FPGA virtualization layer for non-deterministic state machines

Tormod Heimark

Master of Science in Electronics
Submission date: June 2015
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Summary

In this thesis a virtual layer for running self-cloning state machines on FPGAs has been developed. The goal has been to connect software with hardware resources, and to make partial reconfigurability more available. Previous work has been done on defining self-cloning state machines that can run on an FPGA, but was not tested with partial runtime reconfiguration. A framework for reconfiguration has been used in this thesis, which had previous shown some difficulties regarding synchronous designs.

Specifications for the virtual layer were defined, and the different modules constructed. The virtual layer was implemented on a Virtex-4 FPGA, with an embedded PorwerPC microprocessor running a Linux operating system. The virtual layer gives software application an interface for defining state machines, which will be mapped to the FPGA and executed. The modules on the FPGA are separated into two parts, one reconfigurable region and one static region. The static region contains a back-end that handles the control of the NFSM and communication with the processor. The reconfigurable region contains the NFSM, which is divided into several clones. The clones can be inserted or removed by using partial runtime reconfiguration.

Many difficulties were experienced when trying to implement the virtual layer with support for partial runtime reconfiguration. The tool support was lacking and the space on the FPGA became a problem. Only one clone could be fitted on the FPGA. Therefore the verification of the system was divided in two. A state machine with four clones was tested and verified. The virtual layer was able to take input from software and map this into a functional self-cloning state machine. Some limitations had to be put on the system to make it possible to implement. A second test was performed with partial runtime reconfiguration to show that clones could be added or removed from the design at runtime. The test was successful, but could only be done with one active clone. The limitations of the Virtex-4 platform can be avoided by implementing the virtual layer on a more state of the art FPGA. The system defined in this thesis should work on any FPGA, but will require a lot of work, especially porting of the framework for reconfiguration.

Sammendrag

I denne oppgaven har et virtuelt lag for selv-kopierende tilstandsmaskiner blitt utviklet. Målet med oppgaven har vært å knytte software sammen med ressurser på hardware, samt å gjøre partiell rekonfigurering enklere å bruke. Tidligere arbeid har blitt gjort med å definere og teste selv-kopierende tilstands maskiner, men dette ble aldri testet på et fullt system med rekonfigurering. Et rammeverk for rekonfigurering har blitt brukt i denne oppgaven, tidligere har det vært problemer å bruke dette på synkrone design.

Spesifikasjoner ble satt opp for det virtuelle laget, og de forskjellige modulene som trengtes ble konstruert. Dette ble implementert på en Virtex-4 FPGA, som har en PowerPC microprosessor med et Linux operativsystem innebygd. Det virtuelle laget tilbyr et grensesnitt for å definere tilstandsmaskiner i software, disse definisjonene blir så lastet ned på FPGAen og realisert som en kjørende tilstandsmaskin. FPGAen er delt inn i to deler, en statisk del, og en rekonfigurerbar del. Den statiske delen består av en back-end som håndterer kontrollen av tilstandsmaskinene som kjører på FPGAen, og den håndterer kommunikasjonen mellom FPGAen og prosessoren. Det rekonfigurerbare delen inneholder tilstandsmaskinene, som er delt opp i forskjellige kloner. Klonene kan bli satt inn eller fjernet ved hjelp av partiell rekonfigurering ved kjøretid.

Det var en del vanskeligheter når systemet skulle implementeres, og spesielt når det gjaldt den partielle rekonfigureringen. De tilgjengelige verktøy var vanskelige å bruke, og hadde en del mangler. Det viste seg også at plass på FPGAen var et problem. Det var bare plass til en klone om gangen, hvis støtte for partiell rekonfigurering ble lagt til. Derfor ble testingen av systemet delt i to. Den første delen testet en hel tilstandsmaskin men fire tilkoblede kloner. Testingen viste at det virtuelle laget kunne ta en definisjon i fra software og gjøre dette om til en kjørende selv-kopierende tilstandsmaskin. For å gjøre det mulig å implementere dette systemet måtte det settes en del begrensninger, spesielt for hvilke typer tilstandsmaskiner som kunne kjøres. Den andre testen ble utført med partiell rekonfigurering av en klone. Dette fungerte og klonen kunne bli byttet ut. Begrensingene på Virtex-4 plattformen gjorde at et større system med rekonfigurering ikke kunne bli gjennomført, men det virtuelle laget skal kunne fungere på nyere FPGAer også. Å implementere det virtuelle laget på andre FPGAer vil kreve en del arbeid, spesielt med tanke på rammeverket for rekonfigurasjon, som akkurat nå er bare laget for Virtex-4.

Preface

Making a system that runs on both hardware and software is very interesting, and is the reason for why I chose to work with this in my thesis. It has been challenging but I have learned a lot. I want to thank Professor Kjetil Svarstad for the help and advice he has given me.

Table of Contents

Summary i

Sammendrag iii

Preface iv

Table of Contents vii

List of Tables ix

List of Figures xi

List of Source code listings xiii

Abbreviations xiv

1 Introduction **1**

1.1 Problem description 1

1.2 Motivation 1

1.3 Contribution 2

1.4 Method 2

1.5 Report Structure 3

2 Previous work **5**

2.1 Previous work from NTNU 5

2.1.1 Daniel Blomkvist's master thesis (June 2013): Self-cloning state machines on FPGA 5

2.1.2 Sverre Hamre' master thesis (June 2009): Framework for self reconfigurable system on Xilinx FPGA 5

2.1.3 Vegard Endresen's master thesis (June 2010): Hardware-software intercommunication in reconfigurable systems 6

2.1.4 Sindre Hansens's master thesis (June 2011): Self Reconfiguration of Clock Networks on FPGA . . . 6
2.2 Other sources . . . 7
2.2.1 A heterogeneous multicore system . . . 7
2.2.2 A homogeneous network on-chip system . . . 7

3 Theory and background **9**
3.1 Development environment . . . 9
3.2 Field programmable gate array . . . 11
3.3 The Virtex-4 . . . 12
3.4 Software . . . 12
3.5 Runtime reconfiguration . . . 13
3.5.1 Bus macros . . . 14
3.6 Framework for reconfiguration . . . 14
3.6.1 CLBRead . . . 14
3.6.2 icap_write . . . 15
3.7 Finite state machine . . . 15
3.7.1 Cloning . . . 16

4 Design of the virtual layer **21**
4.1 Requirements . . . 21
4.2 System overview . . . 21
4.2.1 Reconfigurable module . . . 25
4.2.2 Back-end . . . 27
4.2.3 Software . . . 29
4.2.4 Datapath . . . 29
4.3 Limitations . . . 30
4.4 Implementation alternatives . . . 31
4.4.1 A Self-Reconfigurable Gate Array Architecture . . . 31
4.4.2 Xilinx development flow . . . 31

5 Implementation and verification **33**
5.1 Implementing the virtual system . . . 33
5.1.1 Timing issues . . . 34
5.2 Runtime reconfiguration . . . 34
5.2.1 Restriction of logic placement . . . 34
5.2.2 FPGA Editor . . . 35
5.2.3 Clock signal . . . 37
5.3 Drivers . . . 38
5.4 Program . . . 39
5.5 Verification . . . 40
5.5.1 Test with substring detector . . . 40
5.5.2 Partial runtime reconfiguration test . . . 41

6 Use of the virtual layer **43**
6.1 Configuration vectors 43
6.2 Creating the input file 43
6.3 Running the state machine 44
6.4 Reconfiguring with the HWICAP module 44

7 Results **47**
7.1 First test case with four connected clones 47
7.2 Second test case with one clone and partial runtime reconfiguration 49
7.2.1 Timing 51

8 Discussion **53**
8.1 The virtual layer 53
8.2 Partial runtime reconfiguration on Virtex-4 54

9 Conclusion **57**
9.1 Future work 57

Bibliography **59**

Appendix **61**
A Useful commands 61
B Example of input text file used for configuring and running the NFSM 62
C runNFSM source code 63
D Synthesis results 66
E Development computer 67

List of Tables

3.1 Specifications of the Suzaku-V board 11
3.2 Specifications of the Virtex-4 FPGA 12

4.1 Input bits 28
4.2 Output bits 28

6.1 Configuration vectors for the substring detector 44

List of Figures

3.1 The Suzaku-V developement board 10
3.2 The Suzaku-V components 10
3.3 Architecture of a typical FPGA 12
3.4 Organization of the different blocks on the FPGA, taken from Hamre (2009) 13
3.5 Drawing showing the bus macros with legal and illegal routing. Taken from Hansen (2011) 15
3.6 Example of a deteministic finite state machine 16
3.7 Example of a non-deterministic finite state machine 16
3.8 State machine for detecting substrings of three consecutive ones, based on state machine from Svarstad and Volden (2011) 17
3.9 Two active clones 17
3.10 Three active clones 18
3.11 Four active clones, accepted input 19

4.1 Overview of the system 22
4.2 Structure of the virtual layer 23
4.3 Connection of clones 24
4.4 Internal components of the clone 25
4.5 Example structure of a configuration vector 26

5.1 Modules divided into pblocks in *PlanAhead* 35
5.2 Example of illegal routing crossing the bus macro CLB columns 36
5.3 Example of routing in the I/O column 36
5.4 Example of a completed rerouting process with only legal wires 37
5.5 Global clock routing 38
5.6 Closer look of the clock signal 39

Listings

5.1 Changes made to the HWICAP driver. 39
5.2 Changes made to the *icap_write* program. 39
6.1 Installing the driver and running the state machine 44
6.2 Extration of a partial bitfile 45
6.3 Reconfiguring CLB columns 45
7.1 Output from testing substring detector 47
7.2 Partial runtime reconfiguration output 49

Abbreviations

FPGA	=	Field-programmable gate array
NFSM	=	Nondeterministic finite state machine
DFSM	=	Deterministic finite state machine
CLB	=	Configurable logic block
LUT	=	Lookup-table
NFS	=	Network file system

Chapter 1

Introduction

1.1 Problem description

Previous work on implementing self-cloning state machines on FPGAs has been done in Blomkvist (2013). This thesis will continue on that work by implementing a system for cloning the state machines using partial runtime reconfiguration, and defining a virtual layer that will combine both hardware and software resources. The goal of this thesis is to implement a system that can make the speed and parallel features of the FPGA more available to applications running on software.

1.2 Motivation

The ever shrinking transistor size comes with an increase in complexity. In the recent years there has been a struggle to keep up with this increasing complexity, and efforts to increase the productivity of designs have been made. One solution has been to use reconfigurable hardware to bridge the gap between hardware and software. Many systems today use FPGA modules as hardware accelerators, often able to perform certain tasks much faster than a processor, and much cheaper to develop than ASIC designs. The FPGA offers the flexibility and reconfigurability of software, and the fast and parallel features of hardware. Partial runtime reconfiguration, where only parts of the FPGA is reconfigured at a given time, extends the flexibility and makes it possible to adapt at runtime. The drawback is that the availability and use of these features vary from manufacturers, and that adding partial runtime reconfiguration to a system requires detailed knowledge of the device. The support from development tools have been scarce, leading to a much more complicated implementation process and possible risk of device failure.

Using self-cloning state machines as a way to make partial runtime reconfiguration easier to use and more available was first introduced by Svarstad and Volden (2011). State machines can be used to solve many computational problems, for example applications that are centered on pattern matching. If they can be implemented on FPGAs, they can be

used to execute certain tasks for software, saving time and resources. Self-cloning state machines will clone/copy themselves whenever they encounter problems that can have two or more possible outcomes. They cannot know which outcome that is correct, and have to clone several state machines to evaluate all outcomes. When the correct outcome becomes clear, all the other state machines can be removed. The cloning aspect makes them ideal to fully make use of the partial runtime reconfiguration features on FPGAs. A more thorough introduction to the self-cloning state machine will be given in the *Theory and background* chapter. By using state machines one can create an interface for the applications, where the resources on the FPGA can be used without knowing any specific details about the hardware. The idea behind the virtual layer is to create a system that can handle job requests from applications in software, and make sure they are executed correctly in hardware, all while hiding factors like implementation and resource usage. This layer will create the same interface no matter what device it is implemented on. This will help to connect software and hardware even more, and remove some of the drawbacks of using FPGA mentioned earlier. The virtual layer will even make it possible for defining systems that exceeds the actual hardware capacity, just like virtual memory.

1.3 Contribution

The work in this thesis will mostly be based on combining and further developing the previous work done on the self-cloning state machines and partial runtime reconfiguration on FPGAs. The contributions in this thesis are:

- Defining a system that can perform partial runtime reconfiguration of the self-cloning state machines.
- Defining a virtual layer with an interface that makes the self-cloning state machines available for the applications running on software.
- Implementation of these systems on the Suzaku-V platform. Simple testing and experiments to verify the systems.

1.4 Method

As mentioned earlier the tool support for implementing partial runtime reconfiguration is lacking. Previous work on the subject has offered some helpful guidelines and tutorials, but will still require a great deal of insight and knowledge to use. The Suzaku-V system includes an FPGA and a microprocessor with its own operating system. The number of different components needed and the level of complexity is high, and a lot of time has been spent on setting up all the parts required to run the system. This includes all the hardware modules, software programs, drivers, and the development tools. To correctly implement and use all these parts requires a good understanding of how they work. Implementing the partial reconfiguration is the most challenging, since a lot of intricate designs steps are needed. Without a very good understanding of the reconfiguration process, the FPGA can easily be damaged.

Most of the work in this thesis has been done with an experimental approach. Some of the design decisions made have been educated guesses, since it has been difficult to foresee any effects before proper testing.

1.5 Report Structure

The next chapters are divided into *Previous work*, *Theory and Background*, *Design of the virtual layer*, *Implementation and verification*, *Use of the virtual layer*, and *Results*. The first chapter presents some of the earlier work done on this subject. The next chapter introduces some theory and background information useful for understating the scope of this thesis better. The presentation of the work done in this thesis starts with the chapter *Design of the virtual layer*, were details of the design are given. In the next chapter a closer look at the implementation and verification of the system is presented. A guide to use the finished virtual layer on the Suzaku-V platform is given in the following chapter. Results of the verification process are presented in the *Results* chapter.

Chapter 2

Previous work

This chapter introduces some of the previous work done on this subject. The work that is studied consists of three different subjects, running self-cloning state machines on FPGAs, partial runtime reconfiguration on FPGAs and implementing virtual systems on FPGAs.

2.1 Previous work from NTNU

A natural place to start is to investigate the work that has been done on NTNU. This thesis is a continuation on the work done on a previous thesis from NTNU. Also there has been some work done on partial runtime reconfiguration on NTNU.

2.1.1 Daniel Blomkvist's master thesis (June 2013): Self-cloning state machines on FPGA

In the master thesis, Blomkvist (2013), a single state representation for self-cloning state machines was developed. Several single states was connected together and used to represent non-deterministic state machines on a Virtex-4 FPGA. These state machines were set to copy themselves when multiple transitions from the same input occurred. Much of the work was focused on making the single state representation generic, so that it could represent any state in any state machine. The single state contained registers that were used for configuring the state, these configuration vectors together with the input was used to determine if the state should be active. A datapath was connected to the state machine to perform a multiply-accumulate operations of the FPGA, using the state machine to control the flow of operations.

2.1.2 Sverre Hamre' master thesis (June 2009): Framework for self reconfigurable system on Xilinx FPGA

Sverre Hamre has in his master thesis, Hamre (2009), developed a framework for partial runtime reconfiguration on FPGAs. The framework was set up on a Suzaku-V board,

which utilizes either a Virtex-2 or Virtex-4 FPGA. The FPGA's include a PowerPC microprocessor that was set up with a Linux operating system. The main contributions in this thesis are the two programs *icap_write* and *CLBRead*. Also included is a general method for implementation, with the use of bus macros, to ensure that partial runtime reconfiguration can be executed properly. The program *CLBRead* can be used to extract partial FPGA designs from the bitfiles generated from the standard tools of FPGA development. The program *icap_write* can be used on the FPGA at runtime, to read or write to the FPGA's configuration memory. Some work was also put in defining a hardware operating system (HWOS) that can be used alongside with the Linux operating system, and offer better control over the hardware resources.

2.1.3 Vegard Endresen's master thesis (June 2010): Hardware-software intercommunication in reconfigurable systems

Vegard Endresen has in his master thesis, Endresen (2010b), worked to further develop the framework for partial reconfiguration. He developed a system for exchange of data between software and reconfigurable modules on the FPGA. He made both hardware and software components, with the focus of fast communication between them. A back-end module was put on the FPGA to facilitate the operation of the reconfigurable modules, and also to support loading and saving of the processes running on the reconfigurable modules. This last feature was added so that interrupts could be supported. A HWOS was implemented, running in the background of the Linux system. All user application requests from running processes on the hardware goes through the HWOS, which was fitted with a scheduler to facilitate a fair distribution. The support for interruption means that a process that is not finished after using up the time slot, can be stopped and saved in memory, giving time to other processes.

2.1.4 Sindre Hansens's master thesis (June 2011): Self Reconfiguration of Clock Networks on FPGA

Sindre Hansen has in his master thesis, Hansen (2011), studied partial reconfiguration of synchronized modules and clock networks. This was found to be problematic in Endresen (2010b), where partial reconfiguration of synchronized modules failed. The details of the local and global clock networks on the Virtex-4 FPGA were studied. It was found that clock signals could not be routed through the bus macros, and that clock signals had their own separate network. It was found that the existing framework for reconfiguration could be used to reconfigure these clock networks, the only thing that has to be ensured when reconfiguring synchronized modules is that the clock signal has to enter the reconfigurable modules at the exact same spot for all designs. Efforts were also made to integrate the software hardware communication interface from Endresen (2010b) with clocked modules. The HWOS was completely rewritten, including a new round-robin scheduler. Much focus was put on making a robust system. However when trying to reconfigure some of the larger synchronous designs with this system, the tests failed and the FPGA locked up.

2.2 Other sources

Other than the work done in Blomkvist (2013), there has not been much work done on implementing NFSMs on FPGAs. The little that exists does not use the self-cloning approach. When it comes to implementing a virtual layer on FPGAs some work can be found. These implementations are not made specifically for NFSMs, but rather for a more general use. Still the implementation ideas could be useful when defining a system in this thesis.

2.2.1 A heterogeneous multicore system

The work done in Hubner et al. (2011) introduces a virtual FPGA layer that separates the FPGA into several heterogeneous virtual cores. These cores can be configured in different sizes. The virtual cores are made up of logic cells within the physical FPGA. They can run any application that is described in a hardware descriptive language. Together with these cores is a built in microprocessor in the FPGA connected with an AMBA bus. The virtual cores offer the possibility of partial reconfigurability even for FPGA families that does not originally support this feature. The applications that can run on this design will work independently of the FPGA hardware the virtual layer is made of. In the FPGA there is a configuration controller that is responsible for configuring the virtual cores. The controller is connected to the cores with a configuration bus. The configuration inside the virtual cores is realized by using configuration units, containing shift registers.

2.2.2 A homogeneous network on-chip system

The paper Yang et al. (2010) talks about how to realize a true virtual FPGA. It stating by saying that the concept of virtual FPGA is the same as virtual memory, but that it is not that widespread because that virtual FPGA is much harder to accomplish than virtual memory. One of the reasons is that memory is place independent and hardware is not. A unit needs to communicate with other units, but the routing between them is place dependent. The paper states that some critical requirements need to be fulfilled for the system to be real virtual FPGA. The first is partial reconfigurability, this enables the system to reconfigure some part of the system while the other is running, so that task that are completed or that is not running anymore can be switched out with other task. The second requirement is removing circuit location dependency and I/O independency. The system proposed is made up of several homogeneous configurable regions (CRs), and a network on-chip (NOC) that can connect all the CRs together in an efficient manner. The NoC also connects I/O blocks to the other units in the system. The CRs are large enough to accommodate most single modules. The paper uses paging partitioning, which divides CRs into block of fixed size, compared to segmentation where the blocks have variable sizes. The system also has a configuration module responsible of deciding when to swap configurations in and out of the CRs, and also loading these configurations onto the CRs.

Chapter 3

Theory and background

This chapter presents some background information on the development environment used, and also some theoretical information on certain key subjects for this thesis.

3.1 Development environment

The development board Suzaku-V form Atmark-Techno has been used to implement and test all systems in this thesis. The board has a Xilinx Virtex-4 FPGA, which is also fitted with a PowerPC microprocessor. The board comes with a functional Linux operating system already installed, based on μClinux. An image of the OS can also be downloaded from the Atmark-Techno (2015) download page, if a custom system is wanted. Atmark-Techno also supplies an FPGA project download that includes all the basic modules needed for running the OS and the FPGA together. Custom user modules can then be added to this base design. Specifications of the board and FPGA are listed in Table 3.1. A block diagram of the Suzaku-V is depicted in Figure 3.2, the FPGA is a different version of the one used in this thesis but the components are the same. The actual board is shown in Figure 3.1. The board is easy to use and requires minimal time to set up. The OS on the board does not have any compiler support, so any program or driver must be compiled on a cross development environment and then added. This can be done in two ways, the first requires the compiled source code to be added to the OS image before it is downloaded. The second option uses a network file system (NFS), and makes it possible to update drivers and user programs in a folder on the same network. The drawback of using the Suzaku-V board is that the FPGA has become slightly outdated, with a lack of space compared to newer versions. The tools used for development are all form the ISE Design Suite, made by Xilinx. An overview of the programs used and their purpose is shown below.

- **ISE:** Used for developing VHDL design files, verifying and testing done with test-benches and waveforms.
- **EDK:** Used for setting up the implemented system with the rest of the components on the FPGA, synthesizing, and place and route for the Virtex-4.

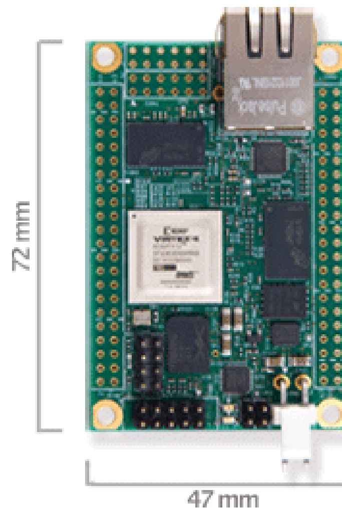

Figure 3.1: The Suzaku-V developement board

- **PlanAhead:** Used for grouping and restricting placement of logic on the FPGA.
- **FPGA Editor:** Used for inspecting and rerouting the designs from EDK.
- **bitinint** Used for initializing the completed bitfiles before downloading them to the FPGA.
- **bitgen** Used to generate bitfiles from custom edited place and route files.

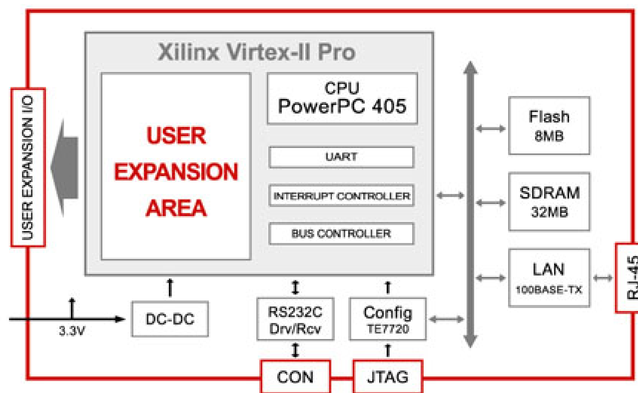

Figure 3.2: The Suzaku-V components

The version of the ISE Designs Suite used in this thesis was both the 11.5 and the 10.1. The reason for this is that 11.5 was used at first, but it was later discovered that some

parts of the reconfiguration framework did not fit with this version, and 10.1 was used instead. The newest version of ISE Design Suite is 14.7, but the FPGA project files given by Atmark-Techno are only updated to the 11.5 version. The different versions are not very compatible with each other and in most cases the programs refuse to open projects that are not specifically made for that version. The advantage of using newer versions is that many modules have been updated to work better, and to use less resources. Updating all the design files to the newest version is probably possible, but will require a lot of work.

Model	SZ410-U00
FPGA-device	Xilinx Virtex-4 FX XC4VFX12-SF363
FPGA CPU	PowerPC 405 (32bit RISC core)
CPU Clock	350MHz
Crystal Oscillator	100MHz
DRAM	32MB
Flash Memory	8MB (SPI)
Ethernet	10BASE-T/100BASE-TX
Linux version	2.6.18-at11
User I/O Pins	86
Serial Port	1ch (RS232C)
Configuration	SPI Flash
Board Size	72x47 [mm]
Power Input	DC3.3V

Table 3.1: Specifications of the Suzaku-V board

3.2 Field programmable gate array

A field programmable gate array (FPGA) is an electronic circuit that can be configured after manufacturing, and that can be reconfigured multiple times. The positive effects of this are many, for example the possibility for fixing design problems and bugs in the field. Compared to an application-specific integrated circuit (ASIC), that cannot be reconfigured after production, an FPGA is much more flexible. Production costs are unusually much higher on ASIC designs, so on low volume productions, FPGAs are often the better choice. However the FPGA suffers in speed and resource consumption compared to ASIC designs. Processors and microprocessors can also offer flexibility, but lacks in speed and parallel capabilities compared to the two other choices.

The internal architecture of the FPGA depends on the manufacturer, but will generally consist of several configurable logic blocks (CLBs), and ways to connect them together. The CLBs are usually made up of several lookup tables (LUTs), which will do the actual computations in the FPGA. They can be configured to model any n-input boolean function. Other blocks like I/O , RAM and DSPs can in many cases also be found on the FPGA.

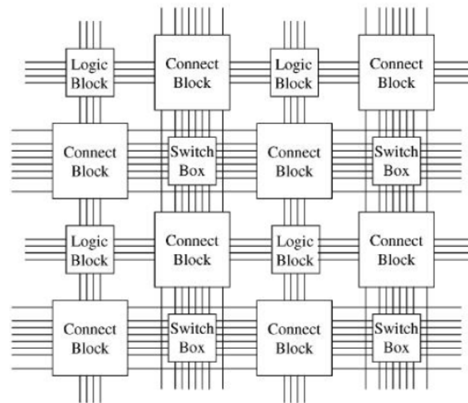

Figure 3.3: Architecture of a typical FPGA

3.3 The Virtex-4

The FPGA on the Suzaku-V is a Virtex-4. The internal specifications of the FPGA is listed in Table 3.2. The CLBs are distributed in an array of 64x24, with four major rows 16 CLBs in height. Each CLB has four slices, which again contains 2 LUTs and 2 flip-flops. Some of the rows and columns are used by the processors, so the total number of available LUTs is 10944. The FPGA is also fitted with BRAM, DSP and I/O blocks, placed in between the CLB blocks. The FPGA blocks are depicted in Figure 3.4. The I/O columns are marked in red, the BRAM in blue and the DPS in green. The rest of the columns are CLB columns, which are put in sets of four. The black box is the microprocessor placed on the FPGA.

Model	Virtex-4 FX XC4VFX12-SF363
Speed grade	-10
CLBs (Rows x Cols)	64x24
Slices	5472
LUTs	10944
Flip-flops	10944

Table 3.2: Specifications of the Virtex-4 FPGA

3.4 Software

As mentioned earlier some changes had to be made to the default OS running on the Suzaku board. Module support for the kernel had to be enabled, so that drivers could be installed from the NFS. To enable NFS, the device was set up with a static IP address, some network features were removed, and a NFS server was set up on a host computer. A detailed tutorial for setting up the Linux distribution and the NFS can be found in the appendix in Hansen (2011). Device drivers were used to connect hardware with software

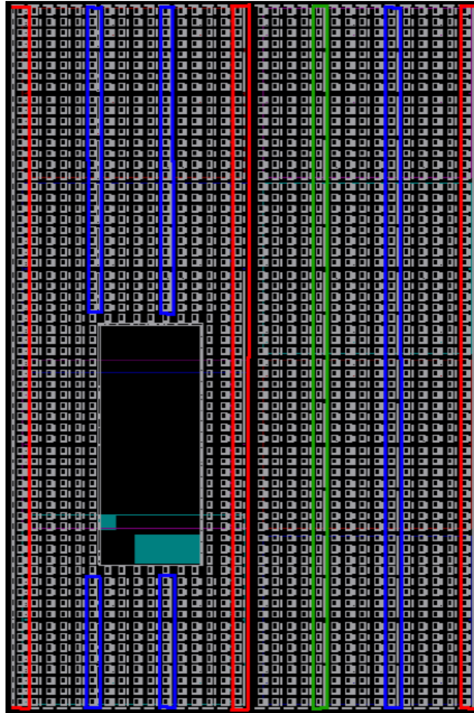

Figure 3.4: Organization of the different blocks on the FPGA, taken from Hamre (2009)

applications. The hardware modules implemented on the FPGA cannot directly be accessed by user applications, so device drivers can be used to facilitate the access for user applications. Two drivers were used in this thesis, the *sw_access_reg* driver used to access a software accessible register, written by Hansen (2011), and a *xilinx_hwicap* driver used to access the HWICAP module described in the next section.

3.5 Runtime reconfiguration

Typically an FPGA device is configured before it is booted, but in some cases it can be useful to reconfigure the FPGA while it is running. For example if a different communication protocol is needed, and a complete reboot of the device is not possible. Any system that can be optimized at runtime can benefit from using runtime reconfiguration. The Virtex-4 used in this thesis supports runtime reconfiguration. Reconfiguring only a part of the FPGA, while the rest is still running is called partial runtime reconfiguration. The reconfiguration is done by writing changes to the configuration memory on the FPGA. This memory is divided into several frames, a frame is the smallest unit that can be reconfigured. On the Virtex-4 a frame consists one column in a major row, a total of 16 CLBs. There are also frames for the other blocks, but they will not be reconfigured in this thesis.

The configuration memory can be reached by using the internal configuration access

port (ICAP) on the FPGA, which can be used for both writing and reading. To access the ICAP interface from software, Xilinx has made a HWICAP module that can be placed on the FPGA. The module uses the PLB bus and is connected to the processor. A driver for this module is also available, with a few modifications described in Hamre (2009), it can be used to write or read content in the configuration memory. Configuration of the FPGA can be stored in a bitfile, and this bitfile can written to the configuration memory. A partial bitfile can be written to the configuration memory to perform a partial reconfiguration.

3.5.1 Bus macros

Executing partial runtime reconfiguration on FPGAs is not a straight forward process. The problem lies in the connection between the part that is reconfigured, and the rest of the FPGA. If no considerations are taken, the routing in and out of the reconfigurable area will not be the same after reconfiguration, and some wires will be unconnected. This will leave the FPGA in an unusable state, and can possibly damage the hardware. What is required is a permanent connection between the static parts of the FPGA and the areas to be reconfigured. This can be solved by placing bus macros between the static and the reconfigurable area. The bus macros are hard macros, they have already been routed, and can be placed on CLB columns connecting the static and reconfigurable area. This routing will not change during the reconfiguration, and serve as a static connection. As long as there is no other routing entering or leaving the reconfigurable area, the partial reconfiguration will succeed. An example showing bus macros connecting the static part of the FPGA with the reconfigurable part, and examples of legal and illegal routing is depicted in Figure 3.5. The drawback by using bus macros is that the place and size of any reconfigurable area must be known at design time, and the number of connections in and out cannot be changed at runtime. This requires that the maximum number of connections for any module placed in the reconfigurable area must be known before the design is downloaded to the FPGA. The bus macros used in this thesis are taken from Hamre (2009), and has a width of 8 bits. More bus macros can be added if 8 bits is not enough. They can only be used in one direction, so there must be at least one set for input and output.

3.6 Framework for reconfiguration

A framework for partial runtime reconfiguration has been developed for the Suzaku-V and the Virtex-4. This includes the programs *CLBRead* and *icap_write*, bus macros and drivers for both user modules and the HWICAP module.

3.6.1 CLBRead

This program can be used to read out one or more CLB columns from a bitfile. The program handles only the CLB blocks, and not the BRAM, DSP and I/O blocks. The CLB(s) will be placed in a separate partial bitfile. The program can also insert this partial bitfile into another bitfile, replacing the corresponding frames. In this project only the first functionality was used.

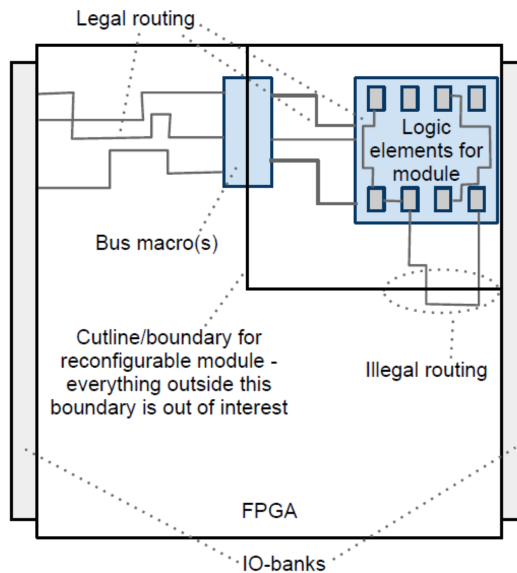

Figure 3.5: Drawing showing the bus macros with legal and illegal routing. Taken from Hansen (2011)

3.6.2 icap_write

This is a program that is used to write bitfiles partial to the configuration memory via the HWICAP module. The driver for the HWICAP module will only read and write packets to the configuration memory, and does not facilitate the required setup that has to be done before a partial reconfiguration. This is handled by the program. It takes a partial bitfile as input, the start and end columns, and the total numbers of frames in the partial bitfile. In a bitfile, it takes 22 frames to describe one CLB frame in the configuration memory. So a partial reconfiguration of CLB columns 21 to 23, would require 66 frames in the bitfile. This is a mostly experimental program, and the addresses are hard coded.

3.7 Finite state machine

A finite state machine (FSM) is an abstract machine that can be used for many purposes. The state machine consists of a finite number of states, and can only be in one of these states at a time. This state is called the current or active state. The state machine receives an input, and will transition to new states based on this input. In Figure 3.6 a simple state machine is shown. The state machine has three states, with one start state and one accept state. If the state machine reaches the accept state, the state machine will accept the input. In the example the input *ab* is needed for the state machine to accept the input. This state machine is called a deterministic finite state machine (DFSM), because for any state there is only one transition for any allowed input.

An example where this is not the case is shown in Figure 3.7. In the second state there

is two transitions for the input *b*, one leading to the accept state and one leading back to the same state. This is called a non-deterministic finite state machine (NFSM). As shown in the example an NFSM can have several transitions to other states given the same input. An NFSM can solve the same problems as a DFSM, and it is possible to convert an NFSM to a DFSM, but creating NFSMs for certain tasks can sometimes be easier. The self-cloning state machines used in this thessi will be NFSMs.

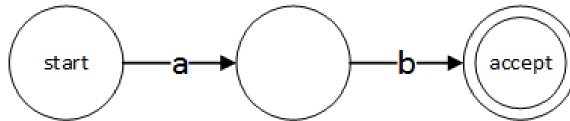

Figure 3.6: Example of a deteministic finite state machine

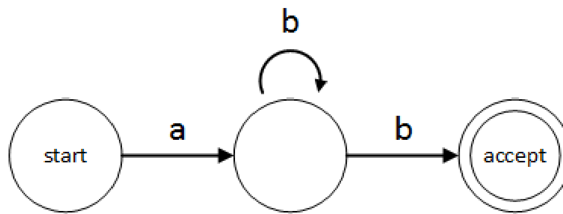

Figure 3.7: Example of a non-deterministic finite state machine

3.7.1 Cloning

When running NFSMs on FPGAs, it can be challenging to handle multiple transitions from one state. When such a transition occurs the NFSM will be in multiple states at once. In this project these multiple active states will be treated like multiple state machines running in parallel. That means that for every multiple transition encountered, the state machine has to copy or clone itself into one or several more state machine. For later reference such a transition will be called a copy transition, and the new state machines will be called clones. A state machine is shown in Figure 3.8. This state machine was used in Svarstad and Volden (2011) to show how the self-cloning state machine worked. It has four states, where the start state has a copy transition for the input '1'. This state machine will accept substrings of three consecutive ones. The copy transition is what makes the NFSM able to accept all substrings, and not just the first. The NFSM starts by having one state machine in the first state. In Figure 3.9 an input of '1' is sent to the state machine. This triggers the cloning of the state machine, since there has to be two active states. The initial state machine goes back to the initial state, while the new clone transitions to the second state. The NFSM now has two active clones. In Figure 3.10, a second '1' is sent to the NFSM. The initial clone triggers another copy transition, and a new state machine is cloned. The second clone activates its third state. There are now three clones active. When the third '1' is sent to the NFSM, in Figure 3.11, the second clone reaches the accept state, and a fourth

clone is added. The NFSM has accepted the input of three consecutive ones. If another '1' is sent to the input, the third clone will accept this substring. Since the second clone has reached the accept state, it can be removed. At the same time a new clone is also added, keeping the active number of clone at four as long as the input is never a '0'. If a '0' is sent to the input, all clones except the initial clone will encounter an illegal input, and can be removed. This state machine will be used later in this thesis to verify the cloning process on the FPGA. The state machine will be referred to as the substring detector.

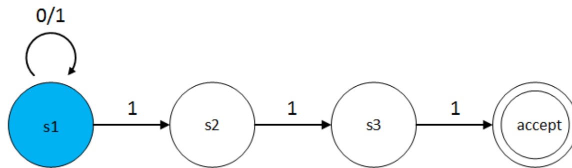

Figure 3.8: State machine for detecting substrings of three consecutive ones, based on state machine from Svarstad and Volden (2011)

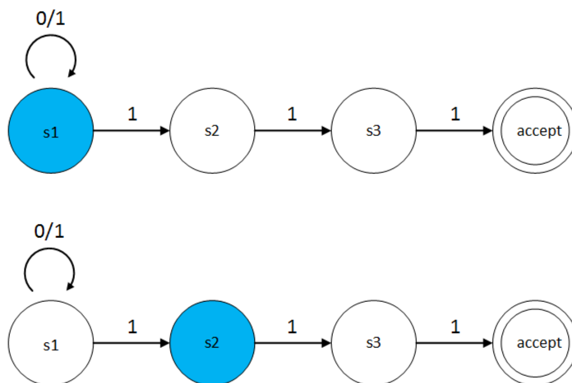

Figure 3.9: Two active clones

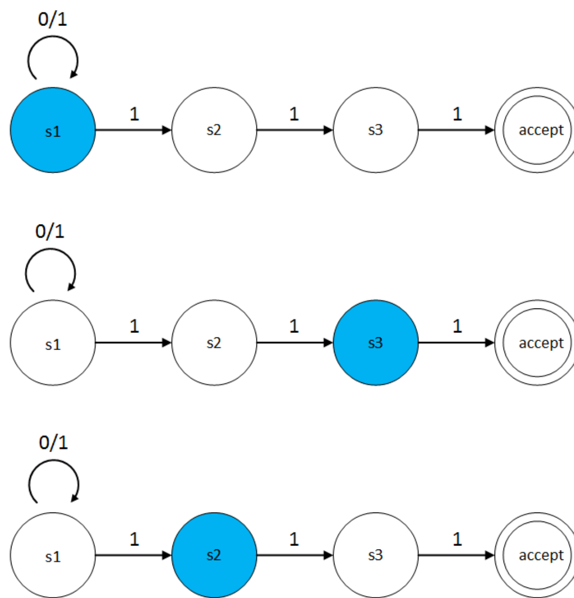

Figure 3.10: Three active clones

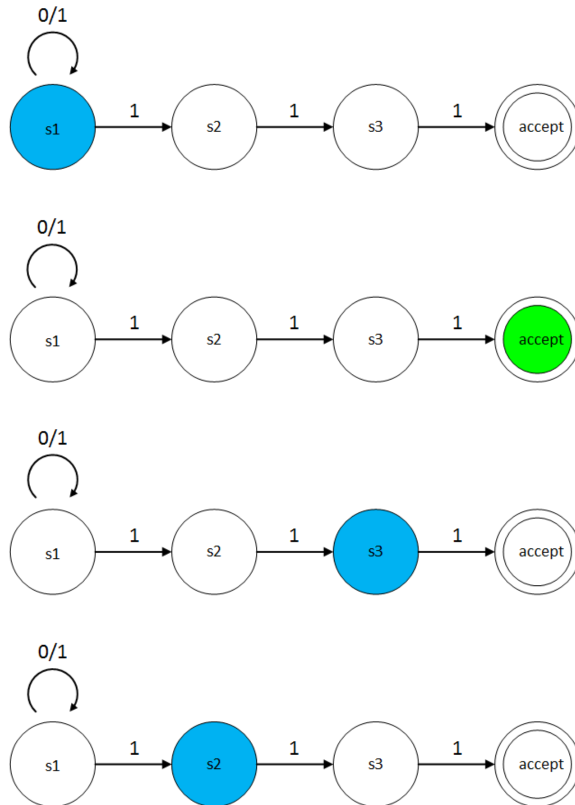

Figure 3.11: Four active clones, accepted input

Chapter 4

Design of the virtual layer

The purpose of the virtual layer designed in this thesis is to make the FPGAs resources more accessible to software applications. A self-cloning state machine has been implemented on the FPGA, with an interface for the software to make it possible for applications to define state machines and run them on the FPGA. This chapter describes the system that was implemented, and why it was chosen.

4.1 Requirements

Before any designs steps can be taken it is important to clearly state the requirements of the system. This makes the designs process much easier because the end goals will be very clear. A list of the system requirements follows:

- The virtual layer has to consist of both hardware and software components, and a connection between them. The hardware components will be placed on the FPGA, while the software components will be running on the PowerPC microprocessor.
- The virtual layer has to provide an interface for the software to configure and run any NFSM.
- There has to be a static part on the FPGA that can maintain connections to the NFSM during partial runtime reconfiguration. The NFSM has to be placed at a reconfigurable part of the FPGA.
- The system must be able to copy an NFSM when a copy transition is detected, and it must be able to remove any clones that are no longer valid.

4.2 System overview

For the virtual layer to map any NFSM down to the FPGA at run-time, the normal approach where an NFSM is designed before configuring the FPGA cannot be used. Instead a

generic NFSM design is needed. Making the state machine generic can be achieved by adding registers. These registers can then be used as lookup tables, were every transition for the state machine is stored. The state machine can use the lookup table to find every active state for each input. When the state machine is implemented the registers can be configured with the correct lookup table, and the state machine is ready to run. This is pretty much the same approach as used in Hubner et al. (2011). Naturally this will consume more resources, but is necessary if the NFSM is to be generic.

In this thesis the NFSM consist of one or more clones, depending on the input and transitions. Ideally this would be the only logic needed on the hardware side, and the clones could be freely placed all over the FPGA. There is however a need for several other modules as well. These modules are needed to facilitate connections between hardware and software, and also to set up a connection to the configuration memory. Also the clones have to be separated from the rest of the components on the FPGA to make partial runtime reconfiguration possible. Without this separation the FPGA will fail and possibly also be damaged during reconfiguration. Therefore a reconfigurable module that is separated from the other logic on the FPGA was added to the design. This module contains the NFSM. On the static part of the FPGA a second module was placed. This is the back-end of the system, and is needed so that a permanent connection with the software running on the processor can be maintained. It is also useful because any feature that does not need to be a part of the reconfigurable module can be placed here. For example handling the configuring of clones, and handling the communication back and forth. This helps to make the reconfigurable module less complicated, and save space. The resource usage in the back-end is less important than in the reconfigurable module, mostly due to restrictions in the reconfigurable area of the FPGA. A detailed description on these restrictions is given in the next chapter.

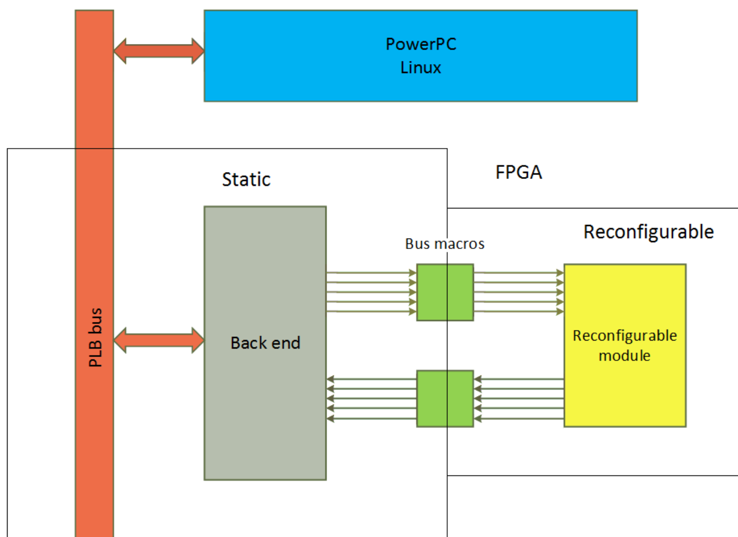

Figure 4.1: Overview of the system

An overview of the system is depicted in Figure 4.1. The main components of the system are the reconfigurable module, the bus macros, the back-end, and the PowerPC microprocessor running the user applications. The bus macros connect the reconfigurable module to the back-end. The number of bus macros needed depended on how many clones that is connected. There had to be at least two bus macros connected to every possible clone, with every bus macro providing eight bits of width. One half of the bus macro is static and the other half will be reconfigured during partial reconfiguration. This means the number of bus macros also has to be static and set at design time. During design and testing it was found that two input bus macros were needed, while only one bus macro was needed for the output. For the experiments done in this thesis only four clones were needed, so the number of bus macros was set to match this.

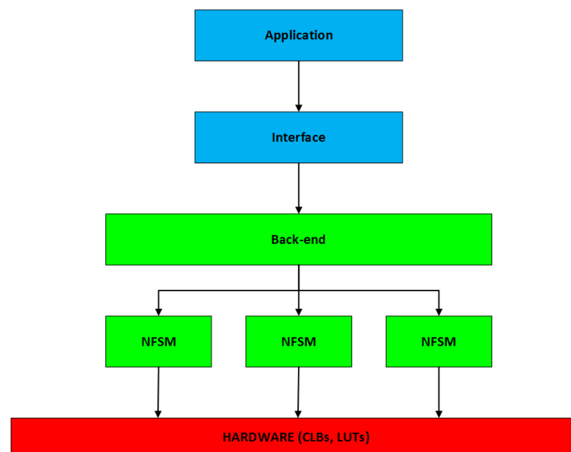

Figure 4.2: Structure of the virtual layer

Figure 4.2 shows the process of mapping a definition of a self-cloning state machine down to the hardware on the FPGA. The user application interacts with the interface for defining and configuring a state machine. The information is transferred to the back-end module on the FPGA, and further to the NFSM clones. The state machine can now be realized as logic on the FPGA. Software components are marked in blue, user modules on the FPGA in green, and the actual hardware components in red.

The back-end needs a way to communicate with the processor. The PowerPC PLB bus is already in place on the FPGA and offers a standard communication interface between hardware and software. This is more than enough for the simple systems tested in this thesis, so the back-end is connected to a software accessible register, and added as a slave of the PLB bus peripheral. One half of the register is used as the input to the back-end, written from software, and the other half is used as the output of the back-end and can be read by the software. The size of this register can be altered after the needs of the system.

One of the major decisions that had to be made is how to handle the copying of clones. The ICAP interface has the ability to both read and write to the configuration memory. A copy operation can be done by reading a clone's configuration frames, and then writing them to another location on the FPGA. However in Endresen (2010b) it was shown that the

read operation of the ICAP interface is considerably slower than the write operation. It is safe to assume that copying a NFSM clone by reading the configuration frames will also be slow. The time spent on reconfiguration is critical, since all operation in the NFSM must be halted. Even though the rest of the FPGA can run during reconfiguration, the timing between the different clones of the NFSM must be the same. In Endresen (2010b) a faster solution was found by implementing a module on the static part of the FPGA that could load and save the state of a connected reconfigurable module. It was decided to adopt this approach, and implement support for loading and saving the states of NFSM clones in the back-end module. In this way, new clones added during partial reconfiguration will be unconfigured and will receive the configuration vectors from the back-end. There is still be a need for partial runtime reconfiguration, for example when there are no more available clones, or the parameters of the clones needs to be changed, but the slow read operation of the ICAP interface will be avoided. This implementation strategy raises the possibility for adding more clones during reconfiguration than what is actually needed at the time. When the next clone is needed it can already be in place. Just like prefetching in processors, this can possibly help save time in larger systems. Obviously adding too many clones would be inefficient, so proper analysis of such a system would be necessary.

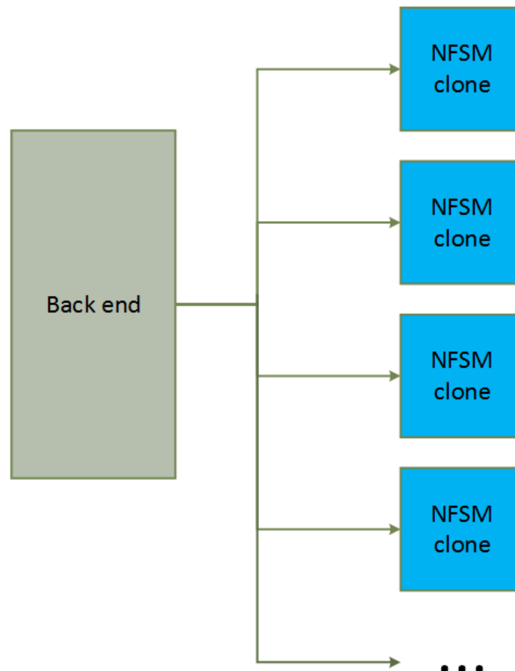

Figure 4.3: Connection of clones

4.2.1 Reconfigurable module

The reconfigurable module is the main part of the virtual layer since it contains the actual NFSM. The virtual layer will actually consist of several reconfigurable modules, since it will need room for one or more clones. All of the clones are connected to the same back-end, but they have no connections between themselves. Figure 4.3 shows how the clones are connected. With reconfiguration the number of clones can be changed at runtime, although the maximum number of clones must be known at design time since there has to be an equal amount of bus macros in place. In Blomkvist (2013) a singular state was intended to be the unit that could be cloned and reconfigured. This strategy requires a reconfigurable module with bus macros for every state in every NFSM clone, and for each cloning process all states must be cloned. Alternatively one would have to find a way to manipulate connections between modules at runtime, perhaps with a NOC system like in Yang et al. (2010). This was thought to be unnecessary complicated, and it was decided to place a whole NFSM clone with all its states inside one reconfigurable module. This makes the cloning process much easier. Still many of the implementation ideas from Blomkvist (2013) were used, and some of his designs files were used as a starting point.

The internal design of the NFSM clone is depicted in Figure 4.4. It consists of a controller, the actual NFSM with all its states, and a configuration memory.

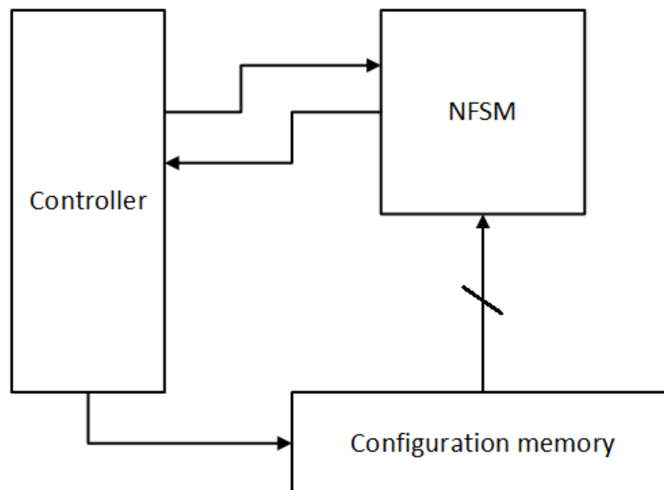

Figure 4.4: Internal components of the clone

The controller handles the communication with the back-end, controls the operation of the NFSM and handles the configuration of the memory registers. The controller has three modes of operation:

- IDLE: Nothing is done, no information in the registers, the NFSM is not operating. The controller waits for an enable signal from the outside (back-end) that configuration vectors are ready.
- CONFIGURE: Configuration vectors are received on the input. These vectors are

serially shifted into the configuration memory. The time this takes depends on the number of registers used per state. The NFSM is still not operating.

- OPERATE: The configuration memory is ready to be used. The NFSM is now activated, and will update the active state. The controller is monitoring the NFSM, and will signal the back-end if the NFSM either accepts the input stream, finds a copy transition, or if it has no legal transition.

The configuration memory stores the configuration vectors used by the NFSM to determine the next state. The memory contains several sets of registers, one set for each state and an extra set for the possible copy transitions. The vectors are shifted in serially by the controller and connected in parallel to the NFSM. The size of the memory can vary, both in the number of vectors and in the number of registers per vector. The number of vectors is determined by how many states there are in the NFSM, and how many vectors there are used for each state. Proper analysis is needed to find suiting number vectors for each state, but as a rule of thumb the number of vectors per state can be set to match the number of states in the NFSM. The number of registers per vector is determined by the number of states and the number of inputs to the state machine. This is where the runtime reconfiguration really shows its worth, because if the configuration memory is too small to operate a given NFSM, the memory can be expanded by swapping the clones in the reconfigurable modules.

As the number of inputs and states can vary between different state machines, the structure of the vectors also changes. This can be challenging when matching the inputs and active states with the vectors. As a general rule all the input bits are placed to the left of the vectors and the rest of the bits are used for the states, the rightmost bit representing the first state. In Figure 4.5 a vector with two input bits and four states is shown. To match this vector the state machine has to receive two ones on the input and the first state has to be active. The vector will be placed with the state that should be activated during this transition.

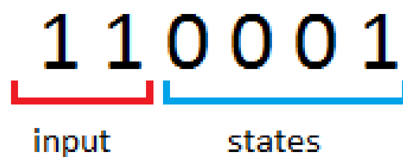

Figure 4.5: Example structure of a configuration vector

The NFSM module contains the actual states of the state machine simulated. It evaluates the next transition by comparing the current active state and input with the vectors stored in the configuration memory. One of the states will be designated as the accept state of the NFSM, if it becomes active the NFSM module signals the controller that the input stream is accepted. This will also disable the NFSM, since it cannot transition out of the accept state. If there is no match found in the configuration memory, no state will be activated and the NFSM will signal the controller that the input is rejected. The whole

NFSM clone should now be deactivated and deleted. If a copy transition match is found, the NFSM will also signal the controller. The state machine encountering a copy transition should just transition to the state that is indicated in the configuration vectors. Copying of the new clone that will transition to the other state is handled by the back-end.

If more than one clone is active in the system, the timing between them becomes important. All clones should receive inputs at the same time. This becomes an issue when one of the clones is copied, or a reconfiguration is performed. In this thesis it is assumed that such operations will be able to finish before any new input is sent to the state machines. In other words, the shortest time between two consecutive input signals will be longer than any operation done by the system, so that the entire state machine will always be ready to receive when the next input arrives. This may not always be the case in a real world example, but the assumption is important in this thesis because the system is experimental and not developed on all areas. The long time between inputs means that the state machines will have to idle in many clock cycles waiting for the next input. To avoid the state machines continuing to evaluate their next active state even with no new inputs arriving, a control signal was inserted in the back-end to signal the state machines when a new valid input would be available.

4.2.2 Back-end

The back-end is the bridge between the software and the NFSM clones. The back-end controls the configuration of new clones, and deletion of clones that are no longer running. The back-end does not control the actual reconfiguration of modules, this has to be controlled in software and the HWICAP module. That means that the back-end can only copy a new clone if it is already placed on the FPGA, either at the start or after partial runtime reconfiguration. The back-end sends information back to the software about what modules are active, and if the input stream is accepted.

The back-end is part of the static design, and will not be altered during partial runtime reconfiguration. This requires the back-end to be carefully designed since it cannot be changed like the reconfigurable modules. The back-end receives the configuration vectors from the software. Since it has to be able to configure several clones at different times, a memory module to store the configuration vectors was included in the back-end. This memory has to be large enough to support NFSMs with big configuration vectors. One problem with the back-end being static is that the connections to the reconfigurable modules also have to be static, which means the maximum number of clones connected to the back-end cannot be altered at runtime. If the maximum number of clones is big enough this should not be too much of an issue. Nevertheless it will set restrictions on what kind of NFSMs that can be implemented on the FPGA. A solution to this can be to use one of the features of a virtual system, like virtual memory that can support more memory than the actual capacity, the virtual layer could be made to support more NFSM clones that there actually would be capacity for. This would mean having some of the clones stored in memory in the software, while the rest would be running on the FPGA. Then the remaining clones could be swapped into the FPGA while the other clones would be stored. It is easy to see that this could give great value to a system, especially a system with limited resources. This could only be done is all state machines could be evaluated before the next input would arrive. The support for such a feature was not implemented in this project, but

could be interesting to develop in future work.

To communicate with the software, the back-end has access to the software accessible register. The register can be added when inserting user modules in *EDK*. The smallest size available is 32 bit, which is more than enough for the experimental work done in this thesis. Increasing the register size is possible if a larger system is to be implemented. The register is divided in two. The first 16 bits are used by the software to send inputs to the back-end, and the other 16 bits are used by the back-end to send its outputs to the software. A table showing the use of the input bits is shown in Table 4.1, and a table for the output bits is shown in Table 4.2.

15-11	10	9	8	7-0
Not used	Sync flag	Start configure	Reset	Input vector

Table 4.1: Input bits

15-5	4-1	0
Not used/Debug	Active clones	Accepted input

Table 4.2: Output bits

Due to the nature of the software accessible register some kind of synchronization is needed between software and the back-end. Without this it will be difficult for the back-end to know if a new input vector has arrived from the software, for example if two consecutive vectors are exactly the same. To handle the synchronization the *Sync flag* input was added. The idea is that this bit, sent form software, has to match a sync flag stored in the back-end. Each time they match, the back-end reads the input and changes its internal sync flag. If a new input has to be sent, the software has to update the sync flag, so the two flags match again. The *Start configure* signal is used to tell the back-end that simulations can start and that configuration vectors will be shifted in on the next clock cycle. The *Input vector* is used to shift in the configuration vectors and also to send the input stream. If larger configuration vectors are needed, the ordering of input and output bits can of course be changed, but this cannot be done after configuring the FPGA. The back-end configuration memory needs to match the size of the *Input vector*.

Fewer bits are used on the output bits of the register. Some are used for debug purposes, like signaling the state of the back-end and what the next sync flag should be, to ease the testing of the system. The *Active clones* signals have one bit designated for each clone to tell if they are active or not. This was mostly used to see if the system acted as expected when testing, but could potentially be used for a system that keeps track of when new clones has to be added to the FPGA. The *Accepted input* signal tells if any of the active clones has accepted the input stream.

The operation of the back-end has these modes of operation:

- IDLE: Nothing is done, no information in the back-end memory, all connected clones are unconfigured. The back-end waits for software to signal that the configuration vectors are ready.

- INIT: Configuration vectors are sent to the software register serially, the back-end shifts in the vectors to the memory module. All clones are kept at idle.
- CONFIGURE-SM: All vectors are now ready in the back-end memory. This mode is used to configure the clones, only one clone can be configured at a time.
- RUN: Configured clones are now activated. The back-end waits for a valid input to arrive in the software accessible register. All clones are put on hold until a valid input is received.

4.2.3 Software

The software part of the virtual layer consists of two programs, *runNFSM* and *icap_write*. The *runNFSM* program handles the configuration and input to the NFSMs on the FPGA. Any application running on the operating system can use this function to run a desired NFSM, all that is required is the NFSMs configuration vectors and input stream. At this point the input stream must be set before the simulation starts, but this could easily be extended to real time functionality if needed. The program sends the configuration vectors and the input stream, and reads the output from the system. The program was based on a test program taken from Hansen (2011), that was made to send input and read from the software accessible register. The configuration vectors can be stored as a textfile. Ideally the input to the program running the NFSM could be something like regular expressions, expressing computational problems, and then having a program to translate this into a state machine with a set of configuration vectors. Most of the focus in this thesis has been of the hardware features, so configuration vectors are sent to the program directly.

The connection between the user program and the software accessible register is handled by a device driver. The driver is a char driver, it allows the user program to access the register with simple open, read, and write operations. Since the system only uses a simple 32-bit register for the communication, the driver is relatively simple. The char driver used in this system has been compiled on the cross development environment from Atmark-Techno. Since the Linux running on the Virtex-4 supports kernel modules, the compiled driver can be put on the NFS shared folder and installed after boot up of the system.

The second program is the *icap_write* program, described in more detail in 3.6.2, is used to facilitate the partial runtime reconfiguration. The program *CLBRead* is needed to extract partial bitfiles, but this can be done before boot up time and does not have to be handled by the virtual layer.

4.2.4 Datapath

To make an FSM more powerful, a datapath can be added to the system. The state machine will act as a controller, controlling the flow of operation, and the datapath performs a set of operations. This has been done in Blomkvist (2013). For each state machine a datapath was included. The datapath was used to perform a multiply-accumulate operation. Adding a datapath to the current system could make it more porwerful and more flexible in the tasks that it could solve, but would also cause some challenges, especially considering the partial reconfiguration. A datapath would have to be connected to each clone, but since

different tasks would require different datapaths there would have to be a system, just like with the clones, for generically implementing the datapaths. Perhaps a similar system with lookup tables could be used, or something more instruction based like a processor. A datapath has not been added to the design in this thesis, but would be a useful next step in the further development of the design.

4.3 Limitations

The virtual layer has been made with the idea that it can be implemented and tested on the FPGA, and actually run some selected NFSMs on the system. Given the nature of an NFSM, which in theory can have an infinite number of variations, some limitations had to be set on the system. The system was mainly designed to simulate small and simple state machines, with not too many complex connections. It certainly is capable of handling larger state machines, but not all. Further, there's only support for one initial state. There can only be one state with a copy transition, and it will only handle one copy. Also there can only be one accept state. To add support for more complicated state machines more information would have to be added to the configuration vectors, this increases the size of the reconfigurable modules, and requires more communication between the static and reconfigurable modules. As there is only 16 bits for the input to the system, the number of bits for the input stream and the configuration vectors are limited. The size of the input and output register could of course be expanded, but this would again require the memory in the back-end to be expanded since it has to be able to handle the larger input and configuration vectors. This will require more resources. If this system should ever be used in any real way, it would probably be a good idea to analyze the resource requirements, so the maximum size of the system could be determined.

What limits this system the most is having the support for runtime reconfiguration. Setting up a reconfigurable module with bus macros is a cumbersome process, and if it is not done correctly the reconfiguration will fail and leave the FPGA in an unusable state before it can be reflashed. The problem is that the space on the Virtex-4 FPGA is limited. The virtual layer with the back-end and the reconfigurable modules does not take up too much resources, but the system needed for the PLB bus and the HWICAP module does. This would not be a such a big problem if the normal tool chain with the many placement optimizations could be used. But since the support for reconfiguration requires the reconfigurable modules to be completely free for all other routing than from themselves and the bus macros, there are limits on how close other modules can be to the reconfigurable modules. What makes this worse is that when designing the system with the tools from Xilinx, restrictions can only be set on where the logic should be placed and not the routing. This causes routing from nearby logic to be routed into the reconfigurable logic. There are tools to reroute such signals, but they are not very effective, and can only handle a handful of illegal signals. The lack of space was such a big problem that only one reconfigurable module could be placed on the FPGA. However one of the main advantages of developing the virtual layer is that it should be possible to implement on other FPGA devices. The interface would stay the same, and for the user applications there would be no difference. The reason for why the Suzaku-V development board was chosen is that it came with an installed operating system and a cross development environment

available. The fact that the framework for reconfiguration was made for this board made it an obvious choice. Other boards with newer and larger FPGAs could have been used, but would require a lot of time setting up a functional system, especially converting the framework for reconfiguration to this new board.

4.4 Implementation alternatives

It can be useful to evaluate some alternative implementations to better evaluate the implementation chosen in this thesis.

4.4.1 A Self-Reconfigurable Gate Array Architecture

The paper Sidhu et al. (2000) presents a new alternative to hardware architecture like the FPGA. The self-reconfigurable gate array architecture (SRGA) offers much better support for reconfiguring devices at runtime. Architectures like the Virtex-4 used in this thesis offers support for runtime reconfiguration, but is slow and comes with little support. This new architecture is capable of doing partial reconfiguration with single cycle context switching and single cycle random access to the on-chip configuration memory. Trying to implement the hardware part of the virtual layer on such architecture could be interesting, and would most likely ease the process of partial runtime reconfiguration. This process has proved to be very cumbersome on the Virtex-4 architecture. However one of the advantages of implementing the virtual layer on the Virtex-4 is that it has been easy to use and set up the connection between hardware and software, something that probably would take some time on architectures like the SRGA.

4.4.2 Xilinx development flow

The support for partial runtime reconfiguration on Xilinx devices has been scarce, but a few years ago they added support for their own partial runtime reconfiguration development flow. Support was added in the program *PlanAhead*. User modules on the FPGA could now be set as reconfigurable modules. This makes it possible place and route the design without adding bus macros and without having to check and reroute the design in *fpga_editor*. The program can also extract partial bitfiles, with complete address locations. The reason this alternative was not used was that the partial bitfiles can only be placed at one particular reconfigurable module, compared to the approach used in this thesis where one partial bitfile can be placed in all the reconfigurable areas. In a big system with many reconfigurable modules and many versions of the design the amount of partial bitfiles will increase dramatically. Also later on it was discovered that this development flow is only supported in ISE 12, and not in the 10.1 and 11.5 versions used in this thesis. Updating the basic FPGA design from Atmark-Techno would be extensive work as described in 3.1. The framework for partial reconfiguration would also have to be updated.

Chapter 5

Implementation and verification

In this chapter the method for implementing and verifying the virtual system is shown. The final verification of the virtual layer was divided into two separate tests cases, one test case with four NFSM clones and no runtime reconfiguration, and a second test case with only one NFSM clone but with runtime reconfiguration. The reason for separating the testing in two parts was that the substring detector, with four available clones, could not be fitted on the FPGA together with support for partial runtime reconfiguration. Because of this the virtual layer was first implemented without support for partial runtime reconfiguration and tested. Then a smaller system was tested with partial runtime reconfiguration. The details of these two test cases is described in 5.5. As mentioned earlier the virtual layer was designed with an experimental approach that took a lot of trial and error to find the best solutions.

5.1 Implementing the virtual system

The implementation of the system was done in several steps. The development flow was a bottom up approach, and began with designing the NFSM clone. This was done in the ISE program, were synthesis and simple testbenches could be used to verify the behavior of the module. The focus of these tests was to see if the clone could be configured properly and use the lookup table to find the next active state. When the NFSM clone was determined to function properly, the back-end was added to the design, and new testbenches were made. Further, the back-end was expanded to be able to handle more than one clone. Four clones were connected to the back-end, and the system was tested to see if the back-end could configure new clones when copy transitions were detected.

Then the system with four NFSM clones and the back-end was added to the EDK project from Atmark-Techno, where all modules needed for the hardware software connection were already in place. The bus macros had to be inserted between every clone and the back-end. The guide Endresen (2010a) gives a detailed description on how to add a custom module to the FPGA, and how to connect it to the PLB bus. The module was given a 32-bit software accessible register, and connected as a slave to the PLB peripheral. The

bus address was set to be '0x81000000', same as in the guide. This address has to match the one given in the driver. The project was then synthesized, and a bitfile was generated. Before this bitfile was downloaded to the FPGA, it was put through the program *bitinit* to initialize the memory on the FPGA. If this is not done the FPGA will not boot after the bitfile has been downloaded. The initialized bitfile was downloaded to the FPGA, and the system was tested.

5.1.1 Timing issues

When doing initial testing on the FPGA, the system acted strangely and there seemed to be problems with the timing. The output of the system was not as expected for given inputs, and the configuration memory in the back-end seemed to not be correctly configured. It was hard to confirm this suspicion since debugging the system on the FPGA was limited, and the content of the memory registers could not be read from software. Further simulations were done in ISE to try to locate the source of the problem, but the problems only occurred when testing on the FPGA. It was suspected that delays in the circuits caused the problem, so the system was given more time to handle each input. This did not fix the problem, so something else had to be wrong. After further debugging on the FPGA, it was discovered that if the input was set before updating the sync flag, the system gave the correct response. It seemed like the two input bytes arrived at different times, and that this caused the system to receive the vectors at wrong times. It was concluded that the driver sent the input in one byte at the time, and that the first byte would be detected by the system before the other could be sent. This became a problem because the sync flag was not in the same byte as the input for the system. The problem was fixed by being careful and sending the input to the system without updating the sync flag, and then sending the input with the correct sync flag.

5.2 Runtime reconfiguration

To make partial runtime reconfiguration possible, several steps had to be taken. The logic had to be restricted in different sections, and the routing had to be inspected to make sure that the reconfigurable module did not have any routed wires crossing in and out except for the bus macros. The HWICAP module was added to the project design, connected to the PLB peripheral and given the start address of '0xF0F00000'.

5.2.1 Restriction of logic placement

At first the system was added to EDK and synthesized, but instead of routing and placing the design, the synthesized system was opened in *PlanAhead*. Here the different modules were put in different Pblocks, and placed on the FPGA. The reconfigurable module was placed in the top right corner. The other modules were placed as far away from the reconfigurable module as possible to give space, and to avoid any unwanted logic crossing into the reconfigurable area. The bus macros were not placed in Pblocks, but rather placed manually by specifying the slices they would occupy in the constraints file. They were placed on the 20th and 21st CLB column. The design with all the modules placed in Pblocks

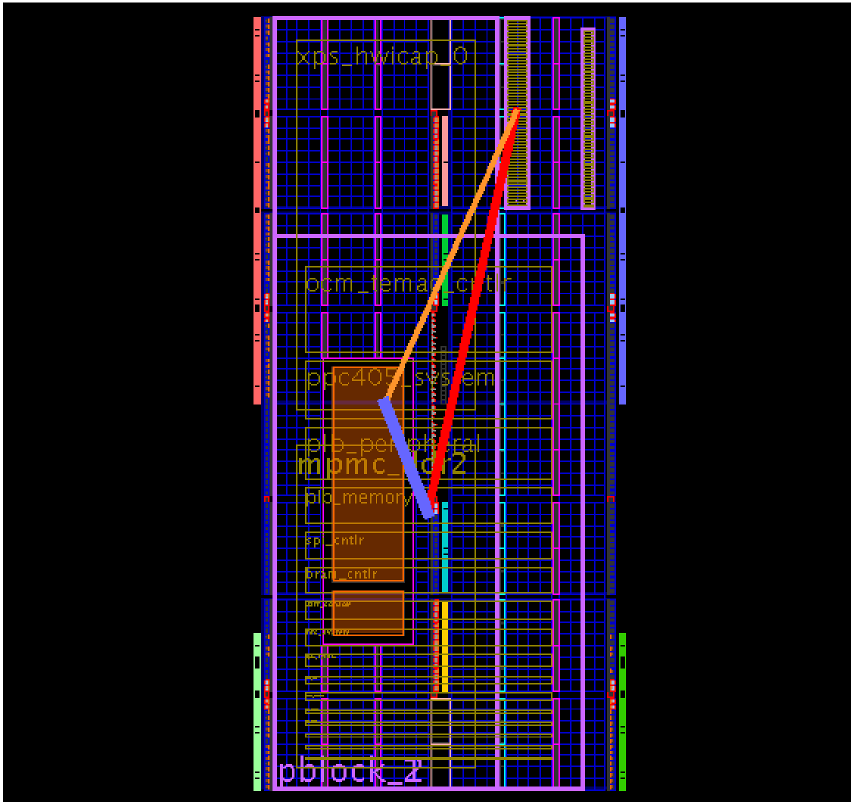

Figure 5.1: Modules divided into pblocks in *PlanAhead*

is shown in Figure 5.1. The reconfigurable module can be seen as the small rectangle in the top right corner, and the back-end module to the left of it. The PLB connections and the other modules like the HWICAP were placed in the other Pblocks, making up most of the space on the FPGA. The placement restrictions were exported to a constraints file, and merged with the constraints file from the EDK project. The system was then placed and routed.

5.2.2 FPGA Editor

Before the bitfile could be used, the routing and placement was inspected in *FPGA Editor*. Routing that crossed the 20th and 21st CLB columns without being a part of the bus macros had to be rerouted. In Figure 5.2 an example of such illegal routing is shown. The illegal wires crossing the CLB columns are marked in red, while the bus macros are marked in white. The rerouting was done by a trick showed in Endresen (2010a), that unroutes and connects an unused slice to the illegal wires, and then routes them again by using the autoroute function. The added slices were placed in areas that were ideal for the wires to go through, avoiding a path crossing the reconfigurable module. This method worked well

in most cases, and was considerably easier than to manually route signals. However there were some cases with signals that refused to change their path. Adding more unused slices was tried, as suggested by Hansen (2011), but sometimes the path of the signal could not be changed no matter how many slices were added. In these cases the place and route was done again, until a more manageable routing was found.

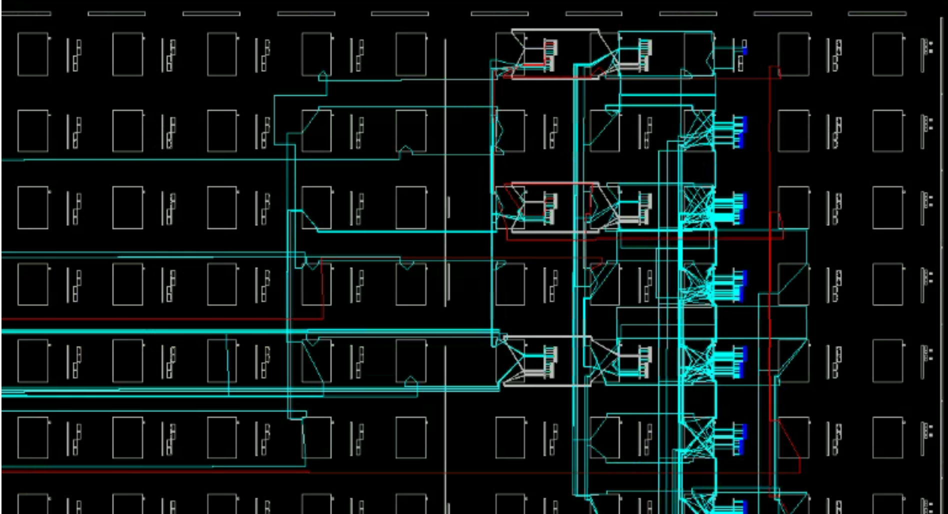

Figure 5.2: Example of illegal routing crossing the bus macro CLB columns

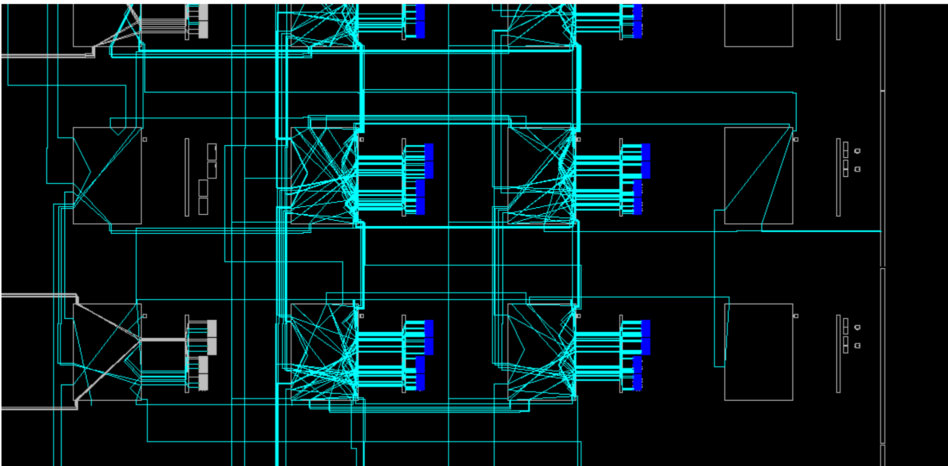

Figure 5.3: Example of routing in the I/O column

When inspecting the routing of the design, one of the I/O columns would sometimes be used to route signals. An example of this is shown in Figure 5.3. The I/O columns is located to the right of the three CLB columns used for the reconfigurable module. It

seemed that even though it had no active logic, the place and route algorithms would use the column for routing. This is not an ideal situation, considering that the available framework for reconfiguration is only able to handle CLB columns. When partial bitfiles are extracted by the *CLBRead* program, they will not include the routing that goes through the I/O columns. If partial reconfiguration is done with such bitfiles, the resulting routing on the FPGA is likely to become incomplete and the FPGA will be damaged not able to boot. This is also true for any of the other columns on the FPGA like the BRAM and DSP. When the rerouting of the design was done, great care was taken to make sure there was no routing in any of these columns inside the reconfigurable module. In Figure 5.4 a completed rerouted design is shown, the bus macros are highlighted in red.

Figure 5.4: Example of a completed rerouting process with only legal wires

In fact this problem is probably the reason why the sequential multiplier designs in both Endresen (2010b) and Hansen (2011) failed to reconfigure. The reconfigurable areas of these designs crossed though a BRAM column, and had routing in the I/O column.

After the routing was checked, and illegal routing rerouted. The bitfile was created by using the program *bitgen*. Then the partial bitfiles were extracted using *CLBRead*.

5.2.3 Clock signal

According to Hansen (2011) the clock signal cannot be routed through the bus macros, and has to be routed trough the global clock network instead. In Figure 5.5 an example of the reconfigurable module's connection to the global clock network is shown. The clock signal enters the module via the global network and does not go through the bus macros. This would normally be considered illegal routing. However, since the clock signal has to enter the reconfigurable module from somewhere, and cannot use the bus macros, it

must be considered legal. To not cause any problems during partial reconfiguration, like unconnected wires, the reconfigurable module's connection to the global clock network has to be exactly the same before and after every reconfiguration. The routing inside the module can be different, since the routing of the clock signal in the CLB columns is handled by the *CLBRead program*. In Figure 5.6 a closer look of the routing of the clock signal at a branch point is depicted. All the wires shown are part of the global clock network. The clock signal connected to the reconfigurable module is marked in red. It does not matter which branch that is used, as long as all of the reconfigurable designs have routed the clock signal through the same branch when entering the reconfigurable module. If this is done, the clock signal should stay intact after a partial reconfiguration. The clock signal for both designs tested was studied to make sure they used the same routing branch. When this was confirmed, the clock signal was considered ready for reconfiguration.

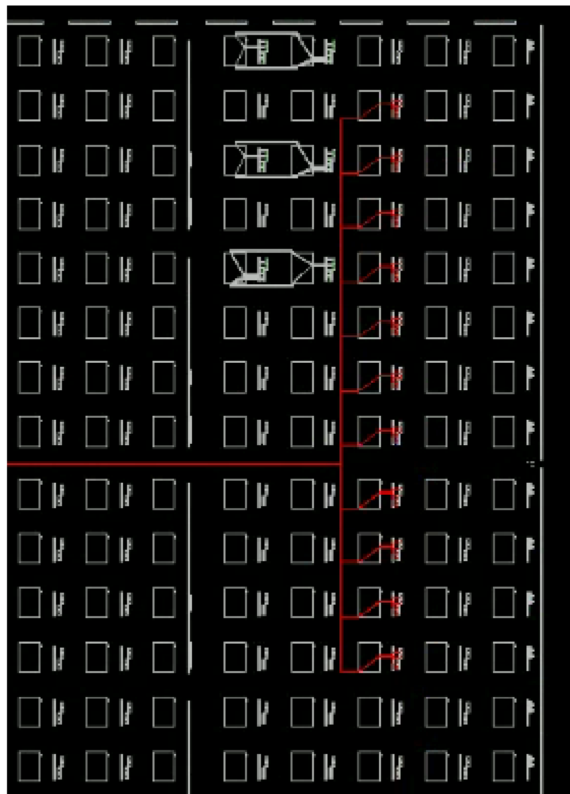

Figure 5.5: Global clock routing

5.3 Drivers

Both the drivers that were used was put in the NFS folder and tested. The driver for the software accessible register was installed successfully, and worked as expected. There

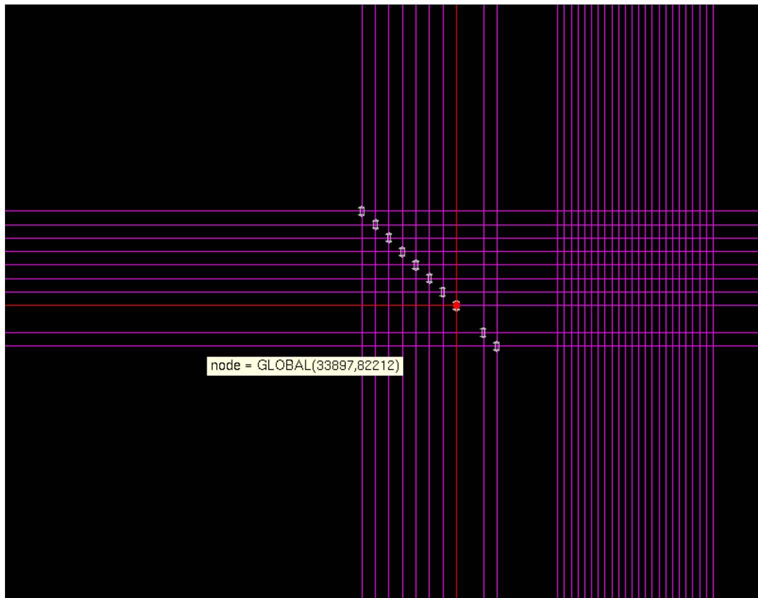

Figure 5.6: Closer look of the clock signal

was however a problem with the HWICAP driver. It refused to be installed with the major number given in the source file for the driver, and the default location for the driver */dev* had a write protection. Therefore some minor changes had to be made to the driver. The major number was reduced to 250, which was accepted, and the location was changed to */var*. Since the location folder of the driver was changed, the minor changes were also made in *icap_write*. This fixed the problems and the driver was installed successfully. The alterations made in the driver and program are shown below.

```
#define XHWICAP_MAJOR 250
```

Listing 5.1: Changes made to the HWICAP driver.

```
#define ICAP "/var/icap"
```

Listing 5.2: Changes made to the *icap_write* program.

5.4 Program

At first all these design steps were taken in the ISE 11.5 design suite, since this was the newest version available with the FPGA project from Atmark-Techno. This worked fine until it was time to use the HWICAP driver and user program. The driver was not able to

contact the HWICAP module. The reason for this was that the HWICAP module in 11.5 had been updated from the 10.1 version. A driver for this version probably exists, but it is likely that several changes would have to be done to make it run on the Linux 2.6.18 distribution. The user program *icap_write* would probably also need big changes. It was uncertain how much time this would require, and if at all possible to do. Therefore it was decided to move the project over to the 10.1 version. The design process was in most cases exactly the same, but the HWICAP module used considerably more resources. This made it more challenging to place the logic on the FPGA. The resources used by the different HWICAP modules can be seen in Appendix D.

5.5 Verification

To verify the virtual layer the substring detector presented in 3.7.1 was implemented and tested. This is a simple state machine that requires only a few configuration vectors, while still containing all the aspects of an NFSM needed to test the virtual layer properly. The most important thing is that is contains a copy transition, so copying can be tested. Originally it was planned to test all features of the virtual layer in one complete test, but when implementing the partial reconfiguration and going through the design steps described in the previous sections, this proved to be problematic. The resources required by the non-user modules and the HWICAP module were high, and left little space for the modules of the back-end and the reconfigurable module(s). Sadly there was only room for one reconfigurable module with support for reconfiguration on the FPGA. Therefore it was decided to separate the verification process into two tests. One full test of substring detector, without any reconfiguration, and a second test to verify the partial runtime reconfiguration with only one clone.

5.5.1 Test with substring detector

In the first test case the back-end is connected to four NFSM clones. This is done to test how the system handled copying and deletion of clones, and how it will handle the control of several clones. One of the four clones is set to be the initial clone and will always be active. The other clones are set to be inactive, and will wait to be initialized and configured by the back-end.

The test is started by sending a series of ones to the NFSM, to see if it will be able to copy new clones during a copy transitions. These new clones should be configured by the back-end and put in the correct state. When three consecutive ones have been sent, the NFSM should accept the input string and then delete the accepted clone. The next one on the input should be treated as a separate accepted substring of ones, and the NFSM should continue to accept the input string until a zero is received. When a zero is received the NFSM should delete all the clones that signal for an illegal transition, which should be all clones except the initial clone. The NFSM should now reject the input string.

5.5.2 Partial runtime reconfiguration test

In the second test only one clone is connected to the back-end, this should be enough to verify the reconfigurability of the virtual layer. The general procedure of runtime reconfiguration is the same no matter how many clones are connected, and only one clone can be reconfigured at a time anyway. In the first test case, clones with room for four states are used. The aim of this test is to change the type of clones during runtime. The clones can be varied in the number of states and in the size of the configuration memory. It does not really matter what kind of clones that are used as long as it can be shown that they can be swapped. It was decided to use the same 4-state clone as in the first test, and then to design a new clone with different parameters. A different clone needs a separate bitfile, so a design with a different clone had to be put through the same implementation steps as mentioned in the first sections of this chapter. At runtime all the different clones must have their own bitfile. The bitfiles can be used to write the desired clone to reconfigurable modules on the FPGA.

Originally the second clone used in this test was planned to have more states than the first clone, but the increase of space made it difficult to fit a bigger clone inside the reconfigurable module. It was decided to use a smaller clone instead. A clone with three states and a smaller configuration memory was implemented, and a bitfile was extracted. The design for the 4-state clone NFSM was downloaded to the FPGA, and the the partial bitfile for the 3-state clone was added to the NFS folder. The 3-state clone could now be swapped with the 4-state clone by using runtime reconfiguration.

The second test uses the detector from in the first test case, but with the copy transition removed. The 4-state clone is configured as a state machines that accepts three consecutive ones, and the second clone is configured as a state machine that accepts two consecutive ones. Configuring these state machines and sending inputs so they accept should only work with their respective clones. The 4-state clone is tested and then swapped with the 3-state clone. Using the different configurations and test should verify a successful change of clones.

Chapter 6

Use of the virtual layer

In this chapter gives a guide of how to use the virtual layer to run a self-cloning NFSM on the FPGA. This guide presumes that the virtual layer has been implemented on the FPGA, and that all the required components are in place.

6.1 Configuration vectors

The first step is to transform the state machine into configuration vectors, for the system to read. The format of these vectors will be as shown in Figure 4.5 and Table 4.1. Every transition has to represent one vector, were the transitions belong to the state they transition to. As an example the transforming of the substring detector is described in this chapter. The first state has two transitions. The second state would normally have one, but this is part of the copy transition that should lead to a copy of a new state machine, and copying is handled by the back-end which also includes forcing the new state machine into this second state. Therefore there are no configuration vectors in the second state. However the transition has to be put in the copy transition section of the vectors, so that they can be detected by the NFSM. These vectors will not cause any transitions in the NFSM, but will signal the back-end. The actual transition for the copy transition is in the first state. The next two states both have one transition each. The amount of configuration vectors for each state has to be the same, where the minimum number of vectors for this example is two. To fill any unused registers one can either fill them with copies of other vectors in that state, or fill all the registers with ones, in both cases this should prevent the state machine of having any uncontrollable transitions. The configuration vectors for the substring detector are shown in Table 6.1, the vectors are given in hexadecimal.

6.2 Creating the input file

The input text file can now be created. One byte is needed for the configuration vectors. A second byte is also needed, containing the sync and enable configuration signals. This

State 1	State 2	State 3	State 4	Copy
01	ff	12	14	11
11	ff	12	14	11

Table 6.1: Configuration vectors for the substring detector

byte will be put first due to the PowerPC endian byte system, more on this can be found in Endresen (2010b). To set the FPGA ready for receiving the configuration vectors, one has to send a *Start configure* signal, and preferably also reset the modules on the FPGA. When sending the vectors the sync flag has to be updated for every vector. Due to the issues discussed in 5.1.1 extra words have to be sent when changing the input byte. An example of a complete configuration file is shown in Appendix B. This file configures the detector, where five vectors were used for every state. The first vector for every state has the correct input but the incorrect sync flag. Therefore six vectors are used for every state. There are also six vectors for the copy transition. The number of configuration vectors and input vectors must be given at the beginning of the file. The completed input textfile can now be put in the NFS folder.

6.3 Running the state machine

Including the input textfile, the *runNFSM* program and the driver must be in the NFS folder. The name of the driver for the software accessible register is *sw_access_reg*, and must be installed as a kernel module. The state machine is now ready to run. The commands for installing the driver and running the state machine is given in Listing 6.1.

```
cd var
mknod swreg c 240 0
cd tmp
insmod sw_access_reg.ko
./runNFSM < input.txt
```

Listing 6.1: Installing the driver and running the state machine

6.4 Reconfiguring with the HWICAP module

To perform partial run time reconfiguration, the HWICAP driver, the *icap_write* program, and a partial bitfiles must be placed in the NFS folder. The process of extracting a partial bitfile is shown in Listing 6.2, where the 3-state clone was extracted from the full design and put in a separate partial bitfile. The address of the columns reconfigured is hard coded in the *icap_write* program. Columns 21 to 23 were used in this thesis, so the program has been set to configure these columns. Other columns can also be reconfigured. This can be done by changing the addresses in the program and recompiling. The process

of reconfiguring the 21st, 22nd and 23rd CLB columns with a partial bitfile is shown in Listing 6.3.

```
$ ./test -i 3-st-reru-10.1.bit -o part-3-st.bit -fmR -sc 21
    -ec 23 -verbose
```

Listing 6.2: Extration of a partial bitfile

```
cd var/
mknod icap c 250 0
cd tmp/
insmod xilinx_hwicap_m.ko
./icap_write -i part-3-st.bit -f 66
```

Listing 6.3: Reconfiguring CLB columns

Chapter 7

Results

This chapter presents the results from the verification process.

7.1 First test case with four connected clones

The results from testing the substring detector is shown below in Listing 7.1. At first the driver for the software accessible register was installed. Then the NFSM was configured with the configuration vectors from the input textfile, and the input stream was sent to the state machine, by running the program *runNFSM*. The NFSM started with one active clone. After the configuration completed the output of the system was '0x0302'. An output of '03' means that the back-end has reached the *RUN* state. So the back-end had successfully been configured, and the configuration vectors had also been sent to the clone. The '02' output told that the first clone was active, and the the input was not accepted. The system was now waiting for the input stream. When the first '1' was sent on the input, the output of the system changed to '0x0706'. This meant that a second clone had been activated by the back-end. The input stream was still not accepted. When the next '1' was sent, a third clone was activated, and when the third '1' was sent a forth clone became active. The back-end now signaled that the input was accepted. When more ones were sent on the input the NFSM continued to accept all the substrings of three consecutive ones, and four clones remained active. This meant that the NFSM managed to delete the accepted clone, and copy a new clone at the same time. When a '0' was sent to the state machine, the input stream was rejected, and all the clones except the initial one were deactivated. A new series of three ones was sent to the state machine to see if it still was able to accept new sub strings. The state machine accepted the new sub string of three ones.

```
# uname -a
Linux SUZAKU-V.SZ410 2.6.18-at11 #1 Mon Mar 16 12:29:01 CET
    2015 ppc unknown
# cd var/
```

```
# mknod swreg c 250 0
# cd tmp/
# insmod sw_access_reg.ko
Major number captured: 0
# ./runNFSM < input.txt}
Program for running NFSM. Program takes configuration
    vectors and input from textfile

Accessing Hardware
FPGA Memory access successful
Configuring NFSM...
NFSM configuration finished.
Output from NFSM: 0x0302
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input to NFSM
Sending input: 1
Output from NFSM: 0x0706
Number of active NFSM clones: 2
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x030E
Number of active NFSM clones: 3
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x071F
Number of active NFSM clones: 4
Accepted input (0 no, 1 yes): 1

Sending input: 1
Output from NFSM: 0x031F
Number of active NFSM clones: 4
Accepted input (0 no, 1 yes): 1

Sending input: 1
Output from NFSM: 0x071F
Number of active NFSM clones: 4
Accepted input (0 no, 1 yes): 1

Sending input: 0
Output from NFSM: 0x0302
Number of active NFSM clones: 1
```

```
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x0706
Number of active NFSM clones: 2
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x030E
Number of active NFSM clones: 3
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x071F
Number of active NFSM clones: 4
Accepted input (0 no, 1 yes): 1
```

Listing 7.1: Output from testing substring detector

7.2 Second test case with one clone and partial runtime reconfiguration

The results from the partial runtime reconfiguration test is shown below in Listing 7.2. Both the software accessible register driver and the HWICAP driver were added and installed in the kernel. At first the 4-state clone was tested by running the program *runNFSM*, three consecutive ones was sent on the input and the state machine accepted. Then *icap_write* was used to change the clone in the reconfigurable module. The CLB columns 21 to 23, containing the NFSM clone was replaced using the partial bitfile for the 3-state clone. The reconfiguration completed successfully after a couple of seconds. The NFSM was tested again, this time with the configuration for the 3-state clone. The NFSM accepted the input after the second '1' was sent to the input.

Synthesis results with resource usage of all the systems tested can be found in Appendix D.

```
# uname -a
Linux SUZAKU-V.SZ410 2.6.18-at11 #2 Fri Apr 17 14:23:19 CET
    2015 ppc unknown
# cd var/
# mknod icap c 250 0
# mknod swreg c 240 0
# cd tmp/
# insmod sw_access_reg.ko
Major number captured: 0
# insmod xilinx_hwicap_m.ko
```

```
Xilinx ICAP driver init
Xilinx ICAP driver platform_driver_register
Xilinx ICAP driver probe
Xilinx ICAP driver setup
icap icap.0: Xilinx icap port driver
icap icap.0: ioremap f0f00000 to c5008000 with size 1000
setup after device_create
# ./runNFSM < input-for-4-states.txt
Program for running NFSM. Program takes configuration
    vektors and input from textfile

Accessing Hardware
FPGA Memory access successful
Configuring NFSM...
NFSM configuration finished.
Output from NFSM: 0x0702
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input to NFSM
Sending input: 1
Output from NFSM: 0x0302
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x0702
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x0303
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 1

# ./icap_write -i part-3-st.bit -f 66
Frames: 66
Access Hardware
Xilinx ICAP driver open
FPGA Memory accessing
Xilinx ICAP driver write
Writing frames
Xilinx ICAP driver write
Finished writing frames
Postconfig
```

```
Xilinx ICAP driver write
Getting Status of FPGA
Stat acces word 2800E001
Xilinx ICAP driver write
Xilinx ICAP driver read
Xilinx ICAP driver write
Status: 78FC
closing device
Xilinx ICAP driver release
# ./runNFSM < input-for-3-states.txt
Program for running NFSM. Program takes configuration
    vektors and input from textfile

Accessing Hardware
FPGA Memory access successful
Configuring NFSM...
NFSM configuration finished.
Output from NFSM: 0x0702
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input to NFSM
Sending input: 1
Output from NFSM: 0x0302
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 0

Sending input: 1
Output from NFSM: 0x0703
Number of active NFSM clones: 1
Accepted input (0 no, 1 yes): 1
```

Listing 7.2: Partial runtime reconfiguration output

7.2.1 Timing

In Hansen (2011) the reconfiguration of large synchronous designs was reported to take up to several seconds, which was in contrast to what the other reports, Endresen (2010b) and Hamre (2009), on partial reconfiguration had found. In their experiments the reconfiguration would only take milliseconds. When performing the same tests in this project the time to completion was also several seconds. Sindre speculated that this could be because the clock signal was also rerouted, or it could have been because of hardware failure, although very unlikely. Because this problem also occurred in this project, the problem was investigated further. The HWICAP driver and *icap_write* program uses a lot of print outs to the terminal for debug purposes. It was suspected that this caused the slow reconfiguration times, since the program has to spend a lot of time writing debug information to the

terminal. The debug print outs were removed from the driver and the program. When partial runtime reconfiguration was tested, the program finished much faster, under a second. A more accurate measurement was not possible since the OS lacked any proper analysis tools. This seemed to fit more with the reconfiguration times reported in the other reports. This was probably also the reason why Sindre's runtime reconfiguration took so long, but it is hard to confirm since the actual terminal output was not omitted in the report.

Chapter 8

Discussion

8.1 The virtual layer

When tested, the virtual layer performed as expected. From configuration vectors defined in software, the system managed to send the information to the hardware modules and implement the vectors as a running NFSM. When encountering copy transitions the system managed to duplicate itself and add new state machines. The system also managed to remove those state machines that no longer had valid transitions, and made place for new ones. The system managed to control the timing when controlling several NFSM clones at once. The test of the virtual system was done with relative simple state machines, but verified that all the implemented features worked. Setting up tests with the virtual layer was hard, since little information could be extracted from the FPGA if anything went wrong. For example inspecting all the memory registers after configuration was not possible. Running the same tests on simulation software from Xilinx could only do so much, and would not simulate any components from the processor and hardware bridge.

By using small state machines and only having one module for reconfiguration available, it is hard to evaluate how effective the system is. It would be interesting to compare the speed of the system to a pure software implementation. One would probably have to test quite large systems to observe any speedup, if any. It is also plausible that the use of self-cloning state machines is only suitable for certain tasks, where others tasks gives no speedup at all.

To make the virtual layer able to simulate any state machine, memory registers had to be added. These registers were the primary use of resources, and the amount increased linearly with the number of states and inputs in the state machines. In one sense, the NFSMs implemented on the FPGA could be seen as something like a processor, flexible and capable to perform many tasks, but sacrificing in resources and speed. To truly make use of this virtual layer something like the datapath discussed in Chapter 4 would have to be added. This would make the system able to handle more complex computations, and make it more powerful.

The system that was implemented has some restrictions in regards to what NFSMs

can be used. It was necessary to set limitations, so that the system did not become too complex at first. This was a good approach and helped to simplify some of the problems encountered in the early phases of the project. Features could then be added along the way, as the system became more and more complete. There are definitely more feature that can be added, some more easier than others.

When running much larger systems, the implementation of a hardware operating system (HWOS) would certainly be interesting. Support for running more than one user application at the same time could be implemented by adding a scheduler. A more complex system for reconfiguration could be added, keeping track of all the resources used by the system. One could also add the support for running some of the NFSM clones while having the rest stored in memory.

8.2 Partial runtime reconfiguration on Virtex-4

When trying to implement support for partial runtime reconfiguration it became clear that the Virtex-4 platform was limited in what kind of system that could fit on the relative small FPGA. The experimental work done in this thesis was done with small and simple state machines, and even so the limit of the resources on the FPGA was reached. However the lack of logical elements was not the only limiting factor. When restricting the placement of logic on the FPGA by using Pblocks, it helped create a clear boundary between the static and reconfigurable areas on the FPGA, but definitely also affected how much the routing and placement could be optimized. The fact that routing could not be restricted in the same manner as logic also made things worse. It became nearly impossible to have any logic placed near the reconfigurable area without unwanted routing crossing over. With the limited tools for rerouting available the only solution was to place other logic even further away from the reconfigurable area. This made only the top row of CLB columns available to place reconfigurable areas. A reconfigurable module was placed in the far right corner, so that the least amount of illegal routing would go through the area. There would probably have been space for one or two more of these small reconfigurable modules on the top row, but there would have been no way to stop routing from these modules crossing in to each other. In the end only one reconfigurable module was implemented on the FPGA. When inspecting the routing in *fpga_editor* it became clear that signals were also routed through the non-CLB blocks, like the I/O blocks shown in Figure 5.3. Again the tools available for rerouting made it difficult do anything about this, other than trying to move the logic further away. What had once been three columns of CLBs for the reconfigurable module became only one, just to simply achieve a result without illegal routing.

Still, two different designs were implemented, and by using the framework for reconfiguration they could be swapped at runtime. These tests proved that NFSM clones could be added or altered by using partial runtime reconfiguration. This adds flexibility to the design. It can be used to create a highly customized system, reacting to the environment at run time. Having the ability to add clones only when they are needed can help save power. Alternatively the reconfigurable areas could be used by other systems that share the same resources, creating possibilities for enhanced effectiveness.

Only small synchronous designs like one or two flip-flops had successfully been reconfigured in previous work. The fact that these reconfigurable modules, like the sequen-

tial multiplier in Endresen (2010b), crossed through several non CLB columns probably caused the reconfigurations to fail. There were simply no way to reconfigure these columns, since the support for this had not been included in the *CLBRead* program.

During testing, a single partial runtime reconfiguration took several seconds when using the HWICAP driver given by Hamre (2009). This was the same results as reported in Hansen (2011). However when the relatively large amount of debug print outs was removed, the time was reduced to under a second, corresponding better to the results seen in both Hamre (2009) and Endresen (2010b). The debug print outs was most likely also the cause of the slow reconfiguration times in Hansen (2011).

Chapter 9

Conclusion

Using self-cloning state machines to bridge the gap between hardware and software is an interesting idea. Even though FPGA designs have gotten considerably easier to implement in the recent years, the process is still not close to being as easy to implement as software designs. The virtual layer makes hardware resources available without having to go through complicated design processes. It also makes the partial runtime reconfigurability features of the FPGA available to use, without having to go through a slow and complicated process that requires a detailed knowledge of the hardware. During testing the virtual layer was shown to be functional, and state machines could be described in software and mapped to hardware. The process of changing the clones at runtime was successful, but only a limited test could be performed.

It became clear that the Virtex-4 is not suitable for implementing large designs. This project focused on using this development environment because it meant that more time could be spent on defining and implementing the virtual layer, instead of using a lot of time on finding and setting up a better development environment.

9.1 Future work

- Implement the framework for partial reconfiguration on a more modern FPGA development environment, like the Xilinx Virtex-6 or -7. This will require rewriting of the programs *CLBRead* and *icap_write*, or complete new ones, to fit the new CLB and bitstream structure. Also adding the possibility to reconfigure DSP, I/O and BRAM blocks could be very benefitial. Alternatively, research could be done in extending the Xilinx own partial reconfiguration flow, so that partial bitfiles will not be limited to a specific area.

- Do a closer study on the timing requirements of the system, and the partial reconfiguration. How fast could a application use this system, and would it be any faster than running everything on a processor.

- Implement a HWOS like in Hansen (2011) and Endresen (2010b) to better facilitate resources on the FPGA.
- Add support for a datapath to extend the use and power of the self-cloning state machines.

Bibliography

Atmark-Techno, 2015. Downlaod page.
URL http://download.atmark-techno.com/

Blomkvist, D., 2013. Self-cloning state machines on fpga. Master's thesis, NTNU.

Endresen, V., 2010a. Creating a reconfigurable fpga system. Tutorial.

Endresen, V., 2010b. Hardware-software intercommunication in reconfigurable systems. Master's thesis, NTNU.

Hamre, S., 2009. Framework for self reconfigurable system on a xilinx fpga. Master's thesis, NTNU.

Hansen, S., 2011. Self reconfiguration of clock networks on fpga. Master's thesis, NTNU.

Hubner, M., Figuli, P., Girardey, R., Soudris, D., Siozios, K., Becker, J., May 2011. A heterogeneous multicore system on chip with run-time reconfigurable virtual fpga architecture. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. pp. 143–149.

Sidhu, R., Wadhwa, S., Mei, A., Prasanna, V., 2000. A self-reconfigurable gate array architecture. In: Hartenstein, R., Grünbacher, H. (Eds.), Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing. Vol. 1896 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 106–120.
URL http://dx.doi.org/10.1007/3-540-44614-1_12

Svarstad, K., Volden, K., 2011. Replicating non-deterministic finite state machines as a mechanism for run time reconfiguration on fpgas.

Yang, J., Yan, L., Ju, L., Wen, Y., Zhang, S., Chen, T., June 2010. Homogeneous noc-based fpga: The foundation for virtual fpga. In: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. pp. 62–67.

Appendix

A Useful commands

This section presents some commands and use of programs that could be useful when using the framework for reconfiguration or setting up designs on the Suzaku-V platform:

```
//Downloading a Linux image to the FPGA via the serial port

sudo hermit download -i image.bin -r image --port /dev/
    ttyUSB0
```

```
//Flashing the FPGA with a bitfile, using the serial port

sudo hermit download -i test_bitgen.bit -r fpga --force-
    locked --port /dev/ttyUSB0
```

```
//Using bitgen to generate a bitfile from ncd file

bitgen -w -f bitgen.ut completed-design.ncd
```

```
//Using bitinit to generate initialize a bitfile before
    flashing the FPGA, performed in the root folder of the
    EDK design. First for the 10.1 and second for the 11.5
    version.

bitinit xps_proj.mhs -pe ppc405_system ppc405_system/code/
    executable.elf -bt implementation/test.bit -o
    test_bitgen.bit

bitinit -p xc4vfx12sf363-10 xps_proj.mhs -pe ppc405_system
    ppc405_system/code/executable.elf -bt implementation/
    xps_proj.bit -o test_bitgen.bit
```

B Example of input text file used for configuring and running the NFSM

```
34      --Number of configuration vectors
12      --Number of input vectors
00 00
01 00   --Reset
00 00
02 00   --Start configure

00 01   --Vectors for 1st state
04 11
00 11
04 11
00 11
04 11

04 ff   --Vectors for 2nd state
00 ff
04 ff
00 ff
04 ff
00 ff

00 12   --Vectors for 3rd state
04 12
00 12
04 12
00 12
04 12

04 14   --Vectors for 4th state
00 14
04 14
00 14
04 14
00 14

00 11   --Vectors for copy transition
04 11
00 11
04 11
00 11
04 11
```

```
04 01   -Vectors for input
00 01
04 01
00 01
04 01
00 01
00 00
04 00
04 01
00 01
04 01
00 01

*When using the textfile it cannot contain comments
```

C runNFSM source code

```
/* Application for accessing software accessible register
   on the FPGA, and configuring and sending input to the
   modules on the FPGA.

   Author: Tormod Heimark, based on setreg.c by Sindre
       Hansen
*/


#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>

#define FPGADEVICE "/var/swreg" // Memory mapped FPGA

//A function that takes the output bytes from the FPGA and
    calculates the number of active clones
int active_NFSM(int8_t byte) {
        int active = 0;
        int i;
        for ( i=1;i<5;i++){
                if( byte & (1 <<(i))){
                        active++;
                }
```

```
        }
        return active;
}

//Function for determing if the input is accepted
int accept_o(int8_t byte){
        return byte & (1 <<(0));
}


int main(int argc, char * argv[])
{
  char readbuf[12]; //Buffers for input and output
  char writebuf[12];
  unsigned int A, B;
  int handlemem;
  int ret,i, j,n,conf_count,input_count;

  //Scanning the input file to find the number of
      configuration vectors and input vectors
  scanf("%d", &conf_count);
  scanf("%d", &input_count);
  printf("Program for running NFSM. Program takes
      configuration vectors and input from textfile\n\n");

  printf("Accessing Hardware \n");
  handlemem = open(FPGADEVICE,O_RDWR); //Access through the
      driver
  if(handlemem == -1){
    printf("FPGA Memory could not be accessed\n");
    goto err;
  }
  else {
    printf("FPGA Memory access successful\n");
  }

 printf("Configuring NFSM...\ n");
 for ( n=0; n < conf_count ; n++) { //Taking the vectors
    from the file and sending them to the software
    accessible register

  scanf("%x", &A);
  scanf("%x", &B);
  writebuf[0] = A;
  writebuf[1] = B;
```

```
  ret = pwrite(handlemem, writebuf, 2, 0);

}
  printf("NFSM configuration finished.\n");
  ret = pread(handlemem, readbuf, 4, 0);

  printf("Output from NFSM:");
  if (readbuf[2] < 16) {
          printf(" 0x0%X", readbuf[2]);
  } else {
        printf(" 0x0%X", readbuf[2]);
  }

  if (readbuf[3] < 16) {
          printf("0%X\n", readbuf[3]);
  } else {
        printf("0%X\n", readbuf[3]);
  }

  printf("Number of active NFSM clones: %d\n",active_NFSM(
      readbuf[3]));
  printf("Accepted input (0 no, 1 yes): %d\n\n",accept_o(
      readbuf[3]));

 printf("Sending input to NFSM\n");
  int last = writebuf[0];
  for ( n=0; n < input_count ; n++) {
    scanf("%x", &A);
    scanf("%x", &B);
    writebuf[0] = A;
    writebuf[1] = B;
    ret = pwrite(handlemem, writebuf, 2, 0);
    if (last != writebuf[0]){ //Check for avoiding
        unecessary print outs
      printf("Sending input: %d \n", accept_o(writebuf[1]))
          ;
      ret = pread(handlemem, readbuf, 4, 0);
      printf("Output from NFSM:");
      if (readbuf[2] < 16) {
         printf(" 0x0%X", readbuf[2]);
      } else {
        printf(" 0x0%X", readbuf[2]);
      }

      if (readbuf[3] < 16) {
```

```
        printf("0%X\n", readbuf[3]);
      } else {
        printf("0%X\n", readbuf[3]);
      }

      printf("Number of active NFSM clones: %d\n",
          active_NFSM(readbuf[3]));
      printf("Accepted input (0 no, 1 yes): %d\n\n",
          accept_o(readbuf[3]));
      last = writebuf[0];
    }
  }

 err:
  return 0;
}
```

D Synthesis results

Design with 4 clones, no HWICAP module

	LUTs	FFs	Slices
All modules	5374	5190	3166
User modules	665	791	483
Reconf modules	432	572	350
Single clone	108	143	88

Reconfigurable design with 4-state clone

	LUTs	FFs	Slices
All modules	8293	5105	5060
User modules	214	237	146
Single clone	50	67	42

Reconfigurable design with 3-state clone

	LUTs	FFs	Slices
All modules	8254	5086	5036
User modules	175	218	134
Single clone	41	48	30

HWICAP module

	LUTs	FFs	Slices
From 10.1	3264	418	1992
From 11.5	970	848	518

E Development computer

Setting up a design on the Suzaku-V board requires a development computer, different operating systems were used for various tasks:

- **Linux Ubuntu 14 64-bit** Used for setting up the serial port connection to the board, and also host the other operating systems as virtual machines. Does work with some of the Xilinx development tools, but not all.
- **Atmark-Techno's virtual machine** Atmark-Techno delivers a virtual machine with all required packages for the cross compiling of source code and building the OS image. A detailed tutorial for setting this up can be found in Hansen (2011).
- **CentOS 5.11 32-bit** This operating system was the only one tested were all programs of the Xilinx 10.1 development tools worked. The OS can be installed as a VM. These were the steps taken to set up the tools once the OS was installed:

  Install Xilinx 10.1 and update ISE, EDK and PlanAhead to 10.1.03

  The setup might not run on a fresh install of CentOS, if so go to System - Administration - Security Level and Firewall, set SELinux to ”Disabled”

  Install two packages: yum install openmotif22.i386 yum install libstdc++.so.5

  Then try to open fpga_editor, remember to set settings32.sh, something like ”source /opt/Xilinx/10.1/ISE/settings32.sh”

  If this still does not work, try setting the DISPLAY environment variable to :0 instead of :0.0

  ”export DISPLAY=:0”