Norwegian University of
Science and Technology

Framework for self reconfigurable system on a Xilinx FPGA.

Sverre Hamre

Master of Science in Electronics
Submission date: June 2009
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Earlier projects have shown that dynamic run time reconfiguration is possible on a Xilinx FPGA using bus-macros for communication between the static and run-time reconfigurable modules. This task is a continuation and broadening of that work.

The main objective will be to propose and develop a framework for a self reconfigurable system on a Xilinx FPGA. Special focus will be on:

- Defining a procedure or method for generating the reconfigurable modules to be used in run time reconfiguration.
- Defining a method for communication with and between the modules, this can be a Network on Chip (NoC) or other equivalent system solutions.
- Specification and partial design of an underlying hardware operation system (os) controlling the loading/unloading and communication with the modules needs to be done. This OS should be able to run on an embedded processor within the FPGA (e.g. MicroBlaze) that can communicate directly with the reconfigurable modules.
- Analysing a few applications to determine restrictions concerning design of reconfigurable modules.
- In addition, a suggestion for a method of swapping modules while the application is running should be proposed.

If time allows, the framework and system should be tested with real applications.

Assignment given: 15. January 2009
Supervisor: Kjetil Svarstad, IET

Summary.

Partial self reconfigurable hardware has not yet become main stream, even though the technology is available. Currently FPGA manufacturer like Xilinx has FPGA devices that can do partial self reconfiguration. These and earlier FPGA devices were used mostly for prototyping and testing of designs, before producing ASICS, since FPGA devices was to expensive to be used in final production designs. Now as prices for these devices are coming down, it is more and more normal to see them in consumer devices. Like routers and switches where protocols can change fast. Using a FPGA in these devices, the manufacturer has the possibility to update the device if there are protocol updates or bugs in the design. But currently this reconfiguration is of the complete design not just modules when they are needed.

The main problem why partial self reconfiguration is not used currently, is the lack of tools, to simplify the design and usage of such a system. In this thesis different aspects of partial self reconfiguration will be evaluated. Current research status are evaluated and a proof of concept incorporating most of this research are created. Trying to establish a framework for partial self reconfiguration on a FPGA.

In the work the Suzaku-V platform is used, this platform utilizes a Virtex-II or Virtex-IV FPGA from Xilinx. To be able to partially reconfigure these FPGA's the configuration logic and configuration bitstream has been researched. By understanding the bitstream a program has been developed that can read out or insert modules in a bitstream.

The partial reconfiguration in the proof of concept is controlled by a CPU on the FPGA running Linux. By running Linux on the CPU simplifies many aspects of development, since many programs and communication methods are readily available in Linux.

Partial self reconfiguration on a FPGA with a hard core powerPC running Linux is a complicated task to solve. Many problems were encounter working with the task, hopefully were many of these issues addressed and answered, simplifying further work. Since this is only the beginning, showing that it is possible and how it can be done, but more research must be done to further simplify and enhance the framework.

Table of Contents

Summary....I
Introduction....1
2.1.Ambient Hardware, Embedded Architectures on Demand....1
2.2.Contributions derived from my work with this project:....2
Background....3
3.1.Reconfigurable hardware need....3
Previous work....9
4.1.Work at NTNU....9
4.1.1.Ingar Hauge....9
4.1.2.Stian Reinersen Arntsen....9
4.1.3.Fredrik Gravdal....9
4.1.4.Anders E. Vestnes, Torbjørn Øvrebekk....10
4.1.5.Sverre Hamre....10
4.2.Work regarding dynamic reconfiguration....10
4.3.Work regarding hardware operating system....12
Methodology....13
5.1.A collection of theories, concepts or ideas....13
5.2.Comparative study of different approaches....13
5.3.Critique of the individual methods....14
Theory....15
6.1.Suzaku-V....15
6.2.Xilinx Virtex-IV Configuration Control Logic....15
6.3.Xilinx Internal Configuration Access Port (ICAP)....18
6.4.Runtime self reconfiguration....18
6.5.Linux μClinux ATMARK-dist....18
6.6.Virtex Bitstream design....19
6.6.1.Virtex II....19
6.6.2.Virtex IV....20
6.7.Suzaku-V setup....22
6.7.1.Network File System....22
6.7.2.Adding hardware to the Suzaku-V FPGA design....23
6.7.3.Accessing hardware on Suzaku-V with Linux 2.6.x....24
6.7.4.Adding ICAP hardware module to Suzaku-V FPGA design....24
6.7.5.Getting HWICAP driver to work for Linux kernel 2.6.18-at11....28
6.7.6.Using the HWICAP Linux drivers....31
My work....33
7.1.Difficulties with dynamic self reconfiguration on FPGAs....33
7.2.My work relative to previous work....33
7.3.Design of reconfigurable modules....34
7.4.Procedure/method for generating reconfigurable modules....36
7.5.Reconfigurable module back-end design....37
7.6.Method for communication with and between modules....38
7.7.Hardware OS....39
7.7.1.Design....39
7.7.2.The Launcher....40
7.7.3.Communication....40
7.7.4.The Scheduler....41
7.7.5.The Placer....42

7.7.6.Prototype....42
7.8.Analysis of a few applications....42
7.9.Suggestion of method for swapping modules while running....43
7.10.Proof of concept run-time reconfiguration....44
7.11.Code....45
7.11.1.Hardware Modules connected to the OPB bus....46
7.11.2.ModifiedCLBRead....46
7.11.3.icap_write and icap_test....47
7.11.4.Genbitfile....48
Results....49
Discussion....53
Conclusion....56
Future work....57
References....58

Figure Index

Figure 1: AHEAD version 1.0....1
Figure 2: Standard platform setup, CPU and peripheral....3
Figure 3: Multi-functional reprogramable logic PCI card....4
Figure 4: Complete reconfigurable platform with CPU and peripherals on a reconfigurable device..6
Figure 5: Reconfigurable platform overview, including reconfigurable modules with and without I/O capabilities. ICAP (internal configuration access port) for enabling self reconfiguration....7
Figure 6: Logic Transistors vs. Logic Productivity....8
Figure 7: Suzaku Board configuration overview....15
Figure 8: VirtexII XC2VP4 logic modules illustration....19
Figure 9: VirtexIV 4FX12 logic module distribution....21
Figure 10: Virtex-4fx12 FPGA logic modules....21
Figure 11: Virtex-4fx12 bitfile frame organization....21
Figure 12: EDK screenshot with the Suzaku-V fpga_project. With a user module added....23
Figure 13: EDK screen shot showing address range, for modules in the Suzaku-V fpga_project.....24
Figure 14: Planahead screen shot with placement restrictions and resource usage shown....26
Figure 15: Screen shot of routed design with hwicap module and reconfigurable module in top right corner....27
Figure 16: Reconfigurable module with busmacros in top right corner of fpga....28
Figure 17: System overview....35
Figure 18: Reconfigurable module design....36
Figure 19: Flowchart for designing reconfigurable module....37
Figure 20: Reconfigurable module back-end....38
Figure 21: HWOS illustration....40
Figure 22: HWOS task states....41

Index of Tables

Table 1: Type1 Packet header format. “R” means reserved for further use....16
Table 2: Type 2 packet header. "R" means reserved for further use....16
Table 3: Opcode format....16

Table 4: Configuration Registers..17
Table 5: FAR register composition...17
Table 6: FAR addressing details explained..17
Table 7: Frames and location in bitstream, bitstream location start to end, left to right. For XC2VP4 20
Table 8: CLB logic frame format, from left to right are top to bottom on FPGA and first to last in bitfile...20
Table 9: BRAM logic frame format, from left to right are top to bottom on FPGA and first to last in bitfile...20

Introduction

This project is an extension to an earlier EU commissioned project called AmbieSense [1].

AmbieSence focuses on Ambient Intelligence where the surroundings can contribute with context information. Using context tags that can communicate with mobile devices in the area and distribute local information. This local information will be available at once without unnecessary interaction from the user. The tags can be remotely uploaded with new information continuously.

2.1.Ambient Hardware, Embedded Architectures on Demand

AHEAD is an extension to the AmbieSense project. In general AHEAD uses tags as in AmbieSense but instead of contextual information they includes processing power to reduce the strain on mobile devices. This enables mobile devices to do more complex tasks or reduce the energy consumption thus increasing battery life.

The concept and version 1.0 of AHEAD is shown in Figure 1.

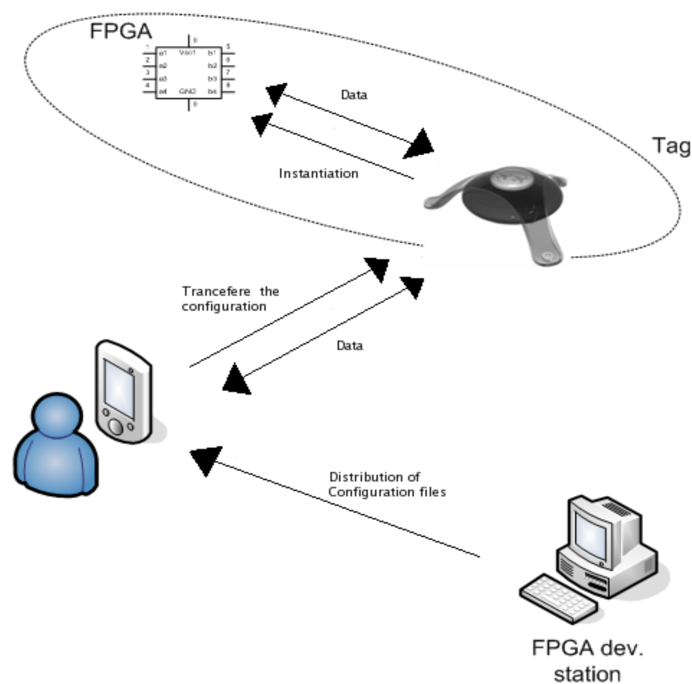

Figure 1: AHEAD version 1.0

To reduce the cost and size of the tags a solution with reconfigurable hardware was chosen instead of a complete computer. This is possible since hardware is usually a faster and more efficient way of running algorithms. The problem with this solution is the integration and generation of the programs to be run on the tag. The first version 1.0 shown in Figure 1 one has to develop a configuration in HDL, compile this to a configuration file and store this on the mobile device. When the program is to be executed the configuration file needs to be uploaded to the tag, the tag then needs to configure the FPGA with this configuration. This solution will only allow one configuration to run simultaneously, on the tag, the overhead from transferring complete configurations to the tag will be large.

To make the system more scalable and dynamic a second version of the system includes partial reconfiguration of the FPGA enabling multiple configurations to

exist simultaneously. By using a partial reconfigurable system the configuration files that the mobile user needs to have stored will be smaller, since they only contain the necessary task configuration.

This thesis will focus on the framework and what is necessary to have a self reconfigurable system continuing the work done in [2]. The focus will be on different aspects: how modules are designed, interface between static design and reconfigurable modules and a hardware operating system controlling everything.

2.2.Contributions derived from my work with this project:

Bitstream composition for Xilinx Virtex-II and Virtex-IV FPGA's.

Programs utilizing the bistream composition for reading out writing in configurations from/to a bitstream.

Configuration principles for Xilinx FPGA's, custom read and write programs for partial configuration of the FPGA.

How to create a reconfigurable module connected to the OPB bus enabling a module to be reconfigurable and connected to a static design by using bus-macros.

Placement constraints, rerouting and inspection of a hardware module to ensure that the module can be reconfigured, defining the module borders.

A prof of concept for self reconfiguration on the Suzaku-V platform using the Virtex-IV FPGA.

Linux driver examples for accessing hardware and program utilizing these drivers.

A concept framework for a HWOS.

Background

3.1.Reconfigurable hardware need.

The advancements in hardware and programming has come a long way. Current systems can do calculations and tasks unimaginable not many years ago. The x86 processor is a old processor design that has been extended, tuned and modified inn all directions to increase speed and efficiency. The x86 and other similar processor types uses instruction-setts to create a general purpose processor. Until recently the way to increase speed on the processor was done mostly by increasing clock speed. In later years designs has moved towards multi core designs to increase processing speed/capacity.

These processors computation capabilities continues to increase and are able to do practically “anything”, they are general and will not be the best or most efficient way to do certain tasks. Also these general purpose processors needs to connect to a multitude of peripherals to have real functionality. Peripherals here can range from communication modules to graphics accelerators. As an example Error: Reference source not found shows a general purpose CPU connected with some peripherals on a bus.

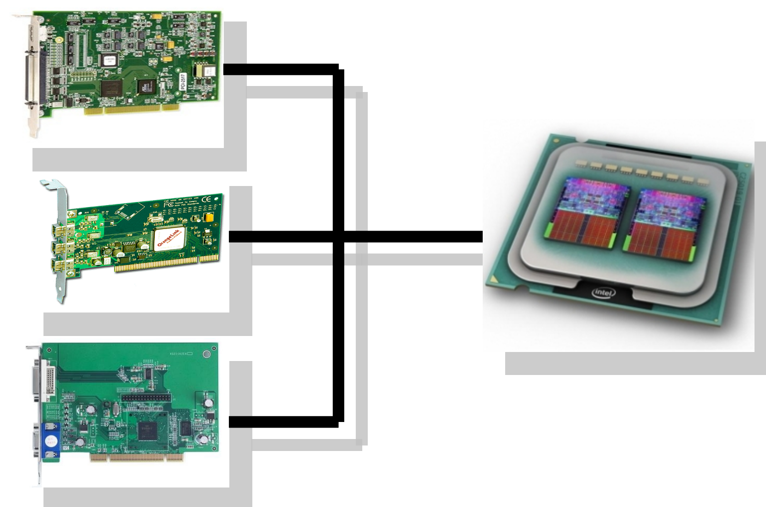

Figure 2: Standard platform setup, CPU and peripheral

These peripherals normally apply to some protocol or functionality, trying to solve a problem in the most efficient way. Normally this implies that the peripheral device has specialized logic doing one task and one task only. When new functionality is needed, support for new protocols, or when bugs are found in the logic are currently an expensive problem. The solutions is if possible trying to do workarounds in software, with the connected processor (if any), or create new logic.

Workarounds in software is not the best solution, unnecessary processing overhead are induced on the CPU and software developer. Sometimes it is not even possible to make workarounds in software due to openness and so on, sometimes a given software has too big market share (no name...) so workarounds are actually done in hardware to comply with the software bugs.

Creating new logic is a costly and time consuming task. The new logic needs to be produced, tested and integrated in the final product. If bugs/faults are found in testing the process needs to be repeated further increasing the cost and time.

To simplify this process and also enable the possibility of fixing ad-hoc solutions, reprogrammable logic can be used. The most used reprogrammable logic design used is the FPGA (field programmable gate array). The FPGA design normally uses LUT (look up tables) to set logic functions for input signals.

SRAM based FPGA`s has to be programmed when power is cycled on, normally a specialized chip does the programming when cycling the power on.

These reprogrammable logic chips were/are mostly used to prototype logic design as they have been expensive. In later years prices on these reprogrammable devices has gone down and designs incorporating reprogrammable devices has increased. By using reprogrammable logic on a device it is possible to update the logic in case of protocol changes or bugs. Thus it is possible to release a product that will be compliant with a protocol before the protocol is fully defined. If bugs are found a simple update of the configuration can fix the bug.

The next step when using reconfigurable devices are the possibility to use the reconfigurable device as a multi purpose device, in the same way as illustrated in Figure 4 where one reconfigurable device can be configured to many other types of devices.

The reconfigurable platform as illustrated in Figure 4 could be configured to

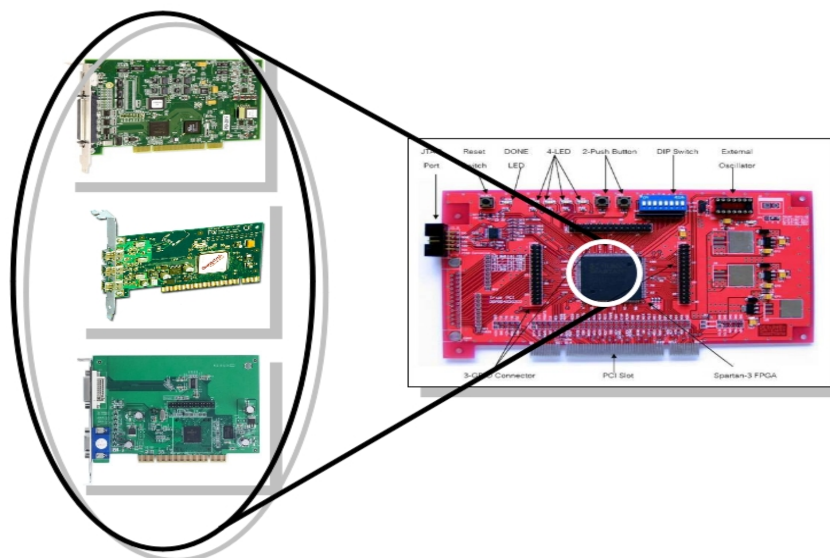

Figure 3: Multi-functional reprogramable logic PCI card

function as one of the other cards when the functionality is needed. When other functionalities are needed the platform could be reconfigured.

Just as software on the CPU these hardware tasks could be scheduled, when the functionality are needed the card can be configured for the task.

The previous solution by timesharing the reconfigurable logic are not plausible for all types of devices. Since many of these devices works on a continuous stream of data in one or both directions.

A solution to the constant stream of data to and from the device can be solved by loading the different configurations simultaneously. This means that the reconfigurable device will be shared for multiple logic designs simultaneously. As long as there is enough “area” on the device to have all the configuration logic loaded simultaneously this will work. There needs to be provided a framework controlling the partial configuration of the device for this to work.

The framework needs to control the parameters of the reconfigurable device, “area” available on the device, physical pin arrangement for communication in and out and also communication with the CPU or other modules.

In Error: Reference source not found the design is taken a step further from what is illustrated in Error: Reference source not found and Figure 2, the complete system is integrated on a reconfigurable device. The CPU can be used to control reconfiguration of modules and other normal tasks.

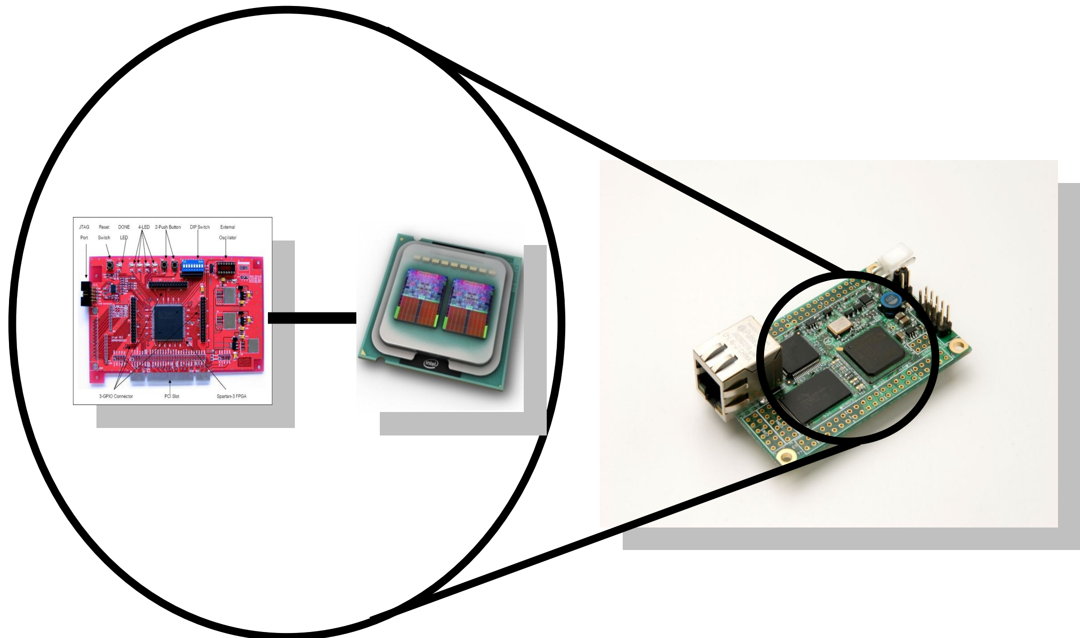

Figure 4: Complete reconfigurable platform with CPU and peripherals on a reconfigurable device

With the system shown in Figure 4 it is actually possible to upgrade the configuration system as this is on the same reconfigurable logic.

Figure 5 illustrates a module design idea for utilization of a reconfigurable device. Here one assumes that the CPU is the main module, controlling reconfiguration and communication local and external. The design is thought to have reconfigurable modules connected to the pins on the device RIO (reconfigurable I/O module) and reconfigurable logic modules (R*... in figure).

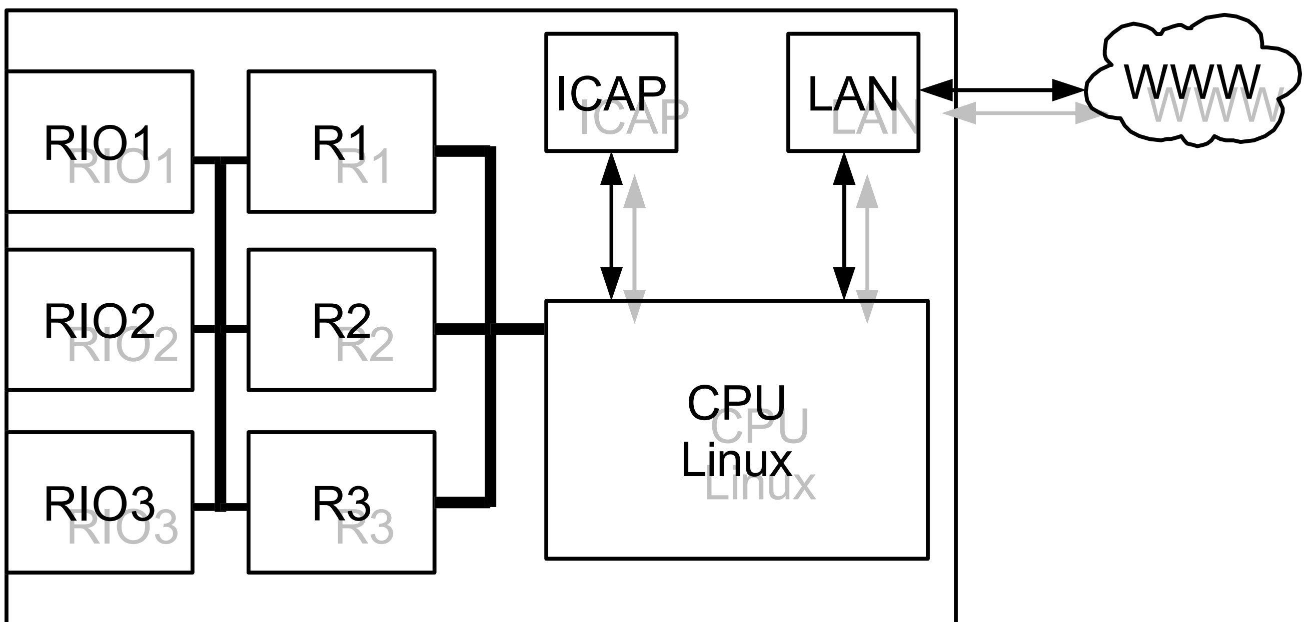

Figure 5: Reconfigurable platform overview, including reconfigurable modules with and without I/O capabilities. ICAP (internal configuration access port) for enabling self reconfiguration.

The main idea here is that all aspects are reconfigurable, but there will be some physical limitations. Depending on the design of the reconfigurable device (FPGA) the input pins can be located in a defined location on the reconfigurable device, this will create some limitations. Also the external connections of the device can not easily be reconfigured (physical connections to sensors and so on).

As an example a serial interface that are connected to one sett of pins can not change pins used. The serial interface will probably also be listening for incoming commands so there needs to be logic always available for handling incoming data. When the serial interface is not used or in listening mode, most of the logic can be removed. This results in a possibility to reduce unused logic, for sending data, thus having only the basic functionality available at all times.

As illustrated in Figure 5 the reconfigurable IO modules are connected with a network to the reconfigurable logic modules. Using some flavor of network system the modules can communicate without direct addressing, resulting in absolute position of modules not being a necessity.

A driving force for reconfigurable systems are the increasing gap in utilization of transistors versus available transistors. As shown in Figure 6 from [3] one can clearly see that the gap between available complexity and productivity are increasing. This gap can be reduced by having a reconfigurable system. The reason is that a specific design does not need to utilize the maximum available complexity. A number of designs and functionality from different designers can be combined in a reconfigurable design, utilizing the available complexity and allowing modification while in use when functionality or need changes. This is the same concept as with NoC [4], just further enhanced.

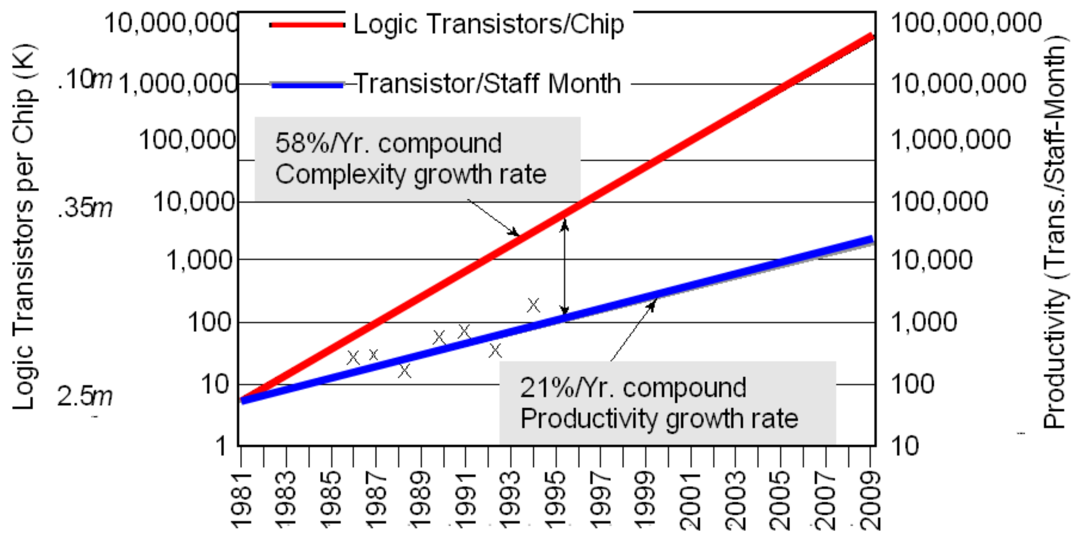

Figure 6: Logic Transistors vs. Logic Productivity.

As can be seen from these trends and the increased demands from the public for systems with increased functionality and reduced power consumption a reconfigurable system can be the best solution. The framework that wraps this together in a complete package giving a hardware module interface for developers to work against, in a similar fashion as API (application programmer interface) does for software.

Key concepts, what is done and what can be done are points that will be further discussed in this thesis.

Previous work.

4.1.Work at NTNU

4.1.1.Ingar Hauge

Ingar Hauges master thesis from 2006 [5] explains in detail how the bitfile for Xilinx Spartan-3 FPGA is composed. With the details of the bitfile Hauge created a program able to convert the bitfile into a bmp image file. This enables one to use a standard picture program to “cut and paste” modules in a bitstream file.

In his thesis he also shows how to use Xilinx tools to ones advantage and explains some limitations in the tools. By using manual routing in the design steps he shows how different designs can be joined together since the same “signal wires” are used.

Ingar Hauges work on the bitfile composition and FPGA details is a major contribution to the work done by Sverre Hamre in [2].

4.1.2.Stian Reinersen Arntsen

The thesis from Stian Reinersen Arntsen [6] describes work on a reconfigurable system where the CPU on the reconfigurable device reconfigures the device. This is done by writing a new bitstream to the flash where the configuration is loaded from, when powering up the device.

As a result, the device needs to be reset for the new configuration to be uploaded. The solution works and shows that it is possible to reconfigure the device on demand with a CPU on the device. A negative point with this solution is that the device needs to be restarted resulting in a full configuration and long boot up time when Linux is used.

Since Linux and the device is restarted, the solution does not allow for multiple users using the device simultaneously.

The resulting reconfiguration time was approximately 70 seconds.

4.1.3.Fredrik Gravdal

Fredrik Gravdal continued working with partial reconfiguration by manipulating the bitstream for Spartan-3 FPGA. His thesis [7] is a continuation of the work done by Ingar Hague [8], [5] creating a program to do the read-out and write-in of logic instead of using a graphic manipulation program.

These small programs created: CLBread and CLBwrite work on the CLB level in the bitfile. By using these programs a CLB or range of CLB`s can be read to and from a file.

Fredrik Gravdal showed how to reconfigure a module connected to some IO pins on the Suzaku-S platform. In his thesis Fredric Gravdal did not show how to make connections to a reconfigurable module for communication between reconfigurable modules, or between reconfigurable modules and the static design.

4.1.4.Anders E. Vestnes, Torbjørn Øvrebekk

The project report by Anders E. Vestnes and Torbjørn Øvrebekk [9] shows some of the problems that are apparent when trying to set up complex systems. Anders and Torbjørn had problems getting the tools working setting up the system. It is a difficult task to understand and sett up a complex system consisting of CPU, memory, communication modules and so on. Also, getting Linux to work with the system requires a deep system understanding which is hard to acquire in the short time available.

4.1.5.Sverre Hamre

The work done in the project report by Sverre Hamre [2] is a continuation of the previous work done in the AHEAD project at NTNU.

Because of problems described in earlier work [9] and the general complexity of the system a couple of tutorials was developed. These tutorials were meant to have a basic approach for getting the system to work, reducing the frustration of not getting even a basic system to work.

The work done in [5] and [7] (by Ingar Hauge and Fredrik Gravdal) abut the composition of the bitfile and the program to read out logic was further enhanced. By using the LUT-based bus-macro proposed by Michael Hüebner, Tobias Becker and Jüergen Becker [10]. It was shown how a module connected to a static design could be reconfigured.

Because of the underlying reconfigurable device used, Spartan-3, it was not possible to do runtime reconfiguration. The functionality and proof of consent was shown by modifying the bitsream off-line and uploading the modified version. With this experiment, it was shown that modules could be located in bitstreams, cut out and inserted in another bitstream changing the functionality of the configuration.

4.2.Work regarding dynamic reconfiguration

Dynamic self reconfiguration is becoming more and more of interest because of the technology vs. productivity gap. Also because of the decrease in production cost for reconfigurable devices like FPGAs. When better design methodology becomes available for hardware software co-design the price of these devices will be further reduced, partly by mass production and partly by competition.

There is much research done regarding run-time dynamic self reconfiguration and some of the work, used in this thesis, will be described here. One of the major contributors is the work done at Jürgen Becker`s Embedded Systems Group [11] at the institute for information processing technology, in the university of Karlsruhe.

A reconfigurable system needs to have special design elements, enabling communication to and from reconfigurable modules. In [10] Michael Hübner et al. describes a LUT based network replacing the TBUF elements previously used, a example with NoC and TBUF bus macros are shown in [12]. The TBUF elements are not available in newer devices like the Virtex-IV and also not as flexible as the LUT based design. The LUT based network are created as hard macros that can be included in designs. By including hard macros the connections are prerouted so all instances of the macro will utilize the same signal wires/routing. This LUT based network hard macro is most commonly known as the Bus macro.

Jürgen Becker et al. have worked a lot with Xilinx Spartan-3 FPGA because of low

price and power consumption. To further decrease the power consumption, partial reconfiguration has been utilized. [13] and [14] explores how dynamic reconfiguration can be used as a method for reduction of power consumption. In [14], Katarina Paulsson et al. shows that using dynamic reconfiguration on a system can reduce the dynamic power usage. The amount of reduction depends on the application, but a result of these measurements is that reconfiguration time is important for the efficiency of the method. The usage of an internal configuration port is therefore highly recommended.

A problem with Spartan-3 is that it does not have a internal configuration access port (ICAP) for self reconfiguration. As a solution to this, a virtual internal configuration access port (JCAP) has been developed [15]. This module uses external pins on the Spartan-3 to connect to the external configuration ports, enabling self reconfiguration. In [16] it is also shown an example of a self reconfiguration access port for Spartan-3. The cPCAP (compressed parallel configuration access port) core design is stated to be able to configure the device at a speed of up to 50Mbyte/s. Usage, extension and first results from implementation of the JCAP design is shown in [17].

With a working system able to partially reconfigure parts of the device, more issues becomes apparent. One problem is the placement and fragmentation that will follow when modules of varying size are used. Also, the priority of modules becomes important when the system is not “large” enough to support the complete system at once. Michael Ullmann et al. proposed a run-time system for dynamic reconfiguration with adaptive priorities in [18]. Here a 1d placement is used for the reconfigurable modules. The 1d model will result in unused area since no module will perfectly utilize the available area of one or more module units. In [19], Michael Hübner et al. proposed a new 2d placement technique. The technique uses a NoC design with router modules spanning the height of the device (Virtex-II) where reconfigurable modules can be placed next to a router module and connect to the router module. Jürgen Becker et al. also described 1d and 2d placements in [20] and also how reconfigurable systems can be used for increased availability, by the redundancy caused by multi-adaptive systems. In [21] Love Singhal et al. proposed a technique for floorplanning in the design steps that reuses components of different parts of the reconfigurable design, reducing reconfiguration time and wire length this can give a improvement in clock period.

Manipulating placement on reconfigurable devices can be a complex operation depending on the device. Much work has been done on manipulating bit-streams for Xilinx FPGAs, [22], [23], [24], [25], [26], [27], [28] and [29] all describes bit-stream composure in more or less detail for Xilinx FPGAs.

One problem with dynamic reconfiguration is debugging the system, as a step towards better debugging of run-time reconfigurable systems Michael Hübner et al. in [30] proposes a on-line visualization of the current configuration on a Xilinx Virtex-II FPGA. The visualizer shows an image similar to the overview image in Xilinx FPGAeditor. This is done by reading out the current configuration, interpret the data and showing it on a VGA output. With this system, it is possible to see placement of modules, fragmentation issues and logic utilization at run-time.

There is also work done on coarse-grained reconfigurable systems, where the reconfigurable modules are on a higher level then what is done with fine grained reconfigurable systems like a FPGA. In [31] Jürgen Becker et al. explores the integration of the general-purpose, Sparc-compliant Leon processor with the

extreme processing platform reconfigurable data path, resulting in a course grained reconfigurable platform.

To be able to easily design and utilize reconfigurable systems the need for modeling and simulation is of utmost importance. In [32] a modified version of the SytemC kernel is used to include reconfiguration in the simulation with software hardware co-design. Florian Dittmann et al. introduces a synthesis methodology in [33] for reconfigurable systems. This system respects the specific requirements of run-time reconfiguration which is vital for the correctness when modeling a reconfigurable system.

It is possible to simplify and generalize reconfiguration of device if the reconfiguration uses abstraction layers. A solution to this is to connect the reconfiguration hardware module to a general purpose CPU, drivers for the module can then be written more generally and be compiled for the CPU used. An example of this is shown in [34], [35] and [36], here different approaches to reconfiguration controlled by a CPU is shown.

Self reconfiguration is not just meant like a technology for the next generation of devices, many devices currently used included reconfigurable devices that can be utilized without physical change. [37] Show an example of how partial reconfiguration can be made available to devices without modifications required with current methodologies.

4.3.Work regarding hardware operating system

The work done in this thesis relies on a operating system running on a CPU controlling the hardware. There are some work done in the area using Linux as a platform for dynamic reconfiguration, [35] shows the simplicity of reconfiguration using Linux as the operating system. The examples shown in [35] are only basic commands and does not integrate in the Linux Kernel for scheduling of hardware tasks in the same manner as for software tasks. For further details concerning scheduling with respect to real time in Linux [38] is a good place to start.

In [39] Rodolf Pellizzoni et al. describes a system where software tasks running on a CPU can coexist with hardware tasks. Also the system incorporates the possibility for hardware tasks to migrate to software and vice versa.

Christoph Steiger et al. discusses design issues for reconfigurable hardware operating systems in [40]. The scheduling, placement and other aspects of reconfiguration is discussed. The runtime scheduler used is shown to have a low overhead. Herbert Walder et al. also discuss an hardware operating system in [41].

The scheduling of tasks is one of the most important aspects in an os. Klaus Danne et al. in [42] explores different schedulers for scheduling real time tasks on reconfigurable hardware.

Methodology

5.1.A collection of theories, concepts or ideas.

The work done in this thesis is mainly experimental work. Most if not all of the methods and theories used in this work are from existing work concerning reconfiguration.

One of the big issues with this work is the complexity. The base system used here is more or less a complete computer with hardware and operating system details to worry about. To get the system to work the hardware has to be set up for reconfiguration with all necessary modules and so on, to get this correct and understand the system takes quite some time.

When the hardware is set up correctly and one has a good understanding of how it is connected, the operating system used (Linux) needs to be configured for the specified hardware. Drivers and test programs for the different specialized modules needs to be created.

One major issue is that one can not be tested without the other. So to test the hardware system and ensure that it is correct the OS, drivers and test programs needs to be working correctly. This is the same for the OS, drivers and test programs to be tested the hardware needs to be available.

With these limitations and uncertainties at different stages the approach taken is to first do a massive literature search. With a deep understanding and recognition of what is done within different areas, it is possible with more or less certainty to understand what will and will not work on a given platform.

As the case with many articles explaining and proposing ideas the implementation is not available. As shown in the previous work section there is done a substantial work in the reconfigurable computing area. The issue is that it is not so simple to get a similar system up and running as the implementation details are not available.

5.2.Comparative study of different approaches.

The approach taken in this thesis are a very focused approach. With a other tasks it is often possible to do a more general study of a system taking choices for the given system. The difference are since the work in this thesis tries to solve a problem with a currently available system, in difference from creating a system to solve the task.

When creating a system to solve a task, a constructive approach, one can do design evaluations for the different stages in the system. For this kind of task it is much easier to do a hypotheses, test and result approach.

In the constructive task where a system is designed to do a task every part of the design can be evaluated and tested with following results that can be summed up for the final product.

With the task in this thesis since a solution are to be given using a existing system it is not possible to reliably test the individual parts separately giving grounds for the complete system. Here all parts needs to function to be able to test the other parts. As an example a Linux driver for a hardware module can not be tested without the module, and the module can not be tested without the driver.

5.3.Critique of the individual methods.

The method used in this project is focused around experimentation. Understanding what is done and how it is done, then try to do the same. This results in much work trying to understand the undocumented features and getting functionality not related directly to the work but essential for getting results.

To get the base system running cannot be done in much different ways. It is straight forward get the data, code the algorithms and test the system.

To be able to propose a framework for a self reconfigurable system on a Xilinx FPGA with ICAP (internal configuration access port) there are first a couple of design points that needs to be tested and proven. If these 3 points are not confirmed to be working with each other the system and framework can not be proven to be correct.

1. A hardware design needs to be created for the FPGA including connection to the ICAP and a reconfigurable module connected to the CPU with bus macros.
2. The bitstream for the given FPGA needs to be understood, addressing algorithms in the bitstream needs to be developed and a program to "cut and past" from the bitstream are needed.
3. Drivers for accessing the ICAP and reconfigurable module needs to be developed for the Linux kernel running on the plattform.

With all these points working together one has a self reconfigurable system. With the base system it is simpler to test further designs.

The main method of work in this thesis is to understand what is done how this is done and then implement this on the Suzaku-V platform.

The system has not been evaluated completely from the beginning, trying to make a design spec of the system from start without knowing all aspects are difficult at best. But without a very detailed design goal it is easy to get stuck when problems arises.

Theory

6.1.Suzaku-V

The Suzaku-V, board computer, is based on the Xilinx Virtex-II Pro or Virtex-IV fx FPGA. These FPGA's include a hard-core processor “PowerPC405” and also additional peripheral cores on the FPGA. As default, the Suzaku-V uses Linux as operating system.

The configuration that the system arrives with, can be built with the Xilinx EDK (Embedded Development Kit). It is also possible her to customize the configuration, if necessary.

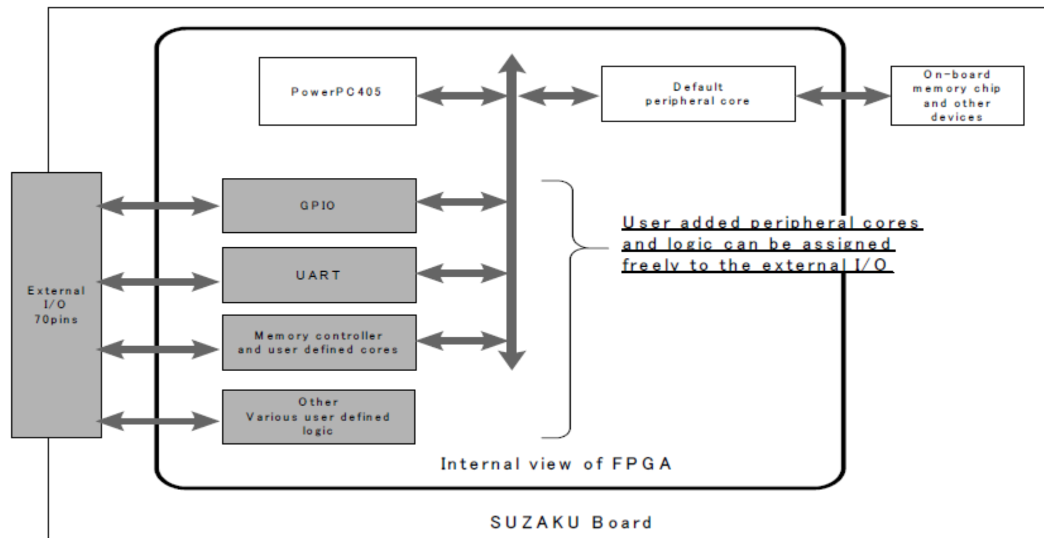

Figure 7: Suzaku Board configuration overview

Figure 7 shows the Suzaku platform original setup and where the platform can be expanded. The platform incorporates 70 external I/O pins that the user can use.

For communication the platform includes LAN and serial communication. Linux includes all necessary protocols, so setup and usage of the different communication methods are quite simple. More detailed information can be found in [43] and [44].

By using a board computer that is configured with peripherals and a working Linux kernel my work is simplified. Setting up a complete system and getting all aspects to work, is a major task and not really possible in the short timespan available. A minus with the Suzaku platform, is that the FPGA used is very small and expansion possibilities are limited.

6.2.Xilinx Virtex-IV Configuration Control Logic

The Virtex-IV FPGA integrates a package processor controlling the configuration logic. By writing commands and data to the FPGA over the configuration interfaces (SelecctMAP, JATAG, Serial), the package processor interprets the commands and uses the data to configure, analyze or read out the configuration of the device. The registers accessed by the package processor controls all aspects of configuration.

Configuring the FPGA is done with two types of packages, sent to the package processor, Type1 and Type2.

A type1 packet is used for register reads and writes. In Virtex-4 FPGA only 5 out of 14 register address bits are used. After a Type1 packet a Data section follows, the data sections consists of a sequence of 32 bit words, specified in the Type1 package.

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Table 1: Type1 Packet header format. “R” means reserved for further use.

A type2 packet must follow a Type1 packet and extends the amount of data that can be written. No addressing is done in a Type2 packet as this is done in the Type1 packet before the Type2 packet. The data following a Type2 package has the amount of 32 bit words specified in the Type2 header.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Table 2: Type 2 packet header. "R" means reserved for further use.

Opcode	Function
00	NOP
01	Read
10	Write
11	Reserved

Table 3: Opcode format

There are 14 registers accessible from the package processor, shown in Table 4. These registers are written or read from, to configure the FPGA.

Reg. Name	Read/Write	Address	Description
CRC	Read/Write	00000	CRC register
FAR	Read/Write	00001	Frame Address Register
FDRI	Write	00010	Frame Data Register, Input (write configuration data)
FDRO	Read	00011	Frame Data Register, Output register (read configuration data)
CMD	Read/Write	00100	Command Register
CTL	Read/Write	00101	Control Register
MASK	Read/Write	00110	Masking Register for CTL
STAT	Read	00111	Status Register
LOUT	Write	01000	Legacy Output Register (DOUT for daisy chain)
COR	Read/Write	01001	Configuration Option Register
MFWR	Write	01010	Multiple Frame Write
CBC	Write	01011	Initial CBC value register
IDCODE	Read/Write	01100	Device ID register
AXSS	Read/Write	01101	User bitstream access register

Table 4: Configuration Registers

The FAR register is of special interest, as addressing on the FPGA is used when writing modules to the FPGA.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 x x x	x x x x	x x x x	x x x x	x x x x	x x x x

Reserved	T/B	Block	Row	Column	Minor

Table 5: FAR register composition

Top/Bottom (bit 22)	Sets if top or bottom of FPGA is addressed. 0 = Top, 1 = Bottom.
Block Type	CLB/IO/CLK = 000 BRAM interconnect = 001 BRAM Content = 010 (CFG_CLB 011 CFG_BRAM 100)
Row Address	Selects a row of frames. Addressing starts at center of FPGA and increases outwards, defined by Top/Bottom bit.
Column Address	Addresses columns of given block type. (Major address).
Minor Address	Selects a minor address within major address, (frames in a CLB).

Table 6: FAR addressing details explained.

The full configuration description and register description can be found in [45].

6.3. Xilinx Internal Configuration Access Port (ICAP)

The Internal Configuration Access Port (ICAP) gives access to the FPGA configuration, in the same manner as the SelectMAP. ICAP uses the same interface signals as SelectMAP, apart from the bus. ICAP uses a separate bus for read and write signals.

The bus width of ICAP can be configured to 8 or 32 bits, this is the same as SelectMAP and SelectMAP32.

For Virtex-IV devices there are two possible sites for the ICAP: TOP and BOTTOM. These connections uses the same underlying logic, the difference is the location on chip and the interconnect they can be connected with. For future info refer to [45].

6.4. Runtime self reconfiguration

Runtime self reconfiguration are simply put: the concept of something changing its function by it self, while running. To be able to change the functionality, the base design needs to be configurable. An example on a reconfigurable system is the FPGA (Field Programmable Gate Array). A FPGA can be configured to do any number of logic functions, by its LUT (Look Up Table) design. These LUT's can be reprogrammed and the interconnects connecting the signals can be reconfigured any number of times. Some of the FPGA designs available can reconfigure parts of the FPGA, wile the rest is running.

Runtime reconfiguration can result in less hardware needed for some designs, tasks can be time sliced, instead of always available but not running.

Runtime reconfiguration can also result in longer lifetime of products, bugs can be fixed, protocols can be updated and new services can be made available.

6.5. Linux µClinux ATMARK-dist

In this thesis much work are done around the Linux kernel, with the simplifications and complications the Linux system gives. The Linux kernel in it self is not of much use without a basic system, utilizing the kernel. In the same way as Ubuntu is a distro for desktop computers, collecting a big sett of user programs, setting up and making the system work; ATMARK-dist is a distribution for embedded systems.

ATMARK-dist builds on the µClinux-dist which is a lightweight operating system, first built for processors without MMU but now support a wide range of processors. µClinux-dist can even be used on a x86 processor.

The "µClinux-dist" software package contains libraries, applications and tools. It can be configured and built into a kernel with root file system. It was first released by Greg Ungerer in 1999 as the µClinux-coldfire package. In the following years it came to support many architecture families, and now can even build standard Linux architectures (such as x86) as well.

The "µClinux-dist" userland utilities contain tiny http servers, a small 'sh like' shell, and even a fun ASCII art Star Wars film. It also contains many other well known Open Source packages, like Samba and FreeS/WAN, all of which run on µClinux systems. -From WWW.absoluteastronomy.com/topics/UClinux

When developing and crating programs for the Suzaku platform, the programs are included in the μClinux image and not compiled into the Linux kernel. Of course the Kernel and distro goes hand in hand and the programs are dependent on the Kernel. But the distinction can be important, to understand how the system works.

Understanding how the complete system work are of utmost importance in this project.

6.6. Virtex Bitstream design

6.6.1. Virtex II

The configuration bitstrem configures the device, setting all the interconnects and logic used in the design. The VirtexII device series configuration bitstream, consists of frames spanning the total height of the device. So for all devices, the frame length depends on the number of rows with logic (the hight of the FPGA logic). An example of the logic placements are shown in Figure 8, for the XC2VP4 FPGA. This FPGA contains a hard core powerpc, the black box in the figure. The powerpc will not have an impact on the frames in the reconfiguration bitstrem. The configuration bits for logic where the powerpc is, are included in the bitstream but they are not used (making the bitstream consistent). In Figure 8 the IOBs are highlighted (red), these surrounds the logic on left, right, top and bottom side. The BRAM modules can be seen in Figure 8 (blue), are placed in columns on the FPGA.

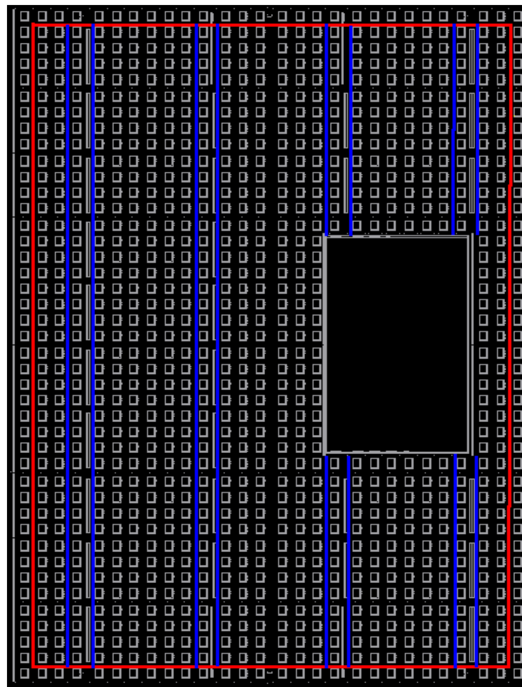

Figure 8: VirtexII XC2VP4 logic modules illustration

The configuration bitstream is organized with first the initializing commands, for the FPGA, then the configuration frames and finally the finishing commands, (starting the FPGA). The frames organization, in the configuration frames section, in the bitstream is shown in Table 7. Using Table 7, one can find the frame location in the bitfile, for a specific logic element. Each CLB column consists of 22 frames, stretching from top to bottom of the device. On the XC2VP4 there are 22 CLB columns, resulting in 484 frames for CLB logic. Table 8 and Table 9 shows the frame composition for CLB and BRAM logic. Using this information it is possible

to locate where in a frame a certain logic element is located.

Frame type:	GCLK	IOB	CLB+IOB	IOB	BRAM
Frames:	4	26	484	26	344

Table 7: Frames and location in bitstream, bitstream location start to end, left to right. For XC2VP4

Logic type:	Pad/CLK	IOB	CLB X_aY_{max}		...	CLB X_aY_0		IOB	Pad/CLK
Bits:	16	80	40Slice	40Slice	...	40Slice	40Slice	80	16

Table 8: CLB logic frame format, from left to right are top to bottom on FPGA and first to last in bitfile.

Logic type:	Pad/CLK	Top DCM	RAM X_aY_{max}	...	RAM X_aY_0	Bottom DCM	Pad/CLK
Bits:	16	80	320	...	320	80	16

Table 9: BRAM logic frame format, from left to right are top to bottom on FPGA and first to last in bitfile.

6.6.2. Virtex IV

The VirtexIV architecture is different from the VirtexII architecture. Unlike the VirtexII platform, where the configuration frame size depends on the device size, VirtexIV configuration frames are equal for all devices. On all VirtexVI devices one frame is 16 CLBs, which gives the smallest reconfigurable unit on a VirtexIV to be 16CLBs. Figure 10 Shows how the logic modules are organized on the Virtex-4fx12 FPGA.

Figure 9 shows the logic distribution on the Virtex 4FX12 FPGA, including the hard core powerpc located on the left side. In Figure 9 IOB's are read, BRAM are blue and DSP are marked as green. The frames in the bitfile located where the power pc is will not be truncated.

The configuration is different from the VirtexII in many ways, for one: there are no IOBs on the top and bottom row. Instead VirtexIV has extra IOB columns on the device, (one extra collumn for the Virtex-4fx12 FPGA). In the configuration bitstream the data is organized as shown in Figure 11. A note to Figure 11 is that the overhead comes after all rows, so row 0 shown in Figure 10 comes first in the configuration bit stream. Then follows row 1, 2 and 3 then the overhead frames. Figure 11 Shows that BRAM interconnect are separated from the BRAM data, this is an important point. The DSP frames are not separated like the BRAM interconnect and data in the bitfile.

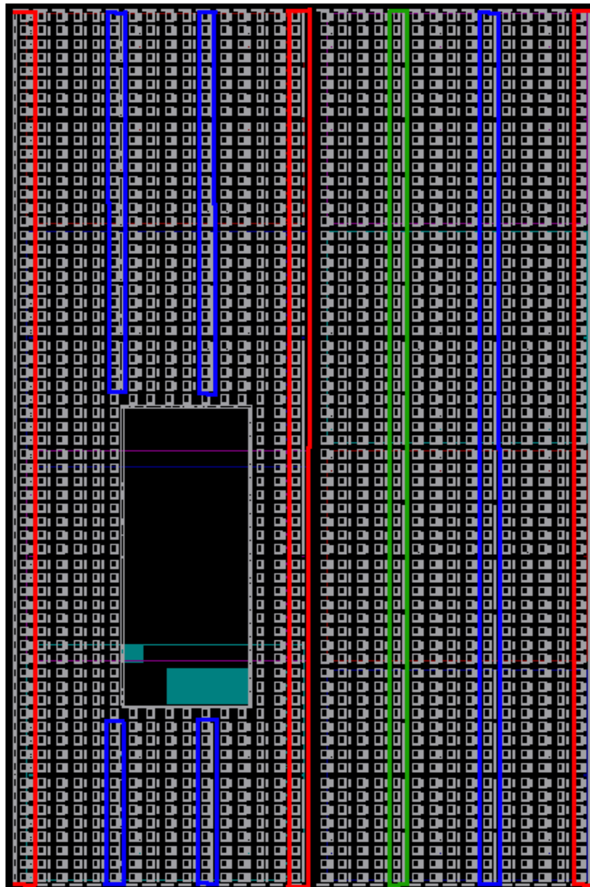

Figure 9: VirtexIV 4FX12 logic module distribution.

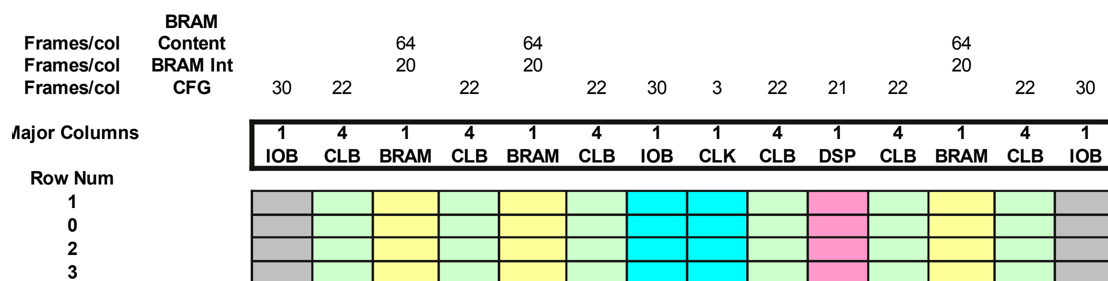

Figure 10: Virtex-4fx12 FPGA logic modules

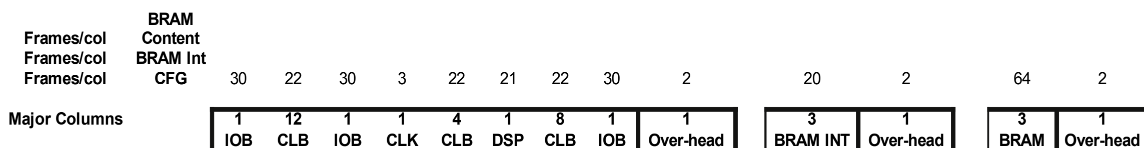

Figure 11: Virtex-4fx12 bitfile frame organization

In the bitstream the CLBs consist of 22 frames, 20 of these are for the routing and 2 are for the logic. The DSP also has 20 frames for routing and 1 for logic. From Figure 11 one can also see that BRAM interconnect consists of 20 frames. These frames can be interchanged, as the interconnect for CLB, DSP and BRAM are equal.

If logic is to be read out from the bitfile, the routing in the BRAM interconnect or

DSP routing has to be included. If the logic in the DSP or the data in the BRAM is not used a module placed at a location utilizing the DSP interconnect can be moved to a location with BRAM interconnect, this results in more processing of the bitfile when reconfiguring.

A frame in the bitstream is 41 words, 1312 bits, this is equal to 16 CLBs plus one word in the center of the bitstream for global logic. The configuration bitstream bits are all relative to the center of the FPGA. So the bottom half is a mirror image of the top half, minus the center word which is for the global routing, this word is symmetric by design.

6.7.Suzaku-V setup

6.7.1.Network File System

When working with the Suzaku platform, experimenting with code, it is time consuming to upload a new Linux image with the program/module each time modifications are made. This can be simplified by using a network file system (nfs). By using a nfs one can instantly share the program compiled with a cross-compiler, on a host machine with the Suzaku platform. The Linux configuration for the Suzaku-v platform in the **atmark-dist-20080717** distribution is set up to including nfs support. Only modification one might want to do is to remove the **dhcp-new** module, in the vendor configurations, to use a static IP on the Suzaku. This is explained in the howtos on Atmark Techno's web-page (http://suzaku-en.atmark-techno.com/dev/howto/software). Here booting from a nfs is also explained, this is smart if experimenting with modifications in the kernel.

A nfs-server needs to be running on the host, setup of this depends on the distro used. The best way to get it correctly configured is to find a tutorial for the given distro (google is a good tool here). The Linux-box I used as host, used Archlinux and the nfs setup tutorial (http://wiki.archlinux.org/index.php/Nfs) were used to configure the nfs-server.

When the server is running and connection with the Suzaku is established, (it is possible to ping the Suzaku, from the host and the host from the Suzaku) the nfs can be mounted with the following command:

mount -o nolock -t nfs server:/location/ /mnt/

This mounts the server (change to ip address) /location/ (change to path of nfs share) to /mnt on the Suzaku. /mnt can be changed to another folder on the Suzaku. Now it is possible to access files shared on the nfs, and store files if the server is configured for it.

This is also useful for kernel drivers built as modules, making it possible to change drivers without uploading a complete new kernel. The kernel needs be built with module support for this to be possible.

6.7.2.Adding hardware to the Suzaku-V FPGA design

The Suzaku-V comes with the base design project for Xilinx EDK. Using EDK it is easy to extend the project, using modules either from Xilinx or modules created by the user. The Suzaku-V 410 board, using the Virtex-IV FPGA, only uses the PLB bus. So modules added that needs to communicate with the PPC, needs to use the

PLB bus.

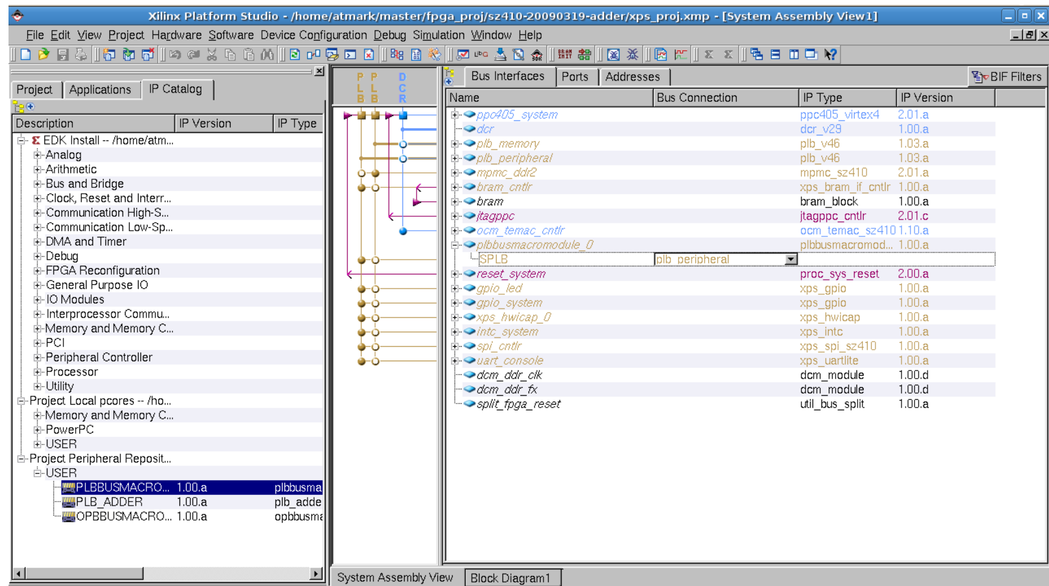

Figure 12: EDK screenshot with the Suzaku-V fpga_project. With a user module added.

To add a module to the fpga_project, for the Suzaku-V platform, is quite simple. The screen shot of EDK shown in Figure 12, shows the IP catalog on the left side and the current configuration and bus connections on the right side. To add a module from the IP catalog is as simple as a right click and chose **Add IP**. The module will now appear on the right side, with the other modules in the design. But the module will not be connected to anything.

On the Suzaku-V platform, with the Virtex-IV FPGA, the module should connect with the PLB bus. In the fpga_project for the Suzaku the plb_peripheral bus are available for peripherals. To be able to address the module, from the PPC, the module address range needs to be set. This is done under the Addresses tab in EDK shown in Figure 13. Here the base address and size can be set, (high address will be calculated from base address and size). The address given to a module included in the fpga_project, for the Suzaku-V platform, should be check to not conflict with any other modules. The addresses of the other modules can be seen in the same window and are also included in [43], under memory map.

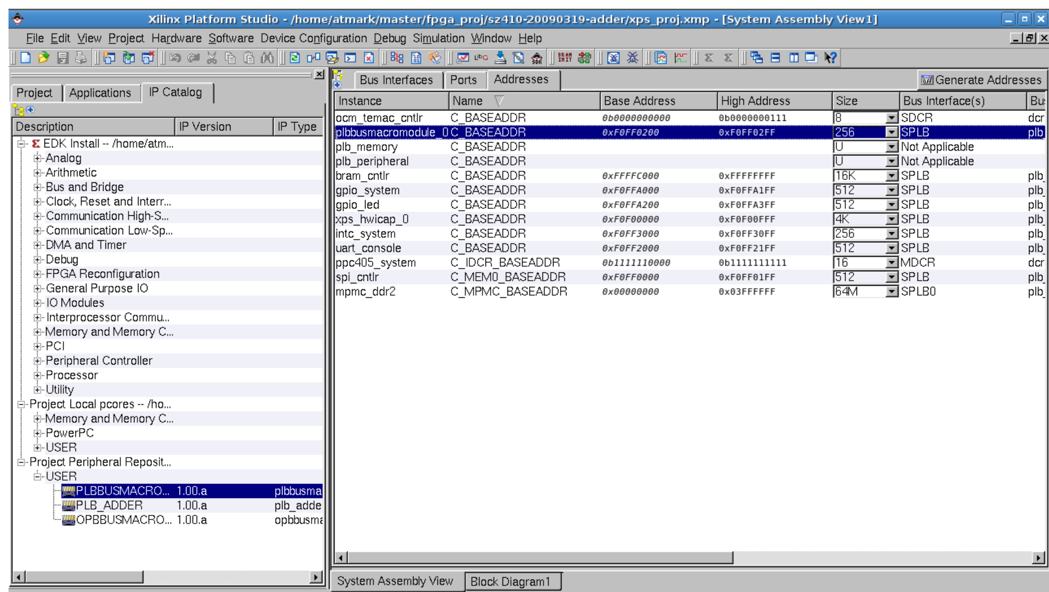

Figure 13: EDK screen shot showing address range, for modules in the Suzaku-V fpga_project

The address given the module, is needed when the module is accessed from software.

6.7.3.Accessing hardware on Suzaku-V with Linux 2.6.x

The hardware modules connected to the PPC on the Suzaku-V, when running a Linux kernel, can not simply be accessed using a pointer to a given address. It is possible to access the modules in user space, but this is not recommended, the best way is to create a device driver. The driver should take care to restrict access, so only one process reads or writes to the device . Implementing a char driver, the driver will work as a file, where the usual file functions open, read, write and close will work on the driver. The Linux device drivers [46] book, is a good reference when writing device drivers for Linux.

Using a driver for accessing hardware modules, are a smart solution. The user space program does not need to know how the hardware works, or how to access it. The driver can just be opened as a file and written/read from as necessary. An example driver used in this project is included in the .zip file, (under /code/linux_drv/Adder/).

The example driver module can be compiled with the following command:

make -C ~/kernel-2.6 M=`pwd` modules

Where kernel-2.6 is the build directory for the kernel, for the Suzaku this is the atmark-dist folder, used to build the Linux image the module will be used against. Pwd gives the current folder, where the module is.

6.7.4.Adding ICAP hardware module to Suzaku-V FPGA design

In Xilinx EDK, there are two HWICAP modules that can be included in the design. The modules might not show with standard configuration of EDK, if the modules does not show in the IP catalog they need to be included. This is done in preferences, under IP catalog and IP config dialog, cross of Display “Available” IP cores (including legacy PLB/OPB cores) in IP catalog.

The HWICAP modules should be under the FPGA reconfiguration tab, in the IP catalog. There are two modules, one connecting to the OPB bus and one connecting

to the PLB bus. For sz410 only the PLB bus is available, so the xps_hwicap module is used. The xps_hwicap needs 4K of address space, so ensure that the chosen base address and high address does not conflict with other modules. The memory map can be found in [43].

Xps_hwicap module uses the ICAP found inside Virtex-4 and Virtex-5 devices. The ICAP port interface is similar to the SelectMAP interface, but is accessible from general interconnects rather than the device pins. The JTAG or "Boundary Scan" configuration mode pin setting (M2:M0 = 101) will disable the ICAP interface. Therefore, when using the HWICAP core, another mode pin setting must be selected to avoid disabling the ICAP interface. JTAG configuration will remain available because it overrides other means of configuration, and the HWICAP core will function as intended. Besides being disabled by the Boundary Scan mode pin setting, the ICAP will also be disabled if the persist bit in the device configuration logic's control register is set. When using bitgen, the Persist option must be set to No, which is the default. -From [47].

In the Suzaku fpga_project project files the bitgen.ut file was changed for M2:M0 settings, Here M2:M0 were sett to M2:M0=110 which is Slave SelectMAP. The JTAG/Boundary-scan configuration interface is always available, regardless of the MODE pin setting. The JTAG/Boundary-scan configuration mode (M2:M0=101) disables all other configuration modes. This is to prevent conflicts between configuration interfaces.

With the standard configuration and the xps_hwicap module the design should take approximately 90% of the resources on the FPGA, when using the Suzaku-S sz410. When the modules are not restricted in placement the modules will be placed as best the placer can, but it is not possible to know where the modules are. So with this configuration it is possible to test read-back of configuration on the FPGA, but one should not try to program the FPGA since it is not possible to know what is overwritten.

To use the HWICAP to write modules to the FPGA, the basic/static configuration needs to be constraint in placement. So it is known where it is possible to write new modules, without overwriting the basic configuration.

Figure 14 is a screen shot from planahead, a tool from Xilinx, with this tool it is easy to create constraints for the design. From planahead it is possible to export the constraints, for inclusion in the design. This is important as the design needs to be run through **bitinit** to initialize the ram of the CPU with Bboot, for the Suzaku platform, else it will not boot.

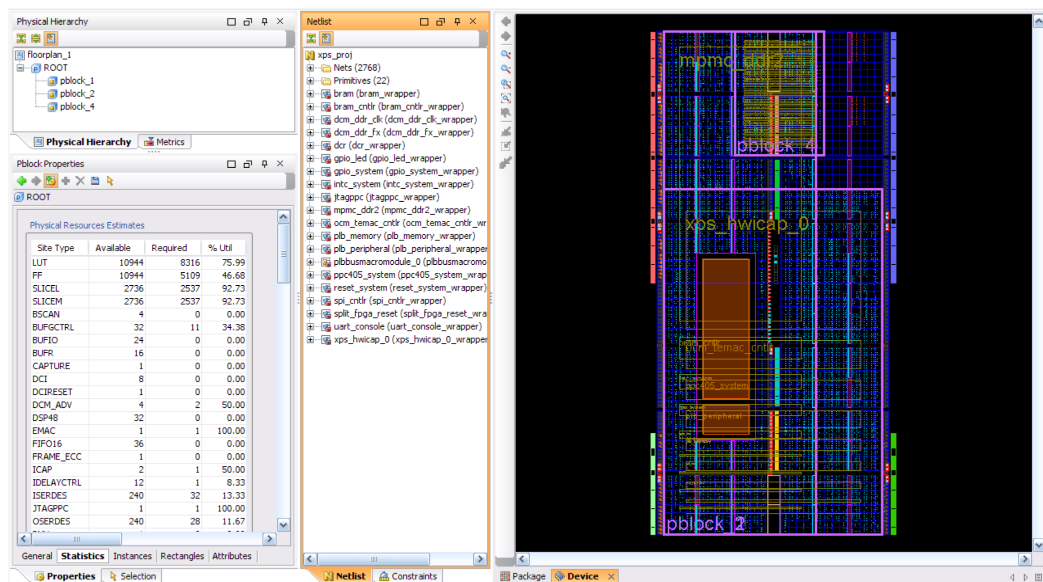

Figure 14: Planahead screen shot with placement restrictions and resource usage shown.

In Figure 14 a reconfigurable module is included, the module is placed in the top right corner. The bus-macros connecting the module to the PLB bus can be seen as red dots in the top right corner. Here there are no placement constraints on the logic in the reconfigurable module connected to the bus-macros. Since the connected logic is small and will be placed next to the bus-macros, but constraints could be created if necessary.

Place and route can be done in planahead and it is nice to run to ensure that the constraints are possible. But to ensure that everything follows the steps necessary for the Suzaku platform, the constraints placed were exported and used in the Suzaku project in EDK.

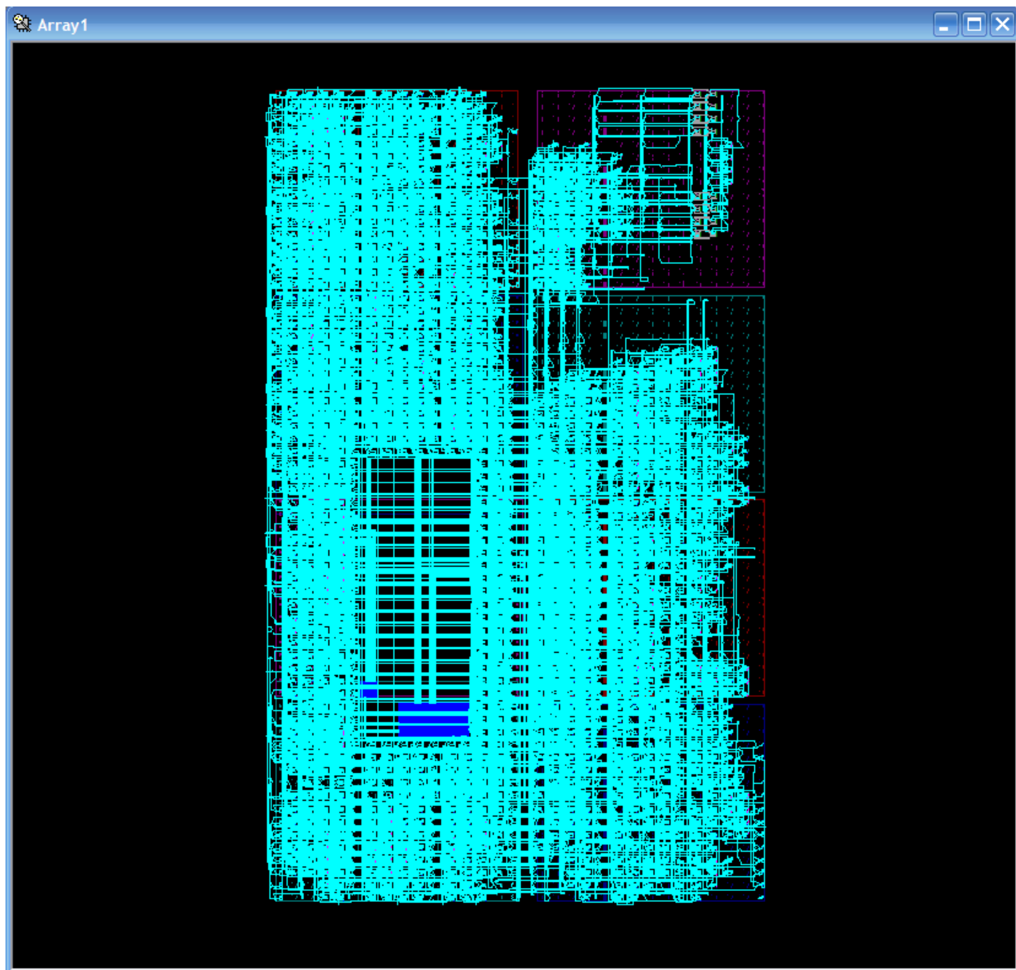

Figure 15: Screen shot of routed design with hwicap module and reconfigurable module in top right corner

In Figure 15 the routed design with the constraints seen in Figure 14 are shown. In Figure 15 one can see that the placement constraints does not constraint the routing, only the logic. So for the reconfigurable module area one has to ensure that no routing crosses the module border. If any routing crosses the boarder this needs to be rerouted, easily done in fpga editor.

Figure 16 shows the reconfigurable module in more close up. Her one can see the bus-macros placed and also that no routing crosses the the middle of the bus-macros but the bus-macro routing. The last three CLBs with the full height of one frame (16 CLBs) can be reconfigured in this module.

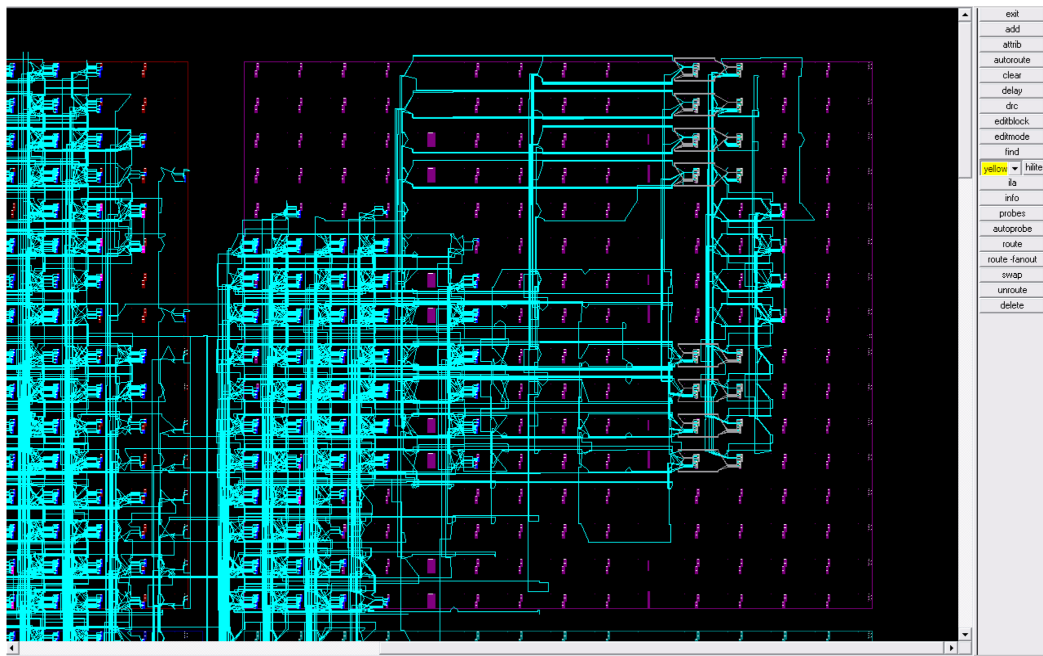

Figure 16: Reconfigurable module with busmacros in top right corner of fpga

6.7.5.Getting HWICAP driver to work for Linux kernel 2.6.18-at11

The internal configuration access port (ICAP) can be connected with a soft or hard core microprocessor, using the HWICAP module supplied from Xilinx. The HWICAP module connects the ICAP to a bus so the microprocessor can communicate with it. To be able to use the HWICAP module from Linux, the driver for the module needs to be included/built for the kernel.

The Xilinx HWICAP driver was included upstream in the Linux kernel from 2.6.25, but it is not included in the Linux kernel that are included with the Suzaku platform, (kernel used here is the 2.6.18-at11). The driver used in this example is from the Linux kernel 2.6.28.7 but the driver included in the 2.6.29 kernel is identical. To get the driver to compile, using the 2.6.18at11 kernel headers, some files needs to be patched.

The “include/linux/cdev.h” file needs to be patched:

The following is the diff between 2.6.18at11 original and 2.6.18at11 modified kernel.

```
diff linux-2.6.18-at11/include/linux/cdev.h linux-2.6.18-at11-
mod/include/linux/cdev.h

4a5,12

> #include <linux/kobject.h>

> #include <linux/kdev_t.h>

> #include <linux/list.h>

>

> struct file_operations;

> struct inode;

> struct module;

>
```

Source 1: Diff for cdev.h in the Linux kernel. Patch to get the HWICAP driver to work.

The patch shown in Source 1 can be found on the kernel mailing list. After using this patch the xilinx_hwicap module should be able to compile. To compile the module as a module outside the kernel build system some modifications is needed to be done on the Makefile for the module. Modifications shown in Source 3.

There are two HWICAP modules in EDK IP library, one for the OPB bus and one for the PLB bus. On the Virtex-IV there are only PLB bus available, so the PLB module needs to be used. In the Linux driver there are two configurations, one for the OPB bus module and one for the PLB bus module. Since the OF PLATFORM system is not supported in the 2.6.18 kernel, the configuration is not automatically selected. In Source 2 the patch to support the HWICAP xps PLB bus module is shown. Also the patch to downgrade the device_create() call to the kernel 2.6.18 version are shown here.

```
diff modified/xilinx_hwicap.c original/xilinx_hwicap.c

677c658

< device_create(icap_class, dev, devt, "%s%d", DRIVER_NAME, id);

---

> device_create(icap_class, dev, devt, NULL, "%s%d",
DRIVER_NAME, id);

759c740

< &fifo_icap_config, regs);  // Changed by sverre to fifo for
the xps hwicap module

---

> &buffer_icap_config, regs);
```

Source 2: Patch to get the driver to support the xps PLB bus module, and patch the device_create() function call.

When the driver is loaded the driver will ask the kernel to connect to a specific device, the HWICAP module on the bus. For the driver to be loaded, the kernel needs to know if the device is connected to the bus. The device instantiation is normally done in the platform section of the Linux kernel, as this will be specific to the platform. The device instantiation sets the address of the device and the driver name, for the driver that can use it. To simplify the process I have included this instantiation in the driver instantiation shown in Source 4.

```
diff modified/Makefile original/Makefile
4,6d3
< ifneq ($(KERNELRELEASE),)
<       #obj-$(CONFIG_XILINX_HWICAP) += xilinx_hwicap_m.o
<       obj-m += xilinx_hwicap_m.o
8,17c5,7
<       xilinx_hwicap_m-y := xilinx_hwicap.o fifo_icap.o 
buffer_icap.o
<
< else
<       KERNELDIR ?= /home/atmark/project/suzaku-v/linux/atmark-
dist-20090318
<       PWD := $(shell pwd)
<
< default:
<       $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
<
< endif
\ No newline at end of file
---
> obj-$(CONFIG_XILINX_HWICAP) += xilinx_hwicap_m.o
>
> xilinx_hwicap_m-y := xilinx_hwicap.o fifo_icap.o buffer_icap.o
```

Source 3: Diff for xilinx_hwicap driver Makefile. To be able to compile module outside kernel build system.

```
diff modified/xilinx_hwicap.c original/xilinx_hwicap.c


115,124d111

< struct platform_device *icap_dev;

<

< struct resource icap_res = {

<    .start = 0xF0F00000,

<    .end   = 0xF0F00FFF,

<    .name  = DRIVER_NAME,

<    .flags = IORESOURCE_MEM,

< };

<

<

875,876d855

<        icap_dev = platform_device_register_simple(DRIVER_NAME,
0, &icap_res, 1); // register hwmodule added by Sverre

<
```

Source 4: Patch for device instantiation, setts the address of the device as specified in EDK.

With the driver patched for the 2.6.18-at11 kernel the module should compile. If the Suzaku platform has been configured to be modular and have nfs, the module can be tested without uploading a new Linux image to the platform. Before initializing the kernel module a node needs to be created in the /dev file system that the driver can be associated with. This can be done with the following command:

mknod /dev/icap c 259 0

This creates a "file" in the dev directory associating a char driver, with major number 259 and minor number 0. The major and minor numbers needs to be the same as the driver is instantiated with in the code. The name also needs to be the same as the driver name, which is the same name as the device uses to associate with the driver.

When all this is done the driver can be instantiated with the **insmod** command.

6.7.6.Using the HWICAP Linux drivers

The HWICAP driver simply reads from or writes to the HWICAP module, the driver does not take into account setting up the FPGA for read out or write in. Setting up the FPGA for read out or write in of frames are meant to be done in user space.

When the icap driver is instantiated, using it to write a configuration can be done by simply piping a bitfile to the device. This can be done with the following command: **cat bitfile.bit > /dev/icap** (not tested). Here the bitfile needs to contain the setup commands for the FPGA, the frames to be written and the closing commands to the FPGA.

To read out data from the icap driver one first has to write some commands to the icap driver to initiate a read back operation. The following read back configuration details can be found in [45].

1. Write the synchronization word to the device.

2. Write 1 NOOP command.

3. Write the RCFG command to the CMD register.

4. Write the starting frame address to the FAR.

5. Write the read FDRO register packet header to the device. The FDRO read length is given by:

   FDRO read length = (words per frame) x (frames to read + 1) +1

   One extra frame is read to account for the frame buffer. The frame buffer produces one dummy frame at the beginning of the read and one at the end.

6. Write two dummy words to the device to flush the packet buffer.

7. Read the FDRO register. The FDRO read length is the same as in step 5.

8. Write NOOP instruction.

9. Write DESYNCH command.

The register descriptions are described in the FPGA configuration section.

The test program created for usage of the ICAP Linux driver are somewhat modified from the steps here. After some tests the driver seemed to work correctly only when one or two frames where read, (this is most likely a bug in the test program). Also more NOOP commands where used between FPGA commands. Following the driver example files in EDK for HWICAP driver (these example codes are for running directly on the processor without a OS).

My work

7.1. Difficulties with dynamic self reconfiguration on FPGAs

As expressed in earlier work [9], the complexity of a complete reconfigurable system consisting of reconfigurable logic, a CPU and a OS running on the CPU is large. To be able to test a hypothesis or theories, by ones own making or from the research community takes a great deal of work.

Also the issue with available implementation from the research community is an issue:

Even though all the key ideas are available in a paper, re-using

the ideas in such a document takes a lot more time than working

with the software directly. You can reuse software without fully

understanding it, but you can't re-implement software without fully

understanding it! - From [48].

In the work done here, the underlying reconfigurable hardware is FPGAs from Xilinx, these devices supports reconfiguration. The problem with the FPGAs from Xilinx is that even though the hardware have supported run-time reconfiguration a long time, the software does not fully support it. Much of the details about reconfiguration is poorly documented, if at all. So to be able to utilize the reconfiguration possibilities, on the devices, in a new and innovative way new tool sets needs to be developed.

7.2. My work relative to previous work

In Work regarding dynamic reconfiguration on page 9, a lot of background work has been summarized. Most of the work explained is of interest and the work done here tries to realize some of the concepts proposed for the FPGA device used in this project.

For the reconfigurable module, the bus macros proposed by Jüergen Becker et al. in [10] are used. The bus macros and some design ideas used in this work are proposed in [49], where Xilinx early access tools are described. If other bus macro designs than the ones gained from Xilinx early access program are needed, then a more thorough explanation of the bus macro can be found in [2].

There are many articles describing reconfiguration of Xilinx FPGAs, but not all uses manipulation of the bitstream and very few of the ones that uses bitstream manipulation describes the algorithms, or design in details. For Xilinx FPGAs [28] explains more in detail bitstream design and how modules can be relocated in a bitstream.

In the work done here the bitstream of both Virtex-II and IV has been examined. The concept of reading out modules from the bitstream and writing in the module in another design has been tested and proven. Relocation has not been possible to test because the platform and device used were not capable, (there were not enough logic on the device for two reconfigurable modules).

Using Linux to simplify dynamic reconfiguration as done in [35], is shown to work in this project. The reconfiguration is done by using the ICAP drivers in Linux to read out and write in partial design modules. The scheduling and placement programs running under Linux are just for proof of concept, these areas are well researched in OS theory. Placement algorithms can use similar concepts as for virtual memory since this is a similar concept.

7.3.Design of reconfigurable modules

The hardware modules that are to be used as reconfigurable modules in this framework needs some extra considerations:

- Number input/output signals.
- Bus-macro version.
- Bus-macro relative position.
- Size limited, logic and routing.
- Swappable.
- Clock demands.
- I/O

The reason for these considerations comes from the way the system is designed. The framework here is based on manipulation of the bitstream for configuration of the FPGA.

A bitstream is the configuration bits for a FPGA and the lowest level one can go when manipulating hardware modules.

In the bitstream the logic are placed and routed, in the framework design used here it is not possible to reroute signals by manipulating bits in the bitstream.

In the static design, where the reconfigurable module will be connected, connections are prepared for reconfigurable modules. These connections are based on CLB slice based bus-macros. The bus-macros are prerouted hard macros that will not change the routing when running place and route tools. These bus-macros can thus be viewed as sockets that modules connect to.

The sockets in the module needs to be of the same type, have equal size and placed at the same position relative to each other as in the static design.

Since the static design is a general design, the number of inputs and outputs to a reconfigurable module will be set to a given number and all reconfigurable modules will have this constraint. Because of this the back-end to the modules will shift data words in and out of the module serially, implying that a serial data flow controller needs to be implemented in the module, at least if more then one word of the defined length is needed as input or output.

Module size is an important factor with height, width and position of bus-macros. The maximum size a reconfigurable module can have, will be defined by the FPGA and the static design where the module will be loaded. Needless to say this can wary as there are different sized FPGAs, but as long as the static design has the same version bus-macros and same relative position of these A small module can work on a small, medium and large system while a medium module will only work on a medium and large system and so on. Given that modules are designed in defined size

increments, not free size.

Depending on the static design, it is possible to decide that a small module is the smallest design that can be inserted and all other sizes are defined by a number of small modules. In this sense a small system can have one module and a medium system can have two small modules or one medium module. If designed properly this also gives the possibility to increase the amount of inputs and outputs from the module when sizes increase. A bigger module can use two or more set of connections, where the connections are designed for one small module.

If the module is to be swappable while it is working, the module needs to be designed for it. For a module to be swappable, the state of the module and data it is working on needs to be stored before the module is removed. So a swappable reconfigurable module needs to incorporate a way to flush the state and data it is holding and also a way to restore the data and state to continue working on it, when reinserted.

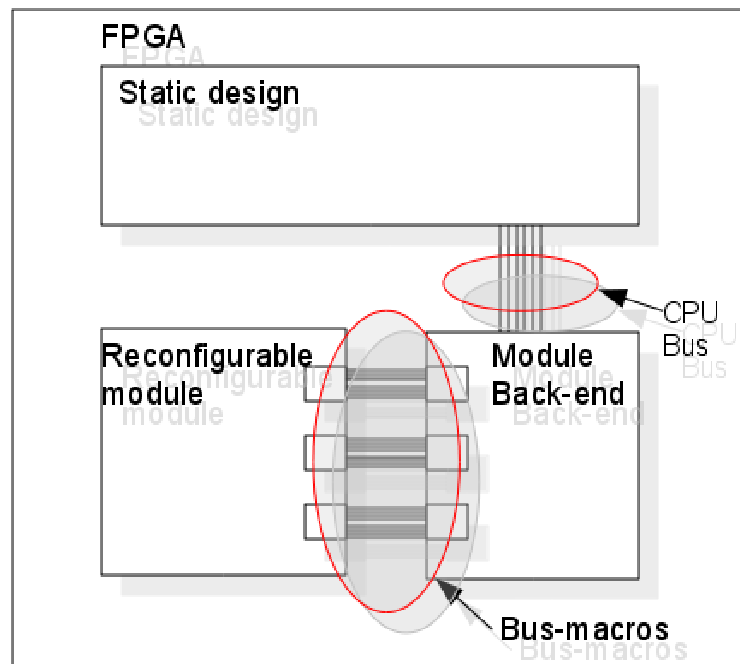

Figure 17: System overview

Reconfigurable modules can be inserted in different systems and this can potentially lead to modules being designed with a lower clock speed constrain and thus not function properly when instantiated. A guard for this shortest delay issue is to include, in the repacking of the module, speed constraints for the module. These constraints will have to be checked by the static design before the module is inserted into the system. Further enhancements could be to include a clock divider in the module back-end, resulting in a possibility for varying clock speed for modules.

Physical inputs outputs to the system is also an issue to tackle with reconfigurable modules. There are different possible solutions to the I/O case.

One solution can be to create a I/O module that connects to the I/O pins one needs and then connect this to the processor, controlling reconfiguration, letting the processor control access. Thus reducing it to a software problem.

The software I/O component can be removed if the reconfigurable module back-end has a network on chip architecture and direct communication between modules can be achieved.

Another solution is to create a reconfigurable module that has the inputs/outputs in the module, this of course means that the module can not be relocated.

7.4.Procedure/method for generating reconfigurable modules

Design and generation of reconfigurable modules can be done mostly by following standard design flow and tools. The design can be done using normal HDL language

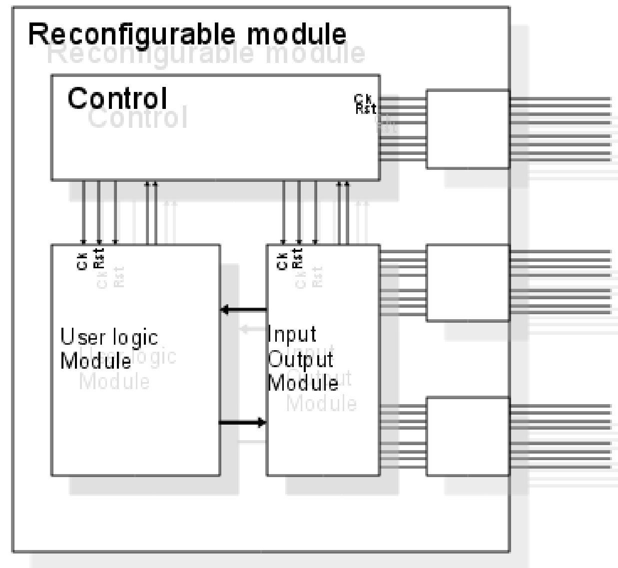

Figure 18: Reconfigurable module design

and synthesis, but some extra considerations needs to be made since the module will connect to an already existing system.

- Number of inputs/outputs to the module are set.
- Bus-macro type/version used.
- Bus-macro relative position.
- Size limitations of module.
- Ensure correct routing.

It is assumed that the design follows the standards used in the static design, following the same protocols for getting data in/out of the module and the control signals.

The target FPGA, for the design, needs to be the same FPAG used when implementing the module. The size of the FPGA does not mater, but it must be bigger than the reconfigurable area on the target device. For the Virtex-IV there are also limitations concerning designs crossing the center of the FPGA, as the configuration bits are mirrored on top vs. bottom.

Simulations on the design can be done with or without the bus-macros, also a final test using a test bench simulating the back-end should be used. This test bench simulates the back-end, that the module is connected to and will thus transfer the data into and out of the module the same way as the back-end. The test-bench will also simulate the control signals that the back-end is designed to use. This test bench needs to be implemented by the people designing the static design, so it is equal in functionality.

When place and route tools is run, placement constraints must be set for the module, this can be done by editing a User Constraint File (UCF). A UCF can easily be created in a tool like Xilinx planahead, where the FPGA layout is viewed from above and a constraint can be added, moved around and edited with ease in a visual manner.

Bus-macro type/version and relative position are absolute essential points. This is since the bus-macros have a defined routing so the same signal path is used in and out of the reconfigurable module. If another bus-macro type/version or the relative position of the bus-macros are used the routing will probably be different and the module will not be connected with the static design. This can result in short circuits in the FPGA, this can damage the FPGA.

When the design is placed, following the placement constraints and routed the routing needs to be checked, this is because the placement constraints does not constrain routing. The routing must be inspected so it does not cross the boundaries of the reconfigurable module. Also wires from the outside can not cross into the module, this is less of a problem when only the module is synthesized. The bus-macros should have the only wires crossing the boundaries, if other wire cross they need to be rerouted. The inspection and rerouting can be done with a tool like the fpga-editor from Xilinx.

When the design is finished routed and run through a bitfile generation tool, like bitgen from Xilinx, the reconfigurable module can be read out. This is done with a small programm developed for this framework, (modifiedCLBRead can be found in the zip file). Giving this program the module placement boundaries the module will be read out and written to a new file, this file can be used in the framework to insert the module in another design. Since just the module is extracted the file will be much smaller than the complete bitfile.

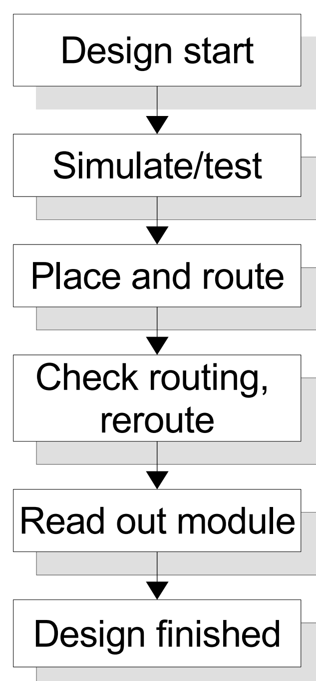

Figure 19: Flowchart for designing reconfigurable module

7.5. Reconfigurable module back-end design

The reconfigurable module back-end is the logic that connects the module with the

CPU. The CPU connection can be a bus as used in the proof of concept, or a direct connection.

The module back-end has different tasks that it must serve, for the reconfigurable system to work properly. The reconfigurable modules are connected to the system using bus-macros. Because of this all reconfigurable modules will have the same amount of input/output signals. Since different module designs will have a varying demand on data in/out of the module, the back-end need to implement a serial transfer of data words. This can not be handled directly from the CPU as long as the module and CPU are connected with a non deterministic connection.

A simple solution to this is to have a control module in the module back-end that can buffer data words from the module and CPU. This module will also control serial communication with modules. This control module would also control flushing and restoring the state of a module for replacement.

If the back-end had a local memory cash, it would be much easier for fast flushing and restoring of modules, given that they are restored to the same module. This would have to be controlled by the HWOS. If designed properly with regard to HWOS and module back-end, this should result in shorter reconfiguration times.

7.6.Method for communication with and between modules

Communication between modules are an important issue that are not easily solved for a self reconfigurable system. When the system controls removal and insertion of modules, the modules does not know if the modules they want to communicate with

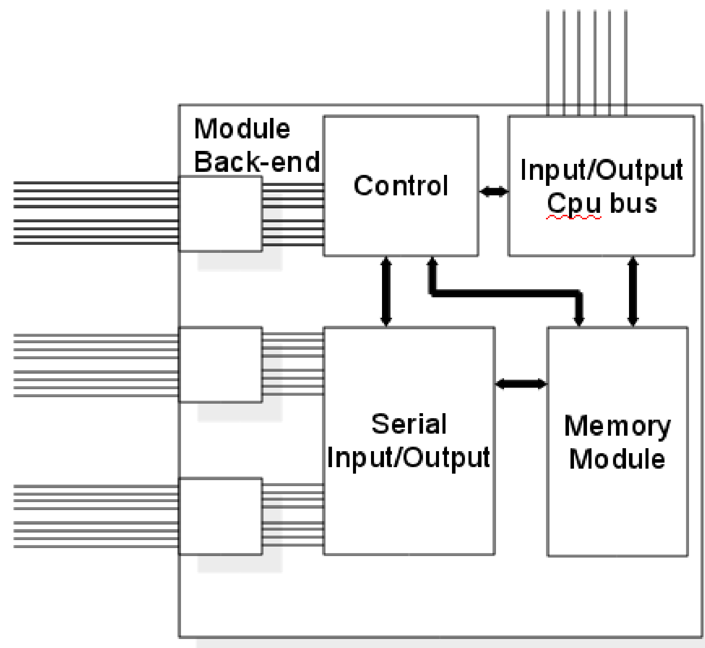

Figure 20: Reconfigurable module back-end

are available. So communication between modules will not be as simple using up a “dumb” network on chip. By dumb I mean the system is not dynamic, data gets sent to an address and addresses are related to a module position, not the module.

With the proof of concept the reconfigurable module are connected to the PLB bus and other modules connected to the bus, could be made able to communicate with

each other on that bus. For this to work, the HWOS configuring modules needs to tell the modules where the other module are, or if the module is available at all. This solution makes the communication less dumb, since addresses and availability will change dynamically, controlled by the CPU.

The easiest solution, will be to communicate to other modules through the HWOS, where the HWOS controls all communication. Some issues with this are an increase in load on the CPU and no determinism for when the message will be delivered. Since the communication goes through the HWOS, depending on the implementation, the message will be delivered to the module when the module gets reinserted, if the module is removed at the time of the communication.

For communication between modules in a reconfigurable system I propose that the system use communication through the HWOS as a starting point. There needs to be much more research in this field, to determine good solutions. As the proof of concept system developed only includes one reconfigurable module and there were not enough time to implement a version of the HWOS, an implementation of communication with modules were not developed. One of the issues with communication between modules are how to address modules that always keep changing. How modules address data sent to other modules not instantiated in hardware yet can be an issue when the HWOS gives addresses to modules.

Addressing of packages could be done with ipv6 addressing, where every module created would have a specific ip address. Or the addressing could be name based, where modules have names that are given to the HWOS when instantiated. This will depend on implementation and are not in the scope of this thesis.

7.7.Hardware OS

The hardware OS depict in the Work regarding hardware operating system section are mostly focused toward an OS in hardware. By OS in hardware one means that the OS is not running on a sequential designed processor, as normal software OS are.

The OS design concept here is a simple one, building on normal software OS designs. Also the design idea is meant as an extension for a normal software OS, so the OS is designed to run under a software OS, or be integrated with the software OS.

As a test implementation for the concept described here a user process program where coded for Linux. The use of Linux as a “base” OS simplifies many aspects. Especially since a proof of concept for dynamic self reconfiguration has been developed using Linux device drivers, for accessing the FPGA ICAP. Also, Linux already has the IP stack implemented and other communication protocols, this enables and simplifies initiation of hardware tasks from remote locations.

7.7.1.Design

The hardware OS design uses most of the same ideas as a software OS. As for a software OS, the OS are supposed to control tasks that wants to execute, do memory management, resource control and so on. The hardware OS, needs to control hardware tasks that are to be executed, control resources, schedule the hardware tasks and placement of tasks in hardware.

To design and create a prototype of a complete hardware OS is a task much to complex for this project. Instead a simple design is proposed, where the hardware

OS is a user space program running under Linux. Since the hardware OS is running under Linux many abstractions from Linux can be utilized.

The hardware OS consists mainly of four parts:

- The Scheduler
- The Placer
- The Launcher
- Communication

Most of the other concepts, that are needed for a HWOS are available from Linux. This includes locking, communication, software memory management and so on.

Following will be a short description of the four main parts.

7.7.2.The Launcher

The launcher handles new incoming tasks, checking that they are able to run on the system. The main task here is to check if the hardware module is ready to be placed on the FPGA. To be able to place the task there are some parameters that needs to be checked for the current platform. The hardware task needs to be created for the FPGA used, a hardware task synthesized for another platform can not be placed. The size of the module are important the module must be able to fit in the available reconfigurable area. The hardware task also needs to state what mode it wants to run in, continuously, one time, periodic and what deadline demands it might have.

Depending on how the task is delivered to the launcher, the mentioned data needs to be included.

The launcher is the outward interface for accessing the hardware OS. The launcher should ask the scheduler if the task can be executed, find out if there are available hardware slots and processing time for the module.

In Figure 22 the launcher is illustrated as a separate part of the HWOS with IP communication to the Scheduler and Placer processes. This is only for illustration, the Launcher could be a part of or separate depending on the implementation. But it is important to have IP communication at some stage to allow external processes to initiate hardware tasks, using IP communication.

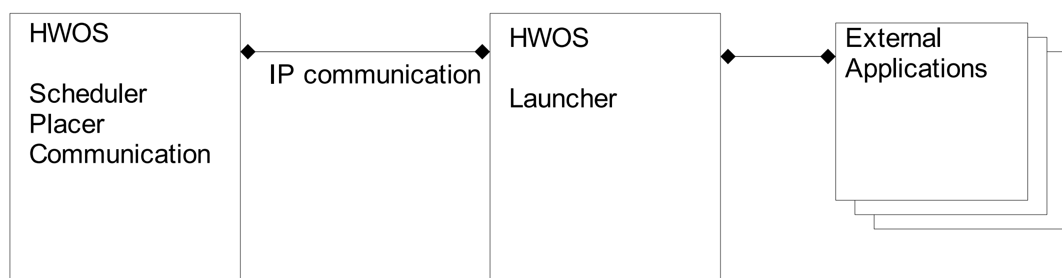

Figure 21: HWOS illustration

7.7.3.Communication

The communication part of the HWOS are to serve communication to and from the reconfigurable modules, when placed in hardware. As shown in Figure 22 there are IP communication to the launcher and conceptually the launcher does not need to be on the same platform as the reconfigurable system.

In the proof of concept the, communication to and from the reconfigurable module

are done with a driver in Linux. A driver will be the underlying means of communication. The communication has to take into account which reconfigurable module is used (if many are available). Also if the module are removed and placed again the communication access method to the module will change.

The communication part of the OS should abstract this away from the external program. This can be done by deciding on a specific API (Application Programmer Interface) in the implementation, letting the communication part of the OS control the actual addressing of data.

7.7.4.The Scheduler

The main task for the scheduler is to give tasks run time on the hardware. There are much work done in scheduling, for software OS and there will not be a big discussion here about different scheduling algorithms, as many can easily found in many text books.

The scheduler should accept tasks if it is possible to run the task, with the timing demands the task has. A task might want to run continuously, periodic, aperiodic and so on.

There are some points of interest for the hardware scheduler that differs from a software scheduler.

- Reconfiguration time.
- Preemption possibilities.

Reconfiguration time is important to incorporate in the scheduler model. If the scheduler switches tasks to often, the reconfiguration overhead will become significant. Having long periods between scheduling will result in a system where tasks have long delays between execution.

Depending on the system it might not be plausible to have hardware tasks swapped in and out periodically. This is because the task might be working on a constant stream of data, with real time demands.

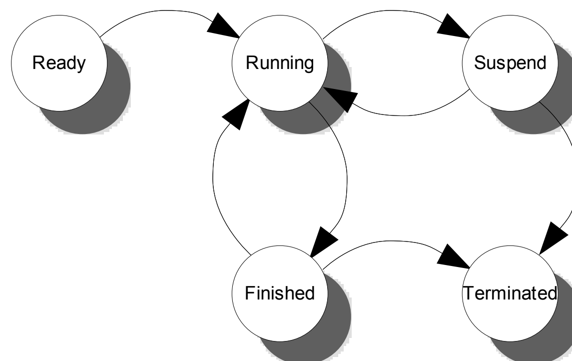

Figure 22: HWOS task states

Preemption is also an important aspect for the scheduler. If a task with higher priority wants to execute and there are no available hardware spots, a running task needs to be interrupted and removed. A issue with this is how to store the internal state of a preempted task. Different approaches can be taken to this problem, but the main point is that it will generate a overhead that needs to be taken into account.

7.7.5. The Placer

The placer is a part of the HWOS that are not a part of a software OS. As hardware modules will differ in demand on resources/area, the placer needs to control resources available and place new tasks as best possible. Depending on the reconfigurable module design solution, the placers complexity will vary greatly. If a 1 dimensional placement is used the complexity will be much less compared to a 2 dimensional placer. Also the smallest reconfigurable unit size will have a big impact on the placer algorithm complexity.

When modules can be reconfigured, swapped in and out, the system will suffer from fragmentation. For 1D approach the research done on fragmentation in memory can be used, since this is a very similar case.

7.7.6. Prototype

A framework/prototype for a HWOS was implemented and are included in the zip file following this thesis. The framework is only meant as a test for some concepts in Linux that can simplify the HWOS.

Since there were many complications in researching and implementing the prof of concept for self reconfiguration on a FPGA, there were not much time for implementation and testing of the HWOS. Also because the prof of concept only were able to have one reconfigurable module and only two reconfigurable modules were created, the HWOS was only tested on the development computer.

The HWOS implementation given are only meant as a starting point for future work showing some key concepts.

Some of the Linux based programming concepts that might ease further development are threaded functions with communication through message queues. These can be used for communication between processes, if some parts of the code are separated out, like the launcher function.

For the system a simple linked list were developed, this were used for task queues used by the scheduler.

The framework also includes code example for launching an external process, making it possible to incorporate the test program in the reconfiguration proof of concept for writing a configuration to the ICAP driver.

The code for the HWOS are by no means finished or functional, but can serve as a starting point, incorporating some coding techniques that might reduce the time to get a system working. For experienced coders I would recommend starting from scratch with a better design, trying to better incorporate the functionality with the Linux kernel.

7.8. Analysis of a few applications

The reconfigurable system discussed here, has some limitations for designs that are to be integrated/used in the system. Most obvious of these limitations are the interface to the reconfigurable modules.

The interface to a reconfigurable module consists of signals in and out of the module, connecting the module to the static design. When the LUT based bus-macros are used, the signals are one directional and it is not possible for an application to adjust the amount of input-signals vs. output-signals. For all modules

the amount of input-signals and output-signals will be fixed.

A further limiting factor are the clock signal that the module wants to use. Depending on implementation of the reconfigurable system, there might be only one clock signal that will have a given period. With this limitation, modules that are designed to run at a lower or higher clock can fail. This is an important point and designers of reconfigurable modules needs to design robust system, following specifications given for the reconfigurable system.

Another big issue with reconfigurable modules are that they can be swapped out. This is not a big issue for modules that complete their task in one clock period, like the simple example modules used in the proof of concept. For a module taking several hundred of clock cycles, to finish a task, the module should be able to store its internal state. So when the overlaying system wants to switch it out it can be done. Dumping the data from a reconfigurable module could become a security risk, if a en/de cryption module are told to dump its half en/decrypted data, getting the en/decryption keys could become trivial. This is definitely an issue that will need more study and analysis.

A enhancement for a reconfigurable system, is that modules could be further divided. Manny applications has some parts for debugging or parts of the design seldom used. These parts could further be separated in other modules, creating a more compact system that can become more reliable, enhancing the reconfigurable platform concept.

7.9. Suggestion of method for swapping modules while running

As the system are able to reconfigure it self, while it runs, the need to replace hardware tasks while they run will be essential. There can be tasks with higher priority, that only needs to run some times or load balancing of available tasks. To be able to execute these or other interchanging tasks without having to restart the tasks each time, the system needs to be able to store the state of the module before it is reconfigured.

The reconfigurable module that are to support being swapped in and out during execution, can not be of asynchronous design. A asynchronous designed module will have to be restarted if it is swapped.

The design ideas here, are for reconfigurable modules that utilizes a control flow, running through determinable states. When the system has determinable states, it is possible to store the state and the data sett currently being worked on. This makes it possible to restore the state and data sett later.

A solution is to just read out the configuration bits, storing the complete module in the current state, like a snapshot. To do this correctly the module needs to be stopped so that the module are in a stable state. The overhead from reading out the module will be deterministic and depend on the size of the module. This is given by clock speed and bit width of the configuration port. A minus with this approach, is that the data read out will mainly be composed of interconnect data and not logic or state data.

A CLB in Virtex-4 is composed of 22 frames where 20 of these are interconnect. 2 frames are for LUT logic. This results in a massive overhead of unnecessary data as the interconnects will not change.

It is possible to read out just the LUT logic but this will result in extra processing

time, when reconfiguring the module, as the logic needs to be reinserted with the original module data.

I propose a different solution, as the actual data and data sett being worked on will always be less then the logic frames in a CLB (some of the logic will be just logic). To minimize the data having to be moved around, the module will dump the state and data when a signal from the control unit are sent. For this approach to work the modules needs to be designed for it.

The data being dumped from the reconfigurable module could be dumped to the CPU and stored in CPU memory if needed, but this might take to long. I propose a memory system for storing data from the reconfigurable module that are swapped out. When the HWOS initiates a hardware task, the task must state how much memory it needs for storing data when stopped. This will also give the reconfiguration time, time to dump date. The HWOS can then schedule tasks depending on speed of reconfiguration. The HWOS will allocate memory, at the reconfigurable module back-end, for the module. The memory allocated will be available depending on the state of the task. Task state example are shown in 36. When the task is terminated the HWOS will reclaim the memory.

With this approach, the time to store the state of a module depends on the amount of data to store and speed to transfer the data. The speed to transfer data depends on clock speed and bit width, (bits pr clock period). As the ICAP normally uses 8 bit and the system clock, a module using 32 bit or more reading out less data will be more efficient.

7.10. Proof of concept run-time reconfiguration

The basis for this project and most of the developed concepts depends on the point that self reconfiguration is possible, for the given device/platform.

As discussed earlier, the difficulties with self reconfiguration on a FPGA are many. The complexity of the FPGA device, difficulties with synthesize tools and how to utilize the tools in new and undocumented ways are difficult. Working with Linux, understanding how a complex OS like Linux works, utilizing its design and functionality to simplify the design. These are time consuming tasks not directly leading to any results.

Because of these difficulties, a lot of time in this project has been used to get a proof of concept, utilizing self reconfiguration, to work. The proof of concept incorporates a lot of work done by other researchers, as explained in Work regarding dynamic reconfiguration on page 9. The concept shows that it can be done on the Suzaku platform used here.

The proof of concept incorporates a hardware design in FPGA, consisting of a static design part and a reconfigurable module. The reconfigurable module are only 3x16 CLBs in size and can thus not really fit any modules with more extended functionality. The reconfigurable modules designed for the test are simple addition and subtraction modules. The reconfigurable modules are connected to the static design with bus-macros, these bus-macros are placed with the same placement constraints for both modules. Since the bus-macros are placed in the same location the connecting wires used are the same.

The static design used are the standard Suzaku FPGA design with the HWICAP module added to allow self reconfiguration. The static design has placement

constraints added, so it will not be placed in the reconfigurable module area. To ensure that the routing does not cross the reconfigurable module border the design was viewed in the Xilinx fpga editor software and the design was rerouted where necessary.

Two complete designs were created, one with the subtraction module and one with the addition module. The modules with system could thus be tested to ensure correct operation before any modification.

The functionality of the modules were tested with a test program running under Linux that used a driver developed for accessing the hardware modules. The driver and test program are included in the zip-file, (/code/linux_drv/adder/ and /code/adder/).

When the module functionality was determined correct, by running some simple tests checking that the result were as expected, the modules were read out of the bitfile. By using a program with algorithms for accessing and locating the different logic in the bitfile (modifiedCLBRead), derived from the theory and data of bitfile design, the modules were read out. The program for reading out modules is included in the zip-file, but the program has some bugs and should not be used blindly. The program access algorithms work correctly for the top right corner of the FPGA bitfile, not including routing connected to the BRAM and DSP.

With the modules read out another program was developed to use the ICAP driver in Linux to read and write configurations. As the module read out from the bitfile is just the configuration, there are some more data that needs to be written to the FPGA configuration logic for correct configuration. The test program for reading and writing configurations to the FPGA using the ICAP driver are included in the zip-file, (/code/icap_test/ and /code/icap_write/).

To ensure correctness of the program, that reads out modules and the ICAP driver, the module read out from the FPGA was compared with the module read out from the bitfile. This ensures that the addressing are equal for both programs, and that there has not been any changes to the configuration when configuring the FPGA.

After ensuring that the different parts were working correctly the final test was to write the modules to the reconfigurable area changing the functionality. The modules were tested with the same test program using the same driver accessing the hardware.

The prof of concept was running on the Suzaku-V platform on the sz410 board using the Virtex-4 FPGA. The FPGA design, placements constraints and bitstream generation was done as explained in Suzaku-V setup on page 21. The test procedures and commands that were run are shown in the results chapter.

With the read out of configuration from the FPGA working as intended, the possibility to read out “stopped” modules are available. This makes it possible to read out a module when it is stopped and thus saving its state. This will only be possible for Virtex-4 as it is not possible to read out memory elements from Virtex-II.

7.11. Code

The program code explained in this section will be included in the zip file. In the zip file the program code will be in folders similar to most of the headings in this section, where the code functionality is explained.

7.11.1.Hardware Modules connected to the OPB bus

It is relatively simple to create a module connected to the OPB bus, since Xilinx EDK creates a template with all necessary logic for you. It is somewhat more difficult when the bus-macros are used and the logic are placed in another file. Following will be an explanation of the file structure of the module created from Xilinx EDK and what parts need modifications, so that the module will work when included in a design.

When a module is created by EDK there will be a folder named the same as the module, under the pcores folder where the design is stored. Under this folder there are three folders named: *data, devl and hdl*.

The *hdl* folder contains the logic vhdl or verilog files.

The *devl* folder contains some creation log files, a *README.txt* file explaining some of the files involved and synthesis script files.

The *data* folder contains the *.pao* file (Peripheral Analysis Order), if some logic are placed in another file, the file needs to be included here so it gets synthesized.

When bus-macros are included in the design, the files needs to be available when synthesized. I have not been able to find the script, or file, to include the bus-macro files. So when bus-macros are used these files needs to moved to the implementation folder when synthesizing. In the Suzaku project this has to be done after synthesis is started, since the implementation folder is not created before. Also each time the clean commando is used the folder is deleted, so the files needs to be moved again. There are most likely a file where the bus-macro can be included automating this procedure, this is just an issue that has not been given any time to solve.

The implementation can be done as any other hdl design. Xilinx expects all user logic to go into the *user_logic.vhd* file. I found it easier to visualize a border between reconfigurable module and static design by having them in separate files. The instantiation and connection for the bus-macros, user_logic and reconfigurable module design, where done in the top design file created by the wizard and named as the module name given in the wizard. For an example of a top file in the module see the file:

hdl/cores/MyProcessorIPLib/pcores/opbbusmacrointerface_v1_00_a/hdl/vhdl in the included zip file.

7.11.2.ModifiedCLBRead

The ModifiedCLBRead program reads out and writes in frames from/to a bitfile. The ModifiedCLBRead program, builds on the bitstream theory with algorithms for locating the different logic in a bitstream and reading this out.

The VirtexIV version is the one mostly worked on. But this has some known bugs in the algorithm for locating correct position in the bitstream and also probably some unknown bugs. The code should not be used as is, but after understanding the code it can be used, this is to reduce the probability of making some mistake. When manipulating the bitstream a mistake can definitely render the FPGA defect. Wrong manipulations can create internal shorts in the design which can lead to a defective FPGA.

The program is created to take a complete bitstream as input and read out a frame range and write these to a new file. This new file can then be used for partial

reconfiguration, by the icap_write program or modifiedCLBRead can read the frames from this file and insert them in another complete bitstream.

If a module read out are to be inserted in another complete bitstream, the complete bitstream needs to have CRC turned of since modifiedCLBRead does not calculate a new CRC value.

The VirtexIV version also has a print CLB snippet when issuing the modifiedCLBRead with -sc [**nr**] command the CLB number **nr** will be printed. This can be handy when comparing the wiring in FPGAeditor with the CLB frames, making it easier to understand how the CLB data are coded in the bitstream.

7.11.3.icap_write and icap_test

The icap_write and icap_test programs are written to utilize the Linux icap driver. These are just test programs, especially since they have hard coded the addressing and has a lot of printf() for debugging.

Important points to note in these test programs are the configuration setup used, with configuration details of the FPGA explained previously. The configuration used in these test programs are tested and works, so they are a good starting point for further optimization and inclusion in a HWOS.

When writing to the FPGA, addressing can be a bit tricky. This is done with a packet to the FAR (Frame Address Register) in the FPGA control logic. In icap_write this is set in the write_header(..) function.

```
// Top/bottom bit, block type, row address, column, frame
address = (0<<22) | (0<<19) | (row<<14) | (column<<6) | frame;
// Setup FAR
data[10] = FAR_PACKET;
// address (top, CLB, row, column, minor)
data[11] = address;
```

In this code snippet from write_header(..) in icap_write.c the addressing are set for the FPGA. For the Virtex-IV FPGA the FAR register is composed as follows:

Reserved, bit 23 to 31.

Top/Bottom bit, bit 22.

Block Type, bit 19 to 21.

Row address, bit 14 to 18.

Column address, bit 6 to 13.

Minor address, bit 0 to 5.

Definitions:

Top/Bottom bit: 0=Top, 1=Bottom.

Block Type: CLB/IO/CLK = 000.

BRAM interconnect = 001.

BRAM Context = 010.

(CFG_CLB 011, CFG_BRAM 100)

Row Address: Selects a row of frames, 16 CLB's in height. Increases away from the middle in both directions, Depending on Top/Bottom bit.

Column Address: Selects a major column CLB, IOB, CLK, DSP, … Starts at 0 on the left and increases to the right.

Minor Address: Selects a memory cell address line within a major column, (frame in a CLB).

Data from [45].

7.11.4.Genbitfile

The genbitfile program was developed for merging a module in to other frames especially meant for the VirtexII since one configuration frame here spans the complete height of the FPGA.

Since the ICAP driver for the VirtexII FPGA did not work, the work on this program is incomplete. In the VirtexIV the frame length is always 16 CLB's so the need for merging frames are not so apparent.

Results

Following will be the execution of codes done to test the proof of concept. First the modifiedCLBRead is built and run on a bitfile where a reconfigurable module is placed in the top right corner, (shown in previous chapter).

(please overlook spelling in code).

```
[unknown@archbox virtexIV]$ uname -a
Linux archbox 2.6.29-ARCH #1 SMP PREEMPT Wed May 20 06:42:43 UTC
2009 x86_64 AMD Athlon(tm) 64 FX-62 Dual Core Processor
AuthenticAMD GNU/Linux
[unknown@archbox virtexIV]$ pwd
/home/unknown/project/school_master/code/modifiedCLBRead/virtexIV
[unknown@archbox virtexIV]$ make
gcc -Wall -c -o main.o main.c
gcc -Wall -c -o CLBRead.o CLBRead.c
gcc -Wall -o testing/test main.o CLBRead.o
[unknown@archbox virtexIV]$ cd testing/
[unknown@archbox testing]$ ls
adder_module.bit sub-modified-to-adder.bit sz410-adder-32in-32out-
bm.bit sz410-sub-32in-32out-bm.bit
orig_module.bit sub_module.bit sz410-hwicap-adder.bit         test
[unknown@archbox testing]$ ./test -version -author
Version: 0.2
Author:  Sverre Hamre
[unknown@archbox testing]$ ./test -h
Usage: [comand]...

-i [filename]       Input file name
-o [filename]       Output file name
-verbose            Explain what is beeing done
-fmR                Frames mode (reads out frames)
-fmW                Frames mode (writes out frames)
-sc [nr]            Start CLB
-ec [nr]            End CLB
-version            Prints version
-author             Prints author of code
[unknown@archbox testing]$ ./test -i sz410-adder-32in-32out-bm.bit
-o temp_adder_module.bit -fmR -sc 21 -ec 23 -verbose
Input file name:        sz410-adder-32in-32out-bm.bit
Output file name:       temp_adder_module.bit
Frame mode, reading out frames
read_frame: acessing
read_frame: Memory alocated
locateFrameStart: Synch word = aa995566
locateFrameStart: FDRI word = 30004000
locateFrameStart: word = 50024090
locateFrameStart: Type2 header detected
locateFrameStart: Word count: 147600
locateFrameStart: Number of configuration bits: 4723200
locateFrameStart: Configuration bytes = 4803
locateFrameFromCLBnr: frame: 644
locateFrameFromCLBnr: CLBcol: 21
locateFrameFromCLBnr: frame: 1190
read_frame: frames to read out: 66
read_frame: frameStart: 199963
read_frame: Read out CLBs startCLB: 21 endCLB: 23
read_frame: startCLB >= 16 startCLB < 24
warning BRAM/DSP connections read out not implemented
read_frame: Closing files
[unknown@archbox testing]$ ./test -i sz410-sub-32in-32out-bm.bit -o
temp_sub_module.bit -fmR -sc 21 -ec 23 -verbose
Input file name:        sz410-sub-32in-32out-bm.bit
```

```
Output file name:      temp_sub_module.bit
Frame mode, reading out frames
read_frame: acessing
read_frame: Memory alocated
locateFrameStart: Synch word = aa995566
locateFrameStart: FDRI word = 30004000
locateFrameStart: word = 50024090
locateFrameStart: Type2 header detected
locateFrameStart: Word count: 147600
locateFrameStart: Number of configuration bits: 4723200
locateFrameStart: Configuration bytes = 4803
locateFrameFromCLBnr: frame: 644
locateFrameFromCLBnr: CLBcol: 21
locateFrameFromCLBnr: frame: 1190
read_frame: frames to read out: 66
read_frame: frameStart: 199963
read_frame: Read out CLBs startCLB: 21 endCLB: 23
read_frame: startCLB >= 16 startCLB < 24
warning BRAM/DSP connections read out not implemented
read_frame: Closing files
[unknown@archbox testing]$ ls
adder_module.bit sub-modified-to-adder.bit sz410-adder-32in-32out-
bm.bit sz410-sub-32in-32out-bm.bit temp_sub_module.bit
orig_module.bit sub_module.bit sz410-hwicap-adder.bit
temp_adder_module.bit test
```

First step finished, the addition module is read out of its bitfile and the subtracter module read out of its bitfile. These files are copied to the nfs_share folder. The rest of the work is done on the Suzaku-V. The Suzaku-V is loaded initially with the design including the subtraction module.

```
# uname -a
Linux SUZAKU-V.SZ410 2.6.18-at11 #6 Tue Apr 21 16:46:57 JST 2009
ppc unknown
# mount
/dev/mtdblock7 on / type romfs (ro)
/proc on /proc type proc (rw)
none on /var type ramfs (rw)
none on /etc/config type ramfs (rw)
192.168.0.1:/home/unknown/project/nfs_share on /mnt type nfs
(rw,vers=3,rsize=32768,wsize=32768,hard,nolock,proto=udp,timeo=7)
# cd mnt
# ls
adder               icap_read           temp.hex
adder.ko            icap_write          xilinx_hwicap_m.ko
adder_module.bit    orig_module.bit
frames.bit          sub_module.bit
# insmod adder.ko
# ./adder
Adds two numbers.
Type inn numbers A: 33
A is: 33
Type inn numbers B: 55
B is: 55

input =  33
Access Hardware
FPGA Memory accessing
write_fpga: writing data 33005500 to address c5002200
ret: 4
fpga_read: reading data 33005500 fom address c5002200
fpga_read: reading data 2200 fom address c5002204
fpga_read: reading data 0 fom address c5002208
ret: 12
```

```
output =  33 0 55 0
output =  0 0 22 0
output =  0 0 0 0
# insmod xilinx_hwicap_m.ko
icap icap.0: Xilinx icap port driver
icap icap.0: ioremap f0f00000 to c5008000 with size 1000
# ./icap_write -h
icap_write -i [filename] -f [frames]
This program has the address hardcoded inn.
Frames will be written to CLB 21 and on, on top row.
# ./icap_write -i adder_module.bit -f 66
Frames: 66
Access Hardware
icap icap.0: initializing
icap icap.0: Reset...
icap icap.0: Desync...
icap icap.0: Reading IDCODE...
icap icap.0: IDCODE = 21e58093
icap icap.0: Desync...
FPGA Memory accessing
Writing frames
Finished writing frames
Postconfig
Getting Status of FPGA
Stat acces word 2800E001
Status: 78FC
closing device
# ./adder
Adds two numbers.
Type inn numbers A: 55
A is: 55
Type inn numbers B: 33
B is: 33

input =  55
Access Hardware
FPGA Memory accessing
write_fpga: writing data 55003300 to address c5002200
ret: 4
fpga_read: reading data 55003300 fom address c5002200
fpga_read: reading data 8800 fom address c5002204
fpga_read: reading data 0 fom address c5002208
ret: 12

output =  55 0 33 0
output =  0 0 88 0
output =  0 0 0 0
# date; ./icap_write -i sub_module.bit -f 66; date
Thu Jan  1 00:17:27 UTC 1970
Frames: 66
Access Hardware
icap icap.0: initializing
icap icap.0: Reset...
icap icap.0: Desync...
icap icap.0: Reading IDCODE...
icap icap.0: IDCODE = 21e58093
icap icap.0: Desync...
FPGA Memory accessing
Writing frames
Finished writing frames
Postconfig
Getting Status of FPGA
Stat acces word 2800E001
Status: 78FC
closing device
Thu Jan  1 00:17:27 UTC 1970
```

```
# ./adder
Adds two numbers.
Type inn numbers A: 38
A is: 38
Type inn numbers B: 49
B is: 49

input =  38
Access Hardware
FPGA Memory accessing
write_fpga: writing data 38004900 to address c5002200
ret: 4
fpga_read: reading data 38004900 fom address c5002200
fpga_read: reading data 1100 fom address c5002204
fpga_read: reading data 0 fom address c5002208
ret: 12

output =  38 0 49 0
output =  0 0 11 0
output =  0 0 0 0
```

In the code here one can see that the output from the adder program, which accesses the hardware changes functionality. When the adder module is inserted the returned data in the test program shows an addition (the code has bad formating and the numbers are in hex for debugging). When the subtraction module is inserted, the output when accessing the same hardware module changes from addition to subtraction.

In the last example where the sub_module is inserted the command is run with a simple date before and after (; separates commands). This gives only a vague indication on configuration time. Other commands for measuring execution time was not available. As seen the time is less then a second for reconfiguration, this includes the printf() commands, so the configuration program is not optimal.

The example shows reconfiguration of modules done in under a second, without rebooting Linux. This is a major improvement over previous reconfiguration time of over 70 seconds (including reboot of Linux) [6].

The main time differences in the two examples are that the 70 second is mostly because of rebooting Linux, but also because of a full configuration of the FPGA. The example here only reconfigures a part of the FPGA and Linux does not have to reboot. Since Linux does not have to reboot other users could be connected to the platform and would not notice any difference, unless they were using that hardware module.

Discussion.

A self reconfigurable system is complex, when using current technology to realize it. Understanding visualizing, conceptualizing and implementing are time consuming. With only 20 weeks there are limits to what can be done, especially when one has to understand the FPGA device used, its development tools, the Linux kernel and distro with tools.

The problem is not that the different issues or tasks are hard to do or difficult to understand. It's the amount of information to go through to find the correct data and also the lack of documentation.

Problems aside dynamic self reconfiguration is an interesting field that can enhance many systems and tasks. Some enhancements are the ability to remove bugs in hardware, after shipment, ability to comply to not yet defined protocols, reduced hardware need. Since not all tasks are needed to be in the final configuration, leading to power saving advantage and less complexity.

The ability to remove hardware bugs after shipment can be seen as a good and bad thing. As a positive side a device with faults can be updated fast, without having to change the device. The developer can easily change a defect in the design without it taking 3+ months to run a new batch of the device, resulting in faster time to market. As a negative side, products can be rushed even more then what is done currently, further building on the beta society that rules now. With this I mean products will be rushed to the customer even earlier as it is possible to fix problems, further implement functionality in the system after the customer gets the system. This can result in devices not functioning as anticipated before some time after arrival, further enhancing peoples distrust and frustration on devices not functioning as good as they should.

The ability to adjust to new protocols is really important for users, this can make devices work seamlessly when entering a new environment. An example can be when traveling to different countries and the cellphone will automatically update the protocols and system for that country, without having extensive hardware capable of this.

As a result of this project a proof of concept has been created demonstrating self reconfiguration on the Suzaku-V platform utilizing a Virtex-IV FPGA. The prototype has a small reconfigurable area, 16x3 CLB's, so the modules implemented for test has simple functionalities, addition and subtraction. As the system can reconfigure it self and the modules can be used with communication from the CPU it is shown a method for self reconfiguration on a Xilinx FPGA.

The proof of concept does not have multiple reconfigurable modules, so it is not tested how reconfigurable modules can be moved/replaced. With the structure of the bitfile explained in the theory section, one can see the problems related to moving modules in the bitstream. Just moving CLB data is not a problem as these are uniform, but when a bigger module consisting of multiple CLB's and spanning over other logic, like BRAM or DSP the problem changes. If BRAM or DSP logic are used the module can only be moved to places where these elements are available. When a module uses routing in a DSP or BRAM slice but not the logic, the module can be moved to a location without this logic, since the 20 first frames for a CLB, DSP or BRAM are the interconnects. The interconnects are equal for all logic. Even if it is possible, this will result in more processing as the bitstream needs to be

modified when uploaded, not just preprocessed at creation.

Dynamic self reconfiguration is of most interest when it is possible to reconfigure a part of a design that connects to a design that does not change, static design. This gives the most flexibility and enhancement to a system. In this thesis it is shown how the bus-macros can give this flexibility and also the possibility to design modules without knowing or caring about the static design.

The bus-macro design used here is the LUT based design as earlier explained. This design gives a great deal of flexibility as these can be created in all sizes and directions . An issue with the LUT based bus-macro is that it uses LUT's that could be used for logic, giving a logic usage overhead to the reconfigurable module size. There is only one reason for the bus-macros need to use LUT's and that is that LUT's are the smallest unit that can be set as a hard macro by Xilinx tools. It could be as simple as Xilinx patching their tools to enable hard macros defining wires or switch boxes. By just defining wires crossing a border, a reconfigurable module would have no logic overhead.

With reference to Xilinx tools it is possible to set area constraints limiting logic inside a “box”. Also using Xilinx own partial reconfiguration tool flow, it is possible to limit routing for a reconfigurable module inside the same “box”. A limitation with Xilinx tools is that the static design routing can not be limited to not cross into a reconfigurable module, so the static routing cross into the reconfigurable module. Xilinx partial reconfiguration tools thus has to compile all reconfigurable modules at the same time, since routing used in the reconfigurable module for the static design has to be included in all modules.

In a reconfigurable system a network solution for communication to and from the modules are absolute necessary. Since the modules are not always present in the configuration, this network becomes more complex. Modules trying to communicate with each other, one module can not know if the other module is present. A solution to this is to route all communication through the configuration controller (CPU), the messages can then be routed to the correct address or put on hold if the module is not present.

Controlling a self reconfigurable system is a complex task. The HWOS ideas and framework work done in this thesis are only a scratch on the surface on this important subject. The lack of a functioning system, for testing of different solutions is mostly the reason for this, more time was given to getting actual self reconfiguration working, on a Virtex-IV FPGA. With a functioning system for self reconfiguration more work can be done concerning the HWOS aspects. Important focus points here are scheduling of tasks, placement and fragmentation of modules, communication between modules and to external applications. Integration of the HWOS in Linux can simplify much work, by using available resources and implementations done in Linux, the IP-stack, network protocols, scripting and so on.

Swapping of modules are essential for the self reconfigurable system. The swapping of a module will take some time, resulting in a overhead for execution of tasks in hardware. It is not possible to start a task before it has been fully configured. This is in difference to software where parts of the program can be loaded from memory and executed. Smaller module sizes could be achieved by dividing a module and thus reducing some of the overhead. But this will most likely not be possible in most designs. Storing the work done in a module when switching it out, can be time consuming. This can be done by stopping the module, then read the complete module out and writing it in again when execution can be continued. As proposed

earlier, designing a module so it can flush data when signaled, the time it takes to store the state of a module can be reduced greatly. The actual processed data in a module will be much less then logic and routing data, if the complete module configuration is read out.

Conclusion

Difficulties:

As described throughout this thesis, self reconfiguration is a complex task. The system is complex and to prove correctness the complete system needs to be tested. It is not possible to correctly prove functionality by separately testing concepts.

Bus-Macro:

To be able to realize a partially reconfigurable system, where modules and static design can communicate with each other, bus-macros were used. Bus macros are prerouted modules included in a design, ensuring that the same “wires” are used. The bus-macro are a vital part of the partial reconfigurable system. Bus-macros uses the smallest unit that can be defined in a macro. Further enhancement to Xilinx tools could enable wiring to be the smallest definable unit, this would be really useful.

Proof of concept:

To test and show the bitstream data composition of the FPGA, a partial self reconfiguration proof of concept was created for the Suzaku-V platform. Because of the limited size on the FPGA, only one reconfigurable module was included. The concept was proven to work with two simple modules, an addition and a subtraction module. The proof of concept is a excellent starting point for further enhancements, as it is a working system, but it is only a proof of concept and should not be used as is. The proof of concept realized reconfiguration without rebooting Linux, a module was reconfigured and ready for use in less then a second.

Many aspects needs more work:

The main focus of the work done in this thesis, has been to get a system working and how to create the system with available tools. There are many aspects of this that needs more work. Creating reconfigurable modules now demands a lot from the designer, checking and rerouting a FPGA design is tiresome. The concept of reading out and writing in modules has been shown, the programs doing this needs further work to be more dynamic and bug free. The HWOS needs much thought and design to be made more generic and optimal. Manny ideas and concepts has been shown or described in this thesis, most of these are in early stages and needs more work.

I feel the work done in this thesis will be of great value for further work. Some concepts has been proven and they are shown how to implement. There has also been proposed some ideas and concepts for further work. Hopefully this will simplify the process of further development and research regarding self reconfigurable systems.

Future work

There are multiple areas that needs more work in the area of dynamic self reconfiguration. Following are some points that can be of interest as a continuation of the work done in this thesis.

- Upgrade platform to use a FPGA with more logic enabling multiple bigger reconfigurable module areas.
- Work with placement, the possibility of hardware modules accelerating the bitstream modification. Making this module a reconfigurable module.
- Creating models in SystemC for testing placement and scheduling algorithms for the system. This can reduce the time consuming task of getting the system to work, and also a testbed for new concepts.
- Other platforms? Work in this thesis are for the Xilinx FPGA, make a survey of possible systems.
- Thorough study on Xilinx tools, what is possible with regards to limit routing and placement of logic.
- Reverse engineer the Xilinx bitstream for the ability to modify logic, read out data and check routing conflict cross reconfigurable module borders.

References

[1] "Ambiesense."

[2] S. Hamre, *Self Reconfigurable System on a Xilinx Spartan3 FPGA by using Bus Macros.* , NTNU, 2008.

[3] J.M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*.

[4] A. Jantsch and H. Tenhunen, "Will Networks on Chip Close the Productivity Gap?," *Networks on Chip*, 2003, pp. 3-18.

[5] I. Hauge, "Analyse, dekomponering og rekonstruksjon av fpga-konfigurasjon for ahead," NTNU, 2006.

[6] S.R. Arntsen, "FPGA-plattform for AHEAD."

[7] F. Gravdal, "Selvrekonfigurering av FPGA," 2007.

[8] I. Hauge, *Arkitekturbeskrivelse for AHEAD*, 2005.

[9] A.E. Vestnes and T. Øvrebekk, *Selvrekonfigurering*, 2007.

[10] M. Heubner and T. Becker, "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration," *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, 2004, pp. 28-32.

[11] J. Becker, "Embedded Systems Group," *Embedded Systems*.

[12] M. Huebner, M. Ullmann, L. Braun, A. Klausmann, and J. Becker, "Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems," *Field Programmable Logic and Application*, 2004, pp. 1037-1041.

[13] K. Paulsson, M. Hubner, and J. Becker, "Cost-and Power Optimized FPGA based System Integration: Methodologies and Integration of a Low-Power Capacity-based Measurement Application on Xilinx FPGAs," *Design, Automation and Test in Europe, 2008. DATE '08*, 2008, pp. 50-55.

[14] K. Paulsson, M. Hübner, and J. Becker, "Dynamic power optimization by exploiting self-reconfiguration in Xilinx Spartan 3-based systems," *Microprocessors and Microsystems*, vol. 33, Feb. 2009, pp. 46-52.

[15] K. Paulsson, M. Hubner, G. Auer, M. Dreschmann, L. Chen, and J. Becker, "Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 351-356.

[16] S. Bayar and A. Yurdakul, "Self-reconfiguration on Spartan-III FPGAs with compressed partial bitstreams via a parallel configuration access port (cPCAP) core," *Research in Microelectronics and Electronics, 2008. PRIME 2008. Ph.D.*, 2008, pp. 137-140.

[17] K. Paulsson, U. Viereck, M. Hübner, and J. Becker, "Exploitation of the External JTAG Interface for Internally Controlled Configuration Readback and Self-Reconfiguration of Spartan 3 FPGAs," *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI - Volume 00*, IEEE Computer Society, 2008, pp. 304-309.

[18] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities," *Field Programmable Logic and Application*, 2004, pp. 454-463.

[19] M. Hiibner, C. Schuck, M. Kiihnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 2006, p. 6 pp.

[20] J. Becker, A. Donlin, and M. Huebner, "New tool support and architectures in adaptive reconfigurable computing," *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, 2007, pp. 134-139.

[21] L. Singhal and E. Bozorgzadeh, “Multi-layer Floorplanning on a Sequence of Reconfigurable Designs,” *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1-8.

[22] S. Donthi and R.L. Haggard, “A survey of dynamically reconfigurable FPGA devices,” *System Theory, 2003. Proceedings of the 35th Southeastern Symposium on*, 2003, pp. 422-426.

[23] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, “REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems,” *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005, p. 151b.

[24] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, “Modular dynamic reconfiguration in Virtex FPGAs,” *Computers and Digital Techniques, IEE Proceedings -*, vol. 153, 2006, pp. 157-164.

[25] F. Ferrandi, M. Morandi, M. Novati, M. Santambrogio, and D. Sciuto, “Dynamic Reconfiguration: Core Relocation via Partial Bitstreams Filtering with Minimal Overhead,” *System-on-Chip, 2006. International Symposium on*, 2006, pp. 1-4.

[26] M. Koester, M. Porrmann, and H. Kalte, “Task placement for heterogeneous reconfigurable architectures,” *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. 43-50.

[27] Y. Krasteva, E. de la Torre, T. Riesgo, and D. Joly, “Virtex II FPGA Bitstream Manipulation: Application to Reconfiguration Control Systems,” *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1-4.

[28] T. Becker, W. Luk, and P. Cheung, “Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration,” *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 2007, pp. 35-44.

[29] J. Carver, N. Pittman, and A. Forin, *Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams.*

[30] M. Hubner, L. Braun, J. Becker, C. Claus, and W. Stechele, “Physical Configuration On-Line Visualization of Xilinx Virtex-II FPGAs,” *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, 2007, pp. 41-46.

[31] J. Becker and A. Thomas, “Scalable processor instruction set extension,” *Design & Test of Computers, IEEE*, vol. 22, 2005, pp. 136-148.

[32] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. Melcher, “Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC,” *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, 2007, pp. 35-40.

[33] F. Dittmann, M. Gotz, and A. Rettberg, “Model and Methodology for the Synthesis of Heterogeneous and Partially Reconfigurable Systems,” *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1-8.

[34] B. Blodget, S. McMillan, and P. Lysaght, “A Lightweight Approach for Embedded Reconfiguration of FPGAs,” *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, IEEE Computer Society, 2003, p. 10399.

[35] J. Williams, N. Bergmann, and A. Brisbane, “Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip.”

[36] A. Zeineddini and K. Gaj, “Secure partial reconfiguration of FPGAs,” *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. 155-162.

[37] J. Galindo, E. Peskin, B. Larson, and G. Roylance, “Leveraging Firmware in Multichip Systems to Maximize FPGA Resources: An Application of Self-Partial Reconfiguration,” *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, 2008, pp. 139-144.

[38] H. Austad, *A Survey of Real-Time Scheduling algorithms for the Linux Kernel*, 2008.

[39] R. Pellizzoni and M. Caccamo, “Adaptive Allocation of Software and Hardware Real-Time Tasks for FPGA-based Embedded Systems,” *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, 2006, pp. 208-220.

[40] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *Computers, IEEE Transactions on*, vol. 53, 2004, pp. 1393-1407.

[41] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," *IN PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ARCHITECTURES (ERSA*, 2003, pp. 284--287.

[42] K. Danne and M. Platzner, "A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware," *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 568-573.

[43] Atmark Techno, "Suzaku-V Hardware manual en," 2005.

[44] Atmark Techno, "Suzaku-v Software manual en," 2005.

[45] X. Inc, *Virtex-4 Configuration Guide (UG071)*, January, 2007.

[46] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*, O'Reilly Media, Inc., 2005.

[47] Xilinx, "XPS HWICAP (v1.00.a)," 2007.

[48] K. Curtis, *After the Software Wars*.

[49] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs," *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1-6.