北京大学
PEKING UNIVERSITY

人工智能的硬件基石

从物理器件到计算架构

第三讲：复杂逻辑与计算单元设计

主讲：陶耀宇

2025年春季

注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况

- **第1次作业将在周一3月3号（今晚）上线**

  2周时间完成，第1次作业提交截止日期：**3月17号晚11:59**

  6次作业可以使用总计6个Late day

  Late Day耗尽后，**每晚交1天扣除20%当次作业分数**

- **第1次lab时间：3月10-4月10**

- **第2次lab时间：4月10-6月10**

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

- 请各位选课同学能够初步熟练使用CLAB平台！
- Clab网址：clab.pku.edu.cn
- **Clab问题请联系助教詹喆同学**

目 录

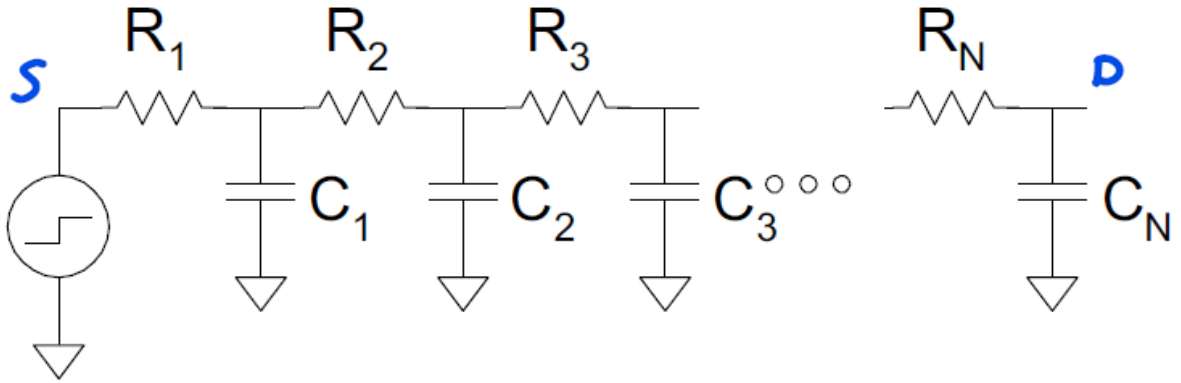
CONTENTS

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）

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

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

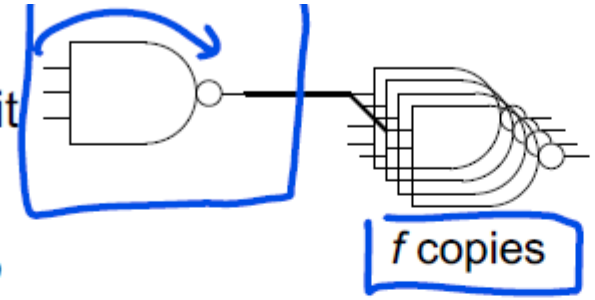


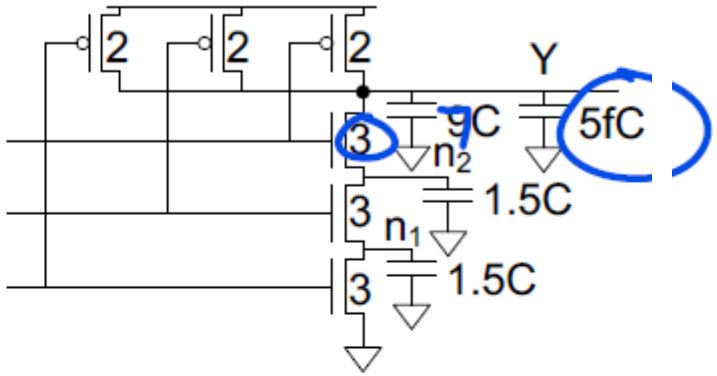
逻辑电路的Elmore Delay模型

- 拓展多级的RC模型 – 3-input NAND gates

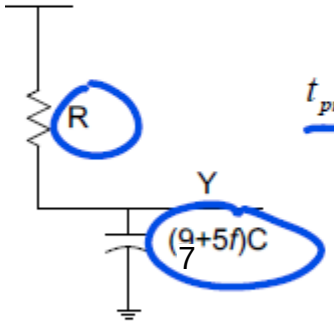
估算驱动f个相同栅极的3输入与非门的上升/下降延迟的最差情况

- R=13KΩ/unit
- C=0.21fF/unit

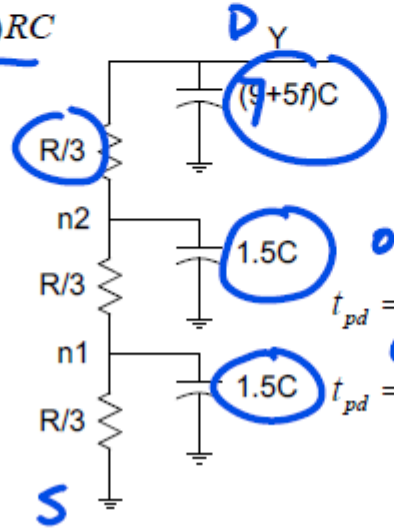




Worst-case pull up



$t_{pu} = (9+5f)RC$



Worst-case pull down

$t_{pd} = 1.5C\frac{R}{3} + 1.5C\left(\frac{R}{3}+\frac{R}{3}\right) + [(9+5f)C]\left(\frac{R}{3}+\frac{R}{3}+\frac{R}{3}\right)$

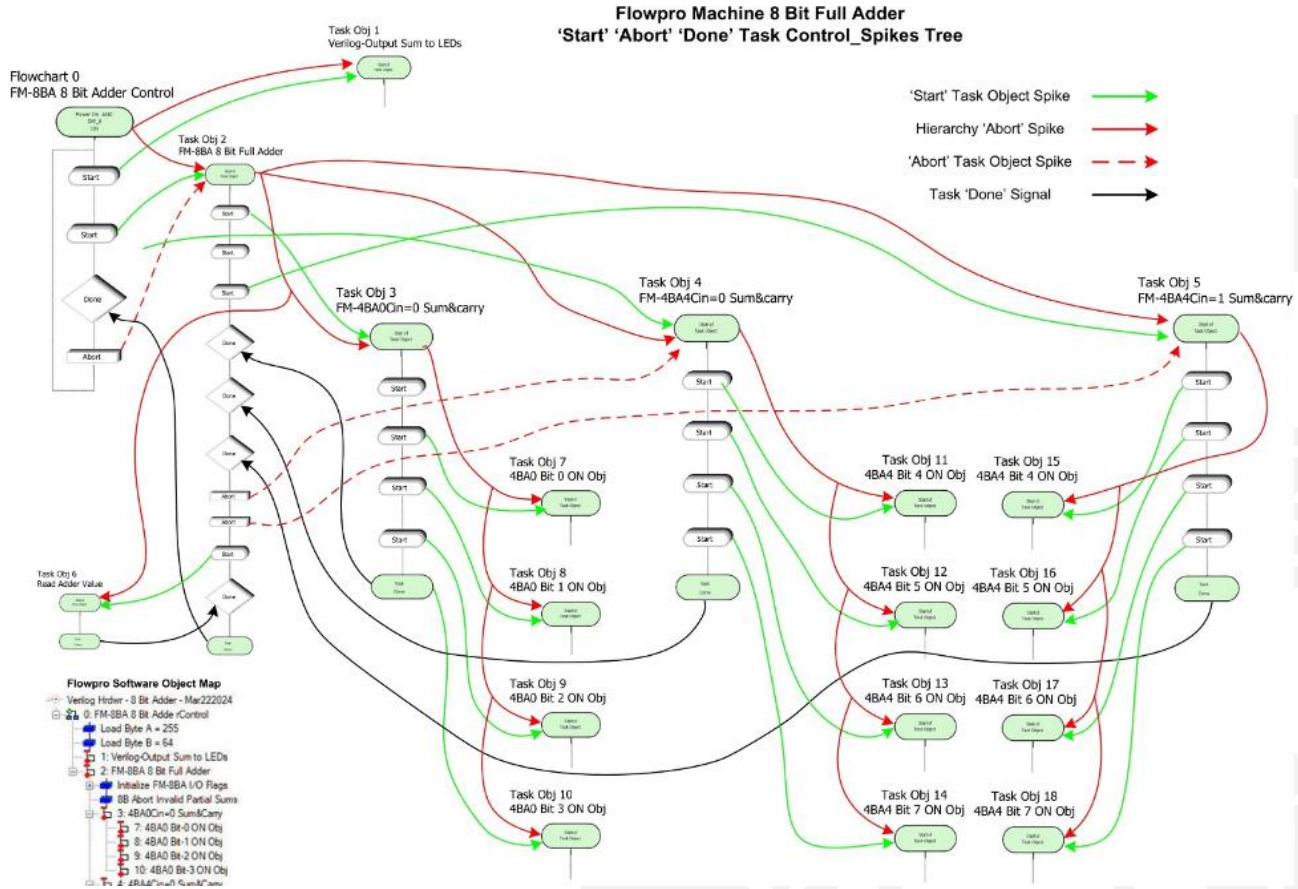
$t_{pd} = (10.5+5f)RC$

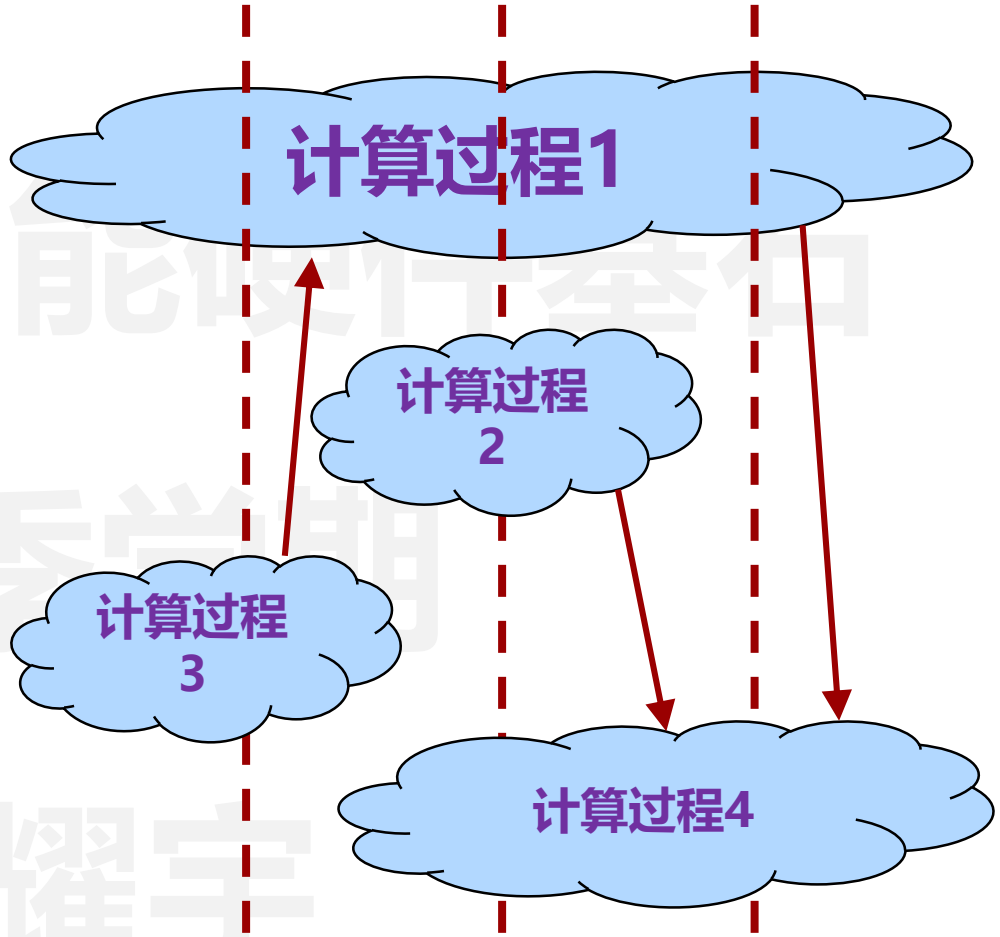
目 录

CONTENTS

电路时序的基本概念

- 电路为什么需要一个时钟?



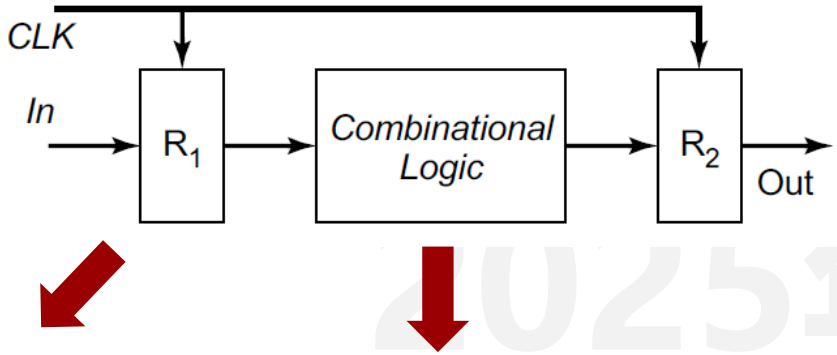


无时钟：非常难以控制每一个信号的有效时间

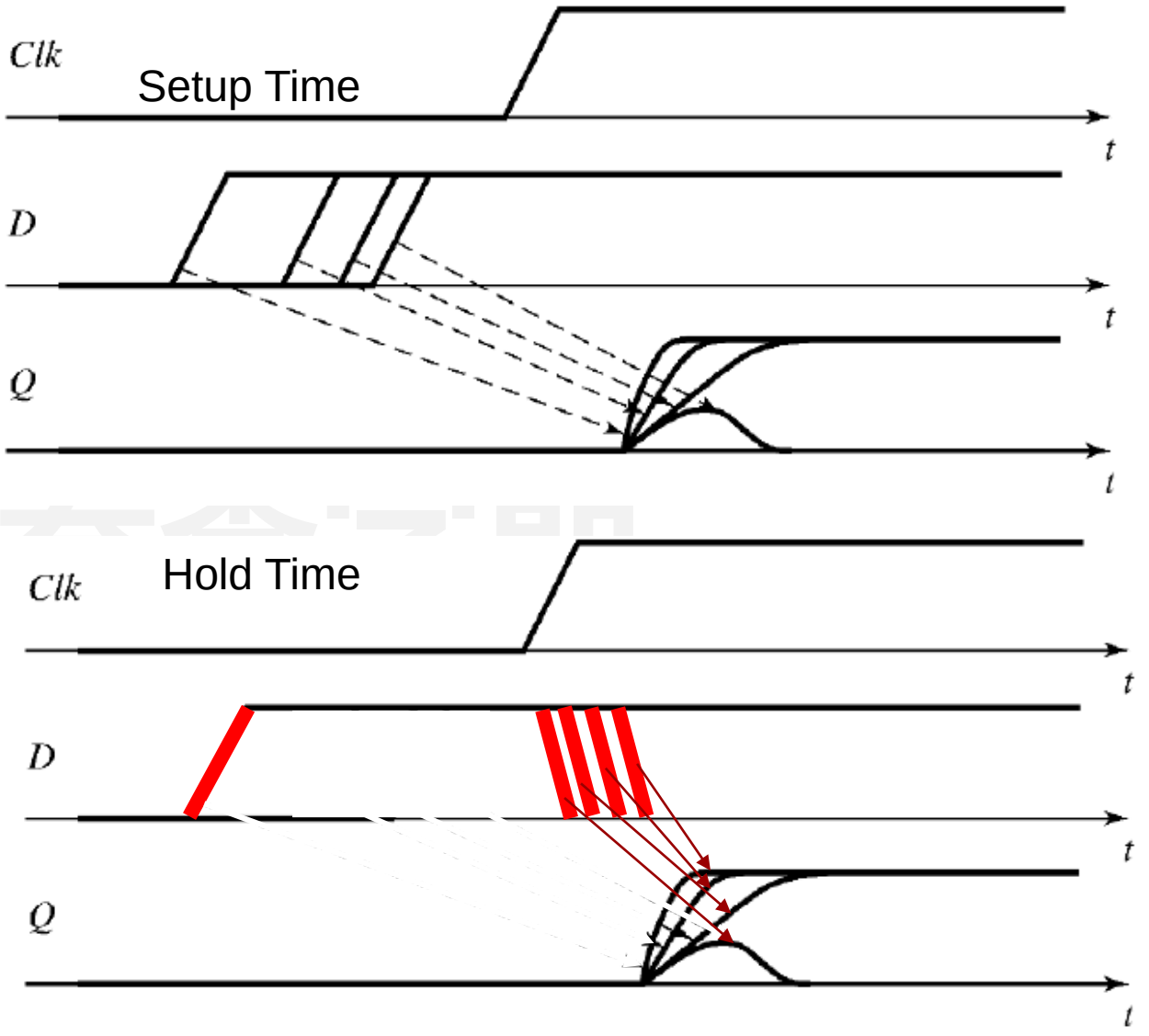
引入时钟：每隔一段计算将结果同步一次

电路时序的基本概念

- 同步时序（Synchronous Timing）

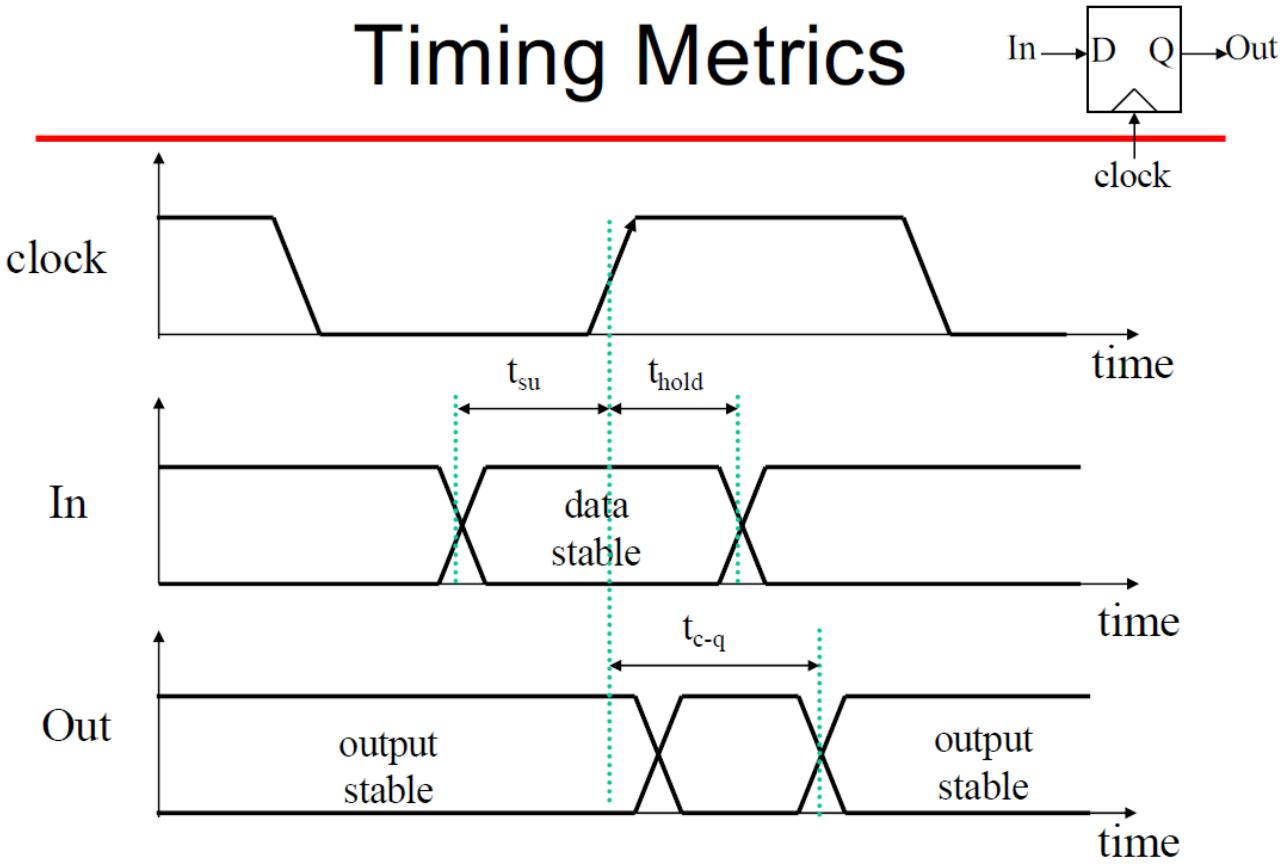


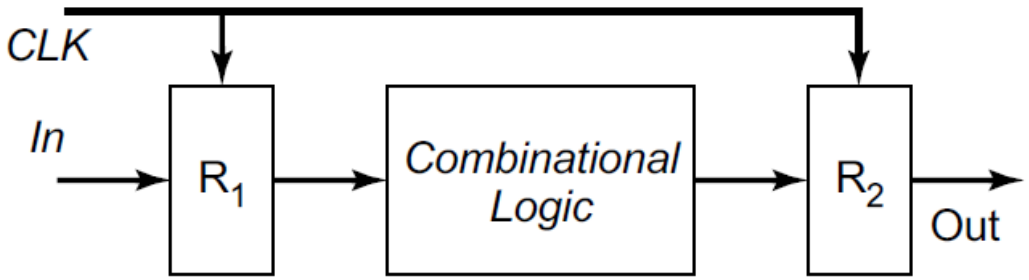
触发器　组合逻辑（各种逻辑门电路）
(Register)



电路时序的基本概念

- 同步时序（Synchronous Timing）





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

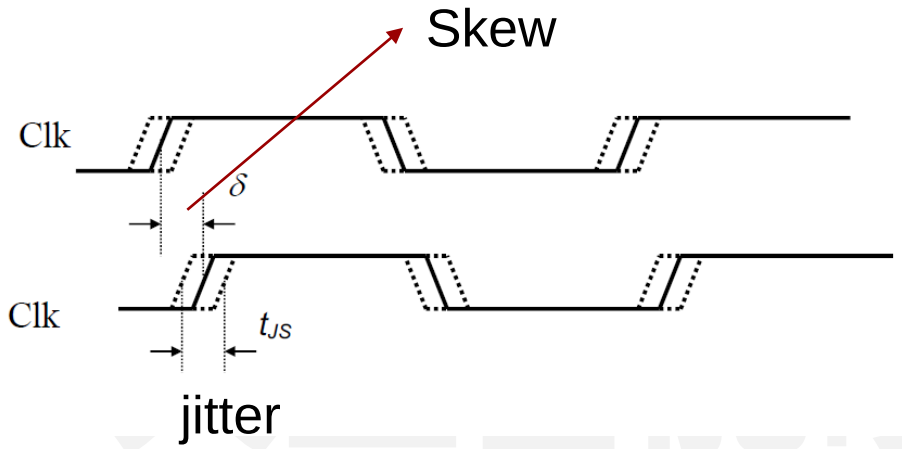
min

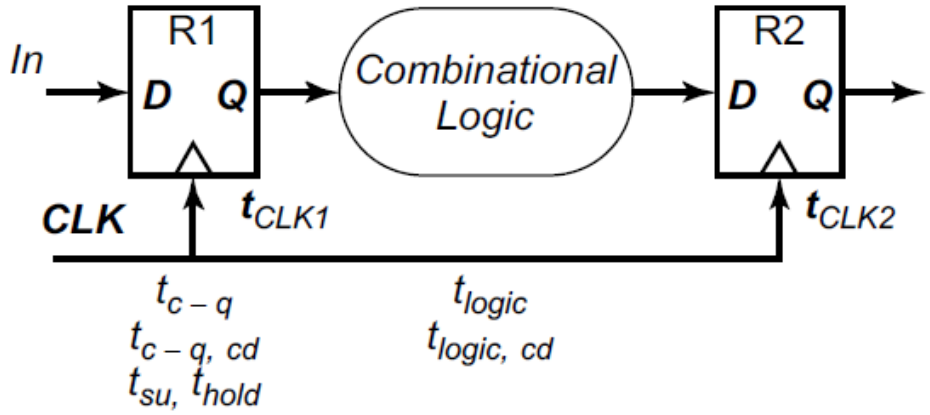
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

max

电路时序的基本概念

- 时钟的不稳定性

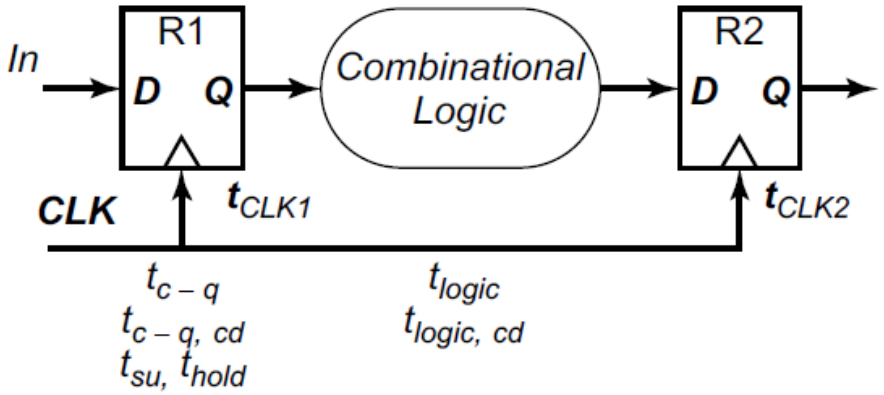




Minimum cycle time:

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

最坏情况为接收边沿过早到达（negative δ）



Hold time constraint:

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

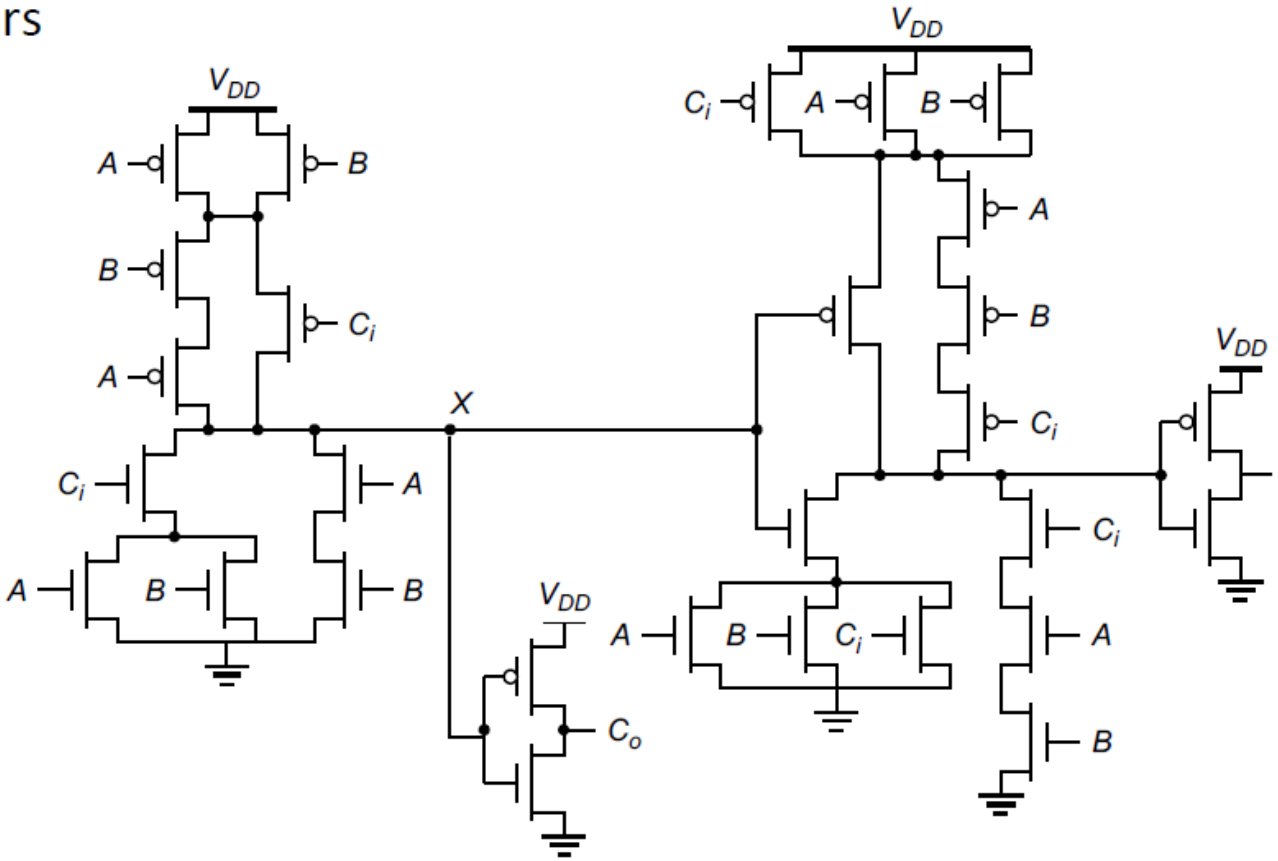
最坏情况为接收边沿过晚到达（正偏差）
数据和时钟之间的竞争
Cd：contamination delay（最快可能延迟）

目 录

CONTENTS

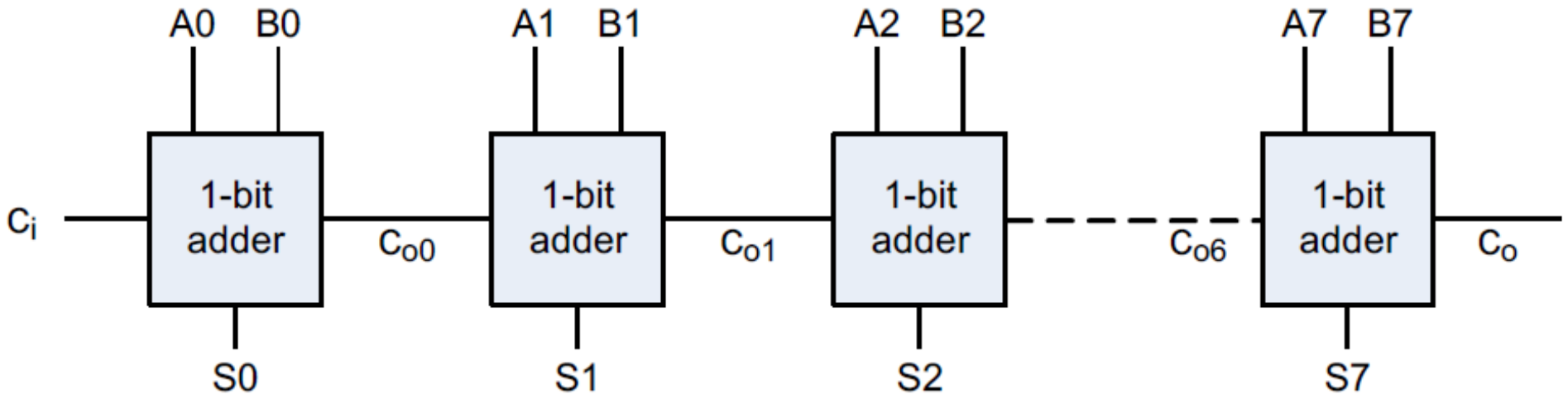
加法器设计

- 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

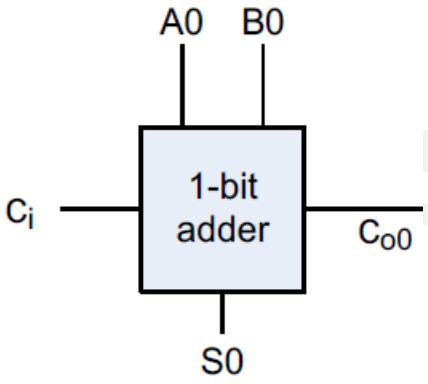
$$t_d = O(N)$$

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

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

- 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

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

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

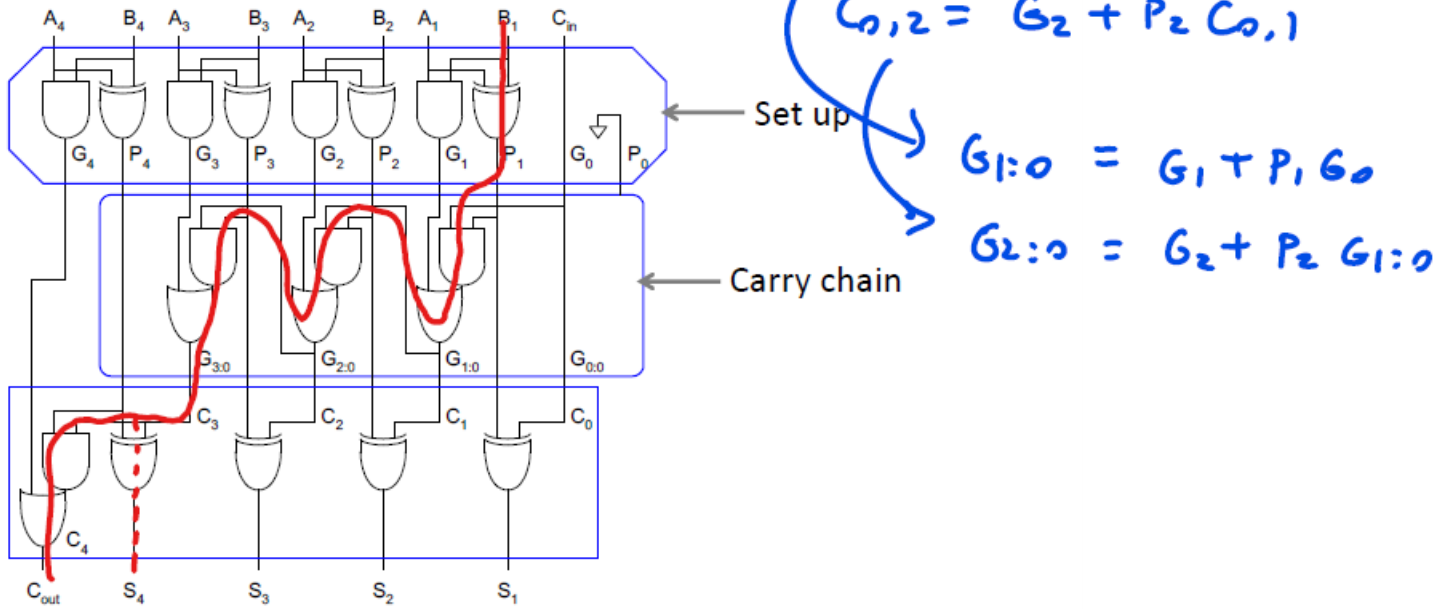
$P = A \oplus B$

$P = A + B$

- 基于PGK的加法器设计方法

Carry-Ripple using P and G

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



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

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

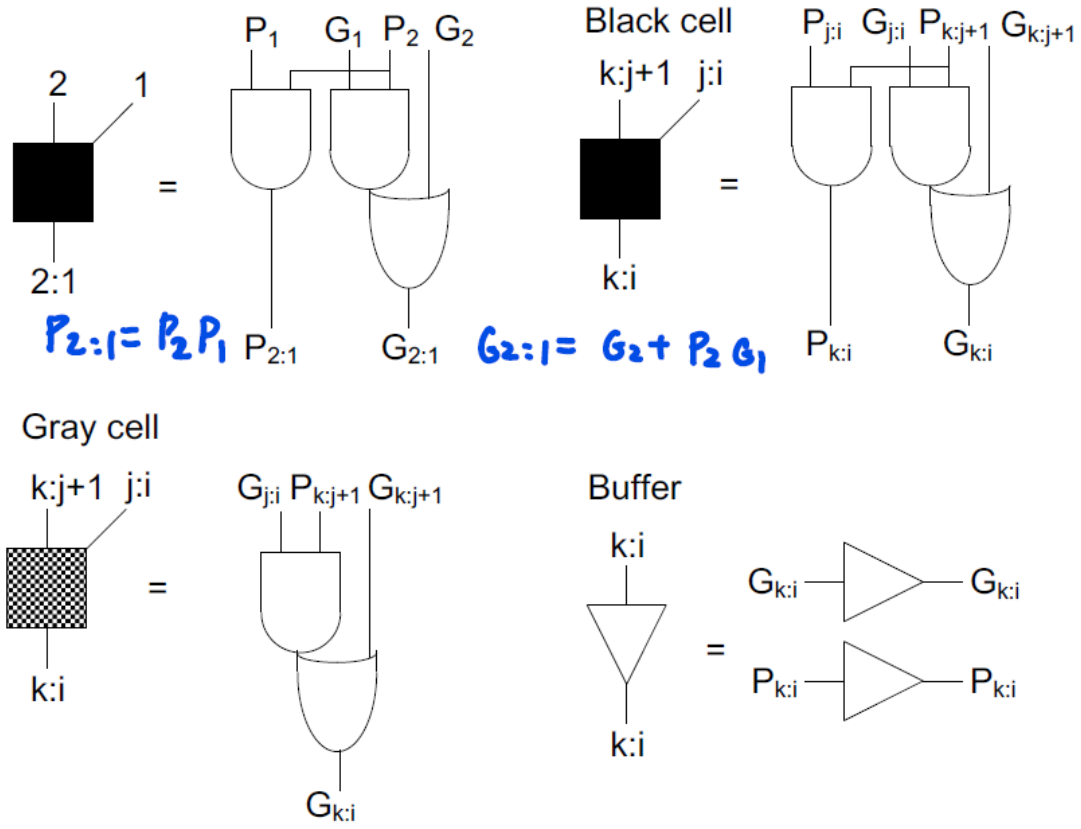
$G_{1:0} = G_1 + P_1 G_0$

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

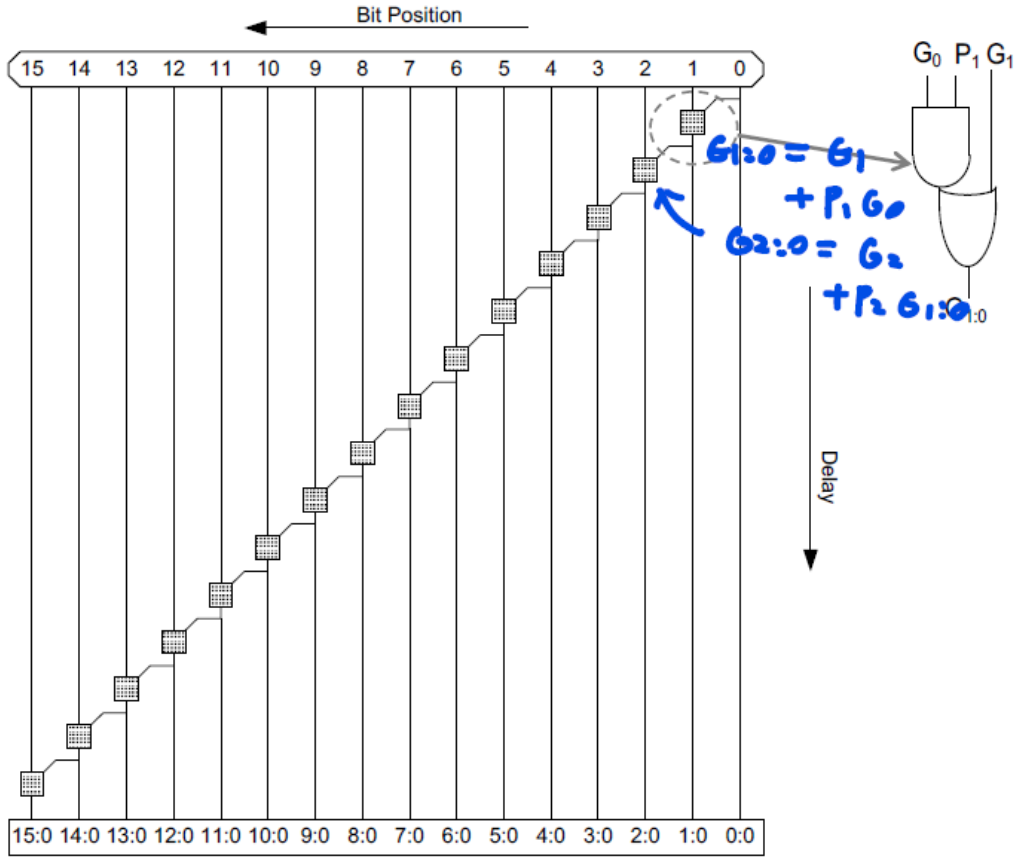
$t_{adder} = t_{setup} + (N-1)\, t_{carry} + \max(t_{carry}, t_{sum})$

加法器设计

- 基于PGK的加法器设计方法



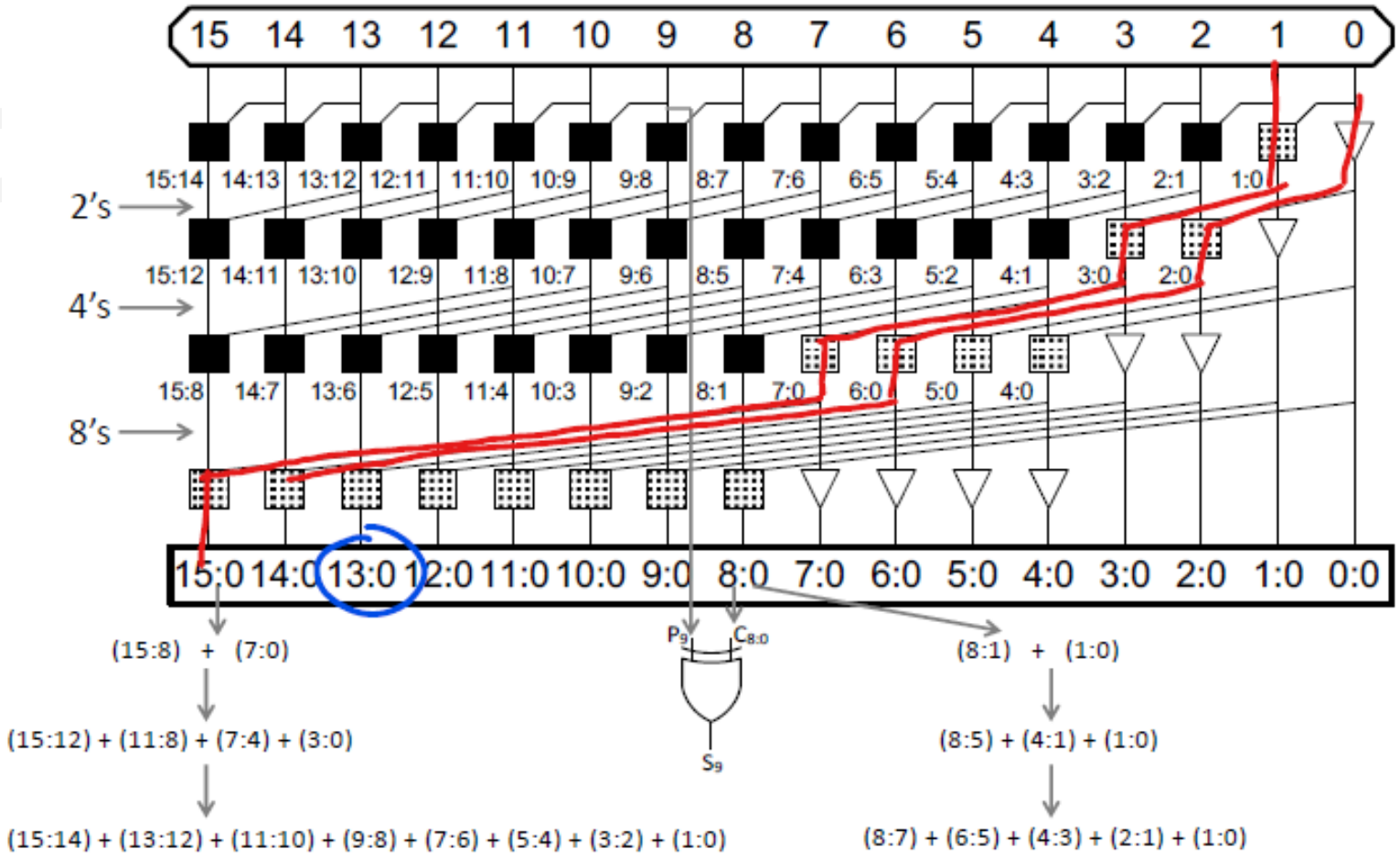
PG生成逻辑



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



$\log_2(N)$

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

$G_{13:6} = G_{13:10} + P_{13:0}\, G_{9:6}$

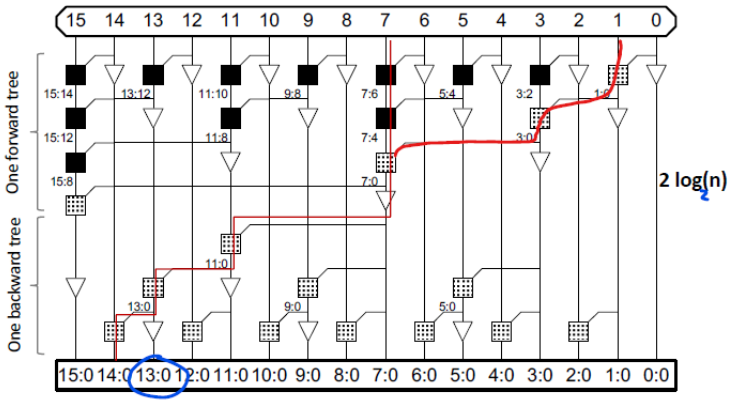
$P_{13:6} = P_{13:10}\, P_{9:6}$

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

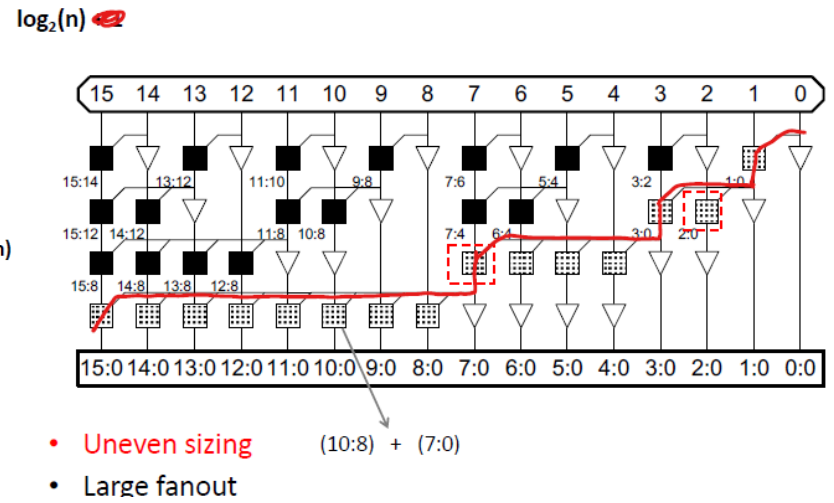
加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器

Brent-Kung

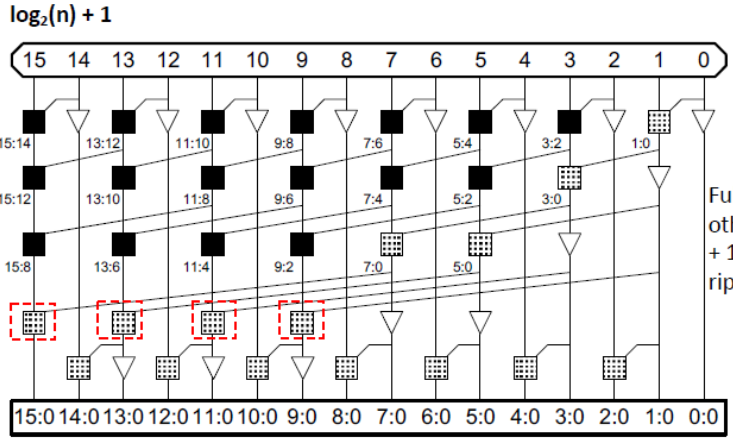


Sklansky



- Uneven sizing (10:8) + (7:0)
- Large fanout

Han-Carlson



Low fanout, tradeoff between logic levels and wiring
Reduces wire length by half ! → half power compared to Kogge Stone

- 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:

1100	:	12_{10}	multiplicand
0101	:	5_{10}	multiplier
1100			
0000			partial products
1100			
0000			
00111100	:	60_{10}	product

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

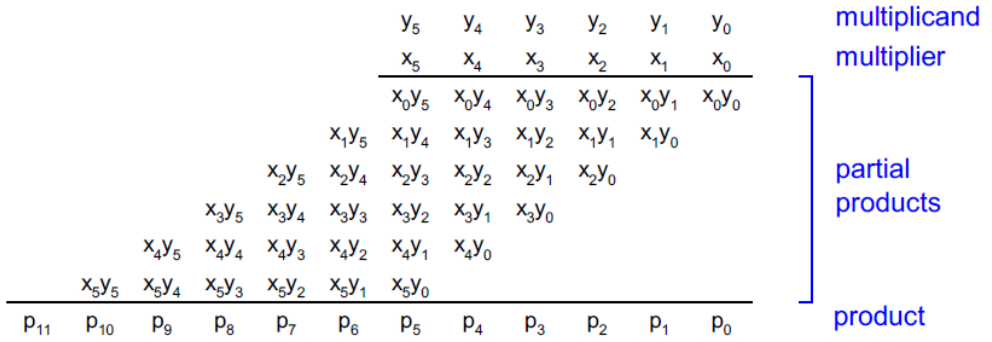
乘法器设计

- 乘法器设计的核心是部分和累加

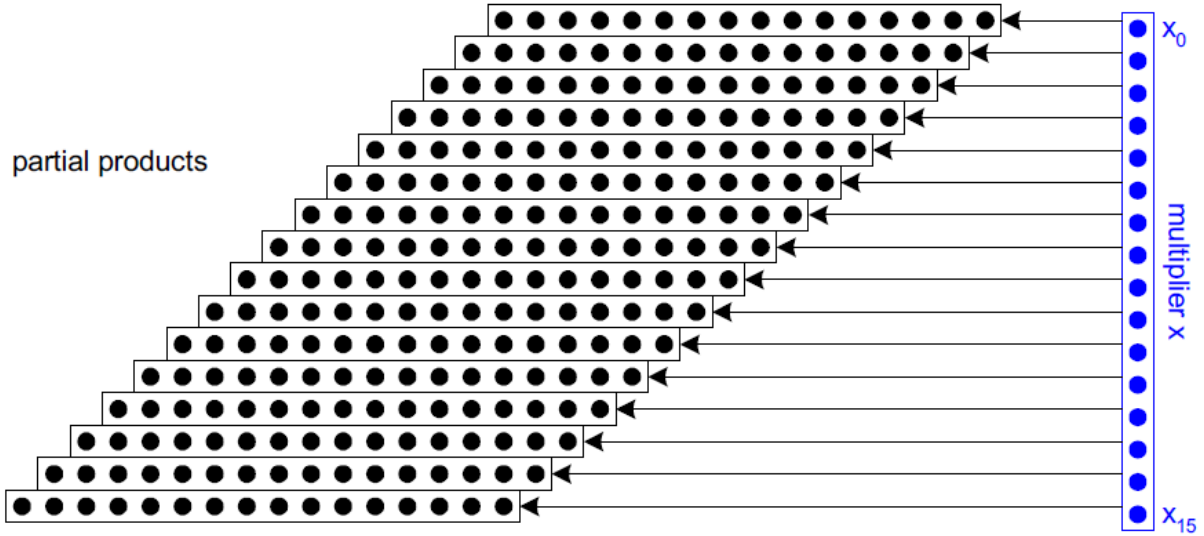
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

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

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

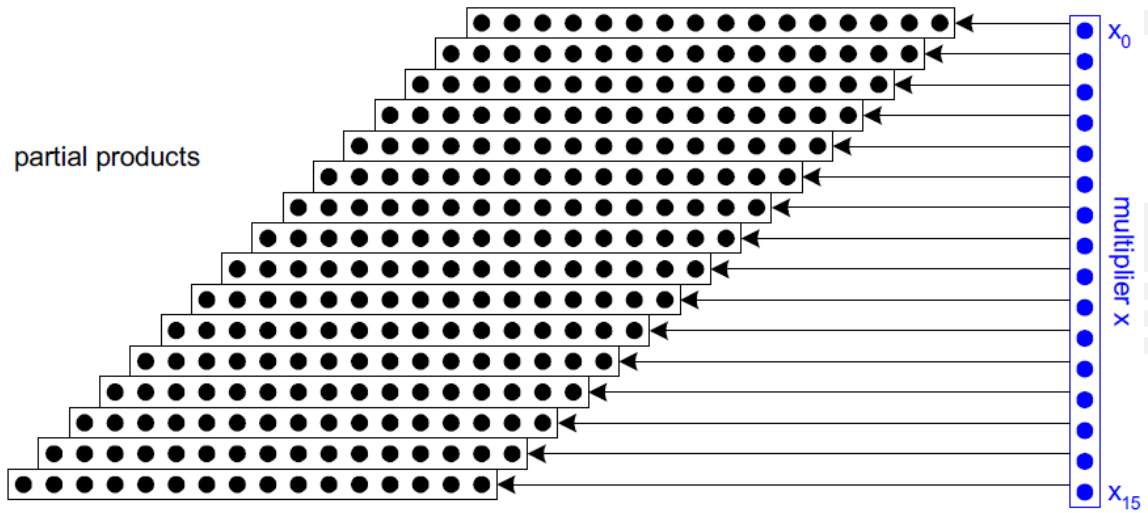


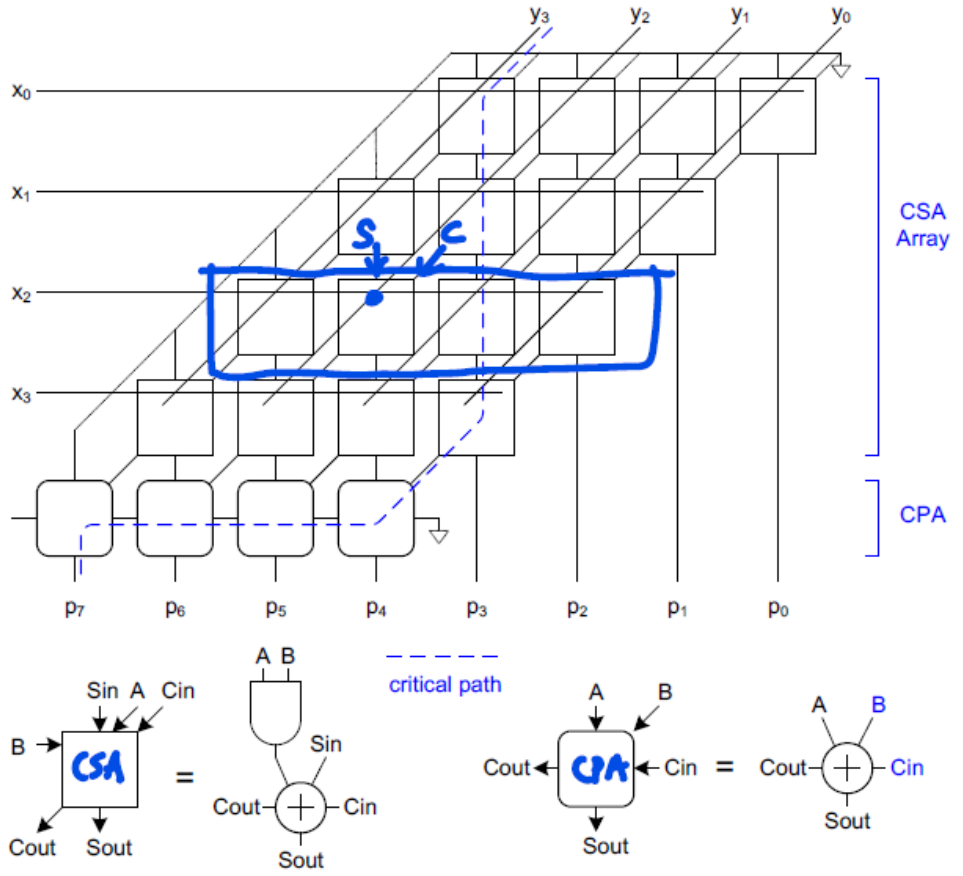
Each dot represents a bit



乘法器设计

- 乘法器设计的核心是部分和累加

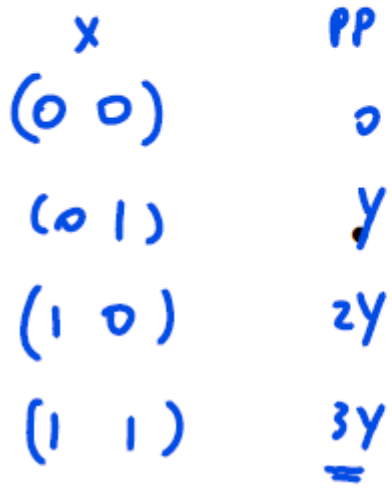


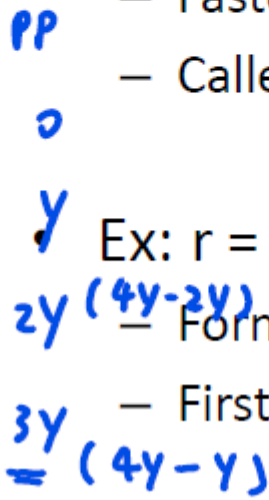


乘法器设计

- 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
    -------------------
```

北京大学 PEKING UNIVERSITY

- 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

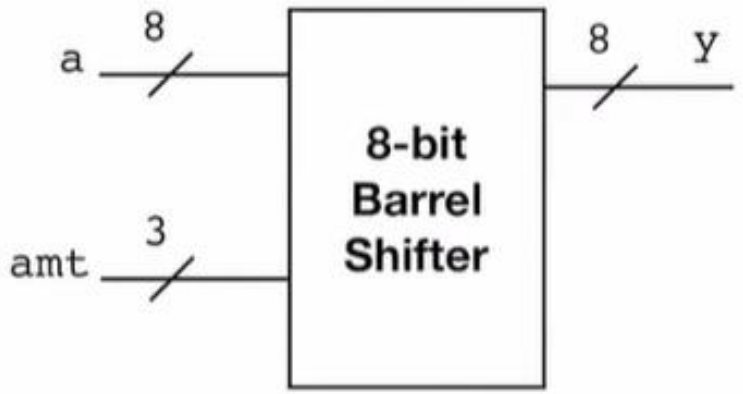
结果 = 000000000110（6）+ 111111010000（-48）= -42

位移器设计

- Shifter也是重要的数字电路模块之一

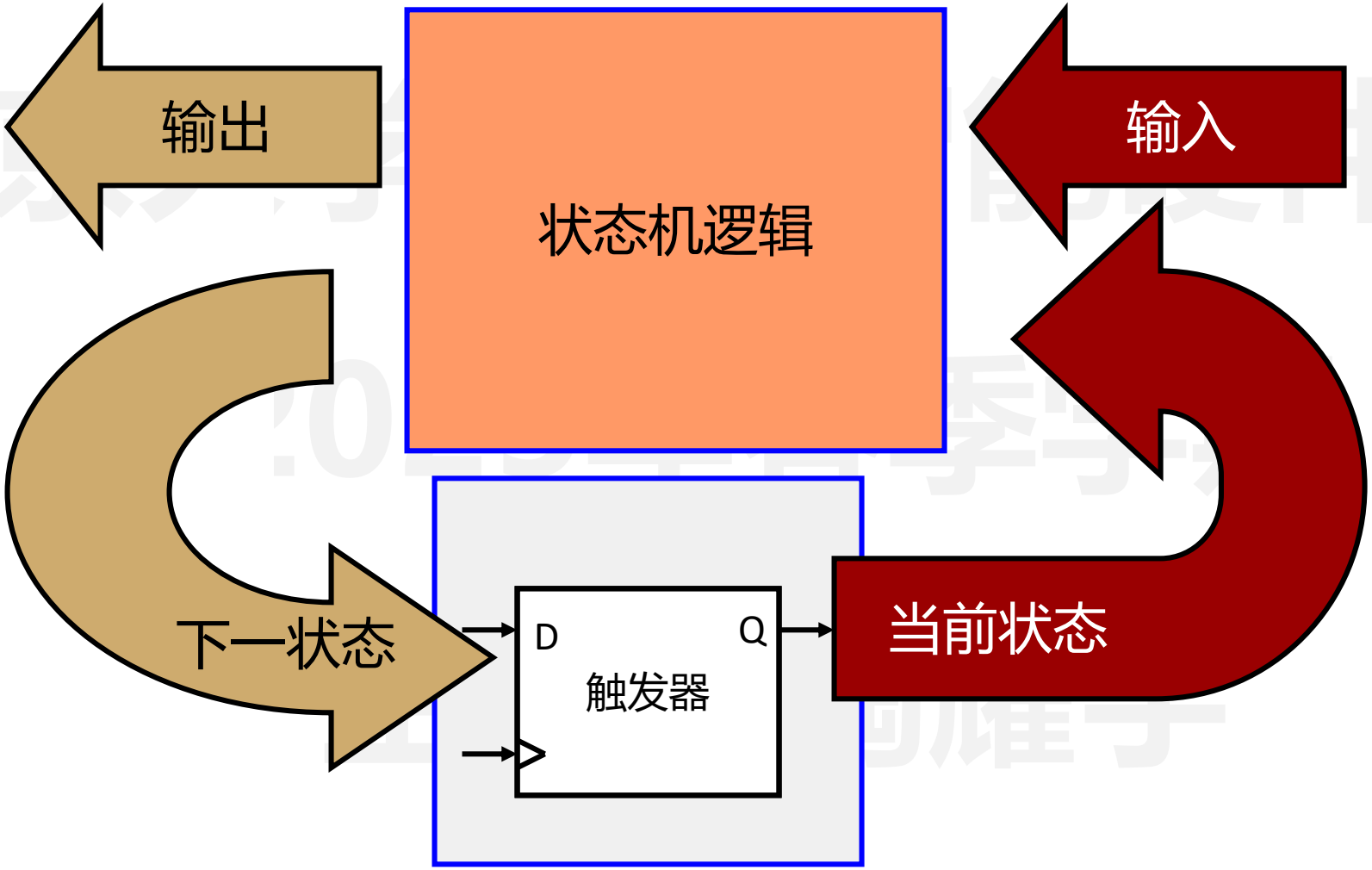
```
module barrel_shifter
    (
        input logic [7:0] a,
        input logic [2:0] amt,
        output logic [7:0] y
    );

    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

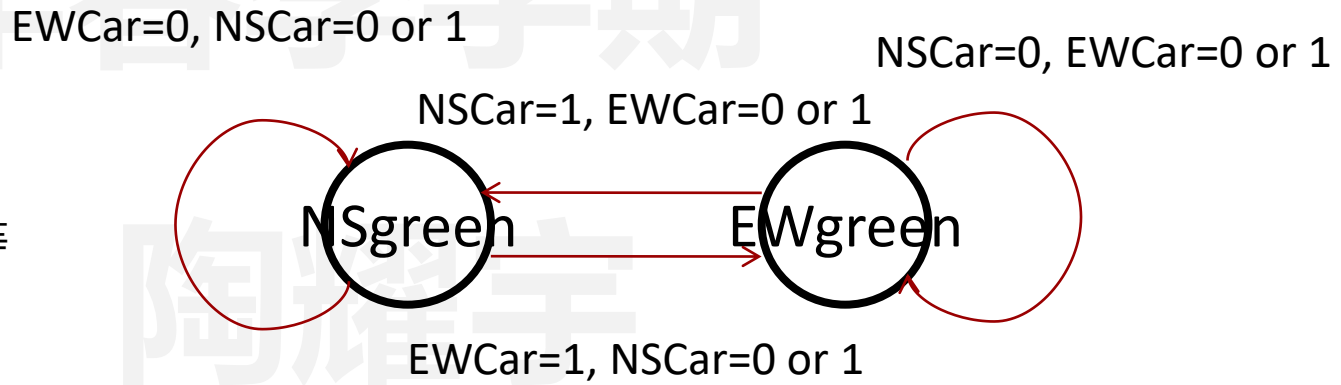
为什么需要状态机

- 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

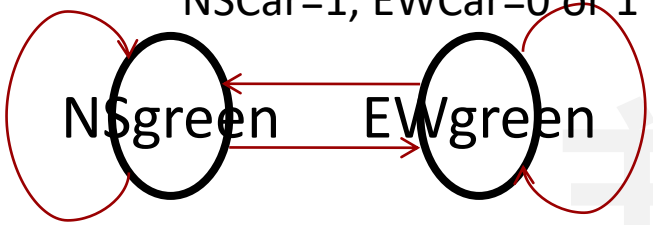
- 需要2个状态
  - Nsgreen、EWgreen

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

EWCar=0, NSCar=0 or 1

NSCar=0, EWCar=0 or 1

NSCar=1, EWCar=0 or 1



EWCar=1, NSCar=0 or 1

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

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

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

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

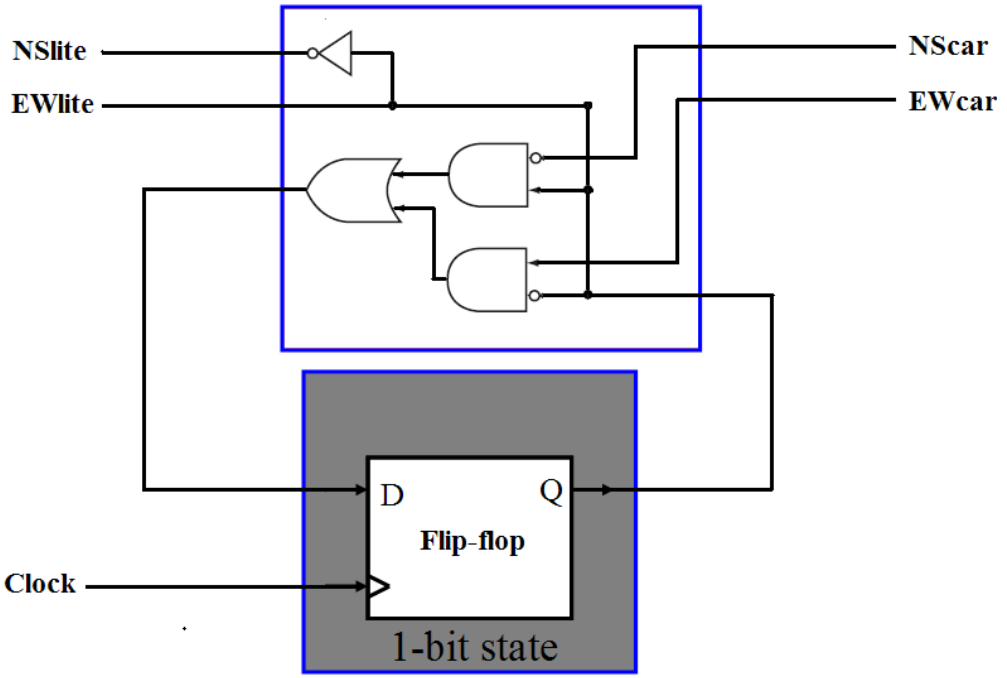
为什么需要状态机

- 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

北京大学 PEKING UNIVERSITY

- **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

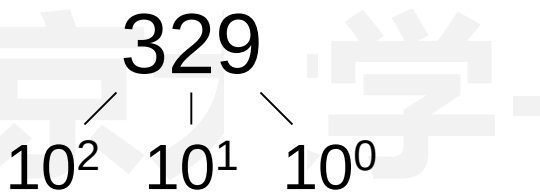
Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

Step 4 – 画出实际电路图

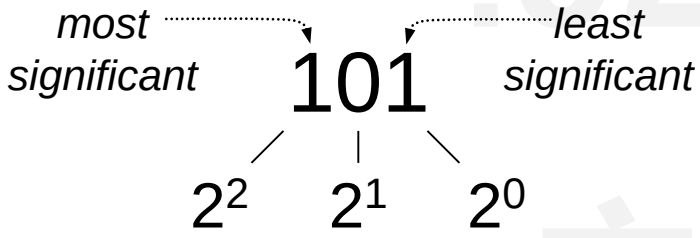
状态将在每一个时钟上升沿更新

量化与数据格式

- 最基础的二进制数据格式 – 原码（无符号数）



3x100 + 2x10 + 9x1 = 329



1x4 + 0x2 + 1x1 = 5

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

An *n*-bit unsigned integer

represents 2^n values:

from 0 to 2^n-1

量化与数据格式

- 最基础的二进制数据格式 – 原码（无符号数）

```
  10010        10010          1111
+  1001      +  1011        +    1
-------      -------        ------
  11011        11101          10000

                10111
              +   111
              -------
                11110
```

carry

量化与数据格式

- **有符号数的表现格式**

一个n bit数可以表示2^n不同的值

- 近一半赋值到正整数(1 ~ (2^{n-1}-1))

  近一半赋值到负整数((- (2^{n-1}-1)) ~ (-1))

- 还剩下两个值：表示0

正整数

同无符号数– 最高位为0

00101 = 5

负整数

对于原码来说，将最高位设为1代表负数，其他比特同无符号数一样

10101 = -5

量化与数据格式

- **有符号数的表现格式 – 补码**

原码(sign-magnitude)有什么问题？

0有两种重复表示 (+0 and –0)

计算电路复杂

对负数做加法时，实际上需要减法操作

需要考虑减法中的借位操作

```
  00101  (5)        01001  (9)
+ 11011  (-5)     + 10111  (-9)
  00000  (0)        00000  (0)
```

2的补码表示方法可以让计算电路更简单

- 对于每个正数 X，保证其相反数(-X)满足 X + (-X) = 0,其中的加法为忽略最高位进位的普通加法

量化与数据格式

- **有符号数的表现格式 – 补码**

若数字为正数或是0

- 正常二进制表示方法

若数字为负数

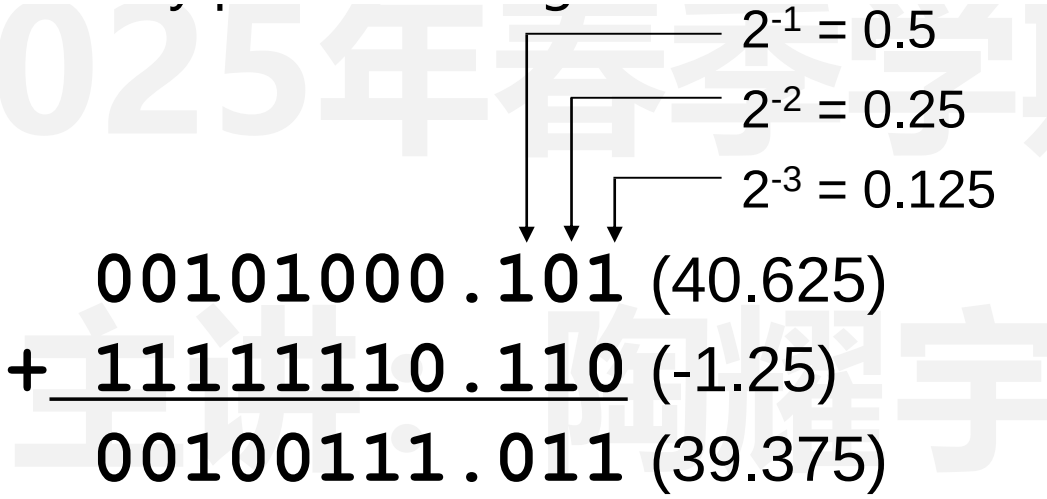
- 写出和它互为相反数的那个正数
- 翻转每一个比特
- 最后加1

```
  00101  (5)          01001  (9)
  11010  (1's comp)   10110  (1's comp)
+     1              +     1
  11011  (-5)         10111  (-9)
```

- 定点数 – Fixed-point

如何表示分数？

- 使用二进制小数点来分开2的正数次幂和负数次幂（同十进制相似）
- 2的补码加法和减法依然成立
  - 前提时小数点对齐



No new operations -- same as integer arithmetic.

量化与数据格式

- **特别大和特别小的数：浮点数 – Floating-point**

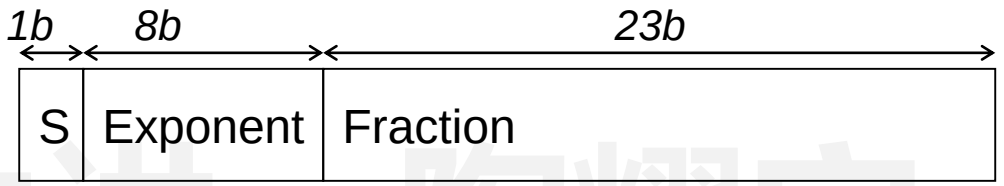
Large values: 6.023×10^{23} -- requires 79 bits

Small values: 6.626×10^{-34} -- requires >110 bits

使用科学计数法的等效： $F \times 2^{E}$

需要表示分数F (fraction), 指数E (exponent), and 符号位S(sign).

IEEE 754 浮点数标准(32-bits):



$$N = -1^{S} \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \ 1 \le \text{exponent} \le 254$$

$$N = -1^{S} \times 0.\text{fraction} \times 2^{-126}, \ \text{exponent} = 0$$

量化与数据格式

- 特别大和特别小的数：浮点数 – Floating-point

Single-precision IEEE floating point number:

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000… = 0.5 (decimal).

Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.