Unit-II

ARITHMETIC FOR COMPUTER

Session Objectives

- To explain the fixed point addition and subtraction operation handled by the processor.
- To explain the fixed point Multiplication operation handled by the processor
- To explain the fixed point Division operation handled by the processor
- To explain the floating point addition/subtraction operation handled by the processor
- To explain the floating point Multiplication operation handled by the processor
- To explain the floating point Division operation handled by the processor

ARITHMETIC AND LOGIC UNIT

- The ALU performs arithmetic and logical operations on data.
- Data is stored in registers and the results of an operation are stored in registers.
- The control unit coordinates the operations of the ALU and the movement of the data in and out of the ALU.
- Most computers have registers called Accumulator(AC), or general purpose registers.
- The accumulator is the basic register containing one of the operand during operation in ALU.
- If the operation is add, the number stored in the AC is augend and the addend will be located and these operands (addend & augend) are added up and the result is stored back in the AC.
- The original augend will be lost in AC after addition.

Arithmetic operations

- The 4 basic arithmetic operations are
  - Addition
  - Subtraction
  - Multiplication
  - Division

  - The arithmetic operations are
    - Fixed point Arithmetic
    - Floating point Arthritic

INTEGER REPRESENTATION FIXED POINT

- Signed-Magnitude Representation
- Two’s Complement Representation

Signed-Magnitude Representation

- **+ve and –ve** numbers are differentiated by most significant bit in the word as a **sign bit**.
  - If the sign bit is 0, the number is positive
  - If the sign bit is 1, the number is negative
- +18 = 0 0 0 1 0 0 1 0

  ↓ ↓

  one sign bit seven bit magnitude

- Rang $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
  - Eg. A 7 bit register can store numbers from –63 to +63
- **Drawbacks**

1. Addition and subtraction requires both the sign bit and magnitude bits to be considered.

2. Two representation for zero (0)

- +0 → 0 0 0 0 0 0 0 0
- –0 → 1 0 0 0 0 0 0 0

Two's Complement Representation

- In two's complement system, forming the 2's complement of a number.
- Eg. Representation of –4
  - In Sign-magnitude 1 1 0 0
  - In 1's complement 1 0 1 1
  - In 2's complement 1 1 0 0
- **Advantages**

i. Only one arithmetic operation is required while subtracting using 2's complement notation.

ii. Used in arithmetic applications

Sign Extension

- **Sign Extensions**
- Sometimes it is desirable to take an **n-bit** integer and store it in **m bits**, where **m > n**.
- **In sign-magnitude**:
- simply move the sign bit to the new leftmost position and fill in with zeros.
- **In 2's complement :**
- Move the sign bit to the new leftmost position and fill it with copies of the sign bit.
  - For +ve no's fill it with zeros.
  - For –ve no's fill it with ones.
- Eg.

+18 = 0 0 0 1 0 0 1 0 ← 8 bit notation

= 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 ← 16 bit notation in sign magnitude

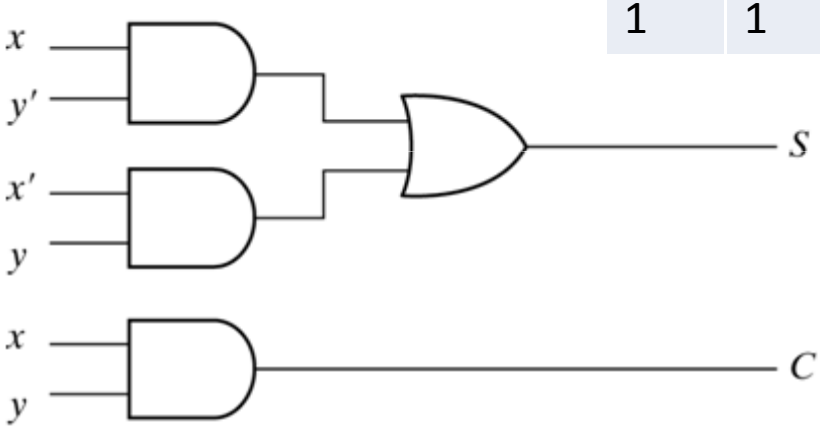
–18 = 1 1 1 0 1 1 1 0 ← 2's complement 8 bit notation

1111111111101110 ← 2's complement 16 bit notation

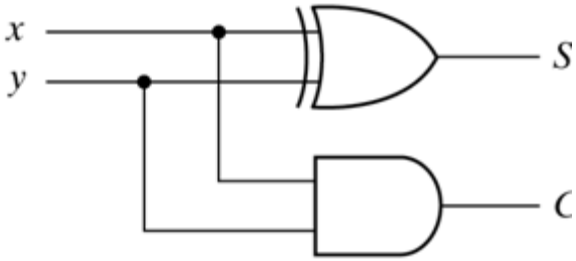
Adder

Half Adder

X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

Full Adder

- Truth table:

X	Y	Z	C	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

Note:

Z - carry in (to the current position)

C - carry out (to the next position)

$S = \Sigma m(1,2,4,7)$

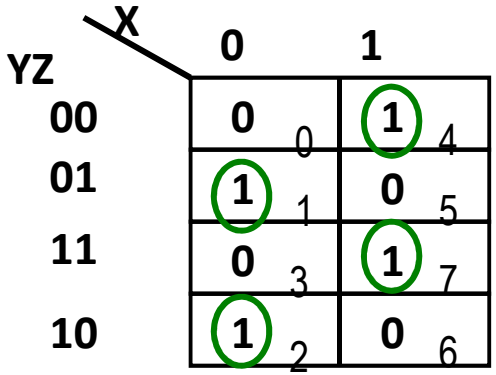
$C = \Sigma m(3,5,6,7)$

- Using K-map, simplified SOP form is:

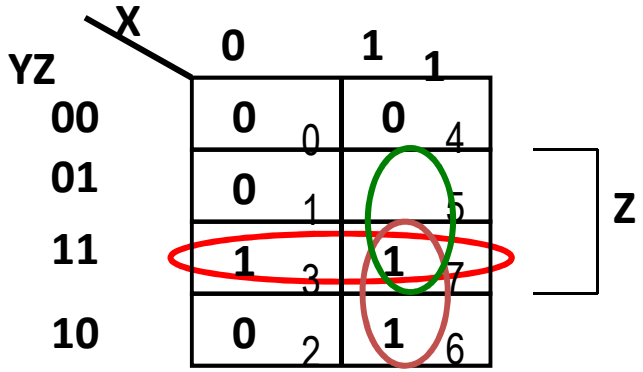
$C = XY + XZ + YZ$

$S = X'Y'Z + X'YZ' + XY'Z' + XYZ$

Sum



Carry



Full Adder Circuit

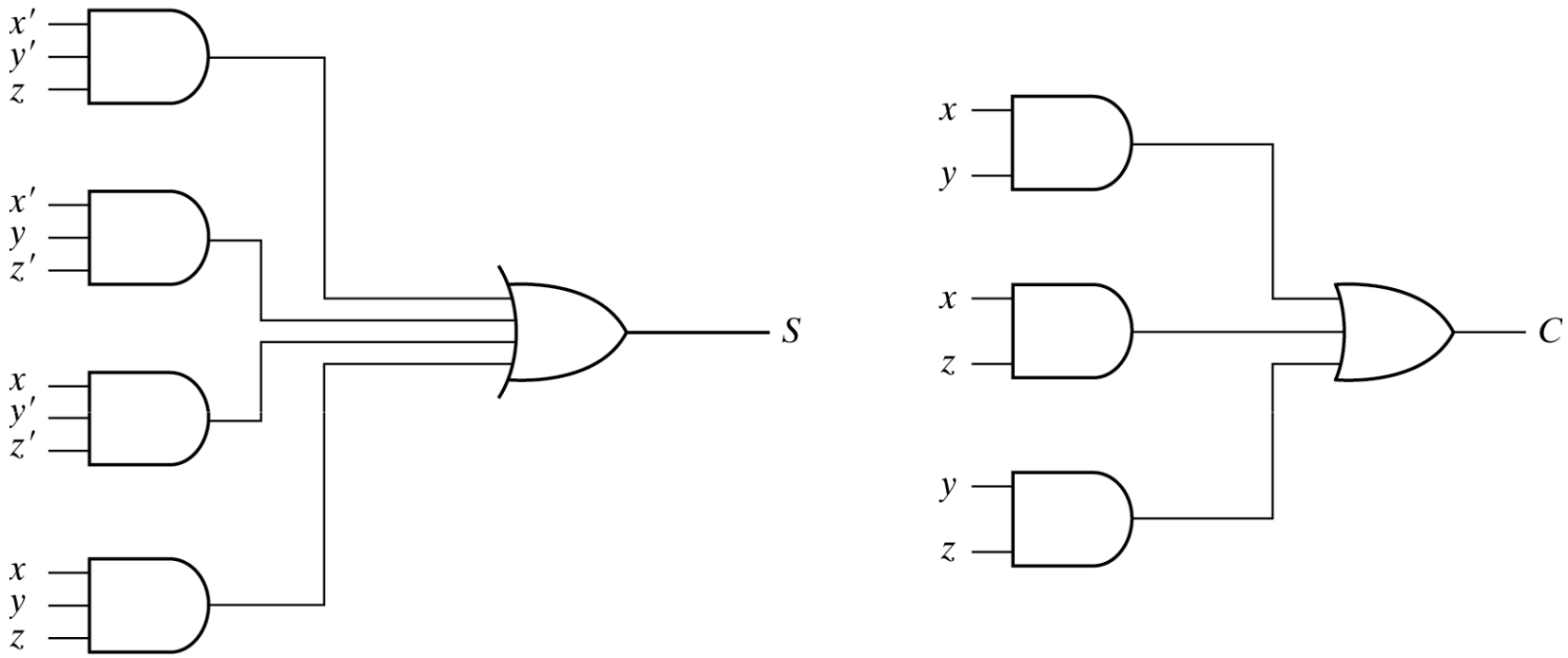


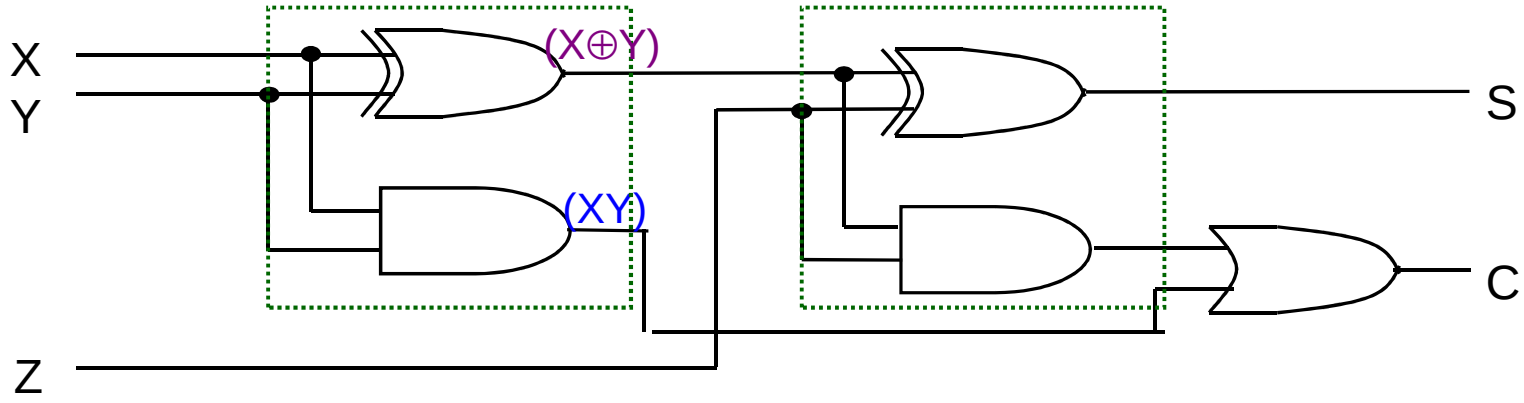
Fig. 4-7 Implementation of Full Adder in Sum of Products

Gate-level Design: Full Adder

- Circuit for above formulae:

  $C = XY + (X \oplus Y)Z$

  $S = X \oplus Y \oplus Z$



Full Adder made from two Half-Adders (+ OR gate).

4-bit Adder Circuit

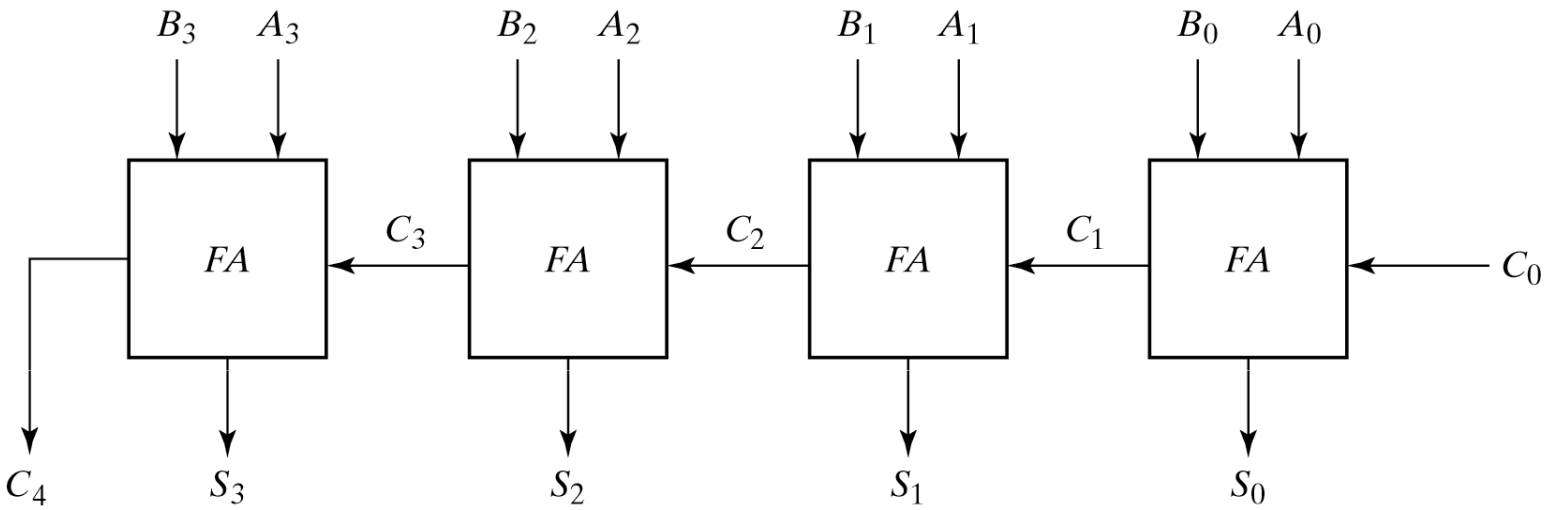
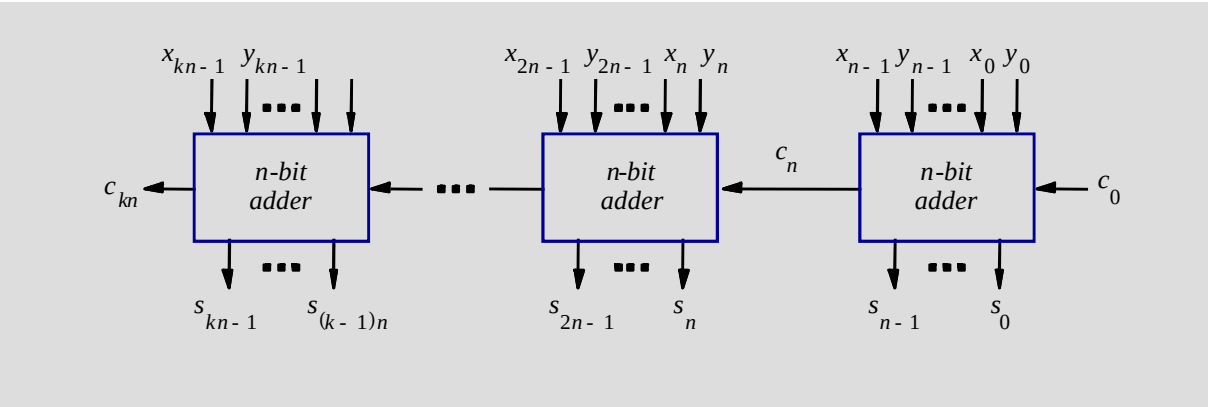


Fig. 4-9 4-Bit Adder

But this is slow...

K n-bit adder

K n-bit numbers can be added by cascading k n-bit adders.



Each *n*-bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

N bit Ripple carry adder

- Straight-forward design
- Simple circuit structure
- Easy to understand
- Most power efficient
- Slowest (too long critical path)

Bit -sliced ALU

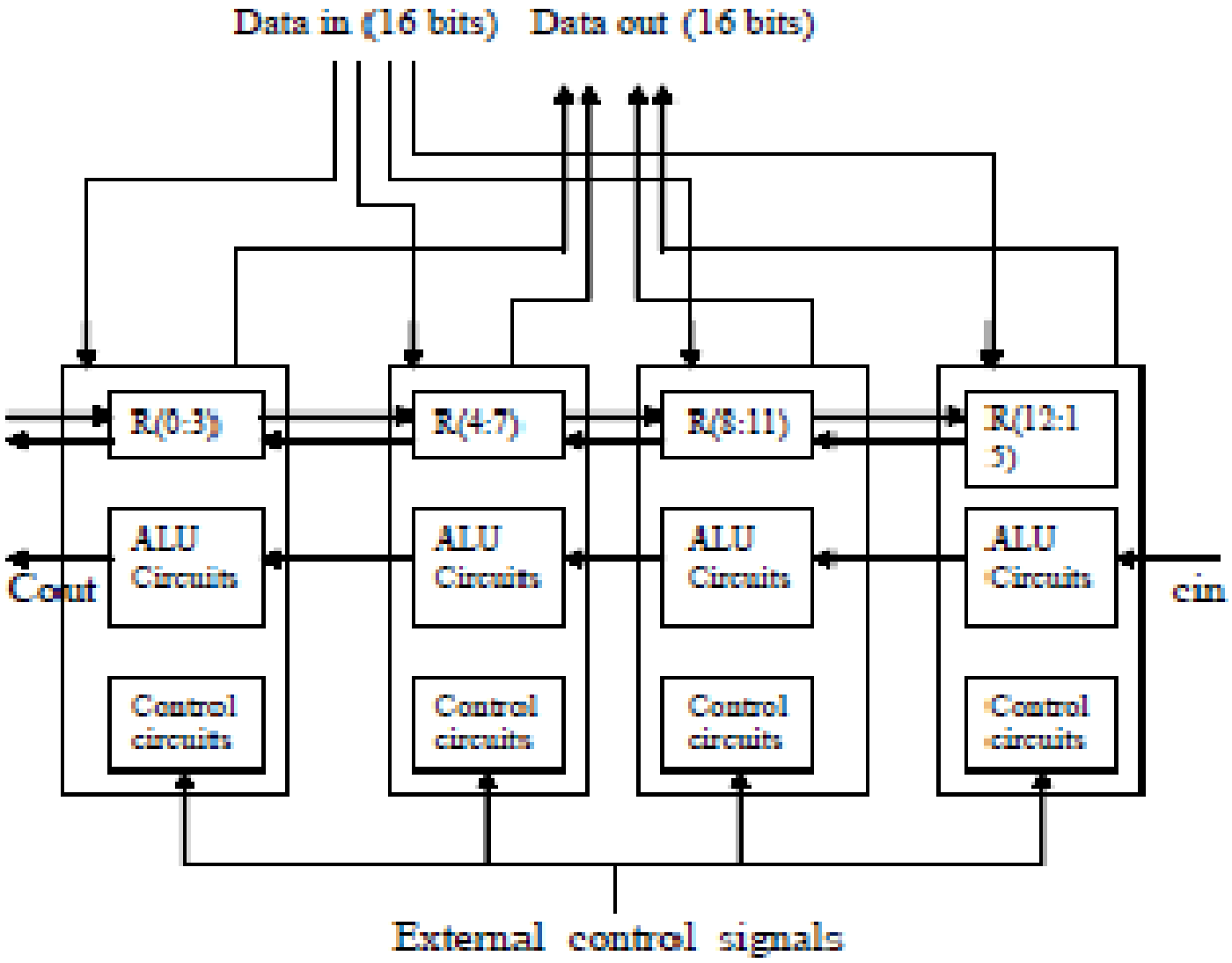


Fig: Bit Sliced ALU

Bit -sliced ALU

- Bit Sliced ALU
  - construct an entire fixed point ALU on a single IC chip if the word size **m** is kept small ex: 4 or 8 bits.
  - **m-bit ALU** can be designed to be expandable in that **k** copies of the ALU , can be connected to form a single ALU processing **km**-bit operands.
  - This array like circuit is called **bit sliced ALU** because each component chip processes an independent slice of m bits from each km bit operand.

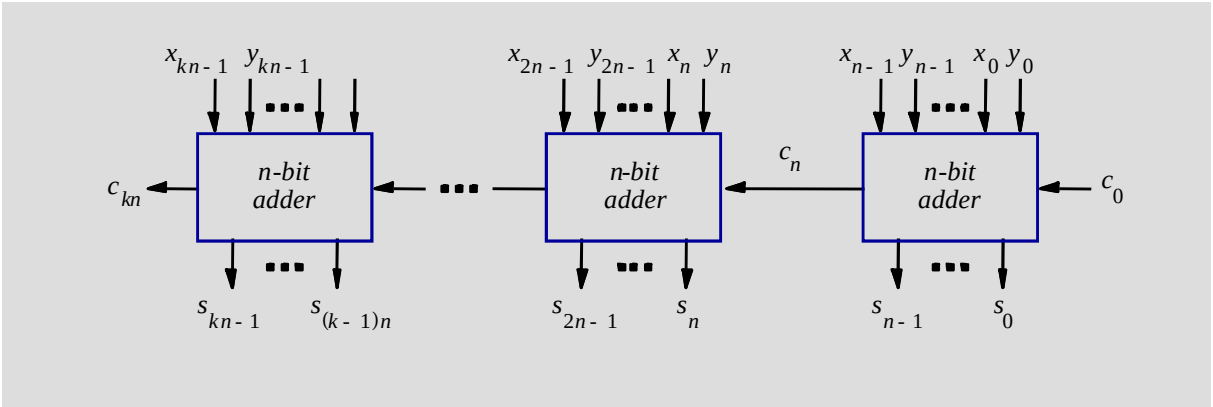
- **Advantage** :
- any desired word size or even several different word sizes can be handled by selecting the appropriate number of components (bit slices).

Bit -sliced ALU

- Data buses & registers of the individual slices are placed to increase their sizes from 4 to 16 bits.
- The control lines that select and sequence the operation to be performed are connected to every slice.
- so that all slices execute the same operation in step with one another.
- Each slice performs the same operation on a different 4-bit part (slice) of the input operands ,and produces only the corresponding part of the results.
- Certain operations require information to be exchanged between slices.
- Ex:
  - shift operation to be implemented then each slice must send a bit to and receive a bit from left or right neighbors
  - addition the carry bits may have to be transmitted between the neighboring slices.

Delays in the ripple carry adder

- This is called a ripple carry adder, because the inputs A_0, B_0 and CI "ripple" leftwards until CO and S_3 are produced.
- Ripple carry adders are slow!
  - For an n-bit ripple carry adder
    - Carry takes 2n gate delays
    - Sum takes (2n-1) gate delays
  - Imagine a 64-bit adder. The longest path would have 128 gates!

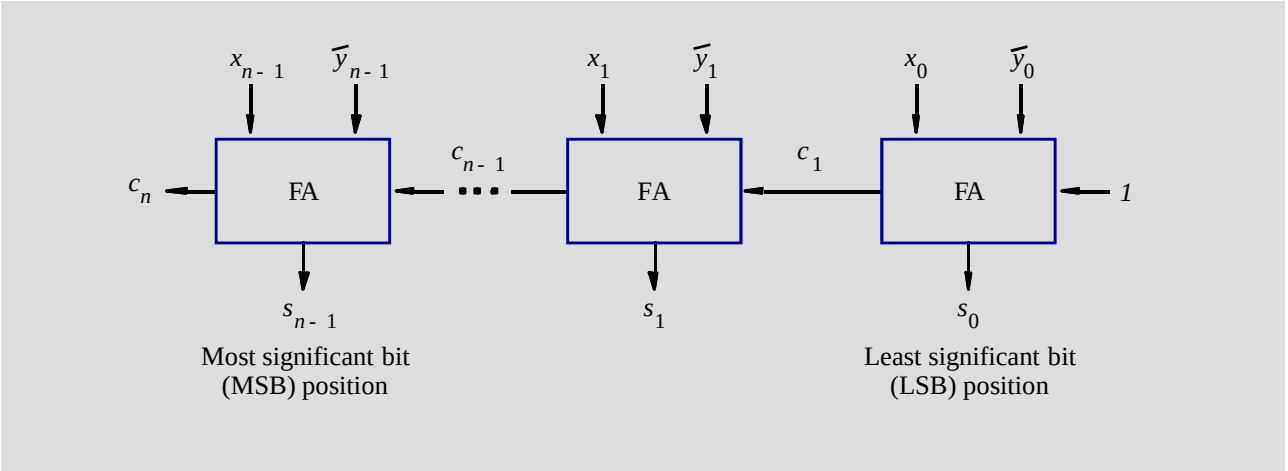


Binary Subtractor

- Subtraction is done by using complements
- A's 2's Complement = A'+1
- A-B= A + B'+1

n-bit subtractor

- Recall *X* – *Y* is equivalent to adding 2's complement of *Y* to *X*.
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + \overline{Y} + 1$
- 2's complement of positive and negative numbers is computed similarly.



4-bit adder subtractor

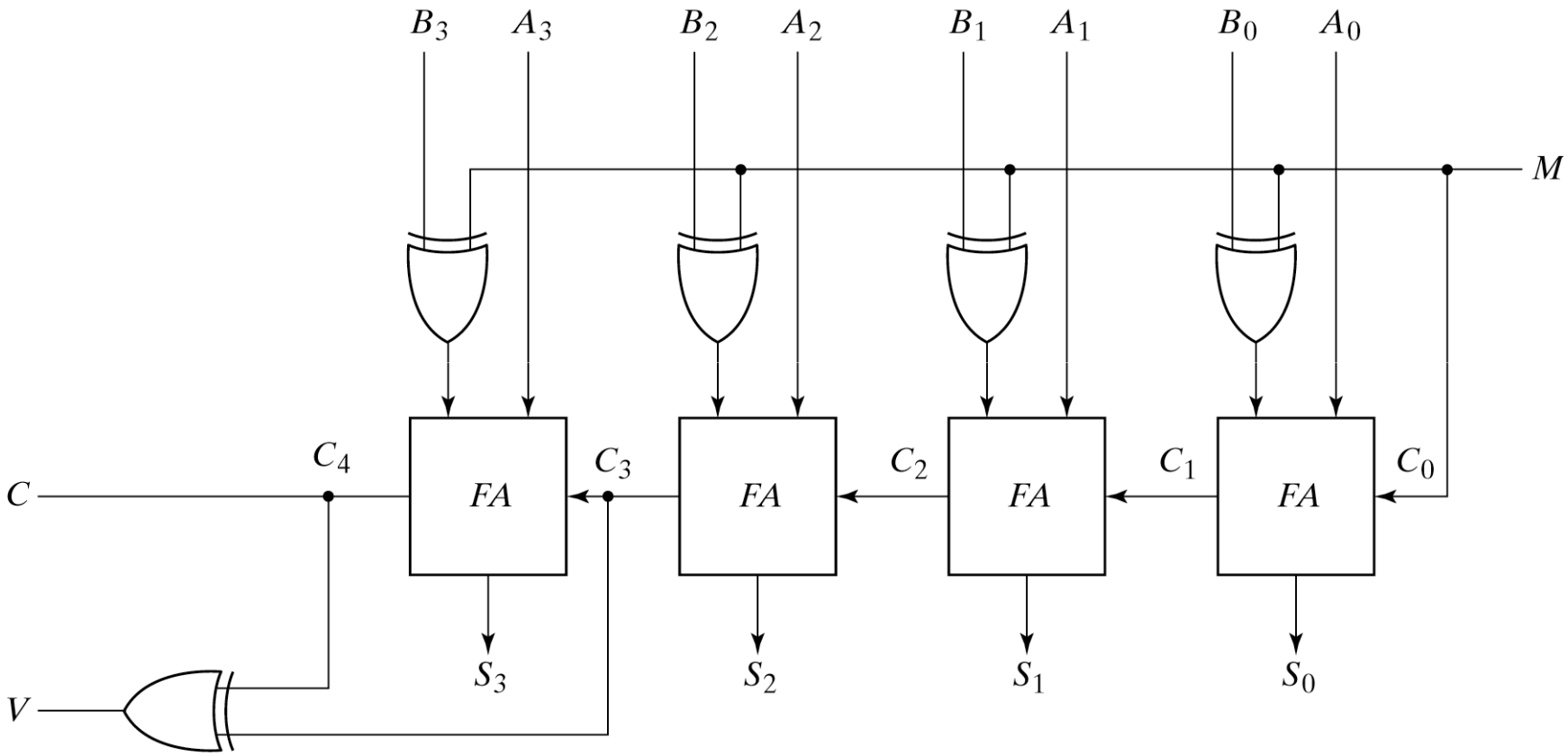


Fig. 4-13 4-Bit Adder Subtractor

Carry Lookahead adder

A faster way to compute carry outs

- Instead of waiting for the carry out from all the previous stages, we could compute it directly with a two-level circuit, thus minimizing the delay.
- First we define two functions.
  - The "generate" function g_i produces 1 when there *must* be a carry out from position i (i.e., when A_i and B_i are both 1).

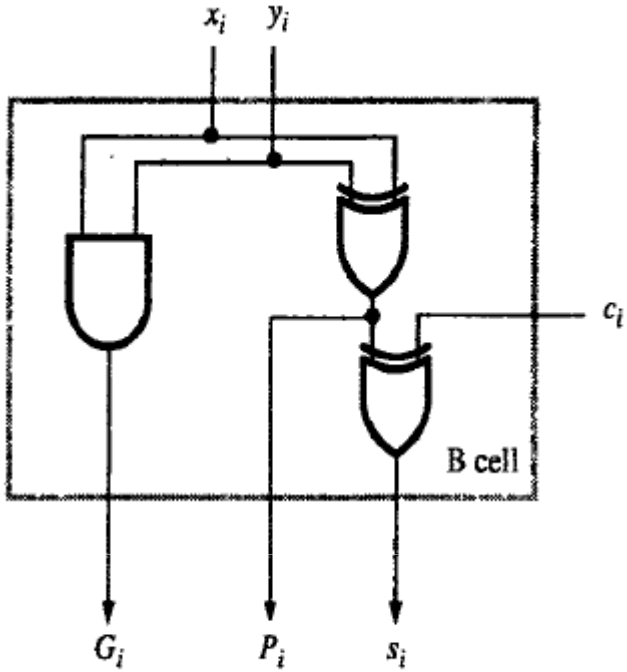
$$g_i = A_i B_i$$

  - The "propagate" function p_i is true when, if there is an incoming carry, it is propagated (i.e, when A_i=1 or B_i=1, but not both).

$$p_i = A_i \oplus B_i$$

- Then we can rewrite the carry out function:

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



A_i	B_i	C_i	C_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A Note On Propagation

- We could have defined propagation as A + B instead of A ⊕ B
  - As defined, it captures the case when we propagate but don't generate
    - I.e., propagation and generation are mutually exclusive
  - There is no reason that they need to be mutually exclusive
  - However, if we use ⊕ to define propagation, then we can share the XOR gate between the production of the sum bit and the production of the propagation bit

Algebraic carry out

- Let's look at the carry out equations for specific bits, using the general equation from the previous page $c_{i+1} = g_i + p_i c_i$:

$c_1 = g_0 + p_0 c_0$

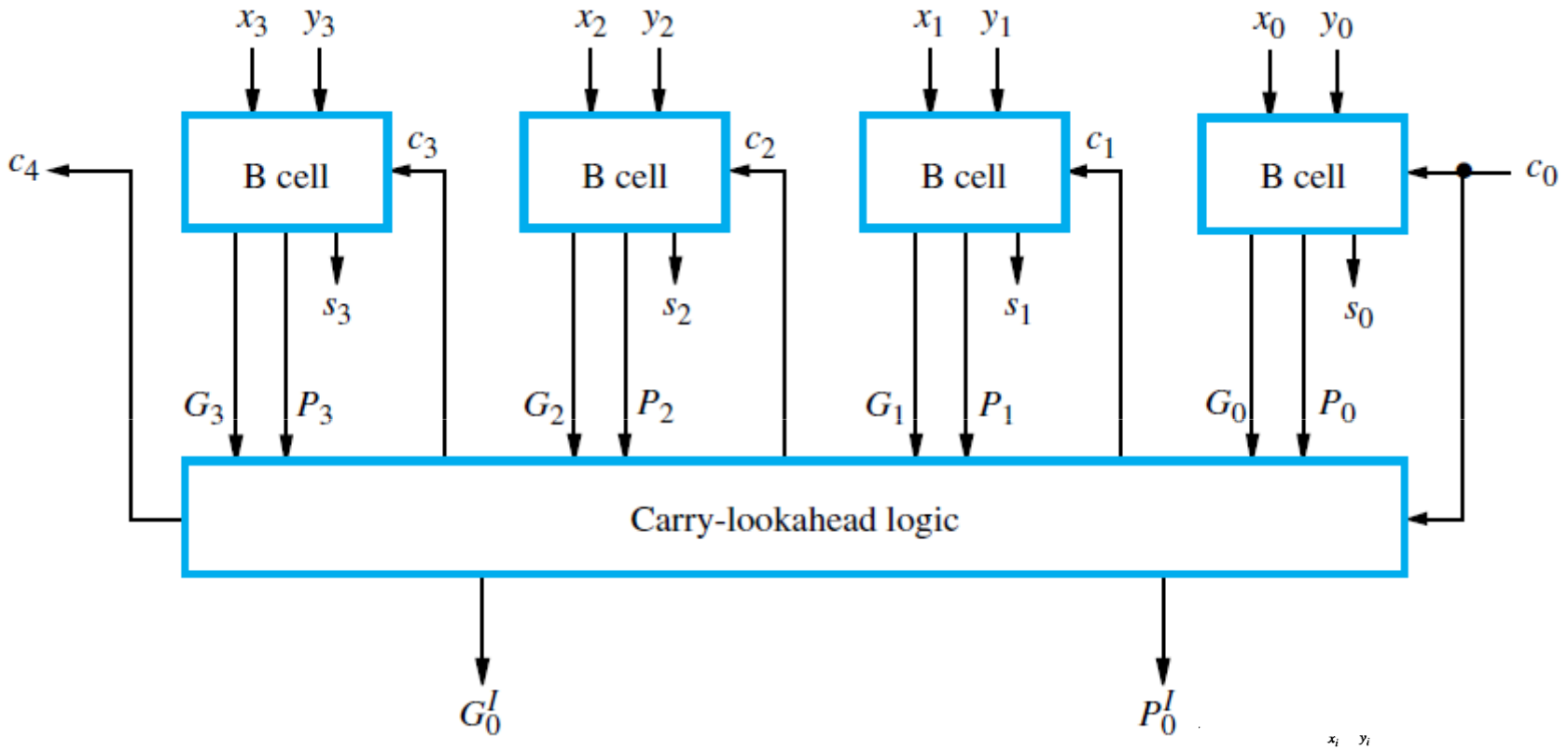
$c_2 = g_1 + p_1 c_1$
$= g_1 + p_1(g_0 + p_0 c_0)$
$= g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 c_2$
$= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0)$
$= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

$c_4 = g_3 + p_3 c_3$
$= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0)$
$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

- These expressions are all sums of products, so we can use them to make a circuit with only a two-level delay.

4 bit Carry lookahead adder



Gate Delays 4 bit CLA

G_i, P_i = 1 gate delay

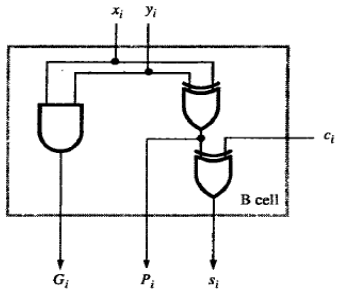
C_i = 3rd gate delay

S_i = 4th gate delay

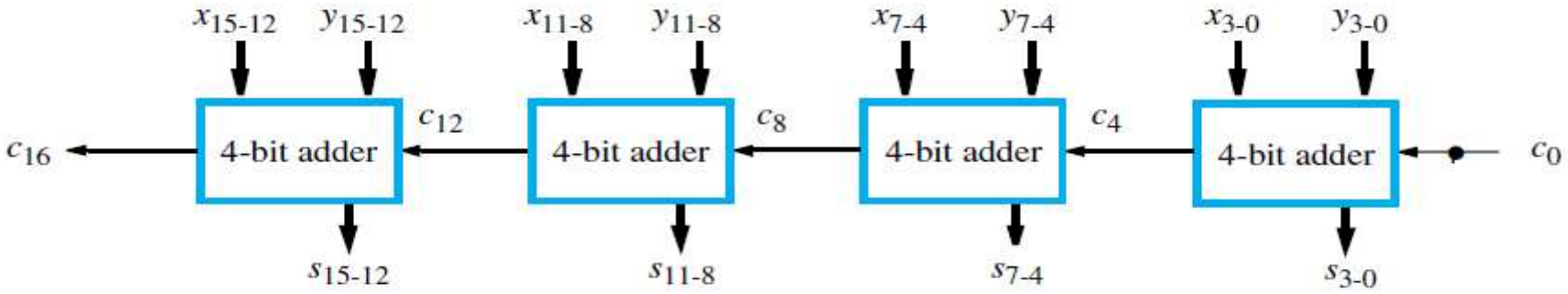
Gate Delays 4 bit ripple carry

C_i = 8 gate delay

S_i = 7 gate delay



16 bit ripple carry adder using 4 bit Carry lookahead adder



Gate Delays for 16 bit

1. Ripple Carry Adder

Sum 31

Carry 32

2. CLA

G,P = 1 gate delay

C_4 = 3rd gate delay,

C_8 = 5th gate delay

C_{12} = 7th gate delay

C_{16} = 9th gate delay

S_{15} = 10th gate delay

Gate Delays for 32 bit

1. Ripple Carry Adder

Sum 63

Carry 64

2. CLA

G,P = 1 gate delay

C_4 = 3rd gate delay,

C_8 = 5th gate delay

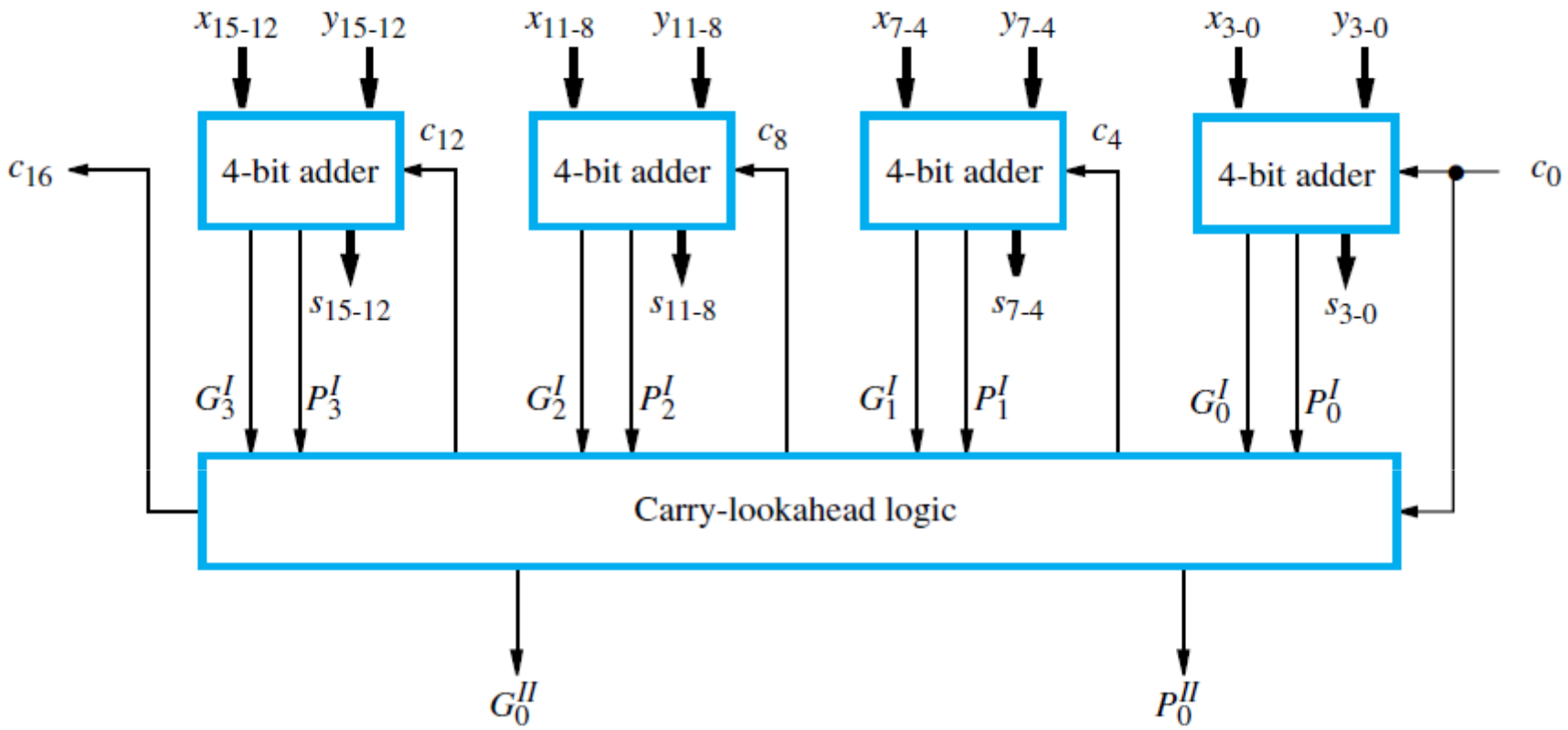
C_{12} = 7th gate delay ...

C_{32} = 17th gate delay

(1 for g,p +2*8 for each 4 bit adder)

S_{32} = 18th gate delay

High level Generator and Propagator function



4 CLA forms 16bit adder

Gate delays

G,P =1

G^I,P^I = 3rd gate delay

C4,C8,C12,C16 =5th gate delay

C12 to C15 = 7th gate delay

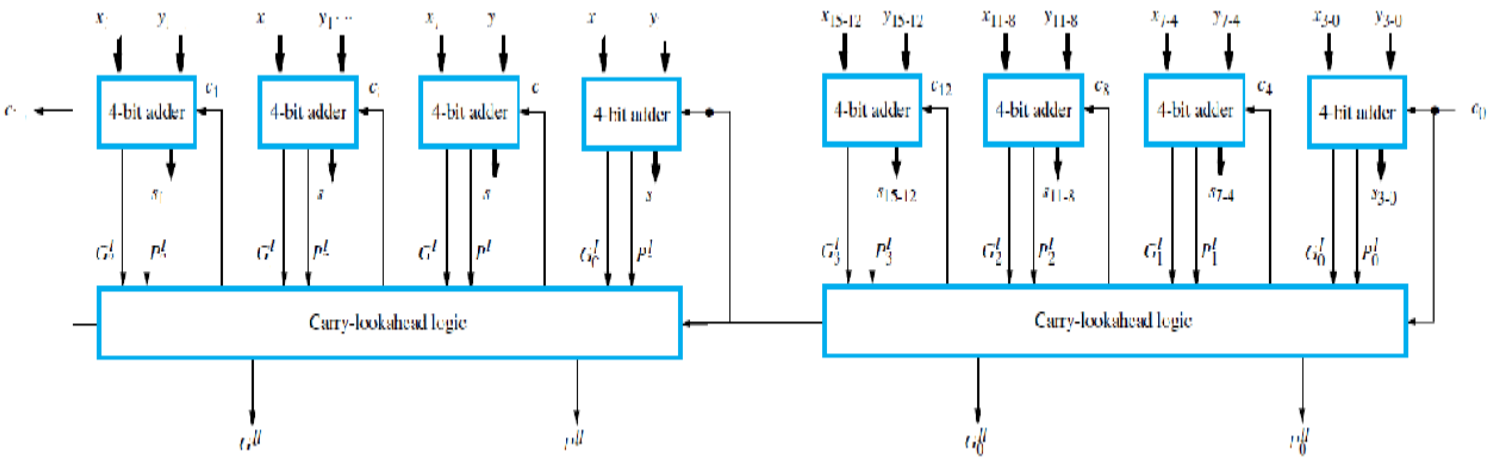
S15 = 8th gate delay

$$P_0^I = P_3P_2P_1P_0$$

$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

$$c_{16} = G_3^I + P_3^IG_2^I + P_3^IP_2^IG_1^I + P_3^IP_2^IP_1^IG_0^I + P_3^IP_2^IP_1^IP_0^Ic_0$$

32 bit adder cascading two 16 bit Carry lookahead adder



Gate Delay 32 bit adder

G,P =1

G^I,P^I = 3rd gate delay

C4,C8,C12,C16 =5th gate delay

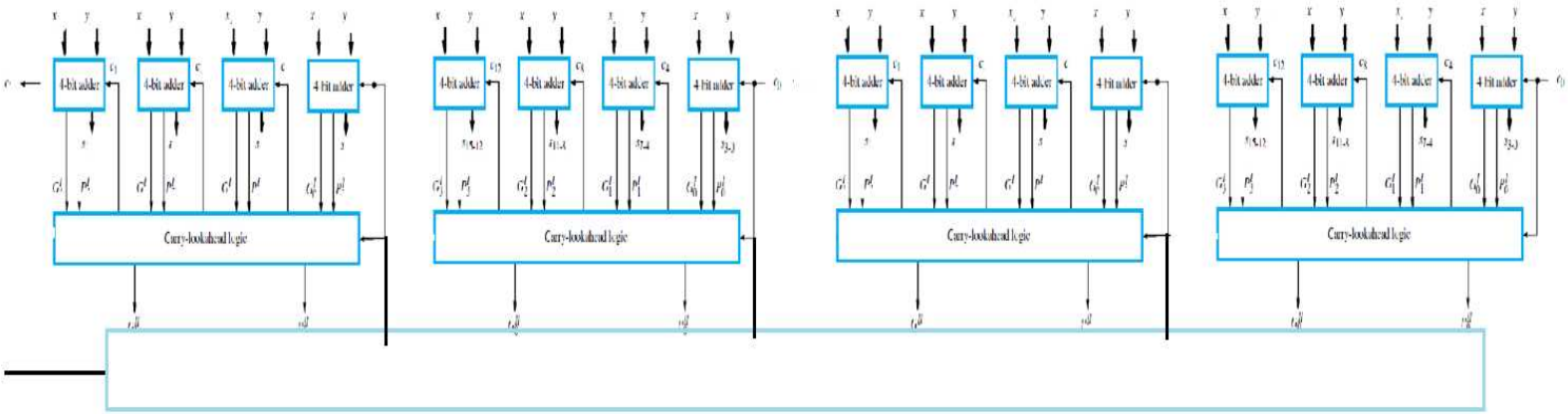
C12 to C15 = 7th gate delay

G^I,P^I = 7 th gate delay

C20,24,28,32 = 9th gate delay

S31 = 10th gate delay

64 bit Carry lookahead adder



Gate Delay

G,P =1

G^{I},P^{I} = 3rd gate delay

G^{II},P^{II} = 5 th gate delay

C16,C32,C48,C64 = 7^{th} gate delay

C52,C56,C60 =9^{th} gate delay

C61,C62,C63 = 11^{th} gate delay

S63 = 12^{th} gate delay

Overflow conditions

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Overflow conditions

Sequence of MIPS instructions can discover overflow - signed addition

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

Overflow conditions

Sequence of MIPS instructions can discover overflow - unsigned addition

```
addu $t0, $t1, $t2      # $t0 = sum
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2      # (2^32 - $t1 - 1) < $t2
                        # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow  # if(2^32-1<$t1+$t2) goto overflow
```