中山大學
SUN YAT-SEN UNIVERSITY

SYSU

计算机组成原理

第三章：计算机中的运算

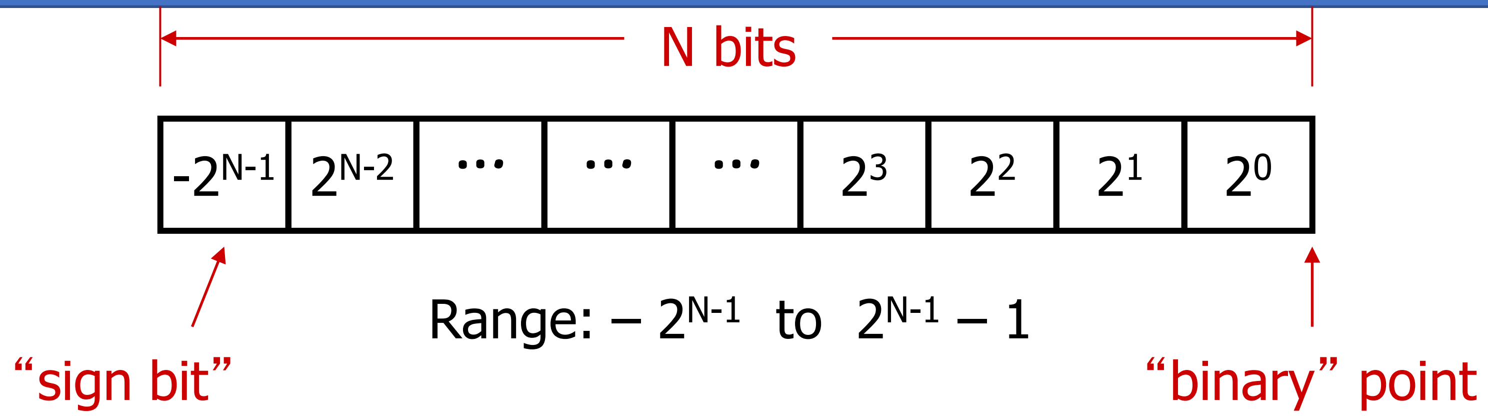
中山大学计算机学院
陈刚

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

Review: 2’s Complement



- 8-bit 2’s complement example:

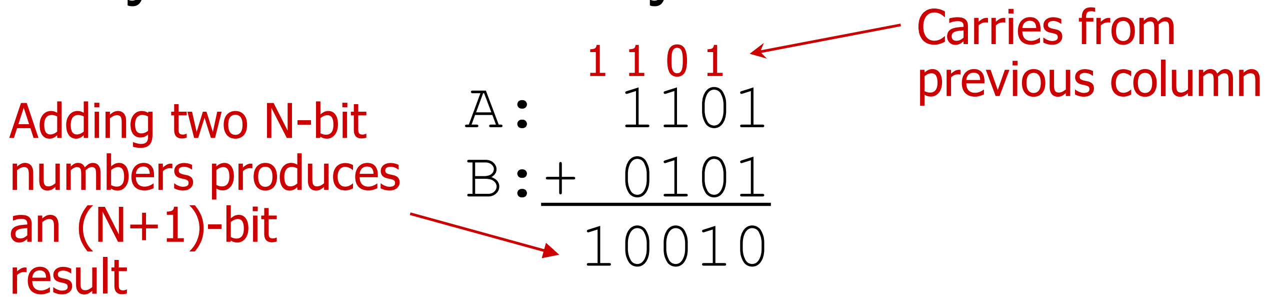
  11010110 = - 128 + 64 + 16 + 4 + 2 = - 42

- Beauty of 2’s complement representation:
  - same binary addition procedure will work for adding both signed and unsigned numbers
    - as long as the leftmost carry out is properly handled/ignored
- Insert a “binary” point to represent fractions too:

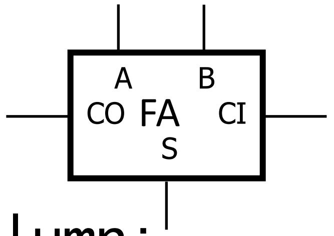
  1101.0110 = - 8 + 4 + 1 + 0.25 + 0.125 = - 2.625

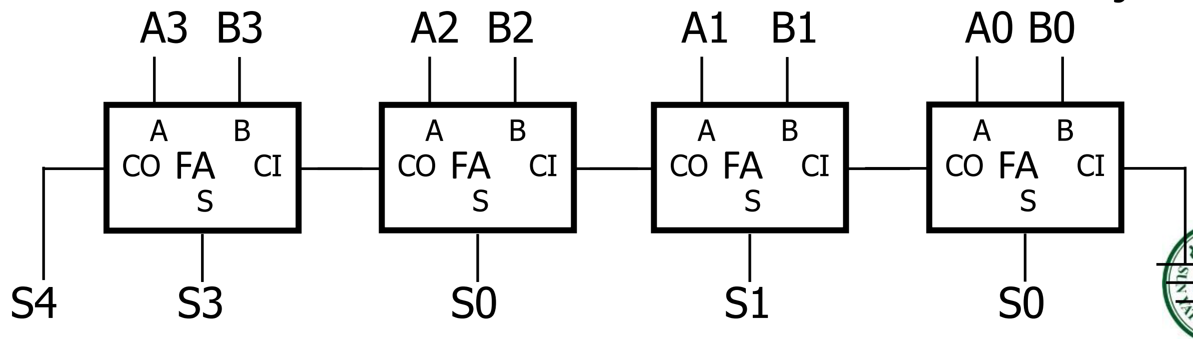
Binary Addition

- Example of binary addition “by hand”:


Carries from previous column
1 1 0 1
A: 1101
B: + 0101
10010
Adding two N-bit numbers produces an (N+1)-bit result

- Let’s design a circuit to do it!
  - Start by building a block that adds <u>one column</u>:
    - called a “full adder” (FA)
  - Then cascade them to add two numbers of any size…





Designing an Adder (1位全加器)

- Follow the step-by-step method
  1. Start with a truth table:
  2. Write down eqns for the "1" outputs:

$$C_o = \overline{C_i}AB + C_i\overline{A}B + C_iA\overline{B} + C_iAB$$

$$S = \overline{C_i}\,\overline{A}B + \overline{C_i}A\overline{B} + C_i\overline{A}\,\overline{B} + C_iAB$$

  3. Simplifying a bit (seems hard, but experienced designers are good at this art!)

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

真值表:

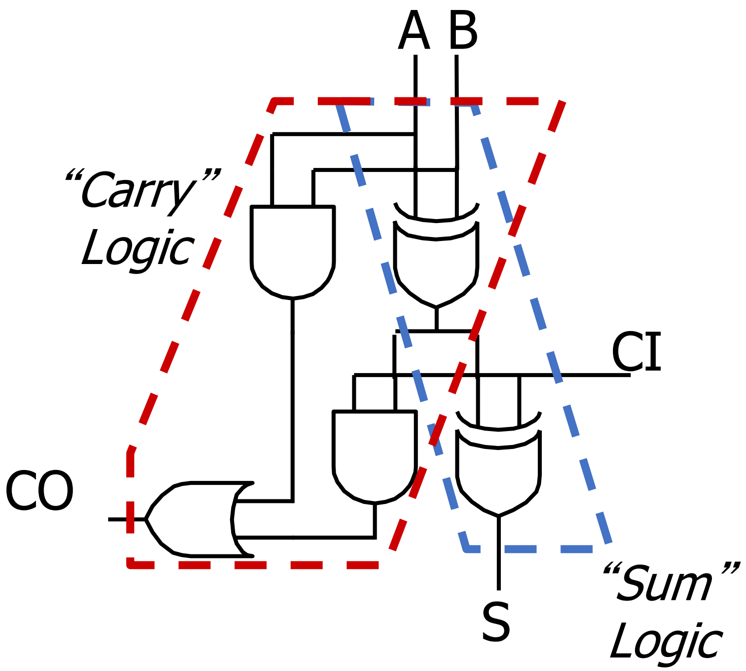
C_i	A	B	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

For Those Who Prefer Logic Diagrams

- A little tricky, but only 5 gates/bit!

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

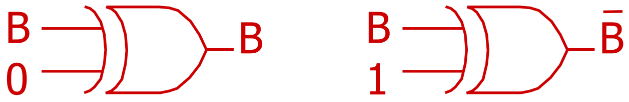


XOR的真值表:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Subtraction: A-B = A + (-B)

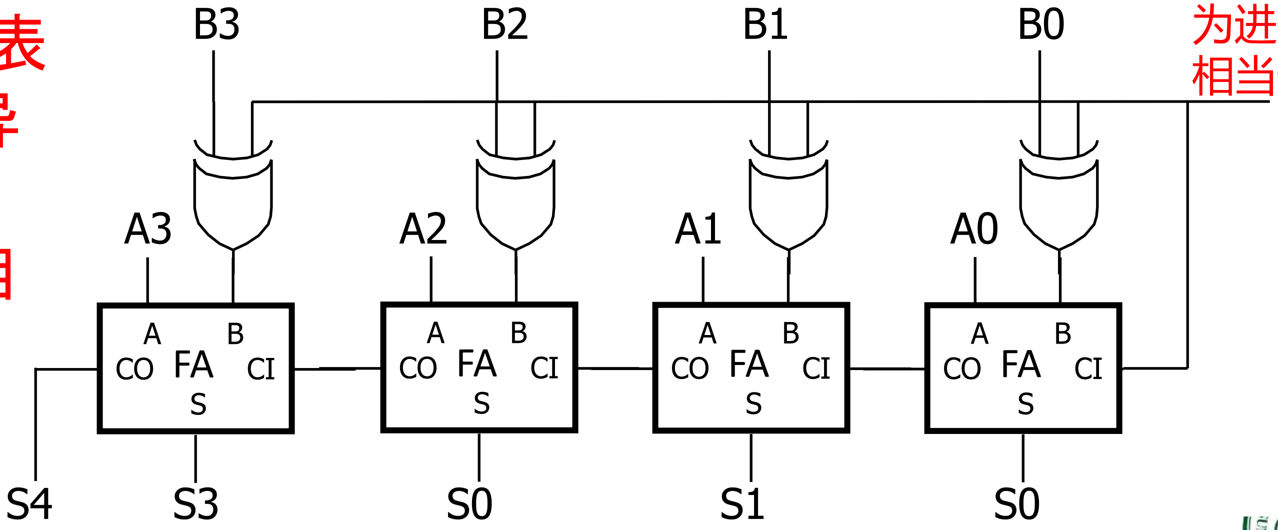
- Subtract B from A = add 2's complement of B to A
  - In 2's complement: $-B = \sim B + 1$ ← ~ = bit-wise complement



- Let's build an arithmetic unit that does *both* add and sub
  - operation selected by *control input*:

1表示减，0表示加,1与1异或为0,0与1异或为1，相当于取反

Subtract
1表示减，作为进位输入，相当于加1



Condition Codes

- One often wants 4 other bits of information from an arith unit:
  - Z (zero): result is = 0
    - big NOR gate
  - N (negative): result is < 0
    - S_{N-1}
  - C (carry): indicates that most significant position produced a carry, e.g., "1 + (-1)"
    - Carry from last FA
  - V (overflow): indicates answer doesn't fit
    - precisely: $V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

To compare A and B, perform A–B and use condition codes:

Signed comparison:

$\boxed{\text{LT}\ N \oplus V}$	
LE	$Z + (N \oplus V)$
EQ	Z
NE	$\sim Z$
GE	$\sim (N \oplus V)$
GT	$\sim (Z + (N \oplus V))$

Unsigned comparison:

LTU	C
LEU	$C + Z$
GEU	$\sim C$
GTU	$\sim (C + Z)$

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible (Examples are 8-bit numbers)

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

1	1	1	1	1					
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	248 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

	1								
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

Carry = 0 Overflow = 1

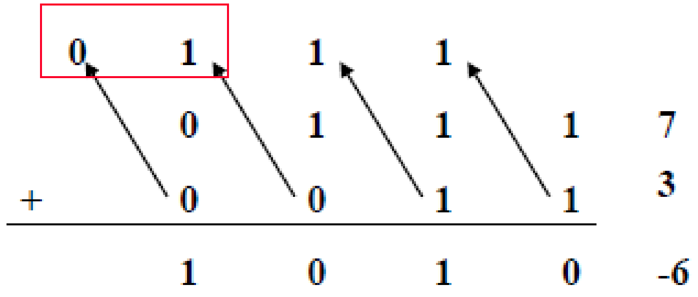
1			1	1					
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

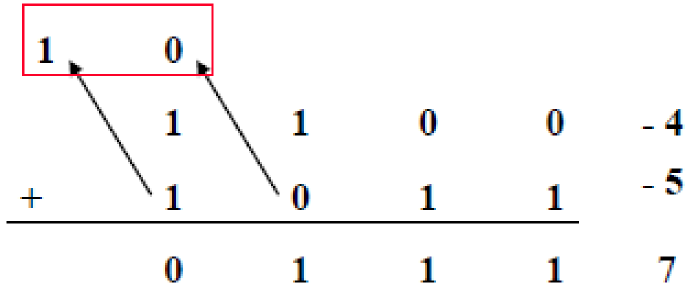
Carry = 1 Overflow = 1

溢出检测

- 溢出：结果超出了正常的表示范围（太大 和 太小）
  - 例如：－ 8 < = 4位二进制数 <= 7
- 当对具有不同符号的操作数进行加法运算时，不会出现溢出！
- 当进行下列加法时，出现溢出：
  - 两个正数相加，结果为负
  - 两个负数相加，结果为正
- 溢出可以用如下方法检测：
  - 输入到最大位的进位 != 从最大位输出的进位

(Carry into MSB ! = Carry out of MSB)

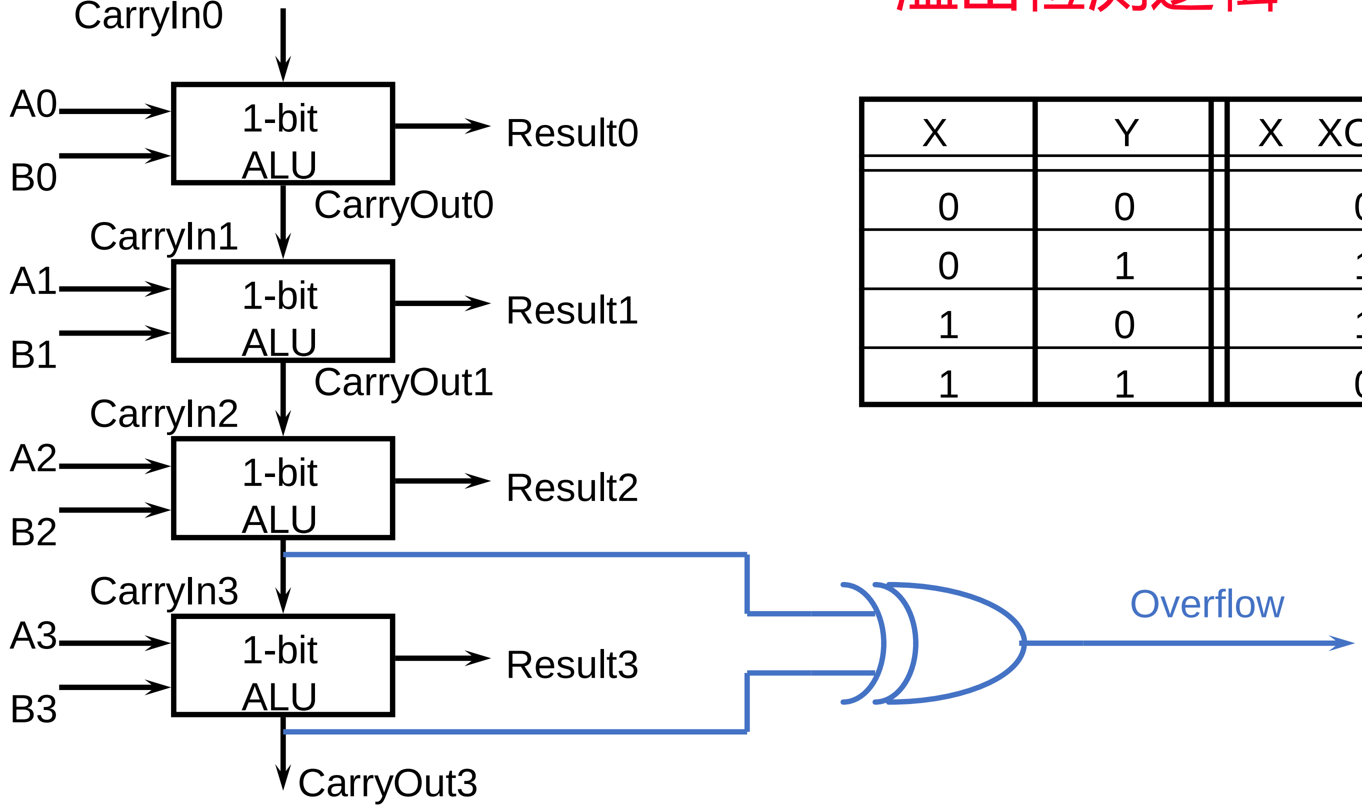




Overflow Detection Logic

- Overflow = CarryIn[N-1] XOR CarryOut[N-1]

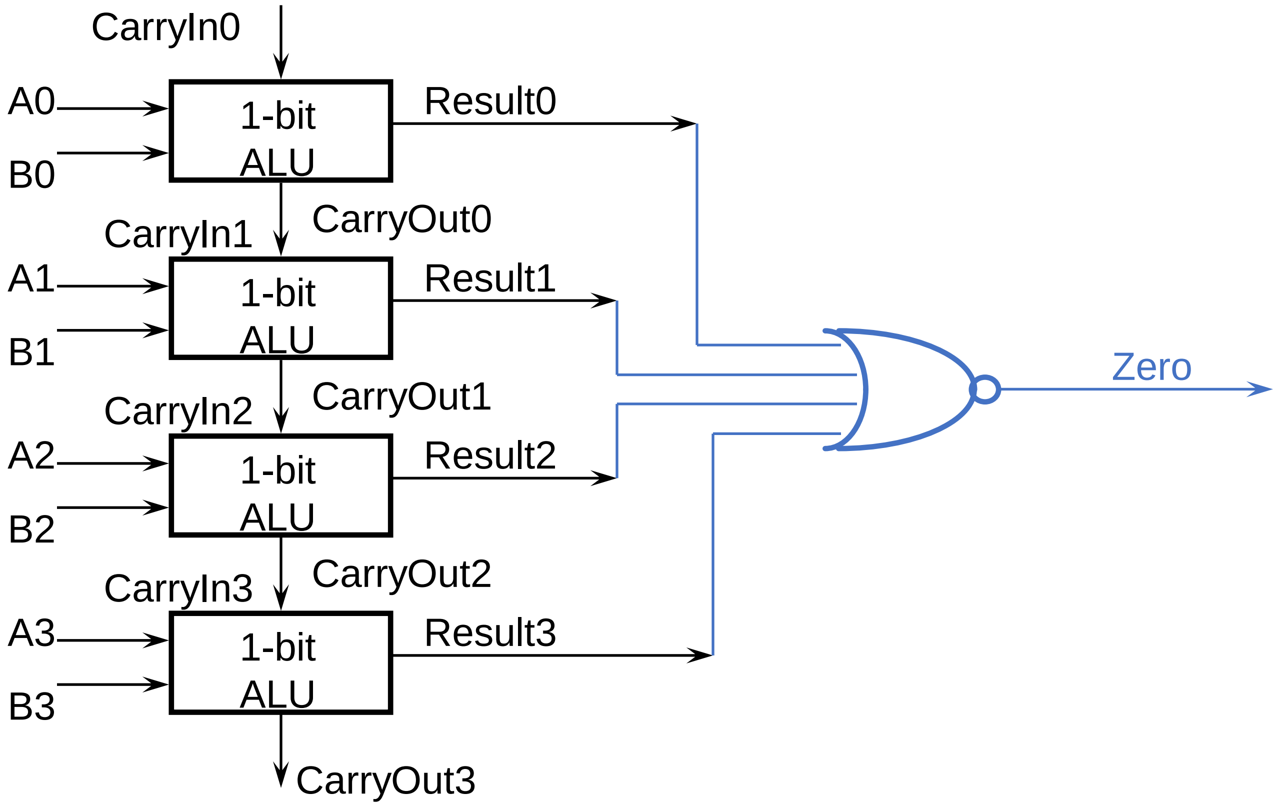
溢出检测逻辑



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Zero Detection Logic

- Zero Detection Logic is a one BIG NOR gate (support conditional jump)



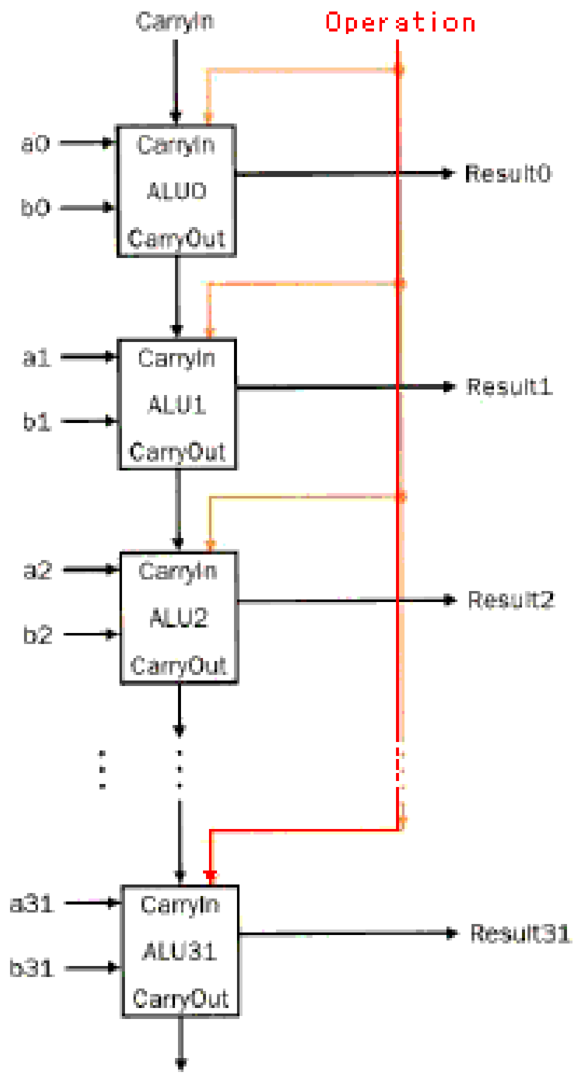
```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
module addsub
#(parameter WIDTH=8)          //指定数据宽度参数，缺省值是8
(
   input [(WIDTH-1):0] a,      // 缺省位数由参数WIDTH决定
   input [(WIDTH-1):0] b,
   input  sub,                 // =1为减法
   output [(WIDTH-1):0] sum,
   output cf,                  // 进位标志
   output ovf,                  // 溢出标志
   output sf,                  // 符号标志
   output zf                   // 为0标志
   );
wire [(WIDTH-1):0] subb,subb1;
     wire cf2;    // 进位
     assign subb1 = b ^ {WIDTH{sub}};  // 对于减法是取反
     assign subb  =  subb1 + sub;   // 对于减法是加1，sub=1（减法）sub=0（加法）

      assign {cf2,sum} = a + subb;
      assign sf = sum[WIDTH-1];
      assign zf = (sum == 0) ? 1 : 0 ;
      assign cf = cf2 ^ sub;
      assign ovf = (a[WIDTH-1] ^ sum[WIDTH-1]) & (subb[WIDTH-1] ^ sum[WIDTH-1]);

endmodule
```

Basic 32 bit ALU

- Inputs parallel
- Carry is cascaded
- Ripple carry adder
- Slow, but simple
- 1st Carry In = 0



Set on Less Than

- Functions
  - AND
  - OR
  - Add
  - Subtract
- Missing: comparison
  - Slt rd, rs, rt
  - If rs < rt, rd=1, else rd=0 $N \oplus V$? ? ?
  - All bits = 0 except the least significant
  - Subtraction (rs − rt), if the result is negative -> rs < rt
  - Use of sign bit as indicator N

- Missing: comparison
  - Slt rd, rs, rt
  - If rs < rt, rd=1, else rd=0 N⊕V? ? ?
  - All bits = 0 except the least significant
  - Subtraction (rs – rt), if the result is negative -> rs < rt
  - Use of sign bit as indicator N
  - Why N⊕V? ? ? Why requires to check the overflow?
- Consider:
- case 1: rs<0 , rt>0, (rs<rt)
- case 2 rs>0 , rt<0, (rs>rt)
- If no overflow occurs, case 1: N is 1; case 2: N is 0
- If overflow occurs, case 1: N is 0; case 2: N is 1

Condition Codes

- One often wants 4 other bits of information from an arith unit:
  - Z (zero): result is = 0
    - big NOR gate
  - N (negative): result is < 0
    - S_{N-1}
  - C (carry): indicates that most significant position produced a carry, e.g., "1 + (-1)"
    - Carry from last FA
  - V (overflow): indicates answer doesn't fit
    - precisely:

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

To compare A and B, perform A–B and use condition codes:

Signed comparison:

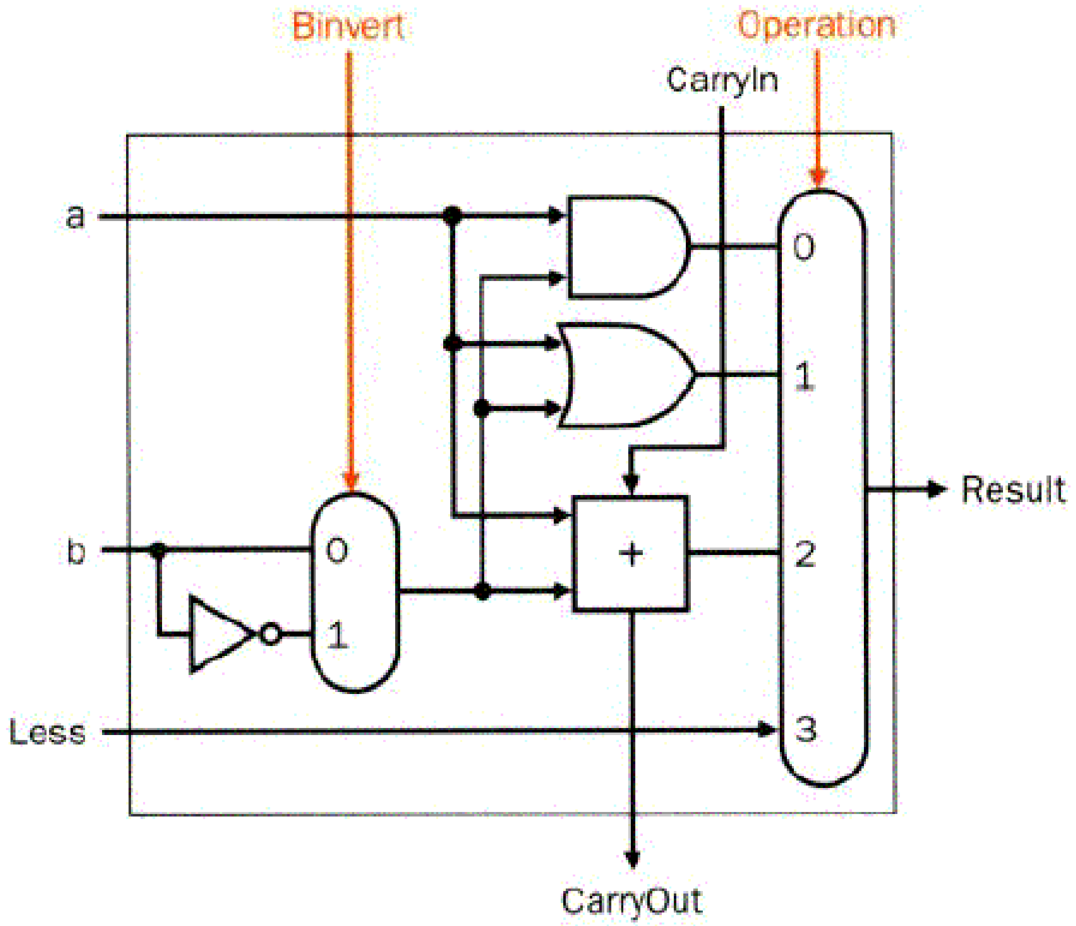
LT	$N \oplus V$
LE	$Z + (N \oplus V)$
EQ	Z
NE	$\sim Z$
GE	$\sim (N \oplus V)$
GT	$\sim (Z + (N \oplus V))$

Unsigned comparison:

LTU	C
LEU	$C+Z$
GEU	$\sim C$
GTU	$\sim (C+Z)$

Extended 1 bit ALU

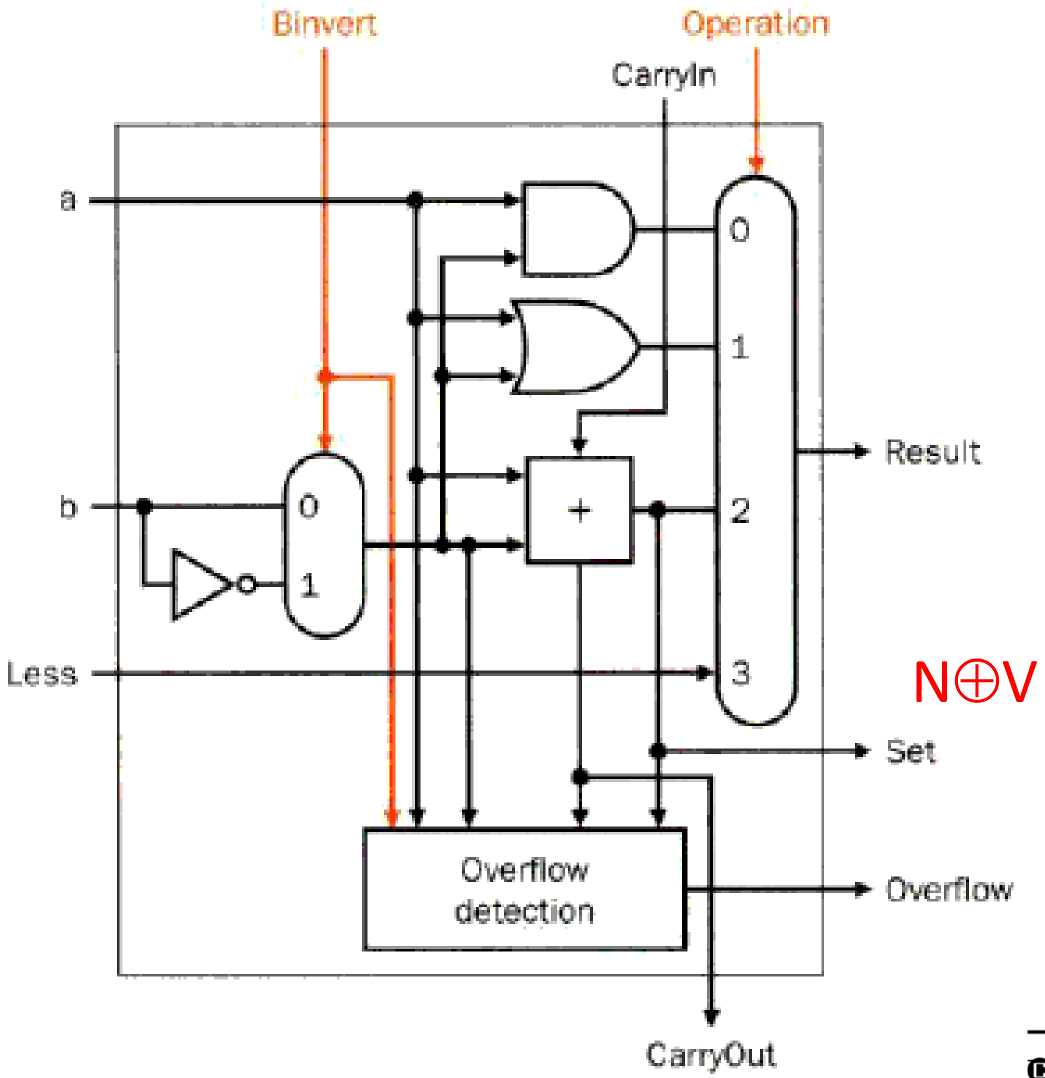
- ALU bit with input for Less data



AND
OR
Add Subtract
less

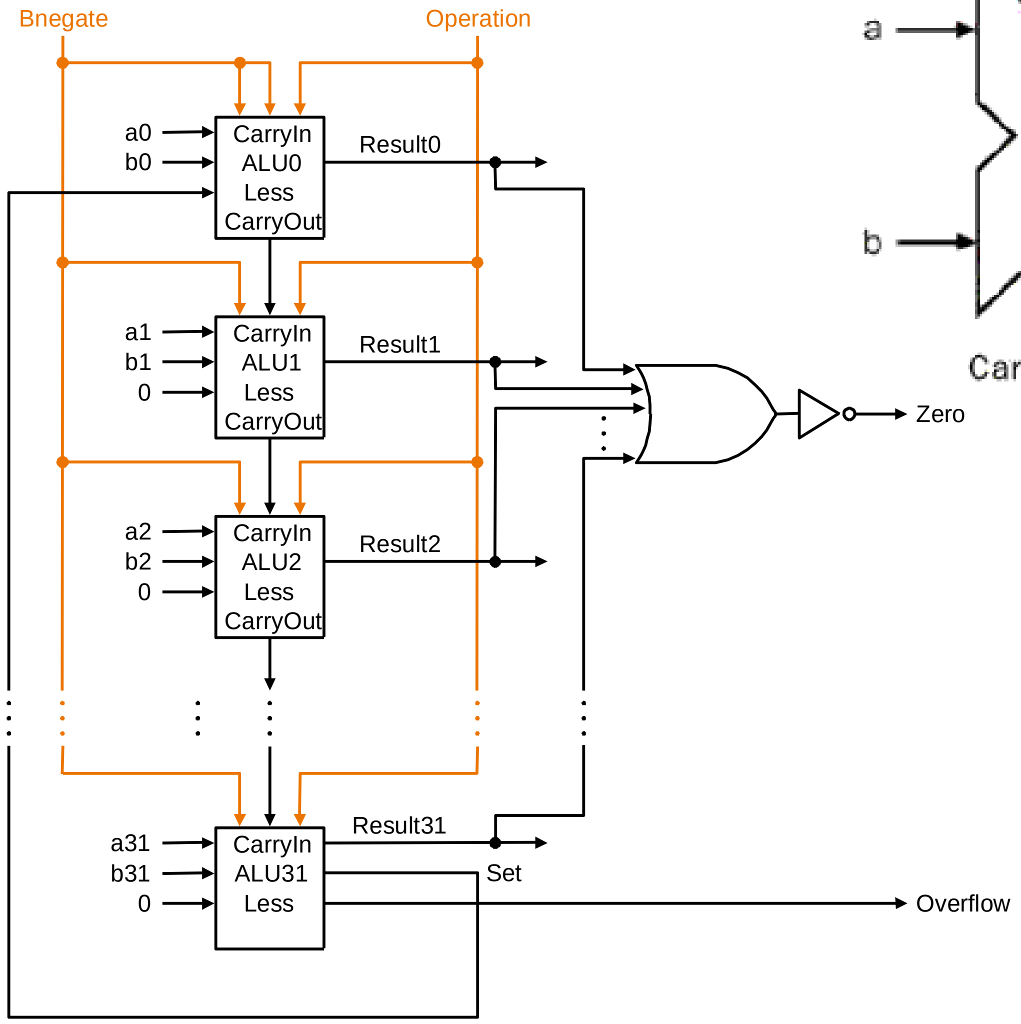
Most significant bit

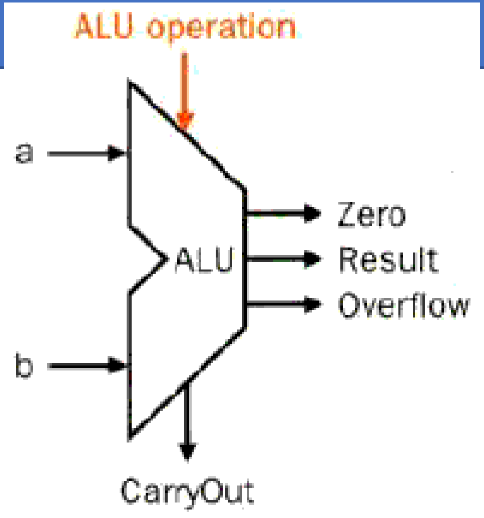
- Set for comparison
- Overflow detect



Complete ALU

- Input
  - A
  - B
- Control lines
  - Binvert
  - Operation
  - Carryin
- Output
  - Result
  - Overflow





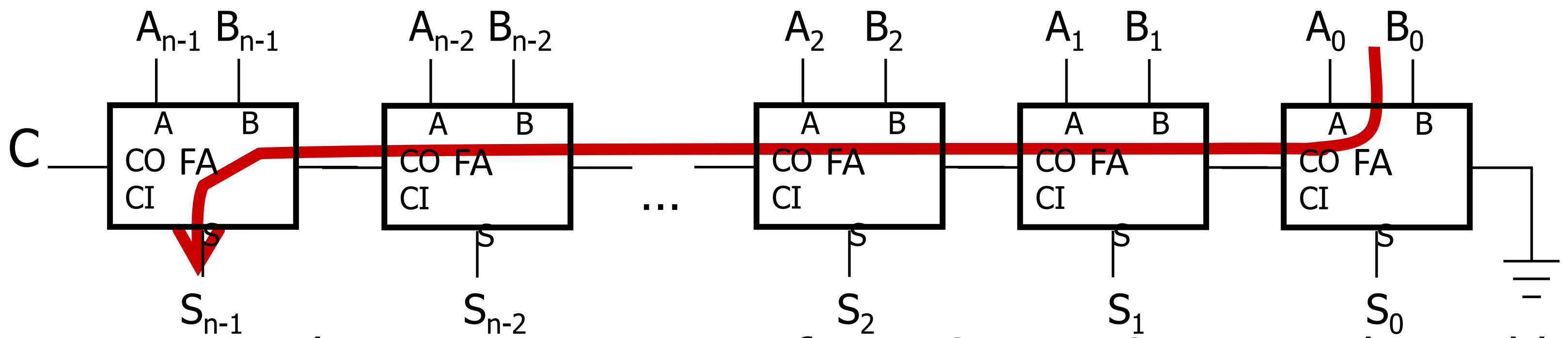
行波进位的缺点　　并行加法器－－串行进位

行波进位加法器（Ripple Carry Adder)

- 加位将从最小位(LSB)传播到最大位(MSB)
- N位加法器在最坏情况下的延迟: 2N个门延迟

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR}$

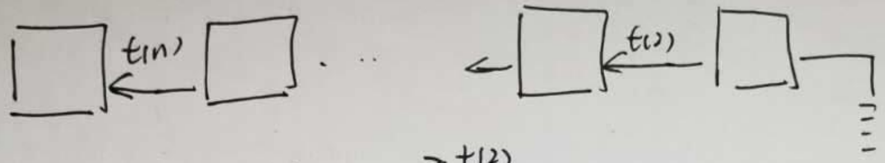
$\approx \Theta(N)$

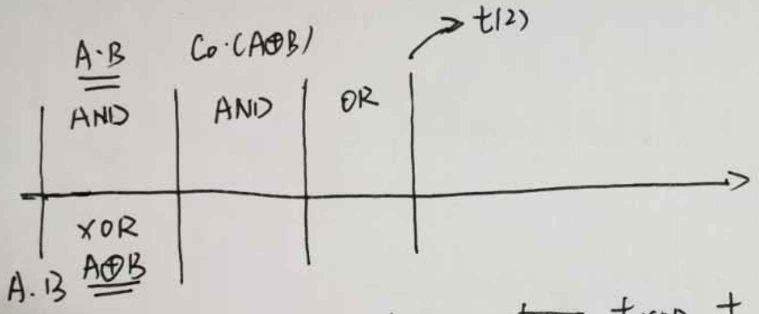
$\Theta(N)$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

t(n)





$$\Rightarrow t(2) = \cancel{t_{AND}}\; t_{XOR} + t_{AND} + t_{OR}$$

$$t(n) = t(n-1) + \boxed{t_{OR} + t_{AND}}$$

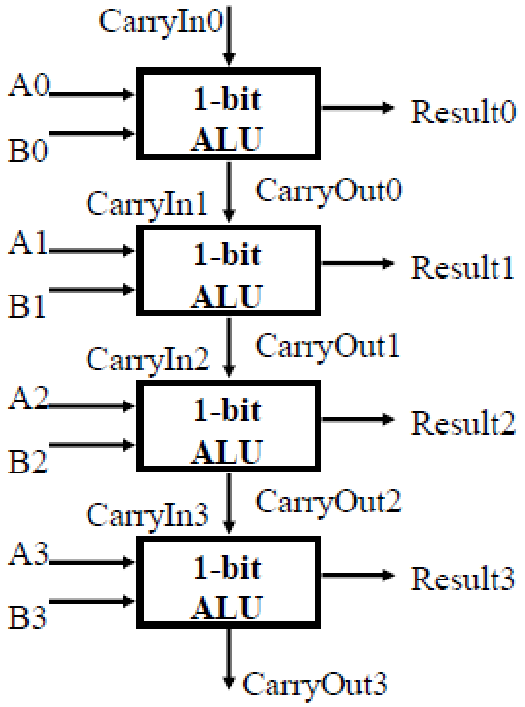
$$\Rightarrow t(n) = (t_{OR} + t_{AND})\cdot(n-2) + t(2)$$

$$= \cancel{t_{O}}\; t_{XOR} + t_{AND} + t_{OR} + (n-2)(t_{OR} + t_{AND})$$

$$\Rightarrow t_{PD} = t(n) + t_{OR}\; \cancel{= t_X}$$

但是，性能如何呢?

- n位行波进位加法器的关键路径为CP



Carry Propagation

设计技巧: 减少关键路径上的硬件延迟

Fast adders

并行加法器－－并行进位（或先行进位）

- All functions can be represented in 2-level logic.
- But:
  - The number of inputs of the gates would drastically rise
- Target:

Optimum between speed and size

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

令： Generate Carry at Bit i:

gi = Ai * Bi

Propagate Carry via Bit i:

pi = Ai xor Bi

并行进位的特点

同时产生进位

加法延时缩短

实现相对复杂

Cin1=g0+(p0*Cin0)
Cin2=g1+(p1*g0)+(p1*p0*Cin0)
Cin3=g2+(p2*g1)+(p2*p1*g0)+(p2*p1*p0*Cin0)

先行进位（Carry Lookahead）的理论基础续

$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

- Now define two new terms:
  - Generate Carry at Bit i: gi = Ai * Bi
  - Propagate Carry via Bit i: pi = Ai xor Bi

第i位产生的进位 gi = Ai & Bi
通过第i位的传播进位(Propagate Carry) pi = Ai Xor Bi

- We can rewrite:
  - Cin1=g0+(p0*Cin0)
  - Cin2=g1+(p1*g0)+(p1*p0*Cin0)
  - Cin3=g2+(p2*g1)+(p2*p1*g0)+(p2*p1*p0*Cin0)
- Carry going into bit 3 is 1 if

•进入第3位的进位是1，如果在第2位，产生了进位(g2)

•或者在第1位产生了进位(g1)并且 第2位传播了这个进位(p2 & g1)

•或者在第0位产生了进位(g0)并且 第1位和第2位都传播了这个进位(p2 & p1 & g0)

•或者在第0位有输入进位(Cin0) 并且 第0位、第1位和第2位都传播了这个进位(p2 & p1 & p0 & Cin0)

Four bit carry look ahead adder

- c1 = g0 + (p0 * c0)
- c2 = g1 + p1*c1 = g1 + (p1 * g0) + (p1 * p0 * c0)
- c3 = g2 + p2*c2 = g2 + (p2 * g1) + (p2 * p1 * g0) + (p2 * p1 * p0 * c0)
- c4 = g3 + p3*c3 = g3 + (p3 * g2) + (p3 * p2 * g1)
  +(p3 * p2 * p1 * g0) + (p3 * p2 * p1 * p0 * c0)

COMMENT:

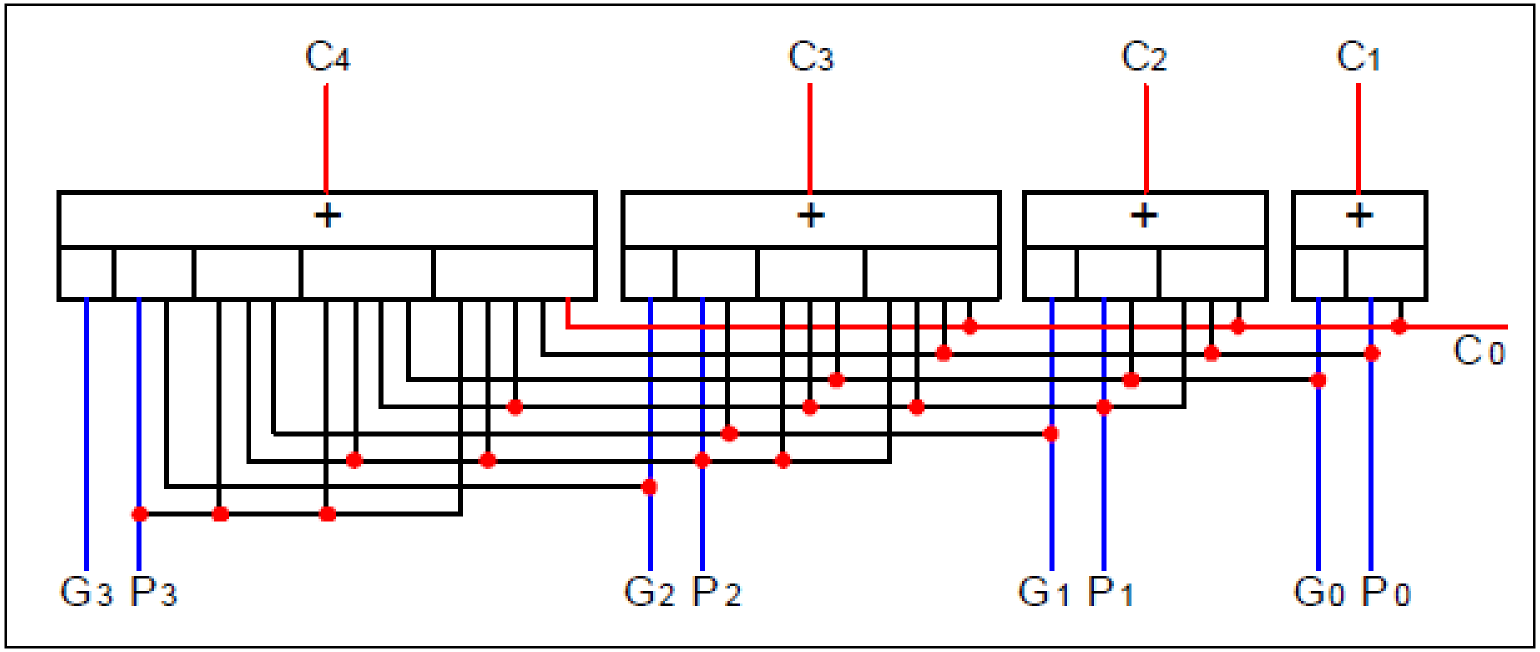
This kind of adder will be faster than the ripple carry adder, and smaller than the adder with the tow-level logic.

PROBLEM:

If the number of the adder bits is very large,this kind of adder will be too large. So we must seek more efficient ways.

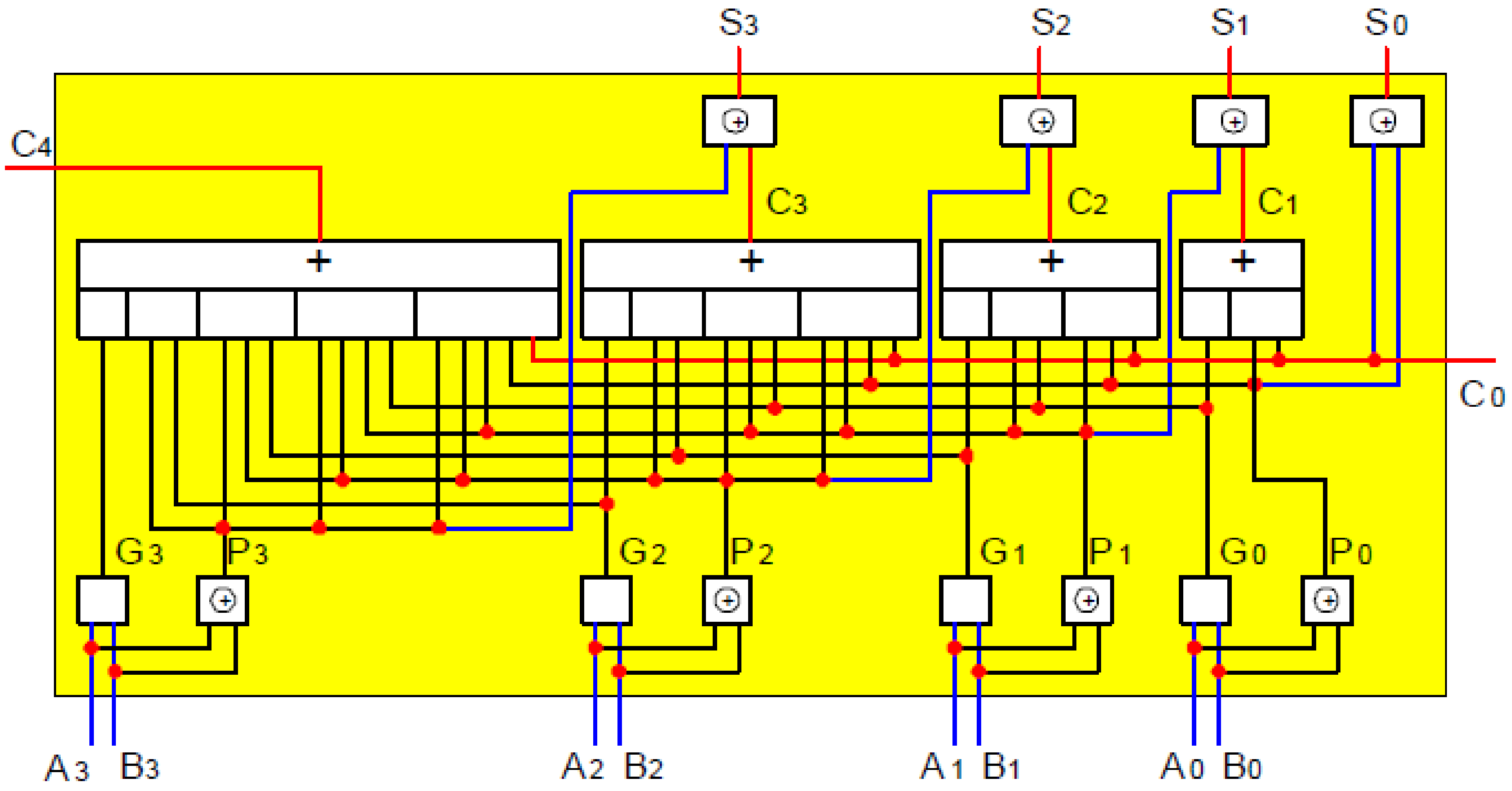
设置P和G两个先行进位输出端

并行进位链



设置P和G两个先行进位输出端

□并行进位加法器

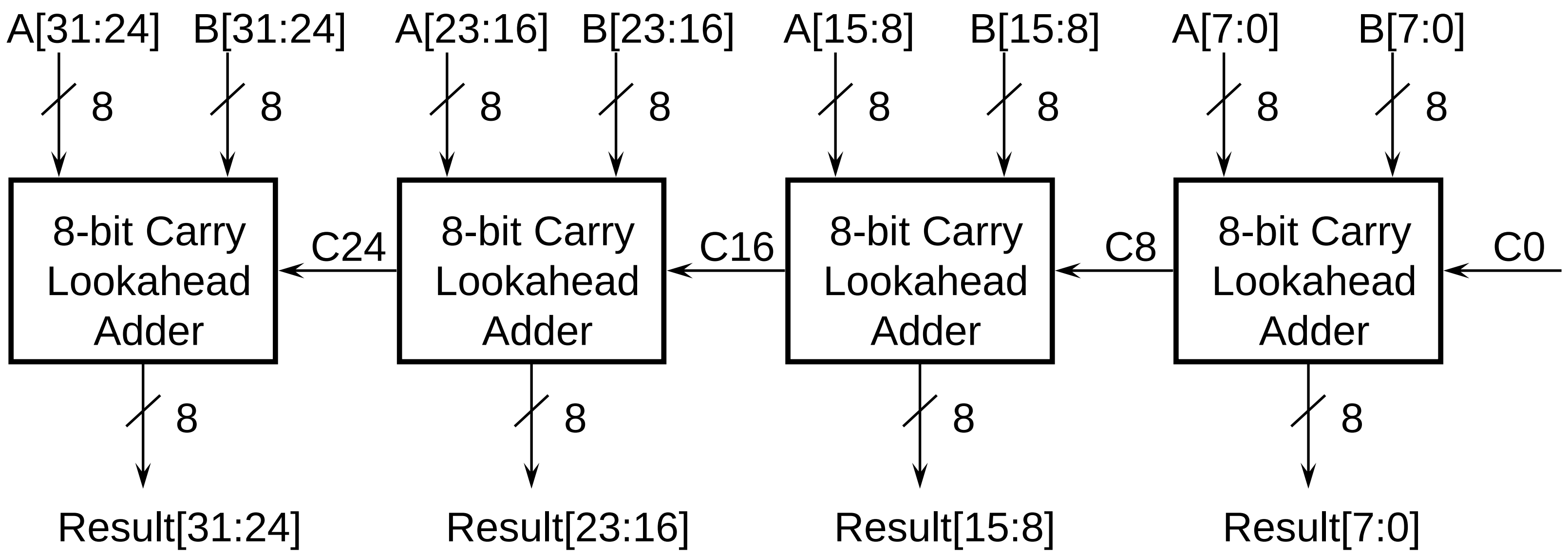


$$C_o = C_i \cdot (A \oplus B) + A \cdot B$$

$$S = C_i \oplus (A \oplus B)$$

局部(级联)先行进位加法器（Partial Carry Lookahead Adder or Cascaded Carry Lookahead）

- 实现全先行进位加法器的成本太高
  - •想象Cin31的逻辑方程的长度
- 一般性经验:
  - •连接一些N位先行进位加法器，形成一个大加法器
  - •例如: 连接4个8位进位先行加法器，形成1个32位局部先行进位加法器

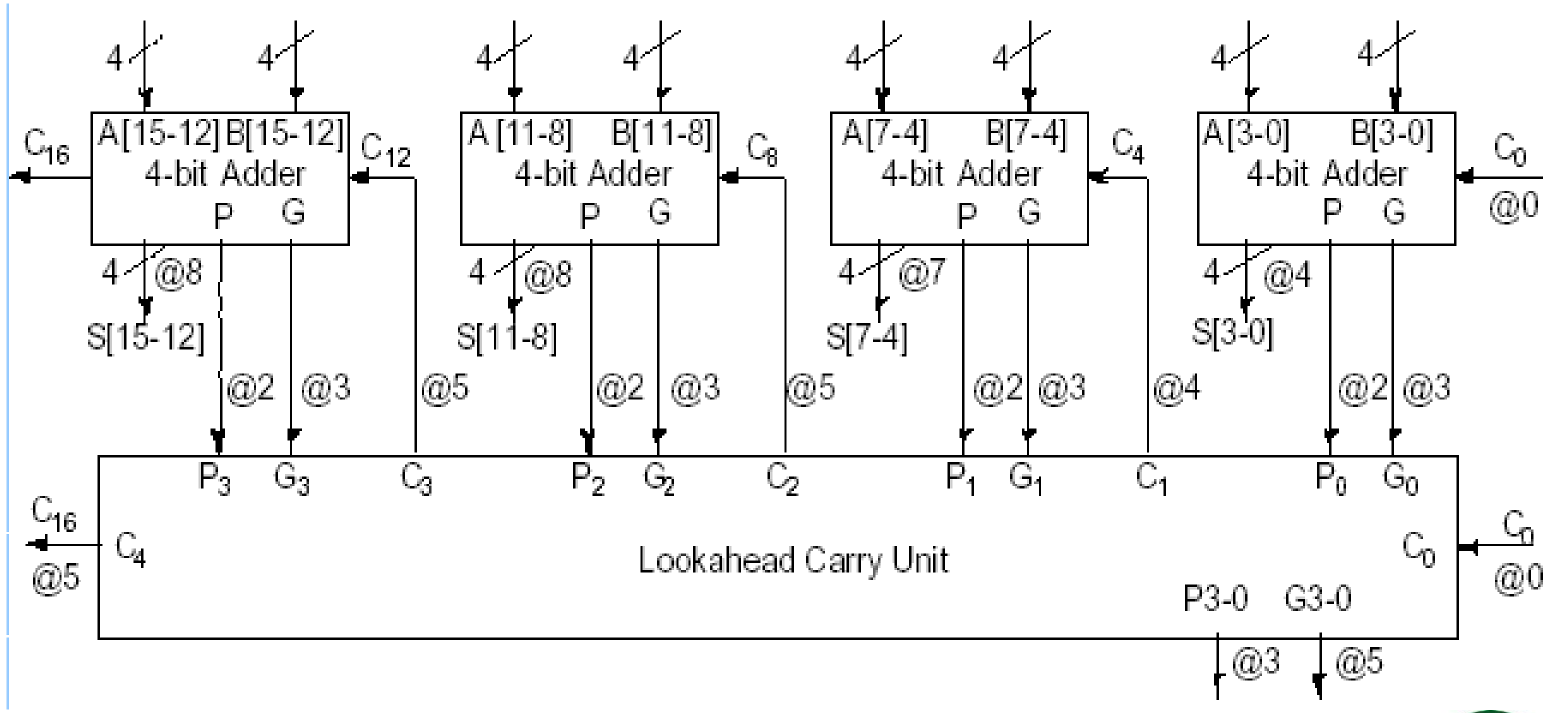


分组并行进位加法器(组内并行，组间传递)

Hybrid CLA + Ripple carry

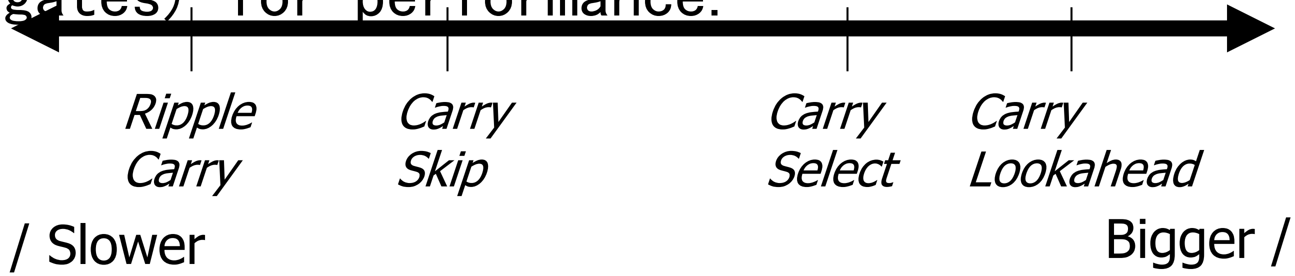
分组并行进位加法器（组内并行，组间并行）

- Realisation: Ripple carry adders and
  - Carry look ahead logic

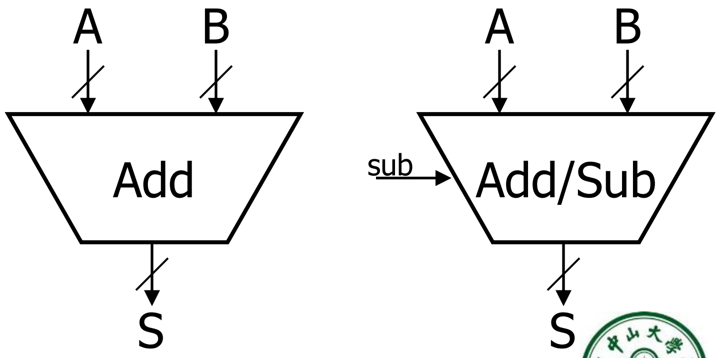


Adder Summary

- Adding is not only common, but it is also tends to be one of the most time-critical of operations
  - As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.

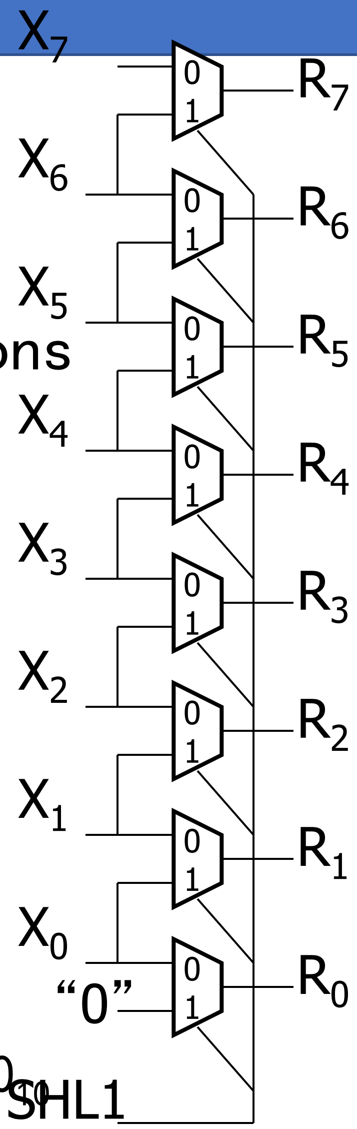


At this point we' ll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.



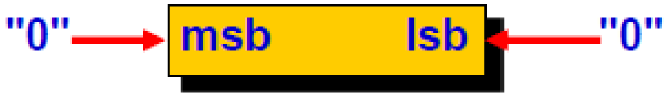
Shifting Logic

- Shifting is a common operation
  - applied to groups of bits
  - used for alignment
  - used for “short cut” arithmetic operations
    - X << 1 is often the same as 2*X
    - X >> 1 can be the same as X/2
- For example:
  - $X = 20_{10} = 00010100_2$
  - Left Shift:
    - $(X << 1) = 00101000_2 = 40_{10}$
  - Right Shift:
    - $(X >> 1) = 00001010_2 = 1010$
  - Signed or “Arithmetic” Right Shift:
    - $(-X >>> 1) = (11101100_2 >>> 1) = 11110110_2 = -10$

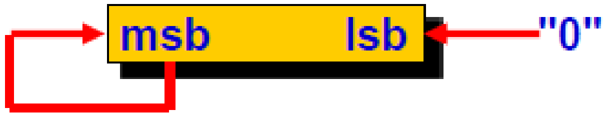


三种移位器

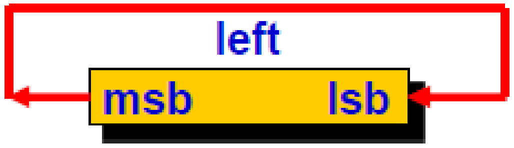
逻辑(*logical*) — 移入的数值总是 "0"

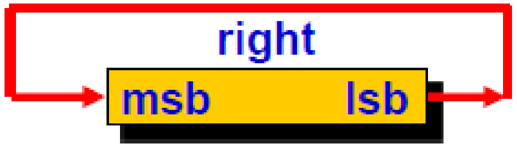


算术(*arithmetic*) — 在右移时，进行符号扩展

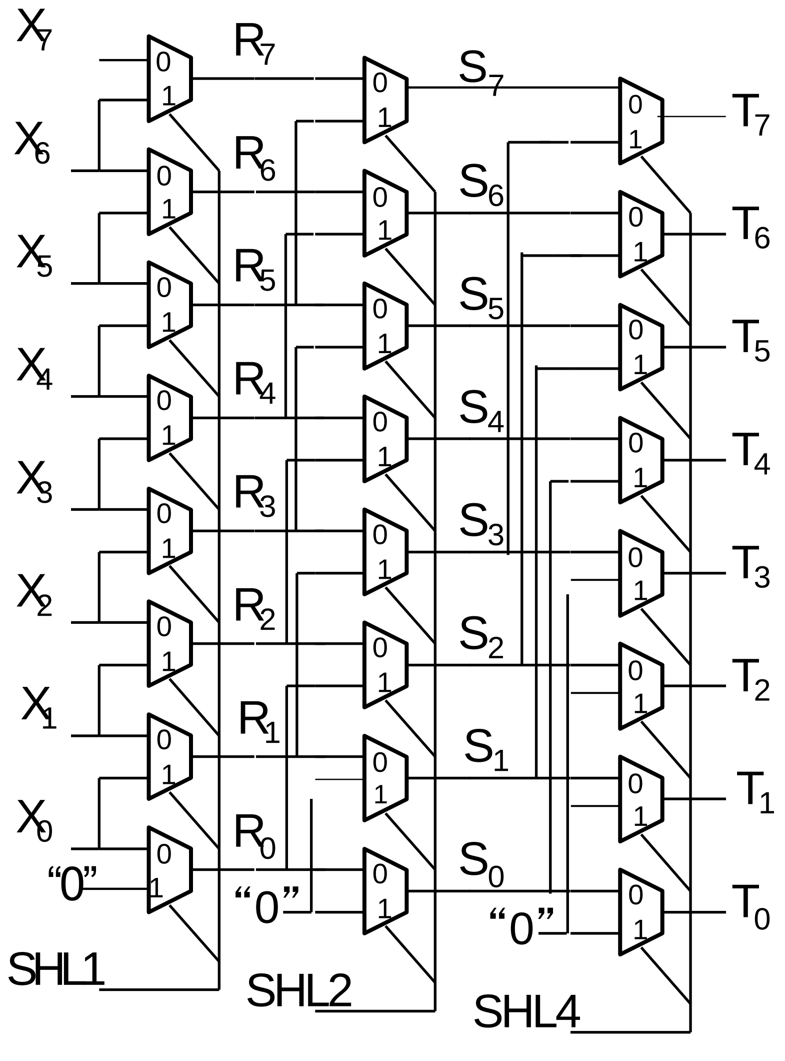


循环(*rotating*) — 移出的位又从另一端移入 (在MIPS中不支持)





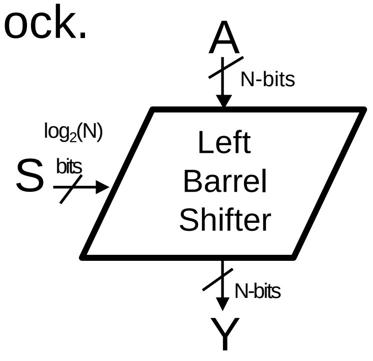
注：需要一位移动。特定指令可能要求进行0⇒32位移动！



If we connect our "shift-left -two" shifter to the output of our "shift-left-one" we can shift by 0, 1, 2, or 3 bits.

And, if we add one more "shift-left-4" shifter we can do any shift up to 7 bits!

So, let's put a box around it and call it a new functional block.

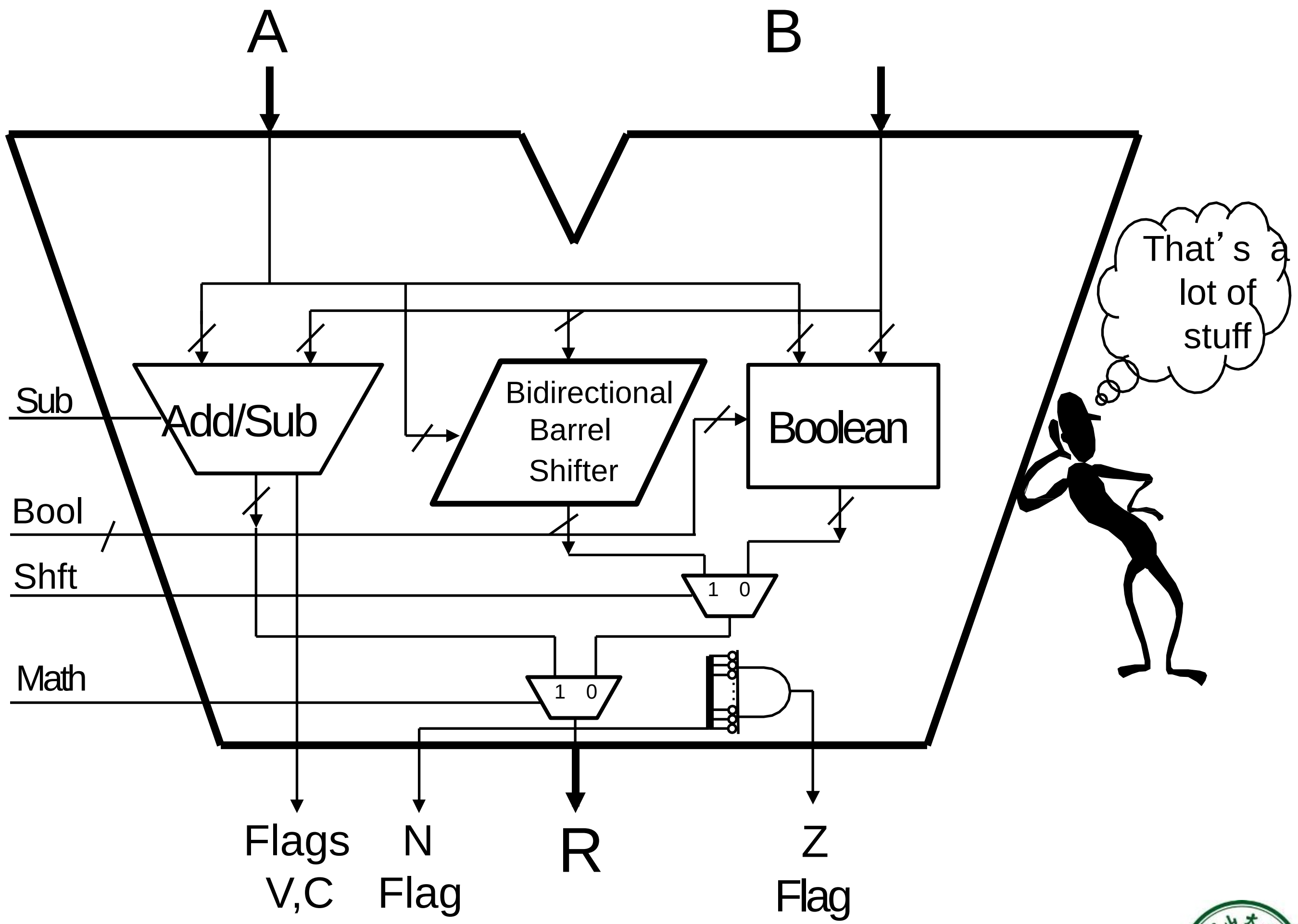


Boolean Operations

- It will also be useful to perform logical operations on groups of bits. Which ones?
  - ANDing is useful for "masking" off groups of bits.
    - ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)
  - ANDing is also useful for "clearing" groups of bits.
    - ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)
  - ORing is useful for "setting" groups of bits.
    - ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)
  - XORing is useful for "complementing" groups of bits.
    - ex. 10101110 ^ 00001111 = 10100001 (1's invert last 4 bits)
  - NORing is useful for.. uhm…
    - ex. 10101110 # 00001111 = 01010000 (0's invert, 1's clear)

An ALU, at Last

We give the “Math Center” of a computer a special name-- the Arithmetic Logic Unit. For us, it just a big box!



Summary

- We will use a subset of MIPS instruction set in this class
  - Sometimes called “miniMIPS”
  - All instructions are 32-bit
  - 3 basic instruction formats
    - R-type - Mostly 2 source and 1 destination register
    - I-type - 1-source, a small (16-bit) constant, and a destination register
    - J-type - A large (26-bit) constant used for jumps
  - Load/Store architecture
  - 31 general purpose registers, one hardwired to 0, and, by convention, several are used for specific purposes.
- ISA design requires tradeoffs, usually based on
  - History, Art, Engineering
  - Benchmark results

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

2022计算机组成...
群号：777625409
扫一扫二维码，加入群聊。