Very Large Scale Integration II - VLSI II

Hardware Arithmetic-1

Prof. Dr. Berna Örs Yalçın

Araş. Gör. Ahmet Çağrı Bağbaba

Araş. Gör. Ercan Kalalı

Araş. Gör. Latif Akçay

Araş. Gör. Y. Fırat Kula

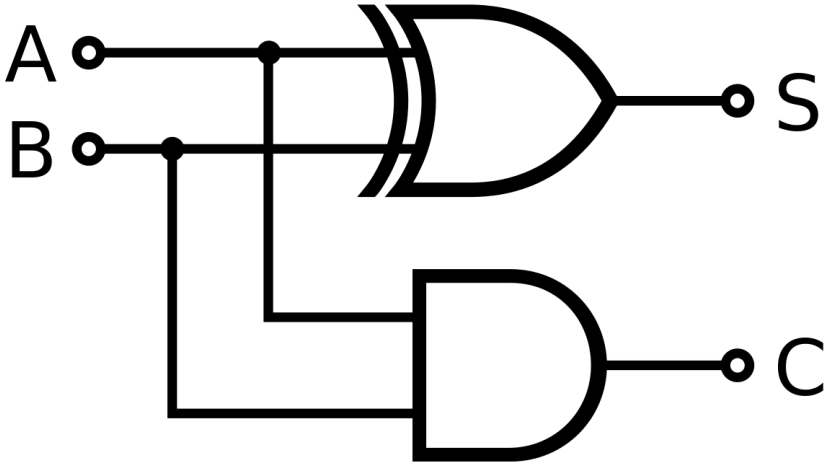
Araş. Gör. Serdar Duran

ITU VLSI Laboratories

Istanbul Technical University

Adder Circuits

Half Adder

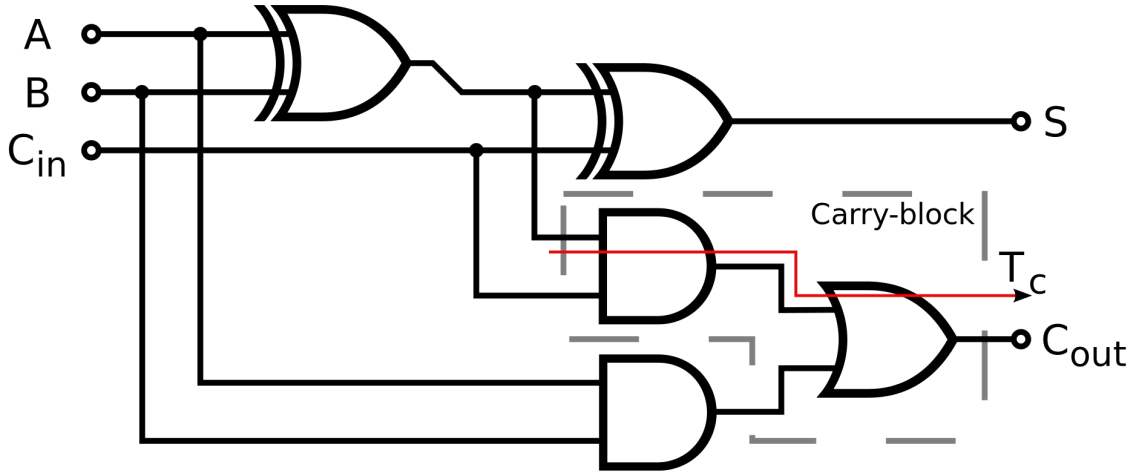


A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Sum = A ⊕ B

Carry = A . B

Full Adder

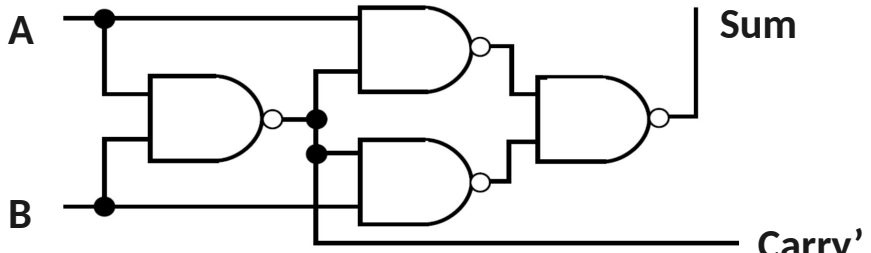


A	B	Cin	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum = A ⊕ B

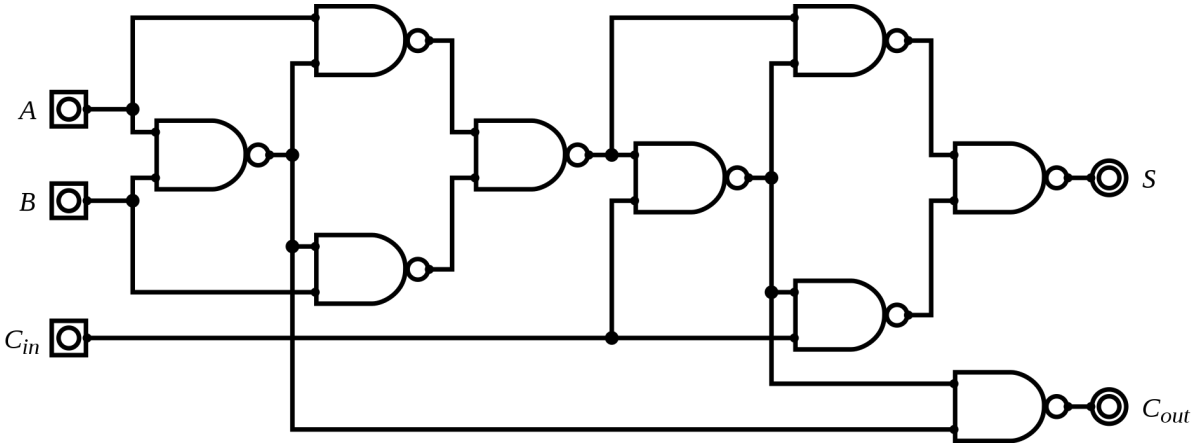
Carry = A . B + C_{in} . (A ⊕ B)

CMOS Implementations



Half - Adder

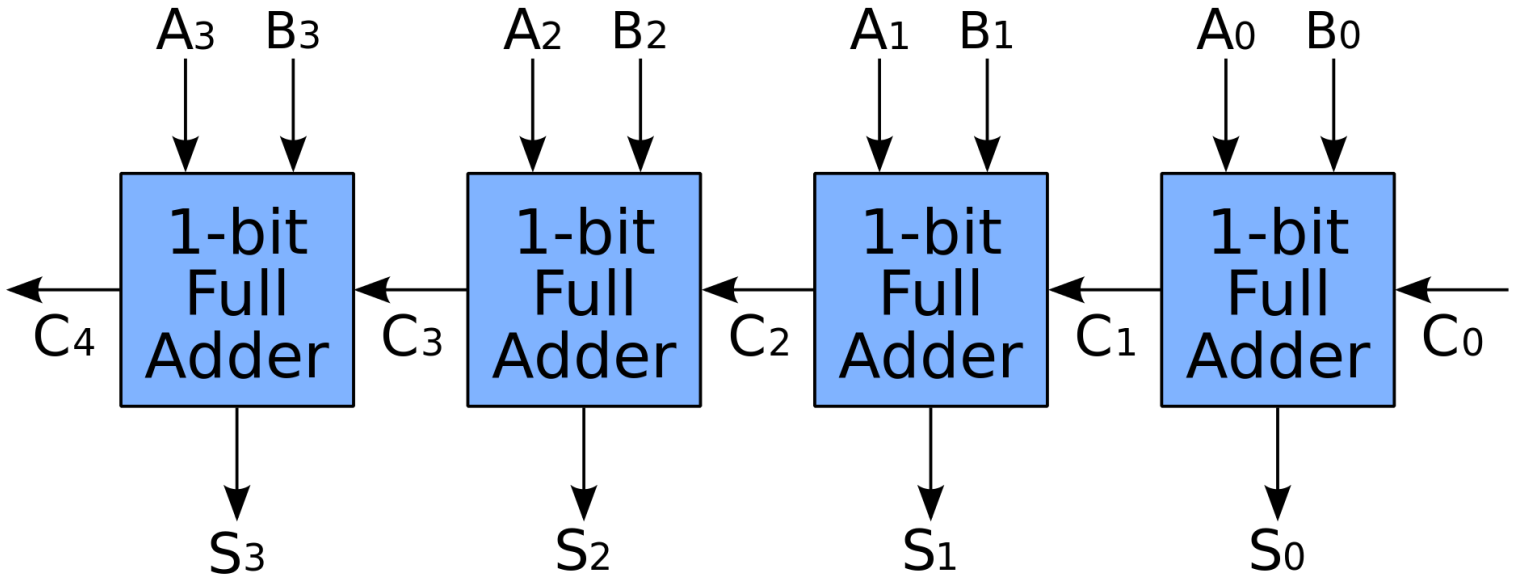
Latency = T_{HA} = 3 . T_{nand}



Full - Adder

Latency = T_{FA} = 5 . T_{nand}

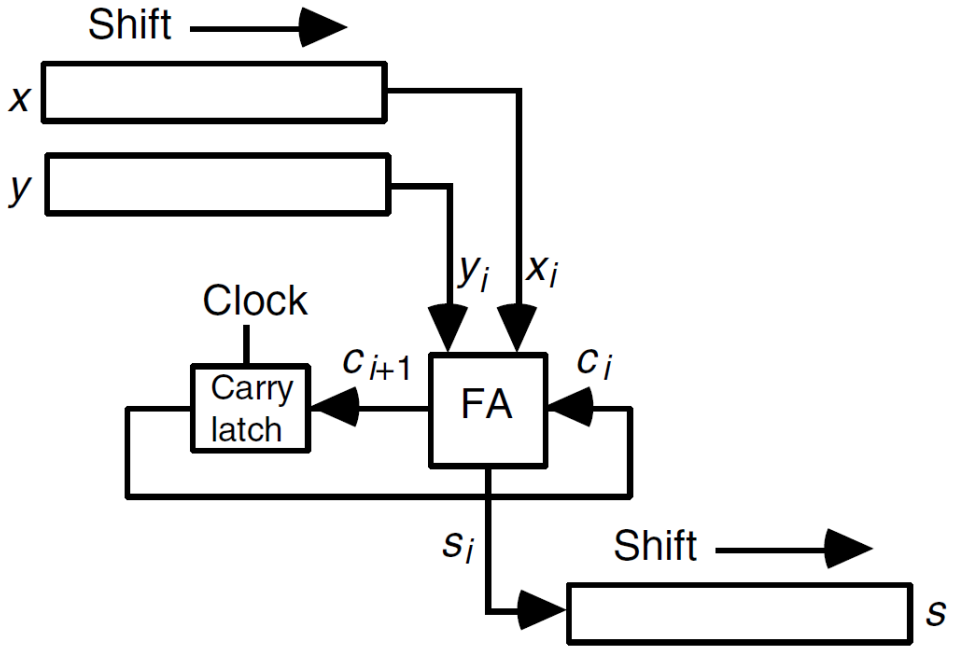
Ripple Carry Adder



***n*-bit numbers => *n* FAs**

Latency $\approx n \cdot T_{FA} = n \cdot T_{carry}$

Bit-serial Adder



***n*-bit numbers => *n* clock cycles**

Latency linear with n = O(n)

2's Complement Adder

- For 2's-complement addition, overflow occurs when two numbers of same sign are added and a result of the opposite sign is produced.

Overflow = x . y . s' + x' . y' . s

Overflow = $c_n \oplus c_{n-1}$

x = 0 1 1 1 (+7)
y = 0 1 1 0 (+6)

Sum = 1 1 0 1 (-3)
Cout = 0
Overflow = 1

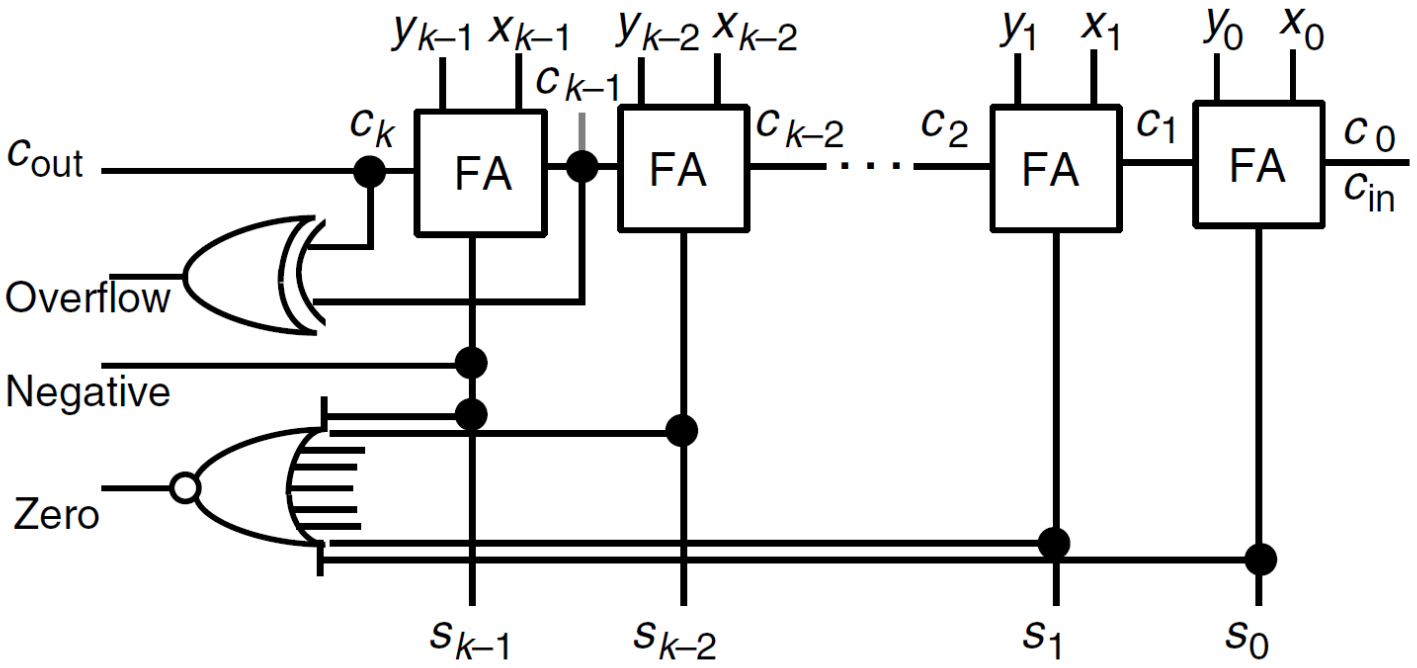
Correct result = 0 1 1 0 1 (+13)

x = 1 0 0 0 (-8)
y = 1 0 0 1 (-7)

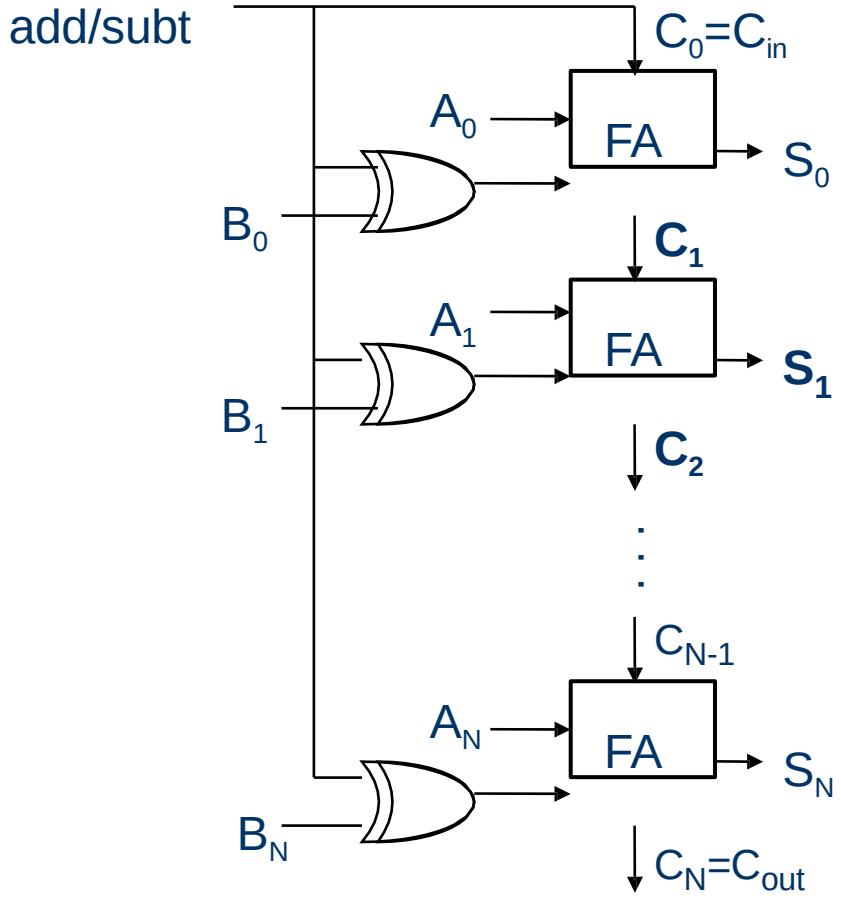
Sum = 0 0 0 1 (+1)
Cout = 1
Overflow = 1

Correct result = 1 0 0 0 1 (-15)

2's Complement Adder



Ripple Carry Adder / Subtractor

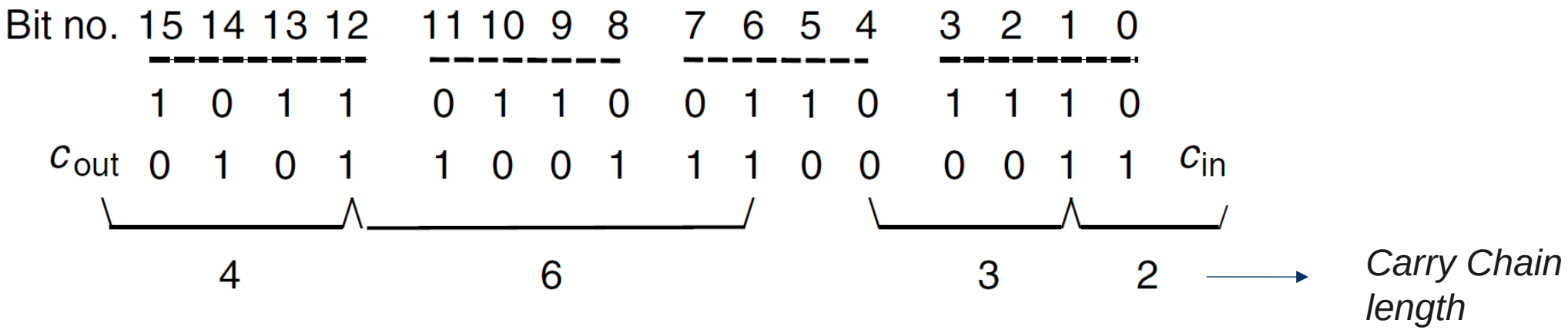


Subtraction:

complement all subtrahend bits (XOR) and set Cin to 1

A – B = A + (B + 1)

Carry Chains



- The length of a **carry chain** is the number of digit positions from where the carry is generated up to where it is finally absorbed or annihilated.

Carry Chains

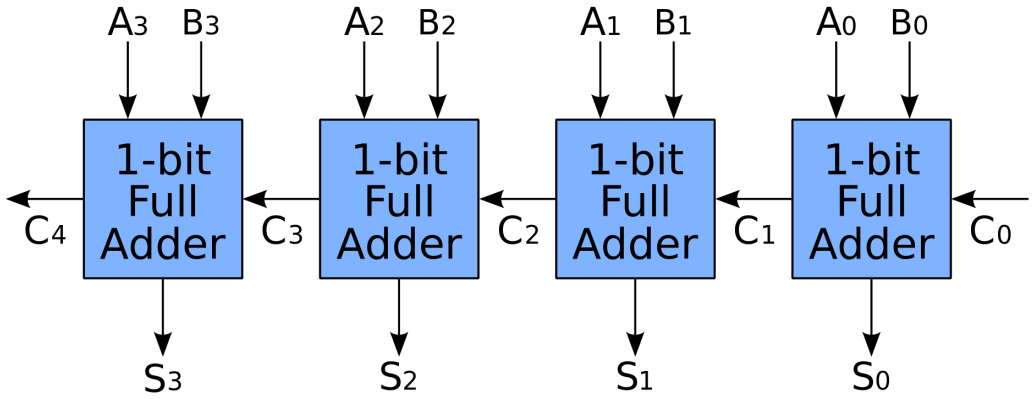
- The expected length of the carry chain that starts at bit position i is:

$$2 - 2^{-(k-i-1)}$$

- For i << k, length ≈ 2 . Therefore it is expected that carry chains are usually quite short.
- The longest carry chain in an adder for k-bit numbers is of length $\log_2 k$ on average (expected value).

Carry Chains

- Ripple-carry adders are the simplest and slowest adder designs.
- For k-bit operands, both the worst-case delay and the implementation cost of a ripple-carry adder are linear in k.
- However worst-case carry-propagation chain of length k almost never realizes.



***Cost* = *n* FAs**

$$Latency \approx n \cdot T_{FA}$$

$$= O(n)$$

Carry Chains

- The key point to fast addition is a low-latency carry network
- For the design of a carry network, what matters is whether in a given position a carry is generated, propagated, or annihilated (absorbed).

$$s_i = x_i \oplus y_i \oplus c_i$$

$$g_i = x_i \cdot y_i$$

$$p_i = x_i \oplus y_i$$

$$a_i = (x_i \cdot y_i)'$$

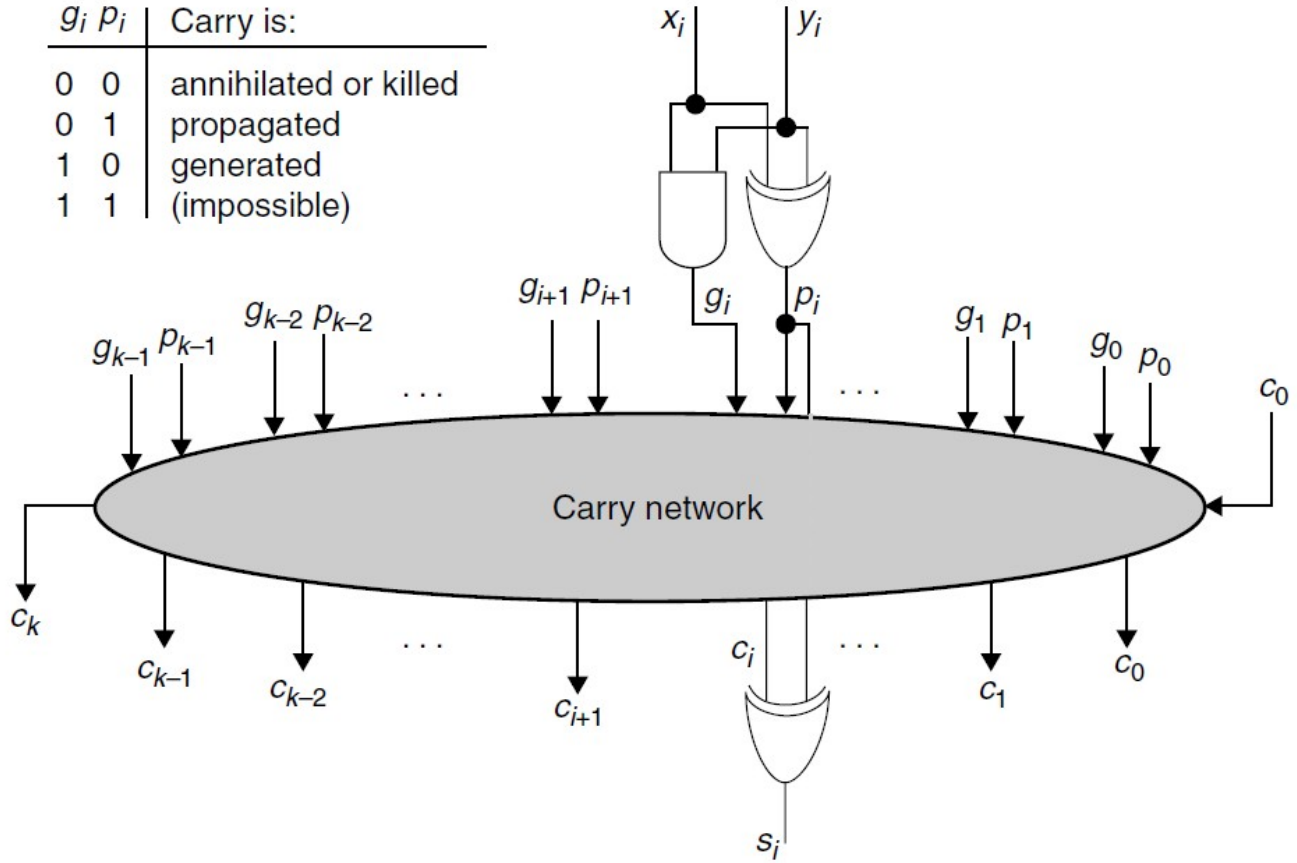
Carry Generate

Carry Propagate

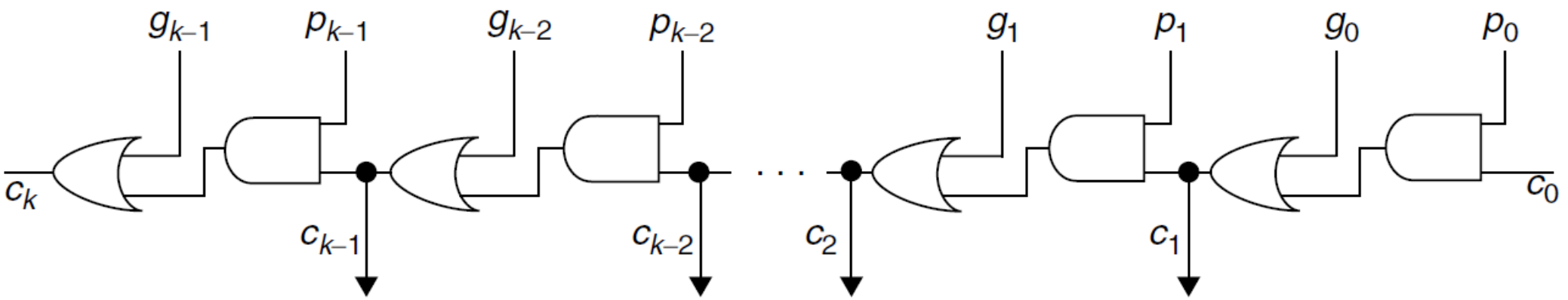
Carry Annihilate

Generic Adder Structure

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)



Generic Adder Structure

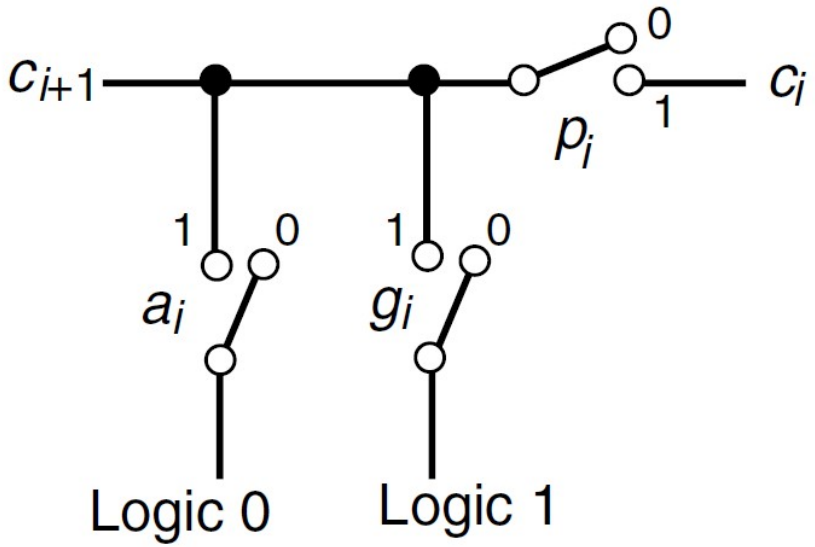


Alternate view of a Ripple-Carry Network

Manchester Carry Chain Adder

- The **carry recurrence** forms the basis of a simple carry network known as Manchester carry chain.

$c_{i+1} = g_i + c_i \cdot p_i$ ⟶ Carry Recurrence

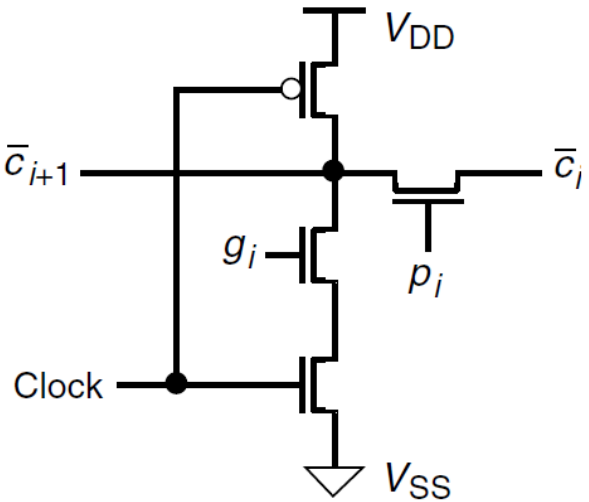


Latency = propagation delay through k switches in the worst case.

Manchester Carry Chain Adder

- Manchester Adders are suitable for short chains (up to 8 bits) in which fast addition and improved performance can be achieved.
- Switch delays are lower than gate delays as a result Manchester Carry Adders outperforms RCAs.

Latency linear with k = O(k)



Cmos Implementation

Carry-Lookahead Adders

- The method is calculation of the carry recurrences beforehand. (unrolling)

$s_i = p_i \oplus c_{i-1}$

$c_{i+1} = g_i + c_i \cdot p_i$

Sum
Carry Recurrence

$g_i = x_i \cdot y_i$

$p_i = x_i \oplus y_i$

$a_i = (x_i \cdot y_i)'$

Carry Generate
Carry Propagate
Carry Annihilate

$$
\begin{aligned}
c_i &= g_{i-1} + c_{i-1} \cdot p_{i-1} \\
&= g_{i-1} + (g_{i-2} + c_{i-2} \cdot p_{i-2}) \cdot p_{i-1} = g_{i-1} + g_{i-2} \cdot p_{i-1} + c_{i-2} \cdot p_{i-2} \cdot p_{i-1} \\
&= g_{i-1} + g_{i-2} \cdot p_{i-1} + g_{i-3} \cdot p_{i-2} \cdot p_{i-1} + c_{i-3} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1} \\
&= g_{i-1} + g_{i-2} \cdot p_{i-1} + g_{i-3} \cdot p_{i-2} \cdot p_{i-1} + g_{i-4} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1} + c_{i-4} \cdot p_{i-4} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1}
\end{aligned}
$$

Carry-Lookahead Adders

- The unrolling can be continued until the last product term contains $c_0 = c_{in}$.
- The interpretation:
  - carry enters into position i if and only if a carry is generated in position i −1 (g_{i-1}), or a carry generated in position i − 2 is propagated by position i − 1 ($g_{i-2} . p_{i-1}$), or a carry generated in position i − 3 is propagated at i − 2 and i − 1 ($_{gi-3} . p_{i-2} . p_{i-1}$), etc.

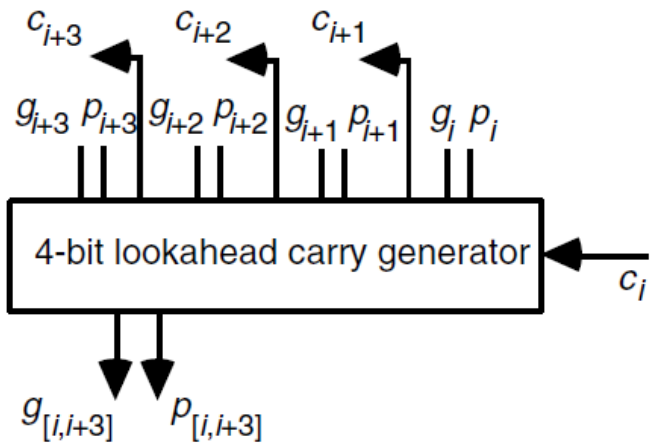
Carry-Lookahead Adders

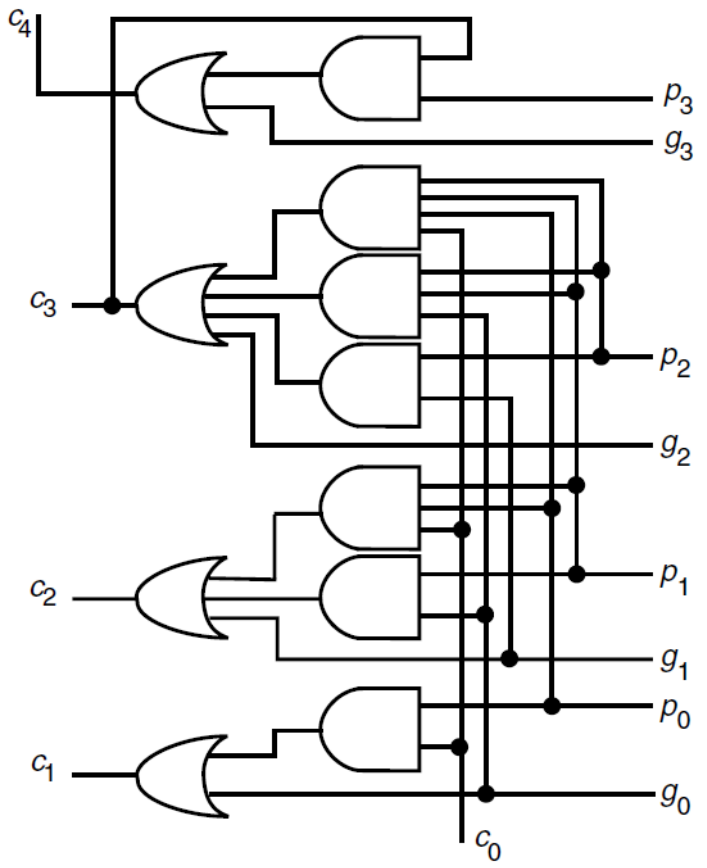
- After full unrolling, all the carries in a k-bit adder can be computed directly from the auxiliary signals (gi, pi).
- For k = 4:

$$c4 = g3 + g2 \,.\, p3 + g1 \,.\, p2 \,.\, p3 + g0 \,.\, p1 \,.\, p2 \,.\, p3 + c0 \,.\, p0 \,.\, p1 \,.\, p2 \,.\, p3$$
$$c3 = g2 + g1 \,.\, p2 + g0 \,.\, p1 \,.\, p2 + c0 \,.\, p0 \,.\, p1.\, p2$$
$$c2 = g1 + g0 \,.\, p1 + c0 \,.\, p0.\, p1$$
$$c1 = g0 + c0 \,.\, p0$$

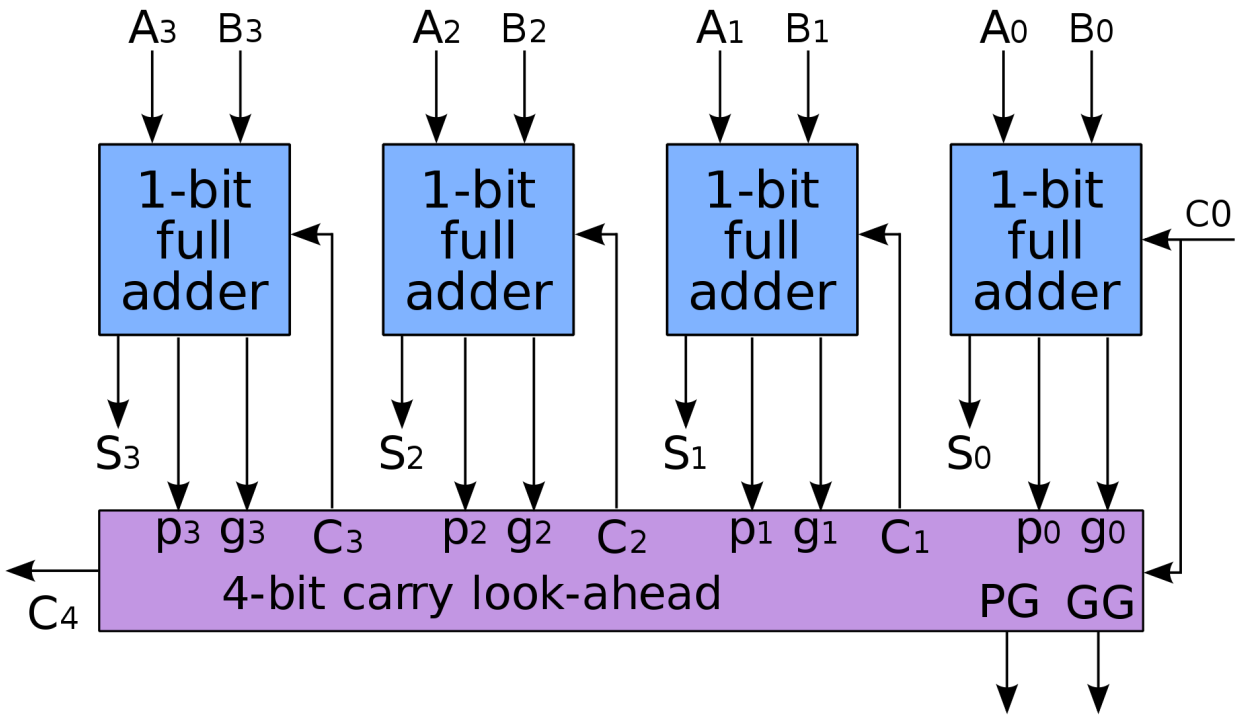
Carry-Lookahead Adders

- Full carry lookahead is impractical for wide words. The fully unrolled carry equation for c31, for example, consists of 32 product terms.



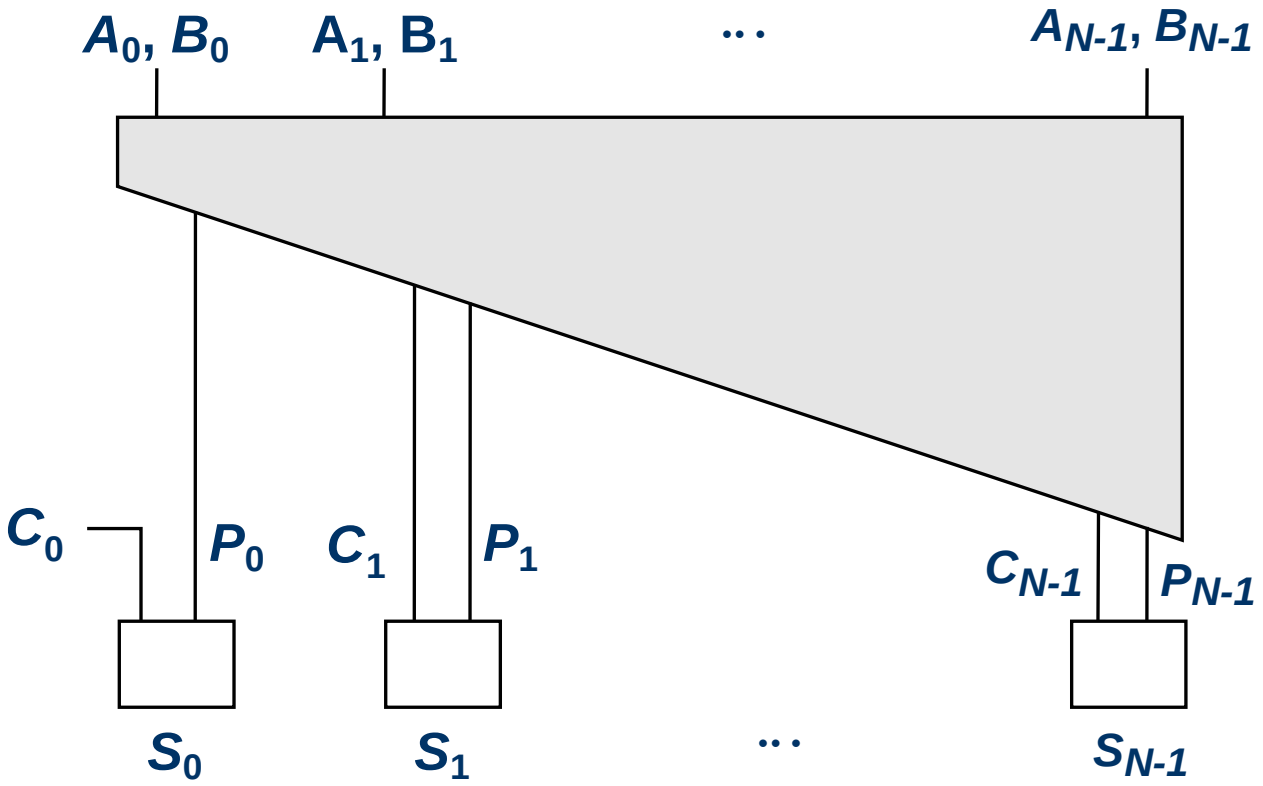


Carry-Lookahead Adders



$$PG = p3 \,.\, p2 \,.\, p1 \,.\, P0$$
$$GG = g3 + g2 + g1 + g0$$
$$c4 = GG + PG \,.\, c0$$

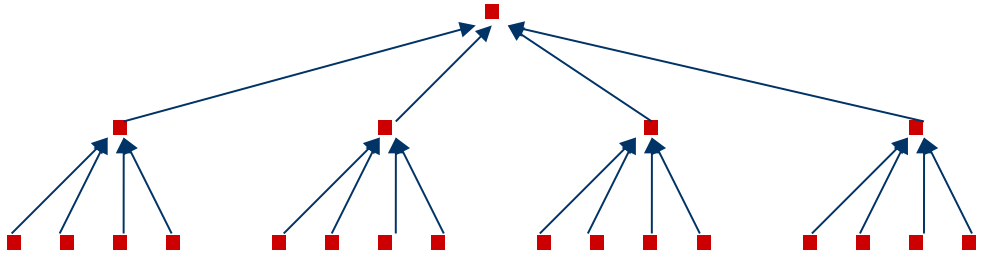
Carry-Lookahead Adders



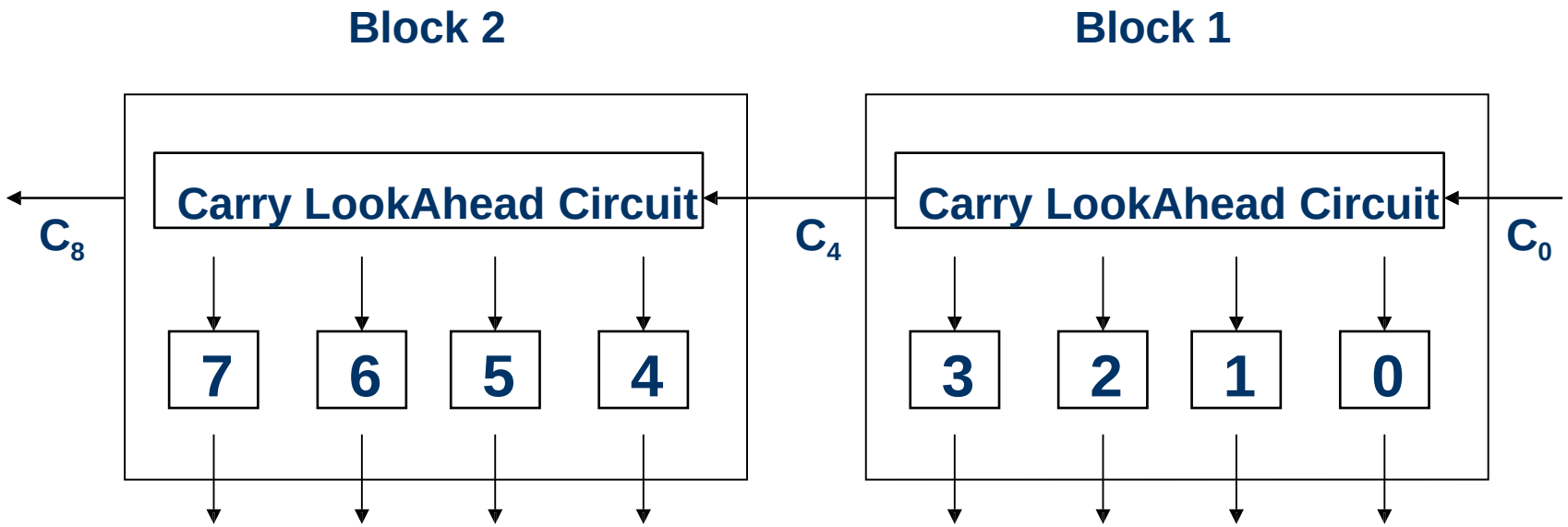
Divide and Conquer

- A Carry-Lookahead Adder, can be broken into groups (of 4) and each group computes its group-generate and group-propagate.

- For example, to add 32 numbers, you can partition the task as a tree:

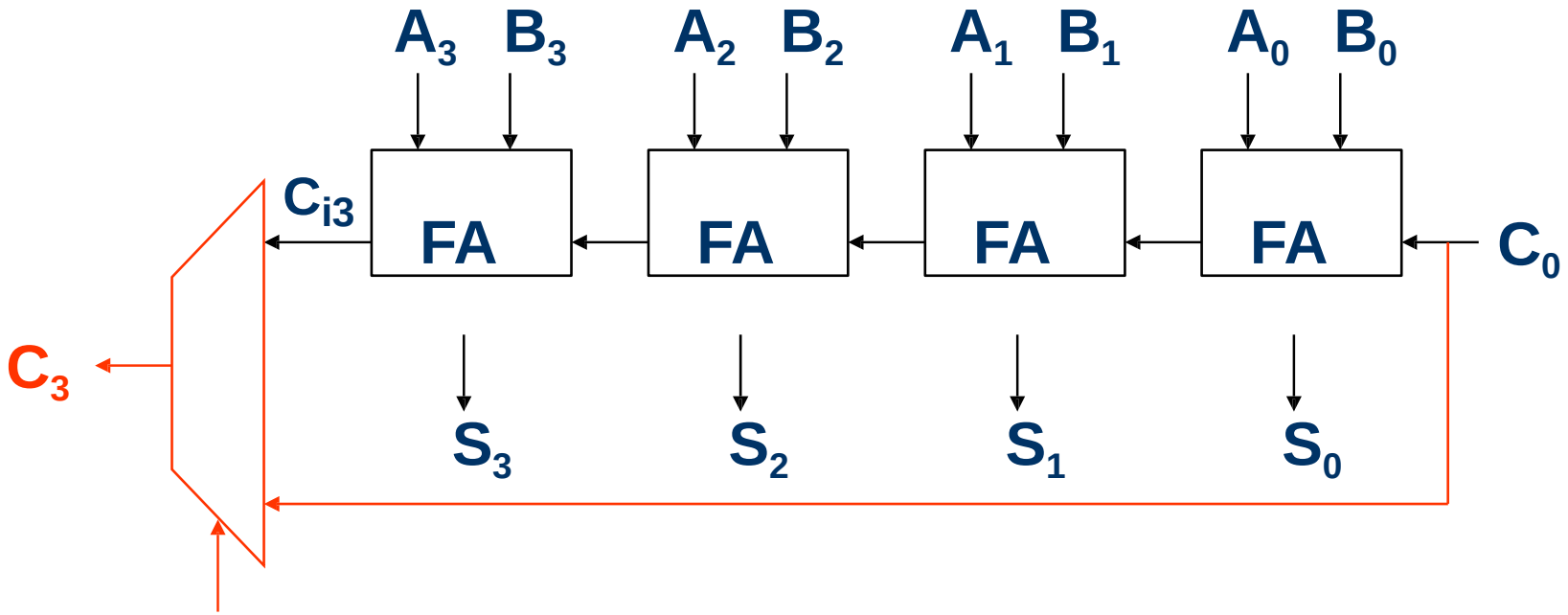


Carry LookAhead Adder



$$T\ (\text{N-bit adder}) \approx (N/4)\ T_{carry}$$

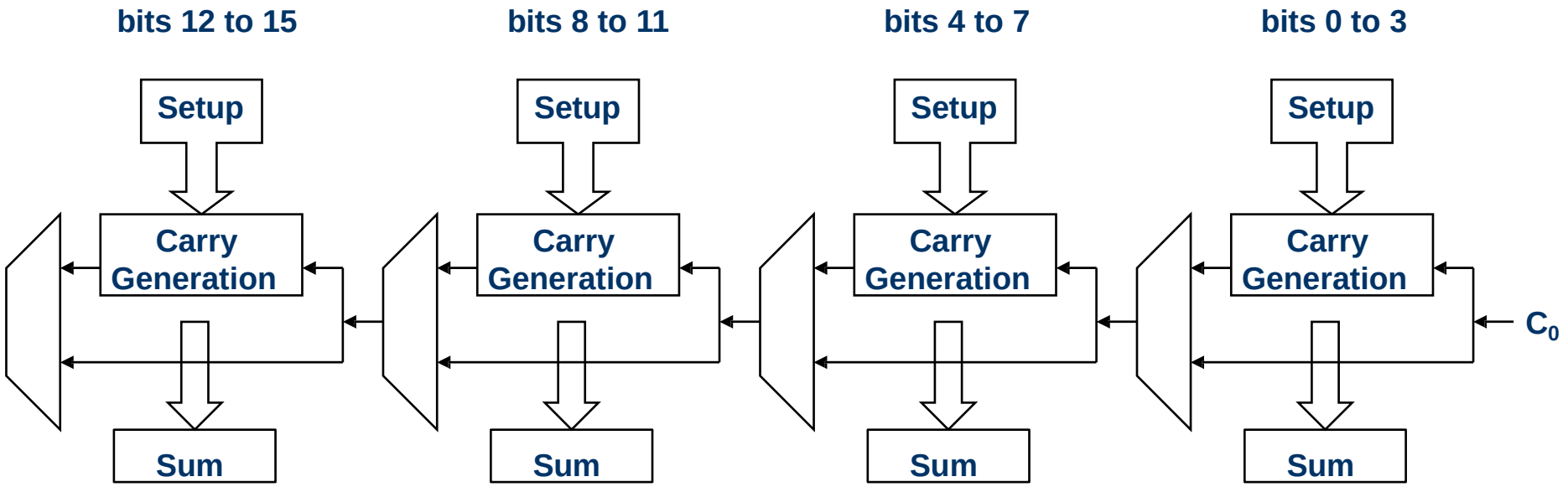
Carry-Skip (Carry-Bypass) Adder



$PG = P_0 P_1 P_2 P_3$ (Block Propagate)

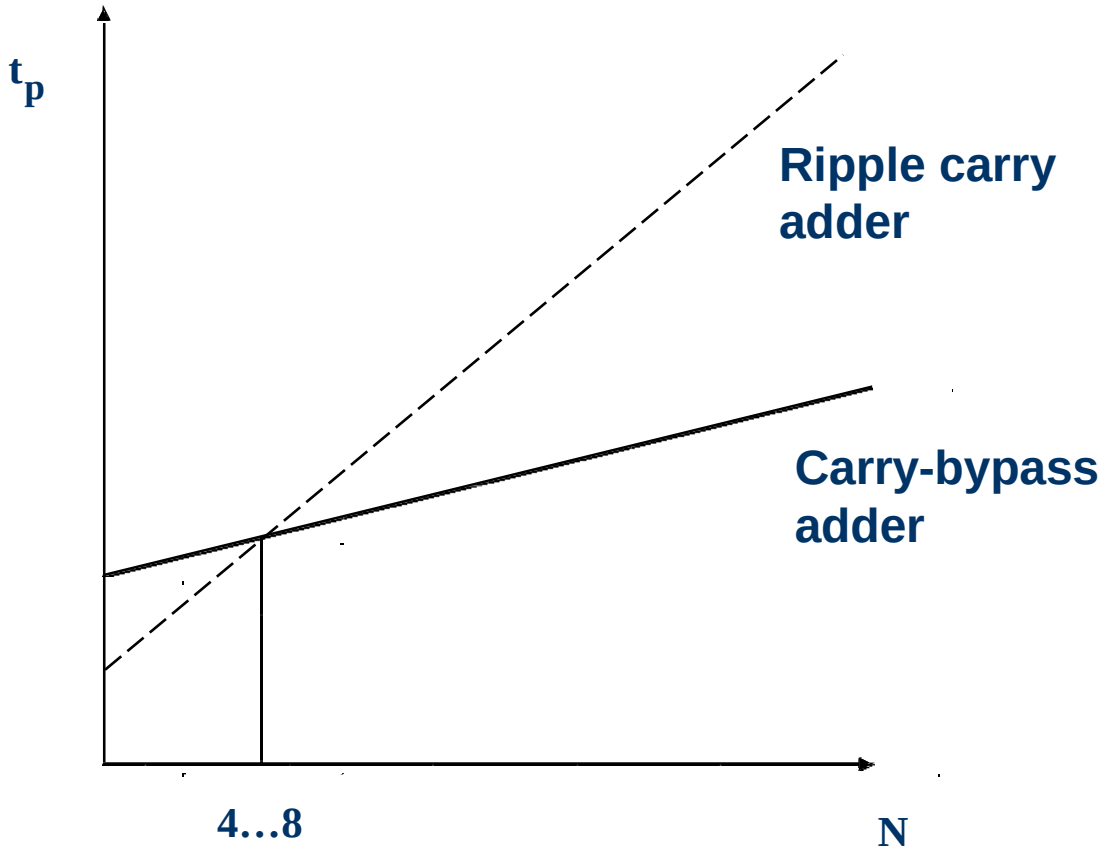
If ($P_0 P_1 P_2 P_3 = 1$) then $C_3 = C_0$ otherwise the block itself deletes or generates the carry internally

4-bit Block Carry-Skip Adder



$$t_{add} = t_{setup} + M\,t_{carry} + ((N/M) - 1)\,t_{mux} + (M-1)\,t_{carry} + t_{sum}$$

Ripple Carry vs Carry-Bypass Adder



Shifter Circuits

Shift Operations

Logical Shift

Shift right 1: $B_3B_2B_1B_0 \Rightarrow 0B_3B_2B_1$

Shift left 1: $B_3B_2B_1B_0 \Rightarrow B_2B_1B_0 0$

Arithmetic Shift

Shift right 1: $B_3B_2B_1B_0 => B_3B_3B_2B_1$

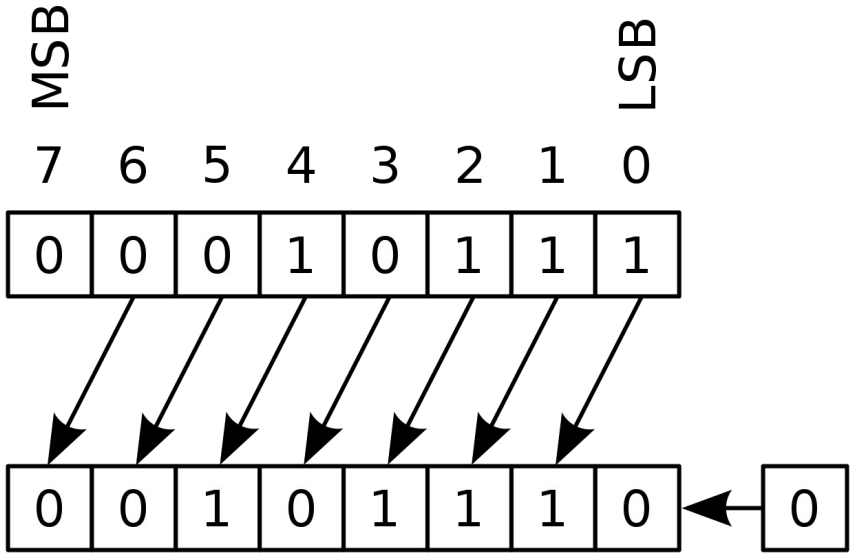
Shift left 1: $B_3B_2B_1B_0 => B_2B_1B_0 0$

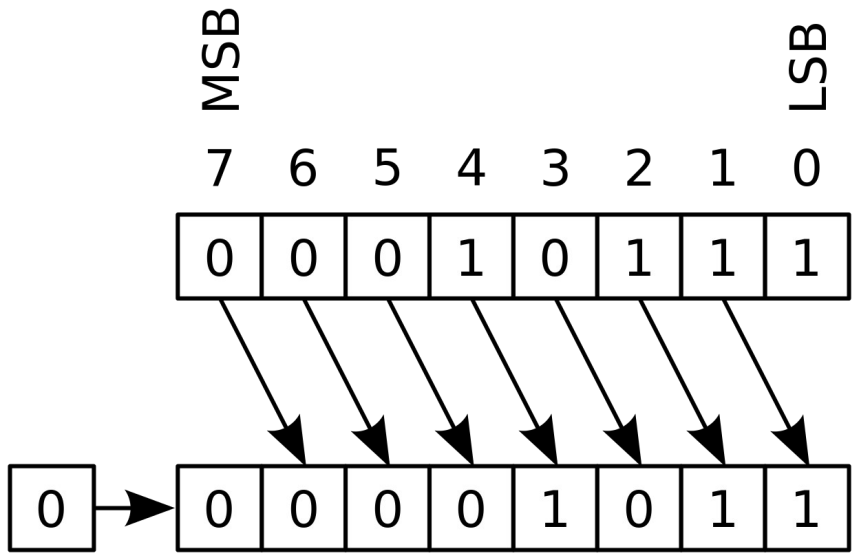
Circular Shift (Rotate)

Shift right 1: $B_3B_2B_1B_0 => B_0B_3B_2B_1$

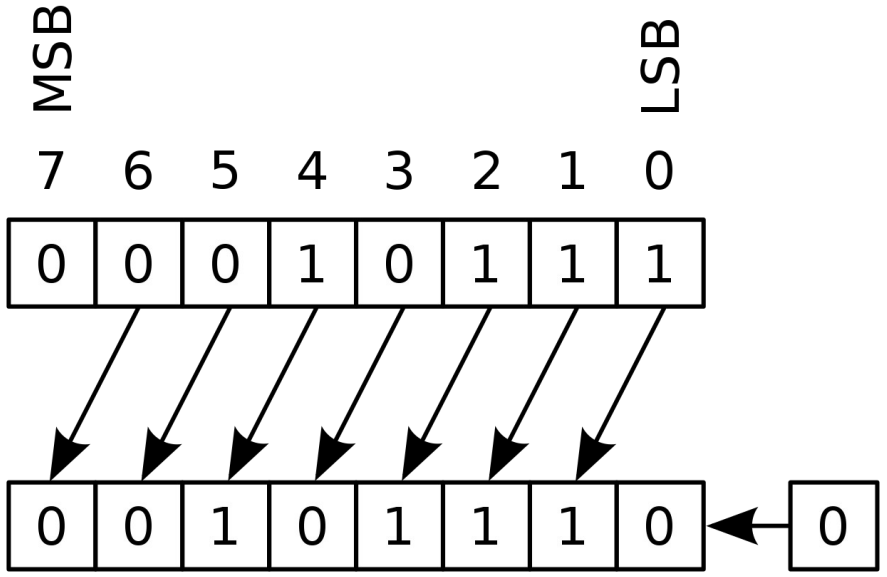
Shift left 1: $B_3B_2B_1B_0 => B_2B_1B_0B_3$

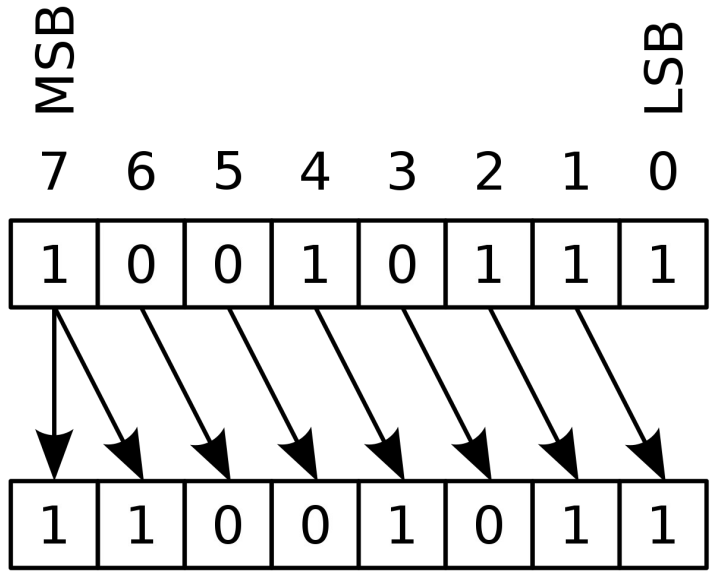
Logical Shift



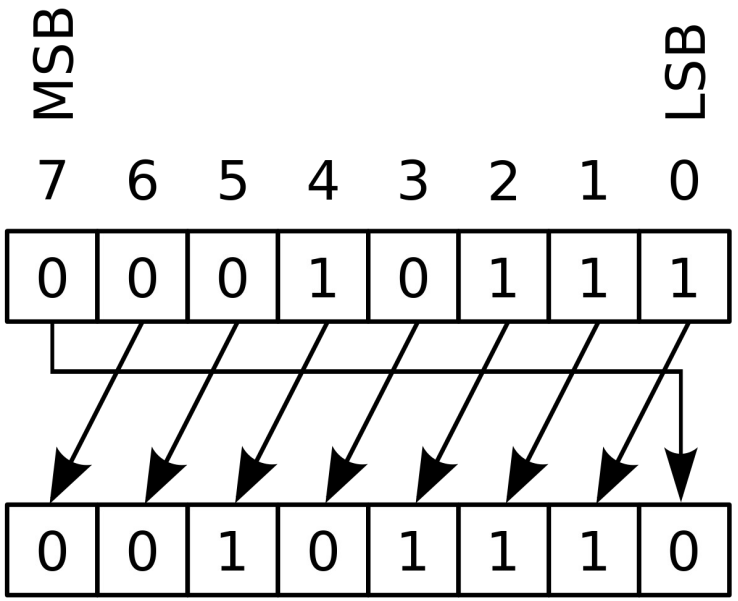


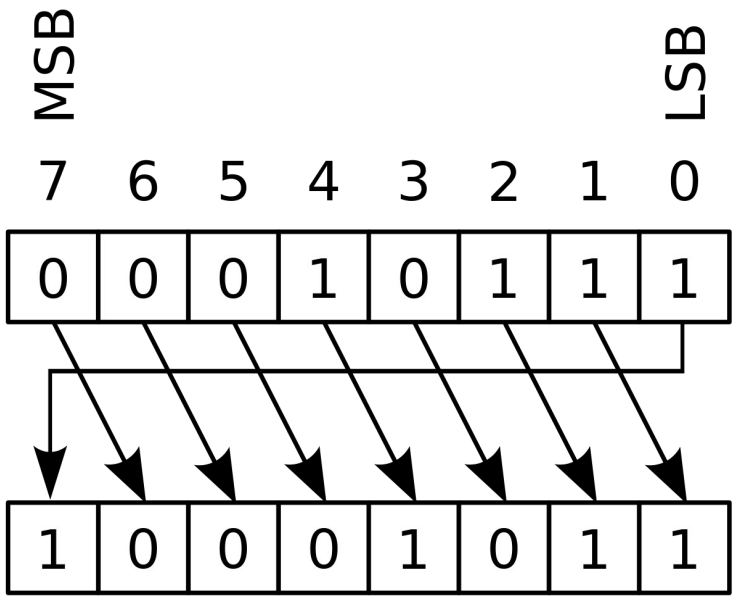
Arithmetic Shift



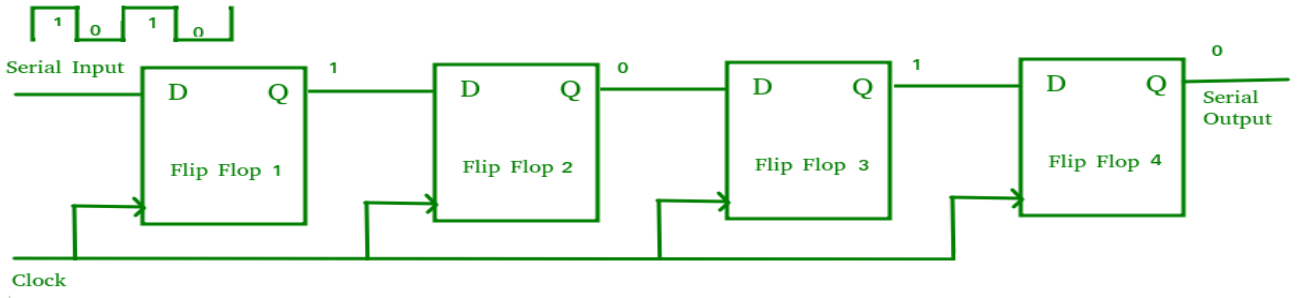


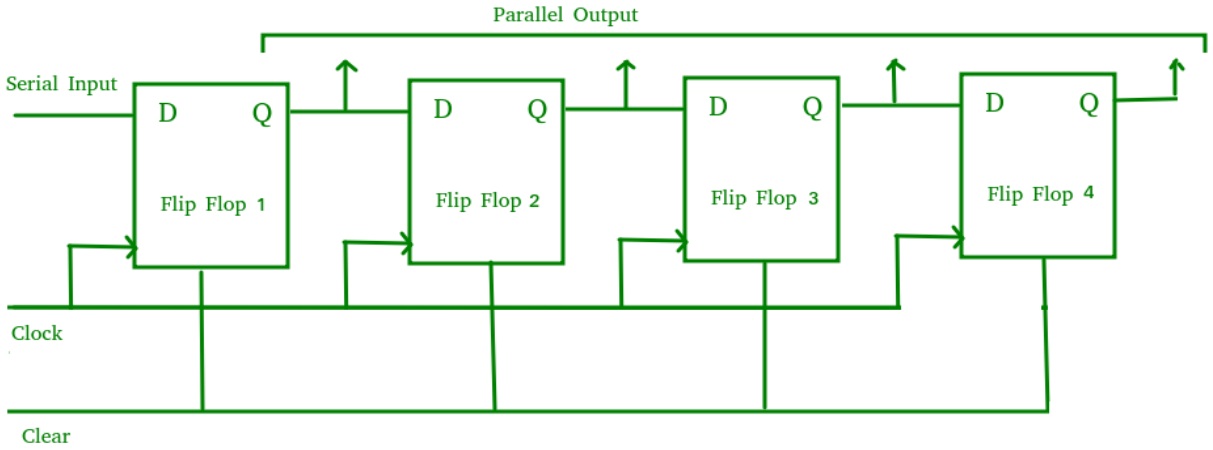
Circular Shift



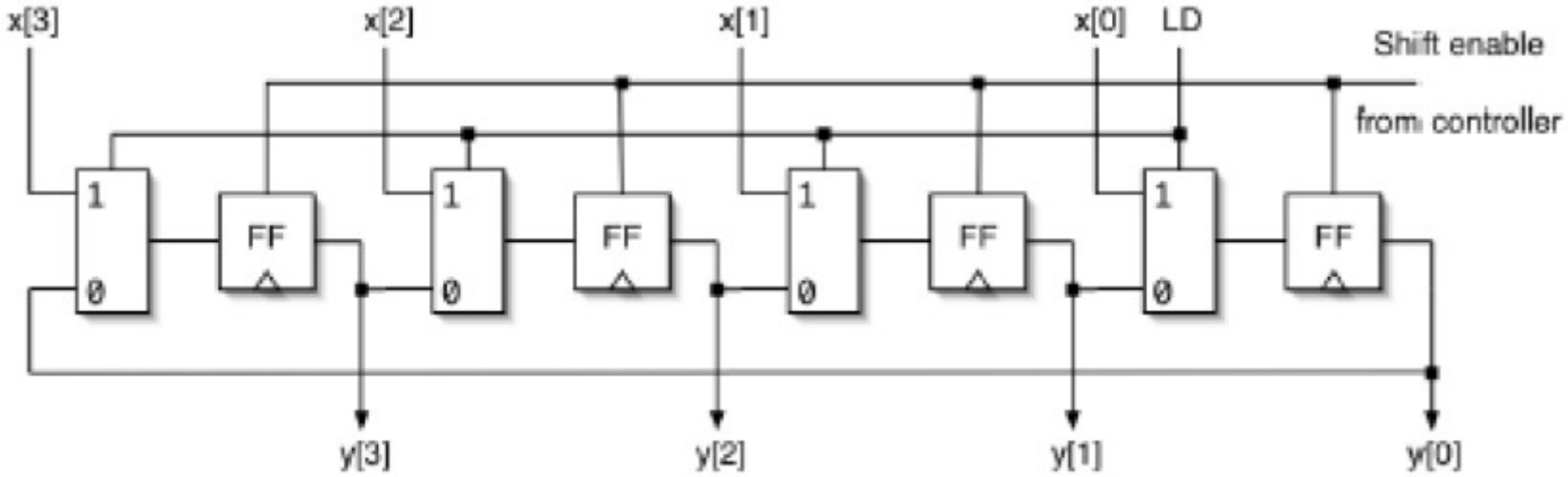


SI-SO, SI-PO

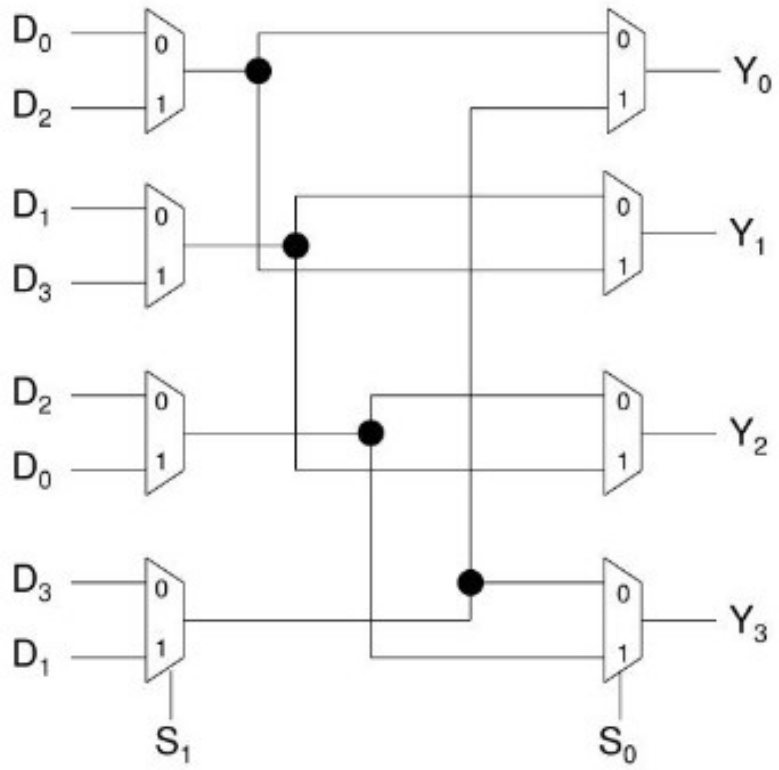

1
0
1
0
Serial Input
D
Q
Flip Flop 1
1
Flip Flop 2
0
Flip Flop 3
1
Flip Flop 4
0
Serial Output
Clock


Parallel Output
Serial Input
D
Q
Flip Flop 1
Flip Flop 2
Flip Flop 3
Flip Flop 4
Clock
Clear

PI-PO Shift Register Circuit



Barrel Shifter



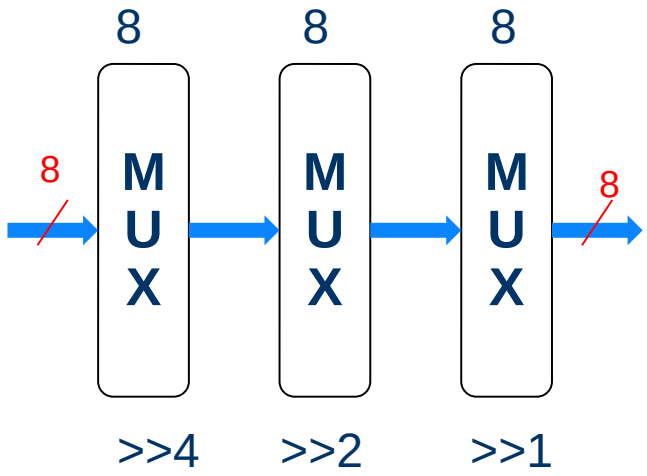
S1	P0-3	S0	Y0-3
0	$D_0D_1D_2D_3$	0	$D_0D_1D_2D_3$
0	$D_0D_1D_2D_3$	1	$D_3D_0D_1D_2$
1	$D_2D_3D_0D_1$	0	$D_2D_3D_0D_1$
1	$D_2D_3D_0D_1$	1	$D_1D_2D_3D_0$

James Morizio, ECE 261,Duke University

Barrel Shifter

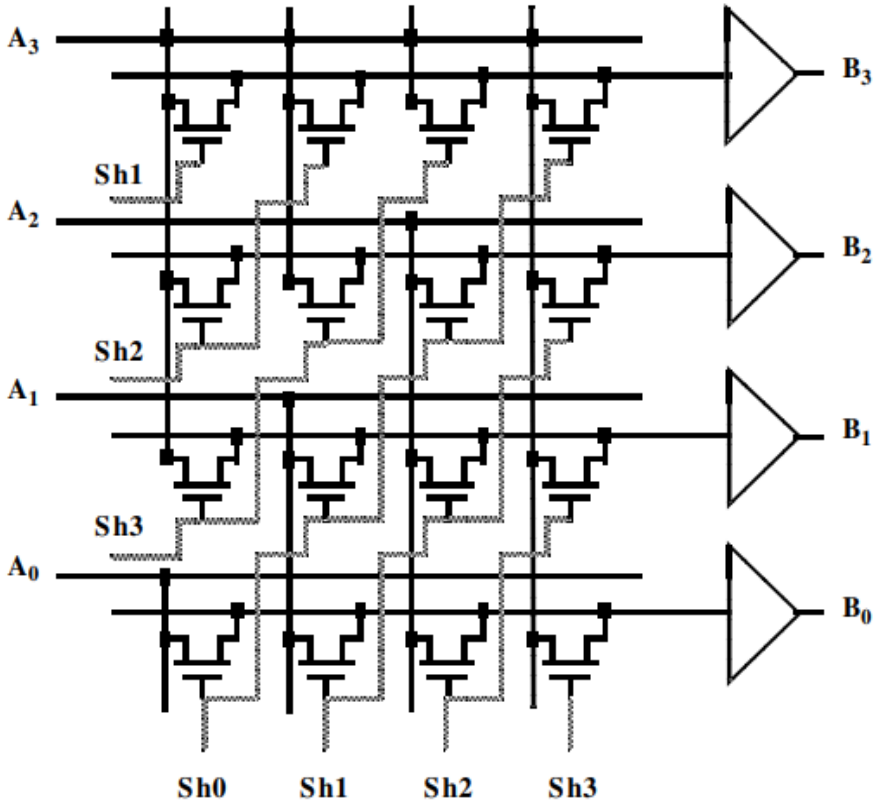
- Takes single cycle to shift and rotate n bits
- nlog2n : Total number of 2x1 multiplexers required to shift n-bit data

•For n=8 we get;



James Morizio, ECE 261,Duke University

Barrel Shifter



Shift by 0 to 3 bits

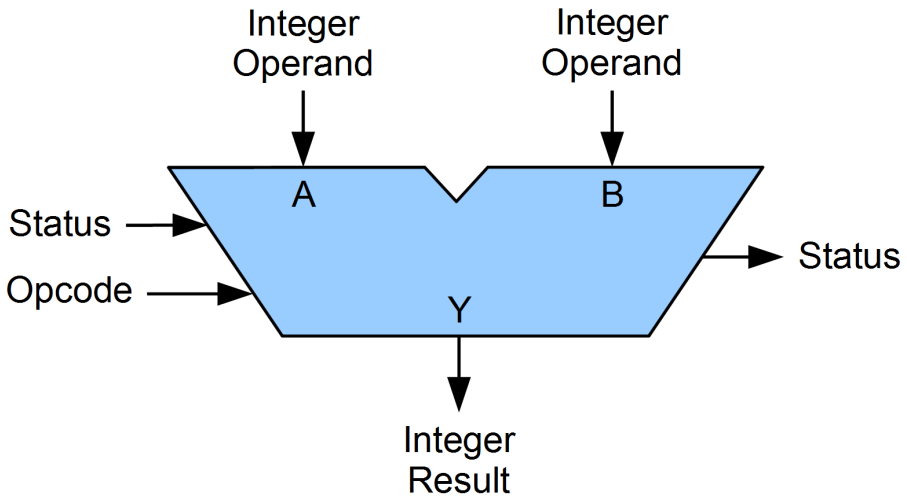
: Data Wire

: Control Wire

Arithmetic Logic Unit (ALU)

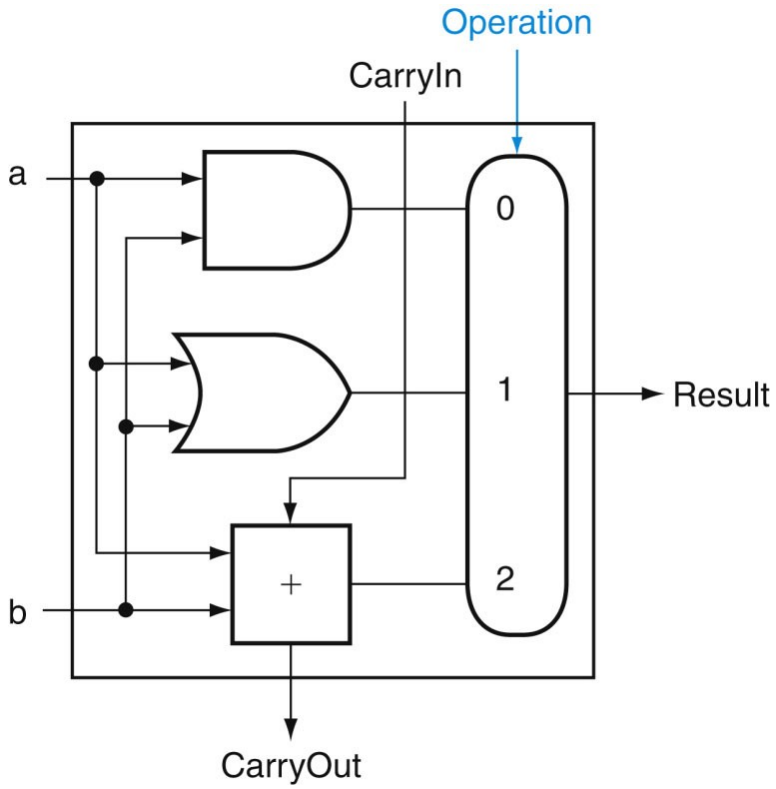
Arithmetic Logic Unit (ALU)

- Must be able to support the arithmetic/logic operations in the ISA:
  - add, addi, sub, mult, multu, div…
  - Xor, xori, and, andi, or, ori…
  - Beq, bne, slt, slti, sltu…



1-bit ALU

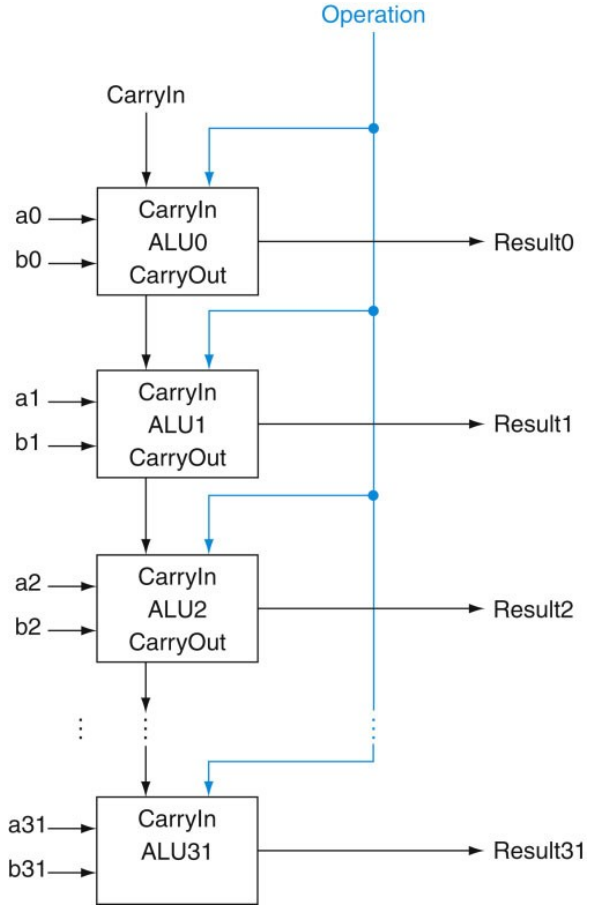
- Multiplexer selects the operation between And, Or, Add.



Can be extended to 32 bits

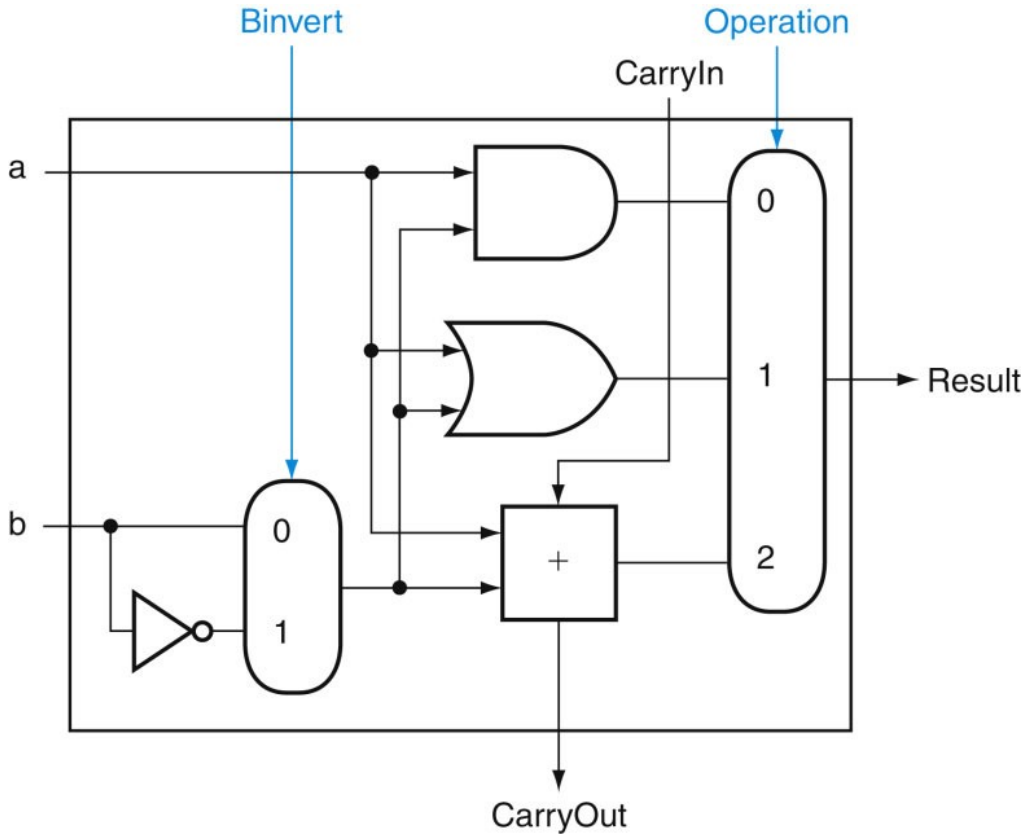
Ripple Carry Adder

- 32 1-bit ALU blocks are cascaded to construct 32-bit ALU.
- Carry-out of a block going into the carry-in of the next block.

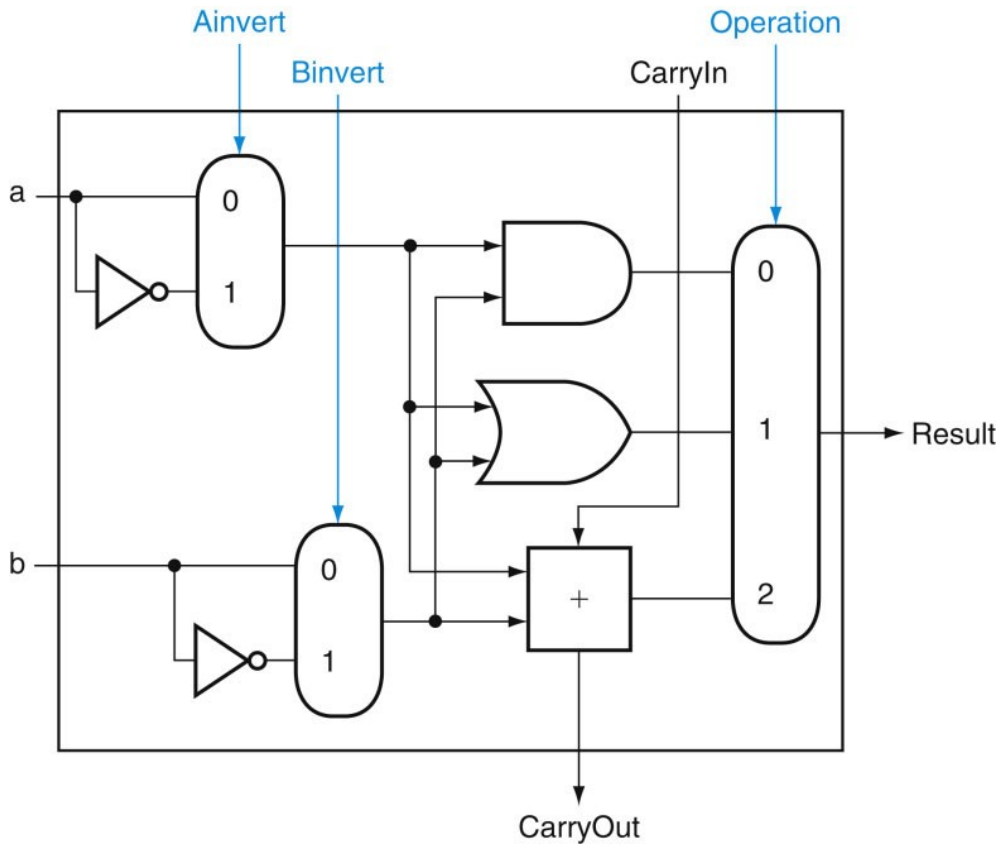


Subtraction in ALU

- Must invert bits of B and add 1.
  - Include an inverter
  - CarryIn is selected as 1

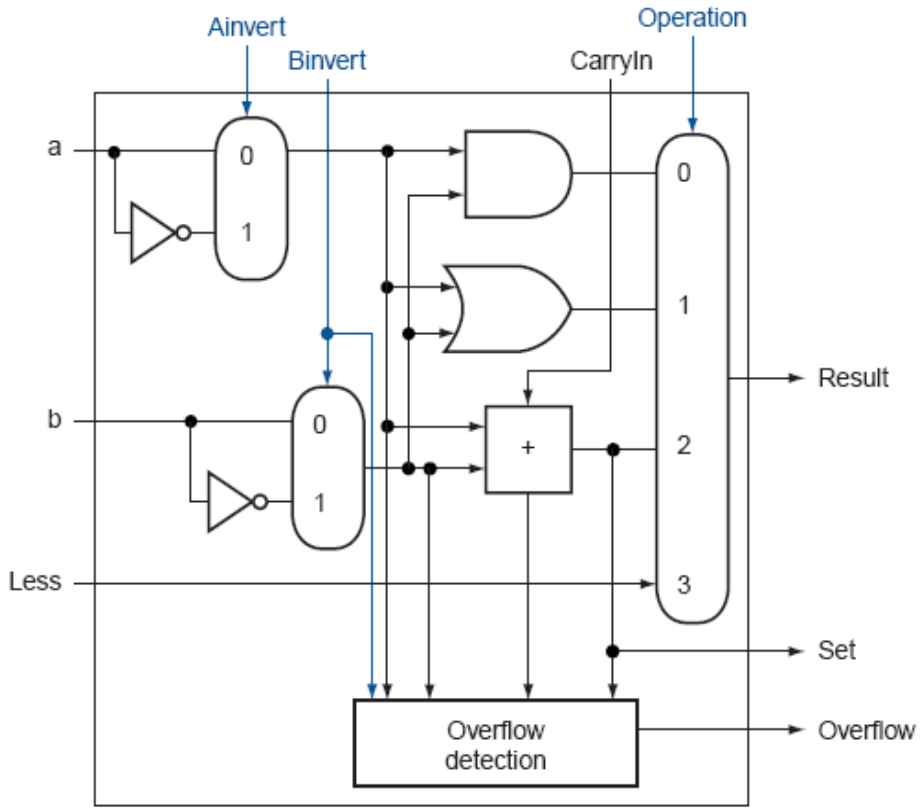


NOR and NAND


Ainvert
Binvert
CarryIn
Operation
a
b
0
1
0
1
0
1
2
+
Result
CarryOut

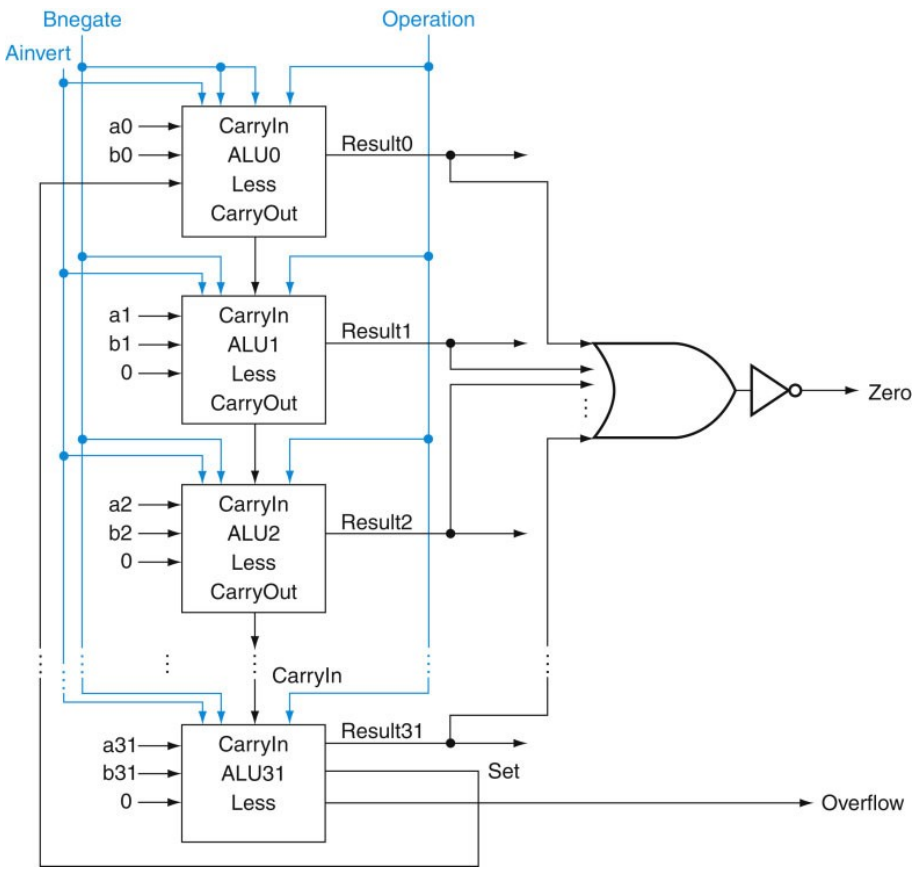
Set Less Than

- Perform a – b and then check the sign.
- The 31st block has a unit to detect overflow and sign bit (set signal).
- If overflow is 1, then carryOut signal is wired to the less signal in the 1st block. Else sign (set) signal is wired.



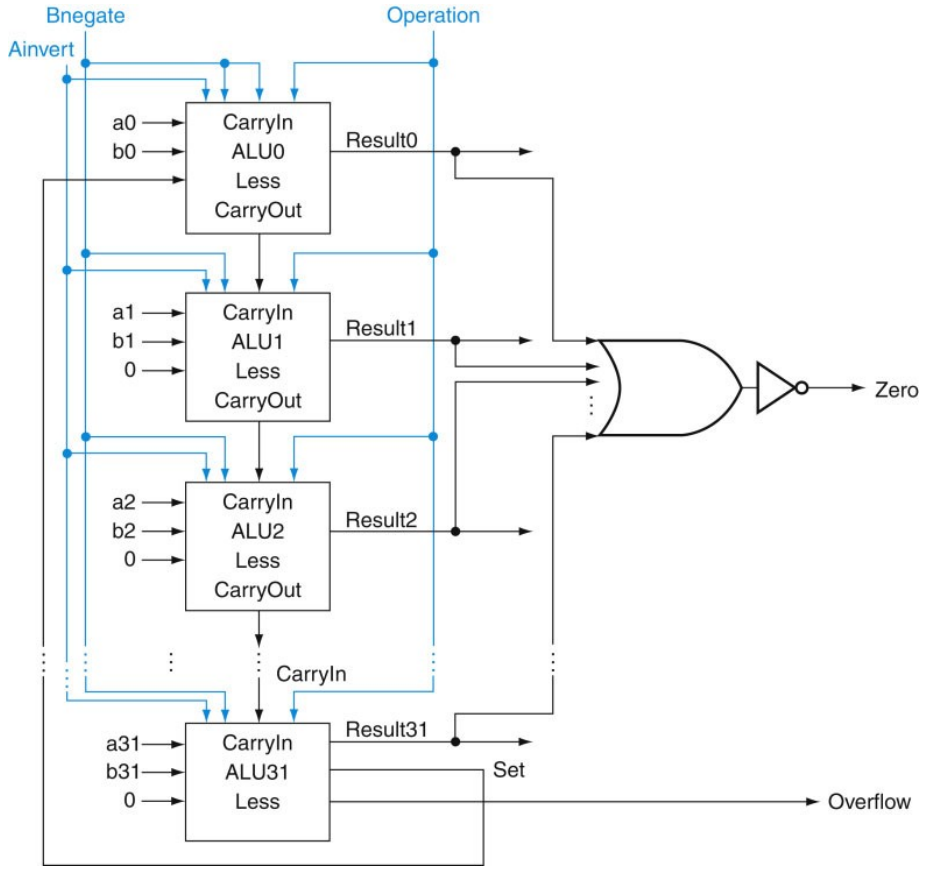
Branch if Equal

- Perform a – b and check that the result is all zero's.
- Perform nor operation for 32 bits.



Control Lines

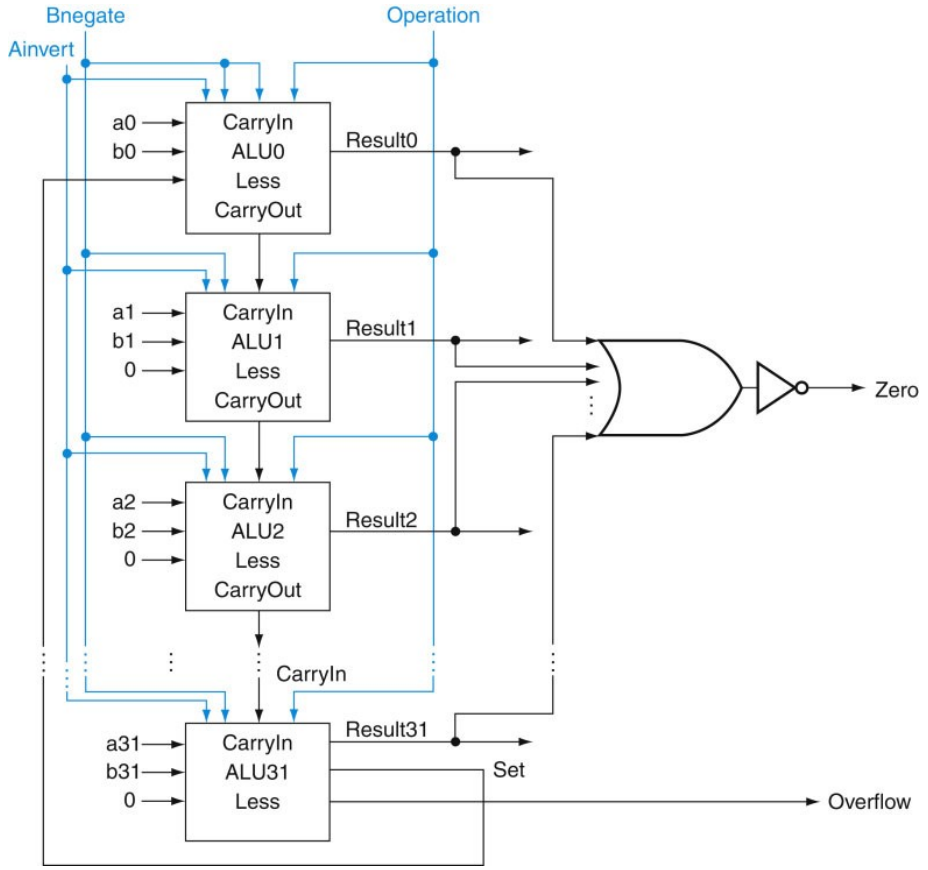
- What are the values of the control lines and what operations do they correspond to?



Control Lines

- Ainvert => nand, nor
- Binvert => sub, slt, beq
- Operation => and, or, add

	Ai	Bn	Op
AND	0	0	00
OR	0	0	01
Add	0	0	10
Sub	0	1	10
SLT	0	1	11
NOR	1	1	00
NAND	1	1	01



References

- B. Parhami, Computer arithmetic: Algorithms and hardware designs. New York: Oxford University Press, 2010.