XAPP290 (v1.0) May 17, 2002

Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations

Author: Davin Lim and Mike Peattie

Summary

An important feature in the Xilinx Virtex™ architecture is the ability to reconfigure a portion of the FPGA while the remainder of the design is still operational. Partial reconfiguration is useful for applications that require the loading of different designs into the same area of the device or the flexibility to change portions of a design without having to either reset or completely reconfigure the entire device. With this capability, entirely new application areas become possible:

- In-the-field hardware upgrades and updates to remote sites
- Runtime reconfiguration
- Adaptive hardware algorithms
- Continuous service applications

Other benefits include:

- Reduced device count
- Reduced power consumption
- More efficient use of available board space

This application note describes the exact steps required to successfully design, implement, verify, and actively reconfigure portions of Virtex and Virtex-II series FPGAs. References to Virtex or Virtex-E families also apply to Spartan™-II or Spartan-IIE families. Two implementation flows are described in this application note: Module-based partial reconfiguration and a small-bit manipulation method of partial reconfiguration.

Shutdown partial reconfiguration, where the non-reconfigurable portion is held in reset, is not within the scope of this application note.

Introduction

Active partial reconfiguration of Virtex devices (simplified to "partial reconfiguration" in this document) is accomplished in either slave SelectMAP mode or Boundary Scan (JTAG) mode. Instead of resetting the device and performing a complete reconfiguration, new data is loaded to reconfigure a specific area of a device, while the rest of the device is still in operation. For current FPGA devices, data is loaded on a column-basis, with the smallest load unit being a configuration bitstream "frame," which varies in size based on the target device.

Partial Reconfiguration Overview

Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is done when the device is active. Except during some inter-design communication, certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming.

Styles of Partial Reconfiguration

There are also several styles of partial reconfiguration.

Multi-Column Partial Reconfiguration, Independent Designs

For designs where the modules are completely independent (e.g., no common I/O except for clocks; no communication between modules). In this case, reconfiguring one module does not affect the operation of another module. The **Module-Based Partial Reconfiguration** flow is used for these designs.

Multi-Column Partial Reconfiguration, Communication Between Designs

For modules that communicate with each other, a special bus macro (described in **Bus Macro Communication**) allows signals to cross over a partial reconfiguration boundary. Without this special consideration, intermodule communication would not be feasible as it is impossible to guarantee routing between modules. The bus macro provides a fixed "bus" of inter-design communication. Each time partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections. The **Module-Based Partial Reconfiguration** flow is used for these designs.

Small Bit Manipulations

This method of partial reconfiguration is accomplished by making a small change to a design (normally done in *FPGA_editor*), and then by generating a bitstream based on only the differences in the two designs. Switching the configuration of a module from one implementation to another is very quick, as the bitstream differences can be extremely smaller than the entire device bitstream. The **Small-Bit Manipulations** flow is appropriate for these designs.

Module-Based Partial Reconfiguration

The first partial reconfiguration flow is based on the Xilinx Modular Design methodology. The designer should read and understand the Modular Design chapter of the "Xilinx Development Systems Reference Guide (Chapter 3)" before proceeding.

Reconfigurable Module Overview

Partial reconfiguration involves defining distinct portions of an FPGA design to be reconfigured while the rest of the device remains in active operation. These portions are referred to as *reconfigurable modules*.

Reconfigurable modules have the following properties:

1. The reconfigurable module height is always the full height of the device.
2. The Reconfigurable module width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments.
3. Horizontal placement must always be on a four-slice boundary; the leftmost placement being x = 0, 4, 8, …
4. All logic resources encompassed by the width of the module are considered part of the reconfigurable module's bitstream "frame." This includes slices, TBUFs, block RAMs, multipliers, IOBs, and most importantly, all routing resources.
5. Clocking logic (BUFGMUX, CLKIOBs) is always separate from the reconfigurable module. clocks have separate bitstream frames.
6. IOBs immediately above the top edge and below the bottom edge of a reconfigurable module are part of the specific reconfigurable module's resources.
7. If a reconfigurable module occupies either the leftmost or rightmost slice column, all IOBs on the specific edge are part of the specific reconfigurable modules resources.
8. To help minimize problems related to design complexity, the number of reconfigurable modules should be minimized (ideally, just a single reconfigurable module, if possible). This said, the number of slice columns divided by four is the only real limit to the number of defined reconfigurable module regions.

9. A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed.
10. Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special bus macro (described in the **Bus Macro Communication** section).
11. The implementation must be designed so that the static portions of the design do not rely on the state of the module under reconfiguration while reconfiguration is taking place. The implementation should ensure proper operation of the design during the reconfiguration process. Explicit handshaking (e.g., module ready/not-ready) logic may be required.
12. The state of the storage elements inside the reconfigurable module are preserved during and after the reconfiguration process. Designs can take advantage of this fact to utilize "prior state" information after a new configuration is loaded. On the other hand, it is not possible to utilize the FPGA devices global set/reset (GSR) logic to independently initialize the state of the reconfigurable module. If set/reset initialization is required for the reconfigurable module, user-defined set/reset signals should be defined in the source HDL.

A layout of a design with two modules that are reconfigurable (shaded) are shown in Figure 1.

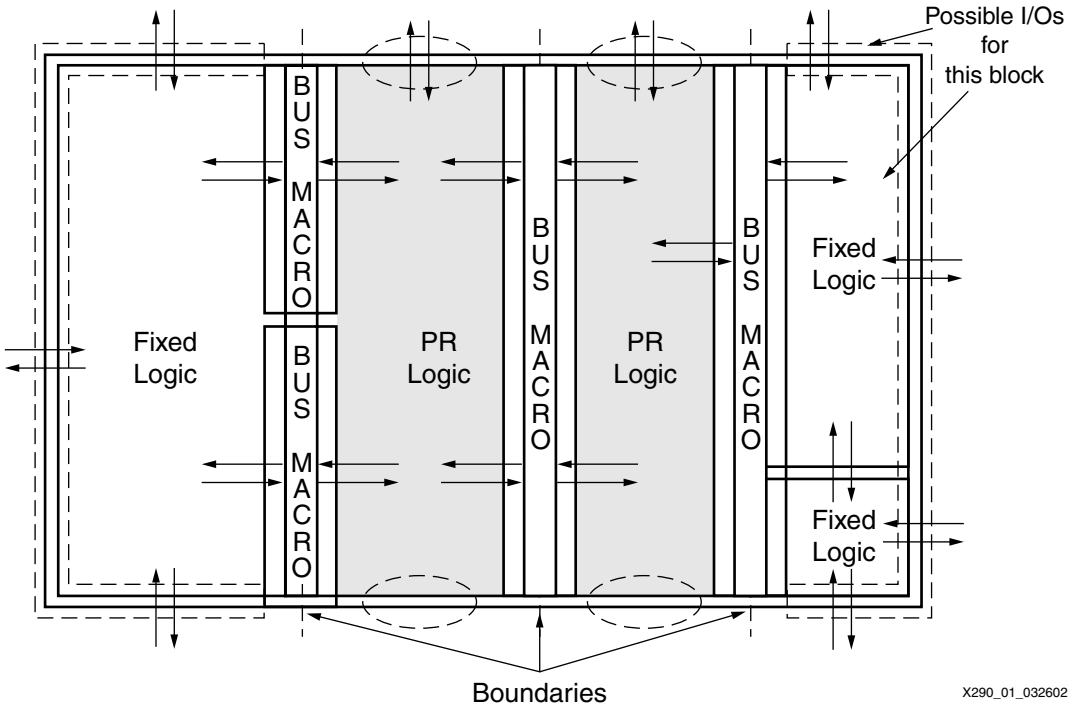


Figure 1: **Design Layout with Two Reconfigurable Modules**

Implementation Flow Overview

Creating a partial reconfiguration design requires the creation and implementation of the design within a set of specific guidelines. The partial reconfiguration flow utilizes a modified form of the Xilinx Modular Design process. For more details on modular design, see the modular design chapter of the Development System Reference Guide.

A general description of the flow is:

1. Design Entry - Write and synthesize HDL code in conformance with partial reconfiguration guidelines.
2. Initial Budgeting - Design the floorplan, constrain the logic, and create timing constraints for top-level design and each module.
3. Run Active Implementation (NGDBUILD, MAP, PAR, etc.) for:

a. Each module of the reconfigurable module.

b. Each configuration of a particular reconfigurable module.

4. Assembly Phase Implementation:

   a. Minimum - Full design (initial power-up configuration).

   b. Recommended - Every possible combination of device configurations of fixed and reconfigurable modules for simulation and/or verification purposes.

5. Verify design (static timing analysis, functional simulation).
6. Visually inspect design using *FPGA_Editor* to ensure no unexpected routing crosses module boundaries. Though the software enforces this rule, it is still important to manually check this result.
7. Create bitstream for full design (initial power-up configuration).
8. Create individual (or partial) bitstreams for each reconfigurable module.
9. Download device with initial power-up configuration.
10. Reprogram reconfigurable modules as needed with individual (or partial) bitstreams.

Recommended Project Directory Structure

Xilinx highly recommends the recommended project directory structure be created and followed. Since most designers are new to the modular design flow, such a structure greatly helps to organize the files created during each major phase of the design and implementation process. This application note assumes a directory structure conforming to the form outlined in Figure 2.

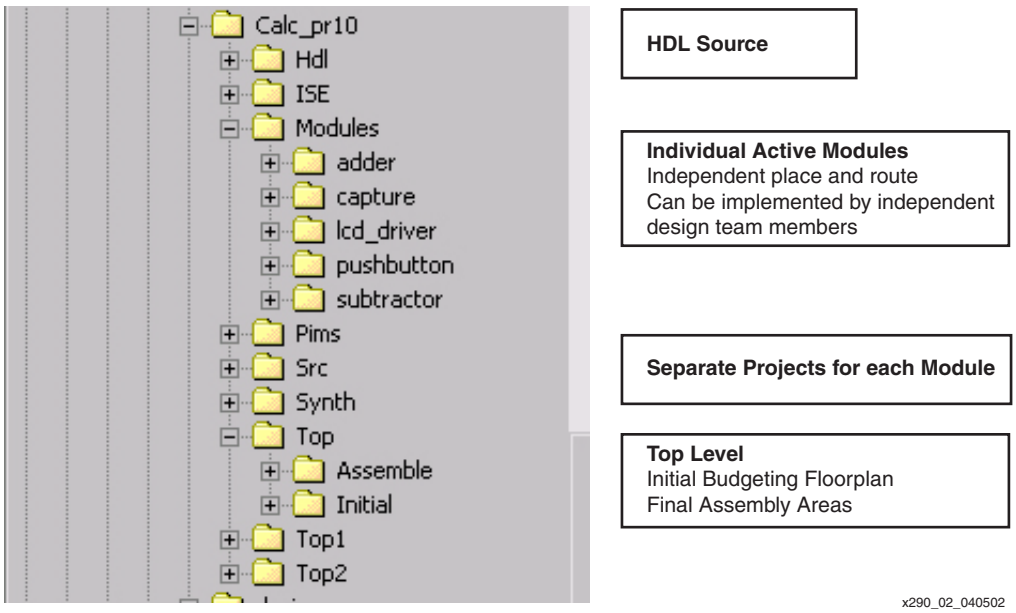


Figure 2: **Recommended Directory Structure**

Design Entry/HDL Coding/Synthesis Process Details

The HDL code for the design should be kept in the HDL folder of the project structure. Further subdirectories should be kept for the top-level design as well as each module. Likewise, synthesis projects and results for the top-level and each module should be kept in corresponding subdirectories of the Synthesis folder.

To conform to the requirements of partial reconfiguration, the HDL coding and synthesis process follows some general structural rules.

1. Overall structure should be a top-level design with each functional module defined as a "black-box" level of hierarchy. Logic at the top level should be limited to I/Os, clocking logic, and the instantiations for the bus macros. There should be no other logic in the top-level design.
2. For new users of partial reconfiguration, it is highly recommended to minimize the number of reconfigurable modules in a design. Ideally, a single reconfigurable module would be most easily handled. Note, however, this is only a recommendation, not a limit imposed by the implementation tools.
3. Each module, whether a reconfigurable module or fixed, should be a self-contained block of the logical hierarchy. Declaring port definitions as input or output are required for all module ports.
4. Bus macros are used as fixed data paths for signals going between a reconfigurable module and another module (Figure 3). The HDL code should ensure that any reconfigurable module signal that is used to communicate with another module does so only by first passing through a bus macro. There are Virtex, Virtex-E, and Virtex-II series specific versions of the bus macro. Be sure to instantiate the version compatible to the chosen device. Each bus macro provides for 4-bits of intermodule communication. As many bus macros as needed must be instantiated to match the number of bits traversing the boundaries of the reconfigurable modules. As an example, if reconfigurable module A communicates via 32 bits to module B, then eight (32/4) bus macros will need to be instantiated to define the data paths between modules A and B. The details of bus macro usage are described in the bus macro communication section.

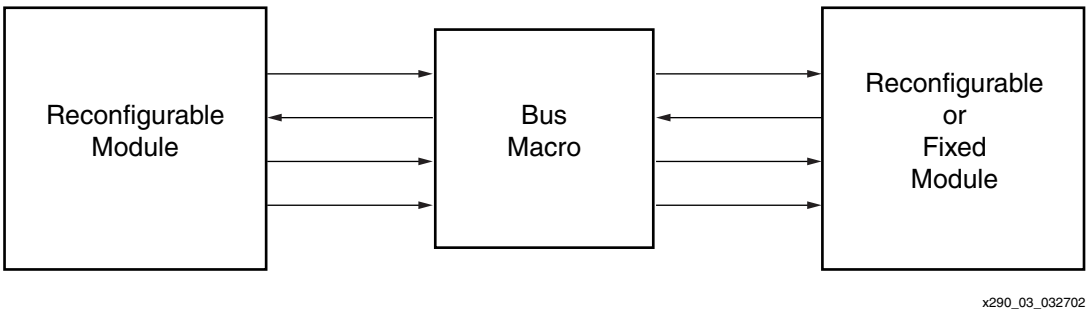


Figure 3: **Bus Macro Used for Intermodule Signals**

5. If a signal "passes through" a reconfigurable module connecting the two modules on either side of the reconfigurable module, bus macros must be used to make that connection. This effectively requires creation of an intermediate signal that is defined in the reconfigurable module. The signal cannot be actively used during the time the reconfigurable module is being configured.
6. For the sake of simplicity, especially for new users of partial reconfiguration, Xilinx highly recommends keeping clock design as straightforward as possible. Avoid designing clocks to use in one variation of a reconfigurable module, but not in others. Though this can be done via a "Clock Template" structure (see **Appendix C**), it is an advanced technique that partial reconfiguration neophytes should avoid. Use of clocking logic, such as DCM, is also best avoided. In future releases of partial reconfiguration, Xilinx will publish how to take advantage of the advanced clocking capabilities within partial reconfiguration designs.
7. All clocks defined must use dedicated global routing resources. Bitstream frames for global clocks are separate from those bitstream frames defining the CLBs. The Partial Reconfiguration flow depends on this separation to keep clocks functional during reconfiguration. Do not define clocks that use "local" (non-global) resources.

8. Reconfigurable modules must not directly share any signals with other modules, except clocks. This includes resets, constants (V_{CC}, GND), enables, etc. See **Appendix B** (Coding Methodology) for examples on creating constants locally within a reconfigurable module.
9. The top-level is synthesized with I/O insertion enabled, producing a top-level netlist.
10. Each module is synthesized with I/O insertion disabled, producing a module-level netlist for each module.

These guidelines are very similar to those specified for the modular design flow.

The modular design chapter of the Development System Reference Guide has further details and examples of proper HDL coding styles and synthesis techniques. To help verify proper practice for the HDL coding and synthesis phase, consult **Appendix D** for a top-level HDL design checklist and a module HDL design checklist.

Bus Macro Communication

To facilitate communication across reconfigurable module boundaries, yet still conform to the partial reconfiguration requirement that routing resources across such boundaries be completely fixed and static, the use of a special bus macro is required.

In the following diagram, the left half "A" is a module and the right-half "B" is another module. Either A, B, or both could be partially reconfigurable. To support communication between modules A and B, a special bus macro is used. Partial reconfiguration requires the signals used as communication paths between or through reconfigurable modules must use fixed routing resources. That is, the routing resources used for such intermodule signals must not change when a module is reconfigured. As shown in Figure 4, the bus macro is a fixed routing bridge between the two sides, facilitating reliable communication. It is a pre-routed hard macro used to specify the exact routing channels and will not change from compilation to compilation. For each of the different design implementations, there is absolutely no variation in the bus macro routing.

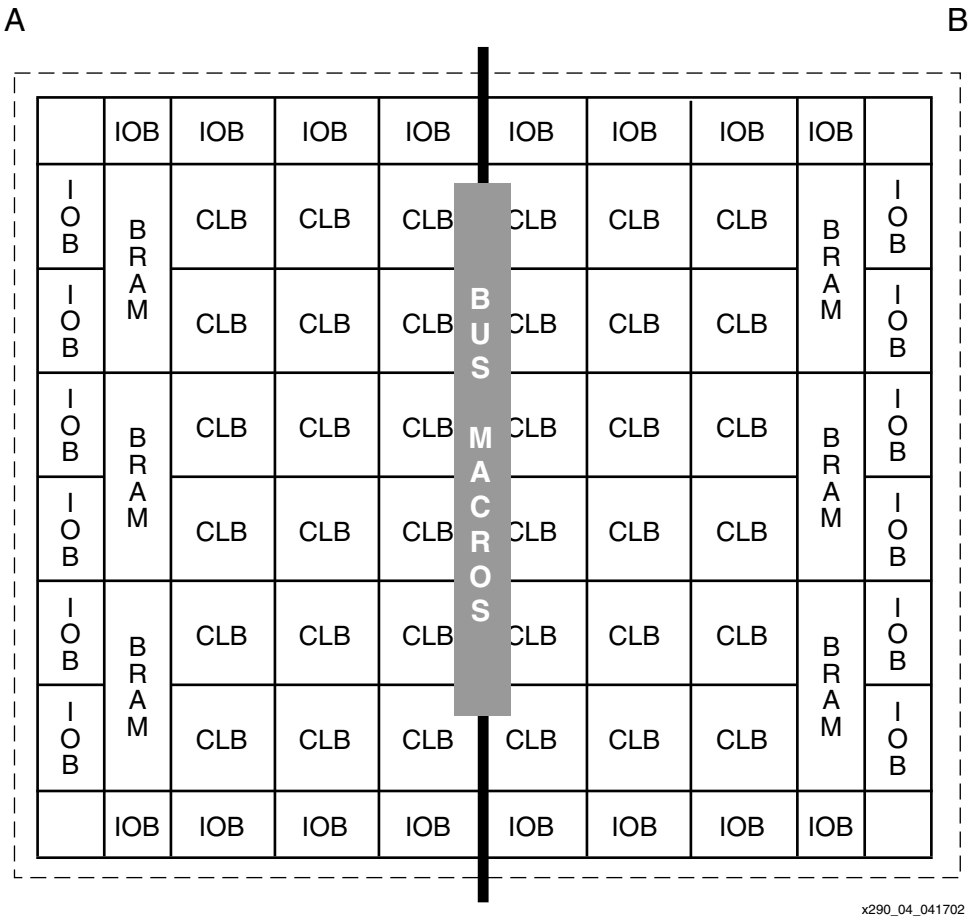


Figure 4: **Bus Macro**

This route locking is required, because if any of the designs choose a different routing for the bus macro (example A1), it will not align properly with other designs (example B1), and the communication between the two halves is effectively broken.

The current implementation of the bus macro uses eight 3-state buffers (TBUFs) hooked up in an arrangement that allows one bit of information to travel either left-to-right or right-to-left, using one TBUF longline per bit (Figure 5). Each row of the Virtex device can support four bits of a bus macro. The bus macro position exactly straddles the dividing line between design A and B, using four columns of TBUFs on the A side, and four columns of TBUFs on the B side. With the Virtex-II architecture, only two columns on either side are used. Design A only connects to the TBUFs in the two or four columns on the Design A side. Likewise, Design B only connects to the TBUFs in the two or four columns on the Design B side. The "fixed bridge" that is pre-routed is comprised of the TBUF output longlines to ensure reliable communication between the two sides.

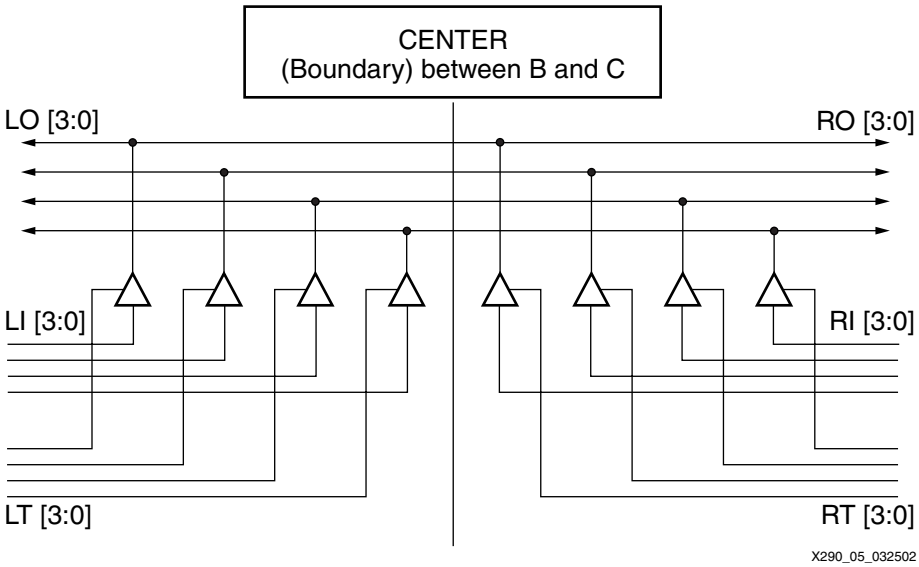


Figure 5: **Physical Implementation of Bus Macro**

The bus macro must be physically locked in such a way as to straddle the boundary line between A and B, and it must be locked in exactly the same position for all compilations. The process of locking the bus macros to proper locations is described in the **Initial Budgeting Phase Details** section. The bus macro can be wired so that signals can go in either direction (left-to-right or right-to-left). For this release, it is strongly recommended that once direction is defined, it should not change for that particular FPGA design, i.e., bus macro signals should neither be bidirectional nor reconfigurable. (Note that this restriction might be removed in future releases of the partial reconfiguration flow.)

The number of bus macro communication channels is limited by the number of horizontal longline routing resources available in each CLB tile.

Implementation Using Modular Design

As defined by the modular design flow, the partial reconfiguration implementation process is broken down into three main phases:

1. Initial Budgeting Phase - Creating the floorplan and constraints for the overall design.
2. Active Module Phase - Implementing each module through the place and route process.
3. Final Assembly Phase - Assembling individual modules together into a complete design.

Initial Budgeting Phase Details

Initial budgeting operations should be done in the top or initial folder of the recommended project directory structure.

The initial budgeting phase has the following main steps:

1. The floorplanning of module areas:
   a. Have a four-slice minimum width.
   b. Have a set width that is always a multiple of four slices (e.g., 4, 8, 12, …)
   c. Are always the full height of the device.
   d. Are always placed on an even four-slice boundary.

e. Attach partial reconfiguration flow-specific properties to the area groups in the .ucf file (See **Appendix A**).

2. The floorplanning of all IOBs:
   a. Shall be wholly contained within the "columnar space" of their associated reconfigurable module. No intermixing between columnar regions is allowed.
   b. All IOBs must be locked down to exact sites.
3. The floorplanning of all global logic:
   a. Logic that is not part of a lower level module must be constrained to specific sites in the device via LOC constraints. Typically the Floorplanner tool can be used to create these constraints.
   b. There must be no unconstrained top level logic.
4. LOC constraints are manually inserted for each bus macro into the .ucf file (the current version of the Floorplanner does not support placement of bus macro elements, this must be done manually). Locate the bus macro to exactly straddle the boundary between the modules forming the communication bridge. Each bus macro will occupy a 1-row by 8-column section of TBUF site space.
5. Global-level timing constraints are created for the overall design, using the Constraints Editor, if desired.

The output of the initial budgeting phase is a .ucf file containing all placement and timing constraints. Each module is implemented using this .ucf file, in addition to any module-specific constraint requirements.

For partial reconfiguration, there is a significant difference from the standard modular design flow; floorplanning in the partial reconfiguration flow does not involve the port-placement process. In partial reconfiguration designs, all reconfigurable module inputs and outputs connect to either primary I/Os, global logic, or bus macros. No signals going to or from a reconfigurable module will load or source any element in another module without first passing through a bus macro. Unlike a standard modular design, a partial reconfiguration design does not have intermodule ports. In fact, if pseudo-drivers or pseudo-loads are found when viewing the design in the floorplanner, the design violates the criteria that all intermodule signals must utilize a bus macro channel. If this occurs, re-examine the HDL source and correct the problem.

As with modular design, the product of the initial budgeting phase is a .ucf file describing all of the following:

1. The area-based floorplan for each module.
2. Fixed location constraints for all top-level (context) logic.
3. Fixed location constraints for all bus macro instantiations.
4. Any global level timing constraints.

This .ucf file is used during the active implementation of each module. To help verify proper practice during the initial budgeting phase, consult **Appendix D**, **Checklist for Initial Budgeting (Floorplanned and other .ucf constraints)**.

Active Module Phase Details

Up to this point, Xilinx has designed, synthesized, floorplanned, and constrained the partial reconfiguration design. Now implementation of ("place and route") all modules (both fixed and partially reconfigurable) for the design can start. Each module will be implemented separately, but always in the context of the top-level logic and constraints. Bitstreams will be generated for all reconfigurable modules. This section describes the overview of how to independently implement each module. Again, this process is in conformance with the flow described for modular design. Copy the .ucf file created during the initial budgeting phase (top or initial folder) to the active implementation directories for each module (`Active/*`).

1. In each active module working directory, augment the local copy of the .ucf file with any module-level timing constraints required to specify the performance requirements for that module.
2. The Constraints Editor can be used to create module-level timing constraints. Run `NGDBUILD`, `MAP`, `PAR` `BITGEN` and `PimCreate` for each module according to the script in the example design described in **Appendix E**. This will result in a placed and routed module as well as a module-specific bitstream.
3. The *PimCreate* process "publishes" the routed design (and associated files) to the *Pims* folder. This will be used during final assembly phase later in the implementation process.
4. Optionally, run `ngdanno`, and `ngd2ver/ngd2vhdl` if module-level simulation is to be done.
5. Using *FPGA_Editor*, visually inspect the routed design to verify that routing does not expand beyond the module boundary except, of course, for signals traversing to other modules via the bus macro structures. Figure 6 is a view in *FPGA_Editor* of such a routed design.

Repeat steps 1 through 4 for each module in the design. To help verify proper practice during the Active Module implementation phase, consult **Appendix D**, *Checklist for Active Module Implementation*

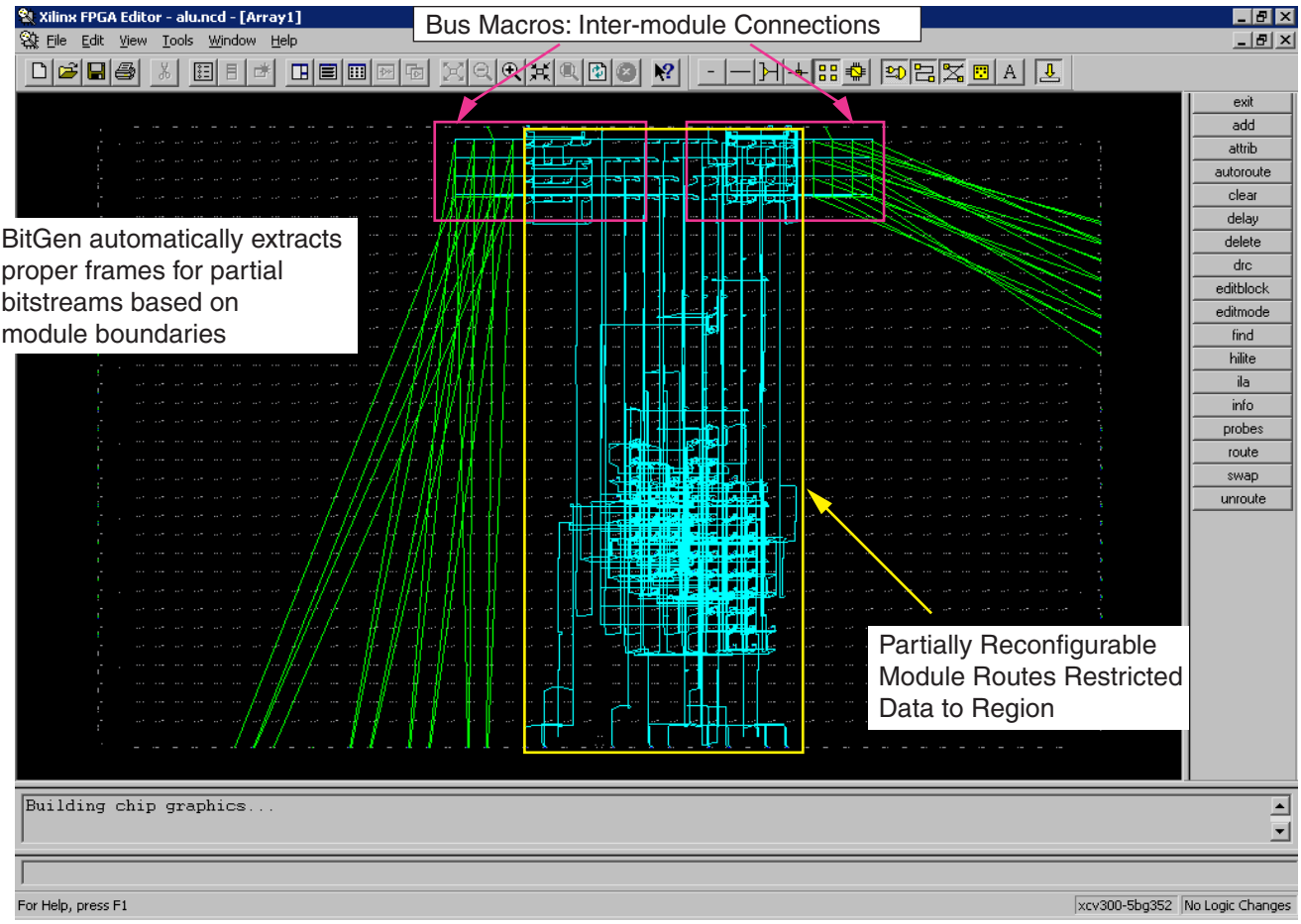


Figure 6: **FPGA_Editor Routed Design for Reconfigurable Module**

Final Assembly Overview

The final assembly phase is the process of combining each of the individual modules back into a complete FPGA design. The placement and routing achieved during the active

implementation phase for each module will be preserved, thereby, maintaining the performance of each module.

The steps of the final assembly phase are to be run in the `Top/Assemble<n>` directories of the recommended project directory structure. Where `<n>` corresponds to each possible combination of fixed and reconfigurable modules.

At the very minimum, at least one final assembly must be run (e.g., `Top/Assemble<1>`). The partial reconfiguration flow requires that the initial bitstream loaded into the FPGA device be a complete design. This is required so that all global, non-reconfigurable logic is placed and locked down, and that only reconfigurable portions of the design will change during partial reconfiguration. However, it is highly recommended that all module combinations are compiled into unique assemblies (top or assemble`<1 ..n>`). Compiling each possible combination allows simulation and verification that each combination functions as intended.

1. Copy the .ucf file created during the initial budgeting phase (top or initial folder) to the final assembly directory for each full design combination (top or assemble`<n>`).
2. Run NGDBUILD, MAP, PAR BITGEN according to the example final assembly script in **Appendix E**.
3. This will result in a placed and routed design as well as a full-design bitstream. Optionally, run `ngdanno` and `ngd2ver/ngd2vhdl` if simulation is to be done.
4. Using *FPGA_Editor*, visually inspect the routed design to verify that local-module routing does not expand beyond the module boundaries except, of course, for signals traversing to other modules via the bus macro structures.

Repeat steps 1 through 4 for each possible combination of fixed and reconfigurable modules in the design. To help verify proper practice during the active module implementation phase, consult **Appendix D**, *Checklist for Assembled Design*.

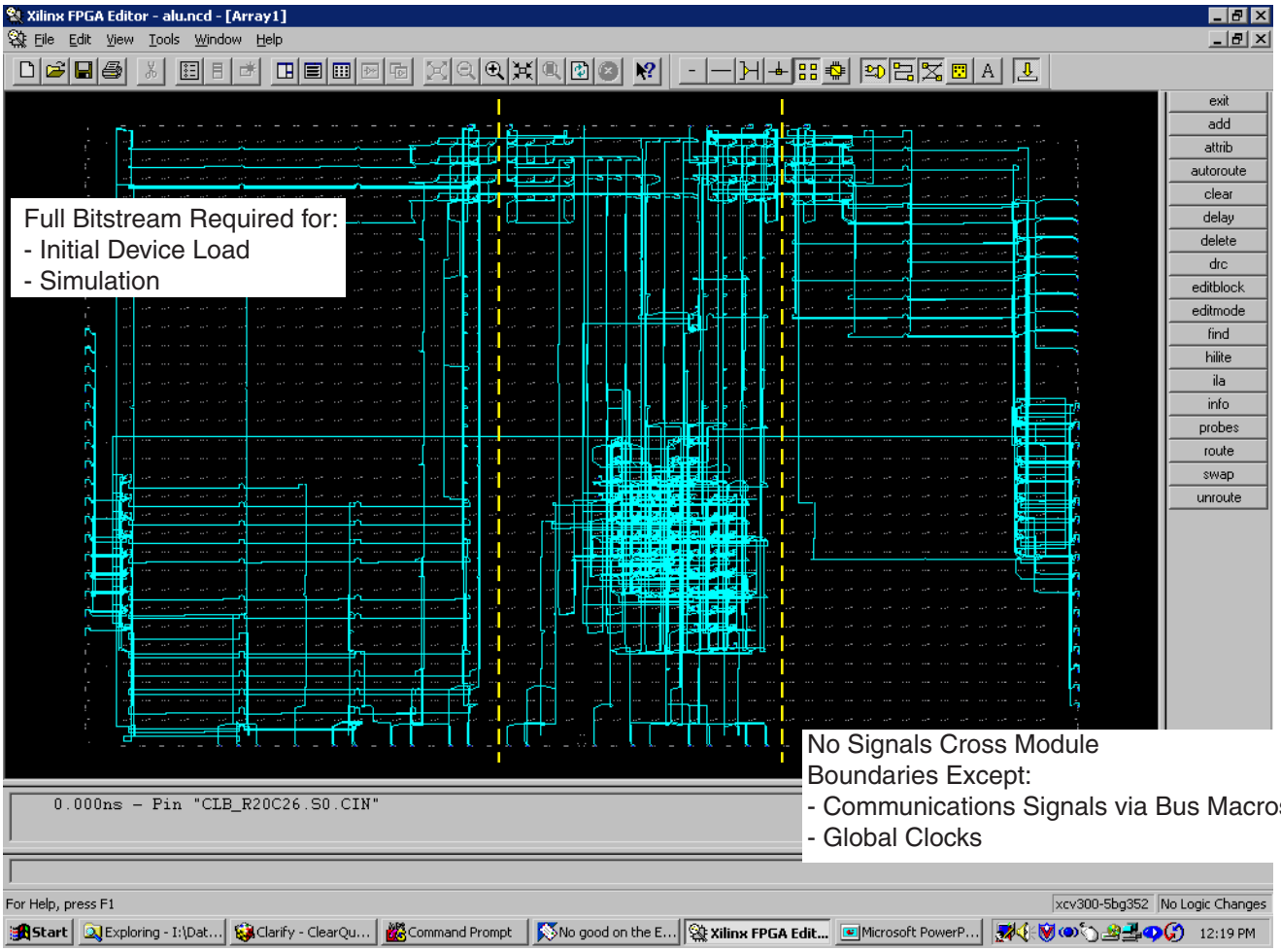


Figure 7: **Routed Design After Module Final Assembly**

Using Bitstreams and Programming the FPGA

At present, multiple bitstreams created for a fully-assembled design require one bitstream (at a minimum) for the initial configuration of the device and one for every partially reconfigurable module variation. As an example, if Module A is reconfigurable, and four possible configurations for Module A are designed, bitstreams for Module A1, A2, A3 and A4 are available.

Partial reconfiguration supports either the parallel SelectMAP or serial JTAG programming options. The Xilinx configuration application, iMPACT software, can be used in conjunction with any Xilinx download cable to interface to target device(s) for configuration. Alternatively, the user can create board-level functions to control device configuration.

Because partial bitstreams are indistinguishable from full bitstreams, end users must be careful to correctly sequence the application of these partial bitstreams to the target devices. The iMPACT software can identify a partial bitstream but cannot determine if it is being applied in the correct sequence order. When downloading a device using a partial bitstream, iMPACT software displays a message indicating that a partial bitstream is being used and that care should be taken to ensure correct sequencing. Beyond that, the end-user configuration experience with partial bitstreams is identical to that of full bitstreams.

When targeting a partial bitstream to a Xilinx configuration PROM using the iMPACT PROM file formatting capabilities, no special options or operations need be followed. The formatting of the PROM data is identical regardless of the bitstream contents. End users should be aware that

when targeting Xilinx XC1700 or XC1800 configuration PROMs these devices do not allow selective loading of configuration data contents. Instead, all data is transmitted to the attached FPGAs. If end users are looking for a solution to provide bitstream selectability, they should consider either the System ACE™ MPM for medium-density solutions or the System ACE CF for high-density solutions.

On device power-up, a full bitstream must be loaded into the device. Only then, can a partial bitstream be loaded to reconfigure a partially reconfigurable module. The state of FFs and RAMs are preserved during the reconfiguration process. Fixed modules or other reconfigurable modules not being reconfigured can remain fully operational during the reconfiguration of a reconfigurable module. However, if the module(s) not being reconfigured relies on the state of signals connected to the bus macro of the module under reconfiguration, the design must account for the "transition time" during partial reconfiguration. For example, the fixed module communicating with a reconfigurable module by way of a bus macro must not rely on signals to or from the bus macro during the time of reconfiguration.

Bit Length and Reprogramming Speed

The bit length and reprogramming speed of a particular partial bitstream are directly proportional.

To help verify proper practice during bitstream generation and configuration phase, consult **Appendix D**, *Checklist for Bitstream Generation and Configuration*.

Known Limitations

The following are known limitations:

1. As with normal modular design, the PCI Core is not yet supported. When support for the PCI core becomes available, this application note will be updated with new instructions on its use within a partially reconfigurable design.
2. In Virtex-II Pro designs, use of the microprocessors is not yet supported within reconfigurable modules. This capability will be in a future release of the partial reconfiguration flow.

Small-Bit Manipulations

Small sections of a Virtex-II design can be modified using the FPGA Editor tool. BitGen switches can produce custom bitstreams that only modify small sections of the device. Switching the configuration of a module from one implementation to another is very quick, as the bitstream differences are smaller than the changes to an entire device bitstream. These bitstreams can be loaded quickly and easily due to their size and software support.

In designs where large blocks of logic are meant to be reconfigured, the Modular Design flow described in the **Module-Based Partial Reconfiguration** section is required. However, there are lots of uses for minute design changes. Perhaps LUT programming or an I/O standard needs to be changed and loaded on the fly. These sorts of changes can be made by directly editing the routed NCD file in the Xilinx FPGA Editor application. If BRAM contents need to be modified, the Data2BRAM utility can help, or these changes can be made in FPGA Editor as well. Once the changes are made, the NCD file can be saved to a different name, and the BitGen program can be used to produce a bitstream that only programs the differences between the original design and the new one. Depending on the changes made, this partial bitstream can be orders of magnitude smaller than the original. All that is required is a good understanding of how to make logic changes using the FPGA Editor application, and the pertinent options to select in BitGen.

Using FPGA Editor to Make Small Changes to a Design

While there are a myriad of different types of changes that can be made to an FPGA design, this application note only addresses three of them — changing I/O standards, BRAM contents, and LUT programming. While it is possible to change routing information, this is not recommended, due to the possibility of internal contention during reconfiguration. If routing changes are desired, using the flow described in this application note is recommended.

Once the routed NCD file is opened in FPGA editor (by specifying it on the command line or using the **File->Open** menu selection), immediately save it under a different name, so the original design is not lost. In the first example (Figure 8), **File->Save As** is selected to change the `and_test.ncd` design to `and_test2.ncd`. The latter file will remain open in FPGA Editor once the operation is completed.

Once the new design is open, make the file available for modification by selecting **File->Main Properties** and changing the **Edit Mode** to **Read Write**.

Changing LUT Equations

The smallest logical element that can be selected is the Slice. First, the block must be viewed. An individual slice can be found using the Find button on the right hand side of the window, or the array view can be navigated, and the slice selected by hand. Once the slice is selected, (shown as red in Figure 8) click the Editblock button to open the Block Editor toolbar.

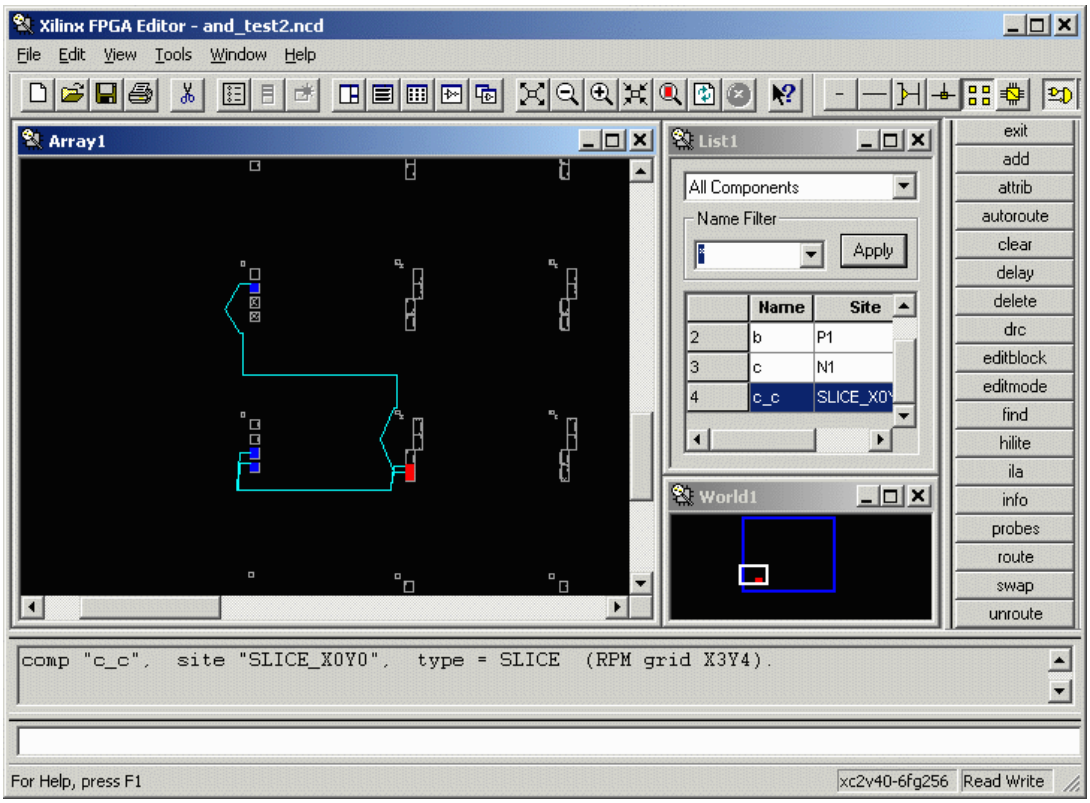


Figure 8: **Viewing a Block**

To prevent accidental edits, by default the internals of a slice can not be edited. Each time a block is opened, to make it editable, select the Begin Editing button (the second button from the left in the Block Editor toolbar). This will change the window background to black.

To view the LUT equations, click on the Show/Hide Attributes button. It is the **F=** toolbar button. This opens a panel at the bottom of the window with the slice name, and the two equations. The valid operators are:

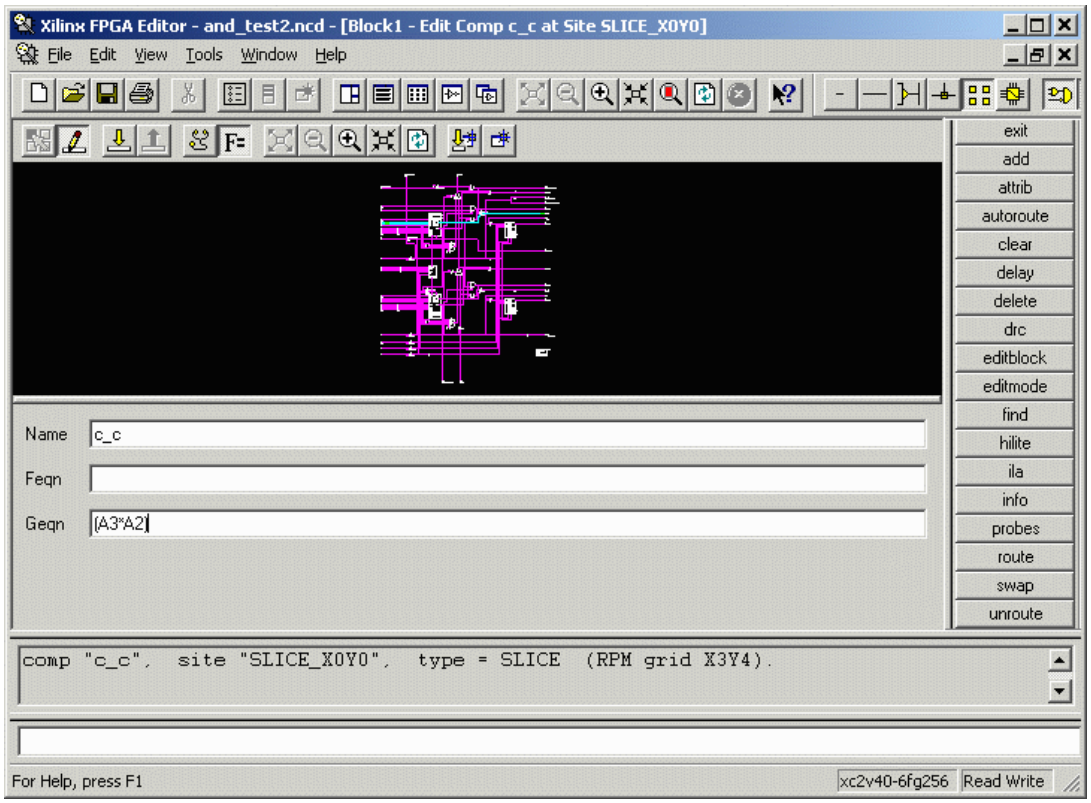
* -> Logical AND

+ -> Logical OR

@ -> Logical XOR

~ -> Unary NOT

Figure 9 shows changing the Geqn from A3*A2 to A3*~A2.



x290_13_042402

Figure 9: **Changing LUT Equations**

Valid equations values are A1, A2, A3, and A4, representing the four address line inputs to the LUT. Parentheses can also be used to group equation sections, e.g. `(A4 * A1) @ ~A3`. Any other names or operators will produce an error (for example):

`ERROR:FPGAEDITOR:24 - "(A3*~A2 + mynet)` is not a valid value for the Geqn attribute.

Once the attributes are changed, select the Saves Changes and Closes Window buttons to close the Block Editor.

Changing Block RAM Contents

The Block Editor for block RAMs (Figure 10) is very similar to the Slice Block Editor. Once in the Block Editor mode, select Show/Hide Attributes to display the contents of the RAM. The format of the data is the same as an INIT constraint in a UCF file. See the *Libraries Guide* for details on the INIT constraint.

Once the changes have been made, select the Saves Changes and Closes Window buttons to clock the window and return to the Array view.

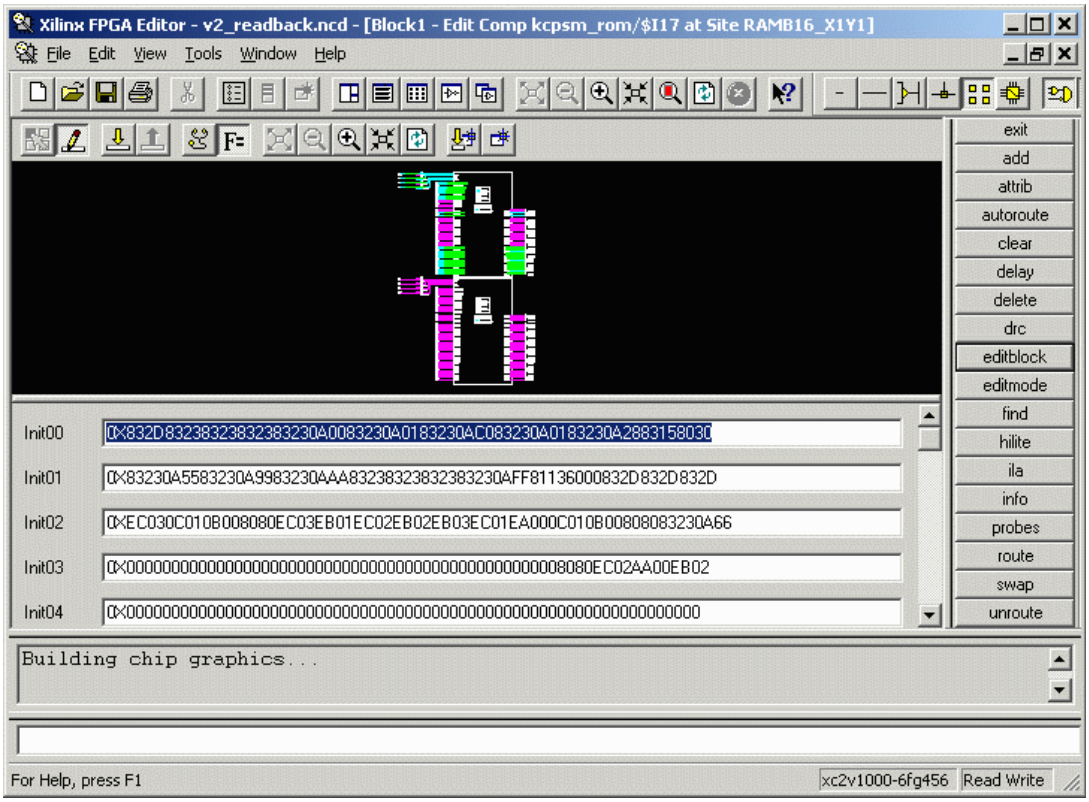
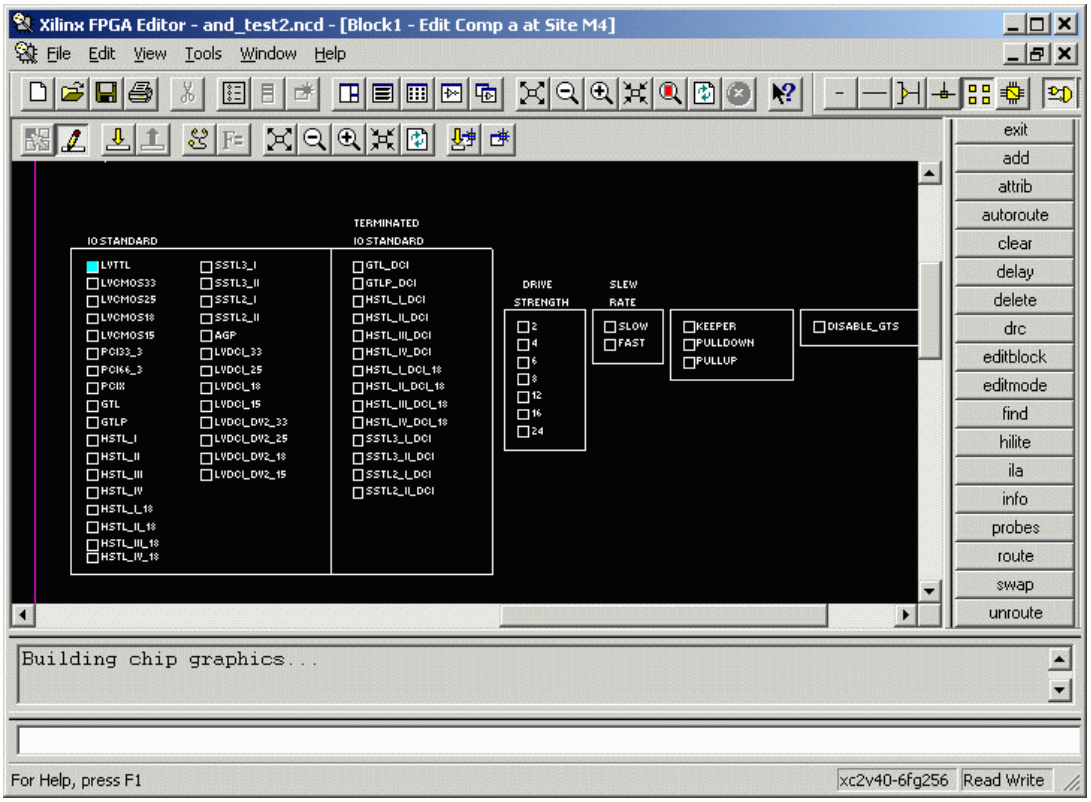


Figure 10: **Changing Block RAM Contents**

Changing I/O Standards

To change the I/O standards, enter the Block Editor the same way as a Slice or block RAM. The I/O standards are in a box in the upper-right corner of the window (Figure 11). To change the I/O standard, select the checkbox next to the desired I/O standard. There are also *Drive Strength* and *Slew Rate* checkboxes. Only select these when applicable. See the *Libraries Guide* and the data sheet for details on which I/O standards have selectable slew rate and drive strength.

Choose I/O standards to match the V_{REF} voltages (or the absence of a V_{REF} voltage) with the other I/Os in the bank or the changed I/Os will not function properly. For example, it is not possible to change an LVTTL I/O in the middle of a bank of LVTTL I/Os to the GTL standard. GTL requires V_{REF} voltages, LVTTL does not.



x290_15_042402

Figure 11: **Changing I/O Standards**

Other Changeable Elements

A number of muxes and changeable properties in slices, IOBs, and block RAMs are eligible for an active reconfiguration flow. Some changeable properties are: muxes that invert polarity, flip-flop initialization and reset values, pull-ups or pull-downs on external pins, or block RAM write modes. All of these properties can be modified in the actual slice, IOB, or block RAM as appropriate. It is not recommended to change any property or value that would impact routing, due to the risk of internal contention.

Creating a Partial Reconfiguration Bitstream

A partial reconfiguration bitstream can be created using the BitGen program. Properly used, the `-r` switch will produce a bitstream that contains only the differences between the input routed `.ncd` file, and the old bit file. In addition to the `-r` switch, the `-g ActiveReconfig:Yes` switch is required. This means that the device remains in full operation while the new partial bitstream is being downloaded. If the `ActiveReconfig:Yes` is not specified (or the `-g ActiveReconfig:No` is specified), the partial bitstream will contain the Shutdown and AGHIGH commands used to deasserted DONE. All the I/Os and internal routing should be high impedance, and writing to registers should be disabled. The changes described in this application note can be done safely with the `-g ActiveReconfig` option set to Yes. An example is:

```
bitgen -g ActiveReconfig:Yes -r and_test.bit and_test2.ncd
and_test2_partial.bit
```

This will produce a configuration file (`and_test2_partial.bit`) that only configures the frames that are different between *and_test* and *and_test2*. When downloading this file, the *and_test* configuration file **MUST** already be programmed into the device.

This partial reconfiguration option can be used in conjunction with any other BitGen option, including the `-b` option (create `.rbt` file) or any `-g` options specifying configuration options, except for encryption. A device that has been configured with an encrypted bitstream cannot be partially reconfigured. Similarly, a device cannot be partially reconfigured with an encrypted bitstream.

Reference Design

The reference design files for this application note is provides an example of the partial reconfiguration flow in VHDL. It is available on the Xilinx FTP site at: **ftp://ftp.xilinx.com/pub/applications/xapp/xapp290.zip**.

Conclusion

This application note can be very useful when trying to partially reconfigure Xilinx FPGAs. Since there is much to do, the user is urged to read this document thoroughly before proceeding with implementation.

Appendix Overview

Appendix A: Specific UCF Constraints for Partial Reconfiguration

Appendix B: HDL Coding Considerations

Appendix C: Clock Template

Appendix D:

- **Checklist for Top-Level HDL design** and **Checklist for Module HDL design**
- **Checklist for Initial Budgeting (Floorplanned and other .ucf constraints)**
- **Checklist for Active Module Implementation**
- **Checklist for Assembled Design**

Appendix E: Command Line Compilation Scripts

Appendix F: Special Cases in the Bitstream Architecture

Appendix G: Static Partial Reconfiguration

Appendix A

Special UCF Constraints for Partial Reconfiguration

Area Group Properties

For each module defined "area group", manually attach properties to properly handle the design for the partial reconfiguration flow. For example:

```
AREA_GROUP "My_PR_AG" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING
RECONFIG_MODE;
```

Location Constraints for Bus Macros

A separate location constraint must be created for each bus macro. The Floorplanner cannot be used to create these constraints and they must be manually entered into the.ucf file. For example:

```
INST "BM_MyBusMacro_1 "LOC = "TBUF_X0Y8" ;
```

Note: Must be on 4-column boundary x = 0, 4, 8, etc.

Appendix B

HDL Coding Considerations

Bus Macro Instantiation

Examples of bus macro instantiation can be found in the supplied example tutorial designs.

VHDL: `Calc_pr9/Hdl/calctop.vhd`

Verilog: `alu_pr/Hdl/alu.v`

Generation of Local Constant for Modules

The simplest way to generate local constants (PWR and GND) for modules is to create the constants from otherwise unused IOBs and drive them into the modules as needed. Of course, each module needs to have enough unused IOBs for this method to work. The alternative is to create "dummy" LUTs in each module for each constant needed.

For example, to create a local "Logic_0" (GND) signal, instantiate a LUT1 primitive.

```
LUT1 mylut1 (.I0 (), .O(Dummy_gnd));
```

Then, in the .ucf file for that module, add the following:

```
inst mylut1 LOCK_PINS;    # prevent trimming of the instantiated LUT
inst mylut1 init=0;       # initialize to drive a "0"
```

For a local "Logic_1" (V_{CC}):

```
LUT1 mylut1 (.I0 (), .O(Dummy_vcc));
```

Then, in the .ucf file for that module, add the following:

```
inst mylut1 LOCK_PINS;    # prevent trimming of the instantiated LUT
```

```
inst mylut1 init=1;       # initialize to drive a "0"
```

Once created, the signals *Dummy_gnd* and *Dummy_vcc* can be instantiated into the module and used as local constants. Each module must have its own local source of constants. Constants cannot be shared between modules.

Appendix C

Clock Template

In order to make some partial reconfiguration designs work, it may be necessary to use a clock template specially constructed for this purpose. It is the starting point for each reconfigurable module illustrated in Figure 12.

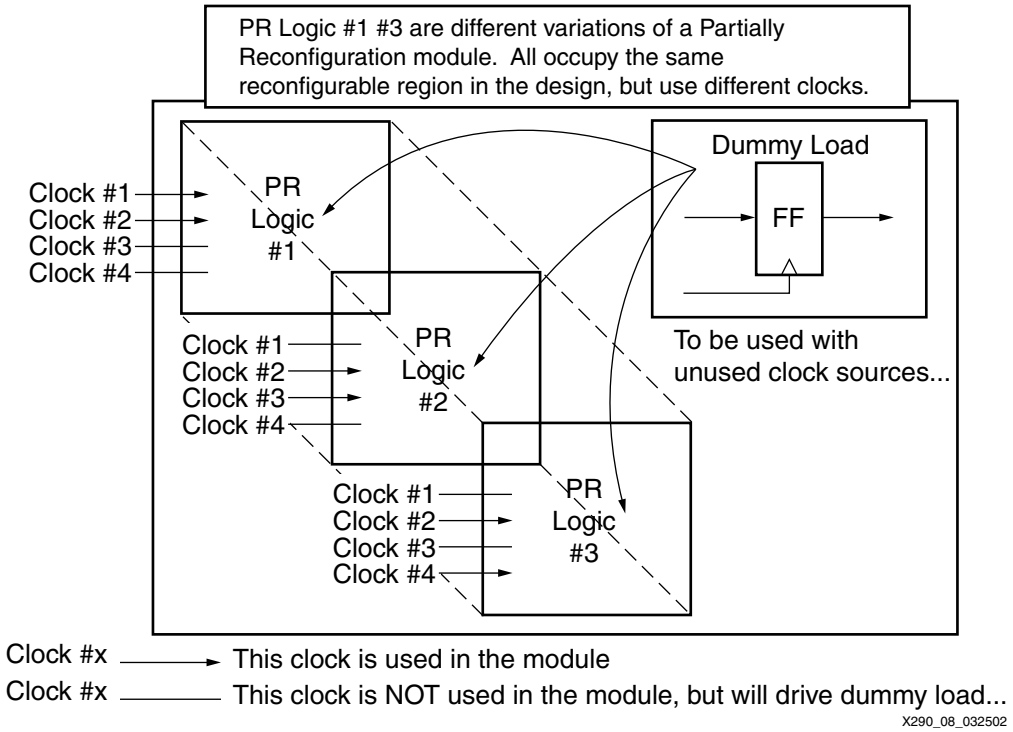


Figure 12: **Example of Design Requiring Clock Templates**

If a given reconfigurable module uses different clocks in each module variation, a clock template is a necessary step in the partial reconfiguration process. This is required in order to program the clock frames correctly. As an example, if reconfigurable module #1 is loaded into the device, the clock frames are programmed to meet the clocking needs of reconfigurable module #1, but might not accommodate the needs of reconfigurable module #2. If reconfigurable module #2 is then loaded, it will either overwrite the clock frames (and simultaneously overwrite the clock programming of reconfigurable module #1), or it will not program the clock frames at all, causing the clock programming from reconfigurable module #2 to be non-existent.

The clock template programs each of the clocks used in a system, even if the clock is not used in a particular design implementation. As an example, if reconfigurable module #1 uses Clock 1 and Clock 2, and reconfigurable module #2 uses Clock 2 and Clock 3, then the clock template consists of Clocks 1, 2, and 3. Reconfigurable module #1 routes Clock 3 to a dummy flip-flop and adds the appropriate constraints so that the loadless Clock 3 is not removed. Similarly, reconfigurable module #2 ties a dummy load to the unused Clock 1.

In this way, when reconfigurable module #1 is compiled, and its bitstream writes the clock frames, the clock programming for Clocks 1, 2, and 3 will be present, and both module #1 and #2 will work properly. This method is extendable; for example, if design reconfigurable module

#3 also uses Clock 4, then Clock 4 is a necessary part of the clock template. The total number of different clocks in all designs must not exceed the clock resources of the device.

Because of the way modular designs are compiled, creating a clock template is not necessary if there are no clock or clock programming differences between any of the reconfigurable modules.

Appendix D

Checklist for Top-Level HDL design

1. Check that no shared signals (other than clocks) between modules.
2. All intermodule signals must use bus macro instantiations (4-bits each).
3. All I/Os in the design must be at the top-level.
4. Design accounts for proper handshaking to ensure no operational dependencies on the state of the reconfigurable module while it is being reconfigured

Checklist for Module HDL design

1. Check that no instantiated I/Os are inside the module.
2. All constants must be local to each module (not shared with other modules).

Notes:

1. Some of these HDL coding restrictions are in place to simplify the design process for new users. These restrictions might be relaxed in future releases of this flow.

Checklist for Initial Budgeting (Floorplanned and other .ucf constraints)

1. Check that areas always span full height of the device.
2. Areas for reconfigurable modules that communicate via bus macros must share a common boundary with no "white space" between the areas.
3. Boundaries fall on x=0,4,8, etc.
4. Minimum width is 4 slice columns.
5. If bus macro in leftmost position, bits 0 and 1 cannot go right-to-left.
6. If bus macro in rightmost position, bits 2 and 3 cannot go left-to-right.
7. The bus macro must be exactly centered on the module's boundaries.
8. Add proper partial reconfiguration-specific properties to Area groups (**Appendix A**).
9. All IOBs must be locked down.
10. IOBs can only be placed in sites within the columnar area for the module of which they are a member. If the area is immediately adjacent to the left or right edge of the device, all IOBs on that edge are also available for assignment to connections in that module.
11. All clock buffers must be locked down to specific user-defined locations.
12. When Floorplanning, make sure there are ***no*** pseudo-drivers or pseudo-ports. If such elements are found, correct the design HDL source to use bus macros for all intermodule communication.

Checklist for Active Module Implementation

1. Copy "top-level" .ucf into active module implementation directory.
2. Create module-specific timing constraints.
3. After implementation, verify signals are wholly contained within then module boundary, except those traversing boundary via the bus macro (use *FPGA_Editor*).

Checklist for Assembled Design

1. Create all possible module combinations as "assembled designs" for simulation/verification purposes.
2. Verify signals are wholly contained within module boundary except those traversing boundary via bus macro (use *FPGA_Editor*).

Appendix E

Command Line Compilation Scripts

The Xilinx partial configuration web page has an example design called `Calc_pr10`, containing complete compilation scripts to run the design through the flow. The top-level script, `run_flow.cmd` (UNIX) and `run_flow.bat` (PC) executes lower-level scripts for each main phase (initial budgeting, active module, final assembly) of the implementation process.

Appendix F

Special Cases in the Bitstream Architecture

The following are some special cases in the bitstream architecture:

1. Programming information for specific Virtex, Virtex-E, Spartan-II, and Spartan-IIE pins are located in the corners of the device and would be programmed by any partial design that included that corner in its column area.

   CCLK, DONE, M0, M1, M2, PROG, TCK, TDI, TDO, TMS

   Through `bitgen -g` options, CCLK, DONE, M0, M1, M2, PROG, TCK, TDI, TDO, TMS, and any "unused" pins can be configured pullup/pulldown for all Virtex and Virtex-II series devices.

   The same is true for specific Virtex-II and Virtex-II Pro pins:

   CCLK, DONE, HSWAP_EN, M0, M1, M2, Powerdown, PROG, TCK, TDI, TDO, TMS, and any "unused" pins.

   These `-g` options must be set identically for each module that uses `bitgen` to create partial bitstreams, as well as on any full bitstreams that are produced from the final assemblies.

2. Certain `bitgen -g` option settings result in the setting of programming information in the corners of the device.

   For Virtex and Virtex-E devices, the bitstream `userid` and the programming to modulate delay values for the DLLs (GCLKDEL0 through GCLKDEL3) are stored in this manner.

   For Virtex-II and Virtex-II Pro devices, the power option for the DCM, `DisableBandgap`, is stored in this manner.

   These `-g` options must be set identically for each module that uses `bitgen` to create partial bitstreams, as well as on any full bitstreams that are produced from the final assemblies.

3. For any IOBs requiring a V_{REF} or V_R pin, the V_{REF} or V_F pin cannot be in the same bitstream area. If two modules include the same I/O bank, then during active module implementation, it is possible for each module to specify IOB standards that do not result in banking rules violations, but will create banking rules violations when the modules are combined during final assembly. DRC alerts the user to these violations, so it is important for the designer to run final assembly on all combinations of modules to determine if there are any banking rules violations.

4. For Virtex-II and Virtex-II Pro devices, there are programming bits in the corners of the devices to control the DCI operation for IOBs that require a V_R. The bits in each corner control the DCI IOB standards in the V_{REF} banks that touch it (there are eight banks, so two banks touch each corner). If a reconfigurable module defined an IOB standard that

required certain V_R and V_F pins, but one of these pins was not in the reconfigurable module's configuration area, but in another module's reconfigurable area, then a banking rules violation would occur during final assembly, when the two modules are combined This will also be picked up by DRC banking rules checking at final assembly, so it is again important that every combination of modules be run through final assembly to check for banking rules conflicts.

5. For Virtex devices, the feedback connections from the clock resources to the DLLs can be programmed to use dedicated clock resources, or generic routing. For a given DLL, that programming is located above the block RAM columns on the edge of the device in the same quadrant as the DLL. If a clock used in a module is using the DLL in a particular way, that programming must be identical for every module that includes that corner of the device.

Appendix G

Static Partial Reconfiguration

This document is primarily devoted to the capability of "active" partial reconfiguration. That is, a module can be reconfigured while the remainder of the device remains active. In contrast, static partial reconfiguration is done before the device is fully active or when the device is inactive. This can be accomplished by deasserting the chip select (CS) during configuration, for example, to load in special data. For the partial reconfiguration to take place, the rest of the device is in shutdown mode and is brought up again once the reconfiguration is completed.

Static partial reconfiguration works with all Virtex and Virtex-E devices, but does not for Virtex-II ES (Engineering Sample) devices. Xilinx recommends using active partial reconfiguration for Virtex-II designs, because of the GHIGH programming problem found in the Virtex-II ES devices. Active Partial Reconfiguration is ensured by invoking BitGen with the `-g ActiveReconfig:Yes`.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/17/02	1.0	Initial Xilinx release.