XAPP1145 (v1.2) September 8, 2015

Developing Secure Designs with the Spartan-6 Family Using the Isolation Design Flow

Author: Steve McNeil

Summary

This application note is written for FPGA designers wishing to implement security or safety-critical designs, such as information assurance (single chip cryptography [SCC]), avionics, automotive, and industrial applications, using the Xilinx® isolation design flow (IDF). This document explains how to:

- Implement isolated functions in a single Spartan®-6 or Spartan-6Q (defense grade) FPGA using the IDF, e.g., red/black logic, redundant Type-I encryption modules, or logic processing multiple levels of security.
- Verify the isolation using the Xilinx Isolation Verification Tool (IVT).

With this application note, designers can develop a fail-safe single-chip solution using the IDF that meets fail-safe and physical security requirements for high-grade, high-assurance applications. To add additional security to the design, the Security Monitor IP, developed by Xilinx, can be purchased. To embed this IP, modifications to the steps in this document must be made as described in XAPP1152, *Security Monitor 2.0 Integration and Verification for Spartan-6 FPGAs*. Contact your local Xilinx representative for more information.

If the target application is a Type 1 cryptographic application (i.e., SCC applications), defense grade (XQ) devices must be selected.

Introduction

The flexibility of programmable logic affords the security and safety-critical industries 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 IDF was developed to allow independent functions to operate on a single chip (SCC). Examples of such SCC applications include, but are not limited to, redundant Type-I cryptographic modules, 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 and to provide safety-critical functions in avionics, automotive, and industrial applications.

Isolation Design Flow

Developing a safe and secure single-chip solution containing multiple isolated functions in a single FPGA is made possible through Xilinx partition technology along with a partial reconfiguration (PR) license. While the Spartan-6 FPGA does not support PR, the license and the features it enables are necessary to provide controls to achieve the isolation needed to meet certifying agency requirements. To better understand the details of the IDF, the designer should have a solid understanding of the standard partition design flow. Many of the terms and processes in the partition flow are utilized in the IDF. Areas that are different supersede the partition 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, e.g., an AES encryptor.
Logic	Circuits used to implement a specific function, for example, a flip-flop (FF), lookup table (LUT), block RAM, etc.
Region/ Area Group/ pblock	A physical area configured to implement logic.
Partition	A user-defined logical construct that can be used to isolate one piece of hierarchy from another.

Isolation Design Flow

A secure or safety-critical solution can be achieved while preserving FPGA design techniques and coding styles with only moderate modifications to the development flow. IDF 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 blocks (IOB). In addition to early floorplanning, the development flow is partition based (i.e., each function a user desires to isolate must be at its own level of hierarchy). From here, one of two approaches can be taken. To ensure that unwanted optimization of redundancy does not occur, each isolated function must be synthesized and implemented independently of the other partitions. After each partition is implemented, the design is merged into a flattened FPGA design for device configuration. To use other techniques to prevent such optimization, the designer can synthesize the full design while being careful to maintain at least one level of hierarchy such that IDF constraints can be applied to each partition that requires isolation. While this flow requires the FPGA designer to break away from traditional FPGA development flows, the partition approach does have certain advantages. If an isolated partition requires a change late in the design cycle, only that specific function is modified while the remaining partitions remain unchanged.

Note: All logic should belong to an isolated partition except for global clocks, resets, and IOBs.

To achieve an FPGA based IDF solution, a few unique design details must be adhered to. 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:

- Each isolated function must be in its own partition
- Each partition must consist of a single module instantiation
- A fence must be used to separate isolated partitions within a single chip
- IOBs must be inferred or instantiated inside isolated partitions for proper isolation of the IOB
- 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 the ISE® tools using the partial reconfiguration licensed features, verified by IVT and provided to the designer as a reference.

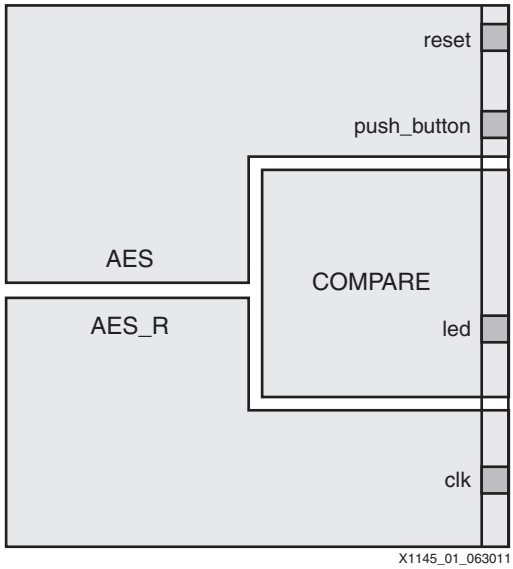


Figure 1: **An SCC Solution Using IDF Implementing Two Redundant AES Modules and Compare Logic**

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

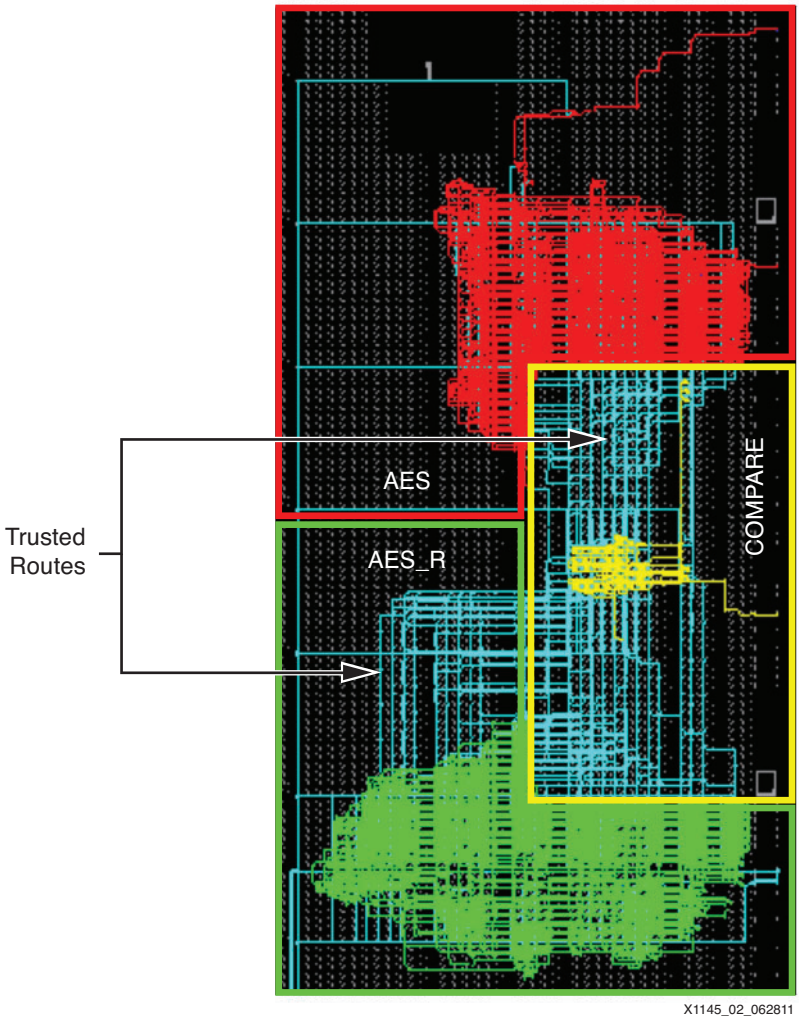


Figure 2: **Implemented SCC Solution from Figure 1**

For simplicity, the AES modules are 128-bit 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 AES modules 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 IOBs of each module.

Table 1: **Reference Design IOB for Each Isolated Module Including the Top Level**

Module	Module Inputs	Module Outputs
TOP	clk, reset, push_button	led
AES	clk, reset, push_button, start, mode, load, key, data_in	data_out, done, reset_out
AES_R	clk, reset, push_button, start, mode, load, key, data_in	data_out, done
COMPARE	clk, reset, done1, done2, data_in1, data_in2	reset_out, start_aes1, start_aes2, load_aes1, load_aes2, led

Per IDF details, the reference design implementation is performed using modular synthesis. Each function to be isolated has its own level of hierarchy and is implemented independently from the others. When completed, the independent modules are then merged together using

the PlanAhead™ tool during the import phase of the project generation. Figure 3 shows the hierarchy of the reference design.

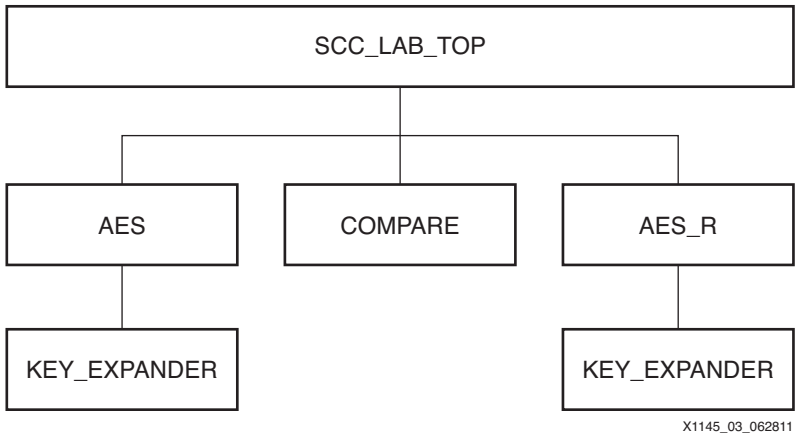


Figure 3: **Hierarchy for the SCC Implementation of the Reference Design**

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. As a companion to this reference design, XAPP1104, *Implementation of a Fail-Safe Design in the Spartan-6 Family Using ISE Design Suite 12.4* details step-by-step instructions on generating a similar reference design and can be found on the Isolation Design Flow page on Xilinx.com.

Isolation Fence

The complexity and density of the Spartan-6 or Spartan-6Q 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 13 members of the Spartan-6 family. However, Xilinx FPGAs are modular tile-based devices. No matter which device is chosen, the FPGA consists of the same basic building blocks tiled over and over again. In the Spartan-6 architecture, the basic building blocks are composed of CLBs (LX and MX), block RAM, DSPs, etc. Each tile is constructed of some hardened logic and one or more General Routing Matrix (GRM) blocks. Analysis of the FPGA is done at the tile level, and this is how isolation fences are determined (what tiles are valid and how many). A high-level view of this construction is shown in Figure 4.

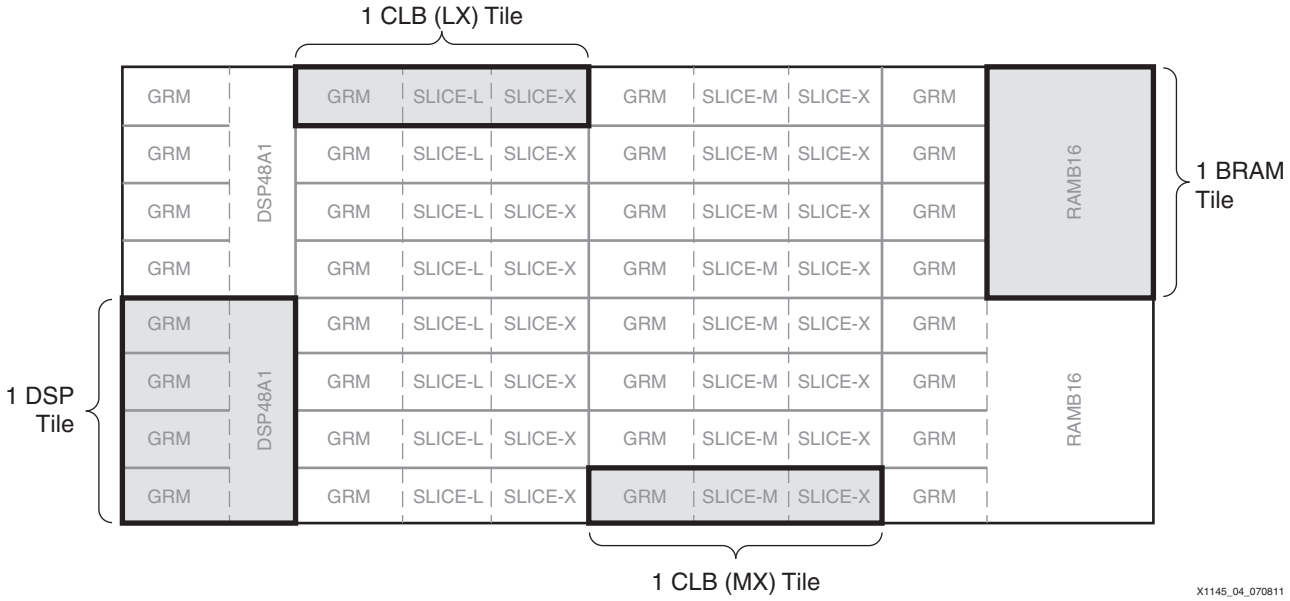


Figure 4: **Representation of the Tile-Based Spartan-6 Architecture**

To achieve isolation within a single device, the concept of a fence is introduced. The fence is just a set of unused tiles, as described above, in which no routing or logic is present. The results of the isolation analysis performed by Xilinx shows that one or sometimes two such tiles placed between isolated regions guarantee(s) that no single point of failure exists that would compromise the isolation between the two regions. An example of fence placement between two isolated modules is shown in Figure 5.

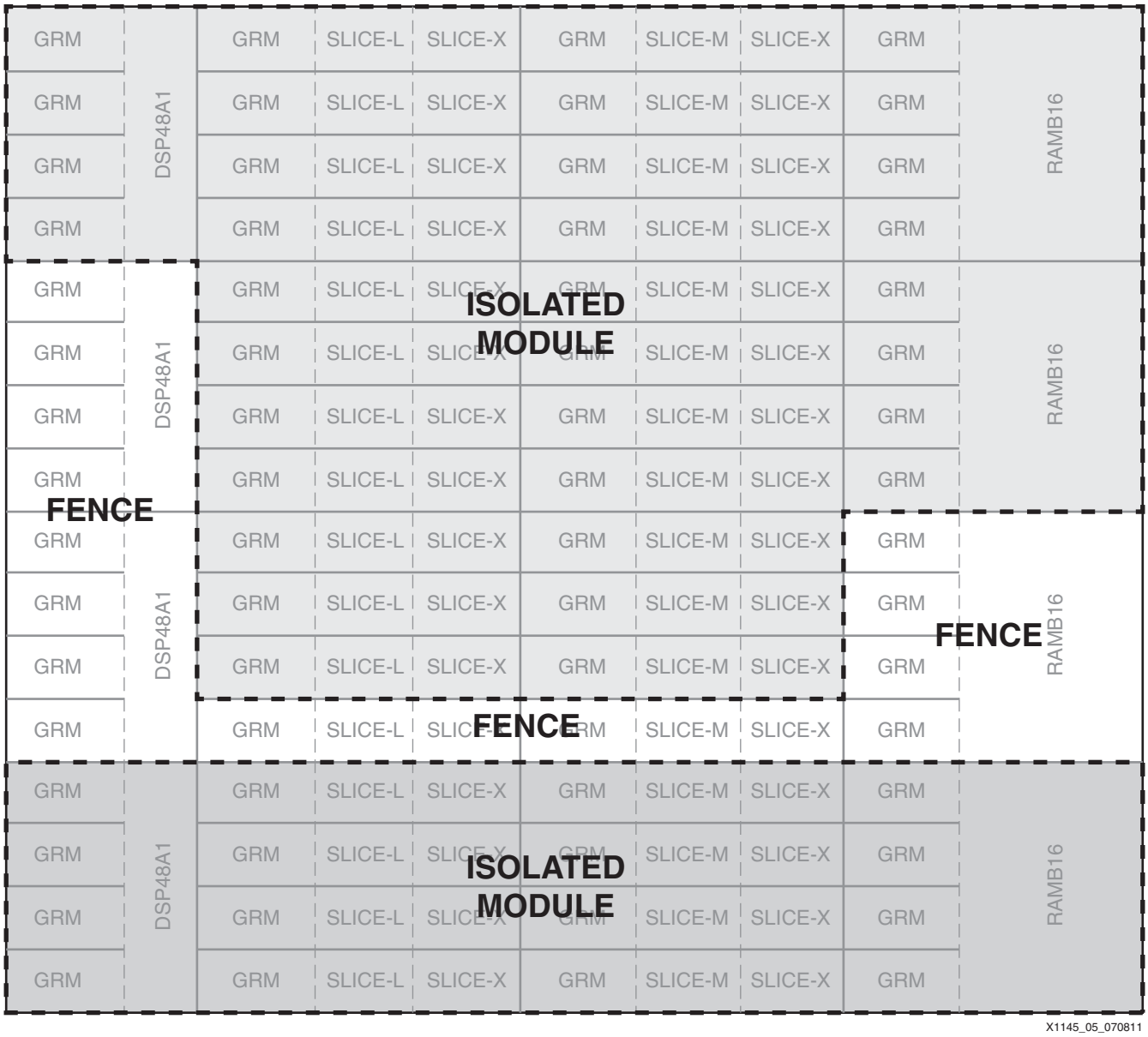


Figure 5: **Isolated Functions Separated by a Fence of Unused Logic Tiles**

The fence location is not directly specified by the designer. 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 encryption modules and the comparator block.

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 tile between the two intended regions. Refer to On-Chip Communication, page 14 for more details.

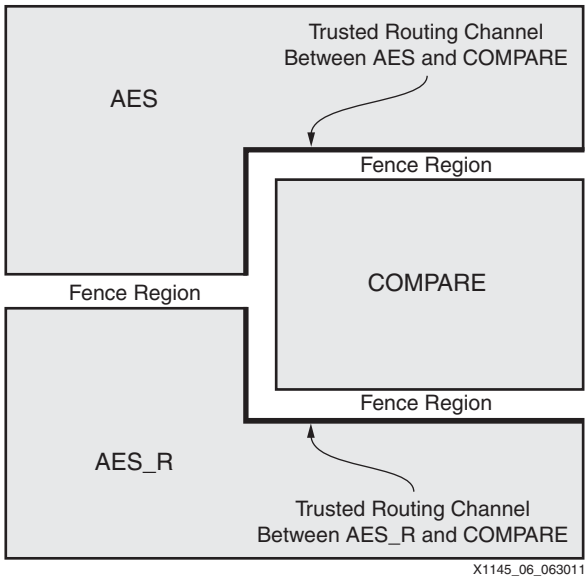


Figure 6: **Floorplan of the Reference Design**

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 encryption modules and comparator functions has 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 that enough dedicated resources exist. The dedicated resources can be IOB, FFs, LUTs, block RAM, DSP48E slices, etc. The PlanAhead tool, which relates AREA_GROUPs with physical blocks (PBLOCKS), 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, IODELAYs, BUFGMUXs, GTPs, DSP48s, 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
  - RAMB16
  - DSP48
  - 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_X72Y190:SLICE_X115Y191;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=DSP48_X3Y38:DSP48_X3Y47;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=RAMB16_X4Y76:RAMB16_X5Y94;`
- `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=PAD133,PAD158;`

Note: I/O pads are added to the area group range by a unique method. Pads must be added individually because they are not ranged sites.

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 a 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 design analysis tool. As an exception to this rule, the hardened memory controller block (MCB) should only be added to an area group if it is used in that region.

Communicating with Isolated Functions

Communication between isolated functions in the FPGA and other components within a system occurs through user IOBs. However, to maintain isolation from logic to the IOBs, and ultimately the PCB, the IOBs must be considered to be part of the isolation function and contained in the isolated region associated with that function. If the IOBs are not included in the isolated region, any routing from the isolated function to the IOBs cannot be contained to stay in the isolated region, and thus isolation cannot be guaranteed (Figure 7).

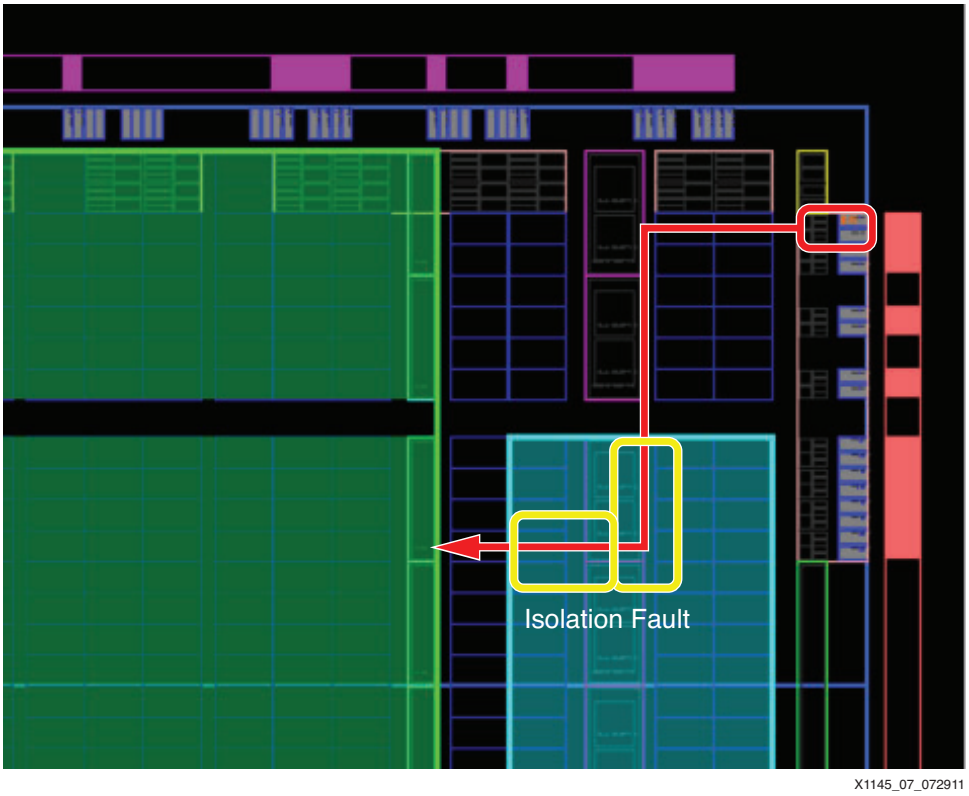


Figure 7: **An Uncontrolled Route from an IOB to an Isolated Region**

When the IOBs are included in the isolated region, the routing is controlled and isolation is guaranteed (Figure 8).

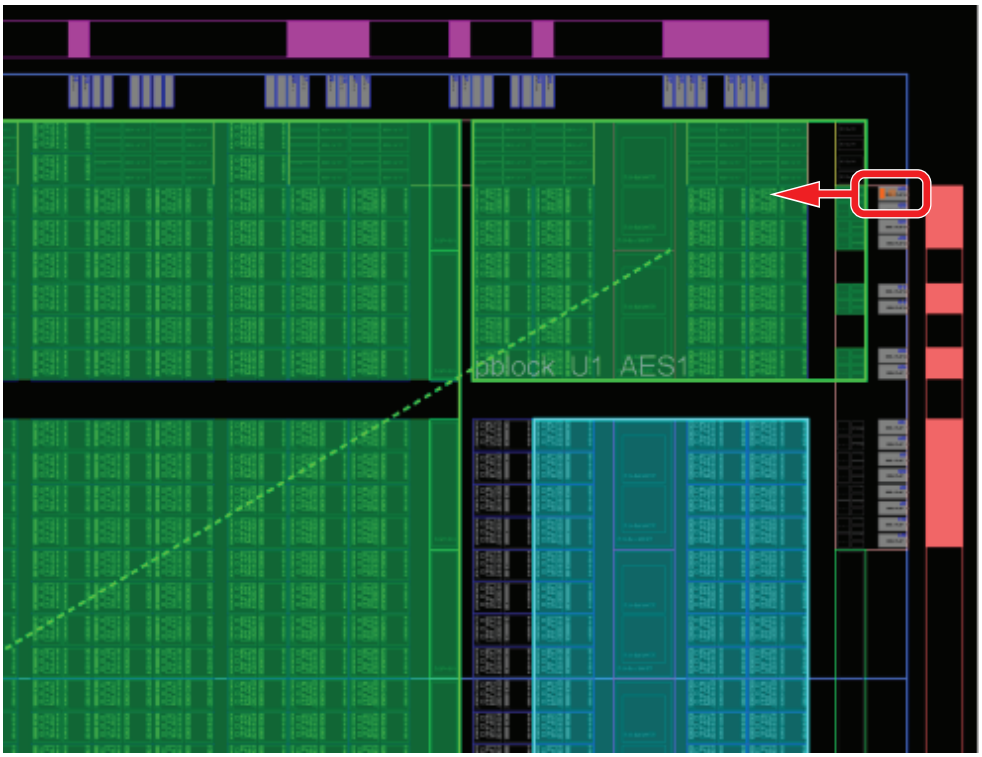


Figure 8: **Controlled Routing Between an IOB and its Isolated Region**

Communication between two isolated functions within the FPGA typically occurs internally using trusted routing. A less preferable alternative is to use IOBs 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.

Note: In the Spartan-6 architecture, the area group does not surround the actual IOB pad. Rather, it surrounds the logic associated with that pad (ILOGIC, OLOGIC). In addition to the area group surrounding the logic, the PAD_RANGE constraint must be set to add the pad itself to the area group.

Off-Chip Communication

If an isolated function has inputs or outputs that must come from or go off-chip, these signals must have their IOBs inferred inside an isolated partition. 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 IOB to the function. If the IOB is not part of the isolated partition, there is no control over how the signal is routed from the IOB to the logic. Standard design practice is to define all of the FPGA IOBs in the top-level port map (for VHDL users). During synthesis, all of the IOBs are inferred at the highest level of the target hierarchy. However, in the IDF, IOB insertion is specifically controlled and usually happens at the lower level partitions. For simulation, all of the IOBs are still defined at the top level in the IDF. However, where the IOBs are inferred depends on whether or not the signal must be isolated. For the IDF, the only signals that should have IOB 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 IOBs inferred at the lower level modules. Because the IDF 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. Because an isolated function might not be at the top level, a method must be employed to identify which signals the development tools should add IOBs to and which it

should not. A simple Xilinx Synthesis Tool (XST) attribute statement is used to keep IOBs from being inferred:

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

Note: All IOBs 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 IOBs. 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 IOB inserted.

Example syntax for the top-level HDL, `SCC_LAB_TOP.vhd`, is:

```
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;
```

Note: The line commented out means that no such attribute is applied. This is to show that clk pin is part of this level of hierarchy and needs to get its IOB inferred here.

To infer only the AES1 dedicated off-chip IOBs in the `aes` module, the attribute is applied to all signals that are not meant to be external IOBs. Example syntax for the aes HDL, `aes.vhd`, is:

```
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
      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 IOBs that go off-chip are the signals push_button and reset. All of the other signals do not have IOB 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 AES2 dedicated off-chip IOBs in the `aes_r` module, the attribute is applied to all signals that are not meant to be external IOBs. Example syntax for the aes_r HDL, `aes_r.vhd`, is:

```
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
      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 IOBs. Therefore, the attribute is applied to all of the signals.

To infer only the COMPARE dedicated off-chip IOBs in the `compare` module, the attribute is applied to all signals that are not meant to be external IOBs. Example syntax for the compare HDL, `compare.vhd`, is:

```
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
);
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 IOB that goes off-chip is the signal led. All other signals do not have IOB buffers inferred because they are either inferred elsewhere (for example, in the top-level hierarchy) or are not required because the signals are internal only.

In summary, isolation from the logic, through the IOB and onto the PCB, can be achieved by using the buffer_type attribute statement. This attribute dictates which signals have IOB 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). For details on the user rules required when using Trusted routing, refer to Trusted Routing Rules, page 25.
- 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. While possible, this method is not preferred due to both complexity and practicality.

Constraining Placement

Placement constraint is now achieved by the addition of components to the area group. Examples of this syntax with respect to the AES1 isolated function are listed below. A full listing of this syntax can be found in UG625, *Xilinx Constraints Guide*.

- `AREA_GROUP "pblock_U1_AES1" RANGE=DSP48_X3Y38:DSP48_X3Y47;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=PAD133,PAD158;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=RAMB8_X4Y76:RAMB8_X5Y95;`
- `AREA_GROUP "pblock_U1_AES1" RANGE=SLICE_X72Y190:SLICE_X115Y191;`

Components and resources 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 is a directive to the top-level function that allows top-level routes to use components owned by each area group. In the included reference design, the PRIVATE constraint is set to NONE for each isolated function.

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

Note: NONE is the default setting for the PRIVATE constraint. Even if not specifically set, this is the presumed value.

Constraining Routing

Routing control is no longer controlled using constraints. Rather, routing is controlled using physical ownership. No resource can be controlled if it is not physically owned (included in an area group statement). Global logic (DCM, PLL, BUFG, etc.) can use any resource as long as it is physically owned. Local logic (logic and routing of an isolated function) can only use resources physically owned by its own isolated region.

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 Spartan-6 or Spartan-6Q FPGA. The definition of whether a function is reconfigurable or simply isolated is no longer contained in the user constraint file (UCF). It is determined in the PlanAhead tool by setting the module as a partition and then attaching the SCC_ISOLATED attribute to it.

All constraints applied to the compare function are shown below.

```
INST "U3_Comp" AREA_GROUP = "pblock_U3_Comp";
AREA_GROUP "pblock_U3_Comp" RANGE=SLICE_X104Y105:SLICE_X127Y150;
AREA_GROUP "pblock_U3_Comp" RANGE=DSP48_X3Y28:DSP48_X3Y35;
AREA_GROUP "pblock_U3_Comp" RANGE=ILOGIC_X35Y102:ILOGIC_X35Y133;
```

```
AREA_GROUP "pblock_U3_Comp" RANGE=IODELAY_X35Y102:IODELAY_X35Y133;
AREA_GROUP "pblock_U3_Comp" RANGE=OLOGIC_X35Y102:OLOGIC_X35Y133;
AREA_GROUP "pblock_U3_Comp" RANGE=RAMB16_X4Y54:RAMB16_X5Y72;
AREA_GROUP "pblock_U3_Comp" RANGE=RAMB8_X4Y54:RAMB8_X5Y73;
AREA_GROUP "pblock_U3_Comp" PRIVATE=NONE;
AREA_GROUP "pblock_U3_Comp" RANGE=PAD190;
```

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 but highly recommended. The goal of the UCF checking is to identify potential isolation problems before commitment to board layout. 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, Windows 7, 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.

Constraint Checking (IVT - UCF Mode)

IVT, in UCF mode, checks the following:

- Pins from different isolation groups are not physically adjacent, vertically or horizontally, at the die.
- 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 IOB bank.
  - While IVT does fault such conditions, only specific security related applications require such bank isolation. The majority of applications allow for sharing of banks. Bank sharing is dependent on the specific application.
- The AREA_RANGE constraints in the UCF are defined such that a minimum of a one tile-wide fence exists between isolated regions.

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
```

Available command-line arguments for IVT operation on the UCF are shown in Table 2.

Table 2: **IVT Command-Line Arguments for UCF Mode**

IVT Command Line Argument	Description
-device *device_name*	Specifies the device.
-package *package_name*	Specifies the package.
-group *isolation_group area_group*	Associates the area group name to an arbitrary isolation group name. At least two distinct isolation groups are required.
[-pig *pin_isolation_groups*[.pig]]	If omitted, no pin-related analysis is performed.
[-output *output*[.rpt]]	Name of the output file.

Table 2: **IVT Command-Line Arguments for UCF Mode *(Cont'd)***

IVT Command Line Argument	Description
[-verbose]	Writes out all IVT reporting information.
[-f *argument_file*]	Command-line arguments are stored in an external file.
constraint_file[`.ucf`]	Constraint file for the design (UCF).

Notes:

1. Optional arguments are in brackets "[ ]". Italics denote variables.

IVT UCF Input Files

IVT, in UCF mode, requires two input files but accepts three, with the third one being a parameter file to reduce the length of the command line. From these inputs, IVT outputs two unique reports. The text report is the official output for use in an evaluation while the second, a graphical output, is useful for quick location of any isolation faults. Figure 9 represents a block diagram of the inputs and outputs of IVT while in UCF mode.

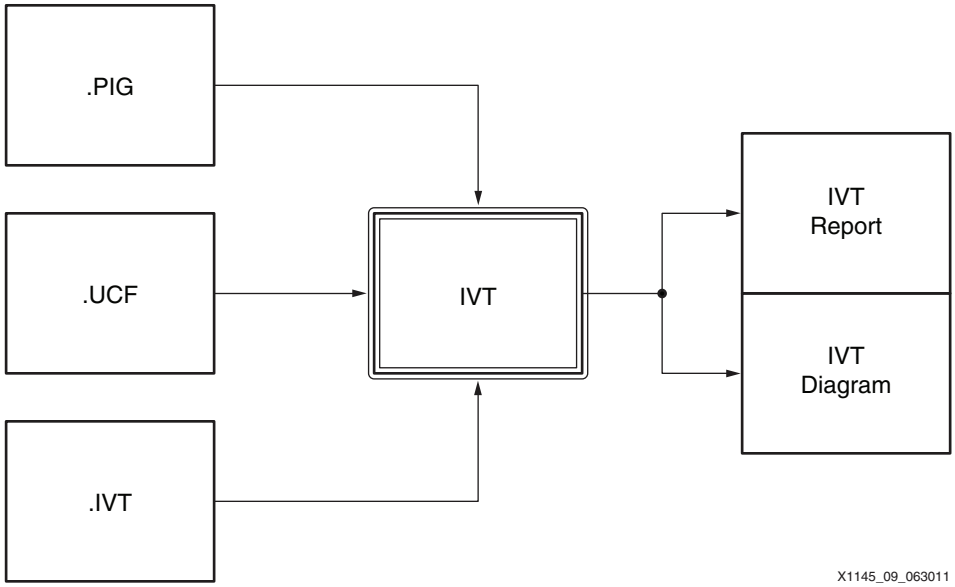


Figure 9: **IVT Inputs and Outputs in UCF Mode**

User Constraint File

The UCF is the floorplan of the user design.

Parameter File

The command-line arguments for the parameter file (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). The UCF mode parameter file for the reference design is shown here.

```
-device xc6slx150t -package fgg676

# 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
../PlanAhead/FloorPlan_SCC/FloorPlan_SCC.runs/config_1/SCC_LAB_TOP.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 rule 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 this syntax:

```
ISOLATION_GROUP group_name BEGIN
.
. #Cut and paste IOB location
. #constaints from UCF here
.
END ISOLATION_GROUP
```

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

Below is an example PIG file from the reference design.

```
# Place all Global (top level) signals here (each commented out)
# NET "clk"          LOC = U25;

ISOLATION_GROUP AES BEGIN
NET "push_button"    LOC = N25;
NET "reset"          LOC = M21;
END ISOLATION_GROUP

ISOLATION_GROUP AES_r BEGIN
# There are no pins in AES_r
END ISOLATION_GROUP

ISOLATION_GROUP COMPARE BEGIN
NET "led"            LOC = V24;
END ISOLATION_GROUP
```

Note: The net syntax is a copy/paste from the actual UCF file. The clock net is commented out because it is a top-level net that is not associated with any specific isolated region. The other IOB declarations are embedded in their appropriate ISOLATION_GROUP.

Final Isolation Verification (IVT - NCD Mode)

IVT, in NCD mode, checks the following:

- Pins from different isolation groups are not physically adjacent at the die. Adjacency is defined vertically, in either a north or south direction. Because the I/Os in Spartan-6 FPGAs are in a ring format, adjacency in the diagonal direction at the corners of the die is also checked.
- 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 fence consisting of empty user tiles, as defined in Table 4, page 28.

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
```

Available command-line arguments for IVT operation on the user design file (NCD) are shown in Table 3.

Table 3: **IVT Command-Line Arguments for NCD Mode**

IVT Command Line Argument	Description
-group *isolation_group user_instance_name*	Associates an arbitrary group name to an isolated instance in the user routed design. At least two distinct isolation groups are required.
-out *output*[`.rpt`]	Name of the output file.
-f *parameter_file*	Command-line arguments stored in an external file.
user_design[`.ncd`]	Fully routed user design file (NCD).

Notes:

1. Optional arguments are in brackets "[ ]". Italics denote variables.

IVT NCD Input Files

IVT, in NCD mode, requires one input file but accepts two, with the second one being a parameter file to reduce the length of the command line. From these inputs, IVT outputs two unique reports. The text report is the official output for use in an evaluation while the second, a graphical output, is useful for quick location of any isolation faults. Figure 10 represents a block diagram of the inputs and outputs of IVT while in NCD mode.

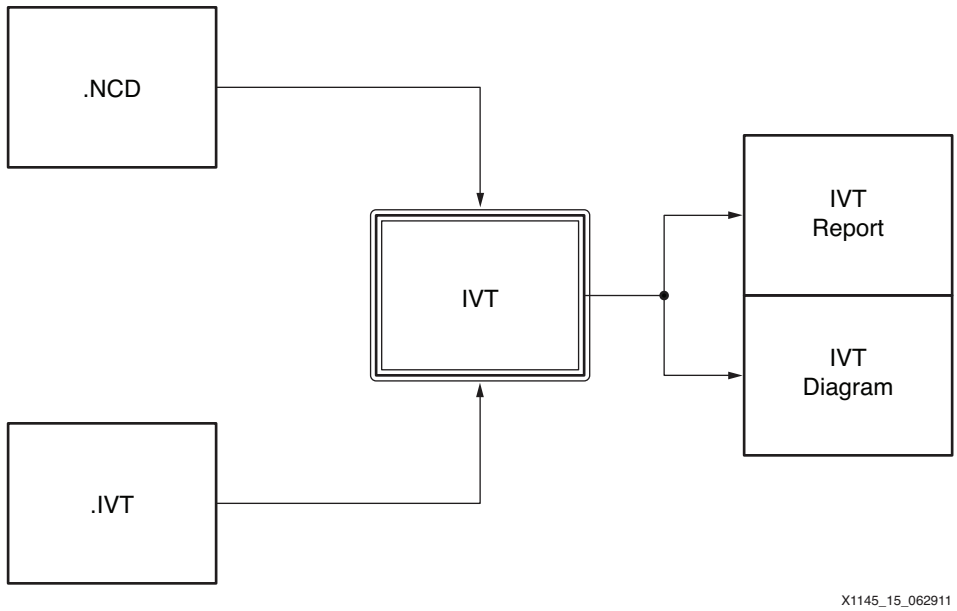


Figure 10: **IVT Inputs and Outputs in NCD Mode**

User Netlist

The user netlist, or NCD file, contains the fully routed user design.

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). The NCD mode parameter file for the reference design is shown here.

```
# comment the next line for reduced detail in the report file.
-verbose

# Groups    Isolation Group    Instance Name in Final NCD
#--------   ----------------   --------------------------
-group      AES                U1_AES1
-group      AES_r              U2_AES2
-group      COMPARE            U3_Comp

# Fully Routed design
SCC_LAB_TOP.routed.ncd

# Output Report File
-output SCC_LAB_TOP_ncd.rpt
```

IVT Output

As seen in Figure 9 and Figure 10, IVT has two outputs. The first, a graphical file, is useful as a high-level overview of the user design and is helpful in quick identification of isolation faults. The second, a text file, is a detailed report that is to be used in any official evaluation. Faults and other items of concern are described in this report. The content of these reports is dependent on the mode in which IVT was called (UCF or NCD).

UCF Mode

IVT, in UCF mode, verifies the user floorplan. Specifically, IVT ensures that user area groups have the required fence width between them and that the IOBs are properly isolated at both the die and package level.

Text Report

A typical UCF mode report contains these sections:

- Provenance: Includes the date, IVT version, ISE version used for the run, ISE version against which IVT was compiled, the location of the ISE installation, the command line, the current working directory, the output report location, and the part/package.
- Area range constraints: For each isolation group, lists the corresponding area groups and associated site ranges used to define the floorplan of the design.
- Package pins, I/O buffers, and I/O banks: The pin assignments of the design are presented with coordinates, I/O banks, isolation groups, and net names.
- Pin isolation summary: Lists the number of isolation violations for die pins (I/O buffers), package pins, and I/O banks.
- Area fault summary: Lists the number of area ranges from distinct isolation groups that are not separated by an adequate fence.
- Isolation verification summary: Lists the total number of constraints violated, reports completion, and reports the elapsed time to perform the analysis and generate the report.

A more detailed description and examples can be found in the IVT user guide provided with the reference design.

Graphical Report

For ease of use, IVT outputs a silicon vector graphics (SVG) file. This generic output allows for quickly identifying where faults are. While the text report is all-encompassing, the SVG file gives a high-level view of any faults and is useful in debugging the user floorplan. Figure 11 shows a typical example of the SVG output as run on the reference design.

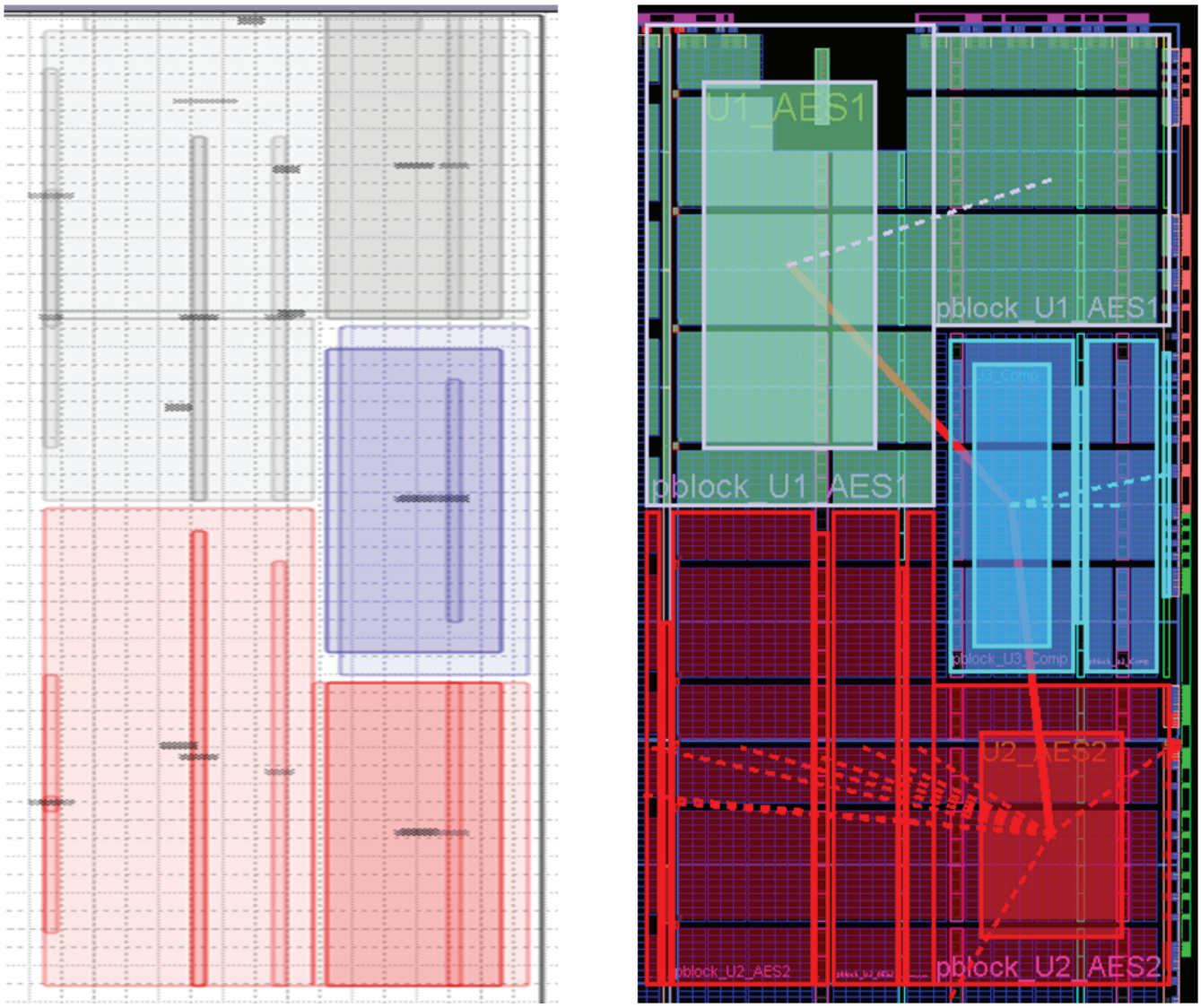


Figure 11: **SVG Output File (Left) Run on Reference Design Floorplan Next to Actual floorplan (Right)**

NCD Mode

IVT, in NCD mode, makes all the same checks it did in UCF mode but now works on the final routed design. Instead of looking at area group isolation, IVT now looks to make sure all the components and nets of each isolated module have a valid fence between them.

Text Report

A typical NCD mode report contains these sections:

- Provenance: Includes the date, IVT version, ISE version used for the run, ISE version against which IVT was compiled, location of the ISE installation, command line, current

working directory, output report location, part/package, and the name of the combined design.

- Isolated Modules: Lists the isolation groups and associated design blocks or partial NCD files.
- Uncategorized User Global Nets: Lists nets (signals) in the design that are above the isolated modules in the design hierarchy and might connect isolated functions. All such nets must be examined for their impact on the data separation and independence requirements of the system. Ideally, the only nets listed in this section would be nets specifically intended to connect isolated functions. In practice, some global clock signals appear here as well. In the report, uncategorized global nets are said to be “found in multiple isolation groups.” This is an artifact of the original implementation of IVT in which multiple partial NCD files were used to specify isolation groups. Global resources are duplicated in all the partial NCD files.
- Categorized Nets: Has many possible subsections corresponding to various categories of nets that for one reason or another are benign with respect to isolation. Examples include constants, global clocks, trusted bus macros, and clocking inserted automatically to mitigate negative-bias temperature instabilities.
- Trusted Bus Macros: Lists all the instances of trusted bus macros in the design and the nets connected to them.
- Area Range Constraints: For each isolation group, lists the corresponding area groups and associated site ranges used to define the floorplan of the design.
- Net Fault-Cost Violations (Failing Paths): For Virtex®-4 FPGA designs, lists pairs of putatively isolated nets that cannot be shown to be sufficiently isolated with fault-cost-based routing search. Examples of low-cost paths between isolated nets are listed, and several customer designs have produced isolation violations. However, in all cases, the violations turned out to be due to insufficient information in the cost function, not an actual vulnerability.
- Tiles with Net Content Violations: Lists the contents of all tiles that contain isolated logic or routing from more than one isolation group.
- Tiles in the Fence Containing Nets: Lists the contents of tiles that are outside all the isolated area ranges. This list is advisory. It is permissible for inter-region signals and clocking to exist in the fence. Akin to the list of Uncategorized Global Nets, nets in tiles that are outside of all isolation groups must be vetted for their impact on data separation and independence requirements. This section provides low-level details of the specific nodes and wire segments used to implement routing in the fence.
- Tiles in the Fence Containing Programming: Lists tiles in the fence that are associated with components and therefore are not entirely unused. All such tiles must be examined for their impact on the data separation and independence requirements of the system.
- Tiles in the Fence Containing Used PIPs: Lists tiles containing programmable interconnect points (PIPs), which are nodes that have the potential to connect to other nodes. For example, suppose a certain type of node spans five horizontal tiles called A, B, C, D, and E, and that this node has PIPs in tiles A, C, and E. This node can be used to connect two isolated regions provided the only tiles in the fence are B or D.
- Tiles with Net Adjacency Violations: Lists all pairs of tiles that are adjacent and contain isolated logic or routing resources from distinct isolation groups, in other words, pairs of tiles that should be separated by a fence tile.
- Package Pins, I/O Buffers, and I/O Banks: The pin assignments of the design are presented with coordinates, I/O banks, isolation groups, and the net names.
- I/O Buffer Isolation Violations: Lists pairs of I/O buffers from distinct isolation groups that are adjacent on the die.
- I/O Bank Violations: Lists examples of pins from distinct isolation groups that are members of a single I/O bank.

- Package Pin Isolation Violations: Lists pairs of pins from distinct isolation groups that are adjacent on the package.
- Isolation Verification Summary: Lists the total number of constraints violated by category, reports completion, and reports the elapsed time to perform the analysis and generate the report. For each section of the report containing violations, there is a line in the summary with a tally of the violations in that section.

A more detailed description and examples can be found in the IVT user guide.

Graphical Report

For ease of use, IVT outputs an SVG file. This generic output allows for quickly identifying where faults are. While the text report is all encompassing, the SVG file gives a high-level view of any faults and is useful in debugging the user floorplan. Figure 12 shows a typical example of the SVG output as run on the reference design.

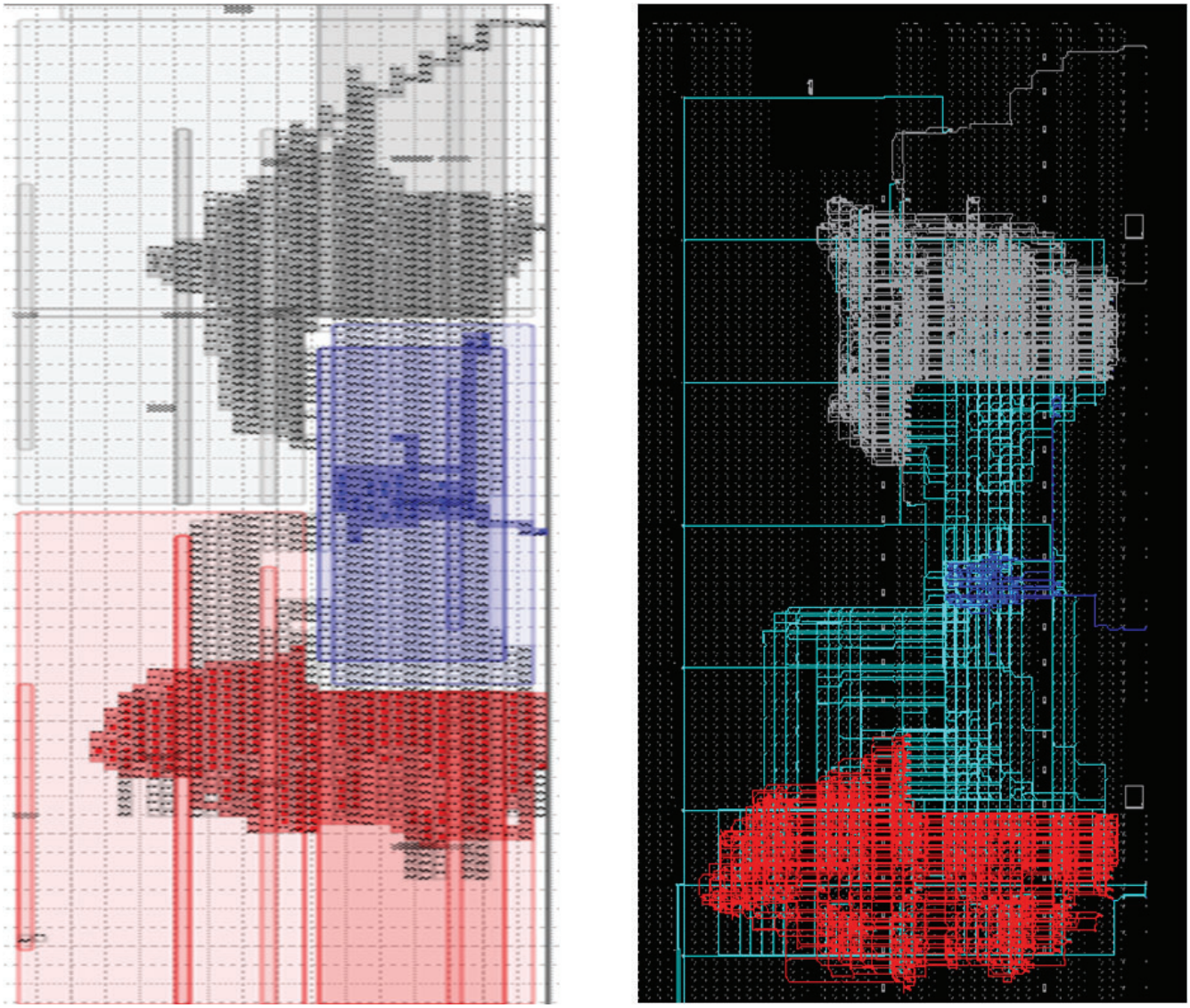


Figure 12: **IVT SVG (Left) Run on Reference Design/FPGA Editor View of Same Design (Right)**

Design Guidance

SCC and the Isolation Design Flow

SCC is built upon the IDF software flow. IDF, along with significant analysis performed by Xilinx, is what allows for SCC implementations. SCC is simply a single application using the IDF. IDF, however, is more generic in nature and is not that dissimilar from a traditional design flow. Figure 13 shows a flow diagram of both flows side by side.

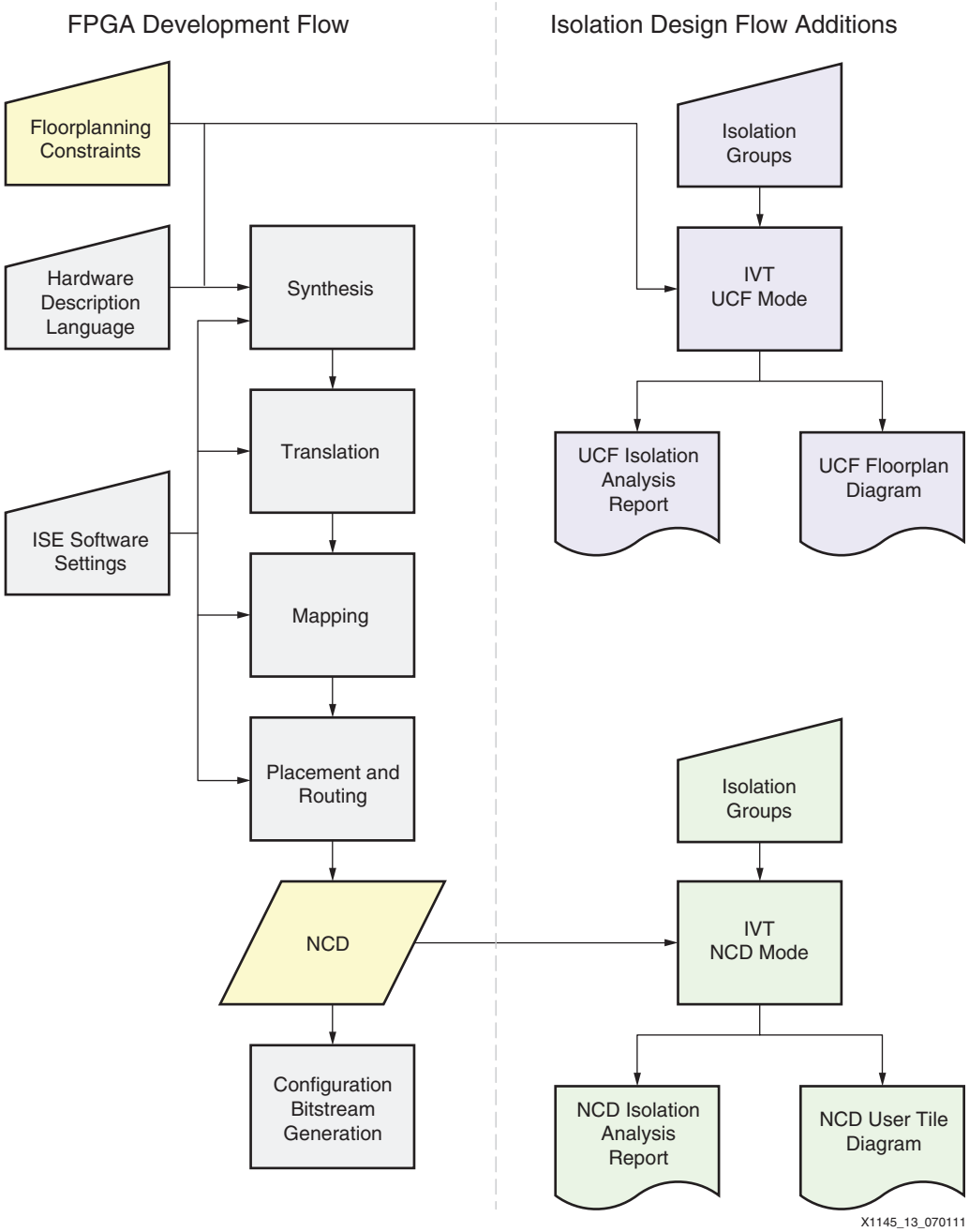


Figure 13: **The Isolation Design Flow Relative to a Typical FPGA Design Flow**

Partition Overview

All the partition definitions and controls are done in the `xpartition.pxml` file. This file can be created by hand 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. The ISE tools, versions 12.1 and later, are required.

To implement the `xpartition.pxml` file automatically using the PlanAhead tool, the new file is copied by the included implementation scripts of the reference design up to the top level of the project for future command-line use. The commands in the `xpartition.pxml` file are case sensitive.

Trusted Routing Design Guidelines

While trusted routing is automatic, there are a few rules and concepts the user must be aware of. The following sections touch on the concept of ownership and then detail the rules that the user must follow to ensure proper communication between isolated regions.

Concepts

- Logical ownership: Relates to HDL partition (right side of Figure 14):
  - Each HDL file logically owns the logic instantiated within it.
  - Only global logic (BUFG, DCM, PLL) allowed at the top logical level of design and the global logic cannot logically be placed in a partition.
  - Global logic must be logically owned by the top level of the design.
- Physical ownership: Relates to area groups/pblocks (left side of Figure 14):
  - No user logic component can be used if it is not physically owned by an isolated region (area group). Ownership is defined by addition to the area group statement.
  - No global logic component can be used if it is not physically "owned" by some isolation group.

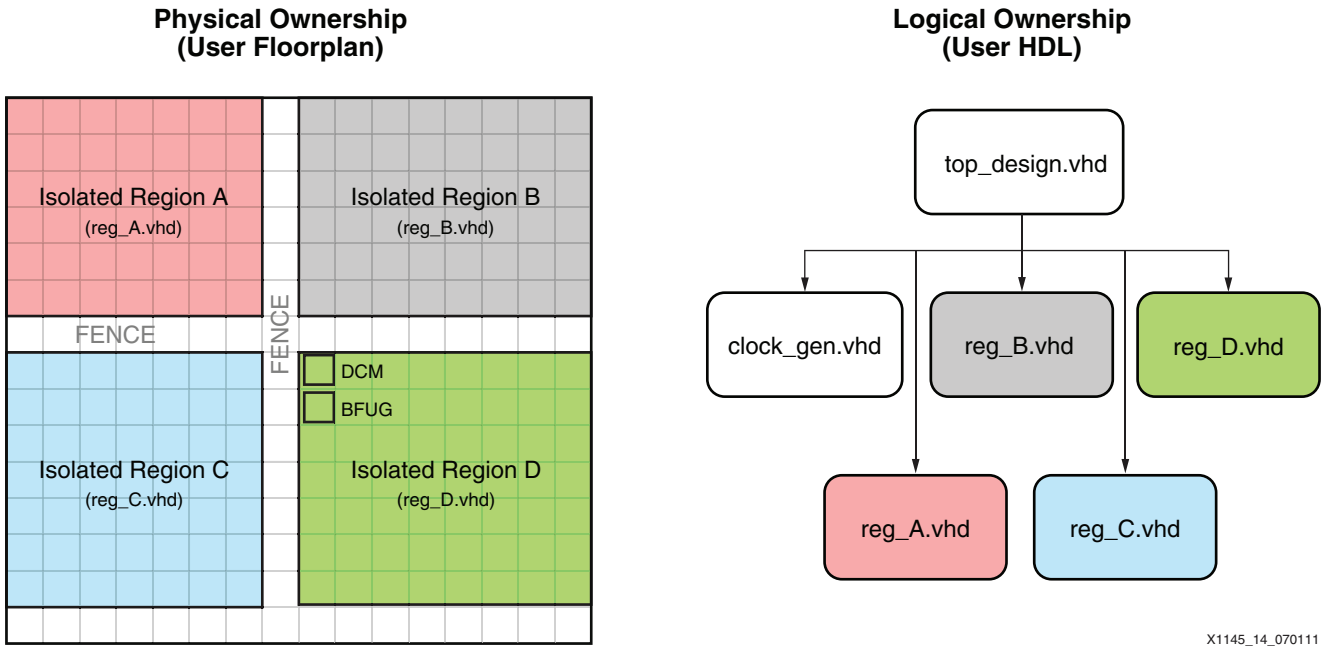


Figure 14: **Side by Side View of Ownership Concepts**

Note: The DCM and BUFG instances in Figure 14 are logic owned by `clock_gen.vhd`, because that is the level of HDL that instantiated the primitives, and physically owned by Isolated Region D because their physical sites are encompassed by the area group Isolated_Region_D.

Trusted Routing Rules

Trusted routing is automatic. The design tools recognize the communication between isolated regions and use the trusted routing resources. However, some rules must be adhered to if safe communication between isolated functions is to be guaranteed:

1. Feed-through signals are not allowed without buffering of some kind (LUT, FF, etc.):
   a. If a signal is directly connected to both an input port and an output port, it must be buffered.
   b. Direct instantiation of a buffer (LUT1, for example) is necessary. This isolates area groups, with the LUT preventing a shorted net.

   Problem: The statement “reset_out <= reset” violates IDF rules by creating a short between isolated area groups (Figure 15).

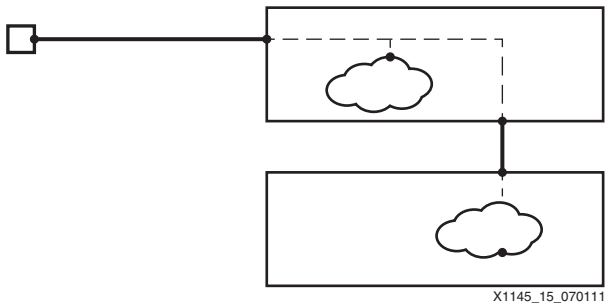


Figure 15: **“Short” Created by Feed-Trough Signal**

Problem: Feed-through signals need to be LUT buffered. This can be achieved either through HDL coding to ensure that there is some unique driver on the output port or by direct instantiation of a LUT buffer (Figure 16). The problematic code above can be fixed as follows:

```
lut_reset_out: LUT1
GENERIC MAP (INIT => X"2")
PORT MAP (I0 => reset, O => reset_out);
```

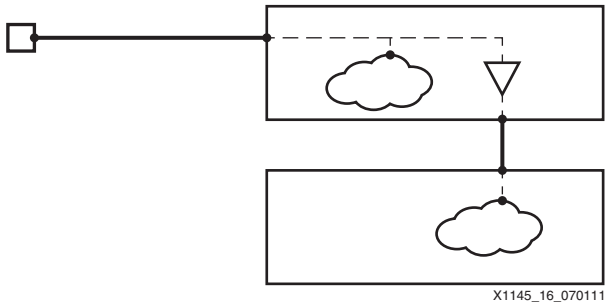


Figure 16: **Elimination of a Short by Instantiation of a LUT Buffer**

2. A function output port cannot connect to more than one function input port. Stated differently, port-to-port connections must be singular:
   a. The user must create two different ports for such a connection.
   b. Each port must not violate rule 3 below.

   Problem: The following segment of VHDL code creates a common connection between unintended isolated functions (Figure 17).

```
U3_Comp:  compare
  port map(
     clk          => clk_i,
     start        => start_i,
     load         => load_i,
     led          => led );

  U4:  aes1
  port map(
     clk          => clk_i,
     start        => start_i,
     load         => load_i);

  U5:  aes2
  port map(
     clk          => clk_i,
     start        => start_i,
     load         => load_i);
```

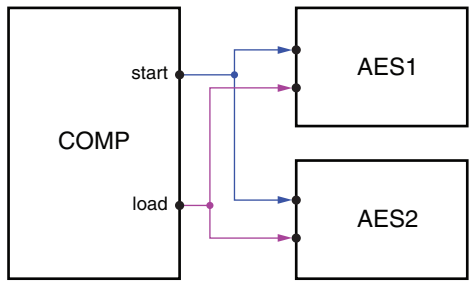


Figure 17: **Multi-Port Connection Causing Connectivity between AES1 and AES2**

Solution: The user must create multiple output ports if required to drive multiple input ports of other isolated functions (Figure 18).

```
U3_Comp:  compare
  port map(
     clk          => clk_i,
     start_aes1   =>
     start_aes1
     start_aes2   =>
     start_aes2,
     load_aes1    =>
     load_aes1,
     load_aes2    =>
     load_aes2,
     led          => led );
U4:  aes1
  port map(
     clk          => clk_i,
     start        =>
     start_aes1,
     load         =>
     load_aes1);
U5:  aes2
  port map(
     clk          => clk_i,
     start        =>
     start_aes2,
     load         =>
     load_aes2);
```

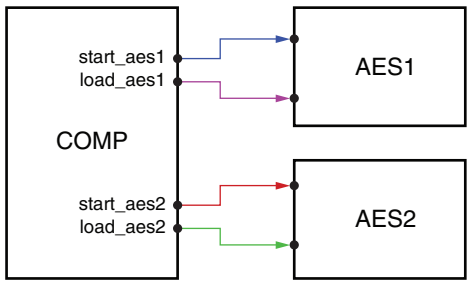


Figure 18: **Elimination of Unintended Connection Using Multiple Output Ports**

3. One signal cannot drive two different output ports of the same function:
   a. Each port must have a unique driver.
   b. Direct instantiation of a buffer (LUT1, for example) is necessary. This isolates area groups, with the LUT preventing a shorted net.

   Problem: The statements below create unintended connectivity between two isolated functions (Figure 19).

```
start_aes1 <= start_i;
start_aes2 <= start_i;
```

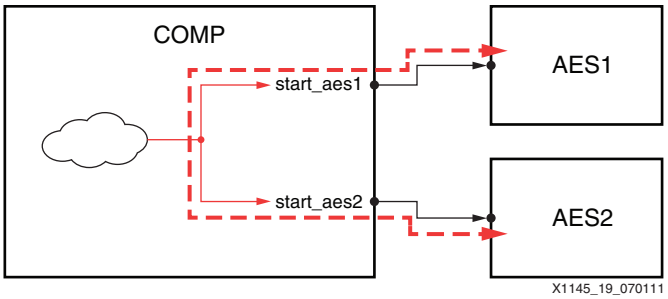


Figure 19: **Untended Connectivity between AES1 and AES2 inside COMP**

Solution: Each port driver needs to be LUT buffered. This can be achieved either through HDL coding to ensure that there is some unique driver for each output port, or by direct instantiation of a LUT buffer (Figure 20). The problematic code above can be fixed as follows:

```
lut_start_out: LUT1
GENERIC MAP (INIT => X"2")
PORT MAP (I0 => start_i, O => start_aes1);

lut_start_out: LUT1
GENERIC MAP (INIT => X"2")
PORT MAP (I0 => start_i, O => start_aes2);
```

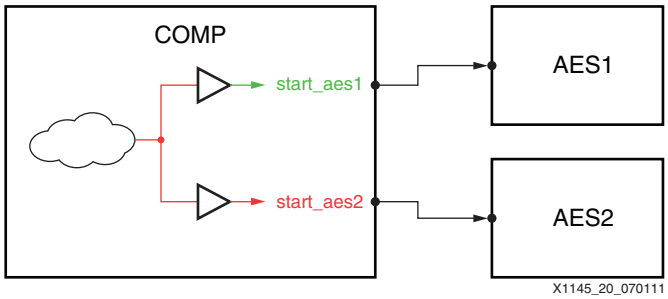


Figure 20: **Elimination of Unintended Connectivity Using LUT Buffers**

Spartan-6 FPGA Fencing Rules

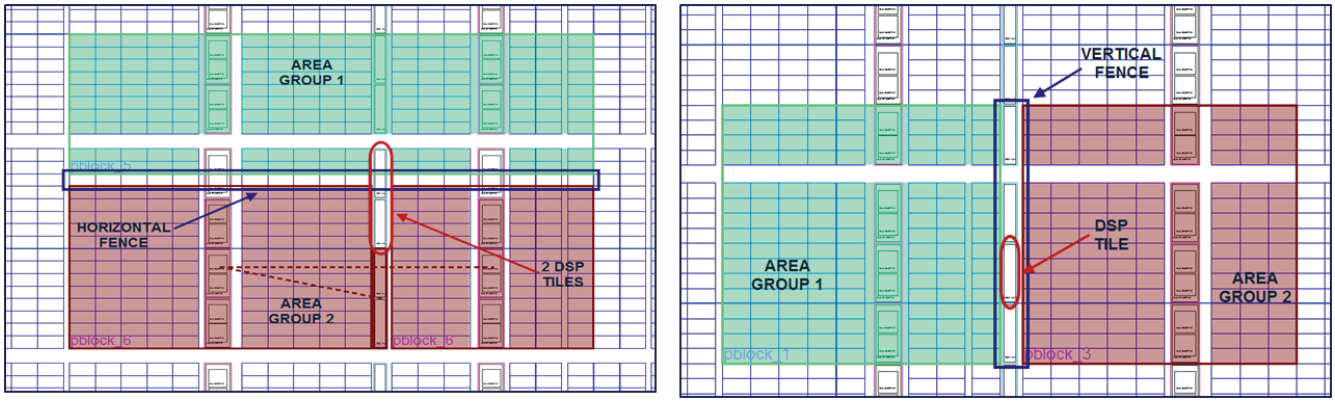
To achieve isolation between floorplanned regions, a fence is necessary to separate them. This fence is not specifically drawn so much as it is what is left over between drawn regions. This empty space must meet certain criteria to be considered a fence.

1. The fence tile must be empty. While the tools enforce this, manual edits can be made to violate this rule. Such edits cause a reduction in fault immunity and are identified by IVT as a failure.
2. The fence tile must be a valid one as listed in the table of user tiles with the appropriate number of tiles (Table 4).

Table 4: **Spartan-6 FPGA Fence Tile Requirements**

User Tile	Tile Description	Minimum Count for Vertical Fence	Minimum Count for Horizontal Fence
CLB	Configurable Logic Block	1	1
BRAM	Block RAM	1	1
DSP	Digital Signal Processor	1	2
MCB	Hardened Memory Controller	1	1
IOB/IOI	Input-Output Block (pair) Input-Output Logic	1	1
DCM	Digital Clock Manager (Global Clocking)	Not Allowed	1
PLL	Phase Locked Loop (Global Clocking)	Not Allowed	1
GTP	High Speed Transceiver	Not Allowed	Not Allowed
PCIE	PCI Express EndPoint	Not Allowed	Not Allowed

For clarity, those user tiles that have unique usage as fence tiles are shown in Figure 21 and Figure 22.

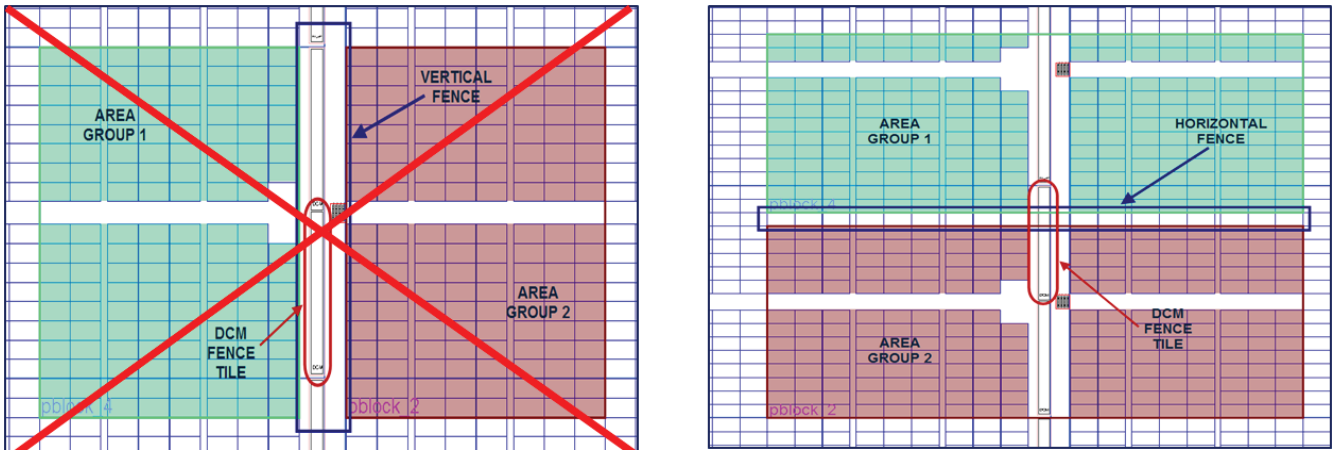


Valid DSP Usage in a Horizontal Fence

Valid DSP Usage in a Vertical Fence

X1145_20_062911

Figure 21: **DSP Usage in a Fence**



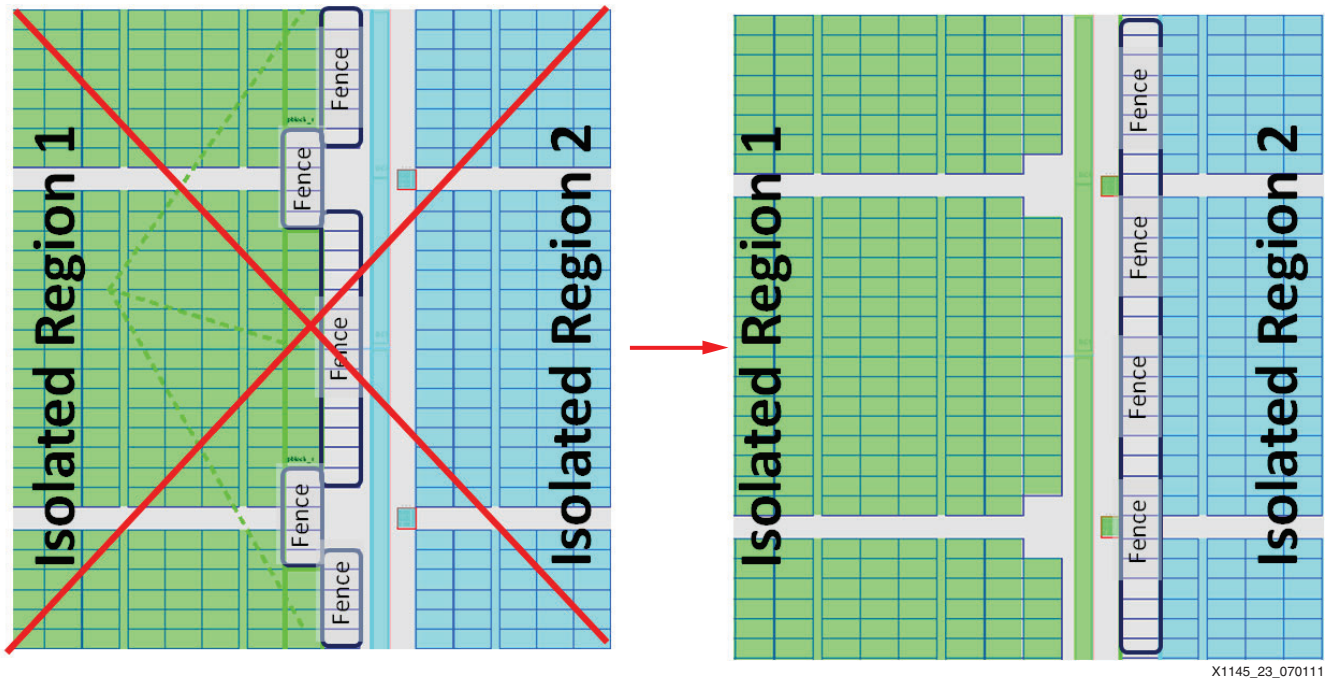
Invalid DCM/PLL Usage in a Horizontal Fence

Valid DCM/PLL Usage in a Horizontal Fence

X1145_22_070111

Figure 22: **DCM/PLL Usage in a Fence**

Additional care should be taken when placing a fence near the clock column that contains the DCM and PLL user tiles. The column of CLBs to the left contain necessary routing resources for the global clocking tree. As such, these CLBs should not be made part of a fence. Rather, if the fence needs to be in this area, the fence should be placed on the right side of the clock column (Figure 23).



X1145_23_070111

Figure 23: **Invalid and Valid Fence Placement Adjacent to Clock Column (DCM/PLL)**

Hints and Guidelines

There are various complexities to developing isolated designs using the IDF and the Spartan-6 family. To help the developer through some of the known complexities, various hints and guidelines have been developed and are listed below.

Spartan-6 FPGA I/O Ring Complexities

Due to a limitation in the way the ISE software calculates overlapping area group ranges, care must be taken when creating area group rectangles on the corners of the die. Wrapping I/O sites around the edge in a single range constraint creates an error where the “calculated” rectangle overlaps a neighboring area group. An example of this issue and how to correct it is shown in Figure 24.

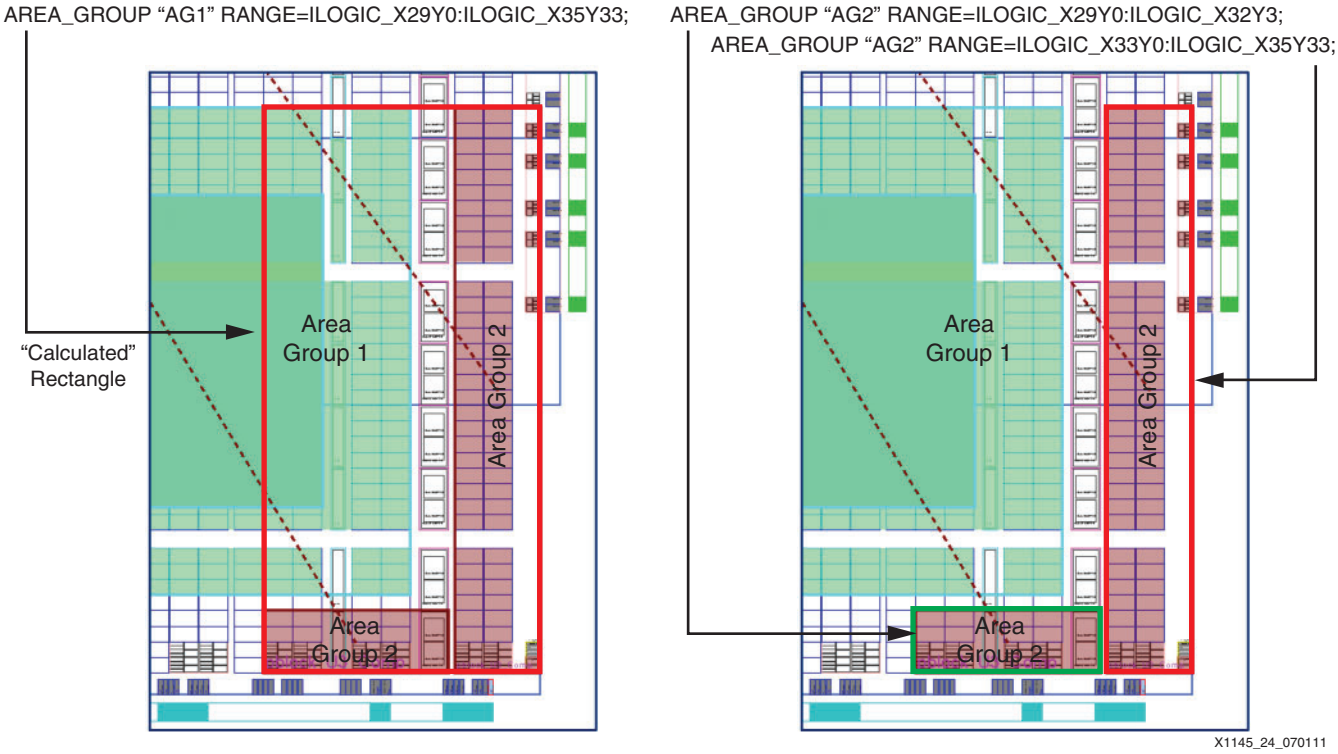


Figure 24: **Area Groups at Spartan-6 FPGA Corners**

Auto-Insertion of BUFIO2/BUFIO2FB at MAP

MAP adds BUFIO2FB and BUFIO2 components when a DCM or PLL instance is placed in a design without including them in the instantiation. This is required for proper DCM operation. However, the IDF requires that all top-level logic (global clocking) needs to be physically placed. An issue arises when MAP creates these components after the user floorplanning phase. To avoid this issue, all DCM and PLL instantiations should include these components so that they are available for placement at the floorplanning stage. Adding these sites in the user HDL eliminates this auto-insertion.

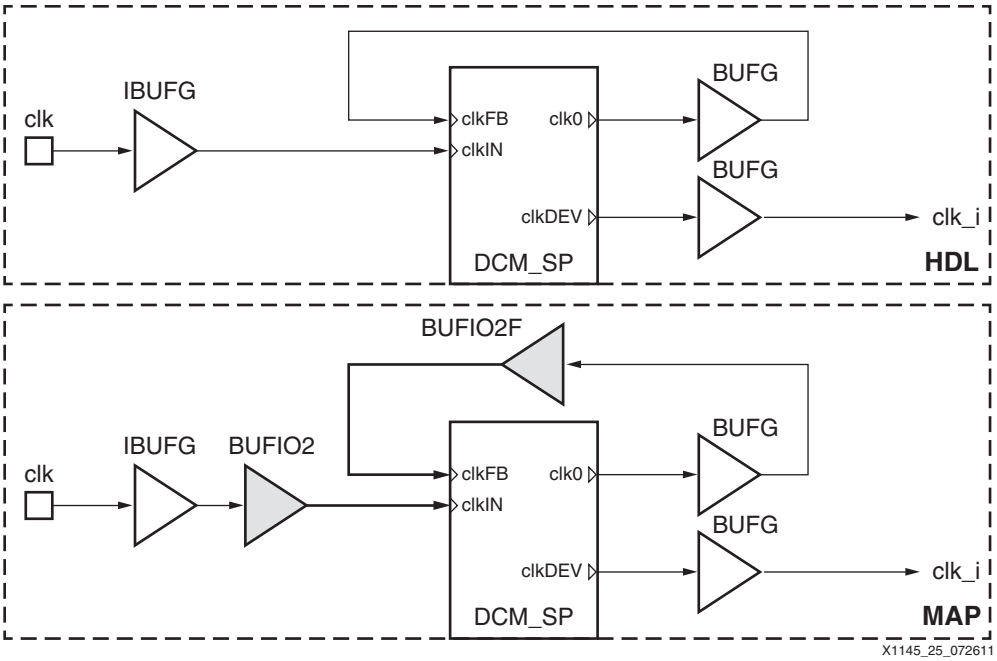


Figure 25: **Auto-Insertion of BUFIO2 and BUFIO2FB Components at MAP**

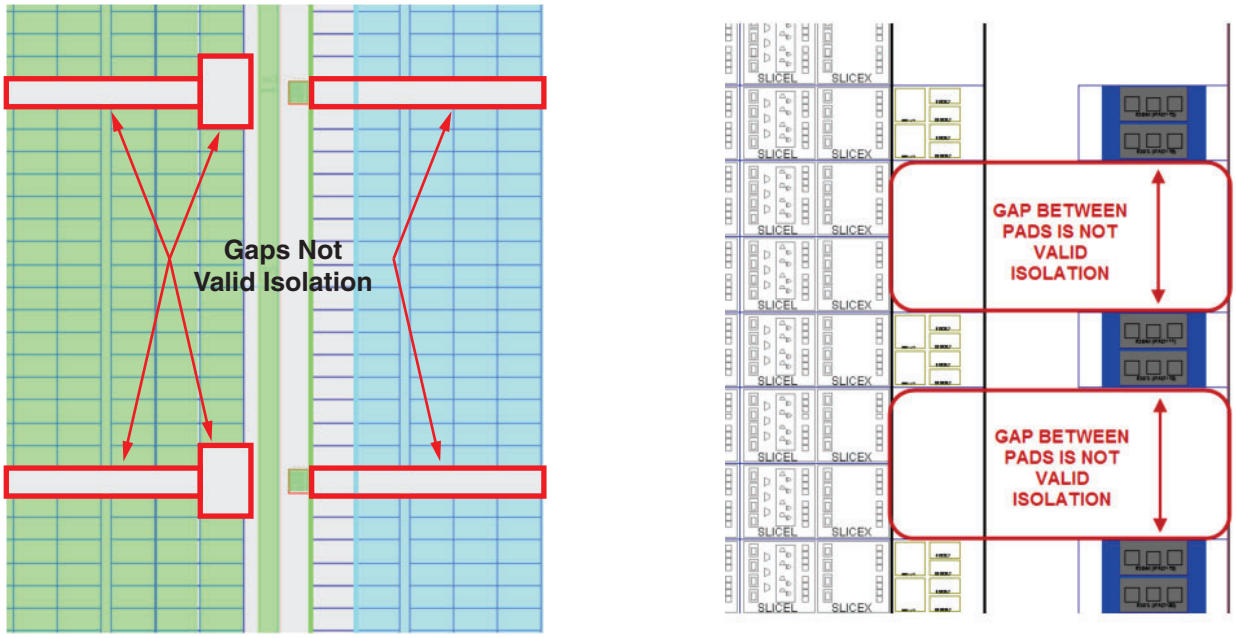
PAD to IOB/IOI Mapping

Pads in the Spartan-6 device come in pairs. A single IOI (IOB Logic Site) drives or receives information from a pair of IOB pads. An IOB pair cannot be separated and is driven by a very specific IOI site. The graphical depiction on the top and bottom of the Spartan-6 device gives the impression that the IOB and IOI are not contiguous. This is not correct. On the left and right, there is a connection between the IOB pads and the IOI directly adjacent to them. However, on the top and bottom, a little more clarity is necessary.

Figure 26: **Top and Bottom IOI/IOB Mapping**

Gaps in Floorplan

The user should be mindful of apparent gaps in the floorplan as seen by FPGA_EDITOR or PlanAhead. These gaps are an artifact of how the software model is depicted in the user GUI. Such gaps do not represent any form of isolation and cannot be used as a fence tile because the gaps are not a user tile at all. The most notable of these gaps are in the I/O ring and to the left of the clock column.



Gaps near Clock Column

Gaps in I/O Ring

Figure 27: **Example of Gaps in the Spartan-6 Device**

SCC Limitations for Spartan-6 Family

For SCC designs, the Spartan-6 family has the following limitations:

1. Designs must use defense-grade devices. The current offering is limited to the XC6SLX75 and the XC6SLX150. This Includes their “T” derivatives.
2. LUTRAM components are not allowed.
3. SRL16 components are not allowed.

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=343395

The trusted routing design files (including the instantiation templates), files, and simulation models are also 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: **Reference Design Checklist**

Parameter	Description
General	
Developer Name	Xilinx
Target devices (stepping level, ES, production, speed grades)	Spartan-6 LX Family
Source code provided	Y
Source code format	VHDL

Table 5: **Reference Design Checklist *(Cont'd)***

Parameter	Description
Simulation	
Functional simulation performed	Y (At the module level only)
Functional simulation performed	Y (At the top level only)
Test-bench used for functional and timing simulations	Y
Test-bench format	VHDL
Simulator software/version used	ISE tools, v12.4
SPICE/IBIS simulations	N
Implementation	
Synthesis software tools/version used	XST
Implementation software tools/versions used	ISE tools, v12.4
Static timing analysis performed	Y
Hardware Verification	
Hardware verified	N
Hardware platform used for verification	N/A

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
08/23/11	1.0	Initial Xilinx release.
10/14/11	1.1	Updated Final Isolation Verification (IVT - NCD Mode).
06/19/13	1.1.1	In Reference Design Files, added URL to download reference design and replaced SCC lounge with IDF page.
09/08/15	1.2	Removed sentence saying that Security Monitor IP and associated application note are export controlled from Summary.

Notice of Disclaimer

use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.