Unit 15

Common Hardware Components
WIDE MUXES
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 \textbf{m-bit wide (i.e. m-lane) n-to-1 mux}, use \textbf{m individual n-to-1 muxes}
Wide Multiplexer Example 1

• This 2-to-1, 32-bit wide mux is really:
  – 32 individual 2-to-1 muxes, each handling 1 "lane" of the 32-bit highway merger

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

Select bits $= 1_2 = 1_{10}$. 
Wide Multiplexer Example 2

- This 4-to-1, 8-bit wide mux is really:
  - 8 individual 4-to-1 muxes, each handling 1 "lane" of the 8-bit highway merger

Thus, input 0 = A[7:0] is selected and passed to the output

Select bits = 00₂ = 0₁₀.
Exercise

• How many 1-bit wide muxes and of what size would you need to build a 4-to-1, 16-bit wide mux (i.e. there are 4 numbers: \(W[15:0]\), \(X[15:0]\), \(Y[15:0]\) and \(Z[15:0]\) 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?
Using muxes to control when register save data

REGISTER WITH ENABLES
Register Resets/Clears

- When the power turns on the bit stored in a flip-flop will initialize to a random value
- Better to initialize it to a known value (usually 0's)
- Can use an asynchronous or synchronous "reset" to force the flip-flops to 0's

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>Dᵢ</th>
<th>Qᵢ*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>X</td>
<td>X</td>
<td>Qᵢ</td>
</tr>
<tr>
<td>↑↑</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Register Problem

- The value on the D input is sampled at the clock edge and passed to the Q output and holds until the next clock edge
- Problem: Register will save data on EVERY edge
  - Often we want the ability to save on one edge and then keep that value for many more cycles

4-bit Register – On clock edge, D is passed to Q
Solution

- Registers (D-FF’s) will sample the D bit every clock edge and pass it to Q.
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge.
- We can add an enable input and some logic in front of the D-FF to accomplish this.

### FF with Data Enable
(Always clocks, but selectively chooses old value, Q, or new value D)

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>EN</th>
<th>Dᵢ</th>
<th>Qᵢ*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Qᵢ</td>
</tr>
<tr>
<td>↑</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Qᵢ</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
 Registers w/ Enables

- When EN=0, Q value is passed back to the input and thus Q will maintain its value at the next clock edge.
- When EN=1, D value is passed to the input and thus Q can change at the edge based on D.

When EN=0, Q is recycled back to the input.

When EN=1, D input is passed to FF input.
4-bit Register w/ Data (Load) Enable

- Registers (D-FF’s) will sample the D bit every clock edge and pass it to Q.
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge.
- We can add an enable input and some logic in front of the D-FF to accomplish this.

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>EN</th>
<th>$D_i$</th>
<th>$Q_i^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>$Q_i$</td>
</tr>
<tr>
<td>††</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>††</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>$Q_i$</td>
</tr>
<tr>
<td>††</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>††</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4-bit register with 4-bit wide 2-to-1 mux in front of the D inputs.
Registers w/ Enables

- The D value is sampled at the clock edge only if the enable is active.
- Otherwise the current Q value is maintained.
Counters

• Count (Add 1 to Q) at each clock edge
  – Up Counter: \( Q^* = Q + 1 \)
  – Can also build a down counter as well \( (Q^* = Q - 1) \)

• Standard counter components include other features
  – Resets: Reset count to 0
  – Enables: Will not count at edge if \( EN=0 \)
  – Parallel Load Inputs: Can initialize count to a value \( P \) (i.e. \( Q^* = P \) rather than \( Q+1 \))
Sample 4-bit Counter

- **4-bit Up Counter**
  - **RST:** synchronous reset input
  - **PE and P_i inputs:** loads Q with P when PE is active
  - **CE:** Count Enable
    - Must be active for the counter to count up
  - **TC (Terminal Count) output**
    - Active when Q=1111 AND counter is enabled
    - \( TC = EN \cdot Q3 \cdot Q2 \cdot Q1 \cdot Q0 \)
    - Indicates that on the next edge it will roll over to 0000
    - Used to create 8-, 12-, 16-bit, etc. counters from these 4-bit building blocks

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>PE</th>
<th>CE</th>
<th>Q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counter Design

• Sketch the design of the 4-bit counter presented on the previous slides
Counters

SR=active at clock edge, thus Q=0

Q*=Q+1

Enable = off, thus Q holds

Q*=Q+1

Q*=Q+1

PE = active, thus Q=P

Q*=Q+1

Q*=Q+1

Mealy TC output: EN•Q3•Q2•Q1•Q0
Counter Exercise
DESIGN OF A SIMPLE INSTRUCTIONS SET AND PROCESSOR
Arithmetic and Logic Units

- Arithmetic and Logic Units (ALUs) can perform 1 of many potential arithmetic or logic operations.
- Let's define and design an ALU that will perform various operations...

![ALU Diagram]

We will design what is inside this block.

We just made up these code assignments and the various operations. Remember, we definitely need to support ADD, SUB, AND, and CLR (R=0).

<table>
<thead>
<tr>
<th>F[2:0]</th>
<th>Op./Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>R = X + Y</td>
</tr>
<tr>
<td>001</td>
<td>R = X - Y</td>
</tr>
<tr>
<td>010</td>
<td>R = X</td>
</tr>
<tr>
<td>011</td>
<td>R = Y - X</td>
</tr>
<tr>
<td>100</td>
<td>R = X &amp; Y</td>
</tr>
<tr>
<td>101</td>
<td>Unused</td>
</tr>
<tr>
<td>110</td>
<td>R = 0</td>
</tr>
<tr>
<td>111</td>
<td>Unused</td>
</tr>
</tbody>
</table>
Blank ALU To Complete

2-to-1, 4-bit wide mux

2-to-1, 4-bit wide mux

4-bit Binary Adder

2-to-1, 4-bit wide mux

S0 = __________

S1 = ______

S2 = ______

S3 = ________

00

0

0

0

F2

F1

F0

R0

R1

R2

R3

X0

X1

X2

X3

EE109 ALU

000   |  R = X + Y  |  100  |  R = X & Y
001   |  R = X - Y  |  101  |  Unused
010   |  R = X      |  110  |  R = 0
011   |  R = Y - X  |  111  |  Unused

000   |  R = X + Y  |
001   |  R = X - Y  |
010   |  R = X      |
011   |  R = Y - X  |
### Control Logic

**Table:**

<table>
<thead>
<tr>
<th>R</th>
<th>FS[2:0]</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>Ci</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X+Y</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X-Y</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y-X</td>
<td>011</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X &amp; Y</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>unused</td>
<td>101</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>d</td>
<td>1</td>
<td>0</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>unused</td>
<td>111</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

- S0 = F1•F0
- S1 = F1•F0'
- S2 = F1'•F0
- Ci = F0
- S3 = F2
Completed ALU

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>R = X + Y</td>
<td>100</td>
<td>R = X &amp; Y</td>
</tr>
<tr>
<td>001</td>
<td>R = X - Y</td>
<td>101</td>
<td>Unused</td>
</tr>
<tr>
<td>010</td>
<td>R = X</td>
<td>110</td>
<td>R = 0</td>
</tr>
<tr>
<td>011</td>
<td>R = Y - X</td>
<td>111</td>
<td>Unused</td>
</tr>
</tbody>
</table>
Aside: Impacts of Coding (1)

• What if we changed the codes used for each operation?

<table>
<thead>
<tr>
<th>F[2:0]</th>
<th>Op./Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>R = X + Y</td>
</tr>
<tr>
<td>001</td>
<td>R = X - Y</td>
</tr>
<tr>
<td>010</td>
<td>R = X</td>
</tr>
<tr>
<td>011</td>
<td>R = Y - X</td>
</tr>
<tr>
<td>100</td>
<td>R = X &amp; Y</td>
</tr>
<tr>
<td>101</td>
<td>Unused</td>
</tr>
<tr>
<td>110</td>
<td>R = 0</td>
</tr>
<tr>
<td>111</td>
<td>Unused</td>
</tr>
</tbody>
</table>

We just made up these code assignments and the various operations. Remember, we definitely need to support ADD, SUB, AND, and CLR (R=0).
Aside: Impacts of Coding (2)

<table>
<thead>
<tr>
<th>R</th>
<th>FS[2:0]</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>Ci</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X + Y</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y - X</td>
<td>001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X - Y</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>d</td>
<td>1</td>
<td>0</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X&amp;Y</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>Unused</td>
<td>110</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Unused</td>
<td>111</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

- S0 = F2'F0
- S1 = F2F0' + F1F0
- S2 = F2F0' + F2'F1'F0
- Ci = F1+F0
- S3 = F1F0 + F2F0

Notice how much more logic this coding yields.
MULTIPLIERS
Unsigned Multiplication Review

• Same rules as decimal multiplication
• Multiply each bit of Q by M shifting as you go
• An m-bit * n-bit mult. produces an m+n bit result (i.e. n-bit * n-bit produces 2*n bit result)
• Notice each partial product is a shifted copy of M or 0 (zero)

```
   1010  M (Multiplicand)
*   1011  Q (Multiplier)
   10110
```
Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result (i.e. n-bit * n-bit produces 2*n bit result)
- Notice each partial product is a shifted copy of M or 0 (zero)

\[
\begin{array}{c}
1010 \\
\times 1011 \\
\hline
1010 \\
1010 \\
0000 \\
+ 1010 \\
\hline
01101110
\end{array}
\]

M (Multiplicand)  Q (Multiplier)  PP (Partial Products)  P (Product)
Combinational Multiplier

• Partial Product (PP<sub>i</sub>) Generation
  - Multiply Q[i] * M
    • if Q[i]=0 => PP<sub>i</sub> = 0
    • if Q[i]=1 => PP<sub>i</sub> = M

\[
\begin{array}{c}
\text{1010} \\
\times \text{1011} \\
\hline
\text{1010} \\
\text{1010} _{\text{PP (Partial Products)}} \\
0000 _{\text{PP (Partial Products)}} \\
+ \text{1010} _{\text{PP (Partial Products)}} \\
\hline
\text{01101110} _{\text{P (Product)}}
\end{array}
\]

M (Multiplicand)
Q (Multiplier)
P (Product)
Combinational Multiplier

• Partial Product (PPₗ) Generation
  – Multiply Q[i] * M
    • if Q[i]=0 => PPᵢ = 0
    • if Q[i]=1 => PPᵢ = M
  – AND gates can be used to generate each partial product
Combinational Multiplier

- Partial Products must be added together
- Combinational multipliers require long propagation delay through the adders
  - propagation delay is proportional to the number of partial products (i.e. number of bits of input) and the width of each adder
Multiplication Overview

- Combinational: Array multiplier uses an array of adders
  - Can be as simple as N-1 ripple-carry adders for an NxN multiplication

AND Gate Array produces partial product terms

\[
\begin{array}{cccc}
  m_3 & m_2 & m_1 & m_0 \\
\times & q_3 & q_2 & q_1 & q_0 \\
  m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 \\
  m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 \\
  m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 \\
  m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
+ & m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 \\
\end{array}
\]
Array Multiplier

- Maximum n-bit * n-bit delay is proportional to 2*n