Friedrich-Alexander-Universität
Erlangen-Nürnberg

Designing Partial and Dynamically Reconfigurable Applications on Xilinx Virtex-II FPGAs using HandelC

C. Bobda, M. Huebner, A. Niyonkuru,
B. Bloget, M. Majer, A. Ahmadinia

bobda@cs.fau.de, huebner@itiv.uni-karslruhe.de, niyonkur@hsu-hh.de
Brandon.Blodget@xilinx.com, mateusz@cs.fau.de, ahmadinia@cs.fau.de

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany

Co-Design-Report 3-2004

December 6, 2004

Abstract. In this paper, the problem of designing partial reconfigurable application using the HandelC language is investigated. We address the main aspects of partial reconfiguration on the Xilinx Virtex FPGA and explain how to deal with HandelC designs. We illustrate this approach using one example in video rendering. We also provides all the files used in this tutorial to make the design process understandable. This tutorial is not a HandelC tutorial. We assume that the designers have the necessary HandelC knowledge to understand the code we provide.

1. Introduction

In the last, decade has experienced an increase in performance and flexibility of Field Programmable Gate Arrays (FPGAs). FPGA have grown from simple glue logic elements to complex devices able to implement several complex hardware applications. Modern state-of-the-art FPGA devices like Xilinx Virtex FPGAs [1] additionally support partial dynamic run-time reconfiguration which reveals new aspects for the designer who wants to develop future applications demanding adaptive and flexible hardware. Especially in the domain of mobile computing high-end mobile communication applications will benefit from the capabilities of the new generation of reconfigurable devices. There exist some new approaches deploying Virtex/Virtex-II FPGAs in multimedia applications using their capabilities for a dynamic function-multiplex showing new ways for the efficient deployment of partial run-time reconfiguration [2] [4]. A new approach to create systems that are able to manage configuration is run-time reconfigurable systems. These systems use the flexibility of an FPGA by changing the configuration partially [3]. The only necessary functions are configured in the chip's memory. A function can be replaced by another function while other parts stay operative. To solve the problem of substitution and I/O management, the configuration needs a main module to control the tasks. With such a system it is possible to save resources like output pins and energy because of outsourcing configuration data [4]. The need of chip area becomes smaller and therefore the power consumption can be reduced. Nevertheless the power requirements of such applications will grow with increasing rate of configuration [5]. Creating such a system has two aspects: reducing amount of chip area and reducing power consumption by designing a control system which manages the content of FPGAs configuration in an intelligent way to minimize reconfiguration rate. Additionally this management can control the on chip communication bus to prevent an overhead of bus size. The development of partial reconfigurable application on the Xilinx platform is done according to the "modular design flow" provided by Xilinx in [11]. The modular design flow provides a guided flow on how to generate the constraints required by the placement and configuration tools for the generation of full and partial bitstreams representing modules to be downloaded at run-time to reconfigure the device fully or partially. Despite their usefulness, all tutorials provided up to now target only VHDL and Verilog designs. With the increasing interest in software-like hardware design language like SystemC and HandelC, the need for incorporating those languages in the modular design flow process is high. This tutorial addresses the design of partial reconfigurable applications written in HandelC. By mean of one simple example, we explain how to express modular designs in HandelC. The rest of the paper is organized as follow: In Section 2 , we briefly describe the internal structure of Virtex FPGA that are targeted in this tutorial, Section 3 briefly explains the main steps of the modular design flow. In Section 4, we

explain how to provide the placement constraints for components while Section 5 addresses some troubleshooting that we have experienced and provided some hint to solve errors. In section 6 we explain how to design for partial reconfiguration using HandelC and present our experiences with the RC200 board from Celoxica in section 7. We explain the design of a reconfigurable video rendering module which sources are provided with this tutorial in section 8. Finally, Section 9 concludes the work.

2. Internal Structure of Xilinx Virtex FPGAs

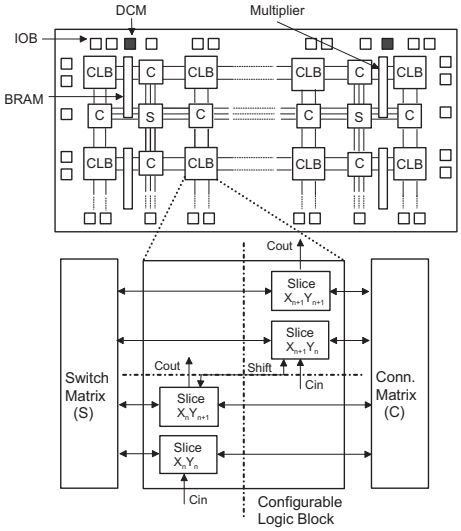

Figure 1: Architecture of Xilinx Virtex FPGAs

The key-component of each Xilinx Virtex FPGA is the CLB, which consists of four slices each of them providing two 4-input function generators, carry logic, arithmetic logic gates, function multiplexers and two storage elements (figure 1) [10][12]. The four slices are directly linked to the global switching matrix and to the fast connection matrix, so that routing is done between slices and not at the CLB-level. Thus, the view of a Virtex FPGA using e.g. Xilinx Floorplanner shows the slices available. When implementing partial reconfiguration, it is important to place reconfigurable modules horizontally on a four-slice boundary. This means that the leftmost slice number ('X'-index) for any reconfigurable module must be 0, 4, 8… Vertically, the reconfigurable module fits into the full height of the device [11]. Section 4 (Placement Constraints) provides more details how to do it.

3. Modular Design Flow

The Modular Design Flow is a method generally used to enable partial and dynamic reconfiguration. Originally it is intended to allow many designers to work on the same design project in parallel. Xilinx recommends it to implement partial reconfiguration [11]

When implementing a partial reconfigurable design using this technique the first step – as in any hardware design project - consists of the design entry. This can be done using any established design entry method (Schematics, HDLs, etc.). Before enabling partial and dynamic reconfiguration one has to verify that the required functionality is well defined and implemented. Thus, only after a successful functional simulation, synthesizing the design can be started. Synthesis for a Modular Design Flow is done separately for the top-level designs and their fixed and reconfigurable modules

One specific requirement for partial reconfiguration is that they must be as many top-level design projects as the number of reconfigurable modules planned; each top-level includes a different alternate reconfigurable module. The top-level designs include fixed modules and reconfigurable modules all instantiated as "black boxes". The corresponding attribute for VHDL-design entry is: "*attribute box_type of <module name>: component is "BLACK_BOX*" which is defined in the Xilinx *unisim* library. Communication between fixed and reconfigurable modules is implemented by instantiating as many bus macros as required to assure signal integrity during partial reconfiguration. Therefore, the bus macro component and its implementation file (*bm_4b_v2.nmc*) are provided. Each top-level design can be handled as a single design project and therefore synthesized in a separate directory.

In order to get a clear but also essential organization of the design files, Xilinx recommends to set up a well defined directory structure. An example of such a structure can be found in Xilinx Application Notes 290 [11]. Thus, the project files corresponding to each top-level design are stored in a separate directory. If for example the design project includes three reconfigurable modules, there will be accordingly three top-level designs and three different directories (*…\Top1, …\Top2, …\Top3*). Each of these directories has two subdirectories:

- *Initial* : stores files used to implement the top-level design in its "initial budgeting phase"
- *Assemble*: stores files used and created during the "final assembly phase"

Furthermore, it is recommended to have a directory where to store design entry files (...*HDL*) and another one for synthesis files (…*Synth*). A further important directory is the ...*Module* directory with as many subdirectories as the number of all possible modules (reconfigurable and fixed). These subdirectories store the implementation files for each module which are created in the "active module implementation phase". The last directory required is the …*Pims* directory which stores the physically implemented modules, i.e. placed and routed implementation files for each module.

After all these prerequisites have been fulfilled, it is time to begin with the Modular Design Flow properly. It consists basically of three phases:

- Initial budgeting phase
- Active module implementation
- Final assembly phase

The initial budgeting phase has to be run for each top-level design. In this phase, the NGC file that has been created after the synthesis of the top-level design is used within the following command:

ngdbuild -p <target device> -modular initial <top-level file>.ngc

The goal of the initial budgeting phase is to position all modules, top-level logic and top-level I/O ports on the target device. Therefore, a top-level constraint file which defines area constraints and all additional constraints specific to partial reconfiguration has to be placed into each top-level directory. The result of this phase is an NGD file with all modules instantiated as unexpanded blocks, I/O ports and top-level logic.

The next phase consists of the active module implementation. During this phase, each module (fixed and reconfigurable) is implemented separately all resulting files are stored its corresponding subdirectory of

the …*Module* directory. First of all, each module is synthesized with the Xilinx ISE option "*Add I/O Buffers*" disabled. The resulting NGC file is used together with one top-level NGD file including this module being actively implemented and with the top-level constraint file within the following commands:

1. Expand one active module into the top-level design and apply top-level constraints:
   ngdbuild –p <target device> -modular module -active <module name> ...\top\Initial\<top-level file>.ngc
2. Map the top-level design with only one module being implemented actively:

map -u -pr b <top-level file>.ngd –o <top-level file>_map.ncd <top-level file>.pcf

3. Place and route the design:

par -w -ol 5 -n 3 -s 3 <top-level file>l_map.ncd mppr.dir <top-level file>.pcf

4. Copy the resulting placed and routed file from the subdirectory *\mppr.dir\4_4_3* to the current module directory:

copy mppr.dir\4_4_3.ncd <top-level file>.ncd

5. Copy this file which defines a set of options to be considered by the *bitgen* utility to the current directory:

copy ..\..\bitgen_v2_jtag.ut

6. Create the bistream file using predefined options

bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes <top-level file>.ncd

7. Run the *trace* utility to verify top-level timing constraints

trce <top-level file>.ncd <top-level file>.pcf

8. Publish the implemented design to the …*Pims* directory

pimcreate -ncd <top-level file>.ncd -ngm <top-level file>_map.ngm ..\..\Pims

After each module has been successfully implemented and published to the …*Pims* directory, the latter then includes as many subdirectories as the number of modules implemented. Thus, the final assembly phase can be started by putting together the implemented modules and the different top-level designs. This assembling is done in the …*Assemble* subdirectory corresponding to each top-level design. Therefore, the top-level design files created during the initial budgeting phase and the top-level constraints file are copied into this directory. Afterwards, followings commands have to be run successively:

- *ngdbuild -p <target device > -u -modular assemble -pimpath ..\..\Pims <top-level file>.ngc*
- *map -u -pr b <top-level file>.ngd -o <top-level file>_map.ncd <top-level file>.pcf*
- *par -w <top-level file>_map.ncd <top-level file>.ncd <top-level file>.pcf*
- *copy ..\..\bitgen_v2_jtag.ut .*
- *bitgen -d -f bitgen_v2_jtag.ut <top-level file>.ncd*
- *trce <top-level file>.ncd <top-level file>.pcf*

4. Placement Constraints

Placement constraints are one of the central points in the partial reconfiguration. They are used exclusively to allocate device area for modules in a given configuration or bitstream. The placement constraints can be done only on the basis of the module's size which can be estimated after compiling the top-level design containing the module as described in the initial budgeting phase. The Xilinx PACE tool can then be used to graphically visualize the module. This is helpful since it provides a good visual estimation on the bounding-box in which the module can be placed. Use the PACE toolkit to constraint the module to be placed in the estimated bounding-box to a given location.

PACE generates (or modifies, if there is one) an ".ucf" file in which the placement constraints are described in the correct notation (e.g. ***AREA_GROUP = usergroup RANGE=SLICE_X3Y1: SLICE_X33Y33*** for the Virtex II and II-Pro and ***AREA_GROUP = usergroup RANGE=CLB_R3C1: CLB_R33C33*** for the Virtex, Virtex E and Spartan).

It is very important to note that all the pins used by a given module should be locked in the area (set of columns) occupied by that

module. Otherwise, bus macros should be used to keep the integrity of the signals across module boundaries, especially when driving a signal from one module to a configurable one. Most designers spend 80% of the time on solving such issues which are really not related to the partial reconfiguration itself. Unfortunately, most of the boards on the market were not designed for partial reconfiguration purpose. We recommend studying the pin assignment on a board and only buying it, if it matches the computation-flow of your application.

Example of placement constraints

```
INST "Instance0" AREA_GROUP = "AG_Instance0";
AREA_GROUP "AG_Instance0" RANGE =
SLICE_X0Y0:SLICE_X3Y29;
AREA_GROUP "AG_Instance0" MODE=RECONFIG;
```

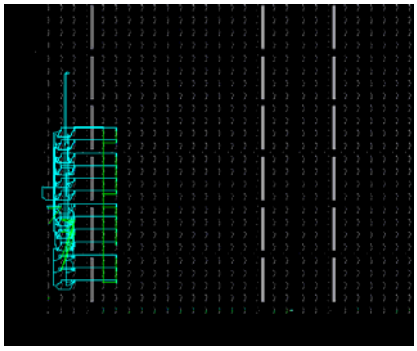

Figure 2: Architecture of Xilinx Virtex FPGAs

1. JBits

JBits [1] is an Application Program Interface (API) to the Xilinx configuration bitstream. It is designed for dynamically modification of Virtex-II bitstreams. JBits contains Java classes which allow having access to internal resources by modifying parts of a partial or complete configuration bitstream. Modification of bitstreams generated by Xilinx design tools or bitstreams read back from a device is possible. Designers are able having access to resources (e.g. CLBs, IOBs, Block RAM and PIPs) and may modify them before writing the bitstream

back to the FPGA. Because of the fast runtime of JBits, changes can be done very fast and maybe even on runtime.
As an example JBits can be used to extract partial configuration data from a complete bitstream. This can be done after the modular design flow (see section 3) to generate partial configuration data. These data can be stored into external memory and used for partial reconfiguration by reading out external Flash memory and sending the data to the Internal Configuration Access Port (ICAP) or the SelectMAP™ interface (see section 6). In [5] this technique is used within the reconfigurable system using static design flow (figure 2). In this example JBits is only used to extract data from an existing bitstream without modification of the data.

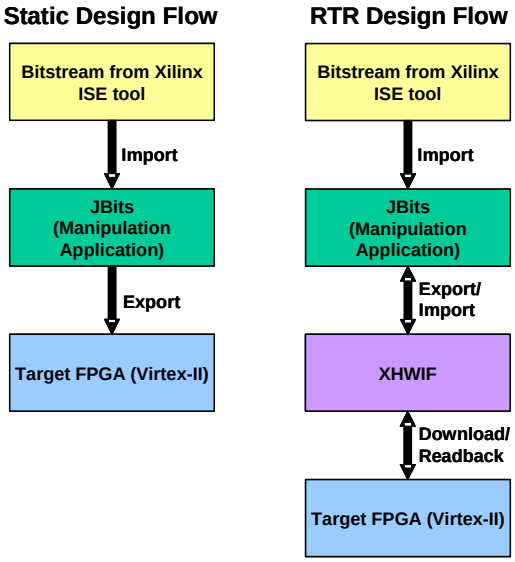

Figure 2 Designflow with JBits

Using the Java Run Time Reconfiguration (RTR) design flow (figure 2), the configuration of the FPGA can be modified while run-time. An application uses JBits for manipulating existing or read back configuration data. Modifications were done and exported to XHWIF (Xilinx HardWare InterFace) for communication to the hardware.
The XHWIF interface can be used for communication with a FPGA board. It provides several methods to describe the FPGA based board and to send data to and off the board. The powerful API XHWIF adds a layer of abstraction to the hardware. This enables the simple porting of

applications to new hardware. XHWIF also provides to debug the hardware while run-time.

2. ICAP Interface

ICAP is an acronym for Internal Configuration Access Port. This component was introduced in the Xilinx Virtex-II devices. It is also present in Xilinx Virtex-II Pro devices. The ICAP provides configuration access to the FPGA logic. This in essence enables self-reconfiguration of Virtex-II devices. Care obviously must be taken when using the ICAP not to reconfigure the circuitry that is controlling the ICAP. Thus the ICAP does not allow full reconfigurations, only partial reconfigurations.

The ICAP interface is a subset of the SelectMAP™ Interface. The process for configuring the device and reading back from the device using the ICAP is essentially the same as it is for SelectMAP. Xilinx application notes 138 and 151 describe the Virtex™ series configuration architecture and the SelectMAP interface [10][11]. Figure 2 shows a comparison between the SelectMAP and ICAP interfaces. Table 1 describes the ICAP ports.

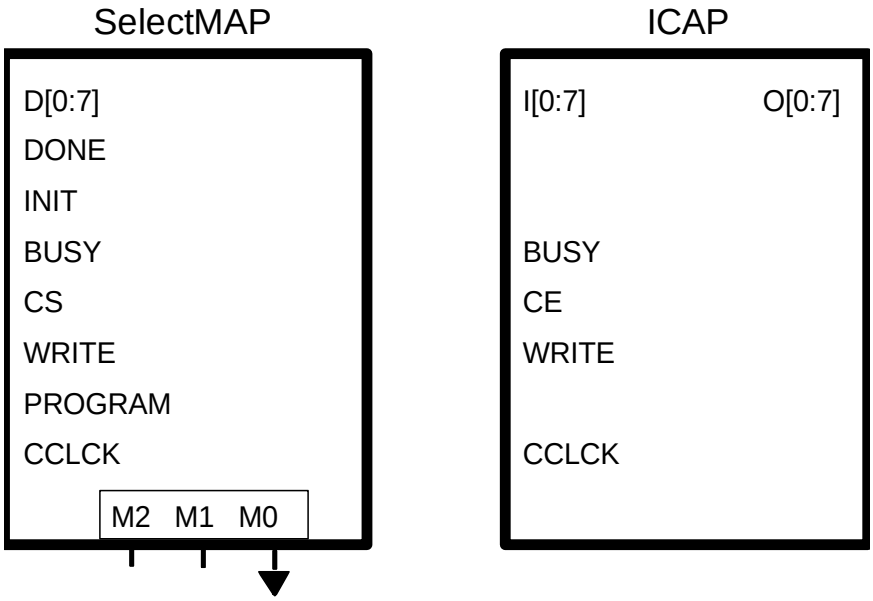

Figure 3 – The SelectMap and ICAP interfaces

ICAP Port	Description
I[0:7]	The I0 through I7 pins function as an input data bus to the ICAP port. Configuration data is written to this bus. The I0 pin is considered the MSB bit of each byte.
O[0:7]	The O0 through O7 pins function as an output data bus from the ICAP port. Readback data is read from this bus. The O0 pin is considered the MSB bit of each byte. This bus also provides status information while the device is being configured.
BUSY	The BUSY output indicates when the FPGA can accept another byte. If BUSY is Low, the FPGA reads the data bus on the next rising CCLK edge where both CE and WRITE are asserted Low. If BUSY is High, the current byte is ignored and must be reloaded on the next rising CCLK edge when BUSY is Low.
CE	Enables the ICAP port. Although not specified in the Libraries guide, the CE pin is active low by default. FPGA Editor shows that it is possible to invert this signal to make it active high. This pin is equivalent to the SelectMAP \CS pin.

WRITE	Like the CE pin the WRITE pin is active low by default. When asserted Low, the WRITE signal indicates that data is being written to Data Input bus (I[0:7]). When asserted High, the WRITE signal indicates that data is being read from Data Output bus (O[0:7]). FPGA Editor shows that it is possible to invert this signal to make it active high. This pin is equivalent to the SelectMAP \WRITE pin.
CCLK	The CCLK signal synchronizes all loading and reading of the data bus for configuration and readback.

Table 1

The ICAP interface is simplified because it only has to support partial reconfigurations. It does not have to support full configurations or multiple modes like SelectMAP. The pins that are missing from the ICAP interface are the mode pins (M2, M1, M0), DONE, INIT, and PROGRAM. The SelectMAP CS pin has been renamed CE on the ICAP. This pin still performs the same functions and it defaults to active low just like in the SelectMAP interface. The SelectMAP bi-directional D[0:7] port is split into two ports in the ICAP interface. These two ports are I[0:7] and O[0:7].

The Xilinx Embedded Developer Toolkit (EDK) provides tools for implementing FPGA designs containing embedded processors. The EDK version 6.2 contains a peripheral called *opb_hwicap*. This peripheral wraps the ICAP with additional logic that can read and write frames to a BRAM. The *opb_hwicap* interfaces with the CoreConnect™ On-Chip Peripheral Bus (OPB)[8]. This peripheral with its associated drivers abstracts away the details of the ICAP configuration interface and enables a self-reconfiguring platform [9].

5. Troubleshooting

When compiling a partial reconfigurable design, they are some troubles which may appear. We have listed some of them below and provide some tips on how to solve them.

In the initial phase
Error: At least on inactive module...
Solution: at top level, each module should be defined as black box
In the Synthesis phase when defining bus macros
Error: a default value is assigned; a signal should be assigned...
Solution: In port map of bus macro, the signal must be grouped.
Error: no pin connected for the output of bus macro
Solution: Define dummy output in the top level entity, and feed the output of bus macro to them. Update the .ucf file accordingly.
In the active phase
Error: multiple drivers...
Solution: Check the project properties of the top level designs and all modules before launching the synthesize tool: the bus delimiter must be set to (), and the I/O buffers must be disabled when synthesizing the modules.
Assembling phase
Error: The STEPPING level for this design is 0.
FATAL_ERROR: Guide: basgitaskphyspr.c:255:1.28.20.1.14.1:137…
Solution: This error is cause by a GLOBAL_LOGIC used implicitly in a sub-module, but locked in the boundary of the device. For example if a BUFG is used in a sub-module this will implicitly declare and use a global logic GLOBAL_LOGIC1_XX. Since GLOBAL_LOGIC1_XX has to be locked in top level design, a fatal error will be given back. Never declare and use a GCC or GND in the top-level module. All the constant signals (e.g. GND=0; VCC=1) needed at the top-level should be defined in a sub-module and fed to the top-level entity. Remove all the GNDs and VCCs signals declared in the sub-level modules which are not used.

6. 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 must be code must be divided in code 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 3. 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.

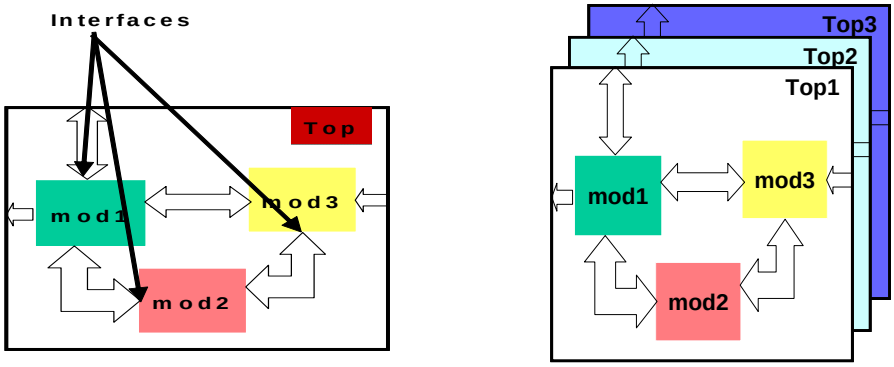

Figure 3: Modules implementation in top-level designs

We explain this modular design for HandelC by 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 follow.

```
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 [21]. 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 5.

7. 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.

8. 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.

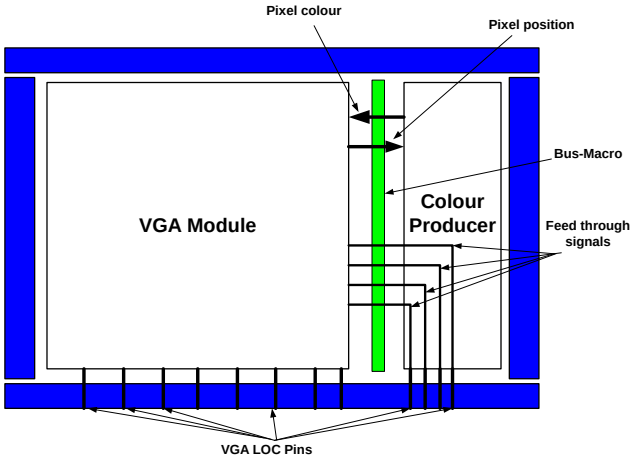

Figure 4: Implementation of a color rendering application

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 4. We generated three top-level with different algorithms for the colour producer. The partial reconfiguration 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. controler

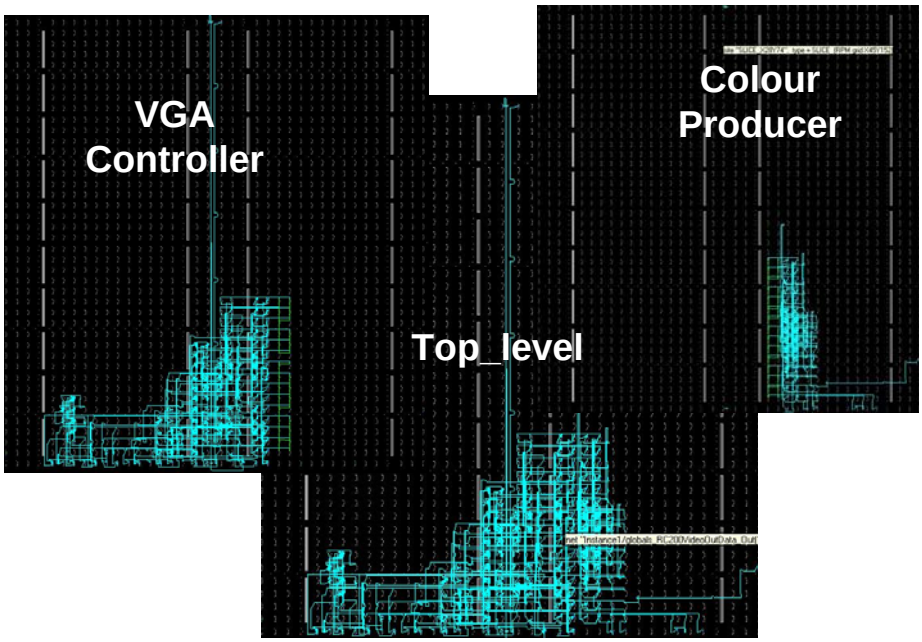

Figure 5: Implementation of the partial reconfigurable colour producer

9. 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 tutorial will provide 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 [5]. 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.

The complete sources used for this tutorial is available. For statistical purpose, interested users must send a request to:

Christophe Bobda
University of Erlangen-Nuremberg
Departement of Computer Science 12
Am Weichselgarten 3
91058 Erlangen
Email: bobda@cs.fau.de

10. References

1. www.xilinx.com
2. Y. Ha, B. Mei, P. Schaumont, S.Vernalde, R. Lauwereins, H. De Man; "Development of a Design Framework for Platform-Independent Networked Reconfiguration of Software and Hardware"; Proc. 11th Int´l Conference on Field Programmable Logic and Applications, Belfast, Ireland, 2001M.
3. J.-Y. Mignolet, S. Vernalde, D. Verkest, R. Lauwereins: “Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances”; Int´l. Conf. on Engineering of Reconfigurable Systems and Algorithms; June 25-27 2002, Las Vegas, USA
4. M. Ullmann, B. Grimm, M. Huebner, J. Becker: “An FPGA Run-Time System for Dynamical On-Demand Reconfiguration”, RAW04, Apr. 04, Santa Fé, USA
5. J. Becker, M. Hübner, M. Ullmann: “Real-Time Dynamically Run-Time Reconfiguration for Power-/Cost-optimized Virtex FPGA Realizations”, VLSI03, Darmstadt, Sep. 03
6. J. Becker, M. Hübner, M. Ullmann: “Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations”, SBCCI03, Sao Paulo, Sep. 03
7. IBM web site. http://www.chips.ibm.com/products/coreconnect (2003)
8. B. Blodget, P. James-Roxby, E. Keller, S. McMillan, P. Sundararajan. “A self-reconfiguring platform”, In Proc. of the Intern. Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal, Sept. 2003.
9. Carmichael, C.: “Virtex FPGA series configuration and readback”. Xilinx Application Note XAPP138, version 1.1, Xilinx, Inc. (1999)

10. “Virtex Series Configuration Architecture User Guide”. Xilinx Application Note XAPP151, version 1.6, Xilinx, Inc. (2003)
11. “Two Flows for Partial Reconfiguration: Module Based or Difference Based”, Xilinx Application Note XAPP290, version 1.1, Xilinx, Inc. (2003)
12. “Virtex-II Platform FPGA User Guide”, version 1.8, Xilinx, Inc. (2004)
13. http://www.xilinx.com/products/software/jbits/