

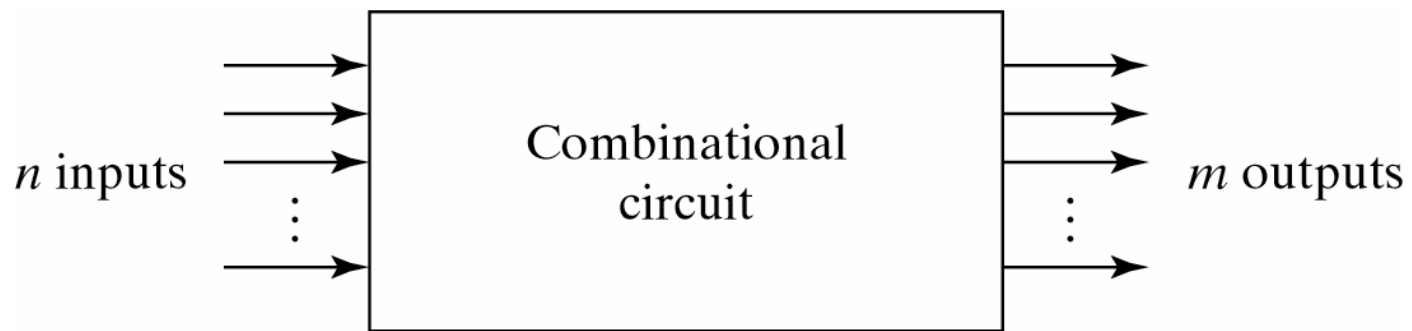
Combinational Logic I

Outline

- Combinational Logic
- Half Adder
- Full Adder
- Binary Addition
- Carry Lookahead
- Subtraction
- Multiplication

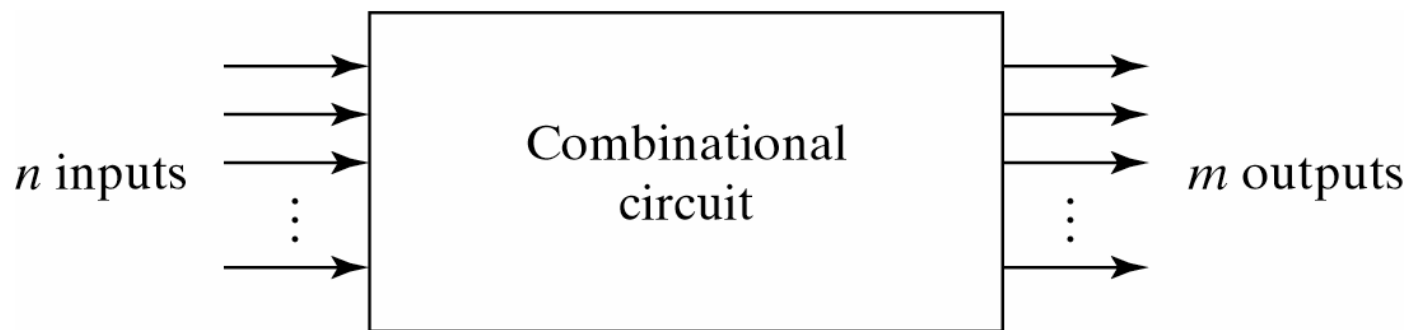
Combinational Logic

- A combinational logic circuit combines n logical inputs using AND, OR, NAND, ... logic gates. The m outputs are described by m logical expressions.
- In previous lectures we have examined Boolean algebra and the Karnaugh map method. These methods provide an efficient and systematic way of creating combinational circuits.



Combinational Logic

- In this lecture we shall examine combinational logic circuits designed to perform arithmetic logic.
- The arithmetic logic unit (ALU) is of central importance in digital processing.
- In the next lecture we shall examine the combinational logic used to decode and multiplex binary data.



Combinational Logic Design Procedure

1. Beginning with a design specification the number of inputs and outputs are determined and labelled using logical symbols.
2. The truth table relating the inputs to the outputs is created.
3. Using Boolean algebra or Karnaugh maps simplified logical functions relating each output to the inputs are obtained.
4. The simplified logical functions are used to specify logical circuits. These logical circuits form the combinational circuit.

Half Adder Specification

- The most basic arithmetic operation is the addition of two bits. The **half adder** is a combinational logic circuit designed to carry out this operation.
- The half adder needs to perform four basic operations:
 $0+0=0$, $0+1=1$, $1+0=1$ and $1+1=10$.



Half Adder Truth Table

- From the specification it can be seen that the half adder requires two inputs X and Y and two outputs S and C (the *sum* and the *carry*).
- The half adder truth table is given on the right.

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

Half Adder Logical Functions

- From the truth table the Karnaugh maps for the carry and sum are drawn.
- Simplification of the logical expression does not occur in this case, i.e. the output functions are those that would result if the truth table alone was used.
- The carry output is given as

$$C = X.Y$$

and the sum output is given as

$$S = X.\bar{Y} + \bar{X}.Y$$

Karnaugh map for Carry (C):

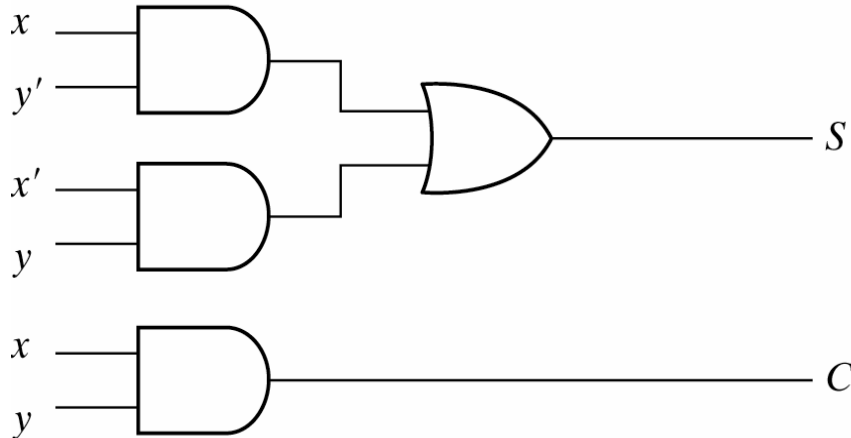
		Y	
		0	1
X	0	0	0
	1	0	1

Karnaugh map for Sum (S):

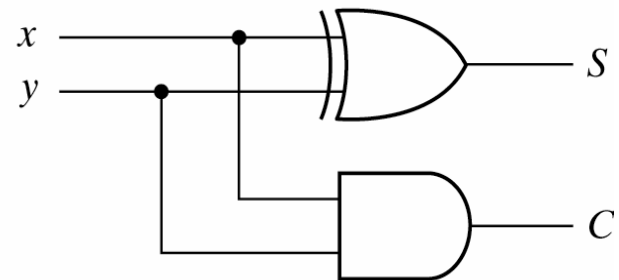
		Y	
		0	1
X	0	0	1
	1	1	0

Half Adder Implementation

- The carry has a straight forward implementation, an AND gate combining the inputs X and Y.
- The sum initially appears to require two AND gates and one OR gate for implementation. It can be implemented using an Exclusive OR (XOR) gate.

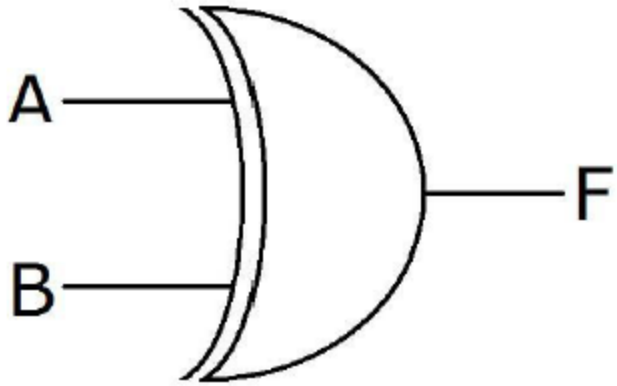


$$(a) \begin{aligned} S &= xy' + x'y \\ C &= xy \end{aligned}$$



$$(b) \begin{aligned} S &= x \oplus y \\ C &= xy \end{aligned}$$

XOR Gate (Exclusive OR)

Logic Gate	Graphical Symbol	Boolean Function	Truth Table															
XOR		$F=A\oplus B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

$$F = A\bar{B} + \bar{A}B$$

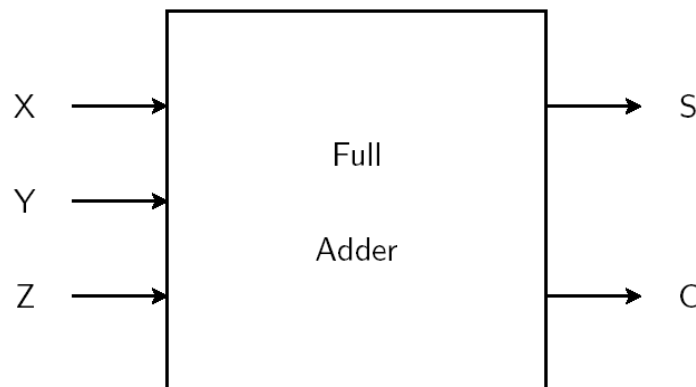
Multiple XOR Gate

- The general XOR function is true when an **odd** number of its arguments are **true**.
- XOR is especially useful for building adders and error detection/correction circuits.

x	y	z	$x \oplus y \oplus z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Full Adder Specification and Truth Table

- Building upon the half adder we now look at the addition of three bits X, Y and Z. The circuit which performs this operation is known as a **full adder**.
- The eight operations that the full adder performs are described by the truth table on the right.



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 Logical Functions

- From the truth table the carry and sum outputs have the Karnaugh maps given on the right.
- The carry output is given as

$$C = X.Y + Y.Z + X.Z$$

and the sum output is given as

$$S = \bar{X}.\bar{Y}.Z + \bar{X}.Y.\bar{Z} + X.Y.Z + X.\bar{Y}.\bar{Z}$$

C

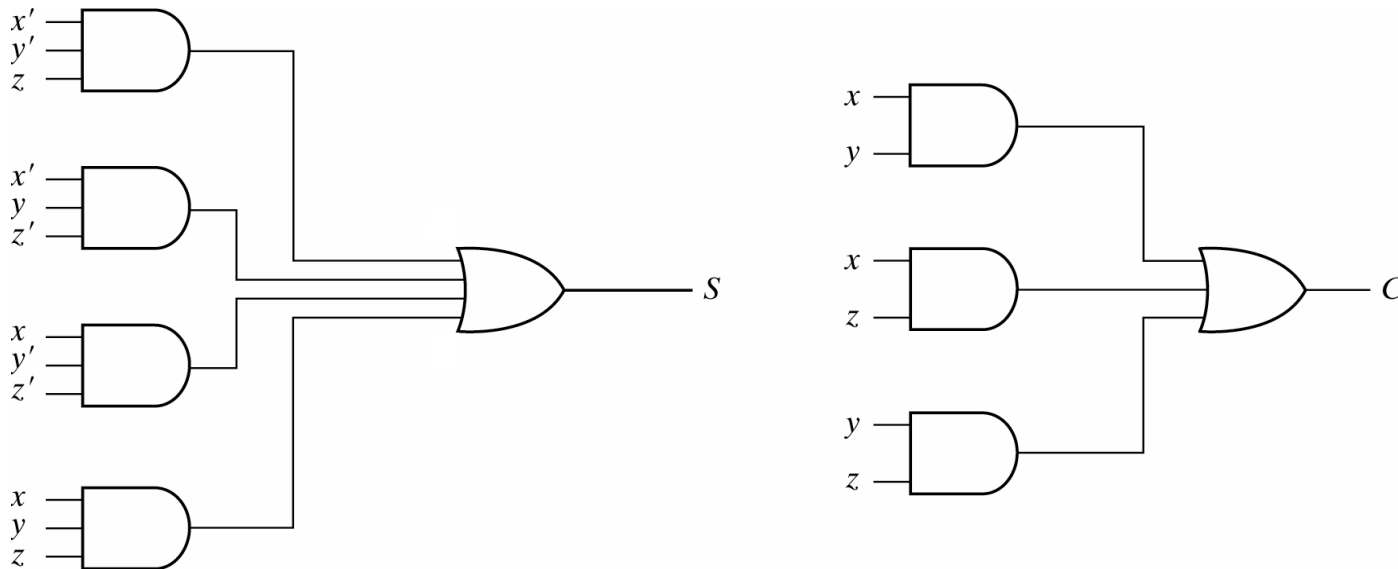
		Z	
		0	1
XY	00	0	0
	01	0	1
	11	1	1
	10	0	1

S

		Z	
		0	1
XY	00	0	1
	01	1	0
	11	0	1
	10	1	0

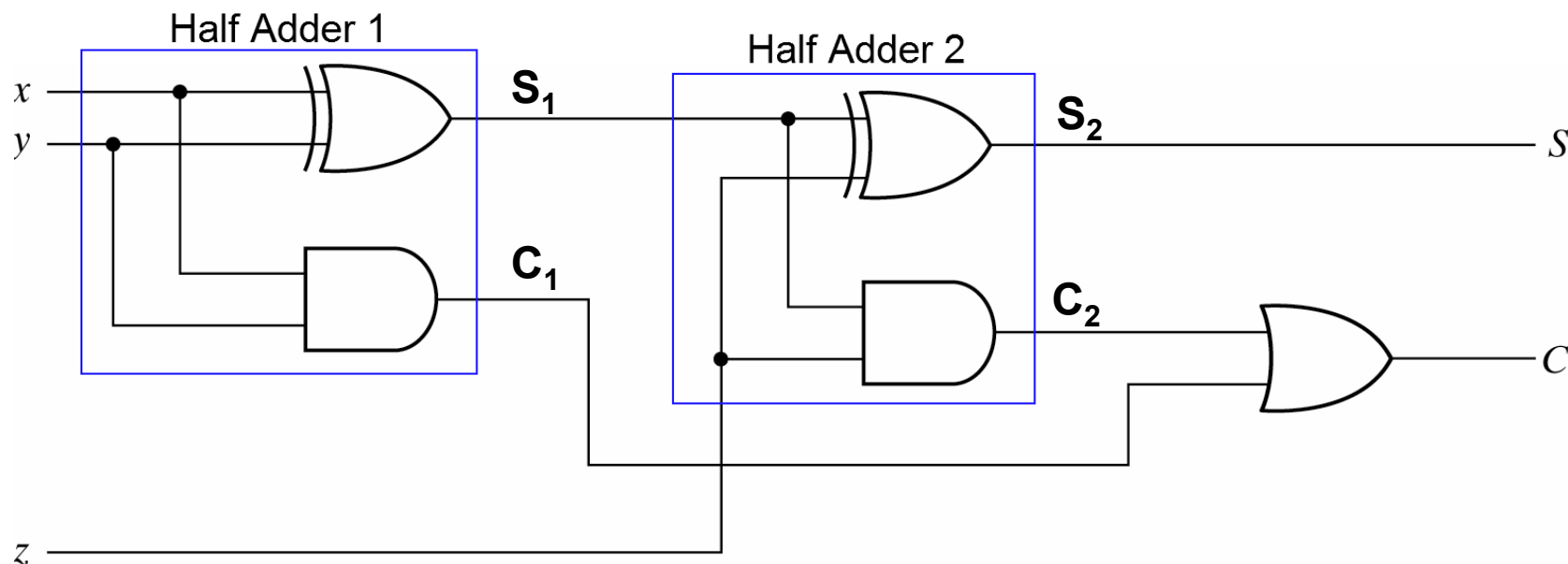
Full Adder Implementation

- Similar to half adder a full adder implementation is straight forward.
- For this implementation the carry requires one OR gate and three AND gates and the sum requires one OR gate and four AND gates.



Full Adder Implementation

- The full adder is usually implemented using two half adders (hence the name).



Full Adder Implementation

- We can verify that both implementations produce the same sum

$$S = S_2$$

$$S = S_1 \oplus Z$$

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

$$S = (X.\bar{Y} + \bar{X}.Y).\bar{Z} + \overline{(X.\bar{Y} + \bar{X}.Y)}.Z$$

$$S = (X.\bar{Y} + \bar{X}.Y).\bar{Z} + (\bar{X}.\bar{Y} + X.Y).Z$$

$$S = X.\bar{Y}.\bar{Z} + \bar{X}.Y.\bar{Z} + \bar{X}.\bar{Y}.Z + X.Y.Z$$

where S_1 and S_2 are the sum outputs of the first and second half adders.

Full Adder Implementation

- This implementation produces a slightly different expression for the carry

$$C = C_1 + C_2$$

$$C = C_1 + S_1.Z$$

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

$$C = X.Y + (X.\bar{Y} + \bar{X}.Y).Z$$

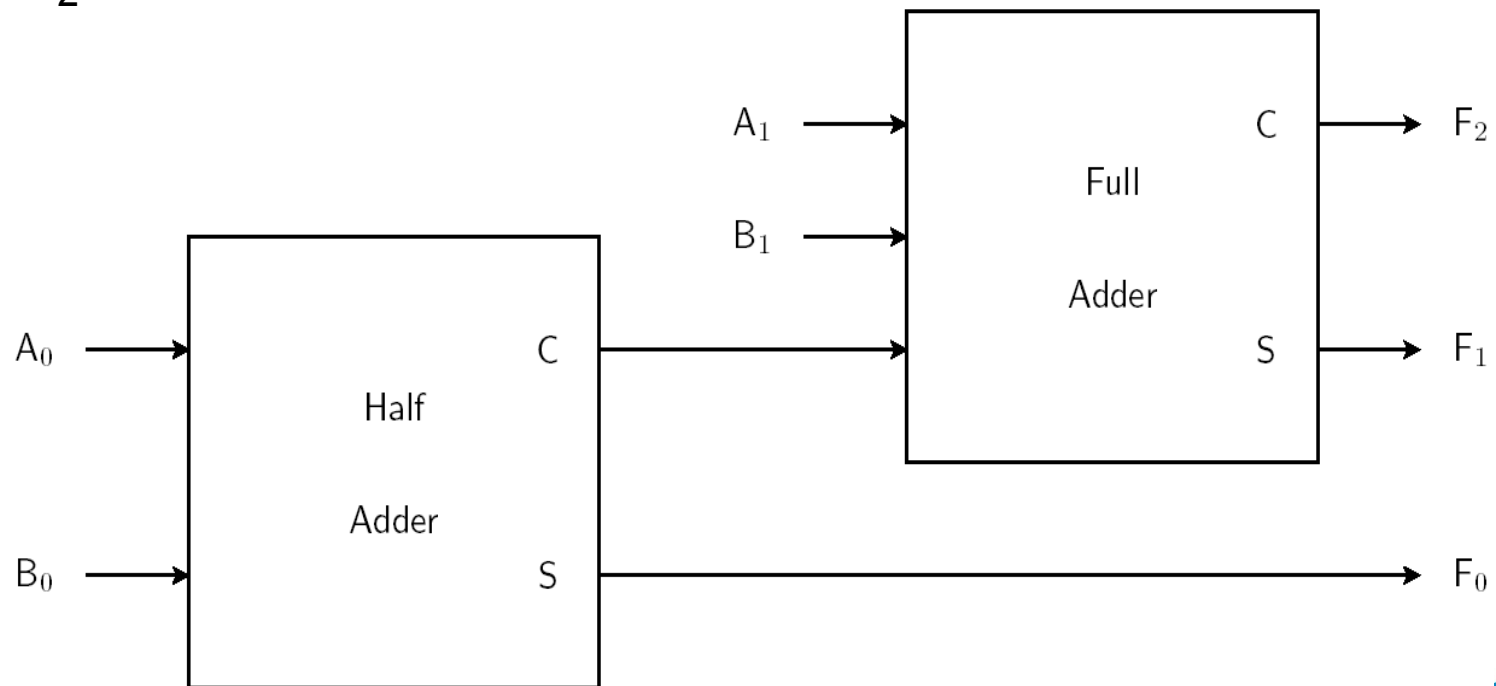
$$C = X.Y + X.\bar{Y}.Z + \bar{X}.Y.Z$$

where C_1 and C_2 are the carry outputs of the first and second half adders.

- Comparison with the truth table verifies the equivalence of both carry functions.

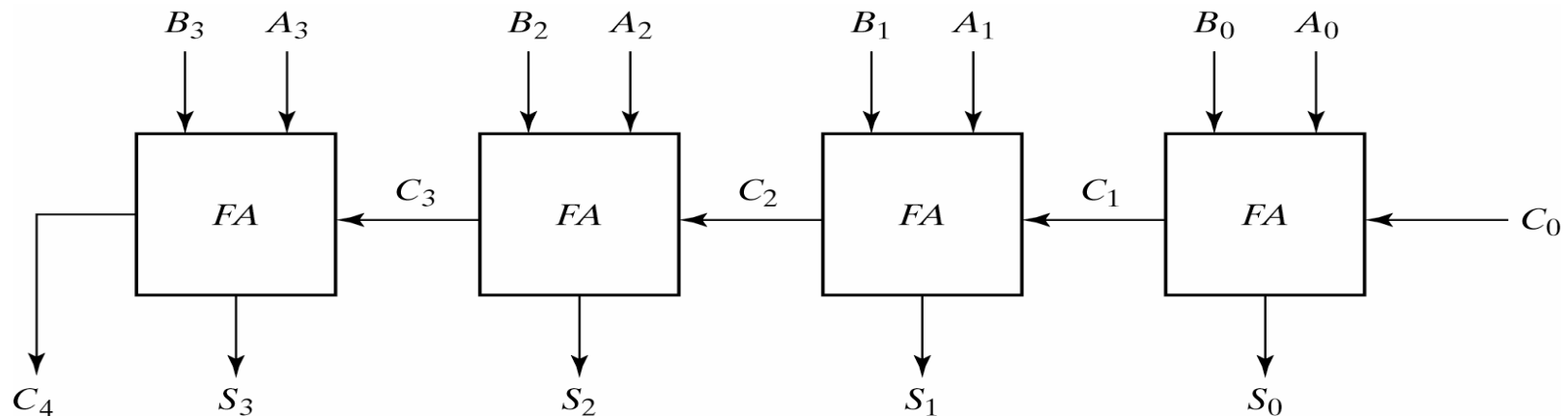
Binary Addition

- Now we look at the example of adding two 2-bit numbers A and B together.
- The first and second bits of A are labelled A_0 and A_1 , a similar notation is used for B .
- The first, second and third bits of the sum are labelled F_0 , F_1 and F_2 .



Binary Addition

- In the 2-bit adder each adder represented a bit position and the carry output from the first stage became an input of the second stage.
- The idea of cascading adders together extends to numbers with multiple bits, with each adder representing a bit position and each output carry being passed to the next stage.
- Here we have a 4-bit ripple carry adder, typically the first carry input C_0 is set to zero.



Carry Propagation

- In the ripple carry adder each full adder only generates outputs when the carry from the previous stage is available. As result the most significant bit of the sum is computed only when the carry has propagated through all full adder stages.
- The total delay is related to the number of bits being added: $2n$ gate delays, each of the order of 10 ns.
- All other arithmetic operations are implemented by successive additions so the time consumed in this step is critical.
- Options to speed up:
 - Faster gates
 - Parallel processing with increased circuit complexity

Carry Propagation

- To work out the truth table and directly implement a 4-bit parallel adder would require $2^9 = 512$ entries in the Karnaugh Map because there would be 9 inputs to the circuit.
- We see the advantage of cascading a standard function to find a simple and straightforward implementation
- Another approach would be to compute the carry outputs in parallel, but this approach also becomes impractical for multiple bit adders as the size of the circuitry grows.
- Even for the third stage the carry expression becomes unwieldy.

$$\begin{aligned}C_2 &= A_2B_2 + A_2C_1 + B_2C_1 \\&= A_2B_2 + A_2(A_1B_1 + A_1C_0 + B_1C_0) + B_2(A_1B_1 + A_1C_0 + B_1C_0) \\&= A_2B_2 + A_1A_2B_1 + B_1B_2A_1 + A_1A_2C_0 + A_2B_1C_0 + A_1B_2C_0 + B_1B_2C_0\end{aligned}$$

Carry Lookahead Generator

- An alternative method is the carry lookahead generator which uses faster additional circuitry to generate carry.
- To examine this circuit two variables associated with the i^{th} bit are introduced, the carry propagate P_i and the carry generate G_i .

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

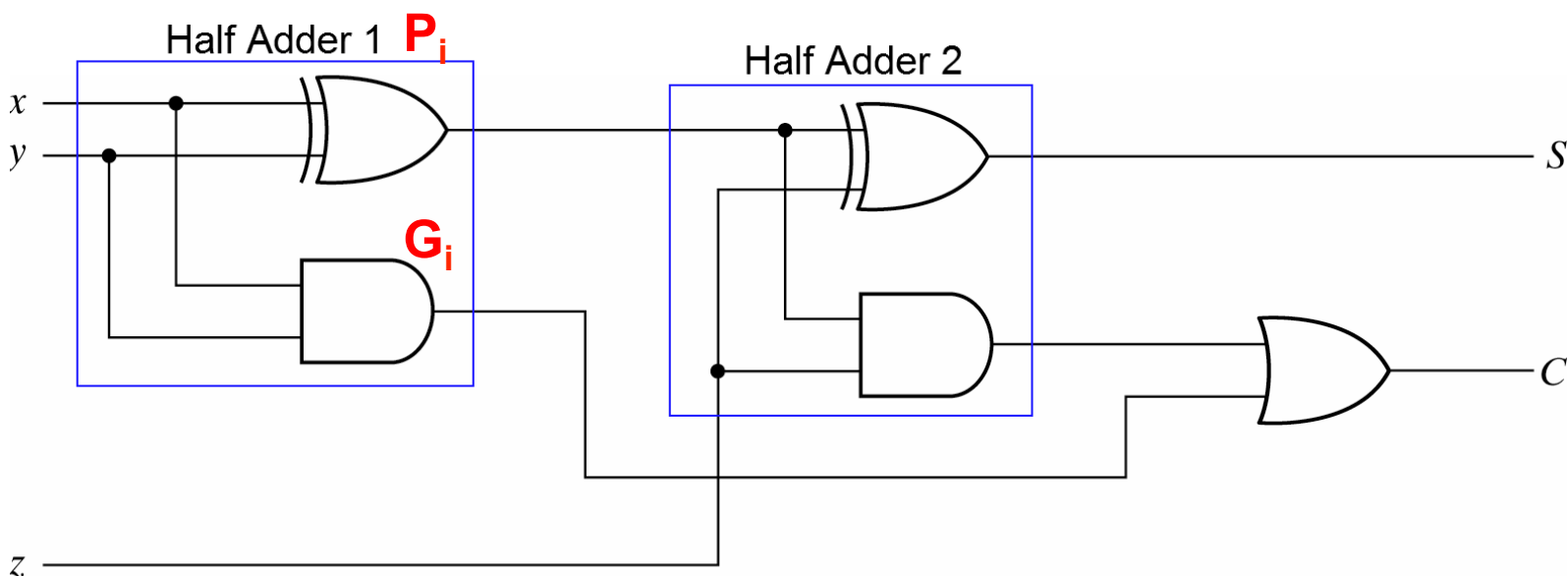
- The sum and carry associated with the i^{th} bit can be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i \cdot C_i$$

Carry Lookahead Generator

- The carry propagate P_i and carry generate G_i can be computed using a half adder.



Carry Lookahead Generator

- The carry outputs for each stage can now be computed in advance, here are the carry outputs for a four bit adder.

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0.C_0$$

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

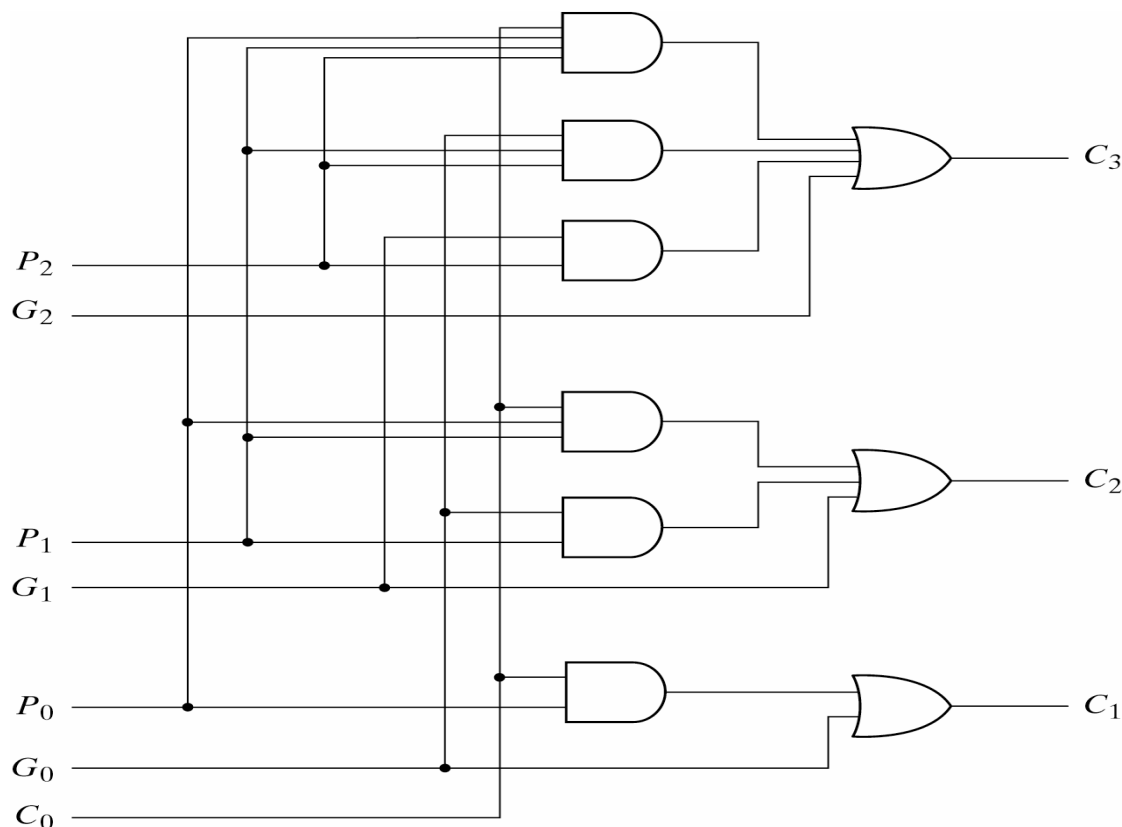
$$C_2 = G_1 + P_1.G_0 + P_1.P_0.C_0$$

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

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

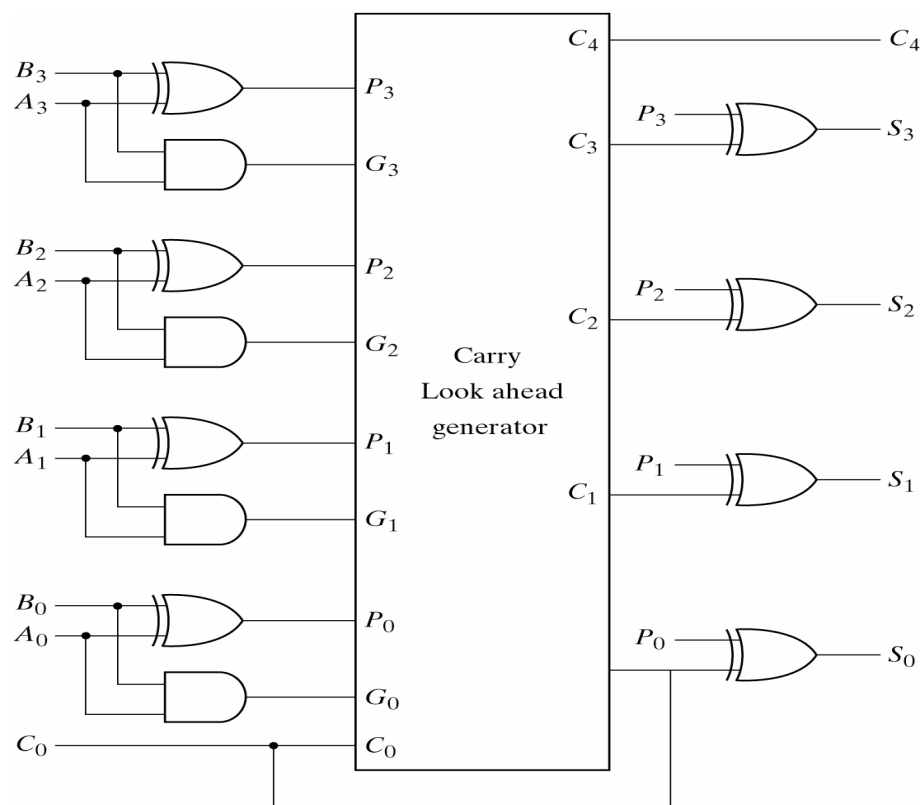
Carry Lookahead Generator

- The combination logic circuit for a 4-bit carry lookahead generator is as follows.



Carry Lookahead Generator

- The 4-bit carry lookahead generator can then be used as part of a 4-bit adder logic circuit.



Practical Solutions

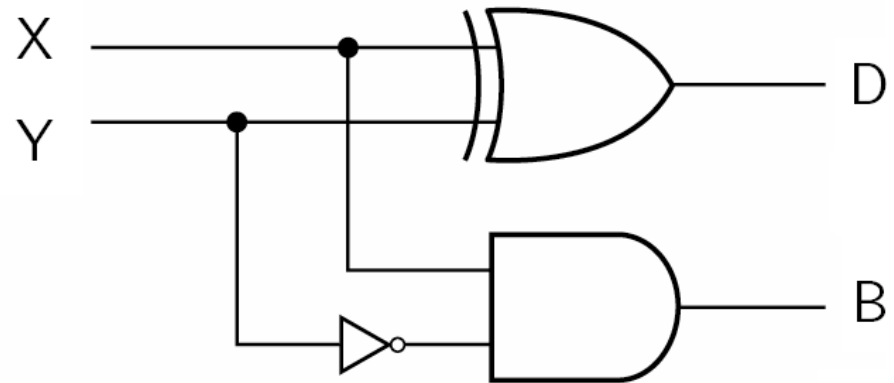
- There are disadvantages to the carry lookahead generator method also:
 - The number of gates increases with $O(n^2)$
 - The number of transistors increases with $O(n^3)$
 - Gates with a large number of inputs are slower and there are technological limits
 - Breaking down into smaller blocks is an option, delay not indep of number of bits
- Often a mixed approach
 - 8-bit adder as 2 4-bit CLA adders connected in ripple adder
 - 16-bit adder as 4 4-bit ripple carry adders connected in CLA way

Subtraction

- Using a similar methodology we can develop a binary subtractor.
- **Specification:** The half subtractor needs to perform four basic operations:
 $0-0=0$, $0-1=-1$, $1-0=1$ and $1-1=0$.
- The half subtractor is similar to the half-adder.
- From the specification it requires two inputs X and Y and two outputs D and B (the *difference* and the *borrow*).

Subtraction

X	Y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0



A full adder can be implemented using two half adders. In a similar way two half subtractors can be used to implement a full subtractor.

Easiest way is to implement subtraction using 2's complement addition

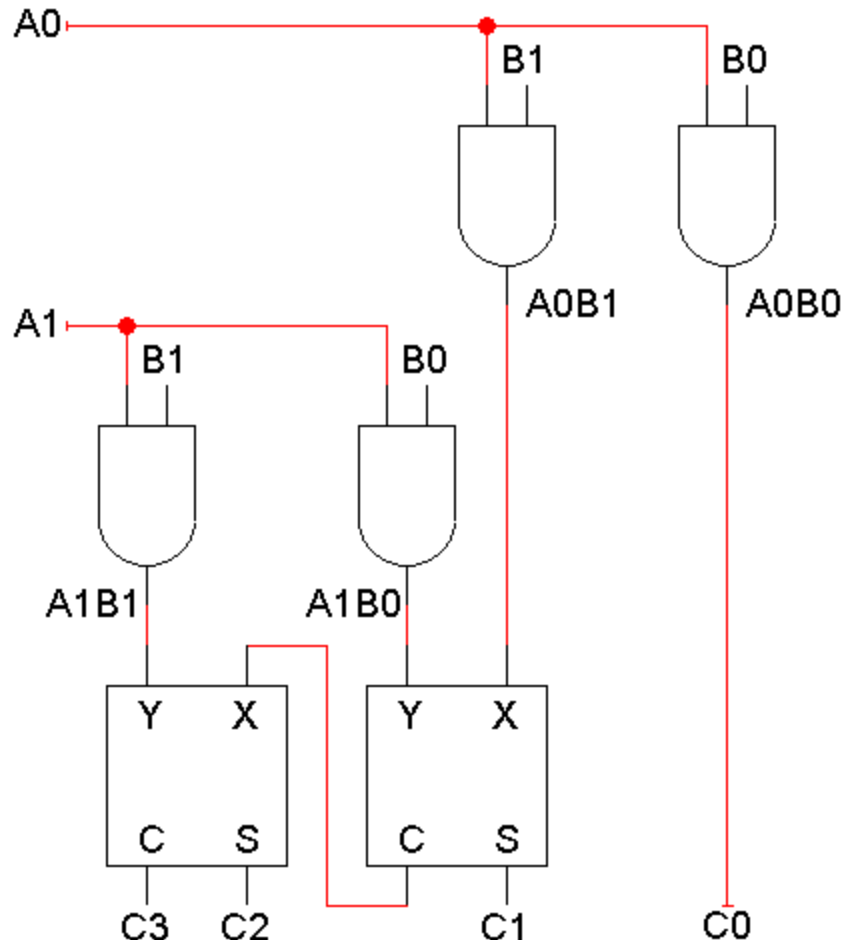
Binary Multiplication

- In lecture 2 we looked at how two binary numbers may be multiplied together.
- Now we first look at the digital circuit which performs this operation for two 2-bit numbers.

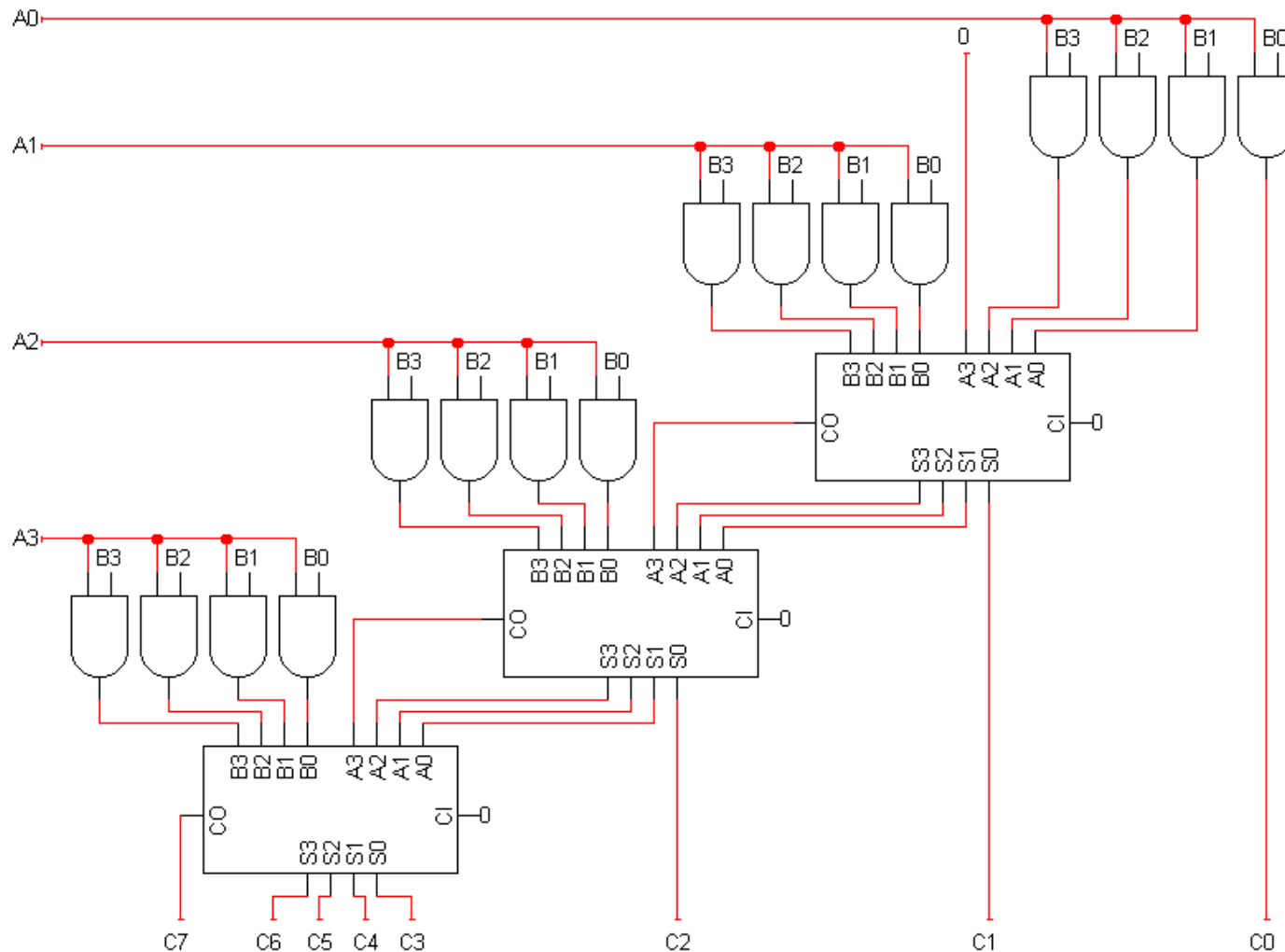
$$\begin{array}{r} 111 \\ \times 101 \\ \hline 111 \\ 0000 \\ + 11100 \\ \hline 100011 \end{array}$$

- The AND gates produce the partial products.
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products.
- Here C_3 - C_0 are the product, not carries!

$$\begin{array}{r}
 B_1 B_0 \\
 A_1 A_0 \\
 \hline
 A_0 B_1 A_0 B_0 \\
 + A_1 B_1 A_1 B_0 \\
 \hline
 C_3 C_2 C_1 C_0
 \end{array}$$



A 4x4 multiplier circuit



Binary Multiplication Special Case

- In decimal, an easy way to multiply by 10 is to shift all the digits to the left, and tack a 0 to the right end.

$$128 \times 10 = 1280$$

- We can do the same thing in binary. Shifting left is equivalent to multiplying by 2:

$$11 \times 10 = 110 \quad (\text{in decimal, } 3 \times 2 = 6)$$

- Shifting left twice is equivalent to multiplying by 4:

$$11 \times 100 = 1100 \quad (\text{in decimal, } 3 \times 4 = 12)$$

- As an aside, shifting to the *right* is equivalent to *dividing* by 2.

$$110 \div 10 = 11 \quad (\text{in decimal, } 6 \div 2 = 3)$$