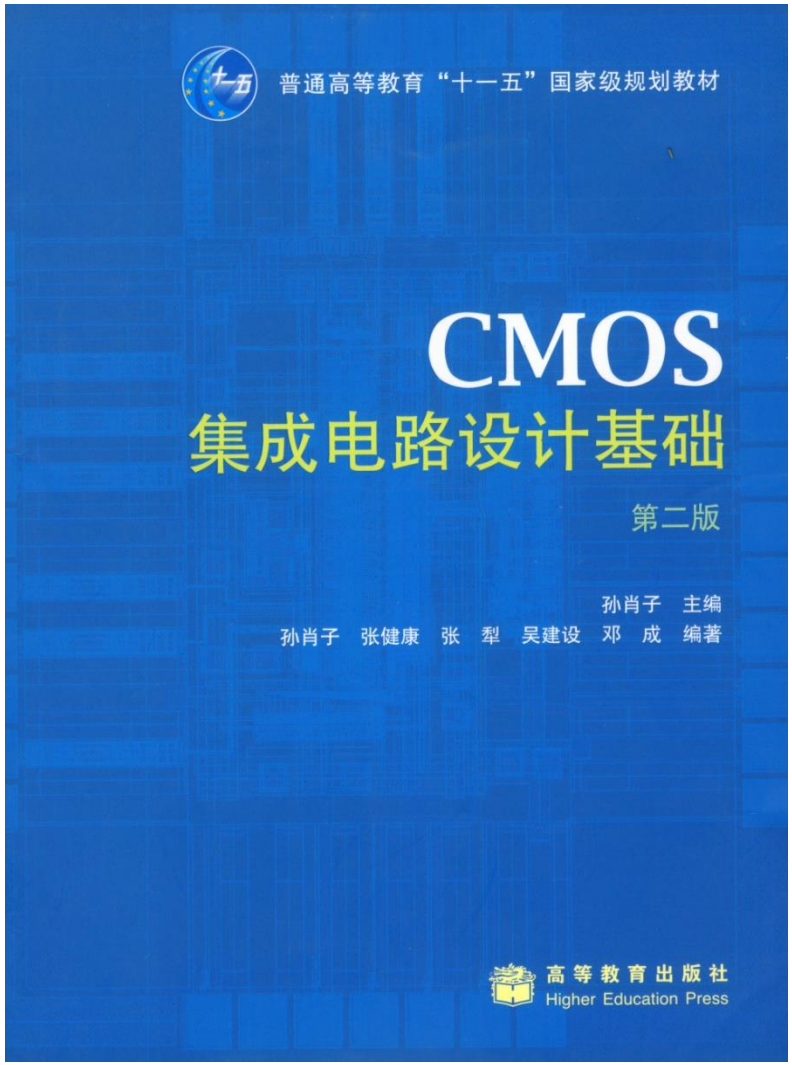
西安电子科技大学

集成电路导论

群名称：集成电路导论1901011
群　号：695982725

通信工程学院　郭杰
北校区科技楼B-502
E-mail: jguo@mail.xidian.edu.cn
Tel: 029-88203116
QQ: 1600273298

教材



“CMOS 集成电路设计基础(第二版)”

孙肖子等编著

高等教育出版社

2008年

课程介绍

- 先修课程

  模拟电子技术基础→工艺、元器件

  数字电路与逻辑设计→集成电路设计
- 测试方法
- 设计工具
- 设计实例
- 后续课程实验及简单系统设计

课程介绍

- 课时分配：

  32学时课堂教学

- 课程要求：

  了解专用集成电路设计的基本概念、注意事项，掌握设计步骤和相关设计方法

- 考核方式：

  闭卷笔试与上机实习相结合

西安電子科技大學

第一章 概述

IC, Integrated Circuit

ASIC, Application Specific Integrated Circuit

通用IC与ASIC的区别

- 每批生产量

几十片~几千片

1万片以上

- 电路设计者

用户或用户委托ASIC设计公司

厂家设计

- 设计思想

面市快

芯片面积最小

- 设计方法

采用ASIC设计的突出优点

- 某些复杂电路系统只能采用ASIC进行设计
- 采用ASIC设计复杂电路系统具有极高的性能/价格比
- 能够减少开发时间，加快新产品的面世速度(Time-to-Market)
- 提高系统的集成度，缩小印制板面积，降低系统的功耗
- 提高了产品的可靠性，使产品易于生产和调试，降低了维护成本

西安電子科技大學

1.1 集成电路的发展历程

集成电路的出现与发展

- 1947-1948年：世界上第一支(点接触)晶体三极管—标志电子管时代向晶体管时代过渡。因此1956年美国贝尔实验室三人获诺贝尔奖
- 1950年：成功制出结型晶体管
- 1952年：英国皇家雷达研究所第一次提出“集成电路”的设想
- 1958年：美国德克萨斯仪器公司制造出世界上第一块集成电路(双极型-1959年公布)
- 1960年：制造成功MOS集成电路

集成电路发展的特点

- 特征尺寸越来越小(0.13μm…45nm→32nm…)
- 晶圆尺寸越来越大(8inch~12inch)
- 芯片集成度越来越高(>2000K)
- 时钟速度越来越高(>500MHz)
- 电源电压越来越低(1.0V)
- 布线层数越来越多(9层)
- I/O引脚数越来越多(>1200)

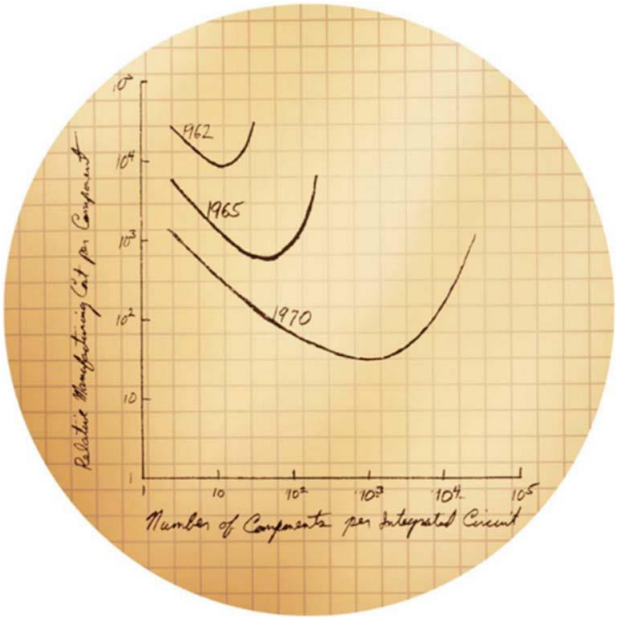
集成电路特征参数的进展情况

发展阶段 / 主要特征	1990	1997	1999	2001	2003	2006
晶体管数/芯片	10^6~10^7	11×10^6	21×10^6	40×10^6	76×10^6	200×10^6
线宽/*μm*	1	0.25	0.18	0.15	0.13	0.1
时钟频率/*MHz*	75	750	1200	1400	1600	2000
芯片面积/mm^2	50~100	300	385	430	520	620
金属布线层		6	6~7	7	7	7~8
DRAM容量		256M	1G	1G~4G	4G	16G
最低供电电压/*V*		1.8~2.5	1.2~1.8	1.2~1.5	1.2~1.5	0.9~1.2
最大晶圆直径/*mm*	150 (6英寸)	200 (8英寸)	300 (12英寸)	300 (12英寸)	300 (12英寸)	300 (12英寸)

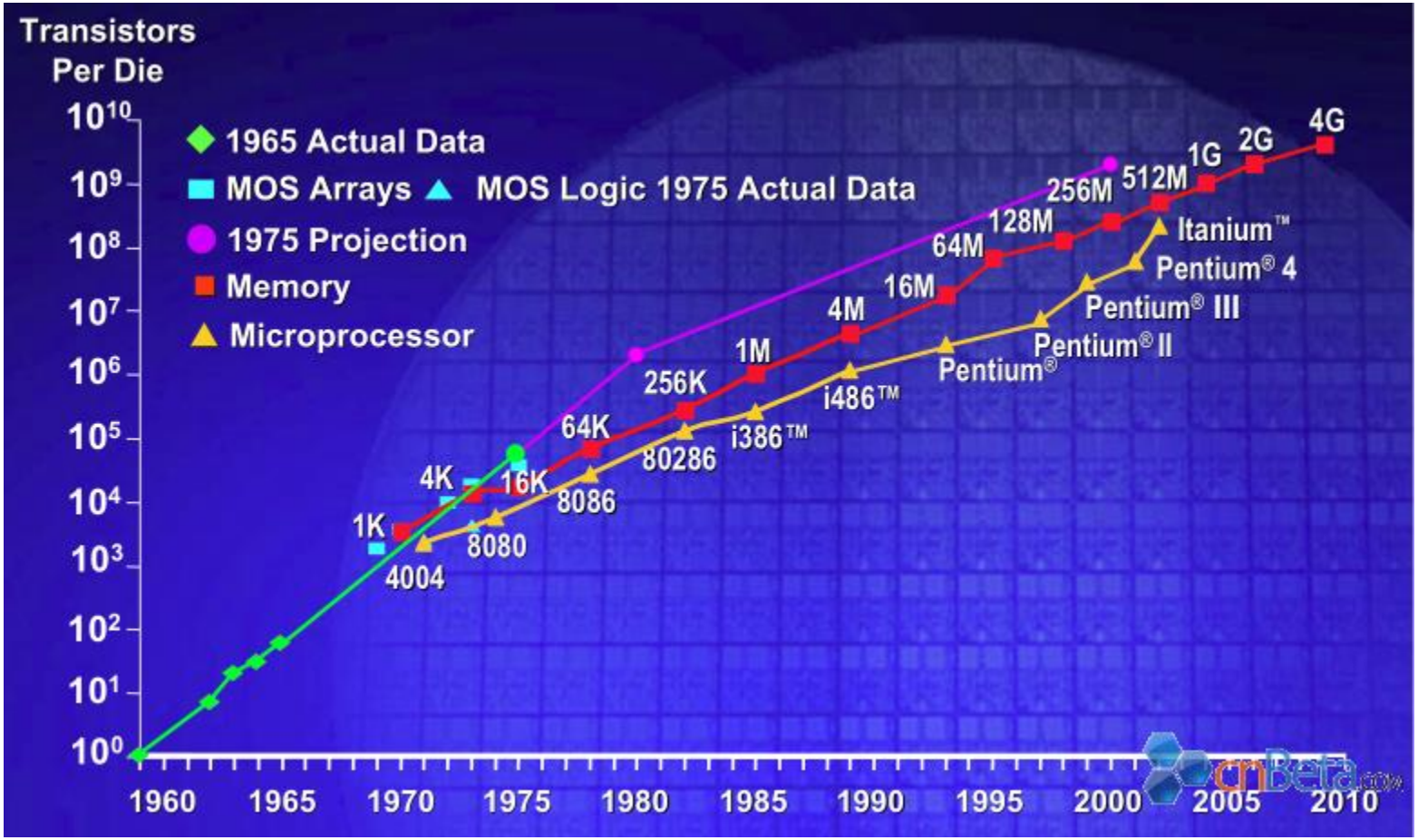
摩尔定律(Moore's Law)

美国Intel公司创始人之一Gordon Moore于1965年总结出的有关集成电路发展趋势的著名预言，其主要内容是：

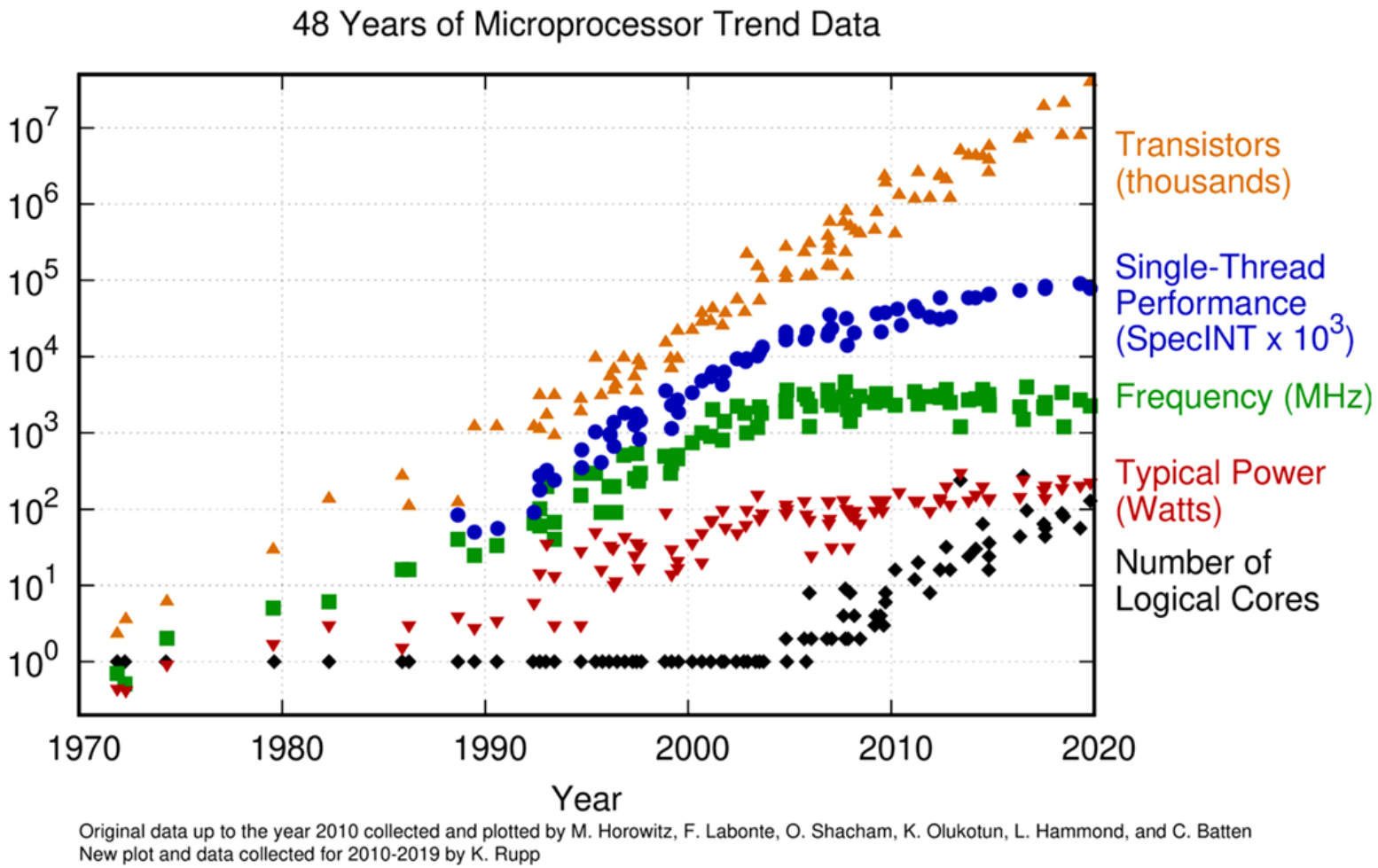
- 最小尺寸以每三年70%的速度下降，集成度每年翻一番；
- 价格每两年下降一半；
- 这种规律在提出后30年内是正确的。



摩尔定律(Moore's Law)

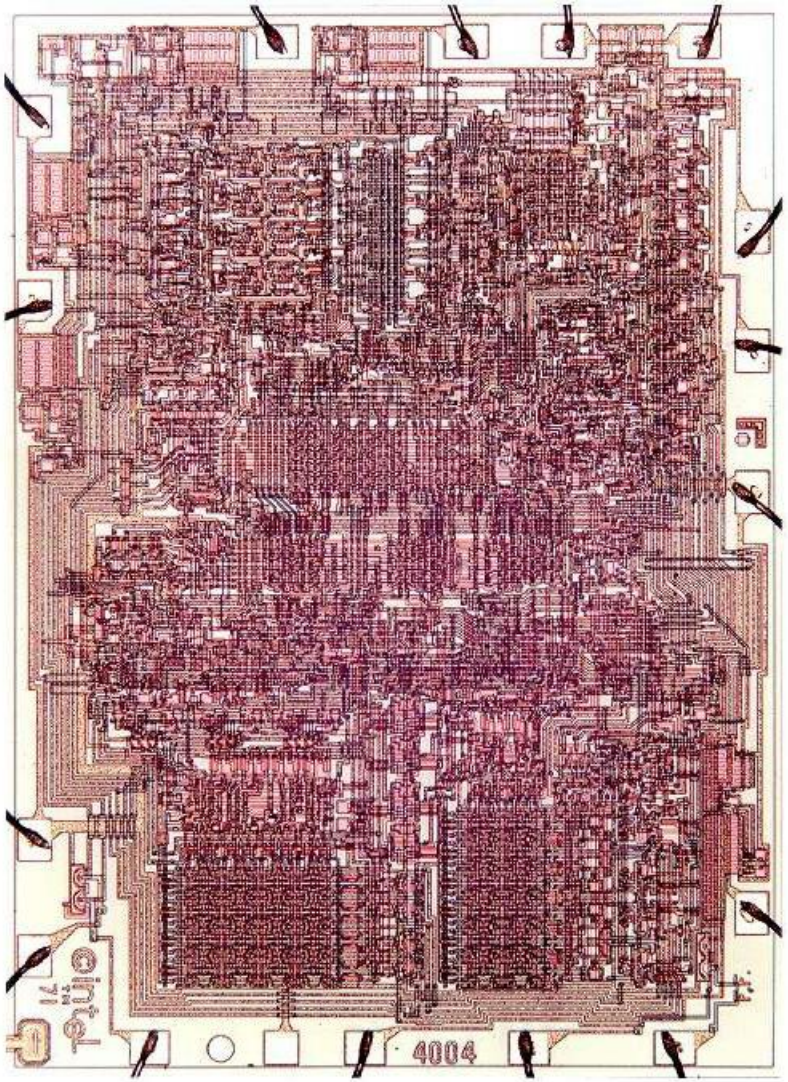


摩尔定律(Moore's Law)



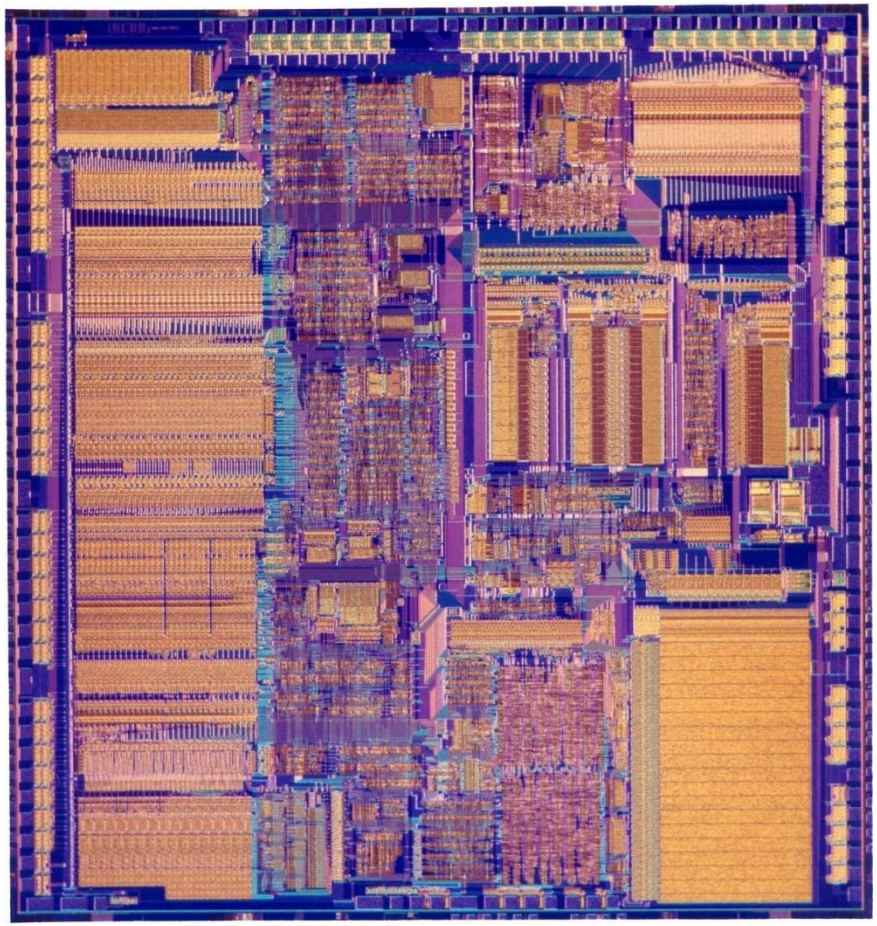
- 摩尔定律的失控及影响

Intel公司第一代CPU－4004(1971)



1. 规模：2300个晶体管
2. 生产工艺：10μm
3. 最快速度：108KHz

Intel公司CPU – 386TM(1985)

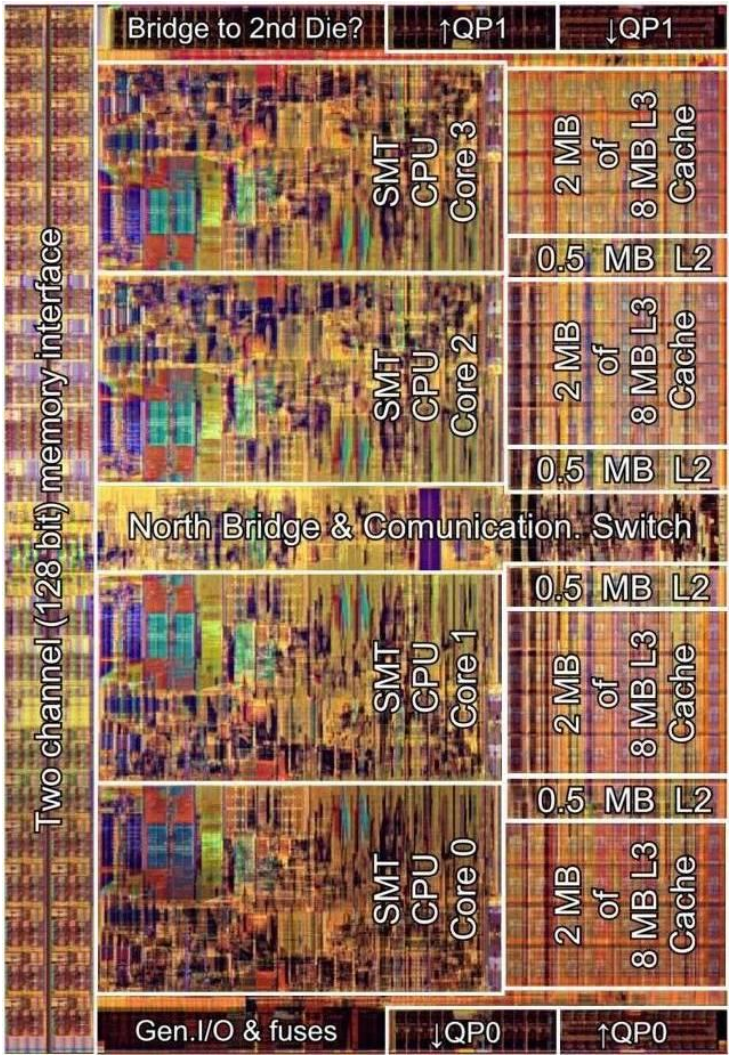


1. 规模：275000个晶体管
2. 生产工艺：1.5μm
3. 最快速度：33MHz

Intel公司CPU – Pentium®4(2000)

1. 规模：42×10^6个晶体管
2. 生产工艺：0.13μm
3. 最快速度：2.4GHz

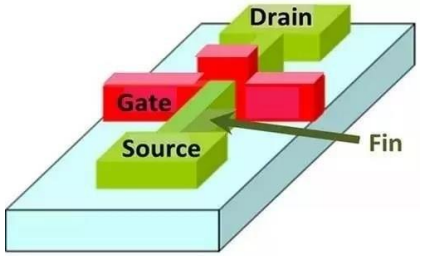
Intel公司CPU – CORE i7(2008)



1. 规模：7.31亿个晶体管
2. 生产工艺：45nm
3. 最快速度：3.33GHz

最新工艺

- Intel ® 酷睿™ i9-11900K
  - 14nm制造处理技术
  - 8核16线程处理能力
  - 处理器频率最高可达5.3GHz
- Xilinx公司Virtex UltraSCALE+系列VU57P
  - 14nm FinFET制造处理技术
  - 高达455Mb的UltraRAM存储器集成
  - 最高可达32.75Gb/s的单端收发带宽



集成电路的发展方向

- 开发超高速度、超高集成度的IC芯片。
- 发展各种ASIC，特别是开发更为复杂的片上系统(SoC, System on Chip)。
- SoC把处理机制、模型算法、软件、结构、各层次电路元器件的设计方法结合起来。
- 微机电系统(MEMS, Micro-Electro-Mechanical Systems)和生物信息技术将成为下一代半导体主流技术。

西安电子科技大学

1.2 集成电路的设计要求

对集成电路设计的主要要求

- 设计周期短(Time-to-Market)
- 设计正确率高(One-Time-Success)
- 芯片面积小、特征尺寸小
- 低功耗、 低电压
- 速度快
- 可测性好
- 成品率高
- 价格低

关于集成电路的速度

一般用单个门电路的最大延迟表示芯片的工作速度。

- 速度计算公式：

$$T_{pd} = T_{pdo} + U_L \frac{C_w + C_g}{I_p}$$

式中：

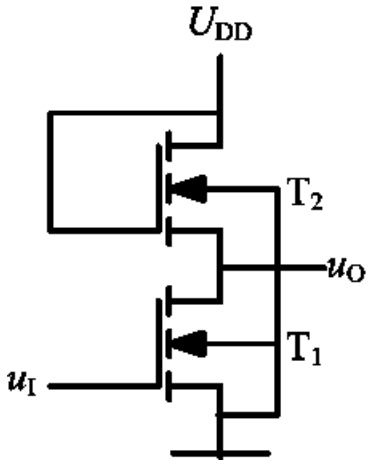
T_{pdo}	晶体管本征延迟时间；
U_L	最大逻辑摆幅，即最大电源电压；
C_g	扇出栅极电容(负载电容)；
C_w	内连线电容；
I_p	晶体管峰值电流；

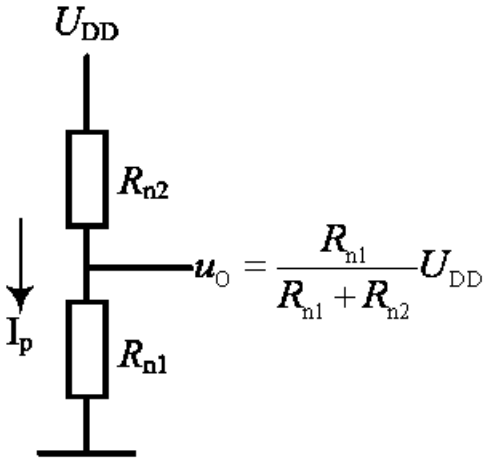
- 由上式可见，晶体管本征延迟越小，内连线电容和负载电容越小，电源电压越低、峰值电流越大，则芯片的延迟时间就越小，工作速度将有很大提高。

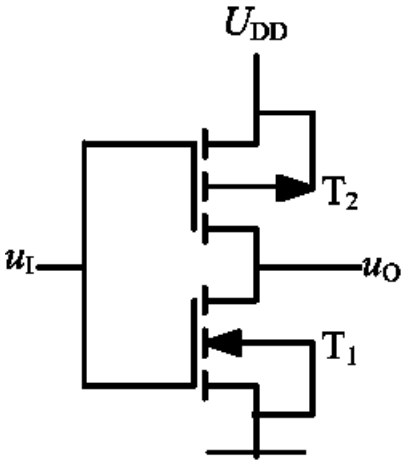
关于集成电路的功耗

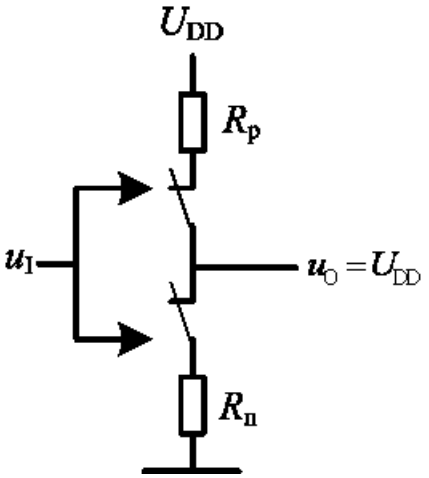
芯片的功耗与电压、电流大小有关，与器件类型、电路形式也关系密切。就MOS集成电路而言，有NMOS电路、PMOS电路和CMOS电路之分。

- 有比电路
- 无比电路









关于集成电路的功耗

- 静态功耗

静态功耗指电路停留在一种状态时的功耗。

有比电路的静态功耗为

$$P_{dQ} = I_P U_{DD}$$

无比电路的静态功耗为

$$P_{dQ} = 0$$

- 动态功耗

动态功耗指电路在(0,1)转换时对电路充、放电所消耗的功率。

无比电路的动态功耗为

$$P_d = fCU_L^2$$

C 各种电容总和；

f 信号频率；

U_L 电压摆幅(U_{DD})

关于集成电路的功耗

- 速度功耗积

由于集成电路的功耗与其工作速度有着密切的关系，因此引入“速度功耗积”来表示速度与功耗的关系，用信号周期表示速度，则速度功耗积为：

$$T \times P_{\mathrm{d}} = \frac{1}{f} \times fCU_{\mathrm{L}}^2 = CU_{\mathrm{L}}^2$$

当电源电压一定、电路电容一定时，若要速度高则功耗必然大。 反之，功耗小则速度必然低，二者乘积为常数。

关于集成电路的价格

性能价格比是集成电路的一项关键指标，如何降低集成电路的设计、生产与使用成本是非常重要的。

- 集成芯片的成本计算公式为

$$C_T = \frac{\text{设计成本及制版费}(C_D)}{\text{总产量}(N)} + \frac{\text{Wafer制造加工费用}(C_P)}{\text{成品率}(y)\times\text{Wafer芯片数}(n)}$$

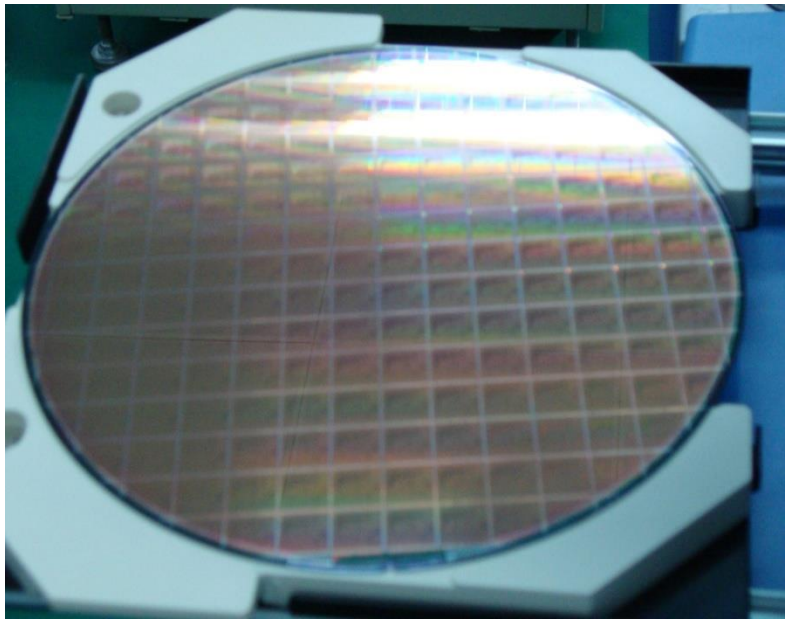
- 降低成本的措施

1. 批量大，总产量大
2. 提高成品率
3. 提高每个大圆片上的芯片总数(尽量缩小芯片尺寸)

关于集成电路的价格

- 缩小芯片面积采取的措施：

1. 优化逻辑设计；
2. 优化电路设计；
3. 优化器件设计，发展亚微米工艺和深亚微米工艺；
4. 优化版图设计，尽量充分利用芯片面积，合理布局/布线，减小连线长度，减小Wafer的无用区。



1.3 集成电路的分类

集成电路的分类

- 按功能分类
  - 数字集成电路
  - 模拟集成电路
  - 数、模混合集成电路
- 按结构形式和材料分类
  - 半导体集成电路
  - 膜集成电路(二次集成，分为薄膜和厚膜两类)
- 按有源器件及工艺类型分类
  - 双极型集成电路(TTL,ECL,模拟IC)
  - MOS集成电路(NMOS,PMOS,CMOS)
  - BiCMOS集成电路——双极型与CMOS混合集成电路

集成电路的分类

- 按集成电路的规模分类
  - 小规模集成电路(SSI)　　电路等效门：10~100
  - 中规模集成电路(MSI)　　电路等效门：100~1K
  - 大规模集成电路(LSI)　　电路等效门：1K~10K
  - 超大规模集成电路(VLSI)　　电路等效门：10K~100K
  - 甚大规模集成电路(ULSI)　　电路等效门：100K~

单片IC实现的电路功能	年代	等效电路规模
单个晶体管	1959	<1(等效门)
单一逻辑门	1960	=1
多功能逻辑	1962	2~10
复杂逻辑模块	1964	10~100 (SSI)
中等规模电路	1967	100~1000 (MSI)
大规模电路	1972	1000~10000 (LSI)
超大规模电路	1978	10000~100000 (VLSI)
甚大规模电路	1989	100000~ (ULSI)
片上系统	2000	100000~ (SOC)

各种电路规模的特征数据

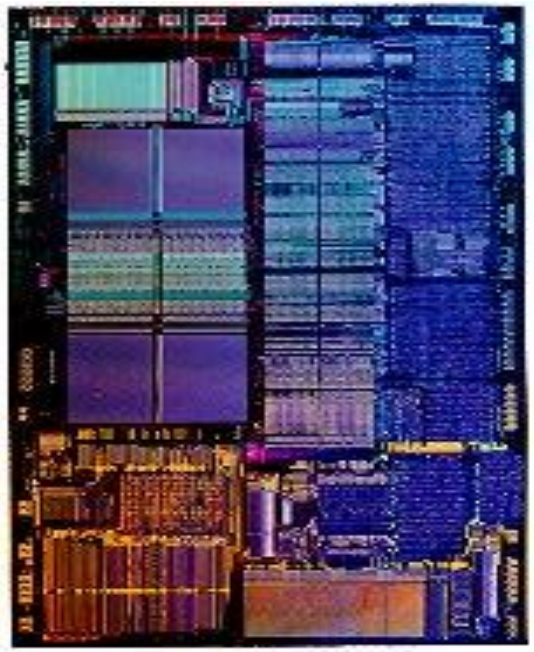
发展阶段 / 主要特征	MSI(1966)	LSI(1971)	VLSI(1980)	ULSI(1990)
元件数/芯片	10^2-10^3	10^3-10^5	10^5-10^7	10^7-10^8
特征线宽(*um*)	10-5	5-3	3-1	<1
速度功耗积(*uj*)	10^2-10	10-1	1-10^{-2}	<10^{-2}
栅氧化层厚度(*nm*)	120-100	100-40	40-15	15-10
结深(*um*)	2-1.2	1.2-0.5	0.5-0.2	0.2-0.1
芯片面积(mm^2)	<10	10-25	25-50	50-100
被加工硅片直径(*mm*)	50-75	100-125	150	>150

集成电路的分类

- 按生产目的分类
  - 通用集成电路(如CPU、存储器等)
  - 专用集成电路(ASIC)
  - 可编程器件
- 按实现方法分类
  - 全定制集成电路
  - 半定制集成电路
  - 可编程逻辑器件

全定制集成电路
(Full-Custom Design Approach)

从晶体管层次上对每个单元进行性能、面积的优化设计，力求做到芯片面积小、功耗低、速度快，达到性能/价格比的最优实现方法。

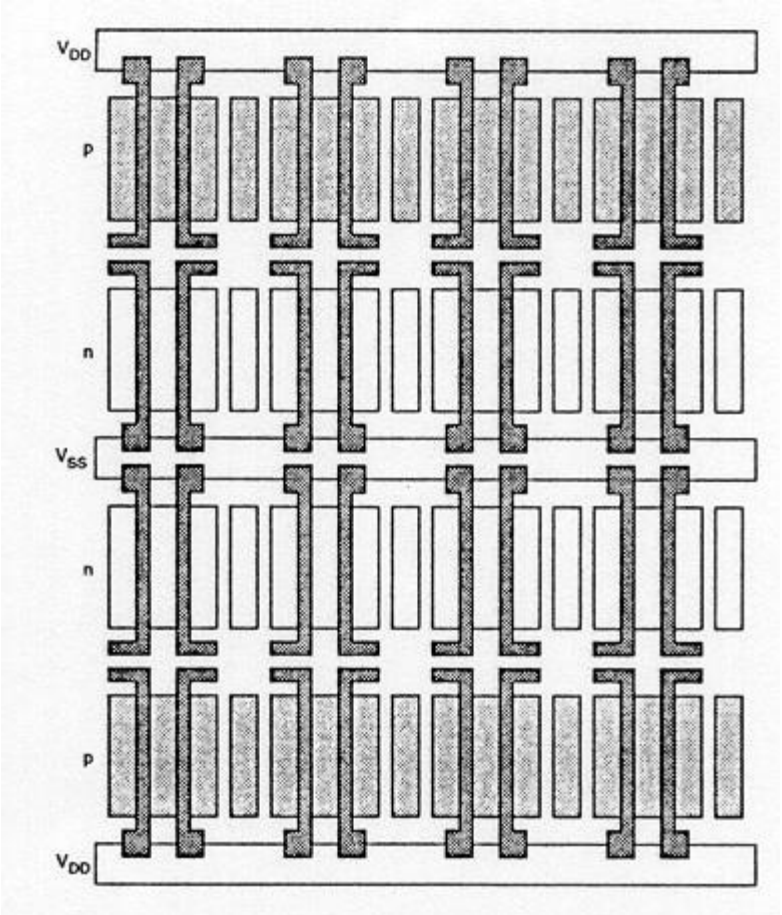


- 优点
  - 所设计电路的集成度最高
  - 产品批量生产时单片IC价格最低
  - 适于标准逻辑单元底层电路和模拟电路
- 缺点
  - 设计复杂度高/设计周期长
  - 设计成本及制版费用高
- 应用范围
  - 集成度极高且具有规则结构的IC(如各种类型的存储器芯片)
  - 对性能价格比要求高且产量大的芯片(如CPU、通信IC等)
  - 模拟IC/数模混合IC

半定制集成电路
(Semi-Custom Design Approach)

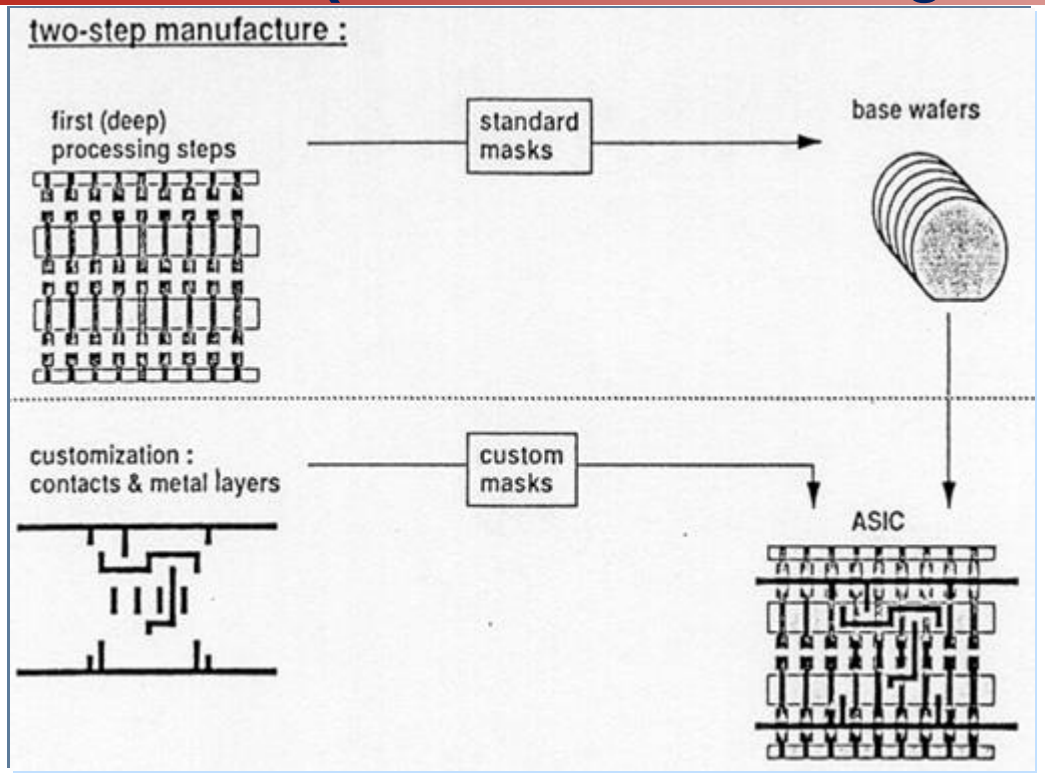
设计者在厂家提供的半成品基础上继续完成最终的设计，只需要生成诸如金属布线层等几个特定层次的掩模，设计周期短。

- 门阵列(GA: Gate Array)——有通道门阵列。将很多规则排列的晶体管用内连线连接起来，构成逻辑门阵列，阵列间有规则布线通道，由此形成门阵列母片。



半定制集成电路

(Semi-Custom Design Approach)



门阵列生产步骤：

(1) 母片制造

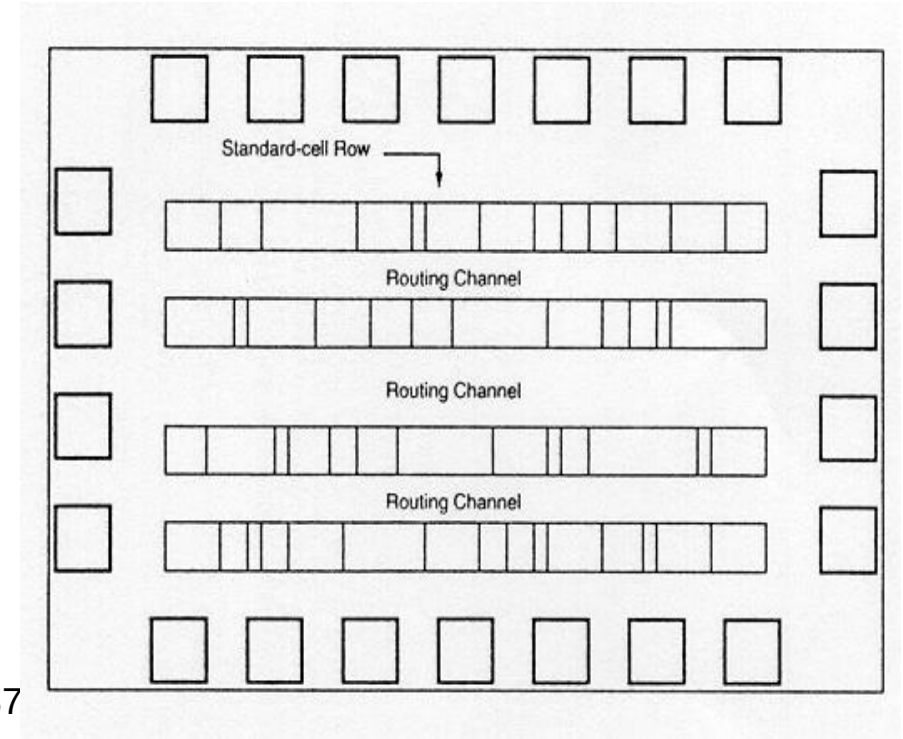
(2) 用户连接和金属布线层制造

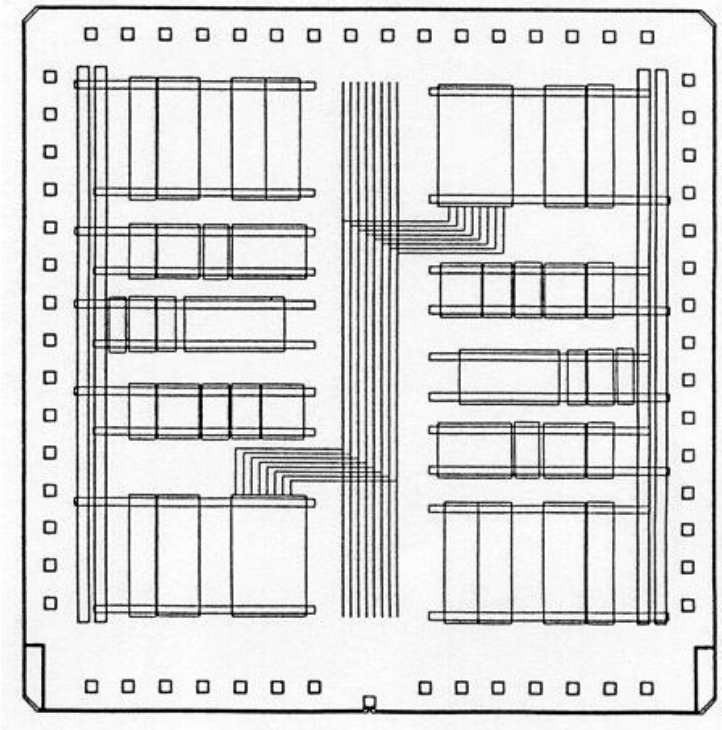
- 门海(SOG: Sea of Gate)——无通道门阵列。也是采用母片结构，它可以将没有利用的逻辑门作为布线区，而没有指定固定的布线通道，以此提高布线的布通率并提供更大规模的集成度。

半定制集成电路

(Semi-Custom Design Approach)

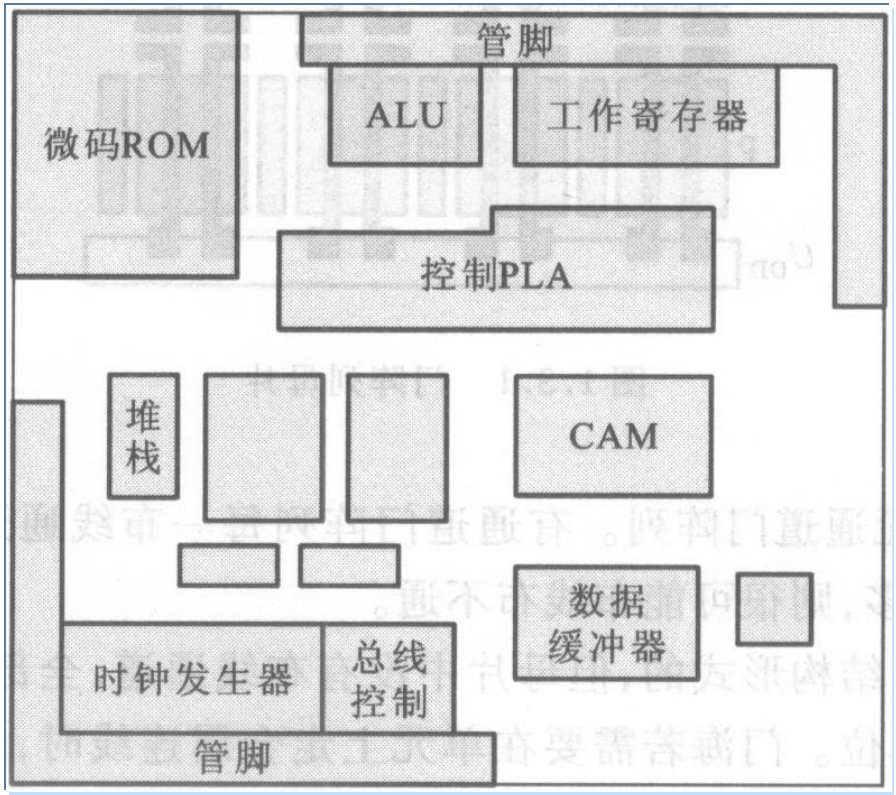
- 标准单元法(Polycell)——多元胞法。是指将电路设计中可能经常遇到的基本逻辑单元的版图按照最佳设计原则，遵照一定外形尺寸要求， 设计好并存入单元库中，需要时调用、拼接、布线。各基本单元的版图设计遵循“等高不等宽”的原则。





半定制集成电路
(Semi-Custom Design Approach)

- 积木块法——以已成熟的产品为单元，将整个芯片划分为若干模块，规定好各模块之间的接口，分别设计各模块，然后将它们“拼接”起来。



可编程逻辑器件

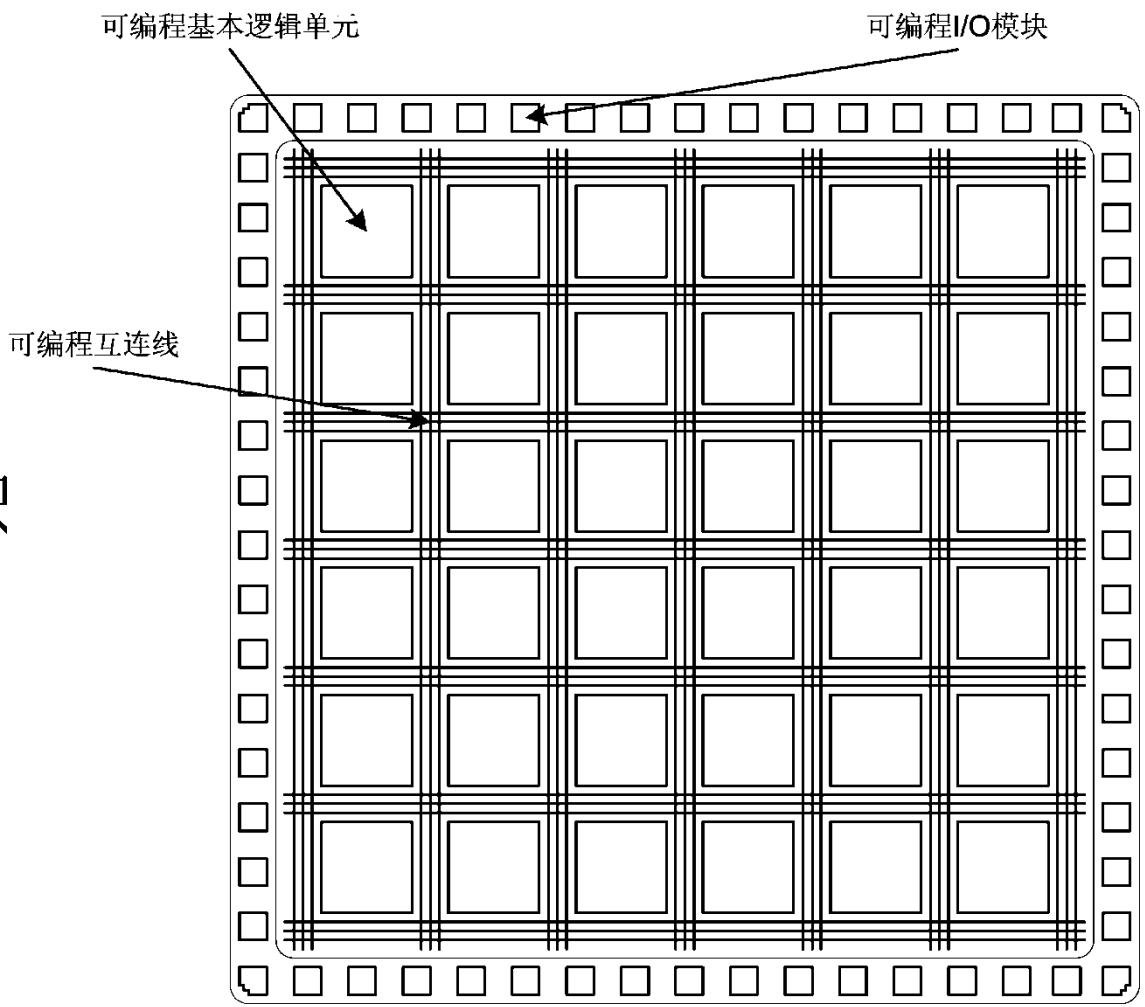
这种器件实际上也是没有经过布线的门阵列电路，其完成的逻辑功能可以由用户通过对其可编程的逻辑结构单元进行编程来实现。在集成度相等的情况下，其价格昂贵，只适用于产品试制阶段或小批量专用产品。

- 可编程逻辑器件分类

- PLD(Programmable Logic Device)
  - SPLD(Simple PLD)
    - PAL (Programmable Array Logic)
    - GAL (Generic Array Logi c)
    - EPLD(Erasable PLD)
  - CPLD(Complex PLD)
- FPGA(Field Programmable Gate Array)

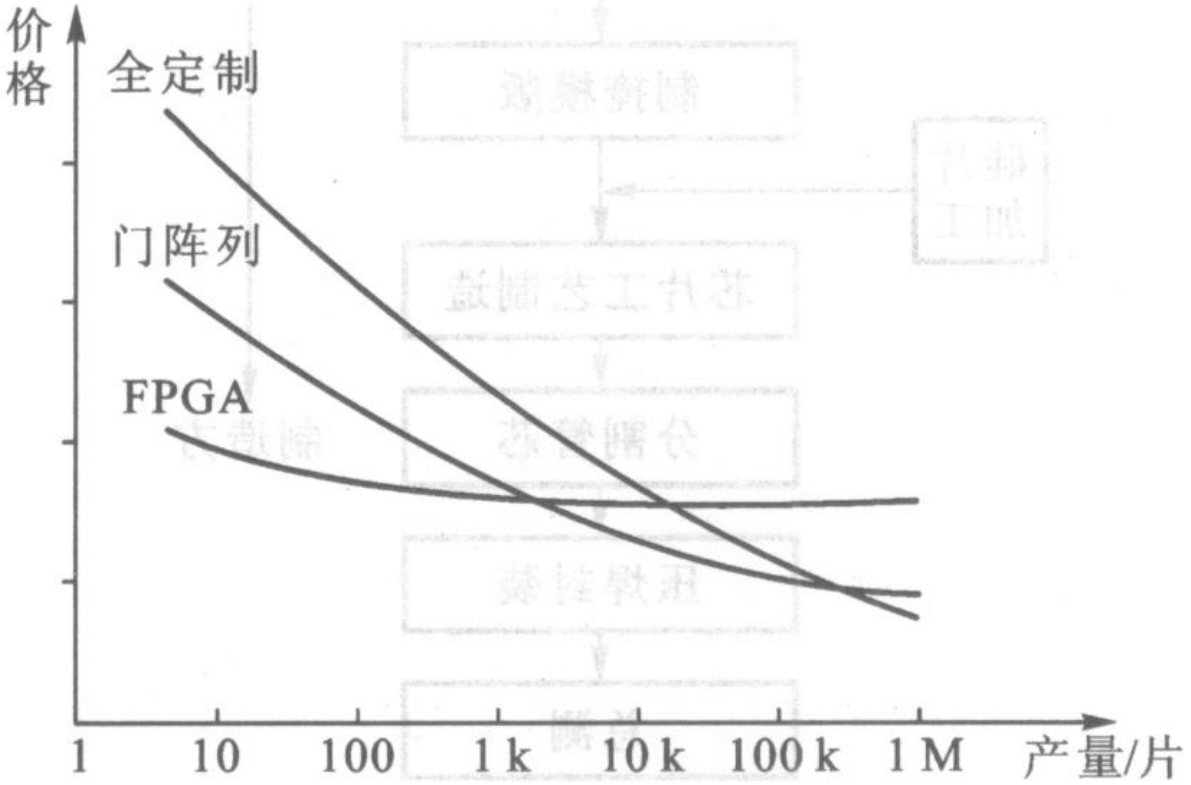
FPGA结构

- 可编程基本逻辑单元
- 可编程I/O模块
- 可编程互连线



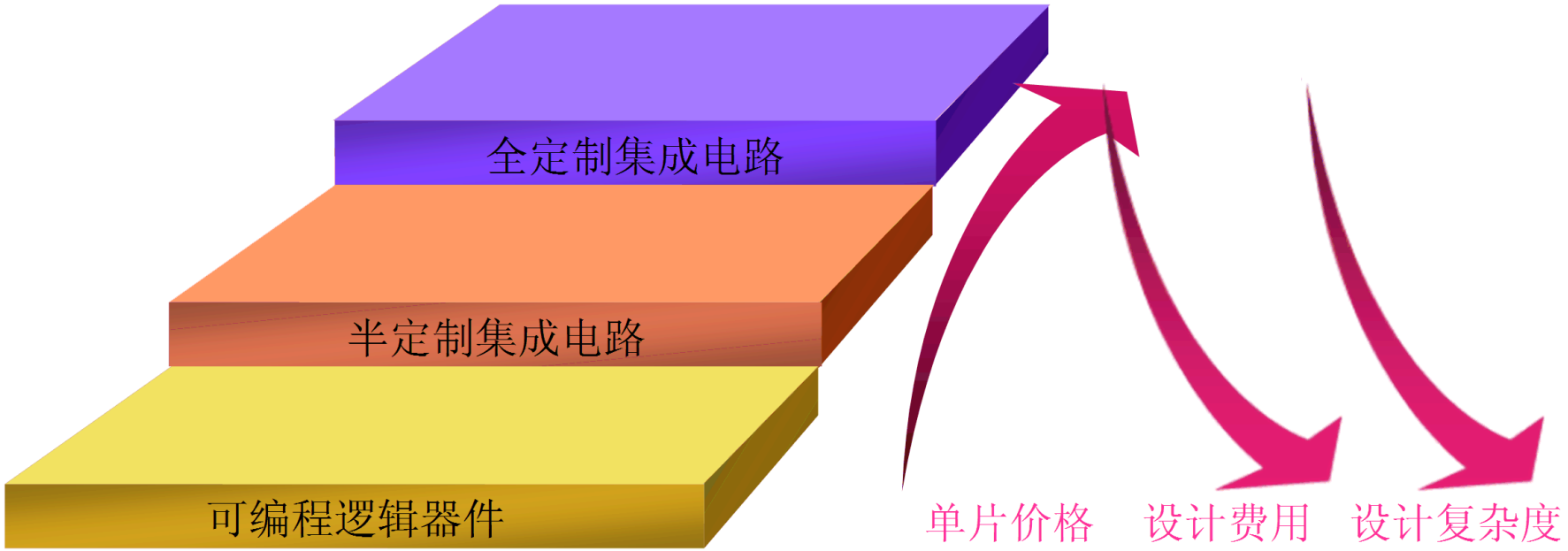
几种集成电路比较

- 不同产量时成本与设计方法的关系



几种集成电路比较

- 设计复杂度及费用比较



Altera公司及其产品简介

- Altera公司, 1983年, San Jose, California, USA

MAX	Cyclone	Stratix	Hardcopy
低成本、低功耗CPLD	低成本、低功耗FPGA	高密度、高端FPGA	低成本结构化ASIC
适用于便携式设备，常用于通信、计算机、消费电子、汽车和其他工业终端	嵌入式18×18 DSP乘法器，主要应用在通信、消费类电子、工业控制和汽车电子	集成嵌入式DSP模块、片内存储器、灵活的I/O	实现从Stratix系列FPGA原型到结构化ASIC的无缝移植

Xilinx公司及其产品简介

- Xilinx公司, 1984年, San Jose, California, USA

Virtex	Spartan	CoolRunner	XC9500
高密度、高性能FPGA	低成本FPGA	高速、高密度CPLD	低成本CPLD
在某些领域已经可以取代ASIC和ASSP (专用标准产品)	适于大批量应用，满足中等ASIC所占据的新兴大批量应用要求	低功耗、高速、高密度CPLD，对于低功耗设计有非常大的优势	提供了当今前沿系统设计所需的高性能、丰富的组合性能及灵活性

西安電子科技大學

1.4 集成电路设计方法

集成电路设计方法

- IC设计有别于板级(PCB, Printed Circuit Board)电路设计的主要方面

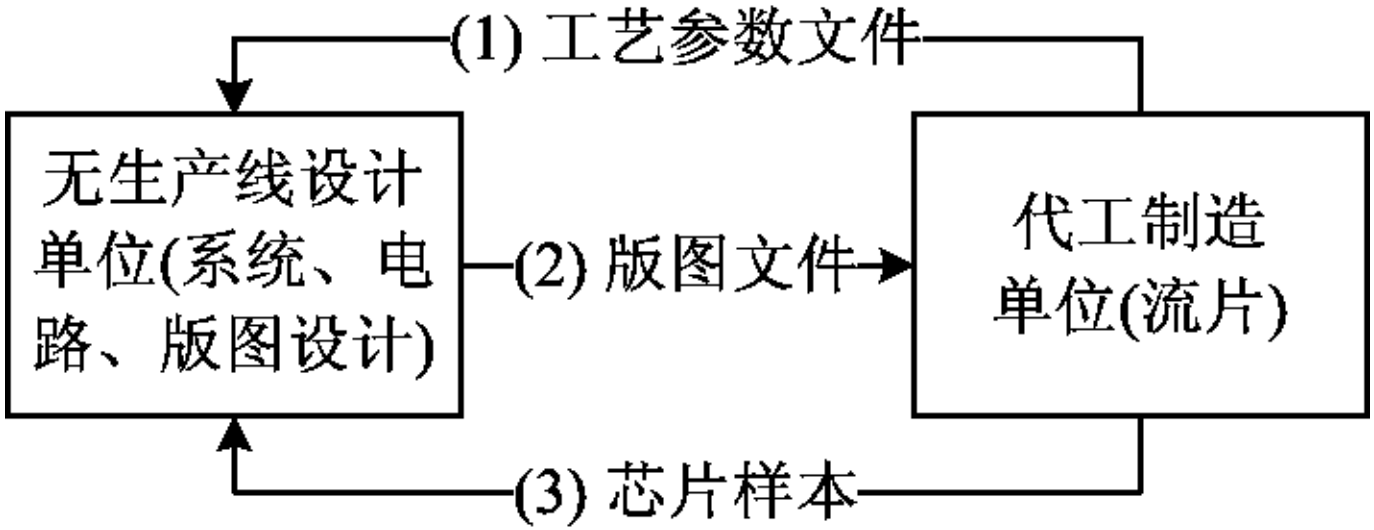
1. 设计层次不同
2. 所使用的设计/调试手段不同
3. 产品的最终结构形式不同
4. 开发费用/风险不同

- 成功IC设计所必备的条件

1. IC设计人员对所设计的电路与系统有充分的理解,并且具备扎实的电路理论功底和丰富的实践经验
2. 具有适当高效的EDA辅助设计软件并能够熟练应用
3. 有一整套完整可靠的设计方法和流程以确保设计中每一步骤的正确性
4. IC设计人员与生产厂家紧密配合

设计与加工分离

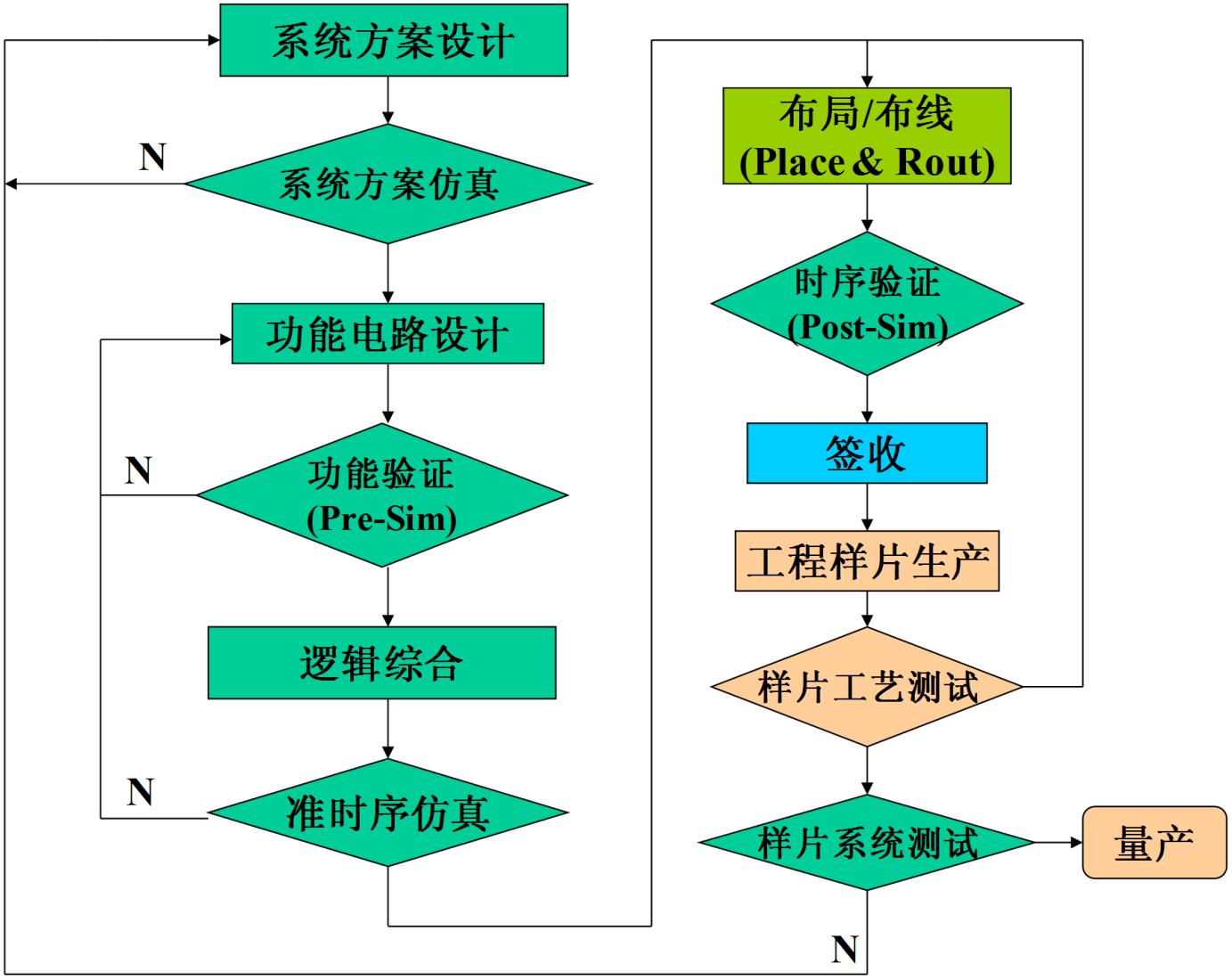
- 设计方 ⟷ 制造方
  - 设计方专注于电路与功能设计
  - 制造方专注于工艺实现



多项目晶圆(MPW)计划

所谓多项目晶圆(MPW, Multi Project Wafer)计划就是将几种至几十种工艺兼容的芯片拼装到一个宏芯片上，这样可使昂贵的制版和硅片加工费用由几十种芯片分担，从而大大降低了芯片研制成本。MPW技术服务更重要的意义在于在无生产线IC设计与代工制造之间建立了信息流和物流的多条公共渠道，将众多的无生产线IC设计单位(学校、研究所、中小企业)和代工制造单位联系起来，以最低的成本、最高的效率，促进微电子设计和制造产业的发展。

集成电路的设计流程


系统方案设计
N
系统方案仿真
功能电路设计
N
功能验证
(Pre-Sim)
逻辑综合
N
准时序仿真
布局/布线
(Place & Rout)
时序验证
(Post-Sim)
签收
工程样片生产
样片工艺测试
样片系统测试
量产
N

系统方案设计阶段

1. 需求分析

——技术文档

系统功能分析

输入/输出信号定义及时序描述

控制/状态寄存器描述

2. 确定电路总体结构形式

——建立系统数学模型和网络模型

3. 确定电路参数

功耗估算：

$$P = Cell \times F_{平均} \times K$$

$K = 0.75\mu W/MHz$(对应电源电压3.3V)

$0.30\mu W/MHz$(对应电源电压2.0V)

系统方案设计阶段

4. 与集成电路生产商沟通

——需要提供的设计信息

电源电压(V_{DD},V_{SS})；I/O引脚数；电平形式；输出电流；

电路规模/功耗；最高工作频率；RAM/FIFO描述；封装形式

——厂家的反馈信息

推荐的生产工艺；NRE(Non-Recurring Engineering)费用/样片数量/量产单价/加工周期；

质量保证/技术支持/中介服务

5. 确定生产工工艺，获取相关文件

——技术手册，工艺库文件(综合、仿真、器件模型)和专用开发工具

功能电路设计/验证(Pre-Sim)阶段

- 电路的设计

——设计方法

自顶向下设计(Top-Down Design)

独立于工艺的设计输入

功能模块划分

——设计原则

同步设计

时钟信号的处理/隔离异步时钟

复位信号的处理

——测试电路生成

- 设计验证(功能仿真)

——测试激励文件

——功能仿真

PC：ModelSim, VCS…

Workstation: Verilog-XL, NC-Verilog…

- 设计验证(FPGA验证)

逻辑综合/仿真阶段

该阶段仅在采用HDL语言描述输入时存在，它要求设计描述必须是可综合的，且必须给综合软件加载有厂方提供的综合工艺库，结果是ASIC电路的网表文件。

- 综合时的注意事项

——综合时必须确定相关工艺参数

——采用自底向上分模块，分层次进行

——注意综合过程中的提示信息以及最终的统计报告

——采用的EDA软件

PC ： Sinplify/Sinplify Pro, FPGA Compiler, …

Workstation: Synopsys ASIC Compiler (DC)…

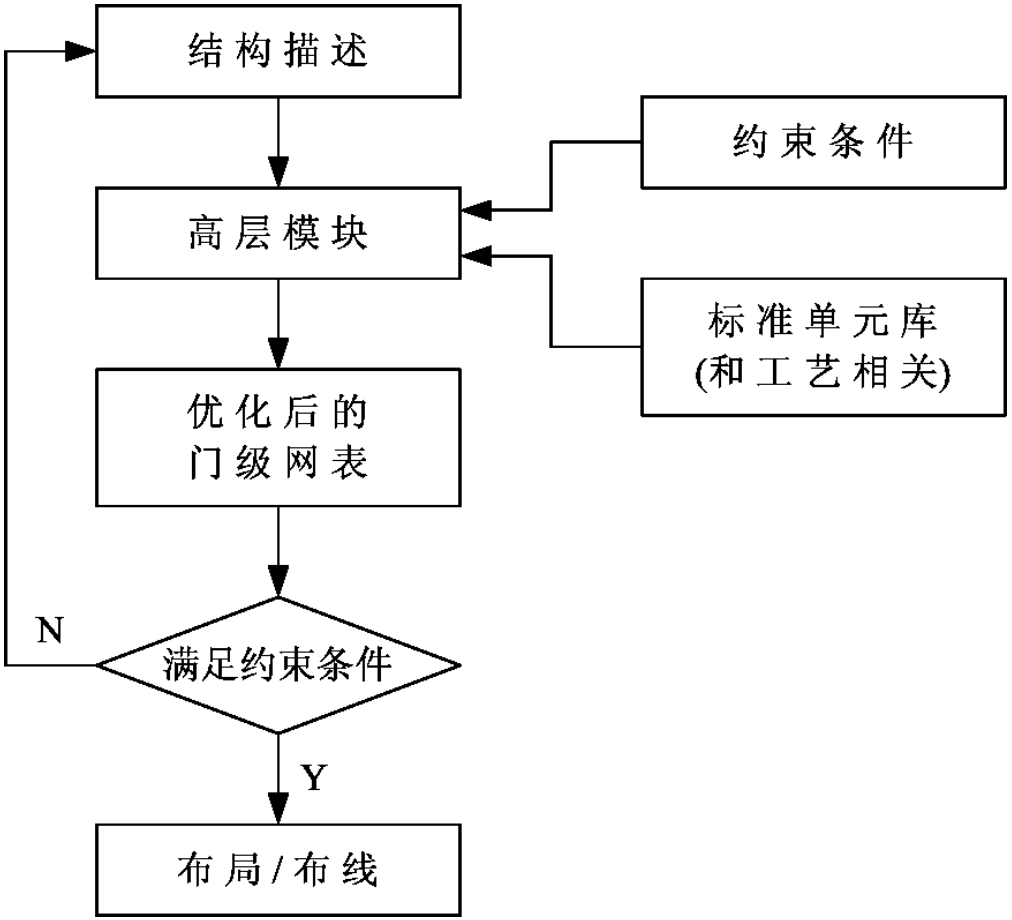
- 综合后的仿真

——是一种准时序仿真

——需要给仿真器加载仿真模型库和统计延迟文件(standard.sdf)

逻辑综合/仿真阶段

- 基本的计算机辅助逻辑综合流程图



布局/布线(P&R)——物理层设计阶段

该阶段的工作在半定制设计时由集成电路厂商负责。

- 前期准备工作

——将由综合器生成的网表文件送交由厂家提供的专用设计软件进行电学规则(ERC)检查，以此确保网表文件符合要求

——生成符合厂方格式的电路网表文件

——填写ASIC生产厂家提供的封装及引脚定义文件

——以草图的形式提供主要功能模块的布局，以供厂家参考

- P&R结束后，由厂方提供包含实际线延迟信息的延迟文件(.sdf)

布局/布线(P&R)——物理层设计阶段

- 物理设计所作的工作：

  ——平面规划(Plan)：在芯片上规划布置各个功能模块的位置

  ——布局(Place)：确定功能模块中每个电路单元的位置

  ——布线(Route)：连接电路中所有的信号连线

  ——参数提取(Parameter Extraction)：确定版图中各个节点处的寄生电容、电阻参数，它们对整个电路的功能和性能有很大的影响

  ——布局后的仿真(Post-Sim)：加入有参数提取获得的各种寄生电学参数后，再次确证电路设计的正确性，包括电学规则检查、设计规则检查和带寄生参数的仿真等

  ——形成标准的版图数据文件

时序仿真(Post-Sim)阶段

时序仿真是验证集成电路设计正确性最重要的一环。用以模拟所设计的电路在实际环境中的工作情况。

- 所需的文件

——提交给厂方的电路网表文件

——由厂方反馈的布线延迟文件(.sdf)

——测试激励文件

- 应做的工作

——三种工作状态(最佳/正常/最差)的仿真

——仿真结果的分析

三种状态下的仿真结果均正确无误，除了输出信号有一定的延迟外，应与功能仿真的结果一致。

- 工艺测试图案(Product Quality Test Pattern)

——特定时序仿真的输出结果送至厂家专用EDA工具后产生，对厂家产品质量的唯一测试方式。

最终签收(Sign Off)和样片生产

这是集成电路设计流程中最后的一步，设计方和生产方通过文件的互签来确认双方所完成的工作，该文一旦签署厂家即进入工程样片的生产。

- 必要的文件

——厂家对工艺测试图案的确认
——签收文件
——工程样片测试报告
注：厂方将只对不能满足工艺测试图案的工艺质量负责

- 样片在系统上测试

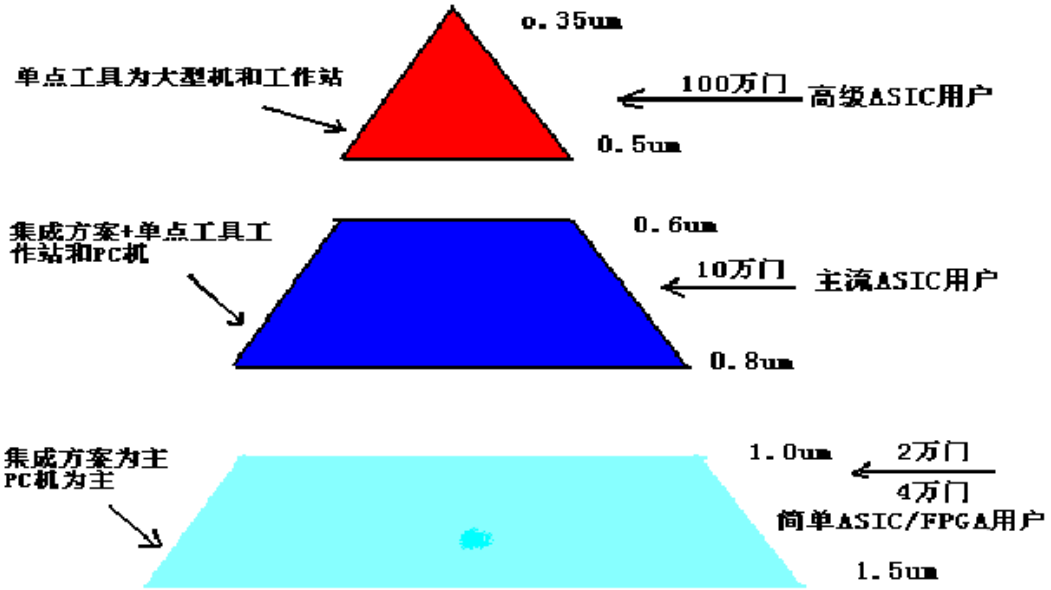
——通过测试，即可进入量产阶段
——测试失败，则需对故障进行分析，再重复整个过程(必须重新交纳NRE费用)

EDA设计工具的选择

- EDA: Electronic Design Automation

EDA供应商
- Cadence
- Synopsys
- Mentor Graphics
- Avant

- EDA工具和计算机的选择方案



西安电子科技大学

第二章

CMOS集成电路制造工艺基础及版图设计规则

本章提要

- 2.1 集成电路材料
- 2.2 基本的半导体制造工艺
- 2.3 CMOS工艺基础
- 2.4 版图设计规则
- 2.5 版图设计中的注意事项
- 2.6 版图检查

西安電子科技大學

2.1 集成电路材料

集成电路材料

集成电路材料 { 导体 / 半导体 / 绝缘体

- 导体在集成电路制造工艺中的功能

1. 构成低值电阻；电容极板；电感绕线；传输线；与轻掺杂半导体构成肖特基接触；与重掺杂半导体构成半导体器件的电极欧姆接触
2. 构成元器件间和层间的互联线
3. 构成与外界电路连接的焊盘

集成电路材料

- 绝缘体在集成电路制造工艺中的功能

1. 构成电容介质；构成MOS管的栅极与沟道间的绝缘层(栅氧)
2. 构成元件之间、互联线之间的横向隔离(场氧)
3. 构成不同工艺层面之间的纵向隔离，防止表面机械损伤和化学污染的钝化层

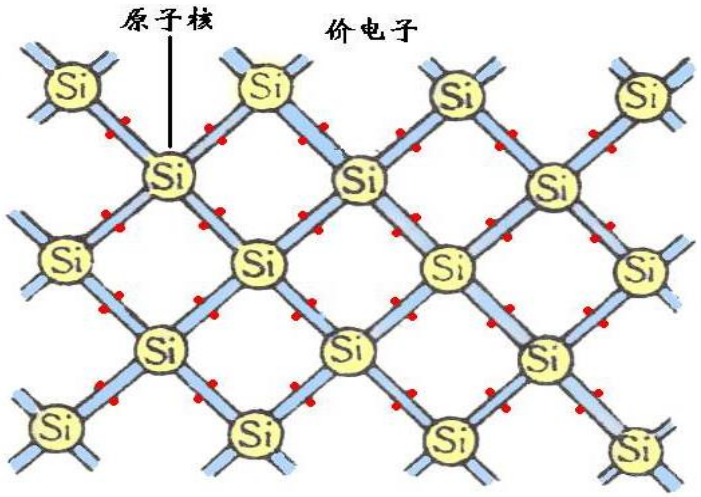
- 半导体在集成电路制造工艺中的功能
- 封装材料

西安電子科技大學

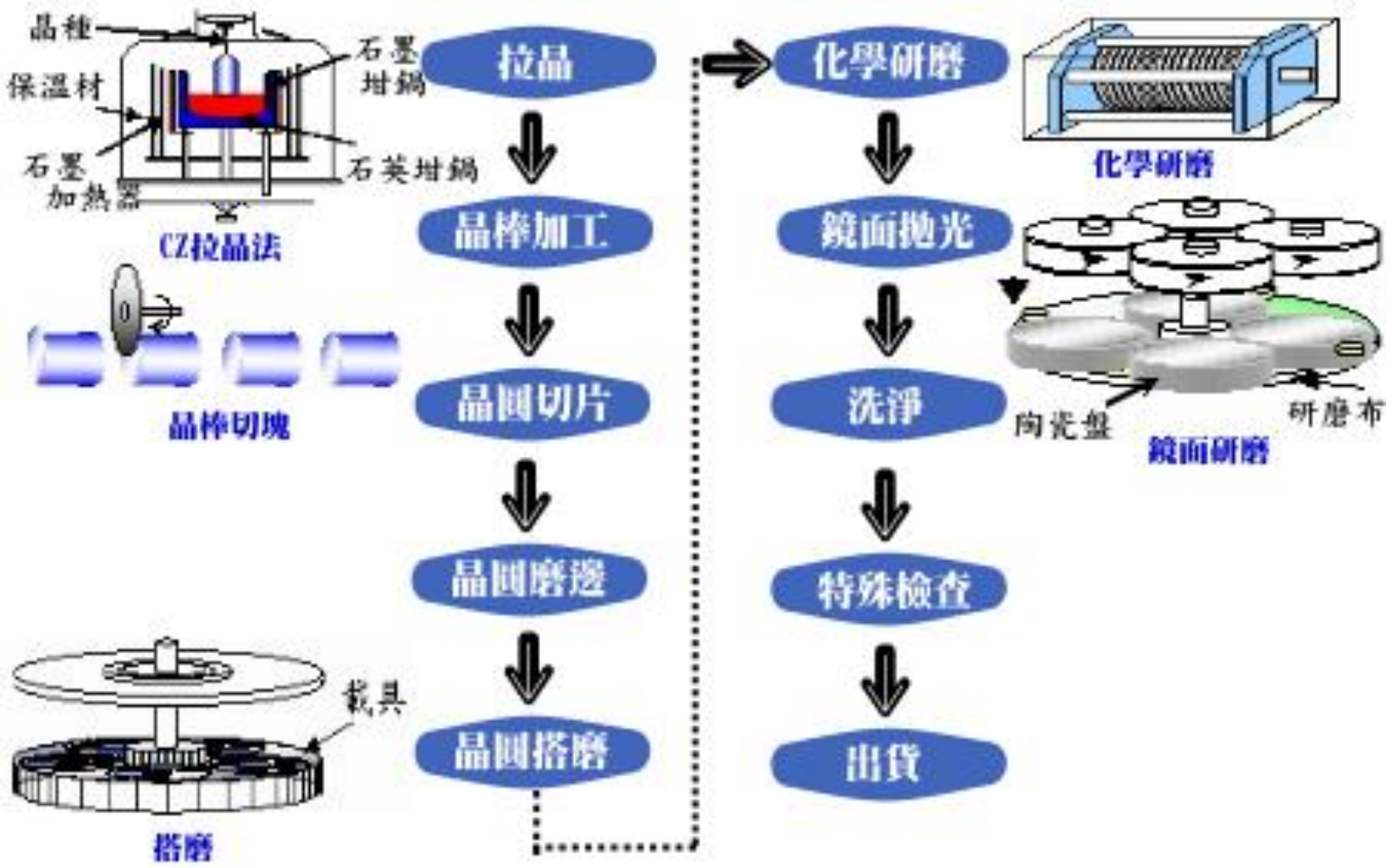
2.2 基本的半导体制造工艺

单晶硅和晶圆

- Wafer厚度0.5~0.7mm，直径100~300mm
- 掺杂后形成N型或P型衬底
- N型衬底电阻率3~5Ω·cm
- P型衬底电阻率12~14Ω·cm
- 外延工艺

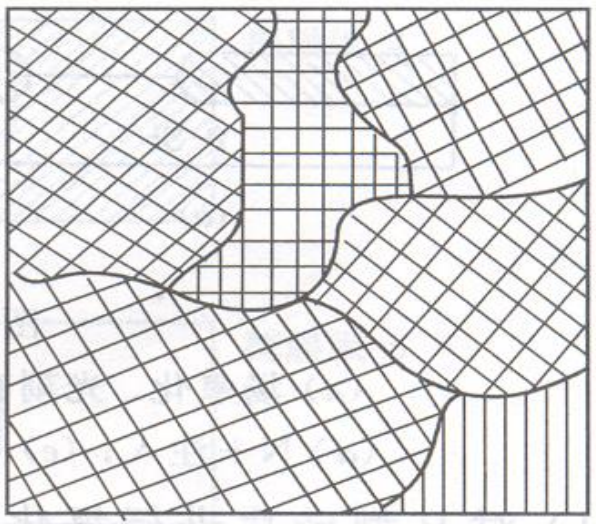


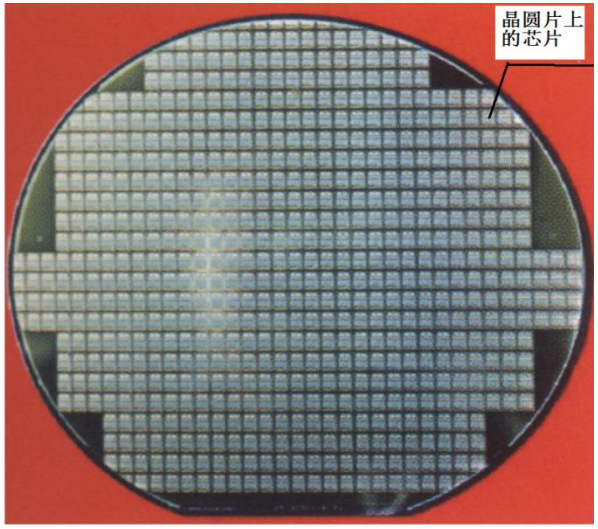
硅片的生产


晶種
保溫材
石墨加熱器
石墨坩鍋
石英坩鍋
CZ拉晶法
晶棒切塊
載具
搭磨
拉晶
晶棒加工
晶圓切片
晶圓磨邊
晶圓搭磨
化學研磨
鏡面拋光
洗淨
特殊檢查
出貨
化學研磨
陶瓷盤
研磨布
鏡面研磨

多晶硅

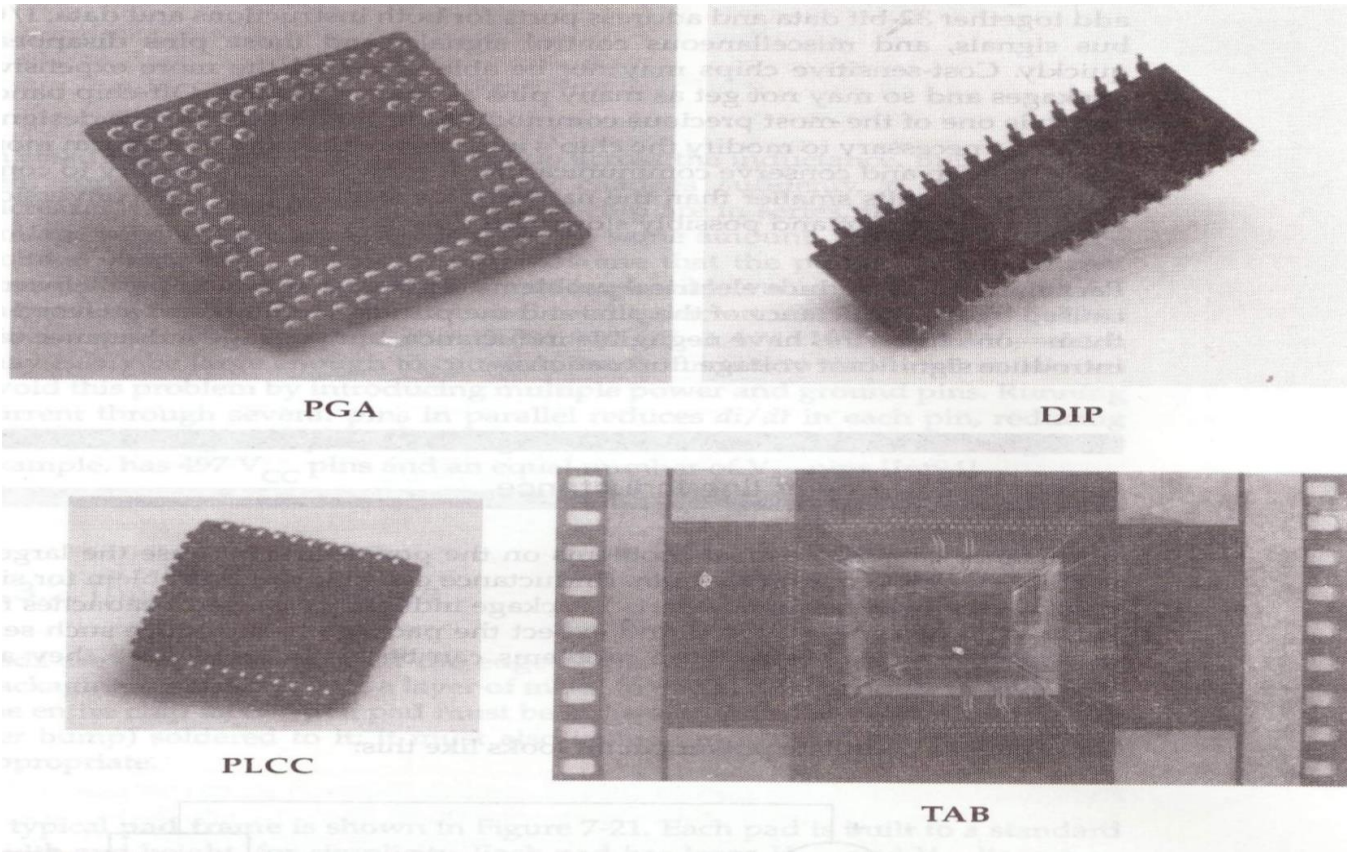
- 在小的区域内原子结构排列整齐，从整体上看并不整齐。
- 多晶硅的用途：
- 电阻
- 互联线
- MOS器件的栅极





封装与管芯

- 几种封装形式



封装与管芯

虽然不同器件的管芯各不相同，但它们都是由在半导体材料上形成的一些PN结所构成。因此，集成电路制造的关键问题就是根据设计要求，在半导体的不同区域形成所需要的PN结，这在生产上主要是通过氧化、掺杂、光刻等多种工艺的多次反复而形成。

氧化工艺

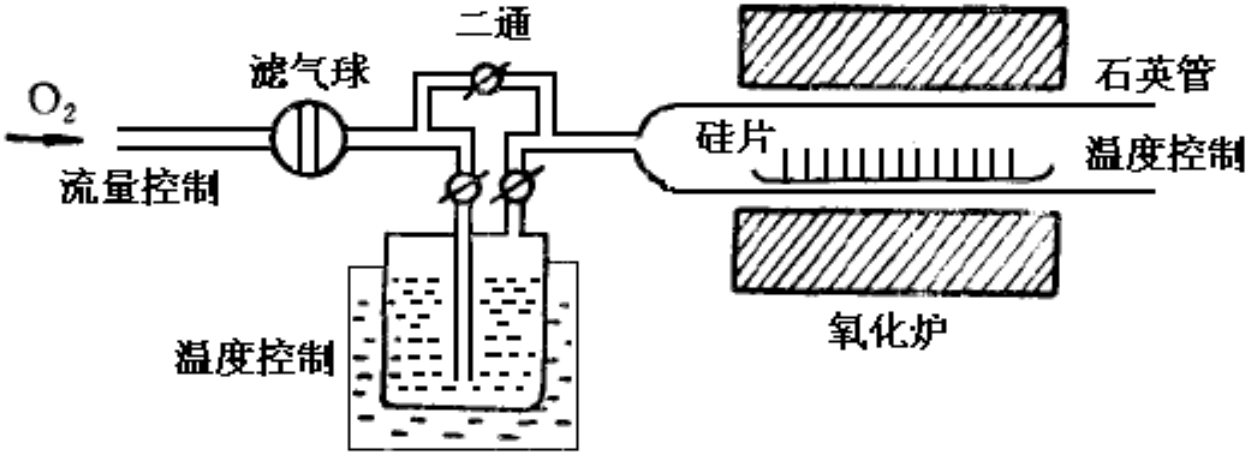
- 氧化工艺——在集成电路的制作过程中，要对硅反复进行氧化，制备SiO_2薄膜。
- SiO_2薄膜在集成电路中的作用

1. 作为对杂质选择扩散区域的掩膜
2. 作为MOS器件绝缘层的绝缘栅材料(栅氧)
3. 作为器件之间、层间的隔离材料(场氧)
4. 作为器件表面的保护(钝化)膜
5. 作为制作集成电路介质

热氧化法

- 热氧化原理与方法

热氧化生成SiO_2薄膜，就是将硅片放入高温(700℃~1200℃)的氧化炉内，然后通入氧气，在氧化气氛中使硅表面发生氧化生成SiO_2薄膜。



热氧化法

根据氧化环境的不同又可把热氧化分为干氧法和湿氧法两种。

- 干氧法：通入的氧化气体是纯氧气

$$Si + O_2 = SiO_2$$

- 优点：SiO_2薄膜结构致密、排列均匀、重复性好，不仅掩蔽能力强、钝化效果好，而且在光刻时与光刻胶接触良好，不宜浮胶。
- 缺点：生长速度太慢。

- 湿氧法：通入的氧化气体除氧气外还带有水汽

$$Si + O_2 = SiO_2$$

$$Si + 2H_2O = SiO_2 + 2H_2\uparrow$$

- 优点：生成SiO2薄膜的速度快。
- 缺点：SiO_2薄膜的表面有硅烷醇的存在，使得它在光刻时与光刻胶接触不良，容易产生浮胶。

$$SiO_2 + H_2O \rightarrow 2(Si\text{-}OH)$$

SiO_2薄膜的致密性显然也不如干氧法。

热氧化法

生产中一般不单独采用某一种方法，而是将两种方法结合起来，采用干氧——湿氧——干氧交替的氧化方式，即在氧化开始时先通一段时间纯氧气(干氧)，然后再加入水汽进行湿氧，湿氧结束后再通一段时间纯氧气。这样就可使湿氧结束后SiO_2薄膜表面的硅烷醇(Si-OH)重新变为SiO_2，明显改善了SiO_2薄膜与光刻胶的接触性能，提高了SiO_2薄膜的质量。

化学气相淀积法

- 将一种或几种化学气体，以某种方式激活后在衬底表面发生化学反应，从而在衬底表面生成SiO_2薄膜，主要用硅烷(SiH_4)和氧反应，或用烷氧基硅分解成SiO_2。

掺杂工艺

在半导体基片的一定区域掺入一定浓度的杂质元素，形成不同类型的半导体层，来制作各种器件。

- 扩散工艺——利用物质微粒总是从浓度高处向浓度低处作扩散运动的特性，实现掺杂目的的工艺。
- 三种扩散类型

交换扩散　　填隙扩散　　空位扩散

点阵中相邻原子交换位置

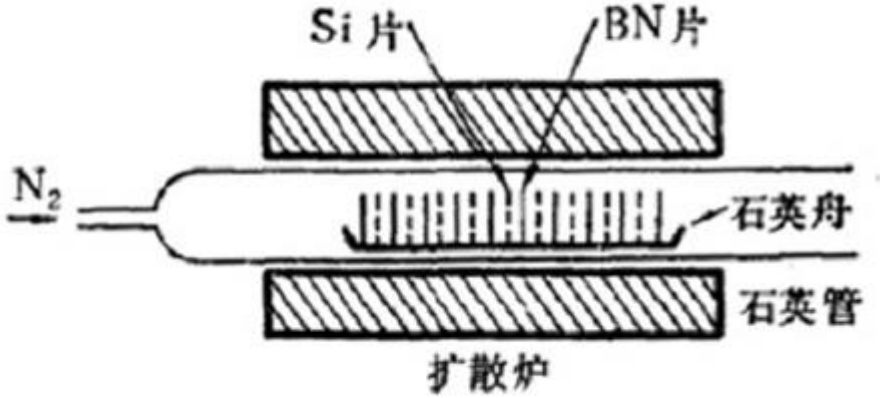
扩散粒子在点阵间隙中运动

点阵中的原子会离开原位成为填隙原子，留下来的空位由附近的原子填补而形成的空位移动和原子运动

扩散工艺

- 杂质浓度在硅中的大小与分布是温度和时间的函数。
- 实际生产中杂质的扩散，往往先进行恒定表面源扩散，这一步工序称为预淀积，在预淀积的基础上，再进行有限表面源扩散，这一步工序常常称为再分布。
- 常用扩散方法

1. 液态源扩散
2. 片状源扩散
3. 固—固扩散
4. 涂层扩散



离子注入工艺

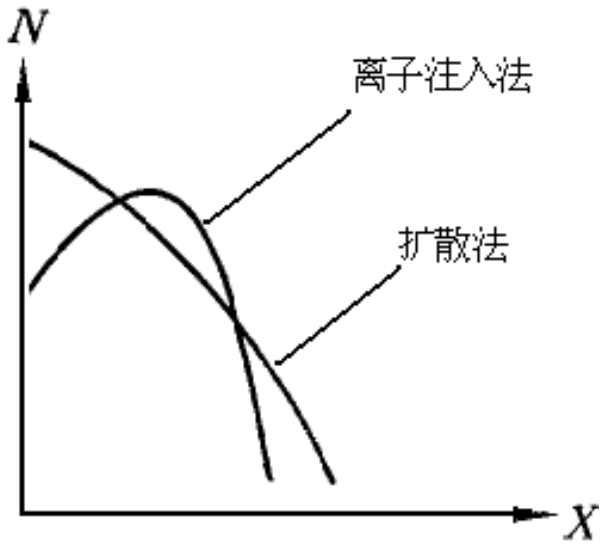
将杂质元素的原子离子化，使其成为带电的杂质离子，然后用电场加速这些杂质离子，使其获得极高的能量并直接轰击半导体基片。当这些杂质离子进入半导体基片后，受到半导体原子的阻挡停了下来，这样就在半导体基片内形成了一定的杂质分布。

- 离子注入工艺的优点

1. 注入温度低(约400℃)
2. 通过控制注入的电学条件可精确控制掺杂的浓度和结深
3. 杂质浓度不受材料固溶度的限制
4. 可采用离子注入的元素种类多，注入纯度高
5. 可实现大面积薄而均匀的掺杂，横向扩散小

离子注入工艺

离子注入结束后，还要对半导体基片进行退火处理，这是因为高能量的杂质离子进入硅片使得一部分硅原子离开了原来的位置，造成晶格损伤，杂质离子也不是正好处在原来硅原子的位置上。退火通常是在氮气的保护下使硅片在一定温度下保持一段时间，从而使晶格恢复，也使杂质离子进入替代硅原子的位置而激活，起到施主或受主的作用。



注入的杂质离子的分布近似为对称高斯分布，杂质浓度最大的地方离硅片表面有一定距离。

掩模的制版工艺

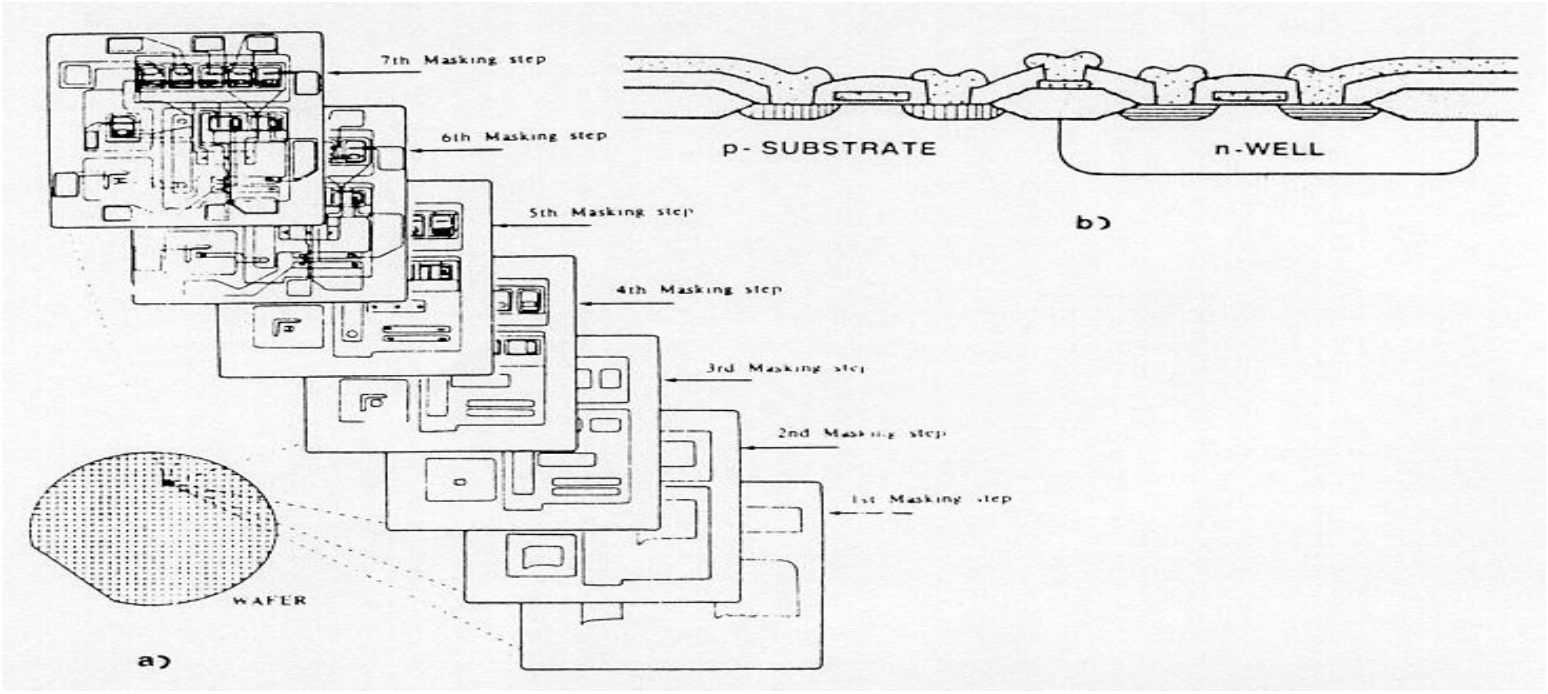
- 掩模制版——将各种元器件的版图转换到硅片上。
- 掩模版制版方法

1. 图形发生器(pattern generator)法
2. X射线法
3. 电子束扫描法

- 掩模版的基本要求：极板平整坚固、热膨胀系数小、图像尺寸准确无畸变等。

光刻工艺

借助于掩膜版，并利用光敏抗蚀涂层发生光化学反应，结合刻蚀方法在各种薄膜上(如SiO_2薄膜、多晶硅薄膜和各种金属膜)刻蚀出各种所需要的图形，实现掩膜版图形到硅片表面各种薄膜上图形的转移。



光刻工艺

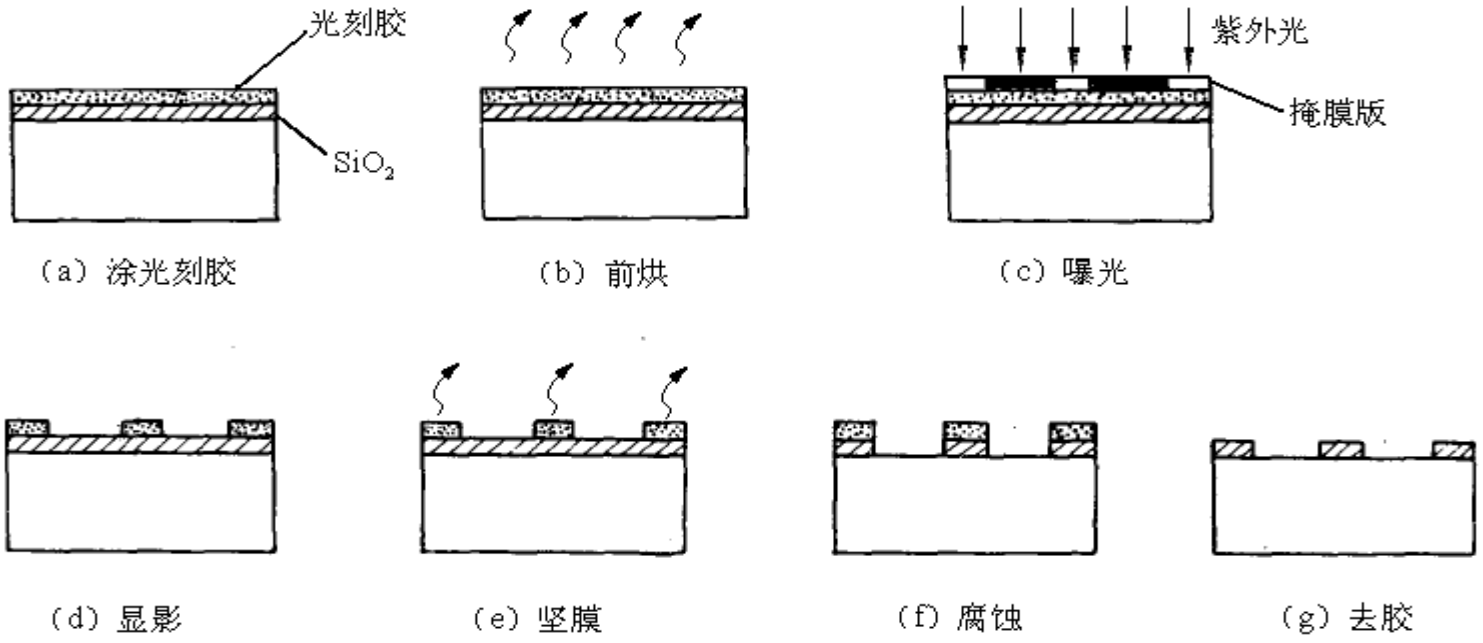
- 光刻系统的组成：曝光机、掩膜版、光刻胶
- 主要指标：
  - ✓ 分辨率W(resolution)—光刻系统所能分辨和加工的最小线条尺寸
  - ✓ 焦深(DOF-Depth of Focus)—投影光学系统可清晰成像的尺寸范围
  - ✓ 关键尺寸(CD-Critical Dimension)控制
  - ✓ 对准和套刻精度(Alignment and Overlay)
  - ✓ 产率(Throughout)
  - ✓ 价格

光刻工艺

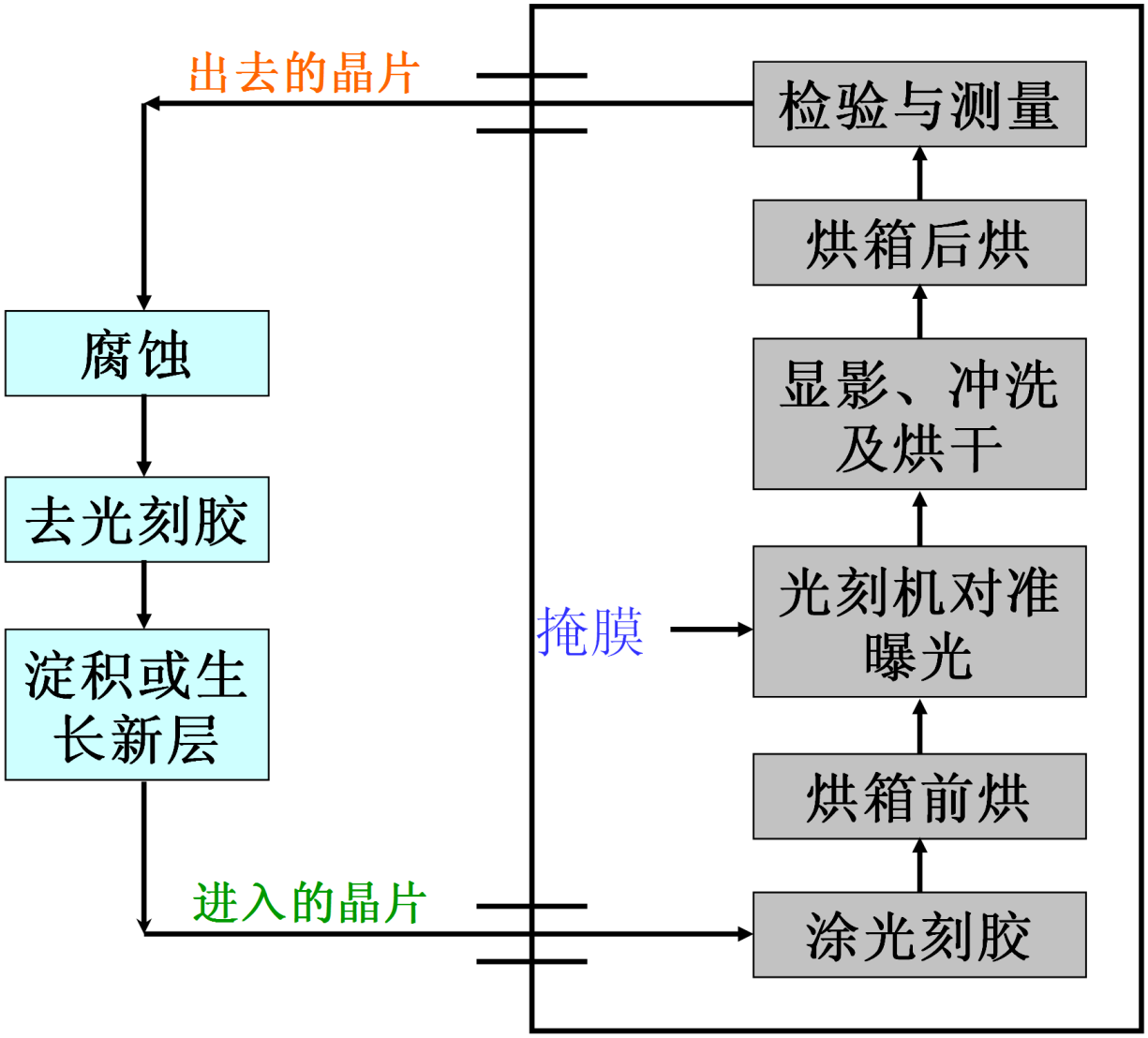
- 两种类型的光敏抗蚀剂——光刻胶
- 正胶：光刻胶膜本来不能被溶剂所溶解，只有当受到适当波长的光照射后(如紫外光)发生光分解反应，变为可溶性的物质，这种胶称为正胶。
- 负胶：光刻胶膜本来可以被溶剂所溶解，只有当受到适当波长的光照射后(如紫外光)发生光聚合反应而硬化，变为不可溶性的物质，这种胶称为负胶。

光刻工艺

- 曝光机：1）将硅圆片与掩膜严格定位；2）为抗蚀剂提供曝光用辐射源。其技术指标有：分辨率、对准度、吞吐量。曝光机是VLSI制造中最为关键的设备之一。



光刻工艺


出去的晶片
检验与测量
烘箱后烘
显影、冲洗及烘干
光刻机对准曝光
掩膜
烘箱前烘
涂光刻胶
进入的晶片
淀积或生长新层
去光刻胶
腐蚀

金属化工艺

- 金属化工艺——完成电极、焊盘和互联线的制备。
- 金属化工艺是一种物理气相淀积，需要在高真空系统中进行。
- 常用方法

1. 真空蒸发法
2. 溅射法

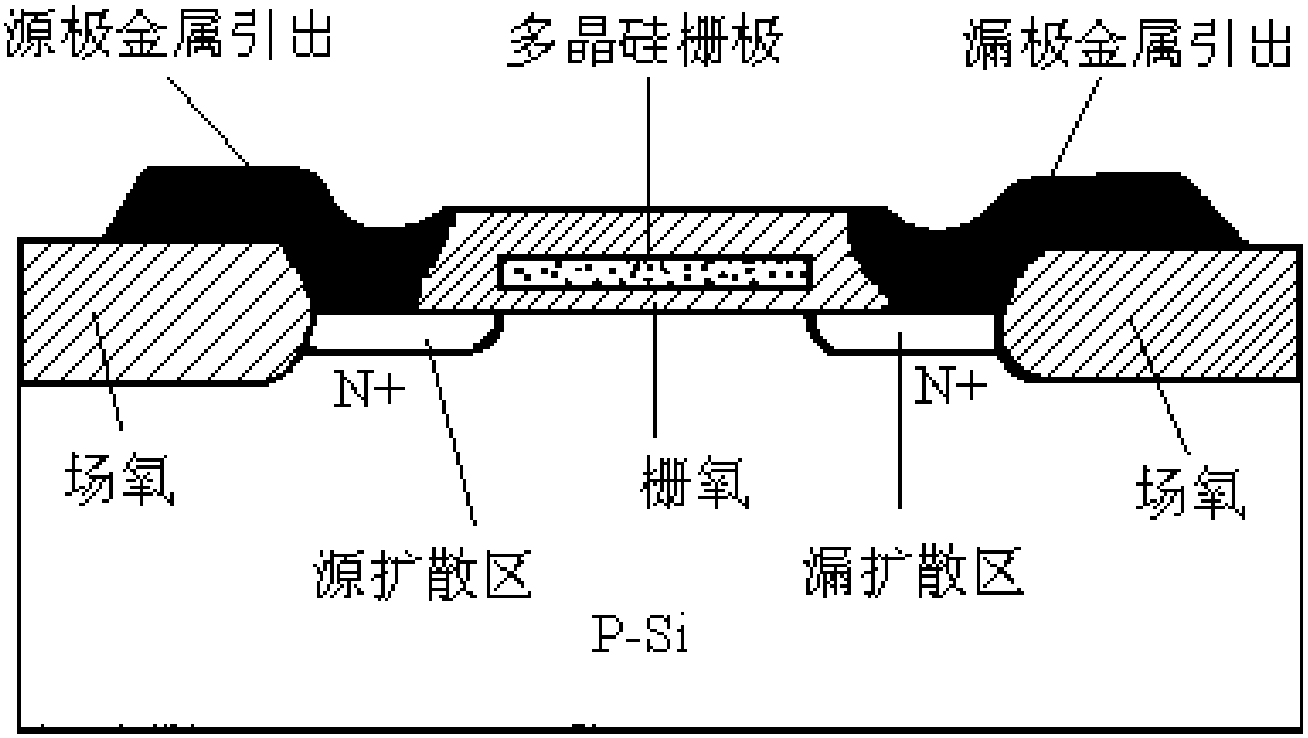
西安電子科技大學

2.3 CMOS工艺基础

自对准技术和标准硅栅工艺

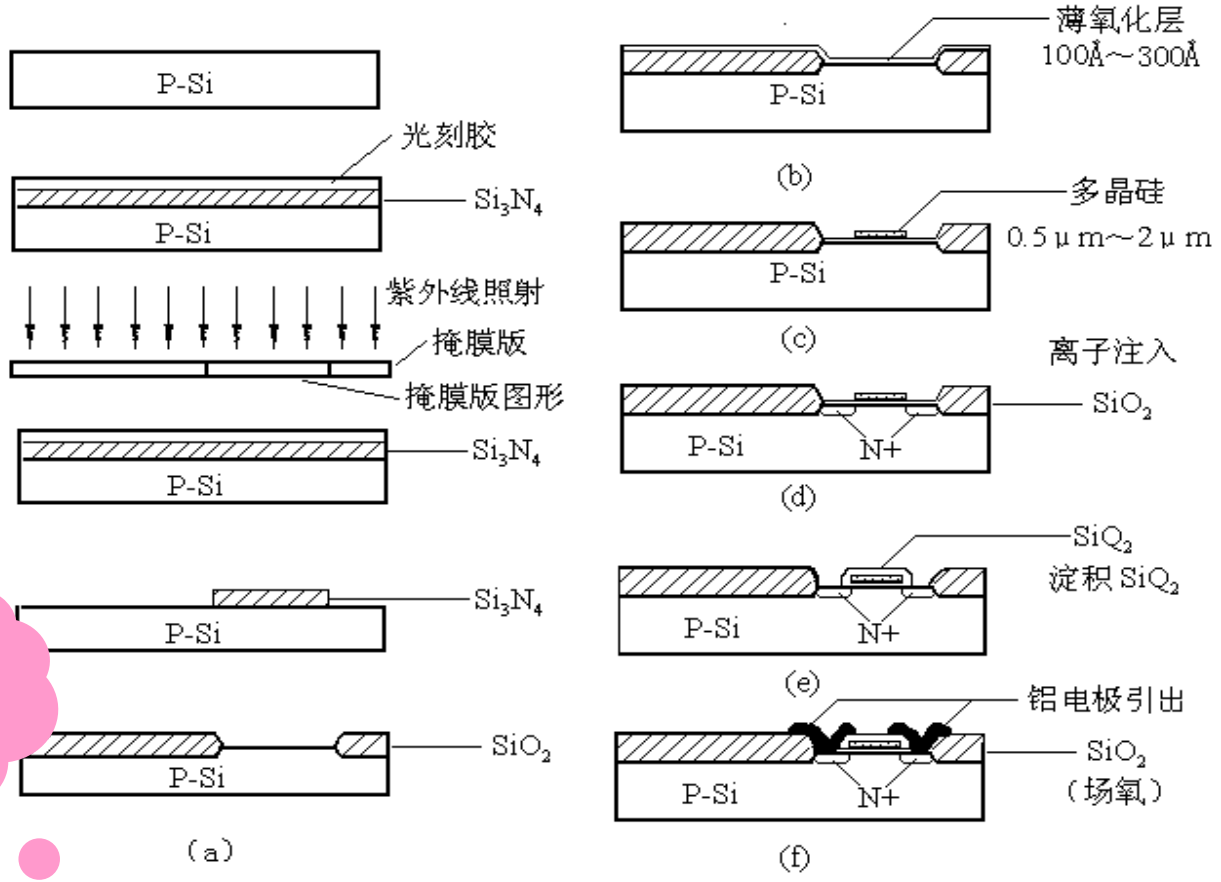
- 硅栅工艺也叫自对准工艺，它有利于减小栅—源和栅—漏之间的覆盖电容。
- 有源区是制作MOS晶体管的区域。硅栅工艺是先做栅极再做源、漏区，这是硅栅工艺和铝栅工艺的根本区别。由于先做好硅栅再做源漏区掺杂，栅极下方受多晶硅栅保护不会被掺杂，因此在硅栅两侧自然形成高掺杂的源、漏区，实现了源—栅—漏的自对准。

自对准技术和标准硅栅工艺



硅栅NMOS管剖面图

自对准技术和标准硅栅工艺



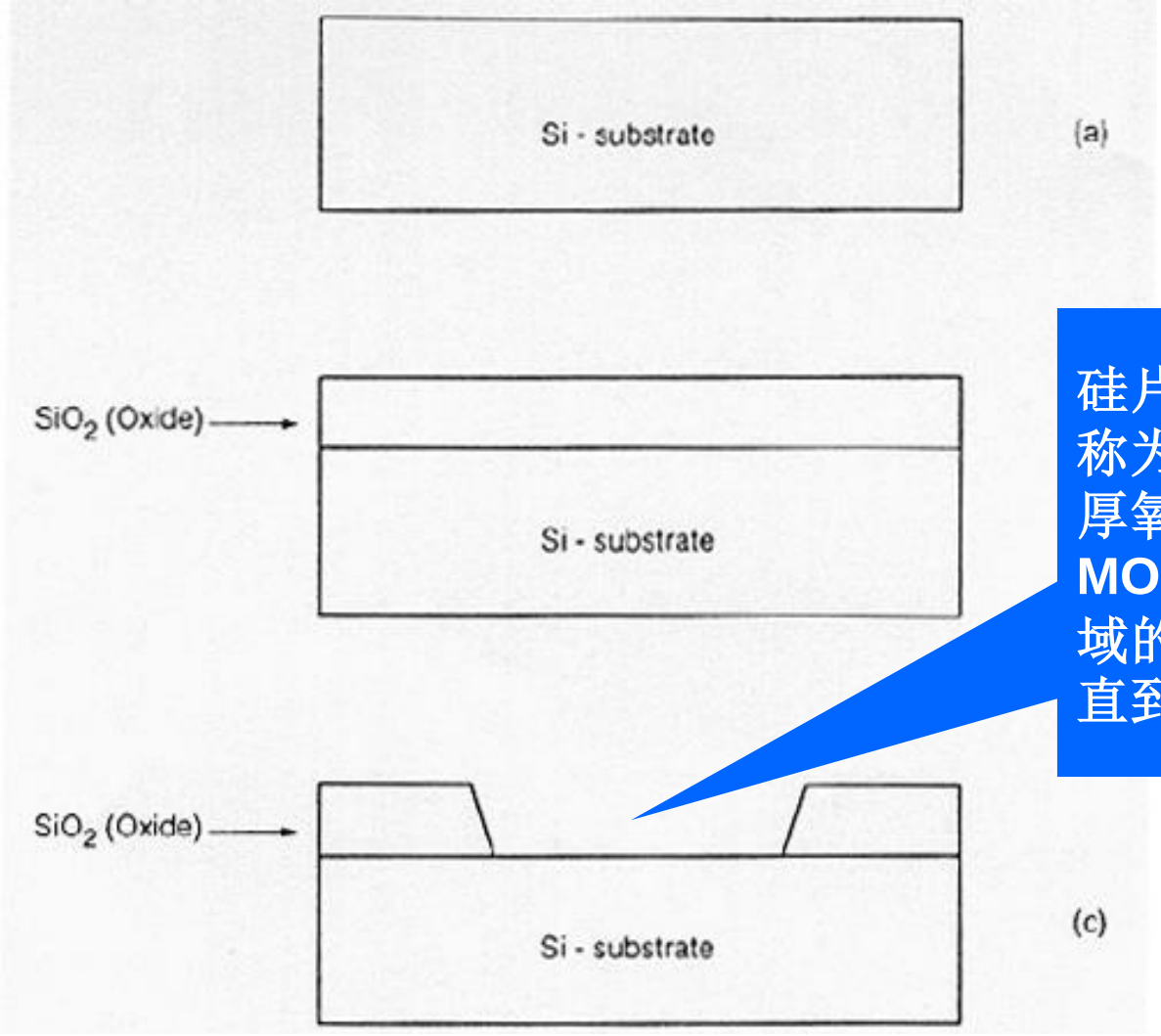
(a)中有什么问题？

硅栅NMOS管的制造工序

(a)场氧化、光刻有源区 (b) 栅氧化 (c) 淀积多晶硅，刻多晶硅

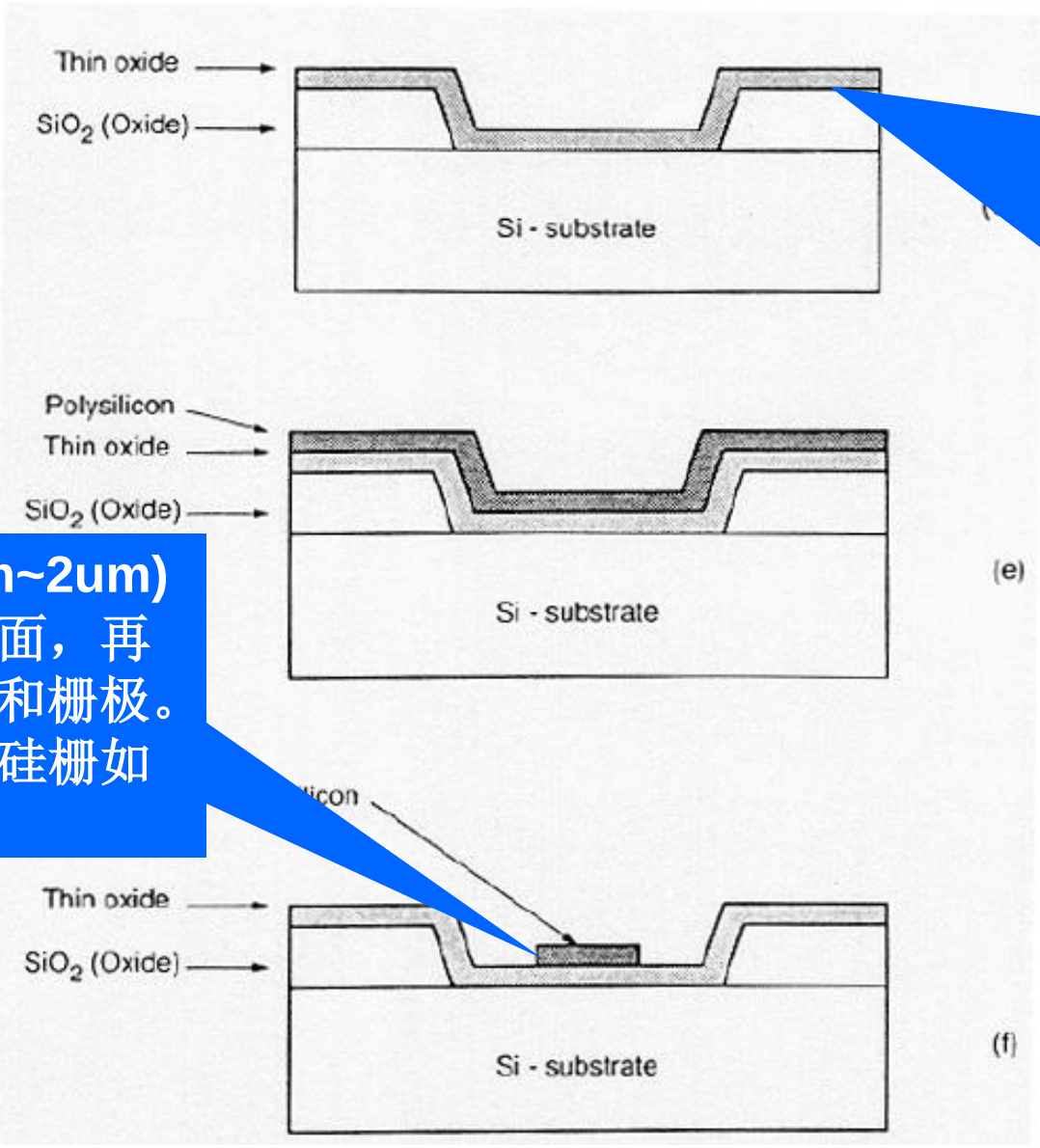
(d) N+注入 (e) 淀积SiO2，刻接触孔 (f) 蒸铝，刻铝电极和互连

硅栅MOS工艺简介



硅片上开始有一层称为场区氧化层的厚氧化层。要形成MOS管的那个区域的场氧被刻蚀掉，直到出现硅表面。

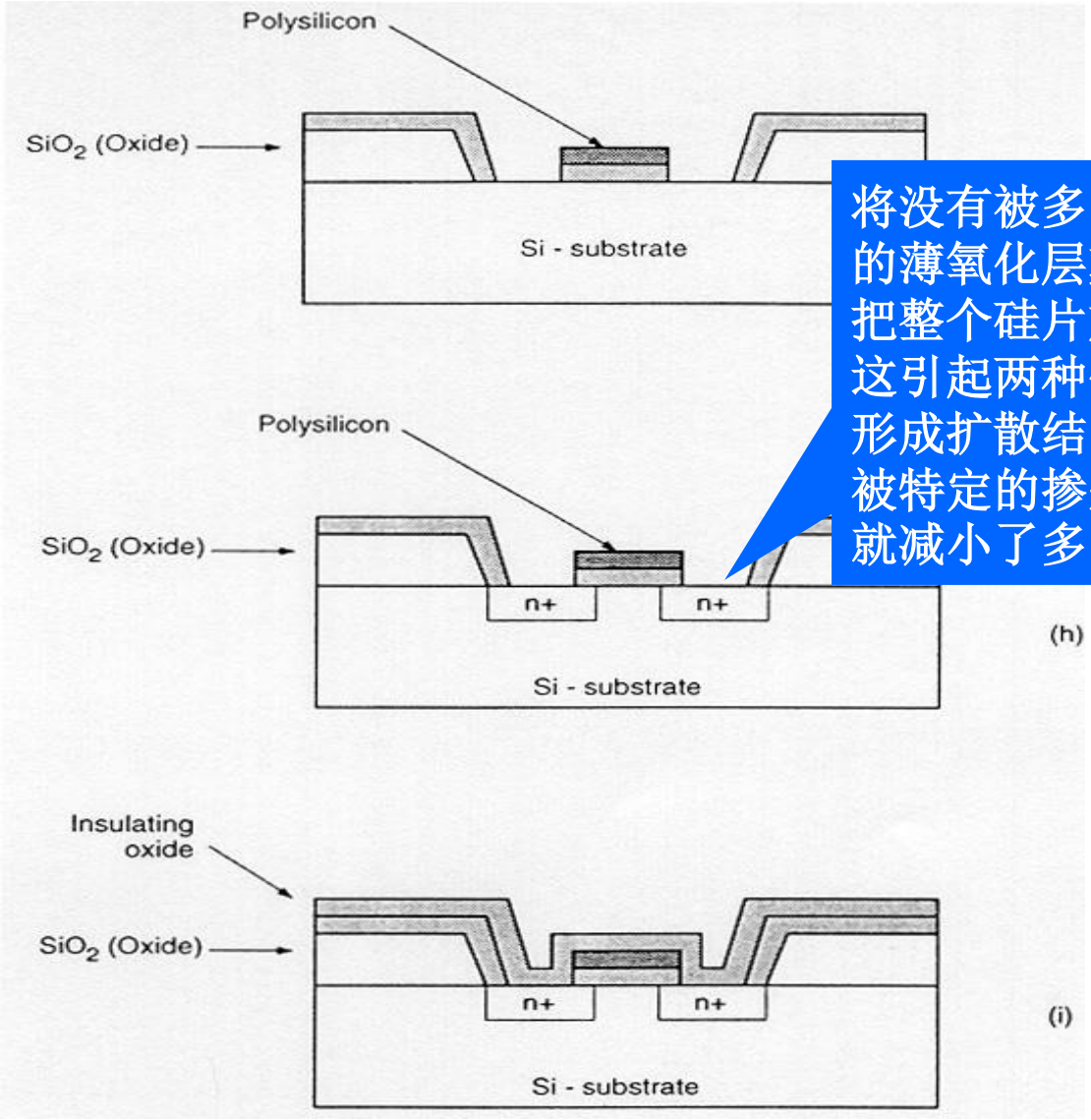
硅栅MOS工艺简介



在暴露的硅表面生长一层精确控制的薄二氧化硅层(150Å～200Å)，这叫栅氧化层或者薄氧化层。

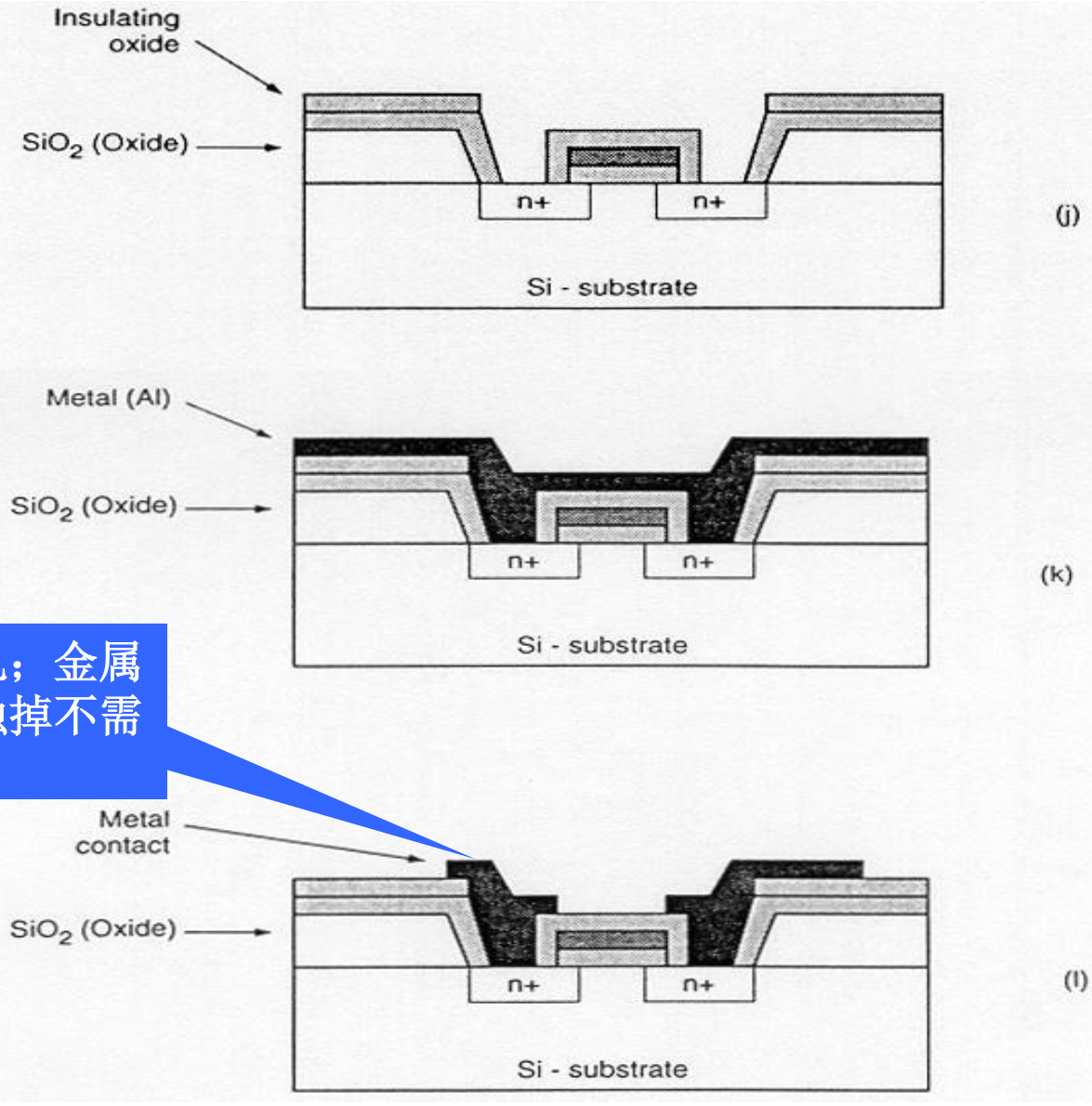
将多晶硅(1um~2um)淀积在硅片表面，再刻蚀内部连线和栅极。刻蚀后的多晶硅栅如图所示。

硅栅MOS工艺简介



将没有被多晶硅覆盖的暴露的薄氧化层刻蚀掉，于是，把整个硅片放在掺杂源下，这引起两种作用：在衬底中形成扩散结，同时多晶硅也被特定的掺杂剂所掺杂，这就减小了多晶硅的电阻率。

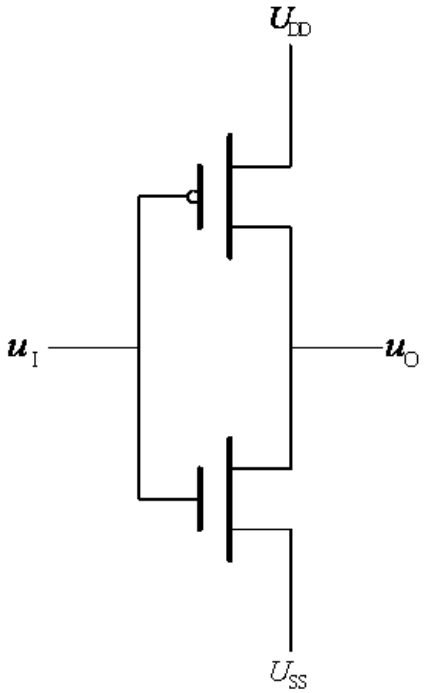
硅栅MOS工艺简介



刻蚀出接触孔；金属化，并且刻蚀掉不需要的金属。

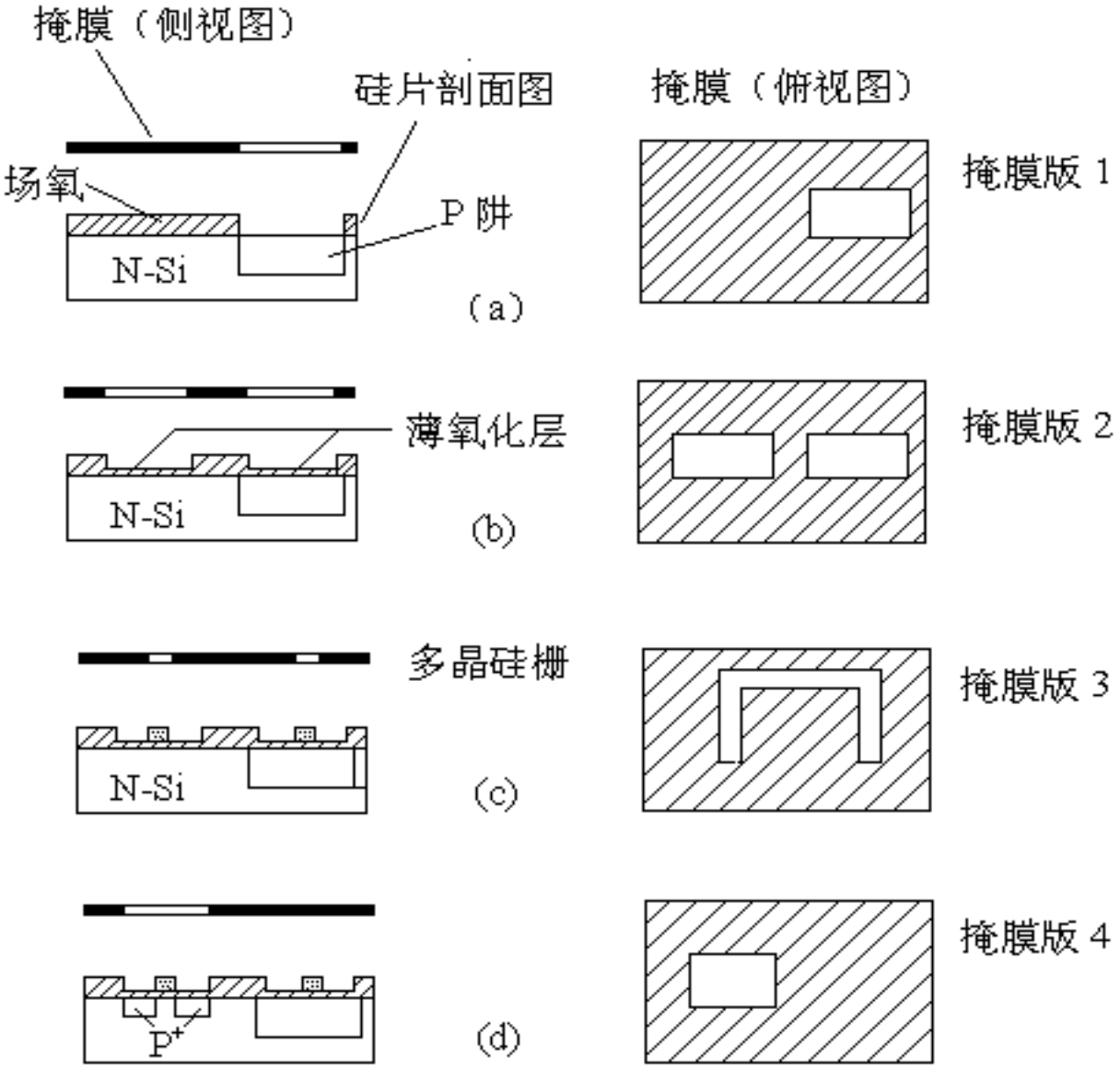
P阱CMOS工艺简介

- P阱CMOS工艺通常是在中度掺杂的N型硅衬底上首先做出P阱，在P阱中做N管，在N型衬底上做P管。
- 工艺过程的主要步骤及所用的掩模版：

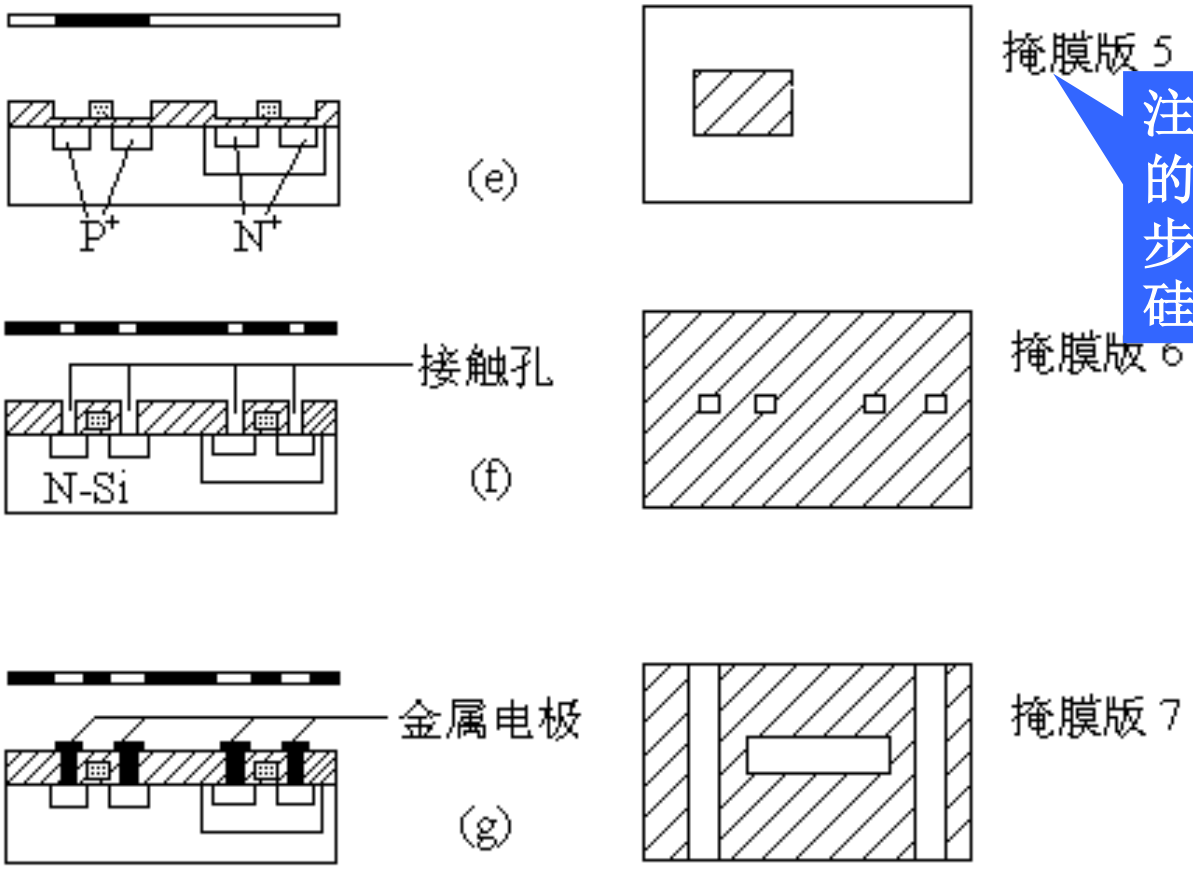


- 第一块掩模版：用来规定P阱的形状、大小及位置；
- 第二块掩模版：用于确定薄氧化层；
- 第三块掩模版：用来刻蚀多晶硅，形成多晶硅栅极及多晶硅互连线；
- 第四块掩模版：确定需要进行离子注入形成P+的区域；
- 第五块掩模版：用来确定需要进行掺杂的N+区域；
- 第六块掩模版：确定接触孔，将这些位置处的SiO_2刻蚀掉；
- 第七块掩模版：用于刻蚀金属电极和金属连线；

P阱CMOS工艺简介


掩膜（侧视图）
硅片剖面图
掩膜（俯视图）
场氧
P 阱
N-Si
(a)
掩膜版 1
薄氧化层
N-Si
(b)
掩膜版 2
多晶硅栅
N-Si
(c)
掩膜版 3
P+
(d)
掩膜版 4

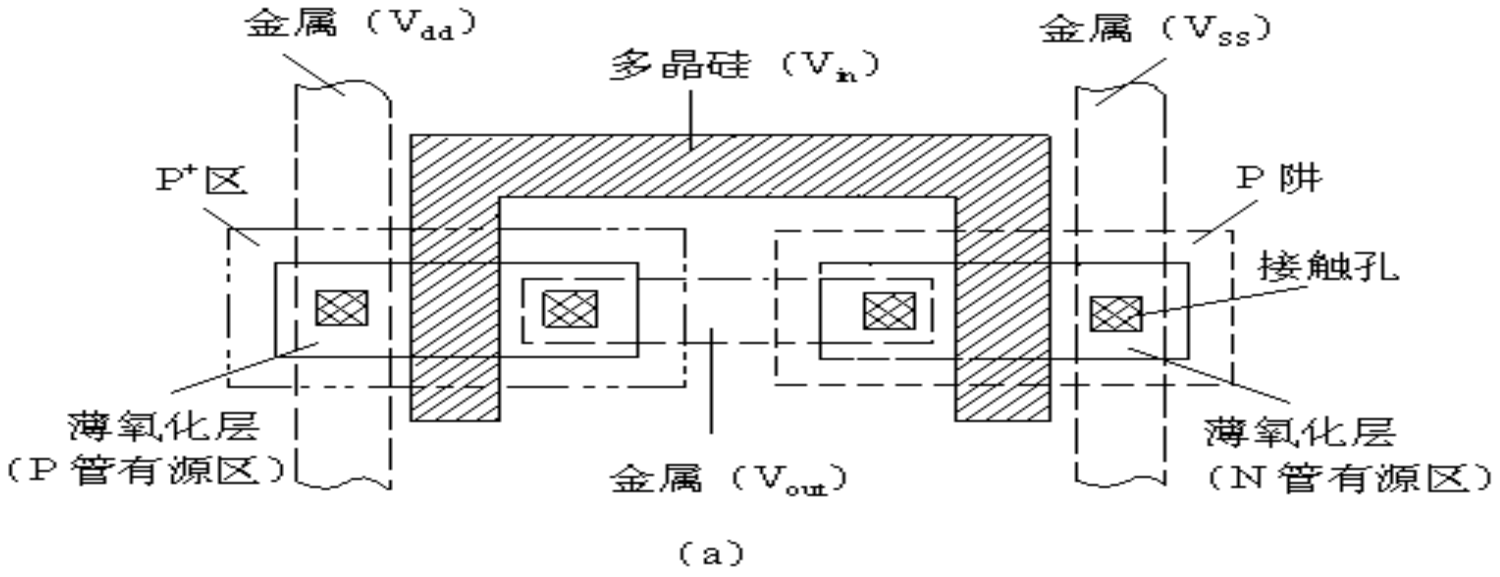
P阱CMOS工艺简介



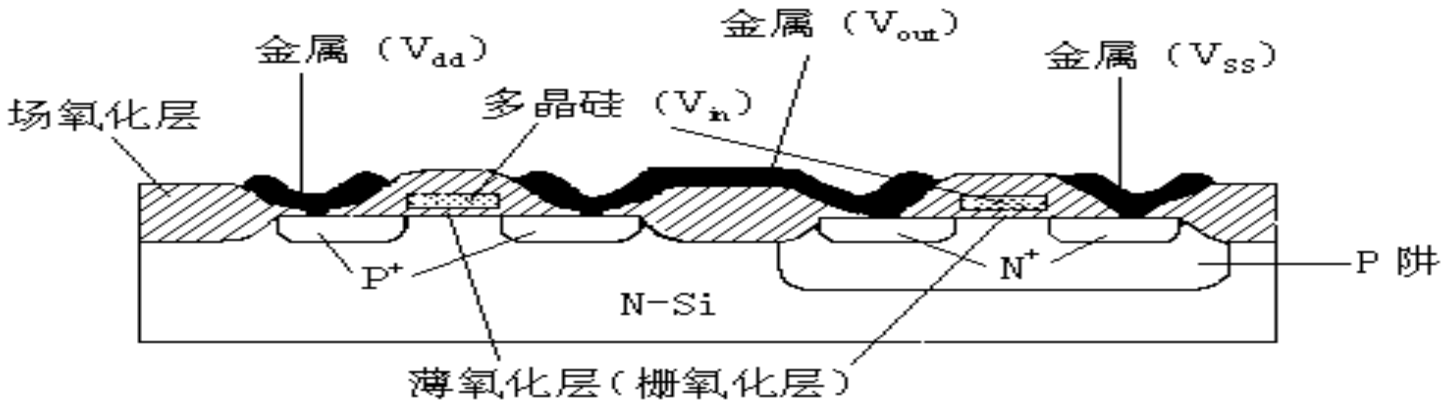
注意：这是版4的负版。从这一步可以看出多晶硅栅的掩膜作用。

P阱CMOS工艺流程

P阱CMOS工艺简介

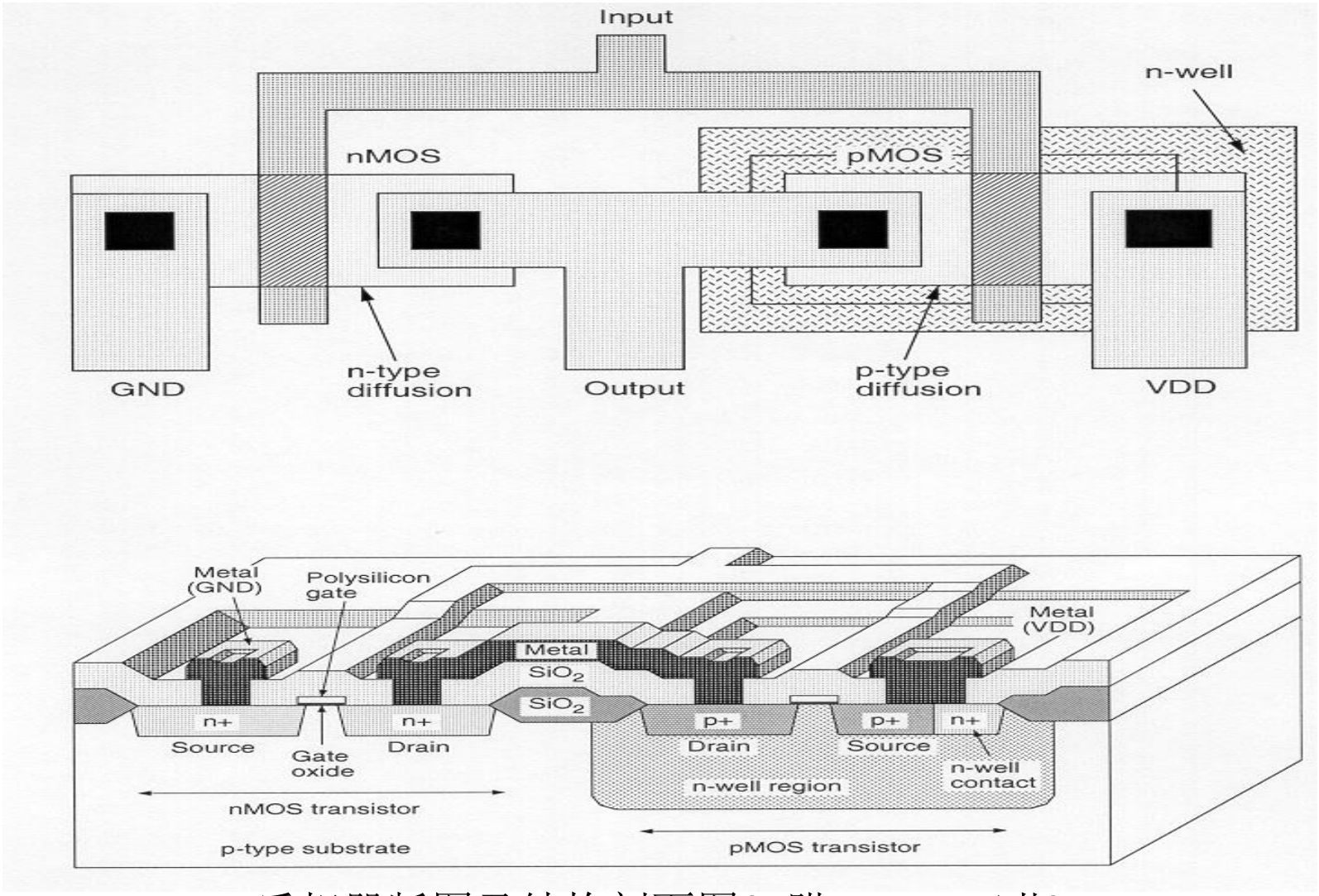


（a）



（b）

N阱CMOS工艺简介



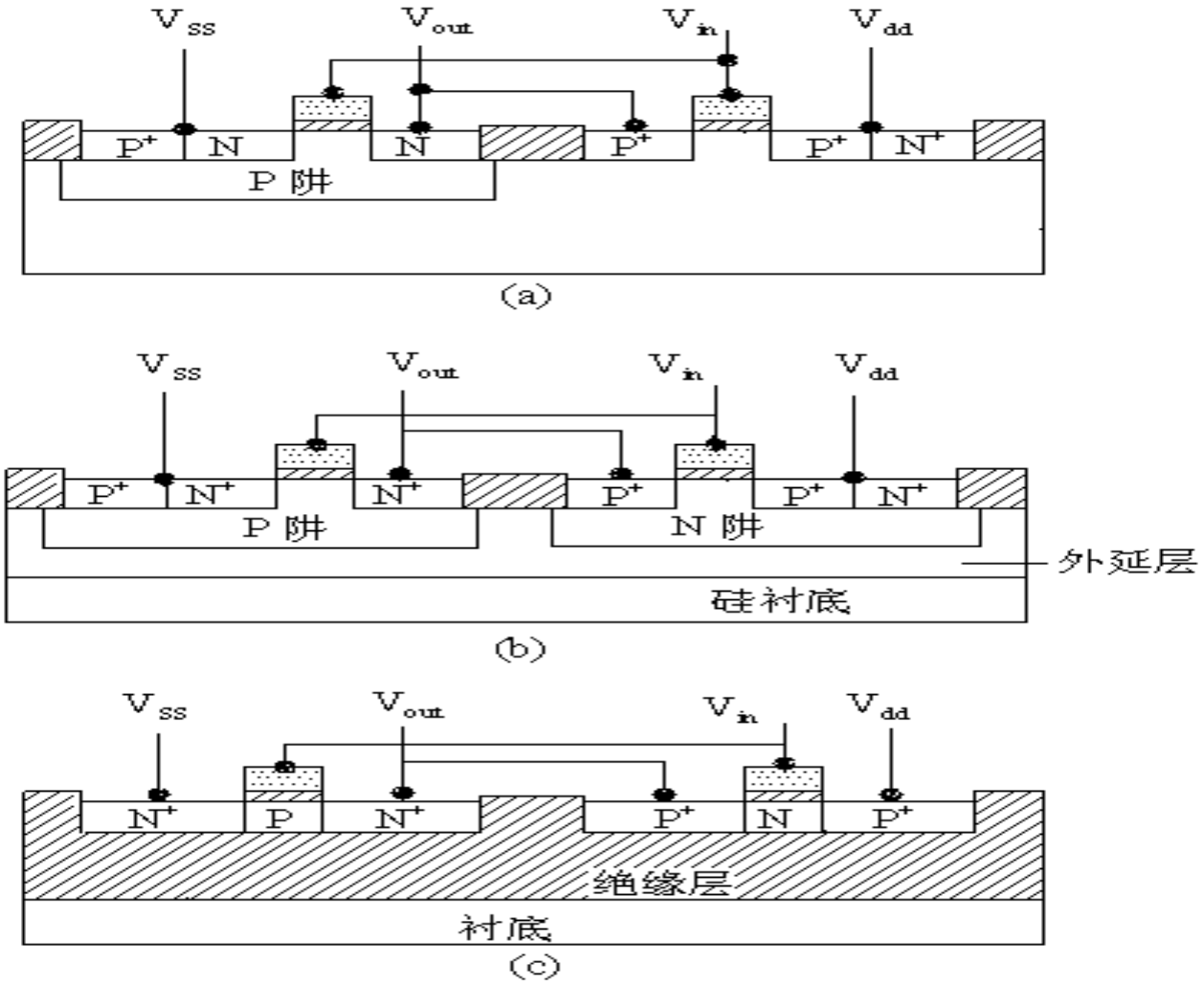
反相器版图及结构剖面图(N阱CMOS工艺)

双阱工艺及SOI CMOS工艺简介

- 双阱工艺：通常是在N+或P+衬底上外延生长一层厚度及掺杂浓度可精确控制的高纯度硅层(外延层)，在外延层中做双阱(N阱和P阱)，N阱中做P管，P阱中做N管。其工艺流程除了阱的形成这一步要做双阱以外，其余步骤与P阱工艺类似 。

- 绝缘体上硅(SOI)的基本思想是在绝缘衬底上的薄硅膜中做半导体器件。例如在蓝宝石上外延硅(SOS)，在薄的硅层上用不同的掺杂方法分别形成N型器件和P型器件。

双阱工艺及SOI CMOS工艺简介



 工艺比较示意图：(a) P阱工艺　(b) 双阱工艺　(c)SOI CMOS工艺

2.4 版图设计规则

版图设计规则

- 内容：设计规则规定了掩膜版各层几何图形宽度、间隔、重叠及层与层之间的距离等的最小容许值。
- 设计规则的作用：是设计和生产之间的一个桥梁;是一定的工艺水平下电路的性能和成品率的最好的折中。
- 设计规则描述：

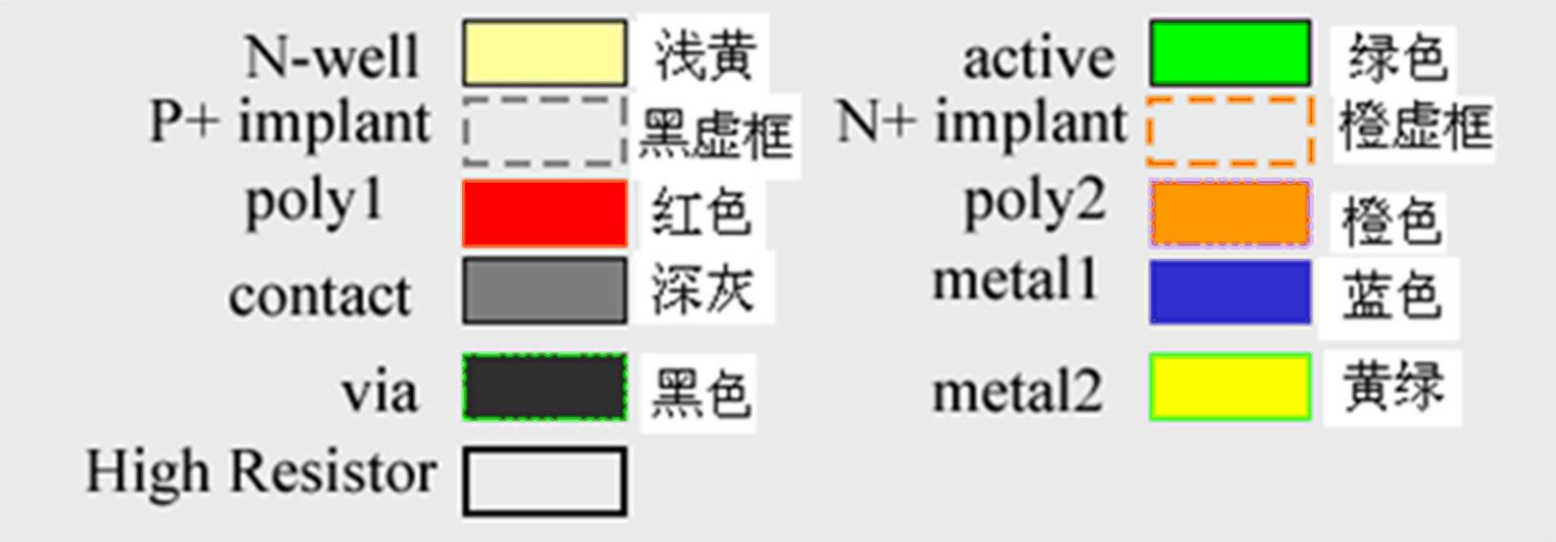
1. 微米设计规则：以微米为单位直接描述版图的最小允许尺寸。
2. λ设计规则：以λ为基准的，最小允许尺寸均表示为λ的整数倍。λ近似等于将图形移到硅表面上可能出现的最大偏差；如限制最小线宽为2λ，窄了线条就可能断开，λ可以随着工艺的改进线性缩小，这就使设计变得更加灵活。

版图设计规则

- 所有的CMOS工艺都可以采用下列特征描述
  - ✓ 两种不同的衬底(P, N)
  - ✓ P型管和N型管掺杂区的形成材料
  - ✓ MOS管的栅极
  - ✓ 内连通路
  - ✓ 层间的接触
- 对于典型的CMOS工艺，可以用不同的形式来表示各层
  - ✓ JPL实验室提出的一组彩色的色别图
  - ✓ 点划线图形
  - ✓ 不同线型图
  - ✓ 上述几种类型的组合

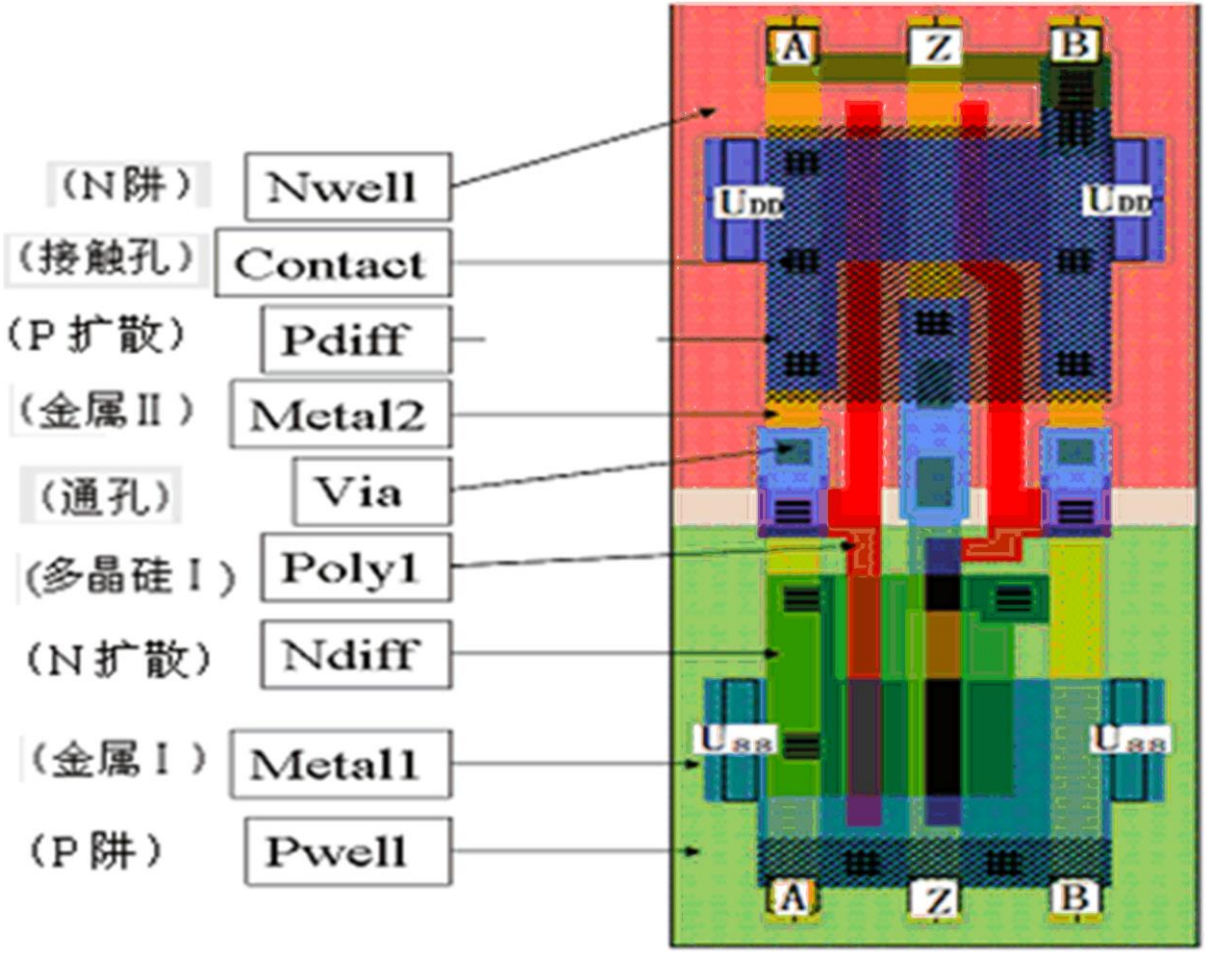
工艺层定义

版图设计是要分层次的，不同公司对工艺层的定义基本相同，但表示的方法和颜色不尽相同。



某公司0.6μm CMOS工艺层定义

工艺层定义



双阱、双层金属布线与非门版图的工艺层

工艺层定义

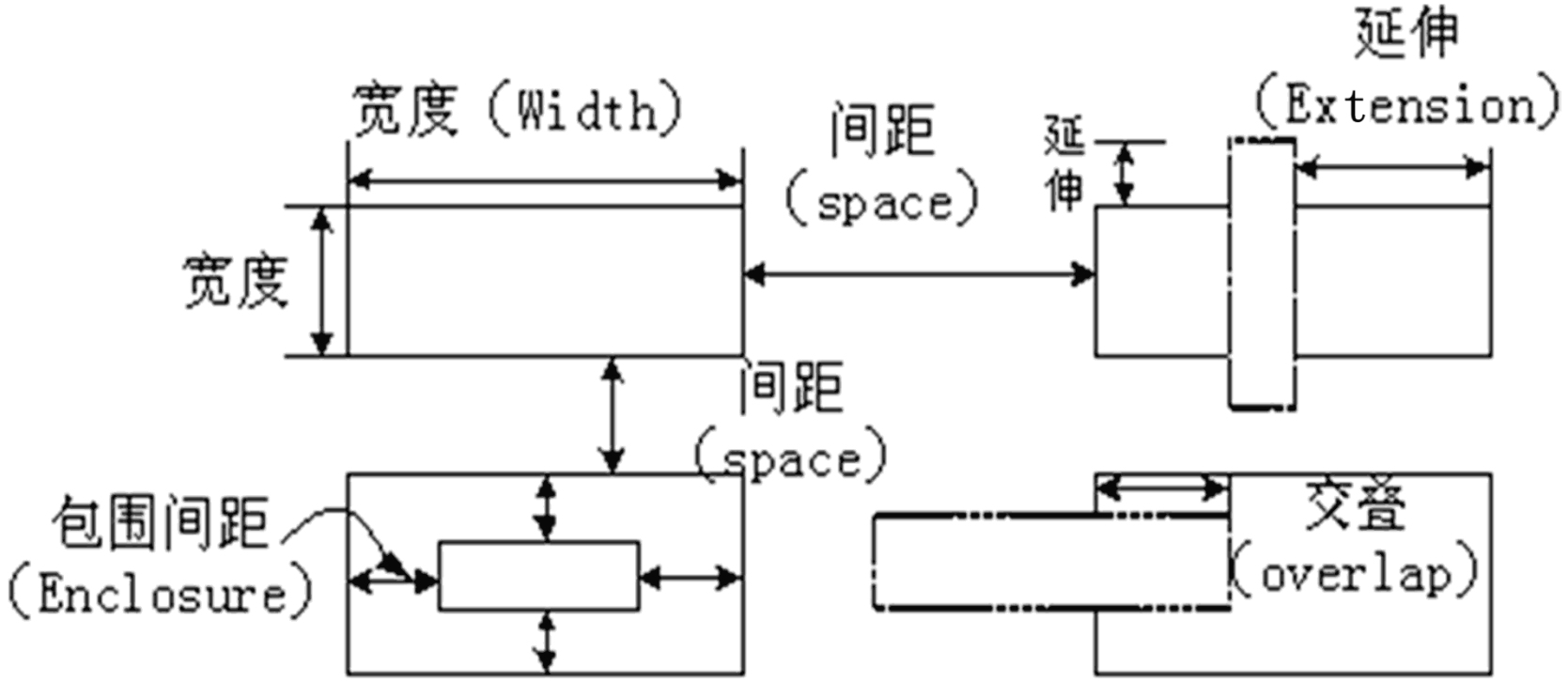
- 典型CMOS工艺层图定义

层次	CIF码	GDS II码	说明
P阱(P-well)	CWP	41	在N衬底上做P阱，在P阱上做N管
N阱(N-well)	CWN	42	在P衬底上做N阱，在N阱上做P管
源区	CAA	43	在源区上做源、漏、栅极
P+注入	CSP	44	离子注入或扩散形成源区，欧姆接触等
N+注入	CSN	45	离子注入或扩散形成源区，欧姆接触等
多晶硅	CPG	46	做栅极或连线，多晶硅电容极板等
接触孔	CCC	25	金属1与有源区，多晶硅的所有接触孔
金属1	CMF	49	第一层金属连线
通孔	CVA	50	连接第一层金属和第二层金属的接触孔
金属2	CMS	51	第二层金属连线

CIF码的第一个字母表示工艺类别，第二个字母代表工艺层

设计规则基本定义

- 版图设计规则规定了几何图形的最小宽度、最小间隔、最小延伸、最小交叠以及层间的最小距离。
- 以λ为基准的版图规则对以下几种掩膜层几何参数进行定义。



西安电子科技大学

2.5 版图设计中的注意事项

匹配设计

- 引起“失配”的原因主要有

1. 随机失配(尺寸，掺杂，氧化层厚度等影响元件值参量的波动等)
2. 系统失配(工艺偏差，接触孔电阻，扩散区互相影响，机械压力，温度梯度等)

- 提高“匹配精度”可采取

1. 差分结构电路布局，布线必须“对称”。
2. 元件单元取整数比，降低工艺偏差和欧姆接触电阻的影响。
3. 采用公用重心设计(common-centroid)以减少梯度效应(热、温度、压力等)
4. 加“虚拟(dummy)”元件，保证周围环境对称度。
5. 匹配元件与其它元件保证一定距离，以减少扩散区的相互影响。

抗干扰设计

- 注意数模混合集成电路的版图布局

① 模拟地与数字地分离

② 模拟电路/数字电路、模拟总线/数字总线尽量分开，而不交叉混合

③ 根据各模拟单元的重要程度，决定其与数字部分间距的大小次序

- 屏蔽和减少寄生干扰，寄生反馈

① 敏感信号线不要平行走线，敏感信号线的线距要加大

② 敏感信号线周围可加地线屏蔽

③ 元件设计和布局、布线要尽量减小寄生电阻和寄生电容

④ 输入和输出最好分布在芯片的两端，以减小输入输出之间的电磁干扰

⑤ 适当增加接触孔数和通孔数，以增加可靠性和减小接触孔寄生电阻

⑥ 宽大尺寸管子可采用若干个小尺寸管子并联或接成多管叉指结构

⑦ 注意电源线允许的电流密度

- 加滤波电容

① 电源线上和芯片版图的空余面积可添加MOS电容进行电源滤波

② 对模拟电路中的偏置电路和参考电压可加多金硅电容滤波

2.6 版图检查

版图检查

版图检查包含设计规则检查(DRC)、电学规则检查(ERC)、版图参数提取(LPE)，以及版图与电路图对照(LVS)。

- 设计规则检查(DRC, Design Rule Check)

设计规则检查的任务是检查和发现版图设计中不符合设计规则的错误，运行DRC，程序可按照相应规则检查文件运行，一旦发现错误，就会在错误处做出标记(mark)，并且做出解释(explain)，设计者都可根据提示进行修改。

版图检查

- 电学规则检查(ERC, Electrical Rule Check)

由于发现电学错误，例如，电源、地、输入输出线的连接错误等需要进行电学规则检查。为了进行ERC检查，首先应在版图中将各有关电学节点做出定义，给出“节点名”。ERC可以发现

节点开路(出现多个相关节点名)

节点短路(一个节点出现多个节点名)

接触孔浮孔(出现接触孔与金属层没有覆盖)

特定区域未接触(出现N阱未接电源，P阱未接地等)

不合理的元器件节点数(扇出系数太高)

版图检查

- 版图参数提取(LPE, Layout Parameter Extraction)

版图参数提取就是从已设计的版图中提取各元器件的参数及其它们的连接关系，以及各种寄生电容和寄生电阻参数，自动建立一种模型。参数提取的用途有：

1. 作为电特性检查基础，利用这些参数将版图还原成电路图，然后与原始电路图对照比较，以便更严格地发现错误。
2. 将提取的器件连接关系和寄生参数作为“后仿真”的输入数据，估计版图寄生参数对电路性能的影响。
3. 如果采用标准单元库自动生成版图，由于标准单元库中的单元都已经过检验，所以只需要提取连线和连线的分布电容和电阻，再进行电路的“后仿真”即可。

版图检查

- 电路图与版图一致性对照检查(LVS, Layout Versus Schematic)

电路图与版图一致性对照检查程序是将有原电路图产生的元件网表和端点列表文件，与由版图提取产生的元件表、网表和端点列表加以上对照比较，发现不一致的元件节点，端点连线都在一个LVS检查文件中，并在电路图和版图中显示出来。

- 经过以上四项检查和修正，保证版图设计正确无误，才算完成了芯片的设计。此时才可以向制造方提供芯片的所有数据。

西安電子科技大學

第三章

CMOS集成电路工艺中的元器件

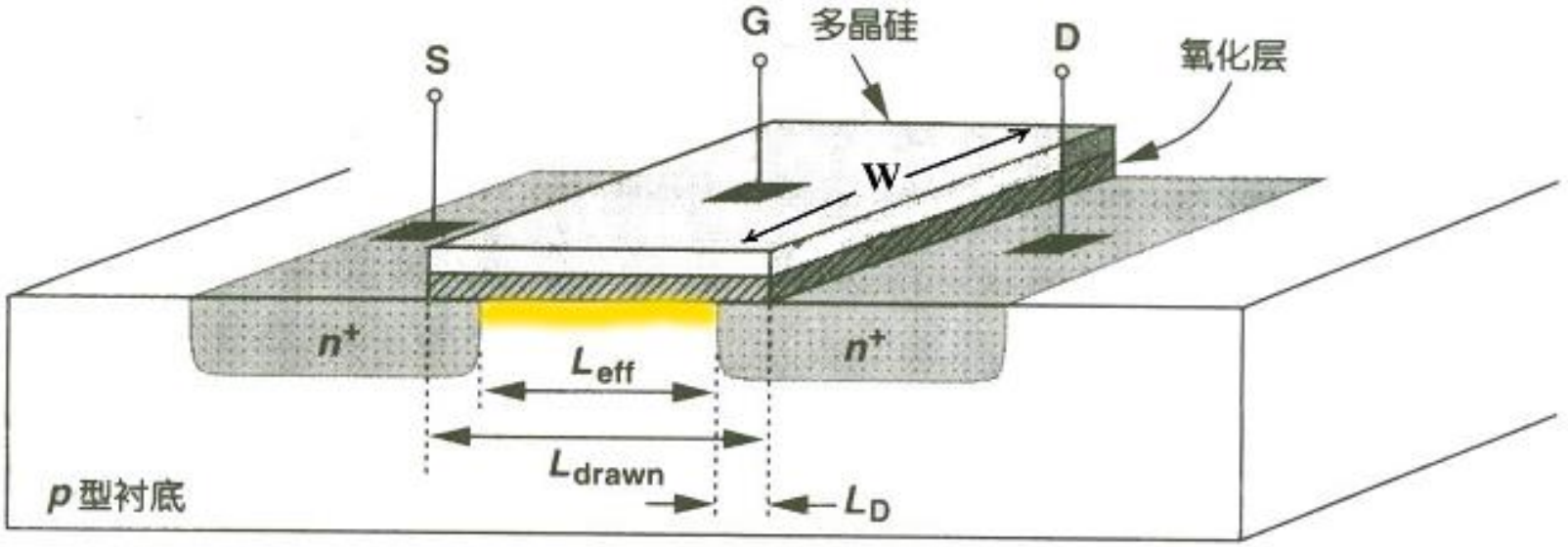
集成电路的元器件种类

集成电路元器件
- MOS 管
- 电容
- 电阻
- 电感
- 连线

西安电子科技大学

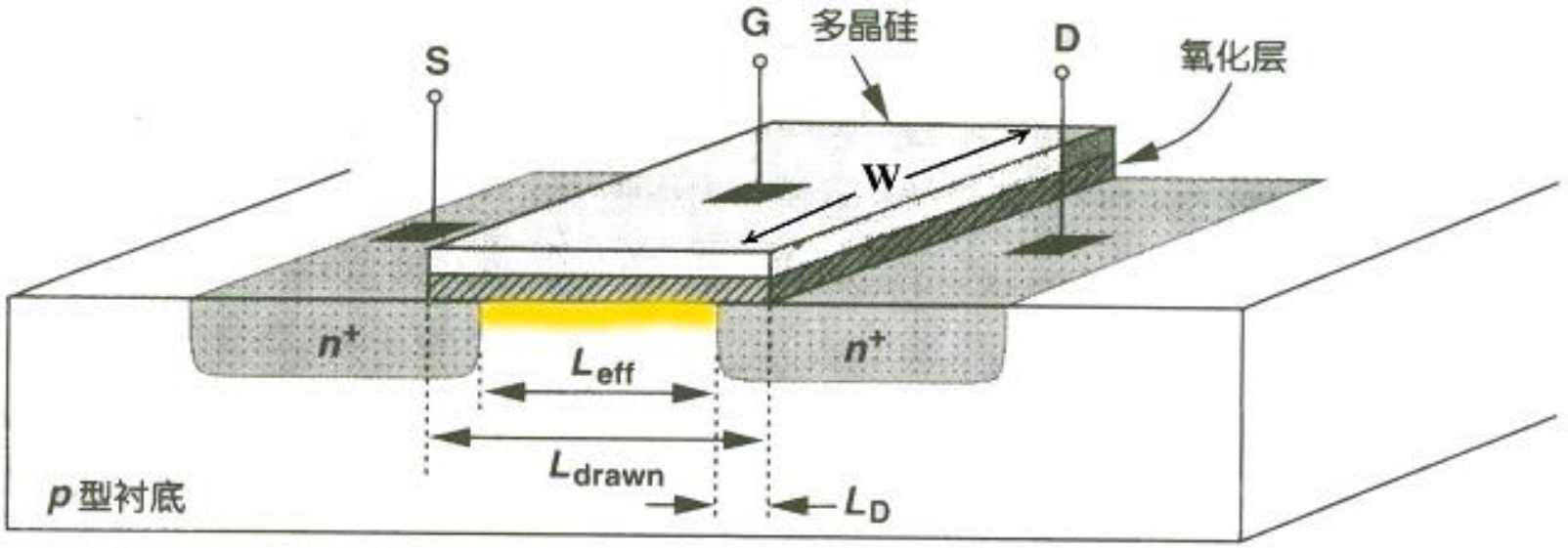
3.1 MOS管的结构及符号

NMOS管的简化结构



- **B：** P型硅衬底 (P-Substrate, Bulk or Body)
- **S和D：** 源区和漏区(重掺杂N+区)
- **G：** 栅极(重掺杂多晶硅区)/栅极薄氧化层
- **Channel ：** 导电沟道(栅极薄氧化层下的衬底表面)

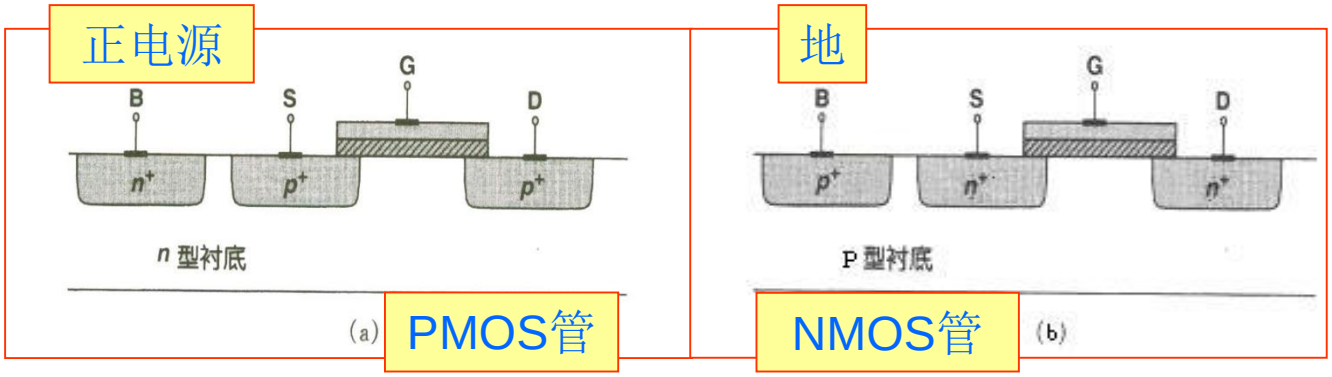
NMOS管的简化结构



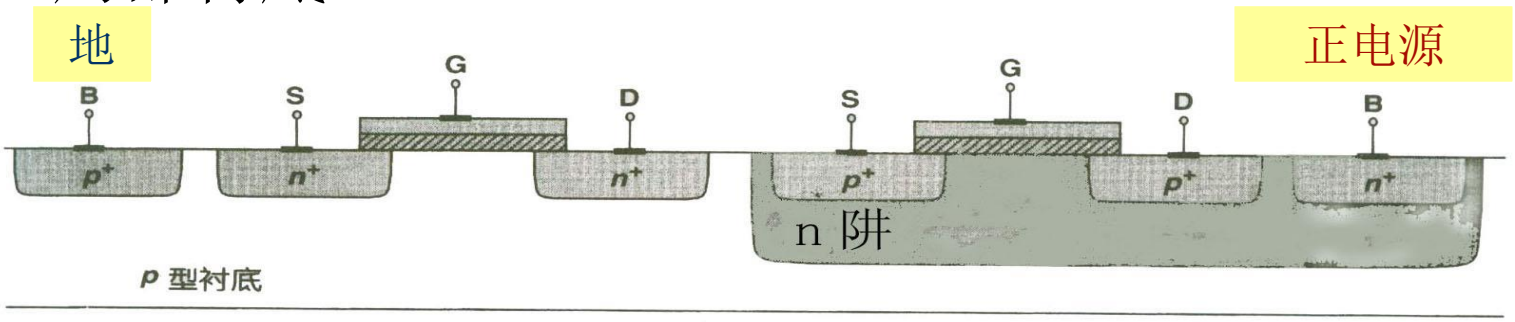
- $L_{eff} = L_{drawn} - 2L_D$
- L_{eff} —有效沟道长度，L_{drawn}—沟道总长度
- 宽长比(***W/L***)和氧化层厚度t_{ox}对MOS电路的性能起着非常重要的作用。
- MOS技术发展中的主要推动力就是缩小沟道长度***L***和氧化层厚度t_{ox}。

N阱及PMOS

为使MOS管的电流只在导电沟道中沿表面流动，源区/漏区以及沟道和衬底之间必须形成反向偏置的PN结隔离。

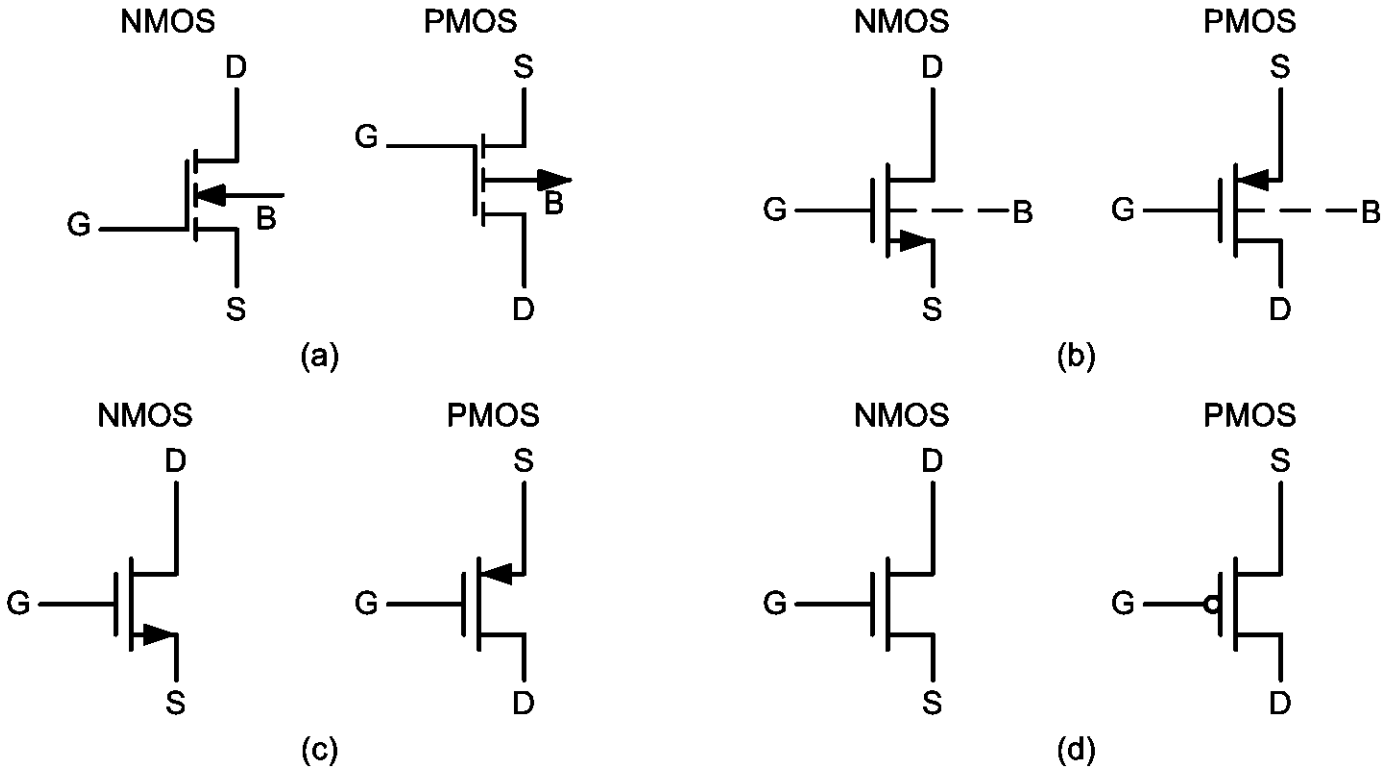


在互补型CMOS中，在同一衬底上制作NMOS和PMOS，因此必须为PMOS做一个称之为“阱(Well)”的“局部衬底”。



MOS管符号

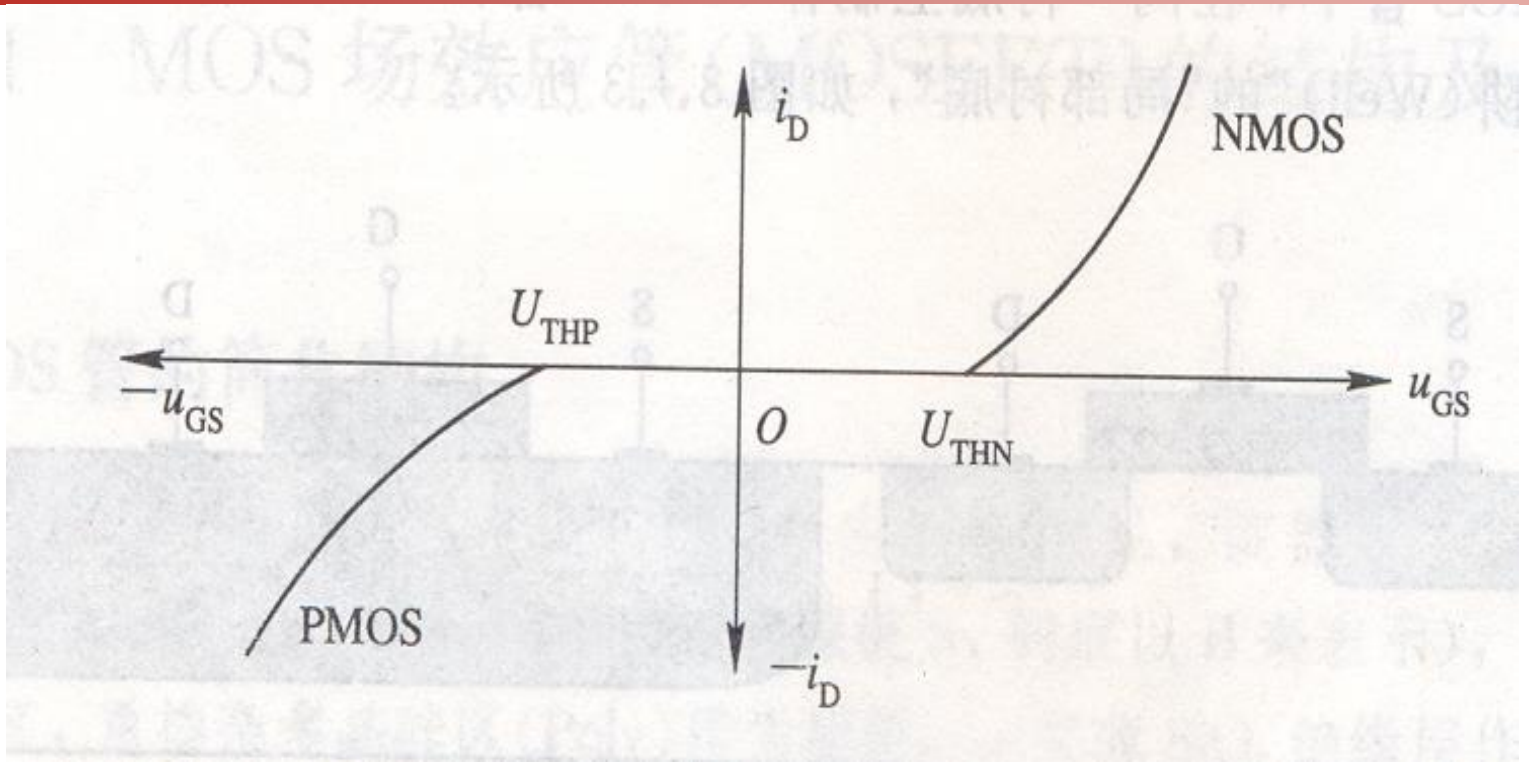
在栅-源极电压(栅偏置)为零时截止(即不导电)的器件称为**增强型器件**；而在栅偏置为零时就导通的器件称为**耗尽型器件**。



西安電子科技大學

3.2 MOS管的电流电压特性

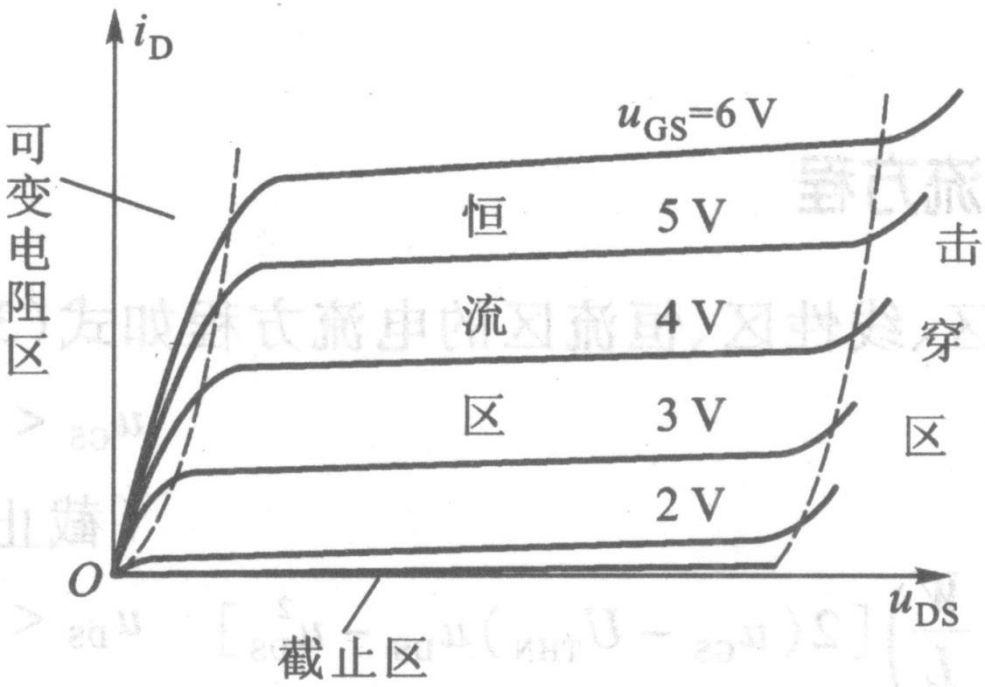
MOS管的转移特性



- 开启电压(阈值电压)U_{THN}和U_{THP}
- U_{THN}的物理意义

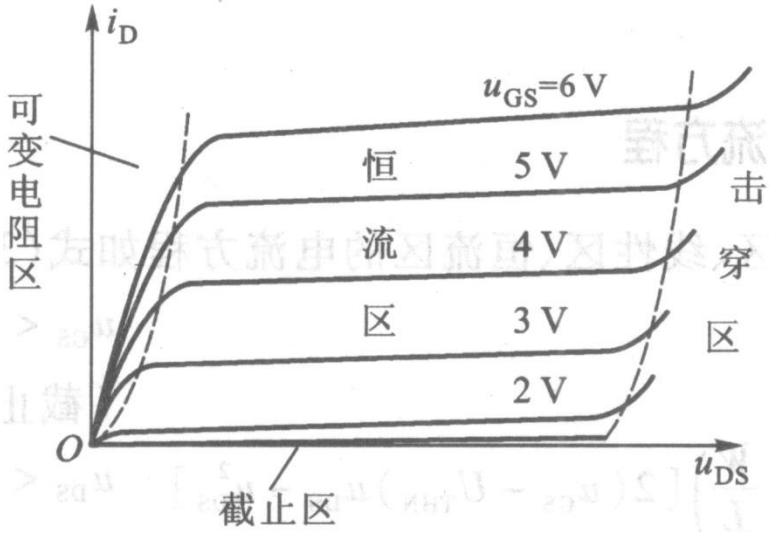
MOS管的输出特性

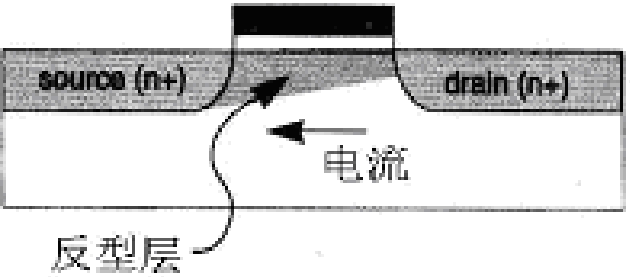
- 栅极电压u_{GS}对漏极电流i_D有明显的控制作用(u_{GS}大于U_{TH}时)。
- 漏极电压u_{DS}对漏极电流i_D的控制作用分成可变电阻区(线性区)和恒流区(饱和区) 两段。

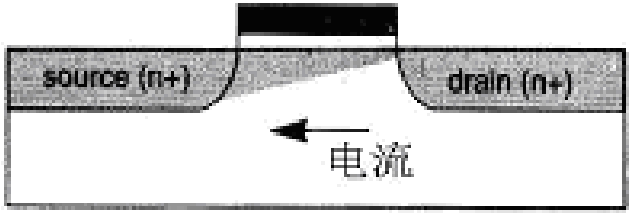


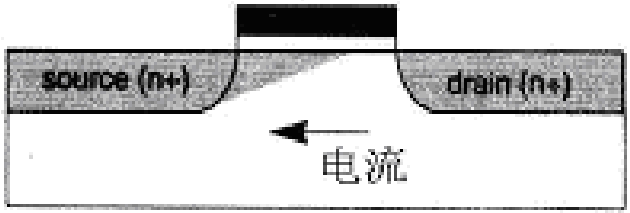
MOS管的输出特性

- 线性区与恒流区是以预夹断点的连线为分界线的，即$u_{DS}=u_{GS}-U_{TH}$。









线性区　　预夹断　　恒流区

$u_{DS}<u_{GS}-U_{TH}$　　$u_{DS}=u_{GS}-U_{TH}$　　$u_{DS}>u_{GS}-U_{TH}$

NMOS管的电流方程

- NMOS管在截止区、线性区和恒流区的电流方程如下

$$i_{\mathrm{DN}}=\begin{cases}0 & u_{\mathrm{GS}}<U_{\mathrm{THN}}\ (\text{截止区})\\ \dfrac{\mu_{\mathrm{n}}C_{\mathrm{ox}}}{2}\left(\dfrac{W}{L}\right)\left[2\left(u_{\mathrm{GS}}-U_{\mathrm{THN}}\right)u_{\mathrm{DS}}-u_{\mathrm{DS}}^{2}\right] & u_{\mathrm{DS}}<u_{\mathrm{GS}}-U_{\mathrm{THN}}\ (\text{线性区})\\ \dfrac{\mu_{\mathrm{n}}C_{\mathrm{ox}}}{2}\left(\dfrac{W}{L}\right)\left(u_{\mathrm{GS}}-U_{\mathrm{THN}}\right)^{2}\left(1+\lambda_{\mathrm{n}}u_{\mathrm{DS}}\right) & u_{\mathrm{DS}}>u_{\mathrm{GS}}-U_{\mathrm{THN}}\ (\text{恒流区})\end{cases}$$

PMOS管的电流方程

- PMOS管在截止区、线性区和恒流区的电流方程如下

$$i_{\mathrm{DP}}=\begin{cases}0 & |u_{\mathrm{GS}}|<|U_{\mathrm{THP}}|\text{(截止区)}\\ -\dfrac{\mu_{\mathrm{p}}C_{\mathrm{ox}}}{2}\left(\dfrac{W}{L}\right)\left[2\left(u_{\mathrm{GS}}-U_{\mathrm{THP}}\right)u_{\mathrm{DS}}-u_{\mathrm{DS}}^{2}\right] & |u_{\mathrm{DS}}|<|u_{\mathrm{GS}}|-|U_{\mathrm{THP}}|\text{(线性区)}\\ -\dfrac{\mu_{\mathrm{p}}C_{\mathrm{ox}}}{2}\left(\dfrac{W}{L}\right)\left(u_{\mathrm{GS}}-U_{\mathrm{THP}}\right)^{2}\left(1+\lambda_{\mathrm{p}}u_{\mathrm{DS}}\right) & |u_{\mathrm{DS}}|>|u_{\mathrm{GS}}|-|U_{\mathrm{THP}}|\text{(恒流区)}\end{cases}$$

MOS管电流方程中的参数

N型衬底电阻率3~5Ω·cm
P型衬底电阻率12~14Ω·cm

- μ_n 电子迁移率(单位电场作用下电子的迁移速度)
- μ_p 空穴迁移率(单位电场作用下空穴的迁移速度)
- μ_n / μ_p 电子/空穴的迁移率
- C_{ox} 单位面积栅电容，且$C_{ox} = \dfrac{\varepsilon_0 \varepsilon_{SiO_2}}{t_{ox}}$

- W/L 导电沟道的宽度和长度之比
- U_{THN}、U_{THP} NMOS、PMOS管的开启电压
- λ_n、λ_p 沟道调制系数，是一个经验的沟道长度修正系数，反应u_{GS}对沟道长度的影响。

MOS管的输出电阻

- 线性区的输出电阻— 压控电阻

$$i_{\mathrm{D}}=\frac{\mu_{\mathrm{n}}C_{\mathrm{ox}}}{2}\left(\frac{W}{L}\right)\left[2\left(u_{\mathrm{GS}}-U_{\mathrm{TH}}\right)u_{\mathrm{DS}}-u_{\mathrm{DS}}^{2}\right]$$

$$\approx\mu_{\mathrm{n}}C_{\mathrm{ox}}\left(\frac{W}{L}\right)\left(u_{\mathrm{GS}}-U_{\mathrm{TH}}\right)u_{\mathrm{DS}}$$

$$R_{\mathrm{ON}}=\frac{\partial u_{\mathrm{DS}}}{\partial i_{\mathrm{D}}}=\frac{1}{\mu_{\mathrm{n}}C_{\mathrm{ox}}\left(\frac{W}{L}\right)\left(u_{\mathrm{GS}}-U_{\mathrm{TH}}\right)}$$

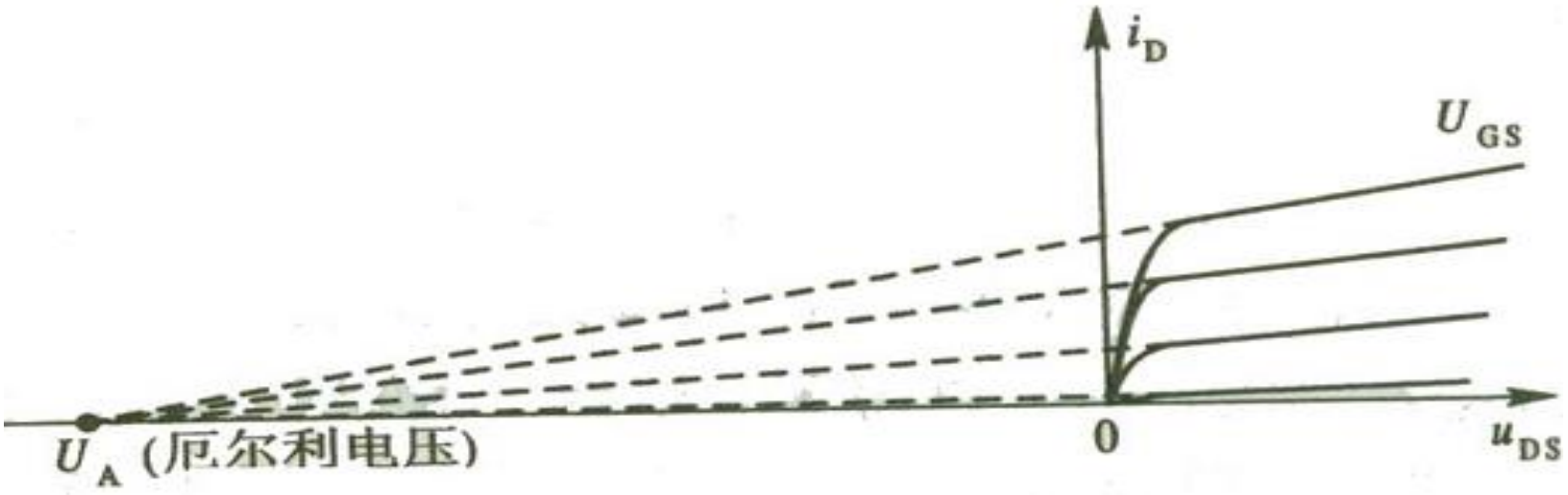
- 由该式可见，处于线性区的MOS管输出电阻R_{ON}是u_{GS}的函数，且与之成反比。该电阻受u_{GS}控制，故称线性区的输出电阻为“压控电阻”，线性区又称为“可变电阻区”。

MOS管的输出电阻

- 恒流区的输出电阻

$$R_{ON}=\frac{\partial u_{DS}}{\partial i_D}=\frac{1}{\lambda_n\frac{\mu_n C_{ox}}{2}\left(\frac{W}{L}\right)\left(u_{GS}-U_{TH}\right)^2}=\frac{1}{\lambda_n I_{DQ}}=\frac{U_A}{I_{DQ}}$$

- U_A是与沟道调制系数有关的厄尔利电压，其定义见下图所示



MOS管的跨导

电流受到栅源过驱动电压控制，可以定义一个性能系数来表示电压转换电流的能力。因此我们把这个性能系数定义为漏电流的变化量除以栅源电压的变化量，称之为“跨导”，用g_m表示。MOS管工作在恒流区时

$$i_D = \frac{1}{2}\mu_n C_{ox}\frac{W}{L}(u_{GS}-U_{TH})^2(1+\lambda_n u_{DS})$$

$$\approx \frac{1}{2}\mu_n C_{ox}\frac{W}{L}(u_{GS}-U_{TH})^2 \qquad (\text{忽略沟道调制影响})$$

$$g_m = \frac{\partial i_D}{\partial u_{GS}} = \mu_n C_{ox}\left(\frac{W}{L}\right)(u_{GS}-U_{TH})$$

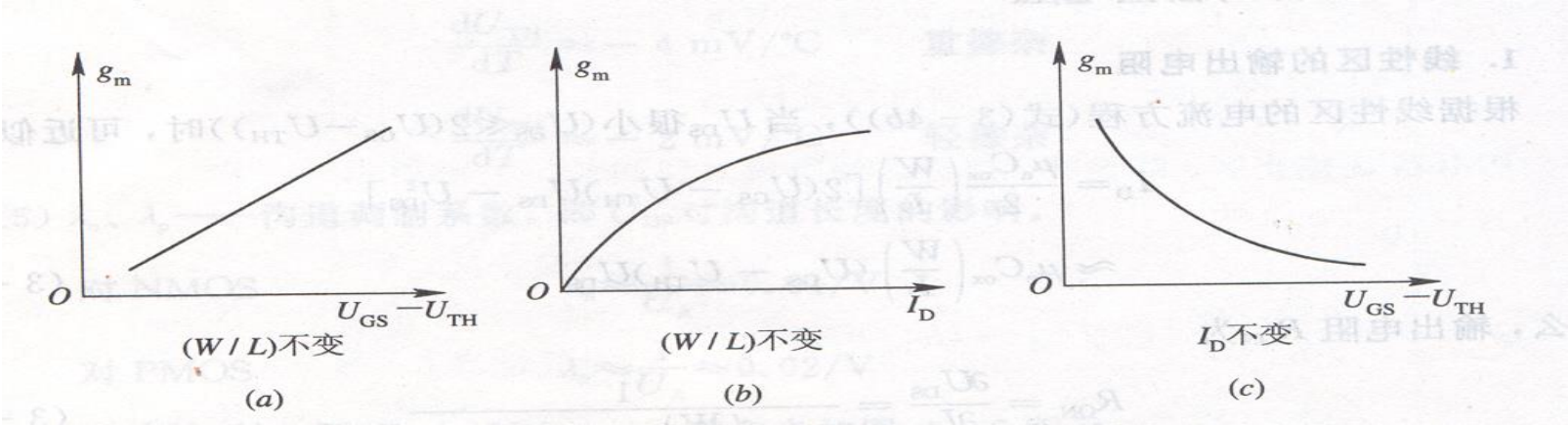
$$= \sqrt{2\mu_n C_{ox}\left(\frac{W}{L}\right)i_D}$$

$$= \frac{2i_D}{u_{GS}-U_{TH}}$$

MOS管的跨导

- 从某种意义上讲，g_m 代表了器件的灵敏度：

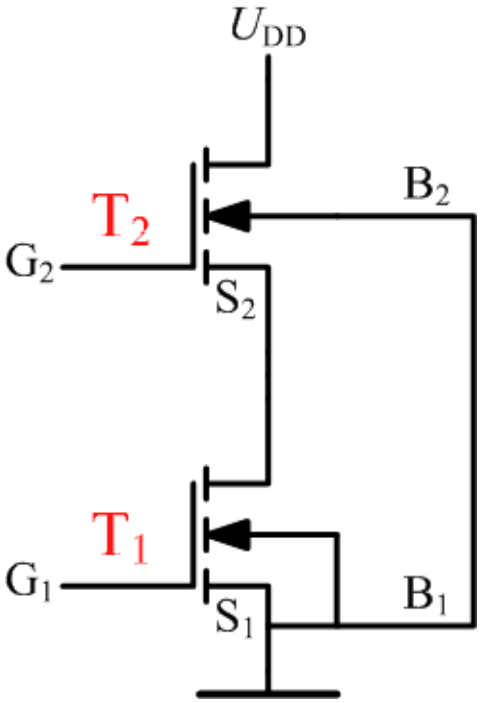
  对于一个大的g_m 来讲，u_{GS}的一个微小的改变将会引起i_D产生很大的变化。



- 在W/L不变的情况下，g_m 与$(u_{GS}-U_{TH})$成线性关系，与i_D的平方根成正比。
- 在i_D不变的情况下，g_m与$(u_{GS}-U_{TH})$成反比。

体效应与背栅跨导

- 在集成电路中，同一片衬底上要集成许多MOS管，为了保证其正常工作，N型管的衬底要接到整个电路的最低电位点，P型管的衬底要接到整个电路的最高电位点。于是MOS管的源极与衬底之间存在电位差。而且为了保证沟道与衬底之间的隔离，其PN结必须反偏，即T_2管的$u_{BS}<0$。



体效应与背栅跨导

- 在u_{BS}<0的情况下，沟道与衬底之间的耗尽层会加厚，导致阈值电压U_{TH}增大、沟道变窄及沟道电阻增大，导致i_D的减小。这种效应称为“体效应”或“衬底调制效应”。
- 考虑到体效应的阈值电压U_{TH}为

$$U_{TH}=U_{THO}\pm\gamma\sqrt{2u_{BS}} \quad (\text{负号适用于PMOS})$$

- 引入背栅跨导g_{mb}来反映u_{BS}对漏极电流的影响

$$g_{mb}=\frac{\partial i_D}{\partial u_{BS}}$$

- 用跨导比η来表示背栅跨导与跨导的关系

$$\eta=\frac{g_{mb}}{g_m}\approx 0.1\sim 0.2$$

场效应管亚阈区特性

- MOS管在从弱反型层向强反型层过渡时沟道中已有电流存在，不过该电流很小。将弱反型层向强反型层过渡的区域定义为“亚阈区”。在该区域，MOS管的电流电压关系符合下式描述的指数关系。

$$I_D = I_{D0}\left(\frac{W}{L}\right)e^{\left(\frac{u_{GS}-U_{TH}}{nU_T}\right)}$$

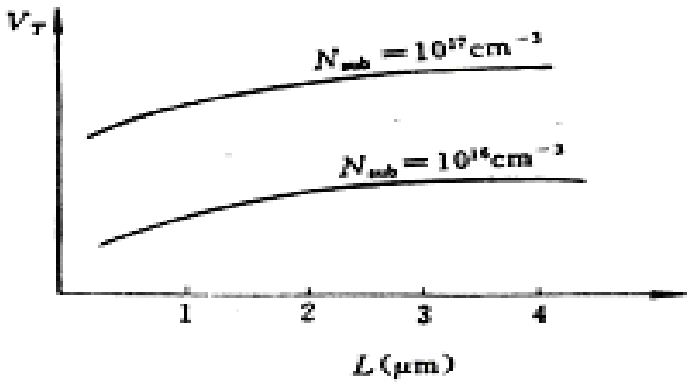
式中：n ——1~2的常数；
U_T ——热电压；
I_{D0} ——工艺系数；

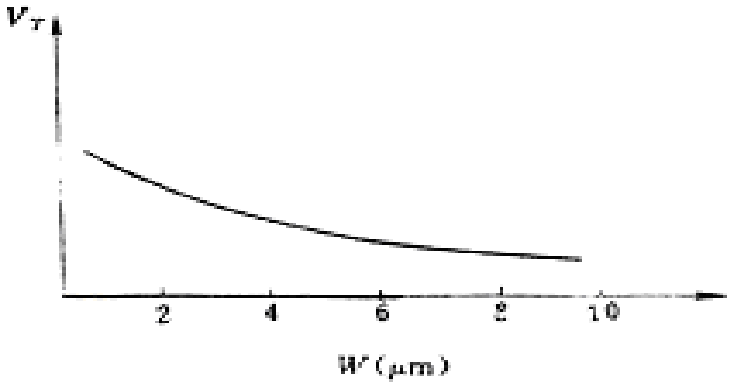
- 在亚阈区的电流电压呈现指数关系，因而栅跨导 g_m 比较大，若放大器工作在此区域将会有较大的增益。
- 影响：亚阈值效应会导致较大的功率损耗(或是模拟信息的丢失)，这在大型电路中(例如内存)是一个特别使人困惑的难题。

沟道尺寸对阈值电压的影响

- 长沟道：L>3~4μm
- 短沟道：L<3μm
- 亚微米工艺：L<1μm的MOS管生产工艺

在短沟道MOS器件中，u_{GS}与沟道尺寸W、L有较大的关系。其中U_{TH}随着L的增大而增大，随W的增大而减小。





温度对开启对电压U_{TH}也有较大的影响，在重掺杂的情况下，U_{TH}的温度系数约为-4mv/ºC，轻掺杂的温度系数约为-2mv/ºC。

MOS管的特征频率

$$f_{\mathrm{T}}=\frac{\mu_{\mathrm{n}}u_{\mathrm{DS}}}{2\pi L^{2}}$$

减小沟道长度L可以大幅度的提高MOS管的特征频率，从而提高器件的工作速度。

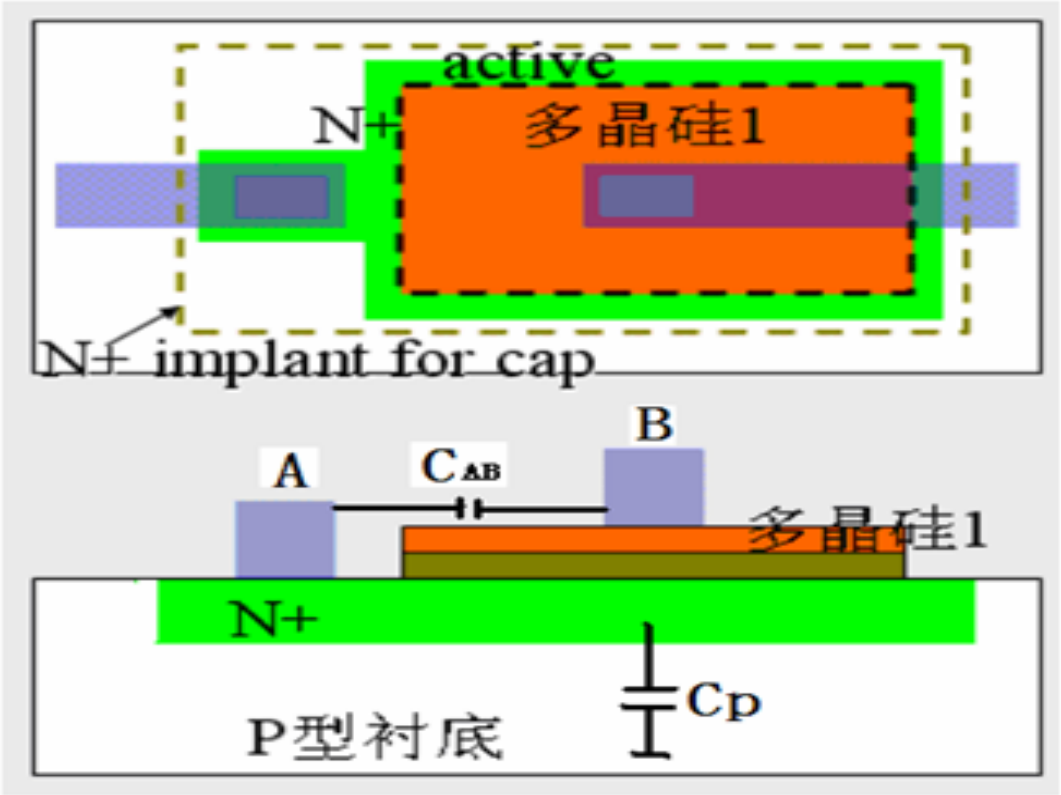
总结：

1. MOS场效应管的性能与沟道宽长比(W/L)有很强的依赖关系。
2. 沟道长度L越小，f_{T}及g_{m}越大，则集成度越高。 因此减小器件尺寸有利于提高器件性能。
3. 提高载流子迁移率μ有利于增大f_{T}及g_{m}，NMOS管的μ_{n}比PMOS管的μ_{p}大2～4倍，所以NMOS管的性能优于PMOS管。
4. 体效应(衬底调制效应) 、沟道调制效应(λ与U_{A})和亚阈区均属二阶效应，在MOS管参数中有所反映。

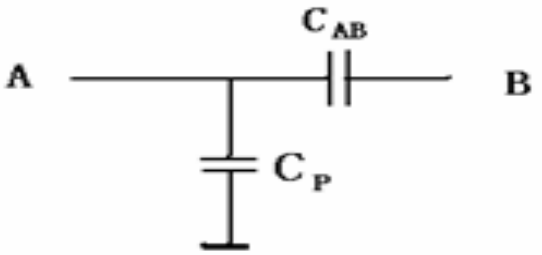
3.3 集成电容

多晶硅—扩散区电容

该电容制作在扩散区上，其上极板是第一层多晶硅，下极板是扩散区，中间的介质是氧化层。

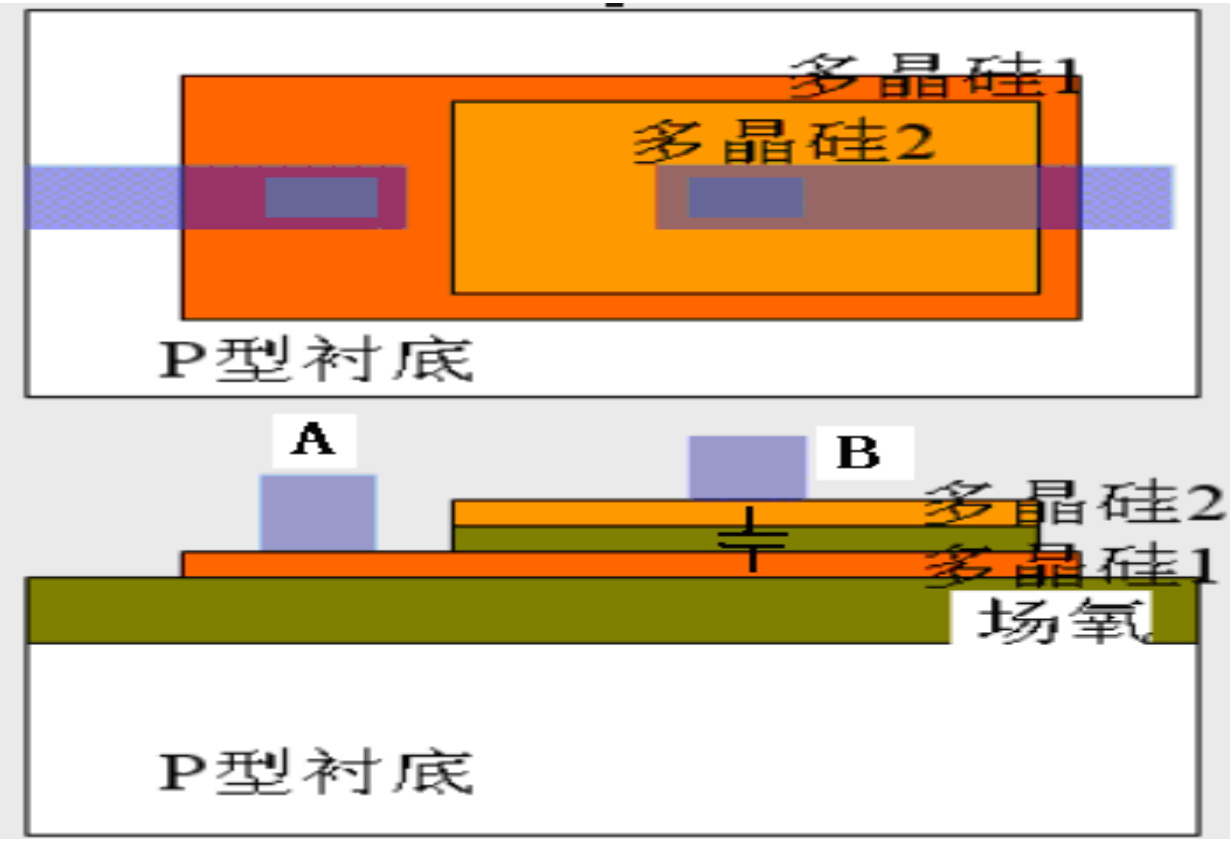


$$C_{AB} = C_{OX} \cdot W \cdot L = C_{OX} A_G$$



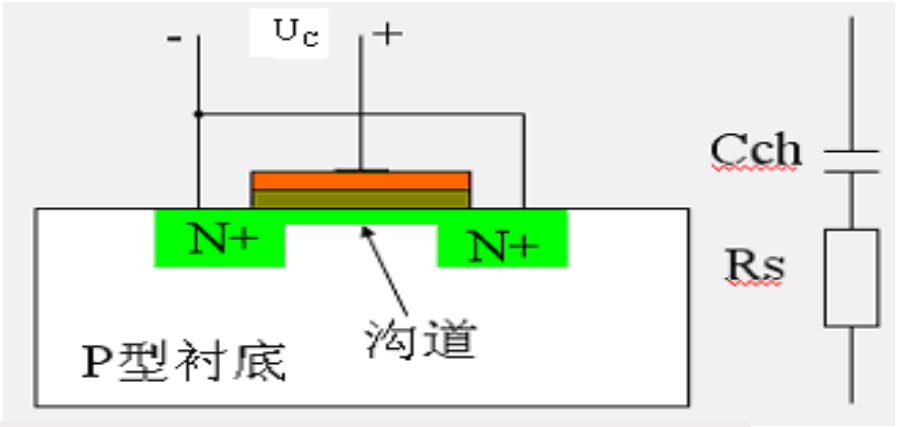
多晶硅—多晶硅电容

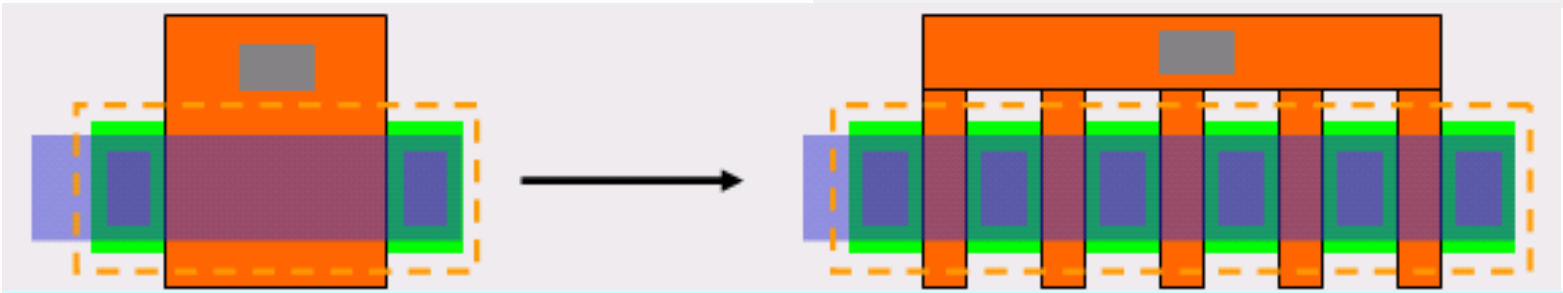
该电容制作在场区上，其两个电极分别是两层多晶硅，中间的介质为氧化层。此类电容的线性特性和底板寄生电容与多晶硅—扩散区电容相似，该电容的典型值为0.7fF/μm^2。



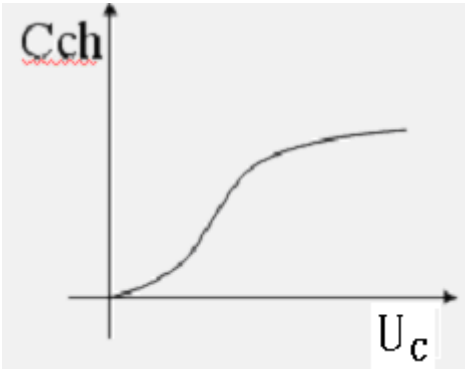
MOS电容 —栅极与沟道间电容C_{ch}

$$C = \varepsilon \frac{S}{t_{ox}} = \varepsilon \frac{WL}{t_{ox}}$$





减小沟道电阻的方法

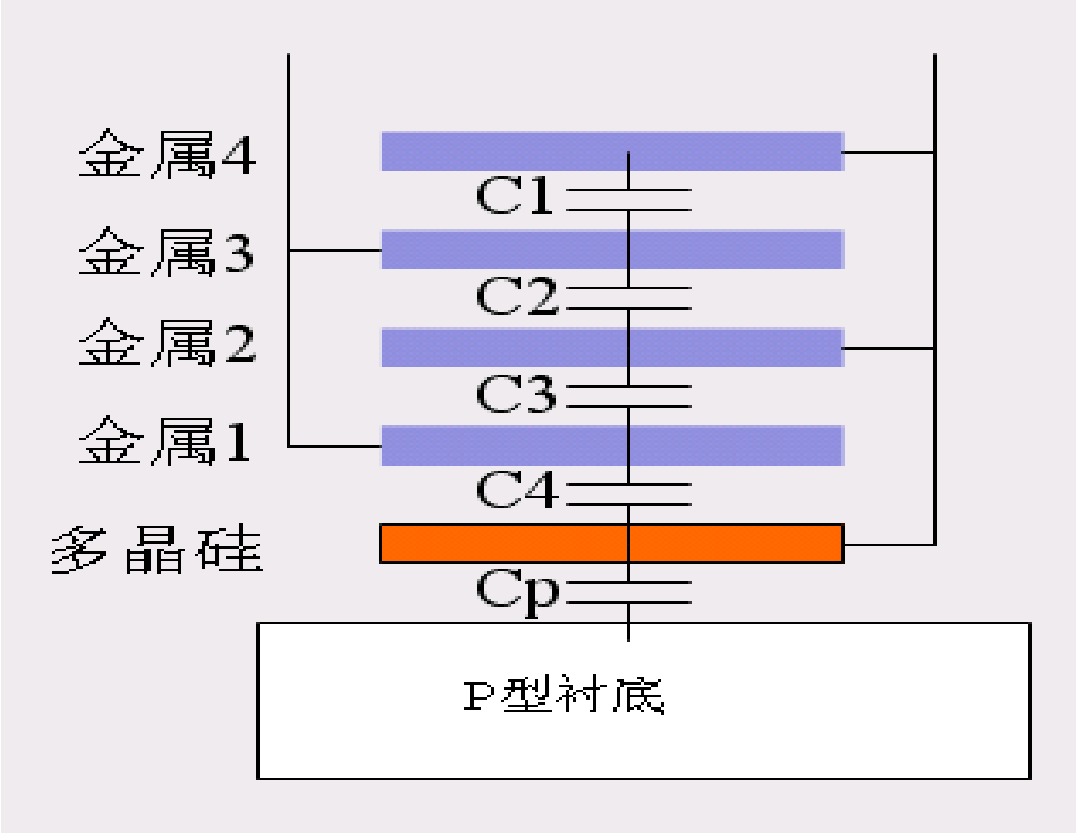


该MOS电容是非线性电容，
该电容主要用于电源滤波电路。

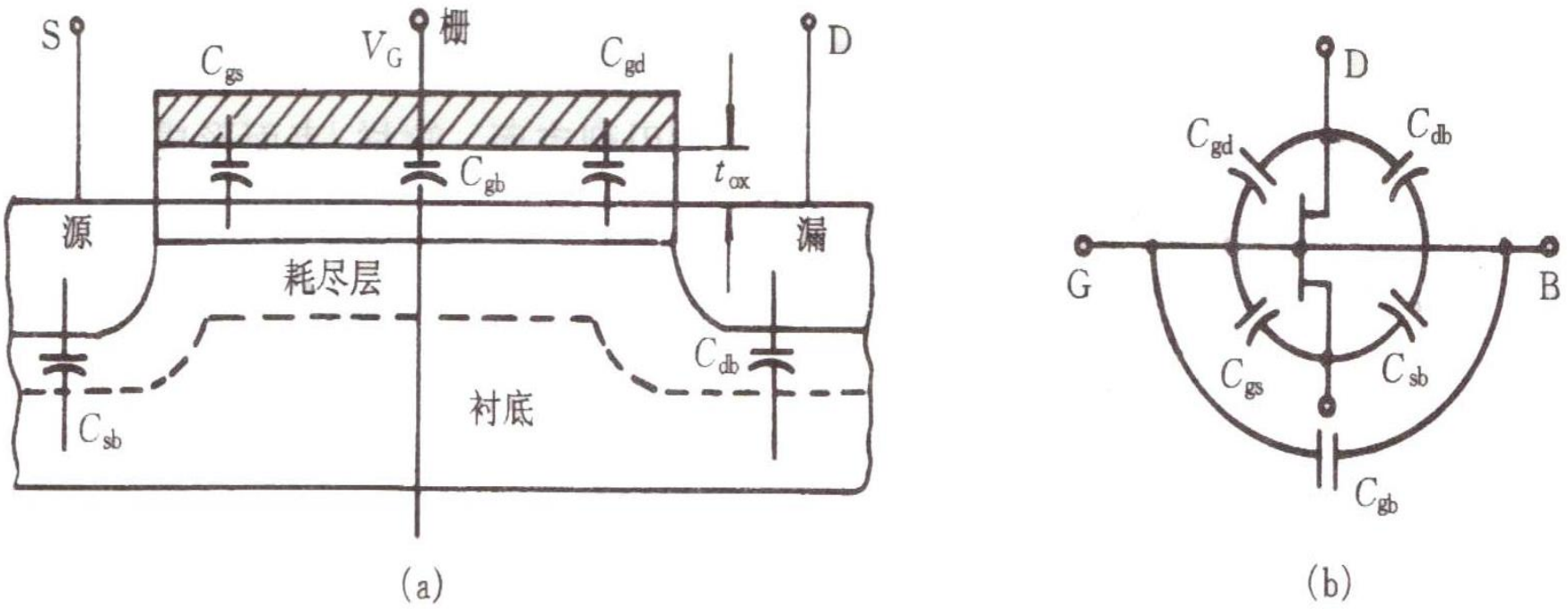
“夹心”电容

总电容值为$C=C_1+C_2+C_3+C_4$

该电容是一种线性电容，底板寄生电容为$C_p=(50\%\sim60\%)C$



MOS管的极间电容和寄生电容



- 栅极和沟道之间的氧化层电容$C=C_{ox}A=C_{ox}WL$
- 衬底和沟道之间的耗尽层电容C_{gb}
- 多晶硅与源、漏之间交叠形成的电容C_{gd}、C_{gs}
- 源、漏与衬底之间的结电容C_{sb}、C_{db}

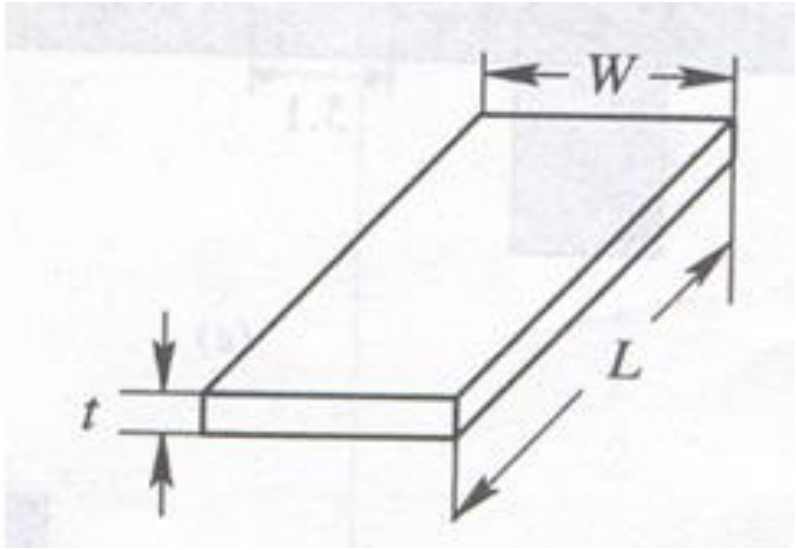
3.4 集成电阻

方块电阻的概念

$$R = \rho\frac{L}{Wt} = \frac{\rho}{t}\left(\frac{L}{W}\right) = R_{\square}\left(\frac{L}{W}\right)$$

$$R_{\square} = \frac{\rho}{t}$$ ——方块电阻

- ρ——导电材料的电阻率；
- L——矩形薄层电阻的长度；
- W——矩形薄层电阻的宽度；
- t——矩形薄层电阻的厚度



方块电阻的概念

常用材料的方块电阻

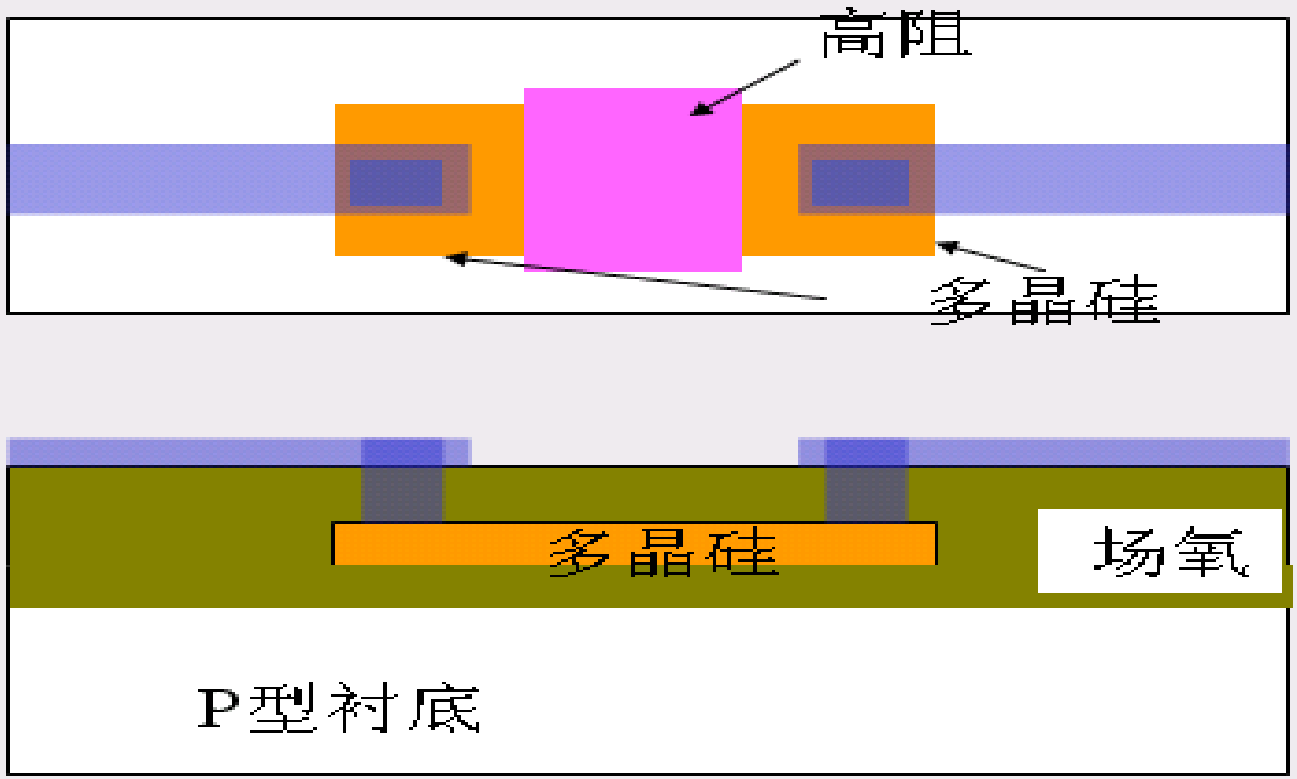
(单位 Ω/□)

材料	最小值	典型值	最大值
互连金属	0.05	0.07	0.1
顶层金属	0.03	0.04	0.05
多晶硅	15	20	30
硅—金属化合物	2	3	6
扩散层(N+, P+)	10	25	100
硅化合物扩散	2	3	10
N阱/P阱	1k	2k	5k

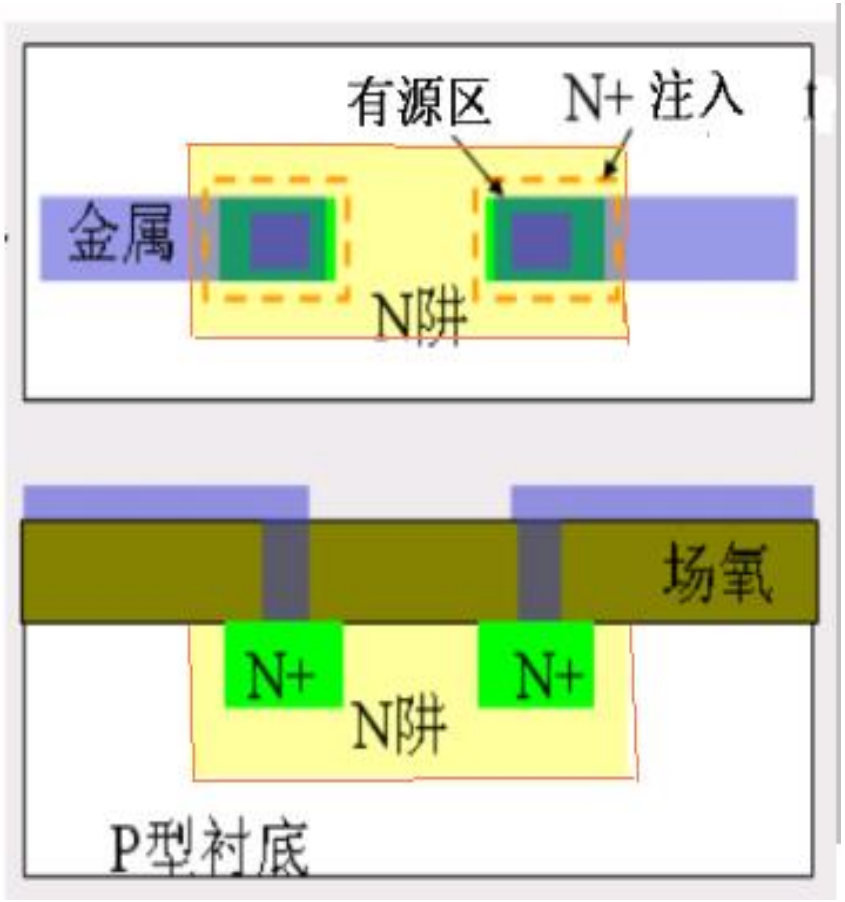
多晶硅电阻

$$R = R_{\square\text{poly}}\left(\frac{L}{W}\right)$$

$$R_{\square\text{poly}} = 1k\Omega/\square$$



N—P阱电阻



$$R_{\square\text{Well}} = 0.82k\Omega/\square$$

$$R = R_{\square}(1+\alpha_1 V+\alpha_2 V^2)$$

$$\alpha_1 \approx 8.5\times10^{-3}\,\mathrm{V}^{-1}$$

$$\alpha_2 = 9.8\times10^{-4}\,\mathrm{V}^{-2}$$

MOS管电阻和导线电阻

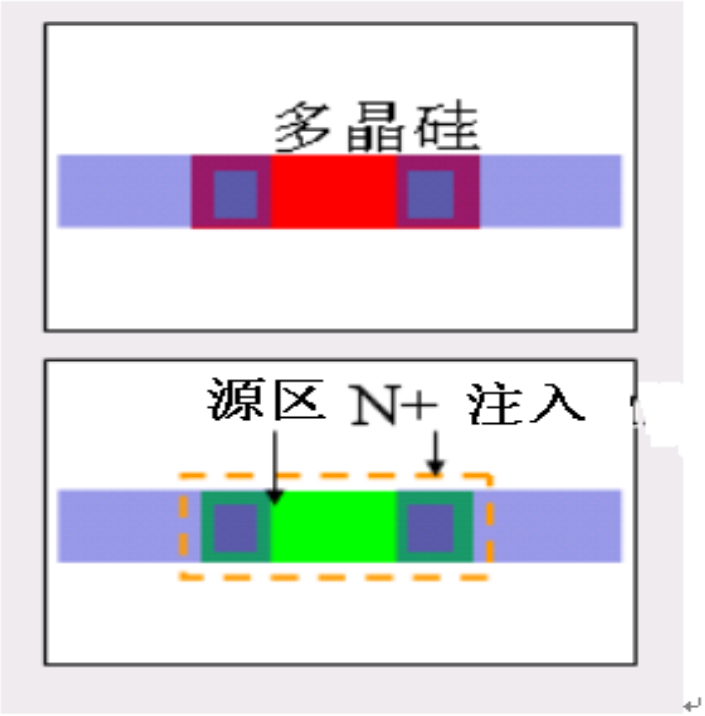
- 工作在可变电阻区的MOS管可用作电阻

$$R_{ds}=\frac{\partial u_{DS}}{\partial i_D}=\frac{1}{\mu_n C_{ox}\left(\frac{W}{L}\right)(u_{GS}-U_{TH})}$$

- 导线电阻

$$R_{□(多晶硅导线)}=10\sim 15\Omega/□$$

$$R_{□(扩散区N+)}=20\sim 30\Omega/□$$

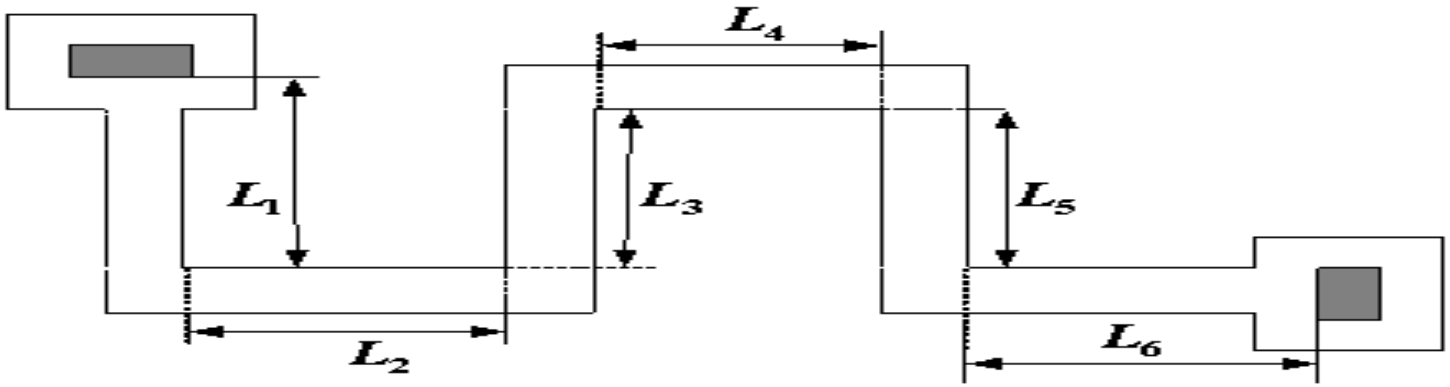


拐弯电阻

- 拐弯电阻计算公式为

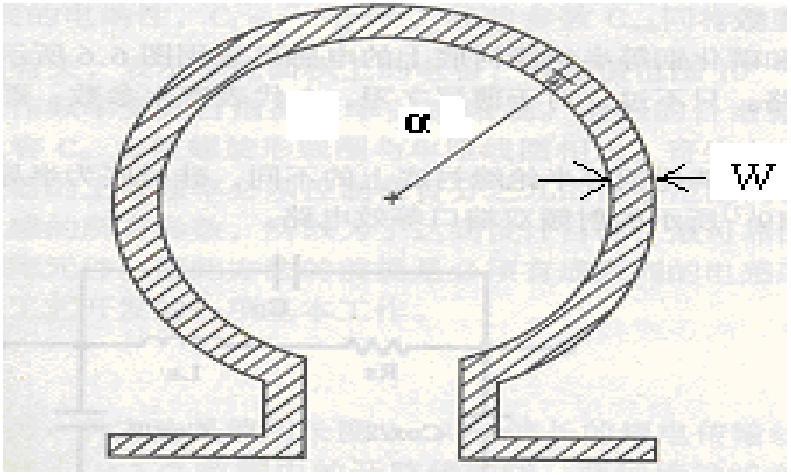
$$R = R_{\square}\left(\frac{L}{W} + 2K_1 + nK_2\right)$$

- K_1——端口修正因子(一般取0.35~0.65)
- K_2——拐角修正因子(一般取0.5)
- n ——拐角数
- W ——线宽

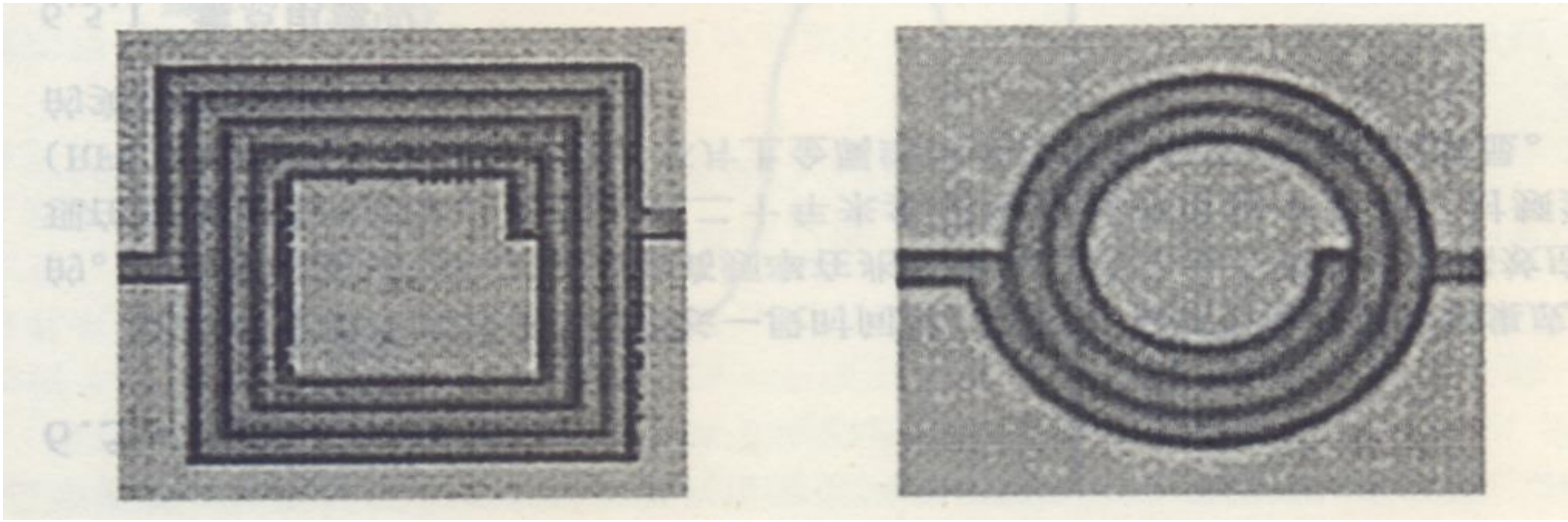


3.5 集成电感

集成电感



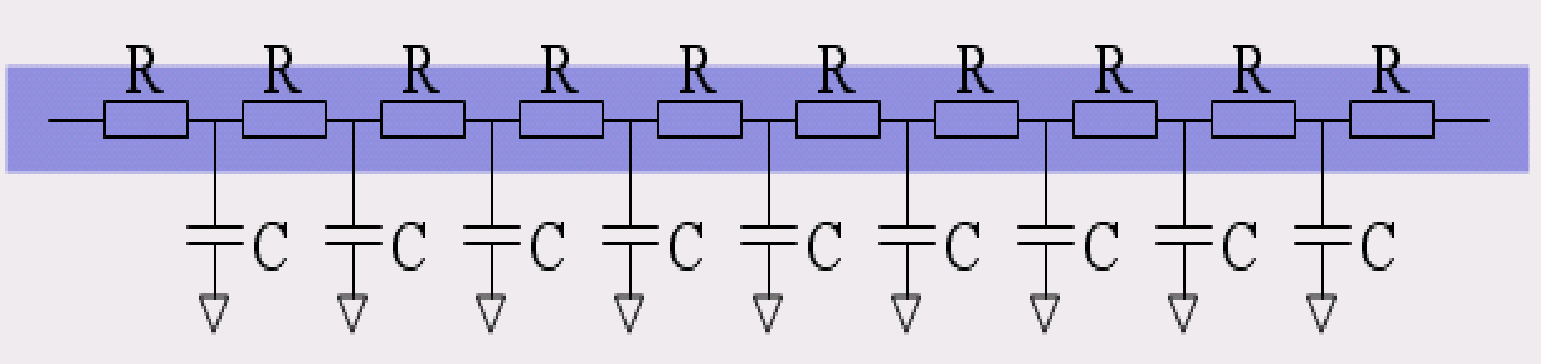
$$L = 1.26\alpha\left[L_n\left(\frac{8\pi\alpha}{W}\right) - 2\right](\text{pH})$$

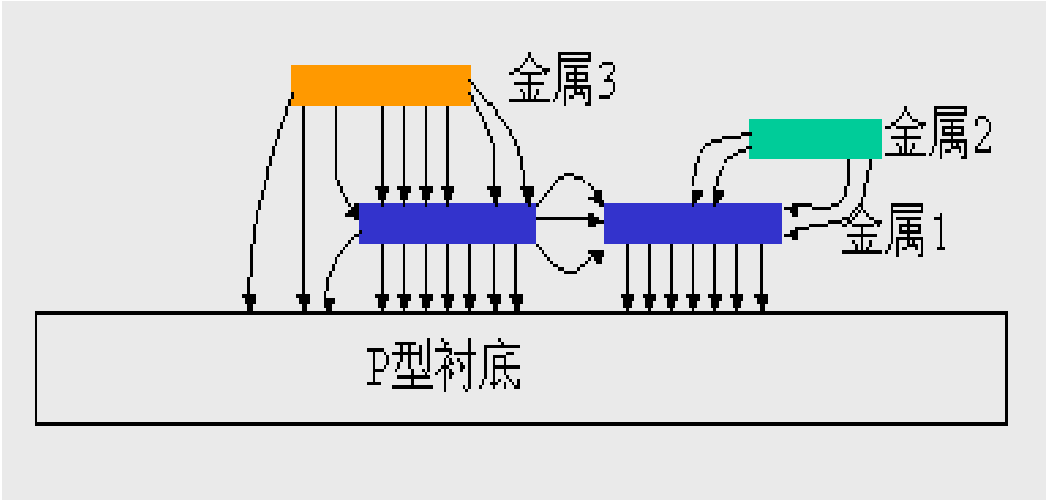


3.6 连线

连线

连线的寄生模型，图中R为串联寄生电阻，C为并联寄生电容。连线越长，寄生参数影响越大。





复杂互连线的寄生电容

3.7 MOS管的spice模型参数

MOS管的spice模型参数

- spice(simulation program with integrated circuit emphasis)是最为普遍的电路级模拟程序
- 目前许多数模混合计算机仿真软件的内核都是spice模型。计算机仿真(模拟)的精度很大程度上取决于器件模型参数的准确性和算法的科学先进性。了解spice模型参数的含义对于正确设计集成电路是十分重要的。

西安電子科技大學

第四章

CMOS数字集成电路设计基础

本章提要

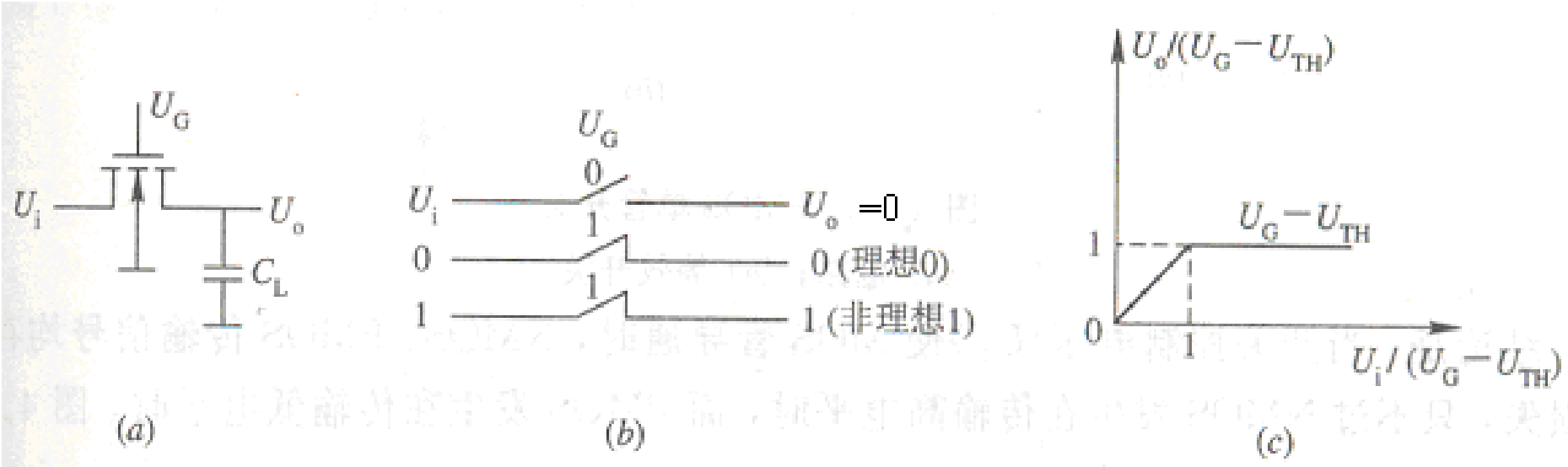
- 4.1　MOS开关及CMOS传输门
- 4.2　CMOS反相器
- 4.3　全互补CMOS集成门电路
- 4.4　改进的CMOS逻辑电路
- 4.5　移位寄存器、锁存器、触发器、I/O单元

西安電子科技大學

4.1 MOS开关及CMOS传输门

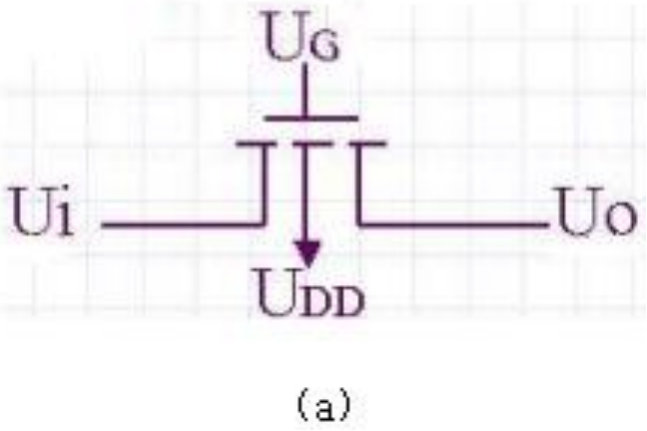
单管MOS开关

- 单管NMOS开关

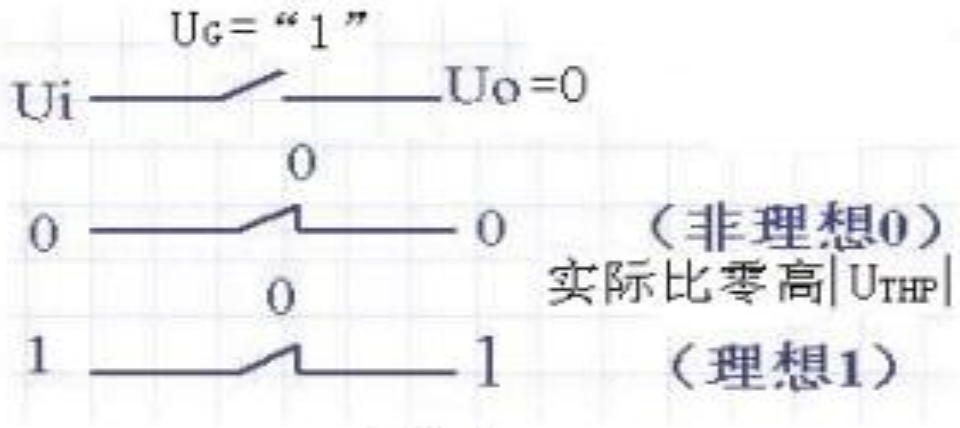


单管MOS开关

- 单管PMOS开关



(a)

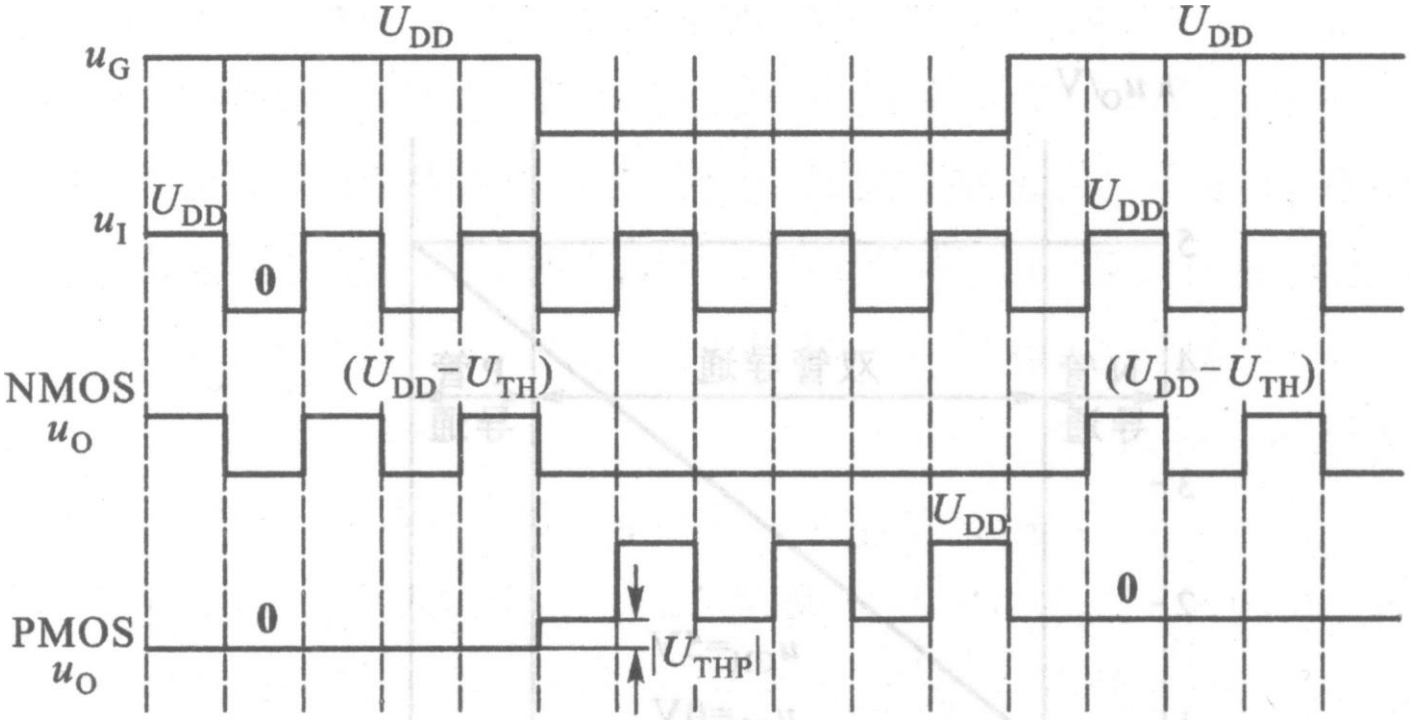


(b)

单管MOS开关

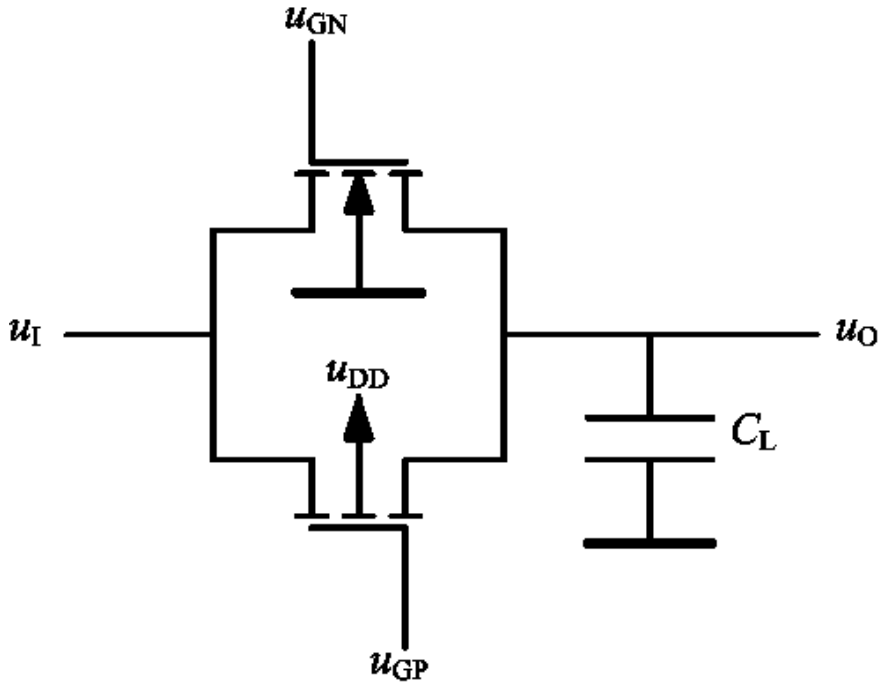
- 结论

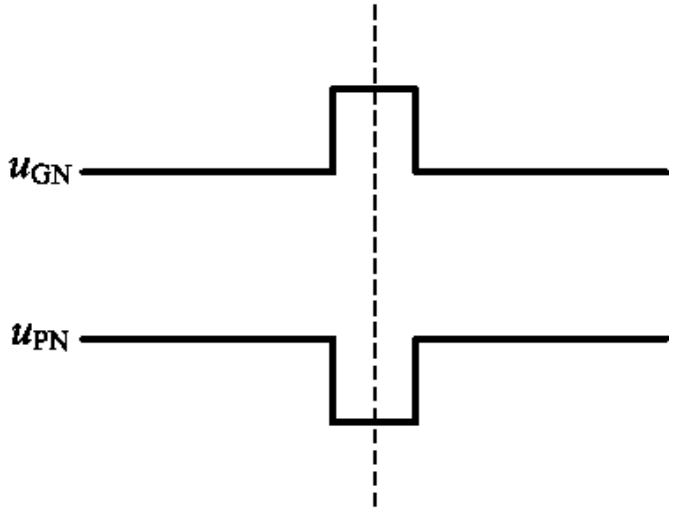
当开关控制电压u_G使MOS管导通时，NMOS、PMOS传输信号均存在阈值损失，只不过NMOS发生在传输高电平时，而PMOS发生在传输低电平时。



CMOS传输门

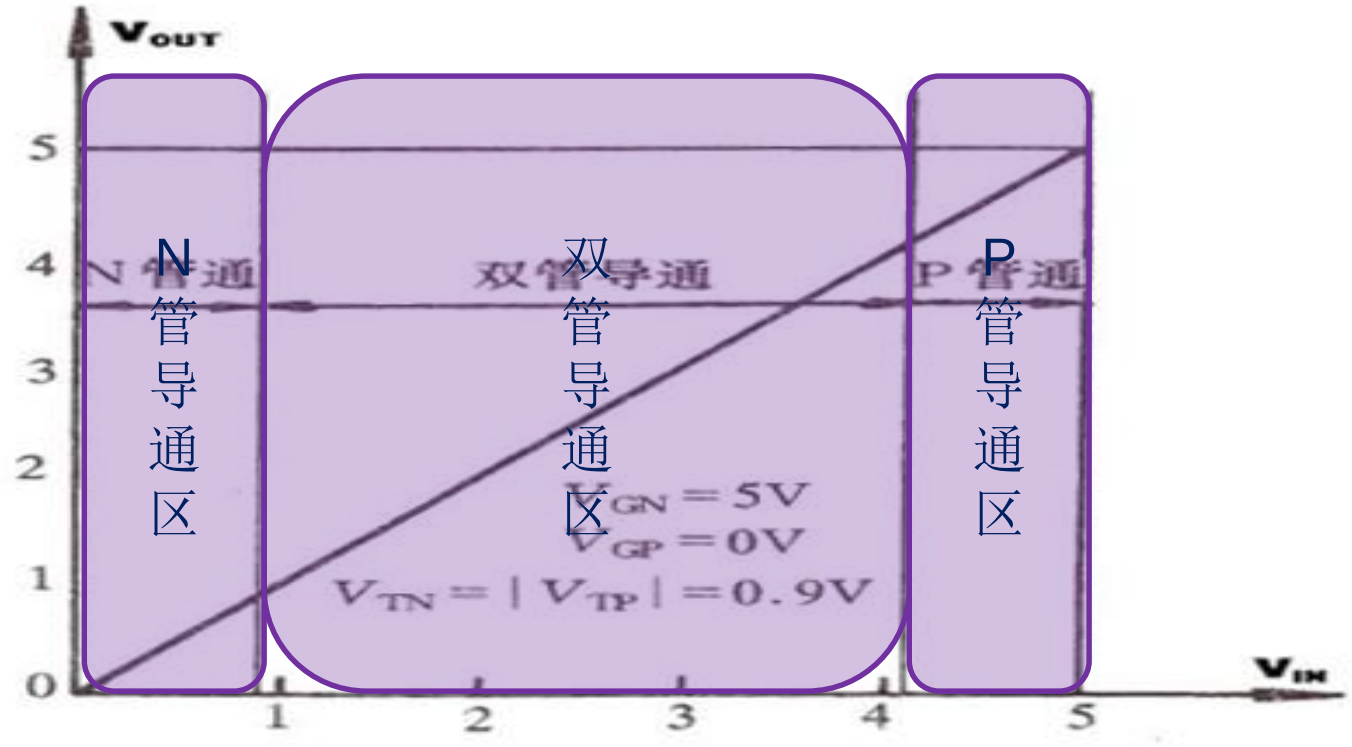
根据NMOS和PMOS单管开关的特性，将其组合在一起，形成一个互补的CMOS传输门，这是一个没有阈值损失的理想开关。





CMOS传输门

CMOS传输门的直流传输特性如图所示，由于它利用了CMOS的互补作用，传输低电平靠N管，传输高电平靠P管，可以使信号做到无损传输。



CMOS传输门

- CMOS传输门的设计

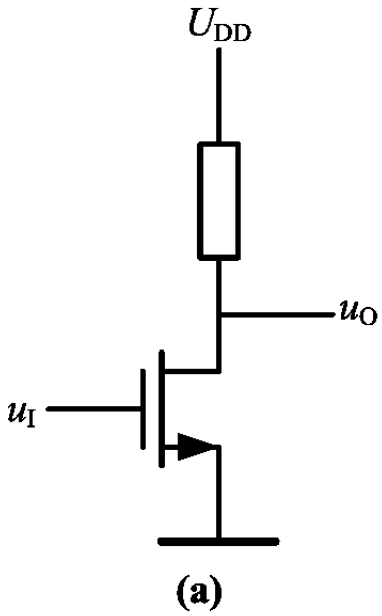
1. 为保证导电沟道与衬底的隔离，N管的衬底必须接地，P管的衬底必须接电源U_{DD}。
2. 沟道电流i_D与管子的宽长比(W/L)成正比，为使传输速度快，要求i_D大些，沟道长度L取决于硅栅多晶硅条的宽度，视工艺而定。一般L取工艺最小宽度(2λ)，那么，要使i_D大，就要将沟道宽度W设计的大些。

4.2 CMOS反相器

CMOS反相器相当于非门，是数字集成电路中最基本的单元电路。搞清楚CMOS反相器的特性，可为一些复杂数字电路的设计打下基础。

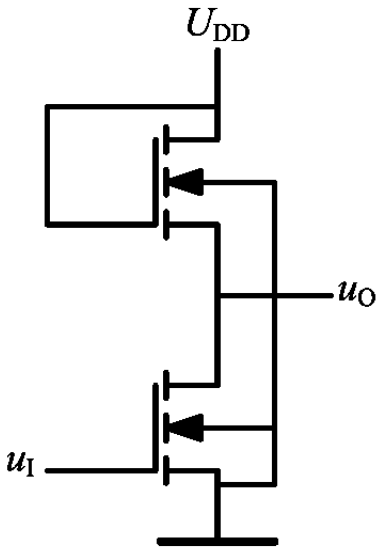
CMOS反相器

- 以下是几种反相器：



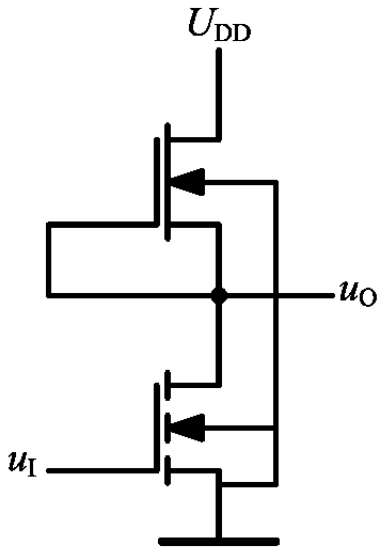
(a)

电阻反相器



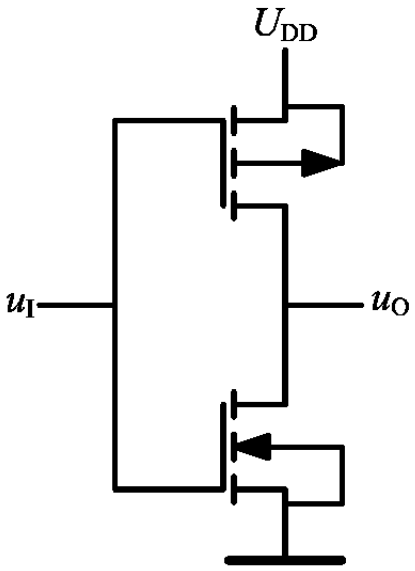
(b)

用增强型NMOS
做负载的E/E反相器



(c)

用耗尽型NMOS
做负载的E/D反相器



(d)

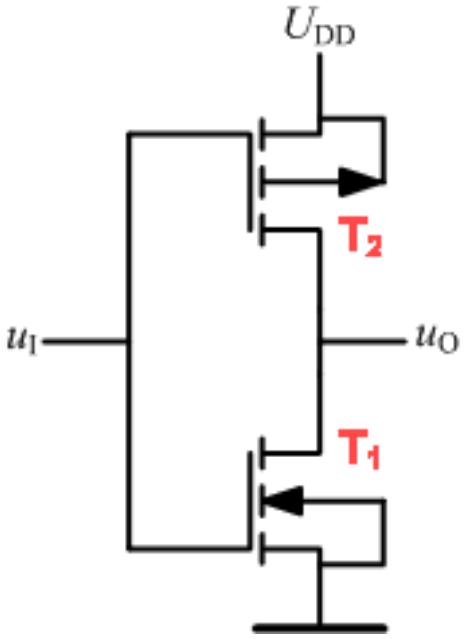
CMOS反相器

CMOS反相器功耗

- 静态功耗
  - 当u_I = 0时，T_1截止，T_2导通，$u_O = U_{DD}$。 （“1”状态）
  - 当u_I = 1时，T_1导通，T_2截止，u_O = 0。 （“0”状态）

结论：无论u_I是“0”还是“1”，
总有一个MOS管是截止的，
即$i_D = 0$。
故静态功耗为：

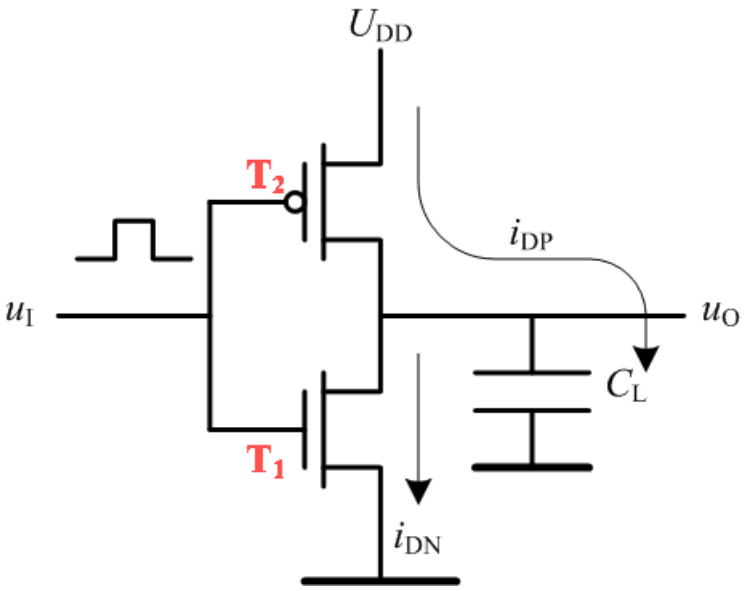
$$P_S = i_D \times U_{DD} \approx 0$$

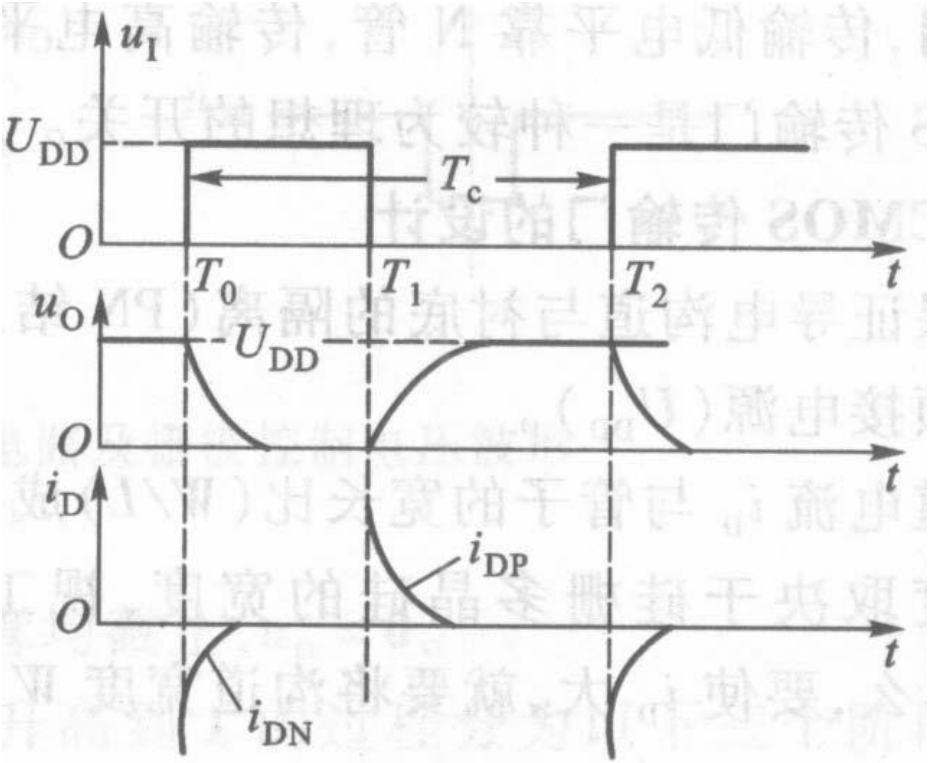


CMOS反相器功耗

- 动态功耗(瞬态功耗)

对负载电容C_L充放电的动态功耗P_{D1}—交流开关功耗





CMOS反相器功耗

- 动态功耗P_{D1}—交流开关功耗

一周内C_L充放电使管子产生的平均功耗

$$P_{D1}=\frac{1}{T_c}\left[\int_0^{T_1}\left(i_{DN}\times U_{DSN}\right)dt+\int_{T_1}^{T_2}\left(i_{DP}\times U_{DSP}\right)dt\right]$$

式中T_c为输入信号周期

$$P_{D1}=\frac{C_L}{T_c}\left[\int_{U_{OH}}^{U_{OL}}\left(u_O-U_{DD}\right)d\left(u_O-U_{DD}\right)+\int_{U_{OL}}^{U_{OH}}u_O du_O\right]$$

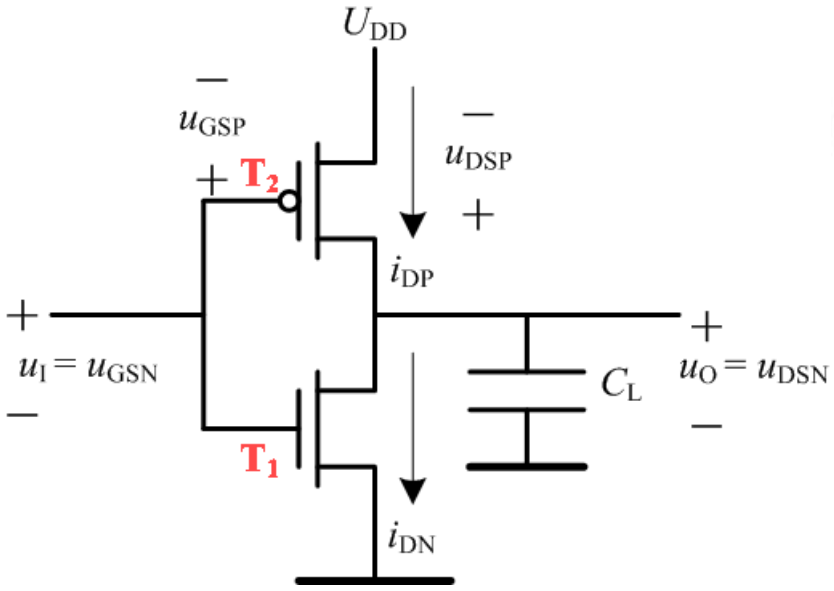
$$=C_L f_c\left(U_{OH}-U_{OL}\right)U_{DD}$$

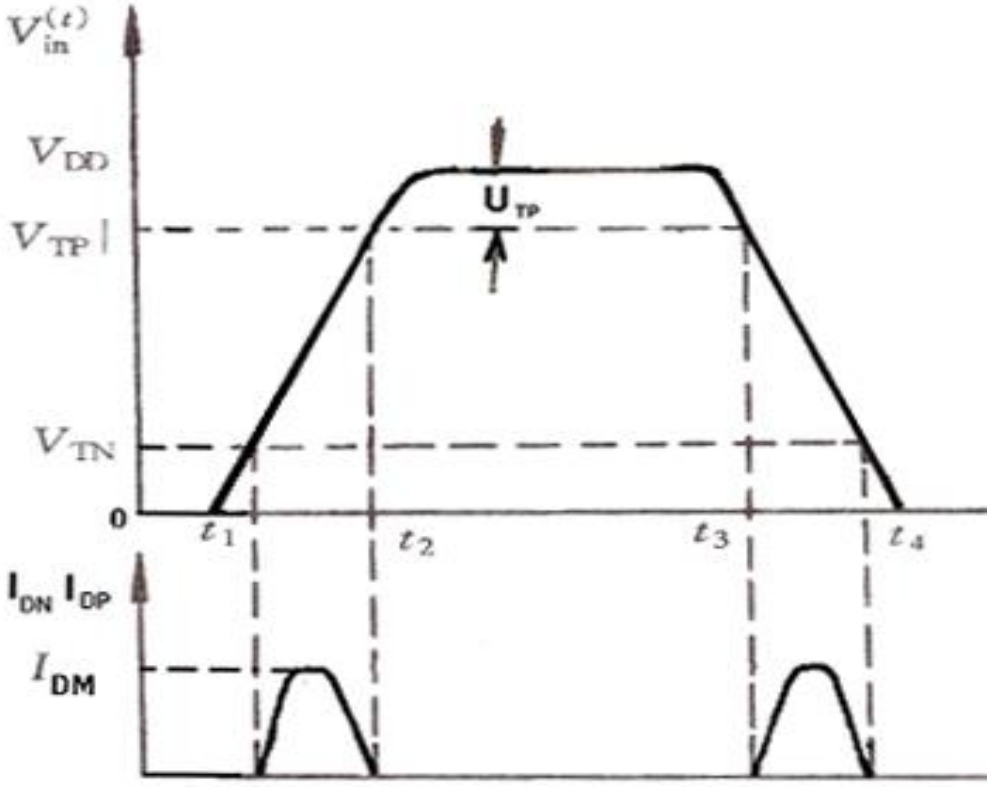
$$=C_L f_c U_{DD}^2$$

CMOS反相器功耗

- 动态功耗

u_I为非理想阶跃波形时引入的动态功耗P_{D2}—直流开关功耗





CMOS反相器功耗

- 动态功耗P_{D2}—直流开关功耗

$$P_{D2} \approx \frac{1}{T_c}\left[\int_{t_1}^{t_2}\frac{I_{DM}}{2}U_{DD}dt+\int_{t_3}^{t_4}\frac{I_{DM}}{2}U_{DD}dt\right]$$

$$=\frac{1}{2}I_{DM}U_{DD}f_c(t_r+t_f)$$

注：I_{DM}是贯穿NMOS管和PMOS管电流的峰值，其平均值约为$I_{DM}/2$。

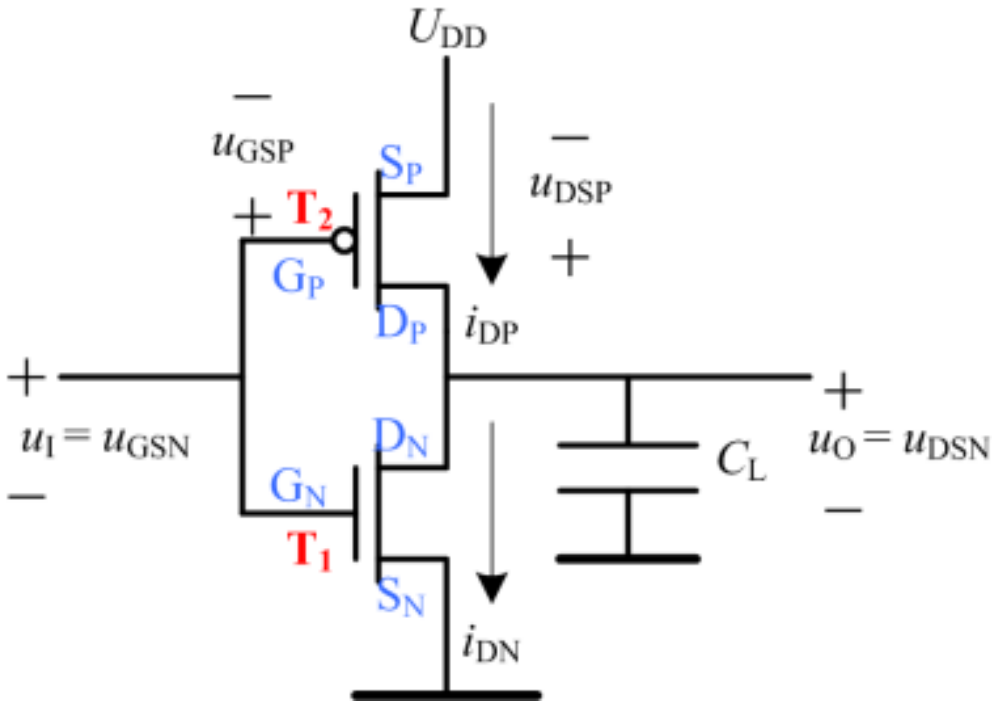
$$I_{DM} \approx \frac{\mu_n C_{ox}}{2}\left(\frac{W}{L}\right)_N (U_{DD}-U_{THN})^2 = \frac{\mu_p C_{ox}}{2}\left(\frac{W}{L}\right)_P (U_{DD}-U_{THP})^2$$

- 反相器总功耗

$$P_D = P_{D1} + P_{D2}$$

CMOS反相器的直流传输特性

- 随着u_I由小变大(0→U_{DD})，反相器的工作状态可分为5个阶段来描述。



$$u_{GSN} = u_I$$

$$u_{DSN} = u_O$$

$$u_{GSP} = -(U_{DD} - u_I)$$

$$u_{DSP} = -(U_{DD} - u_O)$$

CMOS反相器的直流传输特性

- 分段讨论：

- AB段

  在AB段，$0 < u_I < U_{THN}, i_{DN} = 0$，N管截止，P管非恒流(饱和)导通，有：$u_O = U_{OH} = U_{DD}$

- BC段

$$U_{THN} < u_I < u_O + |U_{THP}|$$

即 $$u_{GDP} = |u_I - u_O| < |U_{THP}|$$

此时N管恒流(饱和)导通，P管线性导通，输出电阻r_{DSP}很小，电路相当于一个小增益放大器。

CMOS反相器的直流传输特性

- CD段

当u_I进一步增大，且满足$u_O+|U_{THP}|\le u_I\le u_O+|U_{THN}|$时，两管的栅、漏区进入预夹断状态，同时饱和导通。N管和P管的电流相等，根据电流方程：

$$i_{DN}=\frac{\mu_n C_{ox}}{2}\left(\frac{W}{L}\right)_N(U_{DD}-U_{THN})^2$$

$$i_{DP}=\frac{\mu_p C_{ox}}{2}\left(\frac{W}{L}\right)_P(U_{DD}-U_{THP})^2$$

令 $\beta_N=\mu_n C_{ox}\left(\frac{W}{L}\right)_N$ (N管的导电因子)

$\beta_P=\mu_P C_{ox}\left(\frac{W}{L}\right)_P$ (P管的导电因子)

CMOS反相器的直流传输特性

- CD段

则 $i_{DN}=\frac{\beta_N}{2}(u_I-U_{THN})^2$

$$i_{DP}=\frac{\beta_P}{2}(u_I-U_{DD}-U_{THP})^2$$

由于$i_{DN}=i_{DP}$，可以求得反相器的阈值电压U_{iT}为

$$U_{iT}=U_{THN}+\frac{U_{DD}-U_{THN}+U_{THP}}{1+\sqrt{\beta_N/\beta_P}}$$

CMOS反相器的直流传输特性

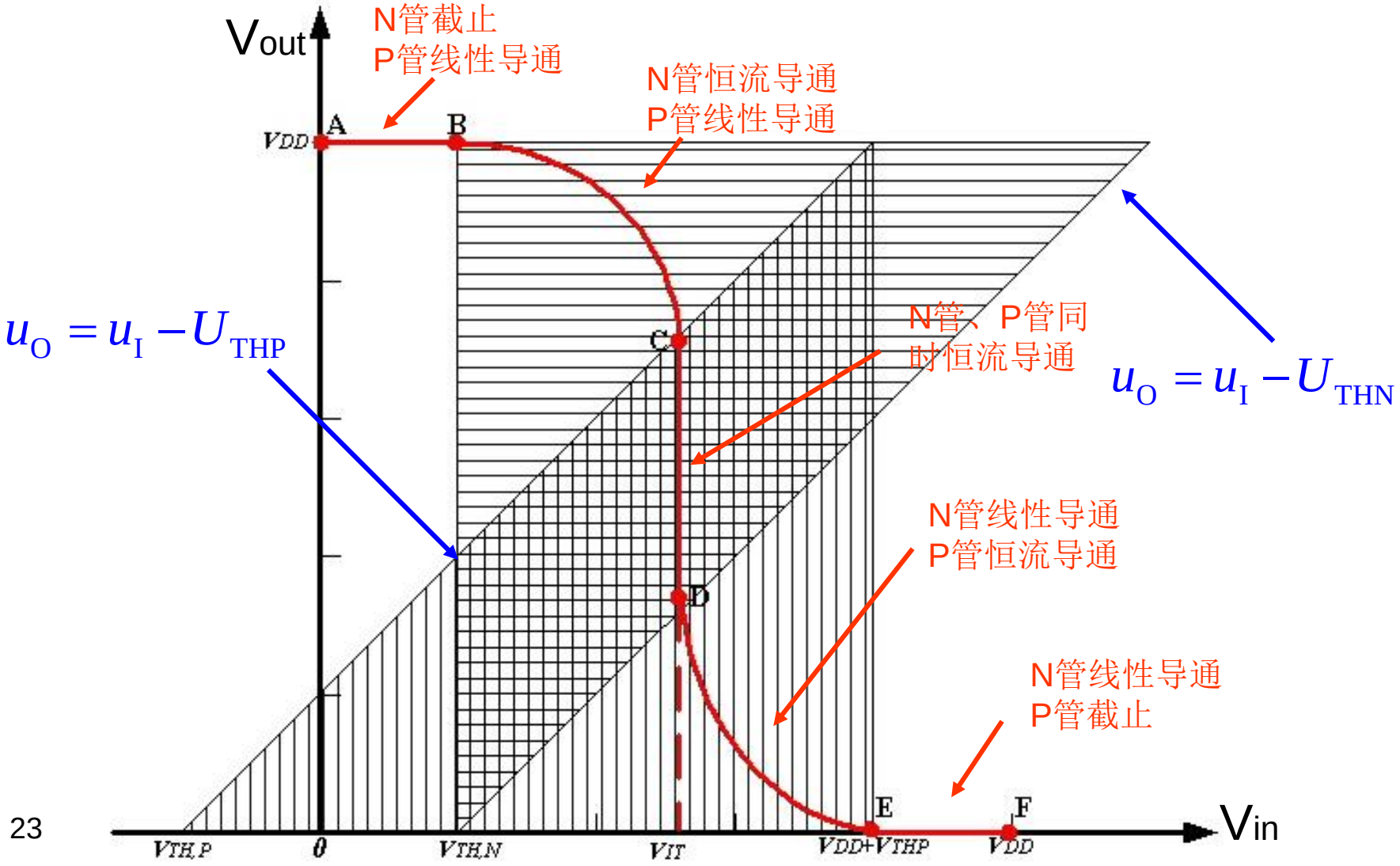
- DE段

随着u_I继续上升，当满足 $u_O + U_{THN} < u_I < U_{DD} + U_{THP}$ 时，N管进入线性导通区，P管仍然维持在饱和导通区。

- EF段

随着Ui 的进一步增大，当满足$U_{DD} + U_{THP} \le u_I \le U_{DD}$ 时，P管截止，N管维持非饱和导通而导致$u_O = 0$。

CMOS反相器的直流传输特性



CMOS反相器的直流传输特性

- CMOS反相器三个工作区各电压之间的关系

	截止区	线性区	饱和区
P型器件	$u_{GSP} > u_{THP}$ $u_I > U_{THP} + U_{DD}$	$u_{GSP} < U_{THP}$ $u_I < U_{THP} + U_{DD}$ $u_{GDP} < U_{THP}$ $u_I - u_O < U_{THP}$	$u_{GSP} < U_{THP}$ $u_I < U_{THP} + U_{DD}$ $u_{GDP} > U_{THP}$ $u_I - u_O > U_{THP}$
N型器件	$u_{GSN} < U_{THN}$ $u_I < U_{THN}$	$u_{GSN} > U_{THN}$ $u_I > U_{THN}$ $U_{GDN} > U_{THN}$ $u_I - u_O > U_{THN}$	$u_{GSN} > U_{THN}$ $u_I > U_{THN}$ $u_{GDN} < U_{THN}$ $u_I - u_O < U_{THN}$

CMOS反相器的直流传输特性

- CMOS反相器直流工作特性总结

区域	条 件	PMOS	NMOS	输出电压
AB	$0 \le u_I \le U_{THN}$	线性	截止	$u_O = U_{DD}$
BC	$U_{THN} \le u_I < U_{DD}/2$	线性	饱和	$u_O = (u_I + 1) + \sqrt{15 - 6u_I}$
CD	$u_I = U_{DD}/2$	饱和	饱和	$u_O \neq f(u_I)$
DE	$U_{DD}/2 < u_I \le U_{DD} + U_{THP}$	饱和	线性	$u_O = (u_I - 1) - \sqrt{6u_I - 15}$
EF	$u_I \ge U_{DD} + U_{THP}$	截止	线性	$u_O = 0$

注：计算条件是U_{DD}=+5V，U_{THP}=-1V，U_{THN}=+1V，β_N/β_P=1。

CMOS反相器的噪声容限

噪声容限——电路在噪声干扰下，逻辑关系发生偏离(误动作)的最大允许输入值。该参数用于确定：当门的输出不受影响时，其输入端允许的噪声电压。

几种直流噪声容限的不同定义

1. 由输入阈值电压定义的噪声容限
2. 由单位增益点定义的噪声容限

CMOS反相器的噪声容限

- 以输入阈值电压U_{iT}为界，低端的噪声容限为U_{NL}，高端的噪声容限为U_{NH}，则

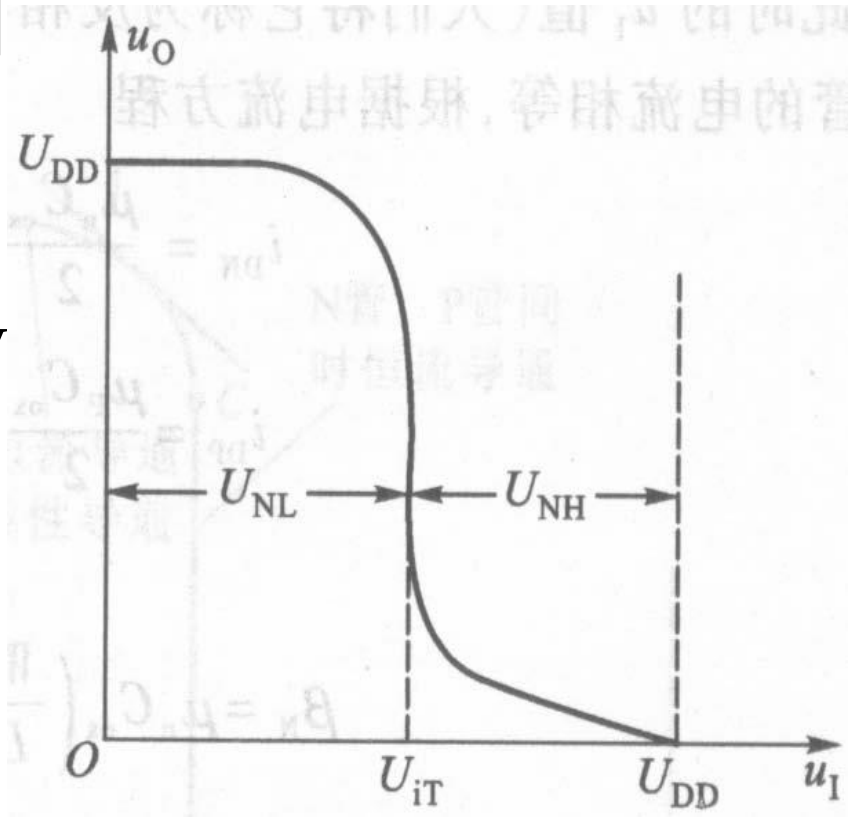
$$U_{NL}=U_{iT}$$

$$U_{NH}=U_{DD}-U_{iT}$$

若要使$U_{NL}=U_{NH}$，当U_{DD}=5V

$$U_{iT}=\frac{U_{DD}}{2}=U_{NL}=U_{NH}=2.5\text{V}$$

（最佳噪声容限）



CMOS反相器的噪声容限

又 $U_{iT}=U_{THN}+\dfrac{U_{DD}-U_{THN}+U_{THP}}{1+\sqrt{\beta_N/\beta_P}}=\dfrac{U_{DD}}{2}$

若P管阈值电压U_{THP}与N管阈值电压U_{THN}相等，得 $\beta_N=\beta_P$

导电因子 $\beta_N=\mu_n C_{ox}\left(\dfrac{W}{L}\right)_N=\beta_P=\mu_p C_{ox}\left(\dfrac{W}{L}\right)_P$

则 $\left(\dfrac{W}{L}\right)_P=\dfrac{\mu_n}{\mu_p}\left(\dfrac{W}{L}\right)_N=(2-4)\left(\dfrac{W}{L}\right)_N$

因此

$$\begin{cases}L_P=L_N\\ W_P=(2-4)W_N\end{cases}$$

或

$$\begin{cases}\left(\dfrac{W}{L}\right)_P=\left(\dfrac{W}{L}\right)_N\\ \dfrac{\beta_N}{\beta_P}=\dfrac{\mu_N}{\mu_P}=(2-4)\end{cases}$$

，U_{iT}偏小，$U_{NL}<U_{NH}$

CMOS反相器的噪声容限

几种直流噪声容限的不同定义

1. 由输入阈值电压定义的噪声容限
2. 由单位增益点定义的噪声

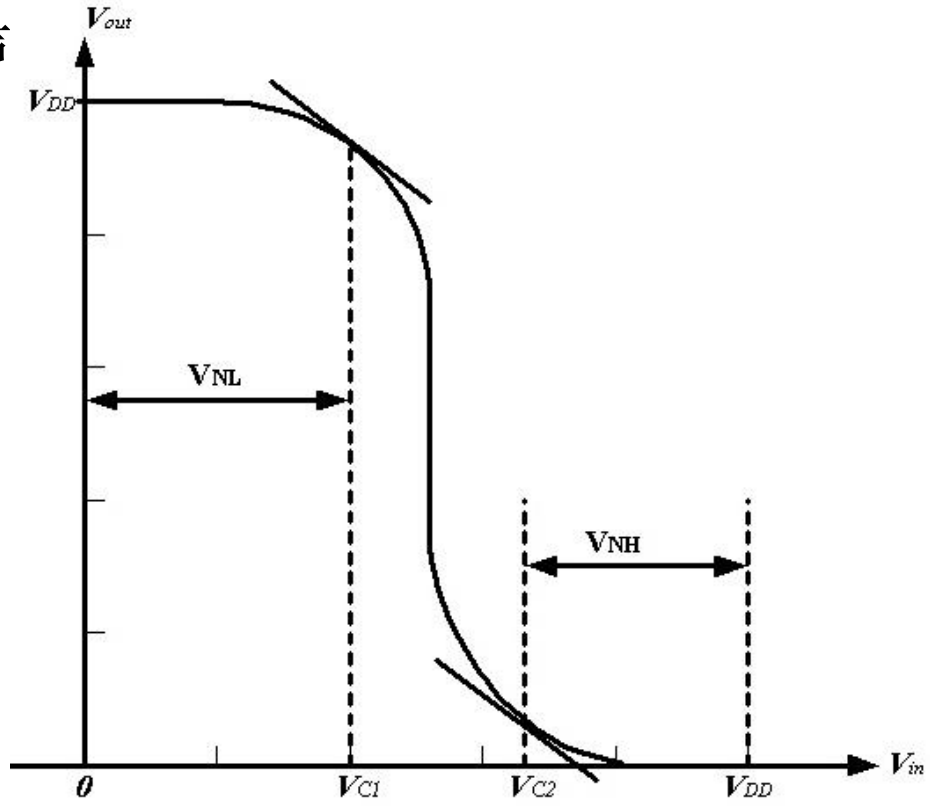
以单位增益点对应的输入电平为所允许的临界电平，它们和理想逻辑电平之间的范围为CMOS电路的直流噪声容限。

- 输入高电平的噪声容限：

$U_{NH}=U_{DD}-V_{C2}=2.125V$

- 输入低电平的噪声容限：

$U_{NL}=V_{C1}=2.125V$



CMOS反相器的噪声容限

- 最佳噪声容限条件下的CMOS反相器设计

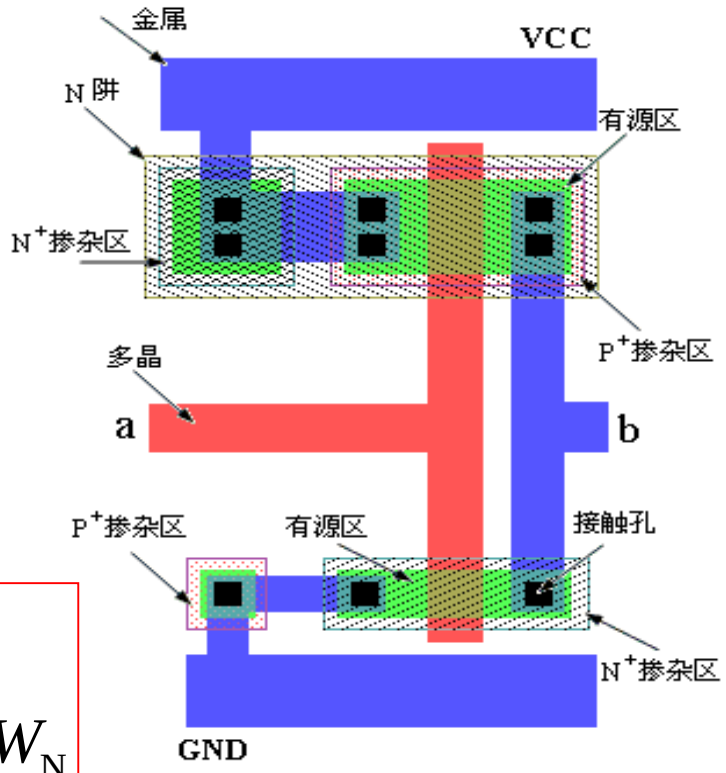
  为了使CMOS反相器获得最佳性能，常采用对称设计。使反相器中的NMOS管和PMOS管性能完全对称。

即满足$U_{THN} = U_{THP}$

$\beta_N=\beta_P$ (跨导系数相等)

$$\mu_n = (2 \sim 4)\mu_p$$

所以在取$L_N = L_P$时，$W_P = (2 \sim 4)W_N$



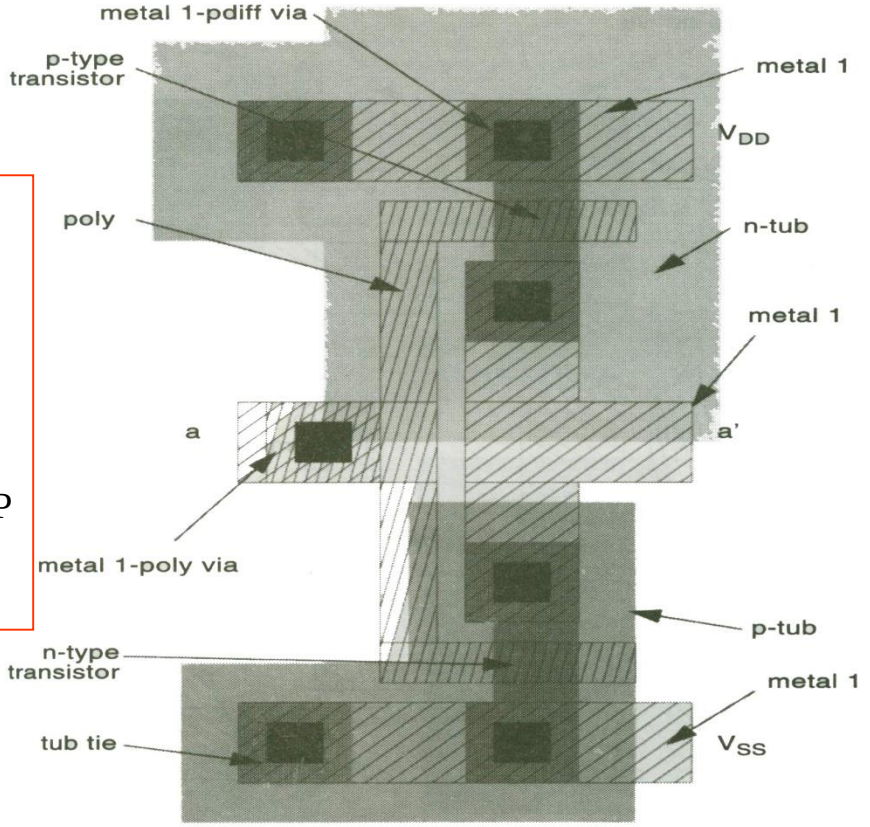
CMOS反相器的噪声容限

- 等尺寸条件下的CMOS反相器设计

若要求 $(\frac{W}{L})_P = (\frac{W}{L})_N$

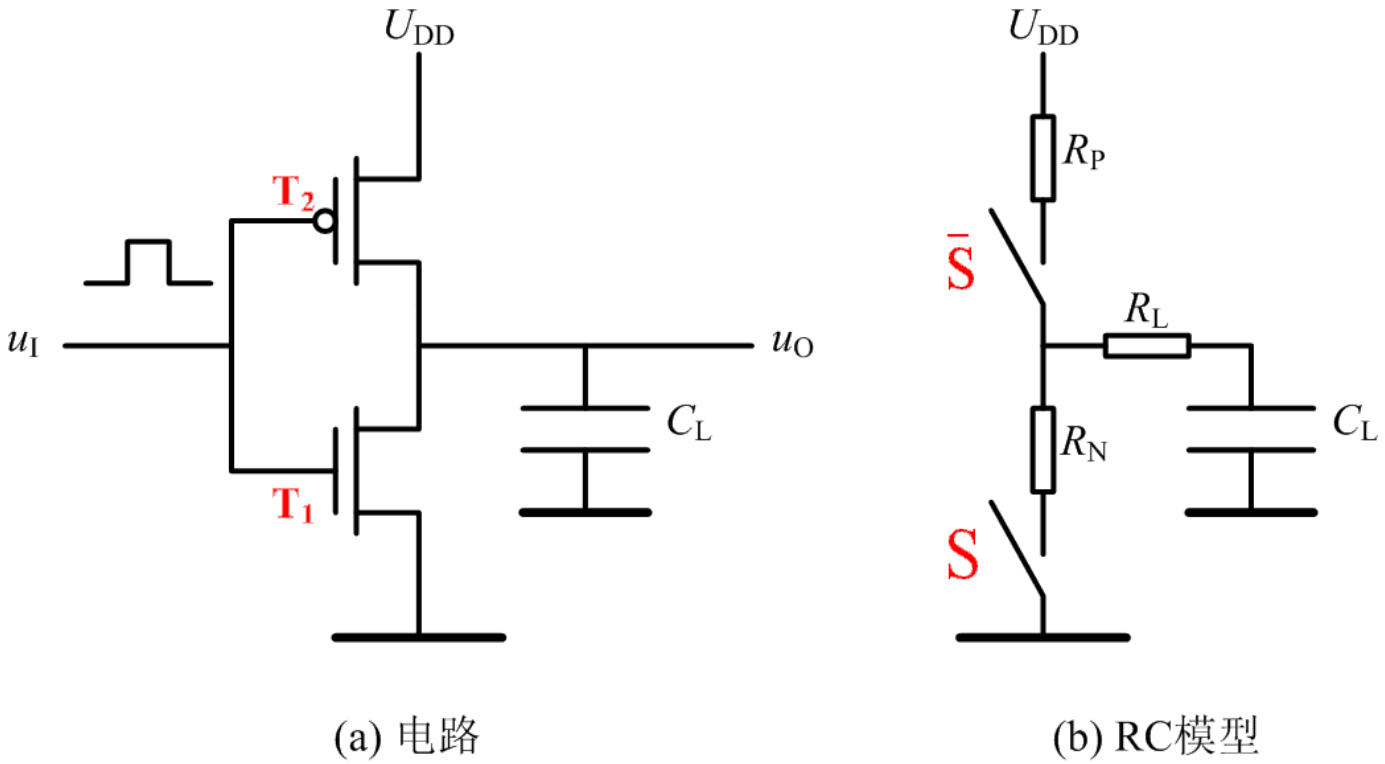
$\because \beta_N = \mu_n C_{ox}\left(\frac{W}{L}\right)_N, \beta_P = \mu_p C_{ox}\left(\frac{W}{L}\right)_P$

$\therefore \beta_N > \beta_P$



CMOS反相器的门延迟及连线延迟

- CMOS反相器的延迟分析模型

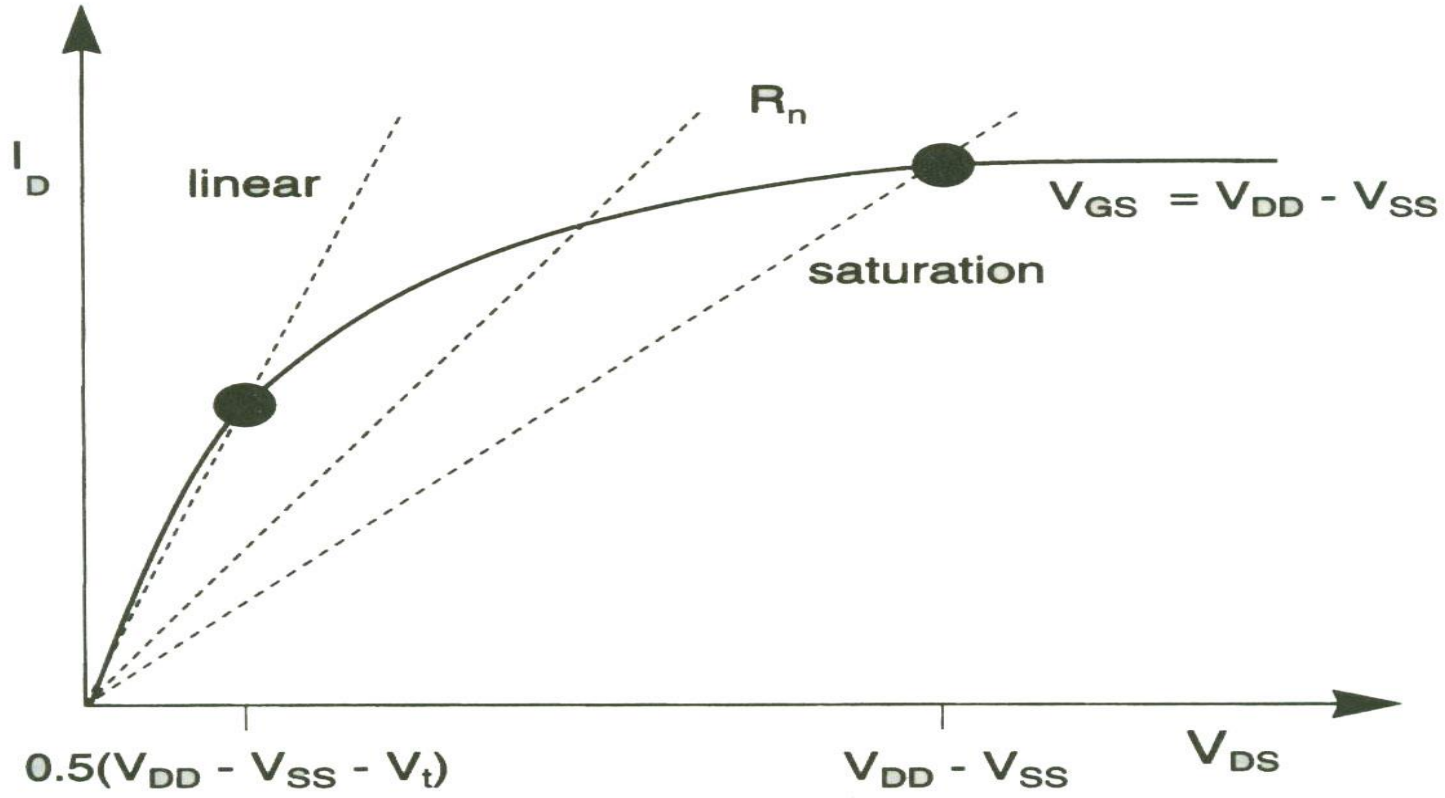


(a) 电路　　(b) RC模型

CMOS反相器的门延迟及连线延迟

- R_p, R_n的估算

在u_I从0到U_{DD}变化的过程中，N管的工作状态由截止区→饱和区→线性区变化。



CMOS反相器的门延迟及连线延迟

- R_p, R_n的估算

  对于NMOS管

  - 线性区电压：$u_{lin}=(U_{DD}-U_{THN})/2$，而$R_{lin}=u_{lin}/i_{lin}$;
  - 饱和区电压：$u_{sat}=U_{DD}$，而$R_{sat}=u_{sat}/i_{sat}$;

  取其平均值作为NMOS管的等效电阻R_N

$$R_N=(R_{lin}+R_{sat})/2$$

根据线性区与饱和区NMOS管的电流方程，可以得到计算R_N的近似公式：

$$R_N=(2.5\sim4)/\beta_N(U_{DD}-U_{THN})$$

依据同样的方法可以推导出计算R_P的近似公式。

CMOS反相器的门延迟及连线延迟

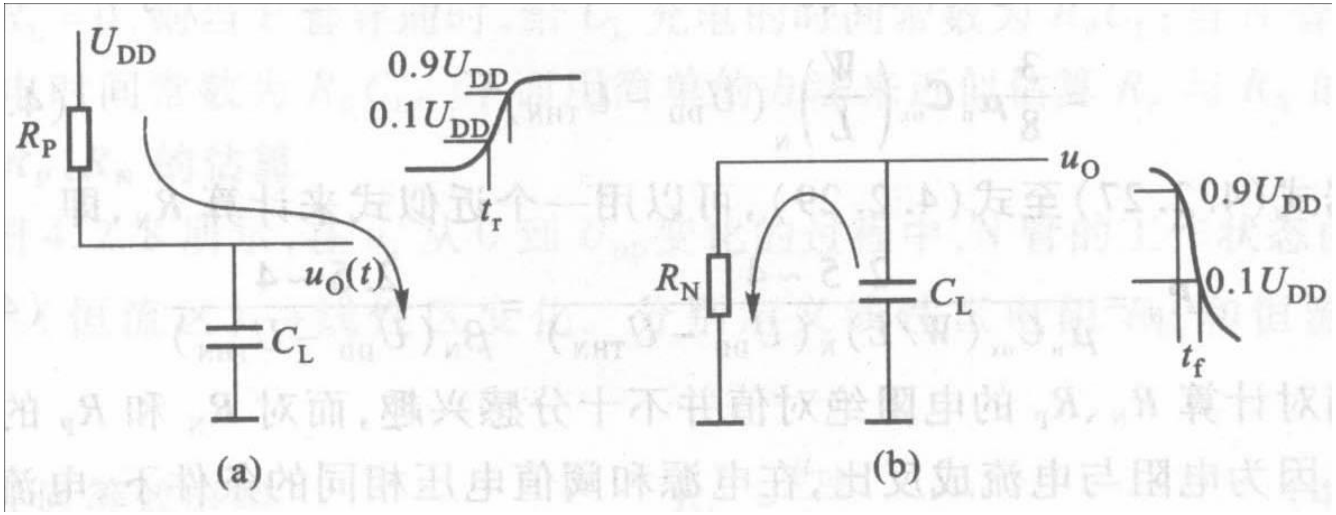
- CMOS反相器上升时间t_r, 下降时间t_f, 延迟时间t_d的计算

  定义

t_r —输出电压u_O从$0.1U_{DD}$上升到$0.9U_{DD}$所需的时间

t_f —输出电压u_O从$0.9U_{DD}$下降到$0.1U_{DD}$所需的时间

t_d —输出电压u_O从0上升到$0.5U_{DD}$所需的时间



CMOS反相器的门延迟及连线延迟

- 上升时间t_r, 下降时间t_f的计算

$$t_r = 2.2R_P C_L$$

$$t_f = 2.2R_N C_L$$

$$\frac{t_r}{t_f} = \frac{R_P}{R_N} = \frac{\mu_n \left(\frac{W}{L}\right)_N}{\mu_p \left(\frac{W}{L}\right)_P}$$

若要求t_r与t_f相等，则需要增大P管的尺寸

$$\left(\frac{W}{L}\right)_P = (2 \sim 4)\left(\frac{W}{L}\right)_N$$

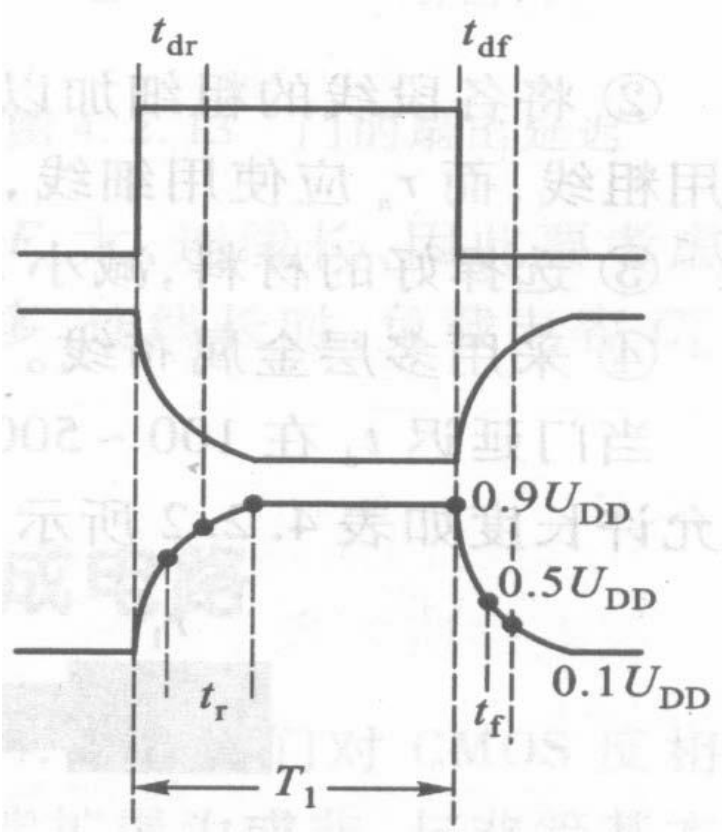
CMOS反相器的门延迟及连线延迟

- CMOS反相器延迟时间t_d的计算

反相器延迟时间分为上升延迟时间t_{dr}和下降延迟时间t_{df}，其平均延迟时间t_d的计算如下

如果输入为理想阶跃波形，那么经过一级非门后的延迟时间为

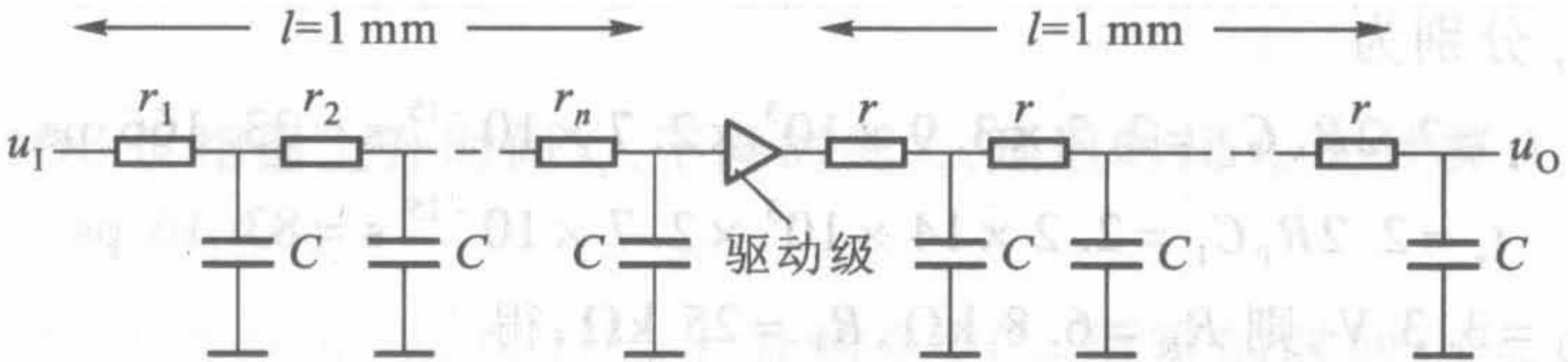
$$t_d = \frac{t_{dr} + t_{df}}{2} = \frac{t_r + t_f}{4}$$



CMOS反相器的门延迟及连线延迟

- 连线延迟

采用多晶硅做连线时，可将其等效为若干段分布RC网络的级联，使信号传输速度下降，产生延迟



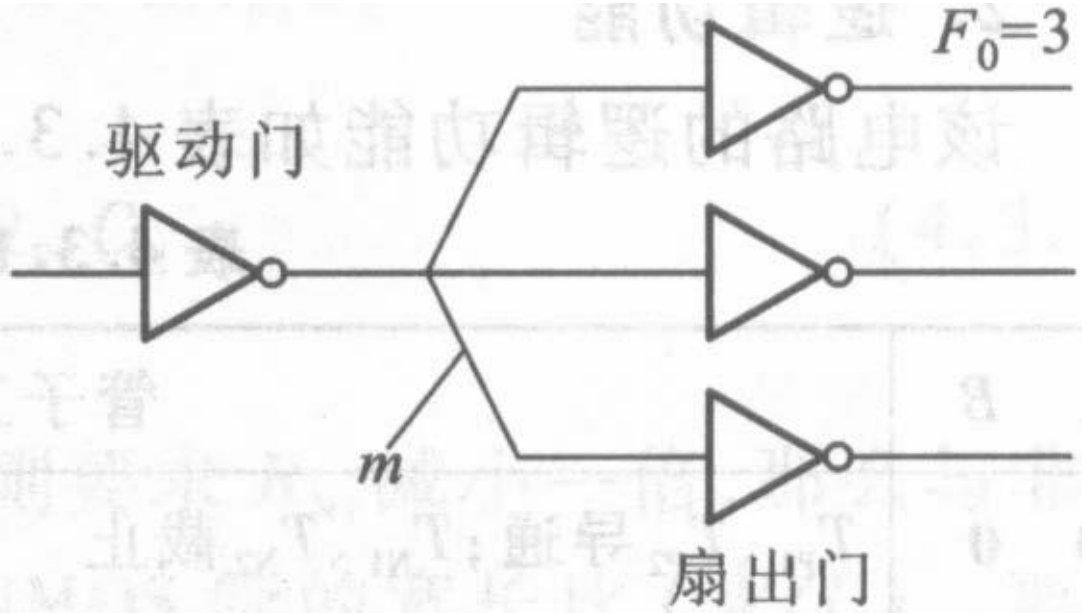
计算此连线延迟的近似公式为

$$t_{dl} \approx \frac{rCl^2}{2}$$

CMOS反相器的门延迟及连线延迟

- 逻辑扇出延迟

若一个反相器不止驱动一个反相器，要同时驱动多个反相器，称之为门的扇出，扇出系数F_0表示被驱动的门数。



延迟时间的估算公式为

$$t_{dF} \approx (m + F_0)t_{dl}$$

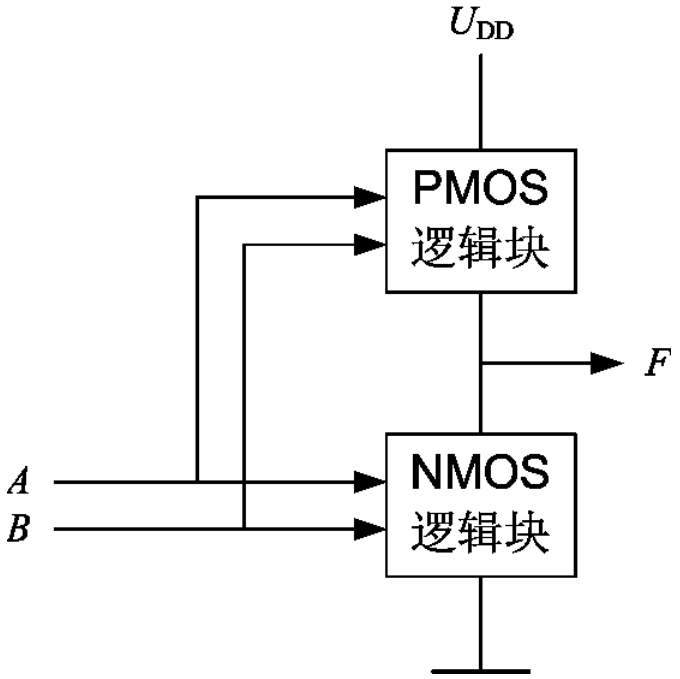
西安電子科技大學

4.3 全互补CMOS集成电路

全互补CMOS集成门电路

通常，CMOS采用正逻辑，由NMOS管组成的逻辑块电路和由PMOS管组成的逻辑块电路分别代替单个NMOS管和单个PMOS管，对于NMOS逻辑遵循“**与串或并**”的规律；对于PMOS管逻辑块，则遵循“**或串与并**”的规律。在这种全互补集成电路中，P管数目和N管数目是相等的。

管子个数=输入变量数×2



CMOS与非门设计

- 电路

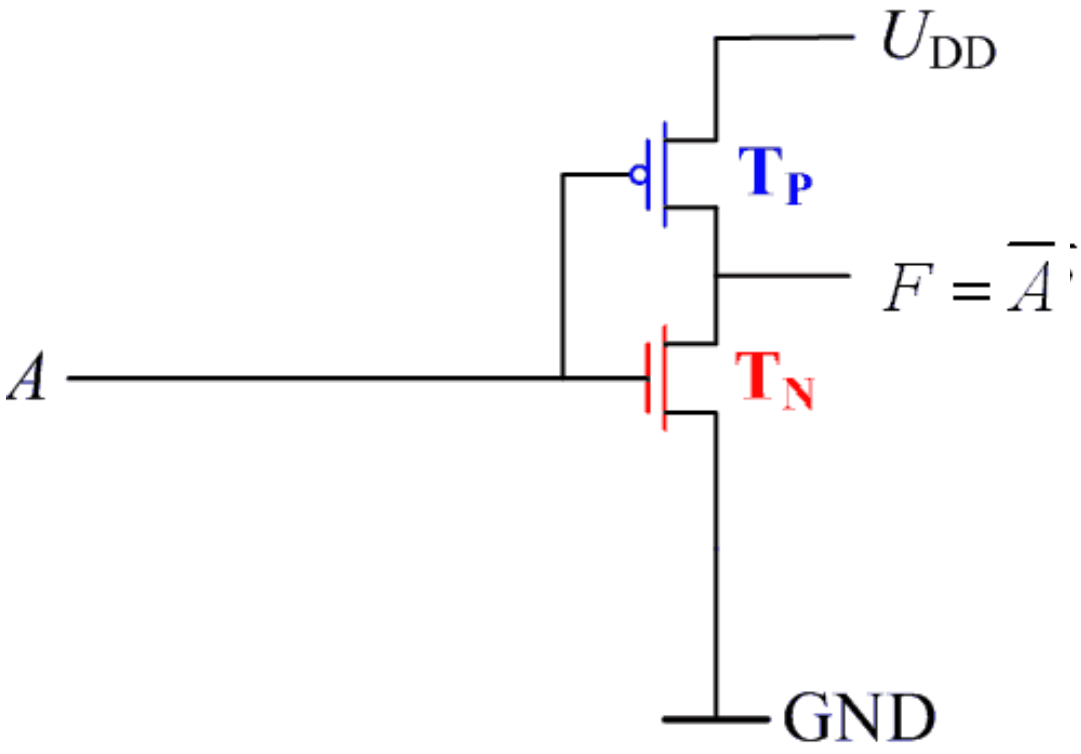
CMOS与非门电路如图所示：

其中NMOS管串联，

PMOS管并联，

A、B为输入变量，

F为输出。



与CMOS反相器的关系？

CMOS与非门设计

- 逻辑功能

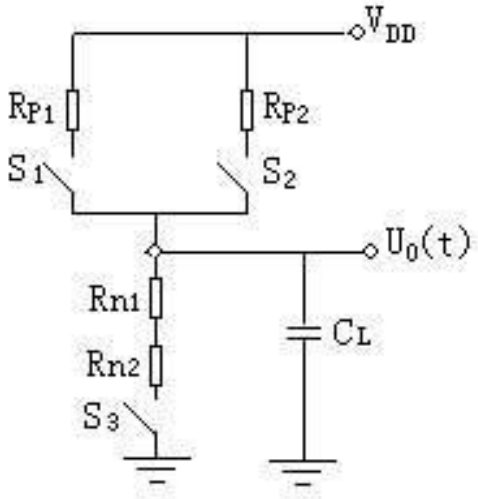
A	B	管子工作状态	$F=\overline{AB}$
0	0	T_{P1}、T_{P2}导通；T_{N1}、T_{N2}截止	1
0	1	T_{P1}导通，T_{P2}截止；T_{N1}截止，T_{N2}导通，但因为串联，故T_{N1}、T_{N2}均截止	1
1	0	T_{P1}截止，T_{P2}导通；T_{N1}、T_{N2}因串联，仍为截止	1
1	1	T_{P1}、T_{P2}截止；T_{N1}、T_{N2}导通	0

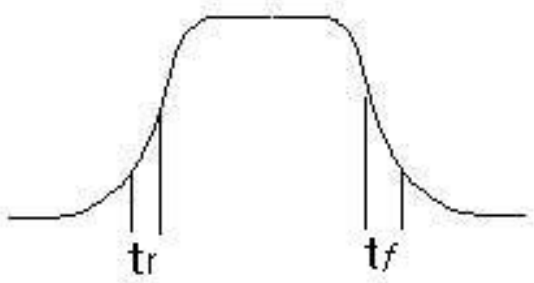
与非门所用的管子数m=输入变量数×2

如何实现三输入与非门？

CMOS与非门设计

- 与非门的RC模型及t_r与t_f的计算





上升时间

$$t_r = 2.2R_{P1}C_L = 2.2R_{P2}C_L$$

下降时间

$$t_f = 2.2\left(R_{N1} + R_{N2}\right)C_L \approx 2.2 \times 2R_{N1}C_L$$

与非门的版图设计

若要求下降时间与标准反相器相同，则

$$\left(\frac{W}{L}\right)_{\mathrm{N}} = 2\left(\frac{W}{L}\right)_{\mathrm{ON}}$$

若要求上升时间与下降时间相同，则$2R_{\mathrm{N1}}=R_{\mathrm{P1}}$，有

$$\left(\frac{W}{L}\right)_{\mathrm{P}} = \frac{\mu_{\mathrm{n}}}{2\mu_{\mathrm{p}}}\left(\frac{W}{L}\right)_{\mathrm{N}} \approx 1.3\left(\frac{W}{L}\right)_{\mathrm{N}}$$

CMOS或非门设计

- 电路

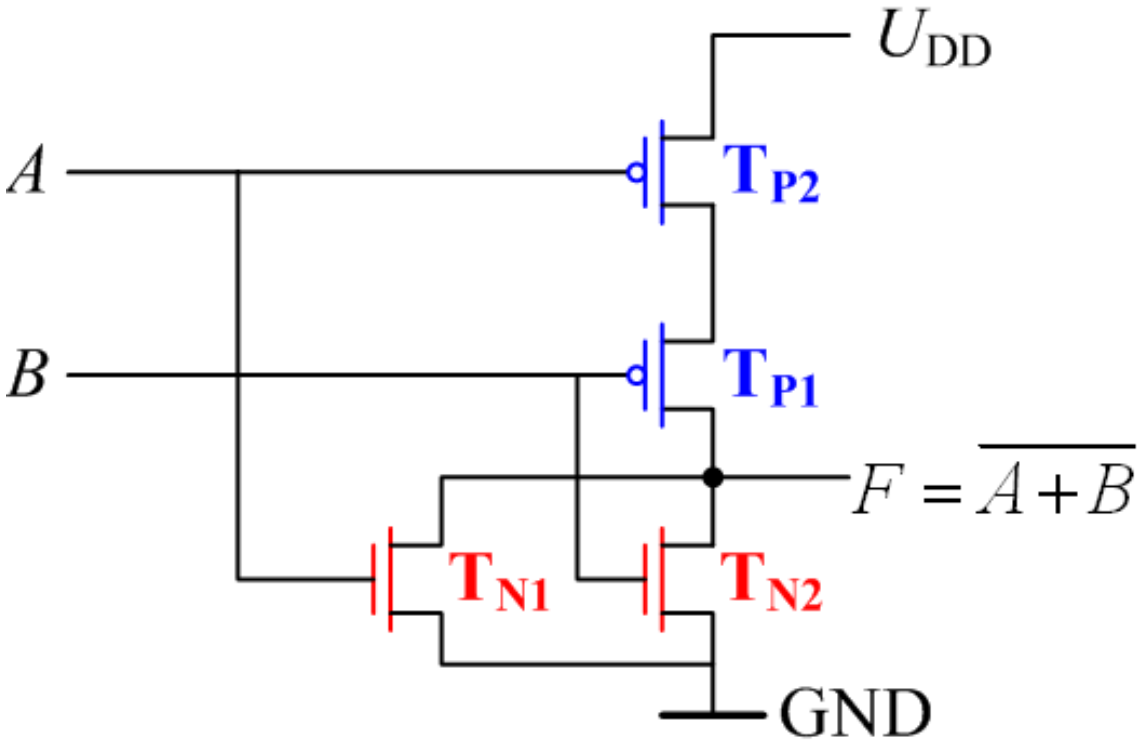
CMOS或非门电路如图所示：

其中NMOS管并联，

PMOS管串联，

A、B为输入变量，

F为输出。



CMOS或非门设计

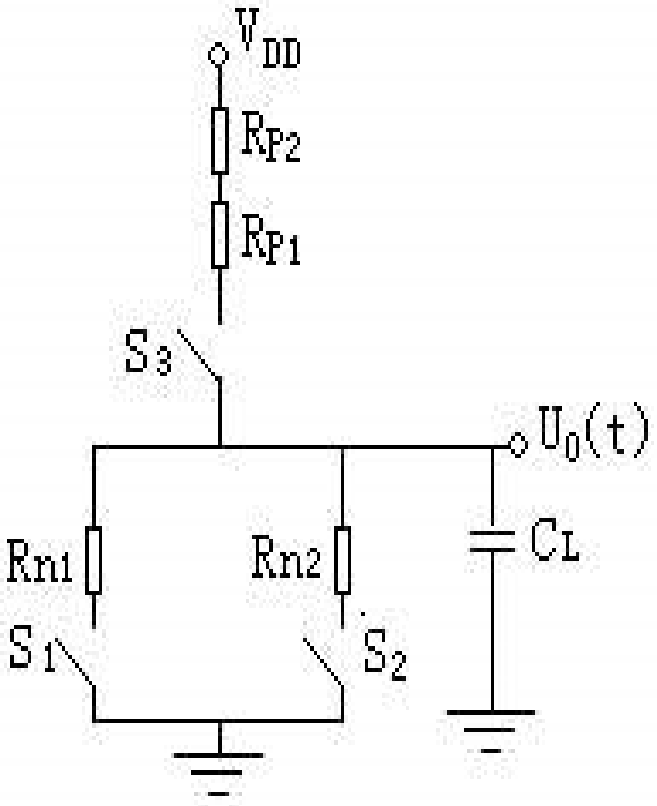
- 逻辑功能

A	B	管子工作状态	$F=\overline{A+B}$
0	0	T_{N1}、T_{N2}截止；T_{P1}、T_{P2}导通	1
0	1	T_{N1}截止，T_{N2}导通；T_{P1}、T_{P2}截止	0
1	0	T_{N1}导通，T_{N2}截止；T_{P1}、T_{P2}截止	0
1	1	T_{N1}、T_{N2}导通；T_{P1}、T_{P2}截止	0

与非门所用的管子数m=输入变量数×2

CMOS或非门设计

- 或非门的RC模型及t_r与t_f的计算



$$t_r = 2.2\left(R_{P1} + R_{P2}\right)C_L = 2.2 \times 2R_{P1}C_L$$

$$t_f = 2.2 \times \frac{R_{N1}}{2}C_L \quad \left(\text{双管导通}\right)$$

$$t_f' = 2.2 \times R_{N1}C_L \quad \left(\text{单管导通，最坏情况}\right)$$

CMOS或非门设计

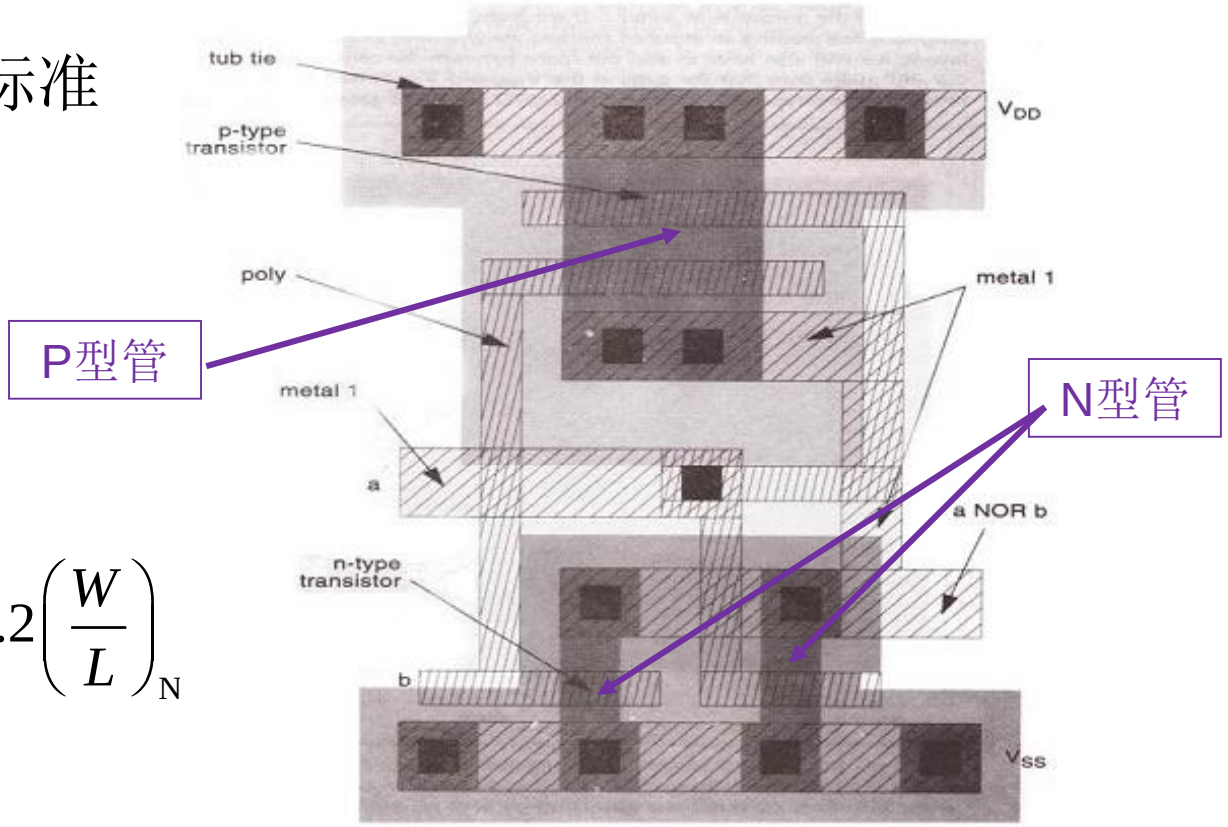
- 或非门的版图设计

若要求驱动能力与标准反相器相同，则

$$2R_{P1} = R_{N1}$$

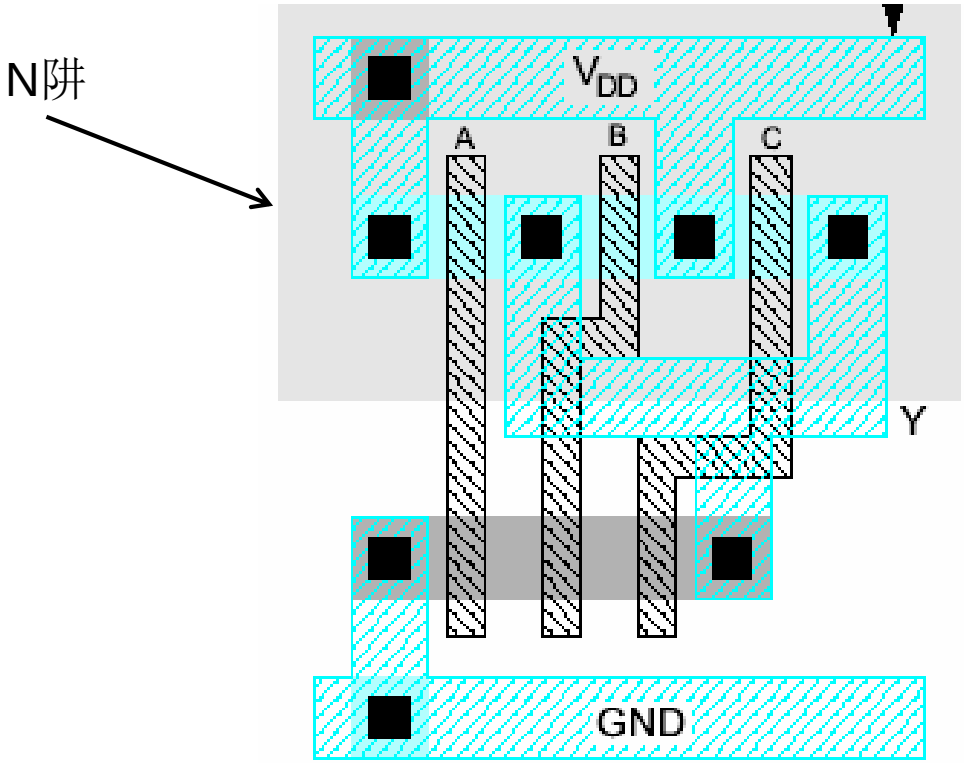
则

$$\left(\frac{W}{L}\right)_P = 2\frac{\mu_n}{\mu_n}\left(\frac{W}{L}\right)_N \approx 5.2\left(\frac{W}{L}\right)_N$$



CMOS与非门设计

[练习] 分析以下某电路的版图，给出晶体管级电路图和逻辑表达式。



CMOS与或非门设计

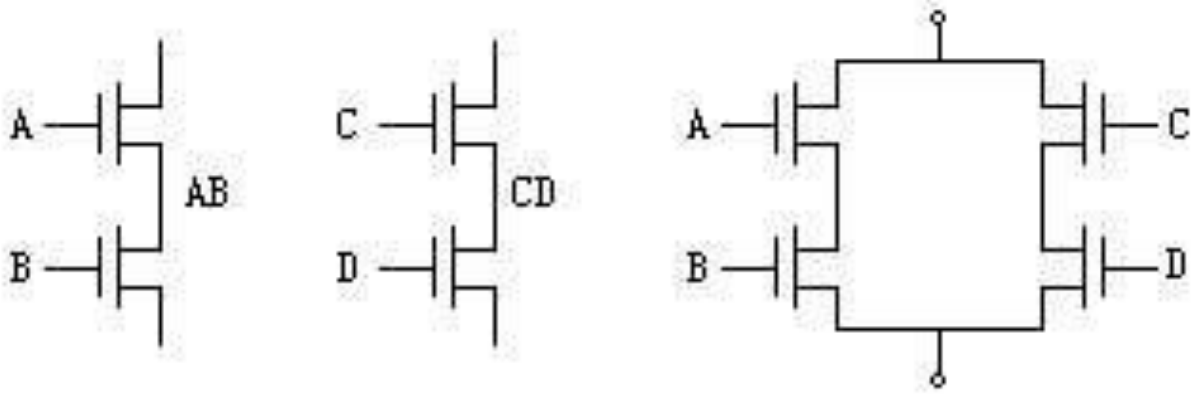
- 实现的逻辑函数

$$F = \overline{AB + CD}$$

- 电路

NMOS逻辑块电路的设计

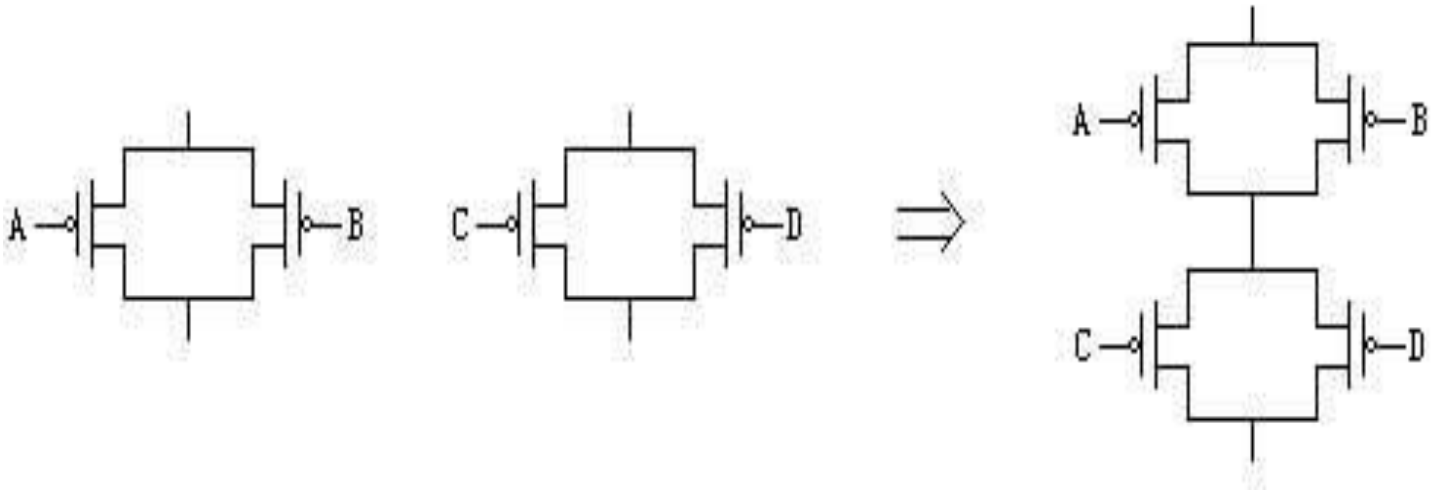
根据NMOS逻辑块与串或并的规律构成N逻辑块的电路。



CMOS与或非门设计

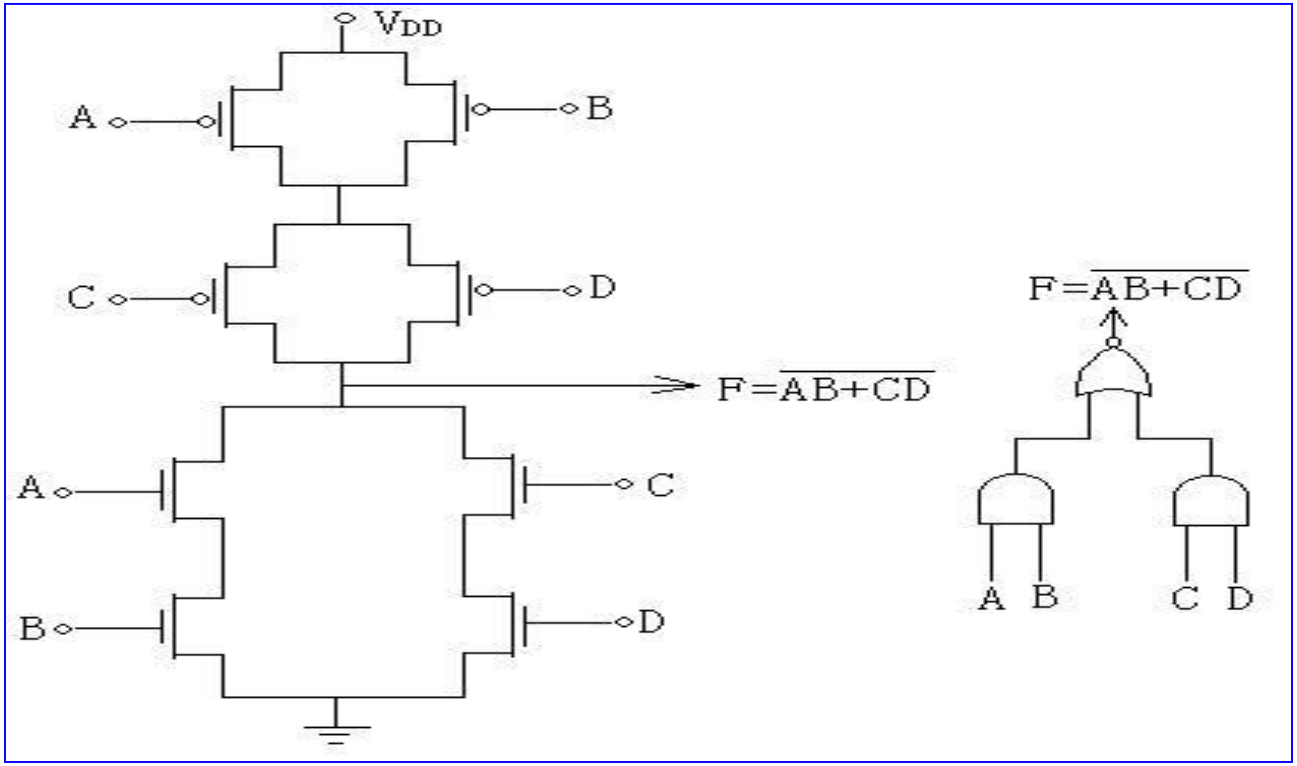
PMOS逻辑块电路的设计

根据PMOS逻辑或串与并的规律构成PMOS逻辑块电路。



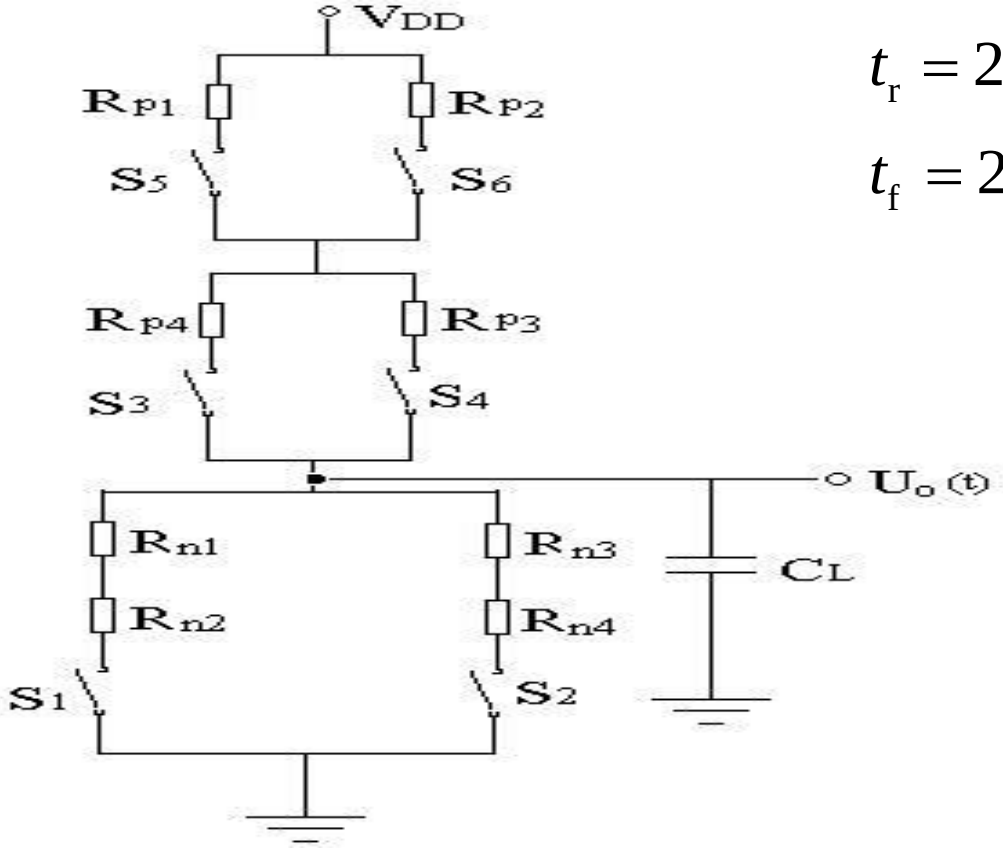
CMOS与或非门设计

将NMOS逻辑块与PMOS逻辑块连接，接上电源和地，构成完整的逻辑电路。



CMOS与或非门设计

- RC模型及管子尺寸设计



$$t_r = 2.2\left(R_{P1} + R_{P3}\right)C_L = 2.2\times 2R_{P1}C_L$$

$$t_f = 2.2\left(R_{N1} + R_{N2}\right)C_L = 2.2\times 2R_{N1}C_L$$

若要求C_L充、放电的驱动能力一致，则

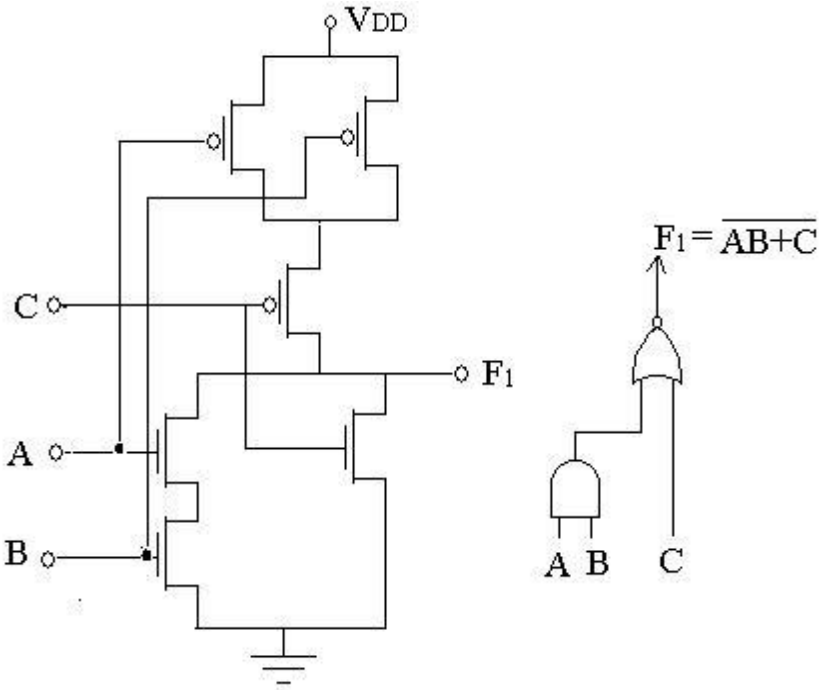
$$R_{P1} = R_{N1}$$

有

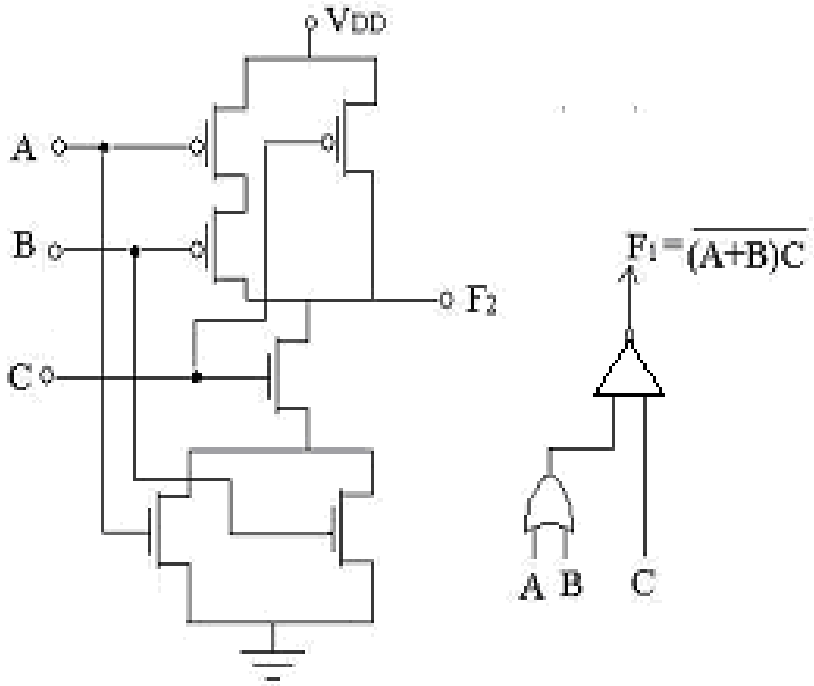
$$\left(\frac{W}{L}\right)_P = \frac{\mu_n}{\mu_p}\left(\frac{W}{L}\right)_N = 2.6\left(\frac{W}{L}\right)_N$$

CMOS与或非门设计

- 另一种与或非门和或与非门电路

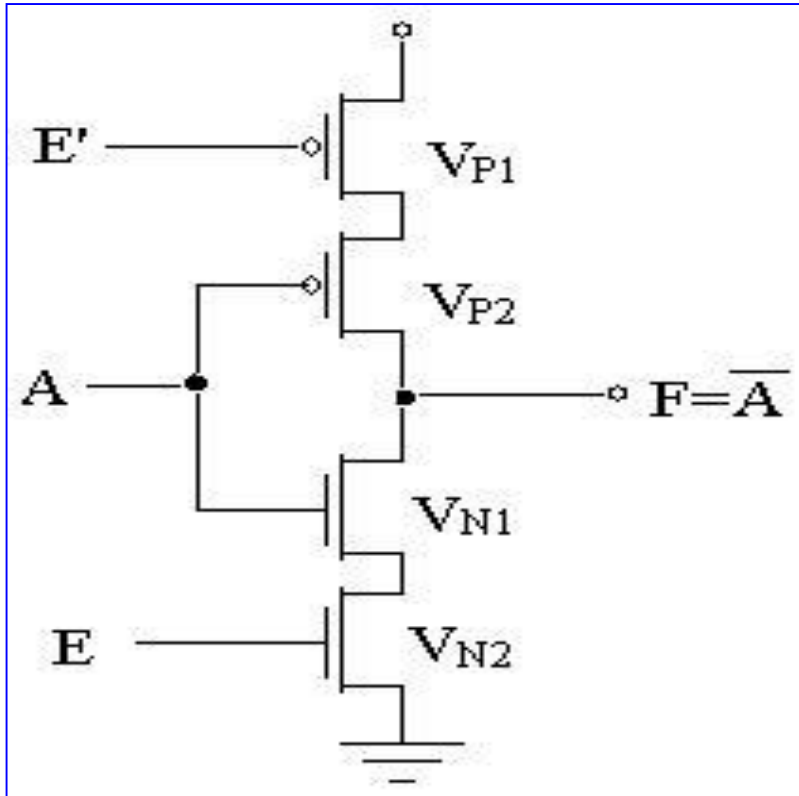


$$F = \overline{AB + C}$$



$$F = \overline{(A + B)C}$$

CMOS三态门和钟控CMOS逻辑电路



(a)

E	A	F
1	0	1
1	1	0
0	0	高阻
0	1	高阻

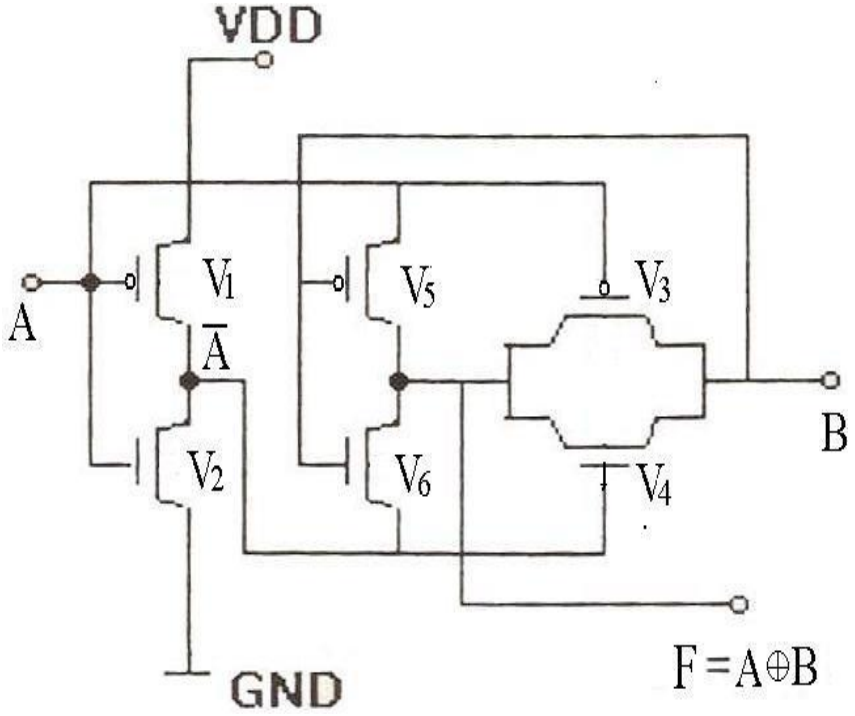
(b)

CMOS异或门设计

- 逻辑函数

$$F = A \oplus B = A\overline{B} + \overline{A}B$$

- 电路与逻辑关系



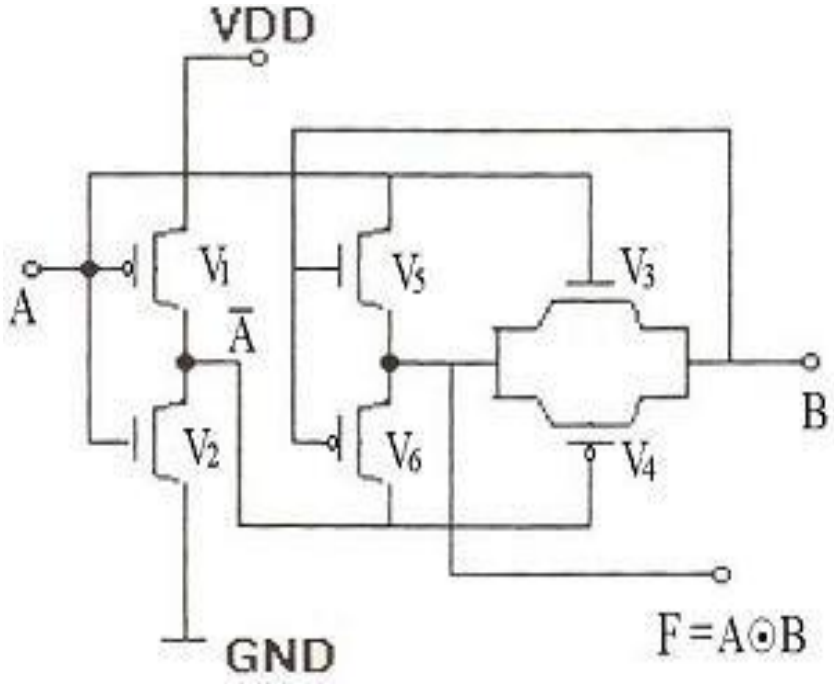
A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

CMOS同或门设计

- 逻辑函数

$$F = A \odot B = \overline{A \oplus B} = \overline{AB} + AB$$

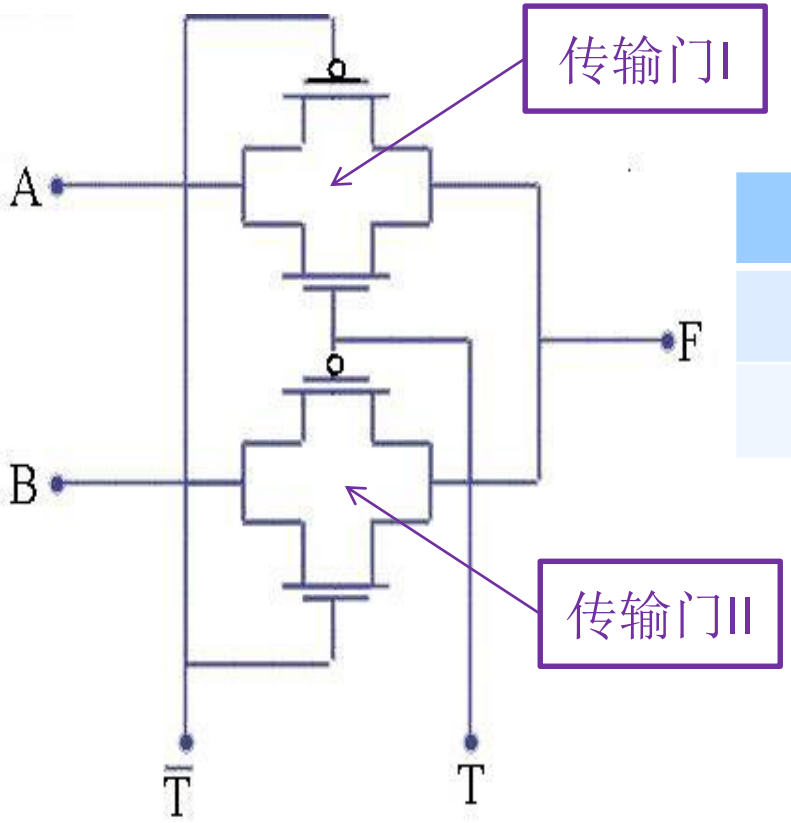
- 电路与逻辑关系



A	B	F=A⊙B
0	0	1
0	1	0
1	0	0
1	1	1

CMOS数据选择器

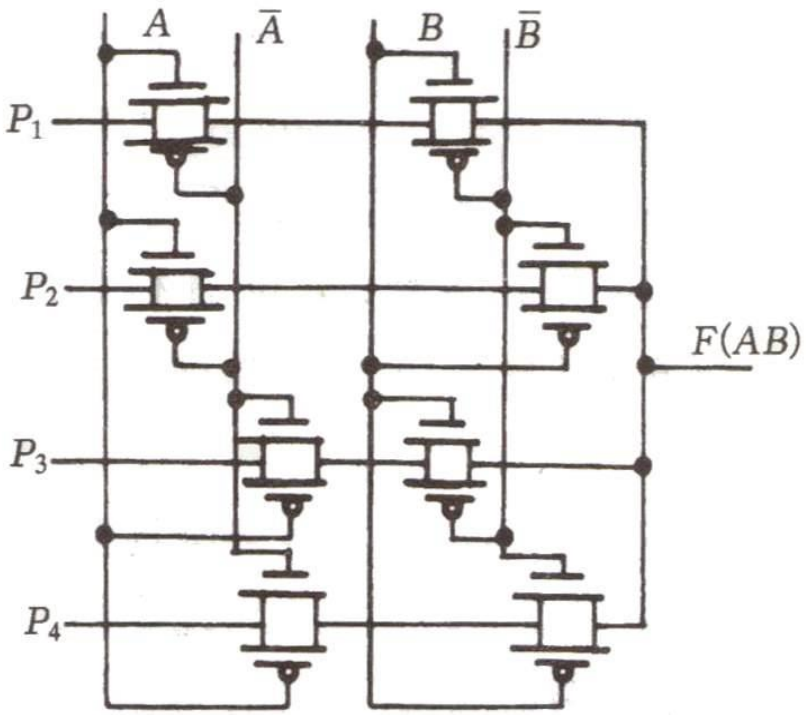
- 电路与逻辑功能



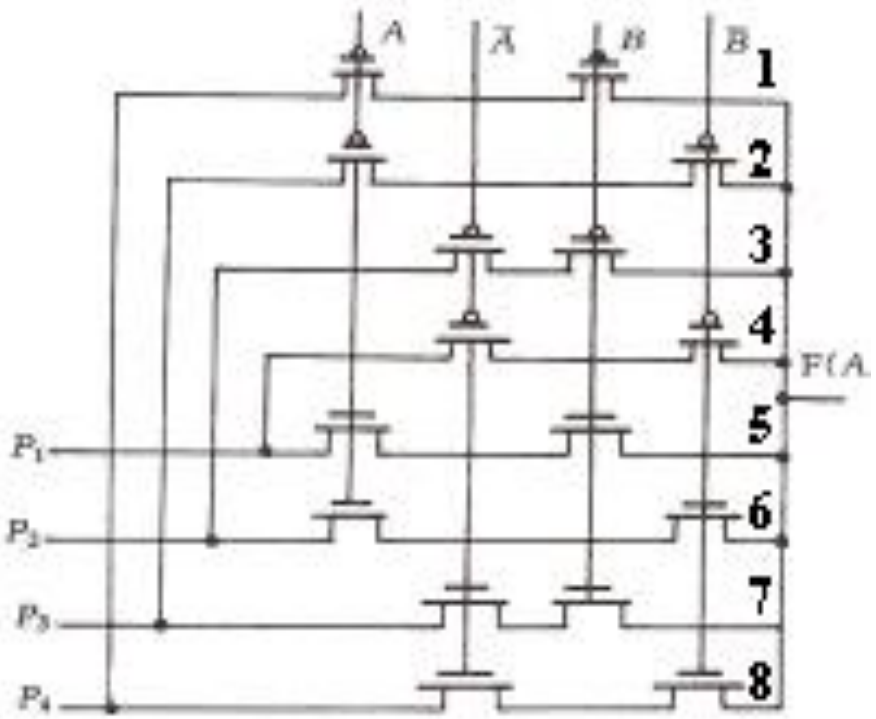
T	$\overline{T}$	工作状态	输出F
0	1	I截止，II导通	B
1	0	I导通，II截止	A

布尔函数逻辑—传输门的又一应用

- 电路



全传输门型



改进版图的CMOS型

布尔函数逻辑—传输门的又一应用

- 布尔函数卡诺图

B \ A	0	1
0	P_4	P_2
1	P_3	P_1

- 布尔函数逻辑电路的逻辑功能

	P_4	P_3	P_2	P_1	实现的操作
控制条件	0	0	0	1	AND(A, B)
	0	1	1	0	XOR(A, B)
	0	1	1	1	OR(A, B)
	1	0	0	0	NOR(A, B)
	1	1	1	0	NAND(A, B)

布尔函数逻辑—传输门的又一应用

- 该电路实现4选1数据选择器功能

A	B	$\overline{A}$	$\overline{B}$	导通的行号	F(输出)
0	0	1	1	第1行，第8行	P_4
0	1	1	0	第2行，第7行	P_3
1	0	0	1	第3行，第6行	P_2
1	1	0	0	第4行，第5行	P_1

CMOS全加器

- 逻辑关系

$$S = A \oplus B \oplus C_i$$

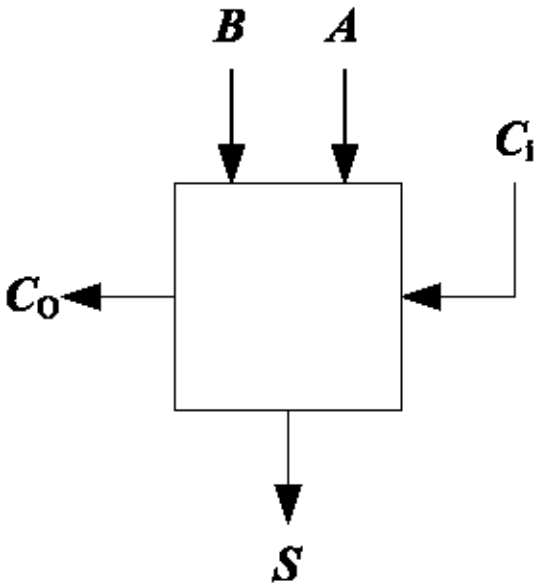
$$C_o = AC_i + BC_i + AB = (A+B)C_i + AB$$

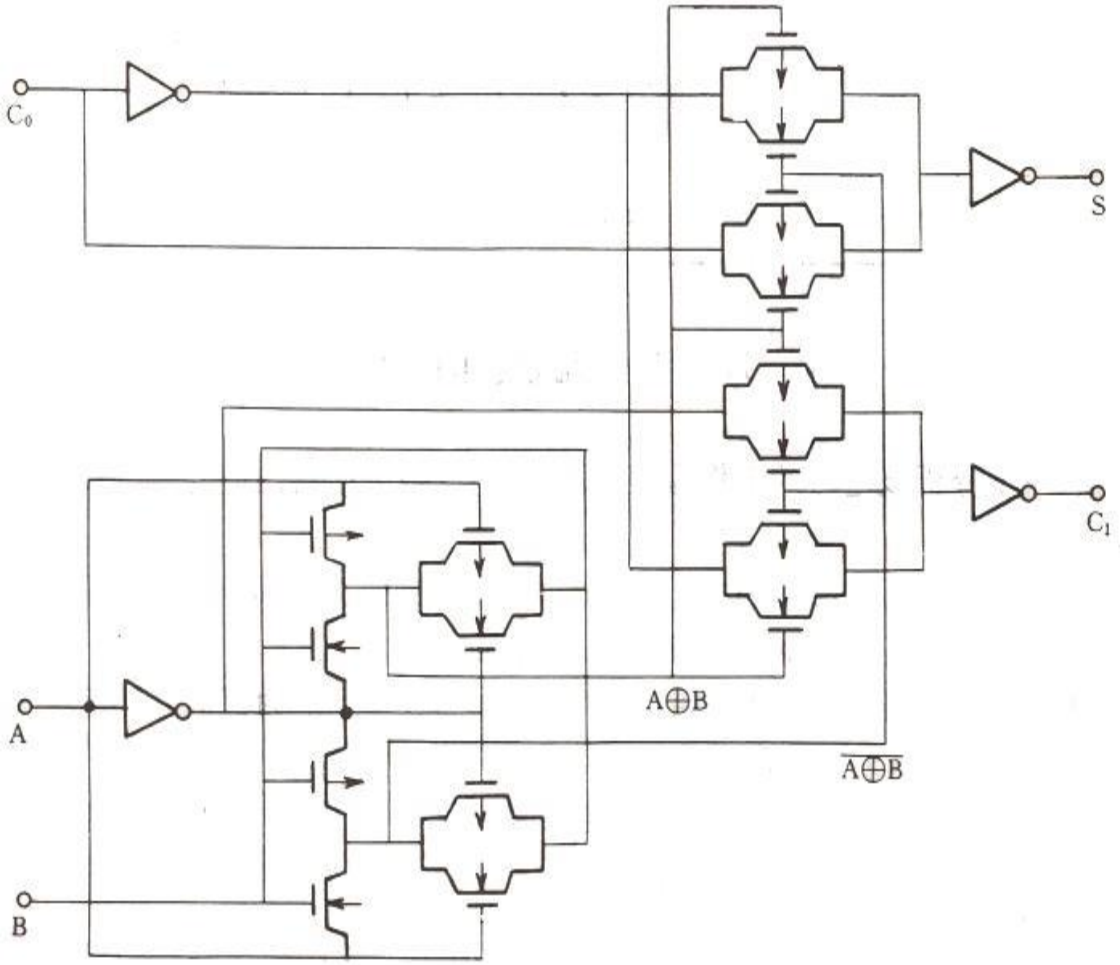
- 全加器真值表

A	B	Ci	S	Co	A	B	Ci	S	Co
0	0	0	0	0	1	0	0	1	0
0	0	1	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1
0	1	1	0	1	1	1	1	1	0

CMOS全加器

- 电路





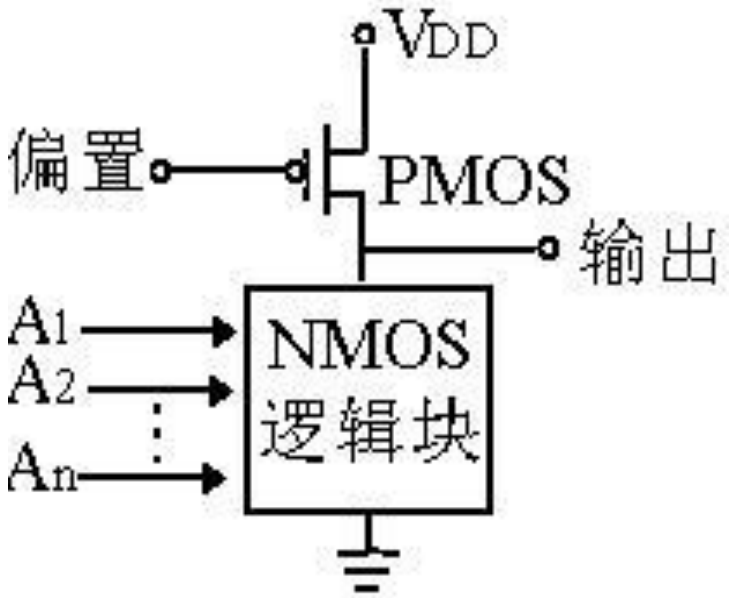
4.4 改进的CMOS逻辑电路

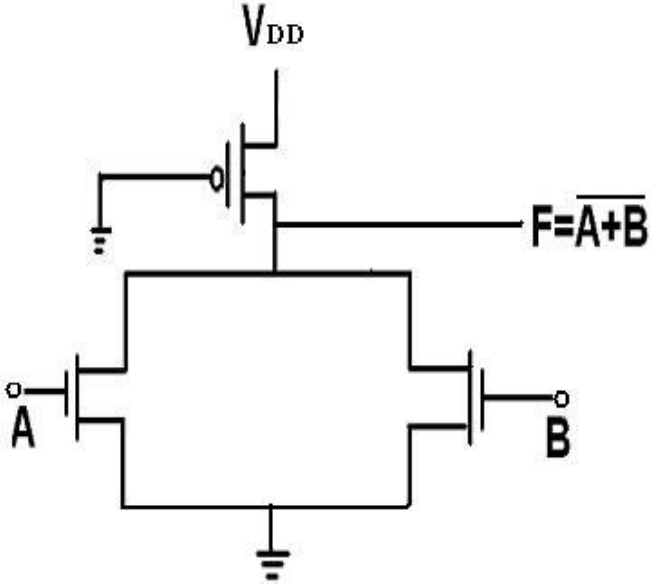
全互补CMOS电路中，NMOS管和PMOS管数目相等，但**逻辑块功能有时并不对等，导致输入电容加倍，影响速度**。为此在保证功能不变的情况减少PMOS管数目，从而节省硅片面积，减少功耗，提高运行速度。

伪NMOS逻辑电路

- 伪NMOS 2输入或非门

整个逻辑电路由一个NMOS逻辑块和一个作为负载的PMOS管构成，所用管子数为： 输入变量个数+1。



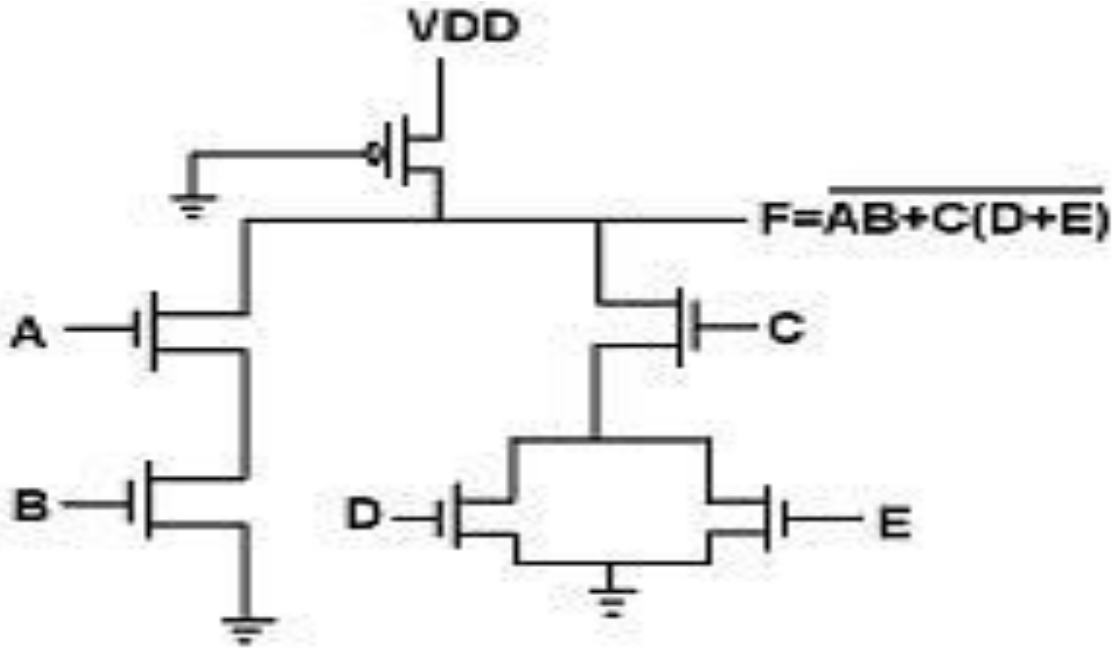


伪NMOS逻辑电路

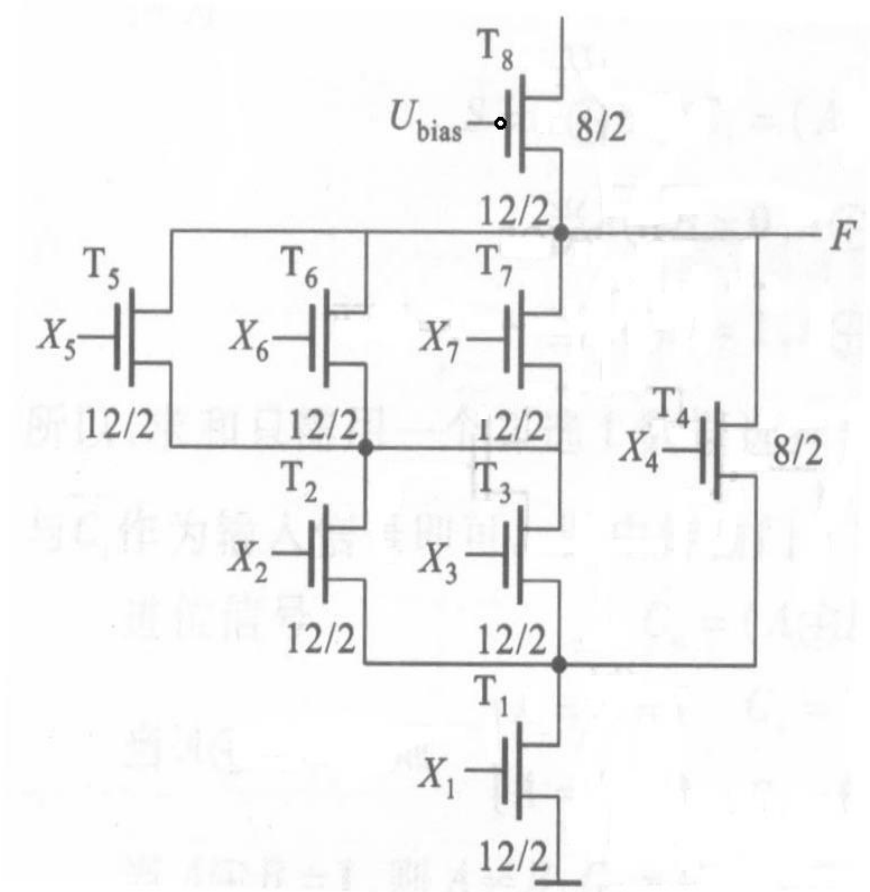
- 伪NMOS逻辑电路

例：如图所示，该电路的N逻辑块由5个管子组成，而PMOS管只有一个，实现的函数关系：

$$F = \overline{AB + C(D + E)}$$

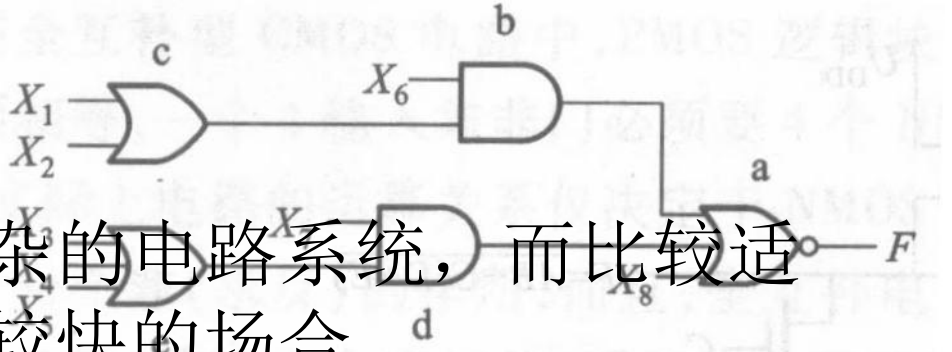


伪NMOS逻辑电路



1～例4.4.3

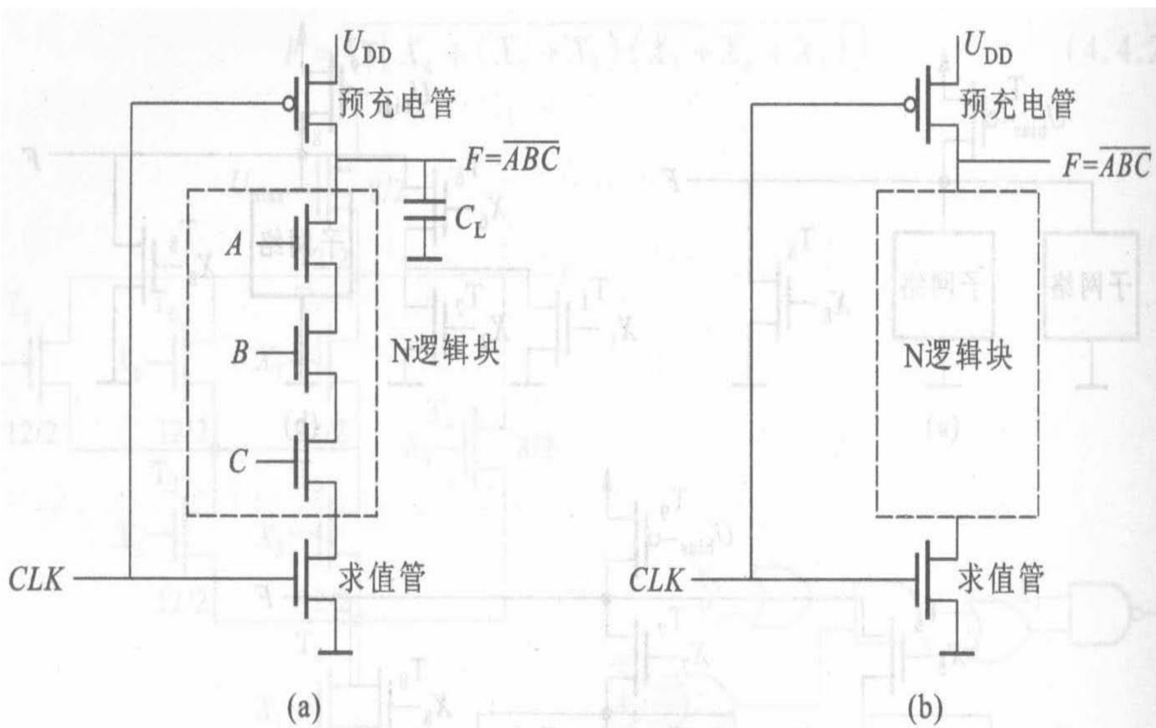
$$F=\overline{\left(X_1+X_2\right)X_6+\left(X_3+X_4+X_5\right)X_7+X_8}$$



杂的电路系统，而比较适
较快的场合。

动态CMOS逻辑电路（预充电CMOS电路）

- 电路——针对伪NMOS静态功耗大的问题，人们提出了一种动态CMOS电路。这种电路用的管子数比全互补型CMOS少，静态功耗也比NMOS电路小。



当CLK为"0"时，NMOS管截止，PMOS管导通，输出为"1"，与N逻辑块的输入变量无关，输出始终为"1"，此阶段称为预充电阶段。

当时钟CLK为"1"时，PMOS预充电管截止，NMOS求值管导通，输出结点是否放电与N逻辑块有关，所以称为求值阶段。若A=B=C=1，N逻辑块导通，F放电至0。

A、B、C中任一为零，N逻辑块不导通，F不放电，F始终为1。

动态CMOS逻辑电路（预充电CMOS电路）

- 动态CMOS电路的特点

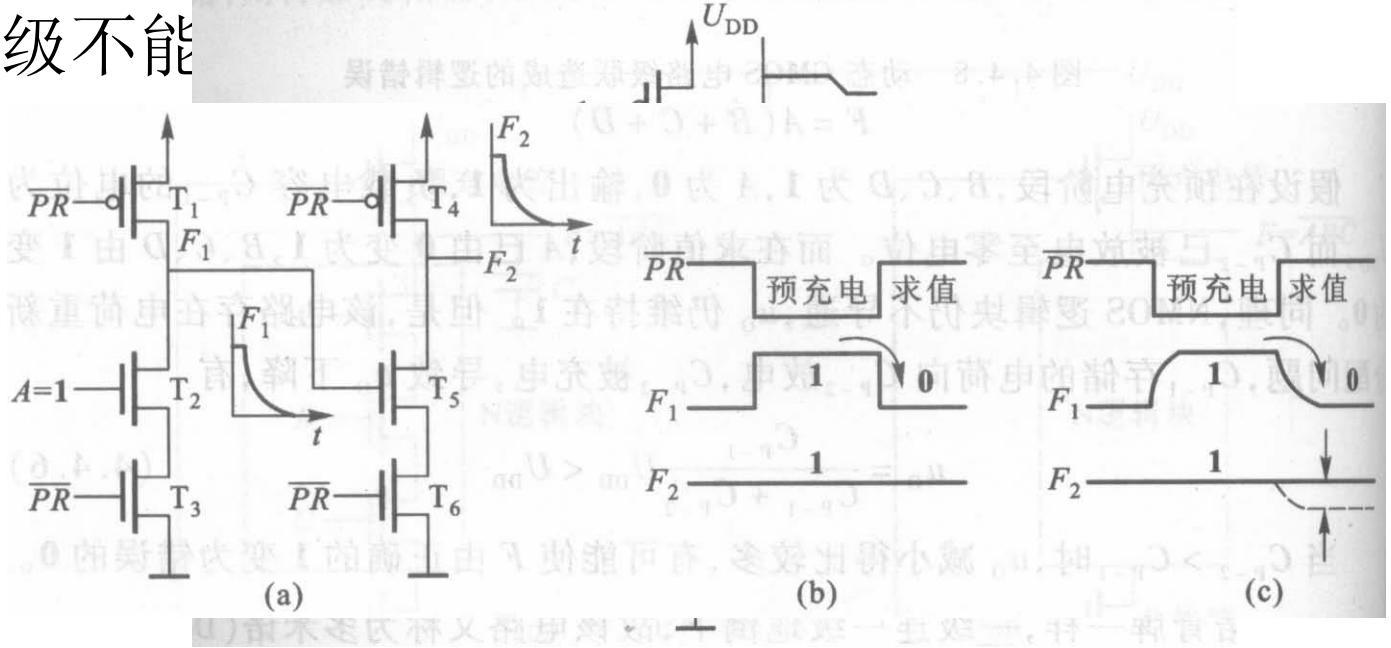
1. 由于求值管和预充电管交替导通和截止，所以这种电路是无比电路，保证了静态功耗为0。
2. 所用管子总数=输入变量数+2，比全互补CMOS电路少得多，比伪NMOS电路仅多出一个。
3. 每个输入只接一个NMOS管，输入电容比全互补CMOS电路少一倍。

- 动态CMOS电路减少了元件数，功耗低，提高了集成度和工作速度。

动态CMOS逻辑电路（预充电CMOS电路）

- 动态CMOS电路存在的问题

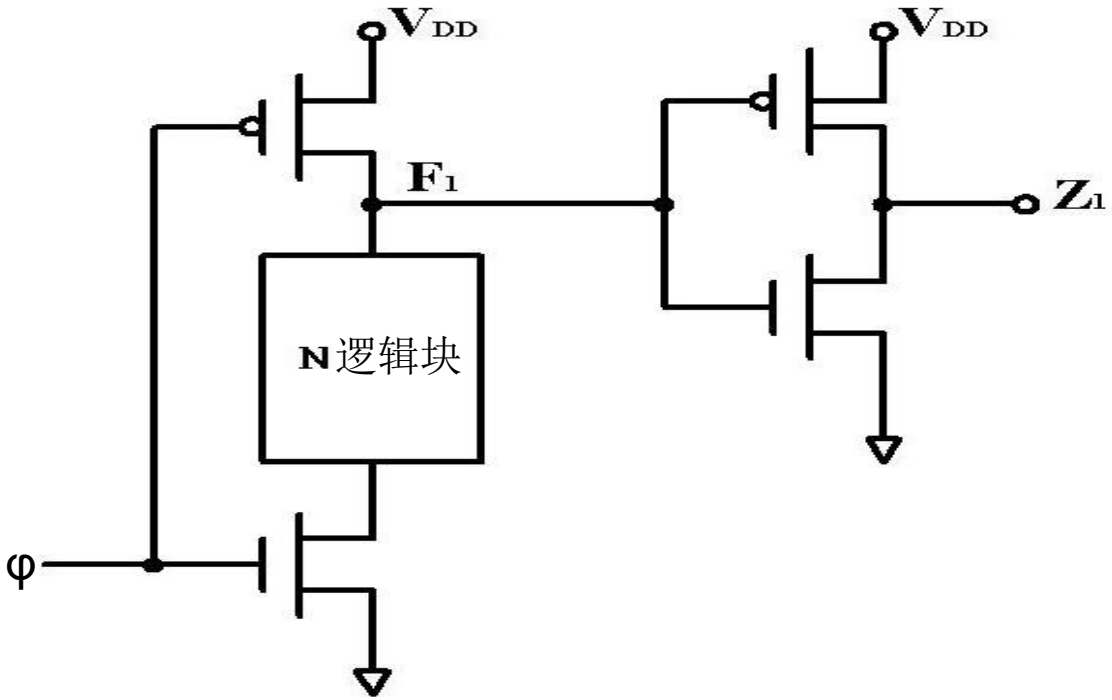
1. 输入变量只能在预充电期间变化，在求值阶段必须保持稳定。
2. 因为有分布电容的存在，产生了电荷再分配问题，而使输出高电平下降，容易造成逻辑混乱和错误。
3. 多级不能



多米诺逻辑

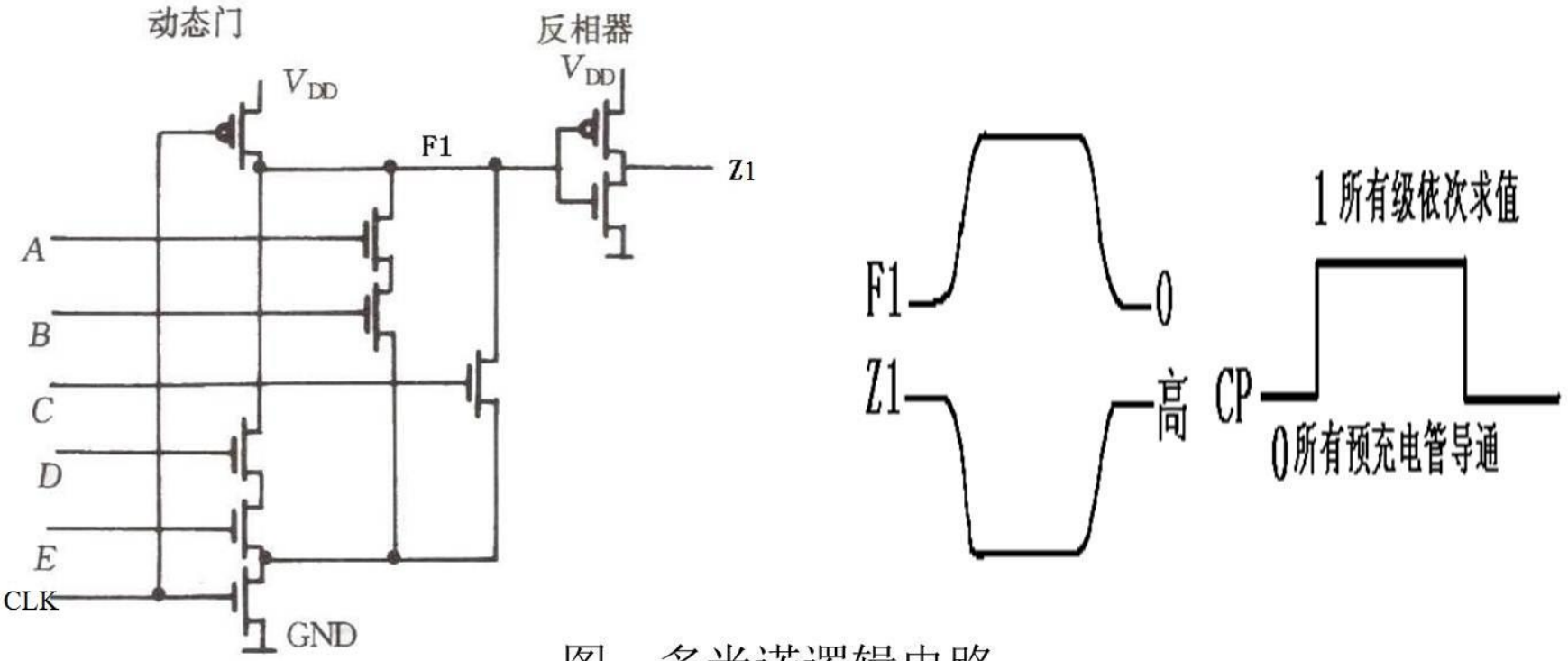
- 多米诺逻辑电路—加反相器隔离

加反相器隔离为了克服普通动态CMOS电路不能直接级联的问题，可以在第一级的输出和第二级的输入之间插入一级反相器作缓冲，将两级隔离开。



多米诺逻辑

- 多米诺逻辑电路

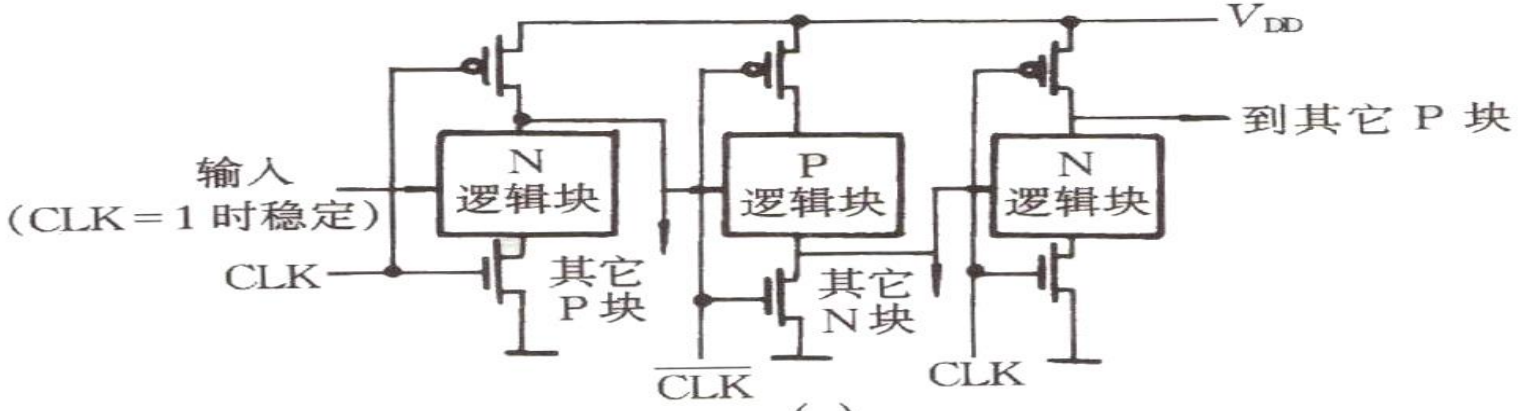


图：多米诺逻辑电路

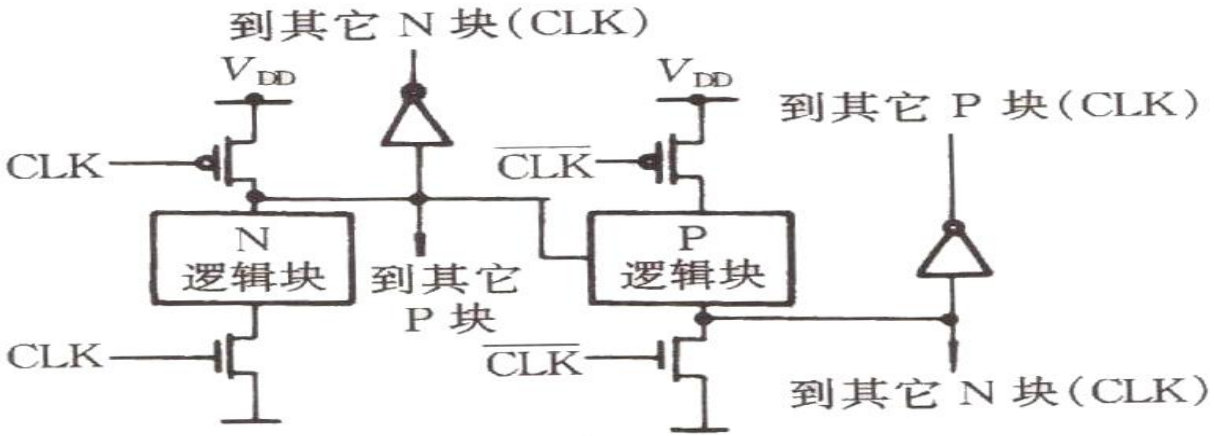
（左）电路　　（右）波形

N－P逻辑块交替的多米诺逻辑

新的多米诺电路，将NMOS逻辑块电路PMOS逻辑块电路交替级联，省去了反相器，又保证逻辑关系不混乱。



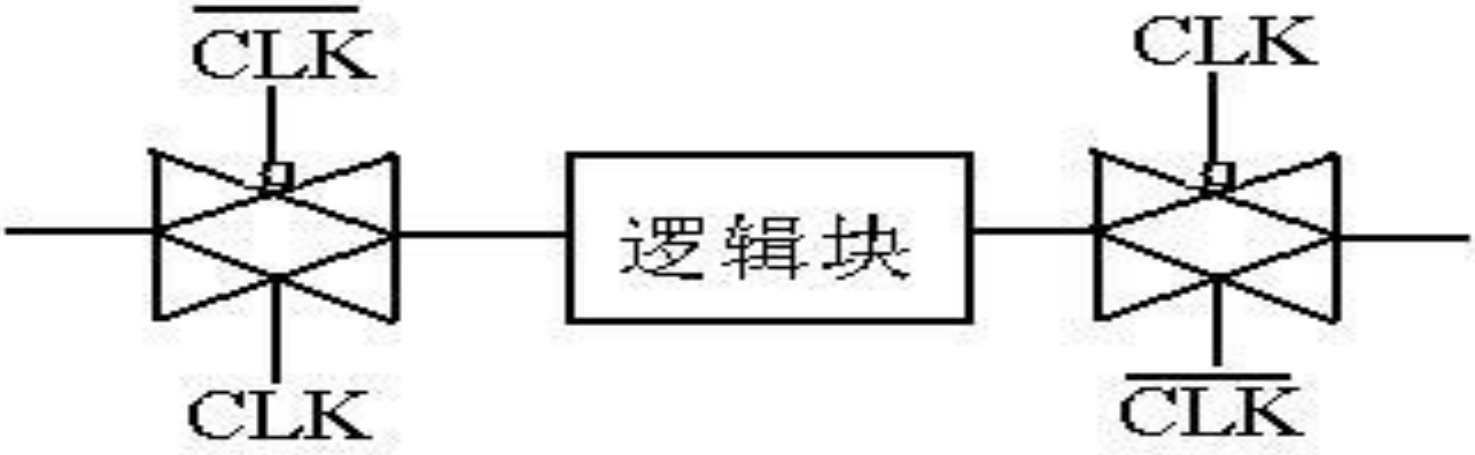
(a)



(b)

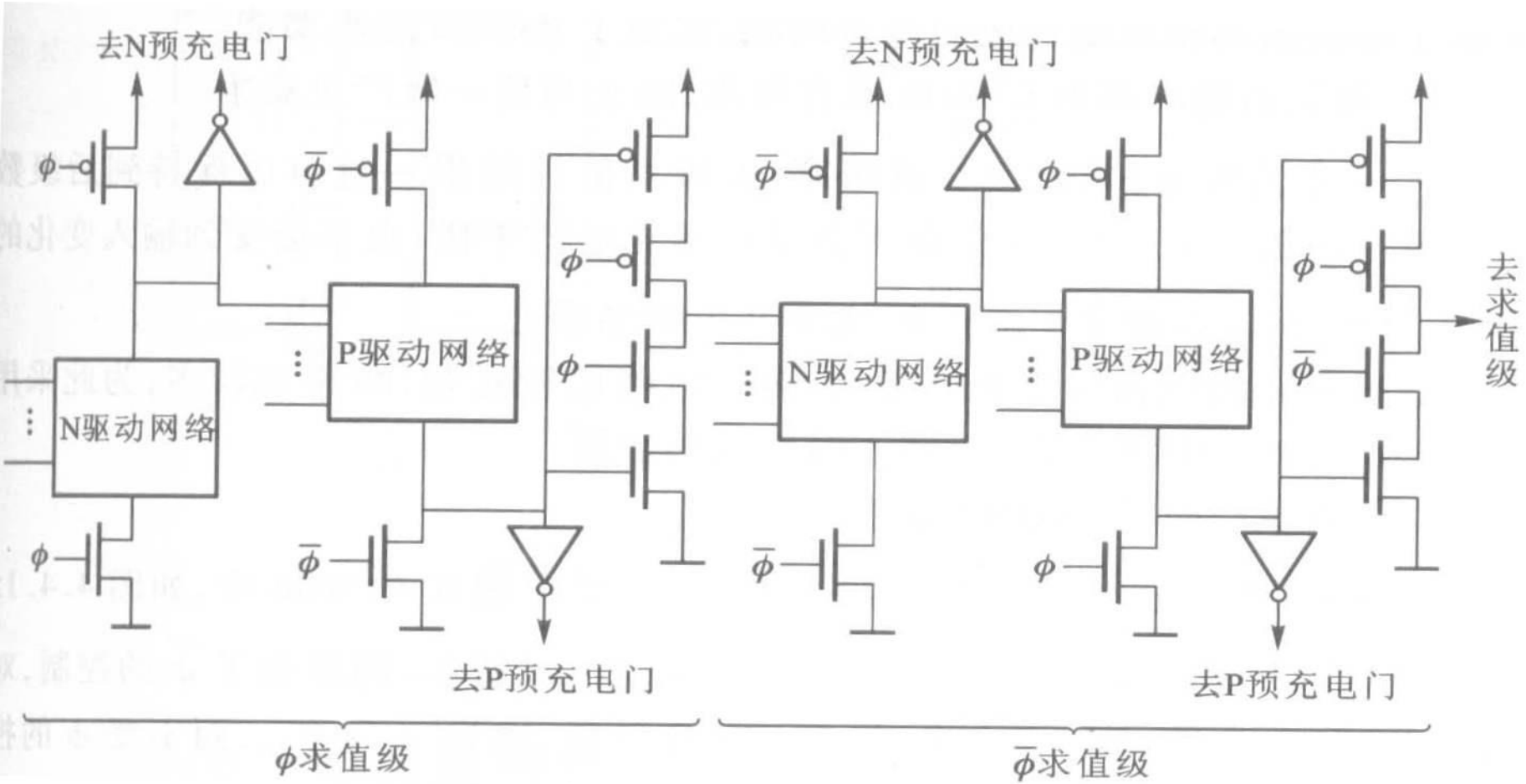
流水线逻辑和无竞争技术

流水线作业使系统的运行速度有了很大的提高。在流水线逻辑中，数据是沿着流水线顺序逐步加工的，各级之间往往用传输门隔离，如图所示是流水线中最基本的一节。



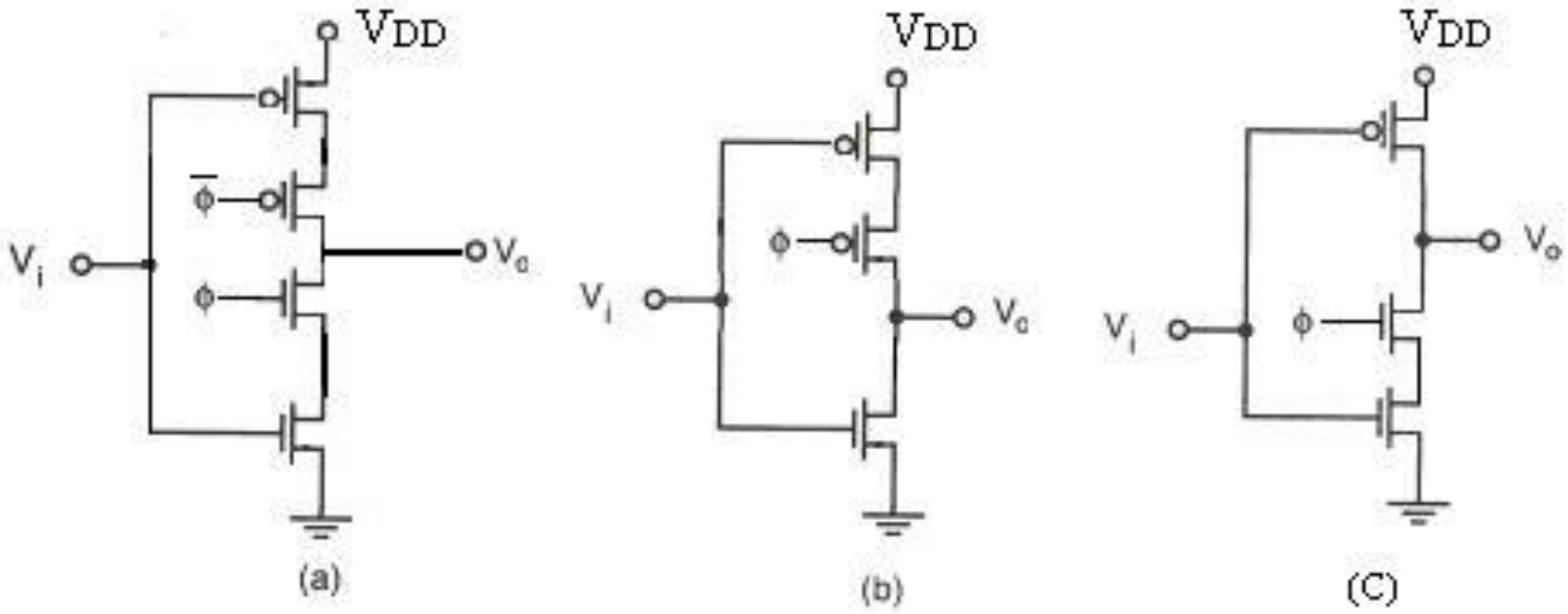
无竞争逻辑(NORA, No Race Logic)

无竞争逻辑中的一级由三部分组成：N型动态CMOS电路(简称N段)，P型动态CMOS电路(简称P段)和C²MOS电路。



“真单相时钟”NORA逻辑

由于无竞争逻辑电路的倒相器必须是偶数，而且电路比较复杂，时钟线较多，为此采用单相时钟”NORA”电路，以减少时钟线的数量。



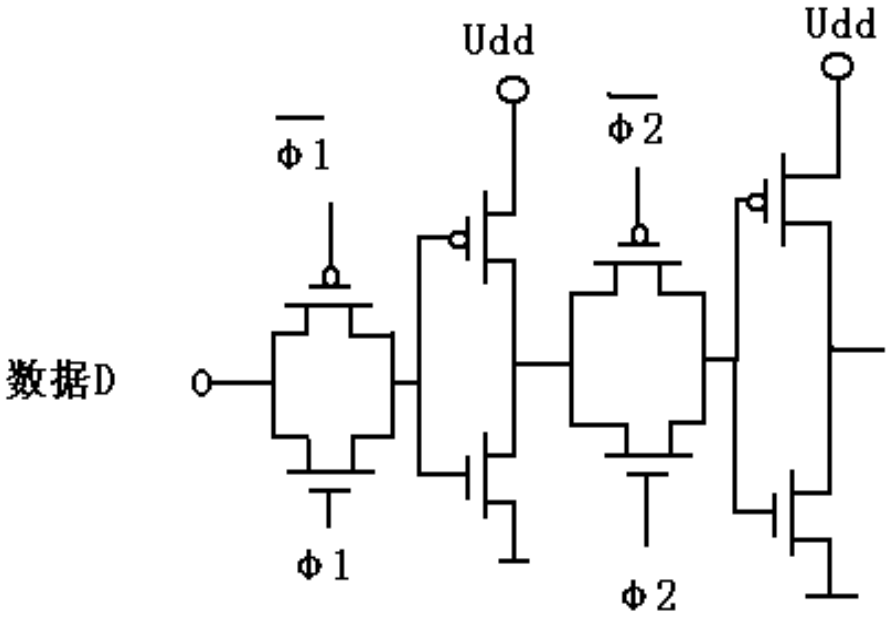
西安电子科技大学

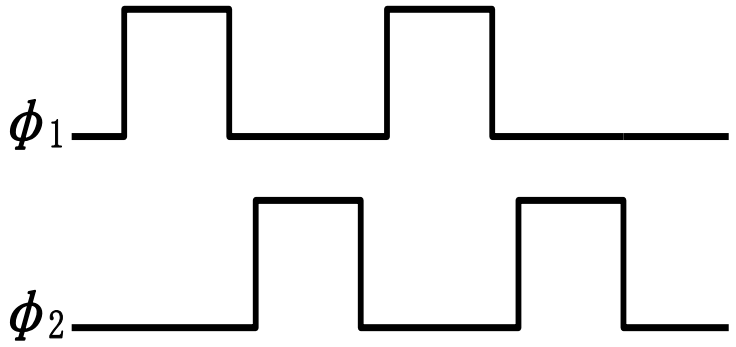
4.5 移位寄存器、锁存器、触发器、I/O单元

移位寄存器(Shift Register)

- 动态CMOS移位寄存器

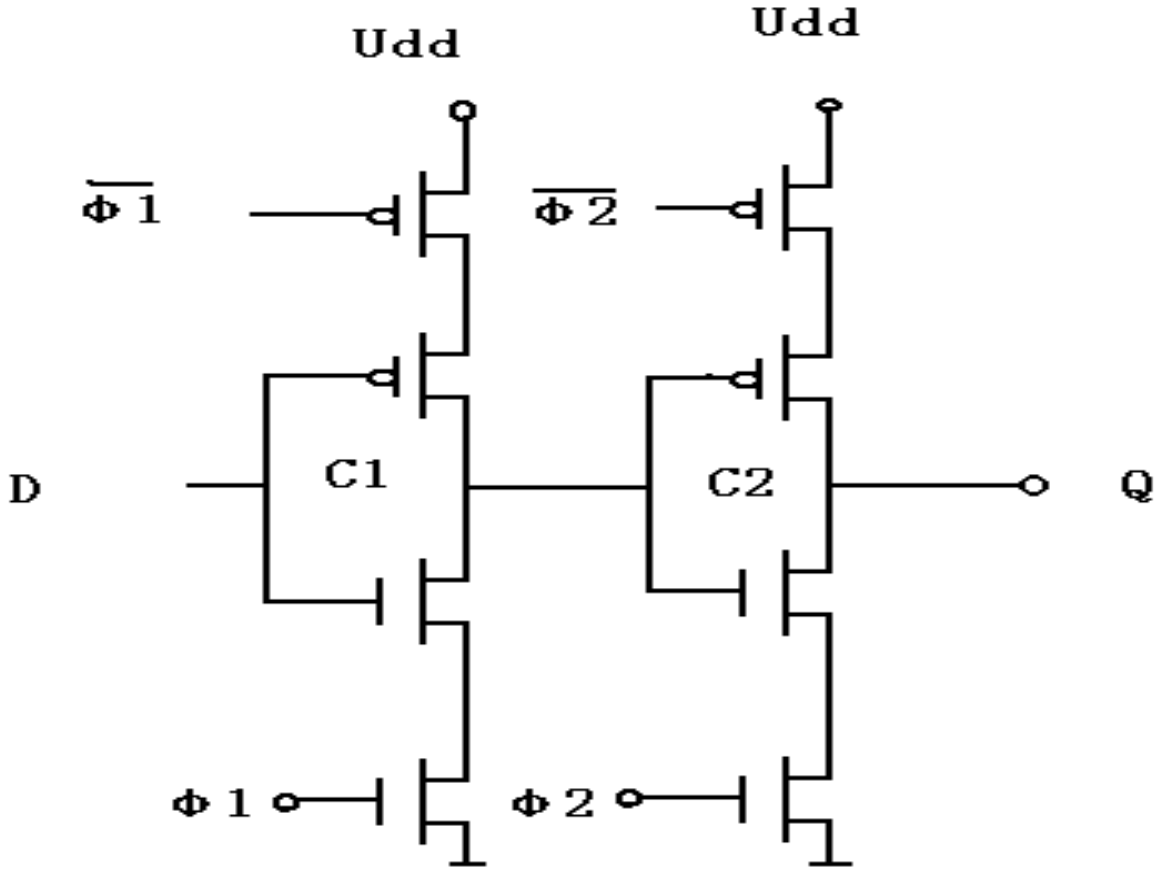
  动态CMOS移位寄存器及时钟波形如下：





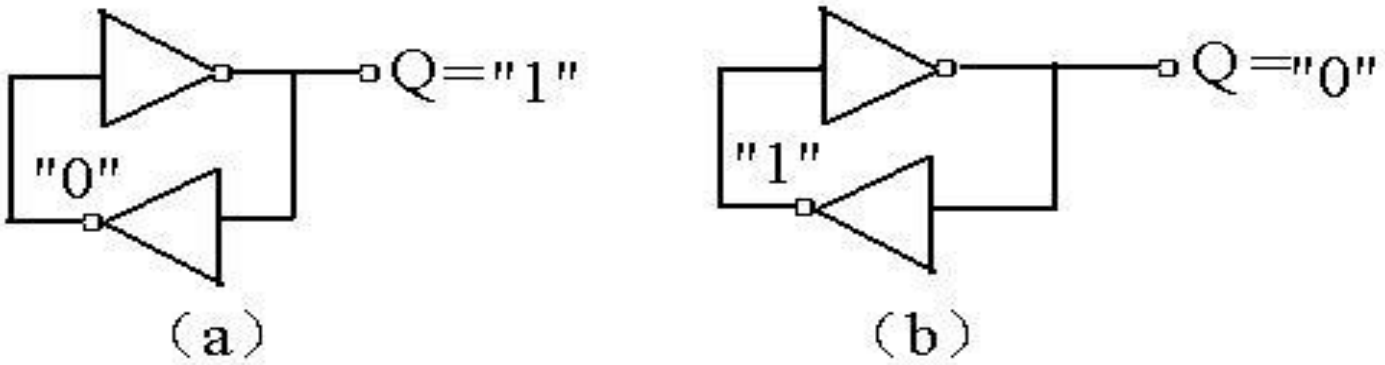
移位寄存器(Shift Register)

- 动态C^2MOS移位寄存器

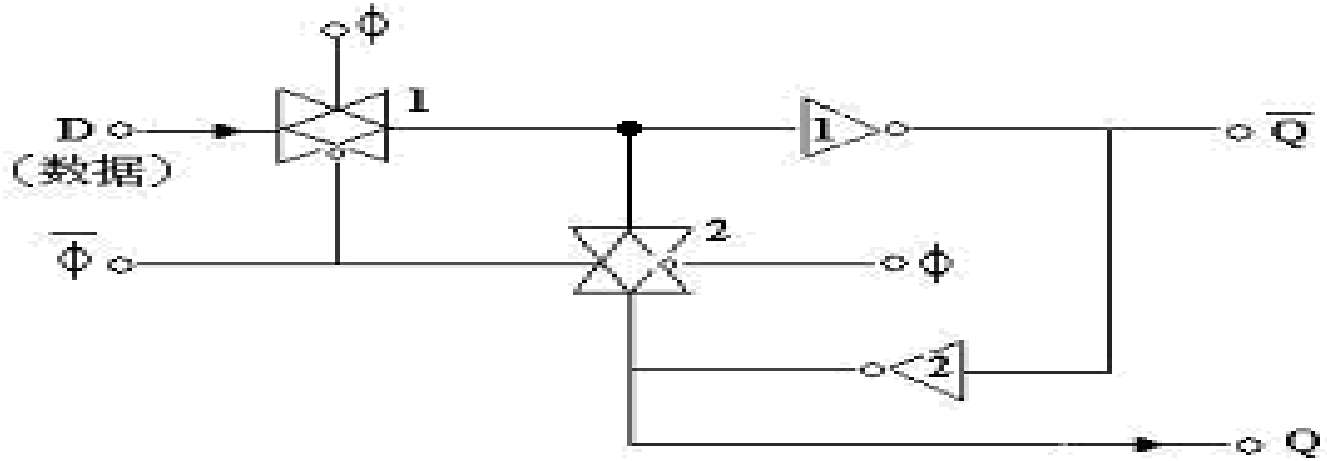


锁存器(Latch)

- 两个反相器构成正反馈闭环

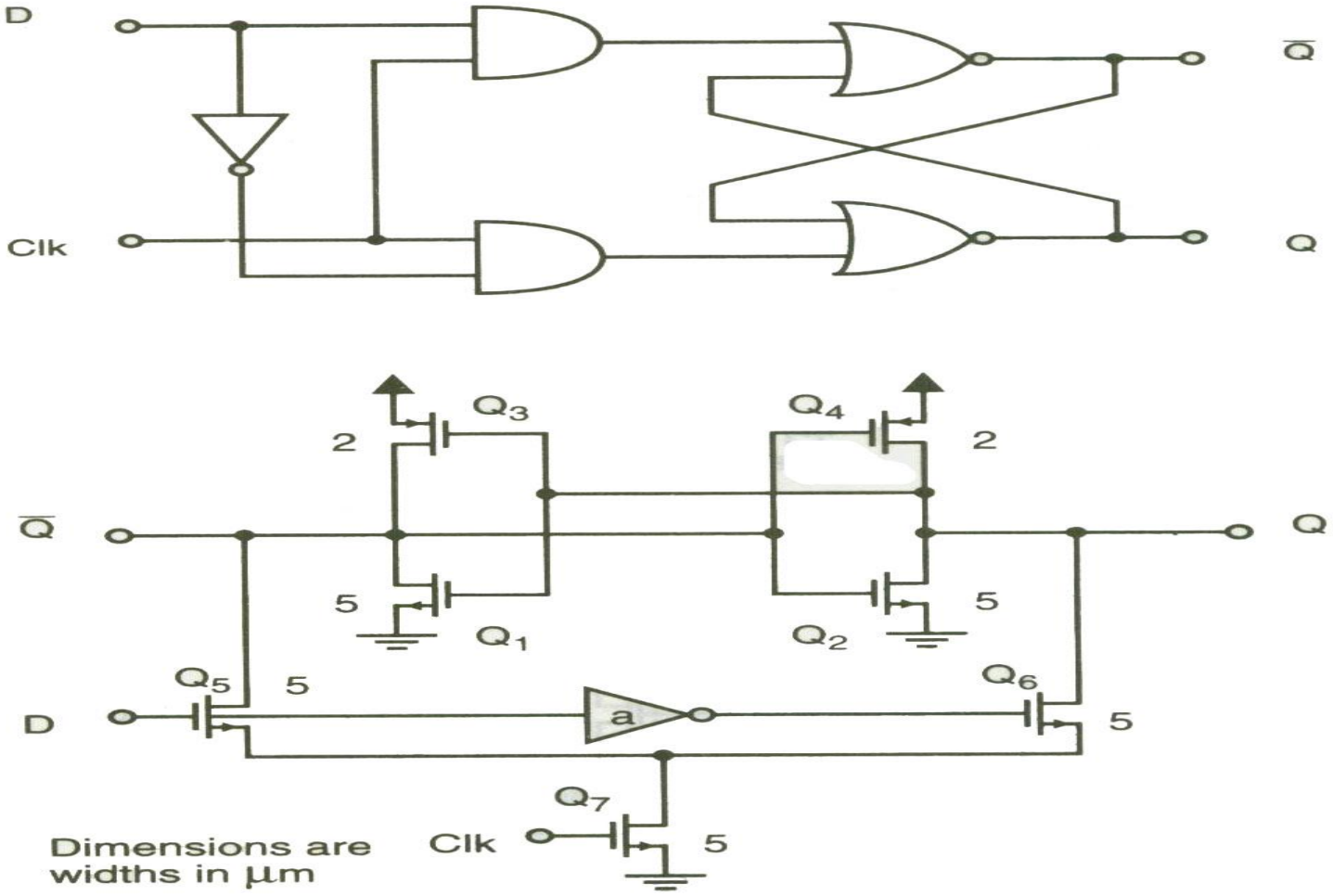


- 在反相器正反馈环中引入传输门构成锁存器



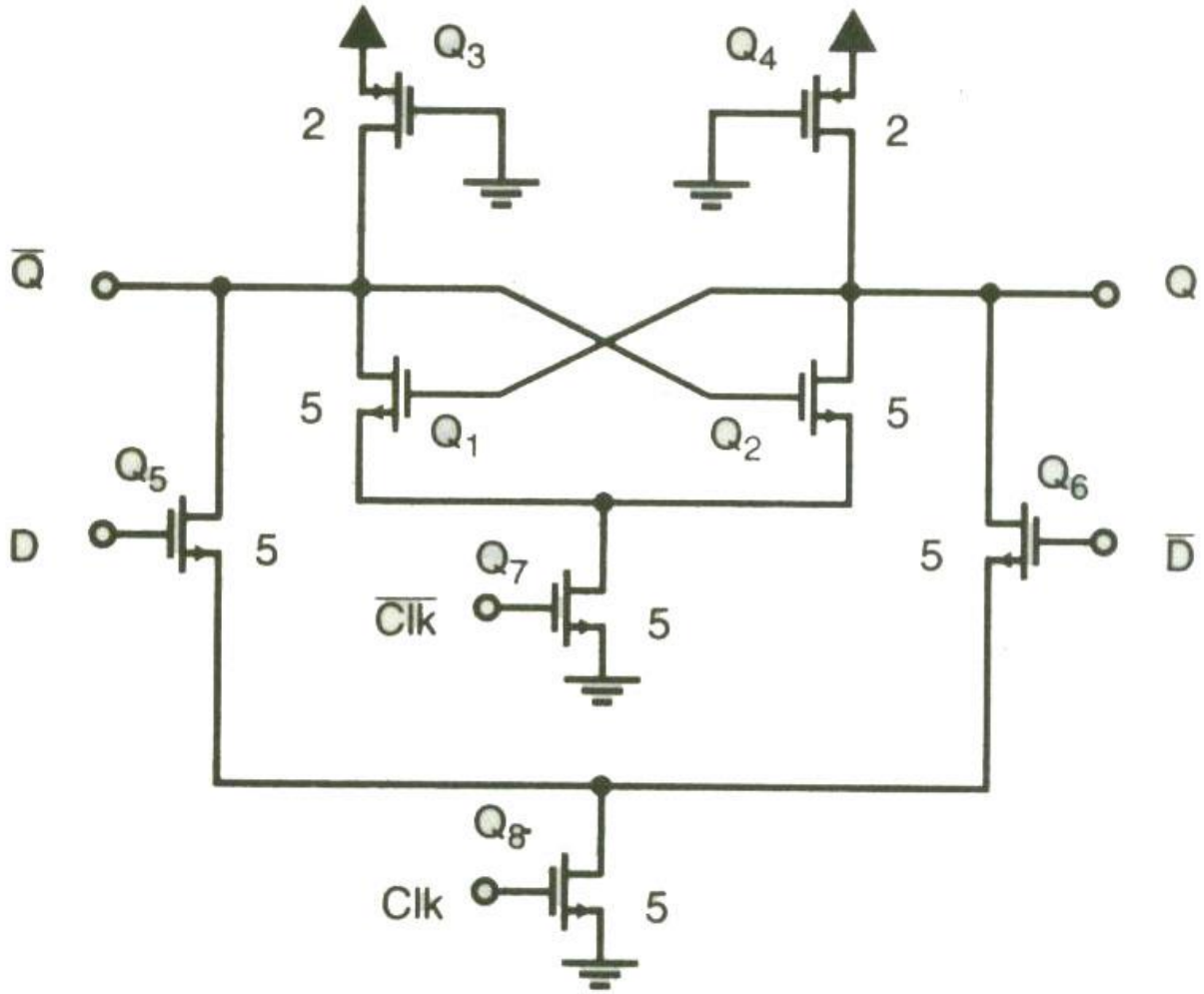
锁存器(Latch)

- 基于交叉耦合或非门锁存器及其CMOS实现



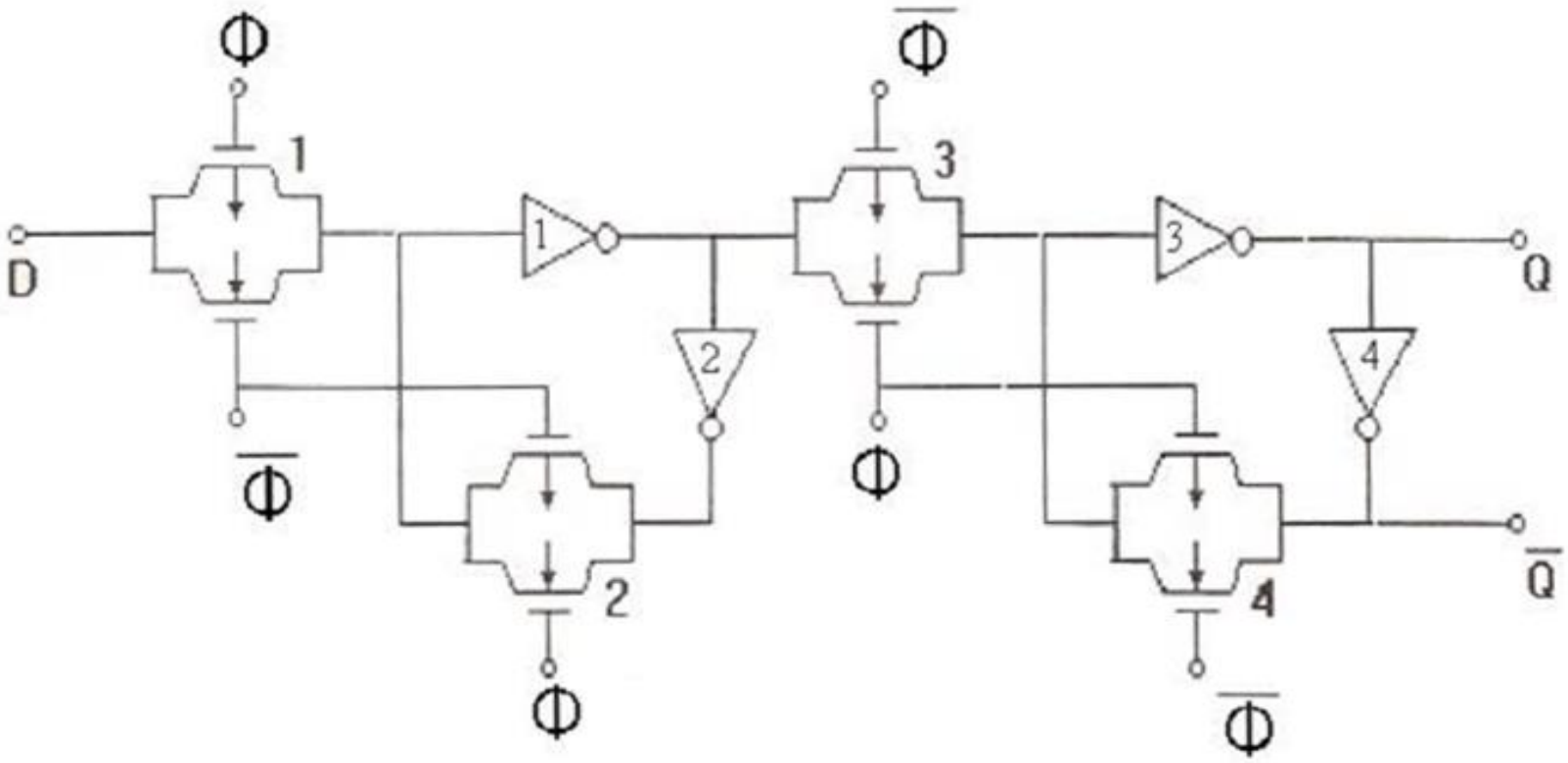
锁存器(Latch)

- 伪NMOS锁存器



触发器(Flip-Flop)

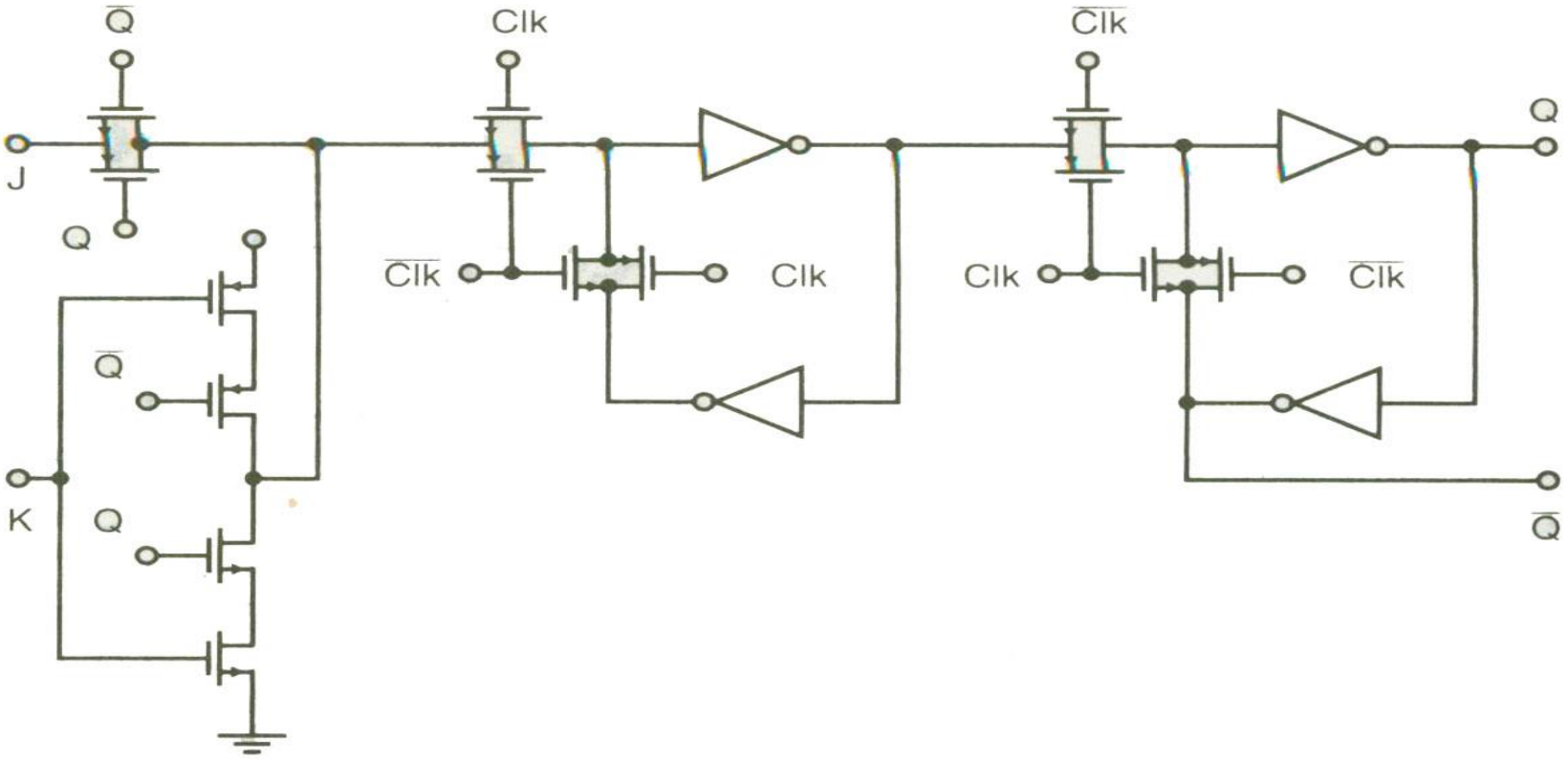
- CMOS主从D触发器



特征方程：$Q(t+1) = D(t)$

触发器(Flip-Flop)

- 传输门JK触发器

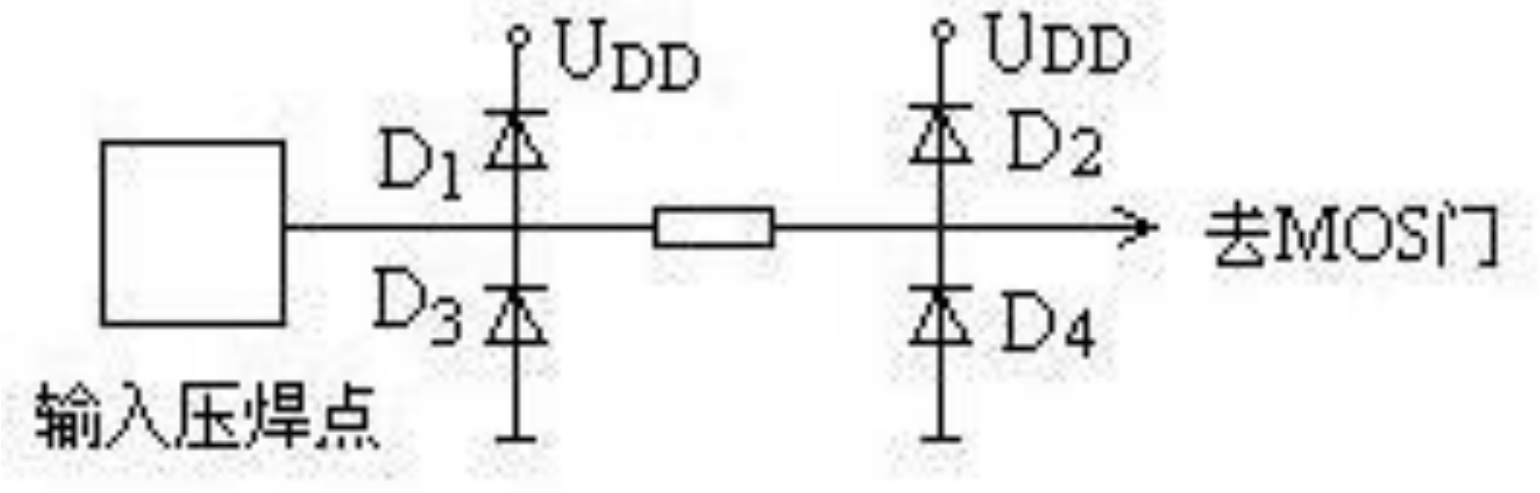


特征方程：$Q(t+1)=J(t)\overline{Q(t)}+\overline{K(t)}Q(t)$

通用I/O单元

- 输入保护电路

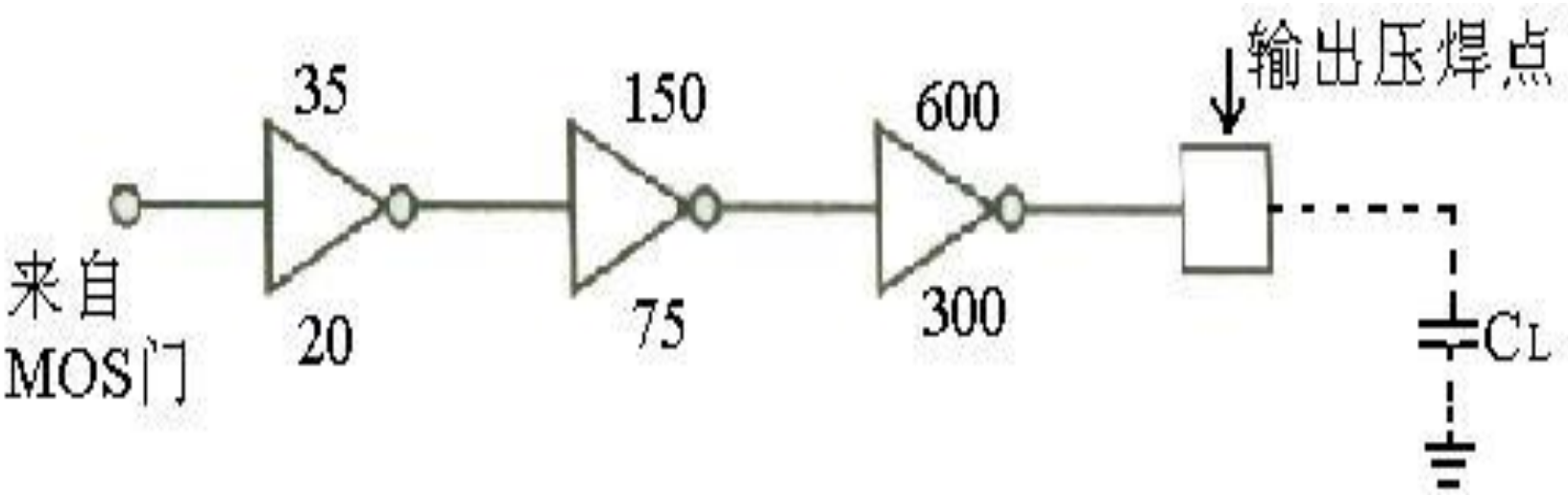
MOS管栅极不能悬空，而且正负最大电压都要加限幅措施以保证MOS管的安全。



通用I/O单元

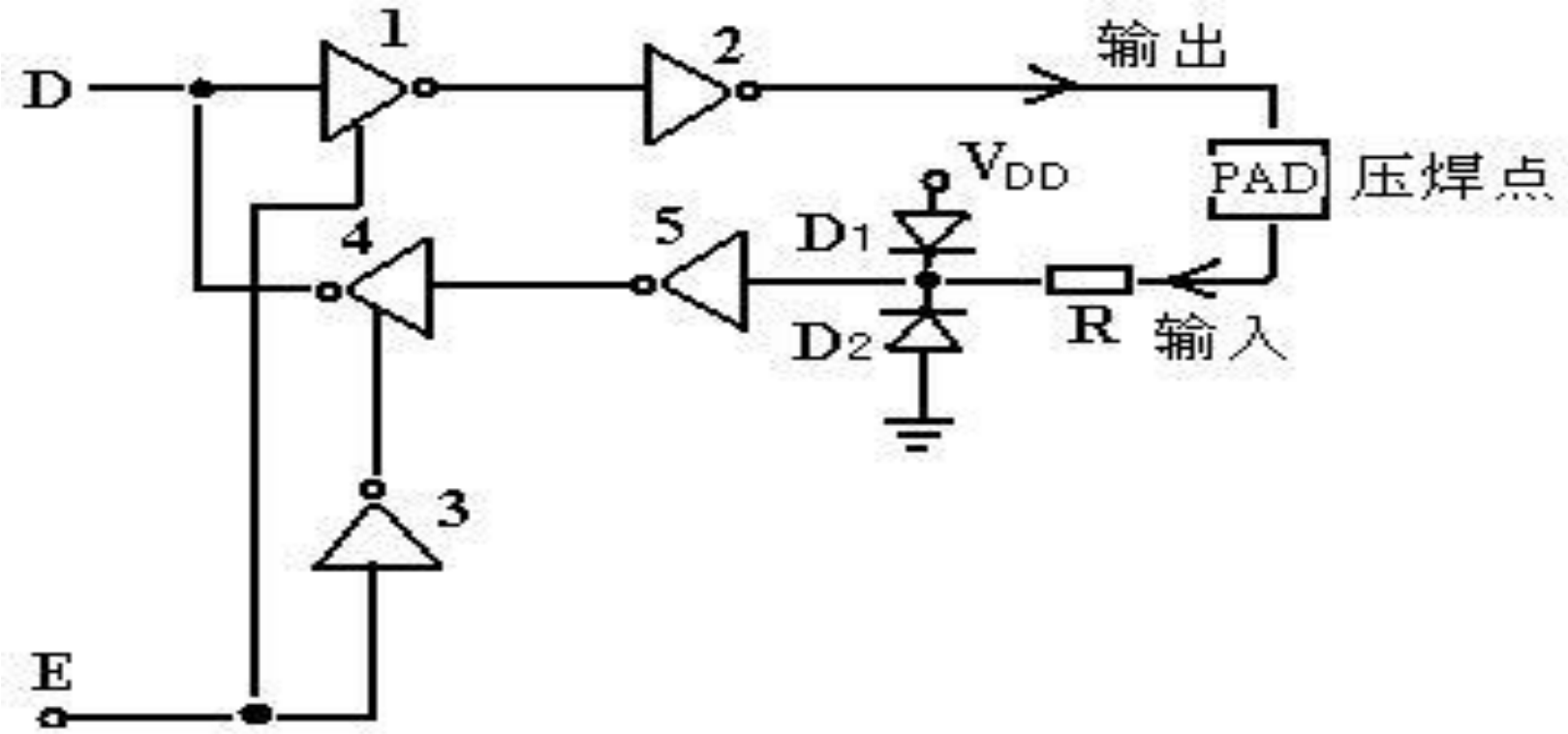
- 驱动大电容负载

如果负载电容C_L很大，则输出缓冲级的MOS管尺寸必须设计的很大，而且应采取逐级增大的方式。



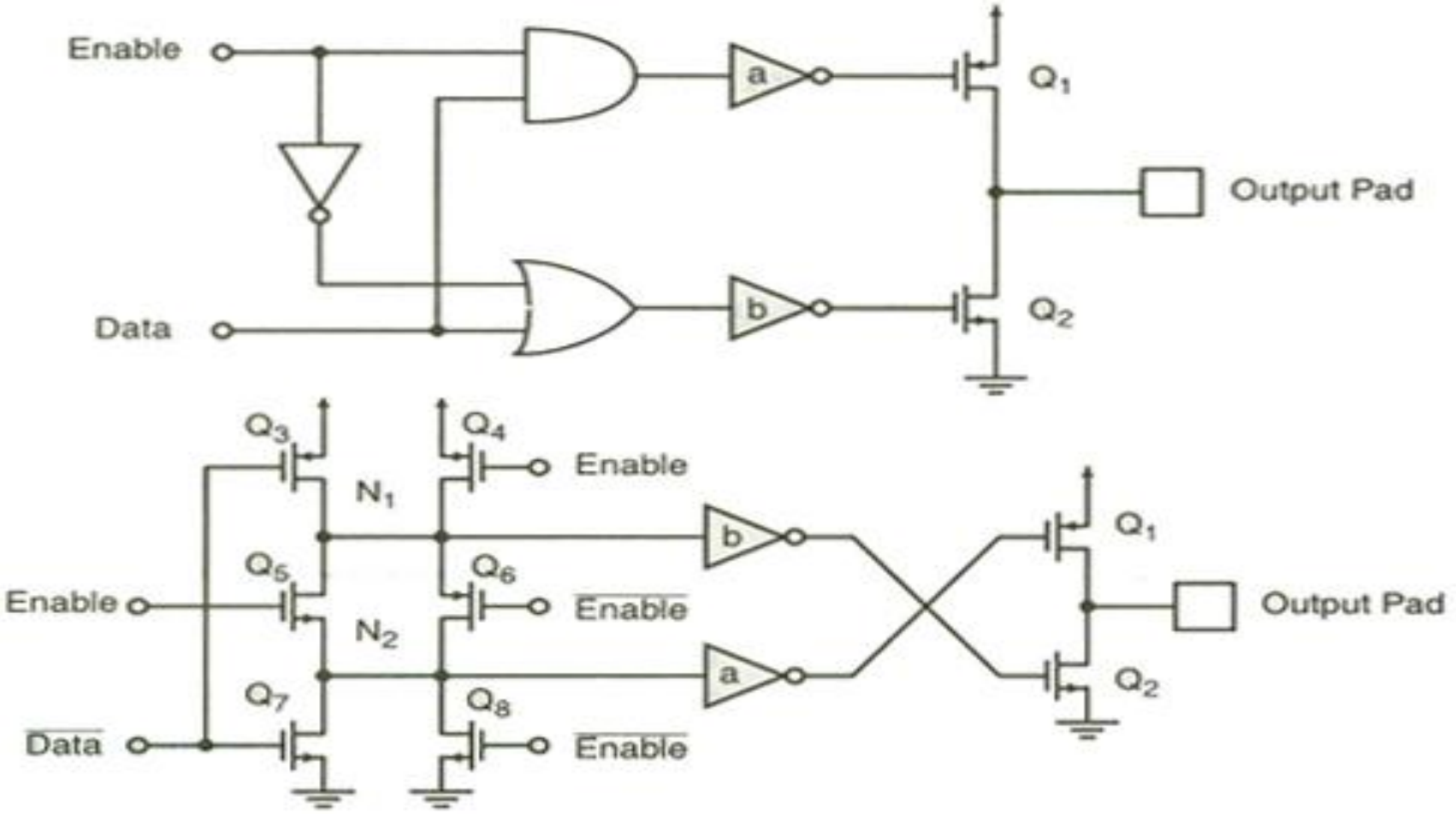
通用I/O单元

- 具有三态反相器的通用I/O单元



通用I/O单元

- 一种改进的三态输出电路



西安电子科技大学

假期快乐！

西安电子科技大学

第五章

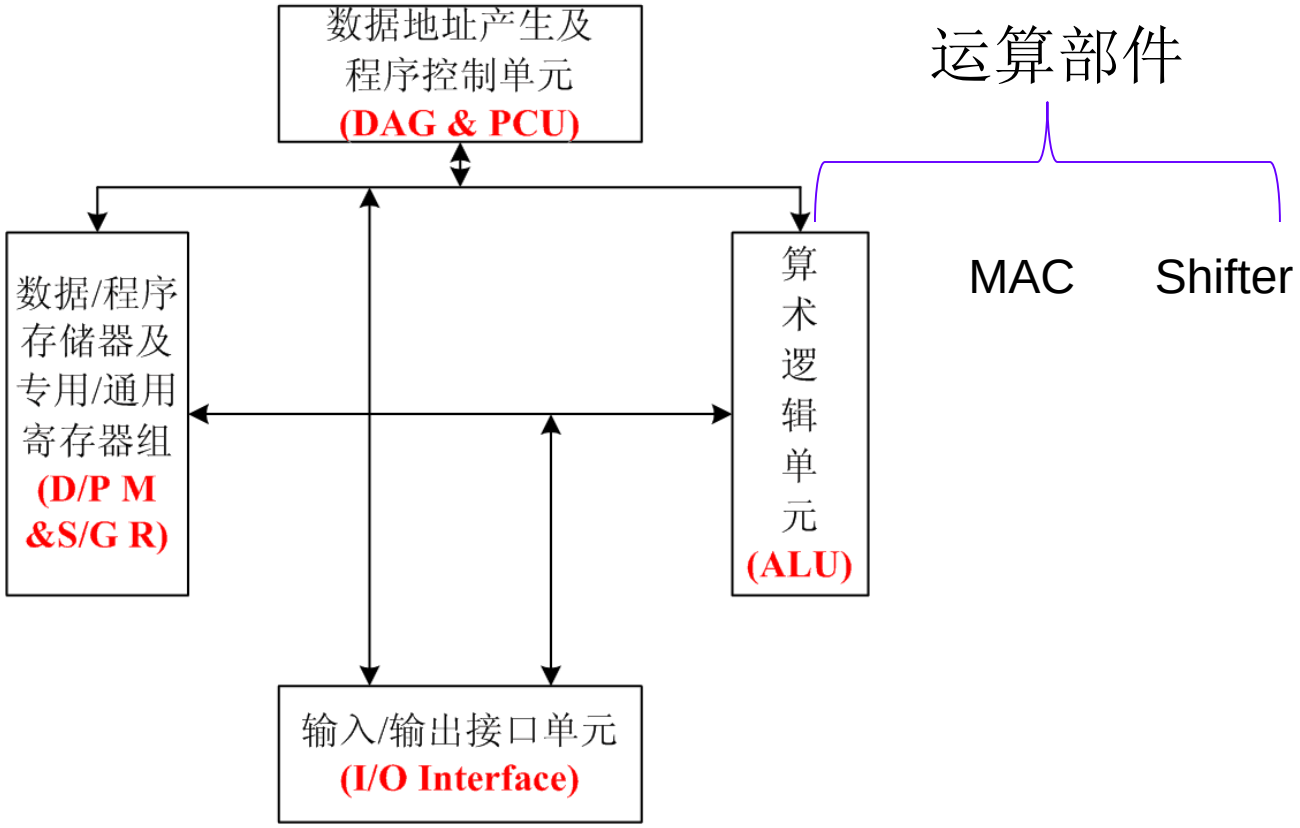
CMOS数字集成电路系统设计

本章提要

- 5.1　二进制加法器
- 5.2　桶形移位器
- 5.3　算术逻辑单元
- 5.4　二进制数乘法器
- 5.5　片上系统(SoC)的设计

典型数字信号处理器结构图

- 典型数字信号处理器结构图



西安电子科技大学

5.1 二进制加法器

1位半加器(Half Adder)

- 真值表

A	*B*	*S*	*C*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- 逻辑表达式

$$S = A \oplus B$$

$$C = A \cdot B$$

1位全加器(Full Adder)

- 真值表

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

- 逻辑表达式

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

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

进位产生函数$G = A \cdot B$

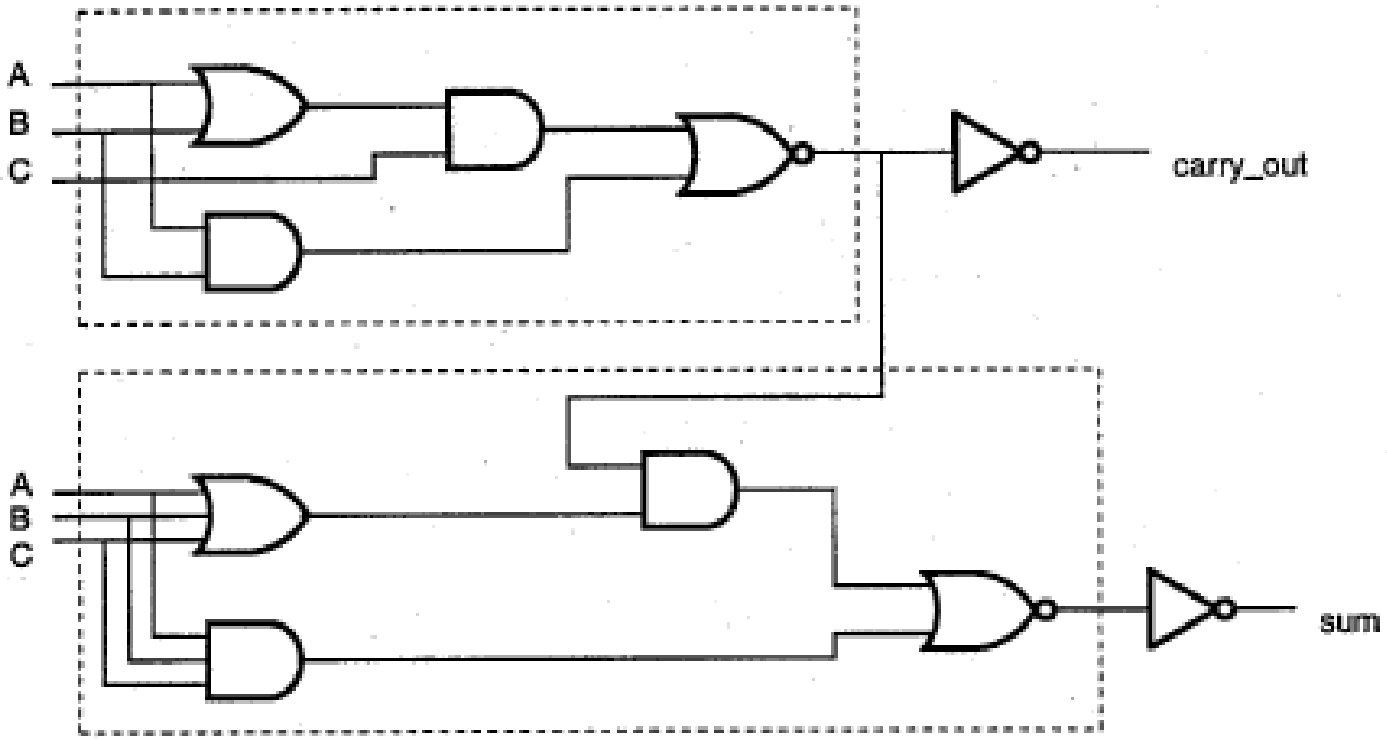
进位传递函数$P = A \oplus B$

$$S = P \oplus C_{\text{in}}$$

$$C_{\text{out}} = G + PC_{\text{in}}$$

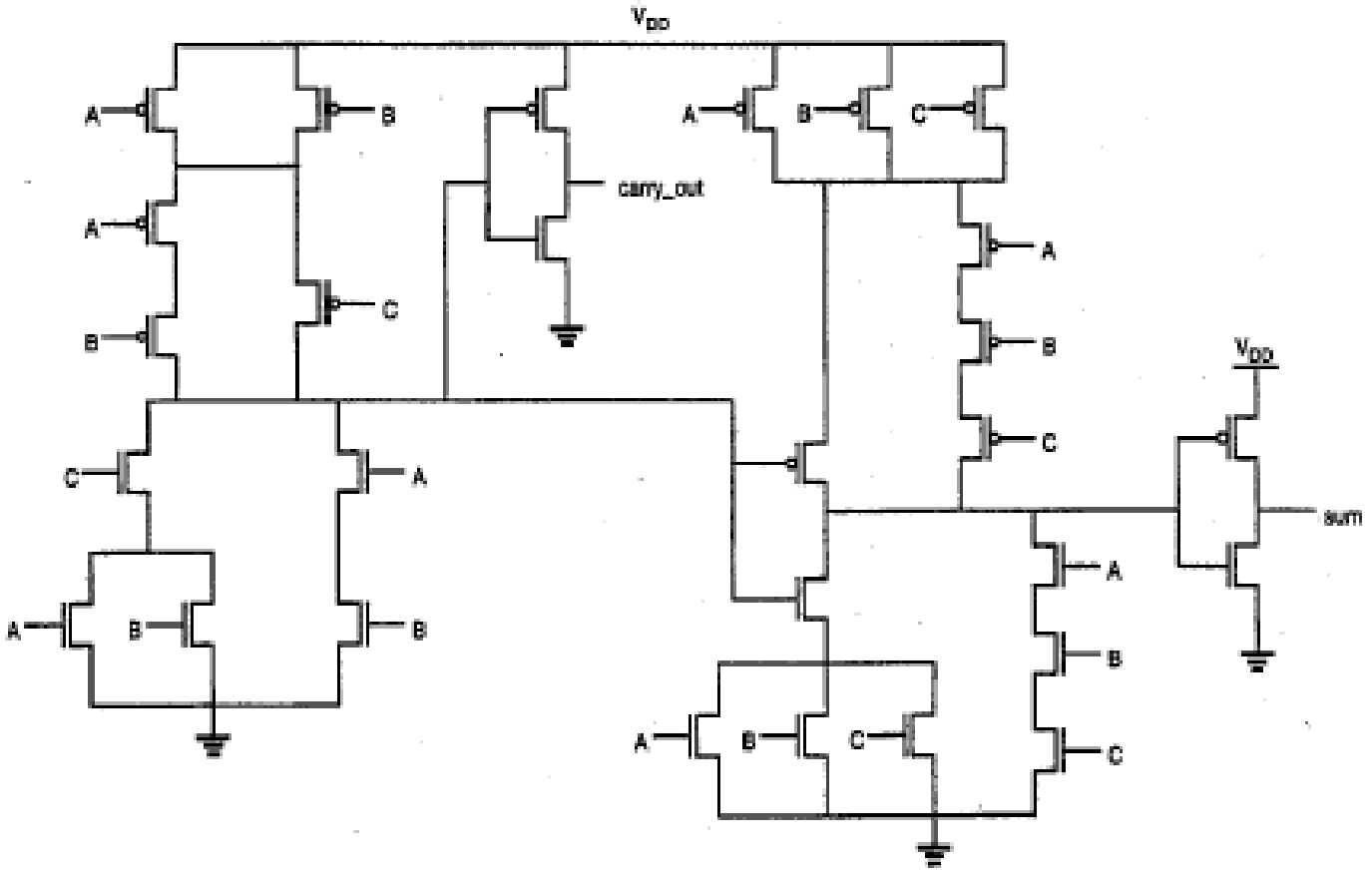
1位全加器(Full Adder)

- 1位全加器的逻辑原理图



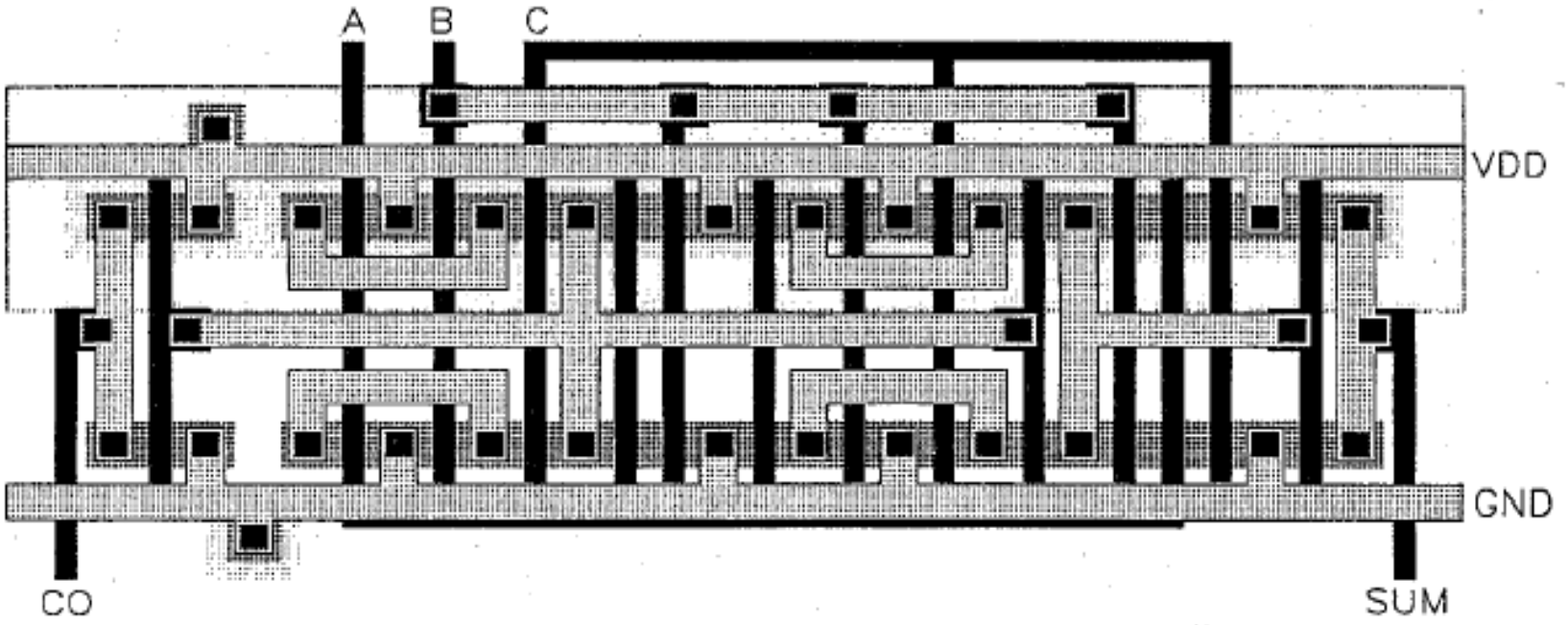
1位全加器(Full Adder)

- 1位全加器的全互补静态CMOS电路图



1位全加器(Full Adder)

- 1位全加器的集成电路版图



n位并行加法器

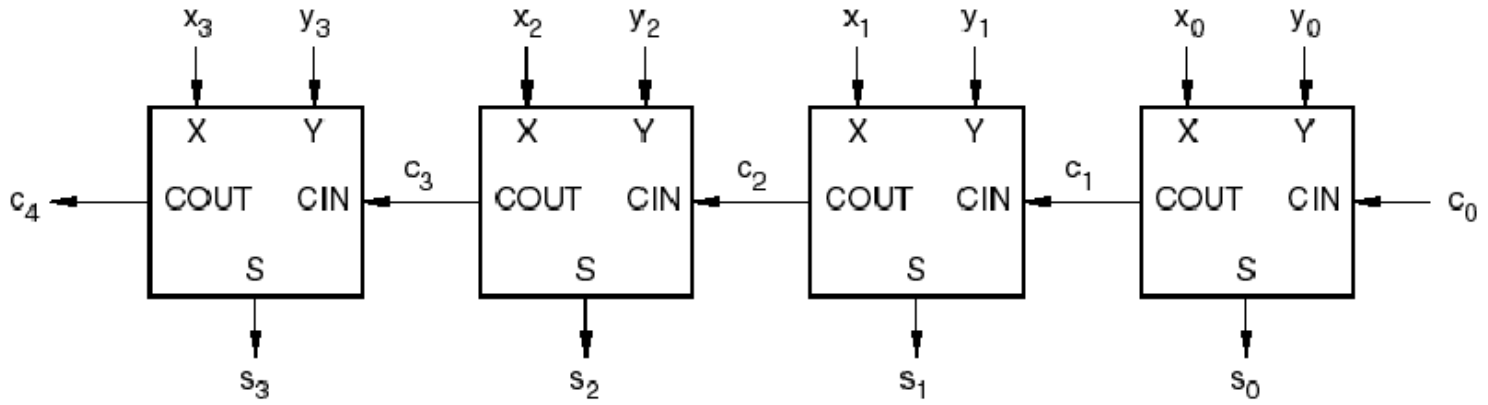
- 并行相加是指n位被加数中的每一位与n位加数中的各个对应位同时相加。n位并行加法器由n个一位全加器相互连接构成，其连接方式决定了该加法器的电路复杂程度和运算速度。

并行加法器
- 行波进位加法器（RCA,Ripple Carry Adder）
- 旁路进位加法器（CBA,Carry Bypass Adder）
- 选择进位加法器（CSA,Carry Select Adder）
- 超前进位加法器（CLA,Carry Lookahead Adder）

行波进位加法器

- 结构特点

该加法器每一位的进位输入均由相邻的低位送来，在最高位(n-1)得到最后的进位输出C_{out}，输出的"和"(S)则从各个相应位取得



- 性能特点

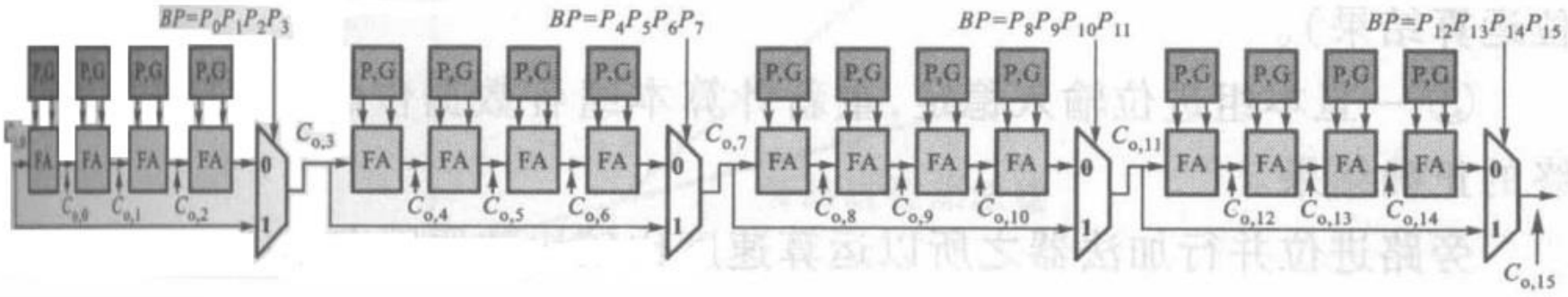
1. 电路简单、规则，易于IC版图的设计与实现；
2. 进位信号从最低位向最高位逐级传递，获得正确的结果。

- 延迟计算 $t_{\text{adder}} = (n-1)t_{\text{carry}} + t_{\text{s}}$

旁路进位加法器

- 结构特点

将n位加法运算划分成多个m位(通常m=4)的分组，组内采用行波进位，通过进位旁路选择将各分组连在一起。



- 求和与进位

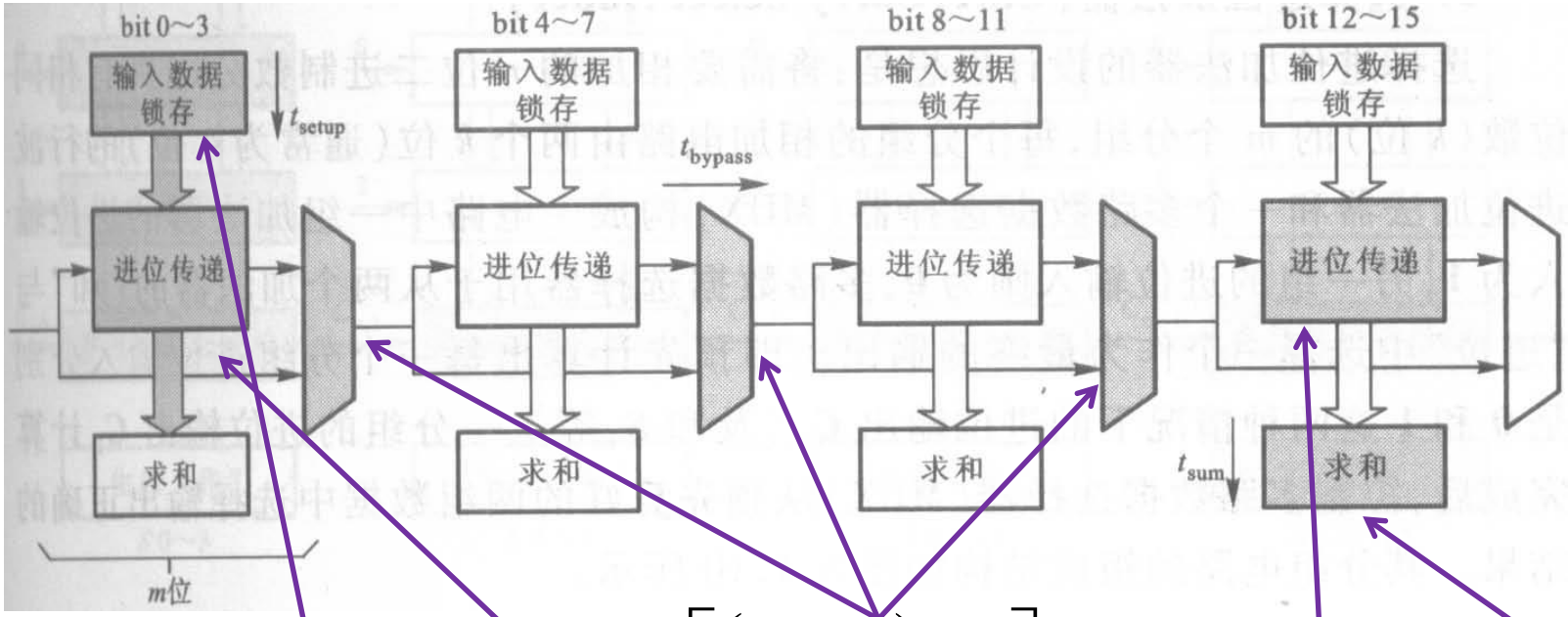
1. 进位旁路；
2. 非进位旁路。

旁路进位加法器

- 性能特点

1. 各分组的P和G以及BP函数可以同时并行计算；
2. 事先完成BP计算，进位不需在分组内，只需在分组间传递。

- 延迟分析

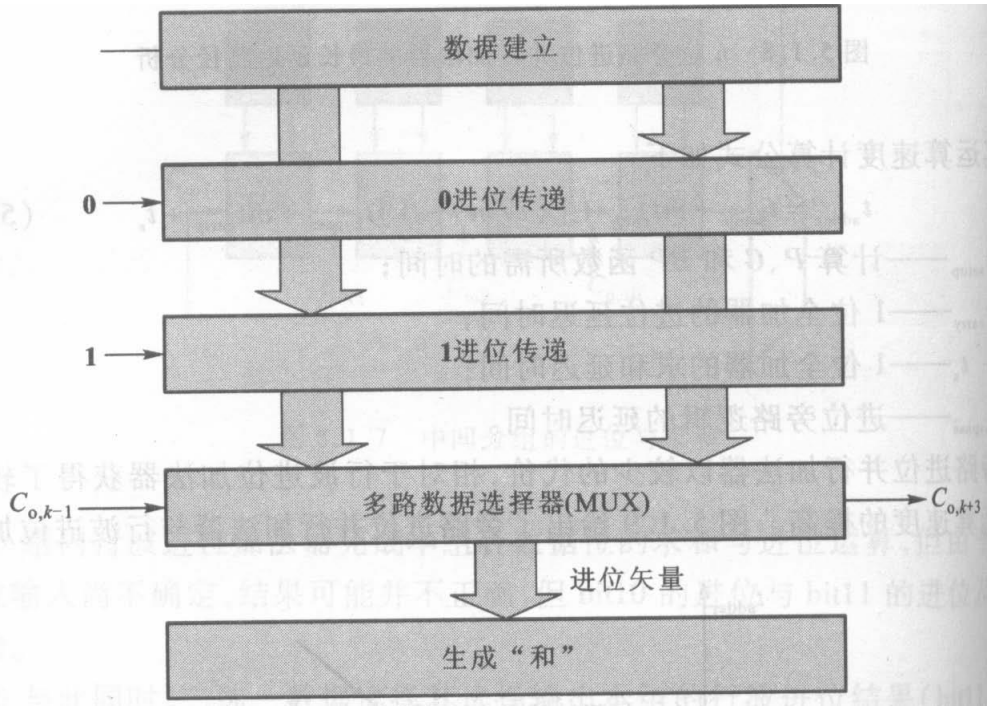


$$t_{adder} = t_{setup} + mt_{carry} + \left[(n/m) - 1\right] t_{bypass} + mt_{carry} + t_s$$

选择进位加法器

- 结构特点

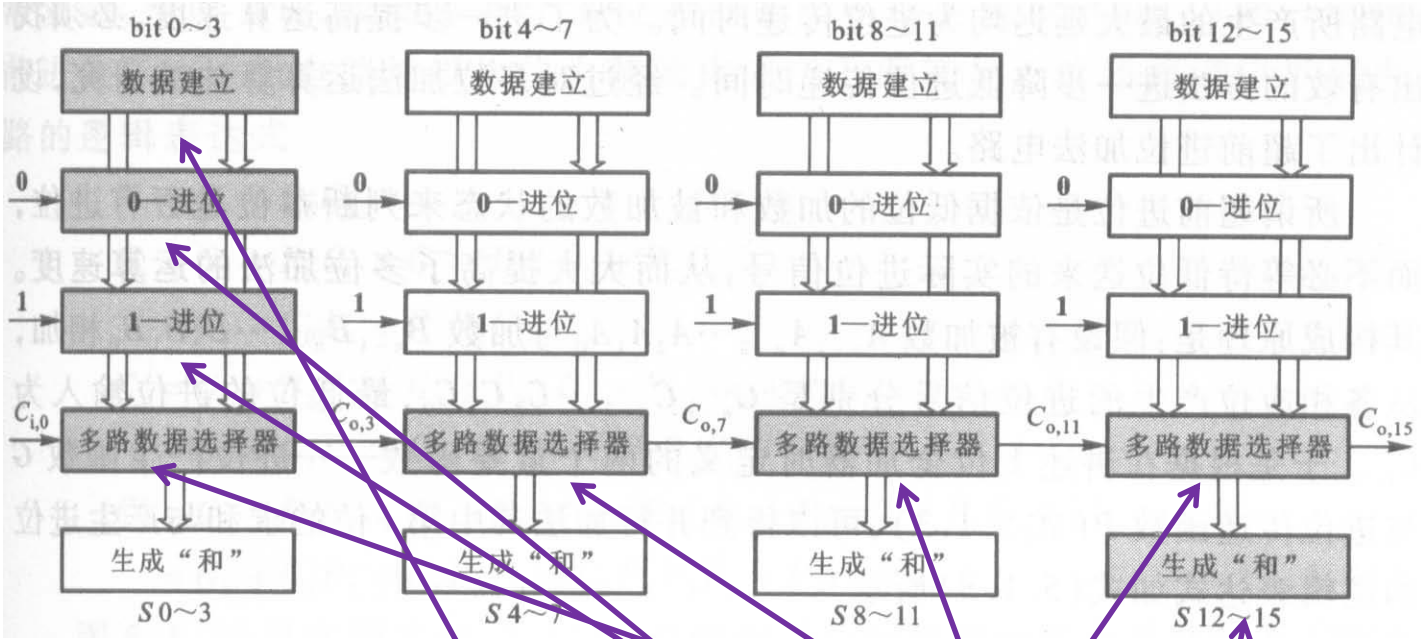
1. 将n位二进制数分成相同位数(k位)的m个分组
2. 每一组由两个k位的行波进位加法器和一个数据选择器(MUX)构成
3. 两个加法器的进位输入分别是” 1”和” 0”
4. MUX用于从两个加法器的“和”中选择一个作为最终的结果



选择进位加法器

- 性能特点

1. m个分组的数据以完全并行的方式相加；
2. 分组内的k位数据则按行波进位的方式相加；



- 延迟分析

$$t_{\text{adder}} = t_{\text{setup}} + \left(\frac{n}{m}\right) t_{\text{carry}} + m t_{\text{mux}} + t_{\text{s}}$$

超前进位加法器

- 结构特点

设计高速并行加法器的关键在于如何设计出延迟时间最小的进位信号处理电路。超前进位是在对多位加法运算算法进行深入研究的基础上，依据低位的加数和被加数的状态来判断本位是否有进位，而不必等待低位送来的实际进位信号，从而大大提高多位加法的运算速度。

- 超前进位算法

$$
\begin{aligned}
C_{\mathrm{i}} &= G_{\mathrm{i}} + P_{\mathrm{i}} C_{\mathrm{i-1}} \\
&= G_{\mathrm{i}} + P_{\mathrm{i}} \left(G_{\mathrm{i-1}} + P_{\mathrm{i-1}} C_{\mathrm{i-2}} \right) \\
&= \cdots \\
&= G_{\mathrm{i}} + P_{\mathrm{i}} \left\{ G_{\mathrm{i-1}} + P_{\mathrm{i-1}} \left[G_{\mathrm{i-2}} + P_{\mathrm{i-2}} \left(\cdots P_1 \left(G_0 + P_0 C_{\mathrm{in}} \right) \cdots \right) \right] \right\}
\end{aligned}
$$

超前进位加法器(CPA)

- 四位超前进位加法运算算法

$$C_0 = G_0 + P_0 C_{\text{in}}$$

$$C_1 = G_1 + G_0 P_1 + P_1 P_0 C_{\text{in}}$$

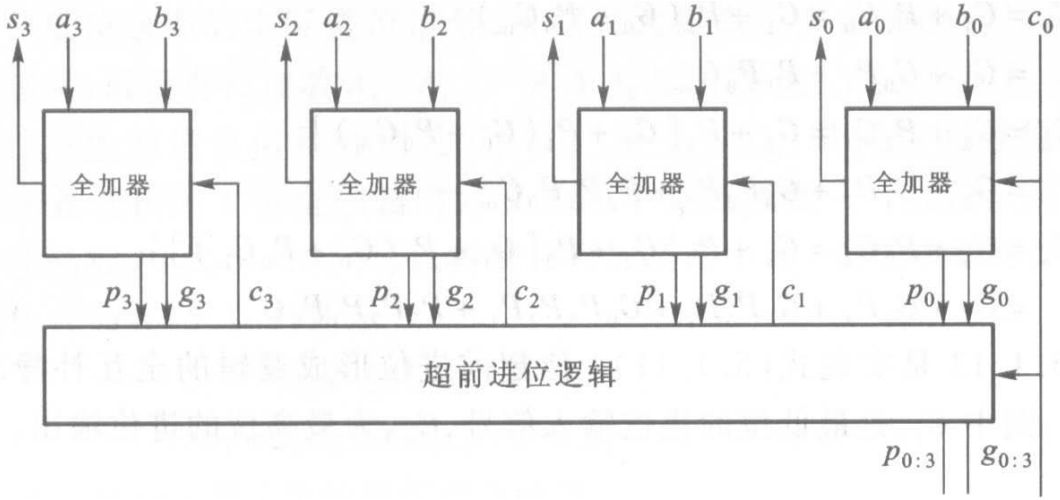
$$C_2 = G_2 + G_1 P_2 + G_0 P_2 P_1 + P_2 P_1 P_0 C_{\text{in}}$$

$$C_3 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + P_3 P_2 P_1 P_0 C_{\text{in}}$$

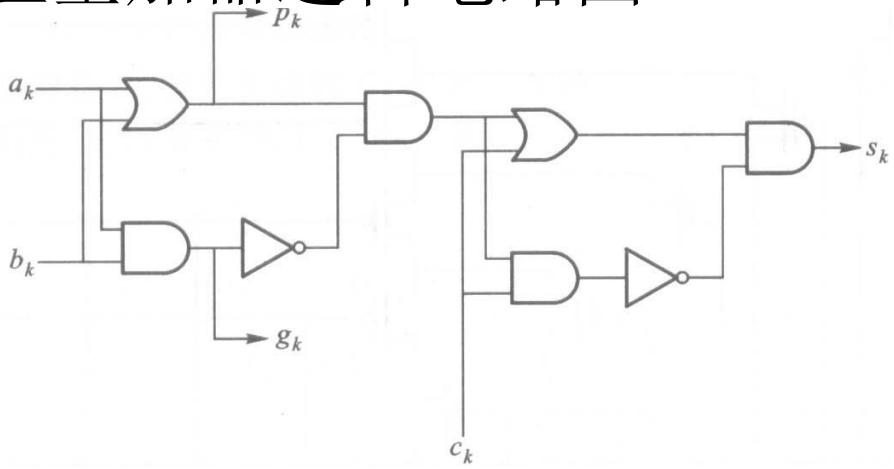
从上面公式可以看出，每一特定位的进位信号可以直接从本位以及比它低的各位加数、被加数和C_{in}的状态来作出判断，而不需要等待低位实际送来的进位信号。这样一来，任意一位所需的进位信号只要各个相关信号输入后经过两级门延迟即可获得，加法的运算速度与参与运算操作数的位数无关。

超前进位加法器(CPA)

- 4位超前进位加法器电路



- 改进后的1位全加器逻辑电路图



5.2 桶形移位器(Barrel Shifter)

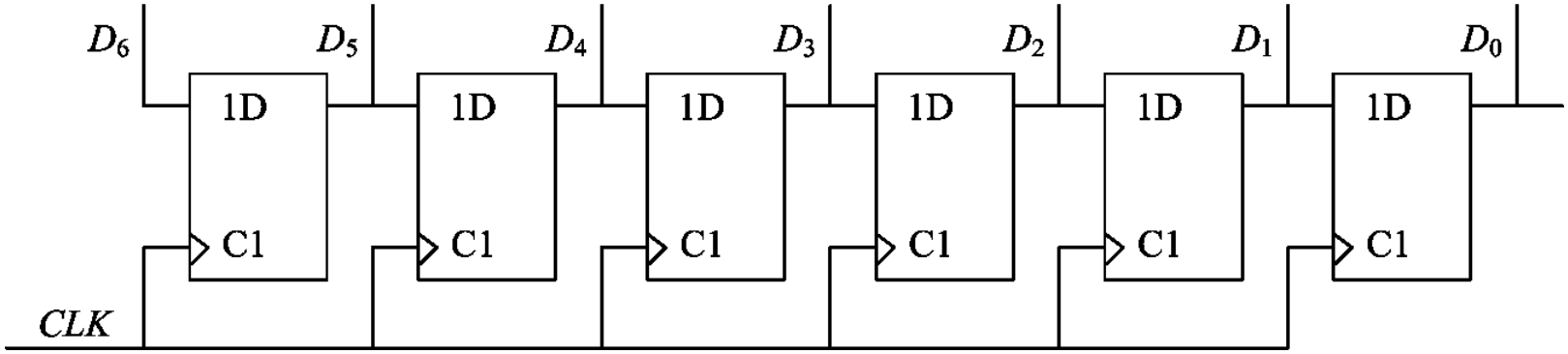
移位操作的类型

- 逻辑左移
- 算术左移
- 循环左移
- 逻辑右移
- 算术右移
- 循环右移

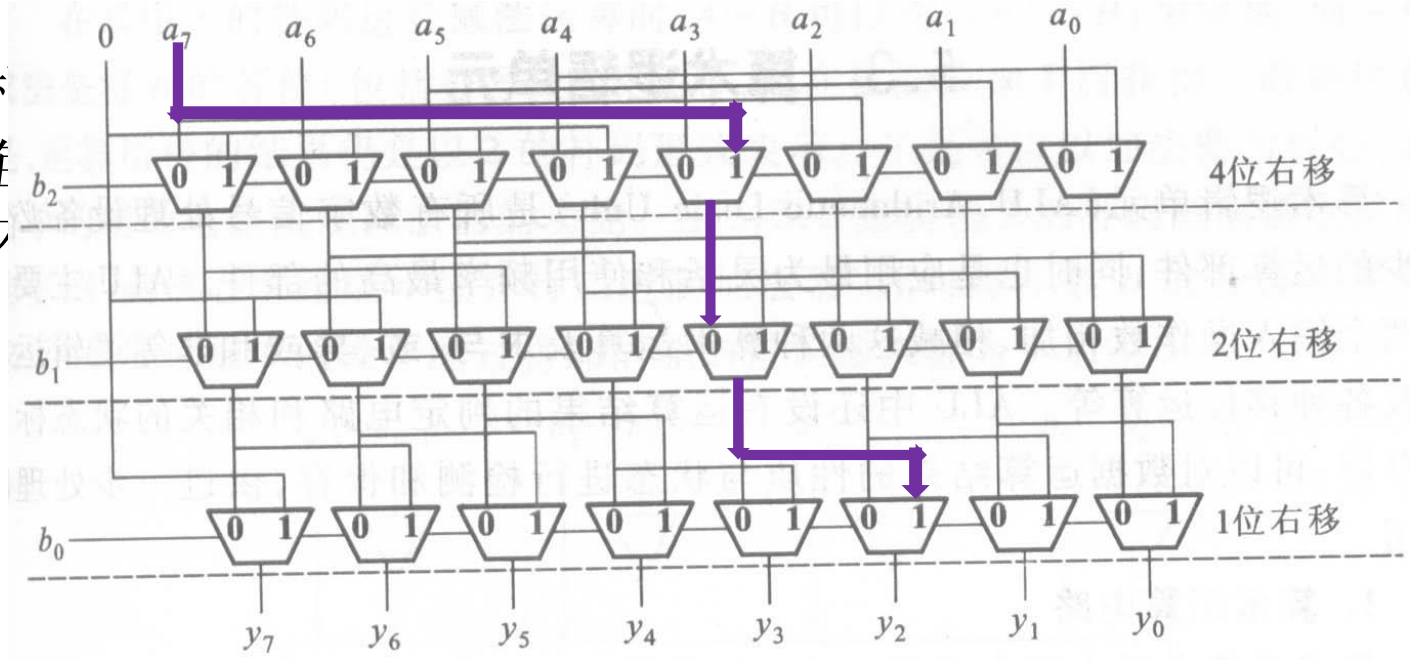
操　作　类　型	运　算　结　果
3位逻辑左移	$d_4\ d_3\ d_2\ d_1\ d_0\ 0\ 0\ 0$
3位算术左移	$d_7\ d_3\ d_2\ d_1\ d_0\ 0\ 0\ 0$
3位循环左移	$d_4\ d_3\ d_2\ d_1\ d_0\ d_7\ d_6\ d_5$
3位逻辑右移	$0\ 0\ 0\ d_7\ d_6\ d_5\ d_4\ d_3$
3位算术右移	$d_7\ d_7\ d_7\ d_7\ d_6\ d_5\ d_4\ d_3$
3位循环右移	$d_2\ d_1\ d_0\ d_7\ d_6\ d_5\ d_4\ d_3$

用触发器构成的移位电路

- 电路特点
  - 典型的时序电路
  - 每个时钟周期只能移动1位
  - 完成不同位数的移位操作所需时间不同
- 6位右移移位寄存器



桶形移位器电路



- 电路特点

1. 完全由组合逻辑电路构成；
2. 移位的速度取决于MUX组分级的层数，与移位操作的位数无关

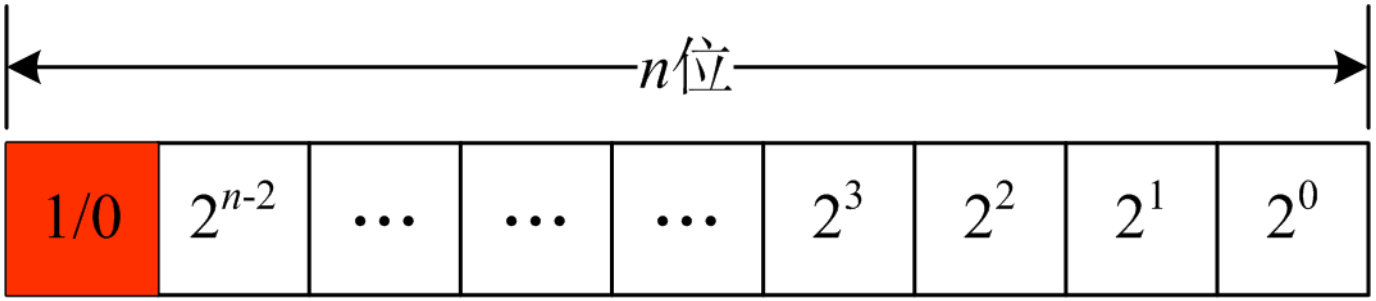
5.3 算术逻辑单元
(ALU, Arithmetic Logic Unit)

ALU主要完成输入操作数的相加、相减这两种**算术运算**及与、或、异或和非等**逻辑运算**以及**移位运算**。

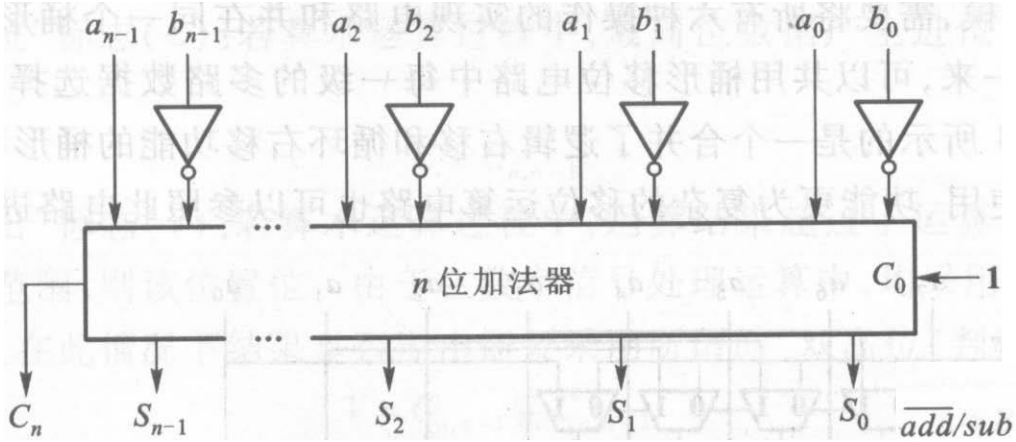
二进制补码

- 二进制补码的特点

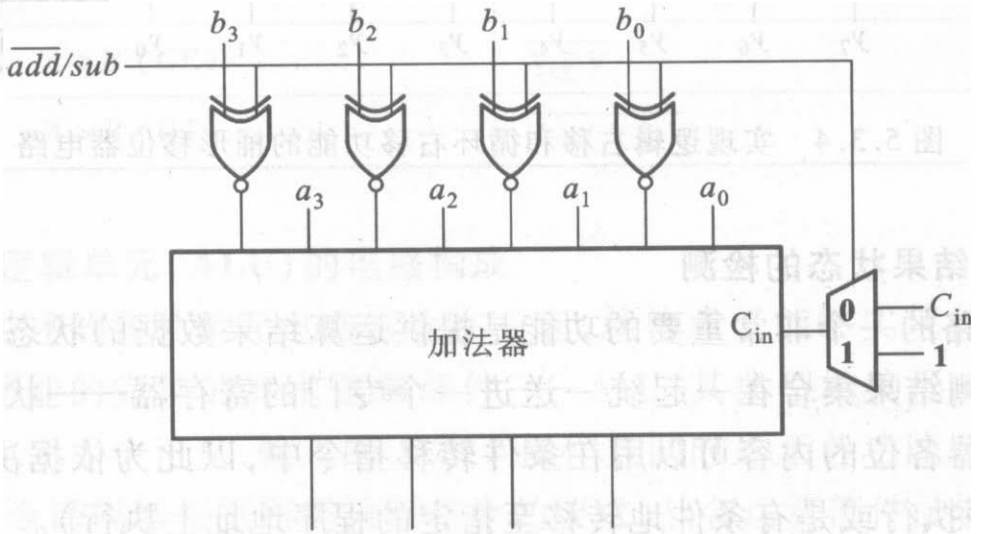
1. 最高位是符号位，“0”表示正数，“1”表示负数
2. 与有符号数的原码表示不同，数字0的表示是唯一的
3. 正数的补码与原码相同
4. n位补码对有符号数整数的表示范围：-2^{n-1}~$2^{n-1}-1$



算术运算电路



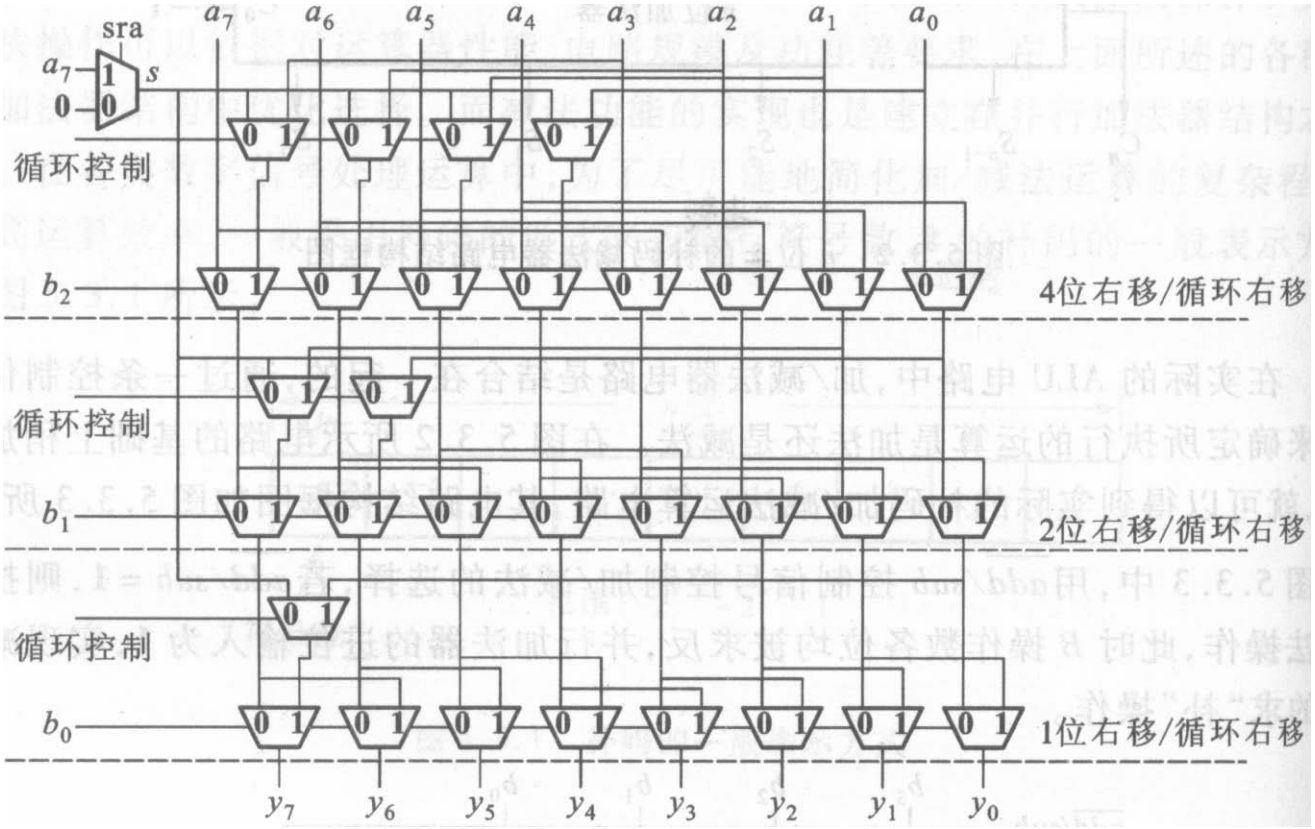
*n*位二进制的补码
减法器电路结构框图



*n*位二进制的补码
加/减法器电路结构框图

逻辑运算和移位运算电路

- 逻辑运算电路
- 移位运算电路



运算结果状态的检测

ALU电路的状态检测结果集合在**状态寄存器**中。

- 结果数据表示为$R = r_{n-1}r_{n-2}\cdots r_2 r_1 r_0$
- “零”标志(Z)：若运算结果各位均为0，则该位置位(置为1)

$$Z = \overline{r_{n-1} + r_{n-2} + \cdots + r_2 + r_1 + r_0}$$

- “负”标志(N)：若运算结果为负数，则该位置位

$$N = r_{n-1}$$

- “进位”标志(C)：若最高位数据产生进位，则该位置位

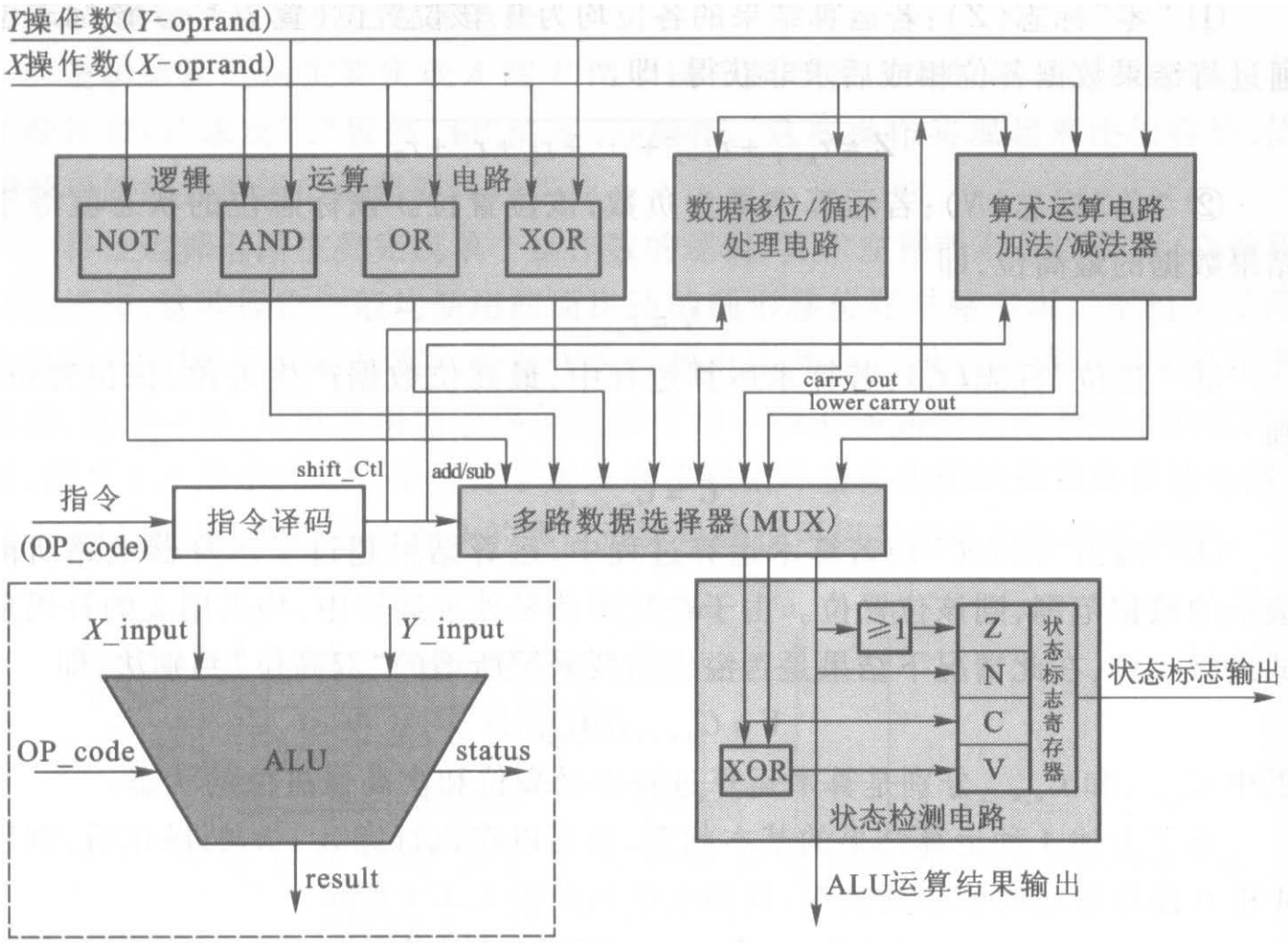
$$C = C_{o,n-1}$$

- “溢出”标志(V)：若结果超过了数据范围，则该位置位

$$V = C_{o,n-1} \oplus C_{o,n-2}$$

算术逻辑单元的电路构成

- 电路构成



5.4 二进制数乘法器

目前，大部分数字信号处理器DSP(Digital Signal Processor)中集成有专门的乘－加运算电路MAC(Multiplier and Accumulator)，对乘法器最基本的要求就是实现乘法运算的速度尽可能快。

二进制数乘法运算

			1	0	1	0	1	0	被乘数(6bit)
×					1	1	1	1	乘数(4bit)
			1	0	1	0	1	0	
		1	0	1	0	1	0		
	1	0	1	0	1	0			
+	1	0	1	0	1	0			
1	0	0	1	1	1	0	1	1	0	乘积结果(6+4=10bit)

部分积

$$(111111)_2 \times (1111)_2 = (1110110001)_2$$

				A_3	A_2	A_1	A_0
×				B_3	B_2	B_1	B_0
A_iB_j称为“部分积”			A_3B_0	A_2B_0	A_1B_0	A_0B_0	
		A_3B_1	A_2B_1	A_1B_1	A_0B_1		
	A_3B_2	A_2B_2	A_1B_2	A_0B_2			
+	A_3B_3	A_2B_3	A_1B_3	A_0B_3			

n bit数据乘以*m* bit数据，其乘积的结果是(*n*+*m*) bit

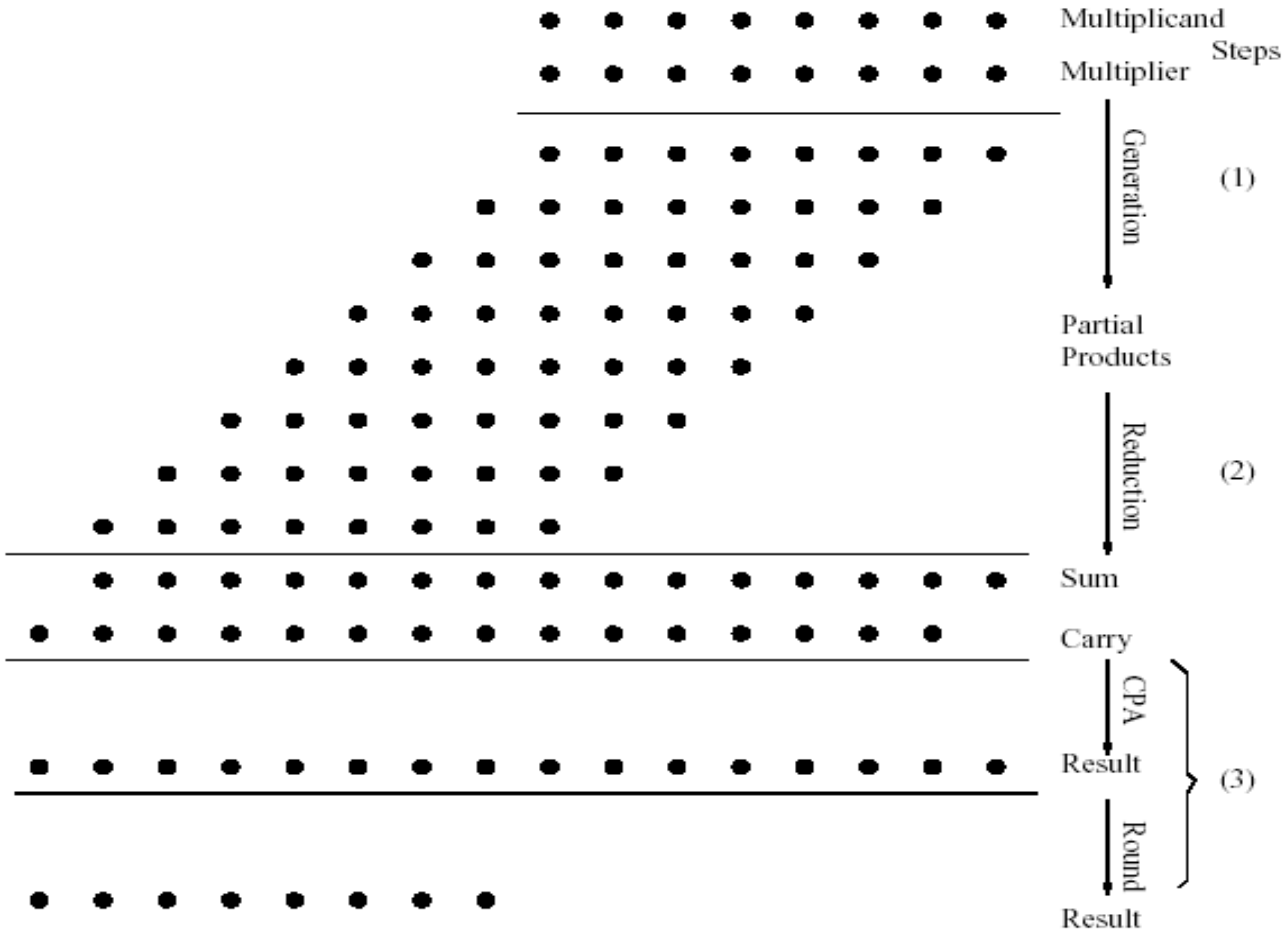
二进制数乘法运算

- 二进制数乘法运算的步骤

1. 从输入数据中依照乘数的状态**产生部分积**(如何产生部分积使乘法运算速度加快是设计乘法器电路的主要问题之一)。
2. 将各个**部分积**沿垂直方向**相加产生最终的结果**(要采用某种运算策略，将所有的部分积最终合并(化简)成部分积和(Sum)与部分积进位(Carry)两部分。由于该运算策略与电路的实现结构关系紧密，所以它也是乘法器电路研究的重要问题之一)。
3. 将上一步骤获得的部分积**和(Sum)与部分积进位(Carry)相加**获得最终的乘积。

二进制数乘法运算

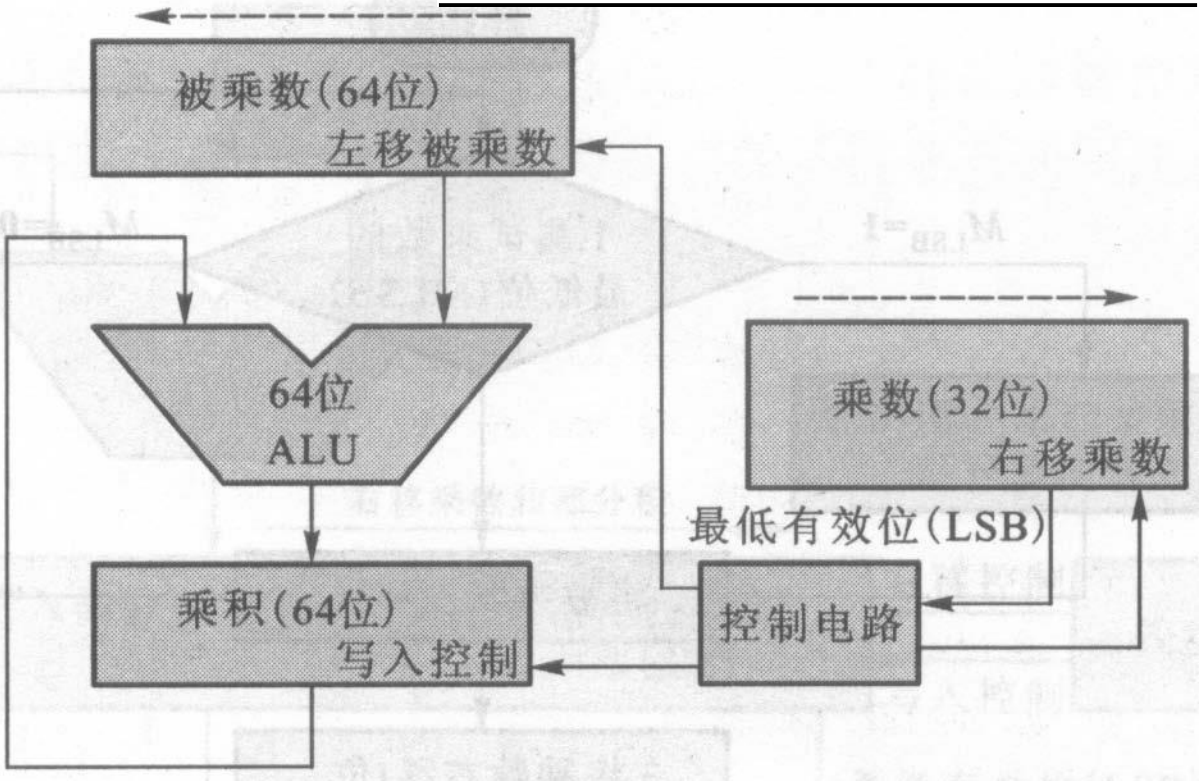
- 用“点图(Dot Diagram)”来表示二进制乘法运算的步骤



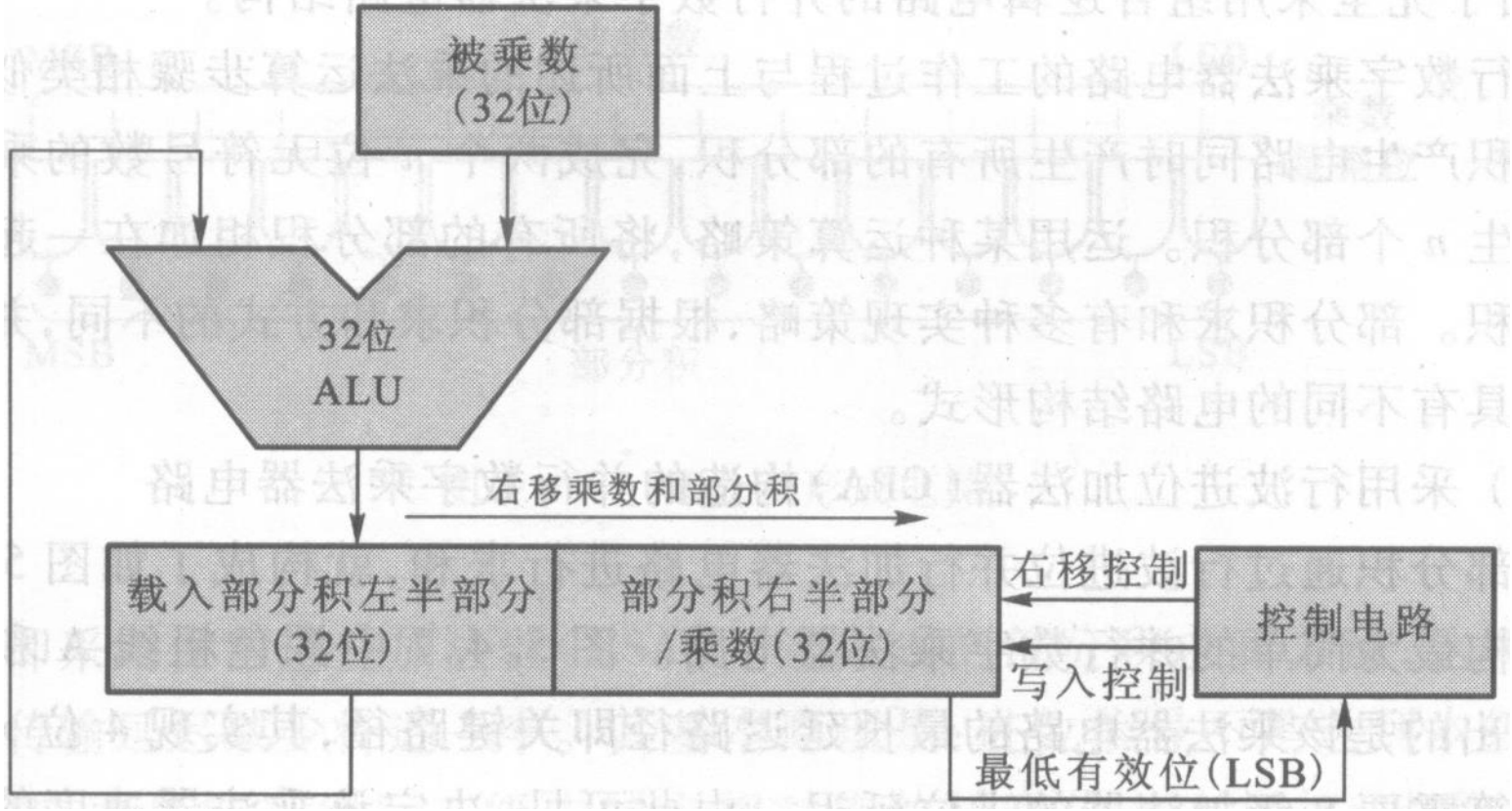
移位式无符号乘法器

- 电路构成
  - 乘数/被乘数移位寄存器
  - 乘积结果寄存器
  - 并行加法器
  - 相关控制电路
- 结构特点
  - 在时钟信号(clock)作用下同步进行

$$
\begin{array}{llllllll}
 & & & & A_3 & A_2 & A_1 & A_0 \\
\times & & & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$



实用型移位式无符号乘法器

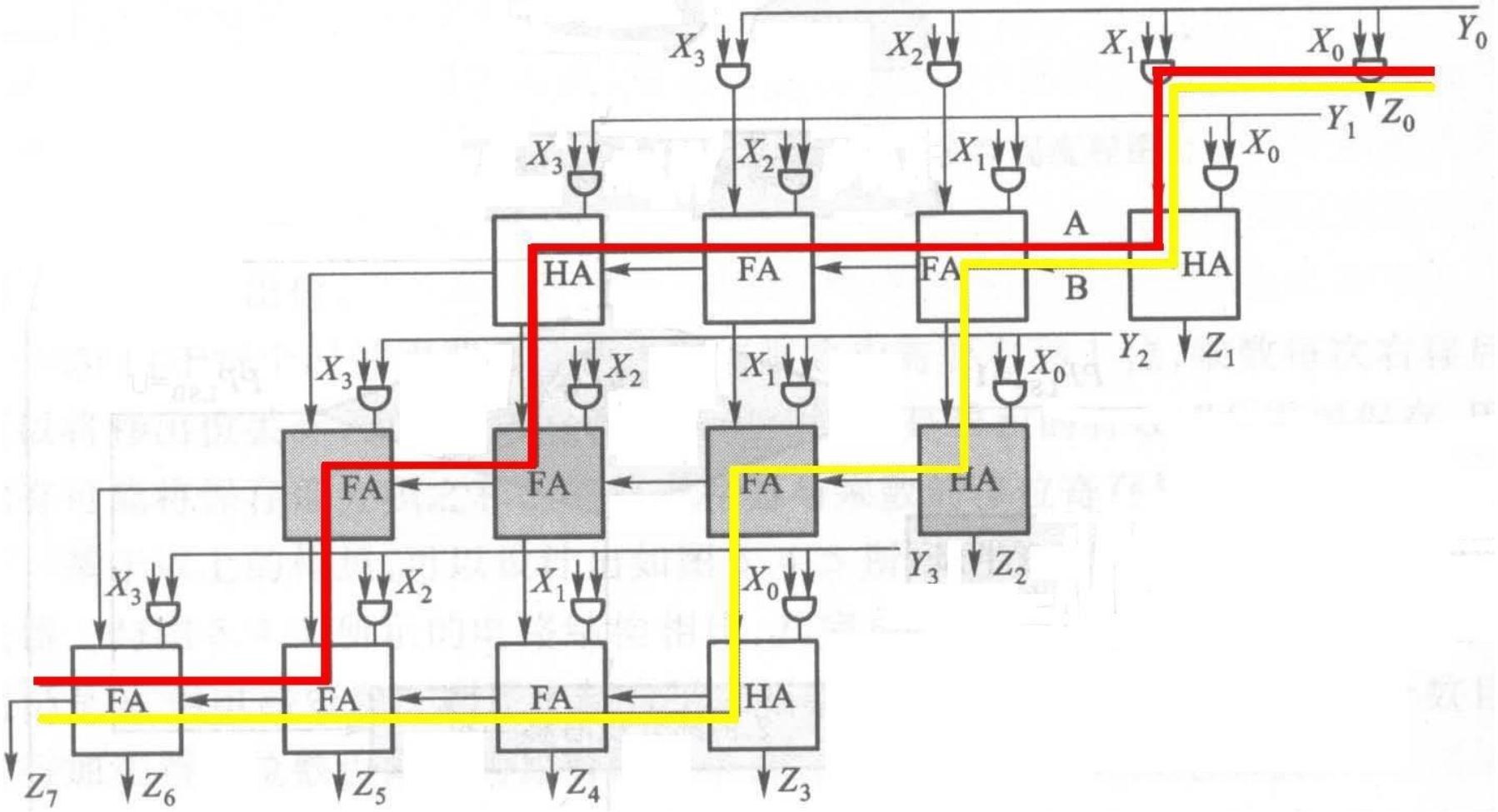


并行无符号数乘法器

并行数字乘法器完全采用组合逻辑电路，其工作过程与上面所述的乘法运算步骤相类似，即：通过部分积产生电路同时产生所有的部分积，运用某种运算策略，将所有的部分积最终合并(化简)成部分积和(Sum)与部分积进位(Carry)两部分，然后将这两部分通过多位并行加法器相加得到最终的结果。根据部分积化减策略的不同，并行数字乘法器具有不同的电路结构形式。

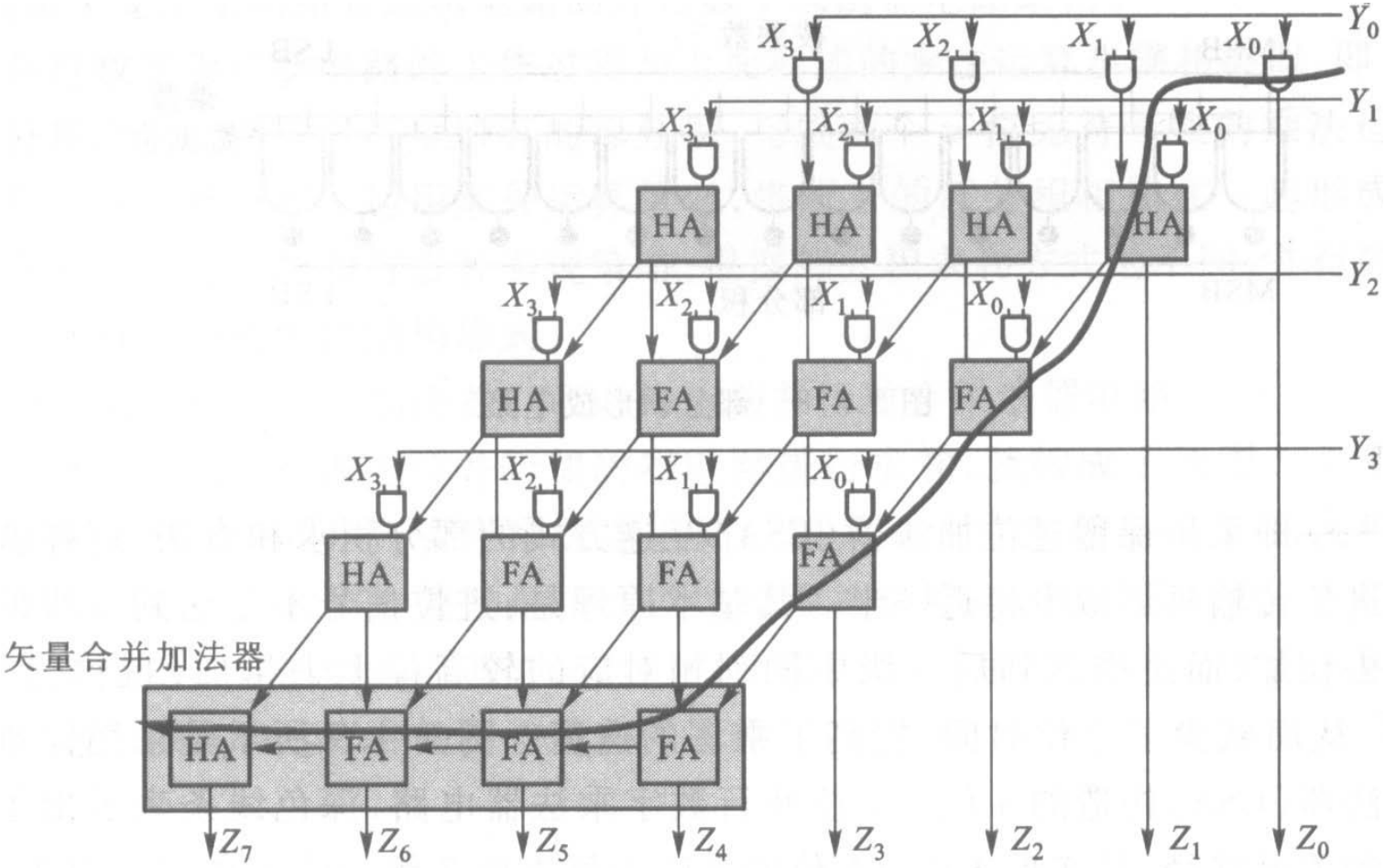
- 分类
  - 采用CPA构造的并行乘法器
  - 采用CSA构造的并行乘法器
  - 华莱士树形结构乘法器

采用CPA构造的并行乘法器



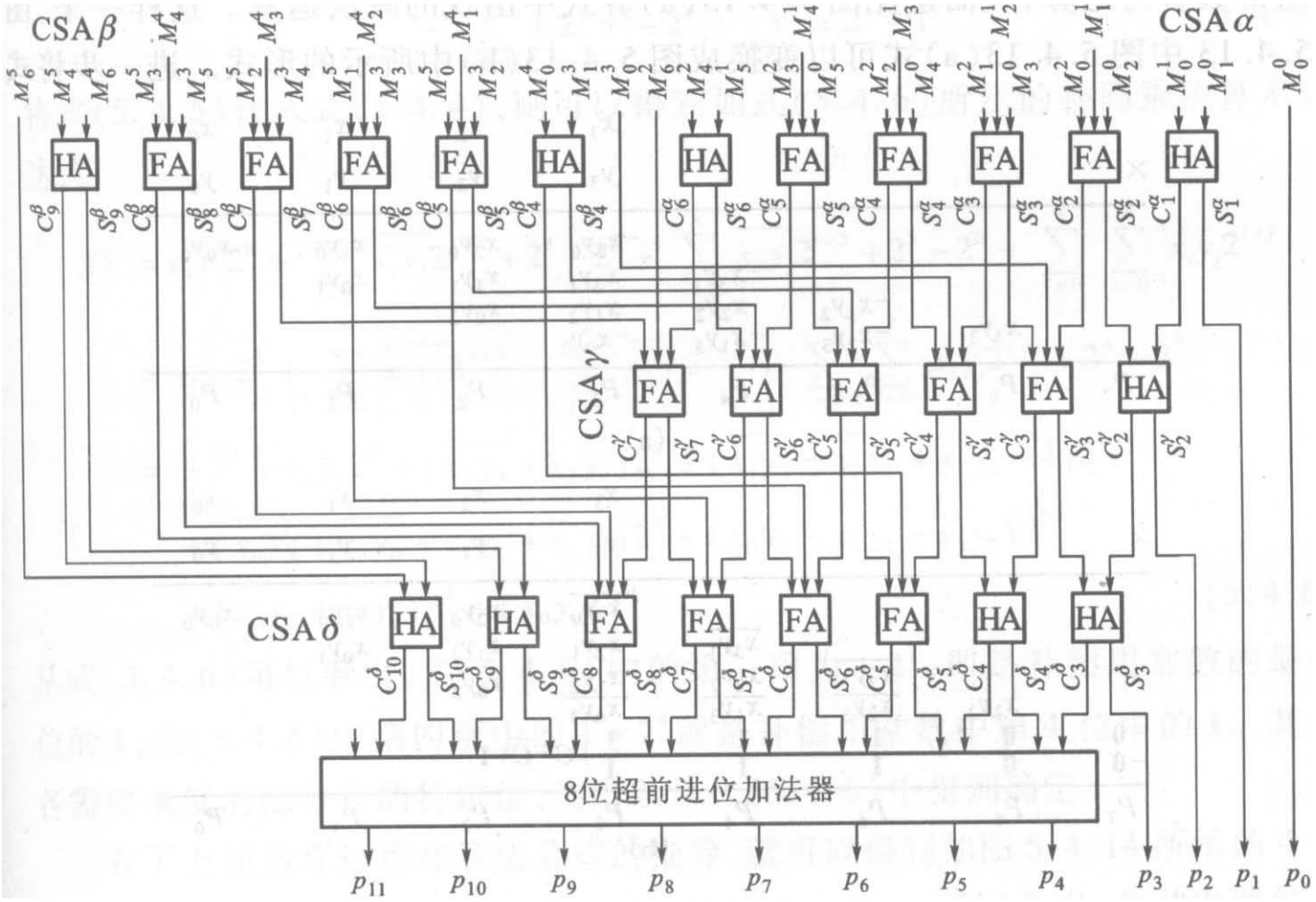
采用CPA构造的4位×4位并行乘法器电路

采用CSA构造的并行乘法器



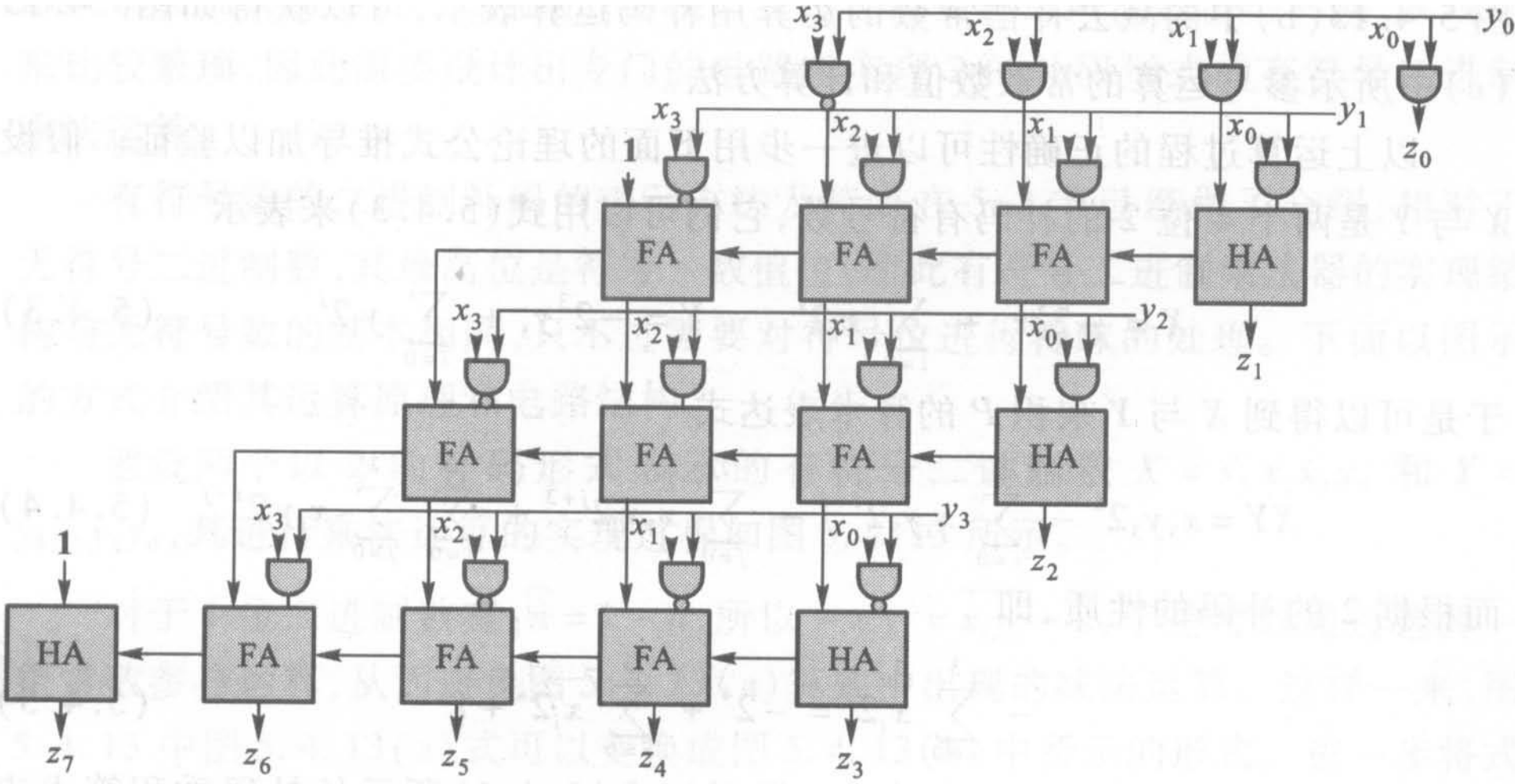
采用CSA构造的4位×4位并行乘法器电路

华莱士树(Wallace tree)形结构乘法器



华莱士树形乘法器的电路结构图

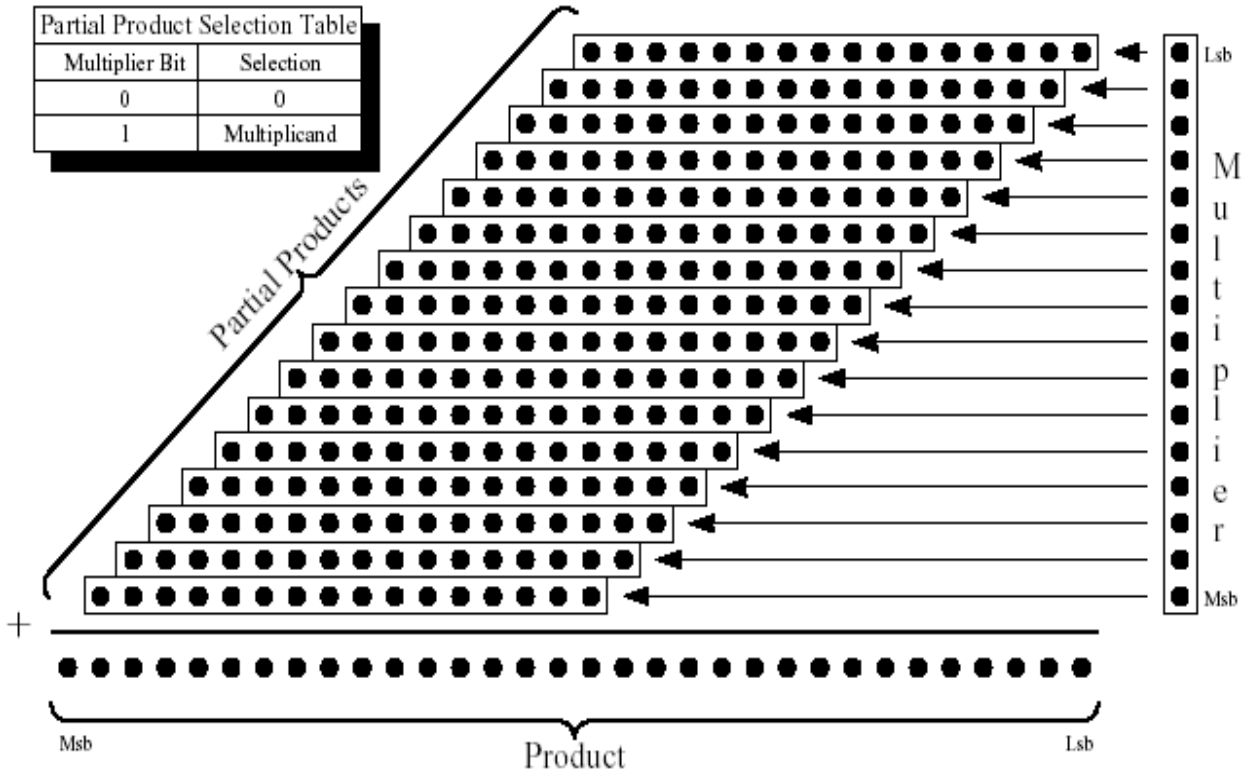
有符号二进制数乘法器



4位×4位二进制的补码乘法运算电路结构图

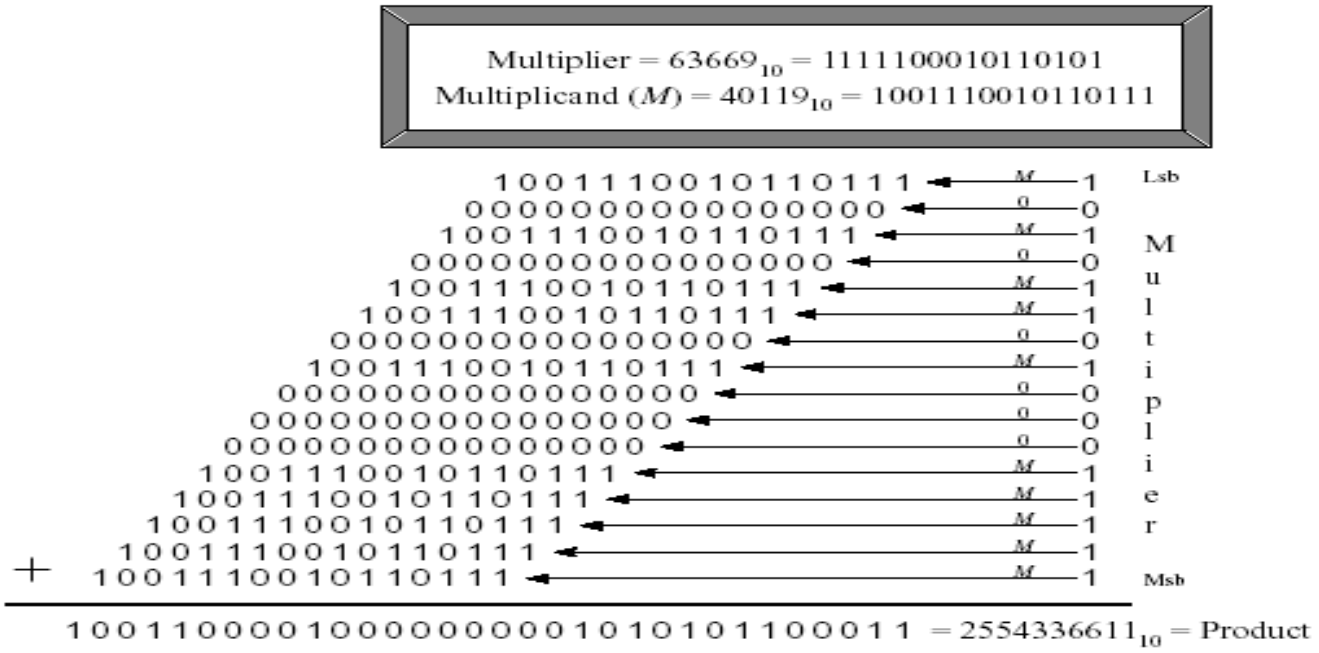
部分积的产生方法

乘法运算中的第一步就是以一定的算法产生部分积。最为简单产生部分积的方法可以用下面的点图说明

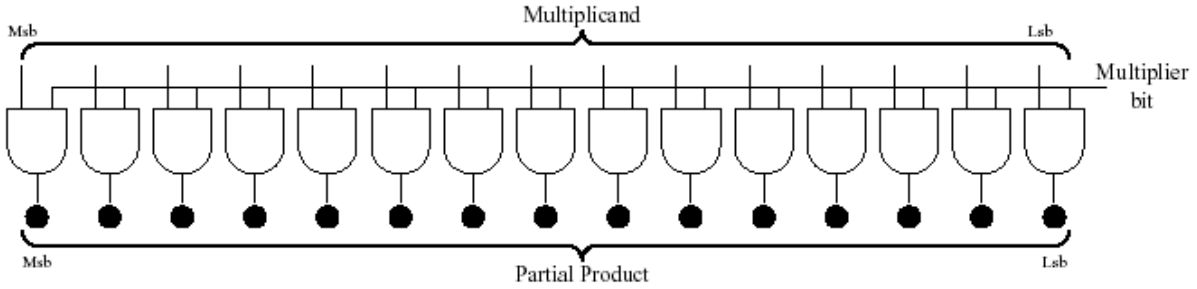


部分积的产生方法

- 部分积产生的实例



- 部分积产生电路



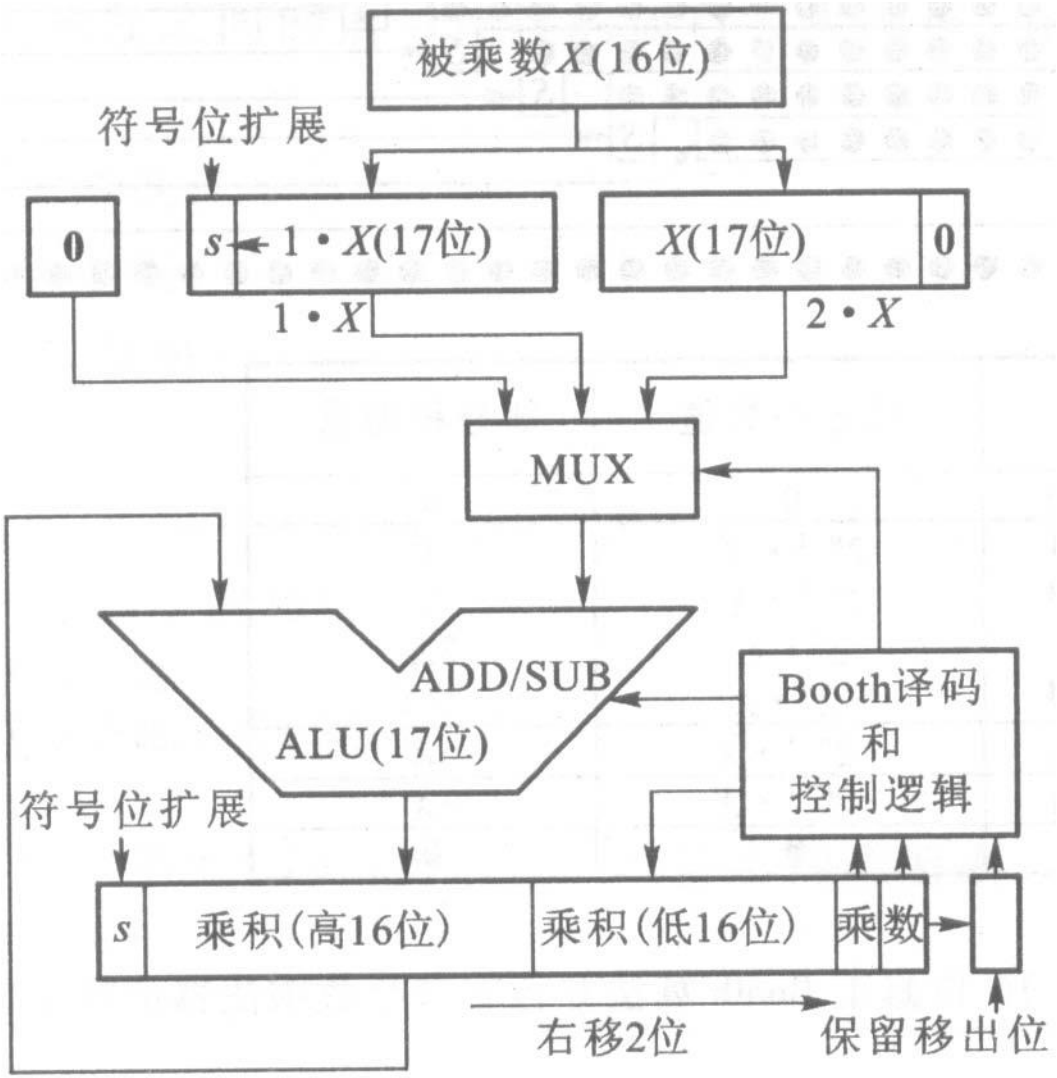
Booth算法

对乘数中的每一位，都要产生与其对应的部分积。而修正Booth算法按照乘数每2位的取值情况，一次求出对应于该2位的部分积，以此来减少部分积的个数。在运算中，每2位乘数有四种可能的组合，每种组合所对应的操作如下：

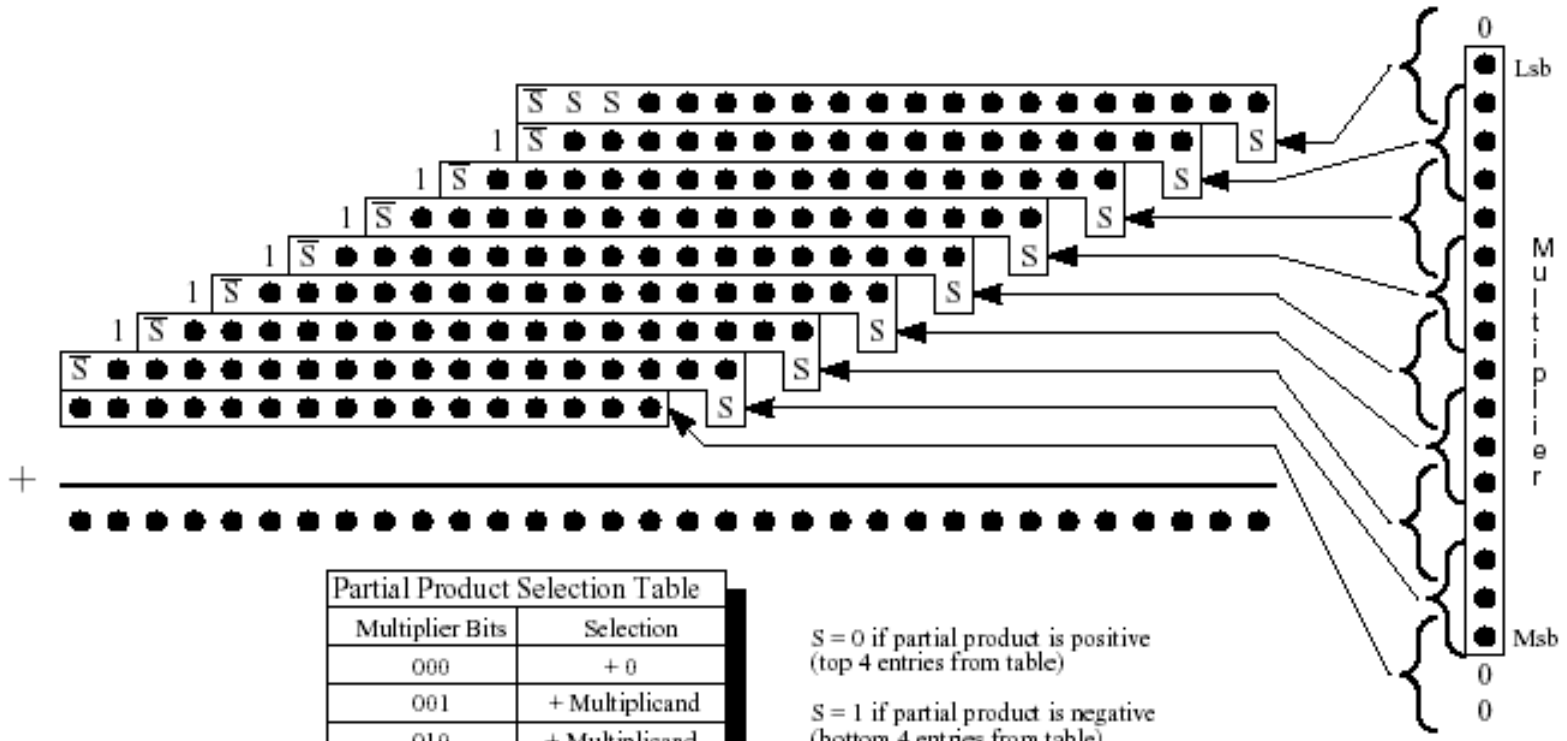
- **00 — 部分积相当于0•M，同时左移2位；**
- **01 — 部分积相当于1•M，同时左移2位；**
- **10 — 部分积相当于2•M(被乘数左移1位后即可获得)，同时左移2位；**
- **11 — 部分积相当于3•M，同时左移2位；**

修正Booth算法

- 原理
- 电路结构



修正Booth算法



Partial Product Selection Table

Multiplier Bits	Selection
000	+ 0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	-2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	- 0

修正Booth算法并行有符号数乘法器的点图

5.5 片上系统(SoC)的设计

SoC包括微处理器/数字信号处理器模块、存储器模块、数字电路/模拟电路子系统、接口电路/人机交互子系统等。

片上系统(SoC)的设计

- SoC的特性

1. 实现复杂系统功能的VLSI
2. 采用超深亚微米工艺技术制造
3. 内部使用一个或数个嵌入式CPU或DSP
4. 具备外部对芯片进行编程控制的能力
5. 主要采用第三方提供的IP核进行设计

- SoC的优点

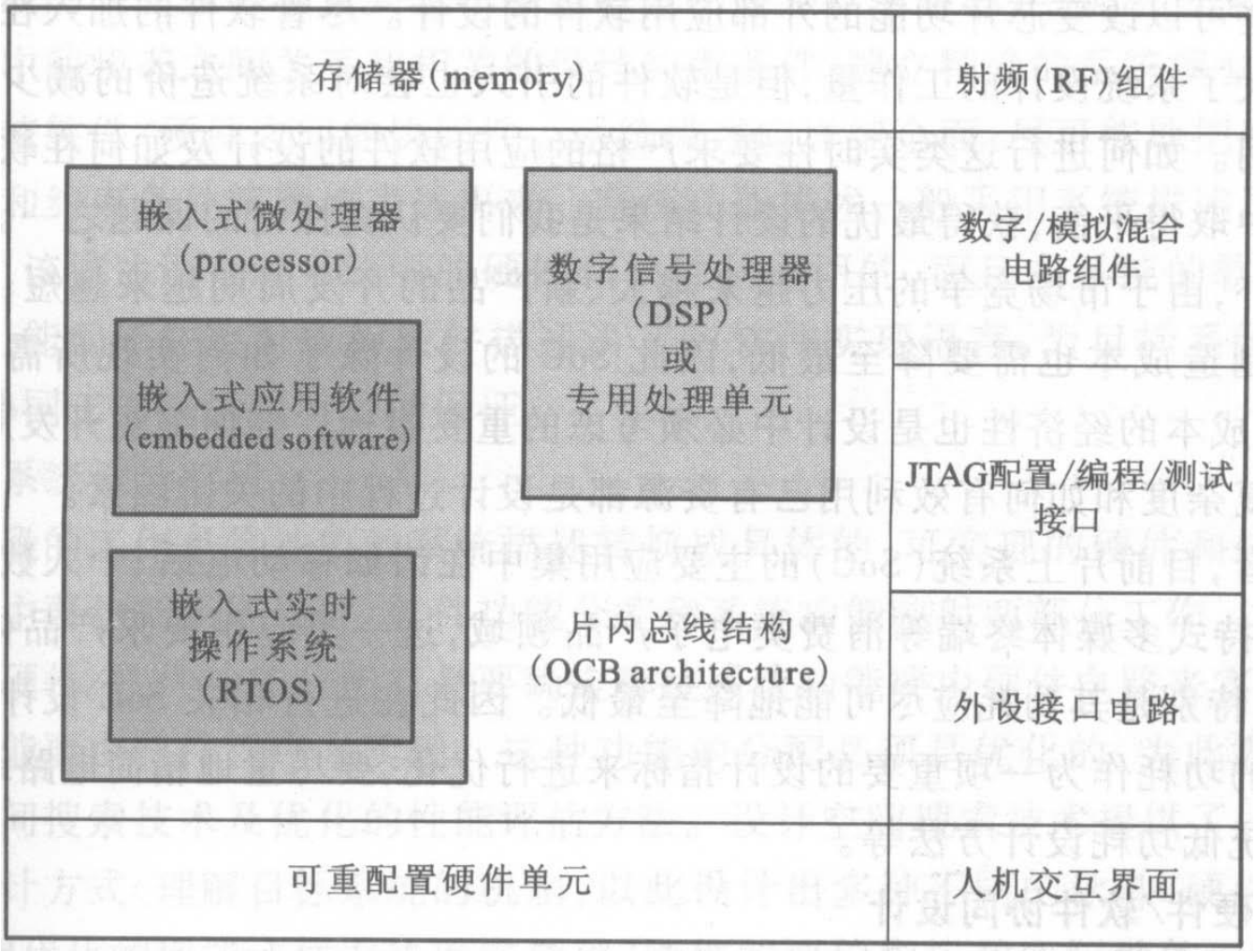
1. 成本大大降低
2. 执行效能增强
3. 体积小、功耗低
4. 可靠性增强

片上系统(SoC)的设计

- **JTAG**—Joint Test Action Group
- **RTOS**—Real Time Operating System
- **VSIA**—Virtual Socket Interface Alliance
- **OCP-IP**—Open Core Protocol International Partnership
- **SPIRIT**—Structure for Packaging, Integrating and Reusing IP within Tool-flows
- **APB**—Advanced Peripheral Bus

片上系统(SoC)的结构形式

- 片上系统的基本组成结构



片上系统(SoC)的设计方法

- 传统集成电路设计与SoC设计的区别

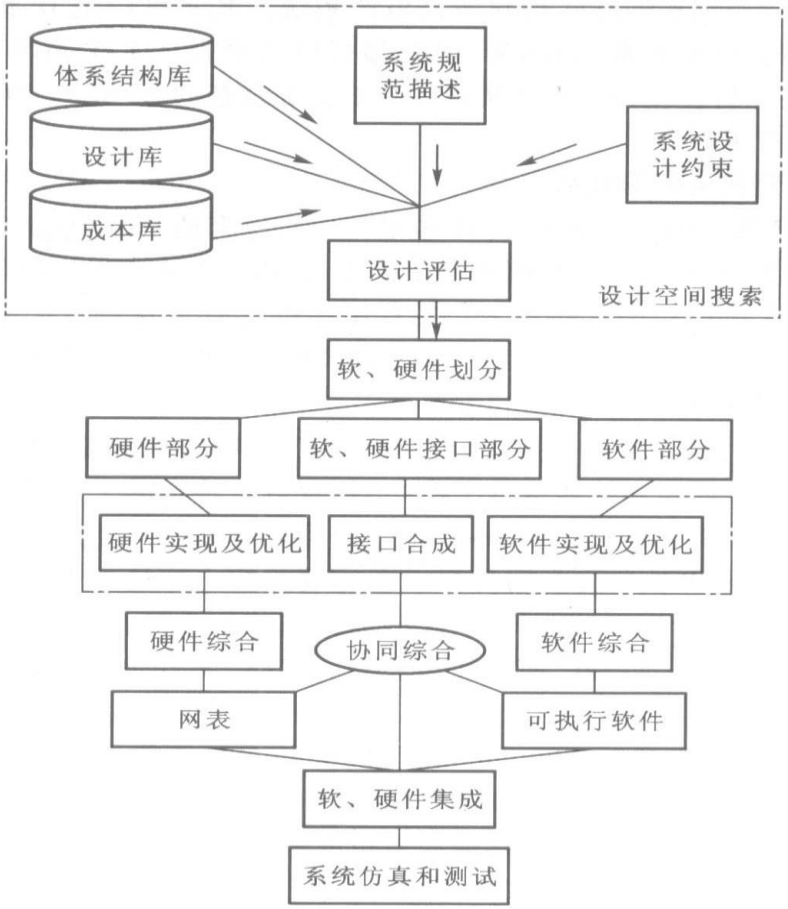
- SoC的设计方法

  - 硬件/软件协同设计(H/S Co-design)
  - 设计“复用”(re-use)技术
  - 可重配置设计技术

硬件/软件协同设计(H/S Co-design)

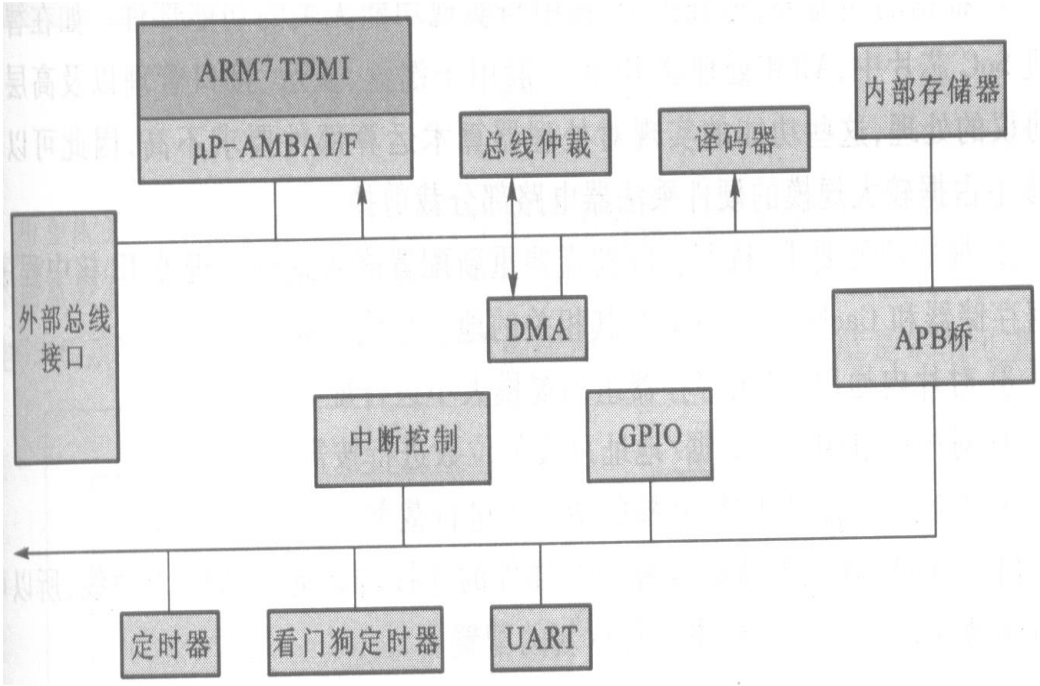
H/S Co-design是在系统设计目标的指导下，通过综合分析系统中H/S的功能及现有资源，最大限度挖掘系统中H/S工作中的并发性，协调一致地设计系统中H/S的体系结构，以达到系统中的H/S均工作于最佳状态的目标。

- 系统功能描述阶段
- 系统设计阶段
- 系统评价阶段
- 系统综合实现阶段



设计“复用”(re-use)技术

- 设计“复用”的根本是使用已有的IP Core。
- IP Core的分类
  - 软IP核(Soft IP core)
  - 硬IP核(Hard IP core)
  - 固IP核(Firm IP core)
- IP Core的“复用”
  1. 再使用标准的选择
  2. 复用资源的提供与获取



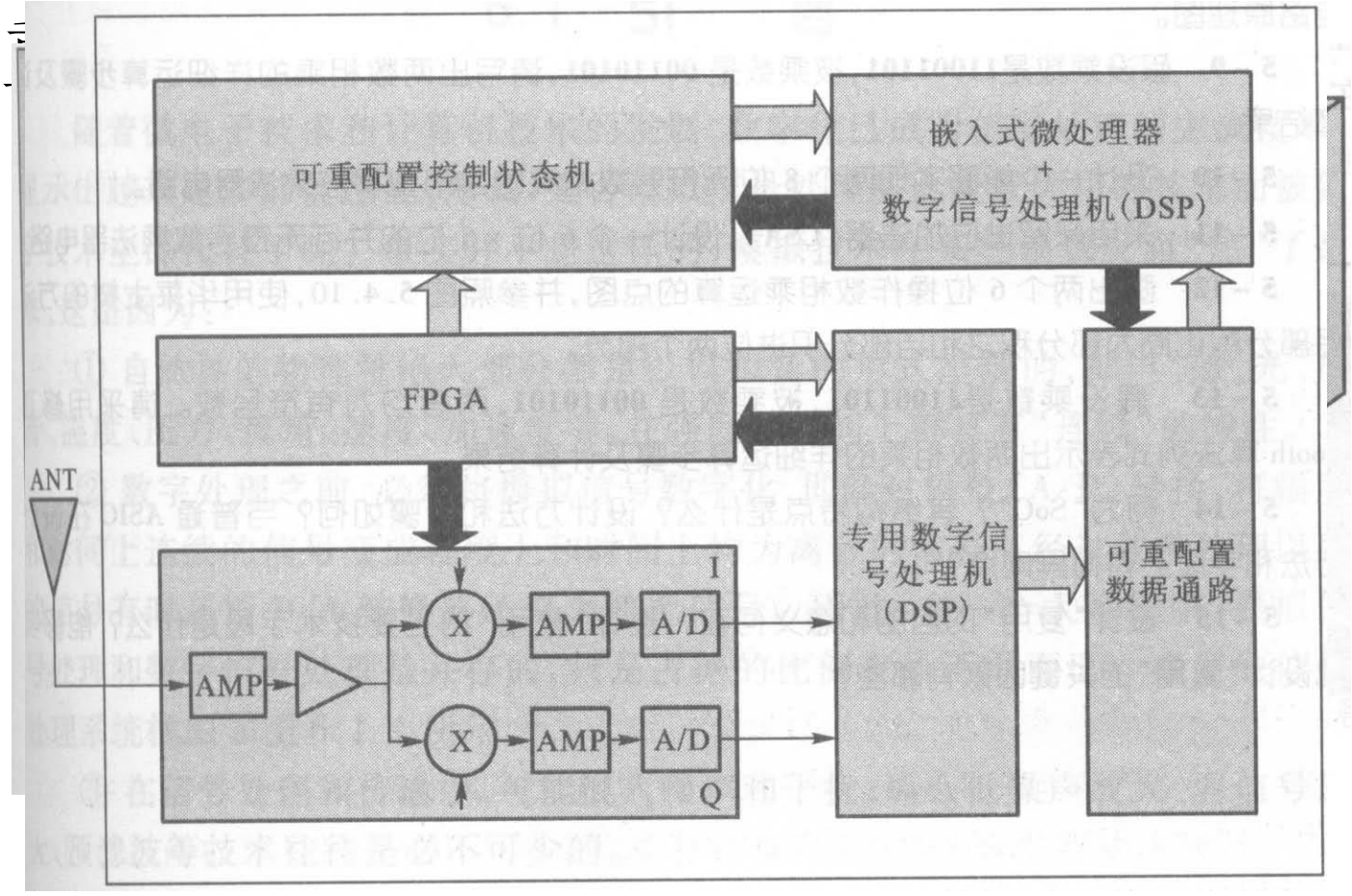
- 用于SoC设计IP核的实例

IBM－Power PC、MIPS	ARM－ARM	Intel－Strong ARM、XScale

可重配置设计技术

对所使用的IP核进行重新配置或“剪裁”，以达到所设计的电路为相关功能“量身定做”。

- 可



西安電子科技大學

第七章

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.
- 董磊等主编,数字电路的FPGA设计与实现—基于Xilinx和VHDL, 北京: 电子工业出版社, 2022.
- 高亚军编著, Vivado/Tcl零基础入门与案例实战, 北京: 电子工业出版社, 2021.

7.1 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。

7.2 Verilog的基本语法规则

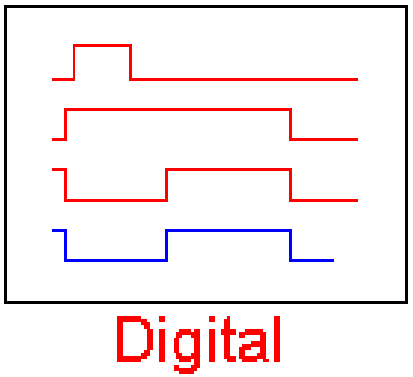
Verilog HDL的历史

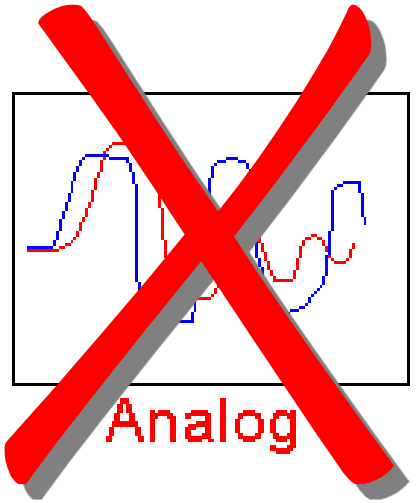
- Verilog HDL是在1983年由GDA(GateWay Design Automation)公司的Phil Moorby所创。Phil Moorby后来成为Verilog-XL的主要设计者和Cadence公司的第一个合伙人；
- 在1984~1985年间，Moorby设计出了第一个Verilog-XL的仿真器；
- 1987年，Synopsys公司的综合软件开始接受Verilog输入；
- 1989年，Cadence公司收购GDA，进一步扩大Verilog的影响；
- 1991年，Cadence公司公开发表Verilog语言，成立了OVI(Open Verilog International)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)

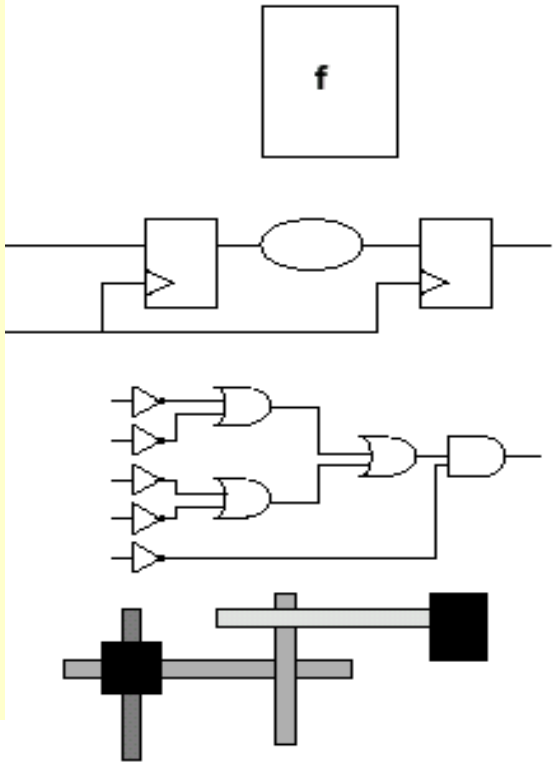
- Verilog既是一种行为描述的语言也是一种结构描述语言。Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别包括：

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)

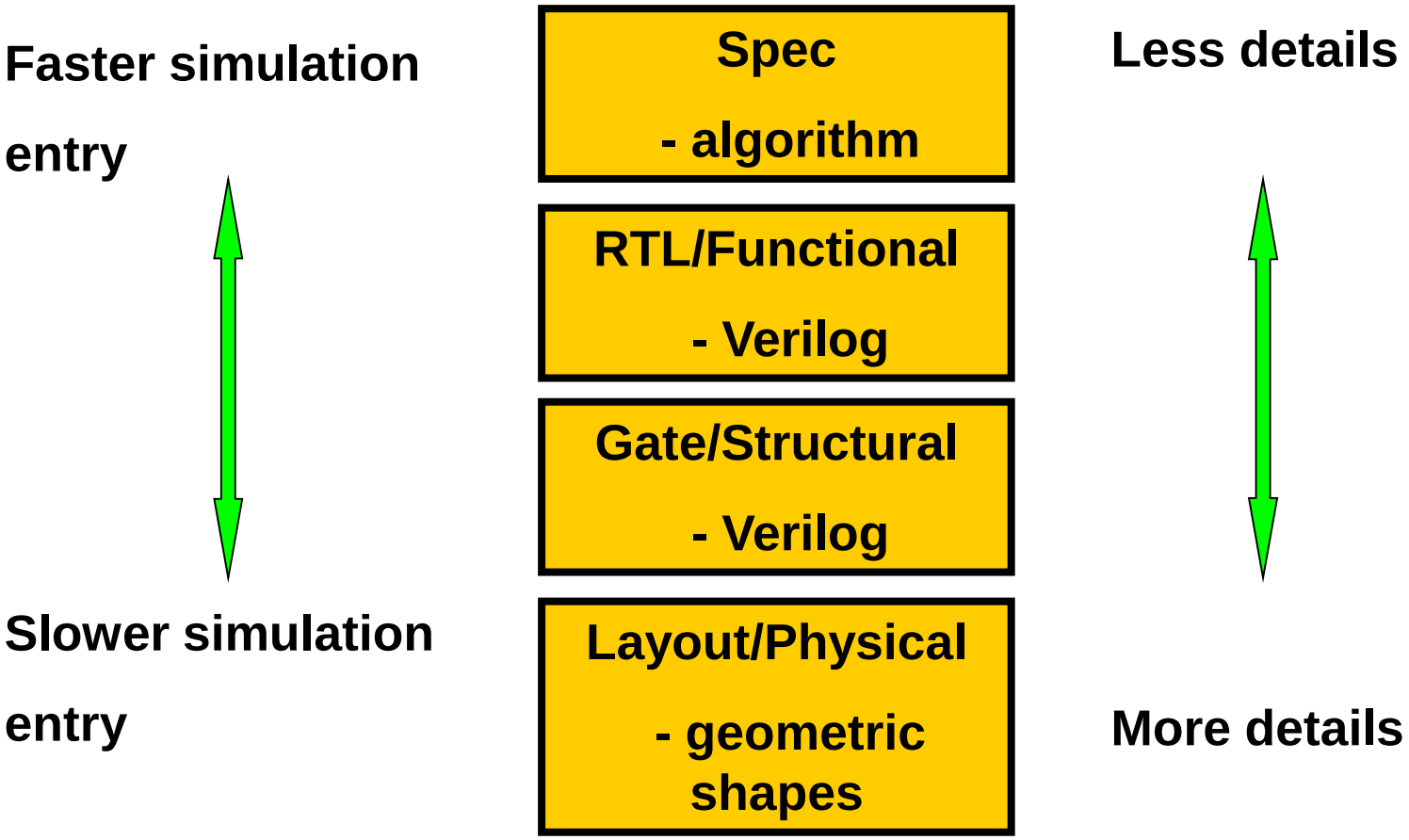
- 用功能块内部或功能块之间的数据流和控制信号描述系统
- 基于一个已定义的时钟的周期来定义系统模型

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)

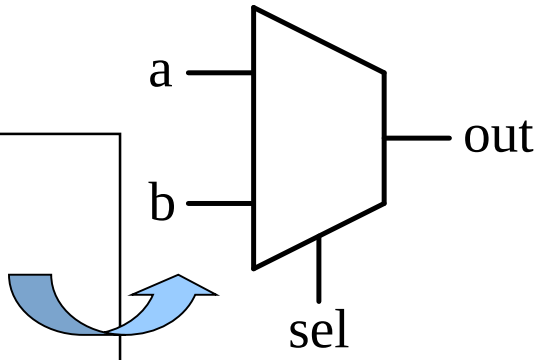
- 各级描述风格之间的Trade-offs

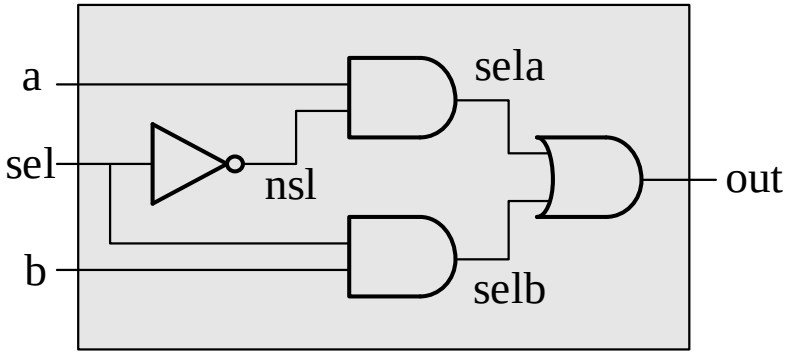


抽象级(Levels of Abstraction)

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

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

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

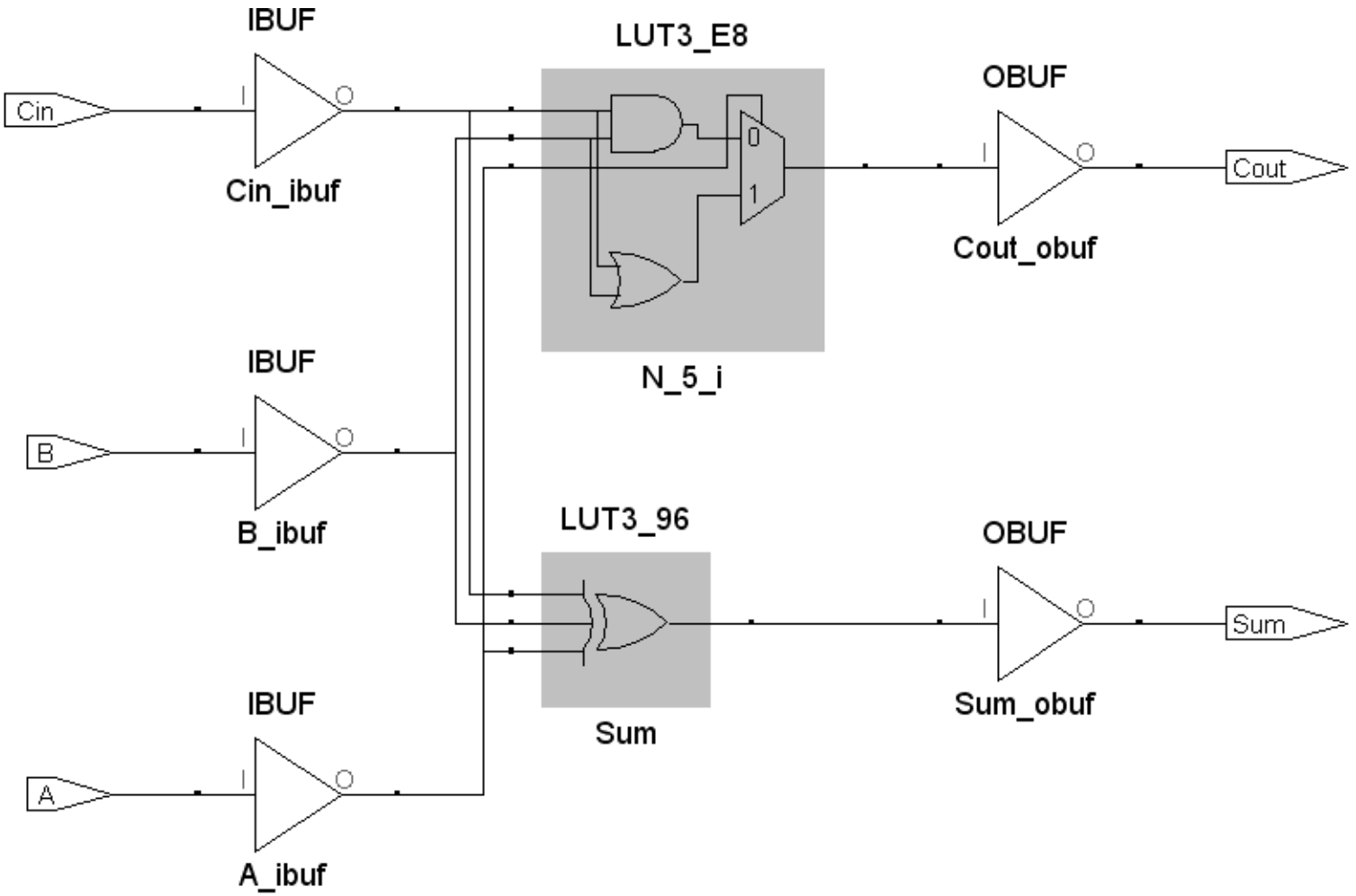
抽象级(Levels of Abstraction)

- 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

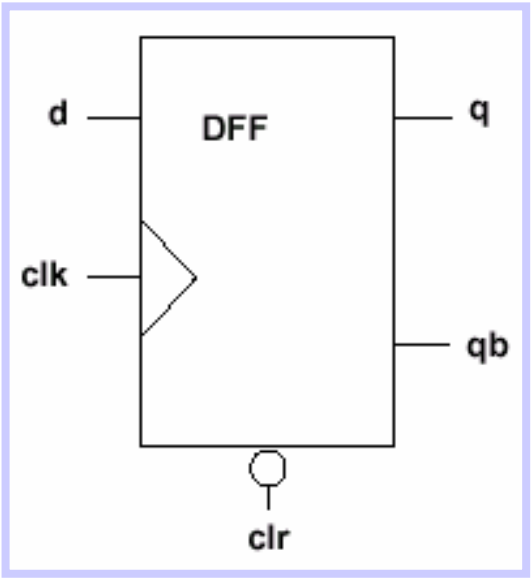
端口及内部信号
信号类型：
net
register
parameter

```
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

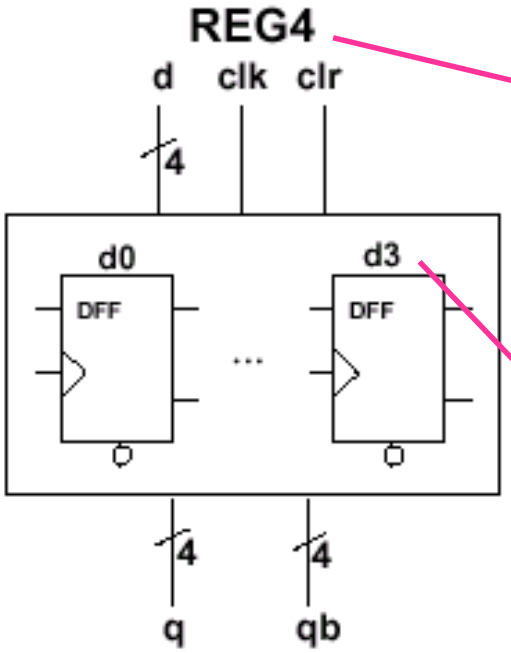
端口可以说明为*input, output*及*inout*

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

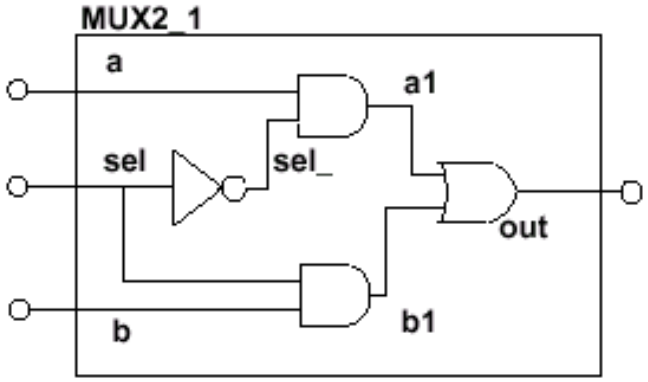
`define <macro_name> <macro_text>

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)

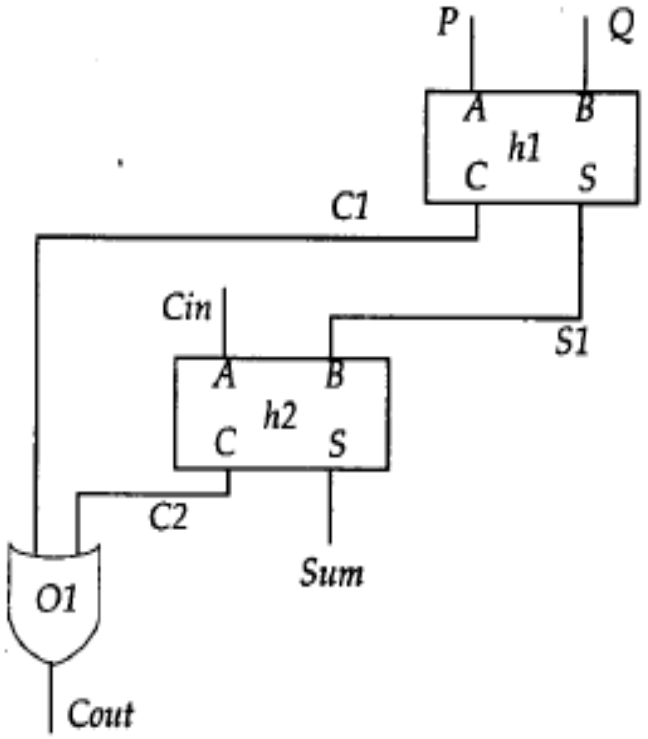
```
Micro M1 (UdIn[3:0], {WrN, RdN}, Status[0], Status[1] ,
                &UdOut[0:7], TxData ) ;
```

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
```



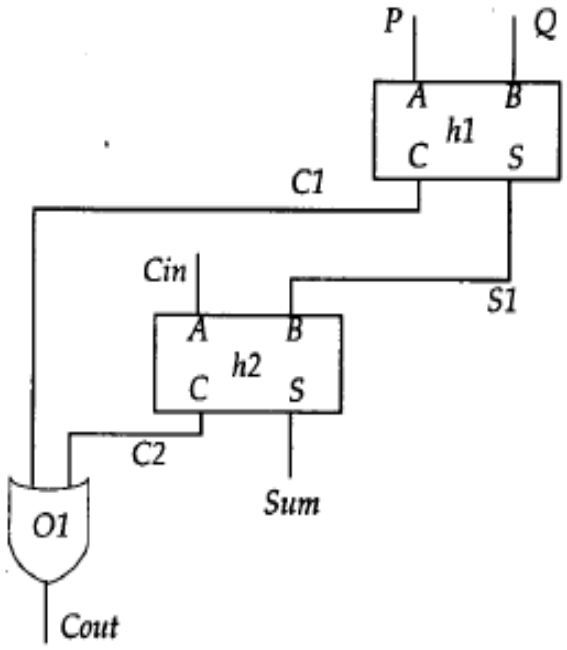
考虑如何模块参数化？

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

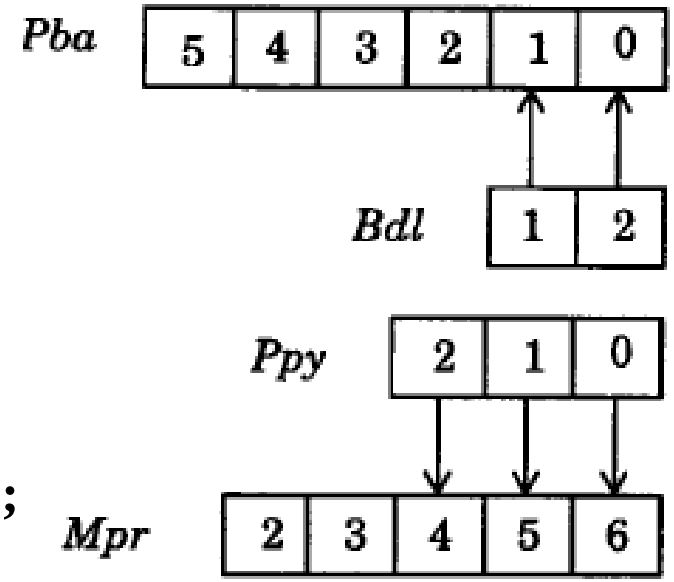
- 通过将端口表达式表示为空白来指定为悬空端口

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

端口长度不同

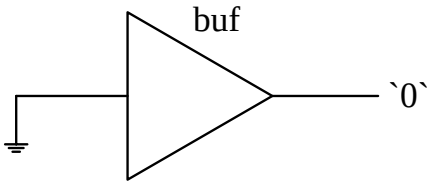
- 通过无符号数的右对齐或截断方式进行匹配

```
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
```

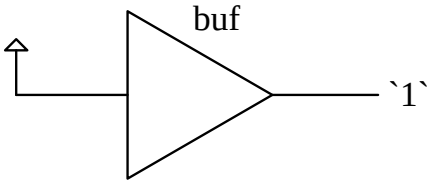


4值逻辑

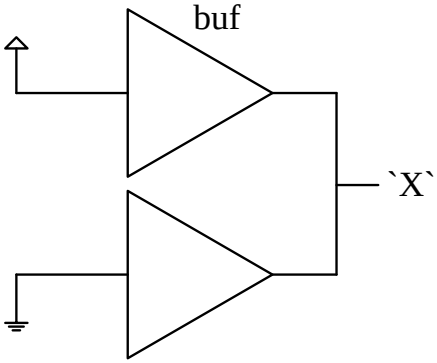
Verilog采用4值逻辑 0，1，X，Z



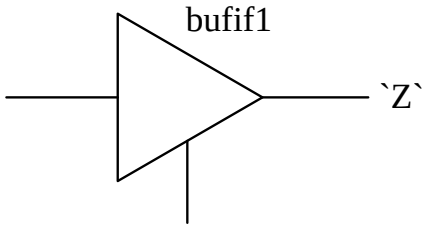
‘0’表示Zero，Low，Logic Low，False，Ground，VSS，Negative Assertion



‘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

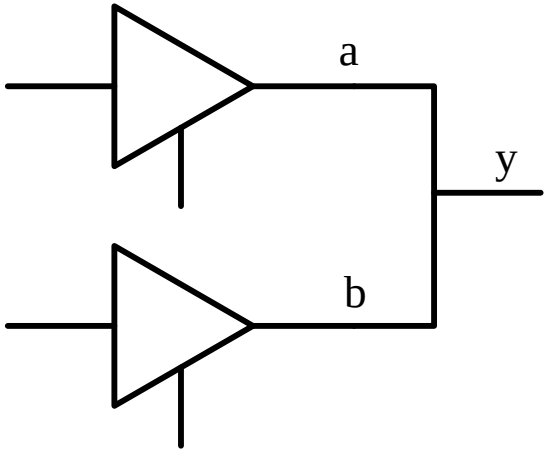
- 有多种net类型用于设计(design-specific)建模和工艺(technology-specific)建模

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

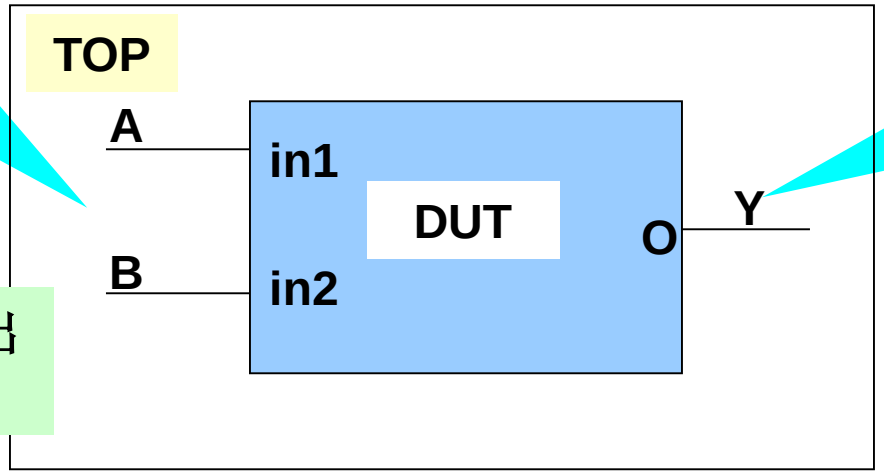
- 举例：

```
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
```

```
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

在过程块中只能给register类型赋值

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

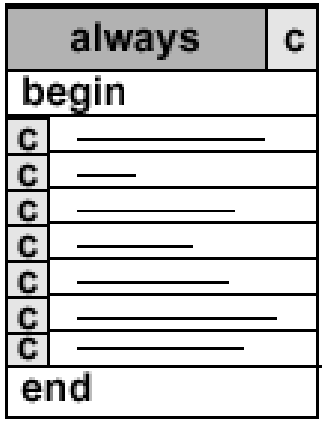
- FPGA设计中的时延仅在功能仿真时有效，不影响实际电路生成。

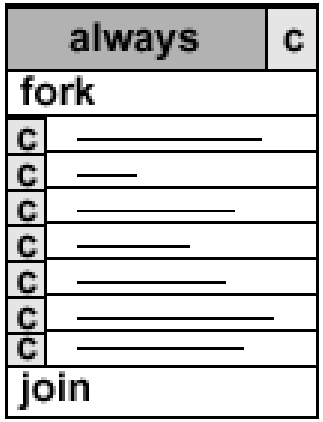
持续赋值： 数据流建模

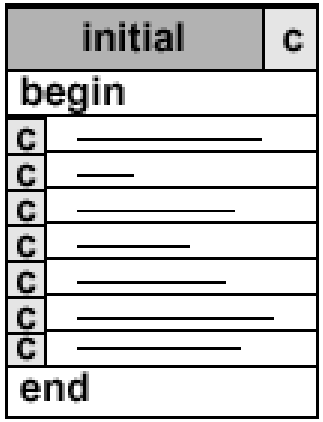
- 数据流建模注意事项：
  - 1）wire型变量如果不赋值，默认值为z；
  - 2）数据流建模没有存储功能，不能保存数据；
  - 3）wire型变量只能在声明时赋值或者assign语句赋值；
  - 4）assign语句并发执行，实际的延迟由物理芯片的布线结果决定。
  - 5）最基本的FPGA设计源代码描述语句之一，用于生成组合逻辑，定制LUT的逻辑功能。常作为中间信号的描述用于控制寄存器的输入输出。

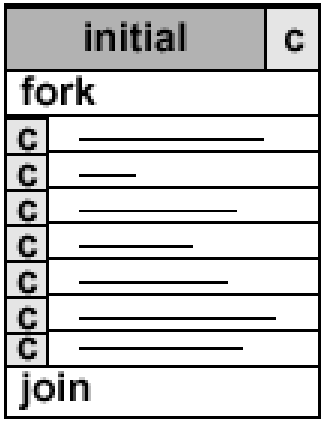
块语句

- 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









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
```

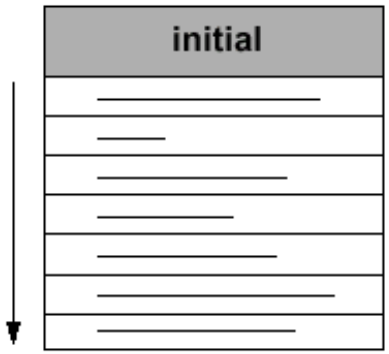
上面2个例子在功能上是等价的

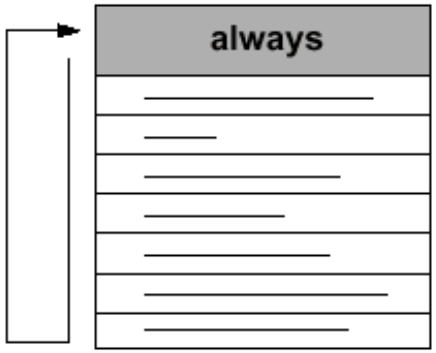
但是fork/join模块综合器不支持！！

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:

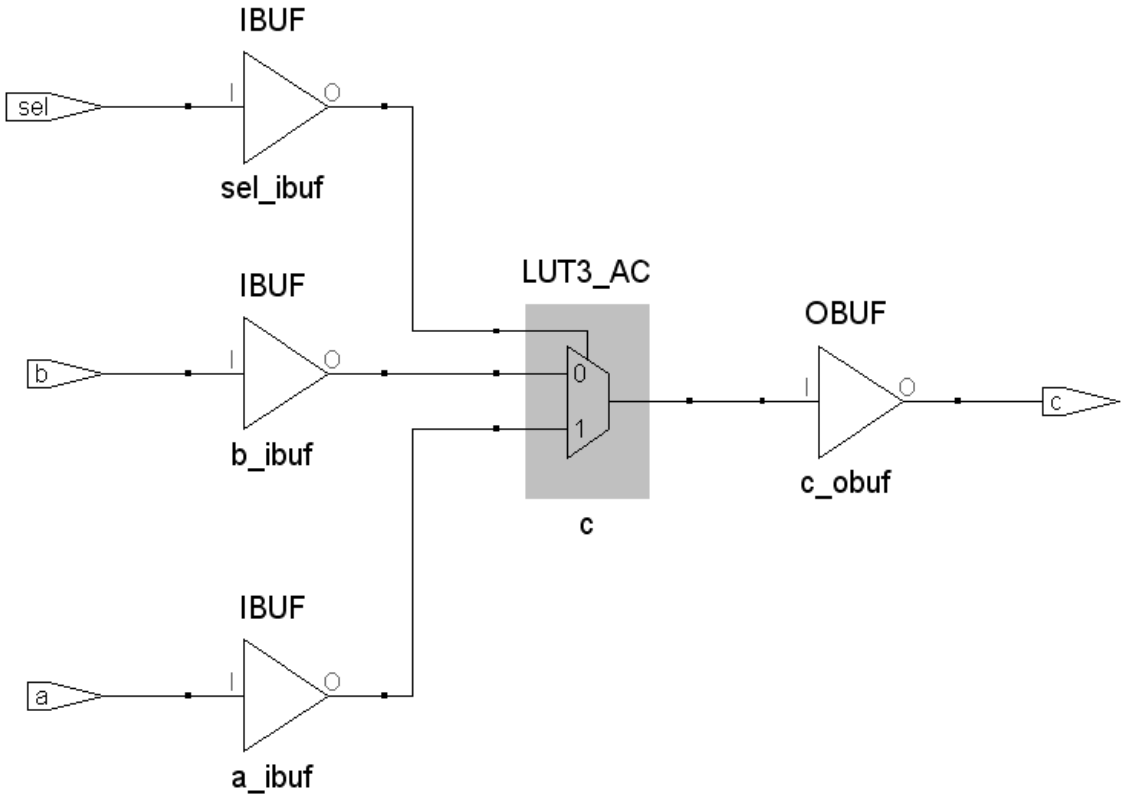
(1)虽然c是reg型，但综合的结果是组合电路；

(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

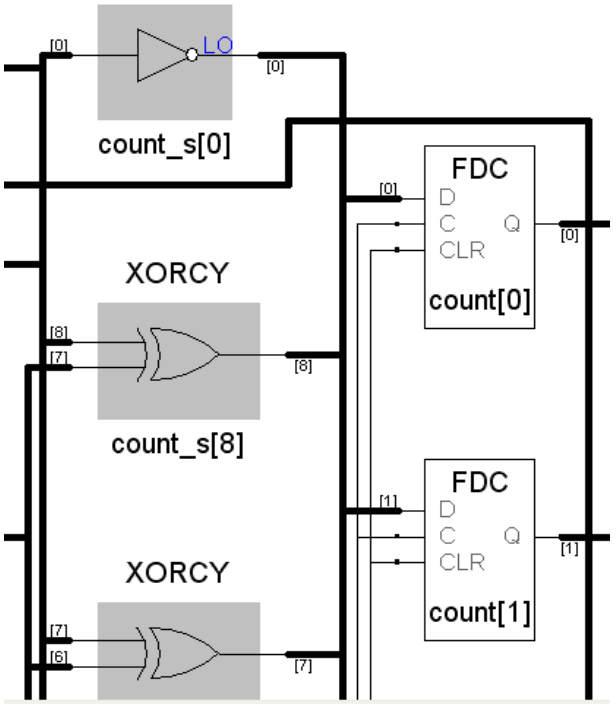
此外，容易产生锁存器



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
```

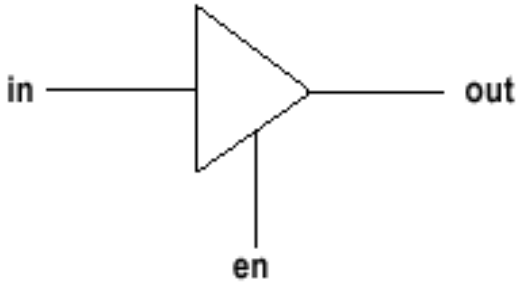
先补后移

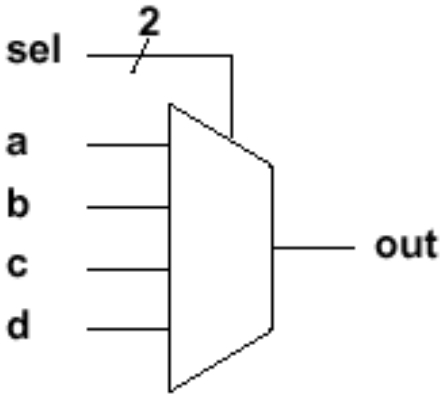
先移后截

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
```

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。

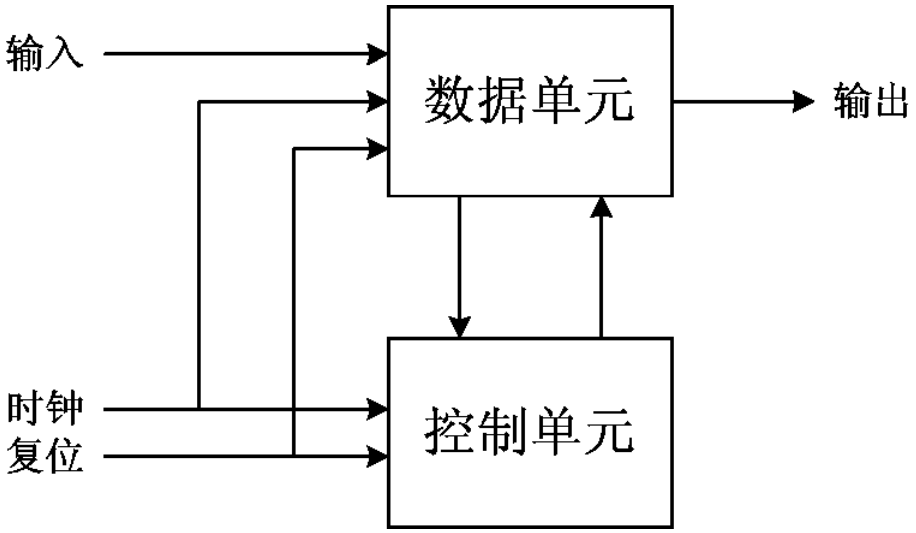
时序逻辑电路设计

- 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。

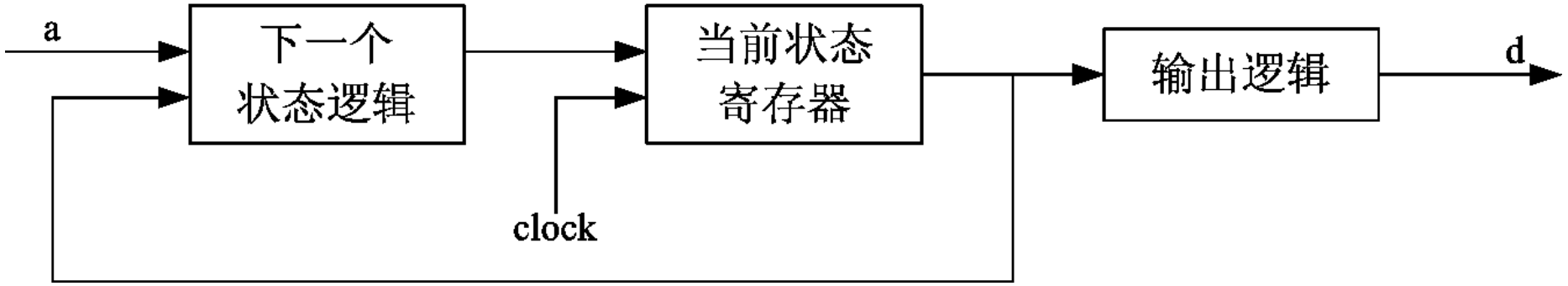
有限状态机的设计

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

```
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

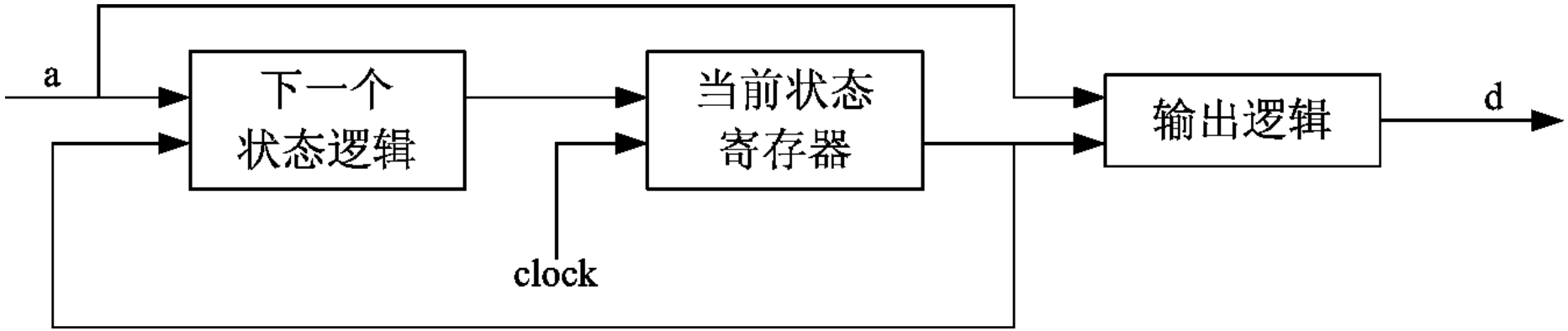
```
        st2 :
          casex(data_in)
           2'b0x   : next_state = st1;
           2'b1x   : next_state = st3;
           default : next_state = st0;
          endcase
        st3 :
          casex(data_in)
           2'bx1   : next_state = st4;
           default : next_state = st3;
     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

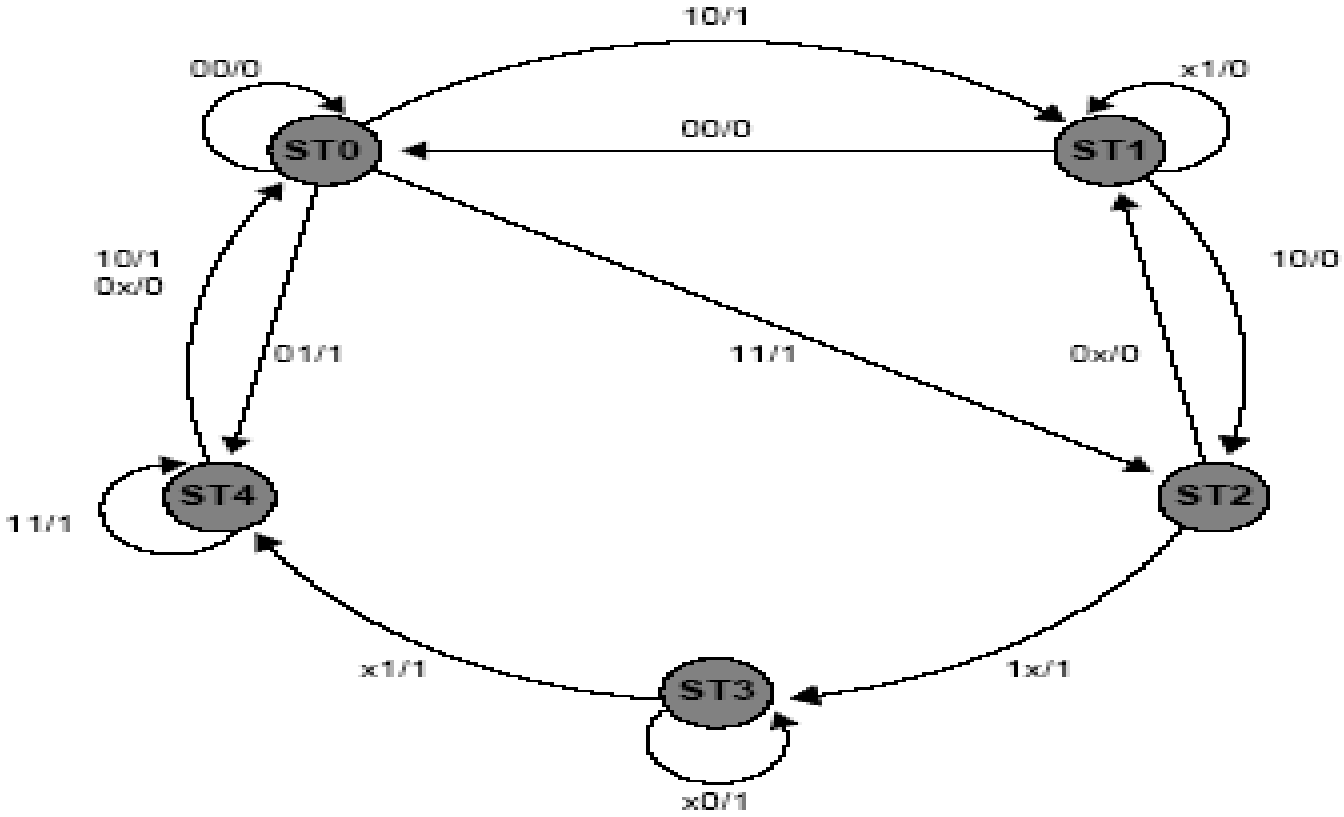
Mealy状态机的设计

- 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

- 5个状态的状态机的状态转移图



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

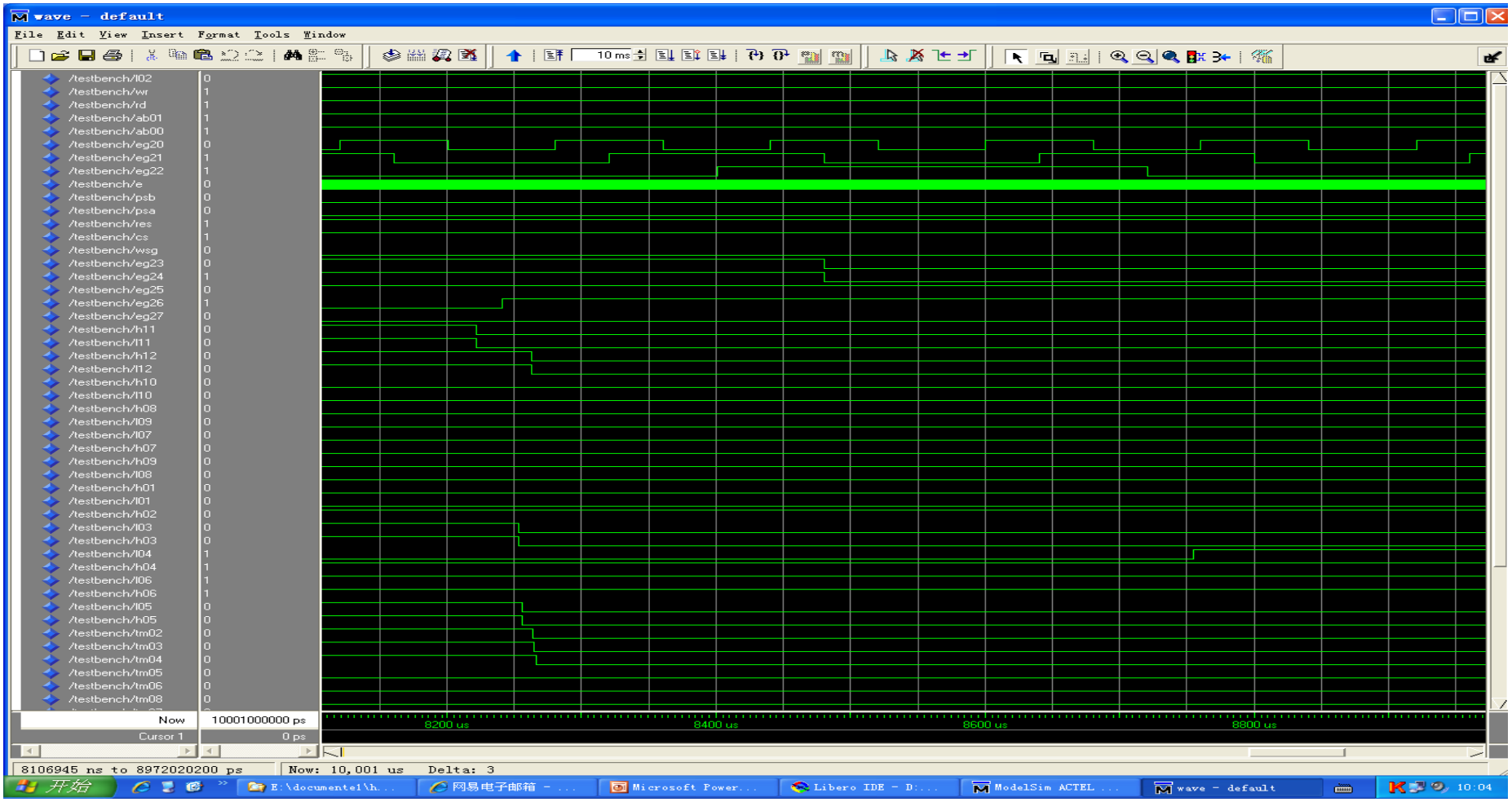
1. 仿真器(simulator)
2. 激励信号产生工具
3. 波形观察器

- 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
3. 文件读写

- 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

基于ModelSim的功能仿真课程实践

从实验一、二、三中选择一题完成

课程结束前将所有工程文件及

报告电子版合并成压缩包上交

仿真实验要求

- 实验一(难度C)：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

- 实验二(难度B)：以UART方式实现参数寄存器的配置与反馈
  - 5MHz时钟，9600波特率，参数配置协议如下

同步头7E7E(固定2字节)	参数标记(1字节)	参数信息(1字节)

  - 参数配置：参数标记范围为00~07，参数信息自设
  - 参数反馈：参数标记为88，将参数寄存器00~07中的信息传送出来
  - 设计出参数配置和参数反馈这两种工作模式，并通过ModelSim做出对应的波形

仿真实验要求

- 实验三(难度A)：图像插值功能实现
  - 设计图像插值功能，插值功能函数可自定义
  - 分别设计出文件读入原始图像数据并将插值后图像数据写入文件
  - 原始图像数据按照光栅格式输入，插值后图像数据也按照光栅格式输出，并通过ModelSim做出对应的输入/输出波形

西安電子科技大學

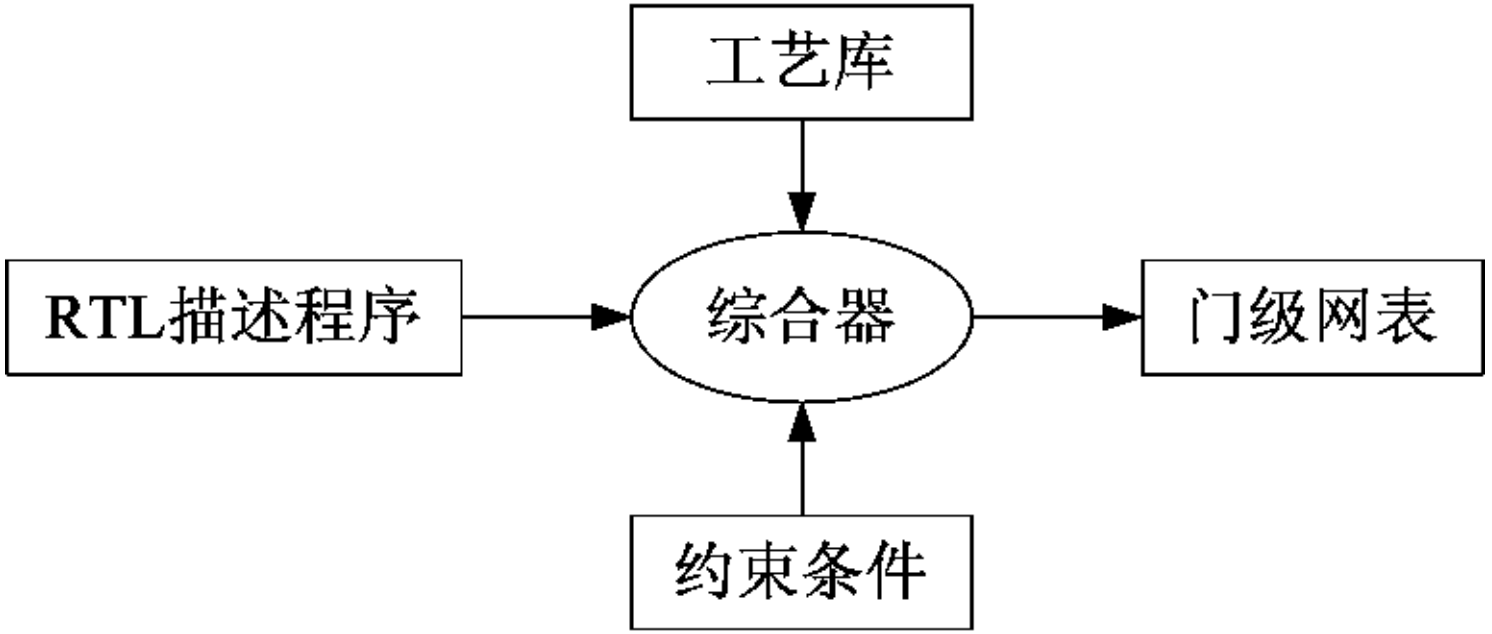
7.5 Verilog综合

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

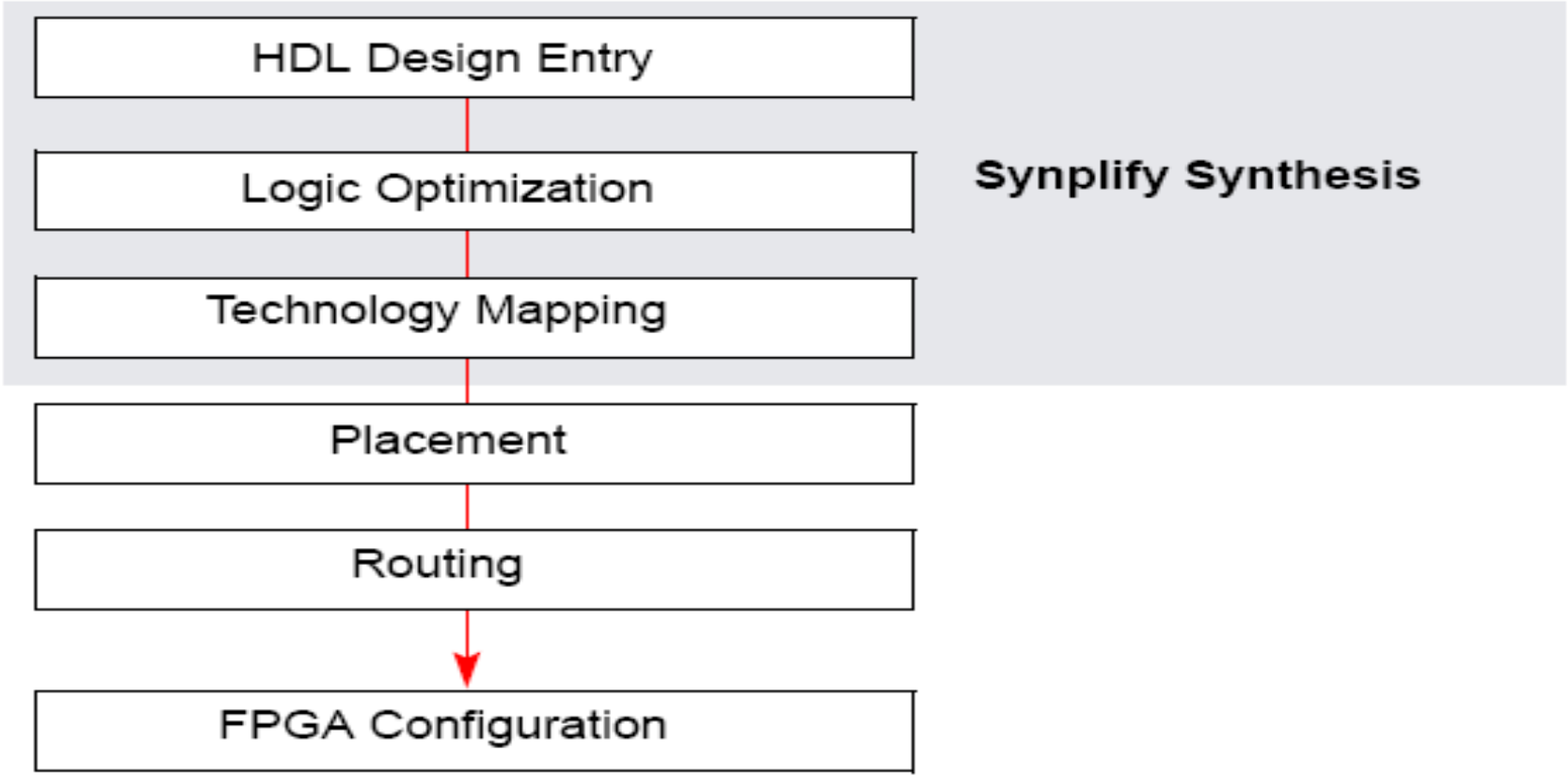
- 一般逻辑综合的过程如下图所示



- 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。

西安电子科技大学

第八章

数字集成电路的测试与可测性设计

数字集成电路的测试与可测性设计

- 概述
- 批量生产的测试方法
- 测试的基本概念
- 测试码的生成
- 故障模拟
- 可测性设计法

8.1 概述

仿真

- 仿真或称模拟，是在电路系统设计过程中用来对设计者的硬件描述和设计结果进行查错、验证的工具。
- 所谓仿真是指从电路的描述(语言描述或图形描述)抽象出模型，然后将外部激励信号或数据施加于此模型，通过观察该模型在外部激励信号作用下的反应来判断该电路系统是否实现预期的功能。

仿真

- 仿真工具(simulator)的种类：

1. **电路级仿真**：电路仿真的对象是用晶体管及电阻、电容等组成的电路网络。电路模型是组容等效电路。仿真的方法就是用解方程法解由组容等效电路对应的电路方程。表示电路信号的数据是电压和电流。SPICE是典型的仿真工具。
2. **逻辑仿真**：逻辑仿真的对象是以门和功能块为描述电路的元件，也称门和功能块级仿真。仿真的目的就是检查电路是否具有规定的功能，包括逻辑功能、延迟特性以及负载特性等。仿真的方法一般是在电路的外部输入端加入激励信号，通过信号沿着元件和线网向输出传播，在输出端上得到响应波形。通过观察和分析波形关系判断其功能和时序关系是否正确。
3. **开关级仿真**：开关级仿真介于电路级和逻辑级之间。它与电路级的相同之处是都用晶体管表示电路结构，但电阻和电容不作为电路元件而作为晶体管和节点的参数描述，对电路的描述有所简化。
4. **寄存器传输级仿真**：基本元件是寄存器、存储器、总线、运算单元等，并描述数据在这些元件中流动的条件和过程。仿真通过控制数据和信号按照描述的条件和过程，来观察描述是否正确。这个级别主要通过数据在寄存器元件之间的流动来仿真系统的行为，也隐含表达了电路的大致结构。
5. **高层次仿真**：以行为算法和结构的混合描述为对象。高层次描述一般用硬件描述语言描述。主要着眼于系统功能和内部运行过程。其基本元素是操作和过程。各操作之间主要考虑其数据传输、时序配合、操作流程和状态转换。仿真时观察作为运行结果的输出数据及其时间配合关系或状态转移关系，来判断描述的正确性。

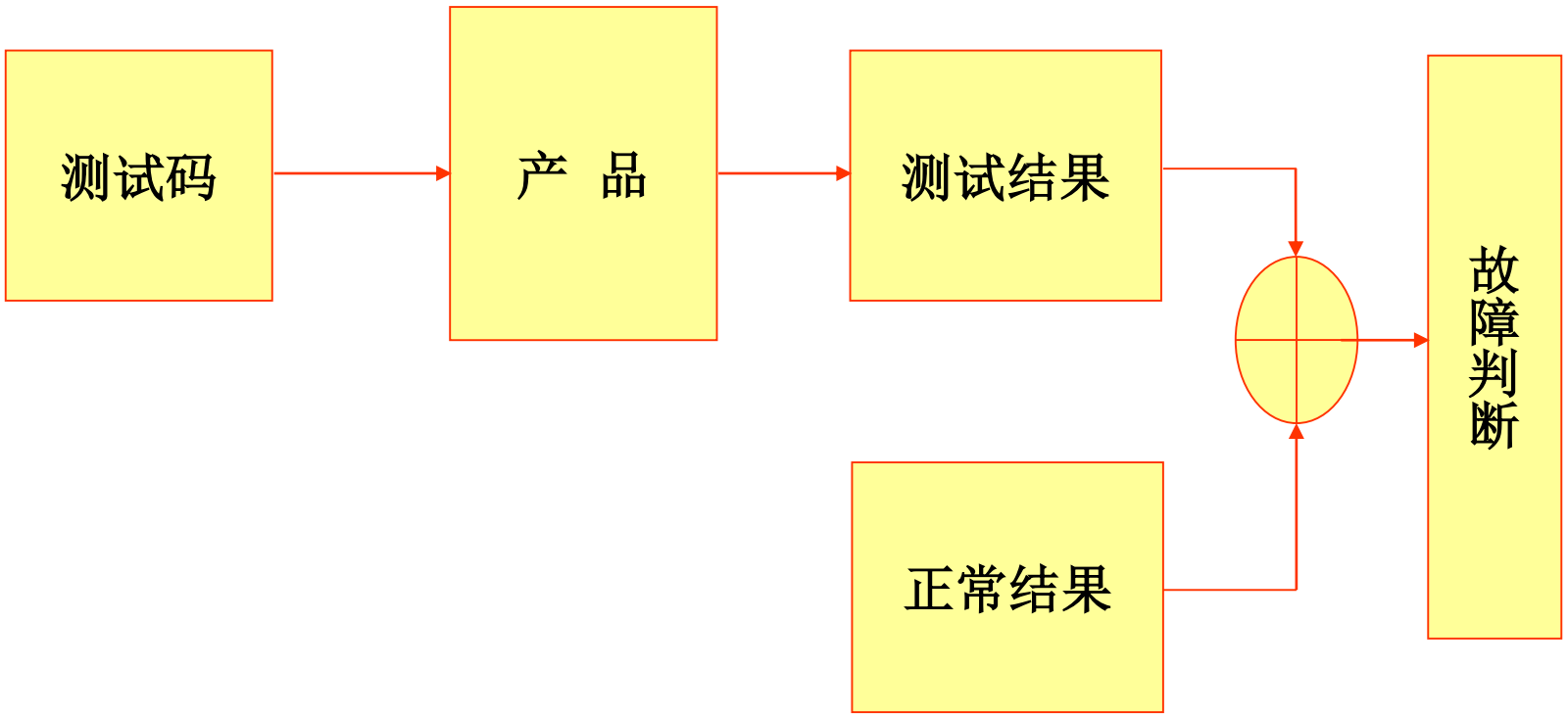
测试

测试与模拟不同。模拟是对设计过程中得到的电路数据验证其正确性的，是在产品未生产之前进行的；而测试是判断产品是否合格，是在电路生产之后进行的，是产品制造的最后一道工序。

- 测试的分类

1. 功能测试：目的在于验证设计是否能正确的按照技术条件实现其功能。
2. 制造后测试：或称为结构测试，目的在于检查生产的每一个芯片是否合格。

测试过程



可测性设计

测试码生成是根据电路结构和功能实现的。随着集成电路规模越来越大。测试码的生成变得越来越困难，甚至不可能找到测试码，所以人们逐渐把注意力集中到电路设计方面，使用某些电路结构可以使测试码容易得到，或者直接在电路内部增加测试机制，自动测试，自动判断是否存在故障。这种在设计过程中考虑可测性的设计方法称为可测性设计(DFT, Design For Test)。

故障检测与故障诊断

1. **故障检测**(fault detection)：判断故障是否存在，即只判断有无故障，称为故障检测。

2. **故障诊断**(fault diagnosis)：不仅判断是否存在故障，而且需要指出故障的位置，称为故障诊断。

西安电子科技大学

8.2 批量生产的测试方法

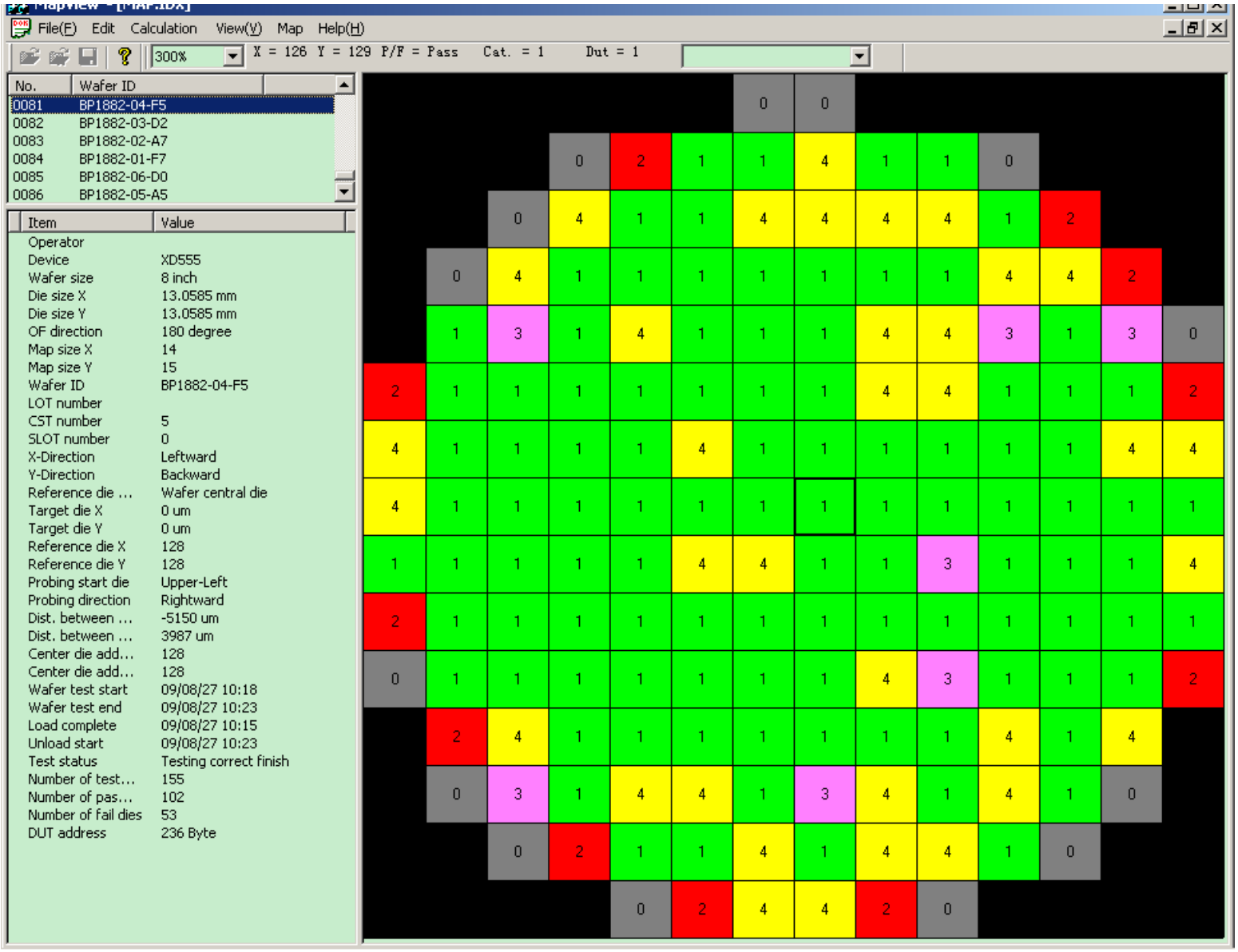
批量生产的测试方法

- 在批量生产时，ASIC测试可分为两个阶段：

1. 第一阶段是在制造完成后在大圆盘上对ASIC芯的测试，用一组探针对大圆盘上的ASIC芯一次测一个，不合格者用墨水自动标出，然后用金刚石锯把ASIC芯切开，合格的就送去封装；

2. 第二阶段对封装后的ASIC进行最后测试，通常两个阶段的测试矢量是相同的。

批量生产的测试方法


File(F) Edit Calculation View(V) Map Help(H)
300%
X = 126 Y = 129 P/F = Pass Cat. = 1 Dut = 1
No. Wafer ID
0081 BP1882-04-F5
0082 BP1882-03-D2
0083 BP1882-02-A7
0084 BP1882-01-F7
0085 BP1882-06-D0
0086 BP1882-05-A5
Item Value
Operator
Device XD555
Wafer size 8 inch
Die size X 13.0585 mm
Die size Y 13.0585 mm
OF direction 180 degree
Map size X 14
Map size Y 15
Wafer ID BP1882-04-F5
LOT number
CST number 5
SLOT number 0
X-Direction Leftward
Y-Direction Backward
Reference die ... Wafer central die
Target die X 0 um
Target die Y 0 um
Reference die X 128
Reference die Y 128
Probing start die Upper-Left
Probing direction Rightward
Dist. between ... -5150 um
Dist. between ... 3987 um
Center die add... 128
Center die add... 128
Wafer test start 09/08/27 10:18
Wafer test end 09/08/27 10:23
Load complete 09/08/27 10:15
Unload start 09/08/27 10:23
Test status Testing correct finish
Number of test... 155
Number of pas... 102
Number of fail dies 53
DUT address 236 Byte

批量生产的测试方法

- 故障的后果

  沿系统总装方向愈远，更换故障元件的代价愈大。

- 决定测试方法的因素

  故障覆盖率

8.3 测试的概念

测试的概念

- 初始化

ASIC芯片初始化到一个已知状态。

- 故障与故障模型

1. 错误(failure)：由于背离了特定行为而产生的现象。
2. 故障(fault)：电路中的物理缺陷，故障可能引起错误，也可能不引起错误。

- 故障一般可分为参数故障和逻辑故障。

① 参数故障指电路参数的变化引起的故障。

② 逻辑故障指使电路逻辑功能发生错误的故障。

测试的概念

3. 故障模型(fault model)：一个电路或元件的物理故障多种多样，故障的种类和故障的数目都有很大的差别。为了便于研究，按照故障的特点和影响将其归类，称为故障模型。故障模型应能准确反映某一类故障对电路或系统的影响，即模型化故障应具有典型性、准确性和全面性。另一方面，模型应尽可能简单，以便作各种运算和处理。

4. 滞留故障(Stuck-at fault)：数字电路中最常用的故障模型是滞留故障，它假设故障在一个逻辑门上引起逻辑门的输入或输出固定在逻辑"1"或逻辑"0"。滞留故障有两种滞留状态，即：

- ✓ 滞留于1：即使一个结点被驱动到低电平，它也始终处于高电平。
- ✓ 滞留于0：即使一个结点被驱动到高电平，它也始终处于低电平。
- ✓ 对于一个有*n*个结点的电路，有单个滞留故障的不同电路总数为2*n*。

5. 故障覆盖率

故障覆盖率 = 能识别的有单个滞留故障的电路数目/2*n*

测试的概念

- 可控制性与可观察性

可控制性与可观察性是可测试性的两个方面。对可控制性和可观察性有许多不同的定量度量方法。

1. 原始输入(Primary input)：通过芯片引脚或板子连接器而加到电路的输入。
2. 原始输出(Primary output)：通过芯片引脚或板子连接器而观察到的输出。
3. 可控制性(Controllability)：通过电路的原始输入把测试矢量加到被测子电路的能力。
4. 可观察性(Observability)：通过电路的原始输出或其它输出点能观察被测子电路的响应的能力。

西安电子科技大学

8.4 测试码生成

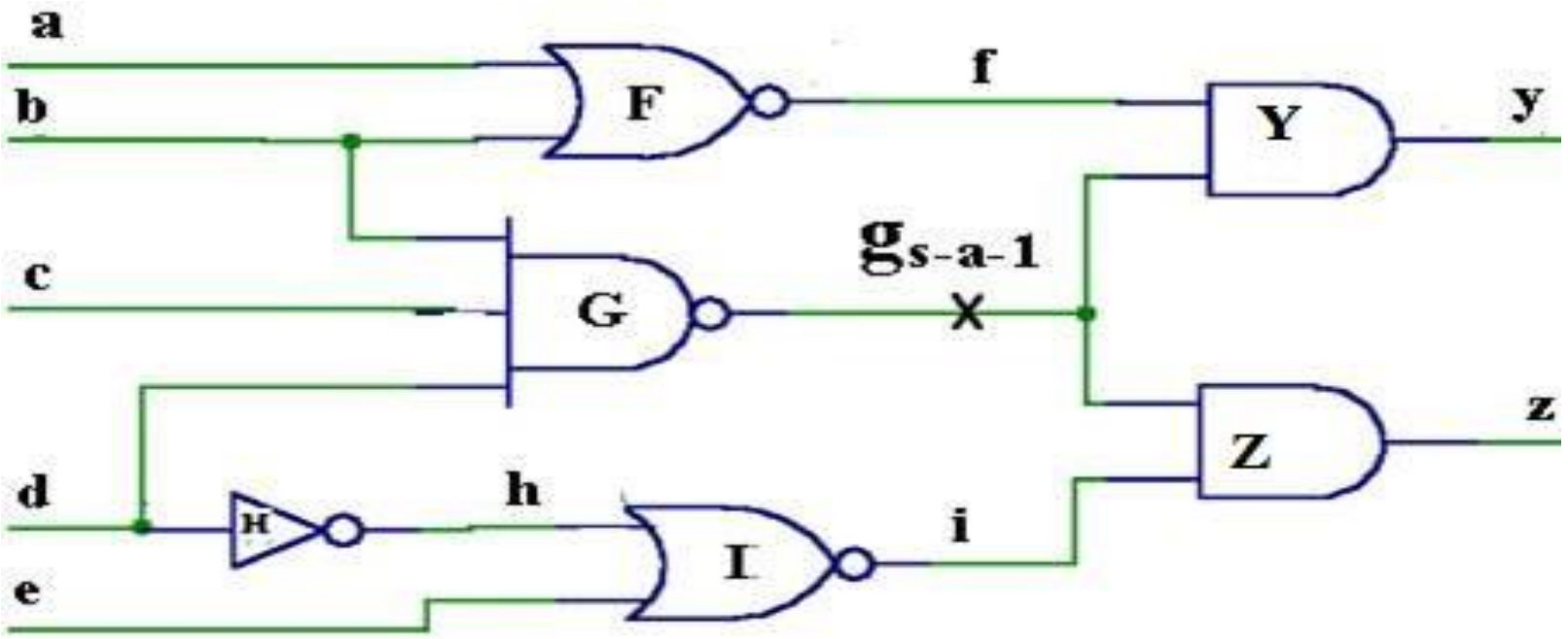
测试码生成方法

- 测试码生成的方法

1. **穷举测试码**(exhaustive test pattern)：根据电路的输入端个数，将所有可能的输入矢量组合作为测试集。对组合电路来说，穷举测试码是完备的测试集。对于规模不大的电路，穷举测试码可以根据真值表得到，再经过适当化简，可以形成相当不错的测试集。但如果电路规模较大，测试码的数目随输入端增加而指数增加，往往是不可接受的。
2. **伪随机测试码**(pseudo-random pattern)：对于*n*输入端电路产生一些*n*位二进制数作为测试输入矢量，这些二进制数近似于随机数，称为伪随机测试码。这种测试码容易产生，测试矢量数目也比较少。如果能达到一定的故障覆盖率，就不失为一个好的测试集。
3. **测试生成算法**(test generation algorithm)：根据逻辑电路本身的结构用算法自动生成测试码，称为测试码自动生成(ATPG, Automatic Test Pattern Generation)。迄今为止，出现了多种测试生成算法，如组合逻辑的两种测试码自动生成算法——**单路径敏化法**和**D算法**。

单路径敏化法

- 对指定故障点的测试码生成算法的基本思想是通过输入端测试矢量把故障传播到输出端，使得故障情况电路的输出与正常电路的输出结果不同。



单路径敏化法

- 为了把故障传播到外部输出端，要有两个条件：

1. 输入测试矢量应能够使得故障点g在故障情况下与正常情况下状态值不同。本例中因故障值为**1**，要求正常值为**0**。即要求输入矢量使g的状态值为**0**，称为**故障敏化**。
2. 有至少一个外部输出端的正常值与有故障时的值不同。为了能做到这一点，要求从故障点出发能找到一条或几条路径到达输出端，使该路径上每个结点的正常值与有故障时的值不同。这条路径称为**敏化路径**(sensitized path)。

- 通过寻找敏化路径来求测试集的方法称为敏化路径法。

单路径敏化法

- ✓ 把正常为0，有故障时为1的信号线状态记为**0/1**；
- ✓ 把正常为1，有故障时为0的信号线状态记为**1/0**。

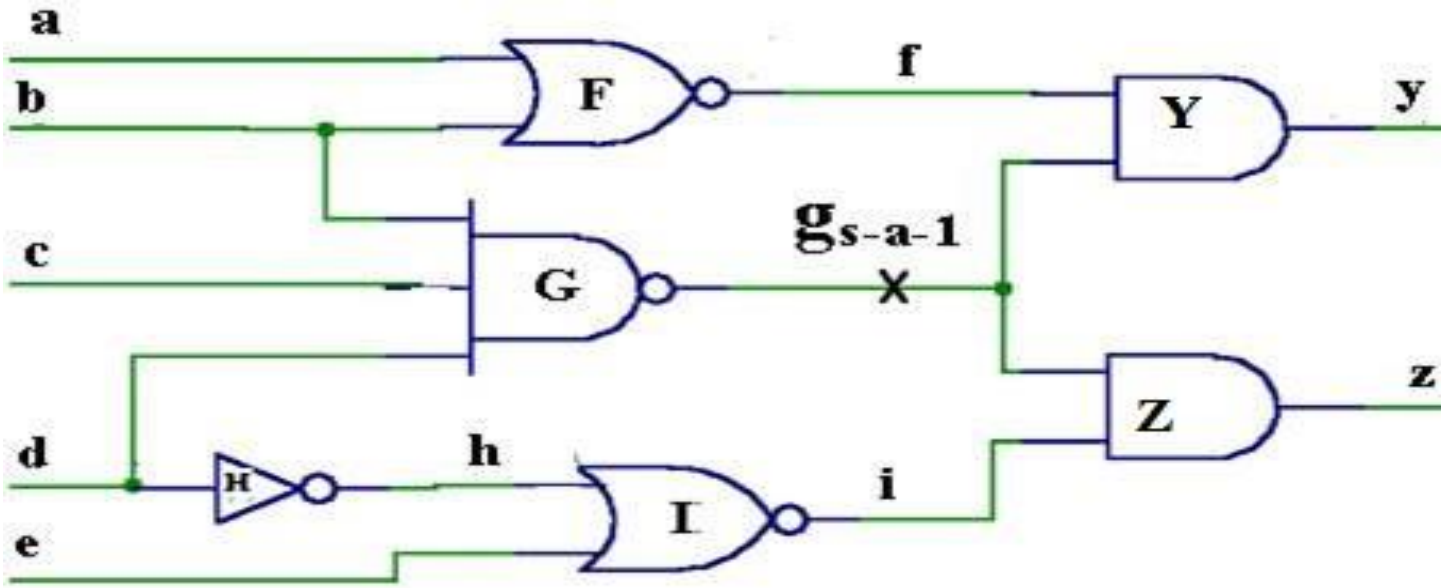
- ❑ 因为g点为滞留为1的故障，故该点的敏化值应为**0/1**。

- ❑ 各种门传播故障的条件为：

1. 非门：均可传播；
2. 与门、与非门：其他各端置1；
3. 或门、或非门：其他各端置0；
4. 异或门：另一端置1、置0均可。

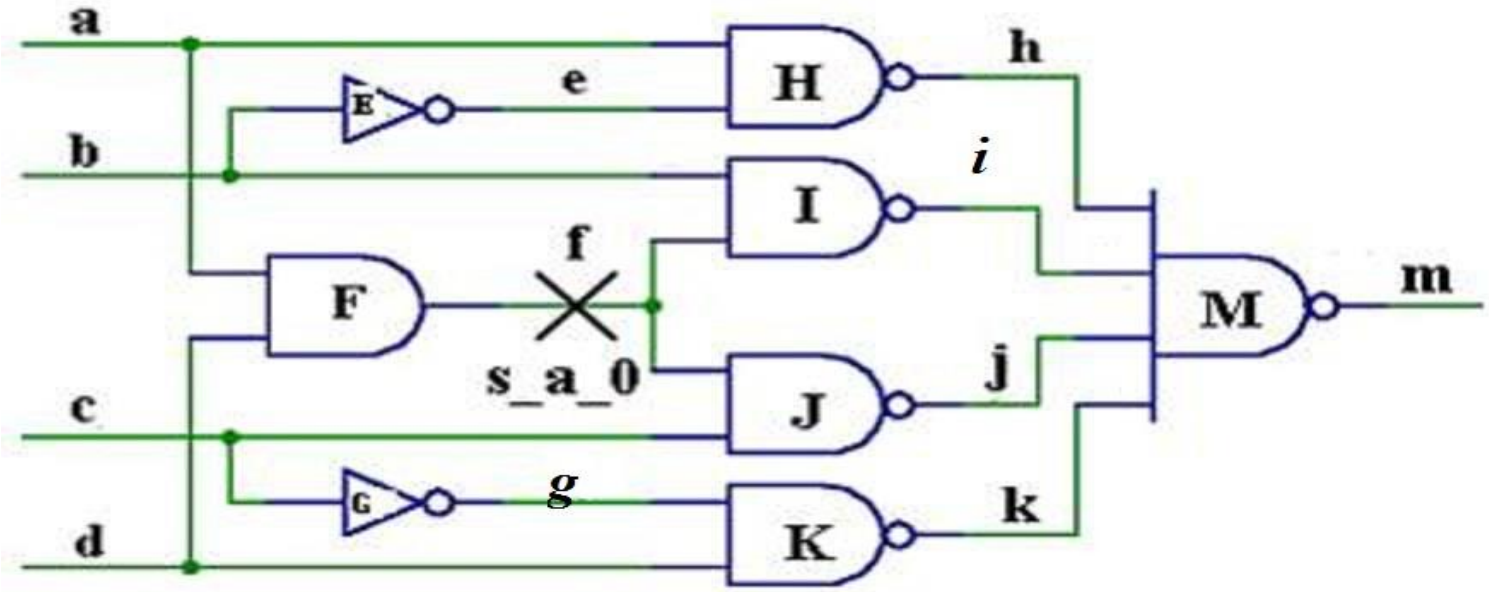
单路径敏化法

- 在选择路径时，总是只选择一条敏化路径，故这种方法称为单路径敏化法。
- 测试矢量*T*={(*X*,1,1,1,0);(0,0)}
- 故障输出向量为(0,1)。



多路径敏化法

- 单路径敏化法不能保证对任一非冗余故障都能找到测试矢量。典型的情况是故障处于再会聚路径(reconvergent path)中。



8.5 故障模拟

故障模拟

- 故障模拟(fault simulation)是检验输入向量(或序列)是否成为有效测试码的手段。

- 故障模拟的方法

1. 串行故障模拟
2. 并行故障模拟
3. 并发故障模拟

西安电子科技大学

8.6 可测性设计法

可测性设计法

- 可测性设计应考虑的问题：

1. 变不可测故障为可测故障；
2. 测试数据生成的时间应尽量少；
3. 测试数据应尽量少。

- 常用的可测性设计方法主要有：

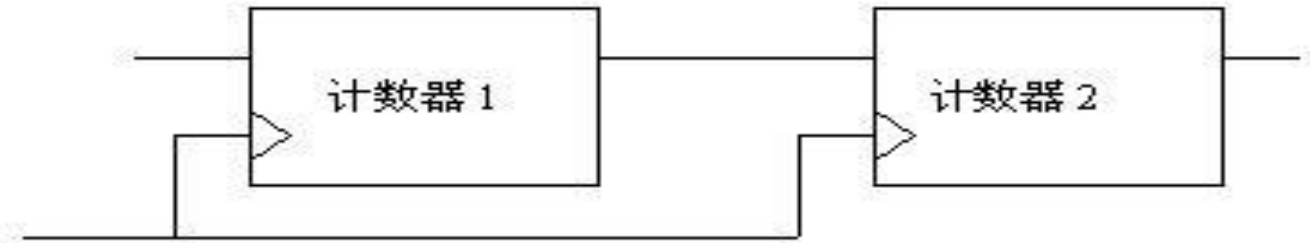
1. 特设法
2. 扫描路径法
3. 内设自测试法(**BIST**, Built-In-Self-Test)
4. 边界扫描法

特设法

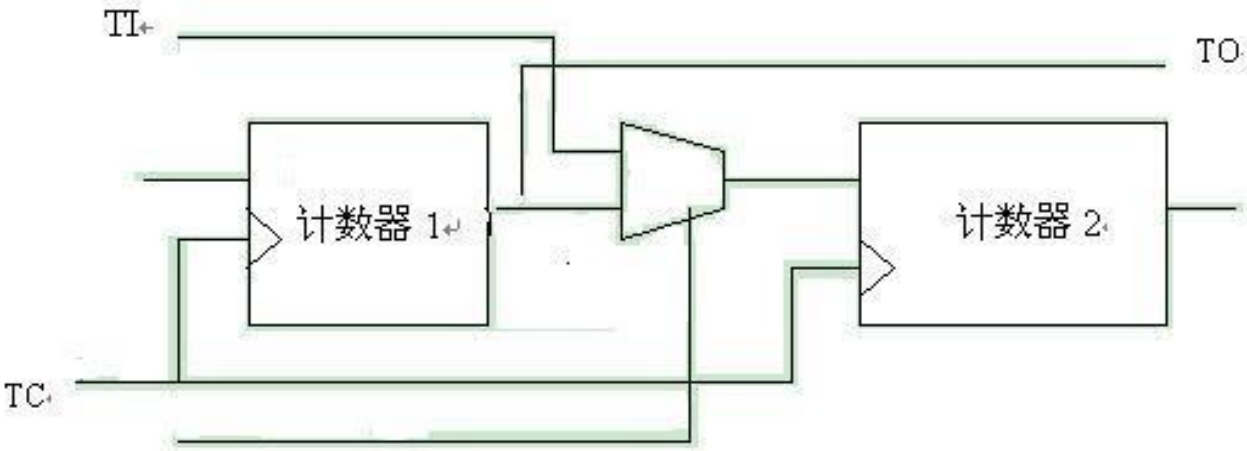
- 分两步实现，首先判定电路中有困难的结点(难以观察或难以控制的结点)，然后插入附加电路把它们直接连到原始输入或原始输出端。
- 一般用可测性分析器来判定这些结点，也有一些经验法则添加测试点：

1. 应沿着关键路径设置测试点；
2. 在控制逻辑设置测试点，如：时钟信号、控制信号等；
3. 在逻辑功能块的结合部设置测试点，如计数器组、移位寄存器组、编译码器及多路选择器等处；
4. 测试点的设置应首先考虑可控制性。如用测试点把计数器链断开；
5. 测试点的设置应考虑可观察性。

特设法



计数器链



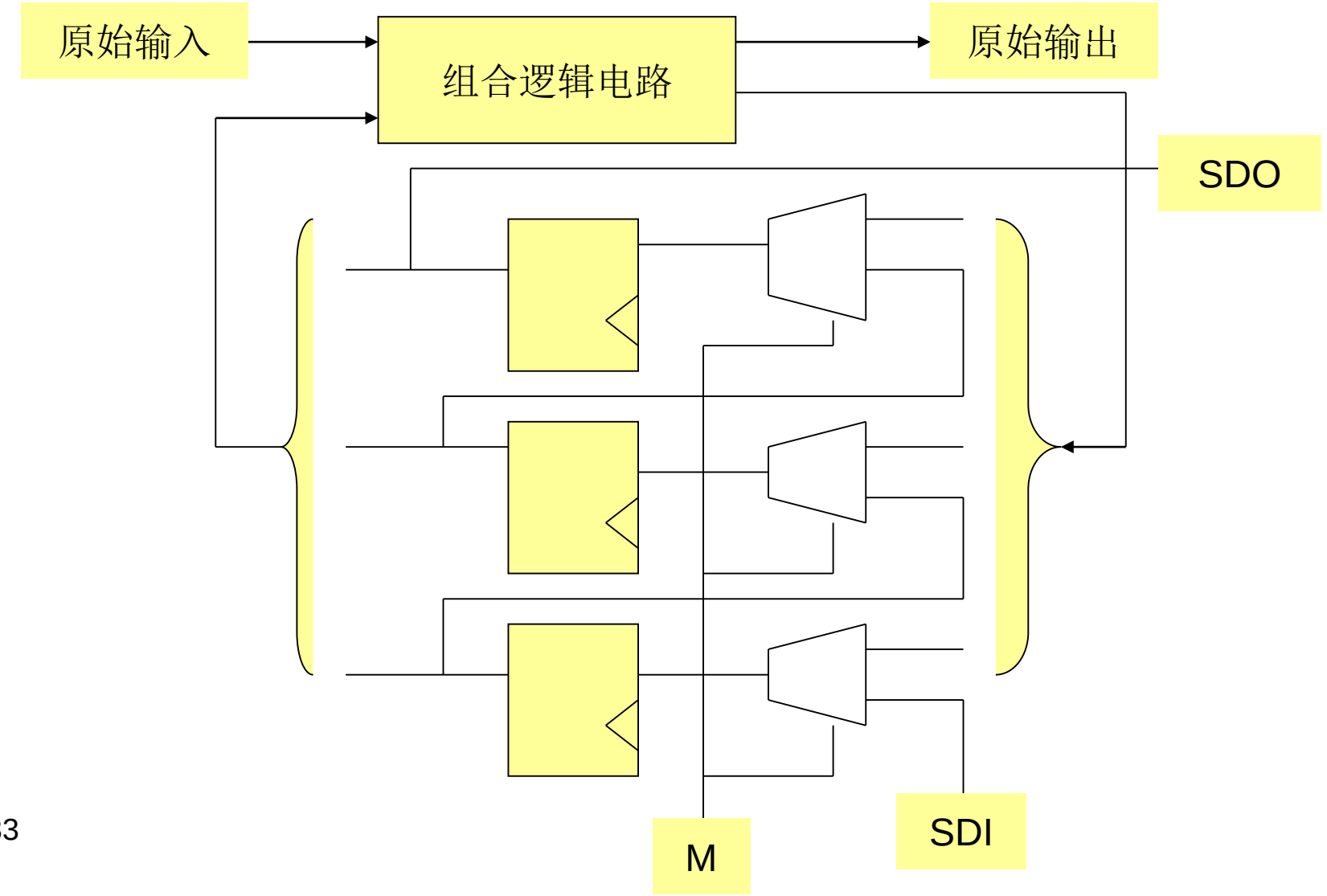
增加测试点后的计数器

特设法的缺点：增加测试脚。

扫描路径法

- 一个同步时序系统一般可看成由组合电路(下一个状态电路和输出电路)和时序电路两部分组成，如果能把这两部分分开测试将大大降低测试的复杂度。扫描路径法就是这样一种测试同步时序系统的方法。
- 扫描路径法的基本原理是：把系统中的所有寄存器连成一个移位寄存器链，如下图所示，这个移位寄存器有一个模式控制端***M***，在正常工作模式时，***M***=0，多路选择器连接组合电路和寄存器完成同步时序系统正常的逻辑功能；在测试模式时，***M***=1，多路选择器使寄存器形成一个移位寄存器链。移位寄存器的输入为扫描数据输入端***SDI***，输出为扫描数据输出端***SDO***。

扫描路径法



扫描路径法

- 扫描路径是通过以下步骤实现对时序电路测试的：

1. 使M=1，测试移位寄存器链中的触发器

如果给SDI端加上一串0、1序列，则经过n个(n等于移位寄存器链中触发器的个数)时钟周期在SDO端将会出现相同的0、1序列。可用”00110…”序列作为输入序列，这样就可测试触发器状态是否反转、触发器是否稳定。

2. 测试组合逻辑

a) 使M=1，通过SDI把一个测试矢量加到n个触发器上。

b) 使M=0，加一个测试矢量在原始输入端，观察原始输出端的输出情况。在触发器的时钟端加一个时钟把把组合电路的部分输出(通过多路选择器与触发器相连的那部分输出)装入触发器中。

c) 使M=1，经过n-1个时钟周期把触发器采集的数据通过SDO移出芯片。

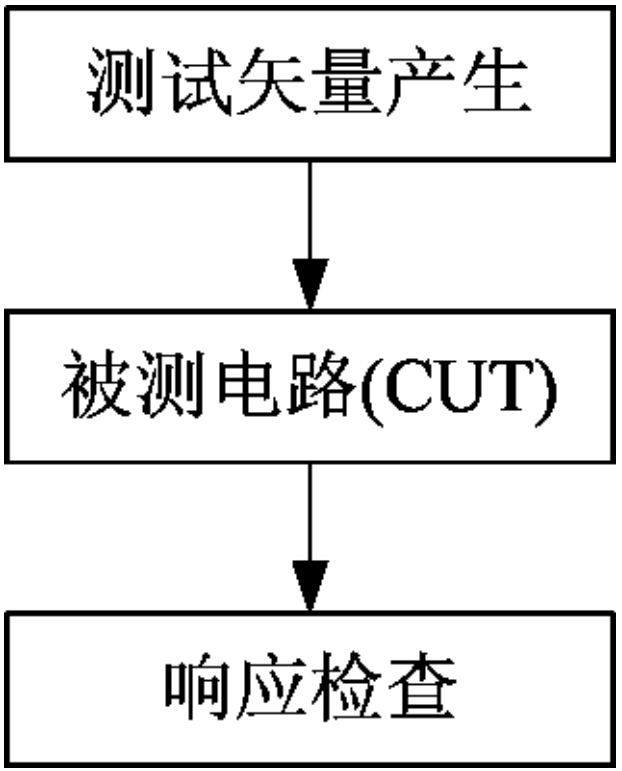
对于第二次测试第a)步也可在第c)步中同时完成，即在触发器中数据移出的同时新数据也可同时移入，以便为下一次测试做准备。

扫描路径法

- 扫描路径法的优点是用特别设计的测试矢量，把每个组合逻辑部分分别处理。反馈环自动切断、计数器链断开，电路的逻辑深度大大降低。所有的存储单元，通过扫描路径，直接连到一个原始输入端和一个原始输出端。这样能大大减少测试时间。
- 扫描路径法的缺点是要增加额外的测试引脚M、SDI和SDO(SDI和SDO可与系统其他引脚共用)，增加用于实现扫描路径的多路选择器，这样就造成芯片面积的增加、功耗的增加及系统性能的下降。
- 扫描路径法还有一些改进形式：如多扫描路径和部分扫描路径。

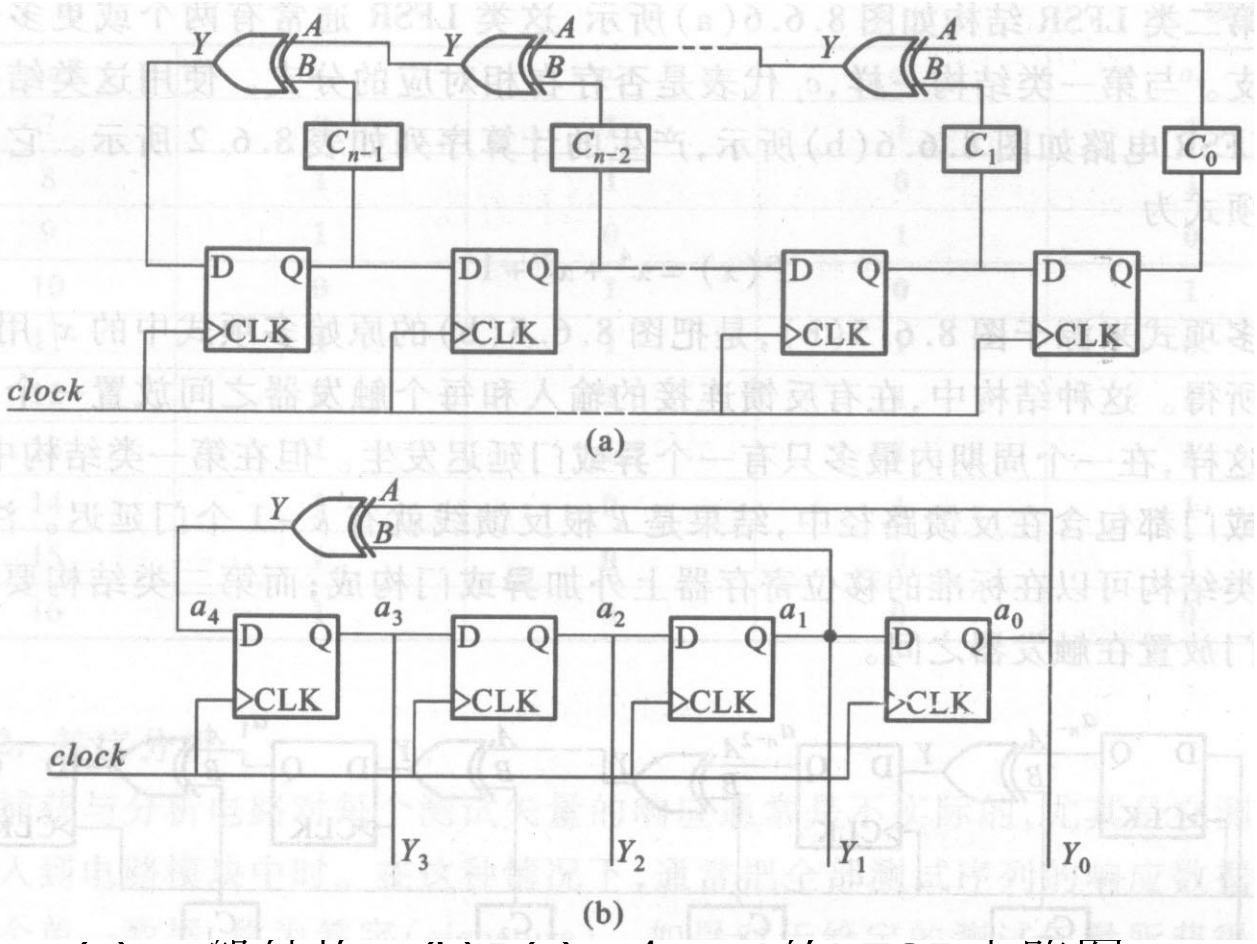
内设自测试法

- 内设自测试法是指在ASIC中包含测试矢量的产生与电路响应判别电路的测试方法，这种方法不仅可以简化测试设备、降低测试设备的成本，而且允许对器件进行现场测试。



内设自测试法

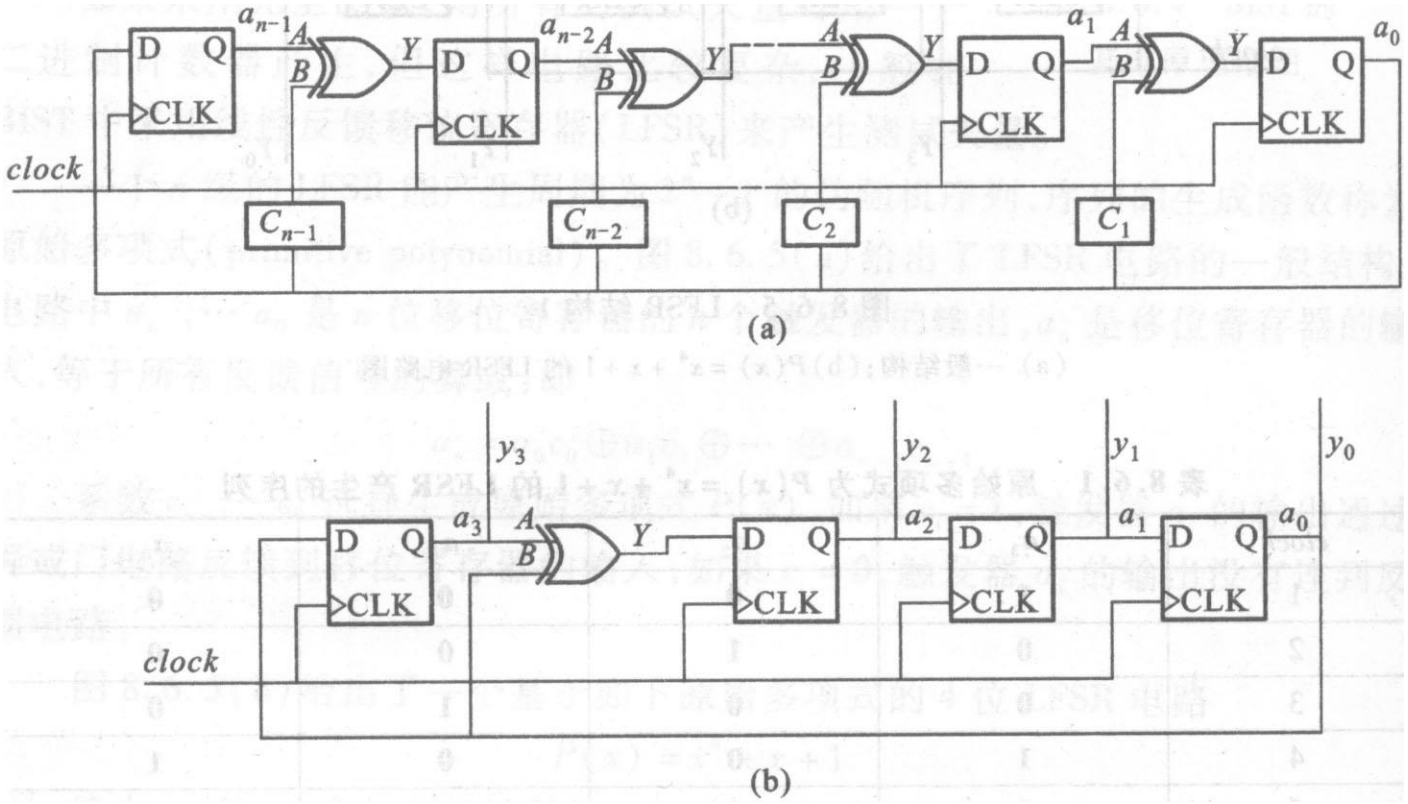
- 测试矢量产生



(a) 一般结构　(b)$P(x)=x^4+x+1$的LFSR电路图

内设自测试法

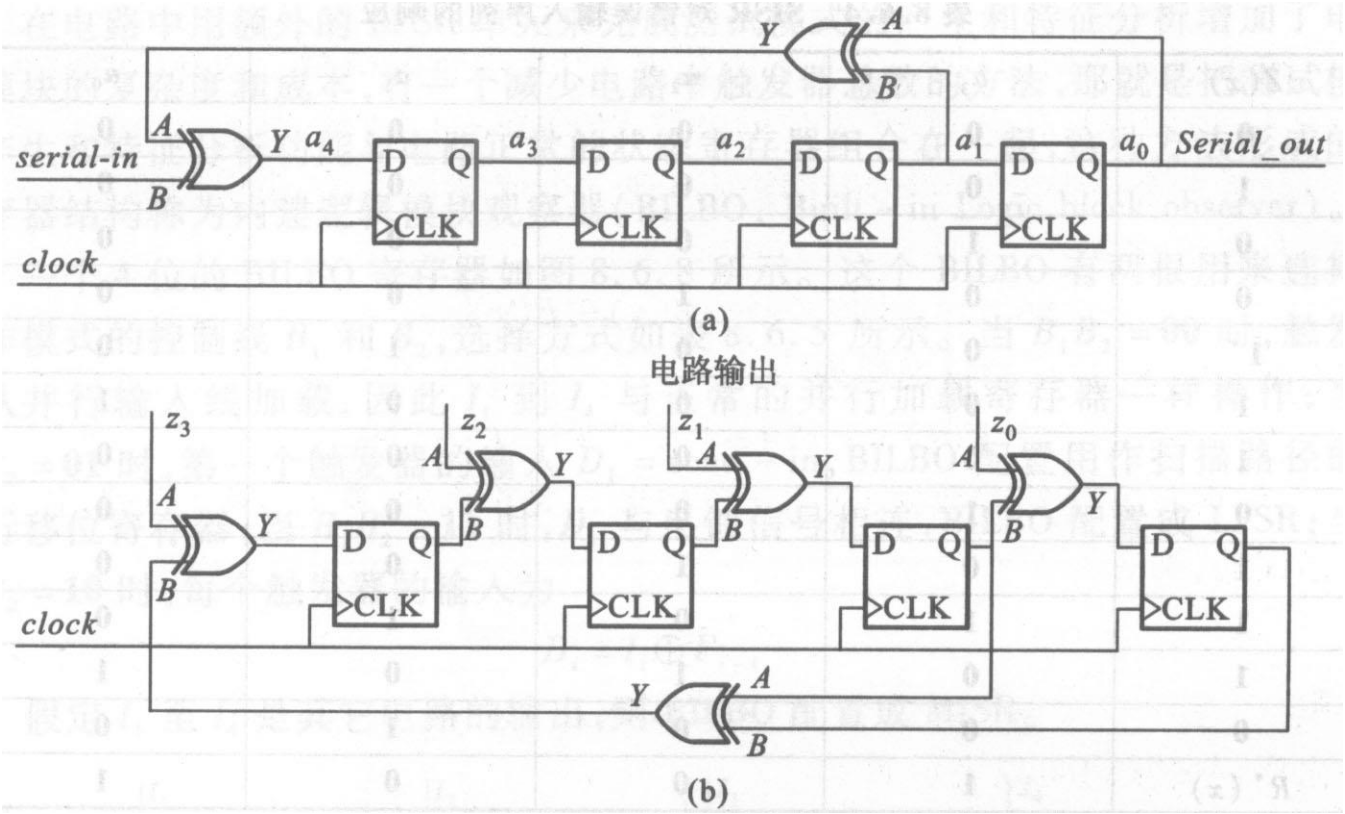
- 测试矢量产生



(a) 一般结构　(b)$P(x)=x^4+x^3+1$的LFSR电路图

内设自测试法

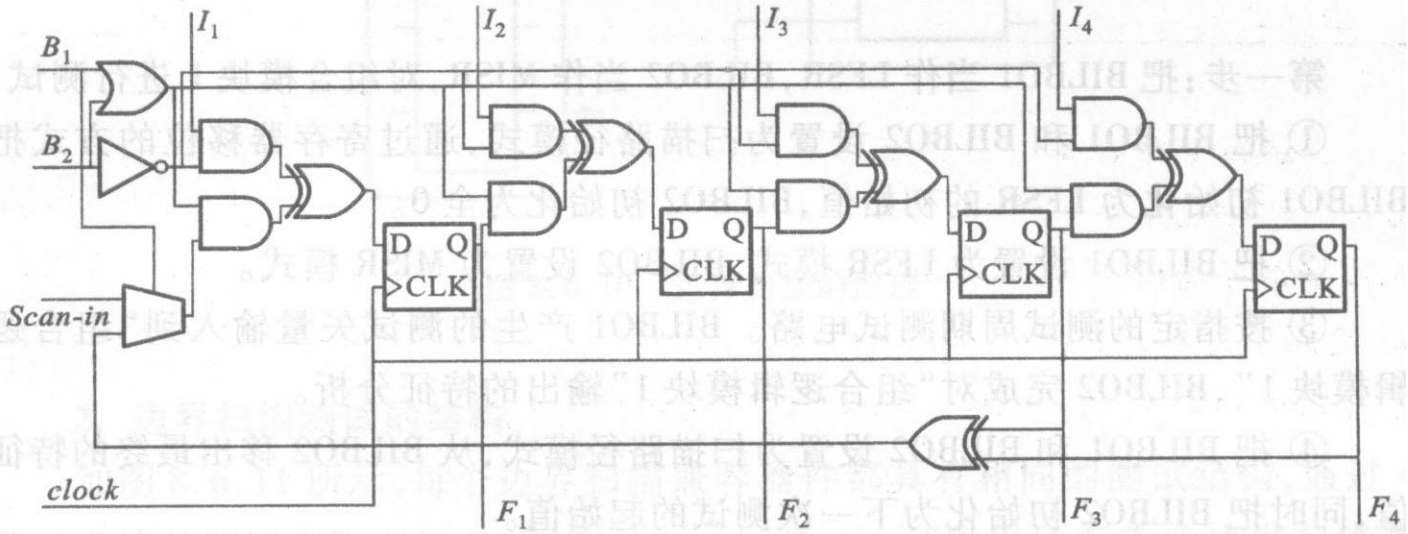
- 签字分析



(a) 单输入签字寄存器　(b) 多输入签字寄存器

内设自测试法

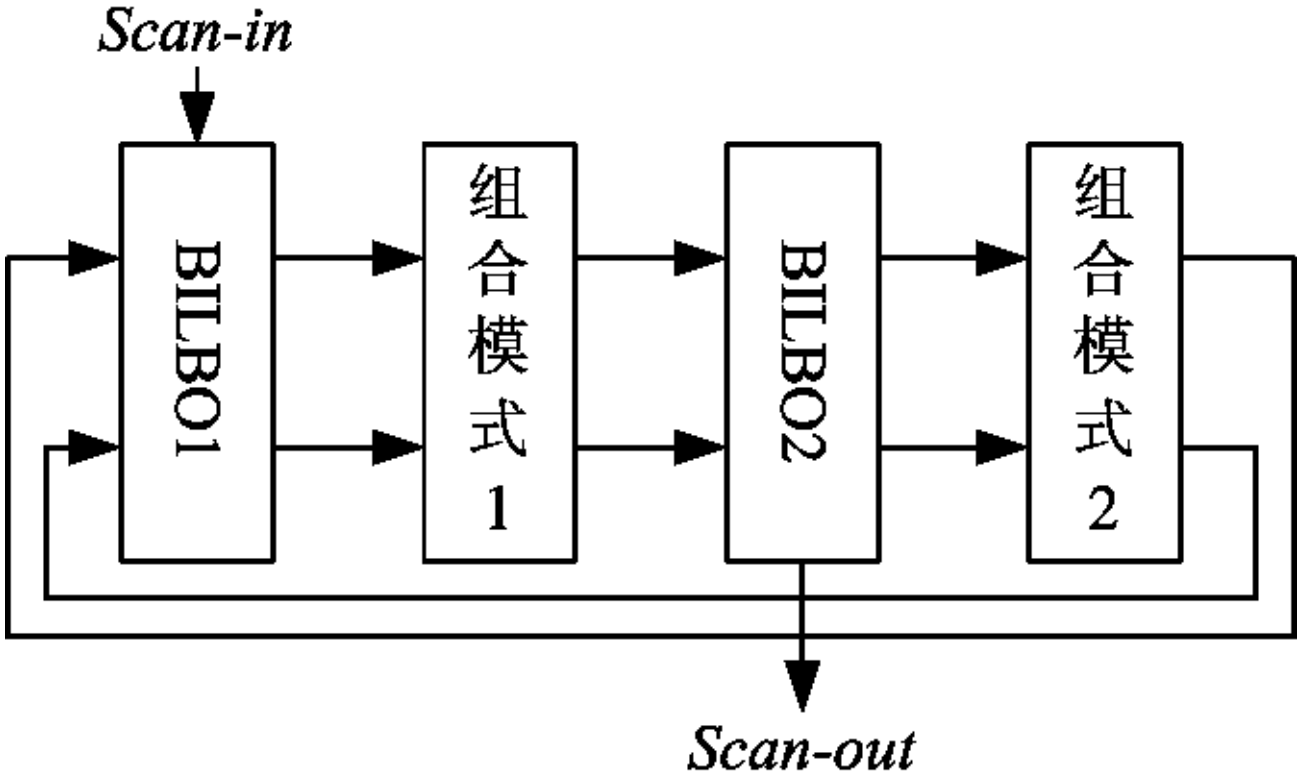
- 内建逻辑模块观察器



B_1	B_2	选择输入	D_i	功　能	测试功能
0	0	Scan-in	I_i	并行加载	正常模式
0	1	Scan-in	F_{i-1}	线性移位	扫描路径模式
1	0	Feedback	I_i XOR F_{i-1}	MISR	特征分析
1	1	Feedback	F_{i-1}	LFSR	测试模式产生

内设自测试法

- 内建逻辑模块观察器



内设自测试法

- 内建逻辑模块观察器

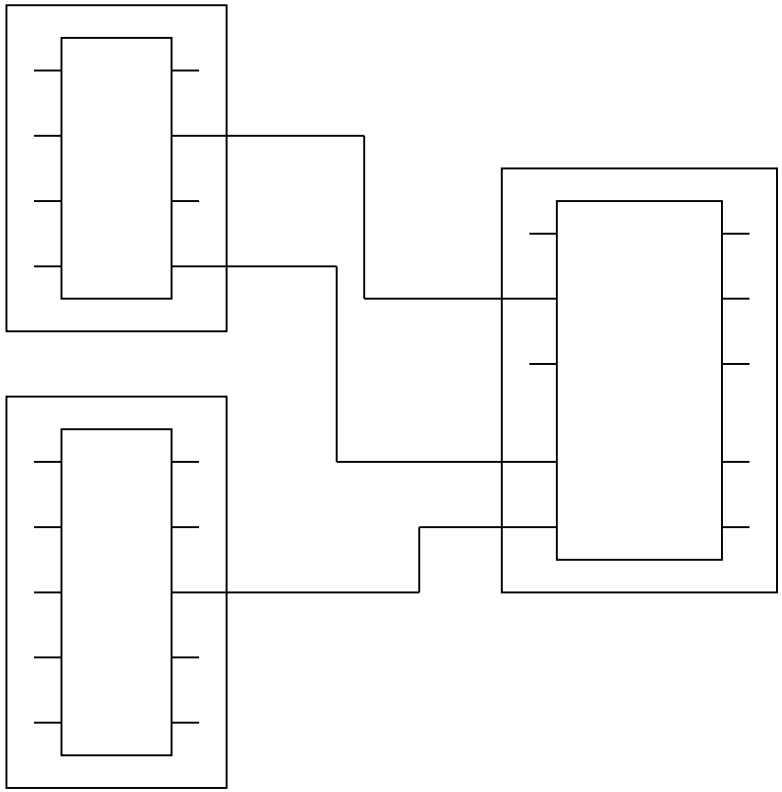
第一步：把BILBO1当作LFSR，BILBO2当作MISR，对组合模块1进行测试：

1. 把BILBO1和BILBO2设置为扫描路径模式，通过寄存器移位的方式把BILBO1初始化为LFSR的初始值，BILBO2初始化为全0；
2. 把BILBO1设置为LFSR模式，BILBO2设置为MISR模式；
3. 按指定的测试周期测试电路。BILBO1产生的测试矢量输入到”组合逻辑模块1”，BILBO2完成对”组合逻辑模块1”输出的特征分析；
4. 把BILBO1和BILBO2设置为扫描路径模式，从BILBO2移出最终的特征值，同时把BILBO2初始化为下一次测试的起始值。

第二步：把BILBO2当作LFSR，BILBO1当作MISR，测试“组合逻辑模块2”。交换两个BILBO的角色，重复上面第一步中2)到4)。

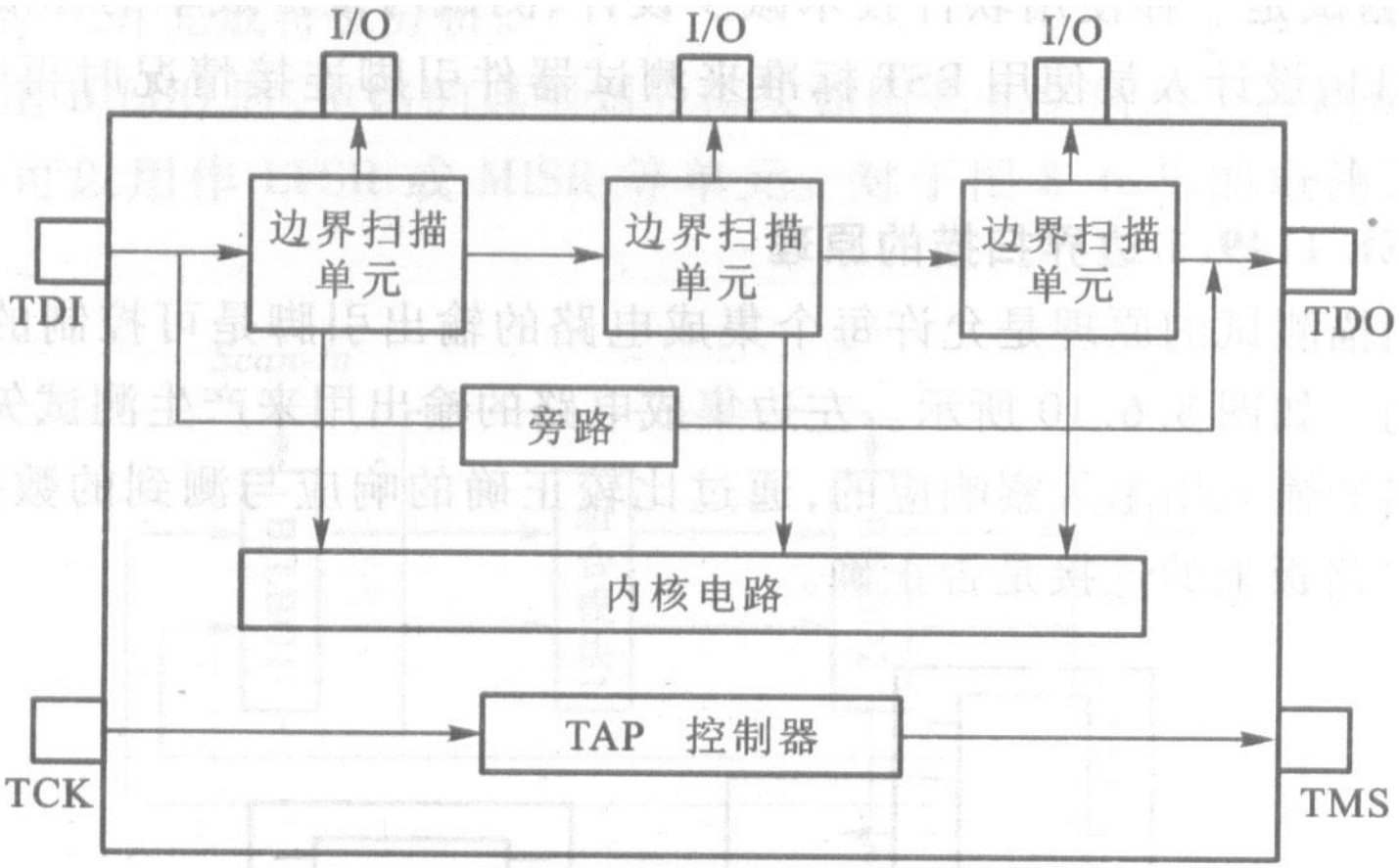
边界扫描法

- JTAG测试是一种使用软件技术减少设计、测试与维护成本的标准，器件有了JTAG接口，设计人员使用BST标准来测试器件引脚连接情况时再也不必使用物理探针了。



边界扫描法

- 边界扫描测试的结构



边界扫描法

- 测试存取端口TAP由4个引脚或5个引脚组成，它们分别为：

① 测试时钟TCK(Test Clock)：用来同步内部边界扫描状态机(TAP控制器)操作的时钟信号。

② 测试模式选择TMS(Test Mode Select)：内部状态机的模式选择信号，在TCK信号上升沿到来时其电平高低决定了下一个状态机状态。

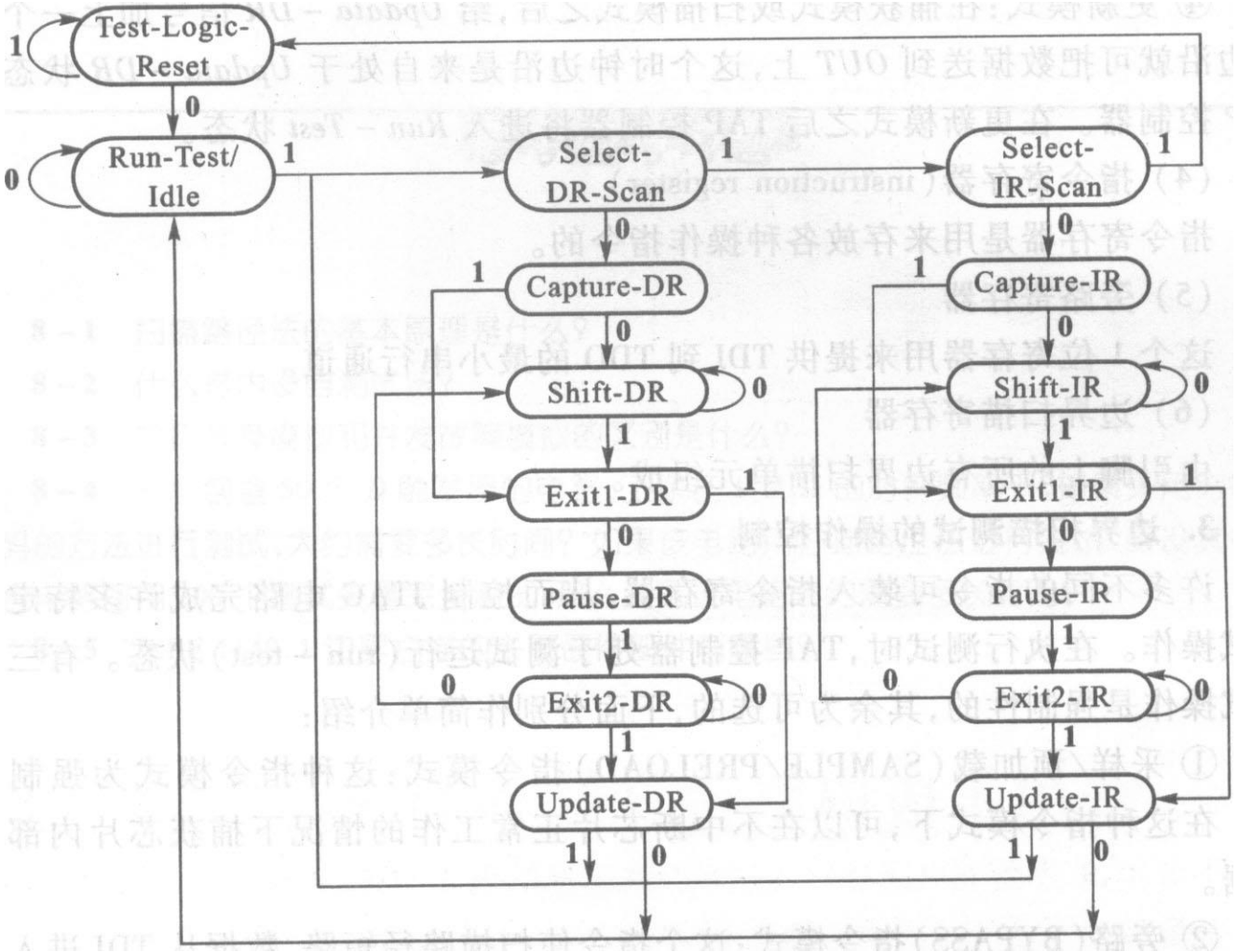
③ 测试数据输入TDI(Test Data In)：指令和测试编程数据的串行输入引脚，数据在TCK上升沿时刻移入。

④ 测试数据输出TDO(Test Data Out)：测试编程数据的串行输出引脚，当内部状态机处于正确的状态时，数据在TCK的下降沿移出。如果数据不是正在移出时，该引脚处于三态。

⑤ 测试复位输入TRST(Test Reset)：异步复位端口，当为低电平时内部状态机立即跳至复位状态。由于此脚为可选引脚，而多一个引脚将增加成本，同时也由于内部状态机的同步复位机制较好，因此有些器件中无此引脚。

边界扫描法

- TAP控制器的状态图



边界扫描法

- 边界扫描单元有以下四种工作模式：

1. 正常模式：数据从IN传到OUT。
2. 扫描模式：ShiftDR信号选择SCAN_IN作为输入，ClockDR作为扫描路径的时钟。ShiftDR信号是由TAP控制器中Shift-DR状态驱动的。当TAP控制器处于Capture-DR或Shift-DR状态时ClockDR有效。
3. 捕获模式：ShiftDR信号选择IN作为输入，数据在ClockDR时钟作用下移入扫描路径寄存器，从而获得系统的观察值。
4. 更新模式：在捕获模式或扫描模式之后，给Update-DR信号加上一个时钟边沿就可把数据送到OUT上，这个时钟边沿是来自处于Update-DR状态的TAP控制器。在更新模式之后TAP控制器将进入Run-Test状态。

边界扫描法

- 边界扫描测试的操作控制

1. 采样/预加载(SAMPLE/PRELOAD)指令模式：这种指令模式为强制性的。在这种指令模式下，可以在不中断芯片正常工作的情况下捕获芯片内部的数据。
2. 旁路(BYPASS)指令模式：这个指令使扫描路径短路，数据从TDI进入旁路寄存器，从TDO输出。
3. 外测试(EXTEST)指令模式：通过在输出引脚加外测试矢量和在输入引脚捕获测试结果，从而测试器件与其它JTAG兼容器件在PCB板上的连接情况。
4. 执行BIST(RUNBIST)指令模式：运行器件上的内建自测试。
5. ID码(IDCODE)指令模式：这种指令模式用于实现对JTAG链中器件的隐蔽访问。当选用IDCODE模式时，标志寄存器装入32位由厂商定义的标志码，并连接到TDI和TDO之间。通过使用IDCODE指令模式，可以判别连接到JTAG口上的器件名，还可以对链中的FPGA/CPLD器件有选择地进行配置。
6. 用户码(USRCODE)指令模式：用来检查用户器件周围连接的电气特征。当选用这种指令时，USR寄存器连接到TDI和TDO之间。

边界扫描法

- 边界扫描描述语言

为了使不同厂家生产的边界扫描兼容部件能一起工作，人们规定了一个标准的描述语言——边界扫描描述语言(Boundary Scan Description Language，BSDL)，它是VHDL的一个子集。BSDL并不是用来仿真的，也不包含任何边界扫描部件的模型。BSDL提供了一种标准方法用来描述包含IEEE 1149.1边界扫描的ASIC的特性和行为，同时也提供了向测试产生软件传递信息的标准方法。利用BSDL测试软件可以检查器件的BST特性是否正确。