CMPE 212
Principles of Digital Design

Lecture 15

Designing With Digital Components

March 20, 2016

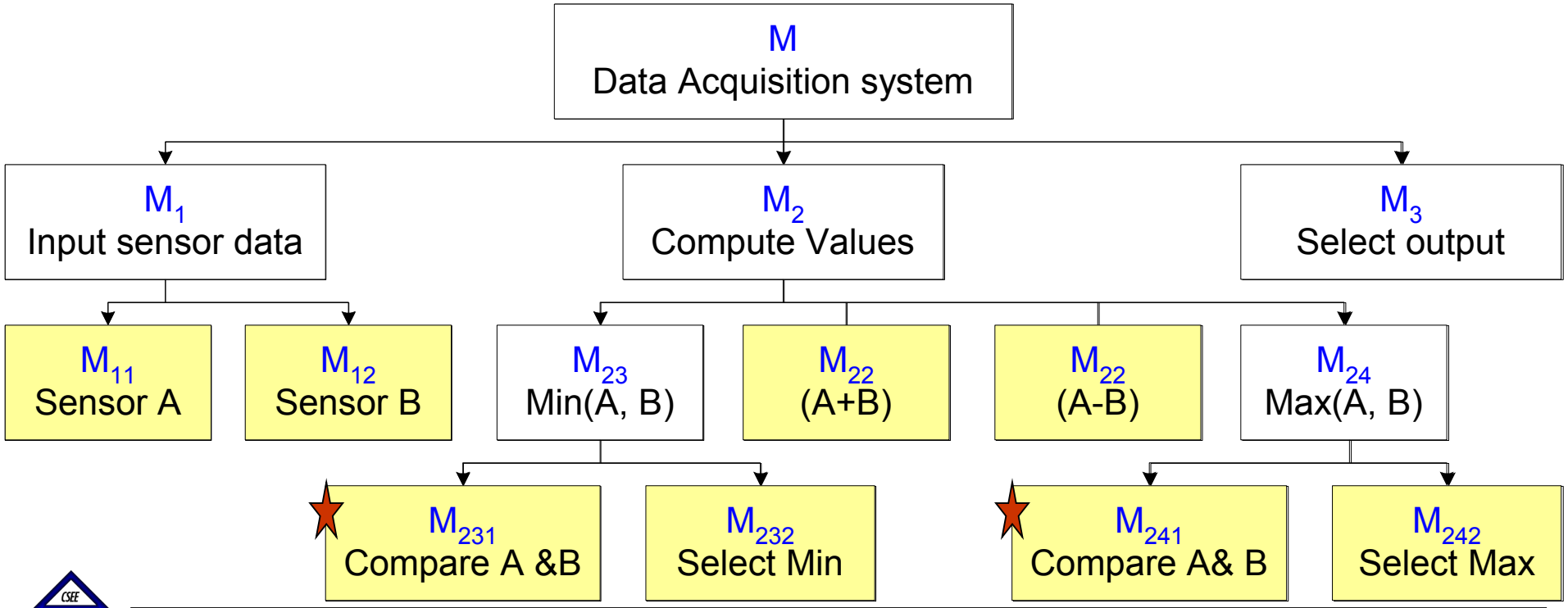
www.csee.umbc.edu/~younis/CMPE212/CMPE212.htm

Lecture's Overview

- ***Previous Lecture:***
  - ➔The Quine-McCluskey algorithm
    (successive reduction, table of choices, Coverage process)
  - ➔ Petrick's algorithm
    (Coverage expression, prime implicants selection)
- ***This Lecture***
  - ➔ Modular Combinational Logic
  - ➔ Examples of medium scale integration components
  - ➔ Designing with digital components
  - ➔ Binary adders

Modular Design

- *Top-down* modular design is the most popular methodology
- A function is initially specified at a high level of abstraction and then decomposed into lower-level modules
- Decomposition enables reuse of sub-functions and available modules (adder, comparators, etc.). After implementation, the modules are integrated from the *bottom-up.*



Digital Components

- High level digital circuit designs are normally created using collections of logic gates referred to as components, rather than using individual logic gates.
- Levels of integration (numbers of gates) in an integrated circuit (IC) can roughly be considered as:
  - Small scale integration (SSI): 10-100 gates.
  - Medium scale integration (MSI): 100 to 1000 gates.
  - Large scale integration (LSI): 1000-10,000 logic gates.
  - Very large scale integration (VLSI): 10,000-upward logic gates.
- These levels are approximate, but the distinctions are useful in comparing the relative complexity of circuits.
- Examine some of the popular digital modules that are available as MSI components, e.g., 74XX series.

* Slide is (partially) courtesy of M. Murdocca and V. Heuring

Decoder

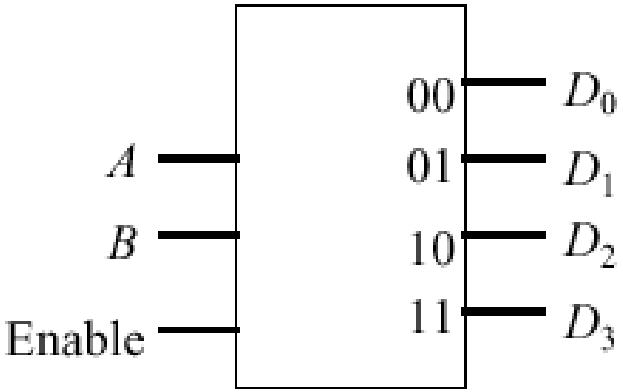
- An n-to-2^n decoder is a multiple-output circuit, with each corresponding to a specific combination of the inputs (exactly one of its output is high at a time, i.e., can be viewed as minterm generator)

Enable = 1					
A	B	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

Enable = 0					
A	B	D_0	D_1	D_2	D_3
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0

Examples:

1. decoding memory address lines in order to access a word
2. Seven segment display



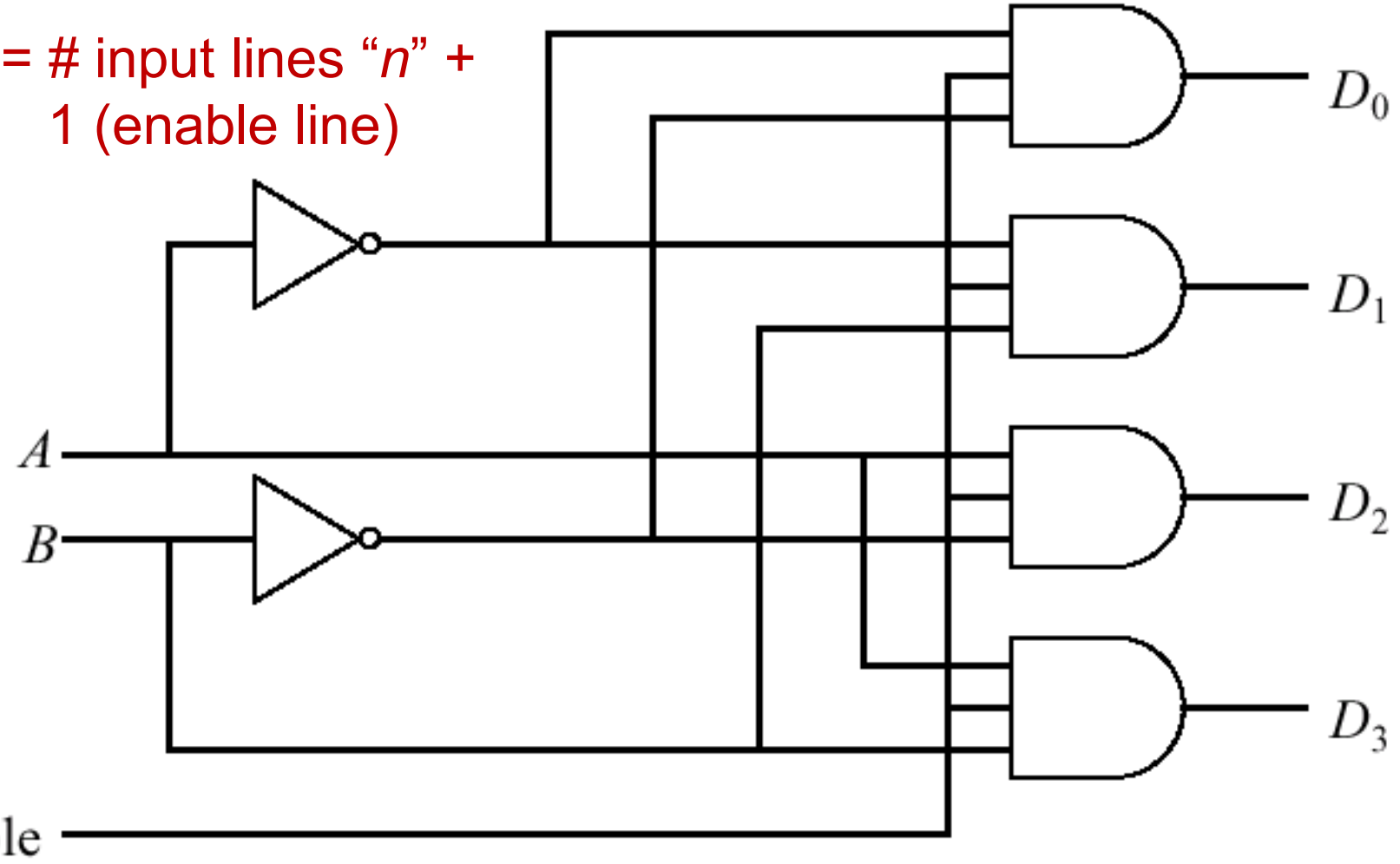
$$D_0 = \overline{A}\,\overline{B} \qquad D_1 = \overline{A}\,B$$

$$D_2 = A\,\overline{B} \qquad D_3 = A\,B$$

Gate-Level Implementation of Decoder

- The "Enable" line allows all output to be set to zero in case it is not appropriate to pick any

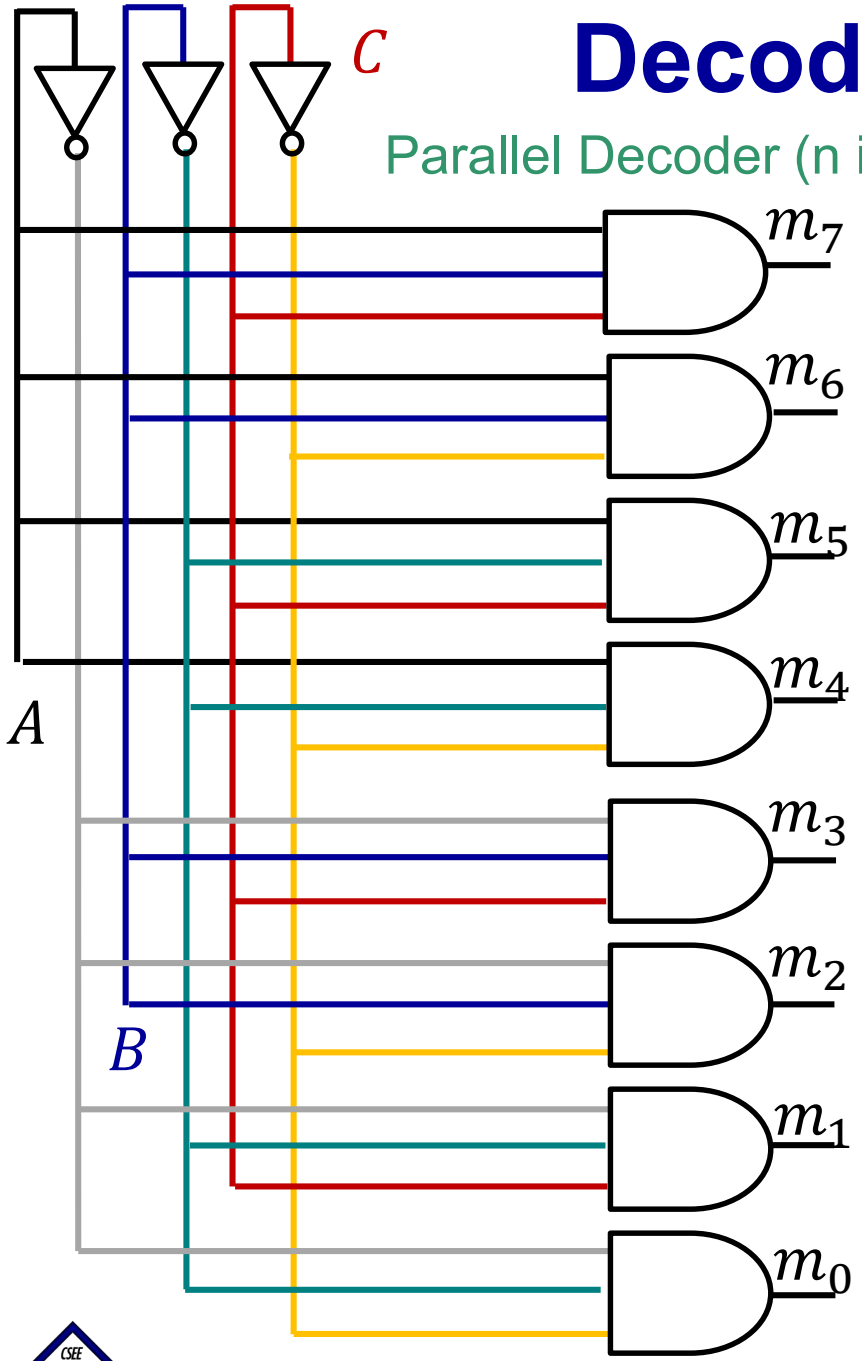
Fan-in = # input lines "*n*" + 1 (enable line)

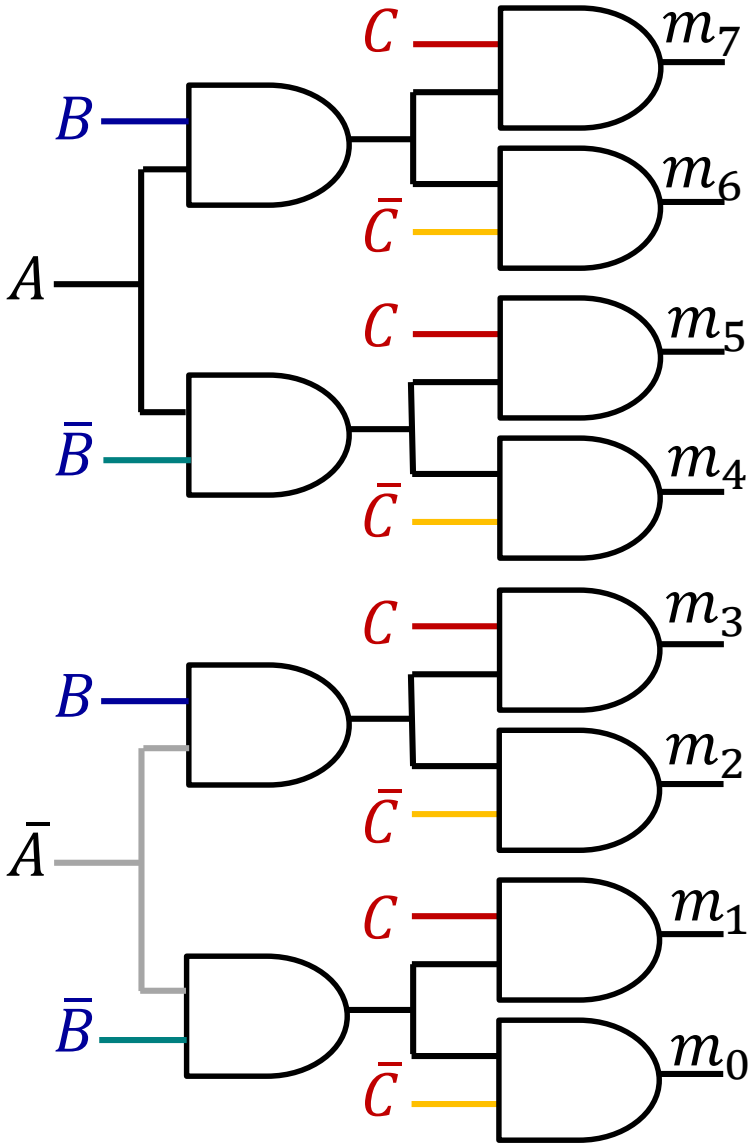


* Slide is courtesy of M. Murdocca and V. Heuring

Decoder Configurations

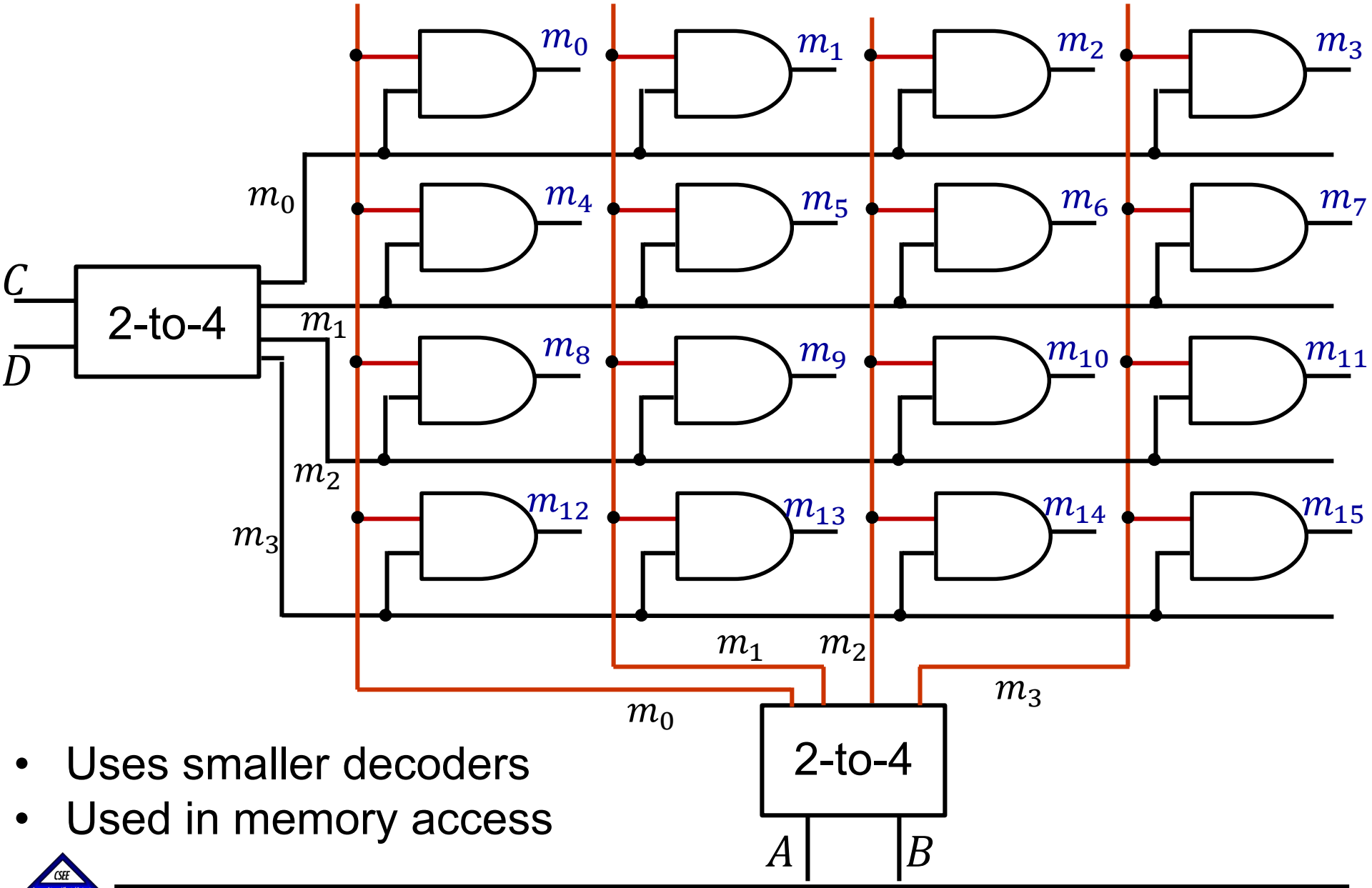
Parallel Decoder (n inputs/gate)





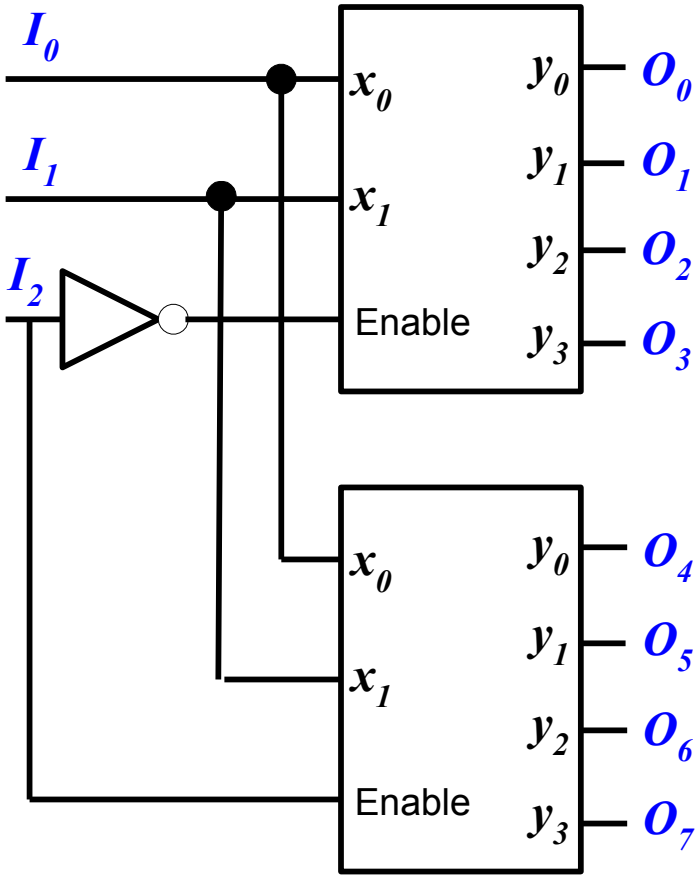
Tree Decoder (fan-in constraints)

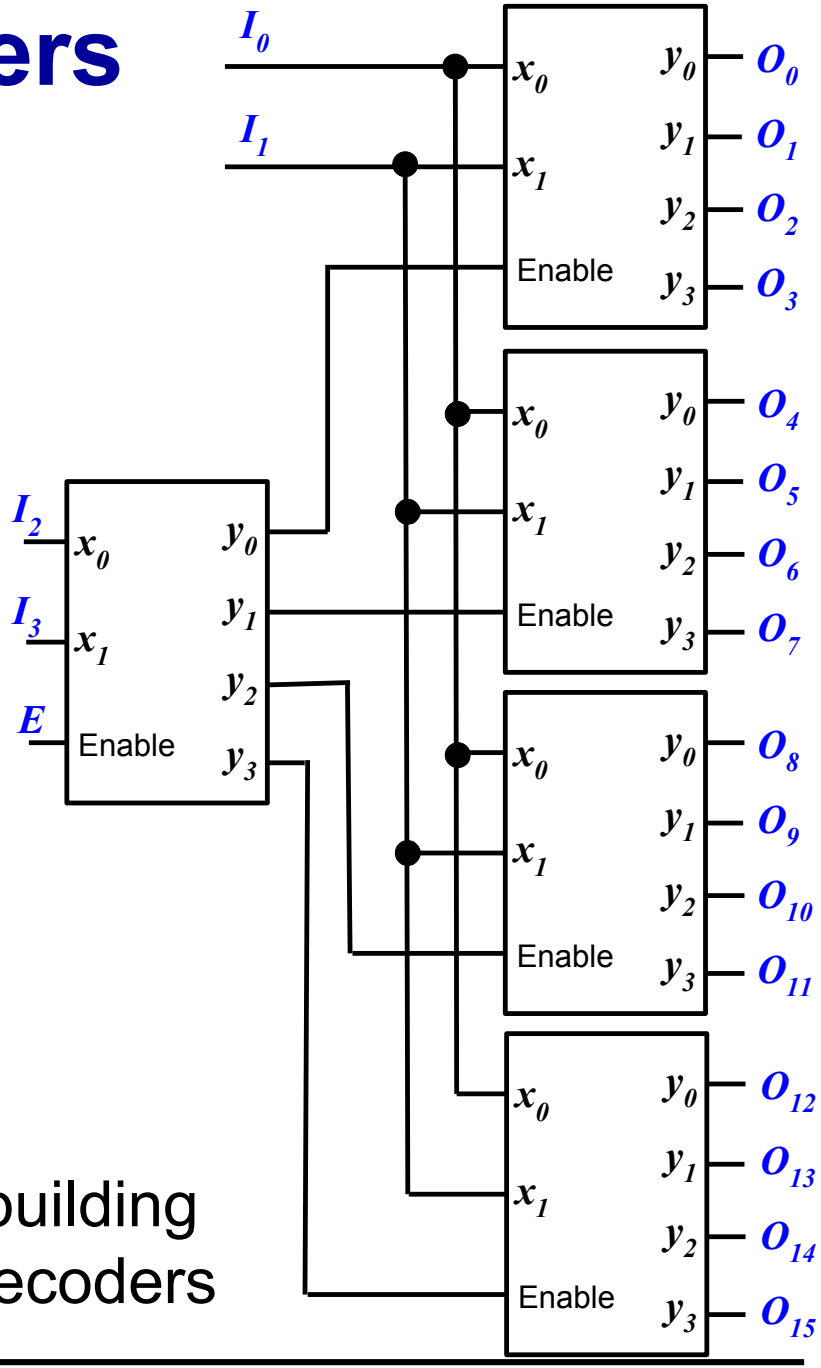
Dual-tree Decoder



- Uses smaller decoders
- Used in memory access

Building Large Decoders Using Small Ones

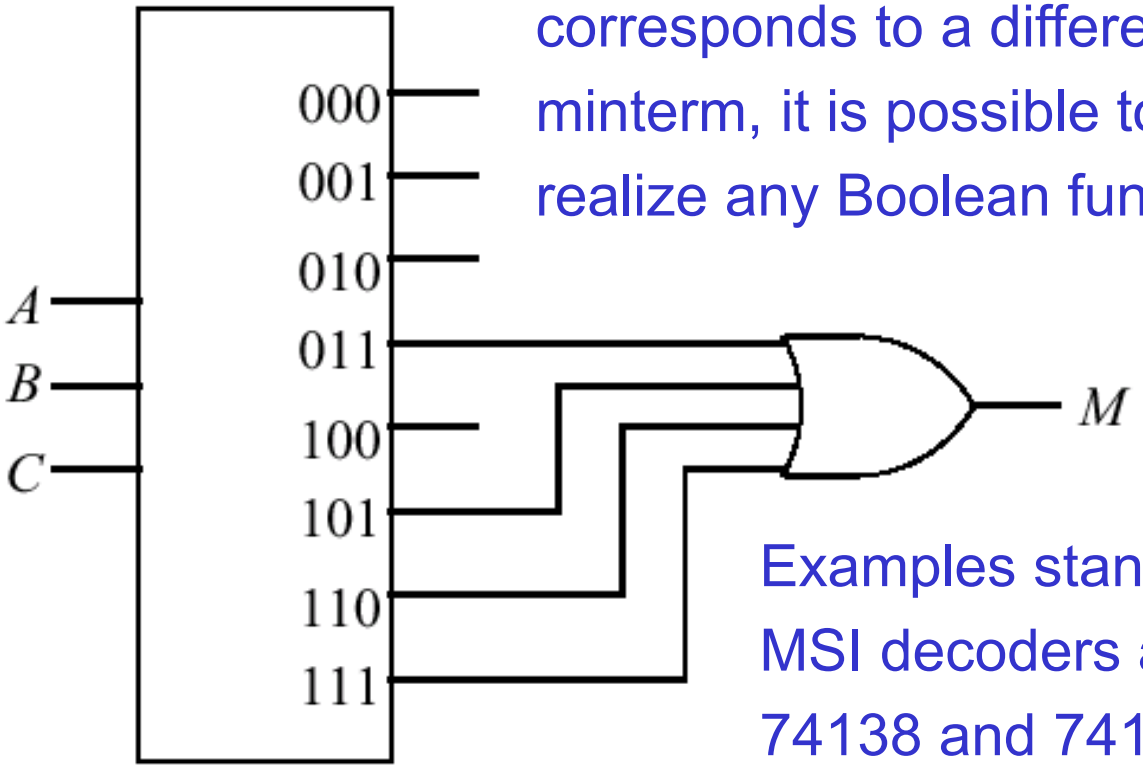




Using of 2-to-4 decoder modules as building blocks to realize 3-to-8 and 4-to-16 decoders

Decoder Implementation of Majority Function

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Since each output line corresponds to a different minterm, it is possible to realize any Boolean function

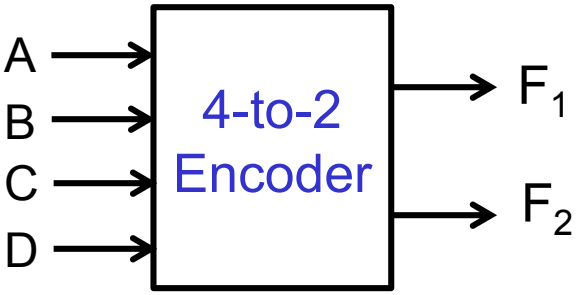
Examples standard MSI decoders are 74138 and 74154.

Note that the enable input is not always present (we use it when discussing decoders for memory devices)

* Figure is courtesy of M. Murdocca and V. Heuring

Encoder

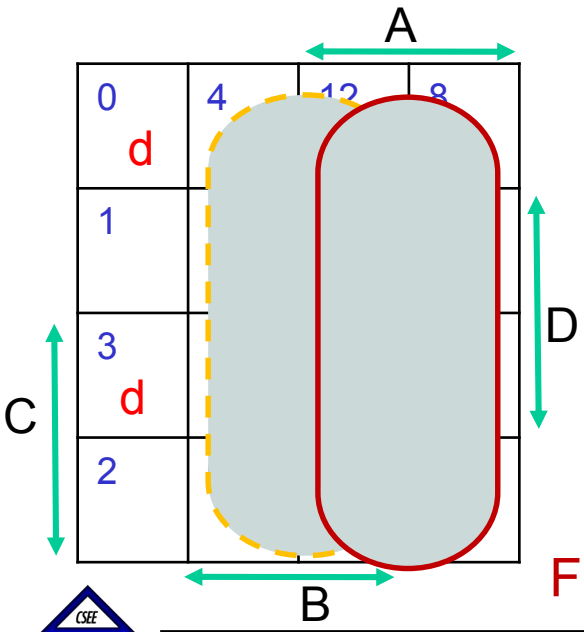
- An encoder translates a set of inputs into a binary encoding (can be thought of as the converse of a decoder)
- Inputs are mutually exclusive, i.e., only one of them is active at a particular time

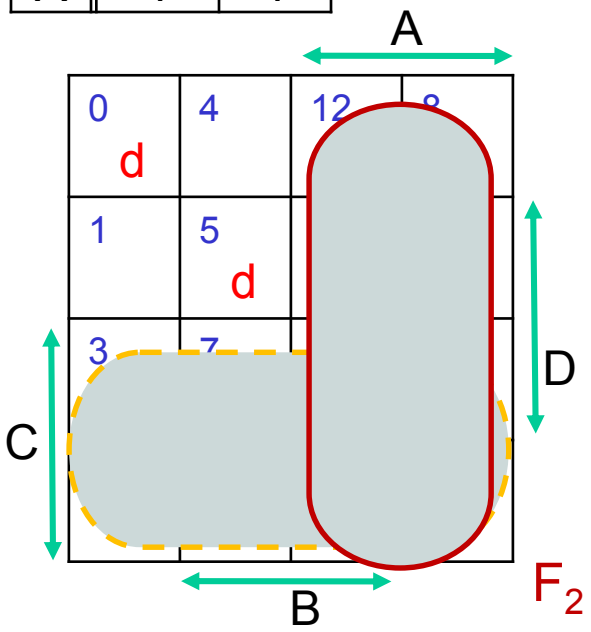


	F_1	F_2
D	0	0
C	0	1
B	1	0
A	1	1

$F_1 = A + B$

$F_2 = A + C$



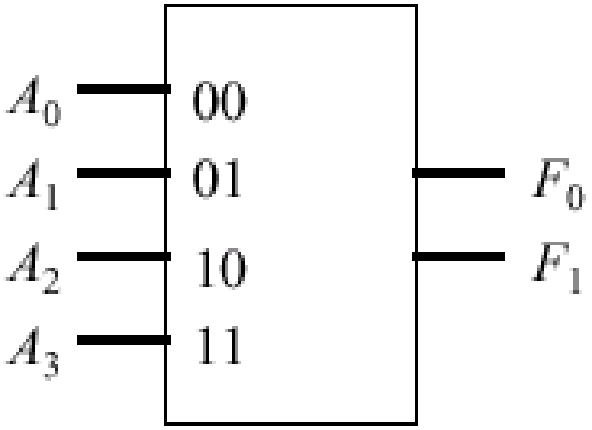


Inputs				Outputs	
A	B	C	D	F_1	F_2
0	0	0	0	d	d
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	d	d
0	1	0	0	1	0
0	1	0	1	d	d
0	1	1	0	d	d
0	1	1	1	d	d
1	0	0	0	1	1
1	0	0	1	d	d
1	0	1	0	d	d
1	0	1	1	d	d
1	1	0	0	d	d
1	1	0	1	d	d
1	1	1	0	d	d
1	1	1	1	d	d

An example Encoder

Priority Encoder

- An encoder translates a set of inputs into a binary bit pattern
- A priority encoder imposes an order on the inputs.
- Useful for connecting interrupts lines (I/O devices)



A_i has a higher priority than A_{i+1}

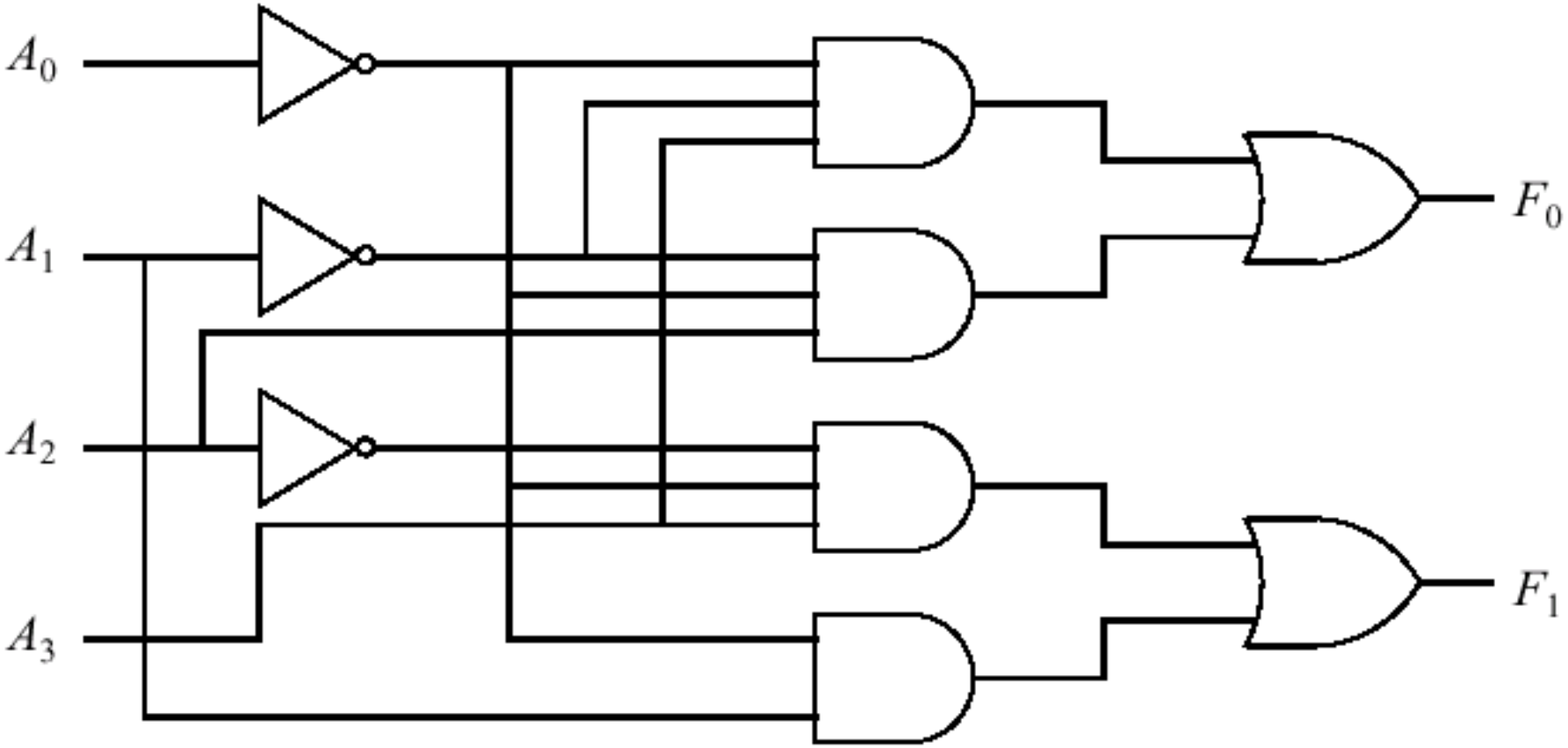
A_0	A_1	A_2	A_3	F_0	F_1
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

$$F_0 = \overline{A_0}\,\overline{A_1}\,A_3 + \overline{A_0}\,A_1\,A_2$$

$$F_1 = \overline{A_0}\,A_2\,A_3 + \overline{A_0}\,A_1$$

* Slide is courtesy of M. Murdocca and V. Heuring

AND-OR Implementation of Priority Encoder



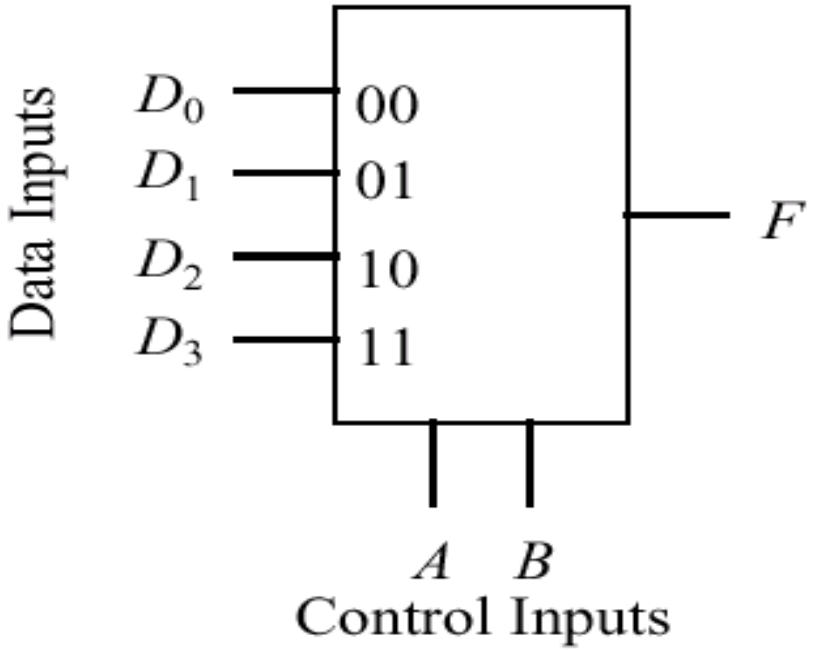
$$F_0 = \overline{A_0}\,\overline{A_1}\,A_3 + \overline{A_0}\,A_1\,A_2$$

$$F_1 = \overline{A_0}\,A_2\,A_3 + \overline{A_0}\,A_1$$

* Slide is courtesy of M. Murdocca and V. Heuring

Multiplexer

- It is a component that connects multiple inputs to a single output
- Control lines are used to select one of the input lines to be accessible from the output line (resembles a multi-setting switch)
- Also called “date selector” and is categorized by the number of channels, i.e., n-to-1 line multiplexer

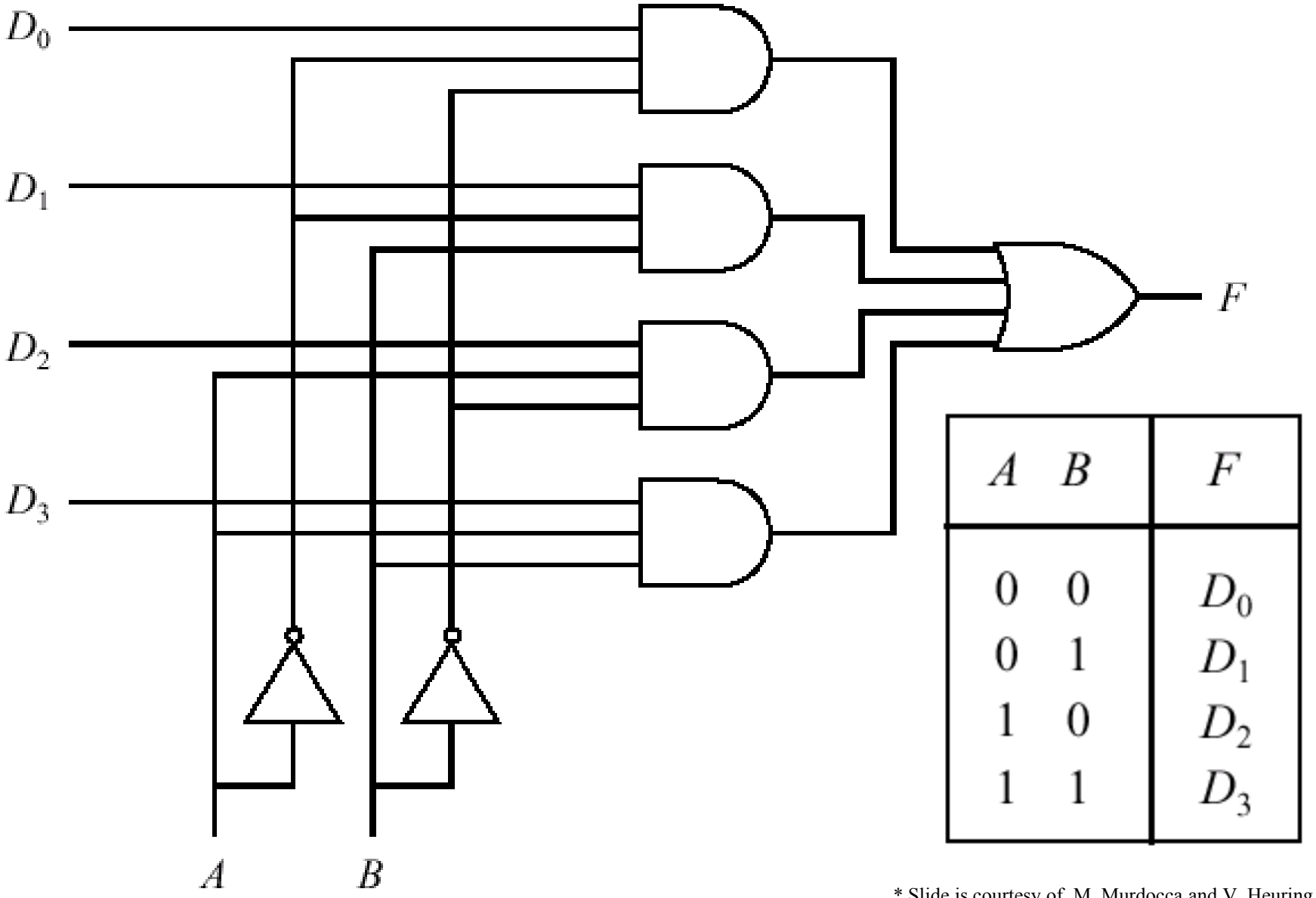


A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$$F = \overline{A}\,\overline{B}\,D_0 + \overline{A}\,B\,D_1 + A\,\overline{B}\,D_2 + A\,B\,D_3$$

* Slide is courtesy of M. Murdocca and V. Heuring

AND-OR Implementation of MUX



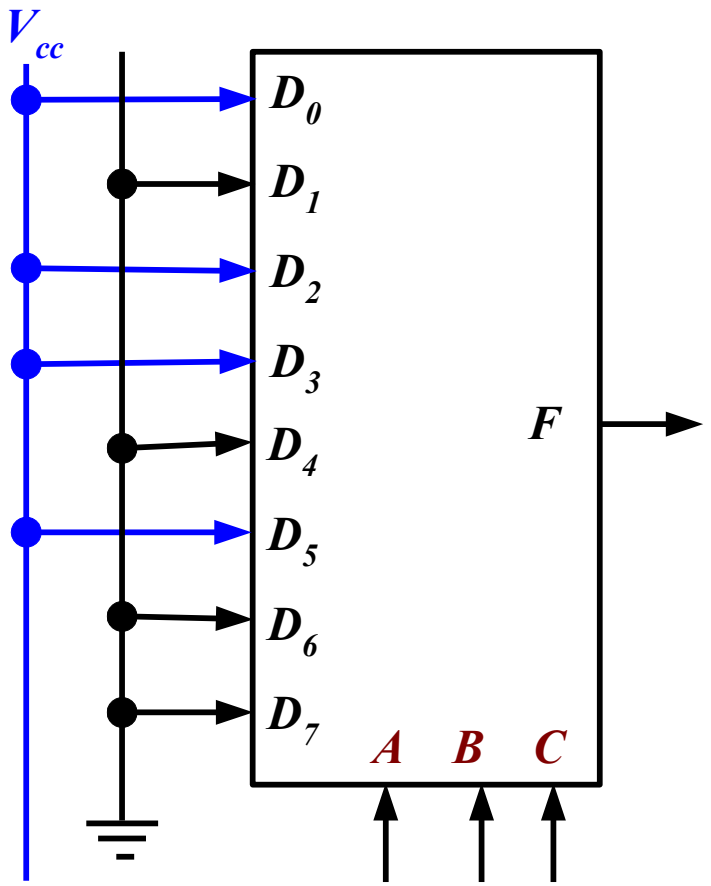
A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

* Slide is courtesy of M. Murdocca and V. Heuring

MUX-based Implementation of Switching Functions

- Principle: Use the selection inputs to select minterms.

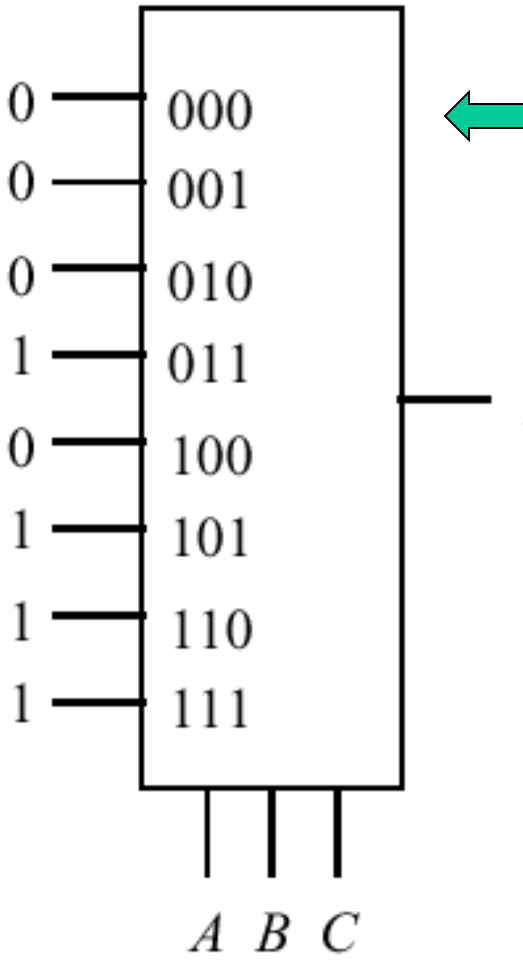
A	B	C	F	
0	0	0	1	$D_0=1$
0	0	1	0	$D_1=0$
0	1	0	1	$D_2=1$
0	1	1	1	$D_3=1$
1	0	0	0	$D_4=0$
1	0	1	1	$D_5=1$
1	1	0	0	$D_6=0$
1	1	1	0	$D_7=0$



Implementation of 3-Var Functions using 8-to-1 multiplexers

- Principle: Use the 3 MUX control inputs to select (one at a time) the 8 data inputs.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

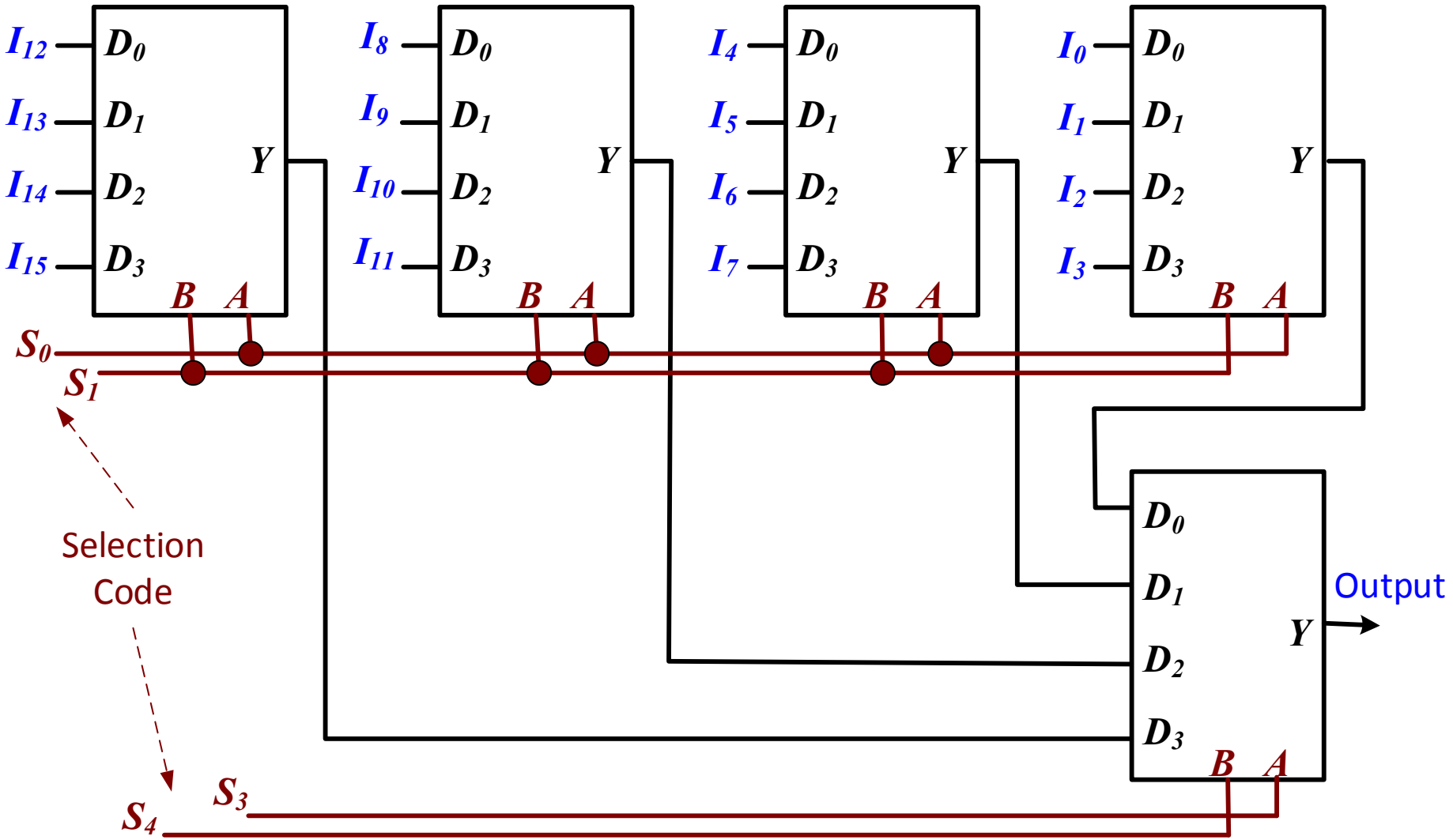


Example: Implementation of Majority function

- Available MSI multiplexers include 74151A (8-1 MUX), 74150 (16-1 MUX), 74153 and74157 multi-bit multiplexers, etc.
- Like decoders, hierarchical set of small multiplexers can be used to realize large ones

* Slide is (partially)courtesy of M. Murdocca and V. Heuring

Multiplexers as Building Blocks

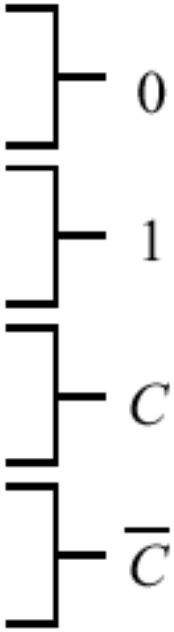


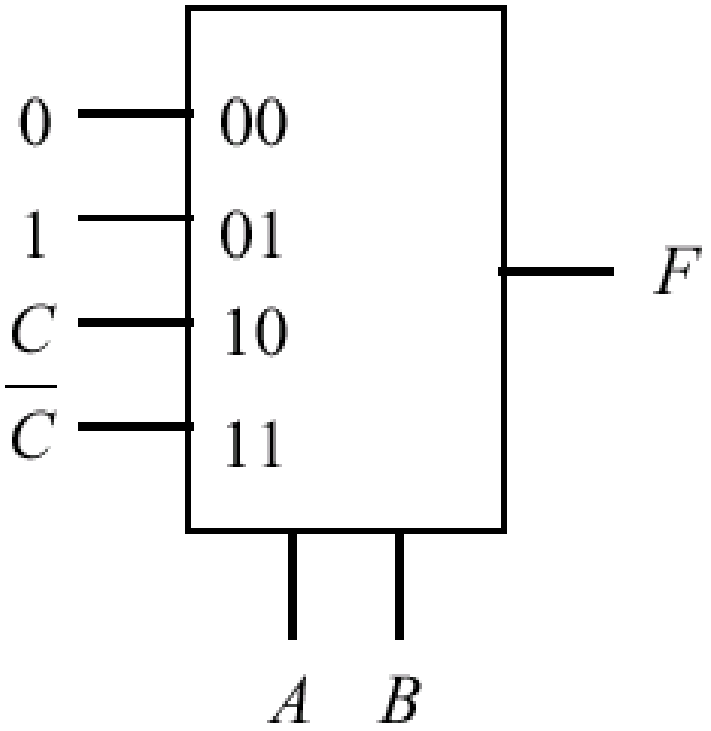
16-to-1 MUX realized with tree-type network of 4-to-1 multiplexers

4-to-1 MUX Implements 3-Var Function

- Principle: Use the A and B inputs to select a pair of minterms.
- The value applied to the MUX data input is selected from $\{0, 1, C, \bar{C}\}$ to achieve the desired behavior of the minterm pair.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

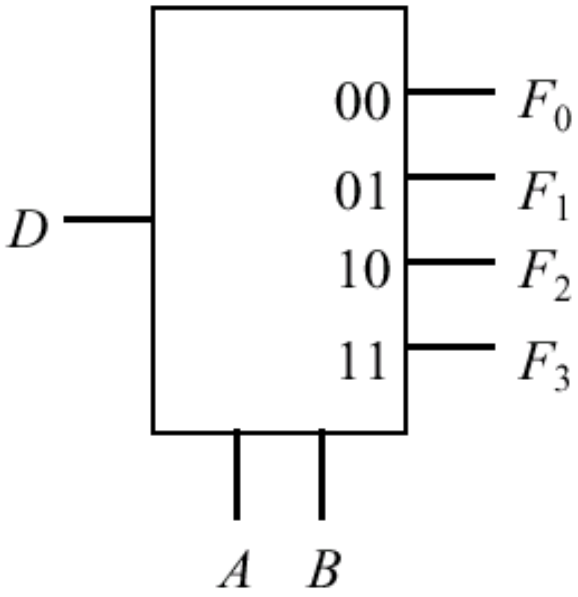




* Slide is courtesy of M. Murdocca and V. Heuring

Demultiplexer

- DEMUX is the converse of a MUX (send its single data input to a selected one of its output)
- Example application: a call request button for an elevator to the closest elevator car



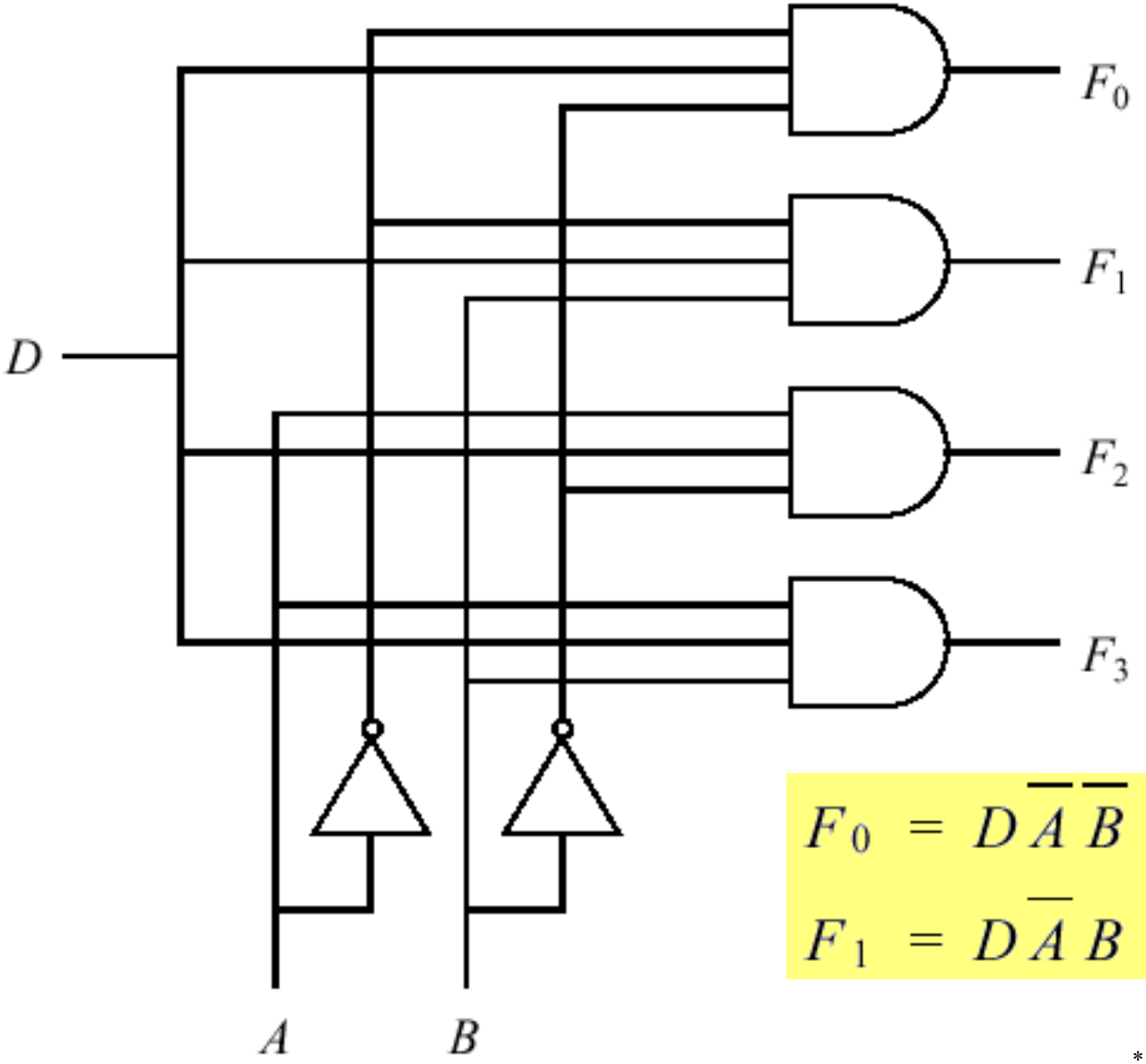
D	A	B	F_0	F_1	F_2	F_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

$$F_0 = D\overline{A}\,\overline{B} \qquad F_2 = DA\overline{B}$$

$$F_1 = D\overline{A}B \qquad F_3 = DAB$$

* Slide is courtesy of M. Murdocca and V. Heuring

Gate-Level Implementation of DEMUX

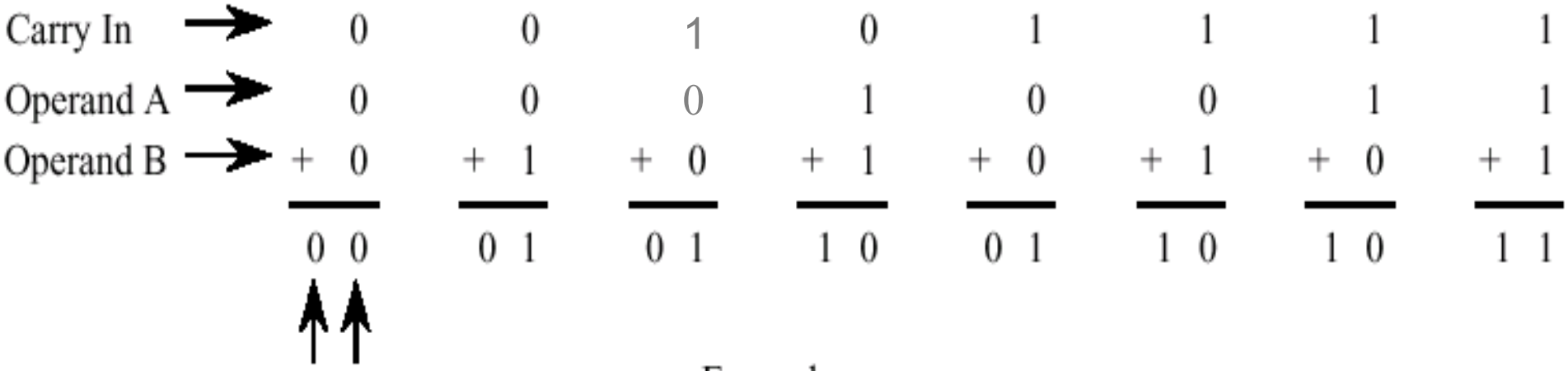


$$F_0 = D\overline{A}\,\overline{B} \qquad F_2 = D\,A\,\overline{B}$$

$$F_1 = D\overline{A}\,B \qquad F_3 = D\,A\,B$$

* Slide is courtesy of M. Murdocca and V. Heuring

Ripple-Carry Addition

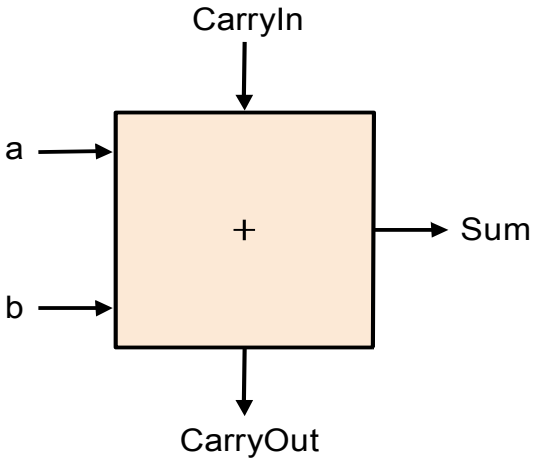


Example:

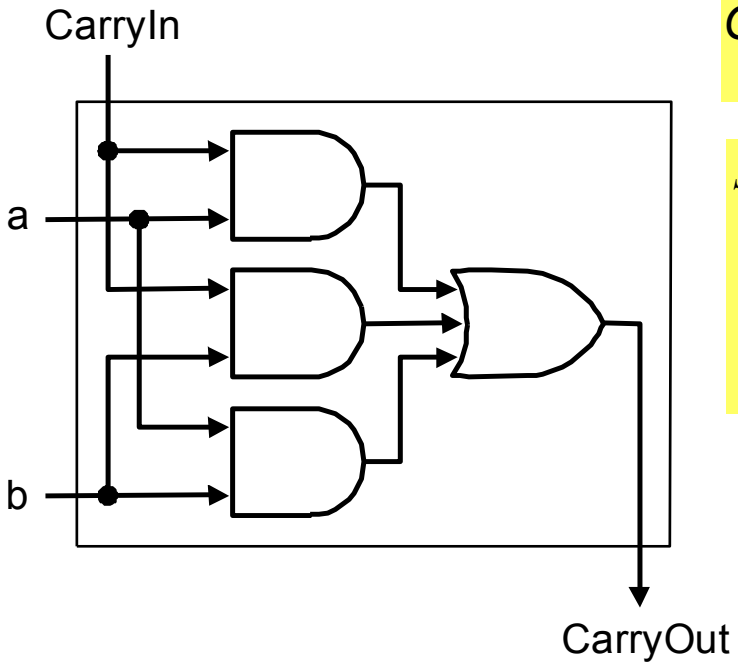
Carry	1	0	0	0
Operand A	0	1	0	0
Operand B +	0	1	1	0
Sum	1	0	1	0

* Slide is courtesy of M. Murdocca and V. Heuring

A 1-Bit Full Adder



Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
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

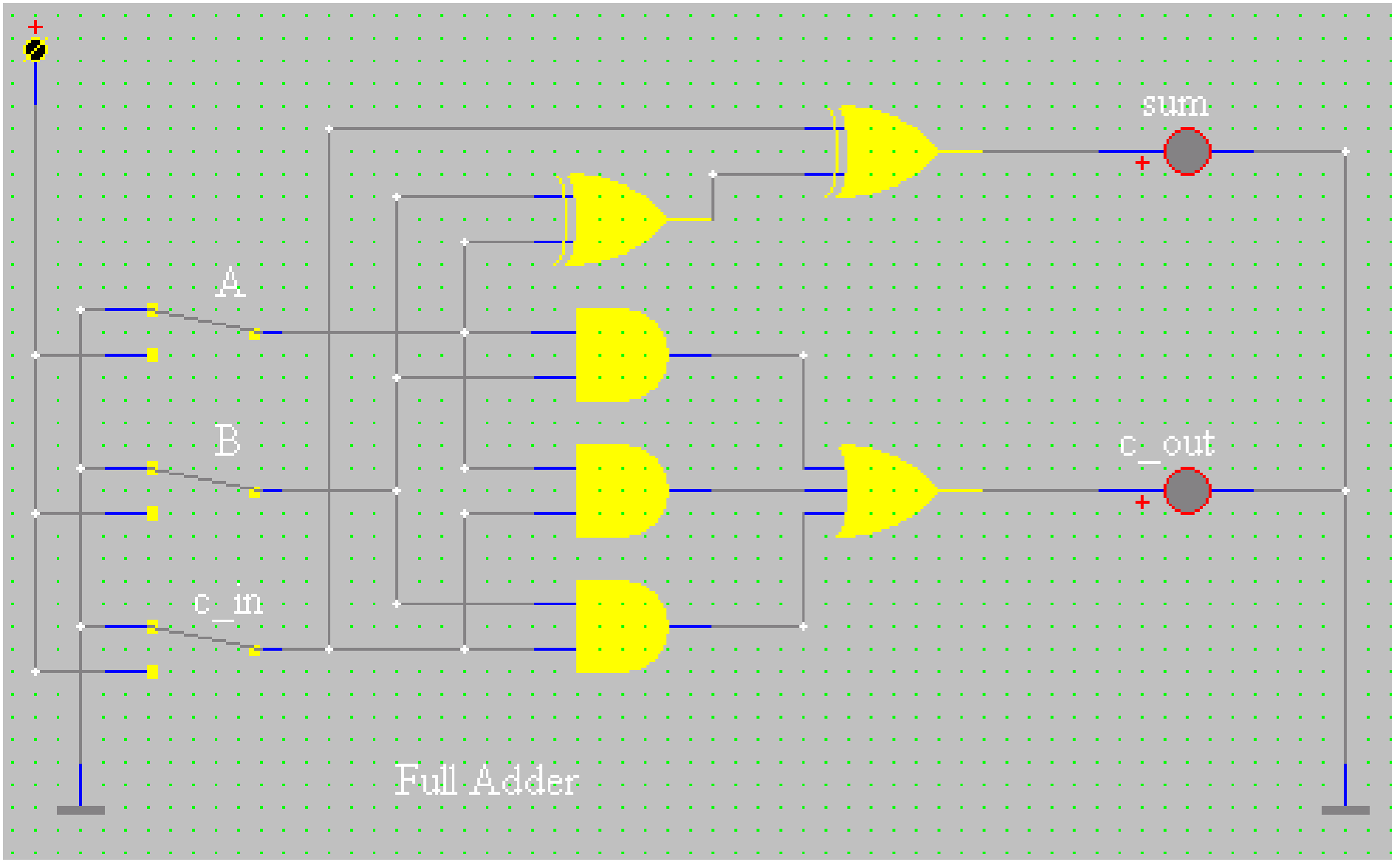


$$CarryOut = (b.CarryIn) + (a.CarryIn) + (a.b) + (a.b.CarryIn)$$
$$= (b.CarryIn) + (a.CarryIn) + (a.b)$$

$$Sum = (a.\bar{b}.\overline{CarryIn}) + (\bar{a}.b.\overline{CarryIn}) + (\bar{a}.\bar{b}.CarryIn) + (a.b.CarryIn)$$
$$= \overline{CarryIn}(a.\bar{b} + \bar{a}.b) + CarryIn(\bar{a}.\bar{b} + a.b)$$
$$= \overline{CarryIn}\,(a \oplus b) + CarryIn\,\overline{(a \oplus b)}$$
$$= CarryIn \oplus (a \oplus b)$$

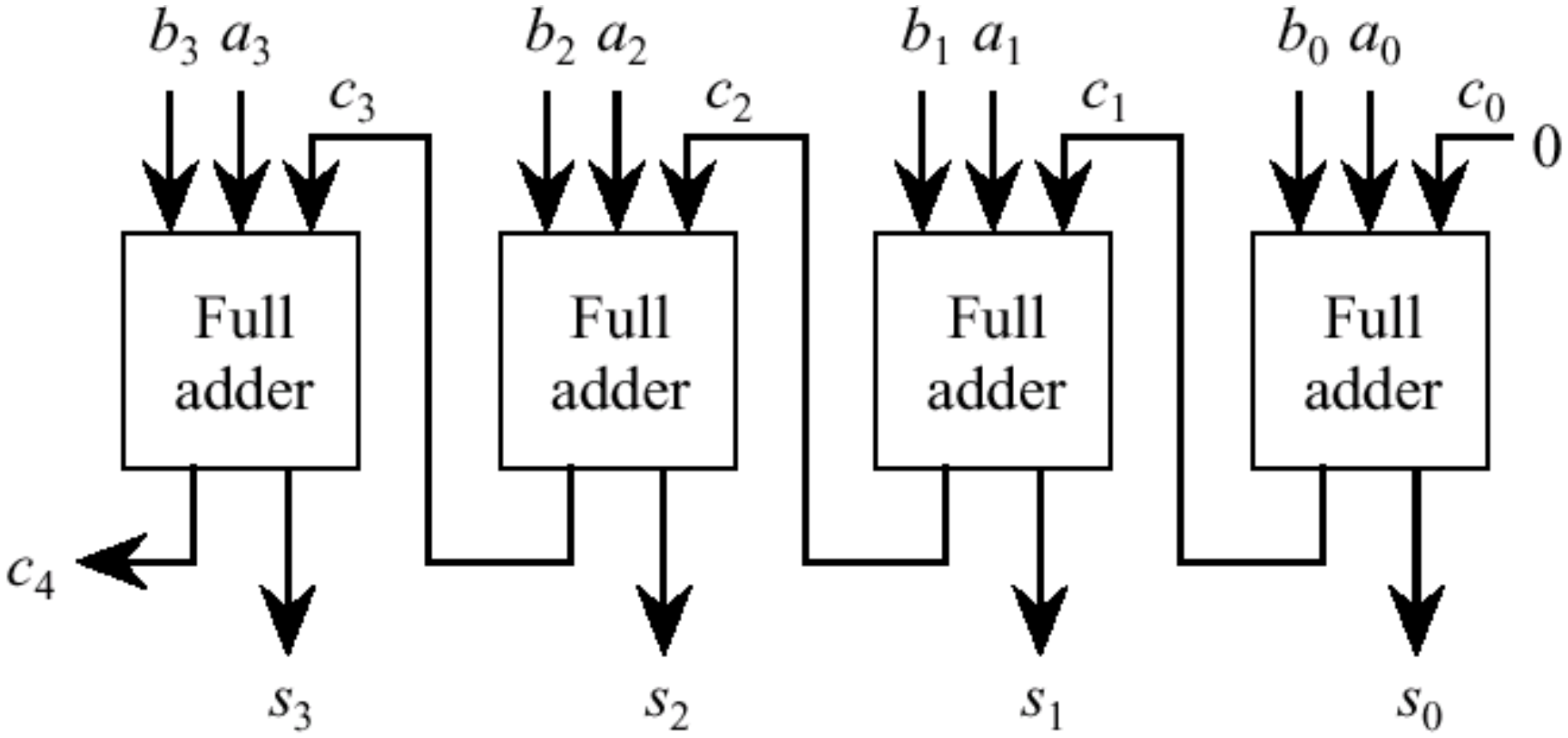
A single bit adder has 3 inputs, two operands and a carry-in and generates a sum bit and a carry-out to passed to the next 1-bit adder

Full Adder Circuit



Four-Bit Ripple-Carry Adder

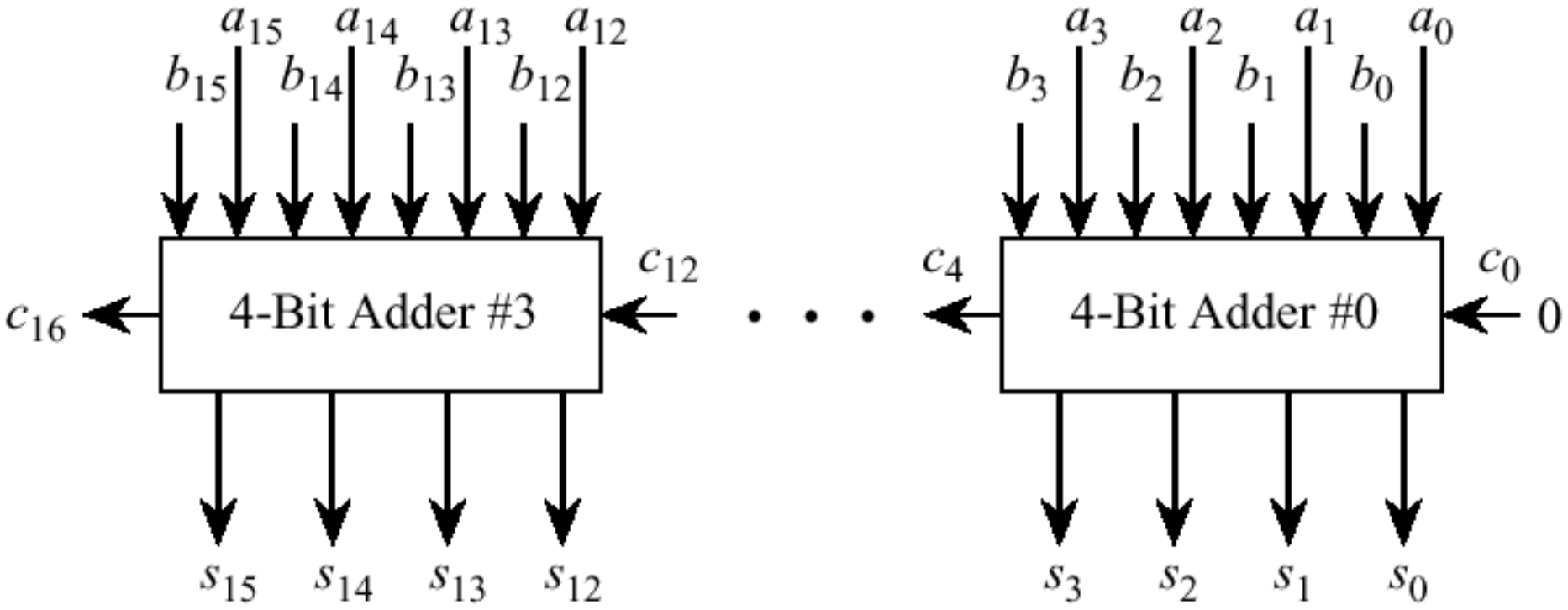
- Two binary numbers A and B are added from right to left, creating a sum and a carry at the outputs of each full adder for each bit position.
- Four full adders connected in a ripple-carry chain form a four-bit adder.



* Slide is courtesy of M. Murdocca and V. Heuring

Constructing Larger Adders

- A 16-bit adder can be made up of a cascade of four 4-bit ripple-carry adders

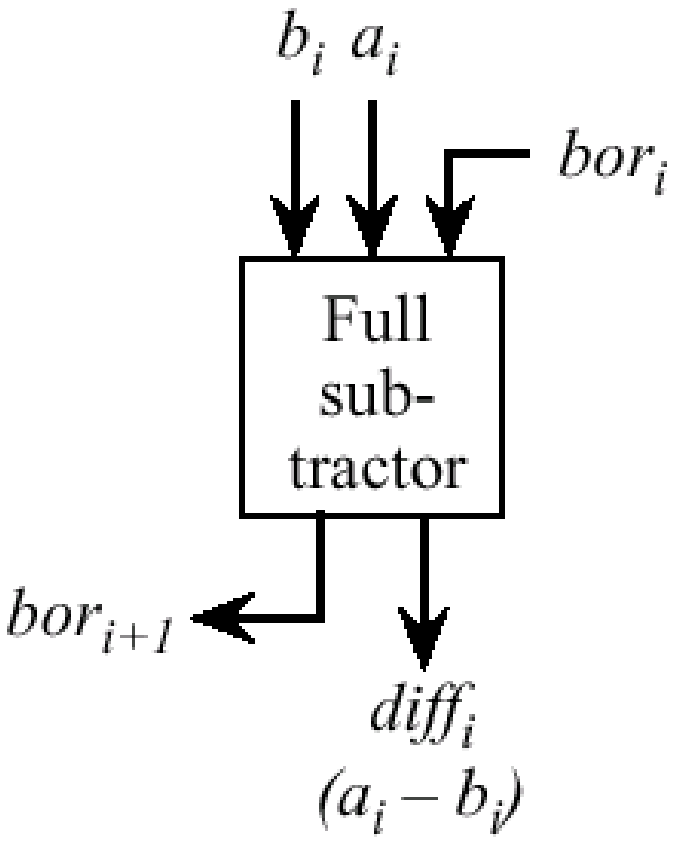


* Slide is courtesy of M. Murdocca and V. Heuring

Full Subtractor

- Truth table and schematic symbol for a ripple-borrow subtractor:

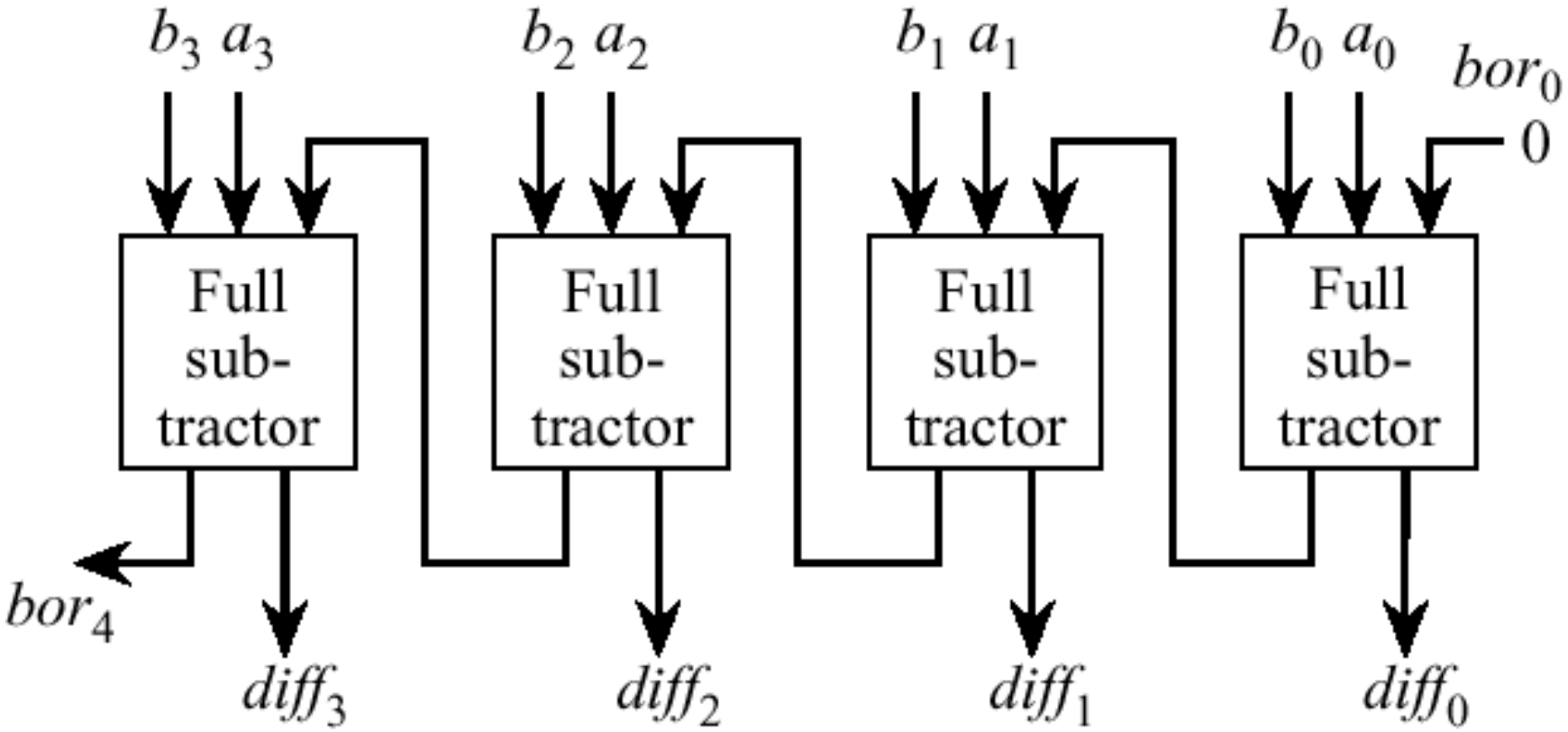
a_i	b_i	bor_i	$diff_i$	bor_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



* Slide is courtesy of M. Murdocca and V. Heuring

Ripple-Borrow Subtractor

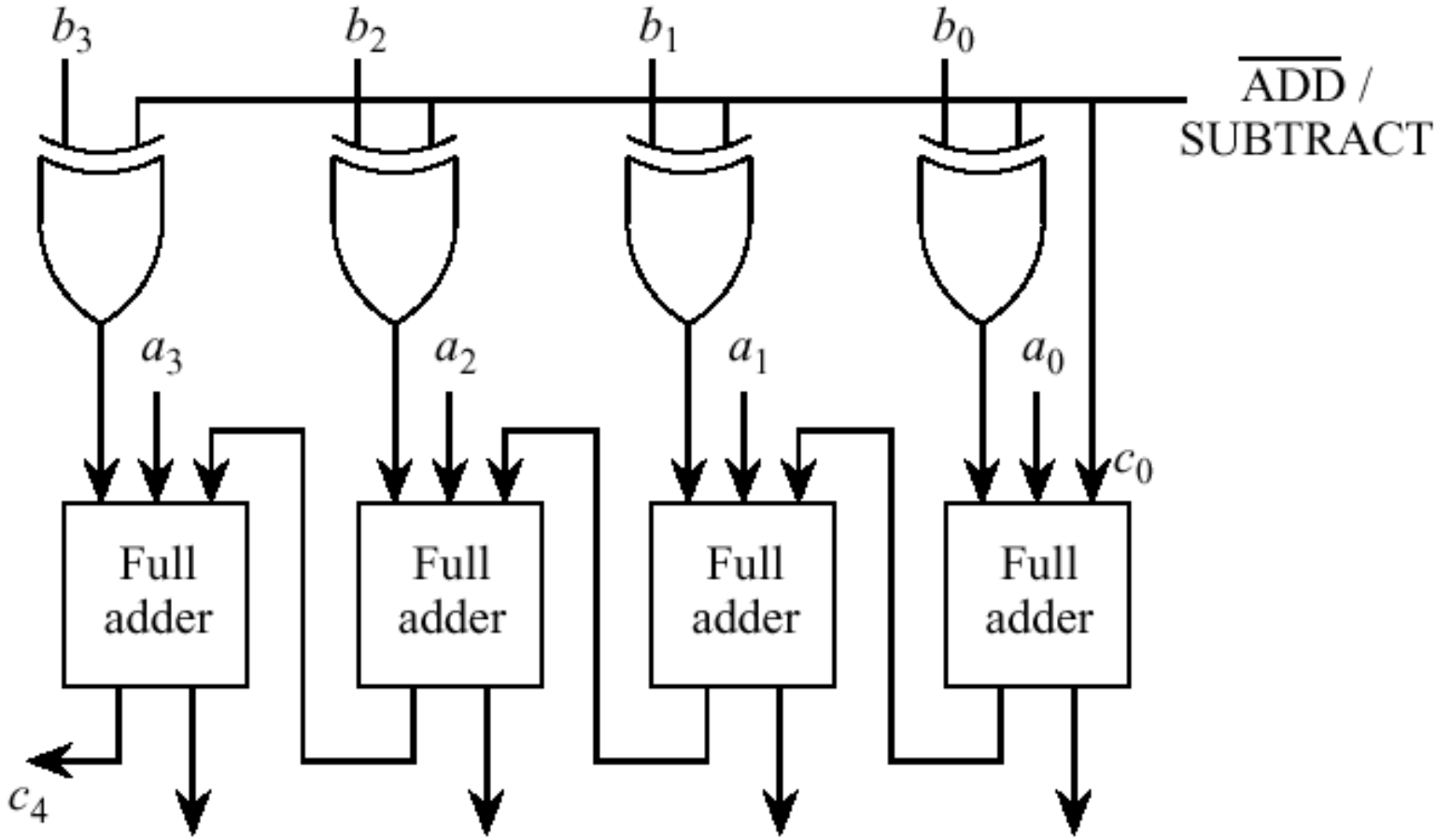
- A ripple-borrow subtractor composed of a cascade of full subtractors
- Two binary numbers A and B are subtracted from right to left, creating a difference and a borrow at the outputs of each full subtractor for each bit



* Slide is courtesy of M. Murdocca and V. Heuring

Combined Adder/Subtractor

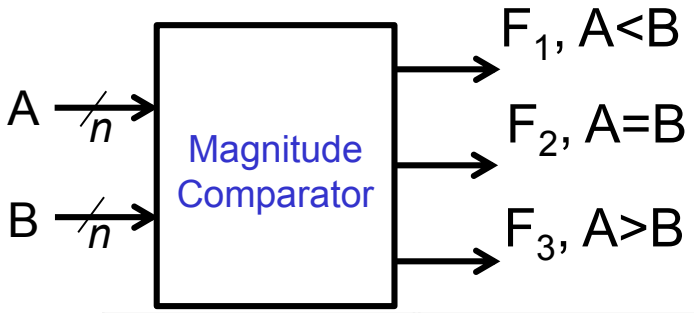
- A single ripple-carry adder can perform both addition and subtraction, by forming the two's complement negative for B when subtracting. (Note that +1 is added at c_0 for two's complement.)



* Slide is courtesy of M. Murdocca and V. Heuring

Comparators

- A comparator checks the magnitude of two binary numbers and determines if they are equal or which one is larger.

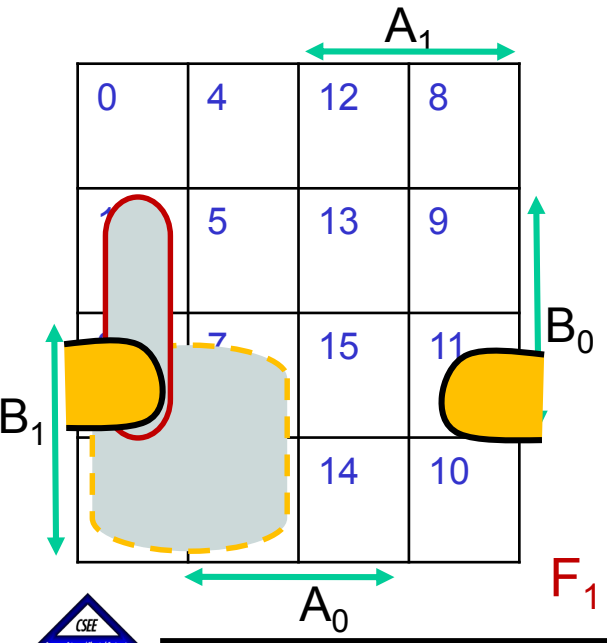


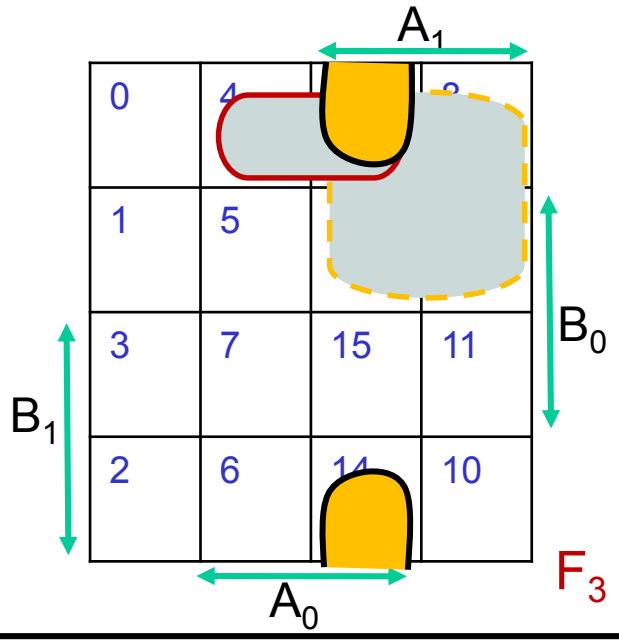
$$F_1 = \overline{A_1}B_1 + \overline{A_0}\,\overline{A_1}B_0 + \overline{A_0}B_0B_1$$

$$F_2 = \overline{A_0}\,\overline{A_1}\,\overline{B_0}\,\overline{B_1} + A_0\overline{A_1}\,B_0\overline{B_1} + \overline{A_0}\,A_1\overline{B_0}B_1 + A_0A_1B_0B_1$$

$$F_3 = A_1\overline{B_1} + A_0\overline{B_0}\,\overline{B_1} + A_0A_1\overline{B_0}$$

Inputs				Outputs		
A_0	A_1	B_0	B_1	F_1	F_2	F_3
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0





Conclusion

- ***Summary***
  - ➔ Modular Combinational Logic
  - ➔ Examples of medium scale integration components
    (Multiplexers, Decoders, Encoders, etc.)
  - ➔ Designing with digital components
    (Tree-type arrangements, implementing switching functions)
  - ➔ Binary ripple-carry adders
- ***Next Lecture***
  - ➔ Programmable logic devices

Reading assignment: Sections 4.1 – 4.6 in the textbook