NTNU
Norwegian University of
Science and Technology

Hardware-software intercommunication in reconfigurable systems

Vegard Haugen Endresen

Master of Science in Electronics
Submission date: June 2010
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

In earlier work a framework for self reconfigurable systems has been tested. For a more flexible solution there is a need for a system for exchanging data between programs running on an embedded processor and reconfigurable modules. Using such a system for exchanging input and output data will provide basic functionality when placing processing in hardware and can applied to make hardware processes persistent through interruption and resmumption.

This thesis consists of the following tasks:

- Suggest and analyze methods for handling state, input,output and static program data for reconfigurable modules

- Test a selection of methods on FPGA and embedded processor

- Show how the data exchange system can be used as part of a larger system, like a video-scaler, and if time allows implement and test this integration.

Assignment given: 16. January 2010
Supervisor: Kjetil Svarstad, IET

NTNU
Norwegian University of Science and Technology

Hardware-software intercommunication in reconfigurable systems

Candidate:	Vegard Endresen
Candidate title:	Master of Science in Electronics
Submission date:	June 7, 2010
Supervisor:	Professor Kjetil Svarstad

Summary

In this thesis hardware-software intercommunication in a reconfigurable system has been investigated based on a framework for run time reconfiguration. The goal has been to develop a fast and flexible link between applications running on an embedded processor and reconfigurable accelerator hardware in form of a Xilinx Virtex device.

As a start the link was broken down into hardware and software components based on constraints from earlier work and a general literature search. A register architecture for reconfigurable modules, a reconfigurable interface and a backend bridge linking reconfigurable hardware with the system bus were identified as the main hardware components whereas device drivers and a hardware operating system were identified as software components. These components were developed in a bottom-up approach, then deployed, tested and evaluated.

Synthesis and simulation results from this thesis suggest that a hybrid register architecture, a mix of shift based and addressable register architecture might be a good solution for a reconfigurable module. Such an architecture enables a reconfigurable interface with full duplex capability with an initially small area overhead compared to a full scale RAM implementation. Although the hybrid architecture might not be very suitable for all types of reconfigurable modules it can be a nice compromise when attempting to achieve a uniform reconfigurable interface.

Backend bridge solutions were developed assuming the above hybrid reconfigurable interface. Three main types were researched: a software register backend, a data cache backend and an instruction and data cache backend. Performance evaluation shows that the instruction and data cache outperforms the other two with an average acceleration ratio of roughly 5-10. Surprisingly the data cache backend performs worst of all due to latency ratios and design choices. Aside from the BRAM component required for the cache backends, resource consumption was shown to be only marginally larger than a traditional software register solution. Caching using a controller in the backend-bridge can thus provide good speedup for little cost as far as BRAM resources are not scarce.

A software-to-hardware interface has been created has been created through Linux character device driver and a hardware operating system daemon. While the device drivers provide a middleware layer for hardware access the HWOS separates applications from system management through a message queue interface. Performance testing shows a large increase in delay when involving the Linux device drivers and the HWOS as compared to calls directly from the kernel. Although this is natural, the software components are very important when providing a high performance platform.

As additional work specialized cell handling for reconfigurable modules has been addressed in the context of a MPEG-4 decoder. Some light has also been shed on design of reconfigurable modules in Xilinx ISE which can radically improve development time and decrease complexity compared to a Xilinx Platform Studio flow. In the process of demonstrating run time reconfigurations it was discovered that a clock signal will resist being piped through bus macros. Also broken functionality has been shown when applying run time reconfiguration to synchronous designs using the framework for self reconfiguration.

Table of Contents

Front page .... **I**

Summary III

Table of contents IV

1: Introduction **1**

1.1: Reconfigurable hardware 1

1.2: Run time reconfiguration 2

1.3: AHEAD 3

1.4: Contributions 3

2: Theory **4**

2.1: Sukzaku-V 4

2.2: uCLinux Atmark distribution 5

2.3: Linux kernel 5

2.4: Development tools 6

2.5: Hardware acceleration 6

3: Previous work, thesis basis and goals **9**

3.1: A framework for run time reconfiguration 10

3.2: HWOS – the application interface 10

3.3: Thesis basis and goals 11

4: Problem description **11**

5: Design **13**

5.1: Register Architecture **13**

5.1.1: Register building blocks 13

5.1.2: Reconfigurable interface 15

5.1.3: Generic register architectures 17

5.1.4: Sequential multiplier 18

5.2: Backend bridge **19**

5.2.1: Software register backend 22

5.2.2: Single port BRAM 23

5.2.3: Dual port BRAM 24

5.2.4: Combining BRAM and software registers 26

5.2.5: Data cache backend 26

5.2.6: Instruction and data cache 27
5.2.7: Backend synthesis 27
5.3: Device driver encapsulation 28
5.4 Software components 29
5.4.1: Dynamic memory allocation 29
5.4.2: HWOS – The application interface 30
6: Verification 32
6.1: Backend 32
6.2: Sequential multiplier 33
6.3: Device driver 34
6.4: HWOS 35
7: Performance evaluation 36
8: Case: MPEG transcoder 36
8.1: Source analysis 36
8.2: Specialized cells 39
8.3: MPEG decoder RAM migration 39
9: Run time reconfiguration 40
10: Results 42
10.1: Register architectures 42
10.2: Backend bridge 46
10.3 Performance Evaluation 48
10.3.1: Kernel Space 48
10.3.2: User Space 50
10.4 HWOS 52
10.5 Runt time reconfiguration 53
11: Discussion 55
12: Conclusion 57
13: Further work 58
14: References 58
15: Thanks to 58

1: Introduction

1.1: Reconfigurable hardware

The hardware and software domains are two different domains in many ways. One of the most important distinctions is performance versus flexibility. Traditionally ASICs have been used in high performance applications where processors cannot provide the required processing power. Graphic accelerators are examples of products from the mainstream ASIC market where high performance hardware is utilized to cope with high processing requirements. On the other hand a processor is very flexible compared to an ASIC solution as its behavior is to execute one or more programs that can be modified any number of times. The generality of most processors' instruction set will also allow programs to perform virtually any kind of processing though the performance suffers from the very same generality.

Another important field that comes into play when comparing the hardware and software domain is economics. As processors are general purpose they are relatively cheap since production cost can be amortized over a large product volume. ASICs are application specific which generally leads to higher price since an average ASIC product volume is less than for a general purpose processor. Another difficulty is the reduction of transistor sizes which causes an increase in complexity. The increase in complexity manifests itself as extra costs in the hardware production process. This reaction is commonly presented as the productivity gap illustrated in figure 1.1.1

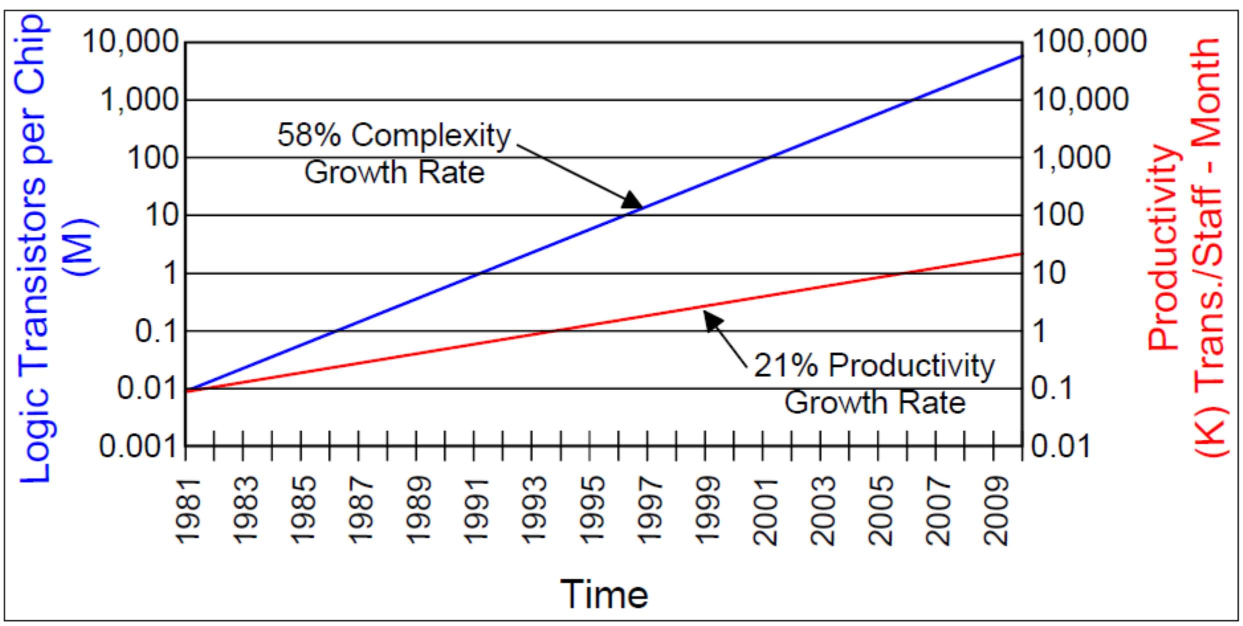


Figure 1.1.1: The productivity gap

The productivity gap states that the available chip transistor complexity grows faster than the ability to utilize it. To compensate for the increasing complexity a longer design process is required which adds to the total costs. Because technology trends it thus becomes more and more important to be able to amortize a product over a large volume in order to achieve a reasonable prices while running a profitable business.

Many approaches try to address and solve the previously mentioned problems; reconfigurable hardware is one of them. Reconfigurable hardware is as the word suggests hardware that can be modified as opposed to hard silicon implementations. In these terms reconfigurable hardware is thus general purpose hardware and is mostly associated with FPGAs. While reconfigurable hardware provides flexibility through its reconfiguration property, a major drawback is that it will be outperformed by hard silicon. On the other hand reconfigurable hardware can provide high performance compared to a processor.

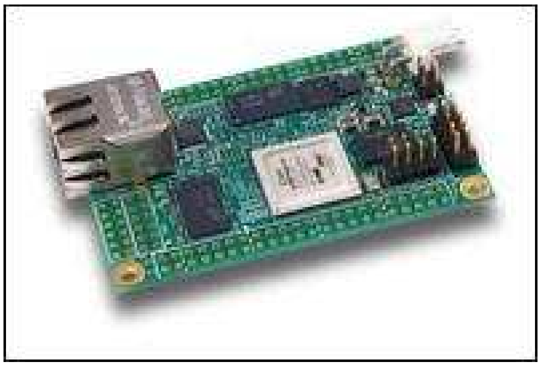


Figure 1.1.2: Suzaku-V, deploying reconfigurable hardware

Traditionally FPGAs have been used for prototyping of hard silicon designs but has become more and more accepted as a product technology covering the increasing gap between ASICs and processors [1]. One field of reconfigurable hardware that is in particular focus in this report is run time reconfiguration.

1.2: Run time reconfiguration

Run time reconfiguration in this context is the concept of reconfiguring hardware while active components are executing on it. Traditionally programming and executing hardware on a FPGA have been considered as two separate processes, but as suggested in [1] and [2] run time reconfiguration can increase both flexibility and performance of systems implemented on FPGA. Reconfiguration time is a critical factor when applying run time reconfiguration as it will lock parts of the target hardware and possibly occupy other system resources. The base line is that the reconfiguration process will incur overhead.

For many systems it is not possible to perform a full run time reconfiguration because the reconfiguration process or the system itself might requires components implemented in the reconfigurable hardware. By performing a partial reconfiguration, such critical components can be left untouched while only safe regions are modified. An immediate advantage with partial reconfiguration is that it is efficient as reconfiguration can be applied only to the regions requiring it.

1.3: AHEAD

This thesis is a contribution to the AHEAD project at NTNU Norway. AHEAD specifies the use of reconfigurable hardware in elements called tags which purpose is to act as co-processing units for mobile clients. An illustration is given in figure 1.3.1

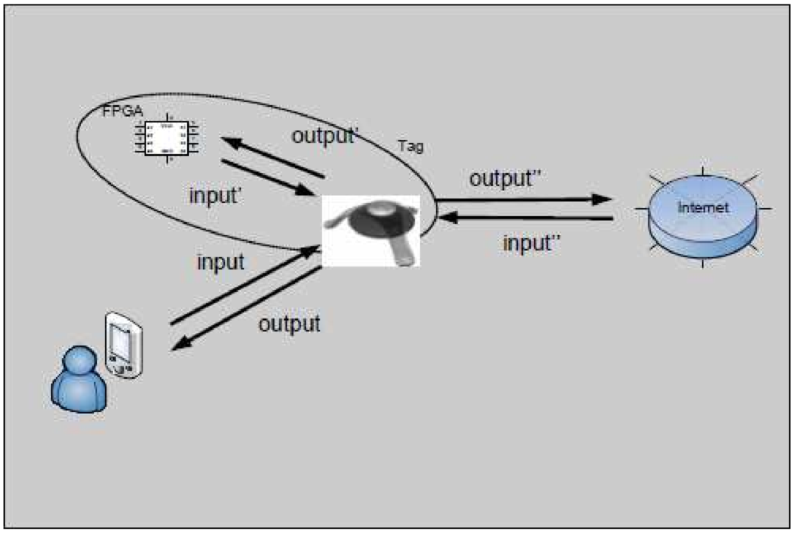


Figure 1.3.1: AHEAD system sketch

The main goal is to create a flexible platform with high performance using reconfigurable hardware. Distributing the platform through tags in a mobile device environment can thus provide powerful applications in a flexible way.

1.4: Contributions

In more specific details this thesis has contributed to software and hardware intercommunication based on the work on run time reconfiguration in [3]. Different register architectures for reconfigurable modules, reconfigurable hardware entities, have been explored together with firm backend solutions as a bridge between the system bus and application hardware. Supporting context switching of reconfigurable hardware has been emphasized. As embedded Linux has been an assumption in this thesis, Linux device drivers has been developed for providing user space access to hardware. A hardware operating system, HWOS, as suggested in [3] has been developed as a top level control mechanism with focus on acting as an application interface, separating applications from system management. Different solutions and parts have been tested during the design process to evaluate goodness, which together form a communication link from hardware to software for intentional use in reconfigurable systems.

As an additional contribution, specialized cell requirements for reconfigurable modules have been addressed through the concept of cell migration. Design of reconfigurable modules in Xilinx ISE has also been addressed to improve on the Xilinx Platform Studio flow which suffers from both long build time and manual rerouting complexity. Finally, reconfiguration of synchronous modules using the framework for self reconfiguration has been applied revealing some challenges ahead.

2: Theory

2.1: Suzaku-V

All implementation and testing of concepts in this thesis has been done on a platform called Suzaku developed by Atmark Techno. More specifically a Suzaku-sz410 platform has been used which is a part of the Suzaku-V family. An illustration of the Suzaku-V architecture is shown in figure 2.2.1.

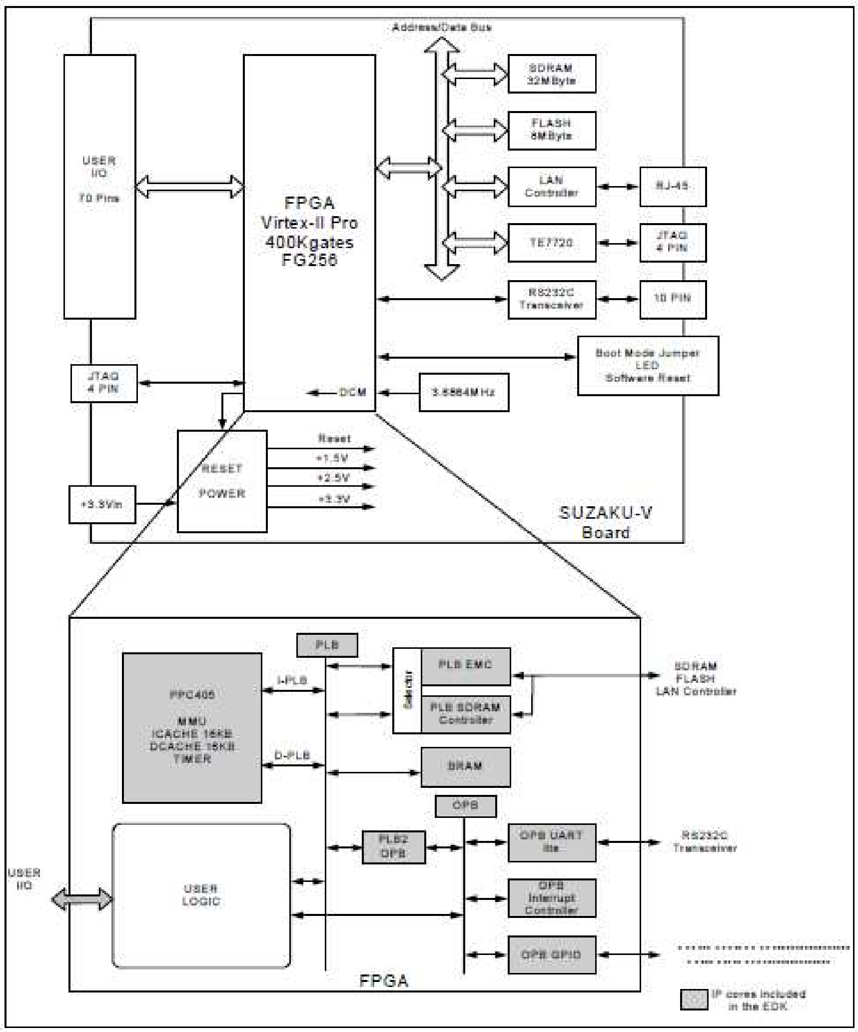


Figure 2.2.1: Suzaku-V architecture

The Suzaku-V platform consist a set of firm hardware components along with an embedded Xilinx Virtex FPGA which serves as reconfigurable hardware. The firm hardware part consists of a PowerPC processor, different I/O modules, memory, bus connections and pins. Flash memory serves as non-volatile memory which amongst other things used to store a Linux image and FPGA configuration data. Serial port, JTAG and Ethernet controllers constitutes basic user interface for programming and accessing the platform.

As mentioned the Xilinx Virtex FPGA on the FPGA provides reconfigurable hardware. It should be noted that the Suzaku-V family offers both a Virtex-II pro and a Virtex-IV FPGA through the models sz310 and sz410 respectively. Figure 2.2.1 assumes a sz310 board but the overall family architecture is very similar

In the bottom of figure 2.2.1 the FPGA is shown, containing a set of components. The FPGA contains two buses, OPB and PLB, and a set of modules. OPB and PLB are abbreviations for On-Chip Peripheral Bus and Processor Local Bus respectively. All system modules are connected to at least one of the two buses, and is due to this relation called a peripheral. The OPB and PLB buses in figure 2.2.1 connect the PowerPC processor with the firm hardware on the board through controller modules, marked as gray shaded entities. This configuration of the FPGA is required for running Linux on the Suzaku-V and is also referred to by Atmark as a basic or minimal configuration.

Building a custom system that takes advantage of the Linux is done by extending the minimal configuration. Custom hardware modules can be added to the FPGA configuration and if required connected to the PowerPC processor by encapsulating them in peripherals.

2.2: uCLinux atmark-distribution

Atmark has developed a custom Linux distribution for the Suzaku-V platform which is based on uClinux. While the Linux kernel manages hardware resources the distribution provides applications that are very useful for development and verification. Amongst other things, busybox provides a Linux command line which can be linked with a development computer through serial port. NFS, network file system, allows fast and flexible file transfers through network mounted file systems.

Another powerful tool is the GCC framework. As the uCLinux atmark-distribution does not include any compilers, cCompilation of programs for the Suzaku's embedded processor is instead done on a development computer (i.e. cross compilation). Although it is possible to do onboard compilation one should keep in mind that resources are limited on embedded platforms such as Suzaku.

2.3: Linux kernel

In Linux hardware is managed by the kernel and user space applications rely on accessing hardware through it. Access of hardware on the Suzaku platform is based on a global address map defined by the user in system configuration tools. A PowerPC core is included as default CPU in the Suzaku-V series which uses MMU. Hardware access is therefore not straight forward though as the Linux kernel deploys a real to linear address map and addressing must therefore done explicitly through the kernel API. Source 2.3.1 shows a sufficient but not necessary selection of API functions for this purpose.

```
/*Map address space*/
void *ioremap(unsigned long offset, unsigned long size);
/*Unmap address space*/
void iounmap(void * addr);
/*Write data*/
int writel(void * addr,int data);
/*Read data*/
int readl(void * addr);
```

Source 2.3.1

2.4: Development tools

Systems development for the Suzaku platform can be divided into two main parts, hardware development and software development. For designing and deploying hardware, Xilinx tools are required. Xilinx Design Studio 10.1 is the tool set used in this thesis.

Though Xilinx provide a SDK for software development an equally good option for the Suzaku platform is to develop software outside the SDK. Compilation can be done through GCC using cross compile tools found at [4]. [5] contains a good description of how to develop a basic system for the Suzaku platform.

Deploying both hardware and software can be done by transferring files from development computer to a Suzaku board through *hermit*. A less time consuming alternative for deploying software is by utilizing the NFS feature as shown in [3] and [6].

When it comes to debugging Minicom links the Suzaku's Busybox shell to a console on a dev-computer. Software debugging using print statements is thus a validation mechanism that can be applied to hardware as well through kernel modules. Files can also be created in the Linux file system for testing, reconfiguration and other purposes.

2.5: Hardware acceleration

AHEAD's purpose is to use dynamic reconfiguration for co-processing. One of the subsequent goals is to achieve speedup by doing processing in hardware, where a reconfigurable module is the equivalent of an accelerator. [7] discuss the process of accelerator design where the first step is speedup evaluation in form of equations (1) and (2).

$$t_{accel} = t_{in} + t_x + t_{out} \tag{1}$$

$$S = n(t_{CPU} - t_{accel}) \tag{2}$$

Equation (1) describes the total time taken running a task on an accelerator incorporating transferring input and output data. Equation (2) describes the total speedup of running n tasks on the accelerator compared to running n tasks on a CPU. This paper focuses on infrastructure for a dynamic reconfigurable system so the execution time t_x in (1) is not of particular interest as it is related to accelerator implementation. Data transfer time will, however, have a significant impact on the speedup. The data transfer time may depend on a number of factors like architecture topology, bit-widths, clock frequency, buffering and exploitation of parallelism and so on. A system infrastructure will partially determine some of the characteristics and thus also have an impact on accelerator performance which should be considered during the design process.

In addition to the conventional transmission of input and output data for a processing element, transmission of state data is also considered in the proceeding equations. This attempts to describe performance for interruptible processing elements. Consider two versions of a hardware task, a persistent version which can be resumed after it has been interrupted and a volatile version which has to restart prior to an interrupt. Factoring out all the common contributions to their run time, equations (3) and (4) describe their relative run time.

$$T_{PERSISTENT} = 2T_{STATE} + T_{RESUME} \tag{3}$$

$$T_{VOLATILE} = T_{FULL} \tag{4}$$

In order for the persistent versions to complete before the volatile version, equation (5) has to be met.

$$T_{RESUME} < T_{FULL} - 2T_{STATE} \tag{5}$$

Assuming that the data transfer bus and the processing core operate on the same clock frequency and disregarding any extra time penalty or overhead involved in state data transfer, satisfying (5) with respect to T_{STATE} involves two important factors: the encoding of the task state and the bit width of the bus used to transfer the data to persistent storage.

Consider first a state automata in figure 2.5.1 consisting of N states.

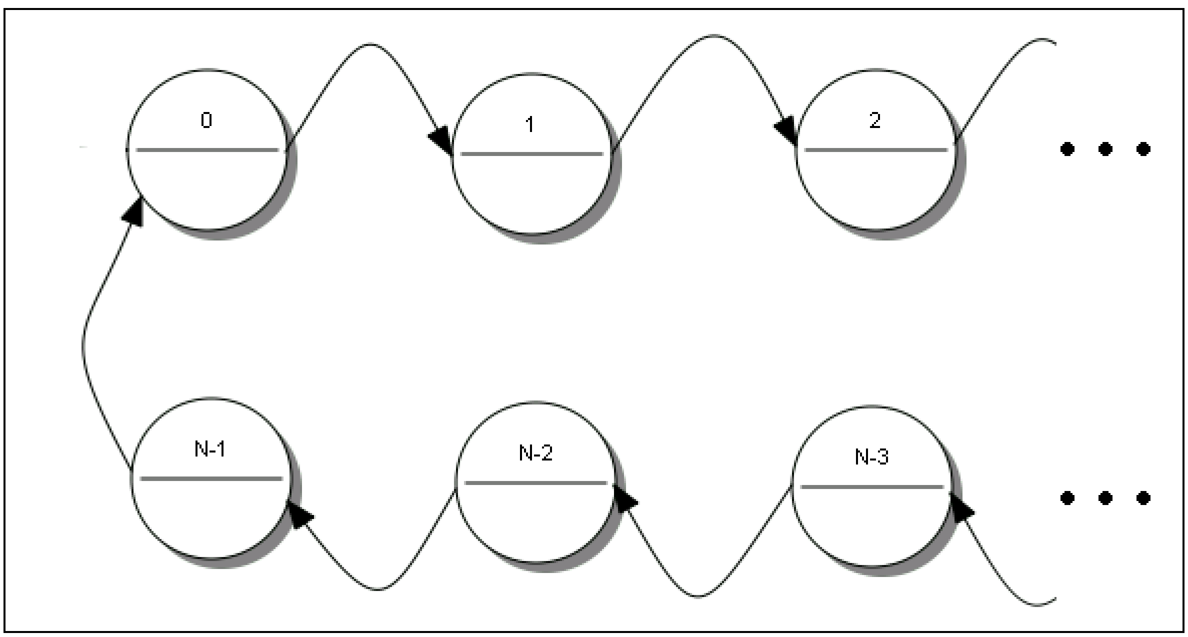


Figure 2.5.1: Finite state automata

Its state can be encoded compact by $\lceil log_2(N) \rceil$ bits. Assuming a compact encoding and with the above clock assumptions, equation (5) can be expressed as equation (6). C denotes clock cycles, N denotes the number of states and B denotes the bit width of the bus used for data transfer.

$$C_{RESUME} < N - 2\left\lceil \frac{\lceil log_2(N) \rceil}{B} \right\rceil \tag{6}$$

Using a hot-one encoding will, however, produce equation (7) from equation (5)

$$C_{RESUME} < N - 2\left\lceil\frac{N}{B}\right\rceil \tag{7}$$

In any case the persistent version will not always be faster than the volatile version, but the persistent version might be statistically superior if (8) is satisfied.

$$P(T_{Persistent} < T_{Volatile}) > 0.5 \tag{8}$$

For the above example this is equivalent to (9), assuming a uniform probability for interruption within each state.

$$C_{RESUME} < \frac{N}{2} \tag{9}$$

As N becomes large, equation (6) yields a large statistical gain even with a serial link for data transfer, B = 1. Equation (7) however depends entirely on the bit width of the data link. A bit width larger than 4 is required to make the persistent version statistically faster.

Achieving a speed gain through interrupt support for hardware thus requires taking both expensiveness of state encoding and the speed of data transfer into account. From a designer's point of view, the composition of a hardware module will place constraints on the data transfer mechanism in order to make a persistent version profitable in terms of speedup. If the composition of a hardware core allows it, storing parts of the entire state space might be feasible. Using the latter approach will result in some loss of progress as a tradeoff for less complex persistent state management. The above equations are based on a basic example circuit and do not consider extra delays which may be inflicted by other system components. In any case all this seems to point out that some sort of analysis should be done in order to decide to what extent a hardware module should be made persistent. Even though an analysis might be strongly simplified it could suggest the feasibility of a fully persistent, partially persistent or completely volatile hardware implementation.

3: Previous work, thesis basis and goals

3.1: A Framework for run time reconfiguration

In [3] Sverre Hamre describes a *framework for self reconfiguration* that he has developed. The framework is a collection of programs written in C for Xilinx Virtex bitstream manipulation. The physical hardware manipulation is performed by a program using a Linux device driver that interacts with the Virtex ICAP interface. In terms from section 2, the framework is a partial run time reconfiguration mechanism. Due to the ICAP dependency the framework is intended for use on Xilinx Virtex devices but has only been proved correct for the Virtex-IV architecture. Work in [6] has added some documentation on the practical use of the framework and confirmed that it is working correctly on a Virtex-IV.

In order to do develop a dynamically reconfigurable system number many problems must be solved and decisions have to be made. Figure 3.1.1 illustrates some of the architectural assumptions the *framework for dynamic reconfiguration* makes.

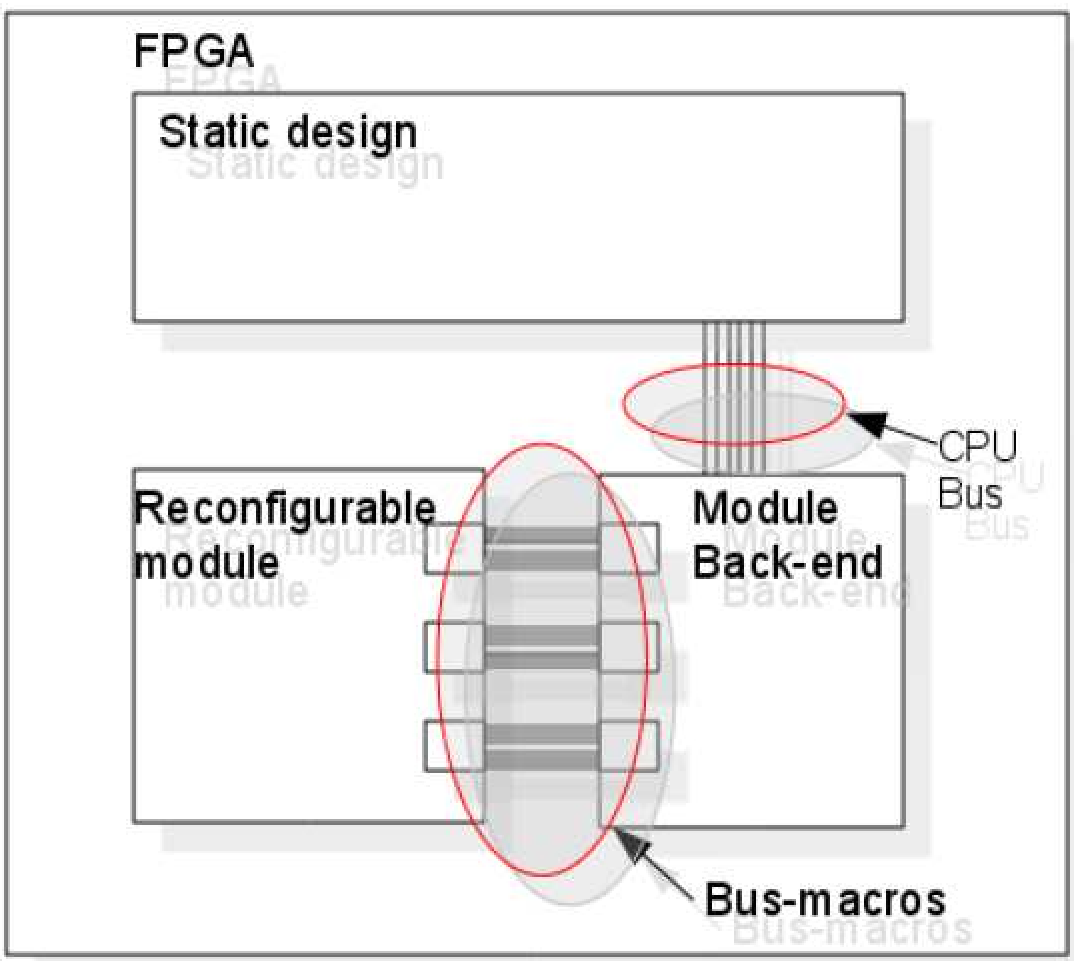


Figure 3.1.1: Framework for self reconfiguration architecture

[3] divides the reconfigurable system into three different categories. The static category contains parts of the system that will never be reconfigured. Communication infrastructure and essential components that cannot be altered without corrupting the system falls into this category. The system is also assumed to contain one or more backend modules which in turn are connected to reconfigurable modules. As illustrated these modules provide a communication link between reconfigurable modules and the static part of the system. Backend modules are not to be reconfigured so they are in this sense part of the static system. As indicated by figure 3.1.1 reconfigurable modules are connected to a backend module through components called bus macros.

In order to achieve functionally correct run time reconfiguration of a module it is essential that all wiring in and out of the physical region being altered is identical before and after reconfiguration, but because physical mapping and routing of a hardware design to a platform is performed by a CAD tool there is no guarantee for this equivalence.. To solve this problem reconfigurable modules are therefore connected to the rest of the system through a fixed number of wires with fixed placement also known as bus macros. The connection between a reconfigurable module and the rest of the system is also known as a reconfigurable interface in reference literature.

3.2: HWOS

In order to manage a dynamically reconfigurable system run time, the concept of a HWOS was formed in [3]. Figure 3.2.1 shows the original illustration of the HWOS concept.

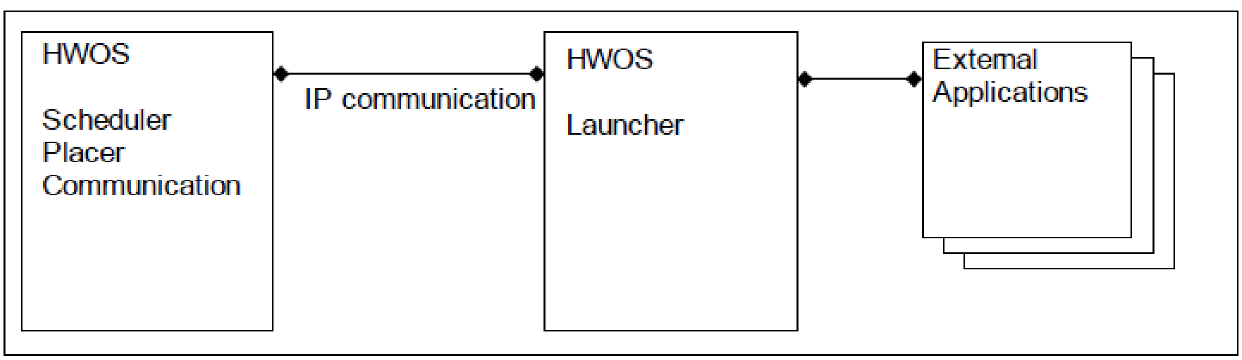


Figure 3.2.1: HWOS

A more detailed description of the HWOS idea can be found in [3] but the basic principle is to use a software application for managing the dynamically reconfigurable system run time. As shown in figure 3.2.1 the HWOS acts as the interface for external applications wanting execution time on reconfigurable hardware on the platform. In addition it consists of parts for managing run time execution and reconfiguration.

3.3: Thesis goals

In [6] the *framework for dynamic reconfiguration* was explored and some of the work Sverre Hamre left was pursued. One of the recommendations for further work in [3] was to have a look at the possibility for transferring data between a reconfigurable module and CPU prior to reconfiguration. Implementing saving and loading of state data in reconfigurable modules will allow hardware modules to be used like processes in multitask-systems which is an interesting concept. This problem is generalized a bit more in this paper to comprise the overall communication between CPU and reconfigurable modules. Results from [6] support the hypothesis in [3] that loading and storing state data using a CPU program that operates on ICAP will be quite slow.

This thesis aims to explore different architectures and solutions to create a fast link between reconfigurable modules and a CPU as the dynamically reconfigurable system's centralized control unit. In addition to normal data flow the link should support context switching to enable interruption and resumption of reconfigurable modules. As a pure software solution seems to be slow there will be a need for developing hardware components for this purpose. Parts of the solution will thus be hardware oriented while other parts will be software oriented. As part of software exploration it is natural to link development to the HWOS concept.

4: Problem decomposition

Before commencing design, an overview of the system and the components that were to be researched and developed was made. Figure 4.1 illustrates the system composition with hardware components in the lower parts, the Linux kernel as a middleware layer and a HWOS as a top level interface to the platform.

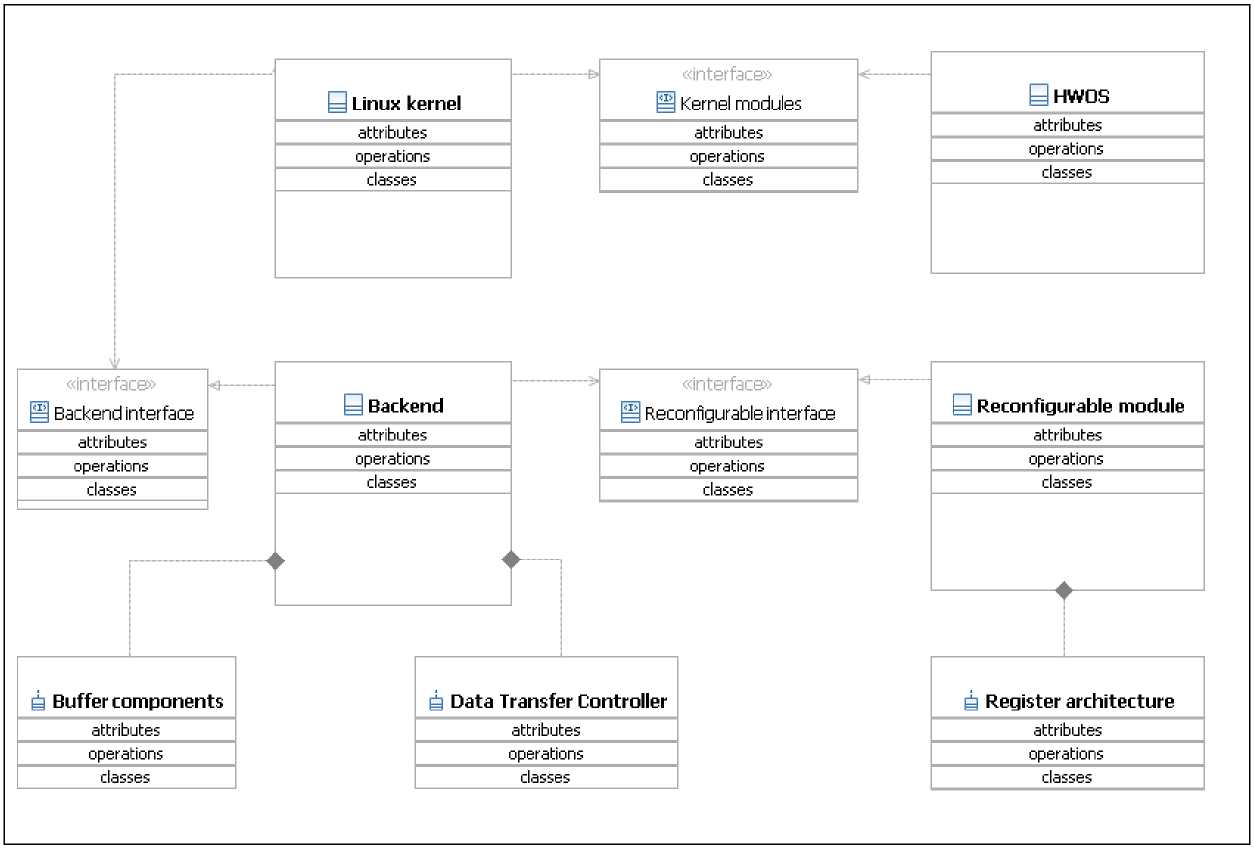


Figure 4.1 System decomposition

Reconfigurable modules need an identical interface towards the backend in order to be run time interchangeable. Creating a general and flexible reconfigurable interface was therefore considered an important task to provide a good foundation for run time reconfiguration. One dependency which had to be determined was choosing the register-architectures that were to be used in the reconfigurable modules for data transfers.

Transporting data between a backend and a reconfigurable module requires buffers and controllers, which is also the case for the transportation of data between the backend and external memory. With these above hardware components there would be a need for working with the Linux kernel for user space access to the hardware. For managing the dynamically reconfigurable system run time, the HWOS was listed as the top controlling entity.

With respect to the system decomposition in figure 4.1, components for the dynamically reconfigurable system were developed and assembled in a in a bottom-up approach.

5: Design

5.1 Register architectures

When researching register architectures for the reconfigurable module in the context of dynamic reconfiguration, not many references could be found. Disregarding the dynamic reconfiguration topic and looking into normal register architectures, it was decided to develop generic models of a fully addressable RAM architecture, a shift based architecture and a hybrid of these two. Mapping these architectures onto FPGA and analyzing synthesis data combined with simulations could give some pointers to strengths and weaknesses.

5.1.1: Register building blocks

To create a basis for different register architectures, different register building blocks were created in VHDL. The basic idea was that a high usage of structural description would ease the design process through reuse and a clean component hierarchy. Three different main types of register blocks for were developed which are now shortly presented.

Figure 5.1.1.1 shows the composition of the basic shift block. It has a serial input and output, *shift_i* and *shift_o,* for data transfer with the back end along with parallel input and output, *par_i* and *par_o,* for internal data operations. Two control signals *write* and *shift* manage select between the write and shift data operations respectively.

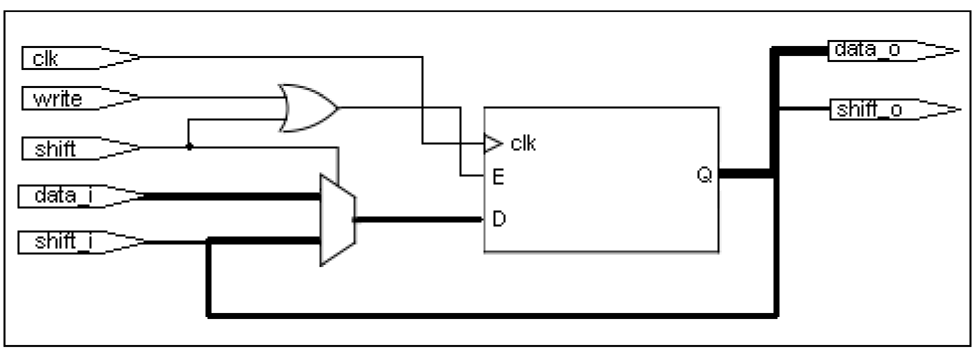


Figure 5.1.1.1: Serial shift block

The behavior of the parallel shift component in figure 5.1.1.2 is identical to the serial shift block apart from extended bit-width on the shift input and output channels. Chaining of smaller regular registers in addition to some control logic produces the multiple-bit shift register block. The smaller registers are of uniform size where the size of each register represents the shift width and the sum of their bit widths represents the total bit width of the register macro they constitute.

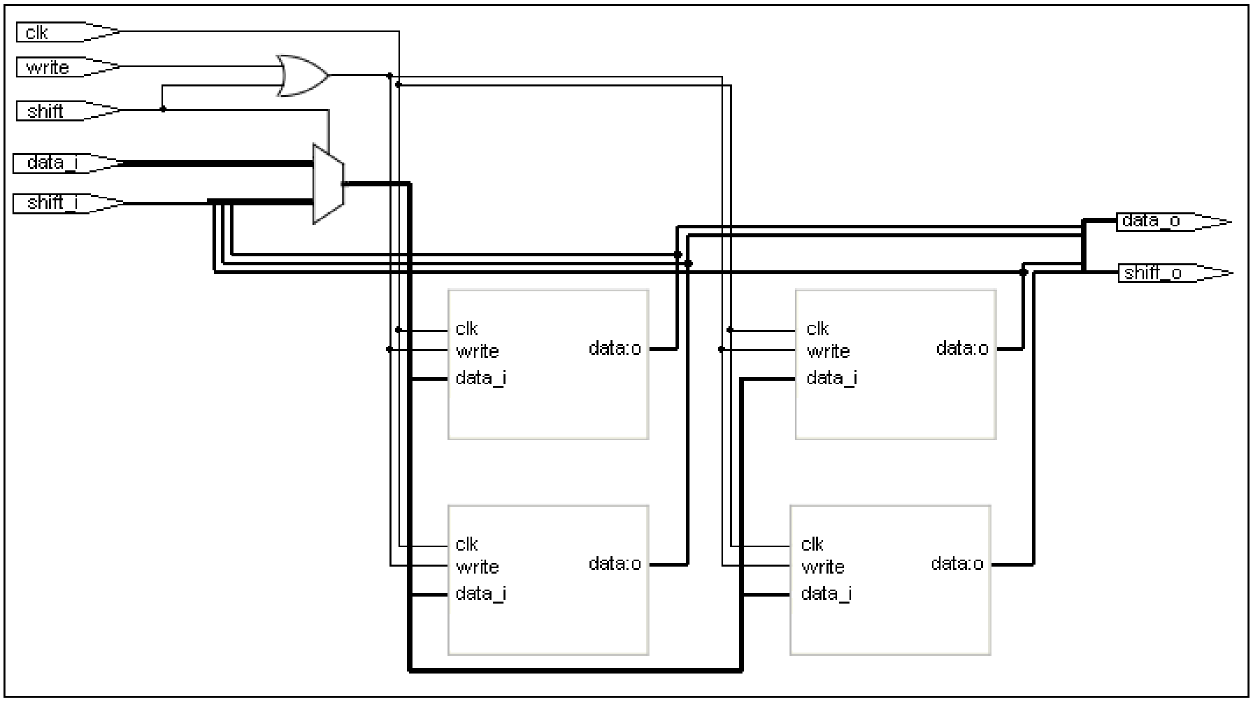


Figure 5.1.1.2: Parallel shift block

The basic addressable block is a normal register realized as a set of DFFs with a *write* control signal for managing writing. In addition there is multiplex logic to be able to select from two possible sources, namely backend data or internal data.

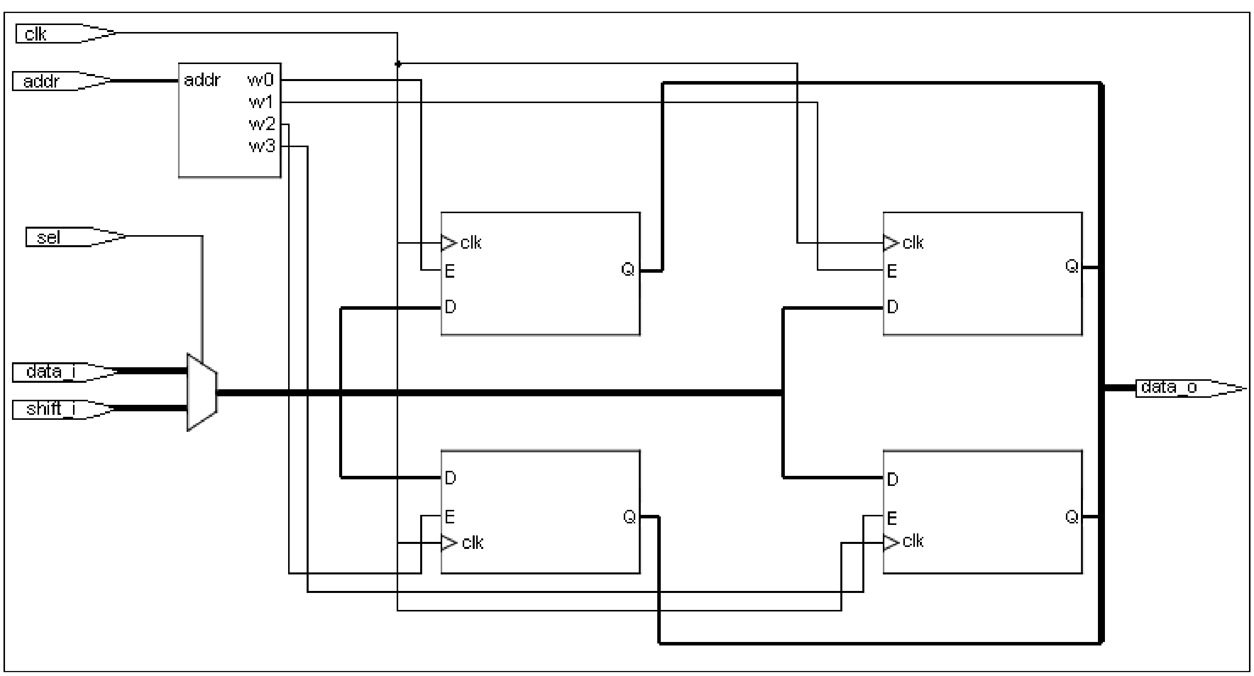


Figure 5.1.1.3: Addressable block

Synthesis in Xilinx ISE confirmed that the different building blocks required the same amount of resources in terms of LUTS and slices. The number of I/O pins, summarized analytically in table 10.1.1, will vary however.

In proper use the register IO pins are linked to signals and is thus an indicator of resource consumption. The overall architecture must however be factored in before making a more complete profile. In addition to the above blocks, some different variants were developed with different generics, inputs and outputs, barrel shift capability and so forth. The VHDL *open* statement was acknowledged to minimize the need for too many variants of the register building blocks.

5.1.2: Reconfigurable interface

Based on the proposed register architectures and building blocks different interfaces between the back-end and the reconfigurable module were researched. Instead of using standardized transfer protocols like SPI, USB or PCI, proprietary protocols linked more directly to the reconfigurable module's register architecture were researched. The PowerPC PLB bus works as a standardized data transfer mechanism all the way to the backend anyway and in this thesis it was emphasized to evaluate general interfaces instead of attempting to determine the goodness of existing standards.

Furthermore emphasis was put on creating interfaces of modest size both for simplicity and in order not to lock too many FPGA resources.. Handling reconfiguration requirements from [3] is likely to become more complex as the reconfigurable interface grows. In this context is should be noted that a single instance of Xilinx bus macros obtained as attachments to [3] has a default bit width of 8 bits, so in order to fully utilize the capability the number of wires in each interface should be dividable by 8. Bus macros in the form of LUT macros are unidirectional though so a full utilization might be hard to achieve.

As the different register architectures are based on different principles, the corresponding interfaces became different but shared some common signals. The interfaces that were composed are illustrated in figure 5.1.2.1 and are now shortly presented and discussed.

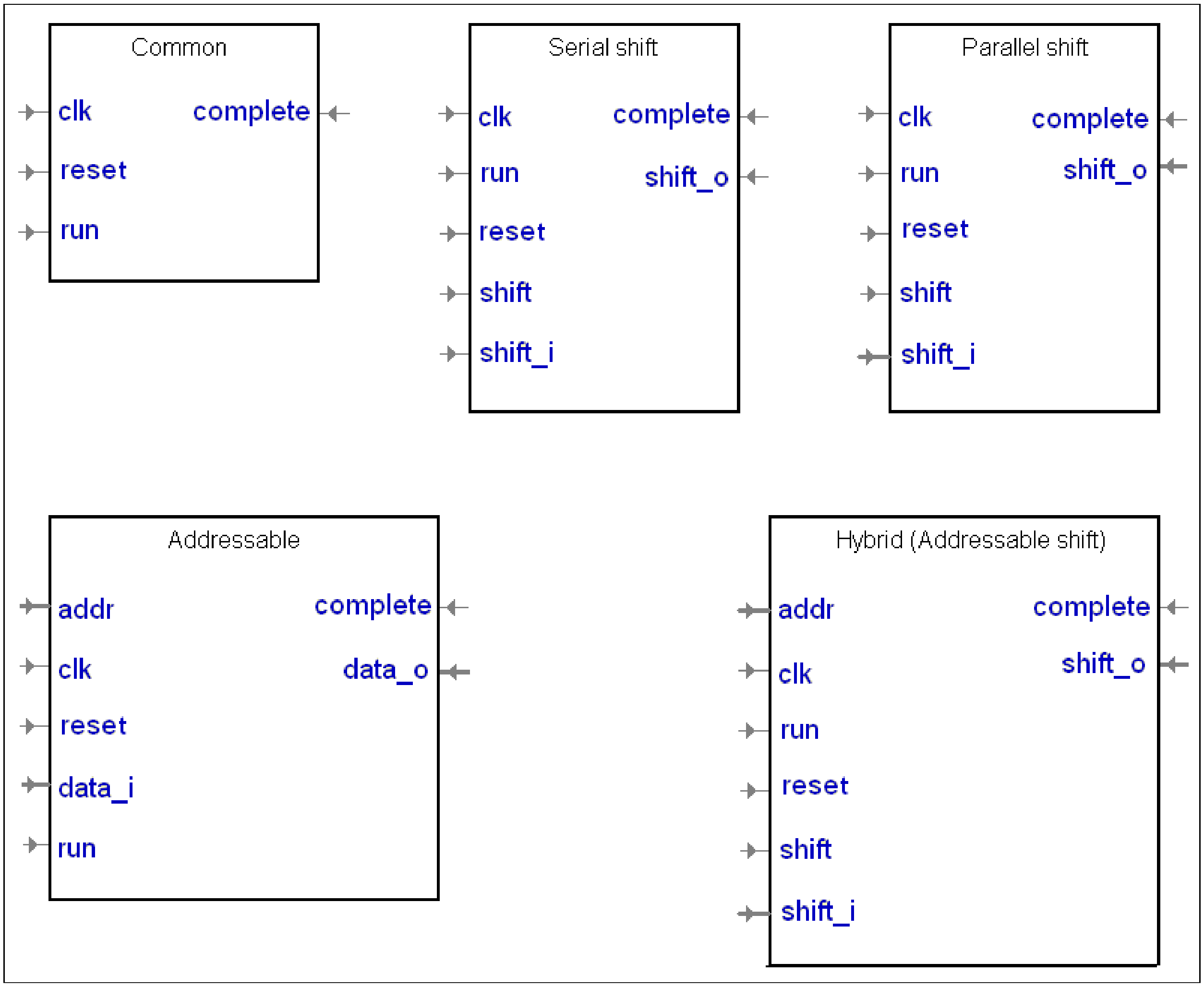


Figure 5.1.2.1 Reconfigurable Interfaces

Common:

As all signals connecting a reconfigurable module with the rest of the system must be propagated through the reconfigurable interface, the clock signal *clk* is part of the common interface to support synchronous reconfigurable modules. *Reset* allows to put the reconfigurable module in a safe state. *Reset* will in turn control the mode of operation. The idea is that by toggling the value of *run* the reconfigurable module can be started, frozen and resumed, ensuring that the reconfigurable module's internal operations of does not interfere with back-end data transfer. Finally the *hw_complete* signal servers as a notification to the back-end that processing has been completed and a new job can be accepted.

Shift:

The serial shift interface extends the common interface with three signals: *shift_g,shift_i* and *shift_o*. , *Shift_g* signals a shift along one and only shift chain used in regular shift architecture. The reconfigurable module is responsible for propagating data through the shift chain where *shift_i* is the shift input and *shift_o* is the output. As with the building blocks the parallel shift interface is identical to the serial shift interface apart from the bit widths of *shift_i* and *shift_o*.

Addressable:

The addressable interface allows random access of registers in the reconfigurable module. In order to specify a source for a read or write operation the signal *addr* was added to the interface. *Write* specifies whether the selected is to be read or written. *Data_i* and data_*o* serve as data input and output respectively.

Hybrid (Addressable shift):

Last, the hybrid interface extends the parallel shift interface and the signal *addr* is added for selecting between different shift chains. It thus combines elements of both the addressable architecture and shift architecture.

5.1.3: Generic register architectures

In order to see the technology mapping of the different architectures and how they scale, generic models were created in VHDL descriptions and synthesized. None of the architectures were created with any local behavioral description, so in order to prevent the synthesis tool from reducing passive branches unused signals were looped. .

A small script system was created in order to automate synthesis of a register architectures for different generic values. First a Java program was developed that reads an input VHDL file and alters one or more generic fields determined by hard Java code. Since the four different architecture models contain different generics, four slightly different versions of the program were created.

Automated synthesis was performed through TCL scripts, one for each architecture model, which interacted with Xilinx ISE. All TCL scripts specified the target FPGA as a Xilinx Virtex 4 xc2vp4fx which is the FPGA available in the Suzaku-sz410 platform. Both a BAT script for MS Windows and a BASH script for UNIX/Linux were developed as the top control unit for automated synthesis. These scripts iterated the execution of the Java program in order to create VHDL files with correct generic parameters and the subsequent execution of the TCL script performing synthesis. Execution would provide a set of synthesized projects organized in a file structure. By inspecting the project files for a single ISE project it was found that a synthesis results were contained in a file *top_module_name.syr*. Another Java program was created to read the relevant data from the synthesis result files and write them into an M-file as arrays.

Automated synthesis was then performed with respect to I/O constraints. Data bit-widths for all architectures were set to 8 bit with exception of the serial architecture where the I/O bit-width is 1 by definition. Synthesis results can be found in section 10.1.

5.1.4: Sequential multiplier

As a case for a simple reconfigurable module, a sequential multiplier was developed. In addition to testing register architectures this provided a real exploration of persistent state hardware. In accordance with section 2.5 a multiplier might not be a very feasible module for interrupt support. A multiplier is nor a representative for the wide variety of different IPs that exists. It was chosen as a simple example that might demonstrate some important properties.

First a normal sequential multiplier was implemented as a VHDL description according to the architecture described in [8]. After adjustments and verifying that the behavior was correct, this worked as a template for implementing the different register architectures.

Interrupt implementation was done by replacing all memory in the original behavioral template, both explicit and implied, with different register building blocks. Due to the architecture of the multiplier some new modules were created instead of using only the original register blocks. he reason being that it was easier and possibly more efficient than bruteforcing a pure structural use of the original building blocks. As the implied memory was replaced with structural components it was necessary to use manually coded states and counters. Some of the register architectures do have similarities, which was utilized by reuse after implementation the multiplier in some architecture.

After implementing the different register architectures, the simulations were conducted through testbenches. To test the interrupt property, the multiplier was first run from scratch and stopped at an arbitrary point of time where DFF contents constituting the multipliers state was extracted. Table 5.1.4.1 shows a partial state for the computation of $0xABCDEF12^2$.

Parameter	**Value**
Cstate	SHIFTING(0x2)
Count	0x9
Operand_a	*0x00ABCDEF*
Operand_b	*0xABCDEF12*
Partial_result	0x5BF134F0A2000000

Table 5.1.4.1: Multiplier state

Another stimuli process was then created in the same testbench writing this state to the multiplier and resuming it. Some minor fixes were needed but eventually it could be verified that the multiplier produced the same result when started from a sub-state as from start for all the implementations.

In order to demonstrate a performance enhancing effect on the shift chain structure the signal *zerofill* was added to the addressable shift implementation of the multiplier. The basic idea was that when the multiplier is used for multiple jobs and no reconfiguration is performed, it would be profitable to reset the answer while scanning it out and scanning in new operands. *Zerofill* will simply incur a small change to the scan chain structure to achieve this. Increased performance can be achieved for the other architectures as well but the zerofill primitive is very simplistic.

Synthesis was performed for operands of 32 bit only. It was decided not to perform any further synthesis exploration as it was considered to require much work for results of potentially little importance.

5.2: Backend bridge

A quick literature search was performed when exploring possible backend solutions Articles like [7] and [9] refer to use of software accessible registers combined with a hardware controller, mentioning buffering as a way to achieve performance gain.

Based on simulations and the results from generic architecture exploration in section 10.1, the hybrid solution was considered to be the most promising one and served as the assumed register architecture for further development. Based on this decision the very general backend architecture illustrated in figure 5.2.1 was composed.

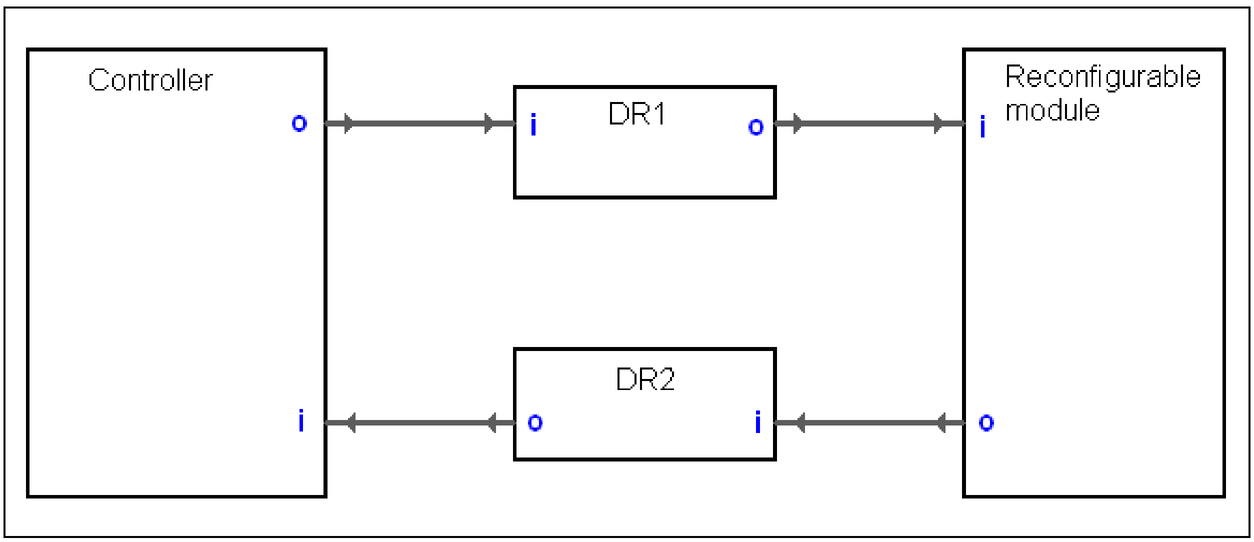


Figure 5.2.1: Backend template

The basic idea was that a controller would be responsible for transferring data between a reconfigurable module and some backend buffer mechanisms through two shift registers *DR1* and *DR2*. *DR1* would be linked as the first shift register in the shift-chain selected by the controller and *DR2* would be the last register in the same chain. A buffer mechanism would in turn be connected to CPU through the system bus. Based on this backend model some different solutions were contemplated.

The simplest backend, illustrated in figure 5.2.2, consists of three software registers, IR DR1 and DR2. The CPU can thus invoke instructions by altering the content of IR, and the data involved in such operations is determined by the contents of DR1 and DR2.

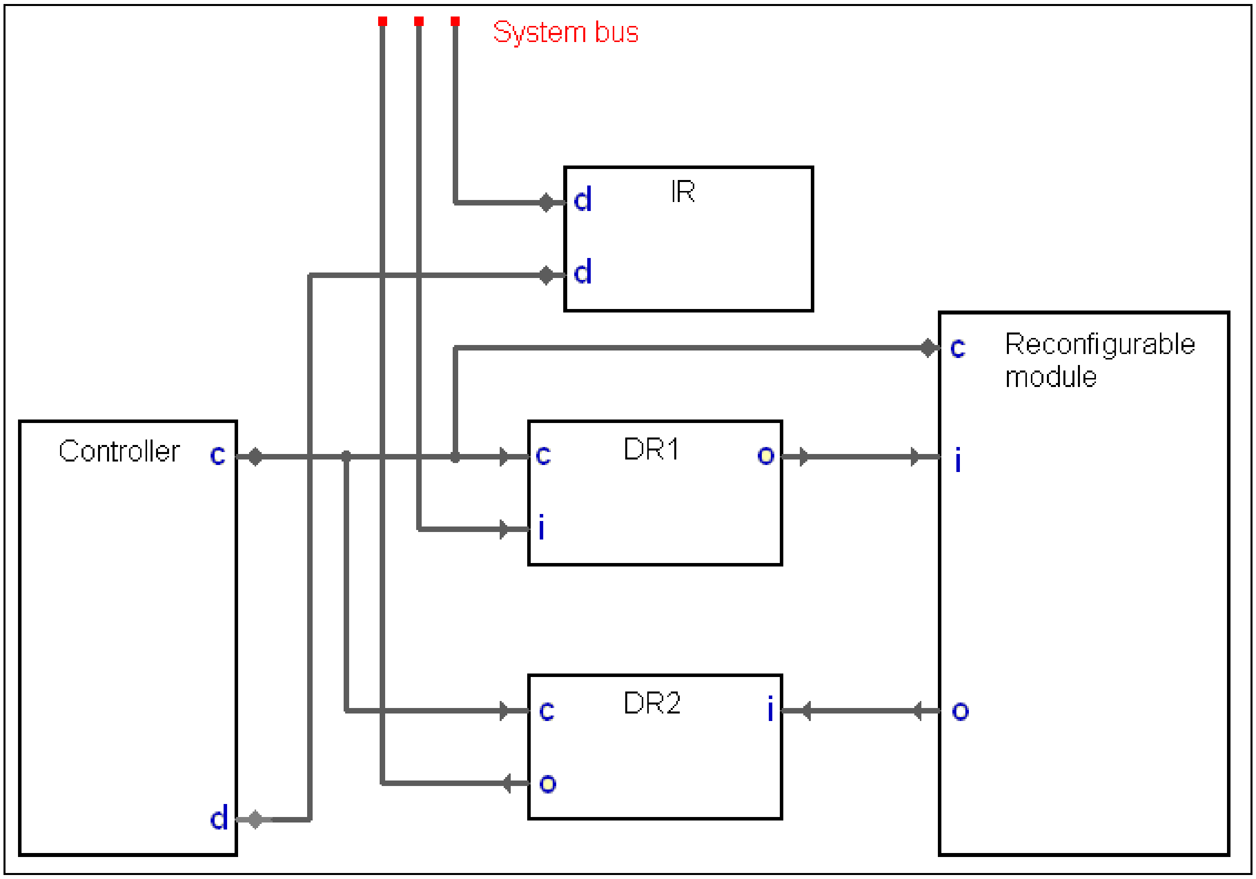


Figure 5.2.2: Software register backend

The second backend attempts to exploit the principle of locality by introducing a data cache in the backend. Distributed RAM blocks called BRAM are available on Xilinx devices for such purposes. Writing instructions to IR will in turn result in flow of data between a BRAM cell and the reconfigurable module. As suggested in the figure the BRAM cell is directly accessible from CPU.

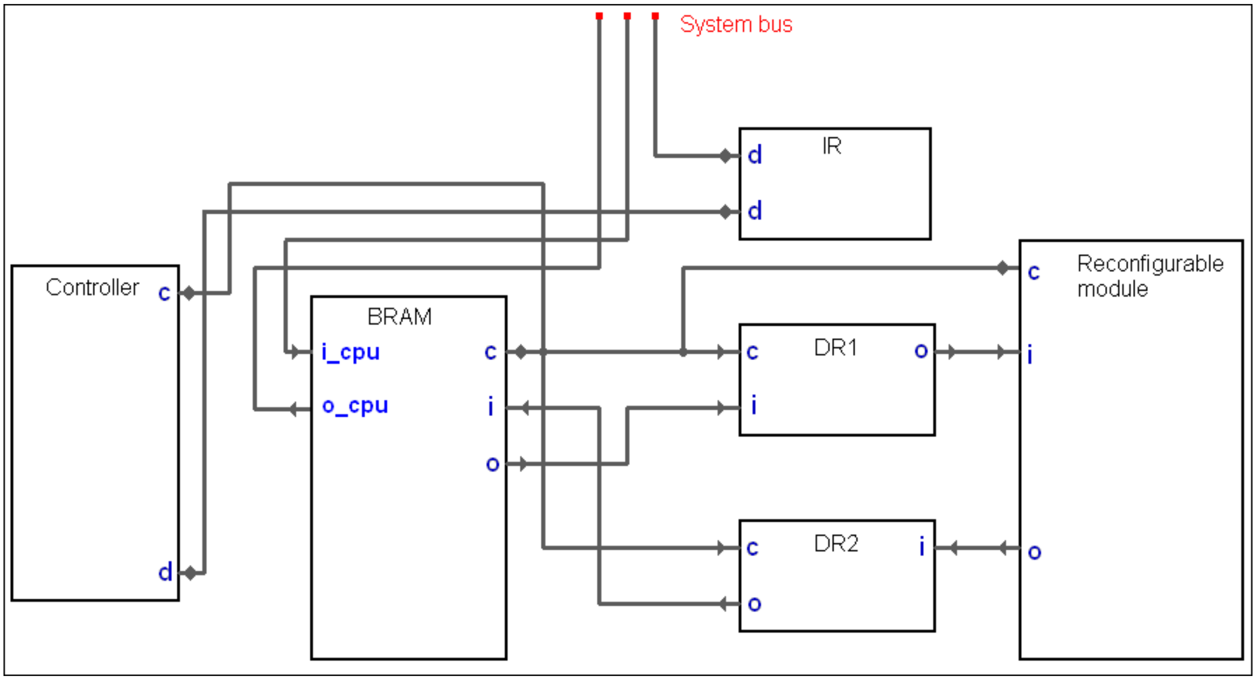


Figure 5.2.3: BRAM data-cache backend

The instruction and data cache backend, illustrated in figure 5.2.4, solely uses the BRAM as buffer mechanism between CPU and the backend. Hence both instructions and data are written to BRAM whereas a controller is in charge of fetching and executing instructions that in turn will move data between the reconfigurable module and the BRAM cell. A configuration register was added to the instruction and data cache backend at a later stage as described in section 5.2.6 with the purpose of providing run time tuning options for this backend type.

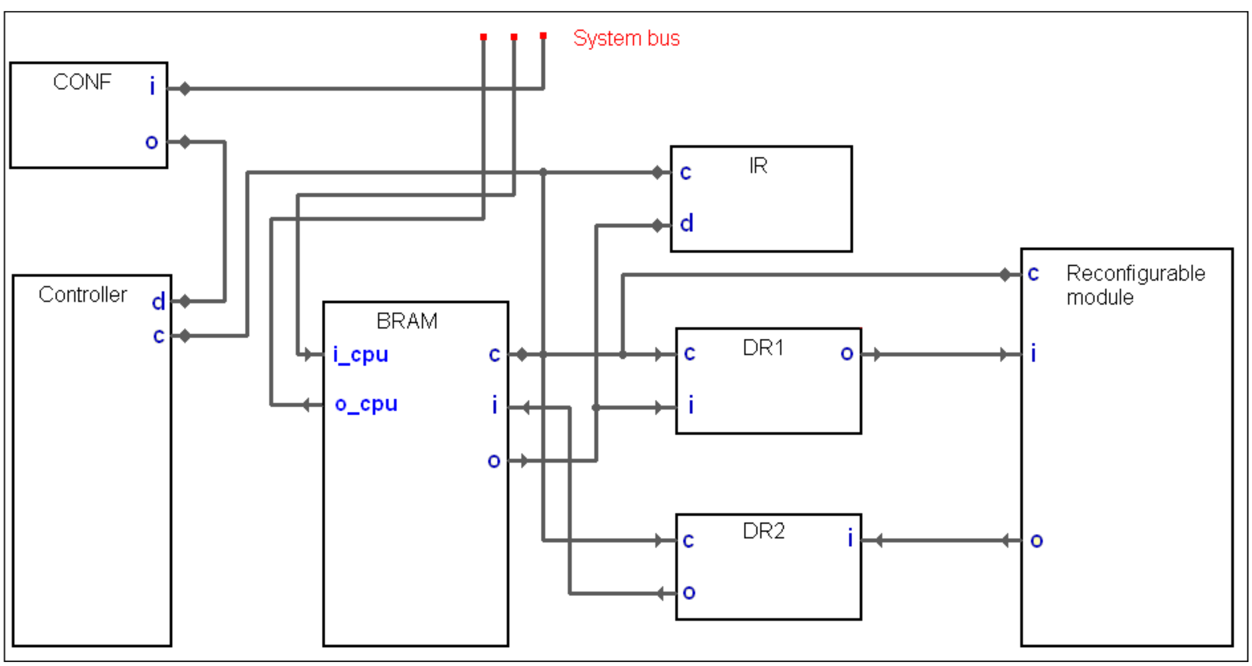


Figure 5.2.4: Shared instruction and data cache backend

5.2.1: Software register backend

The software register backend was developed first as the simplest solution. As part of the structural approach a shift controller was designed. Because of the principle of shifting data in and out of a reconfigurable module, shift primitives could be reused in the other backend solutions as well.

A common decision for all backend controllers was that their instruction sets should be simple. Although the backend solutions presented above are not general computers architectures, some of the reasoning for the simple instruction decision was taken based on RISC versus CISC properties. Providing an adequate instructions basis would allow more complex instructions to be decomposed into sets of simpler instructions. Instruction set extension is something that can be applied later on if analyses suggest so. With the respect to the constraint of a bit-width of 32 for DR1, DR2 and IR, the instruction format in table 5.2.1.1 was defined for the software backend controller.

Field	Opcode	Sync flag	Interrupt flag	Zerofill flag	Chain Address	Shift Count
Bits	7	6	5	4	3:2	1:0

Table 5.2.1.1: Software backend instruction format

Operation	Opcode
Shift	0
Execute	1

Table 5.2.1.2: Software backend opcode table

The backend controller was specified to do two operations, shift data and signal execution to the reconfigurable module. As a program running on CPU would supervise backend operations the *sync flag* bit was allocated for signaling completion of an instruction. The *Interrupt flag* bit allows interrupting execution of a reconfigurable module from processor. The *Zerofill* field was included as used in section 5.1.4. *Chain address* was allocated for selecting among the reconfigurable modules scan chains and *shift count* was allocated for specifying the number of shifts to perform.

The controller was developed with respect to the addressable shift interface from section 5.1.2 and the previous instruction and opcode definitions. After design completion, simulations were performed where the output of DR1 and input of DR2 were shortened to provide a simple test case emphasizing verification of the backend components only. After some minor corrections and satisfying simulations, work on a BRAM implementation was started.

5.2.2: Single port BRAM implementation

Developing the two remaining backend solutions required implementing a BRAM component as distributed memory. By creating a new peripheral in Platform Studio and specifying *user memory space* as interface, a default BRAM template with PLB bus connection was created.

It was decided to investigate the BRAM software interface at an early stage to check for dependencies or problems. Searching the web pointed towards a basic C-API consisting of the two functions in source 5.2.2.1. Closer investigation revealed that these functions were part of a Xilinx SDK C-library.

```
/*Read data*/
int XIo_In32(void * addr);
/*Write*/
int XIo_Out32(void * addr, int data);
```

Source 5.2.2.1

In [6], program development for PowerPC processor on the Suzaku has only used the Linux kernel API in source 2.3.1 to access hardware. By inspecting some of the dependencies of source 5.2.2.1 it became apparent that a decent amount of work was required to isolate the API functionality for use outside the SDK. It was instead decided to see if 2.3.1 would work. Applying the Linux kernel API for BRAM operations as described in section 6.1 worked flawlessly and it was decided to disregard the SDK API.

As the BRAM template created by Platform Studio described a straight forward single port RAM there were needs for modifications. Development of backend solutions that utilize distributed memory required BRAM components that could be accessed from both the PLB bus and local hardware. In an attempt to solve this problem the single port BRAM component was left untouched and a multiplex system was created to provide mutual exclusive BRAM access for two separate sources through a shared single port set. The BRAM template was first isolated and the behavior was deduced by applying stimuli through a test bench. After simulations and adding of multiplex logic the design was exported to the Suzaku board for verification.

Surprisingly target testing of this system could not be performed as the Suzaku board malfunctioned after bitfile upload. More thorough simulations were conducted but did not reveal any obvious reasons for the malfunction. Attempts were made to remove parts of the design and then uploading it to the Suzaku board. After a while a hypothesis was formed stating that the custom multiplex logic created illegal values for the PLB bus interface causing a system lock and preventing boot. As simulations could not reveal any further details of the problem it was decided to implement a true dual port BRAM.

5.2.3: Dual port BRAM implementation

After finding a couple of good references on the topic, [10] and [11], the EDK BRAM template was modified to a dual port implementation. Creating a dual port BRAM cell for a Xilinx device is based on a shared variable in VHDL, however due to the nature of the original EDK template quite a lot of modifications were necessary. The main problems were related to structure and timing.

In the original EDK BRAM template generate statements were used to create four parallel BRAM cells with 8 bit data input and output together forming 32 bit word, byte accessible RAM. When using a shared variable within the generate statement resulted in the error message in trace 5.2.3.1 under synthesis in Xilinx ISE

```
ERROR:Xst:2070 - If you are attempting to describe a dual-port block RAM with two separate write ports
for signal <ram1>, please use a shared variable
```

Trace 5.2.3.1

Another attempt was made by creating a global array of shared variables, on shared variable for each iteration in the generate statement. This resulted in the error message in trace 5.2.3.2 after synthesis. The problem was pursued by searching ISE help libraries and using the Internet. As no immediate solution for this problem was found and it was decided to omit the byte addressable solution and go for a simpler word addressable solution.

```
ERROR:Xst - You are apparently trying to describe a RAM with several write ports for signal
<Mram_ramy<3>>. This RAM cannot be implemented using distributed resources.
```

Trace 5.2.3.2

After further work on the EDK BRAM template one noticed that there were some signal inconsistencies compared to [10]. In the original EDK template the address signal for read and write were split into two separate signals where the read address was buffered through DFFs. [10] clearly use one common address source for each port however. After making the corresponding changes to the VHDL code the error message in 5.2.3.2 disappeared which suggested that the use of multiple address signals was the root of the synthesis problems. Source 5.2.3.1 shows such an invalid statement for one of two ports of the BRAM where separate address signals are used depending on whether a read or write is performed.

```
If clk'event and clk='1' then
        if ( write_enable = '1' ) then
                ram(CONV_INTEGER(read_address)) :=  Bus2IP_Data;
        end if;
        mem_data_out(0) <= ram(CONV_INTEGER(mem_address));
end if;
```

Source 5.2.3.1

Instead of creating multiplexed addressing with the original timing, the read address buffering was removed. In the case of the port that was to be connected to custom made hardware this would not cause any problem as changes could be made to fit timing requirements. Signal timing from the PLB bus is however dependent on other system components. Modifying any external system components was of course out of the question, but under the assumption that the read address signal was persistent on the PLB buss during a read operation this was regarded as an opportunity for a nice simplification.

The implementation was packed as a VHDL entity for reuse. Finally a Planahead project was created to see how the BRAM component mapped physically to FPGA. Figure 5.2.3.1 shows the placement of the BRAM cell in the top right corner of a Virtex-IV and figure 5.3.2.2 shows the corresponding resource estimates.

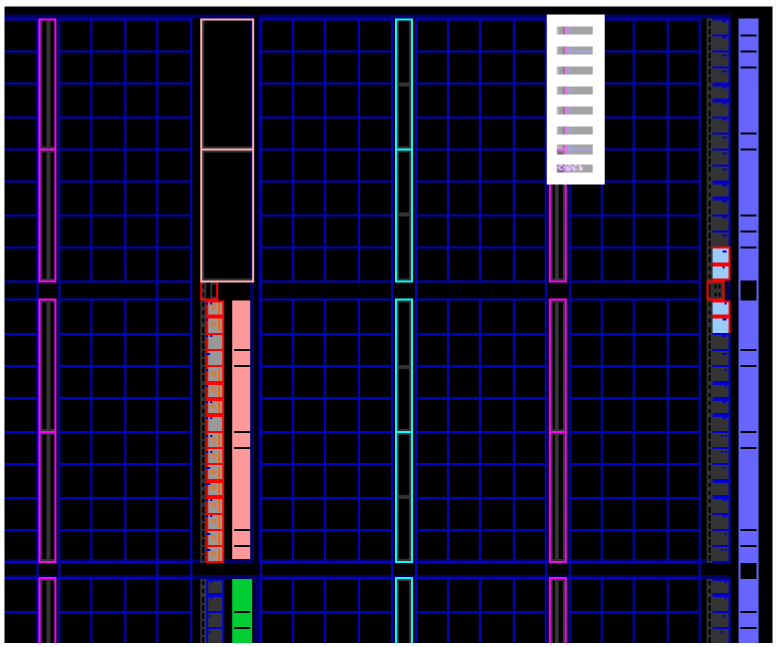


Figure 5.2.3.1: Device layout

Physical Resources Estimates

Site Type	Available	Required	% Util
LUT	40	4	10.00
FF	40	3	7.50
SLICEL	10	2	20.00
SLICEM	10	2	20.00
FIFO16	1	0	0.00
RAMB16	1	1	100.00

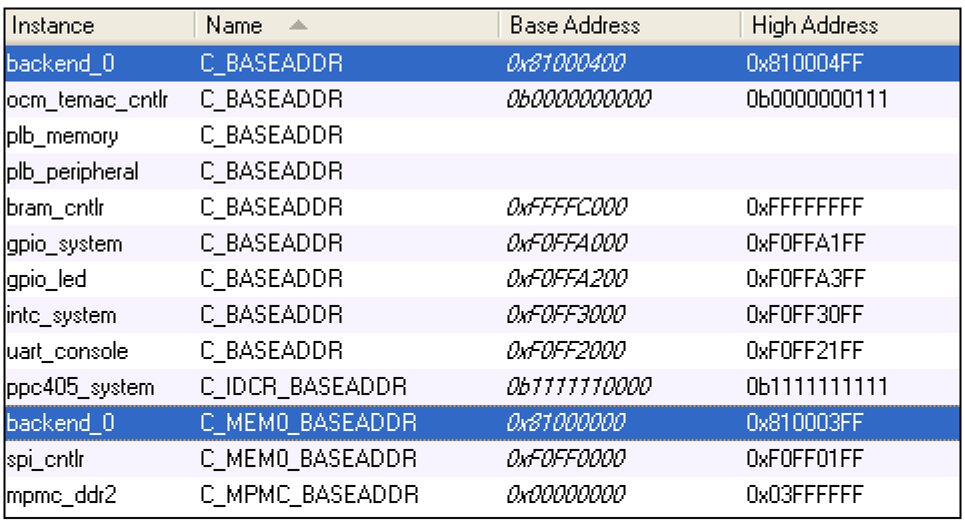
Figure 5.2.3.2: Resource estimates

5.2.4: Combining BRAM and software registers

Another technical dependency to consider was the data cache backend's requirement of combined usage of a BRAM cell and software accessible registers.

To investigate this, a backend module was created using both the BRAM implementation from 5.2.3 and two software registers. The question to be answered was how to distinguish addressing of the three components. Another simplistic test case was created involving software accessible registers in BRAM read and write operations. After verifying the system through simulation it was implemented as a peripheral in Platform Studio project. The backend was assigned a 2K address space as the BRAM consumed 1K and two SWREGS consumed 2 addresses.

When exporting the design to the Suzaku board it locked up again. Different approaches to finding the error were taken. It turned out that by removing either the software registers or the BRAM the system did not lock up. By closer inspection of the system assembly window in Platform Studio it became apparent that two separate address entries for BRAM and software registers were created in the global address map. Both entries were filled out this time, as shown in figure 5.2.4.1.



Instance	Name	Base Address	High Address
backend_0	C_BASEADDR	0x81000400	0x810004FF
ocm_temac_cntlr	C_BASEADDR	0b0000000000	0b0000000111
plb_memory	C_BASEADDR		
plb_peripheral	C_BASEADDR		
bram_cntlr	C_BASEADDR	0xFFFFC000	0xFFFFFFFF
gpio_system	C_BASEADDR	0xF0FFA000	0xF0FFA1FF
gpio_led	C_BASEADDR	0xF0FFA200	0xF0FFA3FF
intc_system	C_BASEADDR	0xF0FF3000	0xF0FF30FF
uart_console	C_BASEADDR	0xF0FF2000	0xF0FF21FF
ppc405_system	C_IDCR_BASEADDR	0b1111110000	0b1111111111
backend_0	C_MEM0_BASEADDR	0x81000000	0x810003FF
spi_cntlr	C_MEM0_BASEADDR	0xF0FF0000	0xF0FF01FF
mpmc_ddr2	C_MPMC_BASEADDR	0x00000000	0x03FFFFFF

Figure 5.2.4.1: System assembly address window

Exporting this solution to the Suzaku board worked fine. The system did not lock up and the case testing, described in section 6.1 worked as anticipated.

5.2.5: Data cache backend

With all necessary components available the data cache backend was developed. Prior to coding the instruction format in table 5.2.1.1 was defined. As the BRAM implementation in section 5.2.3 was a 1kB word addressable memory an operand of 11 bits was allocated, covering address space and including 3 excess bits.

Field	Opcode	Sync flag	Interrupt flag	Operand
Bits	15:14	13	12	11:0

Table 5.2.5.1: Data cache instruction

Instruction	Opcode
Read	00
Write	01
Shift	10
Execute	11

Table 5.2.5.2: Data cache opcode

Table 5.2.5.2 shows the different operations of the controller. The controller will allow data to be scanned in and out and execution of a reconfigurable module as in section 5.2.1. In addition the *read* and *write* operations will allow transferring data between DR1 and DR2, and the BRAM. *Read* will read data from BRAM into DR1 while *write* will write data from DR2 into BRAM where operands will control addressing.

The backend solution was developed according to the previous definitions and the addressable shift interface from section 5.1.2. For simulations the output of DR1 and the input of DR2 were shortened as before. After approved simulations results, focus was turned towards the last backend implementation.

5.2.6: Instruction and data cache backend

As the instruction and data cache backend has many similarities with the data cache backend, the instruction format in 5.2.5.1 and the opcode legend in 5.2.5.2 were reused. It was decided to use a single common BRAM cell as shared instruction and data cache as opposed to separate BRAM cells. Although separate caches could have higher performance based for a given total size, a shared implementation was considered more compact and simplistic. The major difference between the data cache and the shared instruction and data cache solution was that instructions now had to be fetched from BRAM for execution. An instruction fetcher entity was created for this purpose whereas many parts from the data cache backend could be reused. Simulations were conducted with the earlier mentioned shortening of DR1 output and DR2 input.

5.2.7: Synthesis

To confirm the correctness of the different backend bridges, verification was performed on the Suzaku platform through Linux kernel modules as described in section 6.1. In order to do comparison of the different backend modules’ resource requirements, synthesis was performed in Xilinx ISE. All synthesis was performed without any reconfigurable modules connected in the backend to focus on controller complexity. Synthesis results are found in section 10.2.

5.3: Device driver encapsulation

During initial backend validation described in section 6.2, hardware access had been performed solely through kernel modules performing hardware access operations on entry and removal through *insmod* and *rmmod* calls. Although this works for simple verification it is by no means an acceptable run time solution.

In order to provide user space backend access, Linux character device driver were created for the different backends, using [12] and [13] as reference literature. Both the shared instruction and data cache backend and the data cache backend use at least one BRAM and at least one software accessible register. As Platform Studio splits the address spaces for the two component types into two disjoint subspaces, the device drivers for the both these backends were implemented with two separate address pointers thus removing the requirement of a contiguous physical address space for software registers and BRAM. As discovered in section 6.3 the PowerPC uses an endian format. Run time translation between the endian format and a linear format was implemented as part of the device driver's run time functionality.

The key character device driver functions *read()* and *write()* specify a char buffer as function parameter which is used to transfer data between user space and kernel space. Due to backend solutions composition the main interest was transferring integers however. The two functions in source 5.3.1 were created for the purpose of easy translation between character arrays and integers.

```
/*Write int to charbuffer at given position*/
static void intToChar(char* buf, int value, int pos){
        int i,mask;
        mask = 0x000000FF;
        for(i=0; i< 4; i++){
                buf[pos+i] = (value & mask)>> 8*i;
                if(i==3){
                         buf[pos+i] = buf[i+pos] & 0x000000FF;
                        return;
                }
                mask = mask <<8;
        }
}
/*Read int from char buffer from given position*/
static int charToInt(char *buf, int pos){
        int i,mask,res;
        res =0;
        mask = 0x000000FF;
        for(i=0; i< 4; i++){
                res = res+ ( (buf[pos+i]<< 8*i) & mask);
                mask = mask << 8;
        }
        return res;}
```

Source 5.3.1

Verification of the character device drivers was done according to section 6.3 using the same hardware versions as in section 6.2. With different backend solutions developed along with corresponding character device drivers, performance testing was conducted as described in section 7.

5.4: Software components

5.4.1: Dynamic memory allocation

Based on results of performance testing of the developed backends as described in section 7, it was decided to continue work using the instruction and data cache backend. One obvious problem was that some kind of memory allocation mechanism was needed to allow multiple processes to share a BRAM cache. A quick literature search was done and two different approaches for dynamic memory allocation, *memory pools* and *buddy memory allocation*, were found at [14] and [15] respectively. With the present conditions a BRAM cache is likely to be of modest size and the *memory pool* approach was selected because of its simplicity and applicability for small memories. It was decided that memory allocation should only apply to the data part of the id-cache as the instruction segment probably might benefit from special treatment.

An executable C prototype of the *memory pool* algorithm was first developed and tested on the dev-computer. The skeleton is a linked list of descriptors, showed in source 5.4.1.1, which as a whole describes a set of memory pools. In the descriptor the id field describes a unique identifier for a process owning the pool and the size field describes the size of the pool. Although the *memory pool* approach originally specifies a uniform pool size, the size field was added for future flexibility.

```
struct pool_descriptor{
        struct pool_descriptor *next;
        struct pool_descriptor *prev;
        int id;
        int size;
}
```

Source 5.4.1.1

Two API functions *mem_allocate()* and *mem_free()* were written and included in the allocator's header file constituting the assignment interface. The *mem_allocate()* function was designed to find the first continuous space of memory in fractions of pools, large enough to for the requested allocation size. Although there are no hardware requirements for continuous memory it was considered to produce less overhead in descriptors and less overhead in HWOS read and write operations. A mutex from the <*pthread.h*> library was used to ensure mutual exclusion of *mem_allocate()* and *mem_free()*. After some testing on the dev-computer, the C source was copied and a version was cross-compiled for the Suzaku board. Building feedback showed that no there were no problems with libraries or primitives.

5.4.2: HWOS – the application interface

After completing the design of the memory allocator the application-to-hardware interface was now considered. Based on the HWOS concept as described in [3] it was decided to create a master process as a server for software applications requiring access to hardware. As much of this work does not directly involve too many of the originally proposed HWOS components, it was prioritized to create a template which focused on the application interface part, implementing functionality developed in this thesis. Looking into the technical details of the implementation it was found that it is not possible to run ordinary programs in the background on the Suzaku board. Furthermore in a Linux based system, a background process should be spawned as a DAEMON. [16] provided a DAEMON skeleton in C. Figure 5.4.2.1 shows the target structure.

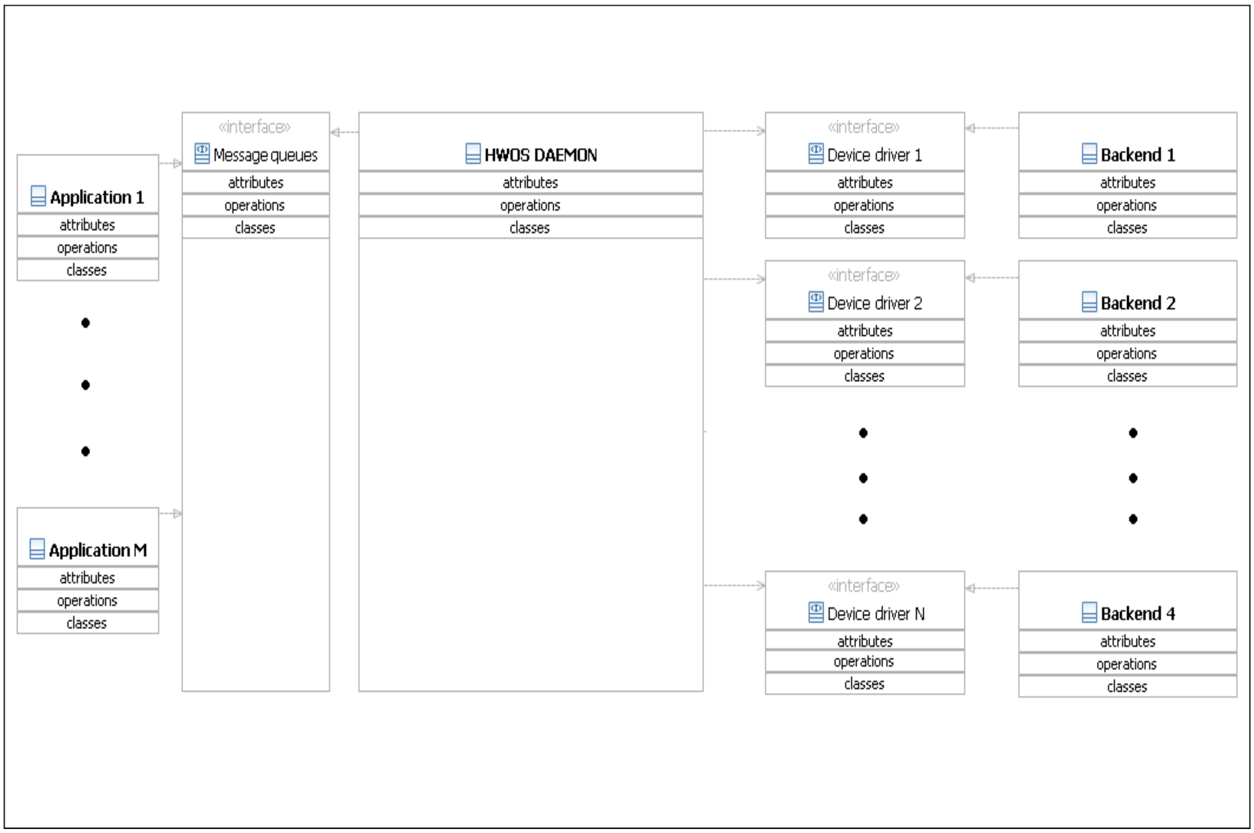


Figure 5.4.2.1 Application interface UML

Extending the DEAMON skeleton was simply done by collecting the existing sources and making some modifications to the infinite executive loop in the DEAMON. To provide communication between the HWOS and applications a message queues from *<sys/msg.h>* was used. One master queue was created in the HWOS for requests from applications. Instead of having a shared queue for all client applications it was decided to create separate receive queues for each application. Limitations and consequences of this choice were not investigated further, but separation of message queues does indeed provide cleaner code. In order to manage separate queues for client applications, a registration procedure was implemented in the HWOS. This is now discussed as part of the request and response messages.

A common request message type was composed as shown in source 5.4.2.1.

```c
typedef struct req_msg{
        long    type;
        char    command;
        int     addr;
        int     bytes;
        char    data[BUF_SIZE];
        int     sender_id;
} req_msg_t;
```

Source 5.4.2.1

The *type* field is required by the message queue primitives. All request messages was decided to have the same type for now however, as different message types seems to require separate polling calls. A command field was included in the message type for signaling what service was requested. The *address* and *byte* field were added considering hardware read and write operations. A compile time parameterized buffer was added for data transfer. It should be noted that pointers were not and cannot be used, generally, in message types. Pointers which are passed from one process to another become invalid because of separate memory spaces. Last the *sender_id* holds a key that is to be used for creating a process' receive message queue. This identifier has to be unique for the idea to work which is not handled properly in the current version.

```c
typedef struct resp_msg{
        long    type;
        int     retval;
        char    data[BUF_SIZE];
} resp_msg_t;
```

Source 5.4.2.2

Source 5.4.2.2 shows the response message. It is pretty straight forward containing two separate fields for return value and return data.

Verification was performed for the HWOS as described in section 6.4 along with performance evaluation as described in section 7.

6: Verification

In addition to simulations developed components were deployed and verified on the Suzaku board. Most verification was done simultaneous with development as a precaution to ease further verification of the composite system. The verification process as a whole is now presented.

6.1 Backend

In order to do simplistic verification of the backend solutions developed in section 5.2, the approach of shortening DR1 and DR2 was used as in simulations. Already when starting testing of the software register backend problems occurred. None of the values obtained through kernel module testing made any sense. By creating a write-from-hardware read-from-software test it was discovered that bits appeared to be shuffled seen from the software side. Some searching on the internet revealed that PowerPC uses an endian byte system, which explained the feedback. Figure 6.1.1 illustrates the PowerPC format versus a linear word format. MSB is zero indexed, "()" encloses hex values and "{}" encloses bytes.

{(24,25,26,27),(28,29,30,31)}{(16,17,18,19),(20,21,22,23)} {(8,9,10,11),(12,13,14,15)} {(0,1,2,3),(4,5,6,7)}

PowerPC endian format

{(0,1,2,3),(4,5,6,7)}{(8,9,10,11),(12,13,14,15)} {(16,17,18,19),(20,21,22,23)} {(24,25,26,27),(28,29,30,31)}

Linear format

Figure: 6.1.1: Endian versus linear bit string

In order to work with PowerPC's endian format the algorithm in source 6.1.1 was created. It translates the PowerPC endian format into the linear format and vice versa. The function is thus its own inverse.

```
int transform(int word){
        int mask,part,res,i,n;
        int shifts[] = {3,1,-1,-3};
        mask =0x000000FF;
        res =0;
        for(i=0; i<4; i++){
                if(shifts[i]>0)     part = (word &mask ) << (shifts[i]*byte);
                else part = (word &mask ) >> (-shifts[i]*byte);
                if(i==3) part = part &     0x000000FF;
                res = res | part;
                mask = (mask <<           byte);
        }
        return res;
}
```

Source 6.1.1

As a backend dependency the BRAM software API was researched by creating a Platform Studio project with a peripheral implementing a 1kB byte accessible single port BRAM component. The peripheral's base address was set to 0x81000000 and the project was synthesized, placed and routed and eventually exported to bitfile. The resulting bitfile was loaded onto the Suzaku through hermit followed by a reboot to update the FPGA configuration. A Linux kernel module was developed, reading and writing hard values to the peripheral and performing debug printing along the way. The test worked as planned and it was concluded that there was no need for the SDK API to access BRAM as it could be achieved through the Linux kernel API with respect to the global memory map in Platform Studio.

To verify the dual port BRAM component developed in section 5.2.3 a simplistic copy test was created. A hardware module was placed in the backend which behavior was to copy the contents of BRAM[0] to BRAM[1]. A kernel module was developed that wrote data to BRAM[0] and subsequently the data at BRAM[1]. Execution verified that the dual port component worked as intended providing access to a single BRAM component for both PLB bus and local hardware. Further backend testing proved the correctness of the dual port BRAM, also in combination with software registers.

After verifying the functionality of the dual port BRAM testing was performed for all backends through Linux kernel modules. Based on the different controller instruction sets, small assembly programs were created to verify functionality. An example program for the instruction and data cache backend is shown in table 6.1.1 along with the resulting operations.

Instruction-word	**Instruction 1**	**Instruction 0**
0x8003000f	Shift 4x	reg0 <= BRAM[f]
0x000b4009	reg0 <= BRAM[b]	BRAM[9] <= reg1
0x400a8003	BRAM[a] <= reg1	Shift 4x
0x8003000c	Shift 4x	reg0 <=BRAM[c]
0xffffffff	HALT	HALT
0x000d400e	reg0 <= BRAM[d]	BRAM[e] <= reg1

Table 6.1.1: Example micro-program

6.2 Sequential multiplier

After verifying the different backends with shortening of DR1 and DR2, the sequential multiplier from section 5.1.4 was inserted as reconfigurable module. Three separate Linux kernel modules, one for each backend type, were developed to test the multiplier in combination with the backend. Multiply operations both from scratch and from an interrupted state were tested. A trace from testing the sequential multiplier combined with the instruction and data cache backend is shown in section 10.2.

6.3 Device drivers

For testing the device driver implementations the sequential multiplier was used as reconfigurable module. Device files were created in the */var* directory on the Suzaku's filesystem. Normally device files are placed in the /dev folder, but by default this path is read only on the Suzaku's file system. Instead of going to any extra efforts to change this, the writable folder */var* was used. Trace 6.3.1.shows the creation of a device file for the instruction and data cache backend followed by insertion of a kernel module describing the char device driver through a NFS shared mapped to */var/tmp*.

```
pwd
/var
mknod  id_backend c 128 0
cd tmp/device_drivers/id_backend
insmod id_backend.ko
```

Trace 6.3.1

An initial error was found when attempting to reinsert a device driver that had already been removed. Appending the unregister call in source 6.3.1 with the correct parameters in the device driver's unload function solved the problem.

```
int unregister_chdev(int major_number,const char []  device_name);
```

Source 6.3.1

Testing of the functionality of each device driver was then performed by writing different C-applications for the developed backend variants, which accessed hardware through corresponding device drivers. Source 5.3.2 shows the syntax accessing a device driver through a related device file.

```
#define BACKEND_PATH "/var/id_backend"
…
Fopen(BACKEND_PATH);
```

Source 5.3.2

The first debugging showed a mix of correctness and failureAs the device drivers were operating on hardware identical to the one used under earlier kernel module testing it was deduced that the errors were probably located in the device drivers themselves. y doing a mixed execution of the previously developed kernel modules and the new device drivers a strange behavior was found. The read and write functions in a char device driver specify a pointer *f_pos* which holds a value that is used to control read and write offset with respect to a base address. Debug prints of the value showed that it always had the anticipated value but it turned out that using byte offsets in the read and write functions resulted in reading and writing only every fourth line. This is in contrast to kernel module testing where byte addressing was required as opposed to word addressing. After modifying offset fields in device driver sources according to source 6.3.3 all functionality turned out as expected.

```
readl(base_pointer + (offset)/4)
writel(base_pointer + (offset)/4)
```

Source 6.3.3

6.4 HWOS

Initially a dev-pc version of the HWOS DAEMON was developed to ensure that the framework was working correctly. After testing on the dev-pc was considered sufficient the HWOS DAEMON was built for the Suzaku board and tested. Trace 6.4.1 shows the execution trace of the HWOS on the Suzaku platform.

```
# ./hwos
# ps | grep hwos
  172 root       256 S   ./hwos_d    //HWOS process
# kill 172
# ps | grep hwos
#
```

Trace 6.4.1

After execution the daemon can be found in the process list and killing the DAEMON removes it correctly. It was first confirmed that the message queue interface worked as expected on the Suzaku platform. The instruction and data cache device driver was then integrated with the HWOS as no real hardware access was performed in the dev-pc version. Only the instruction and data cache backend was integrated with the HWOS, acknowledging synthesis and speed results from section 10.2 and 10.3. An execution trace of an external application accessing hardware through the HWOS is shown in section 10.4.

7: Performance evaluation

Instrumentation:

To compare performance of hardware access operations it was decided to count clock cycles through a software register in the backend rather than using processor timer libraries. Using a hardware defined counter would probably provide more accurate timing. One extra software register was first appended to each backend. As each backend already contained at least one software register with some free bits, fields in existing register were used for control signals to start, stop and reset the timer. After simulation and some minor corrections were applied and the new hardware was exported to bitfiles prepared for testing on the Suzaku board.

Testing:

Testing was divided into three different subtests, one directly from kernel space and two from user space. The user space testing comprised hardware access directly from applications through device drivers and hardware access through the HWOS. Performance of the cache backend modules was expected to vary depending on whether a cache hit or cache miss occurred. To narrow down the testing a full cache hit and a full cache miss were tested as counterparts for both the instruction and data cache backend and the data cache backend.

As the test results were expected to have high correlation with the testing software, some measures were taken to ensure that the different backend tests ran under fair. First all excess code between timer start and timer stop were removed. It was also ensured that reading and writing was performed following the same pattern. The initial test scenario was the transfer of the multiplier state in table 5.1.4.1. Afterwards the correlation between performance and amount of data in a transfer was explored by running separate tests where the amount of data transferred was varied in multiples of 20 bytes. Multiples of 1,2,3 and 4 were used, distinguishing user space versus kernel space and cache hit versus cache miss as before. Results are found in section 10.3.

8 Case: MPEG transcoder

As part of this thesis use of components from a MPEG transcoder as reconfigurable modules was researched. Creating a truly reconfigurable implementation for the framework for dynamic reconfiguration in [3] might be quite time consuming so it was prioritized to relate such an implementation to the work in this report.

8.1 Source analysis

VHDL source for a MPEG transcoder was obtained from [17]. As only the project report was available at first hand a small Java program was written to parse the VHDL source code as it contained line numbers, page numbers and invalid characters. After parsing the source to text files and compiling them in AHDL trace 8.1.1 appeared.

```
Library "XilinxCoreLib" not found.
```

Trace 8.1.1

Searching the file hierarchy of the installed Xilinx tools revealed that all sources for this library could be found in the path given in trace 8.1.2. Considering platform independency the Xilinx path is emphasized in bold casing.

```
F:\Xilinx\ISE\vhdl\src
```

Trace 8.1.2

The sources were copied to a safe location, compiled as a library in AHDL and included in the project providing a clean compile with exception of a few warnings. As an MPEG transcoder is a complex entity which requires a large number of resources the approach was to consider a piped execution where only parts of the complete entity is executing in hardware at any given time. Evaluating the composition of the different parts of the MPEG transcoder it was found that the decoder contains much explicit memory which also might make it suitable for investigating performance enhancing interrupt and resume functionality as presented in section 2.5. More specificly the Inverse-Scan module was chosen for research as because of appliance to these properties.

To clarify how all the Inverse-Scan module mapped to Virtex-IV's resources, synthesis was performed in ISE. Two separate syntheses was performed, one for a single register file and one for the entire inverse scan module. Resource consumption for a single register file is shown in trace 8.1.3.

```
Device utilization summary:
---------------------------
Selected Device : 4vfx12sf363-10

 Number of Slices:               636  out of   5472    11%
 Number of Slice Flip Flops:     768  out of  10944     7%
 Number of 4 input LUTs:         468  out of  10944     4%
 Number of IOs:                   39
 Number of bonded IOBs:           39  out of    240    16%
 Number of GCLKs:                  1  out of     32     3%
---------------------------
```

Trace 8.1.3

A screenshot from Planahead is showed figure 8.1.1, roughly illustrating the area requirements. The rectangle consisting of several brownish sub-rectangles is the register file. Adjacent BRAM cells, RAMB16, are highlighted in white.

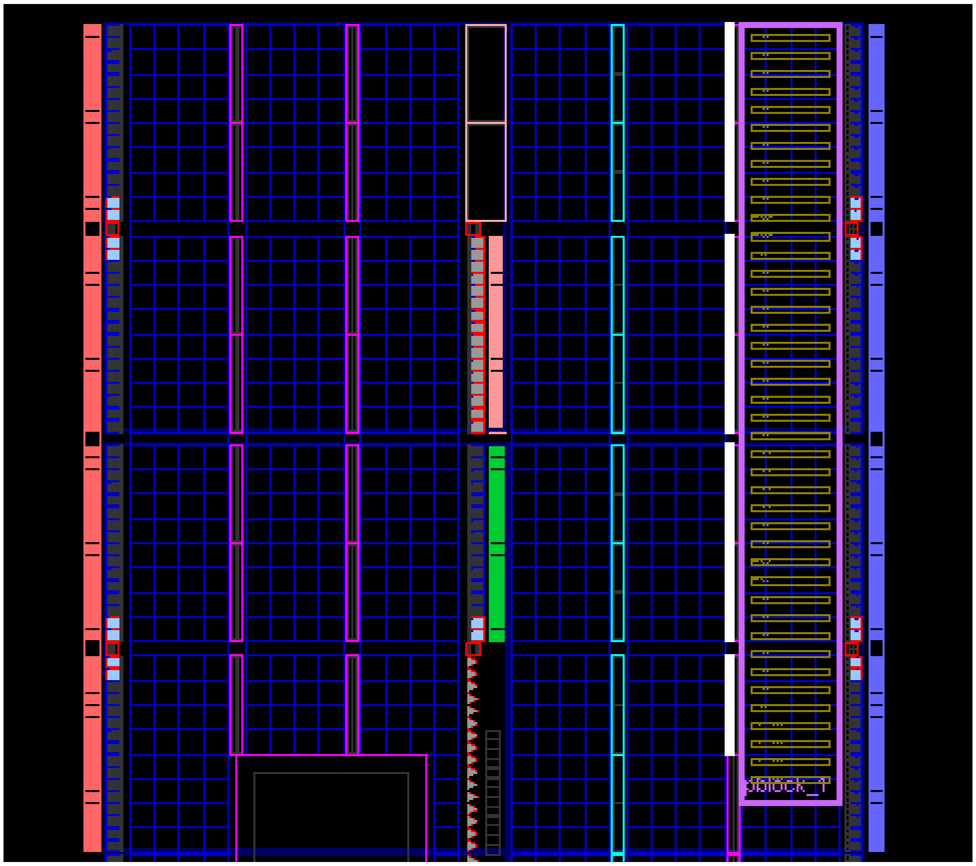


Figure 8.1.1: Registerfile mapped to FPGA

After finding an address space of 6 bits and a bit width of 12 for each field of the register file the storage capacity was calculated according to (10).

$$2^{A_W} \times D_W = 2^6 \times 12 = 768\ bits \tag{10}$$

The ISE synthesis results for the entire inverse scan module, consisting of four register files, is shown in trace 8.1.4. In a comparison, a single RAMB16 component can store 10.66 register.

Device utilization summary:

Selected Device : 4vfx12sf363-10

Number of Slices:	2734	out of	5472	49%
Number of Slice Flip Flops:	3155	out of	10944	28%
Number of 4 input LUTs:	5041	out of	10944	46%
Number of IOs:	36			
Number of bonded IOBs:	36	out of	240	15%
Number of GCLKs:	1	out of	32	3%

Trace 8.1.4

8.2 Specialized cells

Because of the extensive resource requirements of the register file component and the inverse scan module as a whole it was concluded that LUT buffering needs to be reduced before any attempts to use the inverse scan module in the reconfigurable system under the current conditions. Another interesting property of the MPEG decoder is that it requires a RAM cell. The problem of how a reconfigurable module should interact with specialized cells like DSP, FIFO and BRAM, has not been addressed in this paper yet.

One possibility for handling cell dependencies is to rely on the physical placement of a reconfigurable module to provide necessary cell components. As an alternative specialized cells can be migrated across the reconfigurable interface where a pair of controllers is in charge of performing operations. It was decided to go for the latter solution, illustrated in figure 8.2.1.

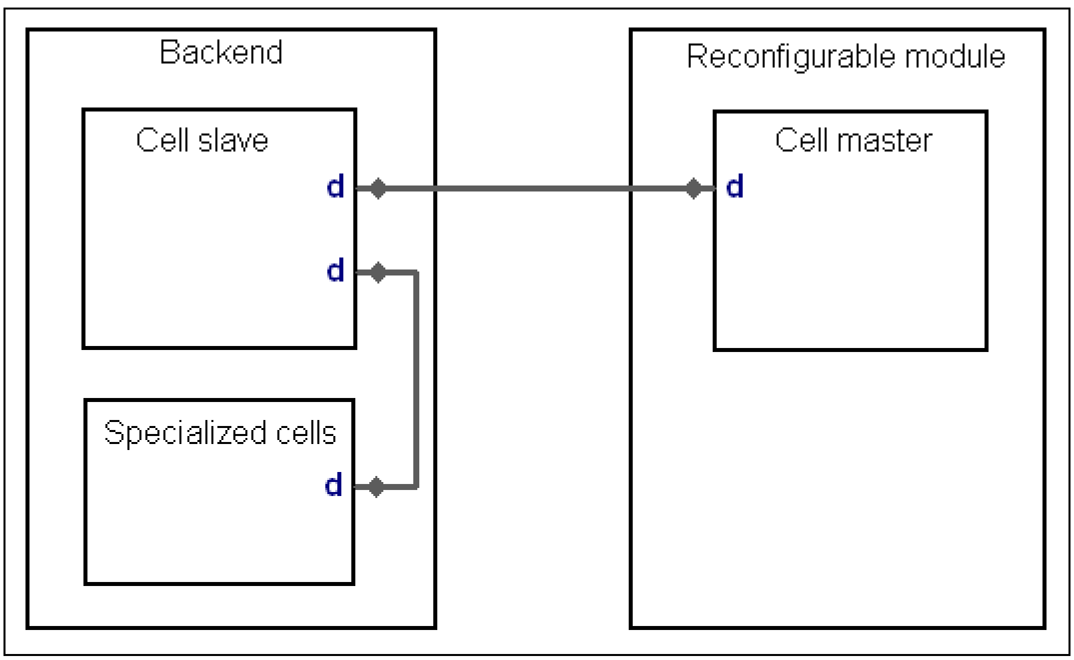


Figure 8.2.1: Special cell migration through controller pair

Advantages and disadvantages of this approach are mentioned later in discussion. As an example of cell migration it was decided to move the RAM component from the MPEG decoder across the reconfigurable interface.

8.3 MPEG decoder RAM migration

By investigating the MPEG decoder RAM implementation it was found that it uses a BRAM with a default Xilinx architecture. It indicated that reads and writes are performed from separate sources through signal names and separate address and clock signals. The RAM implementation is thus of dual port type. Simulation confirmed that it contained 256 words where 12 bits of each word were used in accordance with the register file bit-depth. Simulations also clarified control signal functionality.

An assumption was made before starting the design process that processing and traditional input/ output and state operations are mutually exclusive. Under this assumption the data channels in the reconfigurable interface is either used for transferring input, output and state, or used for communication with specialized cells. This might be a false assumption if a reconfigurable module buffers future input values and readout of results are performed during execution. These are classic speedup techniques as mentioned in [7], but the simplifications were made to ease the cell migration problem due to time limitations.

Instead of compressing control signals under this hypothesis, the main emphasize was to reuse the existing data channels in the reconfigurable interface for cell operations. Since the address space of the BRAM in question is 8 bits, it was decided to pipe addressing through data channels as well to keep the maintain a reconfigurable interface of a modest size. An operation control signal was added to the reconfigurable interface whereas clocks were assumed to be synchronized with respect to the backend master clock. The operation signal covered no-operation, read, write and reset instructions to address the migrated RAM.

A fresh workspace was created for designing and validating communication between a RAM-master and RAM-slave through the reconfigurable interface and the operations on the MPEG decoder RAM module. To exploit existing functionality a shift register macro was created in the master controller and using existing shift primitives for data transfer. The operation signal for cell operations thus served as request signals resulting in a data transfer using the global shift primitive and existing data channels. In order to perform read and write operations, the target address would be shifted in to the slave first followed by the operation and finally the shifting of data. Validation of the design was conducted through simulations but no deployment was performed due to time limitations.

9: Run time reconfiguration

With the components for hardware-software intercommunication designed, some light was shed on the actual process of run time reconfiguration at the end of the thesis work.

In earlier work, reconfigurable modules has been created by building an entire FPGA system for the Suzaku platform and then extracting one or more reconfigurable modules from the resulting bitfile. This is very inefficient as building the entire system is a hard processing task that may take 30-40 minutes or so. A much easier solution is to do creation of reconfigurable modules in Xilinx ISE as one can build only the reconfigurable module along with bus macro connections standalone.

As there was too little time to do proper MPEG transcoder research simple test modules were used for reconfiguration. First the entire Suzaku system was implemented with the sequential multiplier as a true reconfigurable module according to [18]. This now provided the system base and reconfigurable module design in Xilinx ISE was explored by creating a sequential comparator and transforming it into a true reconfigurable module.

As with the Platform Studio flow, Planahead was used to create placement constraints after synthesis in form of a UCF file which was added as a project source in ISE. Two important results was derived when inspecting the routed design. The clock signal refused to be piped through bus macros and IOBs at the top right of the FPGA was used for top entity’s input and output signals. By reference search on internet

[19] was found which states that piping of clock signals through bus macros is discouraged and depreciated by implementation tools. The general solution would be to ensure that the same clock branches are used in and out of a reconfigurable module. To address the IOB problem, the UCF code for banning use of IOBs showed in source 9.1.1 was applied to BANK6.

```
CONFIG PROHIBIT= [Location];
```

Source 9.1.1

Figure 9.1.1 and 9.1.2 shows screenshots from FPGA-editor of the resulting systems. The reconfigurable region is contained in a blue rectangle, bus macros are showed in red in the left top whereas the global clock is the red wire bellow distributed to slices in the multiplier. It can be seen that both bus macros and the global clock have the same location in both designs.

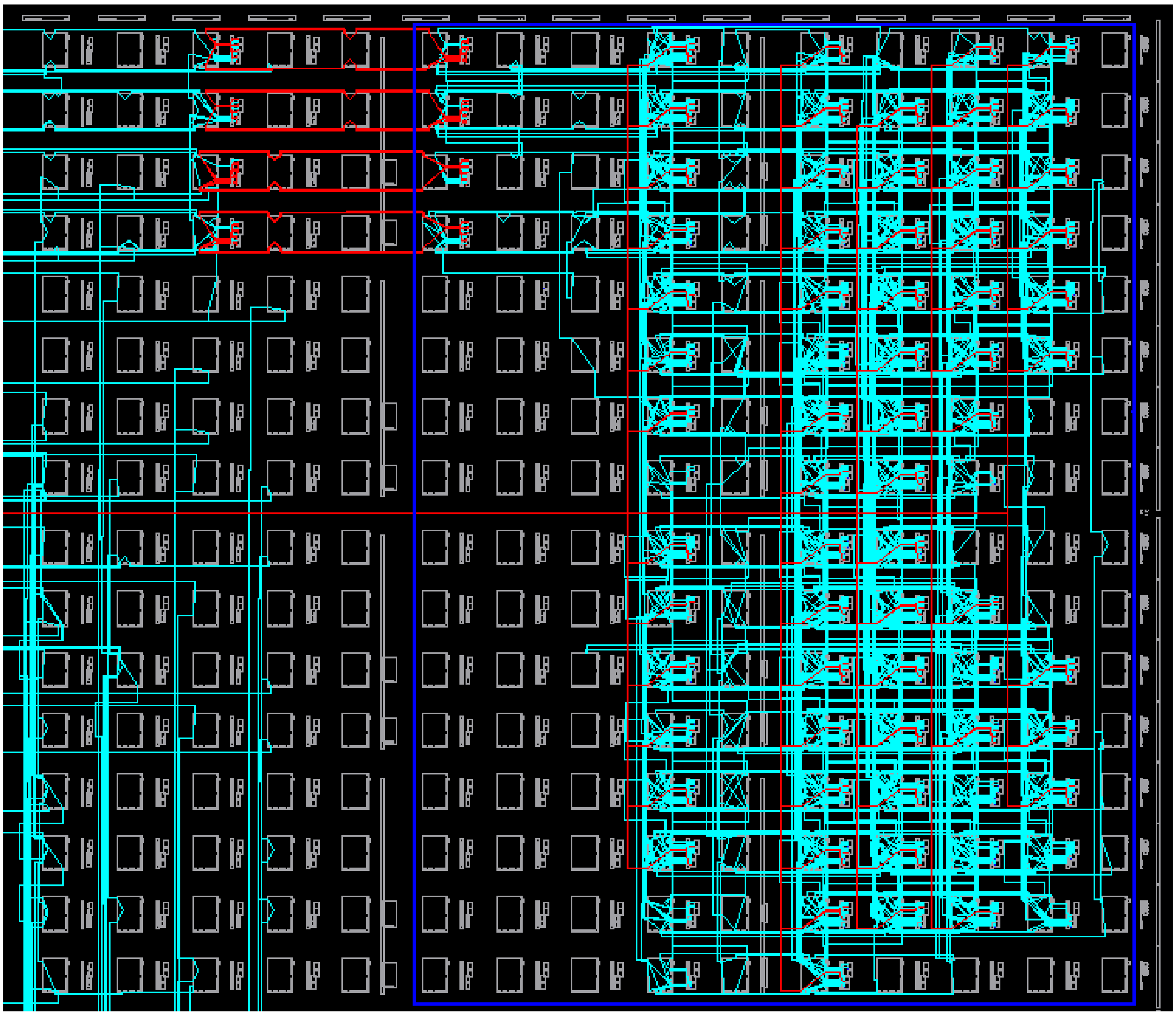


Figure 9.1.1: System and sequential multiplier (Xilinx Platform Studio)

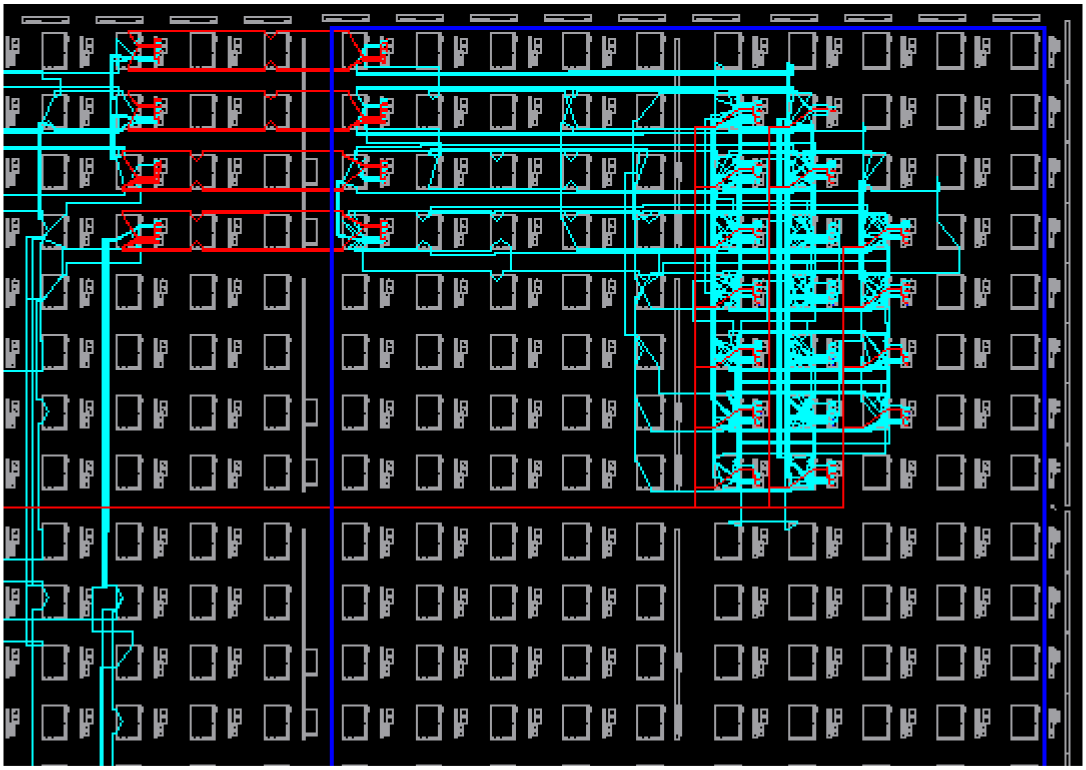


Figure 9.1.2: Sequential comparator (Xilinx ISE)

A referral to [20] is left for further technical details on design of reconfigurable modules in Xilinx ISE. After exporting the two designs to bitfile using *bitgen*, the system file was processed through bitinit and uploaded to the Suzaku board. The partial bistream describing the comparator in figure 9.1.2 was extracted using modifiedCLBread according to source 9.1.2 and icap_write was modified according to source 9.1.3. As can be seen it was modified to start writing at column 20, which is the absolute column address of CLB column 18. An execution trace of the reconfiguration and testing procedure is found in section 10.5.

```
# ./test –i comparator.bit –o comparator_part.bit –fmR –sc 18 –ec 23
```

Source 9.1.2

```
if(write_header(handlemem, 1, 20, 0, frames) < 0){
```

Source 9.1.3

As the above synchronous reconfiguration failed, an asynchronous adder was developed in ISE in the same manner as the comparator. The trace in section 10.5 includes the response to inserting and testing the asynchronous adder.

10 Results

10.1 Register architectures

Figures 10.1.1 – 10.1.4 shows the results of synthesizing generic architectures for different register types. LUT usage is compared with the sum of bits stored by the register collection.

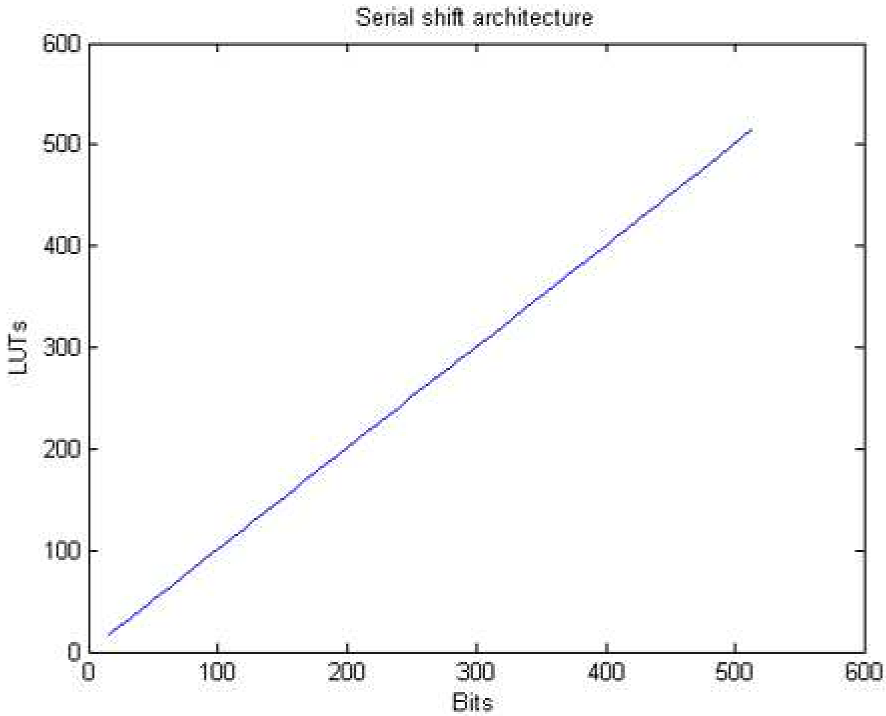


Figure 10.1.1

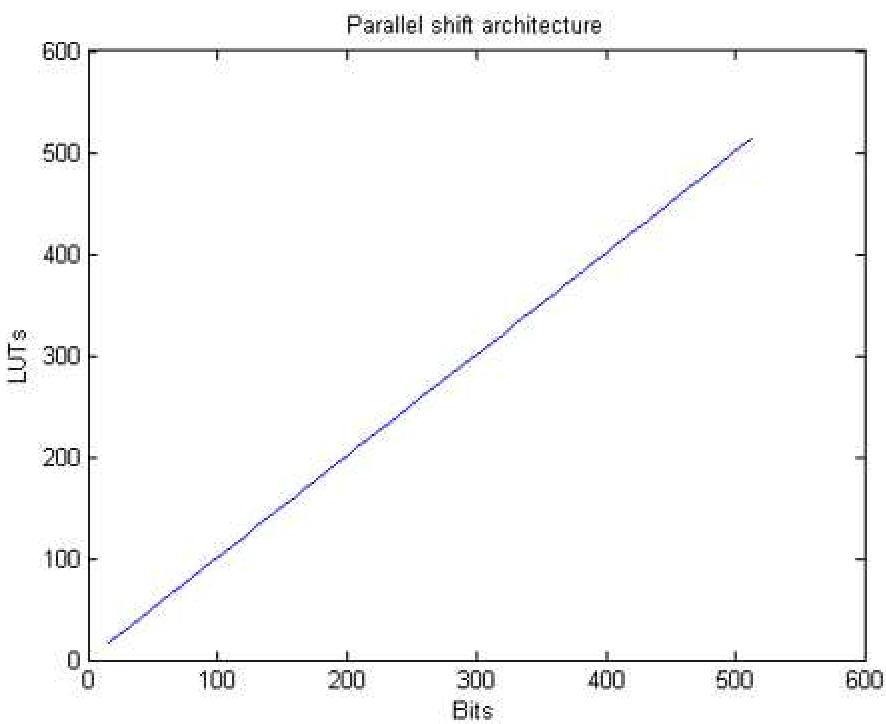


Figure 10.1.2

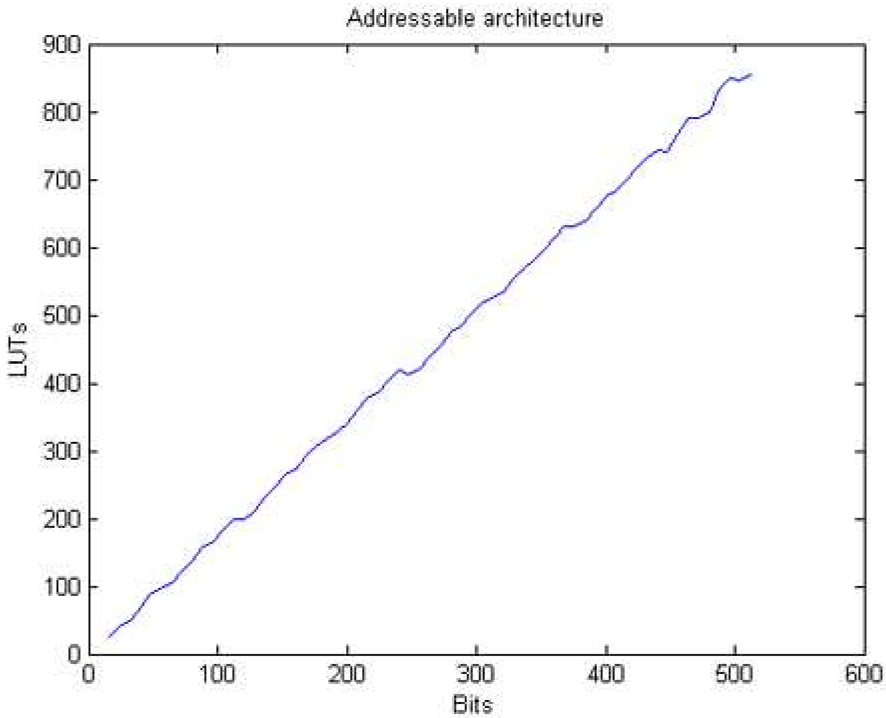


Figure 10.1.3

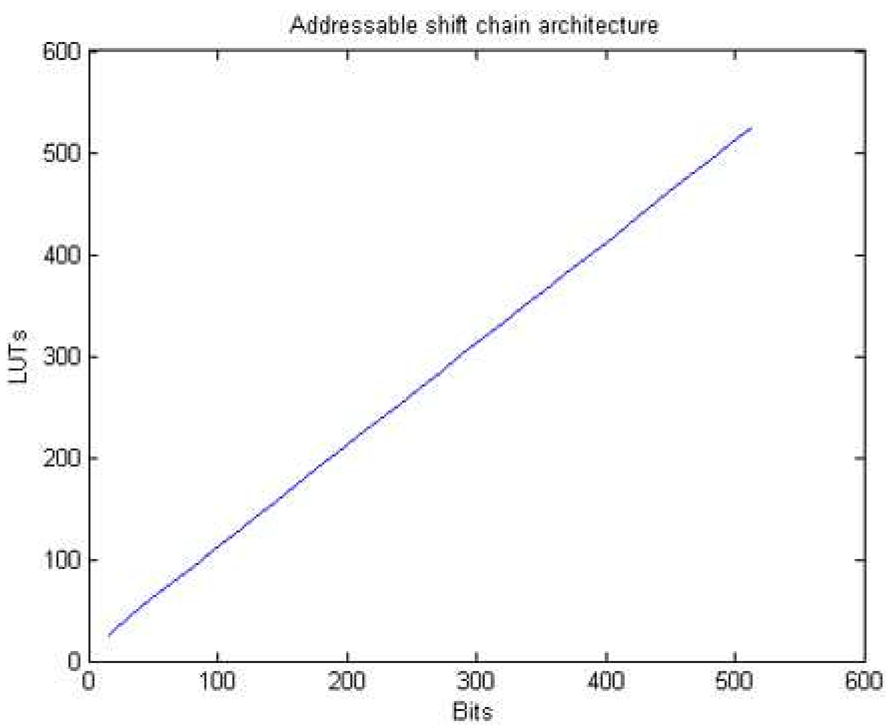


Figure 10.1.4

Table 10.1.1 summarizes the number of IO pins for the different register building blocks from section 5.1.1 analytically. N denotes DFF memory in bits and S denotes the bit width of the shift terminals for the parallel shift block.

Register type	**IO-Pins**
Serial shift block	2N+5
Parallel shift block	2(N+S)+3
Addressable block	3N+2

Table 10.1.1: Pins for different register building blocks

Table 10.1.2 shows the maximum clock frequencies for the different register architectures, which are invariant of the total number of bits stored

Architecture	**CLK[MHz]**
Serial shift	848
Parallel shift	848
Addressable	895
Addressable shift chain	848

Table 10.1.2: Clock frequencies

10.2 Backend bridge

Trace 10.2.1 shows a trace from executing two multiplications with the instruction and data cache backend.

```
# insmod id_cache.ko
Module loaded
Remapping address space…
writing program ...
bram[0]: f000f000  // synchronization halt for start
bram[1]: 8003000c
bram[2]: 8003000d
bram[3]: 8003000e
bram[4]: 8004000f
bram[5]: 10c000
bram[6]: 40128013
bram[7]: 80130011
bram[8]: f0004013  //halt operation
bram[9]: 40148013
bram[10]: 40158013
bram[11]: a2000000
bram[12]: 5bf134f0
bram[13]: abcdef12
bram[14]: abcdef
bram[15]: 4c
bram[16]: 12345678
bram[17]: 12345678
executing ...
bram[18]: ce169d44
bram[19]: 734cc30a
# rmmod bram.ko
# insmod bram.ko
Module loaded
Remapping address space
writing sync ...
writing program ...
reg: a000001
bram[18]: ce169d44
bram[19]: 734cc30a
bram[20]: 1df4d840
bram[21]: 14b66dc
```

Trace: 10.2.1

First $0xABCDEF12^2$. is computed from an interrupted state. This is followed by the computation of $0x12345678^2$, which demonstrates that resuming computation from a halt instructions works. Traces 10.2.2 – 10.2.4 shows ISE synthesis results for the different backends.

Device utilization summary:

Selected Device : 4vfx12sf363-10

```
 Number of Slices:                      91  out of   5472     1%
 Number of Slice Flip Flops:            130  out of  10944     1%
 Number of 4 input LUTs:                 162  out of  10944     1%
 Number of IOs:                         109
 Number of bonded IOBs:                  84  out of    240    35%
 Number of FIFO16/RAMB16s:              1  out of     36     2%
 Number used as RAMB16s:                1
 Number of GCLKs:                       1  out of     32     3%
```

Timing Summary:

Speed Grade: -10

```
   Minimum period: 4.125ns (Maximum Frequency: 242.427MHz)
   Minimum input arrival time before clock: 4.331ns
   Maximum output required time after clock: 7.197ns
   Maximum combinational path delay: 7.958ns
```

Trace 10.2.2 Instruction and data cache

Device utilization summary:

Selected Device : 4vfx12sf363-10

```
 Number of Slices:                 76  out of   5472     1%
 Number of Slice Flip Flops:          92  out of  10944     0%
 Number of 4 input LUTs:             143  out of  10944     1%
 Number of IOs:                   79
 Number of bonded IOBs:                77  out of    240    32%
 Number of GCLKs:                    1  out of     32     3%
```

Timing Summary:

Speed Grade: -10

```
 Minimum period: 4.427ns (Maximum Frequency: 225.866MHz)
 Minimum input arrival time before clock: 2.909ns
 Maximum output required time after clock: 5.951ns
 Maximum combinational path delay: 7.900ns
```

Trace 10.2.3: Data cache

```
Device utilization summary:
---------------------------
Selected Device : 4vfx12sf363-10

 Number of Slices:                      58  out of   5472     1%
 Number of Slice Flip Flops:            91  out of  10944     0%
 Number of 4 input LUTs:               100  out of  10944     0%
 Number of IOs:                        109
 Number of bonded IOBs:                 83  out of    240    34%
 Number of FIFO16/RAMB16s:               1  out of     36     2%
    Number used as RAMB16s:              1
 Number of GCLKs:                        1  out of     32     3%

Timing Summary:
---------------
Speed Grade: -10

   Minimum period: 3.586ns (Maximum Frequency: 278.858MHz)
   Minimum input arrival time before clock: 4.305ns
   Maximum output required time after clock: 7.353ns
   Maximum combinational path delay: 7.648ns
```

Trace 10.2.4: Software register backend

10.3 Performance Evaluation

The following tables list clock cycles consumed for data transfers of varying size. As earlier mentioned evaluation is performed for kernel calls, device driver calls and application calls through the HWOS.

10.3.1 Kernel Space

Sample	Type	Hit	Cycles
1	ID	Yes	92
2	ID	Yes	92
3	ID	Yes	92
4	ID	Yes	95
5	ID	No	618
6	ID	No	619
7	ID	No	616
8	ID	No	616
9	D	Yes	1173
10	D	Yes	1186
11	D	Yes	1173
12	D	Yes	1173
13	D	No	1442
14	D	No	1513
15	D	No	1512

16	D	No	1513
17	SW	#	828
18	SW	#	822
19	SW	#	822
20	SW	#	826

Table 10.3.1.1: Multiplier state loading, 17 bytes

Sample	**Multiple**	**Hit**	**Cycles**
1	1x	Yes	183
2	1x	Yes	183
3	1x	Yes	183
4	1x	Yes	183
5	2x	Yes	255
6	2x	Yes	255
7	2x	Yes	255
8	2x	Yes	255
9	3x	Yes	295
10	3x	Yes	295
11	3x	Yes	295
12	3x	Yes	295
13	4x	Yes	344
14	4x	No	344
15	4x	No	344
16	4x	No	344
17	1x	No	382
18	1x	No	382
19	1x	No	390
20	1x	No	390
21	2x	No	659
22	2x	No	659
23	2x	No	659
24	2x	No	659
25	3x	No	906
26	3x	No	906
27	3x	No	906
28	3x	No	911
29	4x	No	1112
30	4x	No	1112
31	4x	No	1112
32	4x	No	1112

Table 10.3.1.2: Instruction and data cache, multiples of 20 bytes

Sample	Multiple	Cycles
1	1x	822
2	1x	827
3	1x	822
4	1x	827
5	2x	1567
6	2x	1567
7	2x	1567
8	2x	1567
9	3x	2317
10	3x	2317
11	3x	2309
12	3x	2309
13	4x	3077
14	4x	3075
15	4x	3087
16	4x	3069

Table 10.3.1.3: Software register backend, transfer of multiples of 20 bytes

10.3.2 User space

Sample	Type	Hit	Cycles
1	ID	Yes	1944
2	ID	Yes	1900
3	ID	Yes	1938
4	ID	Yes	1943
5	ID	No	2687
6	ID	No	2527
7	ID	No	2528
8	ID	No	2500
9	D	Yes	16108
10	D	Yes	13874
11	D	Yes	14145
12	D	Yes	13699
13	D	No	14949
14	D	No	14635
15	D	No	14805
16	D	No	14531
17	SW	#	9647
18	SW	#	9511
19	SW	#	9295
20	SW	#	9594

Table 10.3.2.1: Multiplier state loading, 17 bytes device driver

Sample	Multiple	Hit	Cycles
1	1x	Yes	1776
2	1x	Yes	1767
3	1x	Yes	1807
4	1x	Yes	1766
5	2x	Yes	1776
6	2x	Yes	1767
7	2x	Yes	1807
8	2x	Yes	1766
9	3x	Yes	1819
10	3x	Yes	1791
11	3x	Yes	1803
12	3x	Yes	1821
13	4x	Yes	1864
14	4x	No	1803
15	4x	No	1785
16	4x	No	1791
17	1x	No	1820
18	1x	No	1787
19	1x	No	1862
20	1x	No	1820
21	2x	No	2734
22	2x	No	2428
23	2x	No	2444
24	2x	No	2442
25	3x	No	2835
26	3x	No	2785
27	3x	No	2835
28	3x	No	2835
29	4x	No	3059
30	4x	No	3029
31	4x	No	3059
32	4x	No	3299

Table 10.3.2.2: Instruction and data cache, multiples of 20 bytes by device driver

Sample	Multiple	Cycles
1	1x	9252
2	1x	9613
3	1x	9284
4	1x	9300
5	2x	15976
6	2x	15821
7	2x	15712
8	2x	15623
9	3x	20079

10	3x	20178
11	3x	20239
12	3x	20238
13	4x	23720
14	4x	23956
15	4x	23800
16	4x	23948

Table 10.3.2.3: Software register backend, multiples of 20 bytes by device driver

Sample	**Multiple**	**Hit**	**Cycles**
1	1x	Yes	20811
2	1x	Yes	16977
3	1x	Yes	17300
4	1x	Yes	17173
5	2x	Yes	17193
6	2x	Yes	17245
7	2x	Yes	17306
8	2x	Yes	17037
9	3x	Yes	17203
10	3x	Yes	17311
11	3x	Yes	17292
12	3x	Yes	17124
13	1x	No	22965
14	1x	No	22964
15	1x	No	23003
16	1x	No	22965
17	2x	No	23307
18	2x	No	23112
19	2x	No	23193
20	2x	No	23173
21	3x	No	26084
22	3x	No	23380
23	3x	No	23712
24	3x	No	23539

Table 10.3.2.4: Instruction and data cache backend, multiples of 20 bytes through HWOS

10.4 HWOS

Trace 10.4.1 shows the execution of the HWOS followed by the execution of an application accessing a hardware defined sequential multiplier. Its instructions to the multiplier is the computation of $0xABCDEF12^2$ followed by the computation of $0x12345678^2$. Printing the return values shows the correctness of the execution.

```
# ./hwos
# ps | grep hwos
 172 root       256 S   ./hwos_d    //HWOS process
# cd ../multiplier
# ./app
 Connected to master message queue...
 Creating receive message queue...
 writing data to BRAM ...
 executing ...
 RES00 = ce169d44
 RES01 = 734cc30a
 RES10 = 1df4d840
 RES11 = 14b66dc
```

Trace 10.4.1

10.5 Run time reconfiguration

```
# insmod mult.ko
Module loaded
Remapping address space
writing sync ...
writing program ...
resetting timer...timer 0
executing ...
bram[18]: ce169d44
bram[19]: 734cc30a
bram[20]: 1df4d840
bram[21]: 14b66dc
# cd /var;
# ls
run  tmp
# mknod icap c 250 0 //Create device file
# ls
icap  run   tmp  //Device file created
# cd tmp/dyn_reconf/xilinx_hwicap/
# insmod xilinx_hwicap.ko  // Insert Xilinx ICAP device driver
Xilinx ICAP driver init
Xilinx ICAP driver platform_driver_register
Xilinx ICAP driver probe
Xilinx ICAP driver setup
# cd ../icap_write
#./icap_write -i comp_part.bit -f 176 //Reconfigure 8 columns
...
...
# cd ../comparator/module
# insmod comp.ko
bram[18]: fffffff
```

```
bram[19]: ffffffff
# rmmod comp.ko
#cd ../../icap_write
#./icap_write –i add_part.bit -f 176 //Reconfigure 8 columns
…
…
#cd ../adder/module
# insmod add.ko
bram[18]: dddddddd
bram[19]: dddddddd
# rmmod add
# insmod add.ko
bram[18]: dddddd29
bram[19]: dddddd29
```

Trace 10.5.1

Trace 10.5.1 shows the execution of the sequential multiplier realized as a true reconfigurable module. It can be seen that it works correctly which in turn proves that the piping of signals through bus macros works. After the execution of the multiplier module a device file is created for the ICAP linux device driver. Reconfiguration is performed, replacing the multiplier with a comparator. Insertion of a test module shows that all outputs are corrupted, pulled high, after reconfiguration. Output value was invariant of any applied inputs. Finally the comparator is replaced with an asynchronous adder adding 0xDB to the input value. With an input of 0 the resulting in the faulty answer is DD. Applying 0xEF as stimulus yields 0x29.

11: Discussion

Section 4 specifies the basis and goals for this thesis. Creating components for hardware-software intercommunication in a dynamically reconfigurable system has involved a lot of details. Some of the most important aspects and findings of the work will now be discussed.

When considering what register architecture might be suitable for a reconfigurable module many factors comes into play. One important result from synthesis in section 10.1 is that when implementing a shift-architecture there is no extra cost of expanding the shift bit width. A single shift register and a multiple shift register are mapped to identical FPGA components; the only difference is the routing of resources. Bandwidth can thus be increased by increasing the shift factor. There are some side effects of this proposal though, as embedding other modules and developing controllers might prove to be harder. Creating chains of multiple-shift registers is likely to include DFF's that are not used but are necessary for scan chain consistency. The number of unused DFF's is likely to increase with the shift size. Another potential problem arises if the reconfigurable module uses single bit shift register in processing algorithms. Implementing both single and multiple bit shift for a register will produce a noticeable LUT consumption increase. A large reconfigurable interface might also be a practical problem.

Comparing the generic register architecture synthesis results one can see that the shift based solutions require less resources than the addressable architecture. The random access property thus comes with a cost. It is also important to question whether a random-access architecture is necessary in the context of reconfigurable modules and run time reconfiguration. Considering the multiplier implementation in this paper the addressable architecture could not provide any immediate advantages compared to the shift based architecture. On the contrary the shift based architecture is able to operate in full duplex mode and simulations show that the addressable RAM architecture actually performs worse. Creating a full duplex random architecture would increase performance but require even more resources.

The hybrid architecture uses shifting for data transfer but implementing an address space for shift-chain addressing which seems to be a good compromise. It allows utilizing the full duplex property while providing more manipulation flexibility through multiple shift chains. For large bit strings it is an obvious simplification to be able to shift content as opposed to accessing it through multiple addressed calls. Resource requirements of the hybrid architecture will depend on how extensive addressing the architecture provides. In section 10.1 one can see that an address space of 2 has virtually no cost when comparing figure 10.1.2 with figure 10.1.4.

All backend implementations have been verified and work properly. Backend performance testing surprisingly reveals that the data cache solution has a very poor performance. The fact that it is slower than a regular software register solution can be explained by latency ratios and controller structure. As the data caching is done through a BRAM cell, extra cycles are involved in data transfers compared to the software register backend. With a software accessible instruction register, data caching cannot be exploited as execution is too slow. Expressed in another way the data cache bakend is a poorly balanced system. System latencies are simply not large enough to make data caching profitable. Speedup would require a faster flow of instructions or alternatively instructions could be more complex. Effectively making the data cache backend more independent of CPU control could yield better performance, but this is exactly what the instruction and data cache backend does.

The instruction and data cache solution provides good performance relative to the other two. When loading the 17 byte interrupt state the instruction and data cache backend provides a fear speedup for both cache hit and cache miss compared to the software register solution. When loading multiples of 20 bytes an increased speedup ratio compared to the software register backend. The instruction and data cache backend is roughly 5-10 times faster than the software when comparing both device driver and kernel space results. No cache policies or such has been discussed in this thesis but with a reasonable hit ratio the instruction and data cache backend can indeed provide a pretty good speedup. Looking at timer values for hardware access through the instruction and data cache backend using the HWOS in table 10.3.2.4, it is also suggested that large statistical oscillations occur. As the large variance occurs solely for the HWOS testing this might suggest that system level testing requires a more statistical approach. Judging by the other performance results the simple performance test system has been adequate though. From section 10.3 it can also be seen that accessing hardware from user space produces a major increased in delay compared to doing it directly from kernel space. An obvious reason is the extra software overhead incurred in device drivers and applications. This emphasizes the importance of fast software components in reconfigurable co-processing system.

Synthesis of the backend modules for a fixed generics shows that the resource requirements are proportional to the complexity of the backends. Disregarding the BRAM, results reveals that the variance in resource requirements for the different backend modules is small. As the three different backed implementations in this thesis have roughly the same resource requirements if disregarding the BRAM, the instruction and data backend can provide good speedup for little resource overhead. If a system contains parts with large RAM requirements this might be a challenge though.

When evaluating the absolute speed of the communication link that has been created, table 10.3.1.1 shows a time of 90-100 cycles for loading a multiplier state with the instruction and data cache backend. Comparing this to simulations, which show an optimal time of some 60 cycles with no overhead involved, the link is indeed quite fast. System synthesis show a PLB clock frequency of roughly 120 MHz which can be used to estimate performance in time from cycle performance in section 10.3.

Work on the MPEG transcoder case has suffered from time limitations in this thesis. As the inverse scan module from section 9.1 depended on a set of large LUT based register files it was decided to have a look at how to handle requirements for special cells as part of a reconfigurable module. A correctly simulating controller pair solution has been designed to handle migration of the RAM module part of the MPEG-4 decoder. Although it is an obvious solution it provides flexibility for reconfigurable modules. A great risk is limitations due to the bandwidth of the reconfigurable interface however. Such elements should be considered when tuning the reconfigurable interface.

Much of the work in this thesis is weakly related to earlier work done by Sverre Hamre in [4]. As a result it was not prioritized to extend the originally proposed HWOS but rather build a prototype including functionality developed in this thesis. It is emphasized that that the HWOS in only a template as there are many dependencies and factors to consider when developing a full feature version. Dynamic memory allocation can provide a flexible API to external applications when using a cache based backend bridge. Considering HWOS performance, shown in table 10.3.2.4, it can be seen that delays are quite large compared to device driver and kernel module calls in 10.3.2.2 and 10.3.1.1 respectively. Memory protection and other features might further reduce system throughput. It is implied that a compromise of security, functionality and performance must be made.
Use of Xilinx ISE to create reconfigurable modules has been applied in this thesis. Results show that it can radically improve design time compared to building a full system for the purpose of extracting

reconfigurable modules. No formal time comparisons have been done but the example comparator from section 9 took roughly 5 minutes to build in ISE whereas a Platform Studio flow would likely require 30-40 minutes. Consistency must be verified thoroughly when separating reconfigurable module design from the system itself though.

Performing run time reconfiguration with a synchronous basis using the framework for self reconfiguration fails however. This was also the assumption when testing as the framework deals solely with LUTs. In the theory section of [3] it is described how there are separate frames in the bitstream for routing of clock signals. As noted in section 9 the clock signal does indeed behave in a different way than ordinary signals as it will not be routed through bus though it might be possible to hack such a mapping. Replacing the multiplier with the comparator with run time reconfiguration seems to fail completely as all outputs take the static value 1. Inserting the asynchronous adder seems to produce less corruption however. Although reconfigurable module design in Xilinx ISE should be demonstrated with a working example this was not done. [6] has showed correct reconfiguration with an asynchronous basis and the NCD files created with Xilinx ISE in this thesis suggest that there is nothing wrong with the ISE design flow. Reconfiguration of synchronous designs seems to fail because the clock signal is distinguished from normal signals, which is not handled by the framework for self reconfiguration.

12: Conclusion

By analyzing requirements and decomposing it into components, a fast link for hardware-software communication for intentional use in a reconfigurable system has been created. Most alternatives have been evaluated through implementation. Though the implementation process is time consuming it has provided accurate metrics for deciding which branches to follow. This thesis claims that a mixed shift and addressable register architecture is the best choice for reading and writing data to a reconfigurable module. Furthermore it has been shown that a cache-controller backend can provide good speedup for hardware-software intercommunication. A software-to-hardware API for the developed hardware components has been provided through Linux device drivers and a HWOS template as a top level entity. A separation of applications from accelerator hardware and system mechanisms has also been accomplished through the HWOS.

Implementation on a Suzaku-sz410 board has confirmed that the system and its parts are working. That said some components of the communication framework have been designed without a very thorough pre-analysis. Primarily this is the case for the HWOS where there might be much to gain by proper analysis and design.

Design of reconfigurable modules using Xilinx ISE has been demonstrated. Run time reconfiguration of synchronous reconfigurable modules has been shown to fail. As the clock signal is distinguished from regular signals, support for reconfiguration of clock networks must be added to the framework for self reconfiguration to enable synchronous reconfigurable modules.

13 Further work

Sverre Hamre's framework for dynamic reconfiguration should be merged with the functionality of the HWOS developed in this paper. As mentioned in [3] the framework should be used carefully as it potentially can destroy FPGA chips. It is also fronted that some more work should be done as some functionality is not implemented or incomplete like reconfiguration of clock networks. The HWOS template developed in this thesis is as mentioned incomplete and unpolished. Its primary objective is to act as an application interface and lacks parts for managing run time configuration. In many aspects it is not thoroughly designed which opens for improvements and modifications.

Another branch of work is combining the hands on dynamic reconfiguration with AHEAD's NOC project. As of now the hardware-software communication infrastructure assumes a direct system bus connection between the backend and the processor. In order to be able to use the developed backend architecture with a NOC framework, a translation layer is needed to replace system bus connections. Although the concept might be straight forward this will probably require a fair amount of work.

14 References

[1] Compton and Hauck, Reconfigurable computing: A survey of systems and software, 2002
[2] Glesner,Hollstein…, Reconfigurable Platforms for ubiquitous computing, 2004
[3] S. Hamre, Framework for self reconfigurable system on a Xilinx FPGA, 2009
[4] http://download.atmark-techno.com/
[5] S. Hamre, Tutorial for adding an adder to the suzaku hardware, 2008
[6] V. Endresen, Hardware task execution in reconfigurable systems, 2009 *
[7] W. Wolf, A Decade of HW/SW Codesign, IEEE Computer, 2003, Chapter 7: Hardware accelerators
[8] http://lapwww.epfl.ch/courses/archord1/labs/A_8bit_Sequential_Multiplier.pdf
[9] P. Marwedel, Embedded system design, Chapter 5: Implementing embedded systems, Springer, 2006
[10]http://www.ece.gatech.edu/academic/courses/spring2007/ece4170/MiscDocuments/Xilinx%20Implementation%20Tutorial.pdf
[11] http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf
[12] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel 3rd edition, O'Reilly, 2000
[13] Corbet,Rubini,Kroah-Hartman, Linux device drivers 3rd edition, O'Reilly,2005
[14] http://en.wikipedia.org/wiki/Memory_pool
[15] http://en.wikipedia.org/wiki/Buddy_memory_allocation
[16] http://peterlombardo.wikidot.com/linux-daemon-in-c
[17] Krohn,Linnerud, MPEG transcoder for Xilinx Spartan, 2008
[18] V. Endresen, Reconfigurable system design, 2009 *
[19] http://newsgroups.derkeiler.com/Archive/Comp/comp.arch.fpga/2007-12/msg00133.html
[20] V. Endresen, Reconfigurable module design in Xilinx ISE, 2010 *

* Found at http://folk.ntnu.no/vegarend

15 Thanks to

- Professor Kjetil Svarstad as project coordinator.
- Ph.D student Bahram Najafi Uchevler and M.Sc candidate Andreas Hepsø for advice.
- Xilinx representative Jan Anders Mathisen for providing the Xilinx Design Suite 10.1.
- Øyvind Strømsvik for evaluating final report drafts and providing feedback.