Functional Blocks: Addition

- **Binary addition used frequently**
- **Addition Development:**
  - ***Half-Adder* (HA), a 2-input bit-wise addition functional block,**
  - ***Full-Adder* (FA), a 3-input bit-wise addition functional block,**
  - ***Ripple Carry Adder*, an iterative array to perform <u>binary addition</u>, and**
  - ***Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.**

Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

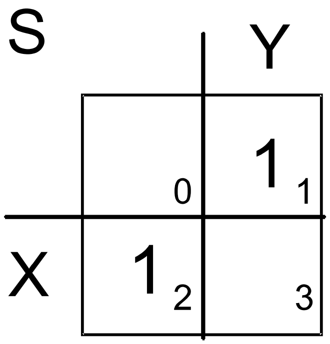
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

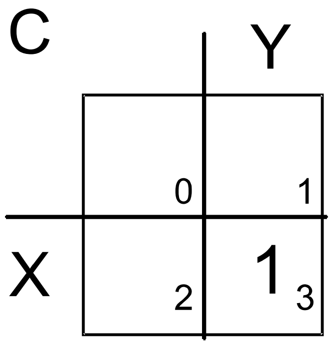
- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a sum bit , S and a carry bit, C
- The half adder can be specified as a truth table for S and C ⇒

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

Logic Simplification: Half-Adder

- The K-Map for S, C is:
- This is a pretty trivial map! By inspection:





$$S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y$$

$$S = (X + Y) \cdot (\overline{X} + \overline{Y})$$

- and

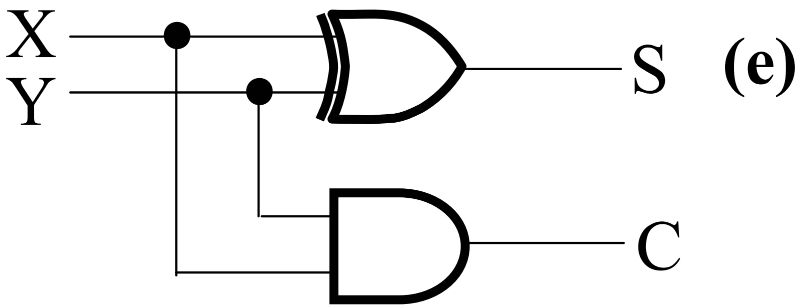
$$C = X \cdot Y$$

$$C = \overline{(\overline{(X \cdot Y)})}$$

Implementations: Half-Adder

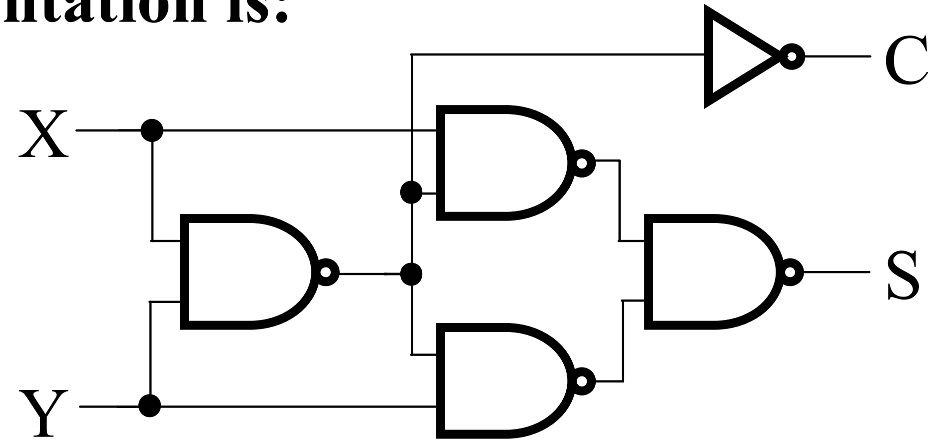
- **The most common half adder implementation is:**

$$S = X \oplus Y$$
$$C = X \cdot Y$$



- **A NAND only implementation is:**

$$S = (X + Y) \cdot C$$
$$C = \overline{(\overline{(X \cdot Y)})}$$



Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.
  - For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

  - For a carry- in (Z) of 1:

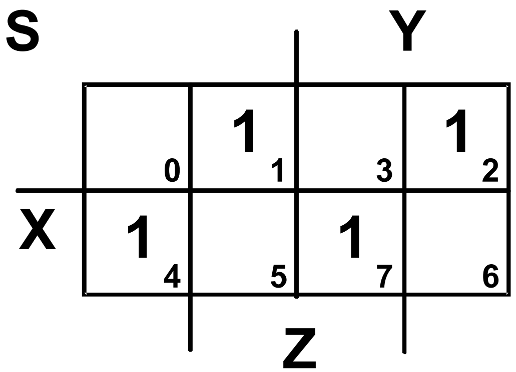
Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

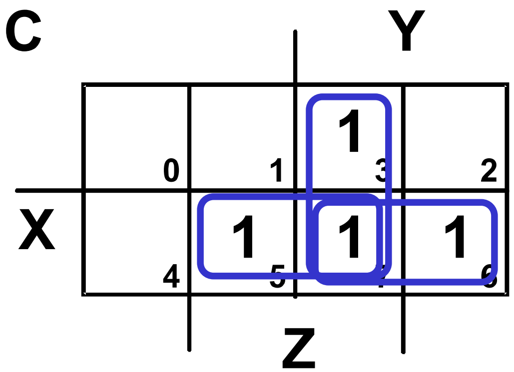
Logic Optimization: Full-Adder

- **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

- **Full-Adder K-Map:**





Equations: Full-Adder

- From the K-Map, we get:

$$S = X\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z + XYZ$$
$$C = XY + XZ + YZ$$

- The S function is the three-bit XOR function (Odd Function):

$$S = X \oplus Y \oplus Z$$

- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

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

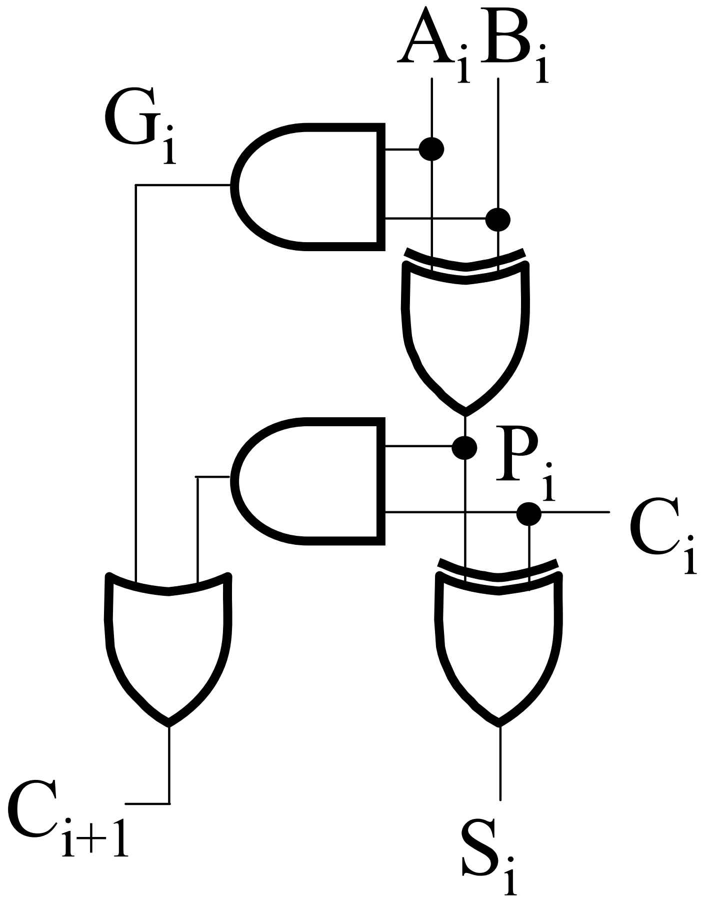
- The term $X \cdot Y$ is *carry generate*.
- The term $X \oplus Y$ is *carry propagate*.

Implementation: Full Adder

- **Full Adder Schematic**
- **Here X, Y, and Z, and C (from the previous pages) are A, B, C_i and C_o, respectively. Also,**

  G = generate and

  P = propagate.
- **Note: This is really a combination of a 3-bit odd function (for S)) and Carry logic (for C_o):**



(G = Generate) OR (P =Propagate AND C_i = Carry In)

Co = G + P · Ci

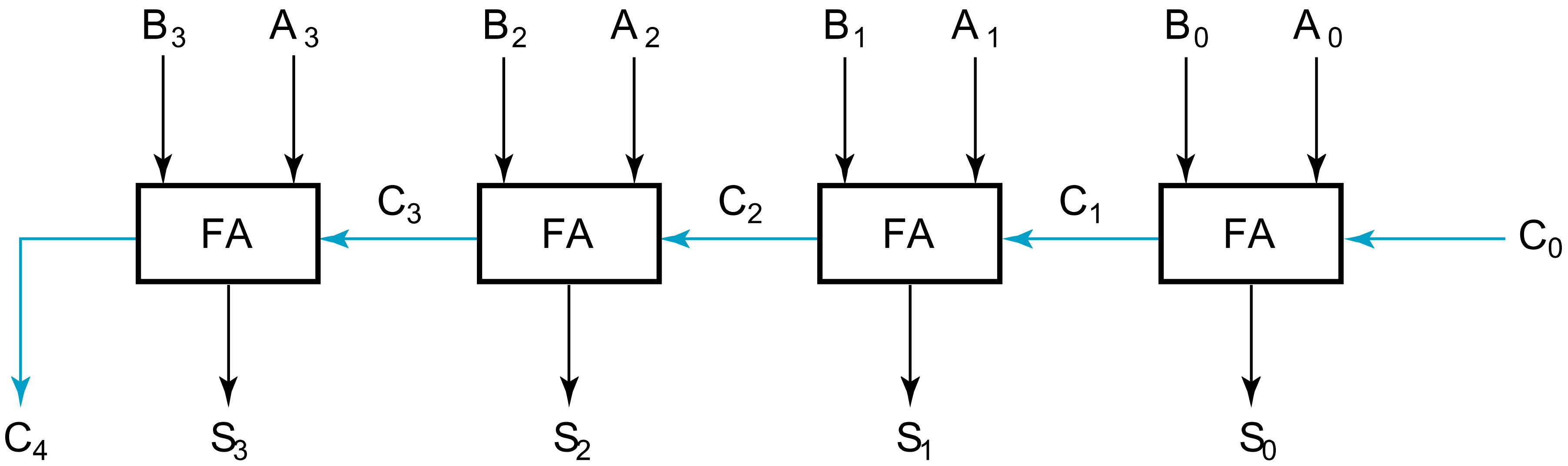
Binary Adders

- To add multiple operands, we "bundle" logical signals together into vectors and use functional blocks that operate on the vectors
- Example: <u>4-bit ripple carry adder:</u> Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)
- Note: carry out of cell i becomes carry in of cell i + 1

Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	C_i
Augend	1 0 1 1	A_i
Addend	0 0 1 1	B_i
Sum	1 1 1 0	S_i
Carry out	0 0 1 1	C_{i+1}

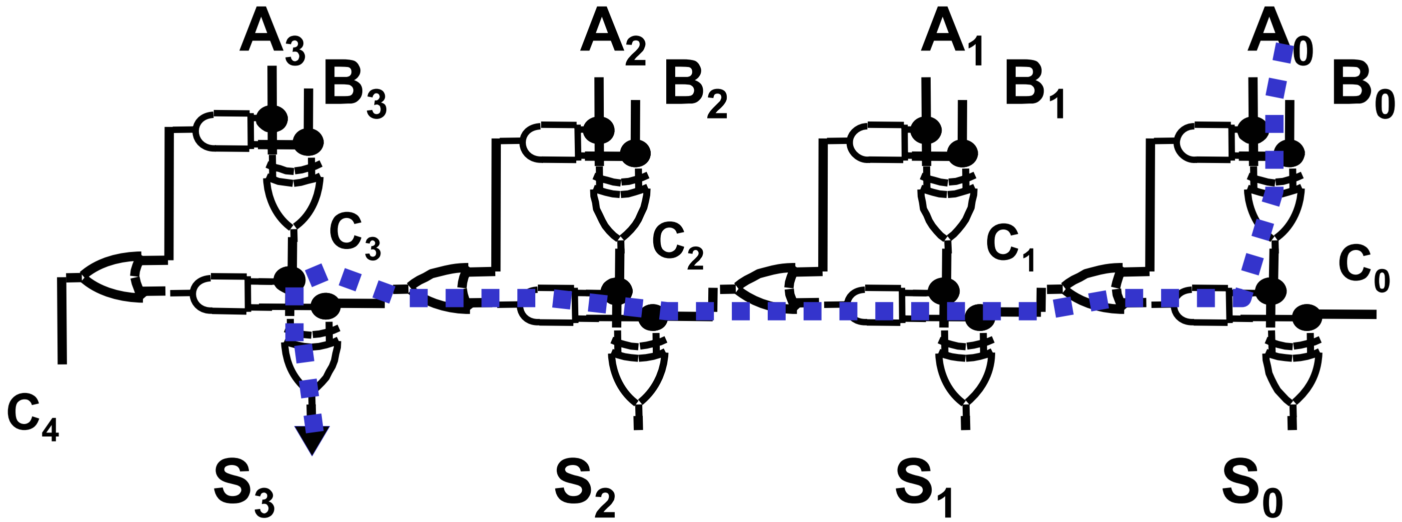
4-bit Ripple-Carry Binary Adder

- **A four-bit Ripple Carry Adder made from four 1-bit Full Adders:**



Carry Propagation & Delay

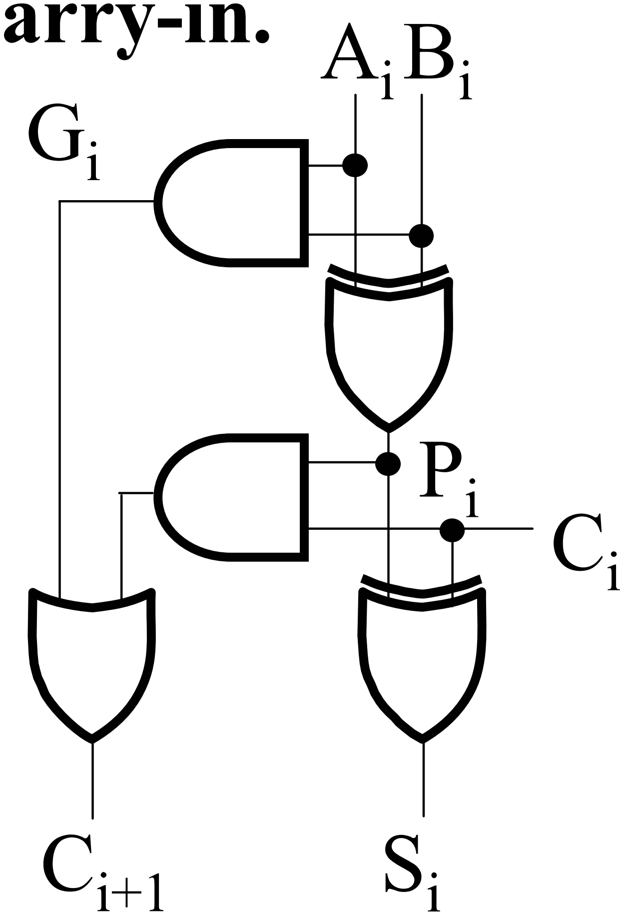
- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:



- Note: The "long path" is from A_0 or B_0 though the circuit to S_3.

Carry Lookahead

- Given Stage i from a Full Adder, we know that there will be a carry generated when $A_i = B_i =$ "1", whether or not there is a carry-in.
- Alternately, there will be a carry propagated if the "half-sum" is "1" and a carry-in, C_i occurs.
- These two signal conditions are called *generate*, denoted as G_i, and *propagate*, denoted as P_i respectively and are identified in the circuit:



Carry Lookahead (continued)

- **In the ripple carry adder:**
  - **Gi, Pi, and Si are <u>local</u> to each cell of the adder**
  - **Ci is also local each cell**
- **In the carry lookahead adder, in order to reduce the length of the carry chain, Ci is changed to a more global function spanning multiple cells**
- **Defining the equations for the Full Adder in term of the P_i and G_i:**

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

Carry Lookahead Development

- C_{i+1} can be removed from the cells and used to derive a set of carry equations spanning multiple cells.
- Beginning at the cell 0 with carry in C_0:

$$
\begin{aligned}
C_1 &= G_0 + P_0\,C_0 \\
C_2 &= G_1 + P_1\,C_1 = G_1 + P_1(G_0 + P_0\,C_0) \\
&= G_1 + P_1G_0 + P_1P_0\,C_0 \\
C_3 &= G_2 + P_2\,C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0\,C_0) \\
&= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0\,C_0 \\
C_4 &= G_3 + P_3\,C_3 = G_3 + P_3G_2 + P_3P_2G_1 \\
&\quad + P_3P_2P_1G_0 + P_3P_2P_1P_0\,C_0
\end{aligned}
$$

Group Carry Lookahead Logic

- **Figure 5-6 in the text shows shows the implementation of these equations for four bits. This could be extended to more than four bits; in practice, due to limited gate fan-in, such extension is not feasible.**
- **Instead, the concept is extended another level by considering *group generate* ($G_{0\text{-}3}$) and *group propagate* ($P_{0\text{-}3}$) functions:**

$$G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0$$

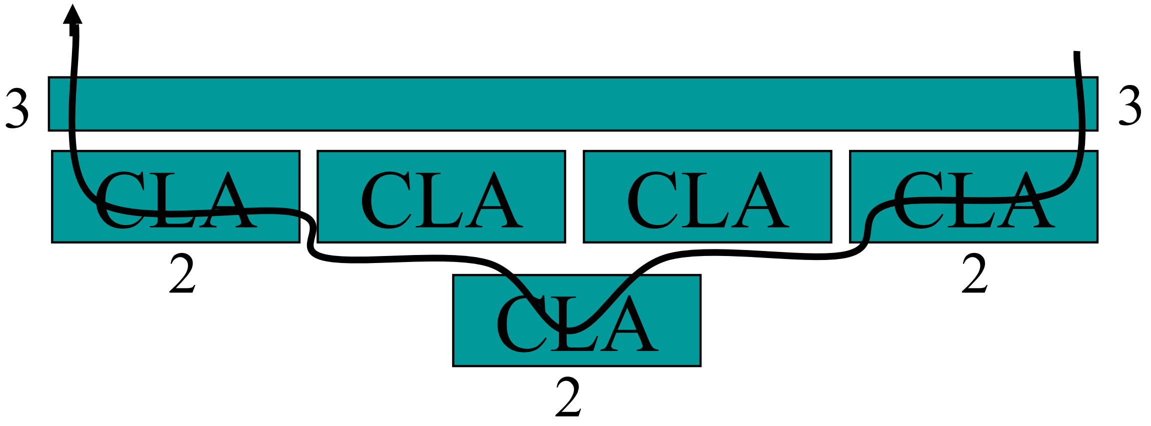
$$P_{0-3} = P_3 P_2 P_1 P_0$$

- **Using these two equations:**

$$C_4 = G_{0-3} + P_{0-3} C_0$$

- **Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition**

Carry Lookahead Example



- Specifications:
  - 16-bit CLA
  - Delays:
    - NOT = 1
    - XOR = Isolated AND = 3
    - AND-OR = 2
- Longest Delays:
  - Ripple carry adder* = 3 + 15 × 2 + 3 = 36
  - CLA = 3 + 3 × 2 + 3 = 12

*See slide 16

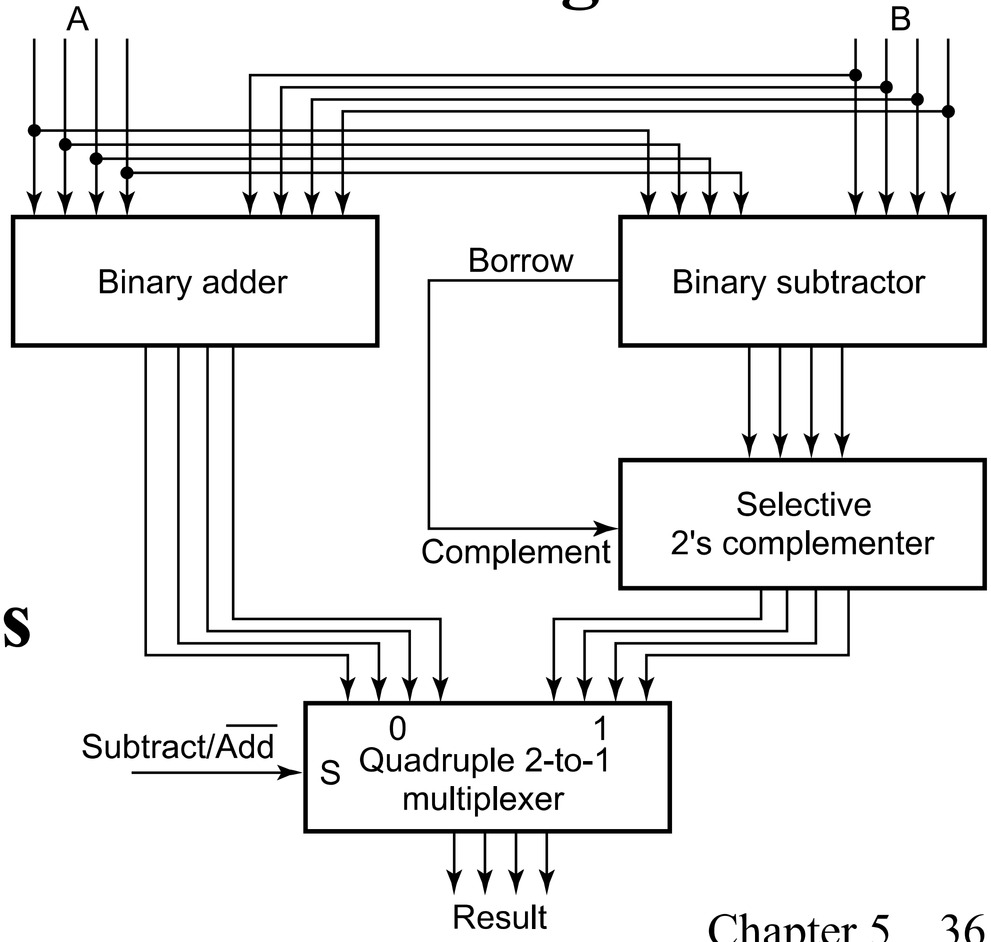
Unsigned Subtraction

- Algorithm:
  - Subtract the subtrahend N from the minuend M
  - If no end borrow occurs, then M ≥ N, and the result is a non-negative number and correct.
  - If an end borrow occurs, the N > M and the difference M – N + 2n is subtracted from 2n, and a minus sign is appended to the result.
- **Examples:**

$$\begin{array}{rr} 0 & 1 \\ 1001 & 0100 \\ -\underline{0111} & -\underline{0111} \\ 0010 & 1101 \\ & 10000 \\ & -\underline{\ 1101} \\ & (-)\ 0011 \end{array}$$

Unsigned Subtraction (continued)

- The subtraction, 2^n – N, is taking the 2's complement of N
- To do both unsigned addition and unsigned subtraction requires:
- Quite complex!
- Goal: Shared simpler logic for both addition and subtraction
- Introduce complements as an approach



Binary 2's Complement

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:

  $(r^n) = 256_{10}$ or 100000000_2

- The 2's complement of 01110011 is then:

$$\begin{array}{r} 100000000 \\ -\ \underline{01110011} \\ 10001101 \end{array}$$

- Note the result is the 1's complement plus 1, a fact that can be used in designing hardware

Subtraction with 2’s Complement

- For n-digit, <u>unsigned</u> numbers M and N, find M – N in base 2:
  - Add the 2's complement of the subtrahend N to the minuend M:
    $$M + (2^n - N) = M - N + 2^n$$
  - If $M \geq N$, the sum produces end carry r^n which is discarded; from above, M – N remains.
  - If $M < N$, the sum does not produce an end carry and, from above, is equal to $2^n - (N - M)$, the 2's complement of (N – M).
  - To obtain the result – (N – M) , take the 2's complement of the sum and place a – to its left.

Unsigned 2’s Complement Subtraction Example 1

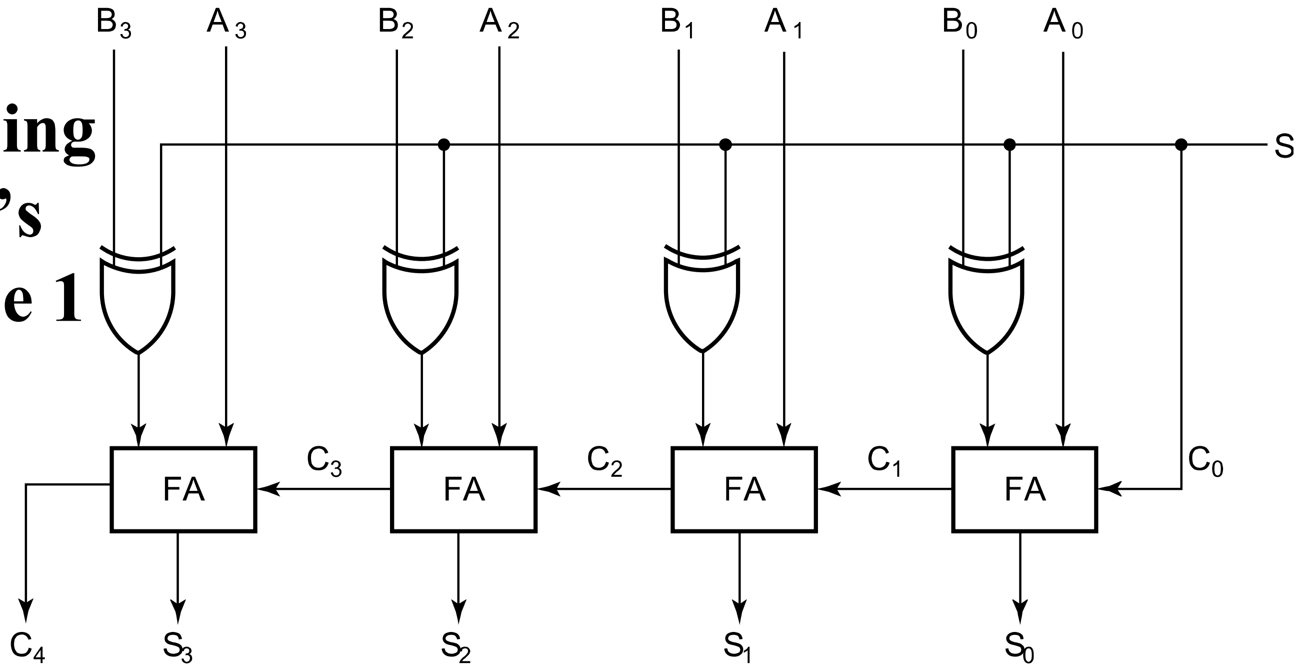
- **Find $01010100_2 - 01000011_2$**

$$\begin{array}{rrrl} & 01010100 & & {}^{1}01010100 \\ - & \underline{01000011} & \xrightarrow{\text{2's comp}} & +\ \underline{10111101} \\ & & & \ \ \ 00010001 \end{array}$$

- **The carry of 1 indicates that no correction of the result is required.**

2’s Complement Adder/Subtractor

- **Subtraction can be done by addition of the 2's Complement.**
  1. **Complement each bit (1's Complement.)**
  2. **Add 1 to the result.**
- **The circuit shown computes A + B and A – B:**
- **For S = 1, subtract, the 2’s complement of B is formed by using XORs to form the 1’s comp and adding the 1 applied to C_0.**
- **For S = 0, add, B is passed through unchanged**



Overflow Detection

- ***Overflow*** **occurs if *n* + 1 bits are required to contain the result from an n-bit addition or subtraction**
- **Overflow can occur for:**
  - **Addition of two operands with the same sign**
  - **Subtraction of operands with different signs**
- **Signed number overflow cases with correct result sign**

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 0 & 0 & 1 & 1 \end{array}$$

- **Detection can be performed by examining the result signs which should match the signs of the top operand**

Overflow Detection

- **Signed number cases with carries C_n and C_{n-1} shown for correct result signs:**

$$\begin{array}{cccc} 0\quad 0 & 0\quad 0 & 1\quad 1 & 1\quad 1 \end{array}$$

$$\begin{array}{rrrr} 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 0 & 0 & 1 & 1 \end{array}$$

- **Signed number cases with carries shown for erroneous result signs (indicating overflow):**

$$\begin{array}{cccc} 0\quad 1 & 0\quad 1 & 1\quad 0 & 1\quad 0 \end{array}$$

$$\begin{array}{rrrr} 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 1 & 1 & 0 & 0 \end{array}$$

- **Simplest way to implement overflow $V = C_n \oplus C_{n-1}$**
- **This works correctly only if 1's complement and the addition of the carry in of 1 is used to implement the complementation! Otherwise fails for – 10 ... 0**

Binary Multiplication

- The binary digit multiplication table is trivial:

(a × b)	b = 0	b = 1
a = 0	0	0
a = 1	0	1

- This is simply the Boolean AND function.
- Form larger products the same way we form larger products in base 10.

Review - Decimal Example: $(237 \times 149)_{10}$

- Partial products are: 237 × 9, 237 × 4, and 237 × 1
- Note that the partial product summation for *n* digit, base 10 numbers requires adding up to *n* digits (with carries).
- Note also $n \times m$ digit multiply generates up to an $m + n$ digit result.

$$
\begin{array}{cccccc}
 & & & 2 & 3 & 7 \\
 & & \times & 1 & 4 & 9 \\
\cline{3-6}
 & & 2 & 1 & 3 & 3 \\
 & & 9 & 4 & 8 & - \\
+ & 2 & 3 & 7 & - & - \\
\hline
 & 3 & 5 & 3 & 1 & 3
\end{array}
$$

Binary Multiplication Algorithm

- We execute radix 2 multiplication by:
  - Computing partial products, and
  - Justifying and summing the partial products. (same as decimal)
- To compute partial products:
  - Multiply the row of multiplicand digits by each multiplier digit, one at a time.
  - With binary numbers, partial products are very simple! They are either:
    - all zero (if the multiplier digit is zero), or
    - the same as the multiplicand (if the multiplier digit is one).
- Note: No carries are added in partial product formation!

Example: (101 x 011) Base 2

- **Partial products are: 101 × 1, 101 × 1, and 101 × 0**
- **Note that the partial product summation for *n* digit, base 2 numbers requires adding up to *n* digits (with carries) in a column.**
- **Note also *n* × *m* digit multiply generates up to an *m* + *n* digit result (same as decimal).**

$$\begin{array}{cccccc} & & & 1 & 0 & 1 \\ & & \times & 0 & 1 & 1 \\ \cline{3-6} & & & 1 & 0 & 1 \\ & & 1 & 0 & 1 & \\ & 0 & 0 & 0 & & \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Multiplier Boolean Equations

- We can also make an *n* × *m* “block” multiplier and use that to form partial products.
- Example: 2 × 2 – The logic equations for each partial-product binary digit are shown below:
- We need to "add" the columns to get the product bits P0, P1, P2, and P3.
- Note that some columns may generate carries.

$$\begin{array}{cccc} & & b_1 & b_0 \\ & \times & a_1 & a_0 \\ \cline{2-4} & & (a_0 \cdot b_1) & (a_0 \cdot b_0) \\ + & (a_1 \cdot b_1) & (a_1 \cdot b_0) & \\ \hline P_3 & P_2 & P_1 & P_0 \end{array}$$

Multiplier Arrays Using Adders

- **An implementation of the 2 × 2 multiplier array is shown:**