Combinational Logic Design

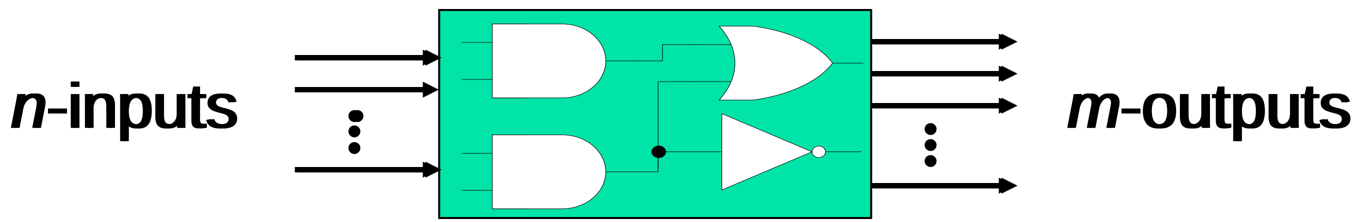
Combinational Functions and Circuits

Overview

- Combinational Circuits
- Design Procedure
  - Generic Example
  - Example with don't cares: BCD-to-SevenSegment converter
- Binary Decoders
  - Functionality
  - Circuit Implementation with Decoders
  - Expansion
- Multiplexers (MUXs)
  - Functionality
  - Circuit Implementation with MUXs
  - Expansions

Combinational Circuits

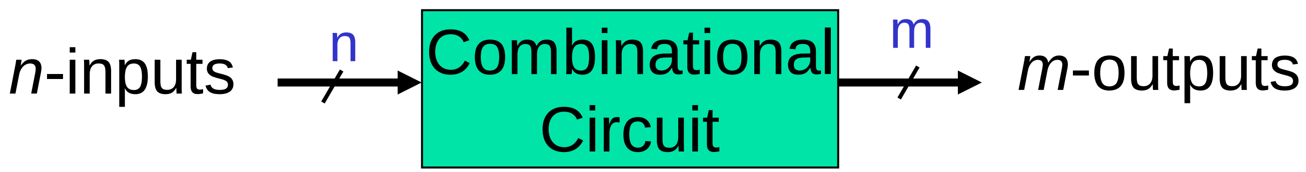
- A combinational circuit consists of logic gates



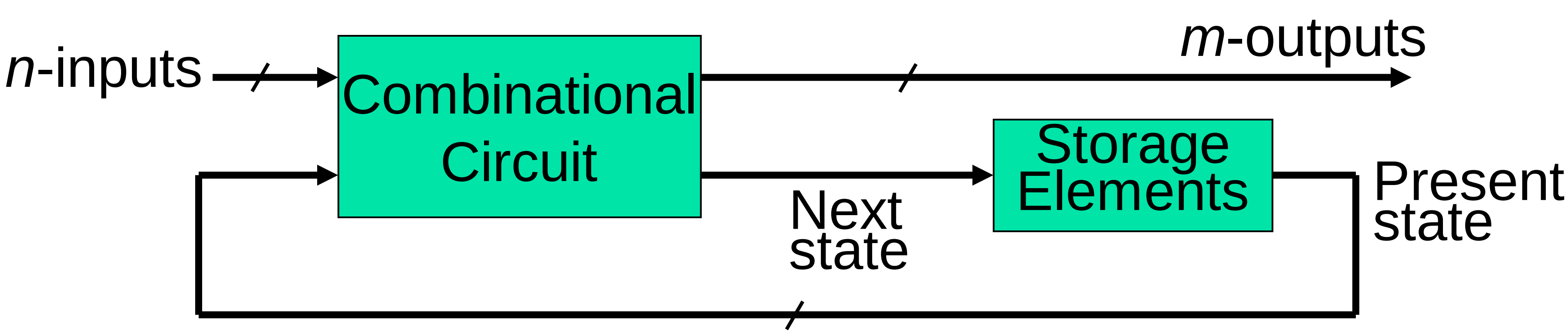
- The circuit outputs, at any time, are determined by combining the values of the inputs
- For ***n*** inputs, there are 2^n possible binary input combinations
- For each combination, there is one possible binary value on each output
- Hence, a combinational circuit can be described by:
  - Truth Table
    - lists the output values for each combination of the inputs
  - *m* Boolean functions, one for each output

Combinational vs. Sequential Circuits

- Combinational circuits are memory-less!
  - Thus, the output values depend ONLY on the current input values



- Sequential circuits consist of combinational logic as well as memory (storage) elements!
  - Memory elements used to store certain circuit states
  - Outputs depend on BOTH current input values and previous input values kept in the storage elements



Combinational Circuit Design

- ***Design*** of a combinational circuit is the development of a circuit from a description of its function
- It starts with a problem specification
- It produces
  - a logic diagram

  OR
  - set of Boolean equations that represent the circuit

Design Procedure

Consists of 5 major steps:

1. Determine the required number of inputs and outputs and assign variables to them
2. Derive the truth table that defines the required relationship between inputs and outputs
3. Obtain and simplify the Boolean functions

   -- Use K-maps, algebraic manipulation, CAD tools, etc.

   -- Consider any design constraints (area, delay, power, available libraries, etc.)
4. Draw the logic diagram
5. Verify the correctness of the design

Design Example

- Design a combinational circuit with 4 inputs that generates a 1 when
  - the # of 1s on the inputs equals the # of 0s

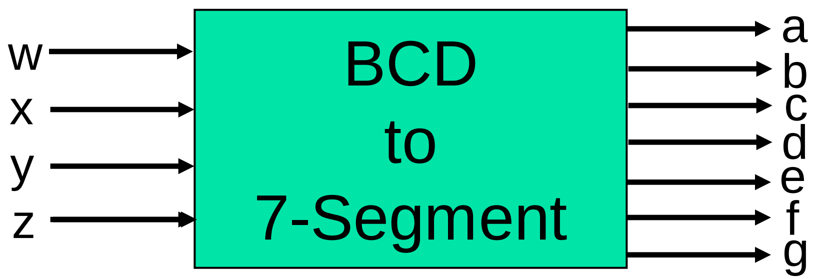
  OR

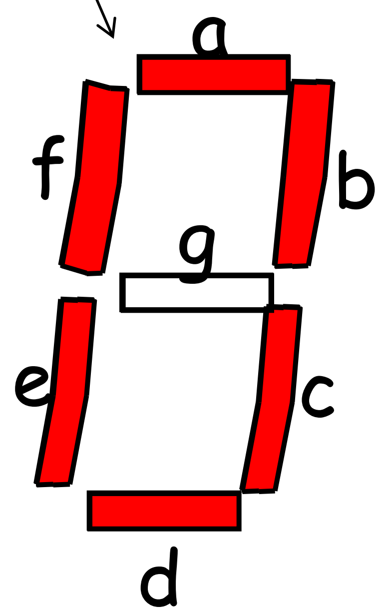
  - the # of 1s on the inputs equals to 1
- Constraints: Use only 2-input NAND gates!

Let us do it on the black board

Another Example: BCD-to-Seven-Segment Converter

- Converts BCD code to Seven-Segment code
  - Used to display numeric info on 7 segment display
  - Input is a 4-bit BCD code (w, x, y, z)
  - Output is a 7-bit code (a,b,c,d,e,f,g)





- Example:
  - Input: 0000_{BCD}
  - Output: 1111110 (a=b=c=d=e=f=1, g=0)

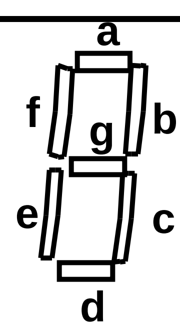
BCD-to-Seven-Segment (cont.)
Truth Table

Digit	wxyz	abcdefg
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	X011111
7	0111	11100X0

Digit	wxyz	abcdefg
8	1000	1111111
9	1001	111X011
	1010	XXXXXXX
	1011	XXXXXXX
	1100	XXXXXXX
	1101	XXXXXXX
	1110	XXXXXXX
	1111	XXXXXXX

??

Continue the design at home …



Design Procedure for Complex Circuits

- In general, digital systems are complex and sophisticated circuits
  - A circuit may consist of millions of gates!
- Impossible to design each and every circuit from scratch using the procedure you have just seen
- There is no formal procedure to design complex digital circuits!

How to design complex digital circuits?

Design Procedure for Complex Circuits

- Fortunately, complex digital circuits can be implemented as composition of smaller and simpler circuits
- These smaller and simpler circuits are fundamental and we call them basic functional blocks
- Basic functional blocks can be designed using the procedure you have just seen!
- Reuse basic functional blocks to design new circuits
- Use Design Hierarchy
- Use Computer-Aided Design (CAD) tools
  - Schematic Capture tools
  - Hardware Description Languages (HDL)
  - Logic Simulators
  - Logic Synthesizers

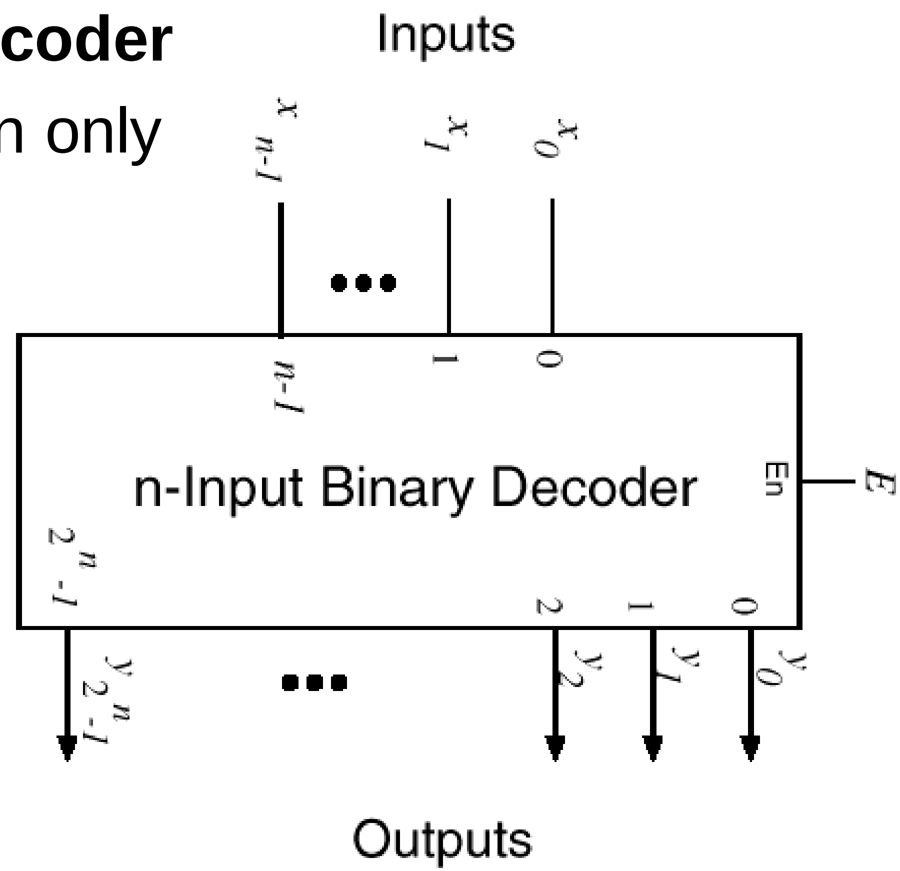
More details will be given at the Hands-on tutorials!

Basic Functional Blocks

- Combinational Functional Blocks
  - Logic Gates
  - Code Converters
  - **Binary Decoders** and Encoders
  - **Multiplexers** and Demultiplexers
  - Programmable Logic Arrays
  - Binary Adders and Subtractors
  - Binary Multipliers and Dividers
  - Shifters, Incrementors and Decrementors
- Sequential Functional Blocks
  - Flip-Flops and Latches
  - Registers and Counters
  - Sequencers
  - Micro-programmed Controllers
- Memories

Binary Decoder

- A combinational circuit that converts an **n**-bit binary number to a unique **2*ⁿ***-bit one-hot code!
  - Circuit is called ***n-to-2ⁿ* decoder**
  - For each input combination only one unique output is 1 (one-hot code)!

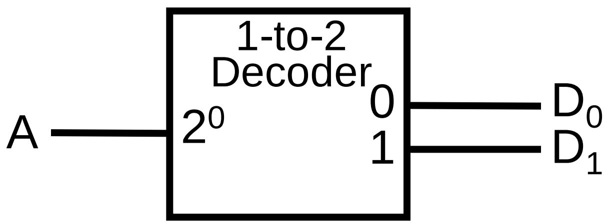


- Enable signal (**E**)
  if **E = 0** then
  all outputs are **0**
  else
  $y_j = f(x_0, x_1, \ldots, x_{n-1})$
  $(j = 0..2^n-1)$

1-to-2 Decoder

- 1-to-2 Decoder without Enable signal

Logic Symbol

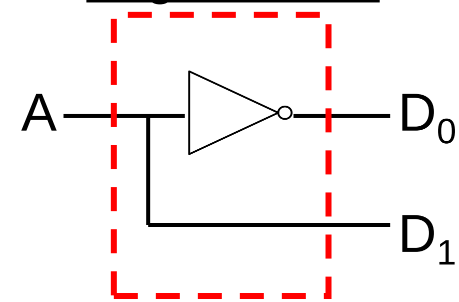


Truth Table and Equations

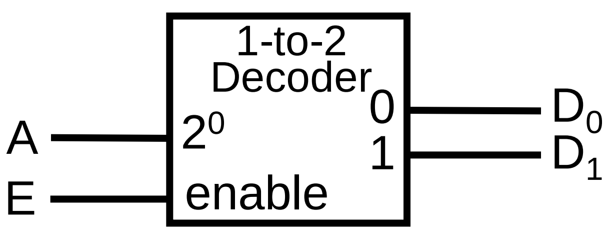
A		D_0	D_1
0		1	0
1		0	1

$D_0 = A'$
$D_1 = A$

Logic Circuit

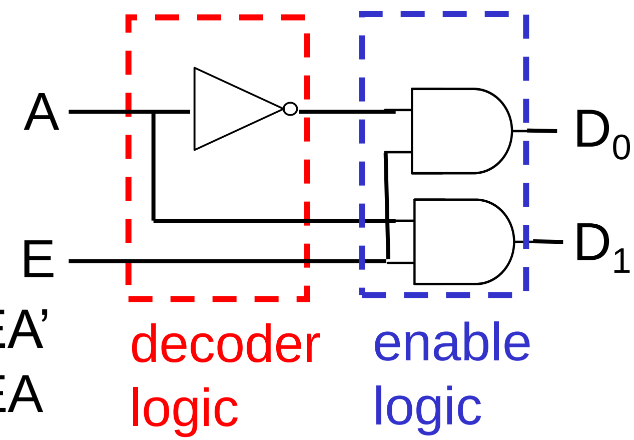


- 1-to-2 Decoder with Enable signal



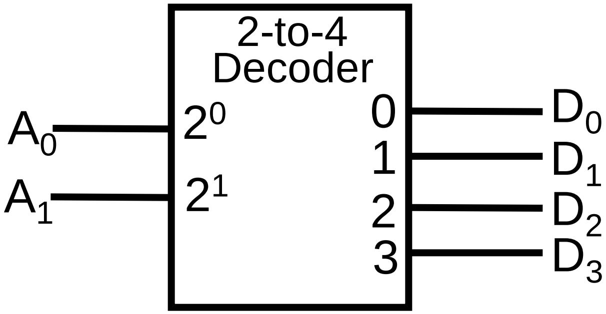
E	A		D_0	D_1
0	0		0	0
0	1		0	0
1	0		1	0
1	1		0	1

$D_0 = EA'$
$D_1 = EA$



2-to-4 Decoder

Logic Symbol



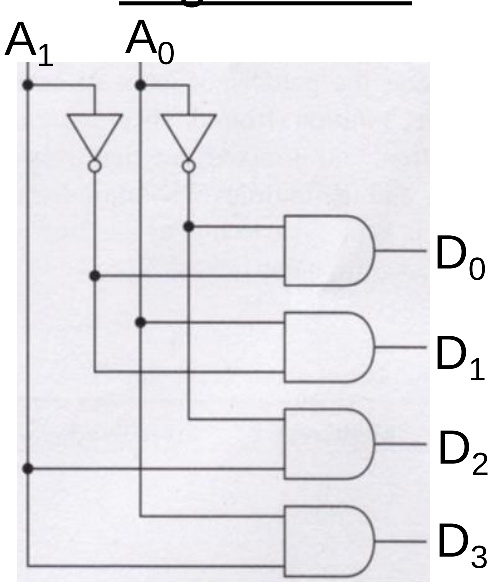
Truth Table and Equations

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

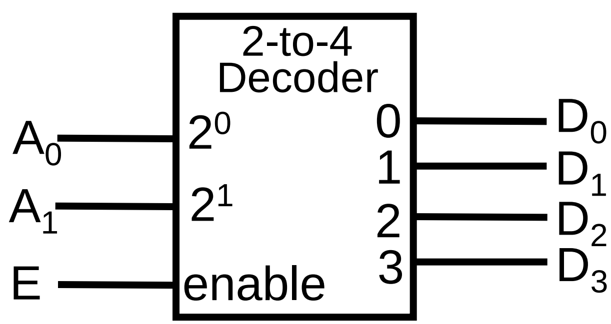
$D_0 = A_1'A_0'$
$D_1 = A_1'A_0$
$D_2 = A_1A_0'$
$D_3 = A_1A_0$

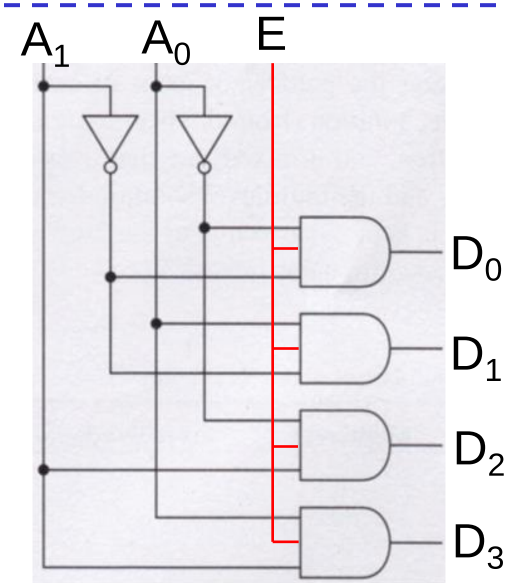
All minterms of 2 variables

Logic Circuit



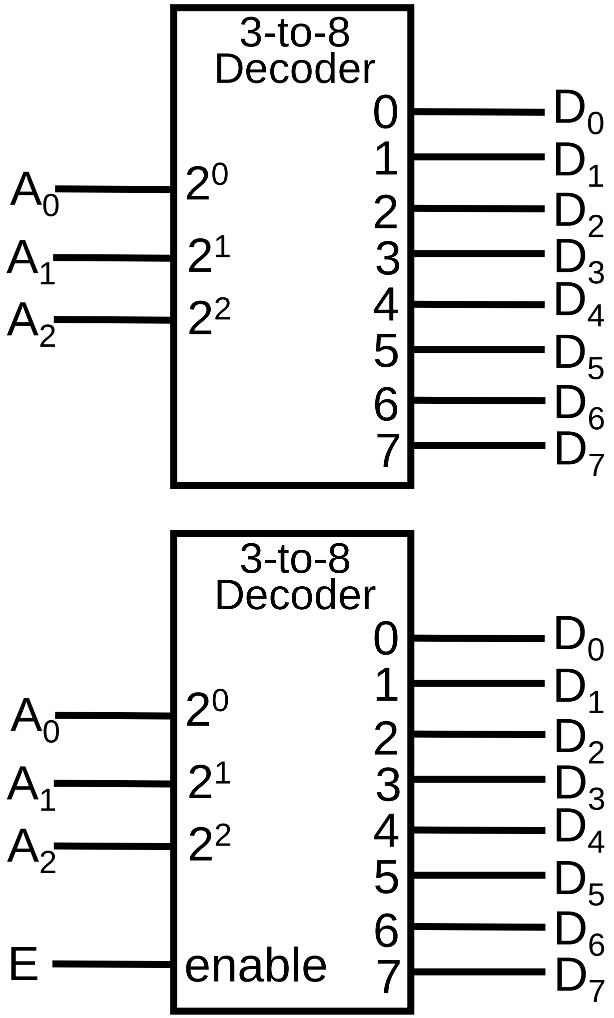
2-to-4 Decoder with Enable





3-to-8 Decoder

Logic Symbol

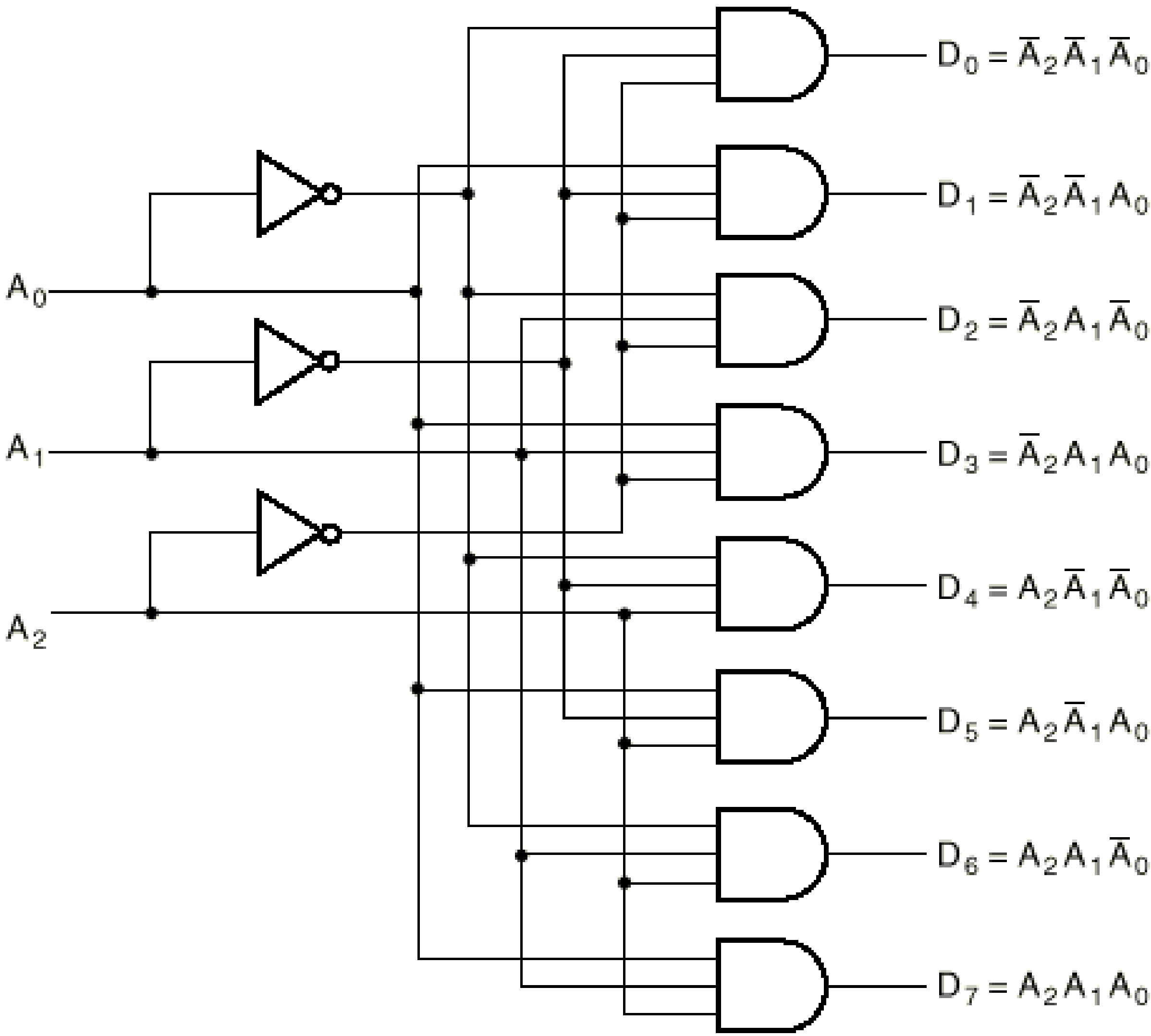


Truth Table

A_2	A_1	A_0		D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0		1	0	0	0	0	0	0	0
0	0	1		0	1	0	0	0	0	0	0
0	1	0		0	0	1	0	0	0	0	0
0	1	1		0	0	0	1	0	0	0	0
1	0	0		0	0	0	0	1	0	0	0
1	0	1		0	0	0	0	0	1	0	0
1	1	0		0	0	0	0	0	0	1	0
1	1	1		0	0	0	0	0	0	0	1

Notice: D0 to D7 represent all minterms of 3 variables.

3-to-8 Decoder (Logic Circuit)



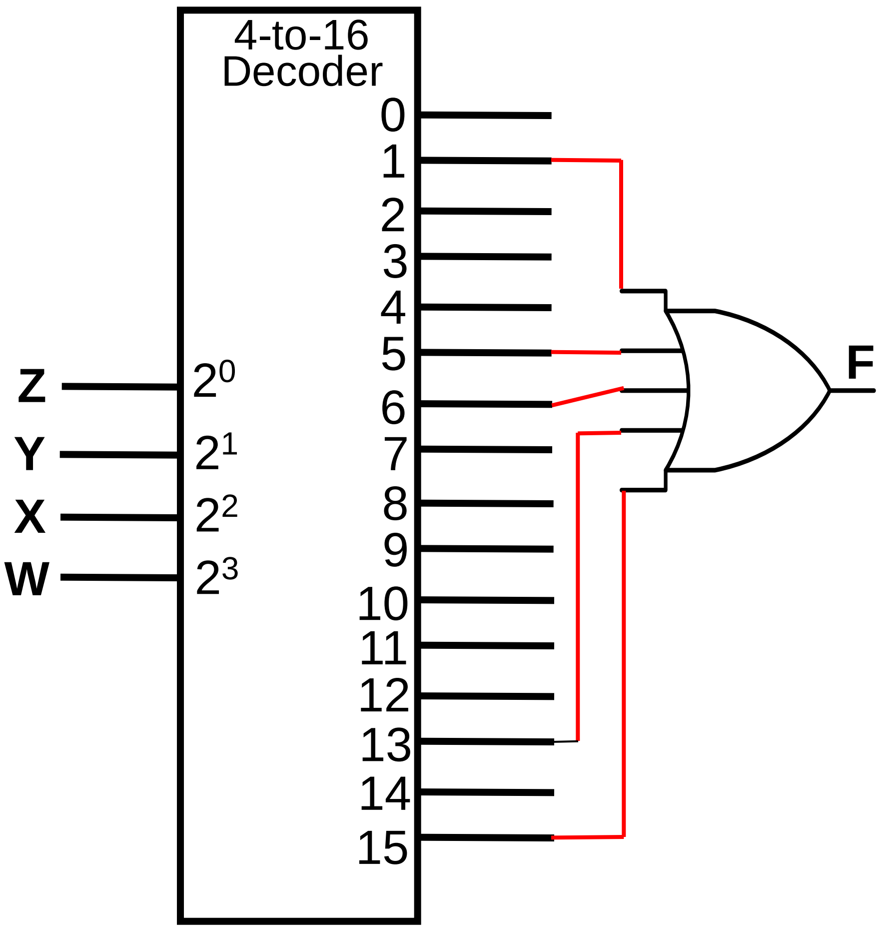
D0 to D7 are all minterms of 3 variables!

n-to-2^n Decoder (generalization)

- ***n*** inputs, A_0, A_1,..., A_{n-1}, are decoded into ***2^n*** outputs, D_0 through D_{2^n-1}.
- Each output $\mathbf{D}_j$ represents one of the minterms of the ***n*** input variables
- $\mathbf{D}_j = \mathbf{1}$ when the binary number $\mathbf{(A_{n-1}\ldots A_1A_0) =} j$
  - Shorthand: $D_j = m_j$
- The outputs are *mutually exclusive*
  - exactly one output has the value 1 at any time
  - the others are 0
- Due to the above properties, an arbitrary Boolean function of ***n*** variables can be implemented with *n*-to-2^n Decoder and OR gates!

Implementing Boolean Functions using Decoders

- Select outputs of a decoder that implement minterms included in the Boolean function
- Make a logic OR of the selected outputs
- Example:
  - Implement Boolean function F(W,X,Y,Z) = Σm(1,5,6,13,15)
  - F is a function of 4 variables → we use 4-to-16 Decoder
- Any combinational circuit can be constructed using decoders and OR gates! Why?



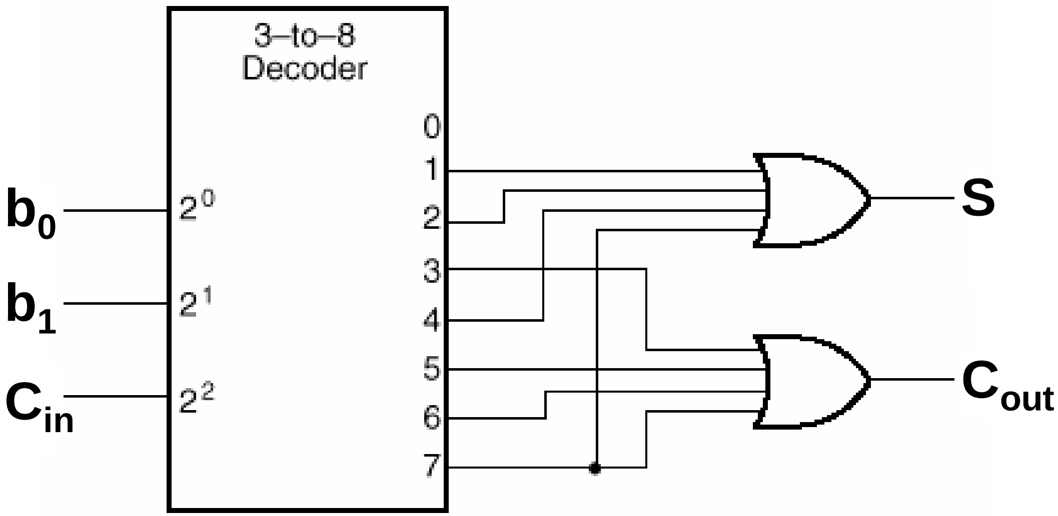
Another Example: Implementing a Binary Full Adder using a Decoder

- Binary Full Adder has 3 inputs and 2 outputs:
  - Inputs: two bits to be added (b_1 and b_0) and a carry-in (C_{in})
  - Outputs: sum ($S = b_0+b_1+C_{in}$) and carry-out (C_{out})
- Logic Functions:

$S(C_{in},b_1,b_0) = \Sigma m(1,2,4,7)$

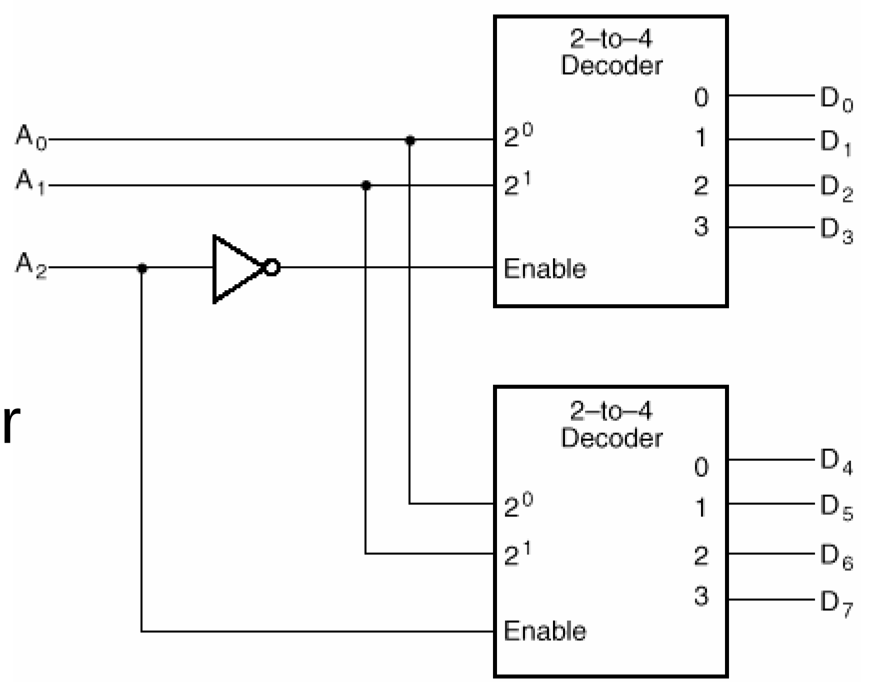
$C_{out}(C_{in},b_1,b_0) = \Sigma m(3,5,6,7)$

C_{in}	b_1	b_0		S	C_{out}
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

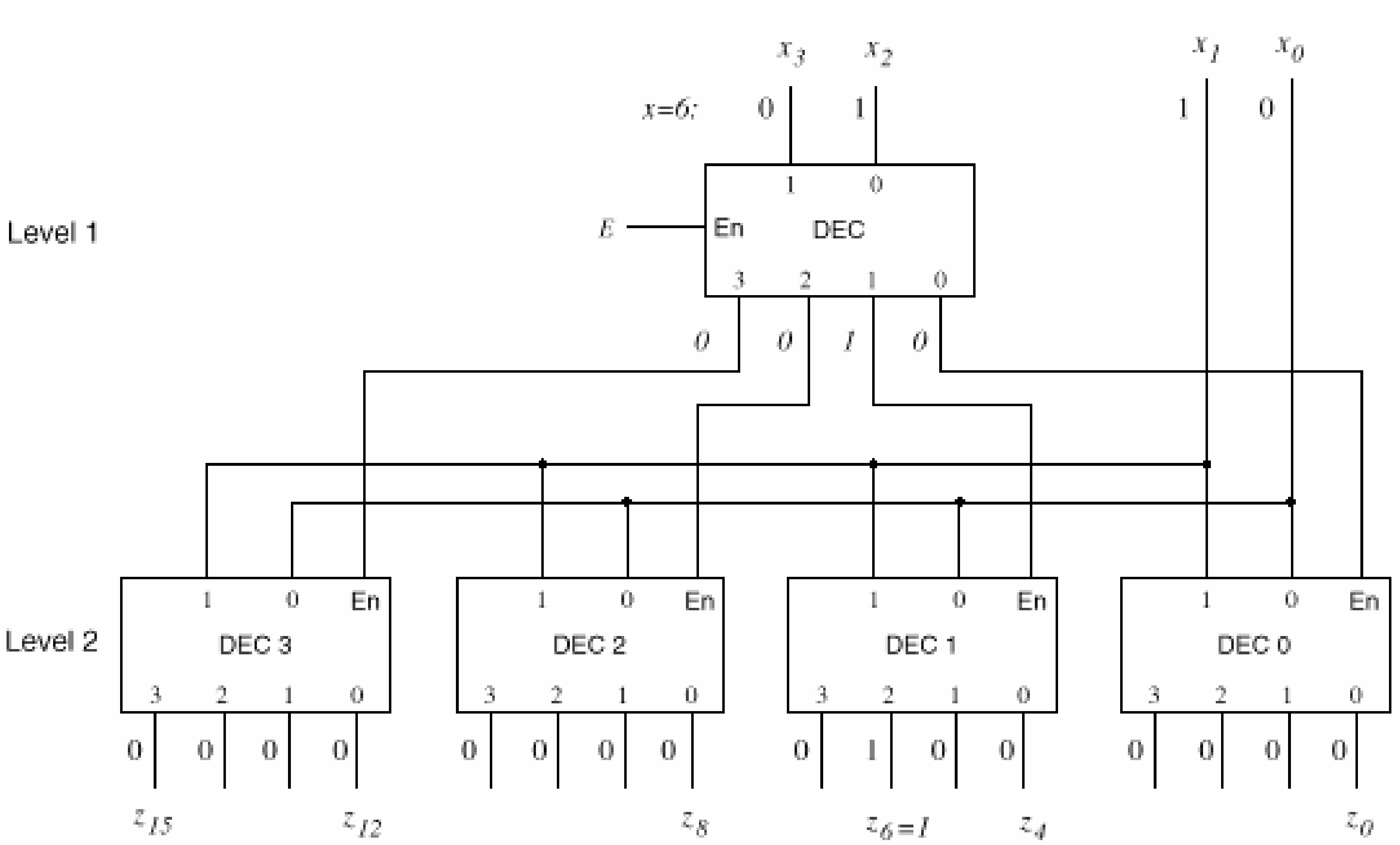


Decoder Expansions

- Larger decoders can be constructed using a number of smaller ones
- Use composition of smaller decoders to construct larger decoders
- Example:
  - Given: 2-to-4 decoders
  - Required: 3-to-8 decoder
  - Solution: Each decoder realizes half of the minterms Enable selects which decoder is active:
    - A2 = 0: enable top decoder
    - A2 = 1: enable bottom decoder

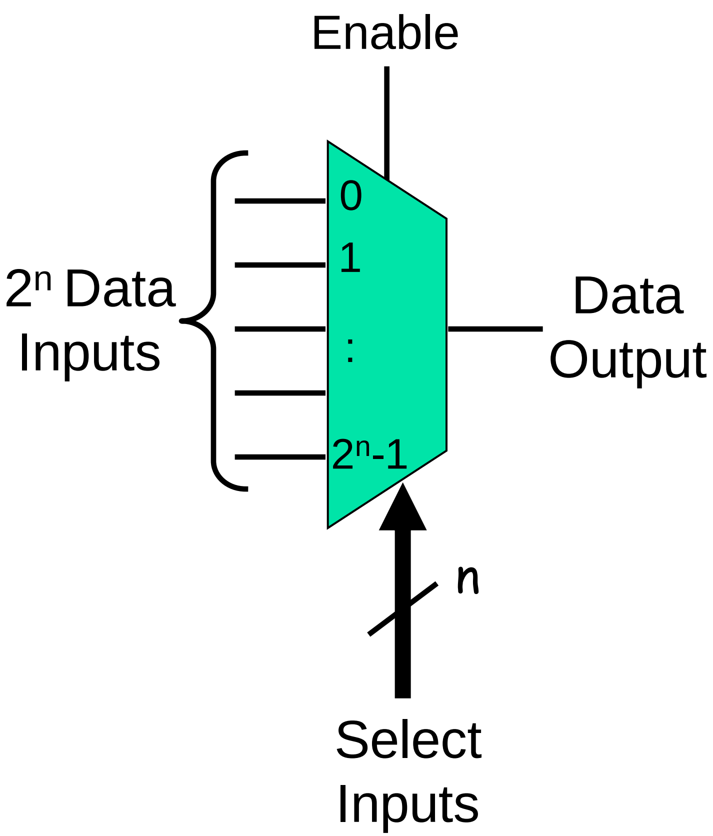


4-to-16 Decoder with 2-to-4 Decoders: Tree Composition of Decoders

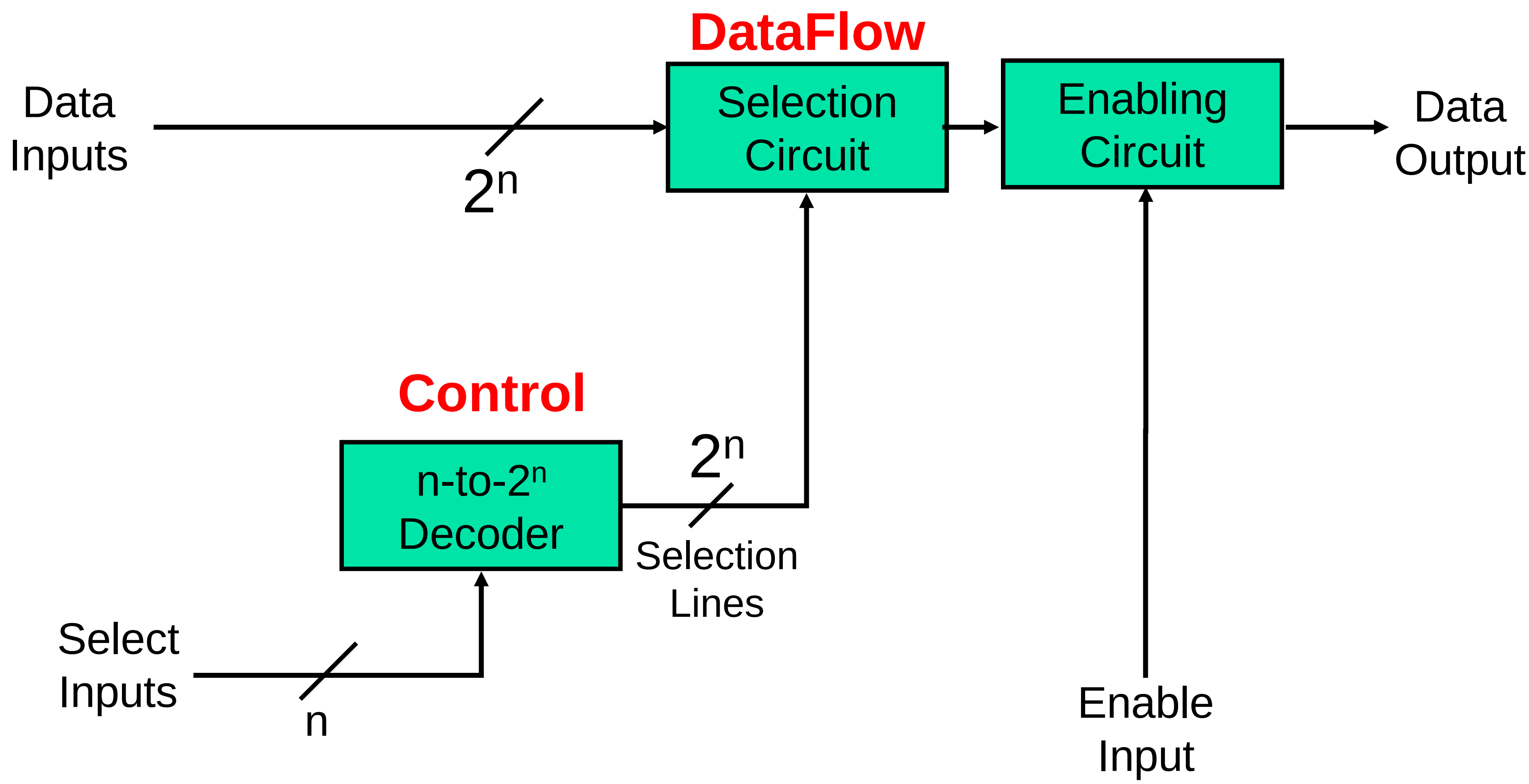


Multiplexer (MUXs)

- Selects one of many input data lines and directs it to a single output line
- Selection controlled by
  - set of input lines
  - whose # depends on the # of the data input lines
- A **2^n-to-1 multiplexer** has
  - **2^n** data input lines
  - **1** output line
  - ***n*** selection lines

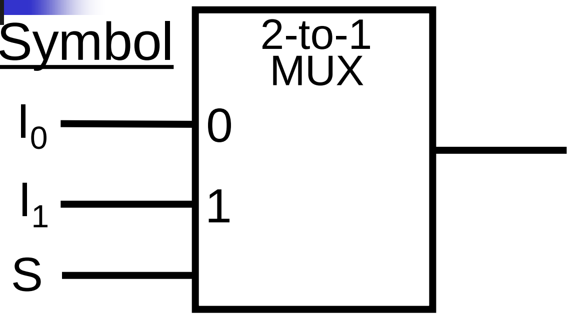


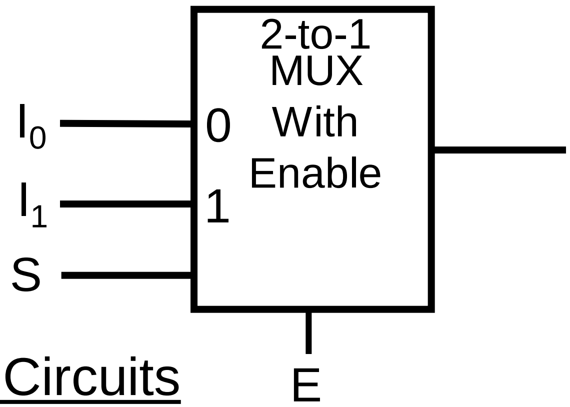
2^n-to-1 Multiplexer (General Structure)



2-to-1 Multiplexer

Logic Symbol



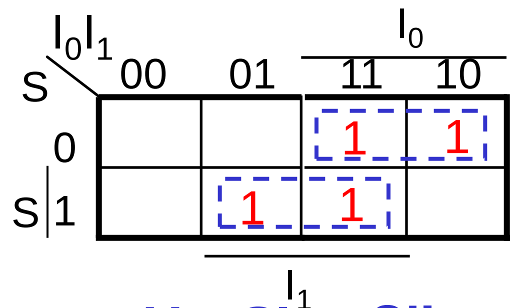


Truth Tables (compact and full)

S	Y
0	I_0
1	I_1

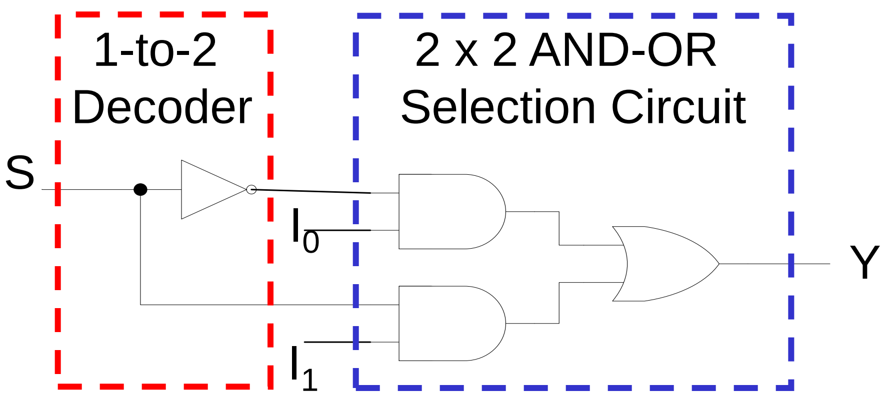
S	I_0	I_1		Y
0	0	0		0
0	0	1		0
0	1	0		1
0	1	1		1
1	0	0		0
1	0	1		1
1	1	0		0
1	1	1		1

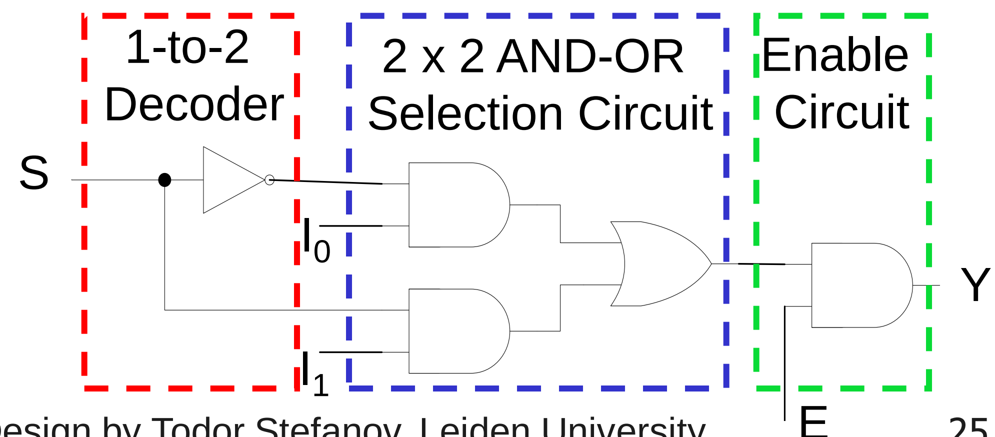
K-map and Equation



$$Y = SI_1 + S'I_0$$

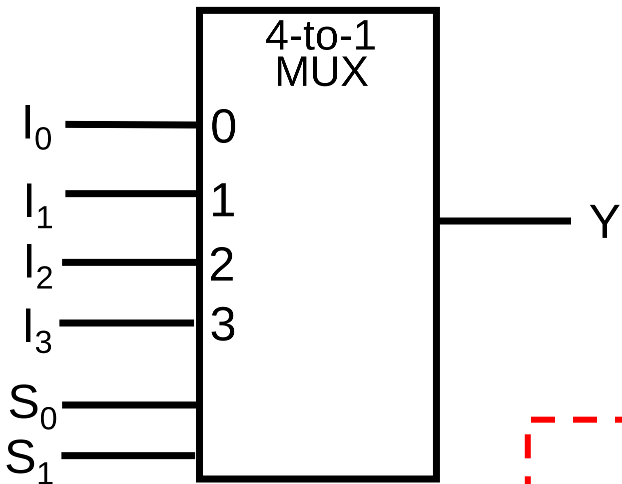
Logic Circuits





4-to-1 Multiplexer without Enable

Logic Symbol



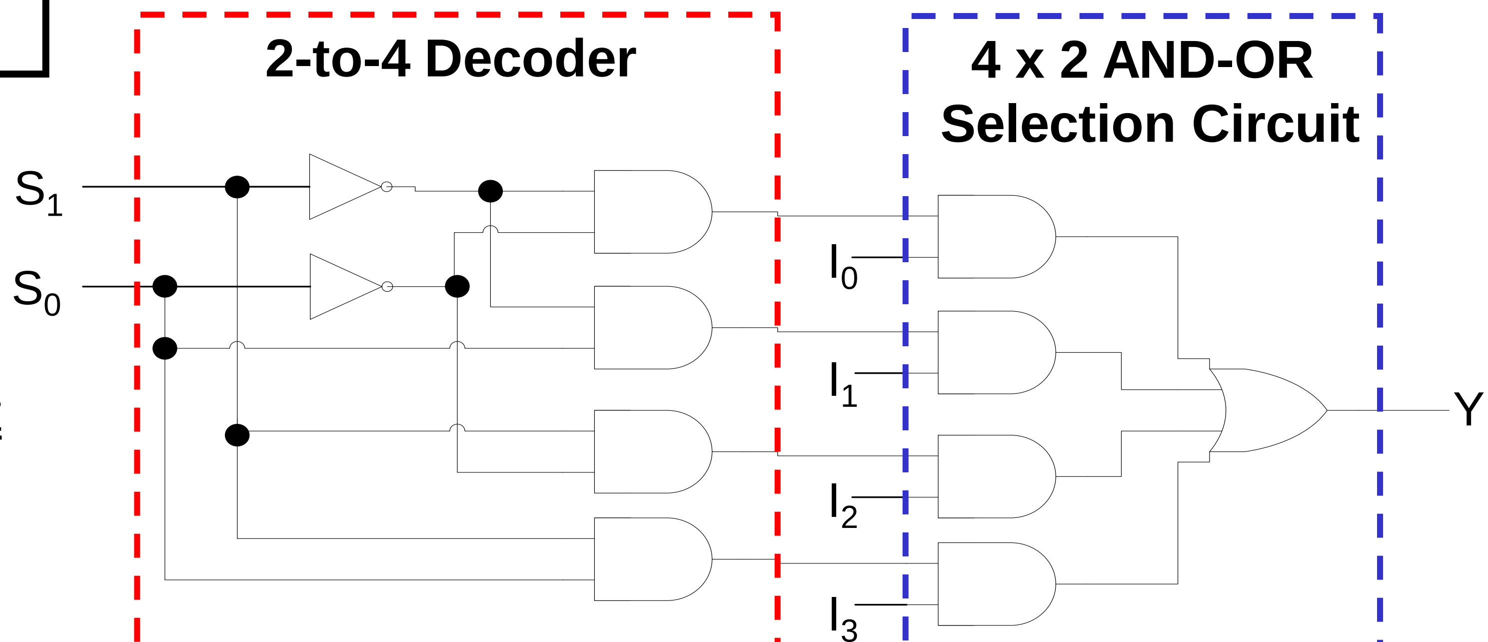
Compact Truth Tables

S_1	S_0		Y
0	0		I_0
0	1		I_1
1	0		I_2
1	1		I_3

Equation

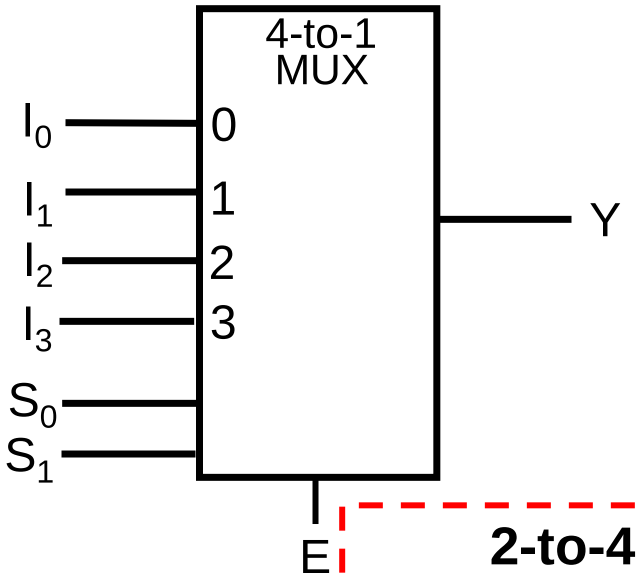
$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

Logic Circuit



4-to-1 Multiplexer with Enable

<u>Logic Symbol</u>



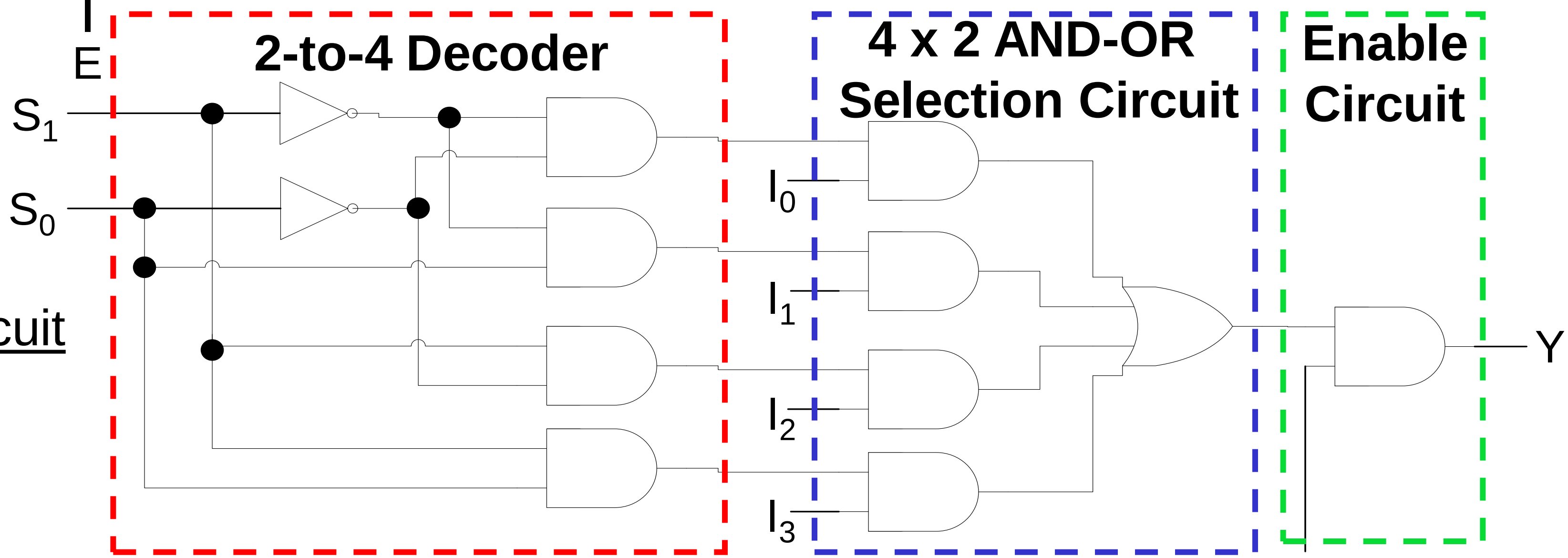
<u>Compact Truth Tables</u>

E	S_1	S_0		Y
1	0	0		I_0
1	0	1		I_1
1	1	0		I_2
1	1	1		I_3
0	x	x		0

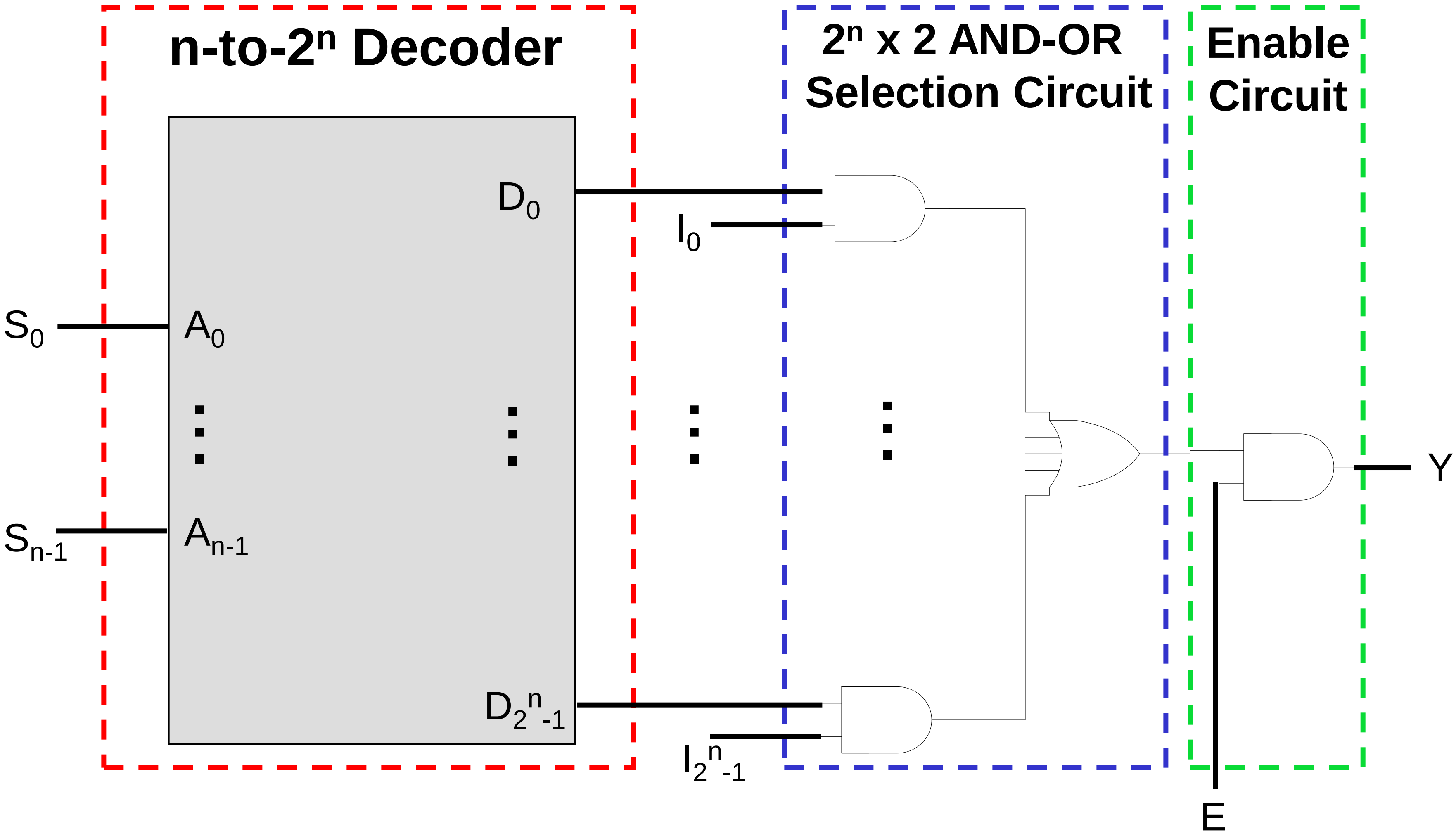
<u>Equation</u>

$$Y = E \bullet (S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3)$$

<u>Logic Circuit</u>

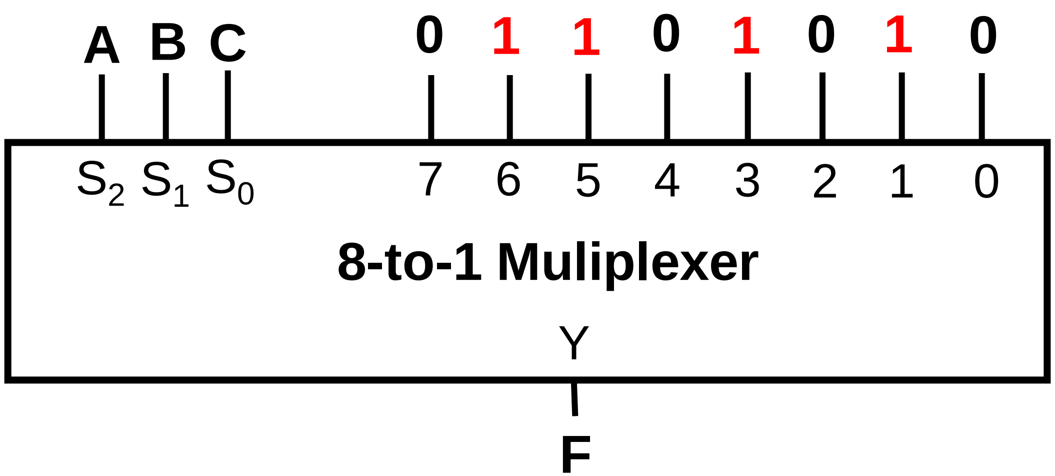


2^n-to-1 Multiplexer



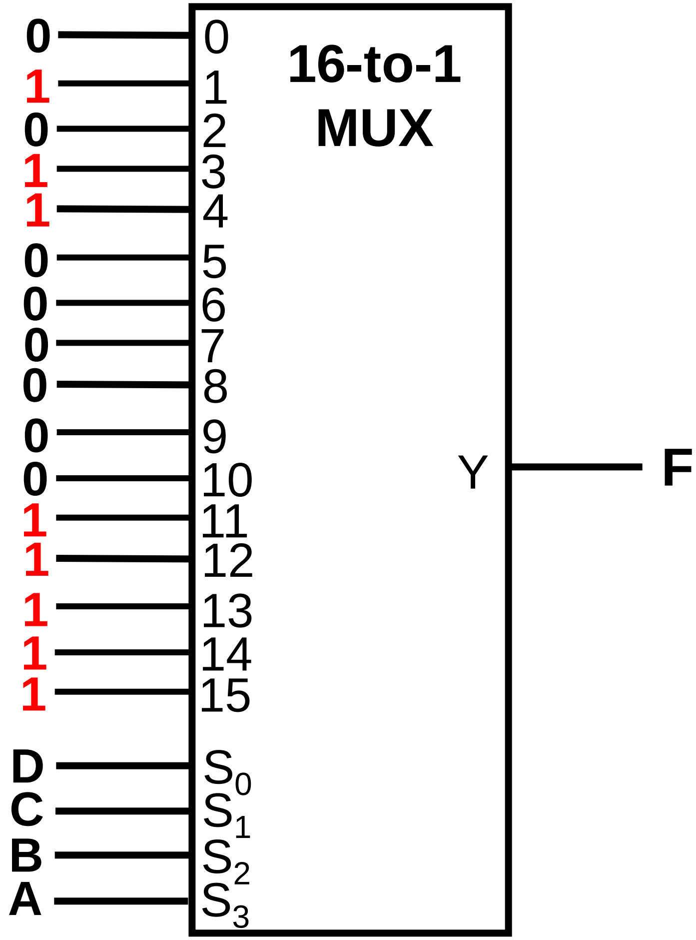
Implementing Boolean Functions using Multiplexers

- Any Boolean function of n variables can be implemented using a 2^n-to-1 Multiplexer. Why?
  - Multiplexer is basically a decoder with outputs ORed together!
  - SELECT signals generate the minterms of the function
  - The data inputs identify which minterms are to be combined with an OR
- Example: Consider function $F(A,B,C) = \sum m(1,3,5,6)$
  - It has 3 variables, therefore we can implement it with 8-to-1 MUX



Another Example

- Consider function F(A,B,C,D) specified by the truth table on the right-hand side
- F has 4 variables → we use 16-to-1 MUX



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

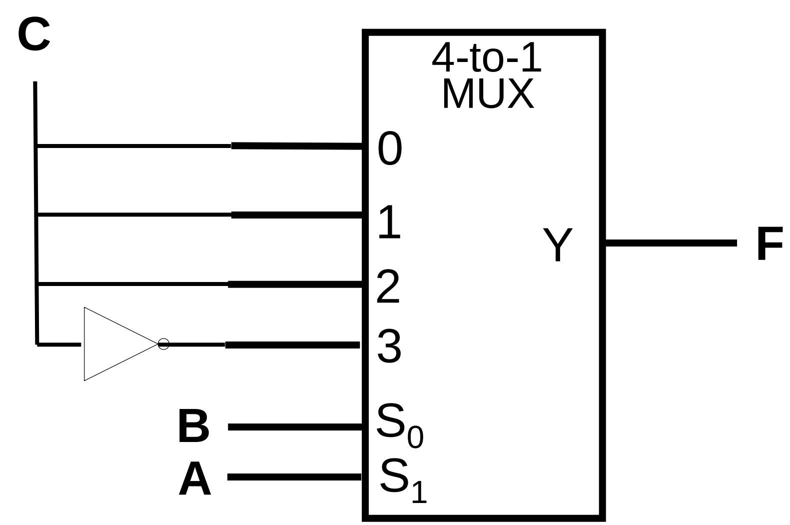
Efficient Method for Implementing Boolean Functions using Multiplexers

- We have seen that implementing a function of *n* variables with *2^n-to-1* MUX is straightforward.
- However, there exist more efficient method where any function of **n** variables can be implemented with 2^{n-1}-to-1 MUX. Consider an arbitrary function $F(X_1,X_2,\ldots,X_n)$:
  - We need a $\mathbf{2^{n-1}}$ line MUX with ***n*-1** select lines.
  - Enumerate function as a truth table with consistent ordering of variables, i.e., $\mathbf{X_1,X_2,\ldots,X_n}$.
  - Attach the most significant ***n*-1** variables to the ***n*-1** select lines, i.e., $\mathbf{X_1,X_2,\ldots,X_{n-1}}$
  - Examine pairs of adjacent rows. The least significant variable in each pair is $\mathbf{X_n = 0}$ and $\mathbf{X_n = 1}$.
  - Determine whether the function output **F** for the $(X_1,X_2,\ldots,X_{n-1},0)$ and $(X_1,X_2,\ldots,X_{n-1},1)$ combination is (0,0), (0,1), (1,0), or (1,1).
  - Attach 0, X_n, X_n', or 1 to the data input corresponding to $(X_1,X_2,\ldots,X_{n-1})$ respectively.

Example

- Again, consider function $F(A,B,C) = \sum m(1,3,5,6)$
- It has 3 variables
- We can implement it using **4-to-1 MUX** instead of 8-to-1 MUX.

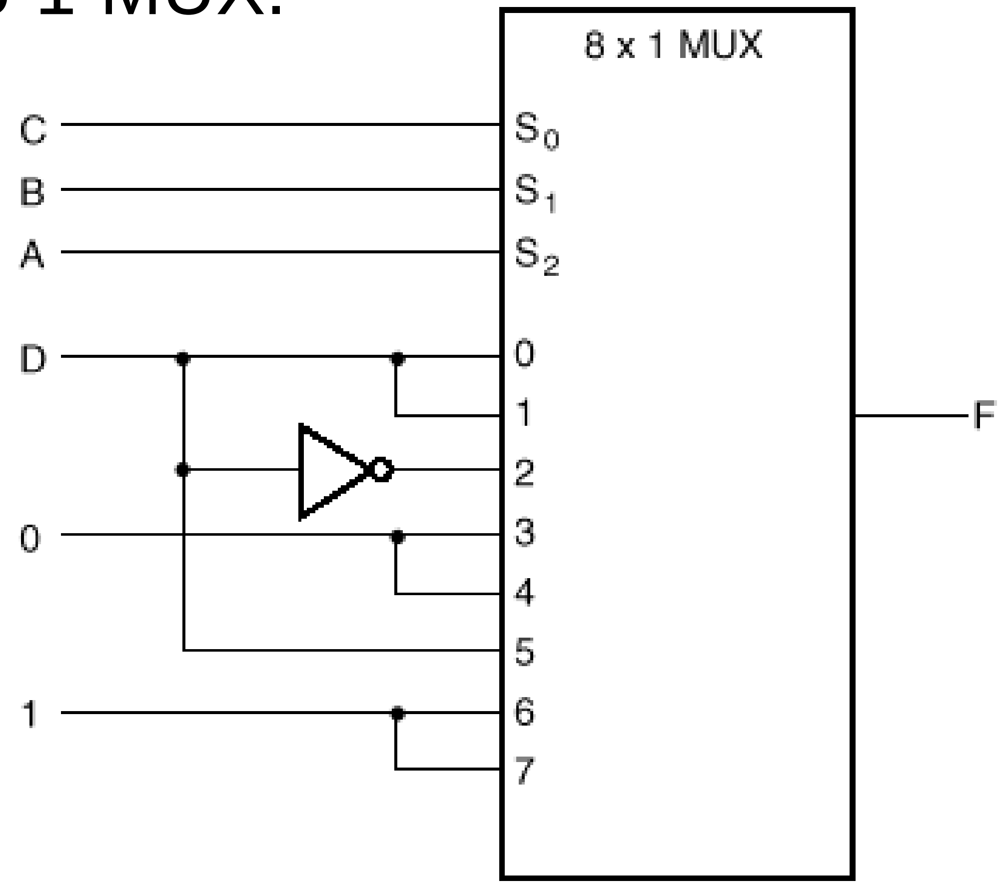
A	B	C	F	
0	0	0	0	**F = C**
0	0	1	1	
0	1	0	0	**F = C**
0	1	1	1	
1	0	0	0	**F = C**
1	0	1	1	
1	1	0	1	**F = C’**
1	1	1	0	



Another Example

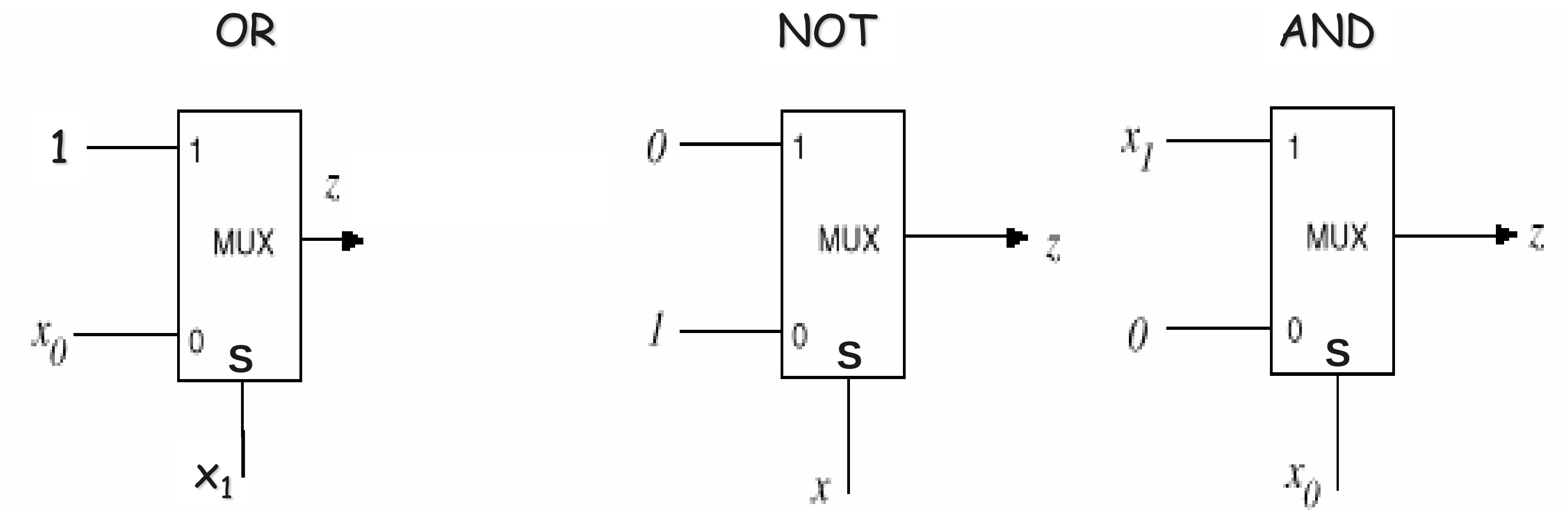
- Again, consider function F(A,B,C,D) specified by the truth table below. F has 4 variables → we use 8-to-1 MUX instead of 16-to-1 MUX.

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	



MUX as a Universal Gate

- We can construct OR, AND, and NOT gates using **2-to-1 MUX**
- Thus, 2-to-1 MUX is a universal gate!
- Recall the equation of 2-to-1 MUX: $Z = S \bullet I_1 + S' \bullet I_0$



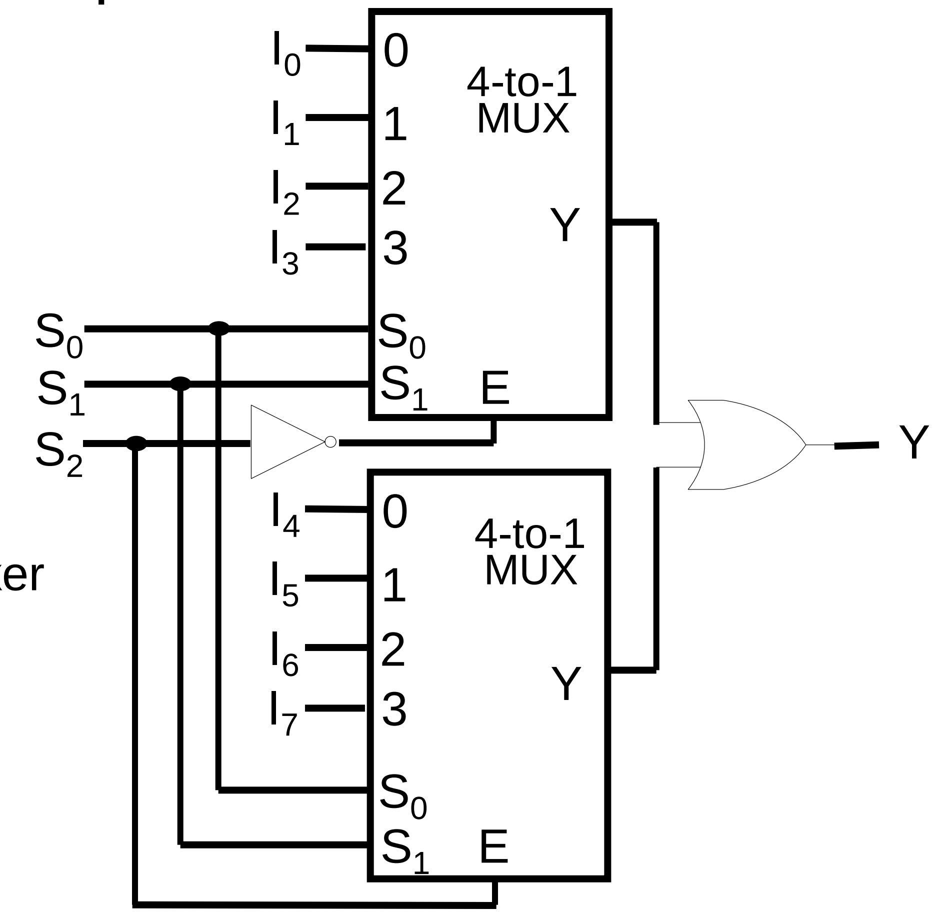
$$z = x_1 \bullet 1 + x_1' \bullet x_0$$
$$= x_1 + x_1' \bullet x_0$$
$$= (x_1 + x_1') \bullet (x_1 + x_0)$$
$$= 1 \bullet (x_1 + x_0) = x_1 + x_0$$

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

$$z = x_0 \bullet x_1 + x_0' \bullet 0$$
$$= x_0 \bullet x_1$$

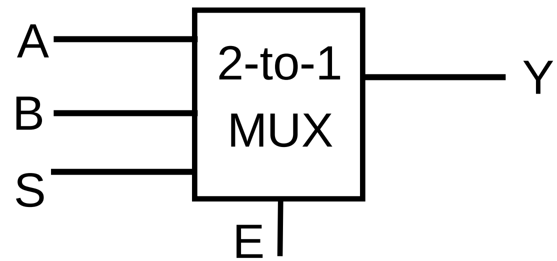
Multiplexer Expansions

- Larger multiplexers can be constructed using a number of smaller ones
- Use composition of smaller multiplexers
- Example:
  - Given: 4-to-1 multiplexers
  - Required: 8-to-1 multiplexer
  - Solution: Each multiplexer selects half of the data inputs. Enable signal selects which multiplexer is active:
    - S_2 = 0: enable top multiplexer
    - S_2 = 1: enable bottom multiplexer

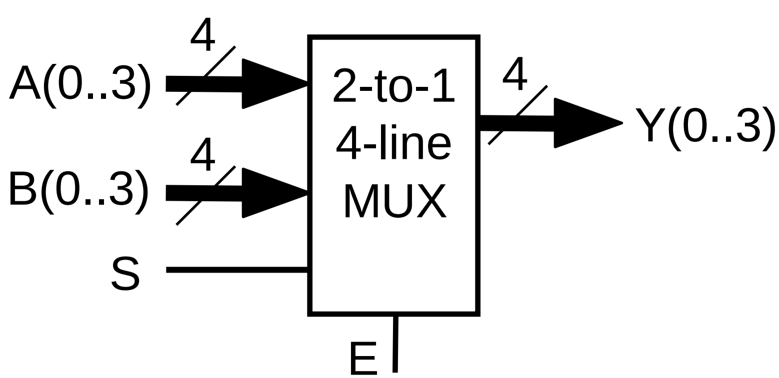


Multiplexer Expansions (cont.)

- Until now, we have examined 1-bit data inputs selected by a MUX



- What if we want to select m-bit data/words?
  - Example: MUX that selects between 2 sets of 4-bit inputs



E	S		Y(0..3)
1	0		A(0..3)
1	1		B(0..3)
0	x		0000

- How to construct this 2-to-1 4-line MUX?

Example: 2-to-1 4-line Multiplexer

- Uses four 2-to-1 MUXs with common select (S) and enable (E)
- Select line chooses between A_i's and B_i's. The selected four-wire digital signal is sent to the Y_i's
- Enable line turns MUX on and off (E=1 is on)