Task: Checking for a Lower-Case Letter

What if we also need logic to check whether an ASCII character is a lower-case letter.

In ASCII, 'a' is 1100001 (0x61), and 'z' is 1111010 (0x7A).

Recall that 'A' is 1000001 (0x41), and 'Z' is 1011010 (0x5A).

Can we reuse our solutions for upper-case letters?
   Of course we can!

Change C₅' to C₅ to Obtain L(C) from U(C)

Let's again say that the ASCII character is in \( C = C₆C₅C₄C₃C₂C₁C₀ \).

By breaking up the truth table, we obtained

\[
U(C) = C₆C₅C₄'C₃ + C₃' + C₂C₁C₀ + C₆C₅'C₄'C₃' + C₃' + C₂'C₄ + C₁'C₉\]

But lower-case characters are only different from upper-case in \( C₅ \), which is 1 instead of 0.

\[
L(C) = C₆C₅C₄'C₃ + C₃' + C₂'C₄ + C₁'C₉\]

Change Comparator Input to Calculate L(C)

Or just change the comparators’ inputs.
Want Logic to Choose Between Two Signals

What if we want one design to check for either upper-case or lower-case letters?
In a few examples,
◦ we added a control signal S
◦ to select between functions.
Can we design logic
◦ that uses a control signal S to select
◦ between two arbitrary signals, D₁ (when S = 1) and D₀ (when S = 0)?

Truth Tables for a 2-to-1 Multiplexer

A full truth table for such logic appears to the right.
But we could shorten it as shown below...

<table>
<thead>
<tr>
<th>S</th>
<th>D₁</th>
<th>D₀</th>
<th>Q</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Unselected inputs do not matter (marked with “x”).

We Normally Use the Most Compact Truth Table

In this case, we can even write outputs in terms of other inputs, as shown here.

<table>
<thead>
<tr>
<th>S</th>
<th>D₁</th>
<th>D₀</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>

Unselected inputs do not matter (marked with “x”).

Expression for a 2-to-1 Multiplexer

Let’s solve with a K-map.

\[ Q = S' D₀ + S D₁ \]

<table>
<thead>
<tr>
<th>S</th>
<th>D₁</th>
<th>D₀</th>
<th>Q</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D₁</th>
<th>D₀</th>
<th>Q</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>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>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>
Expression for a 2-to-1 Multiplexer

But $Q$ just selects $D_0$ or $D_1$ (as desired)!

$$Q = S'D_0 + SD_1$$

Could we have just written this expression using the table to the right?

$Q$ is $D_0$ when $S = 0$, and $D_1$ when $S = 1$...

---

Implementation and Symbolic Form of a 2-to-1 Mux

The circuit below shows a 2-to-1 mux (multiplexer), for which the symbolic form appears to the right.

---

Selecting from More than Two Expressions

What if we want to select between four expressions, $D_3, D_2, D_1, \text{ and } D_0$?

One answer is to use muxes hierarchically:

- start by using one 2-to-1 mux (signal $S_1$)
- to decide between $D_3$ or $D_2$ and $D_1$ or $D_0$

---

For the Second Level, Use More Muxes

But how do we deliver two expressions to each mux input?

Use more muxes (both controlled by $S_0$)!

Notice that $S_1S_0$ then allows us to choose from four expressions.
AND Gates Represent Minterms ANDed with Data Inputs

For something as common as a mux, we typically build directly from gates. Notice that each AND gate produces a minterm of $S_1$, $S_0$ ANDed with the corresponding $D_i$.

A $2^N$-to-1 Mux Requires N Select Bits

The diagram to the right shows the symbolic form of a 4-to-1 mux. We can, of course, further extend this idea to build 8-to-1 muxes, 16-to-1 muxes, and so forth. When selecting amongst $P = 2^N$ inputs $D_{P-1} \ldots D_0$ we need $N$ bits of select input, $S_{N-1} \ldots S_0$.

Can Use Sets of Muxes to Select Amongst Groups of Bits

We can also generalize the idea of multiplexers by

- using a common control signal
- to select between groups of inputs.

Generally,

- an $N$-to-$M$ multiplexer
- represents $M$ separate $(N/M)$-to-1 muxes
- each with $\log_2(N/M)$ select bit inputs
- (typically $N/M = 2^K$ for some integer $K$).

Example of a Set of Muxes with Common Select Input

For example, recall the design of the $N$-bit adder and subtractor. We could have used a $2N$-to-$N$ mux

- to choose between $B_i$ and $B_i'$ for the adder's $B$ input
- based on a common (one-bit) control signal $S$.

(Previously, we used the nature of the mux' data inputs, $B_i$ versus $B_i'$, to simplify each mux' logic to an XOR gate.)
Another Design Problem: Checking Four Types of ASCII

Now think again about our ASCII checker. Say that we want four kinds of comparison:
- control characters (0x00 to 0x1F),
- lower-case letters (0x40 to 0x5A),
- upper-case letters (0x60 to 0x7A), and
- digits (0x30 to 0x39).

How can we design logic to check for any of the four types?

The Answer? Use Muxes! Two 28-to-7 Muxes