WP309 (v1.1) January 21, 2010

Targeting and Retargeting Guide for Spartan-6 FPGAs

By: Brian Philofsky

When targeting or retargeting code from a prior design, some considerations should be made to achieve a quicker and more optimal design when selecting a Spartan®-6 FPGA. This white paper identifies and details the appropriate targeting guidelines and other considerations needed to achieve an improved result for these devices.

Spartan-6 Device Selection

Due to differences in the base architecture of the Spartan-6 device from previous FPGA generations, selection of the right Spartan-6 device for a given design is not always straightforward. Spartan-3 FPGA part numbers reflect system gates while Spartan-6 FPGA part numbers reflect approximate logic cell count divided by 1,000. Also, Spartan-6 devices use different ratios of block RAMs, multiplier/DSP blocks, DCMs, and pins.

Compare the feature summary tables in DS160, *Spartan-6 Family Overview*, DS099, *Spartan-3 FPGA Family Data Sheet*, DS312, *Spartan-3E FPGA Family Data Sheet*, and DS706, *Extended Spartan-3A Family Overview*, for details on the resources and architectures of Spartan-3 and Spartan-6 devices.

Beyond LUTs and flip-flops, each new generation of Xilinx FPGAs tends to include larger block functionality over previous generations, and the Spartan-6 FPGA is no exception. The Spartan-6 architecture includes an integrated memory controller, integrated PCI Express® logic, and gigabit transceivers. In Spartan-6 devices, the 18 Kb block RAMs can be split into two 9 Kb RAMs, which allow for greater utilization in designs not requiring deep memory structures. Compared with devices earlier than Spartan-3A DSP devices, the DSP48A1 slice adds additional functionality over the MULT18X18 primitive as well. The use of such functions compared to prior families adds capabilities not reflected in the logic cell count of the device. This can reduce the logic count as well as reduce the amount of block RAMs, multipliers, I/Os, and other resources. All of these subjects should be considered when determining device selection.

Comparing Logic Utilization in Spartan-6 Devices

Some logic functions benefit from the 6-input LUT more than others. For example, a 32-bit XOR gate consumes seven 6-input LUTs (with some logic to spare), where a 32-bit XOR gate consumes 11 4-input LUTs (with no logic to spare). The use of 6-input LUTs with a 32-bit XOR gate represents a 45% reduction in the required number of LUTs.

A single 2-to-1 MUX maps into a single 4-input LUT in the same manner as a single 6-input LUT. Thus, there is less advantage to using the 6-input LUT with a 2-to-1 MUX (although other logic could potentially be placed into that same LUT).

A 2-input adder requires one 4-input LUT or one 6-input LUT per bit of addition. Thus, certain types of designs that use small MUXes, adders, and other logic functions might not realize improved LUT utilization or performance from a Spartan-6 FPGA.

DSP designs generally use these logic functions and extensive pipelining as the basis for many of their operations. Therefore, DSP designs do not realize as much benefit transitioning to the Spartan-6 architecture outside of the availability of the DSP48A1 slice. Alternatively, some embedded processor designs realize more benefits from Spartan-6 devices because they:

- Use fewer registers (less pipelining).
- Have higher fan-in functions.
- Use fewer functions that do not realize the mapping improvements for the 6-input LUT.

The schematic representation in Figure 1 shows a 32-bit XOR function mapped into Spartan-6 FPGA 6-input LUTs.

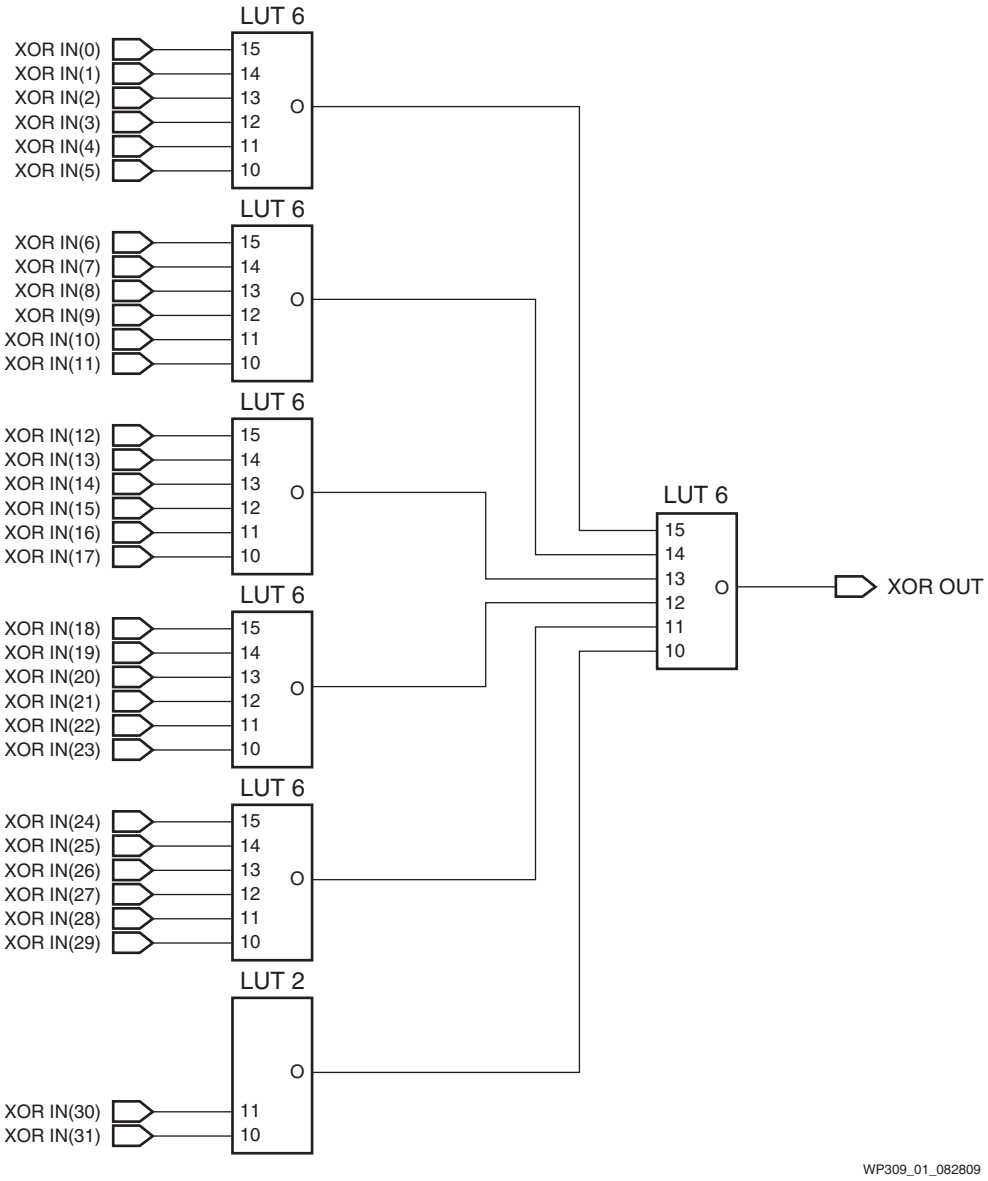

Figure 1: **32-Bit XOR Mapped into 6-Input LUTs of a Spartan-6 FPGA**

The schematic representation in Figure 2 shows a 32-bit XOR function mapped into Spartan-3 FPGA 4-input LUTs.

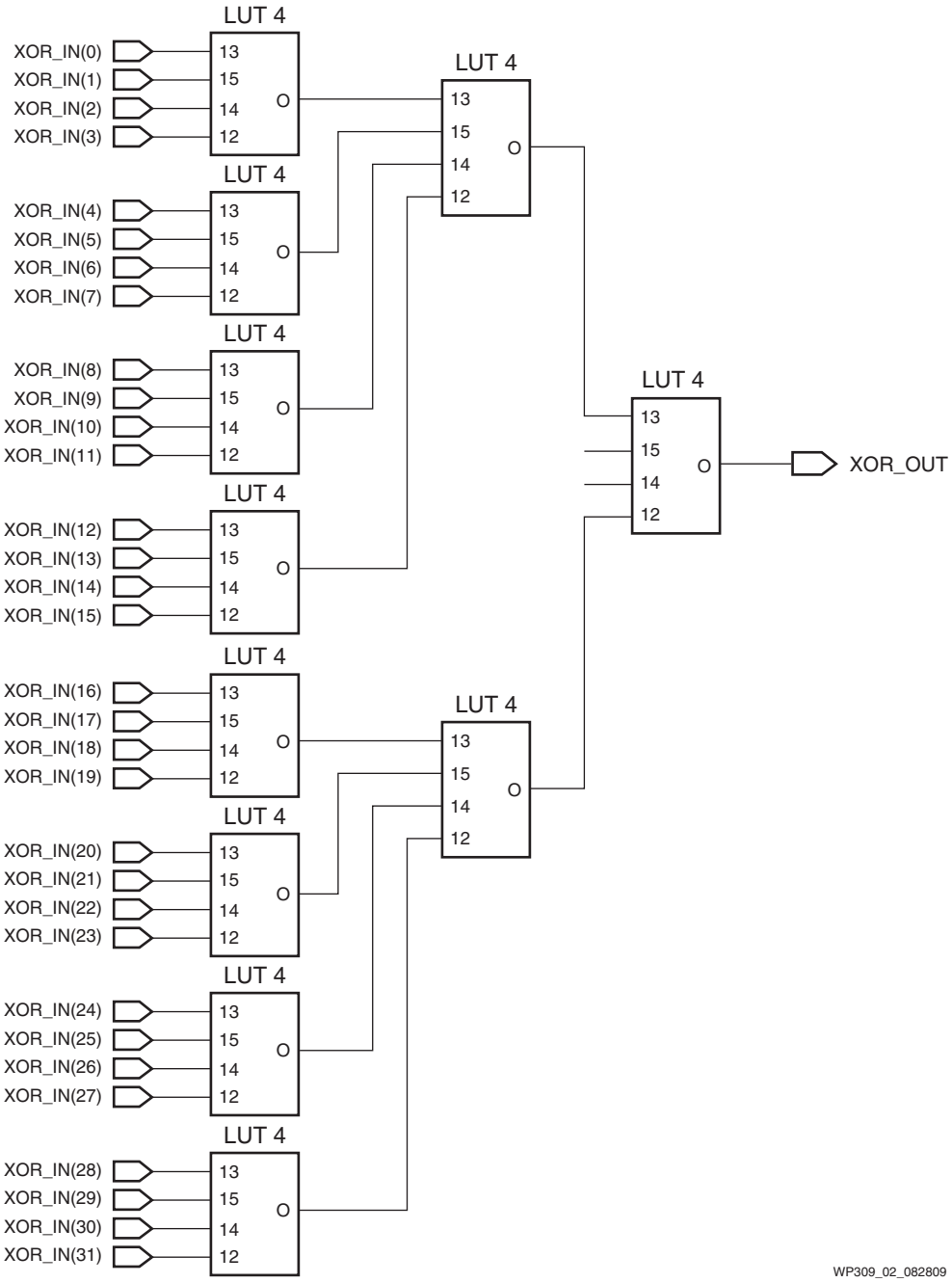

Figure 2: **32-Bit XOR Mapped into 4-Input LUTs of a Spartan-3 FPGA**

Use of LUTs as Route-Thrus

When comparing Spartan-6 FPGA LUT utilization to prior architectures, the use of LUTs as route-thrus must be considered. LUT route-thrus in the Map report are created when access is needed to internal slice logic or registers when no other input path is available into the slice, most commonly when the bypass inputs (AX, BX, CX, or DX) are not available. A LUT route-thru uses a single input to the LUT to obtain access into the slice. A few situations can cause this:

1. A flip-flop, RAM, or other non-LUT source drives a flip-flop (where bypass lines are occupied).
2. A flip-flop, RAM, or other non-LUT source drives the MUXF7/MUXF8 data inputs.
3. A flip-flop, RAM, or other non-LUT source drives a select line of CARRY4 (select line of MUXCY and/or DI of XORCY).

Situations 2 and 3 are not unique to Spartan-6 FPGAs but are applicable to Spartan-3 and Virtex®-5 FPGAs. Situation 1 is the most common.

Spartan-6 devices have eight registers within a slice, which is beneficial for both performance and area. For many designs, fewer registers require more slices due to the higher logic content of the 6-input LUT vs. the 4-input LUT in conjunction with the improved use of the dual outputs of the 6-input LUT structure.

Having eight registers often allows fewer slices to be consumed for registers driven by non-LUT sources (such as block RAM, another flip-flop, or a DSP slice). The Xilinx software tools are tuned to place these slice registers to yield the best characteristics in terms of performance, area, and power. However, when more than four registers are placed into a single slice, a LUT route-thru is necessary and thus reported in the Map file.

In a given design, when comparing LUT resources between Spartan-6 and prior device families, more LUTs can be reported as being used in Spartan-6 devices, but in fact fewer slices are required because of the route-thrus. This can be misleading because it is common to consider LUT usage and not slice usage for utilization comparisons. For designers concerned with how much logic is left, the route-thru consumes one input (generally not the A6 input, but one of the lower inputs to get access to the O5 output) and one output of the LUT, leaving four inputs and one output for any given function. Thus, even when a route-thru is needed, in the worst-case situation, a 4-input LUT remains unused.

In a typical situation, the registers can be distributed over more slices, if necessary, and not consume any route-thrus. Thus, designers get the entire 6-input LUT for logic but might need to consume more slices or possibly see performance penalties for the spreading of these registers. Route-thrus are reported as consuming an entire LUT when in fact they only consume a partial LUT, or in a different placement might not consume a LUT at all. Here is a portion of the resource reporting in the Map (`.mrp`) file that reports route-thrus and their purpose:

```
Number used exclusively as route-thrus:      1,483
      Number with same-slice register load:  1,450
      Number with same-slice carry load:        26
      Number with other load:                    7
```

Figure 3 shows an FPGA Editor view of the use of a route-thru for gaining access to all eight registers within a slice.

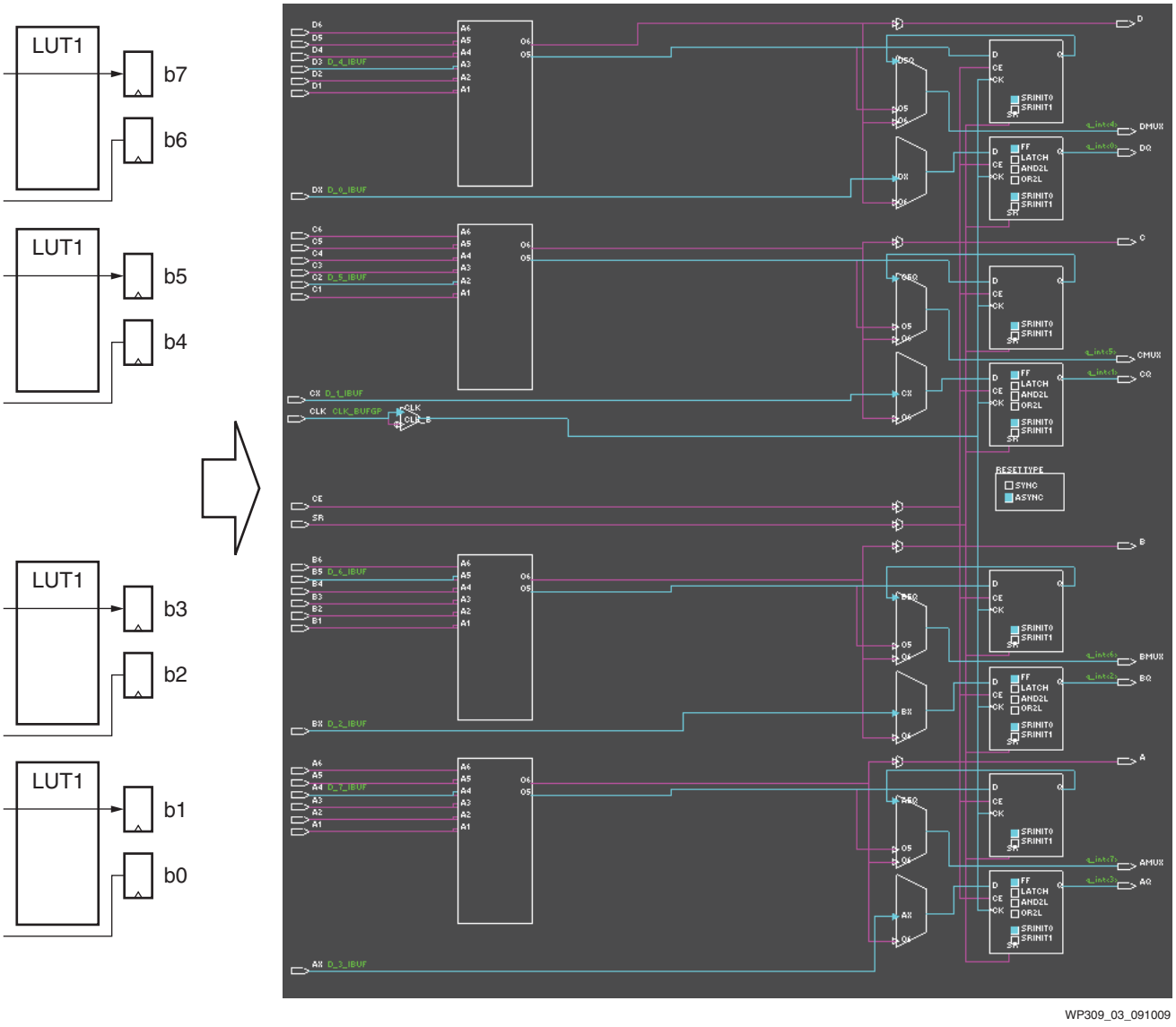

Figure 3: **FPGA Editor Representation of a Route-Thru**

Well-Crafted Designs for Older Architectures

Designs that are tuned to run at top performance in a Spartan-3 device see less improvement in performance or reduction in LUT utilization with a Spartan-6 device than designs that run at more nominal speeds. This is because fast designs that are well-optimized to the 4-input LUT structure, with low fan-in logic and few logic levels between synchronous objects, are more likely to have one-to-one mapping from the 4-input LUT to a 6-input LUT. Thus, there is little potential for LUT or logic reduction with the Spartan-6 device logic structure and little improvement in performance.

Often, retargeted designs that benefit most from the larger LUT structure of Spartan-6 devices are those designs that previously ran slowly in Spartan-3 devices, with many logic levels and larger LUT-to-register ratios. In these designs with large fan-in logic cones, there is greater potential for the 6-input LUT to considerably reduce the number of logic levels as well as the number of LUTs necessary to build a logic

function. Such designs achieve a greater LUT reduction and performance boost when retargeting to the Spartan-6 architecture.

When updating, changing, or building new circuitry for Spartan-6 devices in an attempt to achieve improved speed, area, and power, the user must understand the underlying architecture and note the differences in design choices the 6-input LUT and other architectural features can provide.

Where designers could consider four inputs per logic level in earlier architectures, they can now accept wider fan-in logic and achieve the same amount of logic levels. Carry-chain boundaries considered at 2-bit intervals previously should now be considered with 4-bit intervals. As with prior generations, considering the underlying architecture and how the code generated maps into it almost always yields improved and more predictable results. However, the underlying differences in the Spartan-6 architecture compared with older architectures need to be taken into account when constructing the HDL code.

Use of Old Soft IP, EDIF, or NGC Netlists

It is highly recommended to regenerate or resynthesize any old soft IP or black box netlists in the design prior to implementation in the Spartan-6 device. Most netlists targeting earlier Spartan architectures can be implemented without error when targeting Spartan-6 devices.

However, in almost every case, netlists retargeted to the Spartan-6 architecture can result in slower and larger implementations, that have additional logic levels and utilize more resources.

This trade-off is due to the differences in the underlying FPGA logic architecture (such as 4-input LUTs vs. 6-input LUTs), as well as the timing and optimizations for the different architectures. Retargeting of these black box instances ensures that they are optimized for the Spartan-6 architecture.

When resynthesis is not possible, an alternative is to use the global optimization switch (**-global_opt**) during the Map phase. The global optimization switch has three arguments: speed, area, and power. These arguments indicate the primary goal for the optimization. Although global optimization is generally not as effective as direct resynthesis of the black box instance, it allows the tools to resynthesize and restructure the netlist, which can result in a better implementation.

The main disadvantages of using global optimization are increased run time and possibly further obscurity (renaming and restructuring) to the internal logic paths, which can make a design more difficult to debug. While the **-global_opt** switch might allow the use of some soft IP in Spartan-6 devices, other soft IP must be retargeted due to the use of specific circuitry when targeting Spartan-6 devices. Such IP includes MIG, MicroBlaze™ processors, PCI™ IP, and any with gigabit transceivers. Use of this IP requires retargeting to Spartan-6 devices for proper functionality in this architecture.

Use of Physical Constraints

Any LOCs, RLOCs, BEL constraints, or other physical constraints, embedded in the code, netlist, or UCF file of the existing design should be removed before retargeting to a Spartan-6 FPGA. An optimal placement for an older architecture is likely not optimal in a Spartan-6 FPGA due to differences in the slice structure, the CLB, RAM (LUT RAM or block RAM), logic, I/O layout, and timing. In some cases, errors can occur due to layout and coordinate differences. However, even if no errors occur, timing, density, and power can be suboptimal unless the physical constraints are removed or updated for the new architecture.

Software Algorithms

Current software algorithms for Spartan-6 FPGAs are designed to deliver a balance between device area (and thus cost) and performance. Options in the ISE® software allow designers to improve device area at the cost of performance or improve performance at the cost of device area. There are also options to reduce power that can often result in trade-offs in performance, area, and/or software run time. Options in the software can be specified to achieve design goals when the default balanced approach does not.

First, synthesis timing constraints should be specified that relate realistic timing objectives. The synthesis software can apply area-saving algorithms where performance objectives can still be met in areas with excess timing slack. Timing optimization algorithms can be applied in areas with tight timing slack. Without timing constraints, the synthesis tools must optimize all parts of the design for timing, often at the expense of area.

Second, the LUT and slice packing behaviors can be changed within the Map portion of the ISE software. The ISE software contains a switch within synthesis (XST) and Map called **`-lc`** that employs a LUT compression algorithm to reduce the number of LUTs required in a design. The **-lc** switch accepts three arguments: **off, auto,** and **area**.

- The **-lc auto** setting causes the software to attempt to combine LUTs with little impact on timing.
- The **-lc area** setting causes LUT packing with a greater impact on timing. This algorithm can be used to achieve better LUT utilization for designs that have margin in performance or power. However, this setting is not recommended for designs that have difficulty in meeting performance or power budgets.
- The **–lc off** setting is the best option for performance in general. However, this setting often results in an increased amount of LUTs being used. This is the default setting for Spartan-6 FPGAs.

Another method to reduce the required number of LUTs is slice compression. Compression is done by setting the **-c 1** switch in Map, which invokes both LUT and slice compression, resulting in fewer required slices. However, this method often gives up greater performance and power over using the **-lc area** switch.

Use of Control Signals

The use of control signals (signals that control synchronous elements such as clock, set, reset, and clock enable) can impact device utilization. This is true in almost any FPGA technology. However, the difference in the topology of the Spartan-6 device has changed some considerations in selecting and using control signals. These considerations are necessary to achieve the best device utilization.

Use of Both a Set and a Reset on a Register or Latch

To reduce cost of the overall architecture, slices in Spartan-6 FPGAs do not have a REV pin. As a result, flip-flops no longer implement both a set signal and a reset signal. In addition, a register with a set or reset signal can only have an initialization value of the same polarity. For example, a flip-flop with an asynchronous reset can only have an initialization value of 0. This changes the implementation results, which can change the designer's strategy for register sets, resets, and initialization.

Using Latches for Implementation of Registers

Registers that contain both asynchronous reset and asynchronous set signals and/or contain an asynchronous reset or set signal with an initialization value of the opposite polarity can be implemented using a few components and a latch.

This configuration can be described in RTL or be instantiated with FDCE, FDPE, or FDCPE in HDL, EDIF, or NGC formats. When the software encounters these configurations, it issues a warning message describing the problem and lists the corresponding registers:

```
WARNING:Xst:3002 - This design contains one or more
registers/latches that are directly incompatible with the Spartan-6
architecture. The two primary causes of this is either a register or
latch described with both an asynchronous set and asynchronous
reset, or a register or latch described with an asynchronous set or
reset which however has an initialization value of the opposite
polarity (i.e. asynchronous reset with an initialization value of
1). While this circuit can be built, it creates a sub-optimal
implementation in terms of area, power and performance. For a more
optimal implementation Xilinx highly recommends one of the
following:

    1) Remove either the set or reset from all registers and latches
       if not needed for required functionality
    2) Modify the code in order to produce a synchronous set
       and/or reset (both is preferred)
    3) Ensure all registers have the same initialization value as
       the described asynchronous set or reset polarity
    4) Use the -async_to_sync option to transform the asynchronous
       set/reset to synchronous operation
      (timing simulation highly recommended when using this option)

Please refer to http://www.xilinx.com search string "Spartan6
asynchronous set/reset" for more details.

List of register instances with asynchronous set or reset and
opposite initialization value:
    q_int in unit <r_ff_init1>
```

Register Initialization

Many engineers use the inherent initialization of registers and latches in the FPGA via the global set/reset (GSR) signal by implicitly specifying initialization of an inferred register, thus creating a more robust and sometimes smaller circuit.

In this code example, the reg register is initialized with the value of 1:

```
signal reg: std_logic := '1';
...
process (clk, rst)
begin
  if (rst='1') then
    reg <= '0';
  elsif (clk'event and clk='1') then
    reg <= val;
  end if;
end process;
```

With the restrictions of the initialization polarity in Spartan-6 devices, initialization must be more closely monitored to ensure it does not negatively impact area, power, and performance.

It is still recommended to initialize registers. However, it is suggested to always match this initialization value with the described asynchronous/synchronous set or reset unless needed for design functionality. When a different initialization is needed from the set/reset value, then it is highly recommended to use synchronous sets/resets whenever possible.

When an initial value of a signal is not explicitly specified in VHDL code, it can still exist and depend on signal type. Several design cases are presented as follows.

Type: integer

The default value of the integer type is equal to the left bound value of the integer type definition (similar to the std_logic type). However, the final result differs from the std_logic type, as shown in this example, where the reg signal has an integer type:

```
signal reg: integer range 0 to 7;
...
process (clk, rst)
begin
  if (rst='1') then
    reg <= 7;
  elsif (clk'event and clk='1') then
    reg <= reg + 1;
  end if;
end process;
```

In this example, the default value of the reg signal is equal to 0. Because the reg register is reset to 7 and initialized to 0, the synthesis tool must use additional FPGA resources for its implementation.

If an initialized value of 0 is not necessary for functionality, the reg signal should be manually initialized to 7 to match the reset value specified in the associated process. If an initialized value of 0 is needed, then the asynchronous rst signal should be described synchronously to avoid using latches for its implementation.

Type: enumerated

Another case with state machines using the enumerated type requires the designer's attention. The default value of the enumerated type is equal to the left bound value of the enumerated type definition (similar to std_logic and integer types). In this example, the default value of the next_state state register is not explicitly specified and therefore equal to s1:

```
Type my_statetype is (s1, s2);
signal next_state: my_statetype;
...
process (clk, rst)
begin
  if (rst='1') then
    next_state <= s1;
  elsif (clk'event and clk='1') then
    if (PayloadTaken = '1') then
      next_state <= s1;
    else
      next_state <= s2;
    end if;
  end if;
end process;
```

Because one or more registers results in a different initialization value from the asynchronous reset value, depending on state mapping, the modified register circuit containing a latch must be used.

In addition to the extra logic compared with older FPGA families, extra timing paths are also created. To ensure these timing paths are properly analyzed, the software inserts an additional constraint on any register covered by a PERIOD constraint containing this expansion in order to ensure that the synchronous release of the asynchronous set or reset does not create a timing hazard. The inserted constraint can look like the following:

```
================================================================================
Timing constraint: TS_TO_q_int_0_LDC = MAXDELAY TO TIMEGRP "TO_q_int_0_LDC" TS_CLK
DATAPATHONLY;
 2 paths analyzed, 2 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors)
 Maximum delay is   2.534ns.
--------------------------------------------------------------------------------
```

If this time constraint passes, the asynchronous reset is timed to allow the release within the designated clock period. If this path fails, the deassertion of the reset can fail timing.

In general, it is best practice to avoid using unnecessary sets or resets in the design and to initialize any inferred registers in the design. It is a good idea to describe synchronous sets and resets whenever possible, and to use the same initialization value as the described set or reset to avoid any unnecessary logic inference.

Limit Use of Active-Low Control Signals

It is not recommended to use inferred or instantiated components with active-Low control signals due to the combination of:

- The Spartan-6 device's coarser granular slice composition.
- The absence of a programmable inversion element on the slice control signals in the Spartan-6 device.
- Hierarchical design methods that do not allow optimization across hierarchical boundaries.

In certain situations, device utilization can decrease due to the use of a LUT as an inverter and the additional restrictions of register packing sometimes caused by active-Low control signals.

Timing can also be affected by the use of active-Low control signals. Active-High control signals should be used wherever possible in the HDL code or instantiated components. When a control signal's polarity cannot be controlled within the design (such as when it is driven by an external, non-programmable source), the signal in the top-level hierarchy of the code should be inverted, and active-High control signals driven by the inverter to get the same polarity (functionality) should be described.

Limit Use of Low Fanout Control Signals

The number of unique control signals in the design should be limited to those necessary for design functionality and operation. Low fanout, unique control signals can result in underutilized slices in terms of registers, SRLs, and LUT RAM. These control signals can have negative impacts on placement and timing. A set, reset, or clock enable should not be implemented in the code unless it is required for the active functionality of the design.

Unnecessary Use of Sets or Resets

Unnecessary sets and resets in the code can prevent the inference of SRLs, RAMs (LUT RAMs or block RAMs), and other logic structures that are otherwise possible. To get the most efficiency out of the architecture, sets and resets should only be coded when they are necessary for the active functionality of the design.

Sets and resets should not be coded when they are not required. For example, a reset is not required when it is only used for initialization of the register because register initialization occurs automatically upon completion of configuration.

Another example where a reset is not required is when a circuit remains idle for long periods and a simple reset on the input registers eventually flushes out the data on the rest of the circuit.

A third example is in the case of inner registers when the reset is held for multiple clock cycles. In this case, the inner registers are flushed during reset, so the reset is not necessary. By reducing the use of unnecessary sets or resets, greater device utilization and improved performance can be achieved.

Sets for Multipliers or Adders/Subtractors in DSP48A1 Slice Registers

DSP48A1 slice registers contain only resets and not sets. For this reason, unless necessary, a set (value equals logic 1 upon an applied signal) should not be coded around multipliers, adders, counters, or other logic that can be implemented within a DSP48A1 slice.

Use of Synchronous Sets/Resets

If a set or reset is necessary for the proper operation of the circuit, a synchronous reset should always be coded. Synchronous sets/resets not only have improved timing characteristics and stability but can also result in smaller, better utilization within the FPGA.

Synchronous sets/resets can result in less logic (fewer LUTs), fewer restrictions on packing, and, often, faster circuits. If the designer does not want to recode existing asynchronous resets to synchronous resets, the asynchronous resets can be treated as synchronous resets by using the Asynchronous To Synchronous switch, if available, in the synthesis tool.

If Xilinx Synthesis Technology (XST) is the synthesis tool, the Asynchronous To Synchronous switch is available in the GUI, or the **–async_to_sync** switch can be used as a synthesis option. This option is not as effective as recoding to use a synchronous reset in terms of reducing resources and improving performance. However, it does allow for some register packing, which is not possible otherwise.

Use of Clock Enables

When high fan-out clock enables are used, they should not be manually split or replicated but coded as a single clock enable. If replication becomes necessary for timing or other reasons, it should be controlled within the synthesis tool.

Analysis and Use of Control Signals and Control Sets

As explained in Limit Use of Low Fanout Control Signals, the use of control signals can hinder or facilitate device optimization. Ideally, control signals should be observed both individually and also as a collection or a control set.

A control set consists of the unique grouping of a clock, clock enable, set, reset, and in the case of LUT RAM, write-enable signals. Control set information is important because in any given slice, eight registers must share the same control set to be used. Similarly, all SRLs and LUT RAMs must share the same control signals within a SLICEM.

For designs with few control sets, there is generally no issue with device utilization because there is a lesser impact on packing and combining of elements into slices. However, designs with a high number of control sets can have a negative impact on utilization and performance because register, SRL and LUT RAM placement can be more limited due to incompatible control sets.

When synthesizing a design, the synthesis tool evaluates this situation. For low utilization control sets, the synthesis tool attempts a remapping to reduce the overall control set count. This is important to know for two reasons:

- Logic could be added to the datapath to account for remapping that might not have appeared in prior architectures.
- The use of asynchronous sets or resets severely limits the ability to remap. It can be more effective to change the sets and resets to synchronous.

If it is necessary to analyze the design for the use of unique control sets, the Synthesis report (`.syr` file if using XST, `.srr` file if using Synplicity) or the Map (`.mrp`) report can give details about each control set. When using XST synthesis, the Design Summary reports the number of unique control sets as follows:

```
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:  50479
   Number with an unused Flip Flop:   24773  out of  50479    49%
   Number with an unused LUT:         10703  out of  50479    21%
   Number of fully used LUT-FF pairs: 15003  out of  50479    29%
   Number of unique control sets:      1461
```

Additional details can also be seen in the Clock Information and Asynchronous Signals Information sections of the XST report.

If using Synplify, the synthesis report reveals a detailed breakdown of all control sets and the number of members to each control set. Similar to the XST report, the Map report also reports the number of unique control sets in the resource section of the report as well as the number of register sites lost due to control set incompatibilities. To view more detailed information in the Map report, the **-detail** switch should be enabled. This populates section 13 of the report with a complete breakdown of the control set information from the implemented design.

If it is suspected that a large number of unique, low fan-out control signals are causing low register utilization, steps can be taken to control the inference of control signals for either part of the design (such as a particular net or hierarchy) or globally in the design.

The first and most beneficial item to evaluate is whether all control signals are necessary for design implementation. Any unused or unnecessary set, reset, or clock enable described in the code should be removed. All described asynchronous sets and resets should be evaluated as to whether their descriptions can be changed to synchronous sets and resets. This gives flexibility to remap the set/reset function and/or the clock enable function to the logical datapath for paths with adequate timing slack.

The next step is to examine the settings for the tools. If XST synthesis is used, the Reduce Control Sets option must be set to Auto, as shown in Figure 4.

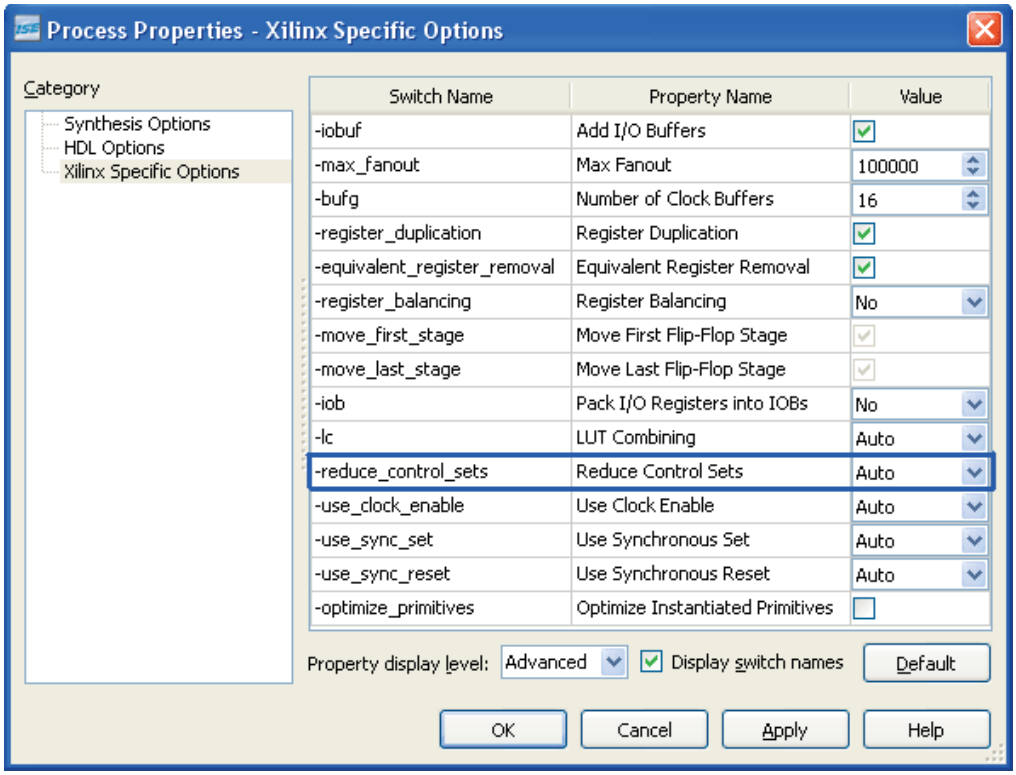

Figure 4: **XST Process Properties**

This setting enables a synthesis algorithm to automatically remap low fan-out control signals. If using Synplify, similar algorithms exist but are not user controlled. For further fine tuning of control set mapping, synthesis attributes can be placed on specific hierarchical instances or inferred registers to dictate mapping.

For instance, when using XST, the following Verilog code results in circuit shown in Figure 5.

```
reg reg_with_inferred_ce;

 always @(posedge CLK)
   if (CE)
      reg_with_inferred_ce <= D;
```

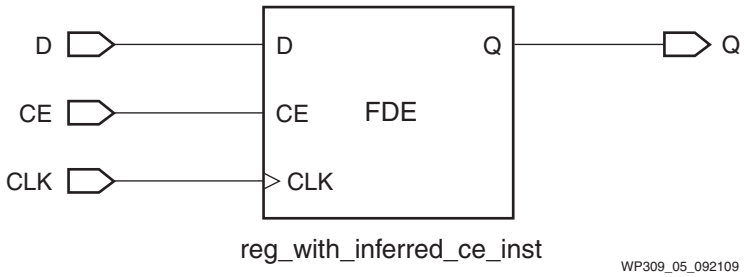

Figure 5: **Default Implementation for the Code Inferring a Register with a Clock Enable**

If CE is determined to be a low fan-out control signal that reduces the overall register density of the design, designers can apply an attribute to the declaration in Figure 5, such as in the following Verilog code, resulting in the circuit shown in Figure 6.

```
(* use_clock_enable = "no" *) reg reg_with_inferred_ce;

 always @(posedge CLK)
   if (CE)
      reg_with_inferred_ce <= D;
```

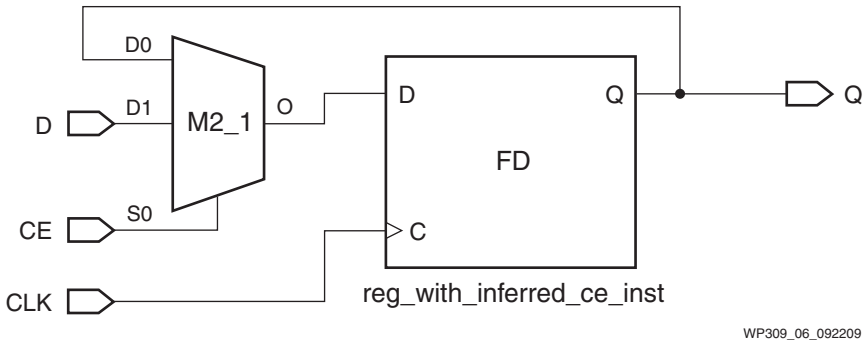

Figure 6: **Adding the use_clock_enable = "no" Synthesis Attribute**

The register in Figure 6 can now be packed with other registers with a common clock. The register also has the clock enable circuitry applied to the input path of the register. While it is not recommended to always manage control signal mapping in this way, this technique can prove useful in certain situations where it is desired to control the trade-offs of register density versus logic (LUT) use.

Use of DSP and Other Arithmetic Intensive Code

Many types of DSP designs are well suited for the Spartan-6 architecture. To obtain best use of the architecture, the underlying features and capabilities need to be understood so that design entry code can take advantage of these resources.

If the design originates from a Spartan-3A DSP FPGA using the DSP48A slice, usually little change is necessary. Well constructed filters using the DSP48A slice should directly target the DSP48A1 slice for Spartan-6 devices.

An important change is the addition of the MOUT port. If the DSP block is being used as a multiplier without post-add functionality, the MOUT port provides faster implementation that consumes less power.

For instantiated DSP48A slices not using the post-adder, it is not necessary to change the code. However, moving the output data from the P port to the M port can result in a faster and more power efficient implementation.

For designs using MULT18X18, MULT18X18S, or the MULT18X18SIO, these components retarget to a DSP48A1 slice. However, it is suggested to evaluate how the multiplier is used. If the pre-adder and/or post-adder can be used, it is likely that an improved implementation can be created by recoding the function to more efficiently use the capabilities of the DSP48A1 slice. Refer to UG389, *Spartan-6 FPGA DSP48A1 Slice User Guide*, for details on using this slice.

Another notable difference in the Spartan-6 architecture is the addition of the SLICEX. The SLICEX is similar to the SLICEL; however, the SLICEX has no carry logic. In most

designs, carry logic is an under-utilized feature, and thus its absence has little impact on performance. In adder intensive designs, the lack of carry logic could be a limitation.

If the target design is expected to contain a large number of adders and thus consume a large portion of carry logic resources, it is suggested to evaluate the design to make greater use of the DSP48A1 slice adders. For example, with FIR filters, the adder cascade can be used to build a systolic filter rather than using multiple successive add functions (adder trees).

If adder trees are necessary, the 6-input LUT architecture can efficiently create ternary addition (A + B + C = D) using the same amount of resource as a simple 2-input addition. This can help save and conserve carry logic resources, when needed. In most cases, there is no need to use these techniques. However, by knowing these capabilities, the proper trade-offs can be acknowledged.

Clocking

The Spartan-6 architecture has twice the global clocking over the Spartan-3 architecture (16 BUFGs compared with 8 BUFGs in the Spartan-3 architecture). In many cases, clocking availability is not an issue when migrating designs from a Spartan-3 architecture to a Spartan-6 architecture.

A designer might choose to use local clocking resources to capture input data and synchronize it to the FPGA system clock. In these cases, a change must be made to the clocking structure to enable a safe target to the Spartan-6 architecture. It is suggested to use the BUFIO2 clocking resource to capture and synchronize data. Consult UG382, *Spartan-6 FPGA Clocking Resources User Guide*, for details.

Depending on how the clocking resources are used, the Spartan-6 architecture can appear to have greater or fewer global clocking resources compared with Spartan-3A and Spartan-3E architectures. Spartan-3A and Spartan-3E architectures have eight BUFGs that can source any synchronous element in the array, similar to the Spartan-3 architecture. However, Spartan-3A and Spartan-3E architectures have eight additional buffers on the left and right sides, which can only source half of the device.

As previously mentioned, the Spartan-6 architecture has 16 clocks that can source the entire device. If more than 16 unique clocks are required, clocking changes might be required to either consolidate or use alternative clocking resources like the BUFIO2s or the BUFPLLs in place of the BUFGs used in Spartan-3 architecture.

BUFGMUXes can be used in the same manner as previously used. If an asynchronous transfer of clocking is desired, a capability in the Spartan-6 architecture allows asynchronous clock transfers in case of a stopped clock or when a faster switchover is needed.

To enable this functionality, the attribute, CLK_SEL_TYPE, should be added to the parameter passing (Verilog) or generic map (VHDL) of the instantiated BUFGMUX component, which enables this behavior. If this feature is used, runt clock pulses or glitches might occur on the clock lines, so design practices that can tolerate these are highly recommended.

Driving Non-Clock Loads with Global Buffers

As with prior architectures, it is highly recommended to only drive clock loads with a global buffer. The connectivity to non-clock sources is limited and can, under some conditions, result in unroutable situations. For this reason, the software issues the following error when a BUFG is driving non-clock loads:

```
ERROR:Place:1136 - This design contains a global buffer instance, <BUFG_inst>,
   driving the net, <clk_int>, that is driving the following (first 30)
   non-clock source pins.
   < PIN: D_AND_CLK1.A6; >
   This is not a recommended design practice in Spartan-6 due to limitations in
   the global routing that may cause excessive delay, skew or unroutable
   situations. It is recommended to only use a BUFG resource to drive clock
   loads. If you wish to override this recommendation, you may use the
   CLOCK_DEDICATED_ROUTE constraint (given below) in the .ucf file to demote
   this message to a WARNING and allow your design to continue.
   < PIN "BUFG_inst.O" CLOCK_DEDICATED_ROUTE = FALSE; >
```

Adding the CLOCK_DEDICATED_ROUTE constraint as a temporary override of this condition is not a guarantee that the design will implement reliably, but it can be used as a temporary measure to allow for design analysis. Whenever this error is encountered, it is suggested to modify the code so that a global buffer only drives clock sources, if possible.

Synthesis tools will attempt to infer correct clocking. However, clocking networks can be improperly constructed when using soft IP that appears to be black boxes. For this reason, it is always suggested to read in all implementation netlists into synthesis for them to be constructed properly. Otherwise, skew and other possible clocking issues might be encountered.

RAM Considerations

To maximize the use of block RAMs and LUTs in the Spartan-6 architecture, certain considerations must be understood when retargeting block RAM and LUT RAM by inference, instantiated primitive, or CORE Generator™ software. If the CORE Generator software is used for RAM generation, the IP should be regenerated for the Spartan-6 device, or the RAM should be recoded for proper synthesis inference.

Either method often gives good results for utilization and performance. However, it is recommended to infer memory where possible to improve understanding of the code, simulation, and future portability of the code.

Instantiating RAMs

The recommendations in this section are for cases in which RAM primitives are instantiated in the design or when it is not possible to regenerate CORE Generator software IP for Spartan-6 devices. These suggestions should also be implemented by code that infers RAM, especially when using synthesis attributes to guide which RAM resources are used (such as syn_ramstyle = blockram). The suggestions are divided by RAM depth, which generally is the most important factor in determining which RAM resource to use.

Depths Less Than 128 Bits

Due to the larger LUTs and deeper LUT RAMs in Spartan-6 FPGAs, the criteria for choosing between a block RAM and LUT RAM are different compared to those for previous FPGA generations. In general, a LUT RAM should be used for all memories that consist of 64 bits or less, unless there is a shortage of logic resources (LUTs) and/or SLICEMs for the target device.

Using LUT RAMs for memory depths of 64 bits or less, regardless of data width, is generally more efficient in terms of resources, performance, and power. For depths greater than 64 bits but less than or equal to 128 bits, the decision on the best resource to use depends on several factors:

1. Are extra block RAMs available? If not, LUT RAM should be used.
2. What are the latency requirements? If asynchronous read capability is needed, LUT RAMs must be used.
3. What is the data width? Widths greater than 16 bits should probably use block RAM, if available.
4. What are the necessary performance requirements? Registered LUT RAMs generally have shorter clock-to-out timing and fewer placement restrictions than block RAMs. If the design already contains instantiated LUT RAMs with depths greater than 16 bits, the deeper primitive (for example, RAM32X1S or RAM64X1S) should be used.

RAM16X1Ss, used in conjunction with MUXF5s or other logic, are not properly retargeted to automatically use the greater depth LUT. In such cases, the code should be modified to properly use the deeper primitives.

Depths Greater Than 128 Bits and Less Than or Equal to 256 Bits

Memory depths greater than 128 bits and less than or equal to 256 bits can be efficiently mapped into a block RAM or a LUT RAM. The decision here depends on the necessary width, RAM requirements, and the available RAM resources. In most cases, if block RAM is available and the width is greater than 8 bits, it is advantageous to use an 8 Kb block RAM. However, in some cases the block RAM can be underutilized. Synthesis algorithms exist so that some configurations of inferred RAMs can be placed into a single block RAM if common clock and addressing are used.

The special case of a simple dual-port configuration greater than 18 bits but less than or equal to 36 bits can efficiently map into a RAMB8. However, many configurations of memory in this range can result in underutilized block RAM. Conversely, if LUT RAM is selected, an entire SLICEM is used per bit, which might not be the most efficient use the slice resources. So these two factors must be considered for RAM depths in this range. In most cases, it is suggested to simply infer the RAM and allow the synthesis tool to decide the appropriate RAM components based on available resources, performance requirements, and other factors.

Depths Greater than 256 Bits

In most cases, described RAM for the Spartan-3 architecture properly retargets to Spartan-6 devices without modification regardless of how it is entered. Added features in block RAM of Spartan-6 FPGAs should be evaluated to determine if they could benefit the design. Some of these features include:

- Output Register. Spartan-3A DSP devices contain an output register for the block RAM. However, designs created prior to the Spartan-3A DSP architecture do not use this feature. Use of the output register can significantly improve performance (clock-to-out) of the block RAM, while also improving power and device utilization. If a design is ported into the Spartan-6 architecture from a prior architecture, the code should be re-examined to see if the output register can be incorporated into the design.
- Byte-Wide Write Enables. Spartan-6 devices have byte-wide write enables. This feature can be beneficial to the block RAM access and utilization for the device. In some cases, more efficient use of block RAMs and other resources can be seen with the use of this feature.
- Enable/Reset Priority. Spartan-6 FPGAs can change the priority of enables versus resets, allowing for greater consistency of output register control to that of slices and I/O registers.

Current Limitations on Inferring Block RAMs

For information on limitations on inferring block RAMs, refer to Answer Record 33474 at http://www.xilinx.com/support/answers/33474.htm.

Refer to UG383, *Spartan-6 FPGA Block RAM Resources User Guide*, for more details on the Spartan-6 block RAM resources.

Other Primitive Retargeting Considerations

Some device primitives instantiated for previous Spartan architectures are not automatically retargeted to the Spartan-6 architecture, or they are retargeted with some notable differences. Refer to UG615, *Spartan-6 Libraries Guide for HDL Designs*, for more details on any UNISIM component when targeting Spartan-6 FPGAs.

A few components are highlighted in this section.

IBUF_DLY_ADJ

The IBUF_DLY_ADJ primitive from the Spartan-3A architecture is not supported and cannot be retargeted to the Spartan-6 architecture. If this component is being used in the current design, it must be replaced with a normal IBUF followed by an IODELAY2 to gain similar functionality.

DCM

The DCM_SP component used in Spartan-6, Spartan-3A, and Spartan-3E FPGAs behaves differently from the DCMs in Spartan-3 and Virtex FPGAs. For example, in variable phase shift, the taps advance at different intervals. For many designs, this might not have an effect if the phase shift control looks solely at data patterns. Re-examination might be needed if the control circuitry expects a certain number of steps account to a certain delay.

PLL_ADV

The PLL_ADV might retarget in some circumstances, and might not retarget in others. Clock switchover functionality is not supported in the Spartan-6 architecture; thus, use of the CLKIN1 and CLKIN2 and the associated CLKINSEL can result in an error condition.

Use of the DRP is also not currently recommended. The recommended component to use when targeting the Spartan-6 architecture PLL is the PLL_BASE. However, it is strongly suggested to simply use the Clocking Wizard to ensure a correctly constructed PLL for this architecture.

MUXCY, XORCY, MULTAND, MUXF5, and MUXF6

Due to differences in the Spartan-3 and Spartan-6 architectures, the MULTAND, MUXF5, and MUXF6 are not always retargeted in the most efficient way for area, density, or timing. Instantiated MUXCY and XORCY components are generally retargeted efficiently but must be grouped into a larger CARRY4 components to properly model the timing differences when targeting Spartan-6 devices.

Designs that contain these instantiated components should be re-evaluated to determine whether the code section containing these components can be inferable or replaced with the structural equivalent primitives from the Spartan-6 architecture. These primitives are automatically retargeted.

RAMB4

If a design contains RAMB4 primitives from the original Virtex or Spartan-II families, those block RAM primitives are not automatically retargeted. They must be manually changed to either inference code (preferred) or an instantiation primitive supported by the Spartan-6 device. The primary reason why this retargeting is not supported is the gross inefficiency associated with a direct retarget of the 4 Kb block RAM to the 18 Kb RAM.

I/O Considerations

This section describes retargeting considerations specifically for the Spartan-6 FPGA I/Os. It is strongly suggested that designers become familiar with UG381, *Spartan-6 FPGA SelectIO Resources User Guide*, for details and specifics of the Spartan-6 FPGA I/O features and capabilities to get the most use out of this resource.

Choosing a Pinout

Selecting a pinout in today's FPGAs can be challenging. Designers must weigh different I/O standards, operating and reference voltages, along with other banking rules. At the same time, designers must take into account clocking resources, PCB, simultaneous switching, signal integrity, and pinout selections that give the best possible performance and power results.

This difficult challenge involves many trade-offs not initially apparent. For this reason, it is suggested to always use the PlanAhead™ tool's pin planning environment. Using the PlanAhead tool reduces the risk of creating an invalid pinout and allows better visualization of how the selected pinout can affect design placement, routing, and eventual performance. The PlanAhead tool has many associated design rule checks and other tools to help validate pinouts, reducing the chance of a PCB respin.

Use of the IBUF_DELAY or IFD_DELAY Attributes

The Spartan-6 architecture does not directly support adding a finite input delay to an input path with the IBUF_DELAY and IFD_DELAY attributes. If these attributes are used, it is suggested that the designer instead instantiate an IODELAY2 component and set the appropriate fixed tap value to set an amount of input delay to a path.

Use of DCI

DCI functionality is no longer supported in the Spartan-6 architecture. An alternative is to use the IN_TERM and OUT_TERM attributes. These attributes are easier to use and do not require the external circuitry that DCI requires. However, the I/O impedances are uncalibrated for the effects of PVT, and thus, might have less precise values than DCI.

Conclusion

In general, targeting or retargeting to a Spartan-6 FPGA design can be accomplished without much planning or modification of the existing design. To get the best use of the underlying innovations of the Spartan-6 architecture, additional considerations must be taken into account. Understanding and following many of the guidelines outlined in this white paper can result in improved density (cost), performance, and power.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
10/07/09	1.0	Initial Xilinx release.
01/21/10	1.1	Updated example from the Map file in Use of LUTs as Route-Thrus. Changed the default for the **-lc** switch in Software Algorithms to **-lc off**. Removed the AUTOBUF section.

Notice of Disclaimer