第5章 FPGA设计和硬件描述语言

第一讲 可编程逻辑器件和FPGA设计

第二讲 HDL概述

第三讲 Verilog语言简介

第四讲 Verilog建模方式

第五讲 Verilog代码实例

（除存储器基本概念外，本章其他内容不作要求）

第一讲 可编程逻辑器件和FPGA设计

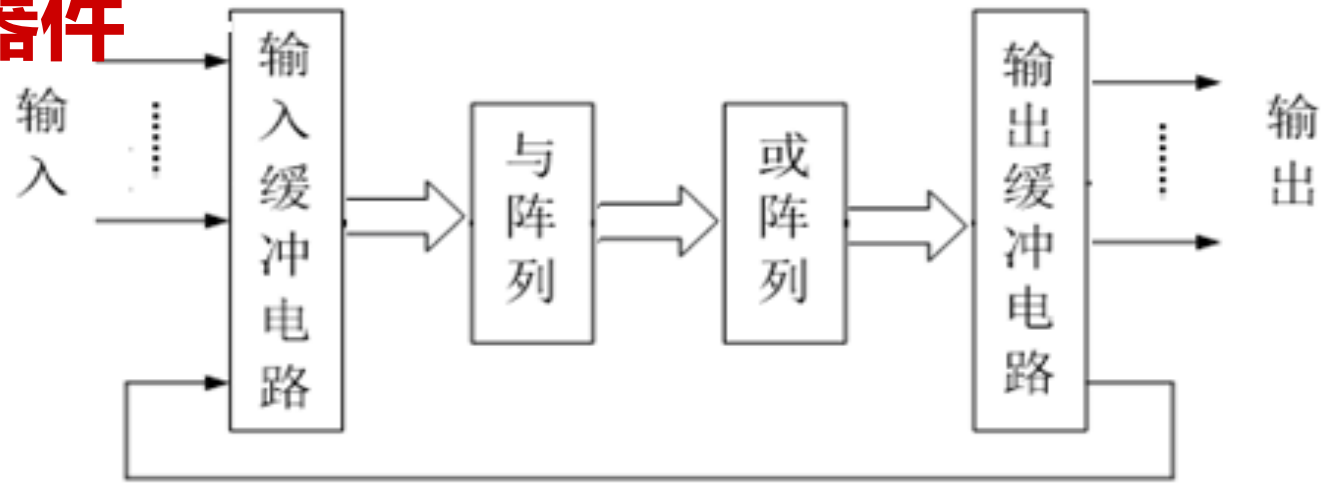
- 可编程逻辑器件
- 存储器阵列
- FPGA设计概述
- 专用集成电路

1 可编程逻辑器件和FPGA设计

- 固定逻辑标准芯片（如74系列芯片）曾被广泛使用
- 但固定逻辑芯片功能单一，不能随电路设计的需求而任意改变
- 固定逻辑芯片逐渐被可编程逻辑器件（Programmable Logic Device，PLD）取代
- PLD是一种用于实现逻辑电路的通用器件
- PLD包含多个逻辑单元，可根据需要通过编程开关连接进行编程，以构成不同功能的逻辑电路
- PLD的结构主要由与阵列和或阵列构成

1.1 PLD器件

- PLD结构框图

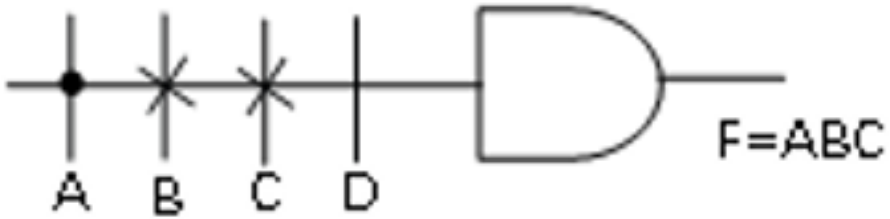


- PLD中基本电路符号

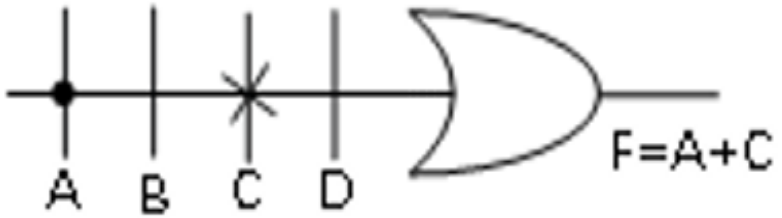
互补缓冲器

阵列连线

与阵列表示（与门仅示意与阵列）



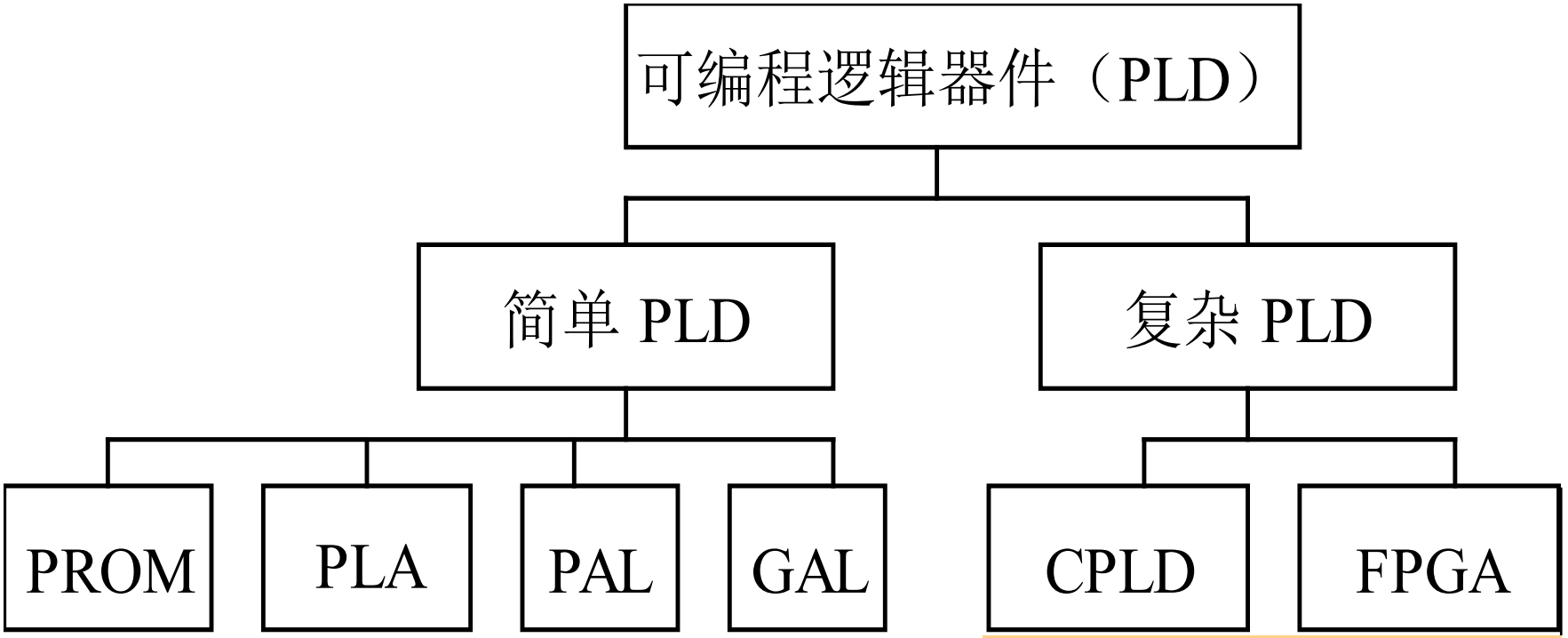
或阵列表示（或门仅示意或阵列）



1.1 PLD分类

简单PLD：逻辑门数在500门以下

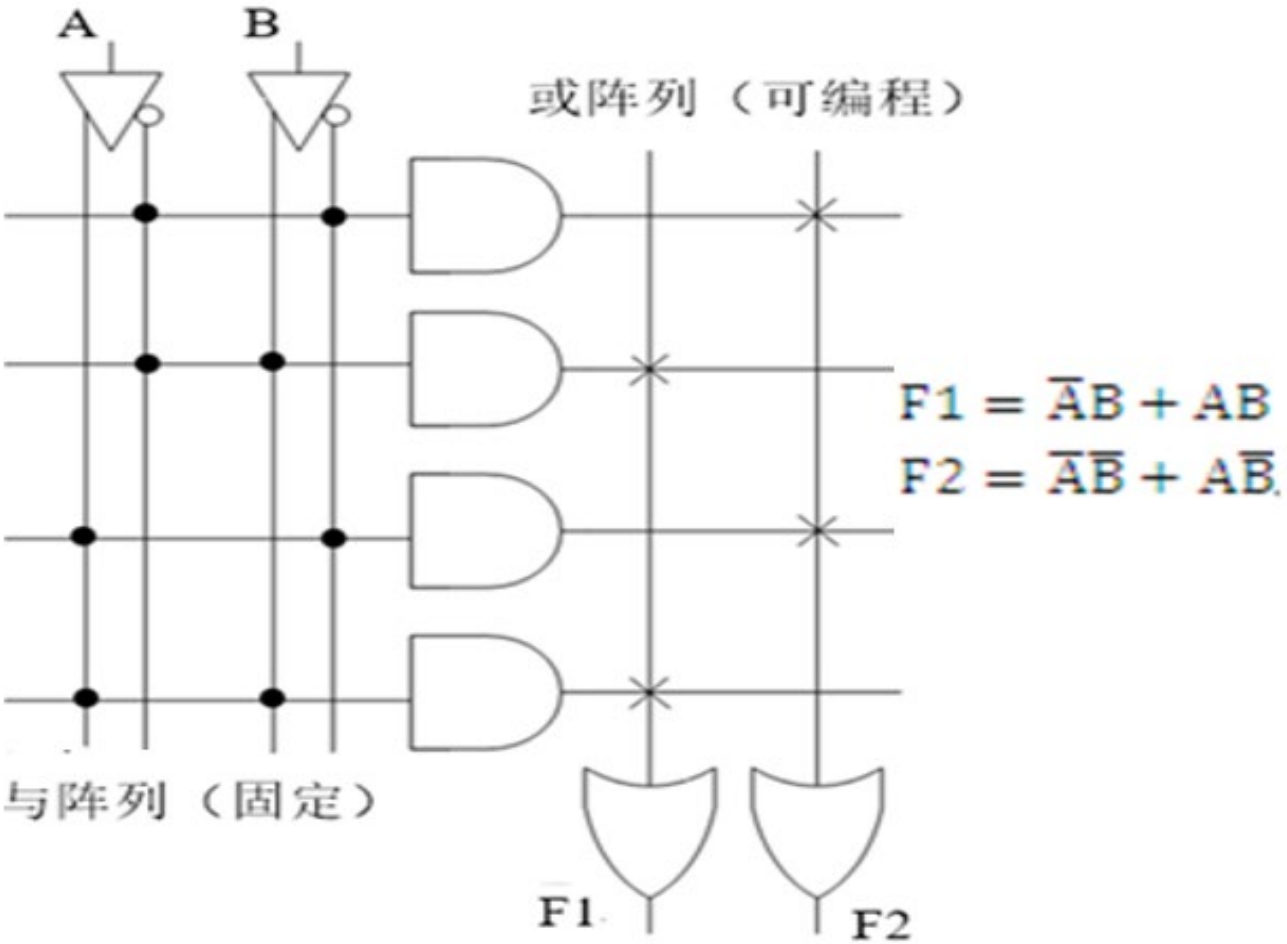
复杂PLD：集成度高，逻辑门数在500门以上



将以乘积项结构方式构成逻辑行为的器件称为CPLD。
将以查表法结构方式构成逻辑行为的器件称为FPGA

1.1 PLD器件——PROM

- 可编程只读存储器（Programmable Read Only Memory，PROM）是一种**与阵列固定**、**或阵列可编程**的简单PLD。

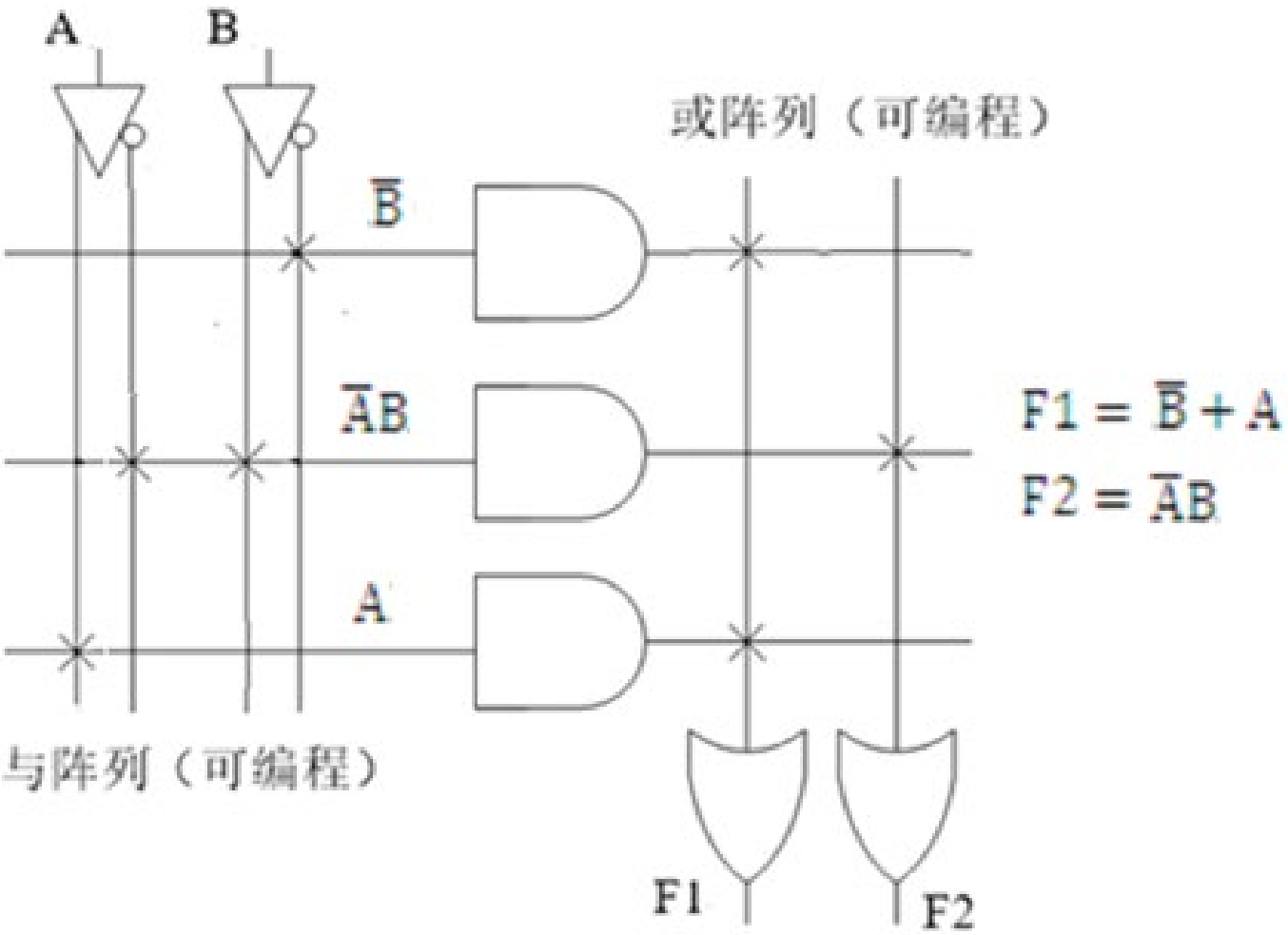


任何逻辑函数转换成标准与-或表达式后，可用PROM来实现

与阵列的水平线输出对应标准与-或表达式中的标准乘积项，即最小项。

1.1 PLD器件——PLA

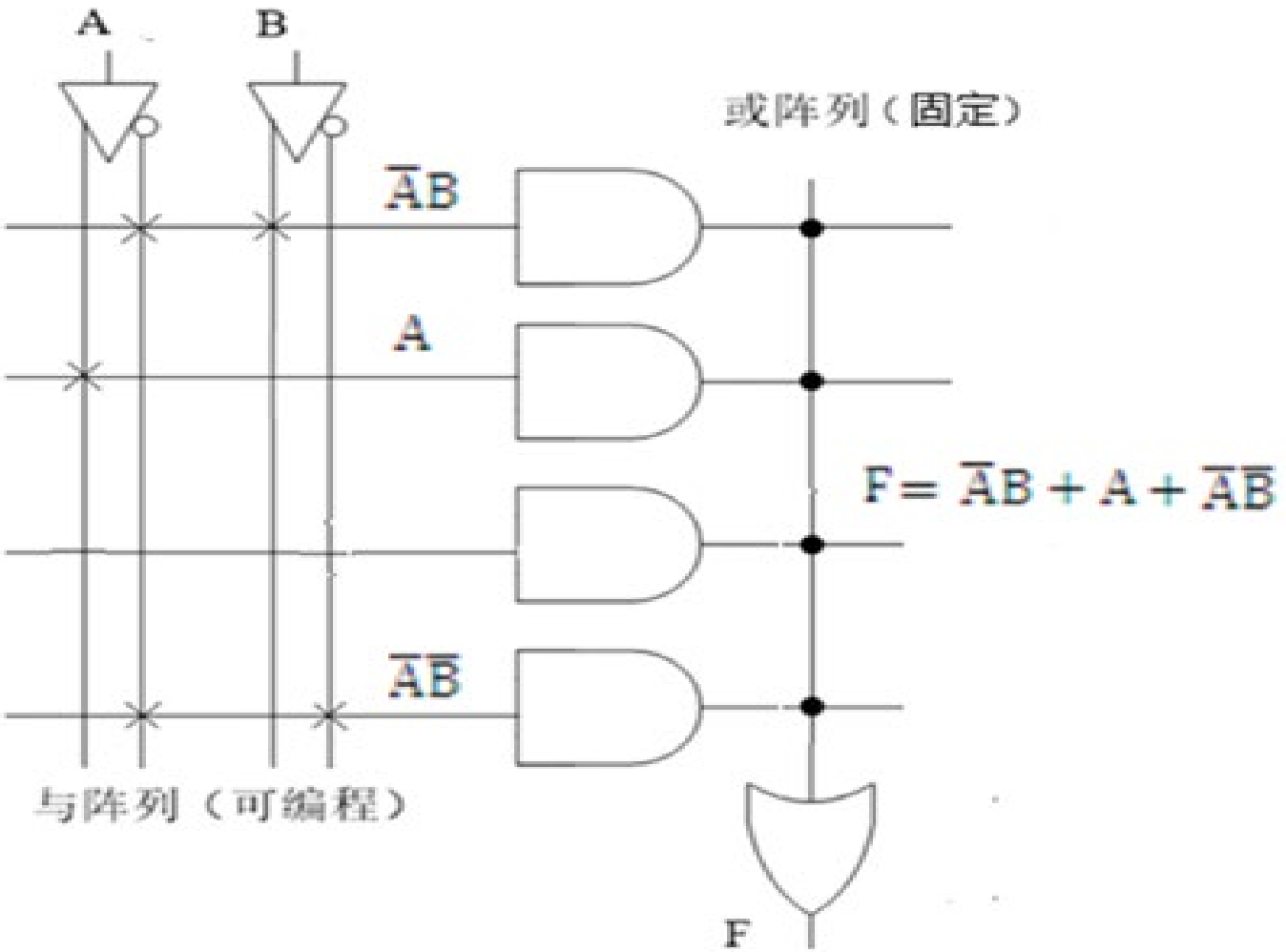
- 可编程逻辑阵列（Programmable Logic Array，PLA）是一种与阵列、或阵列**都可编程**的逻辑阵列。



无须像PROM那样将逻辑函数转换成标准与-或表达式，而只要化简成最简与-或表达式即可，可节省编程资源。

1.1 PLD器件——PAL

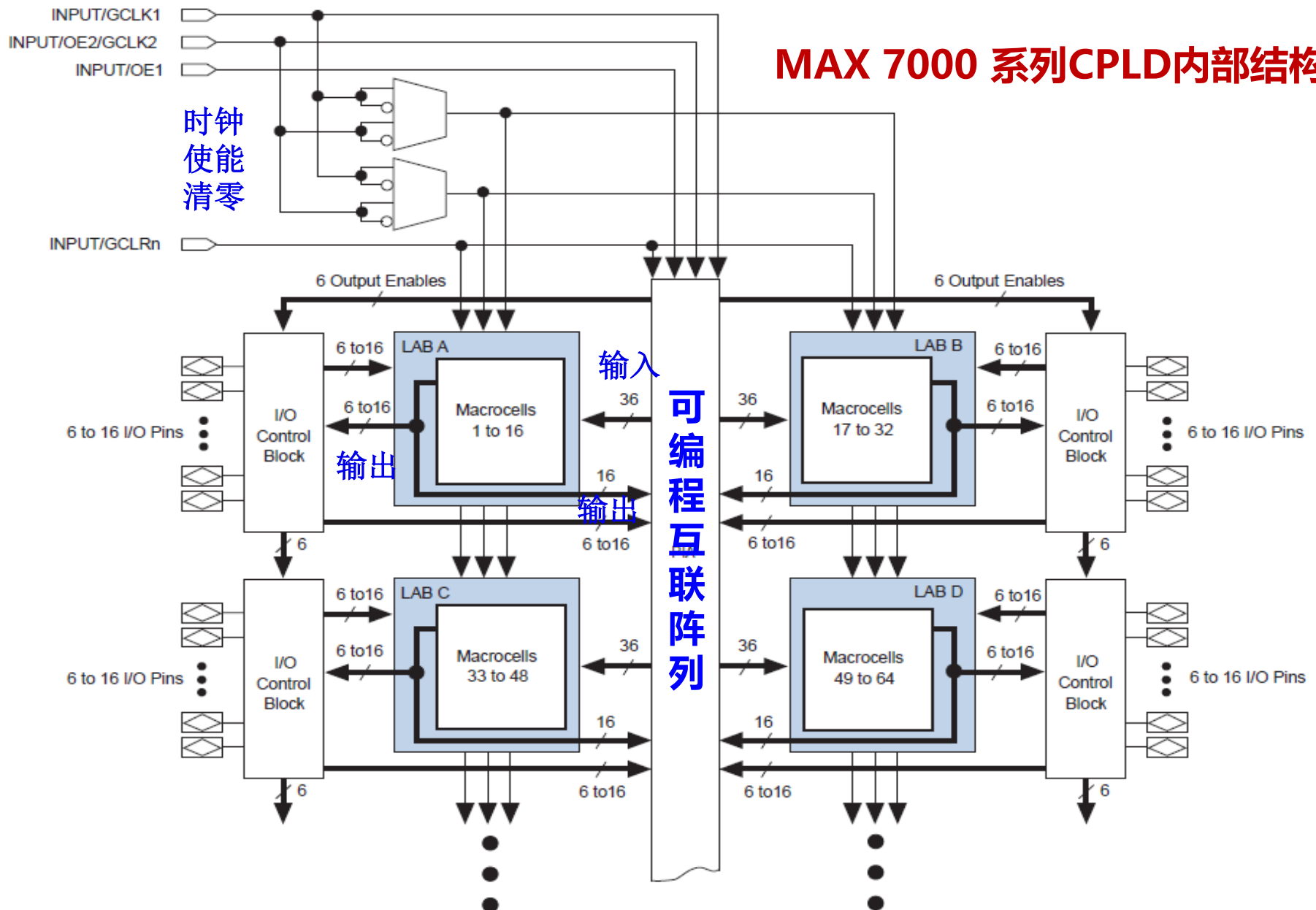
- 可编程阵列逻辑（Programmable Array Logic，PAL）是一种**与阵列可编程**、**或阵列固定**的逻辑阵列。



1.1 PLD器件——CPLD

- 复杂可编程逻辑器件（Complex PLD，CPLD）主要包括：
  - 逻辑阵列块（Logic Array Block，LAB）
  - I/O控制块：用于和芯片的I/O引脚互连
  - 可编程互联阵列（PIA）
- 每个LAB由4～20个宏单元（Macrocell）构成。宏单元包括：
  - 可编程逻辑阵列
  - 乘积项选择矩阵
  - 可编程寄存器：可以编程实现D、JK或钟控SR触发器等。
- 宏单元有多种配置方式，也可级联使用
- PIA用于连接所有宏单元，并与芯片时钟、复位、使能等引脚连
- CPLD集成度远高于PAL和GAL
- CPLD通常提供带片内RAM/ROM的嵌入式存储器阵列

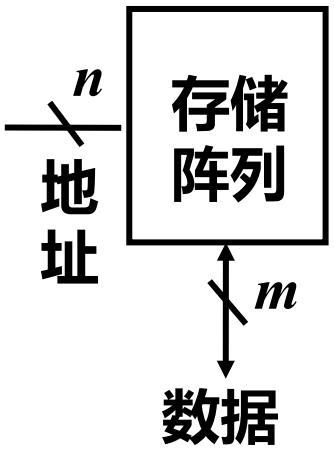
1.1 PLD器件——CPLD



MAX 7000 系列CPLD内部结构图

1.2 存储器阵列

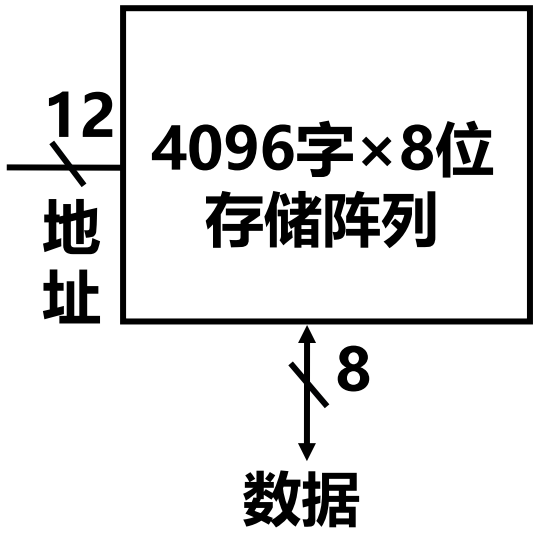
- 存储器可用来存储数字电路中的数据。
  - 寄存器（由触发器构成）用来存储少量数据，速度快
  - 存储器阵列用来存储大量数据，速度比寄存器慢
- 在CPLD和FPGA芯片中通常会提供片内存储器阵列
- 存储器阵列中每位数据对应一个记忆单元（cell），称为存储元



$2^n \times m$位存储阵列

000	0	1	0	0
001	0	0	1	1
010	0	1	0	1
011	1	0	1	1
100	1	1	0	0
101	1	0	1	1
110	0	1	1	0
111	1	0	0	0

8×4位存储器阵列



4KB存储器阵列

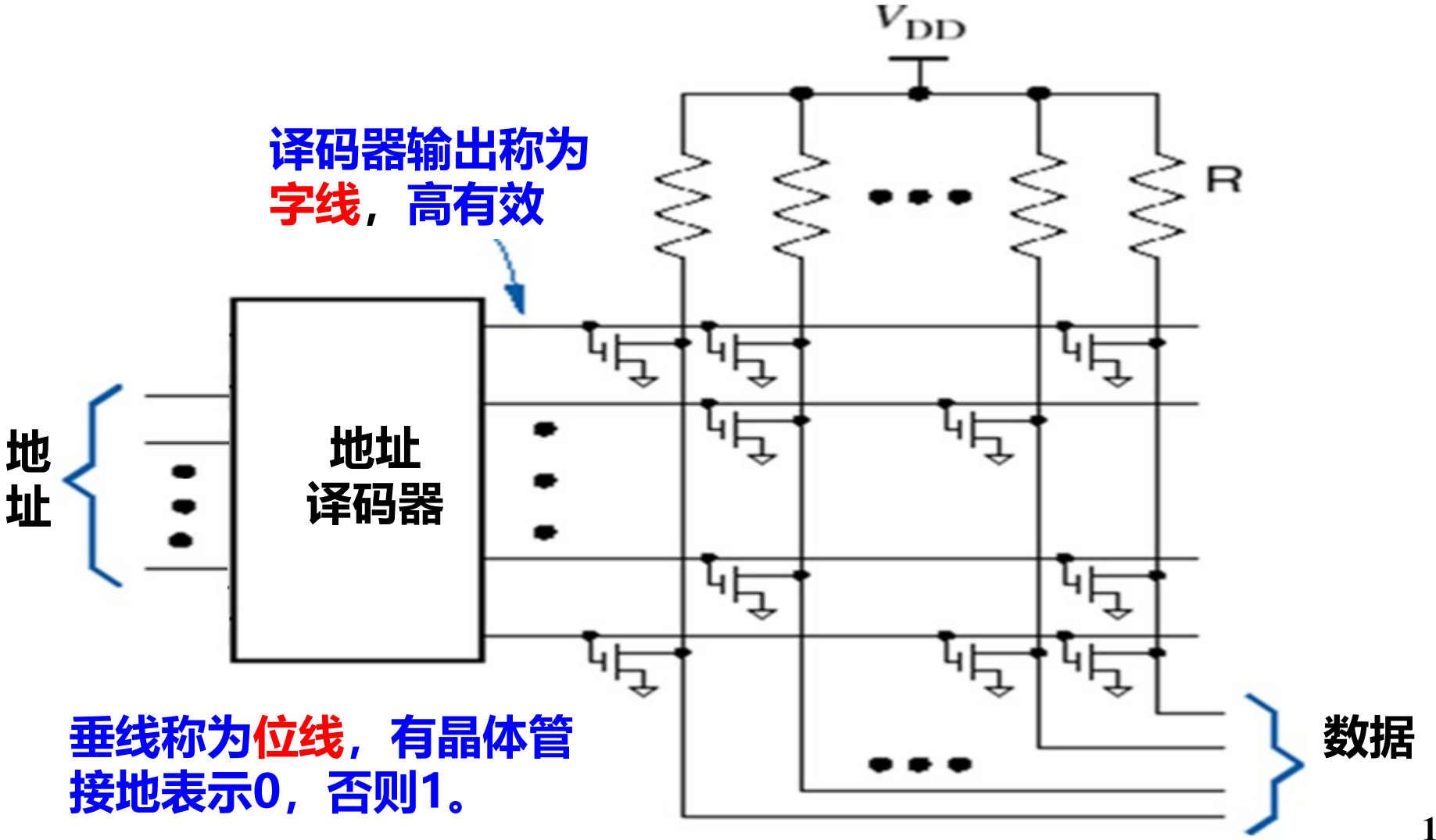
1.2 存储器阵列——ROM，RAM

- **按功能可分为：只读存储器(Read-only Memory，ROM)和随机存取存储器(Random-access Memory, RAM)**
  - **ROM属于非易失性存储器，即使电源断电，ROM中存储的数据也不会消失（例如存放BIOS自检启动程序的地方）。根据工艺的不同，分为：**
    - 掩膜只读存储器MROM（信息固定）
    - 一次可编程只读存储器PROM（半成品，可编程）
    - 光擦除可编程只读存储器EPROM（紫外线全部擦除）
    - 电擦除可编程只读存储器EEPROM（E^2PROM）

      （电擦除个别字，再重新改写）
  - **RAM属于易失性存储器，一旦电源断电，RAM中存储的数据就消失。**
    - **静态RAM（Static RAM，SRAM）**
    - **动态RAM（Dynamic RAM，DRAM）**

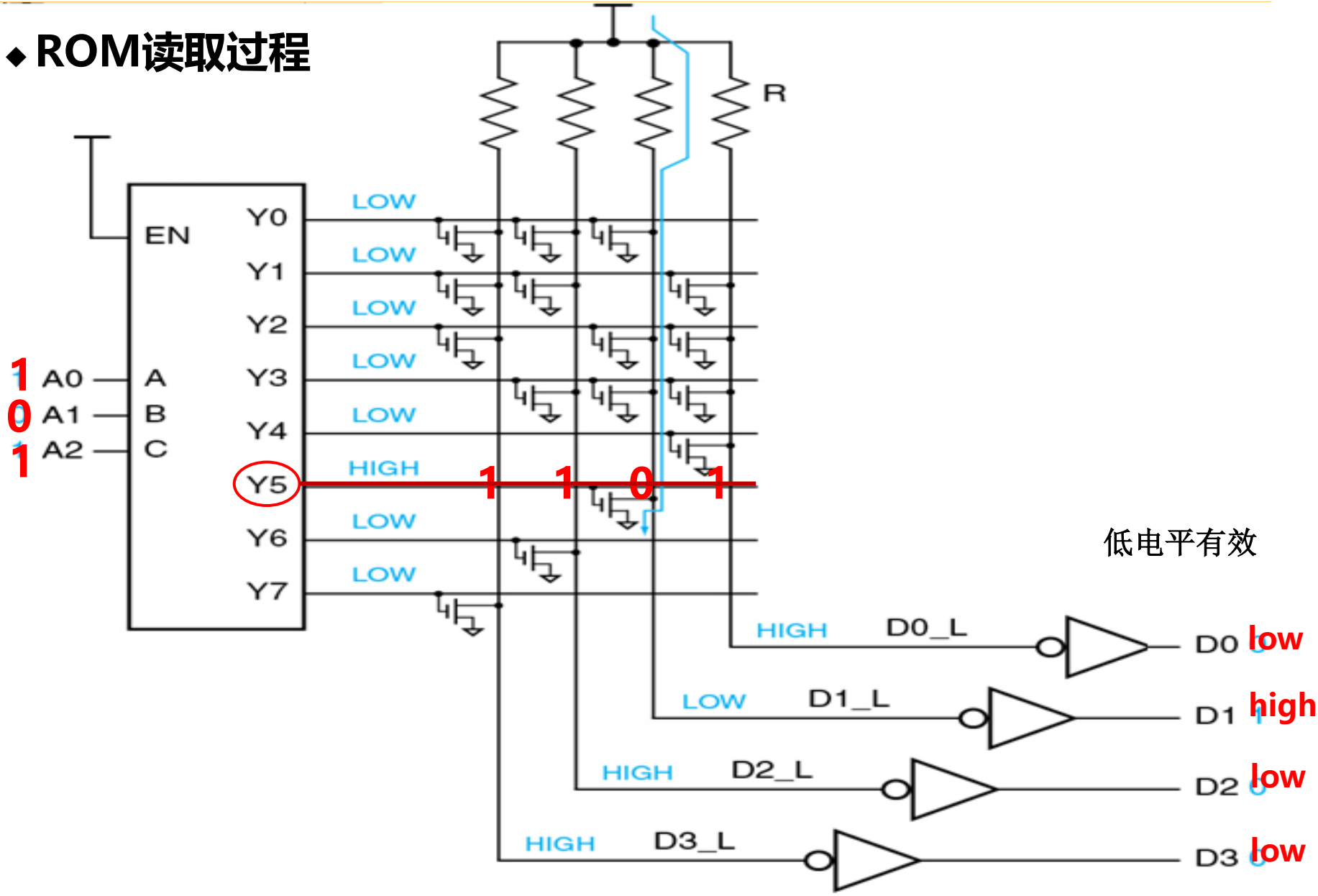
1.2 ROM

- ROM存储阵列根据MOS晶体管的有无来区分存储0和1
- 不同类型的ROM，主要区别在于MOS晶体管的特性不同



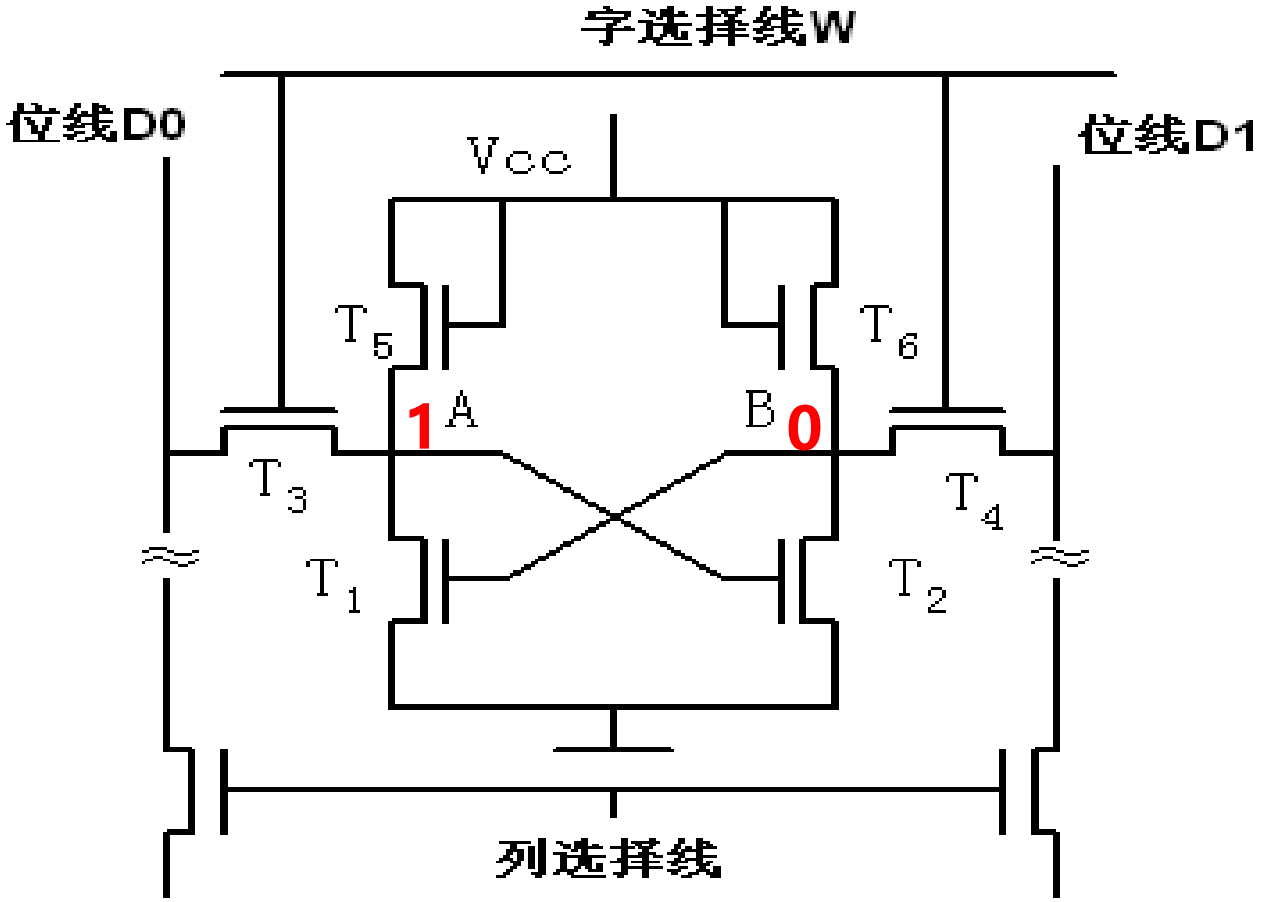
1.2 ROM读取

- ROM读取过程



1.2 静态存储器SRAM

- 静态存储器SRAM：存储单元使用6个MOS晶体管来实现



T1和T2构成触发器，T5、T6为负载管，T3、T4为门控管。

假设存“1”状态时A点为高电平，B点为低电平，此时T2管导通，T1管截止。

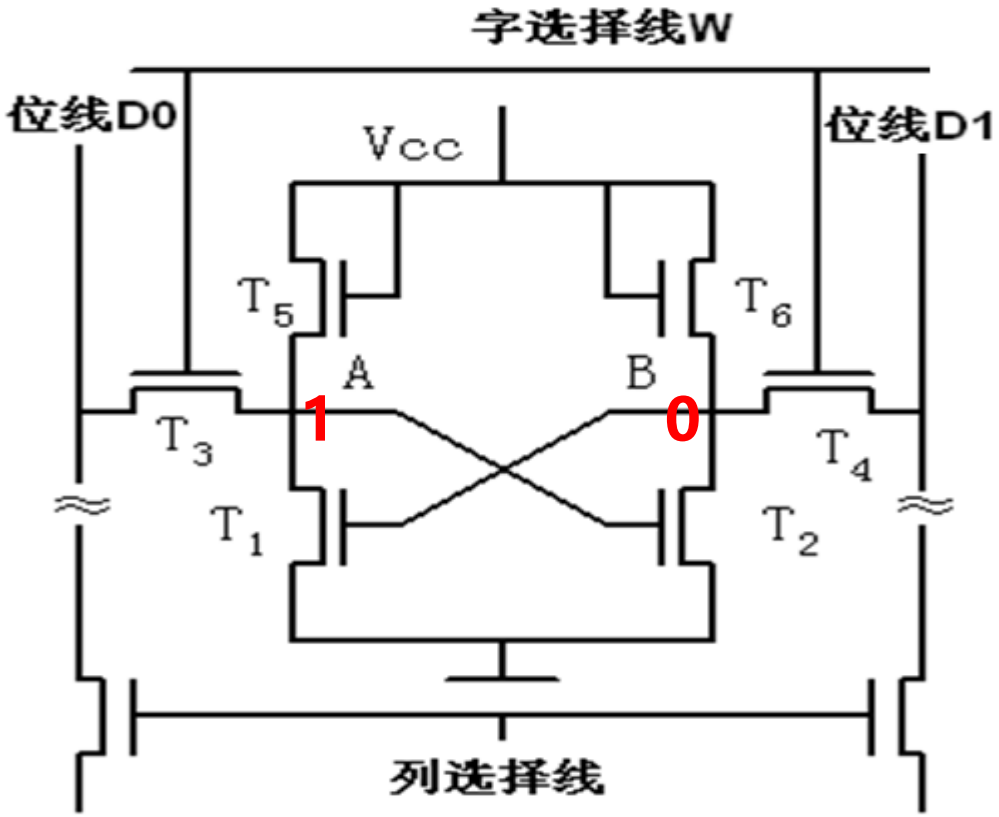
字选择线W为低电平时，T3与T4截止，此时触发器与外界隔离，从而保持信息不变。

只要保持电源，存储单元中存放数据就保持不变

缺点：器件多，因此价格高、功耗大、集成度低。

1.2 SRAM读写

- 读取时，先在D0、D1上加高电平，再在字线W上加高电平。
  - 若存储为1，则B点为低电平，电流经T2流到地，因T4导通，位线D1高电平被拉低而产生一个负脉冲；若存储为0，则A点为低电平，电流经T1流到地，因T3导通，位线D0高电平被拉低而产生一个负脉冲



- 写入时，字线W上加高电平
  - 若要写“1”，则在位线D1上加低电平（D0为高），因T4导通，B点电位下降，T1截止，A点电位上升，使T2管导通完成写“1”
  - 若要写“0”，则在位线D0上加低电平（D1为高），因T3导通，A点电位下降，T2截止，B点电位上升，使T1管导通完成写“0”

优点：读写速度快、无需刷新。

1.2 动态存储器DRAM

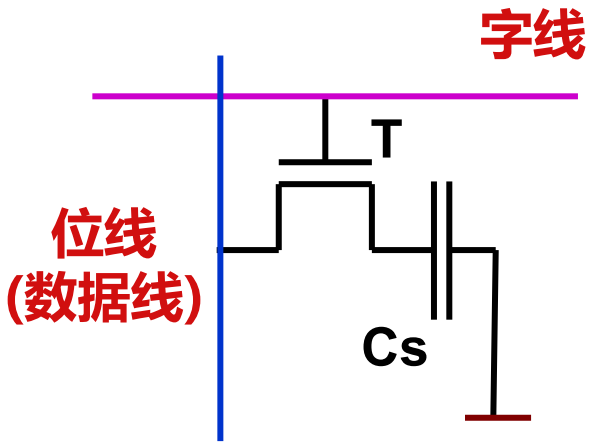
动态存储器DRAM：单MOS管，电容上存有大量电荷为1，否则0

读写原理：字线上加高电平，使T管导通。

写“0”时，数据线加低电平，使电容C_S上电荷对数据线放电；

写“1”时，数据线加高电平，使数据线对C_S充电；

读出时，数据线上有一读出电压。它与C_S上电荷量成正比。



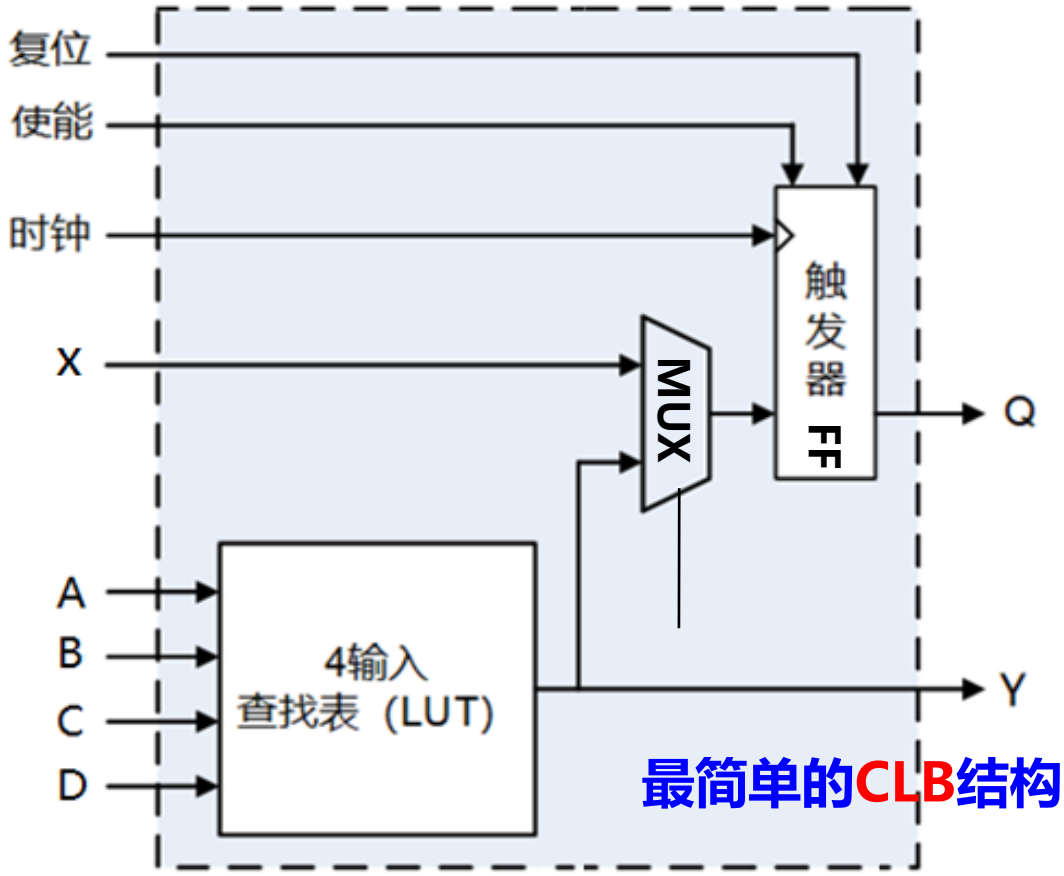
优点：电路元件少，功耗小，集成度高，用于构建主存储器

缺点：速度慢，是破坏性读出（需读后再生），需定时刷新

刷新：DRAM的一个重要特点是，数据以电荷的形式保存在电容中，**电容的放电使得电荷通常只能维持几十个毫秒左右**，相当于1M个时钟周期左右，因此要定期进行刷新（读出后重新写回），按行进行（所有芯片中的同一行一起进行），刷新操作所需时间通常只占1%~2%左右。

1.3 FPGA设计概述

- 现场可编程门阵列（Field Programmable Gate Array，FPGA）是一种高集成度的复杂可编程逻辑器件，可通过EDA (Electronic Design Automation)软件对其进行配置和编程，可反复擦写。
- FPGA内部包含大量可配置逻辑块CLB，它由若干查找表(Look-Up Table，LUT）及多路选择器、进位链、触发器FF等附加逻辑组成。
- 可对CLB进行不同配置。
  如右图：可编程配置输出为LUT实现的组合逻辑电路的结果Y；也可配置为输出时序逻辑电路的结果Q。
- LUT多采用SRAM实现。



通过HDL描述逻辑电路后，EDA工具会自动计算逻辑电路所有可能的结果，并以真值表形式事先写入RAM中

1.3 FPGA设计概述

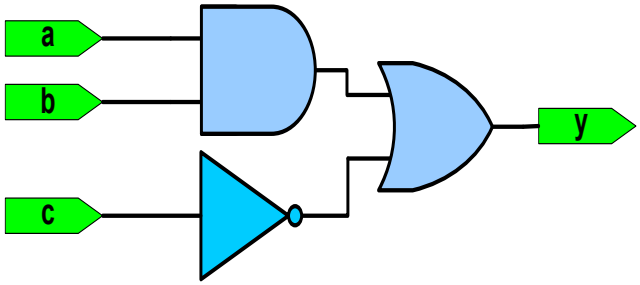
- 函数发生器通过**查找表**LUT实现，其中的内容**可编程配置**
- LUT存储单元中存放函数输出值，用于实现一个小规模逻辑函数

举例：若要实现函数 $f(a,b,c) = ab+\bar{c}$

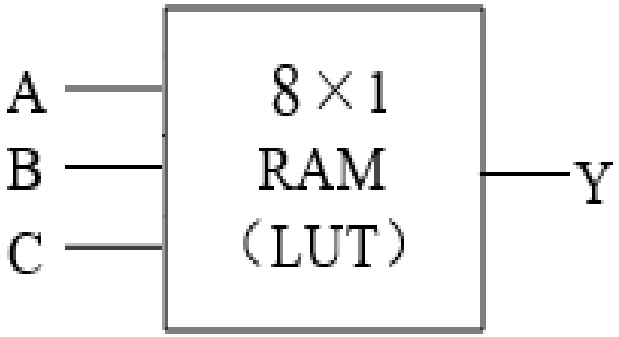
真值表

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

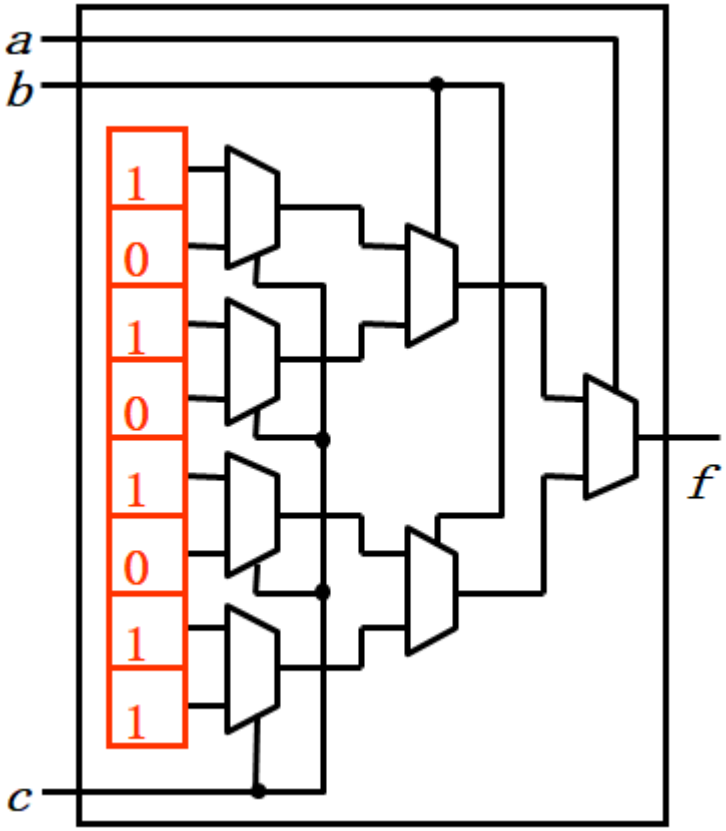
门电路实现



3输入LUT



3输入LUT实现



1.4 专用集成电路ASIC

- 专用集成电路（Application-Specific Integrated Circuit，ASIC）是一种应特定用户要求和特定电子系统需要而设计、制造的集成电路。
  - 全定制：设计者完成所有设计，速度更快
  - 半定制：使用标准库里的标准逻辑单元（标准单元）
- FPGA和ASIC目前都是电子设计领域的主流产品。
  - ASIC面向特定用户的需求，具有体积小、功耗低、可靠性高、性能高、保密性高、成本低等优点，一般用于批量大的专用产品中。
  - FPGA可编程特性使其应用非常灵活，但芯片内部逻辑门的使用率大幅降低，导致功耗高、速度慢、资源冗余且价格昂贵，一般用于小批量产品设计中。

第二讲 HDL概述

- VHDL和Verilog HDL
- 基于HDL的数字电路设计流程

2.1 VHDL和Verilog HDL

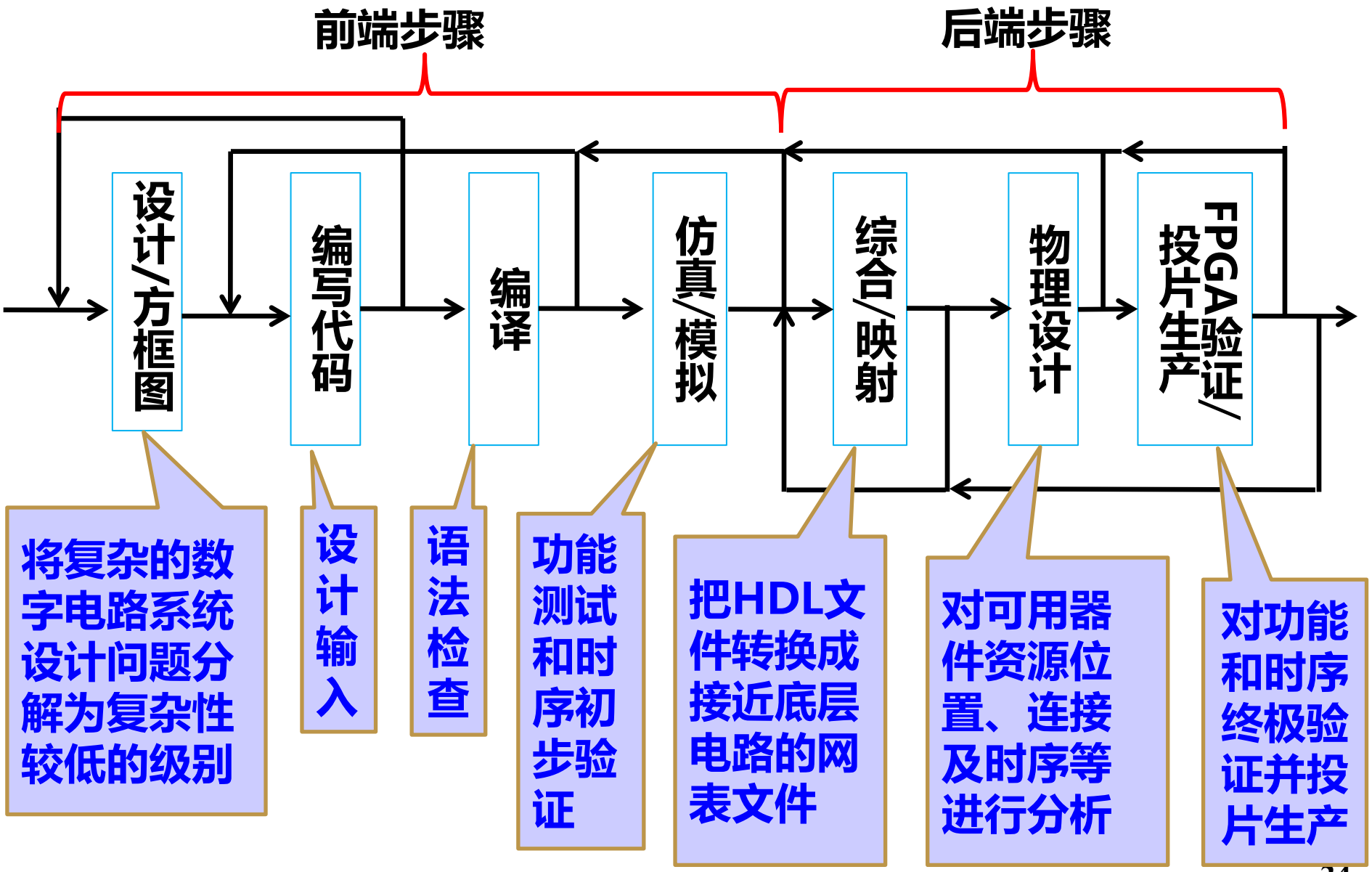
- VHDL最初于1981年由美国军方组织开发。
- Verilog 由Gateway Design Automation公司于1984年作为一个逻辑模拟的语言开发。
- VHDL 和Verilog 共同的特点在于：
  - 能形式化地抽象表示电路的结构和行为；
  - 支持逻辑设计中层次与领域的描述；
  - 可借用高级语言的精巧结构来简化电路行为的描述；
  - 具有电路仿真与验证机制以保证设计的正确性；
  - 支持电路描述由高层到低层的综合转换；
  - 硬件描述与实现工艺无关；
  - 便于文档管理；
  - 易于理解和设计重用。

2.1 VHDL和Verilog HDL

- Verilog HDL的特点
  - 易于学习和掌握，只要有C语言的编程基础，就可以快速掌握并使用
  - 在门级开关电路描述方面比VHDL强
  - 拥有更广泛的应用群体，成熟的资源比VHDL丰富
  - 比较合适作为学习HDL设计方法的入门和基础

- VHDL的特点
  - VHDL在系统级抽象方面比Verilog HDL强
  - VHDL比Verilog HDL的语句冗长且不灵活
  - VHDL较难掌握，需要有Ada编程基础并进行专业培训

2.2 基于HDL的数字电路设计流程



2.2 基于HDL的数字电路设计流程

- 设计并进行HDL编码
  - 根据目标电路的功能需求编写Verilog程序
  - 基本单元是模块module，包含声明和语句

例如：与门电路的Verilog程序代码

```
1 module top (
2   input a,
3   input b,
4   output c
5 );
6
7   assign c = a & b;
8
9 endmodule
```

关键字：module、assign、endmodule

模块名称：top，后接I/O端口列表

两个1位输入端口a、b

一个1位输出端口c

assign：后续为一个连续赋值语句，将赋值号右边的电路输出接入到左边的信号。

endmodule：模块结束

2.2 基于HDL的

- **仿真**：通过软件来模拟数字电路行为，观察电路中信号的变化过程，并且检查这些变化是否符合预期。
- **测试激励**：对**待测试模块**提供模拟输入的模块，驱动待测试模块进行工作。

右边是前述与门模块top对应的测试激励模块

6-9行：与门模块实例化

12-21行：仿真启动代码块

23-25行：always过程语句

begin和end之间为代码块

```
1  module sim_top ();
2
3    reg a_in, b_in;
4    wire c_out;
5
6    top dut(
7      .a(a_in),
8      .b(b_in),
9      .c(c_out)
10   );
11
12   initial begin
13     a_in = 1'b0;
14     b_in = 1'b0;
15     # 2
16     b_in = 1'b1;
17     # 2
18     a_in = 1'b1;
19     # 2
20     $finish;
21   end
22
23   always @(*) begin
24     $display("a=%d,b=%d,c=%d",a_in,b_in,c_out);
25   end
26
27 endmodule
```

initial是关键字，表示相应代码块会在仿真启动时执行

$finish是系统任务，表示结束仿真

$display是系统任务，用于控制台输出

always @(*)： always是关键字， @(*)表示a_in、b_in和c_out中任一变量的值发生变化就执行后面的代码块

2.2 基于HDL的数字电路设计流程

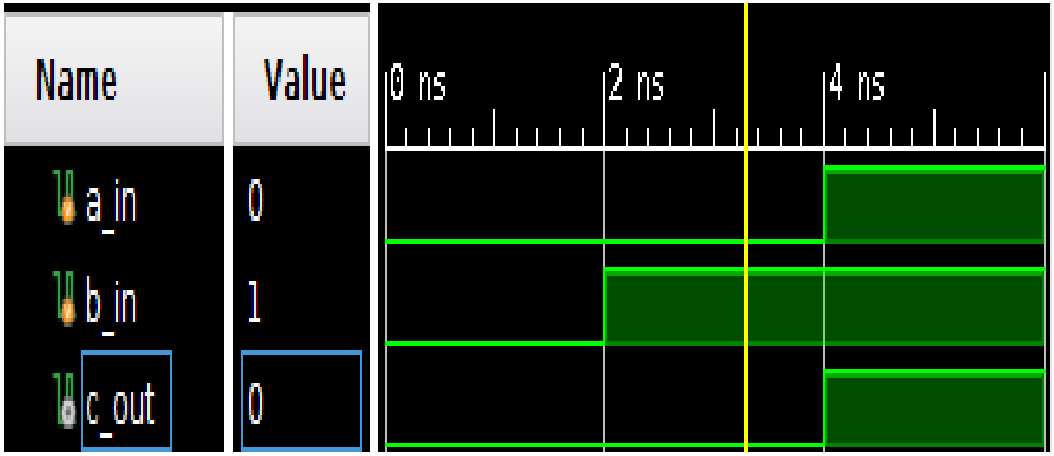
- 在仿真软件（如Vivado仿真器或modelsim）中运行测试激励模块
- 在控制台中输出：

  a = 0, b = 0, c = 0

  a = 0, b = 1, c = 0

  a = 1, b = 1, c = 1

- 在波形图中显示：



该例验证对象仅是一个模块，为模块级验证

验证多模块连接的数字系统，为系统级验证

```
1 module sim_top ();
2
3    reg a_in, b_in;
4    wire c_out;
5
6    top dut(
7       .a(a_in),
8       .b(b_in),
9       .c(c_out)
10   );
11
12   initial begin
13      a_in = 1'b0;
14      b_in = 1'b0;
15      # 2
16      b_in = 1'b1;
17      # 2
18      a_in = 1'b1;
19      # 2
20      $finish;
21   end
22
23   always @(*) begin
24   $display("a=%d,b=%d,c=%d",a_in,b_in,c_out);
25   end
26
27 endmodule
```

2.2 基于HDL的数字电路设计流程

- **综合**：利用EDA综合工具将HDL代码的描述转换成一种接近电路的底层描述-网表文件（netlist）
- 步骤如下：
  - **代码解析**（parsing）：根据语言规范进行解析得到层次结构信息（语法树）
  - **多级综合**（multi-level synthesis）：将解析到的层次结构信息转换成电路描述，对电路进行一定的优化
    - 如与门模块中的assign语句被转换为真正的与门描述
    - 若与门的一个输入端恒为0，则优化为输出恒为0
    - 并不是所有HDL代码都可综合，如$display，仅用于仿真时的控制台输出，是**不可综合代码**
  - **工艺映射**（technology mapping）：将电路描述进一步转换成特定工艺的**标准单元**，输出网表

2.2 基于HDL的数字电路设计流程

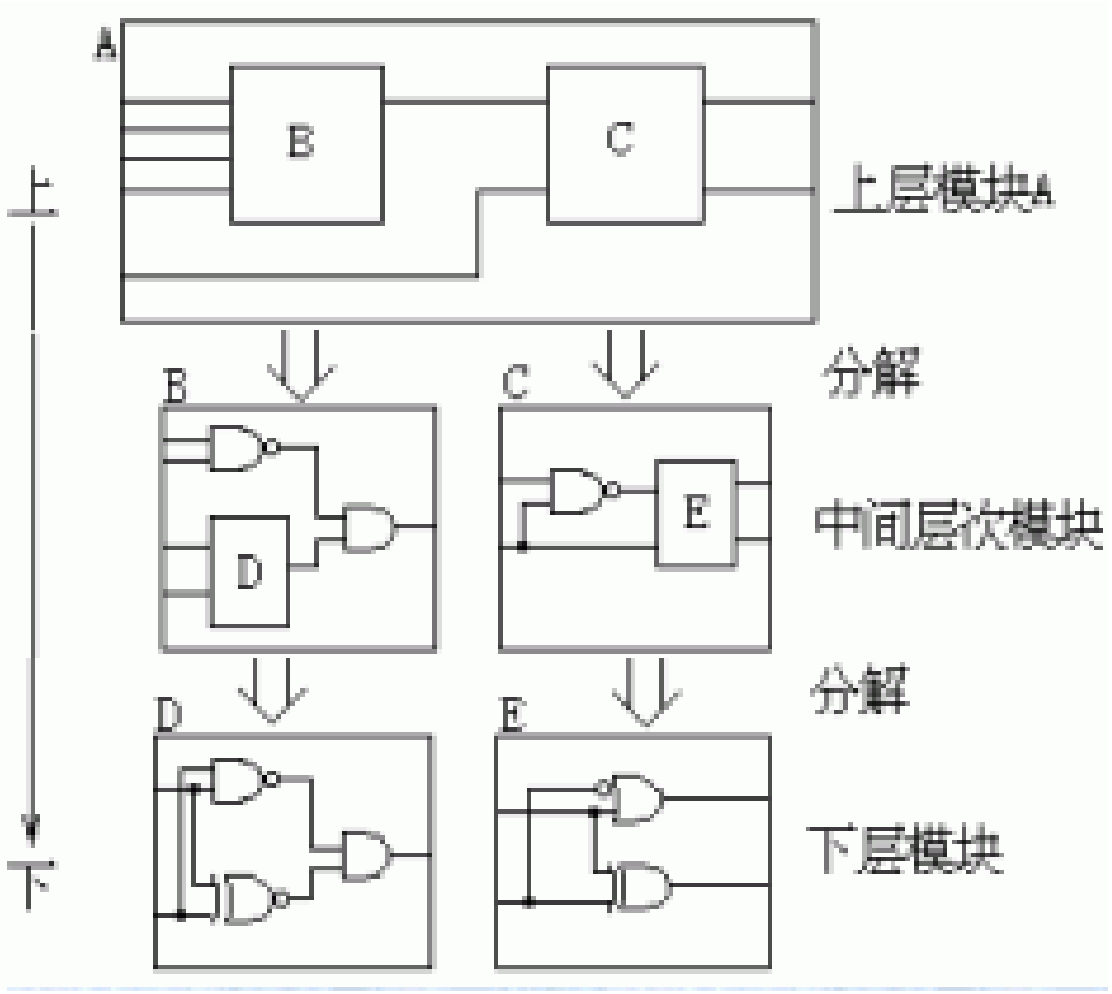
- **物理设计**：确定网表中标准单元如何连接以及连线的具体位置
- 过程：
  - 布局（placement）：确定标准单元在三维空间中的位置
  - 布线（routing）：将标准单元通过物理走线连接起来
  - 静态时序分析（static timing analysis）：根据标准单元和物理走线的结果，分析每条数据路径的延时情况，报告电路能运行的最大频率。
    - 延时最长的路径称为关键路径，它是阻碍电路频率进一步提升的瓶颈，设计者可以根据关键路径的信息对电路设计进行迭代优化
  - 电路和规则检查：版图和原理图的一致性检查、设计规则检查
  - 生成物理设计结果：对于ASIC流程，将会生成版图文件；对于FPGA流程，将会生成相应的比特流文件。

第三讲 Verilog语言简介

- 模块、端口和实例化
- 标识符、常量和注释
- 数据类型
- 运算符及其优先级

3 Verilog语言简介

- 数字电路：器件+连线
- 数字设计：模块+连线
- 器件抽象成模块
- 模块可逐级分解



3.1 模块、端口和实例化

- 模块（module）是Verilog编程的基本单元。其定义格式：

```
module 名称(端口定义列表);
   内部信号定义

   功能语句描述

endmodule
```

一个Verilog源文件可有多个模块

端口定义列表：模块通过输入、输出端口与其他模块交互。

input：输入到模块内部的信号

output：输出到模块外部的信号

inout：表示双向信号

内部信号定义：列出模块内需使用的信号。如sim_top模块中的a_in

功能语句描述：模块的主体部分，用于描述模块的功能。如top模块中的assign语句；sim_top模块中的实例化语句top dut(……)。

模块实例化语句的语法如下：

模块名 实例标识符(端口关联列表)

端口关联列表：按顺序关联或按端口名字关联

模块实例化语句相当于模块调用如，sim_top是top的父模块

3.2 标识符、常量和注释

- 标识符：以字母或下划线开头，可包含字母、数字、下划线和美元$符号，用来描述“对象”的名称。如模块名、端口名、变量名、常量名、实例名等。
  - 标识符不能与关键字同名。
  - 应采用有意义的名字。
  - 变量名区分大小写。
  - 合法的名字：A_99_Z，Reset，_54MHz_Clock$，Module
  - 非法的名字：123a，$data，module，7seg.v

3.2 标识符、常量和注释

- 整数常量：<size>'<base><value>
  - size为二进制位数，用十进制数表示。
  - base为常数的基数，可为二(b)、八(o)、十(d)、十六(h)进制，缺省为十进制。
  - value为指定进位制中的任意有效数字，可通过下划线“_”来分隔数字，用于提升可读性。

    8'b1010_1011：8位二进制常量1010 1011，真值为171

    64'hff01：64位十六进制常量，真值为65281；

    9'O17：9位八进制常量，真值为15。
- 注释
  - 单行注释符：//
  - 多行注释符：/* … */

3.3 数据类型

- 数据类型用来表示数字电路中数据存储和传送方式。
- 主要的数据类型有：
  - 网线类型（net）：表示元件或模块之间的物理连接，最常用的是wire类型，传送信号。
  - 寄存器类型（register）：表示抽象的存储元件，最常用的是reg类型，存储数值（0, 1，x，z）。

    x表示非法值或不定态；z表示浮空值或高阻态
  - 参数类型（parameter）：用于给常量赋予有意义的标识符，提高可读性。

3.3 数据类型

1. wire类型

用于表示元件或模块之间的物理连接，对应物理电路中的连线。

- wire类型变量需要被元件或模块的输出端口持续驱动。
- wire类型变量主要用途：
  （1）用于指定模块的输入输出端口；
  （2）声明将要在模块内的结构描述中建立连通性信号。
- wire类型变量缺省为1位的wire类型

2. 寄存器类型reg

用于表示存储元件的变量值，如用于描述触发器和锁存器的结果

- 用于过程赋值语句中赋值号左边的变量类型。
- 取值可以为0, 1，x，z。

3.3 数据类型

3. 向量和数组

- 向量：将多个1位信号组成一个整体进行操作，向量的长度称为位宽，通过有序界[MSB:LSB]来定义。

  wire[7:0] a;　// 一根位宽为8的连线a

  reg[31:0] rdata, wdata; // 位宽为32的寄存器rdata和wdata

  对于可综合电路来说，子界范围通过常量给出。

- 数组：具有相同数据类型的有序集合，通过索引访问各元素

  定义格式：数据类型 数组名[first_addr : last_addr]

  wire b[2:0]：数组b包含3根位宽为1的连线

  reg r[9:0]：数组r包含10个位宽为1的寄存器

  可用reg[MSB : LSB] m[first_addr : last_addr]定义存储器m

  reg[15:0] mem[1023:0]：1K×16 b的存储器mem

  数组变量不能整体引用，也不能子界范围引用，只能引用单个元素

  如b[0], r[8], mem[10]（表示mem中地址为10的一个16位元素）

3.3 数据类型

4. parameter类型

用于声明一个参数，表示有名字的常量，提升代码的可读性。

- 参数的定义是局部的，作用域是当前模块。

```
parameter WORD_WIDTH = 16, ADDR_WIDTH = 10;
```

5. 模块端口类型

- 模块实例化建立了父模块信号和子模块端口之间的关联，可看成物理连接。

  （1）子模块输入端口只能定义为wire类型，而与之关联的父模块中实例化输入信号作为驱动源，可以是wire类型或reg类型

  （2）子模块输出端口可以是wire或reg类型，而与之关联的父模块中实例化输出信号作为被驱动源，只能定义为wire类型

- 模块端口可以定义成向量，但不能定义成数组。
- 允许关联信号的位宽与端口位宽不同，但会报警。
- 允许端口保持未关联状态，其缺省值为高阻态，但被综合成0，可能会导致非预期结果，故应避免未关联。

3.3 数据类型

◆ 子模块端口类型定义

```
module Right (
  input in1,                  // 正确，未声明数据类型时缺省为wire类型
  input wire in2,             // 正确，输入端口只能为wire类型
  input [3:0] in3,            // 正确，端口可定义成向量
  output out1,                // 正确，未声明数据类型时缺省为wire类型
  output [3:0] out2,                  // 正确，端口可定义成向量
  output reg [1:0] out3               // 正确，输出端口可为reg类型
);
  ……
endmodule

module Wrong (
  input reg in1,                      // 错误，输入端口只能为wire类型
  output out1 [2:0]                   // 错误，端口不能为数组类型
);
  ……
endmodule
```

3.3 数据类型

◆ 父模块端口类型定义

```
module Top (……);
  wire w1, w2;
  wire [3:0] wv1;
  wire [1:0] wv2;
  reg r1, r2;
  reg [3:0] rv1;
  wire [5:0] too_long;
  Right warning_or_wrong (     // 对Right模块进行实例化
    .in1(1'b0),       // 正确，输入端口可与常量关联
    .in2(),           // 正确但有警告，输入端口可处于未关联状态
                      // 输入端口in3未给出，正确但有警告，可处于未关联状态
    .out1(r2),        // 错误，输出端口不能与reg类型信号关联
    .out2(),          // 正确，输出端口可处于未关联状态
    .out3(too_long)       // 正确但有警告，向量位宽与端口定义不一致，
                          // too_long[5:2]处于未关联状态
  );
endmodule
```

3.4 运算符及其优先级

- 数字电路设计中的数据本质上是位串，数据类型含义表示的是数字电路中数据存储和传送的方式。
- HDL描述的是电路的物理结构，高级编程语言描述的是程序的执行流程。
- Verilog中每一种运算符都有其对应的电路结构

1. 算术运算符

- 包括“+”、“-”、“*”、“/”和“%”，表示加、减、乘、整除和取模运算。
- “+”和“-”运算结果的位宽为两个操作数中位宽较长者再加1，多出来一位用于表示进位或借位；将综合出补码加减运算电路。
- “*”运算结果的位宽为两个操作数的位宽之和；将综合出阵列乘法器电路。
- “/”和“%”运算结果的位宽与第一个操作数的位宽相同。有的综合器会综合出阵列除法器电路。
- 阵列乘法器、阵列除法器比较复杂、延迟长。因此在高性能处理器设计中，通常不直接使用“*”、“/”、“%”运算符，而是会编写高性能乘法器和除法器的HDL代码。

3.4 运算符及其优先级

2. 位运算符

- 包括“~”、“&”、“|”、“^”和“^~”（或“~^”），分别表示按位取反、按位与、按位或、按位异或和按位同或运算。
- 位运算结果的位宽与操作数的位宽相同，如两个操作数的位宽不同，则短操作数先进行零扩展（即高位补“0”），再进行运算。
- 综合与位宽数量相同的一个或多个门电路

3. 归约运算符

- 包括“&”、“~&”、“|”、“~|”、“^”和“^~”（或“~^”），分别表示与归约、与非归约、或归约、或非归约、异或归约和同或归约运算。
- 无论操作数的位宽是多少，归约运算结果的位宽均为1。
- 将操作数最低位依次与前面各位进行与、或、异或等运算。

  若信号a的位宽为4，则&a = ((a[0] & a[1]) & a[2]) & a[3]
- 综合出一个输入端口数量与操作数位宽相同的门电路

3.4 运算符及其优先级

4. 逻辑运算符

- 包括“&&”、“||”和“!”，分别表示逻辑与、逻辑或和逻辑非运算。
- 将操作数当做布尔值，非零为真，结果位宽为1。
- 其行为可通过归约运算符和位运算符表达，从而得到综合出的电路。

如：a && b = (|a) & (|b)，a || b = (|a) | (|b)，!a = ~(|a)

5. 等式运算符

- 主要包括“==”和“!=”，分别表示等于判断和不等于判断。
- 等式运算将两个操作数的每一位分别进行比较，若均相同，结果为1。
- 等式运算符的行为可以通过归约运算符和位运算符表达，从而得到等式运算符综合出的电路。

如：(a == b) = ~(|(a ^ b))，(a != b) = |(a ^ b)

3.4 运算符及其优先级

6. 关系运算符

- 包括“<”、“<=”、“>”和“>=”，分别表示小于、小于等于、大于和大于等于判断。
- 关系运算结果的位宽均为1。
- 综合出带标志位的补码加减运算电路，通过输出标志位来判断关系。

7. 位拼接运算符

- “{ }”用于将两个或多个信号按顺序拼接起来。
- “{n{m}}”表示将n个m拼接。
- 是唯一一个操作数数量可变的运算符，操作数之间用“,”分开。
- 位拼接运算结果的位宽为所有操作数的位宽之和。

  如：若a = 2'b11，b = 4'b1101，则{a, b[1:0], 3'b0} = 7'b1101000；{a, {2{c, b}}} = {a, c, b, c, b}。
- 位拼接运算符的行为相当于信号集线器，无需综合出逻辑门器件。

3.4 运算符及其优先级

8. 移位运算符

- 包括“>>”和“<<”，分别进行逻辑右移和逻辑左移运算。
- 移位位数由右边的操作数给出，并用相应数量的0填补移出的空位。
- 对于右移，结果位宽和被移位操作数的位宽相同；
- 对于左移，结果位宽为被移位操作数的位宽加上左移的位数。

  如：4'b1001 >> 3 = 4'b0001；4'b1001 << 2 = 6'b100100；1 << 4 = 36'b10000。
- 若移位位数为常量，则移位运算符的行为可通过位拼接运算符和下标选择来表达。

  如：假设被移位操作数a的位宽为wa，移位位数为b，wa和b皆为常量，则 (a >> b) = {{b{1'b0}}, a[wa-1:b]}

  (a << b) = {a, {b{1'b0}}}
- 若移位位数为变量，则移位运算符将综合出移位器电路（组合逻辑电路，如桶型移位器）。

3.4 运算符及其优先级

7. 条件运算符

- “条件？表达式1：表达式2”：若条件为真，则将表达式1的作为条件运算的结果；否则将表达式2的作为条件运算的结果。
- 条件运算结果的位宽与表达式的位宽相同。
- 若两个表达式的位宽不同，则位宽较短的表达式将进行零扩展。
- 条件运算符综合出二路选择器，条件对应的电路输出作为其控制信号

3.4 运算符及其优先级

- 运算符的优先级

类 别	运 算 符	优先级
逻辑、位运算符	! ~	高
算术运算符	* / %	
	+ -	
移位运算符	<< >>	
关系运算符	< <= > >=	
等式运算符	= = != === !==	
归约、位运算符	& ~&	
	^ ^~或~^	
	| ~|	
逻辑运算符	&&	
	||	
条件运算符	? :	低

为提高程序的可读性，建议使用括号来控制运算的优先级！

第四讲 verilog的建模方式

- 三种建模方式
  - 结构化建模
  - 数据流建模
  - 行为建模
- 行为建模中的过程语句

4.1 三种建模方式

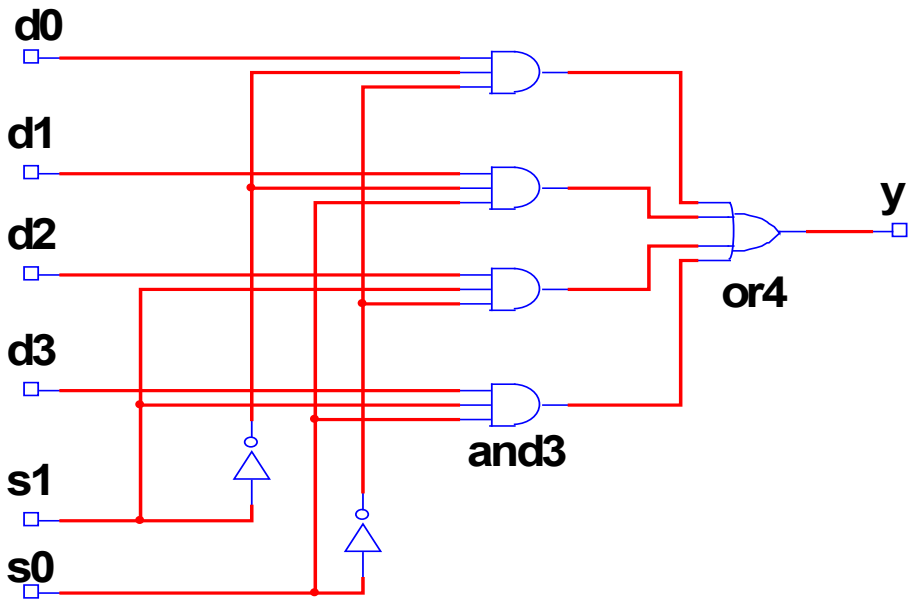
- 结构化建模
  - 将电路描述成一个分级的子模块系统，并通过逐层调用子模块来构成功能复杂的数字系统。
  - 根据子模块不同的抽象级别，结构化建模方式分成以下三类：

模块级：调用用户设计的子模块(如前例sim_top模块调用top子模块)

门级：调用Verilog提供的基本门级元件 (以下4-路选择器举例)

开关级：调用Verilog内建的基本开关元件 (无须了解)

1位4-路选择器的门级结构




```
……
// 调用非门，生成s1_n 和 s0_n
not (s1_n, s1);
not (s0_n, s0);
// 调用三输入与门
and (y0, d0, s1_n, s0_n);
and (y1, d1, s1_n, s0);
and (y2, d2, s1, s0_n);
and (y3, d3, s1, s0);
// 调用四输入或门
or (y, y0, y1, y2, y3);
……
```

4.1 三种建模方式

- **数据流建模**
  - 从数据的视角出发，通过描述数据在电路中的流动方向来给出电路的功能。
  - 适合组合逻辑电路的描述，不适合时序逻辑电路的描述。
  - 主要通过连续赋值语句进行建模，语法为：

    assign 网线 = 表达式;

    用赋值号右边的表达式对应电路的输出来驱动赋值号左边的网线
  - 注意点

    -赋值号右边数据类型可以是wire和reg，左边则不能是reg类型

    -多个连续赋值语句之间是并行关系，没有前后之分

    -对一个未定义的变量赋值，相当于驱动一个位宽为1的wire型变量

    -相同的网线不能进行重复驱动

    -描述的电路不能出现组合回路

4.1 三种建模方式

例：使用数据流模式实现1位四路选择器

使用布尔表达式来代替基本门级元件的实例化

```
module mux4_to_1 (
    output y,
    input d0, d1, d2, d3,
    input s0, s1
);
    assign y=(~s1 & ~s0 & d0) | (~s1 & s0 & d1) | (s1 & ~s0 & d2) | (s1 & s0 & d3);
endmodule
```

采用条件运算符来描述

```
module mux4_to_1 (
      output y,
      input d0, d1, d2, d3,
      input s0, s1
);
      assign y = (s1) ? (s0 ? d3 : d2) : (s0 ? d1 : d0);
endmodule
```

数据流建模方式比门级结构化建模方式的代码更简洁。在现代数字系统的开发过程中，大部分组合逻辑电路都采用数据流建模方式来描述。

4.1 三种建模方式

- 行为建模方式
  - 通过高级编程语句和结构编写的过程块来描述数字系统的功能。
  - 从系统的功能出发进行描述，无需关注电路的具体结构，由综合器根据过程块所描述的行为综合出电路结构。
  - 关键要素是always语句，基本语法：

    always @(事件信号列表) 过程语句

    -含义：当事件信号列表中的任意一个信号发生变化时，过程语句中的信号将按照所描述的行为进行更新。

    -过程语句中被赋值的左边变量只能是reg类型变量。

    -事件信号列表中有多个信号时，需用关键字or或者逗号来连接。

    -（*）隐式事件信号列表，*表示过程语句赋值号右侧所有信号集合，含义是赋值号右侧任一信号发生变化都会导致左侧信号更新

    -可通过隐式事件信号列表的always语句对组合逻辑电路进行建模

4.1 三种建模方式

例：使用行为建模方式实现1位四路选择器

```
module mux4_to_1 (
    output reg y,        // 注意此处为reg类型
    input d0, d1, d2, d3,
    input s0, s1
);
    always @(*)          // 相当于 @(s1, s0, d0, d1, d2, d3)
        case ({s1, s0})
            2'b00: y = d0;
            2'b01: y = d1;
            2'b10: y = d2;
            2'b11: y = d3;
        endcase
endmodule
```

4.2 行为建模中的过程语句

1. begin-end复合语句
   - 在语法上将多条语句看成一条语句。类似{ }。
2. 两种过程赋值语句
   - 阻塞赋值语句，其语法为：寄存器 = 表达式;
     - 立即更新，后续语句中按照新值计算
     - 按在代码中出现的**顺序执行**
     - 通常用来描述**组合逻辑电路**
   - 非阻塞赋值语句，其语法为：寄存器 <= 表达式;
     - 滞后更新，新值在其它所有操作都结束后才会进行更新
     - 一组非阻塞赋值语句是**并行执行的**
     - 利用时钟变量作为事件信号时，赋值符合数据存储的特点
     - 通常用于描述**时序逻辑电路**

4.2 行为建模中的过程语句

2. 两种过程赋值语句（续）

- 例1：用两种过程赋值语句对全加器建模有何不同?

```
always @(*)
    begin
        p = a ^ b;
        g = a & b;
        s = p ^ cin;
        cout = g | (p & cin);
    end
```

```
always @(*)
    begin
        p <= a ^ b;
        g <= a & b;
        s <= p ^ cin;
        cout <= g | (p & cin);
    end
```

若开始a、b、cin都是0，则输出p、g、s、cout都是0。随后a从0改为1，则

（阻塞方式下按顺序执行赋值语句，非阻塞方式下并行执行赋值语句）

阻塞赋值方式下，p=1•0=1，g=1•0=0，s=1•0=1，cout=0+(1•0)=0

非阻塞赋值方式，p=1•0=1，g=1•0=0，s=0•0=0，cout=0+(0•0)=0。

由此可见，采用非阻塞方式实现的加法器有问题!

4.2 行为建模中的过程语句

2. 两种过程赋值语句（续）

- 例2：用非阻塞方式对计数器建模。

```
reg [31:0] cnt;
    always @ (posedge clk)
       cnt <= cnt + 1'd1;
```

上述代码可以综合出一个“每来一个时钟自增1”的计数器

赋值号右边的cnt是时钟到来前的值，左边cnt是增量后的

posedge表示在时钟clk的上升沿触发

negedge表示在时钟clk的下降沿触发

4.2 行为建模中的过程语句

3. if-else语句

- 根据判断条件的真假更新相应分支中的寄存器变量。
- 格式：
  - if (表达式) 语句1；
  - if (表达式) 语句1；else 语句2；
  - if (表达式1) 语句1；
    else if （表达式2）语句2；
    else if （表达式n） 语句n；

  -这里“表达式”为逻辑表达式或关系表达式，值为0或z，判定结果为“假”；值为1，判断结果为“真”。

  -多语句时使用“begin-end”，形成复合块语句。
- 可转化为条件运算符。
- 可综合成选择电路，即多路选择器。

4.2 行为建模中的过程语句

4. case语句

- 根据表达式的值从多个分支中选择其中一个并按照所选择分支的描述来更新寄存器变量。
- case语句的语法为：

```
case（选择表达式）
    值1：语句1；
    值2：语句2；
      …
    值n：语句n；
    [default: 语句n+1；]
endcase
```

- 每个“值”可以是表达式
- 可转换为嵌套的if-else
- 用于译码器、多路选择器等
- 在组合逻辑电路设计中，case要覆盖所有分支情况，否则会综合成锁存器保持原值，与预期电路不符。

4.2 行为建模中的过程语句

5. 循环语句

- 提供了对于相似结构电路的一种简洁描述方式。
- 在高级语言中，循环语句表示时间上重复执行的相似代码；而在HDL中，循环语句表示在空间上重复描述的电路。
- 使用较多的是for循环结构：

  for （表达式1；表达式2；表达式3）语句
- 并不存在“执行”for循环的电路，只是空间上有多个相似电路
- 若循环结束条件表达式中除了循环变量外还有其他变量的话，电路会变得很复杂

例如，for (i = 0; i < threshold; i = i + 1) 将综合出比较器

for (i = 0; i < 8; i = i + 1) 仅综合出8个相似的电路即可

第五讲 verilog代码实例

- 组合逻辑代码实例
- 时序逻辑代码实例

5.1 组合逻辑电路代码设计

- 设计3-8译码器

```
module decode3to8 (
    output reg [7:0] out,
    input [2:0] in
);
    always
      case (in)
        0: out = 8'b00000001;
        1: out = 8'b00000010;
        2: out = 8'b00000100;
        3: out = 8'b00001000;
        4: out = 8'b00010000;
        5: out = 8'b00100000;
        6: out = 8'b01000000;
        7: out = 8'b10000000;
      endcase
endmodule
```

行为建模方式

```
module decode3to8 (
      output reg [7:0] out,
      input [2:0] in
);
      integer i;
      always
          for (i = 0; i < 8; i = i + 1)
                out[i] = (in == i);
   endmodule
```

行为建模方式

```
module decode3to8 (
      output [7:0] out,
      input [2:0] in
);
      assign out = 1 << in;
endmodule
```

数据流建模方式

5.1 组合逻辑电路代码设计

- 七段数码管译码电路

仅包含0~9这10种情况，因而在case语句最后需加default分支，使得当I为0~9以外的数时，能够将O设置为0。否则，综合器将会综合出锁存器，使得在输入0~9以外的数时保留上次的输出，这与预期的功能不符。

```
module decode7seg (
    output reg [0:6] O,
    input [0:3] I
);
    always
        case (I)
            0: O = 7'b1111110;
            1: O = 7'b0110000;
            2: O = 7'b1101101;
            3: O = 7'b1111001;
            4: O = 7'b0110011;
            5: O = 7'b1011011;
            6: O = 7'b1011111;
            7: O = 7'b1110000;
            8: O = 7'b1111111;
            9: O = 7'b1110111;
            default: O = 7'b0;
        endcase
endmodule
```

5.2 时序逻辑电路设计

- 设计带复位端rst和使能端en的D触发器

```
module Dff (
    input clk, rst, en, d,
    output reg q
);
    always @(posedge clk)
        begin
            if (rst) q <= 1'b0;
            else if (en) q <= d;
        end
endmodule
```

```
module AsyncDff (
    input clk, rst, en, d,
    output reg q
);
    always @(posedge clk or posedge rst)
        begin
            if (rst) q <= 1'b0;
            else if (en) q <= d;
        end
endmodule
```

同步复位方式

只有当时钟上升沿到达后，才能进行复位

异步复位方式

只要复位信号rst从0变为1就可以复位，无需等待时钟上升沿到来

5.2 时序逻辑电路设计

- **自动售货机出售1.5元的零食，只接收五角和一元两种输入**

```
module VendingMachine (
  input clk, arst,  //clk-时钟信号，arst-异步复位信号
  input in5, in10, // 是否投了五角或一元硬币
  output snack,     // 是否出货
  output out5   // 是否找五角零钱
);
  reg [2:0] state;  //5个状态，需要3位编码
  parameter s_idle=0,s_05=1,s_10=2,s_15=3,s_20=4;  //状态编码赋值给状态名称
  always @(posedge clk or posedge arst) begin
    if (arst) state <= s_idle;
    else begin
     case (state)
       s_idle: begin
           if (in5) state <= s_05;
           if (in10) state <= s_10;
        end
      s_05: begin
           if (in5) state <= s_10;
           if (in10) state <= s_15;
      end
           s_10: begin
              if (in5) state <= s_15;
              if (in10) state <= s_20;
           end
           s_15: state <= s_idle;
           s_20: state <= s_idle;
        endcase
     end
  end
  assign snack=(state==s_15)||(state==s_20);
  assign out5=(state == s_20);
endmodule
```

这里用case语句实现了一个有限状态机