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

第四讲：复杂计算单元与指令集

主讲：陶耀宇

2026年春季

注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况

- **作业将在3月底-4月中旬、4月中旬-5月初、5月中旬-6月初**

  第一次作业时间：3.30-4.20（3周）

- **第1次lab时间：4月10日-5月10日**
- **第2次lab时间：5月10日-6月15日**
- **助教安排硬件Verilog/SystemVerilog编写及设计、验证全流程入门介绍，有兴趣的同学请积极参与！**

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

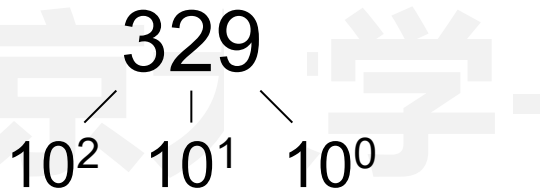
- **第1次lab时间：4月10-5月10**
  - **2个基础任务（50%+50%）+ 1个Bonus任务（可2选1，50%）**
  - 基础任务提供完整环境（Verilog+TB+SW Reference，10个测试激励）
    - **基础任务1：面向卷积加速的1D Winograd电路**
    - **基础任务2：面向三角函数加速的Cordic电路**
  - Bonus任务（利用基础任务代码，自行编写Verilog+TB+SW Reference和测试激励）
    - 将1D Winograd扩展至2D Winograd
    - 将Cordic扩展至支持离散傅里叶变化DFT
  - **请撰写<=3页的实验报告**，简单解释代码结构与设计思路

目 录

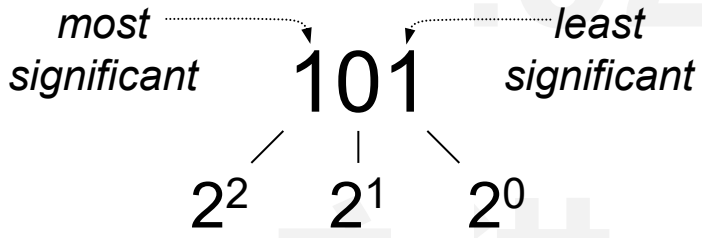
CONTENTS

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

- **有符号数的表现格式 – Sign and Magnitude**

  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**

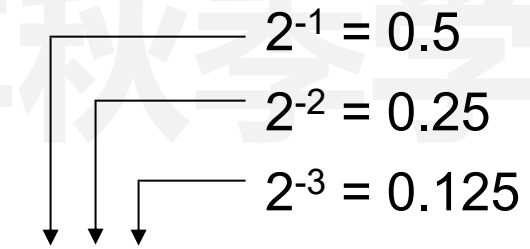
```
  00101  (5)          01001  (9)
  11010  (1's comp)   10110  (1's comp)
+     1              +     1
  11011  (-5)         10111  (-9)
```

量化与数据格式

- 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




```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  ------------
  00100111.011 (39.375)
```

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

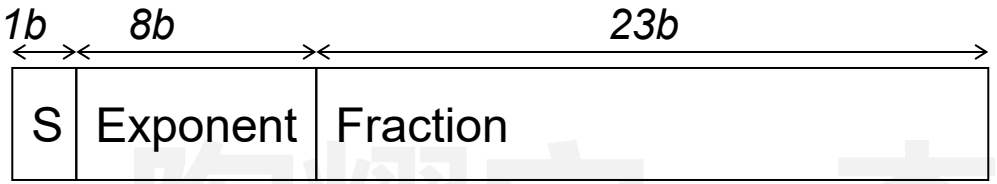
Large values: 6.023×10^{23} -- requires 79 bits

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

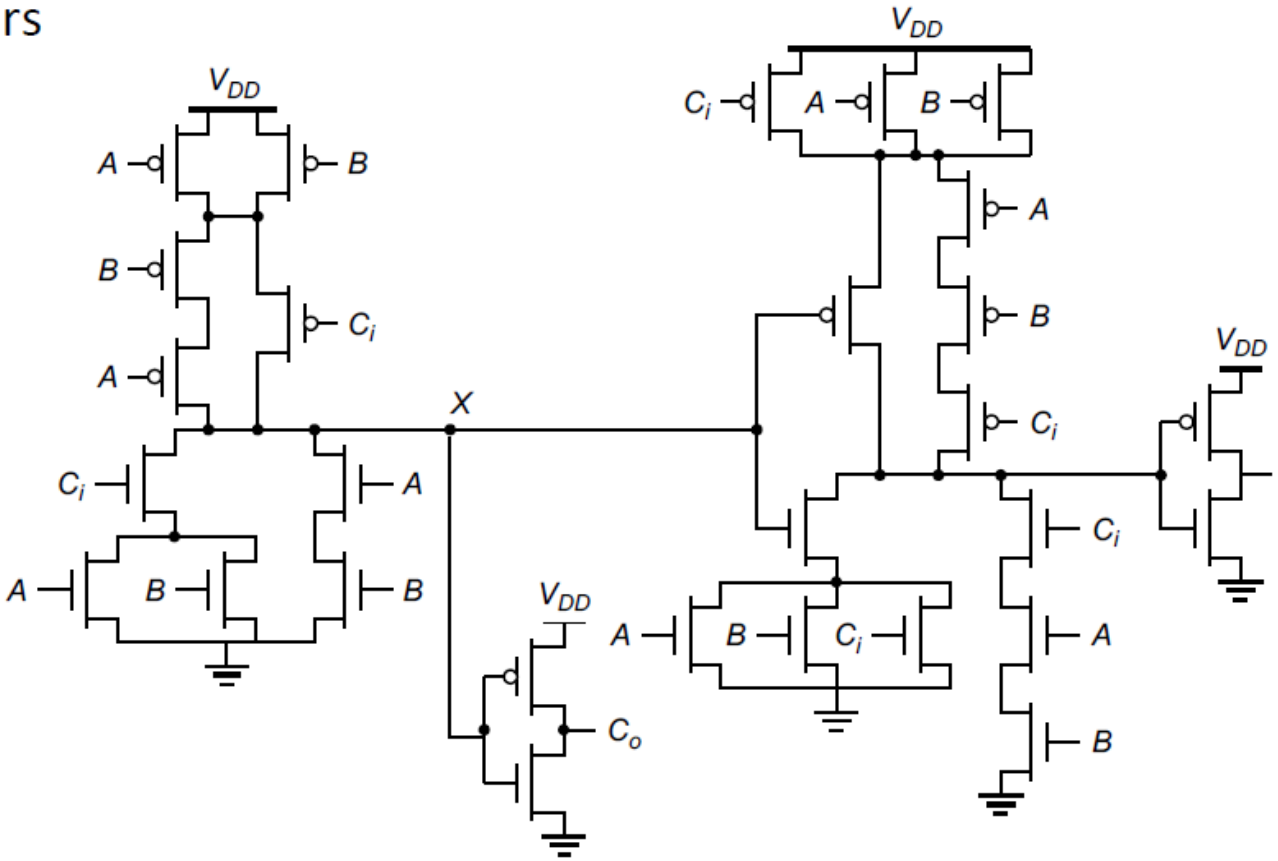
Single-precision IEEE floating point number:

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

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

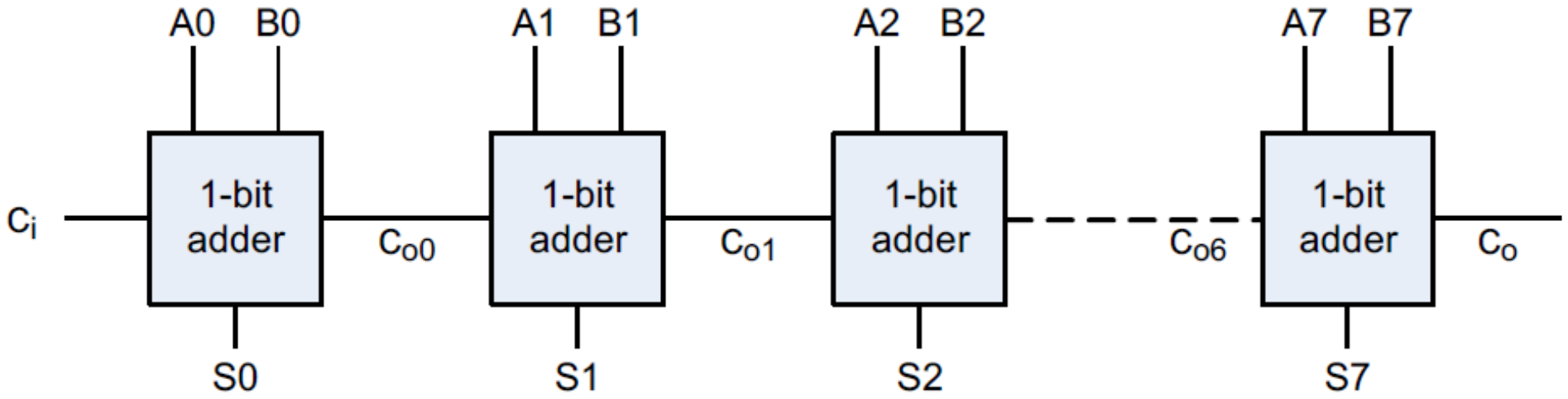
加法器设计

- 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

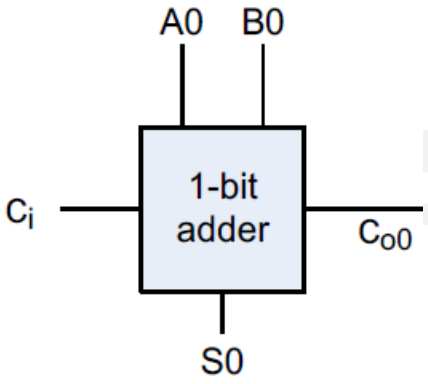
$$t_d = O(N)$$

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

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

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

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

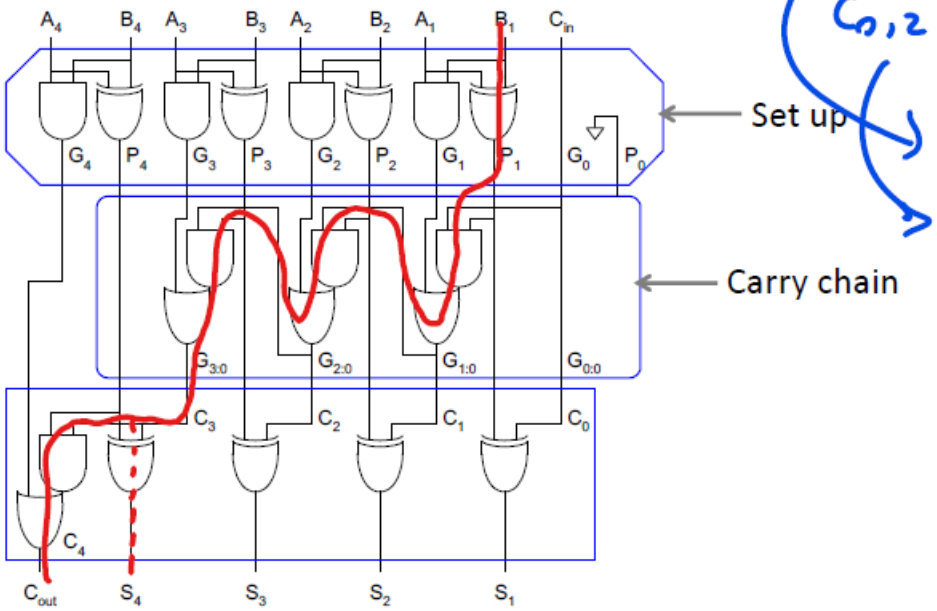
$P = A \oplus B$

$P = A + B$

- 基于PGK的加法器设计方法

Carry-Ripple using P and G

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



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

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

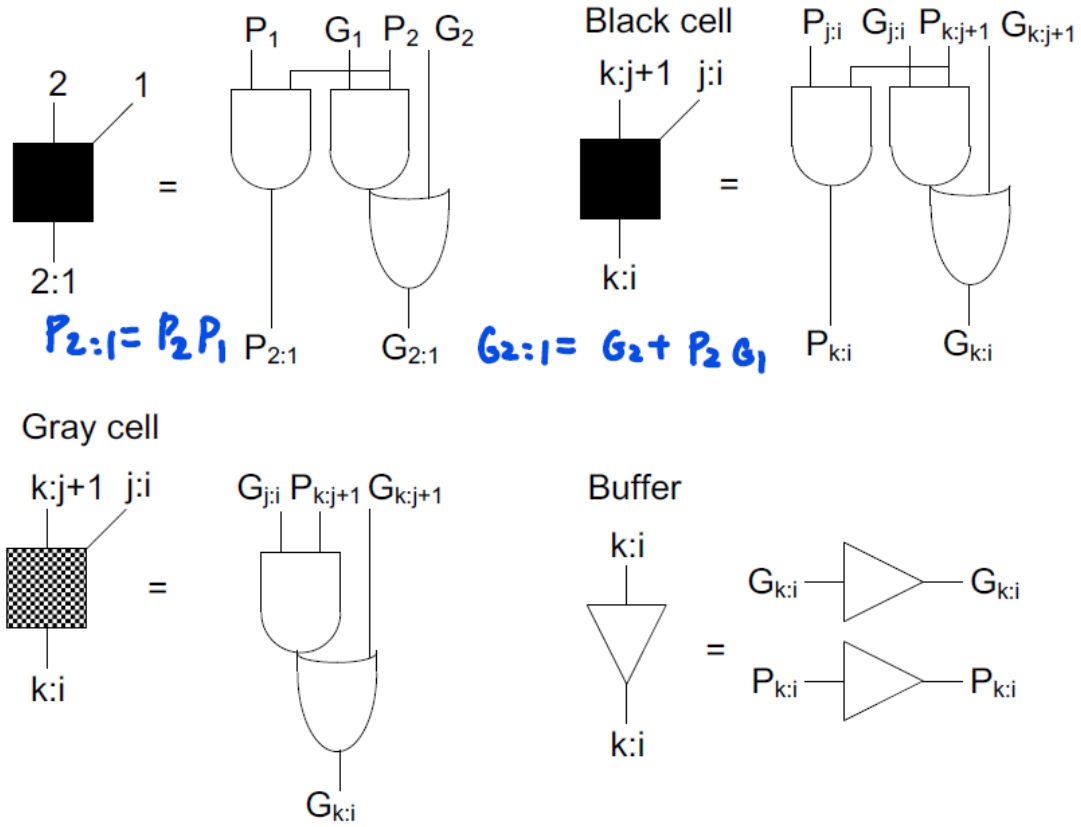
$G_{1:0} = G_1 + P_1 G_0$

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

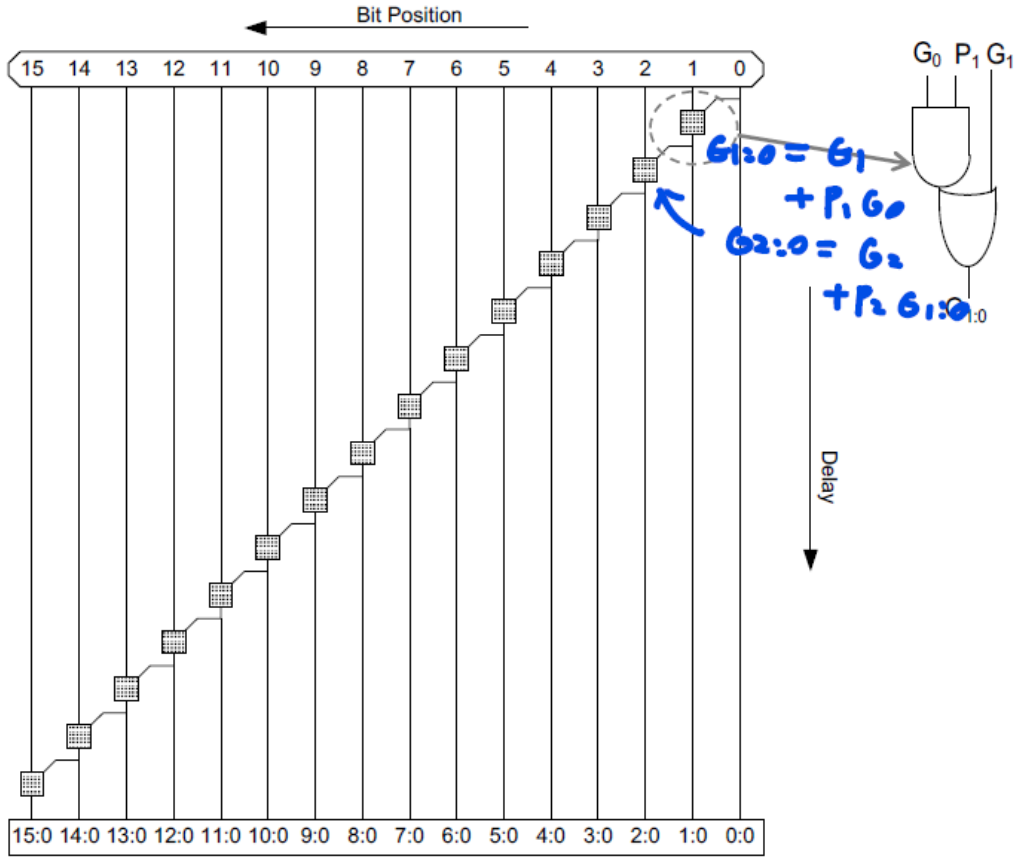
$t_{adder} = t_{setup} + (N-1)\, t_{carry} + \max(t_{carry}, t_{sum})$

加法器设计

- 基于PGK的加法器设计方法



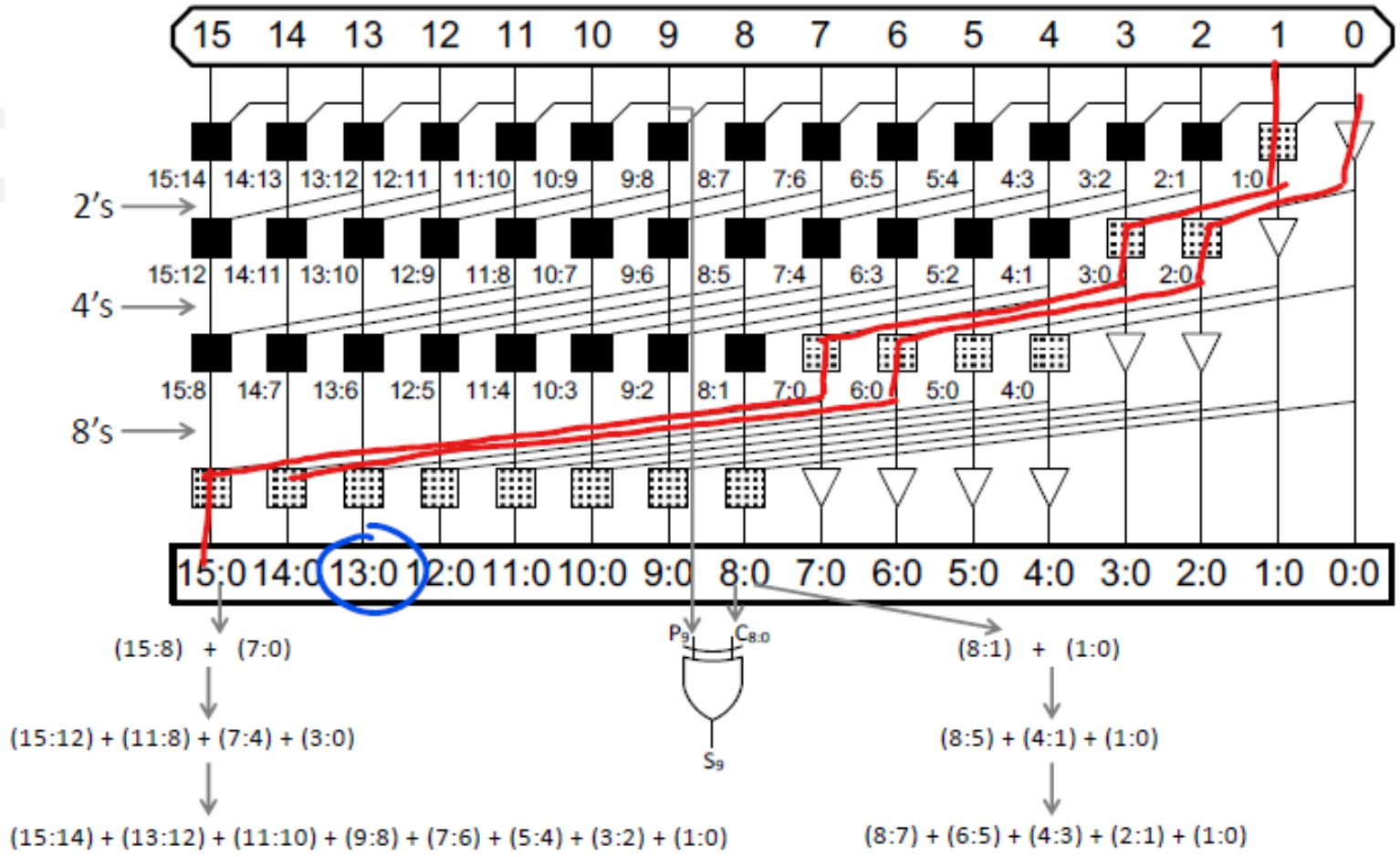
PG生成逻辑



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



$\log_2(N)$

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

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

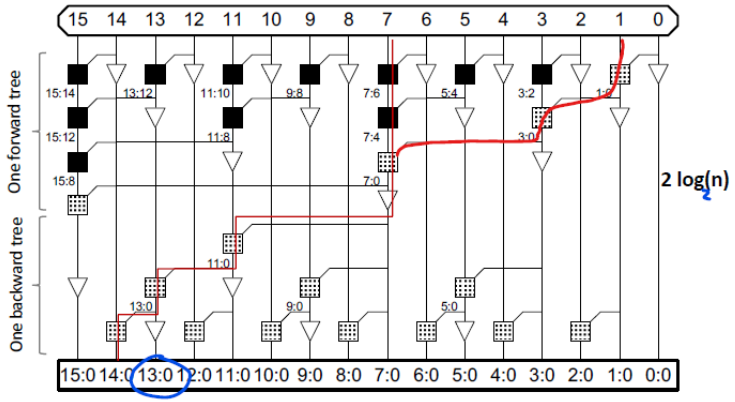
$P_{13:6} = P_{13:10}\,P_{9:6}$

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

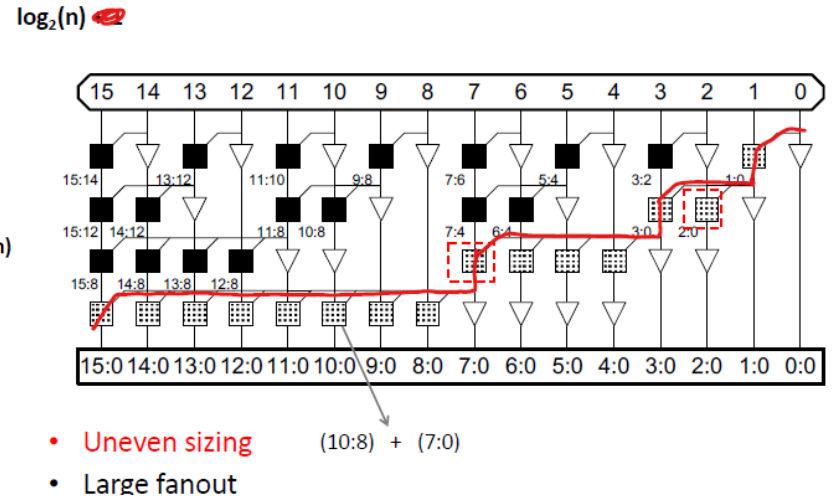
加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器

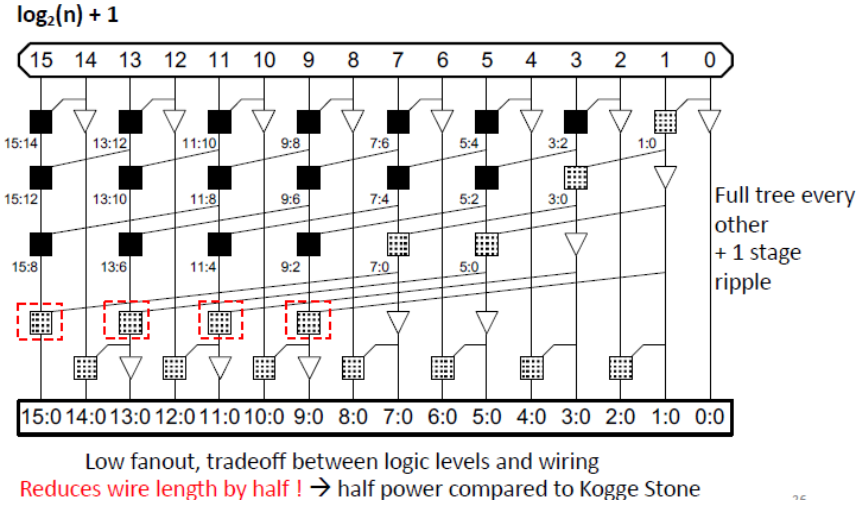
Brent-Kung



Sklansky



Han-Carlson



- 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

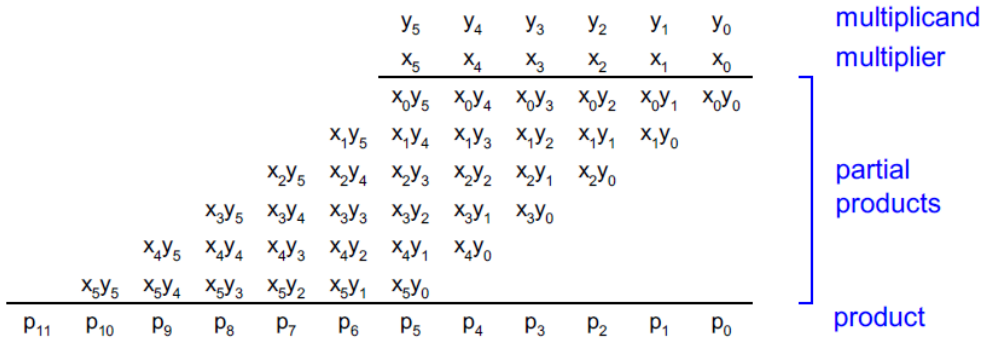
乘法器设计

- 乘法器设计的核心是部分和累加

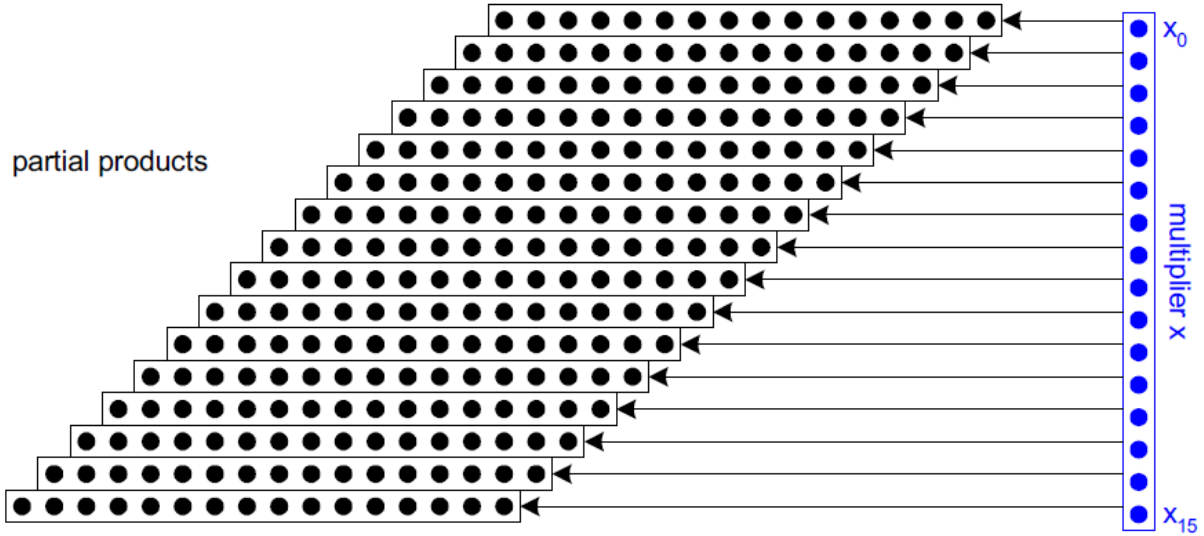
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

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

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

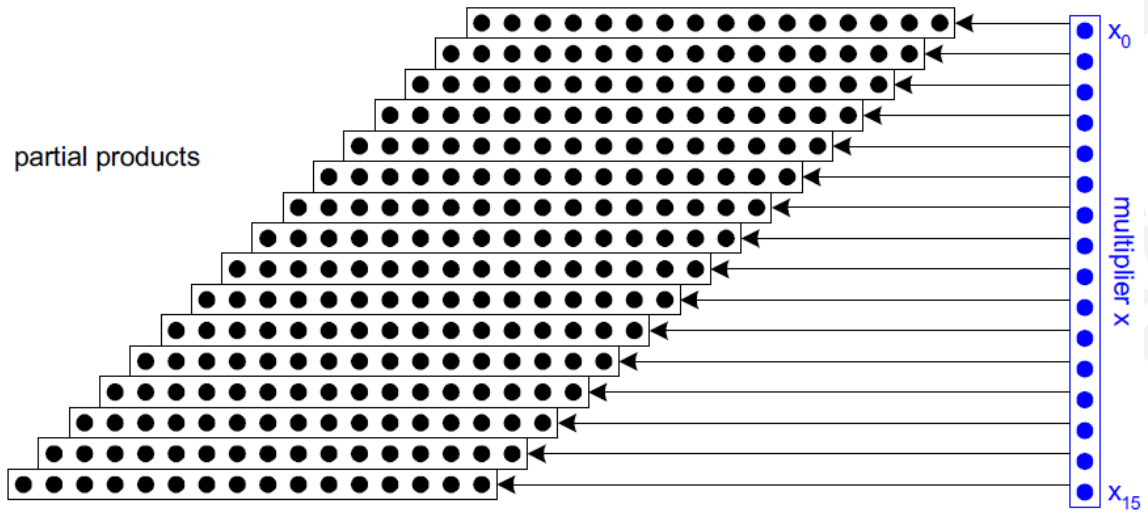


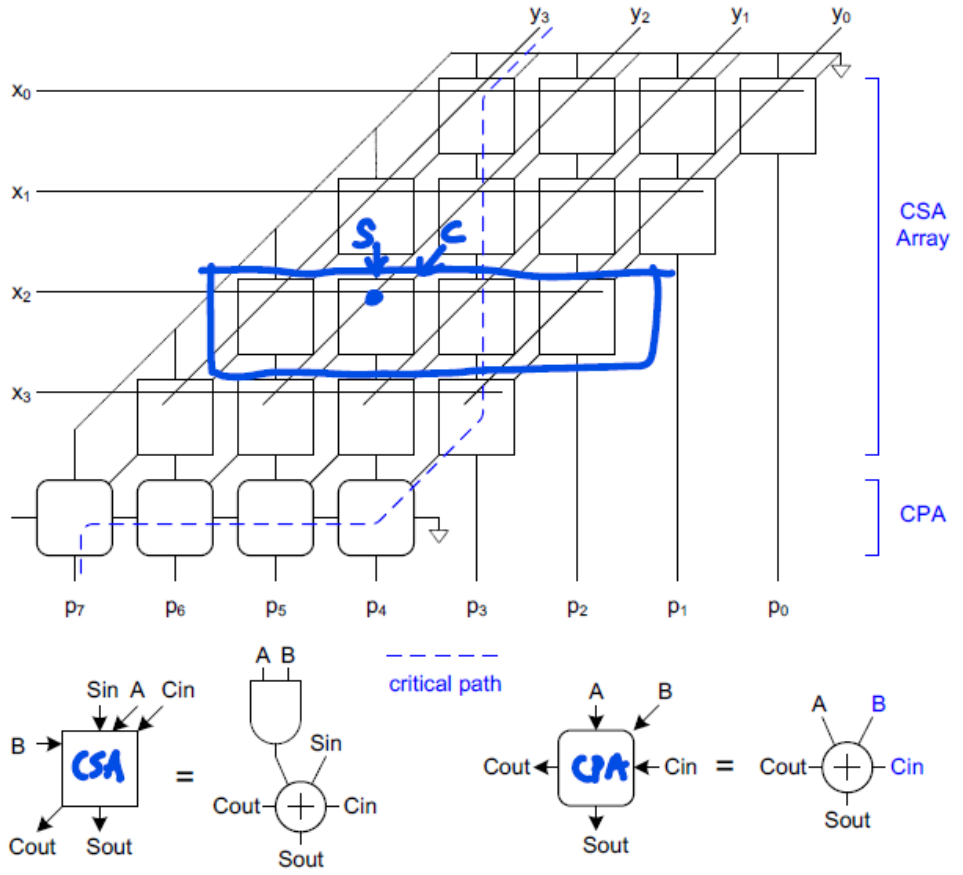
Each dot represents a bit



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

X | PP
(0 0) | 0
(0 1) | Y
(1 0) | 2Y (4Y-2Y)
(1 1) | 3Y = (4Y-Y)

- 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?

$$x \times y$$

$$y = -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + \cdots 2^2 y_2 + 2y_1 + y_0 + y_{-1}, \quad y_{-1} = 0$$

$$\begin{aligned} y = {} & -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + 2^{n-2}y_{n-2} - 2^{n-2}y_{n-2} + 2^{n-3}y_{n-3} + 2^{n-3}y_{n-3} \\ & - 2^{n-3}y_{n-3} + \cdots + 2y_1 + 2y_1 - 2y_1 + y_0 + y_0 - y_0 + y_{-1} \end{aligned}$$

$$\begin{aligned} = {} & -2^{n-1}y_{n-1} + 2 \times 2^{n-2}y_{n-2} - 2^{n-2}y_{n-2} + 2 \times 2^{n-3}y_{n-3} - 2^{n-3}y_{n-3} + \cdots + 2 \times 2y_1 \\ & - 2y_1 + 2 \times y_0 - y_0 + y_{-1} \end{aligned}$$

$$\begin{aligned} = {} & -2^{n-1}y_{n-1} + 2^{n-1}y_{n-2} - 2^{n-2}y_{n-2} + 2^{n-2}y_{n-3} - 2^{n-3}y_{n-3} + \cdots + 2^2 y_1 - 2y_1 \\ & + 2y_0 - y_0 + y_{-1} \end{aligned}$$

$$= 2^{n-1}(-y_{n-1} + y_{n-2}) + 2^{n-2}(-y_{n-2} + y_{n-3}) + \cdots 2(-y_1 + y_0) + (-y_0 + y_{-1})$$

部分和累加次数是否减少?

乘法器设计

- 如何减少部分和累加的次数?

$$y = -2 \times 2^{n-2} y_{n-1} + 2^{n-2} y_{n-2} + 2^{n-3} y_{n-3} + 2^{n-3} y_{n-3} - 2^{n-3} y_{n-3} + 2^{n-4} y_{n-4}$$
$$+ 2^{n-5} y_{n-5} + 2^{n-5} y_{n-5} - 2^{n-5} y_{n-5} + \cdots + 2^3 y_3 - 2^3 y_3 + 2^2 y_2 + 2y_1$$
$$+ 2y_1 - 2y_1 + y_0 + y_{-1}$$

$$= -2 \times 2^{n-2} y_{n-1} + 2^{n-2} y_{n-2} + 2 \times 2^{n-3} y_{n-3} - 2 \times 2^{n-4} y_{n-3} + 2^{n-4} y_{n-4}$$
$$+ 2 \times 2^{n-5} y_{n-5} - 2 \times 2^{n-6} y_{n-5} + \cdots - 2 \times 2^2 y_3 + 2^2 y_2 + 2 \times 2y_1 - 2y_1$$
$$+ y_0 + y_{-1}$$

$$= -2 \times 2^{n-2} y_{n-1} + 2^{n-2} y_{n-2} + 2^{n-2} y_{n-3} - 2 \times 2^{n-4} y_{n-3} + 2^{n-4} y_{n-4} + 2^{n-4} y_{n-5}$$
$$- 2 \times 2^{n-6} y_{n-5} + \cdots - 2 \times 2^2 y_3 + 2^2 y_2 + 2^2 y_1 - 2y_1 + y_0 + y_{-1}$$

$$= 2^{n-2}(-2y_{n-1} + y_{n-2} + y_{n-3}) + 2^{n-4}(-2y_{n-3} + y_{n-4} + y_{n-5}) + \cdots + 2^2(-2y_3 +$$
$$y_2 + y_1) + (-2y_1 + y_0 + y_{-1})$$

部分和累加次数是否减少?

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

4y-2y 1 0
4y-y 1 1

Y -Y Y

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

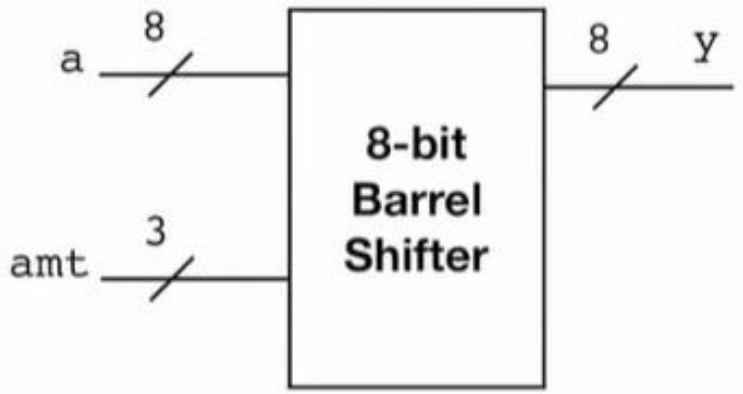
Step3：Q = 0001110

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
```



复杂模块电路设计1 - 卷积加速模块

北京大学 PEKING UNIVERSITY

- 卷积 – AI计算中最常用的算子

卷积最初是一种信号处理滤波操作 –

“AI神经网络也可看作是一种滤波”

WinoGrad算法起源于1980年，是Shmuel Winograd提出用来减少FIR滤波器计算量的一个算法

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

输入尺寸 = 2 + 3 -1 = 4

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

输出尺寸　卷积核尺寸

6次乘法

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

存在重复元素

Winograd

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1 + d2)\frac{g0 + g1 + g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2 - d1)\frac{g0 - g1 + g2}{2}$$

预算好一次g的加减后可重复复用 -> 4次乘法

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

element-wise multiplication（Hadamard product）

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1 + d2)\frac{g0+g1+g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2 - d1)\frac{g0-g1+g2}{2}$$

$$Y = A^T\left[(Gg) \odot (B^T d)\right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

1D Winograd

卷积加速模块设计

北京大学 PEKING UNIVERSITY

- **卷积 – AI计算中最常用的算子**

将一维卷积运算定义为$F(m,r)$，m为Output Size，r为Filter Size，则输入信号的长度为$m+r-1$，卷积运算是对应位置相乘然后求和，**输入信号每个位置至少要参与1次乘法**，所以乘法数量最少与输入信号长度相同，记为

$$\mu(F(m,r))=m+r-1$$

在行列上分别进行一维卷积运算，可得到二维卷积，记为$F(m\times n,r\times s)$，输出为$m\times n$，卷积核为$r\times s$，则输入信号为$(m+r-1)(n+s-1)$，乘法数量至少为

$$\begin{aligned}\mu(F(m\times n,r\times s))&=\mu(F(m,r))\mu(F(n,s))\\&=(m+r-1)(n+s-1)\end{aligned}$$

若是直接按滑动窗口方式计算卷积，一维时需要$m\times r$次乘法，二维时需要$m\times n\times r\times s$次乘法，**远大于上面计算的最少乘法次数**。

使用Winograd算法计算卷积快在哪里？一言以蔽之：**快在减少了乘法的数量**，将乘法数量减少至$m+r-1$或$(m+r-1)(n+s-1)$。

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

$$Y = A^T\left[(Gg)\odot(B^T d)\right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

g：表示卷积核

d：表示输入信号

G：卷积核变换矩阵，尺寸为$(m+r-1)\times r$

B^T：输入变换矩阵，尺寸$(m+r-1)\times(m+r-1)$

A^T：输出变换矩阵，尺寸$m\times(m+r-1)$

计算过程可分为4步：

（1）输入变换

（2）卷积核变换　　1D Winograd

（3）外积

（4）输出变换

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

$$Y = A^T\left[(Gg) \odot (B^T d)\right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

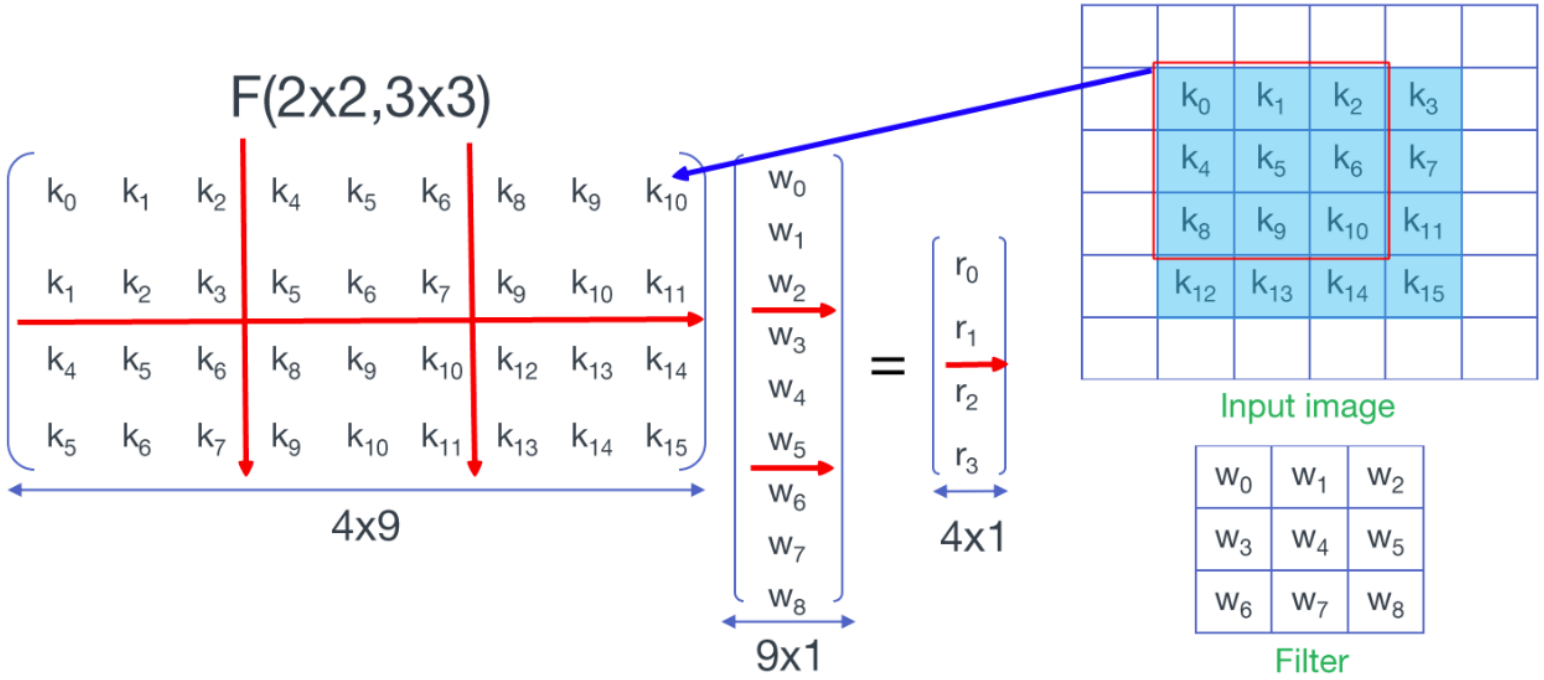
$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

2D Winograd

A minimal 1D algorithm F(m, r) is **nested with itself** to obtain a minimal 2D algorithm (m×m, r×r).

$$Y = A^T\left[\left[GgG^T\right] \odot \left[B^T dB\right]\right]A$$

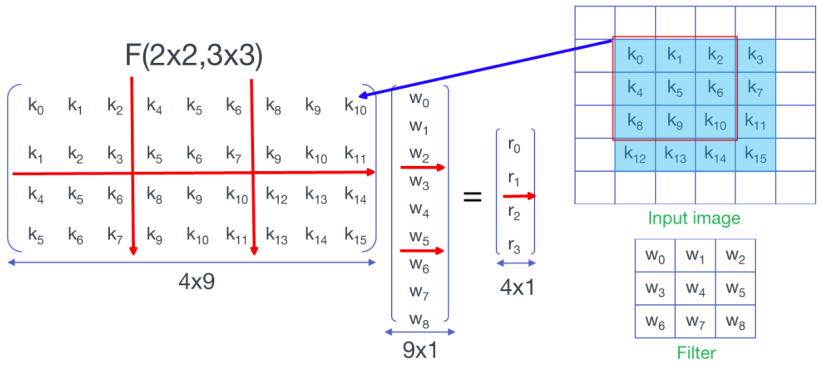
g为$r \times r$ Filter，d为$(m+r-1) \times (m+r-1)$



卷积加速模块设计

- **卷积 – AI计算中最常用的算子**

令$D_0 = [k_0, k_1, k_2, k_3]^T$，即窗口中的第0行元素，$D_1$ D_2 D_3表示第1、2、3行；$W_0 = [w_0, w_1, w_2]^T$，

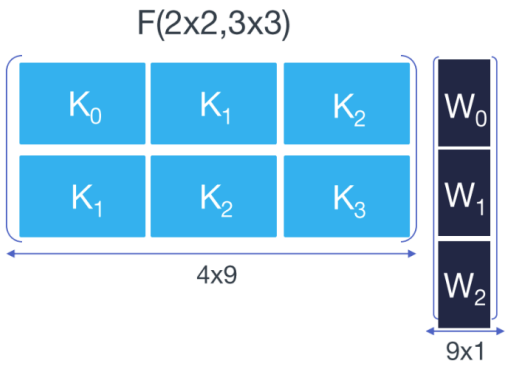


$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} K_0W_0 + K_1W_1 + K_2W_2 \\ K_1W_0 + K_2W_1 + K_3W_2 \end{bmatrix}$$

$$= \begin{bmatrix} A^T\left[(GW_0) \odot (B^TD_0)\right] + A^T\left[(GW_1) \odot (B^TD_1)\right] + A^T\left[(GW_2) \odot (B^TD_2)\right] \\ A^T\left[(GW_0) \odot (B^TD_1)\right] + A^T\left[(GW_1) \odot (B^TD_2)\right] + A^T\left[(GW_2) \odot (B^TD_3)\right] \end{bmatrix}$$

$$= A^T\left[\left[G[W_0\ W_1\ W_2]G^T\right] \odot \left[B^T[d_0\ d_1\ d_2\ d_3]B\right]\right] A$$

$$= A^T\left[\left[GgG^T\right] \odot \left[B^TdB\right]\right] A$$



$$\begin{pmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{pmatrix} \begin{pmatrix} W_0 \\ W_1 \\ W_2 \end{pmatrix} = \begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{pmatrix}$$

Matrix multiply F(2,3)! **4** multiplications

$$M_0 = (K_0 - K_2) \cdot W_0$$

$$M_1 = (K_1 + K_2) \cdot \frac{W_0 + W_1 + W_2}{2}$$

$$M_3 = (K_1 - K_3) \cdot W_2$$

$$M_2 = (K_2 - K_1) \cdot \frac{W_0 - W_1 + W_2}{2}$$

卷积加速模块设计

- NVDLA中的Winograd卷积核设计

在MAC并行计算单元中计算

$$A^T\left[\left[GgG^T\right]\odot\left[B^TdB\right]\right]A$$

预先算好　　在输入datapath计算

For m = 3, r = 2

$$B^T=\begin{bmatrix}1&0&-1&0\\0&1&1&0\\0&-1&1&0\\0&-1&0&1\end{bmatrix},G=\begin{bmatrix}1&0\\\frac{1}{2}&\frac{1}{2}\\\frac{1}{2}&-\frac{1}{2}\\0&1\end{bmatrix}$$

$$A^T=\begin{bmatrix}1&1&1&0\\0&1&-1&0\\0&1&1&1\end{bmatrix}$$

For m = 4, r = 3

$$B^T=\begin{bmatrix}4&0&-5&0&1&0\\0&-4&-4&1&1&0\\0&4&-4&-1&1&0\\0&-2&-1&2&1&0\\0&2&-1&-2&1&0\\0&4&0&-5&0&1\end{bmatrix}$$

$$G=\begin{bmatrix}\frac{1}{4}&0&0\\-\frac{1}{6}&-\frac{1}{6}&-\frac{1}{6}\\-\frac{1}{6}&\frac{1}{6}&-\frac{1}{6}\\\frac{1}{24}&\frac{1}{12}&\frac{1}{6}\\\frac{1}{24}&-\frac{1}{12}&\frac{1}{6}\\0&0&1\end{bmatrix}$$

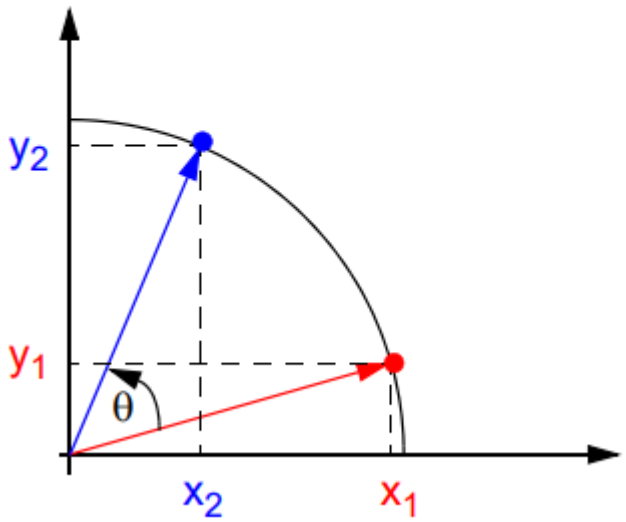
$$A^T=\begin{bmatrix}1&1&1&1&1&0\\0&1&-1&2&-2&0\\0&1&1&4&4&0\\0&1&-1&8&-8&1\end{bmatrix}$$

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

$$x_2 = x_1\cos\theta - y_1\sin\theta$$
$$y_2 = x_1\sin\theta + y_1\cos\theta$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$



$$x_2 = x_1\cos\theta - y_1\sin\theta = \cos\theta(x_1 - y_1\tan\theta)$$
$$y_2 = x_1\sin\theta + y_1\cos\theta = \cos\theta(y_1 + x_1\tan\theta)$$

$$\hat{x}_2 = \cancel{\cos\theta}(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$
$$\hat{y}_2 = \cancel{\cos\theta}(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$

当θ足够小接近于0时，先忽略cosθ（后面会scale回来）

伪旋转（pseudo rotations）

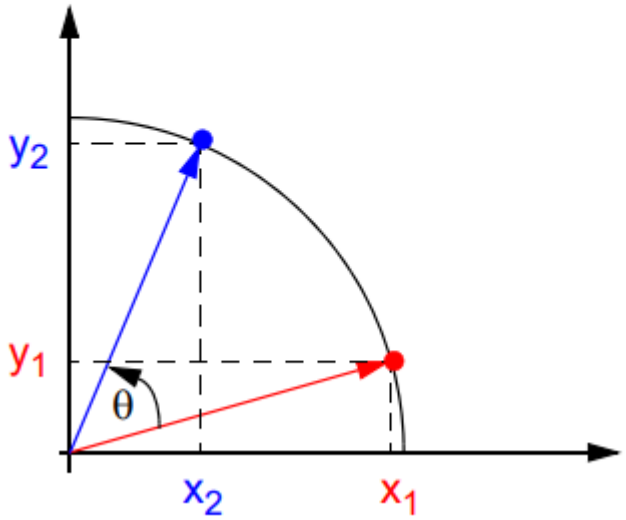
复杂模块电路设计2 - CORDIC模块设计

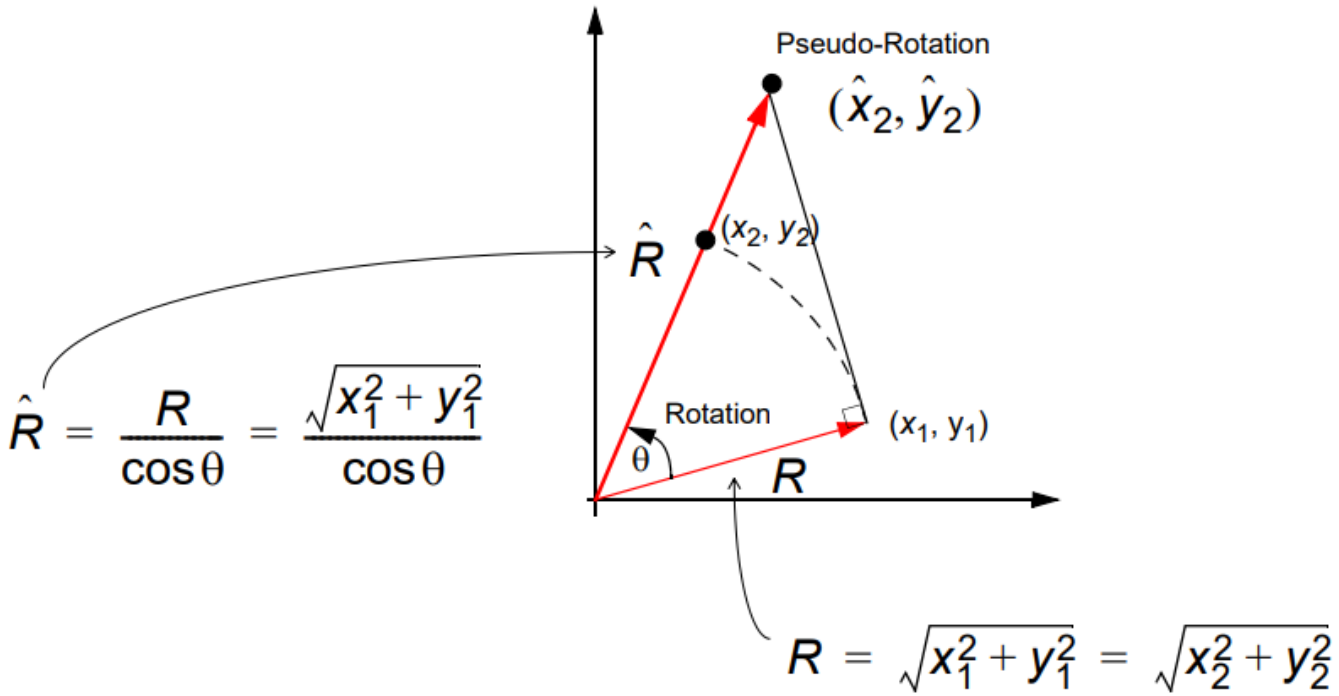
- CORDIC可以用来实现多种复杂非线性函数

$$x_2 = x_1\cos\theta - y_1\sin\theta$$
$$y_2 = x_1\sin\theta + y_1\cos\theta$$

伪旋转（pseudo rotations）

$$\hat{x}_2 = \cancel{\cos\theta}(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$
$$\hat{y}_2 = \cancel{\cos\theta}(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$





复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

伪旋转（pseudo rotations）

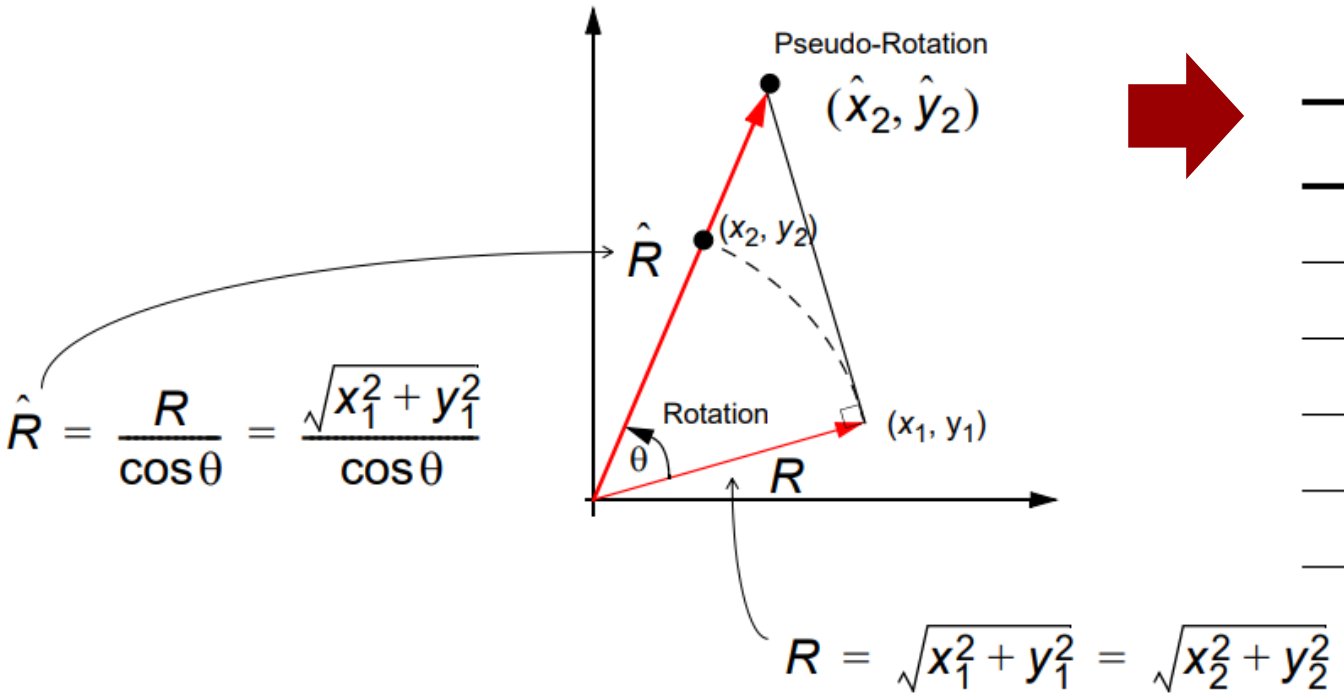
$$\hat{x}_2 = \cos\theta(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$

$$\hat{y}_2 = \cos\theta(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$

选择旋转角度 $\tan\theta^i = 2^{-i}$

$$\hat{x}_2 = x_1 - y_1\tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1\tan\theta = y_1 + x_1 2^{-i}$$



i	θ^i (Degrees)	$\tan\theta^i = 2^{-i}$
0	45.0	1
1	26.555051177...	0.5
2	14.036243467...	0.25
3	7.125016348...	0.125
4	3.576334374...	0.0625

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

1st iteration: rotate by 45°; 2nd iteration: rotate by 26.6°, 3rd iteration: rotate by 14°

i	tanθ	Angle, θ	cosθ
1	1	45.0000000000	0.707106781
2	0.5	26.5650511771	0.894427191
3	0.25	14.0362434679	0.9701425
4	0.125	7.1250163489	0.992277877
5	0.0625	3.5763343750	0.998052578
6	0.03125	1.7899106082	0.999512076
7	0.015625	0.8951737102	0.999877952
8	0.0078125	0.4476141709	0.999969484
9	0.00390625	0.2238105004	0.999992371
10	0.001953125	0.1119056771	0.999998093
11	0.000976563	0.0559528919	0.999999523
12	0.000488281	0.0279764526	0.999999881
13	0.000244141	0.0139882271	0.99999997

cos 45 x cos 26.5 x cos 14.03 x cos 7.125 ... x cos 0.0139 = 0.607252941

13次伪旋转后，需要乘以

1/0.607252941 =

1.6467602

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

$$\hat{x}_2 = x_1 - y_1 \tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1 \tan\theta = y_1 + x_1 2^{-i}$$

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i} y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i} x^{(i)})$$

$$z^{(i+1)} = z^{(i)} - d_i \theta^{(i)} \quad \text{(Angle Accumulator)}$$

where d_i = +/- 1

The symbol d_i is a decision operator and is used to decide which direction to rotate.

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions

复杂模块电路设计2 - CORDIC模块设计

北京大学 PEKING UNIVERSITY

- CORDIC可以用来实现多种复杂非线性函数

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

where d_i = +/- 1

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions

Scaling Factor

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n (\sqrt{1+2^{(-2i)}})$$

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n \left(\sqrt{1+\tan^2\theta^{(i)}}\right) = \prod_n \left(\sqrt{1+2^{(-2i)}}\right)$$

$$K_n \to 1.6476 \text{ as } n \to \infty$$

$$1/K_n \to 0.6073 \text{ as } n \to \infty$$

n = number of iterations

复杂模块电路设计2 - CORDIC模块设计

北京大学 PEKING UNIVERSITY

- CORDIC可以用来实现多种复杂非线性函数

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

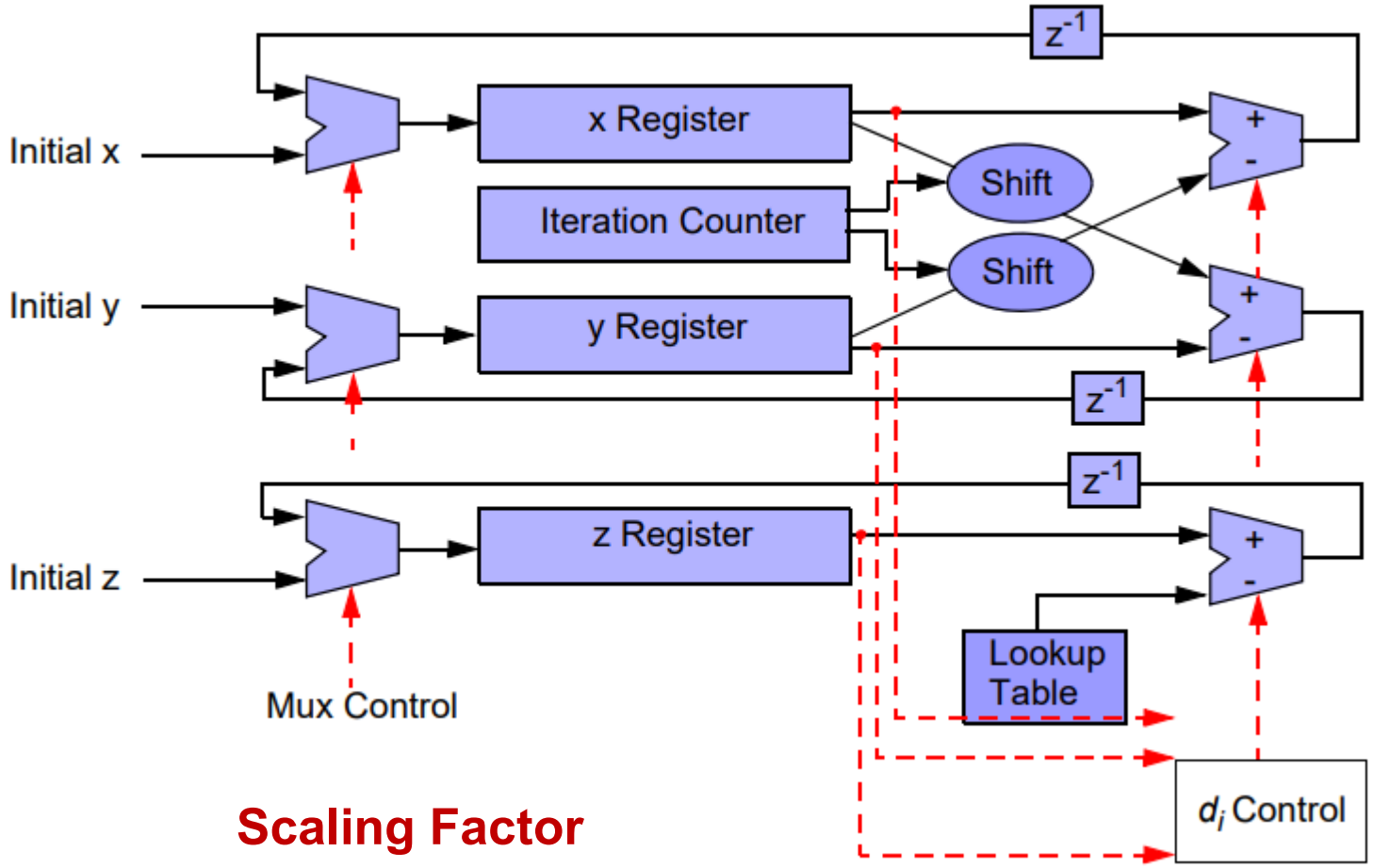
$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

where d_i = +/- 1

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions



Scaling Factor

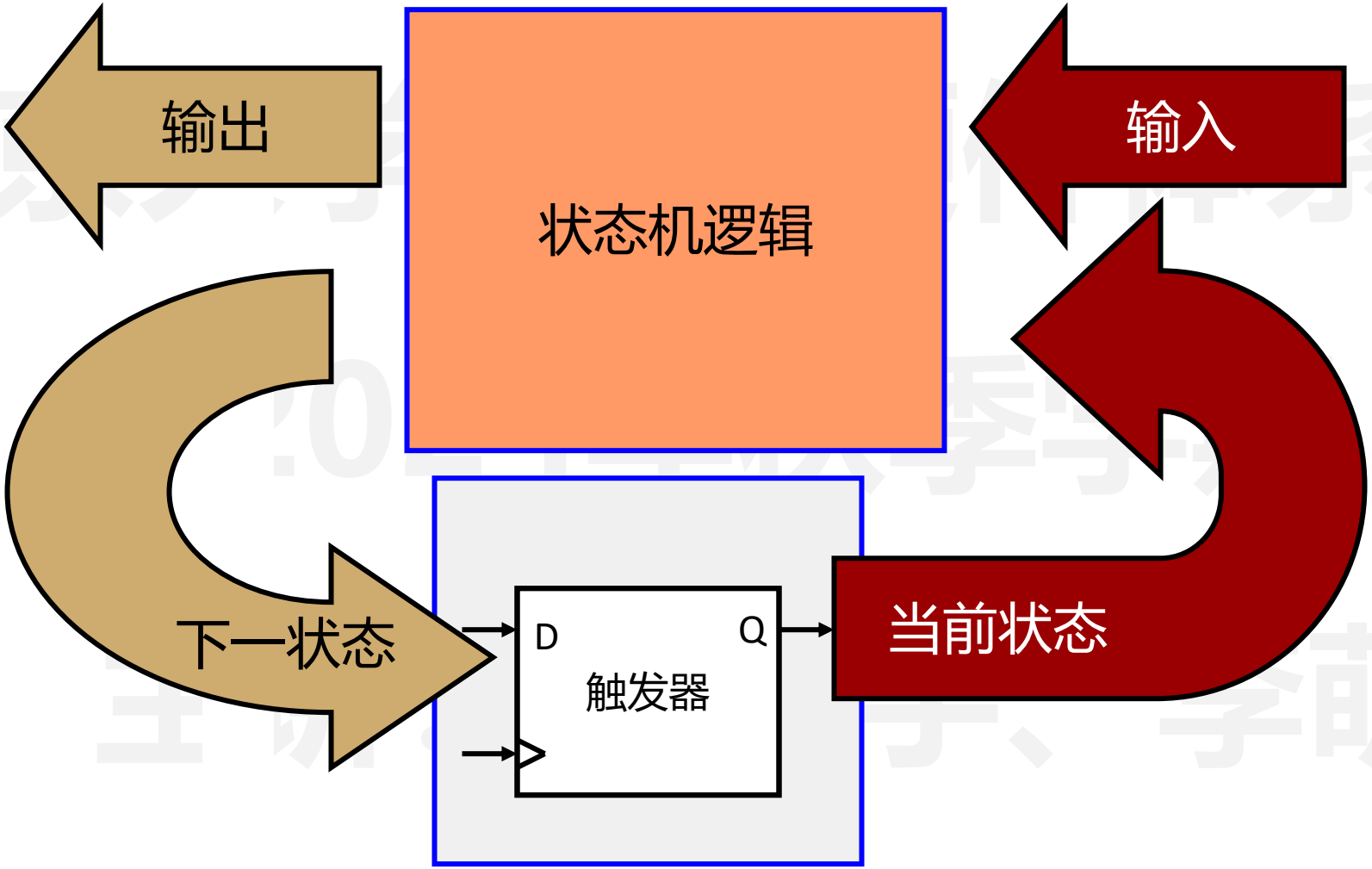
$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n (\sqrt{1+2^{(-2i)}})$$

目 录

CONTENTS

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

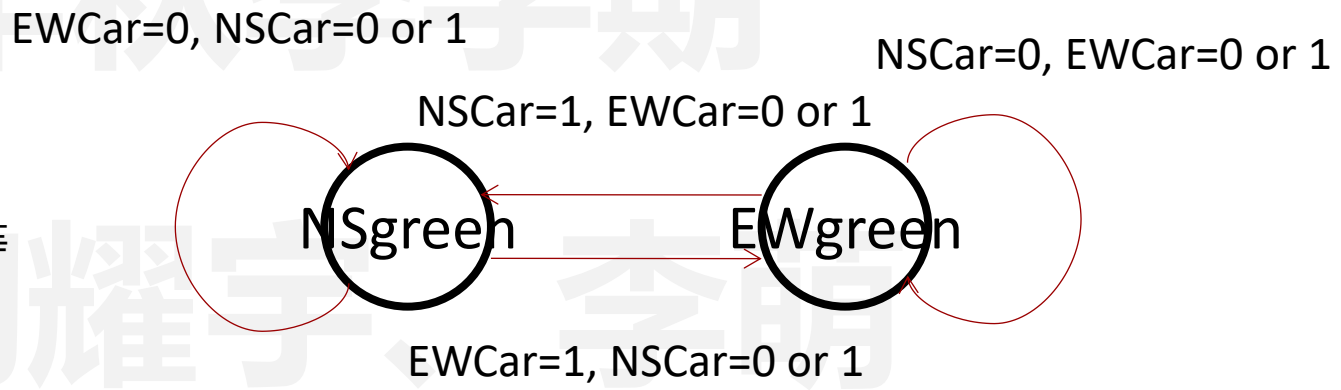
为什么需要状态机

- 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



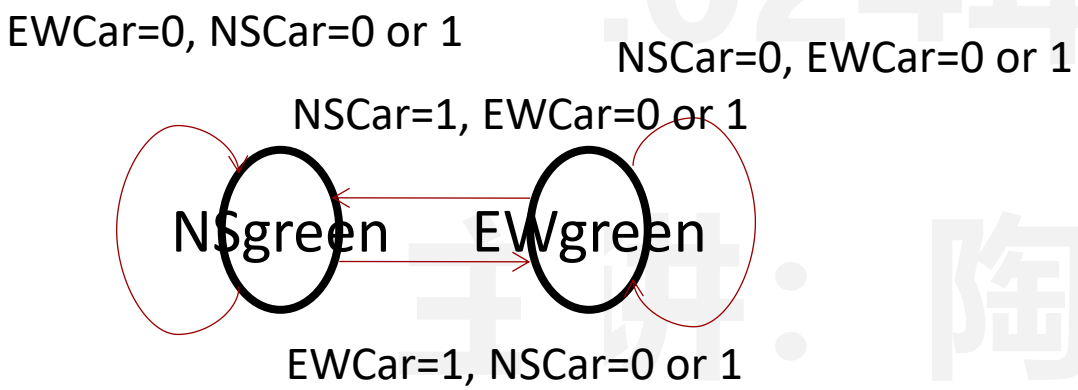
为什么需要状态机

- 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

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

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

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

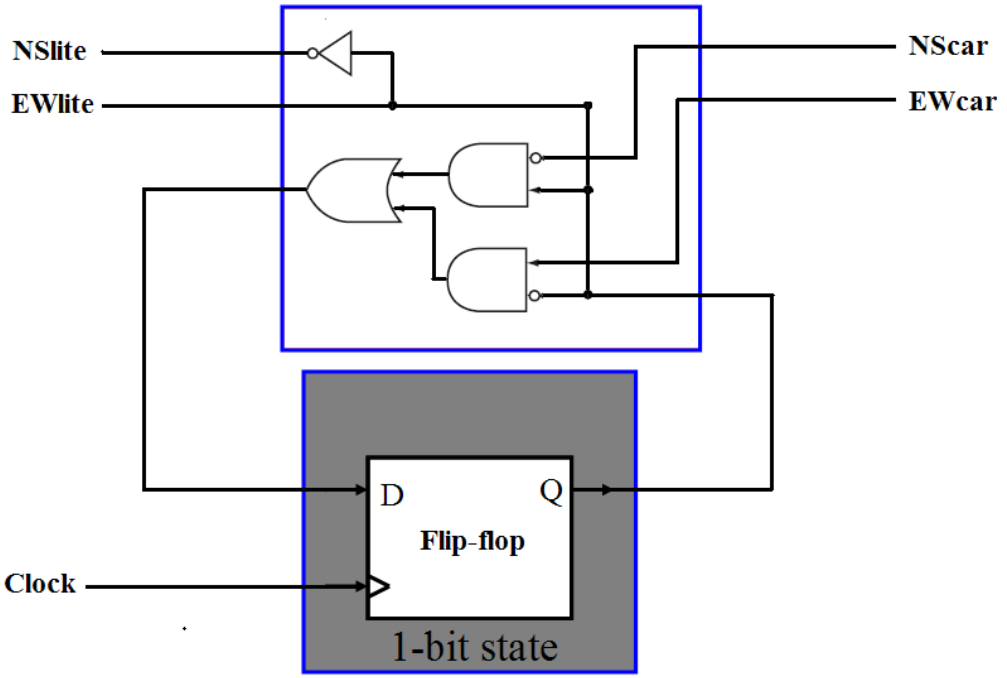
为什么需要状态机

- 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

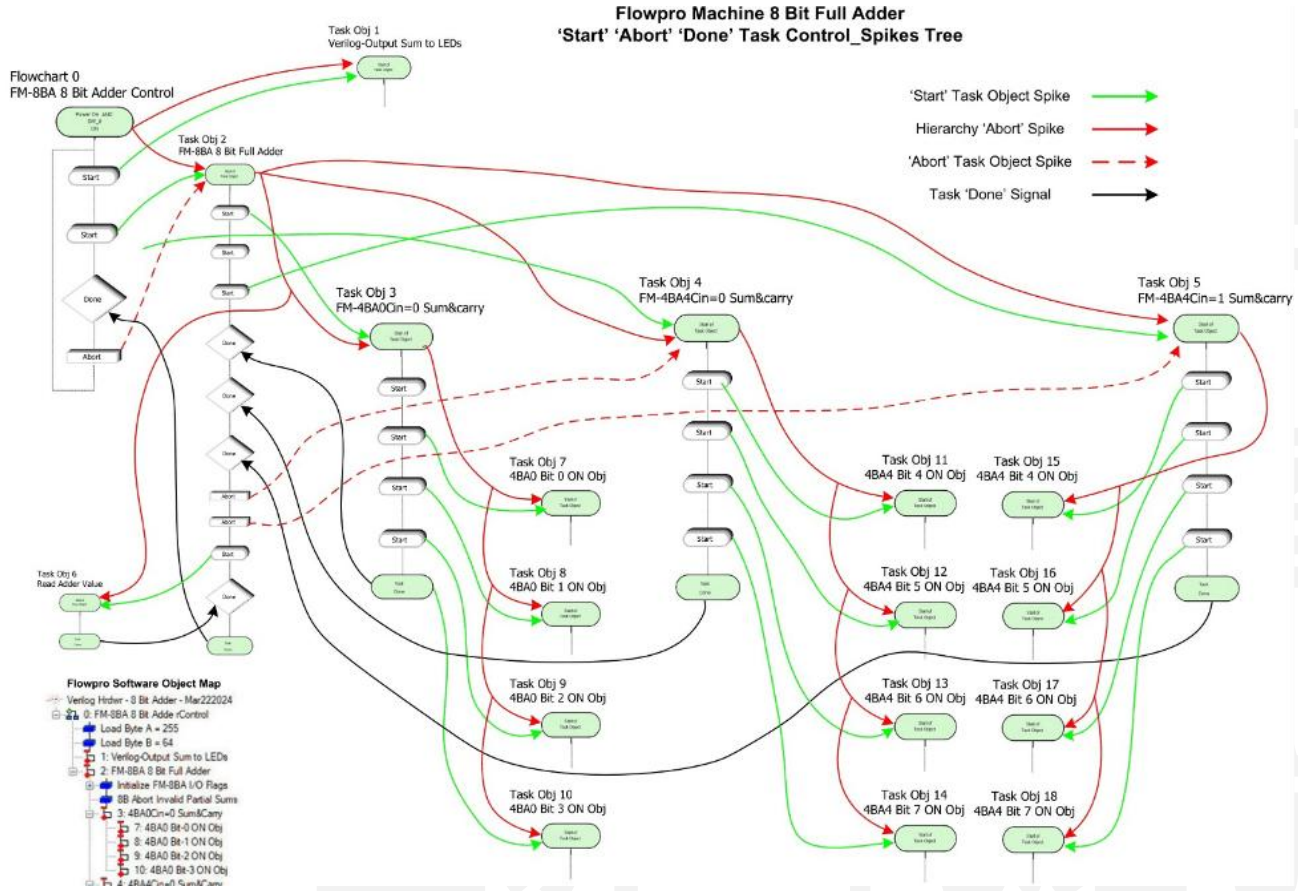
Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

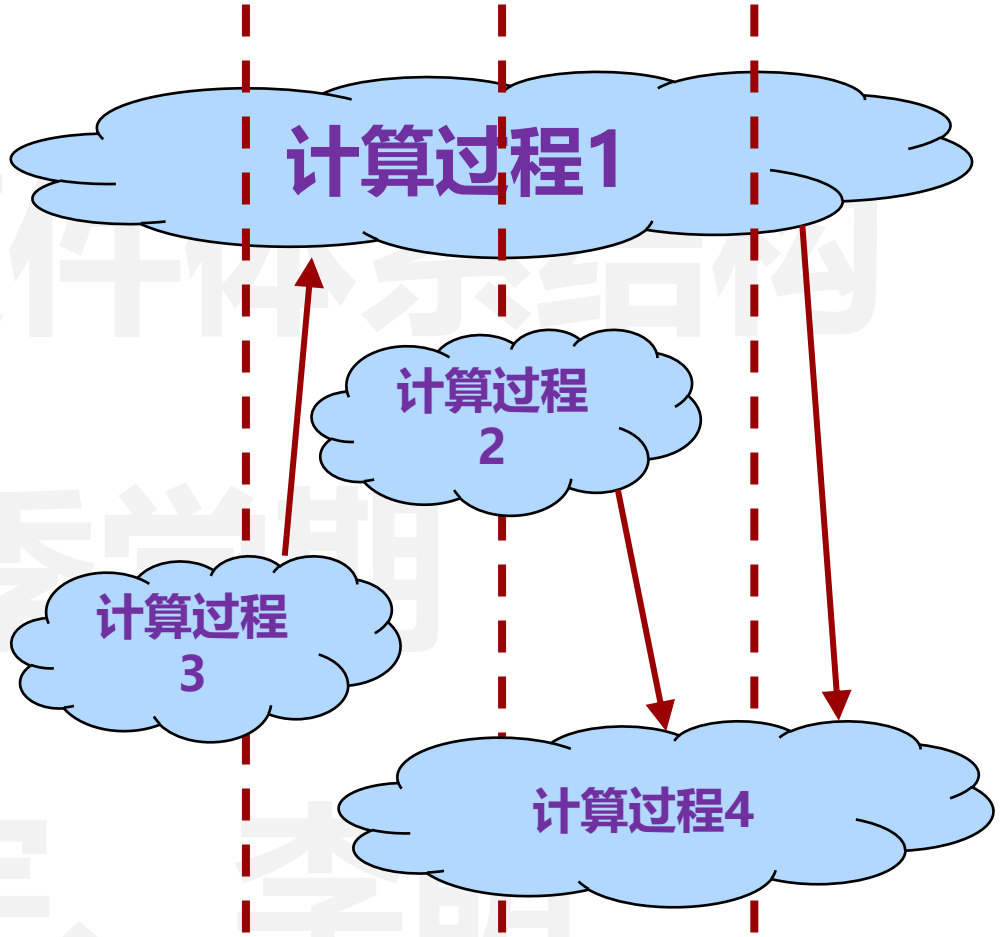
Step 4 – 画出实际电路图

状态将在每一个时钟上升沿更新

电路时序的基本概念

- 电路为什么需要一个时钟?



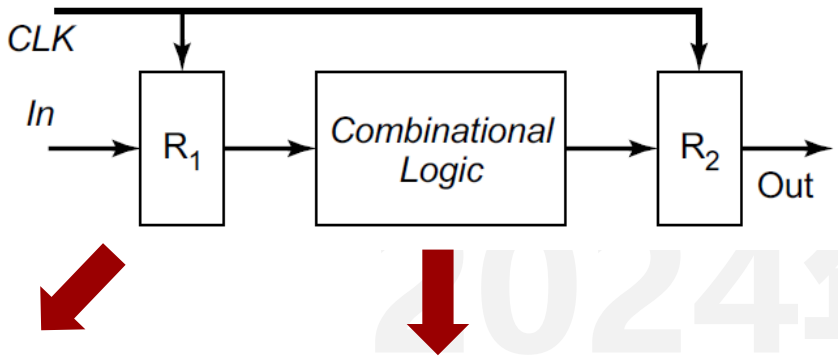


无时钟：非常难以控制每一个信号的有效时间

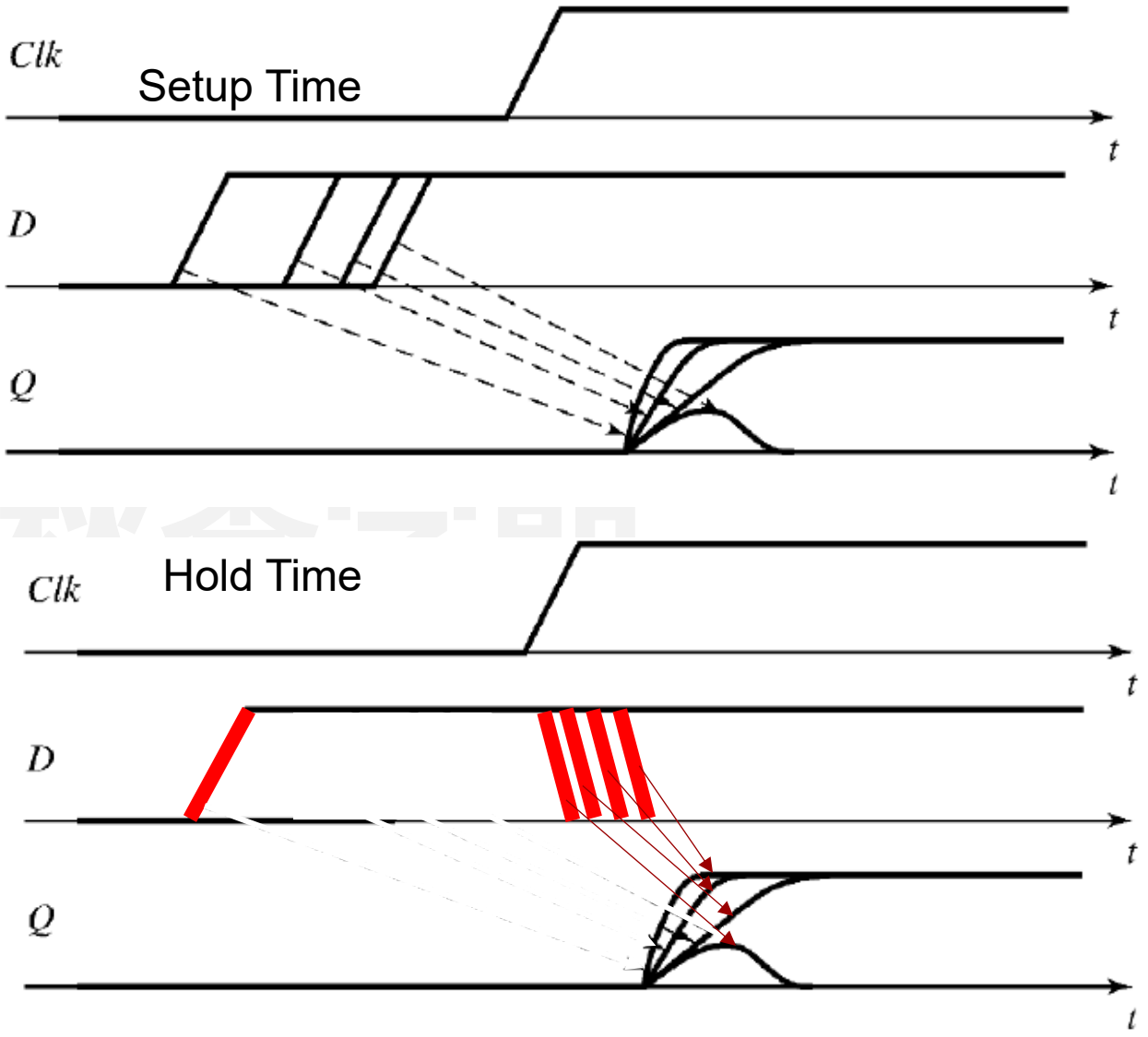
引入时钟：每隔一段计算将结果同步一次

电路时序的基本概念

- 同步时序（Synchronous Timing）

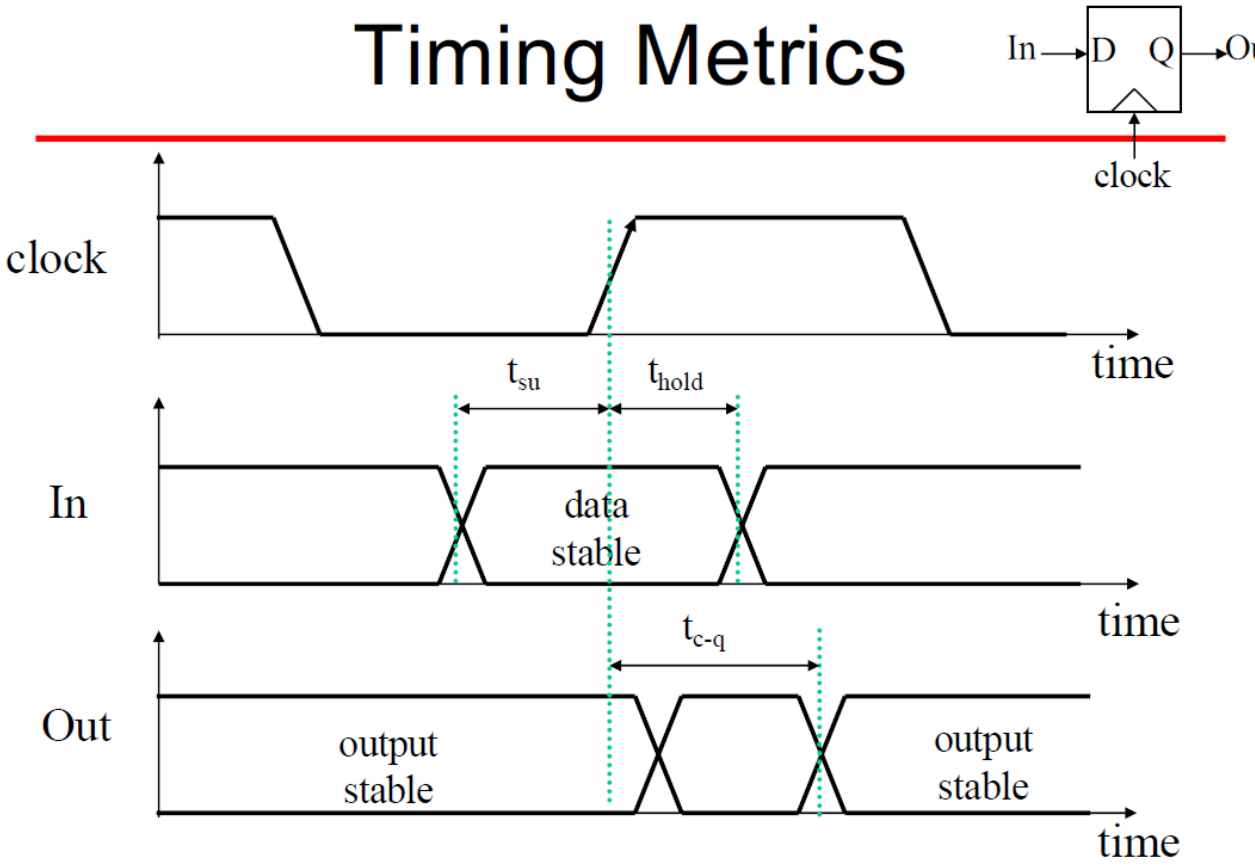


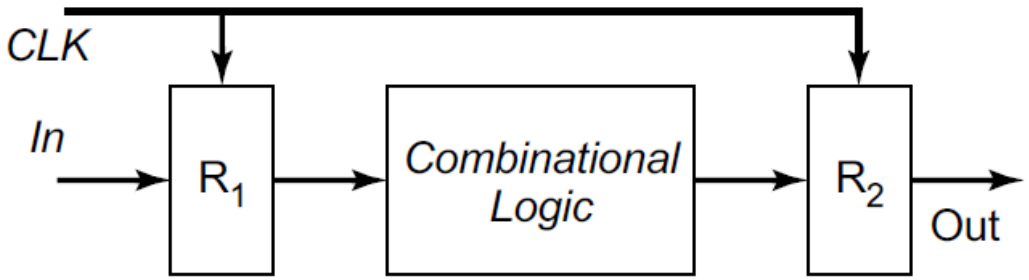
触发器　组合逻辑（各种逻辑门电路）
(Register)



电路时序的基本概念

- 同步时序（Synchronous Timing）





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

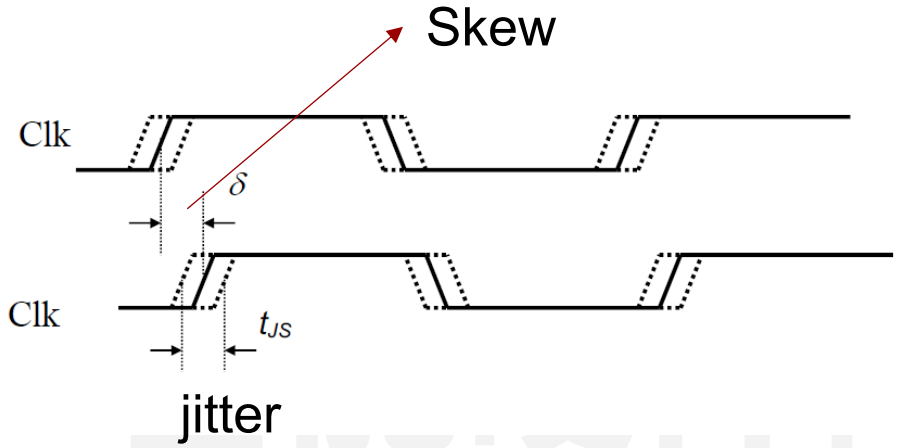
min

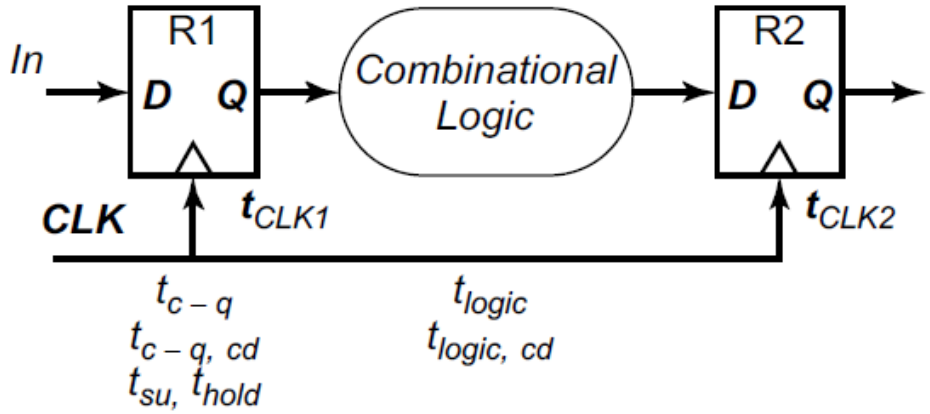
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

max

电路时序的基本概念

- 时钟的不稳定性

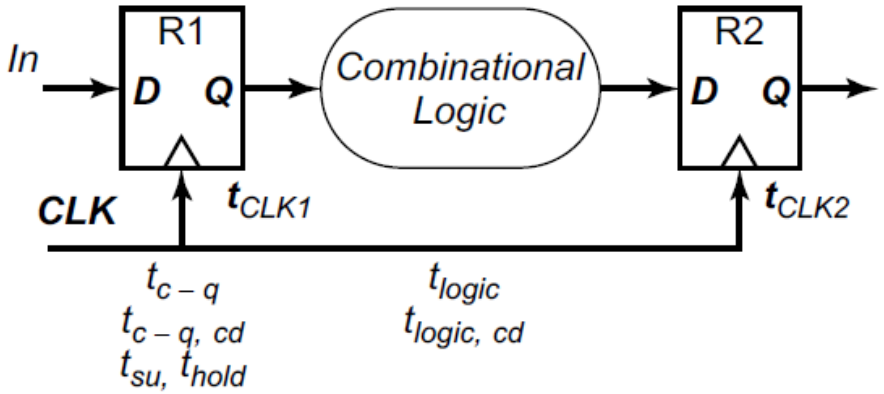




Minimum cycle time:

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

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

为什么需要指令集?

- 指令集可以看做链接软件和硬件的一个协议

ISA (instruction set architecture)

A well-defined hardware/software interface

一个定义完善的软/硬件接口

软件与硬件之间的“协议”

- 对硬件支持操作、模式和存储位置的功能定义
- 对如何调用获取硬件资源的精确描述

以下内容不通过ISA定义

- 具体如何实现操作
- 在不同场景下，不同操作速度的快慢
- 不同操作的功耗

为什么需要指令集?

北京大学 PEKING UNIVERSITY

- **指令集可以看做链接软件和硬件的一个协议**

- 编程者可见的变量
  - 编程计数器，通用计数器，存储器，控制寄存器
- 编程者可见的行为（状态转换）
  - 要做什么，什么时候做

Example “register-transfer-level” description of an instruction

```
if imem[pc]==“add rd, rs, rt”
then
        pc  ⇐ pc+1
        gpr[rd]=gpr[rs]+grp[rt]
```

- A binary encoding

ISAs last 25+ years (because of SW cost)…

…be careful what goes in

指令集的分类

- **RSIC和CISC两种指令集**

- “Iron” law:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC (Complex Instruction Set Computing) **例如X86等商用指令集**
  - 用复杂的指令改善了“instruction/program”
  - 对软件编程者友好，代码量小
- RISC (Reduced Instruction Set Computing) **例如MIPS/ARM/RISC-V**
  - 通过许多单周期指令来改善“cycles/instruction”
  - 增加了“instruction/program”，但代价不大
    - 编译器对此帮助很大
  - 有时会改善时钟周期长度
    - 精简指令允许了更激进的代码与硬件实现

指令集设计思路

- 兼顾软件可编程性、硬件可实现性和兼容性
- **Programmability**
  - 可以高效且容易的表达程序
- **Implementability**
  - 能够设计出高性能的硬件实现
    - 低功耗设计
    - 高可靠性设计
    - 低开销设计
- **Compatibility**
  - 在编程语言与程序更新迭代后，能够保持可编程性与硬件可实现性
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, Pentium-II, Pentium-III, Pentium4, …
  - MIPS、RISC-V、ARM…

软件编译过程

- 软件代码通过编译器和指令集，编译成硬件可直接运行的汇编代码

- Demo of assembler
  - $ g++ -Og -c -S file1.cpp
- Demo of hexdump
  - $ g++ -Og -c file1.cpp
  - $ hexdump -C file1.o | more
- Demo of objdump/disassembler
  - $ g++ -Og -c file1.cpp
  - $ objdump -d file1.o

```
void abs(int x, int* res)
{
  if(x < 0)
    *res = -x;
  else
    *res = x;
}
```

Original Code

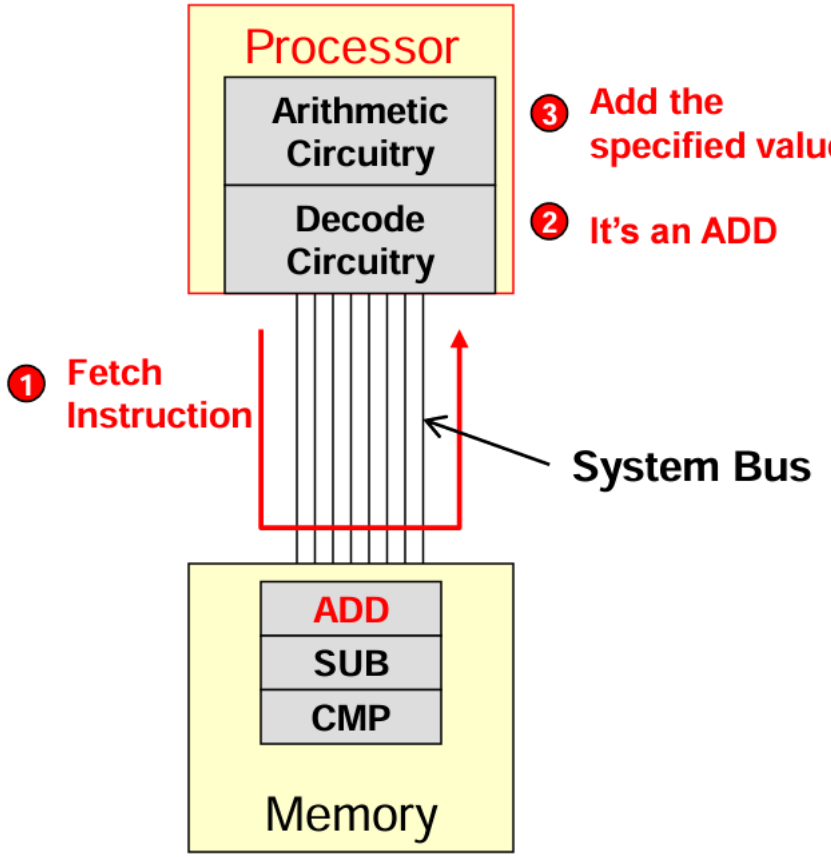
```
Disassembly of section .text:

0000000000000000 <_Z3absiPi>:
   0:   85 ff    test   %edi,%edi
   2:   79 05    jns    9 <_Z3absiPi+0x9>
   4:   f7 df    neg    %edi
   6:   89 3e    mov    %edi,(%rsi)
   8:   c3       retq
   9:   89 3e    mov    %edi,(%rsi)
   b:   c3       retq
```

Compiler Output
(Machine code & Assembly)
Notice how each instruction is turned into binary (shown in hex)

传统冯诺依曼架构的指令集

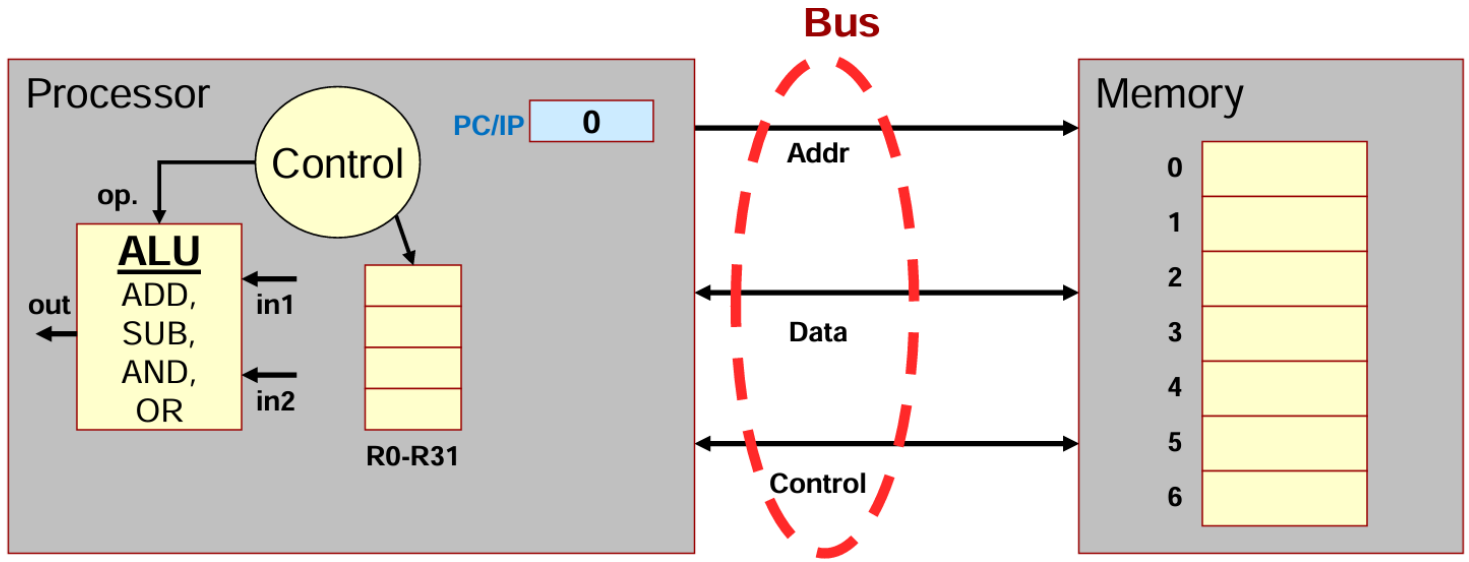
- 需要3类指令：读取、写回和运算

- 来回执行以下同样三种类型的指令
  - Fetch：从存储器中取出指令
  - Decode：解码这个指令
    - 此指令是ADD，SUB或是其他?
  - Execute：执行指令
    - 执行特定操作
- 每个指令运行的过程被称为指令周期



传统冯诺依曼架构的指令集

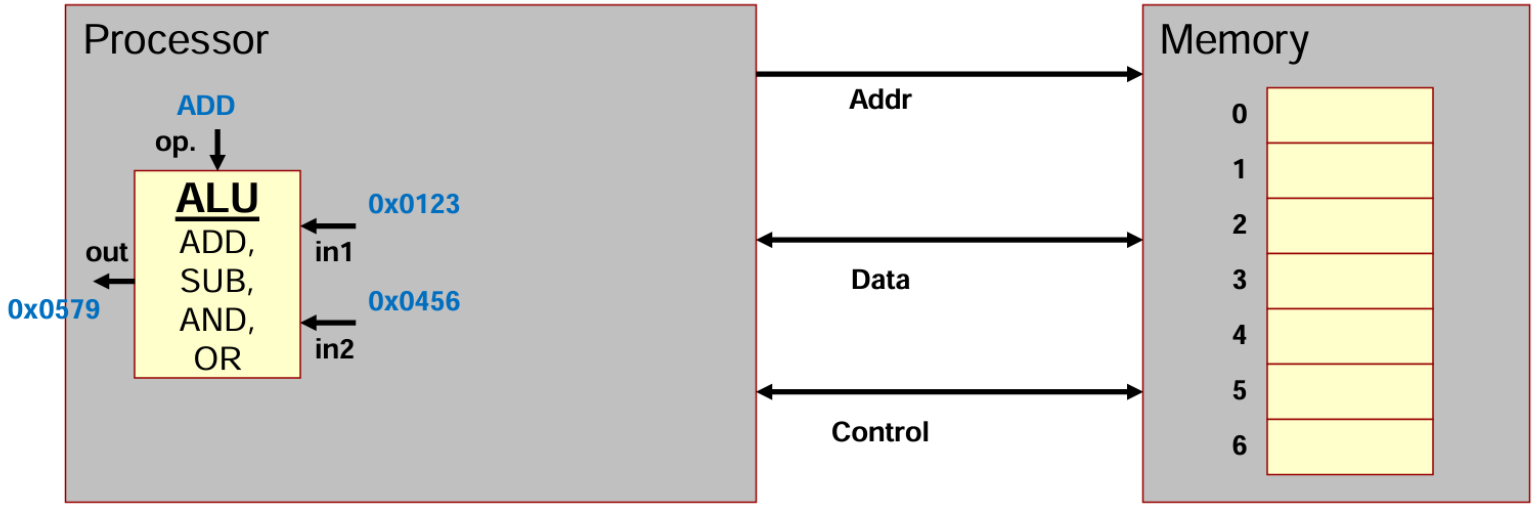
- 需要3类指令：读取、写回和运算
  - 处理器中3种主要的组成
    - ALU（算术逻辑单元）
    - 寄存器
    - 控制电路
  - 通过地址、数据和控制总线（bus）与存储器和I/O连接



传统存算分离的指令集架构 – 核心部件1：ALU

- ALU是指令集架构的核心部件，负责完成所有实际的计算功能

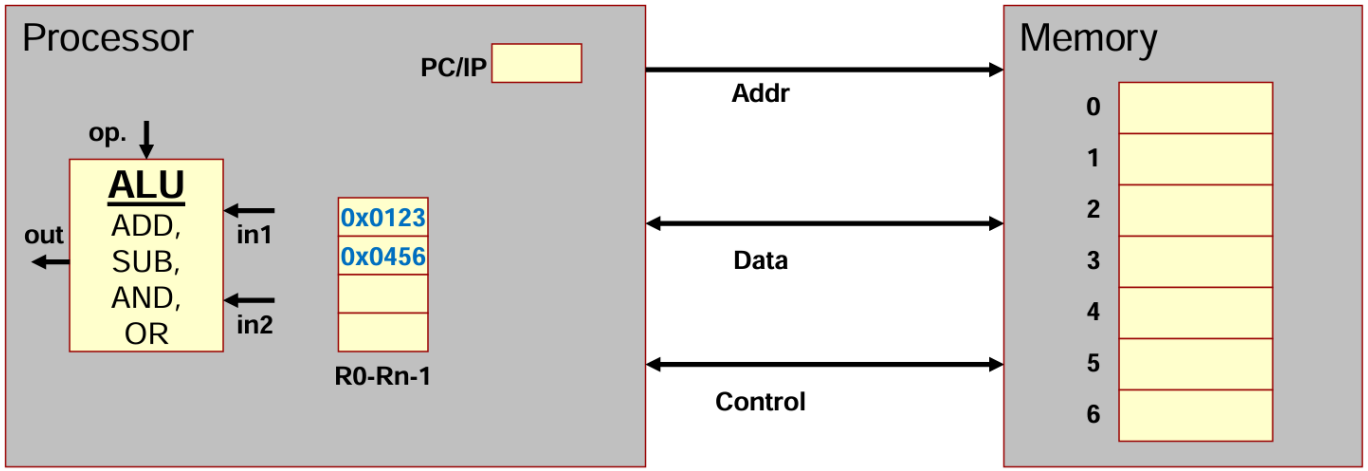
  - 执行加减、逻辑运算等(AND,OR,etc.)算术操作的数字电路



传统存算分离的指令集架构 – 核心部件1：Register

- Register负责将ALU运算结果暂存在靠近ALU的地方

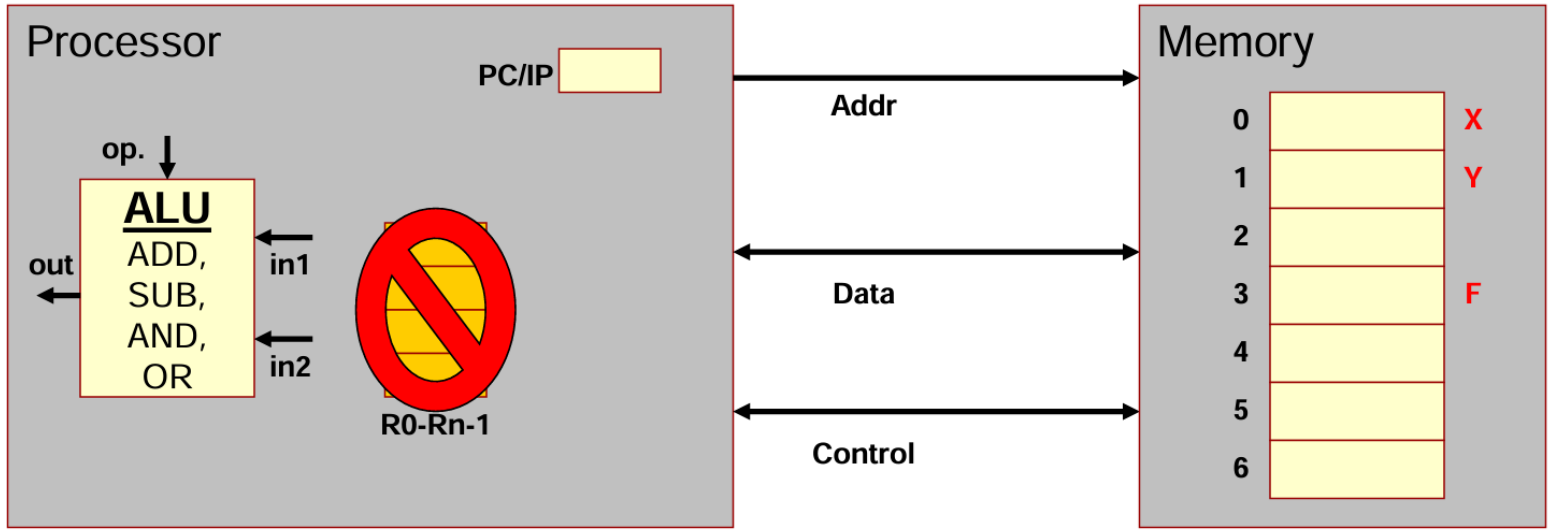
  - 访问存储器的时间通常比处理器要慢
  - 寄存器在处理器内部提供了快速的暂时存储位置



- 软件指令可以调用寄存器用于编程与编译
- 编程/编译使用这些寄存器作为输入（源位置）和输出（目标位置）

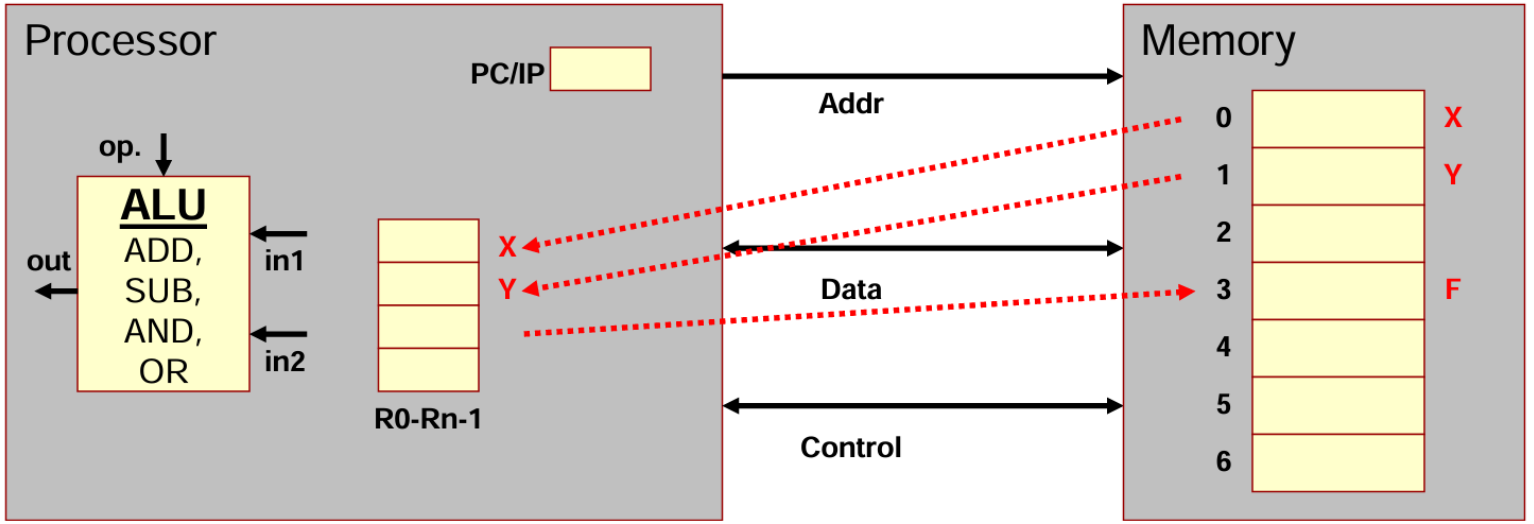
传统存算分离的指令集架构 – 核心部件1：Register

- Register的存在大幅减少了长延时的Memory访问
  - 无寄存器时计算 F = (X+Y)-(X*Y):
    - 需要ADD，MUL，SUB这些指令
    - 无寄存器
      - ADD：从存储器加载X和Y，存储计算结果到存储器
      - MUL：再次从存储器加载X和Y，存储计算结果到存储器
      - SUB：加载ADD和MUL的计算结果，存储计算结果到存储器
      - **共9次访存**



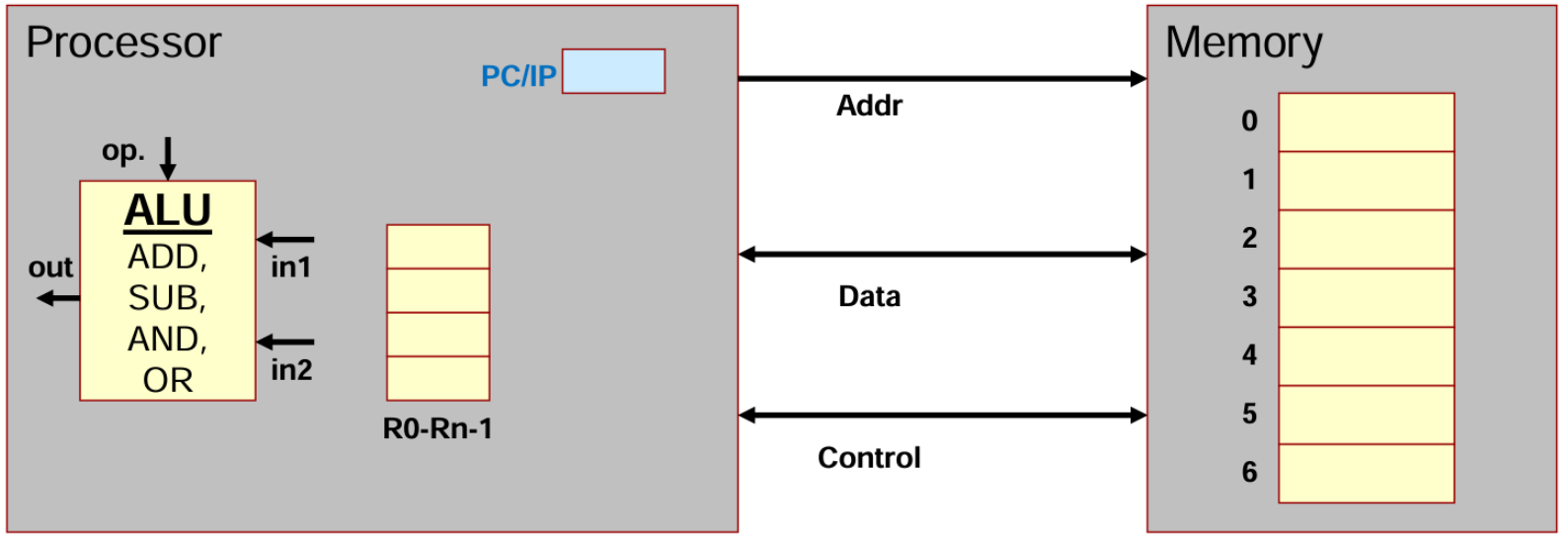
传统存算分离的指令集架构 – 核心部件1：Register

- Register的存在大幅减少了长延时的Memory访问
  - 使用寄存器时计算 F = (X+Y)-(X*Y):
    - 从存储器加载X和Y到寄存器R0，R1
    - ADD：计算R0+R1并存储到R2
    - MUL：计算R0*R1并存储到R3
    - SUB：计算R2-R3并存储到R4
    - 存储R4到存储器
    - 3次访存



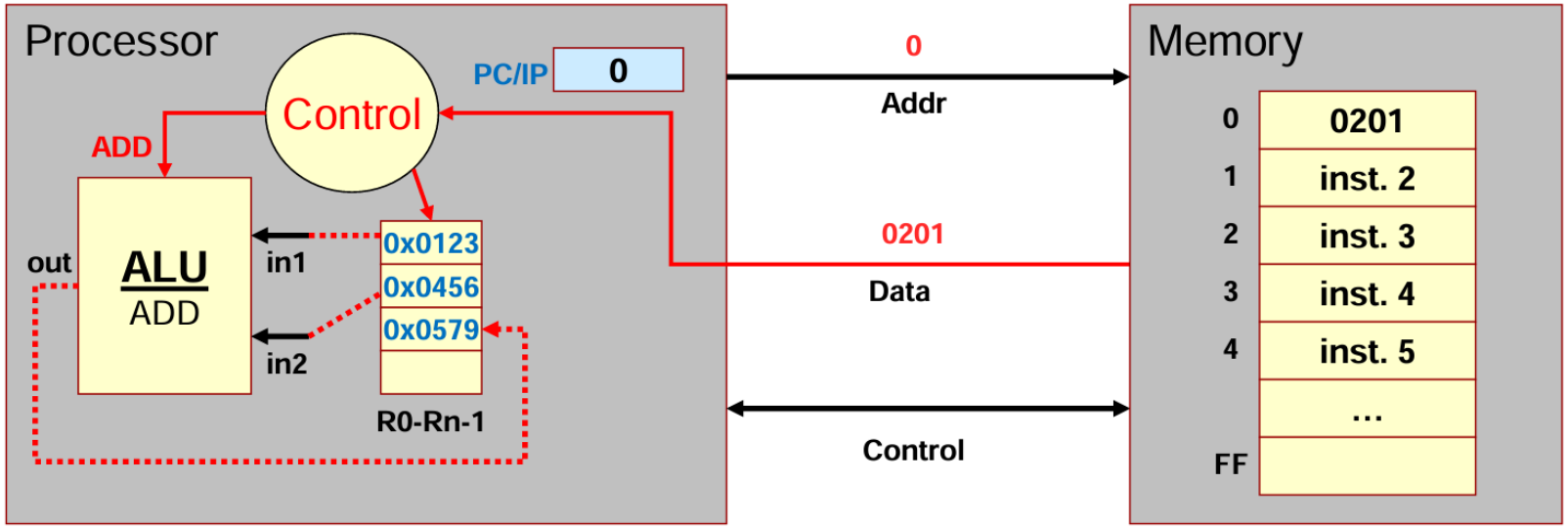
传统存算分离的指令集架构 – 核心部件1：Register

- Register还包括用于记录程序状态与指令状态的PC/IP
  - 要使处理器正确运行需要一些状态信息
  - 如程序计数器/指令指针（PC/IP）寄存器
    - 上文讲到处理器在译码与执行指令之前要从存储器获取指令
    - PC/IP寄存器保留下一个要获取指令的地址



简单指令集架构的操作流程

- 指令从Memory中读取，ALU进行运算（可内含加、减、乘、除、逻辑、复杂计算单元等）
  - 假设0x0201是R2=R0+R1这个ADD指令的机器码
  - 控制逻辑将会是
    - 选取源寄存器（R0 和 R1）
    - 告诉ALU做加法
    - 选取目标寄存器（R2)



指令集数据的位置

- 数据可以存储在register、主存memory或指令内部
  - 源操作数存储在以下三个位置内
    - 寄存器值(eg. %rax)
    - 主存中的值（eg. 0x0200e8)
    - 在指令内部存储的常量（又叫“立即数immediate”）[eg. ADDI $1,D0]
      - $表示是常量或者是立即数
  - 目标操作数存储在
    - 寄存器
    - 存储器（由其地址指定）