北京大学
PEKING UNIVERSITY

# 智能硬件体系结构

## 第三讲：半导体晶体管与数字逻辑基础

主讲：陶耀宇、李萌

2025年秋季

# 注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况



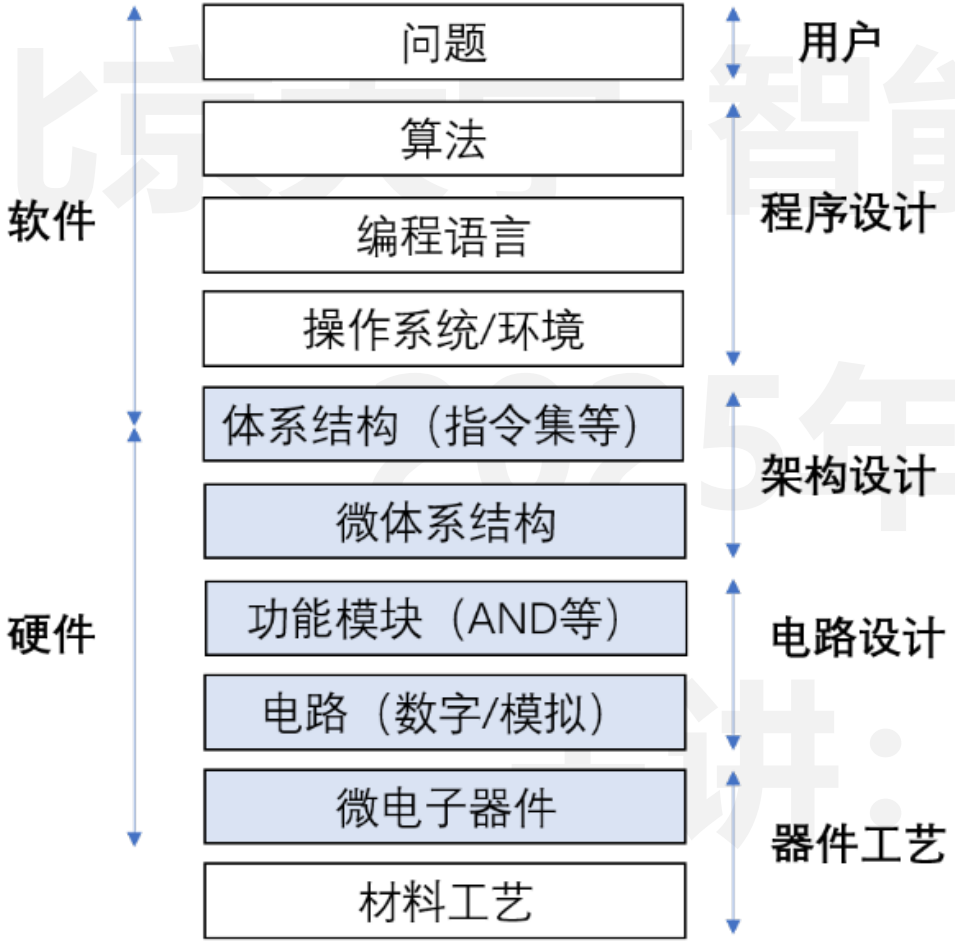
- 请各位选课同学确定能够正常登陆CLab平台
- **国庆节后会在课程上讲解Verilog语言的语法和代码编写范式**
- **课程资料参照：https:aiarchpku.com**
- **Clab问题请联系助教詹喆同学**

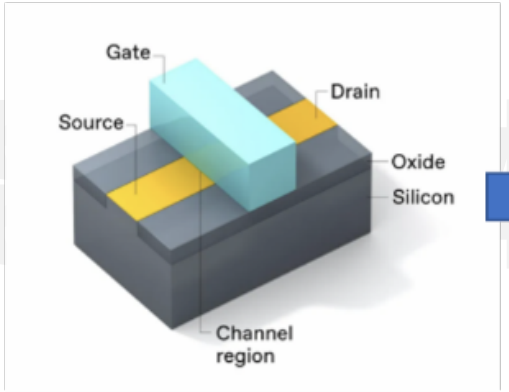
# 目 录

CONTENTS

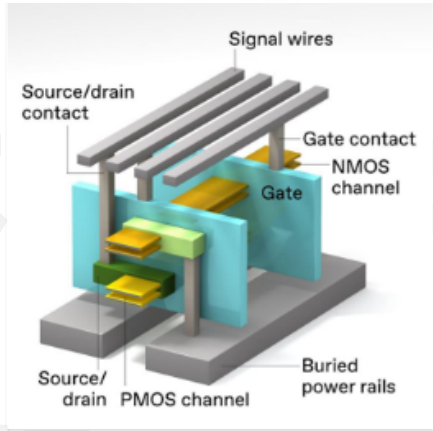
# MOSFET晶体管 – 现代芯片的基石

- 半导体晶体管是现代芯片的基石

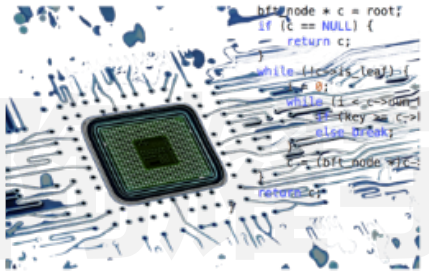


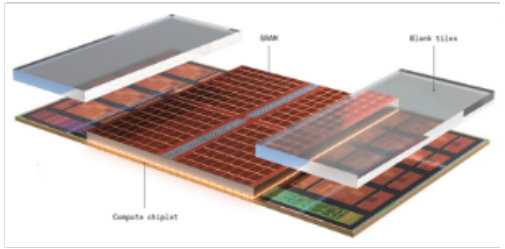

半导体物理与晶体管




数字逻辑与复杂电路



软硬件协同设计

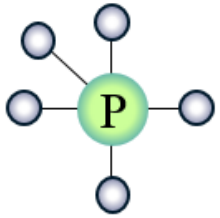


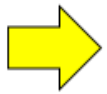
数据流、指令与芯片架构

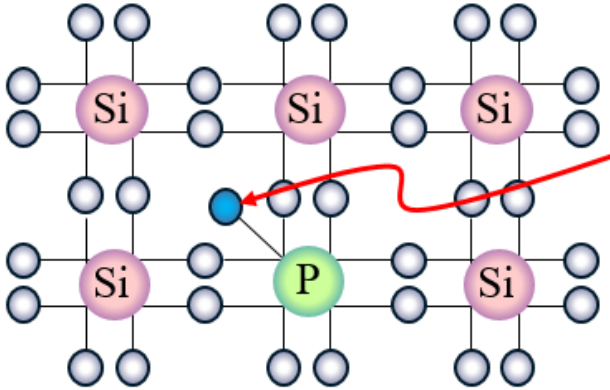
# MOSFET晶体 – 现代芯片的基石

- N型与P型半导体的概念

n型半导体





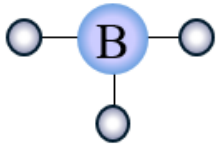

电子处于“多余”状态。

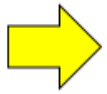
半导体内富含自由电子。

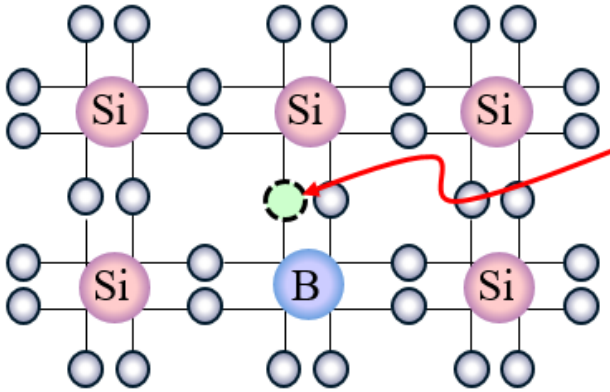
外加电压后电子就会被吸向+极

半导体变为导电的状态

p型半导体



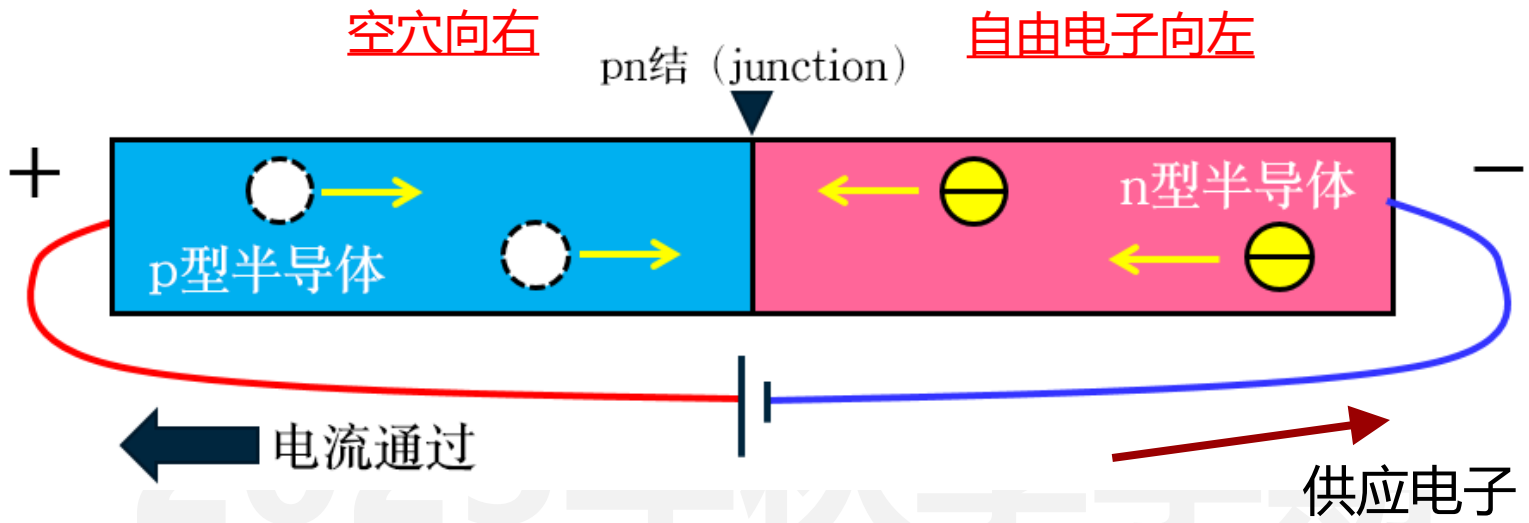



没有电子的“空位”状态

这种空位叫空穴，也就是说空穴中无实体，也叫虚拟粒子

# MOSFET晶体 – 现代芯片的基石

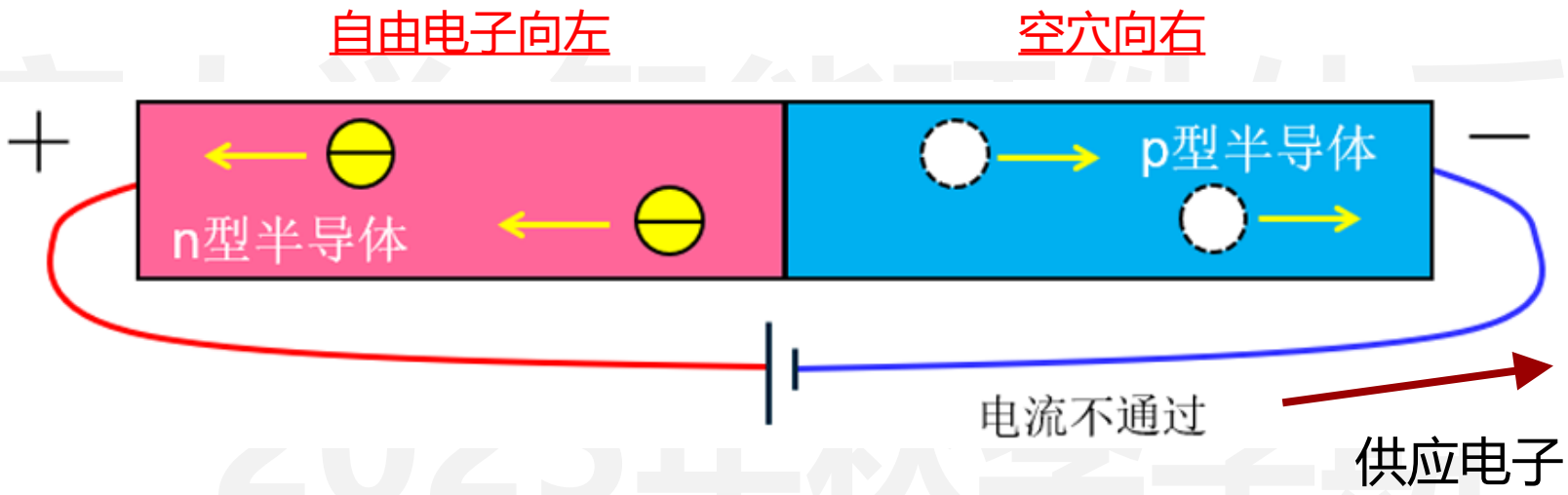
- PN结的概念 – 导通状态




- 对PN结外加电压使P为正极，空穴和电子都向结面移动
  当空穴与电子在结面（Junction）相遇时，电子飞入空穴，两者抵消
- **相应的新电子从电源补充流入n层**，同时电子从p层流出而产生新的空穴，如此反复，电流不断通过

# MOSFET晶体 – 现代芯片的基石

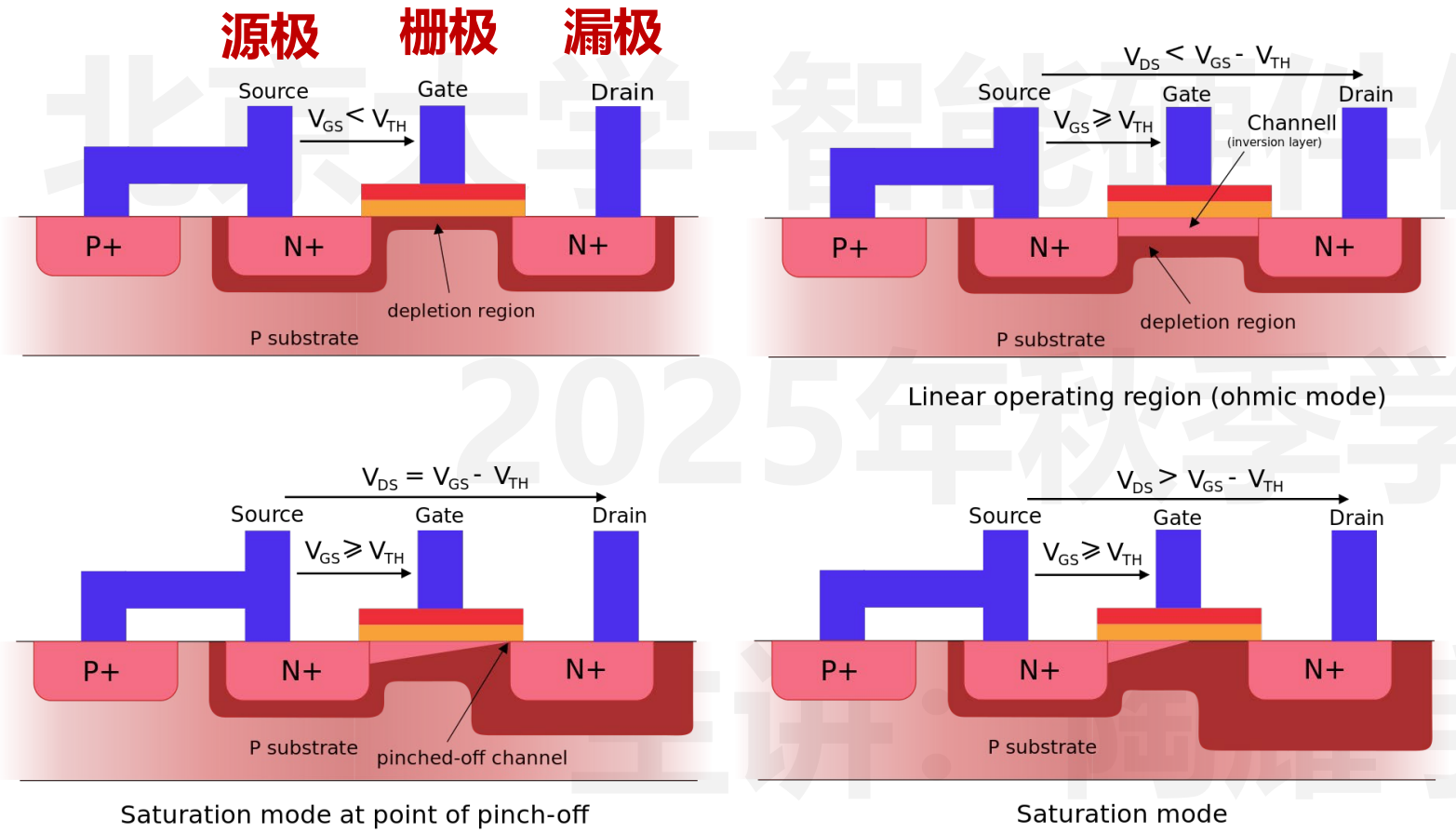
- PN结的概念 – 断开状态




- 对PN结外加电压使N为正极。
- **空穴和电子向相互远离的方向移动**，因此不会在结面相遇，电流无法通过
- **在结面附近会形成既无空穴又无电子存在的区域，叫做耗尽层**，它会产生耐压。
- 从上述可知pn结**具有整流作用**

# MOSFET晶体 – 现代芯片的基石

- MOSFET结的概念 – 栅极(gate)、漏极(drain)和源极(source)

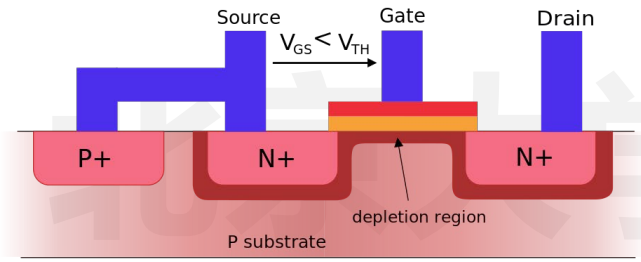



- 当栅极-源极间的电压Vgs<Vth时，N接了电源正极无论漏极-源极间的电压Vds为多少，**因为PN结的单向导通性**，不会有电流从漏极流向源极

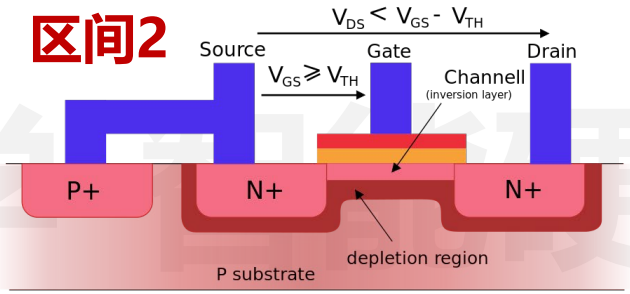
- 当Vgs超过阈值电压Vth后，**会形成一个横跨二氧化硅层的电场**，在这种电场的作用下，**SiO2层和P区交界处附近的电子会被吸引至SiO2侧**，在SiO2侧形成一个局部电子浓度相对较高的带状区域(沿着SiO2表面)，即**N型导电沟道**(depeletion)

# MOSFET晶体 – 现代芯片的基石

- MOSFET有三个工作区间：断开、线性（欧姆区间）、饱和（电压不随电流线性增加）

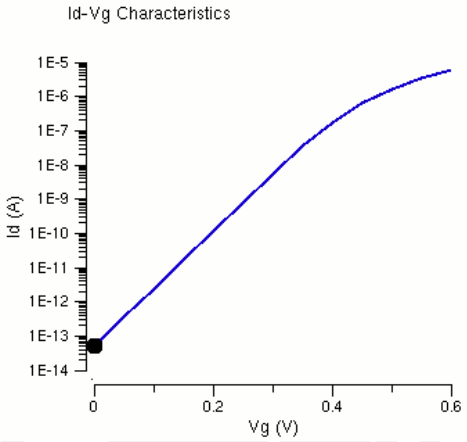


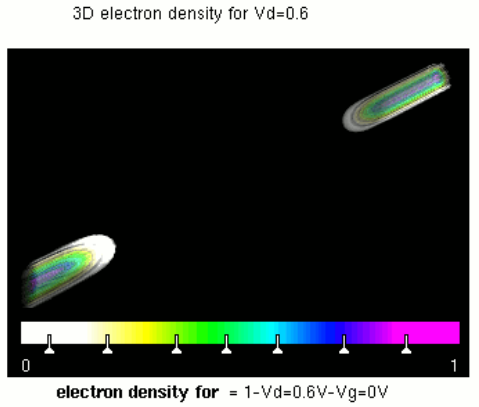

区间1　核心：Vg不足

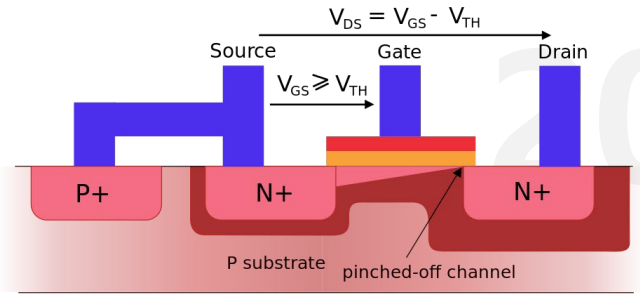



核心：Vd不足

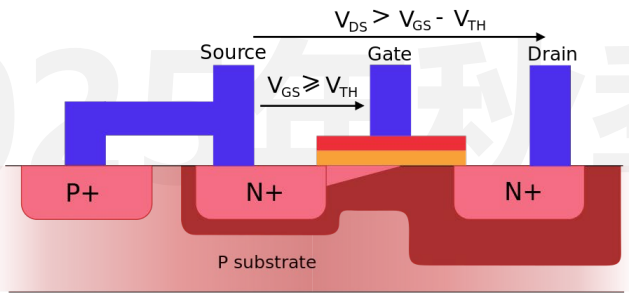
Linear operating region (ohmic mode)



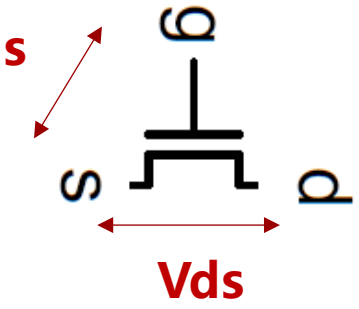




Saturation mode at point of pinch-off




Saturation mode



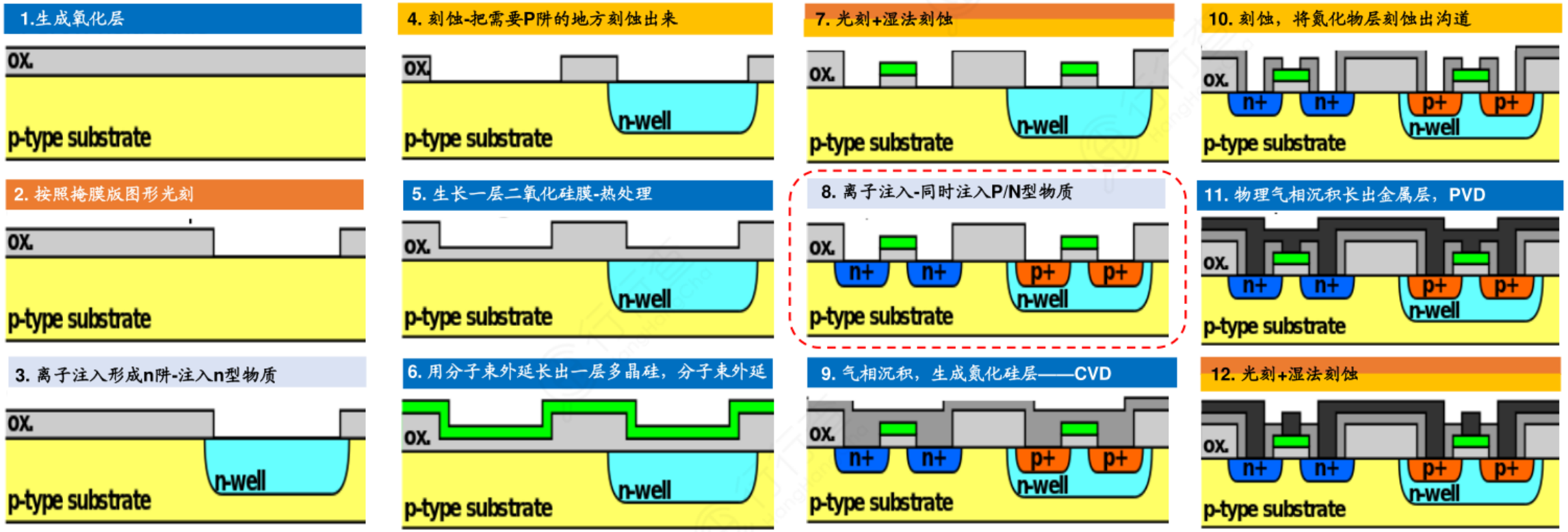

区间3　核心：Vd过大

Vth：MOSFET的阈值电压
与工艺相关

1、断开区间：Vgs < Vth　2、线性区间：Vds < Vgs - Vth　3、饱和区间：Vds > Vgs - Vth

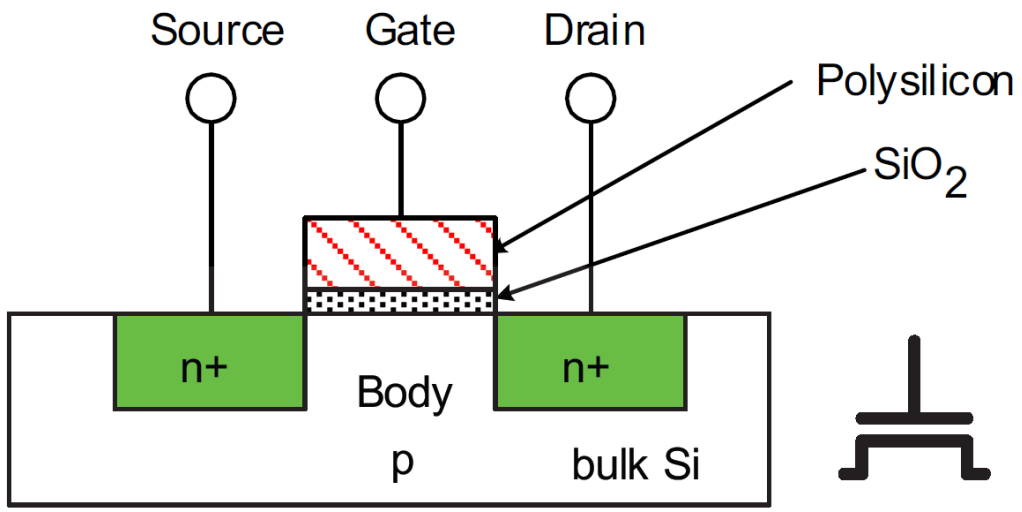
# MOSFET晶体 – 现代芯片的基石

- MOSFET的制造过程总结

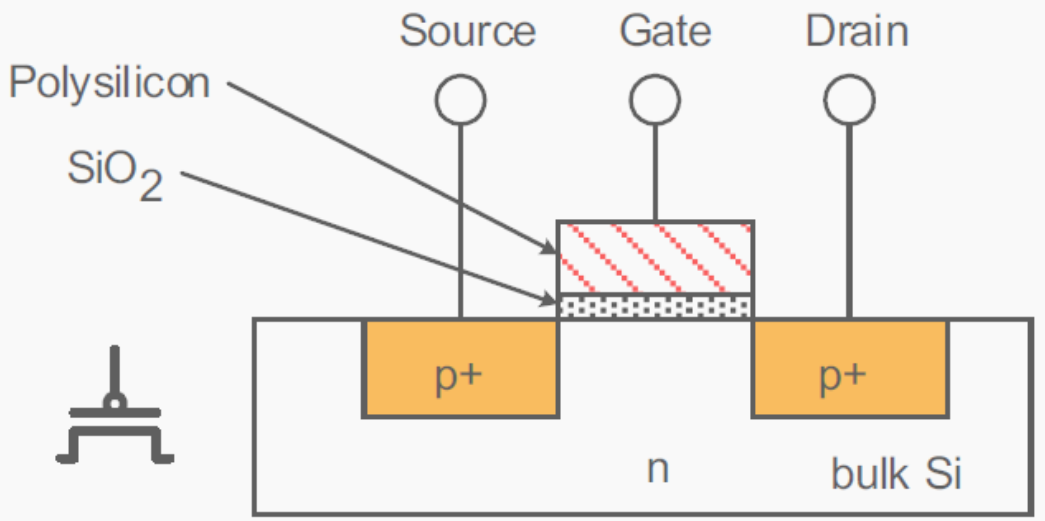


# MOSFET晶体 – 现代芯片的基石

- NMOS与PMOS




NMOS



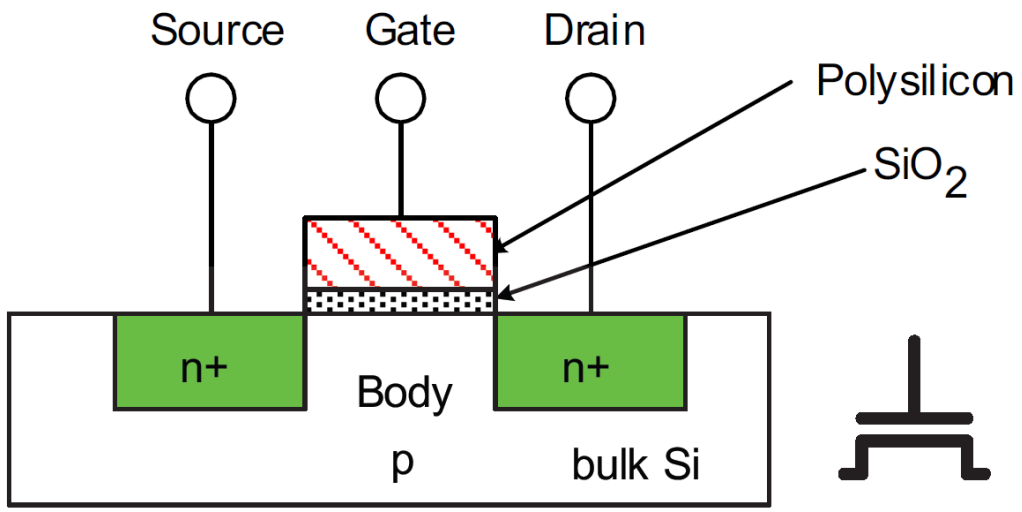

PMOS

当栅极处于低电压：

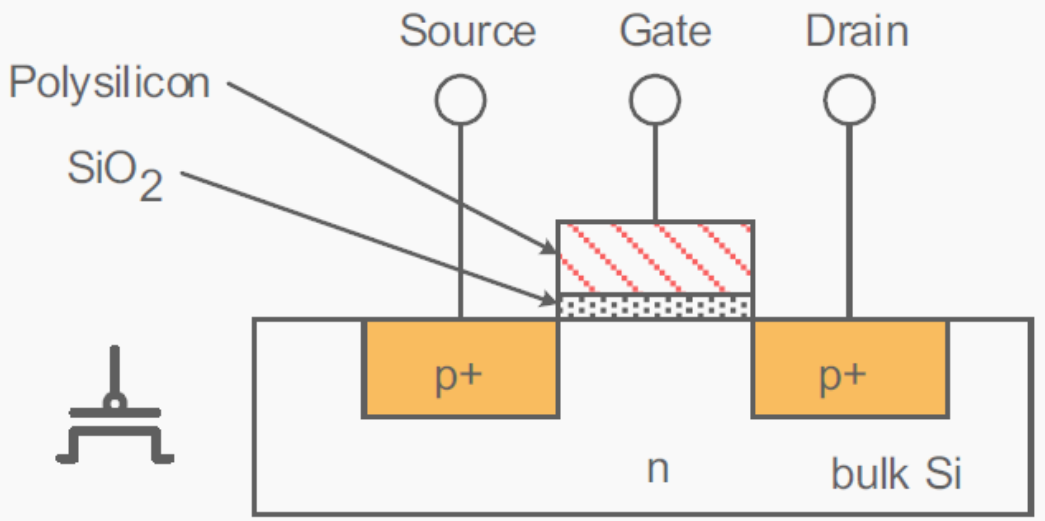
§ 体端（body）接地，即低电压
§ 栅极（gate）下通道中不存在导电通路
§ 无电流流动，晶体管关闭

# MOSFET晶体 – 现代芯片的基石

- NMOS与PMOS




NMOS




PMOS

当栅极处于高电压：

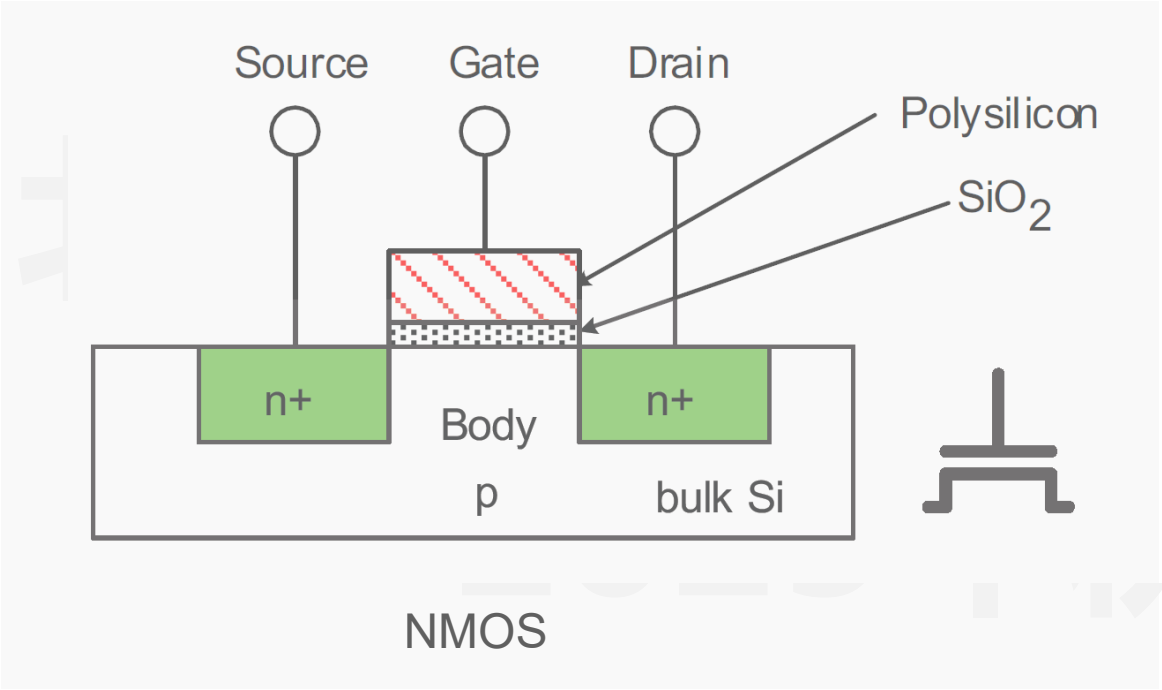
§MOS 电容栅极上有正电荷，体端则有负电荷

§将栅极下的通道反转为 n 型

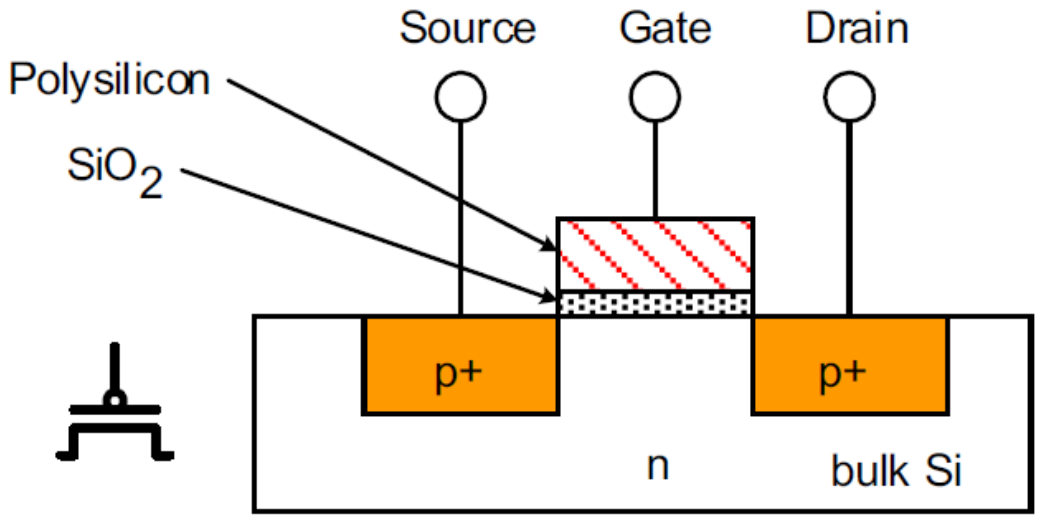
§现在电流通过通道流过n型硅，晶体管导通

# MOSFET晶体 – 现代芯片的基石

- NMOS与PMOS




NMOS



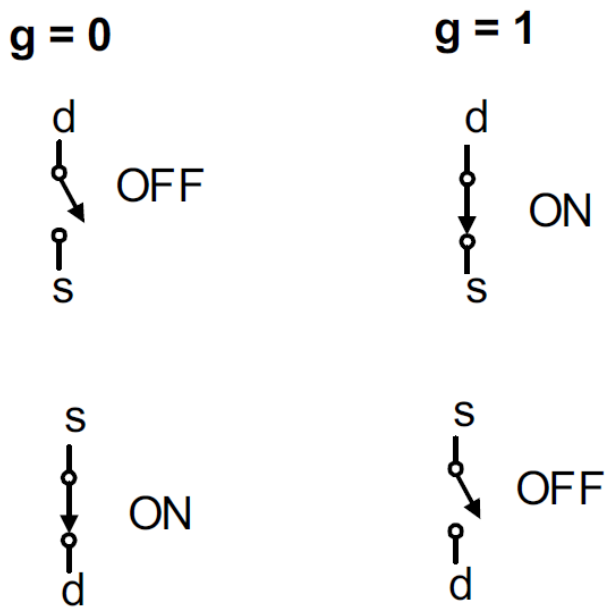

PMOS

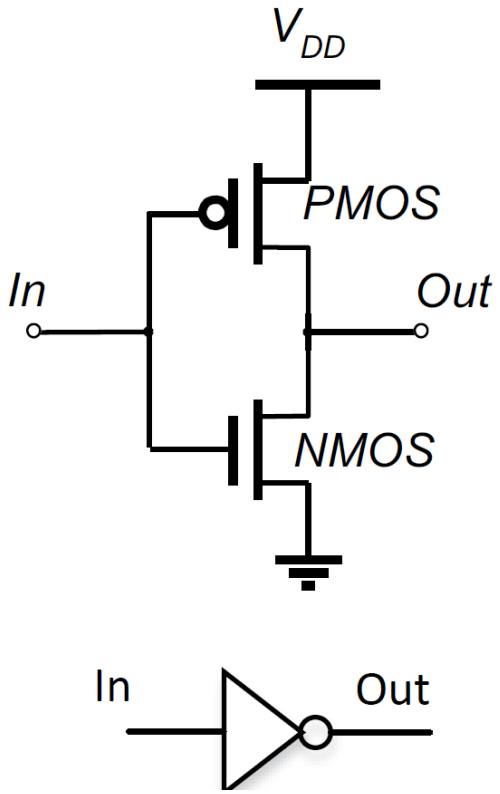
类似NMOS，但掺杂和电压相反

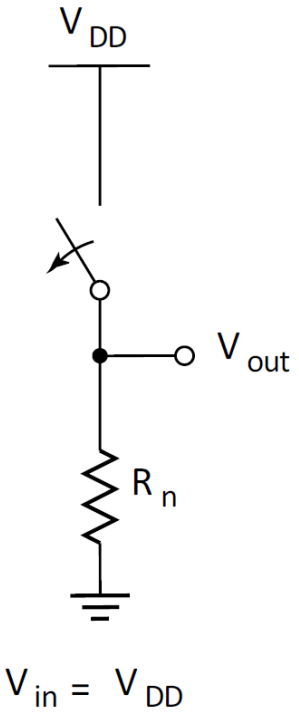
- §体端与高电平相连（VDD）
- §**栅极低电平**：晶体管导通
- §**栅极高电平**：晶体管关闭

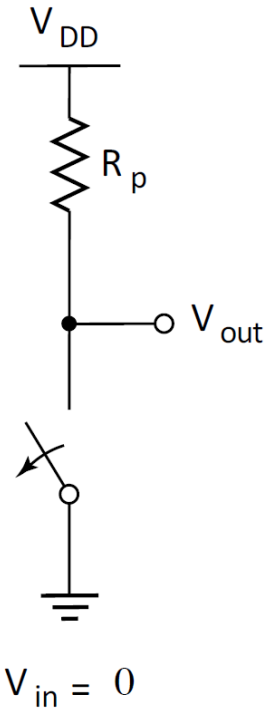
# MOSFET作为开关的行为级模型

- NMOS、PMOS和简单反相器

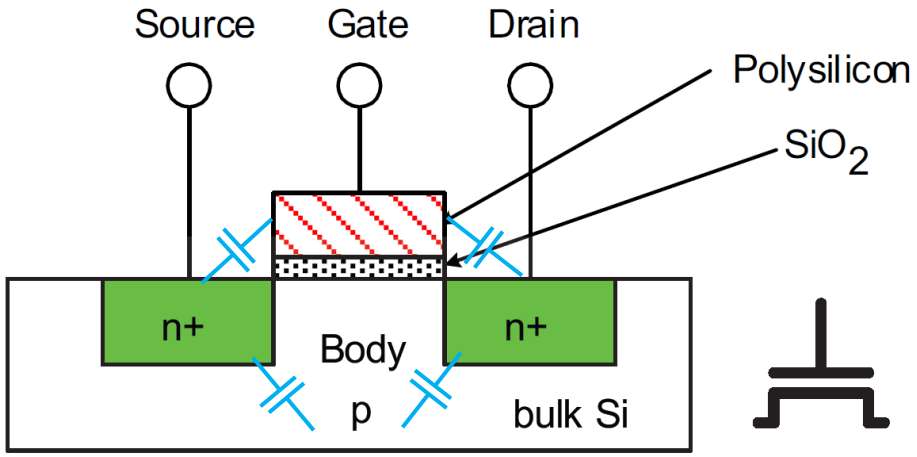



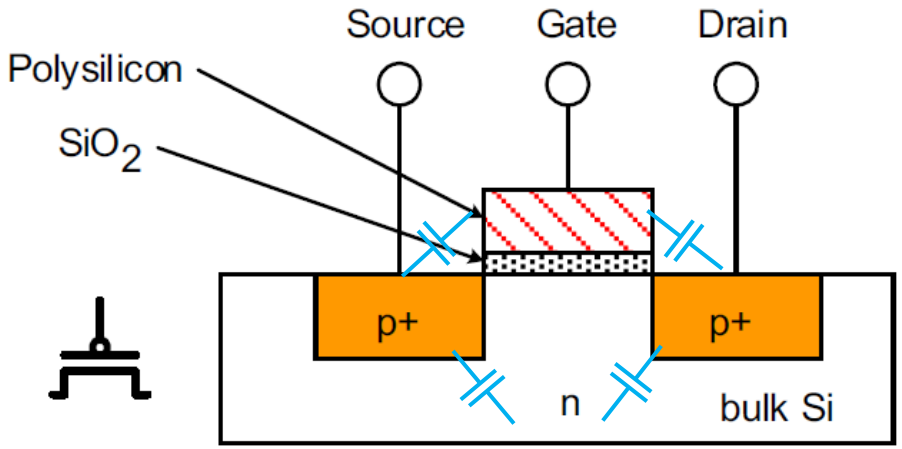



# MOSFET晶体 – 现代芯片的基石

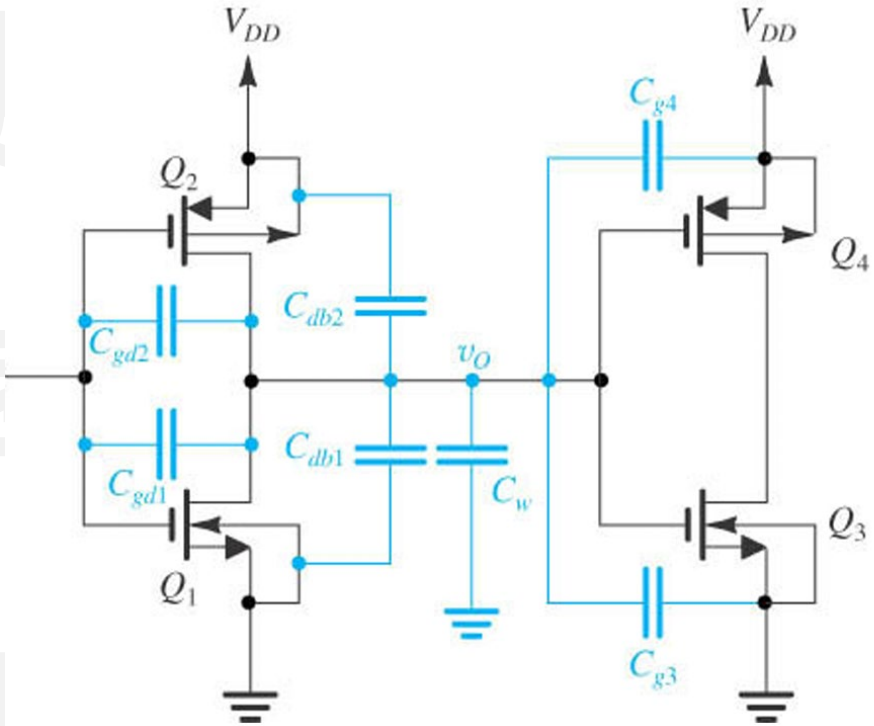
- NMOS与PMOS的寄生电容




NMOS




PMOS



# 逻辑代数简介

- **逻辑函数的定义**

(1) 逻辑变量和逻辑函数的取值只有0和1。（定义域，值域：{0,1}）

(2) 函数和变量之间的关系由“**与、或、非**”三种基本运算决定。

设某一逻辑电路的输入为$A_1A_2……A_n$，输出函数为F，当$A_1A_2……A_n$的值确定之后，F的值就唯一的确定了。称F为$A_1A_2……A_n$的逻辑函数。记为：

$F = f(A_1A_2……A_n)$

- **与（AND）** $\cdot$
- **或（OR）** $+$
- **非（NOT）** $\overline{\ }$

# 逻辑代数简介

- 逻辑函数定义与基本公理

一、真值表(便于直观的观察变量和函数之间的关系)

二、逻辑函数表达式(便于获得逻辑电路图)　　A+BC

| $A$ | $B$ | $A\cdot B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

## 一、公理（Axioms）

$$0\cdot 0=0 \qquad 1+1=1$$
$$0\cdot 1=0 \qquad 1+0=1$$
$$1\cdot 0=0 \qquad 0+1=1$$
$$\bar{0}=1 \qquad \bar{1}=0$$

· 与（AND）

\+ 或（OR）

¯ 非（NOT）

跟数值函数一样，逻辑函数表达式由：运算符，变量，常量，运算次序所构成

# 逻辑代数简介

- 逻辑函数的基本定理

## 二、公式(可由公理推出)

$0 \cdot A = 0$　　$1 + A = 1$　　(有界性Boundedness )

$1 \cdot A = A$　　$0 + A = A$　　(单位元素Element )

$A \cdot A = A$　　$A + A = A$　　(等幂性Idempotency )

$A \cdot \overline{A} = 0$　　$A + \overline{A} = 1$　　(互补性Complement)

$A \cdot B = B \cdot A$　　$A + B = B + A$　交换律（Commutativity)

· 与（AND)

+ 或（OR)

− 非（NOT)

# 逻辑代数简介

- 逻辑函数的基本定理

## 二、公式(可由公理推出)

A(BC)=(AB)C=(AC)B

A+(B+C)=(A+B)+C=(A+C)+B

A(B+C)=AB+AC　　乘法分配律

A+BC=(A+B)·(A+C)　　加法分配律

证:右式=AA+AC+AB+BC=A+AC+AB+BC

=A(1+C+B)+BC=A+BC=左式

- 与（AND）
+ 或（OR）
– 非（NOT）

北京大学 PEKING UNIVERSITY

- 逻辑函数的基本定理

## 二、公式(可由公理推出)

$$\overline{AB} = \overline{A} + \overline{B} \qquad \overline{A+B} = \overline{A} \cdot \overline{B}$$

证：用真值表法证明 $\overline{AB} = \overline{A} + \overline{B}$

| $A$ | $B$ | $A \cdot B$ | $\overline{A \cdot B}$ | $\overline{A}$ | $\overline{B}$ | $\overline{A} + \overline{B}$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

摩根律 (De Morgan’s Laws)

· 与（AND）

+ 或（OR）

– 非（NOT）

# 逻辑代数简介

- 逻辑函数的基本定理

## 二、公式(可由公理推出)

$$\overline{ABCD\cdots} = \overline{A} + \overline{B} + \overline{C} + \overline{D} + \cdots$$

$$\overline{A + B + C + D + \cdots} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} \cdots$$

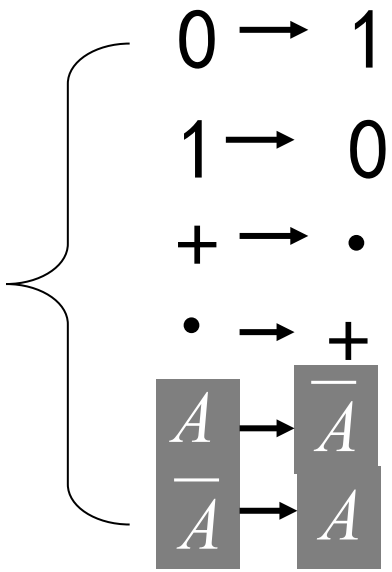
摩根律 (De Morgan’s Laws)

- · 与（AND）
- + 或（OR）
- – 非（NOT）

- 逻辑函数的基本定理

## 三、反演规则

即由 $F(A,B,C\cdots)$ 求 $\overline{F}(A,B,C\cdots)$




0，1 互换，

与，或互换

$A$，$\overline{A}$互换

使用反演规则时，应注意保持原函数式中的运算符号的优先顺序不变(先与后或，除非另加括号)

· 与（AND）

\+ 或（OR）

– 非（NOT）

# 逻辑代数简介

- 逻辑函数的基本定理

例1：$F = \overline{A} + \overline{B}(C + \overline{D}E)$　　$\overline{F} = A \cdot [B + \overline{C}(D + \overline{E})]$

例2：$F = \overline{A\overline{B} + C}$　①$\overline{F}$ = $A\overline{B}$ + C (直接去掉反号)

②　$\overline{F} = \overline{\overline{A\overline{B} + C}} = \overline{(\overline{A} + B) \cdot \overline{C}} = \overline{\overline{A} + B} + C = A\overline{B} + C$

(原函数两边取非，摩根律化简)

其实反演规则就是摩根律的推广。

例3：　$F = A\overline{B} + B(\overline{A} + C)$

①　按反演规则可直接写出：

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

# 逻辑代数简介

- 逻辑函数的简化案例

例1:

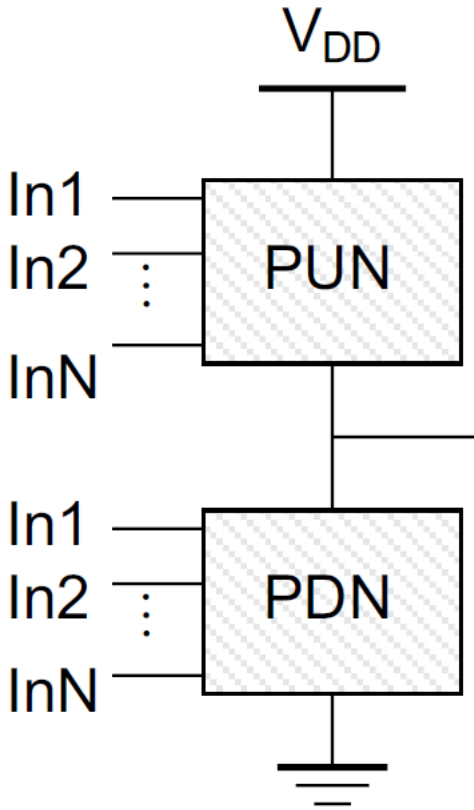
$F = (A+B)(A+\overline{B})(B+C)(A+C+D)$ 求对偶式

$F\not\subset = AB + A\overline{B} + BC + ACD = A + BC + ACD$

$= A + BC$ 提取共同项A 提取共同项A

# 经典静态逻辑电路

- NMOS、PMOS可以组成PDN和PUN




PMOS only

**上拉逻辑设计：**

Pull-up network: make a connection from $V_{dd}$ to F when F(In1,In2…)=1

F(In1,In2,…InN)

NMOS only

Pull-down network: make a connection from Ground to F when F(In1,In2…)=0

**下拉逻辑设计：**

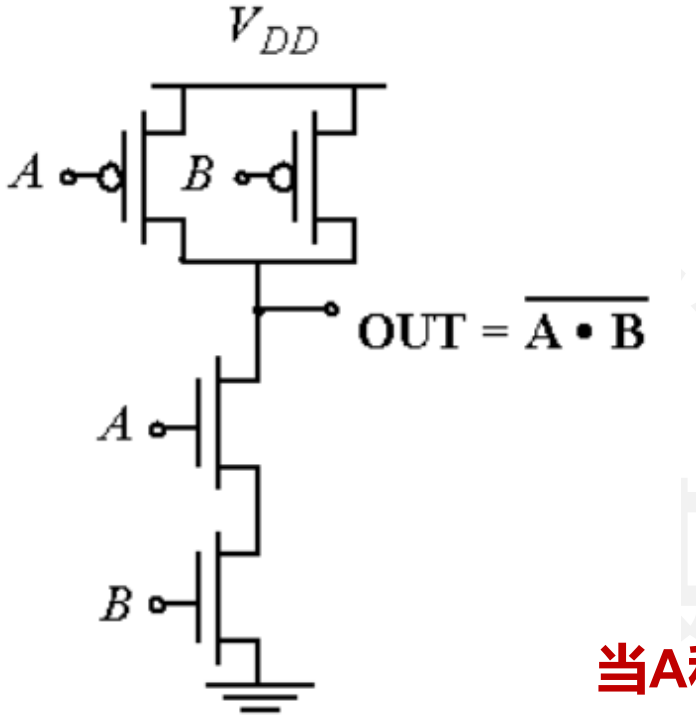
**PUN and PDN are *dual* networks**

# 经典静态逻辑电路

- 由PDN和PUN组成的NAND门电路

| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Truth Table of a 2 input NAND gate




$OUT = \overline{A \bullet B}$ NAND门

PDN: Connects OUT to ground when A●B=1

PUN: Connects OUT to $V_{dd}$ when $\overline{A} + \overline{B} = 1$

So OUT = Complement of PDN function

Also OUT = PUN function with each input inverted

**当A和B同为1时，输出为0**

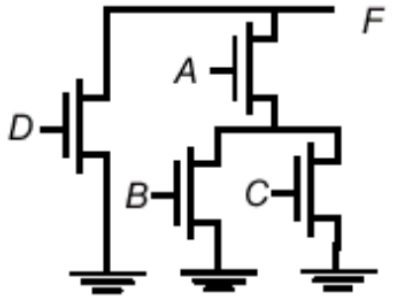
**当A和B有0时，输出为1**

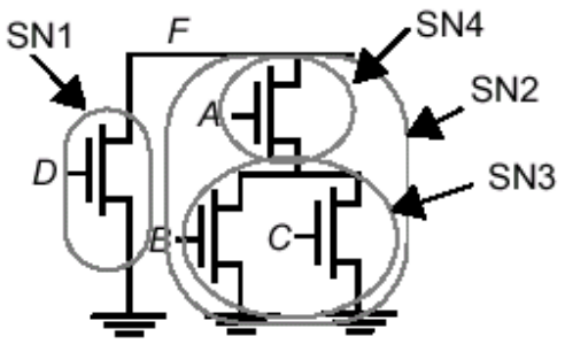
# 经典静态逻辑电路

- 由PDN和PUN组成的复杂静态逻辑电路

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

**什么情况下F为0 -> D+A(B+C) = 1**




忽略取非操作，直接构建PDN

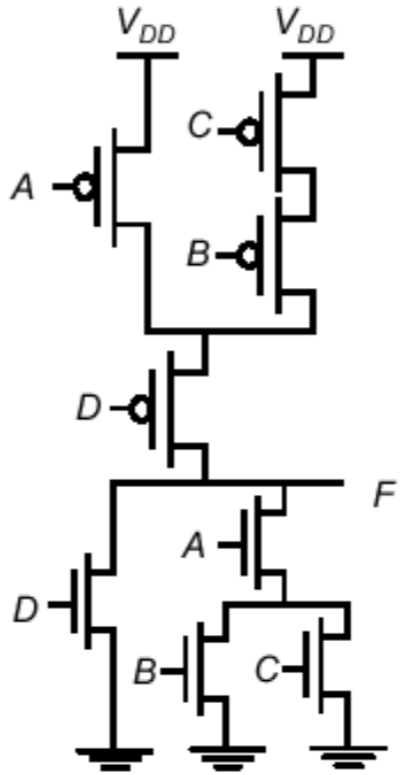
**Ex：D+X意味着D与X并联**

构建PDN的亚网络

**在SN3内，B和C是并联的，**

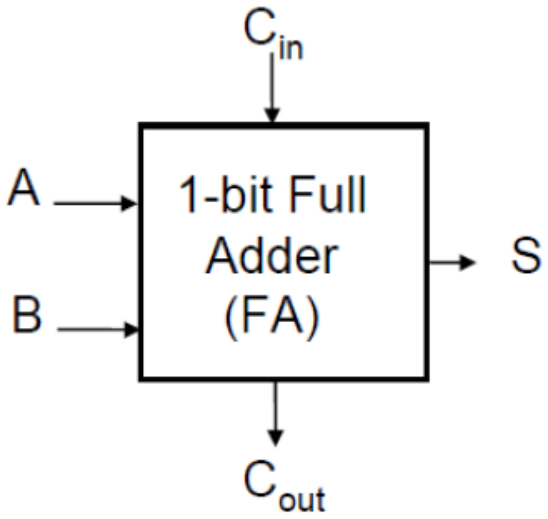
则在PUN中，他们串联

自下向上构建



# 经典静态逻辑电路

- 简单1bit加法器电路
  - A[n-1:0] + B[n-1:0] = S[n-1:0]



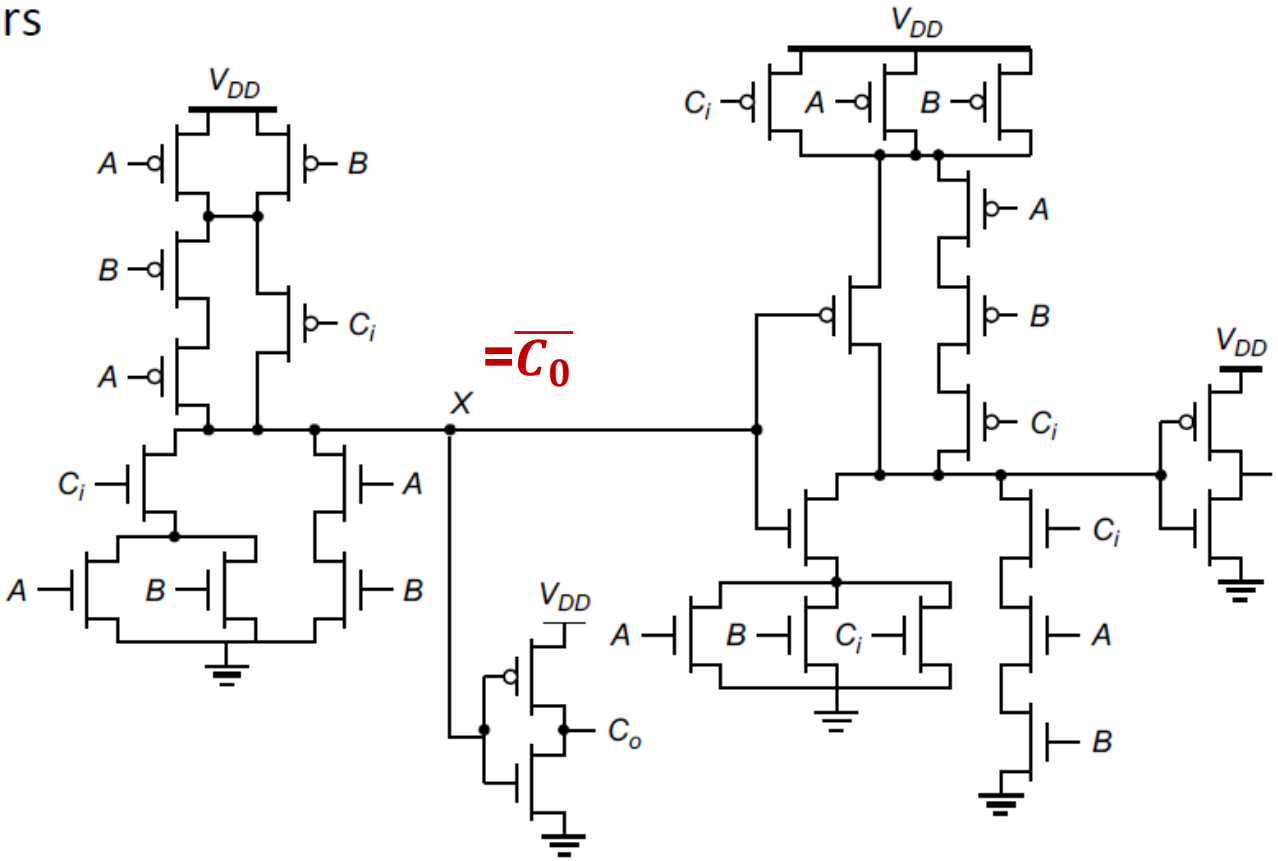

| A | B | $C_{in}$ | $C_{out}$ | S | carry status |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | kill |
| 0 | 0 | 1 | 0 | 1 | kill |
| 0 | 1 | 0 | 0 | 1 | propagate |
| 0 | 1 | 1 | 1 | 0 | propagate |
| 1 | 0 | 0 | 0 | 1 | propagate |
| 1 | 0 | 1 | 1 | 0 | propagate |
| 1 | 1 | 0 | 1 | 0 | generate |
| 1 | 1 | 1 | 1 | 1 | generate |

**单比特加法器逻辑设计**

- $C_o = AB + BC_i + AC_i = AB + (A + B)C_i$
- $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A + B + C_i)$
- $G = AB,\ K = \overline{A}\,\overline{B},\ P = A \oplus B$

# 经典静态逻辑电路

- 简单1bit加法器电路
  - $C_o = AB + BC_i + AC_i = AB + (A+B)C_i$
  - $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A+B+C_i)$
  - 28 transistors

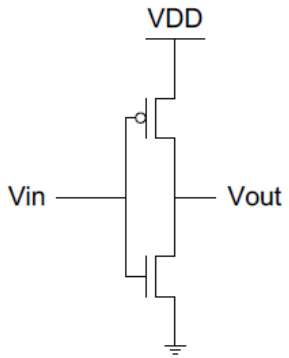


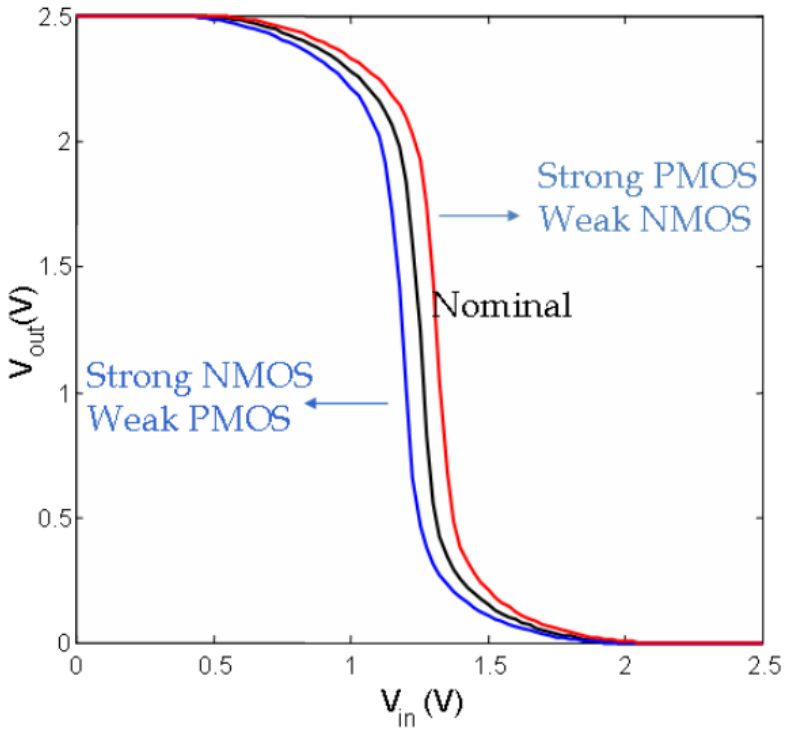
# 目 录

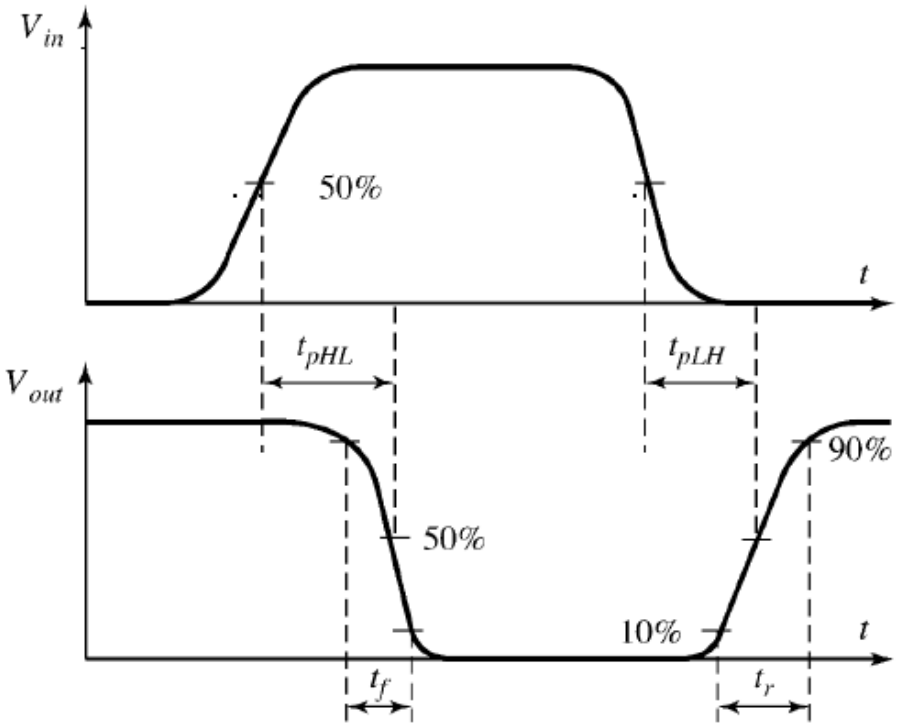
CONTENTS

# 什么是电路的延迟

- 以Inverter反相器为例



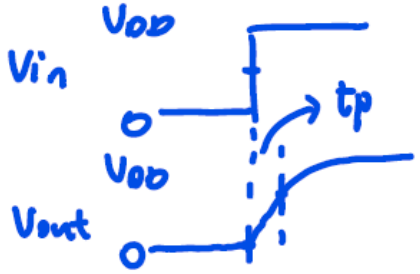



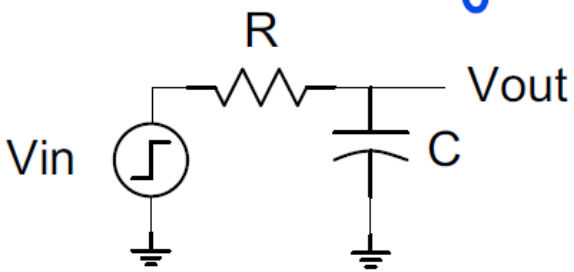



# 什么是电路的延迟

- 一阶RC延迟分析

## A First-Order RC Network




$$V_{out}(t) = (1 - e^{-t/\tau})V_{DD} = \frac{1}{2}V_{DD}$$

$$e^{-t/\tau} = \frac{1}{2}$$

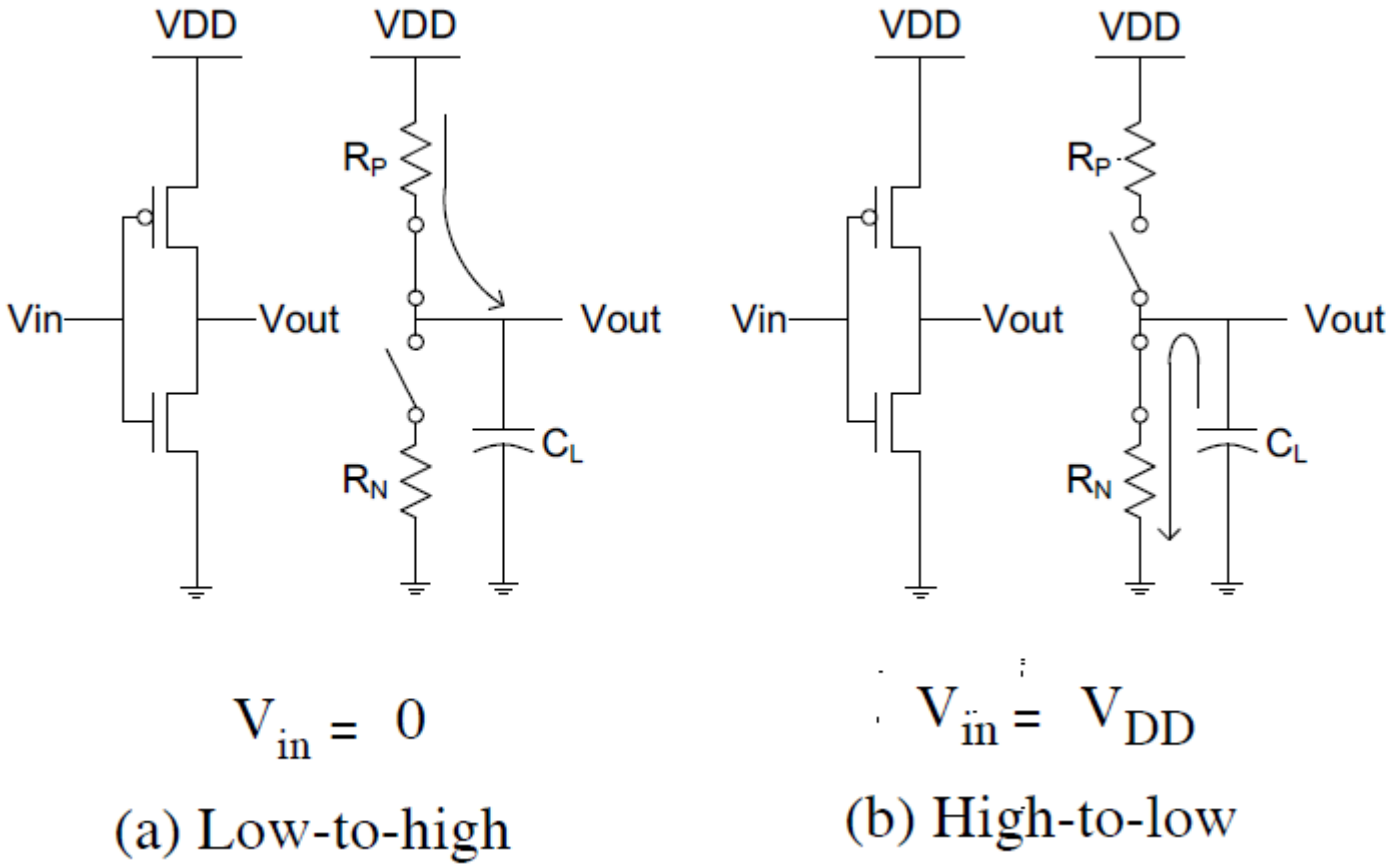
$$-t/\tau = -\ln 2$$

$$\tau = R \times C$$

$$t_p = \ln(2)\tau = 0.69R \times C \qquad t = \ln(2)\tau$$

# 反相器延迟

- 利用一阶RC延迟分析方法




$V_{in} = 0$

(a) Low-to-high

$V_{in} = V_{DD}$

(b) High-to-low

$$t_{pHL} = f(R_N \times C_L)$$

$$t_{pHL} = 0.69 R_N \times C_L$$

$$t_{pLH} = 0.69 R_P \times C_L$$

# 降低延迟的设计方法

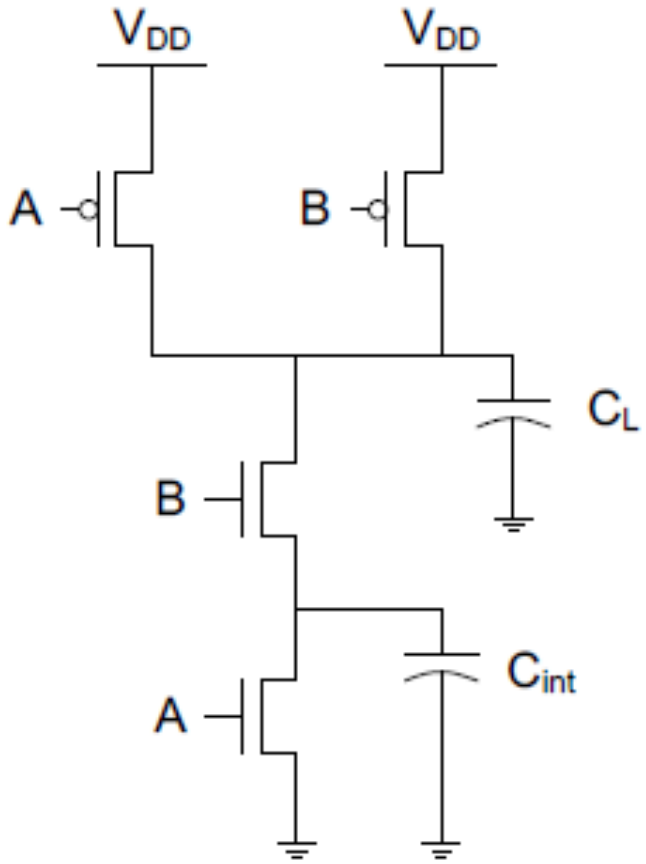
- 利用一阶RC延迟分析方法

$$t_{pHL} = f(R_N \times C_L)$$
$$t_{pHL} = 0.69 R_N \times C_L$$
$$t_{pLH} = 0.69 R_P \times C_L$$

- **利用较小的电容 – 降低C**
  - 版图紧凑, 布局合理
  - 保持较短走线&减少diffusion routing
- **增加晶体管尺寸 – 降低 R**
  - 避免self-loading出现，否则会导致寄生电容增大
- **增加电源电压**
  - 同时会影响可靠性与功耗，因而一般不采用

# 输入Pattern对延迟的影响

- 利用一阶RC延迟分析方法

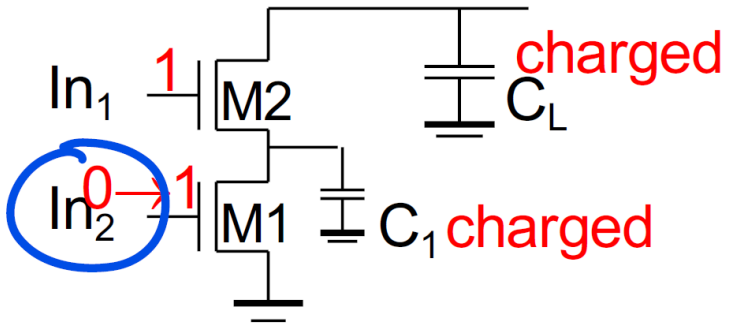



**电路延迟与输入的顺序有关！**

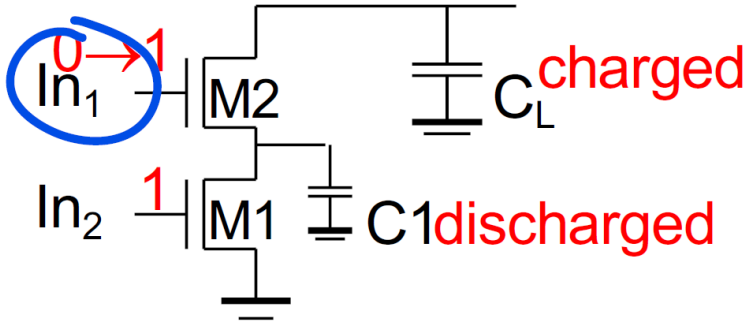
- *Ignore $C_{int}$ for the moment!*
- Low to high transition
  - both inputs go low
    - delay is 0.69 $R_p/2$ $C_L$
  - one input goes low
    - delay is 0.69 $R_p$ $C_L$
- High to low transition
  - both inputs go high
    - delay is 0.69 $2R_n$ $C_L$

# 利用Transistor Ordering提升逻辑速度

- 复杂的Transistor Ordering需要仿真工具支持




延迟由$C_L$与$C_1$放电时间决定




延迟由$C_L$放电时间决定

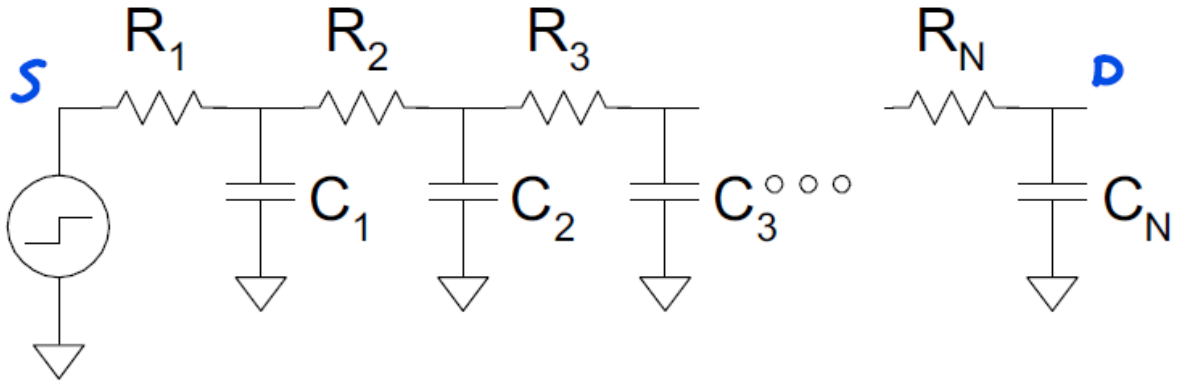
# 逻辑电路的Elmore Delay模型

- 拓展多级的RC模型

  - 导通晶体管看作电阻
  - 电路网络建模为RC阶梯
  - RC阶梯的Elmore延迟
  - Apply to complex gates（i.e.,stacks),also interconnect（later）

$$t_{pd} \approx \sum_{\text{nodes } i} R_{i-to-source} C_i$$

$$= R_1 C_1 + (R_1 + R_2) C_2 + \ldots + (R_1 + R_2 + \ldots + R_N) C_N$$

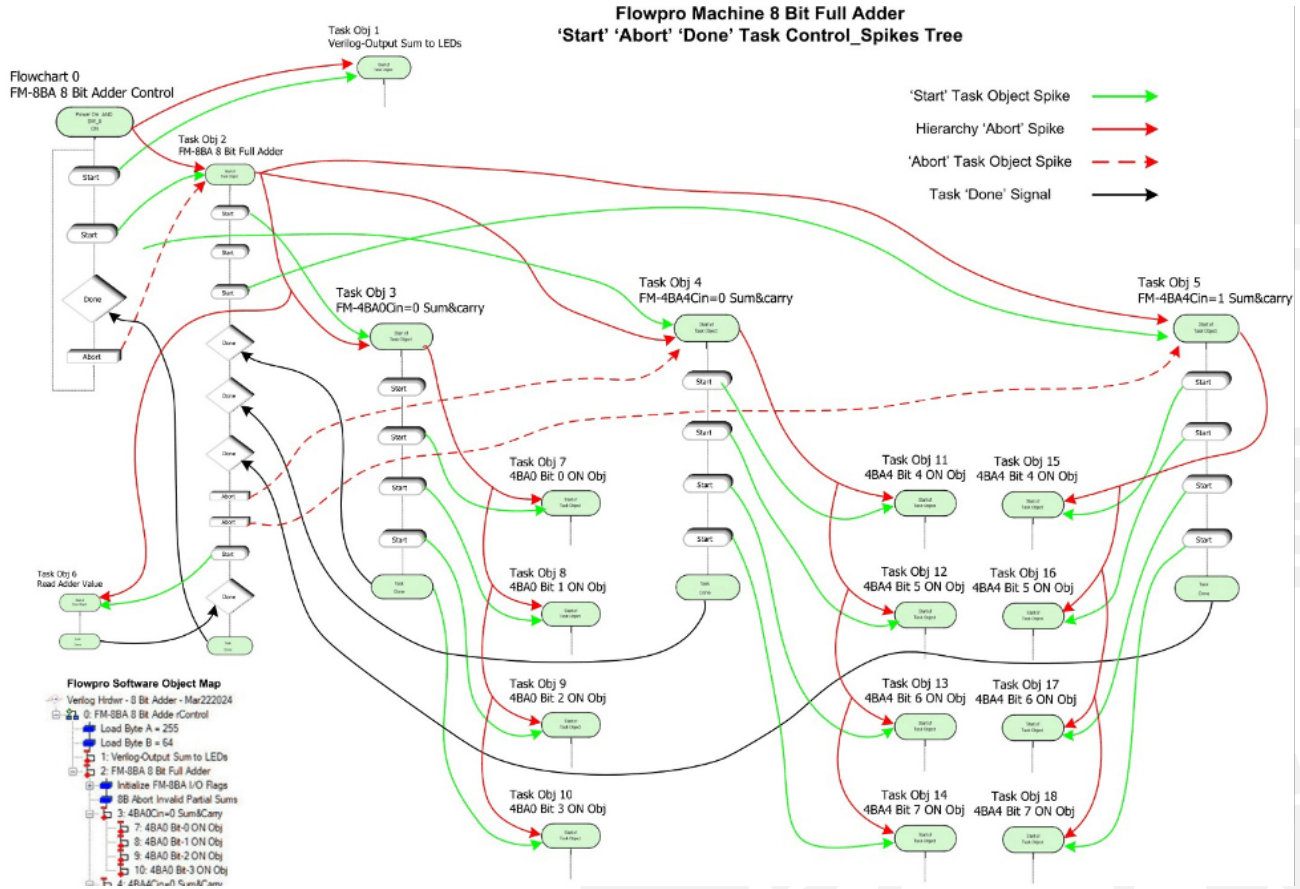


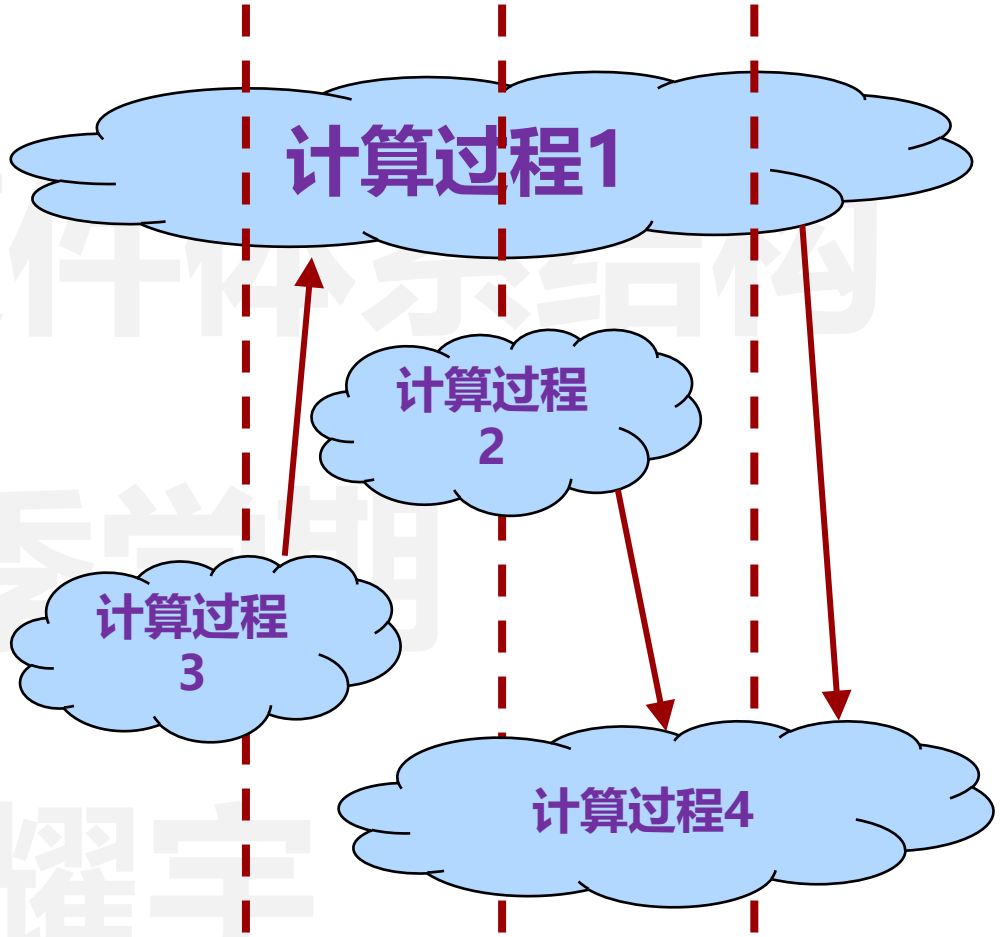
# 目 录

CONTENTS

# 电路时序的基本概念

- 电路为什么需要一个时钟?



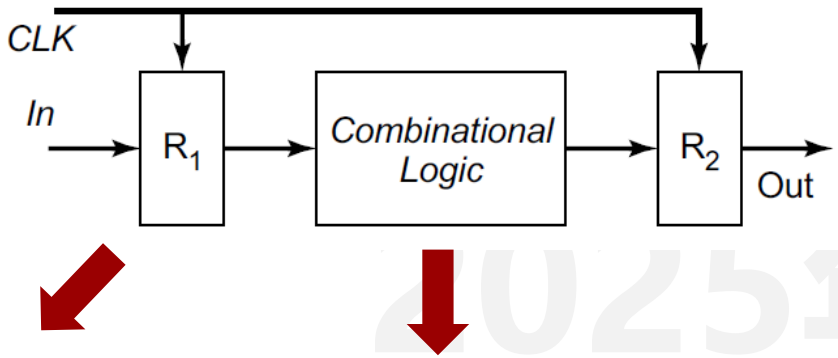




**无时钟：非常难以控制每一个信号的有效时间**

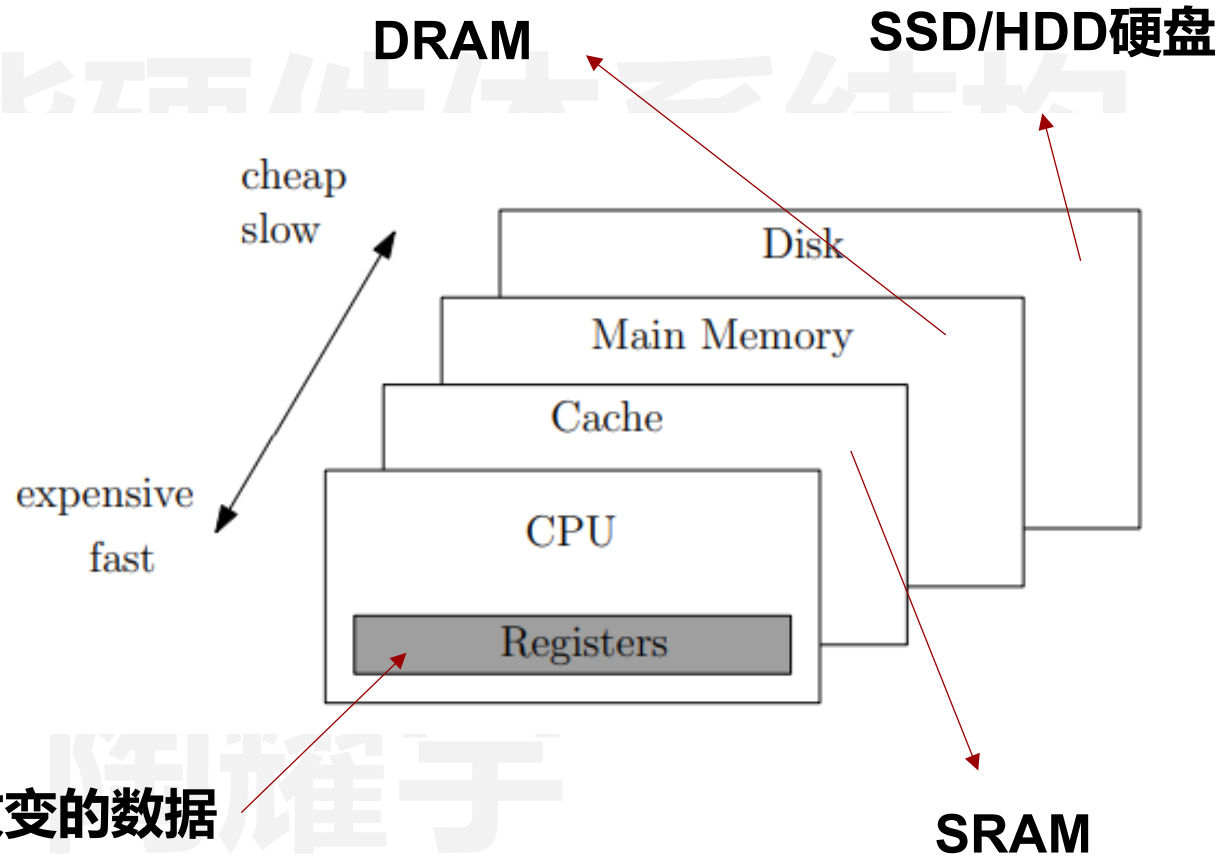
**引入时钟：每隔一段计算将结果同步一次**

# 电路时序的基本概念

- 同步时序（Synchronous Timing）



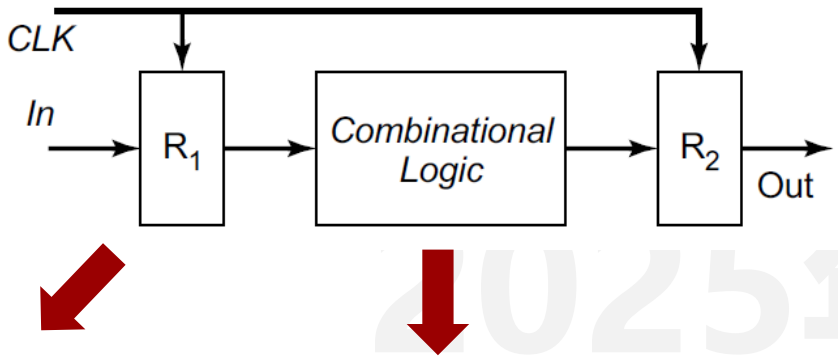

寄存器 (Register)　组合逻辑（各种逻辑门电路）




在芯片内用于暂存随时可能需要改变的数据

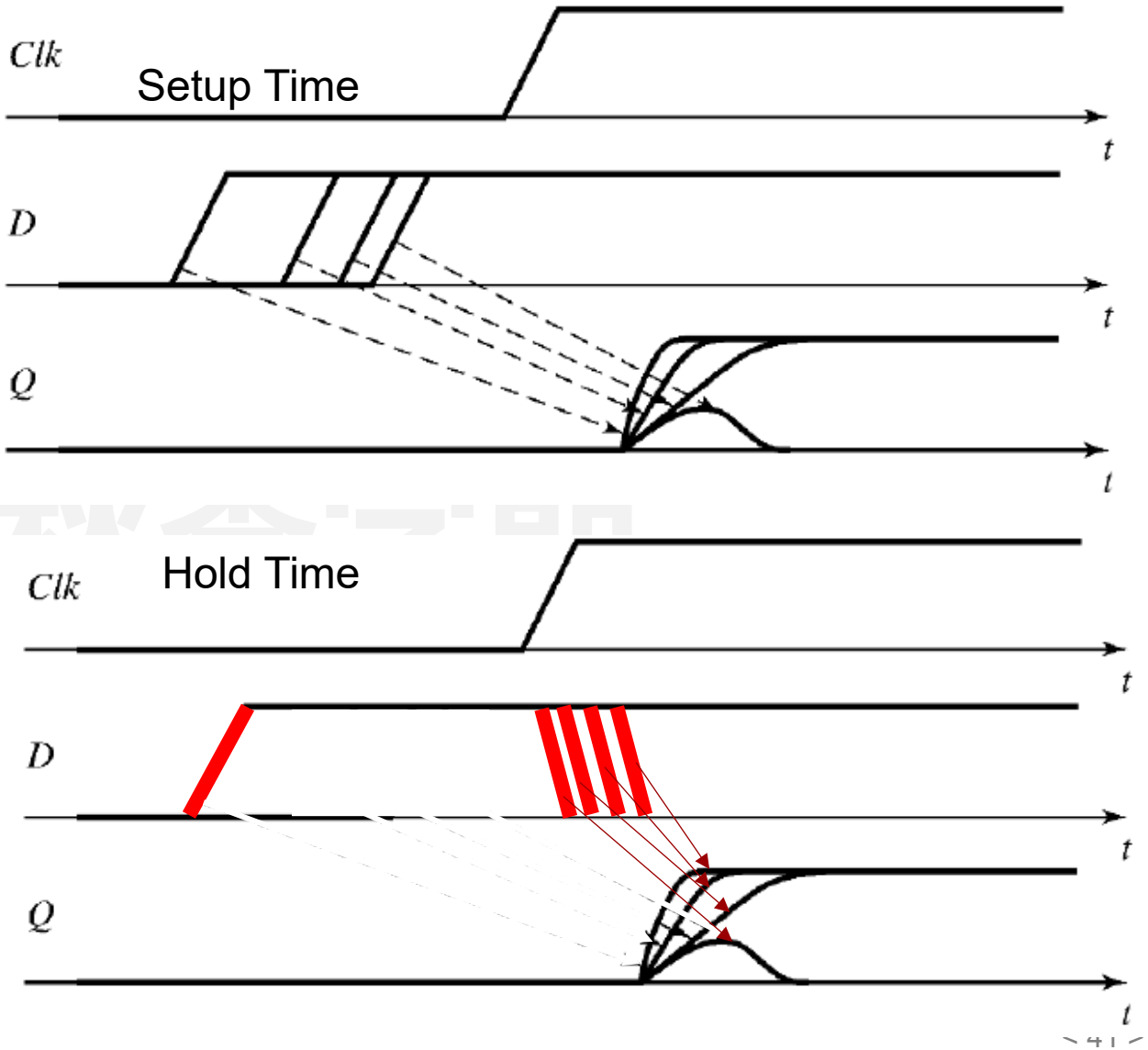
# 电路时序的基本概念

- 同步时序（Synchronous Timing）



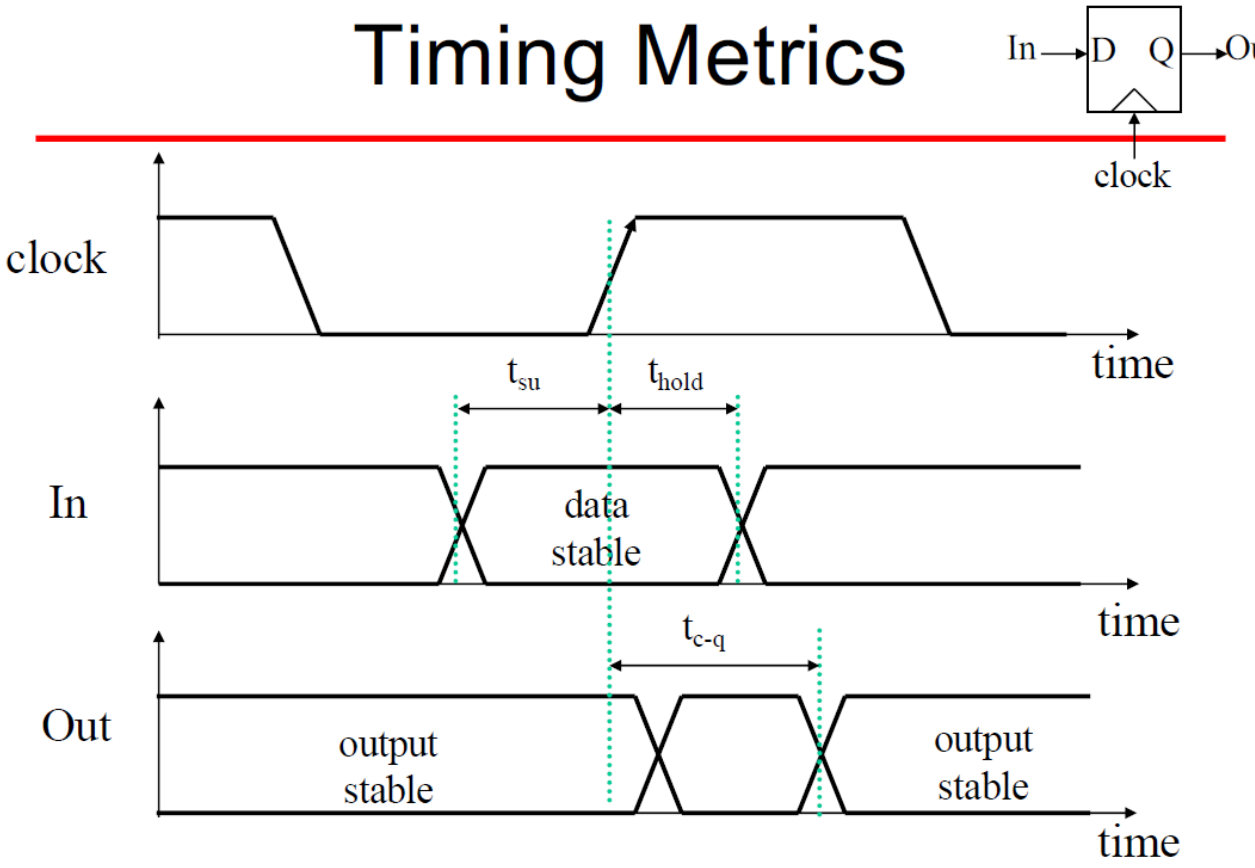

寄存器　组合逻辑（各种逻辑门电路）
(Register)

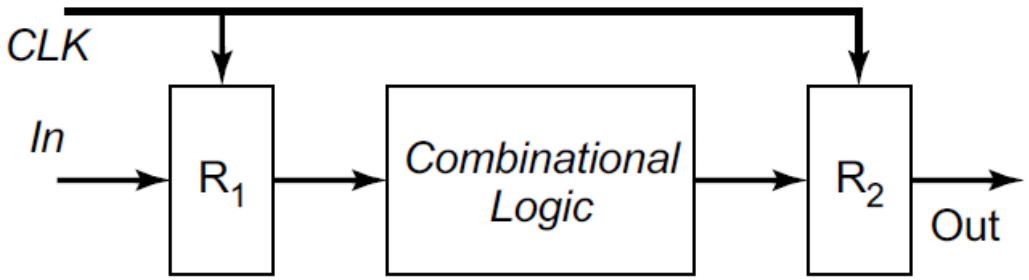
用于暂存随时可能需要改变的数据



# 电路时序的基本概念

- 同步时序（Synchronous Timing）




$$T_{c\text{-}q} + t_{plogic,min} \geq t_{hold}$$

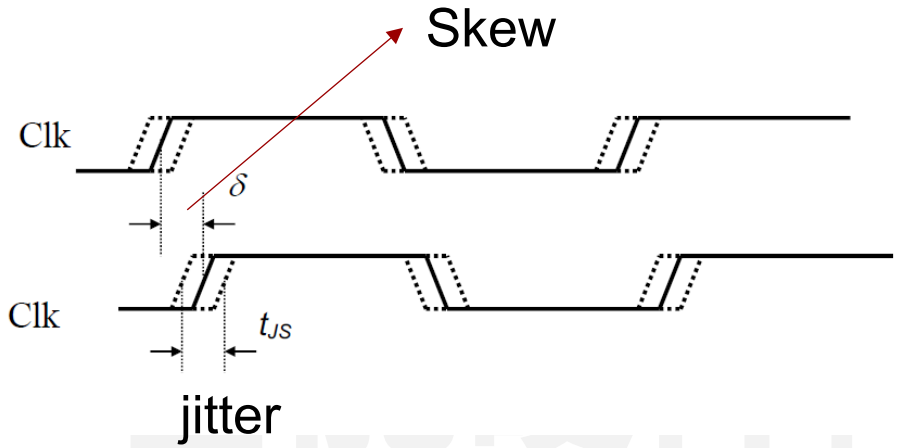
min

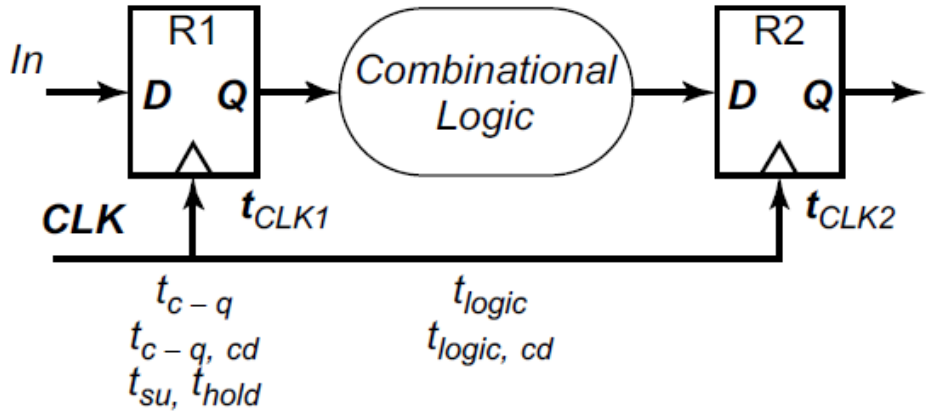
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

max

# 电路时序的基本概念

- 时钟的不稳定性

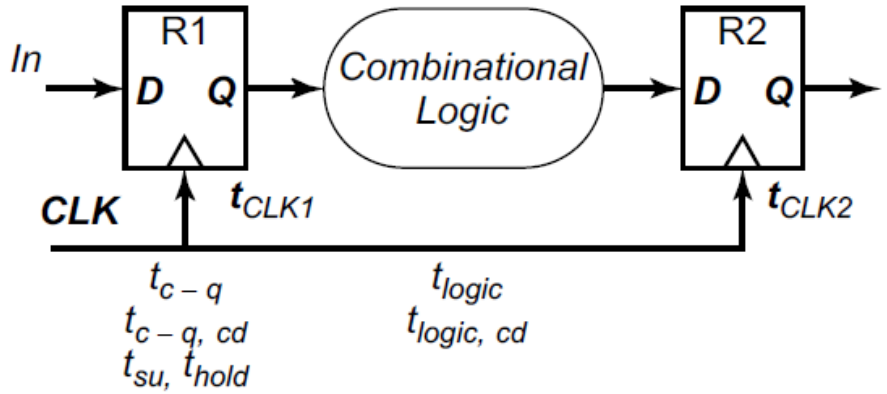



Minimum cycle time:

$T \geq t_{c\text{-}q} + t_{su} + t_{logic} - \delta$

**最坏情况为接收边沿过早到达（negative δ）**




*Hold time constraint:*

$t_{(c\text{-}q,\ cd)} + t_{(logic,\ cd)} > t_{hold} + \delta$

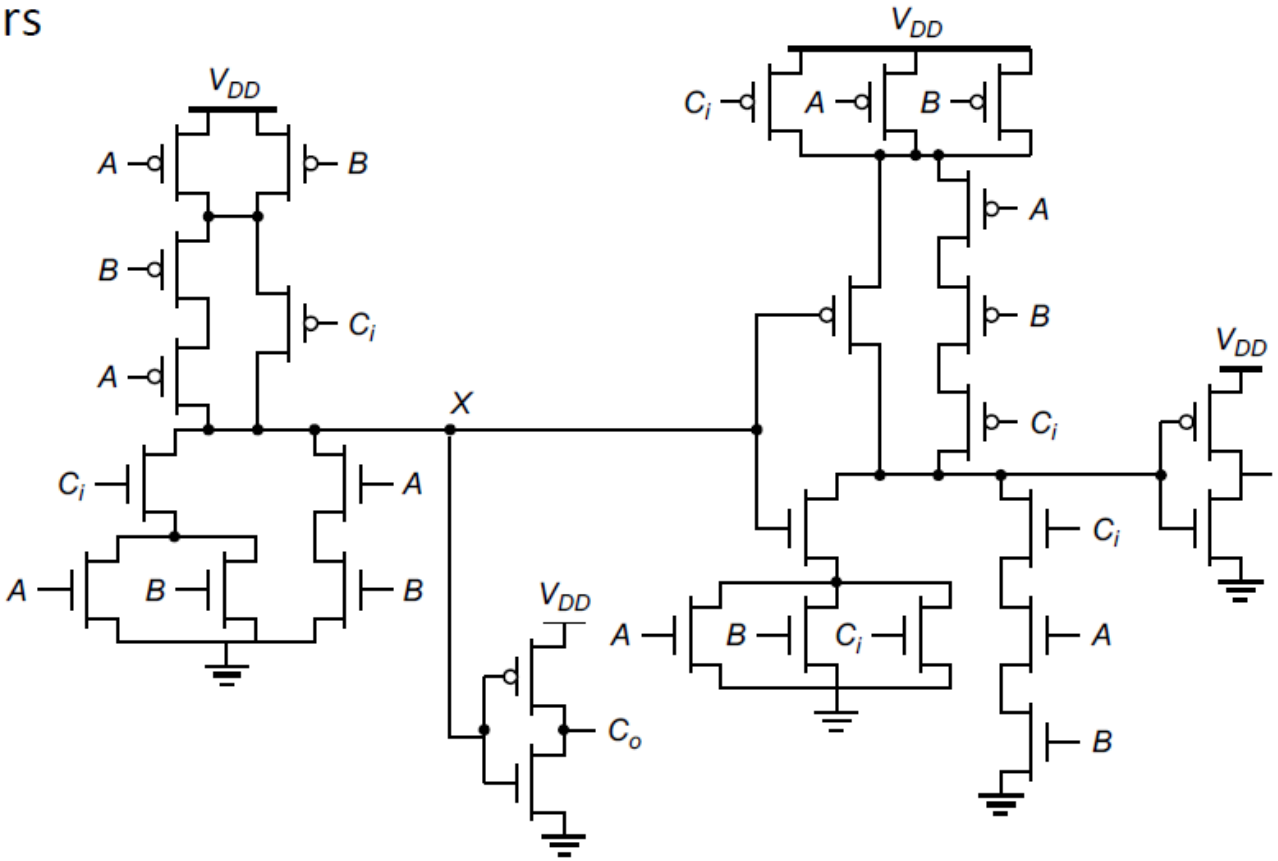
最坏情况为接收边沿过晚到达（正偏差）
数据和时钟之间的竞争
**Cd：contamination delay（最快可能延迟）**

# 目 录

CONTENTS

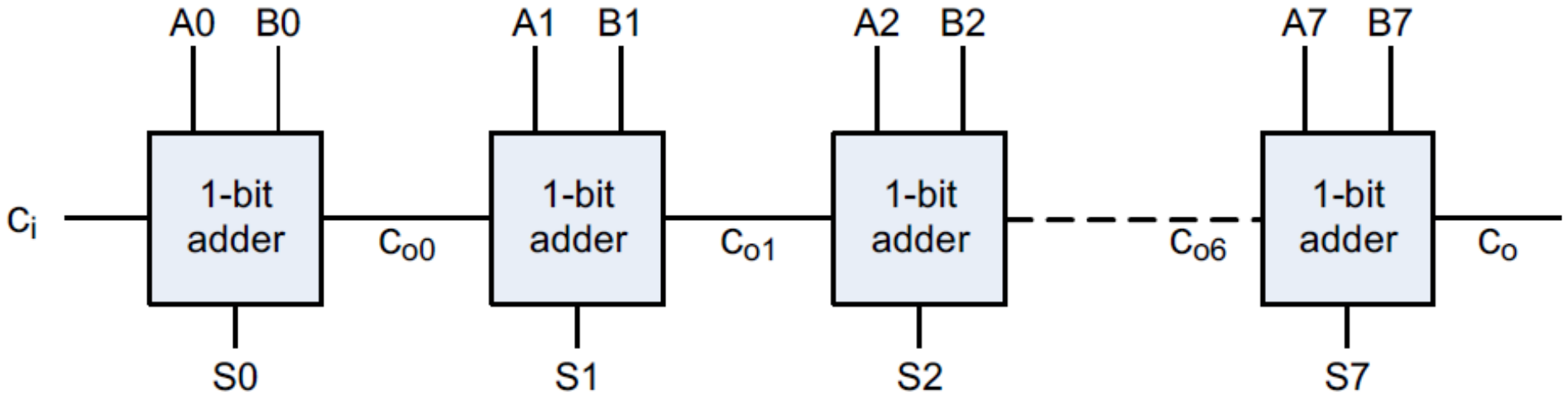
# 加法器设计

- 简单1bit加法器电路
  - $C_o = AB + BC_i + AC_i = AB + (A + B)C_i$
  - $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A + B + C_i)$
  - 28 transistors



# 加法器设计

- Ripple Carry加法器电路




最差延迟与比特数呈线性关系

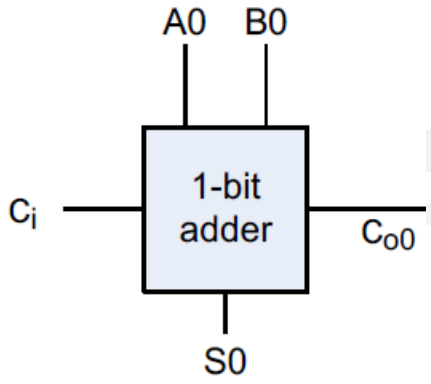
$$t_d = O(N)$$

$$t_{adder} = (N\text{-}1)t_{carry} + t_{sum}$$

**目标：设计拥有最快可能进位路径的电路**

# 加法器设计

- 基于PGK的加法器设计方法




Generate (G) = AB

Propagate (P) = A ⊕ B

$$C_o(G,P) = G + PC_i$$

$$S(G,P) = P \oplus C_i$$

$P = A \oplus B$

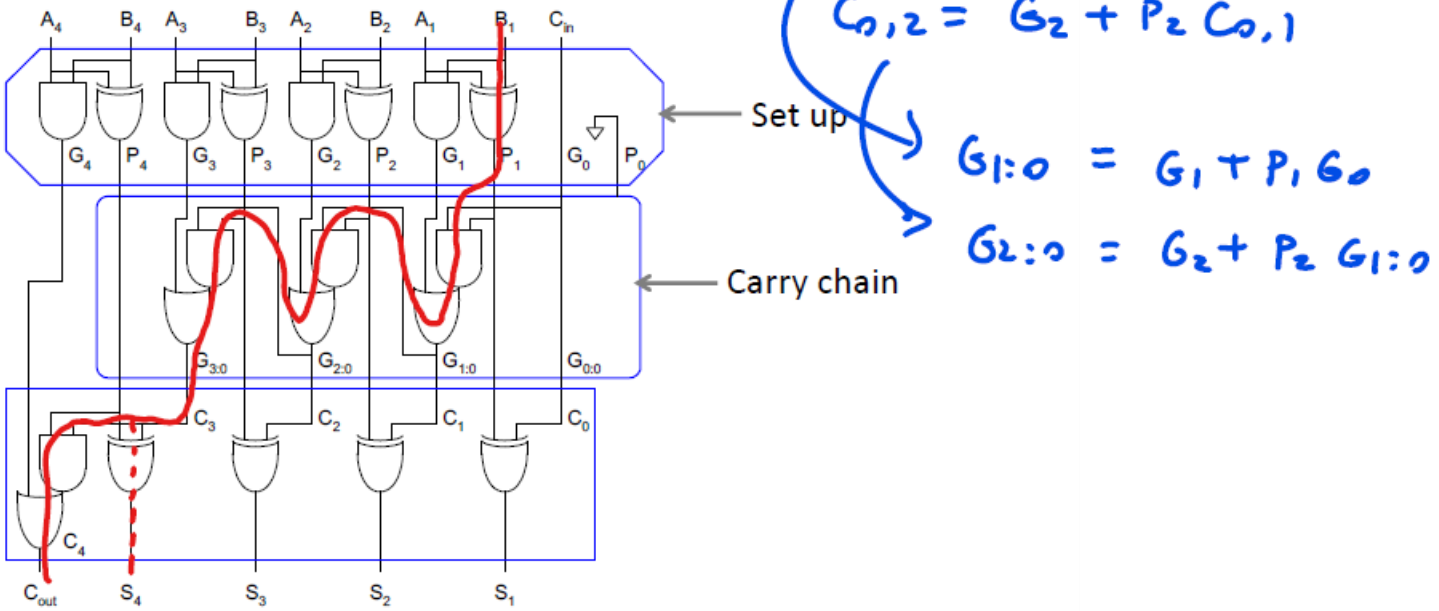
$P = A + B$

- Generate: $C_{out}$ = 1 independent of $C_{in}$
  - G = A • B
- Propagate: $C_{out}$ = $C_{in}$
  - P = A ⊕ B
- Kill: $C_{out}$ = 0 independent of $C_{in}$
  - K = ~A • ~B

- 基于PGK的加法器设计方法

## Carry-Ripple using P and G

$$C_{i:0} = G_i + P_i \cdot C_{i-1:0}$$
$$G_{0:0} = C_{in}$$
$$P_{0:0} = 0$$
$$C_{out,i} = G_{i:0}$$




$C_{0,1} = G_1 + P_1 C_{in}$

$C_{0,2} = G_2 + P_2 C_{0,1}$

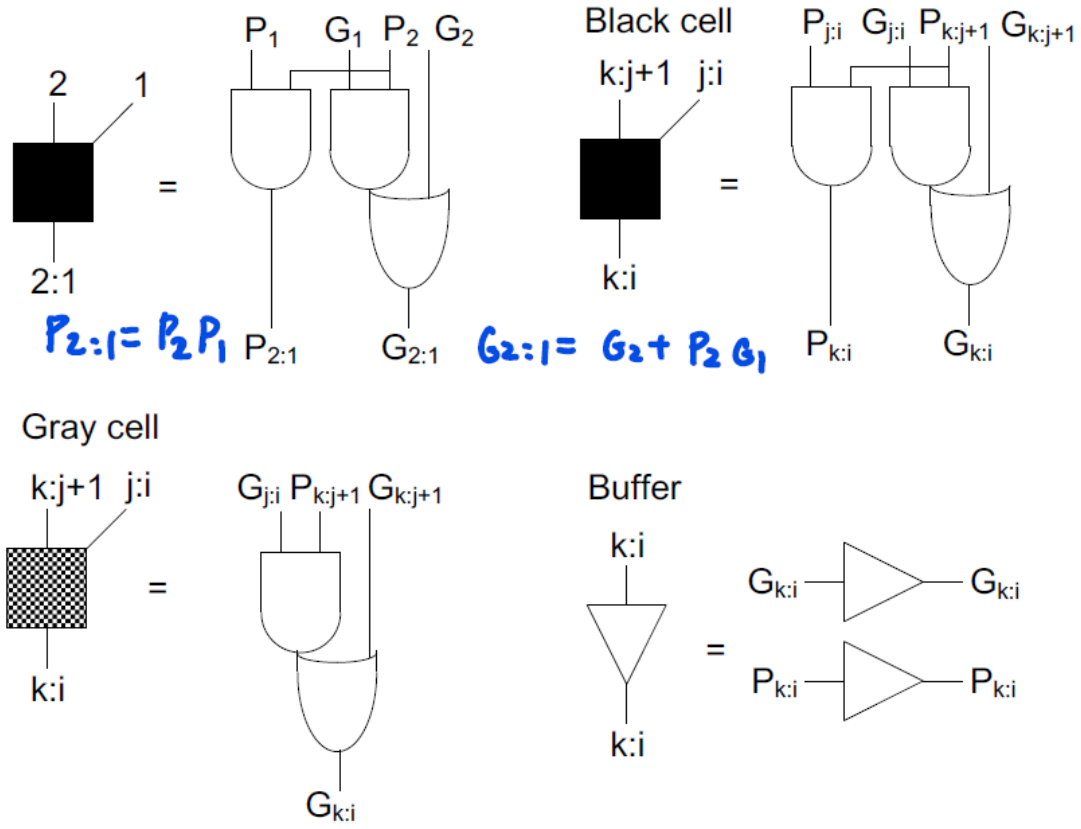
$G_{1:0} = G_1 + P_1 G_0$

$G_{2:0} = G_2 + P_2 G_{1:0}$

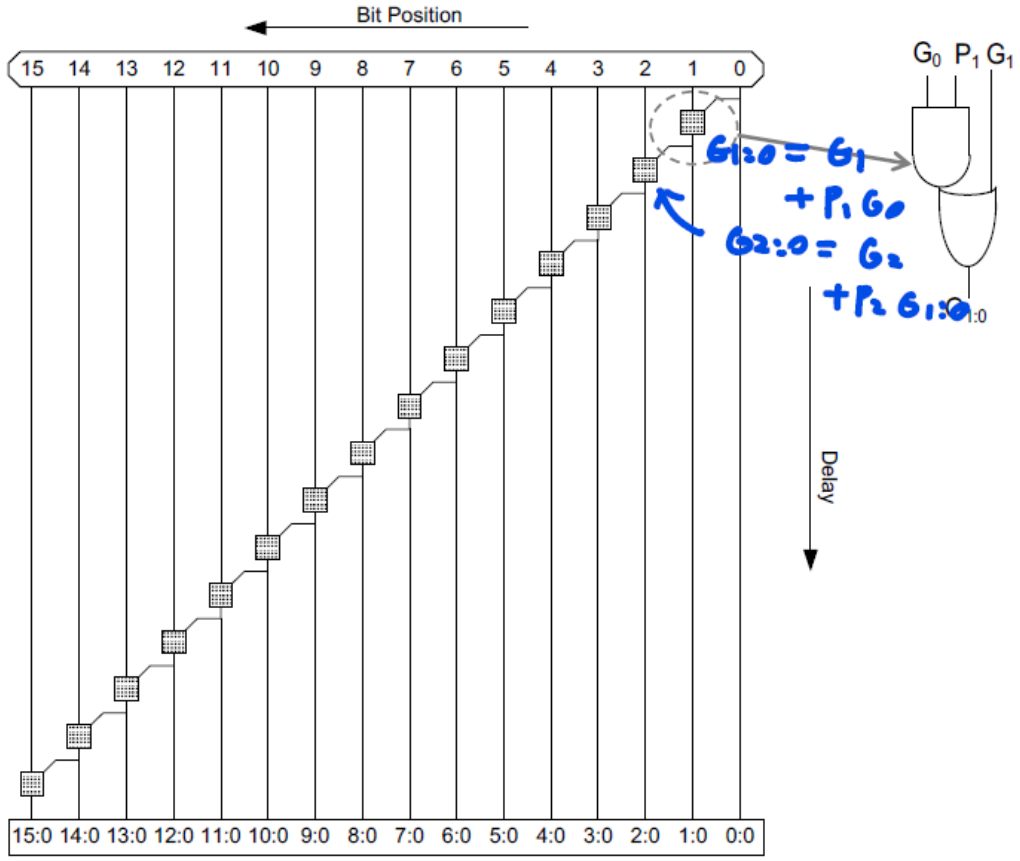
$t_{adder} = t_{setup} + (N-1)\, t_{carry} + \max(t_{carry}, t_{sum})$

# 加法器设计

- 基于PGK的加法器设计方法



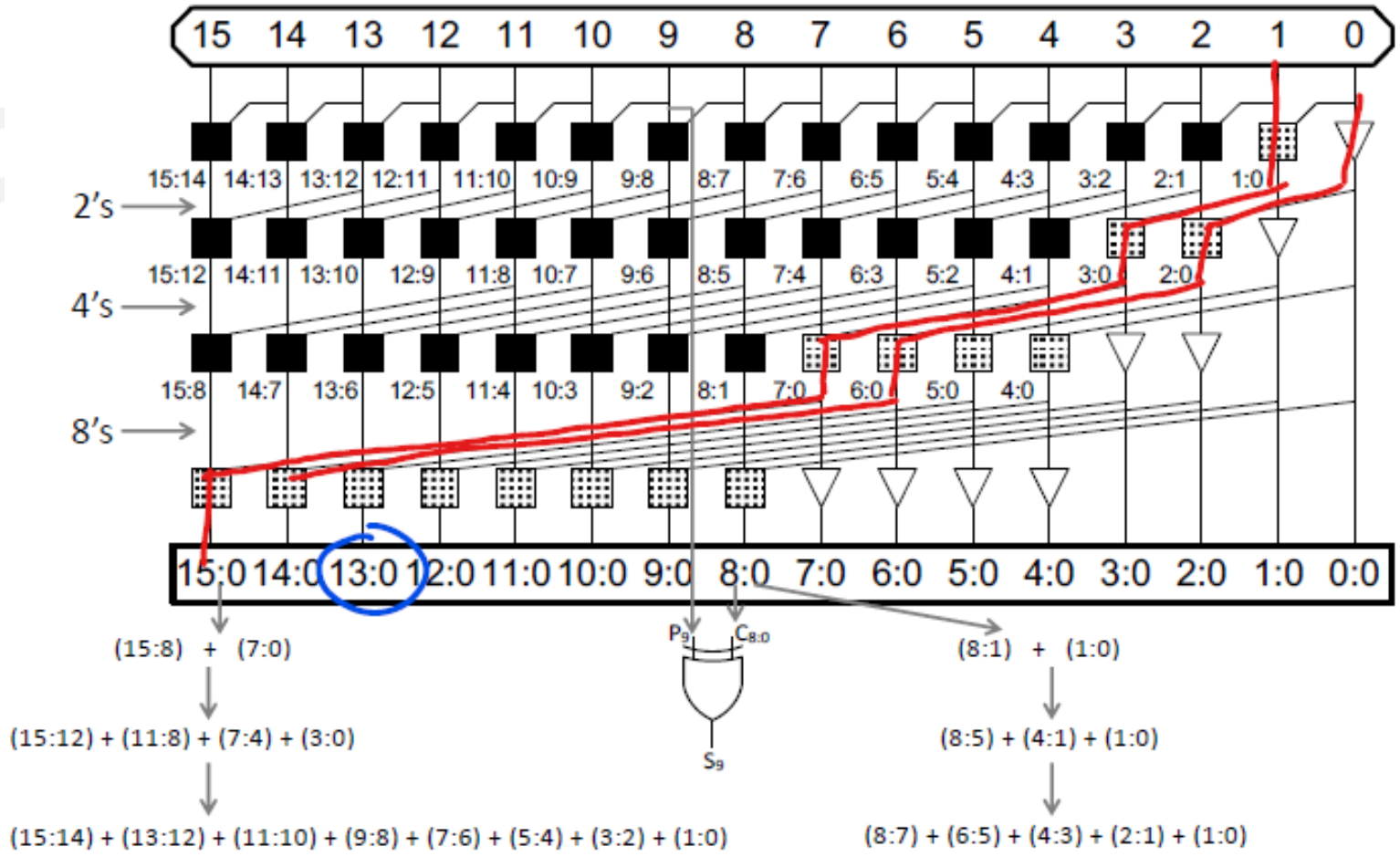

PG生成逻辑




Carry Ripple的PG图

# 加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器




$\log_2(N)$

$G_{15:0} = G_{13:6} + P_{13:6}\, G_{5:0}$

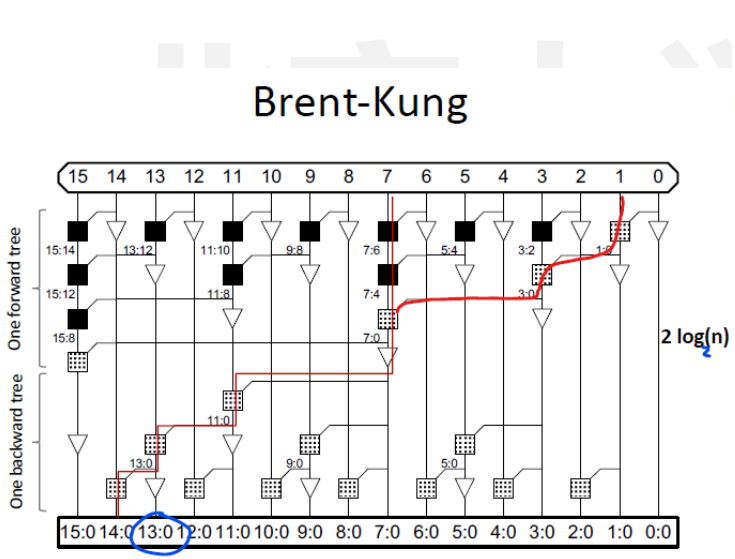
$G_{13:6} = G_{13:10} + P_{13:10}\, G_{9:6}$

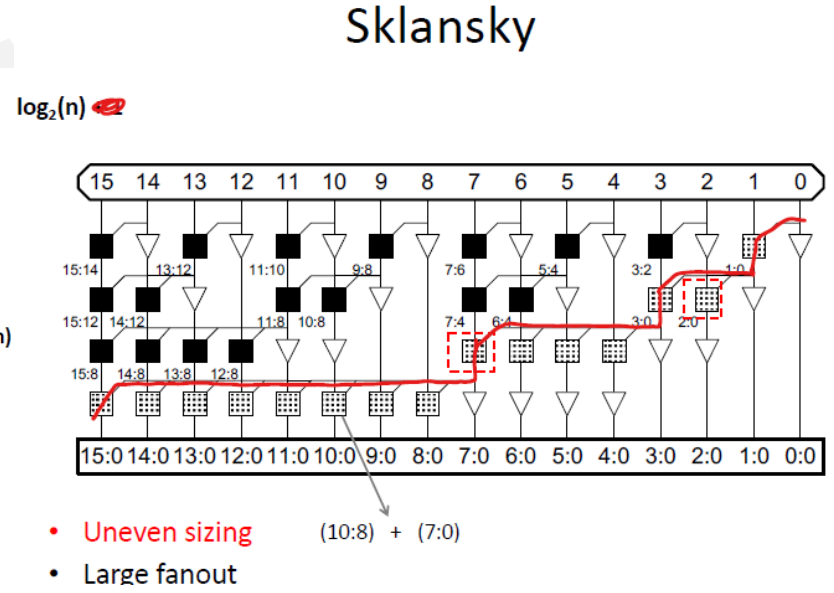
$P_{13:6} = P_{13:10}\, P_{9:6}$

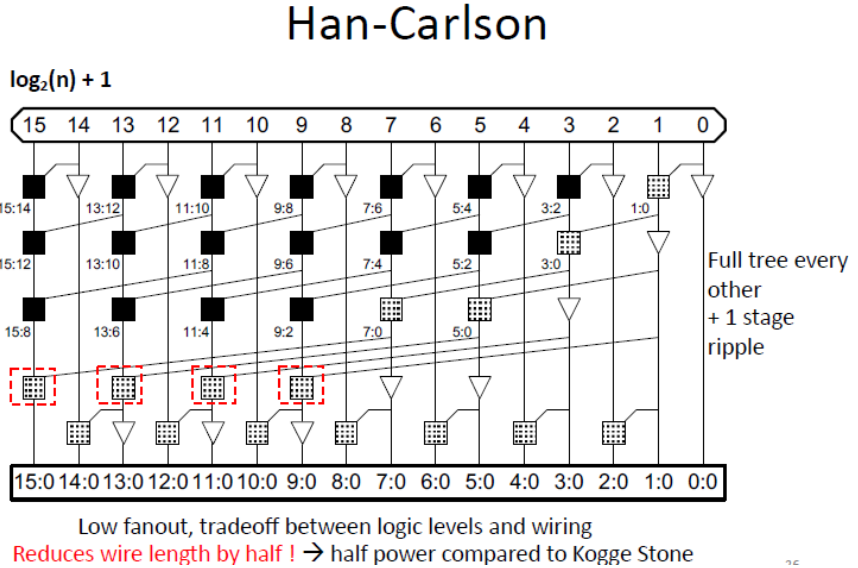
$G_{5:0} = G_{5:2} + P_{5:2}\, G_{1:0}$

# 加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器



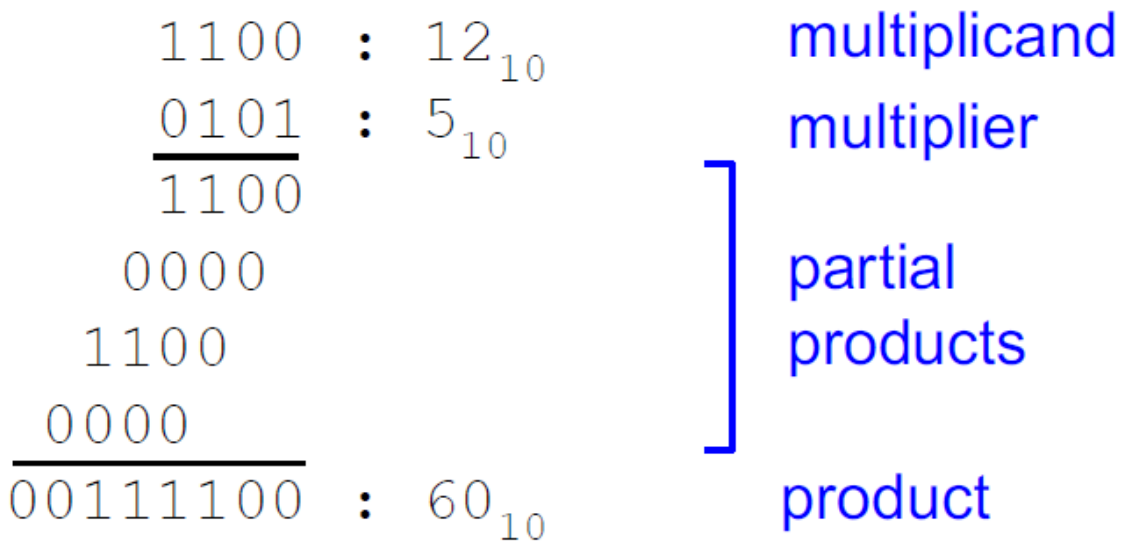




- Kogge-Stone: low logic levels, low fanout, high wiring
- Brent-Kung: low fanout, low wiring, high logic levels
- Sklansky: low logic levels, low wiring, high fanout

# 乘法器设计

- 乘法器设计的核心是部分和累加

Example:




M x N比特乘法

- 产生N个M比特部分乘积
- 求和得到M+N比特的结果

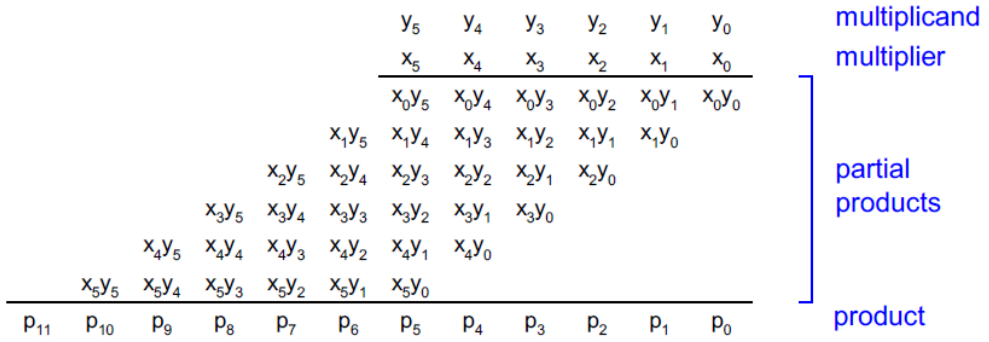
# 乘法器设计

- 乘法器设计的核心是部分和累加

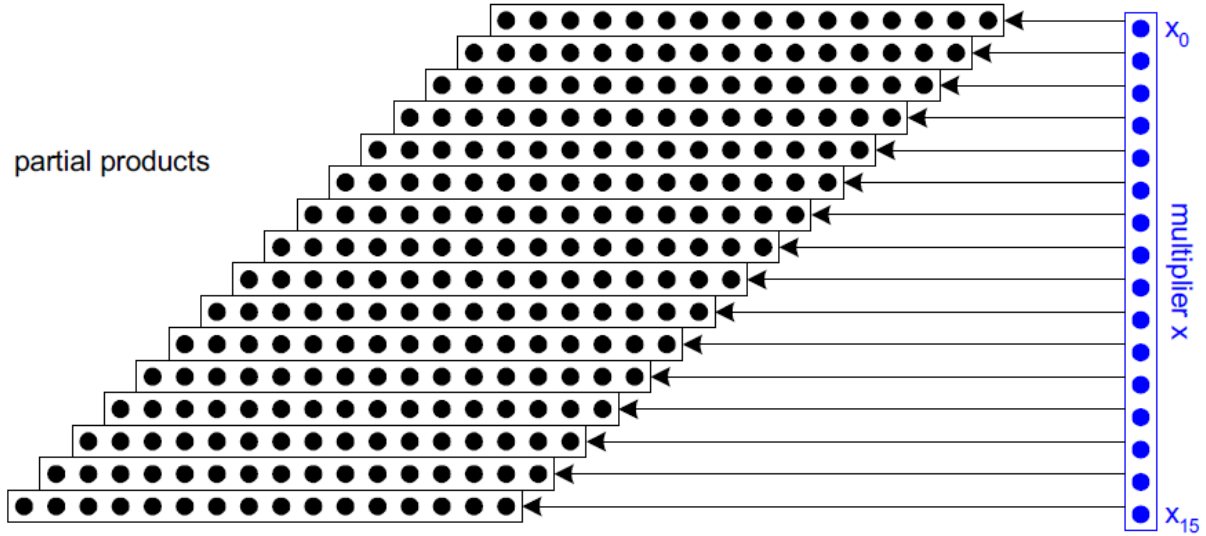
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$

Product:
$$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

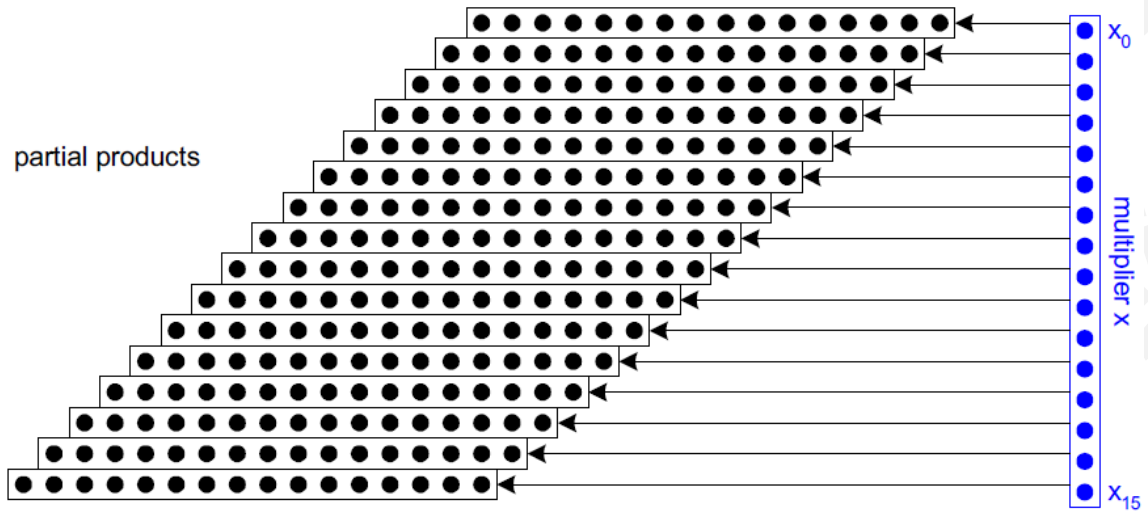


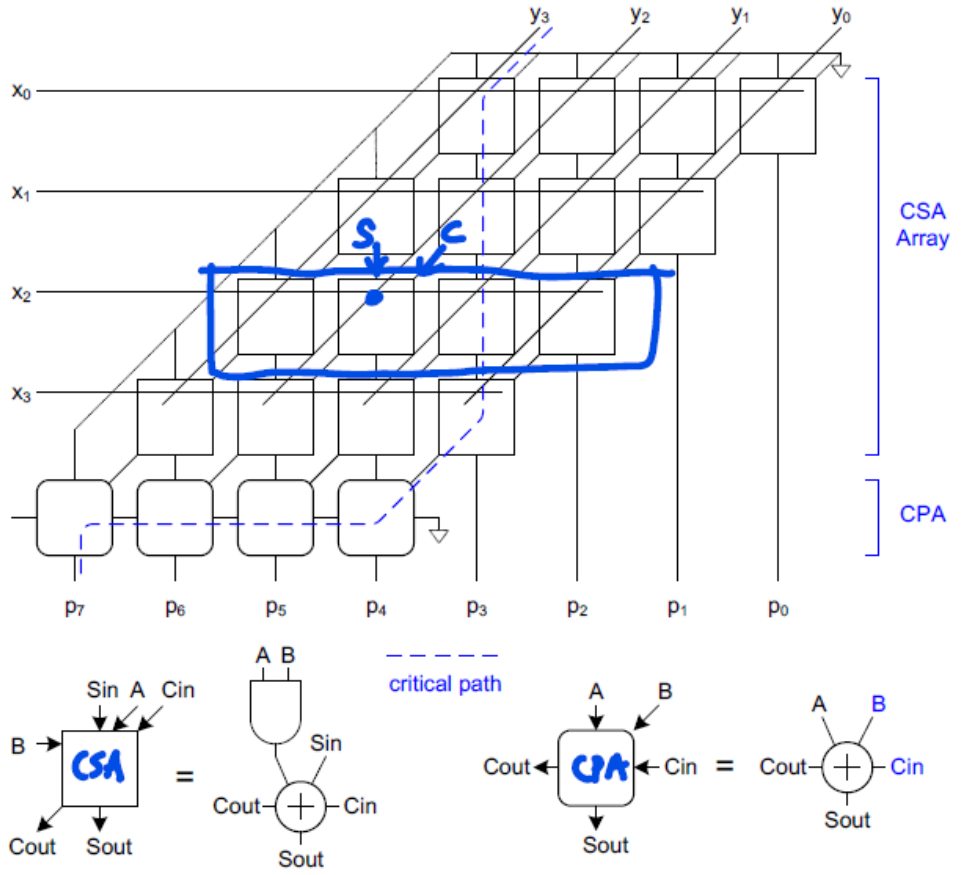

Each dot represents a bit



# 乘法器设计

- 乘法器设计的核心是部分和累加



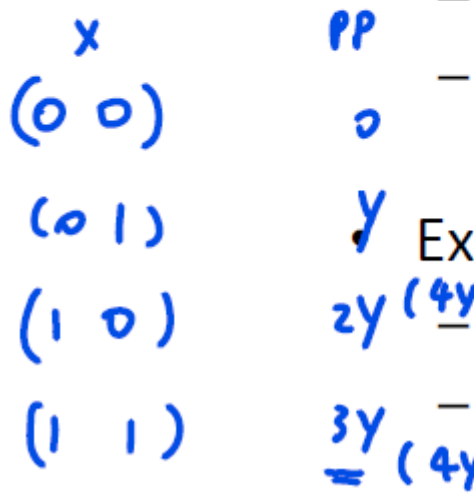



# 乘法器设计

- 如何减少部分和累加的次数?

- 阵列乘法器需要N个部分结果

- 如果我们将乘数以r bits为单位分组做乘法，我们将获得N/r个部分结果
  - Faster and smaller?
  - Called radix-$2^r$ encoding




- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder ☹

```
          1   1   0   0
        (0   1) (0   1)
        ---------------
          a   a   a   a
    b   b   b   b
    -------------
```

# 乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

  - PPi = 3Y时，可以用-Y表示并在下一级部分积中加4Y
    通过这种方式，部分积的计算中只用到了移位和补码计算
  - 相似的，PPi = 2Y时，可以用-2Y表示并在下一级部分积中加4Y

| Inputs | | | Partial Product | Booth Selects | | |
|---|---|---|---|---|---|---|
| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | $SINGLE_i$ | $DOUBLE_i$ | $NEG_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 0 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 1 | $2Y$ | 0 | 1 | 0 |
| 1 | 0 | 0 | $-2Y$ | 0 | 1 | 1 |
| 1 | 0 | 1 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 0 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 1 | $-0\ (=0)$ | 0 | 0 | 1 |

# 乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

| Inputs | | | Partial Product | Booth Selects | | |
|---|---|---|---|---|---|---|
| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | SINGLE$_i$ | DOUBLE$_i$ | NEG$_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 0 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 1 | $2Y$ | 0 | 1 | 0 |
| 1 | 0 | 0 | $-2Y$ | 0 | 1 | 1 |
| 1 | 0 | 1 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 0 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 1 | $-0\ (=0)$ | 0 | 0 | 1 |

假设计算 Y x Q = -6 x -7，Q 是乘数，Y 是被乘数（4bit）

1、Y = -6 =1010　Q = -7 = 1001　-Y = 6 = 0110

2、乘数 Q 后补零，Q = 10010

3、被乘数双符号位，Y = 11010，-Y = 00110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 10010

A = 11111010　Q = 1001　Q-1 = 0　补码符号扩展

Step 2：Q = 10010

A = 00110000　Q =1001　Q-1 = 0　左移补零

结果：11111010 （-6） + 00110000 （48） = 42

# 乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

| Inputs | | | Partial Product | Booth Selects | | |
|---|---|---|---|---|---|---|
| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | $SINGLE_i$ | $DOUBLE_i$ | $NEG_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 0 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 1 | $2Y$ | 0 | 1 | 0 |
| 1 | 0 | 0 | $-2Y$ | 0 | 1 | 1 |
| 1 | 0 | 1 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 0 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 1 | $-0\ (=0)$ | 0 | 0 | 1 |

假设计算 Y x Q = -6 x 7，Q 是乘数，Y 是被乘数（6bit）

1、Y = -6 =111010　Q = 7 = 000111　-Y = 6 = 000110

2、乘数 Q 后补零，Q = 0001110

3、被乘数双符号位，Y = 1111010，-Y = 0000110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 0001110

A = 000000000110　Q = 000111　Q-1 = 0　补码符号扩展

Step 2：Q = 0001110

A = 111111010000　Q =000111　Q-1 = 0　左移/符号位扩展

Step3：Q = 0001110

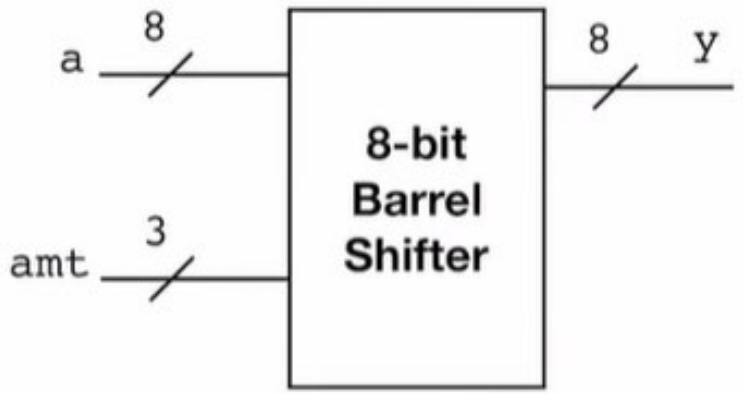
结果 = 000000000110（6）+ 111111010000（-48）= -42

# 位移器设计

- Shifter也是重要的数字电路模块之一

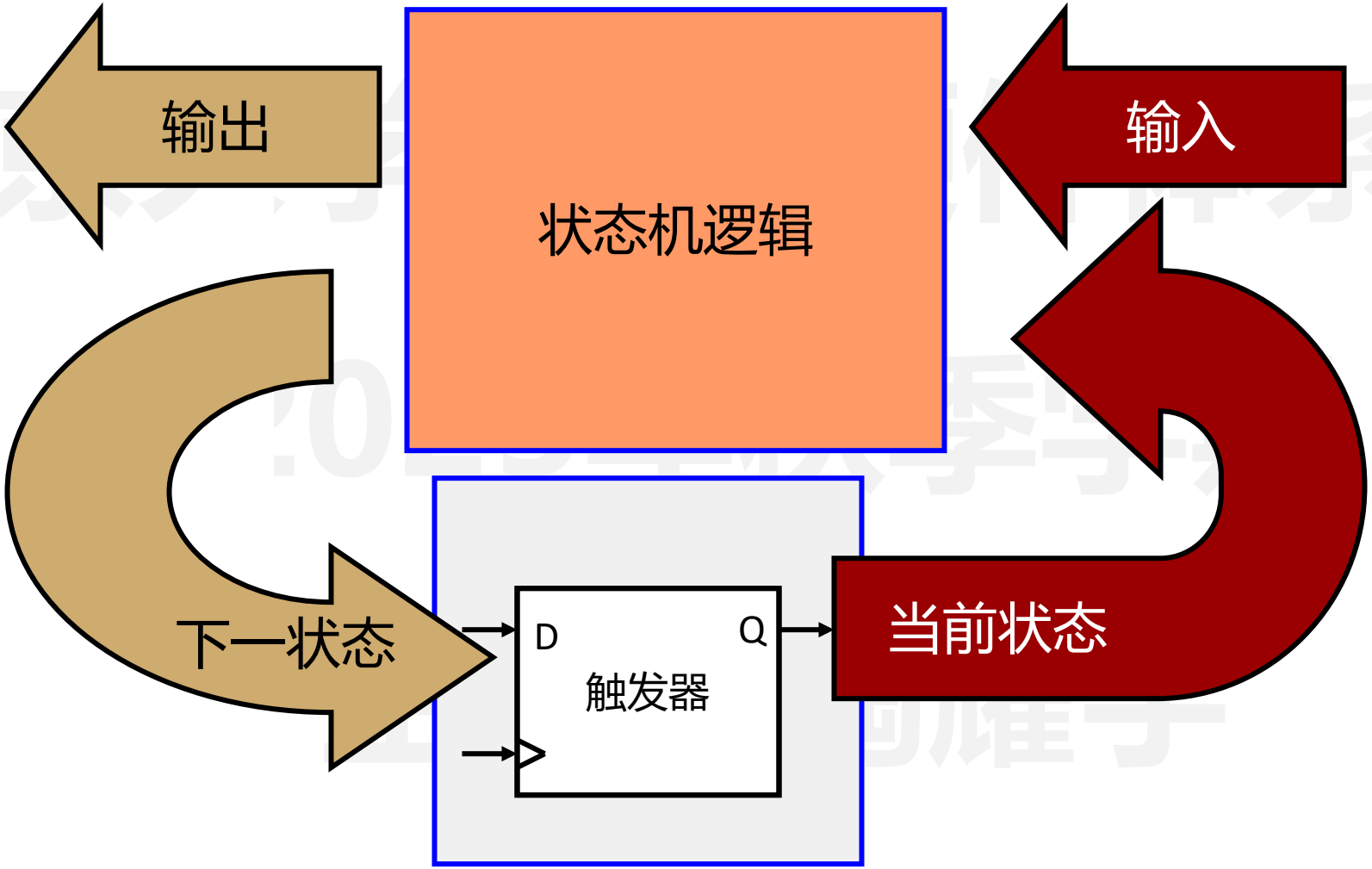
```
module barrel_shifter
    (
        input logic [7:0] a,
        input logic [2:0] amt,
        output logic [7:0] y
    );

    always_comb
        case(amt)
            3'b000: y = a;
            3'b001: y = {a[0], a[7:1]};
            3'b010: y = {a[1:0], a[7:2]};
            3'b011: y = {a[2:0], a[7:3]};
            3'b100: y = {a[3:0], a[7:4]};
            3'b101: y = {a[4:0], a[7:5]};
            3'b110: y = {a[5:0], a[7:6]};
            3'b111: y = {a[6:0], a[7]};
            default: y = a;
        endcase
endmodule
```



# 为什么需要状态机

- 控制电路的基石



# 为什么需要状态机

- 控制电路的基石

## 状态机实例1 – 控制一个红绿灯

- 仅考虑红灯和绿灯，灯转换的速度不快于每次30s（0.033 Hz 时钟）
- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

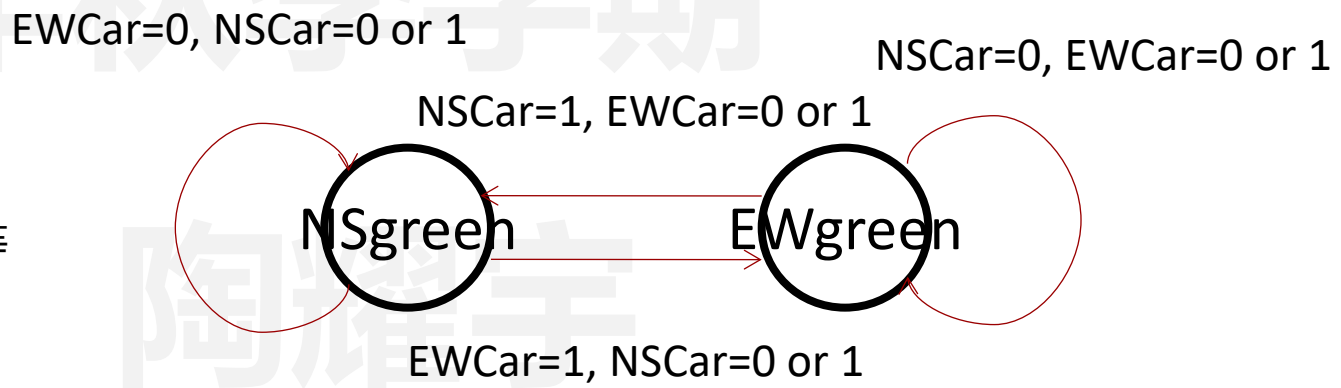
# 为什么需要状态机

- 控制电路的基石

## 状态机实例1 – 控制一个红绿灯

- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

- 需要2个状态
  - Nsgreen、EWgreen



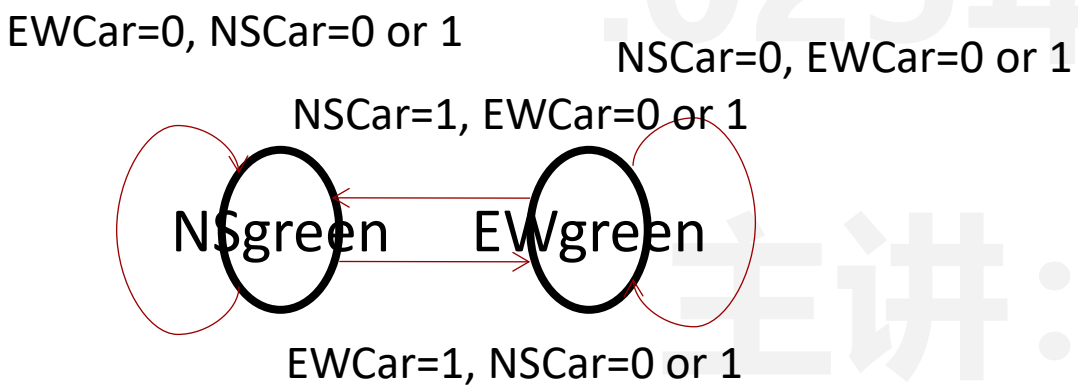
# 为什么需要状态机

- 控制电路的基石

## 状态机实例1 – 控制一个红绿灯

- 需要2个状态
  - Nsgreen、EWgreen

| Current state | Inputs | | Next state |
|---|---|---|---|
| | NScar | EWcar | |
| NSgreen | 0 | 0 | NSgreen |
| NSgreen | 0 | 1 | EWgreen |
| NSgreen | 1 | 0 | NSgreen |
| NSgreen | 1 | 1 | EWgreen |
| EWgreen | 0 | 0 | EWgreen |
| EWgreen | 0 | 1 | EWgreen |
| EWgreen | 1 | 0 | NSgreen |
| EWgreen | 1 | 1 | NSgreen |




| Current state | Outputs | |
|---|---|---|
| | NSlite | EWlite |
| NSgreen | 1 | 0 |
| EWgreen | 0 | 1 |

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

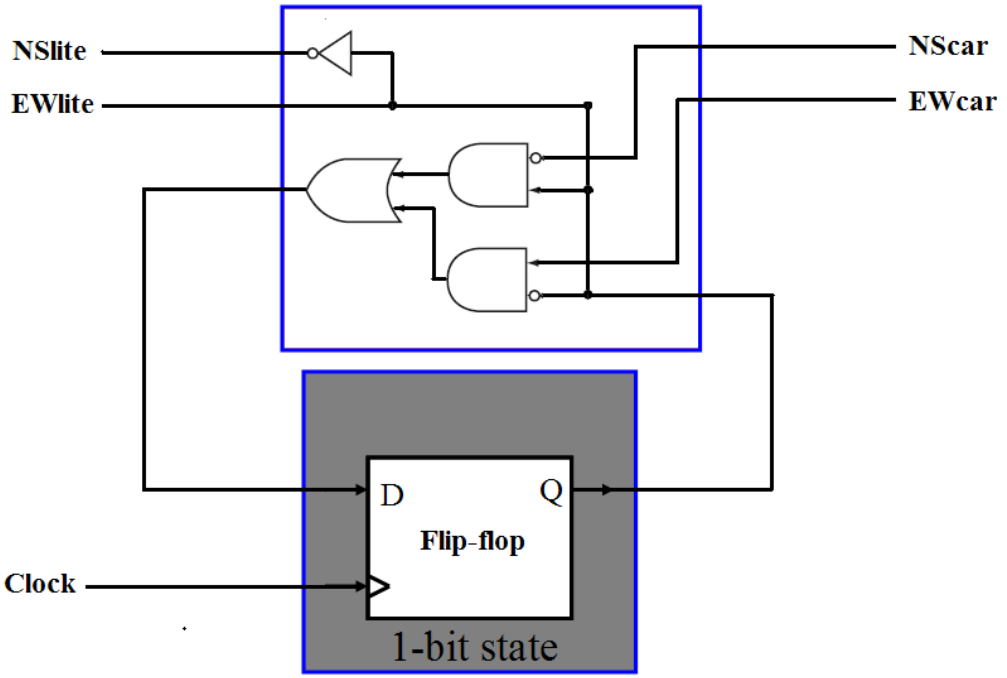
# 为什么需要状态机

- 控制电路的基石

## 状态机实例1 – 控制一个红绿灯

| Current state | Inputs | | Next state |
|---|---|---|---|
| | NScar | EWcar | |
| NSgreen | 0 | 0 | NSgreen |
| NSgreen | 0 | 1 | EWgreen |
| NSgreen | 1 | 0 | NSgreen |
| NSgreen | 1 | 1 | EWgreen |
| EWgreen | 0 | 0 | EWgreen |
| EWgreen | 0 | 1 | EWgreen |
| EWgreen | 1 | 0 | NSgreen |
| EWgreen | 1 | 1 | NSgreen |

| Current state | Outputs | |
|---|---|---|
| | NSlite | EWlite |
| NSgreen | 1 | 0 |
| EWgreen | 0 | 1 |




$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

# 为什么需要状态机

- 控制电路的基石

Step 1 – 定义状态并画出状态转换图

Step 2 – 给每一个状态赋值并更新状态转换图

Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

Step 4 – 画出实际电路图

状态将在每一个时钟上升沿更新