

# **COMP3222/9222 Digital Circuits & Systems**

## **3. Number Representation & Arithmetic Circuits**

# Objectives

- [Review] representation of numbers in computers
- Circuits for performing arithmetic operations
- Performance issues in large circuits
- VHDL for specifying arithmetic circuits

# Unsigned integers

- Numbers that are positive only are called *unsigned*
  - Numbers that can be negative are called *signed*
- 
- Unsigned binary numbers are represented using the positional number representation as in

$$B = b_{n-1}b_{n-2}\dots b_1b_0$$

which is an integer that has the value

$$\begin{aligned} V(B) &= b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{n-1} b_i \times 2^i \text{ with } b_i \in \{0, 1\} \end{aligned}$$

- 
- Note that the positional number representation can be used for any radix  $r$ , in which the unsigned number

$$K = k_{n-1}k_{n-2}\dots k_1k_0$$

has the value

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i \text{ with } k_i \in \{0..r-1\}$$

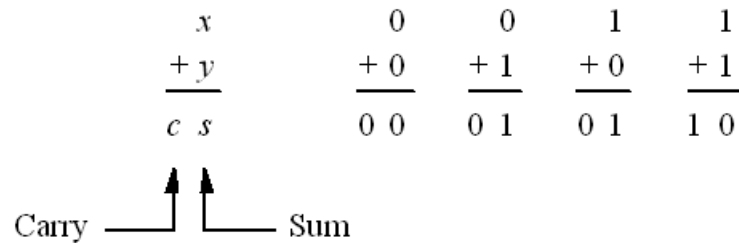
# Numbers in different systems

- Apart from radix 10 (decimal) and 2 (binary), radices 8 (octal) and 16 (hexadecimal, or simply “hex”) are useful
- Octal & hexadecimal are useful shorthand notations for binary numbers, which are predominantly used in computers
  - A binary number is converted into octal (hex) by taking groups of three (4) bits, starting from the least significant bit, and replacing them with the corresponding octal (hex) digit
  - Conversion from octal (hex) to binary requires that each digit be replaced by the corresponding three (4) bits denoting the same value
- Confirm decimal → binary conversion is understood

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

Note the need to introduce symbols ‘A’ through ‘F’ in hex in order to represent the digits 10 through 15.

# Binary addition – the half-adder circuit



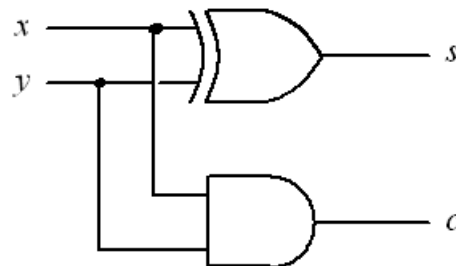
(a) The four possible cases

		Carry	Sum
$x$	$y$	$c$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

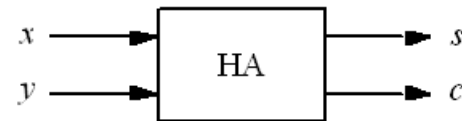
(b) Truth table

Note that the truth table

- a) Is symmetric w.r.t.  $x$  and  $y$ , and
- b) Converts the unary value  $xy$  into binary value  $cs$



(c) Circuit



(d) Graphical symbol

# Binary addition of more than 2 bits

- In general, this requires at position  $i$  the addition of three bits (two inputs and a carry-in from position  $i-1$ ) and results in a sum bit and a carry-out bit for the next position to the right

$X = x_4 x_3 x_2 x_1 x_0$	0 1 1 1 1	$(15)_{10}$
$+ Y = y_4 y_3 y_2 y_1 y_0$	0 1 0 1 0	$(10)_{10}$
<hr/>	<hr/>	
$C = c_4 c_3 c_2 c_1$	1 1 1 0	← Generated carries
<hr/>	<hr/>	
$S = s_4 s_3 s_2 s_1 s_0$	1 1 0 0 1	$(25)_{10}$

# The full-adder circuit

Consider the addition of two bits and a carry in at each bit position, resulting in a sum bit and a carry out

$x_i$	$y_i$	$c_i$	$c_{i+1}$	$s_i$
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

(a) Truth table

$x_i y_i$	00	01	11	10
$c_i$				
0		1		1
1	1		1	

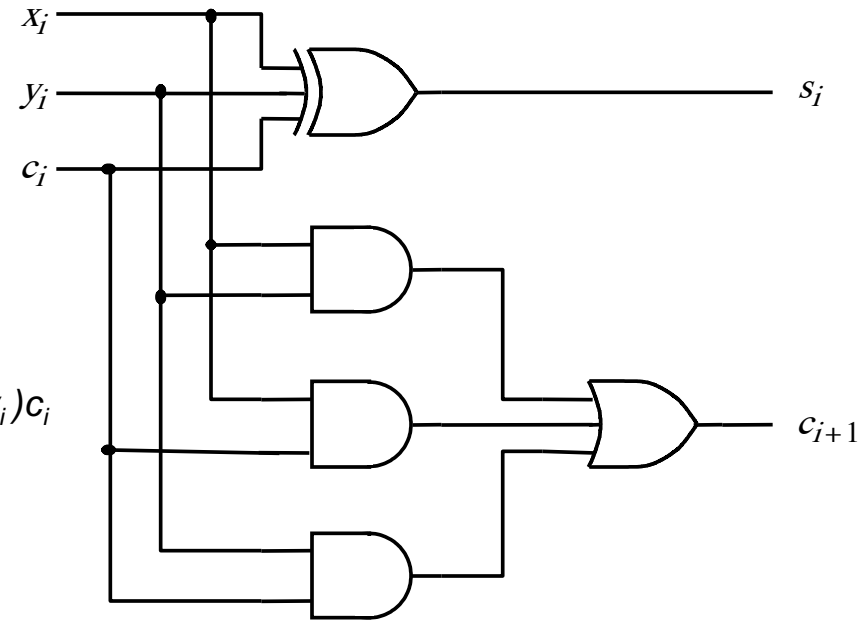
$$\begin{aligned}
 s_i &= (\bar{x}_i y_i + x_i \bar{y}_i) \bar{c}_i + (\bar{x}_i \bar{y}_i + x_i y_i) c_i \\
 &= (x_i \oplus y_i) \bar{c}_i + (\bar{x}_i \bar{y}_i + x_i y_i) c_i \\
 &= (x_i \oplus y_i) \oplus c_i
 \end{aligned}$$

a.k.a. the "odd" function

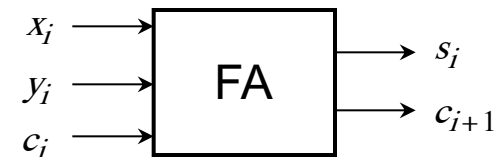
$x_i y_i$	00	01	11	10
$c_i$				
0			1	
1		1	1	1

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(b) Karnaugh maps

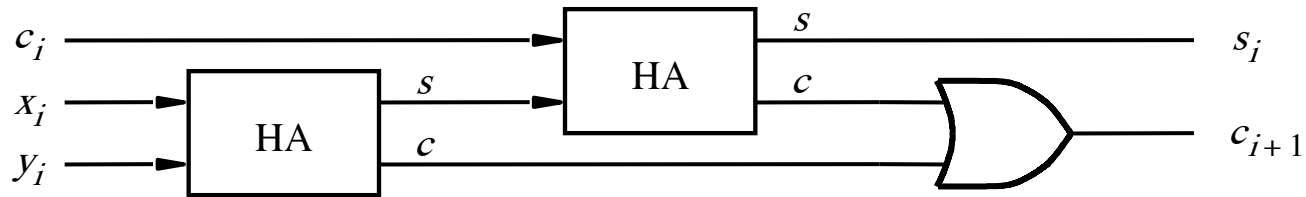


(c) Circuit

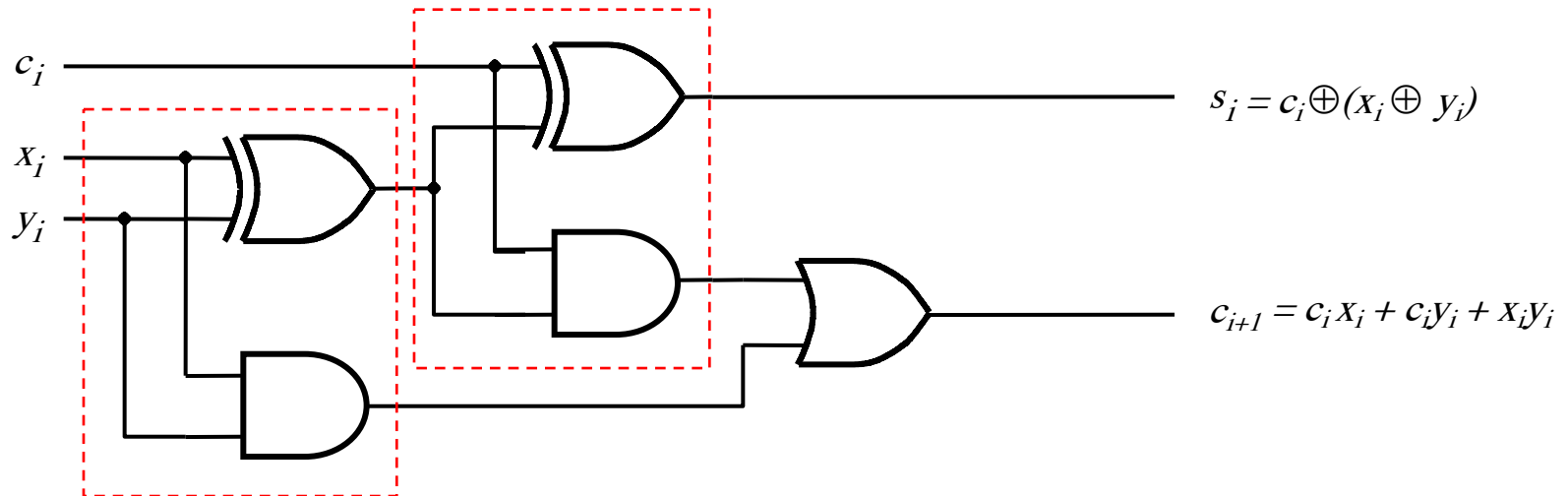


(d) Graphical symbol

# Decomposed full-adder



(a) Block diagram



(b) Detailed diagram

Note that due to circuit resistance and capacitance, gates have a signal “propagation delay” associated with them

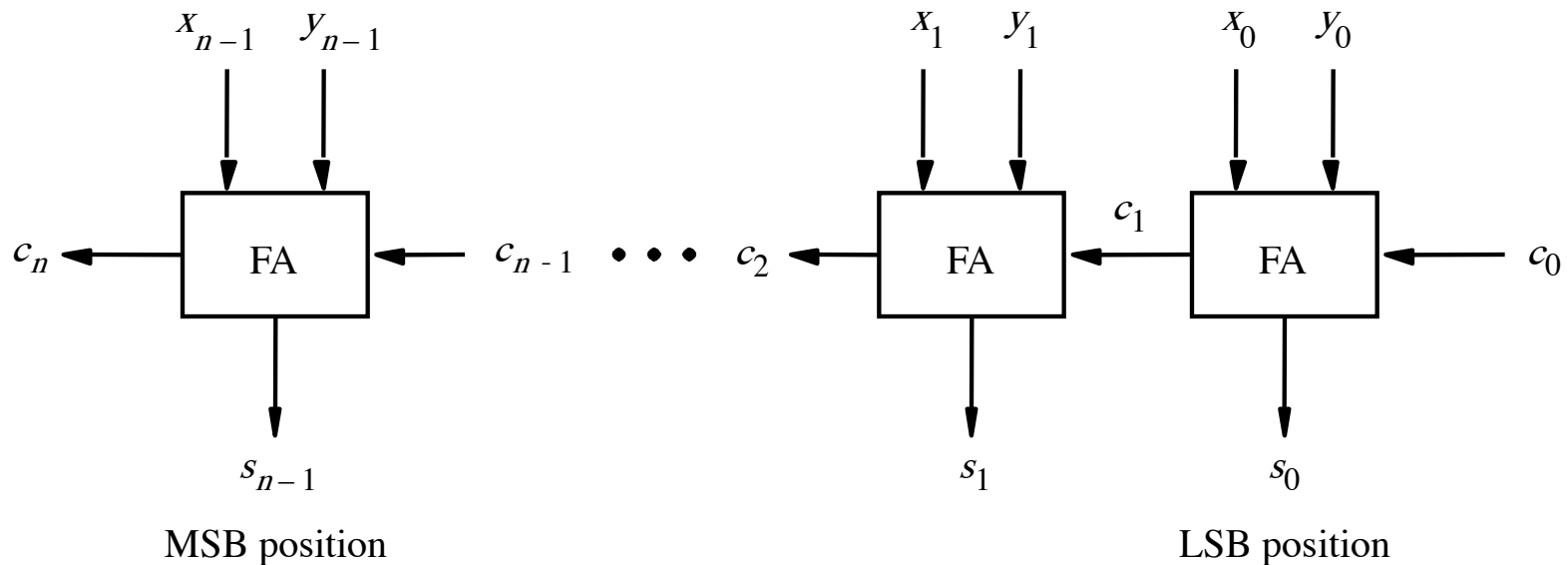
Q1: What is the delay in computing  $s_i$  and  $c_{i+1}$ ?

Q2: AFTER  $s_i$  and  $c_{i+1}$  have settled, what effect does changing any input have?



# An $n$ -bit ripple-carry adder

- Just as in manual decimal addition, binary addition can be performed by adding pairs of bits together starting at the least-significant bit (LSB) position – a carry-out of position  $i$  is added to the operands in position  $i+1$



- When the operands  $X$  and  $Y$  are applied to the adder, it takes some time before the output  $S$  is valid
  - If the time to produce the carry-out from the LSB is  $\Delta t$ , and this delay is identical for all adder stages, then the delay for the carry-out to be produced from the MSB (most significant bit) is  $n\Delta t$
  - The adder derives its name from the manner in which carries ripple through the adder from right to left [imagine changing  $Y = 0..00 \rightarrow 0..01$  after adding  $Y$  to  $X = 01..1$ ]

# VHDL code for a full-adder

```
LIBRARY ieee ;
```

```
USE ieee.std_logic_1164.all ;
```

```
ENTITY fulladd IS
```

```
    PORT ( Cin, x, y          : IN STD_LOGIC ;  
          s, Cout             : OUT STD_LOGIC ) ;
```

```
END fulladd ;
```

```
ARCHITECTURE LogicFunc OF fulladd IS
```

```
BEGIN
```

```
    s <= x XOR y XOR Cin ;
```

```
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
```

```
END LogicFunc ;
```

# VHDL code for a four-bit adder

Declare internal wires/signals

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y : IN STD_LOGIC ;
          s, Cout : OUT STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR
            (Cin AND y) ;
END LogicFunc ;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin : IN STD_LOGIC ;
          x3, x2, x1, x0 : IN STD_LOGIC ;
          y3, y2, y1, y0 : IN STD_LOGIC ;
          s3, s2, s1, s0 : OUT STD_LOGIC ;
          Cout : OUT STD_LOGIC ) ;
END adder4 ;
```

Need to include ieee.std\_logic library before each entity declaration within a source file

```
ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN STD_LOGIC ;
              s, Cout : OUT STD_LOGIC ) ;
    END COMPONENT ;
```

Component declaration

```
BEGIN
    stage0: fulladd -- e.g. positional association
        PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd
        PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd
        PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd -- e.g. named association
        PORT MAP ( Cin => c3,
                  Cout => Cout,
                  x => x3,
                  y => y3,
                  s => s3 ) ;
```

```
END Structure ;
```

Component instantiations; Note: two port association methods

Subcomponent can also appear in a separate file in the “working” (current project) directory

# Declaration of a user-defined package

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
...  
-- your component descriptions (entities & architectures)  
...  
PACKAGE fulladd_package IS  
    COMPONENT fulladd  
        PORT ( Cin, x, y      : IN STD_LOGIC ;  
              s, Cout       : OUT STD_LOGIC ) ;  
    END COMPONENT ;  
END fulladd_package ;
```

- Avoids need to declare the component in the architecture part
- Allows component to be reused across multiple projects
- The package declaration can appear at the end of the source file containing the component(s)

# Instantiation of component using package

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
USE work.fulladd_package.all ; -- user defined package defined in the  
                                -- current, "working" directory
```

```
ENTITY adder4 IS  
    PORT ( Cin           : IN      STD_LOGIC ;  
           x3, x2, x1, x0 : IN      STD_LOGIC ;  
           y3, y2, y1, y0 : IN      STD_LOGIC ;  
           s3, s2, s1, s0 : OUT     STD_LOGIC ;  
           Cout          : OUT     STD_LOGIC ) ;  
END adder4 ;
```

```
ARCHITECTURE Structure OF adder4 IS  
    SIGNAL c1, c2, c3 : STD_LOGIC ;  
BEGIN  
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;  
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;  
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;  
    stage3: fulladd PORT MAP (  
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;  
END Structure ;
```

# Use of multibit signals

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
USE work.fulladd_package.all ;
```

```
ENTITY adder4 IS
```

```
    PORT ( Cin      : IN      STD_LOGIC ;  
          X, Y      : IN      STD_LOGIC_VECTOR(3 DOWNT0 0) ;  
          S         : OUT     STD_LOGIC_VECTOR(3 DOWNT0 0) ;  
          Cout      : OUT     STD_LOGIC ) ;
```

```
END adder4 ;
```

```
ARCHITECTURE Structure OF adder4 IS
```

```
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
```

```
BEGIN
```

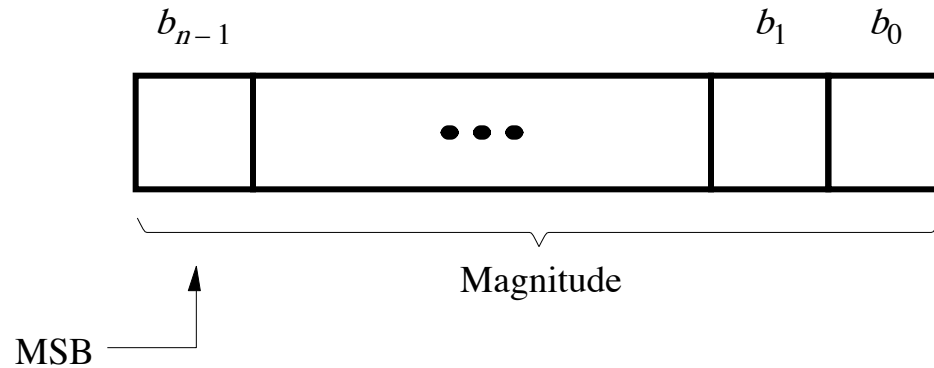
```
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;  
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;  
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;  
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
```

```
END Structure ;
```

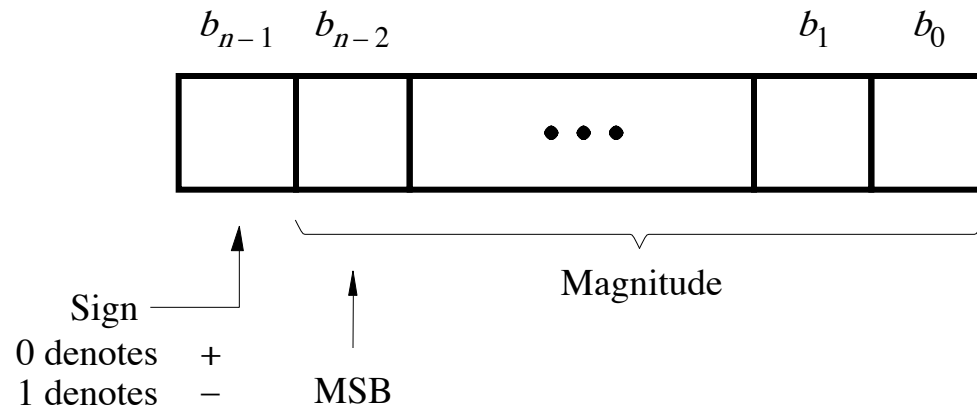
Declaration of signals  
representing numeric values

Declaration of collected,  
unrelated or non-numeric signals

# Signed numbers



(a) Unsigned number



(b) Signed number

# Potential signed number representations

- Sign and magnitude
  - Use the same positional number representation as for unsigned numbers with a sign bit in the left-most position
- 1's complement
  - Derive an  $n$ -bit negative number  $K_1$  by subtracting the equivalent positive number  $P$  from  $2^n - 1$  i.e.  $K_1 = (2^n - 1) - P$
  - Equivalent to flipping every bit
- 2's complement (*preferred, as we shall see*)
  - Derive an  $n$ -bit negative number  $K_2$  by subtracting the equivalent positive number  $P$  from  $2^n$  i.e.  $K_2 = 2^n - P = K_1 + 1$
  - Algorithm for deriving 2's complement:
    - Given a signed number  $B = b_{n-1}b_{n-2}...b_1b_0$ , its 2's complement  $K_2 = k_{n-1}k_{n-2}...k_1k_0$  can be obtained by examining the bits of  $B$  from right to left and taking the following action:
      - copy all bits of  $B$  that are 0 and the first bit that is 1;*
      - then simply complement all the remaining bits on the left*



# Interpreting $B=b_3b_2b_1b_0$ as a signed integer

**Table 5.1** Interpretation of four-bit signed integers.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

- Note the two representations for zero in the sign and magnitude and 1's complement forms
- Note also the larger range that can be represented using 2's complement

# Addition with *sign and magnitude* representation

- If both operands have the same sign, then the addition is simple – add the magnitudes and preserve the sign
  - This is only trouble free if the sum can be represented given the number of bits available for the magnitude
- If the operands have opposite signs, the magnitude of the smaller number has to be subtracted from the magnitude of the larger one
  - Logic for comparing and subtracting numbers is also needed
  - Hence unattractive for use in computers

# 1's complement addition

$$\begin{array}{r} (+5) \\ + (+2) \\ \hline (+7) \end{array} \quad \begin{array}{r} 0101 \\ + 0010 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} (-5) \\ + (+2) \\ \hline (-3) \end{array} \quad \begin{array}{r} 1010 \\ + 0010 \\ \hline 1100 \end{array}$$

$$\begin{array}{r} (+5) \\ + (-2) \\ \hline (+3) \end{array} \quad \begin{array}{r} 0101 \\ + 1101 \\ \hline 10010 \\ \text{Carry } 1 \rightarrow \\ \hline 0011 \end{array}$$

$$\begin{array}{r} (-5) \\ + (-2) \\ \hline (-7) \end{array} \quad \begin{array}{r} 1010 \\ + 1101 \\ \hline 10111 \\ \text{Carry } 1 \rightarrow \\ \hline 1000 \end{array}$$

- While generating a negative number is easy, sometimes a correction is involved in the addition
  - Addition then takes twice as long as for two unsigned numbers

# 2's complement addition


$$\begin{array}{r} (+5) \quad 0101 \\ + (+2) \quad 0010 \\ \hline (+7) \quad 0111 \end{array}$$

$$\begin{array}{r} (-5) \quad 1011 \\ + (+2) \quad 0010 \\ \hline (-3) \quad 1101 \end{array}$$

$$\begin{array}{r} (+5) \quad 0101 \\ + (-2) \quad 1110 \\ \hline (+3) \quad 10011 \end{array}$$

  
ignore

$$\begin{array}{r} (-5) \quad 1011 \\ + (-2) \quad 1110 \\ \hline (-7) \quad 11001 \end{array}$$

  
ignore

- Straightforward addition not involving complications

# 2's complement subtraction

- Most readily done by negating the subtrahend and adding it to the minuend
  - Involves finding 2's complement of the subtrahend and then performing addition
- As we ignore carry-outs of the MSB, the same adder circuit can be used for both addition and subtraction

$$\begin{array}{r}
 (+5) \quad 0101 \\
 - (+2) \quad \underline{0010} \\
 (+3)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 0101 \\
 + 1110 \\
 \hline
 10011
 \end{array}$$

↑  
ignore

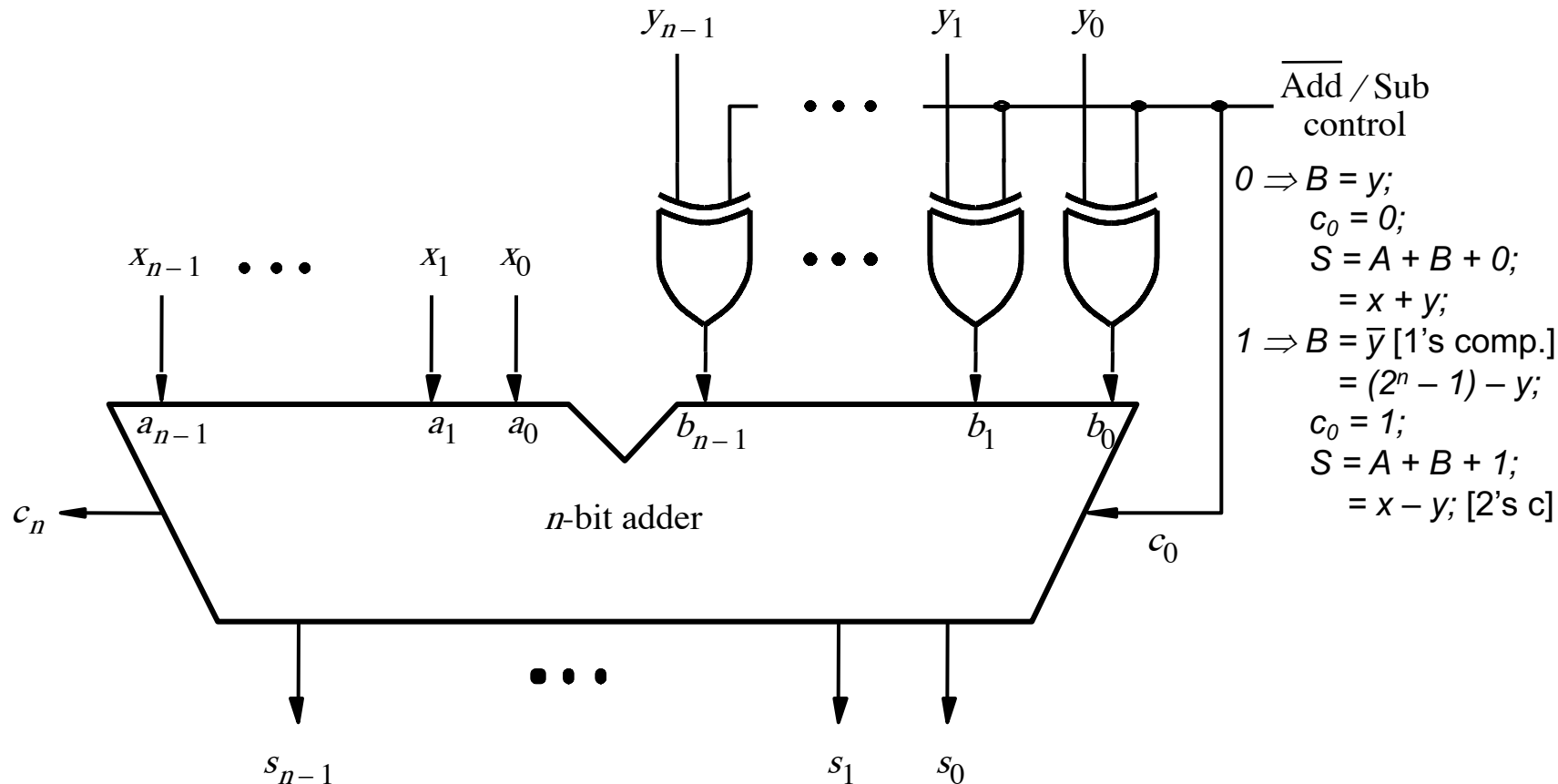
$$\begin{array}{r}
 (-5) \quad 1011 \\
 - (+2) \quad \underline{0010} \\
 (-7)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 1011 \\
 + 1110 \\
 \hline
 11001
 \end{array}$$

↑  
ignore

$$\begin{array}{r}
 (+5) \quad 0101 \\
 - (-2) \quad \underline{1110} \\
 (+7)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 0101 \\
 + 0010 \\
 \hline
 0111
 \end{array}$$

$$\begin{array}{r}
 (-5) \quad 1011 \\
 - (-2) \quad \underline{1110} \\
 (-3)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 1011 \\
 + 0010 \\
 \hline
 1101
 \end{array}$$

# Adder/subtractor unit



Good design rule: More generally, (mech) components, (SW) functions, etc.

Develop **circuits** to be as flexible as possible and exploit common portions for as many tasks as possible

- Minimizes the number of gates needed and the wiring complexity

# Performance issues

- The speed of a circuit is limited by the longest delay along the paths through the circuit
  - In the case of the adder/subtractor circuit of slide L03/S22, the longest delay is along the path from input  $y_0$  through the XOR gate and through the carry circuit of each adder stage
- The longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path*
- Better performance can be achieved using faster circuits by either:
  1. Using superior gate technology (process/technology innovation), or
  2. Changing the overall structure of a functional unit (architectural innovation).

# Fast addition: Carry-lookahead addition

- Carry propagation delay can be reduced by quickly evaluating the carry-in for each stage
- Recall, the carry-out for stage  $i$  can be realized as:

$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

which can be factored as:

$$C_{i+1} = x_i y_i + (x_i + y_i) c_i$$

and rewritten as:

$$C_{i+1} = g_i + p_i c_i \quad (1)$$

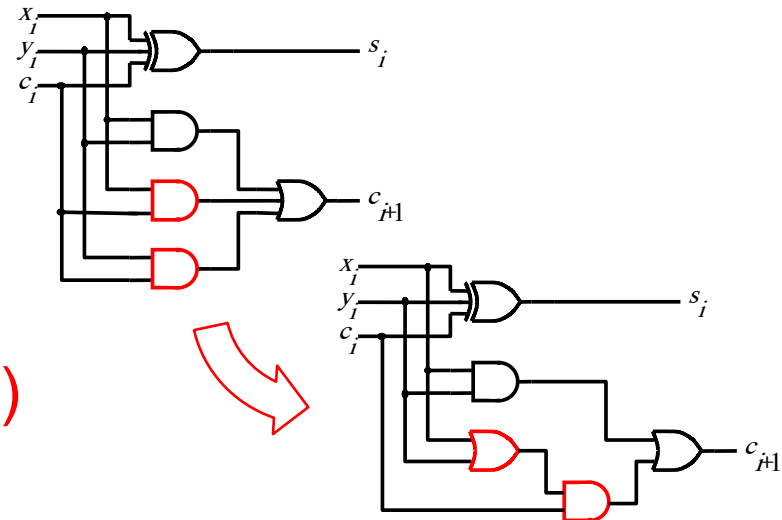
where

$$g_i = x_i y_i$$

carry **generated** when both input bits are 1

$$p_i = x_i + y_i$$

carry **propagated** when either input bit is 1





# Carry-lookahead addition

- Expanding the previous expression in terms of stage  $i-1$  gives:

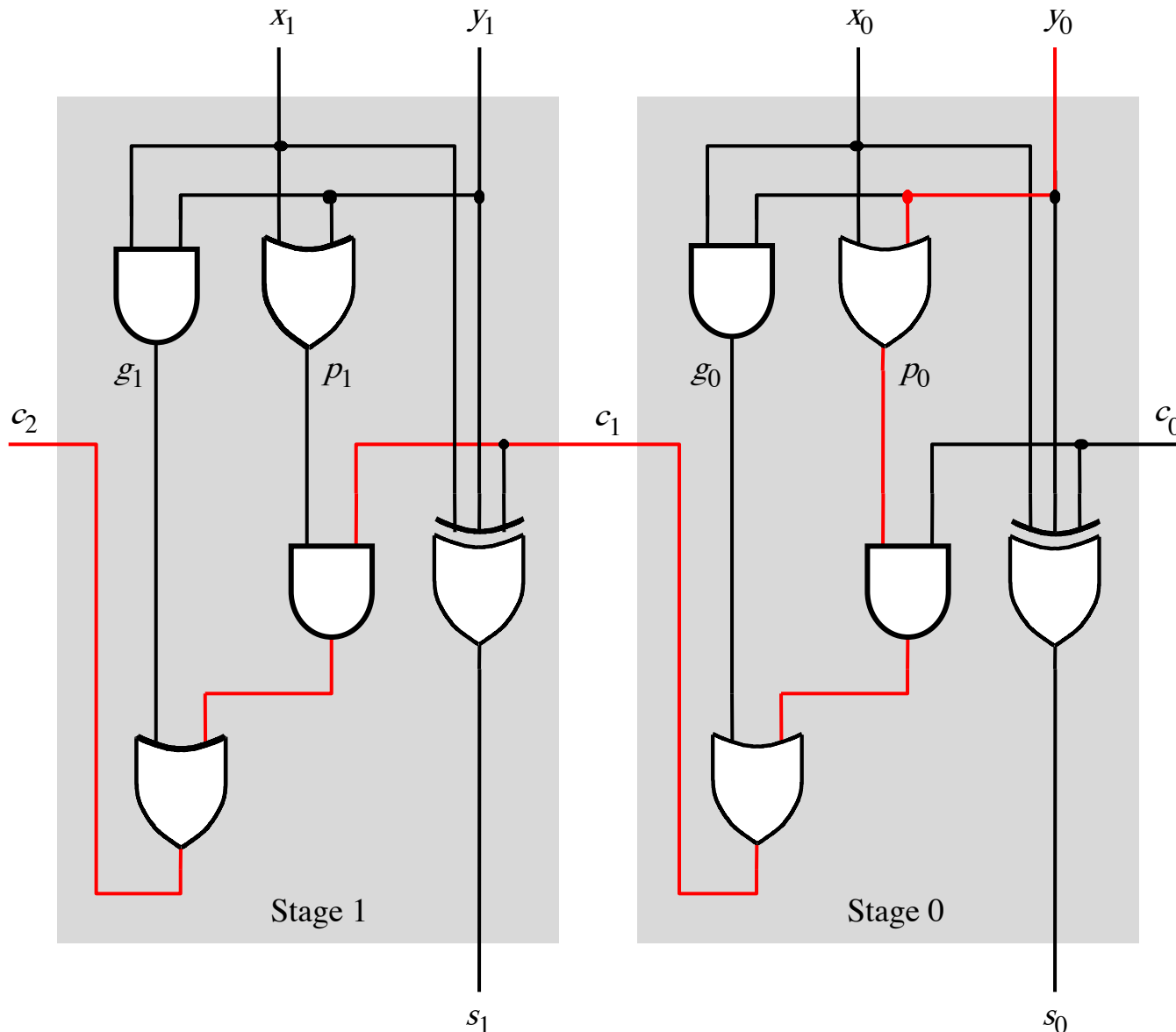
$$\begin{aligned}c_{i+1} &= g_i + p_i c_i \\&= g_i + p_i (g_{i-1} + p_{i-1} c_{i-1}) \\&= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\c_{i+1} &= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \dots \\&\quad + p_i p_{i-1} \dots p_2 p_1 g_0 + p_i p_{i-1} \dots p_1 p_0 c_0 \quad (2)\end{aligned}$$

which represents a *two-level AND-OR circuit* in which  $c_{i+1}$  is evaluated more quickly.

- An adder based on this expression is called a *carry-lookahead adder*

# A ripple-carry adder based on L03/S24 (1)

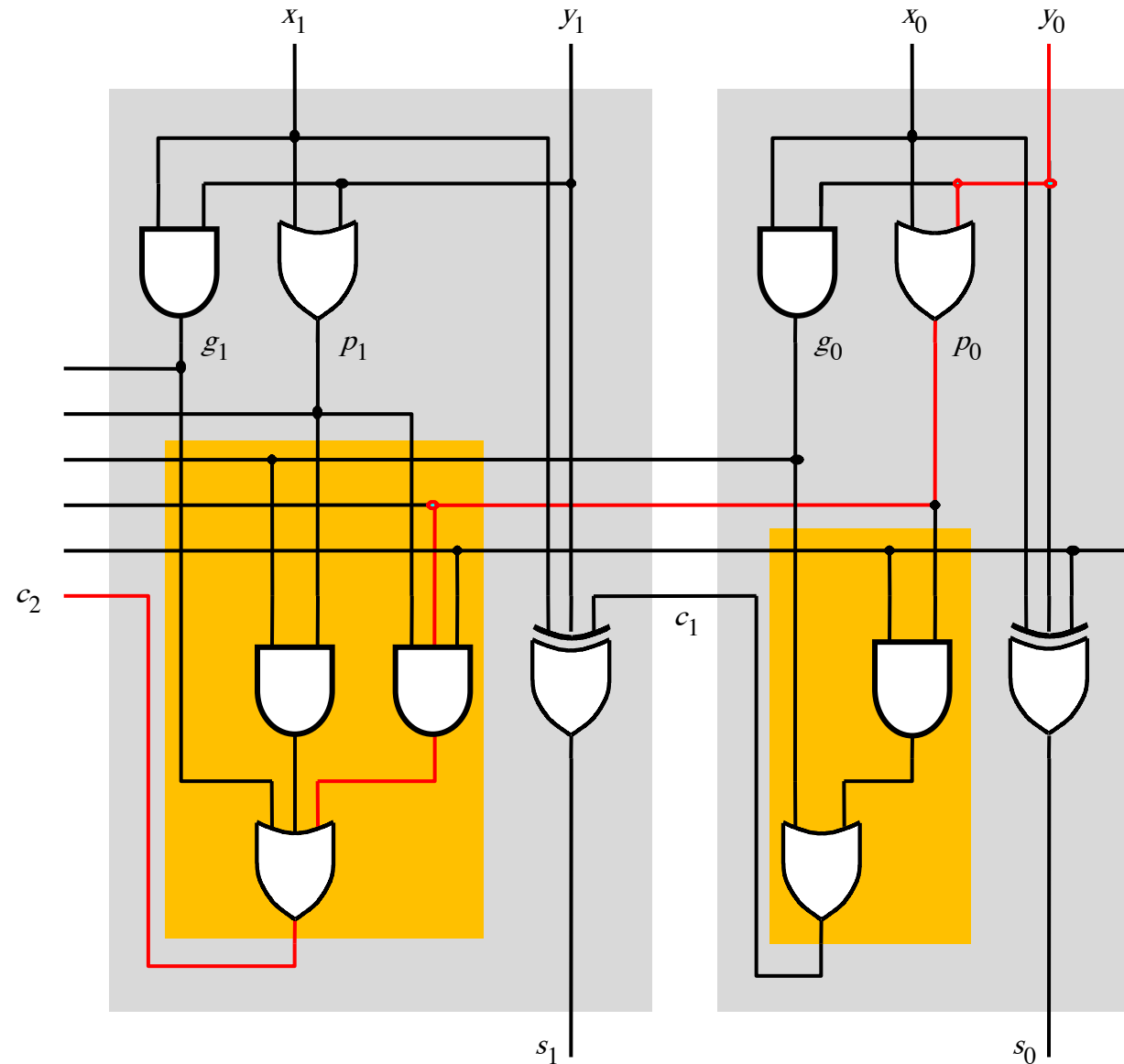
$$c_{i+1} = g_i + p_i c_i$$



- Replaces one AND gate in FA circuit (L03/S7) with an OR gate
- Critical path length (*shown in red*)  $2n+1$  gates:
  - All  $g_i$  &  $p_i$  signals available after 1 gate delay
  - Each stage adds 2 more gate delays to compute  $c_{i+1}$

# Carry-lookahead adder based on L03/S25 (2)

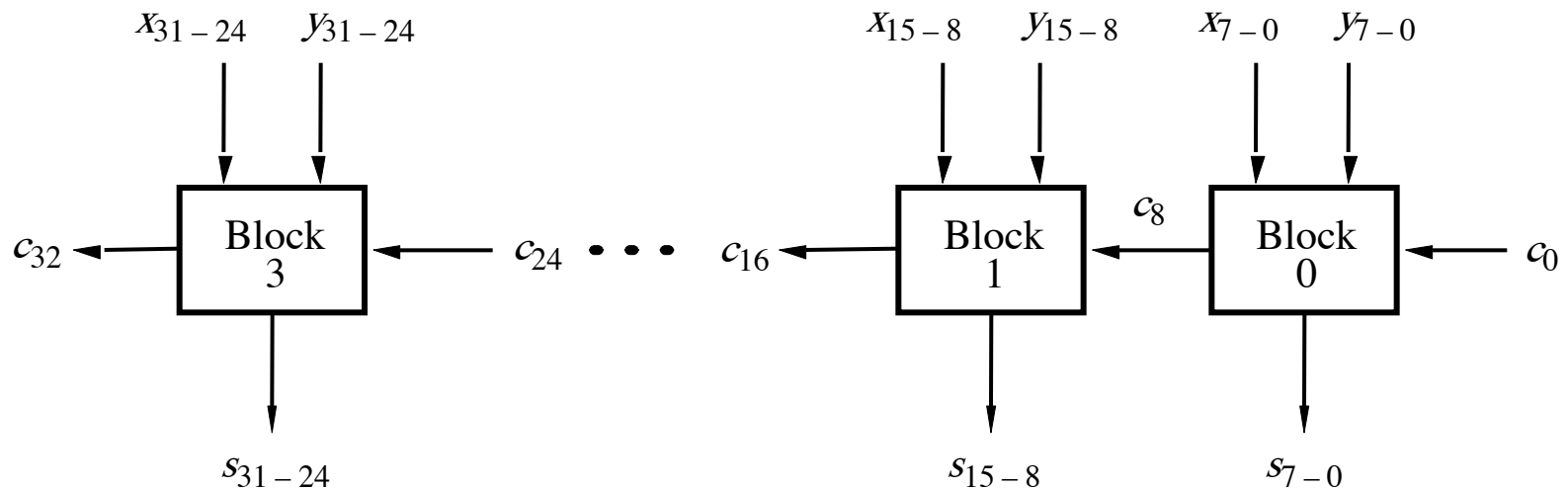
$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \dots + p_i p_{i-1} \dots p_2 p_1 g_0 + p_i p_{i-1} \dots p_1 p_0 c_0$$



- All carries produced at the same time after 3 gate delays
- All sum bits computed in 4 gate delays i.e. 1 gate delay after carries determined
- The complexity grows at every stage
  - 2 more wires enter/leave per stage
  - 1 more AND gate with one more input per stage
  - One more input to OR gate per stage

# Carry-lookahead adder (CLA) with ripple-carry between blocks

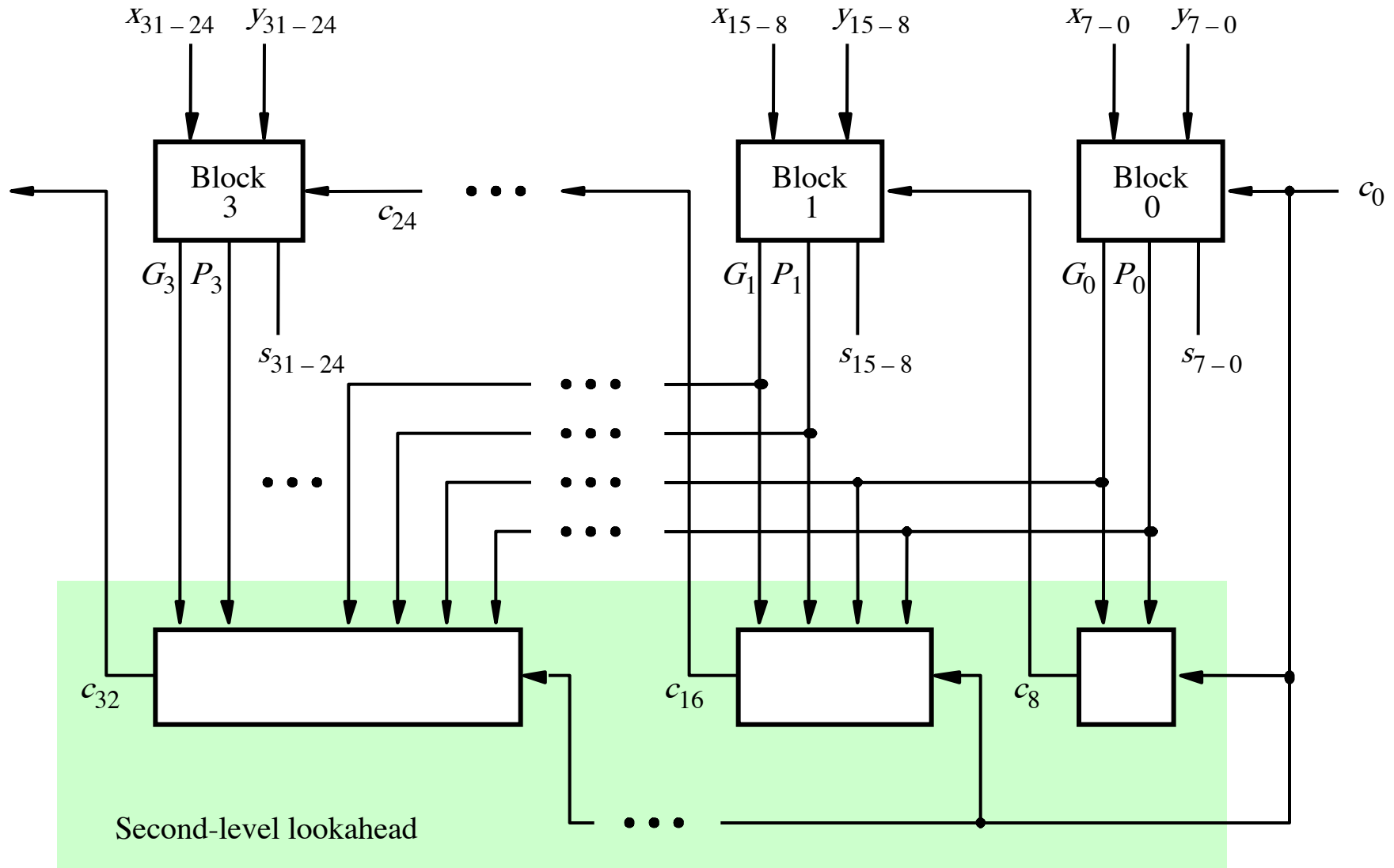
- The complexity of an  $n$ -bit CLA grows rapidly with  $n$
- Thus, use a hierarchical approach to implement large adders
- For example, split a 32-bit adder into four 8-bit blocks
  - Each block can be a CLA
  - Interconnect blocks *either* in ripple-carry fashion:



**Q: What is the critical path delay of this adder architecture?**

# Hierarchical carry-lookahead addition

...*or* using a second level of carry-lookahead:



# Hierarchical carry-lookahead addition

- Uses a second (or more) levels of carry-lookahead to reduce the time to produce carry signals between blocks
- Each block produces its own “block” generate and propagate signals instead of carries-outs – denote these as  $G_j$  &  $P_j$  for block  $j$
- We then have for block 0

$$\begin{aligned}c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + \dots + p_7p_6\dots p_2p_1g_0 + p_7p_6\dots p_1p_0c_0 \\ &= G_0 + P_0c_0\end{aligned}$$

with

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \dots + p_7p_6\dots p_2p_1g_0$$

$$P_0 = p_7p_6\dots p_1p_0$$

Then

$$c_{16} = G_1 + P_1c_8 = G_1 + P_1G_0 + P_1P_0c_0$$

and

$$c_{24} = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$

$$c_{32} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Note that higher numbered  $G_j$  &  $P_j$  just renumber the indices in the expressions for  $G_0$  &  $P_0$

# Hierarchical carry-lookahead addition

- This scheme takes two more gate delays to produce the carry signals  $c_8$ ,  $c_{16}$  &  $c_{24}$  than the time needed to compute the block  $G_j$  and  $P_j$  functions (3 gate delays)
- Two more gate delays are needed to produce the carry signals for each bit within each block, and one more to produce the sums bits
  - After inputs change, a total of 8 gate delays are required to compute the result (critical path computes  $s_{32}$ )
  - In contrast, a 32-bit ripple-carry adder needs 64 gate delays to compute a result (critical path computes  $c_{32}$ )
- IN PRACTICE:
  - Gate fan-in/technology limitations force us to use multilevel implementations of the generate and propagate functions unless the blocking factor (number of bits added per block and hierarchy level) is kept low [see p. 278 for gory details]

# Detecting overflow

$$\begin{array}{r}
 (+7) \quad 0111 \\
 + (+2) \quad 0010 \\
 \hline
 (+9) \quad 1001 \\
 c_4 = 0 \\
 c_3 = 1
 \end{array}$$

$$\begin{array}{r}
 (-7) \quad 1001 \\
 + (+2) \quad 0010 \\
 \hline
 (-5) \quad 1011 \\
 c_4 = 0 \\
 c_3 = 0
 \end{array}$$

$$\begin{array}{r}
 (+7) \quad 0111 \\
 + (-2) \quad 1110 \\
 \hline
 (+5) \quad 10101 \\
 c_4 = 1 \\
 c_3 = 1
 \end{array}$$

$$\begin{array}{r}
 (-7) \quad 1001 \\
 + (-2) \quad 1110 \\
 \hline
 (-9) \quad 10111 \\
 c_4 = 1 \\
 c_3 = 0
 \end{array}$$

- When the carry-out of the MSB  $\neq$  carry-out of the sign-bit position, overflow has occurred:

$$\text{Overflow} = \bar{c}_3 c_4 + c_3 \bar{c}_4 = c_3 \oplus c_4$$

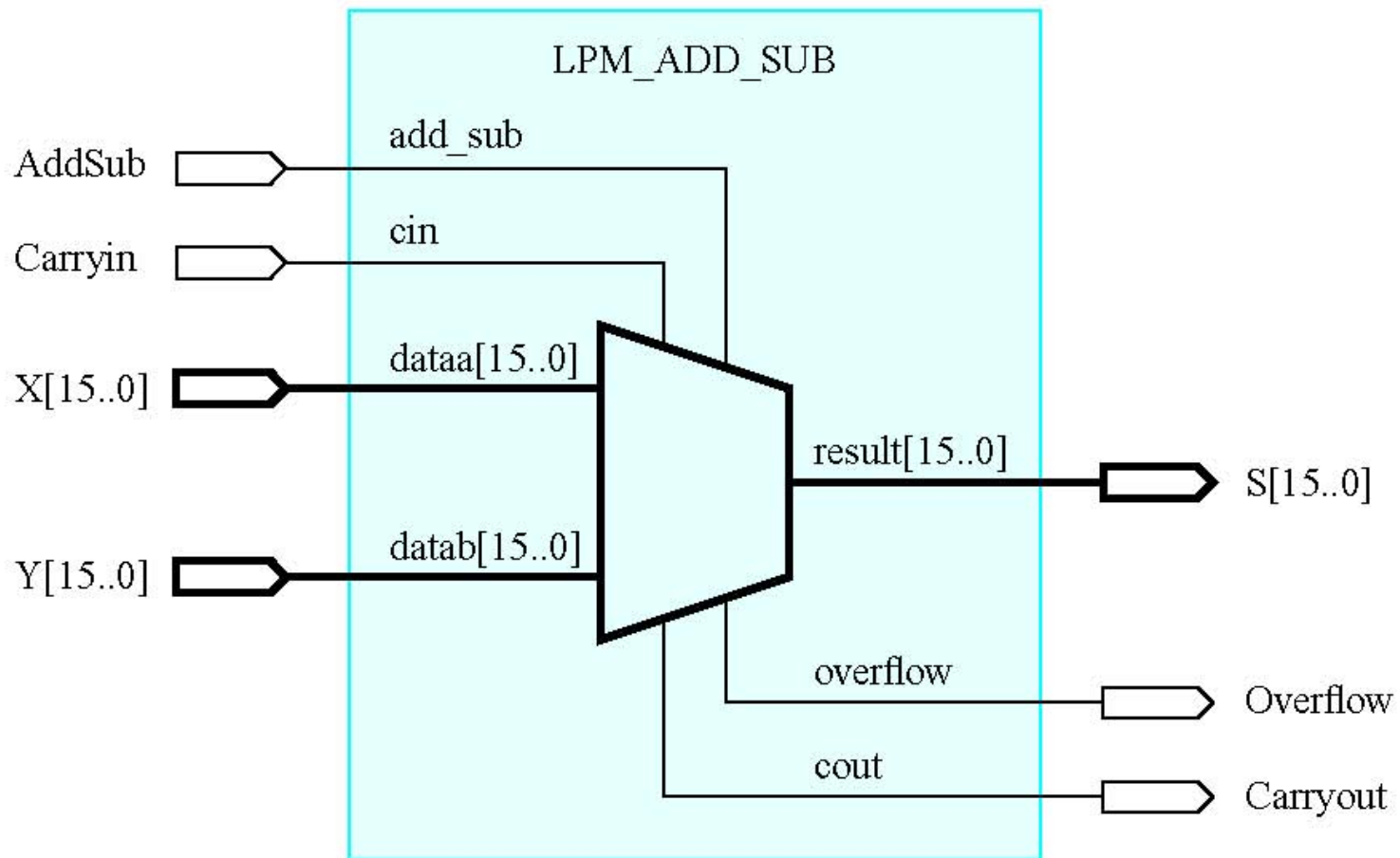
- For n-bit numbers:

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

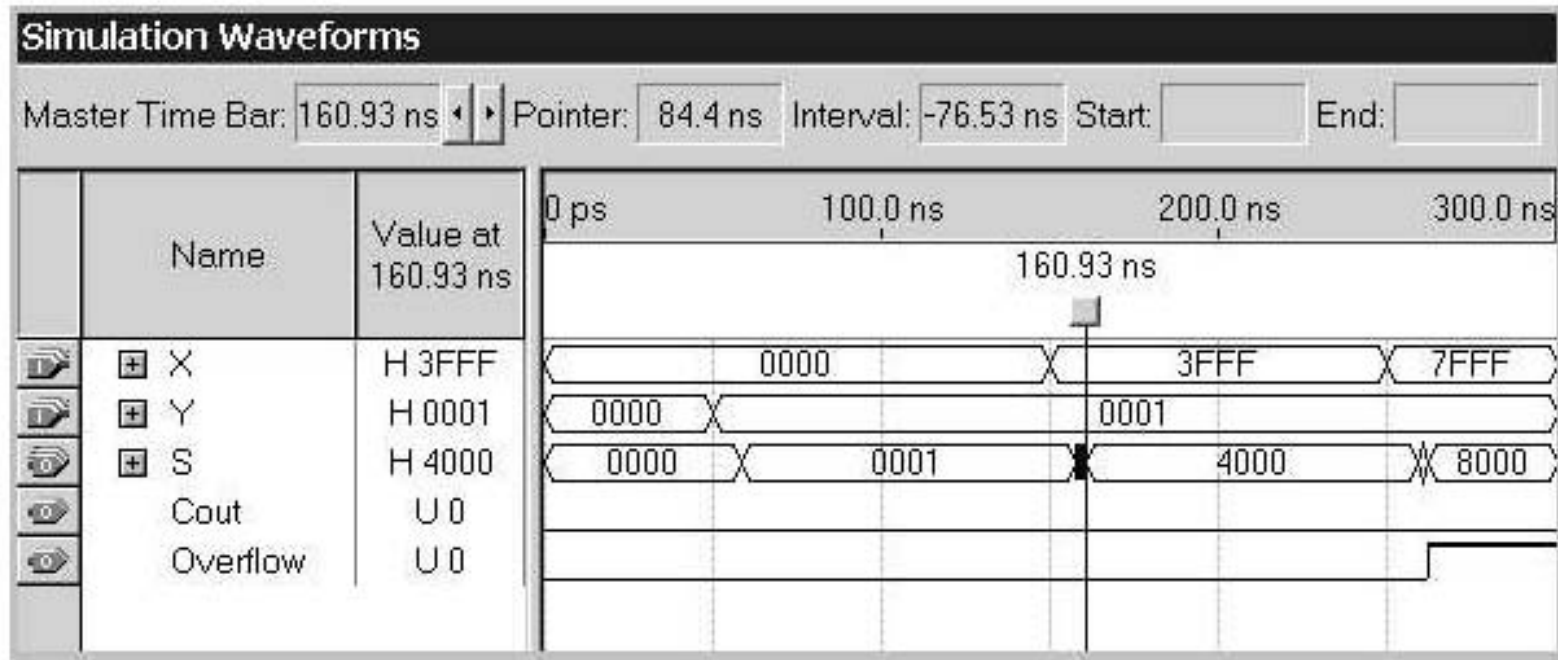


# Design of arithmetic circuits using schematic capture

- Use of Altera's *Library of Parameterized Modules*
  - Bit width and signed/unsigned representation controllable
    - Here parameterized for 16-bit wide addition



# Timing simulation of an LPM adder



- Note the delay observed in updating the sum when an input changes
  - A period of glitching in the S output occurs as carries ripple through the adder

# Behavioural code for a 16-bit adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( X, Y      : IN      STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S          : OUT     STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ; -- math ops +, - and * are usually automatically synthesized
END Behavior ;
```

- Note use of std\_logic\_signed package
  - Defines arithmetic ops on signed data types
  - std\_logic\_unsigned does so for data of type unsigned
  - LPM component of the required size automatically inferred

# Accessing the internal signals

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
USE ieee.std_logic_signed.all ;
```

```
ENTITY adder16 IS
```

```
    PORT ( Cin      : IN    STD_LOGIC ;  
          X, Y      : IN    STD_LOGIC_VECTOR(15 DOWNT0 0) ;  
          S          : OUT   STD_LOGIC_VECTOR(15 DOWNT0 0) ;  
          Cout, Overflow : OUT STD_LOGIC ) ;
```

```
END adder16 ;
```

```
ARCHITECTURE Behavior OF adder16 IS
```

```
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNT0 0) ;
```

```
BEGIN
```

```
    Sum <= ('0' & X) + ('0' & Y) + Cin ; -- note concatenation operator used
```

```
    S <= Sum(15 DOWNT0 0) ;
```

```
    Cout <= Sum(16) ;
```

```
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
```

```
END Behavior ;
```

C(15)

# Using inbuilt INTEGER signals

```
ENTITY adder16 IS
    PORT ( X, Y    : IN    INTEGER RANGE -32768 TO 32767 ;
           S      : OUT   INTEGER RANGE -32768 TO 32767 ) ;
END adder16 ;
```

```
ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

- Does not require library as INTEGER is an inbuilt type
- But, accessing internal signals, so as to determine overflow, is impossible