Module #6 – MSI Logic

• Topics
  A. MSI Logic Definition / Functional Simulation
  B. Decoders
  C. Encoders
  D. Multiplexers
  E. Demultiplexers
  F. Adders

• Textbook Reading Assignments
  - 6.4-6.5, 6.7, 6.10

• Practice Problems
  - VHDL Inverter Design & Simulation (see M6_HW1 handout)
  - VHDL 2:4 Decoder Design & Simulation (see M6_HW2 handout)

• Graded Components of this Module
  - 2 homework, 2 discussion, 1 quiz
    (homeworks will be uploaded to the course Dropbox. Discussions & quiz are online)
Module #6 – MSI Logic

What you should be able to do after this module

- Understand the operation of Decoders, Encoders, Multiplexers, Demultiplexers, and Adders
- Synthesizes the gate level schematic for each of these MSI circuits
- Use VHDL to describe and simulate the MSI circuits covered in this module
### Integrated Circuit Scaling

- **Integrated Circuit Scales**

<table>
<thead>
<tr>
<th>Scale</th>
<th>Description</th>
<th>Example</th>
<th># of Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSI</td>
<td>Small Scale Integrated Circuits</td>
<td>Individual Gates</td>
<td>10's</td>
</tr>
<tr>
<td>MSI</td>
<td>Medium Scale Integrated Circuits</td>
<td>Mux, Decoder</td>
<td>100's</td>
</tr>
<tr>
<td>LSI</td>
<td>Large Scale Integrated Circuits</td>
<td>RAM, ALU's</td>
<td>1k - 10k</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integrated Circuits</td>
<td>uP, uCNT</td>
<td>100k - 1M</td>
</tr>
</tbody>
</table>

- we use the terms SSI and MSI. Everything larger is typically just called "VLSI"

- VLSI covers design that can't be done using schematics or by hand.
Decoders

- Decoders
  - a decoder has \( n \) inputs and \( 2^n \) outputs
  - one and only one output is asserted for a given input combination

ex) truth table of decoder

<table>
<thead>
<tr>
<th>“Inputs”</th>
<th>“Outputs”</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>Y3 Y2 Y1 Y0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

- To design the gate level circuitry, we write a logic expression for **EACH INDIVIDUAL OUTPUT**

- Remember that Boolean Algebra & K-maps produce a 1-bit output expression.
Decoder

• Decoder Structure

- The output stage of a decoder can be constructed using AND gates
- Inverters are needed to give the appropriate code to each AND gate
- Using AND/INV structure, we need:

  \[
  2^n \text{ AND gates} \quad n \text{ Inverters}
  \]

  \[
  \begin{array}{|c|c|c|c|c|}
  \hline
  A & B & Y3 & Y2 & Y1 & Y0 \\
  \hline
  0 & 0 & 0 & 0 & 0 & 1 \\
  0 & 1 & 0 & 0 & 1 & 0 \\
  1 & 0 & 0 & 1 & 0 & 0 \\
  1 & 1 & 1 & 0 & 0 & 0 \\
  \hline
  \end{array}
  \]

  Showing more inverters than necessary to illustrate concept

\[
Y0 = A' \cdot B'
\]

\[
Y1 = A' \cdot B
\]

\[
Y2 = A \cdot B'
\]

\[
Y3 = A \cdot B
\]
Decoders in Structural VHDL

• Decoder Example

- Let's design a 2-to-4 Decoder using Structural VHDL (i.e., connecting AND and INV components)

- We know we need to describe the following structure:

- We know what we'll need:

  \[ 2^n \text{ AND gates} = 4 \text{ AND gates} \]
  \[ n \text{ Inverters} = 2 \text{ Inverters} \]

  Showing more inverters than necessary to illustrate concept
Decoders in Structural VHDL

- **Decoder Example**

  Let's design the inverter using concurrent signal assignments....

  ```vhdl
  entity inv is
    port (In1 : in BIT;
          Out1 : out BIT);
  end entity inv;

  architecture inv_arch of inv is
  begin
    Out1 <= not In1;
  end architecture inv_arch;
  ```
Decoders in Structural VHDL

• Decoder Example

- Let's design the AND gate using concurrent signal assignments….

entity and2 is
  port (In1, In2 : in BIT;
         Out1 : out BIT);
end entity and2;

architecture and2_arch of and2 is
begin
  Out1 <= In1 and In2;
end architecture and2_arch;
Decoders in Structural VHDL

• Decoder Example

- Now let's work on the top level design entity called "decoder_2to4"

```vhdl
entity decoder_2to4 is
    port (A, B : in BIT;
    Y0, Y1, Y2, Y3 : out BIT);
end entity decoder_2to4;
```
• Decoder Example

- Now let's work on the top level design architecture called "decoder_2to4_arch"

```vhdl
architecture decoder_2to4_arch of decoder_2to4 is

    signal A_n, B_n : BIT;

    component inv
        port (In1    : in BIT;
              Out1    : out BIT);
        end component;

    component and2
        port (In1,In2 : in BIT;
              Out1    : out BIT);
        end component;

    begin

        ........
```

![Decoder Circuit Diagram](image-url)
Decoders in Structural VHDL

• Decoder Example

  - cont….

  
  begin
  
  U1 : inv port map (A, A_n);
  U2 : inv port map (B, B_n);
  
  U3 : and2 port map (A_n, B_n, Y0);
  U4 : and2 port map (A, B_n, Y1);
  U5 : and2 port map (A_n, B, Y2);
  U6 : and2 port map (A, B, Y3);
  
  end architecture decoder_2to4_arch;

end architecture decoder_2to4_arch;
**Decoders in Behavioral VHDL**

- **Decoder Example**
  
  - Let's design a 2-to-4 Decoder using **Behavioral VHDL** (i.e., using signal assignments & operators)
  
  - Now let's work on the top level design architecture called "decoder_2to4_arch"

```vhdl
entity decoder_2to4 is
  port (A,B : in BIT; Y0,Y1,Y2,Y3 : out BIT);
end entity decoder_2to4;

architecture decoder_2to4_arch of decoder_2to4 is
begin
  Y0 <= (not A) and (not B);
  Y1 <= (not A) and (B);
  Y2 <= (A) and (not B);
  Y3 <= (A) and (B);
end architecture decoder_2to4_arch;
```
Decoders in VHDL

- Decoder Example

- What would this look like in a functional simulation?

1 and only 1 output is asserted for each 2-bit input code.
Encoders

- **Encoder**
  - an encoder has $2^n$ inputs and $n$ outputs
  - it assumes that one and only one input will be asserted
  - depending on which input is asserted, an output code will be generated
  - this is the exact opposite of a decoder

ex) truth table of binary encoder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>00</td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
</tr>
<tr>
<td>0100</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
</tr>
</tbody>
</table>
Encoders

- Encoder

  - an encoder output is a simple OR structure that looks at the incoming signals

ex) 4-to-2 encoder

<table>
<thead>
<tr>
<th>I3</th>
<th>I2</th>
<th>I1</th>
<th>I0</th>
<th>Y1</th>
<th>Y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y1 = I3 + I2
Y0 = I3 + I1
Encoders in Structural VHDL

- **Encoders in VHDL**

  - 8-to-3 binary encoder modeled with Structural VHDL

  entity encoder_8to3_binary is

  port (I : in BIT_VECTOR (7 downto 0);
        Y : out BIT_VECTOR (2 downto 0) );

  end entity encoder_8to3_binary;

  architecture encoder_8to3_binary_arch of encoder_8to3_binary is

  component or4 port (In1,In2,In3,In4: in BIT; Out1: out BIT); end component;

  begin
  U1  : or4 port map (In1 => I(1), In2 => I(3), In3 => I(5), In4 => I(7),  Out1 => Y(0) );
  U2  : or4 port map (In1 => I(2), In2 => I(3), In3 => I(6), In4 => I(7),  Out1 => Y(1) );
  U3  : or4 port map (In1 => I(4), In2 => I(5), In3 => I(6), In4 => I(7),  Out1 => Y(2) );

  end architecture encoder_8to3_binary_arch;
Multiplexer

- Multiplexer
  - gates are combinational logic which generate an output depending on the current inputs
  - what if we wanted to create a “Digital Switch” to pass along the input signal?
  - this type of circuit is called a “Multiplexer”

ex) truth table of Multiplexer

<table>
<thead>
<tr>
<th>Sel</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
</tbody>
</table>
Multiplexer

- **Multiplexer**
  
  - the outputs will track the selected input
  
  - this is in effect, a “Switch”

ex) truth table of Multiplexer

<table>
<thead>
<tr>
<th>Sel</th>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplexer

- we can use the behavior of an AND gate to build this circuit:

  \[ X \cdot 0 = 0 \quad \text{“Block Signal”} \]
  \[ X \cdot 1 = X \quad \text{“Pass Signal”} \]

- we can then use the behavior of an OR gate at the output state (since a 0 input has no effect) to combine the signals into one output
Multiplexer

- if we wanted to explicitly form the logic expression, we could use a verbose truth table that lists each possible input value for A and B

<table>
<thead>
<tr>
<th>Sel</th>
<th>Out</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sel</th>
<th>A</th>
<th>B</th>
<th>Out</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>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>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>

Out = Sel·A + Sel·B
Multiplexer in Structural VHDL

- **2-to-1 Multiplexers in Structural VHDL**

  entity mux_2to1 is
  port (D : in BIT_VECTOR (1 downto 0);
        Sel : in BIT;
        Y : out BIT);
  end entity mux_2to1;

  architecture mux_2to1_arch of mux_2to1 is
  signal Sel_n : BIT;
  signal U2_out, U3_out : BIT;

  component inv1 port (In1: BIT; Out1: out BIT); end component;
  component and2 port (In1,In2 : in BIT; Out1: out BIT); end component;
  component or2 port (In1,In2 : in BIT; Out1: out BIT); end component;

  begin
  U1 : inv1 port map (In1 => Sel, Out1 => Sel_n);
  U2 : and2 port map (In1 => D(0), In2 => Sel_n, Out1 => U2_out);
  U3 : and2 port map (In1 => D(1), In2 => Sel, Out1 => U3_out);
  U4 : or4 port map (In1 => U2_out, In2 => U3_out, Out1 => Y);

  end architecture mux_2to1_arch;
Multiplexer in Behavioral VHDL

- Behavioral Model

```vhdl
entity mux_2to1 is
    port (D : in BIT_VECTOR (1 downto 0);
          Sel : in BIT;
          Y  : out BIT_LOGIC);
end entity mux_2to1;

architecture mux_2to1_arch of mux_2to1 is
begin
    Y <= D(0) when Sel='0' else D(1);
end architecture mux_2to1_arch;
```
Multiplexer in Behavioral VHDL

- Behavioral Model

```vhdl
entity mux_2to1 is
  port (D : in BIT_VECTOR (1 downto 0);
        Sel : in BIT;
        Y : out BIT_LOGIC);
end entity mux_2to1;

architecture mux_2to1_arch of mux_2to1 is
begin
  with Sel select
    Y <= D(0) when '0',
         D(1) when '1';
end architecture mux_2to1_arch;
```

2-to-1 Multiplexers in Behavioral VHDL (Selected Signal Assignments)
**Demultiplexer**

- **Demultiplexer**
  - this is the exact opposite of a Mux
  - a single input will be routed to a particular output pin depending on the Select setting

ex) truth table of Demultiplexer

<table>
<thead>
<tr>
<th>Sel</th>
<th>Y0</th>
<th>Y1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>In</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>In</td>
</tr>
</tbody>
</table>
Demultiplexer

- we can again use the behavior of an AND gate to “pass” or “block” the input signal

- an AND gate is used for each Demux output

<table>
<thead>
<tr>
<th>Sel</th>
<th>In</th>
<th>Y0</th>
<th>Y1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Y0 = Sel' \cdot In

Y1 = Sel \cdot In
Demultiplexer in Structural VHDL

- **Demultiplexers in VHDL**

  - **Structural Model**

    entity demux_1to2 is
    port (D : in BIT;
           Sel : in BIT;
           Y : out BIT_VECTOR (1 downto 0));
    end entity demux_1to2;

    architecture demux_1to2_arch of demux_1to2 is

    signal Sel_n : BIT;

    component inv1 port (In1: in BIT; Out1: out BIT); end component;
    component and2 port (In1,In2: in BIT; Out1: out BIT); end component;

    begin
    U1 : inv1 port map (In1 => Sel, Out1 => Sel_n);
    U2 : and2 port map (In1 => D, In2 => Sel_n, Out1 => Y(0));
    U3 : and2 port map (In1 => D, In2 => Sel, Out1 => Y(1));
    end architecture demux_1to2_arch;
Demultiplexer in Behavioral VHDL

Demultiplexers in VHDL

- Behavioral Model

entity demux_1to2 is
  port (D : in BIT;
        Sel : in BIT;
        Y : out BIT_VECTOR (1 downto 0));
end entity demux_1to2;

architecture demux_1to2_arch of demux_1to2 is
begin
  Y(0) <= D and (not Sel);
  Y(1) <= D and Sel;
end architecture demux_1to2_arch;
MSI Adders

• Remembering Binary Addition

- Let’s start with 1-bit Addition.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

+0 +0 +1 +1

0 1 1 10

- Notice that one of the additions (1+1) generated a “Carry”.

- We want to build a circuit that can add two inputs and create the Sum and Carry (Cout).

- Let’s list out the Truth Table for our circuit:

By inspection, we see that:

Sum = A \oplus B
Cout = A \cdot B
MSI Adders

- **Binary Addition**
  - We call this a “Half Adder” because it doesn’t consider a Carry-In from a prior addition

  ![Half Adder Diagram]

  \[
  \text{Sum} = A \oplus B \\
  \text{Cout} = A \cdot B
  \]

  - If we wanted to start doing multi-bit addition (i.e., n-bits + n-bits), we need to handle the Carry-Out from the prior bit position addition.

  - If we consider the “Carry-In”, we now have a 3-input circuit called a “Full Adder”

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>Cout</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
MSI Adders

• Binary Addition

  - If we can make a Full Adder, we can chain them together to perform multi-bit addition

- This is called a “Ripple Carry Adder”, because each subsequent stage needs to wait until the Cout in generated by the prior state. The Cout from the prior stage is then used as the current stage’s Cin.
MSI Adders

- Full Adders

Let’s look at the circuitry for a Full Adder:

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>Cout</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{Sum} = \text{Cin}' \cdot A' \cdot B + \text{Cin}' \cdot A \cdot B' + \text{Cin} \cdot A \cdot B + \text{Cin} \cdot A' \cdot B'
\]

\[
\text{Cout} = \text{Cin} \cdot A + A \cdot B + \text{Cin} \cdot B
\]
MSI Adders

• Full Adders

- We can accomplish the Full Adder using these logic expression. But could we do it with Half Adders?

- Let’s start by manipulating the expression for “Sum”

\[
\text{Sum} = \text{Cin}' \cdot A' \cdot B + \text{Cin}' \cdot A \cdot B' + \text{Cin} \cdot A \cdot B + \text{Cin} \cdot A' \cdot B'
\]

\[
\text{Sum} = A' \cdot (\text{Cin}' \cdot B + \text{Cin} \cdot B') + A \cdot (\text{Cin}' \cdot B' + \text{Cin} \cdot B)
\]

factor out A and A’

Notice that \((\text{Cin}' \cdot B + \text{Cin} \cdot B') = (\text{Cin} \oplus B)'\) i.e., an XNOR

Notice that \((\text{Cin}' \cdot B + \text{Cin} \cdot B) = (\text{Cin} \oplus B), \text{ i.e, an XOR}\)

\[
\text{Sum} = A' \cdot (\text{Cin} \oplus B) + A \cdot (\text{Cin} \oplus B')
\]

Notice that this is ALSO an XOR.
For example, if \(X = (\text{Cin} \oplus B)\), then we would have:

\[
\text{Sum} = A' \cdot X + A \cdot X' = A \oplus X
\]

Rearranging…

\[
\text{Sum} = A \oplus (\text{Cin} \oplus B)
\]

\[
\text{Sum} = (A \oplus B) \oplus \text{Cin}
\]

- Since the Sum of a “Half Adder” is \(A \oplus B\), we can use two Half Adders to produce the Full Adder Sum
MSI Adders

- Full Adder using Two Half Adders
  - Now we have the Sum taken care of, but what about the Carry Out?
MSI Adders

- **Full Adders using Two Half Adders**

  - Let’s look at some of the intermediate logic expressions:

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>1st Half Adder</th>
<th>2nd Half Adder</th>
<th>Full Adder Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This is what we originally wanted for Sum, so this proves our Full Adder “Sum” is correct.
### MSI Adders

#### Full Adders using Two Half Adders

- Notice the Cout Expression:

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>1st Half Adder Cout Sum</th>
<th>2nd Half Adder Cout Sum</th>
<th>Full Adder Cout Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Cout (Full Adder) = Cout (1st Half Adder) + Cout (2nd Half Adder)
MSI Adders

- **Full Adders using Two Half Adders**
  - This gives us our final Full Adder Circuit using:
    - 2 Half Adders (2x XOR, 2x AND)
    - 1 OR Gate