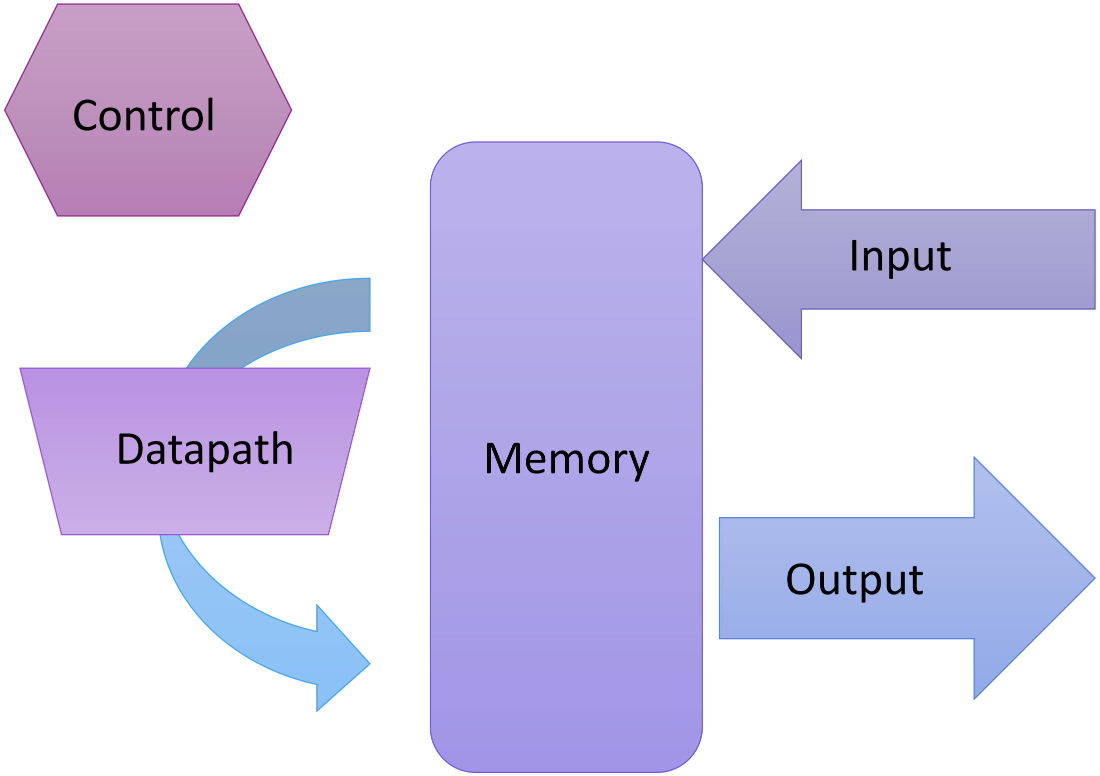
算术运算及电路实现

2022年秋

内容提要

- 运算器功能
- 用于实现运算功能的基础逻辑电路
- ALU设计
- 算术运算的实现

计算机运行机制



Datapath:
完成算术和逻辑运算，通常包括其中的寄存器

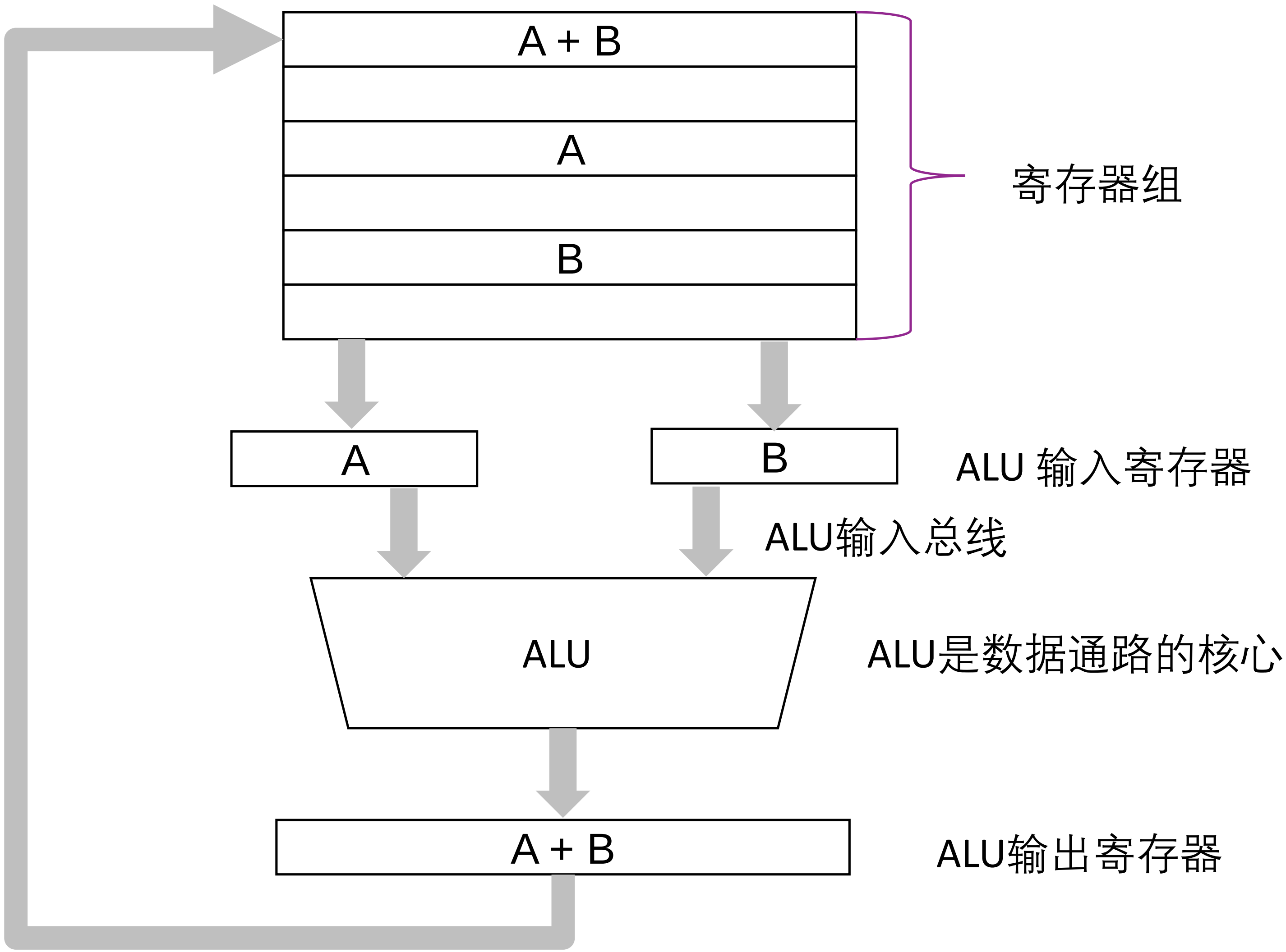
运算器基本功能

- 完成算术、逻辑运算
  - ＋ － × ÷ ∧ ∨ ¬
- 得到运算结果的状态
  - C Z V S
- 取得操作数
  - 寄存器组、数据总线
- 输出、存放运算结果
  - 寄存器组、数据总线
- 暂存运算的中间结果
  - Q寄存器，移位寄存器
- 由控制器产生的控制信号驱动

运算器的基本逻辑电路

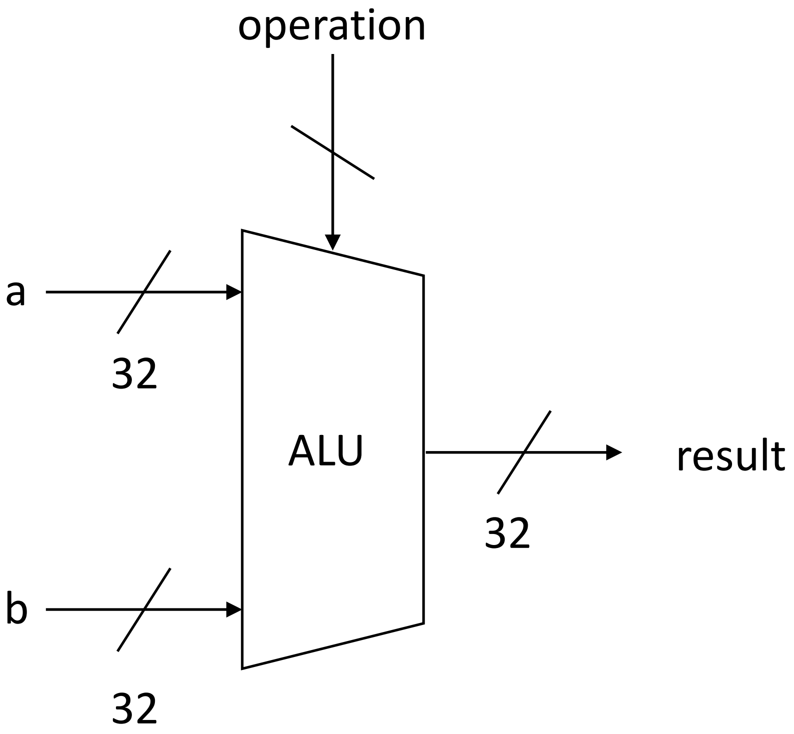
- 逻辑门电路
  - 完成逻辑运算
- 加法器
  - 完成加法运算
- 触发器
  - 保存数据
- 多路选择器、移位器
  - 选择、连通

数据通路（Datapath）



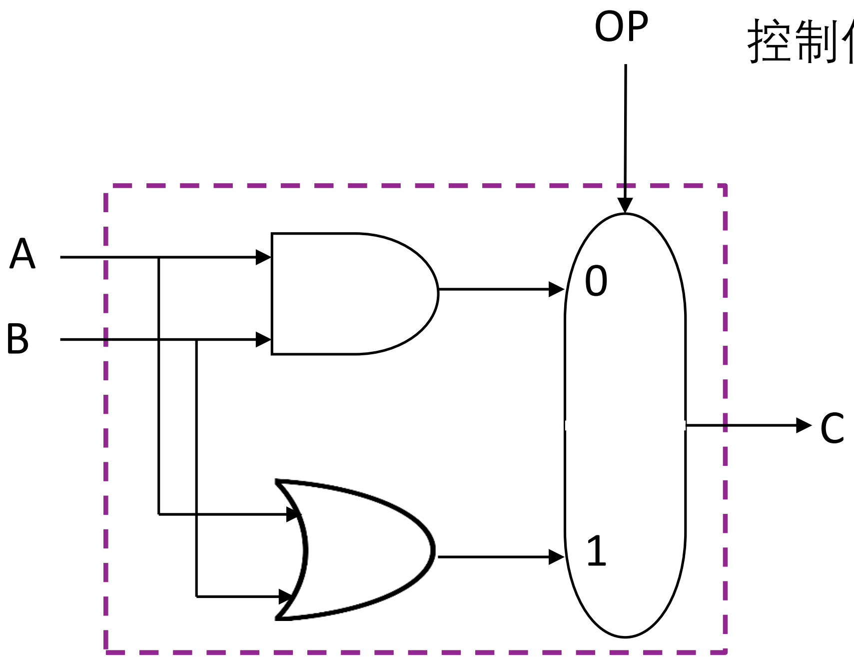
ALU功能和设计

- 功能
  - 对操作数A、B完成算术逻辑运算
  - ADD，AND，OR
- 设计
  - 算术运算
    - 加法器
  - 逻辑运算
    - 与门、或门



1位ALU逻辑运算实现

- 直接用逻辑门实现与和或的功能
- 多路选择器，通过OP控制信号输出结果

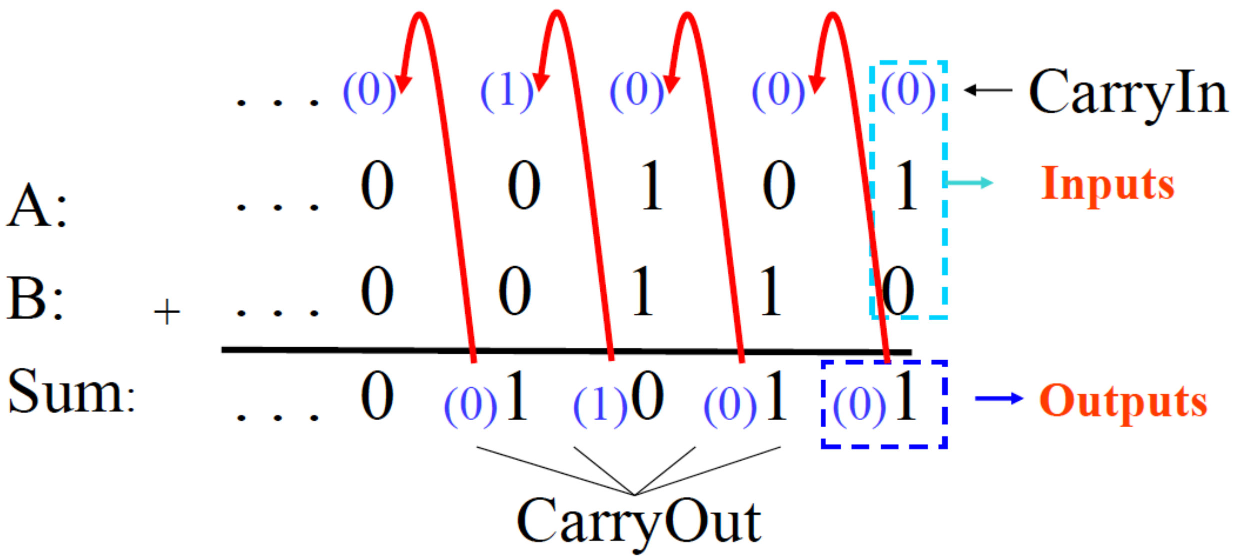


功能表

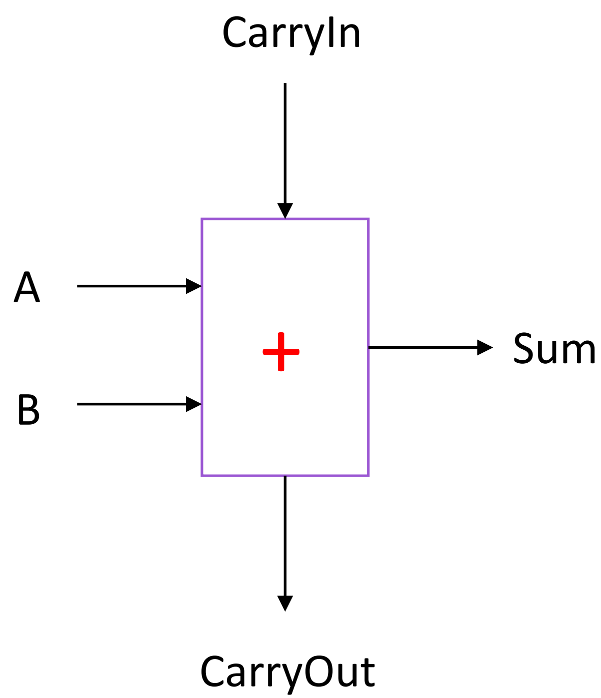
OP	C
0	A AND B
1	A OR B

1位ALU加法运算实现

- 1位的加法：
  - 3个输入信号：A_i, B_i, $CarryIn_i$
  - 2个输出信号：Sum_i, $CarryOut_i$
    - $CarryIn_{i+1} = CarryOut_i$



1位全加器的设计与实现



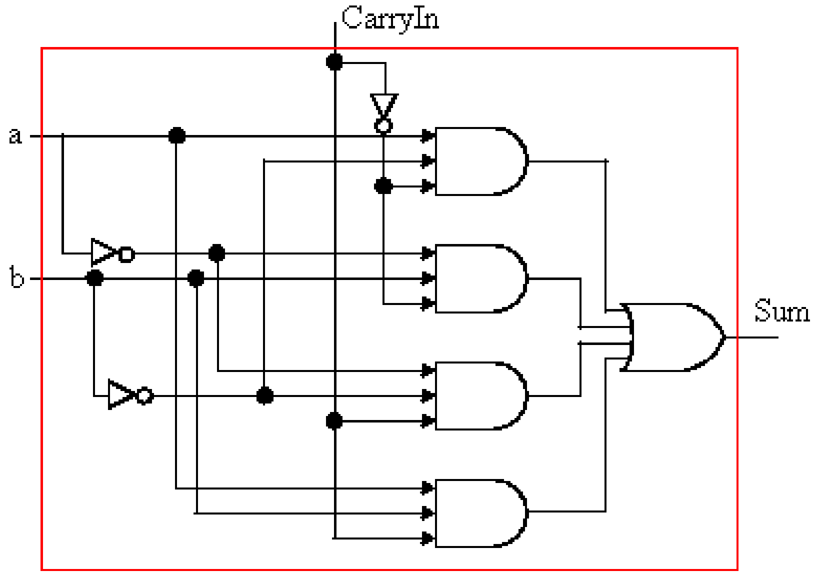
A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

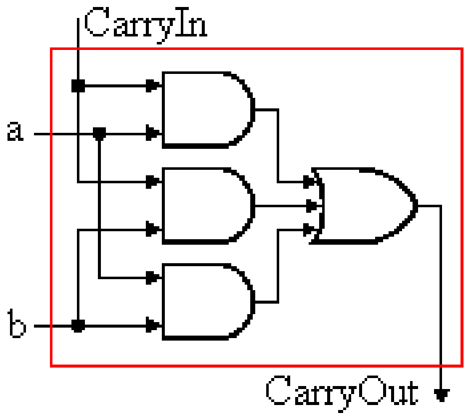
依据真值表可以获得其逻辑表达式，并通过组合逻辑实现

CarryOut = (¬A*B*CarryIn)+(A*¬B*CarryIn)+(A*B*¬CarryIn)+(A*B*CarryIn)
=(B*CarryIn)+(A*CarryIn)+(A*B)
Sum = (¬A*¬B*CarryIn) + (¬A*B*¬CarryIn) + (A*¬B*¬CarryIn) + （A*B*CarryIn)

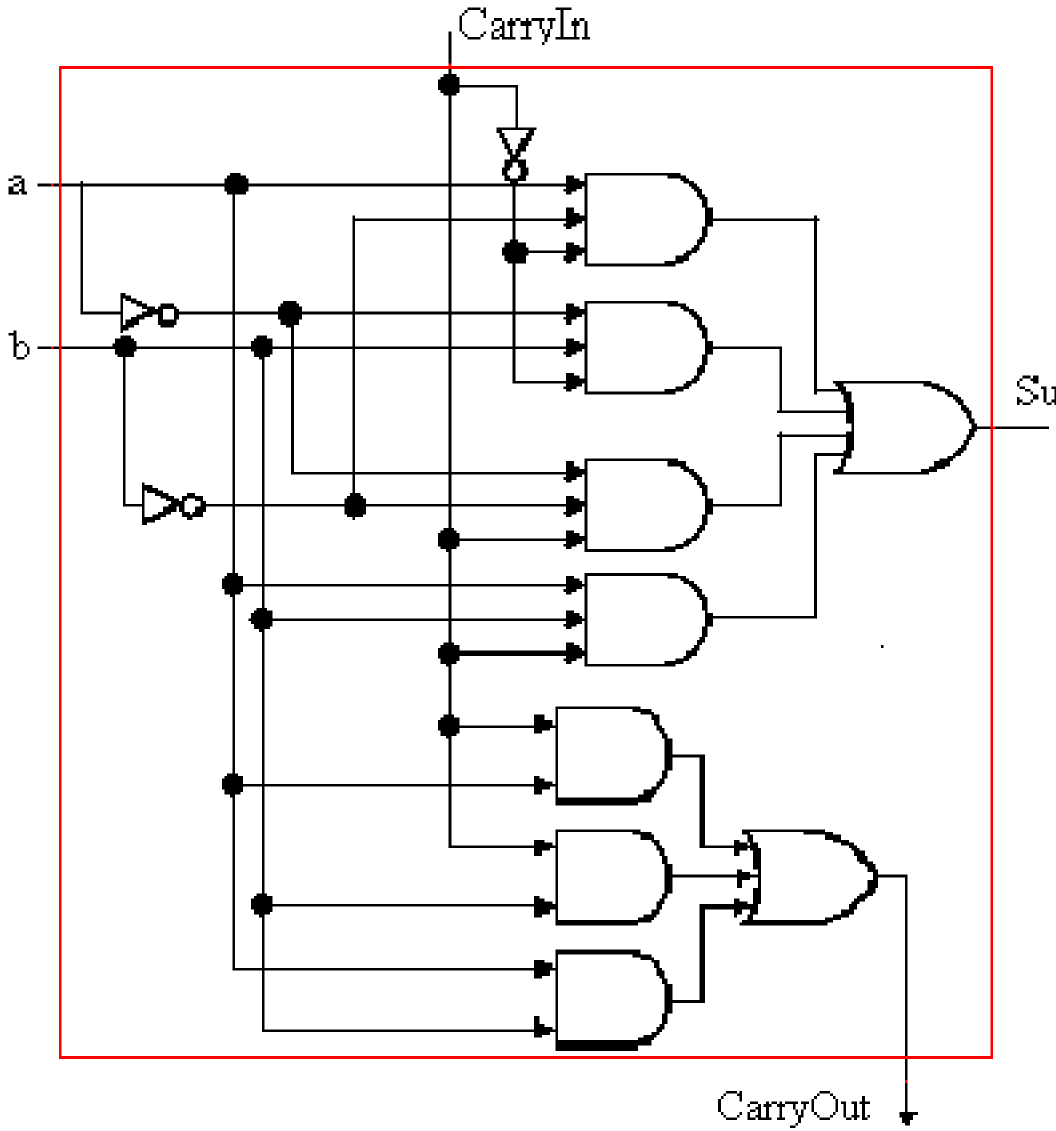
全加器设计与实现

- 用逻辑门实现加法，求Sum
- 用逻辑门求CarryOut
- 将所有相同的输入连接在一起



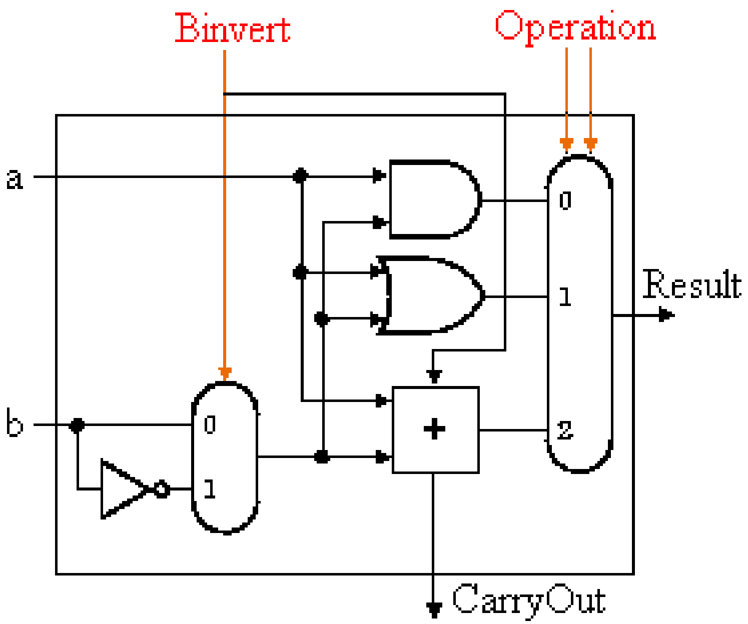


全加器

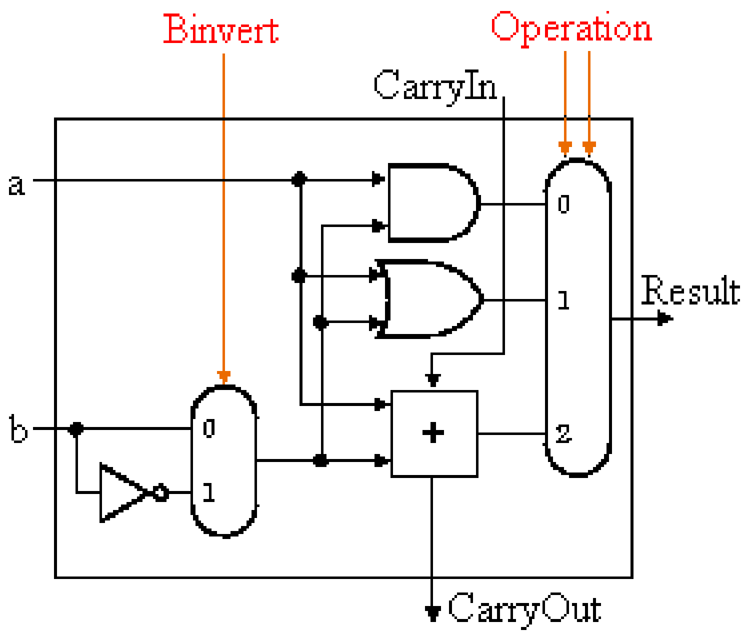


Module adder (
Input a,
Input b,
Input carryin,
Output sum,
output carryout
);
Assign {carryout, sum} = a + b + carryin;
endmodule

1位的ALU



最低位



其它位

1位ALU的设计过程

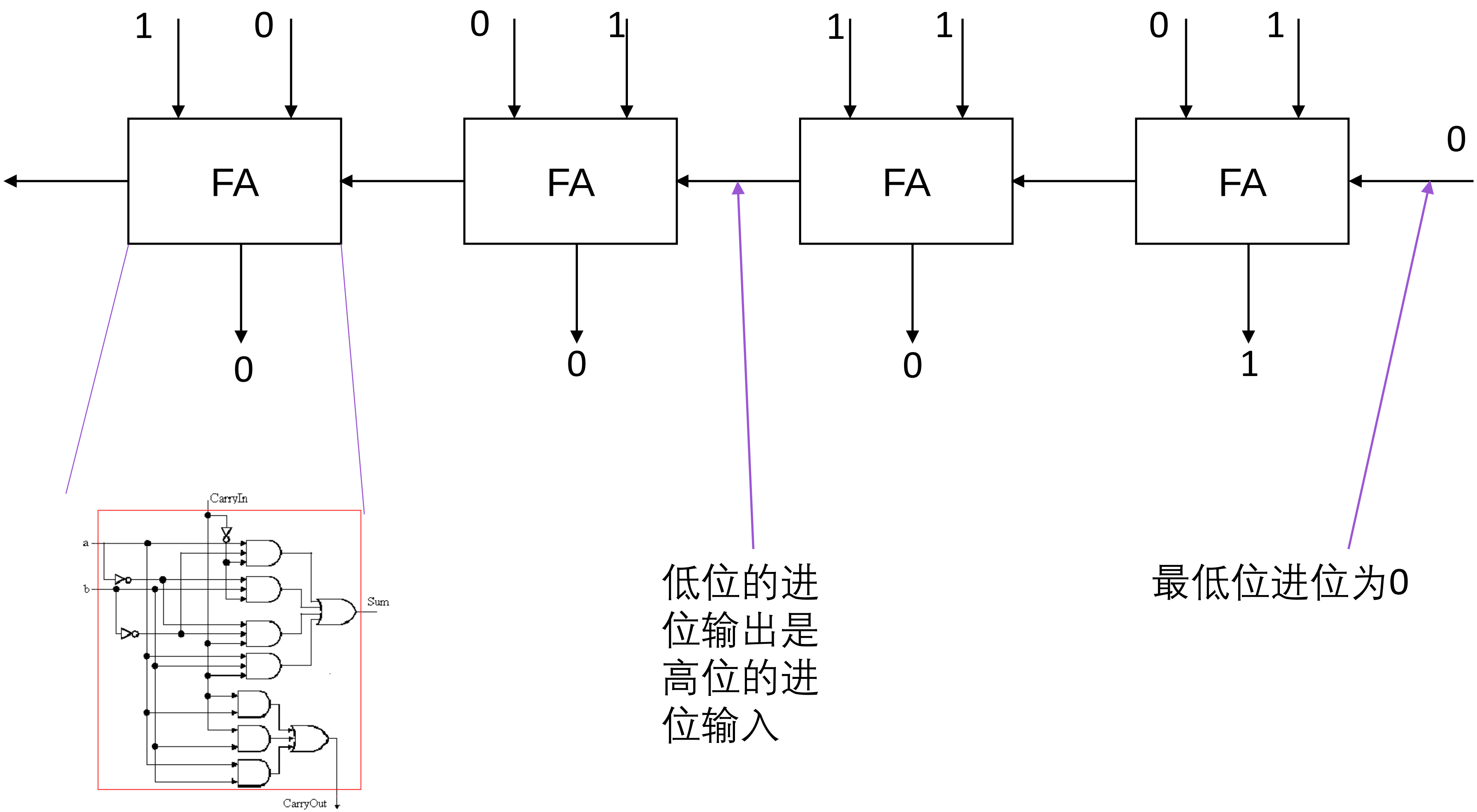
- 确定ALU的功能
- 确定ALU的输入参数
- 根据功能要求得到真值表，获得逻辑表达式
- 依据逻辑表达式实现逻辑电路
- 如何实现4位的ALU呢？

4位ALU实现方法

- 思路1：
  - 同1位ALU设计，写真值表，逻辑表达式，通过逻辑电路实现
- 思路2：
  - 使用1位ALU串联起来，得到4位ALU

$$\begin{array}{ccccc} & 1 & 1 & 0 & 0 \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

4位ALU设计


1
0
0
1
1
1
0
1
FA
FA
FA
FA
0
0
0
0
1
CarryIn
a
b
Sum
CarryOut
低位的进位输出是高位的进位输入
最低位进位为0

超前进位生成

- 如何能提前得到Cout?
- 显然
  - 当a=b=0时，Cout=0
  - 当a=b=1时，Cout=1
  - 当a=1, b=0或a=0, b=1时候，Cout=Cin

$P_i=a_i+b_i$

$G_i=a_i*b_i$

C1=a1b1+(a1+b1)C0=G1+P1C0

C2=a2b2+(a2+b2)C1=G2+P2G1+P2P1C0

C3=a3b3+(a3+b3)C2=G3+P3G2+P3P2G1+P3P2P1C0

C4=a4b4+(a4+b4)C3=……

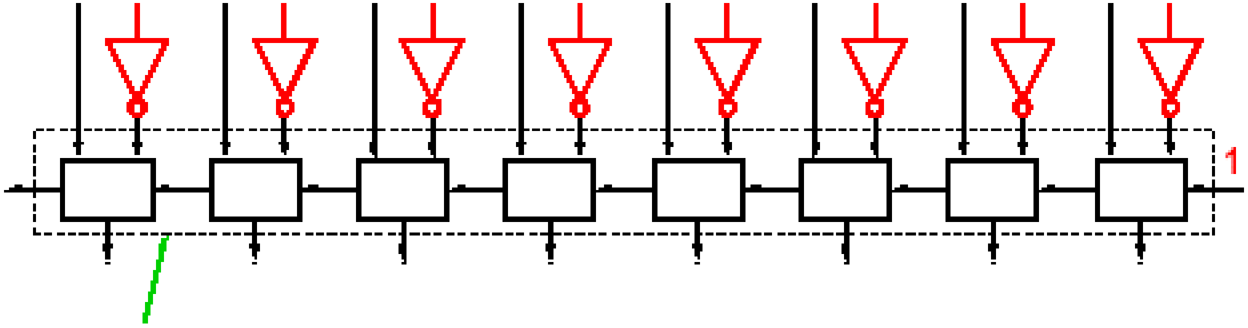
通过单独的进位电路，可以同时得到计算结果和进位

其它的结果标志

- Z=(F1=0)*(F2=0)*(F3=0)*(F4=0)
- S=最高位
- OV=¬F1*¬F2*S+F1*F2*¬S

补码的减法

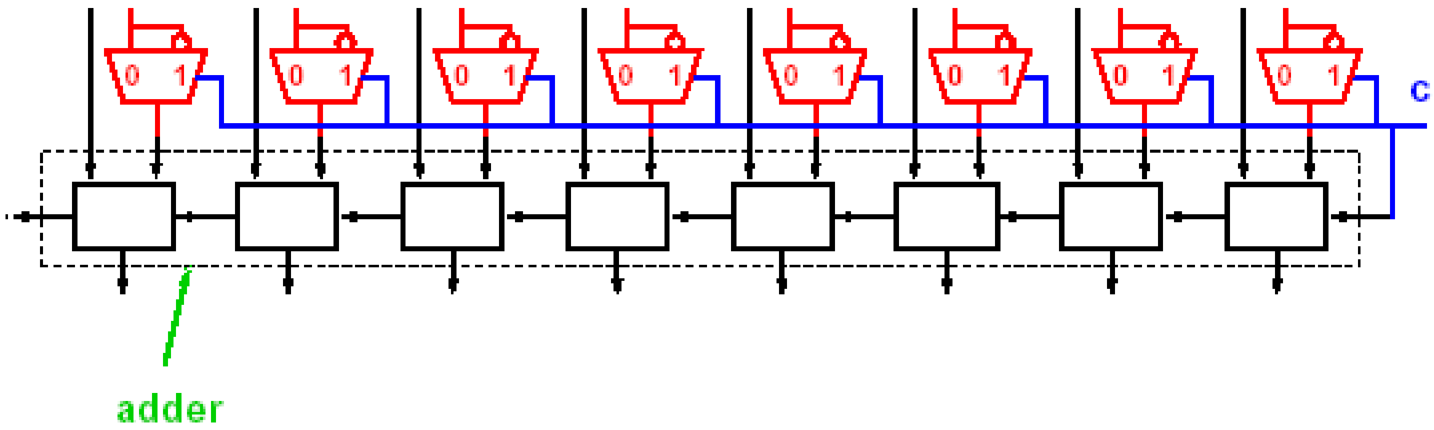
- 根据运算规则：
- $[a-b]_{补}=[a]_{补}+[-b]_{补}$
- $[-b]_{补}$的补码为：将$[b]_{补}$的各位取反，并加1
- 用加法器实现减法



加法器

将加法和减法组合

- 给定控制命令C=0做加法，C=1做减法
- 可以使用选择器来实现



原码乘法

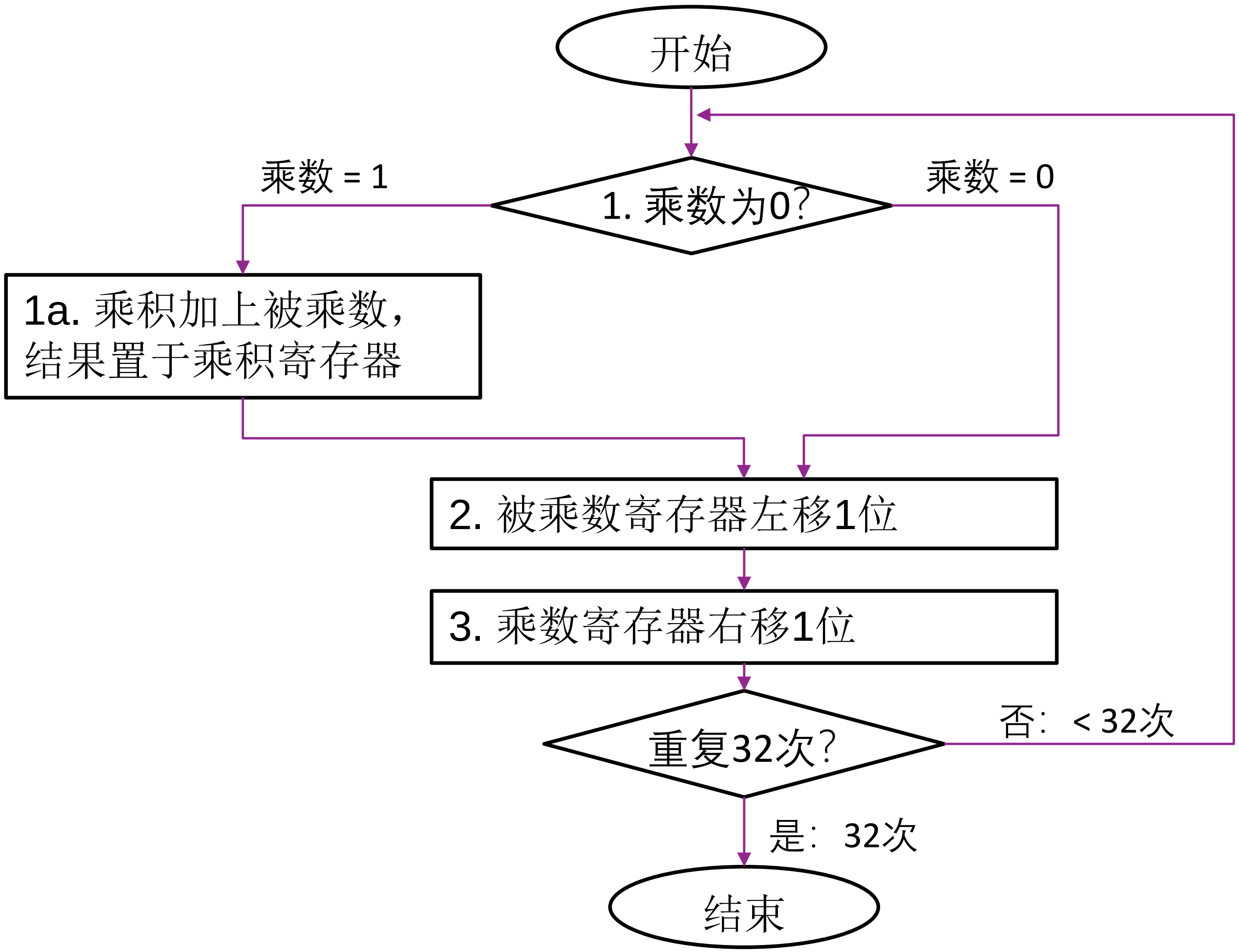
- 从一个简单的例子开始

			1	0	0	0
		x	1	0	0	1
			1	0	0	0
		0	0	0	0	
	0	0	0	0		
1	0	0	0			
1	0	0	1	0	0	0

二进制乘法算法描述

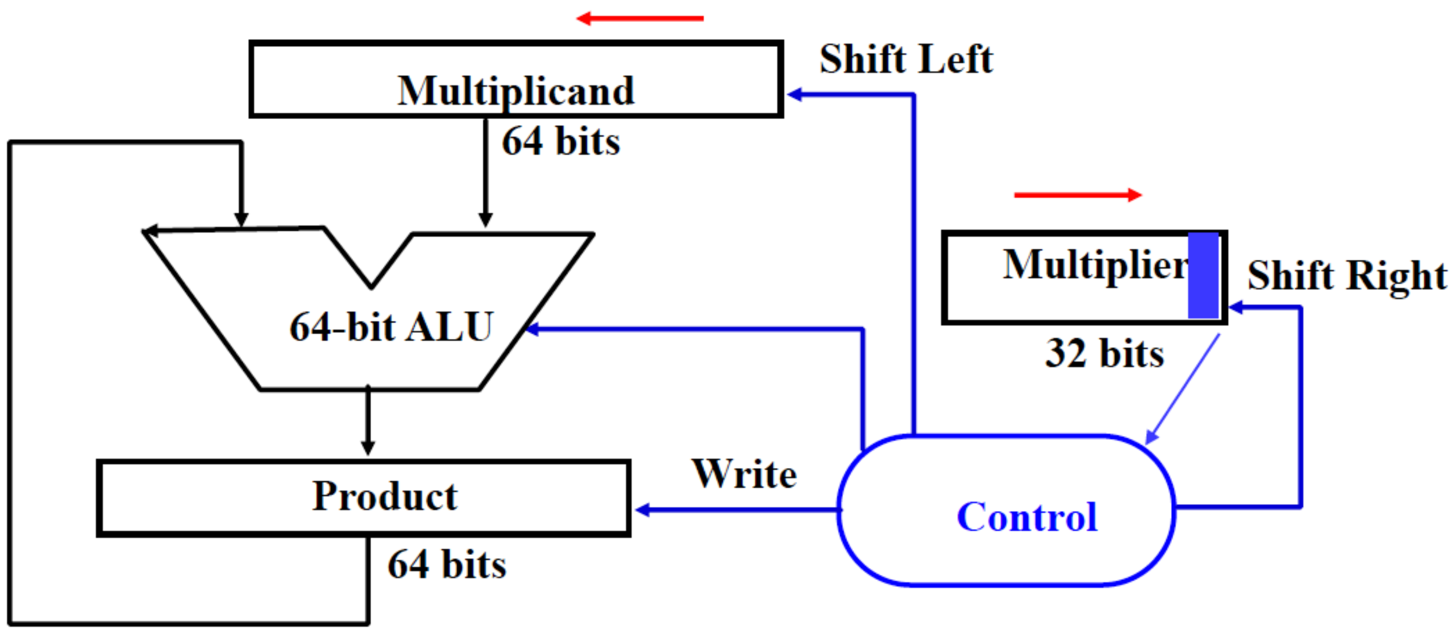
- 基本算法
  - 若乘数的当前位==1，将被乘数和部分积求和
  - 若乘数的当前为==0，则跳过
  - 将部分积移位
  - 所有为都乘完后，部分积即为最终结果
- N位乘数*M位被乘数➔N+M位的积
- 乘法显然比加法更加复杂
  - 但是要比10进制乘法要简单

乘法算法（1）


开始
1. 乘数为0?
乘数 = 1
乘数 = 0
1a. 乘积加上被乘数，
结果置于乘积寄存器
2. 被乘数寄存器左移1位
3. 乘数寄存器右移1位
重复32次?
否：< 32次
是：32次
结束

原码乘法的实现（一）

- 64-位被乘数寄存器，64-位ALU，64-位部分积寄存器，32-位乘数寄存器



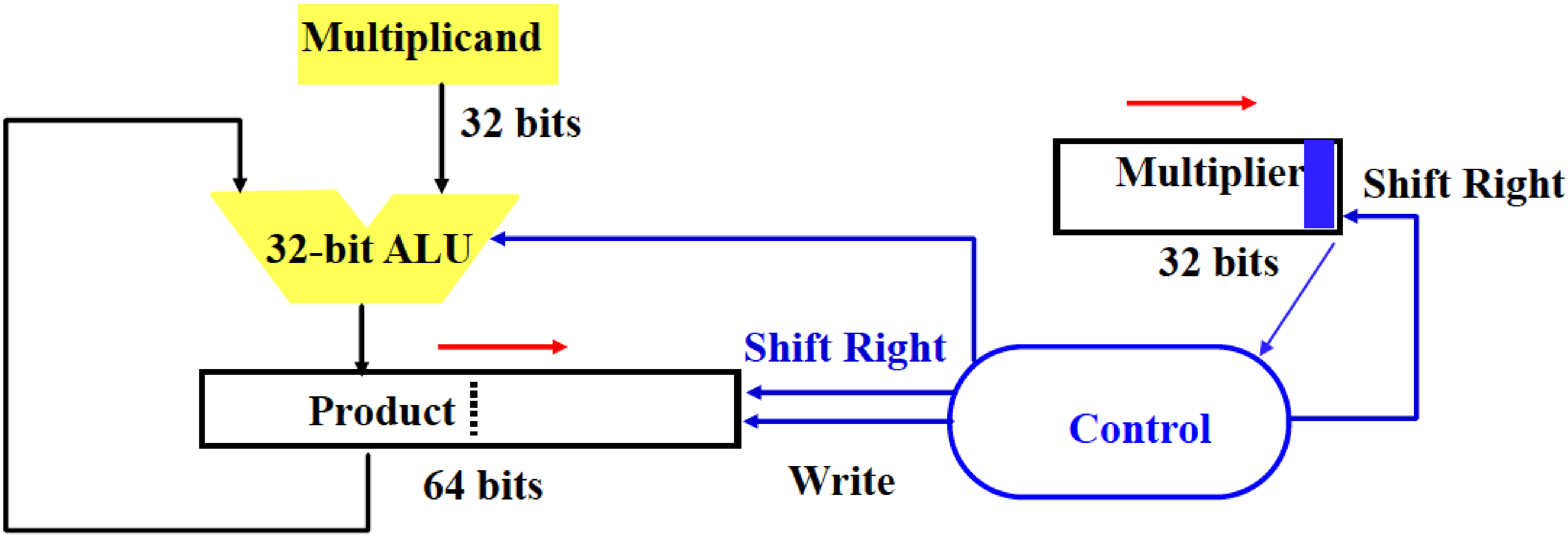
Multiplier = datapath + control

不足

- 实现（一）的不足：
  - 被乘数的一半存储的只是0，浪费存储空间
  - 每次加法实际上只有一半的位有效，浪费了计算能力

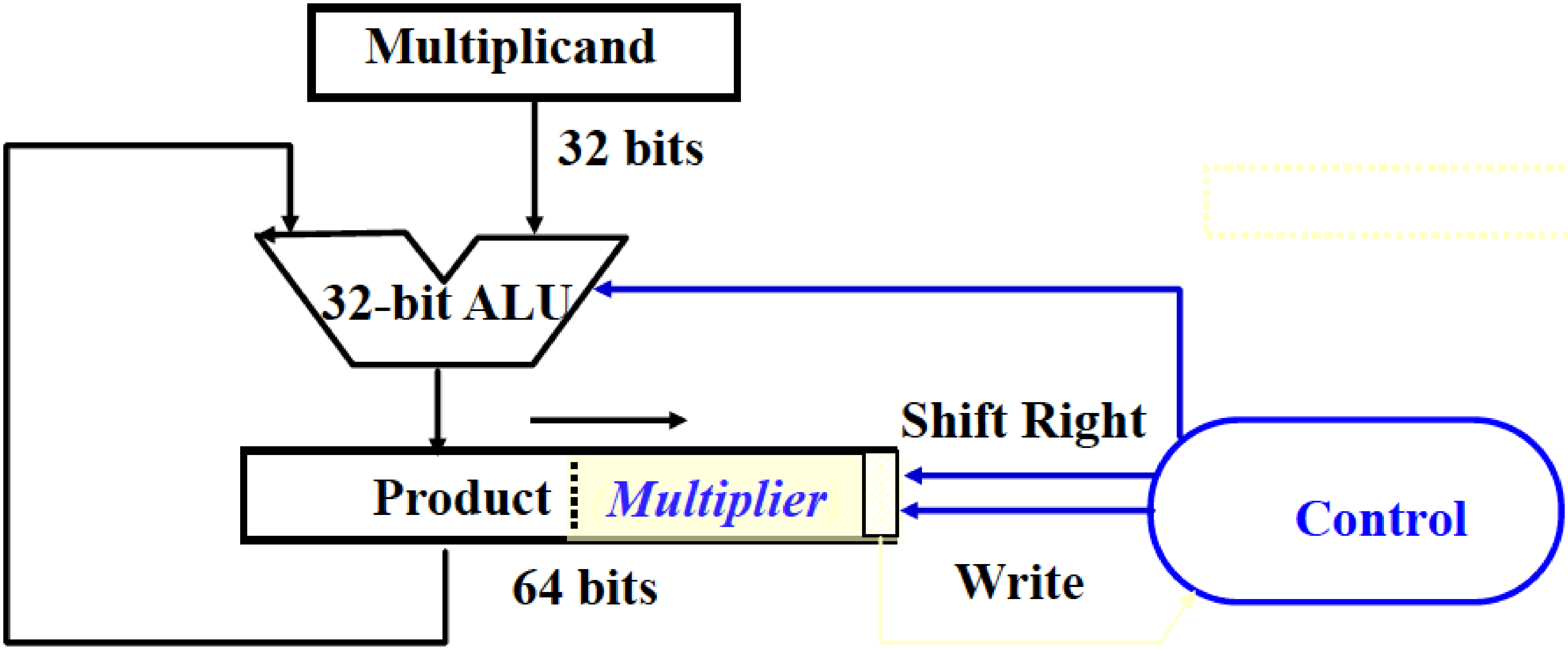
原码的乘法实现（二）

- 32位被乘数寄存器，32位ALU，64位部分积寄存器



原码乘法实现（三）

- 32位被乘数寄存器，32位ALU，64位部分积寄存器（0-位乘数寄存器）



实现（三）的优点

- 实现（二）解决了对加法器位数的浪费
- 需要注意的是，乘数寄存器也存在浪费的情况
  - 把已经完成的乘数位移出，移入的是0
  - 解决这个浪费，可以把乘数和部分积低位结合起来

补码乘法

- 方案一：
  - 将补码转换为原码绝对值，进行原码的正数乘法
  - 依据以下原则得到符号位，并转换回补码表示
    - 同号为正
    - 异号为负
- 方案二：补码直接乘
  - 布斯算法

布斯算法的推导过程

- 布斯算法的原理
  - 虽然乘法是加法的重复，但也可以将它理解成加法和减法的组合。
- 例如：十进制乘法
- 6 x 99 = 6 x 100 – 6 x 1 = 600 – 6 = 594

布斯算法

□ 二进制举例

0111 x 0011 = ?

			0	1	1	1
		x	0	0	1	1
			0	1	1	1
		0	1	1	1	
	0	0	0	0		
0	0	0	0			
0	0	1	0	1	0	1

我们也可以把它看成是下面计算过程:

0 –7*1 + 7*4 = 0 –7 + 28 = 21

补码乘法运算

$$[x]_{补} = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 \geq x \geq -2^n \quad (\bmod 2^{n+1}) \end{cases}$$

若$[x]_{补}=x_{n-1}x_{n-2}\cdots x_1x_0$，则：

$$x = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

同理：$[y]_{补}=y_{n-1}y_{n-2}\cdots y_1y_0$，则：

$$y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} y_i 2^i$$

补码乘法运算

- $[x*y]_补=[x]_补*(-2^{n-1}y_{n-1}+\sum_{i=0}^{n-2}y_i2^i)$
- $=[x]_补*[2^{n-1}(y_{n-2}-y_{n-1})+2^{n-2}(y_{n-3}-y_{n-2})+\cdots+2^0(y_{-1}-y_0)]$
- $= \quad [x]_补*\sum_{i=0}^{n-1}2^i(y_{i-1}-y_i)$ 　　其中：$y_{-1}=0$

- 可直接用补码进行乘法运算
- 根据乘数相邻两位的不同组合，确定是+$[X]_补$或-$[X]_补$

补码乘法运算

- 用Ｙ的值乘［Ｘ］$_{补}$，达到［Ｘ］$_{补}$乘［Ｙ］$_{补}$,求出［Ｘ＊Ｙ］$_{补}$，不必区分符号与数值位。乘数最低一位之后要补初值为０的一位附加线路，并且每次乘运算需要看附加位和最低位两位取值的不同情况决定如何计算部分积，其规则是：
  - ００：＋０
  - ０１：＋被乘数
  - １０：－被乘数
  - １１：＋０

举例：2 X (-5)

步骤	操作	部分积	附加位
0	初始值	0000011011	0
1	−X	1111011011	0
	右移	1111101101	1
2	右移	1111110110	1
3	+X	0000110110	1
	右移	0000011011	0
4	−X	1111011011	0
	右移	1111101101	1
5	右移	1111110110	1

$(1111110110)_2 = -10$

乘法运算：小结

- 与加法比较，需要使用更多的硬件来实现，也更复杂
- 若使用简单的方法来实现，则需要多个计算周期
- 仅仅介绍了乘法运算的一些“皮毛”：有许多提升和优化的空间

除法运算

- 在计算机内实现除运算时，存在与乘法运算类似的几个问题：加法器与寄存器的配合，被除数位数更长，商要一位一位地计算出来等。这可以用左移余数得到解决，且被除数的低位部分可以与最终的商合用同一个寄存器，余数与上商同时左移。
- 除法可以用原码或补码计算，都比较方便，也有一次求多位商的快速除法方案，还可以用快速乘法器完成快速除法运算。

原码一位除运算

$$[Y/X]_{原} = (X_S \oplus Y_S)(|Y| / |X|)$$

- 原码一位除是指用原码表示的数相除，求出原码表示的商。除操作的过程中，每次求出一位商。
- 恢复余数法：被除数-除数，若结果>=0，则上商1，移位；若结果<0，则商0，恢复余数后，再移位；
- 求下一位商；
- 但计算机内从来不用这种办法，而是直接用求得的负余数求下一位商。

原码一位除运算

$$[X/Y]_{原} = (X \oplus Y)(|X| / |Y|)$$

例如： $X = 0.1011$ $Y = -0.1101$

```
             0.1101
       ┌───────────
0.1101 │ 0.10110
           01101
         ─────────
            10010
            01101
          ─────────
             10100
              1101
           ─────────
              0111
```

被除数（余数）	商	
00 1011	00000	初态
01 0110	0000	第 1 次
01 0010	0001	第 2 次
00 1010	0011	第 3 次
01 0100	0110	第 4 次
00 0111	01101	第 5 次

X 和 Y 符号异或为负

最终商原码表示为：

1 1101

余数为： $0.0111 * 2^{-4}$

加减交替除法原理证明

1. 若第**i -1**次求商，减运算的余数为**+ R_{i-1}**，商**1**，余数左移**1**位得**2R_{i-1}**。

2. 则下一步第i 次求商$R_i = 2R_{i-1} - Y$, 若 $R_i \geq 0$，…

若$R_i < 0$, 商0

恢复余数为正且左移1位得$2(R_i + Y)$

3. 则再下一步第**i+ 1**次求商**$R_{i+1} = 2(R_i + Y) - Y$**

$= 2R_i + Y$

- 公式表明, 若上次减运算结果为负, 可直接左移, 本次用**+Y** 求余即可; 减运算结果为正，用**-Y** 求余

加减交替除法

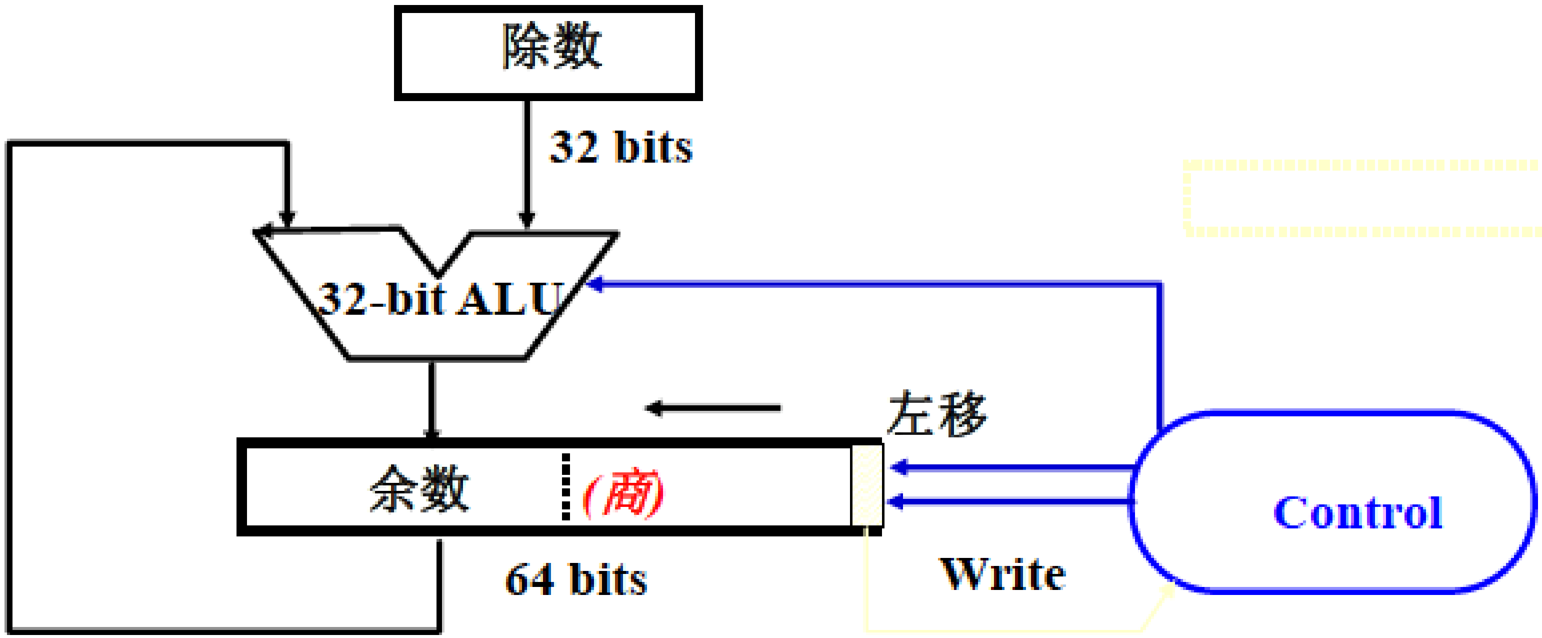
被除数(余数)	商	
0 0 1 0 1 1	0 0 0 0 0	开始情形
+) 1 1 0 0 1 1		-Y
1 1 1 1 1 0	0 0 0 0 0	<0, 商0
1 1 1 1 0 0	0 0 0 0 0	左移1位
+) 0 0 1 1 0 1		+Y
0 0 1 0 0 1	0 0 0 0 1	>0, 商1
0 1 0 0 10	0 0 0 1 0	左移1位
+) 1 1 0 0 1 1		-Y
0 0 0 1 0 1	0 0 0 1 1	>0, 商1
0 0 1 0 1 0	0 0 1 1 0	左移1位
+) 1 1 0 0 1 1		-Y
1 1 1 1 0 1	0 0 1 1 0	<0, 商0
1 1 1 0 1 0	0 1 1 0 0	左移1位
+) 0 0 1 1 01		+Y
0 0 0 1 1 1	0 1 1 0 1	>0, 商1

补码除法运算

- 补码除法与原码除法很类似，差别仅在于:
  - 被除数与除数为补码表示，
  - 直接用补码除,求出反码商，
  - 再修正为近似的补码商.
- 实现中，求第一位商要判２数符号的同异，同号，作减法运算，异号,则作加运算;
- 上商，余数与除数同号，商１，作减求下位商，
- 余数与除数异号，商０，作加求下位商;
- 商的修正：多求一位后舍入，或最低位恒置１

除法的实现

- 32-位除数寄存器，32 -位ALU，64-位余数（被除数）寄存器



小结

- ALU的基本功能：算术、逻辑运算
- 1位ALU：最基本的功能：加法、与、或
- 位数扩展：快速进位
- 功能扩展：减法、乘法、除法

阅读与思考

- 阅读
- 思考
  - ALU的最大延迟如何估算？
  - 运算器其他功能如何实现？

谢谢