计 算 机 组 成 与 结 构

彭柯鑫
pkx@cdut.edu.cn

第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

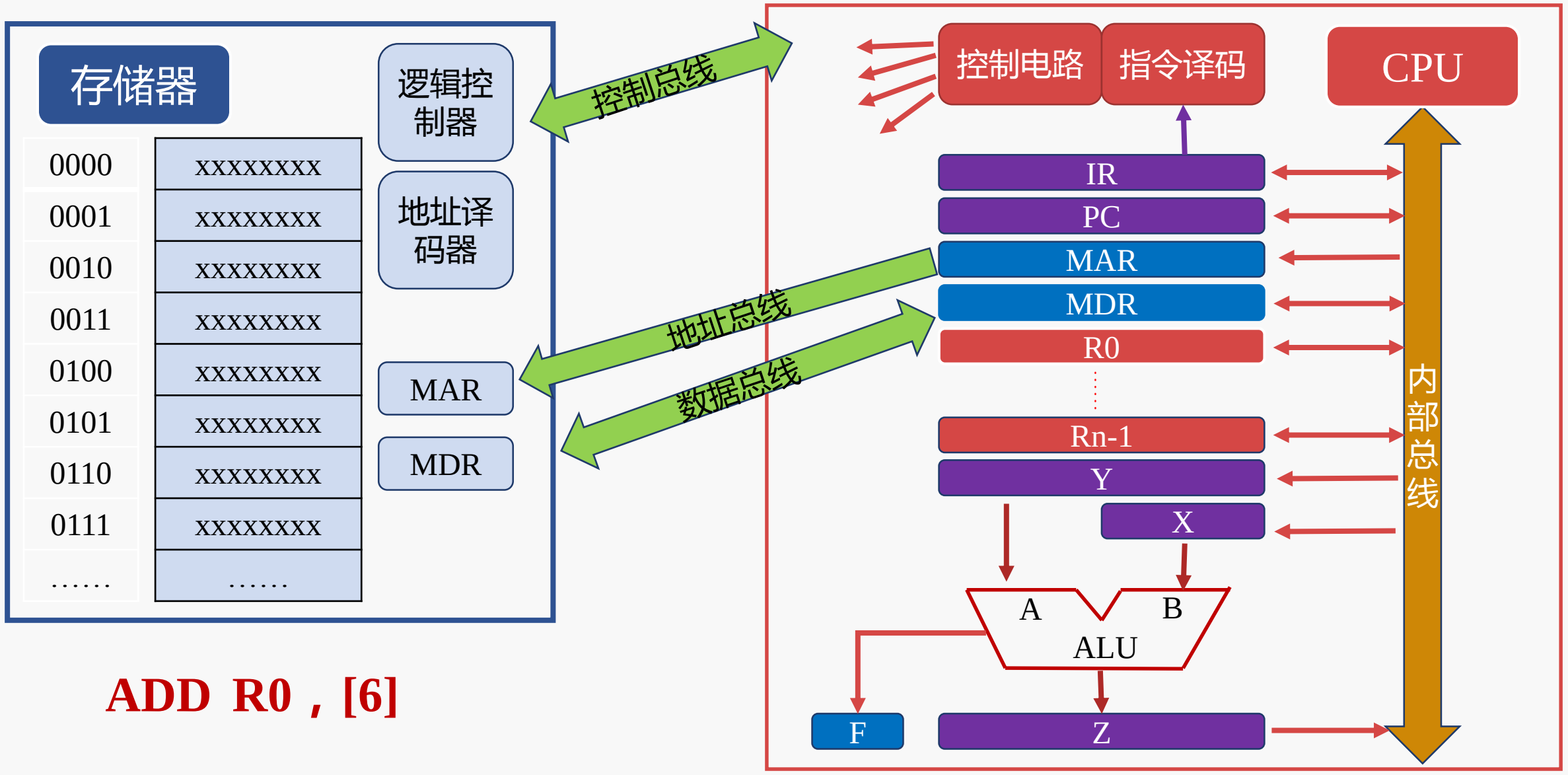
3.寄存器的基本原理

4.逻辑运算的实现

5.加法和减法的实现

6.加法器的优化

计算机结构的简化模型（模型机）



ADD R0，[6]

加法指令的编码示例（1）

add $8，$9，$10 #$8=$9+$10

◦ 查指令编码表得到：

opcode = 0 ，funct = 20hex

shamt = 0（非移位指令）

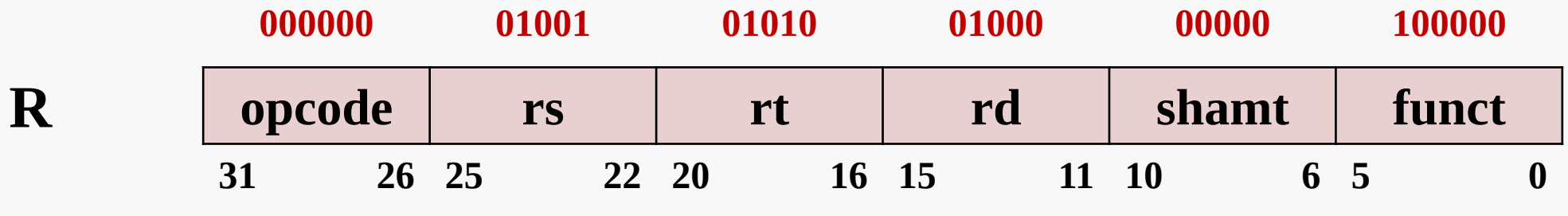
◦ 根据指令操作数得到：

rd = 8（目的操作数），rs = 9（第一个源操作数）

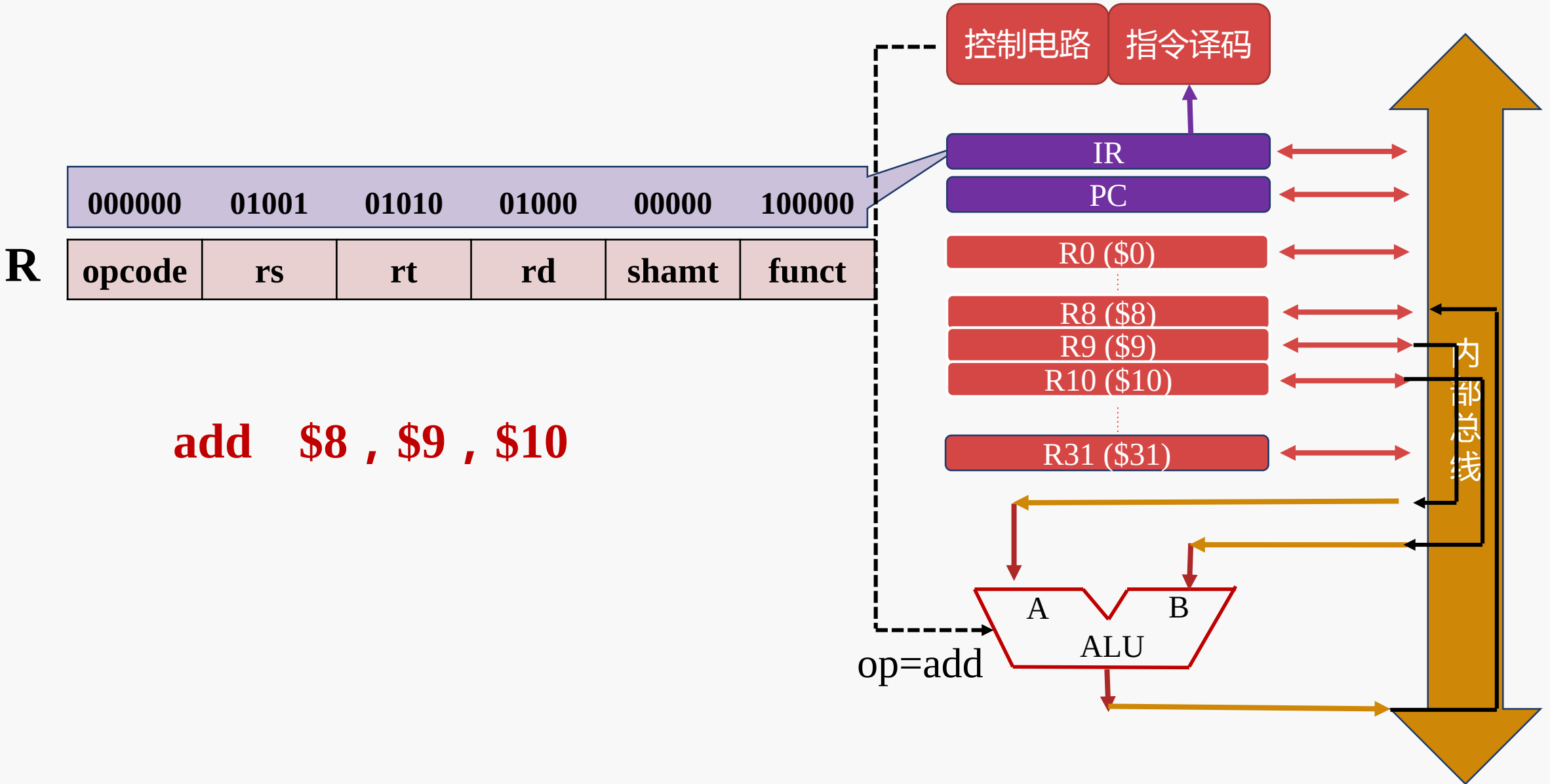
rt = 10（第二个源操作数）

C语言程序

```
int f，g，h;
...
//f→$8,g→$9,h→$10
f = g + h;
```



加法运算示例


控制电路
指令译码
IR
PC
R0 ($0)
R8 ($8)
R9 ($9)
R10 ($10)
R31 ($31)
内部总线
000000
01001
01010
01000
00000
100000
R
opcode
rs
rt
rd
shamt
funct
add $8，$9，$10
A
B
ALU
op=add

算术运算指令（MIPS Core Instruction Set）

R型

- add rd，rs，rt # R[rd]=R[rs]+R[rt] (1)
- addu rd，rs，rt # R[rd]=R[rs]+R[rt]
- sub rd，rs，rt # R[rd]=R[rs]-R[rt] (1)
- subu rd，rs，rt # R[rd]=R[rs]-R[rt]

(1) May cause overflow exception

加法指令的编码示例（2）

addi　$21，$22，-50　　#　$21=$22+(-50)

◦ 查指令编码表得到：

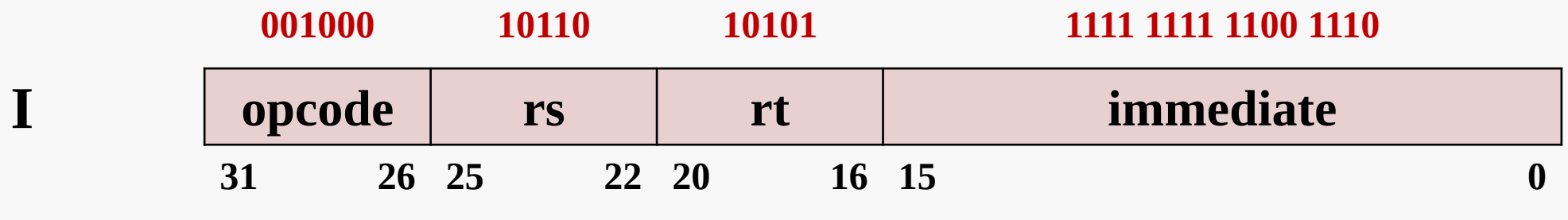
opcode = 8

◦ 分析指令得到：

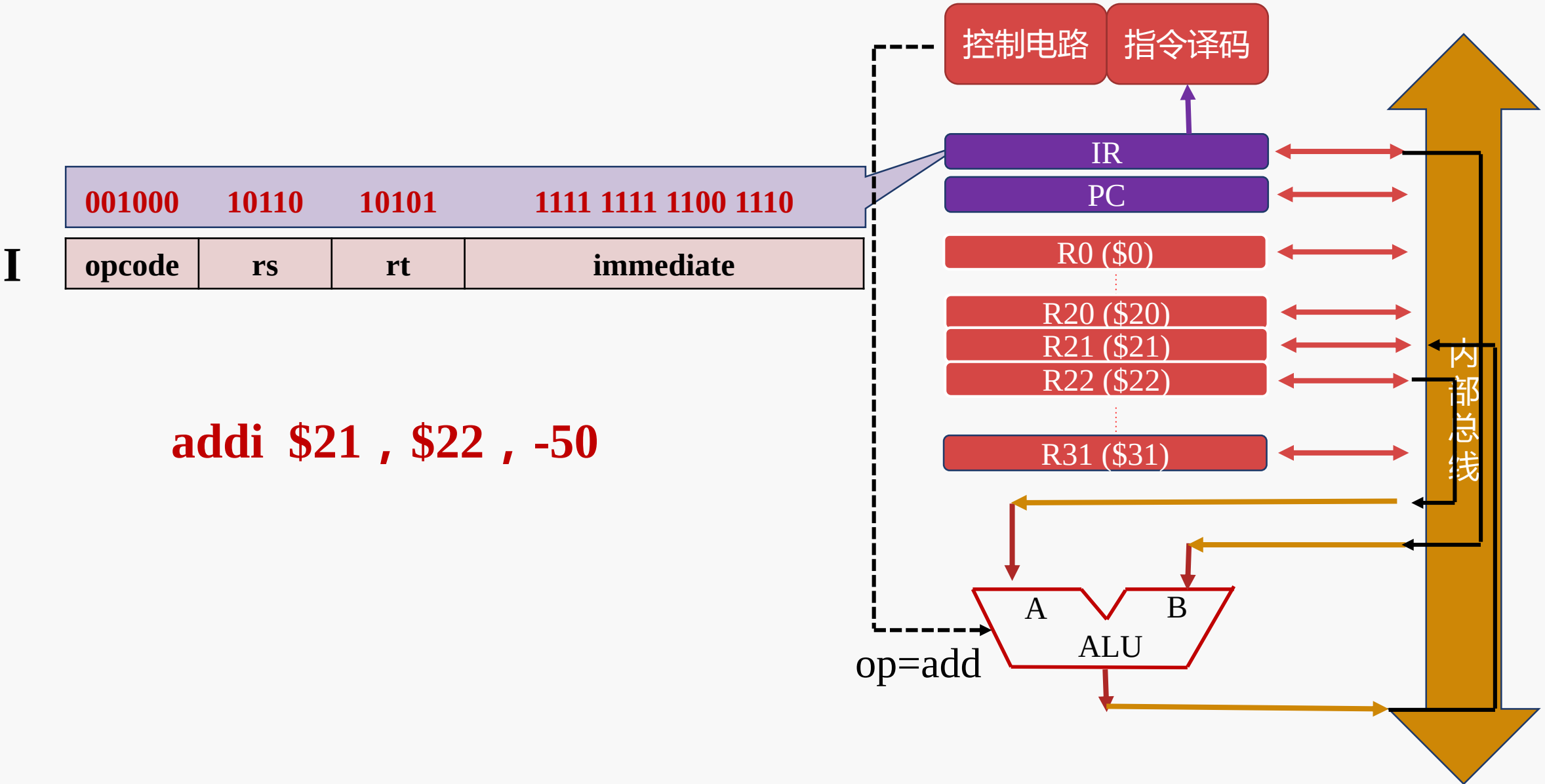
rs = 22（源操作数寄存器编号）

rt = 21（目的操作数寄存器编号）

immediate = -50（立即数）



加法运算示例


I
001000 10110 10101 1111 1111 1100 1110
opcode rs rt immediate
addi $21，$22，-50
控制电路
指令译码
IR
PC
R0 ($0)
R20 ($20)
R21 ($21)
R22 ($22)
R31 ($31)
内部总线
A
B
ALU
op=add

算术运算指令（MIPS Core Instruction Set）

R型

- add rd，rs，rt # R[rd]=R[rs]+R[rt] (1)
- addu rd，rs，rt # R[rd]=R[rs]+R[rt]
- sub rd，rs，rt # R[rd]=R[rs]-R[rt] (1)
- subu rd，rs，rt # R[rd]=R[rs]-R[rt]

I型

- addi rt，rs，imm # R[rt]=R[rs]+SignExtImm (1,2)
- addiu rt，rs，imm # R[rt]=R[rs]+SignExtImm (2)

(1) May cause overflow exception
(2) SignExtImm={ 16{imm[15]}, imm }

逻辑运算指令（MIPS Core Instruction Set）

R型

- and rd，rs，rt　　# R[rd]=R[rs]&R[rt]
- or rd，rs，rt　　# R[rd]=R[rs]|R[rt]
- nor rd，rs，rt　　# R[rd]=~(R[rs]|R[rt])

I型

- andi rt，rs，imm　　# R[rt]=R[rs]&ZeroExtImm　　(3)
- ori rt，rs，imm　　# R[rt]=R[rs]|ZeroExtImm　　(3)

(3) ZeroExtImm={ 16{1'b0}, imm }

逻辑“与”指令的编码示例

R型

- and rd，rs，rt　　# R[rd]=R[rs]&R[rt]
- or rd，rs，rt　　# R[rd]=R[rs]|R[rt]
- nor rd，rs，rt　　# R[rd]=~(R[rs]|R[rt])

I型

- andi rt，rs，imm　　# R[rt]=R[rs]&ZeroExtImm　　(3)
- ori rt，rs，imm　　# R[rt]=R[rs]|ZeroExtImm　　(3)

(3) ZeroExtImm={ 16{1’b0}, imm }

逻辑“与”指令的编码示例

and $8，$9，$10　　#$8=$9&$10

◦ 查指令编码表得到：

opcode = 0 ，funct = 24hex

shamt = 0（非移位指令）

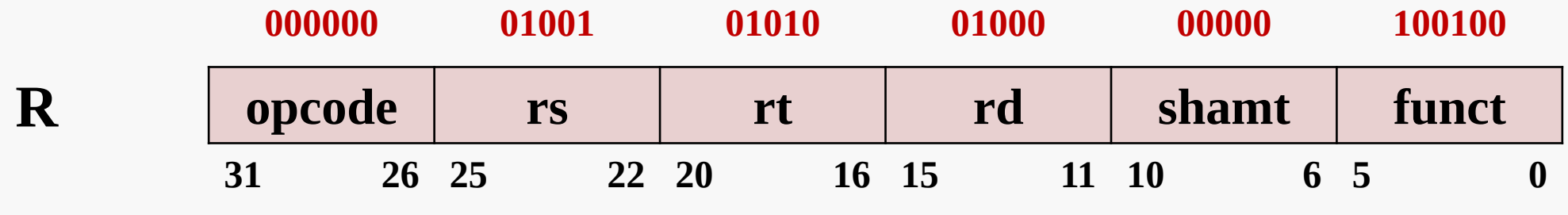
◦ 根据指令操作数得到：

rd = 8（目的操作数），rs = 9（第一个源操作数）

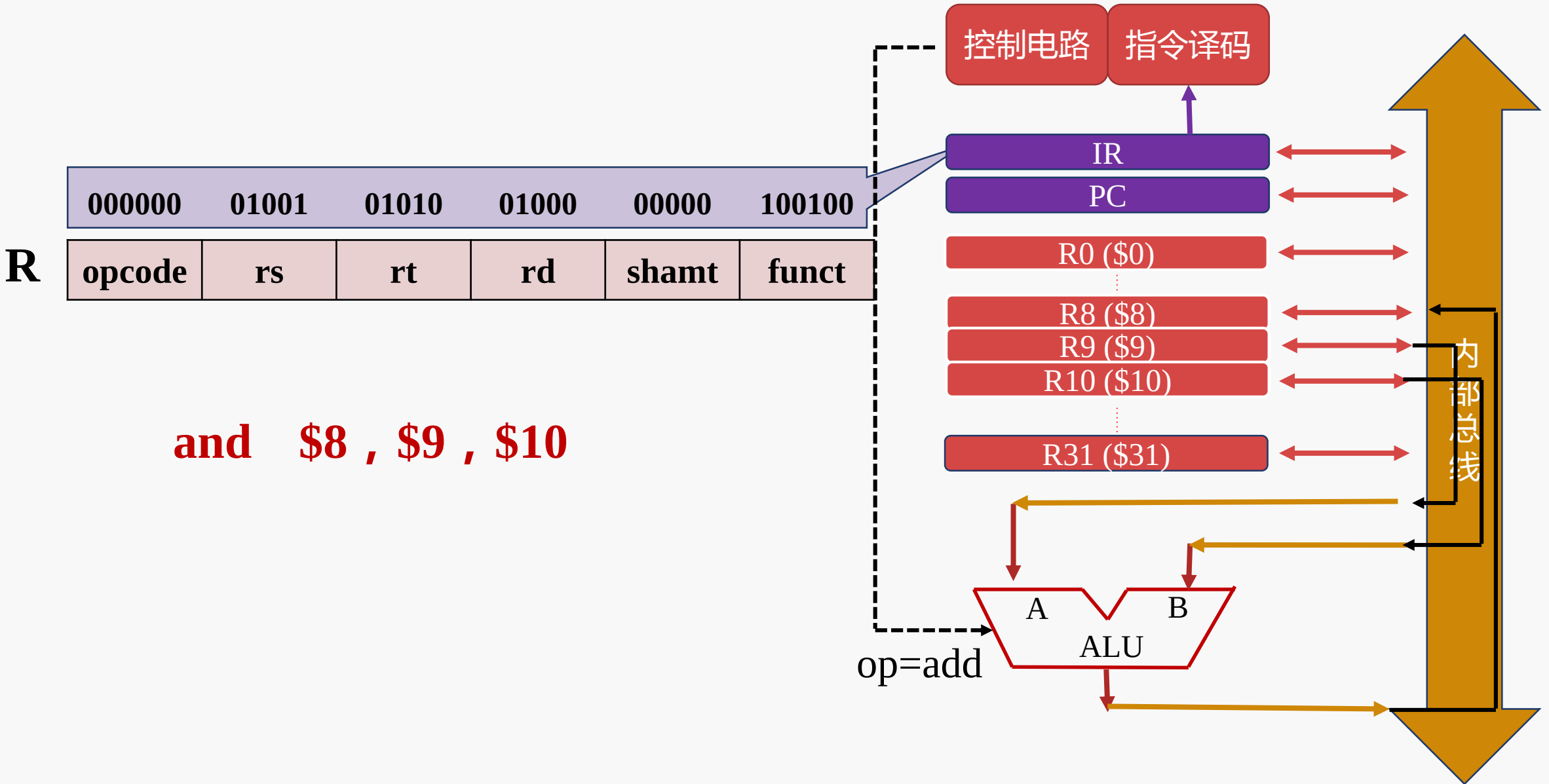
rt = 10（第二个源操作数）

C语言程序

```
int f，g，h;
...
//f→$8,g→$9,h→$10
f = g & h;
```



逻辑“与”运算示例


控制电路
指令译码
IR
PC
R0 ($0)
R8 ($8)
R9 ($9)
R10 ($10)
R31 ($31)
内部总线
000000
01001
01010
01000
00000
100100
R
opcode
rs
rt
rd
shamt
funct
and $8，$9，$10
A
B
ALU
op=add

算术逻辑运算的需求

算术运算

- 两个32-bit数的加法，结果为一个32-bit数
- 两个32-bit数的减法，结果为一个32-bit数
- 检查加减法的结果是否溢出

逻辑运算

- 两个32-bit数的“与”操作，结果为一个32-bit数
- 两个32-bit数的“或”操作，结果为一个32-bit数
- 两个32-bit数的“或非”操作，结果为一个32-bit数

第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

3.寄存器的基本原理

4.逻辑运算的实现

5.加法和减法的实现

6.加法器的优化

晶体管（transistor）

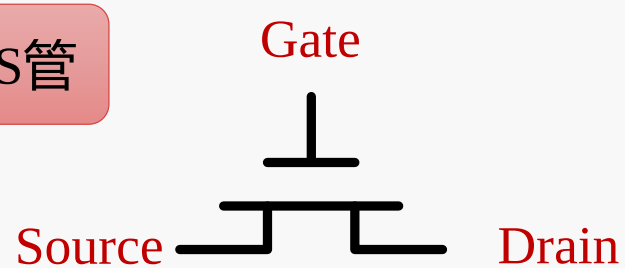
现代集成电路中通常使用MOS晶体管

- Metal-Oxide-Semiconductor：金属-氧化物-半导体

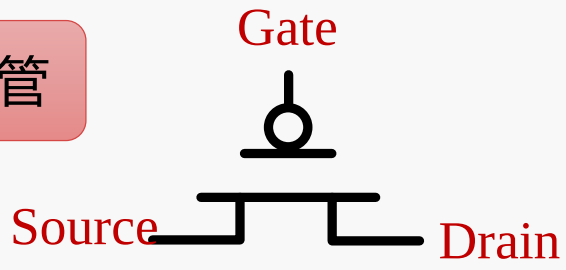
CMOS集成电路（Complementary MOS）

- 由PMOS和NMOS共同构成的互补型MOS集成电路

N型MOS管

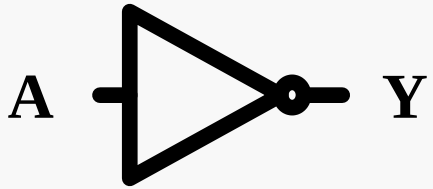


P型MOS管



非门（NOT gate）

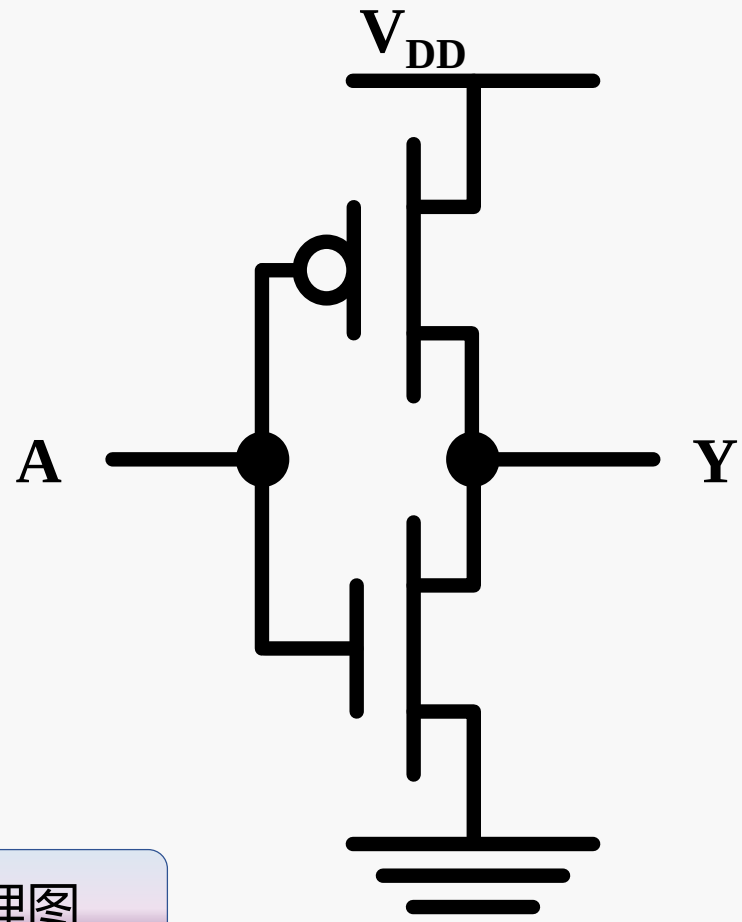
逻辑符号



真值表

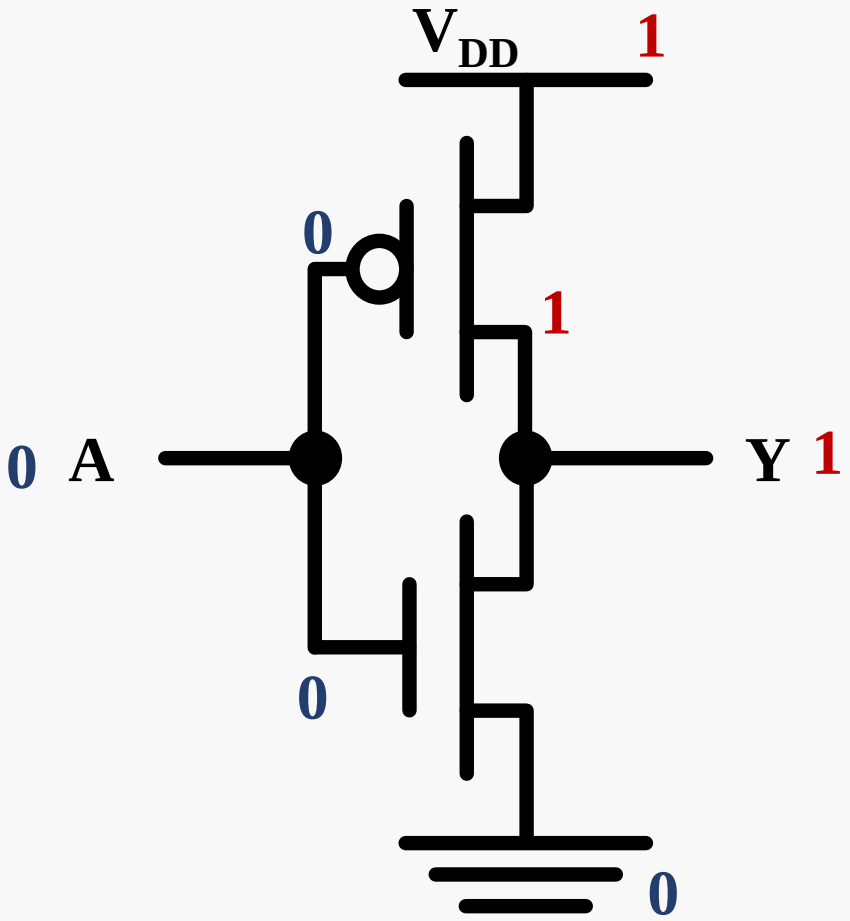
输入A	输出Y
0	1
1	0

逻辑函数表示 $Y = \overline{A}$
（Y=~A，Y=!A）

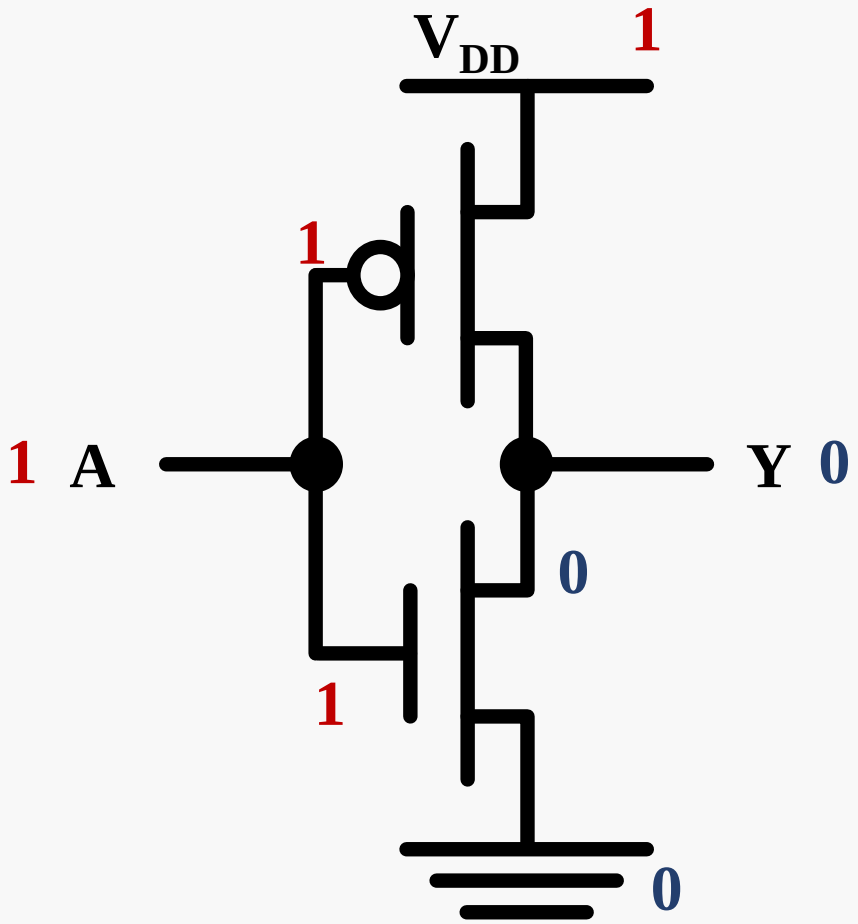


非门原理图

非门的工作过程示例



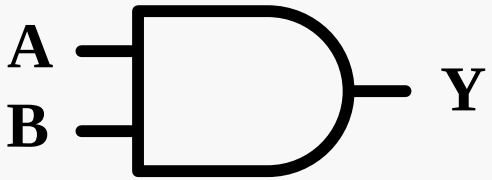
A=0 → Y=1



A=1 → Y=0

与门（AND gate）

逻辑符号

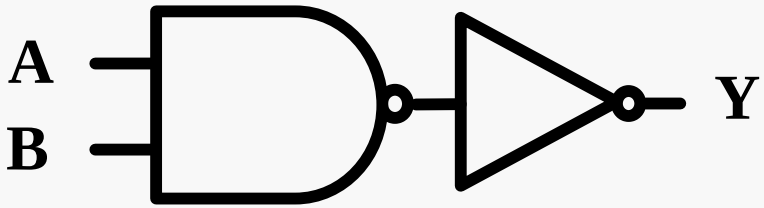


逻辑函数表示

$Y=A \cdot B$

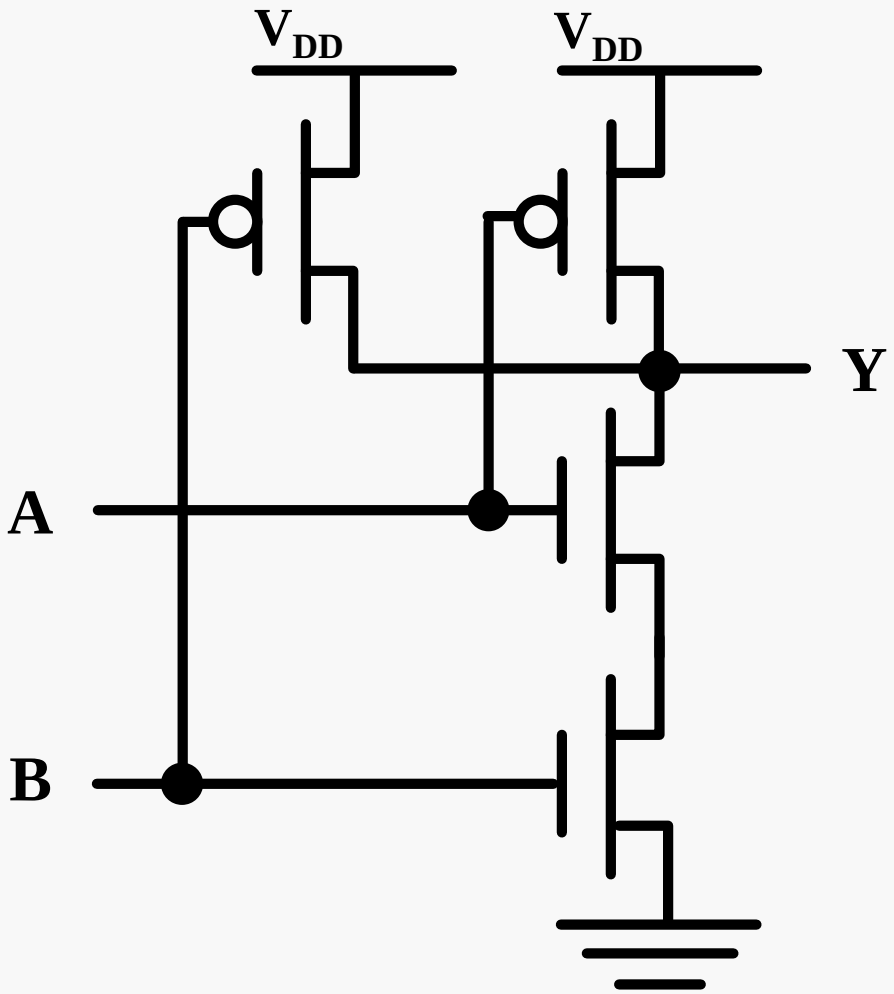
真值表

输入A	输入B	输出Y
0	0	0
0	1	0
1	0	0
1	1	1

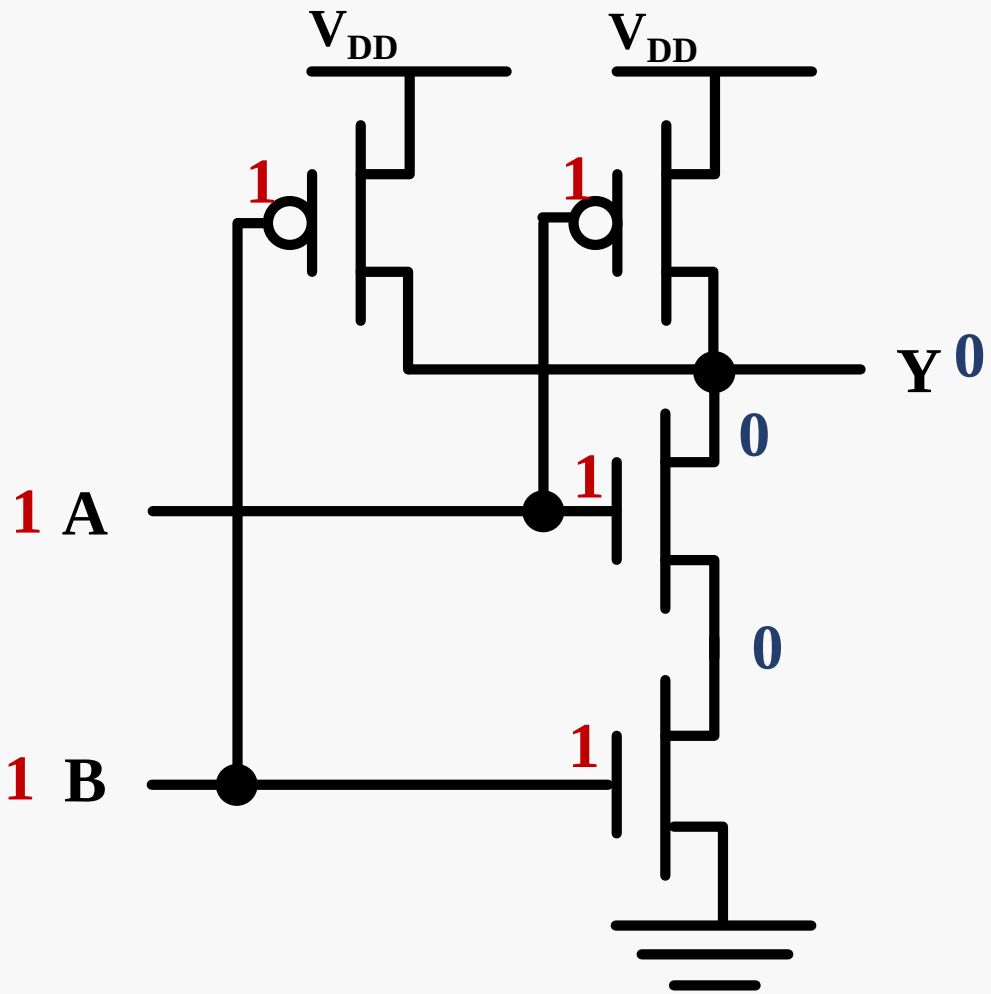


（实际用“与非门”和“非门”实现“与门”）

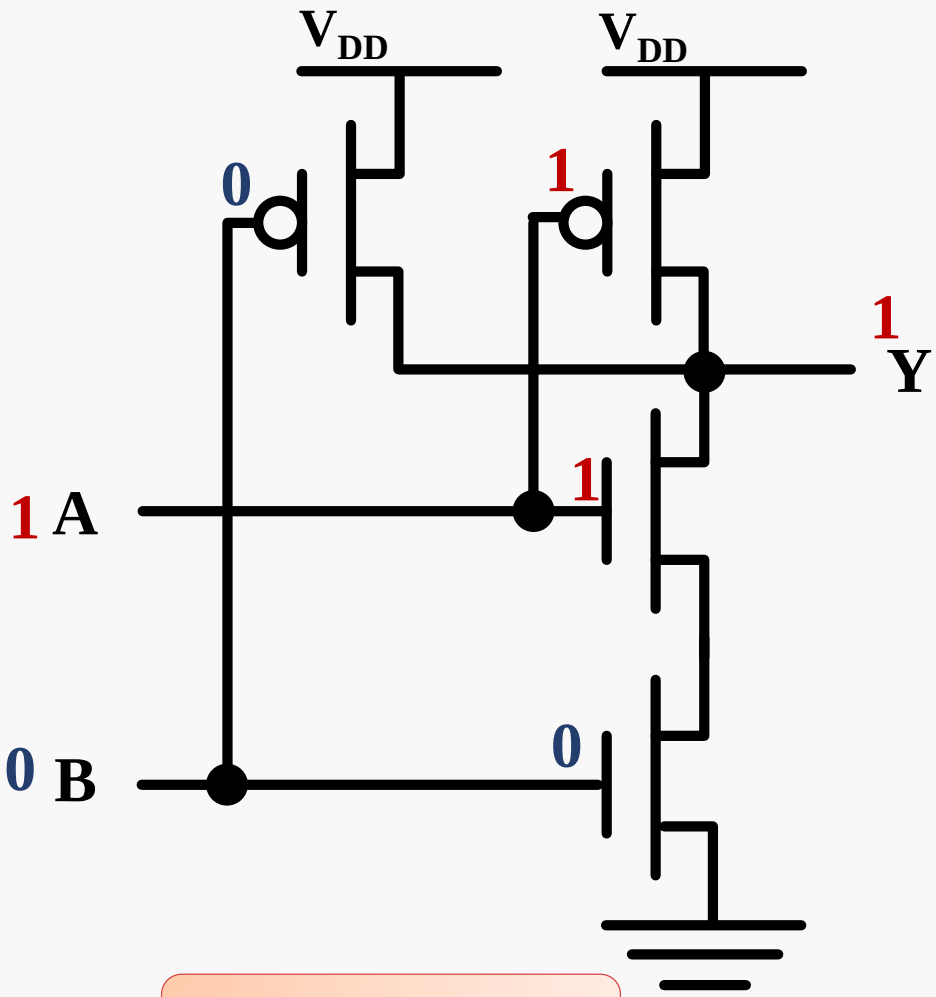
与非门原理图



与非门的工作过程示例



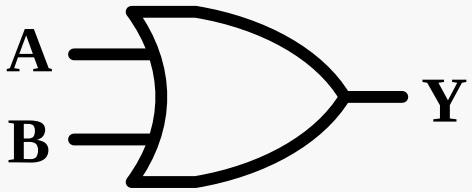
A=1, B=1 → Y=0



A=1, B=0 → Y=1

或门（OR gate）

逻辑符号



逻辑函数表示
Y=A+B

真值表

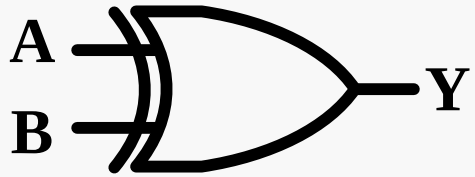
输入A	输入B	输出Y
0	0	0
0	1	1
1	0	1
1	1	1

异或门（Exclusive-OR gate, XOR gate）

异或运算：$A \oplus B=(\bar{A} \cdot B) + (A \cdot \bar{B})$

- 两个值不相同，则异或结果为真。反之，为假。

逻辑符号



逻辑函数表示

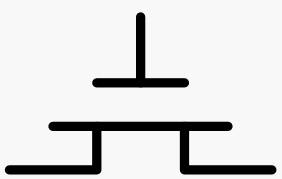
$Y=A \oplus B$

Y=A^B

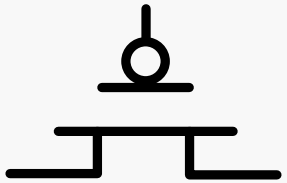
真值表

输入A	输入B	输出Y
0	0	0
0	1	1
1	0	1
1	1	0

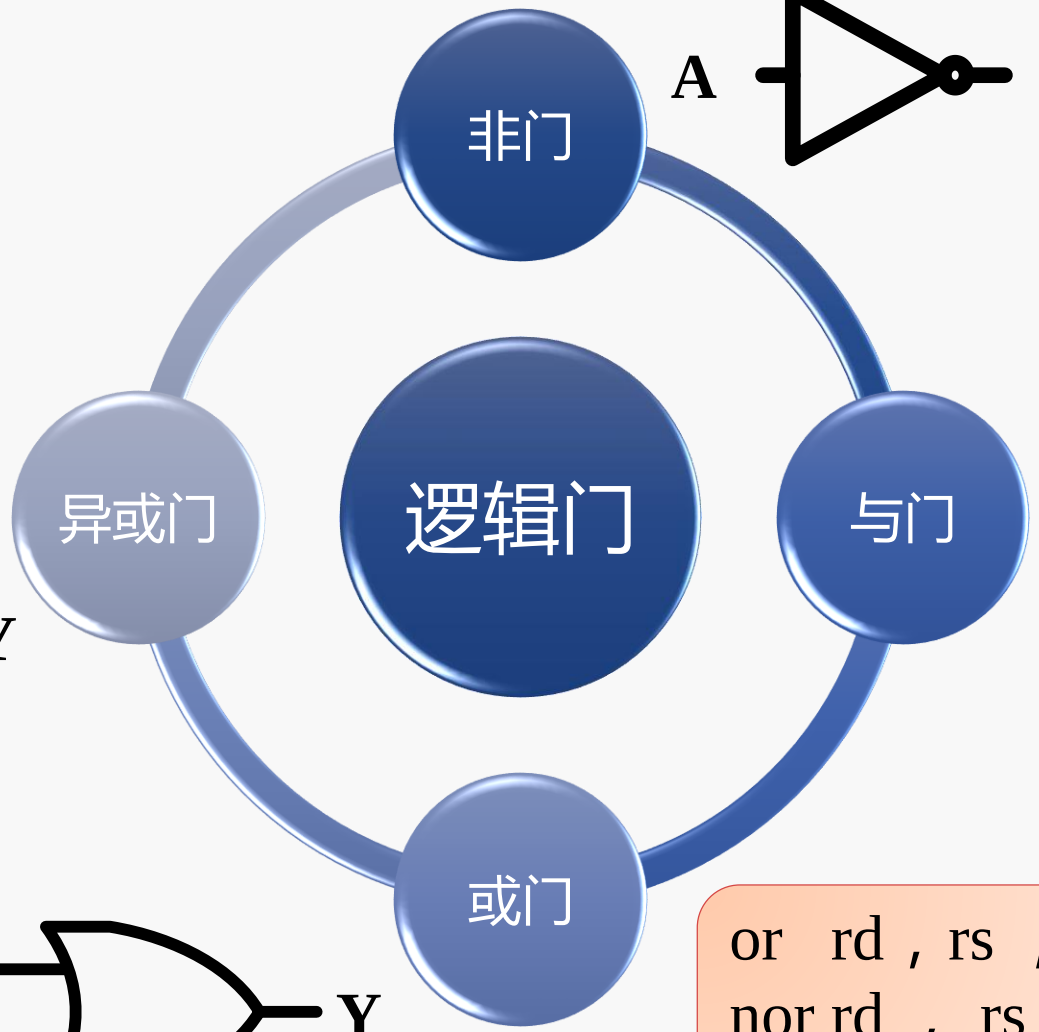
晶体管、逻辑门

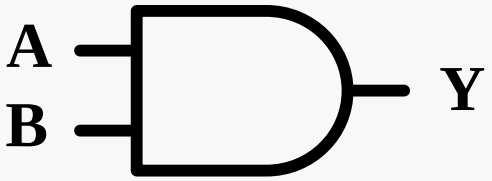


NMOS

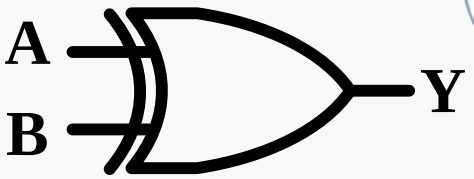


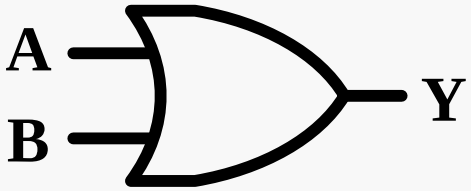
PMOS





and rd , rs , rt
andi rt , rs , imm





or rd , rs , rt
nor rd , rs , rt
ori rt , rs , imm

第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

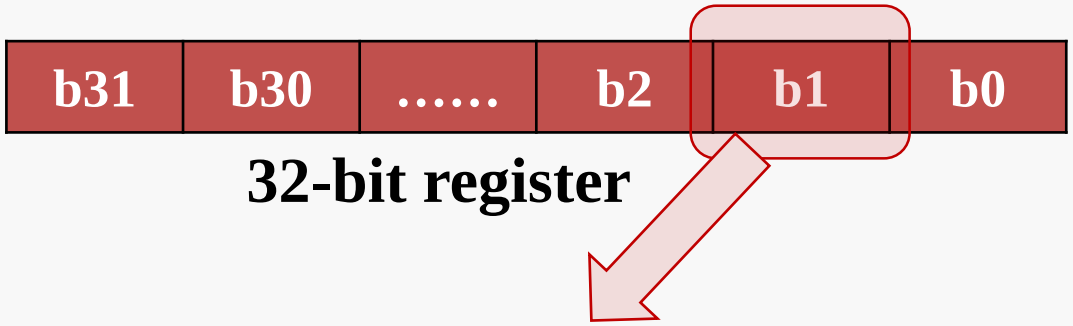
3.寄存器的基本原理

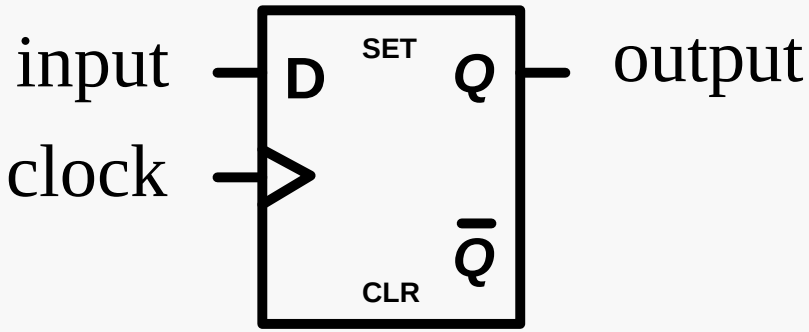
4.逻辑运算的实现

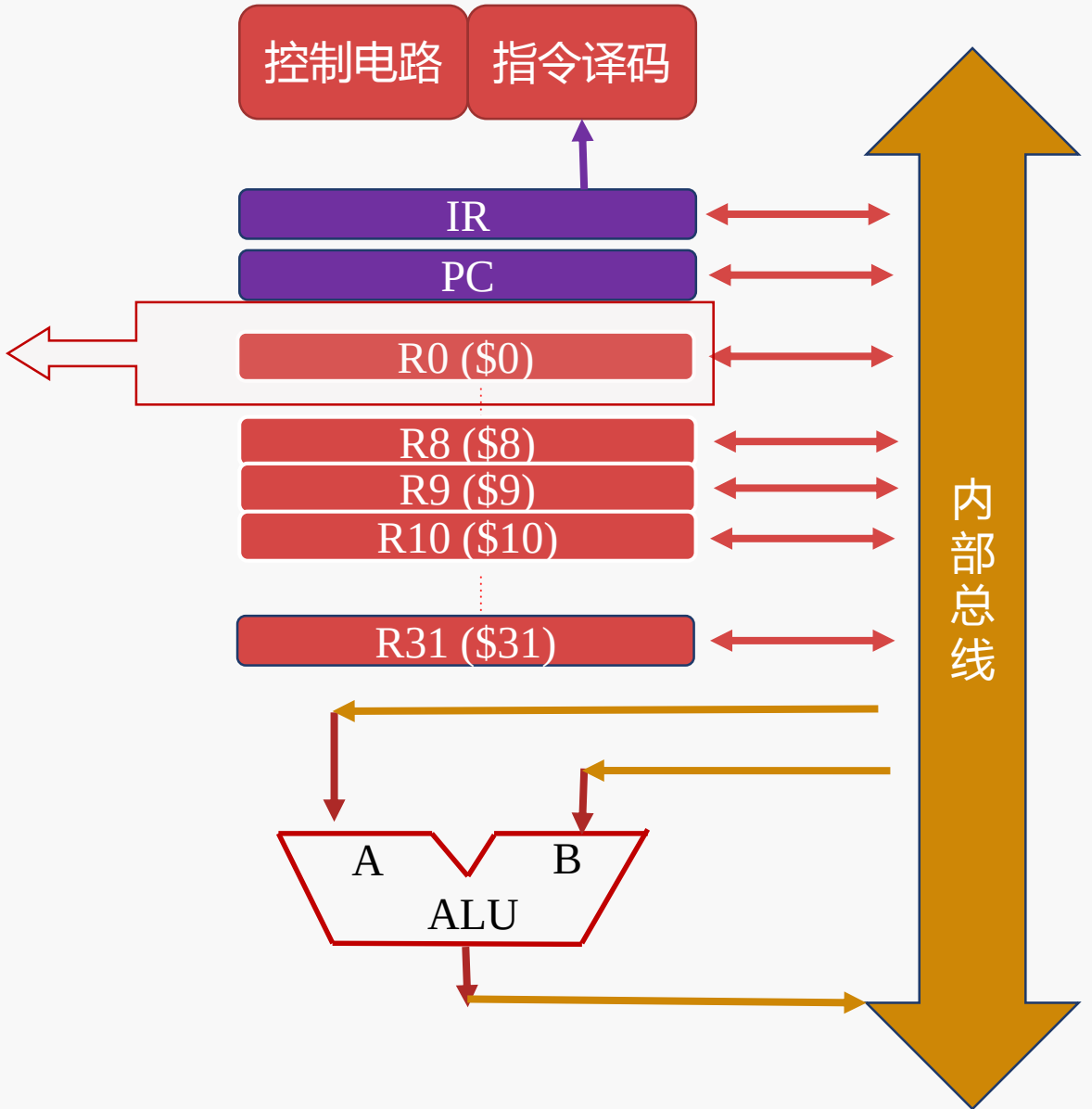
5.加法和减法的实现

6.加法器的优化

寄存器的内部结构


b31
b30
……
b2
b1
b0
32-bit register

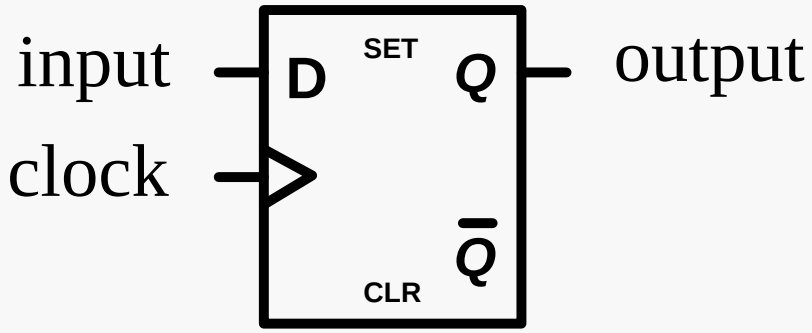

input
clock
D
SET
Q
Q
CLR
output


控制电路
指令译码
IR
PC
R0 ($0)
R8 ($8)
R9 ($9)
R10 ($10)
R31 ($31)
内部总线
A
B
ALU

D触发器（D flip-flop，DFF）

D触发器

- 具有存储信息能力的基本单元
- 由若干逻辑门构成，有多种实现方式
- 主要有一个数据输入、一个数据输出和一个时钟输入
- 在时钟clock的上升沿（0→1），采样输入D的值，传送到输出Q，其余时间输出Q的值不变



D触发器的工作原理

照相机+显示器 → D触发器

按快门后1秒钟，显示器上显示照片 → CLK-to-Q时间为1秒

每10秒钟按一次快门 → 时钟频率为0.1Hz

D触发器的工作原理

照相机+显示器 → D触发器

每10秒钟按一次快门 → 时钟频率为0.1Hz

按快门后1秒钟，显示器上显示照片 → CLK-to-Q时间为1秒

按快门前后，待拍摄的画面不能有变化 → Setup/Hold时间

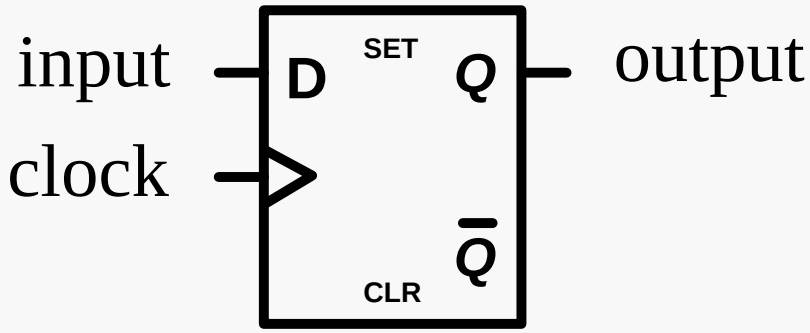
D触发器的工作原理

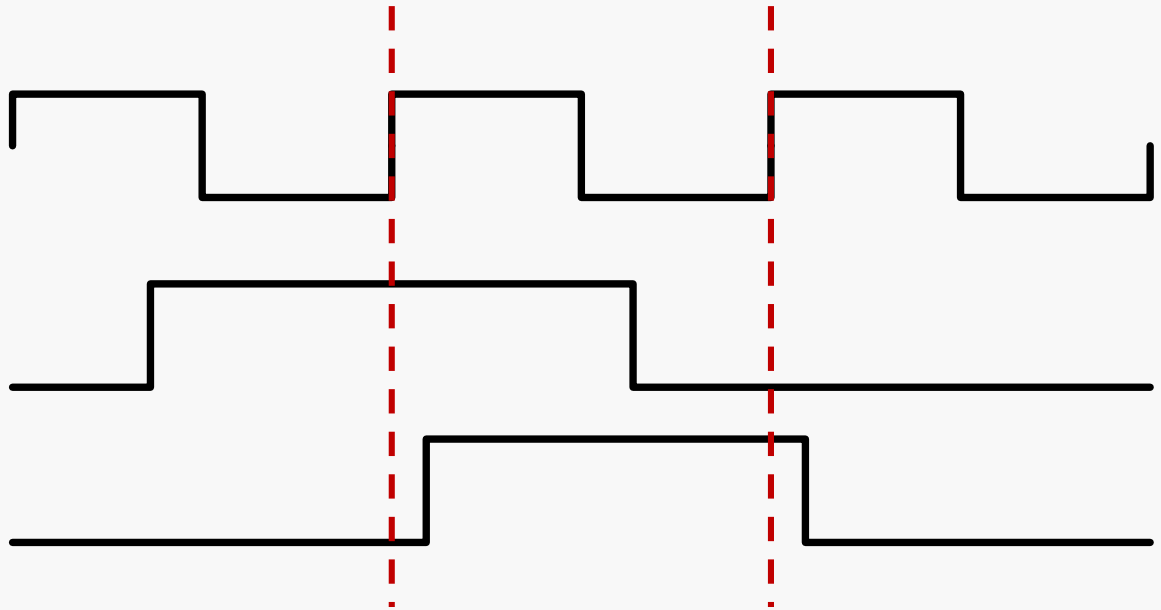
两个相连的D触发器

D触发器（D flip-flop，DFF）

D触发器

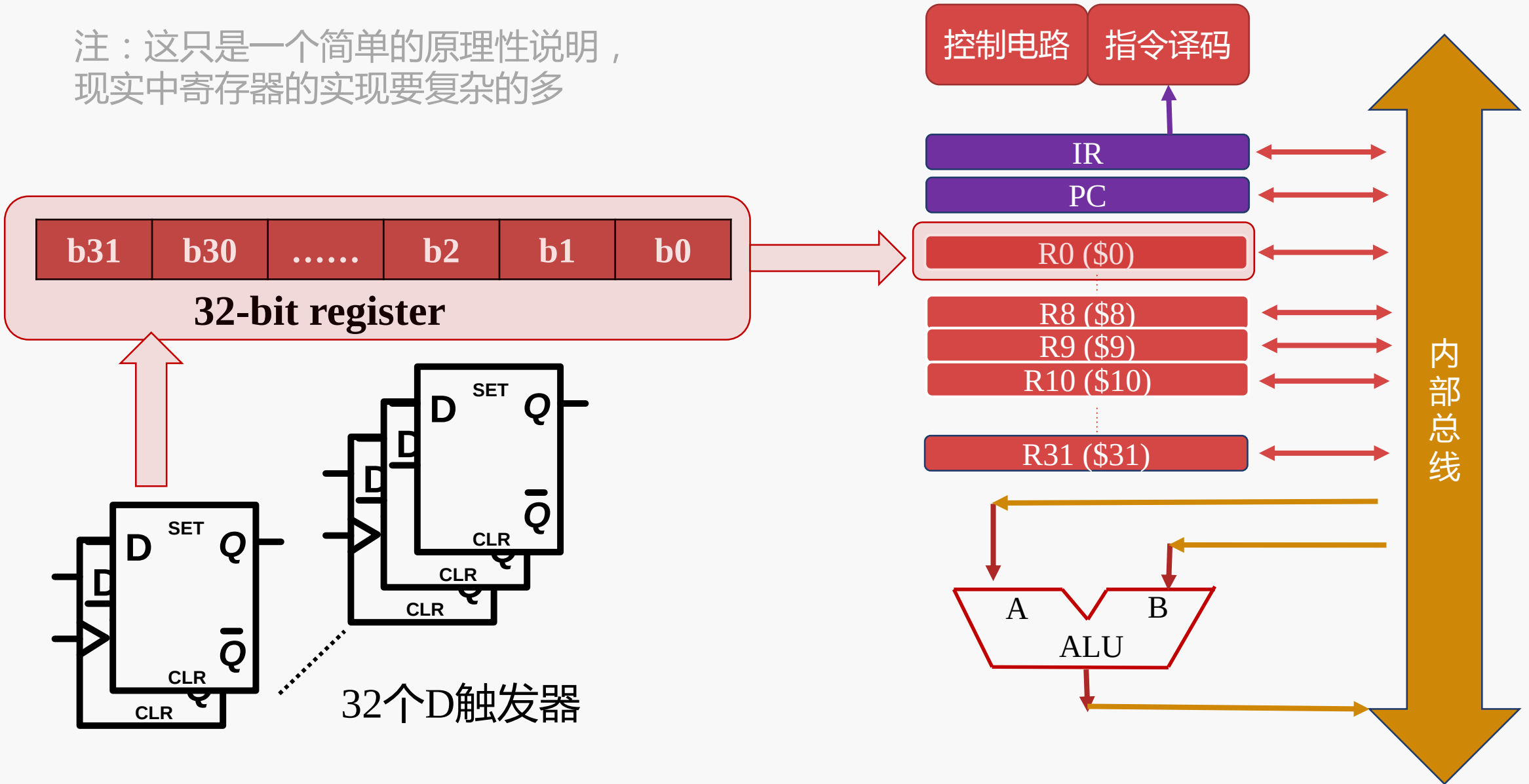
- 具有存储信息能力的基本单元
- 由若干逻辑门构成，有多种实现方式
- 主要有一个数据输入、一个数据输出和一个时钟输入
- 在时钟clock的上升沿（0→1），采样输入D的值，传送到输出Q，其余时间输出Q的值不变





寄存器的构成

注：这只是一个简单的原理性说明，
现实中寄存器的实现要复杂的多



第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

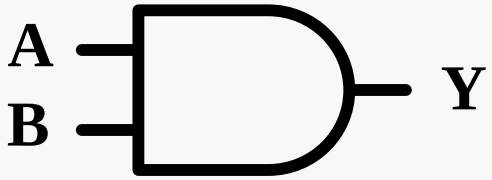
3.寄存器的基本原理

4.逻辑运算的实现

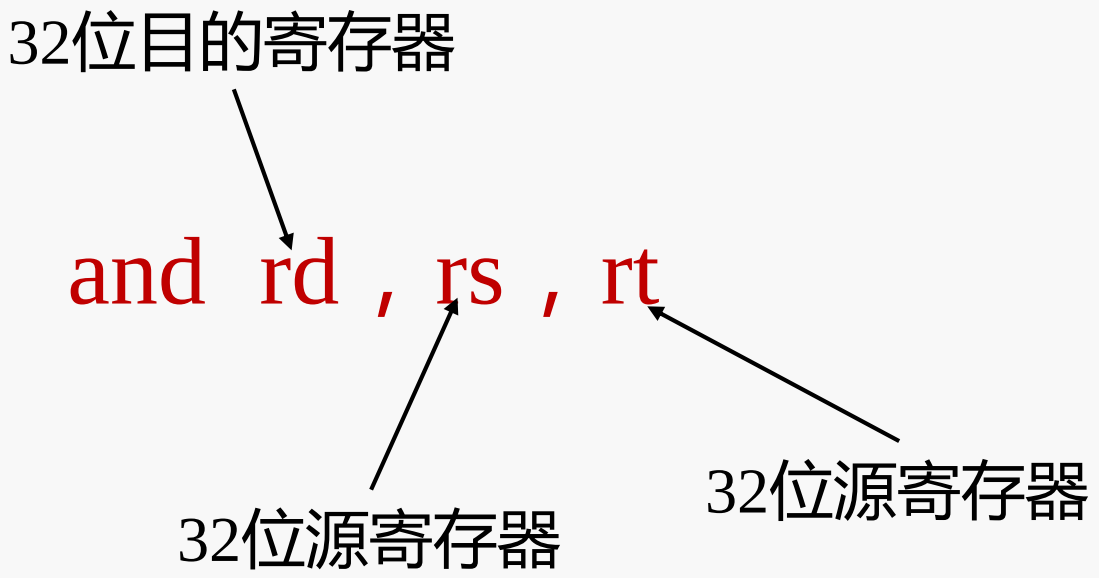
5.加法和减法的实现

6.加法器的优化

与门 和 与运算指令

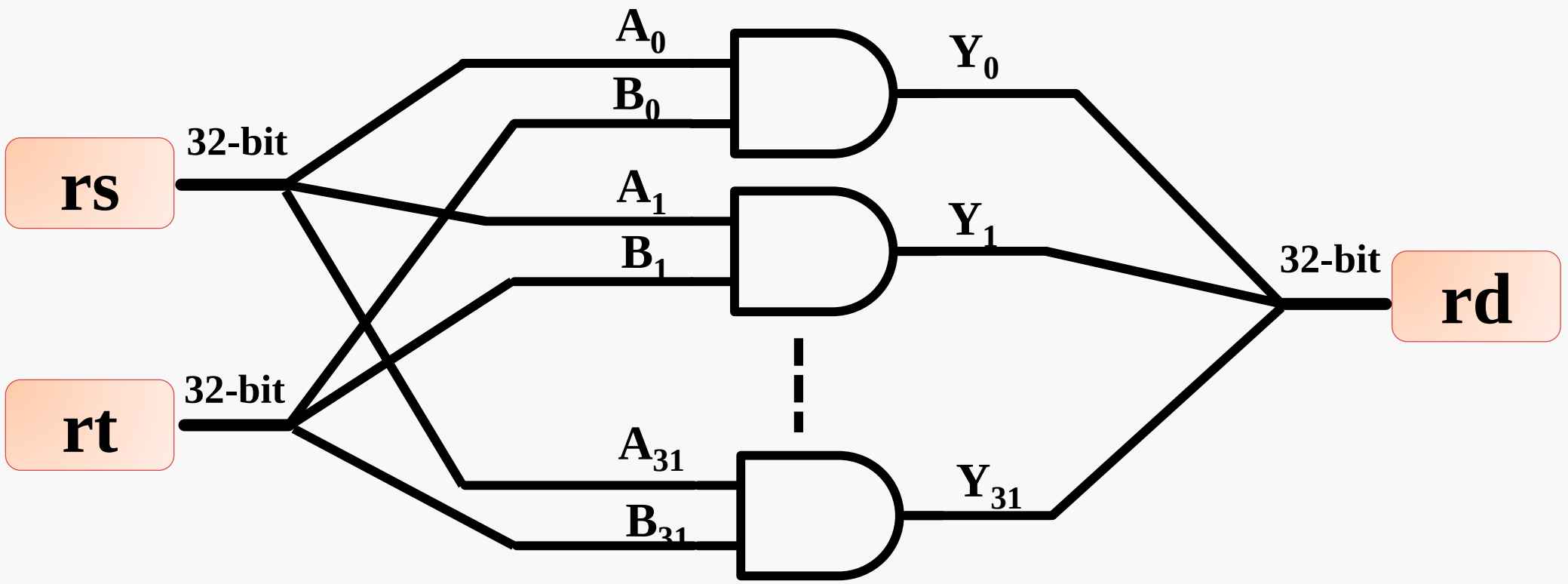


输入A	输入B	输出Y
0	0	0
0	1	0
1	0	0
1	1	1



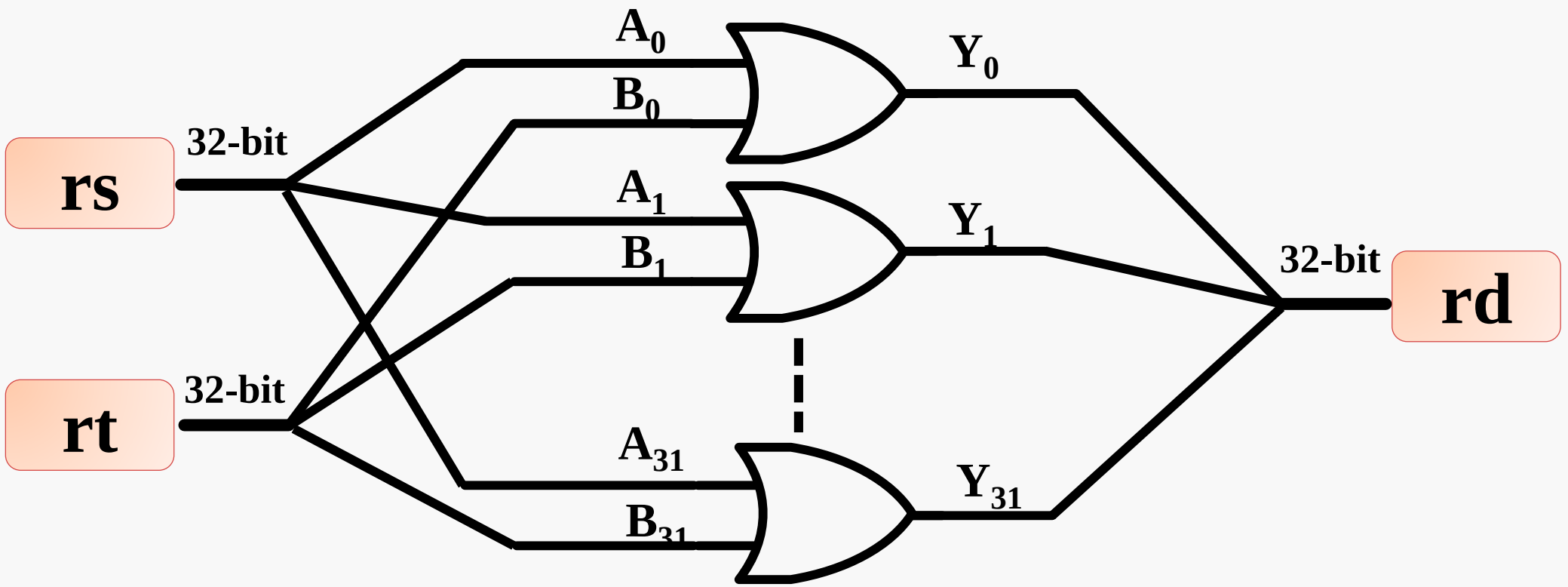
与运算的实现

and rd，rs，rt

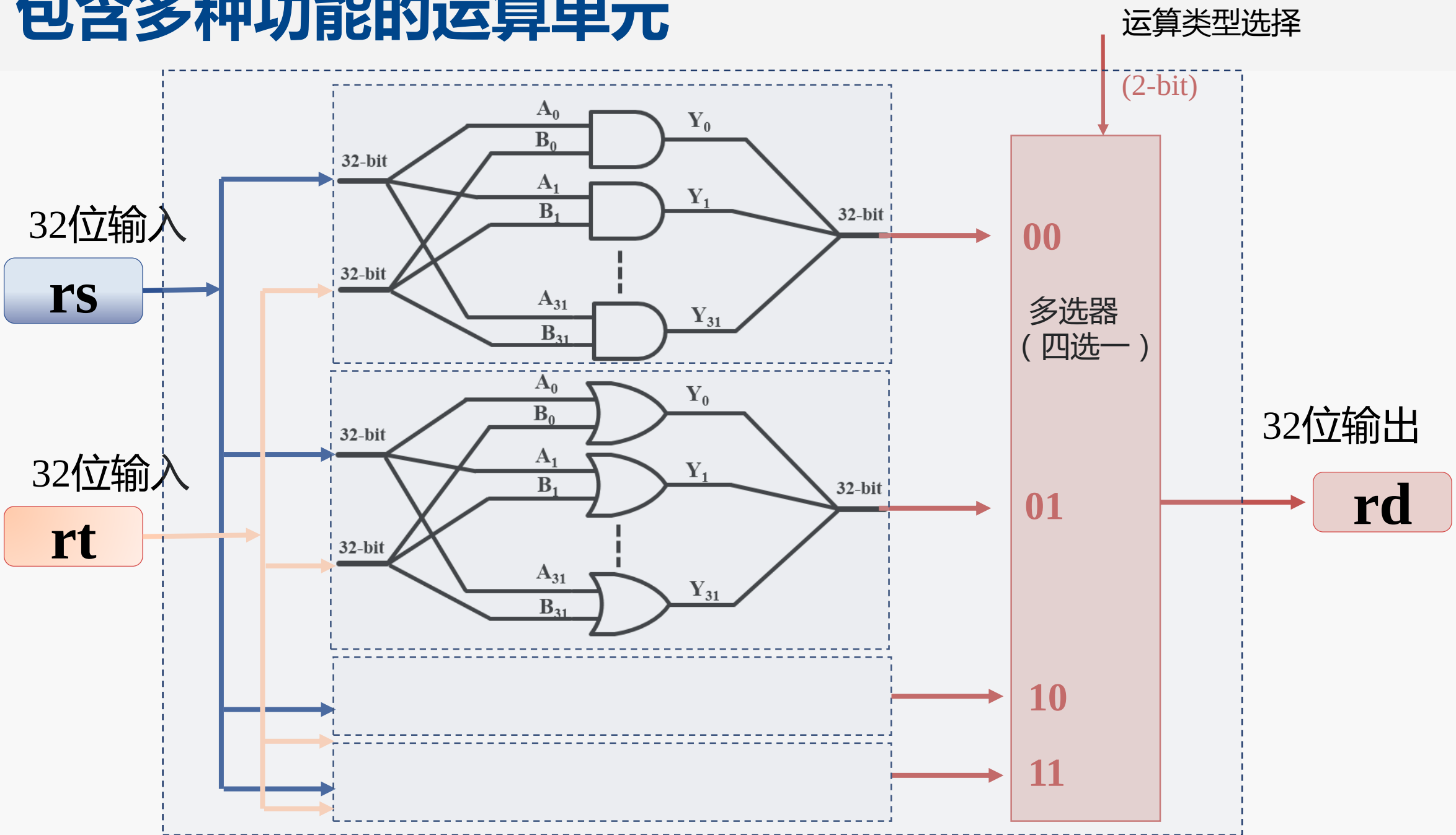


或运算的实现

or　　rd，rs，rt

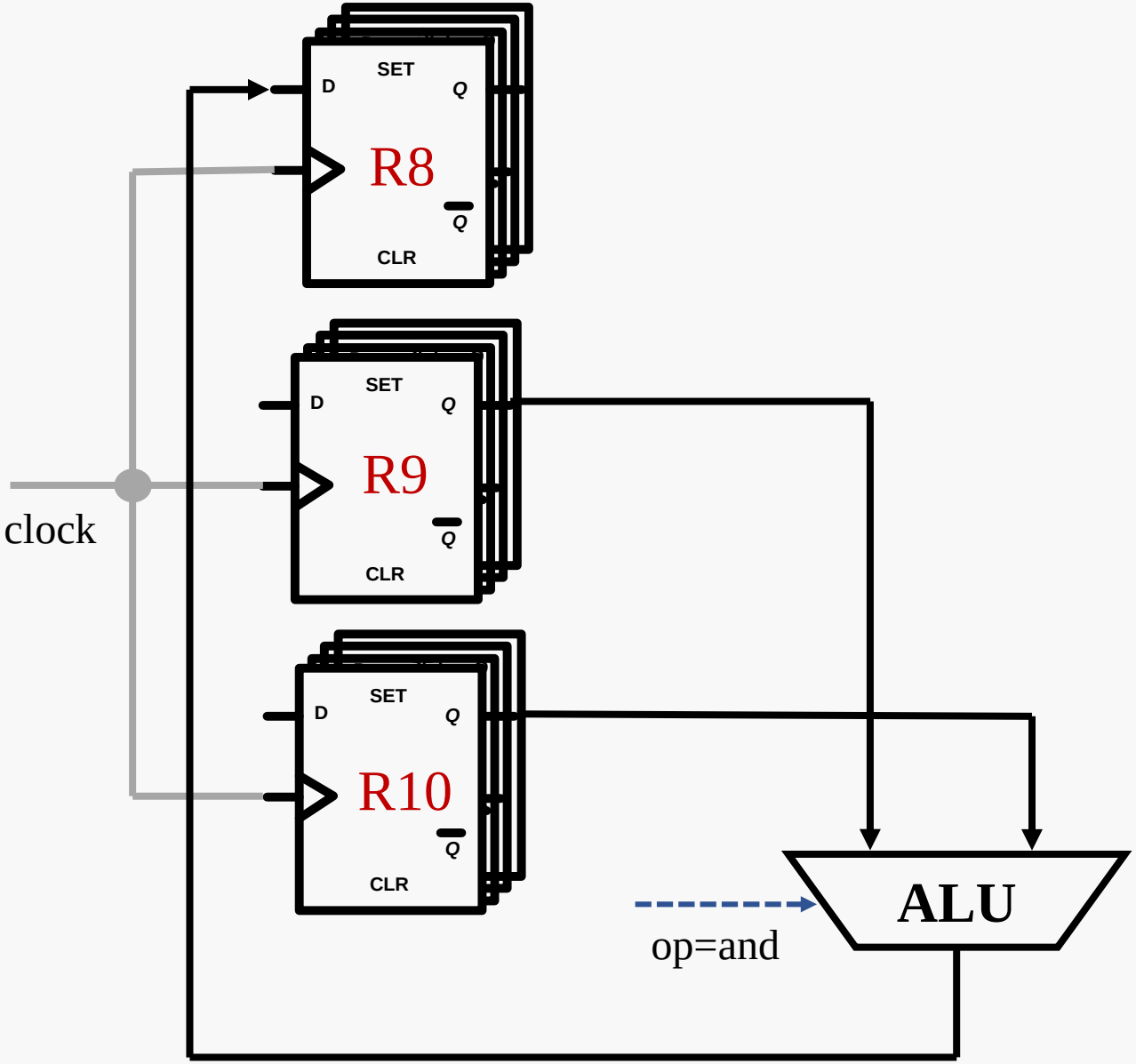


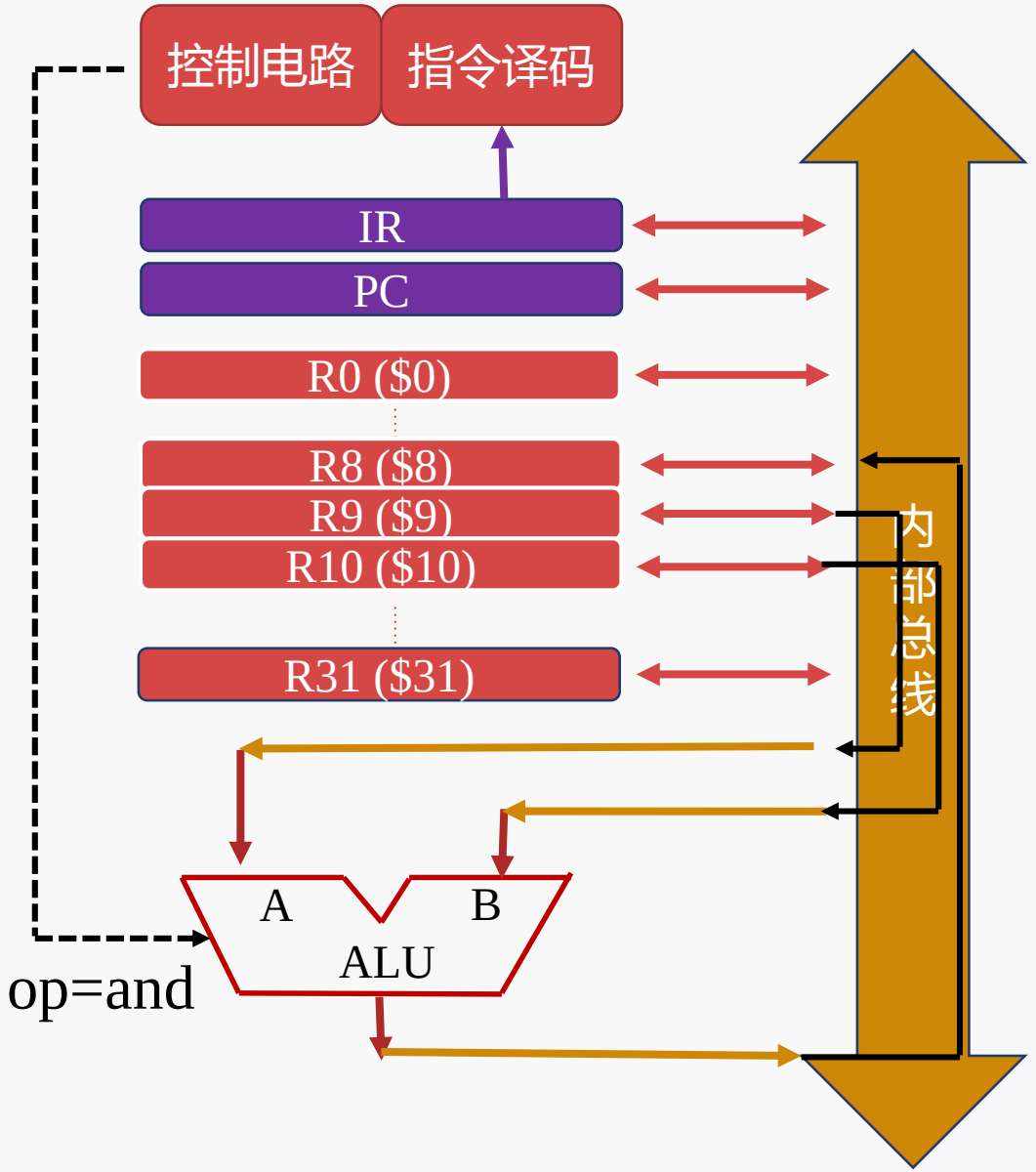
包含多种功能的运算单元



逻辑运算示例

and $8，$9，$10





第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

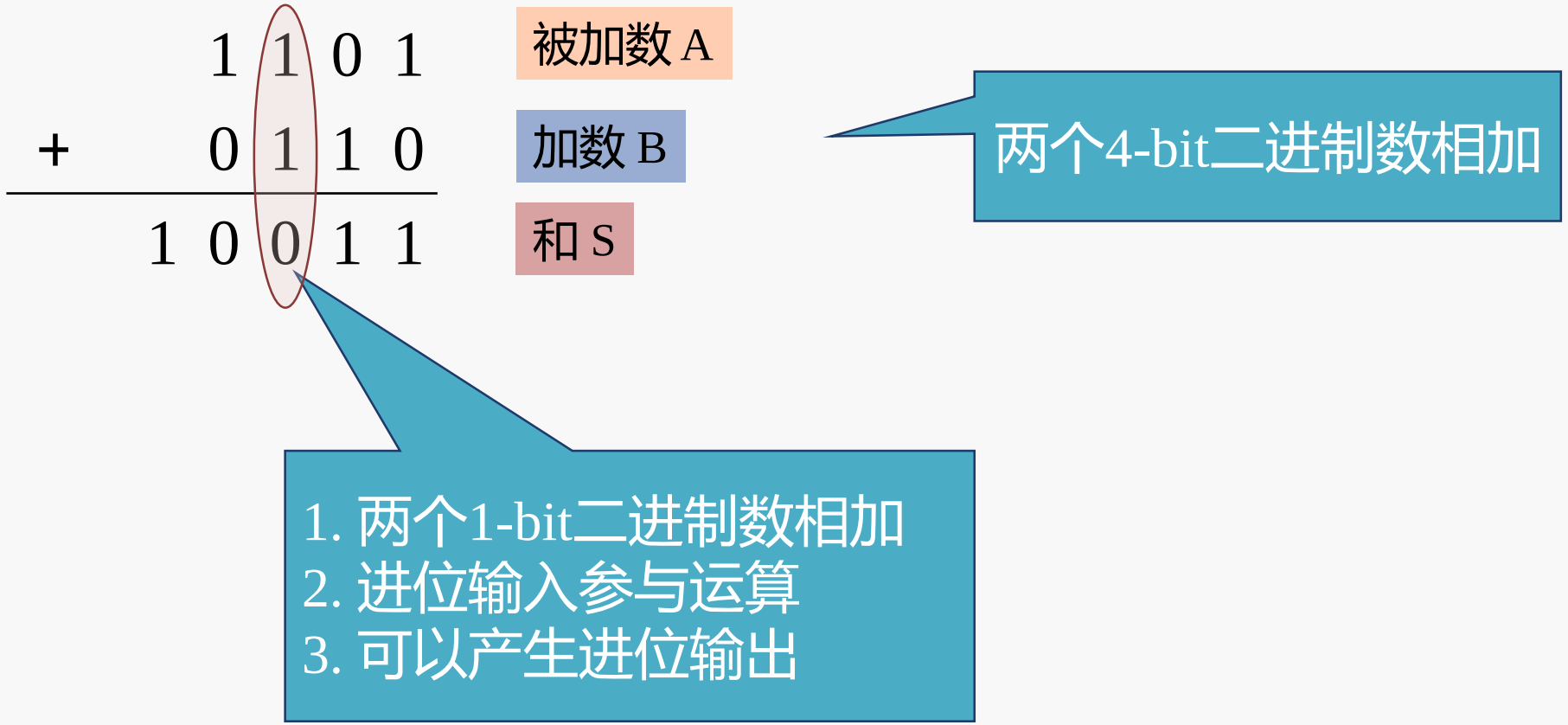
3.寄存器的基本原理

4.逻辑运算的实现

5.加法和减法的实现

6.加法器的优化

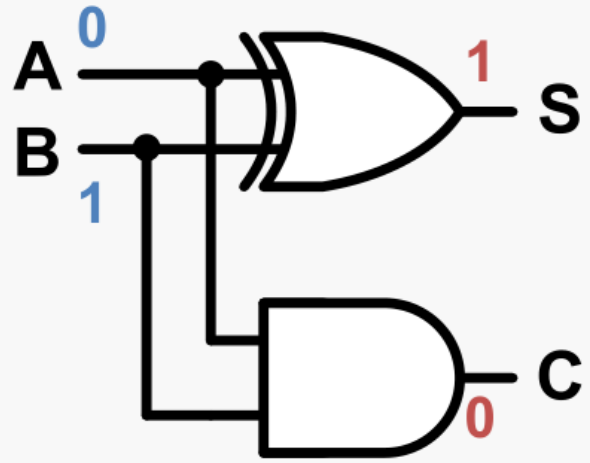
二进制的加法


1 1 0 1
+ 0 1 1 0
1 0 0 1 1
被加数 A
加数 B
和 S
两个4-bit二进制数相加
1. 两个1-bit二进制数相加
2. 进位输入参与运算
3. 可以产生进位输出

半加器（Half Adder）

半加器的功能是将两个一位二进制数相加

- 输入端口A、B
- 输出端口S（和）、C（进位）

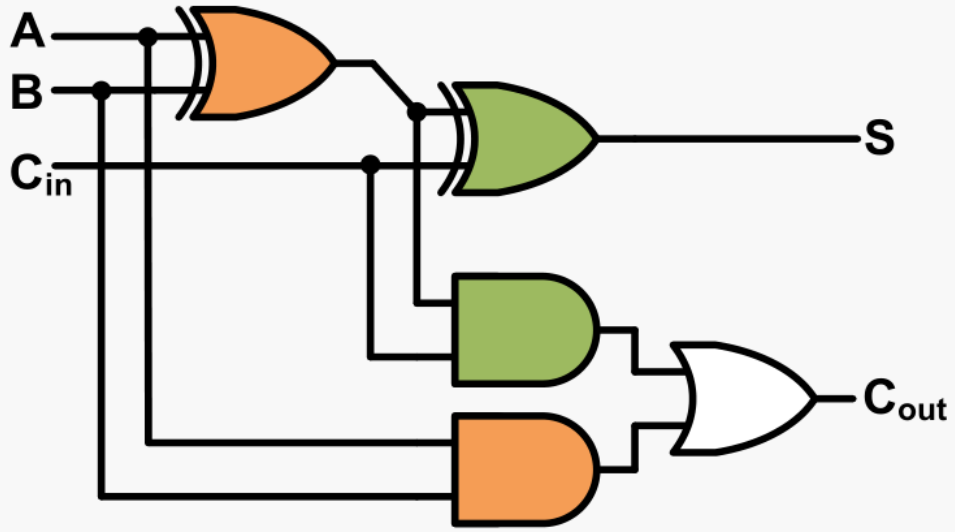


A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

全加器（Full Adder）

全加器由两个半加器构成

- 输入端口A、B、C in（进位输入）
- 输出端口S（和）、C out（进位输出）



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

4-bit加法器

```
        1 1 0 1
+       0 1 1 0
-----------------
    1   0 0 1 1
```

被加数 A

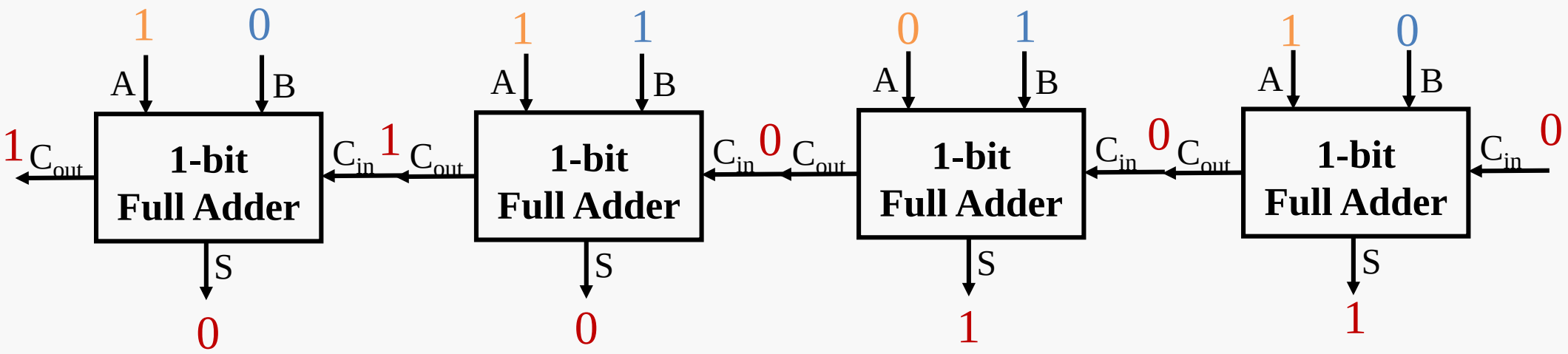
加数 B

进位C

和 S

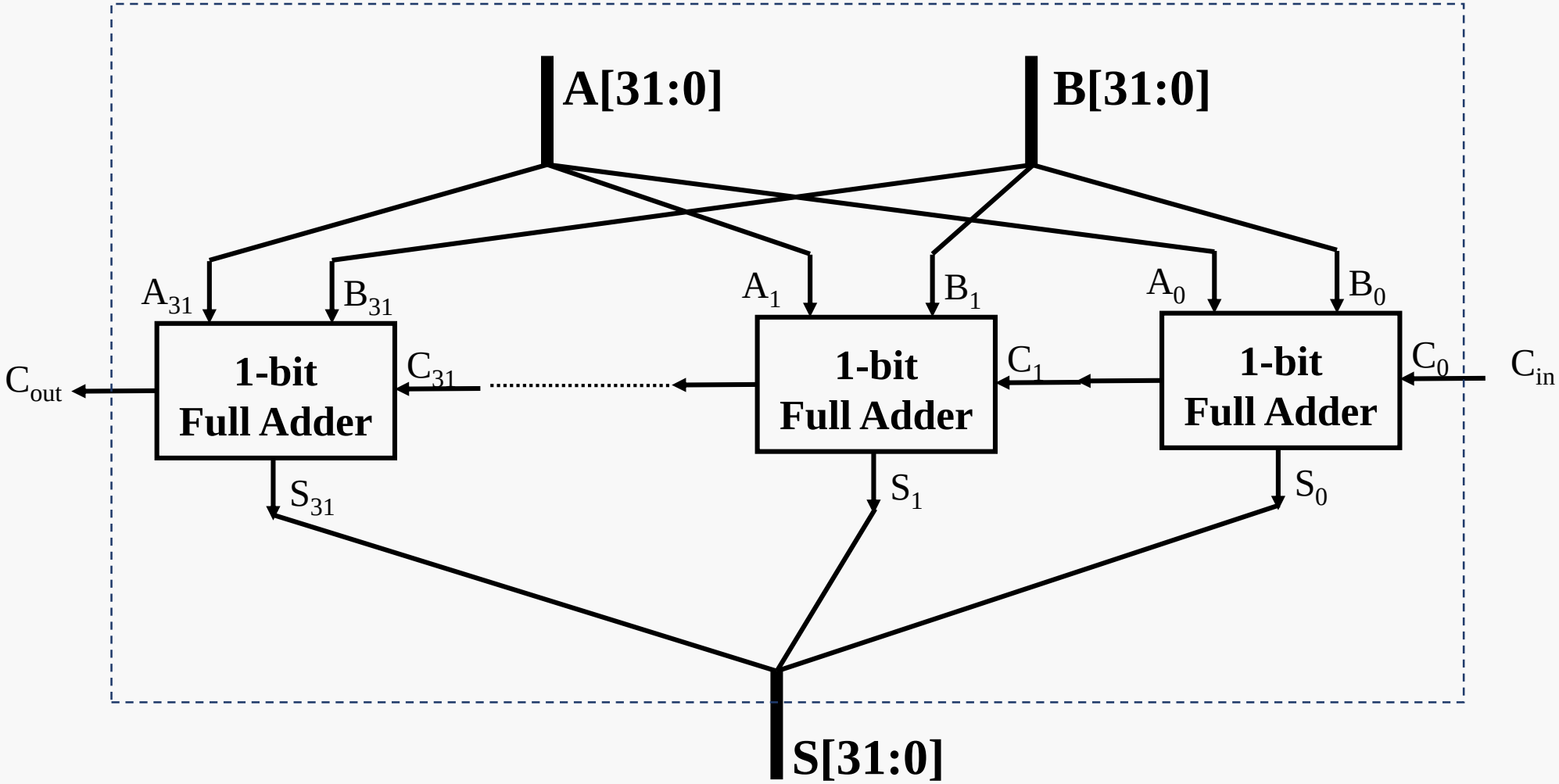
两个4-bit二进制数相加

由四个全加器
构成的4-bit加法器



加法运算的实现示例

add rd，rs，rt　　addu rd，rs，rt



检查加法运算结果是否溢出

“溢出”（overflow）

- 运算结果超出了正常的表示范围

“溢出”仅针对有符号数运算

- 两个正数相加，结果为负数
- 两个负数相加，结果为正数

```
  0 0 1 1
+ 0 1 0 1
---------
  1 0 0 0
```

无符号数：3 + 5 = 8

有符号数：3 + 5 = (-8)

“进位”和“溢出”示例

注意区分“进位”和“溢出”

- 有“溢出”时，不一定有“进位”
- 有“进位”时，不一定有“溢出”

无符号数：3 + 5 = 8
有符号数：3 + 5 = (-8)

```
  0 0 1 1
+ 0 1 0 1
---------
0 1 0 0 0
```

有“溢出”，无“进位”

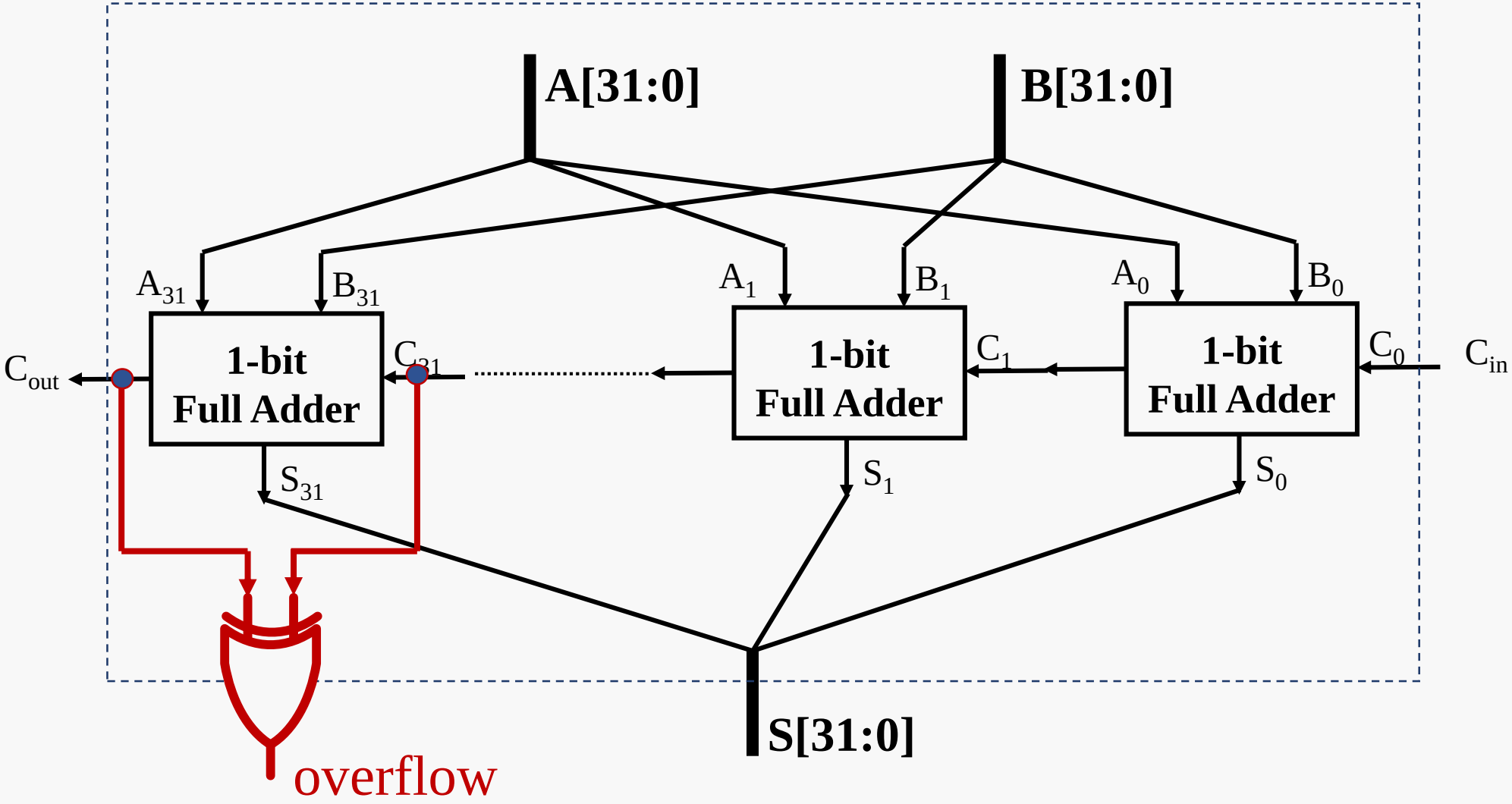
无符号数：14 + 12 = 8
有符号数：(-2) + (-4) = (-6)

```
  1 1 1 0
+ 1 1 0 0
---------
1 1 0 1 0
```

有“进位”，无“溢出”

“进位”和“溢出”示例

“最高位的进位输入”不等于“最高位的进位输出”



对“溢出”的处理方式：MIPS

提供两类不同的指令分别处理

（1）将操作数看做有符号数，发生“溢出”时产生异常

- add rd，rs，rt　　#　R[rd]=R[rs]+R[rt]
- addi rt，rs，imm　　#　R[rt]=R[rs]+SignExtImm

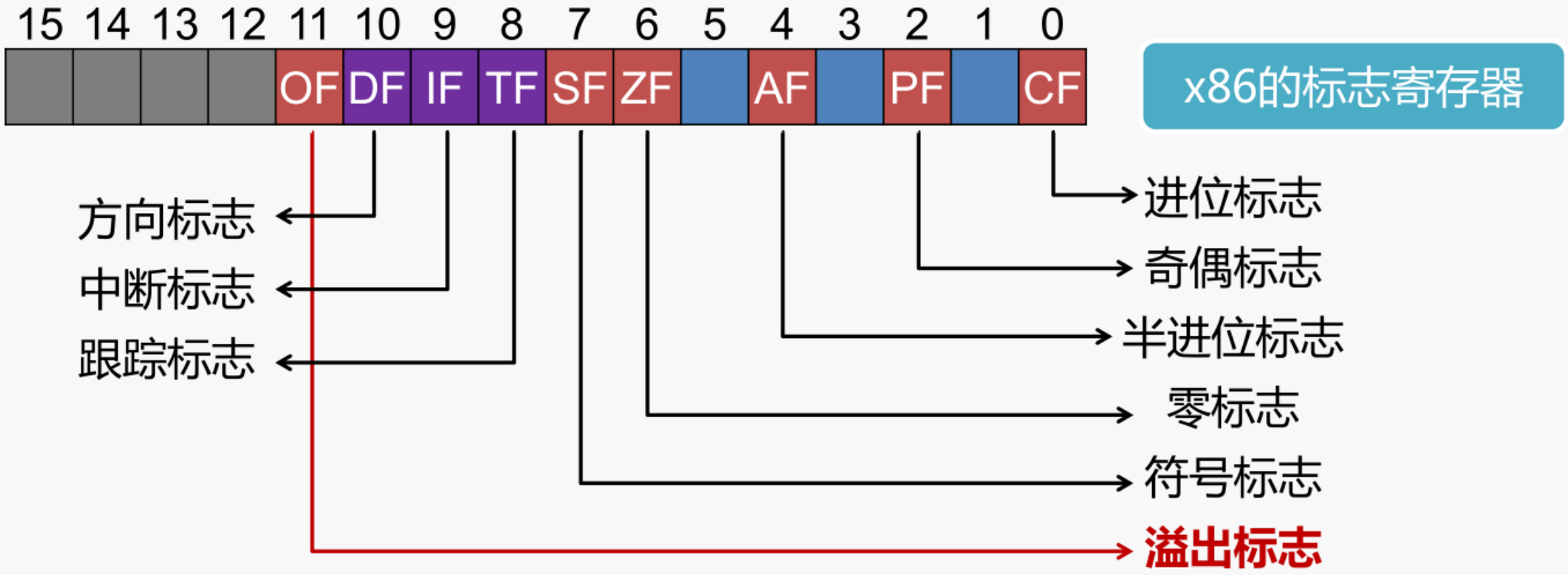
（2）将操作数看做无符号数，不处理“溢出”

- addu rd，rs，rt　　#　R[rd]=R[rs]+R[rt]
- addiu rt，rs，imm　　#　R[rt]=R[rs]+SignExtImm

对“溢出”的处理方式：x86

溢出标志OF（Overflow Flag）

- 如果把操作数看做有符号数，运算结果是否发生溢出
- 若发生溢出，则自动设置OF=1；否则，OF=0



对“溢出”的处理方式：x86

减法运算均可转换为加法运算

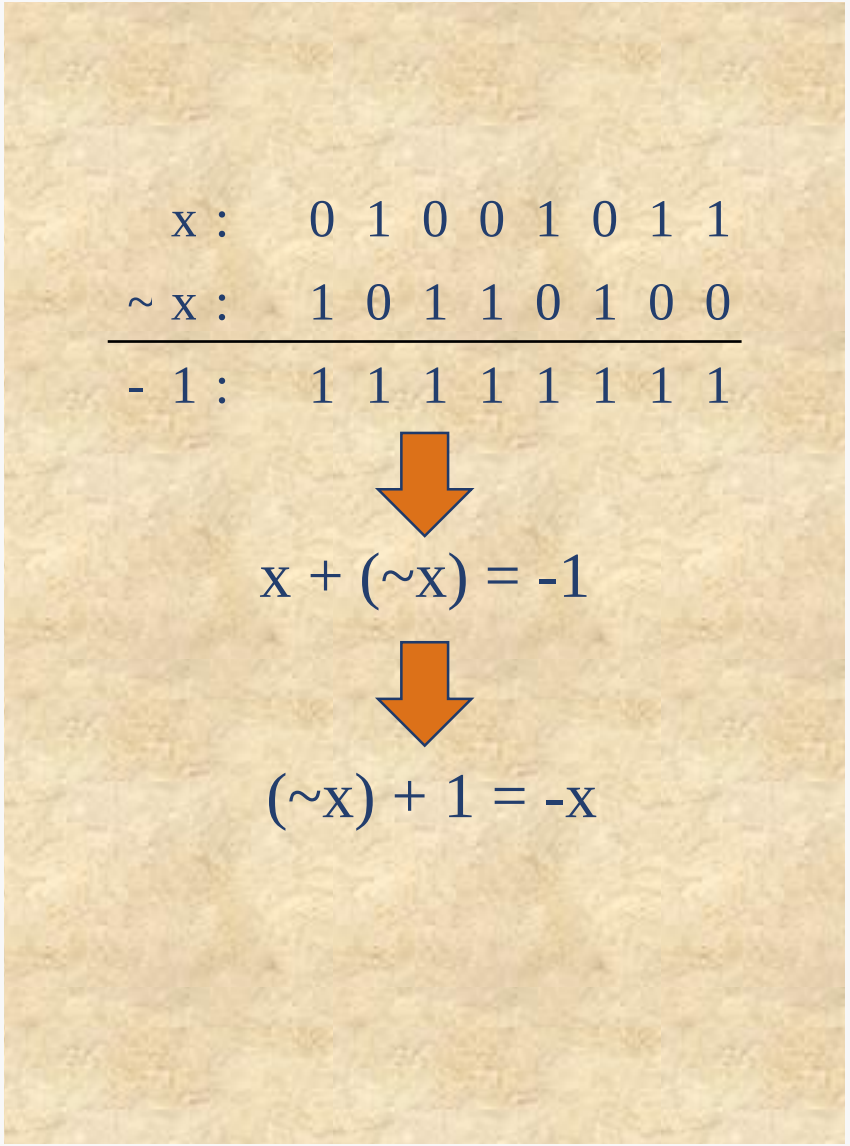
- A – B = A + (-B)

补码表示的二进制数的相反数

- 转换规则：按位取反，末位加一

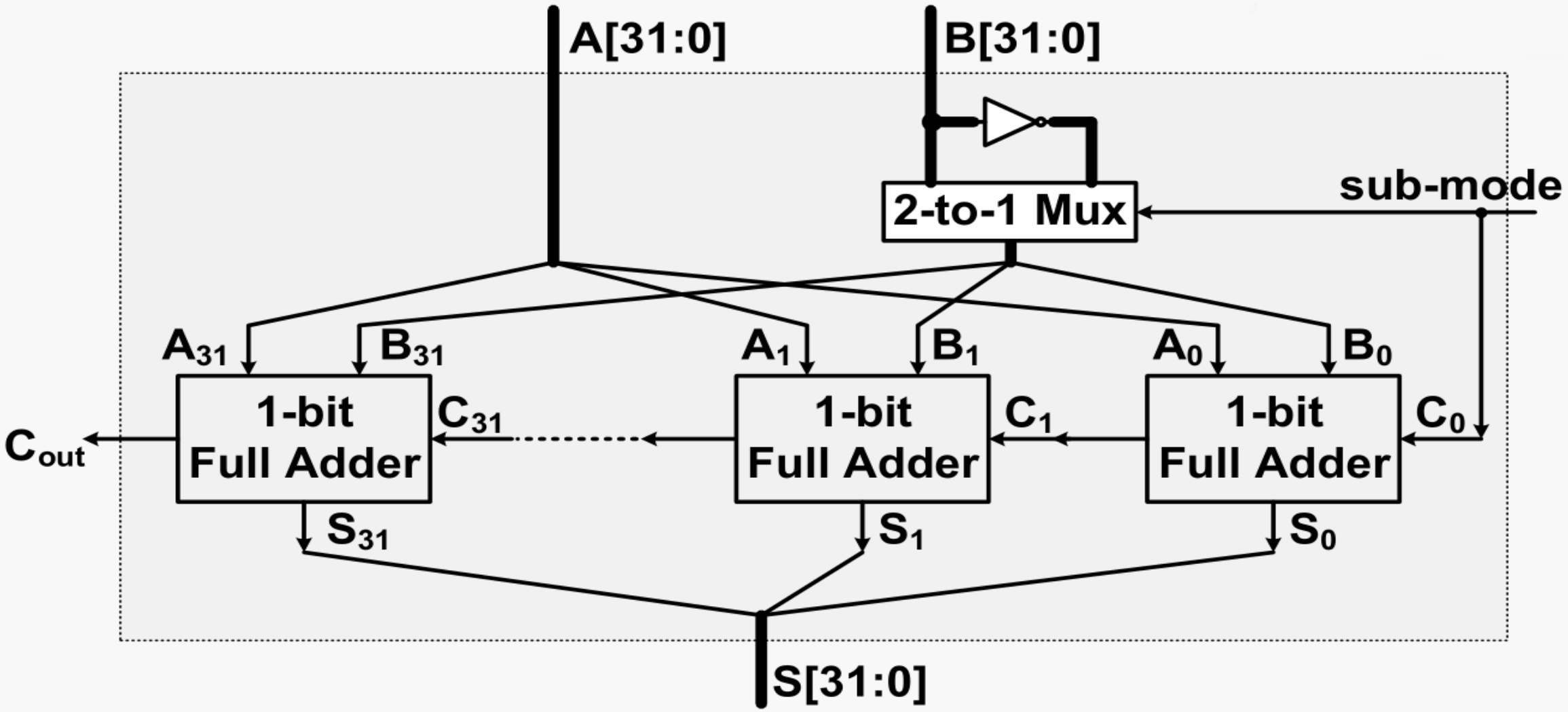
在加法器的基础上实现减法器

- A + (-B)= A + (~B+1)



减法运算的实现示例

A-B=A+(~B+1)



第三章　算术逻辑单元

1.算术运算和逻辑运算

2.门电路的基本原理

3.寄存器的基本原理

4.逻辑运算的实现

5.加法和减法的实现

6.加法器的优化

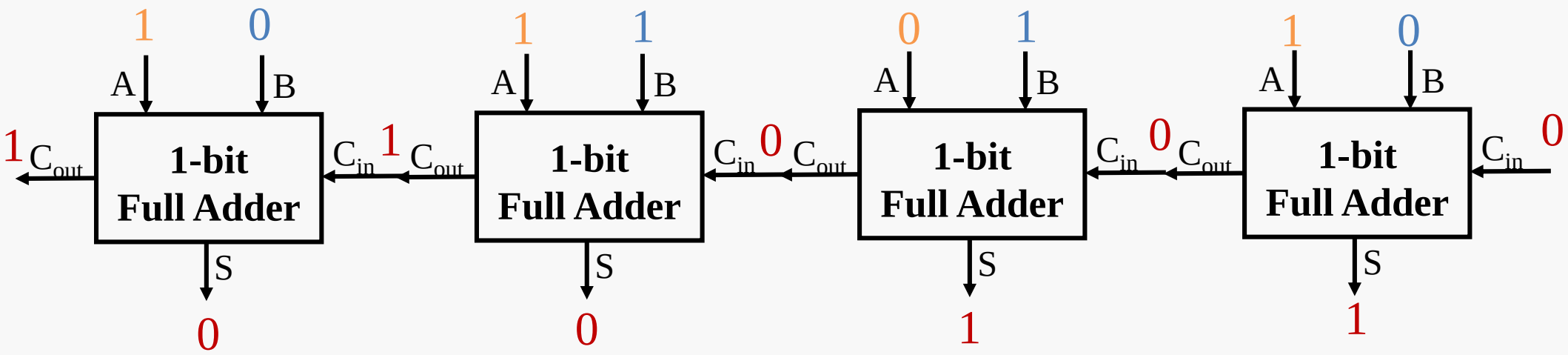
4-bit加法器示例

```
      1 1 0 1     被加数 A
+     0 1 1 0     加数 B
-----------------
  1   0 0 1 1
```

进位C　　和 S

两个4-bit二进制数相加

由四个全加器
构成的4-bit加法器



行波进位加法器（Ripple-Carry Adder，RCA）

结构特点

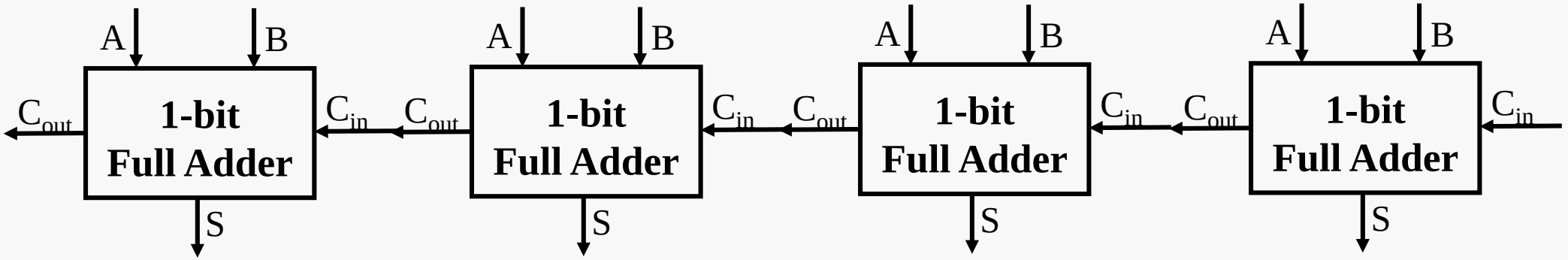
- 低位全加器的C out 连接到高一位全加器C in

优点

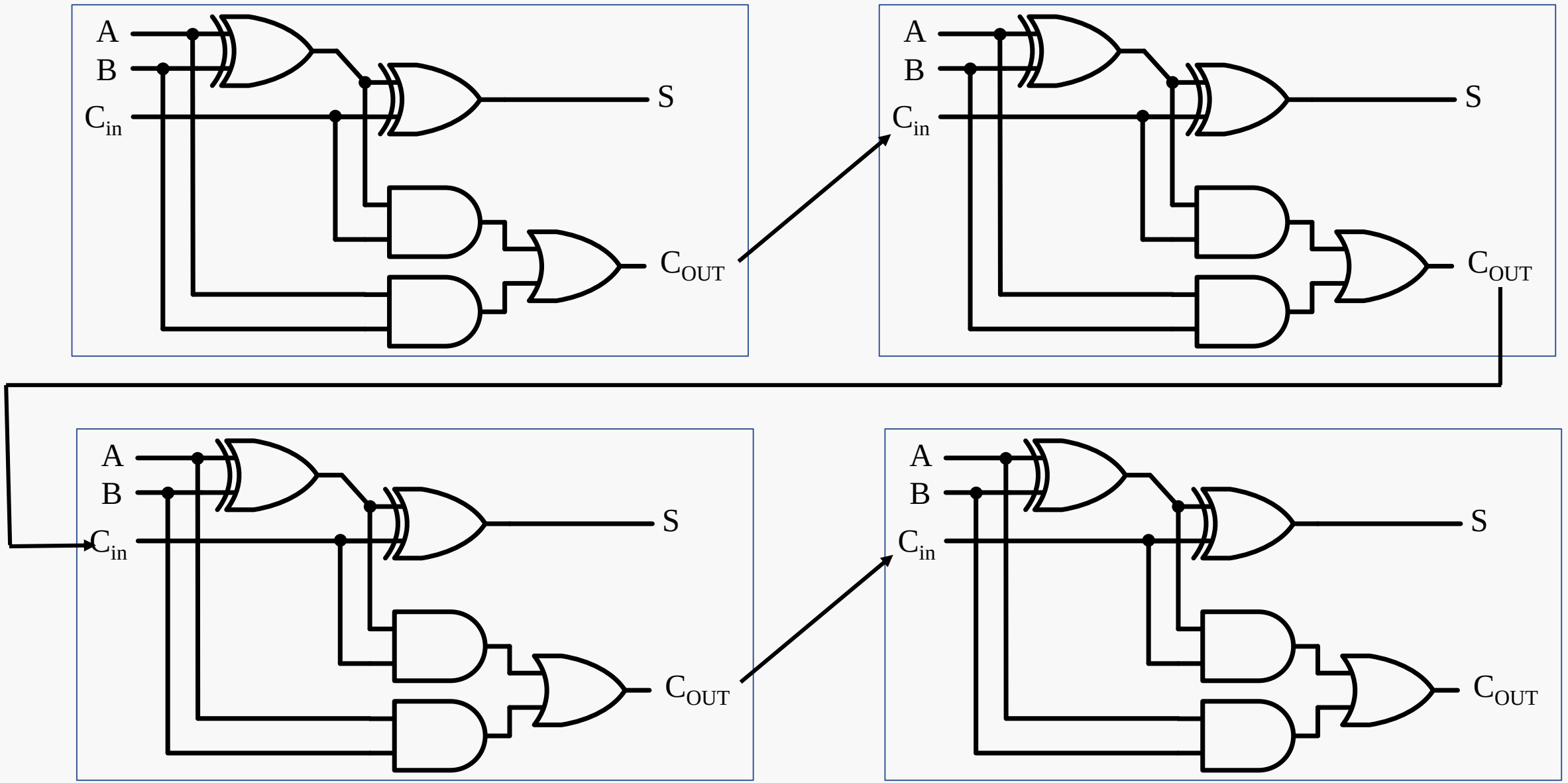
- 电路布局简单，设计方便

缺点

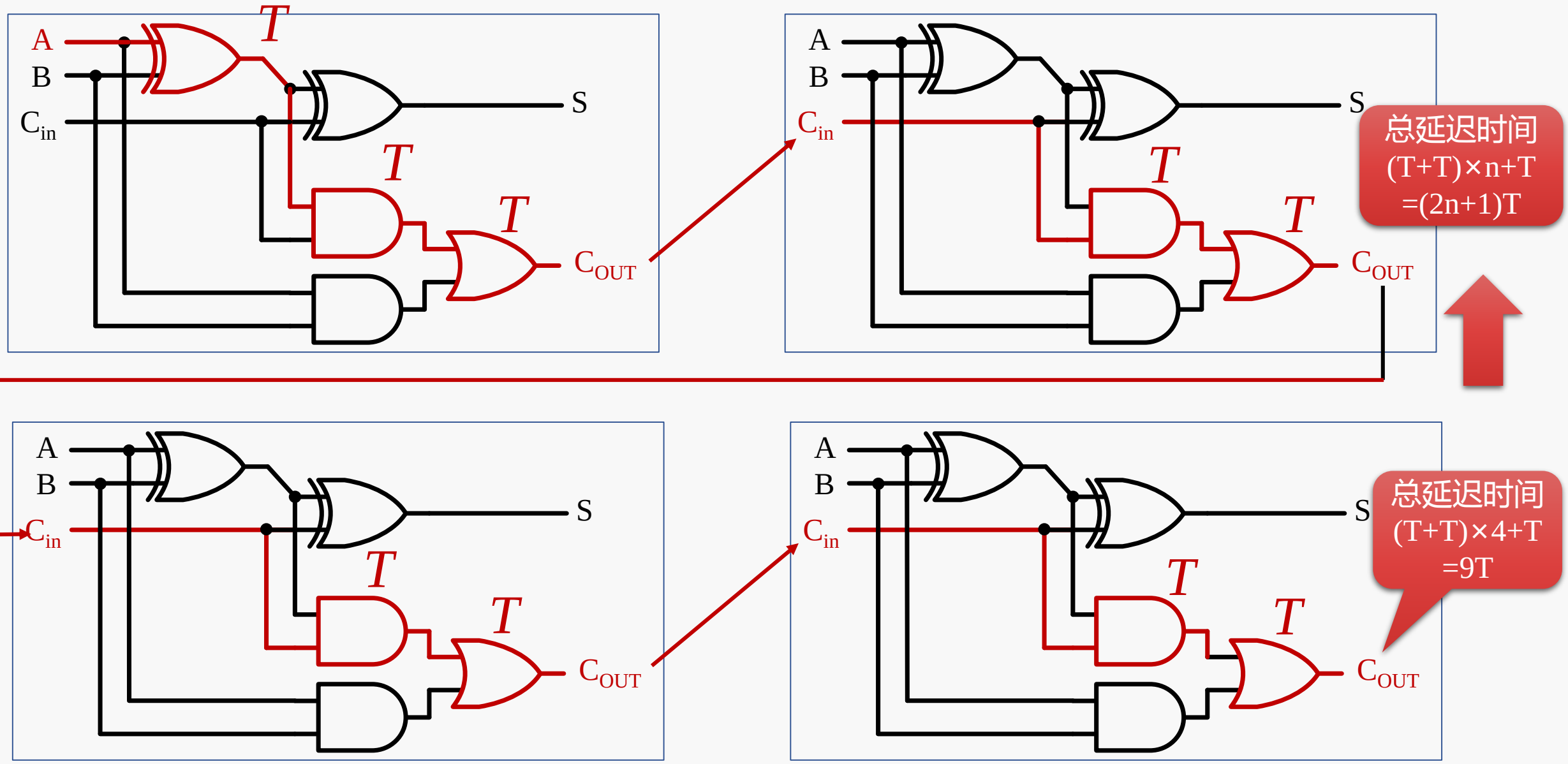
- 高位的运算必须等待低位的运算完成，延迟时间长



4-bit RCA的门电路实现



4-bit RCA的关键路径（延迟最长的路径）

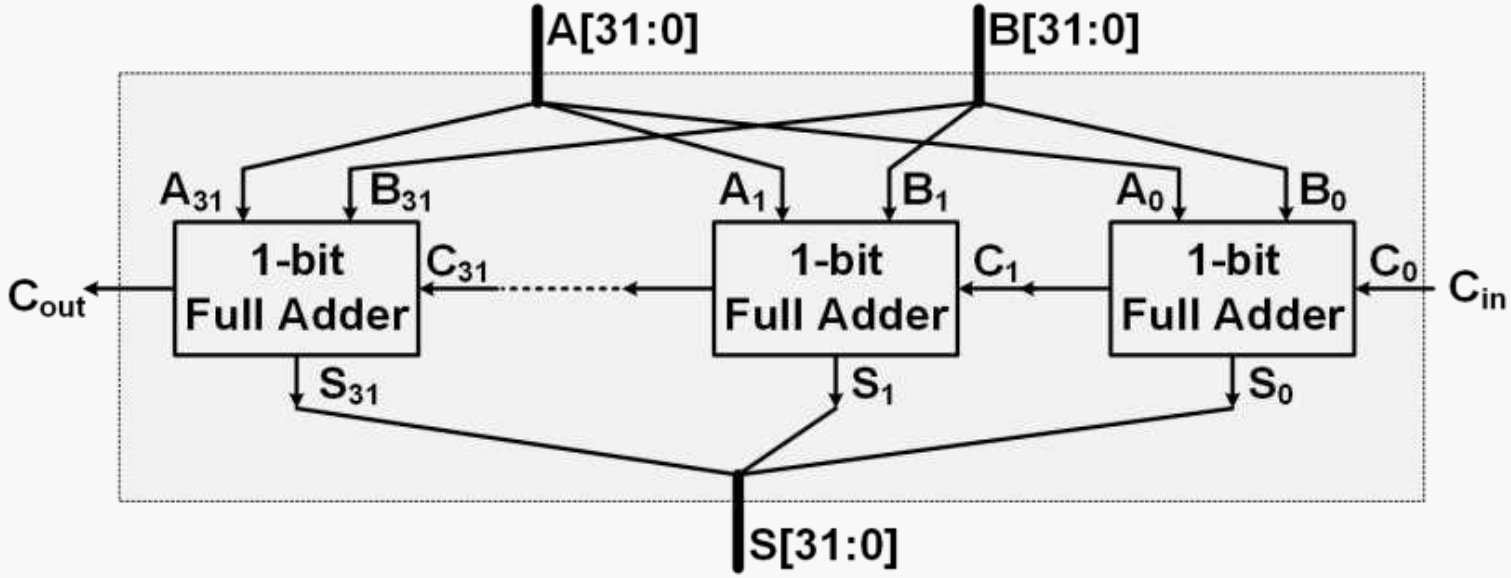


32-bit RCA的性能分析

总延迟时间：
$(2n+1)T = (2\times32+1)\times T = 65T$

参考值
水果智能手机5s的A7 SoC
采用28nm制造工艺
主频1.3GHz（0.66ns）

	延迟时间	时钟频率
4-bit RCA	0.18ns	5.56GHz
32-bit RCA	1.3ns	769MHz
注：参照28nm制造工艺，门延迟T设为0.02ns		



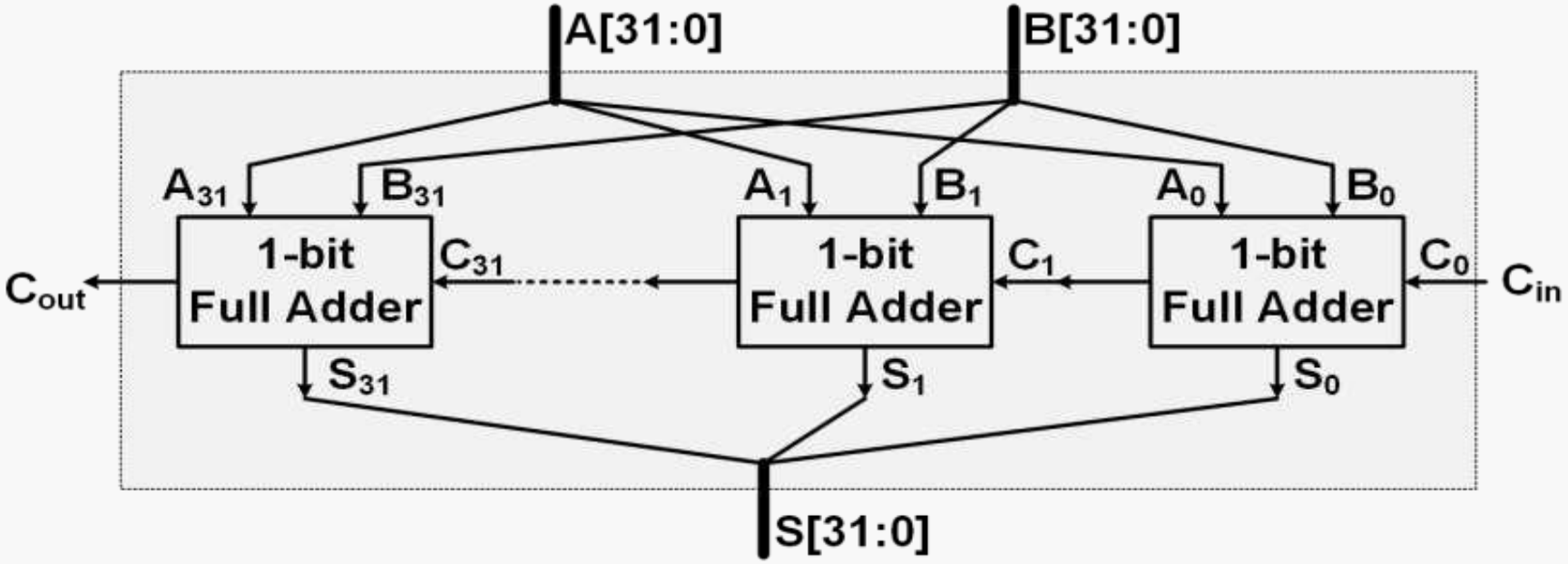
加法器的优化思路

主要问题

- 高位的运算必须等待低位的“进位输出信号”

优化思路

- 能否提前计算出“进位输出信号” ？



进位输出信号的分析

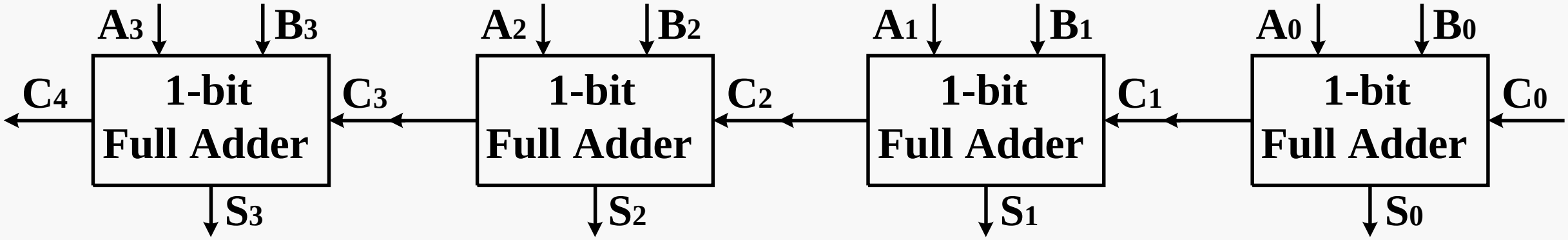
$$C_{i+1} = (A_i \cdot B_i) + (A_i \cdot C_i) + (B_i \cdot C_i)$$

$$= (A_i \cdot B_i) + (A_i + B_i) \cdot C_i$$

设：

- 生成（Generate）信号：$G_i = A_i \cdot B_i$
- 传播（Propagate）信号：$P_i = A_i + B_i$

则：$\mathbf{C_{i+1} = G_i + P_i \cdot C_i}$



如何提前计算“进位输出信号”

$C_1 = G_0 + P_0 \cdot C_0$

$C_2 = G_1 + P_1 \cdot C_1$

$= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$

$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$

$C_3 = G_2 + P_2 \cdot C_2$

$= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$

$= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$

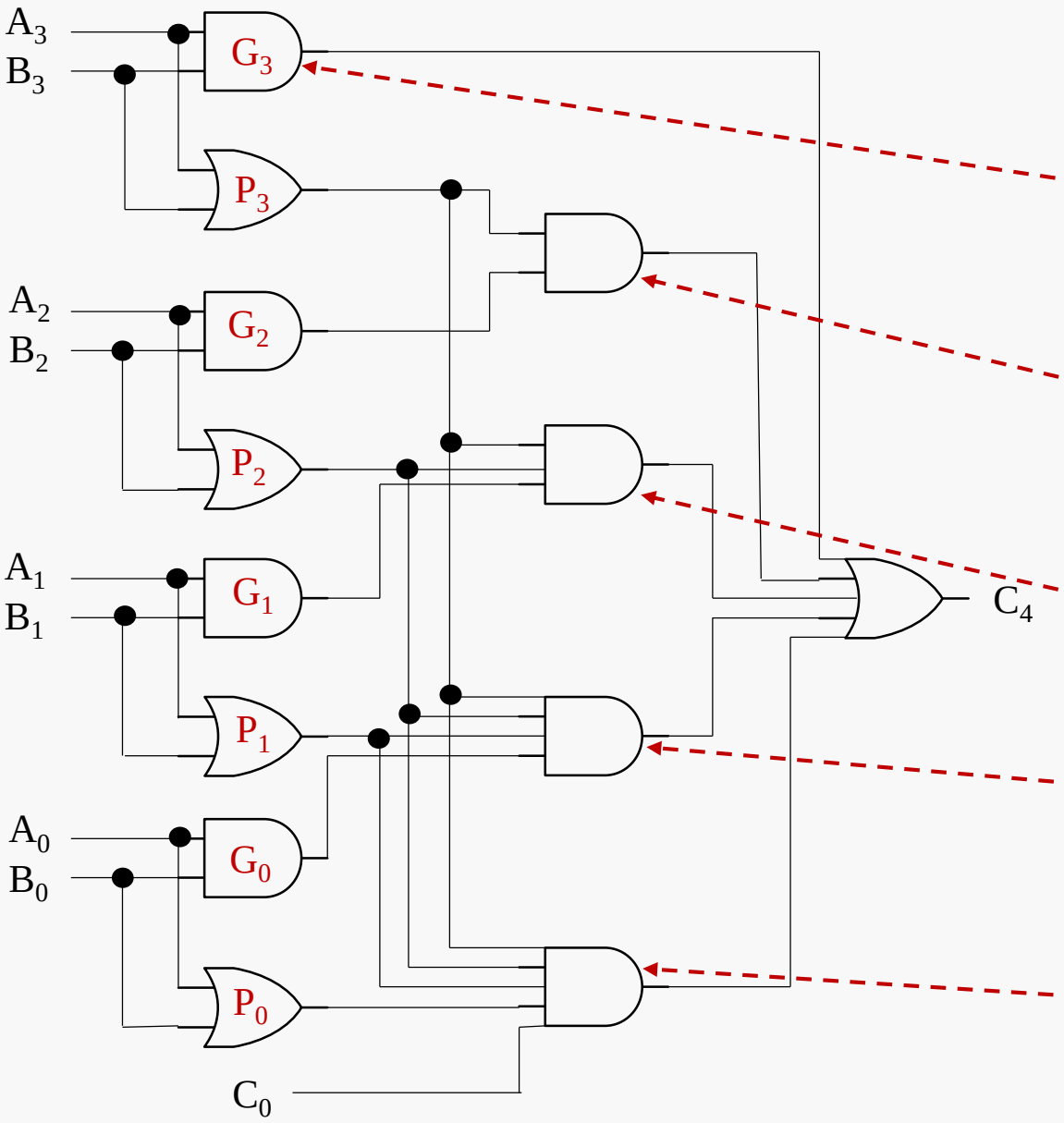
$C_4 = G_3 + P_3 \cdot C_3$

$= G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0)$

$= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$

$C_{i+1} = G_i + P_i \cdot C_i$

提前计算C4的电路实现

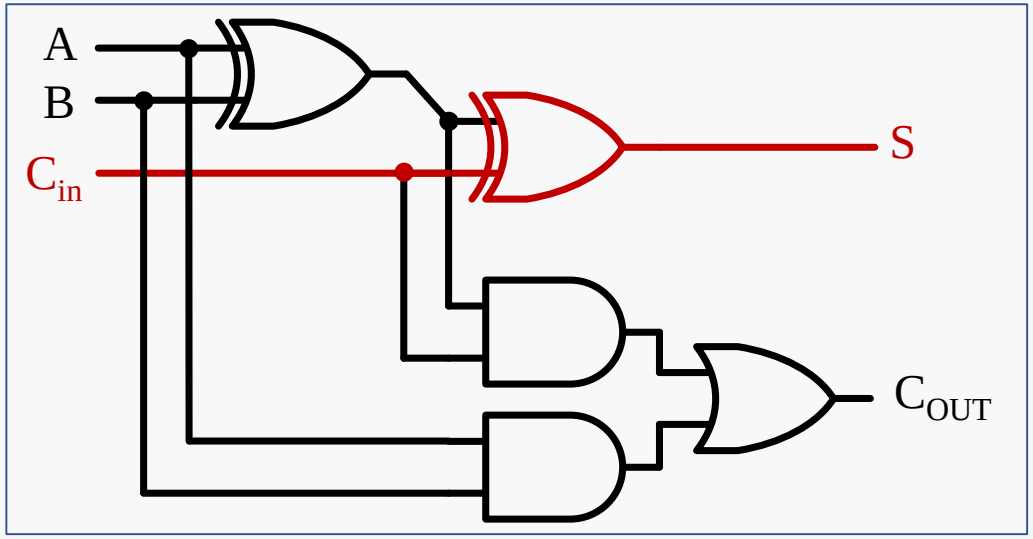


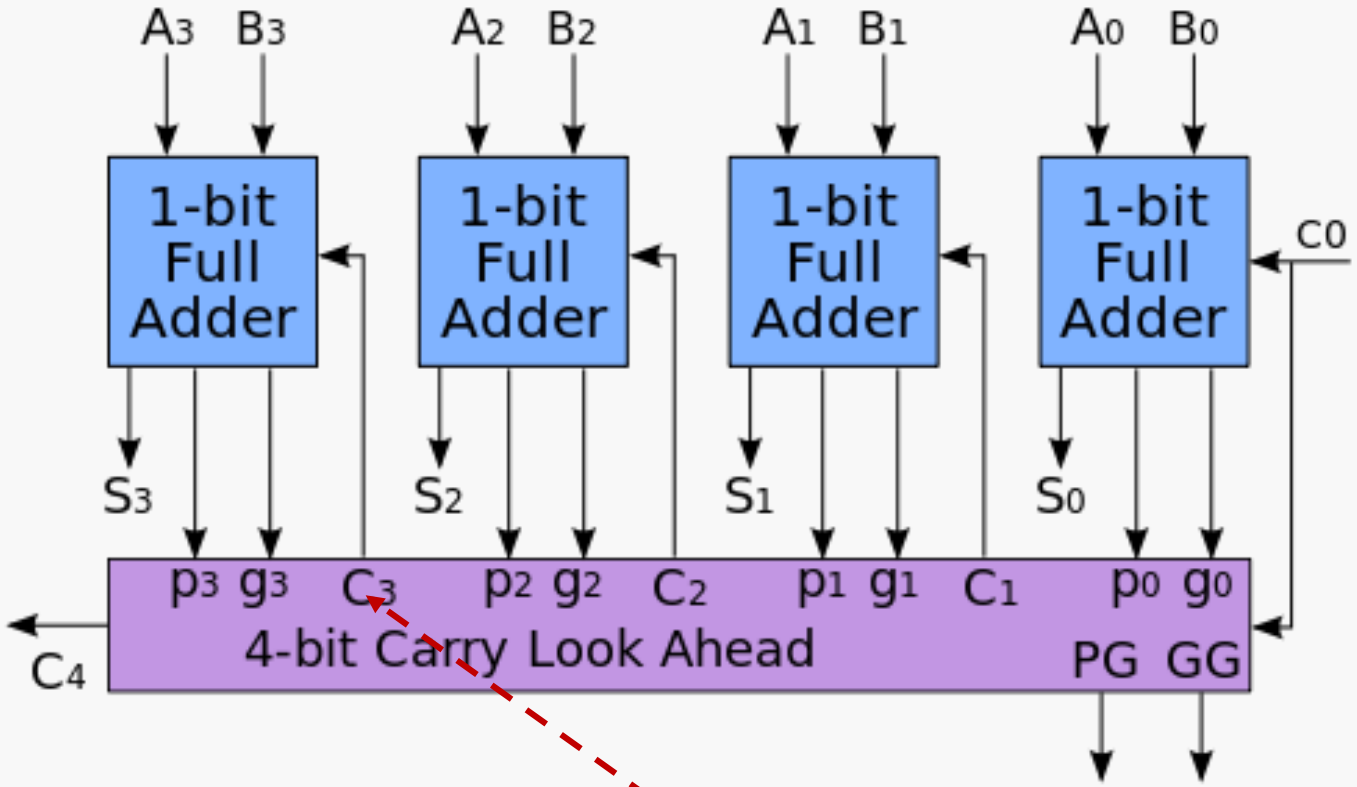
$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

优点：计算Ci+1的延迟时间固定为三级门延迟，与加法器的位数无关

缺点：如果进一步拓宽加法器的位数，则电路变得非常复杂

超前进位加法器（Carry-Lookahead Adder，CLA）





最后一级全加器还需要1级门延迟

总延迟时间为4级门延迟

计算C_3需要3级门延迟

参考值：4-bit行波进位加法器的总延迟时间为9级门延迟

32-bit加法器的实现

如果采用行波进位

- 总延迟时间为65级门延迟

如果采用完全的超前进位

- 理想的总延迟时间为4级门延迟
- 实际上电路过于复杂，难以实现

通常的实现方法

- 采用多个小规模的超前进位加法器拼接而成
- 例如，用4个8-bit的超前进位加法器连接成32-bit加法器

	延迟时间	时钟频率
32-bit RCA	1.3ns	769MHz
单个CLA	0.08ns	/
4级CLA	0.26ns	3.84GHz
注：参照28nm制造工艺，门延迟设为0.02ns		

$$C31=G30+P30\cdot G29+P30\cdot P29\cdot G28+\ldots$$
$$+P30\cdot P29\cdot P28\cdot\ldots\cdot P2\cdot P1\cdot P0\cdot C0$$

需要32输入的与门和或门？！