Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

Run-Time Partial Reconfiguration on the Virtex-II Pro

Stefan Raaijmakers

Abstract

Reconfigurable Computing entails the utilization of a general-purpose processor augmented with a reconfigurable hardware structure (e.g. a field-programmable gate array). Normally, a complete reconfiguration is needed to change the functionality of the FPGA even when the change is only minor. Moreover, the complete chip needs to be halted to perform the reconfiguration. Dynamic partial reconfiguration (DPR) enables the possibility to change parts of the hardware while other parts of the FPGA remain in use.
In this paper, we propose an additional solution to perform dynamic partial reconfiguration by providing a methodology to generate bitstreams for removal of old hardware, and placement and routing of new hardware within an FPGA. This means that functionality can be removed from, and additional functionality can be added to the FPGA at any location. Our solution is able of connecting the additional functionality to the already running parts of the chip. More over, bus macros are no longer necessary and no synthesis is needed to implement the routing. We implemented our solution on a Xilinx Virtex-II Pro series FPGA, specifically the XC2VP30 on the XUP board, and demonstrated that the solution works.

CE-MS-2007-07

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Run-Time Partial Reconfiguration on the Virtex-II Pro

Generating bitstreams for adding and removing hardware and signal routing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Stefan Raaijmakers
born in Haarlem, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Run-Time Partial Reconfiguration on the Virtex-II Pro

by Stefan Raaijmakers

Abstract

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2007-07
Committee Members	:	

Advisor:	Stephan Wong
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Stephan Wong
Member:	Nick van der Meijs
Member:	Georgy Gaydadjiev

Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**
1.1 Reconfigurable Computing . . . 2
1.2 Field-programmable gate arrays . . . 3
1.3 Problem Statement . . . 5
1.4 Goal and Methodology . . . 5
1.5 CE Research in Reconfigurable Computing . . . 6
1.5.1 Delft Workbench . . . 7
1.5.2 Molen . . . 7
1.6 Overview . . . 7

2 Background **9**
2.1 Related Work . . . 10
2.1.1 1 Dimensional Placement, Mostly circuit switched . . . 10
2.1.2 2D Placement, Mostly Network-on-Chip . . . 11
2.2 Xilinx Virtex-II Pro . . . 13
2.3 Xilinx Design Flow for static configurations. . . . 17
2.4 Partial Reconfiguration Using the XAPP290 Design Flow . . . 19
2.5 Proposed Method for Partial Reconfiguration . . . 21
2.5.1 Proposed Design flow, Building the Modules . . . 21
2.5.2 Proposed Approach to Runtime Partial Reconfiguration . . . 21
2.5.3 Advantages and Disadvantages . . . 23
2.5.4 Conclusion . . . 24

3 Work **25**
3.1 Resources in the FPGA . . . 25
3.2 Operating System and Programming Language . . . 26
3.3 The XUP Evaluation Board . . . 26
3.4 Building the Dataset . . . 27
3.4.1 Extracting Switch Settings/Building Wire Databases . . . 27
3.4.2 *fpga_editor* . . . 27
3.4.3 Bitstream Description . . . 30
3.4.4 Extracting Switch Settings/Building Wire Databases . . . 32
3.5 Building Bit-Manipulation Tools . . . 34
3.5.1 Generating Routing . . . 34
3.6 Advanced Tools . . . 35

3.6.1 Hardware Core Isolation 36
3.6.2 Unrouting 37
3.6.3 Hardware Core Placement 37
3.7 Device Composition Description 37
3.8 Partial Reconfiguration 38
3.9 Verification of Data and Methodology 39
3.10 Conclusion 39

4 Results 41
4.1 Testing Feasibility 41
4.2 Testing Runtime Reconfiguration for Building Partial Bitstreams 41
4.3 Comparing Bitstreams of Routes with *fpga_editor* 42
4.4 Simple Example (3 Input AND) for Verification 43
4.5 Partial Reconfiguration, Replacing the AND With An XOR Gate 43
4.6 Automating the Process, Using a Composition File 44
4.7 Conclusion 47

5 Conclusions and Recommendations 49
5.1 Conclusions 49
5.2 Main Conclusions 50
5.3 Future Work 51
5.3.1 Bitstream Generation in Hardware 51
5.3.2 Reduction of Reconfiguration Time 52
5.3.3 Routing 53
5.3.4 Network on Chip 53
5.3.5 System Integration 54

Bibliography 57

List of Figures

2.1 XC2VP7 structures scaled to size in bit-stream 14
2.2 Virtex-II Pro Slice configuration 15
2.3 CLB as seen in fpga_editor 16
2.4 Virtex-II Pro wires as taken from the datasheet 17
2.5 Virtex-II Pro clock distribution as taken from the user-guide 18
2.6 ISE design flow 18
2.7 XAPP290 vertically stretching modules 20
2.8 Exchanging one hardware core for another 22

3.1 Virtex-II Pro wires as shown in fpga_editor 29
3.2 Virtex frame buffer 30
3.3 Text representation of a bit file partially reconfiguring the 5th frame of the first column 31
3.4 Hexadecimal count of bit occurrence in the frames of a CLB 33
3.5 A simple implementation for routing 2 points 34

4.1 Routing two switches to two LEDs and reversing the connections 42
4.2 AND-Routing by ISE (left), our router (middle) and XOR-routing (right) 44
4.3 Device composition input file 45
4.4 logic functions of 3 input AND, XOR and a combination 46

I would like to acknowledge my parents, brother, sister in law, fellow governing board members of O.J.V. de Koornbeurs 2005-2006, everybody at CE and all my friends for their support, and for suffering me as a student for way too long.

Introduction 1

A modern computer system consists of various separate components with a specific function. Even novice computer users should know that the processor is the most important part that governs almost all the operations within the computer system. Many functionalities are integrated on the main board these days, but it is still possible to choose a vast array of extras, for instance: a fast graphics card, or two of them, network cards of different quality and speeds, or special hardware RAID cards for fast and big storage capacity on multiple disks. When the need arises for a new system to be assembled, there are many of choices to be made. Just for a normal IBM compatible (x86) system there are processors available from several manufacturers. Although they will all run normal software, some will have additional multimedia instructions, or 64-bit support, or multiple cores. Having fixed hardware has its disadvantages:

- Chances are that the main processor will not be replaced during the life time of the computer system, so the user will be locked into the technology picked.
- If a new hardware feature becomes available, it is impossible to reap the benefits of it unless you buy this new hardware for your system.
- By now, we are used to update our software on a regular basis. It would be desirable to have this option for our hardware as well. Of course it is always possible to change the graphics card or the processor, but most of the time the user will have to do without these new features until it is time to purchase a new system.

The ability to update hardware just as easily as software will is quickly becoming a reality. Field-programmable gate arrays (FPGAs) are chips that contain programmable hardware. This means that the off-the-shelf component has no functionality what so ever, but using the hardware description of for instance a microprocessor it can be configured to become that microprocessor. It could just as well be configured to become a video processor or a hard disk controller. Until now, FPGAs are used mainly for prototyping and products with relatively small production numbers. However, more applications for their use are being found. The C-one reconfigurable computer [11] is a project in which the Commodore 64, a popular home computer from the 80's, is emulated by an FPGA. The same board can also be used to emulate a Amstrad/Schneider CPC46. Similar projects exist for a number of different home computers which where popular in that period.

Modern computers are too complex to fit inside an FPGA. An implementation would take up approximately 10 times more space and would be 3 times as slow, as compared to a specifically designed chip (ASIC) [6]. However, there are significant advantages of having a general-purpose processor augmented with an FPGA:

- It is possible to implement common operations on these kinds of chips.
- This can alleviate the main processor of computationally extensive tasks.
- A hardware implementation on an FPGA is usually faster than software because it can make more efficient use of the chip resources and it can make better use of parallelism.
- One of the major advantages is that a new algorithm or new standard does not mean there is a need for new hardware, just a new hardware description.

The Computer Engineering laboratory has implemented several algorithms in hardware. For MPEG4, AES encryption/decryption and VOIP applications we have developed hardware accelerators. Imagine a user watching a film over an encrypted connection. Suddenly, his Internet phone rings. He pauses the film and picks up the phone, which sets up a secured connection for the conversation. As we have a hardware description for AES, the decoding of the film and the VOIP implementation, the computer can perform all operations efficiently. Suppose the FPGA is not large enough to hold all three hardware implementations. As both applications use encryption/decryption, the AES hardware has to remain configured. Luckily the film has been paused. This means the MPEG4 decoder can be temporarily replaced with the VOIP hardware. The FPGA can reconfigure one part of its structure while the rest of the system remains in operation. This is called dynamic partial reconfiguration (DPR). It is, therefore, possible to replace the MPEG4 decoder with the VOIP hardware while the AES encryption/decryption remains in use. The user is of course oblivious to all the processes involved.

The research field of using reconfigurable hardware to augment a computer system is called reconfigurable computing (RC). It can be expected that this technology will first manifest itself in embedded devices. For the user it will not be apparent that an HD-DVD or Blue-Ray player will contain an FPGA. It is not unlikely that future codecs, with which the films are stored onto the disk, will change over time. The advantage for the consumer is that they will not need to replace their hardware as often while they can reap the benefits of new developments. Manufacturers can update their hardware to the new standard when needed. Whether this technology will make it to desktop systems is an open issue.

1.1 Reconfigurable Computing

In the early 60's, there was an interest to look beyond the conventional general-purpose machines and to develop new computing paradigms. As a result, reconfigurable computing was conceived [12] as a means to extend the capabilities of general-purpose computing. In reconfigurable computing, parts of a program can be described in hardware, which can result in several hardware implementations for different stages of execution of a single program. Consequently, during program execution the processor has to reconfigure the hardware to the needed functionality. However, technology in the 60's was not mature enough to sufficiently implement the concept of reconfigurable computing. With the introduction of programmable logic devices in 1984 and field-programmable gate arrays (FPGA) in later years, reconfigurable computing became increasingly more accessible.

When using an FPGA for reconfigurable computing, this is referred to as fine-grained reconfigurable computing. This means for instance that the FPGA is made up of small lookup tables which take only a few bits as input to produce single-bit results. All signals are individually routed through the chip. The advantage is that the hardware description can be meticulously tuned to the algorithm, but this comes at a disadvantage. The reconfiguration data is very large, which implies that reconfiguration will take a relatively large amount of time. It also implies that functional densities are low, which adds delay to the signal paths as they have to travel longer distances and can pass through multiple switches. Most FPGAs have additional functional units like multipliers and small RAMs, which are beneficial to many applications to improve their performance. In course-grained reconfigurable computing [14], the smallest structures rarely operate on data pathways with less that 4 bits. Architectures vary extensively, but they have in common that signals are routed through buses instead of individually. Operations on the data can be performed by anything from programmable arithmetic and logic units (ALUs) to small 'processor' cores. This simplifies routing as the number of pathways are greatly reduced, but it means the algorithm has to be mapped onto an existing structure instead of a freely chosen one. It also means that there are a lot less configuration bits, which is advantageous for reconfiguration times. Some architectures are running on specially fabricated chips, others only exist in an emulator. Development for these architectures can therefore be expensive. We will only be targeting fine-grained reconfigurable computing as we primarily have access to this type of technology.

1.2 Field-programmable gate arrays

FPGAs are integrated circuits composed of programmable logic interconnected with programmable networks. This allows for the construction of any digital circuit. Compared to ASICs, FPGAs can be a cost effective replacement if used in relatively small quantities (less then approximately a thousand to ten thousand units). FPGAs have found their way into commercial products, often because they ensure a quicker time to market or improved flexibility. With firmware updates, incorporated FPGAs make it possible to correct small design errors in the hardware during or after production. They can take over signal processing functions from DSPs and are bridging the gap between general-purpose microprocessors or microcontrollers and ASICs. These are all applications in which the FPGA has a fixed function, the configuration of the FPGA is rarely updated during the lifetime of the device.

To most people (including most computer engineers), normal digital chips are composed of standard building blocks, like: logic gates, RAM cells, flip-flops, etc. In a full custom design, these building blocks are freely placeable on the chip. The designers have full flexibility, but this comes at the price of high production costs, as placing the structures on the chip requires many production steps. This technology is only attractive when a large number of the same chip are produced. A cheaper alternative are sea-of-gate chips, which are entirely composed of these standard building blocks, or even just transistors. This reduces the production costs significantly because these gates only need to be connected with a few metal layers. The disadvantage is that the design has to be mapped on to the pre-placed components. As a consequence not all components are being used because it would be impractical, impossible, or too expensive to wire them up. These chips will therefore be larger and slower than a full custom version, but they would be cheaper to design and produce.

Field-programmable gate arrays are composed of standard building blocks, like: flip-flops, multipliers or RAMs, and programmable building blocks in the form of lookup tables and multiplexors. Where a designer would use an AND and an XOR gate to make a half adder, in the FPGA lookup tables are set to represent this AND or an XOR gate. It basically means the component is represented by its logic table. The lookup tables take multiple inputs and , therefore, can represent more then a single gate. It would be possible to build an FPGA just out of lookup tables, but by adding a few extra elements like flip-flops, the functional densities and speed greatly improve. These various elements are usually combined in a regular structure, that Xilinx for instance calls slices. To route the signals in the FPGA, switching elements of various composition are used. This mostly depends on the delay they cause. The critical pathways in a design need to have the fastest connections. For this reason, there are special pathways for carry, clock, and global signals. Carry signals use small switches directly connecting to neighboring slices. Clock and global signals are routed through a fixed network from the center of the chip outwards. This is to minimize clock-skew. Normal signals are routed through switch matrices which connect to wires of varying lengths. These can route the signal to neighboring resources, resources in the vicinity or to chip-spanning wires. As the switches are connected to hundreds of wires it would become too complex a structure if it could connect every wire to the other. Usually the pathways a signal can take are limited.

All these switches and lookup tables are combined with RAMs to contain their configuration information. The FPGA is made of regular tiles containing switches, lookup tables, and their configuration memories. In Xilinx terminology, these are called complex logic blocks (CLBs). The manner the configuration memory can be accessed depends on the implementation. Some devices, like the Atmel 94k series [1], allow for byte-sized access to the memory. Others, like the Xilinx Virtex series [34], use frames which can be much larger (+/- 500–1500 bits). Sometimes memory can be written directly, or it has to be configured serially by moving the data through a large shift register. The speed and granularity of the device configuration is a major factor in successfully implementing reconfigurable computing. Some devices can only be configured in their entirety. This usually involves shutting down the device, rendering it useless for the time it is being reconfigured. The delay caused by reconfiguring the device can be seconds, which is substantial for a computer systems which can perform several billion operations each second. Some FPGAs facilitate a way to reconfigure only a part of the device. In this way the rest of the device can remain in use while a small portion is set up for the new functionality. This is used for runtime partial reconfiguration in reconfigurable computing. The advantage is that reconfiguration delays can be hidden as long as the new functionality is being set up long before it will be in use. Other functions will remain accessible to the computational process. The reconfiguration time is also reduced as the size of the reconfiguration bitstream is related to the size of the structure which is being reconfigured. As we only reconfigure a small portion of the chip, the bitstream size will be much smaller than for reconfiguring the entire device.

1.3 Problem Statement

When reconfiguring an FPGA for a new function, we encounter two problems:

- Changing functionality of the device suffers from lengthy reconfiguration latencies.

Configuring an entire device can take a tenth of a second to several seconds. One solution is to *partially* reconfigure the device, replacing only a small portion of the reconfiguration data to change a functionality or just some parameters. This reduces the reconfiguration time and can increase functional densities for some applications [13]. It can also be used to hide reconfiguration latencies by setting up the next hardware accelerator while the other accelerators are in use.

- For each device within a family and each combination of hardware, a new reconfiguration bitstream has to be synthesized.

Normally, each device type, even within a family, needs a different reconfiguration bit stream. Current methodologies for generating partial reconfiguration bitstreams entail using an ad-hoc manner for generating full device configuration of each device and module combination, and extracting the differences at compile time [16]. It is a cumbersome method to distribute an application, especially when multiple cores are used and interchanged, and comes with considerable restrictions:

- Synthesis of device configuration is done in compile time
- Existing methods use fixed floorplans
- Existing methods use bus macros to affix routing
- Existing methods can not place arbitrarily sized modules

1.4 Goal and Methodology

In this thesis, we introduce a solution to overcome these problems. This research is focused on developing a method that enables (semi-)arbitrary removal and placement of hardware implementations, and if needed, perform the necessary routing to disconnect and connect them to other implementations present in the device. It produces full and partial (re)configuration bitstreams for first time setup and subsequent transitions.

Our goal is to show that it is possible to replace a hardware core with another one by manipulating only the bit streams. We perform operations on the bit streams only because we want to avoid lengthy synthesis cycles. To achieve this result we take the following steps:

- We propose a design flow and implementation to replace one hardware module for another

- We limit our work to the resources and wiring which are of most interest
- Simple tools are made for manipulating the bitstream with the information obtained
- Tools for isolation of hardware cores, placement and routing are developed
- A framework is build to automate the removal, placement and routing of various hardware cores.

Our method has several advantages over existing methods:

- It no longer reserves module specific space on FPGA.
- Does not reserve routing paths.
- Does not make prerequisites for the size or shape of the module.
- Does not use bus macros, thereby using less resources.
- Can do free placement as long as the module structure matches the underlying FPGA resources.
- Can route through existing structures.
- Can place modules on top of existing routing if there is no conflicting use of wires.

The advantage of our solution is we no longer reserve a specific area for the modules, nor do we make use of reserved routing paths. This means we can truly arbitrarily place and connect any module to any other module. We do not make use of bus macros, nor do we place any restriction on the size and shape of the modules, although they do have to fit on to the device without causing conflicts with the existing configuration. As long as the router can generate a pathway, routing can go through existing structures. As long as existing pathways do not conflict with the routing within a new module, modules can be placed on top of existing routing. We can do 2d placement, enabling more efficient area usage and routing densities are only limited by the available resources on the FPGA. Although the developed techniques are intended for on-line bitstream generation, the Achilles heel is in the computation required for doing routing.

1.5 CE Research in Reconfigurable Computing

Without much doubt, the utilization of reconfigurable hardware adds flexibility and performance, mainly due to the exploitation of parallelism in hardware. This has been proven as the reconfigurable hardware can outperform general-purpose computing for a wide range of algorithms and in some cases by a large margin [29]. It is therefore desirable to exploit this advantage to accelerate general-purpose computing. Currently, this is done in an ad-hoc fashion, where specifically designed hardware cores are implemented for each application. This methodology is very much dependent on the platform it targets and is impractical for most software developers. The computer engineering (CE) laboratory is working on various projects to change this.

1.5.1 Delft Workbench

The Delft workbench project [9] strives to develop a semi-automatic platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. We are developing an integrated environment to make it possible for a 'software' developer to implement applications on a reconfigurable platform. The process of designing an application not only involves producing a functional implementation, but also entails the identification of the components which will have the most potential to improve overall execution time, by translating them into hardware. The workbench helps to identify these components and also helps the designer with the translation process, in which a sequential algorithm has to be rewritten to make use of the parallelism (hArtes project). This is inherent to a hardware implementation. It is possible to just use library components to accelerate the application with hardware. In achieving the best results in optimizing an application it is important that the developer has knowledge of hardware design, however the group is developing a compiler that will translate C into a hardware description (MORPHEUS project). This will offer the designer the ability to find the best trade-off between development time and application acceleration, which is done by choosing one of these three methods for the implementation. Because of the close coupling between the general-purpose processor and the FPGA, there are delays caused by reconfiguring the FPGA. A retargetable compiler will have to take care of this issue (RCOSY project), by issuing the reconfiguration process as soon as possible. The target platform of the Delft workbench is the Molen reconfigurable processor.

1.5.2 Molen

The Molen architecture [28] is a combination of a general-purpose processor (GPP) and a reconfigurable processor (RP). This is an architecture in which there is a close relation between software running on the general-purpose processor, and the hardware implementations in the FPGA which are meant to accelerate the application. It extends the instruction set of the GPP with four to seven instructions to add reconfigurable functionality. The instructions encompass the configuration of the RP, the execution of the configured functions, the exchange of data between the GPP and the RP and synchronization between the two.

This research focuses on the replacement of one hardware configuration by another on an FPGA. The platform we focus on is the Virtex-II Pro, which is a common FPGA that has (among other features) two PowerPC processors on chip. For this platform we have an implementation of the MOLEN reconfigurable microcoded processor. We can utilize our results for an implementation of the 'set' instruction, which initializes the RP for a new function.

1.6 Overview

This thesis is organized as follows. Chapter 2 describes what runtime reconfiguration is, explains our methods and presents related work. In Chapter 3, we discuss the assumptions made and the methodology followed to arrive at our implementation and the tools we developed for this. Chapter 4, we give examples to demonstrate how our methodology evolved and to show that it works. Finally, in Chapter 5 we draw some conclusions and give recommendations on how to proceed from here.

Background 2

In this section we will present related work by others, and compare them with our own. Because the only practical design flow up for partial reconfiguration until now has been XAPP290, and we have developed a new way to do this, we will mainly focus on different forms of connectivity between hardware implementations on an FPGA and the way this can be improved by our methods. The related work will show:

- Connectivity can be split up into direct circuit switched, logically circuit switched and network-on-chip.
- Placement can be 1 Dimensional or 2 Dimensional.
- The area for the placed hardware is fixed, slot based, or has significant restrictions.
- The module sizes are often predetermined and modifications to these require changes to the floor planning at different levels in the design.
- All forms of partial reconfiguration are using bus macros to affix routing to specific locations.
- All implementations use the Xilinx tools to generate the bitstreams and require resynthesizing a project at compile time for each modification.

Although many authors are focused on implementing networks on chip, they need (partial) reconfiguration to produce and modify these. Our methods will present a more dynamic approach to partial reconfiguration. We propose a method that:

- Will perform partial reconfiguration in a way that can be applicable to any form of connectivity and hardware implementation.
- It should do away with bus macros as they are a waste of resources.
- It should pose the minimum amount of restrictions on modules size, shape and the position it can be located at.
- Placement of hardware should be 2 dimensional.
- It should generate partial reconfiguration bitstreams without the use of the Xilinx tools, to enable migration of device reconfiguration to runtime instead of compile time.
- The partial reconfiguration has to be dynamic in order to hide reconfiguration delay.

- The method should be implementable on existing reconfigurable platforms, like the Molen platform.
- The implementation has to be incorporated into a design flows for application on future projects.

In this chapter we will also provide an overview of the normal design flow for partial reconfiguration, we will propose a modified version for our own implementation and we provide an overview of the known information of the targeted FPGA device.

2.1 Related Work

The result of the work done for this thesis is a method for placing and connecting modules onto a Virtex-II Pro. If needed old modules are removed to replace one hardware implementation for another. This is done dynamically using partial reconfiguration to hide reconfiguration delay. The normal method for placement is 1 dimensional, but there has been some research on 2 dimensional placement for this family of devices. 1 Dimensional placement, using vertical slots, is the most practical implementation, but it wastes a lot of space. 2 Dimensional placement, as we propose to do, is superior in efficient use of area [27]. The method chosen in this thesis for connecting the modules is a direct connection, also know as direct circuit-switched. Other methods are logical circuit-switched and network on chip. Network-on-chips are packet switched networks. Direct circuit switched networks have the least communication delay, as they are short unbuffered connections, but they can be considered static networks. For pathways that are not static, but do not change frequently, logical circuit switched networks are a good alternative to packet switched networks. In logic circuits-switched networks, pathways are negotiated before use, setting various switch nodes to form a direct link between modules. Networks on chip (NoC) are packet switched networks, which buffer the packets between each router, and can therefore be clocked higher as distances between routers are shorter compared to direct connections. They do however take up a lot of resources due to their complexity. In the next section we present work by others related to these subjects.

2.1.1 1 Dimensional Placement, Mostly circuit switched

In Bieser, *et al* [3] the modules are fixed in dimensions and stretch the height of the device. The modules are proposed to be IP-Cores which have been tested thoroughly and can be used as standard building blocks. The routing problem is overcome by using a shared bus to which the modules attach. This bus makes use of the tri-state drivers that are in the Virtex-II. This means that only one module at a time can make use of the bus. The designer can merge the module bitstream with the configuration bitstream using an application based on JBITS [31]. It is possible to create user defined IPs.

JBITS development has been stopped, there is no support for newer device families like the Virtex-II Pro. Our implementation could isolate the various modules created into separate snippets of bitstreams. As there is a fixed bus to connect to, no routing is necessary. We can produce the bitstreams that dynamically remove and place these modules at runtime. Taking device restrictions into account it can be possible to modify this implementation to do 2 dimensional placement. Although we refrain from exploring the use of the tri-state bus structures, it may be possible to extend our methods to make use of these and reroute them dynamically. Other network topologies are advisable as communication can only be with one module at a time.

Bobda, *et al* [5] have proposed two methods for exchanging signals between modules, also using partial reconfiguration. One is based on 1 dimensional vertical-slot shaped reconfigurable modules which are connected using a reconfigurable multiple bus (RMB) , similar to the one suggested by ElGindy [10] . This RMB uses logical circuit-switched routing in the form of a standard switch matrix for each module, with a controller added to perform worm-hole routing. The established link can transfer data from source to destination each clock cycle.

This might be the best trade-off between design complexity and the computationally intensive approach we take with respect to routing the signals, though direct circuit-switched networks have a little less delay. The RMB could be substituted by dynamically reconfigured direct circuit switched networks, using our implementation. This can only be successful if connections do not need frequent reconfiguration. As with Bieser, *et al*, our methods could perform device reconfiguration for this implementation because this is also a slot-based implementation, though using a different type of bus. The bus in question can enable multiple modules to communicate to each other. Extending our methods, the interfaces can be dynamically added or removed and could be modified to do 2 dimensional placement. The second approach is discussed later on.

Karsteva, *et al* [20] propose how to relocate part of the bitstream for 2d placement, although this information is easily deduced from the user guide. They have only implemented a 1 dimensional method of placement and have information on the execution time for merging bitstreams and producing the partial bitstream for reconfiguring the device functionality.

This is interesting, as we do not focus on execution time for this research, but it is a major factor limiting practical implementation of our method. The use of perl-scripts and the intermediate steps we take makes it impractical to compare our work with this, as our approach is obviously much slower than an C implementation.

2.1.2 2D Placement, Mostly Network-on-Chip

Sedcole, *et al* [25] proposed a method for reconfiguring hardware cores, in the form of modules, that is more flexible than described in the Xilinx application notes [33]. They provided a way to place hardware cores above each other, whereas the Xilinx method dictates that modules stretch the entire height of the device. The positions and size for these hardware cores have been predetermined. The issue with static routes passing through the modules, were resolved by reserving the long lines as pass through regions in the modules. Signals are connected to static routing through bus macros. The operations necessary are performed at a bitstream level with the use of stored configuration data of the modules.

As the bus and connectivity for the modules is static, no routing has to be done. Due to the fixed module size however it is inefficient with the usage of the available space. By reserving signals for pass-through routing this reduces the effective number of available wires within a module. It could therefore result in larger modules, operating with longer wires which have more delay. Our methods will do 2 dimensional placement without reserving space for it, nor will there be any restrictions on the shape and size of the modules as long as they fit.

Hübner, *et al* [17] implemented online routing using regular routing structures which stretch vertically through the device. A module can be attached to the structure at any location. In this manner, they provide arbitrary 2D placement of modules of any size, as long as they do not conflict with the routing structures, vertically interrupting the FPGA configuration regularly. The routing structures are lookup table based and have to be reconfigured at the the position it attaches to the module. The number of signals that pass through the structure is limited, due to the use of lookup tables. The operations are performed by a C program running on the PowerPC or Microblaze processor using stored configuration data for the routing structures and modules.

The use of lookup tables adds delay to the routing as the signals pass through quite a few lookup tables, which is not the case for our directly connecting method. The routing pathways are configured statically and in advance, hence the length of the pathways is longer. Its implementation appears to have slot-based modules which are slaves to a single master controlling this bus structure, indicating that there is no communication between modules. This is different for their network-on-chip based approach, which is more complex. Their lookup table based method could be considered a reconfigurable logical circuit switched network. The modules have restrictions as they may not conflict with the routing structures. This implementation comes closest to the goals we want to achieve. It does not provide the highest degree of freedom in using the device resources and still poses restrictions on the modules and has restricted communication capabilities.

Möller, *et al* [23] use a modified Hermes network on chip, named Artemis and bus macro like structures to provide for dynamic reconfiguration of the cores attached to the router. The routers themselves are part of a fixed network. The reconfiguration support is added with custom macros which can block signals going through while the modules are being reconfigured. This stops possible transients caused by the reconfiguration process from entering the network. They resemble the standard lookup table bus macros used in the Xilinx method. The method for positioning the modules is stated to be slow, although it was implemented in C. As we are not focusing on reducing the time to calculate and reconfigure the device our implementation will not be faster. The modules have the appearance to be placed in two dimensions, but it may be a clever way to hide 1 dimensional slot-based placement. Our method can reconfigure both modules and the network implementation freely, although the suppression of transients caused by reconfiguration have to be addressed by the designer, incorporating an extra and-gate to block signals during the reconfiguration phase.

For the second part of their Journal paper, Bobda, *et al* [5] assume an FPGA that is capable of 2d placement, and have build a network on chip that can cope with routing packets, even though the structure of the network is irregular. They avoid creating obstacles by surrounding each module with a network ring. In this way packets can always get around modules, no part of the network gets cut off by placing a module at a certain location. They show that their routing algorithm is probably deadlock-free. They only show how this would work for a static configurations, but are working on an implementation that can handle changing the network configurations. Their use of JBITS means they can only implement this on older FPGAs. Because their implementation is mostly based on placing modules within an existing routing mesh, the routing problem has been transfered to the packet routers. As the interfaces are standardized and connected automatically by placing the modules at the correct position. Using a more dynamic network, it might be possible to reduce the number of routers using our methods, as the networks surrounding the modules can be adapted to the existing structures. This would greatly improve on the available resources for the modules themselves, because the routers takes up a lot of resources.

Hilton and Nelson [15] describe a circuit switched network called a programmable network on chip (PNoC) without going into the specifics of the reconfiguration process, though they have an implementation for the XC2VP30. They propose logical circuit switched routers that can handle dynamically changing networks by performing updates on the routing tables when a change in network occurs. They supply various topologies for the network as an example, which suggest they do 2 dimensional placement, but as they focus on the network and not the reconfiguration they only specify the number of slices and block-rams their router takes up. Because not much information is given on their implementation except for a desire to apply their network to dynamic module replacement as it was designed for this purpose. Their current demonstration application is a static network. Our methods would be well suited as it can deal with varying shapes and sizes of modules, coping with multiple consecutive substitutions without the hindrance of a predetermined floor plan.

2.2 Xilinx Virtex-II Pro

Although reconfigurable computing is not a new field, only in the last fifteen years have we had the opportunity to explore its potential. The technology is showing promise, but there are still quite a few hurtles to take. One of the problems is caused by the time it takes to set up the reconfigurable processor (RP) to do a new task. Although a hardware implementation can be much faster than the equivalent in software, this advantage is counteracted by time it takes to reconfigure the device, especially compared to the cycles that the general purpose processor has available in the same interval. Runtime partial reconfiguration is important for reducing and hiding the delay of reconfiguring for a new task before it can be used. First we have to look at the reconfiguration process, taking into account the device we have chosen.

Because reconfiguration bitstream are specific to a device family, we prefer to choose an accessible common FPGA to work on. Therefore, we chose to target the Xilinx Virtex-II Pro family. It is in common use, can do partial reconfiguration and the Xilinx university program board [18] has an attractive price tag. It has an XC2VP30 as its main device, offering 2 power PC cores 30k+ logic cells 136 multipliers and 428kBit of RAM. Most importantly, there is an implementation of the Molen processor for it.

Figure 2.1 depicts the arrangement of resources in the XC2VP7, scaled to the number of bits they use in the bitstream. The most common resource is the complex logic block (CLB). These logic blocks contain a portion of the hardware description and use

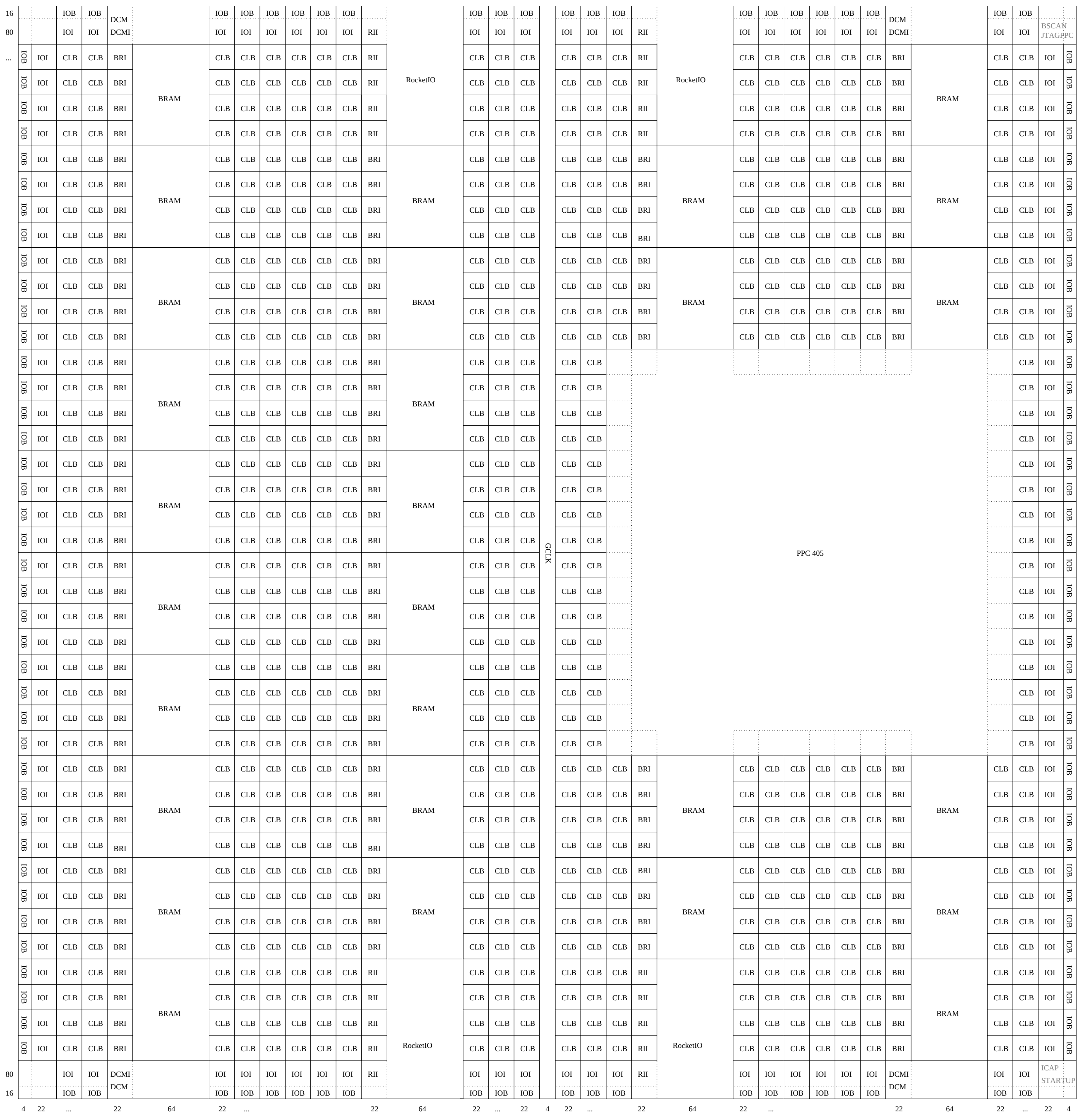

Figure 2.1: XC2VP7 structures scaled to size in bit-stream

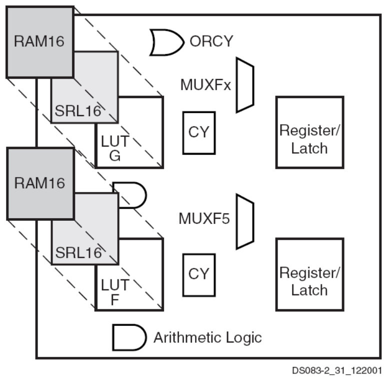

Figure 2.2: Virtex-II Pro Slice configuration

a switch matrix for routing signals to other resources. Resources like BlockRAMs and RocketIO interfaces are distributed throughout the matrix column-wise at regular intervals. These are connected by BlockRAM interconnections (BRI) and RocketIO interconnections (RII), which are very similar to the switch matrix of the CLB. BlockRAM interconnections also provide connectivity to the multipliers which do not take any reconfiguration data and so do not show up in the bitstream. Around the edges of the device the connection to the outside world is made through IO interconnections (IOI). It can be seen that there is regularity in the distribution of the resources. The most prominent feature is the PowerPC core. Although it seems to take a lot of reconfiguration data, it is highly unlikely that it uses a large portion of this. This indicates the frames are meant to fit the largest reconfiguration-data structures on the device, and that a lot of information is discarded for the sake of uniformity of the bit-stream. For now, we will limit ourselves to the use of CLBs only, although we do take the problems caused by the heterogeneous character of the device into account.

Complex logic blocks (CLBs) are composed out of 4 slices of which a graphic representation is depicted in Figure 2.2 as taken from datasheet [35]. Each slice contains 2 lookup tables, 2 flip-flops, multiplexors and some logic gates. The lookup tables take 4 inputs and produce 1 result, hence taking 16-bits of configuration data. It can represent logic, slice distributed RAM or a shift register. The multiplexors are used to set the slice up to perform specific functions. They can be used to produce larger lookup tables build up out of multiple standard 4-input version. They can also, for instance, be configured to make use of fast carry logic, enabling fast arithmetic functions using incorporated logic gates specifically for these types of functions. As a simple construction element of a larger system, the CLB can perform a great number of tasks.

Figure 2.3 depicts a complete CLB as shown by the *fpga_editor* tool of Xilinx. The 4 smaller squares to the right represent the slices, the large square to the left represents the switch matrix. Most of the wiring for the slices go through the switch matrix, with the exception of the wires forming a carry chain, which are directly liked to neighboring CLB slices. The switches themselves are not fully connected. A single node on the bottom of the switch matrix has been selected. The yellow connections represents possible signal

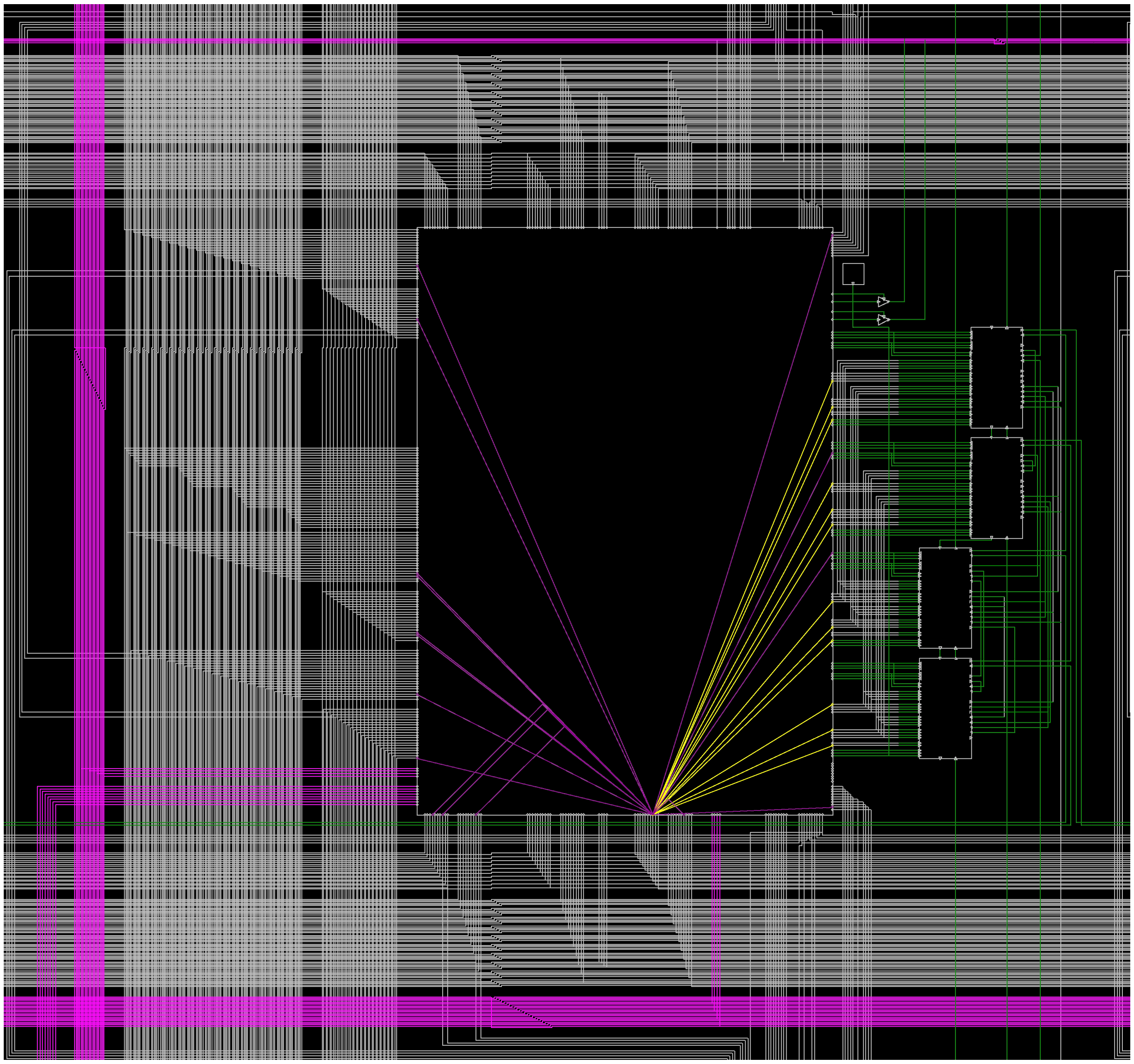

Figure 2.3: CLB as seen in fpga_editor

routes leaving the wire. The purple connections represent possible signal routes entering the node. As can be seen, the switch can not connect every node to another, this would become too complex a structure for the large number of nodes that are available.

Figure 2.4 depicts the different types of wires connecting the various switch matrices. According to the datasheet, for each switch matrix there are 16 wires connecting to

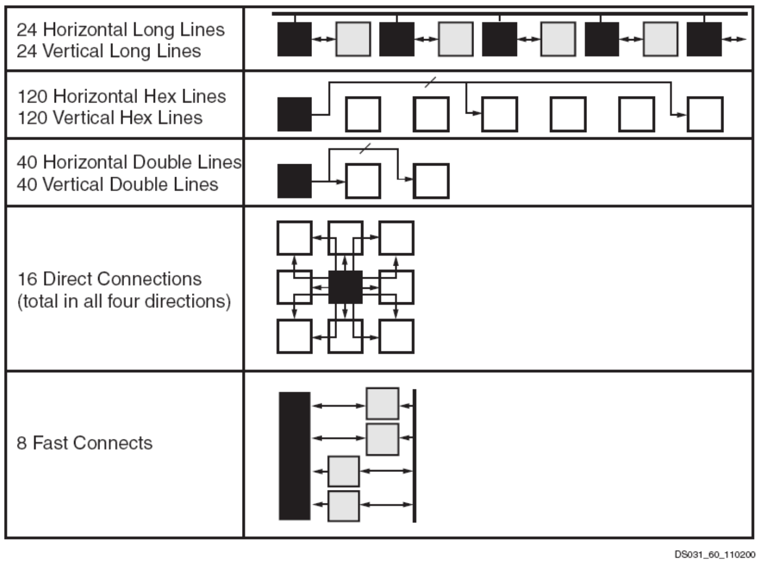

Figure 2.4: Virtex-II Pro wires as taken from the datasheet

the neighboring resources, 40 wires that connect to the resources which are spaced two and four positions apart and 120 wires that connect to resources spaced six and twelve positions apart. 24 The wires run in horizontal and 24 in vertical directions. These numbers are somewhat polished as they represent the maximum number of wires passing through the CLB, not the number of wires connected to the switch box. In reality, if we only count outgoing signal pathways, they would come to 16 neighboring wires, 40 'double' wires, 40 'hex'-wires, and only 4 vertical and 4 horizontal long lines. This still makes the number of wires available quite formidable.

The long lines span the width or height of the entire device. There are also special tristate wires. Most of the wires have only a single driver ensuring signals only travel in one direction, this however is not true for the tristate bus, nor for the long lines. These can be driven from any point, possibly causing damage if a wire is driven from multiple locations. On a device-wide level, there are special wires for distributing clock and global signals throughout the entire device. These run out from the center of the device to the edges, making sure that the delay to each point is approximately the same as is depicted in Figure 2.5 , taken from the user-guide. This is to prevent problems with clock skew and to have specific clock regions.

2.3 Xilinx Design Flow for static configurations.

Xilinx provides an integrated development environment (IDE) for programming their devices in various descriptions, including VHDL and Verilog, called ISE. As taken from the ISE help file, Figure 2.6 depicts the normal design flow of the ISE package. This does not include the design flow for partial reconfiguration as explained in application note XAPP290 [33].

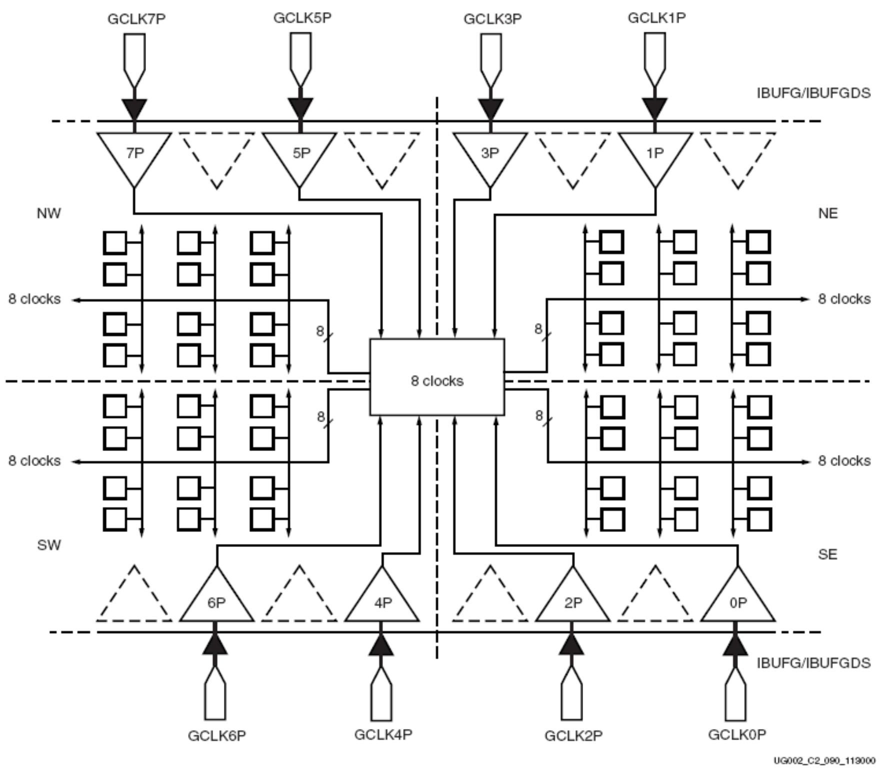

Figure 2.5: Virtex-II Pro clock distribution as taken from the user-guide

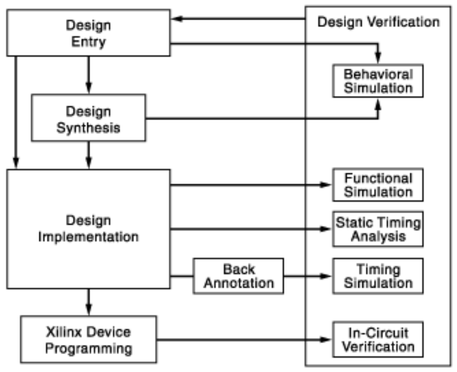

Figure 2.6: ISE design flow

During the design entry phase, the hardware is described in a high level hardware description language like VHDL or Verilog. It is also possible to design the hardware using a schematic representation, but it is not common. A user constraints file (.ucf) specifies parameters for the entire design process. It allows the user to provide the ISE package with information it can not derive from the high level description. The user can specify constraints on placement, timing and synthesis of the hardware design. Placement constraints can for instance determine which pins are connected to which signals. Area constraints determine the location to which the hardware implementation is bound. The user can also specify timing constraints to ensure the a critical path meets certain demands for signal delays. Synthesis constraints instruct the synthesis tools to perform specific operations, usually to improve timing or reduce area. If too little information is provided in the constraints file, the software package will try to generate a optimal solution without it. For example, not providing placement constraints on which signals should be connected to which pins, results in the tools assigning a pin by itself. In case the design is only simulated, this may not be a problem. When implementing the hardware description on a device it becomes essential to specify this information.

The *xst* tool parses the HDL description. It does synthesizes of the design and does logic optimizations. The hardware descriptions are synthesized to the register transfer level (RTL) representation of the design for behavioral simulation. It can also be compiled to a gate level description. This representation is in terms of generic symbols, such as adders, multipliers, counters, AND gates, and OR gates, etc., which can be used for a functional simulation. *xst*, produces a .ngc-file, which contains the design implementation mapped to the targeted technology. This is a representation in terms of logic elements optimized to the target architecture, for example, in terms of LUTs, carry logic, I/O buffers, and other technology-specific components. For the Virtex-II Pro for instance, this means the targeted technology uses 4-input lookup tables, dual ported BlockRAMs and 18x18 bit multipliers. The 'translation' step of the design implementation acts as a linker would for a software compiler. The *ngdbuild* program takes the netlists, design constraints and .ngc-files containing the designs, and outputs a .ngd design database. The mapping process, performed by the *map* program, fits this to a specific device in the family, for instance the XC2VP30. It describes the design in the form of the specific resources of the targeted device, for instance CLBs or IOBs. The Place and Route (*par*) process takes a mapped .ncd-file, places and routes the design, and produces an .ncd-file that is used as input for bitstream generation or post-synthesis simulation. It assigns the location of the various slices the design has been translated to and performs the routing to connect them up. For device programming, the finished design database (.ncd-file) is translated into a configuration bitstream by the *bitgen* program. This bitstream can be offered to the device in various ways, for instance through the JTAG interface using the *impact* program, or it can be programmed into a configuration ROM which is loaded automatically after the device is powered up.

2.4 Partial Reconfiguration Using the XAPP290 Design Flow

XAPP290 describes two methods for doing dynamic partial reconfiguration. The modular method, as depicted in Figure 2.7 divides the FPGA up into portions for specific functions. These portions come with considerable restrictions. They must span the full height of the device. All resources within the module are part of the module, this means no wires passing through may be present. Pins which fall inside the module can only be used by that specific module and unused resources like BlockRAMs or multipliers are

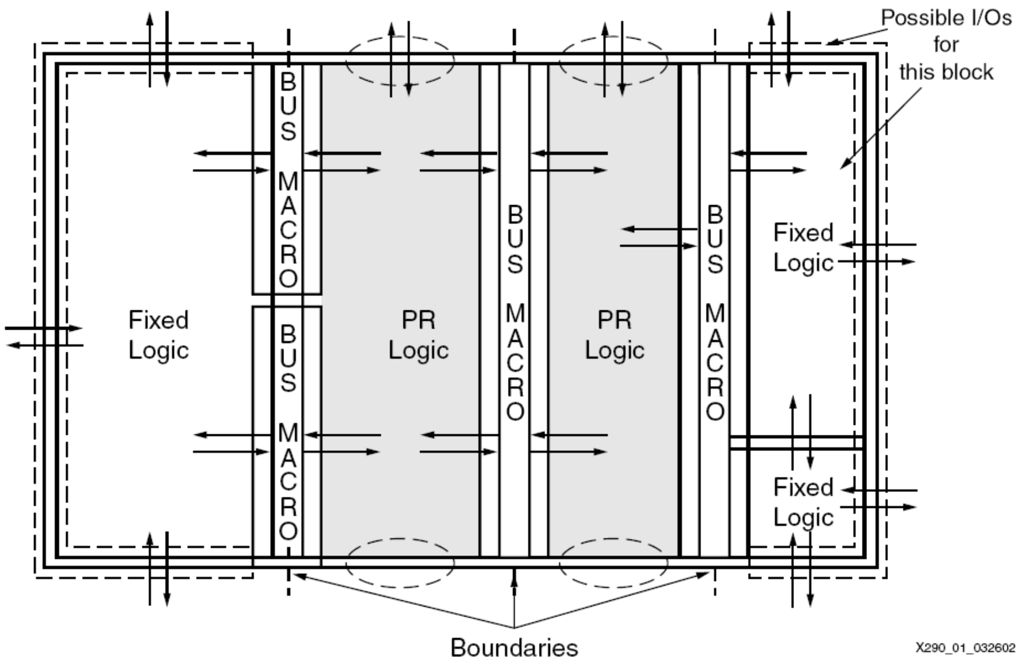

Figure 2.7: XAPP290 vertically stretching modules

unavailable for other modules. The slot size of a module is fixed, this means that only modules with a similar size and shape can be placed inside that space. Communication between modules must be done through bus macros situated at the edges of the module. These are needed to have a fixed position to hook signal routing to, as there is no direct way to influence the routing paths the Xilinx tools generate. The design flow follows the normal Xilinx design flow for each module, taking some limitations into account. These modules are combined in a top-level design in which they are assigned a specific area and information is provided on the connectivity between modules. Each module is synthesized separately and can be simulated individually. For each combination of modules the top-level design is synthesized and tested, using the modules created in the first step. Of these designs a partial bitstream is created that reconfigures the device for the next function. For each possible combination of modules, a new design has to be synthesized. For each transition between configuration a specific partial bitstream has to be synthesized in advance. If a module does not fit, the top-level floor plan has to be altered and all modules which fit the same spot or are adjacent have to be resynthesized to the new dimensions. The difference based method, described as the second method in the application note, is proposed for small changes to the design. The design is opened in the *fpga_editor* and the manipulations are done manually. These manipulations usually encompass changing the LUT contents, loading different content into the BlockRAMs or changing the I/O standards of the pins. This method is not very well suited to do big changes to the routing. The application note urges users to use the modular based approach instead. This makes sense, because changing the routing may lead to damage to the device when performing active reconfiguration. This approach would also be too labor intensive to implement on a large scale.

2.5 Proposed Method for Partial Reconfiguration

We propose to develop a more idealized method of partial reconfiguration. It will have to be able to handle modules of varying shapes and sizes which will be combined at runtime. In this way, as long as there is binary compatibility between devices within the same family there is no need to resynthesize a module for each device. It also means there is no precompiled bitstream to do the transitions of the various combinations, they are generated at runtime. It would become possible for the system to 'load' a module instead of setting up the reconfigurable device in a predetermined way.

2.5.1 Proposed Design flow, Building the Modules

The design flow for the modules is similar to the module based reconfiguration method in XAPP290, but with much less restrictions. We propose to combine the modules at runtime, so the configurations are not synthesized in advance. This allows us to have no restrictions on the specific size of a module, as long as it can fit the design when it needs to be configured. The modules no longer need to span the entire height of the device, as we aim to do 2D placement. For this we do need to do routing at runtime as the specific location and routing paths can not be predetermined. The design flow would not need to be specifically module-based, as long as hardware cores have area constraints, though it would be more natural for developers to use the module based approach. There is no need to add bus macros as we do not need the routing to be at the edges, because we have control over the routing. The modules can be simulated and tested using regular methods. More over, changing the floor plan has less influence on the modules already synthesized, as they can be repositioned at design time. In the design flow described in XAPP290 this would mean a complete rebuild of all the modules and top-level designs. The information needed to extract the hardware core can be taken from the Xilinx files. The .ucf-file contains area constraints which can be used to determine where the hardware core is placed on the device. The .ncd-file can be translated into a bitstream but the content of the database can also be dumped in a more readable eldif-format by the *xdl* tool. From the output of this tool we can determine the names of the routed nets, as used in the high-level language. For this thesis we do not retrieve the names of the nets, nor do we extract information form the constraints files. The netlist combining the various cores can be determined from the top level design.

2.5.2 Proposed Approach to Runtime Partial Reconfiguration

Replacing a hardware core for another one during runtime is depicted in Figure 2.8. As a minimal set for the reconfiguration we require a description of the current configuration, an isolated configuration file of the new core, as well as a netlist for the signals that have to be routed between the new core and the existing design. For reconfiguration we first need to extract the current configuration of the FPGA from the device, or use a copy of

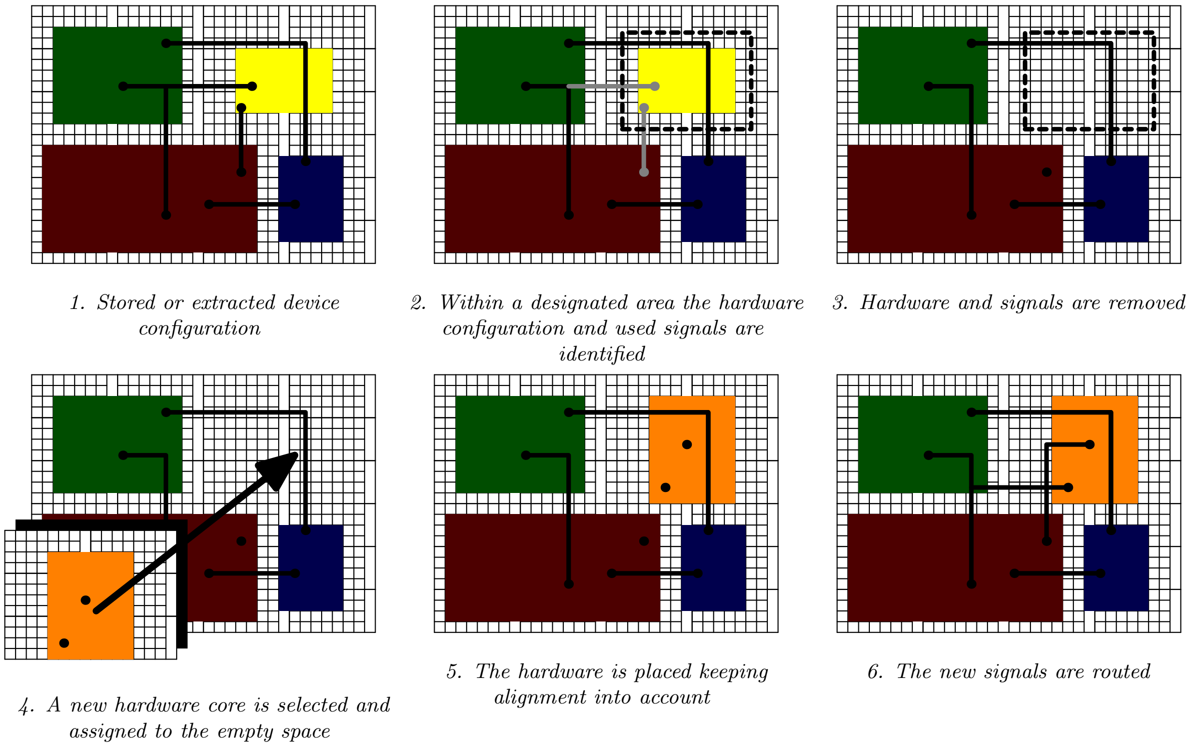
1. Stored or extracted device configuration
2. Within a designated area the hardware configuration and used signals are identified
3. Hardware and signals are removed
4. A new hardware core is selected and assigned to the empty space
5. The hardware is placed keeping alignment into account
6. The new signals are routed

Figure 2.8: Exchanging one hardware core for another

the file it was configured with. We have to identify the hardware core, which has to be removed from the bitstream. This can be done by specifying an area in which it is known to reside. It is also possible to search for a specific core in the bitstream by comparing it with the stored version. For this thesis we assume it is known where the core resides and we have the binary file for it.

In the designated area we isolate the hardware core and we can trace internal wires and external signals. As we only plan to make use of the lines that have a single driver, the communication signals between the modules can only be incoming or outgoing. We have to take into account that incoming signals can be in use on multiple locations of the core. Outgoing signals can be eliminated, taking into account the possible internal use of the same signal. The bits that have been found to be a part of the hardware core or its internal wiring can be subtracted from the bitstream. The extracted list of communication signals is used to unroute the connectivity between the remaining configuration and the module which has been removed. To make sure there are no damaging effects when directly switching from one core to another, it is advisable to first use this bitstream to remove the core before configuring the new one.

The bitstream for the new core has been previously isolated in a separate bit file. When placing the new core it is important that it is aligned properly. The matrix of CLBs is interrupted by a column of BlockRAMs and multipliers at an interval of 6 columns.

These have a height equivalent of 4 CLBs. If BlockRAM resources are used by the core it is important that positioning it in the vertical direction is done in steps of 4. Most of the time horizontal displacement can only done with multiples of 7 because of the column of BlockRAMs and multipliers. The last step involves routing the signals to and from the core. These do not have to be in the same position as the previous core, nor do they have to connect to the same terminals of the hardware remaining in use. With the aid of a netlist the routing is automatically generated and added to the bitstream. The resulting bitstream is the complete description of the new device configuration. This is compared with the original configuration and the intermediate configuration where the old core is removed. Only the frames that are different will be reconfigured, effectively only removing the old core and reconfiguring the new.

2.5.3 Advantages and Disadvantages

The advantages of this technique are mainly due to the routing. An existing configuration can be reused, and new routes can pass through existing structures on the FPGA. Because there is no use of bus macros or other techniques for affixing the routing to a predefined position, the delay of the signals through the wires can be less because they are expected to be shorter. We can arbitrarily place modules anywhere on the device as we are not dependent on specific routing structures, though we are dependent on the heterogeneous distribution of the device resources. The density of the routing is also higher compared to bus macros or other solutions as they make use of the lookup tables and can only route 8 signals for each CLB. The biggest disadvantage also comes from the routing process, as it is very computationally intensive to do. It is, therefore, less practical to do in a real world application. It is, however, a good starting-point from which to develop techniques which accelerate this process. For instance, free routing has to be used, only if the routing problem can not be solved by general routing structures. These general routing structures can be made with common bus structure made out of regular ‘modules’, consisting only of regular wiring, which can be placed after each other. By gaining insight in the actual wiring structure, the bits and frames involved, and the processes that are needed to generate the pathways, we have the possibility to develop better methods of producing efficient standard building blocks. We can develop techniques to manipulate modules by rerouting conflicting pathways before placing them or to rearrange a module to better fit the design.

Another problem has to do with runtime reconfiguration. The runtime combined modules have not been simulated and tested as a complete system, and therefore timing problems may occur which could have been detected and resolved by applying some test vectors. All runtime reconfiguration solutions suffer from this problem, but just as the approach of XAPP290 can simulate the entire system, the configurations could actually be simulated in the ISE package as might be able to recreate all the bit manipulations using the *fpga_editor* or the *xdl* program. Analysis after a problem has occurred can be done, however it is more important that such a timing problem never happen. Other problems are general to applying 2d-placement to the Virtex-II Pro FPGA. As we configure frames, we actually overwrite configuration data over the entire height of the device. As long as this data is the same as was already there, this will not be an issue because in this case is stated to be transient free in the datasheet [35]. In BlockRAMs, and in some cases the lookup tables as well, configuration data is actually used as a storage element. When writing a frame to configure one part of the device, all configuration data in that frame is overwritten. When the configuration bits are used as storage elements, they are replaced while being in use. This is an unwanted side effect which cannot be avoided with this device. The Virtex 4 family addresses this issue by dividing the frames up into regions. With the Virtex-II Pro, adding or removing routing to the configuration is not a problem, as the storage elements do not share frames with the routing configuration data.

2.5.4 Conclusion

All current methods rely on fixed interfaces between the modules and the routing paths, in the form of a bus macro or lookup table based primitives. We propose a method which does true direct switched routing of the signals. This provides more flexibility, as there are no predetermined routing paths or dedicated wires. It makes it possible to route through existing structures even though they do not have specifically reserved space for this. The lack of using bus macros or LUTs saves resources and reduces delays. Furthermore, the number of signals that can be connected is limited only by the available resources of the FPGA itself. We do 2d placement of arbitrarily sized and shaped modules. We do not reserve specific areas for the modules. As long as there are no conflicts with existing structures and configurations, a module can be placed and connected anywhere on chip. As all operations are done on a bitstream level, not using 3rd party tools, they can be implemented on the PowerPC or Microblaze. We make use of stored configuration data for the module and have a database of configuration data for the switch settings, used for routing the signals. Wires crossing boundaries in to other modules are not a problem, as long as they do not directly conflict with the wiring of the other module. Transients cased by reconfiguration have to be resolved in the modules themselves. Adding a logic gate, disabling communication during reconfiguration can be enough.

3 Work

In this chapter we will discuss the technical background of the work that has been done. Our goal is to show that it is possible to replace a hardware core with another one by manipulating only the bitstreams. We perform operations on the bitstreams only because we want to avoid lengthy synthesis cycles. To achieve this result we take the following steps:

- A device to use for our approach has already been selected in the previous chapter
- An overview of the structure and composition of it's resources has been presented in the previous chapter
- We will need to obtain a more detailed description of all the internal structures
- We limit our work to the resources and wiring which are essential to achieve our goal
- The bitstream is decomposed for routing and the resources of interest
- Simple tools are made for manipulating the bitstream with the information obtained
- Tools for isolation of hardware cores, placement and routing are developed
- A framework is build to automate the removal, placement and routing of various hardware cores.

3.1 Resources in the FPGA

As there are many internal structures in an FPGA, limits have to be set to only work on the resources which are the most important. It would take too much time to completely decompose the configuration options the entire device. For now, research has been limited to the use of CLBs only. The method can be expanded to take the use of the other resources into account. The slice configuration data is taken as-is. We use the normal ISE tools to do synthesis. Most effort has been put into deconstructing the wiring. Even so, some wiring structures are ignored. There are special tri-state bus structures, which could be used to connect multiple modules to the same bus. There are also long lines, which span the width or height of the entire device. To prevent accidental damage to the device we will not use wires that can be driven from multiple locations. This means we exclude the utilization of long lines and tri-state lines.

3.2 Operating System and Programming Language

The default operating system on CE workstations is Linux. It is also the OS of choice for the particular MSC student doing this research. As Xilinx offers a Linux version of its software there is no need for using Microsoft products. As for the programming-language of choice, due to the different file formats used by the Xilinx software and the various descriptions that have to be developed, the choice was made to use Perl. It also provides a good excuse to have a taste of a different language.

3.3 The XUP Evaluation Board

The Molen polymorphic processor has been implemented on Xilinx ML310 and XUP evaluation boards. The specific FPGA device on these boards is the XC2VP30. At the CE department we have several Xilinx University Program (XUP) board made by Digilent, Inc. All experimentation has been done on this board. It features various interfaces and expansion ports, including Ethernet, VGA(XSGA), audio and general purpose I/O. Most of these features have not been used in this research.

Programming is done though the on-board USB Jtag interface, which is probably equal to the ds300 programming cable. As we prefer to run the software on Linux, and Xilinx is inconsistent with their support, there have been difficulties getting this interface to work. The drivers supplied by Xilinx have been developed for the 2.4 kernel. Most of the recent distributions use the 2.6 kernel. Even though Xilinx provides a download [32] for this new kernel it contains an older version of windrvr which does not compile with newer 2.6 kernels. A newer version can be obtained from the Jungo [19] web-site. Only the source in the /redist directory needs to be compiled and installed. The xpc4drvr driver for the parallel Xilinx JTAG is also necessary. Recently a new package has been made available by Xilinx containing newer versions of the xpc4drvr and windrvr6 (v8.01) software. This new package also contains versions for 64-bit systems. None the less this package does not install correctly on the departments Fedora core 5 systems and therefore the preferred solution is compiling all the packages ourselves. There is no suitable solution yet for installing the driver for 64-bit systems.

The on-board programming cable uses a FX-2 (EZ-USB) 8051 micro controller made by Cypress [8]. When plugged into a USB port the micro controller receives its firmware from the USB driver. Linux uses *hotplug* services to facilitate automatic driver loading for USB devices. Newer distributions use *udev* instead of *hotplug*. We have developed scripts to facilitate the use of udev as well, because newer Fedora distributions lack hotplug support. For loading the firmware we needed to install the *fxload* [7] package. The Xilinx software installs *hotplug* scripts and firmware, but the ones that came with early versions of the 7.1 software needed some tweaking with device id's. This has also been addressed by Xilinx with the newer patches. There might be some issues with user rights to access the file handle to the device in /dev. We had to tweak the udev configuration to fix this. This did not solve all of our problems with this interface. The windows driver seemed to function without any problems. It was first thought that there was a difference between the firmware loaded into the FX-2 micro controller. After using a USB sniffer and comparing the packets under Linux and Windows it was determined that they where identical. By accident it was noticed that the windows driver releases the USB device twice. Normally, when the USB device is plugged in for the first time the firmware is loaded in to the micro controller and the USB device is released. The microcontroller boots the firmware and reestablishes the USB connection. Apparently it is necessary to temporarily break the connection after this to make the interface run properly. When this is recreated under Linux by manually unplugging and reinserting the device there where no more problems. Recently this issue has been resolved with version 1025 of the microcontroller firmware. The information obtained to get the interface working has been added to an online wiki: http://gentoo-wiki.com/HOWTO_Xilinx

3.4 Building the Dataset

3.4.1 Extracting Switch Settings/Building Wire Databases

Although Xilinx does provide some information on the bitstreams needed to reconfigure the device, this only covers how the stream is divided into commands and frames. There is no detailed description on which bits to set to provide a pathway for a signal. To obtain this information, we therefore need a way to build low level device configurations and relate this to the bitstream. The *fpga_editor* program in the ISE [30] software package allows for manual manipulation of the FPGA resources. It provides a visual interface depicting the internals of the FPGA. It also has a scripting facility to store and repeat manipulations. Among its functionalities there is a way to specify the wires over which a signal has to be routed, selecting each individual segments along the path. We can use this mechanism and can derive all the wires which the signals can possibly take through inspection. This provides enough information to build ad-hoc scripts that systematically generates pathways and produce the bitstreams using the *fpga_editor*. From the bitstream, we have determined which bits are responsible for selecting a specific route. This information is stored in a wire database. In hindsight it would have been much better to use ”xdl -report -pips” to determine the device structures and use ”xdl -xdl2ncd” to generate the .ncd files as this would be a lot less device dependent or labor intensive. Unfortunately this option came to the attention when the data was already gathered and we wanted to move on.

3.4.2 *fpga_editor*

The *fpga_editor* program of the ISE package allows for low level manipulation of FPGA resources. It has a graphical user interface depicting the layout of the resources in the device. This can provide an abundance of information on the device we are studying. It also provides a means of manipulating the resources, allowing the user to manually place and route components. It seems to be the only tool that can be used to direct the pathway of routing. The manual routing of a few traces is possible in the *fpga_editor*, but it is too labor intensive a process to perform on large structures with a large number of lines and destinations. It involves manually selecting all the wires along the path, but due to the limited possibilities of wires to chose from at each switch box, the chances of winding up at the correct endpoint are not great. We will concentrate on deriving all the information necessary for automating routing. We need a way to extract the information from the program. It would be too much work to manually copy the information by hand. Fortunately the program provides scripting facilities. These scripts are a transcription of all the actions done by the user. It includes selection of wires, adding hardware, nets and routing.

Figure 2.3 provides a view of a single CLB in the *fpga_editor* program. On the right side, 4 smaller boxes represent the slices in a CLB. Each slice contains two 4-input lookup tables as well as two flip-flops. The lookup tables can also be configured to work as a small RAM or a shift register. There are some direct paths between the slices and neighboring slices, for producing fast carry chains. We are not going to do any synthesis so the actual structure of the slices is of less importance to us. We are mainly interested in the types of wires and the switch settings that are possible. Normal signals are connected to a large switch box as depicted on the left. As far as we have been able to determine by inspection, the CLB switches connect to 435 wires, most of them are input or output only. As far as could be determined, there are 3264 connections possible between these wires. Each of the 160 outgoing signals can be connected with between 4 to 37 incoming signals.

The types of wires displayed by the *fpga_editor* are depicted in Figure 3.1. For normal signals, they are divided into five groups:

- Wires connecting to neighboring cells, including wires that connect two of them, forming a square excluding one of the CLBs.

- Wires connecting to the second and fourth CLB, called double lines.

- Wires connecting to CLBs spaced 3 and 6 away, called hex lines. A few of these are fully connected to all 6 CLBs, which the wire passes. Both double and hex lines sometimes have an extra CLB connection at the end, offset by one row.

- Long lines, for routing critical paths over large distances, which stretch the entire width or height of the device. These have only 4 connections for each CLB, but there are 24 running along the column or row. The CLBs connected to the same signal are spaced 6 CLBs apart from each other. Long lines can be driven from each connection, so care has to be taken to prevent a double driven wires.

- Wires to create tristate buses in the chip. These involve special drivers depicted at the top of Figure 2.3. As the same problem can occur with tristate buses these are also types of wires we which use should be careful of, although these might be better protected against damaging themselves. We will refrain from using these wires all together to prevent accidental damage to the device.

As the switch is not fully connected we need to derive all the possible switch settings that can be done for each wire. Fortunately the *fpga_editor* program provides a visual way of determining this, which was intended as an aid for choosing the right path when manually routing a signal. These connections have been copied down to a file manually. Later it was discovered that there was an easier way to determine this, using the "xdl -report -pips" command. The depiction of the FPGA is largely dominated by the wires

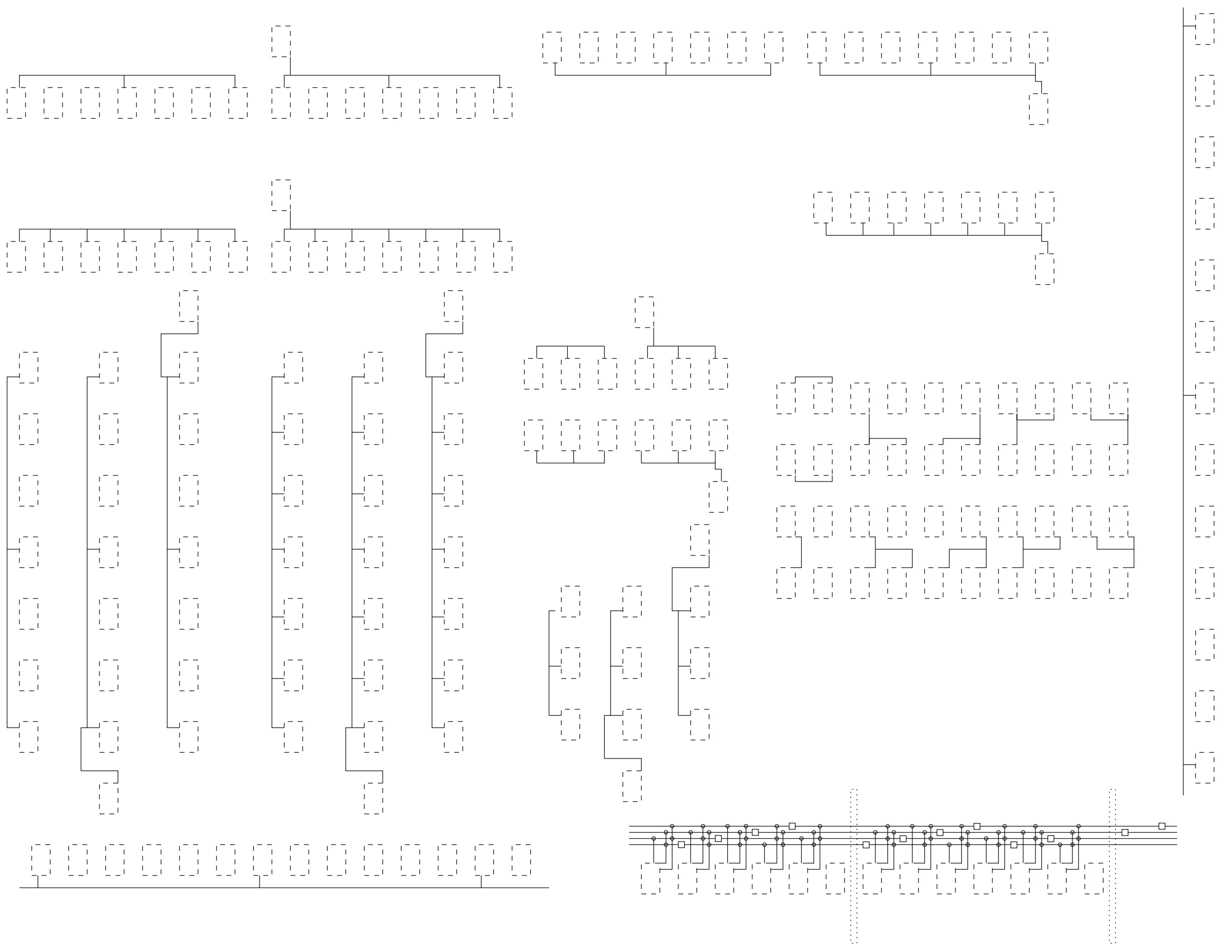

Figure 3.1: Virtex-II Pro wires as shown in fpga_editor

running through the device. Each wire has a unique address of 2 numbers referring what seems to be an x and y position. Inspecting the sequence of addresses for these wires we observed that there is some correlation between the addresses of the wires with the same type, but there are lots of discrepancies. These discrepancies seem to be caused by underlying structures which are not always visible. It was decided that we needed a wire database containing the addresses of most of the wires. This database has been obtained by trying to interpolate all the addresses as good as possible. Using the scripting facilities of the *fpga_editor* program the addresses did not have to be copied down manually, but could be determined from the script recordings. The wires have to be selected in sequence. The erratic jumps where mainly in the direction the wires where running, so a horizontal wire showed anomalies in the X coordinate. This enabled the interpolation of most of the wires in the Y direction. Ad-hoc scripts, interpolating the data, where refined as the process went along. A number of discrepancies where resolved by manually correcting the errors. This method has been successful resulting in the addresses of 1,005,054 wires, although it has been rather labor intensive and only covers the wires connecting the CLBs, and BlockRAM interconnects. This is all the information needed to generate the software for routing pathways automatically. The relation of the routing to the bitstream, configuring the device, is the last step of information we need to realize our goal.

3.4.3 Bitstream Description

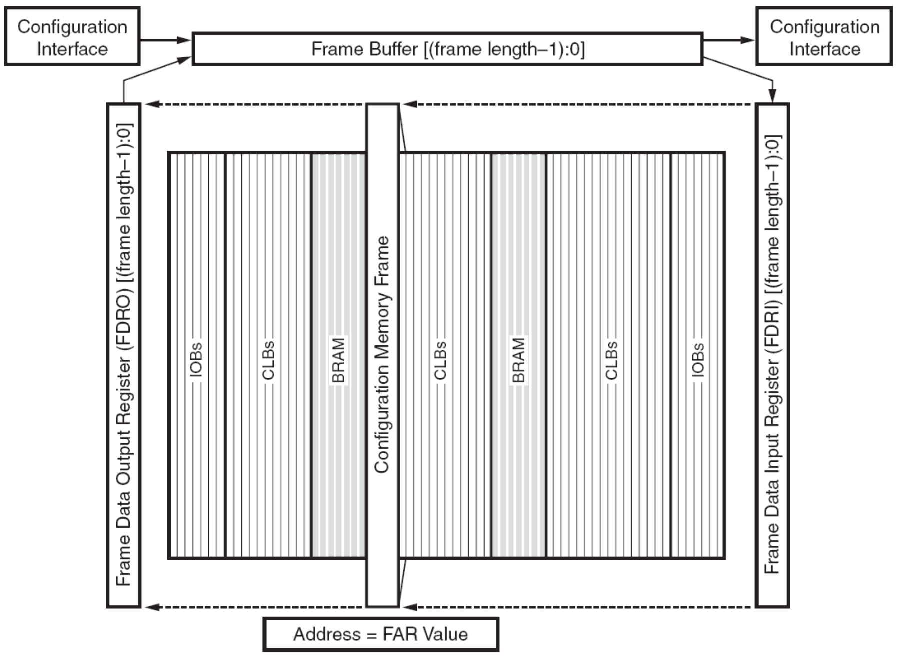
Configuration Interface
Frame Buffer [(frame length–1):0]
Configuration Interface
Frame Data Output Register (FDRO) [(frame length–1):0]
IOBs
CLBs
BRAM
Configuration Memory Frame
CLBs
BRAM
CLBs
IOBs
Frame Data Input Register (FDRI) [(frame length–1):0]
Address = FAR Value

Figure 3.2: Virtex frame buffer

The Virtex-II Pro has an internal interface for reconfiguration: the ICAP interface. It is a derivative of the external SelectMAP interface. The 8 bit interface accepts reconfiguration bitstreams consisting of synchronization, commands, data and padding as specified by the user guide [34] . The configuration logic uses a frame buffer to hold the data, which was read to and from the device memory (Figure 3.2). The size of the frames depends on the number of CLBs in a column. In the XC2VP30 it takes 6529 bits or 824 bytes per frame. When constructing the reconfiguration bitstreams, it is important to use padding frames. For instance, the first frame which is read from the interface contains what is left over in the frame buffer, and is therefore not useful. When writing a single frame to the device the contents of the frame buffer has to be shifted out while the new data is shifted into the buffer. This is followed by shifting the configuration data from the buffer to the device. While shifting the configuration data from the buffer to the device, the frame buffer is being filled up by a padding frame which is part of the reconfiguration bitstream.

Xilinx have developed their own layer on top of the ICAP interface, known as XPART [4], hiding the generation of reconfiguration bitstreams and the IO necessary to interface with the ICAP interface. This has been implemented in software as an API for the Microblaze and Power PC architectures. It provides seemingly random read/write access to CLBs, as well as the option of copying them from one location to another. It can also manipulate individual bits of a CLB. This API has never been released. Another option of applying the bitstream manipulations is to use JBITS, which is a java-based tool to reconfigure and manipulate the bitstream. Unfortunately this software only supports the Virtex-II and older devices, not the Virtex-II Pro and up. We will be using the JTAG interface to deliver the bitstreams with the *impact* program. This means we have to generate the bitstreams ourselves. As the bitstream for all interfaces is the same, the method of delivering them is not important.

```
Source file: top.ncd, part: 2vp30ff896, date: 2006/04/12, \
                        time: 11:17:46, length: 82100 bytes
30008001  Type 1 packet, write 1 word(s) to CMD : RCRC
3001c001  Type 1 packet, write 1 word(s) to IDCODE  0127e093
30012001  Type 1 packet, write 1 word(s) to COR  00053fe5
30008001  Type 1 packet, write 1 word(s) to CMD : SHUTDOWN
30000001  Type 1 packet, write 1 word(s) to CRC  00007a94
20000000  Type 1 packet,  0 word(s) to CRC
20000000  Type 1 packet,  0 word(s) to CRC
20000000  Type 1 packet,  0 word(s) to CRC
20000000  Type 1 packet,  0 word(s) to CRC
30008001  Type 1 packet, write 1 word(s) to CMD : AGHIGH
30008001  Type 1 packet, write 1 word(s) to CMD : WCFG
30002001  Type 1 packet, write 1 word(s) to FAR           \
                           ba:0 mja:3 mna:4 byte:0 -> CLB 1
300040ce  Type 1 packet, write 206 word(s) to FDRI
0,3,4 IOB    : 0300  ***********
0,3,4 IOI 082: 0000 0000 0000 0000 0000
0,3,4 CLB 081: 0000 0000 0000 0000 0000
0,3,4 CLB 080: 0000 0000 0000 0000 0000
0,3,4 CLB 079: 0000 0000 0000 0000 0000
0,3,4 CLB 078: 0000 0000 0000 0000 0000
0,3,4 CLB 077: 0000 0000 0000 0000 0000
0,3,4 CLB 076: 0000 0000 0000 0000 0000
0,3,4 CLB 075: 0000 0000 0000 0000 0000
0,3,4 CLB 074: 0000 0000 0000 0000 0000
0,3,4 CLB 073: 0000 0000 0000 0000 0000
0,3,4 CLB 072: 0000 0000 0000 0000 0000
0,3,4 CLB 071: 0000 0000 0000 0000 0000
0,3,4 CLB 070: 0000 0000 0000 0000 0000
...
```

Figure 3.3: Text representation of a bit file partially reconfiguring the 5th frame of the first column

The .bit-file used for configuring the device consist of a header containing information about the targeted device, source files, date of compilation etc., followed by the configuration bitstream. Using the information provided by Xilinx in the user guide [34] for the device family, we have constructed the *anabit.pl* program that reads the binary bit file used for configuring the device and translates it in a more human readable form, displaying the commands and showing the frame content in a hexadecimal notation. The data sheet also provides the information to show which frame relates to which column in the device. As the number CLBs in a column is known and they are distributed evenly over the device it can be deduced to which row the bits in a frame are related. Each line of frame data in the text file therefore contains 80 bits, which are all the bits related to a CLB in a certain row. The data is preceded by the frame address, the type of data, and the row number. The output showing the first 28 lines of a bit file used for partially reconfiguring a XC2VP30 is depicted in Figure 3.3. The *mkbit.pl* tool translates this human readable form back to a bit file which can be used to program the device.

The hexadecimal representation of commands are depicted first. The word size is 32 bits. The packet type determines the maximum size of a the packet. Type 1 packets are used for commands and short sequences of frames. Type 2 packets are used only for frame data and can be long enough to contain a complete device configuration. The number of words written are displayed as well as the register to which it is written. Commands written to the command (CMD) register are translated into a human readable text. Addresses written to the frame address register (FAR) are translated to a readable representation of their position in the bitstream, as well as the column it represents in the device, including the column type. Frames are followed by a CRC code which can be a dummy value if CRC checking is disabled. We have refrained from using the CRC checking feature of the bitstream. Due to a buffer being used by the device the last frame written to the device is a padding frame to facilitate the writing of the last data to the actual device configuration memory.

Replacing the bits of a certain CLB therefore becomes equivalent to replacing certain lines in the textual representation of the bit file. Addition or subtraction of bits from the bitstream for a certain CLB means adding or subtracting these bits to or from a few lines of the text file. For the convenience of not doing these manipulations manually there are the *addstr.pl*, *substr.pl* and *repstr.pl* programs for adding, subtracting and replacing bits within a bitstream. They take the textual representation of a bitstream and a list of address and row numbers of the frame data to which the manipulation is applied. The tools detect the frame and row addresses at the beginning of the line and perform the operation on the data in that line.They produce a bitstream identical to the input bitstream, except for the manipulations that are applied to it.

3.4.4 Extracting Switch Settings/Building Wire Databases

With the information of the types of wires, the possible switch settings and the addresses the wires have in *fpga_editor* we can systematically generate pathways which contain a CLB with only a single switch setting. For this we generate all pathways possible starting from the same point. We limit the number of switches taken to a predetermined value. A number of criteria are applied to this list, to form a list of pathways we are going to synthesize. The pathways which will be investigated are chosen in such a way that only CLBs are used, none of the other device structures are of interest. If it contains a switch setting which is not in the wire database and only contains that single setting at the CLB of interest, it is a candidate for further investigation. The pathways are translated into a script for the *fpga_editor* which opens a clean device configuration, routes the specific path and saves the configuration in an .ncd-file. The device is cleared and the next path is generated and saved in a new configuration file. The resulting .ncd-files can be translated in a .bit-file by the *bitgen* application. These .bit-files are translated into a .txt-file by the *anabit.pl* tool. When generating the pathways, a separate information file is also generated which contains the location of the CLB of interest and the switch setting it represents. Using another script the bits for this specific switch setting at the given location are extracted from the .txt-file and stored in a separate file containing only the bit patterns of the lines which contain data, and a frame number. Curiously almost all switch setting need only two bits to be set. We have determined 3264 possible connections for the switch matrix, in this process we had to generate 7588 paths, generate the associated .bit-files and analyze the data.

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
40044004400440047008800740044004400440044004400440044004700880074004400440044004
40444440440444440440644444604406444446040033334430440644444604404444404404444404
44444444444444444466644446646646644446664540322544464664444666444444444444444444
46444444644644444464464444446446444444644644444464464444446446444444644644444464
66644446666664444666666444466666644446666664444666666444466666644446666664444666
99999999999999990780087099999999999999999999999999999999078008709999999999999999
99999999999999998888888899999999999999999999999999999999888888889999999999999999
44444444444444444344443444444444444444444444444444444444434444344444444444444444
44444444444444444434434444444444444444444444444444444444443443444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444444444444444
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
44444444444444444444444444444444444444444444444444444444444444444444444444444444
2424a45424245424a42424242424442424244424242444242444242424242424a454242454a42424
```

Figure 3.4: Hexadecimal count of bit occurrence in the frames of a CLB

The distribution of bits over the frames is depicted in Figure 3.4. It can be clearly seen that the bits in certain frames are used more often then others. These bits are used for wires which can be connected up to more position than other wires.

With this wire database, the knowledge of the switch settings, and pathways the wires can take, we have built a tool chain that facilitates the manipulation of bitstreams. As the tools are meant for an academic environment, the bitstreams are translated to a format better suited for viewing and manual manipulation. The tools can handle this text-based readout after which the bitstream is translated back into a binary format that can be used to program the device.

3.5 Building Bit-Manipulation Tools

The snippets of bitstream, stored in the wire database, are represented in a similar format as the human readable output of the *anabit.pl* tool. Adding a new signal path to the bitstream is the equivalent of prefixing the appropriate address for frame and row to the lines in the wire database and use the tools to add the bits to an existing bitstream. Similarly, the wire database can be searched for the bits encountered in the bitstream, thereby extracting the switch settings used for a signal path. With these switch settings the routing of a signal in the current configuration can be retraced. We have constructed programs that extract switch settings and can backtrack these settings to deduce a wire list.

The *dumpswitch.pl* program takes the text representation of the .bit-file and matches each switch configuration with the database. The program takes the first bit out of a list containing the switch configuration and searches the database to find all switch settings which contain this bit. It determines the setting matching the maximum number of bits. These bits are taken out of the list and it searches for other switch settings in the remaining bits. The output is the frame and row of the switch, and the switch settings.

The *dumpwire.pl* does the same internally, but also backtracks the switch-settings to to the root node, starting from a random switch. It then forwardtracks all the switches in the wire and removes all of them from the list of switch settings, and repeats this process until all switch settings have been moved to the wire list. The wires are printed in a random order. Each line first contains the root node and the destination nodes, followed by the switches used in the pathway, traversed in a top-down manner. The addresses of the switches are now X-Y locations, and we have developed programs which can translate these setting back to the bits needed to configure them. We also have a program that can translate the switch settings back to a *fpga_editor* script which selects the wires involved. This enables us to generate bitstreams and check them against the same manipulations done in the *fpga_editor* program to prove them correct.

3.5.1 Generating Routing

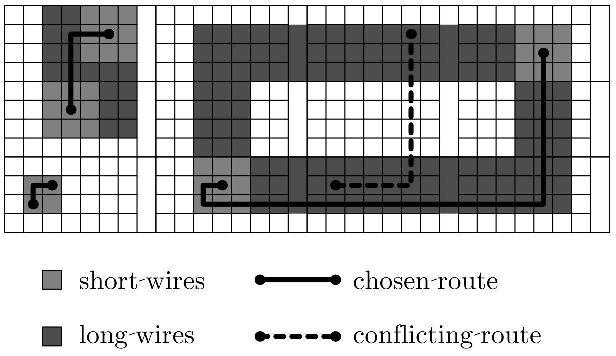

Figure 3.5: A simple implementation for routing 2 points

With the the information gathered we can also route a signal between two points. It is fairly cumbersome to do routing by hand as is the case with the *fpga_editor* tool. We have developed a simple implementation of a router which is able to generate a path between two points. The employed algorithm as depicted in Figure 3.5 searches for all possible routes between the two points. As this is incredibly inefficient, the search space has been limited. We differentiate between short lines connecting only to neighboring CLBs and longer lines connecting CLBs spaced 2 or 6 positions apart from another. For short distances we only search the short lines in the surrounding area. For larger distances we only allow usage of short wires near the points we want to route between. The search space for long lines is restricted to the rectangular area between the points, excluding the area in the middle. We also employ a restriction on the number of switches used relative to the distance traveled from the source and we have an absolute maximum to the number of switches. The paths generated are steel square shaped. The parameters can be tweaked if no suitable route has been found. The resulting router is far from optimal and very slow, but as building good routers is in a field of its own, for now we are not interested in improving it.

The input of the router is a wire list, containing the root and destinations of the net. It also takes an existing device configuration in text representation. It first maps the wires in the existing configuration and only routes the wiring that is not present already. If the generated routes meet an existing wire that has the same root, the route up till that connection is removed from the list. Because the routes are sorted to length these will automatically be preferred as they are the shortest. Routes that encounter switch settings with a different root are removed all together. The first item of each of the lists of routed nets are tested for conflicts. If these occur, the next in line is selected for the route causing the most conflicts. This is repeated until a suitable combination is found or a threshold is met, in which case the routing has failed. The output is a list with the switch settings of each wire per line. This can be translated to a bit representation or a *fpga_editor* script.

3.6 Advanced Tools

These tools would be enough to combine various modules by hand. From existing configuration files, the slice configuration data can be cut and paste. The wiring can be extracted and repositioned by replacing it at the correct location. New routes can be generated using the router. Of course this is a cumbersome method, and has only been done to test the methodology for correctness and feasibility. All work done up to this point was the basis for the ProRISC2006 publication. There was a need to develop tools to automate this process. For isolating the various modules from an existing design we have developed a tool that takes a configuration text file and the region the module resides in. It extracts just the configuration data of the module and a list of nodes through which it communicates externally. A placement tools relocates the configuration data to another location on the chip. It performs all the necessary checks to test if placement is possible. An unrouting tool takes a device configuration and a netlist containing the source and destination(s) of the routes that have to be removed. This can be a subset of the destinations a particular net has, it only produces a list of switch settings which have to be removed for those particular pathways. Finally a tool has been build that combines all operations. It takes a configuration description accepting multiple instances of various modules at a user specified location. It takes a netlist and performs all the necessary operations for placing and routing. If a modification is made to the configuration description it rebuilds the bitstreams, removing the old configuration and placing the new. It calculates only the differences and produces a bitstream that does partial configuration.

3.6.1 Hardware Core Isolation

It is observed that all bits for configuring the routing are positioned from the 3rd frame on in the 22 frames for a column of CLBs. This indicates that the first three frames are used for configuring the slices. This data is treated as-is, it contains the hardware description. We want to isolate a portion of the configuration data, essentially preparing it for use as a macro which can be placed anywhere in the device. For this we need to designate the area the hardware resides in. The ISE software allows for assigning an area for a specific piece of hardware with the floor planner. All we need to extract the module are the coordinates in which the hardware resides. In this area we can identify the CLBs that contain hardware, the internal wiring, the incoming and outgoing signals and wires which just pass through the area. Isolating the hardware encompasses the discrimination of these wires, detecting the actual area hardware resides in (as it can be smaller than the actual designated area), and removing all the unwanted elements. As the internal wiring can be larger than the area the hardware resides in, and the ISE software does not allow for limiting the area of this wiring, we also have to determine the area in which these are placed. This has been accomplished by an extra addition to the wire-tracing algorithm. Not only the root of each switch node is determined, but also a list of destinations is attached to the switch node. The combination of root and destinations is matched to criteria to determine if a switch-node is part of an external network or part of the internal wiring.

A windowed bitstream is created based on the area in which hardware has been detected. This bitstream is empty, except for this area, which is copied from the original bitstream. Subtracted from this bitstream are wires which are just passing through, wires that have a root inside the area, but only destinations outside the area, and wires which have a root outside of the area. To this bitstream, we do have to add wires have both root and destination inside of the designated area, but which partially traverse a pathway outside of it. The outgoing signal wires need some special attention, as they can also be for internal use. The resulting data is a portion of reconfiguration data, containing the hardware and internal wiring only. All wires which are only passing through are removed, as well as the incoming and out-going signal wires.

The *isolate.pl* program takes a text representation of a bitstream and an area in which the hardware core resides, and produces a file containing the inputs and output positions of the signals, and the configuration data within the designated area. As an input signal can be used at multiple locations these are placed on the same line, for later routing. The same program can be used to remove an existing core from the device configuration. For this, the hardware core which has to be removed is isolated. These are subtracted from the existing device configuration, still leaving the routing of the signals to and from the module configured. These have to be identified and removed separately. It is be better to store the data of the placed hardware core for easy subtraction afterwards, instead of deducing it from the bitstream every time.

3.6.2 Unrouting

As a derivative of the isolation tool, an unrouting tool was also developed. It takes a text representation of a configuration file and a netlist of routes that have to be removed. It determines the existing configuration and produces a list of wires which meet the requirements of the removal netlist. These wires can be translated into bits which can be subtracted from the bitstream.

3.6.3 Hardware Core Placement

Placement of hardware is simply repositioning the data to the appropriate location. The problem resides in the heterogeneous character of the Virtex-II Pro. Every six columns of CLBs are interrupted by a Column of BlockRAMs and multipliers. This means that we have to check that no CLB data in the hardware core are positioned at the position of a BlockRAM and vice versa. Likewise a BlockRAM/multiplier has an equivalent height of 4 CLBs. If a BlockRAM is used in the design vertical shifts are limited by multiples of 4 and horizontal shifts are limited to multiples of 7. For the wiring, checks have to be made testing if the additional wiring does not conflict with the current device configuration. The implementation of the *place.pl* program takes an existing device configuration, an isolated module and the X-Y location of the bottom left corner for the new location of the module. As the module is only isolated, and is not normalized to a standard location, the hardware and wiring area have to be redetermined using the same techniques as in hardware core isolation. The repositioned data is stored in exactly the same format as an isolated module, all frame-addresses and signal positions are skewed to the appropriate location. In this manner we can reuse the relocated core as an input-file. It is up to the user to determine problems with distributed RAM and BlockRAM corruption when doing partial reconfiguration. No checks are made as this will not pose a problem for full-device configuration, only in runtime partial reconfiguration.

3.7 Device Composition Description

To fully automate the placement and exchange of hardware cores, a tool has been developed that uses the previous described tools to do just that. A file format has been specified in which the user can assign multiple instances of various modules to user-specified locations. It also has a netlist describing the signals between the various modules. A default configuration bitstream and signal positions are used as a base configuration. This is needed to arrange for communication with the world outside of the clique of modules, for instance to set the drivers of the various IO pins. Eventually this could be a full configuration of the Molen platform. The arrangement of cores, the netlist and base configuration bitstream determine the composition of the full device configuration.

The *build.pl* scripts takes this composition file and uses a cache directory to store its data. If the cache is empty, a full device configuration will be generated. All modules are relocated and stored separately in the cache directory. The list of input and output signals are related to the netlist in the composition file. From this, a netlist for the router is generated. In a second step, all the modules are combined with the base configuration bitstream to form a configuration with all the modules placed. The router is used to generate the connectivity between the modules. After adding the configuration bits produced during routing to the bitstream, the generation is finished. The composition file is stored in the cache directory for later use. When an alteration is made to the composition file, this can be compared with the stored version. The bitstreams of the modules that have to be removed from the composition file can be removed from the full configuration file by subtracting the cached versions of them from the bitstream. New modules can be stored in temporary files. A new netlist is generated from the new composition. This is compared with the old netlist which was previously stored. From this a removal list is derived, with which the switch settings that have to be removed are extracted by the unroute tool. These bits are also subtracted from the current configuration file. This forms an intermediate configuration with all the hardware and wiring no longer used in the new composition removed. This is stored for later use.

The new modules are placed on this intermediate configuration file, and the router takes this and the netlist to produce the new connectivity between modules. As the router only routes pathways which are not yet in the existing configuration, only the new routes are generated. These are added to the bitstream to form the final new configuration. In the cache directory we now have the old, intermediate and new configuration. To prevent damage to the device, the partial reconfiguration bitstream between the old configuration and the intermediate configuration is produced. The same is done for the intermediate and the new configuration. These are combined to form a single partial reconfiguration file, removing the old and configuring the new modules. For now we will take this conservative approach, but it might be possible to reduce the size of the bitstream for this process. The result of all the work done has been the basis of the FPL2007 paper.

3.8 Partial Reconfiguration

The *partial.pl* program takes 2 full device configuration bitstreams as input and determines the differences between their frames. It produces an output which only reconfigures the frames which are different. This is the essence of partial reconfiguration. It is up to the user to produce the bitstreams, there is no consideration for conflicting switch settings which may occur during reconfiguration. The *bitgen* program can do the same for a .ncd- and a .bit-file, but not two .bit-files. It does however employ a bitstream compression technique in which it combines frames with the same content into a single write operation. We have not implemented this feature. A large portion of the partial reconfiguration data is padding, which has to do with the sequence of events for writing data to random frames. This starts with writing the starting address to the frame address register, followed by the frame data for the consecutive frames, and ending with a padding frame to write the last frame residing in the frame buffer to the correct location. As the frames containing differences are mostly non consecutive, this involves using a lot of padding frames. The compression techniques writing the same frame to multiple locations can therefore save a lot of data, by not only reducing the amount of data written but also the number padding frames.

3.9 Verification of Data and Methodology

During the whole process, proof of correctness is difficult. Some of the data, as for instance the possible switch settings, was copied down manually. This is a process that is prone to error. As the scripting facilities of the Xilinx tools where used for gathering data, an error or lack of output while processing these scripts was an indication that there was an error in the data already obtained. Because we can translate the generated pathways to bits in the bitstream or to a script for *fpga_editor*, we can test if these produce identical bitstreams. As long as both methods produce the same results we can be confident that our bitstream is correct. The data was also tested for duplicates and discrepancies where investigated and corrected. An interesting result was obtained with the use of long lines, as these sometimes take 3 configuration bits and sometimes 2. It is suspected that this has to do with a driver that has to be enabled for driving the long line, but as the use of long lines has been abandoned for this project this has not been investigated. Testing if a pathway can be taken involves inspecting if a wire is already driven from another source. As there is no specific knowledge of the internal structures of the FPGA we can only derive some conclusions from the bitstream itself. Inspecting the bits for configuring the switch matrix reveals that each bit only has involvement with driving a specific wire. The bit combinations themselves select a source. Because of this it is unlikely that a switch setting for driving one wire will have a conflicting influence on the switch settings for another wire, they seem to be independent. Therefore we only test for driving conflicts on the basis of the wire driven. Fully testing interdependence would involve generating over five million scripts for *fpga_editor*, which would take too much time. This test is used for determining if a configuration does not cause any electrical conflicts, and is used in various of the tools but also in a special *erc.pl* program, to test if a configuration file can cause damage to the device.

3.10 Conclusion

The pathways of all 435 wires connecting to a CLB switch-matrix have been determined and their regularity has been observed. As far as could be determined through inspection in *fpga_editor*, there are 3264 connections possible between these wires. Each of the 160 outgoing signals can be connected with between 4 to 37 incoming signals. Through interpolation and ad-hoc scripting we have determined the addresses of 1,005,054 wires in *fpga_editor*, although it has been rather labor intensive and only covers the wires connecting the CLBs and BlockRAM interconnects. The bitstream was decomposed in commands and frames. The bits associated with individual CLBs have been determined, to be able to relate specific switch settings to specific parts of the bitstream. We have determined the CLB bit settings to all 3264 possible connections for the switch matrix, in this process we had to generate 7588 paths, generate the associated .bit-files and analyze the data. It is observed that all bits for configuring the routing are positioned from the 4th frame on in the 22 frames for a column of CLBs This indicates that the first three frames are used for configuring the slices. It is also observed that almost all switch settings connecting one type of wire to another involve setting only two bits in the bitstream. Tools have been built to manipulate bits in the bitstream, to determine switch settings throughout the bitstream and to traverse wires within the bitstream to produce a wire list. A primitive router has been created to take a netlist and produce the wiring needed, as manual routing is hardly possible. To unroute full and parts of a net, an unrouting tool has been created that takes a netlist and produces a bitstream with these nets (or just the parts connecting to specific points) removed. For isolating a hardware core from the bitstream a tool has been created that, provided with the area in which the hardware core resides, determines the communication wires with outside structure, removes any unnecessary wiring, and produces a windowed bitstream containing just the hardware core and its internal wiring. This is combined with the information of the LUTs to which external signals have to be routed to. A placement tool has been created that performs checks to make sure a hardware core is allowed to be placed at the desired location, not conflicting with existing configuration data or device resources, and produces a bitstream with the hardware core added. For partial reconfiguration we have produced a tool that takes two bitstreams, and produces a partial reconfiguration bitstream that reconfigures the frames which differ between the two input bitstream. All these tools are combined in a framework that takes a base bitstream for the fixed parts of the device configuration, a list of module instances and their location, and a netlist on how to connect them. It produces a full bitstream configuring the device for the first time. When the framework is rerun with a change to the composition it will detect the changes and produce a partial bitstream, dynamically reconfiguring only the parts that have changed. The work presented has resulted in a publication for ProRISC2006 and FPL2007.

Results 4

In this chapter we present the tests and results to prove that the methods we developed are feasible, and perform as expected. We start out by testing out partial reconfiguration of routing using the difference based technique of XAPP290. Analyzing the bitstream for partial reconfiguration, we tested if partial reconfiguration bitstreams are shifted into the configuration on a frame by frame basis, or instantaneously after all data has been clocked in. With the data gathered and part of the tools built, we had to test if we could extract routing pathways and replace them by alternative routes on a bitstream level. Next, we test the feasibility of placing and routing a AND gate at an arbitrary position using both our own methods, and an identical approach in *fpga_editor* to compare the resulting bitstreams. Using manual manipulation we then replace a AND gate with a XOR gate at a different location, after which we reproduce the same results using an automated framework.

4.1 Testing Feasibility

When starting this project, it first had to be tested if it where possible to reroute signals using difference-based partial reconfiguration. A simple test was devised, routing two switches to two LEDs on the XUP board. The routing was done manually using the *fpga_editor* program. A second instance was made using almost exactly the same routing, but reversing the LED connectivity. This involved a minor change to only a single switch box. Using the partial reconfiguration option of the *bitgen* program a transition of one configuration to the other was tested, which worked. This simple test encouraged further research into this method of partial reconfiguration. The two routes are depicted in Figure 4.1.

4.2 Testing Runtime Reconfiguration for Building Partial Bitstreams

Later on during the design process, we used the previously described bitstreams for partial reconfiguration to test if these changes are applied during device configuration, or if they are postponed until the last command of the bitstream. Using the *anabit.pl* program a dump was made of the reconfiguration bitstreams from route1 to route2 and vice versa. These bitstreams where alternated multiple times, intertwined with large fields of NOPs. In this manner we created a bitstream that would either alternate the settings a few times a second if the partial reconfiguration is applied during the processing of the bitstream, or just a single transition or none at all if it is applied after the last frame was configured. Testing this theory proved that in partial reconfiguration frames

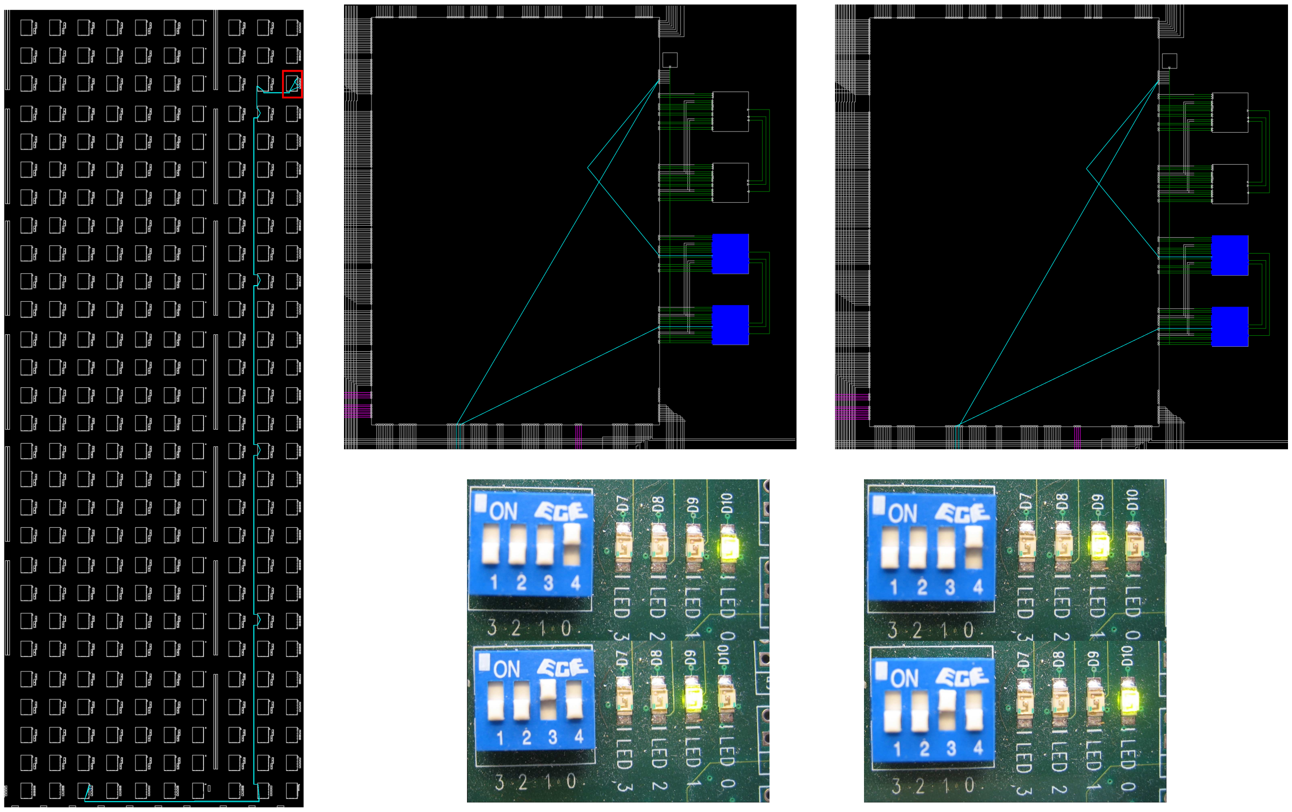

Figure 4.1: Routing two switches to two LEDs and reversing the connections

are applied immediately after they have been loaded into the device configuration, as the LEDs where alternating during the device configuration. This is a good result, as we can combine operations to remove old configurations and place new configurations using a single partial bitstream, an advantage over using two separate streams.

4.3 Comparing Bitstreams of Routes with *fpga_editor*

The previous tests have been done by manual routing. To test the switch-position extraction, a dump was made or the wiring in the previous bitstream. The wire configuration of route 1 was subtracted from its bitstream and the wire configuration extracted from route 2 was added. This should result in an identical bitstream to that of route 2, as was the case. Proving we can transpose one configuration to another by just adding and subtracting the bits of wire pathways. This has been done on both a bitstream and an *fpga_editor* level, although this required some minor manual manipulations as we did not have all addresses of the wires in the I/O interconnect (IOI). This also provided insight in the composition of the switch matrix in the IOI and BlockRAM interconnect (BRI). They seem to be connected almost identically to the switch matrix for CLBs allowing for the assumption that they can be configured as if they where CLBs, simplifying the routing problem. The original plan was to use a default configuration, which makes the external signals available through wiring which we can access, now we just need the CLB equivalent of the lines which holds the signals in the IOI area. As this specific switch matrix has not been elaborately studied we will not make any specific claims to this. The same holds for the BlockRAM interconnect (BRI) switches. Because we only use these for pass-through routing, not connecting directly to the resource, they too will be treated equally to a CLB switch matrix. The plan was also to program the router not to use these switches at all, in the end this was not necessary.

4.4 Simple Example (3 Input AND) for Verification

As a first example of placing a module we use a 3-input AND gate, which can be mapped to a single LUT. The ISE tools are used to assign the hardware to a specific LUT and generate the appropriate .ncd and configuration files. Using the .bit-file we can extract the wiring and determine the sources and destinations of the wires. The resulting netlist is fed into the router and the switch settings are produced as an output. These switch settings can be translated in bits or can be translated to a script file for the *fpga_editor* to do the same thing with the dump2*scr.pl* tool. The manipulations on the bitstream encompassed subtracting the existing pathways and adding the new pathways. The AND-gate remained in place. Figure 4.2 depicts the routes as generated by ISE and by our own router. The resulting .bit-files are essentially the same and produce the same result.

4.5 Partial Reconfiguration, Replacing the AND With An XOR Gate

The easiest way to replace the 3 input AND gate with a 3 input XOR gate would be to only replace the content of the lookup table. Because we are implementing the reconfiguration of entire cores, we have to develop a method that has to be a little more sophisticated. Similar to the 3 input AND gate a 3-input XOR-gate has been generated using the ISE tools. We have developed tools to isolate the modules from the rest of the bitstream and to generate a bitstream from these files which places them at a new location. The isolated AND and XOR-gates are stored as a separate hardware core using the *isolate.pl* tool. We then place the XOR gate at a different location from the AND gate using the placement tool. We can simulate these manipulation with the *fpga_editor* tool by placing it in ”read/write” mode and dragging the slice content to the appropriate location. We use our routing tool to generate the new pathways and we generate the new .bit-file and the script for implementing the routing in the *fpga_editor*. The resulting bitstreams are compared to see if they are equal. We can test this new configuration with the XOR-gate separately, they both function the same

Next we can have the *bitgen* tool produce a partial reconfiguration bitstream to substitute the 3-input AND gate with the 3-input XOR gate. Because this takes a .ncd- and a .bit-file, we could not be able to use this tool in our own tool chain, as we only generate .bit-files. We have therefore developed a program that produces partial bitstreams. The partial reconfiguration files produced by *partial.pl* are not identical to the ones produced by the bitgen tool. The latter user a bitstream compression technique. The results of applying the partial reconfiguration bitstreams to a device are equal. The pathways generated are depicted in Figure 4.2.

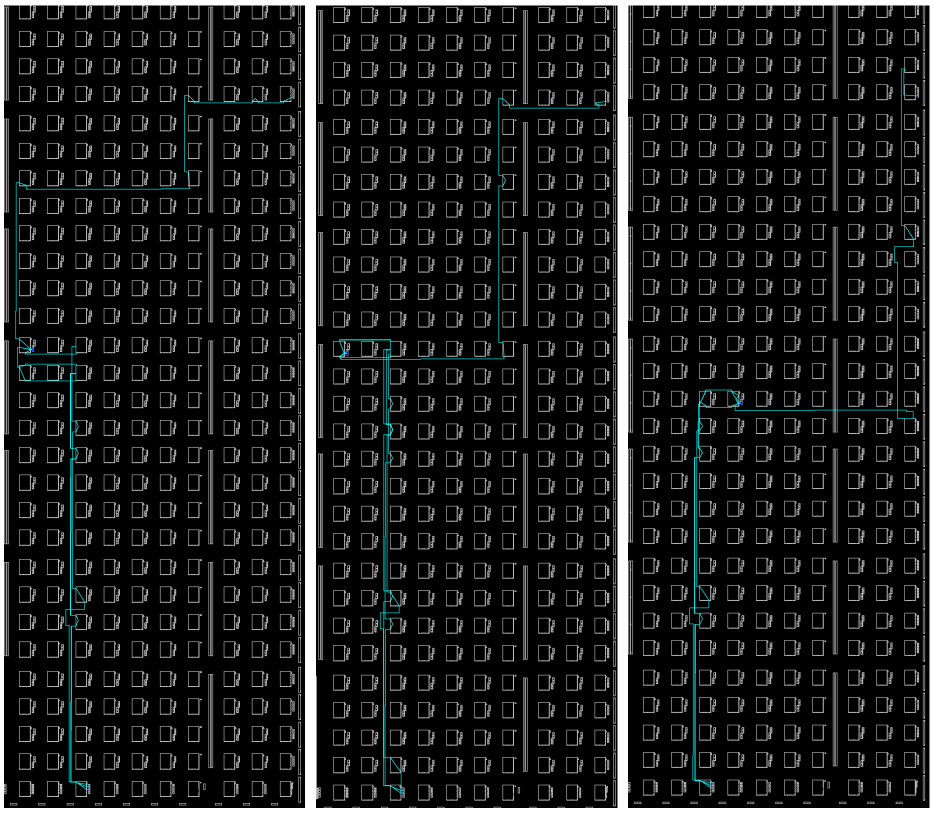

Figure 4.2: AND-Routing by ISE (left), our router (middle) and XOR-routing (right)

Because some manipulations have to be done manually, and there is a desire to automate as much as possible, as a last step a fully automated *build.pl* program has been developed. No comparison is done against the *fpga_editor* results, just a functionality test.

4.6 Automating the Process, Using a Composition File

The ultimate goal of this research is to develop a better technique to do partial reconfiguration. This not only involves showing the possibilities but also to implement them in a reasonably usable fashion. Up till now some steps needed manual manipulations to produce the proper bitstreams. As a last step we automate the bitstream generation process.

As a starting point for this process the previous example is used. This involves having a pre-generated 'empty' configuration bitstream, in this case configuring the IOBs to take

```
.cache=./cache
.mods=./mods
.basis_strm=./basis_strm.txt
.basis_nodes=./basis_nodes.txt
.reconfigstrm=./update.txt

.define 3and.txt and1(46,16)
.define 3xor.txt xor1(47,16)

*    in  in  in  out
and1 sw0 sw1 sw2 led0
xor1 sw0 sw1 sw2 -
```

Figure 4.3: Device composition input file

the signals from the switches and drive the LEDs properly. There has to be a file to indicate the location or pads of these signals on the chip. Default modules are created using the *isolate.pl* tool for the 3-input AND and XOR gates. These already contain the information for the signal locations. Finally we need a file that describes how all these parts fit together. The composition file, depicted in Figure 4.3, can change some default values for:

- the cache directory
- the directory in which the modules reside
- the default configuration stream in which the modules are placed
- the file describing the signals involved with communication for this default configuration
- a file name for the output stream

The actual composition is described in the definition of instances for a specific module, with a user specified location, and a netlist for these instances. The signal names are free to choose. There is a special token: '-', for describing an unconnected signal.

The first time the build script is run it produces a full configuration bitstream. The device can be programmed using this bitstream and the *impact* program. If a modification is done to the composition file, a partial reconfiguration bitstream is calculated, resulting in a modification to the new configuration. A test has been conducted using two different types of modules, the 3-input AND and XOR gates used earlier. As a series of tests, the 3-input AND is placed at a random location, it is replaced by a XOR at another location. The AND port is then placed again at the same location, but only the incoming signals are routed. After this, the LED is connected to the output of the 3-input AND. Finally the output of the AND function is used as an input to one of the

XOR functions and the output of the XOR is re-attached to the LED. All these transitions, while taking a lot of time for calculating the routing, are functional as expected. This concludes our tests.

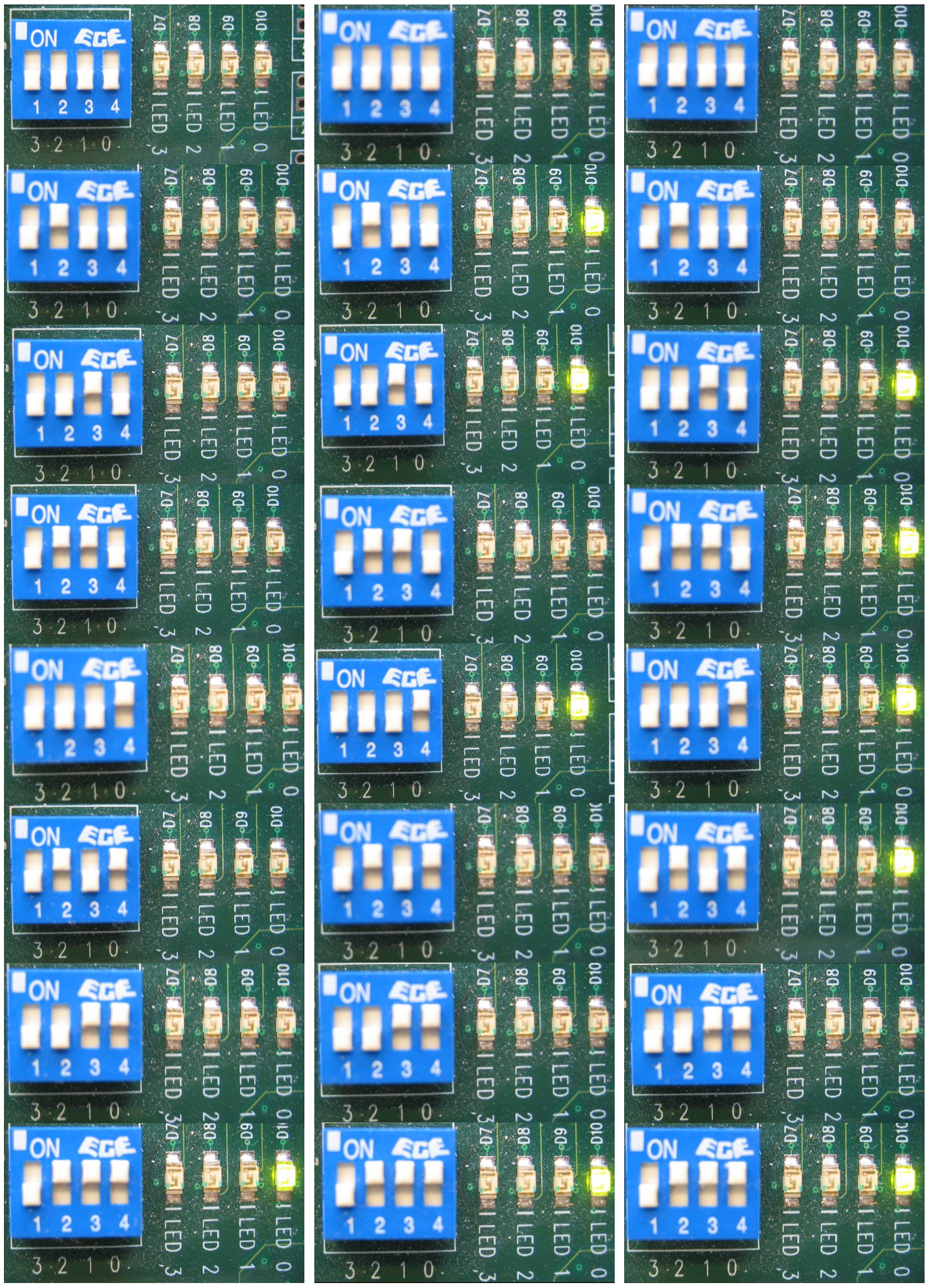

Figure 4.4: logic functions of 3 input AND, XOR and a combination

4.7 Conclusion

Inspecting our results we can conclude that our methods are performing as expected. The data we have gathered, and the tools produced are complete enough to perform the operations necessary for removing, placement, routing and unrouting of structures on an FPGA. This can be done partially during runtime. The frames are immediately shifted into configuration memory as the stream is processed, so care has to be taken in preventing damage to the device by first removing existing hardware configurations and routing before configuring new ones. The automatic framework producing these bitstreams takes care of the order, combining two partial bitstreams which first remove existing structures and then place and route a new ones.

Conclusions and Recommendations 5

5.1 Conclusions

In Chapter 2, we have shown that all current methods for partial reconfiguration rely on fixed interfaces between the modules and the routing paths, in the form of a bus macro or lookup table based primitives. We propose a method which does true direct switched routing of the signals. This provides more flexibility, as there are no predetermined routing paths or dedicated wires. It makes it possible to route through existing structures even though they do not have specifically reserved space for this. The lack of using bus macros or LUTs saves resources and reduces delays. Furthermore, the number of signals that can be connected is limited only by the available resources of the FPGA itself. We do 2d placement of arbitrarily sized and shaped modules. We do not reserve specific areas for the modules. As long as there are no conflicts with existing structures and configurations, a module can be placed and connected anywhere on chip. As all operations are done on a bitstream level, not using 3rd party tools, they can be implemented on the PowerPC or Microblaze. We make use of stored configuration data for the module and have a database of configuration data for the switch settings, used for routing the signals. Wires crossing boundaries in to other modules are not a problem, as long as they do not directly conflict with the wiring of the other module. Transients cased by reconfiguration have to be resolved in the modules themselves. Adding a logic gate, disabling communication during reconfiguration can be enough.

In Chapter 3, the pathways of all 435 wires connecting to a CLB switch-matrix have been determined and their regularity has been observed. As far as could be determined through inspection in *fpga_editor*, there are 3264 connections possible between these wires. Each of the 160 outgoing signals can be connected with between 4 to 37 incoming signals. Through interpolation and ad-hoc scripting we have determined the addresses of 1,005,054 wires in *fpga_editor*, although it has been rather labor intensive and only covers the wires connecting the CLBs and BlockRAM interconnects. The bitstream was decomposed in commands and frames. The bits associated with individual CLBs have been determined, to be able to relate specific switch settings to specific parts of the bitstream. We have determined the CLB bit settings to all 3264 possible connections for the switch matrix, in this process we had to generate 7588 paths, generate the associated .bit-files and analyze the data. It is observed that all bits for configuring the routing are positioned from the 4th frame on in the 22 frames for a column of CLBs. This indicates that the first three frames are used for configuring the slices. It is also observed that almost all switch settings connecting one type of wire to another involve setting only two bits in the bitstream. Tools have been built to manipulate bits in the bitstream, to determine switch settings throughout the bitstream and to traverse wires within the bitstream to produce a wire list. A primitive router has been created to take a netlist and produce the wiring needed, as manual routing is hardly possible. To unroute full and parts of a net, an unrouting tool has been created that takes a netlist and produces a bitstream with these nets (or just the parts connecting to specific points) removed. For isolating a hardware core from the bitstream a tool has been created that, provided with the area in which the hardware core resides, determines the communication wires with outside structure, removes any unnecessary wiring, and produces a windowed bitstream containing just the hardware core and its internal wiring. This is combined with the information of the LUTs to which external signals have to be routed to. A placement tool has been created that performs checks to make sure a hardware core is allowed to be placed at the desired location, not conflicting with existing configuration data or device resources, and produces a bitstream with the hardware core added. For partial reconfiguration we have produced a tool that takes two bitstreams, and produces a partial reconfiguration bitstream that reconfigures the frames which differ between the two input bitstream. All these tools are combined in a framework that takes a base bitstream for the fixed parts of the device configuration, a list of module instances and their location, and a netlist on how to connect them. It produces a full bitstream configuring the device for the first time. When the framework is rerun with a change to the composition it will detect the changes and produce a partial bitstream, dynamically reconfiguring only the parts that have changed. The work presented has resulted in a publication for ProRISC2006 and FPL2007.

In Chapter 4, inspecting our results we can conclude that our methods are performing as expected. The data we have gathered, and the tools produced are complete enough to perform the operations necessary for removing, placement, routing and unrouting of structures on an FPGA. This can be done partially during runtime. The frames are immediately shifted into configuration memory as the stream is processed, so care has to be taken in preventing damage to the device by first removing existing hardware configurations and routing before configuring new ones. The automatic framework producing these bitstreams takes care of the order, combining two partial bitstreams which first remove existing structures and then place and route a new ones.

5.2 Main Conclusions

- Our methods showed that we can remove a portion of the hardware configuration on an Virtex-II Pro FPGA and replace it by another.

- This involves the removal of the configuration data and the routing of the old hardware core, after which a new one is placed and connected.

- These manipulations are done at a bitstream level without the use of the Xilinx tools.

- We have tested our methods with simplistic hardware to show that it works on a functional level.

- We have developed tools to automate the process for developing the bitstreams used in runtime partial reconfiguration.

- The advantage of our solution is we no longer reserve a specific area for the modules, nor do we make use of reserved routing paths. This means we can truly arbitrarily place and connect any module to any other module.
- We do not make use of bus macros
- We do not place any restriction on the size and shape of the modules, although they do have to fit on to the device without causing conflicts with the existing configuration.
- As long as the router can generate a pathway, routing can go through existing structures.
- As long as existing pathways do not conflict with the routing within a new module, modules can be placed on top of existing routing.
- We can do 2d placement, enabling higher densities [27]. The Virtex-II Pro is however crippled in this department and it can be argued that doing 2d placement is not a suitable solution for this device family. It is better to concentrate efforts on the Virtex 4 family as it has region based partial reconfiguration, better suited for the purpose.

  Although the developed techniques are excellent for use in on-line bitstream generation, the Achilles heel is in the computation required for doing routing. We propose a number of steps which can be taken in the future to further develop and implement this technique.

5.3 Future Work

5.3.1 Bitstream Generation in Hardware

We propose to add a simple hardware implementation to the reconfiguration interface that can produce the desired reconfiguration stream independent of the processor. This will ensure that the reconfiguration process can run simultaneously with the software, hiding reconfiguration delay from the program. The Virtex-II Pro has an internal interface for reconfiguration: the ICAP interface. It is a derivative of the external SelectMAP interface. The 8 bit interface accepts reconfiguration bitstreams consisting of synchronization, commands, data and padding. When constructing the reconfiguration bitstreams, it is important to use padding frames, but these could be easily generated instead of stored. The hardware is to be kept simple at first, serving more as a cache for the reconfiguration bitstream. The technology can later be extended with an instruction set that can do more complex manipulations, exporting the actual placement of hardware to the 'reconfiguration processor', instead of generating these reconfiguration bitstreams off-line. For further development, the reconfiguration processor could use a standard dataset. The synthesized module data could be mapped to this standard set. The mapped data can then be translated to device specific implementations using the standard set of data. This enables the loading of the module onto devices of different families, although this might come at the price of larger and slower modules, just as full custom chips are smaller and faster then their sea-of-gates counterpart. If the critical path is mapped to a device specific implementation, and the less critical components are mapped onto a standard dataset, these problems may be overcome. Using standard building blocks, it may also be possible to modify the shape of a module without much penalty, as the critical paths can be 'fixed' and there is a relative degree of freedom to the placement of the non-critical components.

Oliver and Makell [24] aim to create a low overhead dynamic execution environment. They propose a hardware description based on what they call an object-oriented design. The objects in question are small modules, performing small functions. The hardware is described in terms of these functions. Combining functions entails placing them after each other in such a manner that the interfaces connect. The practicality remains to be seen as the hardware has to be mapped onto these standard blocks and it seems unlikely that they will fit neatly in each other. The interfaces are in a fixed position, which already suggests limited freedom in the placement of functional blocks. The techniques developed in this thesis can overcome these problems. Their proposed functional blocks are replacements of software functions. This is intended as a software to hardware translation, but if the modules are made smaller, it will probably be more efficient to map to HDL.

5.3.2 Reduction of Reconfiguration Time

None of our efforts have been in reducing the size of the partial reconfiguration bitstreams. There has been some related work on this matter, some of it can not be applied to the Virtex-II Pro as it would involve changes to the interface, but we can make use of, and expand on these methods.

Malik and Diessel [21] suggest the modification of the reconfiguration interface to take the previous contents of the memory and reuse most of it when partially reconfiguring the device. This greatly reduces the reconfiguration data, and the current configuration can be read back through the ICAP interface, but if we have to modify the interface anyway it makes sense to suggest more modifications for the benefit of partial reconfiguration which would not add to much complexity. Malik and Diessel [22] propose compressing the bitstream with Golomb encoding or run-length encoding. As bitstreams are largely composed of 0's this is highly effective in reducing the amount of reconfiguration data. Although it by no means accelerates device reconfiguration, as in practice decompressed reconfiguration bitstream still needs to be offered to the normal interfaces. The proposed architecture does reduce the reconfiguration time to within approximately 15%. It shows that it is very profitable to compress bitstreams for any case. Stepien and Vasilko [26] suggest making placement and routing algorithms aware of the columns based frames of the FPGA instead of a uniform structure. This reduces the usage of the number of frames up to 60%, it does however increase timing with 8-20%. This is mainly important for frame based organization of the FPGAs configuration memory. The previous mentioned 'standard' dataset could also be implemented in hardware, for instance by adding a single frame which configures these building blocks for the entire CLB. This can involve both standard slice configurations as well as standard connectivity configurations. The connectivity configurations can involve standard bus structures to quickly reconfigure chip-wide connections as well as local connectivity. This could involve assigning signals to default wires, much like bus macros are meant to affix a signal to a know location, but without making use of LUTs. As an addition or completely separate from this, the configuration bits can be sorted to default pathways. This can reduce the number of frames which have to be reconfigured. It can be looked upon as a combination of coarse and fine grain reconfiguration. The only way to properly implement this is if the placement and routing algorithms are adaptable, which probably involves building an proper version of them.

5.3.3 Routing

For partial reconfiguration it can be advantageous to have default routing structures preconfigured. With proper placement and a router that can make use of these structures, this could reduce the number of reconfiguration frames. It can even be possible that a good implementation only requires placement of a module to connect it up to a standard bus. The effects of transients on the routing during partial reconfiguration is unexplored. If communication through a signal pathway can be temporarily stalled this could open up the possibility of rerouting these structures while in use.

The current router implementation is not that good. As it was not our intention to build a fully featured, efficient router this will do for now. It is a good idea to make the software work with VPR [2], an academic router based on simulated annealing. This will undoubtedly be a faster implementation. Due to the way VPR is implemented, a change of a single wire will result in a completely different structure. The Xilinx router behaves in the same way, and may be based on VPR. A different behavior in which the synthesis is more consistent in the output it produces has its advantages. It can be made aware of previous and next configurations, it could be made aware of already existing, and future structures and even damaged portions in the FPGA matrix and route around these.

5.3.4 Network on Chip

The methods developed for this thesis are ideal for direct circuit-switched networks on chip. The development of standard routing buses could accelerate the routing problem to a level that is acceptable for on-line routing. It is up to the module designers to cope with the transients which are present during the reconfiguration of the new bus-structures. Logic circuit-switched networks can be made to cope with this phenomenon by default. Signals indicating a reconfiguration in process can be partially reconfigured themselves, thereby giving the ability to guide reconfiguration process through use of the bitstream only. Packet switched networks can be implemented as a combination of a circuit-switched network and a router module. None of these networks are mutually exclusive and there might be a way to combine them. Transitions between the various forms of networks may become a challenge. It would be possible, based on profiling or statistics gathered during runtime, to adjust the bus-width to cope with changing network traffic.

5.3.5 System Integration

The interaction between hardware and software has been described by the Molen platform. The device reconfiguration has been conceived as a single instruction for the processor. No specific implementation of the set instruction has been developed yet. As there are many mechanisms involved in reconfiguration we would like to suggest a complex interaction between the operating system and a caching reconfiguration processor. Decisions on where to place modules and the routing to connect them are best left to the operating system, much like memory management. These operations are computationally intensive and better left to the main processor, possibly augmented by hardware acceleration. The generation of the specific bitstreams are better left to a reconfiguration processor. This can also cache the various transitions so repetitive reconfiguration does not require recalculation every time they occur. This system in itself can become very complex, in the case of multi-core systems and multiple hardware core reconfigurations are performed at the same time.

Bibliography

[1] Atmel, *AT94KAL Series Field Programmable System Level Intergrated Circuit*, `http://www.atmel.com/dyn/resources/prod_documents/doc1138.pdf`, 2004.

[2] Vaughn Betz, *VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs* , `http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html`.

[3] Carsten Bieser, Martin Bahlinger, Matthias Heinz, Christian Stops, and Klas D. Mueller-Glaser, *A novel partial bitstream merging methodology accelerating Xilinx Vitex-II FPGA based RP system setup*, FPL, IEEE Circuits and Systems Society, 2006, pp. 701–704.

[4] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan, *A selfreconfiguring platform*, FPL, 2003, pp. 565–574.

[5] Christophe Bobda and Ali Ahmadinia, *Dynamic interconnection of reconfigurable modules on reconfigurable devices*, IEEE Des. Test **22** (2005), no. 5, 443–451.

[6] Stephen D. Brown, Robert Francis, Johnathan Rose, and Zvonko Vranesic, *Field-Programmable Gate Arrays*, Kluwer International Series in Engineering an d Computer Science, 1992.

[7] David Brownell and Greg Kroah-Hartman, *Hotplug*, `http://linux-hotplug.sourceforge.net/`.

[8] Cypress, *Cypress EX-USB FX2*, `http://www.cypress.com`.

[9] Computer Engineering dept. Delft University, *Delft Workbench*, `http://ce.et.tudelft.nl/DWB/`.

[10] H. ElGindy, A. K. Somani, H. Schroder, H. Schmeck, and A. Spray, *Rmb – a reconfigurable multiple bus network*, hpca **00** (1996), 108.

[11] Jeri Ellsworth, *C-one Reconfigurable Computer*, `http://c64upgra.de/c-one/`.

[12] Gerald Estrin, *Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer*, IEEE Annals of the History of Computing **24** (2002), no. 4, 3–9.

[13] J. Hadley and B. Hutchings, *Design Methodologies for Partially Reconfigured Systems*, IEEE Symposium on FPGAs for Custom Computing Machines (Los Alamitos, CA) (Peter Athanas and Kenneth L. Pocek, eds.), IEEE Computer Society Press, 1995, pp. 78–84.

[14] Reiner Hartenstein, *A decade of reconfigurable computing: A visionary retrospective*, DATE **00** (2001), 0642.

[15] C. Hilton and B. Nelson, *Pnoc: a flexible circuit-switched noc for fpga-based systems*, IEE Proceedings - Computers and Digital Techniques **153** (2006), no. 3, 181–188.

[16] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour, *Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration*, DAC '02: Proceedings of the 39th conference on Design automation (New York, NY, USA), ACM Press, 2002, pp. 343–348.

[17] Michael Hubner, Christian Schuck, Matthias Kuhnle, and Jurgen Becker, *New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Microelectronic Circuits*, ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (Washington, DC, USA), IEEE Computer Society, 2006, p. 97.

[18] Digilent inc., *The Xilinx XUP-V2Pro Board*, `http://www.xilinx.com/univ/xupv2p.html`.

[19] Jungo, *Windriver*, `http://www.jungo.com/windriver_usb_pci_driver_development_software.html`.

[20] Y.E. Karsteva, E. de la Torre, T. Riesgo, and Didier Joly, *Virtex II FPGA Bitstream Manipulation: Application to Reconfigurable Control Systems* , FPL, IEEE Circuits and Systems Society, 2006, pp. 717–720.

[21] Usama Malik and Oliver Diessel, *A configuration memory architecture for fast run-time-reconfiguration of fpgas.*, FPL, 2005, pp. 636–639.

[22] Usama Malik and Oliver Diessel, *The Entropy of FPGA Reconfiguration* , FPL, IEEE Circuits and Systems Society, 2006, pp. 261–266.

[23] Leonardo Möller, Ismael Grehs, Ney Calazans, and Fernando Moraes, *Reconfigurable Systems Enabled by Network-On-Chip* , FPL, IEEE Circuits and Systems Society, 2006, pp. 857–860.

[24] Timothy F. Oliver and Douglas L. Maskell, *Execution Objects for Dynamically Reconfigurable FPGA Systems* , FPL, IEEE Circuits and Systems Society, 2006, pp. 865–868.

[25] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, *Modular dynamic reconfiguration in virtex fpgas*, Computers and Digital Techniques, IEE Proceedings-**153** (2006), no. 3, 157–164.

[26] Piotr Stepien and Milan Vailko, *On Feasability of FPGA Bitstream Compression During Placement and Routing* , FPL, IEEE Circuits and Systems Society, 2006, pp. 749–752.

[27] J. van der Veen, S.P Fekete, M. Majer, A. Ahmadinia, C. Bobda, F. Hannig, and J. Teich, *Defragmenting the module layout of a partially reconfigurable devices*,

Proceedinggs 2005 International Conference on Engieering of Reconfigurable Systems and Algorithms (Toomas P. Plaks, ed.), CSREA Press, 2005, Available at http://arxiv.org/abs/cs.AR/0505005, pp. 92–101.

[28] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G.K. Kuzmanov, and E. Moscu Panainte, *The Molen Polymorphic Processor*, IEEE Transactions on Computers (2004), 1363– 1375.

[29] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, *Programmable Active Memories: the Coming of Age*, IEEE Trans. on VLSI,Vol. 4, NO. 1, 1996, pp. 56–69.

[30] Xilinx, *Xilinx ISE*, `http://www.xilinx.com/products/design_resources/design_tool/`.

[31] ______, *Xilinx JBITS*, `http://www.xilinx.com/products/jbits/`.

[32] ______, *Xilinx Linux Drivers*, `ftp://ftp.xilinx.com/pub/utilities/M1_workstation/linuxdrivers.2.6.tar.gz`.

[33] ______, *Two Flows for Partial Reconfiguration: Module Based of Difference Based*, `http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf`, 2004.

[34] ______, *Virtex II Pro and Virtex II X FPGA User Guide*, `http://direct.xilinx.com/bvdocs/userguides/ug012.pdf`, 2004.

[35] ______, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data sheet*, `http://www.xilinx.com/bvdocs/publications/ds083.pdf`, 2005.

I have started this project with the attitude that it is finished when I have something proper to show for my efforts. It has therefore taken longer and involved a little more work. This also has to do with the fact that this is research, not an internship. Apart from that, I am easily distracted to do other things, and was often asked to perform tasks which where not related to my subject. The Xilinx tools proved to be a total nightmare. Each system upgrade presented new problems and I was constantly consulted on getting the USB-interface and drivers up and running. This became so commonplace that the system administrator granted me *root access*. The work has translated into a poster/paper for ProRISC 2006, a as of yet unpublished article for the Maxwell and a poster/publication for FPL 2007. Apart from this I've organised some social events, namely a karting event and borrel middagen. For SARC and ICS I have set up small and large teleconferencing systems used for paper-reviews. This included a streaming web-cast with a web-cam and a separate stream for what was on the computer screen and beamer. For the SAMOS conference I have set up and maintained a mailing list of over 22.000 possible participants, over a third have been harvested by myself. This included a script used for bulk-mailing the list. I was also involved in administrating the group website, having set up the calendar function of the social pages and keeping the internship pages up to date. I also sometimes helped with the occasional Linux problems people had. As a side project I've made an RF-ID transmitter, needed for another project. Aside from involvement with the department, during this project I wanted get my drivers license and I have spend a year on the governing board of O.J.V. de Koornbeurs as the secretary. I was determined to do these things before I graduated. As a hobby I also participated in the UK championships and other Robotwars events with a 2 person team, competing with a feather and a heavy weight robot.