XAPP1134 (v1.3) September 8, 2015

Developing Secure Designs Using the Virtex-5 Family

Author: Steven McNeil

Summary

This application note is written for FPGA designers implementing the single-chip crypto (SCC) technology developed by Xilinx. This document explains how to:

- Implement isolated functions (for example, red/black logic, redundant Type-I encryptors or logic processing multiple levels of security) in a single Virtex®-5 or Virtex-5Q FPGA.
- Verify the isolation using the Xilinx® Isolation Verification Tool (IVT).

With this application note, designers can develop a single-chip cryptographic solution that meets fail-safe and physical security requirements for high-grade, high-assurance applications.

The Security Monitor IP developed by Xilinx provides additional security services for Virtex-5 and Virtex-5Q FPGAs. Contact your local Xilinx representative for more information.

Type 1 Virtex-5 FPGA Crypto applications require defense-grade (XQ) devices for mask control. The design example used in this application note was created using a non defense-grade Virtex-5 XC5VLX85-FF676-2 device.

Introduction

The flexibility of programmable logic affords the aerospace and defense industry many advantages. However, prior to this work, in applications such as information assurance, government contractors and agencies could not realize the full capability of programmable logic due to isolation, reliability, and security concerns. To address these concerns, the SCC solution was developed to allow independent functions to operate on a single chip. Examples of such single-chip applications include, but are not limited to, redundant Type-I encryptors, resident red and black data, and functionality operating on multiple independent levels of security. The successful completion of this work has allowed Xilinx to provide new technology for the information assurance industry.

Single-Chip Crypto Design Flow

Developing a solution with multiple isolated functions in a single FPGA is made possible through Xilinx partial reconfiguration (PR) technology. While not all applications partially reconfigure a device in-system, the PR toolset and design methodology provides the necessary controls to achieve the isolation needed to meet certifying agency requirements. For those applications that require algorithm agility, the same design flow can be used to create partial bitstreams and allow a system to swap cryptographic equipment applications (CEAs), or other logic, on the fly. The designer should have a solid understanding of the standard partial reconfiguration design flow. Many of the terms and processes are utilized in the SCC design flow. Areas that are different supersede the PR design flow and are identified in this application note.

Common Terminology

Throughout this document the terms function, logic, region, and partition are used extensively. They are defined thus:

Function	A collection of logic that performs a specific operation, for example, an AES encryptor.
Logic	Circuits used to implement a specific function, for example, a flip-flop (FF), lookup table (LUT), block RAM, etc.
Region	A physical area configured to implement logic.
Partition	A user-defined portion of logic that can be isolated from other logic by assigning it to a specific region.

Design Flow Details

A single-chip solution can be achieved while preserving FPGA design techniques and coding styles with only moderate modifications to the development flow. SCC development requires the designer to consider floorplanning much earlier in the design process to ensure that proper isolation is achieved in logic, routing, and I/O. In addition to early floorplanning, the development flow is modular—each function a user desires to isolate must be at its own level of hierarchy. From here the designer can take one of two approaches. If the designer wishes to partially reconfigure one or more of any region, each isolated function must be synthesized and implemented independently of the other modules. After each module is implemented the design is merged into a flattened FPGA design for device configuration. If the designer requires only isolation, he or she can synthesize the full design while being careful to maintain at least one level of hierarchy such that SCC constraints can be applied to each module that requires isolation. While this flow requires the FPGA designer to break away from traditional FPGA development flows, the modular approach does have certain advantages. A single-chip cryptographic solution can be generated. If an isolated function requires a change late in the design cycle, only that specific function is modified while the remaining modules remain unchanged.

Note: This is true except for the top-level architecture which is only allowed to contain global clocks, resets, I/Os, and module instantiations.

There are a few unique design details that must be adhered to in order to achieve an FPGA based single-chip cryptographic solution. The designer should carefully consider all aspects of these design details, which are explained in detail in subsequent sections of this application note. These considerations include:

- An isolated function must be at its own level of hierarchy.
- A *fence* is used to isolate functions within a single chip.
- I/Os can be inferred in lower level HDL modules for proper isolation at the I/O.
- On-chip communication between isolated functions is achieved through the use of trusted routing.

Reference Design

For clarity, an example design (Figure 1) is used throughout this application note to describe the design details and tool flow. In addition, this design has been implemented with ISE® tools using the partial reconfiguration licensed features, verified by the Isolation Verification Tool (IVT) and provided to the designer as a reference.

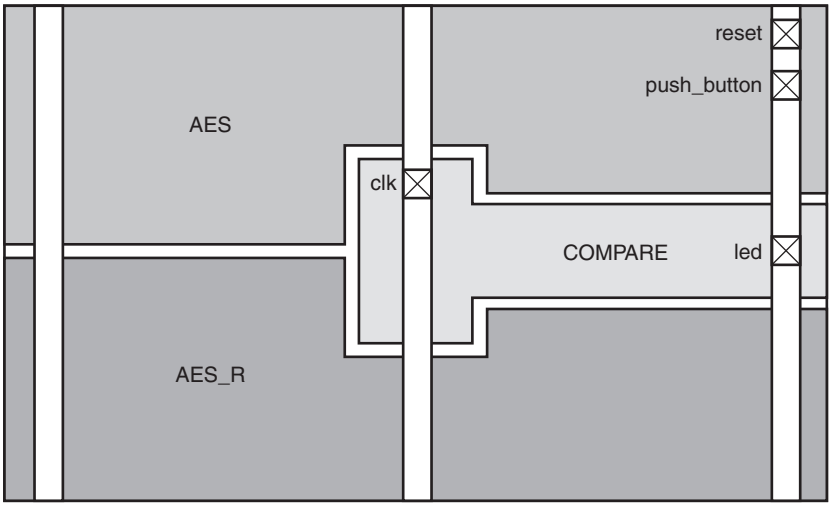
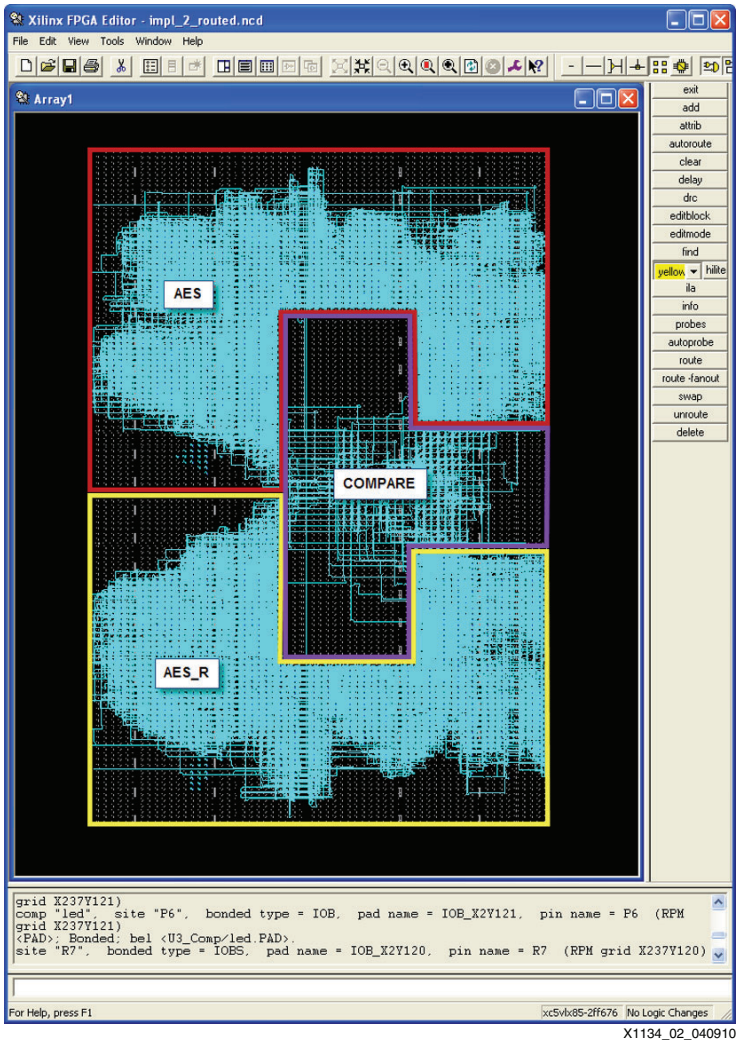


Figure 1: **An SCC Solution Implementing Two 128-bit Redundant AES Encryptors with Comparator Logic**

The example design consists of two redundant advanced-encryption-standard (AES) encryptors whose outputs are sent to a comparator block. The redundant AES encryptors and comparator functions are all isolated within a single FPGA as shown in the FPGA Editor view in Figure 2.



X1134_02_040910

Figure 2: **An SCC Solution Implemented in a Virtex-5/Virtex-5Q LX85 FPGA**

For simplicity, the encryptors are 128-bit AES engines running in codebook mode with a static plain-text word. The design can inject errors via a pushbutton located on one of the AES engines. An LED, driven by the comparison block, indicates when the outputs of the encryptors do not match. While this architecture is chosen somewhat arbitrarily and is not meant to represent an actual user design, it does provide a valuable example in explaining SCC design details.

Table 1 lists the ports and I/Os of each module.

Table 1: **I/Os of Each Module in the SCC Reference Design**

Module	Module Inputs	Module Outputs
TOP	clk(1), reset, push_button	led
AES	clk, reset(1), push_button(1), start, mode, load, key, data_in	reset_out, data_out, done
AES_R	clk, reset, push_button, start, mode, load, key, data_in	data_out, done

Table 1: **I/Os of Each Module in the SCC Reference Design *(Cont'd)***

Module	Module Inputs	Module Outputs
COMPARE	clk, reset, done1, done2,data_in1, data_in2	reset_out, start, load, led(1)

Notes:

1. These IOBs are instantiated in their respective blocks.

Per Design Flow Details, page 2, the example design implementation is performed in a modular fashion. Each function to be isolated has its own level of hierarchy and is implemented independently of the others. When completed, the independent modules are then merged together. Figure 3 shows the hierarchy of the reference design.

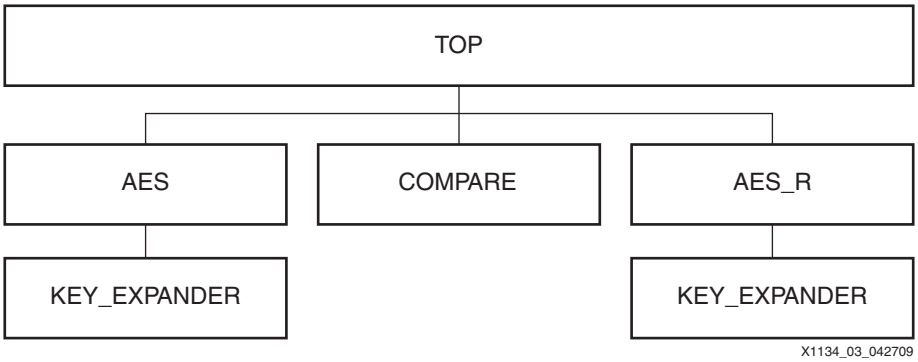


Figure 3: **Hierarchy for SCC Implementation of Reference Design**

Additional lower level hierarchy is allowed for each function. In addition, there is no theoretical limit on the number of functions allowed. For example, if the Security Monitor is to be integrated into this design, it is a fourth function with its own hierarchy under Top.

Isolation Fence

The complexity and density of the Virtex-5 or Virtex-5Q device might seem to preclude any efficient isolation methodology. In addition, the difficulty of performing an isolation analysis is assumed to be magnified considering there are 26 members of the Virtex-5 family. However, Xilinx FPGAs are modular devices. No matter which device is chosen, the FPGAs consist of the same basic building blocks tiled over and over again. In the Virtex-5 architecture, the basic building block is a configurable logic block (CLB). As shown in Figure 4, the CLB includes a global switch matrix (GSM) used for interconnect common to all features in the FPGA. In other words, the same GSM provides interconnect from CLB to CLB, or from any CLB to dedicated functions within the FPGA such as embedded PowerPC® processors, digital clock managers (DCMs), DSP48E blocks, etc.

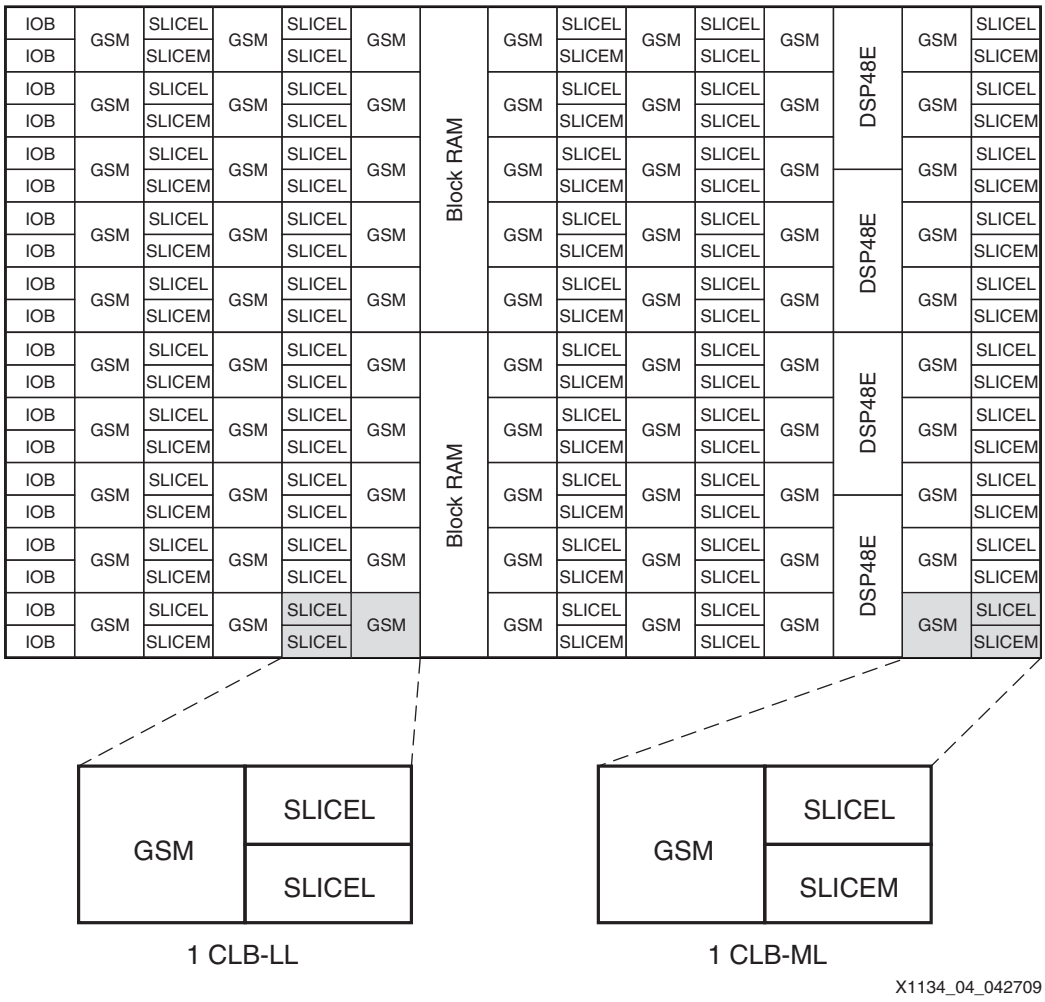


Figure 4: **Representation of the CLB-Based Virtex-5 Architecture**

To achieve isolation within a single device, the concept of a fence is introduced. The fence is just a set of unused CLB(s), in which no routing or logic is present. The result of the isolation analysis performed by Xilinx shows that a single, unused CLB between isolated regions guarantees that no less than three failures are required to compromise isolation. The layout of the reference design, zoomed in for more detail, with isolated regions separated by a single-CLB fence, is shown in Figure 5. The fence location is not directly specified by the designer, rather it is created indirectly by applying the appropriate logic and routing constraints to each user-defined region to be isolated. In the reference design, the three functions that must be isolated are the two redundant AES encryptors and the comparator block.

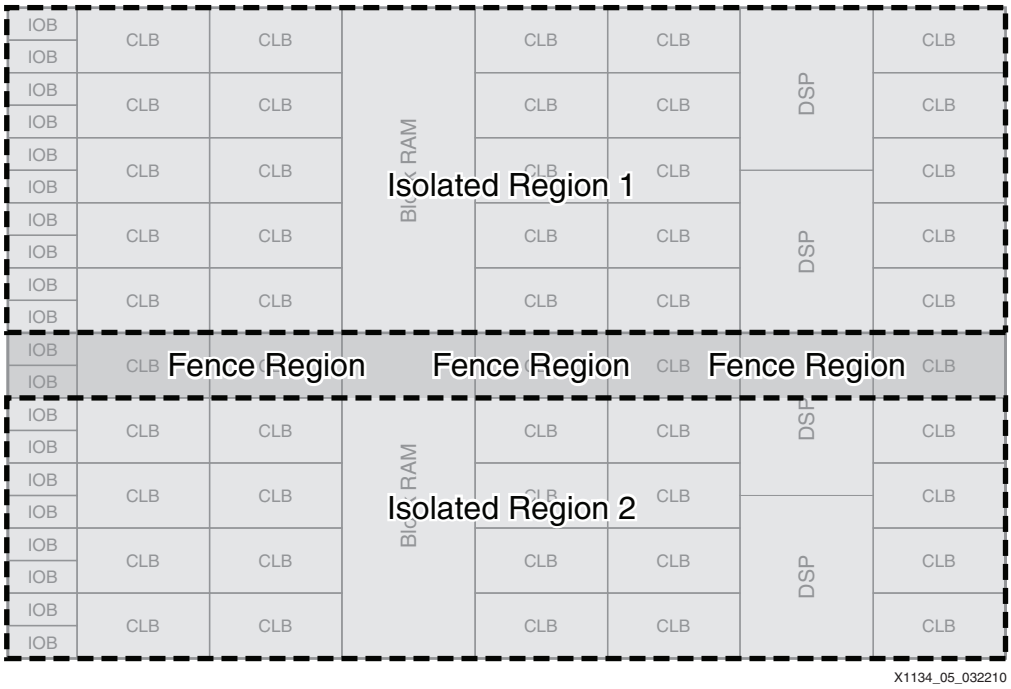


Figure 5: **Isolated Functions Separated by a Fence of Unused Logic**

Floorplanning the Reference Design

The floorplan of the reference design is shown in Figure 6. Where inter-module communication is necessary, regions must be coincident with each other with a fence cell between them (refer to On-Chip Communication, page 13 for more details).

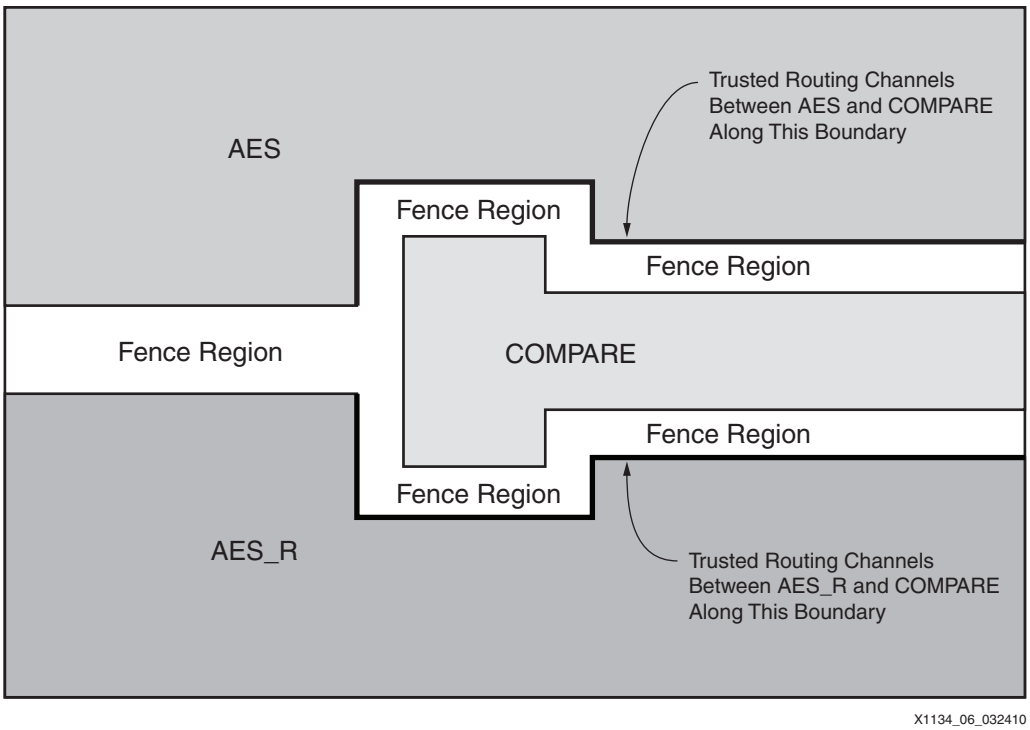


Figure 6: **Floorplanning of the Reference Design**

To constrain the logic to a specific region of the FPGA, the AREA_GROUP constraint is utilized. This constraint is applied to an instance in the HDL representing the function to be isolated. In this example, each of the AES encryptors and comparator functions have an AREA_GROUP constraint applied to it. For more information on the AREA_GROUP constraint and the FPGA coordinate system, see the Constraints Guide installed during software installation. The first step in creating the AREA_GROUP constraint is to analyze the region the function is to be mapped into to ensure enough dedicated resources exist. The dedicated resources can be I/O, FFs, LUTs, block RAM, DSP48E slices, etc. The Xilinx PlanAhead™ tool can be used to assist the designer in this analysis.

The next step is to assign the HDL instance name of the module to a user-specified AREA_GROUP name (in the reference design, the HDL module instance names are U1_AES1, U2_AES2, and U3_Comp). The assignment of the instance name to the AREA_GROUP for the reference design is:

- `INST "U1_AES1"  AREA_GROUP = "pblock_U1_AES1";`
- `INST "U2_AES2"  AREA_GROUP = "pblock_U2_AES2";`
- `INST "U3_Comp"  AREA_GROUP = "pblock_U3_Comp";`

Next, AREA_GROUPs must be assigned to a specific range of logic in the FPGA. AREA_GROUPs can be defined in terms of SLICEs, RAMB16s, IOBs, ILOGICs, OLOGICs, PLLs, DCMs, PPCs, IDELAYs, IDLYCTRLs, BUFGs, BUFGCTRLs, MGTs, GT11CLKs, DSP48Es, etc.

The generic syntax for adding components to an area group is:

```
AREA_GROUP "<AG Name>" RANGE=<comp name>_XaYb:<comp name>_XcYd;
```

Where:

- <AG Name> = name of area group
- <comp name> = name of desired component
  - SLICE
  - RAMB36
  - DSP48E
  - IOB
  - ILOGIC
  - OLOGIC
  - PLL
  - DCM

  The component name can be identified by pointing at it in the PlanAhead tool and reading it off the bottom right hand of the screen.
- a, b, c, d = Coordinates of the starting component and the ending components

  Coordinates can also be identified from the same location in the bottom right-hand corner of the PlanAhead tool that the component name was identified.

Some examples of this syntax are as follows:

- `AREA_GROUP "pblock_U1_AES1" RANGE=SLICE_X76Y71:SLICE_X107Y119;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=DSP48_X0Y24:DSP48_X0Y47;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=RAMB36_X3Y15:RAMB36_X3Y23;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=IOB_X2Y142:IOB_X2Y239;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=ILOGIC_X2Y142:ILOGIC_X2Y239;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=OLOGIC_X2Y142:OLOGIC_X2Y239;`

- `AREA_GROUP "pblock_U1_AES1" RANGE=IDELAYCTRL_X2Y4:IDELAYCTRL_X2Y5;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=IODELAY_X2Y142:IODELAY_X2Y239;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=DCM_ADV_X0Y8:DCM_ADV_X0Y11;`

Note: Although the reference design does not use DSPs or block RAMs, they have been added to the area group so that the routing resources contained by these blocks can be used. As a general rule, all available resources should be assigned to the area group unless there is specific need to exclude that resource. This is the default selection when generating the constraint file using the PlanAhead tool.

These constraints can be generated by hand or with the recommended PlanAhead development tool.

Communicating with Isolated Functions

Communication with isolated functions in the FPGA and other components within a system occurs through user I/Os. However, to maintain isolation from logic to the I/Os and ultimately the PCB, the I/Os must be considered part of the isolation region. If the I/Os are not included in the isolated region, then any routing from the isolated region to the I/Os cannot be controlled and isolation cannot be guaranteed (Figure 7). When the I/Os are included in the isolated region, the routing is controlled and isolation is guaranteed (Figure 8).

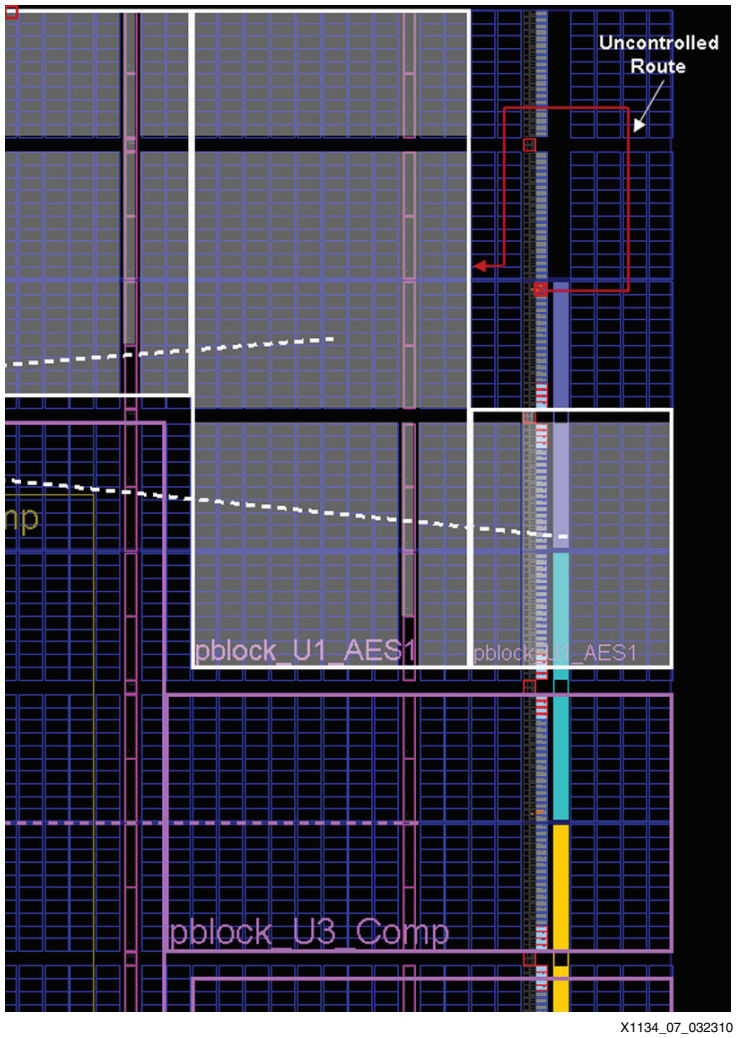


Figure 7: **An Uncontrolled Route from an I/O to an Isolated Function**

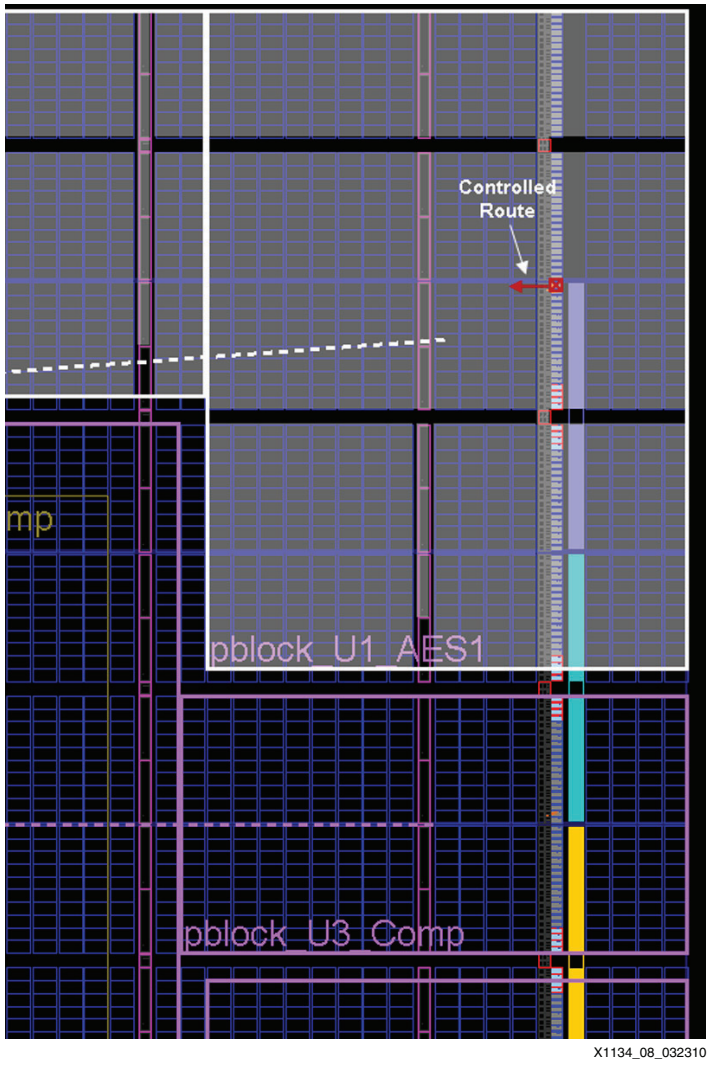


Figure 8: **Controlled Routing Between an Isolated Function and the I/O Included the Isolated Region**

Communication between two isolated functions within the FPGA typically uses trusted routing. A less preferable alternative is to use I/Os to go off-chip and then back on again into a different region. Each of these alternatives has specific constraints that must be considered before choosing how to communicate with isolated functions.

Off-Chip Communication

If an isolated function has inputs or outputs that must come from or go off-chip, these signals must have their I/O buffers inferred at the lower level module. While this is different from standard FPGA design practice, it is required in order to have control over the routing of the signals from the I/O buffer to the function. If the I/O is not part of the isolated region, then there is no control of how the signal is routed from the I/O buffer to the logic. Standard design practice is to define all of the FPGA I/Os in the top-level port map (for VHDL users). During synthesis, all of the I/O buffers are inferred at the top level. However, in the SCC design flow, I/O buffer insertion is done at both the top level and lower level HDL modules. For simulation, all of the I/Os are still defined at the top level in the SCC flow; however, where the I/O buffers are inferred depends on whether or not the signal must be isolated. For the SCC design flow, the only signals that should have I/O buffers inferred at the top level are global clocks, resets, and other control signals that do not require isolation. All other signals that go off-chip should have their I/O buffers inferred at the lower level modules. Because the SCC development flow is modular

and each isolated function is implemented independently of the other, the development tools treat each module port declaration as being the design top level. As an isolated function might not be at the top level, a method must be employed to identify which signals the development tools should add I/O buffers to and which it should not. A simple Xilinx Synthesis Tool (XST) attribute statement is used to keep I/Os from being inferred:

```
attribute buffer_type: string;
attribute buffer_type of signal_name: signal is "none";
```

Note: All I/Os are still identified in the top level of the hierarchy to enable design verification through standard functional and back-annotated timing simulations.

For the reference design, each of these functions must be isolated from one another, and each of these functions contain dedicated I/Os. To infer only the global clock and reset in the top-level port declaration, the attribute is applied to all signals that should not have their I/O buffers inserted.

```
entity SCC_LAB_TOP is
Port (
       push_button      : in  std_logic;
       reset            : in  std_logic;
       clk              : in  std_logic;
       led              : out std_logic
     );

attribute buffer_type: string;
attribute buffer_type of push_button      : signal is "none"; -- Pin in AES1
attribute buffer_type of reset            : signal is "none"; -- Pin inside AES1
--attribute buffer_type of clk            : signal is "none"; -- Top Level Pin
attribute buffer_type of led              : signal is "none"; -- Pin in Compare

end SCC_LAB_TOP;
```

To infer only the AES dedicated off-chip I/Os in the AES1 module, the attribute is applied to all signals that are not meant to be external I/Os.

```
entity aes is
port(
      clk          : in  std_logic;
      reset        : in  std_logic;
      push_button  : in  std_logic;  -- must be put in AES1 block for inc flow
      start        : in  std_logic;  -- to initiate the encryption/decryption
                                     -- process after loading
      mode         : in  std_logic;  -- to select encryption or decryption
      load         : in  std_logic;  -- to load the input and keys.has to
      key          : in  std_logic_vector(63 downto 0);
      data_in      : in  std_logic_vector(63 downto 0);
      reset_out    : out std_logic;
      data_out     : out std_logic_vector(127 downto 0);
      done         : out std_logic
     );

attribute buffer_type: string;
attribute buffer_type of clk         : signal is "none";   -- Top level PIN
--attribute buffer_type of reset       : signal is "none"; -- Hooked up to IBUF
--attribute buffer_type of push_button : signal is "none"; -- Hooked up to IBUF
attribute buffer_type of start       : signal is "none";   -- Port
attribute buffer_type of mode        : signal is "none";   -- Port
attribute buffer_type of load        : signal is "none";   -- Port
attribute buffer_type of key         : signal is "none";   -- Port
attribute buffer_type of data_in     : signal is "none";   -- Port
attribute buffer_type of reset_out   : signal is "none";   -- Port
attribute buffer_type of data_out    : signal is "none";   -- Port
attribute buffer_type of done        : signal is "none";   -- Port

end aes;
```

Note: The only AES I/Os that go off-chip are the signals push_button and reset. All of the other signals do not have I/O buffers inferred because they are either inferred elsewhere (for example, in the top level of hierarchy) or are not required because the signals are internal only.

To infer only the AES_r dedicated off-chip I/Os in the AES2 module, the attribute is applied to all signals not intended to be external I/Os:

```
entity aes_r is
port(
      clk       : in std_logic;
      reset     : in std_logic;
      push_button  : in std_logic;  -- must be put in aes128_fast block for inc flow
      start     : in std_logic;     -- to initiate the encryption/decryption process
                                    -- process after loading
      mode      : in std_logic;      -- to select encryption or decryption
      load      : in std_logic;      -- to load the input and keys.has to
      key       : in std_logic_vector(63 downto 0);
      data_in   : in std_logic_vector(63 downto 0);
      data_out  : out std_logic_vector(127 downto 0);
      done      : out std_logic
     );

attribute buffer_type: string;
attribute buffer_type of clk         : signal is "none";  -- PIN at top level
attribute buffer_type of reset       : signal is "none";  -- PORT only
attribute buffer_type of push_button : signal is "none";  -- PORT only
attribute buffer_type of start       : signal is "none";  -- PORT only
attribute buffer_type of mode        : signal is "none";  -- PORT only
attribute buffer_type of load        : signal is "none";  -- PORT only
attribute buffer_type of key         : signal is "none";  -- PORT only
attribute buffer_type of data_in     : signal is "none";  -- PORT only
attribute buffer_type of data_out    : signal is "none";  -- PORT only
attribute buffer_type of done        : signal is "none";  -- PORT only

end aes_r;
```

Note: The redundant AES function has no external I/Os; therefore, the attribute is applied to all of the signals.

To infer only the COMPARE off-chip I/Os in the COMPARE module, the attribute is applied to all signals not intended to be external I/Os:

```
entity compare is
port(
      clk             : in  std_logic;
      reset           : in  std_logic;
      reset_out       : out std_logic;
      done1           : in  std_logic;
      done2           : in  std_logic;
      start_aes1      : out std_logic;
      start_aes2      : out std_logic;
      load_aes1       : out std_logic;
      load_aes2       : out std_logic;
      data_in1        : in  std_logic_vector(127 downto 0);
      data_in2        : in  std_logic_vector(127 downto 0);
      led             : out std_logic -- must be put in compare block for inc flow
     );

attribute buffer_type: string;
--attribute buffer_type of led         : signal is "none"; -- PIN at top level
attribute buffer_type of clk         : signal is "none";   -- PORT only
attribute buffer_type of reset       : signal is "none";   -- PORT only
attribute buffer_type of reset_out   : signal is "none";   -- PORT only
attribute buffer_type of done1       : signal is "none";   -- PORT only
attribute buffer_type of done2       : signal is "none";   -- PORT only
attribute buffer_type of start_aes1  : signal is "none";   -- PORT only
attribute buffer_type of start_aes2  : signal is "none";   -- PORT only
attribute buffer_type of load_aes1   : signal is "none";   -- PORT only
attribute buffer_type of load_aes2   : signal is "none";   -- PORT only
```

```
attribute buffer_type of data_in1    : signal is "none";   -- PORT only
attribute buffer_type of data_in2    : signal is "none";   -- PORT only

end compare;
```

Note: The only COMPARE I/O that goes off-chip is the signal led. All other signals do not have I/O buffers inferred because they are either inferred elsewhere (for example, in the top level of hierarchy) or are not required because the signals are internal only.

In summary, isolation from the logic, through the I/O and onto the PCB, can be achieved by using the buffer_type attribute statement. This attribute dictates which signals have I/O buffers inferred to guarantee isolation without altering the coding style.

On-Chip Communication

When communication between isolated functions is required, there are two possible solutions:

- Trusted routing (preferred method).
- Signals can be taken off-chip from one isolated region, routed through the PCB, and then brought back on-chip in a separate isolated region. This method is not preferred.

Trusted Routing Rules

Trusted routing is automatic. The design tools recognize the communication between isolated regions and use the trusted routing resources. These rules for trusted routing must be adhered to:

- Rule 1: Feedthrough signals are not allowed without buffering (see Figure 9).
  - If a signal is directly connected to both an input port and an output port, it must be buffered.
  - Direct instantiation of a buffer (LUT1, for example) might be necessary. This isolates area groups, with the LUT preventing a shorted net.

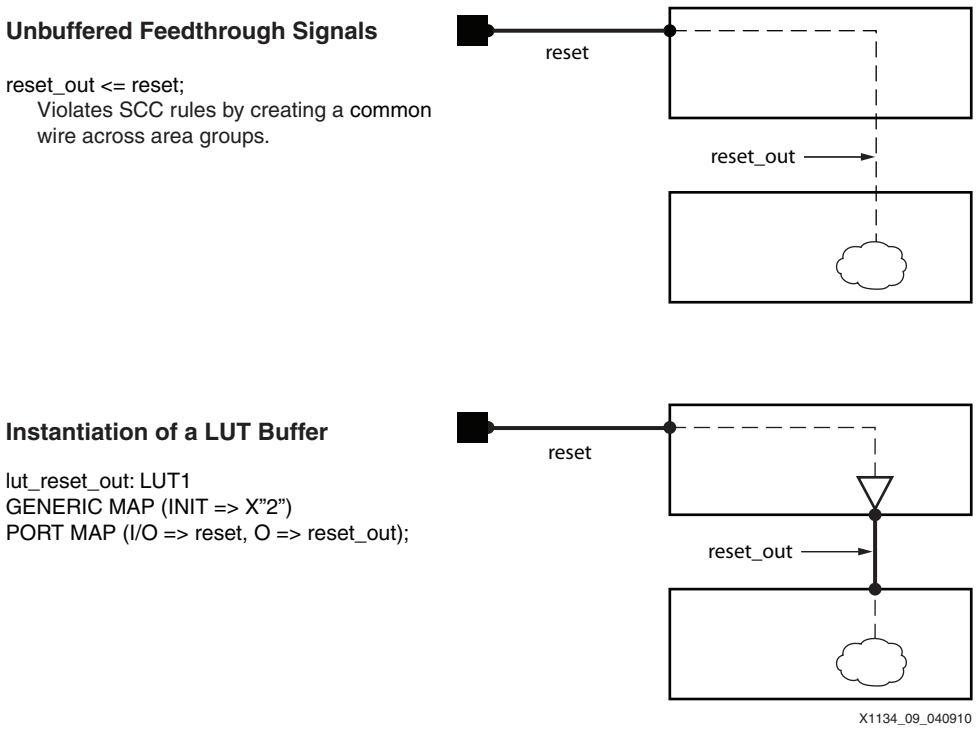


Figure 9: **Trusted Routing Rule 1 – Feedthrough Signals**

- Rule 2: A module output port cannot connect to more than one module input port (see Figure 10).
  - The user must create two different ports for such a connection.
  - Each port must not violate Rule 3.

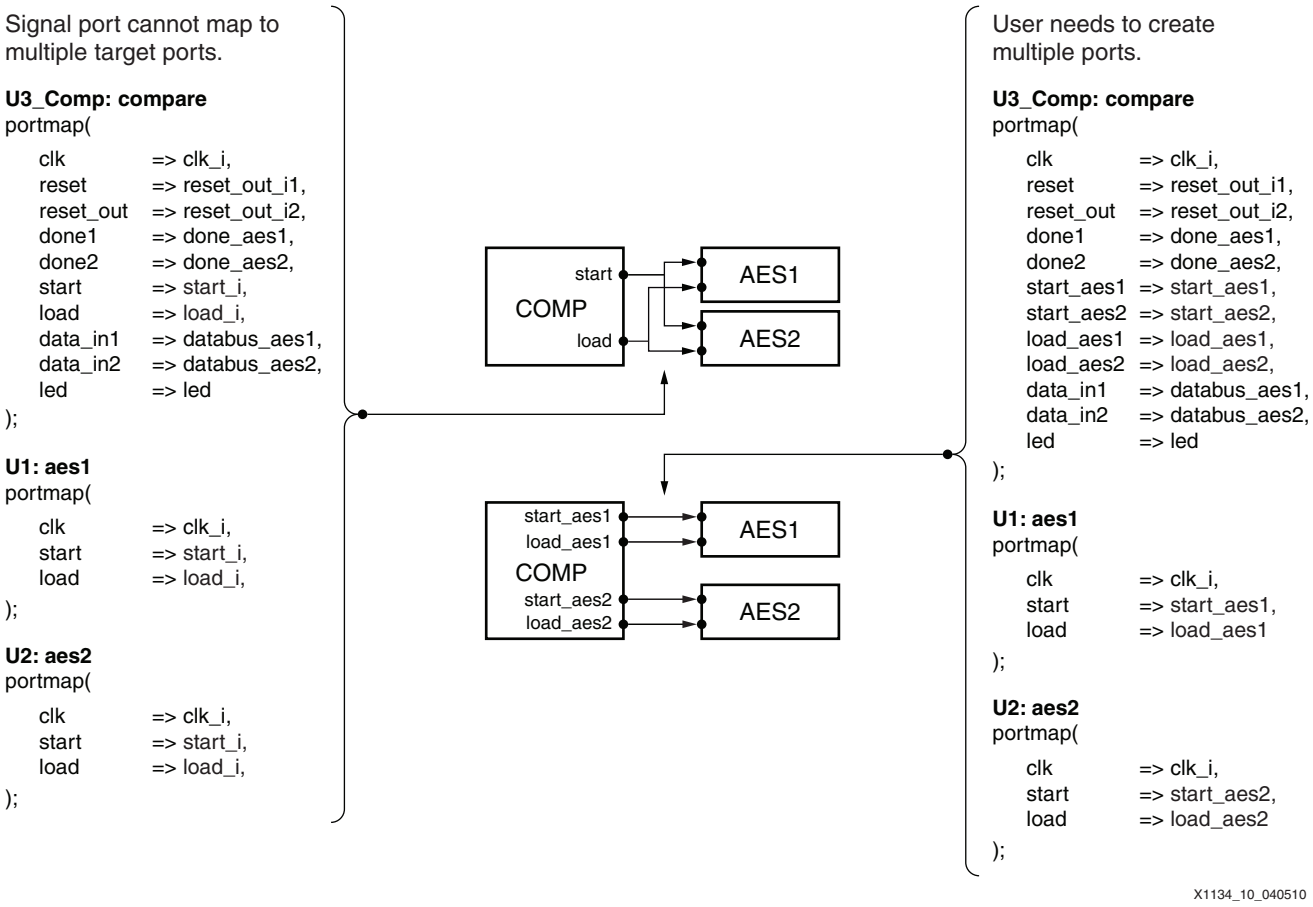
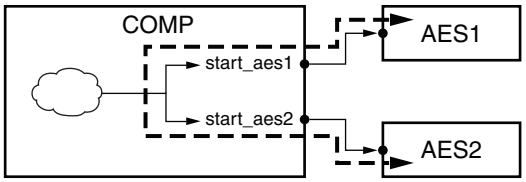


Figure 10: **Trusted Routing Rule 2 – Multi-Port Connections**

- Rule 3: One signal cannot drive two different output ports of the same module (see Figure 11).
  - Each port must have a unique driver.
  - Direct instantiation of a buffer (LUT1, for example) might be necessary. This isolates area groups, with the LUT preventing a shorted net.

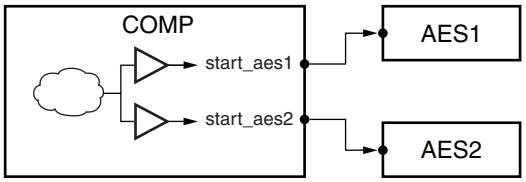
Unbuffered Branch Signals

```
start_aes1 <= start_i;
start_aes2 <= start_i;
    Violates SCC rules by creating a
    short between area groups
```



Instantiation of a LUT Buffer

```
lut_start_aes1: LUT1
    GENERIC MAP (INIT => X"2")
    PORT MAP (I0 => start_i, 0 => start_aes1);
lut_start_aes2: LUT1
    GENERIC MAP (INIT => X"2")
    PORT MAP (I0 => start_i, 0 => start_aes2);
```



X1134_11_040510

Figure 11: **Trusted Routing Rule 3 – Branched Signals**

Constraining Placement

Placement constraint is now achieved by the addition of components to the area group. Components not in an area group cannot be used. Because this includes top-level clock components such as DCMs, an additional constraint is necessary. The constraint PRIVATE = NONE allows top-level routes to use components owned by each area group. The PRIVATE constraint is set to NONE as follows:

- `AREA_GROUP "pblock_U1_AES1" PRIVATE = NONE;`
- `AREA_GROUP "pblock_U2_AES2" PRIVATE = NONE;`
- `AREA_GROUP "pblock_U3_Comp" PRIVATE = NONE;`

Constraining Routing

In the past, the complexity of controlling routing has made it difficult to create isolated functions within a single chip. Controlling routing is now as easy as assigning the CONTAINED constraint to ROUTE (CONTAINED = ROUTE). As the name implies, this constraint keeps all routing in a specific module within its boundary defined by the area group. An example of the CONTAINED constraint applied to the reference design is shown below:

- `AREA_GROUP "pblock_U1_AES1"  CONTAINED = ROUTE;`
- `AREA_GROUP "pblock_U2_AES2"  CONTAINED = ROUTE;`
- `AREA_GROUP "pblock_U3_Comp"  CONTAINED = ROUTE;`

While the contained constraint does not allow a used portion of a routing resource to exit the user-defined region, the BOUNDARYCROSS constraint disallows the unused portion (“stubs”) of a routing resource to exit the user-defined region. In other words, the constraint takes into consideration the entirety of the route when determining if the route could cross the boundary. For example, a PENT route, which traverses five CLBs, has one destination point three CLBs from the source and a second destination five CLBs from the source. In this example, there can be logic in the source and first destination CLB. The route from the first destination to the second destination is unused but active (a stub). The BOUNDARYCROSS constraint takes this stub into consideration. If any portion of the route crosses the boundary, including the stub, the

development tools prevent its use. An example of the BOUNDARYCROSS constraint applied to the reference design is shown below:

- `AREA_GROUP "pblock_U1_AES1" BOUNDARYCROSS = NO;`
- `AREA_GROUP "pblock_U2_AES2" BOUNDARYCROSS = NO;`
- `AREA_GROUP "pblock_U3_Comp" BOUNDARYCROSS = NO;`

Constraint Summary

Some initial architecting and floorplanning of the FPGA, coupled with a small list of constraints, are all that is required to achieve isolation of specific functions within a single Virtex-5 or Virtex-5Q FPGA.

The definition of whether a module is reconfigurable or simply isolated is no longer contained in the UCF. It is determined in the PlanAhead tool by setting the module as a partition, reconfigurable or otherwise.

All constraints applied to the AES1 encryptor are combined:

```
INST "U1_AES1" AREA_GROUP = "pblock_U1_AES1";
AREA_GROUP "pblock_U1_AES1" RANGE=SLICE_X76Y71:SLICE_X107Y119,
       SLICE_X44Y91:SLICE_X75Y119, SLICE_X0Y60:SLICE_X43Y119;
AREA_GROUP "pblock_U1_AES1" RANGE=BUFIO_X2Y16:BUFIO_X2Y23,
       BUFIO_X0Y12:BUFIO_X0Y23;
AREA_GROUP "pblock_U1_AES1" RANGE=BUFR_X1Y8:BUFR_X1Y11,
       BUFR_X0Y6:BUFR_X0Y11;
AREA_GROUP "pblock_U1_AES1" RANGE=DCI_X2Y4:DCI_X2Y5, DCI_X0Y3:DCI_X0Y5;
AREA_GROUP "pblock_U1_AES1" RANGE=DCM_ADV_X0Y8:DCM_ADV_X0Y11;
AREA_GROUP "pblock_U1_AES1" RANGE=DSP48_X0Y24:DSP48_X0Y47;
AREA_GROUP "pblock_U1_AES1" RANGE=IDELAYCTRL_X2Y4:IDELAYCTRL_X2Y5,
       IDELAYCTRL_X0Y3:IDELAYCTRL_X0Y5;
AREA_GROUP "pblock_U1_AES1" RANGE=ILOGIC_X2Y142:ILOGIC_X2Y239,
       ILOGIC_X0Y120:ILOGIC_X0Y239;
AREA_GROUP "pblock_U1_AES1" RANGE=IOB_X2Y142:IOB_X2Y239,
       IOB_X0Y120:IOB_X0Y239;
AREA_GROUP "pblock_U1_AES1" RANGE=IODELAY_X2Y142:IODELAY_X2Y239,
       IODELAY_X0Y120:IODELAY_X0Y239;
AREA_GROUP "pblock_U1_AES1" RANGE=OLOGIC_X2Y142:OLOGIC_X2Y239,
       OLOGIC_X0Y120:OLOGIC_X0Y239;
AREA_GROUP "pblock_U1_AES1" RANGE=PLL_ADV_X0Y4:PLL_ADV_X0Y5;
AREA_GROUP "pblock_U1_AES1" RANGE=PMVBRAM_X2Y5:PMVBRAM_X2Y5,
       PMVBRAM_X0Y3:PMVBRAM_X1Y5;
AREA_GROUP "pblock_U1_AES1" RANGE=RAMB36_X3Y15:RAMB36_X3Y23,
       RAMB36_X2Y19:RAMB36_X2Y23, RAMB36_X0Y12:RAMB36_X1Y23;
AREA_GROUP "pblock_U1_AES1" BOUNDARYCROSS=NO;
AREA_GROUP "pblock_U1_AES1" CONTAINED=ROUTE;
AREA_GROUP "pblock_U1_AES1" PRIVATE=NONE;
```

Isolation Verification Tool

IVT verifies that an FPGA design partitioned into isolated regions meets stringent standards for fail-safe design. IVT is used at two stages in the FPGA design process. Early in the design flow, IVT is used to perform a series of design rule checks on floorplans and pin assignments. Use of IVT at this stage in the design flow is optional though highly recommended. The goal of the UCF checking is to identify potential isolation problems before board layout is committed. After the design is complete, IVT is used again to validate that the required isolation is built into the design. The use of IVT at this stage in the design flow is mandatory, and the results must be supplied to a certifying agency for verification. IVT is a command line tool, running under Windows® 2000, Windows XP, and Linux operating systems and with the released partial reconfiguration software installed. The directory containing the Xilinx executables and dynamic link libraries must be listed in the PATH environment variable for IVT to locate them.

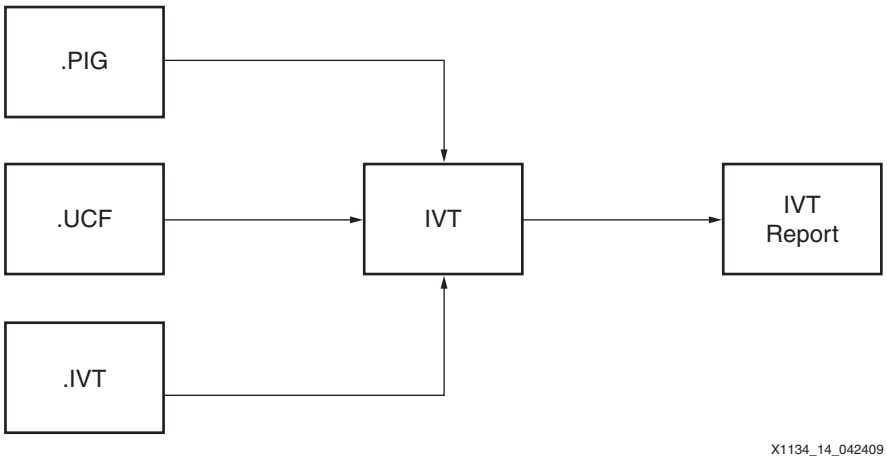


Figure 12: **IVT Inputs and Outputs at the UCF Phase**

Specifically, IVT checks that:

- Pins from different isolation groups are not physically adjacent at the die. I/Os are aligned in columns in the Virtex-5/Virtex-5Q FPGA; therefore, adjacency is defined vertically, immediately above, or below.
- Pins from different isolation groups are not physically adjacent at the package. Adjacency is defined in eight compass directions: north, south, east, west, northeast, southeast, northwest, and southwest.
- Pins from different isolation groups are not co-located in an I/O bank.
- The AREA_RANGE constraints in the UCF are defined such that a minimum of a one CLB-wide fence exists between them.

Because the UCF file might not specify the target device or package, they must be specified via the command line. The association between IVT isolation groups and user-defined area groups is also specified on the command line. The ability for the user to associate an IVT isolation group to the user-defined area groups allows for more than one area group to belong to a single IVT isolation group. An example of IVT execution is:

```
ivt -f parameter_file_name.txt
```

The output file provides provenance, an isolation summary, and details of any isolation failures. Table 2 lists the command-line arguments for IVT operation.

Table 2: **IVT UCF Checker Command Line Details**

IVT Command Line Argument(1)	Description
-device *device_name*	Specifies the device
-package *package_name*	Specifies the package
-group *isolation_group area_group*	At least two distinct isolation groups are required
[-spacing *clbs*]	Ranges from 1 to 10; defaults to 2 CLBs
[-pig *pin_isolation_groups*]	If omitted, no pin-related analysis performed
[-output *output*]	The name of the output text file
[-verbose]	Writes out all IVT reporting information
[-f *parameter_file*]	Arguments can be stored and documented in a file

Table 2: **IVT UCF Checker Command Line Details *(Cont'd)***

IVT Command Line Argument[1]	Description
user_constraint_file	Constraint file for the design (UCF)

Notes:

1. Optional arguments are in brackets: [ ]. Italics denotes variables.

Parameter File

The command-line arguments for IVT are somewhat long, especially for checking UCFs. Therefore, it is recommended that a file containing the arguments be created and supplied to IVT rather than typing them directly at the command line. Arguments can be spread across multiple lines, with blank lines and lines beginning with # ignored (comments):

```
-device xc5vlx85 -package ff676

# Groups     Isolation       Group Area Group
#---------- ------------    --------------------
-group        AES           pblock_U1_AES1
-group        AES_r         pblock_U2_AES2
-group        COMPARE       pblock_U3_Comp

# Pin Isolation Groups
-pig SCC-LAB.pig

# User Constraint File
..\fp_v5lx85_ff676-2_1.ucf

# Output Report File
-output SCC-LAB_ucf.rpt
```

Pin Isolation Group File

Designs containing red and black logic and routing can require isolation at the device inputs and outputs. In this scenario, a pin isolation group (PIG) file is required. The PIG file specifies pins belonging to each isolation group, allowing IVT to perform design rules checks. The PIG file format is simple and allows an FPGA designer to cut and paste the I/O location constraints directly from the UCF into the PIG text file (file extension `.pig`). An isolation group is defined with the following syntax:

```
ISOLATION_GROUP group_name BEGIN
# cut and paste I/O location constraints here.
END ISOLATION_GROUP
```

Note: Only the net names are significant. All other syntax within the I/O location constraints is ignored.

Below is an example PIG file for the reference design:

```
# Comment out global signals
#NET "clk"   LOC = F14;

ISOLATION_GROUP AES BEGIN
NET "push_button" LOC = D10;
NET "reset" LOC = D11;
END ISOLATION_GROUP

ISOLATION_GROUP AES_r BEGIN
# NO PINS
END ISOLATION_GROUP

ISOLATION_GROUP COMPARE BEGIN
NET "led" LOC = P6;
END ISOLATION_GROUP
```

Final Isolation Verification

The primary use of IVT is to provide independent verification that the isolation designed into the FPGA remains before the bit file is created for configuration. In this phase, IVT checks pin locations as before. However, rather than checking the design constraints, IVT checks the actual placement and routing.

Running IVT for Final Isolation Verification

The inputs to IVT in this phase are simpler than those used for UCF checking (Figure 13). The inputs are placed and routed databases (NCD files) for each isolated function. Because the SCC design flow is modular, with each isolated function implemented independently, the NCD file from the merged or combined design is required in addition to the individual NCD files for each isolated function.

Note: More than one NCD file can be associated with a single isolation group if necessary. IVT obtains the device, package, and networks of isolated designs directly from the combined NCD file. In this example, the more simple single NCD file approach is taken.

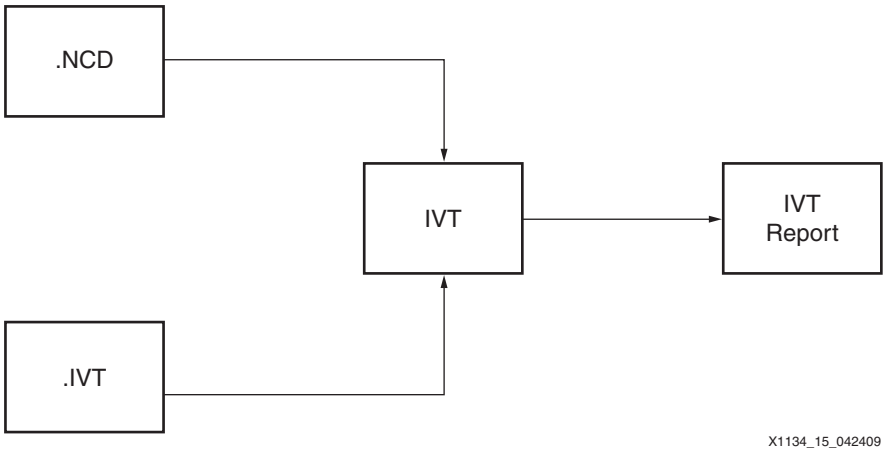


Figure 13: **IVT Inputs and Outputs for Final Verification**

Specifically, IVT checks that:

- Pins from different isolation groups are not physically adjacent at the die. I/Os are aligned in columns in the Virtex-5 or Virtex-5Q FPGA. Therefore adjacency is defined vertically, in either a north or south direction.
- Pins from different isolation groups are not physically adjacent in the package. Adjacency is defined in eight compass directions: north, south, east, west, northeast, southeast, northwest, and southwest.
- Pins from different isolation groups do not share an I/O bank.
- Functions are isolated by a user-supplied number of potential faults.

If information potentially leaks from one isolated region to another by the introduction of fewer than the user-supplied number of faults, IVT reports a faulty path. In this context, the faulty path refers not to an actual defect present in a design, but rather to a possible defect introduced with a limited number of changes or failures that might occur by accident or by tampering.

An example of IVT execution is:

```
ivt -f parameter_file_name.txt
```

Table 3 list the command-line arguments for IVT operation.

Table 3: **IVT Final Isolation Verification Command Line Details**

IVT Command Line Argument(1)	Description
-group *isolation_group area_group*	The isolation group is a user-specified name. The area group name must match the instance name inside the final NCD file.
[-faults *faults*]	Ranges from 2 to 10; defaults to 2 CLBs
[-output *output*]	The name of the output text file
[-verbose]	Writes out all IVT reporting information
[-f *parameter_file*]	Arguments can be stored and documented in a file
combined_ncd_file	Merged NCD file

Notes:

1. Optional arguments are in brackets: [ ]. Italics denotes variables.

An example, the example IVT parameter file (.ivt) for the redundant AES reference design is shown below:

```
# Uncomment the next line for verbose output.
-verbose
# Groups  Isolation Group    Instance Name in Final NCD
#-------- ----------------   --------------------------
-group     AES                U1_AES1
-group     AES_r              U2_AES2
-group     COMPARE            U3_Comp


# Combined design
..\PlanAhead\FloorPlan_SCC\FloorPlan_SCC.runs\impl\floorplans\floorplan_1
\impl_1\impl_1_routed.ncd

# Output Report File
-output SCC-LAB_ncd.rpt
```

Final Verification Output Report

The output file has a number of sections to provide provenance, design details, an isolation summary, and details of any isolation failures. Table 4 provides a brief description of each section written when the verbose option is selected.

Table 4: **Final Verification Report Details**

Section	Description
Isolation Verification Report	Provides provenance.
Input Designs	Provides provenance.
Non-User Digital Clock Manager Networks	DCM calibration logic inserted automatically by the ISE development tools. Because these networks are clocks, they do not require to be isolated.
User Networks Exclusively in the Combined Design	Networks appearing in the merged design and not in each isolated function design are reported.
Unidentified Shared Networks	Networks shared between isolated functions are reported. It is incumbent upon the user to prove these signals do not violate data isolation requirements. Only power, ground, global resets, or explicitly permitted control signals can be shared.

Table 4: **Final Verification Report Details *(Cont'd)***

Section	Description
Identified Shared Networks	There are a number of subsections to this portion of the report. The data is provided for information only and can be safely ignored.
Package Pins, I/O Buffers, and I/O Banks	Detailed summary of signals attached to FPGA I/O. Includes I/O die location, package location, and bank location.
Pin Isolation Summary	Summary of I/O isolation checks.
Network Isolation Summary	Summary of network routing checks. If a routing failure is detected, an FPGA Editor script is included.

A valuable feature of IVT is the FPGA Editor script generated if an isolation failure is detected. When this failure occurs, the user simply loads the merged NCD file into FPGA Editor and then runs the script provided by IVT. When FPGA Editor runs the script, the area of the potential fault is displayed and the fault temporarily selected in yellow (Figure 14). The source net is highlighted in blue, and the load net is highlighted in red. The highlighted error might not actually appear in the user's design. The highlighted net represents an isolation rule violation because the regions can be connected with fewer faults than are allowed by the user.

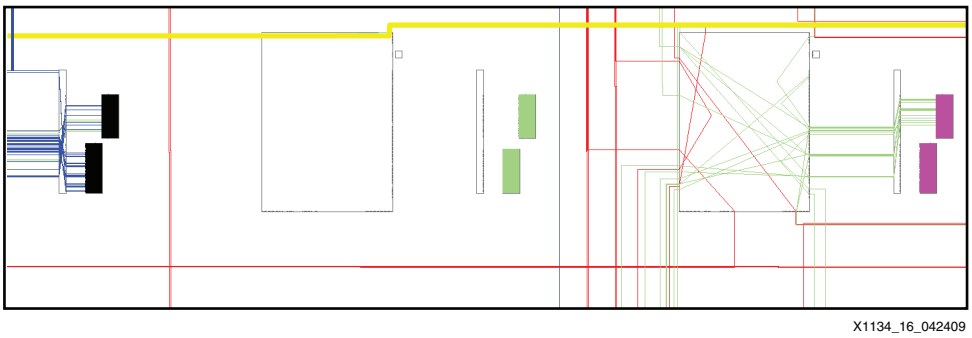


Figure 14: **FPGA Editor View Showing a Potential Fault Highlighted in Yellow**

Design Guidance

This section provides an overview of the SCC design flow, tools utilized, and design guidelines.

Single-Chip Crypto Design Flow

Figure 15 illustrates the SCC design flow.

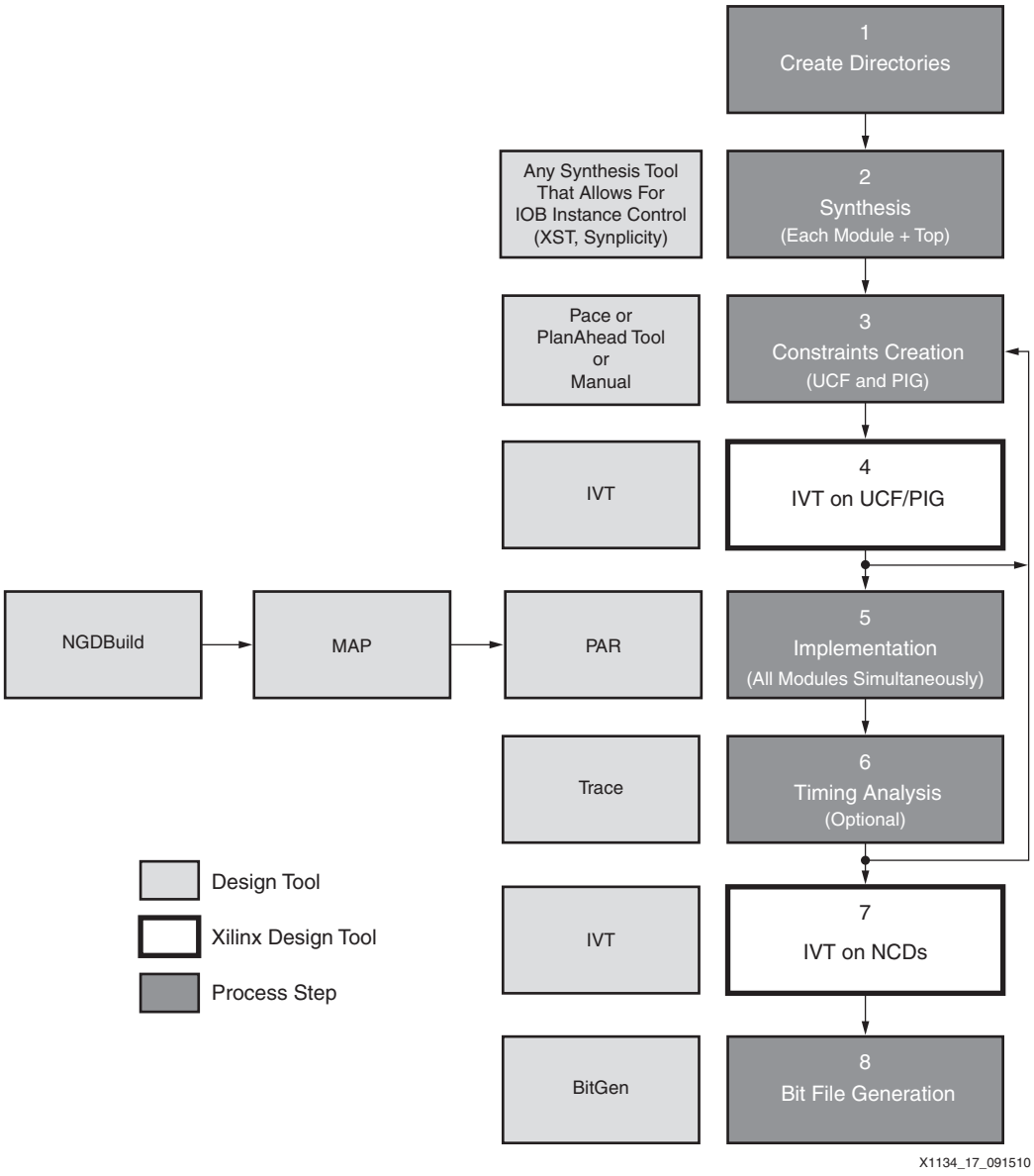


Figure 15: **SCC Design Flow**

Partition Overview

All the partition definitions and controls are done in the `xpartition.pxml` file. This file can be created manually, or automatically with the PlanAhead software GUI, and is described in the Design Preservation Flows chapter in UG748, *Hierarchical Design Methodology Guide*.

Creating the `xpartition.pxml` file automatically is the easier method. The file can be created automatically when launching from the PlanAhead tool. ISE tools, versions 12.1 and later, are required.

If the user wishes to implement the `xpartition.pxml` file automatically using the PlanAhead tool, the new file is copied up to the top level of the project for future command line use. The commands in the `xpartition.pxml` file are case sensitive.

The following is an example `xpartition.pxml` file, with a top-level partition (SCC_LAB_TOP) and three isolated partitions (U1_AES1, U1_AES2, and U3_Comp):

```
<?xml version="1.0"?>
  <Project Name="impl_1" FileVersion="1.2" ProjectVersion="2.0">
    <Partition Name="/SCC_LAB_TOP" State="implement"
    ImportLocation="NONE">
    <Partition Name="/SCC_LAB_TOP/U1_AES1" State="implement"
    ImportLocation="NONE" Reconfigurable="true"
    ReconfigModuleName="module_1" Isolated="true">
    </Partition>
    <Partition Name="/SCC_LAB_TOP/U2_AES2" State="implement"
    ImportLocation="NONE" Reconfigurable="true"
    ReconfigModuleName="module_1"
    Isolated="true">
    </Partition>
    <Partition Name="/SCC_LAB_TOP/U3_Comp" State="implement"
    ImportLocation="NONE" Reconfigurable="false" Isolated="true">
    </Partition>
  </Partition>
</Project>
```

Trusted Routing Design Guidelines

This subsection provides some useful pointers for the designer:

- Each isolated region must contain sufficient device resources to implement all functions that might occupy that region. These functions include LUTs, flip-flops, I/Os, block RAM, DSP48, PPC, etc. The PlanAhead tool can assist in estimating logic for a specific function and the resources available in a specific region of the FPGA.
- CLBs existing along the periphery of an AREA_RANGE constraint do not have all routing resources available to them because the BOUNDARYCROSS = NO constraint prohibits the use of any resource that can cross the boundary. Due to this routing restriction required to meet isolation requirements, unrouted conditions can occur. To resolve any unrouted conditions, the designer can use the PROHIBIT constraint to keep logic out of the peripheral CLBs. The PROHIBIT constraint prevents the placer from putting logic into the CLB. However, the CLB can still be used for routing, if needed.
- Redundant modules must have different entity names. Multiple instantiations of a module at the top level with the same entity name are not allowed.
- A BUFR must always be driven with a BUFIO. If the designer does not specifically instantiate a BUFIO, the connection between the pad and the BUFR will be made with fabric logic.
- Any tile that contains a GSM can be used as a fence (CLB, DSP, block RAM, I/O, etc.).
- “keep_hierarchy” must be applied to the first level of hierarchy below the top for any module that is to be isolated as a partition and not as a reconfigurable module. This is necessary so there is something to “bind” the area constraints to. This does not impact optimization any more that the traditional PR flow.
- Two ISO regions (partitions) can share configuration frames.
- One ISO region and one PR region (reconfigurable module) can share configuration frames.
- Two PR regions cannot share a configuration frame.

Definitions

- Physical ownership defines the partition where the component/logic physically resides in the FPGA.
- Physical placement requirements are defined by constraints (AREA_GROUP) in the UCF file.
- Logic can be logically owned (instantiated) at the top level, even if the logic is physically owned by a partition.

Addition of Components to an Area Group

All components (CLB, DSP, Block RAM, ILOGIC, OLOGIC, and PPC) used should be added to the area group that physically contains the component. All available resources should be included, even if the logic is not used, because excluding them also excludes using their respective routing resources. This includes clock components such as DCMs, PLLs, and BUFGs, even though such components are instantiated (logically owned) at the top level.

A component must be owned to be used. Top-level logic cannot use a component that is not owned by some area group. A component must be fully encompassed in an area group (physically, not logically) to be added to that area group (see Figure 16).

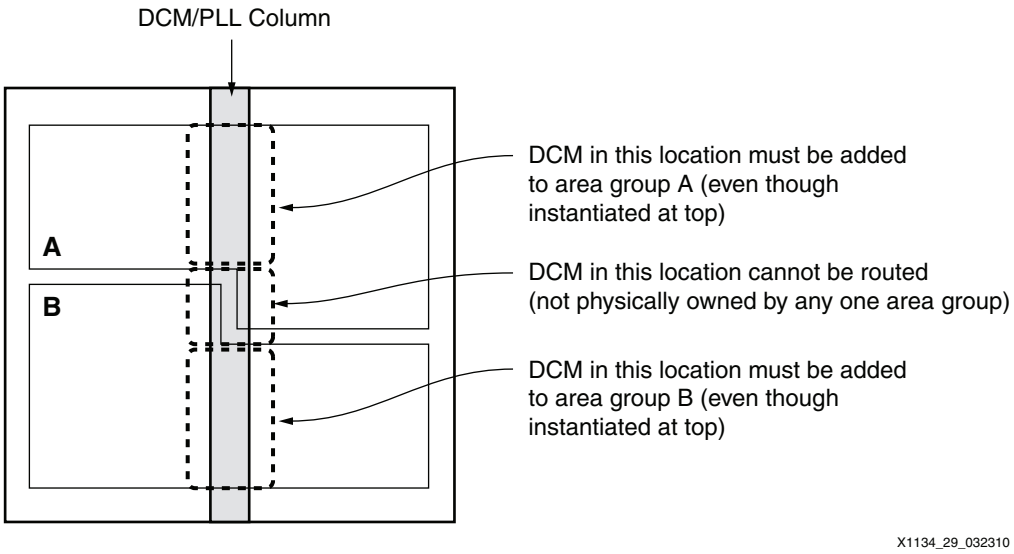


Figure 16: **Trusted Routing Guideline – Component Addition**

Area Group Attributes

All Isolated area groups (partitions) must have the following attributes:

- PRIVATE = NONE
  - AREA_GROUP “<ag name>” PRIVATE = NONE;

The following is a general example of the AREA_GROUP syntax in the UCF file:

```
SLICE
- AREA_GROUP “<ag name>” RANGE=SLICE_X1Y1:SLICE_X2Y2;
ILOGIC / OLOGIC
- AREA_GROUP “<ag name>” RANGE=ILOGIC_X1Y1:ILOGIC_X2Y2;
- AREA_GROUP “<ag name>” RANGE=OLOGIC_X1Y1:OLOGIC_X2Y2;
BRAM
- AREA_GROUP “<ag name>” RANGE=RAMB16_X1Y1:RAMB16_X2Y2;
- AREA_GROUP “<ag name>” RANGE=RAMB8_X1Y1:RAMB8_X2Y2;
DSP
- AREA_GROUP “<ag name>” RANGE=DSP48_X1Y1:DSP48_X2Y2;
```

```
IO Pins
- AREA_GROUP "<ag name>" RANGE=PADX, PADY, PADZ;
--Actual ranges are not supported. Each pad must be declared
BUFG
- AREA_GROUP "<ag name>" RANGE=BUFGMUX_X1Y1:BUFGMUX_X2Y2;
DCM / PLL
- AREA_GROUP "<ag name>" RANGE=DCM_X1Y1:DCM_X2Y2;
- AREA_GROUP "<ag name>" RANGE=PLL_ADV_X1Y1:PLL_ADV_X2Y2;
```

Device Ordering Instructions

Contact your local Xilinx representative for more information.

Reference Design Files

The design files for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=343389

The trusted routing design files (including the instantiation templates), files, and simulation models are available on the Isolation Design Flow page. The design checklist in Table 5 includes simulation, implementation, and hardware details for the reference design.

Table 5: **Design Checklist**

Parameter	Description
General	
Developer Name	Xilinx
Target devices (stepping level, ES, production, speed grades)	Virtex-5 LX
Source code provided	Y
Source code format	VHDL
Simulation	
Functional simulation performed	Y (At the module level only)
Timing simulation performed	Y (At the top level only)
Testbench used for functional and timing simulations	Y
Testbench format	VHDL
Simulator software/ version used	ISE tools, version 11.4(1)
SPICE/IBIS simulations	N
Implementation	
Synthesis software tools/version used	XST
Implementation software tools/versions used	ISE tools, version 11.4(1)
Static timing analysis performed	Y
Hardware Verification	
Hardware verified	N
Hardware platform used for verification	N/A

Notes:

1. It is necessary to use the ISE tools, version 11.4 for this lab.

Conclusion

The single-chip crypto design flow developed by Xilinx enables the information assurance industry to maximize the advantages of programmable logic. Using the SCC design flow, Isolation Verification Tool and Security Monitor IP outlined in this application note, Virtex-5 and Virtex-5Q FPGAs can be used to support redundancy, red/black data, and multiple levels of security on a single chip.

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
05/11/09	1.0	Initial Xilinx release.
04/19/10	1.1	Updated document for ISE tools, version 11.4. Added information and rules for trusted routing.
09/29/10	1.2	Updated Common Terminology, Communicating with Isolated Functions, On-Chip Communication, and Trusted Routing Rules.
08/30/11	1.2.1	Removed EAR marking. Updated disclaimer.
06/19/13	1.2.2	In Reference Design Files, added URL to download reference design and replaced SCC lounge with IDF page.
09/08/15	1.3	Removed sentence saying that Security Monitor IP is export controlled from Summary.

Notice of Disclaimer