CS2100 - L20 - Medium-Scale-Integration (MSI) Components

Week 11 + 12

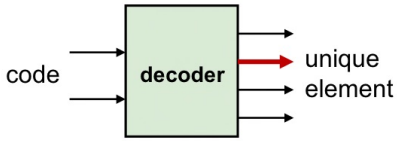
20.1 - MSI components

- Decoders
- Encoders
- Demultiplexers
- Multiplexers

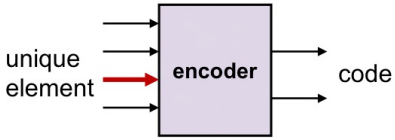
20.2 - Applications

20.1 - MSI components

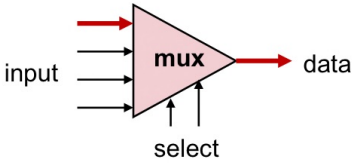
Introduction to Four MSI Circuits



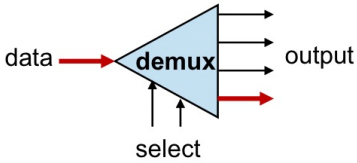
Decoder



Encoder



Multiplexer



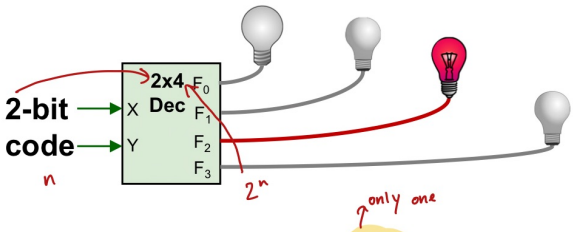
Demultiplexer

Decoders

Decoder : Overview

- A **N-bit** Binary values (codes) can represent up to **2^N** unique elements:
  - 2-bit = {00, 01, 10, 11}
- Decoder **takes the N-bit code** and **convert it into one of the 2^N unique output**
  - 2^N is the upperbound as some of the N-bit code can be unused
- Known as ***n*-to-*m*-line** decoder, or simply ***n*:*m*** or ***n*×*m*** decoder (***m*** $\leq$ **2^n**)

2 x 4 Decoder: Example

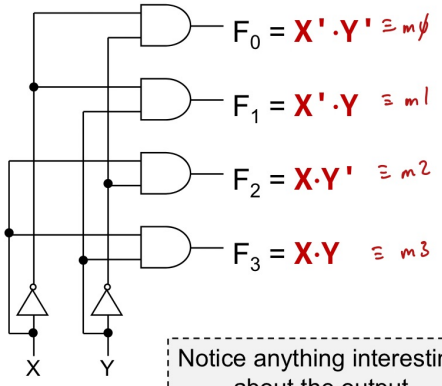


- Depending on the **2-bit code**, one of the light bulbs will be **switched on**
- Truth table:

X	Y	F_0	F_1	F_2	F_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

Decoder: Internal Wiring

X	Y	F_0	F_1	F_2	F_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

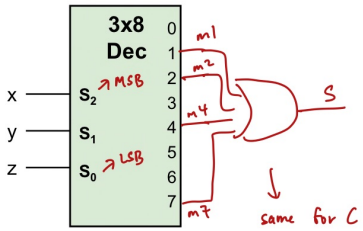


Notice anything interesting about the output expressions?

Decoder: Implementing Boolean Function

- **Key Idea:**
  - Output of decoder corresponds to **minterms**
- Example: **Full adder**

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



S(x, y, z) = Σ **m**(**1**, **2**, **4**, **7**)

C(x, y, z) = Σ **m**(**3**, **5**, **6**, **7**)

Decoder: Implementing Boolean Function

- General Scheme:
  - A Boolean function in sum-of-minterms form can be implemented by:
  - Decoder to generate the minterms with an **OR** gate to form the sum
- So, any combinational circuit with **n** inputs and **m** outputs can be implemented by:
  - **one [n:2^n** decoder] + **m** [**OR** gates]
- Good for circuit with many outputs, and each function is expressed with just a few minterms

Decoders with Enable(2/2)

- The previous decoder has a **one-enable** control signal
- Most MSI decoders is **zero-enable** instead, usually denoted by E' or $\bar{E}$:

  ➔ The decoder is enabled when the signal is **zero** (low)

E	X	Y	F_0	F_1	F_2	F_3
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
0	X	X	0	0	0	0

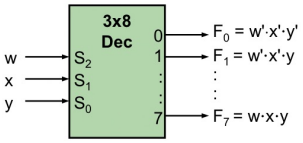
Decoder with 1-enable

E'	X	Y	F_0	F_1	F_2	F_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

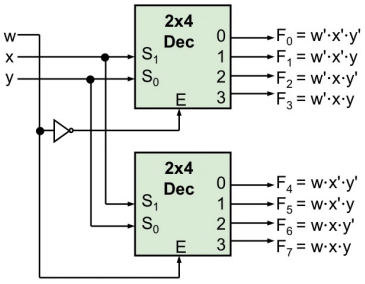
Decoder with 0-enable

Larger Decoders (1/4)

- **Larger decoders** can be constructed from **smaller decoders**



- Example: A **3×8** decoder can be built from:
  - Two **2×4** decoders with one-enable AND
  - An inverter



Larger Decoders (4/4)

- Decoder Variants:

Enable:	Output:
• **Zero-enable**: Use '0' to activate decoder • **One-enable**: Use '1'to activate decoder	• **Normal (active-high)**: Output is '1' when selected • **Negated (active-low)**: Output is '0' when selected

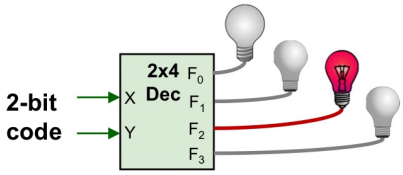
- Exercise: What modifications should be made to provide an ENABLE input for the 3×8 decoder and the 4×16 decoder created in the previous two slides?

- Exercise: How to construct a 4×16 decoder using five 2×4 decoders with enable?

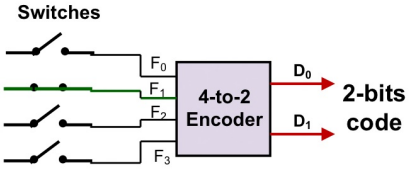
Encoders

Encoders: Overview

- Encoding is the converse of decoding



Decoder:
Input **n-bit code** activates **one output among the 2^n possibilities**



Encoder:
The position of the single active input line among 2^n possibilities is coded as a **n-bit code**

4-to-2 Encoders: Implementation

- Truth table:

F_0	F_1	F_2	F_3	D_1	D_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1
0	0	0	0	X	X
0	0	1	1	X	X
0	1	0	1	X	X
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

also garbage ∵ 00 used to represent F_0 being on

garbage input / output

- With K-map, we obtain:

  $D_0 = F1 + F3$
  $D_1 = F2 + F3$

- Circuit (Use **OR gate**):

Priority Encoders (1/2)

- Suppose we assign **priority to each input**:
  - If two or more inputs or equal to 1, the input with the highest priority **takes precedence**
  - All inputs of 0 is considered **invalid**
- Example: **4-to-2 Priority Encoder**:

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Priority Encoders (2/2)

- Be careful of the '**X**'s in the ***input***
  - Don't interpret it as "either 1 or 0",
  - Instead it is actually "for **both** 1 and 0"
  - Essentially a "compressed" truth table

for inputs: ~~don't care~~ for all

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

$x = D_3 + D_2 \cdot D_3'$
$= D_3 + D_2$
$y = D_3 + D_1 \cdot D_2' \cdot D_3'$
$= D_3 + D_1 \cdot D_2'$
$z = D_0 + D_1 + D_2 + D_3$

- **Exercise:** Obtain the simplified expressions for **x**, **y** and **V**.

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
1	1	0	0	0	1	1
0	0	1	0	1	0	1
0	1	1	0	1	0	1
1	0	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	1	1	1	1
0	0	1	1	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Demultiplexers

Demultiplexers (1/2)

- Given:
  - **One input data line**
  - **N selection lines**

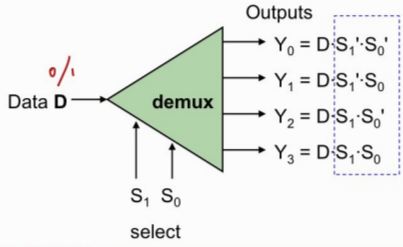
  steers/routes

  only one at any point

  ➔ **Demultiplexer** directs the input data to the **selected output line** among 2^N possibilities

  can be 0/1

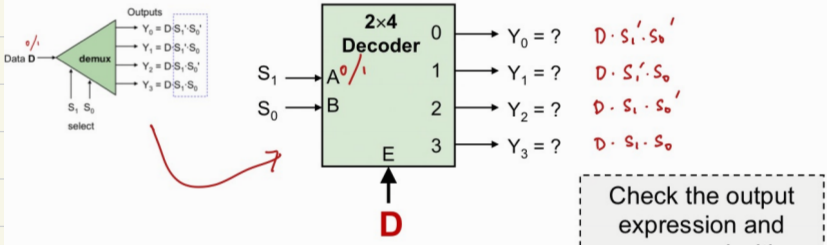
- Example (1-to-4 Demultiplexer):



S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Demultiplexers (2/2)

- It turns out that the **demultiplexer** circuit is actually identical to a **decoder with enable**:

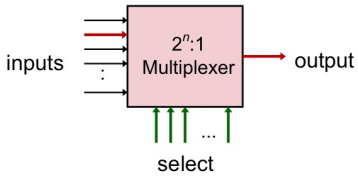


Check the output expression and compared with Demultiplexer's

Multiplexers

Multiplexers: Overview

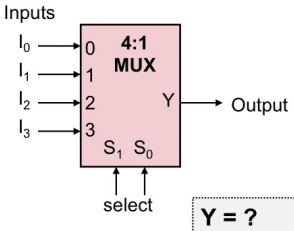
- Given:
  - **2^n** input lines
  - **n** selection lines
  - **One** output line
- Using ***n*** selection lines, **multiplexer** steers one of 2^n inputs to the output line
  - also known as a ***data selector***

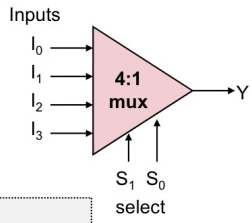


4-to-1 Multiplexers: Truth Table

I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

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



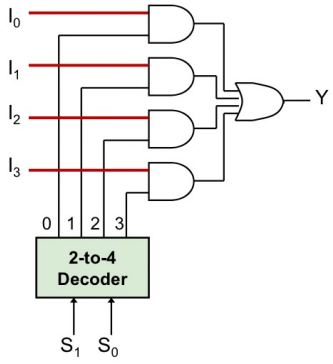


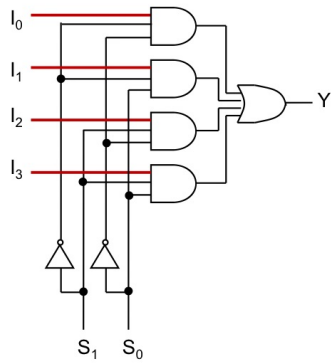
Y = ?

Multiplexers: Construction with Decoder

- Multiplexer's Output =

  "sum of the (product of *data lines* and *selection lines*)"

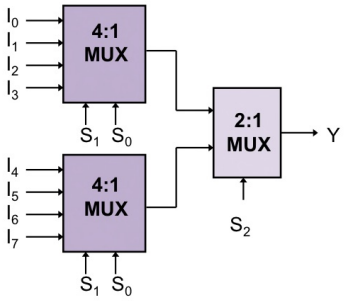




Larger Multiplexers: Overview

- Larger multiplexers can be constructed from smaller ones
- An **8-to-1 multiplexer** example:
  - Note the **placement of selector lines**

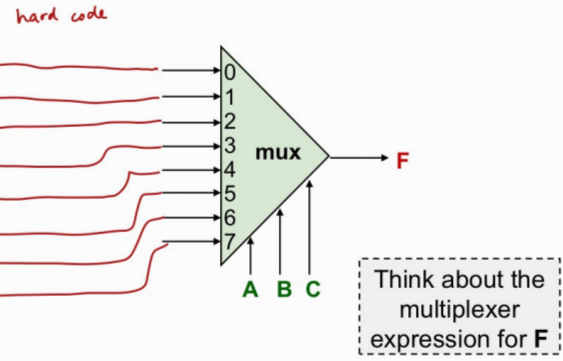
S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7



Implementing Functions: Overview

- Boolean functions can be implemented using multiplexers
- Try for F(A,B,C) = Σ m(1,3,5,6)

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



Implementing Functions: Generalization

- A 2^n-to-1 multiplexer can implement a Boolean function of *n* input variables, as follows:

1. Express in sum-of-minterms form. Example:

   F(A,B,C) = A'·B'·C + A'·B·C + A·B'·C + A·B·C'

   = Σ m(1,3,5,6)

2. Connect *n* variables to the *n* selection lines

3. Put a '1' on a data line if it is a minterm of the function, or '0' otherwise

Using Smaller Multiplexers: **Procedure** (1/4)

1. Express Boolean function in sum-of-minterms form.
   Example: F(A,B,C) = Σ m(0,1,3,6)

2. **Reserve one variable for the input lines** of multiplexer, and use the rest for selection lines
   - We reserve the least significant bit (e.g. C) as input line in previous example

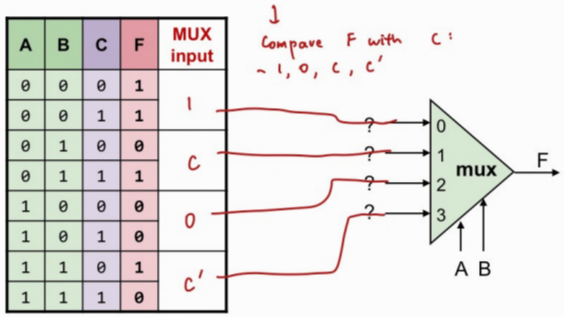
   Example:

   Use C for input lines; A and B for selection lines

Using Smaller Multiplexers: **Procedure** (2/4)

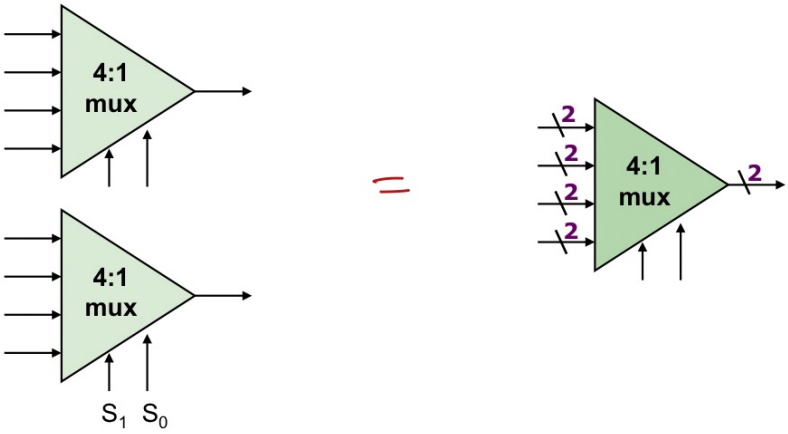
3. Use truth table:
   a. Group inputs by **selection line values**
   b. Determine multiplexer inputs by **comparing input line and function output** for corresponding selection line value groups

A	B	C	F	MUX input
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	C′
1	1	1	0	

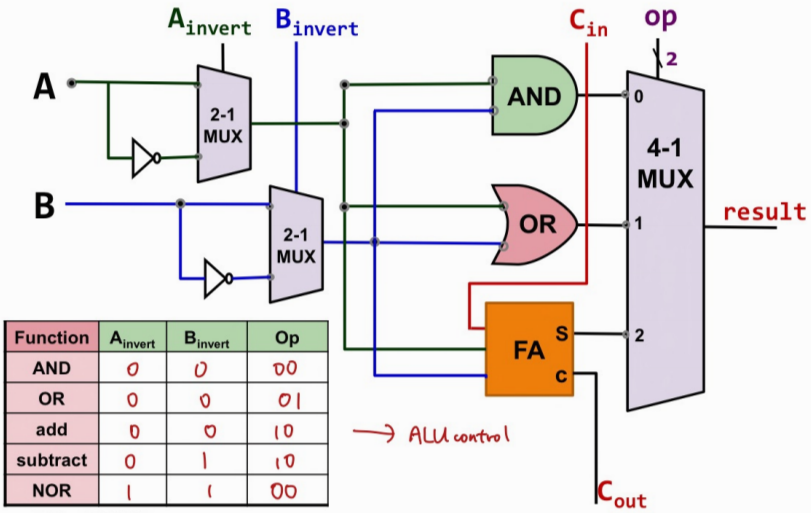


Multiplexer: **Selecting Multi-Bit Data?**

- How to select between 4 input data **W**, **X**, **Y**, **Z**, each a 2-bit value, i.e. W_1W_0, X_1X_0, etc?
  - The output F_1F_0 is similarly a 2-bit value



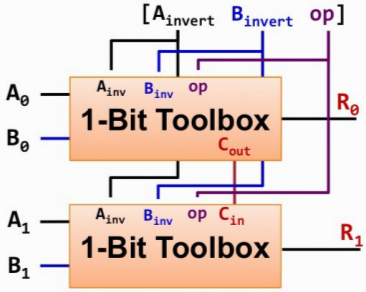
V4.0: Summary



Function	A_{invert}	B_{invert}	Op
AND	0	0	00
OR	0	0	01
add	0	0	10
subtract	0	1	10
NOR	1	1	00

"Swiss Army Knife" V4.0 – **Multi-Bit?**

- By connecting multiple 1-bit toolbox circuits, we can have a multi-purpose toolbox for multi bit inputs!
  - Below show a simplified **2-bit version:**



General Idea:

- Control Signals shared between all 1-bit toolbox
- C_{in} connected to C_{out} from previous bit position
- Each 1-bit toolbox at position **x** take a pair of input A_x, B_x