Partial Configuration Design and Implementation Challenges on Xilinx Virtex FPGAs

Christophe Bobda, Ali Ahmadinia, Kurapati Rajesham, Mateusz Majer
University of Erlangen-Nuremberg
{bobda,ahmadinia,mateusz,teich}@cs.fau.de

Adronis Niyonkuru
University of the Federal Armed Forces Hamburg
niyonkuru@hsu-hh.de

Abstract

In this paper, we address the main aspects of partial reconfiguration on the Xilinx Virtex FPGAs and explain how to overcome main challenges during partial reconfiguration design and implementation, in specific about signal integrity, global logic and inter-module communication. In addition, we discuss the problem of designing partial reconfigurable application using the HandelC language, and illustrate this approach by using one example in video rendering.

1. Introduction

Xilinx is the one of a few reconfigurable device producers in the market. A family of its FPGAs (Virtex Series) provides an important feature called partial reconfiguration [Xil04b]. It is the ability to reconfigure a distinct portion of the FPGA while the remaining is still in operational. Partial reconfiguration is done when the device is active. Except during some cases of inter design communication, certain areas of the device can be reconfigured while other areas still in operational and unaffected by the reprogramming.
Partial reconfiguration has opened doors for some applications that require the downloading of different designs' bitstreams into the same area of the device or the flexibility to modify the parts of a design without having to either reset or completely reconfigure the entire device. Commercially available products that support partial reconfiguration from Xilinx are: Virtex, Virtex-2 and Virtex-2 Pro [Xil03]. But the application notes do not cover all aspects, and many of the tricks that are needed to acheive partial reconfiguration, is neglected. Recently, in [BBHN04] some of the points are explained. Here, we cover more ambiguous issues, that are very helpful for the dynamic reconfiguration implementation.
There are two main styles of partial reconfiguration: Difference Based and Module Based [Xil04b].

1.1. Difference Based Partial Reconfiguration

This method of partial reconfiguration is accomplished by making a small change to a design, 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.
This method can not be adapted, because the changes to the design are not small and the signal integrity is not insured on reconfigurable modules boundaries. That's why, we will not cover this method in detail. Instead module based partial reconfiguration is discussed here.

1.2. Module Based Partial Reconfiguration

This method is based on Modular Design Flow. Modular design is a Xilinx Development System Option [In02]. This feature allows a team of engineers to work independently on different modules of a design and merge them into one FPGA design. Each module can be designed by a team member while the team leader still working on the top level design. This parallel development gains in time saving and allows modifying a module while leaving other unaffected, because each module is treated as an independent module. This method is extensively used in partial reconfiguration of Xilinx devices.

The complete design can be divided into modules and each of these may be independent. If all modules are completely independent, i.e. no common I/O except clocks then there is no need to use any bus macro for inter-module communication. The bus macro provides a fixed "bus" of inter-design communication.
However, for modules that do communicate with each other, a special bus macro allows signals to cross over a partial reconfiguration boundary. 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. Without this special consideration, inter-module communication would not be feasible as it is impossible to guarantee routing between modules. Each time partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections.

2. Reconfigurable Module Overview

Reconfigurable module is a module to be accommodated on a distinct portion of an FPGA. This portion can be reconfigured while the rest of the device in active operation.
Reconfigurable modules have the following properties:

1. Reconfigurable module must be constrained to the full height of the device.
2. The width of the reconfigurable module must be a multiple of four slices, i.e., minimum of four slices and maximum of the full-device width. That means a module's left most placement must always be x = 0, 4, 8, ...
3. All logic resources covered by the reconfigurable module are considered part of the reconfigurable module's bitstream "frame". It includes slices, TBUFs, block RAMs, multipliers, IOBs, and most importantly, all routing resources except clocking logic.
4. Reconfigurable modules must not directly share any signals with other modules, except clocks. This includes resets, constants (VCC, GND), enables, etc. Bitstream frames for global clocks are separate from those bitstream frames defining the CLBs. So clocking logic (BUFGMUX, CLOCKIOBs) is always separate from the reconfigurable module.
5. IOBs immediately above the top edge and below the bottom edge of a reconfigurable module are part of the specific reconfigurable module's resources.
6. 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.
7. A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed.
8. Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special bus macro.
9. 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 logic may be required.
10. The state of the storage elements inside the reconfigurable module are preserved during and after the reconfiguration process.

2.1. 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. To manage different implementations of modules, a directory structure would be useful. Xilinx recommends a structure of design directories shown in Fig.1, can be used for modular design implementation as follows

- " HDL Design" directory (*HDL*): In this directory the top level design and modules HDL source code is placed.
- "Synthesis" directory (*ISE*): In this directory the team leader synthesizes the top-level design, which includes the appropriate HDL file, the project file, and project directories. In the same way, each team member will also have a local synthesis directory for his or her synthesized module.
- "Modules" directory (*Modules*): In this directory, the active implementation of individual modules is done in the respective module's directory.
- "Implementation" directory (*Top*): It includes the directory for initial budgeting and final assembly. In the initial directory the team leader sets up initial budgeting phase for the design. In final assembly directory, the team leader assembles the top-level design and implemented in-

dividual modules from PIMs (Physically Implemented Modules) directory into the final design.

- ”PIMs” directory (*Pims*): The active implementation of individual module creates appropriate module directory in PIMs directory where implemented module files are copied.

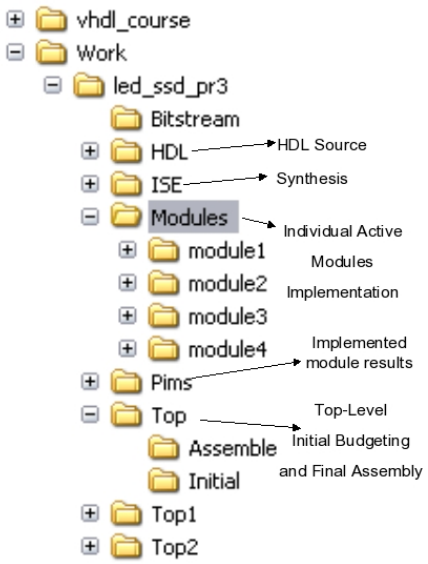

Figure 1. Modular design directory structure

After constructing directories, the following steps should be performed:

1. Design Entry and Synthesis - To conform to the requirements of partial reconfiguration, the HDL coding and synthesis process follows some general structural rules. These guidelines are very similar to those specified for the modular design flow.
2. Initial Budgeting - Design the floorplan, constrain the logic, and create timing constraints for the top-level design and each module.
3. Run Active Implementation (NGDBUILD, MAP, PAR, etc.).
4. Assembly Phase Implementation.
5. Verify design (static timing analysis, functional simulation).
6. Visually inspect design using FPGA-Editor to ensure no unexpected routing crosses module boundaries.
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.

3. Implementation Approach

3.1. Bus Macros

In the non partial reconfiguration flow communication wires between the modules are all routed together which means that they can take an arbitrary path. In the partial reconfiguration flow, the modules are routed unaware of the other modules. In order to establish a contact between them the two modules must use the same physical wires. This is exactly what the bus macro provides. Bus macro is nothing but a fixed ”bus”. This can be used to connect the signals used as communication paths between reconfigurable module and another (fixed or reconfigurable) module. The routing resources used for bus macros are completely fixed and static, i.e. there is no change in the fixed routing resources used for the inter module signals while a module is reconfigured.
Bus macro in Virtex device is limited for 4 bits of communication between two modules. 3-state buffers (TBUFs) are used to implement the bus macros. 8 TBUFs are hooked up in an arrangement that allows 4-bits information to travel either left to right or right to left. So required number of bus macros are instantiated to reach the data width of the signals crossing the boundary of the reconfigurable module. Xilinx recommends not to change the direction once it is defined (either left to right or right to left) for that particular FPGA design. There is also a limitation to use more number of bus macros as needed in a design as this number depends on the availability of horizontal long-line routing resources in a CLB tile of a particular device.
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. In the top level design, 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, Virtex-II, and Virtex-II Pro series specific versions of the bus macro. Be sure to instantiate the version compatible to the chosen device [Xil04a].

3.2. AREA_GROUP Constraint Attribute

The following constraints need to be manually inserted into the top-level design UCF to enable partial reconfiguration functionality. This optional 'attribute' of an AREA_GROUP constraint can be used to specify that a given module is going to be used for partial reconfigurability therefore

- its area should be extended to include all device resources that are part of the same configuration frames
- its internal routing should not consume device resources that lie outside of the boundary of the defined region.

For each module defined "area group", reconfiguration property must be attached to properly handle the design for the partial reconfiguration flow. The format of this constraint is as follows:
AREA_GROUP *module_name* RECONFIG_MODE ;

3.3. Bus Macros Location Constraints

A separate location constraint must be created for each bus macro and they must be entered into the .ucf file. The format of this constraint is as follows:
INST "*BM_BusMacro*_1 "LOC = "TBUF_X0Y8" ;
Hint: Must be on 4-column boundary x = 0, 4, 8, etc.

4. Designing for Partial reconfiguration with HandelC

The structure of the directory previously described is the same. Using HandelC, the goal is to provide the implementation of top-levels designs and separately the implementation of each module. Therefore, the handelC code must be divided in codes for modules each with its own interface. The integration is then done by connecting the modules together using signals in the top-level design as shown in Figure 2. For each module to be reconfigured later, a separate top-level must be produce. The transition from one design to the next one will then be done later using either full reconfiguration or partial reconfiguration with modules representing the difference from one top-level to the next one. We explain this modular design for HandelC by

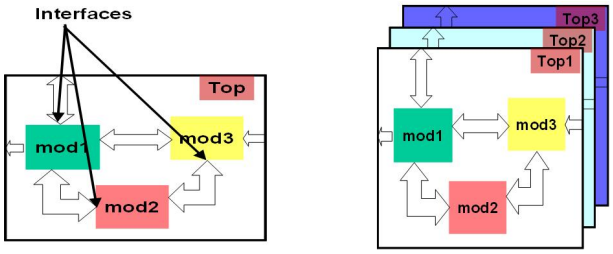

Figure 2. Modules implementation in top-level designs

mean of one example provided below.

Example:

The design consists of an adder in its first configuration and it is partially reconfigured to act as a subtractor. Both modules (adder and subtractor) have to be implemented separately. The following listing shows a HandelC description of an adder:

```
void main()
{
unsigned 32 res;
interface port_in(unsigned 32 var_1 with
{busformat="B<I>"}) Invar_a();
interface port_in(unsigned 32 var_2 with
{busformat="B<I>"}) Invar_b();
interface port_out() Outvar(unsigned 32
Res = res with {busformat="B<I>"});
res = Invar_a.var_1 + Invar_b.var_2;
}
```

The first part of the code is the definition of the interfaces for communication with other modules and the second part realizes the addition with the input values coming from the input interface and the output values sent to the output interface. We need not to provide the implementation of the subtractor, since it is the same like that of the adder, but instead of adding we subtract.
Having implemented the two modules each module will be inserted in a separated to-level. Up to the use of the adder or subtractor, the two top-levels are the same. The design can now be reconfigure later to change the

adder against the subtractor. Connecting the module in the top-level design is done as follows:

```
unsigned 32 operand1, operand2;
unsigned 32 result;
interface adder (unsigned 32 Res)
my_adder(unsigned 32 var_1 = operand1,
 unsigned 32 var_2 = operand2)
with {busformat="B<I>"};


void main()
{
operand1 = produceoperand( 0);
operand2 = produceoperand( 3);
result = my_adder.Res;
dosethingwithresult(result);
}
```

According to the top-level design being implemented, the adder will be replaced by a subtracter. The next question is how to keep the signal integrity of the interfaces. Since there is no bus macro available in HandelC, bus-macros provided by Xilinx must be used as VHDL-component in a HandelC design. For instruction on integrating a VHDL code in HandelC, consult the Celoxica HandelC refernce manuals. Bus Macros are provided with the Xilinx application note on partial reconfiguration [Xil04b]. Before setting constraints on a HandelC design, the designs have to be compiled first. Afterwards, the resulting EDIF-files must be opened with any text-editor and the longest pattern that contains the name of the module as declared in the top-level module is selected. This is useful because the HandelC compiler generate EDIF-code and automatically adds some characters to the module name used in the original design. If the original module name is used it will not be recognized in the following steps of the Modular Design Flow. With the EDIF for the modules and that of the top-level designs we now have all what we need to run the modular design flow explained in Section 2.

5. Experience on the Celoxica RC200 board

Since all the interfaces and pin-locking of the RC200-board are hard-coded in the Celoxica-PSL-library, it is not possible to set the required pin-constraints after the design has been compiled to EDIF. In order to overcome this issue, we first compiled our design in VHDL. In the resulting entities the top-level ports are named with the pin-names of the given device, thus providing the necessary entry in the UCF-file. Note that the pins are not optimally (in term of partial reconfigurability) placed on the RC200-board [Cel03].

6. Practical Demonstration

We have implemented a reconfigurable video rendering application on the RC200 board from Celoxica. The board features a Xilinx VirtexII-1000 FPGA. In our experience, each top-level design consists of two modules. The first one is a VGA controller connected to a VGA interface through the device pins. The second module is a colour producer having no connection to the outside word. The VGA and the colour producer communicate using 24 bit data in each direction. The VGA always send the current pixel position of its scan pointer to the colour producer which in turn returns the value of the colour to be displayed at the current position. The width of the colour data is 24 bits. Each coordinate (X and Y) of the pixel have a width of 12. Therefore the values exchanged between the modules are 24 bits data in each direction. Using the pixel position the colour producer computes colour accordingly thus producing a visual effect on the screen. The reconfiguration process can be use to change the kind of pattern to be produced on the screen. However, using partial reconfiguration, the VGA monitor will always be reset, thus stopping the display process until the end of the reconfiguration. With partial reconfiguration, we need only to change the colour producer part. The VGA part will keep running to avoid the system to be interrupted. This application clearly shows the advantage of reconfiguration on critical application, where resetting the system may have fatal consequences. Therefore it is very important to keep the system running and just partially reconfigure for a new computation. We faced one problem in the design process. The VGA pins are spread on the bottom part of the device from column 3 to the end of the device. According to the rule that each module is assigned all the resource in its placement area, the VGA module will have to be placed from column 3 to the end of the device., since it uses all pins below the device in that area. Only two columns (1 and 2) are left for module implementation. Since our module could not fit on those two columns we placed it right to the VGA module and feed the VGA pins trough the module. Bus macros are used between the two modules to keep the integrity of the signal during the reconfiguration process. The design architecture of each top-level produced is shown in Figure 3. We generated three top-levels with different algorithms for the colour producer. The partial reconfigu-

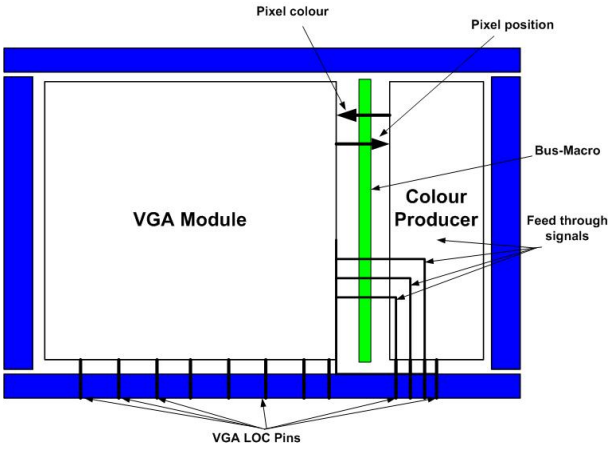

Figure 3. Implementation of a color rendering application

ration was successfully managed using the JTAG interface with Xilinx IMPACT tool. Note that it is not possible to download partial reconfigurable designs on the Celoxica board using the FTU utility provided with board. Instruction on connecting the JTAG cable on the RC200 board is available only in the latest version of the RC200 manual which can be downloaded from the Celoxica home page.

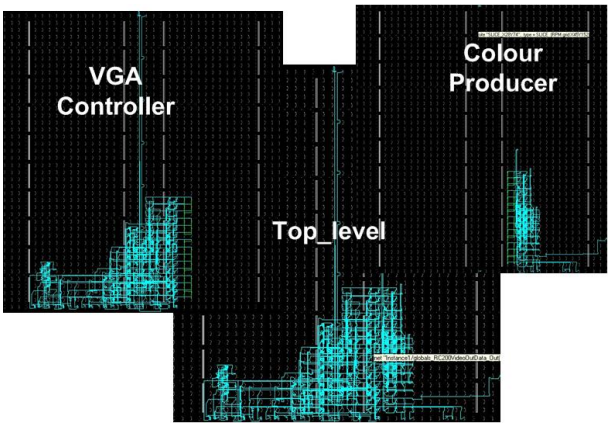

Figure 4. Implementation of the partial reconfigurable color producer

7. Conclusions

Partial and dynamic run-time reconfiguration offers new possibilities for designs with Xilinx Virtex-II FPGAs. In this paper we have presented the desired design flow for partial reconfiguration in general. We then address the design of partial reconfigurable circuit on the RC200 board of Celoxica using the HandelC Language. This paper provides design opportunity for partial reconfiguration not only to the growing community of HandelC designers, but also SystemC designers. One example of the benefit for using dynamic reconfiguration is the possibility to use smaller FPGAs by outsourcing configuration data. Other aspects are the adaptivity of systems to the demand of e.g. applications or environment. This technique opens a great field for investigation and the development of new systems.

References

[BBHN04] Blodget, B., Bobda, C., HÃ¼bner, M., und Niyonkuru, A.: Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs. In: *Field-Programmable Logic and Applications, International Conference FPL*. S. 801–810. 2004.

[Cel03] Celoxica Inc.: *RC-200 Hardware, Software and Function Reference Manual*. 2003.

[In02] Inc., X.: *Development System Reference Guide, Chapter 4, Modular Design*. 2002.

[Xil03] Xilinx Inc.: *Virtex Series Configuration Architecture User Guide*. 2003.

[Xil04a] Xilinx Inc.: 2004. http://www.xilinx.com/bvdocs/appnotes/xapp290.zip.

[Xil04b] Xilinx Inc.: *Two Flows for Partial Reconfiguration: Module Based or Difference Based*. 2004.