WP274 (v1.0) February 4, 2008

Multiplexer Selection

By: Ken Chapman

It is obvious that a multiplexer is used to select one of several input signals; however, in this white paper, I will consider a variety of ways in which multiplexers can be implemented within Xilinx FPGA devices. First, I will consider the straight forward method and identify some potential issues. We will then investigate some alternative techniques that can lead to more efficient and lower cost implementations. In your future designs, you will be able to consider your multiplexer selections in order to select signals.

Multiplexers in Xilinx FPGAs

Multiplexers can be implemented in Xilinx FPGA look-up tables (LUTs) as with any other kind of logic, but all Xilinx Spartan™ and Virtex™ FPGAs also include dedicated multiplexers to combine the outputs of LUTs. These are labeled "MUXFx" (or "FxMUX") where "x" is the equivalent LUT size it provides. For example, the four-input LUTs connect to a MUXF5, which allows the implementation of any function of five inputs. If the two LUTs contain 2-input muxes, the result of the MUXF5 is a 4-input mux, which is a 6-input function (see Figure 1). The MUXF5 connects to the MUXF6, allowing any function of six inputs, or an 8-input mux. The Virtex-II, Virtex-II Pro, Virtex-4, and Spartan-3 Generation FPGAs extend this capability with MUXF7 and MUXF8. The Virtex-5 FPGA starts with a 6-input LUT and adds MUXF7 and MUXF8. The concepts described here can be applied to any Xilinx FPGA family, although the implementation details may vary between families.

The Normal Case

The majority of multiplexers are formed by using the look-up tables (LUTs) and dedicated multiplexer elements of logic slices. These would normally be implemented automatically by a synthesis tool such as XST, so let's take a look at the details and make some observations. The following VHDL code uses a CASE statement to describe a simple 4-to-1 multiplexer:

```
signal   D3 : std_logic;
signal   D2 : std_logic;
signal   D1 : std_logic;
signal   D0 : std_logic;
signal  SEL : std_logic_vector(1 downto 0);
signal   Y : std_logic;
.
.
.
   process (SEL, D3, D2, D1, D0)
   begin
      case SEL is
         when "00" => Y <= D0;
         when "01" => Y <= D1;
         when "10" => Y <= D2;
         when "11" => Y <= D3;
         when others => NULL;
      end case;
   end process;
```

XST is able to recognize this as a multiplexer and automatically exploit the dedicated multiplexer within each slice. The result is a multiplexer that is smaller and faster than it would be if it was implemented using only LUTs, as shown in Figure 1.

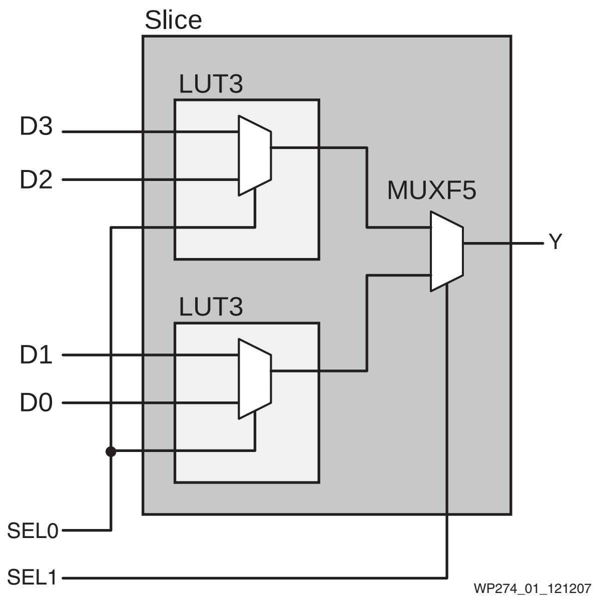

Figure 1: **4:1 Multiplexer Using Dedicated Slice Multiplexer**

Larger multiplexers can be implemented by exploiting the dedicated MUXF6, MUXF7, and MUXF8 components; here again, XST is able to achieve this automatically. For example, an 8:1 multiplexer has the structure illustrated in Figure 2.

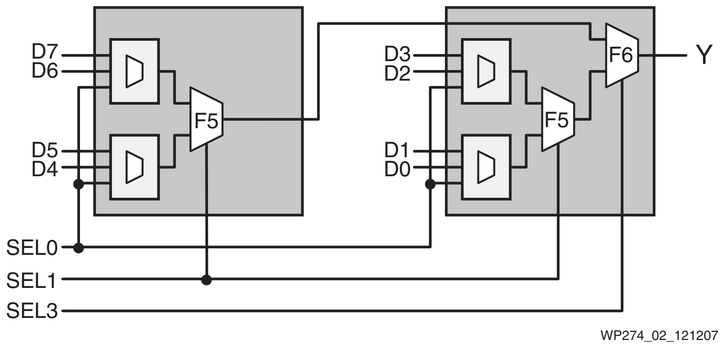

Figure 2: **8:1 Multiplexer Using Two Slices**

Using this implementation technique, it is easy to estimate the size of a multiplexer because each LUT is responsible for two data inputs. Hence the 4:1 multiplexer requires two LUTs (one slice) and the 8:1 multiplexer requires four LUTs (two slices).

The Unpleasant Case

After all that good news, we should take a moment to examine a case in which things are not as nice as we would hope. So far we have only considered multiplexers in which there are a power-of-two number of data inputs. However, this won't be the case in all designs.

Let's take a look at a 5:1 multiplexer and examine the alternatives that must be considered. The VHDL looks very simple, but are you aware of what this will really implement? Take a moment to sketch what you believe is required to implement this piece of code (then read on to see if you are right).

```
signal   D4 : std_logic;
signal   D3 : std_logic;
signal   D2 : std_logic;
signal   D1 : std_logic;
signal   D0 : std_logic;
signal  SEL : std_logic_vector(2 downto 0);
signal  CLK : std_logic;
signal   Y  : std_logic;
.
.
.
   process (CLK)
   begin
      if CLK'event and CLK="1" then
         case SEL is
            when "000" => Y <= D0;
            when "001" => Y <= D1;
            when "010" => Y <= D2;
            when "011" => Y <= D3;
            when "100" => Y <= D4;
            when others => NULL;
         end case;
      end if;
   end process;
```

It is important to observe that all connections to the dedicated MUXFx components are formed by fixed, dedicated connections. Although this makes them fast, data inputs to the MUXF5 components must come directly from the associated LUTs, and data inputs to the MUXF6 must come directly from the associated MUXF5 components. This will have an impact when you implement multiplexers that do not have a power-of-two number of data inputs. Figure 3 and Figure 4 show two potential 5:1 multiplexer implementations that XST may use.

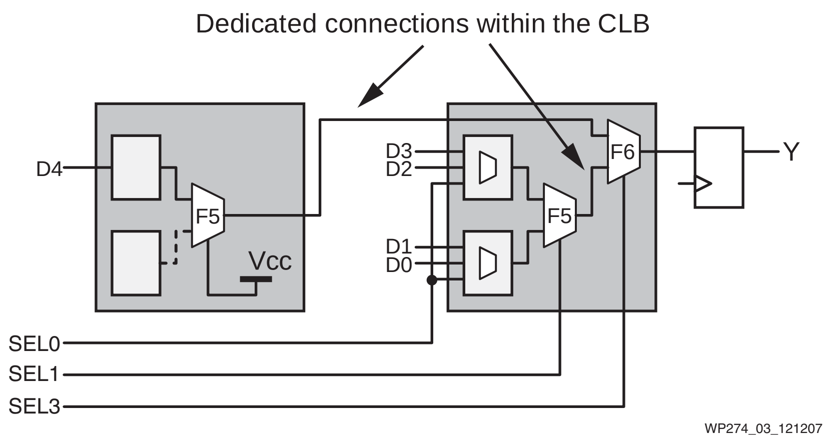

Figure 3: **Potential Implementations of 5:1 Multiplexer: Method A**

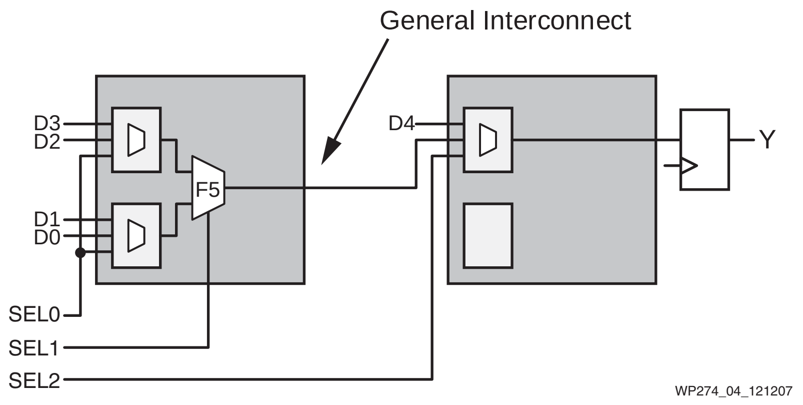

Figure 4: **Potential Implementations of 5:1 Multiplexer: Method B**

Although both methods use 1½ slices (three LUTs), the performances will differ. Method A exploits the dedicated interconnections and multiplexers within the CLB. Notice how the LUT and MUXF5 of the left slice are sacrificed in order to gain access to the MUXF6 in the right slice. All this is done to impose the minimum delay to all signal paths. By contrast, Method B incurs additional delay due to the general interconnect and use of a second LUT-based multiplexer.

It would be natural to assume that Method A is always superior and should be the preferred solution. However, we should consider the cases in which the data inputs are not individual signals, but instead are multi-signal data busses, such as those being selected by a processor. Since Method A requires a MUXF6 for each output bit selected, and only one MUXF6 exists for every two slices, the bus multiplexer will become spread out. For example, 16-bit data busses will result in 32 slices being occupied (compared with 24 slices using Method B). Of course, the 16 unused LUTs within the 32 slices of Method A could be used for simple logic functions; however, such logic is often completely unrelated and cannot always be conveniently merged in with the multiplexer during implementation.

Unintended Functionality

So, did you spot that neither Method A nor B is actually what a synthesis tool will (or should) implement? While the VHDL describes a 5:1 multiplexer, the description also defines functionality I almost certainly didn't want or intend. If you didn't spot the cause of the problem, it would be well worth taking a look at your own designs to see if you, too, have implied similar additional logic that impacts performance and, potentially, the cost of your system.

The beginning of this problem is that the selection is controlled by three signals that have a potential for eight combinations. As the multiplexer only uses five combinations, what should happen to the `Y` output if `SEL` is "101", "110" or "111"? These combinations will never occur in many designs; for those designs in which they could occur, we wouldn't be using the output from the multiplexer anyway, or we really would have a bug in the design.

Fortunately, the VHDL language provides us with the "when others" clause to catch all the undefined conditions of the input. This includes the more esoteric states that don't even exist in the pure digital world (which is why it is still required even when you have apparently covered all the combinations in a power-of-two multiplexer). However, we still need to define what to do in these cases. This, unfortunately, is the cause of all the trouble. The word "NULL" means "do nothing," but it takes logic to achieve this "nothing."

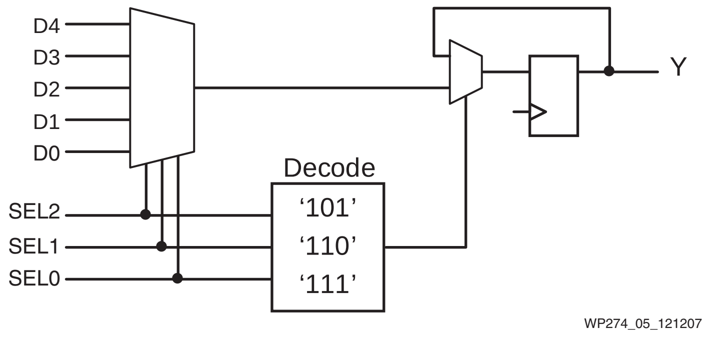

Figure 5: **Additional Logic to Support a "When Others" Clause in a Clocked Process**

"Nothing" means that the multiplexer output must retain the previous value. The functional diagram in Figure 5 illustrates the additional logic required to implement this in a clocked process. A combinatorial process will cause a highly undesirable LATCH to be implied, which is not directly available in the slice. There are numerous ways in which the additional logic could actually be implemented in the slices. However, to consider them just misses the point that we simply do not want any additional logic because this functionality is not needed in the first place.

The solution is very simple. What is required in the “unused” conditions must be more precisely defined. We can specify that the output really doesn’t matter as shown in the following code.

```
case SEL is
   when “000” => Y <= D0;
   when “001” => Y <= D1;
   when “010” => Y <= D2;
   when “011” => Y <= D3;
   when “100” => Y <= D4;
   when others => Y <= ‘X’;
end case;
```

While some people have an aversion to ever observing an “X” during simulation, this is perfectly acceptable, provided the unknown state is never used in the design. The synthesis tool should then realize that it is able to avoid any additional logic and allow the multiplexer to do whatever it happens to do in the undefined conditions. XST will actually implement the multiplexer using Method A described previously and shown in Figure 3, page 5.

Alternatively, we can explicitly define the remaining conditions as shown in the following code.

```
case SEL is
   when “000” => Y <= D0;
   when “001” => Y <= D1;
   when “010” => Y <= D2;
   when “011” => Y <= D3;
   when “100” => Y <= D4;  }
   when “101” => Y <= D4;  }
                           }  Replicated data input
   when “110” => Y <= D4;  }
   when “111” => Y <= D4;  }
   when others => Y <= ‘X’;
end case;
```

The functionality is now fully described and predictable both in simulation and final implementation. Although the definition has artificially expanded the multiplexer to eight inputs, by assigning the same data input for four of the conditions, synthesis tools should be able to reduce the logic required. XST will implement this code using Method B described previously and shown in Figure 4, page 5. This illustrates the importance of appreciating different coding styles and realizing that there is benefit to understanding how your preferred synthesis tool behaves.

Alternative Techniques

Having looked at the obvious implementation of a multiplexer in some detail, we now consider alternative techniques to implement this common function. If we can implement a multiplexer using fewer device resources and/or make a multiplexer faster in a given speed grade, this will help contribute to an overall cost saving in the complete system design. This should be of particular interest to designers of high-volume products that use Spartan devices.

Back to Basics

So what are the logic gates associated with a simple multiplexer? Figure 6 and Figure 7 show the simple truth table and gates for a 2:1 multiplexer. Although we don't normally think in terms of gates when designing with look-up tables, this view reminds us exactly how a multiplexer operates.

SEL	D1	D0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

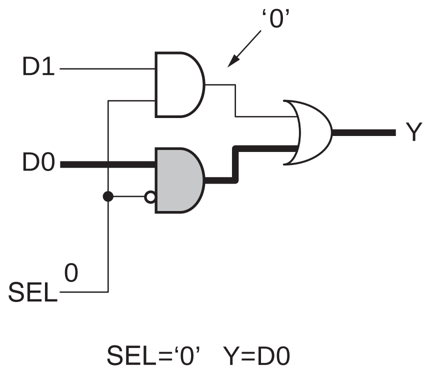

Figure 6: **Truth Table and Gates for a 2:1 Multiplexer: SEL=0, Y=D0**

SEL	D1	D0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

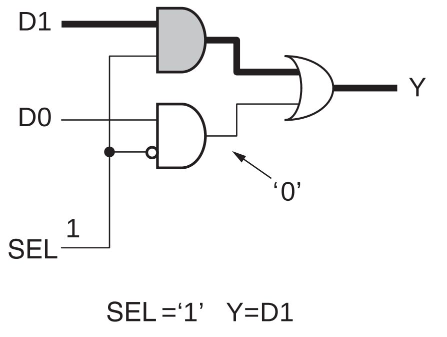

Figure 7: **Truth Table and Gates for a 2:1 Multiplexer: SEL=1, Y=D1**

The select signal is effectively used to "turn on" only one AND gate at a time. This allows the associated data signal to pass through the AND gate to the OR gate. The AND gate that is "off" is guaranteed to generate a logic "0," which means that the output from the OR gate is determined only by the signal from the AND gate that is turned on.

Although we normally think of a multiplexer as being one consolidated function, it is possible to separate the gates into the multiple masking AND gates and the single OR gate signal combiner. A look-up table (LUT) can directly implement any function up to four inputs, which limits the consolidated multiplexer to a 2-to-1 structure with its three input signals. However, a LUT is able to implement a four-input OR gate that would be suitable for a 4:1 multiplexer, as shown in Figure 8.

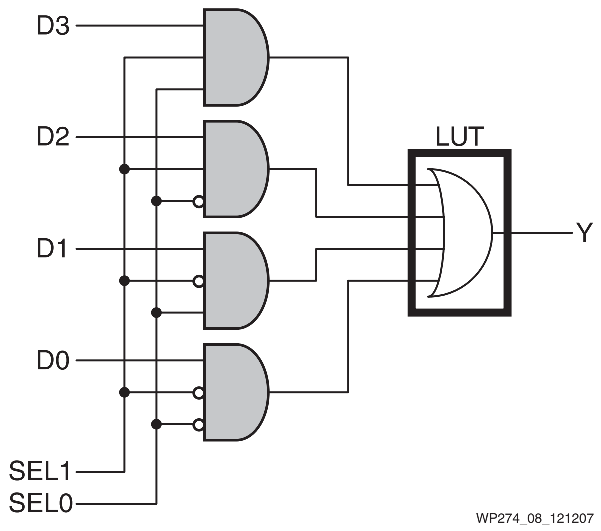

Figure 8: **Gates for a 4:1 Multiplexer**

So far, this appears to be a somewhat ridiculous suggestion because the AND gates each require a LUT to implement, and the total size of a 4:1 multiplexer will be five LUTs. This is both bigger and slower than the multiplexer created around the dedicated MUXF5 seen earlier. Fortunately, there are two ways in which we can *hide* the AND gates in our designs.

The most ideal method is to exploit the dedicated reset logic provided with flip-flops and block memory. As shown in Figure 9, a flip-flop in the path from `D2` is used to force logic "0" into the OR gate by holding the reset control active. Releasing the reset allows the value of `D2` to propagate through the flip-flop and then on to provide the output of the multiplexer structure. Similarly, the block RAM can be controlled to provide `D1`.

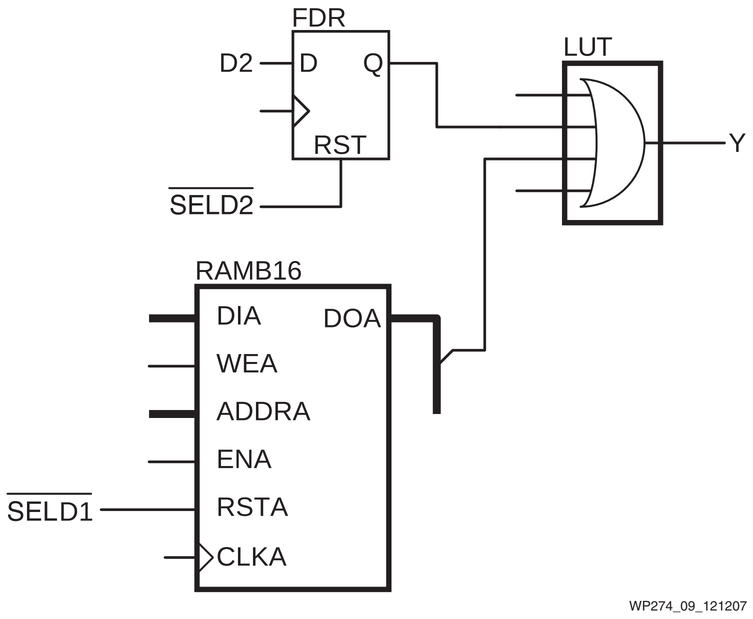

Figure 9: **Using Resets Instead of AND Gates to Select Data**

Obviously, the one clock cycle latency in the data path may be undesirable when considering the functional level, but is exactly what we need in a pipelined system design to maximize performance. Indeed, in a well-pipelined design (and whenever block RAM is used), the pipeline stage exists and simply requires the reset control to be added. Block RAM can continue to be used for writing or full access via the second port.

However, flip-flop contents will be destroyed by the reset control. The reset control prevents this technique from being directly applicable to flip-flops used to implement functions, such as counters and data registers, unless they are only used when continuously selected as input to the multiplexer.

Selection of `D1` and `D2` is achieved by controlling each reset in a one-hot style (only one signal is active at time). A 4:1 multiplexer therefore has four select lines instead of the normal two encoded select signals. If a LUT is used to implement a decoding gate, it nullifies the area saved by the use of the flip-flop to replace the AND gate in the first place.

Hence, the system design should include a plan to implement a one-hot selection scheme from the outset to avoid this conversion logic. The acceptable exception to this is when the multiplexer operation applies to multi-bit busses. In these cases, a LUT is still required to generate the reset control signal, but this is distributed to multiple flip-flops.

When applying the synchronous reset control to the flip-flops, take care to ensure that the selection logic is not over-complicated or enlarged due to a global reset competing for resources (see WP272, *Get Smart About Reset [Think Local, Not Global]*).

It may also be possible to *hide* the AND gate with other combinatorial logic using any spare capacity within LUTs that form the source of the signal. In Figure 10, the AND gate selecting the `D0` input is combined with a traditional 2:1 multiplexer (although it could be any function up to three inputs).

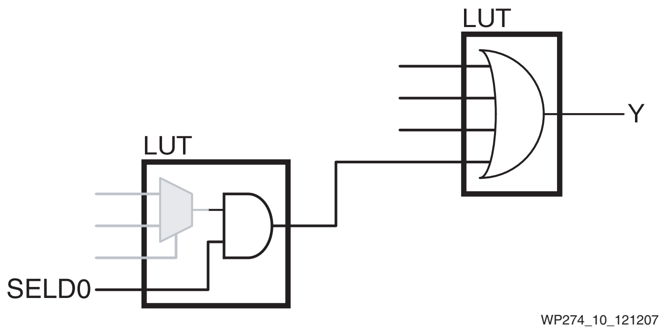

Figure 10: **Hiding the Select AND Gate in Other Logic**

Priority Selection

The final multiplexer technique we will consider is the use of MUXCY components normally associated with carry logic. The technique generally provides one data input per LUT; although this is not the most area-efficient solution in its own right, it does have other advantages. Take a look at an example of a 4:1 multiplexer using the carry chain in Figure 11.

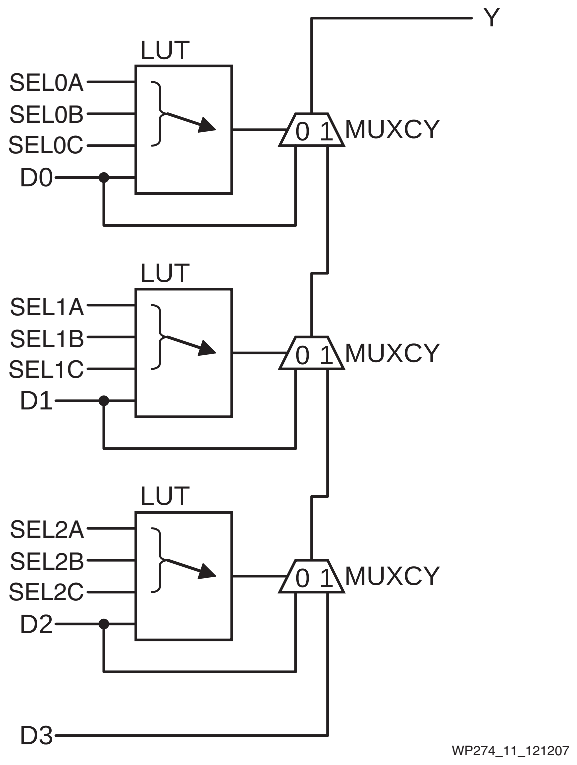

Figure 11: **4-to-1 Multiplexer Using the Carry Chain**

Each MUXCY component is a 2:1 multiplexer with the select line controlled by the associated LUT. The data inputs are fixed connections: the first is one of the four LUT inputs, and the second is the output from the MUXCY below. (To use these components as a data multiplexer in your design, you must accept this fixed connectivity and work with it.) The initial impact is that the LUT is essentially reduced to a three-input function because the fourth input must be the data input and can be any value. However, the three inputs still provide significant decoding capability.

In Figure 11, `D0` is applied to the top MUXCY. To select `D0` and pass it to the `Y` output, the MUXCY select control must be logic "0" and is determined by the top LUT, which allows up to three signals (`SEL0A`, `SEL0B`, and `SEL0C`) to be decoded.

If the condition for selection of `D0` is not satisfied, the output from the top LUT will be a logic "1." So, the top MUXCY selects the carry chain input. It is now possible to select the `D1` input as a result of the decoding of the three inputs `SEL1A`, `SEL1B`, and `SEL1C`. Note that these signals do not need to be the same as those used to select `D0`.

Similarly, if `D0` and `D1` are not selected, the decision passes to the next stage with a decoding of `SEL0A`, `SEL0B`, and `SEL0C` controlling the selection of `D2`. In the event that `D2` is not selected, `D3` becomes the default data selection with no further decoding required.

Therefore, this technique provides a multiplexer structure in which the inputs have a priority in addition to being selected. It is ideal for combining the decoding of signals with the multiplexer logic in those cases where separate signals are used to define the selection. For example, consider the selection of data passed to a processor, as shown in Figure 12.

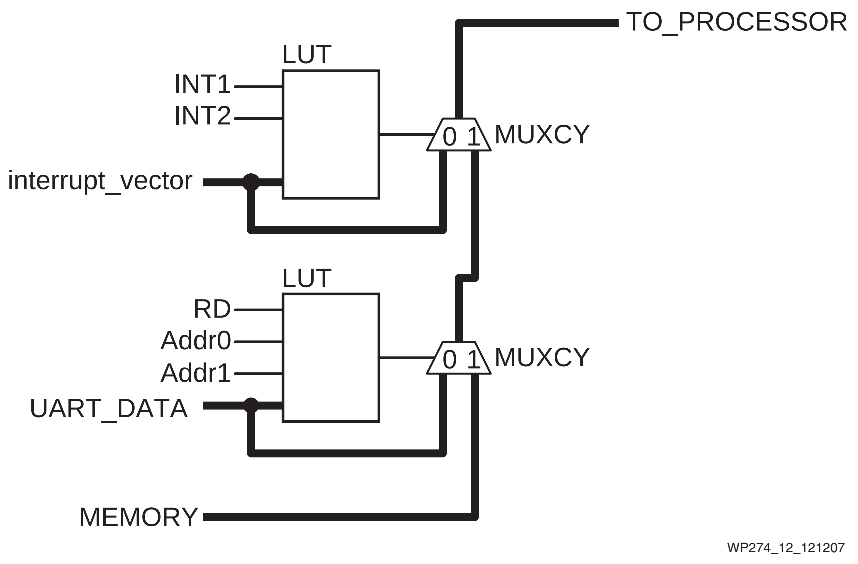

Figure 12: **Data Selection to a Processor**

In the event of an interrupt, which is a very high-priority situation, the processor will read the `interrupt_vector`. This, therefore, occupies the top position in the chain and is controlled by the two interrupt signals. If there is no interrupt to process, the processor is able to access data from a UART or memory. The UART selection is uniquely determined by a couple of address bits and a read strobe. This means that `MEMORY_data` is the ideal default selection for covering a wide range of addresses, even non-contiguous addresses.

The carry chain multiplexer may also be beneficial in some system designs because of the physical structure it imposes on the layout of the design. Figure 13 illustrates the connections from data sources (shown in white) to a multiplexer (shown in black).

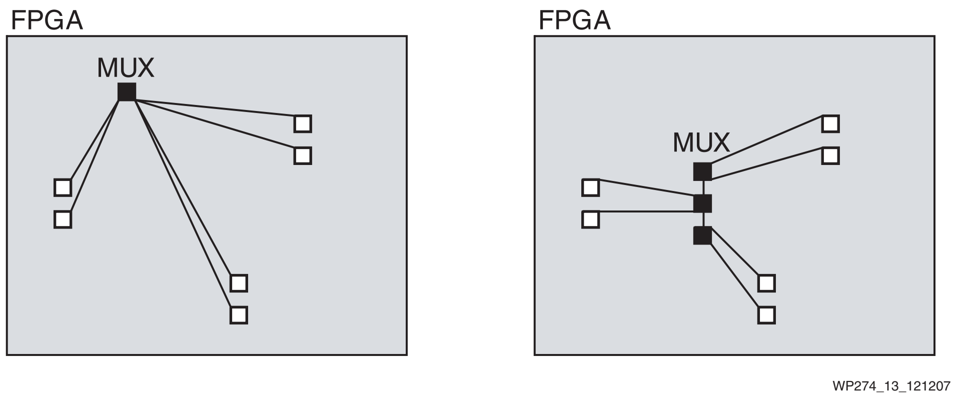

Figure 13: **Layout of Traditional Multiplexer and Carry-based Multiplexer**

The traditional multiplexer implied on the left is a very compact implementation, often within a single CLB. The various data sources distributed around the device must all connect to this one CLB; this leads to some long interconnections or a need to require the implementation tools to force all logic to compete for the same space. In contrast, the carry chain multiplexer implied on the right is distributed across several CLBs in a vertical column, which helps to separate logic and the interconnect. The high performance of the carry chain is then exploited to route the selections to the output.

Conclusion

So, the next time you find yourself describing a multiplexer, take a moment to think about the implementation. Where there isn't a power-of-two number of data inputs, will you decompose the multiplexer into smaller power-of-two multiplexers, or effectively increase size to the next power of two? Whatever your decision, be sure that you don't get more than you wanted.

Multiplexers are generally the glue of many systems. If the traditional implementation implied by HDL code is adequate for your design, then use it. However, if you have performance, size or cost challenges and find that multiplexers and their connectivity are a cause for concern, take some time to investigate these alternative techniques. If you do not have a power-of-two number of data inputs, consider either decomposing into smaller power-of-two multiplexers, or increasing to the next power-of-two. Consider alternative implementations using dedicated resets or carry logic. The requirement for these techniques to be an integral part of the decoding and functionality of the modules of your design may not make it easy to retrofit, but your next design will benefit from the variety of options available to you from the outset.

References

For additional information on using multiplexers in the Spartan-3 Generation FPGAs, see Chapter 8 of UG331, *Spartan-3 Generation FPGA User Guide*.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
2/4/08	1.0	Initial Xilinx release. Some content taken from previous web postings as a TechXclusive.

Notice of Disclaimer