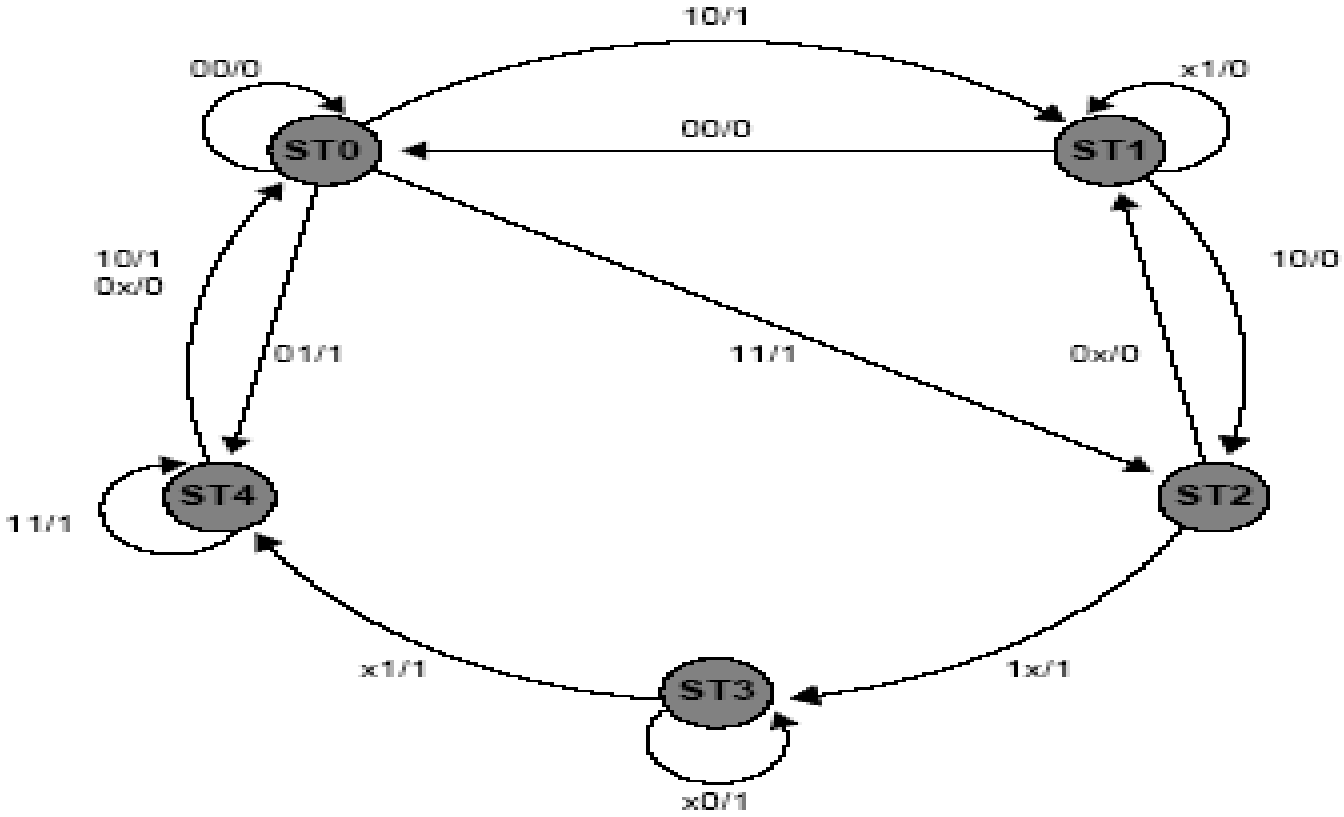
Moore状态机的设计

- Moore状态机的输出只是当前状态的函数，与当前输入无关。下图为Moore状态机的结构框图：



Moore状态机的设计

- 5个状态的状态机的状态转移图



Moore状态机的设计

```
module moore (data_in, data_out, reset, clock);
   input[1:0] data_in;
   input clock, reset;
   output data_out;
   reg    data_out;
   reg [2:0] pres_state, next_state;
   parameter st0=3'd0,st1=3'd1,st2=3'd2,st3=3'd3,st4=3'd4;

// FSM register
always@(posedge clock or negedge reset)
begin
   if (!reset) then
       pres_state <= st0;
   else
       pres_state <= next_state;
end
```

Moore状态机的设计

```
// FSM combinational block
always@(pres_state or data_in)
begin
   case (pres_state)
      st0 :
        case(data_in)
         2'b00   : next_state = st0;
         2'b01   : next_state = st4;
         2'b10   : next_state = st1;
         2'b11   : next_state = st2;
         default : next_state = st0;
        endcase
      st1 :
        case(data_in)
         2'b00   : next_state = st0;
         2'b10   : next_state = st2;
         default : next_state = st1;
        endcase
```

Moore状态机的设计

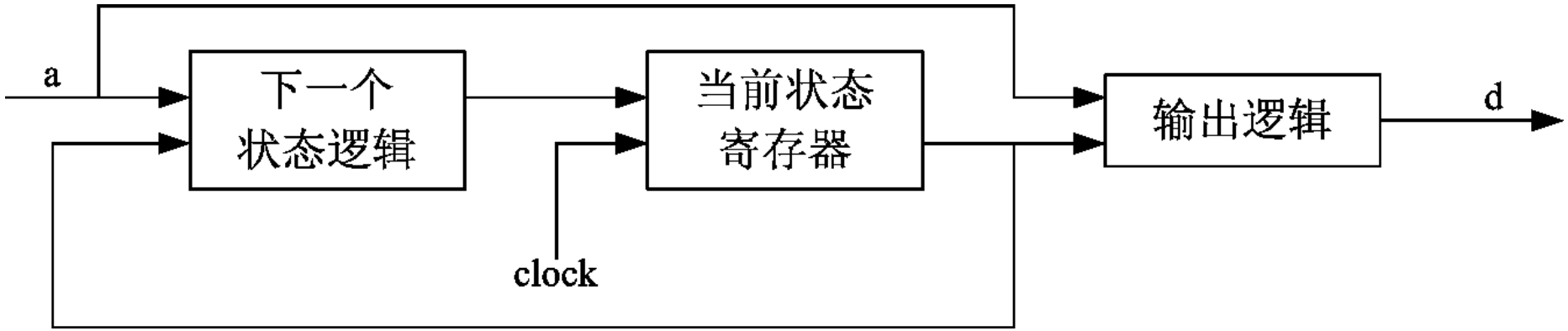
```
        st2 :
          casex(data_in)
           2'b0x   : next_state = st1;
           2'b1x   : next_state = st3;
           default : next_state = st0;
          endcase
        st3 :
          casex(data_in)
           2'bx1   : next_state = st4;
           default : next_state = st3;
          endcase
        st4 :
          case(data_in)
           2'b11   : next_state = st4;
           default : next_state = st0;
          endcase
      default : next_state = st0;
    endcase
end
```

Moore状态机的设计

```
// Moore output definition using pres_state only
always@(pres_state)
begin
   case (pres_state)
      st0     : data_out = 1'b1;
      st1     : data_out = 1'b0;
      st2     : data_out = 1'b1;
      st3     : data_out = 1'b0;
      st4     : data_out = 1'b1;
      default : data_out = 1'b0;
   endcase
end
endmodule
```

Mealy状态机的设计

- Mealy状态机的输出是当前状态和当前输入的函数。下图为Mealy状态机的结构框图：



Mealy状态机的设计

```
module moore (data_in, data_out, reset, clock);
   input[1:0] data_in;
   input clock, reset;
   output data_out;
   reg    data_out;
   reg [2:0] pres_state, next_state;
   parameter st0=3'd0,st1=3'd1,st2=3'd2,st3=3'd3,st4=3'd4;

// FSM register
always@(posedge clock or negedge reset)
begin
   if (!reset) then
       pres_state <= st0;
   else
       pres_state <= next_state;
end
```

Mealy状态机的设计

```
// FSM combinational block
always@(pres_state or data_in)
begin
   case (pres_state)
      st0 :
        case(data_in)
         2'b00   : next_state = st0;
         2'b01   : next_state = st4;
         2'b10   : next_state = st1;
         2'b11   : next_state = st2;
         default : next_state = st0;
        endcase
      st1 :
        case(data_in)
         2'b00   : next_state = st0;
         2'b10   : next_state = st2;
         default : next_state = st1;
        endcase
```

Mealy状态机的设计

```
        st2 :
          casex(data_in)
           2'b0x   : next_state = st1;
           2'b1x   : next_state = st3;
           default : next_state = st0;
          endcase
        st3 :
          casex(data_in)
           2'bx1   : next_state = st4;
           default : next_state = st3;
          endcase
        st4 :
          case(data_in)
           2'b11   : next_state = st4;
           default : next_state = st0;
          endcase
      default : next_state = st0;
    endcase
end
```

Mealy状态机的设计

```
// Moore output definition using pres_state and data_in
always@(data_in or pres_state)
begin
   case (pres_state)
      st0 :
        case(data_in)
          2'b00   : data_out = 1'b0;
          default : data_out = 1'b1;
        endcase
      st1 : data_out = 1'b0;
      st2 :
        casex(data_in)
          2'b0x   : data_out = 1'b0;
          default : data_out = 1'b1;
        endcase
      st3 : data_out = 1'b1;
      st4 :
        casex(data_in)
          2'b1x   : data_out = 1'b1;
          default : data_out = 1'b0;
        endcase
      default : data_out = 1'b0;
   endcase
end
endmodule
```

西安電子科技大學

7.4 Verilog仿真

仿真

- 仿真过程所用的工具

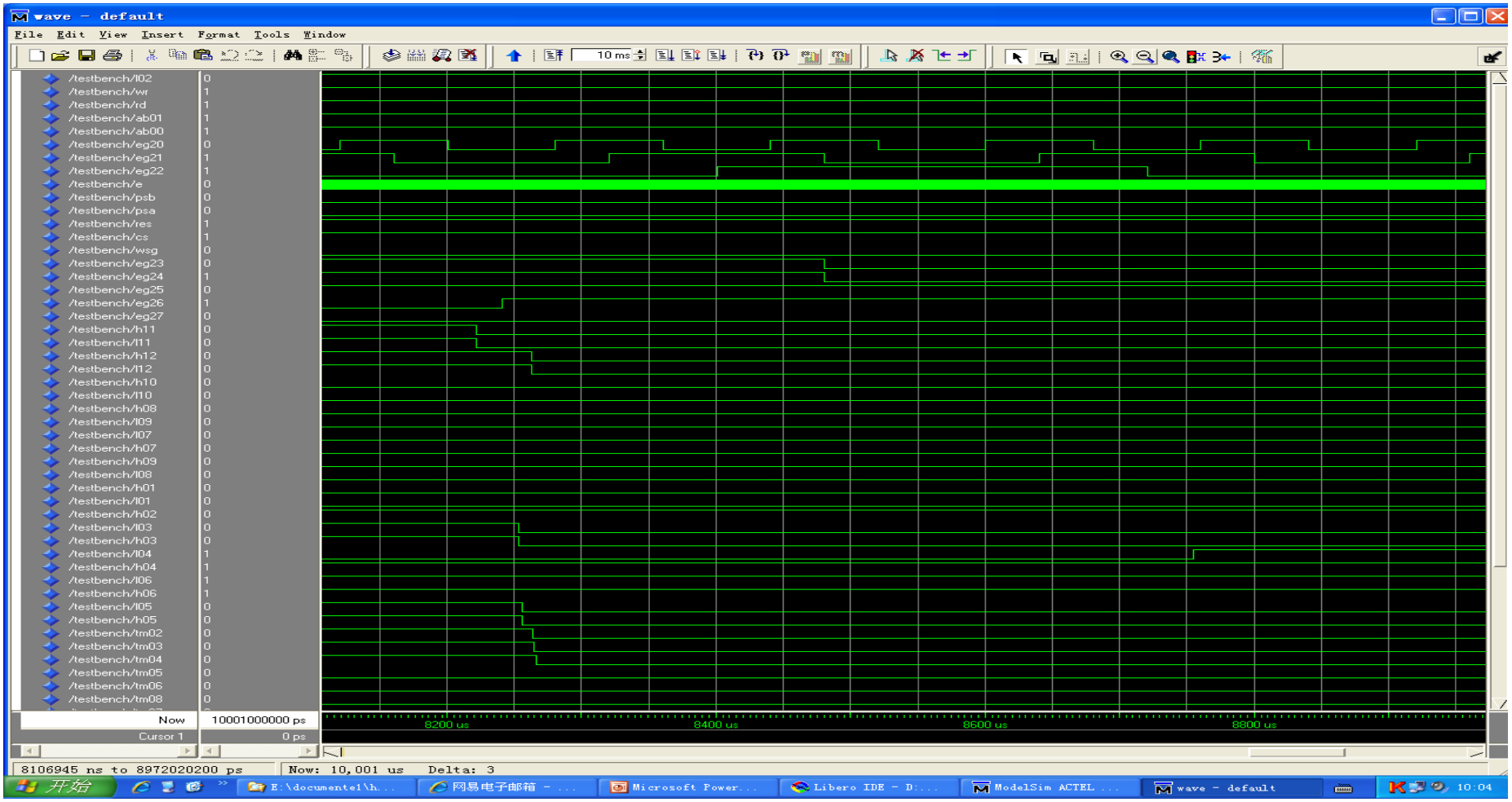
1. 仿真器(simulator)
2. 激励信号产生工具
3. 波形观察器

- 产生激励信号

1. 组合电路
2. 时序电路
3. 时钟信号
4. 数据信号
5. 控制信号

仿真

- 观察波形


wave - default
File Edit View Insert Format Tools Window
/testbench/l02
/testbench/wr
/testbench/rd
/testbench/ab01
/testbench/ab00
/testbench/eg20
/testbench/eg21
/testbench/eg22
/testbench/e
/testbench/psb
/testbench/psa
/testbench/res
/testbench/cs
/testbench/wsg
/testbench/eg23
/testbench/eg24
/testbench/eg25
/testbench/eg26
/testbench/eg27
/testbench/h11
/testbench/l11
/testbench/h12
/testbench/l12
/testbench/h10
/testbench/l10
/testbench/h08
/testbench/l09
/testbench/l07
/testbench/h07
/testbench/h09
/testbench/l08
/testbench/h01
/testbench/l01
/testbench/h02
/testbench/l03
/testbench/h03
/testbench/l04
/testbench/h04
/testbench/l06
/testbench/h06
/testbench/l05
/testbench/h05
/testbench/tm02
/testbench/tm03
/testbench/tm04
/testbench/tm05
/testbench/tm06
/testbench/tm08
Now
10001000000 ps
Cursor 1
0 ps
8200 us
8400 us
8600 us
8800 us
8106945 ns to 8972020200 ps
Now: 10,001 us Delta: 3
开始
wave - default
10:04

Verilog的仿真

- Verilog语言是一种硬件描述语言，我们设计的Verilog程序就是对数字系统的描述。为了验证所设计的模块是否正确，还必需对这些模块进行仿真。仿真采用Verilog仿真器(Simulator)进行。通过仿真器设计者可对各设计层次的设计模块进行仿真，以确定这些设计模块的功能、逻辑关系及定时关系是否满足设计要求。所以，仿真是利用Verilog语言进行硬件设计的一个必不可少的步骤，它贯穿设计的整个过程。
- 仿真可分为功能仿真和定时仿真，功能仿真用于验证设计模块的逻辑功能，定时仿真用来验证设计模块的时序关系。无论哪种仿真，都需要在输入端加输入信号，即激励信号，然后运行仿真器，仿真器根据电路模型产生所设计电路对激励信号的响应，设计者通过对响应信号的分析(如观察波形)以确定所设计电路是否正确。

Verilog的仿真

- 激励信号的产生

1. 图形产生
2. Verilog语言产生

- 测试矢量初始化通过initial过程块来实现

```
initial
begin
    clk   = 1'b0;
    reset = 1'b1;
    in    = 1'b1;
    …… ……
end
```

- 采用always过程块来实现时钟信号

```
always
begin
    #10 clk = ~clk;
end
```

西安电子科技大学

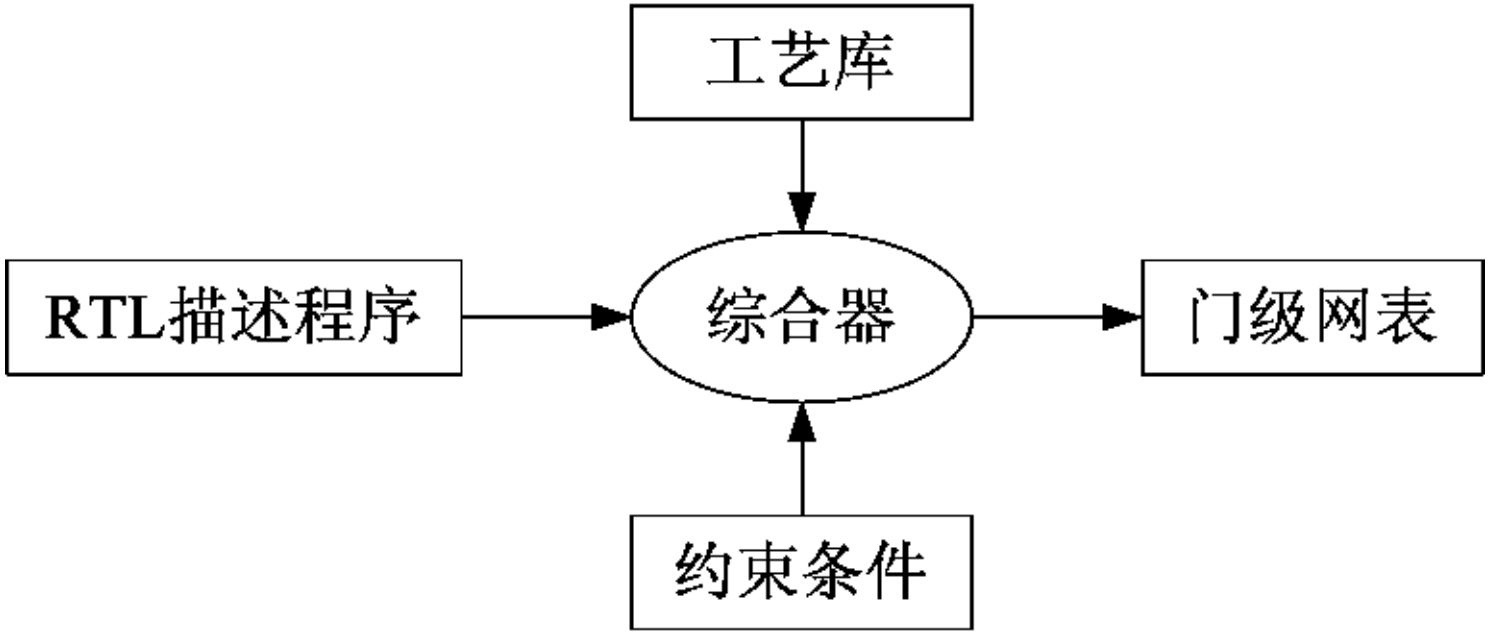
7.5 Verilog综合

逻辑综合

- 所谓综合(Synthesis)就是将高抽象层次的描述自动地转换到较低抽象层次的一种方法。通常综合可分为三个层次：即：**高层次综合**(High-Level Synthesis)、**逻辑综合**(Logic Synthesis)和**版图综合**(Layout Synthesis)，其中：高层次综合负责将系统算法层的行为描述转化为寄存器传输层的描述；逻辑综合负责将系统寄存器传输层(RTL)描述转化为门级网表的过程；版图综合负责将系统电路层的结构描述转化为版图层的物理描述。本节只介绍有关**逻辑综合**方面的内容。

逻辑综合

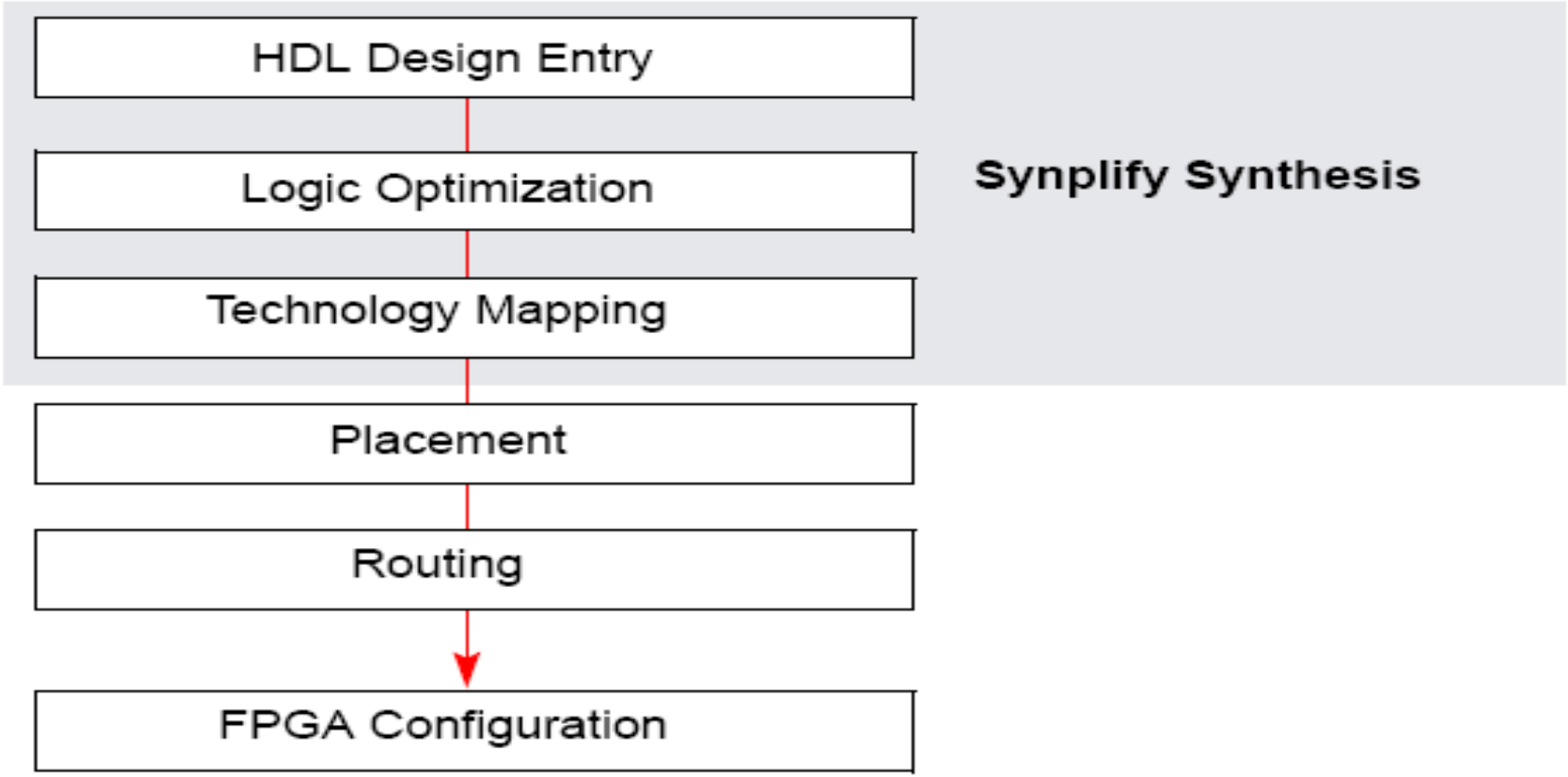
- 一般逻辑综合的过程如下图所示



- 约束条件
- 属性描述
- 工艺库

逻辑综合

- 综合工具工作过程



逻辑综合

- 优化过程

1. area optimization
2. delay optimization

- 工艺映射过程

- 可综合Verilog程序的编写

1. 了解综合软件的局限性
2. 锁存器与触发器

逻辑综合

- 应用逻辑综合工具将RTL描述转换为门级描述有三个步骤：

1. 将RTL描述转换成未优化的门级布尔描述(如与门、或门、触发器等)。该过程不受用户限制，其结果是中间结果，格式因综合工具不同而各异，且对用户是不透明的。按照转换的规则语法，将RTL描述的if、case等语句转换成中间布尔表达式，装配组成或由推论形成触发器和锁存器。
2. 执行优化算法，产生优化的布尔描述。这是逻辑综合过程中的一个重要工作，它采用了大量的算法和规则。
3. 按半导体工艺要求，采用相应的工艺库，把优化的布尔描述映射成实际的逻辑电路。

西安電子科技大學

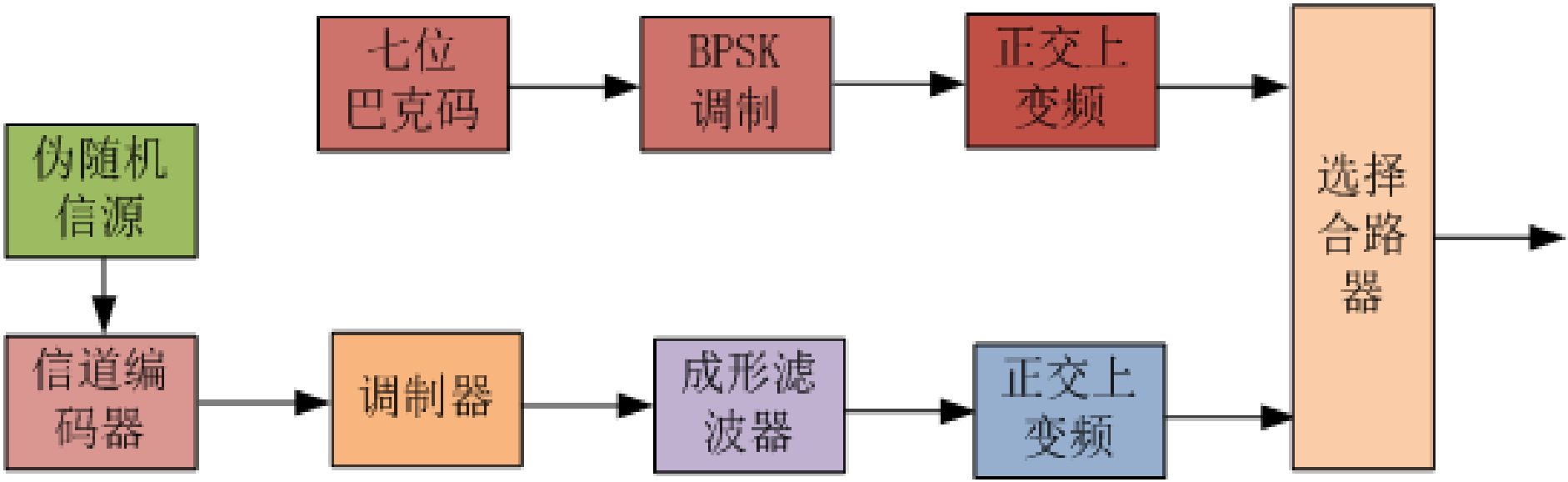
基于Vivado/ModelSim的Verilog仿真实验

仿真实验要求

- 实验一(难度C-75)：8bit异步复位、同步预置、带进位输出的同步计数器
  - 分别设计出异步复位、同步预置和使能计数状态，并通过ModelSim做出对应的波形
- 实验二(难度B-85)：深度为1024、数据位宽为8bit的同步FIFO
  - 设计出同步FIFO的写数据操作、读数据操作以及FIFO满和FIFO空的状态指示信号；
  - 设计出写FIFO、读FIFO和边读边写三种操作模式，并通过ModelSim做出对应的波形

仿真实验要求

实验三(难度A-100)：
按照下图所示，利用verilog实现通信信号的生成



仿真实验要求

实验要求：

（1）码元速率为1Msymbol/s,载波频率为50MHz；

（2）信道编码方式为Gray编码、CRC编码、汉明码等任选一种；

（3）调制方式为QPSK、8PSK、MSK、16QAM等任选一种；

（4）使用根升余弦成形滤波，成形因子0.3(5分)；

（5） 采用七位巴克码完成群同步，发送五帧数据，每帧信息为57个码元 .