Unit 7

Fundamental Digital Building Blocks: Decoders & Multiplexers
CHECKERS / DECODERS
Gates

- Gates can have more than 2 inputs but the functions stay the same
  - **AND** = output = 1 if ALL inputs are 1
    - Outputs 1 for only 1 input combination
  - **OR** = output = 1 if ANY input is 1
    - Outputs 0 for only 1 input combination

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3-input **AND**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3-input **OR**
Checkers / Decoders

- An AND gate only outputs ‘1’ for 1 combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the AND gate as “checking” or “decoding” a specific combination and outputting a ‘1’ when it matches.
Checkers / Decoders

• Place inverters at the input of the AND gates such that
  – F produces ‘1’ only for input combination \{x,y,z\} = \{010\}
  – G produces ‘1’ only for input combination \{x,y,z\} = \{110\}

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

AND gate decoding (checking for) combination 010

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

AND gate decoding (checking for) combination 110
Checkers / Decoders

• An OR gate only outputs ‘0’ for 1 combination
  – That combination can be changed by adding inverters to the inputs
  – We can think of the OR gate as “checking” or “decoding” a specific combination and outputting a ‘0’ when it matches.
Decoder Exercise

- Compilers translate software to instructions that tell the processor to ADD, LOAD from Memory, Store to Memory, etc.
- These instructions are binary codes
- The processor must decode the instruction
- Create an AND gate decoder for each instruction type in the table that will produce '1' when that instruction is about to be executed

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>6-bit OPCODE OP[5:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>001000</td>
</tr>
<tr>
<td>LOAD</td>
<td>100011</td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
</tr>
</tbody>
</table>
| BRANCH           | 000100              

ADD

LOAD

STORE
A full 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’)

**Binary #6**

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’)

Binary #5

Only that numbered output is activated
Decoder Sizes

• A decoder w/ an \textbf{n-bit input} has \(2^n\) outputs
  – 1 output for every combination of the n-bit input

\begin{itemize}
  \item \(n\) inputs \(\rightarrow\) \(2^n\) outputs

\begin{table}[h]
\begin{tabular}{|c|}
\hline
\textbf{Decoder}\tabularnewline
\hline
2-to-4 \tabularnewline
\hline
3-to-8 \tabularnewline
\hline
\end{tabular}
\end{table}
Exercise

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

Output: 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](image)

One output will always be active

![Enable Diagram](image)

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

When E=0, inputs is ignored

Since E=0, all outputs = 0

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

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

\[
\begin{array}{c|c|c}
S_1 & S_0 & Y \\
0 & 0 & \text{i0} \\
0 & 1 & \text{i1} \\
1 & 0 & \text{i2} \\
1 & 1 & \text{i3} \\
\end{array}
\]
Thus, input $2 = C$ is selected and passed to the output.

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

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

Select bits $= 00_2 = 0_{10}$.

4-to-1 Mux, 32-bit wide

<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>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>
Thus, input $I_1 = B$ is selected and passed to the output.

Select bits $= 1_2 = 1_{10}$.

<table>
<thead>
<tr>
<th>$S$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i0</td>
</tr>
<tr>
<td>1</td>
<td>I1</td>
</tr>
</tbody>
</table>
Recall Using T1/T2

- 1st 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)
- 2nd 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 ICHx = ICHx
0 AND ICHx = 0

OR: 0 + ICH1 + 0 + 0 = ICH1

AND: 1 AND ICH1 = ICH1
0 AND ICH1 = 0
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.

\[ S_1S_0 = 11_2 \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>i0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>i3</td>
<td></td>
</tr>
</tbody>
</table>
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.
Building Wide Muxes

- So far muxes only have single bit inputs...
  - \( I_0 \) is only 1-bit
  - \( I_1 \) is only 1-bit
- What if we still want to select between 2 inputs but now each input is a 4-bit number
- Use a 4-bit wide 2-to-1 mux

When we select \( I_0 \) or \( I_1 \) we want all 4-bits of that input to be passed.
Building Wide Muxes

- Use one mux per "lane" (bit)
  - To build a 4-bit wide 2-to-1 mux, use 4 separate 2-to-1 muxes

- Operation:
  - When S=0, all muxes pass their \( I_0 \) inputs which means all the A bits get through
  - When S=1, all muxes pass their \( I_1 \) inputs which means all the B bits get through

- In general, to build an **m-bit wide** (i.e. m-lane) n-to-1 mux, use **m individual n-to-1 muxes**
Multiplexers

Thus, input 0 = A[31:0] is selected and passed to the output.

Select bits = 00₂ = 0₁₀.
Multiplexers

2-to-1 Mux, 32-bit wide mux

Thus, input 1 = B[31:0] is selected and passed to the output

Select bits = $1_2 = 1_{10}$.
Exercise

• How many 1-bit wide muxes and of what size would you need to build a 4-to-1, 8-bit wide mux (i.e. there are 4 numbers: $W[7:0]$, $X[7:0]$, $Y[7:0]$ and $Z[7:0]$ and you must select one)

• How many 1-bit wide muxes and of what size would you need to build a 8-to-1, 2-bit wide mux?
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
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

4-to-1 mux built w/ 2-to-1 muxes

Stage 1

Stage 2

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
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

<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>
<tr>
<td>1</td>
<td>1</td>
<td>D₃</td>
</tr>
</tbody>
</table>

Walk through an example:

S₁S₀ = 01

S₁ = 0 selects our final choice, D₁
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
    - Placeholders to indicate which input/output is which to the outside user
  - **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);
}
```
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

<table>
<thead>
<tr>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>Y</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>i4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>i5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>i7</td>
<td></td>
</tr>
</tbody>
</table>