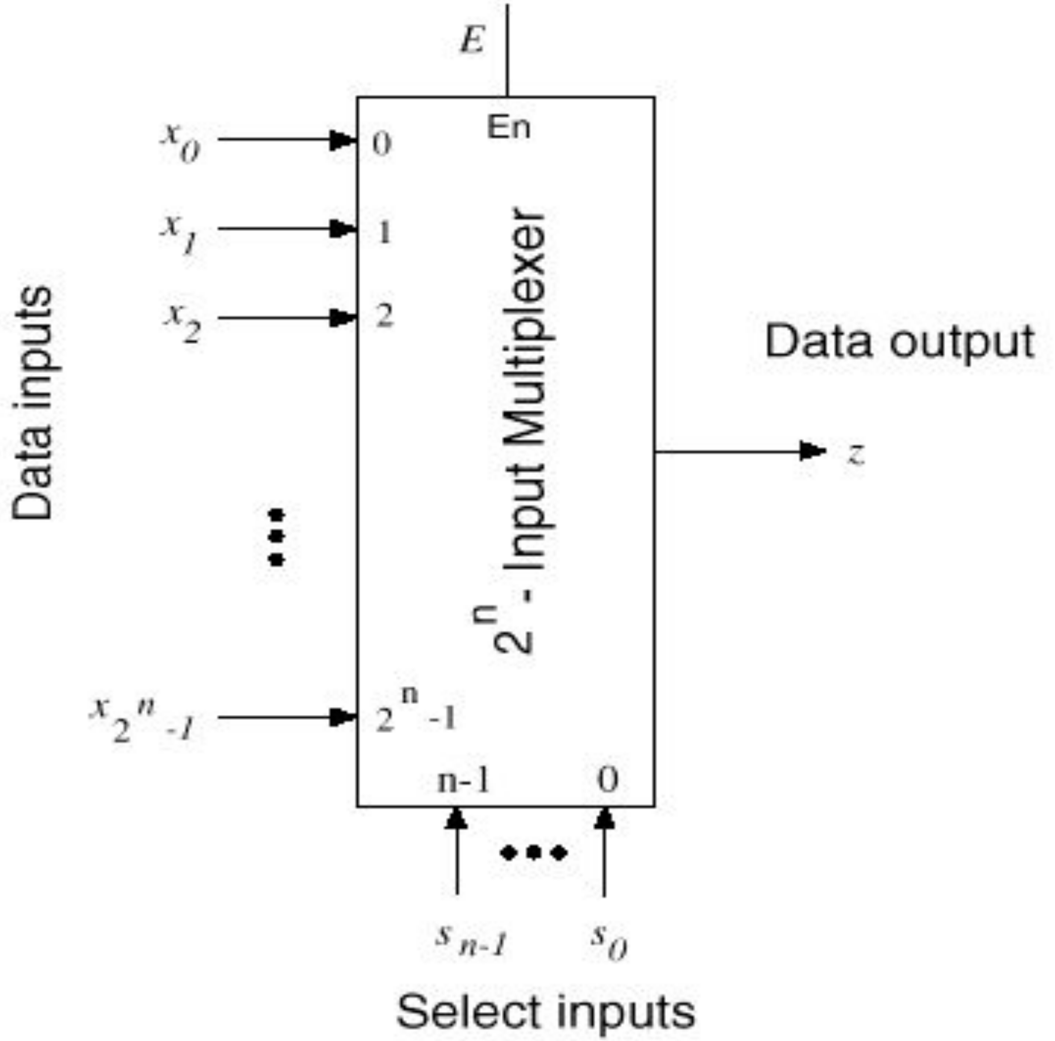
# 18ECC206J- VLSI DESIGN
# UNIT II

*

# MULTIPLEXER

- "Selects" binary information from one of many input lines and directs it to a single output line.
- Also know as the "selector" circuit,
- Selection is controlled by a particular set of inputs lines whose no. depends on the no. of the data input lines.
- For a $2^n$-to-1 multiplexer, there are $2^n$ data input lines and *n* selection lines whose bit combination determines which input is selected.

# Multiplexer (cont.)



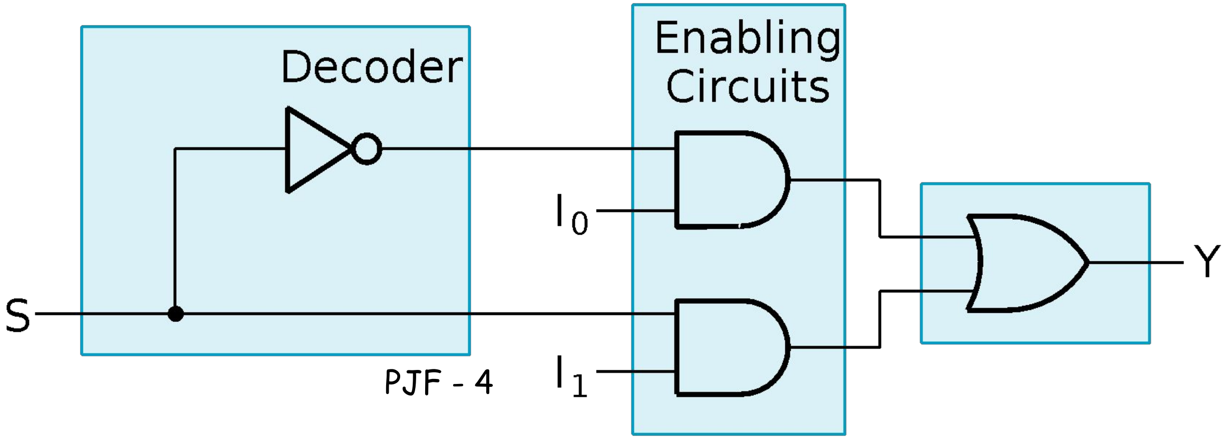
*

# 2-to-1 Multiplexer

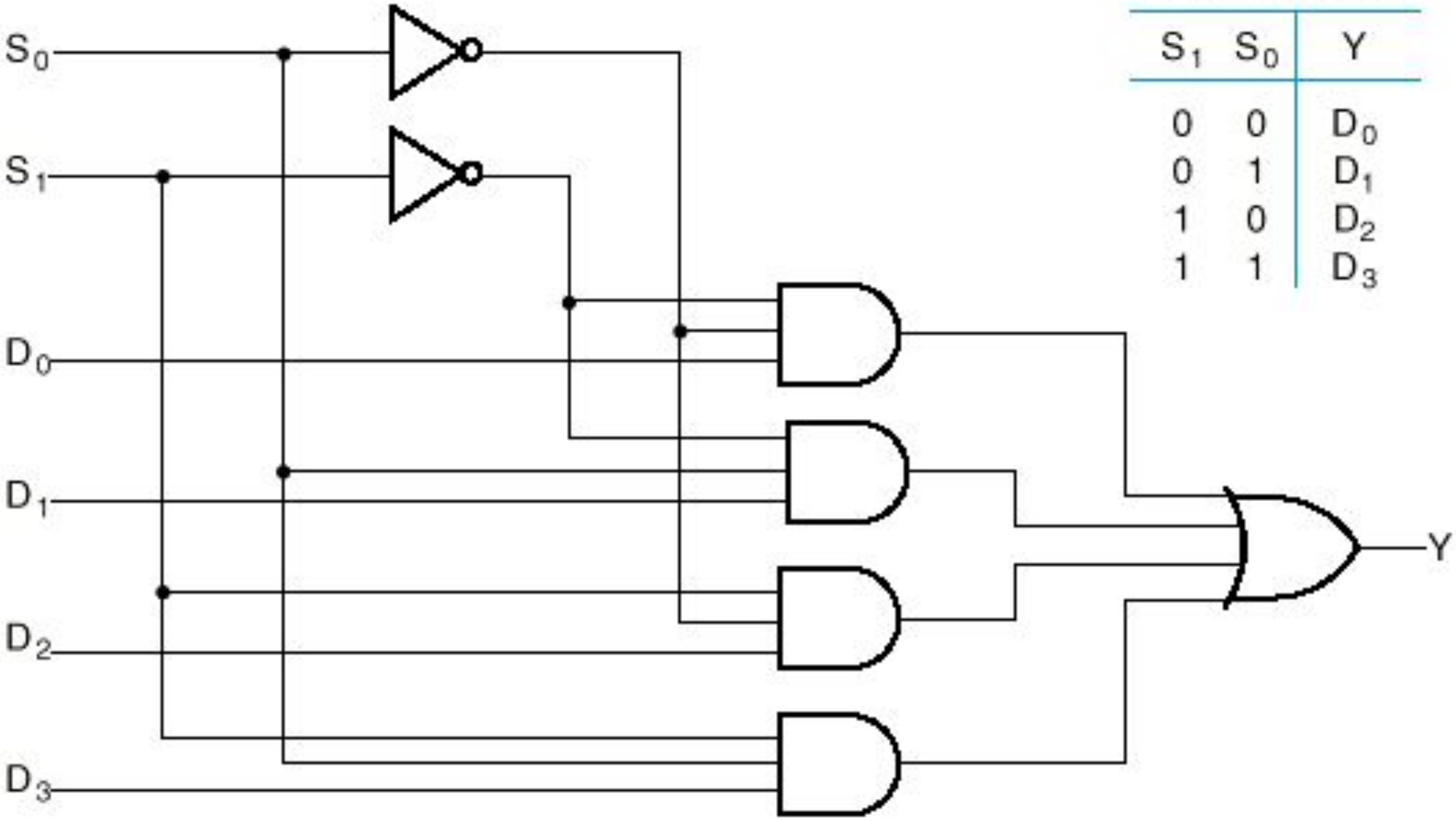
- Since $2 = 2^1$, $n = 1$
- The single selection variable S has two values:
  - $S = 0$ selects input $I_0$
  - $S = 1$ selects input $I_1$
- The equation:

$$Y = S' I_0 + S I_1$$

- The circuit:



# 4-to-1 MUX



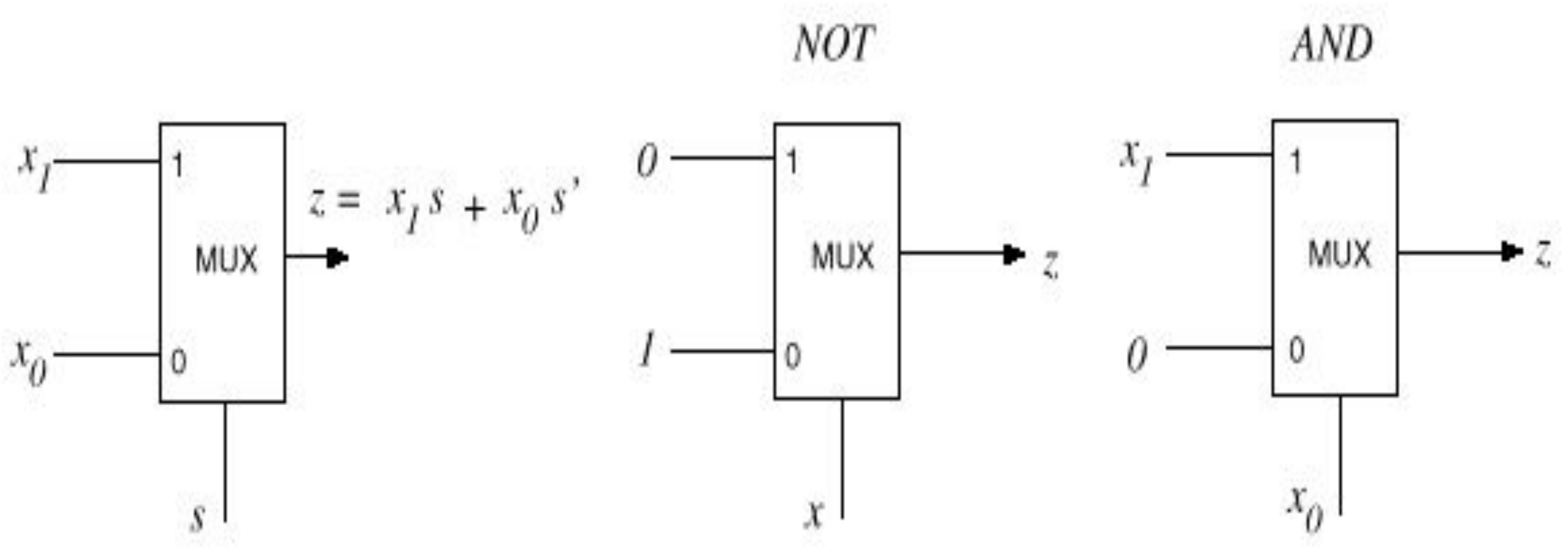
Function table

| $S_1$ | $S_0$ | Y |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

*

# MUX as a Universal Gate

- All basic gates can be deisgned using 2-to-1 MUXs. Thus, 2-to-1 MUX is a universal gate.



$z = 0x + 1x' = x'$

$z = x_1x_0 + 0x_0' = x_1x_0$

# Implementing Boolean functions with Multiplexers

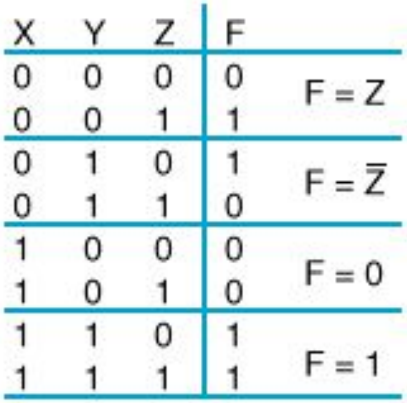
- Any Boolean function of *n* variables can be implemented using a $2^{n-1}$-to-1 multiplexer.
- The SELECT signals generate the minterms of the function.
- The data inputs identify which minterms are to be combined with an OR.
- Example

Implement $F(X,Y,Z) = \Sigma m(1,2,6,7)$ using Multiplexers

$$\Sigma m(1,2,6,7 \quad = X'Y'Z + X'YZ' + XYZ' + XYZ$$

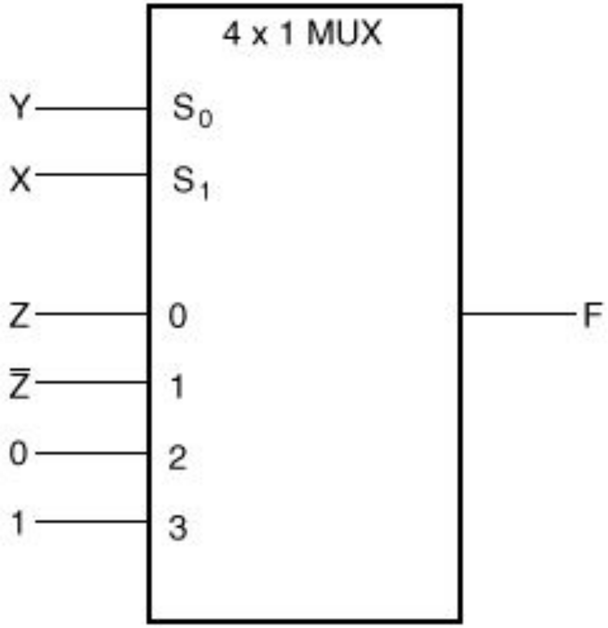
- X’Y’Z + X’YZ’ + XYZ’ + XYZ
- There are n=3 inputs, thus we need a 4-to-1 MUX

- The first n-1 (=2) inputs serve as the selection lines



| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | F = Z |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | F = $\overline{Z}$ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | F = 0 |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | F = 1 |
| 1 | 1 | 1 | 1 | |

(a) Truth table



(b) Multiplexer implementation

*

# Efficient Method for implementing Boolean functions

- For an *n*-variable function (*e.g.*, f(A,B,C,D)):
  - Need a $2^{n-1}$ line MUX with *n*-1 select lines.
  - Enumerate function as a truth table with consistent ordering of variables (*e.g.*, A,B,C,D)
  - Attach the most significant *n*-1 variables to the *n*-1 select lines (*e.g.*, A,B,C)
  - Examine pairs of adjacent rows (only the least significant variable differs, *e.g.*, D=0 and D=1).
  - Determine whether the function output for the (A,B,C,0) and (A,B,C,1) combination is (0,0), (0,1), (1,0), or (1,1).
  - Attach 0, D, D’, or 1 to the data input corresponding to (A,B,C) respectively.

# Example

- Consider F(A,B,C) = ∑m(1,3,5,6).
- We can implement this function using a 4-to-1 MUX as follows.
- The inputs are A,B,C. Apply A and B to the $S_1$ and $S_0$ selection inputs of the MUX
- Enumerate function in a truth table.

When A=B=0, F=C

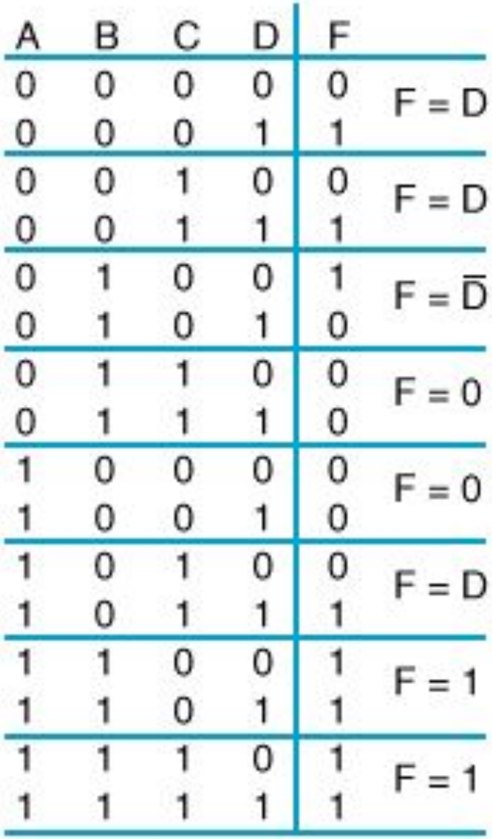
When A=0, B=1, F=C

When A=1, B=0, F=C

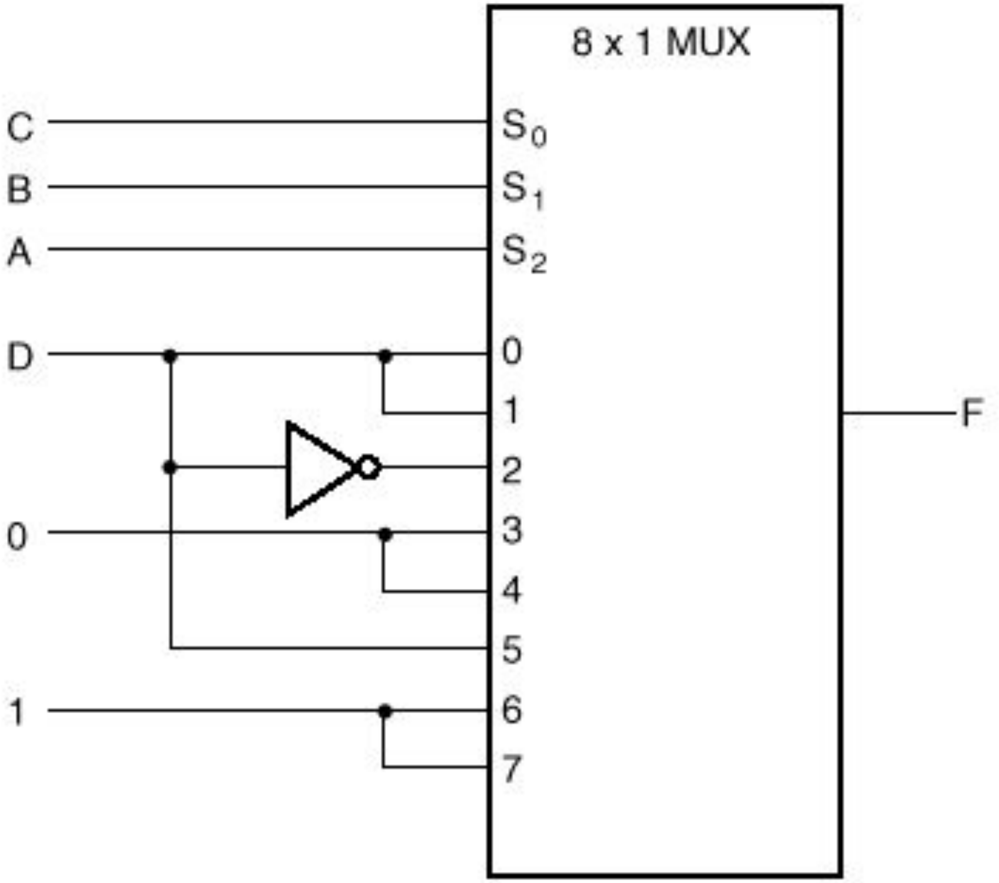
When A=B=1, F=C'

| A | B | C |  | F |
|---|---|---|---|---|
| 0 | 0 | 0 |  | 0 |
| 0 | 0 | 1 |  | 1 |
| 0 | 1 | 0 |  | 0 |
| 0 | 1 | 1 |  | 1 |
| 1 | 0 | 0 |  | 0 |
| 1 | 0 | 1 |  | 1 |
| 1 | 1 | 0 |  | 1 |
| 1 | 1 | 1 |  | 0 |

# A larger Example



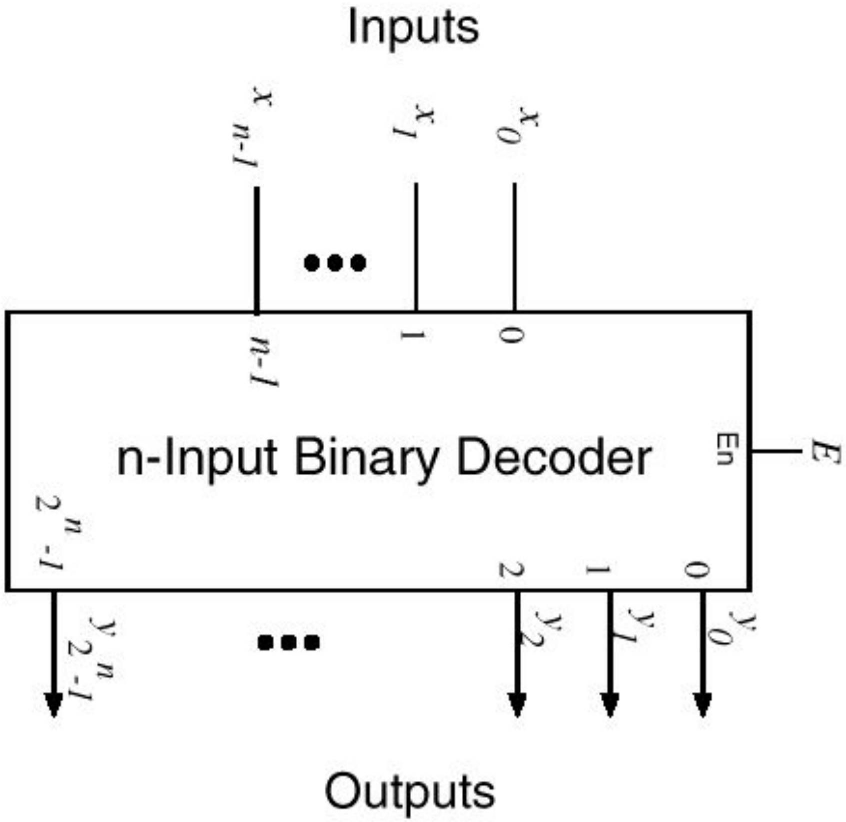
| A | B | C | D | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | F = D |
| 0 | 0 | 0 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | F = D |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | F = $\overline{D}$ |
| 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 0 | F = 0 |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 0 | F = 0 |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 0 | 0 | F = D |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 1 | F = 1 |
| 1 | 1 | 0 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 1 | F = 1 |
| 1 | 1 | 1 | 1 | 1 | |



# DECODERS

- A combinational circuit that converts binary information from *n* coded inputs to a maximum $2^n$ coded outputs
  *n-to-m* decoder, $m \leq 2^n$

  - □ *n-to-* $2^n$ decoder

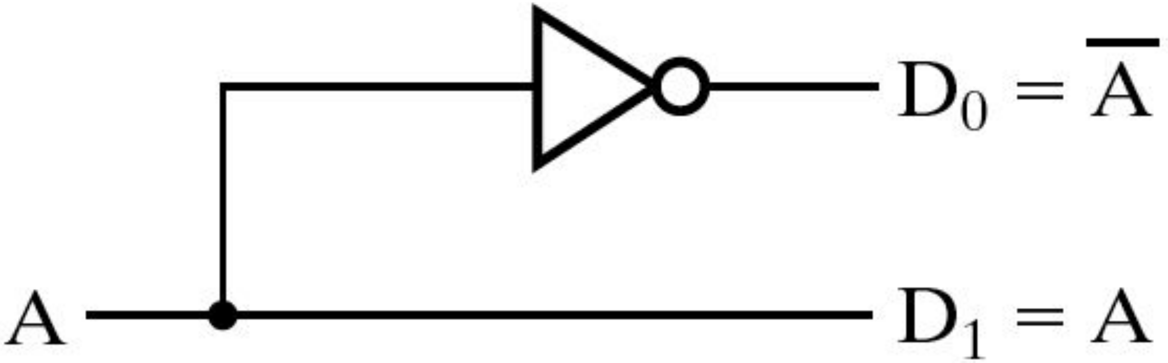


*

# 1-2 Decoder

| A | $D_0$ | $D_1$ |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |

(a)

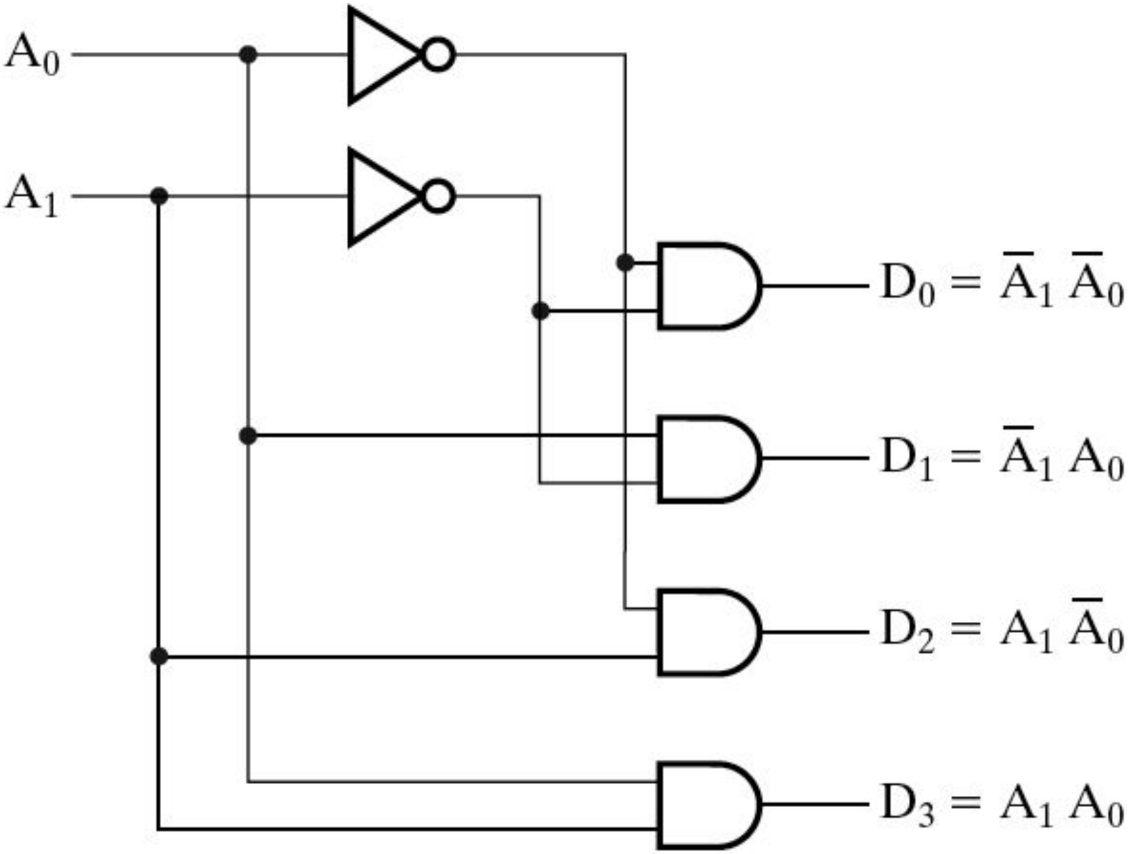


(b)

# 2-to-4 Decoder

| $\mathbf{A_1}$ | $\mathbf{A_0}$ | $\mathbf{D_0}$ | $\mathbf{D_1}$ | $\mathbf{D_2}$ | $\mathbf{D_3}$ |
|---|---|---|---|---|---|
| 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)



(b)

# 2-to-4 Active Low Decoder

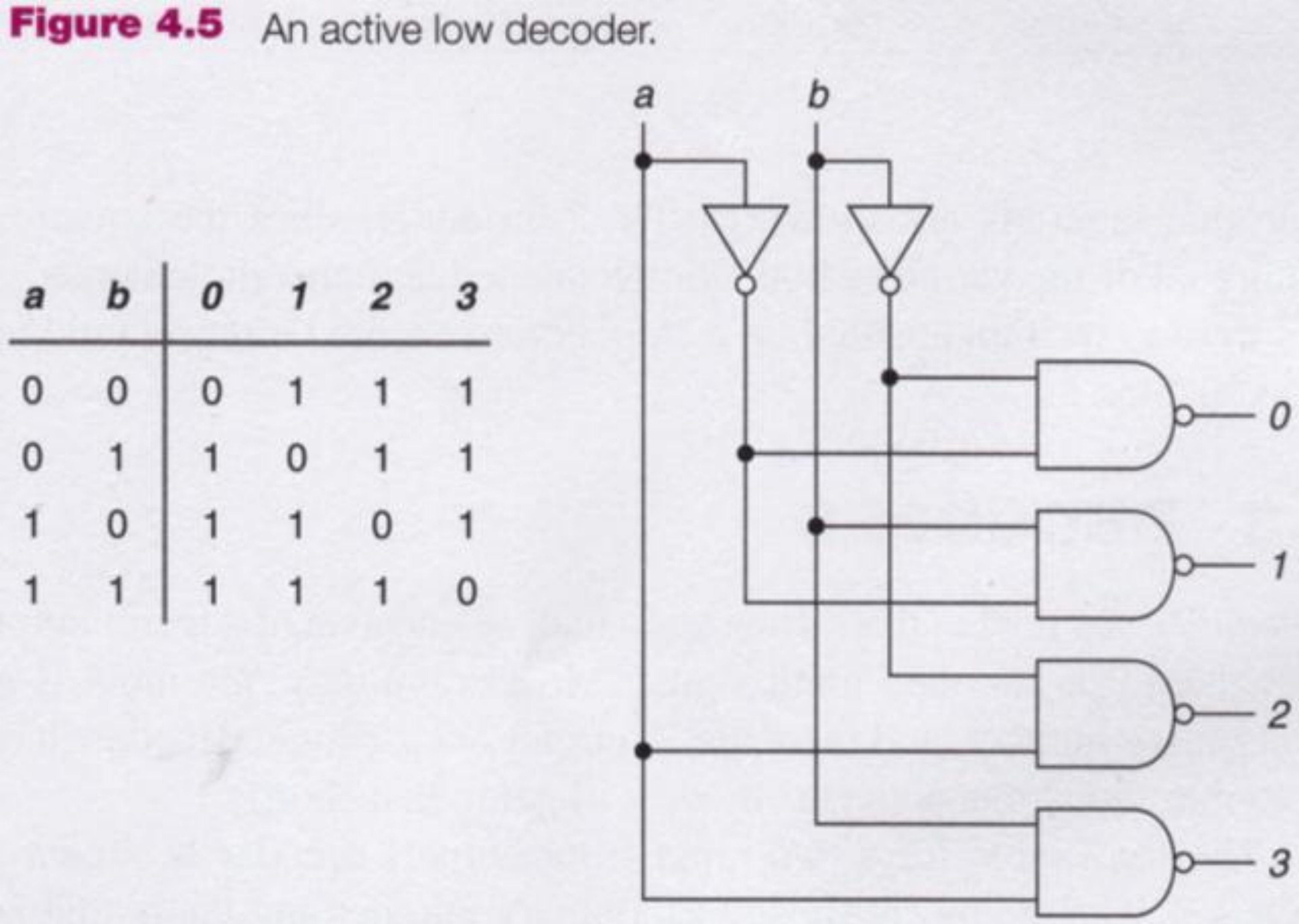
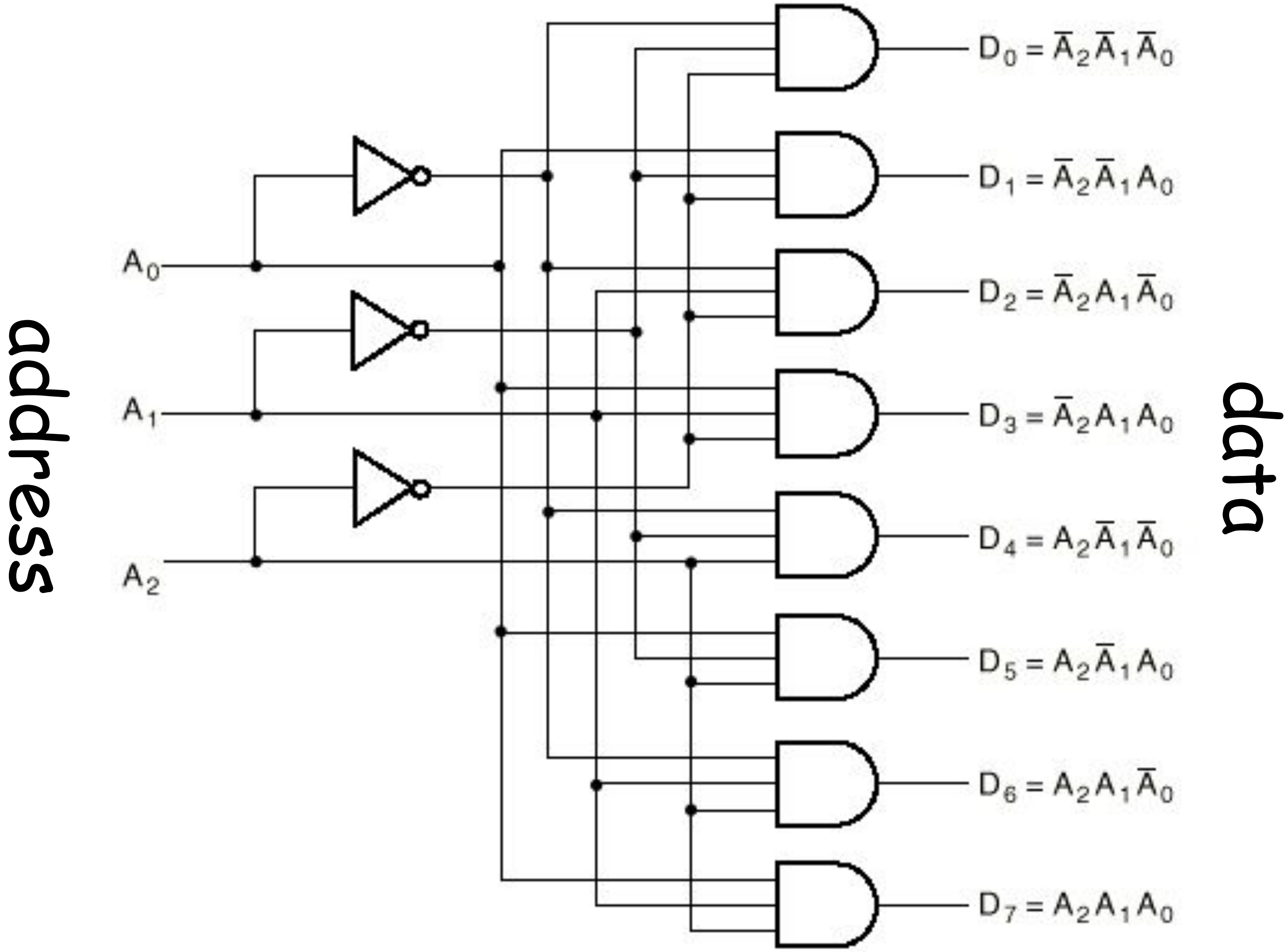


**Figure 4.5** An active low decoder.

| *a* | *b* | *0* | *1* | *2* | *3* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

# 3-to-8 Decoder


address
$A_0$
$A_1$
$A_2$
$D_0 = \overline{A}_2\overline{A}_1\overline{A}_0$
$D_1 = \overline{A}_2\overline{A}_1A_0$
$D_2 = \overline{A}_2A_1\overline{A}_0$
$D_3 = \overline{A}_2A_1A_0$
$D_4 = A_2\overline{A}_1\overline{A}_0$
$D_5 = A_2\overline{A}_1A_0$
$D_6 = A_2A_1\overline{A}_0$
$D_7 = A_2A_1A_0$
data

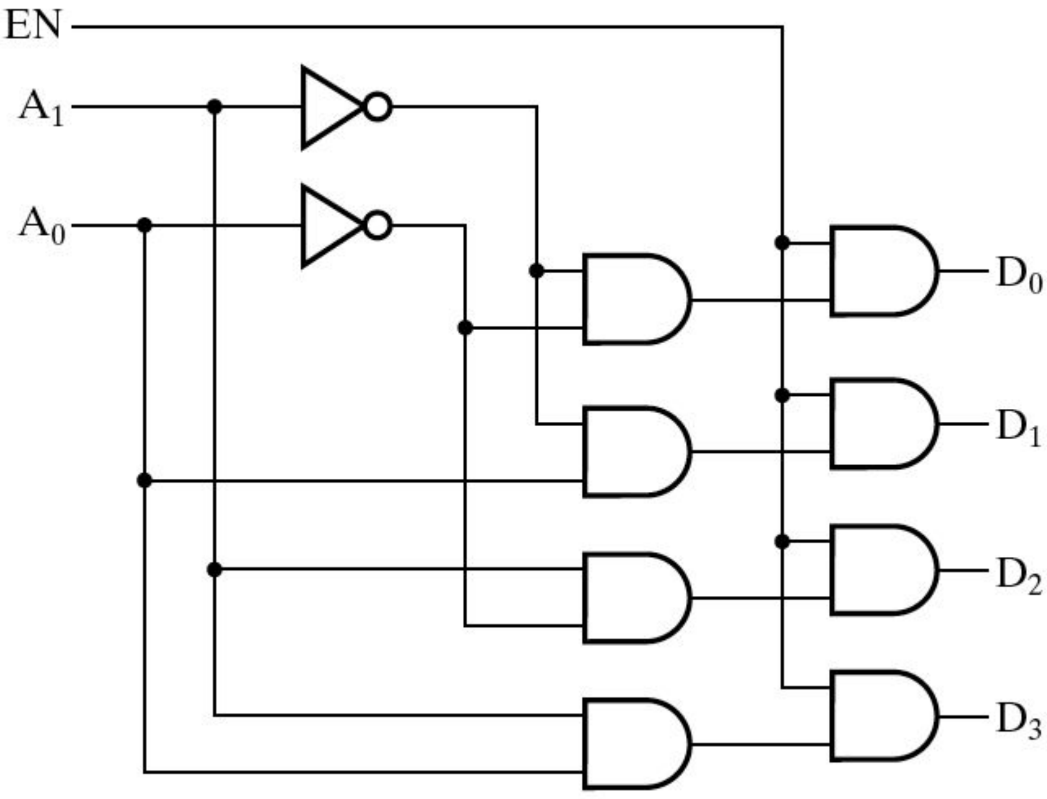
# 3-to-8 Decoder (cont.)

- Three inputs, $A_0$, $A_1$, $A_2$, are decoded into eight outputs, $D_0$ through $D_7$
- Each output $D_i$ represents one of the minterms of the 3 input variables.
- $D_i = 1$ when the binary number $A_2A_1A_0 = i$
- Shorthand: $D_i = m_i$
- The output variables are <u>*mutually exclusive*</u>; exactly one output has the value 1 at any time, and the other seven are 0.

# Decoder with enable

| **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)



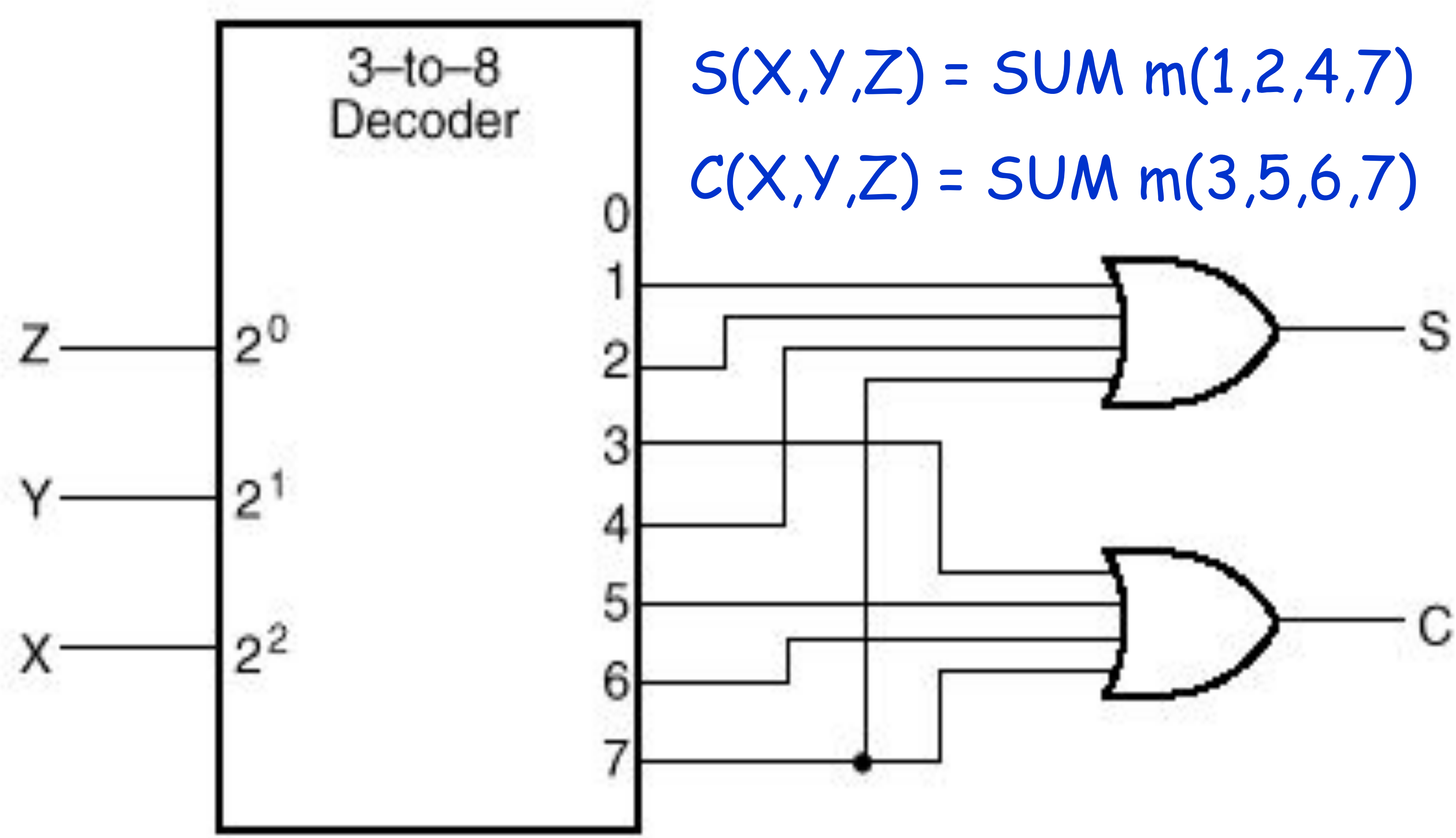
(b)

*

# Implementing Boolean functions using decoders

- Any combinational circuit can be constructed using decoders and OR gates!
- Here is an example:
  Implement a full adder circuit with a decoder and two OR gates.
- Recall full adder equations, and let X, Y, and Z be the inputs:
  - $S(X,Y,Z) = X+Y+Z = \Sigma m(1,2,4,7)$
  - $C\,(X,Y,Z) = \Sigma m(3, 5, 6, 7)$.
- Since there are 3 inputs and a total of 8 minterms, we need a 3-to-8 decoder.

# Implementing a Binary Adder Using a Decoder



S(X,Y,Z) = SUM m(1,2,4,7)

C(X,Y,Z) = SUM m(3,5,6,7)

*

# ENCODERS

- An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has $2^n$ input lines and $n$ output lines.
- The output lines generate the binary equivalent to the input line whose value is 1.

# Encoders (cont.)

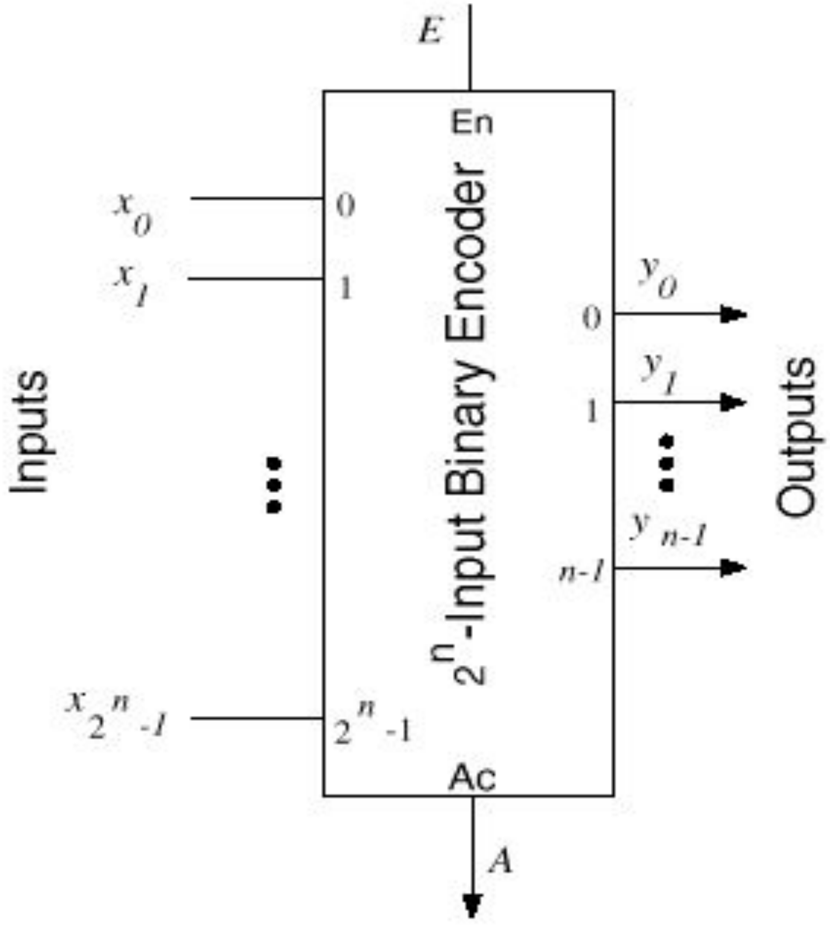
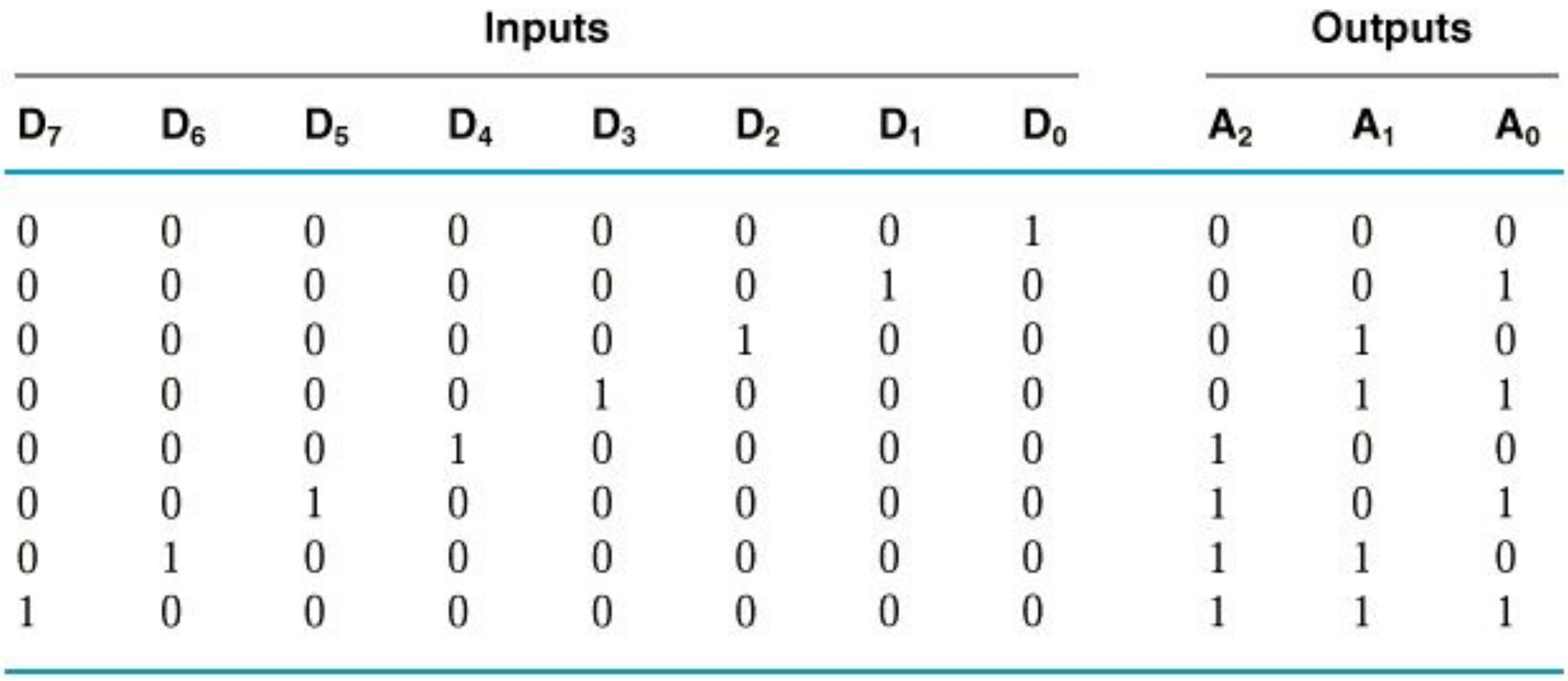


Figure 9.12: 2–-input binary encoder.

# Encoder Example

- Example: 8-to-3 binary encoder (octal-to-binary)



| Inputs | | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | $A_2$ | $A_1$ | $A_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

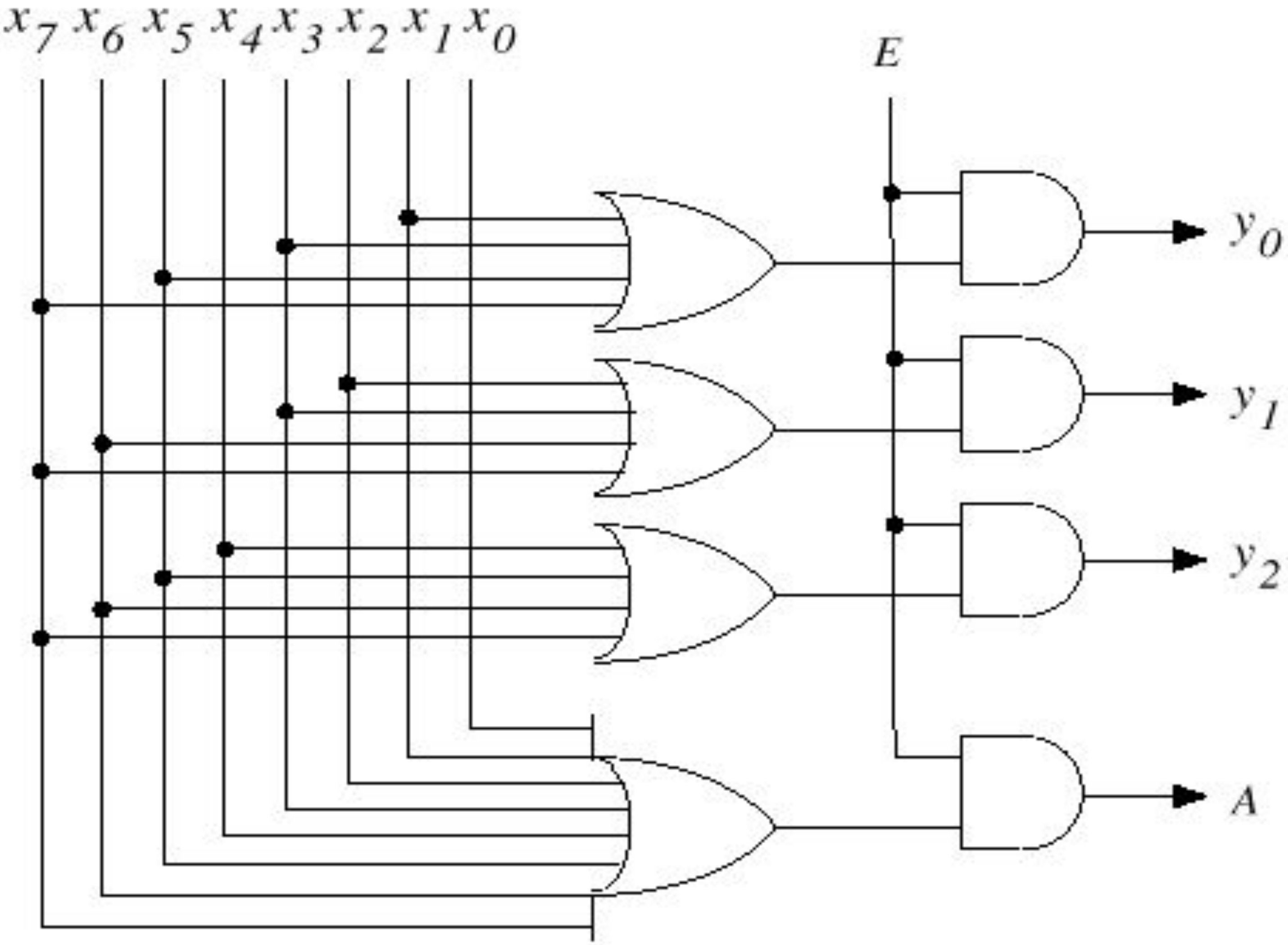


Figure 9.13: mplementation of an 8-input binary encoder.

*

# Encoder Design Issues

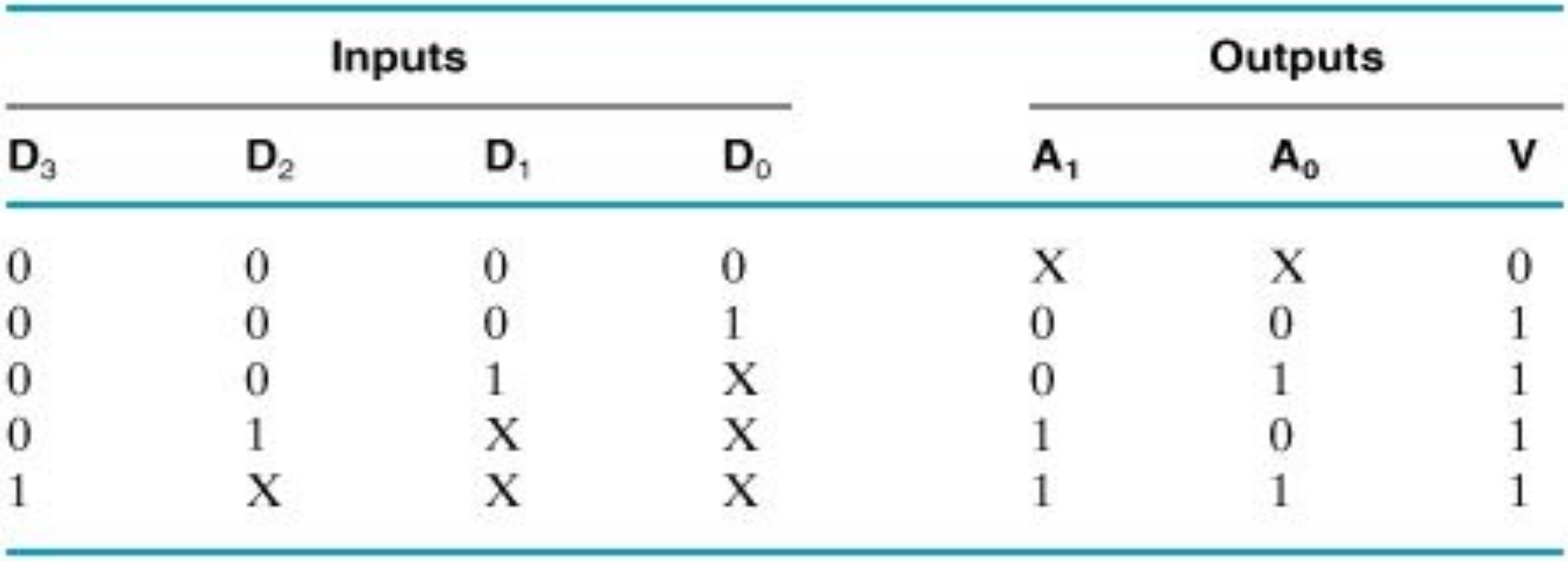
- There are two ambiguities associated with the design of a simple encoder:

1. Only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination (for example, if $D_3$ and $D_6$ are 1 simultaneously, the output of the encoder will be 111.
2. An output with all 0's can be generated when all the inputs are 0's,or when $D_0$ is equal to 1.

# PRIORITY ENCODERS

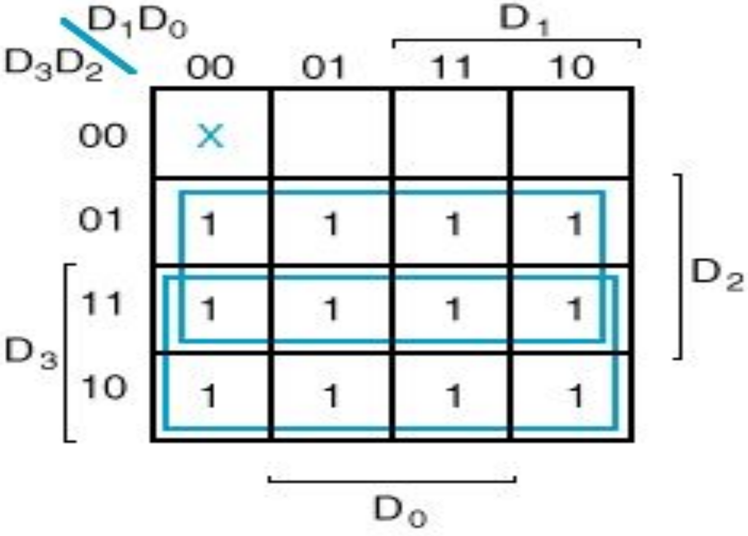
- Solves the ambiguities mentioned above.
- Multiple asserted inputs are allowed; one has priority over all others.
- Separate indication of no asserted inputs.

Example:4-to-2 Priority Encoder



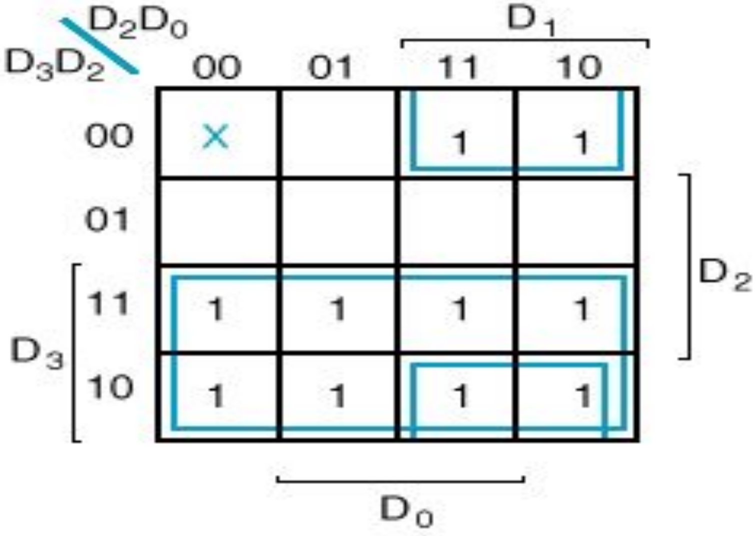
| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| $D_3$ | $D_2$ | $D_1$ | $D_0$ | $A_1$ | $A_0$ | $V$ |
| 0 | 0 | 0 | 0 | X | X | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 1 | 1 |
| 0 | 1 | X | X | 1 | 0 | 1 |
| 1 | X | X | X | 1 | 1 | 1 |

- The operation of the priority encoder is such that:
  - If two or more inputs are equal to 1 at the same time, the input in the highest-numbered position will take precedence.
  - A ***valid output indicator***, designated by V, is set to 1 only when one or more inputs are equal to 1.
  - $V = D_3 + D_2 + D_1 + D_0$ by inspection.



| D₃D₂ \ D₁D₀ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | | | |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

$A_1 = D_2 + D_3$

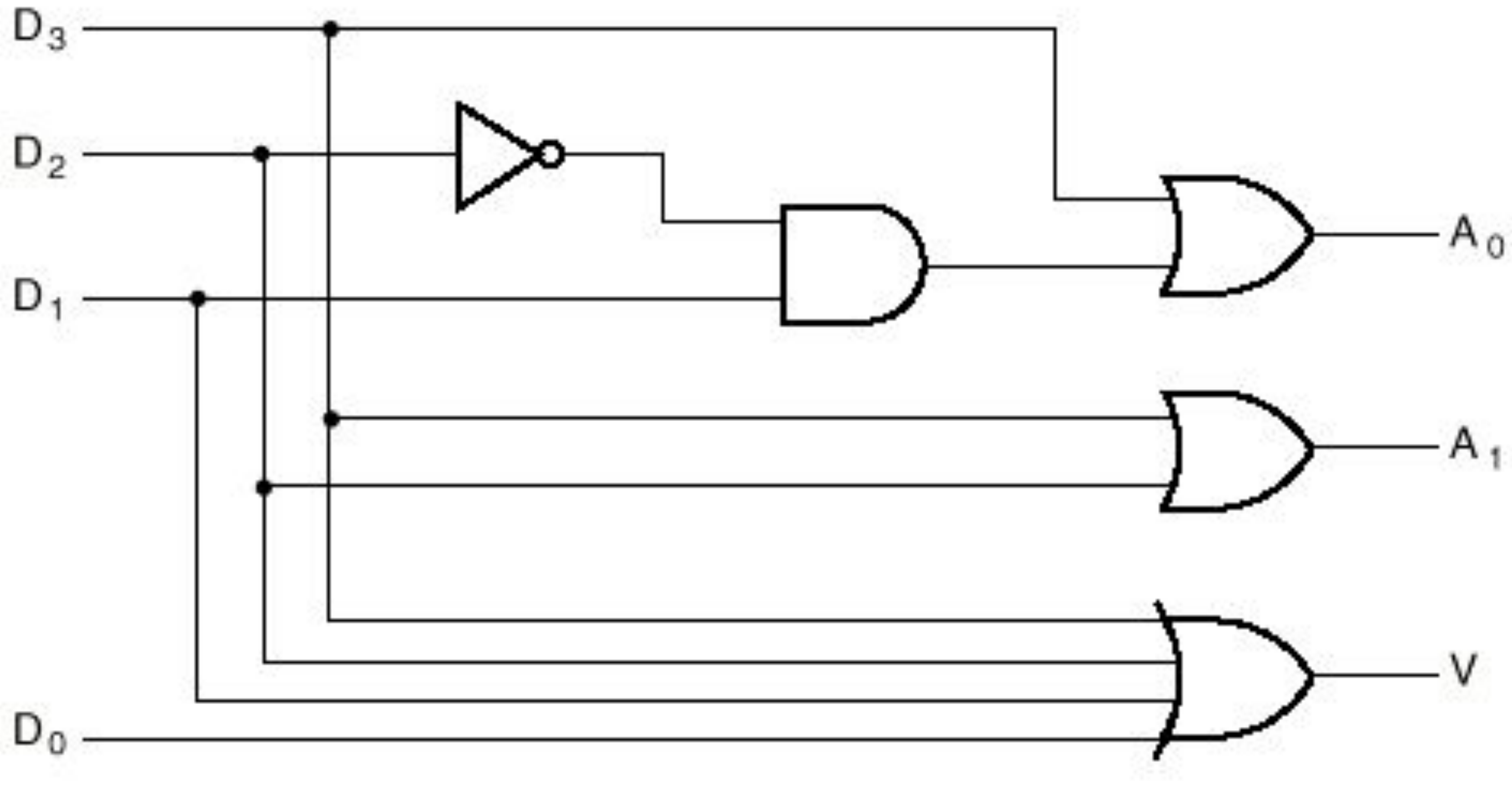


| D₃D₂ \ D₂D₀ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | | 1 | 1 |
| 01 | | | | |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

$A_0 = D_3 + D_1\overline{D}_2$

*

# Logic Diagram



*

# 8-to-3 Priority Encoder

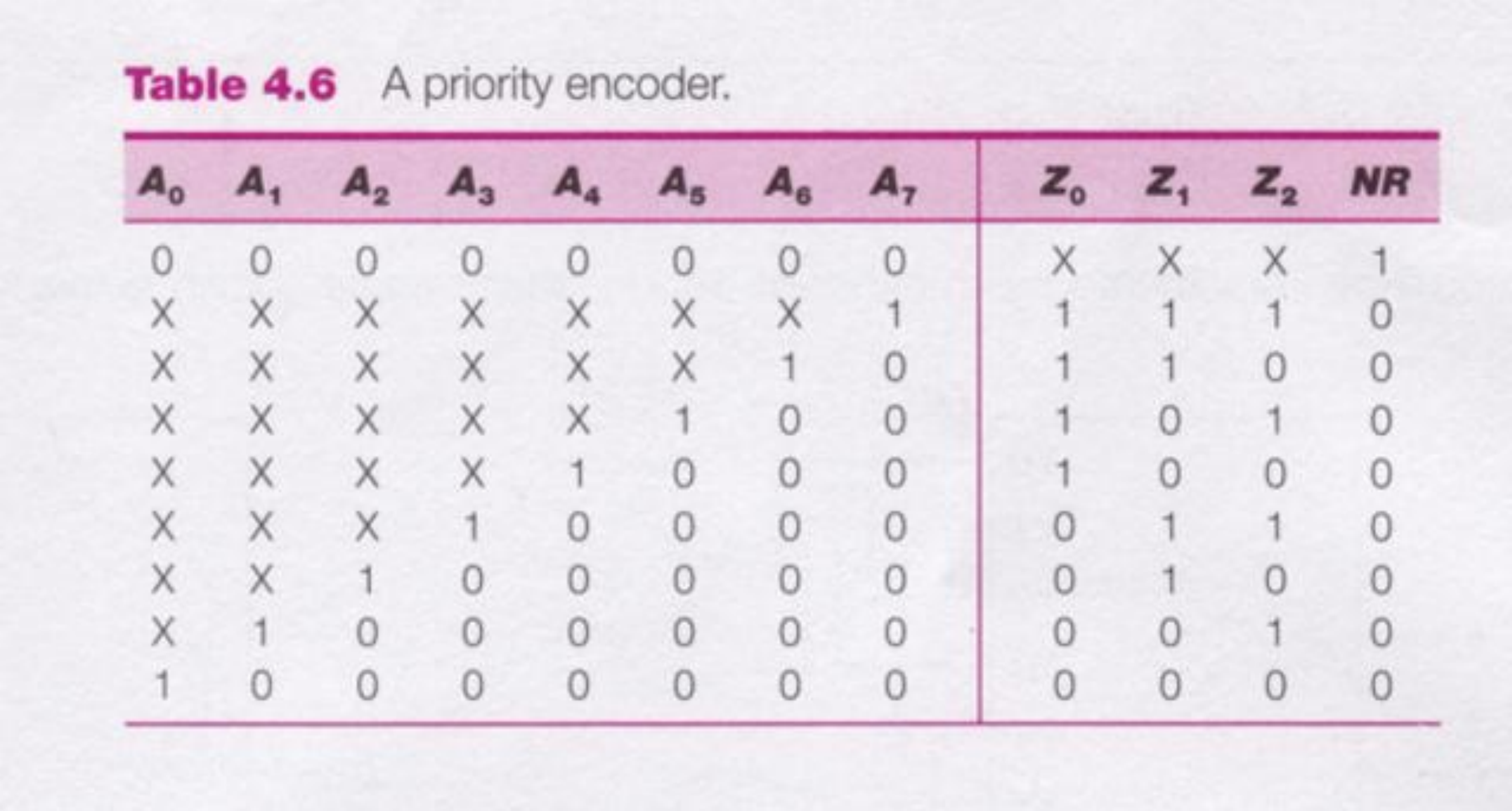
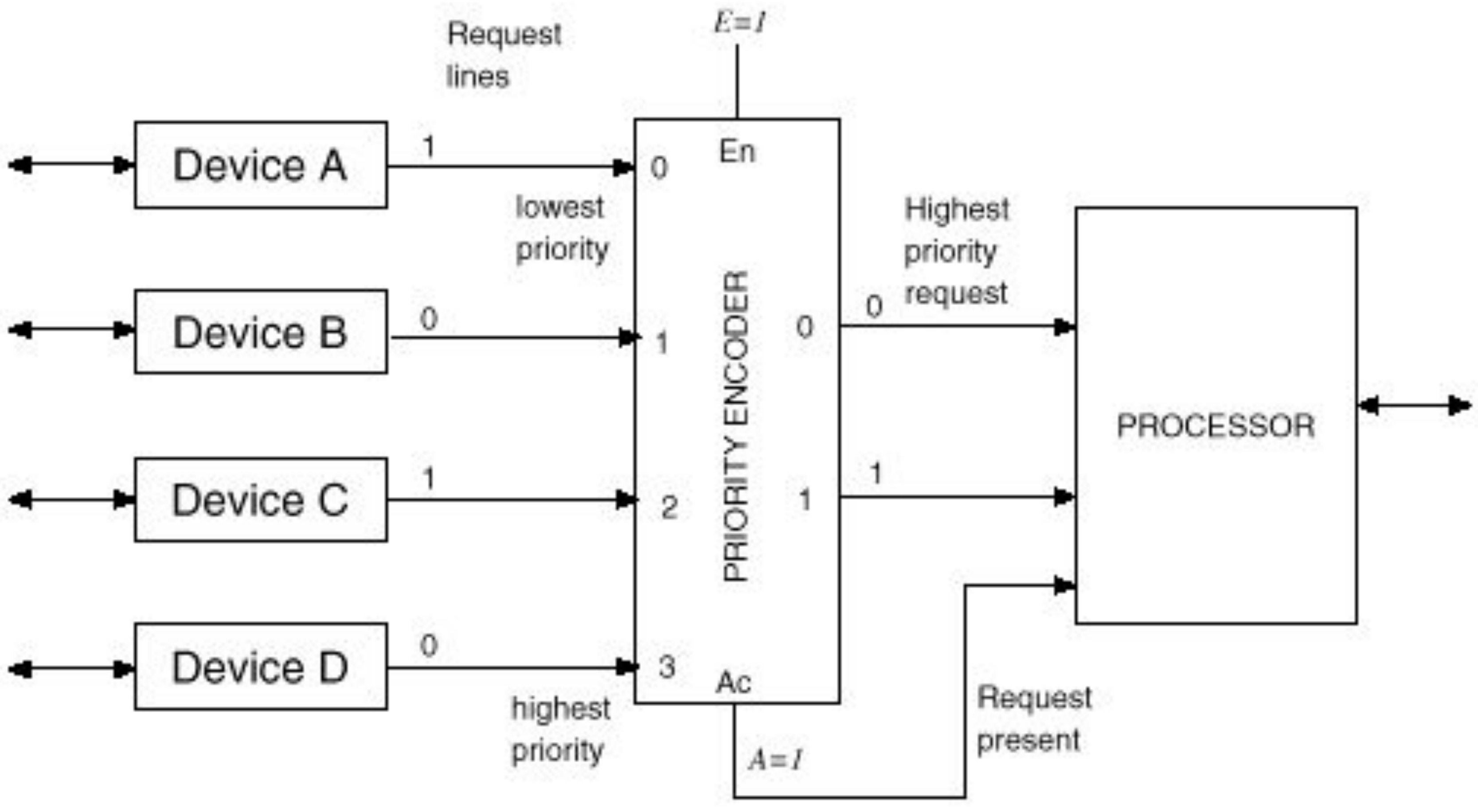


**Table 4.6** A priority encoder.

| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $Z_0$ | $Z_1$ | $Z_2$ | $NR$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 1 |
| X | X | X | X | X | X | X | 1 | 1 | 1 | 1 | 0 |
| X | X | X | X | X | X | 1 | 0 | 1 | 1 | 0 | 0 |
| X | X | X | X | X | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| X | X | X | X | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| X | X | X | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| X | X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Uses of priority encoders



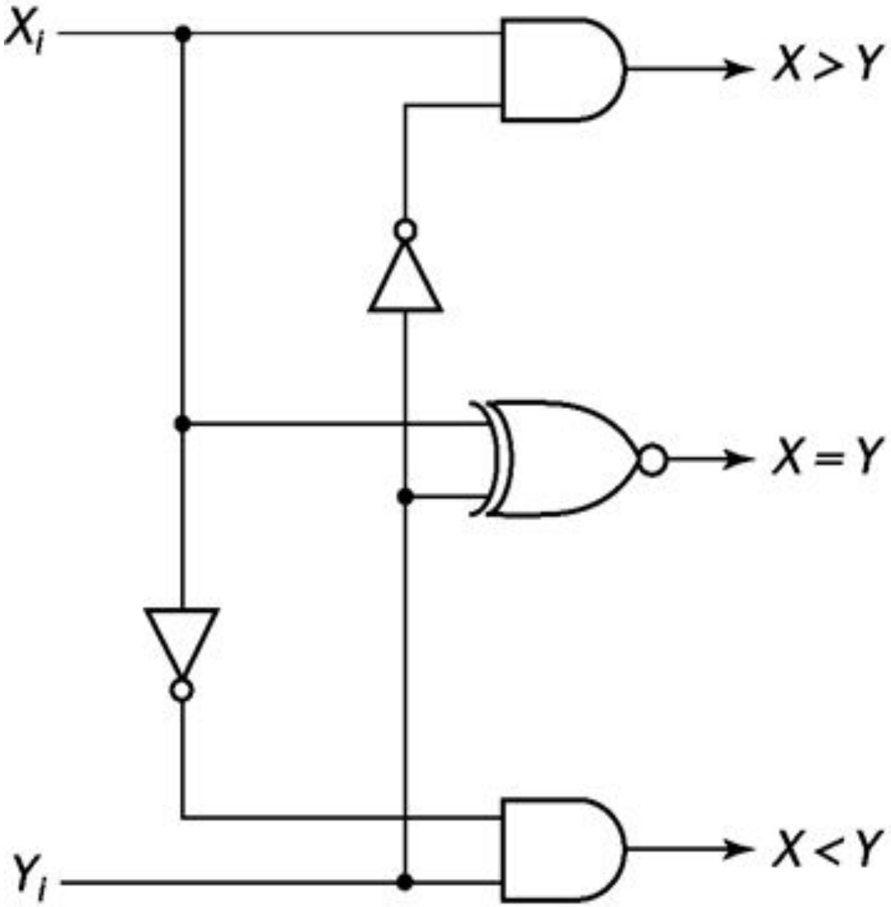
Figurc 9.17: Resolving interrupt requests us ng a prio ity encoder.

# COMPARATOR

## MAGNITUDE COMPARATOR

- A magnitude comparator determines the larger of two binary numbers.
- The comparison of two numbers and gives outputs: A >B, A=B, A<B
- Design Approaches
  - the truth table- $2^{2n}$ entries - too cumbersome for large n
  - use inherent regularity of the problem to reduce design efforts

# 1 bit comparators



(a)

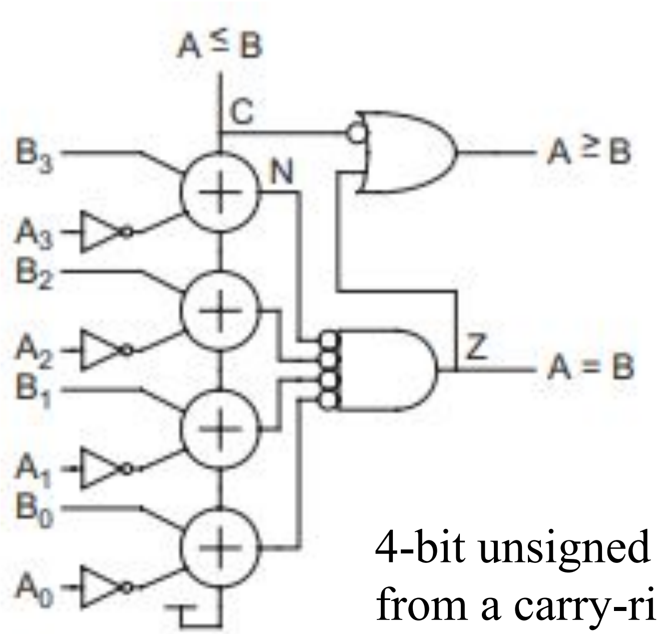
| $X_i$ | $Y_i$ | $X > Y$ | $X = Y$ | $X < Y$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |

(b)

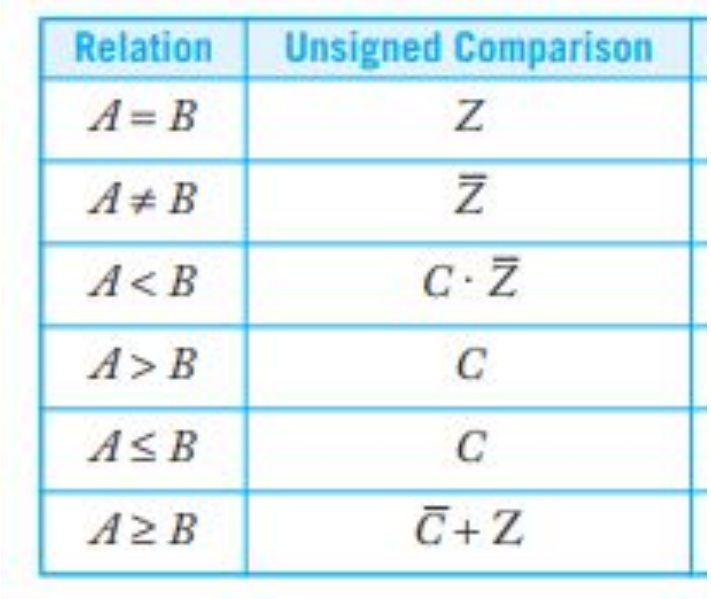
- X>Y only if Xi=1, Yi=0
- X<Y only if Xi=0, Yi=1
- X=Y only if Xi=Yi=0 or Xi=Yi=1

# 4 bit Comparator

- To compare two unsigned numbers A and B, compute B – A = B + A + 1.
- If there is a carry-out, A ≤ B; otherwise, A > B.
- The relative magnitude is determined from the carry-out (C) and zero (Z) signals according to Table



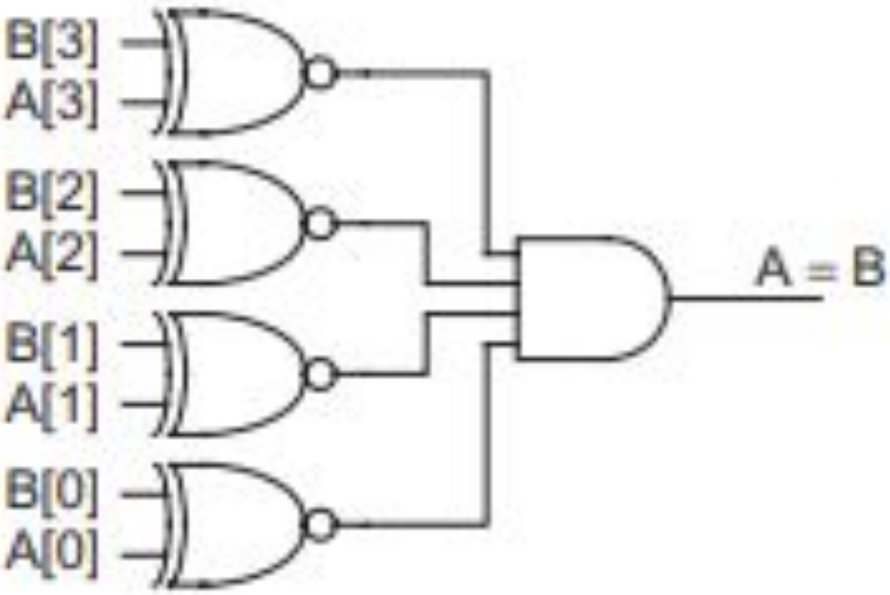
4-bit unsigned comparator built from a carry-ripple adder and two's complementer.



| Relation | Unsigned Comparison |
|---|---|
| $A = B$ | $Z$ |
| $A \neq B$ | $\overline{Z}$ |
| $A < B$ | $C \cdot \overline{Z}$ |
| $A > B$ | $C$ |
| $A \leq B$ | $C$ |
| $A \geq B$ | $\overline{C} + Z$ |

# EQUALITY COMPARATOR

- An equality comparator determines if (A = B) with XNOR gates and a ones detector,

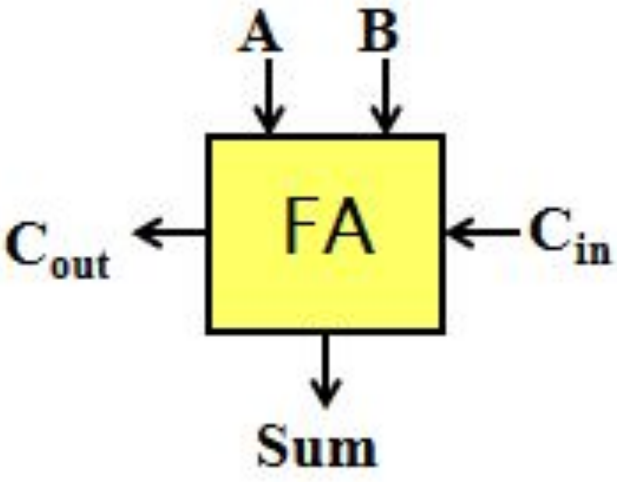


*

# ADDITION OF BINARY NUMBERS

*Full Adder.* The full adder is the fundamental building block of most arithmetic circuits:

The sum and carry outputs are described as:

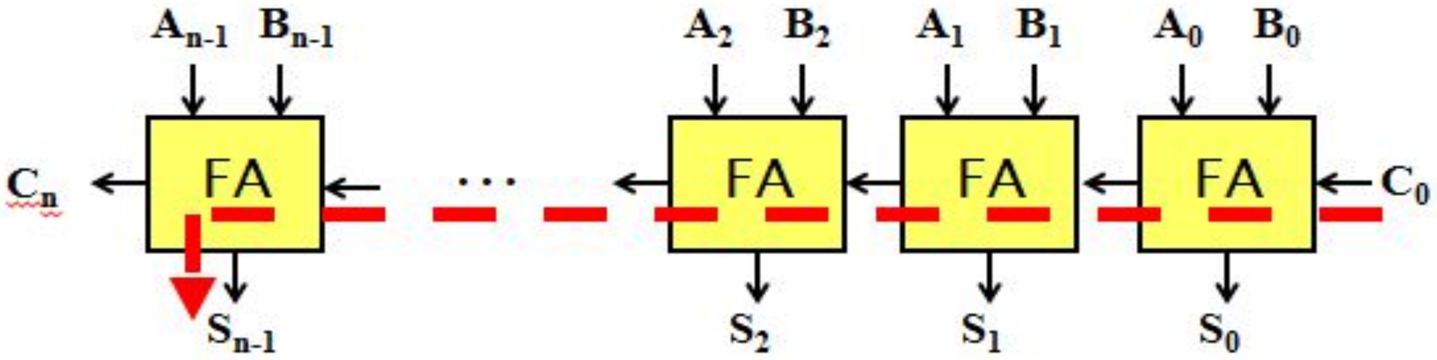


$$\text{Sum} = A \oplus B \oplus \text{Cin}$$

$$\text{Cout} = A.B + B.\text{Cin} + A.\text{Cin}$$

# Carry-Ripple Adder

Simplest design: Cascading full adders



Critical path goes from $C_{in}$ to $C_{out}$

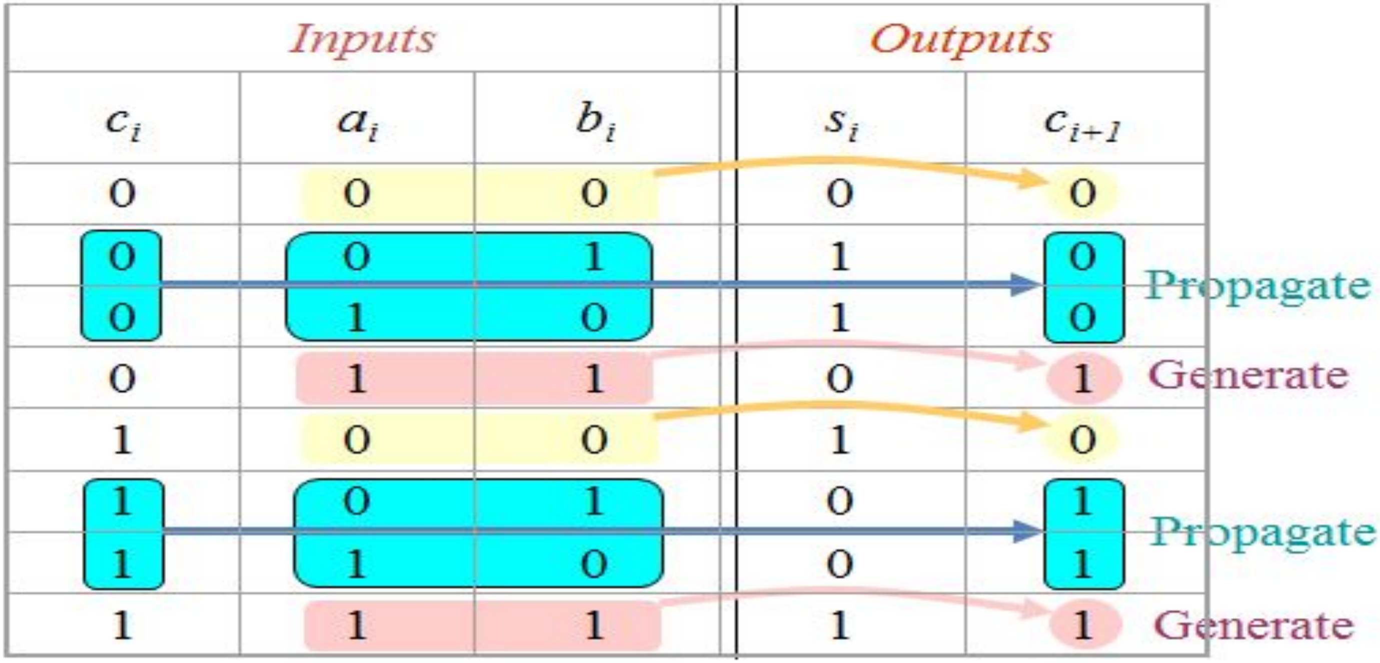
**Worst case delay linear with the number of bits**

$$t_d = O(N)$$

$$t_{adder} \approx (N-1)t_{carry} + t_{sum}$$

# Carry Look-ahead Adder

- Try to “predict” *Ci* earlier than $T_c*n$
- Instead of passing through *n* stages, compute $C_i$ separately
- A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits.



| Inputs | | | Outputs | | |
|---|---|---|---|---|---|
| $c_i$ | $a_i$ | $b_i$ | $s_i$ | $c_{i+1}$ | |
| 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 0 | Propagate |
| 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 1 | Generate |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 0 | 1 | Propagate |
| 1 | 1 | 0 | 0 | 1 | |
| 1 | 1 | 1 | 1 | 1 | Generate |

- Carry look-ahead logic uses the concepts of *generating* and *propagating* carries.

$$g_i = a_i \bullet b_i$$

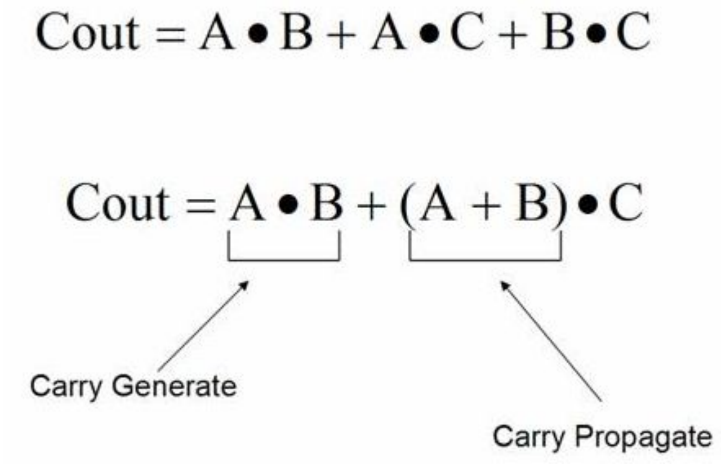
$$p_i = a_i \oplus b_i$$

- The addition of two inputs *A* and *B* is said to *generate* if the addition will always carry, regardless of whether there is an input carry

$$s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$$

- The addition of two inputs *A* and *B* is said to *propagate* if the addition will carry whenever there is an input carry

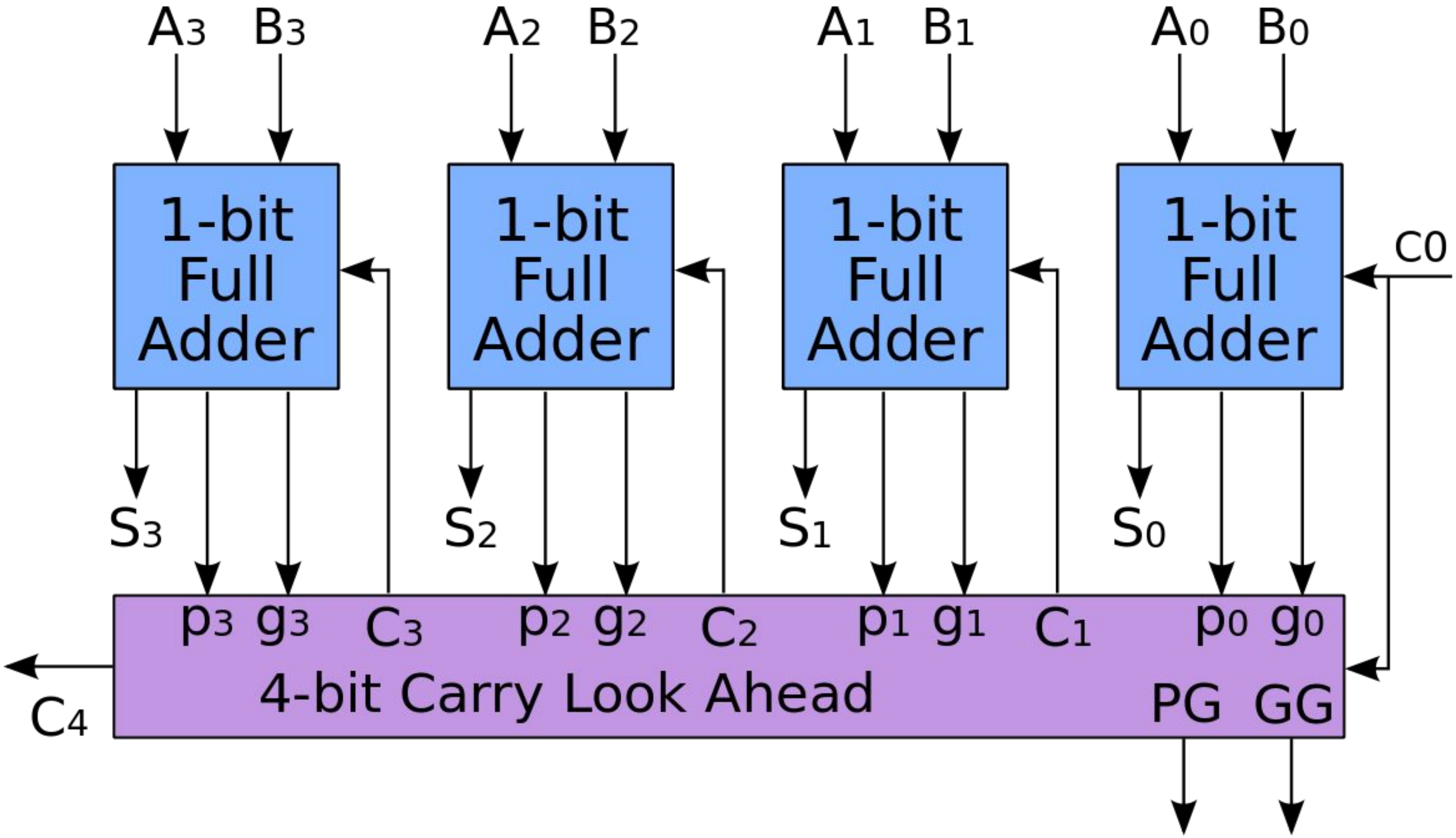
$$c_{i+1} = g_i + p_i c_i$$



$$\mathrm{Cout} = \mathrm{A} \bullet \mathrm{B} + \mathrm{A} \bullet \mathrm{C} + \mathrm{B} \bullet \mathrm{C}$$

$$\mathrm{Cout} = \underbrace{\mathrm{A} \bullet \mathrm{B}}_{\text{Carry Generate}} + \underbrace{(\mathrm{A} + \mathrm{B})}_{\text{Carry Propagate}} \bullet \mathrm{C}$$

# CLA: 4-bit adder



$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1$$

$$C_2 = G_1 + P_1 (G_0 + P_0 C_0)$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2$$

$$C_3 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$$

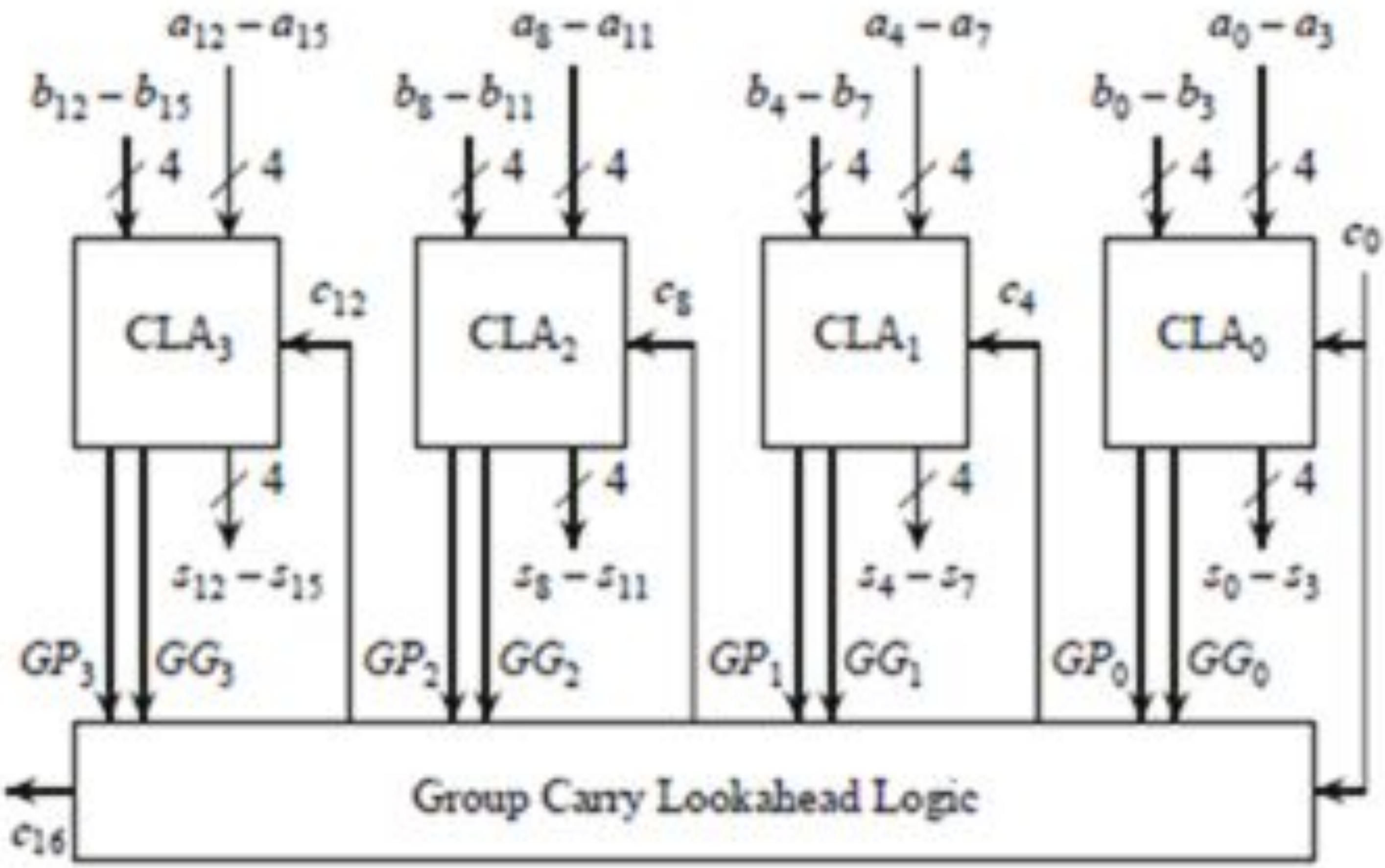
$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3$$

$$C_4 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0)$$

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

- As the  number of bit increases length of the expression increases.
- However each expression is only three logic levels deep.
- Logic does rapidly become cumbersome and provides fan-out and fan-in problem.


$a_{12} - a_{15}$
$b_{12} - b_{15}$
$a_8 - a_{11}$
$b_8 - b_{11}$
$a_4 - a_7$
$b_4 - b_7$
$a_0 - a_3$
$b_0 - b_3$
4
$c_0$
CLA$_3$
$c_{12}$
CLA$_2$
$c_8$
CLA$_1$
$c_4$
CLA$_0$
$s_{12} - s_{15}$
$s_8 - s_{11}$
$s_4 - s_7$
$s_0 - s_3$
$GP_3$
$GG_3$
$GP_2$
$GG_2$
$GP_1$
$GG_1$
$GP_0$
$GG_0$
Group Carry Lookahead Logic
$c_{16}$

Carry lookahead depends on two things:

- Calculating, for each digit position, whether that position is going to propagate a carry if one comes in from the right.
- Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of four digits are chosen. Then the sequence of events goes something like this:

- All 1-bit adders calculate their results. Simultaneously, the lookahead units perform their calculations.
- Suppose that a carry arises in a particular group. Within at most five gate delays, that carry will emerge at the left-hand end of the group and start propagating through the group to its left.
- If that carry is going to propagate all the way through the next group, the lookahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group*, the lookahead unit is immediately (within one gate delay) able to tell the *next* group to the left that it is going to receive a carry – and, at the same time, to tell the next lookahead unit to the left that a carry is on its way.
- The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move four times as fast, leaping from one lookahead carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group.

Let Δ be One gate delay

- Delay of one Δ to calculate p, g
- Delay of two Δ to generate Ci
- Delay of two Δ to generate Si

- Total of five Δ regardless of n.

CLA compared to ripple-carry adder:

- 4 times Faster but delay still linear (w.r.t. # of bits)
- Larger area
  - P, G signal generation
  - Carry generation circuits
  - Carry generation circuit for each bit position (no re-use)
- Limitation: cannot go beyond 4 bits of look-ahead

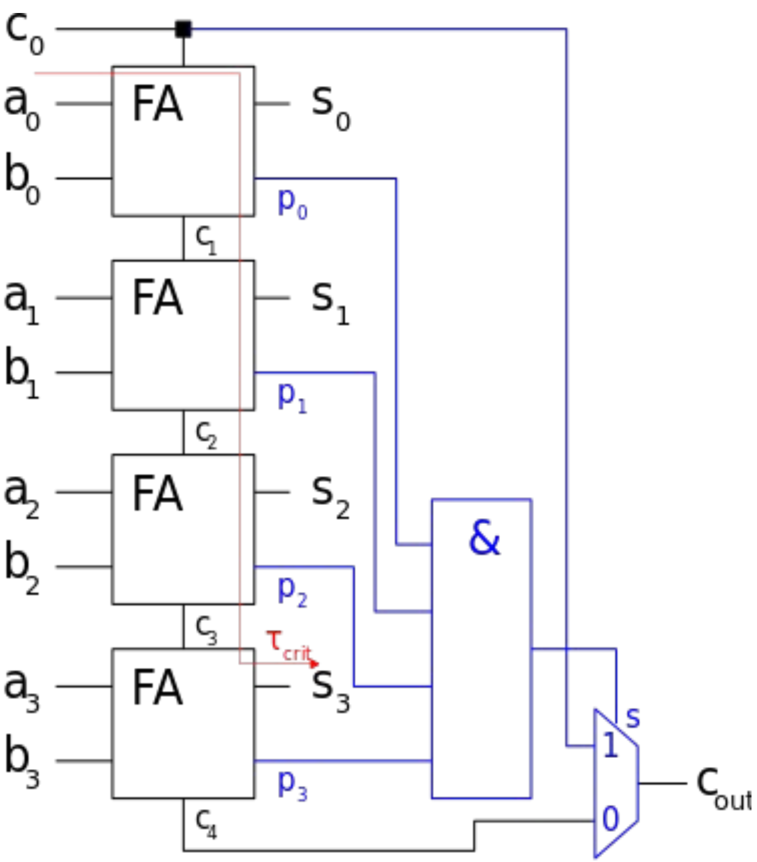
  Large fan-out slows down carry generation

```
module CLA_4bmod(sum,c_4,a,b,c_0);
input [3:0]a,b;
input c_0;
output [3:0]sum;
output c_4;
wire p0,p1,p2,p3,g0,g1,g2,g3;
wire c1,c2,c3,c4;
assign
p0=a[0]^b[0],
p1=a[1]^b[1],
p2=a[2]^b[2],
p3=a[3]^b[3],
g0=a[0]&b[0],
g1=a[1]&b[1],
g2=a[2]&b[2],
g3=a[3]&b[3];
```

```
assign
c1=g0|(p0&c_0),
c2=g1|(p1&g0)|(p1&p0&c_0),
c3=g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&
c_0),
c4=g3|(p3&g2)|(p3&p2&p1&g1)|(p3&p2&
p1&g0)|(p3&p2&p1&p0&c_0);
assign
sum[0]=p0^c_0,
sum[1]=p1^c1,
sum[2]=p2^c2,
sum[3]=p3^c3,
c_4=c4;
endmodule
```

# Carry Skip Adder

- A **carry-skip adder/ carry-bypass adder** improves the delay of a ripple-carry adder.
- A *n*-bit-carry-skip adder consists of a *n*-bit-carry-ripple-chain, a *n*-input AND-gate and one multiplexer.



- For each operand input bit pair ($a_i$ , $b_i$) the propagate-conditions $p_i = a_i \oplus b_i$ are determined using an XOR-Gate.

- Each propagate bit that is provided by the carry-ripple-chain is connected to the *n*-input AND-gate.

- The resulting bit is used as the select bit of a multiplexer that switches either the last carry-bit or the carry-in to the carry-out signal .
- The carry bit for each block can now "skip" over blocks with a *group* propagate signal set to logic 1 . This greatly reduces the latency of the adder
- The number of inputs of the AND-gate is equal to the width of the adder. For a large width, this becomes impractical and leads to additional delays, because the AND-gate has to be built as a tree.
- A good width is achieved, when the sum-logic has the same depth like the *n*-input AND-gate and the multiplexer.

```
module carry_skip_4bit(a, b, cin, sum,
cout);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;
wire [3:0] p;
wire c0;
wire bp;
ripple_carry_4_bit rca1
(a[3:0],b[3:0],cin,sum[3:0],c0);
propagate_p p1(a,b,p,bp);
mux2X1 m0(c0,cin,bp,cout);
endmodule
```

```
 module propagate_p(a,b,p,bp);
 input [3:0] a,b;
 output [3:0] p;
 output bp;
 assign p= a^b;//get all propagate bits
 assign bp= &p;// and p0p1p2p3 bits
 endmodule
```

```
module ripple_carry_4_bit(a, b, cin,
sum, cout);
input [3:0] a,b;
input cin;
wire c1,c2,c3;
output [3:0] sum;
output cout;
full_adder fa0(a[0], b[0],cin,
sum[0],c1);
full_adder fa1(a[1], b[1],c1,
sum[1],c2);
full_adder fa2(a[2], b[2],c2,
sum[2],c3);
full_adder fa3(a[3], b[3],c3,
sum[3],cout);
endmodule
```

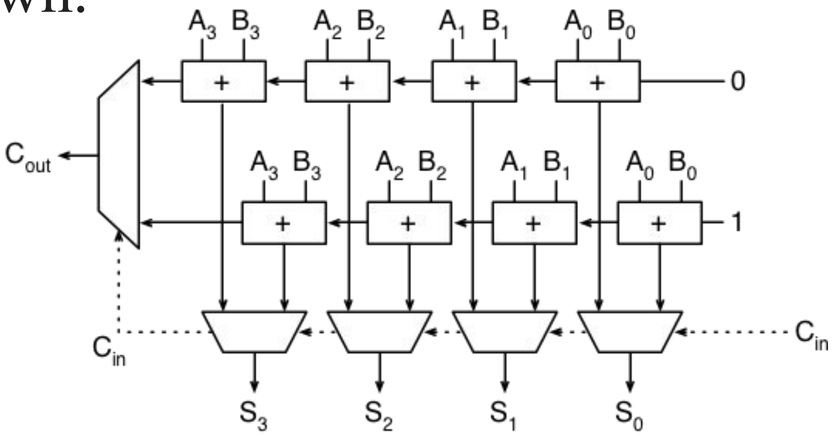
```
module full_adder(a,b,cin,sum, cout);
input a,b,cin;
output sum, cout;
wire x,y,z;
half_adder h1(a,b,x,y);
half_adder h2(x,cin,sum,z);
or or_1(cout,z,y);
endmodule
```

```
module half_adder( a,b, sum, cout );
input a,b;
output sum, cout;
xor xor_1 (sum,a,b);
and and_1 (cout,a,b);
endmodule
```

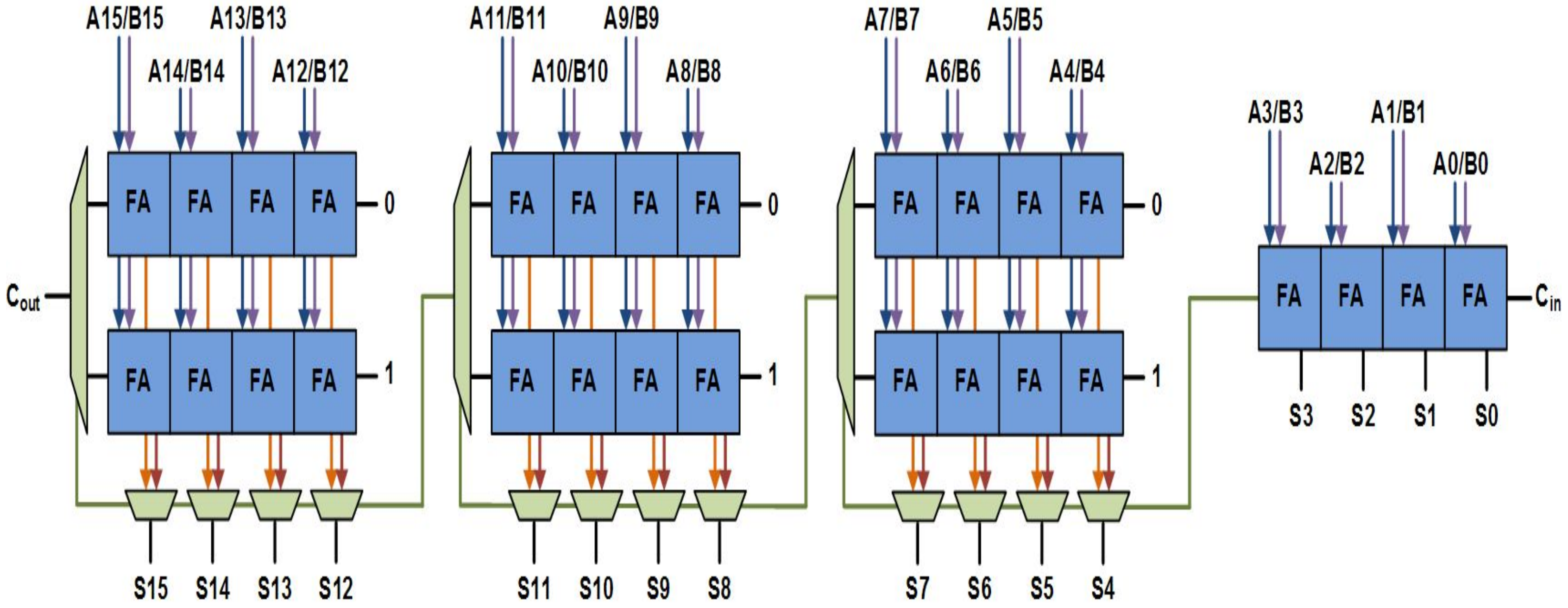
```
module mux2X1( in0,in1,sel,out);
input in0,in1;
input sel;
output out;
assign out=(sel)?in1:in0;
endmodule
```

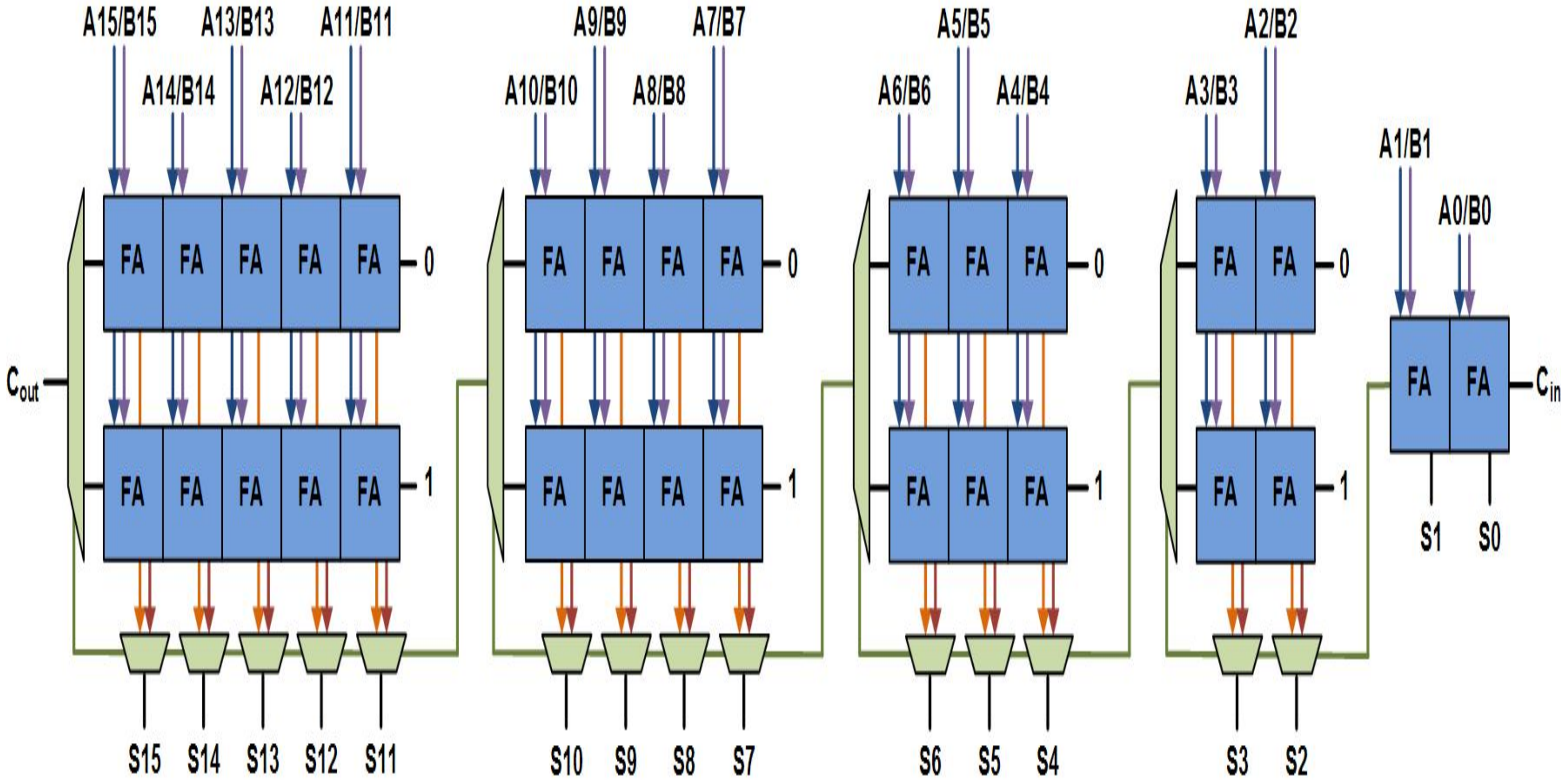
# Carry Select Adder

- Consists of two ripple carry adders and a multiplexer.
- Adding two n-bit numbers with a carry-select adder is done with two adders (therefore two ripple carry adders).
- The calculation is performed twice,
- one time with the assumption of the carry-in being zero the other assuming it will be one.
- After the two results are calculated, the correct sum, as well as the correct carry-out, is then selected with the multiplexer once the correct carry-in is known.



- The number of bits in each carry select block can be uniform, or variable. In the uniform case, the optimal delay occurs for a block size of (O√ n)
- When variable, the block size should have a delay, from addition inputs A and B to the carry out, equal to that of the multiplexer chain leading into it





Tadd=Tsetup+M*tcarry+(N/M)*Tmux+Tsum

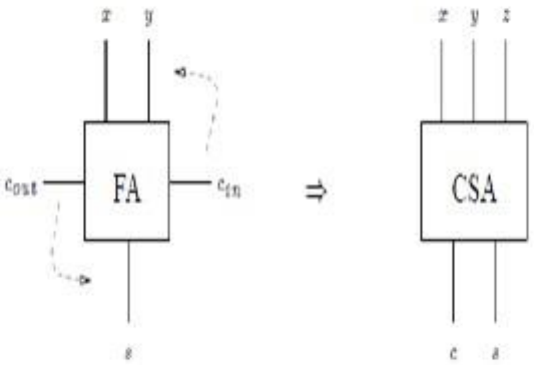
# Carry Save Adder

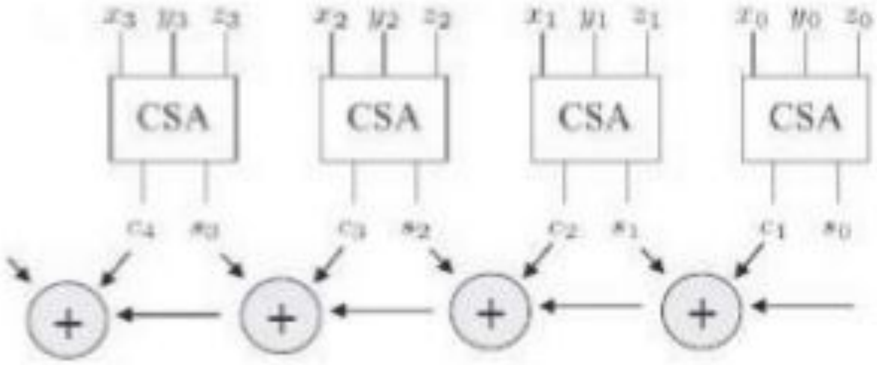
- Set of one-bit full adders, without any carry-chaining.
- n-bit CSA receives three n-bit operands, namely A,B and CIN and generates two n-bit result values SUM,COUT.
- The final stage must be a normal adder because we need to obtain a single output.

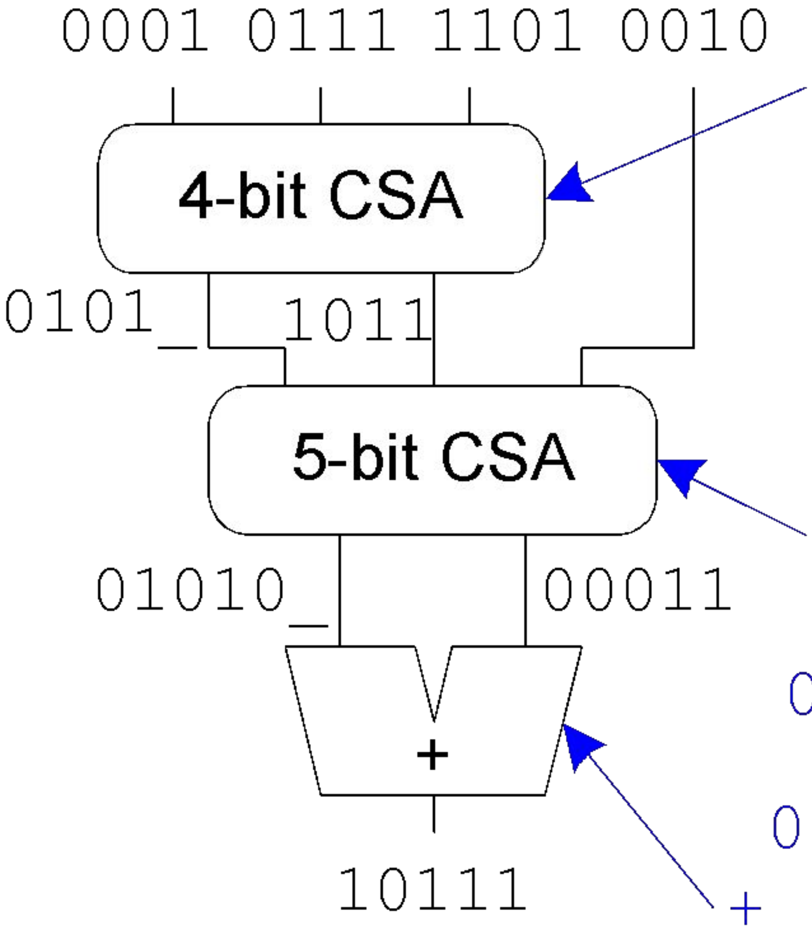
$$(C,S) = C + S = A_0 + A_1 + A_2$$

$$2c_{i+1} + s_i = a_{0,i} + a_{1,i} + a_{2,i}$$

$$i = 0,1,\ldots,n-1$$





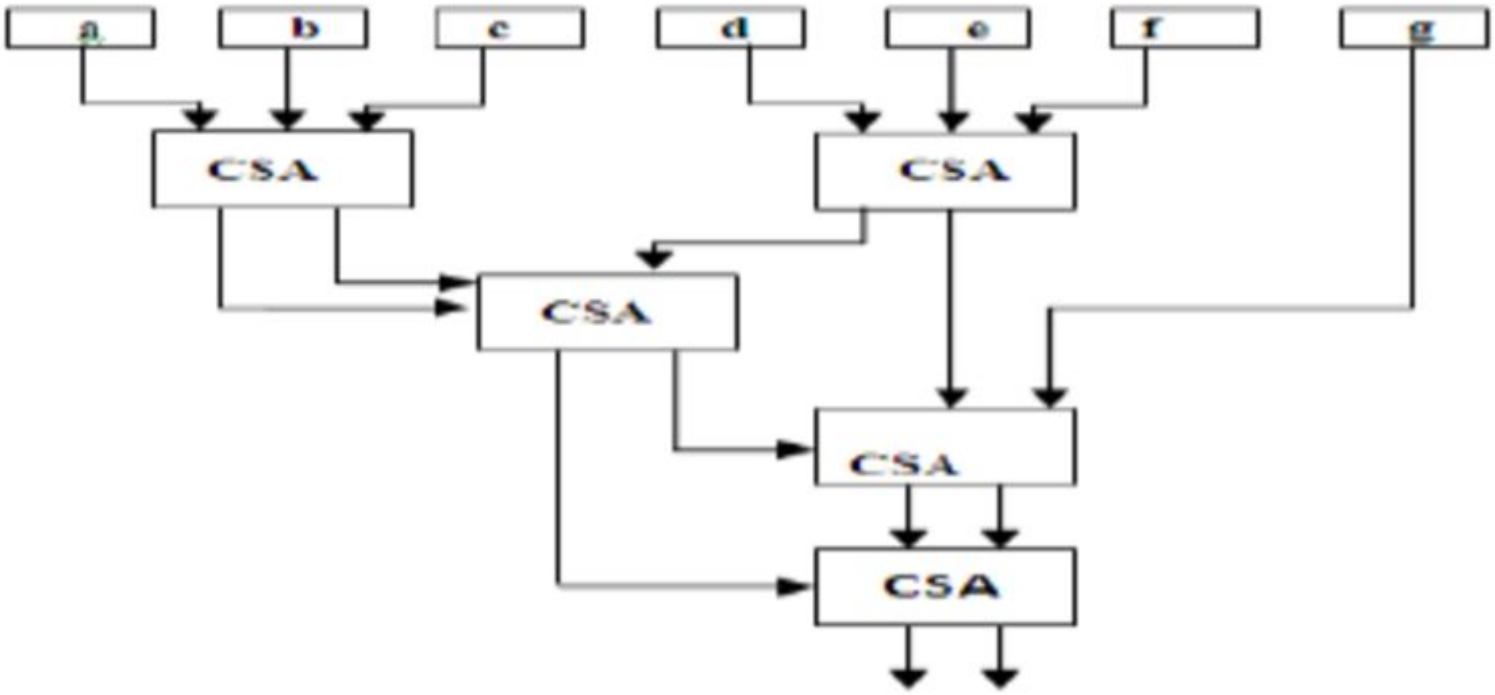


```
  0001   X
  0111   Y
 +1101   Z
 -----
  1011   S
 0101_   C

 0101_   X
  1011   Y
 +0010   Z
 -----
 00011   S
01010_   C

 01010_  A
+ 00011  B
-------
 10111   S
```

- The delay is the same as for a conventional look-ahead adder tree but uses much less circuitry.
- The irregularity of the tree causes a reduction in efficiency but this is relatively small (and becomes even smaller for large K).



# Multiplier

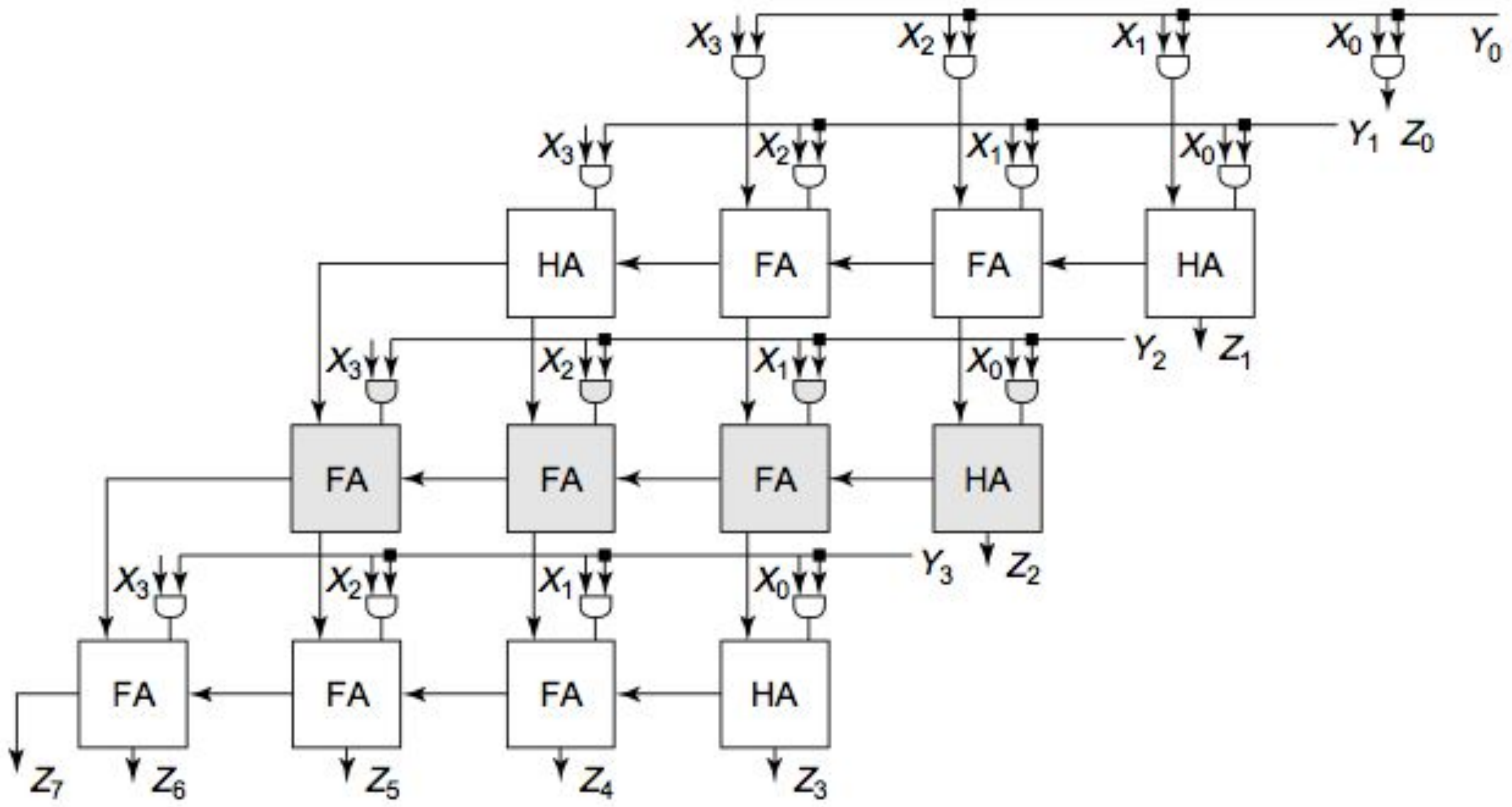
- Multiplicand: Y = ($y_{M-1}$, $y_{M-2}$, …, $y_1$, $y_0$)
- Multiplier: X = ($x_{N-1}$, $x_{N-2}$, …, $x_1$, $x_0$)
- Product:

$$P = \left( \sum_{j=0}^{M-1} y_j 2^j \right) \left( \sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

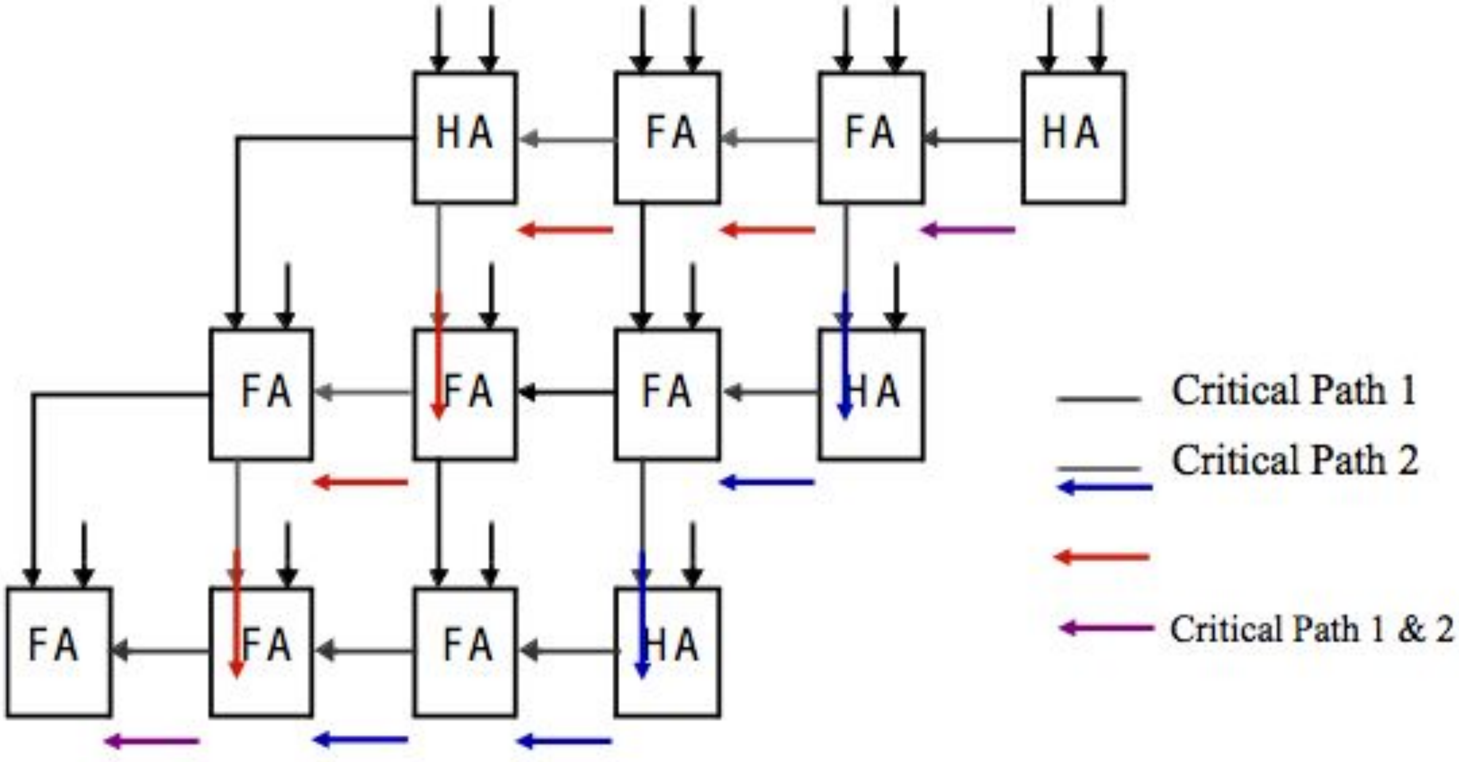
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | multiplicand |
| | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | multiplier |
| | | | | | | $x_0y_5$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | |
| | | | | | $x_1y_5$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | | |
| | | | | $x_2y_5$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | | partial products |
| | | | $x_3y_5$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | | |
| | | $x_4y_5$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | | |
| | $x_5y_5$ | $x_5y_4$ | $x_5y_3$ | $x_5y_2$ | $x_5y_1$ | $x_5y_0$ | | | | | | |
| $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | product |

```
        1 0 1 0 1 0      Multiplicand
x           1 0 1 1      Multiplier
  -----------------
        1 0 1 0 1 0    ┐
      1 0 1 0 1 0      │
    0 0 0 0 0 0        ├ Partial products
+ 1 0 1 0 1 0          ┘
  -----------------
  1 1 1 0 0 1 1 1 0      Result
```

# Array Multiplier



# Critical path



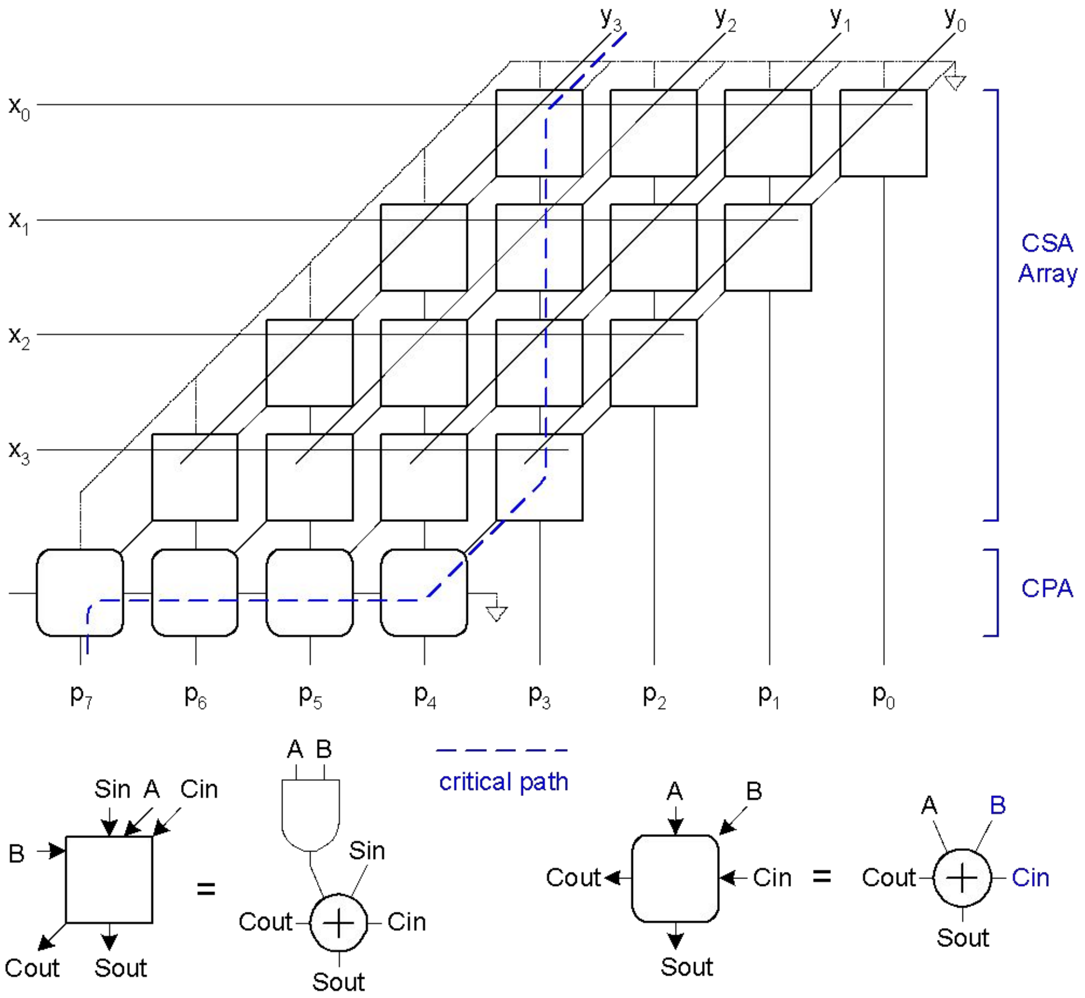
$$t_{mult} \approx [(M-1)+(N-2)]t_{carry} + (N-1)t_{sum} + (N-1)t_{and}$$

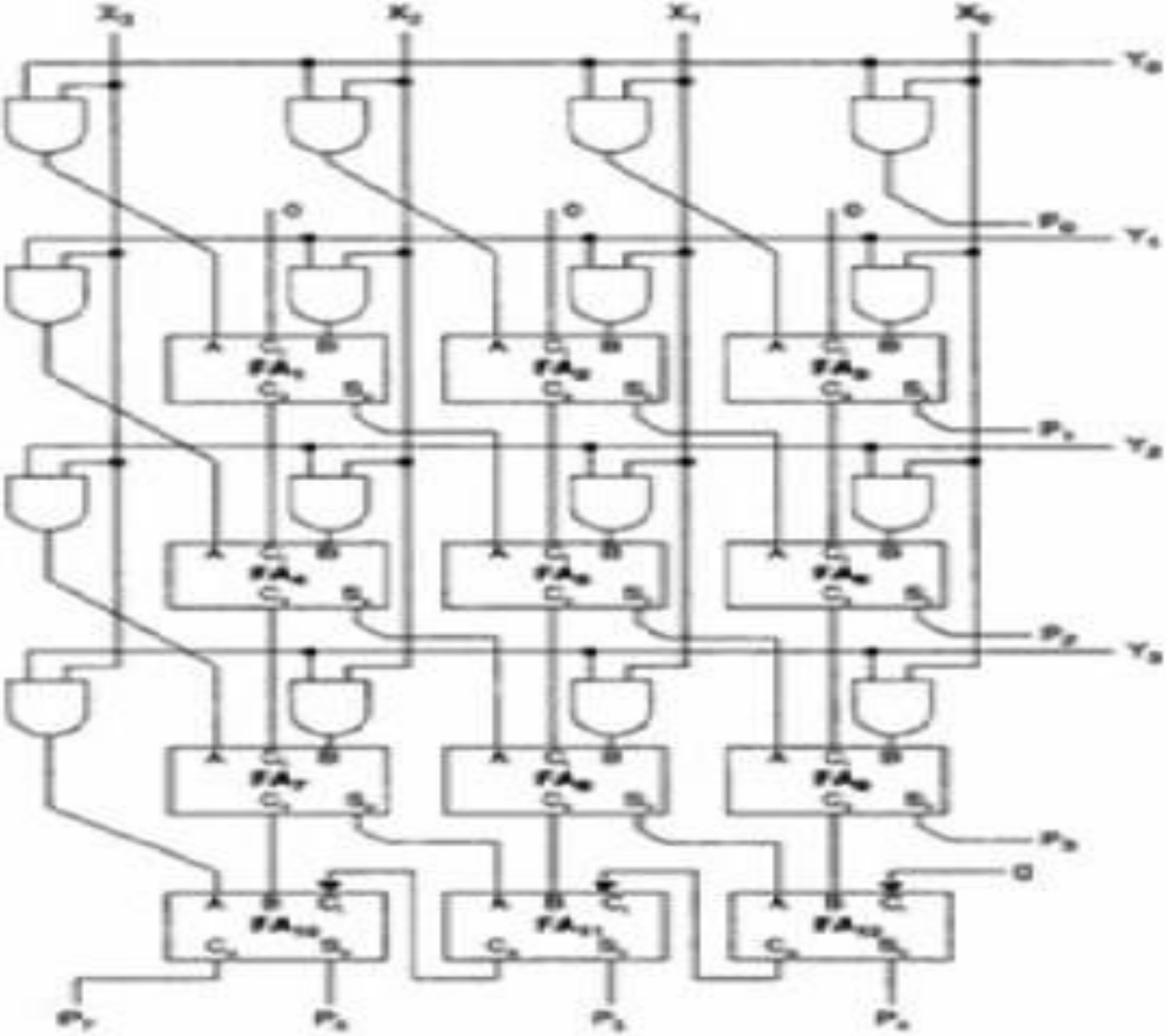
# Multiplication Algorithm

- Generation of partial products
- Adding up partial products
  - Sequentially (sequential shift and add)
  - Serially (combinational shift and add)
  - In parallel
- Speed-up techniques
  - Reduce the number of partial products
  - Accelerate addition of partial products

# BRAUN Multiplier

- Consists of an array of AND gates and CSA arranged in an iterative structure and a final CPA
- Does not require any logic registers
- Restricted to multiplication of two unsigned numbers




X3
X2
X1
X0
Y0
Y1
Y2
Y3
0
0
0
P0
P1
P2
P3
A
Ci
B
S
Co
FA1
FA2
FA3
FA4
FA5
FA6
FA7
FA8
FA9
FA10
FA11
FA12
0
P7
P6
P5
P4

- Performs well for unsigned operands that are less than 16 bits in terms of speed, power and area
- Simple and regular structure as compared to the other multipliers
- No. of components required increases quadratically with the no. of bits

- The delay of the multiplier is given by

$$t_{mult} = (N-1)t_{carry} + (N-1)t_{and} + t_{merge}$$

# Booth's Multiplier

- Multiplies two signed binary numbers in two's compliment notation.
- Uses shifting than adding to increase their speed.

- Recall grade school trick

  When multiplying by 9:
    - Multiply by 10 (easy, just shift digits left)
    - Subtract once
- Booth's algorithm applies same principle

# When using Booth's Algorithm:

- You will need twice as many bits in your **product** as you have in your original two **operands**.

- The **leftmost bit** of your operands (both your multiplicand and multiplier) is a SIGN bit, and cannot be used as part of the value.

- Decide which operand will be the **multiplier** and which will be the **multiplicand**

- Convert both operands to **two's complement** representation using X no. of bits

- X must be at least one more bit than is required for the binary representation of the numerically larger operand

- Begin with a product that consists of the multiplier with an additional 'X no. of leading zero bits'.

- For our example, let's multiply **(-5) x 2**

  The numerically larger operand (5) would require 3 bits to represent in binary (101). So we must use AT LEAST 4 bits to represent the operands, to allow for the sign bit.

- Let's use 5-bit 2's complement

-5 is 11011 (multiplier)
2 is 00010 (multiplicand)

- The multiplier is:

  11011(-5 in 2’s compliment)

- Add 5 leading zeros to the **multiplier** to get the **beginning product**:

  00000 11011

**STEP 1**

- Use the **LSB** and the **previous LSB** to determine the arithmetic action.

  If it is the FIRST pass, use 0 as the previous LSB.

- Possible arithmetic actions:

  00 □ no arithmetic operation
  01 □ add multiplicand to left half of product
  10 □ subtract multiplicand from left half of product
  11 □ no arithmetic operation

## STEP 2

- Perform an **arithmetic right shift** (ASR) on the entire product.

- NOTE: For X-bit operands, Booth's algorithm requires X passes.

- Initial Product and previous LSB

```
    00000 11011 0
```

- (Note: Since this is the first pass, we use 0 for the previous LSB)

- Pass 1, Step 1: Examine the last 2 bits

```
    00000 11011 0
```

- The last two bits are 10, so we need to subtract the **multiplicand** from left half of product

Pass 1, Step 1: Arithmetic action

```
(1)  00000  (left half of product)
    -00010      (mulitplicand)
     11110    (uses a phantom borrow)
```

Place result into **left half** of product

```
        11110 11011
```

Step 2: ASR (arithmetic shift right)

Before ASR

```
        11110 11011
```

After ASR

```
        11111 01101
```

(left-most bit was 1, so a 1 was shifted in on the left)

Pass 1 is complete.

- Current Product and previous LSB
  11111 01101 1

- Pass 2, Step 1: Examine the last 2 bits
  11111 01101 1
- The last two bits are 11, so we do NOT need to perform an arithmetic action just proceed to step 2.
- Step 2: ASR (arithmetic shift right)
  Before ASR
  11111 01101 1
  After ASR
  11111 10110 1
  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 2 is complete.

- Current Product and previous LSB

  `11111 10110 1`

- Pass 3, Step 1: Examine the last 2 bits

  `11111 10110 1`

  The last two bits are 01, so we need to add the **multiplicand** to the left half of the product

Arithmetic action

```
(1)  11111  (left half of product)
    +00010      (mulitplicand)
     00001   (drop the leftmost carry)
```

Place result into **left half** of product

```
   00001 10110 1
```

- Step 2: ASR (arithmetic shift right)

  Before ASR

  `00001 10110 1`

  After ASR

  `00000 11011 0`

  (left-most bit was 0, so a 0 was shifted in on the left)

- Pass 3 is complete.

## PASS 4

- Current Product and previous LSB

  `00000 11011 0`

- Step 1: Examine the last 2 bits

  `00000 11011 0`

- The last two bits are 10, so we need to subtract the **multiplicand** from the left half of the product

- Step 1: Arithmetic action

```
(1)  00000   (left half of product)
    -00010   (mulitplicand)
     11110    (uses a phantom borrow)
```

- Place result into **left half** of product

```
11110 11011 0
```

- Step 2: ASR (arithmetic shift right)

  Before ASR

```
        11110 11011 0
```

  After ASR

```
        11111 01101 1
```

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 4 is complete.

- Current Product and previous LSB

  11111 01101 1

- Pass 5, Step 1: Examine the last 2 bits

  11111 01101 1

- The last two bits are 11, so we do NOT need to perform an arithmetic action just proceed to step 2.
- Step 2: ASR (arithmetic shift right)

  Before ASR

  11111 01101 1

  After ASR

  11111 10110 1

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 5 is complete.

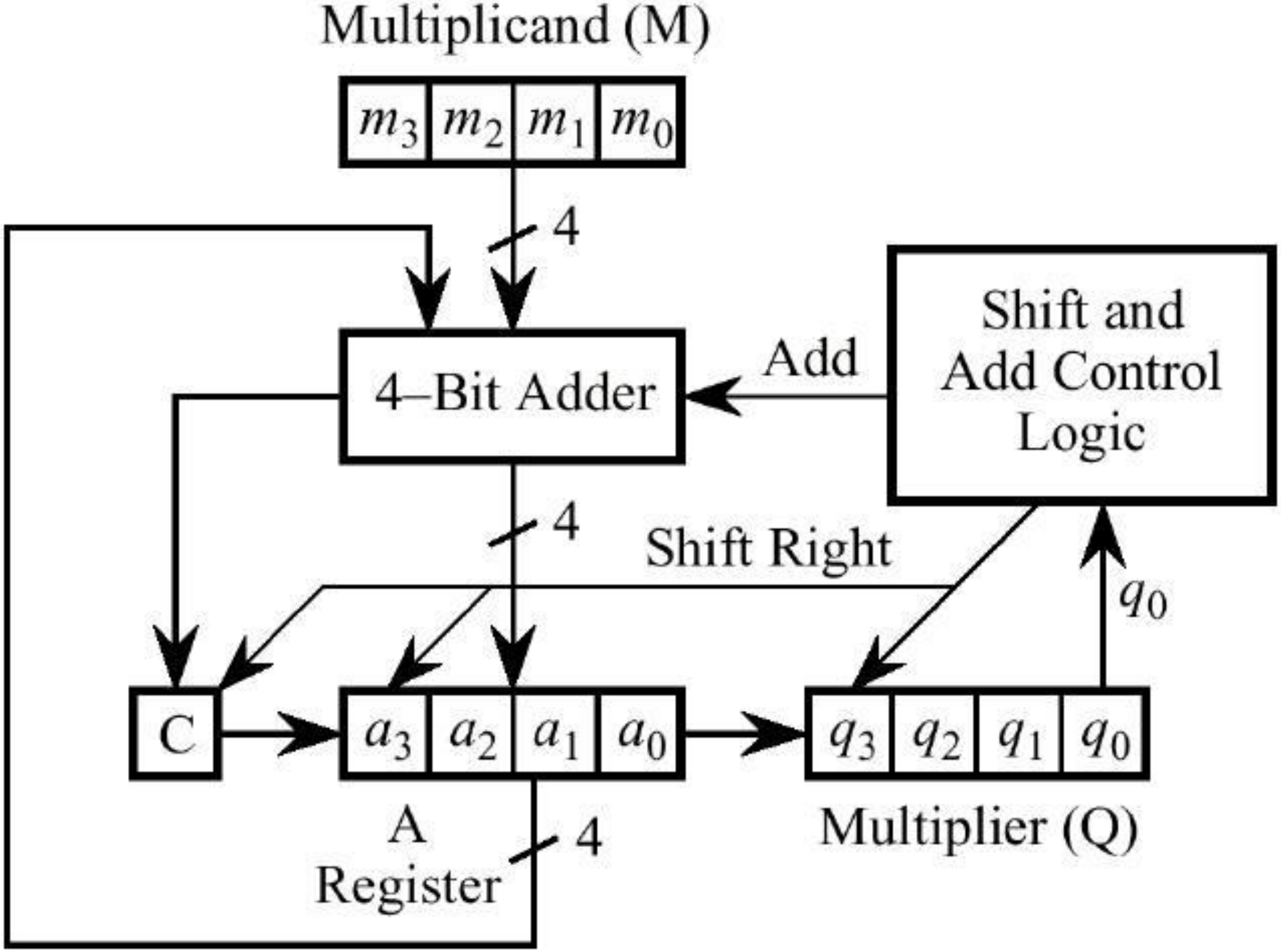
- Dropping the previous LSB, the resulting **final product** is:

  **11111 10110**

- To confirm we have the correct answer, convert the 2's complement **final product** back to decimal.
- Final product: **11111 10110**
- Decimal value: **-10**

  which is the CORRECT product of:

  **(-5) x 2**


Multiplicand (M)
$m_3$ $m_2$ $m_1$ $m_0$
4
4–Bit Adder
Add
Shift and Add Control Logic
4
Shift Right
$q_0$
C
$a_3$ $a_2$ $a_1$ $a_0$
$q_3$ $q_2$ $q_1$ $q_0$
A Register
4
Multiplier (Q)

# Modified Booth Multiplier (Booth Encoding)

- Can encode the digits by looking at three bits at a time
- Booth recoding table:

| i+1 | i | i-1 | add |
|---|---|---|---|
| 0 | 0 | 0 | 0*M |
| 0 | 0 | 1 | 1*M |
| 0 | 1 | 0 | 1*M |
| 0 | 1 | 1 | 2*M |
| 1 | 0 | 0 | –2*M |
| 1 | 0 | 1 | –1*M |
| 1 | 1 | 0 | –1*M |
| 1 | 1 | 1 | 0*M |

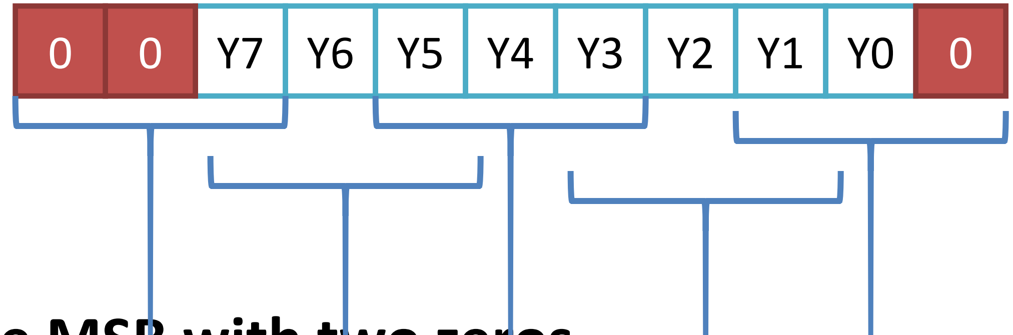
  – Must be able to add *multiplicand* times –2, -1, 0, 1 and 2

# Algorithm

1. Pad the LSB with one zero.
2. Pad the MSB with 2 zeros if n is even and 1 zero if n is odd.
3. Divide the multiplier into overlapping groups of 3-bits.
4. Determine partial product scale factor from modified booth 2 encoding table.
5. Compute the Multiplicand Multiples
6. Sum Partial Produ

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0 | 0 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0 | 0 |

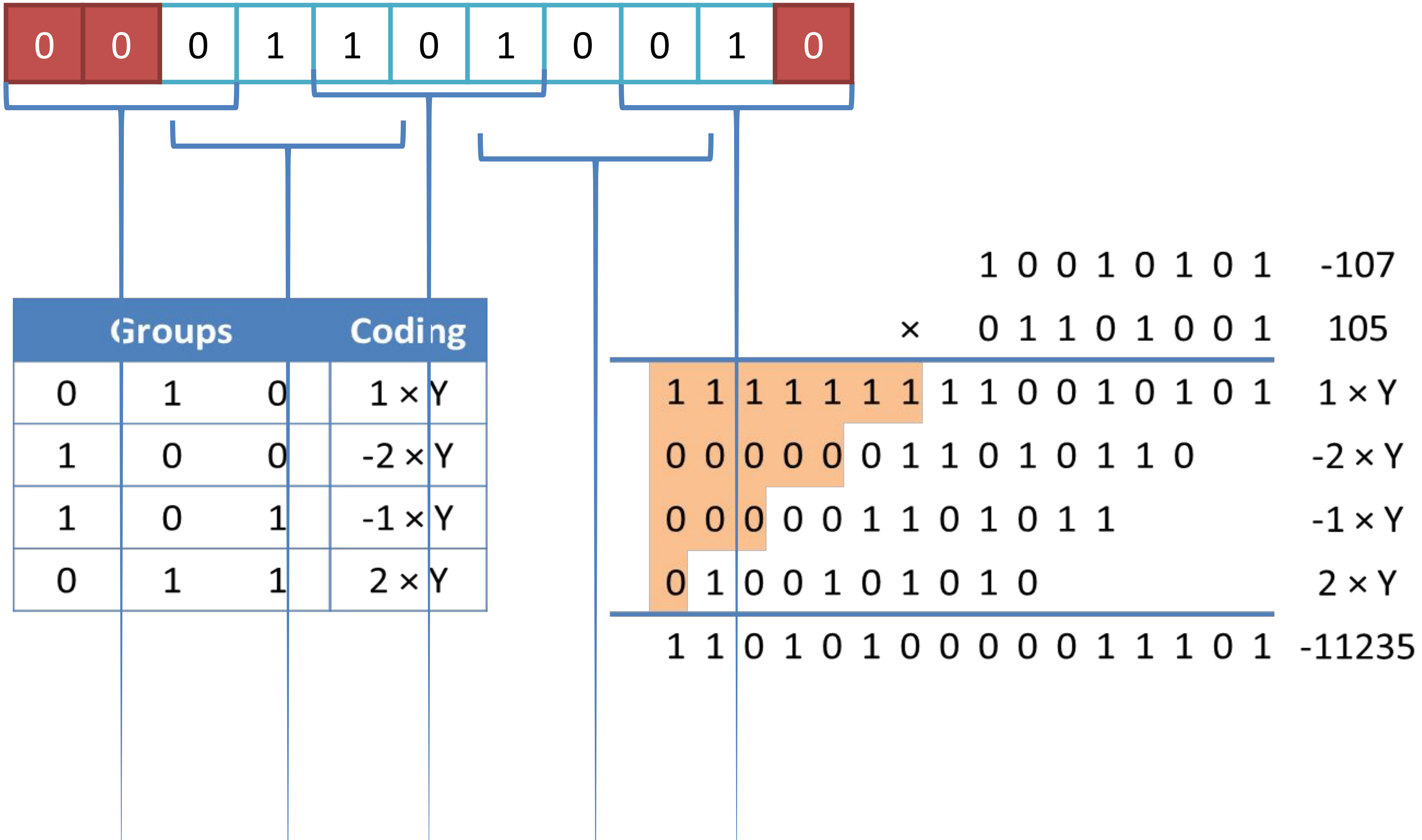


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0 | 0 |

**ad LSB with 1 zero**

**is even then pad the MSB with two zeros**

# Example

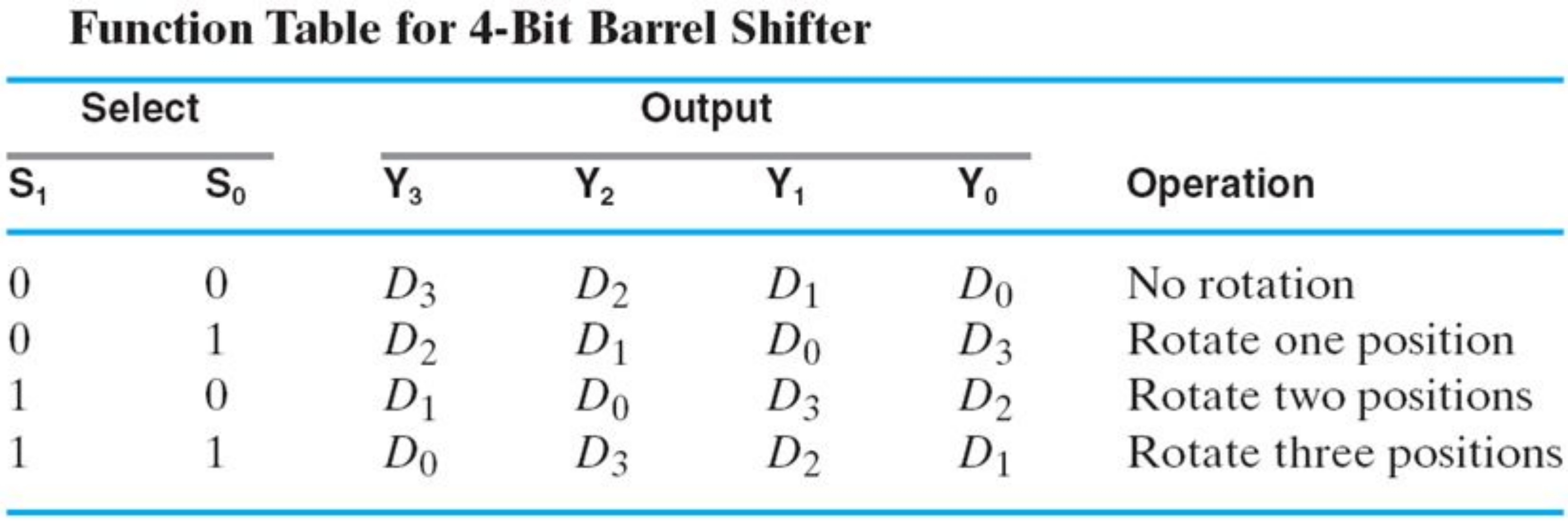


| Groups | | | Coding |
|---|---|---|---|
| 0 | 1 | 0 | 1 × Y |
| 1 | 0 | 0 | -2 × Y |
| 1 | 0 | 1 | -1 × Y |
| 0 | 1 | 1 | 2 × Y |

```
                1 0 0 1 0 1 0 1    -107
            ×   0 1 1 0 1 0 0 1     105
1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 1     1 × Y
0 0 0 0 0 0 1 1 0 1 0 1 1 0        -2 × Y
0 0 0 0 0 1 1 0 1 0 1 1            -1 × Y
0 1 0 0 1 0 1 0 1 0                 2 × Y
1 1 0 1 0 1 0 0 0 0 0 1 1 1 0 1  -11235
```

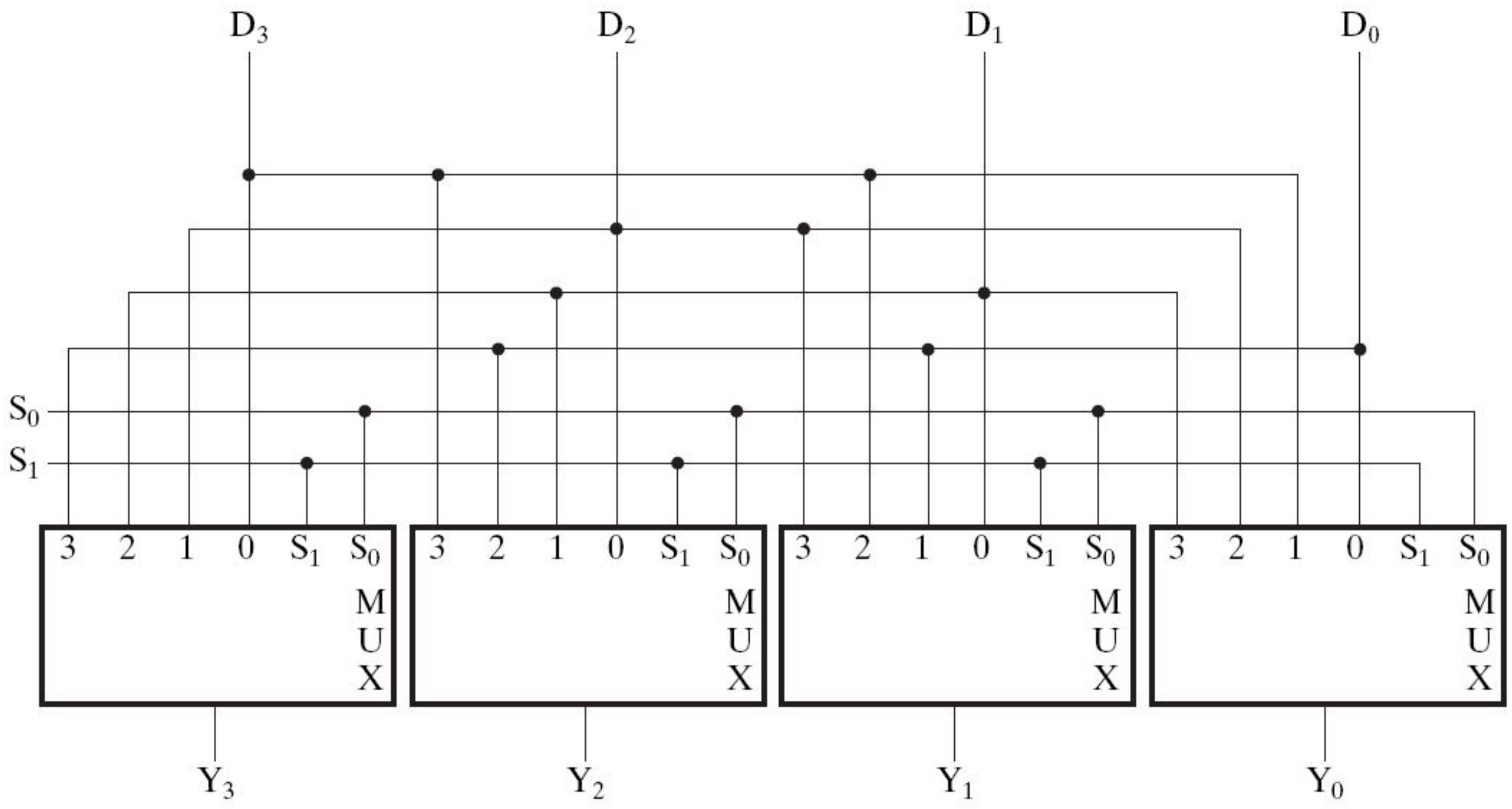
# Barrel Shifter

- A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle.
- It can be implemented with multiplexers (MUX)



**Function Table for 4-Bit Barrel Shifter**

| Select | | Output | | | | |
|---|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | Operation |
| 0 | 0 | $D_3$ | $D_2$ | $D_1$ | $D_0$ | No rotation |
| 0 | 1 | $D_2$ | $D_1$ | $D_0$ | $D_3$ | Rotate one position |
| 1 | 0 | $D_1$ | $D_0$ | $D_3$ | $D_2$ | Rotate two positions |
| 1 | 1 | $D_0$ | $D_3$ | $D_2$ | $D_1$ | Rotate three positions |



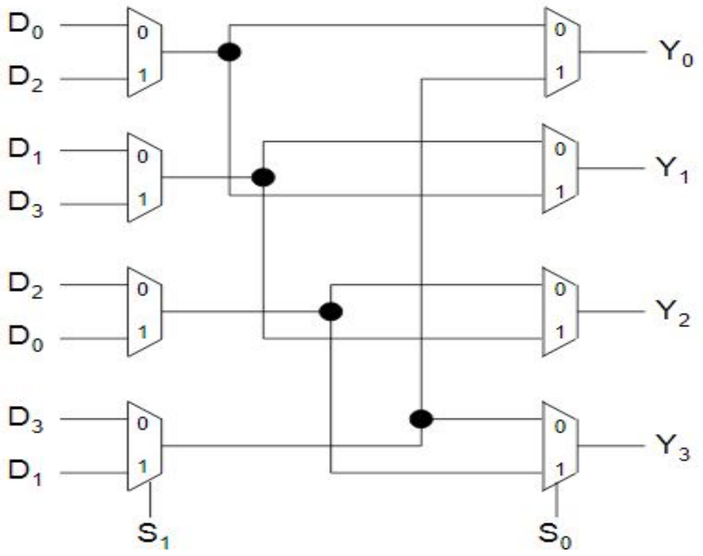
*i* positions of left rotation is the same as $2^n$-*i* bits of right rotation implementing a barrel shifter would be to use N (where N is the number of input bits) parallel N-to-1 multiplexers, one multiplexer that multiplexes all inputs into one of the outputs.

This would create an equivalent N*N to N multiplexer, which would use significant hardware resources (**multiplexer input grows in O($N^2$).**

A more efficient implementation is possible by creating a hierarchy of multiplexers.

- 32-bit barrel shifter: can use 32 32-to-1multiplexers
- However, large fan-in undesirable. So, use layers of multiplexers

Example:
Use 2 layers of 4
2-to-1 multiplexers
for 4-bit barrel
shifter



For an 8 bit rotate component the multiplexers would be constructed as follows:

- the top level multiplexers rotate the data by 4 bits
- the second level rotates the data by 2 bits
- the third level rotates the data by 1 bit.

The number of multiplexers required is *n**log2(*n*), for an *n* bit word.
Four common word sizes and the number of multiplexers needed are:
64-bit — 64 * log2(64) = 64 * 6 = 384
32-bit — 32 * log2(32) = 32 * 5 = 160
16-bit — 16 * log2(16) = 16 * 4 = 64
8-bit — 8 * log2(8) = 8 * 3 = 24

```
module barrel_shifter(d,out,q,c); / Main module of 8-Bit Barrel shifter
  input [7:0]d;
  output [7:0]out,q;
  input[2:0]c;
  mux m1(q[0],d,c);
  mux m2(q[1],{d[0],d[7:1]},c);
  mux m3(q[2],{d[1:0],d[7:2]},c);
  mux m4(q[3],{d[2:0],d[7:3]},c);
  mux m5(q[4],{d[3:0],d[7:4]},c);
  mux m6(q[5],{d[4:0],d[7:5]},c);
  mux m7(q[6],{d[5:0],d[7:6]},c);
  mux m8(q[7],{d[6:0],d[7:7]},c);
  assign out=q;
endmodule
```

# Baugh-Wooley Multiplier

- Efficient way to handle sign bits
- Regular Multipliers, suited for 2's-compliment numbers
- Signed number representation

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

eg: $(1110)_2$

= (-2)

$$A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

$$\begin{aligned}
P &= A \times B \\
&= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) \times \left(-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right) \\
&= a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\
&\quad -2^{n-1}\sum_{i=0}^{n-2} a_i b_{n-1} 2^i - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1} b_j 2^j
\end{aligned}$$

# The Baugh-Wooley Method and Its Modified Form

$-a_4x_0 = a_4(1 - x_0) - a_4$

$= a_4x_0{}' - a_4$

$-a_4 \quad a_4x_0{}'$

$a_4$

In next column

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | | | × | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ |
| | | | | | $a_4\bar{x}_0$ | $a_3x_0$ | $a_2x_0$ | $a_1x_0$ | $a_0x_0$ |
| | | | | $a_4\bar{x}_1$ | $a_3x_1$ | $a_2x_1$ | $a_1x_1$ | $a_0x_1$ | |
| | | | $a_4\bar{x}_2$ | $a_3x_2$ | $a_2x_2$ | $a_1x_2$ | $a_0x_2$ | | |
| | | $a_4\bar{x}_3$ | $a_3x_3$ | $a_2x_3$ | $a_1x_3$ | $a_0x_3$ | | | |
| | $a_4x_4$ | $\bar{a}_3x_4$ | $\bar{a}_2x_4$ | $\bar{a}_1x_4$ | $\bar{a}_0x_4$ | | | | |
| | $\bar{a}_4$ | | | | $a_4$ | | | | |
| 1 | $\bar{x}_4$ | | | | $x_4$ | | | | |
| $\bar{p}_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |

$-a_4x_0 = (1 - a_4x_0) - 1$

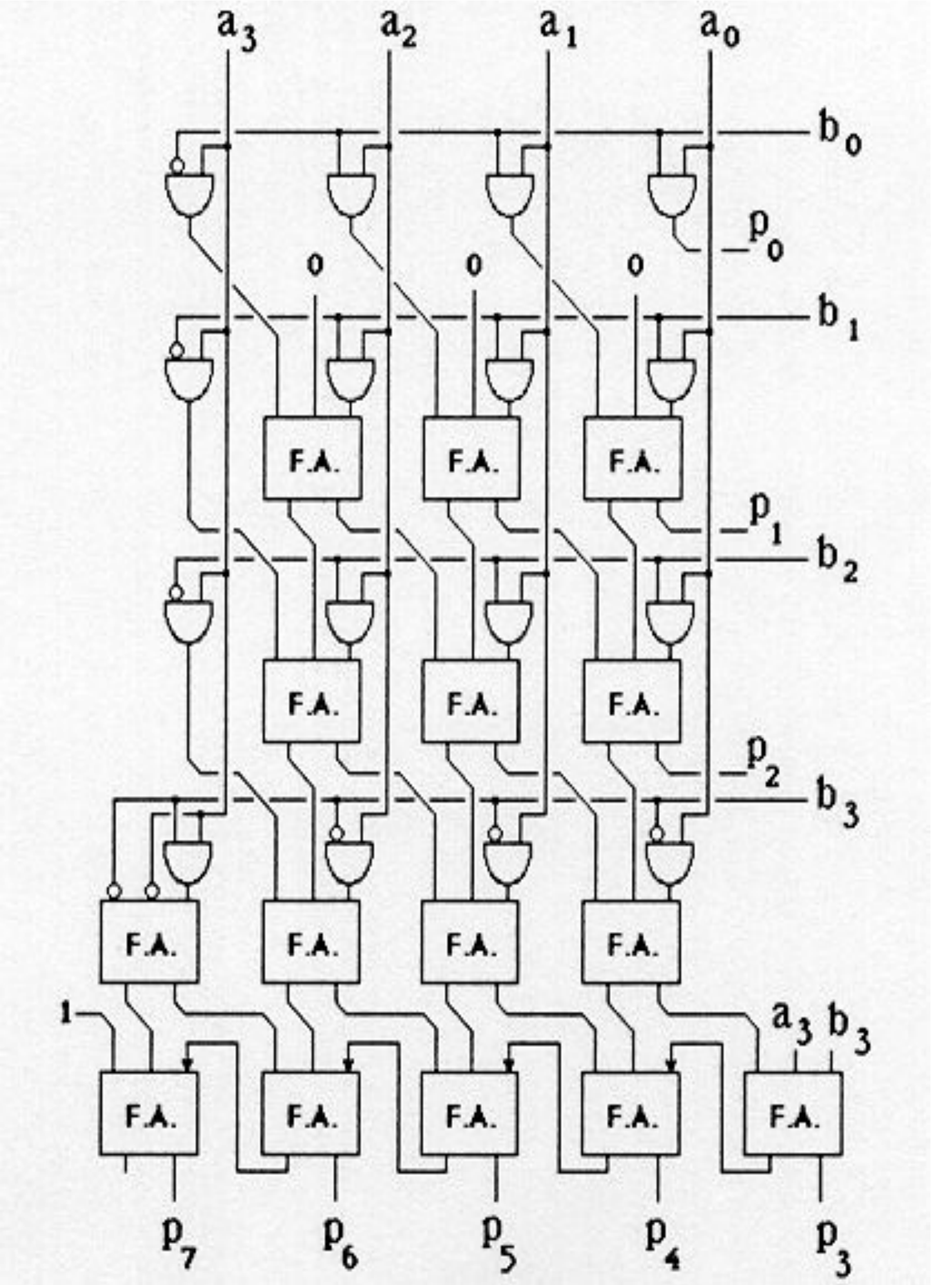
$= (a_4x_0)' - 1$

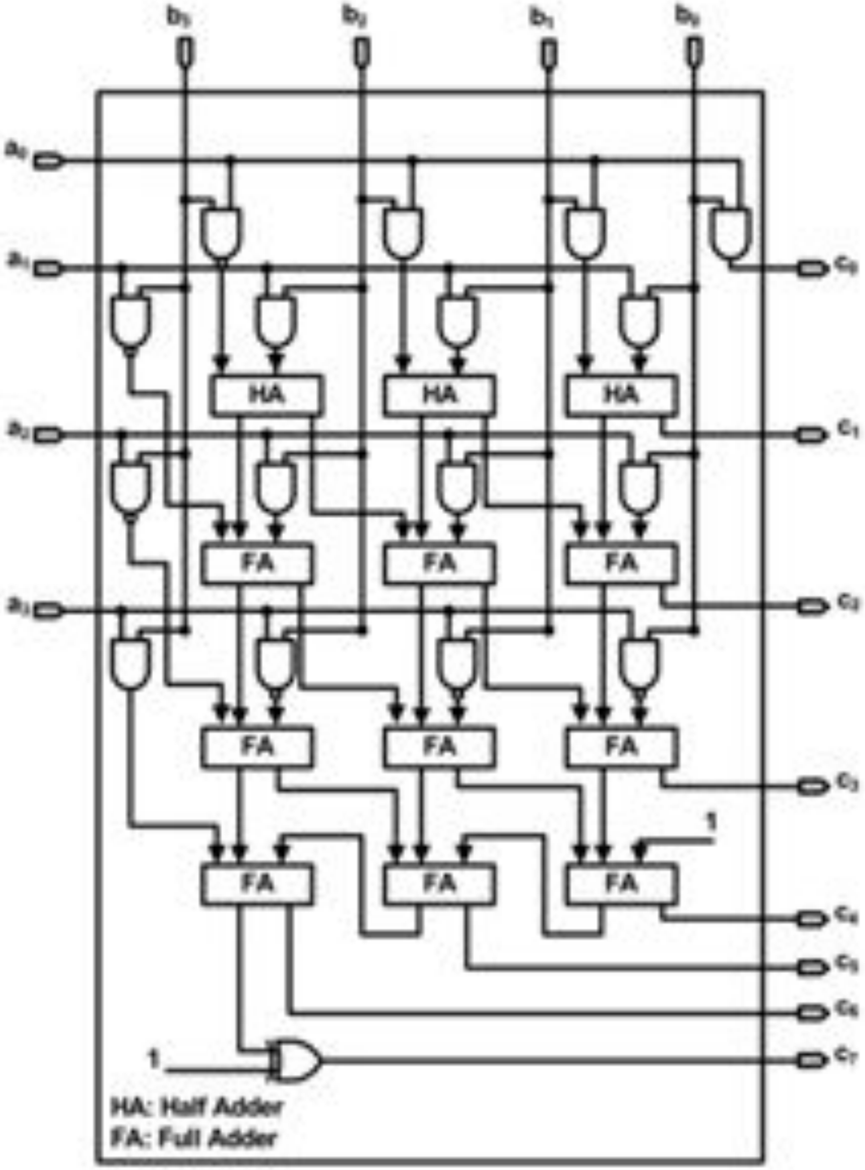
$-1 \quad (a_4x_0)'$

$1$

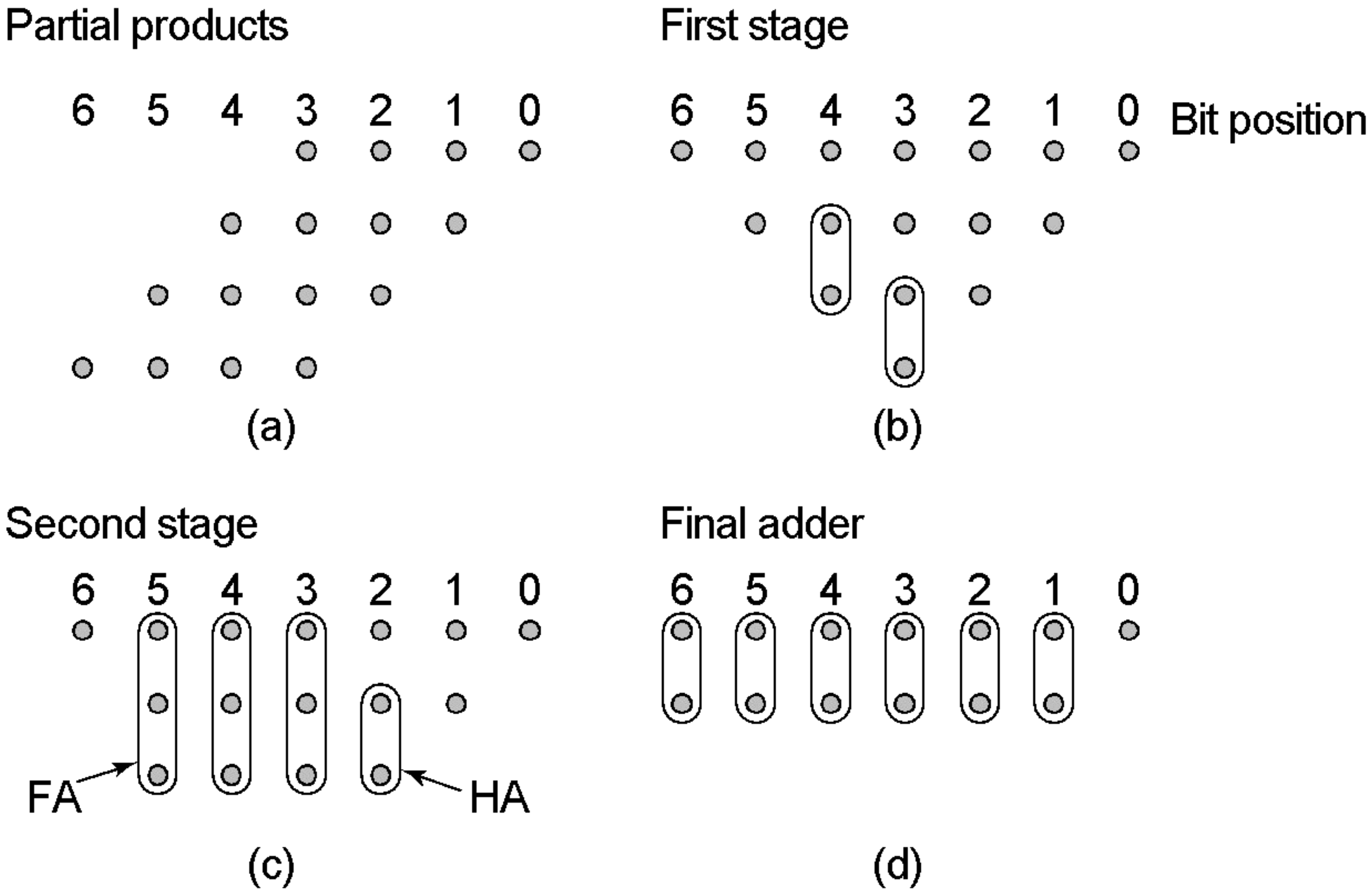
In next column

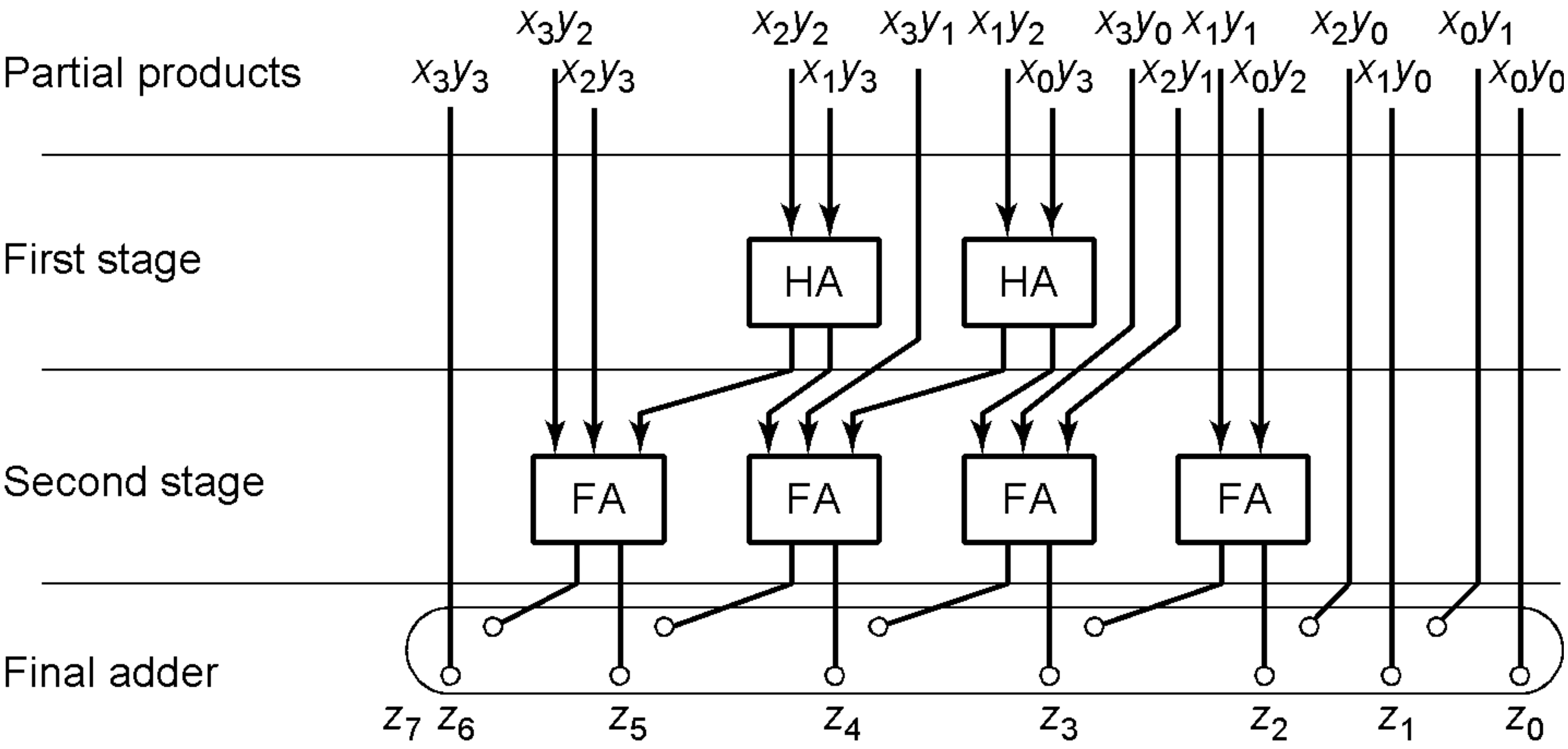
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | | | × | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ |
| | | | | | $\overline{a_4x_0}$ | $a_3x_0$ | $a_2x_0$ | $a_1x_0$ | $a_0x_0$ |
| | | | | $\overline{a_4x_1}$ | $a_3x_1$ | $a_2x_1$ | $a_1x_1$ | $a_0x_1$ | |
| | | | $\overline{a_4x_2}$ | $a_3x_2$ | $a_2x_2$ | $a_1x_2$ | $a_0x_2$ | | |
| | | $\overline{a_4x_3}$ | $a_3x_3$ | $a_2x_3$ | $a_1x_3$ | $a_0x_3$ | | | |
| | $a_4x_4$ | $\overline{a_3x_4}$ | $\overline{a_2x_4}$ | $\overline{a_1x_4}$ | $\overline{a_0x_4}$ | | | | |
| 1 | | | | 1 | | | | | |
| $\bar{p}_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |


a3
a2
a1
a0
b0
p0
0
0
0
b1
F.A.
p1
b2
p2
b3
1
a3
b3
p7
p6
p5
p4
p3


b3
b2
b1
b0
a0
a1
a2
a3
c0
c1
c2
c3
c4
c5
c6
c7
HA
FA
1
1
HA: Half Adder
FA: Full Adder

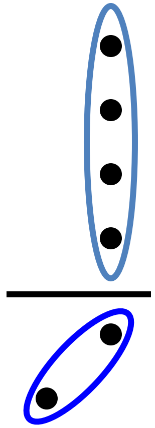
# Wallace-Tree Multiplier





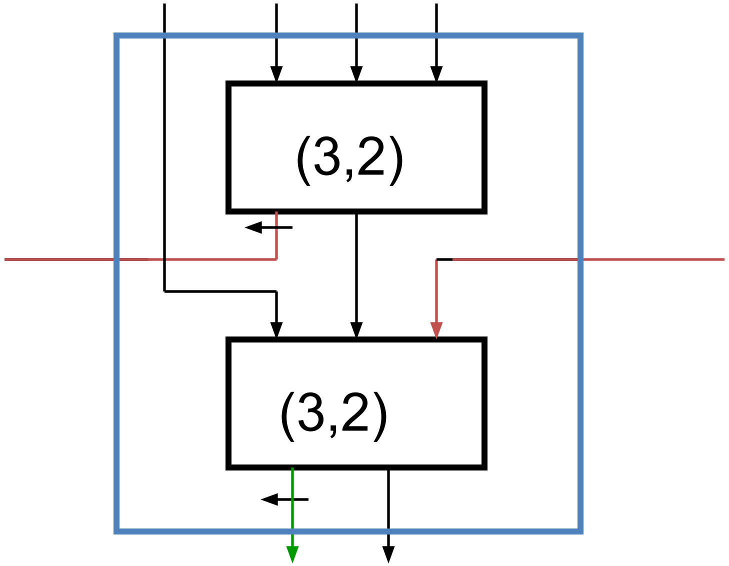
Full adder = (3,2) compressor

# (4,2) Counter



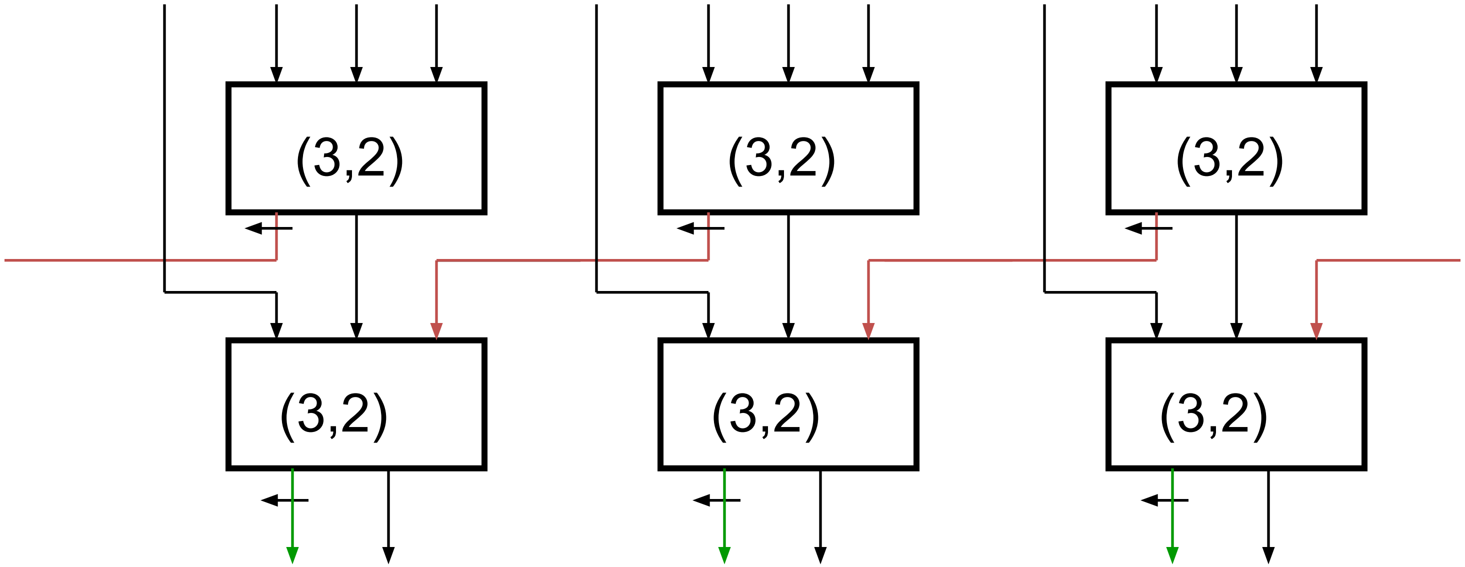
❑ Built out of two (3,2) counters (just FA’s!)

- all of the inputs (4 external plus one internal) have the same weight (i.e., are in the same bit position)
- the internal carry output is fed to the next higher weight position (indicated by the      )



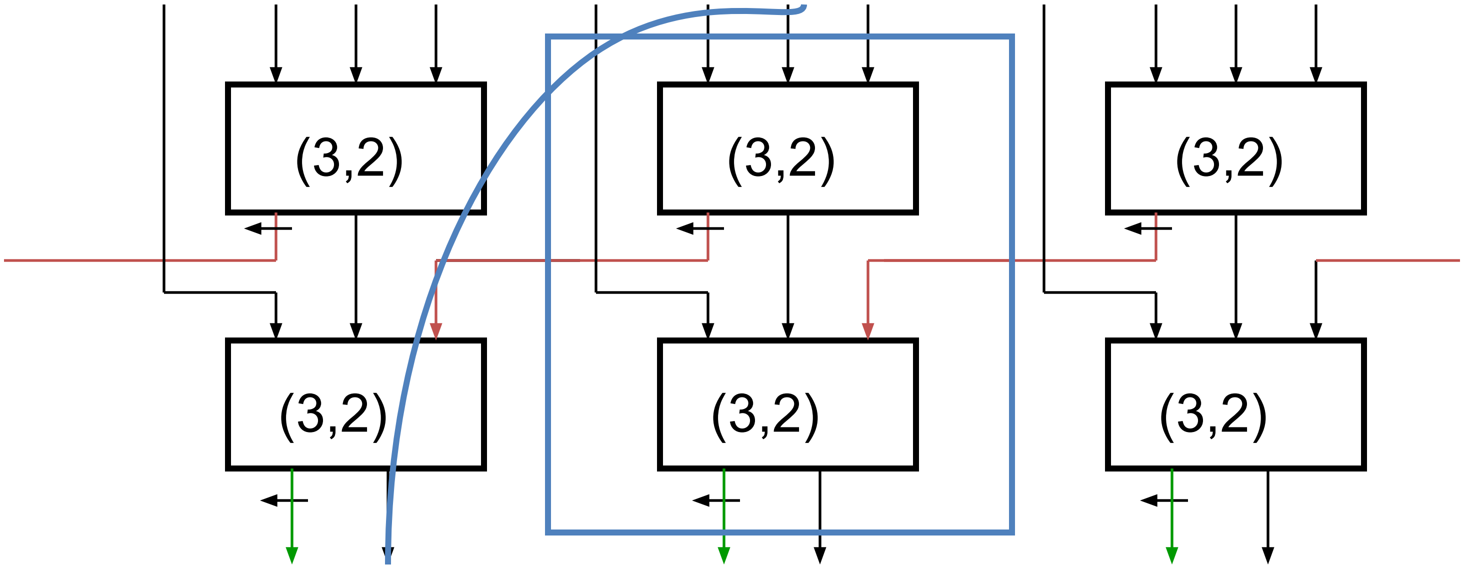
Note:  Two carry outs - one “internal” and one “external”

# Tiling (4,2) Counters



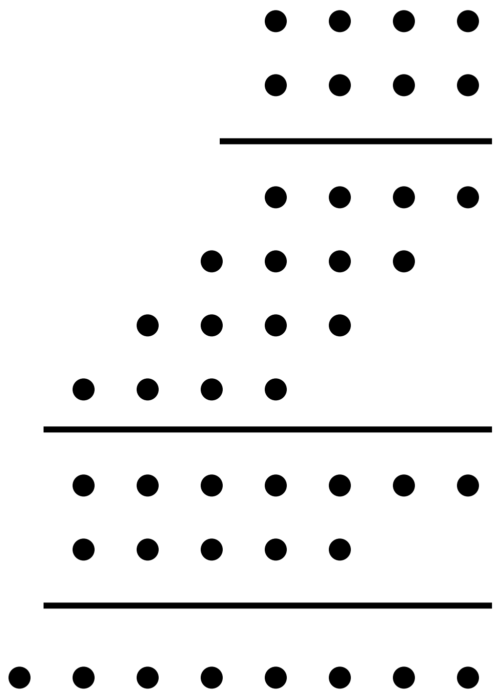
- Reduces columns four high to columns only two high
  - Tiles with neighboring (4,2) counters
  - Internal carry in at same “level” (i.e., bit position weight) as the internal carry out

# Tiling (4,2) Counters



# 4x4 Partial Product Array Reduction

- Fast 4x4 multiplication using (4,2) counters



multiplicand

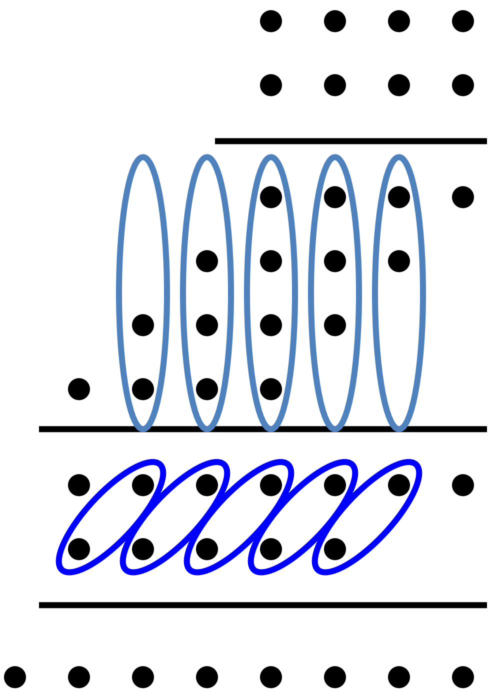
multiplier

partial product array

reduced pp array (to CPA)

double precision product

- Fast 4x4 multiplication using (4,2) counters



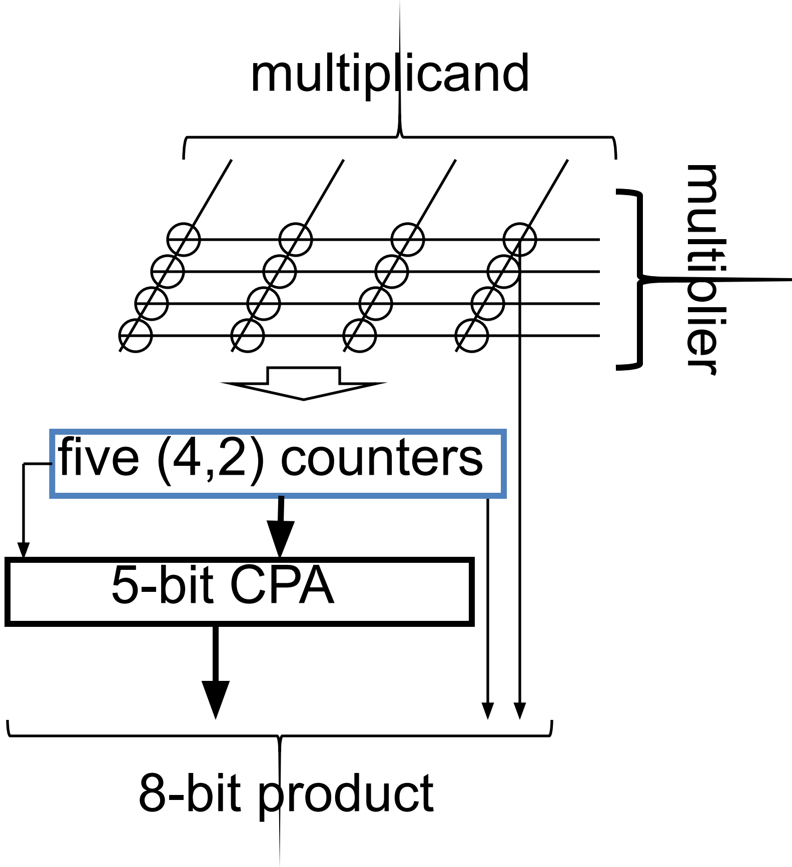
multiplicand

multiplier

partial product array

reduced pp array (to CPA)

double precision product



multiplicand

multiplier

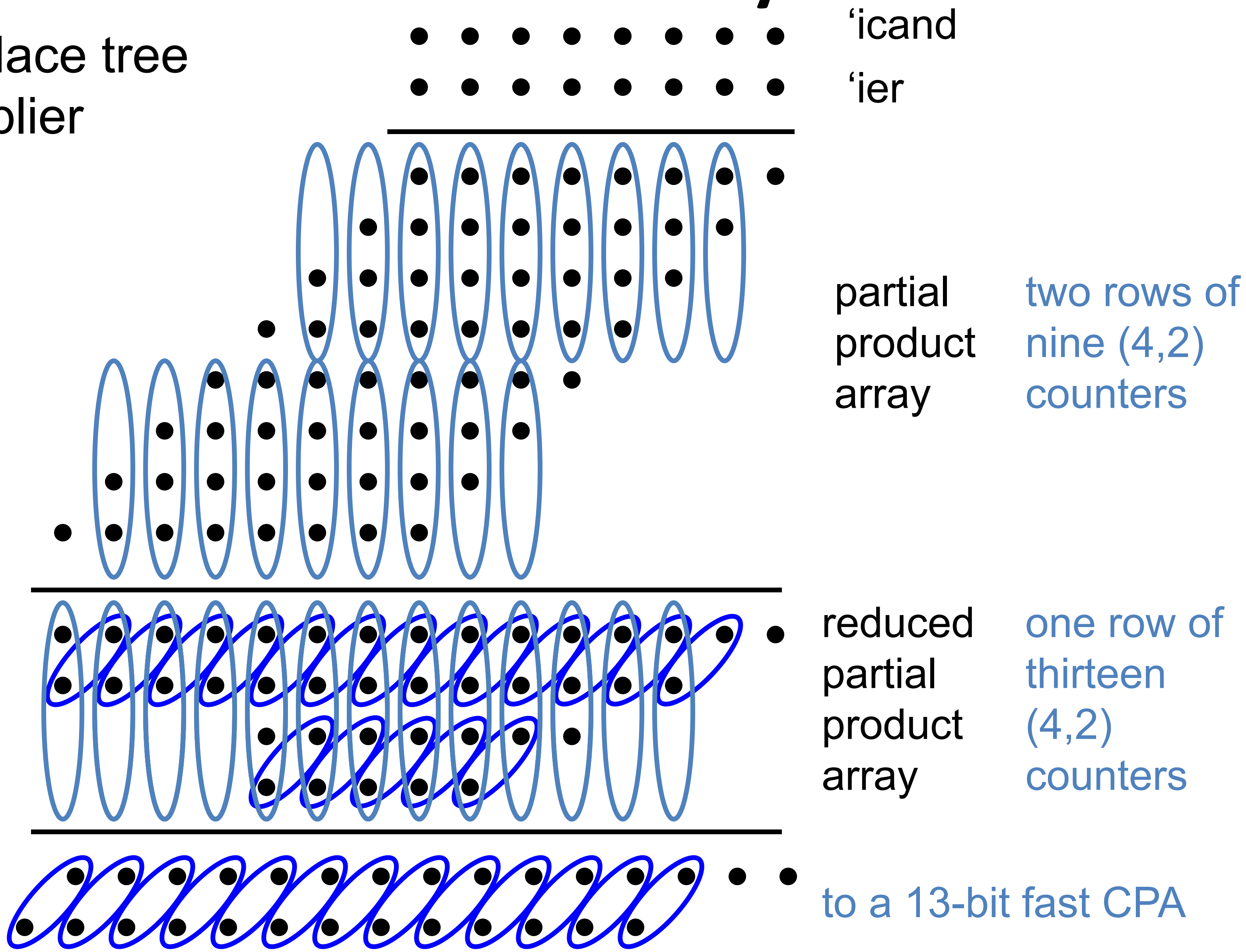
five (4,2) counters

5-bit CPA

8-bit product

# 8x8 Partial Product Array Reduction

- Wallace tree multiplier


‘icand
‘ier
partial product array
two rows of nine (4,2) counters
reduced partial product array
one row of thirteen (4,2) counters
to a 13-bit fast CPA