ProASIC^PLUS® Design Optimization

Introduction

This application note discusses various procedures for area and speed optimization during ProASIC^PLUS design. Techniques for achieving high design performance are important when designing for FPGAs. Similarly, area optimization is required for some designs. Optimizing for area often means larger delays, and you must weigh your performance needs against your area needs to determine what works best for your area design.

Various optimization and timing closure techniques exist, but the effects of these techniques vary from design to design. Applying a single technique does not always improve design results. You can use optimization techniques or adjust the default settings in various tools during design flow, or use both approaches to achieve the best results for your design.

This application note assumes that you have knowledge of Actel Libero® Integrated Design Environment (IDE) design flow. Actel Libero IDE offers synthesis, physical synthesis, and other third party tools. Libero IDE also includes the Designer place-and-route tool. Designer allows detailed timing analysis of your design, including a fully integrated Timing Closure and floorplanning tool. With these tools, you can easily determine and locate critical paths in the targeted device floorplan. Once the critical paths are analyzed, you can use various techniques to optimize their design.

This application note will cover various techniques for design optimization, starting with design architecture and HDL coding. It will also provide optimization techniques using various tools during design flow and explore these techniques to determine which provide the best results for your design. Finally, this application note will cover the overall design flow and give a few design tips and recommendations.

Overview of ProASIC^PLUS Architecture

You must understand the ProASIC^PLUS architecture in order to evaluate the performance of your ProASIC^PLUS design and determine the best optimization techniques. The ProASIC^PLUS device core consists of a Sea-of-Tiles. Each tile can be configured as a three-input logic function (e.g., NAND gate, D-flip-flop, etc.) by programming the appropriate Flash switch interconnections. The logic tile cell has three inputs (any or all of which can be inverted) and one output. Any three-input, one-output gate (except a three-input XOR) can be configured as one tile. The tile can be configured as a latch with clear or set, or as a flip-flop with clear or set. You can also implement flip-flops with both set and clear, but it will require multiple tiles.

ProASIC^PLUS devices also contain embedded, two-port SRAM blocks with built-in FIFO/RAM control logic. They do not have any special carry chain logic. The routing structure of ProASIC^PLUS devices is designed to provide high performance through a flexible four-level hierarchy of routing resources:

- Ultra-fast local resources
- Efficient long-line resources
- High-speed, very long-line resources
- High performance global networks

The ultra-fast local resources are dedicated lines that allow the output of each tile to connect directly to every input of the eight surrounding tiles (a typical delay of 0.3 ns). The efficient long-line resources provide routing for longer distances, spanning one, two, or four tiles. The high-speed, very long-line resources, which span the entire device with minimal delay, are used to route very long or very high fanout nets. The long-line resources can go across the chip, but the delay can be large depending upon macro placement.

The ProASIC^PLUS architecture also contains four segmented global networks that can access all the logic, memory, and I/O tiles on the die (a typical delay of 1.1 ns). You should have a basic understanding of the core tile, memory architecture, and the routing resources, and give attention to these during the design flow. Refer to the *ProASIC^PLUS Flash Family FPGAs* datasheet for details on ProASIC^PLUS architecture.

Design Optimization Overview

The Actel design flow uses various Actel tools as well as third party tools. Figure 1 shows the Libero IDE Design Flow. Note that physical synthesis is an optional step in the design flow and may not be needed for most designs.

You can do advanced design optimization using improved design techniques, or with the automated tools used in the design flow, or both. The design techniques will provide more predictable results and may be easier for the expert user. But it may also require changing the default settings for the design flow tools. This will be discussed in the "General Recommendations" section on page 23. First, we will go over the various design techniques and then we will go over optimization using the various design tools.

The general design technique, default flow, and settings will be sufficient for most designs. For some designs, you may need to use these advanced techniques or advanced settings.

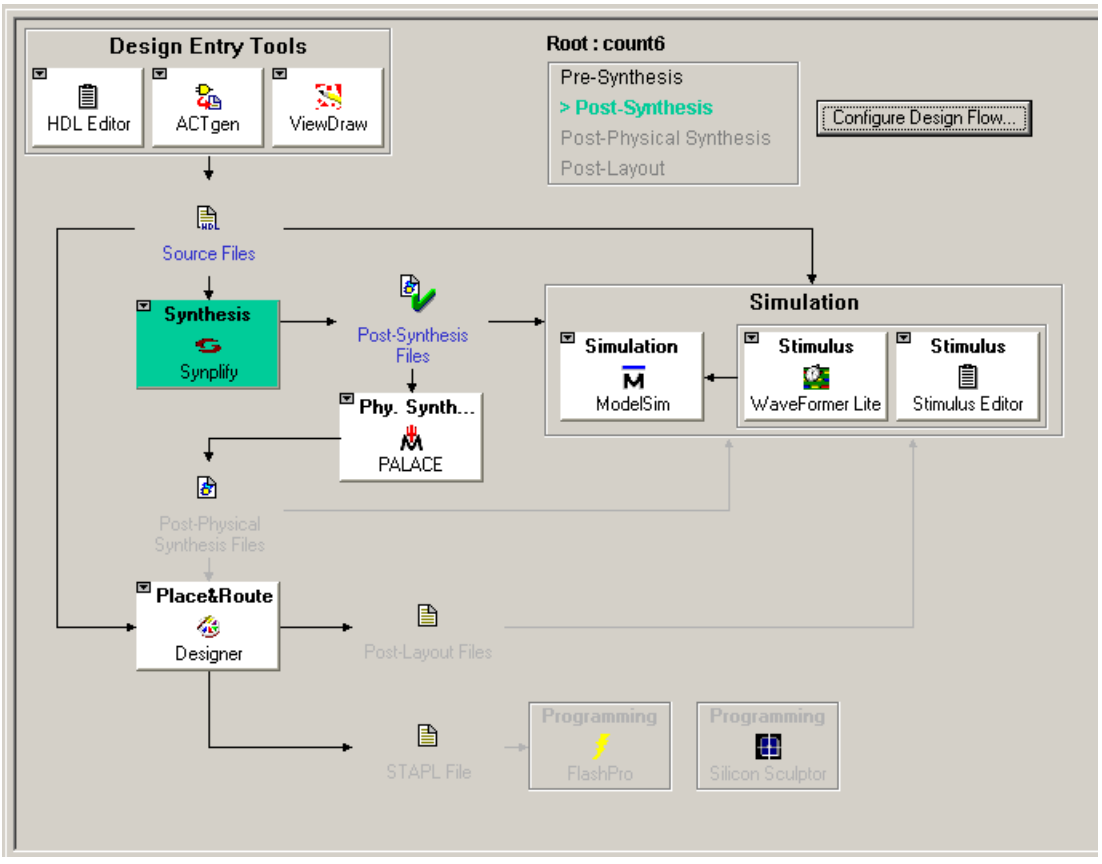
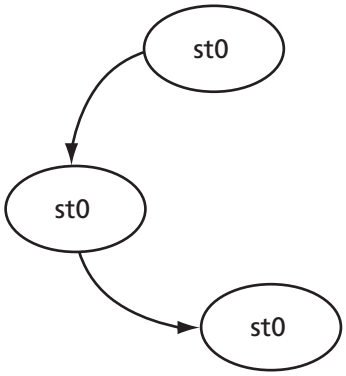


Figure 1 • **Libero IDE Design Flow**

Design Architectural Level Consideration

The ProASIC^PLUS device core consists of a Sea-of-Tiles and has a fine-grained architecture. In the design, the data travels through several layers of logic between the registers. In general, you will see an average delay of 2 ns or less per level of logic after place-and-route. This should be considered when the design is architected, so Actel recommends that you try to estimate the levels of logic while architecting the design. Since the logic is 3-input combinatorial cells, multiply the estimated logic levels by two to get the approximate delay. This approximate delay is reasonable for a typical design. If the approximate delay seems to be greater than the requirement, you must pipeline the design. Note that if there is an opportunity to pipeline a non-critical path, you should do it. This gives place-and-route more margin on non-critical paths, which increases the chance of meeting delay on critical paths.

Depending on the design, there could be several multicycle or false paths. This is true specifically for designs with state-machines. When designing a state-machine, insert wait-states wherever possible and declare associated logical multicycle paths. This allows the place-and-route tool to focus on true critical paths. As an example, consider the state machine shown in Figure 2. The registers reg1 and reg2 are updated in st0 and also act as inputs to an adder function. The output of adder is used in st2. Since there is no direct transition from st0 to st2, paths through this adder can take two clock cycles to propagate. During design flow, you should use an SDC multicycle paths constraint with a path multiplier of 2 so that the place-and-route can focus on other critical paths. Refer to the *ProASIC^PLUS Timing Closure in Libero IDE v5.2* application note for details on setting the SDC constraint and how Timing Driven Place-and Route (TDPR) handles these constraints.




```
If (st0)
reg1 <= data1;
reg2 <= data2;

If (st1)
reg3 <= reg1;

If (st2)
reg4 <= reg1 + reg2;
```

Figure 2 • **State Machine**

While architecting the design, follow the recommendations below to partition the design:

1. Keep the critical path within one block. Actel Designer software allows you to use the set_location constraint to contain logic within a logical block that will be placed within a physical boundary. This helps to contain all portions of the critical path within a close proximity.
2. Register the input/output port for the blocks. When creating different blocks, try to register to output. This will make it easier to meet the timing requirements when you add more blocks.

For the design using heavily loaded internal buses, consider adding a wait-state to allow more than one clock cycle if a single clock cycle cannot be met. You can also consider replicating the driver manually to reduce fanout while avoiding the congestion associated with replicated nets. These techniques are covered in the "HDL Coding" section on page 5.

Designing the Memory Blocks

One of the important issues during design architecting is the memory block design consideration. The basic memory block size for ProASIC^PLUS is 256x9. You should architect the design to make best use of this basic block, using memory in multiples of 256x9 (or 256x8). Rather than using a large quantity of small memory blocks, it is better to combine these small memory blocks into a large memory block. On the other hand, do not use excessively large memory blocks. That would force the design to use multiple memory blocks by cascading in depth or width, and would increase memory access time. ProASIC^PLUS memory also supports parity checking while writing/reading for memory, and no extra logic is required to generate this parity logic. Refer to the *ProASIC^PLUS Flash Family FPGA datasheet* for more information.

Designing the Internal Tristate

ProASIC^PLUS devices support tristates as output ports, but not as internal tristates. All internal tristates must be mapped to gates. In most cases, you do not need to convert internal tristates to multiplexers (MUXes) manually. The Synthesis tool will attempt to map it to equivalent MUX or and-or logic structure. Actel recommends that you do not use internal tristates inside the RTL code. As shown in Figure 3 and in Figure 4 on page 5, internal tristates map to gates during synthesis. But, if you use the multiple edifs flow and attempt to generate an edif netlist for the tristate block, Synplify® will generate an edif netlist with a tristate macro at the port. Tristate is not supported at the lower level, so you would not be able to use this edif netlist with the other edif netlists. For this reason, Actel recommends that you replace these internal tristates with MUXes or and-or logic structure.

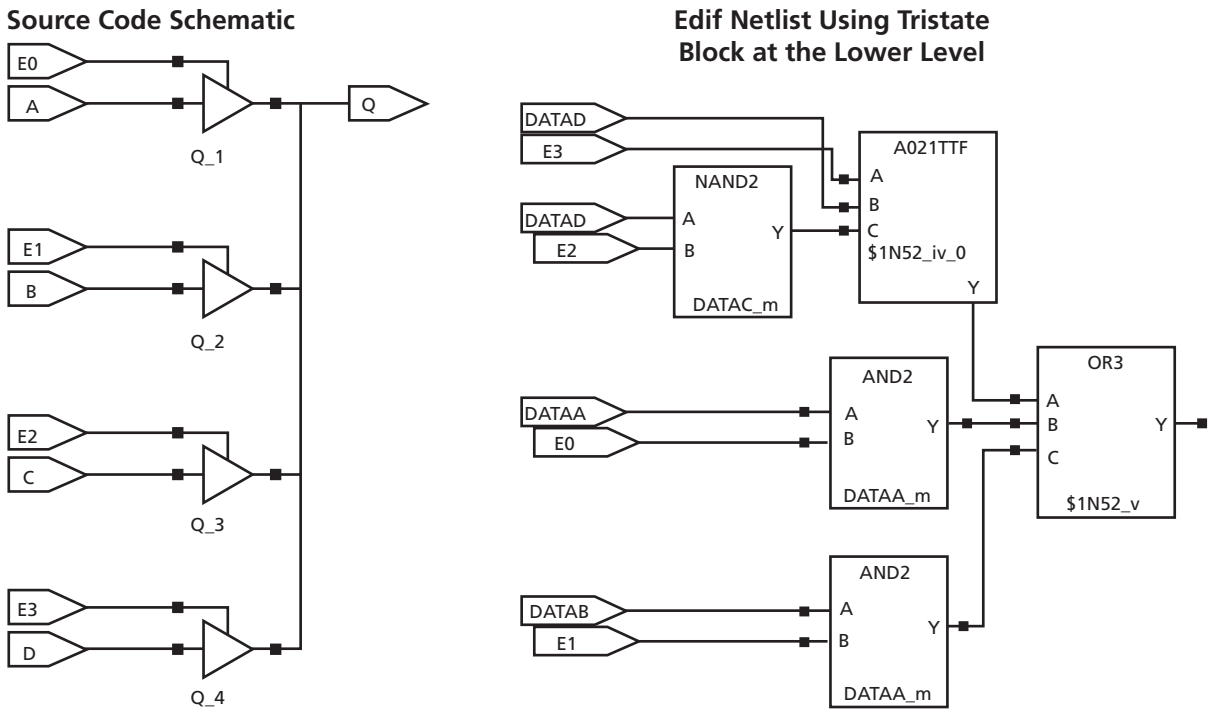


Figure 3 • **Mapping for Internal Tristate Using Block at the Lower Level**

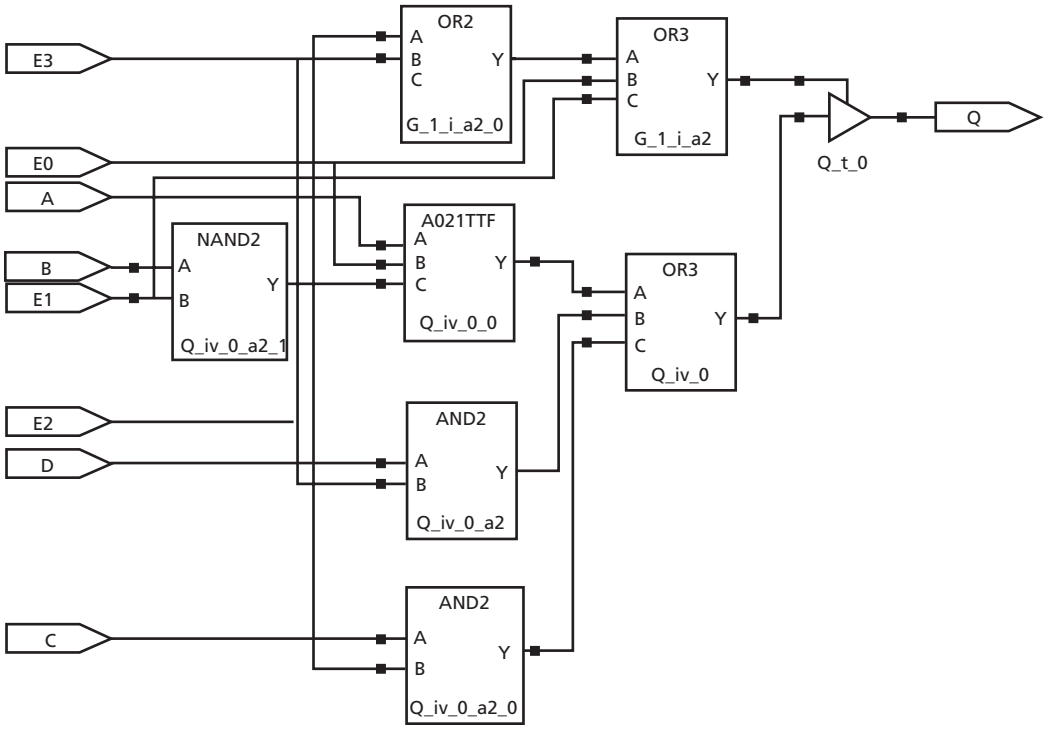


Figure 4 • **Mapping the Internal Tristate at Top Level**

HDL Coding

Once the architecture of the design is decided, you must partition and code the design into multiple blocks. Depending on how a design is coded, results can change dramatically. When writing the HDL code, keep in mind the guideline of 2 ns of delay per level of logic. Estimate the number of levels of logic and multiply it by 2 to get the clock cycle requirement. Do not try to exceed the clock cycle requirements. Here are a few techniques that can be used to reduce delay while keeping the same functionality:

Anticipation

Most of the control signals in a design are generally high fanout signals. These signals are generated on certain events and control how some registers are updated. Most of these control signals are decoded from counters or state-machines. These paths will start from a counter or state-machine, go through a decoder, and drive a high fanout signal that will go to the enable of several registers.

The pseudocode below serves as an example. There is an 8-bit down counter and the terminal count is used to update a wide 64-bit vector. The block diagram is shown in Figure 5.

```
@(posedge clock)
                        if (condition_1 == true)
                                    count <= 8'hff ;
                       else if (condition_2 == true)
                                        count = count - 1;    //down counter
@(posedge clock)
                       wide_bus[63:0] =  (count == 8'b0)? Data1: data2;
```

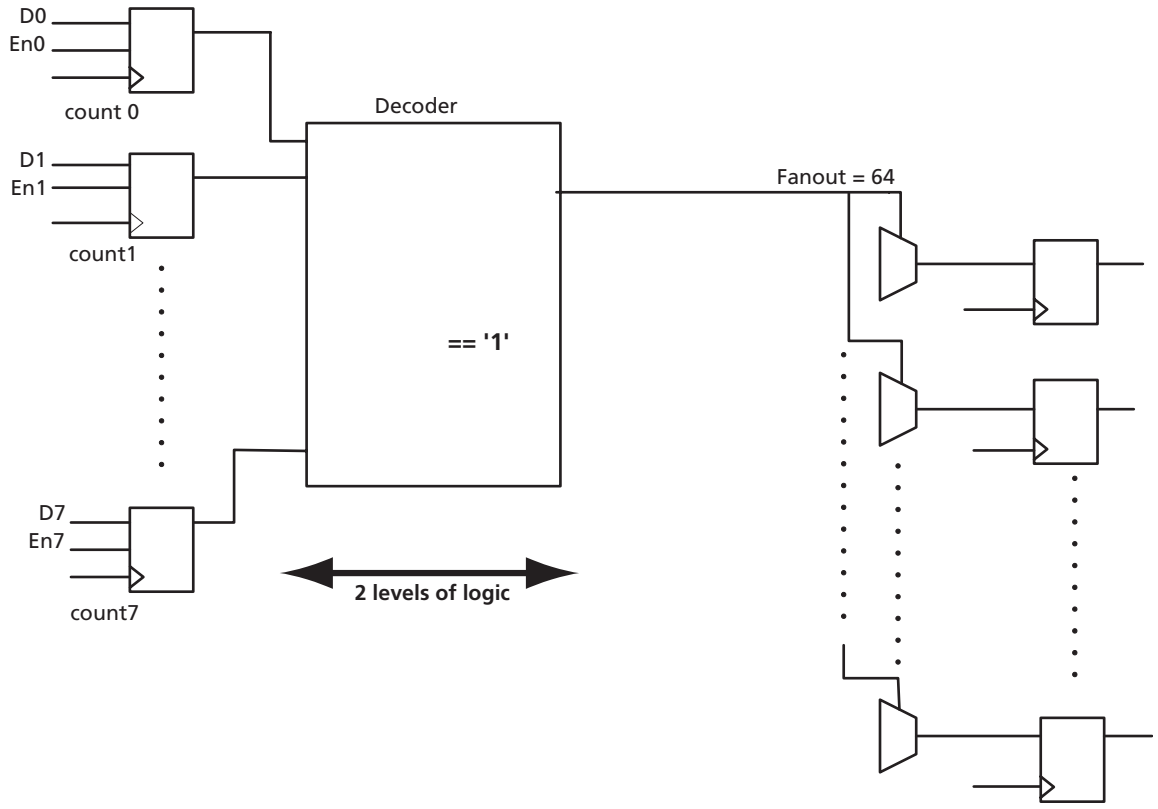


Figure 5 • **8-Bit Down Counter Controlling a Wide 64-Bit Vector**

Because the counter is an 8-bit counter, it will take two levels of logic to decode, provided the Synthesis tool uses a three-input gate during optimization. The decoder output will have a fanout of 64 and this net is driving the select lines of 2:1 MUXes. So, in the critical path there are three levels of logic (two for the decoder and one MUX) and also a high fanout net of 64. The delay on the 64 fanout net can be quite high and a reasonable assumption of the delay per level on this path will be around 3 ns. Once you add these delays, you will get 2 + 2 + 2 + 3 = 9 ns. So, you can run this design at approximately 110 MHz.

If your design requirement is greater than 110 MHz, you can rewrite the code as shown below.

```
@(posedge clock)
                        if (condition_1 == true)
                                  count <=  8'hff;
                                  count_is_0 <= 0;
                       else if (Condition_2 == true)
    begin
                                       count <= count - 1; //down counter
                                       if(count  ==  8'h01)
                                       count_is_0 =   1;
                                      else
                                        count_is_0 =   0;
    end
@(posedge clock)
                        wide_bus[63:0] =  (count_is_0 = = 1) ?Data1: data2;
```

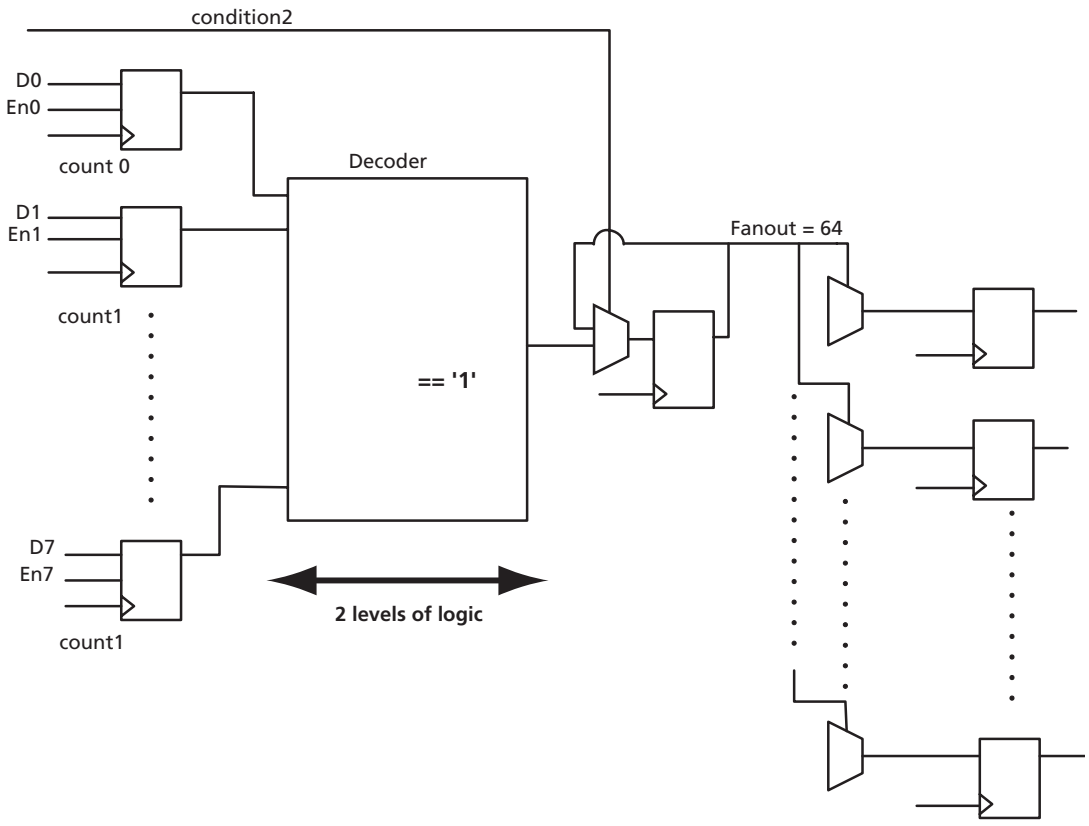


Figure 6 • **Modified Counter Design with Added Register**

This functionally of the new code is identical to that of the original code. In this code we will anticipate in advance (one cycle before) when the down counter is approaching 0. The decoder will drive a register, which will drive the MUXes. The decoder path will have three levels of logic with a fanout of 1, and the high fanout select path will have only one level of logic. The approximate delay between registers using the modified code is around 6 ns and the logic will run at approximately 180 MHz.

Instantiation

While synthesizing the design with Synplify or another tool, the intended results may not occur. Various synthesis options using Synplify are covered in the "Synthesis with Synplify" section on page 16. But, you can control the synthesis result by instantiating a gate level macro in the HDL code, and in most cases this will provide better results. Consider the following two examples.

Example1

Consider the decoder used in the anticipation section. You can map to the decoder symmetrically or asymmetrically, as shown in Figure 7. If all the counters output are critical, you should use the symmetric mapping. If the output of one counter is in the critical path, use the asymmetric mapping. For predictable results, Actel recommends instantiating the decoder code in the design.

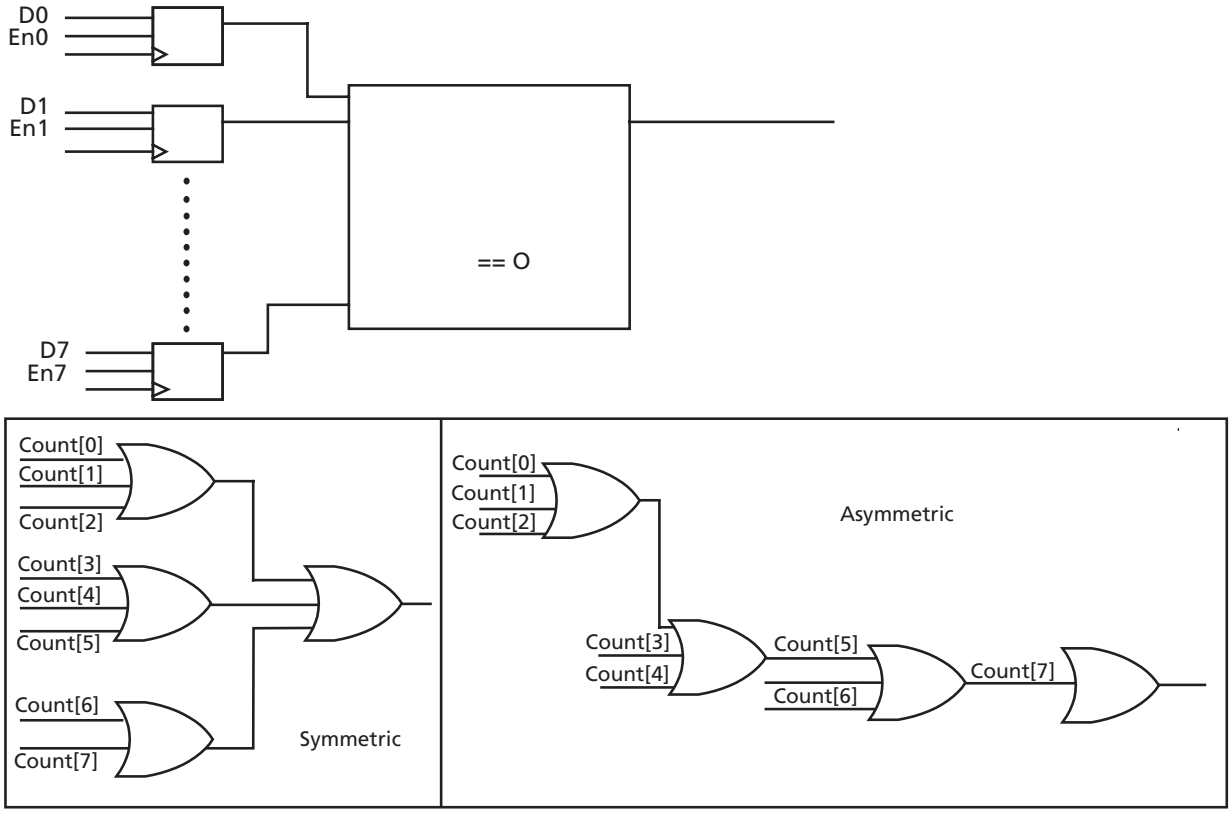


Figure 7 • **Mapping Decoder Block**

Example2

Consider the following counter code:

```
reg [7:0] count;
@(posedge clock)
      if(condition == true)
              count = count + 1;
```

During synthesis, you may map the code as shown in Figure 8 on page 9. If the counter condition is decoded from several other signals, it will have decoder logic followed by a long carry chain through the adder. This implementation is good for architecture with a carry chain. ProASIC^PLUS does not have a carry chain macro and you may need to put in a lot of effort to meet the timing goal during place-and-route.

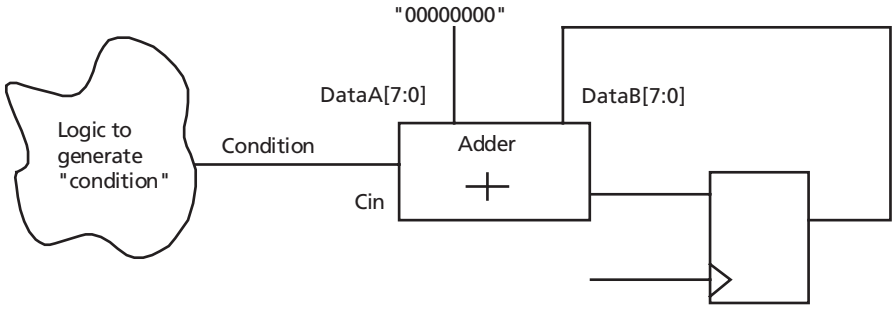


Figure 8 • **Simple Counter with Condition Assignment**

You can modify this code as follows:

```
inc8bit instance1_inc8bit(.DataA(count), .Sum(count_inc));    //instantiate an 8-bit
                                                              //incrementor

      @(posedge clock)
                  if(condition == true)
                              count = count_inc;
```

Here we have instantiated an incrementer in the code for the counting operation as shown in Figure 9. The condition signal will pass through MUX rather than through the carry chain logic, thus reducing the delay.

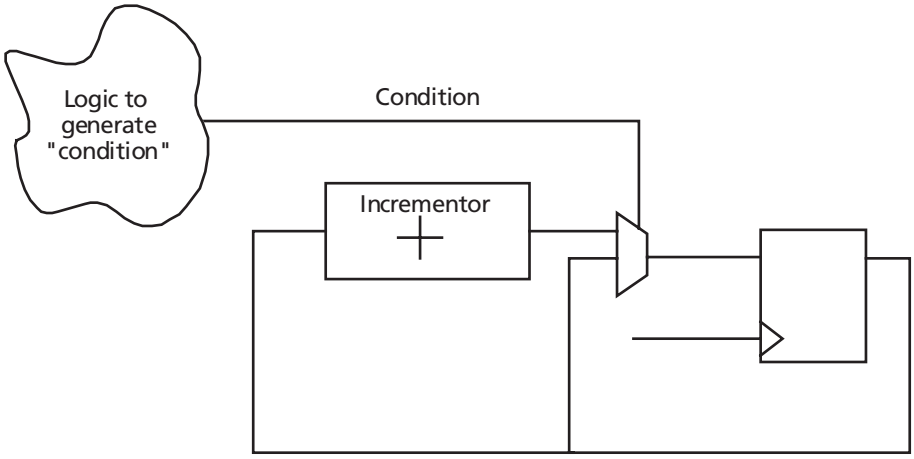


Figure 9 • **Modified Counter with Condition Assignment**

Replication

ProASIC^PLUS routing has built-in buffering for routing channels at repetitive intervals, so Designer removes all the buffers and inverters from the netlist by default. Occasionally the built-in buffering is not sufficient for a design with several high fanout nets. The Synthesis tool allows you to replicate drivers for high fanout nets. Doing a great deal of replication all over the design may increase congestion, leading to excessive timing. Since synthesis has no knowledge of how logic will be placed, it will distribute the load on replicated drivers randomly. During place-and-route, the same logical net may be going in the opposite direction.

Consider the example shown in Figure 10 on page 10. The register FF1 is driving 3 blocks. The output of the FF1 has a fanout of 16 inside BLK2, 24 inside BLK3, and 16 inside BLK4. If you allow synthesis to do replication globally, the result may be a congested design as shown in Figure 11 on page 10.

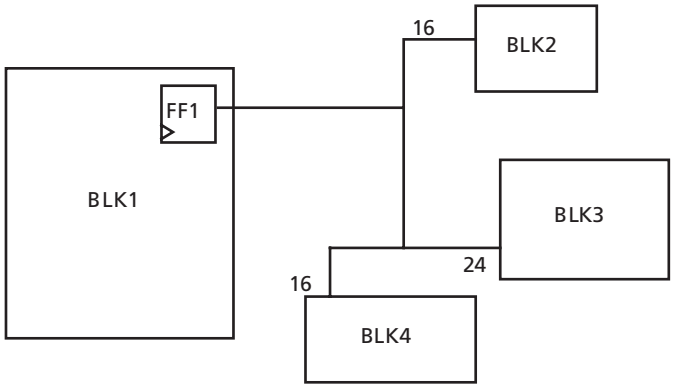


Figure 10 • **FF1 is Driving Signal Inside Three Blocks**

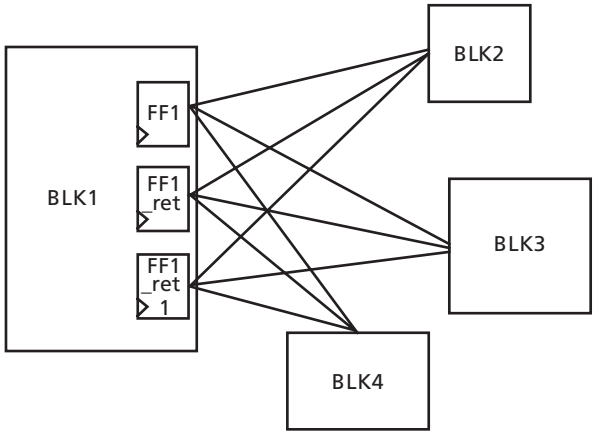


Figure 11 • **Synthesis Register Duplication May Cause Congestion**

You can avoid this by replicating the net driver manually in HDL (replicating the FF1 manually). Instantiate two duplicated registers (FF_cp1 and FF_cp2), as shown in Figure 12 on page 11, and connect them to different blocks. Note that if you use the global retiming options in Synplicity, this may again create congested routing. You can turn off the global retiming option for BLK1 using the Synplify attribute, as shown below. This will reduce the congestion and improve the timing for the design.

```
define_attribute {BLK1} syn_allow_retiming {0}
```

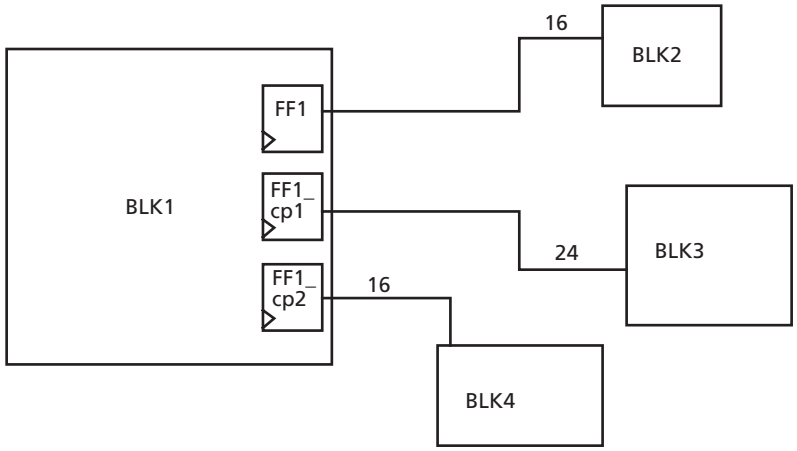


Figure 12 • **Manual Register Duplication to Reduce Congestion**

Redundant Priority

An If-Else statement is used to execute sequential statements conditionally based on a value, whereas a case statement implies parallel encoding. In general, If-Else constructs are much slower for parallel encoding. The If-Else statements are appropriate to use for priority encoders, assigning the highest priority to a late arriving critical signal. Consider the implementation for an If-Else statement shown in Figure 13. The signals e and f will have high delay going to output.

For the case statement, the signal will have less delay than the If-Else statement. The implementation is shown in Figure 14 on page 12. If e or f is in the critical path, it is better to use the case statement. To quickly spot an inefficient nested if statement, scan code for deeply indented code. To avoid long path delays, do not use extremely long nested If-Else constructs. In general, use the case statement for complex decoding and use If-Else statements for speed-critical paths.

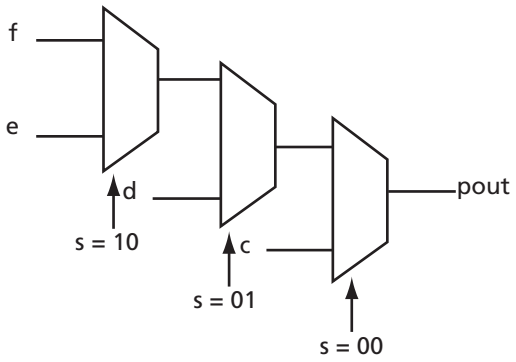


Figure 13 • **Mapping If-Else Statement to a Priority Encoder**

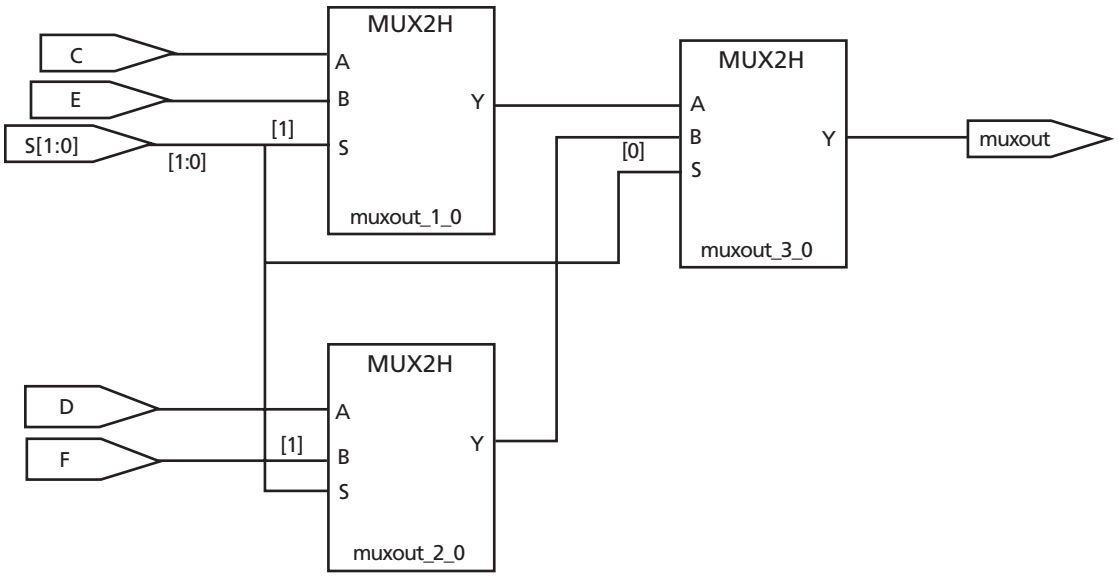


Figure 14 • **Mapping a Case Statement**

Timing Exceptions

When creating HDL, be aware of the timing exceptions and apply them in the place-and-route tool. In the design, you can have long paths that may be allowed more than one clock cycle to propagate. You can use the timing exceptions constraint so that Designer has less difficulty with these long paths.

As an example, consider the design shown in Figure 15 on page 13. In this case, there are three enable signals (E1, E2, and E3) used for the three FFs. The signal E2 comes two cycles later than E1, as shown in Figure 15 on page 13. In the first clock edge, the data will pass to the output of FF1. In the third clock edge, the data will pass to the output of FF2. In the fourth clock edge the data will pass to the output of FF3. Assume that after the initial place-and-route, you found that the delay from CLK:FF1 to D:FF2 is 25 ns and the delay from CLK:FF2 to D:FF3 is 45 ns, which is more than one clock cycle. You need to optimize the timing for CLK:FF2 to D:FF3.

This design does have enough margin for CLK:FF1 to D:FF2 to meet the timing. During TDPR, set multicycle from CLK:FF1 to D:FF2 and force the TDPR to focus on a CLK:FF2 to D:FF3 path. This may enable you to meet timing constraints. Another way to fix the violation is to modify the code so that EN2 comes one cycle later than EN1, and EN3 comes two cycles later than E2. Then place-and-route should be able to meet timing constraints easily, since the CLK:FF2 to D:FF3 path becomes a two-cycle path.

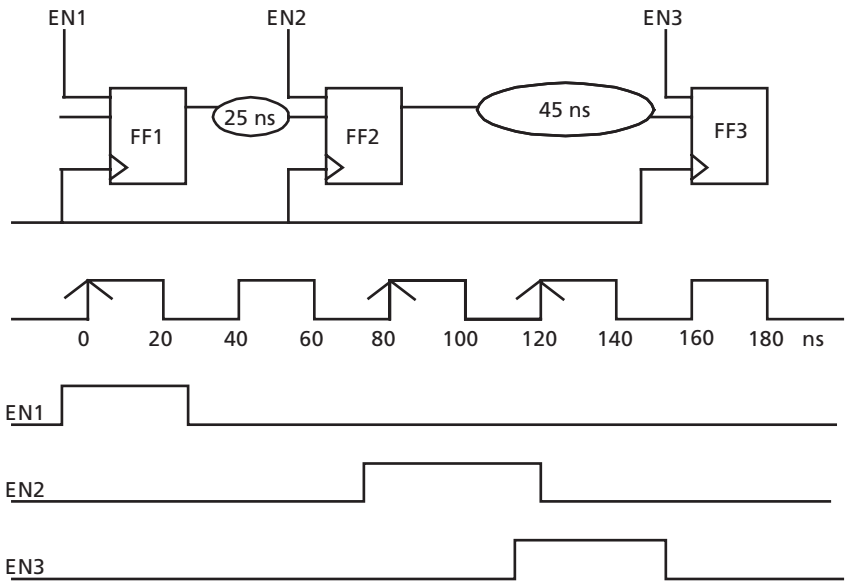


Figure 15 • **Multicycle Design Example**

Additional Tips for HDL Coding

Using Core Tile Instead of RAM Blocks for Small Memories

ProASIC^PLUS has dedicated memory blocks located at the top and bottom of the die. In some cases, the routing between the core tile and the memory blocks may add a long delay. If the design does not require too much memory, you can use the register file to generate memory. The ACTgen tool allows you to generate memory, using core tiles called "distributed memories," and helps to reduce the accessing time.

Modifying Extra Logic on the FIFO Flag from the ACTgen Netlist

During typical design flow, Actel recommends that you use ACTgen to generate memory blocks. During the generation of wide-cascade memory blocks with ACTgen, the code will use OR logic for all the flags. It will use the flag for all the blocks, and then use the OR function to generate the final flag. If your design requires FIFO flags and wide-cascade FIFO implementation, you can use a flag from only one block instead of using the final flag.

As an example, suppose you want to build a FIFO configuration of 32x32 and use the full and empty flag to control other parts of the design. With the default implementation, ACTgen will use four memory blocks and use OR logic for the flags, as shown in Figure 16 on page 14. For this configuration, if one FIFO block is full or empty, all other blocks are full or empty, since data are written simultaneously to all four blocks. You can modify the ACTgen-generated code, eliminate the OR gates used for flags, and take output from one block, as shown in Figure 17 on page 15.

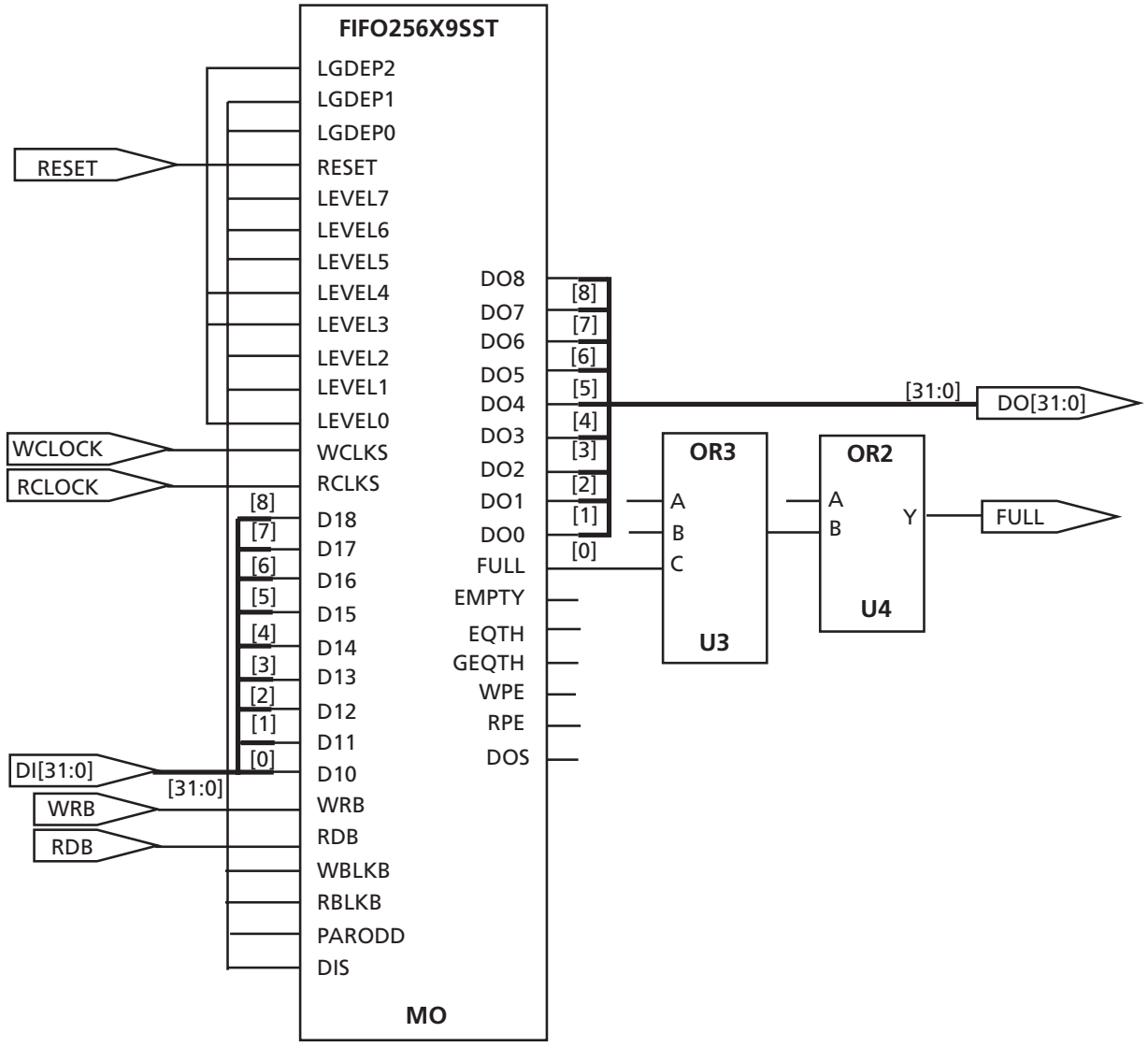


Figure 16 • **Cascaded Memory Blocks from ACTgen**

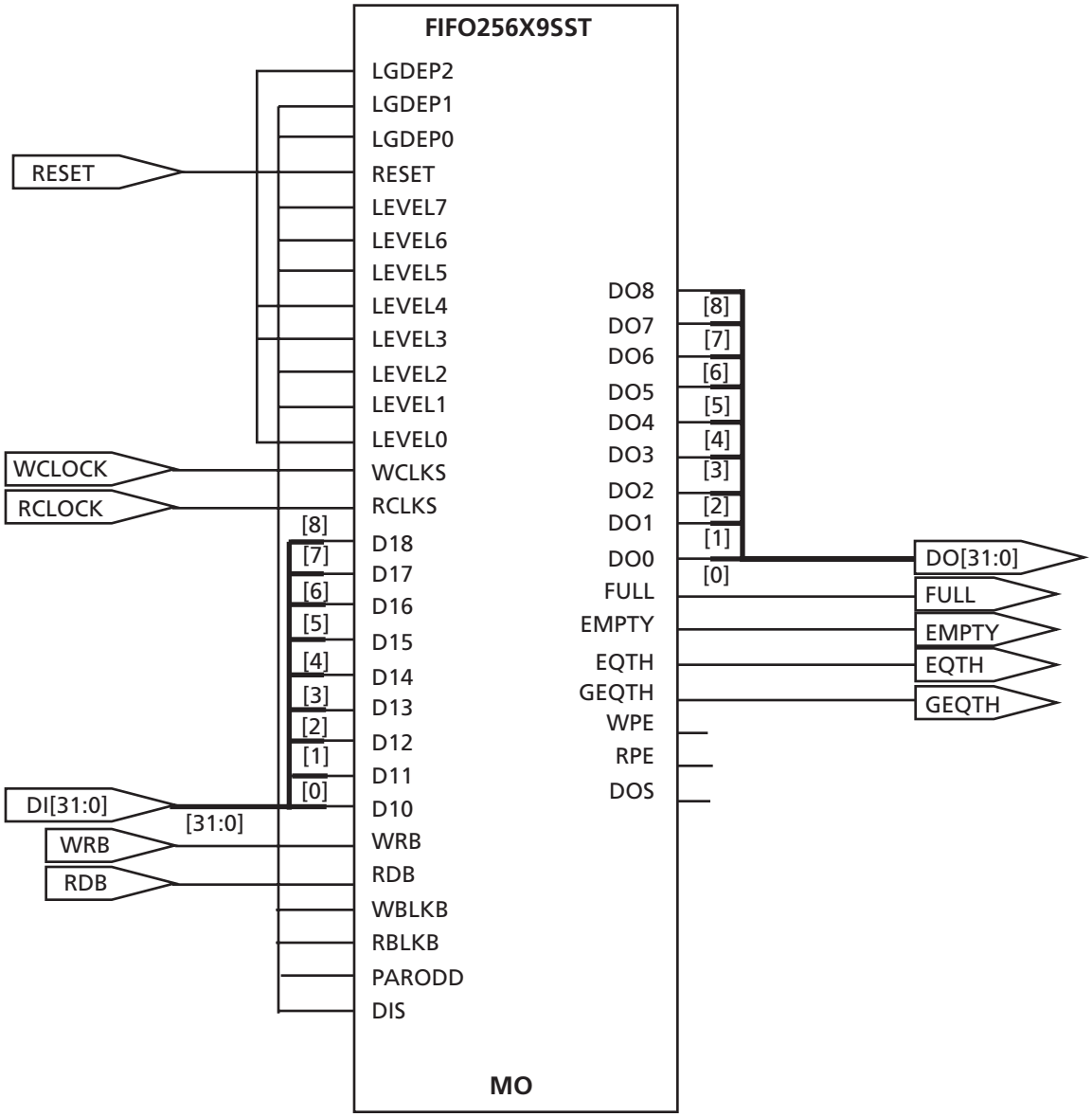


Figure 17 • **Modified Cascaded Memory Blocks**

Register the FIFO Flag

ProASIC^PLUS FIFO has four flags: empty, full, eqth, and geqth. These flags are generated on the falling edge of the clock, as shown in Figure 18 on page 16. This may cause a half-cycle issue and your design may have difficulty meeting it. Actel recommends you add a register at the output of the flags, as shown in Figure 19 on page 16. This may increase your latency, but it increases your throughput.

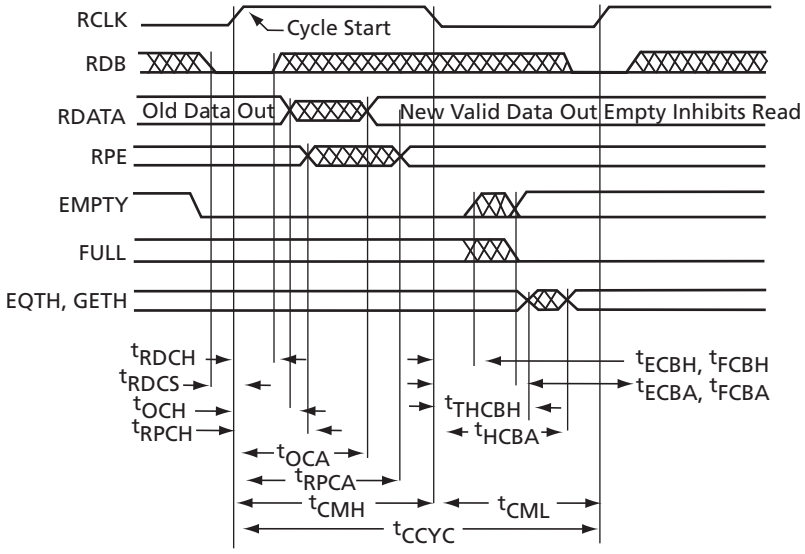


Figure 18 • **ProASIC^PLUS FIFO Flags Timing**

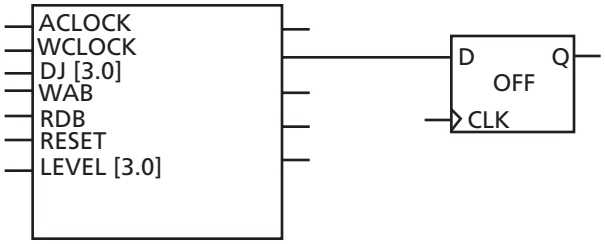


Figure 19 • **Register the FIFO Flag**

Optimization Using Design Flow Tools

The Actel design flow uses various Actel tools as well as third party tools. Figure 1 on page 2 shows the Libero IDE Design Flow. The default flow and settings will be sufficient for most designs. For some designs, you can change the default settings and may need to come back to the beginning or middle of the flow, after finishing place-and-route, to use the iterative flow. The following sections will cover the various techniques you can use when the default flow and settings do not provide the desired results.

The main steps during design flow are as follows:

- Synthesis with Synplicity
- Physical synthesis with PALACE®
- TDPR (timing driven place-and-route) and floorplanning in Designer

The various optimization techniques for these steps are explained in the following sections.

Synthesis with Synplify

The default Synplify options seek to achieve the best result with various tradeoffs. This section describes optimization techniques using Synplify when the default settings do not provide the desired result. This section will provide the efficient design practices that you can use to reduce logic levels on a critical path. First there are three important features for optimization that are available in Synplify: resource sharing, retiming, and multipoint synthesis. These sections are followed by a review of general optimization tips.

Resource Sharing

Resource sharing is an optimization technique in the synthesis tool that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL code. One way to optimize area is to use resource sharing. This is on by default. You can improve timing by disabling resource sharing, but at the expense of increased area. As shown in Figure 20, the default resource sharing will share the adder, and this will cause a high logic level for the enable signal, E1. If you turn it off, then enable E1 will have fewer logic levels and less delay, as shown in Figure 21. If the enable is in the timing critical path, you can turn off resource sharing for this block to get better timing.

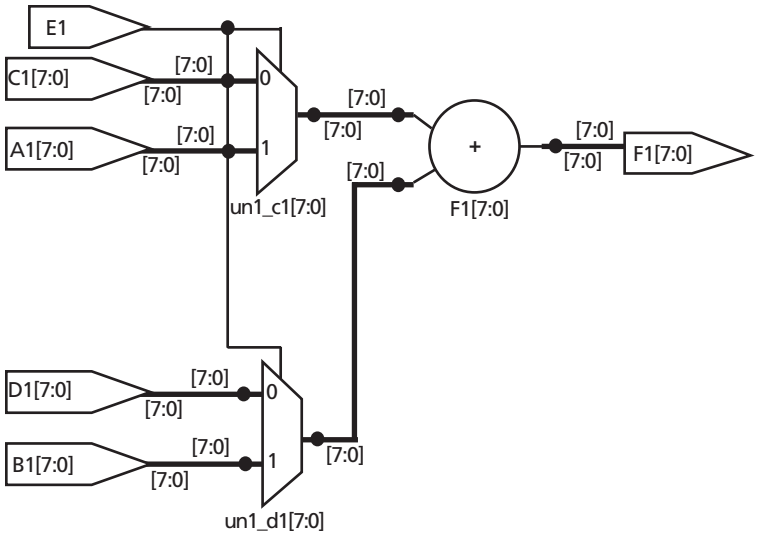


Figure 20 • **Resource Sharing in Synplicity**

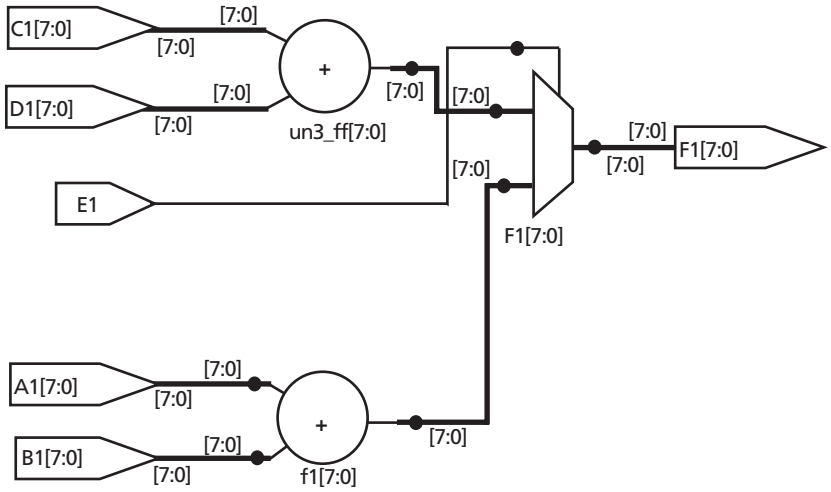


Figure 21 • **No Resource Sharing in Synplicity**

Retiming

Retiming is only available in the Synplify Pro tool. Retiming allows improvement of the timing performance of sequential circuits without the necessity of modifying the source code. It moves and also adds registers across combinatorial gates to improve timing without changing functionality between the inputs and outputs of the design. It does not change the number of registers in a cycle or path from a primary input to a primary output, but increases the total number of registers in the design.

Consider the example in Figure 22 and Figure 23. This example shows how, after retiming, the two registers on the inputs are merged into one register after the OR gate. The design as a whole retains the same functionality before and after retiming, but now the critical path is much slower. Refer to the *Synplify AE for Actel User's Guide* for information about using retiming options.

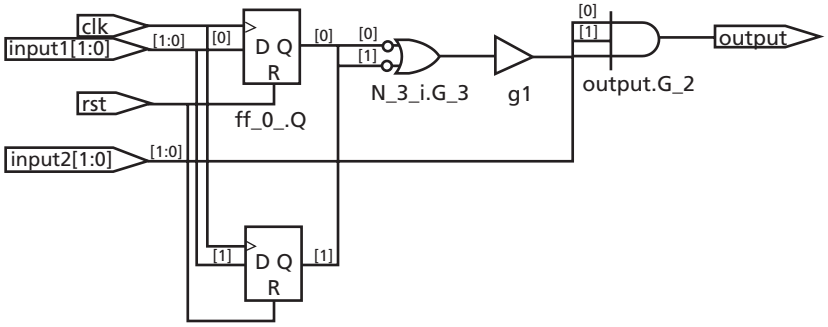


Figure 22 • **No Retiming**

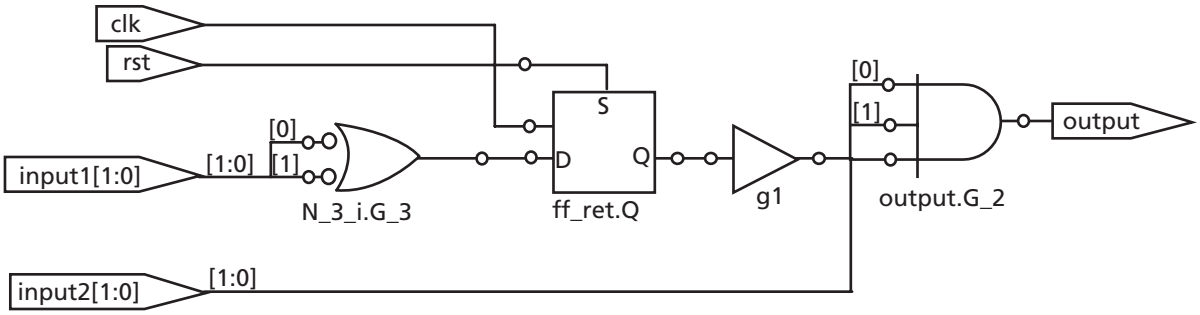


Figure 23 • **Using Retiming**

Multipoint Synthesis

Multipoint synthesis allows you to break down the design into smaller blocks called compile points (available in the Synplify Pro tool only). These smaller blocks are treated as one block for incremental mapping. During resynthesizing, the blocks that have already been synthesized are not resynthesized. This is important if you do not want synthesis to change the result for some blocks during incremental design flow. Refer to the *Synplify AE for Actel User's Guide* for a step-by-step explanation of the Multipoint synthesis flow.

Optimization Tips Using Synplify

The following sections describe various optimization tips using Synplify. The first section covers general optimization tips that apply to both speed and area optimization at the same time. Then speed and area optimization will be covered separately.

General Optimization Tips

This section contains general optimization tips that are not directly area or timing-related.

- For best results during synthesis with a timing-critical design, explicitly define each clock frequency with a constraint, instead of using a global clock frequency. This will relax less non-critical clocks and allow synthesis to focus on more critical clock domains.
- Some heavily congested designs benefit from hierarchical optimization techniques. An easy way to check the congestion is to check the number of Opens/Short nets after the first pass during routing. If you see many open nets during routing in Designer and timing is not important, you can keep the hierarchy for those blocks. During synthesis, set the syn-hier attribute to hard for those blocks.
- The fanout setting is a very important criterion for design optimization in ProASIC^PLUS. The default fanout setting for ProASIC^PLUS is 12. To honor this limit, Synplify either replicates components or adds buffers. It reduces fanout on input ports through buffering and reduces fanout on nets driven by registers or combinatorial logic through replication. For internal signals, the software first tries replication, replicating the net driver and splitting the net into segments. This increases the number of register bits in the design. When replication is not possible, the software buffers the signals. You can control whether high fanout nets are buffered or replicated, using the techniques described here.
  - To use buffering instead of replication, set syn_replicate to a value of 0 globally, or on modules or registers. Replication prevents a signal from taking advantage of the local clock network. If you turn off replication, you have the ability to promote the signal to a local clock during a later stage of the design flow. This is important if you want to promote any signal to a local clock. By default, Designer removes the buffer, so you can promote that signal to a local clock easily. Replicating prevents you from doing that.
  - A simple way to turn off buffering and replication entirely is by setting syn_maxfan to a very high number, such as 1000. Then use the gcf constraint in Designer to do global management, and this may give you better results.

Area Optimization Tips

This section contains information on setting options in Synplify for area optimizing.

- Increase the fanout settings. A higher limit means less replicated logic and a consequently smaller area.
- Check the Resource Sharing option when you set implementation options. With this option checked, the software shares resources like adders, multipliers, and counters wherever possible, and minimizes area.
- For designs with large state machines, use the gray or sequential encoding styles. They typically use the least area.

Timing Optimization Tips

This section contains general optimization tips that are timing-related.

- Apply Timing constraints and use realistic design constraints. The timing constraint should be within 10–15% of the real goal. For example, if the goal is 50 MHz, you can apply 55 MHz during synthesis.
- Use individual clock constraints. Using global clock constraints will not give your desired result. Besides using individual clock constraints, apply false path and multicycle path constraints.
- Use one hot encoding style for state machines as it will have the fastest implementation in most cases. However, if you have a large decoder at the output, use another coding style.
- If you saw warnings about a feedback MUX being created for signals when you compiled your source code, make sure to assign set/resets for the signals. This improves performance by eliminating the extra MUX delay on the input of the register.
- As explained before, you can achieve better timing by disabling Resource Sharing if the critical path goes through arithmetic components.

Finally, check the Synplify log report, which includes usage report, timing report, net buffering report, etc., before moving to the place-and-route tool.

PALACE Physical Synthesis

PALACE® is a physical synthesis tool that optimizes the gate count and performance with respect to device resources and design constraints. The default option normally provides optimal performance. Here are a few key ProASIC^PLUS design optimization techniques using PALACE.

- For designs with small area utilization, use the Physical Effort option to get complete placement information.
- If there are critical clocks, set tight timing constraints on these clocks, and loosen the constraints on some of the non-critical clocks. Similar to Synplify, tighten the constraints on the critical clocks by 10% more than the desired results for iterative optimization.
- Use the "logic synthesis effort" level of 3 for timing-critical designs. During the internal testing, the "logic synthesis effort" level of 3 provides better timing results than using the level of 4.
- For area optimization, use a "logic synthesis effort" level of 1.

To find the best options for your design, you can run the *aa_prun.tcl* file, available in the scripts folder of the PALACE installation disk. Review the results, which will show you the outcome for combinations of various settings.

TDPR and Floorplanning in Designer

Designer is the Actel place-and-route tool. By default, Designer attempts to remove all logics from the netlist that has no effect on the functional behavior. Designer takes advantage of the inverted inputs of logic tiles by removing inverters. This reduces the overall size and also produces a faster place-and-route time. These actions are done during compile. No area optimization will happen after that, so there are not many opportunities for area optimization in Designer. For any design, one of the key points is to have no hold violations. This applies to all synchronous designs, even where performance is not critical. If you have a clock signal with a high fanout, assign the clock signal to a global network or clock spines. This will reduce the clock skew and remove any hold violations.

Timing Optimization with TDPR

For timing optimization, an important step is to apply complete timing constraints during place-and-route. Actel provides a static timing analysis tool called SmartTime for the ProASIC^PLUS family. SmartTime supports a range of timing constraints to provide useful analysis. The SmartTime static timing analysis tool works very tightly with the Actel place-and-route tool. You can also import SDC timing constraints generated by the synthesis tool, as explained in the "Synthesis with Synplify" section on page 16, or you can create your own SDC constraints.

Here are some tips for achieving timing optimization using timing constraints.

- Apply all timing constraints, including false path and multicycle path. Look into the HDL code and determine the false path and multicycle paths for your design.
- Run TDPR and check for violations.
- If there are fewer than 50 timing violations remaining at this point, add specific user path sets and apply the max delay constraint using the SmartTime GUI. Then run TDPR.
- When there are only a few timing violations, perform incremental placement in the layout stage. In this case, timing constraints can be met without affecting the performance of the rest of the design.

Timing Optimization with Physical Constraints and Floorplanning

Floorplanning guides the place-and-route process to ensure the success of your placement into user-defined areas of the device. It is very important to understand the ProASIC^PLUS architecture before you try to floorplan a design.

One of the important architectural details is the ProASIC^PLUS Global Architecture. During compile, Designer will automatically place high fanout nets and some essential nets on global clock networks. However, this assignment might not reflect the best global assignment and may reduce overall performance. To prevent this, manually force Designer to place certain nets on the global networks or clock spines. In most of the design, you do not need to use all four global clock networks. Instead, use the set_auto_global and dont_fix_global commands to free up your global clock networks. This is necessary to

accommodate specific nets you want to promote to clock spine assignments. Also, if the global signal does not need to go to all blocks or has less fanout, you can use region constraints and free up some spines of that global network. Refer to the *Optimal Usage of Global Network Spines in ProASIC^PLUS Devices* application note for details.

Timing optimization can be also achieved by floorplanning. Before creating a floorplan, it is very important to identify the critical paths in your design. You should assign high fanout or critical path nets to a region only after you have used up your global routing and clock spine networks. To get more information on the various floorplanning techniques, refer to the *Floorplanning ProASIC/ProASIC^PLUS Devices for Increased Performance* application note. For some designs, no amount of floorplanning can improve their performance compared to push-button place-and-route. For timing-critical designs, apply region constraints as the last step for achieving timing constraints.

MultiView Navigator Tips

This section includes an additional tip for design optimization using floorplanning. This tip is not covered in the floorplanning application notes. In some cases you may need to move the logic away from the source to get better timing. You can use this technique if you have only a few violations and as a last resource for timing optimization.

For example, consider the design where an FF1 is driving 8 logic blocks, as shown in Figure 24 on page 22. Place the logic blocks adjacently, as in Figure 24 on page 22, to get the best timing for all blocks. For this design, suppose the critical path is going through MUX5. If you move it from the adjacent location of FF1, this will reduce the delay on the net connected between the FF1 and MUX5. See Figure 25 on page 23 for the timing delay.

This is due to the routing structure in ProASIC^PLUS device. In ProASIC^PLUS, the efficient long-line resources provide routing for longer distances and higher fanout connections. These resources vary in length (spanning 1, 2, or 4 tiles). You can take advantage of this routing instead of using ultra-fast connection for a congested design. If you move the MUX5 by 1 tile, you can use the efficient long-line resources instead of the direct connect, and distribute the load to the fuse connecting the long-line.

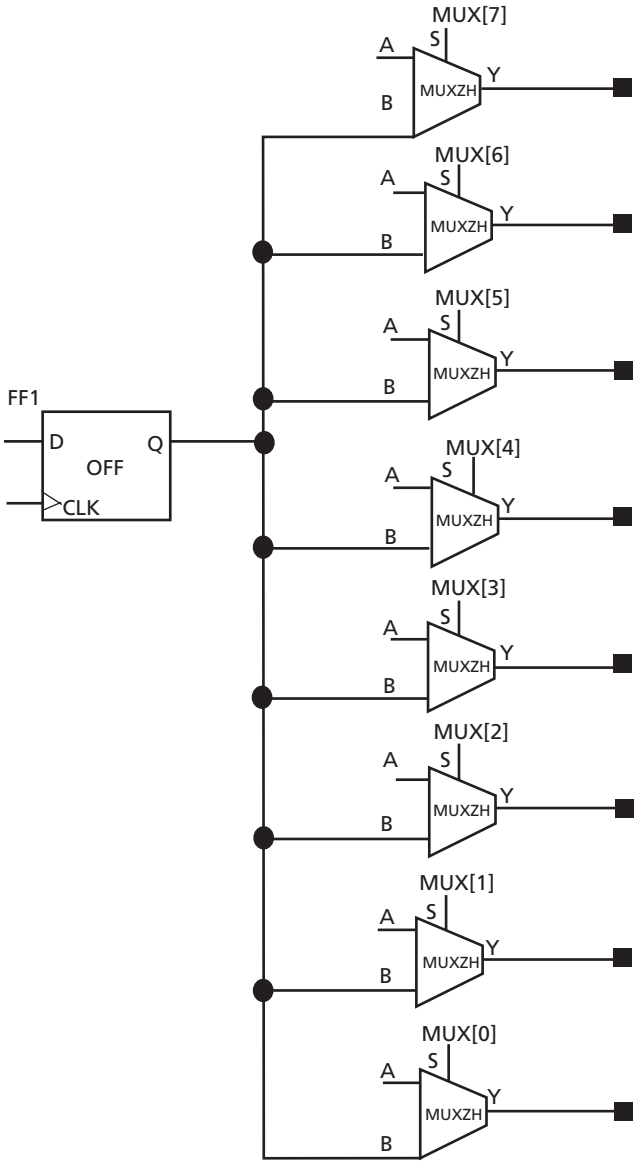


Figure 24 • **Design Schematic**

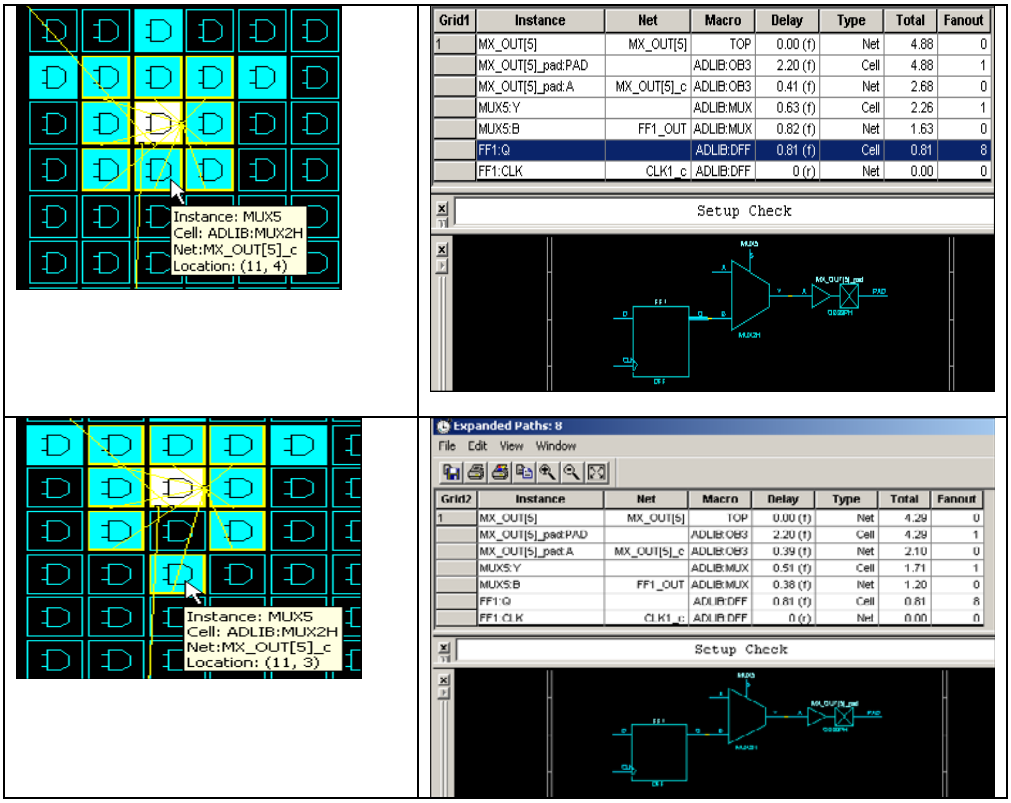


Figure 25 • **Timing for Placing Macro in Different Location**

General Recommendations

Various techniques for area and timing optimization were presented in the previous sections. You may need to experiment to find out the best techniques for your design. It is not feasible to make only one recommendation and point out the technique or techniques that will be best suited for a specific design. In general, you should focus mainly on HDL coding and synthesis. Most HDL-based designs use either a top-down or bottom-up (block-based) design methodology. Depending on size, type, design complexity, design environment, or existing flows, one may be more suitable than another.

For ProASIC^PLUS, here is the design flow Actel recommends.

- Create your RTL source code.
- Run synthesis with default options.
- Run Designer with timing constraints.
- Check the timing. If there are no violations, your design is complete.
- If you see violations, check the number of paths having violations and average delay per gate.
  - If you have only a few violations, try the tips used in the MultiView Navigator and TDPR sections.
  - If you see an average delay of more than 2 ns for each number of logic levels, you can use the timing and floorplanning techniques.
  - If you see the average delay is less than 2 ns per logic level, then you should go to HDL coding and work on it.

Conclusion

This application note describes various ways to get timing and area closure for ProASIC^PLUS designs. Traditional approaches such as working with the constraints or floorplanning add value to the timing closure process, but smarter synthesis options, physical synthesis, and RTL coding help to make it faster and consistently reusable for design iterations. The overriding goal is to help you achieve the design goal more rapidly. After studying the HDL coding guide and the various techniques in this application note, you should be equipped to achieve your design goal quickly.

Related Documents

Application Notes

ProASIC^PLUS Flash Family FPGAs
www.actel.com/documents/ProASICPlus_DS.pdf
ProASIC^PLUS Timing Closure in Libero IDE v5.2
www.actel.com/documents/APA_TimingClosure_AN.pdf
Optimal Usage of Global Network Spines in ProASIC^PLUS Devices
www.actel.com/documents/APA_Spines_AN.pdf
Floorplanning ProASIC/ProASIC^PLUS Devices for Increased Performance
www.actel.com/documents/Flash_Floorplanning_AN.pdf

User's Guides

Synplify AE for Actel User's Guide
http://www.actel.com/documents/synplify_ug.pdf

www.actel.com

Actel Corporation

2061 Stierlin Court
Mountain View, CA
94043-4655 USA
Phone 650.318.4200
Fax 650.318.4600

Actel Europe Ltd.

Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom
Phone +44 (0) 1276 401 450
Fax +44 (0) 1276 401 490

Actel Japan
www.jp.actel.com

EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan
Phone +81.03.3445.7671
Fax +81.03.3445.7668

Actel Hong Kong
www.actel.com.cn

Suite 2114, Two Pacific Place
88 Queensway, Admiralty
Hong Kong
Phone +852 2185 6460
Fax +852 2185 6488

51900111-0/8.05