Unit 8
Fundamental Digital Building Blocks: Decoders & Multiplexers
Checkers / Decoders

• Recall
  – AND gates output '1' for only a single combination
  – OR gates output '0' for only a single combination
  – Inputs (inverted or non-inverted) determine which combination is checked for
  – We say that gate is "checking for" or "decoding" a specific combination
Motivation

• Just like there are patterns and structures that occur commonly in nature, there are several common logic structures that occur over and over again in digital circuits
  – Decoders, Muxes, Adders, Registers

• In addition, we design hardware using a hierarchical approach
  – We design a small component using basic logic gates (e.g. a 1-bit mux)
  – We build a large component by interconnecting many copies of the small component + a few extra gates (e.g. a 32-bit mux)
  – We build chips by interconnecting many large components (e.g. a router)
  – Each components is truly made out of many gates but we the design process is faster and easier by using hierarchy

• Let's look at a few common components
  – We'll start by describing the behavior of the component and then determine what gates are inside
DECODERS
Decoders

- A decoder is a building block that:
  - Takes in an n-bit binary number as input
  - Decodes that binary number and activates the corresponding output
  - Individual outputs for ALL $2^n$ input combinations

There are gates inside to implement each output

1 output for each combination of the input number
Decoders

- A decoder is a building block that:
  - Takes a binary number as input
  - Decodes that binary number and activates the corresponding output
  - Put in 6=110, Output 6 activates ('1')
  - Put in 5=101, Output 5 activates ('1')

Only that numbered output is activated
Decoders

- A decoder is a building block that:
  - Takes a binary number as input
  - Decodes that binary number and activates the corresponding output
  - Put in 6=110, Output 6 activates ('1')
  - Put in 5=101, Output 5 activates ('1')

Only that numbered output is activated
Decoder Sizes

• A decoder w/ an **n-bit input** has **2^n outputs**
  – 1 output for every combination of the n-bit input

![Diagram of a 2-to-4 Decoder](image)

![Diagram of a 3-to-8 Decoder](image)
Exercise

• Complete the design of a 2-to-4 decoder

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Building Decoders

3-bit number [A2:A0]

- Checker for 000
- Checker for 001
- Checker for 010
- Checker for 011
- Checker for 100
- Checker for 101
- Checker for 110
- Checker for 111

O0
O1
O2
O3
O4
O5
O6
O7

A0
A1
A2

O0
O1
O2
O3
O4
O5
O6
O7
Vending Machine Example

Assuming the keypad produces a 4-bit numeric output, add logic to produce the release signals for each of the 16 vending items.

Consider any problems with this design.
Enables

- In a normal decoder exactly one output is active at all times.
- It may be undesirable to always have an active output.
- We can add an extra input (called an enable) that can independently force all the outputs to their inactive values.

2-to-4 Decoder

<table>
<thead>
<tr>
<th>X (MSB)</th>
<th>Y</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

One output will always be active

Enable

Will force all outputs to 0 when E = 0 (i.e. not enabled)
Enables

When E=0, inputs is ignored

When E=1, inputs will cause the appropriate output to go active

Since E=0, all outputs = 0

Since E=1, outputs will function normally
Enables

- Enables can be implemented by connecting it to each AND gate of the decoder.

When $E=0$, $0$ AND anything = $0$

When $E=1$, $1$ AND anything = that anything, which was the normal decoding logic
Multiplexers

- Multiplexers are one of the most common digital circuits
- Anatomy: \( n \) data inputs, \( \log_2 n \) select bits, 1 output
- A multiplexer (“mux” for short) selects one data input and passes it to the output

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_0 )</td>
<td>( Y )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>i0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>i1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>i2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>i3</td>
</tr>
</tbody>
</table>
Multiplexers

Thus, input 2 = C is selected and passed to the output

Select bits = $10_2 = 2_{10}$.

As long as the select bits are $10_2 = 2$, whatever bit value appears on input 2 is copied to the output, same as if we had just wired input 2 directly to the output.
Multiplexers

Thus, input 0 = A is selected and passed to the output.

1. Select bits = 00₂ = 0₁₀.

<table>
<thead>
<tr>
<th>S₁</th>
<th>S₀</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>i₀</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>i₁</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>i₂</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>i₃</td>
</tr>
</tbody>
</table>
Exercise: Build a 4-to-1 mux

• Complete the 4-to-1 mux to the right by drawing wires between the 2-to-4 decode and the AND gates
Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.

\[
S_1S_0 = 01_2
\]
Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.

\[ S_1 S_0 = 11_2 \]
Recall Using T1/T2

- 1\textsuperscript{st} Level of AND gates act as barriers only passing 1 channel
- OR gates combines 3 streams of 0’s with the 1 channel that got passed (i.e. ICH1)
- 2\textsuperscript{nd} Level of AND gates passes the channel to only the selected output

Essentially this logic forms a 4-to-1 mux where one level of gates blocks all but 1 and then the OR gate combines all signals

AND:
1 AND ICH1 = ICH1
0 AND ICH1 = 0
2-to-1 Multiplexers

Thus, input 1 = B is selected and passed to the output.

Select bits = $1_2 = 1_{10}$.
Building a 2-to-1 Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.
Building Large Muxes

• Similar to a tournament of sports teams
  – Many teams enter and then are narrowed down to 1 winner
  – In each round winners play winners

Stage 1
Stage 2
Stage 3
Final output

Railroad Switch Station
Design an 8-to-1 mux with 2-to-1 Muxes
Cascading Muxes

- Use several small muxes to build large ones

Rules

1. Arrange the muxes in stages (based on necessary number of inputs in 1st stage)
2. Outputs of one stage feed to inputs of the next until only 1 final output
3. All muxes in a stage connect to the same group of select bits
   - Usually, LSB connects to first stage
   - MSB connect to last stage
Building a 4-to-1 Mux

Stage 1

Rule 1: Outputs from stage 1 connect to inputs of stage 2

Rule 2: LSB $S_0$ connect to all muxes in first stage. MSB $S_1$ connects to all muxes in second stage

4-to-1 mux built w/ 2-to-1 muxes
Building a 4-to-1 Mux

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$D_0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$D_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$D_2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$D_3$</td>
</tr>
</tbody>
</table>

Walk through an example:

$S_1S_0 = 01$
Building a 4-to-1 Mux

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$D_0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$D_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$D_2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$D_3$</td>
</tr>
</tbody>
</table>

Walk through an example:

$S_1S_0 = 01$

$S_0 = 1$ narrows our choices down to $D_1$ and $D_3$
Building a 4-to-1 Mux

Walk through an example:

\[ S_1 S_0 = 01 \]

<table>
<thead>
<tr>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( D_0 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( D_1 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( D_2 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( D_3 )</td>
</tr>
</tbody>
</table>

\( S_1 = 0 \) selects our final choice, \( D_1 \).
Device vs. System Labels

- When using hierarchy (i.e. building blocks) to design a circuit be sure to show both device and system labels
  - **Device Labels**: Signal names used inside the block
    - Placeholder names the designer of the block uses to indicate which input/output is which to the outside user (Names may vary; read the manual)
  - **System labels**: Signal names used outside the block
    - Actual signals from the circuit being built
    - Can have the same name as the device label if such a signal name exists at the outside level

Analogy: **Formal and Actual parameters in software function calls**
1. a and b are like device labels and indicate the names used inside a block.
2. x and y are like system labels and represent the actual values to be used.

```c
int div(int i0, int i1)
{ int t = i0/i1;
  return t;
}
int main()
{ int d0=10, d1=2;
  int s = div(d0,d1);
}
```

Device Labels: Indicate which input/output is which inside the block.

System Labels: Actual signals from the circuit being built.
Exercise

• Sketch how you could build a 16-to-1 mux with 4-to-1 muxes? 8-to-1 and 2-to1 muxes?
Exercise

• Create a 3-to-1 mux using 2-to-1 muxes
  – Inputs: I0, I1, I2 and select bits S1, S0
  – Output: Y

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>D2</td>
</tr>
</tbody>
</table>
Select-bit Ordering

- If we connect the select bits as shown to build an 8-to-1 mux, show how to label the inputs (i0-i7) so that the correct input is passed based on the binary value of S2:S0.
Another way to multiplex

**TRI-STATE GATES**
Typical Logic Gate

- Gates can output two values: 0 & 1
  - Logic ‘1’ (Vdd = 3V or 5V), or Logic ‘0’ (Vss = GND)
  - But they are ALWAYS outputting something!!!
- Analogy: a sink faucet
  - 2 possibilities: Hot (‘1’) or Cold (‘0’)
- In a real circuit, inputs cause **EITHER** a pathway from output to VDD **OR** VSS

![Diagram of logic gate with transistors for high and low voltage passage]
Output Connections

• Can we connect the output of two logic gates together?
• No! Possible short circuit (static, low-resistance pathway from Vdd to GND)
• We call this situation “bus contention”
Tri-State Buffers

• Normal digital gates can output two values: 0 & 1
  1. Logic 0 = 0 volts
  2. Logic 1 = 5 volts

• Tristate buffers can output a third value:
  3. Z = High-Impedance = "Floating"
      (no connection to any voltage source...infinite resistance)

• Analogy: a sink faucet
  – 3 possibilities:
    1.) Hot water,
    2.) Cold water,
    3.) NO water

\[ \text{Hot Water} = \text{Logic 1} \]
\[ \text{Cold Water} = \text{Logic 0} \]
\[ \text{NO Water} = \text{Z (High-Impedance)} \]
Tri-State Buffers

- Tri-state buffers have an extra enable input
- When disabled, output is said to be at high impedance (a.k.a. Z)
  - High Impedance is equivalent to no connection (i.e. floating output) or an infinite resistance
  - It's like a brick wall between the output and any connection to source
- When enabled, normal buffer

<table>
<thead>
<tr>
<th>En</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Tri-State Buffers

- We use tri-state buffers to share one output amongst several sources
- Rule: Only 1 buffer enabled at a time
Tri-State Buffers

• We use tri-state buffers to share one output amongst several sources
• Rule: Only 1 buffer enabled at a time
• When 1 buffer enabled, its output overpowers the Z’s (no connection) from the other gates

Select source 1 to pass its data

output of 0 overpowers the Z

Disabled buffers output ‘Z’
Enable Polarity

- **Side note**: Some tri-states are designed to pass the input (be enabled) when the enable is 0 (rather than 1)
  - A inversion bubble is shown at the enable input to indicate the "low" polarity needed to enable the tristate

<table>
<thead>
<tr>
<th>En</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>En</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Communication Connections

• Multiple entities need to communicate
• We could use
  – Point-to-point connections
  – A shared bus (set of wires)
Bidirectional Bus

- 1 transmitter (otherwise bus contention)
- N receivers
- Each device can send (though 1 at a time) or receive
Tri-State Gates

- Advantage: don’t have to know in advance how many devices will be connected together
  - Tri-State gates give us the option of connecting together the outputs of many devices without requiring a circuit to multiplex many signals into one
- Just have to make sure only one is enabled (output active) at any one time.