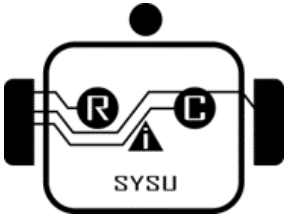
中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

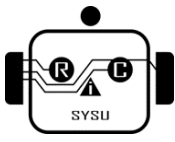
第三章：计算机中的运算

中山大学计算机学院
陈刚

2022年秋季

本讲内容

- 基本运算
  - 加法和减法
  - 位操作
- 乘法运算
- 除法运算
- 浮点运算



从十进制乘法谈起

- 十进制乘法，例：

				2	4	5
	X			6	7	3
$M_0=AXB_3$				7	3	5
$M_1=AXB_2$		1	7	1	5	
$M_2=AXB_1$	1	4	7	0		
AXB	1	6	4	8	8	5

位积 $A \times B_i$

AXB = 164885

由十进制乘法到二进制乘法

- 二进制乘法，例：

逻辑与

				1	1	0	1	
			X	1	0	1	1	
$M_0=A\&B_4$				1	1	0	1	
$M_1=A\&B_3$			1	1	0	1		
$M_2=A\&B_2$		0	0	0	0			
$M_3=A\&B_1$	1	1	0	1				
AXB	1	0	0	0	1	1	1	1

位积 $A \times B_i$

AXB = 10001111

二进制乘法

计算机中怎么实现?

					1	1	0	1
			X		1	0	1	1
$M_0=A\&B_4$	0	0	0	0	1	1	0	1
$M_1=A\&B_3$	0	0	0	1	1	0	1	0
$M_2=A\&B_2$	0	0	0	0	0	0	0	0
$M_3=A\&B_1$	0	1	1	0	1	0	0	0
AXB	1	0	0	0	1	1	1	1

手工运算过程

位积 $A \times B_i$

AXB = 10001111

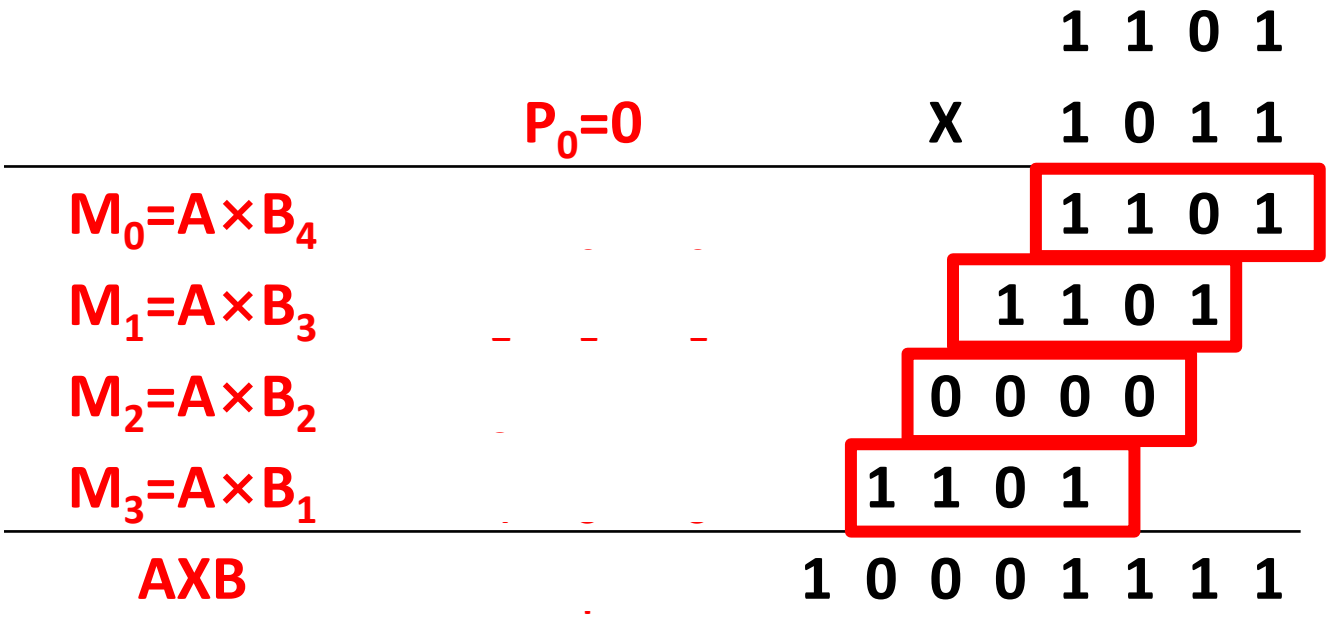
问题:

1.加法器只有两个输入端，无法支持多路输入!

2.需要2n+1位加法器，不能有效利用全加器操作!

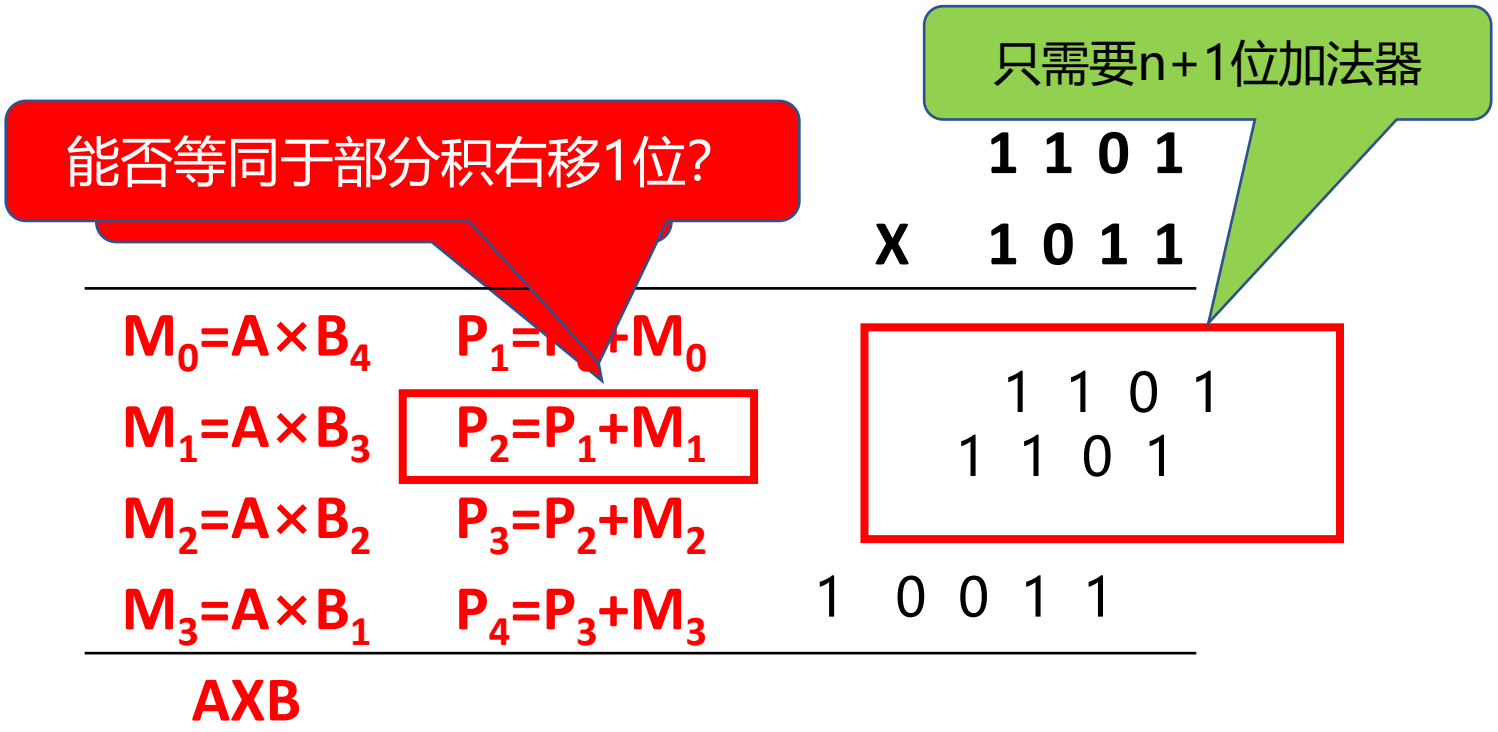
二进制乘法

加法器只需要两个输入端分别输入部分积Pi以及位积Mi，避免了需要加法器拥有多个输入端的问题



改进方案1：

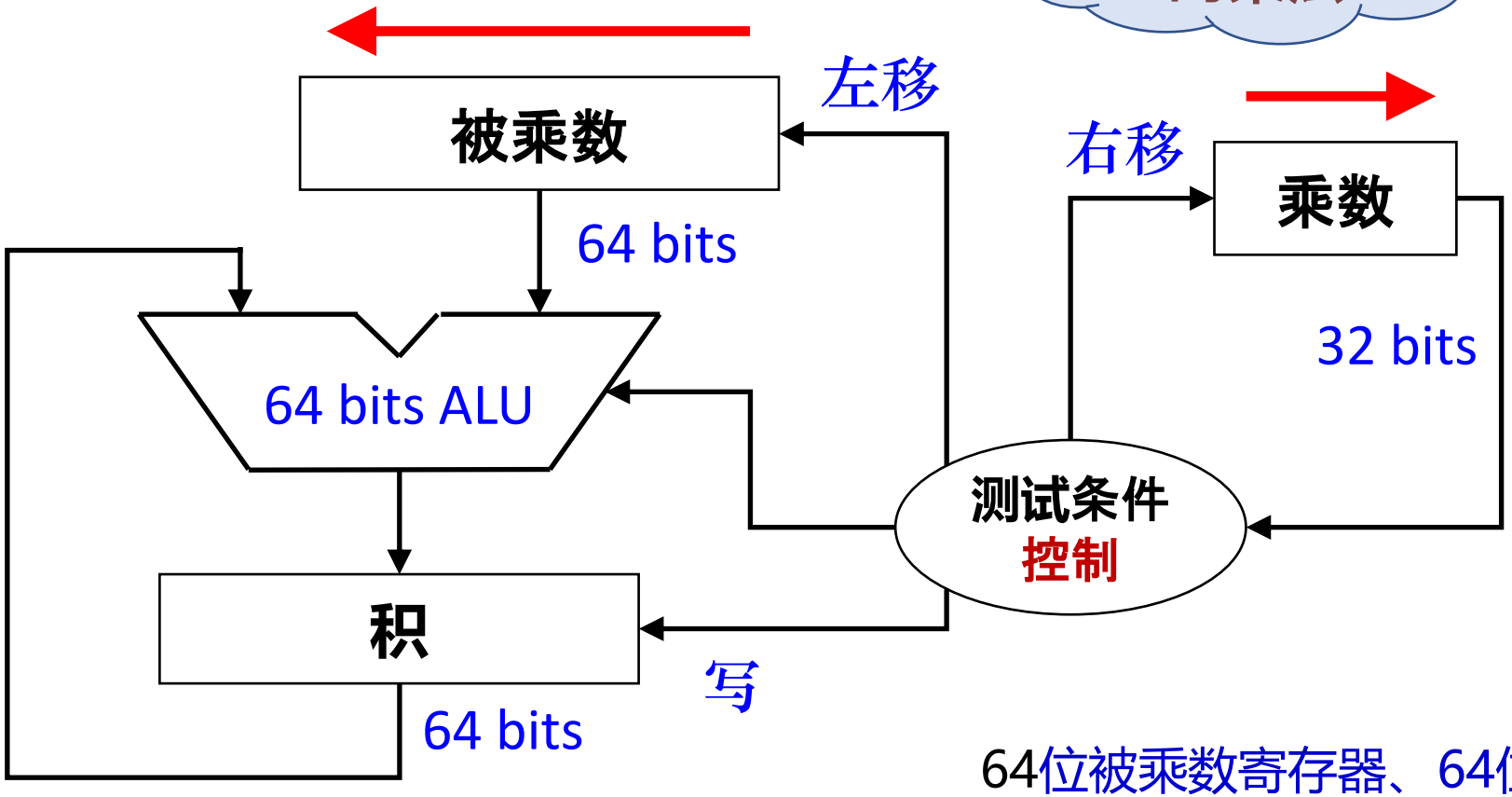
二进制乘法


只需要n+1位加法器
能否等同于部分积右移1位?
1 1 0 1
X 1 0 1 1
M0=A×B4 P1=P0+M0
M1=A×B3 P2=P1+M1
M2=A×B2 P3=P2+M2
M3=A×B1 P4=P3+M3
AXB
1 1 0 1
1 1 0 1
1 0 0 1 1

改进方案2:

第一种无符号移位-加法乘法器

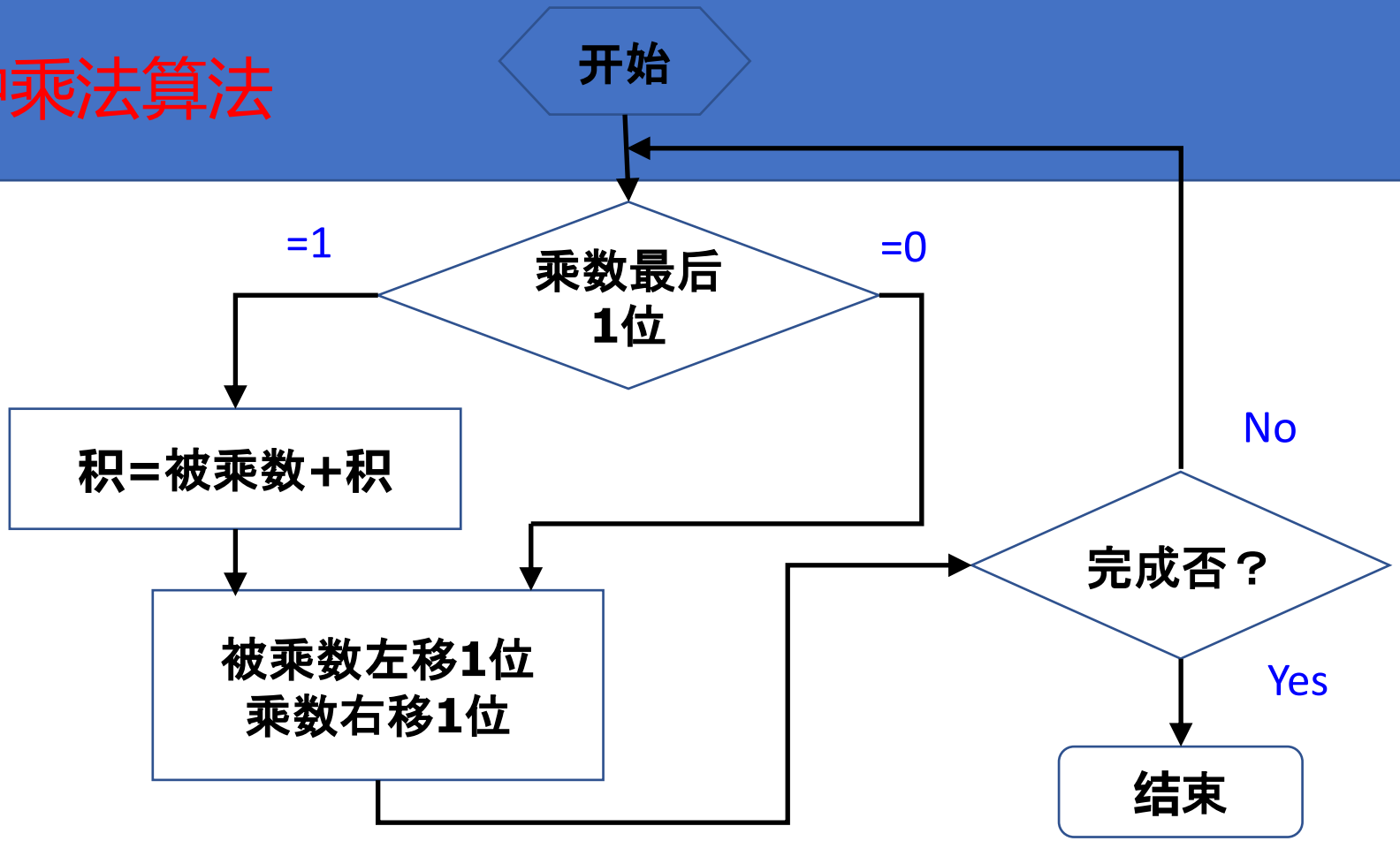
先了解正数的乘法



64位被乘数寄存器、64位ALU、64位乘积寄存器、32位乘数寄存器

乘法器 = 数据通路 + 控制

第一种乘法算法



- Requires 32 iterations （ Addition Shift Comparison ）
- Almost 100 cycles
- Very big, Too slow!

Multiply example using algorithm in Figure 3.5.

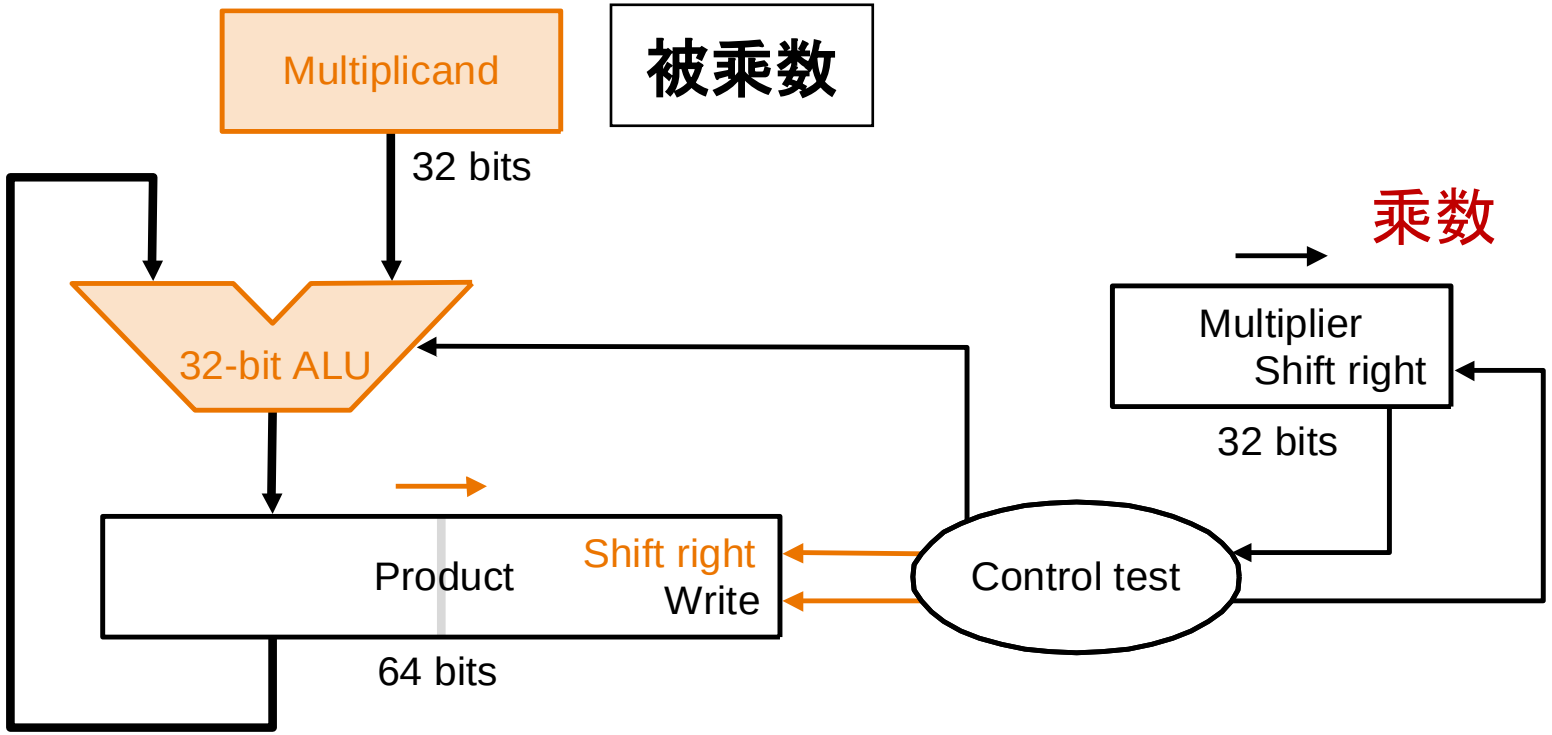
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

第一种乘法的启示

- 每个周期 1个时钟 => 每次乘法大约100（32 3）个时钟
  - 乘法与加法的出现频率比 5:1 ~ 100:1
- 64位被乘数中 1/2的位数 总是为 0 => 使用64位加法器，太浪费 !!!
- 用乘积右移 替代 被乘数左移?

Multiplier V2-- Logic Diagram

- Diagram of the V2 multiplier
- Only **left half of product register is changed**



32位被乘数寄存器、32位ALU、64位乘积寄存器、32位乘数寄存器

Example for second version

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial	1011	0010	0000 0000
1	Test true shift right	1011 0101	0010	0010 0000 0001 0000
2	Test true shift right	0101 0010	0010	0011 0000 0001 1000
3	Test false shift right	0010 0001	0010	0001 1000 0000 1100
4	Test true shift right	0001 0000	0010	0010 1100 0001 0110

Multiplier V 3

乘积寄存器浪费的空间 正好与 乘数中无用的空间 相同
=> 联合使用 乘数寄存器 和 乘积寄存器

- Further optimization
- At the initial state the product register contains only '0'
- The lower 32 bits are simply shifted out
- Idea:

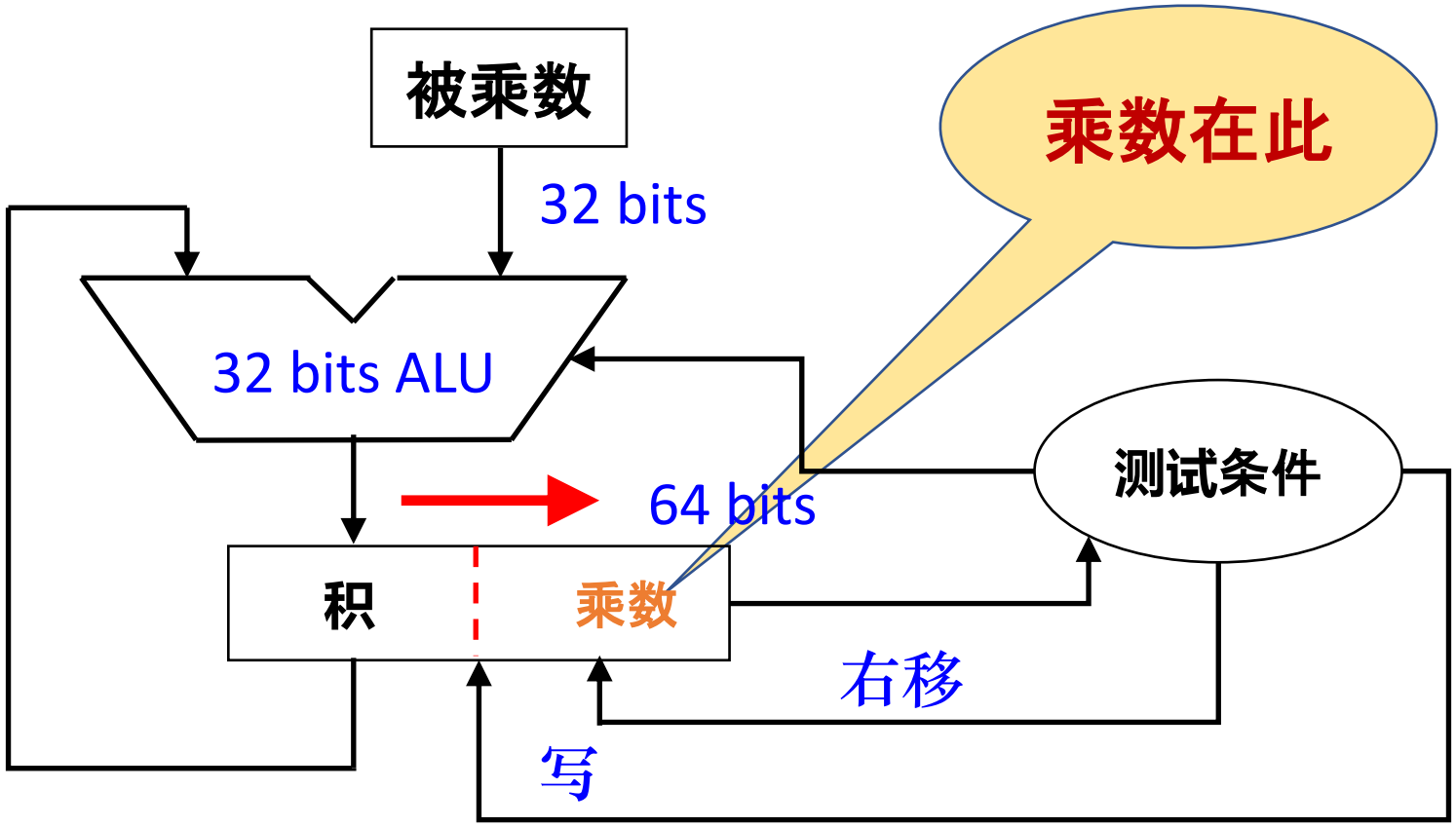
  use these lower 32 bits for the multiplier

0	0	0	1	0	0	0	0	
0	0	0	1	1	0	0	0	0
0	0	0	1	1	1	0	0	0 0
0	0	0	0	1	1	1	0	0 0 0
0	0	0	0	0	1	1	1	0 0 0 0

multiplier

改进的乘法实现硬件

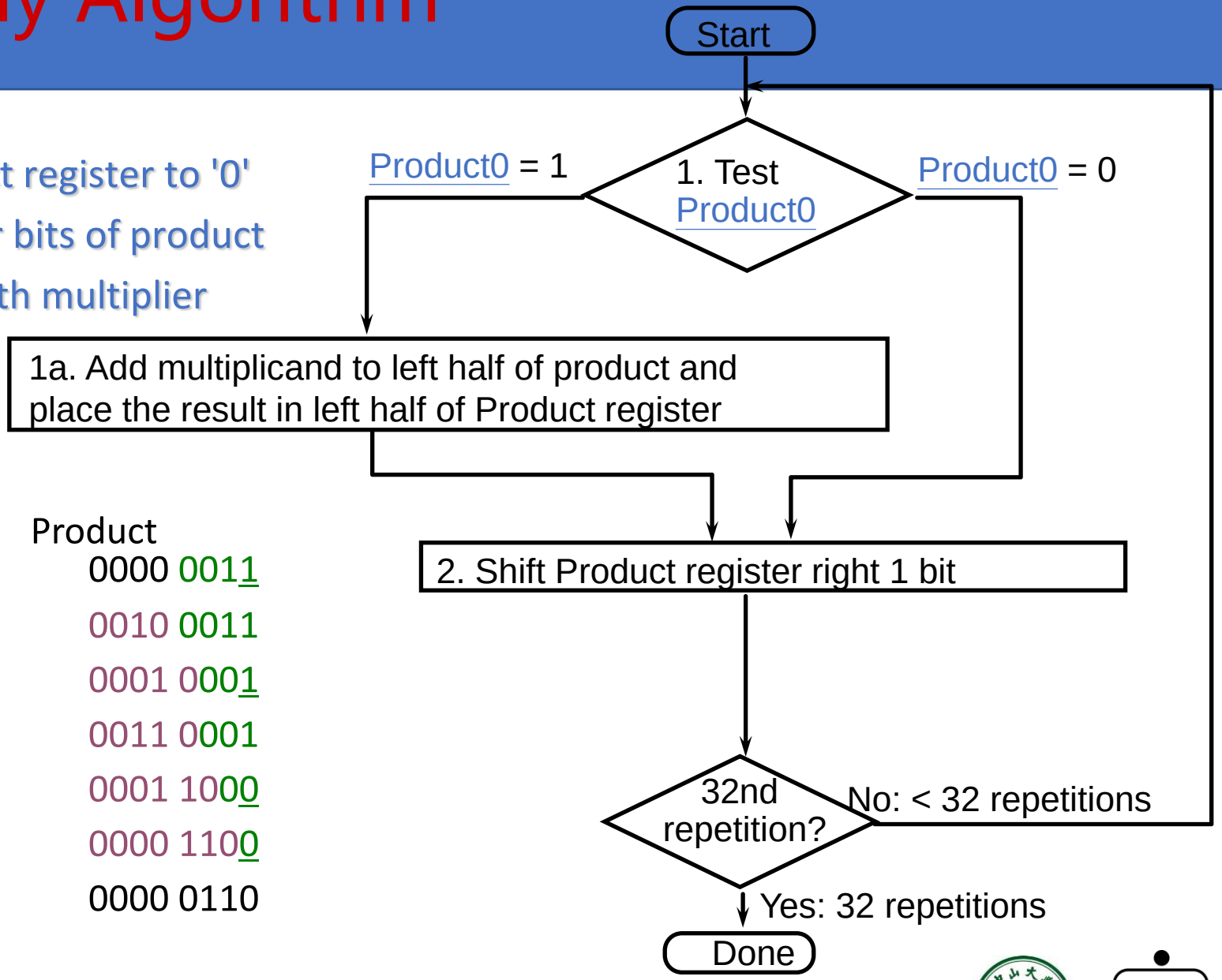
第三种乘法硬件



测试乘数最右边的位，当乘数为1时，将乘数和被乘数移位，同时将被乘数与积相加

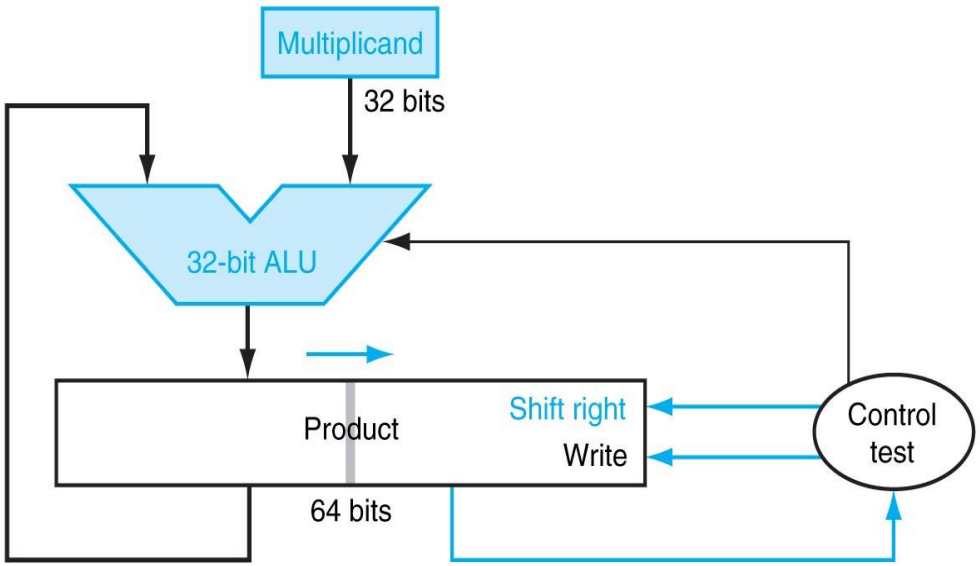
Multiply Algorithm (Ver. 3)

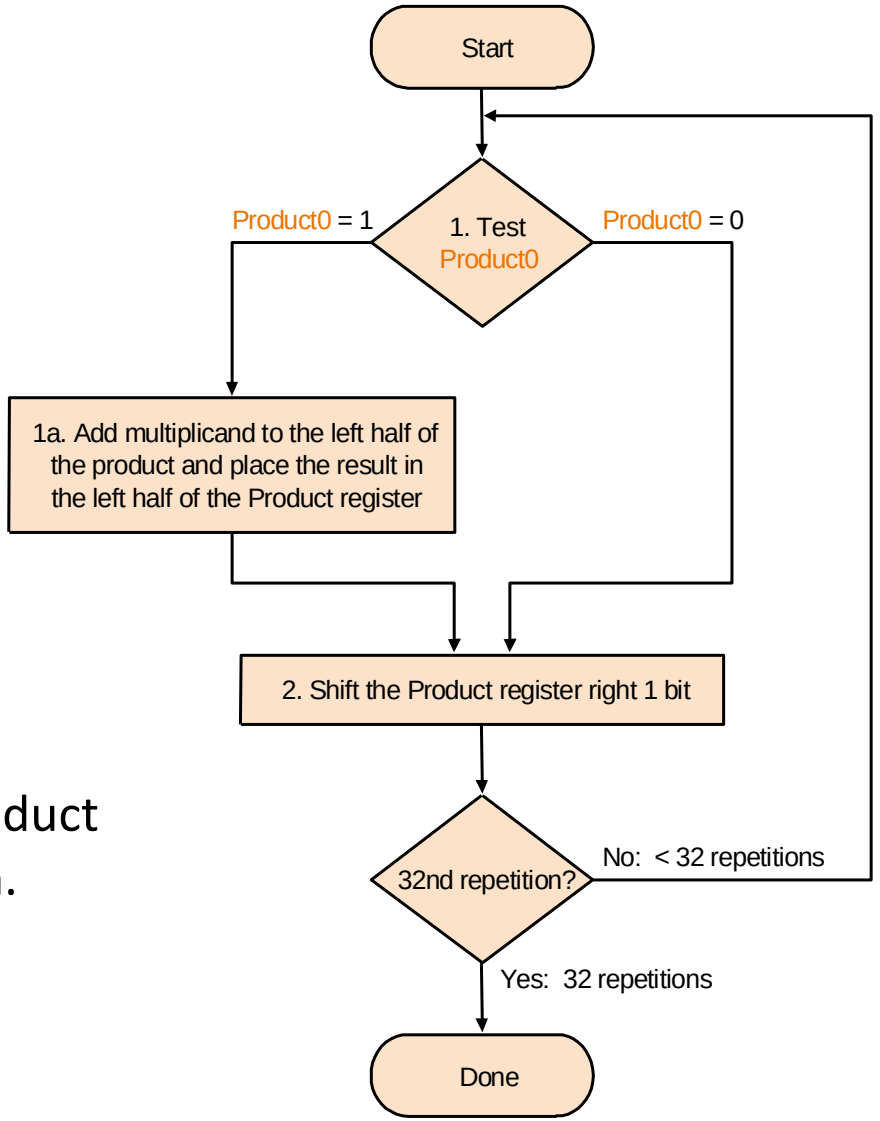
- Set product register to '0'
- Load lower bits of product register with multiplier



Multiplicand	Product
0010	0000 0011
	0010 0011
0010	0001 0001
	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110

Final Version





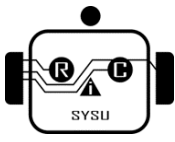
The trick is to use the lower half of the product to hold the multiplier during the operation.

Even More Efficient
Hardware Implementation!

Example with V3

- Multiplicand x multiplier: 0001 x 0111

		Shift out	
Multiplicand:	**0001**		
Multiplier:×	**0111**		
	00000111		#Initial value for the product
1	**00010111**		#After adding 0001, Multiplier=1
	00001011	**1**	#After shifting right the product one bit
	0001		
2	**00011011**		#After adding 0001, Multiplier=1
	00001101	**1**	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	**00011101**		
	00001110	**1**	#After shifting right the product one bit
	0000		
4	**00001110**		#After adding 0001, Multiplier=0
	00000111	**0**	#After shifting right the product one bit



Signed Multiplication

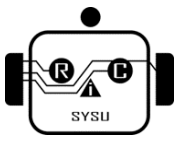
有符号数乘法如何?

- 简单地策略是 假设两个源操作数都是正数, 在运算结束时再对乘积进行修正 (不算符号位, 运算31步)
- 使用补码

-需要对部分乘积进行符号扩展，在最后再进行减法

Better Version:

- Use the unsigned multiplication hardware
- When shifting right, extend the sign of the product
- If multiplier is negative, the last step should be a subtract



Signed Multiplication (Pencil & Paper)

$$v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

Case 1: Positive Multiplier

Multiplicand $1100_2 = -4$

Multiplier $\times 0101_2 = +5$

Sign-extension: 11111100

111100

Product $11101100_2 = -20$

If multiplier is negative, the last step should be a subtract

Case 2: Negative Multiplier

Multiplicand $1100_2 = -4$

Multiplier $\times$ $1101_2 = -3$

11111100

111100

Sign-extension: 0100 (2's complement of 1100)

Product $00001100_2 = +12$

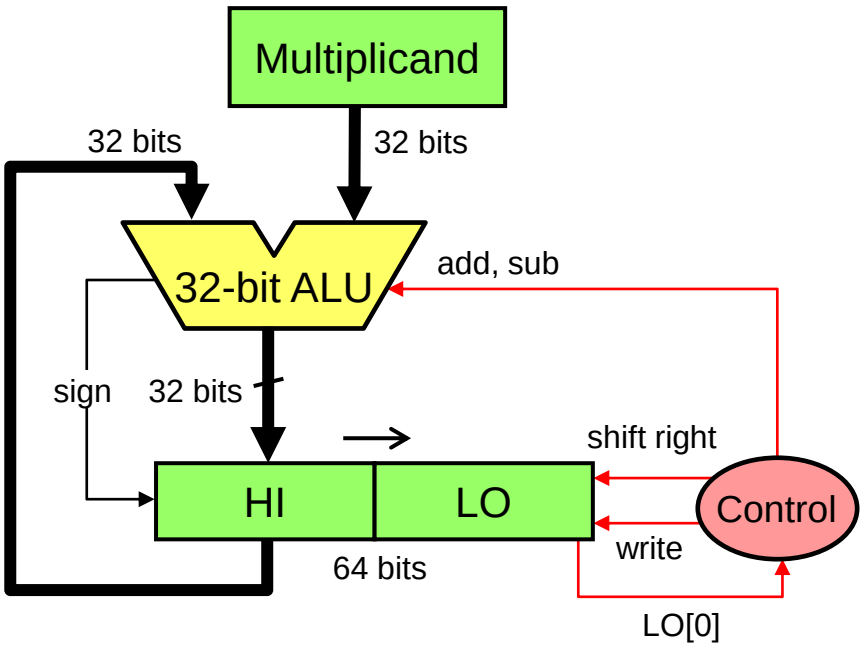
（1100）补码

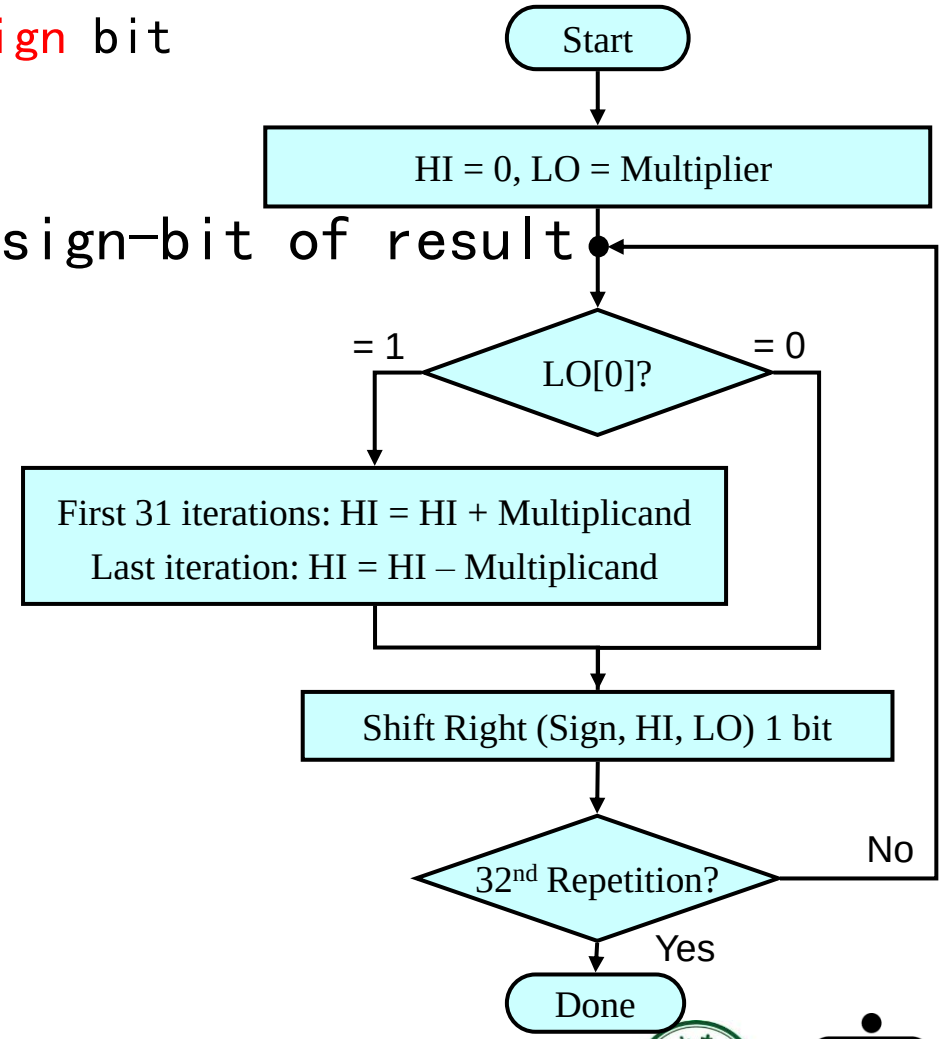
=0100

减1100等于加0100

Sequential Signed Multiplier

- ALU produces 32-bit result + Sign bit
- Check for overflow
  - No overflow ➔ Extend sign-bit of result





Signed Multiplication Example

- Consider: 1100_2 (-4) $\times 1101_2$ (-3), Product = 00001101_2
- Check for overflow: No overflow ➔ Extend sign bit
- Last iteration: add 2's complement of Multiplicand

Iteration		Multiplicand	Sign	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 => ADD		1	1 1 0 0 1 1 0 1
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 0 0 1 1 0
2	LO[0] = 0 => Do Nothing			
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 1 0 0 1 1
3	LO[0] = 1 => ADD		1	1 0 1 1 0 0 1 1
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 0 1 1 0 0 1
4	LO[0] = 1 => SUB (ADD 2's compl)	0 1 0 0	0	0 0 0 1 1 0 0 1
	Shift (Sign, HI, LO) right 1 bit			0 0 0 0 1 1 0 0

并行乘法器

早期的计算机中采用串行的1位乘法方案，即多次执行“*加--移位*”操作来实现。这种方法硬件简单，但速度太低，不能满足科学技术对高速乘法所提出的要求。自从大规模集成电路问世以来，高速的单元阵列乘法器应运而生，出现了各种形式的流水式阵列乘法器，它们属于*并行乘法器。*

设两个**不带符号**的二进制整数

$A=a_{m-1}\cdots a_1a_0$

$B=b_{n-1}\cdots b_1b_0$

则二进制乘积$P=ab=(\sum a_i2^i)(\sum b_j2^j)=\sum\sum(a_ib_j)2^{i+j}$

阵列乘法器

❖基本思路

若将所有的a_ib_j都一起算出来，剩下的就是带进位位的加法求和。这种乘法器要实现n位* n位时，需要n(n-1)个全加器和n 个“与”门。

- 以 4 位无符号数为例

■手工计算： 15×11=165

被乘数 1111

乘数 × 1011

1111

1111

0000

积 + 1111

10100101

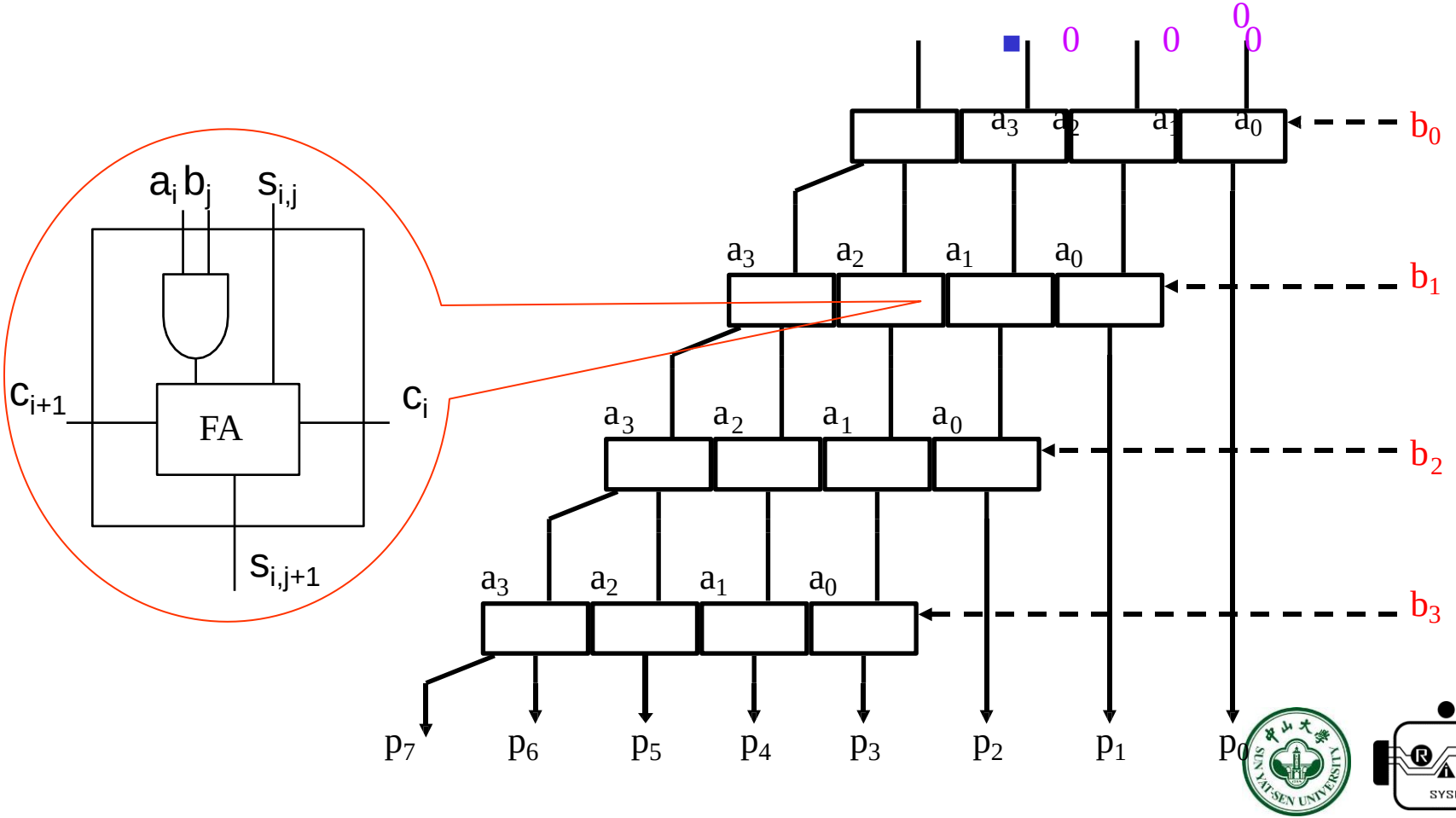
				a3	a2	a1	a0
				× b3	b2	b1	b0
				a3·b0	a2·b0	a1·b0	a0·b0
			a3·b1	**a2·b1**	**a1·b1**	**a0·b1**	
		a3·b2	**a2·b2**	**a1·b2**	**a0·b2**		
	+ a3·b2	**a2·b2**	**a1·b2**	**a0·b2**			
p7	**p6**	**p5**	**p4**	**p3**	**p2**	**p1**	**p0**

其中 $C_{ij}=A_iB_j$

（4）乘法器

ALU电路设计

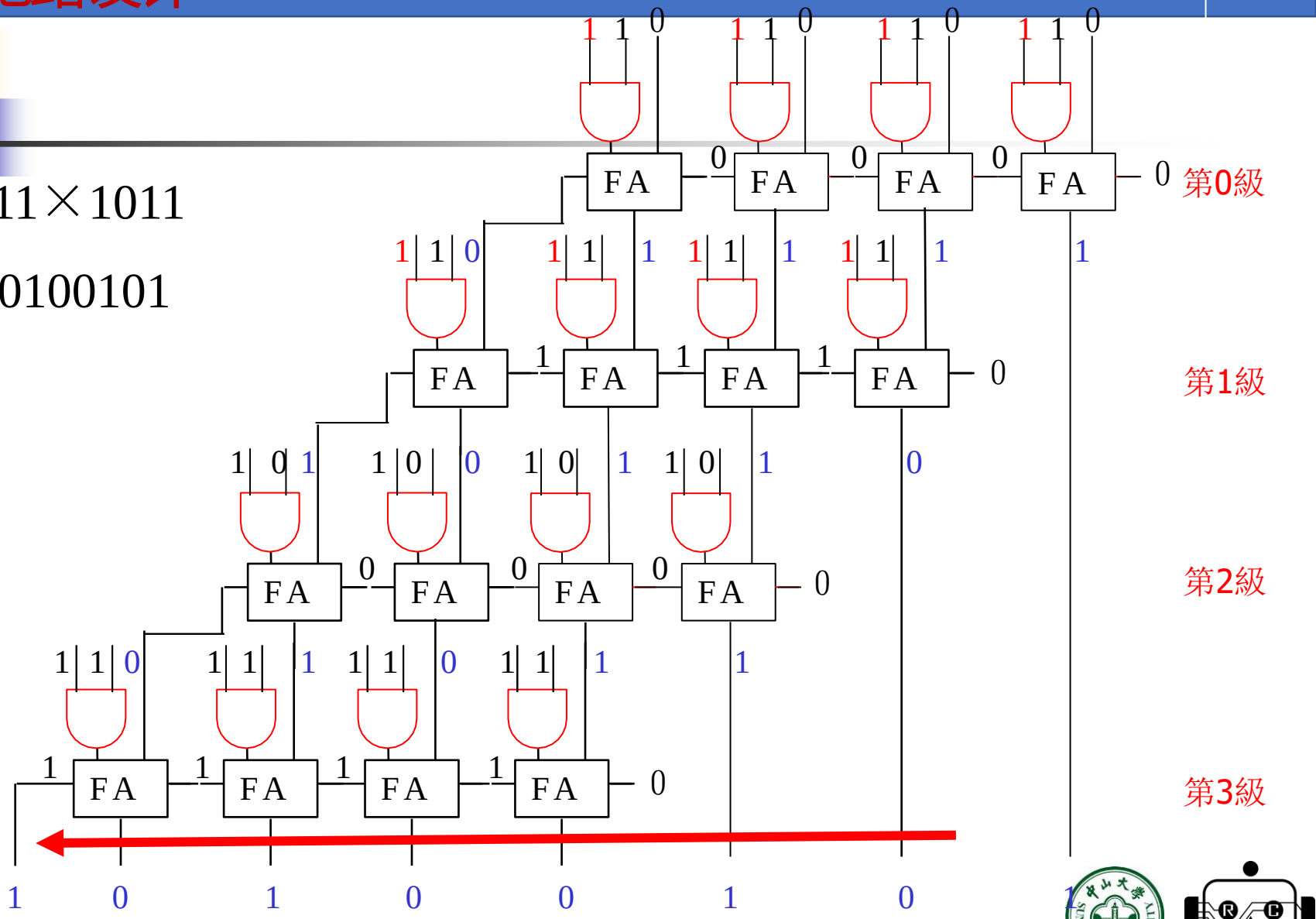
- 阵列乘法器(并行乘法器)——方案



阵列乘法
ALU电路设计

1111×1011

=10100101

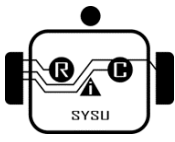


阵列乘法存在的问题
ALU电路设计

- 存在的问题：
  - 第0级的加法没有必要
  - 每一级加法采用串行进位，速度受到影响
- 改进措施：采用CSA——保存进位加法器(Carry Save Adder)
  - CSA输出每一位相加的部分和，同时将每一位的进位保存下来，作为一个输出结果供下一级加法器来处理，而不是向同一级的下一位进位
  - 由于CSA不存在同级每个位之间的串行进位，所以能以更快的速度得到进位和部分和

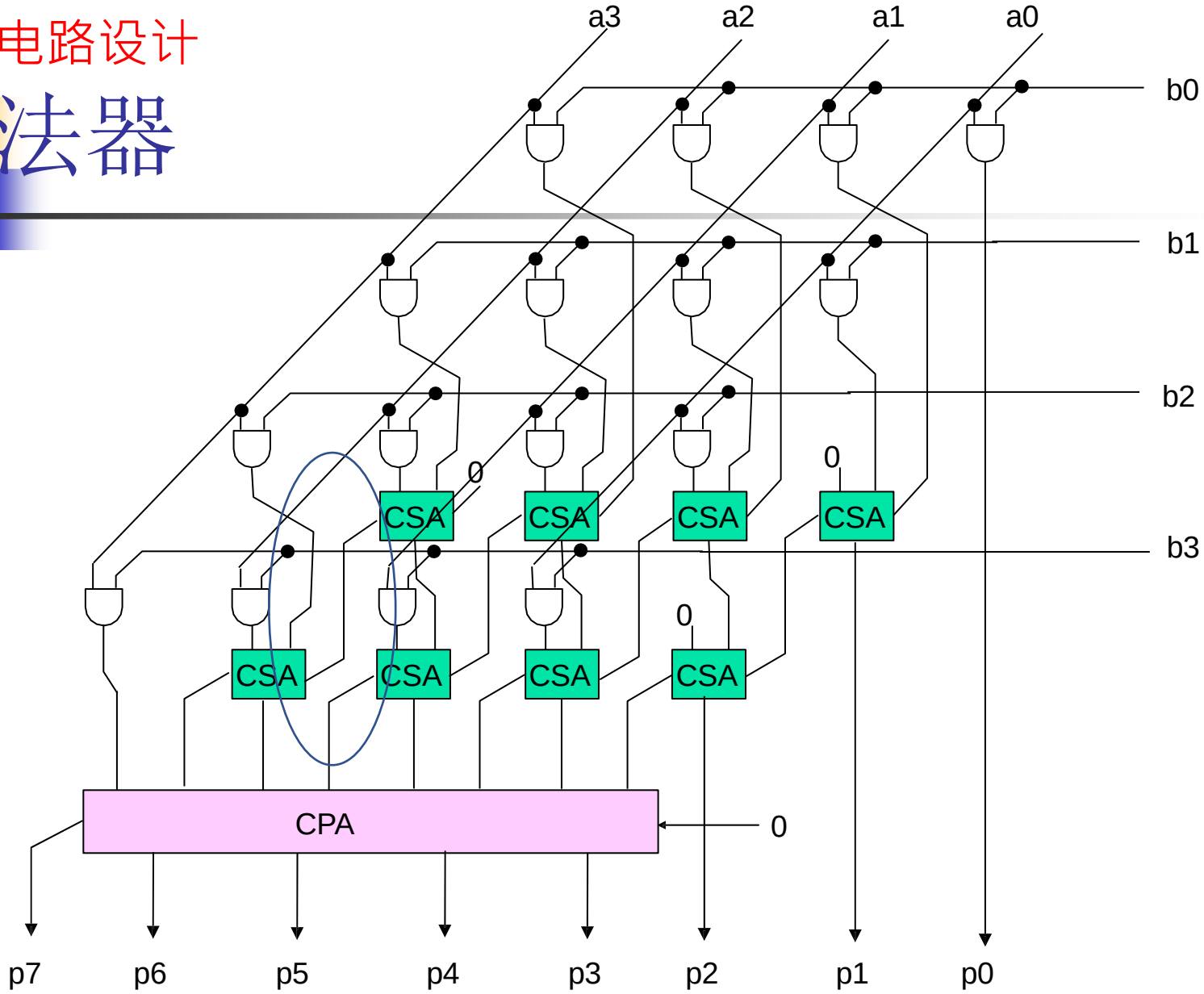
乘法
ALU电路设计

- 从结构和逻辑上看，1位CSA就是一个1位全加器
- 采用CSA构造阵列乘法器，前面几级采用不考虑进位的CSA，最后一级采用常规的进位传播加法器CPA(Carry Propagate Adder)产生最后的乘积结果
- CPA可以采用串行进位加法器，也可以采用先行进位加法器



ALU电路设计

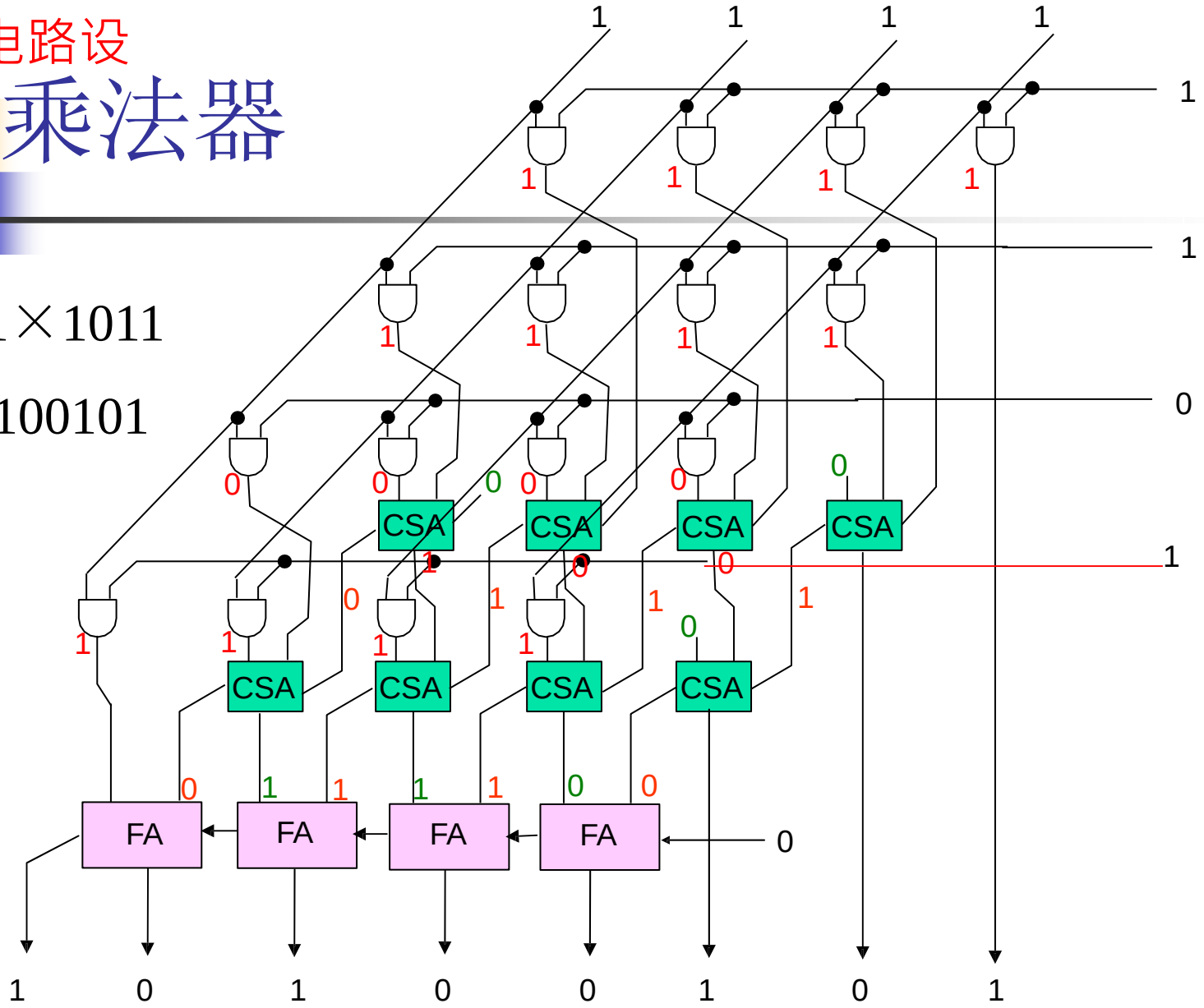
乘法器


a3
a2
a1
a0
b0
b1
b2
b3
0
0
0
0
CSA
CSA
CSA
CSA
CSA
CSA
CSA
CSA
CPA
0
p7
p6
p5
p4
p3
p2
p1
p0

ALU电路设计 乘法器

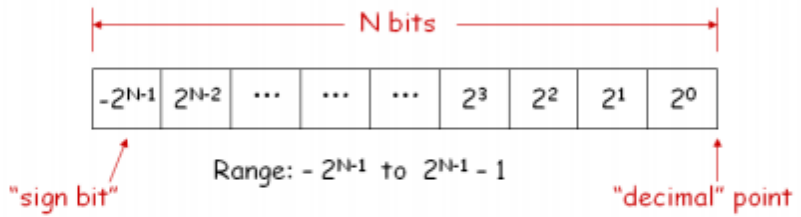
1111×1011

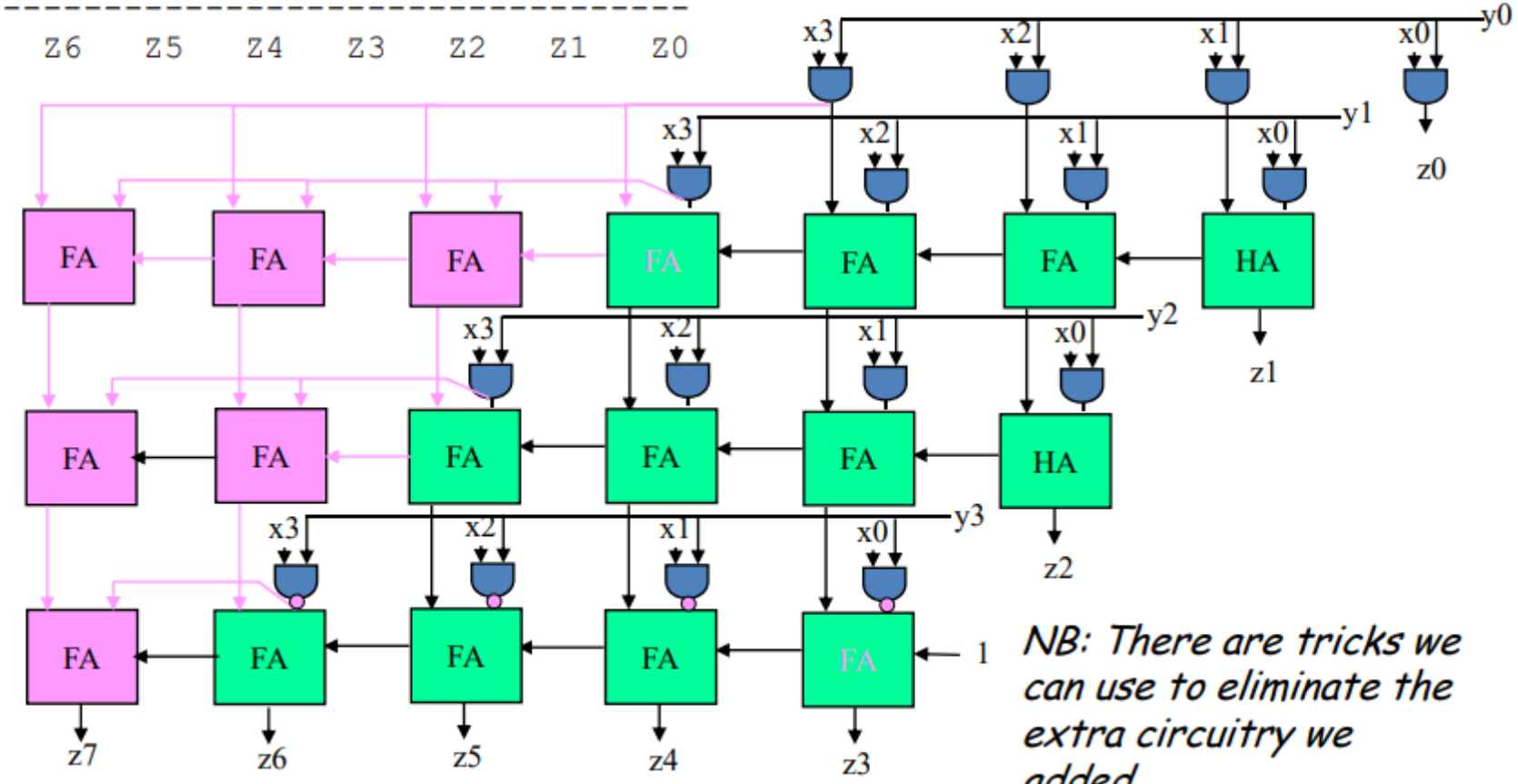
=10100101



Combinational Multiplier (signed!)

```
                        X3    X2    X1    X0
                    *   Y3    Y2    Y1    Y0
                    ------------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
-----------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```



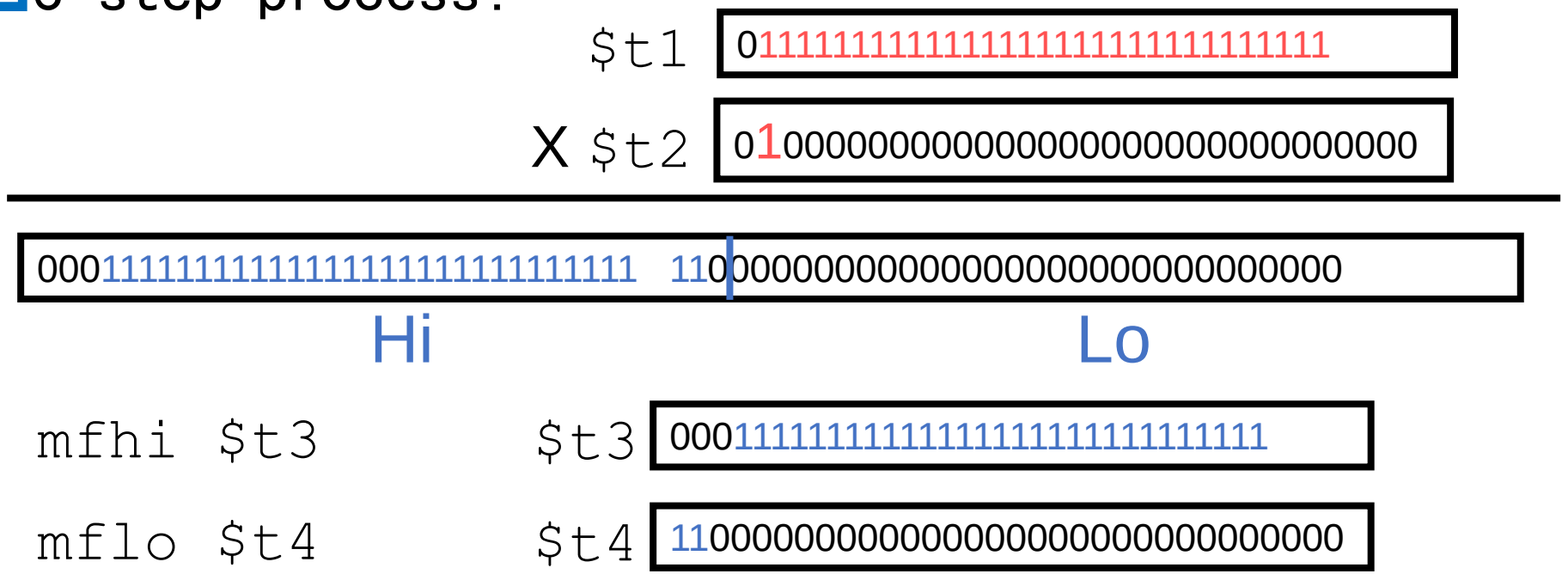


NB: There are tricks we can use to eliminate the extra circuitry we added...

Multiplication in MIPS

```
mult $t1, $t2          # t1 * t2
```

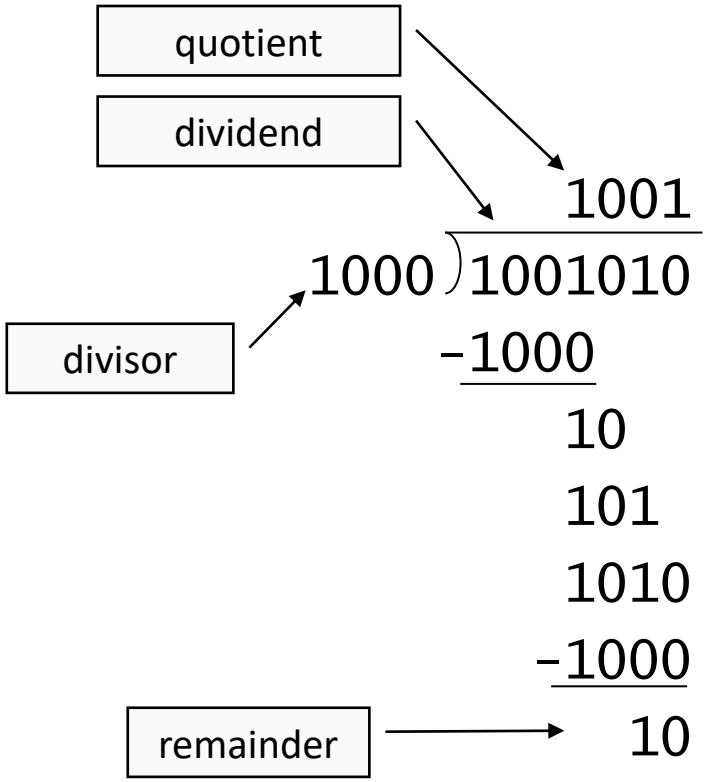
- No destination register: product could be ~2^{64}; need two special registers to hold it
- 3-step process:



```
//hi lo寄存器
    reg [31:0]  hi;
    reg [31:0]  lo;
    wire Write_hi,Write_lo;

    //always@(inst[31:26],inst[5:0])
    always@(negedge clkin)
    begin
        if(inst[31:26]==6'b000000&&inst[5:0]==6'b011000)
        begin //mult
            {hi,lo}=RsData*RtData;
        end
        else if(inst[31:26]==6'b000000&&inst[5:0]==6'b011010)
       begin //div
            lo=RsData/RtData;
            hi=RsData%RtData;
        end
        else if(inst[31:26]==6'b000000&&inst[5:0]==6'b010001)
        begin //mthi
            hi=RsData;
        end
        else if(inst[31:26]==6'b000000&&inst[5:0]==6'b010011)
       begin //mtlo
            lo=RsData;
        end
```

3.4 Division



n-bit operands yield *n*-bit quotient and remainder

笔算除法分析:

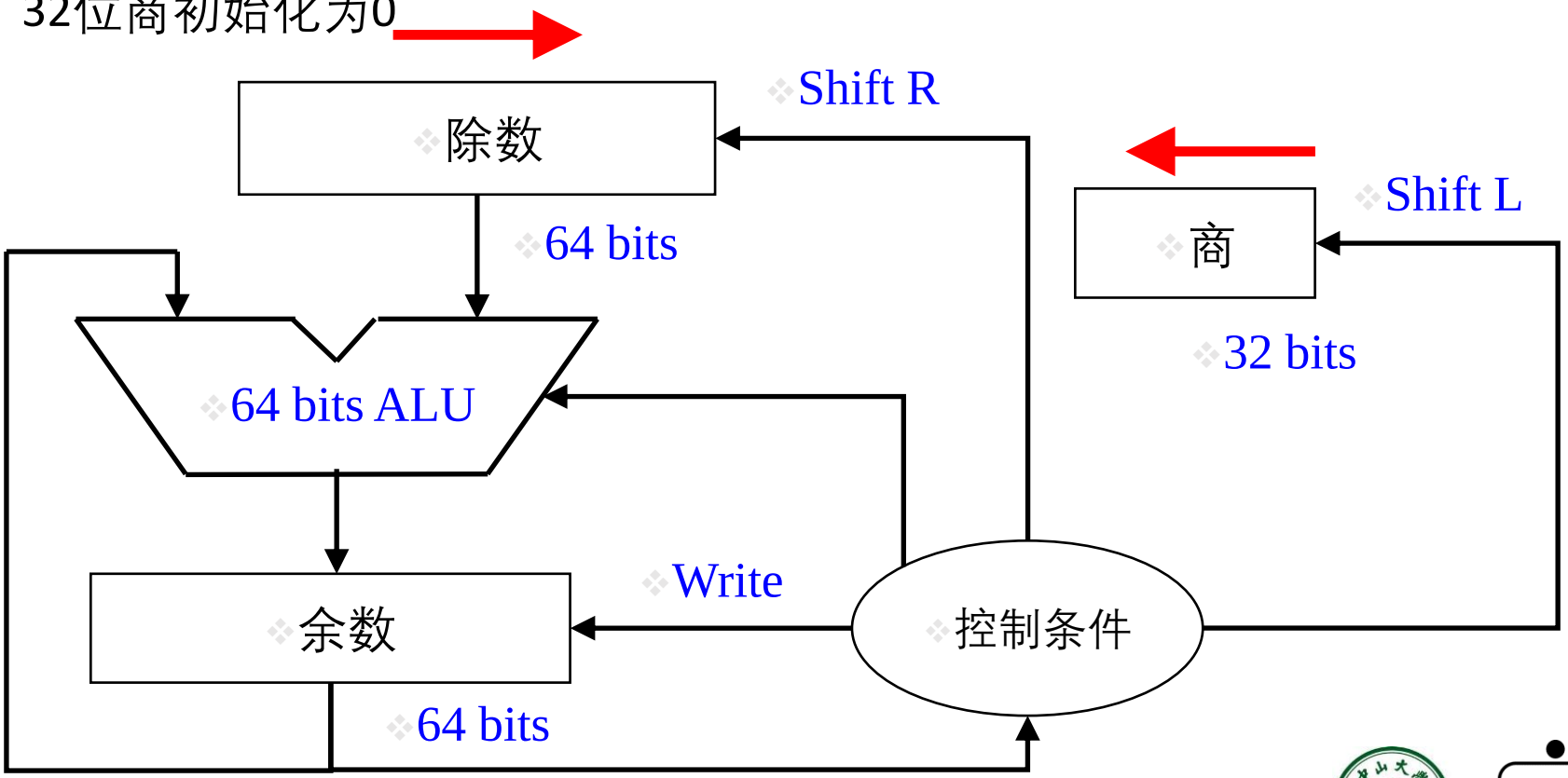
- 1.二进制除法实质是“作被除数（余数）和除数的减法，求新的余数”的过程；
- 每次上商都是由心算来比较余数(被除数)和除数的大小，确定商为1还是0；
- 每做一次减法，总是保持余数不动，低位补0，再减去右移后的除数；
- 商符单独处理。

被除数 = 商 × 除数 + 余数

第一种除法算法

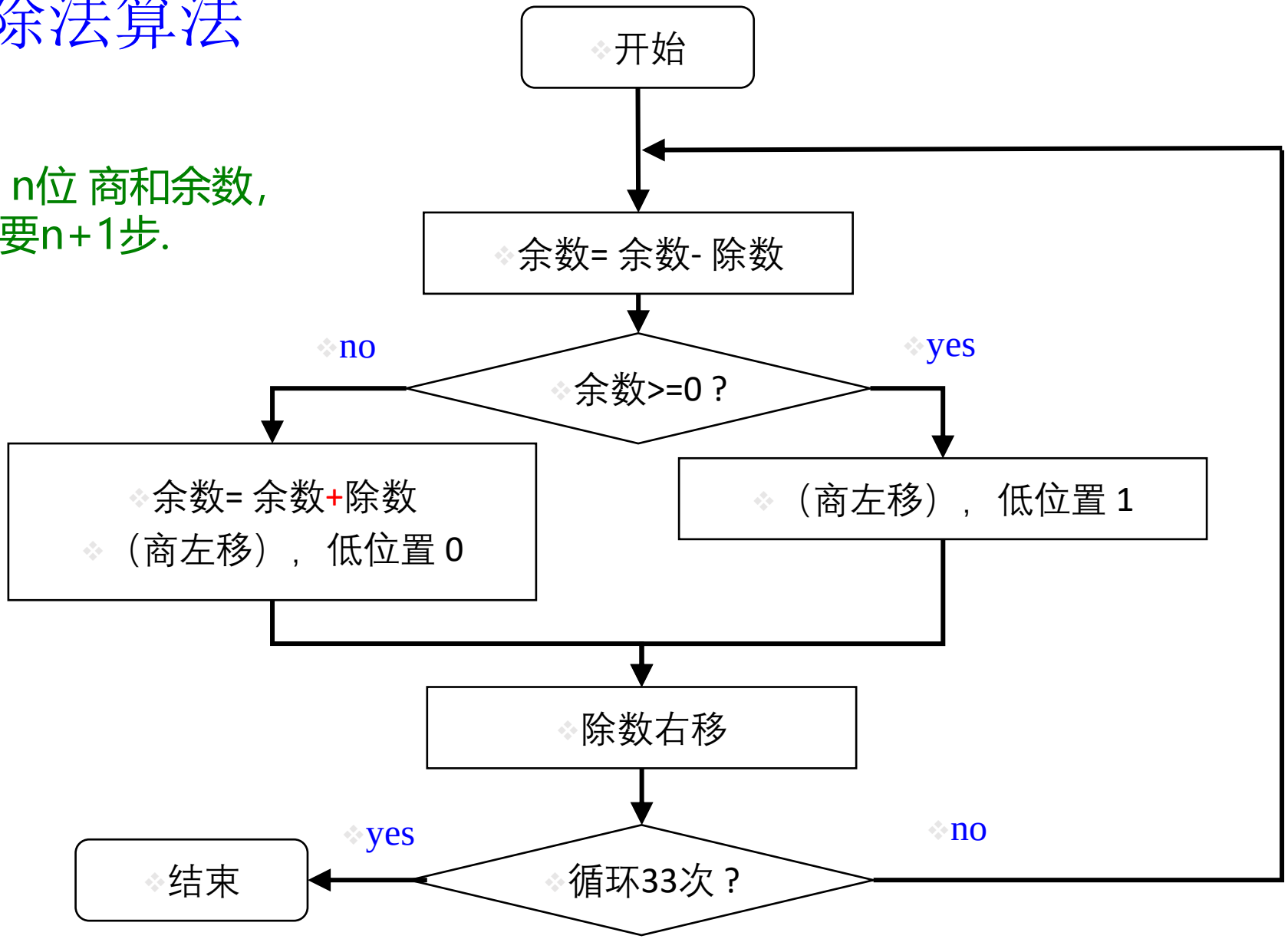
32位二进制除法硬件电路

- 32位被除数放在64位余数寄存器的低32位中；
- 32位除数放在64位除数寄存器的高32位，低32位填0
- 32位商初始化为0



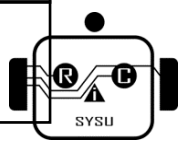
除法算法

对 n位 商和余数，
需要n+1步.



计算：$7_{10} \div 2_{10} = 0111_2 \div 0010_2$

选代	步骤	商	除数	余数
0	初始化	0000	0010 0000	0000 0111
1	余数=余数-除数	0000	0010 0000	1110 0111
	余数=余数+除数 商左移，LSB置0	0000	0010 0000	0000 0111
	右移除数	0000	0001 0000	0000 0111
2	余数=余数-除数	0000	0001 0000	1110 0111
	余数=余数+除数 商左移，LSB置0	0000	0001 0000	0000 0111
	右移除数	0000	0000 1000	0000 0111

计算：$7_{10} \div 2_{10} = 0111_2 \div 0010_2$（续2）

选代	步骤	商	除数	余数
3	余数=余数-除数	0000	0000 1000	1110 0111
	余数=余数+除数 商左移，LSB置0	0000	0000 1000	0000 0111
	右移除数	0000	0000 0100	0000 0111
4	余数=余数-除数	0000	0000 0100	0000 0011
	商左移，LSB置1	0001	0000 0100	0000 0011
	右移除数	0001	0000 0010	0000 0011

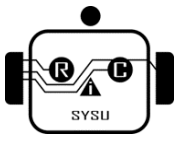
计算：$7_{10} \div 2_{10} = 0111_2 \div 0010_2$（续3）

选代	步骤	商	除数	余数
5	余数=余数-除数	0001	0000 0010	0000 0001
	商左移，LSB置1	0011	0000 0010	0000 0001
	右移除数	0011	0000 0001	0000 0001

补码恢复余数除法。

除数中 1/2的位数 总是为 0 => 64位加法器的1/2浪费 !!!

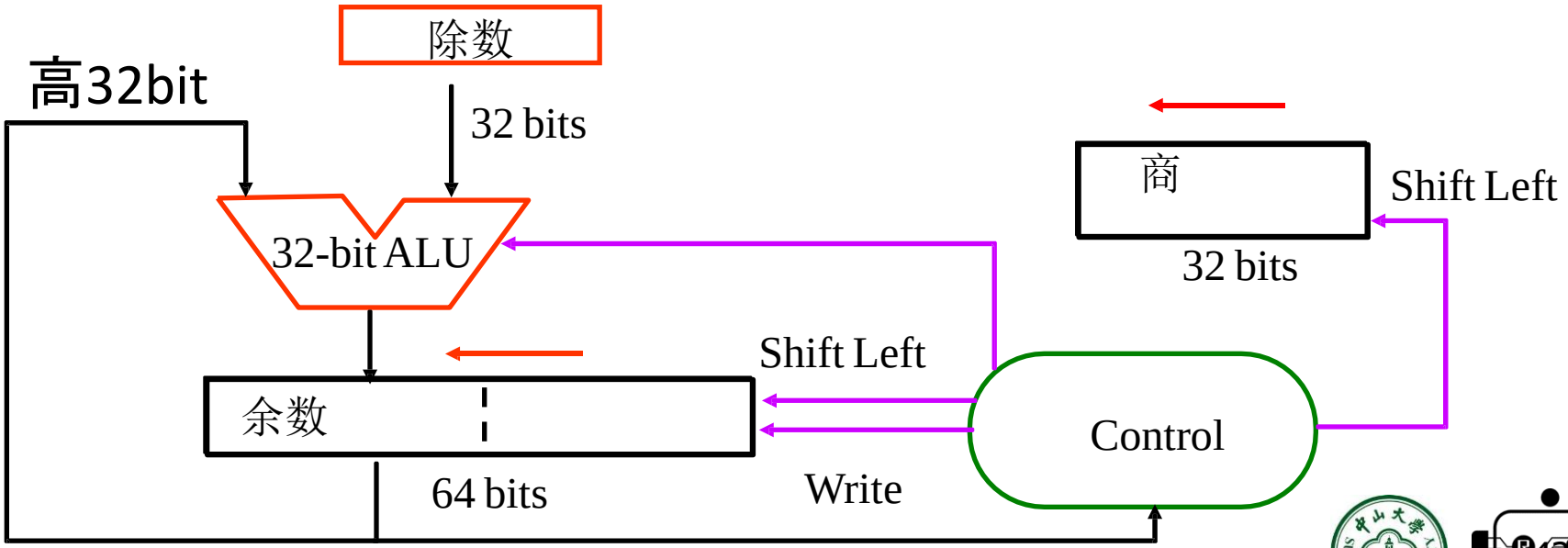
- 是否可以用 余数左移 替代 除数右移？
- => 变换次序 到 首先移位 然后 再减, 可以减少一次迭代



ALU电路设计
除法器

- 除数和ALU的宽度减少一半
- => 变换次序 到 首先移位 然后 再减, 可以减少一次迭代

- 串行除法器算法2
  - 将除数寄存器向右移位改为余数寄存器向左移位，除数寄存器保持不变
  - 除数寄存器为32位，ALU为32位



第二种除法算法 7/2

Remainder Quotient Divisor

0000 0111 0000 0010

Q: 0000 D: 0010 R: 0000 0111

1

1: Shl R 余数左移 Q: 0000 D: 0010 R: 0000 1110

2: R = R-D 余数减去除数 Q: 0000 D: 0010 R: 1110 1110

3b: +D, sl Q, 0余数=余数+除数，商为0 Q: 0000 D: 0010 R: 0000 1110

2

1: Shl R余数左移 Q: 0000 D: 0010 R: 0001 1100

2: R = R-D 余数=余数-除数 Q: 0000 D: 0010 R: 1111 1100

3b: +D, sl Q, 0 Q: 0000 D: 0010 R: 0001 1100

3

1: Shl R Q: 0000 D: 0010 R: 0011 1000

2: R = R-D Q: 0000 D: 0010 R: 0001 1000

3a: sl Q, 1 商左移，商为1 Q: 0001 D: 0010 R: 0001 1000

4

1: Shl R Q: 0010 D: 0010 R: 0011 0000

2: R = R-D Q: 0010 D: 0010 R: 0001 0000

3a: sl Q, 1 Q: 0011 D: 0010 R: 0001 0000

n = 4 here

3b:补码恢复余数

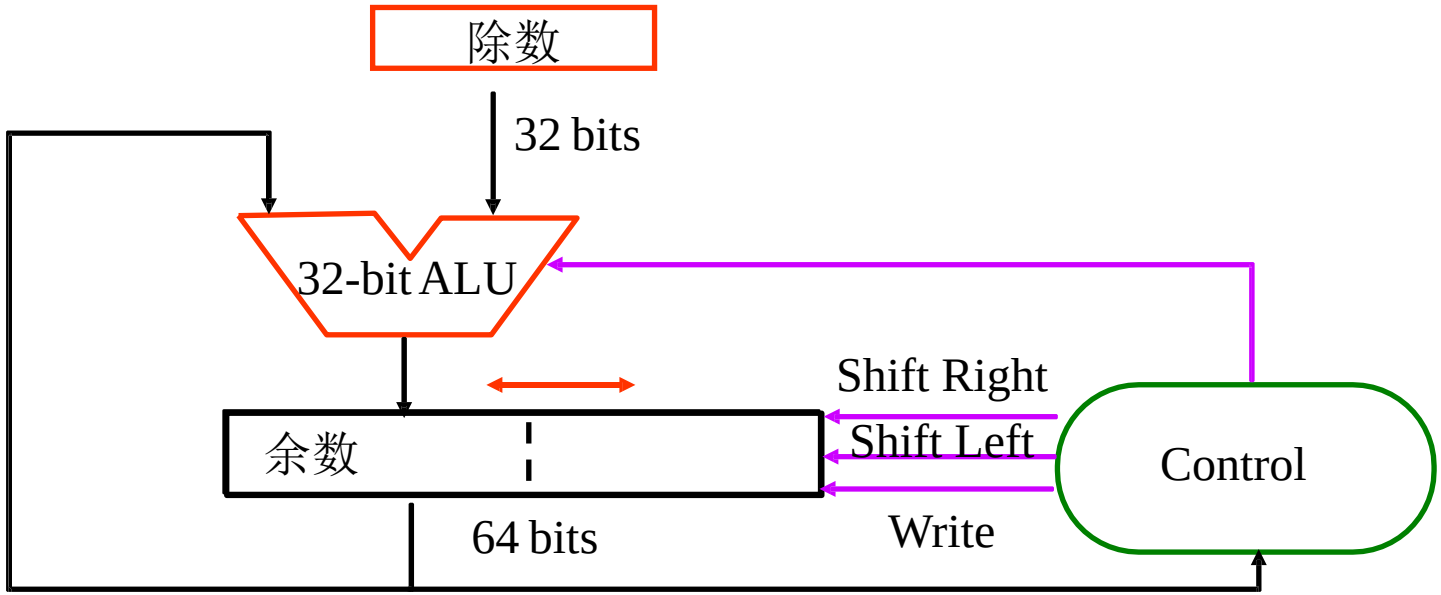
第二种除法的启示

- 通过在左移中，与余数合并，取消商寄存器
  - 象前面一样，以左移余数开始.
  - 其后，由于余数寄存器的移动即可移动左半部的余数，又可移动右半部的商，因而每次循环只包括两步.
  - 将两个寄存器联合在一起 和 循环内新的操作次序的导致余数多左移一次
  - 因而，最后一步必须将这个寄存器左半部中的余数移回

ALU电路设计
除法器

32位除数寄存器、32位ALU、64位余数寄存器(没有商寄存器)

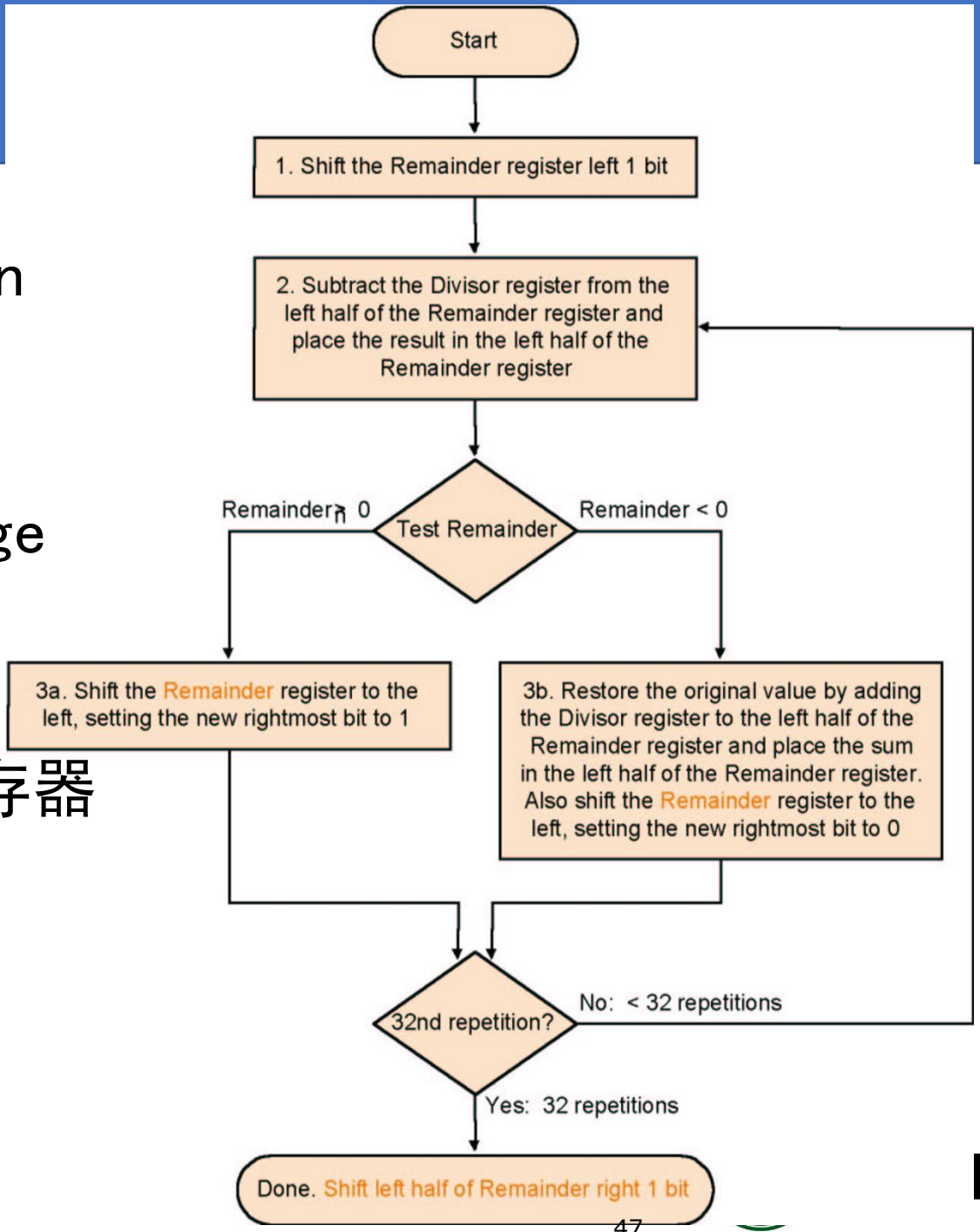
- 串行除法器算法3



Algorithm V 3

- Much the same than the last one
- Except change of register usage

其他一样，只是商寄存器的使用变化到余数寄存器低位



Example 7/2 for Division V3

- Well known numbers: 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	**0000 1110**
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0001 1100**
2	1.Rem=Rem-Div	0010	1111 0110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 1000**
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 →sll R,R_0=1	0010	**0011 0001**
4	1.Rem=Rem-Div	0010	0001 0001
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1 左半部中的余数移回		**0001 0011**

3.4.2 有符号除法

被除数=除数×商+余数

有符号除法: 最简单的方法是记住符号, 进行正数除法, 并根据需要对商和余数进行修正

- 注: 被除数和余数的符号必须相同
- 注: 如果除数和被除数的符号不同, 商为负

商有可能很大: 如果一个64位整数除以 1，那么商就为 64 位（称为“饱和”：saturation）

如果余数不与被除数符号一致，会产生被除数和除数的符号不同，商的绝对值得到不同的结果。

Signed division

- Keep the signs in mind for Dividend and Remainder
  - (+ 7) ÷(+ 2) = + 3　　Remainder = +1
    7 = 3 × 2 + (+1) = 6 + 1
  - (− 7) ÷(+ 2) = − 3　　Remainder = −1
    −7 = −3 × 2 + (−1) = − 6 − 1;
    −7 = −4 × 2 + (+1) = −8 + 1（这个也满足公式）?

(如果余数不与被除数符号一致，结果的绝对值为4)，会产生商的绝对值会因为被除数和除数的符号不同而得到不同结果，保持被除数和余数的符号必须相同可避免此

  - (+ 7) ÷(− 2) = − 3　　Remainder = +1,
    7 = −3 × (−2) + (+1) = 6 + 1
  - (− 7) ÷(− 2) = + 3　　Remainder = −1
  - −7 = 3 × (−2) + (−1) = − 6 − 1

被除数 = 商　除数 + 余数

如果操作数的符号不一致，商为负，

且被除数和余数的符号必须相同

MIPS中的除法

符号数除法指令

```
div  $s2, $s3
# Lo = $s2 / %s3; 商
# Hi = $s2 % $s3; 余数
```

无符号数除法指令

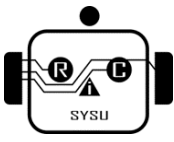
```
divu  $s2, $s3
# Lo = $s2 / %s3; 商
# Hi = $s2 % $s3; 余数
```

取除法运算结果指令

```
mflo  $s1   #$s1 = Lo
mfhi  $s1   #$s1 = Hi
```

No overflow or divide-by-0 checking

Software must perform checks if required



联系方式

- Acknowledgements:
- This slides contains materials from following lectures:
  - Computer Architecture (ETH，NUDT，USTC)

- Research Area：
  - 计算机视觉与机器人应用计算加速，
  - 人工智能和深度学习芯片及智能计算机

- Contact：
  - 中山大学计算机学院
  - 管理学院D101（图书馆右侧）
  - 机器人与智能计算实验室
  - cheng83@mail.sysu.edu.cn