Logic and Computer Design Fundamentals

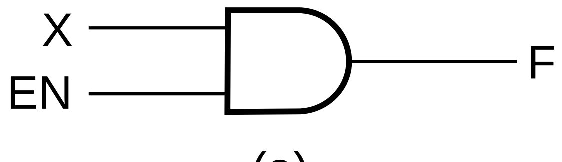
Adders and Multiplication

Overview

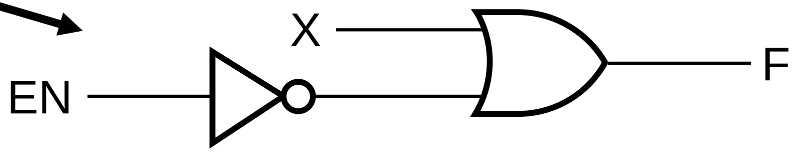
- Iterative combinational circuits
- Binary adders
  - Half and full adders
  - Ripple carry and carry lookahead adders
- Binary subtraction
- Binary multiplication
- Other arithmetic functions

Enabling Function

- ***Enabling*** **permits an input signal to pass through to an output**
- ***Disabling*** **blocks an input signal from passing through to an output, replacing it with a fixed value**
- **The value on the output when it is disable can be Hi-Z (as for three-state buffers and transmission gates), 0 , or 1**
- **When disabled, 0 output**
- **When disabled, 1 output**
- **See Enabling App in text**



(a)



(b)

Decoding

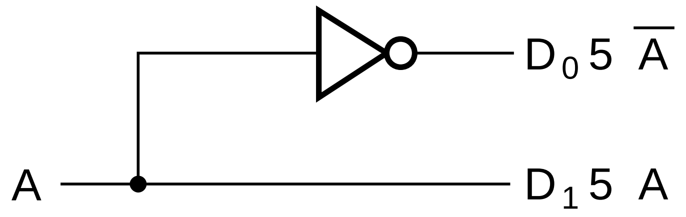
- **Decoding - the conversion of an *n*-bit input code to an *m*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code**
- **Circuits that perform decoding are called *decoders***
- **Here, functional blocks for decoding are**
  - **called *n*-to-*m* line decoders, where $m \leq 2^n$, and**
  - **generate 2^n (or fewer) minterms for the *n* input variables**

Decoder Examples

- 1-to-2-Line Decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

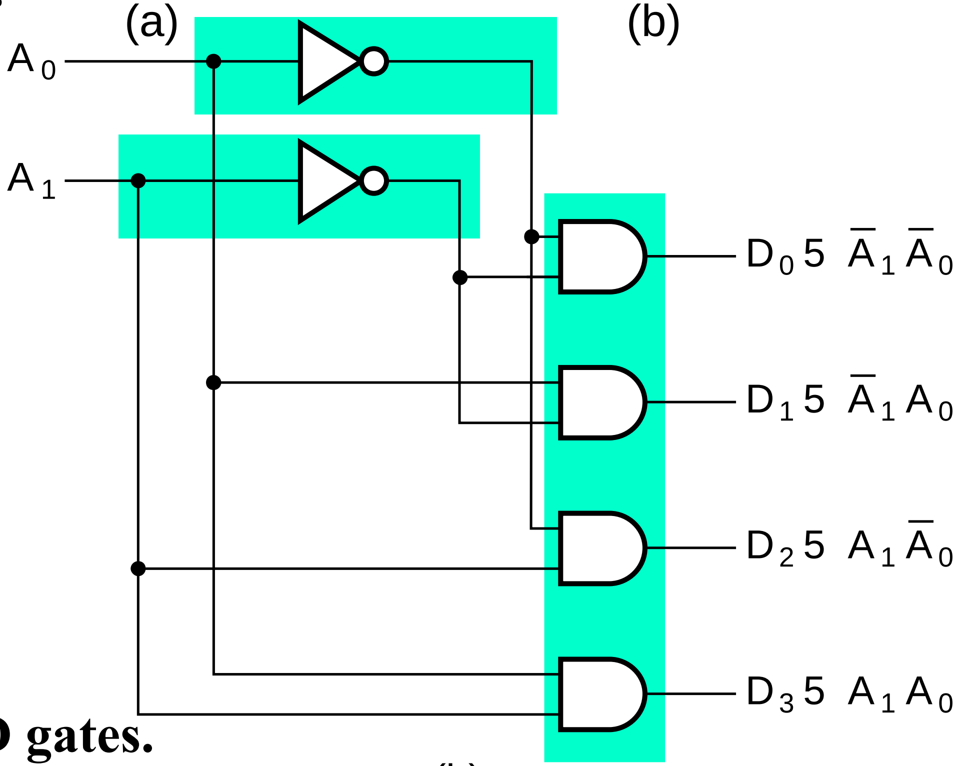


(b)

- 2-to-4-Line Decoder

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



A_0
A_1
D_0 5 $\overline{A}_1 \overline{A}_0$
D_1 5 $\overline{A}_1 A_0$
D_2 5 $A_1 \overline{A}_0$
D_3 5 $A_1 A_0$

(b)

- Note that the 2-4-line made up of 2 1-to-2-line decoders and 4 AND gates.

Decoder Expansion

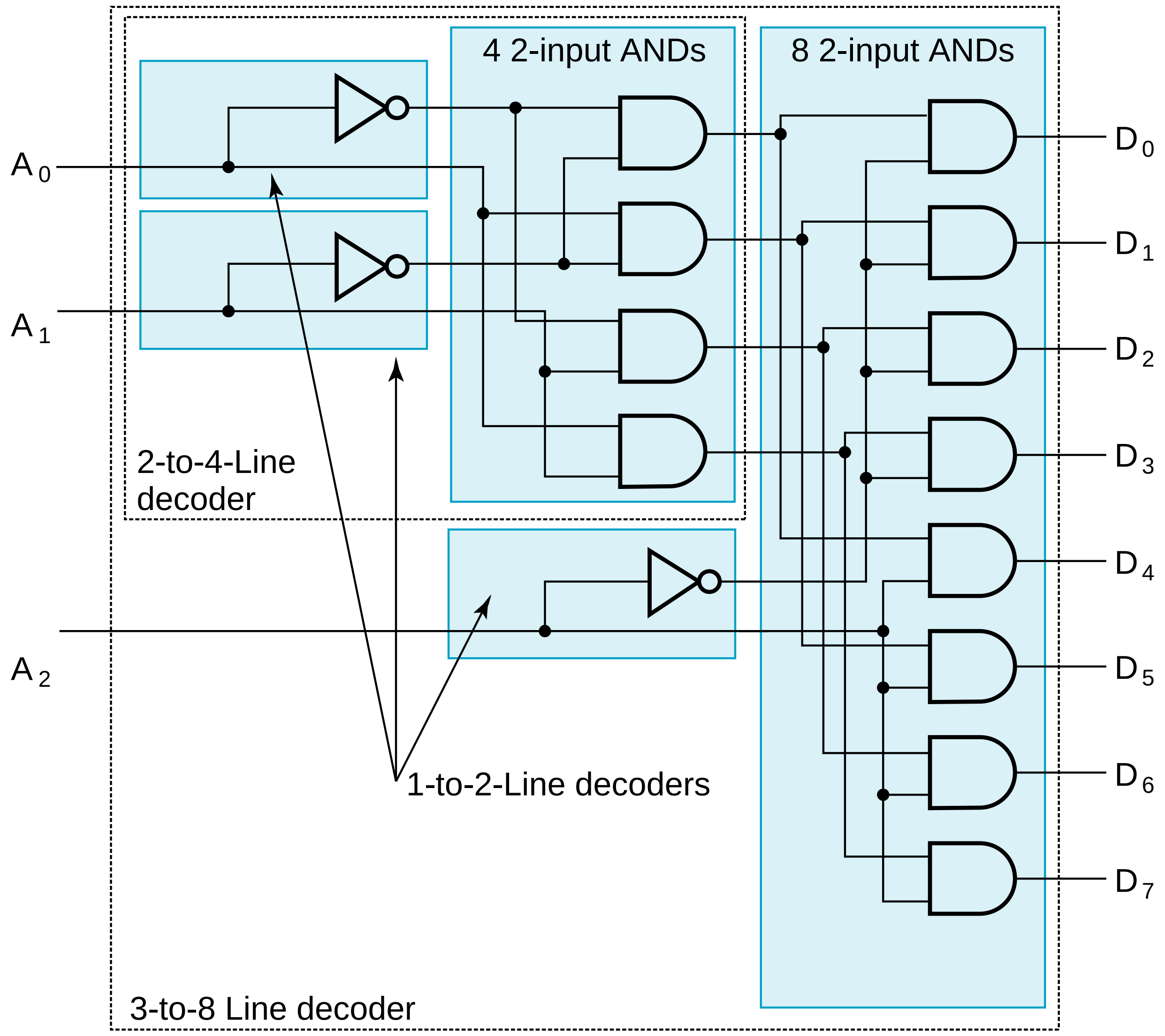
- **General procedure given in book for any decoder with *n* inputs and 2^n outputs.**
- **This procedure builds a decoder backward from the outputs.**
- **The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.**
- **These decoders are then designed using the same procedure until 2-to-1-line decoders are reached.**
- **The procedure can be modified to apply to decoders with the number of outputs $\neq 2^n$**

Decoder Expansion - Example 1

- 3-to-8-line decoder
  - Number of output ANDs = 8
  - Number of inputs to decoders driving output ANDs = 3
  - Closest possible split to equal
    - 2-to-4-line decoder
    - 1-to-2-line decoder
  - 2-to-4-line decoder
    - Number of output ANDs = 4
    - Number of inputs to decoders driving output ANDs = 2
    - Closest possible split to equal
      - Two 1-to-2-line decoders
- See next slide for result

Decoder Expansion - Example 1

- **Result**



Decoder Expansion - Example 2

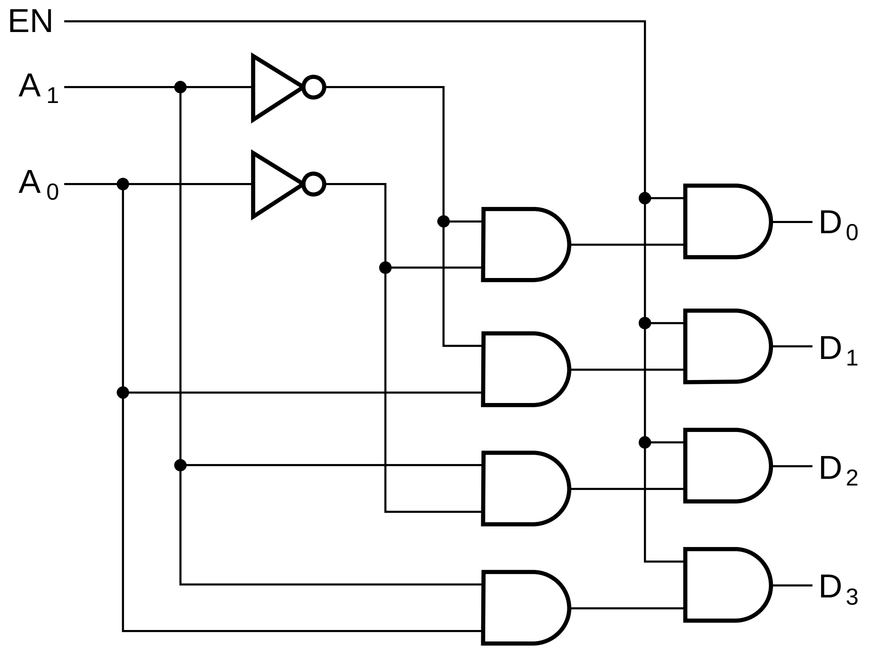
- 7-to-128-line decoder
  - Number of output ANDs = 128
  - Number of inputs to decoders driving output ANDs = 7
  - Closest possible split to equal
    - 4-to-16-line decoder
    - 3-to-8-line decoder
  - 4-to-16-line decoder
    - Number of output ANDs = 16
    - Number of inputs to decoders driving output ANDs = 2
    - Closest possible split to equal
      - 2 2-to-4-line decoders
  - Complete using known 3-8 and 2-to-4 line decoders

Decoder with Enable

- **In general, attach *m*-enabling circuits to the outputs**
- **See truth table below for function**
  - **Note use of X's to denote both 0 and 1**
  - **Combination containing two X's represent four binary combinations**
- **Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs**
- **In this case, called a *demultiplexer***

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

Encoding

- **Encoding - the opposite of decoding - the conversion of an *m*-bit input code to a *n*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code**
- **Circuits that perform encoding are called *encoders***
- **An encoder has 2^n (or fewer) input lines and *n* output lines which generate the binary code corresponding to the input values**
- **Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corres-ponding to the position in which the 1 appears.**

Encoder Example

- A decimal-to-BCD encoder
  - Inputs: 10 bits corresponding to decimal digits 0 through 9, (D_0, …, D_9)
  - Outputs: 4 bits with BCD codes
  - Function: If input bit D_i is a 1, then the output (A_3, A_2, A_1, A_0) is the BCD code for i,
- The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly.

Encoder Example (continued)

- Input D_i is a term in equation A_j if bit A_j is 1 in the binary value for i.
- Equations:

$$A_3 = D_8 + D_9$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

- $F_1 = D_6 + D_7$ can be extracted from A_2 and A_1 Is there any cost saving?

Priority Encoder

- **If more than one input value is 1, then the encoder just designed does not work.**
- **One encoder that can accept all possible combinations of input values and produce a meaningful result is a *priority encoder*.**
- **Among the 1s that appear, it selects the most significant input position (or the least significant input position) containing a 1 and responds with the corresponding binary code for that position.**

Priority Encoder Example

- Priority encoder with 5 inputs (D_4, D_3, D_2, D_1, D_0) - highest priority to most significant 1 present - Code outputs A2, A1, A0 and V where V indicates at least one 1 present.

No. of Min-terms/Row	Inputs					Outputs			
	D4	D3	D2	D1	D0	A2	A1	A0	V
1	0	0	0	0	0	X	X	X	0
1	0	0	0	0	1	0	0	0	1
2	0	0	0	1	X	0	0	1	1
4	0	0	1	X	X	0	1	0	1
8	0	1	X	X	X	0	1	1	1
16	1	X	X	X	X	1	0	0	1

- Xs in input part of table represent 0 or 1; thus table entries correspond to product terms instead of minterms. The column on the left shows that all 32 minterms are present in the product terms in the table

Priority Encoder Example (continued)

- **Could use a K-map to get equations, but can be read directly from table and manually optimized if careful:**

$$A_2 = D_4$$

$$A_1 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 D_2 = \overline{D}_4 F_1, \ F_1 = (D_3 + D_2)$$

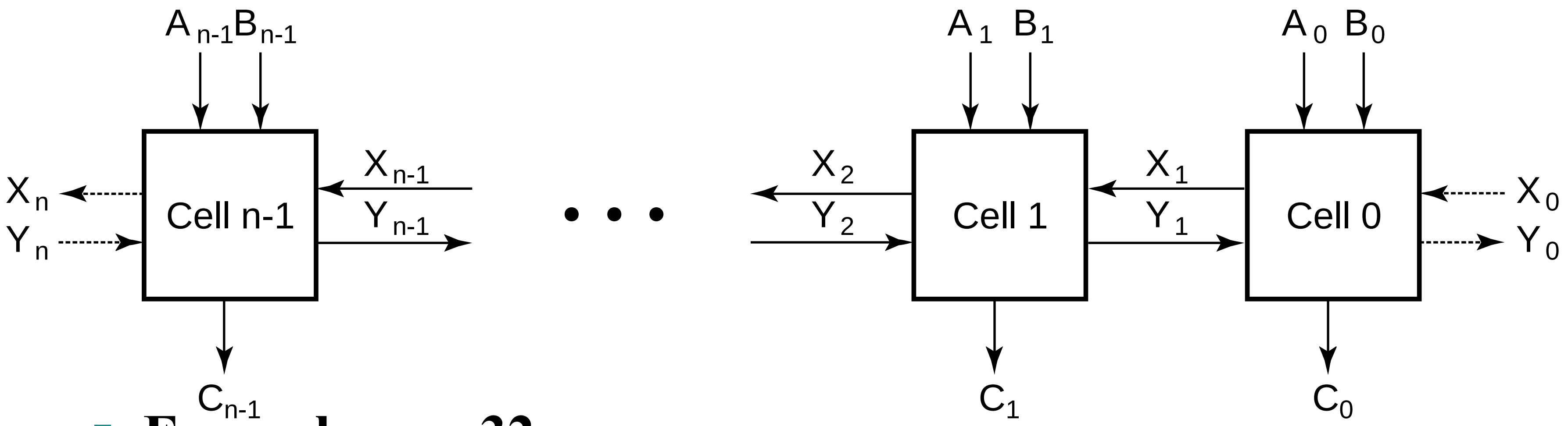
$$A_0 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 \overline{D}_2 D_1 = \overline{D}_4 (D_3 + \overline{D}_2 D1)$$

$$V = D_4 + F_1 + D_1 + D_0$$

Iterative Combinational Circuits

- **Arithmetic functions**
  - **Operate on binary vectors**
  - **Use the same subfunction in each bit position**
- **Can design functional block for subfunction and repeat to obtain functional block for overall function**
- ***Cell* - subfunction block**
- ***Iterative array* - a array of interconnected cells**
- **An iterative array can be in a <u>single</u> dimension (1D) or <u>multiple</u> dimensions**

Block Diagram of a 1D Iterative Array



- **Example: n = 32**
  - **Number of inputs = ?**
  - **Truth table rows = ?**
  - **Equations with up to ? input variables**
  - **Equations with huge number of terms**
  - **Design impractical!**
- **Iterative array takes advantage of the regularity to make design feasible**

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 binary addition, 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:

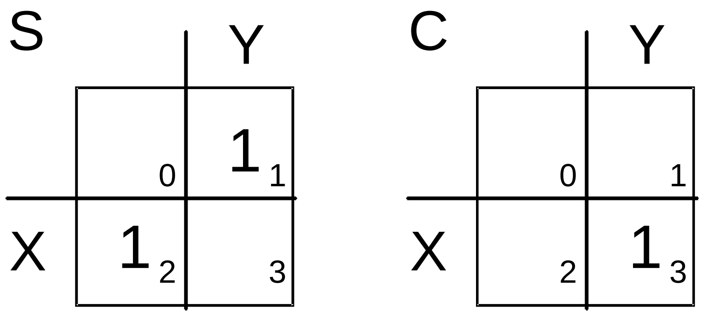
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a <u>sum bit</u> , S and a <u>carry bit</u>, C
- The half adder can be specified as a truth table for S and C $\Rightarrow$

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**

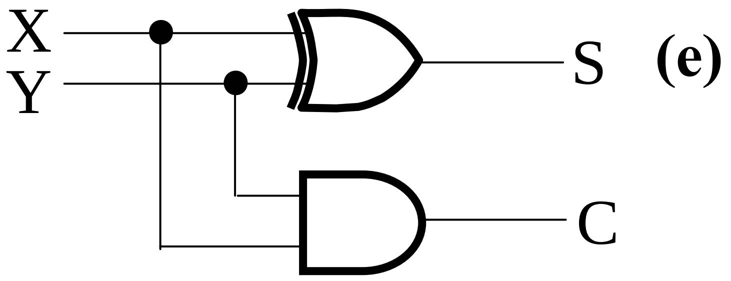
$$C = X \cdot Y$$

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

Implementations: Half-Adder

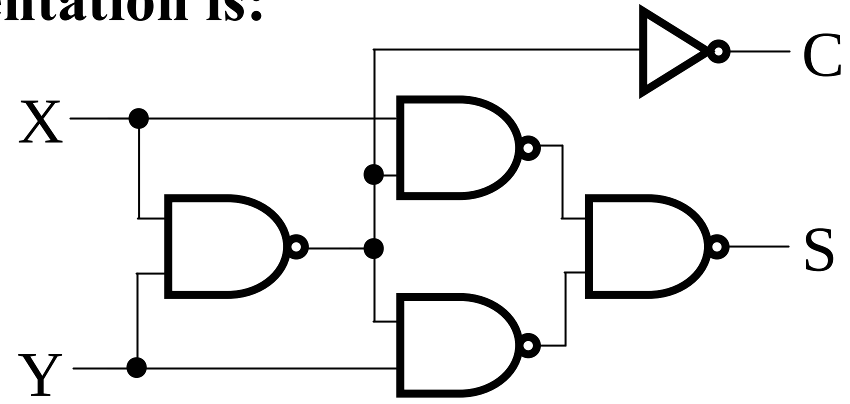
- **The most common half adder implementation is:**

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



- **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:

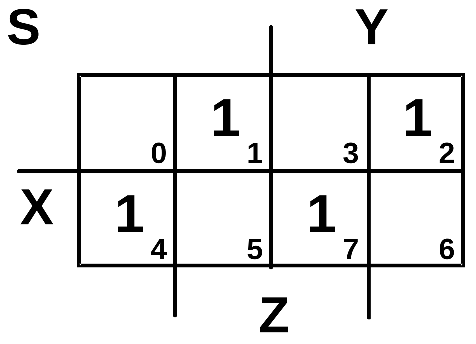
Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

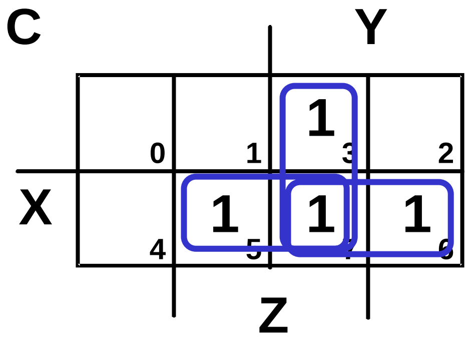
Logic Optimization: Full-Adder

- **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:**

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

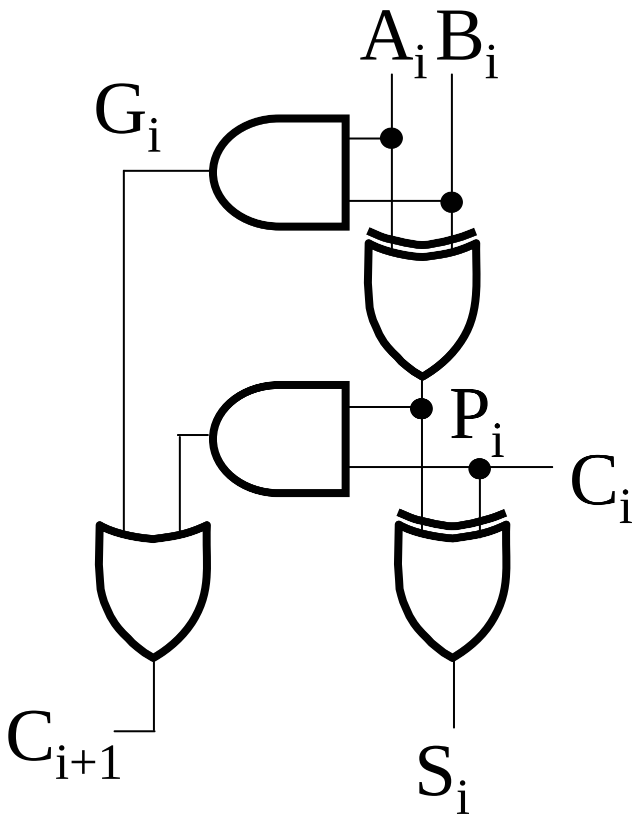
- **The term X·Y is *carry generate*.**
- **The term X⊕Y is *carry propagate*.**

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)

$Co = G + P \cdot Ci$

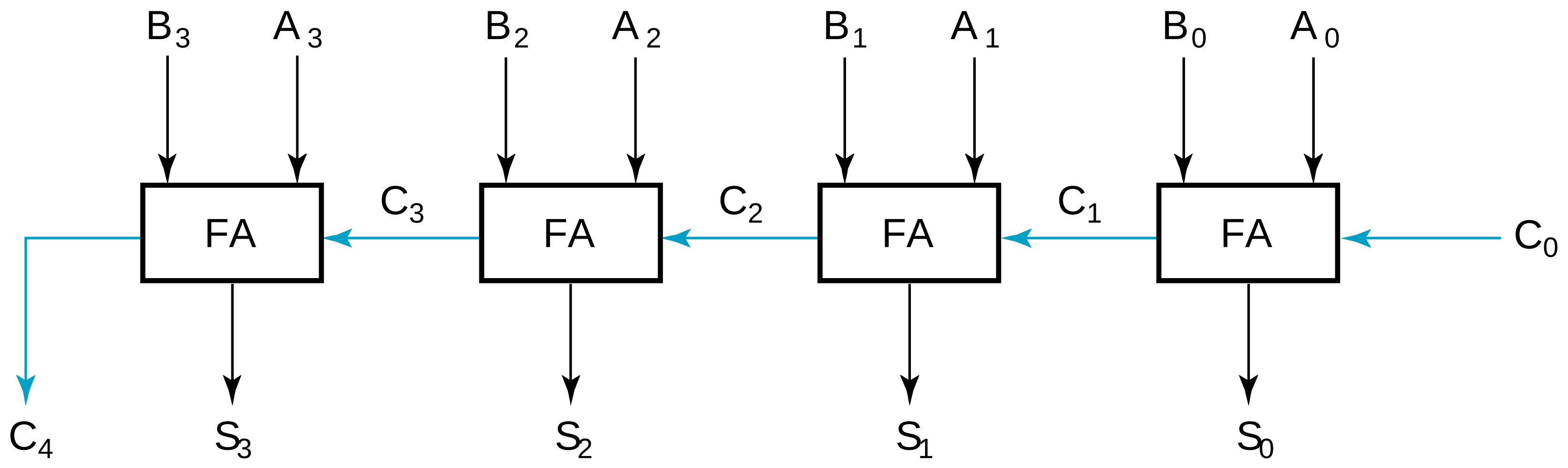
Binary Adders

- 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	$\underline{0\ 0\ 1\ 1}$	B_i
Sum	1 1 1 0	S_i
Carry out	0 0 1 1	C_{i+1}

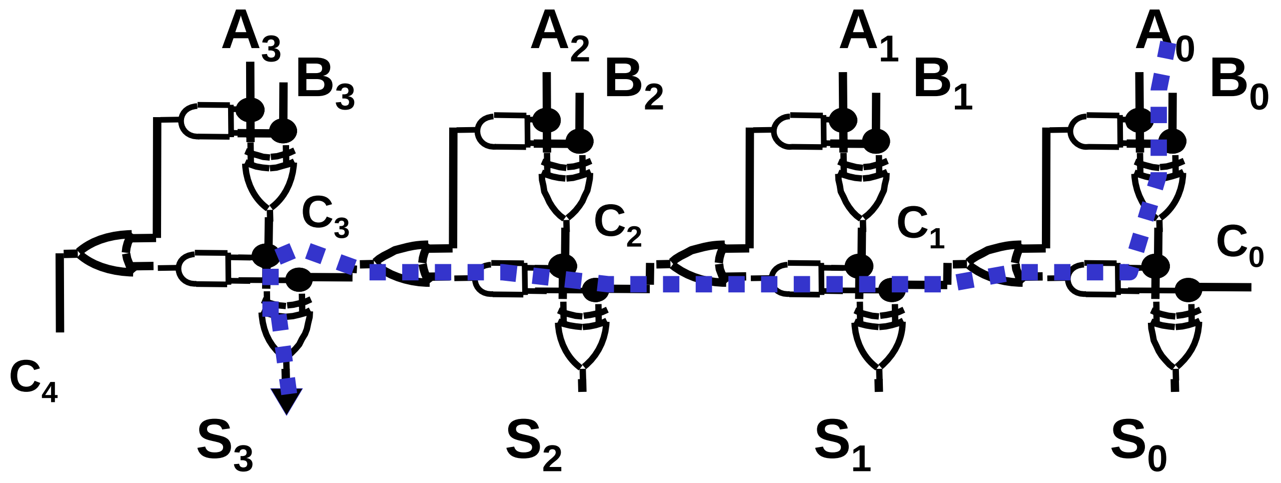
4-bit Ripple-Carry Binary Adder

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



Carry Propagation & Delay

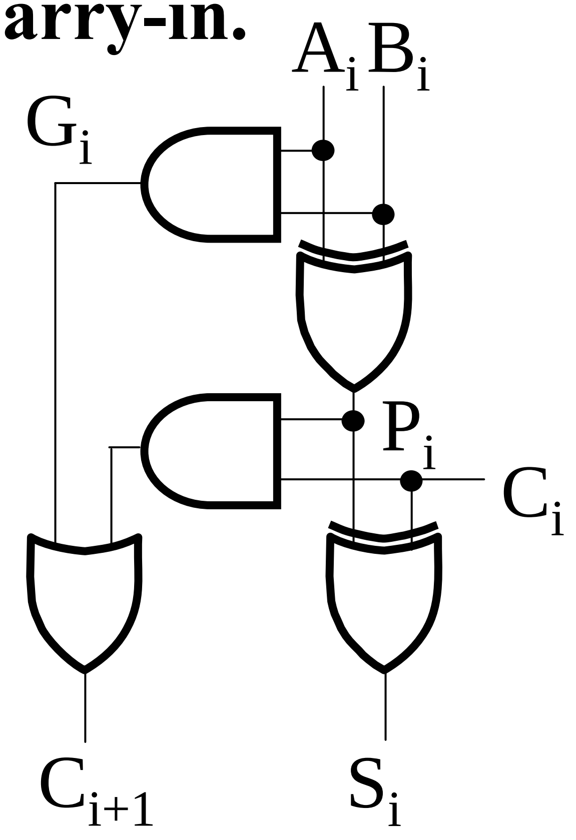
- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:



- 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 <u>carry generated</u> when $A_i = B_i =$ "1", whether or not there is a carry-in.**
- **Alternately, there will be a <u>carry propagated</u> 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:**

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

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

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_{0-3}) and *group propagate* (P_{0-3}) functions:

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

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

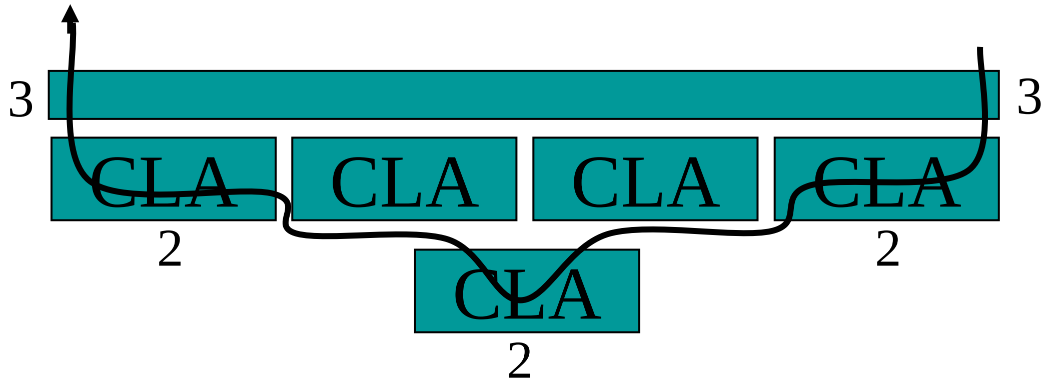
- Using these two equations:

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

- 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**

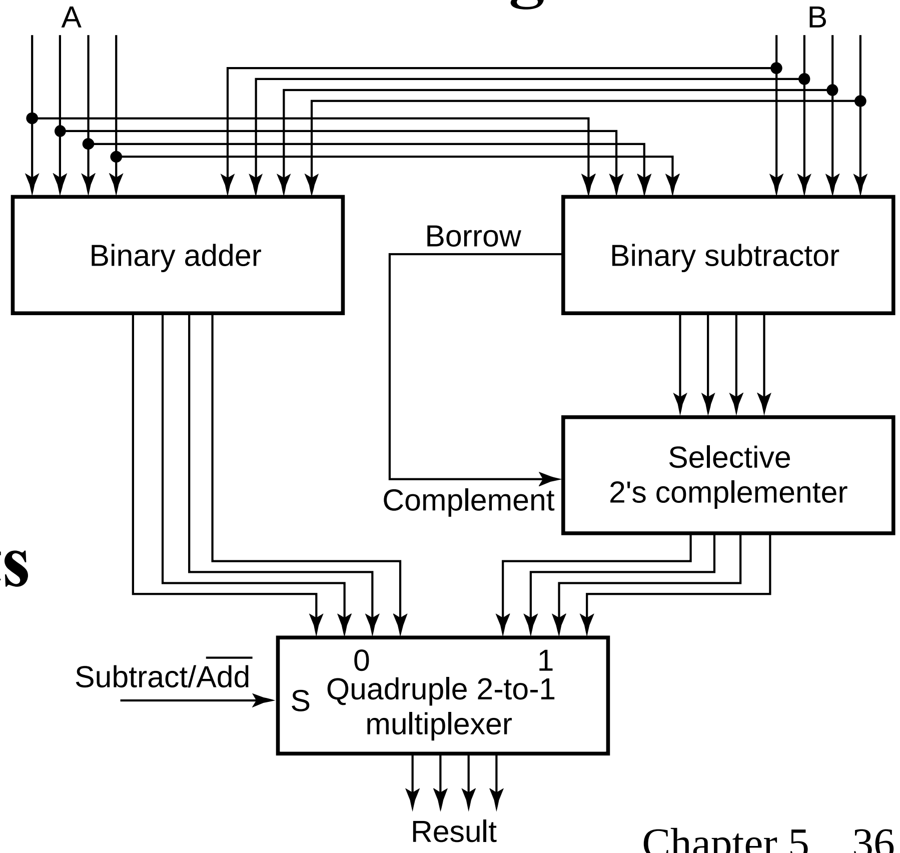
Unsigned Subtraction

- **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:**

```
     0             1
      1001          0100
    – 0111        – 0111
      ----          ----
      0010          1101
                   10000
                  – 1101
                   -----
               (–)  0011
```

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, unsigned 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

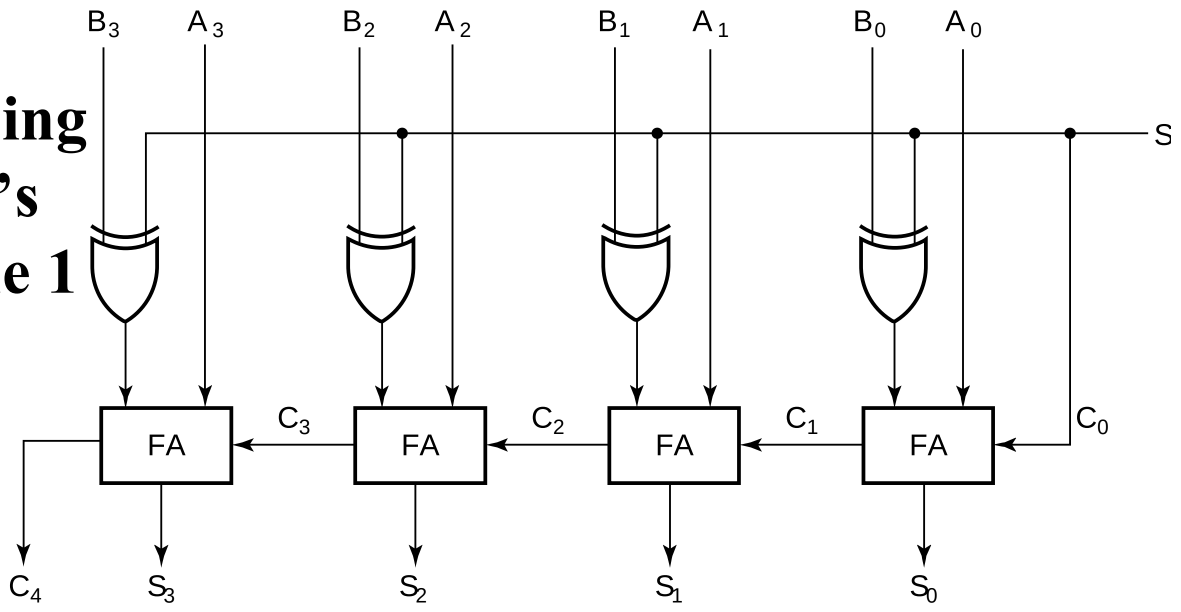
- Find $01010100_2 - 01000011_2$

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

- 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

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

- 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:**

$$\begin{array}{l} 0\quad 0\;0\quad 0\;1\quad 1\;1\quad 1 \end{array}$$

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

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

$$\begin{array}{l} 0\quad 1\;0\quad 1\;1\quad 0\;1\quad 0 \end{array}$$

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

- **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* × *m* digit multiply generates up to an *m* + *n* digit result.**

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

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).**

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

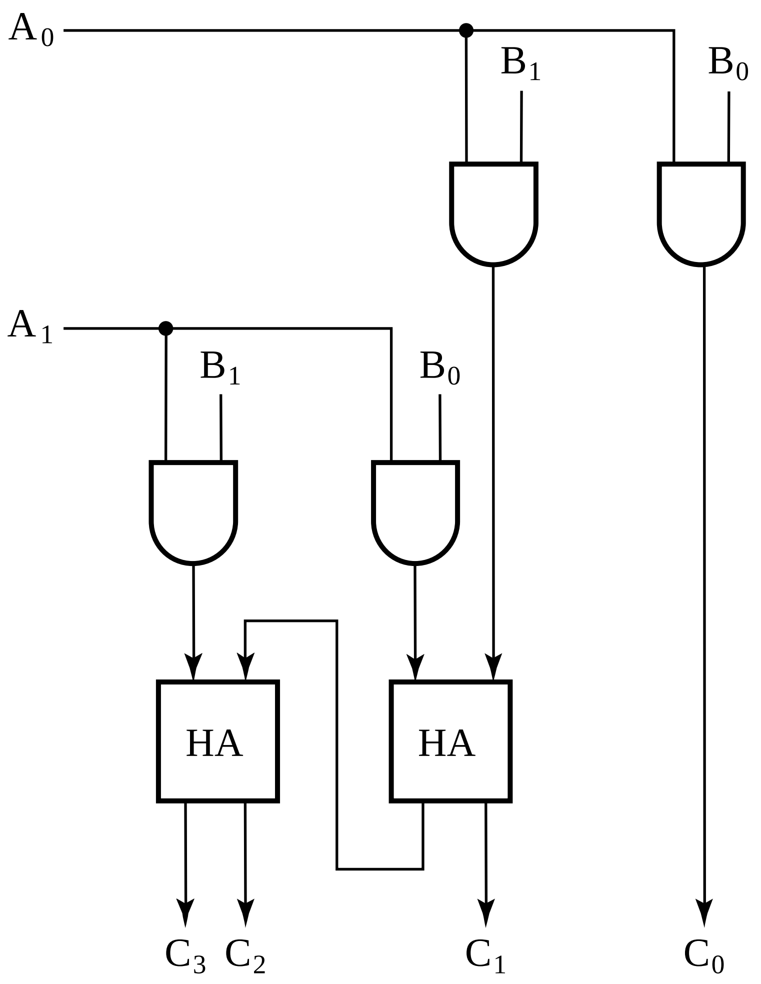
Multiplier Boolean Equations

- We can also make an $n \times m$ “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.

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

Multiplier Arrays Using Adders

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

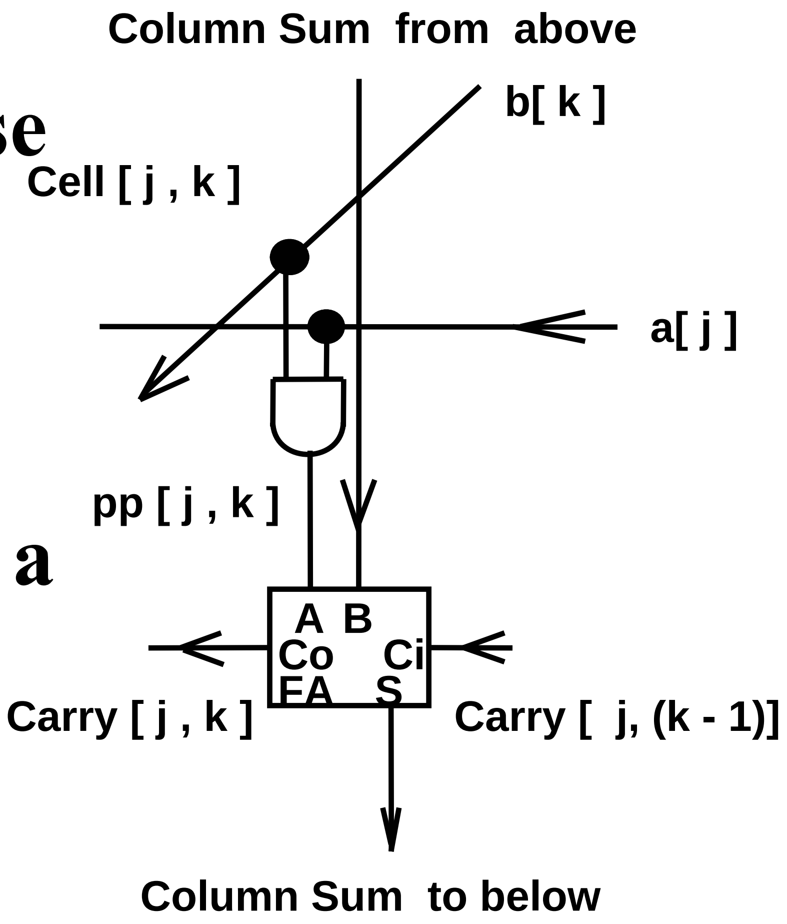


Multiplier Using Wide Adders

- A more "structured" way to develop an $n \times m$ multiplier is to sum partial products using adder trees
- The partial products are formed using an $n \times m$ array of AND gates
- Partial products are summed using m – 1 adders of width *n* bits
- Example: 4-bit by 3-bit adder
- Text figure 5-11 shows a 4 × 3 = 12 element array of AND gates and two 4-bit adders

Cellular Multiplier Array

- Another way to implement multipliers is to use an $n \times m$ cellular array structure of uniform elements as shown:
- Each element computes a single bit product equal to $a_i \cdot b_j$, and implements a single bit full adder



Other Arithmetic Functions

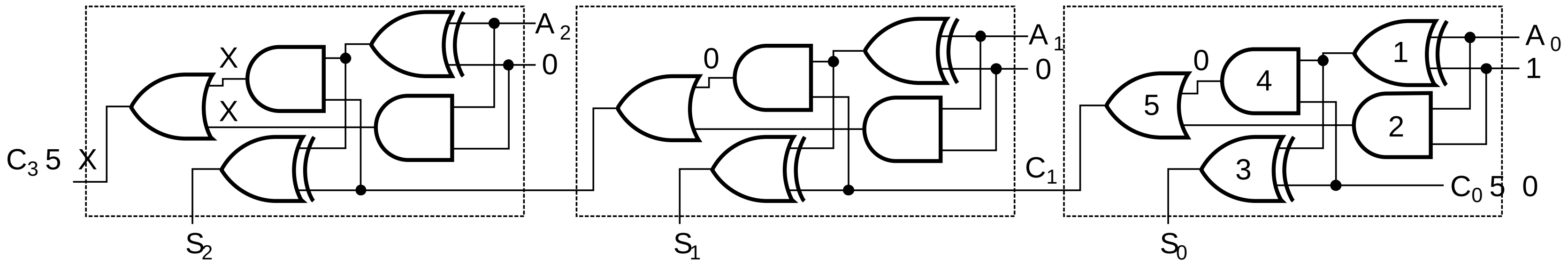
- **Convenient to design the functional blocks by *contraction* - removal of redundancy from circuit to which input fixing has been applied**
- **Functions**
  - **Incrementing**
  - **Decrementing**
  - **Multiplication by Constant**
  - **Division by Constant**
  - **Zero Fill and Extension**

Design by Contraction

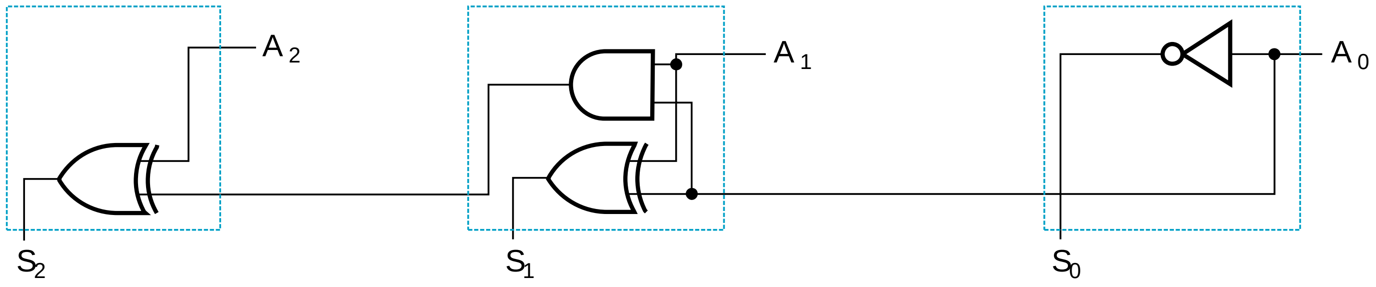
- Contraction is a technique for simplifying the logic in a functional block to implement a different function
  - The new function must be realizable from the original function by applying rudimentary functions to its inputs
  - Contraction is treated here only for application of 0s and 1s (not for X and $\overline{X}$)
  - After application of 0s and 1s, equations or the logic diagram are simplified by using rules given on pages 224 - 225 of the text.

Design by Contraction Example

- **Contraction of a ripple carry adder to incrementer for *n* = 3**
  - **Set B = 001**



(a)



(b)

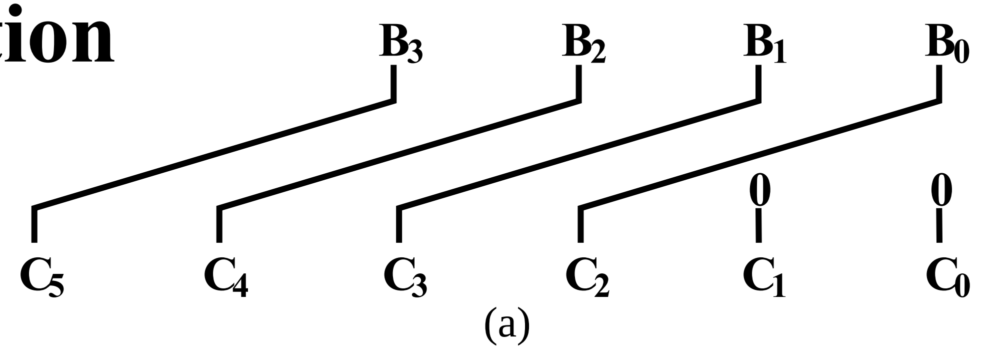
  - **The middle cell can be repeated to make an incrementer with *n* > 3.**

Incrementing & Decrementing

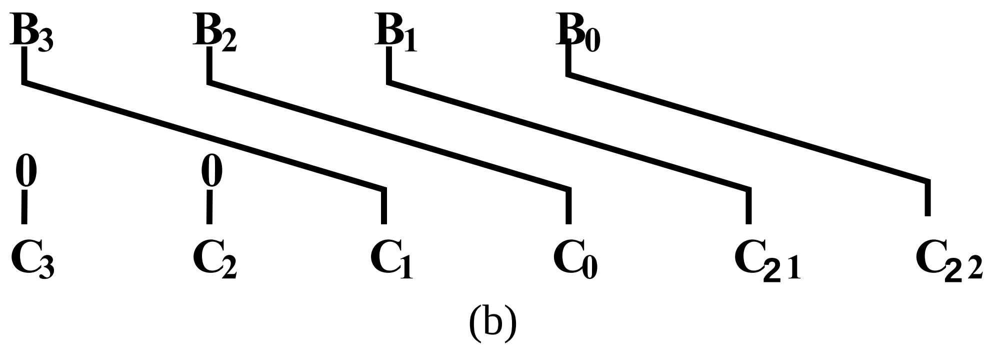
- ***Incrementing***
  - **Adding a fixed value to an arithmetic variable**
  - **Fixed value is often 1, called *counting (up*)**
  - **Examples: A + 1, B + 4**
  - **Functional block is called *incrementer***
- ***Decrementing***
  - **Subtracting a fixed value from an arithmetic variable**
  - **Fixed value is often 1, called *counting (down*)**
  - **Examples: A – 1, B – 4**
  - **Functional block is called *decrementer***

Multiplication/Division by 2^n

- **(a) Multiplication by 100**
  - **Shift left by 2**

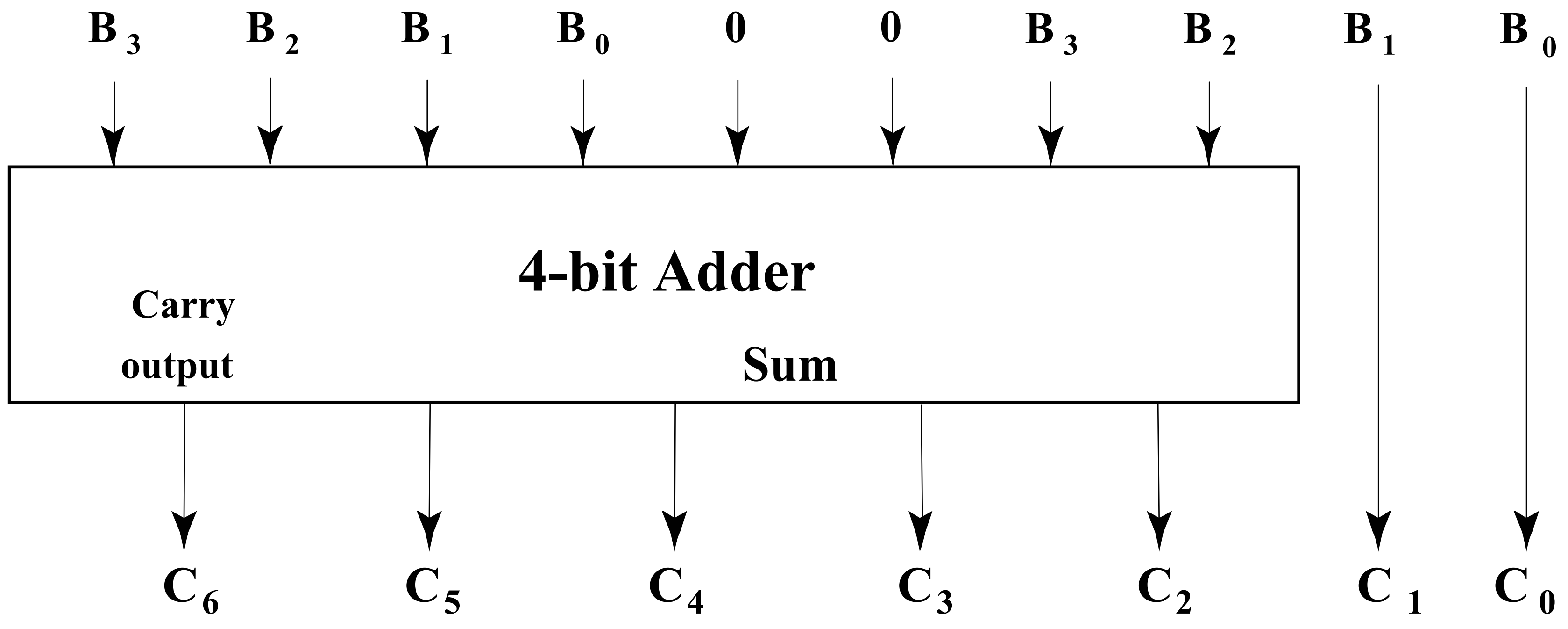


- **(b) Division by 100**
  - **Shift right by 2**
  - **Remainder preserved**



Multiplication by a Constant

- Multiplication of B(3:0) by 101
- See text Figure 513 (a) for contraction



Zero Fill

- ***Zero fill*** **- filling an *m*-bit operand with 0s to become an *n*-bit operand with *n* > *m***
- **Filling usually is applied to the MSB end of the operand, but can also be done on the LSB end**
- **Example: 11110101 filled to 16 bits**
  - **MSB end: 0000000011110101**
  - **LSB end: 1111010100000000**

Extension

- ***Extension*** **- increase in the number of bits at the MSB end of an operand by using a complement representation**
  - **Copies the MSB of the operand into the new positions**
  - **Positive operand example - 01110101 extended to 16 bits:**

    0000000001110101
  - **Negative operand example - 11110101 extended to 16 bits:**

    1111111111110101

Terms of Use

- Return to Title Page