Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Simon Steinegger

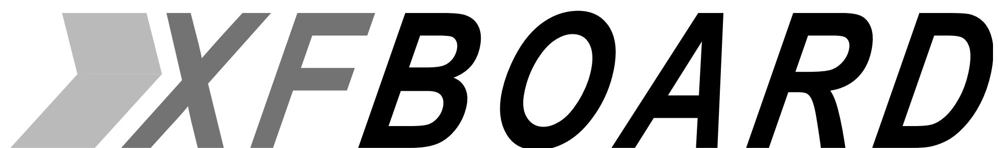
Reconfigurable Hardware OS Prototype - Part FPGA

Masters Thesis DA-2004-05
Winter Term 2003/2004

Tutor: Herbert Walder

Supervisor:
Prof. Dr. Lothar Thiele

7.5.2004



Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

Contents *iv*

List of Figures vi

Introduction *vii*

1: RHWOS Platform: XF-Board / XILINX Virtex-II *1*

1.1 XF-Board Overview 1

1.2 The Virtex-II R-FPGA 3

1.3 Clock Net Structure on the Virtex-II XC2V3000 4

2: Use and Organisation of Reconfigurable Chip Area *7*

2.1 Static vs. Dynamically Reconfigurable Area 7

2.2 Variable-Sized 1D Resource Model 8

2.3 Granular Variable-Sized Resource Model 9

3: The RHWOS Generation: XFOSGen *15*

3.1 Modular Design and Partial Reconfiguration 15

3.1.1 Modular Design Directory Structure 16

3.1.2 Modular Design Entry and Synthesis 16

3.1.3 Initial Budgeting Phase 18

3.1.4 Active Module Implementation Phase 19

3.1.5 Final Assembly Phase 21

3.2 How to Use XFOSGen 21

3.3 XFOSGen Output Files 22

4: Reconfiguration Effects *25*
4.1 Clock Distribution 25
4.1.1 What is the Problem with the Clock Distribution? 25
4.1.2 First Approach to the Clock Net Problem 26
4.1.3 The Quick Solution 30
4.2 Transient Effects 30
4.2.1 Why Do Transient Effects Occur During Reconfiguration? . . . 31
4.2.2 Ways to Deal With Transient Effects 31

5: Communication on the R-FPGA *33*
5.1 The Task Communication Bus (TCB) 33
5.2 The Bus Arbitration OS Service 35
5.3 The Bus Access Controller 36
5.3.1 The BAC - User Task Interface 36

6: Design and Implementation *39*
6.1 Experiment 1: LED Counter 39
6.2 Experiment 2: Knightrider 40
6.3 Experiment 3: Sawtooth 42
6.4 Experiment 4: Write Communication 43
6.5 Experiment 5: Loop Back 44

7: Achievements and Outlook *47*
7.1 Achievements 47
7.2 Outlook 48
7.3 Visions of FPGAs Produced for RHWOS Platforms 49

8: Acknowlegement *51*

A: An Example Synthesis Project File *53*

B: An Example Synthesis Settings File *55*

C: An Example Settings File for Bitstream generation *57*

D: BitStreamParser *59*

List of Figures

1-1 The XF-Board RHWOS Platform 2
1-2 Virtex-II Architecture Overview, [19] 3
1-3 A Virtex-II CLB Element, [19] 4
1-4 The Clock Net Structure in a Virtex-II Device, [19] 5

2-1 Bipartition of FPGA Area into Static and Dynamically Reconfigurable Area 8
2-2 Fixed-Size Task Slot Model from [3] and [12] 9
2-3 The Variable-Sized Task Model Proposed in [24], Configured with a Set of Tasks (T_1, T_2, T_3,T_x) with Different Sizes 10
2-4 Not Explicitly Configured FPGA Area Must Contain the Necessary Routing to Connect Other Tasks to the OS Frame. 10
2-5 Schematic of how Tri-State Buffers are Instantiated in a Bus Macro . 11
2-6 An Unconfigured Area with Width $w < w_{min}$, Emerging from Partial Reconfiguration 11
2-7 A Set of Different Configurations in the Granular Variable-Sized Resource Model 13

3-1 Modular Design Directory Structure 17
3-2 Physical Implementation of a Bus Macro (adapted from [21]) 18
3-3 The XFOSGen Graphical User Interface 22
3-4 XFOSGen configured to Create a Design with a Larger Right OS Frame and a User Task with Width $2 * w_{min}$ 23

4-1 Two Possibilities for Clock Net Problems Caused by Partial Reconfiguration 26
4-2 Screenshot from XILINX FPGA Editor of a Fully Activated Clock Net . 27
4-3 Different routing for signal lines through tasks 31

5-1 Task Communicaton Bus Overview 34
5-2 Finite State Machine for the Bus Arbitration 35

5-3 Finite State Machine Describing the Behaviour of the BAC for Read and Write Interfaces 37

6-1 Schematic of the LED Counter Project 40
6-2 Schematic of the Knightrider Project 41
6-3 Schematic of the Sawtooth Project 42
6-4 Timing Analysis of the Sawtooth Project (excerpt) 43
6-5 Schematic of the Write Communication Project 44
6-6 Schematic of the Loop Back Project 45

D-1 Type 1 Packet Header 59
D-2 Type 2 Packet Header 59
D-3 Configuration Registers (1/2) 60
D-4 Configuration Registers (2/2) 60
D-5 Command Register Commands 60

Introduction

Processing elements in embedded systems have gone through a kind of evolution in the last years. Being split into software-programmable CPUs, fixed-function hardware (ASICs) and reprogrammable hardware, namely the ASICs were more and more replaced by programmable hardware. Reason for that were shorter time to market and the possibility of hardware updating. Having reached large densities and dynamic partial reconfigurability, nowaday's FPGAs can be used much more dynamically. Possible application domains are wearable computing, mobile systems and network processors. Using FPGAs as dynamic processing resources raises the necessity of a *reconfigurable hardware operating system* (RHWOS). Potential benefits of such an operating system could be:

Increased productivity: By means of re-use of reliable code the development cycles can be shortened.

Increased portability: If an OS is supporting different platforms, versions may often be ported just by recompiling/resynthesizing them.

Eased sysem re-partitioning: Tasks running in software on the CPU can be mapped to the FPGA to have an increased performance. Objects can be added or removed without affecting the rest of the system.

Simplified debugging: Monitoring and triggering facilities help to overcome the challenge of debugging dynamically reconfigured hardware and communication hardware.

There is a multiplicity of services a reconfigurable hardware operating system must provide: *Device partitioning*, *placement*, *multitasking*, *task preemption* and *scheduling* to name but a few. More details concerning reconfigurable hardware operating systems can be found in [13], [14], [27] and [28].

This master thesis provides and examines solutions for implementations of hardware tasks on a dynamically reconfigurable FPGA controlled by a reconfigurable hardware operating system.

As an underlying prototype board serves the XF**BOARD** developed by S. Nobs [10]. A short description of the XF**BOARD** is given in chapter 1.

Chapter 2 focusses on use and *organisation of the reconfigurable chip area*.

XFOSGen is a piece of software developped to facilitate the implementation of a partial reconfigurable design. Its use and benefits are described in chapter 3.

In a design that is being dynamically partially reconfigured certain effects might appear, that affect the operation of the FPGA. They are discussed in chapter 4. Namely influences on the *clock distribution* are subject to section 4.1; *transient effects* on signal lines are dealt with in section 4.2.

Similar to clock distribution *communication* on a partial reconfigurable FPGA is not trivial. This issue is discussed in chapter 5.

The *design and implementation* of the partially reconfigurable R-FPGA, done during this thesis, is explained in chapter 6.

Finally chapter 7 takes a look back on the work done during this thesis and also looks forward to further work. How far has the project proceeded? How can the abilities and performance of the R-FPGA and the XF**BOARD** be increased? This chapter will also give some visionary ideas on how an FPGA should be designed to give better support to a system like the XF**BOARD**.

1

RHWOS Platform:
XF-Board / XILINX *Virtex-II*

All practical work in this master thesis targets the XFBOARD. Therefore one needs to be familiar to some extent with this platform. This chapter gathers the properties of the XFBOARD from [10] and [24] that are most relevant for this thesis. The same holds for section 1.2 about the XILINX Virtex-II FPGA: A detailed description of the FPGA would go beyond the scope of this thesis but can be found in [19].

1.1 XF-Board Overview

The two main elements on the XFBOARD are the C-FPGA and the R-FPGA. The C-FPGA (CPU FPGA) is a XILINX Virtex-II XC2V1000, containing a soft core RISC processor, namely the XILINX MicroBlaze (see [18] for further documentation). While such a soft core solution is inferior in performance compared to a dedicated standard processor, its advantage lies in its higher flexibility. The larger XILINX Virtex-II XC2V3000 takes the part of the reconfigurable FPGA (R-FPGA) being managed and dynamically reconfigured during system runtime by the C-FPGA. Thus the R-FPGA contrasts the C-FPGA, which is only configured on system start-up and remains static during system runtime. The two FPGAs are connected by a 40 bit wide general-purpose I/O (GPIO) and an optional general-purpose I/O (OGPIO) bus to provide high communication bandwidth. 16 signals of the GPIO bus are connected to the top left side of the R-FPGA, 24 signals to the top-right side, foreknowing that there they will connect to the static part of the R-FPGA. For the same reason all devices and interfaces connected to the R-FPGA contact on its left and right side respectively. The attached devices are the following:

- 16 bit Audio CoDec AK4563A by AKM [2]

- 24 bit Video DAC HI1178 by Intersil [7]
- 1M x 16 SRAM AS7C34096 by Alliance Semiconductor [1]
- 16M x 16 SDRAM HYB39S256160CT by Infineon Technologies [5]
- 8-LED Bar connected to 8 GPIO lines
- 36 pin I/O Slot (32 signal pins, 5V, 3.3V, 1.5V and ground)
- RS-232 Single Port
- Ethernet PHY with Full-duplex at 10 and 100 Mbps; LXT970A by Intel [6]
- 2 push buttons
- JTAG Header that allows for boundary scan configuration

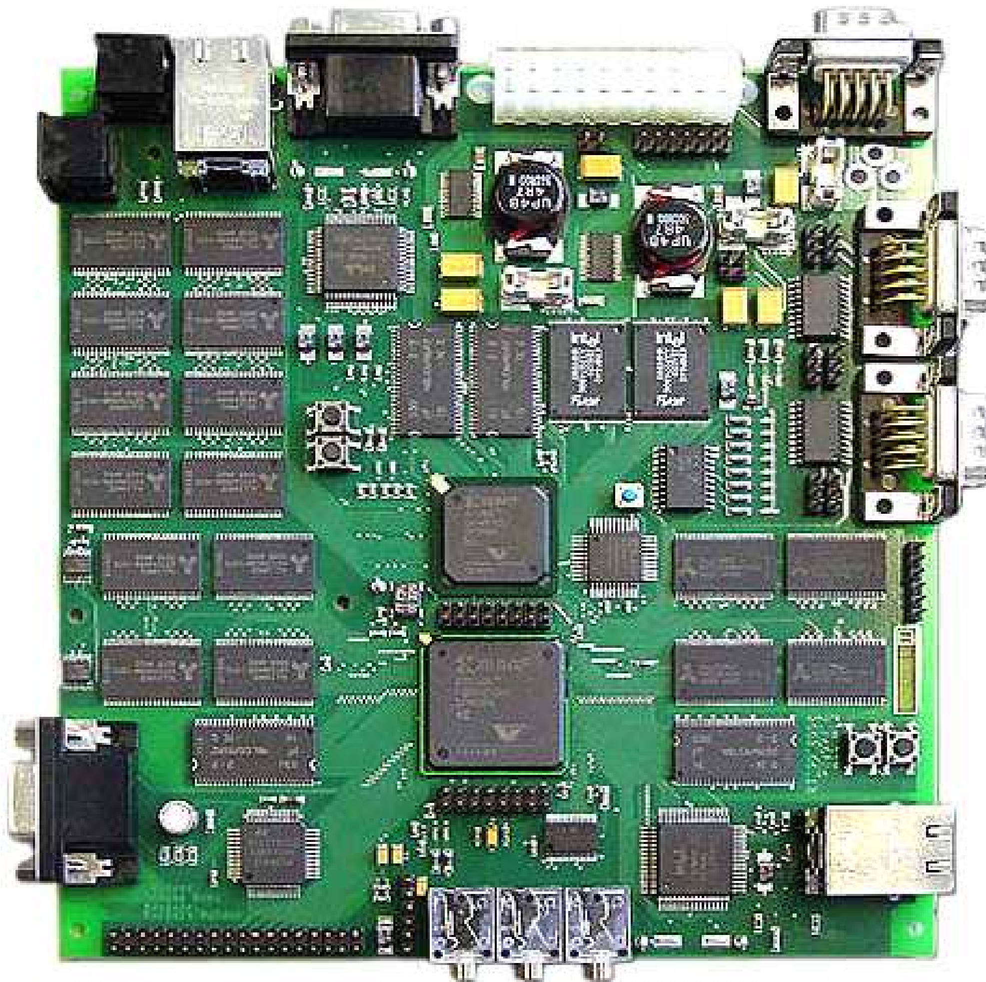


Figure 1-1
The XF-Board RHWOS Platform

1.2 The Virtex-II R-FPGA

The Virtex-II architecture contains input/output blocks (IOBs) and a multiplicity of configurable logic. The configurable logic includes the following elements:

- Configurable Logic Blocks (CLBs)
- Dual-port SRAM in 18 Kbit block SelectRAM resources (BRAM),
- 18-bit $*$ 18-bit Multiplier blocks
- Digital Clock Managers (DCMs)

CLBs are regularly distributed on the FPGA, surrounded by the IOBs. On the XC2V3000 there are 6 columns of 16 BRAM and 16 Multiplier blocks distributed among the CLBs. One BRAM block has a capacity to store 18 kbits. Located above and below these columns are the DCMs. Figure 1-2 gives an overview of a Virtex-II device. Thereby it concerns a smaller device than the XC2V3000. Besides a larger number of CLBs there are six instead of 4 columns of BRAM and Multipliers.

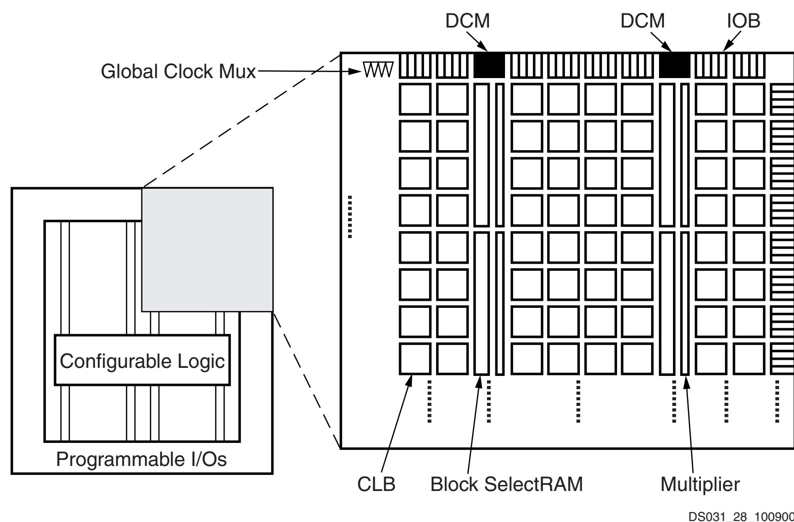


Figure 1-2
Virtex-II Architecture Overview, [19]

A CLB can be looked at in more detail. Each CLB contains 4 so called *slices* and two 3-state buffers. Each slice includes two 4-input function generators that can be configured either as 4-input LUTs, 16 bits distributed SelectRAM memory or 16-bit variable tap shift register element. Also within a slice are two registers, each configurable as either an edge-sensitive flip-flop or a level-sensitive latch. Each CLB can access the general routing matrix via a switch matrix. See figure 1-3 for a visualization of a CLB.

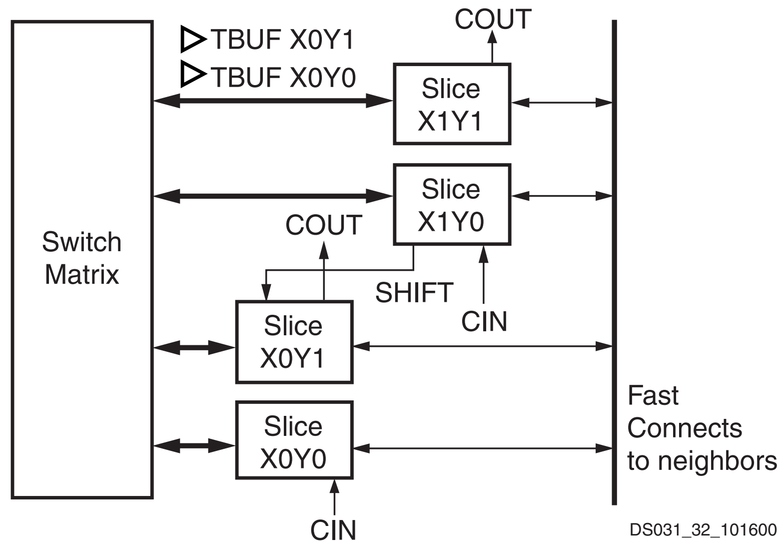


Figure 1-3
A Virtex-II CLB Element, [19]

1.3 Clock Net Structure on the Virtex-II XC2V3000

The Virtex-II devices provides dedicated low-skew clock distribution resources. 16 Global Clock Multiplexer Buffers (BUFGMUX) serve as inputs to these nets. Each BUFGMUX can be driven by a clock pad or an output of a DCM. Eight such clock buffers and pads are located on top and on the bottom of the device. These 16 clock signals form the trunk of the clock tree. There are six branchings to each side of the device. Thereby each quarter of the FPGA can be supplied with at most 8 out of the 16 possible clock signals. The clock signals are distributed further from these branches up and down towards the CLBs. Through the routing matrices each register in a CLB can access any of the eight clock nets in its device quarter. A schematic drawing of the Virtex-II clock net resources is shown in figure 1-4.
Clock net resources are discussed more detailed in section 4.1.

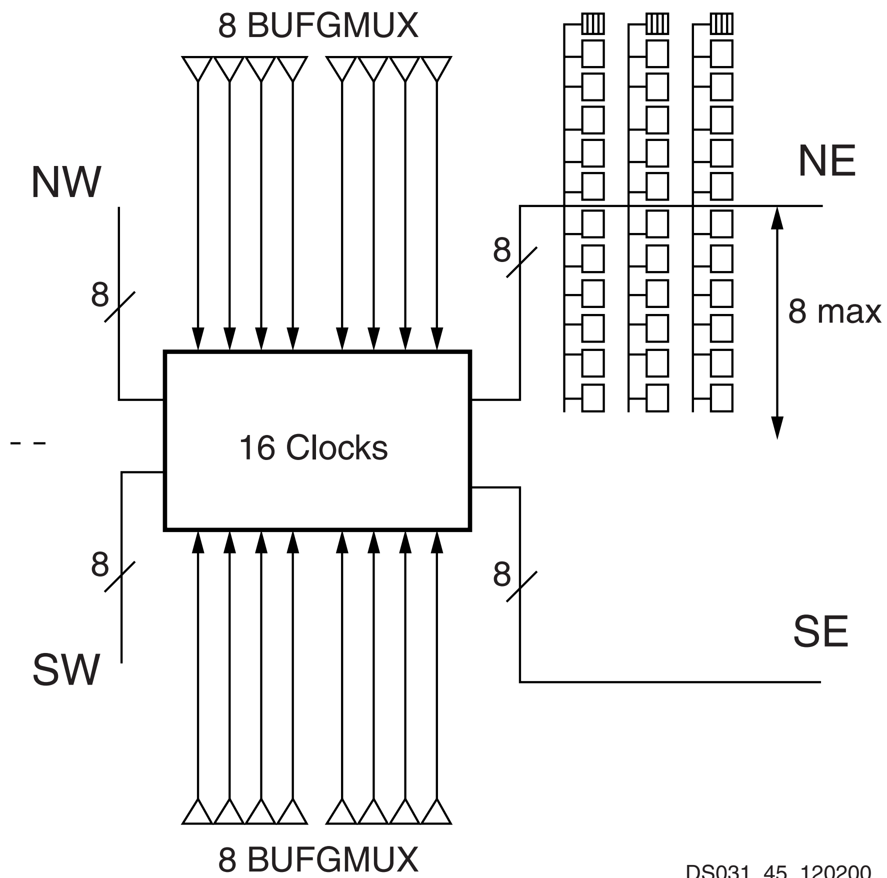


Figure 1-4
The Clock Net Structure in a Virtex-II Device, [19]

2

Use and Organisation of Reconfigurable Chip Area

In dynamically partially reconfigurable computing, FPGA area has become a resource, which needs to be managed similarily to memory. To be able to do so, with effective area utilisation and large flexibility, efforts were taken during this thesis to apply the *variable-sized 1D resource model*, presented in [24].
Section 2.1 of this chapter explains the bipartition of the available area into static and dynamically reconfigurable areas. Afterwards section 2.2 explains the concept of the *variable-sized 1D resource model*. Finally section 2.3 affiliates the *granular variable-sized resource model*.

2.1 Static vs. Dynamically Reconfigurable Area

In the XFBOARD environment, the R-FPGA serves as a resource for different computational and data processing tasks. However there are also operating system services to be implemented on the R-FPGA.

- device drivers (Memory, Audio, Video, Ethernet,...)
- infrastructure for on-chip communication
- infrastructure for communication with the C-FPGA
- memory management
- ...

These OS services need to be present for all or at least several tasks at any time. Thus they are configured statically. This means that this part of the FPGA is configured only once on initial configuration. In this thesis this static section on the device is called *OS frame*. The above list of OS services shows, that the OS frame has to deal with a lot of I/O. Therefore the OS frame needs to be placed at the

border of the FPGA. Actually the OS frame area needs to be split in two parts to access enough IOBs. One part is located on the left side and one on the right side of the FPGA. The remaining area between the *OS frame left* and *OS frame right* is left to be dynamically reconfigurable by any application running on the C-FPGA. The resulting bipartition in static and dynamically reconfigurable area is shown in figure 2-1.

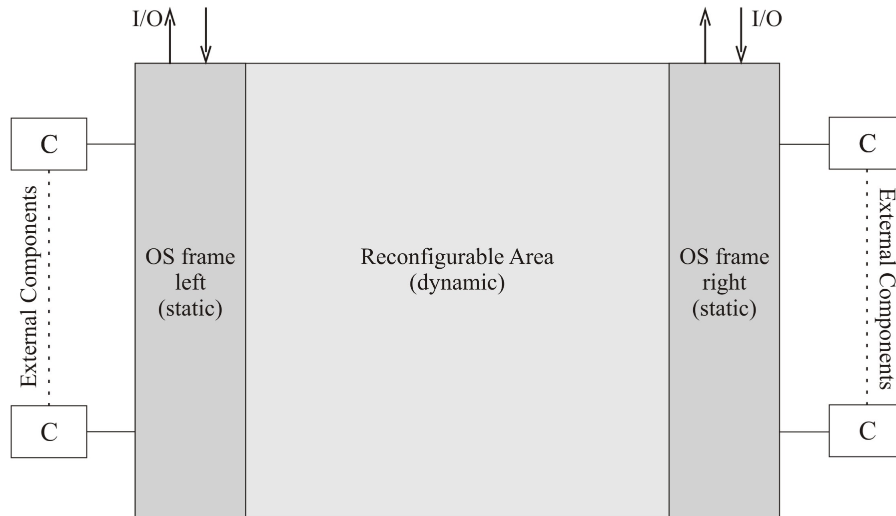


Figure 2-1
Bipartition of FPGA Area into Static and Dynamically Reconfigurable Area

Thereby the resource model is chosen to be one-dimensional due to the fact that Virtex-II devices only support partial reconfiguration in complete columns. In theory other area models are conceivable. Some of them are discussed in [25] and [26].

2.2 Variable-Sized 1D Resource Model

In previous works [3] and [12] the reconfigurable area consisted only of two areas, called *task slots* seperated by a third partition of the OS frame (see figure 2-2). Both tasks are connected to a part of the OS frame via a *standard task interface (STI)*. The three OS frame parts left, middle and right are connected by means of the *Inter-Frame-Communication-Channel-Interface (IFCCI)*. This kind of model allows only two tasks to run concurrently. This concept has two major drawbacks: Firstly, due to the partition in two task slots the maximal possible complexity is limited to the half of the reconfigurable device area. It is not possible to configure the two slots at once with one single large task. Secondly, for tasks with little complexity and little area requirements, there still needs to be configured one complete task slot,

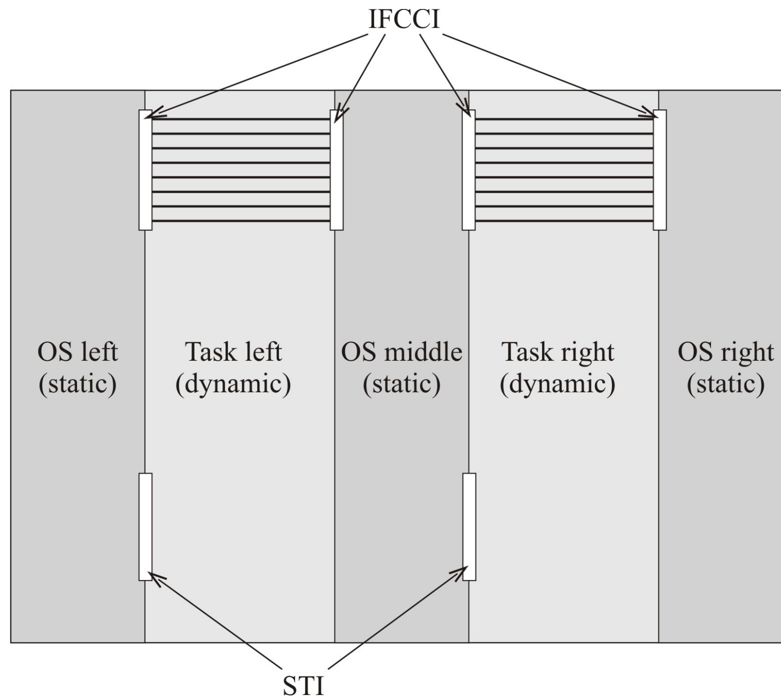


Figure 2-2
Fixed-Size Task Slot Model from [3] and [12]

leading to poor area utilization. This is due to the high slot-internal fragmentaton. On the R-FPGA of the XFBOARD we have one single large reconfigurable area. It would be a great improvement, if the C-FPGA configured only the area just needed by a certain task. So the system can not configure just one out of several given slots with fixed-size, rather it can configure slots in *variable sizes*. Figure 2-3 shows a configuration of an FPGA with *OS frame* and several *variable-sized tasks*.

2.3 Granular Variable-Sized Resource Model

Figure 2-3 reveals two difficulties that are somehow related to each other. One is the fragmentation of the FPGA area that will occur at least after some reconfigurations. The other one is that unused chip area must not be left unconfigured, since this would disconnect communication lines between tasks and the OS frame. Let's postpone the discussion of fragmentation for a while and concentrate on communication lines. It is evident that FPGA regions, that do neither contain a task nor are a part of the OS frame, need to contain the necessary routing ressources, to enable the task to communicate with the OS frame (see fig. 2-4).

Such a task, that contains no logic but only routing, is henceforth called *dummy task*. At this point we need to anticipate the introduction of the *bus macro*. A bus macro needs to be instantiated, for signals that cross the border between two modules (i.e. two tasks or a task and a part of the OS frame). Why we need bus macros and how they work is not of importance at this point. What we need

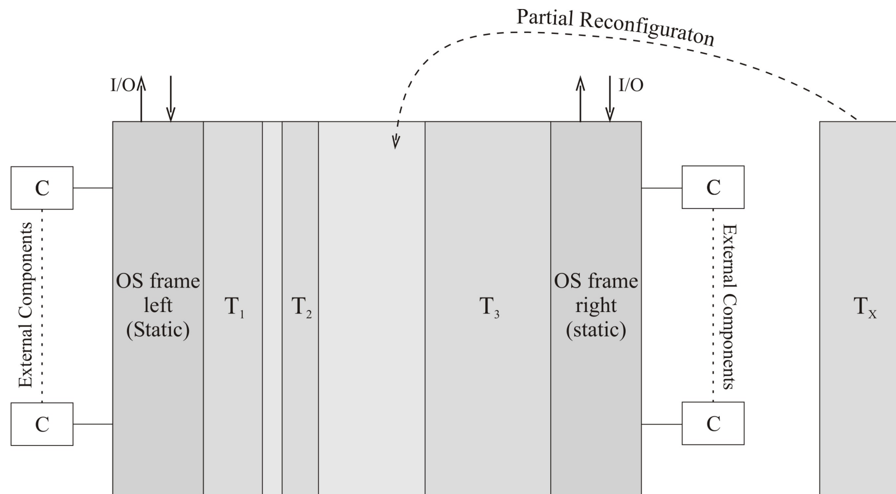


Figure 2-3
The Variable-Sized Task Model Proposed in [24], Configured with a Set of Tasks (T_1, T_2, T_3,T_x) with Different Sizes

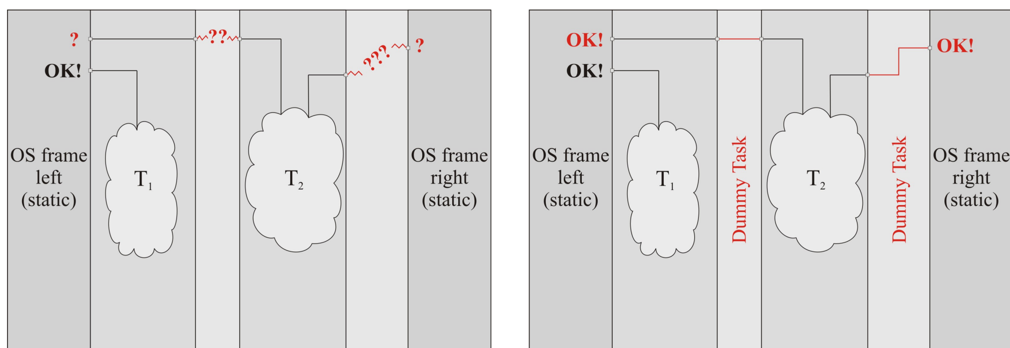


Figure 2-4
Not Explicitly Configured FPGA Area Must Contain the Necessary Routing to Connect Other Tasks to the OS Frame.

to know is that each bus macro instantiates two pairs of tri-state buffers in both affected modules, all positioned in sidewise adjacent CLBs. See figure 2-5 for a schematic of such a bus macro.

As we need the signals to traverse the dummy task, there are two bus macros to be instantiated. One where a signal enters the dummy task and one to guide the signal out of the dummy task. To get the most simple routing possible, these two bus macros are placed on a horizontal line. As there are only two tri-state buffers available per CLB, we need to instantiate the tri-state buffers of 4 CLBs in a line. Thus the *minimal width* (w_{min}) for a dummy task becomes 4 CLBs. This minimal width applies as well to non-dummy tasks[1]. Proper communication is assured if every unused space between tasks and OS frames is configured with a dummy task.

[1]This value is purely theoretical. Especially non-dummy tasks need to be wider to have sufficient routing resources inside.

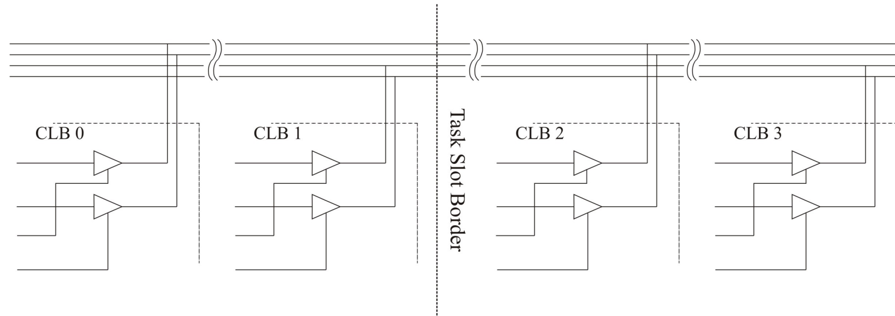


Figure 2-5
Schematic of how Tri-State Buffers are Instantiated in a Bus Macro

Coming back to the fragmentation problem we now can state that no partial reconfiguration process may leave an unused fragment narrower than w_{min}. An example how such a forbidden case can emerge from a partial reconfiguration is shown in figure 2-6.

The granular variable-sized resource model helps to get rid of such cases. This

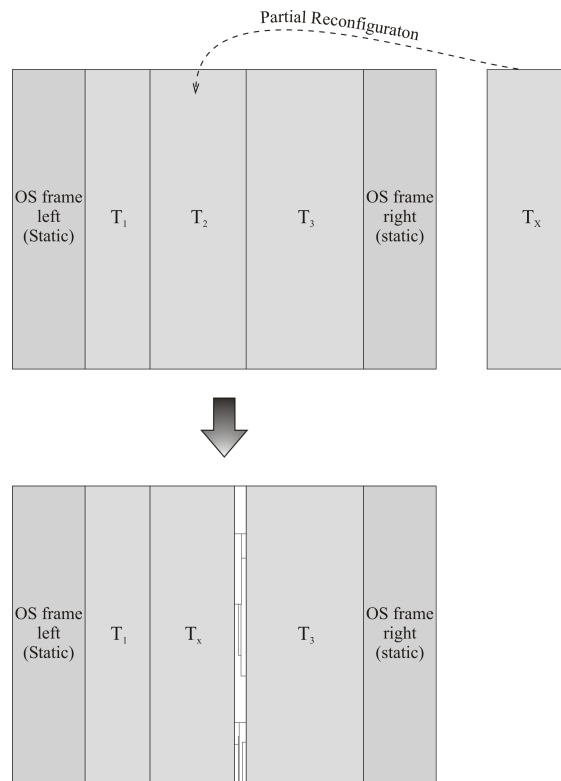


Figure 2-6
An Unconfigured Area with Width $w < w_{min}$, Emerging from Partial Reconfiguration

model is based on the following rules:

1. The whole dynamically reconfigurable area needs to be an integer multiple of the minimal dummy task width w_{min} wide.

2. Tasks, that are to be dynamically reconfigured, need to have widths of integer multiples of w_{min}. (It goes without saying, that a task can not be larger than the whole reconfigurable area)
3. A task must always be configured to areas in such a way, that remaining gaps to the left and right OS frames have widths of an integer multiple of the minimal task width.

This model is much more easier to understand by means of an example: Let's suppose the size of the reconfigurable FPGA area is $5 * w_{min}$ wide. So the FPGA can be configured with up to 5 tasks with width w_{min} (figure 2-7a). Each such task has 5 different location, where it can be configured to. A tasks of with $2*w_{min}$ could be placed in 4 different locations. As shown for tasks T_A through T_D in figures 2-7b and c. And so on, until finally one single task with width $5 * w_{min}$ could be configured to fill the whole reconfigurable area alone (task T_Z in figure 2-7d). It is understood, that tasks configured on the FPGA at the same time can have different sizes (figures 2-7b and c).
Most implementations done during this thesis use OS frame sizes of 8 CLBs. Also minimal-sized dummy tasks and user tasks are 8 CLBs wide. Since the Virtex-II XC2V3000 is exactly 56 CLBs wide this results in a reconfigurable area, which is 40 CLBs wide and therefore provides space for five concurrently configured minimal-sized tasks.

The above example reveals, that the granular variable-sized resource model is somehow constrained in the freedom to choose size and position of tasks, compared to the pure variable-sized model. But there is also an advantage in the need for granularity. At the current state of the project partial bitstreams are not relocatable. This means, that it is not yet evaluated if and how an existing partial bitstream could be edited to configure a certain other part of the FPGA than the one constrained during implementation. From this it follows that for each task to be configured to the R-FPGA, as many partial bitstreams have to be stored, as there are possible target locations. So by applying rule number 3 from the above list, memory needed to store the partial bitstreams is reduced significantly. Consider a design where n CLB columns form the dynamically reconfigurable area. There are now $\frac{n}{w_{min}}$ possible target locations for a minimal-sized task, instead of $2 * (n - w_{min}) + 1$ when the configured task could be placed originating in any slice-column of the reconfigurable area.

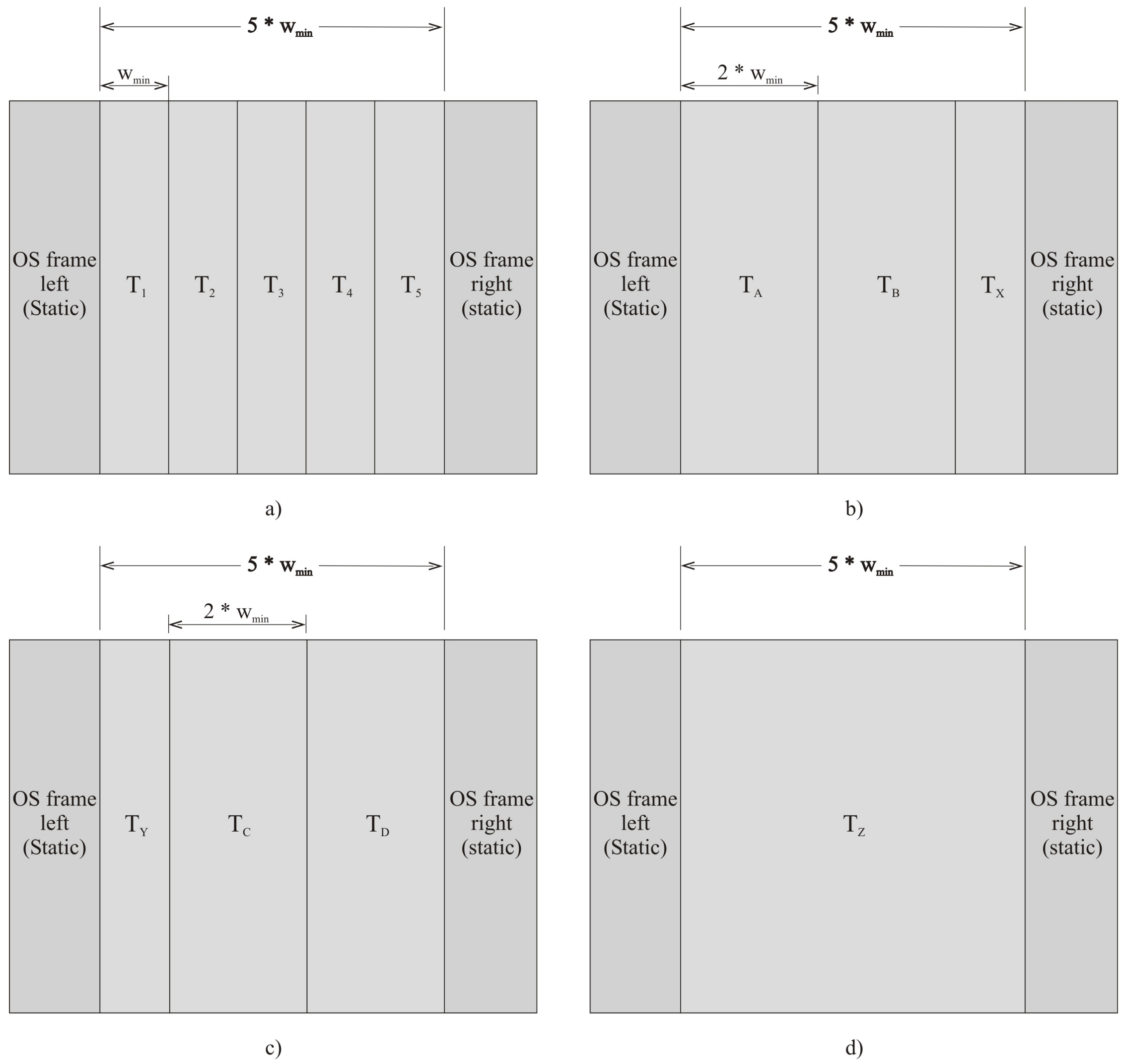


Figure 2-7
A Set of Different Configurations in the Granular Variable-Sized Resource Model

3

The RHWOS Generation: XFOSGen

Creating a partially reconfigurable design such as the one for XFBOARD is a pretty extensive work.
On one side in a partial reconfigurable design with up to five user tasks a countless number of bus macros and signal lines need to be connected. Let's make an example as a substantiation: Assuming the OS frames and the five user tasks are connected with a shared bus of 32 bits width, we would need eight bus macros that guide four signals from module to module, on every module border. There are six such borders. Makes 48 bus macros. These bus macros need to be connected with four input and four output signals plus eight signals determining the direction of the respective signal line. So in addition to the 48 instantiations $48 * 16 = 768$ signals need to be connected. This would be a very boring and also error-prone work to do by hand.
On the other side does XILINX ISE Project Navigator not support Modular Design, which is necessarily used for partially reconfigurable designs. Therefore Modular Design needs to be done with the XILINX ISE-tools ran in command-line mode. However, when XFOSGen has fulfilled its work this is a favorable fact, as it creates a .bat file that controls the complete implementation flow. So the XFOSGen user does not need to worry about Modular Design flow anymore.
Still the next section 3.1 gives an overview on the Modular Design flow. The way XFOSGen is used, is described in section 3.2. Section 3.3 describes the files produced by XFOSGen as well as their content, functionality and relations to the modular design flow.

3.1 Modular Design and Partial Reconfiguration

The term "Modular Design" is introduced in chapter 4 of the XILINX ISE Reference [17]. There, the primary motivation for building modular designs is to allow a team of engineers to independently work on different pieces of one large design. Also individual modules might be modified without affecting other modules that

need no changes. When all modules are implemented, they can be merged into one final design. The main motivation for using the modular design flow in this thesis is that any module can be defined as *partially reconfigurable*. Thereby, in addition to the regular modular design flow, each module gets constraints about where its logic is to be placed on the FPGA. Plus, the place and route tool is directed not to do any routing of the logic outside this module area. So a module can not only be changed during implementation without affecting other modules, it now can also be *partially reconfigured* during runtime.
The detailed description of all aspects of "Modular Design" and "Partial Reconfiguration" is covered by the Xylinx ISE Reference [17] and the XILINX application note 290 [21]. So this section shall not copy these two documents but rather serve as a digest of the factors, that are most important in the context of this thesis. Thereby the division into the three phases *Initial Budgeting*, *Active Module Implementation* and *Final Assembly* shall remain. But before starting with the design entry, it is a good idea to set up a directory structure for the modular design to keep track of the different files generated during design.

3.1.1 Modular Design Directory Structure

XILINX application note 290 [21] recommends to have a well-organised directory structure to keep track of the multitude of files being created during the different phases of modular design. The chosen directory structure, with some adaption made to the one proposed by XILINX, is shown in figure 3-1.

`hdl` contains the source code for the design, written in any HDL.

`synth` contains the files necessary for synthesis. In this thesis XILINX Synthesis Technology (XST) is used for synthesis.

`top\initial` is the directory, where the *Initial Budgeting* is done.

`modules` contains a subdirectory for each module to be implemented in a different area on the device.

`pims` contains intermediate files from the *Active Module Implementation*, which are used later during *Final Assembly*.

`top\assemble` is the directory, where during the *Final Assembly* phase the full configuration bitstream is produced.

`bitstreams` gathers all full and partial bitstreams.

For a better understanding of, what is done during implementation of a modular design read the following subsections 3.1.2 to 3.1.5.

3.1.2 Modular Design Entry and Synthesis

First of all a complete design entry and synthesis for the top-level design needs to be done. Thereby all modules are instantiated as *black boxes* in the top-level

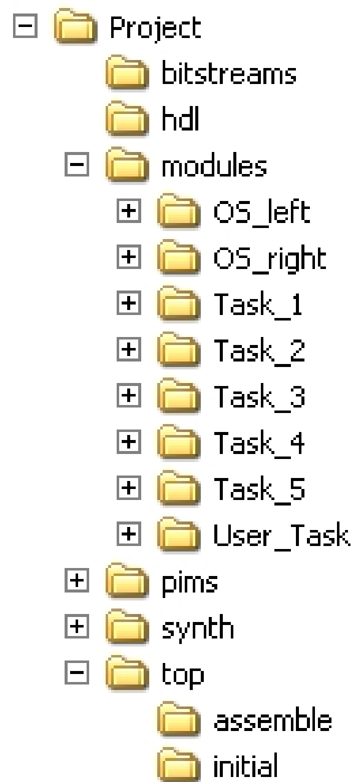


Figure 3-1
Modular Design Directory Structure

design. Their interconnections are realized by means of *Bus Macros* (see subsecton 3.1.2.1). Also included in the top-level design are I/O ports, global logic and signals connecting these elements. Global logic (i.e. logic that is not part of a lower level module) shall be kept to a minimum, because area used for top-level logic is no more available for the modules.

3.1.2.1 Communication via Bus Macros

Communication between two modules in a partially reconfigurable design is not a trivial thing. Recalling the idea of independent design of different modules, made in the introduction of section 3.1. It needs to be assured, that both modules use the same fixed routing resource for communication. This means, the routing used for these intermodule signals must not change when a module is reconfigured. A *Bus Macro* satisfies this demand. It is a hard macro used to specify the exact routing resources between two configuration areas. The physical implementation of a *Bus Macro* is shown in figure 3-2.

As the figure shows, one *Bus Macro* guides exactly four signal lines. These lines can be configured to either guide the signal from the left to the right side or vice versa. To configure the *Bus Macro* to guide the signal from left to right, the `"LT"` inputs

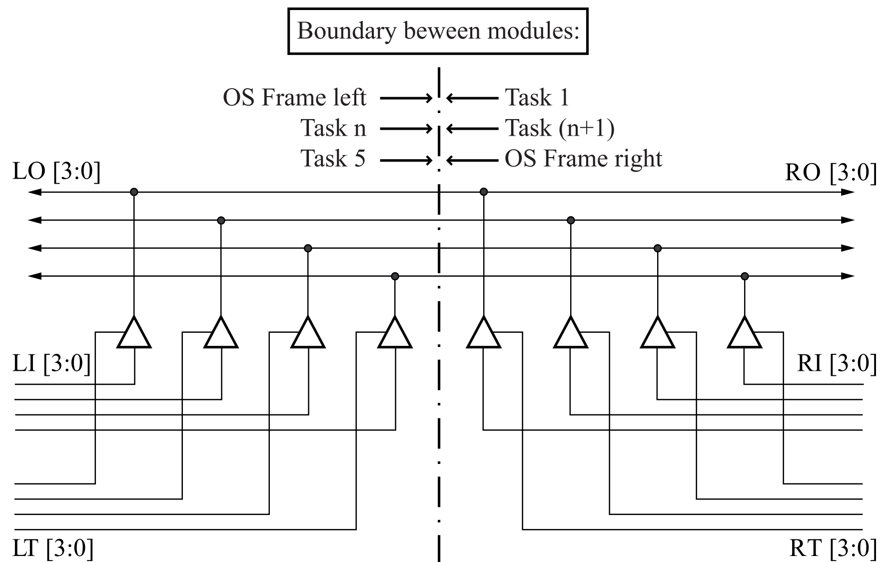


Figure 3-2
Physical Implementation of a Bus Macro (adapted from [21])

needs to be connected to a constant `'0'` and the `"RT"` inputs to `'1'`. For guidance from right to left, the constants are exchanged.

3.1.3 Initial Budgeting Phase

After every module is designed in VHDL or Verilog, it is synthesized seperately in either XILINX Synthesis Technology (XST) or any other XILINX-supported synthesis tool. Then the actual "Modular Design" flow can begin with *Initial Budgeting*.
In this phase each module must be sized and positioned on the target FPGA. Global logic must be positioned as well as all bus macros. To facilitate the setting of these constraints **ngdbuild** is run in the *initial* mode with the top-level design file.

```
ngdbuild -p xc2v3000-4fg676 -modular initial top.ngc
```

The option *-p xc2v3000-4fg676* sets the target device to the XILINX Virtex-II used as R-FPGA on the XFBOARD. *-modular initial* tells **ngdbuild** that the implementation is currently in initial mode and therefore the processed input file is the top-level design file. *top.ngc* denotes the input file for **ngdbuild**. **Ngdbuild** generates a .ngd file with all instantiated modules represented as unexpanded blocks. This file can be used with the *Floorplanner* tool to easily assign the location constraints for each module and the bus macros. This is a very important part in a design for partial reconfiguration. As now the `.ucf` file is provided with the information on where module boundaries will occur. Setting such area group constraints is done either by using *Floorplanner* or by hand, as it is shown here:

```
INST ”u_task_1” AREA_GROUP=”AG_u_task_1”;
AREA_GROUP "AG_u_task_1" RANGE=SLICE_X16Y127:SLICE_X31Y0;
AREA_GROUP "AG_u_task_1" RANGE=TBUF_X16Y127:TBUF_X30Y0;
AREA_GROUP "AG_u_task_1" RANGE=RAMB16_X1Y15:RAMB16_X1Y0;
AREA_GROUP "AG_u_task_1" RANGE=MULT18X18_X1Y15:MULT18X18_X1Y0;
```

This defines the modules as *area groups* in terms of slices, tri-state buffers, BRAM and Multiplierblocks, it is allowed to use.
To assure, that no routing of module logic outside the module area is done, the following command is placed in the `.ucf`:

```
AREA_GROUP "AG_u_task_1" MODE=RECONFIG;
```

Finally the bus macros need to be placed. The command used looks like this:

```
INST "bm_1_we_1" LOC="TBUF_X12Y0";
```

The assigned location is only one TBUF element. The information about other instances is contained in the macro-file. There the relative position to the located (`LOC`) instance is described.
To assign the constraints for the pin locations a web application was created in context with the term thesis [10]. It can be found on the CD attached to this thesis or on the internet at:

http://www.tik.ee.ethz.ch/~xfboard/R-FPGA.htm

However, Floorplanner and the abovementioned web application will only be used for manual changes in the design since XFOSGen will already take care of both. Please take care that **ngdbuild** needs to be rerun, whenever changes to the .ucf constraint file have been made. Otherwise these changes are not applied to the complete design.

3.1.4 Active Module Implementation Phase

This phase has to be run seperately for every module. The general context of the module is provided by the top.ngc file generated in the *Initial Budgeting* phase.

```
ngdbuild -p xc2v3000-4fg676 -uc task_1.ucf
  -modular module -active task_1 ..\..\top\initial\top.ngc
```

Again the -p option denotes the target device. Optionally module-level timing constraints can be added to the `.ucf` file specified in the -uc option. Differently from the *Initial Budgeting* phase, this time the *-modular module* option tells **ngdbuild**, that a *module* is being implemented. The *-active* option tells, which module from the top-level design is being implemented. Attention has to be paid, that the argument to the -active option is the component name and NOT the instance name of the module to be implemented.
Next the resulting `.ngd` file can be mapped, placed and routed:

```
map -u -detail -pr b top.ngd -o top_map.ncd top.pcf
```

The -u option tells the **map** tool not to remove unused logic. This option proved to be necessary in some cases to prevent **map** from optimizing away complete modules. *-detail* produces a more detailed map report. The *-pr b* option allows **map** to place input and output registers in I/O cells. This saves FPGA area and provides better I/O timing. *top.ngd* specifies the input file which resulted from **ngdbuild**. *-o top_map.ncd* denotes the output file. *top.pcf* contains the information from the .ucf file in a different format.

```
par -n 1 -s 1 -ol high -w top_map.ncd mppr.dir top.pcf
```

Options *-n 1* and *-s 1* tell **par** to run one iteration and store its result. Setting these parameters to higher values might be an option if timing constraints are marginally missed. *-ol high* sets the overall effort level for placing and routing. -w allows **par** to overwrite existing output files. *top_map.ncd* is the input file for **par**. *mppr.dir* is the directory where the results of the **par** iterations are stored and again top.pcf denotes the constraints file.

When a module is successfully mapped, placed and routed it is published for inclusion in the final design. The files published are called PIMs, standing for **P**hysically **I**mplemented **M**odule. The necessary command is the following:

```
pimcreate ..\..\pims -ncd top.ncd
```

Pimcreate copies the files ”<design_name>.ngo”, ”<design_name>.ncd” and ”<design_name>.ngm” to ”<design_name>.ngo”, ”<module_name>.ncd” and ”<module_name>.ngm” into the `pims` directory.
To generate a partial bitstream the following command is applied:

```
bitgen -d -f bitgen_v2_std.ut top.ncd task_1.bit
```

The option *-d* disables the DRC check, although this seems something critical to do. Experiences show, that too often errors about signals crossing the module boundaries prevent **bitgen** from successfully creating a bitstream. Bitstreams created later without DRC never fail to operate. For further options *-f bitgen_v2_std.ut* assigns an options file, which contains a large list of options shown in C. Most options are set to their defaults. The most important options for partial reconfigurable designs are the following:

- `g ActiveReconfig:Yes` prevents the assertions of GHIGH and *global set/reset* (GSR) during configuration. This is required for the active partial reconfiguration enhancement features, since GHIGH would set all CLB outputs of the device to ’1’ and GSR would reset all flip-flops and latches for at least one cycle.

- `g Persist:Yes` is needed for readback and partial reconfiguration using the SelectMAP configuration pins. If `Persist` is set to `"Yes"`, the pins used for SelectMAP mode are prohibited for use as user I/O.

- `g ActivateGCLK:Yes` allows any partial bitstream for a reconfigurable area to have its registered elements wired to the correct clock domain.

These and all other options are described in [17], which is left to the reader to consult, since their impact is not of large importance in context with this thesis.

3.1.5 Final Assembly Phase

During this phase, the modules are assembled into one design by running **ngdbuild** in *assemble* mode:

```
ngdbuild -p xc2v3000-4fg676 -modular assemble -pimpath
   ..\..\pims top.ngc
```

Thereby the top-level `.ngo` file , constraints from the top-level `.ucf` file and all PIMs are read to create a design, where all modules are fully expanded. Then for **map**, place and route (**par**) the placement and routing information from each PIM is used to preserve the results from the *Active Module Implementation* Phase. Finally **bitgen** is run with the DRC enabled to create a full bitstream.

```
bitgen -f bitgen_v2_assemble.ut top.ncd
```

3.2 How to Use XFOSGen

Figure 3-3 shows the Graphical User Interface of XFOSGen. In the uper left section several parameters can be set:

- **Device Type:** Currently only Virtex-II 3000 is supported.
- **OS Frame width:** The left (L) and right (R) part of the os frame can be sized seperately. The widths are measured in number of slices. Remember one CLB is 2 slices wide. The sizes of the os frames must be chosen to leave a reconfigurable area width open, that is an integer multiple of 16 slices wide.
- **Number of IFCCs:** The number of signal lines can be set seperately for signals running from west to east (WE) and from east to west (EW). Since one bus macro contains four signal lines, numbers entered must be an integer multiple of four.

In the field 'Design Step' it can be chosen whether only a blank OS frame with dummy tasks inbetween is created, or if additionally a user task is produced. If 'User Task' is selected the following options can be set for the user task:

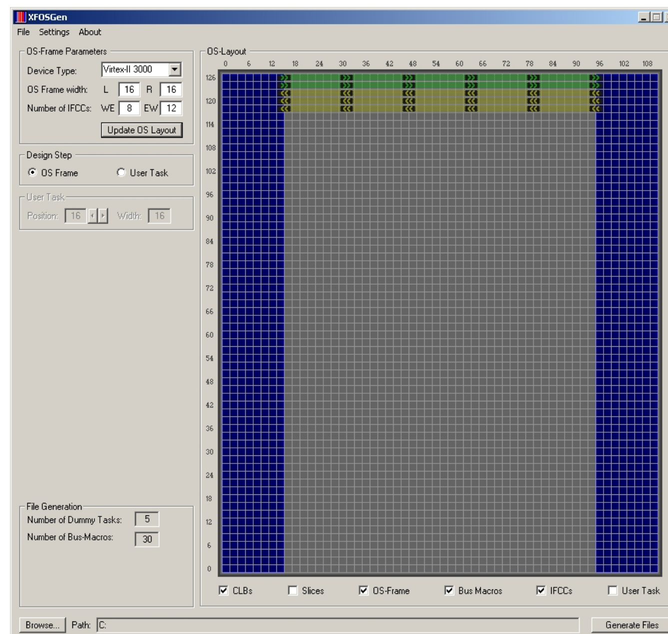


Figure 3-3
The XFOSGen Graphical User Interface

- **Position:** The position where the user task is to be configured can be adjusted in steps of 16 slices.
- **Width:** The width of the user task can be an integer multiple of 16 slices, but it must not exceed the width of the reconfigurable area.

The button 'Update OS Layout' applies the parameters to the visualization in the right part of the GUI. Several check boxes control whether CLBs, slices, OS frame, bus macros, IFCCs and user task are displayed.

The lower left section shows how many dummy tasks are created and how many bus macros are instantiated, with the current settings.

On the bottom line the target directory can be selected with the 'Browse...' button. Finally the generation of the design files is initiated.
Figure 3-4 shows a view of XFOSGen with different settings.

3.3 XFOSGen Output Files

XFOSGen generates a complete directory structure to match the paradigms of modular design flow as already shown in figure 3-1.
There are several sorts of files created inside the design directory structure:

- Batch files, that control the modular design flow.

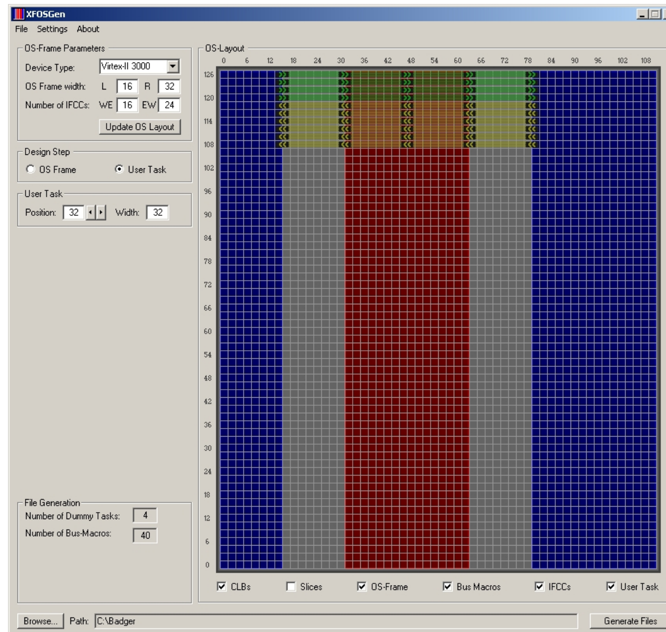


Figure 3-4
XFOSGen configured to Create a Design with a Larger Right OS Frame and a User Task with Width $2 * w_{min}$

- VHDL files, containing application independent code, such as: operating system components, communication infrastructure, instantiations of modules and bus macros, ...
- Constraint files (`.ucf`), that contain: area constraints, bus macro placement constraints and pad location constraints.
- Options files for synthesis and bitstream generation.

Before the implementation is started, the user is free to add his code to the various .vhd files in the hdl directory. operating system components shall be added to "`os_left.vhd`" and "`os_right.vhd`" respectively. If a user task shall be produced its functionality shall be added to "`user_task.vhd`". The dummy tasks '"`task_i.vhd`" need no changes. The logic they contain is necessary to prevent the ISE tools from optimizing them away.
Also the correct *Bus Macro* file needs to be copied to the directories "`Project\top\initial`" and "`Project\top\assemble`".
The implementation is started by double-clicking the "`run_design.bat`" batch file. Thereupon the synthesis is started, using XILINX Synthesis Technology (XST). If the user has added files to the project, they need to be listed in "`Project\synth\project_files\<module>.prj`" before synthesis is started. Where `<module>` stands for the module name wherein the additional component is instantiated. Since by default all area is distributed to os frames and tasks, it is not allowed to add any other components to the "`top.vhd`" file. The listing in the `.prj` files must be in reverse-hierarchical order as shown in the example for an "`os_right.prj` file in appendix A.
Synthesis options are set in the `.xst` files, which are located in

"Project\synth\xst_settings\<module>.xst". An example .xst file can be found in appendix B. Changes to this file shall only be done with precaution. Namely important settings to be left unchanged are:

- bus_delimiter () keeps the netlist compatible to following ISE tools, while other delimiters as [], {} or <> would cause trouble.
- iobuf NO tells the synthesizer not to instantiate I/O pads for port signals. It must be set to "NO" for all modules (including os frames). For the top-level design it must be set to "YES".

The synthesis output files are copied by the "run_design.bat" file to the appropriate directory in the "Project\modules" directory, where they are further processed during *Active Module Implementation*.
After synthesis has taken place the *Initial Budgeting* phase takes place in the "Project\top\initial" directory.
Thereafter "run_design.bat" moves to the "Project\modules" directory to start the subordinate batch files, which control the *Active Module Implementation* of the OS frames, dummy tasks and user task. These batch files run **ngdbuild**, **map**, **par**, **bitgen** and **pimcreate** as described in subsection 3.1.4. The created partial bitstreams are copied to the "Project\bitstreams" directory. An example for the **bitgen** settings file can be found in appendix C.
Then "run_design.bat" moves on to the directory "Project\top\assemble". There the last batch file called "assemble.bat" takes care of the *Final Assembly* phase as described in subsection 3.1.5. "assemble.bat" creates one full bitstream containing the OS frame and the reconfigurable area filled with dummy tasks. Depending on whether a user task was implemented, a second full bitstream is created. It contains the OS frame, the user task and — if necessary — dummy tasks to fill the reconfigurable FPGA area. Again the produced bitstreams are copied to the "Project\bitstreams" directory.

4

Reconfiguration Effects

When an FPGA is used in a static manner, configuration is done on system start-up. As such a configuration is always a full configuration and no part of the FPGA is processing data at configuration time, there are no Reconfiguration effects that would affect functionality of the device. This is different in a design where the FPGA is partially reconfigured during runtime. There two possible effects might affect the proper operation of other, running tasks. These two effects are leaks in the *clock distribution* (discussed in section 4.1) and *transient effects* (discussed in section 4.2).

4.1 Clock Distribution

When designing dynamically partial reconfigurable designs, there are some situations concerning the clock net to be expected, which would lead to undesirable behaviour of the FPGA. Subsection 4.1.1 describes from what these situations could evolve. Thereafter subsection 4.1.2 shows how one can avoid the named complications with the clock nets.

4.1.1 What is the Problem with the Clock Distribution?

Firstly recall that the groundbreaking fact about the dynamic partial reconfiguration: Parts on the FPGA which are not to be overwritten by an applied partial bitstream are able to do their work during and after the partial reconfiguration without an interrupt. To be able to do so without any errors, two preconditions must be fulfilled:

1. No signal lines of the running tasks may cross the border to the area where a task is being reconfigured.

2. The clock signal driving any sequential logic must not be disrupt. As shown in figure 4-1a and b

Precondition 1 is met by using the XILINX design flow for partial reconfiguration with the area constraints, described earlier in section 3.1 of this report. Precondition number two needs to be taken care of.
Furthermore a newly configured task needs to be connected to the proper clock nets to be able to start its operation. Not as depicted in figure 4-1b. One needs to consider this situation as well when creating partially reconfigurable designs.
For the following paragraphs let's call the pitfall in figure 4-1a *cut off clock*, where task T_X disconnects the right OS frame from the clock net. And let's call the one in figure 4-1b *missing clock*, where task T_Y lacks an existing clock net.

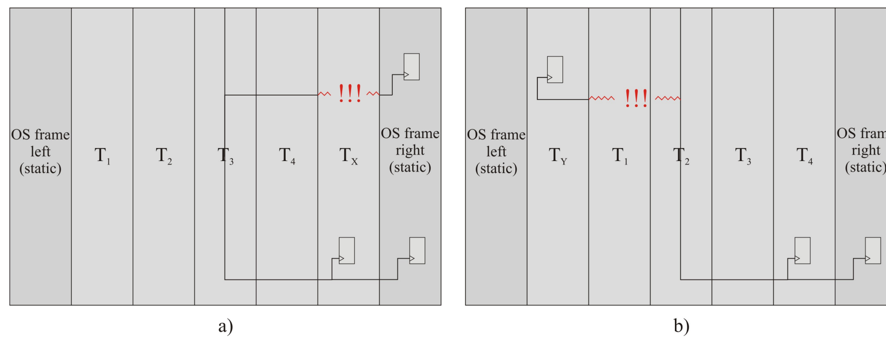


Figure 4-1
Two Possibilities for Clock Net Problems Caused by Partial Reconfiguration

4.1.2 First Approach to the Clock Net Problem

Let's take heavily simplified view of the problems described in the previous section: The problems are inactve wires or wire parts in a partial configuration file. They appear because the bitstreams were produced without a global context, which consists of adjacent areas containing logic and the global clock net. What we need to do, is somehow manually insert this contextual information. So we need to manually activate clock net sections, which are not used by the reconfigured area itself but by another area already configured or one to be configured in the future. Figure 4-2 shows, how one completely activated clock net looks like.

Obviously it would be an enormously extensive and tedious work to check for every part of the clock net, whether it is needed to be active at any time or not. The secure workaround is to activate the complete clock net in the initial full bitstream and also in every partial bitstream.
The manufacturer of the Virtex-II devices is not willing to provide information of the internal structure of the configuration files to its devices. So a direct manipulation of configuration bitstreams is out of the question. Instead, XILINX provides a Java pakage called JBits, which supports generation of new and manipulation

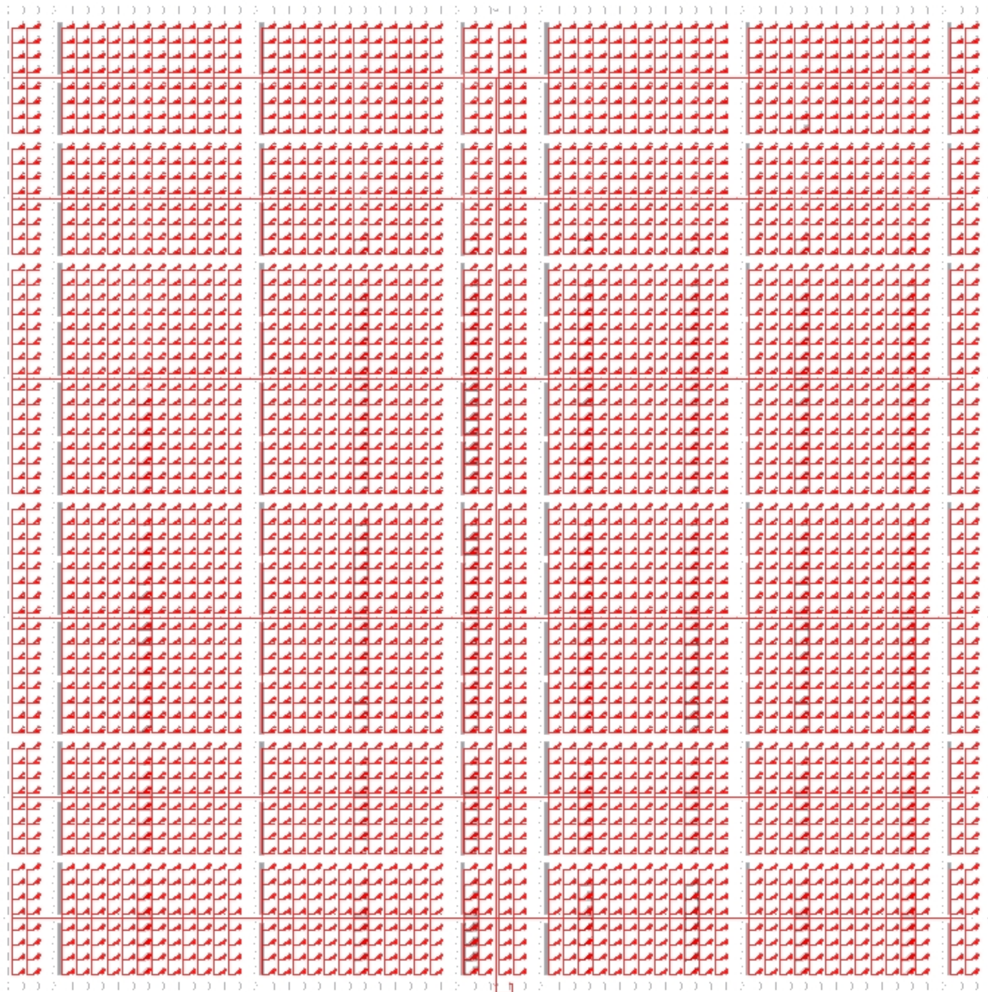


Figure 4-2
Screenshot from XILINX *FPGA Editor of a Fully Activated Clock Net*

of existing bitstreams at lowest level. Therefore, a software to activate a clock net, must be written in Java using XILINX JBits. Unfortunately there exists only little documentation on JBits. The tutorial provided gives a quick introduction on how to do some basic operations as manipulating LUTs or connecting wires. However, to access a specific resource, it is hard to find the correct class and method among the huge number of possibilities. Unfortunately the API provided with JBits does not give much information on how the packages are structured. Also the naming of methods and fields is barely self-explanatory.

The approach to the desired software was pretty much based on trial and error: To be able to see whether a JBits command affects a bitstream in the desired manner, a bitstream was taken, in which all flip-flops on the FPGA and therefore one complete clock net were active. Then using "XILINX FPGA Editor" the clock net was deleted. After producing a bitstream from this edited design, the clock net was step by step reintroduced using JBits. Comparing the manipulated and the

original bitstream with the full clock net helped to check if the used JBits command really did the desired routing. To be able to observe changes made to the bitstream and differ control-bits from configuration-bits, a software called ”BitStreamParser” was written to translate any bitstream from bit format to the XILINX configuration instructions described in [16] and [20]. See appendix D for more details on ”BitStreamParser”. Finally ”ClockActivator” included all the commands necessary to activate a full clock net.

4.1.2.1 Clock Activator

The *ClockActivator* was oritinally written to support designers for the XFBOARD platform. It can only activate 5 clock nets because there are only five pads connected to clock sources. One of them bearing the 50 MHz clock signal from the on-board cristal oscillator. The others being programmable clock signals from the C-FPGA. See table 4-1 for a list of all these clock signals, PADs they are attached to and clock net numbers they are supplying.

Clock	PAD	Clock Net Nr.
Clock 0 from C-FPGA	D13	5
Clock 1 from C-FPGA	G14	1
Clock 2 from C-FPGA	AB13	6
Clock 3 from C-FPGA	AB14	2
Clock 4 from 50 MHz Oscillator	F13	7

Table 4-1: Clock Inputs to the R-FPGA, Clock Pads and Clock Nets

Let’s concentrate on the commands found to be useful to fulfill the needs of clock net activation.

First it is important to know, that for the use with JBits a Virtex-II device is organized in a multitude of so called ”tiles”. The most common ones are called ”CENTER”. They contain the CLB information. To find out which ones contain clock net information, the following method is valuable:

```
MyDevice.getTileType(TileYCoord, TileXCoord)
```

With the knowledge of tile names and coordinates as well as the JBits API [15] it was possible to find out, which tiles of the device contains resources that control the routing of the clock net.
Depending on which clock net needs to be configured the first routing step is either in the middle of the top tile row or in the middle of the bottom tile row. As it is the global clock buffer which needs to be enabled. Clocks 0, 1 and 4 enter the device through the buffer on top, clocks 2 and 3 through buffers on the bottom of the device. The respective tiles are called `"CLKT"` or `"CLKB"`.
The command

```
MyJBits.setTileBits(TileYCoord, TileXCoord,
com.xilinx.JBits.Virtex2.Bits.Logic.Clkt.DISABLE_VALUE.CONFIG[7],
```

```
com.xilinx.JBits.Virtex2.Bits.Logic.Clkt.DISABLE_VALUE.LOW);
```

enables the clock buffer number 7 from the top clock input. Further the command

```
MyJBits.setTileBits(TileYCoord, TileXCoord,
com.xilinx.JBits.Virtex2.Bits.Logic.Clkt.SINV.CONFIG[5],
com.xilinx.JBits.Virtex2.Bits.Logic.Clkt.SINV.S_B
```

selects the input to the buffer by either inverting (`S_B`) or not inverting (`S`) the select input (`SINV`) to the buffer. Like this, the clock signal is routed to the clock trunk placed vertically in the middle of the device. To feed the horizontal branches, the tiles `"CLKVD1"`, `"CLKVD2"`, `"CLKVU1"` or `"CLKVU2"` need to be configured, depending on the clock net to be routed. The following commands supply the horizontal branches to the left and the right with the clock signal sourced on from the top of the device. Affected is clock net number 7.

```
MyJBits.setTileBits(TileYCoord, TileXCoord,
GCLKC_GCLKL7.GCLKC_GCLKL7, GCLKC_GCLKL7.GCLKC_GCLKT7);

MyJBits.setTileBits(TileYCoord, TileXCoord,
GCLKC_GCLKR7.GCLKC_GCLKR7, GCLKC_GCLKR7.GCLKC_GCLKT7);
```

Next step to do is to lead the clock signal from the branches to the routing matrices and thereby to the CLBs. This is done by configuring tiles called `"GCLKH"`.

```
MyJBits.setTileBits(TileYCoord, TileXCoord,
GCLK_UP7.GCLK_UP7, GCLK_UP7.GCLK_B7);

MyJBits.setTileBits(TileYCoord, TileXCoord,
GCLK_DN7.GCLK_DN7, GCLK_DN7.GCLK_B7);
```

Finally the clocks to the slices in each CLB can be connected to the clock net:

```
MyJBits.setTileBits(TileYCoord, TileXCoord, CLK0.CLK0,
CLK0.GCLK7);

MyJBits.setTileBits(TileYCoord, TileXCoord, CLK1.CLK1,
CLK1.GCLK7);

MyJBits.setTileBits(TileYCoord, TileXCoord, CLK2.CLK2,
CLK2.GCLK7);

MyJBits.setTileBits(TileYCoord, TileXCoord, CLK3.CLK3,
CLK3.GCLK7);
```

A look at all these commands reveals, that each causes the clock net to be routed in a either horizontal or vertical direction. Recalling, the two possible pitfalls, described in subsection 4.1.1 we see, that we only need to be afraid of broken or inexistent horizontal clock net sections. These sections are activated in the `"CLKVD1"`, `"CLKVD2"`, `"CLKVU1"` and `"CLKVU2"` tiles. An analysis of how the above JBits commands affect full bitstreams reveals the following. The changes due to the manipulations in the tiles `"CLKT"`, `"CLKB"`, `"CLKVD1"`, `"CLKVD2"`, `"CLKVU1"` and `"CLKVU2"` affect only the very beginning of a full configuration bitstream. That

means configuration words with low addresses. However, these addresses never appear in partial bitstreams. This means that the clock net needs to be set up in the full bitstreams already. Thus it is impossible fot a partial bitstream to cut or configure a horizontal clock line.

4.1.3 The Quick Solution

So when generating bitstreams for an application the designer needs to find out, which branches of which clock net are needed to be active in any configuration. Then he has to assure that in the inital full configuration exactly these branches are activated, for instance because they are used by some dummy logic. The easiest way to do so is to instantiate a register for every used branch of each used clock net. It has to be assured, that this register is placed in a location on the chip, where the respective clock net branch is supplying its signal. One way to do so would be to use a placement constraint in the `.ucf` file. Unluckily, these constraints are not supported in modular design. An easy workaround is to generate a large counter within the design for the *full bitstream*. 128 bit with the XC2V3000. The necessary registers will then be placed in one vertical column across the whole chip, thus using all six branches of the clock net on one half of the FPGA (see section 4.1). So the registers and their clock inputs will be overwritten by any partial reconfiguration of that section but not the activation of the — now possibly unused — clock net branches. So the clock signal will be available again to following partial reconfigurations.
In all designs done during this thesis the solution to the problem is even more simple. The only clock used at all is the 50 MHz system clock. This clock the largest use of it is in the OS frame. Consequently all necessary clock net branches are activated anyway.

4.2 Transient Effects

To understand the impact of transient effects in partially dynamic reconfigurable designs, we must bear in remembrance, that during partial reconfiguration logic shall be able to continue its execution. It was already shown in the previous section, how this is facilitated in terms of clock net. It was also stated, that signal lines must not cross borders between different task slots to avoid them from being disrupt by the partial reconfiguration. In this section transient effects on signal lines are discussed. We have to pay attention to those, because the possibility to continue calculations is worthless, as long as the data being processed is not assured to be valid. So subsection 4.2.1 deals with the reason for transient effects while subsection 4.2.2 shows how a situation during dynamic reconfiguration can be handeled without transient effects affecting processed data.

4.2.1 Why Do Transient Effects Occur During Reconfiguration?

Other than clock signals, data signals have a much larger number of routing resources, they are able to be attached to. Consequently there are also several ways, how a signal can be guided between two pieces of logic or through a task. See figure 4-3 for a visualization: *Task A* leaves enough space to pick the timing-optimal routing. Logic in Task B forces the routed line to leave the direct path. Task C even routes the signal through its own logic. All three of them produce transient signals, when exchanged for each other.

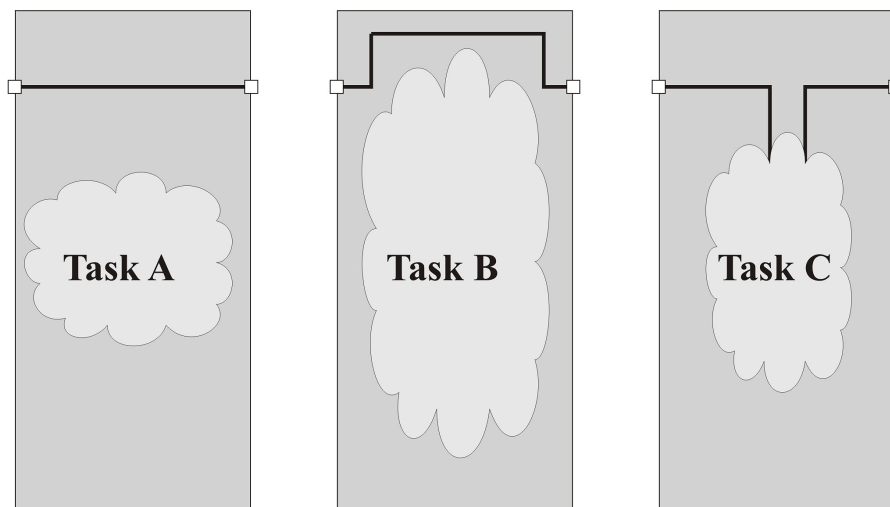


Figure 4-3
Different routing for signal lines through tasks

These transients occur, because the existing signal line cannot be maintained during reconfiguration. Instead it has to be erased and replaced by the new one. Thereby the signal traversing the task slot is interrupted. So during the time of partial reconfiguration it has to be assumed that signals entering, leaving or traversing the reconfigured task are flawed. There is no easy way of having communication lines routed the same way for every implementation of a task. This is because the XILINX routing tool uses a non-deterministic algorithm to connect the placed elements.

4.2.2 Ways to Deal With Transient Effects

In an earlier work [3] Erni and Reichmuth already pointed at the *breaks during the bitstream-download*. Their solution was, to *freeze* all critical signals (Reset, ReadEnable and WriteEnable) during the download. Thereby the affected signals were stored in a flip-flop and conserved until configuration was complete and the *freeze signal* released. To ensure, that the freezing signal is not interrupted itself, it was carried to a pin in the *OS-Middle* and connected to the left and right OS frame by means of an external wire. To recall the situation, please refer back to figure 2-2.

The idea, how to suppress influences of transient effects, realized during this thesis, adheres the idea of *freezing* signals. The difference lies in the fact that there is no need for external wiring anymore. Instead the operating system can initiate a freeze phase, whose duration is proportional to the size of the task to be reconfigured. During this *freeze time* every task neither reads nor writes any signal outside its own area, but it can still continue with intermediate computations. The duration of the freezing is submitted to the tasks via a dedicated signal line. The minimal *freeze time* for a full configuration is $t_{full\ reconfig} = 28\ ms$, this time was reached and measured in [8]. There are 2 ms overhead and 26 ms configuration time. Configuration time degrades linear with the area size to be reconfigured. Therefore one minimal-sized task should take about $t_{part\ reconfig} = 9\ ms$ for reconfiguration. So the freeze signal needs to be active for one clock cycle if a minimal-sized task is to be configured. For each additional w_{min} in task size the signal needs to be kept `HIGH` for an additional period, to keep the freeze signal active long enough. So if the operating system needs to reconfigure a task with a width of $3 * w_{min}$ the freeze signal will be pulled up for 3 periods. As soon as the freeze signal is released a counter is started in each task to freeze all access to the bus for the following $3 * t_{part\ reconfig}$. An example code for the implementation can be found on the CD attached to this report.

5

Communication on the R-FPGA

In sections 2.3 and 4.2 the communication on the R-FPGA has been discussed from the point of view, that they might cause a lot of trouble, if communication lines between interacting modules were not handeled carefully. Thereby almost no information was given on how these modules communicate. Therefore this chapter covers this topic. The chosen bus system is discussed in section 5.1. How the arbitration of the bus is done is topic to section 5.2. Section 5.3 discusses the bus access controller provided to the user tasks in the reconfigurable area. These sections contain the ideas from [29] explained in some more details towards the implementation.

5.1 The Task Communication Bus (TCB)

Choosing an adequate communication infrastructure for the RHWOS on R-FPGA is very much a matter of weighing up data throughput with necessary resources. Maximum throughput would be reached if every task had its dedicated wires, turning the communication infrastructure into a crossbar. Recalling that there are up to five user tasks and two OS frame parts participating in communication on the FPGA, it is evident that a crossbar would demand way too many resources (i.e. FPGA area!). Therefore a shared bus structure was chosen to serve as communication resource for the R-FPGA. Let's call this bus *Task Communication Bus*, or abbreviated *TCB*. Bus arbitration is done in the OS frame with the *Bus Arbiter Left (BARL)* and *Bus Arbiter Right (BARR)*. User tasks have access to the bus by means of the *Bus Access Controller (BAC)*. Figure 5-1 gives an overview on the TCB infrastructure.

As described in subsection 3.1.2.1 the TCB needs to be routed through *bus macros* when traversing module boundaries. The direction, these bus macros are guiding the signals to, is set by constants during design entry. Thus it is not possible to have a bidirectional bus. Therefore the TCB consists not only of one bus, but much more two busses. One which is transmitting signals from the left hand side to the right hand side, and one which is transmitting signals from the right hand side to

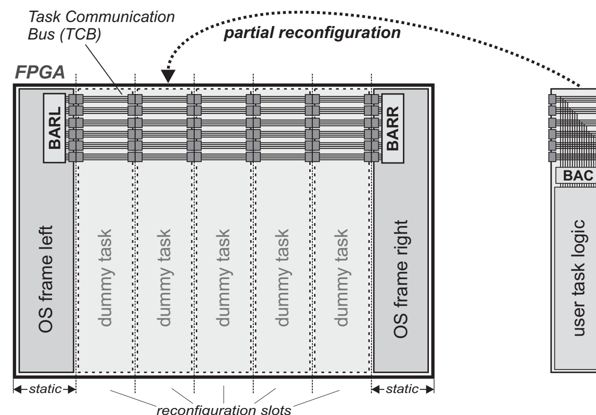


Figure 5-1
Task Communicaton Bus Overview

the left hand side. These 'sub-busses' are arbitrated seperately, consequently two communication events in opposite directions can proceed at the same time. From here the two parts of the bus are called *west-east-bus (WE-bus)* and *east-west-bus (EW-bus)*. These namings refer to the direction of the data being sent, they do not necessarily conform with the direction of control signals also being part of the bus. Such signals are divided into signals using shared bus lines and signals dedicated for commmunication between a single task and the OS frame:

1. Shared bus lines:
    - Task ID signals, identifying the task, that accesses the bus
    - FIFO ID signals, denoting the FIFO, which the task accesses
    - Read/Write request signals
    - Read/Write acknowledge signals
    - Read/Write error signals
2. Dedicated signal lines:
    - Request Read/Write bus signals
    - Grant Read/Write bus signals
    - Reset, enable, ready and done signals, used for task control

The communication protocol used on the TCB is an asynchronous handshake. The reason why this protocol was chosen instead of a synchronous one is due to timing problems on the bus. The observation was made in the design described later in section 6.3.
Communication between tasks or between tasks and OS frame elements is always done by means of FIFO buffers. So any read or write access on the bus targets the memory management unit (MMU) in the OS frame. The MMU is developed by Krisofer Jonsson, so please refer to [8] for more details about the MMU.

5.2 The Bus Arbitration OS Service

The motivation for implementing the bus arbitration is obvious. There might be up to five user tasks configured to the reconfigurable area of the device. Concurrent bus access by more than one such task must be made impossible. Still there must be fair treatment of all tasks requesting the bus at the same time. For these reasons and for its simplicity a round-robin-style algorithm is used. Figure 5-2 shows the finite state machine used to implement the bus arbitration.

As long as no task requests the bus, the bus arbiter scans the request lines of

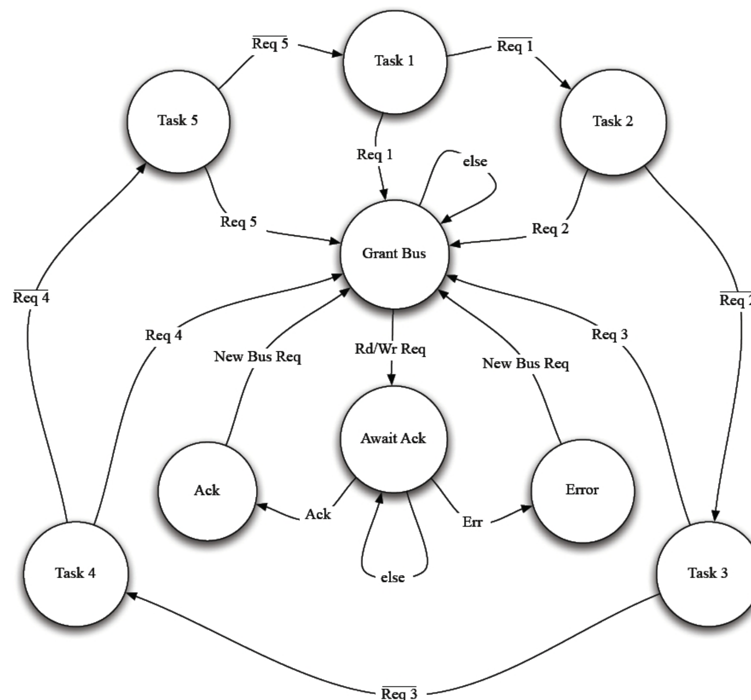


Figure 5-2
Finite State Machine for the Bus Arbitration

all tasks in circular order. When a request is detected, the grant is asserted in state ”Grant Bus”. At the transition to this state, the number of the task currently accessing the bus is stored in a seperate register. This allows to prefer the request of the other tasks as soon as the current access is stopped. So if a transaction is acknowledged and another request (”New Bus Req”) from a different task is already pending, the arbiter FSM immediatly passes to the state ”Grant Bus”, where the bus is granted to the newly active task. Again the task number is stored.

There are some transitions which are not present in figure 5-2 for the sake of clarity: When a transaction finishes — due to either an *acknowledge* (”Ack”), *error* (”Err”) or *bus release* (not ”Req i”) — and no other bus request is pending, the FSM leaves its current state (”Grant Bus”, ”Await Ack”, ”Ack” or ”Error”) and re-enters a ”Task i” state. So the circular scanning of request lines is resumed.

5.3 The Bus Access Controller

The idea behind the *Bus Access Controller (BAC)* is to make the complete Bus Structure, which is needed for communication between the user tasks and the OS frame, transparent for the user tasks. This means that the user task does not have to request the bus and wait for the grant signal from the bus arbiter, before it can accomplish a data transaction. Also a user task does not need to know about the communication protocol used on the bus. The only rule it has to follow is: After initiating a data transaction to or from the OS frame, applying a `DataReady` or `ReadRequest` signal to the BAC, every following `DataReady` or `ReadRequest` signal is ignored, until the transaction has been acknowledged by the OS frame.
In case of a failed transaction (due to full or empty FIFO), the BAC releases the bus, to grant fair access to the bus to other user tasks. But shortly after it will re-request bus access for another attempt on the same transaction. This is done until the transaction is acknowledged, or the operating system detects the repeated failure (in fact it should detect the full FIFO). The operating system then can react in several ways: Either a task is started, which reads from the affected FIFO, or the user task is either reconfigured or reset and/or disabled. The finite state machine showed in figure 5-3 shows the behaviour of the bus access controllers. BACs controlling read and write accesses to a bus have the same behaviour.

5.3.1 The BAC - User Task Interface

Another benefit of the BAC is, that it provides a unified interface for any user task to be implemented. This interface consists of three groups of signals.

1. Write Interface
   - `DataRdyinxSI` is used to initiate a write transaction.
   - `WriteDataxDI` transmits the data to be written.
   - `WriteAckxSO` signals, that the data has been written successfully.
2. Read Interface
   - `ReadReqxSI` requests a read transaction.
   - `ReadAckxSO` reports, that the requested data is available.
   - `ReadDataxDO` carries the requested data.
3. Control and Status Signals
   - `ResetxRO` is a reset signal, that enables exclusive reset of the attached user task. The signal is thought to be controlled by the operating system (i.e. the software part of the OS, running on the C-FPGA).
   - `EnablexSO` allows the operating system to stop the computations in a user task. Initiated data transactions can not be intercepted by removing this enable signal.

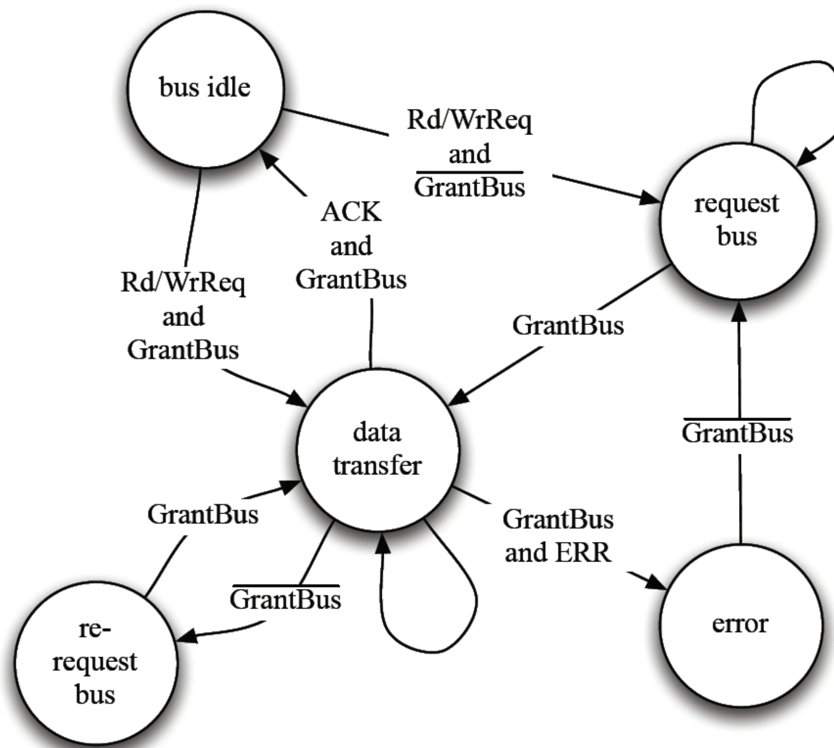


Figure 5-3
Finite State Machine Describing the Behaviour of the BAC for Read and Write Interfaces

- `ReadyxSI` is set by the user task to indicate, when it is ready to process data. For instance after having completed a setup procedure.
- `DonexSI` is assured by the user task, when it has finished its work and therefore could be reconfigured.

6

Design and Implementation

The implementation of the *Reconfigurable Hardware Operating System* services on the R-FPGA was done in an incremental style. In the beginnng the verification of reconfiguration capabilities was the main goal of the designs. The first of these is called *"LED counter"* project. Its purpose, functionality, implementation and results are topic to section 6.1.
In the second design described in section 6.2, the concept of *granular variable-sized tasks* was verified.
In the following designs (sections 6.3, 6.4 and 6.5) more and more OS services were added to prove that their concept is realizable.

6.1 Experiment 1: LED Counter

The "LED counter" was chosen as a starter project for partial reconfigurable design for several reasons. Firstly it should be as simple as possible to not run the risk of semantical and/or syntactical errors. Therefore a simple counter was chosen. Secondly the success of the project should be easily observable. That's why the seven most significant bits (MSBs) of the counter are carried to the LEDs on the XFBOARD. In addition also the signal from one switch on the XFBOARD was carried to a LED. In spite of the simplicity of the project there should also be obtained some insight, that helps to push the project farther. On the one hand a test is done whether a bus structure as described in section 5.1 is possible to implement. On the other hand a first partial reconfiguration should be done.
To keep this second aim simple, the user task only changes the order of the transmitted counter bits. More precisely the MSB is exchanged with the LSB, the second is exchanged with the last but one, and so on. Thereby no logic is applied to the data in the user task. So there are the OS frames left and right connected by the TCB (without arbiters and bus access controllers). In between the OS frame parts remains the 80 CLBs wide reconfigurable area. In a first attempt to implement the

design, the routing in the 10 four CLBs wide dummy tasks failed due to insufficient routing resources. Therefore the width of the tasks was broadened to eight CLBs (this was already noted in section 2.3). Figure 6-1 shows a view of the finally implemented "LED Counter" project: The counter is located in the right OS frame. The signals addressing the LEDs are routed on the EW-bus through the reconfigurable area to the left OS frame, where they are sent back on the WE-bus to the right OS frame. There they are finally written out to the LEDs. So obviously the bus is not a shared medium in this design, as it was described in section 5.1. Much more it is dedicated to the counter signals and the signal from the switch.

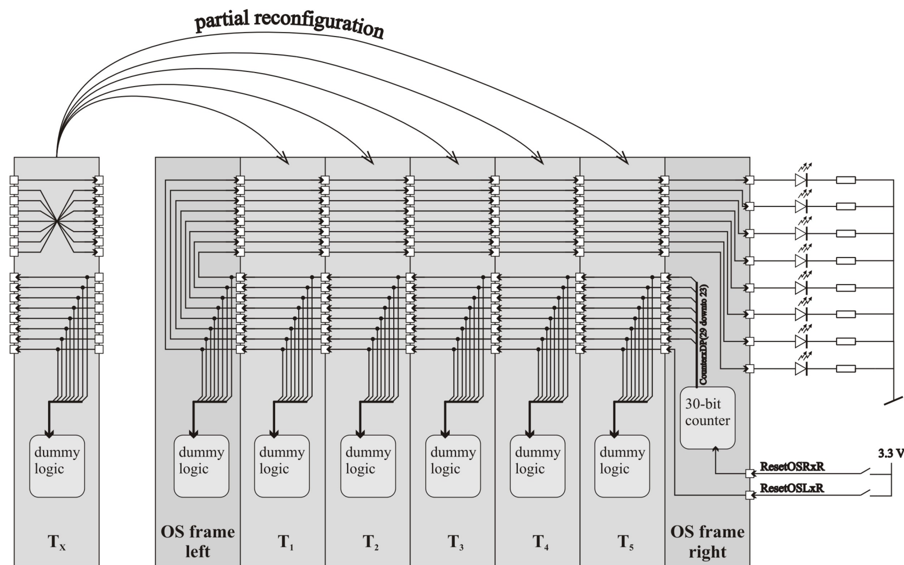


Figure 6-1
Schematic of the LED Counter Project

During the implementation of the "LED counter" project it turned out to be necessary to include dummy logic into modules where only routing is required. Otherwise the complete module got optimized away. Such dummy logic is nothing but signals carried into a register, whose output is never used. Still these registers would get optimized away, so the attribute "keep" must be applied to the registers outputs. The completely implemented design performes as desired. Additionally, during reconfiguration of the user task, the transient effects described in section 4.2 are observable. They appear as somewhat chaotic behaviour of the LEDs. This is due to the unrouting and rerouting in the area being reconfigured, mixed with the fact, that the counter is not stopped for the reconfiguration time.

6.2 Experiment 2: Knightrider

In this second project the idea was to implement user tasks that do not just route but rather generate data. Again the result should be visible. Therefore again the LEDs of the XFBOARD were targeted with the produced signals. As a further extension the signals produced in different tasks are not exclusively written on the bus. The

signal on the bus is much more a superposition of the different OR-ed user task signals. So the functionality of a single user task is to generate a ”Knightrider” signal at a certain speed. This speed is different for the five user tasks targetting the five different positions in the reconfigurable area. Like this it is possible to observe the operation of all concurrently configured user tasks. The operation of

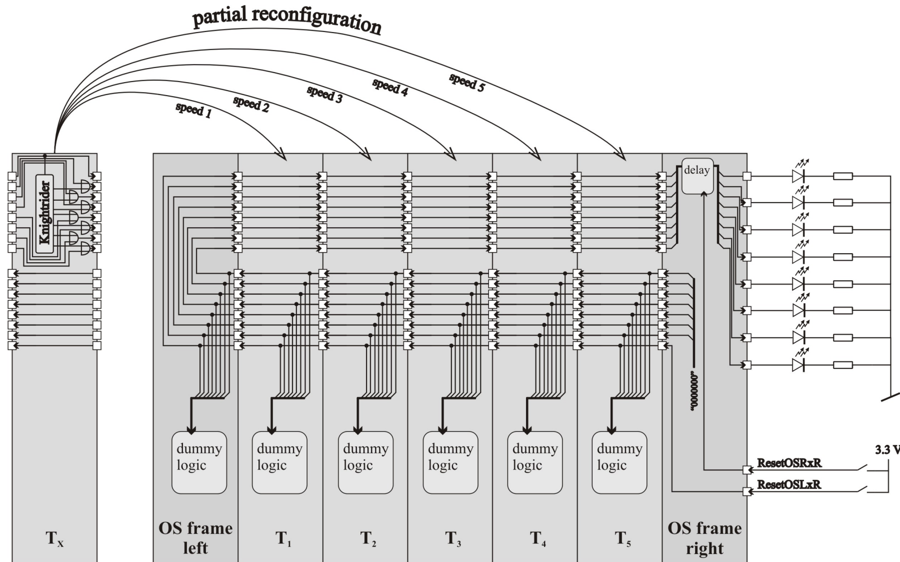


Figure 6-2
Schematic of the Knightrider Project

this project is as desired. All five user tasks can be configured to the device. So up to five LEDs can be watched running back and forth in knightrider-style. Every single one can be removed by configuring the respective task slot with the matching dummy task.

There were two other observations made. The first was, that some of the user tasks need to be reset, before they operate properly. The fact that only some of the five versions of the user task are affected is somewhat confusing, as the implemented code does not differ besides the size of the counter used to determine the speed of the respective signal. So it is reasonable, to implement the possibility to reset tasks individually as described in section 5.1. Implementing only a global reset would undo all advantages of dynamic partial reconfigurability. A second observation made associates with the partial reconfiguring process. After a partial reconfiguration it can happen, that the reconfiguration logic of the device is in a sort of error state, although the configuration actually has succeeded. However this state is left as soon as the status of the configuration logic is either read out or after any other partial reconfiguration. This reconfiguration will not succed then in the first attempt, but surely in the one thereafter.

Thereupon, the ”Knightrider” project was pushed forward one more step. To prove, that the concept of the *granular variable-sized resource model* is successfully applicable, a dummy task with width $2 * w_{min}$ was created. This way it is possible to overwrite two ”knightrider” tasks at one partial reconfiguration. Now it can also be shown that when larger dummy task gets overwritten by a smaller task, a

dummy task must be additionally configured to have a complete bus structure again.

6.3 Experiment 3: Sawtooth

The ”Sawtooth” project was initiated to add some small operating system functionality to the design. Therefore instead of the LEDs the 16 bit audio codec was chosen as the utilized output device. So the necessary service provided by the operating system consisted of the audio driver, implemented in an earlier student thesis [9]. The audio data, which is produced in a user task, is a sawtooth signal. It is written into a FIFO in the right OS frame. From there the audio driver reads the data to be played. In this project only the right-most user task is able to sent its signal to the right OS frame. It is cutting off the signals coming from user tasks on its left hand side. Then it only puts its own signal on the bus. See figure 6-3 for a view on the project.

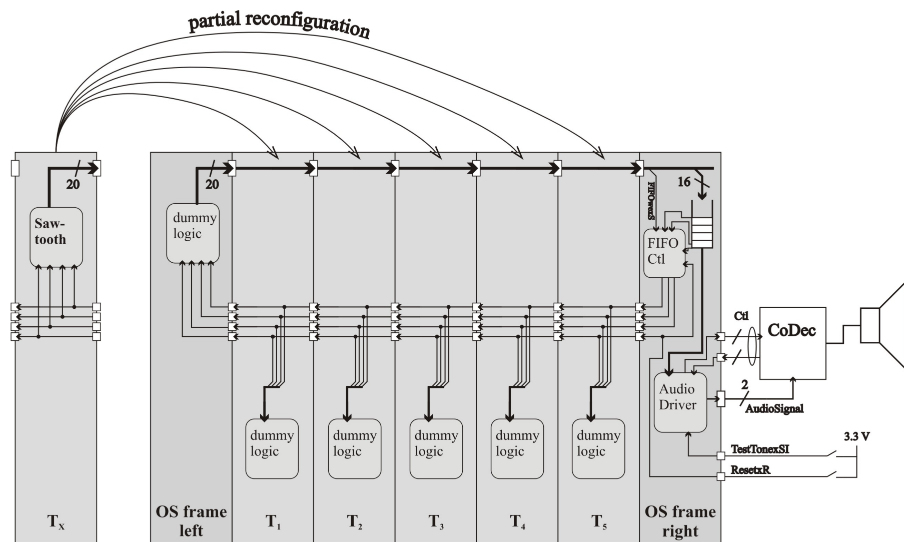


Figure 6-3
Schematic of the Sawtooth Project

It was the ”Sawtooth” project, where it became obvious, that the delay on the bus is too large to have a 50 MHz synchronous communication. This showed as the signal generated in the left-most task slot appeared with the doubled frequency, compared to the one generated in the other task slots. So obviously every other data word was lost on the bus. The analysis of the full bitstreams without the user task logic showed the dummy signals from register outputs in the area of the left OS frame traversing the whole device to the right OS frame area. Thereby six bus macros were travesed. Figure 6-4 shows an extract of the timing analysis done with the ”Timing Analyzer” tool by XILINX.

For each bus macro 0.546 ns combinational delay was estimated. Around 2.144 ns

```
================================================================================
Timing constraint: NET "clkxci_ibufg" PERIOD =  20 nS   HIGH 50.000000 % ;

 66649 items analyzed, 17 timing errors detected. (17 setup errors, 0 hold errors)
 Minimum period is  21.595ns.
--------------------------------------------------------------------------------
Slack:                  -1.595ns (requirement - (data path - clock path skew + uncertainty))
  Source:               u_os_left/delayed1xdp_0 (FF)
  Destination:          u_os_right/osrweregxd_10 (FF)
  Requirement:          20.000ns
  Data Path Delay:      21.587ns (Levels of Logic = 7)
  Clock Path Skew:      -0.008ns
  Source Clock:         clkxc rising at 0.000ns
  Destination Clock:    clkxc rising at 20.000ns
  Clock Uncertainty:    0.000ns
  Timing Improvement Wizard
  Data Path: u_os_left/delayed1xdp_0 to u_os_right/osrweregxd_10
    Delay type         Delay(ns)  Logical Resource(s)
    ----------------------------  -------------------
    Tcko                  0.568   u_os_left/delayed1xdp_0
    net (fanout=6)        0.864   u_os_left/delayed1xdp(0)
    Tilo                  0.439   u_os_left/_n00181
    net (fanout=1)        1.358   busseg_bm1we31xd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  2.144   busseg_bm1we3rxd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  2.144   busseg_bm2we3rxd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  2.144   busseg_bm3we3rxd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  2.144   busseg_bm4we3rxd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  2.144   busseg_bm5we3rxd(2)
    Tio                   0.546   Delay has no logical resource correlation.
    net (fanout=1)     e  3.992   busseg_bm6we3rxd(2)
    Tdick                 0.370   u_os_right/osrweregxd_10
    ----------------------------  ---------------------------
    Total                21.587ns (4.653ns logic, 16.934ns route)
                                  (21.6% logic, 78.4% route)
```

Figure 6-4
Timing Analysis of the Sawtooth Project (excerpt)

routing delay between each pair of bus macro were added. Together with some other combinational delay in the OS frames, the desired 20 ns period could not be reached anymore. So the decision to use the asynchronous handshake protocol was made, to prevent further timing problems on the bus.

6.4 Experiment 4: Write Communication

This project introduces for the first time a bus arbiter and a bus access controller. So access to the bus needs to be requested and can only happen after the request was granted by the bus arbiter. So in the ”Write Communication” project the user task contains the *bus access controller* (BAC) for the bus, which sends data to the right OS frame. The arbiter is located in the right OS frame. The incoming signals connect to the memory management unit (MMU; also located in the right OS frame). To configure the ”MMU” from the CPU-FPGA the ”OS Bridge” module needed to be added. This component was designed in a previous masters thesis [11], more information on the ”OS Bridge” can be found there. The data produced by the user task is again a sawtooth signal to be output to the audio codec. The sawtooth task was implemented according to the interface described in subsection 5.3.1. Thus the sawtooth component does not need to worry about requesting the bus. The complete design is outlined in figure 6-5.

The implementation and debugging of this design was a painstaking process. A

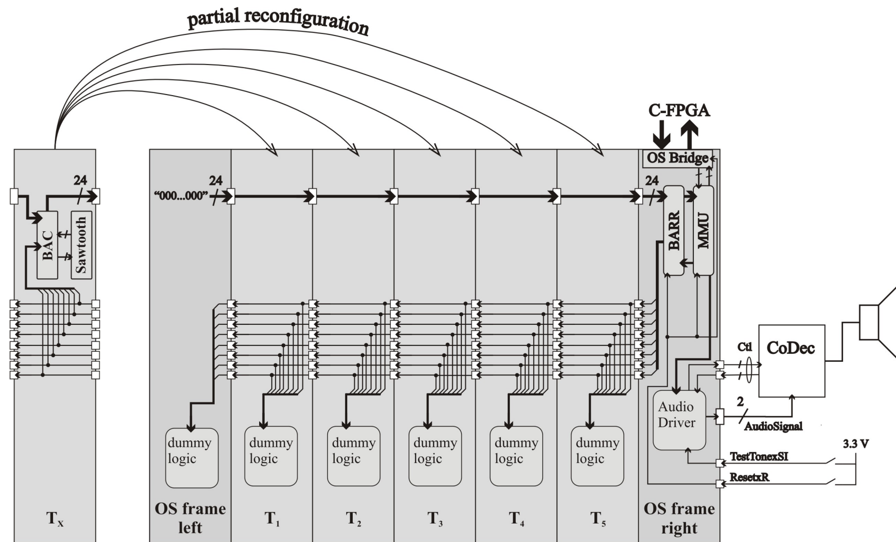


Figure 6-5
Schematic of the Write Communication Project

complete implementation took several hours. Yet the earlier projects showed that most probably the bus structure built of bus macros caused a lot of computing during 'place and route'. In the "Write Communication" project it seems that especially the MMU, being a component with large routing requirements, slowed 'place and route' down. This is an estimation, based on observations on several failed implementation runs. The implementation tools do not give a lot of information why routing was unsuccessful. But partially routed designs can be watched in "FPGA Editor" and unrouted signals were mostly related to the MMU. However in the n-th attempt a successful implementation was obtained, letting the signals from the "sawtooth" user task pass the bus to the MMU and from there to the audio driver and finally the audio codec.

6.5 Experiment 5: Loop Back

The "Loop Back" project is an attempt to add a read interface to the bus access controller. Thereby allowing a user task to read from a FIFO in the OS frame and write the data back to another FIFO. At this time the operating system on the CPU-FPGA allowed to write and read FIFOs in the right OS frame. So the verification of the design should be no problem. Additionally, the infrastructure to freeze, enable/disable and reset existing user tasks was included in the code. Again an outline of the project is shown, this time in figure 6-6.

However a successful implementation could not be reached. Most implementation attempts stopped because of unsuccessful routing. There was one design which was routed completely except for four constant signals, as visual inspection of the design in "FPGA Editor" revealed. Since these signals were constant `'1'`, the attempt was

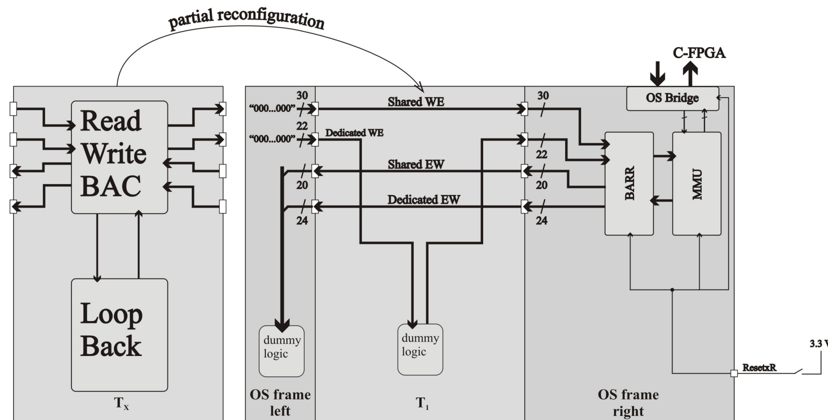


Figure 6-6
Schematic of the Loop Back Project

taken to route them manually to nearby V_{cc} pads. But the received bitstream did not show the desired behaviour. It remains unclear, whether the driving capabilities of the pad are insufficient or any other reasons are causing the failure. Also a timing analysis revealed some very slow paths in the MMU. So the design was rerun for half the clock frequency. Still this attempt brought no improvement.
Regrettably there was no time left to do further debugging on the "Loop Back" project, since the time available for this masters thesis was nearly over. The incomplete implementation files are provided on the attached CD as well as all successful implementations described in this chapter. So in a following thesis the status of this currently unsuccessful project could hopefully be ameliorated.

7

Achievements and Outlook

7.1 Achievements

During this thesis several steps towards a *Reconfigurable Hardware Operating System* for the XFBOARD were successfully taken.

The *Granular Variable-Sized Resource Model* was constructed. This model allows to configure the reconfigurabe device area of the R-FPGA with variable sized modules, without running a risk of discontinuous communication lines.
The implementation of such a partially reconfigurable design with several user tasks turned out to be a laborious task. So a piece of software called "XFOSGEN" was built, to produce the complex VHDL code. Additionally "XFOSGen" provides batch- and settings- files that automatize the whole *Modular Design Flow* for partially reconfigurable designs.
Two possible pitfalls in dynamically partially reconfigurable designs were analyzed. First the impacts of partial reconfigurations on the clock net were evaluated. Resulting in the insight, that the clock net per se is neither activated nor disabled by user tasks. Much more the information about all used clock nets needs to be contained in the full bitstream, which is configured initially in each application.
The second pitfall analyzed, was the transient effects on communication lines between different modules during partial reconfiguration. To avoid negative impact of these phenomenons, the *freeze* functionality was presented.
Further the task communication bus (TCB) was chosen to connect OS frames and user tasks. Bus Access Controlers provide access to the bus via a standardized interface, removing the burden of the bus request procedure from the user tasks. A fair bus arbitration concept was found and implemented.
To prove the concepts above, they were implemented in a step by step manner. OS services implemented in other theses, were also included in these designs. The "LED counter" project was done to demonstrate partial reconfiguration and the concept of the *dummy tasks*. In the "Knightrider" project user tasks were implemented to produce a LED signal. Thereby it was shown that several user tasks can run con-

currently. The concept of the *Granular Variable-Sized Resource Model* was proven by overwriting two minimal-sized user tasks with one larger dummy task.
In the "Sawtooth" project, an audio driver has been successfully added to the right OS frame. Plus the insight was gained, that the delay on the *task communication bus* will be too high to maintain synchronous communication.
The "Write Communication" implementation successfully added more OS services to the design. These were *Bus Arbiter Right* (BARR), *Bus Access Controller* (BAC), *Memory Management Unit* (MMU) and *OS Bridge*.

7.2 Outlook

Generally it can be said, that there are still a lot of things left to do in the development of a RHWOS for the XFBOARD. There are services coded and ready to be included into the OS on the R-FPGA. The ethernet driver was done in [8]. A VGA video driver was developed in [4]. The video codec is connected to the left side of the R-FPGA, therefore several OS services would be necessary to make use of it. Firstly a *MMU* will be necessary in the left OS frame. Also a *OS bridge* to the left OS frame will be of use. It has to be decided how these elements will connect to the *task communication bus*. This rises the question, if new seperate busses will be inserted with their own arbiters, or if the existing *EW-* and *WE-busses* are used for communication with the left OS frame. If the latter is the case arbitration of the existing busses will sureley be more complicate.
"XFOSGen" also has a large improvement potential. Preconditioned all OS services are successfully implemented. It would be nice to augment "XFOSGen" to a application specific operating system generator. This means, that depending on the application to be developed, the user can select any OS service he needs. "XFOSGen" will then produce the design files with only these components instantiated, thereby providing best possible OS frame area allocation.
Another option to improve the RHWOS would be to analyze partial bitstreams on their relocatability. This idea was shortly elucidated in section 2.3. If tasks were relocatable, they need not be stored in several versions for different task slots. Instead a *Task Preparation Unit* (see [29]) would manipulate one stored bitstream to configure the desired task slot before partial reconfiguration takes place. However, to be able to do so without errors, bitstreams and the whole device configuration process need to be extensively analyzed and understood.
The topic of *task preemption* could also be addressed. The problem here is that the current state needs to be either read back to the C-FPGA or stored in BRAM, SRAM or SDRAM. This is to enable an error-free continuation of the task after preemption.
There are also interesting applications ready to be implemented on the XFBOARD with RHWOS. Namely the "MPEG decoder" designed in [30] contains functions which are coded in hardware to speed the whole application up. Also the "Spectrum Analyzer" and "Turbo Decoder" designed in [4] contain functions accelerated as hardware tasks.

Although countless other improvements and applications exist, there are limitations to be expected. Open questions remain. For instance it is uncertain how far the bandwidth of the *TCB* will satisfy the requirements of more complex tasks. The same holds for the *OS bridge* and the communication requirements of the RHWOS between C-FPGA and R-FPGA.
There are observations made during the different implementations described in chapter 6, which might cause for concern. First there are rapidly increasing time requirements of the implementation of designs, namely *place and route* lasts longer with every added OS service. While a "Knightrider" project implementation with five dummy tasks and one user tasks took 1:30 hours, the "Write Communication" project took 3:50 hours, the "Loop Back" even took 6:15 to implement.
Also an increasing number of uncomplete routing efforts was observed.
Lastly the differences in timing analysis between stand-alone implementations of an OS service and the same service included in the OS frames are large — to the disadvantage of the version included in the OS frame.
Based on these reasons the estimation could be done, that routing resources of the Virtex-II FPGA are not sufficient in number or density for a complex structure as a reconfigurable hardware operating system. It is an assumption at this stage, based on observation of the behaviour of the XILINX ISE tools. Only future work will show, if it is true.

7.3 Visions of FPGAs Produced for RHWOS Platforms

Today's FPGAs are optimized for use in static designs. The ability of dynamical partial reconfiguration is a rather recent acquired feature. Also the manufacturer does not expect the user, to make use of the partial reconfigurability for more than one or maybe two fixed modules. Maybe this thesis was the first attempt, to push this concept so far, thereby possibly asking too much of the devices capabilities. So for the last part of this report let's make the assumption that todays FPGAs (for example the Virtex-II) are not exactly the optimal product to implement a partially reconfigurable device as intended on the XFBOARD. What characteristics would make an FPGA more suitable for such a design?
One great improvement would be, if an FPGA was not only column-wise partially reconfigurable. If reconfigurable areas could be rectangular of any size, area utilization was a lot better although scheduling and allocation will become more complex (see [26] for a more detailled discussion). With such a model it was also possible to have only one OS frame part, for instance made of four squares forming a ring at the border of the device, thereby having all I/O resources disposable and avoiding "inter-OS frame" communication difficulties. Like that more than two user tasks could have a direct connection to the OS frame. So it would be possible to manage communication from the OS to a multiplicity of tasks without a bus or large resource overhead via dedicated lines. Even the transient effects during reconfiguration could become harmless, as there are no third participants on a

communication line which is being reconfigured.
Another idea, which might bring some improvements in performance is to mix ASIC and FPGA technology on one device. If there was a fixed configuration of OS frame services, they could be integrated in ASIC manner, while the rest of the device would be reconfigurable FPGA technology. Thereby the danger of a computation-, communication- or routing-resource-bottleneck in the OS frame is largely reduced. That is because ASIC integration density is much higher, and therefore delays on logic and routing are much smaller than on an FPGA.

No one knows, whether these or similar ideas will be applied in future FPGA technology. At least it is most probable that logic densities in FPGA integration will increase. As a consequence the application field for FPGAs will continue its growth.

8

Acknowlegement

There are a lot of people out there, who supported me in many different ways. For their effort in connection with my master thesis I am deeply grateful to the following people:
Herbert Walder, my advisor in this thesis, provided great support, project management, inspiration and motivation at any time. He set up a fascinating project and a great team, which it was always a pleasure to be a part of and work for.
Prof. Dr. Lothar Thiele, the head of the Computer and Networks Lab, gave me the opportunity to do my masters thesis at the TIK.
Simon Künzli gave me a quick crash-course in java. He did a great job, no word too much, none too little.
The whole XF**BOARD** and G69 crew: Samuel Nobs, Silvan Wegmann, Kristofer Jonsson, Roman Plessl, Stéphane Racine, Lukas Hämmerle assured a great working atmosphere during the last six months. Their efforts were of inestimable value. On one side the amount of knowledge they were always willing to share and on the other side their outstanding sense of humour made every day in the G69 a great experience. ”Badger, mushroom, snake!”

Last but not least I want to express my gratitude to my parents Rosa and Jürg Steinegger for all their mental and financial support. Their faith in me was always a priceless backing during my studies.

THANK YOU ALL!!!

A

An Example Synthesis Project File

This is an example for a XILINX Synthesis Technology (XST) synthesis project file (`.prj`). It belongs to the right OS frame of the ”Write Communication” project, which was described in section 6.4.

```
work ../../hdl/globals.vhd
work ../../hdl/osbridge_opcodes.vhd
work ../../hdl/bridge_controller.vhd
work ../../hdl/pfid.vhd
work ../../hdl/vfdl.vhd
work ../../hdl/write.vhd
work ../../hdl/read.vhd
work ../../hdl/osbridge.vhd
work ../../hdl/controller.vhd
work ../../hdl/osb_mmu.vhd
work ../../hdl/xfrosbridge.vhd
work ../../hdl/top_mmu.vhd
work ../../hdl/BusArbiterRightv2.vhd
work ../../hdl/os_right.vhd
```

When adding other source files to this list, take into account, that all files must be listed in this file in reverse hierarchical order.

B

An Example Synthesis Settings File

This is a settings file for synthesis, done with XILINX Synthesis Technology (XST). It belongs to the right OS frame from the ”Write Communication” project, which was described in section 6.4.

```
set -tmpdir xst_settings
set -xsthdpdir ./xst
run
-ifn project_files\os_right.prj
-ifmt VHDL
-ofn os_right
-ofmt NGC
-p xc2v3000-4fg676
-ent os_right
-opt_mode Area
-opt_level 1
-iuc NO
-keep_hierarchy YES
-glob_opt AllClockNets
-rtlview No
-read_cores YES
-write_timing_constraints NO
-cross_clock_analysis NO
-hierarchy_separator _
-bus_delimiter ()
-case lower
-slice_utilization_ratio 100
-fsm_extract YES -fsm_encoding Auto
-ram_extract Yes
-ram_style Auto
-rom_extract Yes
-rom_style Auto
```

```
-mux_extract YES
-mux_style Auto
-decoder_extract YES
-priority_extract YES
-shreg_extract YES
-shift_extract YES
-xor_collapse YES
-resource_sharing YES
-mult_style auto
-iobuf NO
-max_fanout 500
-bufg 16
-register_duplication YES
-equivalent_register_removal YES
-register_balancing No
-slice_packing YES
-iob auto
-slice_utilization_ratio_maxmargin 5
```

Please do only change settings after consulting the XST User Guide [22].

C

An Example Settings File for Bitstream generation

Shown below are the contents of a `"bitgen_v2_std.ut"` settings file. It contains information neccessary for proper configuration of a given device. *Configuration mode*, *configuration clock* and *CRC checking* are just some examples. An exhaustive description of all options can be found in the "Development System Reference Guide" for XILINX ISE 6 [17].

```
-w
-l
-m
-g ReadBack
-g DebugBitstream:No
-g CRC:Disable
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:Pullnone
-g M1Pin:Pullnone
-g M2Pin:Pullnone
-g ProgPin:PullUp
-g DonePin:PullUp
-g DriveDone:No
-g PowerdownPin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullNone
-g TmsPin:PullUp
-g UnusedPin:PullDown
-g UserID:0xFFFFFFFF
-g DCMShutDown:Disable
-g DisableBandgap:No
```

```
-g StartUpClk:Cclk
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Match_cycle:NoWait
-g Security:None
-g Persist:Yes
-g DonePipe:No
-g Encrypt:No
-g ActiveReconfig:Yes
-g ActivateGCLK:Yes
```

Again caution is advisable when changing these settings.

D

BitStreamParser

”BitStreamParser” was written during the analysis of the clock net issue, described in section 4.1. Something even better than such a parser would have been a tool, that visualizes, how a bitstream configures the device. Unfortunately such a tool does not exist. Therefore ”BitStreamParser” was written to have at least an idea on how jBits commands affect a bitstream.

First ”BitStreamParser” finds the *dummy word* `FFFF FFFFh` and the *synchronization word* `AA99 5566h`. These two words mark the beginning of the bitstream data. Next instructions matching ”Type 1 Packet Headers” (fig. D-1) and ”Type 2 Packet Headers” (fig. D-2) are decoded.

Packet Header	Type	Operation (Write/Read)	Register Address (Destination)	Byte Address	Word Count (32-bit Words)
Bits[31:0]	31:29	28:27	26:13	12:11	10:0
Type 1	001	10/01	XXXXXXXXXXXXXX	XX	XXXXXXXXXXX

Figure D-1
Type 1 Packet Header

Packet Header	Type	Operation (Write/Read)	Word Count (32-bit Words)
Bits[31:0]	31:29	28:27	26:0
Type 2	010	10/01	XXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure D-2
Type 2 Packet Header

Then the ”Register Address” is decoded. There are *Internal Configuration Registers* listed in figure D-3 and figure D-4. Furthermore *Command Register Commands* are listed in figure D-5.

To explain the meaning of all these commands is not necessary in context with this thesis, therefore the interested reader is refered to [23]. The most important entry is the `FAR` configuration register. This register contains information about where

Symbol	Register Name	Address
CRC	CRC Register	00000
FAR	Frame Address Register	00001
FDRI	Frame Data Input Register (Write Configuration Data)	00010
FDRO	Frame Data Output Register (Readback Configuration Data)	00011
CMD	Command Register	00100

Figure D-3
Configuration Registers (1/2)

Symbol	Register Name	Address
CTL	Control Register	00101
MASK	Masking Register for CTL	00110
STAT	Status Register	00111
LOUT	Legacy Output Register (DOUT for daisy chain)	01000
COR	Configuration Option Register	01001
MFWR	Multiple Frame Write	01010
FLR	Frame Length Register	01011
IDCODE	Product ID Code Register	01110

Figure D-4
Configuration Registers (2/2)

Symbol	Command	Binary Code
WCFG	Write Configuration Data	0001
MFWR	Multi-Frame Write	0010
DGHIGH	De-asserts GHIGH	0011
RCFG	Read Configuration Data	0100
START	Begin STARTUP Sequence	0101
RCAP	Reset CAPTURE (after Single-Shot Capture)	0110
RCRC	Reset CRC Register	0111
AGHIGH	Assert GHIGH	1000
SWITCH	Switch CCLK Frequency	1001
GRESTORE	Pulse GRESTORE Signal	1010
SHUTDOWN	Begin SHUTDOWN Sequence	1011
GCAPTURE	Pulse GCAPTURE Signal (one shot)	1100
DESYNCH	Forces realignment to 32 bits	1101

Figure D-5
Command Register Commands

configuration data is written to on the device. Thanks to detecting this command the information solving the clock net problem from section 4.1 could have been found.

Bibliography

[1] Alliance Semiconductor. *AS7C4096/AS7C34096, 5V/3.3V 52K x 8 CMOS SRAM*, v.1.8 edition, March 2002. http://www.alse.com/pdf/sram.pdf/fa/AS7C34096.pdf.

[2] Asahi Kasei Microsystems Co., Ltd. *AK4563A Low Power 16bit 4ch ADC & 2ch DAC with ALC*, December 2000. http://www.asahi-kasei.co.jp/akm/en/product/ak4563a/ek4563a.pdf.

[3] Andres Erni and Stefan Reichmuth. Inter-Task-Communication in Reconfigurable Operating Systems. Master's thesis, ETH Zurich, Computer and Networks Lab, March 2003.

[4] Andreas Ess and Tobias Gysi. Signal Processing Tasks for RHWOS. Term thesis, ETH Zurich, Computer and Networks Lab, February 2004.

[5] Infineon Technologies. *HYB39S256400/800/160CT(L) 256MBit Synchronous DRAM*, August 2000. http://www.infineon.com/cmc_upload/0/000/018/241/SDRAM_256M.pdf.

[6] Intel Corp. *LXT970A Dual-Speed Fast Ethernet Transceiver*, January 2001. http://www.intel.com/design/network/products/lan/datashts/24909901.pdf.

[7] Intersil Corportaion. *HI1178 Triple 8-bit, 40MSPS, RGB, 3-Channel D/A Converter*, 2000. http://www.intersil.com/design/images/DataSheet.jpg.

[8] Kristofer Jonsson. Components and Services for RHWOS. unpublished at the time of completion of this thesis.

[9] Pascal Lüdi and Daniel Hobi. Audio Playback Tasks for RHWOS. Term thesis, ETH Zurich, Computer and Networks Lab, February 2004.

[10] Samuel Nobs. Prototype Board for Reconfigurable OS. Term thesis, ETH Zurich, Computer and Networks Lab, July 2003.

[11] Samuel Nobs. Reconfigurable Hardware OS Prototype - Part CPU. Master's thesis, ETH Zurich, Computer and Networks Lab, April 2004.

[12] Michael Ruppen. Reconfigurable OS Prototype. Master's thesis, ETH Zurich, Computer and Networks Lab, 2003.

[13] Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 13rd International Conference on Field Programmable Logic and Application (FPL'03)*, pages 575–584. Springer, September 2003.

[14] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 24th International Real-Time Systems Symposium (RTSS'03)*, December 2003.

[15] Inc. XILINX. JBits API. provided with the jBits package. JBits3\Javadocs\index.html.

[16] XILINX, Inc. *XAPP138: Virtex FPGA Series Configuration and Readback*, July 2002. http://www.xilinx.com/bvdocs/appnotes/xapp138.pdf.

[17] XILINX, Inc. *Development System Reference Guide*, ISE 6 edition, April 2003. http://toolbox.xilinx.com/docsan/xilinx6/books/docs/dev/dev.pdf.

[18] XILINX, Inc. *MicroBlaze Processor Reference Guide*, 6.1 edition, September 2003.

[19] XILINX, Inc. *Virtex™-II Platform FPGAs: Complete Data Sheet*, October 2003. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[20] XILINX, Inc. *XAPP151: Virtex Series Configuration Architecture User Guide*, March 2003. http://www.xilinx.com/bvdocs/appnotes/xapp151.pdf.

[21] XILINX, Inc. *XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based*, November 2003. http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf.

[22] XILINX, Inc. *XST User Guide*, June 2003. http://toolbox.xilinx.com/docsan/xilinx6/books/docs/xst/xst.pdf.

[23] XILINX, Inc. *Virtex-II Platform FPGA User Guide*, April 2004. http://direct.xilinx.xom/bvdocs/userguides/ug002.pdf.

[24] Herbert Walder, Samuel Nobs, and Marco Platzner. XF-Board: Prototype Platform for Reconfigurable Hardware Operating System. Technical Report TIK Nr. 193, Swiss Federal Institute of Technology (ETH), Zurich, March 2004.

[25] Herbert Walder and Marco Platzner. Non-preemptive Multitasking on FPGA: Task Placement and Footprint Transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 24–30. CSREA Press, June 2002.

[26] Herbert Walder and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.

[27] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.

[28] Herbert Walder and Marco Platzner. Reconfigurable Hardware OS Prototype. Technical Report TIK Nr. 168, Swiss Federal Institute of Technology (ETH), Zurich, April 2003.

[29] Herbert Walder and Marco Platzner. A Runtime Environment for Reconfigurable Hardware Operating Systems. submitted to FPL 2004, acceptance is not yet confirmed, 2004.

[30] Silvan Wegmann. Video playback tasks for rhwos. Master's thesis, ETH Zurich, Computer and Networks Lab, March 2004.