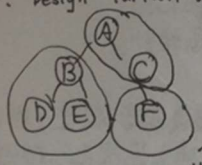
1. Design Specification 设计规格书.

① functionality 功能 ② specific computation algorition and method 算法. ③ clock frequency 时钟频率
④. supply power ⑤ Interfaces 接口 ⑥ communication protocol 通信协议. ⑦ definition of pins 端口定义 ⑧. type of package and technology, etc...

2. Design Partion 设计划分. (自顶而下).



○小模块
大模块.

①. 功能块之间连线尽可能少，接口清晰.
②. 功能块规模适理，各功能块各自独立设立.

3. Design Entry 设计实体，(尽量用 RTL Coding & Testbench)

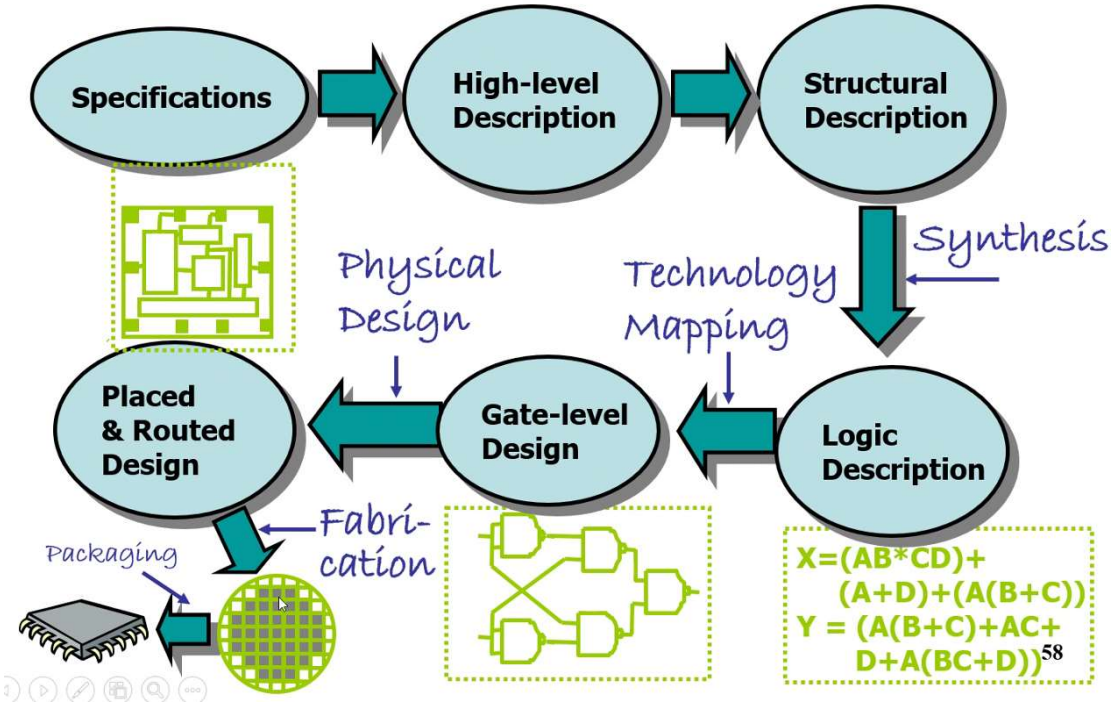
① Behavioural modeling description 行为
②. Structure modeling description
③. RTL modeling description.

4. Simulation 仿真 and Function Verification 验证. 差别：① 对象不同 ② 目标、效果不同.

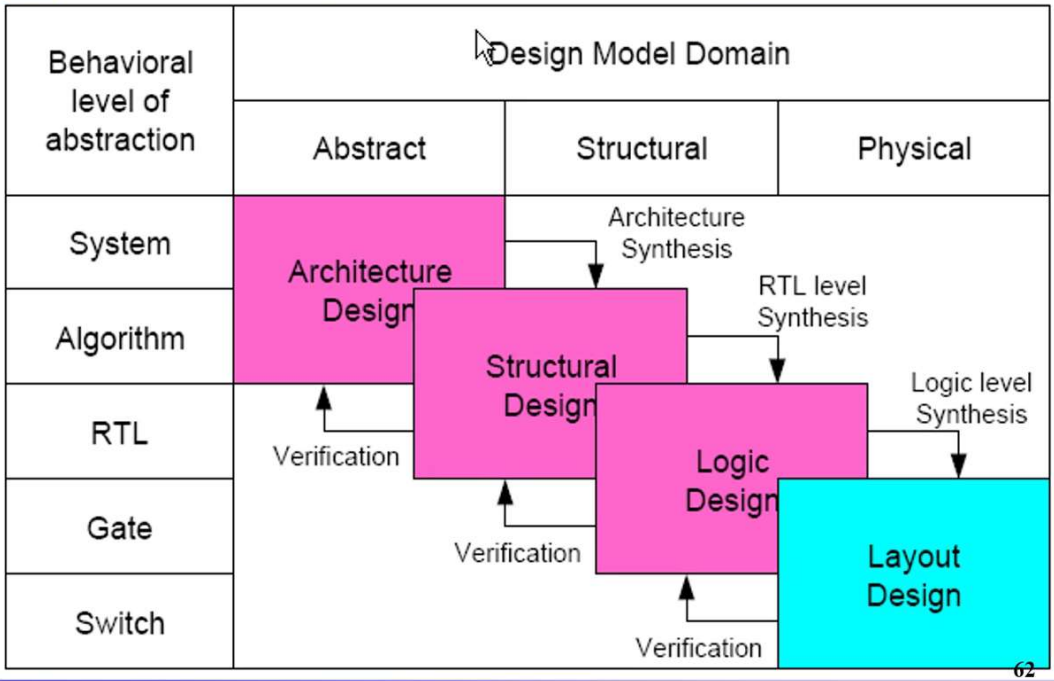
5. 测试 Testing 实物.
(测工艺错误) (用ATE自动测试仪).

6. 编译和综合. Compilation and Synthesis. (综合)

数字IC设计流程

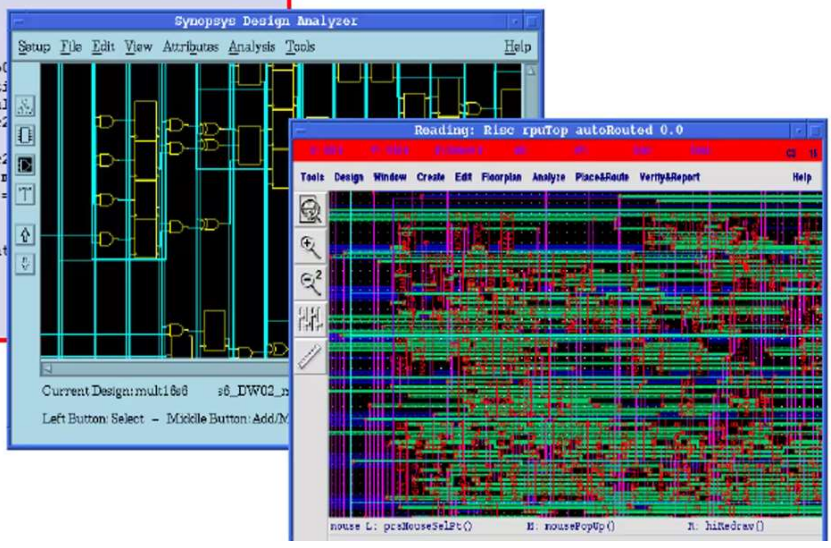


Verilog HDL in Design Domain

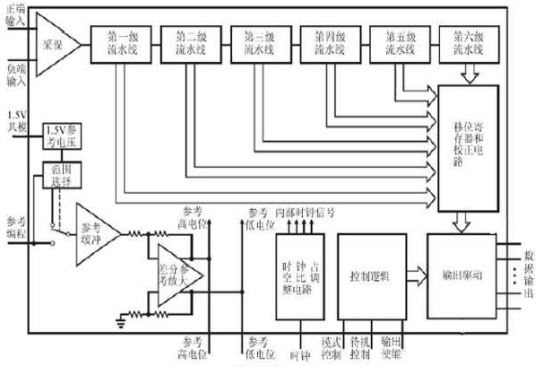


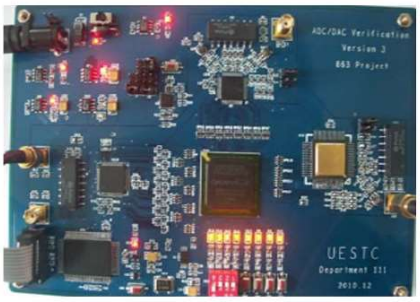
Code, Netlist, and Layout

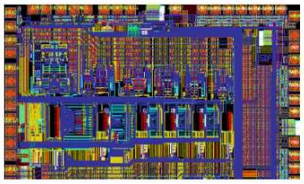
```
always @(posedge clk)
  begin: MULT
    if (reset)
      disable MULT ;

    if ( data_ready )
      begin
        result_ready = 1'b0
        multiplier2 = multi
        multiplicand2 = mul
        while (|multiplier2
          begin
            if (multiplier2
            multiplier2 = m
            multiplicand2 =
          end
        @(posedge clk)
        product = accumulat
        result_ready = 1;
      end
  end
```



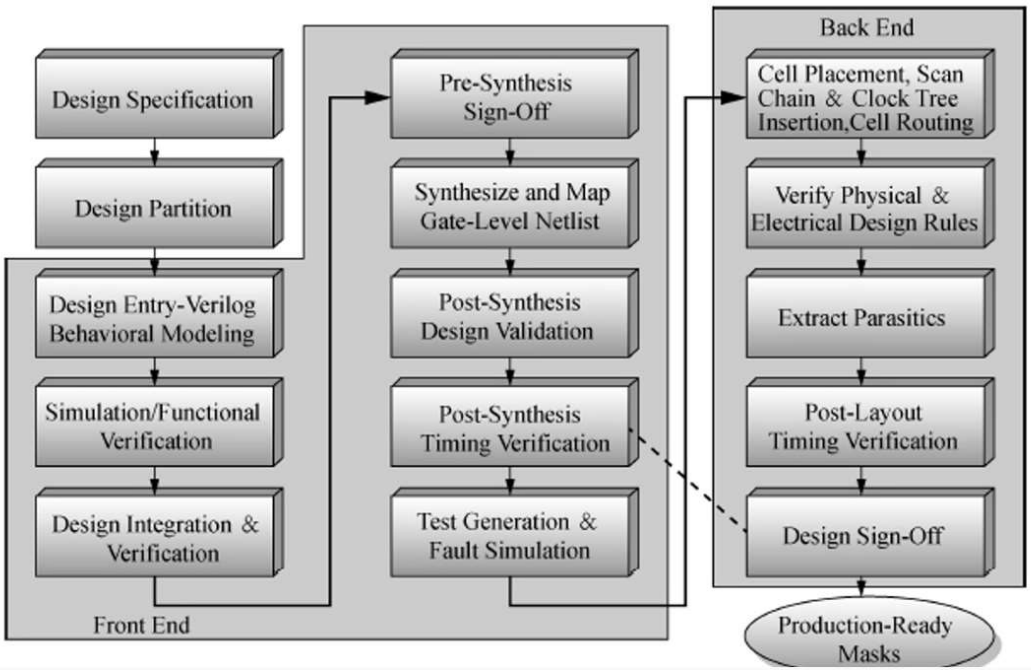
Design example: 100M14位ADC





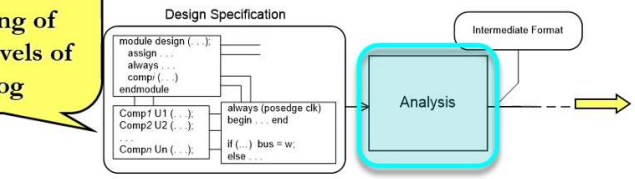


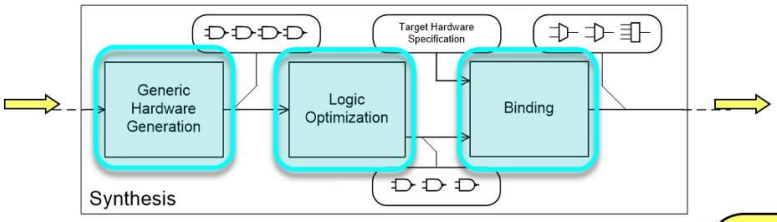
VLSI设计流程

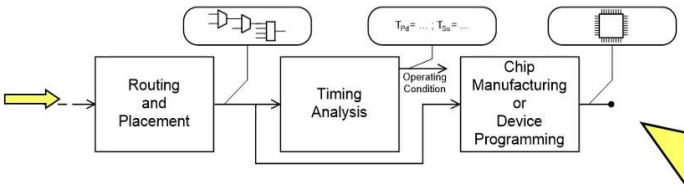


Compilation and Synthesis

Input: Hardware description consisting of various levels of Verilog







Output: A detailed hardware for programming an PLD or manufacturing an ASIC

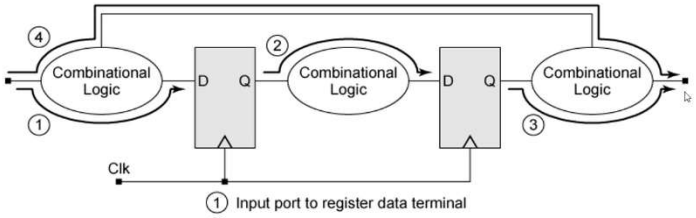
- **Compilation and Synthesis Process**

7. 时序仿真（实例后，延时变真实后）
时序分析：①建立时间，保持时间 ②时钟频率（硬件已知）→结果→各种延时.

8. Hardware Generation. 硬件产生.
FPGA 缺点：①. Array 的门逻辑会浪费，②无用的门逻辑增加延时. ③量产贵，单价不贵.
~~CPLD: Configurable~~
查找表⟺真值表.
一个有效的设计 ①. Area 小好 ②. Critical path delays 关键路径延时，小好. ③. Testability 可测性设计.
④. Power dissipation 功耗.

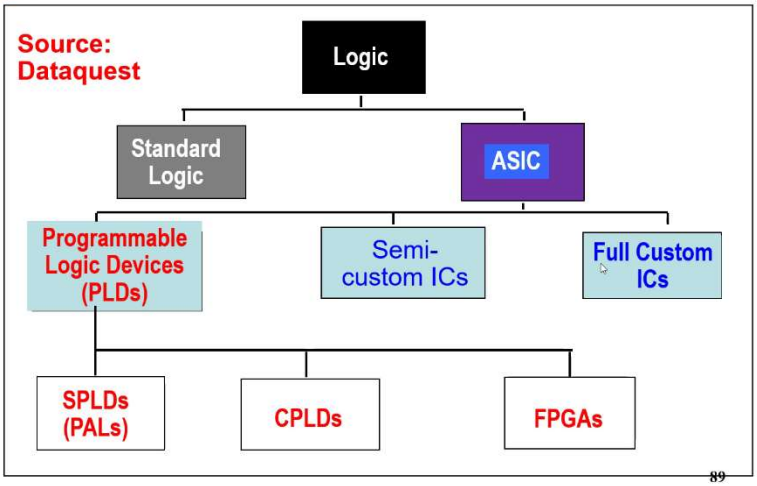
Timing Analysis

- Timing Analysis Phase generates
  - ✓ Required setup and hold times
  - ✓ Delays from one gate to another
  - ✓ Worst-case delays
  - ✓ Clocking speed
- Four types of timing paths



① Input port to register data terminal
② Register to register
③ Register to output port
④ Input port to output port

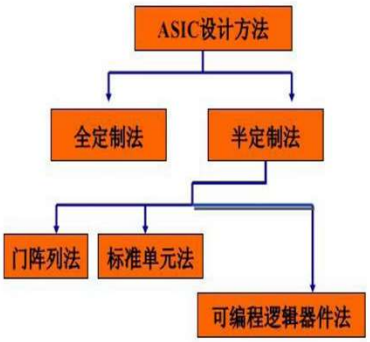
Digital ASIC Types



半定制(Semi-custom)和全定制(Full-custom)实现方法

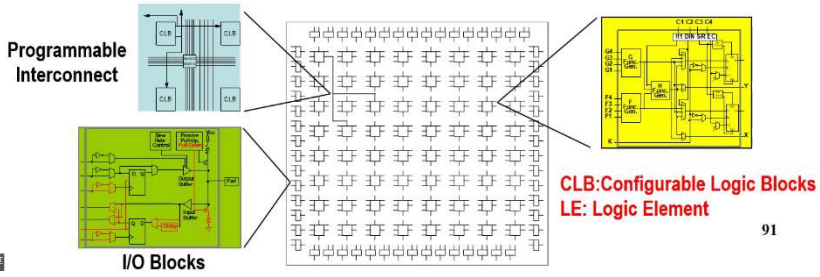
●全定制法是基于晶体管级的，手工设计版图的制造方法，设计周期较长，成本较高，重用性较差

●半定制法是基于标准单元的，根据规范流程进行开发，在EDA工具帮助下得到版图，设计周期较短，成本较低，正确率较高



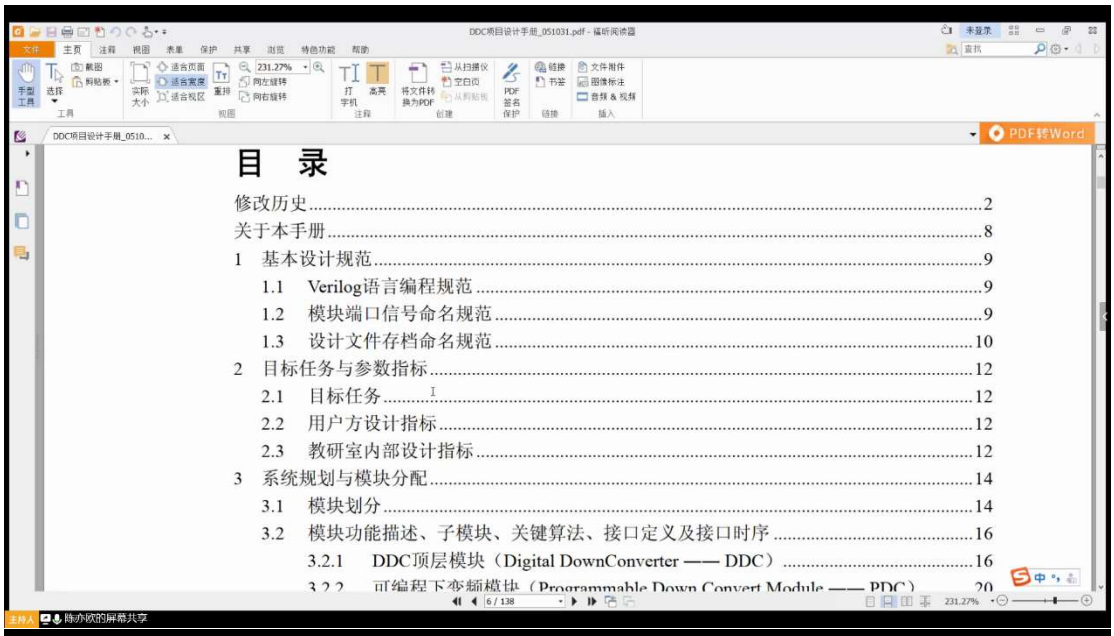
FPGA : Field Programmable Gate Array

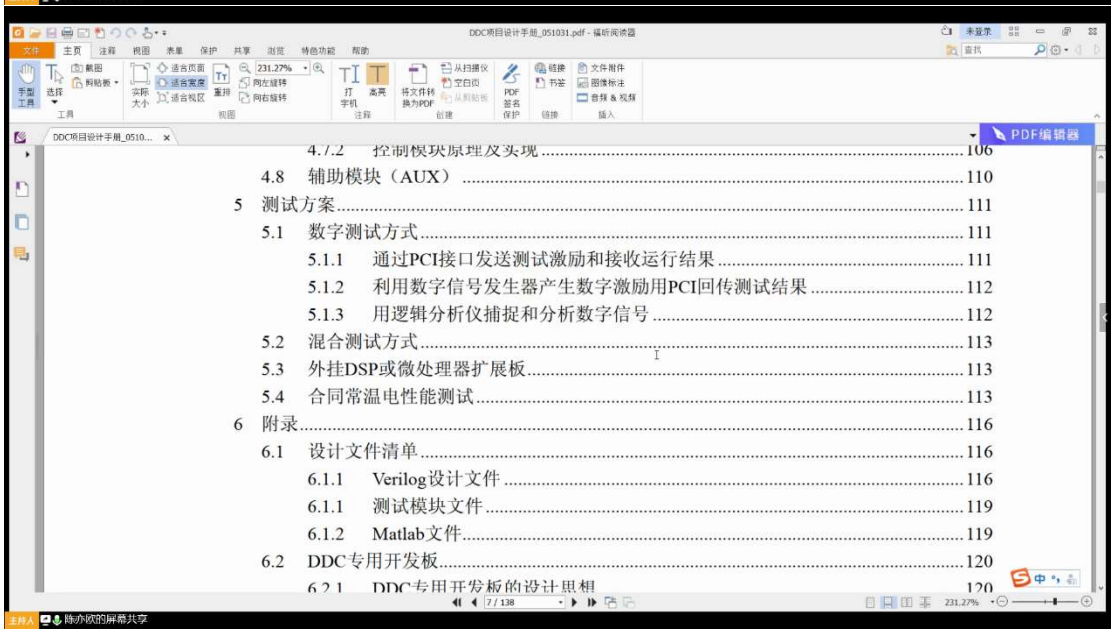
- FPGA的优点
  - FPGA设计灵活，开发周期短
  - FPGA密度提高，适于IP Core开发
  - FPGA成本较低，相应的EDA工具及实现成本较低
- 可编程性
  - 逻辑块可编程实现不同功能
  - 逻辑块之间的互联可编程实现不同连接模式



Technology Selection

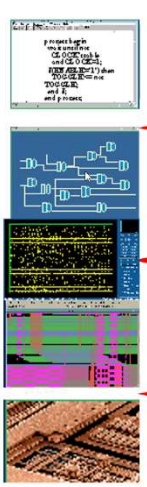
- Time-to-Market
- Performance
- Cost
- Risk
- Technology Migration

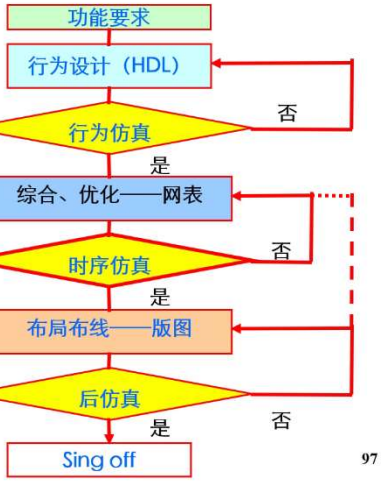

DDC项目设计手册_051031.pdf - 福昕阅读器
PDF转Word
目 录
修改历史 2
关于本手册 8
1 基本设计规范 9
1.1 Verilog语言编程规范 9
1.2 模块端口信号命名规范 9
1.3 设计文件存档命名规范 10
2 目标任务与参数指标 12
2.1 目标任务 12
2.2 用户方设计指标 12
2.3 教研室内部设计指标 12
3 系统规划与模块分配 14
3.1 模块划分 14
3.2 模块功能描述、子模块、关键算法、接口定义及接口时序 16
3.2.1 DDC顶层模块（Digital DownConverter —— DDC） 16
3.2.2 可编程下变频模块（Programmable Down Convert Module —— PDC） 20
6 / 138
231.27%
陈亦欣的屏幕共享


DDC项目设计手册_051031.pdf - 福昕阅读器
PDF编辑器
4.7.2 控制模块原理及实现 106
4.8 辅助模块（AUX） 110
5 测试方案 111
5.1 数字测试方式 111
5.1.1 通过PCI接口发送测试激励和接收运行结果 111
5.1.2 利用数字信号发生器产生数字激励用PCI回传测试结果 112
5.1.3 用逻辑分析仪捕捉和分析数字信号 112
5.2 混合测试方式 113
5.3 外挂DSP或微处理器扩展板 113
5.4 合同常温电性能测试 113
6 附录 116
6.1 设计文件清单 116
6.1.1 Verilog设计文件 116
6.1.1 测试模块文件 119
6.1.2 Matlab文件 119
6.2 DDC专用开发板 120
6.2.1 DDC专用开发板的设计思想 120
7 / 138
231.27%
陈亦欣的屏幕共享

IC设计实例：数字下变频器（Digital Down Converter）芯片设计

设计创意
+
仿真验证

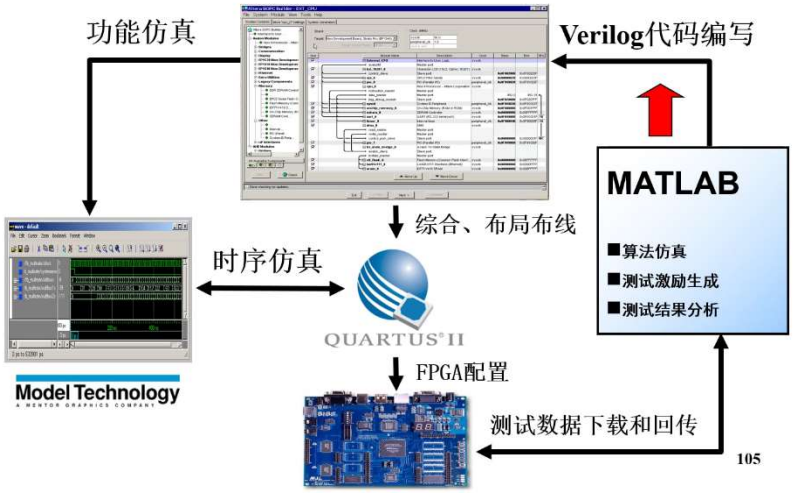




97

主持人 陈亦欧的屏幕共享

设计步骤

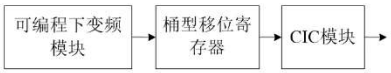


105

主持人 陈亦欧的屏幕共享

CIC模块设计

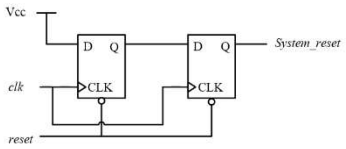
- 输入数据的位宽扩展
  - 避免由滤波器处理增益造成输出数据溢出
  - 当*D*较小时滤波器输出的有效位数少

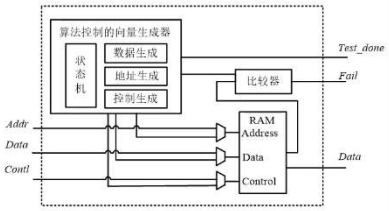


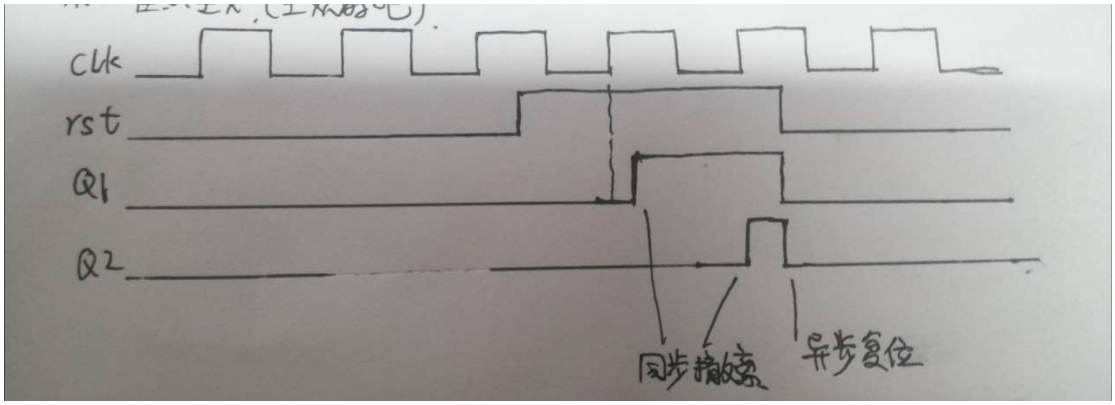
  - 移位值：$SN=fix[25-\log_2(D^5)]$
- 时钟域变换
  - 抽取滤波后进行
  - 两个数据穿越了时钟域：
    - CIC滤波器抽取后数据：寄存两拍传递
    - CIC滤波器抽取指示信号：特殊策略

复位信号、存储器设计

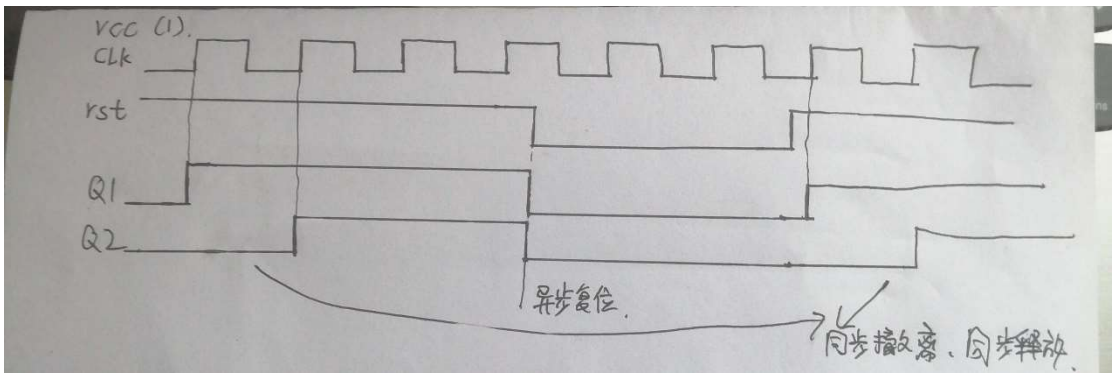
- 异步复位、同步撤离机制
  - 不同时钟域使用不同的复位信号
- 存储器内建自测试
  - 不影响正常功能
  - March算法
  - 99个RAM使用一组向量生成器和比较器
  - 芯片测试时：Test_done信号为1时，若Fail信号变为1，则存储器有物理缺陷







错误。



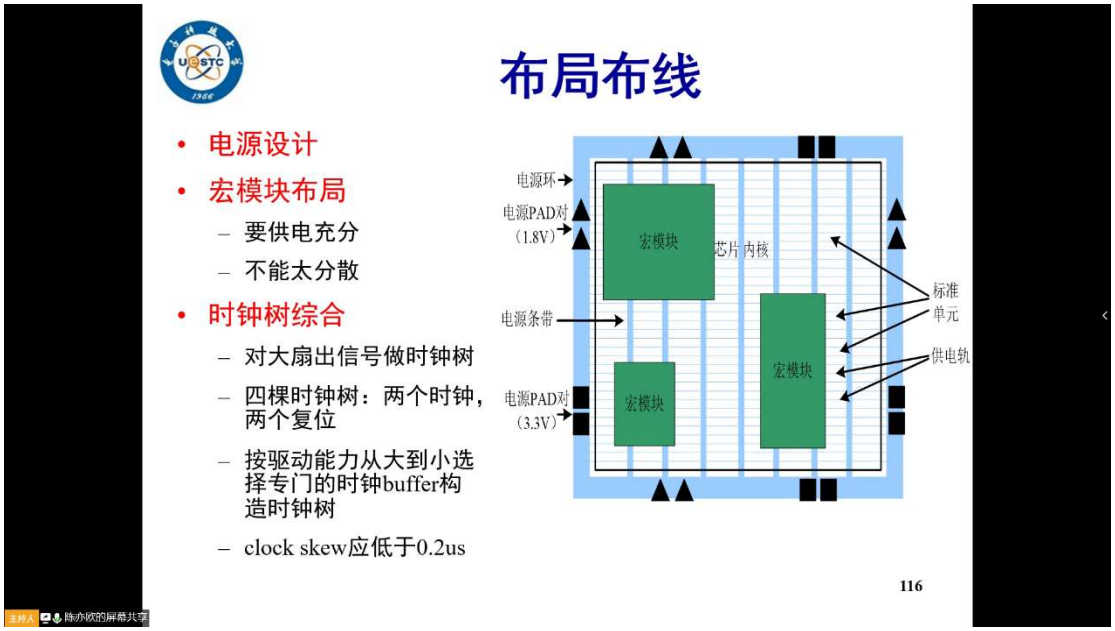
同步复位：就是指复位信号只有在时钟上升沿到来时才有效。

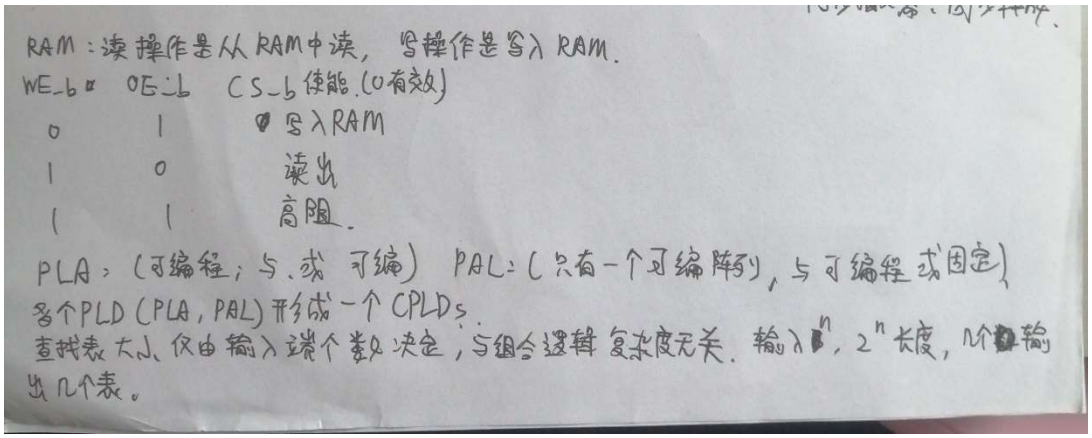
异步复位：指无论时钟沿是否到来，只要对复位信号有效，就对系统进行复位。

异步复位同步释放：在复位信号到来时不受时钟沿影响，而在复位信号释放时受到时钟信号的同步。

参考链接 https://blog.csdn.net/qq_36126787/article/details/60958140?utm_source=app

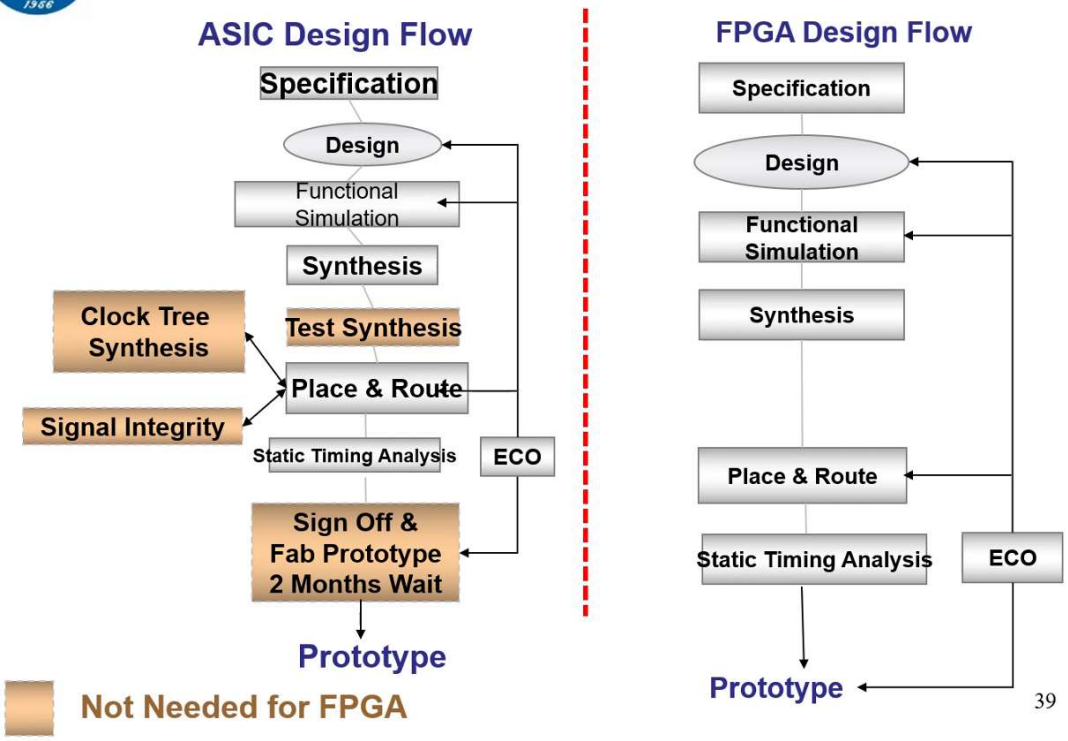
复习时可看一下，代码即可明白。





FPGA – Test. SignalProbe：利用空连接未使用的节点和IO端口使用
SignalTap II：内部逻辑分析仪（自带）
逻辑块：ASIC 标准库 FPGA 自适应逻辑块（可编程）
存储器：Memory Compiler 产生 全都是存储器。
I/O I/O块库（标准库） 可编、专用IO。

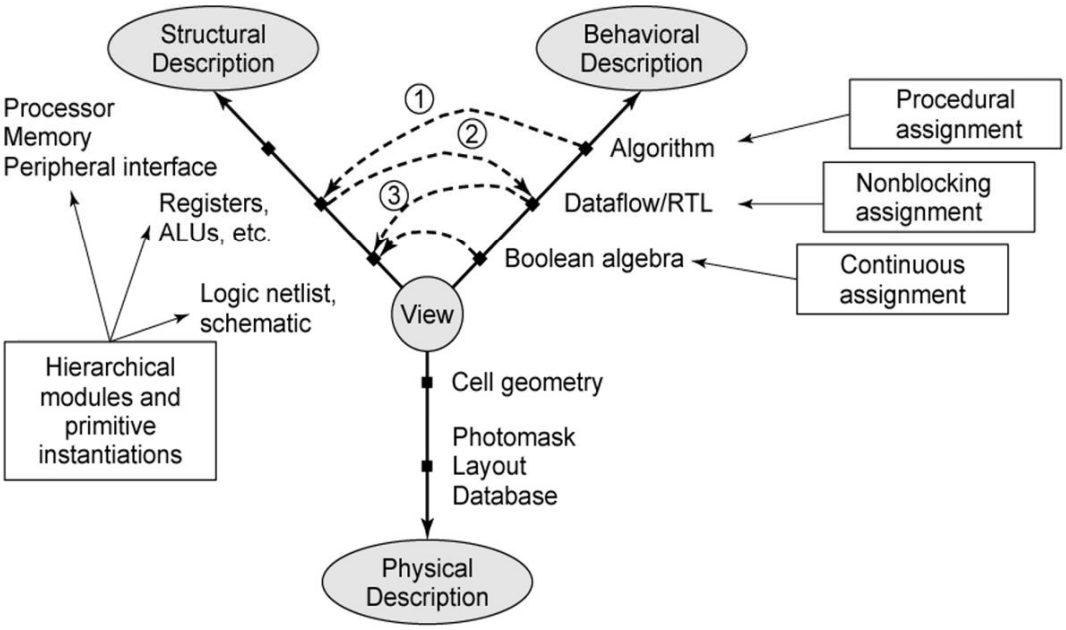
ASIC vs. FPGA Design Flow



ASIC Design v. FPGA : Primary Building Elements

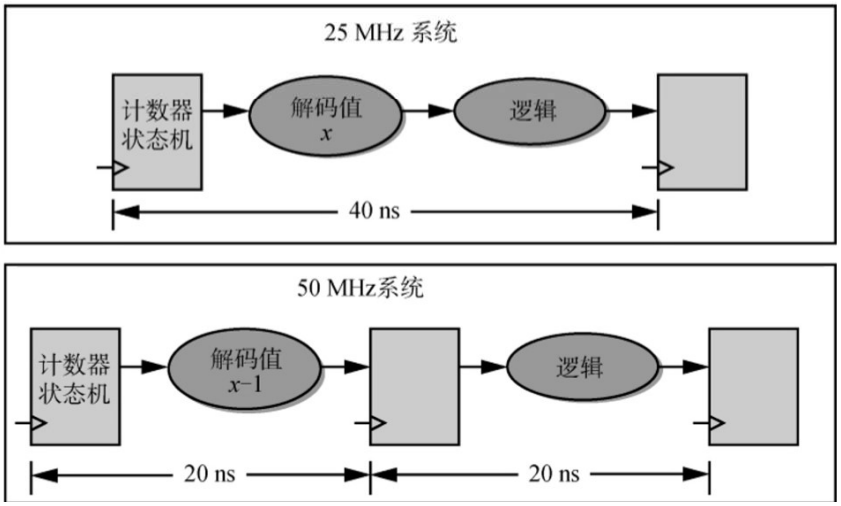
	ASIC	FPGA
Logic Cells	Library of Standard Cells	Adaptive Logic Module with Programmability
Memory	Generated through Memory Compiler	Memory Blocks
I/O	I/O Cell Library	Programmable IOs & Dedicated Special IO Circuitry
Routing	Detailed	Segmented, Distributed & Dedicated
Dedicated Resources	None; but can embed any Custom Blocks	PLL DSP

Y形图



RTL设计

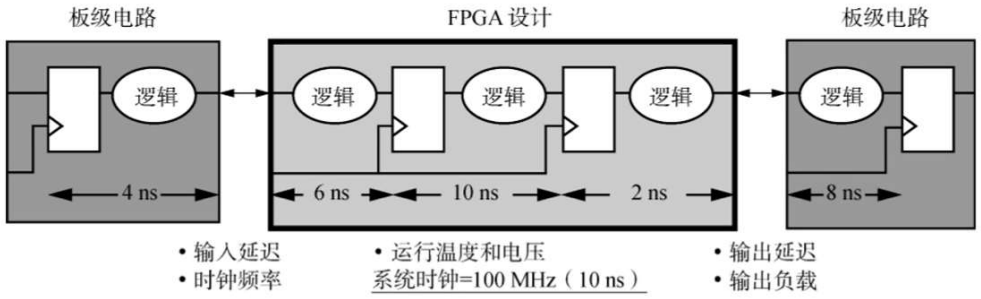
- 具有可移植性：RTL代码通常不包含电路的时间（路径延迟），也不包含电路的面积
- **Think in hardware！！**



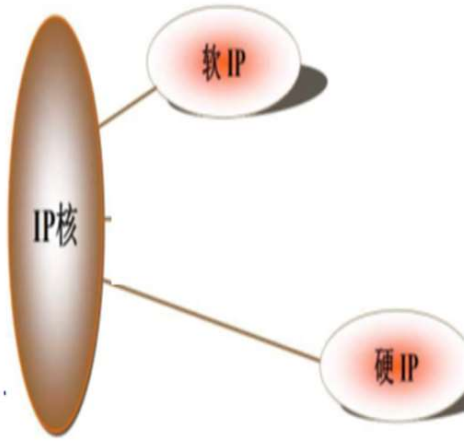
RTL:寄存器转换级

- 明确设计约束

$$T\_clk\_period \geq (T\_clk\_Q + T\_pd(logic) + T\_wiring\_delay + T\_setup + T\_clk\_skew)$$



- IP可以分为:
  - ✓Hard IP：硬IP
  - ✓Soft IP: 软IP
- 完成IP模块设计所花费的代价,硬IP代价最高
- IP模块的使用灵活性,软IP的可重复使用性最高



PLD基本结构大致相同，根据与或阵列是否可编程分为三类：

（1）与固定、或编程：ROM和PROM

（2）与或全编程：PLA

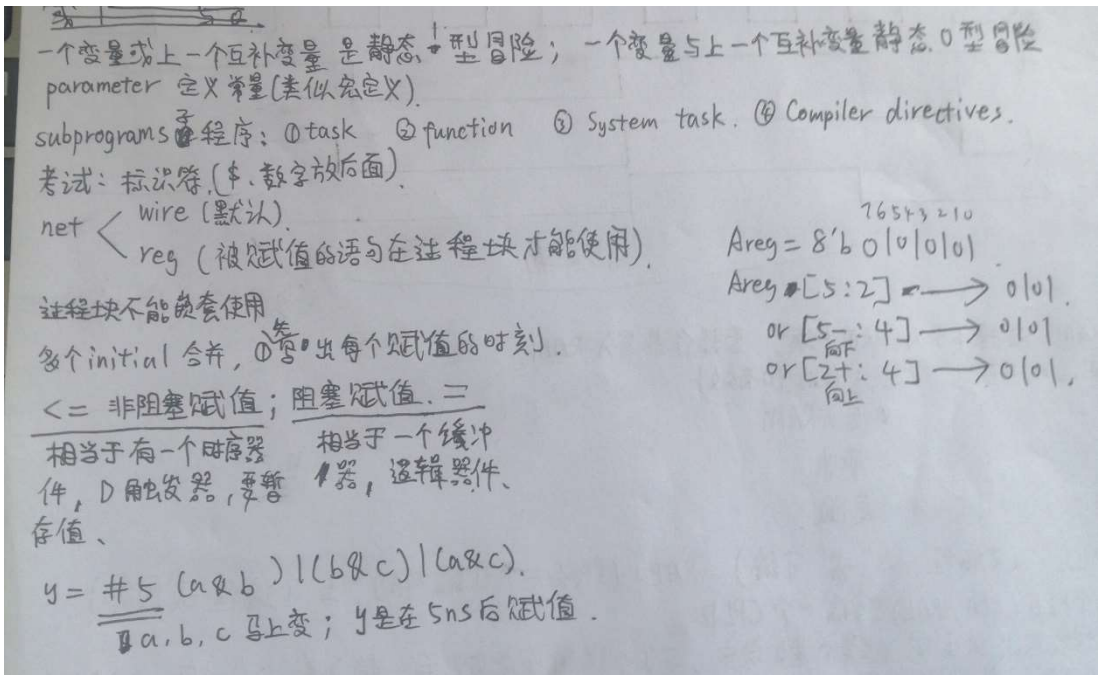
（3）与编程、或固定：PAL、GAL和HDPLD

ROM：OR 可编程

PAL: AND 可编程

PLA：都可编程。（O 和 A 对应）

A+A'=1（所以静态 1 冒险）A 、 A'=0（所以静态 0 冒险）



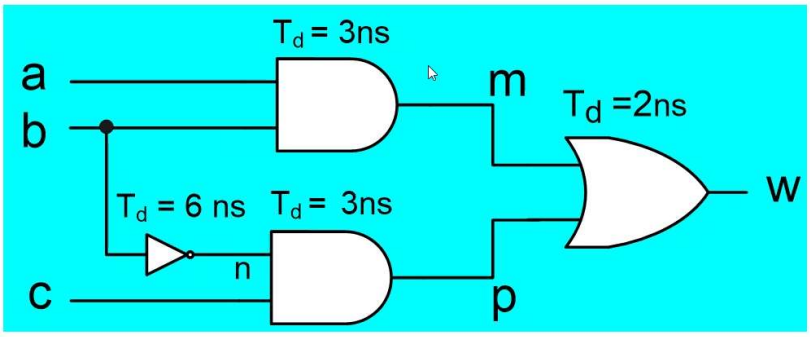
过程块里的顺序与并行语句组

- 顺序语句组
  - begin … end：各条语句顺序执行
- 并行语句组
  - fork … join ： 各条语句并行执行
- 用于仿真的计算机是串行执行的，Verilog仿真器用来模拟硬件的并行行为的方式类似于软件中的多任务操作系统，Verilog仿真是用串行的语义来模拟并行硬件工作的方式

并发性（Concurrency）

```
assign #6 n = ~b;
assign #3 m = a & b;
assign #3 p = n & c;
assign #2 w2 = m | p;
```

并发的
执行顺序与语句在代码里的位置无关

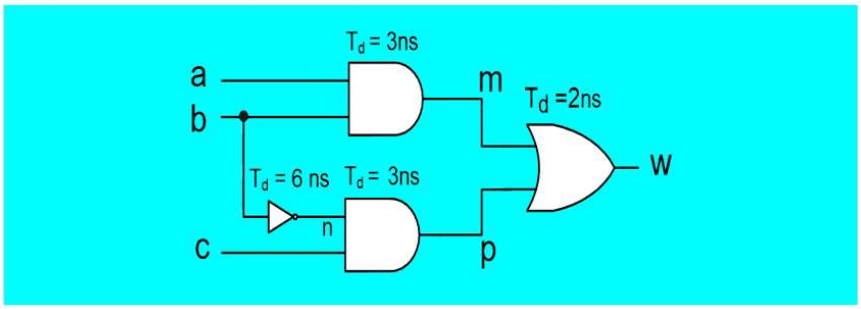


- **An AND-OR Circuit**

时序性

布尔逻辑表达式

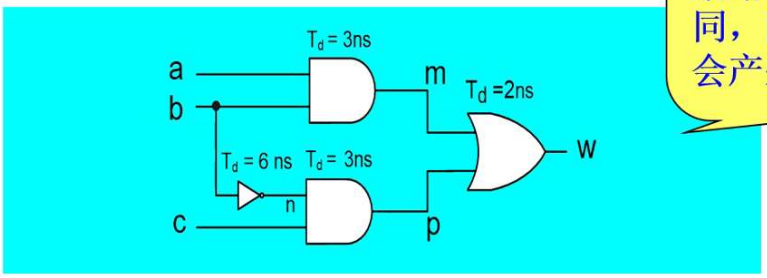
```
assign w1 = a & b | c & ~b;
```



- An AND-OR Circuit

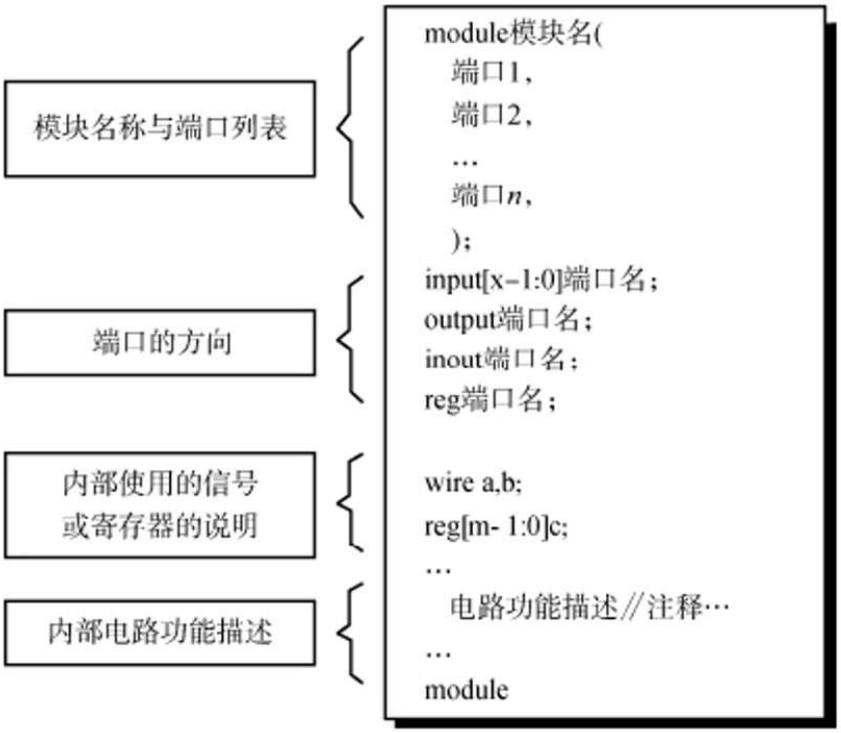
```
assign w1 = a & b | c & ~b;
```

由于门电路是有延迟的，由于各条路径的延迟不同，因此输出**w**会产生毛刺

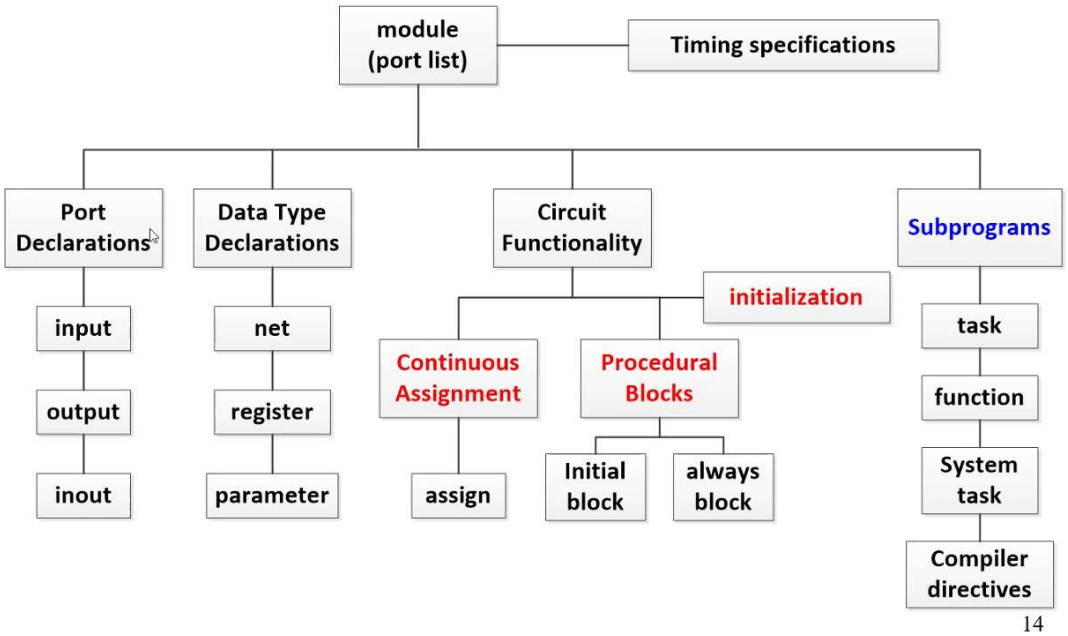


- An AND-OR Circuit

4.1 Verilog HDL的基本结构



Components of a Verilog Module

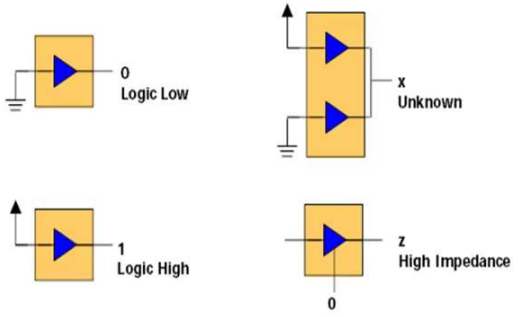


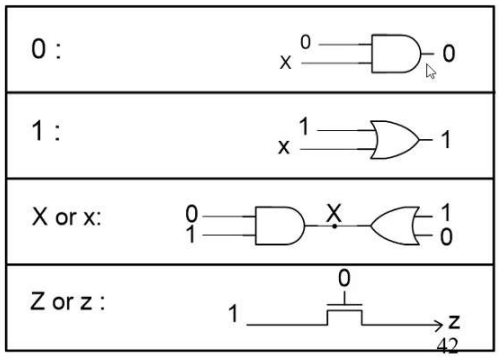
- Verilog是大小写敏感的（*a* 和*A* 是不同)
- 标识符可以是任意一组字母、数字、$（美元符）或_（下划线）的组合
- 标识符的第一个字符必须是字母或下划线

默认情况下端口类型为net，默认net类型为wire
输出端口可定义为reg类型，reg类型端口可在过程块中被赋值
输入/输出端口必须定义为net类型

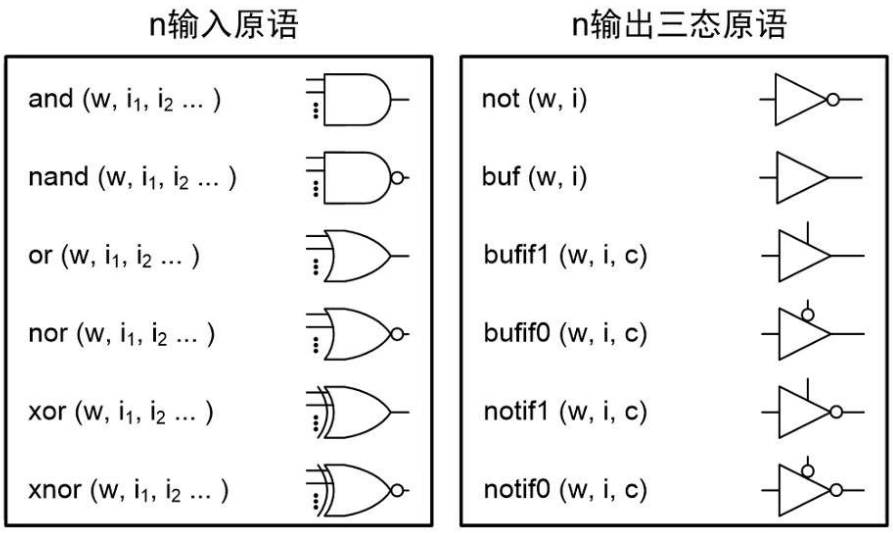
- 多条阻塞赋值是顺序执行的，而多条非阻塞赋值语句是并行执行的
- 注意事项：
  - 使用always块描述组合逻辑时使用阻塞赋值，描述时序逻辑时使用非阻塞赋值
  - 不要在同一个always块里同时使用阻塞赋值和非阻塞赋值
  - 不能在不同的always块里对同一个变量进行赋值

- **Verilog中连线和变量采用4值逻辑系统**
- **逻辑值可为：0, 1, X 和Z**
- **X（或x）表示不确定值**
- **Z（或z）表示未被驱动的高阻值**





Verilog 有26个预定义的门原语，描述基本的逻辑功能

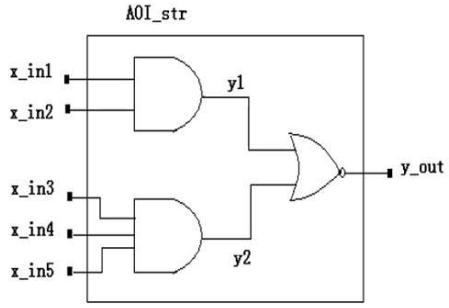



```
and #(2, 4)
        ( im1, a, b ),
        ( im2, b, c ),
        ( im3, c, a );
or #(3, 5) (y, im1, im2, im3);
```

与门的3个实例共享门名称和延迟参数

- Verilog的内置原语对应于基本逻辑门，并且它不包含时序部分
- UDP的声明与模块声明类似，封装在关键字primitive 和 endprimitive之间
- UDP利用状态表来描述时序行为或者较为复杂的组合逻辑
  - 状态表以关键字table开始，endtable结束

例4.11：5输入与或非电路的UDP



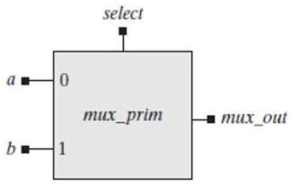

```
primitive AOI_UDP
    (y, x_in1, x_in2, x_in3,
    x_in4, x_in5);

 output y;
 input
    x_in1,x_in2,x_in3,x_in4,
    x_in5;
```

```
table
//x1 x2 x3 x4 x5:  y
  0  0  0  0  0 : 1;
  0  0  0  0  1 : 1;
  0  0  0  1  0 : 1;
  0  0  0  1  1 : 1;
  0  0  1  0  0 : 1;
  0  0  1  0  1 : 1;
  0  0  1  1  0 : 1;
  0  0  1  1  1 : 0;

  0  1  0  0  0 : 1;
  0  1  0  0  1 : 1;
  0  1  0  1  0 : 1;
  0  1  0  1  1 : 1;
  0  1  1  0  0 : 1;
  0  1  1  0  1 : 1;
  0  1  1  1  0 : 1;
  0  1  1  1  1 : 0;

  1  0  0  0  0 : 1;
  1  0  0  0  1 : 1;
  1  0  0  1  0 : 1;
  1  0  0  1  1 : 1;
  1  0  1  0  0 : 1;
  1  0  1  0  1 : 1;
  1  0  1  1  0 : 1;
  1  0  1  1  1 : 0;

  1  1  0  0  0 : 0;
  1  1  0  0  1 : 0;
  1  1  0  1  0 : 0;
  1  1  0  1  1 : 0;
  1  1  1  0  0 : 0;
  1  1  1  0  1 : 0;
  1  1  1  1  0 : 0;
  1  1  1  1  1 : 0;
endtable
endprimitive
```


每个UDP只有一个输出端口，并且这个端口是标量的（单比特），只能为0或1，不能指定为Z；若输入组合没有指定，输出为X

UDP里不能调用其他模块或原语

例：二输入多路复用器的UDP



- 真值表中列的顺序对应输入端口的顺序
- 端口只能为input和output，不能为inout
- 端口上只能为0,1和x
- 仿真时若输入为z，则将其作为x
- 最后一列为输出

```
primitive mux_prim (output mux_out, input select, a, b);
   table
//select    a       b   :    mux_out
  0          0       0   :     0;
  0          0       1   :     0;
  0          0       x   :     0;
  0          1       0   :     1;
  0          1       1   :     1;
  0          1       x   :     1;

//select    a       b   :    mux_out
  1          0       0   :     0;
  1          1       0   :     0;
  1          x       0   :     0;
  1          0       1   :     1;
  1          1       1   :     1;
  1          x       1   :     1;

  x          0       0   :     0;
  x          1       1   :     1;
   endtable
endprimitive
```


■ 用缩写符“？”减少真值表中输入数据的行数

■ 符号“？”代表：0,1或者x

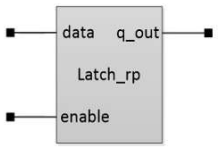
```
table
//select    a       b   :    mux_out
  0          0       0   :     0;
  0          0       1   :     0;
  0          0       x   :     0;
```

- 时序UDP可能是电平敏感的，或者是边沿敏感的，或者是两者的组合
- 时序UDP的真值表格式为：

  输入1 输入2 … ： 当前状态 ：输出（下一状态）
- 时序UDP的输出必须声明为reg类型，因为输出值是由真值表抽象产生的，在仿真过程中保存在存储器中，在不满足响应条件时是要保持原值的

电平敏感元件的UDP设计

- 例：透明锁存器的UDP设计

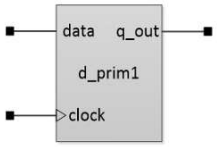


功能描述：

- 正常情况：
  - ✓ 当enable为1时，则将data的值传给q_out；
  - ✓ 当enable为0时，则q_out维持原值；
- 特殊情况：当enable为x时
  - ✓ 当data与state值相同时，输出维持原值；
  - ✓ 当data与state值不同时，输出为x。

```
primitive latch_rp (q_out, enable, data);
  output q_out;
  input  enable, data;
  reg  q_out;
  table
// enable  data   state   q_out/next_state
    1       1   :   ?   :   1 ;
    1       0   :   ?   :   0 ;
    0       ?   :   ?   :   - ;
// above entries do not deal with enable=x.
// ignore event on enable when data=state:
    x       0   :   0   :   - ;
    x       1   :   1   :   - ;
// note: the table entry '-' denotes no
      change of the output
  endtable
endprimitive
```


例：D触发器的UDP设计



功能描述：

- 当clock上升沿到来时，将data的值传给q_out；
- 其余时刻，q_out的值保持不变
  - clock下降沿
  - clock没有沿跳变

```
primitive d_prim1 (q_out, clock, data);
   output  q_out;
   input   clock, data;
   reg     q_out;
   table
  //   clk   data   state    q_out/next_state
       (01)   0   : ?    :  0 ;   //rising clock edge
       (01)   1   : ?    :  1 ;
       (0X)   1   : 1    :  1 ;
       (0X)   0   : 0    :  0 ;
       (?0)   ?   : ?    :  - ;  // falling or steady
                                    //clock  edge
        ?    (??)  : ?    :  - ; // steady clock, ignore
                                    //data transitions
   endtable
endprimitive
```

UDP设计总结

- 选择用UDP还是module实现功能单元的准则：
  - 只有唯一输出端口的单元，才能采用UDP建模
  - 输入端口数过多的单元不适合采用UDP实现
    - 随着输入端口数增大，UDP状态表的输入组合项数呈指数级增长
- 尽可能完整的描述UDP的状态表
- 尽可能用缩写符来表示状态表的输入项组合
- 电平敏感的状态表的输入项，其优先级高于边沿敏感的状态表的输入项

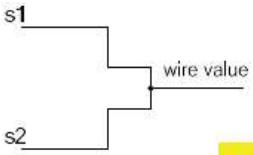
- 当设定的位宽与实际数字位宽不一致时：
  - 设定位宽比实际位宽少：自动截取左边超出的位数
  - 设定位宽比实际位宽多：在左边不够的位置补0
  - 如果遇到x或者z，位宽大于实际数字的位宽，在左边补x

- “b”表示：输入为0或1
- “p”表示：输入为广义上升沿，即(01)、(0x)或(x1)

多驱动的信号解析

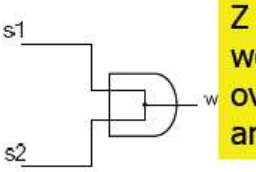
连线（wire）

s1 \ s2	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z



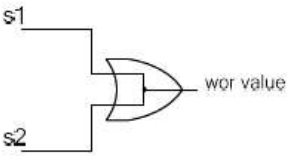
线与（wand）

s1 \ s2	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z



线或（wor）

s1 \ s2	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

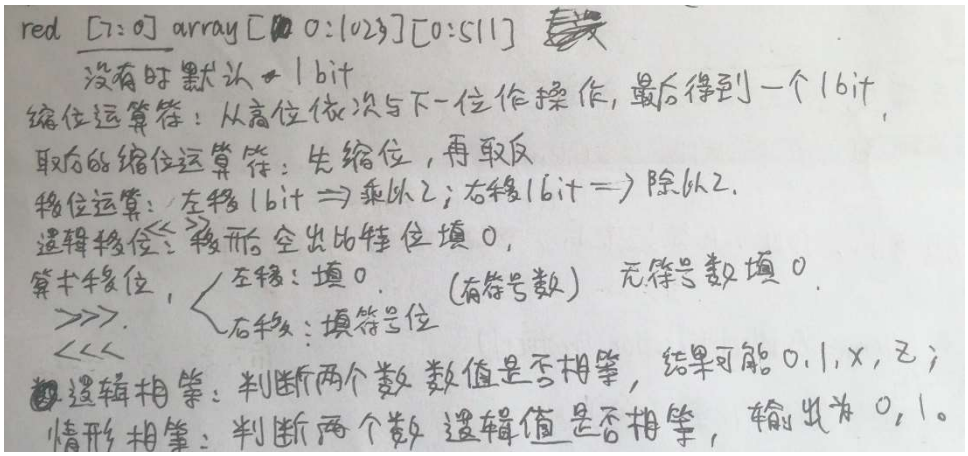


The X is the strongest, and the Z value is the weakest and is overridden by 0, 1 and X values.

如何选择正确的数据类型

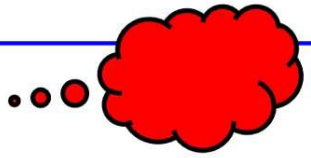
信号类型确定方法：

- ✓信号可以分为端口信号和内部信号。出现在端口列表中的信号是端口信号，其它的信号为内部信号
- ✓对于端口信号：
  - ➢输入端口只能是net类型
  - ➢输出端口默认为是net类型，也可以定义成register类型。若输出端口在过程块中赋值则为register类型；若在过程块外赋值(包括实例化语句），则为net类型
- ✓内部信号类型与输出端口类似



例:未定义位宽

```
reg counter;
initial begin
  counter=0;  repeat (20)  #5 counter=counter+1;
end

counter=?
```



解释：

这是使用Verilog HDL设计数字系统时常犯的错误，即定义counter变量时仅定义了变量类型，没有定义位宽，因而Verilog HDL将其位宽默认为1位。因为一个二进制数只能表示0或者1两个数字，所以counter无论执行多少次加1的操作，其值也只能为0或者1

✓ reg [7:0] a_array [0:1023][0:511];

8'b10110011 > 8'b0011	1
4'b1011 < 10	0
4'b1z10 < 4'b1100	x
4'b1x10 < 4'b1100	x
4'b1x10 <= 4'b1x10	x

- 逻辑相等：比较两个操作数的数值是否相等(==)或者不相等(!=)
  - ✓ 输出结果为1比特的0, 1, 或 x
  - ✓ 如果操作数中有X或者Z，导致无法判断时，输出为X
- 情形相等：比较两个操作数的逻辑值是否相等(===)或不相等(!==)
  - ✓ 逻辑值0，1，x或z进行比较
  - ✓ 输出为0或者1

Example			Results in
8'b10110011	==	8'b10110011	1
8'b1011	==	8'b00001011	1
4'b1100	==	4'b1Z10	0
4'b1100	!=	8'b100X	1
8'b1011	!==	8'b00001011	0
8'b101X	===	8'b101X	1

布尔运算符

运算符类型	运算符符号	操作	操作数数量
逻辑运算符	&&	逻辑与	2
	||	逻辑或	2
	!	逻辑非	1
比特运算符	&	与	2
	|	或	2
	~	取反	1
	^	异或	2
	^~ 或 ~^	异或非	2
缩位运算符	&	与	1
	~&	与非	1
	|	或	1
	~|	或非	1
	^	异或	1
	^~ 或 ~^	异或非	1

拼接运算符

拼接运算符	操作	说明	结果
位连接	{ }	比特连接	多比特
复制	{{ }}	连接并复制	多比特

- 并置运算符用位宽较小的矢量或标量构成新的矢量
- 例如：
  - ✓ a是4比特的1101，aa是6比特的001001

    {a, aa} = 10'b1101_001001
    {aa, {2{a}}, 2'b11} = 16'b001001_1101_1101_11

  - ✓ {a, 2{b,c}, 3{d}} = {a, b, c, b, c, d, d, d}
  - ✓ {2'b00, 3{2'01}, 2'b11} = 10'b00_010101_11

如果 *expression1* 的值为 X 或 Z, 则计算 *expressions2* 和 *expression3* 的值，输出结果为两个表达式对应比特位的组合

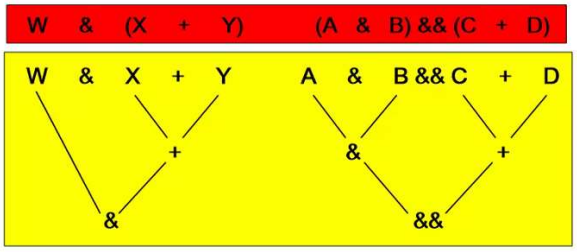
- ✓ 若对应比特都为0，则结果为0
- ✓ 若对应比特都为1，则结果为1
- ✓ 其余情况结果为X

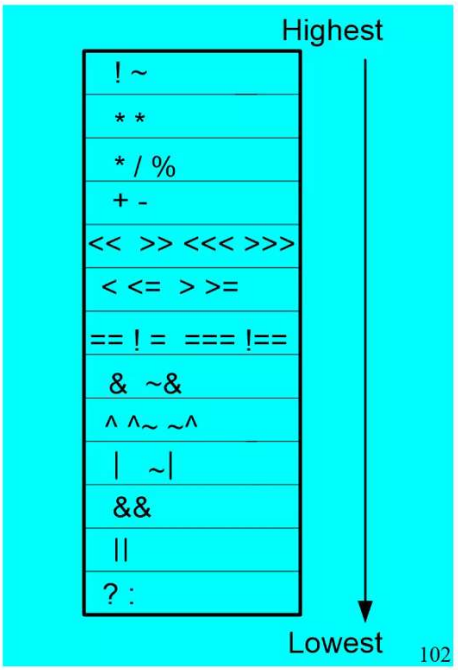
实例：

Example	Results in
1 ? 4'b1100 : 4'b1ZX0	4'b1100
0 ? 4'b1100 : 4'b1ZX0	4'b1ZX0
X ? 4'b1100 : 4'b1ZX0	4'b1XX0

运算符的优先级

- 在包含多种运算符的表达式里，操作的执行顺序是由运算符的优先级来决定的
- 小括号的优先级最高，推荐用小括号把各个操作按顺序括起来，不易出错





连续赋值语句

assign target = expression;

- 被赋值信号必须是线网类型变量，不能是寄存器类型
- 连续赋值语句一直处于激活状态

- 通过布尔表达式来描述电路
- 连续赋值语句和其他的连续赋值、门原语、行为描述语句、实例化模块等是并发执行的
- 连续赋值语句可以隐式的使用，即在声明变量时对其进行赋值（不用assign）
- 不能包含在initial块或always块中

- 用拼接运算符描述的全加器

```
`timescale 1ns/100ps

module add_1bit (input a, b, ci, output s, co);

    assign #(3, 4) {co, s} = {(a & b)|(b & ci)|(a & ci) , a^b^ci};

endmodule
```

用拼接运算符将标量拼成向量
注意向量的位置对应

- 连续赋值语句中对信号赋值时，在关键字后面用括号括起强度值
  - ✓ 一个强度值对应逻辑0：sypply0, strong0, pull0,weak0,...
  - ✓ 一个强度值对应逻辑1：sypply1, strong1, pull1,weak1,...
  - ✓ 它们在括号里的顺序不重要
- 可以用两种方式指定线网强度
  - ✓ 连续赋值语句中对信号赋值时指定

```
assign (strong0, strong1) w = m | p;
```

  - ✓ 在定义线网时指定

```
wand (pull0, supply1) sim;
```


- ✓ 线网的多驱动源强度不等，则选取其中强度级别最高的逻辑值

wand (pull0, supply1) sim;

线网的强度描述

强度可为 large, medium, 或 small, 默认为 medium

Wires (tri) wand (triand), Wor (trior), tri0, tri1		trireg	
Strength value	Level	Strength values	
Supply 0	7		Strength 0
Strong 0	6		
Pull 0	5		
	4	Large	
Weak 0	3		
	2	Medium (0)	
	1	Small (0)	
Highz 0	0		Weak values
Highz 1	0		
	1	Small (1)	
	2	Medium (1)	
Weak 1	3		
	4	Large (1)	
Pull 1	5		
Strong1	6		
Supply 1	7		Strength 1

默认为 strong

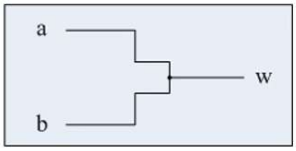
驱动强度的实例

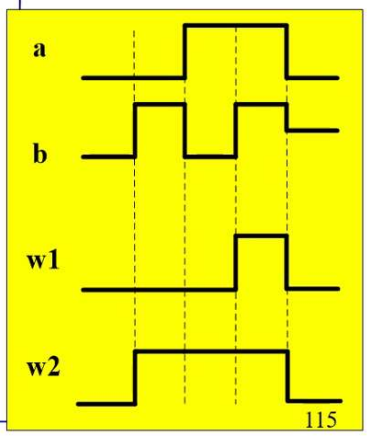
Case 1:

```
wire a, b;
wand w1;

assign w1=a;
assign w1=b;

initial begin
a=0;b=0;
#10 a=0;b=1;
#10 a=1;b=0;
#10 a=1;b=1;
#10 a=0;b=Z; end
```

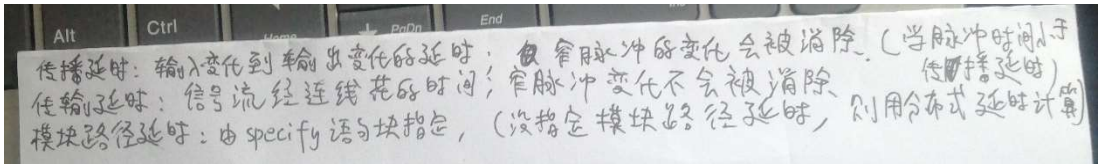
Case 2:

```
wire (pull0, supply1) a;
wire b, w2;

assign w2=a;
assign w2=b;

initial begin
a=0;b=0;
#10 a=0;b=1;
#10 a=1;b=0;
#10 a=1;b=1;
#10 a=0;b=Z; end
```





- 在一个module中，可用下述方式描述一个电路的功能（或称建立一个电路的模型）：
  - ✓数据流方式
  - ✓行为方式
  - ✓结构方式
  - ✓上述描述方式的组合



- 引用时，标明原模块定义时规定的端口名，端口顺序可以与模块定义时不一致（推荐）

Design u1(.端口1(u1的端口1), .端口2(u1的端口2), .端口3(u1的端口3),
……);

```
例4.3：  Add_half_0_delay M1 (.b (b),
                              .c_out(w2),
                              .a (a),
                              .sum (w1));
```

实际名称

形式名称

A<B

B1B0 \ A1A0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

A>B

B1B0 \ A1A0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

A=B

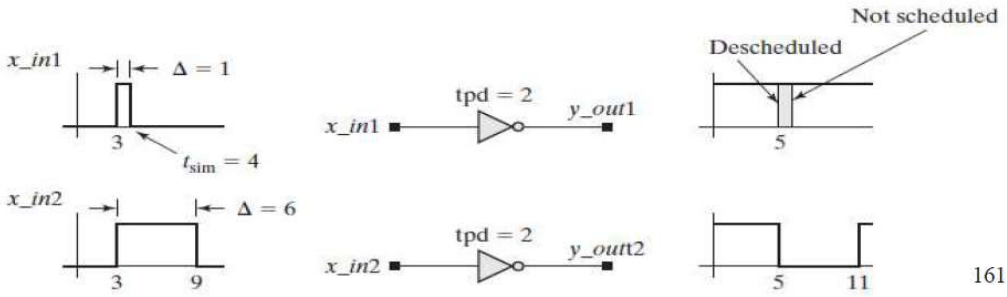
B1B0 \ A1A0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

A_lt_B = A1'B1 + A1'A0'B0 + A0'B1B0

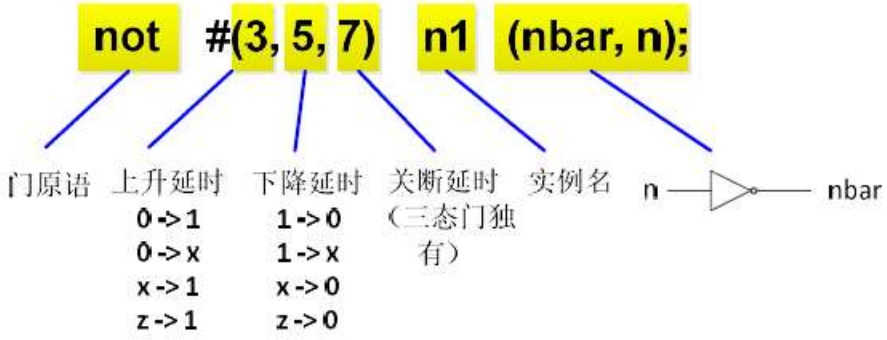
A_gt_B = B1'A1 + B1'B0'A0 + B0'A1A0

A_eq_B = A1'A0'B1'B0' + A1'A0B1'B0 + A1A0B1B0 + A1A0'B1B0'

- Verilog将门的传播延时作为惯性延时，即影响输出的输入脉冲的最小宽度
- 若输入脉冲宽度小于惯性延时，则门的输出不受影响



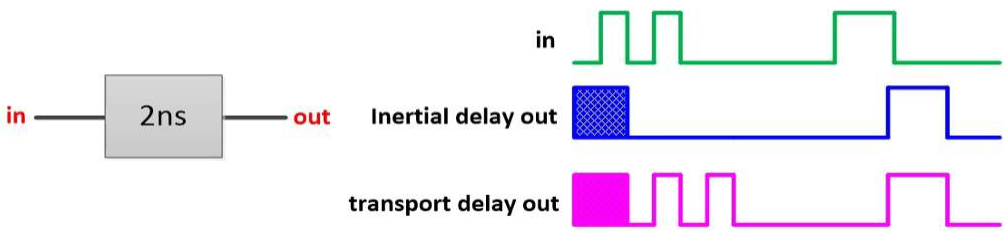
161



□ 用延时表达式来表示延时值，格式为：min:typ:max

- buf #(4,5) b3(e, f); // Rise=4, fall=5
- bufif1 #(4,5,5) b4(g,h,ctrl) //Rise=4, fall=5,turnoff=5
- bufif1 #(3:4:5, 4:5:6, 4:5:6) b5 (c, a, b); // min-typ-max

- 传输延时：信号流经连线所花的时间
- 窄脉冲不会被抑制



模块路径延时

- 模块路径延时可以和门级分布延迟同时存在
- 模块路径延时只有在其比所有内部分布延时之和大时，才会影响输出时序
- 不指定模块路径延时时，默认引脚到引脚的延时值为0，此时用分布延迟计算模块里的路径延时
- 采用 => 而非 *> 表示从源向量到目的向量的延迟
- 例子：(a=>s) = 12
  - ✓ 表示在a的某一位上发生的事件传输到s对应位上的延迟为12ns
  - ✓ 若a[1]的变化导致s[1]和s[3]都变化，则12ns的延迟仅用于s[1]，而对s[3]采用分布式延迟

- 常用的验证方法
  - 逻辑仿真：把激励波形加到电路上，监视器仿真特性，从而确定电路逻辑功能是否正确
  - 形式验证：在不施加激励的情况下，通过复杂的数学论证来证明电路的功能

Verilog测试平台：

- 是一个Verilog模块
- 对待测模块实例化
- 给待测模块施加激励
- 观测待测模块的输出

```
initial begin
        a=1'b1;
    #10 a=1'b0;
    #10 a=1'b1;
    #10 a=1'b0;
    #10 a=1'b1;
    ...
    $finish;
end
```

```
initial begin
   a=1'b0;
   $forever #5 a=!a;
end
```

```
initial begin
    $readmemb("din.txt", mem);
    for (i=0; i<=99; i=i+1)
      #10 a = mem[i];
end
```


仿真

Verilog Simulation with a Testbench

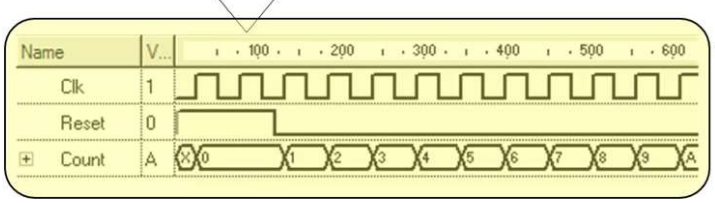
```
`timescale 1 ns / 100 ps
module Chap1CounterTester ();
  reg Clk=0, Reset=0;
  wire [3:0] Count;
  initial begin
    Reset = 0; #5 Reset = 1; #115 Reset = 0;
    #760 $stop;
  end
  always #26.5 Clk = ~ Clk;
  Chap1Counter U1 (Clk, Reset, Count);
endmodule
```

```
module Chap1Counter (Clk, Reset, Count);
  input Clk, Reset;
  output [3:0] Count;
  reg [3:0] Count;
  always @(posedge Clk) begin
    if (Reset) Count = 0;
    else Count = Count + 1;
  end
endmodule
```

Simulator

Testbench

Design to Simulate



The simulation results in form of a waveform

例：半加器的testbench

```
`timescale 1ns / 1ns

module t_add_half();
  wire sum, c_out;
  reg a, b;
  add_half_0_delay  M1 (sum, c_out, a, b);  // UUT
  initial begin   // time out
    #100 $finish;  //system task
  end
  initial   begin     //stimulus
    #10 a=0;b=0;
    #10 b=1;
    #10 a=1;
    #10 b=0;
  end
endmodule
```

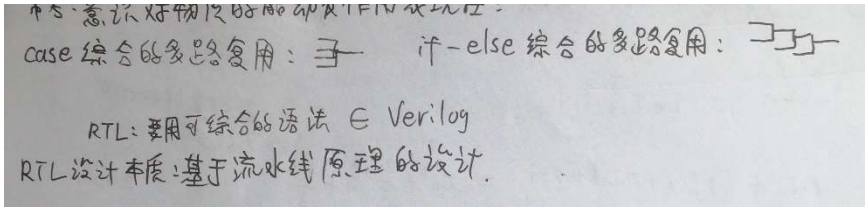
通过仿真器的波形窗口对输出波形和期望值进行比较

输入激励不是由硬件电路产生的，而是由Verilog行为级描述产生的

- 时间精度time_precision不能大于时间单位time_unit
- 时间单位和精度的表示方法：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
- 仿真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，即每隔precision定义的时间就要扫描一次
  - 若时间单位与时间精度差别很大将严重影响仿真速度。
  - 比如定义一个`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ns

Testbench Template

```
module t_DUTB_name ();                  //testbench模块名
    reg … ;                             //定义UUT的输入端口
                                        //输入端口在过程块中赋值，必须为reg类型
    wire … ;                            //定义UUT的输出端口
                                        //实例模块的输出，必须为wire类型
    parameter  time_out=…               //参数化仿真时间
    DUT_name M1_instance (DUT ports go here);
    initial  $monitor ();               //监视输出信号
    initial  #time_out  $finish;        //仿真结束控制
    initial                             //生成输入激励的行为级描述
      begin
      end
endmodule
```



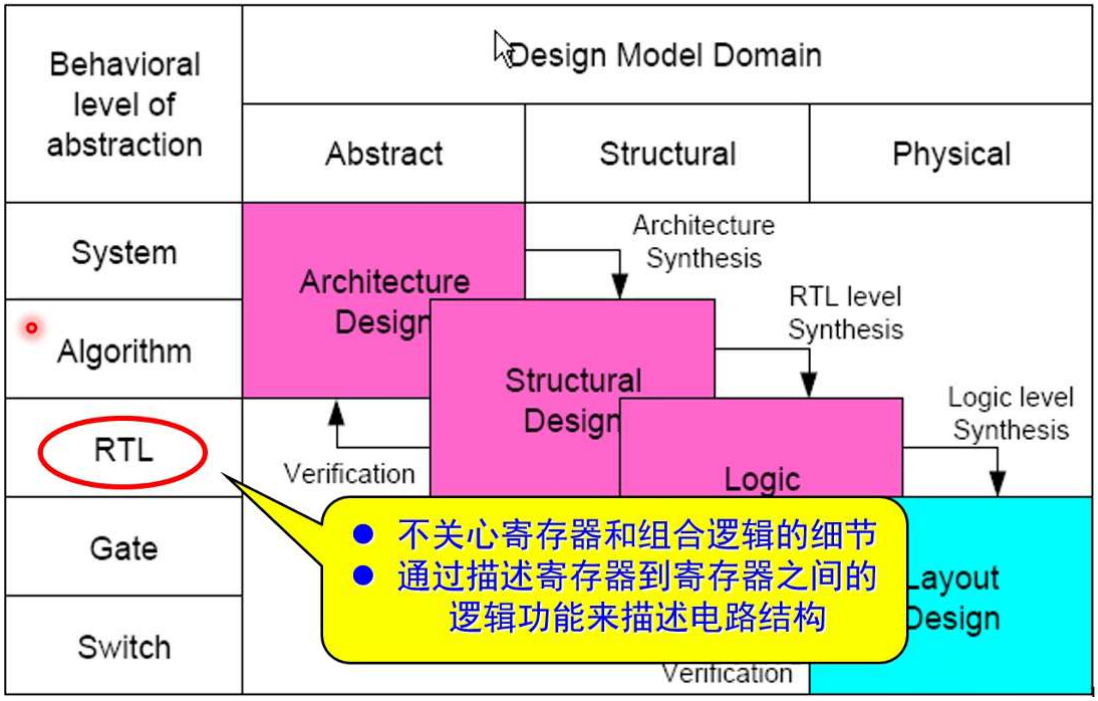
- 数据通路模块

```
module DataPath
   (DataInput, DataOutput, Flags, Opcodes,
   ControlSignals);

   input    [15:0] DataInputs;
   output   [15:0] DataOutputs;
   output   Flags, ...;
   output   Opcodes, ...;
   input    ControlSignals, ...;
   // instantiation of data components
   // ...
   // interconnection of data components
   // bussing specification
endmodule
```

- 控制单元模块

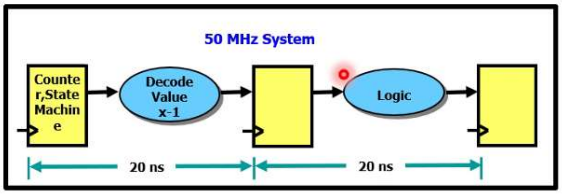
```
module ControlUnit
   (Flags, Opcodes, ExternalControls, ControlSignals);

   input  Flags, ...;
   input  Opcodes, ...;
   input  ExternalControls, ...;
   output ControlSignals;
   // Based on inputs decide :
   // What control signals to issue,
   // and what next state to take
endmodule
```

- 寄存器传输级设计是：（1）用高层次的语法来描述一个设计，（2）根据系统的带宽、时序指标等性能要求将整个设计分解成数个小模块，（3）用总线将小模块互联起来，（4）描述和实现这些小模块的设计方法



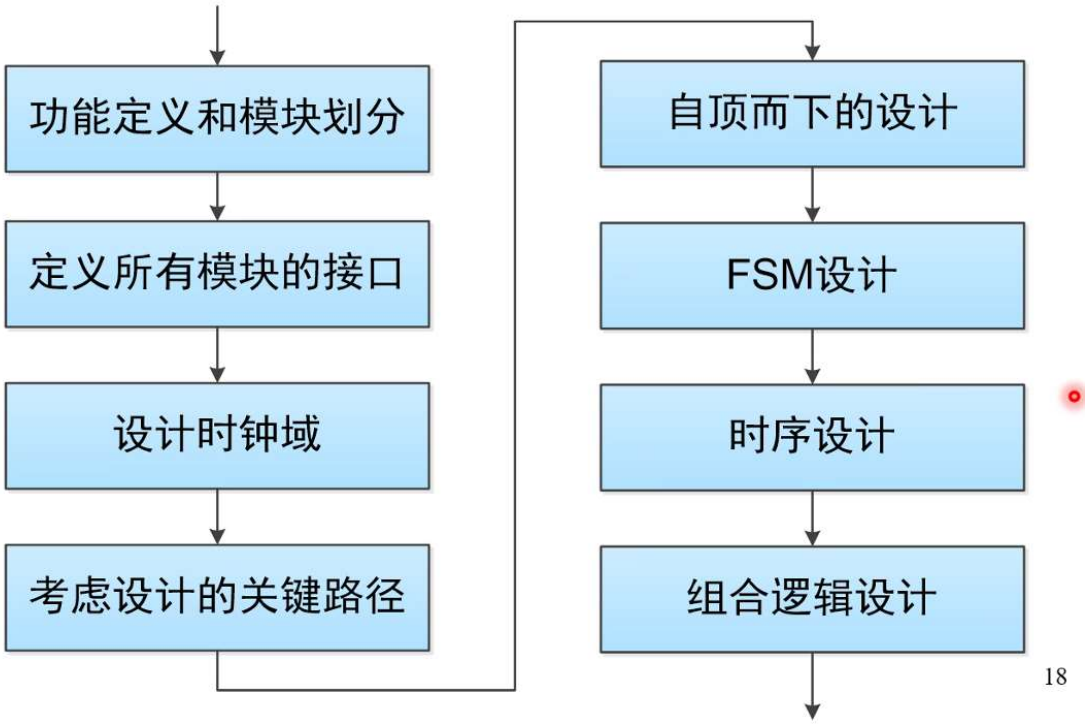
流水（Pipelining）

- 常用的时序优化手段
- 有目的的在关键链路上的组合逻辑之间插入寄存器
  - 缩短关键链路延时
  - 提高时钟频率
  - 增加了延时的级数



- EDA工具的智能化：在综合过程中能根据时序要求进行自动的流水插入

RTL级的设计步骤



RTL设计 VS 行为级设计

- RTL设计和行为级设计处于设计的不同阶段
  - 行为级：基于算法的高层次抽象描述；较多采用直接赋值的形式，看不出数据流的实际处理过程
  - RTL级：从寄存器角度，把数据的处理过程表达出来
- 设计目的
  - 行为级：关注算法，加快仿真速度，常用于仿真验证
  - RTL级：设计可综合代码
- 是否可综合
  - 行为级：部分不可综合
  - RTL级：可综合

乘法器实现用竖式乘法的方法实现（第二个乘数每一比特与第一个乘数相乘后相加（还要注意进位））

锁存器：电平敏感的存储元件；寄存器：上升沿敏感的存储元件。

建立时间是要求输入数据在上升沿到来前保持稳定的时间，所以$setup 先是输入数据，再是上升沿，最后时间；相反，保持时间相反，先上升沿，在输入数据。

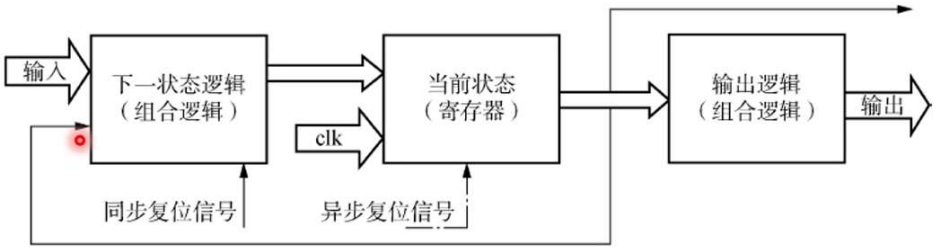
建立时间（Setup Time）

- 建立时间
  - ✓输入数据在送入触发器之前必须保持稳定的最小时间
- 用Verilog的系统任务$setup来检测建立时间
  - ✓$setup任务的参数包括：触发器的数据输入、有效时钟沿和要求的建立时间
  - ✓$setup任务用在specify内

例如：

```
$setuphold  (posedge clk,  d,  5,  3);
```

Moore型状态机



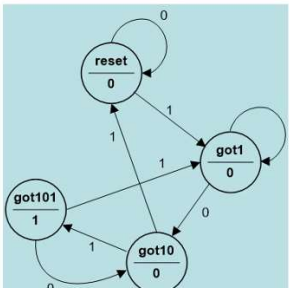
- 所有输出与电路时钟全同步
- 状态图中，每个状态的输出独立于电路的输入
- Verilog代码中，输出表达式中仅含有电路状态变量

Moore型状态机

```
`timescale 1ns/100ps

module moore_detector
      (in
localpar
                  got10=2,
                  got101=3;
 reg [1:0] current;
always @( posedge clk ) begin
  if( rst ) current <= reset;
  else case ( current )
    reset: begin
        if( x==1'b1 ) current <= got1;
        else current <= reset;  end
    got1: begin
        if( x==1'b0 ) current <= got10;
        else current <= got1;  end
```

always 块用于处理状态转换，即产生状态机的current状态.

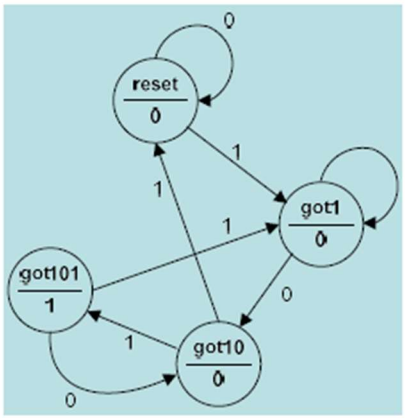
```
    got10: begin
        if( x==1'b1 ) current <= got101;
        else current <= reset;  end
    got101: begin
        if( x==1'b1 ) current <= got1;
        else current <= got10; end
    default: begin
      current <= reset; end
  endcase  end
 assign z = (current==got101) ? 1 : 0;

endmodule
```

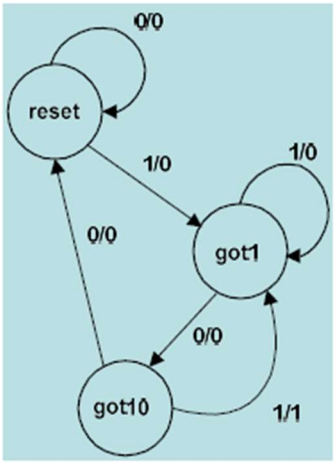


状态机可以采用三种描述方式：

- 状态转移图
- 状态转移列表
- 硬件描述语言

- 硬件描述语言





① Moore 机：输出只与状态有关，所以写在状态里，状态会多，因为最后一个状态可以不写

② Mealy 机：输出与状态和输入有关，所以与输入写在一起。

- wire、reg变量：用于对FSM中各变量进行声明
- parameter：用于编码FSM中的各种状态。状态编码主要有三种：顺序编码、格雷码和独热码
- always过程块：描述组合逻辑与时序逻辑
- case语句：状态操作
- if…else语句：状态转移或生成输出
- task或function：对一些重复使用的逻辑进行封装，提高代码可读性

设计同步状态机，异步状态机是不可综合的

- 三段式状态机的基本格式是：
  - 第一个always语句用组合逻辑生成下一状态（组合）
  - 第二个always语句实现输出（组合）
  - 第三个always语句实现同步状态跳转（时序）

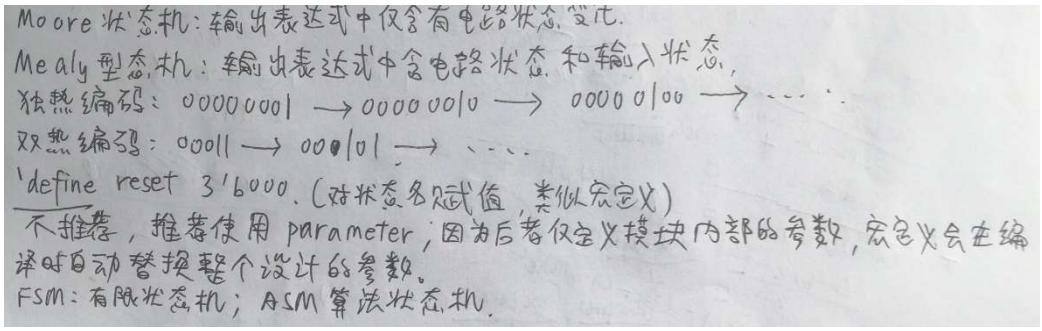
基于ROM的Mealy状态机

```
module mealy_detector7 (input x, clk, rst, output z);
  localparam [1:0]
    reset=2'b00, got1=2'b01, got10=2'b10, got11=2'b11;
  reg [1:0] p_state;
  wire [1:0] n_state;
  reg [2:0] mem[0:7];
  initial
    $readmemb( "mealy.dat", mem );

  assign { z, n_state } = mem[{ x, p_state }];

  always @( posedge clk ) begin:s    uential
    if( rst ) p_state <= reset;
    else p_state <= n_state;
  end
endmodule
```

将mealy.dat的内容以二进制形式载入mem

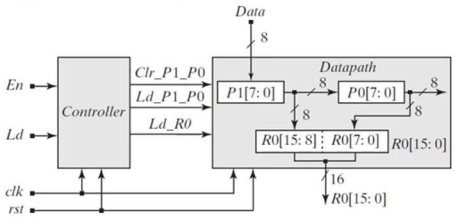
存储器的任一地址上的内容是电路输出与状态机下一状态的并置

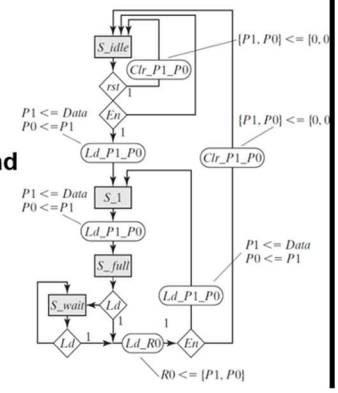
mem的地址由输入x和p_state并置构成



两级流水线寄存器的控制器的Verilog设计

```
module Controller(output reg Clr_P1_P0, Ld_P1_P0,Ld_R0, input En, Ld, clk, rst);
 parameter S_idle = 2'b00, S_1 = 2'b01, S_full = 2'b10, S_wait = 2'b11;
 reg [1: 0] state, next_state;
   always @ (posedge clk)
     if (rst) state <= S_idle;
     else state <= next_state;
   always @ (state, En, Ld) begin
     Clr_P1_P0 = 0; Ld_P1_P0 = 0;
     Ld_R0 = 0;
     case (state)
       S_idle: if (En) begin next_state = S_1; Ld_P1_P0 = 1; end
               else next_state = S_idle;
       S_1: begin next_state = S_full; Ld_P1_P0 = 1; end
       S_full: if (!Ld) next_state = S_wait; else begin Ld_R0 = 1;
               if (En) begin next_state = S_1; Ld_P1_P0 = 1; end
              else begin next_state = S_idle; Clr_P1_P0 = 1; end end
       S_wait: if (!Ld) next_state = S_wait;
                else begin Ld_R0 = 1;
            if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
            else begin next_state = S_idle; Clr_P1_P0 = 1; end  end
     endcase end
endmodule
```





方块：状态；椭圆：控制信号，标注有具体操作；菱形：条件，1 是指向满足时的下一状态。

综合工具的工作步骤：

- 检测并消除冗余逻辑
- 查找组合反馈环路
- 利用无关紧要条件
- 检测出未使用的状态
- 查找并消除等价的状态
- 进行状态分配
- 在满足物理约束条件下，综合出最优逻辑结构

综合工具：排除错误，化简结构。

电路模型的抽象级别：

– 行为级：算法操作

– 结构级：构成数据通路的单元

– 物理级：物理参数

- **Logic synthesis**

  = Translation + Logic optimization + Mapping

逻辑综合（将代码变成门级电路）=翻译（翻译成两级（最大项之积和最小项之和（和之积，积之和））的结构）+逻辑优化+映射/绘图

综合算法中的关键变换

① 分解：Decomposition
② 因式分解：Factoring
③ 替代：Substitution
④ 消去：Elimination

➢ 考虑因素:

Types of component,
Cost,
Delay,
Fan in/out

狄摩根定律
(a+b)'=a'b'
(ab)'=a'+b'

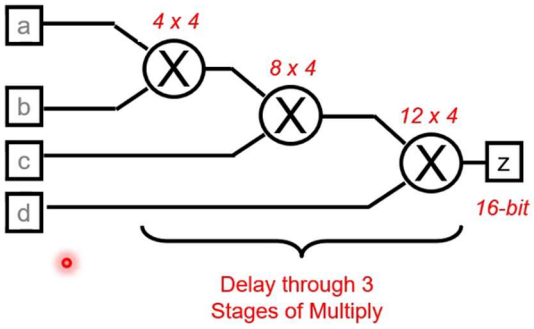
平衡运算符

- 用小括号进行逻辑分组
  - 不会改变电路本来的功能
  - 提高性能与利用率
  - 平衡所有输入-输出之间的延时

a, b, c, d: 4-bit vectors

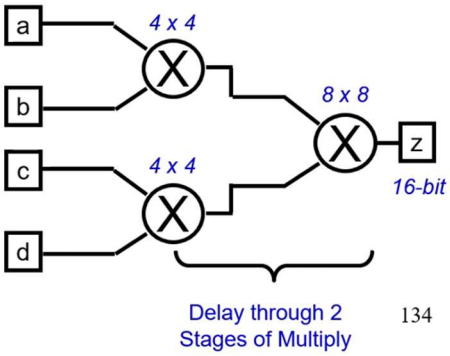
Unbalanced

out = a * b * c * d

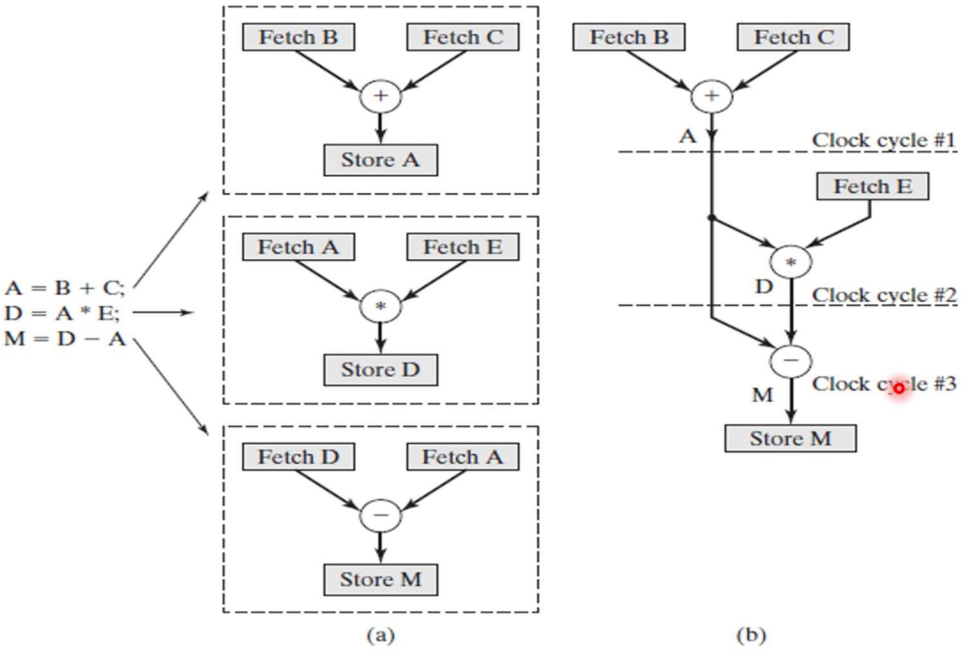


Balanced

out = (a * b) * (c * d)



行为级模型

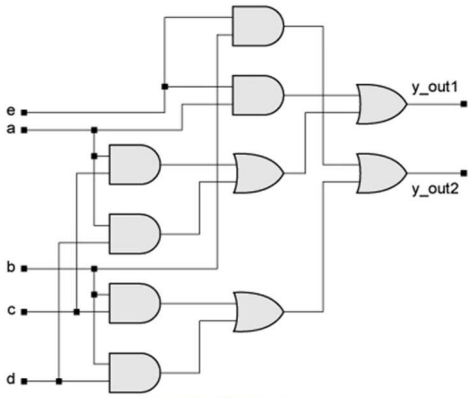


语义树　　数据流图

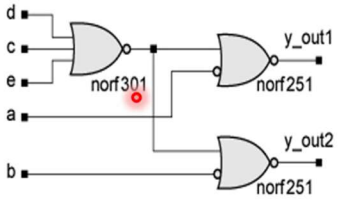
- 可综合的组合逻辑可以描述为：
  1. 结构化的基本门原语构成的网表
  2. 一系列连续赋值语句
  3. 电平敏感的过程块

例：对门原语网表的综合

```
module boole_opt(y_out1, y_out2,
   a, b, c, d, e);
   output y_out1, y_out2;
   input   a, b, c, d, e;
   and  (y1,a,c);
   and  (y2,a,d);
   and  (y3,a,e);
   or    (y4,y1,y2);
   or    (y_out1,y3,y4);
   and  (y5,b,c);
   and  (y6,b,d);
   and  (y7,b,e);
   or    (y8,y5,y6);
   or    (y_out2,y7,y8);
endmodule
```



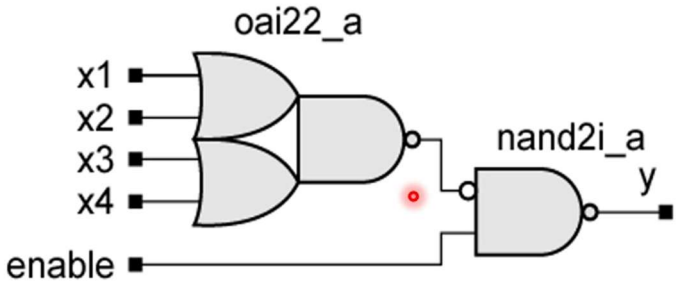
门原语网表



综合后网表

- 连续赋值语句是可综合的

```
module or_nand (output y, input enable,
                                  x1, x2, x3, x4);
    assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```



包含循环结构的电平敏感行为

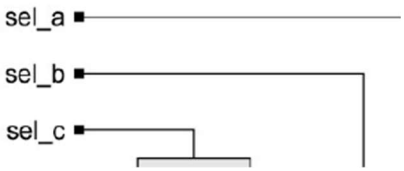
- 如果对输入的每个值都指定了相应的输出值，则包含循环结构的电平敏感行为可以综合成组合逻辑

条件语句的综合

- 如果不包含内嵌的时间控制语句（#，@或wait），if语句和case语句可综合成组合逻辑
- 条件语句一般综合成多路复用器

```
module mux_4pri
(output reg y, input a, b, c, d,
   sel_a, sel_b, sel_c);
always @ (sel_a or sel_b  or
   sel_c or a or b or c or d)
```

通配符：*



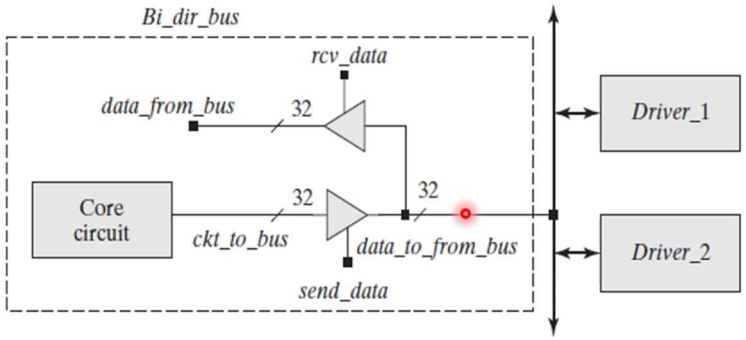
通配符：下面出现的所有输入信号

- 如果条件分支里有X或Z值，则仿真和综合的结果可能不一致
- 综合工具把casex和casez都认为是case语句
- 带有x或z值的条件选项会被当做无关条件进行处理
- 虽然仿真时传输x值，但实际硬件要么传输0，要么传输1

default: 输出=0/ default: 输出=x 区别：等于零时，卡诺图化简，即综合时没法优化；等于 x 可以优化，让它等于一或零。

综合时：不一定是要用低层门电路实现，通过通过同资源共享，库文件可以让综合结果稍微复杂（简洁）。

例：双向总线接口

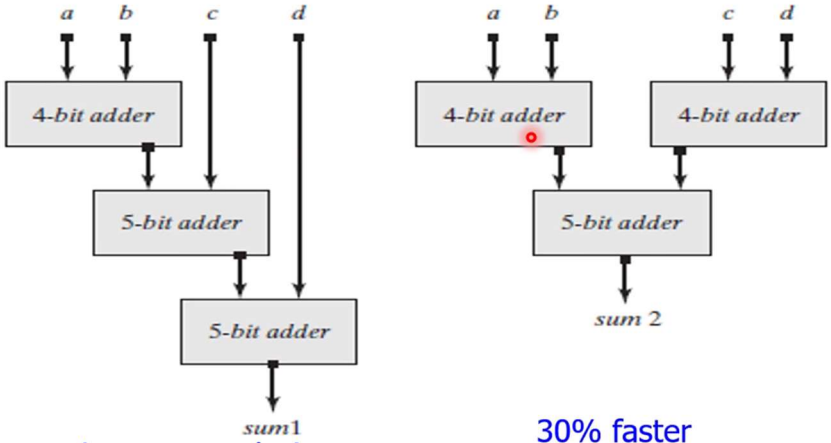



```
module Bi_dir_bus (inout [31: 0] data_to_from_bus, input send_data,
    rcv_data);
     wire [31: 0] ckt_to_bus;
     wire [31: 0] data_from_bus;
     assign data_from_bus = (rcv_data) ? data_to_from_bus : 32'bz;
     assign data_to_from_bus =
                          (send_data) ? ckt_to_bus :  data_to_from_bus;
```

send_data 不满足时不拉高阻，因为，上面有可能会通，不能因为自己不工作，而不让别人不工作。

例：由运算符分组得到的结构改进

```
module operator_group (output [4: 0] sum1, sum2, input a, b, c, d);
    assign sum1=a+b+c+d;
    assign sum2=(a+b)+(c+d);
endmodule
```



d 信号变化很快的时候，可以采用不分组的方式比较好。

综合工具会保留设计的输入与输出端口，但可能对内部连线进行优化

```
module multiple_reg_assign (output reg [4: 0] data_out1, data_out2,
        input [3: 0] data_a, data_b, data_c, data_d, input sel, clk);
      always @ (posedge clk) begin
        data_out1 = data_a + data_b ;
        data_out2 = data_out1 + data_c;
            if (sel == 1'b0)
            data_out1 = data_out2 + data_d;
      end
endmodule
```

表达式替代

```
module expression_sub (output reg [4: 0] data_out1, data_out2,
input [3: 0] data_a, data_b, data_c, data_d, input sel, clk);
      always @ (posedge clk) begin
         data_out2 = data_a + data_b + data_c;
           if (sel == 1'b0) data_out1 = data_a + data_b + data_c + data_d;
           else data_out1 = data_a + data_b;
      end
endmodule
```

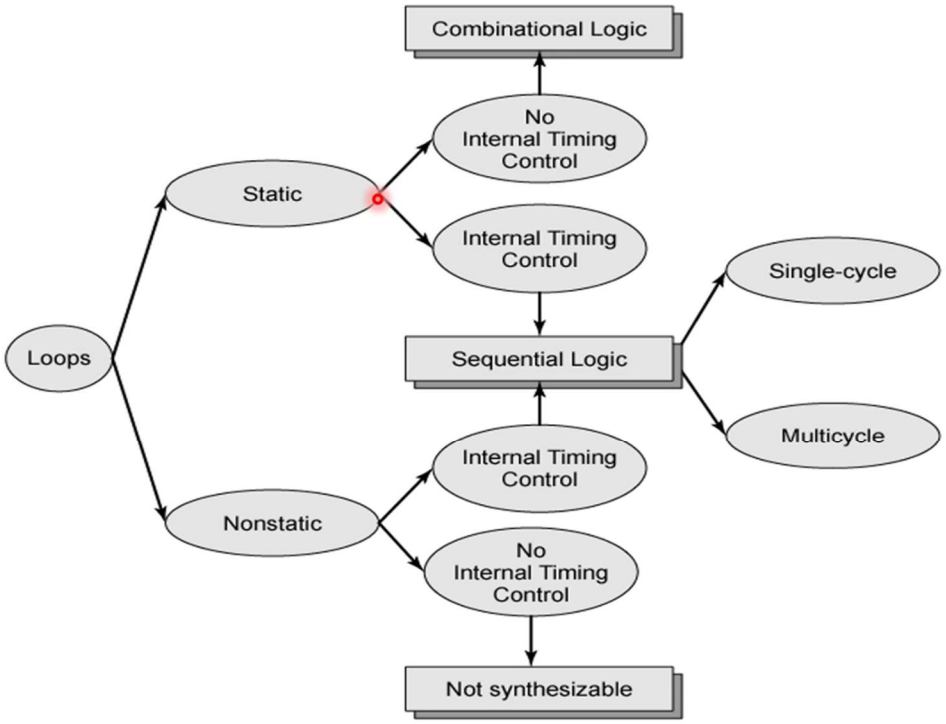
独立、并行的赋值

```
module expression_sub_nb (output reg [4: 0] data_out1nb, data_out2nb,
    input [3: 0] data_a, data_b, data_c, data_d, input sel, clk)  ;
  always @ (posedge clk) begin data_out2nb <= data_a + data_b + data_c;
     if (sel == 1'b0) data_out1nb <= data_a + data_b + data_c + data_d;
     else  data_out1nb <= data_a + data_b;
      end
endmodule
```


循环的综合

- 两类循环：
- **静态循环：数据独立**
  - 循环迭代次数在仿真前能由编译器确定，即迭代次数是固定的，并与数据无关
- **非静态循环：数据相关**
  - 循环迭代次数由运算中的某个变量决定
  - 具有数据依赖性的循环可能对内嵌的定时控制（如事件表达式）具有依赖性

always块中循环结构的分类

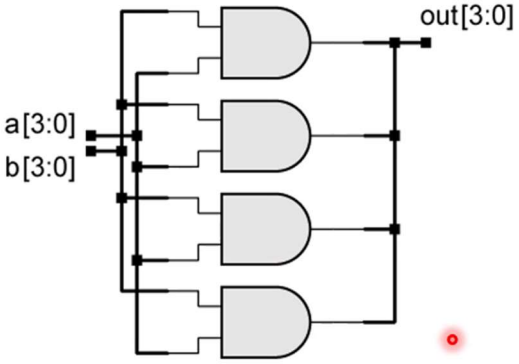


例：不依赖于数据的循环

```
module for_and_loop_comb (output reg [3: 0] out, input [3: 0] a, b);
    reg [2: 0] i;
    always @ (a or b) begin
       for (i=0;i<=3;i=i+1)
          out[i]=a[i] & b[i];
     end
endmodule
```

The unrolled loop is equivalent to the following assignments:

```
out[0] = a[0] & b[0];
out[1] = a[1] & b[1];
out[2] = a[2] & b[2];
out[3] = a[3] & b[3];
```



- 有固定范围的for循环与repeat循环的综合结果是相同的
- 一些综合工具只支持for循环

例：统计数据字中1的个数

```
module count_ones_b2 #( parameter data_width = 4, count_width = 3) (output
reg [count_width -1: 0] bit_count, Input [data_width -1: 0] data, input clk, reset);
    reg [count_width-1: 0] count;
    reg [data_width-1:0] temp;
    integer  index;
    always begin: machine
      for (index=0; index <= data_width; index=index+1) begin
        @ (posedge clk)
            if (reset) begin bit_count =0; disable machine; end
            else if (index==0) begin count = 0; bit_count=0; temp=data; end
            else if (index <data_width) begin
               count = count + temp[0];
                temp = temp >> 1; end
            else
                bit_count = count + temp[0];  end
     end // machine
endmodule
```


disable(跳出 always 过程块)：所以循环次数不确定。

带内嵌定时控制：always 在 for 内，必须要五个时钟周期才能完成操作，此期间 Data 变化也无用。

非静态的循环带内嵌操作也是综合成时序电路。

要避免的设计陷阱

- 避免在多个always语句中对同一个变量进行赋值
- 如果在多个always语句中对同一变量进行赋值，可能存在竞争，导致综合之后的结果与仿真结果不一致
- 因此必须将对同一变量的不同情况赋值整合到一个always语句块中
- 尽量一个always语句对一个变量赋值，不同变量的赋值放在不同的always块里面

例：用函数实现：数据字左移，直到最高位为1

```
module word_aligner #(parameter word_size = 8) (output
   [word_size -1: 0] word_out, input [word_size -1: 0] word_in);

   assign word_out = aligned_word (word_in);

   function [7:0] aligned_word;
    input   [7:0] word_in;
    begin
      aligned_word=word_in;
      if (aligned_word!=0)
       while (aligned_word[7]==0)
          aligned_word=aligned_word <<1;
    end
   endfunction
endmodule
```

194

函数只能返回一个值。

任务与函数的主要区别

比较项目	任务（task）	函数（function）
输入与输出	可有任意个不同类型的参数	至少有一个输入，不能将inout作为输出
定时控制	可以包含定时控制语句	不能包含定时控制语句
调用其他任务与函数	可以调用其他任务和函数	可以调用其他函数，但不能调用任务
调用	可以在过程语句中使用，不能在连续赋值语句中使用	可以在过程语句和连续赋值语句中使用
返回值	不返回值	返回一个值
综合结果	根据被调用的方式，既可以综合得到组合逻辑，也可以综合得到时序逻辑	组合逻辑

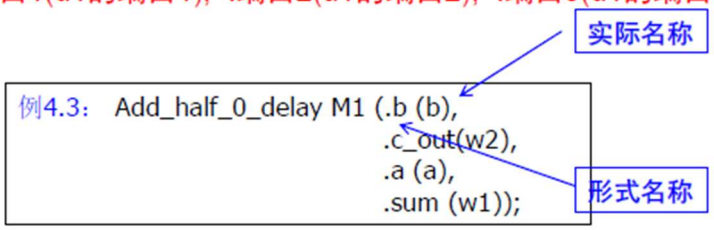
189

模块实例化的端口连接

两种方式：

- 引用时，标明原模块定义时规定的端口名，端口顺序可以与模块定义时不一致（推荐）

Design u1(.端口1(u1的端口1), .端口2(u1的端口2), .端口3(u1的端口3),
……);



- 引用时，不用标明原模块定义时规定的端口名，严格按照模块定义的端口顺序来连接

Design u2(u2的端口1，u2的端口2，u2的端口3，……)； //和Design对应

```
Add_half_0_delay M1 (w2, w1, a, b);
```