FPGA Design for ASIC-Experienced Designers

Actel FPGAs allow designers familiar with ASIC and HLD flow to make an easy transition to FPGA design. The Actel flow is similar to the typical HLD flow, but optimum results are achieved only when the designer keeps in mind a few key concepts: the Actel architecture, coding style, and methodology. This application note introduces the ASIC-knowledgeable designer to designing with Actel FPGAs and describes in detail the key concepts required to obtain good synthesis results.

FPGA Design with High-Level Design Tools

Actel FPGAs closely follow the expected ASIC design flow. HLD design source can be simulated with a behavioral simulator and a test bench. Once the design simulates correctly, a behavioral compiler can generate netlists that can be used for postsynthesis verification. If the verification is successful, the netlist can be compiled in the back-end place-and-route tools from Actel Designer Series 3.1. Post-place-and-route timing information can be backannotated for final timing verification. In addition, Designer Series 3.1 allows timing constraint entry for place and route, which can help ensure that timing targets are achieved by the place-and-route tools.

This flow should be similar to the flow that experienced ASIC designers have used. Actel FPGAs can support this flow because of the good correlation between pre-place-and-route capacity and timing estimates and post-place-and-route results. This is possible because of the abundant routing resources available on Actel FPGAs. These resources ensure that place-and-route issues don't impact capacity and performance estimates.

The ASIC designer can ensure good synthesis results by keeping a few key concepts in mind when using holds for Actel FPGAs. Actel architecture, HLD coding style, and methodology are the most important concepts and they are covered in detail in the rest of this application note.

Actel FPGA Architecture

Actel FPGA devices are based on an architecture that is inherently friendly to high-level synthesis tools. The Actel logic module is optimal in implementing logic synthesized from high-level tools. The logic module is small enough to be efficient at implementing common functions produced by synthesis tools (larger logic modules waste silicon area when only simple functions are required), but it is powerful enough to fit synthesis-produced, speed-critical functions into a single logic module. (For example, a five-input NAND gate and a 4-to-1 multiplexor each fit into a single Actel combinatorial logic module.)

Actel devices also provide an abundance of routing resources by using an antifuse interconnect element. The routing resource available on every Actel device ensures that logic placed into Actel logic modules can be efficiently interconnected automatically. No designer intervention is required. FPGA architectures that use other interconnect technologies (such as SRAM or EPROM) must limit the amount of interconnect available on the device; otherwise, device sizes would grow beyond manufacturing limits. (For example, an 8,000-gate SRAM device typically has 100,000 switching elements; the same-capacity Actel device has 700,000 elements, with a significantly smaller die size.) Thus, the optimal module size and abundant routing resources of Actel devices make them uncommonly friendly to high-level design methodologies.

The Actel 1200XL device architecture is shown in detail in Figure 1. The device is similar in structure to a channeled gate array, with rows of logic modules separated by rows of routing channels. Outputs from the logic modules enter the routing channel and can be connected to any of the routing tracks in the routing channel. Routing tracks are segmented, offering the module output a variety of possible connections. Long connections are available for signals that must span a major portion of the device, and short tracks are available for signals that need to span only a short distance. Module outputs span multiple routing channels, giving each output access to all tracks in the two channels above and below. This large interconnect flexibility makes Actel devices very routable and ensures that timing and capacity estimates made by high-level tools can be achieved by the back-end place-and-route tools.

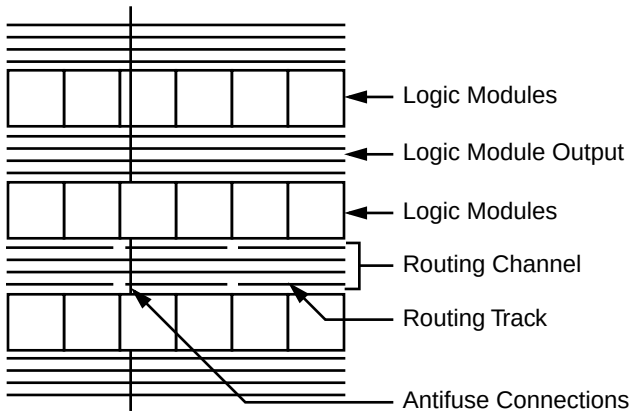


Figure 1 • *Actel 1200XL Device Architecture*

The logic modules used in the Actel 1200XL devices are shown in Figure 2. The combinatorial logic module is a 4-to-1 multiplexer with the addition of an AND gate on one select line and an OR gate on the other select line. This logic module can be used to construct a variety (over 760) of logic functions by connecting the appropriate logic signal and module inputs. Synthesis software automatically creates the required logic functions from these modules, so logic designers need not worry about the details of the logic implementation. (However, it will be shown that a little understanding of the multiplexer structure underlying the Actel architecture can be useful to the high-level designer.) The sequential module in the 1200XL family is similar to the combinatorial module, with the addition of a D-type register at the module output. One of the multiplexer select inputs loses an input to make the active low clear function on the register available. The register can also be turned into a level-sensitive latch, if preferred.

High-Level Design Techniques

It is inherently easy to synthesize logic for Actel devices, but the designer who understands some basics of the underlying architecture can improve the efficiency of the resulting logic. The two main categories under which these techniques fall are coding style and methodology. Coding style is an approach selected from the wealth of possible implementations available in most high-level tools to describe the required behavior of a logic function. Methodology covers the sequence and the various steps (flow) a designer selects to take a finished high-level design description and translate it into silicon.

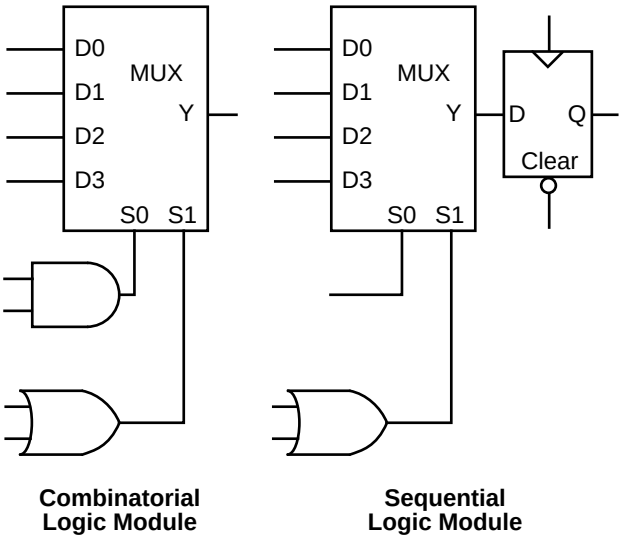


Figure 2 • *Actel 1200XL Logic Modules*

Coding Style

The coding style a designer uses can either help or hinder the synthesis tools in translating the desired logic function into the targeted FPGA. Only a small amount of knowledge about the underlying device architecture is needed to select between alternative approaches to defining a desired logic function. A simple but important example of this can be seen when implementing select functions in high-level languages. Two common alternative approaches to these functions (or two different coding styles) would be to use IF/THEN/ELSE statements or to use CASE statements. Each approach implements the same logic function, but the implementation in Actel FPGAs can be quite different. Figure 3 shows a 4-to-1 multiplexer described in a CASE statement in VHDL. The CASE statement in the middle of the code implements a logic function with one of the logic inputs (c, d, e, or f) selected by the 2-bit vector s4 and provided on the output m2y. The resulting implementation in an Actel logic module is also shown in Figure 3. Only a single logic module is required, since the synthesis software mapped the 4-to-1 select function into a 4-to-1 multiplexer.

Figure 4 shows an alternative implementation of the select function using IF/THEN/ELSE statements. Again, the 2-bit vector s4 is used to select one of the four inputs (c, d, e, or f). The function is identical to the function implemented previously by using the CASE statement. The resulting implementation in Actel, also shown in Figure 4, is much different, however. The IF/THEN/ELSE logic results in a multiple module implementation and increases both silicon area and delay.

A VHDL function with multiplexer coding style would be the following:

```
library ieee;
use ieee.std_logic_1164.ALL;

ENTITY mux IS
 PORT (c, d, e, f : IN std_logic;
       s4 : IN std_logic_vector(1 downto 1);
       m2y : OUT std_logic );
END mux;

ARCHITECTURE my_mux_behave OF mux IS

BEGIN

 CASE s4 IS
       WHEN “00” => m2y <= c;
       WHEN “01” => m2y <= d;
       WHEN “10” => m2y <= e;
       WHEN others => m2y <= f;
 END CASE

END PROCESS mux1

END my_mux_behave;
```

Synthesis would synthesize this VHDL as follows:

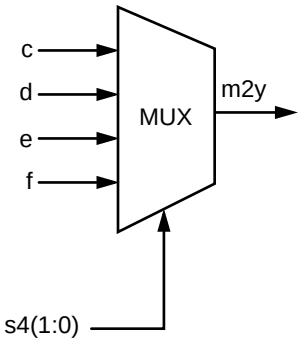


Figure 3 • *CASE Statement Example*

Thus, a designer who understands that Actel devices are optimized for multiplexer-based logic will use CASE statements when possible. This will result in the most efficient silicon implementation when targeting Actel devices, but it doesn't require any device-specific code.

Another architectural aspect of Actel devices that the knowledgeable designer should keep in mind is the structure of the register used in the synchronous logic module. The register is optimized for a D-type with an active low clear. In addition, since the register is preceded by the combinatorial logic module, many more complex register functions can be synthesized by the high-level design software. For example, Figure 5 shows the code for a clearable, D-type register with an Enable. The Enable function can be implemented by the synthesis software in a single logic module by using the multiplexer in front of the register in the synchronous logic module. Alternatively, when the designer uses random logic

The same VHDL function described previously using the CASE statement, can be written as follows:

```
library ieee;
use ieee.std_logic_1164.ALL;

ENTITY my_if_then IS
 PORT (c, d, e, f : IN std_logic;
       s4 : IN std_logic_vector(1 downto 1);
       m2y : OUT std_logic );
END my_if_then;

ARCHITECTURE my_mux_behave OF my_if_then IS

BEGIN

myif1 : PROCESS (s4, c, d, e, f)

BEGIN

if s4 = “00” then
   m2y <= c;
   elsif s4=”01” then
     m2y <= d;
       elsif s4=”10” then
         m2y <= e;
           else
             m2y <= f;
 endif

END PROCESS myif1;

END my_if_behave;
```

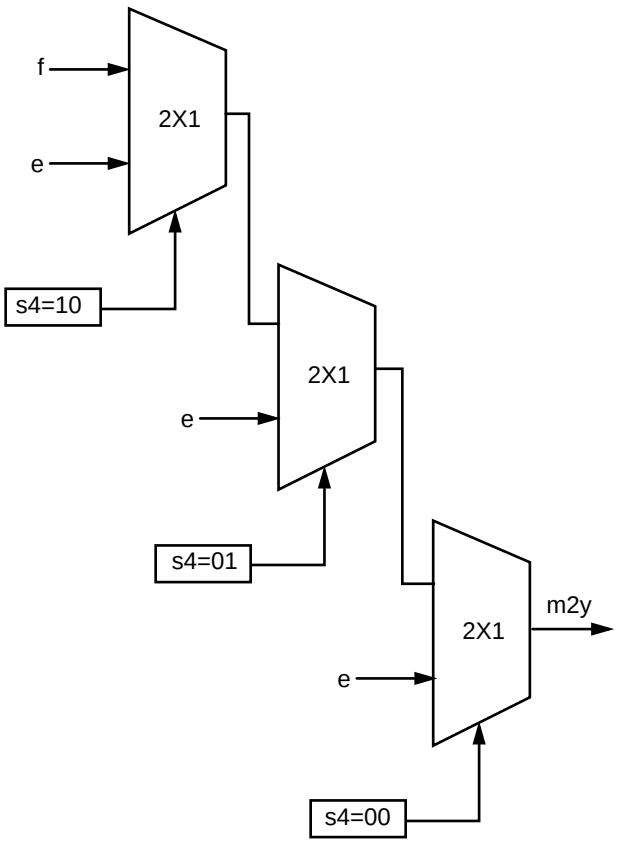


Figure 4 • *If-Then-Else Example*

in front of the register, the synthesis software will specify combinatorial logic, which can be implemented in the logic module in front of the register. Using the simple register description (simple D-type with active low clear) and CASE statements for data-path selection will ensure that functions such as a 4-to-1 multiplexer driving a D-type register efficiently map into a single Actel module.

```
IF (clear = '0') THEN
ELSIF (clk'event and clk='1') THEN
    IF (en= '1') THEN
    Q <= d;
END IF;
END IF;
```

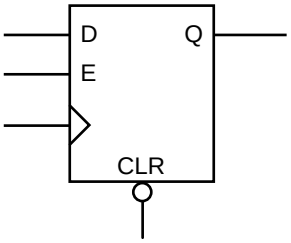
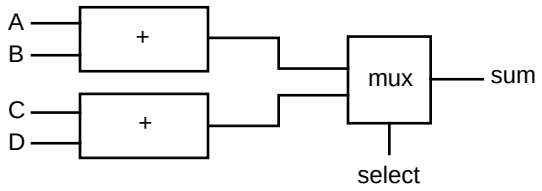


Figure 5 • *Clearable D-type Register with an Enable*

High-level designs often use complex data-path elements. The designer who understands the relative size and performance of these blocks can optimize the result by using proper coding style. Take for example a data path that requires the addition of two pairs of numbers. A sum needs to be generated for A_n and B_n, as well as for C_n and D_n. If the sums are not required simultaneously, the designer might decide to select the outputs after the sums are generated. The code for this approach is given in Figure 6a. First the additions are defined, and then the outputs are selected by using the IF/THEN/ELSE statement. The resulting implementation uses two adders and one multiplexer. If the designer realizes that an adder is much more expensive than a multiplexer (in terms of delay and module count), the definition can be reordered to do the selection first and then the addition. The code for this approach is shown in Figure 6b. The resulting implementation requires two multiplexers, but a single adder. This will result in considerable savings in both modules and delay. Notice that resource sharing approaches are architecturally independent. These savings would apply to almost any target technology.

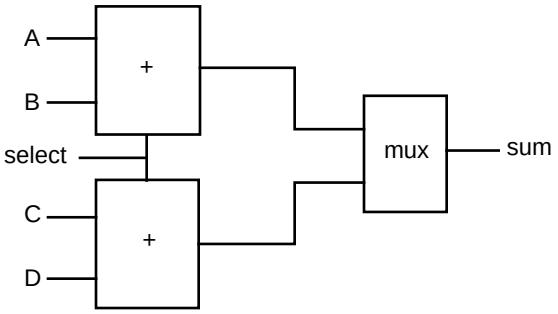
As shown earlier, coding style can have a major impact on the resulting silicon implementation. The designer who understands some key aspects of the underlying technology or the resource costs for key functions can improve the results generated from high-level design tools. With Actel devices, a designer needs only to use CASE statements, simple active low clearable registers, and resource sharing to improve synthesis results. The optimal logic module and abundant routing resources will ensure that the resulting logic will be efficiently mapped into silicon.

```
if (select)
   sum <= A + B;
else
   sum <= C + D;
```



(a) Poor Resource Sharing Example

```
if (select)
   temp1 <= A;
   temp2 <= B;
else
   temp1 <= C;
   temp2 <= D;
sum <= temp1 + temp2;
```



(b) Correct Resource Sharing Example

Figure 6 • *Resource Sharing Example*

Methodology

The methodology a designer selects for a design can also influence the efficiency achieved with synthesis tools. For example, FPGA devices that do a minimum amount of postprocessing on the output of synthesis tools to fit the logic into the device can provide accurate capacity and performance estimates to the high-level tools. This reduces the number of iterations required to go from a high-level description to a silicon implementation of the design. Some architectures must do postprocessing on the high-level synthesis tool output to fit the logic into large logic modules, while keeping the number of inputs to the module low so that routing resources are not overtaxed. Actel devices require very little postprocessing because of the optimal module size and abundant routing resources.

Another methodology issue related to postprocessing is backannotation of delays. If there is a good correlation between the high-level logic and the postrouting fit, accurate delay estimates can be provided to the synthesis tool for timing analysis and simulation. If postprocessing has split logic to improve routability, the correlation between logic delays and the high-level logic representation will be broken, reducing the accuracy of timing simulation to the point of uselessness. In these cases, timing must be handled differently and may break the desired development flow. Again, Actel devices have a high correlation between high-level timing estimates and post-place-and-route delays, making backannotation for simulation easy and accurate.

Actel place-and-route tools are timing driven (Designer Series 3.1), and the timing constraints can be passed from the high-level tools. This allows top-level timing requirements to be easily used by the designer to guide the low-level place-and-route software, without the need to learn new tools or introduce another source of data—thus minimizing the possibility of errors. This "forward annotation" of delays is a powerful feature of the Actel design flow. Pin assignment can also be forward annotated, making it possible to specify completely the function, pinout, and timing for the target device, all at the highest level of the design.

Library support is also an important aspect of the development methodology. Actel provides synthesis vendors with special libraries composed of predesigned macros for common logic functions. In addition, key high-level functions (such as adders, counters, and FIFOs) are predesigned by Actel and can be "called" by the synthesis tools on an as-needed basis. ACTgen Macro Builder is an Actel software tool that can create macros which can be instantiated in Verilog or VHDL designs for synthesis using Synopsys and many other industry synthesis tools.

These macros are defined in a high level language from which you can generate a gate-level Verilog and VHDL netlist. ACTgen can create additional macros that effectively use the Actel architecture to achieve optimum performance, and minimal module count to improve designer productivity.

As each application has its unique attributes, flexibility of design is important. For example, an application may require a nine-bit up counter with enable whereas, another application may require a 22-bit up/down counter. Building a library to support this type of flexibility is difficult; however, the ACTgen Macro Builder provides this needed flexibility for demanding applications.

For more information about ACTgen Macro Builder, refer to the "Designer Series Development System" document in Section 3 of this Data Book.

A new approach to synthesis just now reaching the market does not use the library approach for random logic Boolean equations; instead it implements logic directly in the Actel logic module. This "libraryless" approach improves efficiency since the Actel module can create so many logic functions. Because it is unwieldy to create a library containing all possible elements, the required logic function is generated instead by using the Actel logic module directly. This improves efficiency when the function required is not a common function and would be overlooked by a library-based approach. For example, the logic function shown in Figure 7 would normally map into two logic modules, one a two-input XOR and the other a two-input OR. Synthesis software that maps directly into the Actel logic module (CM8) would find the solution where the entire function fits into a single Actel logic module. The resulting implementation, shown in Figure 8, requires a single logic module and only a single level of delay.

Conclusion

Actel devices are inherently efficient for high-level designs because of their optimized logic modules and abundant routing resources. Experienced designers have found ways to squeeze even more efficiency out of these devices by understanding the influences of coding style and methodology on the resulting silicon implementation. Simple things such as using CASE statements and active low clearable registers can get the last drop of efficiency out of a design. The ability of high-level tools to accurately estimate capacity and performance when mapping to Actel devices also increases the efficiency of using these tools. Using Actel devices and high-level design tools, a designer has the capability to move up to higher-complexity designs.

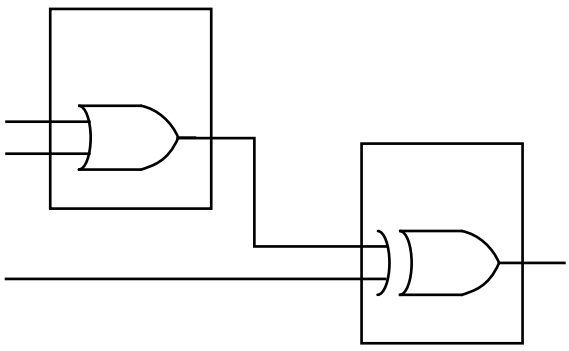


Figure 7 • *Two Module Implementation*

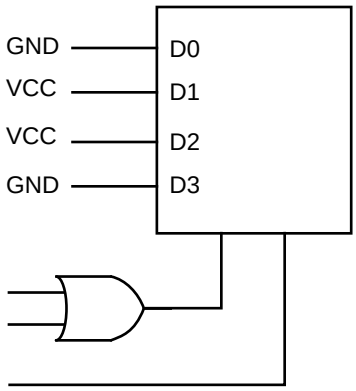


Figure 8 • *Single Module CM8 Implementation*