EFFICIENT HARDWARE FOR LOW LATENCY APPLICATIONS

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Christian Harald Leber
(Diplom-Informatiker der Technischen Informatik)

aus Ludwigshafen

Mannheim, 2012

Dekan:	Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:	Professor Dr. Ulrich Brüning, Universität Heidelberg
Korreferent:	Professor Dr. Holger Fröning, Universität Heidelberg

Tag der mündlichen Prüfung: 20.8.2012

Für meine Eltern

Abstract

Abstract

The design and development of application specific hardware structures has a high degree of complexity. Logic resources are nowadays often not the limit anymore, but the development time and especially the time it takes for testing are today significant factors that limit the possibility to implement functionality in hardware. In the following techniques are presented that help to improve the capabilities to design complex hardware.

The first part presents a generator (RFS) which allows defining control and status structures for hardware designs using an abstract high level language. These structures are usually called register files. Special features are presented for the efficient implementation of counters and error reporting facilities. Native support for the partitioning of such register files is an important trait of RFS.

A novel method to inform host systems very efficiently about changes in the register files is presented in the second part. It makes use of a microcode programmable hardware unit. The microcode is auto generated based on output from RFS. The general principle is to push changed information from the register file to the host systems instead of relying on the standard pull approach.

In the third part a highly efficient, fully pipelined address translation mechanism for remote memory access in HPC interconnection networks is presented. It is able of performing up to one translation per cycle and provides efficient interfaces for software to control the content of its n-way set associative translation cache. The mechanism is centered on a pipeline with a new concept to resolve dependency problems.

The last part of this thesis addresses the problem of sending TCP messages for a low latency trading application using a hybrid TCP stack implementation that consists of hardware and software components. This separation allows low latency while keeping the complexity under control. Furthermore, a simulation environment for the TCP stack is presented, which is used to test the stack within a hardware simulator. This hardware-software co-simulation allows the acceleration of testing and debugging of complex hardware designs. It allows employing standard components like an operating system kernel instead of implementing the functionality again within the simulation environment.

Zusammenfassung

Das Design und die Entwicklung applikationsspezifischer Hardwarestrukturen bedingen einen hohen Grad an Komplexität. Die Menge an Logik Ressourcen ist oft nicht mehr das Limit, sondern die Entwicklungszeit und im speziellen die Zeit, welche zum Testen benötigt wird, sind heute ein signifikanter Faktor welcher die Möglichkeiten Logik in Hardware zu implementieren limitiert. Im Folgenden werden Techniken präsentiert, welche die Fähigkeiten komplexe Hardware zu entwickeln verbessern.

Der erste Teil präsentiert einen Generator, welcher es ermöglicht Kontroll- und Status-Strukturen für Hardware Designs mittels einer abstrakten Hochsprache zu beschreiben. Diese werden für gewöhnlich Registerfiles genannt. Spezielle Funktionen für die effiziente Implementierung von Zählern und Einrichtungen zur Indikation von Fehlern werden vorgestellt. Zentrales Element dieser Entwicklung ist die Fähigkeit die Partitionierung von Registerfiles direkt zu unterstützen.

Eine neuartige Methode um das Hostsystem sehr effizient über Änderungen in besagtem Registerfile zu informieren wird im zweiten Teil präsentiert. Dafür wird eine durch Mikrocode programmierbare Hardwareeinheit genutzt. Der Mikrocode wird basierend auf der Ausgabe des Registerfile Generators automatisch erstellt. Das generelle Prinzip beruht darauf geänderte Informationen vom Registerfile zum Hostsystem zu schicken, anstatt sich auf den Standardansatz zu stützen, dass das Hostsystem die Daten abholt.

In einem dritten Teil wird ein hocheffizienter Mechanismus zur Übersetzung von Adressen für den Zugriff auf entfernte Speicher in HPC Verbindungsnetzwerken gezeigt. Der Mechanismus ist in der Lage bis zu eine Übersetzung pro Takt abzuarbeiten und stellt effiziente Schnittstellen zur Verfügung, so dass Software den Inhalt des n-fach assoziativen Zwischenspeichers für Übersetzungen kontrollieren kann. Dieser Mechanismus benutzt eine Pipeline mit einer neuen Lösung um Abhängigkeiten aufzulösen.

Der letzte Teil dieser Dissertation befasst sich mit dem Problem TCP Nachrichten mit einer möglichst niedrigen Latenz für den Hochgeschwindigkeitshandel an der Börse zu versenden. Dazu wird eine hybride TCP Implementation, welche aus Hardware und Software Komponenten besteht, präsentiert. Diese Trennung ermöglicht niedrige Latenz und erlaubt

die Komplexität unter Kontrolle zu behalten. Weiterhin wird eine Hardware-Software Kosimulation für die besagte TCP Implementation präsentiert, welche dazu dient um diese innerhalb eines Hardware Simulators zusammen mit Standard Komponenten wie einem Betriebssystemkern zu testen, anstatt die Funktionalität innerhalb des Simulator neu zu implementieren.

Contents

TOC

Introduction

Chapter 1

The complexity of recent hardware designs is increasing, and therefore ways are required to handle this complexity efficiently in terms of development time and quality.

“Moore's Law” [29] is still a driving factor in the semiconductor sector, consisting not only out of industry companies but also of academic institutions that have to search for the coming problems and their solutions. The “Law” was updated in 1975 [39] and states that the amount of components per chip doubles every two years. The straight line in figure 1-1 shows the development of the transistor count for CPUs and as it can be seen, technology is keeping up with this prediction. The amount of logic does of course implicate expectations. People want to implement more and more complex functionality in hardware.

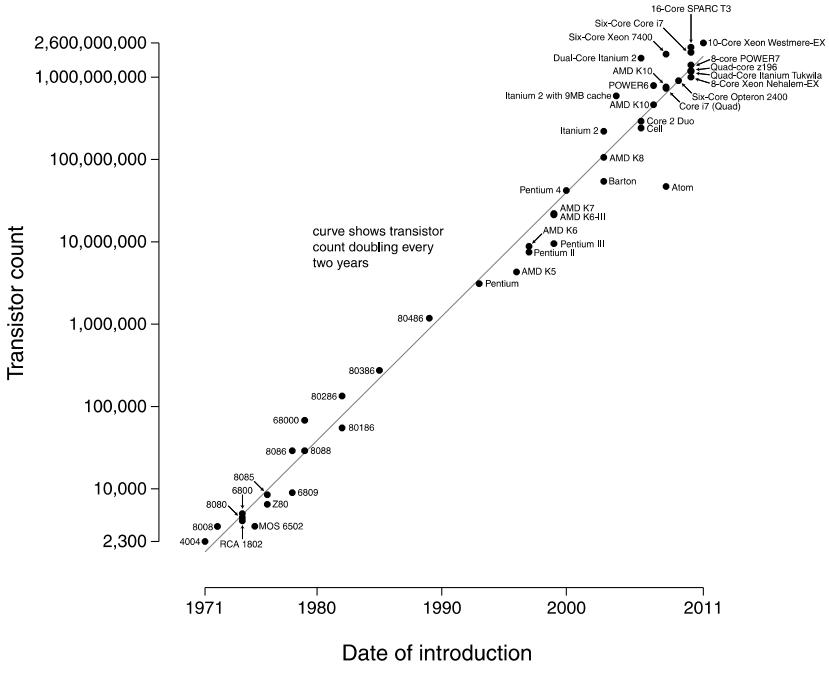


Figure 1-1: Microprocessor Transistor Counts 1971-2011 (from [93])

FPGAs are of course growing with the same rate. Implementations on big high-end FPGAs are the primary focus of this work, because they offer the possibility to implement big designs in hardware. Of course, the possibility of ASIC implementations is also considered, because these offer higher clock frequencies and lower costs, if produced in high volume. Also ASICs allow a fully custom configuration of I/O resources. FPGAs provide a wide range of I/O capabilities for different applications, but this wide range does not fit all applications.

Table 1-1 shows the date of the press announcement for the last three Xilinx high end FPGA product lines. The “maximum LE” (**L**ogic **E**lement) number is the marketing size of the biggest FPGA in each product line, but the “LUT 6” column is the actual number of look-up tables in the biggest FGPA. These numbers provide only three data points, but the exponential tendency can be recognized.

Please note that the announcement date has only a weak correlation to the date of actual availability. The announcement date was only chosen because it is possible to find these dates. There is no information published about the actual availability of production FPGAs.

Announcement	FPGA	maximum LE	LUT 6
May. 2006 [137]	Virtex 5	330K	207360
Feb. 2009 [138]	Virtex 6	760K	474240
June 2010 [139]	Virtex 7	2000K	1221600

Table 1-1: Xilinx Virtex 5-7 Announcements

The problem is in the end that ASICs and as a result also FPGAs are growing with Moore‘s law, but there is a growing gap [84] between the possibilities to develop correctly working logic and the available amount of logic resources. Hardware development capabilities do not scale with the growth of the available logic resources.

Therefore, methods are required to reduce the development time and error probability. A very good possibility to reach the target is the development of tools that make use of a higher abstraction level. A higher abstraction level also eases reusability and accelerates the design process. In this work different methods and development tools are described that simplify the development of several state of the art research projects.

1.1 Problems, Goals and Solutions

The questions of design abstraction and acceleration, the implementation of pipelined micro architectures and the simulation of complex application specific hardware structures are discussed in four different contexts:

- register file generator (RFS)
- system notification queue (SNQ)
- address translation unit (ATU2)
- trading accelerator

Modern and complex hardware designs have a big amount of settings and status information, which are accessible in a so called register file. Efficient methods are required to define such structures on an abstract level. A generator will be presented that allows describing these structures in a high level description language. This generator supports structures that ease the development of hardware structures with a multitude of new features.

The complex hardware structures are often used in combination with host systems and these host systems have to be informed about status information changes fast and efficiently. Thus, a novel programmable hardware unit has been designed for this task. This unit is controlled by microcode which is automatically generated based on the high level description of the register file.

HPC interconnects have to provide, for performance reasons, remote memory access (RMA) facilities. Furthermore, it is latency wise of advantage if RMA accesses can be caused from user level processes without the help of the operating system kernel. Therefore, a fast address translation mechanism is designed for the EXTOLL interconnection network. The mechanism is also providing a security mechanism to protect the user level processes from each other.

A low latency hardware accelerator for stock trading applications requires a method to send orders to the stock exchange with a very low latency. Thus, an innovative approach for a hybrid hardware-software TCP stack is presented. It implements the possibility to send TCP packets with a very low overhead from user-level software. The system test of this complex hardware system is accelerated with a novel hardware-software co-simulation test environment that allows connecting a hardware simulator efficiently with software components.

1.2 Environment of the Hardware Structures

To prepare the introduction of the following two applications specific hardware structures it is necessary to introduce the environment these hardware designs will be based on.

All designs assume that the whole system is composed of a host system that is connecting to the hardware structure with a packet based protocol. This hardware structure can either be implemented in an FPGA or an ASIC. Figure 1-2 depicts such a setup. In the following ASICs and FPGAs are considered to be equivalent, because for the design discussions it is usually no concern if the actual physical device is reprogrammable. Therefore, the following writing will only use the term FPGA.

The hardware structure is composed of multiple functional units (FU), which are interconnected with a crossbar. The RF is also a FU, because it is connected to the crossbar to be accessible from the host system. It contains structures that can be used to control the other FUs. Automatic methods to generate such a structure will be presented in chapter 2.

FUs are using the HyperTransport [30] on Chip (HToC) protocol to communicate with each other, and the protocol is also translated by a bridge to the interface of either a PCIe or HT core.

Therefore, the resulting design principle divides all hardware modules into two classes, the platform specific ones and the platform independent ones. Each different PCB with FPGA can be considered as its own platform. Even when the same FPGA is used there are often big differences. For example, one board might use PCIe and another one HT. A “top_file” describes in hardware designs the hardware description language (HDL) file that contains all the instances that are to be included in the hardware device.

The actual **top_file** contains besides the core_top_file, also the platform specific parts, which consist of at least the following:

- PCIe or HT core
- bridge between PCIe/HT core and HToC
- clocking and reset infrastructure
- board specific interfaces like I2C

All the modules that are contained in the **core_top_file** are as a result independent from the respective PCIe or HT core that is used. Furthermore, it would be easily possible to also support another interface by writing a bridge for this interface. For example, AMBA [128] may be a candidate that could be supported in the future, because it is the protocol that can be used to interface with ARM cores. These cores can also be found on new FPGAs [129].

In this work elements of two designs will be described. Both designs are used with PCIe and HT on different FPGA boards and they share the same general structures.

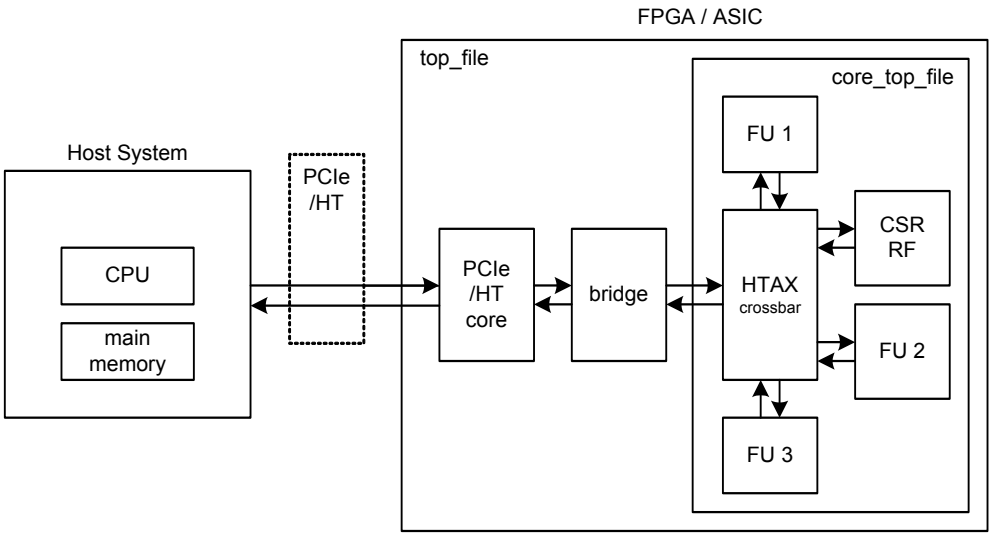


Figure 1-2: Generic Design Environment

1.2.1 HyperTransport on Chip Protocol

This protocol is used by the FUs as described previously. The protocol is an enhanced HyperTransport (HT) protocol that solves the following key problems in the HT protocol:

- only 32 source tags (*SrcTag*)
- a maximum payload size of 64 Bytes

The differences between HT and HToC are relatively small. Therefore the translation between these two protocols is of limited complexity as long as the limitations of such a translation are accepted.

Especially the payload size has to be considered, the bridge that is currently used to interface the HToC protocol with HT does not support splitting up packets. HT packets have a maximum payload size of 64 byte, but in the case of HToC 4 kB are possible. Therefore it is necessary that FUs are aware of the maximum transmission unit (MTU) of their communication destination.

In figure 1-3 the general layout of the header is shown. Not all fields should be described here, but the most important ones are *Cmd*, *SrcTag*, *Count* and *Addr*, these fields have the same functionality like in the case of HT. However, to overcome the packet size and number of source tags restriction extensions for the *Count* and *SrcTag* are provided by *ext_count* and *ext_srcTag*.

The *Cmd* describes the type of the packet, the encoding is the same as in case of HT, and the following are the most important ones:

- posted write (6'b101101): double word write
- read request (6'b011101): double word read
- read response (6'b110000)

The read request and response are matched with the help of the *srcTag*. In the case of double word writes or reads the *Count* field has the number of double words, but the counting starts at zero. Therefore, zero means a packet size of one double word.

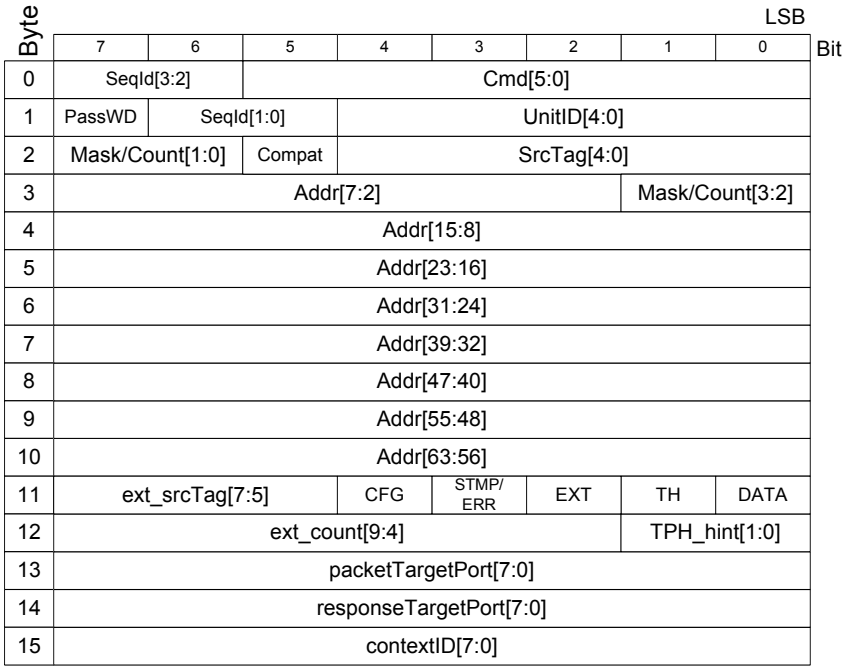


Figure 1-3: HToC Packet

This protocol is meant to be used with the HTAX [38] on chip network. In the development of the CAG this refers to a crossbar switch that is used in different research projects [48]. For multi-stage HTAX crossbars the HToC protocol does contain the *packetTargetPort* and *responseTargetPort* fields.

The HToC protocol can be used either 64 bit or 128 bit wide, an example of a packet with 16 byte data and how it will be transferred over a 64/128 bit wide interface is shown in figure 1-4.

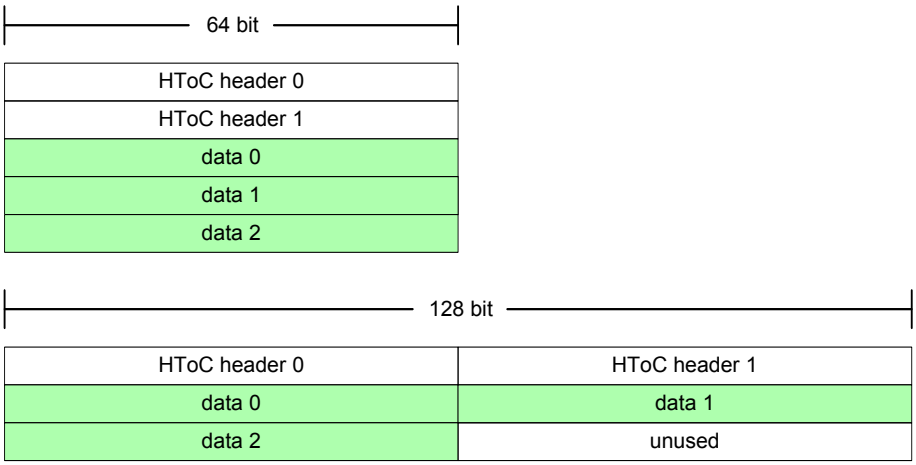


Figure 1-4: HToC 64 and 128 Bit Wide

Virtual Channels (VC) are used on the HTAX to transport the different packet types. Therefore the HTAX is configured with three VCs:

- VC 0: posted writes
- VC 1: read requests
- VC 2: read responses

1.2.2 Interface to the Host System

The HTAX crossbar has to be connected to the host system so that the functional units (FU) are able to communicate with it. Three types of packets have to be supported in both directions:

- posted writes
- non-posted reads
- responses

Implementations exist that allow connecting the HTAX with a bridge to HT and to PCIe.

The first PCIe core that was supported with such a bridge [81] was the PCIe core from Lattice [126]. With the rising interest in PCIe hardware a layered bridging module was developed [114], which is separated in a general bridging modules and a vendor specific part. This separation reduces the development time that is required to support different PCIe cores.

As a result it is possible as of this writing to connect designs that make use of the HTAX crossbar and the HToC protocol to the following host interfaces:

- Altera PCIe core
- CAG HT1/HT2 core
- CAG HT3 core
- Lattice PCIe core
- Xilinx PCIe core

1.2.3 Hardware

For the EXTOLL project special FPGA boards has been developed that provide enough high density connectors, because there are no commercially available FPGA boards providing them. These connectors are required to build EXTOLL interconnection networks.

HyperTransport is the main development platform for the EXTOLL project, therefore first a board for the HTX slot was designed and build. This board is shown in figure 1-5 and named Ventoux. The board makes use of a Xilinx Virtex6 240LX (xc6vlx240t) FPGA and provides six network links with 4x6.25 Gbit/s transfer rate each.

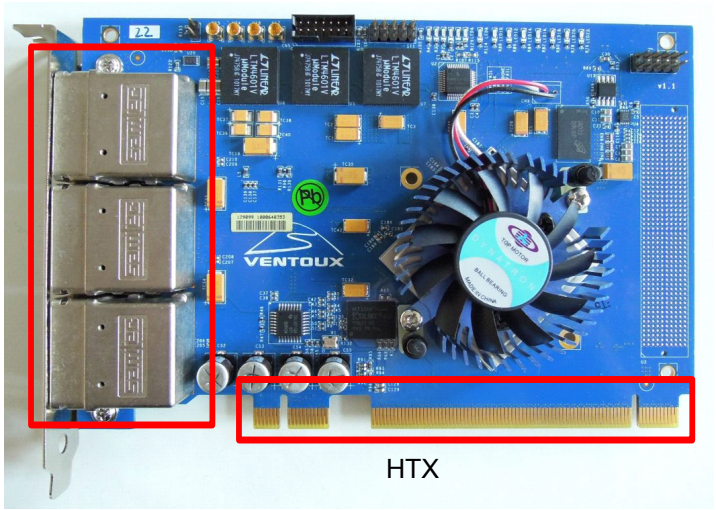


Figure 1-5: Ventoux FPGA Board

Because of the wider availability of systems with PCIe slot also a PCIe board was designed. This board has the name Galibier and is based on the design of Ventoux, despite that two of the network links had to be removed, because the serial I/O is required for the PCIe connector. It is depicted in figure 1-6.

Both boards have a flash memory that is used to store the bit file. The bit file is the configuration for the logic of the FPGA.

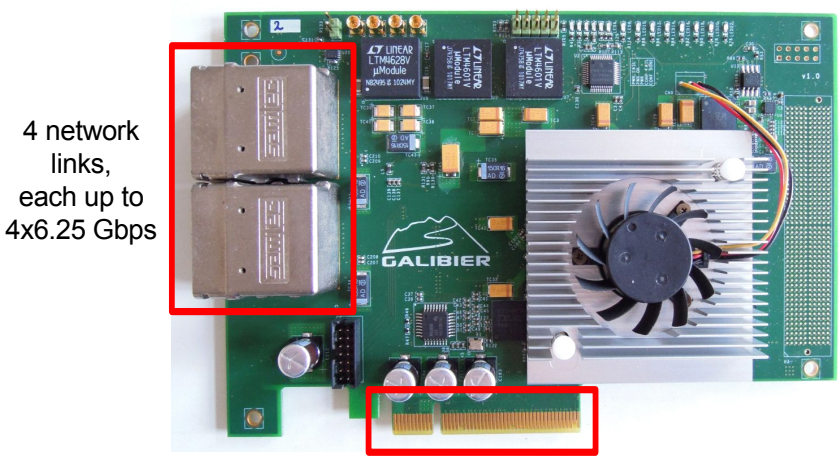


Figure 1-6: Galibier FPGA Board

1.3 Application Specific Hardware Structures

The descriptions and discussions of the different issues and approaches in the context of this thesis require the introduction of two research projects. The main research project of the Computer Architecture Group (CAG) [40] at the University of Heidelberg is EXTOLL, it is an effort to build a HPC interconnect.

A second hardware design that will be used as research object in this thesis accelerates trading applications at stock exchanges, especially in terms of latency.

1.3.1 EXTOLL

For HPC networks not only the bandwidth, but also the latency is important.

Therefore the ATOLL (Atomic Low Latency) [42] network has been developed with the target to provide low latency communication with off the shelf components. It was produced in a 180 nm ASIC process and had four links that allow building 2D tori.

EXTOLL R1 is the successor of the ATOLL project. It is an abbreviation for extended ATOLL. It was targeted for FPGAs and was implemented on HTX boards [41] with Xilinx Virtex 4 FPGAs. Two clusters were established to evaluate the network [90].

New requirements, in terms of bandwidth, in combination with the growing amount of available resources in modern high end FPGA made a redesign of many parts of EXTOLL R1 necessary. EXTOLL R2 does not only target FPGAs, but also an ASIC implementation and is therefore designed to be parameterized for the respective technologies.

EXTOLL R2 has all features of R1, but with improved functionality and performance. An example for such a unit will be shown in chapter 4 starting at page 137.

In this thesis EXTOLL is referring to EXTOLL R2, and the older EXTOLL R1 will be called EXTOLL R1 when this differentiation is necessary.

In the following a very short introduction to HPC networks will be given to prepare the ground for further explanations.

Network Topology

Network topologies can be divided in direct and indirect topologies. Figure 1-7 shows five current networks and how they can be divided in the direct and indirect category. The presentation methods for design space diagrams in this work are described in appendix A. Ethernet and Infiniband are well known indirect networks. The “K Computer” [133] is as of November 2011 the fastest computer of the top500 list [134] and makes use of the TOFU interconnect, which can be used to build direct networks.

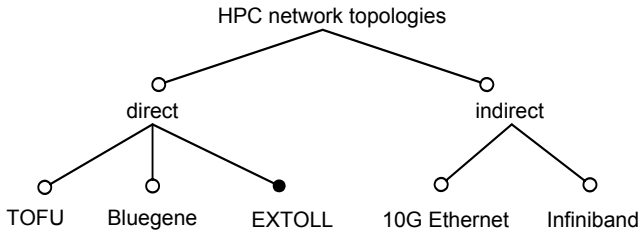


Figure 1-7: HPC Network Topologies

In direct networks the nodes are connected with each other. Indirect networks on the other side make use of one or more central switches as depicted in figure 1-8 to connect the nodes.

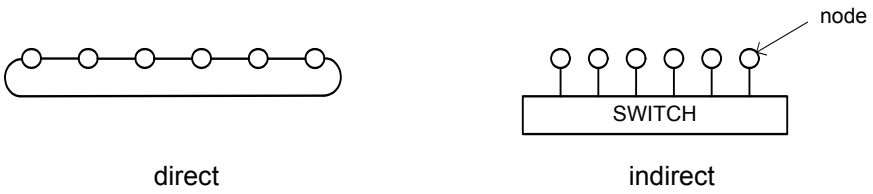


Figure 1-8: Direct and Indirect Networks

The distinguishing feature of direct networks is that in direct networks usually each node contains its own switch. The advantages of direct network are:

- scalability
- cost

Direct networks have in terms of scalability an advantage because the number of switches is growing with the number of nodes in the network.

HPC networks are often based on special ASICs and therefore in a direct network only a single chip is required in all nodes. If an indirect topology is used then NIC chips and switches chips are required. Thus the development and mask costs are reduced when fewer types of chips are used.

Direct networks are characterized by two important numbers: node **degree** and **diameter**. The degree is the number of links from one node to other nodes and the diameter is the maximum number of links that have to be traversed between two nodes.

Most direct networks are n-D tori because this topology has several advantages. The regular structure allows simple routing implementation, and dimension order routing allows deadlock free routing in n-D tori with only two VCs [72].

Furthermore, tori are providing a good relation between degree, number of nodes and the diameter. Kautz graphs as used in the SiCortex systems [140] are even better in this regard, but rely for the small degree on unidirectional links and require a setup of the links that is by far not as regular as for a torus.

Figure 1-9 shows a 2D torus with 16 nodes, with a degree of four and also a diameter of four. A mesh would not have the wrap-around links and therefore a diameter of six.

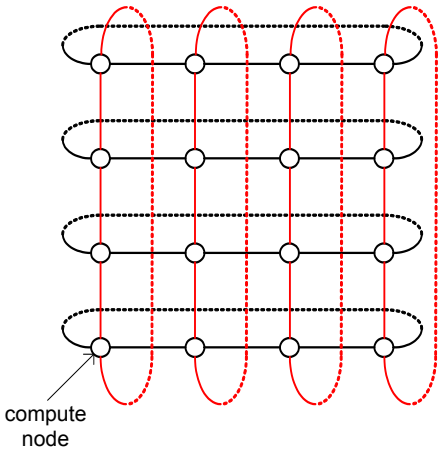


Figure 1-9: 2D Torus

More details about interconnection networks can be found in [100].

The EXTOLL network architecture is flexible. EXTOLL uses table based routing and is not limited to specific topologies. Therefore, every network topology, which can be built with six links, is possible. For instance, a 6D hypercube can be built with EXTOLL. However, for a high number of nodes the best fitting topology is a 3D torus.

Network Interface Controller Architecture

The architecture of the EXTOLL Network Interface Controller (NIC) follows the design principle as outlined earlier. It consists of the hardware independent core_top_file that is instantiated in a hardware dependent top_file. Figure 1-10 shows an overview of the design, which can be divided in three main functional parts:

- host interface
- network interface
- network

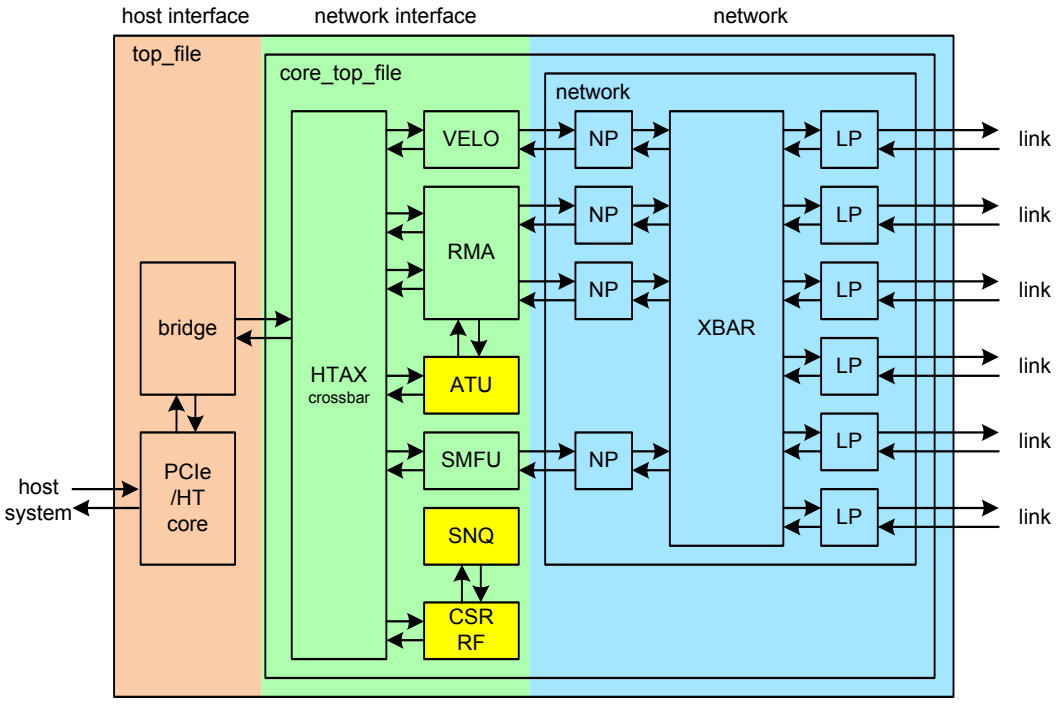


Figure 1-10: EXTOLL NIC

The **host interface** provides the connection between the internal HTAX crossbar that makes use of the HToC protocol and the host system, while the network is responsible for the transport and the routing of the **network** packets. Consequently the **network interface** is translating between the host system side and the packets in the network.

The yellow units (ATU, SNQ and CSR RF) are described in this thesis.

VELO

This is a unit specialized for small messages and it is an abbreviation for **V**irtualized **E**ngine for **L**ow **O**verhead [89].

VELO makes use of user level access to the hardware to avoid using the operating system kernel for communication. By avoiding the kernel it is possible to reduce the latency, because no context switch is necessary.

More details about VELO are presented in [114]. For bigger data transfers the Remote Memory Access (RMA) unit should be used. The exact size when RMA allows higher data rates than VELO depends on the technology on which EXTOLL is implemented.

The lower latency property of VELO as compared to the RMA is mostly reached by the usage of PIO instead of DMA, because a full round trip between device and main memory controller can be saved.

RMA

The RMA unit [25] allows direct access to remote memory and is designated to be used for larger data transfers within the EXTOLL network architecture. This unit provides one-sided communication capabilities for the network, where only the process on one side is actively involved, on the other side the task is handled by the RMA unit in the NIC.

It supports put and get operations. The get operation allows reading from the main memory of another node. In figure 1-11 node0 is sending a get request to node1 for data in the memory of node1 and the RMA unit of node1 returns this data to node0.

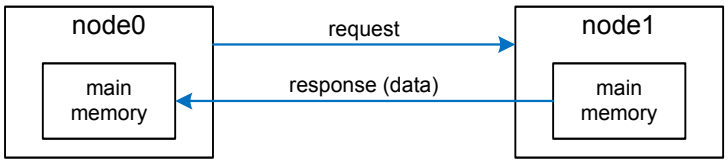


Figure 1-11: RMA Get

The put operation writes directly into the main memory of the destination node as depicted in figure 1-12.

Figure 1-12: RMA Put

However, neither put nor get operations can just operate on arbitrary memory areas, because this would be a security problem. Access to memory is secured by the Address Translation Unit (ATU). Only memory that is registered with this mechanism can be used for RMA operations.

The get and the put operations and their interaction with the address translation unit will be detailed in chapter 4.3.

Get and put operations are by default without completion information and therefore the software that makes use of the RMA does not know when an operation is finished. Thus, RMA offers the possibility to activate a feature that will cause a notification after an operation has been finished. This notification is written into a queue of the corresponding process that caused the operation. It is optionally also possible to trigger an interrupt.

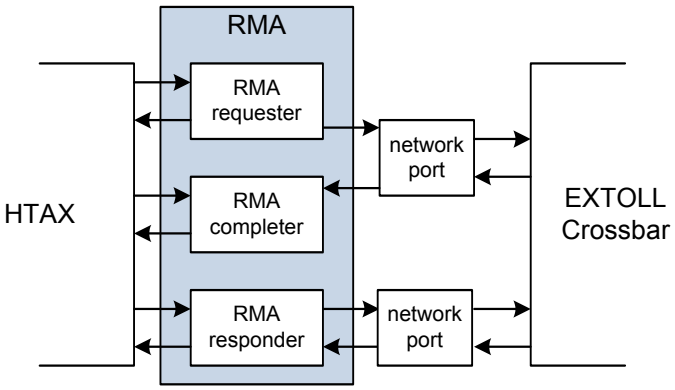


Figure 1-13: RMA: Three Units

The RMA consists of three units as depicted in figure 1-13. The requester sends data (put) and data requests (get) to the network, while the completer writes received data into the main memory. The task of the responder is to answer data requests by sending the requested data.

1.3.2 Trading Accelerator

The second application specific hardware structure is a trading accelerator. Trading at stock exchanges is done today mainly automatic, based on different algorithms. A variant of automated trading is called high frequency trading (HFT) with the following significant property: It has to be fast.

HFT is highly competitive, therefore application specific hardware structures have a place in this field, only the fastest trader gets the trade and can therefore earn money. The win with each trade does not have to be big, because the trades are often and numerous.

In the pursuit to lower the latency of such an application special hardware was designed. A standard setup for a trading system is depicted in figure 1-14, the stock exchange sends the market updates in UDP multicast streams to the NIC of the trading server, and this NIC forwards the received data to the software that runs on the server. Based on the market information the software does decide if the price is an opportunity.

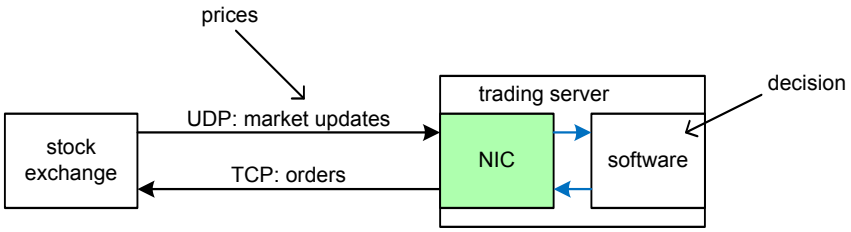


Figure 1-14: Trading Setup

An opportunity exists if it is monetarily worth to buy or sell based on the current market price. The decision is based on different strategies, but this is not an issue this work is concerned with. Based on this decision an order is sent over TCP/IP.

This hardware design tries to lower the latency of the NIC and of the communication between the software and the NIC.

Register File Generation

Chapter 2

2.1 Introduction

The introduction will start with an explanation of what a register file is, why such a register file should be automatically generated and will continue with outlining the problems that have to be solved for the efficient implementation of register files.

2.1.1 Register File

Complex hardware designs have many configuration settings and control options that can or have to be adjusted in order to be able to use the hardware. All these settings have to be stored in the hardware and there have to be ways to change them from outside of the hardware. Furthermore, also methods are required to read out these settings and further status information that may be provided by the hardware.

Such information is generally stored in registers, which are called CSR for Control and Status Registers. A CSR can be accessed typically with the help of a host interface. Figure 2-1 shows a CSR, which is also directly connected to logic.

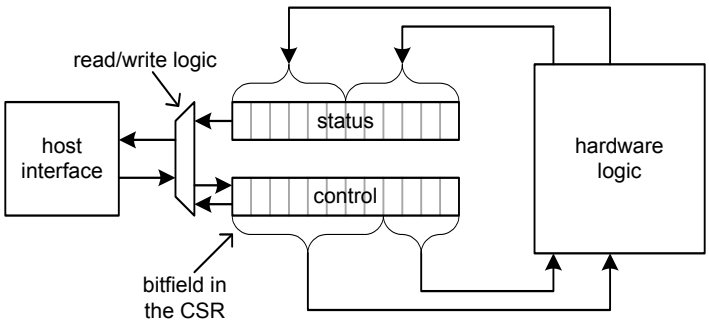


Figure 2-1: Control and Status Register

Concerning the naming convention there are two slightly different areas that make use of the term register file:

- CPUs
- HW devices

Register files are sets of registers that can be addressed for read and write operations. The term register file is often used for one of the basic building blocks of **CPUs**, operands for instructions are read from the register file and the results are written to the register file.

Figure 2-2 shows the schematic of a 4x4 register file that is able to store overall 16 bits. It has only one write and one read port. Register files in CPUs usually have multiple read and write ports. The register file is organized in a regular way, packed as densely as possible. Thus, there is no direct connection to discrete logic like in the case of a CSR.

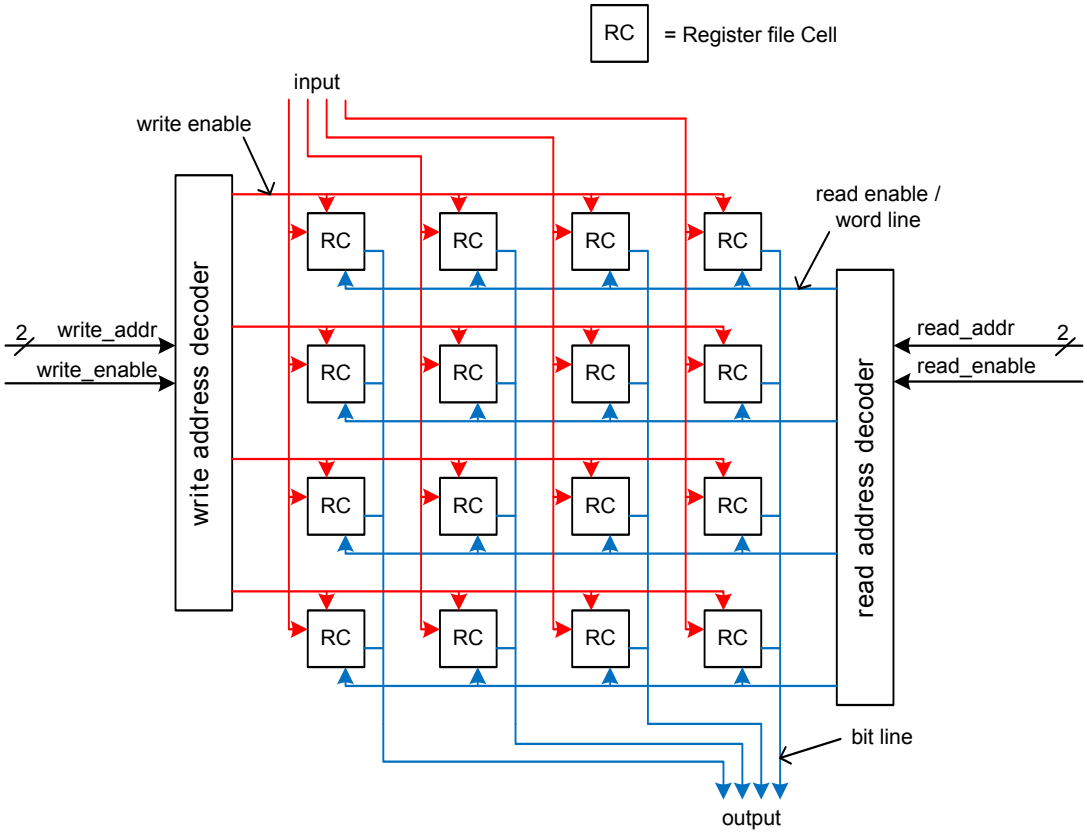


Figure 2-2: Register File

The bit lines are connected to all register file cells (RC), but the outputs of the RCs are only active when the corresponding word line is active.

To be able to reach high performance it is important that register files can perform multiple reads and writes at the same time. Therefore, research is conducted in that area. The journal article [103] presents the implementation of an eight port register file and shows the type of structures that are employed for these.

Register files in the CPU context can not be used to control hardware directly via control signals, because there is no way to either connect control or status wires to the internal storage structure. Such structures can also be used in place of small SRAM IP blocks in ASICs.

In the context of this work a register file (RF) is describing a **collection of multiple CSRs** and similar structures that will be introduced in this chapter. Most hardware devices provide such a facility, for example the 82559ER Ethernet chip from Intel [44], this can be considered to be a generic example.

There is usually only a single read/write interface that is using addresses as shown in figure 2-3. The control information from the RF is provided by the means of output registers of the RF module. Status information is taken directly from inputs.

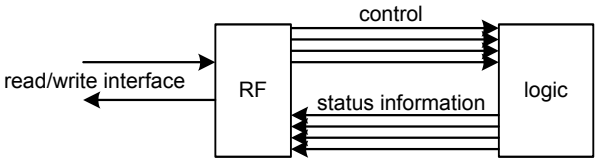


Figure 2-3: RF CSR Functionality

The following example code will demonstrate this by implementing the CSR from figure 2-1.

```
module example_rf(
     input wire res_n,
     input wire clk,
     input wire [2:0] address,
     output reg [63:0] read_data,
     input wire read_en,
     input wire write_en,
     input wire [63:0] write_data,
     output reg [63:0] control,
     input wire [63:0] status
);
always @(posedge clk or negedge res_n) begin
     if(!res_n)
          control <= 64'b0;
     else
     begin
          if((address[2:0] == 3'h0) && write_en) // (b)
               control <= write_data;

          case({read_en,address}) // (a)
               {1'b1,3'h0}: read_data <= control;
```

```
                {1'b1,3'h1}: read_data <= status;
            endcase
        end
    end
    endmodule
```

The example_rf Verilog HDL [83] module shows the simplest complete implementation of a RF that is possible. It has one control register **control** and one status register **status**. Both registers can be read due to the case statement **(a)**, but only the control can be written **(b)**. To implement this, a very simple read/write interface is implemented with *address*, *read_data*, *read_en*, *write_en* and *write_data*.

In the usual case a RF appears to be a linear address space like shown in figure 2-4 with many registers, which can be read and/or written by the software. In most cases the RF is mapped into the address space of the host system. Many hardware development projects of a certain size are combinations of special hardware like an FPGA board and a standard off the shelf computer system. Examples for such systems are the two research projects EXTOLL and the trading accelerator, which were introduced earlier.

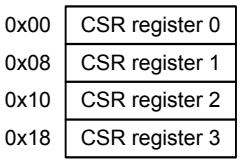


Figure 2-4: RF Address Map

The hardware designs as described in chapter 1.3 have the RF as central element.

2.1.2 Auto Generation

Often the hardware development workflow for generating a RF and the other necessary parts consists of the following manual steps:

1. Writing the specification
2. Implementation of the registers in a hardware description language (HDL)
3. Writing the documentation
4. Using the documentation to write the verification and simulation environment
5. Using the documentation to write drivers and software for the hardware

All these steps can be done manually and are in fact often done manually, because the code to read and write registers is very simple as shown in the code example. Wilson Snyder demonstrates in [22] methods to organize Verilog HDL code for this purpose, but suggests in the course of the paper to automate the task.

The disadvantage inherited by doing this manually can be big, because of the probability for errors. Obviously, every single change in any of these manually written parts has to be propagated into all the others. The resulting problem is depicted in figure 2-5; blocks with dashed lines are not directly related to the RF, but important for the explanation. The HDL is used to generate a bit file for the FPGA and the FPGA is used together with the generated bit file to test the overall design. It is not necessary to be a doomsayer to foresee that at some point a developer will modify the HDL and also the software, but of course neither the specification nor the documentation will be updated.

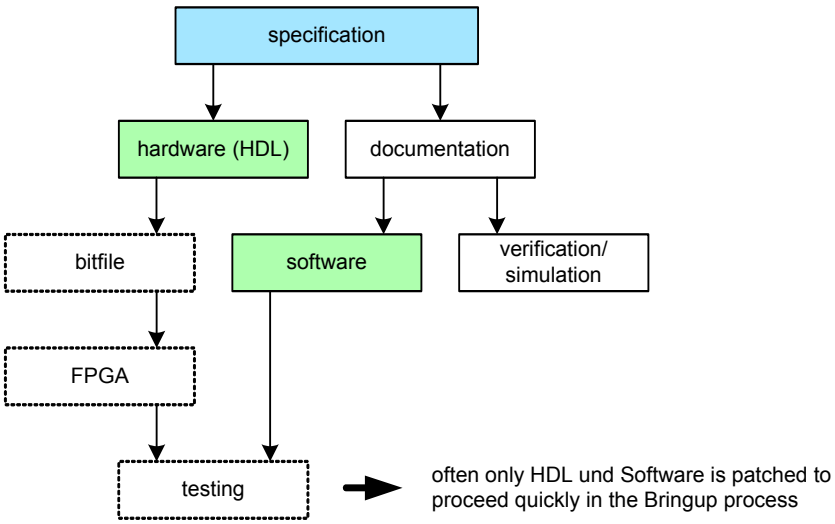


Figure 2-5: RF Workflow

Every change should start at the level of the specification. Therefore as much as possible should be automatically generated from the specification to avoid human errors and save development time.

The target has to be the generation of all code and documentation from a single source with a high abstraction level.

To point out a further reason why manual writing of the HDL is not a good idea an example should be given. The automatically generated code of the EXTOLL RF in the ASIC version consists, as of May 2012, of 52607 lines of Verilog. Even under the assumption that the generated code is ambiguous and hand written code would only be half the size, it is a lot. Thus, automatic generation is desirable to reduce the probability of errors. The EXTOLL RF will also be used in this chapter as a benchmark for the design and implementation decisions.

The typical environment as introduced in chapter 1.2 consists of a standard computer system with an FPGA on a PCI express (PCIe) [43] board. To be able to make use of the lower latency properties of the HyperTransport (HT) [63] protocol, it is also possible to use the

HTX interface [41]. In the Computer Architecture Group [40] this is an often chosen technology. For the software there is no difference between PCIe and HT, despite the lower latency of HT.

However, it is also possible to access the RF by other means. An example for this is the debug port as described in chapter 2.6.2. Furthermore, it is also possible in the EXTOLL design to access remote RFs with the help of the RMA units. These were introduced in chapter 1.3.1.

Figure 2-6 shows a typical setup consisting of a computer system running software, which is able to perform reads and writes on the RF that is placed in an FPGA or an ASIC.

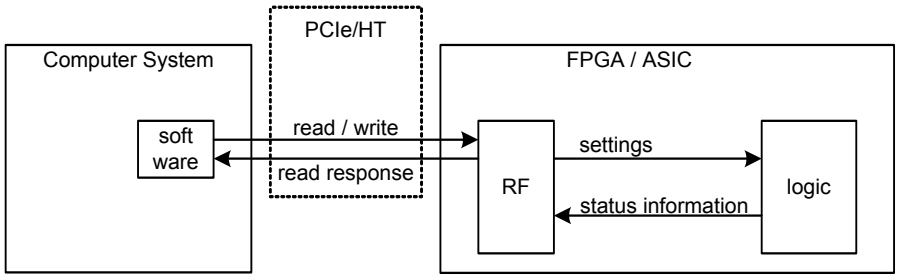


Figure 2-6: RF Environment

At first the problem of automatically generating a RF for such an environment seems to be simple, because it is not very complex to generate the logic that allows writing registers from the software and being able to read them out again. In the end there are issues that are complicating this task:

- hardware control fields within CSRs
- hardware structures different to CSRs are required
- timing constrains
- counters

CSRs are the most important elements in a RF, but there are cases where other structures can be used more efficiently. When no concurrent access to information is required SRAMs can be used to store and retrieve information.

An example for such information is the routing table in a network switch, given that each incoming port has its own table, only one lookup has to be done concurrently, because there can be only one new packet at a time. In the Xilinx Virtex 6 FPGAs each block RAM is 36 kbit in size and can be used for example 64 bit wide and 512 entries deep. This amount of storage in flip-flops would consume a big portion of the FPGA. When flip-flops are used to

store 64*512 bits then this requires 32768 flip-flops, the Virtex6 xc6vlx240t FPGA that was used for development has 301440, so 10.8% of the FPGA would be used. In contrast, the FPGA has 416 block RAMs [10].

Therefore, efficient ways to integrate block RAMs into the RF structure are required to be able to make use of this resource.

Counters are another issue, and can require a lot of space in FPGAs when implemented in standard logic. The usage of special components is discussed and described in chapter 2.4.2.

2.1.3 Hierarchical Decomposition

Reaching **timing** closure is a very big concern in all hardware development projects. Thus, the RF also has to be considered and methods have to be found so that the RF does not negatively impact the maximum frequency.

To demonstrate the problems that have to be solved when big designs are implemented an experiment was conducted for a recent FPGA. Figure 2-7 shows a Xilinx Virtex 7 (xc7vx485t in speed grade 1) [71] with a simple test design. The research group already had an engineering sample of this particular FPGA. Therefore, its performance characteristics were interesting. The test design consists of four registers of the type FDPEs (D Flip-Flop with Clock Enable and Asynchronous Preset) [102] named r1, r2, r3 and r4, which are assigned in ascending order. For this experiment the FDPEs had to be instantiated explicitly, because when normal Verilog registers were used the synthesis detected a shift register and instantiated a single special component instead, rendering the test design useless. The FDPEs were placed in three corners of the FPGA so that the horizontal, vertical and diagonal routing delay could be observed.

Despite the red lines are straight this does not mean that the real routing in the FPGA will follow these, the lines indicate only the linear distance in the FPGA. In the end only four of the 607200 registers are occupied in this evaluation. Given that the FPGA is basically empty it can be safely assumed that these delays are the lower limit.

The result of this experiment is that on modern FPGAs it is not possible to cross the whole chip without intermediate registers, if decent frequencies are targeted. In this example the maximum frequency for logic that uses r2 and r3 without registers in between would be 89.5 MHz.

With a target frequency of 200 MHz it is obvious that it is a severe problem to use only a single RF in such a big FPGA. In figure 2-8 this is illustrated. There is a single RF in the middle of the chip and blocks of logic in the corners of the FPGA. The RF was placed in the

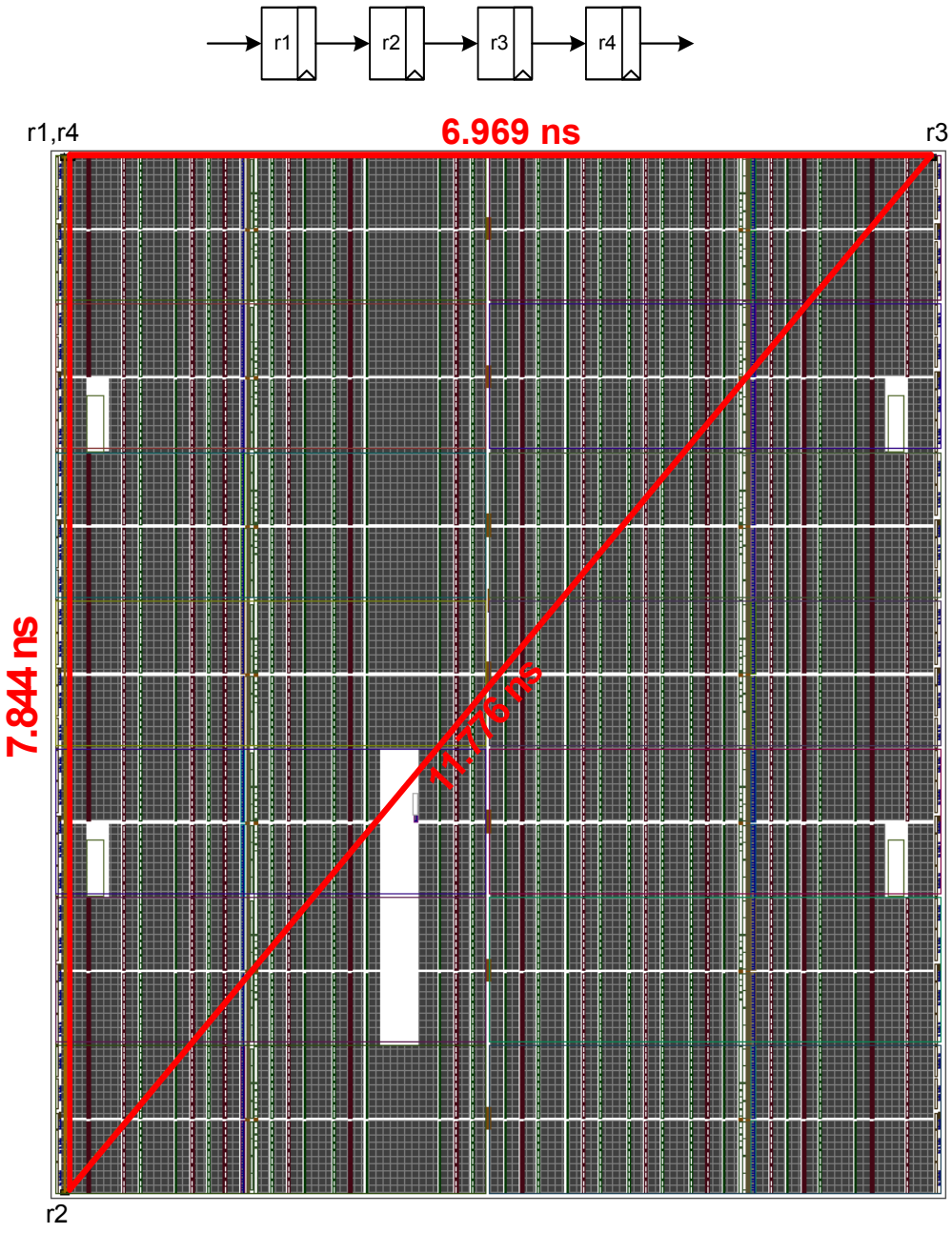


Figure 2-7: FPGA Delay Evaluation

middle because this way the distance to the rest of the FPGA is the shortest. It is only an example to visualize the problem. A real design would of course use more than just the corners of the whole FPGA for logic.

Assuming the Virtex 7 FPGA the routing delay to the corners is about half of the diagonal delay of 11.776 ns and therefore 5.9 ns to the corners of the FPGA. This delay of 5.9 ns is more than the 5 ns, which would be required to reach timing closure at 200 MHz.

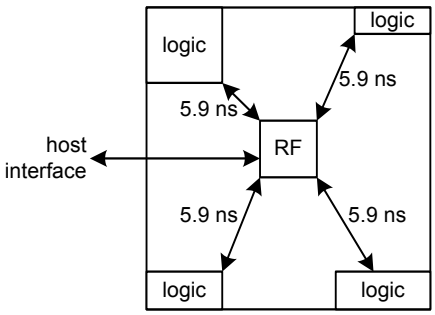


Figure 2-8: Delay Visualization with RF in the Middle

Additionally, the Virtex 7 xc7vx485t FPGA, which was used for this evaluation, is one of the smaller chips of the Virtex 7 series. The biggest Virtex 7 FPGAs will be four times bigger and will therefore have even higher routing delays.

The result is that it is impossible to build a design that is controlled by a single RF in a naive way, when the design is supposed to run at an adequate clock frequency and uses a big part of the FPGA.

Closing the gap between the size of current hardware devices and the too high routing delay on the other side is therefore mandatory. There are the following options:

- pipelining the data paths
- multi cycle path constraints
- hierarchical decomposition of the RF

Adding extra registers into the path between RF and the logic is one possibility to relax the timing by pipelining. It requires extra resources and increases latency.

Multi cycle path constraints override the normal timing constraints that exist within a clock domain. By using them it is possible to allow the place and route tools to use more time to propagate signals, but multi cycle paths are not without issues.

For example, timing closure could be reached if all connections from and to the register file would be constrained to 15 ns. Assuming that there is a two bit wide data path between the RF and the logic and the clock has a cycle time of 5 ns. It can happen that the delay is in the end different on data[0] and data[1] as depicted in figure 2-9. In this case data[0] would arrive a full cycle before data[1], because the delays are in this example 7 ns and 12 ns and have therefore a difference of a full cycle. Thus, if the data output of the RF was 2’b00 and is set to 2’b11 by the RF, then the logic would see 2’b01 for one cycle before it could sense the correct 2’b11 as shown in the timing diagram.

As a result, it is only possible to use multi cycle constraints under certain conditions.

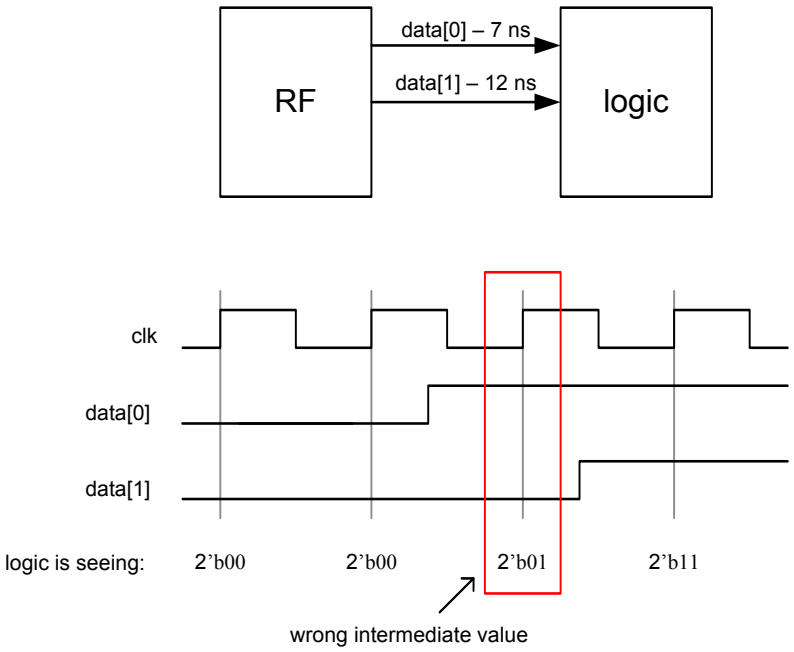


Figure 2-9: Multi Cycle

It has to be ensured by design that intermediate values are not used. This can be done by using extra control signals, which take care that the value is not used too early. One application for multi cycle constrains could be a unit with several control inputs and a single enable signal. The unit would use the control inputs only when it is enabled. Then the hardware design only has to ensure that the enable signal arrives after the maximum delay of the control signals. The enable signal could be pipelined with the right number of stages for the purpose.

The third possibility is hierarchical decomposition by dividing the RF into different parts. These parts can be placed within the logic that makes use of the RF. By placing these parts of the RF closer to the logic the routing delay is less of an issue. This solution is depicted in figure 2-10. Timing on the connection between the main RF and the sub-RFs can be relaxed by pipelining. This will be detailed in chapter 2.8.

The following are the minimum requirements for a RF generator:

- input language with high abstraction level
- logical naming
- efficient generated code
- human readable generated code
- automatic documentation generation

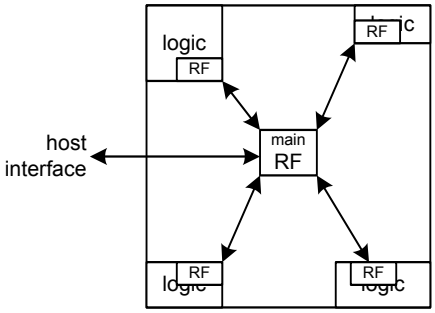


Figure 2-10: Hierarchical Decomposition

The **input language** is required to have an abstraction level that is as high as possible, but should allow controlling details of the generated RFs. Furthermore, the input language should be easy to use.

Developers should not have to be concerned about physical address handling. Only **logical naming** must be used. On the other side the possibility to place RF components at certain addresses has to be available, because it is required in some cases.

The HDL code should be generated in a way so that **efficient hardware** can be synthesized from it. It should be efficient in terms of area and timing. Moreover, the resulting code should be **human readable** so that it can be understood easily for the purpose of debugging.

Documentation should be generated automatically, so that it is easy to get an overview of the generated RF. Software developers should have no problems to extract the necessary information to control the hardware.

The following chapter is destined to describe the problems and design choices that are faced in the design phase of a tool to generate RFs, which can be used in different complex hardware designs. The first section will give a short overview over the range of available products. Furthermore the reasons will be justified that lead to the development of a new RF generation tool. This tool is named RFS.

2.2 State of the Art and Motivation

The generation of a register file is neither a new nor a very special problem. There are multiple products available that try to solve the problem. In the following the currently available products are described and the reasons are explored why an own tool was developed.

2.2.1 Denali Blueprint

This tool [31] can be used to generate RFs. Different semiconductor vendors are using it, as it can be seen in the 2008 press releases from Atheros [45] and Cypress [53].

Denali Blueprint is using SystemRDL [46] as primary input format. SystemRDL is a special language for register files, RDL stands for Register Definition Language. An example of this language is given below. Recently SystemRDL evolved into an industry standard.

Blueprint is also supporting Spirit IP-XACT XML [95] as input language by using an extra spirit2rdl converter program.

Blueprint is able to generate various output files that can be used for RFs. The following list names the most important ones:

- Spirit IP-XACT XML (a description can be found in chapter 2.3.2)
- C code for firmware
- RTL code for synthesis
- documentation

The following example is presented to give an impression of SystemRDL. It demonstrates the relative complex and extensive syntax.

```
reg test_reg1 {
    name       = "TestRegister1";
    desc       = "Description of TestRegister1";
    regwidth= 64;
    field {
         name       = "TestField2";
         desc       = "Description of TestField2";
         hw         = w;
         sw         = rw;
         fieldwidth = 32;
         reset      = 32'h0;
         resetsignal= res_n;
    } test0;
    field {
         name       = "TestField3";
         desc       = "Description of TestField2";
         hw         = w;
         sw         = rw;
         fieldwidth = 32;
         reset      = 32'h0;
         resetsignal= res_n;
    } test1;
};
reg test_reg2 {
    name       = "TestRegister2";
...
};

test_reg1           reg0 @0x00; // address map
test_reg2           reg1 @0x08; // address map
...
```

The example shows a fragment from a RF, in this fragment a register **test_reg1** is defined that contains two 32 bit fields for testing purposes. It is for example possible to configure if the software has read-write access to the register. A second register is named **test_reg2**. Both registers are placed into the address map at the end of the example at the addresses 0x00 and 0x08.

As it could be seen in this example the definition of RF components is separated from the actual placement in the address map. It is possible to use defined **reg**s multiple times, but this is of course only helpful when exactly the same **reg** is required again. The disadvantage is that each register has to be created in two places.

Denali Blueprint was used in different research projects of the Computer Architecture Group before RFS was developed, especially EXTOLL R1 made extensive use of it. With these projects the limit of Blueprint were explored. The most obvious limit was that it was not possible to reach timing closure for EXTOLL R1 with all required registers in the RF, because of two reasons:

- logic complexity
- placement constraints

In the read path for the software the different inputs have to be multiplexed to the output and the more inputs a multiplexer has, the slower it will be.

Furthermore, when there are more registers in a single RF this implicates that they will be used as control and status register for more logic. Consequently, this logic will be distributed over a larger area of the FPGA/ASIC. The result will be a higher routing delay from the registers to the multiplexer and more timing problems.

There is no native support for the hierarchical partitioning of register files in Blueprint, but partitioning was necessary in EXTOLL R1, because otherwise it was not be possible to reach timing closure.

Therefore, Perl [80] scripts were developed to split up the RF in different sub-RFs. Blueprints was used to generate a RF from each of these sub-RFs and afterwards the script combined the different generated RFs into one big RF. Unfortunately this process proofed to be error prone. The generated C header files that are used to access registers had issues that resulted in erroneous addressing. For this reason manual checks and fixes of these files got necessary.

Despite it is possible to change parts of the code generation of Denali with Perl, this approach was not followed further, because significant efforts would have to be invested into learning how to do this.

One of the biggest issues of Blueprint is the SystemRDL description language, because it is complex and not intuitive.

2.2.2 Others

There are multiple other solutions, and the most prominent ones should be mentioned in the following.

Duolog Technologies **Socrates Bitwise** [34] is a product similar to Denali Blueprint, but it also provides a GUI.

Vregs is a free tool that is available from [23]. It uses HTML files, which can be produced with graphical editors like Microsoft Word or Adobe FrameMaker, as input language. No native support for hierarchical RF designs is implemented.

Agnisys **IDesignSpec** [109] provides plug-ins for multiple versions of Microsoft Word, Microsoft Excel, OpenOffice and Adobe FrameMaker. RFs can be specified interactive with these plug-ins, consistency checks can be performed directly.

PDTi **SpectraReg** [108] is a product that is used in a web browser. It can be licensed in an online version and it also can be licensed in a version that can be installed on site. The product provides features similar to Denali, though there are no details on the website.

For completeness, there are at least two further products available: CSRCompiler [111] and GenSys Registers [110], but they have no outstanding features.

2.2.3 Motivation

In prior projects Denali Blueprint was used, but it has weaknesses that motivated the search for alternatives.

Denali provides free licenses for academic uses, but it is unclear if the situation persists and furthermore projects that are not academic will require a license. The licensing fees for Blueprint are high because it has to be licensed in a bundle with other tools.

Native support for hierarchical RFs is necessary for timing reasons, but with Blueprint this was not directly possible. In EXTOLL R1 a lot of resources were used up by the counters that were implemented by Blueprint because they did not make use of special FPGA components.

Consequently, it was desirable to find an alternative and I decided to develop a better alternative.

The most important argument that motivated the development of RFS was the prospect that this step would also allow the implementation of new features and the minor one to save licensing costs.

2.3 Design Decisions

In a first step a small prototype of a RF generator was developed to explore the complexity of the undertaking. This first prototype generator was used for a design that has low requirement regarding the RF. This hardware design is a high-speed flash controller and needs a register file with only eleven CSRs and does not require any special features. The result of this experiment was a positive assertion regarding the feasibility of such a development.

Obviously, in the course of the development of a RFS new requirements did arise and the complexity grew.

A general principle of this code generator is that the output code is supposed to be as human readable as possible, to allow code reviews and debugging, because the possibility of errors can not be denied.

All operations in the RF are designed to be 64 bit wide because that is the natural size for current high performance processors. Some important design decisions are explained in the following.

2.3.1 Features

RFS has to support several features to be usable for the generation of the RFs for projects that are as complex as the EXTOLL project:

- different types of registers
- different access rights
- special purpose registers
- memory type registers in SRAM
- easy replication of structures
- hierarchical organization
- capture registers

Different types of CSR registers are the main feature and it must be possible to adjust certain properties of these registers. It has to be possible to configure if a register is readable, writeable or both. If a register can be limited in this regard, then logic can be saved.

It has to be possible to make use of special hardware components in FPGAs to implement counters. The purpose of this is to save resources.

SRAMs can be used to replace multiple registers of the same type in a RF. This allows saving resources when the hardware application can make use of an SRAM interface. The flexible interface makes it easily possible to connect other units to the RF. An error logging unit that can be connected to an external SRAM interface will be demonstrated in chapter 2.4.5.

The possibility to replicate a set of registers and SRAMs is very important in complex RFs. With such a feature the description of the RF is shorter and provides therefore a better overview. The main reason for such a feature is that this makes it possible to provide an array of C structs [94] in the C header files that describes a repeated set of registers and SRAMs. As a result they can be accessed in an efficient way from the software. Furthermore, this allows parameterizing the number of instantiations for this particular block of registers and SRAMs.

It has to be possible to instantiate sub-RFs due to the strong requirement to be able to compose the RF hierarchically.

Already before the development was started an important desired feature was the possibility to set watch points for certain registers, so that the software could be informed in an efficient way about changed registers. This feature consists of auto generated logic in the RF and an additional unit that will be introduced in the next chapter.

2.3.2 Input Method and Data Format

A way to provide the high level abstract input for RFS had to be found, the first design decision was between GUI and no-GUI. Developing a GUI for design entry would require a lot of effort, because there are so many different options available, a GUI would have to provide them all. Furthermore, it is not even clear if a GUI would save the user any time. In a good text format it is easy to copy, paste and do some small modifications to get the required results. In a GUI this can take much longer.

Accordingly, the decision was made to use a text based format as primary entry format. This does of course not prevent the development of a GUI that outputs this format.

The text format has to fulfill multiple requirements. It should be simple to understand, and users should not have to spend a lot of time to get acquainted to the description language. Furthermore, it should not require verbose descriptions, only what RFS can not do itself should be required to be defined. In the end also readability should be given, so that it is easy to understand and modify an existing description.

First existing description languages that can be used to describe RFs are going to be described and reasons are given why they are not suitable. There are two existing description languages and both are “industry standards”. Using an existing format would have the advantage that there are already tools that could be used.

SystemRDL

It is used by Denali Blueprint. This format [46] is not XML based and has a complex syntax. Writing a parser for this format would have caused a lot of development work. An example is shown in chapter 2.2.1.

Spirit IP-XACT XML

Spirit IP-XACT XML is a recent “industry standard”. At first it seems like a viable possibility. The standard is based on XML.

It has to be mentioned that the IP-XACT standard [95] can describe more than RFs, it can also be used to describe the connections between different hardware blocks [115]. One of the advantages of an industry standard is that it allows using other tools on the same definition, for example a GUI or a generator for verification components.

The following example from [96] should give an impression how the specification of a single register looks like in IP-XACT.

```
<spirit:register>
    <spirit:name>reg1</spirit:name>
    <spirit:description>Description of reg1</spirit:description>
    <spirit:addressOffset>0x0</spirit:addressOffset>
    <spirit:size>32</spirit:size>
    <spirit:access>read-write</spirit:access>
    <spirit:reset>
        <spirit:value>0x00000000</spirit:value>
        <spirit:mask>0xffffffff</spirit:mask>
    </spirit:reset>
</spirit:register>
```

Especially the existing verification components were interesting, because with OVM_RGM [101] there already has been a software package that can make use of IP-XACT specifications to create components for the verification environment. The problem is that OVM_RGM supported only IP-XACT version 1.4, but one of the desired features of the description language was the possibility to instantiate multiple register objects at once, like with the repeat blocks that will be shown later. This feature is not supported in IP-XACT 1.4, because the first version that defines it is version 1.5.

Thus, it was decided not to use IP-XACT, because a decision would have to be made between version 1.4 and 1.5, and in both cases compromises would have to be made. Furthermore, the often use of the word “spirit” in the definition is a nuisance.

In the meantime since this decision IP-XACT was standardized by the IEEE [95] and version 1.5 is not the most recent anymore. Over the time there were multiple different versions of the standard [113].

Own Description Language

Because neither SystemRDL nor IP-XACT did seem to fit I decided to create an own description language. The first decision to be made was which base format should be used.

- Java Script Object Notation (JSON) [32]: Is an open standard for human readable data representation and can serve as alternative to XML. Parsers for JSON are available for numerous programming languages [33].
- Completely self defined: Requires a lot of effort with no apparent advantage.
- XML [4]: Very good support available due to the existence of different libraries, which are able to handle XML.

For reasons of simplicity it was decided that the data format should be XML based, because existing XML parsing infrastructure can be used.

Additionally, it requires the smallest learning effort for the users that are going to use the tool, because most of them are already acquainted with the XML format.

2.3.3 Output Data Formats

To be able to generate hardware, documentation and other required files RFS has to generate multiple output files. First a decision had to be made, which output files were required and which formats should be used, given there is a choice. The most important output format is of course one that allows implementing the actual hardware and also documentation is a minimum requirement.

Hardware Description Language

The foremost purpose of RFS is to generate hardware, a HDL languages has to be used for this purpose. The two obvious choices are: Verilog and VHDL [91]. These two languages can be used to synthesize hardware, at least as long as only the synthesizable subsets are used. Both support features that can not be synthesized to hardware by standard synthesis software, for example text output for debugging is such a feature.

Of course, there are also other HDL languages that can be used to generate hardware. In the following the most important possibilities are given and their fitness to the problem is discussed:

- SystemVerilog
- SystemC
- MyHDL
- VHDL
- Verilog

SystemVerilog [54] is a language that can be used as HDL and verification language. It is close to Verilog. A very simple subset can be synthesized. This subset would already offer very good advantages. For instance, it is possible to define **packed structs** instead of only wires in Verilog. This allows to connecting modules with fewer code. Thus, lots of code lines and possibilities for errors can be avoided.

Unfortunately SystemVerilog can not be used, because Xilinx XST does not support SystemVerilog. Xilinx XST is the synthesis software for Xilinx FPGAs and it is required, because these are the primary target platform.

SystemC [65] is a library for C++ and provides an event driven simulation kernel. It is not possible to directly synthesize SystemC with normal tools. Special software exists that claims to be able to synthesize it or produce synthesizable RTL from SystemC. Catapult C [66] [67] is such a software.

MyHDL [47] is a relatively young development, with the idea to use Python as HDL language. It can be used for the hardware description and as verification language. Standard Electronic Design Automation (EDA) tools are not able to use MyHDL directly, but tools to translate MyHDL to Verilog or VHDL have to be used.

Therefore, in the end only **Verilog** and **VHDL** were left over. However, the language of choice in the Computer Architecture Group is Verilog and therefore it was chosen at the end as output language. The possibility to output also VHDL was not implemented because there was no direct need for it. To use RFS in a VHDL environment there are multiple possibilities:

- VHDL output can be added to RFS, which is an obvious possibility.
- Software that converts Verilog to VHDL can be used. For example the two companies SynaptiCAD [55] and X-Tek Corporation [56] are offering products for that purpose.
- Designs that are a mixture of VHDL and Verilog code are supported in some EDA tools. Xilinx XST [70] is one of these.

Documentation

Human readable documentation is very important for the usability of the generated RFs. Documentation is required by the software developers that make use of hardware. The most important generated information is the address of all elements in the RF, so that they can also be addressed manually and not only by using the C header files or the sysfs file system. The target was to get two different types of output. First, HTML is a requirement so that it can be viewed easily in a browser; this output should also reflect the hierarchical structure

of the RF. Second, a PDF output or a way to generate a single PDF file that contains all information is required. As a result this feature can be used to generate the register specification documentation for the hardware.

Implementing a direct output of these formats from RFS would be possible of course, but for the purpose of documentation generation there are different special intermediate formats and tools available that can save a lot of development effort:

- Doxygen
- DocBook
- RST

Doxygen [57] is a tool that reads special formatted comments from source code and uses this information to generate documentation. Numerous languages are supported directly. More languages are supported with the help of filters. For Verilog there exists such a filter. However, no documentation for the Verilog code is required, but only a documentation of the RF components.

DocBook [58] [59] is an XML language that is used for technical documentations.

ReS**t**ructured**T**ext (**RST**) [106] is a plaintext format with very simple markup syntax. RST files can be written in a way, so that also the plaintext files are easy to read. To give two examples:

- Sections captions are marked by underlying them with “=” or similar signs. The order in which these signs are used in the RST files assign a heading level to them.
- A bullet list entry can be created by putting a “*” in front of the line.

Relatively little effort it required to generate RST files. The RST output files can be processed by Sphinx [92] to generate either HTML documentation or LaTeX files. These LaTeX files can be use with pdfTeX [107] to generate a PDF file.

Thus, the RST format was chosen because of its simplicity and because it allows to generate HTML and PDF output.

C Header Files

An important functionality of a RF is to be able to use the register file from software. The layout has to be known by the software, and therefore headers files containing ANSI C structs are generated. The RF is mapped into the address space of user level processes and the C structs can be used as an overlay to access the registers in an efficient and convenient manner from C and C++ [73].

Annotated XML

This is a single XML file that is written by RFS, the file contains the complete structure of the RF including the addresses of all elements. The purpose of this file is to provide an output that can be used by other tools. Figure 2-11 shows that there are **rgm** and **rfdrvgen** that make use of this output file. Both tools are described in more detail in chapter 2.6. RFS is generating this file by default with the name of the top level XML file extended by “anot”, so the annotated XML for the extoll_rf.xml is named extoll_rf.anot.xml

The annotated XML does not contain all fields from the original XML files, only the information that is relevant for the software interface is contained. Attributes in the XML file, which are generated by RFS, are prefixed with an underscore.

Miscellaneous Files

rfs_v.h: The Verilog header file is generated by RFS. It contains certain parameters of the RF. These allow describing the connections between different parts of the register file in a generic way, because the widths of all connections are defined in this file. Thus, no manual changes will be required in a top level when only a sub-RF is modified. This file contains all necessary address paths, read path, write path and trigger path widths.

rfs_seconds.h: This file contains the time when RFS was called to generate the RF as define:

```
#ifndef RFS_SECONDS
#define RFS_SECONDS 1326977713
#endif
```

The value is the UNIX standard time [60] is 32 bit in size and counts the number of seconds since 1st Jan. 1970. The application of this value will be detailed in section 2.3.4.

snq_te.txt, **snq_tdg.txt**: These files and their application will be explained in section 2.4.9 and the next chapter.

Flow Overview

There are multiple different output files, some of the files can be directly used, and others have to be processed further. In figure 2-11 an overview is shown. Boxes with round edges represent, in this overview, files or file formats, the rectangular boxes describe programs that process or generate these files.

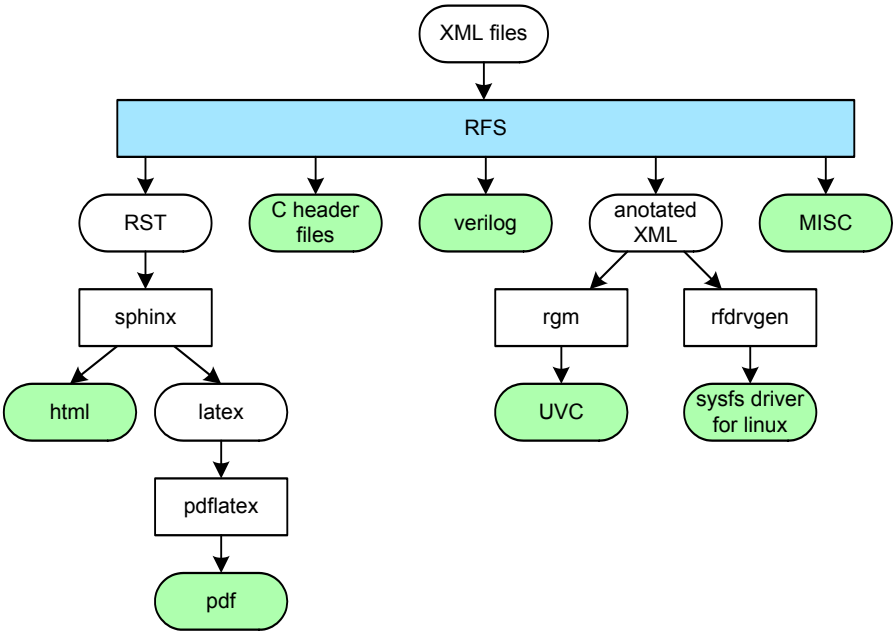


Figure 2-11: Output Files

2.3.4 HW/SW Interface Consistency

The main purpose of the RF is to allow the software to communicate with the hardware. It can make use of different software interfaces for this purpose. All of them are relying on addresses. When the XML specification is changed then this can cause changes in the address setup of the RF.

The usual workflow for bringing up a design in an FPGA is shown in figure 2-12. It illustrates the steps that have to be performed to be able to conduct a test inside the FPGA after changing either the XML specification of the RF or the other Verilog code. As it can be seen in figure 2-12 the C header files are generated depending on the XML input files and they have to match the generated hardware. Using the wrong header files means that probably the wrong registers are used for read or write operations, in the best case this directly leads to an error. But the result could also be a problem, that is hard to find, or a hardware failure. The later could happen, if for example accidently the flash memory of the FPGA board is erased, then it will not work anymore and it will require repair.

There are multiple methods that can be used to enforce that the hardware and the software match each other:

- interface version
- creation time

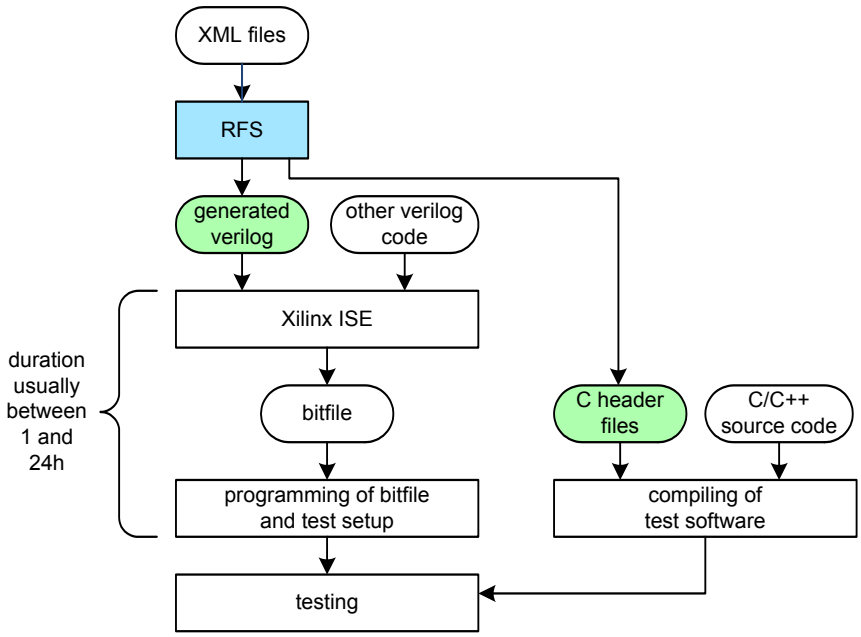


Figure 2-12: Build and Test Flow

- hash value
- include XML

Interface version: A register will carry a version number of the RF interface. Before the software makes use of the hardware it will read that register and compare its content with the expected value. This works in theory of course, but experience has shown that this approach does not work in practice because of the human factor. Very often it was forgotten to change the number. Then the choice is faced if a new bit file should be built with a correct version number or if the risk is accepted that the bit file is used with the wrong software. For testing in the laboratory this is of course not a problem, but if the bit file and software are sent to 3rd parties, then it is more problematic.

Using the version number of the subversion [120] repository is also not a solution, because it does only change when an update or commit was done.

Creation time: To avoid the problem that a human has to change a version number, the UNIX standard time [60] can be used. This is a 32 bit value that contains the number of second since 1st Jan. 1970. The problem that two RFs may have the same creation time is not considered, because it is a very unlikely error condition. This creation time is also stored in the rfs_seconds.h header file so that it can be used by the software to check if the hardware and the software match each other.

Hash value: Another possibility to find out if the hardware matches the software is to generate a hash value for the RF and store it in the hardware so that the software can compare it when it is starting. Examples for adequate hash algorithms are CRC32 [64] and MD5 [61]. The simplest implementation for this would be to generate the hash by using standard UNIX shell tools like tar to put all XML files in a single file and by using md5sum on the resulting file. In the end this would have no advantage over using the creation time, because the creation time will change only if RFS is used to regenerate the RF and this will usually only be done if there are any changes in the XML files. To have an advantage over just using the creation time it would be necessary to create the hash only with the information that describes the parts of the RF that are relevant for the software interface, because this would allow upgrading the hardware without software changes as long as the interface stays the same.

The problem is that this would be relatively complex to either create in RFS a hash directly with exactly only the relevant information or to write another type of XML file with this information to generate the hash value.

Include XML: Another theoretical possibility would be to store the complete annotated XML in hardware, this way all RF related information would be stored inside the hardware and could be read from there. The problem with this approach is the size of the annotated XML file, in the case of EXTOLL it is uncompressed 180 kB and compressed with the very good LZMA [62] algorithms still 16 kB. This would require four 36 kbit block RAMs. The Virtex6 xc6vlx240t FPGA that was used for development has 416 block RAMs [10].

Therefore, the decision has been to make use of the creation time, because it is the easiest method to implement the functionality, and the advantage of the hash solution is not worth the additional effort. Moreover, storing the annotated XML in the hardware is a waste of resources. It also has to be noted that this may require special ROM IP blocks in the case of an ASIC implementation.

When the user is certain that the software interface does fit the hardware despite of differing creation time, then it is always possible to override the check on the software side.

2.3.5 Interface

In the terminology of RFS the RF has a software and a hardware view, the software view is accessible by a bus and the hardware view is accessible by different ways, depending on the generated RF. The interface to the RF is very simple, because there are not many reasonable choices to define it in a different way. It is also very similar to the interface provided by RFs generated with Denali Blueprint. Therefore, it was very simple to replace a RF, generated

by Blueprint, in an existing project with a RF that was generated by RFS. The primary platform for RFS are 64 bit systems, therefore the granularity for addressing the RF is also 64 bit and all accesses to the RF have to happen in the form of 64 bit quadwords.

However, despite the RFs that are generally for 64 bit systems this is only semantics. The actual width of all data paths will be limited by the widest CSR or SRAM. Thus only the data path width that is actually used costs resources.

The interface is depicted in figure 2-13.

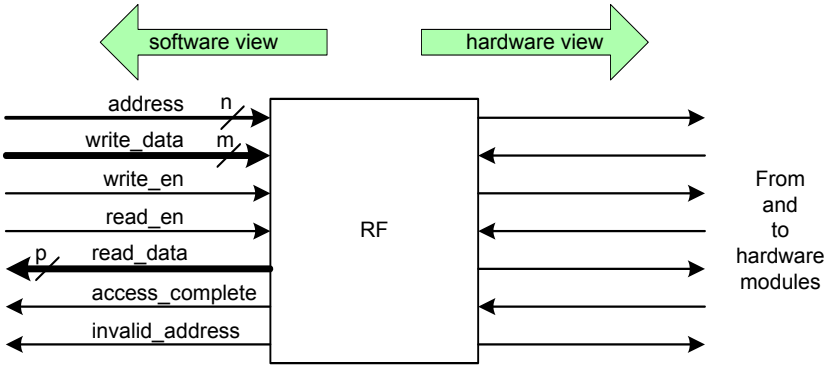


Figure 2-13: Software and Hardware View on the RF

- ***address***: The read and write address, the width **n** is dependent on the RF, when a read or write is performed then this bus has to provide stable addresses until the operation is completed with an *access_complete* signal.
- ***write_data***: This bus exists only when there is any writeable register in the RF, but this is the usual case, the width **m** is between 1 and 64 depending on the biggest writeable element in the RF, also this bus has to stay statically until the operation is completed.
- ***write_en*** / ***read_en***: This signal has to be pulsed for a single clock cycle, if this signal was not pulsed the RF would need an internal state to prevent that certain events happen two times.
- ***read_data***: Returns the read data and exists only when there is any readable element in the RF. The width is designated by the maximum width of the elements in the RF and between 1 and 64 bit.
- ***access_complete***: When the read or write is finished, then this signal is asserted for one clock cycle, in the next clock cycle the next read or write can be performed.
- ***invalid_address***: In the case a read or write is performed on an address that either does not exist in the RF or is not read or writeable, and then this signal is asserted for one clock cycle.

The read access is shown in figure 2-14, as it can be seen the *address* bus has to be set and the *read_en* signal has to be pulsed. In this example the *access_complete* signal is asserted only a single cycle later, this can also take much longer. The time depends on the number of hierarchy levels that have to be crossed to perform the read, each level adds at least two cycles.

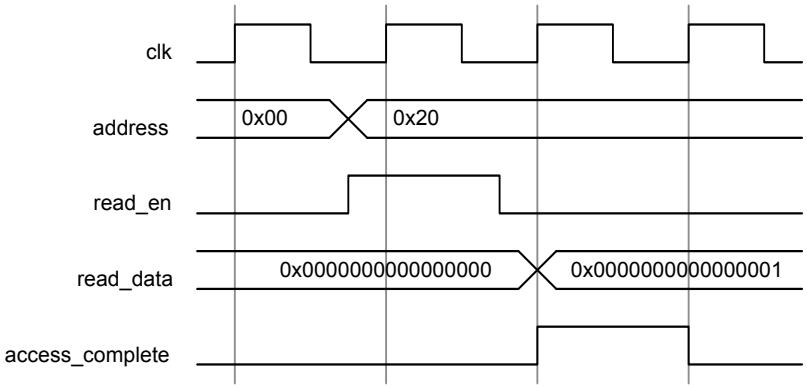


Figure 2-14: Read Access

To perform a write to the RF the address and the *write_data* bus has to be asserted, pulsing the *write_en* for one clock cycle will cause the actual write. This is depicted in the waveform in figure 2-15. The operation in this example takes three clock cycles, but it can also take longer depending on the RF.

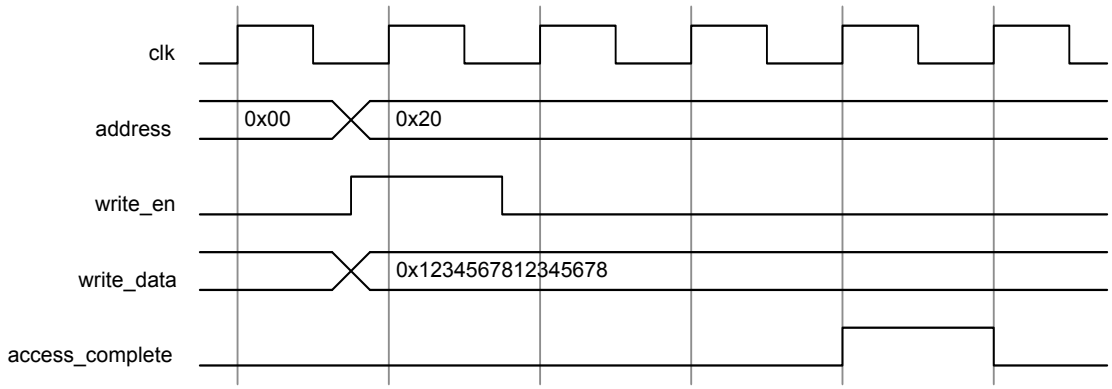


Figure 2-15: Write Access

2.4 XML Specification and Features

After the decision was made to use the Extensible Markup Language (XML) [4] as base format a schema had to be developed that is pleasant for the developers, which define the layout of RFs. Furthermore, it should be as easy as possible to parse. As explained earlier Spirit IP-XACT XML did not meet these requirements.

To aid the understanding of the rest of this section is it crucial to explain how the different parts of an XML representation are named. Figure 2-16 shows a very simple part of a XML file and names different parts of it. Most important, the start and end tag of **blue** together are called element. The same applies to **green**. Attributes can be placed either in a start tag or in an empty tag.

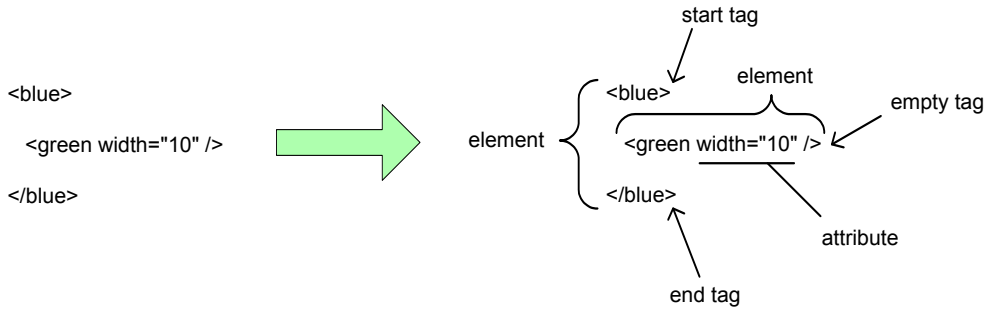


Figure 2-16: XML Legend

In contrast to IP-XACT XML options, like the width of a CSR, will be represented as attributes, the advantage is that this requires fewer code and is therefore easier to read and write.

Generated RFs can have a hierarchical layout as depicted in Figure 2-17, therefore the XML files are also organized in such a way. Level **0** is optional and reserved for the RF_wrapper. The highest level in the actual RF is **1**. This is the level of the top RF. Each sub-RF level below gets a bigger number. Of course, it is also possible to define RFs with only one level, but bigger ones have to be organized in a hierarchical fashion.

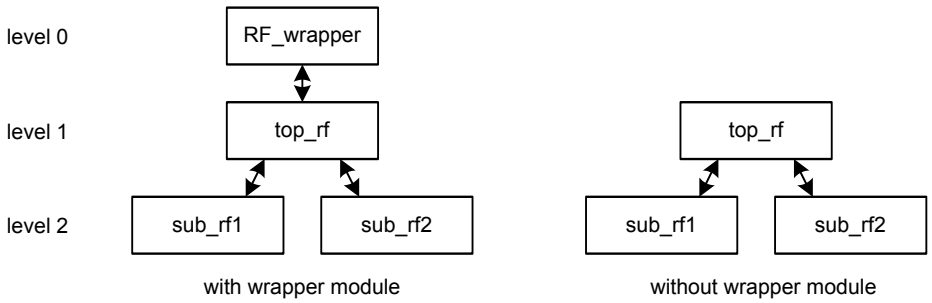


Figure 2-17: Hierarchy Levels

On the right hand side of Figure 2-17 there is a RF that does not make use of a wrapper module, it can be used directly with the RF interface as described before. On the left hand side a wrapper module RF_wrapper is employed, such a module contains code to translate between the RF interface and another protocol. To give an example, the EXTOLL design is using an on chip network (HTAX) that transports the HToC protocol.

This brings up the question why this wrapper module has to be generated and is not statically for each project. The reason for this is that the RF has to be connected with many different wires to the top module and obviously this should not have to be done manually, but rather automatically. A schematic diagram of such a setup is shown in figure 2-18. It seems like if the HT to RF Converter could also be placed in the top_module, this would make the RF_wrapper useless, on a logical level this is correct, but in the physical implementation on an FPGA it is often necessary to specify placement constraints, then it is easier when the RF and the converter reside in the same module.

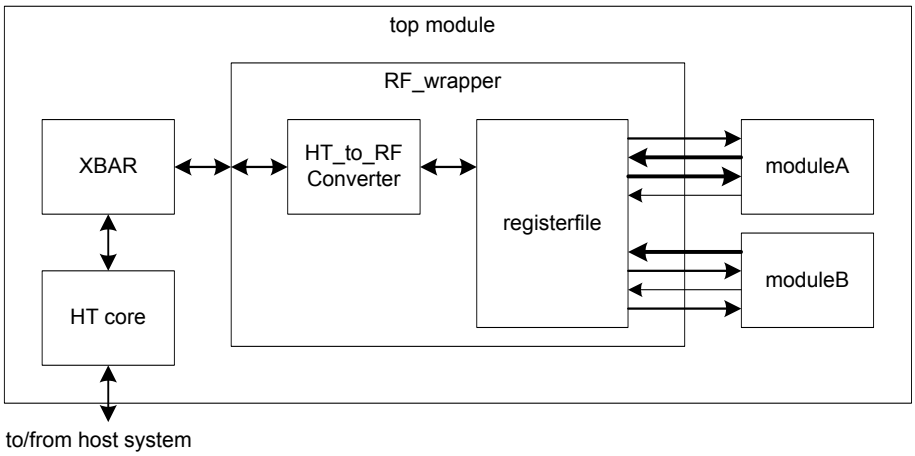


Figure 2-18: Placement RF_Wrapper

2.4.1 Basic XML Example of a RF

To illustrate the XML specification, a very simple example will be explained, which shows already a few properties of it. Based on this example the descriptions of the other elements that can be used in an RF will be easier to follow.

Every RF must have a XML file where the hierarchy starts. This file does by itself not contain any definitions for RF instances. Instances are XML elements that will result in implementations in the HDL output. In this example the file is called example.xml.

```
<?xml version="1.0"?>
<vendorsettings>
    <somesetting/>
</vendorsettings>
<regfile>
    <doc name="test" project="test" version="0.1"
         author="test" copyright="nobody"/>
    <wrapper value="htax">
         <rrinst name="top" file="top.xml" />
    </wrapper>
</regfile>
```

As it can be seen in the example there is a **vendorsettings** element, it will be copied together with all its sub elements to the annotated XML, RFS will not process it in any way. The purpose of this is to be able to stick as closely as possible to the principle of having only a single source of specification, this vendorsettings information can be used to provide information to the software that makes use of the annotated XML file. A concrete example is the management software for EXTOLL. It uses the annotated XML files to find the addresses of RF elements and the information about the hardware from the vendorsettings, like for example the number of link ports.

The actual definition of the RF starts with the **regfile** element. It has to contain two mandatory sub elements. First, the **doc** element contains information for the generation of the documentation. The second element is either a **wrapper** or an **rrinst** element. Rrinst is short for regroot instantiation. The wrapper is optional, it can be placed around the rrinst, depending on the project requirements the RF is built for. The rrinst element requires two attributes, the name of the instance in the RF and the file where the actual regroot is defined.

The following XML code is the **top.xml** file of this example and presents a regroot:

```
<?xml version="1.0"?>
<regroot>
    <reg64 name="testreg">
        <hwreg width="64" sw="rw" hw="rw" reset="$zero" />
    </reg64>
    <ramblock name="white" addrsize="2" ramwidth="64" sw="wo" hw="ro"/>
</regroot>
```

As it can be seen in this example the regroot is used in an own XML file and contains itself other elements. Each regroot has to be in an own XML file. The rrinst element is used to instantiate a regroot within the hierarchy of a RF. A regroot is the XML representation of a single RF within the RF hierarchy, which will result in one HDL module.

The file attribute of the rrinst element can of course also point to other directories, so it is easily possible to put together RFs with definitions from different sub projects. However, it is not possible to have multiple XML files with the same name in different directories, because all XML files must have unique names. The reason for this is that the names of the XML files are used as names for the C header files and Verilog modules, which are generated for each XML file or to be more exact for each regroot. Allowing multiple XML files with the same name would add complexity, but would have no application that is worth the effort. An early version RFS also supported the definition of regroot hierarchies in a single XML file, but the complexity to support both methods was too high and not worth the effort.

The **regroot** in the **top.xml** file contains two elements, a simple 64 bit register and an SRAM.

Details and the numerous options of these elements will be explained in the following sub-sections.

The generation of the RF in this example is simply done by calling RFS it in the following way, because there are no command line options other than the filename.

```
rfs example.xml
```

The result of this example is the layout as shown in figure 2-19. RFS starts filling the address space from 0x0 and places the testreg at this position. When a base address other than 0x0 is required, then it is possible to use an aligner element for this purpose, it will be introduced later in this chapter. On the other side, the ramblock white will start at 0x20, because all elements will be aligned by RFS to at least their size. When an element is aligned this means that it starts at an address that is a multiple in whole numbers of its size.

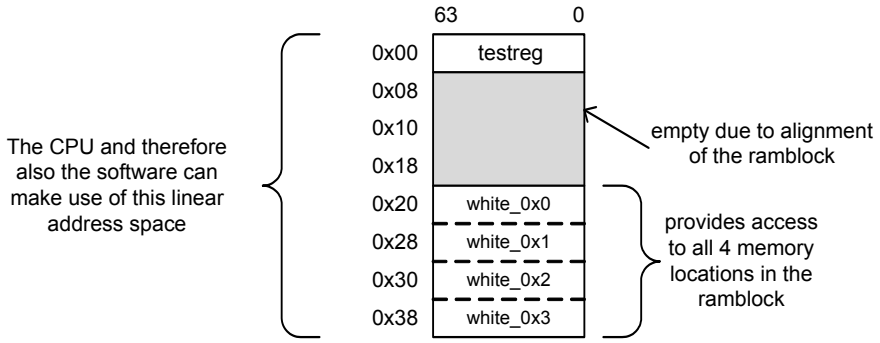


Figure 2-19: Example Address Space

The ramblock has an addrsize of two. It is containing four quadwords of data. Software can only write to it ("wo") and the hardware can only read ("ro") from the RAM.

This example presented a few elements of a RF and showed how RFs are composed in a hierarchical fashion. All elements in the RFS exist within a hierarchy. Figure 2-20 shows all important elements. The figure also shows how often an element can be used inside the hierarchy, **n** times denotes an arbitrary amount. In the case of the reg64 the number of sub elements is limited by the available space, the reg64 is 64 bit wide and therefore the sum of the widths of all sub elements has to be smaller than or equal to 64.

RFS does align all elements at least to their size, rounded up to the next power of two. Each element does also occupy this amount of address space, by doing so the address for sub element does not have to be calculated, and it is enough to select the right parts of the address. Thus, addressing requires fewer resources.

Regroots can not be defined recursively, because this will lead to an undefined behavior.

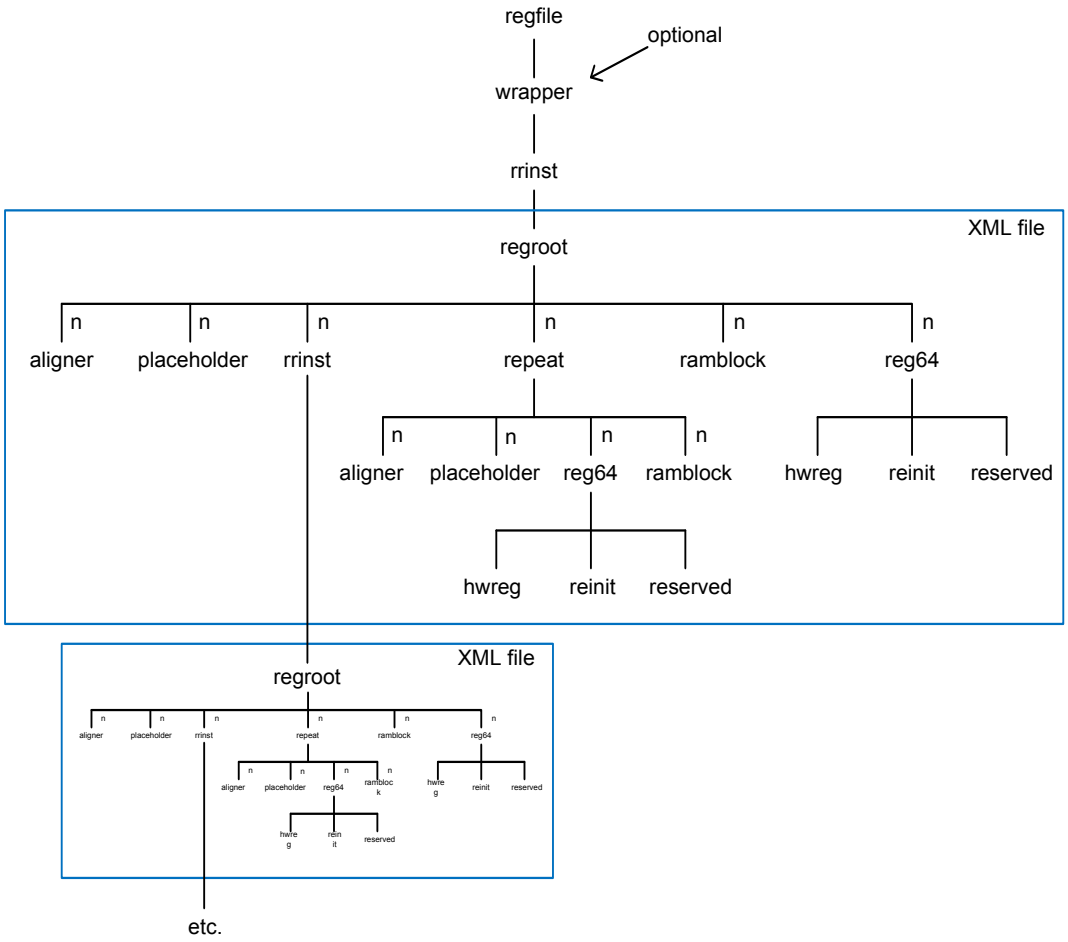


Figure 2-20: Hierarchy of RFS Elements

In the following all possible elements and their features will be explained in more detail and their importance in the context of complex hardware structures will be presented. Some of these features are essential; others are for convenience or are saving development effort.

2.4.2 Hardware Register - hwreg

An hwreg is the hardware representation of a register, and will instantiate the actual register in the hardware. At first it seems that it is enough to have registers that are writeable and readable by software and hardware, but in the end more complex and different solutions are required to be able to design efficient hardware. The hwreg can be considered to be a synonym for the field in SystemRDL or IP-XACT, but the term hwreg fits better, because it denotes an actual **hardware register** in contrast to the reg64 that is only a shell. An hwreg is a part of a reg64 as depicted in figure 2-21.

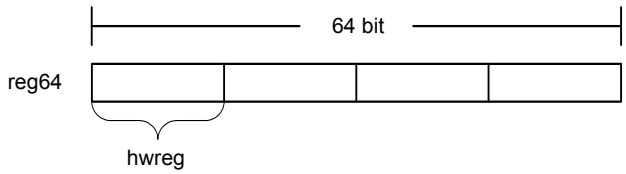


Figure 2-21: hwreg within reg64

This is a typical example for an hwreg:

```
<hwreg name="red" width="64" sw="rw" hw="rw" reset="$zero" />
```

It represents a 64 bit wide register in hardware with the name red. The **width** of an hwreg is between 1 and 64 bits.

Each hwreg despite the first one in a reg64 is required to have a **name**. When the first hwreg has no name, then it will carry only the name of the reg64. Otherwise the name of the inputs and outputs for the hwreg at the RF will be named by concatenating the reg64 name with the hwreg name separated by an underscore.

Figure 2-22 shows a schematic of the hwreg **red** that will result from its definition from above. The reset path was omitted for clarity. It also has to be mentioned that this is only a logical view and the most likely implementation, but it is possible that a synthesis tool chooses another way to implement the functionality.

The **sw** and **hw** attributes control the possibilities to read and write the register. In this example the software and hardware have both read and write access.

As it can be seen ***red_next*** and *write_data* are assigned in figure 2-22 to the actual flip-flops (FF) with a multiplexer (MUX) at **(1)**, therefore the *red_next* is assigned all the time to the FFs as long as the address is not correct and *write_en* is not set. The multiplexer in this schematic is switching to the lower *write_data* input when the select input is 1'b1.

On the side of the hardware view the read path is very simple, because the FFs just have to be connected to the outside of the RF module. The software read path is implemented by using a multiplexer that is controlled by an address decoder.

It makes sense to reduce these access possibilities as far as possible to save hardware resources. Both attributes can have the following values:

- "": no access at all
- "ro": read only
- "wo": write only
- "rw": read write

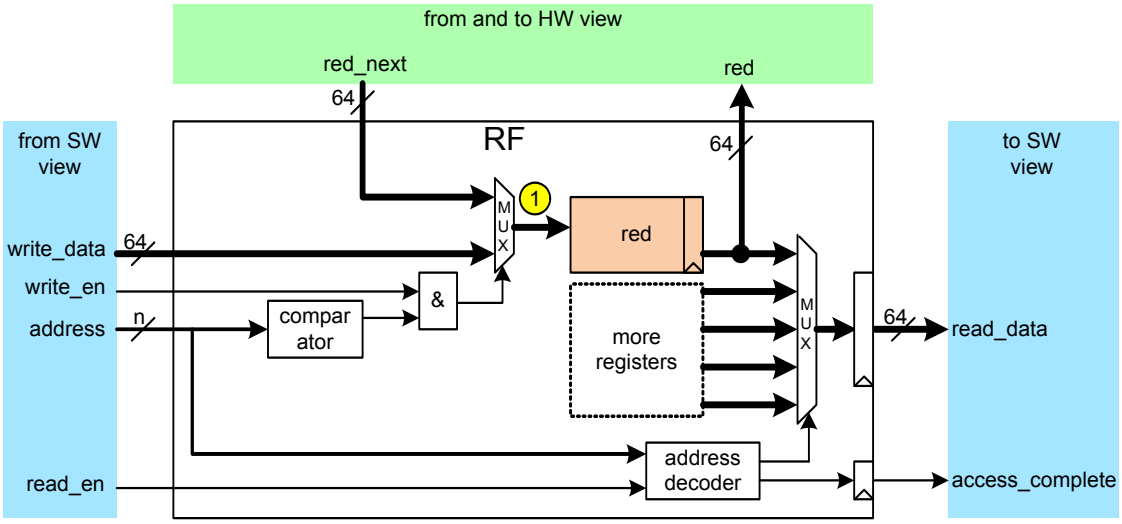


Figure 2-22: hwreg red

Table 2-1 shows all possible combination of the sw and hw attribute, supported ones are marked by **Y**, unsupported ones by **N**. Only combinations that make sense and were already required in a design are implemented. **A** and **B** appear to make no sense at first, but **A** is very useful to provide hardware specific parameters for the software. An example for such an application is storing the RF creation time as described in chapter 2.3.4. **B** is used to provide a location were the software can perform tests.

	hw=""	**hw="ro"**	**hw="wo"**	**hw="rw"**
sw=""	N	N	N	N
sw="ro"	Y(A)	Y	Y	Y
sw="wo"	N	Y	N	Y
sw="rw"	Y(B)	Y	Y	Y

Table 2-1: sw/hw Access hwreg

All cases where the software can neither read nor write also make no sense, because the hwreg would functionally not be more than a register and there is no reason to place it into the RF.

When the **hw** access attribute is set to allow writing, then the value from the *$name_next* input is applied every clock cycle to the internal register.

This is not always the desired behavior, for example an hwreg might be used to store the header of the last packet, received by some module, for debugging purposes. The header may be the first quadword of each packet and detected because there is also a start of packet signal. To have the header of the last packet in the RF it would be necessary to add an extra register to the hardware that is only updated in the case of a SOP.

Therefore, the **hw_wen** attribute was added to avoid the necessity of adding another register in such cases. Setting this attribute to "1" will cause the addition of an extra *$name_hw_wen* input to the RF. Thus, only when it is set to "1", then the input value of the *$name_next* input is applied to the internal register. This feature saves resources because no extra register is required to implement the example from the last paragraph.

In an ASIC all registers have an undefined value after powering up the chip, therefore a reset signal is required. This reset signal should cause the hardware to set initial values to all registers. In an FPGA things are a bit different, it is of course also an ASIC, technology wise, but all registers will be set by the bit file that is loaded into the FPGA. So in theory all registers have a predefined value. This predefined value is zero by default. Subsequently, it is not necessary to have a reset path to assign a value of zero to registers. Nevertheless, when the design is for example a PCIe card, then the hardware should reset all registers, if the reset line of the board is asserted.

Therefore, all RF registers should have a reset value. It is set with the **reset** attribute. The value has to be given in Verilog syntax and the width of the value has to be correct, RFS is performing no check on the width, because of the complexity to implement this check correct for all cases. However, if the reset value is given wrong a warning will be emitted by the linter or synthesis software. Special values exist to simplify setting the reset value and avoiding errors:

- $zero
- $ones
- $seconds

Most likely a register will be initialized to zero, therefore the special value **$zero** exists that will create a reset value in the Verilog format with the correct width. The same applies to **$ones**, but the bits are set to one of course. The special value **$second** can only be used if the register is 32 bit wide, this will use the creation time in seconds since 1st Jan. 1970 as reset value. This special value is required to ensure the consistency of the hardware and software interface as described in chapter 2.3.4. The value is acquired and stored by RFS when it starts to guarantee that the value, which is stored in the rfs_seconds.h file is the same like set by $seconds.

When no reset attribute is present, then RFS assumes the default value $zero. In the special case that there is a design decision to not reset a register, then the attribute has to be given empty. This way there will be no reset value for the register.

```
reset=""
```

The software is controlling the hardware with the help of CSRs. These are changed by the software. To be able to react to changes of these registers by the software there are two possibilities for the hardware:

- comparison
- output signal to indicates a write by the software

Comparing the content of a register to sense a change has two disadvantages. First, it requires an extra register with the same size like the register that has to be compared and additionally a comparator. Second, with this method it is only possible to detect changes, but not that the register was written at all. There are cases where this information is required. An example for such an application can be seen in the interrupt mechanism in chapter 3.6.5.

For this reason the **sw_written** attribute was introduced. When it is set to "1", this means that an output is added to the hardware and this output is 1'b1 for a single clock cycle, as a side effect, when the register was written by the software. This is independent from the content of the register, because the write itself triggers the sw_written pulse and not a comparison with the former value. Denali Blueprint did not offer this possibility.

Hardware modules are in some cases depending on the content of an hwreg, but are also making use of the sw_written feature. Therefore, it would only get the content of the hwreg after a write by the software, despite it could already start working with the default value in the RF. To solve this problem the hardware module could read the content of the hwreg directly after the reset, but this increases the complexity of the module slightly. Therefore, a sw_written="2" variation was implemented, in this special case the sw_written signal will also be asserted one cycle after the reset of the RF, for the duration of one cycle.

Often enable bits in registers are used to control the activation and deactivation of hardware features. Such a register is shown in figure 2-23. To enable for example feature 0 it is necessary to read out the complete 64 bit register, set the corresponding bit to 1'b0 and write the whole 64 bit register back to the hardware. This is a read-modify-write operation. When there are different threads responsible for different features, it is necessary to prevent that the threads do the read-modify-write operation at the same time. This can be done from the software for example with a mutex.

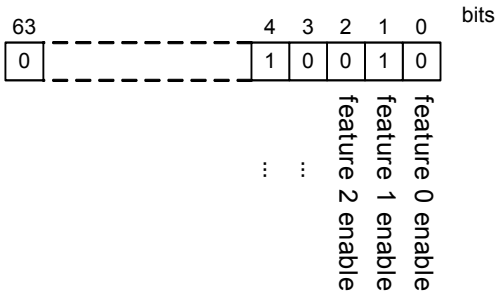


Figure 2-23: Enable Bits

Using a mutex is of course possible and no problem when all threads are in the same process context, but there are also cases where the threads may be in different processes. In this case using a mutex gets more complex and the standard pthread mutex [121] can not be used.

Therefore the **sw_write_xor** attribute exists, when this feature is activated, then writes from the software are applied with XOR to the register. By using this feature the thread only has to care for the enable bits it is responsible for and can invert exactly these bits in the hardware. Thus, this feature provides an atomic method to set or unset bits.

Register bits that are set by the software can be set back to the reset value by the hardware with a special input at the RF. When hw_clr="1" is set, then a clear (*$name_clr*) wire will be added to the interface of the RF to set the register back. This feature is only usable when the register is neither writeable by hw nor it is a counter.

For the implementation of an HT configuration space [30], a special feature is required. The specification describes that certain registers should return to their initial value when the register is written.

> *"Each field is defined as readable and writeable by software (R/W), readable only (R/O), or readable and cleared by writing a **1** (R/C)."*

For this reason the attribute **sw_write_clr** was added to the hwreg. When this feature is activated then any write by the software will set the register to the reset value and not to the value that was written. This is independent from what is written to the register, any write will have this effect. Obviously, this implementation is inaccurate to a certain degree, because it does not require that a **1** is written as required by the specification, but there are two reasons why this was not done. First it would add complexity and it posed no problem when it was tested in system. Second, the specification does not make clear what writing a 1 means, this could be a single 1 in the corresponding field or it could be a 1 in the register that contains the field. RFS will allow the usage of sw_write_clr only when the hwreg is writeable by software.

Settable Only Registers - sticky

Assertions and event indicators are useful to find out about errors in designs or to find out if a specific condition was reached. Characteristic for such an event is that logic detects the condition and sets a register to one.

An example for such a condition is that a FIFO was full and still data was inserted into the FIFO at least once. In this case the *full* signal would be used together with the *shift_in* of the FIFO to set the flag for a FIFO error. Such information can give the developer at least an idea where things are going wrong in the design.

Therefore, the registers have to be set to one by the hardware, and then the software will read them out, and will maybe set the register back to zero. It is very easy to implement at a first glance as shown in figure 2-24.

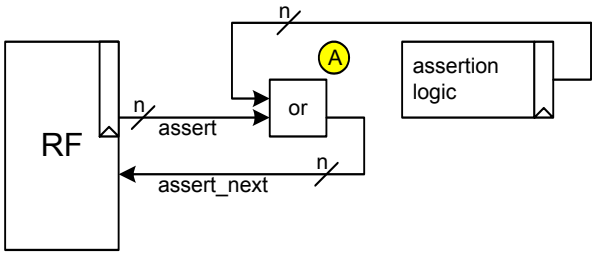


Figure 2-24: hwreg Assert Logic

The problem is that there is a possible read-modify-write hazard in the time between the reading of the register and the writing of the register. All new assertions information, happening in the meantime, will be lost and therefore not be seen by the software, because there is most likely more than a single assertion bit in the reg64.

Therefore, a method is required to avoid the problem, the first part is that the OR gates **(A)** in figure 2-24 are moved into the RF. Otherwise, the following solutions can not be implemented or would have to be implemented manually. This is done by setting the **sticky** attribute to "1".

Next, a mechanism is required to be able to read out all information from the register and setting the bits back to zero without losing information, because of a read-modify-write hazard. There are two possibilities:

- writing with XOR
- clear on read

Earlier the **sw_write_xor** attribute was introduced. Using the XOR operation allows it to set back only specific bits from 1 to 0, therefore the hazard can be avoided by using this feature and no information is lost by doing so.

Even faster is the possibility to clear a register, when a read is performed, by using the **sw_read_clr** attribute. Thereby, only a single transaction is required to read out the event information and reset the registers. However, there are disadvantages associated with this approach. Some systems implement 64 bit reads by reading two times 32 bits, when in RFS a reg64 is wider than 32 bit, then the first read will clear the remaining bits. Fortunately, this is only a problem with old AMD Opteron of the old K8 generation and probably other old hardware. Therefore, sw_read_clr has to be used with caution.

Furthermore, it is very unusual that reading does have an effect and effectively can cause the loss of data. Thus, it has to be ensured that the software developers that make use of the hardware are aware of this.

The following XML code will instantiate a hardware register with this property:

```
<hwreg name="red" width="64" sw="ro" sw_read_clear="1" hw="rw" sticky="1"/>
```

The reset value was omitted in this case because every hwreg will reset to zero by default, if no other reset value is supplied. This XML definition of a register results in the hardware that is depicted in figure 2-25 as schematic representation. Here it also gets obvious why the sticky functionality has to be integrated into the RF, because otherwise the OR gate **(1)** could not be between the MUX and the red register.

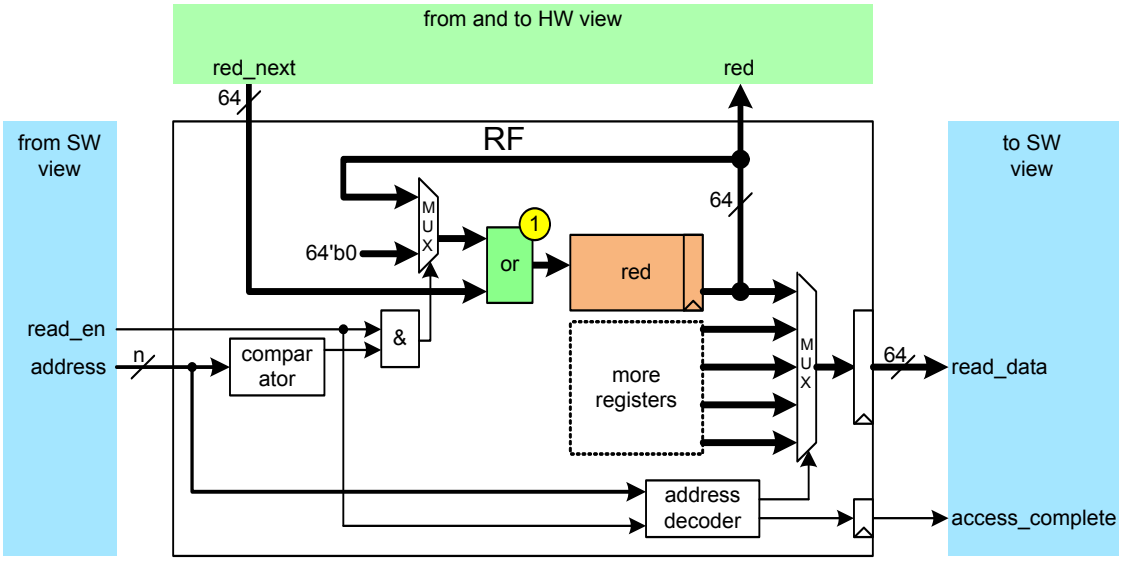


Figure 2-25: hwreg with sticky and sw_read_clr

Counters

Counters are very important in complex designs because there are many possible events. Counting these events can provide valuable information about the hardware behavior. The counter values can be used for different purposes; the most important ones are statistics and debugging.

EXTOLL is, as of January 2012, using 257 counters of different sizes for analysis and debugging purposes in the FPGA version.

An example for using counters to gain statistical information is that two counters can be used to find out what the average packet size in a network is. One of the counters has to count the number of packets, the other one can be used to count the bytes that are contained in all the packets. So the average packet size can be calculated for a time span.

Debugging is often only possible with the help of counters. For example in hardware that is processing packets, it may happen that packets are lost due to errors. When the problem happens after millions of packets then it is practically impossible to simulate. With the help of counters at different positions in the design it is at least possible to find out in which part of the hardware the packet is lost. This can provide information for the further debugging of the issue.

A counter itself is very easy to implement in Verilog, but there are very good reasons to provide special support in RFS.

- Less code has to be written manually in each module, thus sources are less cluttered and less error prone.
- Special hardware functionality can be used more easily to implement the counters.
- Special ways to reset counters can be implemented, these allow saving resources and having functional advantages.

Counters are usually only increased by the hardware and the actual counter value does not have to be readable or writeable by hardware for this reason. For special applications it is also possible to use the hw attribute to allow this.

The EXTOLL design uses a counter in the RF to provide a time stamp counter (TSC). To be able to synchronize it with the help of hardware it has to be possible to write it from hardware. Therefore, it is defined in the following way.

```
<reg64 name="tsc" desc="time stamp counter (TSC)">
    <hwreg counter="1" width="48" sw="rw" hw="rw" hw_wen="1" />
</reg64>
```

As is can be seen in the hwreg element also the hw_wen element is set, this is required when a counter is used with the hw attribute set to "wo" or "rw". If the input value would be assigned each cycle to the counter, then the counter value would be over written in each cycle and it would not work as counter.

Modern FPGAs have special digital signal processing (DSP) components for calculations. In many designs these are not used extensive, therefore they can be claimed for RF purposes. Xilinx FPGAs have DSP slices since the Virtex 4 generation in the form of the DSP48 slices [85]. These components have a multitude of features, to cite the user guide [105]:

> *"The DSP48 slices support many independent functions, including multiplier, multiplieraccumulator (MACC), multiplier followed by an adder, three-input adder, barrel shifter, wide bus multiplexers, magnitude comparator, or wide counter."*

The more recent Virtex 6 has DSP48E1 [86] slices that have even more features. Altera offers similar DSP resources in the Stratix IV devices [87].

All these DSP slices contain at least one multiplier and one adder, but for the RF only a small part of the functionality is used. Not more than an adder and the output register are required. For this purpose a wrapper module exists that implements a counter that is up to 48 bit wide. The interface is specified in the following way:

```
module counter48 #(
    parameter DATASIZE= 16, // width of the counter, must be <=48 bits!
    parameter LOADABLE= 1   // whether the counter can be loaded at runtime
) (
    input wire                  clk,
    input wire                  res_n,
    input wire                  increment,
    input wire[DATASIZE-1:0]    load,
    input wire                  load_enable,
    output wire[DATASIZE-1:0]   value
);
```

Implementations of this module were done for Xilinx, Altera and in generic Verilog for simulations and ASICs.

To have concrete numbers about the resource savings, achieved by using the DSP slices, a small test design was created. This evaluation design contains sixteen 32 bit wide counters in a register file. The repeat element in this example will be detailed in chapter 2.4.7. It is defined in the following way:

```
<reg64 name="init">
    <hwreg name="dummy" width="31" sw="rw" hw="ro"/>
    <rreinit />
</reg64>
<repeat loop="16" name="test">
    <reg64 name="cnt">
```

```
            <hwreg counter="1" width="32" sw="rw" hw="" rreinit="0"/>
        </reg64>
    </repeat>
```

The extra “init” register was added, because it will be needed to demonstrate the rreinit feature later on. To be able to compare the different resource requirements, first a synthesis run was performed with a counter48 module that does not make use of DSP slices and one that uses DSP slices. The comparison, which is shown in table 2-2, was performed for the Xilinx Virtex 4, because it was the FPGA that was used most in the research group when the development of RFS started. It can be seen clearly that not only logic in the form of “Slice LUTs” is saved, but also “Slice Registers”, because the output registers of the DSP slices are used for this purpose.

Resources	no DSP	DSP
Slice Registers (FFs)	612	64
Slice LUTs (4 input LUTs)	1819	282
DSP48E	0	16

Table 2-2: Resource Usage on Virtex 4: Counters

Over the year 2011 all developments shifted to Virtex 6 FPGAs. Thus, the same evaluation was also done for the Virtex 6. The results are shown in table 2-3. In this direct comparison between Virtex 4 and Virtex 6 it also gets apparent how much more efficient the LUTs with 6 inputs are compared to the ones with 4 inputs when a counter is implemented.

Resources	no DSP	DSP
Slice Registers (FFs)	639	79
Slice LUTs (6 input LUTs)	777	214
DSP48E1	0	16

Table 2-3: Resource Usage on Virtex 6: Counters

There are multiple methods that can be used to increase counter values to save development effort. Thus, the attribute counter can have different values, in the standard case it is "1". A waveform diagram of the standard case is shown in figure 2-26; the counter is incremented by one in every cycle as long as the ***countup*** input is one.

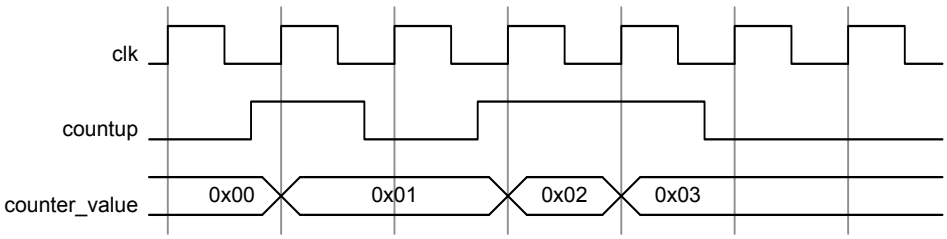


Figure 2-26: Counter Type 1

There are cases where it can be helpful if there are other ways to increment counters, because the standard way with a *countup* signal will require extra code. A common pattern is that a counter has to be increased only when a specific condition in a big case construct is met.

In the following example, the counter should only be increased under one specific condition.

There are two possibilities to implement this with the standard type 1 counter. In the first case the countup signal has to be set to 1'b1 in the case when it should be increased and to 1'b0 in all other cases, this can result in many lines of code and possibilities to forget cases.

```
casex({valid,state})
{1'b1,4'b0001}: begin
    if(condition)
         cnt_countup <= 1'b1; // <= this condition
    else
         cnt_countup <= 1'b0;
end
{1'b1,4'b0010}: begin
    cnt_countup <= 1'b0;
end
... // more possibilities, not shown
endcase
```

The next possibility is to place the control for the countup signal in an extra block.

```
if(valid && (state==4'b0001) && condition)
    cnt_countup <= 1'b1;
else
    cnt_countup <= 1'b0;
```

Consequently, another location in the code has to be maintained and kept in sync with the rest of the code.

The alternative that is presented here, is to use edge detection, because this way a single bit register can be used that is inverted in the case the counter should be increased. This feature can be used by setting the counter attribute to "2".

```
casex({valid,state})
{1'b1,4'b0001}: begin
    if(condition)
```

```
            cnt_edge <= ~cnt_edge;
    end
    {1'b1,4'b0010}: begin

        ...
    end
    ... // more possibilities
    endcase
```

Rather then using many lines of code to set a register either to 1'b0 or 1'b1 it is possible with this method to do the same with only a single line of code. The behavior of counter type 2 is show in the waveform in figure 2-27.

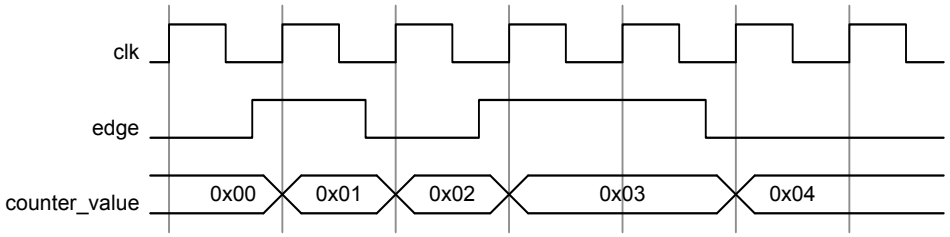


Figure 2-27: Counter Type 2

Consequently, there is also a way to count up only on rising edges. This is for example useful when packets have to be counted which are accompanied by a valid signal. With counter type 3 the valid signal can be connected directly to the RF. The waveform of this counter is depicted in figure 2-28. Counting packets this way does of course only work when they are not back to back, but in this case there has to be another delimiter between the packets that can be used for counting. Falling edges can be counted by inverting the signal.

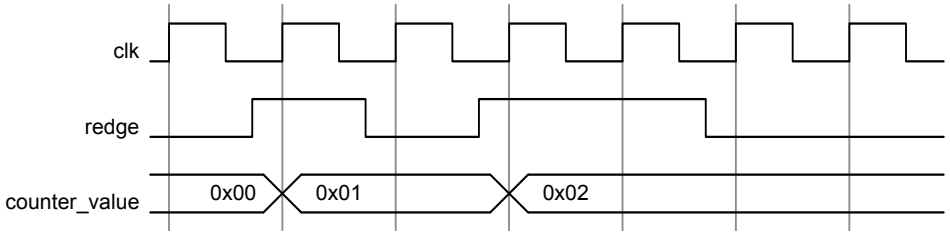


Figure 2-28: Counter Type 3

The advantages of the counters that make use of edge detection also have a price, because the edge detection requires logic and flip-flops. Therefore, the amount of resources required for the counter type 2 and type 3 was evaluated. Table 2-4 shows the resource requirements of the different counter types for a RF with sixteen 32 bit counters. Each of the counters in the case of type 2 or 3 requires two extra registers and one extra LUT compared to the type 1 implementation. The reason for this is the that one register is required to store the value of

the edge input in the last cycle and the other register controls the increment signal to the counter48. However, the extra register provides the advantage that the timing is relaxed, because it is an extra register stage between the input of the RF and the actual counter48.

Resources	type 1	type 2	type 3
Slice Registers	79	111	111
Slice LUTs	214	230	230
DSP48E1a	16	16	16

Table 2-4: Resource Usage Comparison for the rreinit Feature

Reset of Counters - rreinit

In big designs there are often multiple counters in the same regroot, some of these counters usually relate to each other. For example one counter may count the number of packets and another one the amount of data transferred. Thus, they are together providing the information about the average size of a packet. At some points the counters have to be set back to zero, because a new experiment will be done or because one of the counters is overflowing. To be able to calculate the average size accurately, it is important that the counters are set back at the same time.

When the counters are written one by one from software then it can happen that the actual writes do not happen directly behind each other. Two possible reasons for this are that either the thread was scheduled away by the OS or there might be a lot of traffic on the PCIe interface.

Nevertheless, it can not be guaranteed that the counters are set to zero at the same time.

Therefore, it is desirable to have a method to reset multiple counters at once. This feature requires an additional reg64 element in the regroot that contains an rreinit element.

As a result, it is possible to use the **rreinit**="1" attribute for counters. When this attribute is used then it is no longer possible to write the counter directly by software, therefore the sw attribute has to be set to “ro”. Furthermore it is also not possible to write the counters from the hardware view. Thus, the counter example from before has to be written in the XML in the following way:

```
<reg64 name="init">
    <hwreg name="dummy" width="31" sw="rw" hw="ro"/>
    <rreinit />
</reg64>
<repeat loop="16" name="test">
    <reg64 name="cnt">
```

```
            <hwreg counter="1" width="32" sw="ro" hw="" rreinit="1"/>
        </reg64>
    </repeat>
```

Rather than writing all registers that carry the rreinit attribute it is now sufficient to do a single write to the reg64 with the rreinit element and all counters with the attribute will be set to zero. Other initial values were not considered, because there is no good reason for a reset value different from zero in the case of a counter.

This way the counters do not have to be writeable by software and therefore no write path to the counter registers or DSP slices is required. The assumption is that this will save resources, therefore the example RF with sixteen 32 bit counters from the last subsection was used again to find out how big the possible savings are.

Resources	no rreinit	rreinit
Slice Registers	79	64
Slice LUTs	214	199
DSP48E1a	16	16

Table 2-5: Resource Usage Comparison for the rreinit Feature

Table 2-5 shows the results. In the “no rreinit” column there are exactly the same numbers like from the last comparison in table 2-3, for the “rreinit” column the rreinit attribute was set to 1 and additionally the sw access rights were set to read only. The resource saving is small, but nevertheless this is a helpful measure because no routing resources are required this way to write the counters.

Routing resource usage can, at least with the Xilinx FPGA tools, not be accounted easily, but they are a scare resource in congested regions of the FPGA and counters can also not be placed freely by the FPGA tools, because the DSP slices are at fixed positions within the FPGA fabric. Thus, saving routing resources can be crucial to reach timing closure.

2.4.3 64 bit Register - reg64

The 64 bit register is an element that appears to be a single 64 bit register from the software view, this means that it can and has to be used as a 64 bit value by the software. In the following it is called **reg64** to avoid confusions. This is also the name of the XML element. The reg64 is the software view on up to 64 bits of hardware registers, because it can be

addressed from the software view interface. The reg64 has by itself no representation in hardware. The actual hardware elements are represented by sub elements in the XML definition, the reg64 can not be instantiated without sub elements. Each reg64 can contain:

- hwreg
- reserved
- rreinit

The **hwreg** is the most complex element and was explained in the previous subsection.

The **rreinit** element allows a special way to control counters. Instantiating it in a reg64 will create a software writeable “register” that will trigger an rreinit event in the regroot of this instance. It was also explained in the previous section.

Software acquires the content of registers by reading the 64 bit wide reg64 that can contain multiple hwregs. As a result, the software will use masking and shifting to extract the information from the different hwregs. To be able to keep the order and place of bit fields within a reg64 constant it can be very useful to have the ability to reserve bits for future uses. For this reason **reserved** fields can be used as place holders, which have no direct resource requirements. When the software reads from the register then these bits will be zero. This way the software that makes use of this register can be written to anticipate bigger sizes of certain registers.

The size of the reserved field is given by the width attribute. An example is a reg64 that contains a register which stores the size of a packet, in an early phase of the development eight bits might be enough, but the expectation could be that in future more bits will be required.

```
<reserved width="n" />
```

64 bit Register Example

The following example will demonstrate the results from the instantiation of different elements that can be used in a reg64 and how they are combined in the RFS XML specification language:

```
<reg64 name="example">
    <hwreg sw="rw" hw="ro" name="test1" width="16" sw_written="1"
         desc="description”/>
    <hwreg sw="rw" hw="" name="cnt1" counter="2" width="16" />
    <reserved width="16" />
    <hwreg sw="ro" hw="rw" name="test2" width="16" />
</reg64>
```

In figure 2-29 the different fields in the reg64 are depicted.

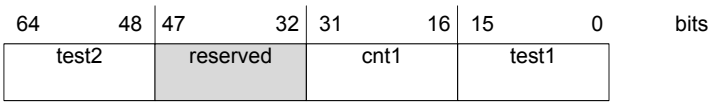


Figure 2-29: Example reg64

RFS will also generate an entry in the C header files; the header contains the reg64 in the form of a uint64_t. To help the software developer, extracting the content of the separate hwregs, masks are provided in the comments. Bit fields were not used, because it is faster when the quadword, which contains multiple hwregs, is read at once and split afterwards in its parts. Using bit fields would cause the compiler to generate code that reads from the RF for every bit field, because the volatile keyword must be used for RFs.

```
/* [15:0] test1 mask=0xffff (desc=description) */
/* [31:16] cnt1 mask=0xffff0000 */
/* [63:48] test2 mask=0xffff000000000000 */
volatile uint64_t example;
```

This example will result in multiple inputs and outputs of the RF that can be used to connect logic. Their number and type depends on the attributes of the hwregs. Figure 2-30 depicts the interface that will be generated for the example.

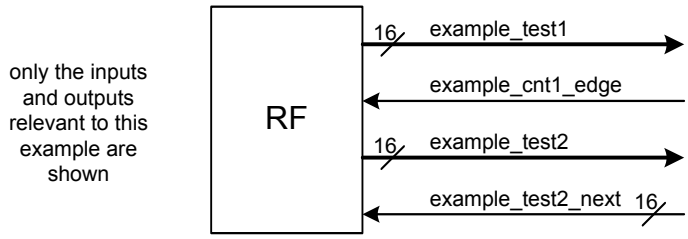


Figure 2-30: Example reg64 Block Diagram

2.4.4 Aligner

There are multiple reasons why a certain alignment may be required in a register file. These reasons may due to either technical or organizational requirements. When RFS is used to model a RF that implements a certain specification then it may be required that some registers are at specific addresses.

The aligner is used to align the following elements to an address with a certain condition in terms of alignment. There are two mutually exclusive methods. Either the **absolute** attribute can be used. It will put the next element in the RF to the given or a higher address, depending on the alignment requirements of the next element.

```
<aligner absolute="0x1008" />
```

This aligner element will enforce that the next element can start exactly at an address of 0x1008. Though, it might be that the next element starts at a higher address if it requires itself a certain alignment.

The next possibility is using the **to** alignment, the parameter denotes the number of least significant bits (LSB) that have to be zero.

```
<aligner to="6"/>
```

With this aligner element the start address of the next element will be a multiple of 2^6, thus it will be aligned to a 64 byte boundary or a cache line.

There are two reasons why such an aligner can or have to be used:

- organization
- security / safety

The **absolute** aligner is often used to organize big RFs and assign different start addresses to sub-RFs of different modules. This is technically not necessary, because RFS can handle the addressing automatically, but it is a design decision of the development team.

When a hardware unit can be used by different user level processes then it has to be guaranteed that hostile user level processes are not able to read or write parameters that they are not supposed to access.

The obvious solution is to use the OS kernel as proxy between hardware and software. So every time a user level process wants to read or write a value it has to inform the OS kernel about this wish. Then the kernel will decide based on an access control list (ACL) if it will perform the access. When the application is latency sensitive then this is out of question because it will be relatively slow.

In x86/x86_64 computer systems the page size is 4 kB, therefore this is the size the MMU can govern. Thus, the aligner can be used to put RF components at 4 kB boundaries, so that the OS kernel can map them into the virtual address (VA) space of user level processes.

The following example will demonstrate how it is possible to create an address setup that allows utilizing the MMU for access control purposes.

```
<aligner to="12" />
<reg64 name="regA">
    <hwreg sw="rw" hw="rw" width="64" />
</reg64>
<aligner to="12" />
<reg64 name="regB">
    <hwreg sw="rw" hw="rw" width="64" />
</reg64>
<reg64 name="regC">
```

```
        <hwreg sw="rw" hw="rw" width="64" />
    </reg64>
    <aligner to="12" />
```

This example shows the definition of the 64 bit registers regA, regB and regC. These registers are encased by aligner elements; the resulting address setup is shown in figure 2-31. The OS kernel can map the 4 kB address range with regA in the address space of one user level process and the address range that contains regB and regC into the address space of another process. These user level processes will only be able to access exactly the registers in the 4 kB page they are supposed to access and will not be able to access any other elements of the RF.

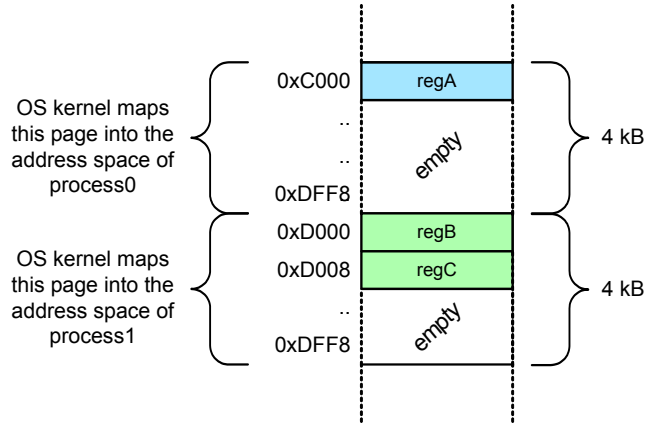


Figure 2-31: Address Setup with aligner

2.4.5 Ramblock

Some values inside a RF can be stored in SRAMs, depending on how the hardware has to be able to access the values. When an SRAM is used then also the hardware view has to use an SRAM interface to read or write the data. If this is a disadvantage or an advantage depends on the type of logic.

The XML element is called ramblock. The ramblock can be used from the software exactly like the corresponding amount of registers.

The main advantage of using a ramblock is the lower resource usage, especially in the FPGAs.

The ramblock element can contain **field** sub elements, which are used to describe the content or purpose of the different bits in the ramblock. RFS does not use this information to generate code, but puts the information in the documentation and the annotated XML. Furthermore, it is comparing the sum of the widths of all fields with the width of the ramblock. This is a small sanity check.

To give an example for a ramblock:

```
<ramblock name="exaram" addrsize="8" ramwidth="33" sw="rw" hw="rw"
    external="0" desc="Example RAMBLOCK">
    <field name="packet_counter" width="24"/>
    <field name="current_packet_length" width="8"/>
    <field name="retry_bit" width="1"/>
</ramblock>
```

Each ramblock is required to have a name. The **ramwidth** can be between 1 and 64 bits. Ramblocks of course also have a size, this size is described by the addrsize attribute, and this attribute can also be 0 for a special application that will be demonstrated in this subsection. The example ramblock has 256 entries.

There are two different types of ramblocks that can be implemented with RFS, they can either be internal, which is the default, or external by setting the **external** attribute to "1". When a ramblock is internal this means that the SRAM will be instantiated automatically within the RF, but when the ramblock is external then an SRAM interface will be provided by the RF and it is up to the developer to connect it.

The example from above instantiates a SRAM within the RF module and provides a read/write interface that can be used by the hardware. All the interface signals that will be provided for the hardware are depicted in figure 2-32.

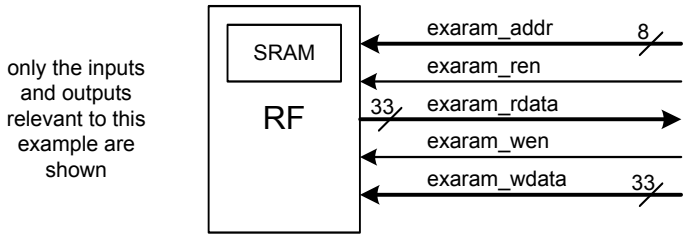


Figure 2-32: Ramblock Interface Example

Ramblocks are supporting the same possibilities to control the access to the SRAMs like hwregs, therefore the sw and hw attributes can be used. Not all combinations of sw/hw attributes are supported; only combinations that make sense and were already required in a design are implemented. For example, a ramblock that is write only for software and hardware makes no sense.

Additionally there is also the possibility to use an external clock to drive the hardware view interface of the RAM, for this the **ext_clk** attribute has to be set to "1".

Depending on the sw/hw combination a different SRAM module will be instantiated. There are four different possibilities:

- ram_1w1r_1c (1 write port, 1 read port, 1 clock)
- ram_1w1r_2c (1 write port, 1 read port, 2 clocks)
- ram_2rw_1c (2 read/write ports, 1 clock)
- ram_2rw_2c (2 read/write ports, 2 clocks)

When the external clock is used this means of course that a variant with two clocks has to be used.

Table 2-6 lists all possible combinations for the sw and hw attribute when no external clock is used. Only few combinations are supported, the reason for this is that most of these make no sense. For example a ramblock that can not be accesses by software (sw="") does not serve any purpose from the perspective of the RF. The remaining combinations that may have useful applications were not implemented, because there was no demand.

	hw=""	**hw="ro"**	**hw="wo"**	**hw="rw"**
sw=""	N	N	N	N
sw="ro"	N	N	N	2rw_1c
sw="wo"	N	1w1r_1c	N	N
sw="rw"	N	2rw_1c	N	2rw_1c

Table 2-6: sw/hw Access Ramblock

The module interface of the first SRAM module ram_1w1r_1c should serve as example, because it is the shortest one:

```
module ram_1w1r_1c #(
    parameter DATASIZE= 78,// Memory data word width
    parameter ADDRSIZE= 9, // Number of memory address bits
    parameter PIPELINED= 0
) (
    input wire                  clk,
    input wire                  wen,           // write enable
    input wire [DATASIZE-1:0]   wdata,         // write address
    input wire [ADDRSIZE-1:0]   waddr,         // write data
    input wire                  ren,           // read enable
    input wire [ADDRSIZE-1:0]   raddr,         // read address
    output wire [DATASIZE-1:0]  rdata,         // read data
    output wire                 sec,           // single error corrected
    output wire                 ded            // double error detected
);
```

This SRAM module has one write and one read port, the former consists of ***wen***, ***wdata*** and ***waddr***, the later consists of ***ren***, ***raddr*** and ***rdata***. It is possible to save power, when the enables are not set all the time.

SRAM blocks in ASICs and FPGAs have a clock to output time that is relatively high in comparison to the clock to output time of flip-flops. Therefore, they should be pipelined usually for timing reasons. The clock to output time of the Xilinx Virtex 6 in speed grade 2 is 1.79 ns [9] [119].

The **pipelined** attribute of the ramblock can be used to control if the SRAM will be pipelined or not, the default is "1", but in some cases it can be fast enough to use no pipelining. Thus, hardware resources (flip-flops) and one cycle of latency are saved.

Earlier an application of the aligner was shown were the alignment was used together with the MMU of the host system to allow user level processes to access specific entries in the RF.

It is possible to reach a similar effect by using ramblocks instead of reg64. If multiple processes should be able to access one or a few entries of a ramblock within a RF, then it is possible to use the **addr_shift** attribute.

The addr_shift attribute allows shifting the address that is used by the software view to the right. For example, by setting this attribute to "9" each ram entry will be in an own 4kB page. This way the operating system (OS) kernel is able to map single entries of the ramblock to different processes in a save way, because the page size is on AMD64 systems 4kB. Figure 2-33 is depicting this.

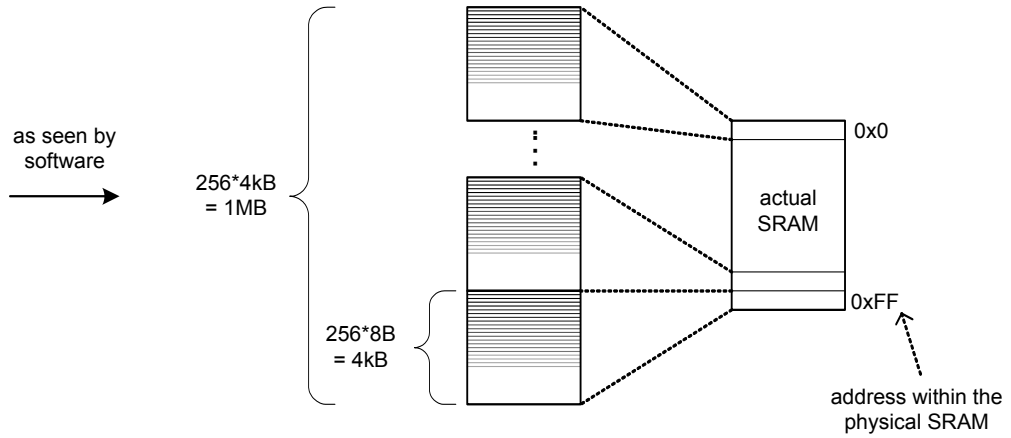


Figure 2-33: addr_shift of 9

External Ramblock

In some cases it is not possible to use the normal ramblock as described previously in a way that allows a direct instantiation within the RF. It is the most convenient way, but it is often either not efficient or impossible.

The problem is that the true dual-port SRAMs are “expensive”, this is true in FPGAs and ASICs. A “true dual-port” SRAM has two ports that both provide the possibility to read and write the memory. For EXTOLL when looking at the two different targets in both cases true dual-port SRAMs should not be used.

In Xilinx Virtex6 FPGAs the BRAM memories [9] have numerous configuration options, but it comes down to that these are only 36 bit wide in true dual-port mode, but can also be used in simple dual-port mode, then they can be used 72 bit wide. In the simple dual-port mode there is one read and one write port, this can be used to implement the ram_1w1r module. The ASIC implementation shares a similar problem, because SRAMs with true dual-port mode are slower.

Consequently, only one read and write port are available for EXTOLL, which is supposed to work on FPGAs and ASICs. Thus, arbitration has to be performed on these ports when software and hardware both should be able to read or write.

There were two possibilities where arbitration could take place, either within the RF or outside of the RF. The design decision was to leave the arbitration to the developer of the hardware logic, because this way the arbitration can be done in a way that fits the logic better. Additionally, the RF interface can be delayed because there is an *access_complete* signal, so the logic can prioritize according to its requirements.

External ramblocks can be instantiated by setting the **external** attribute to "1".

```
<ramblock name="ext" addrsize="8" ramwidth="33" sw="rw" hw=""
    external="1" desc="External RAMBLOCK" />
```

With this XML definition no SRAM will be instantiated in the RF, but an interface to the outside will be provided, the interface protocol is the same that is used to connect RFs. A detailed description was given in chapter 2.3.5. When a ramblock is external, then the hw attribute does not matter, because RFS has no control over this part. Depending on the sw attribute the interface will have a read enable (ren) and/or a write enable (wen). A possible setup with an external ramblock interface and an arbiter is shown in figure 2-34.

This external interface opens up the possibility to connect hardware units that provide the interface, but are no RAMs. It allows accessing special units with very low overhead, in terms of development effort and hardware resources, from software.

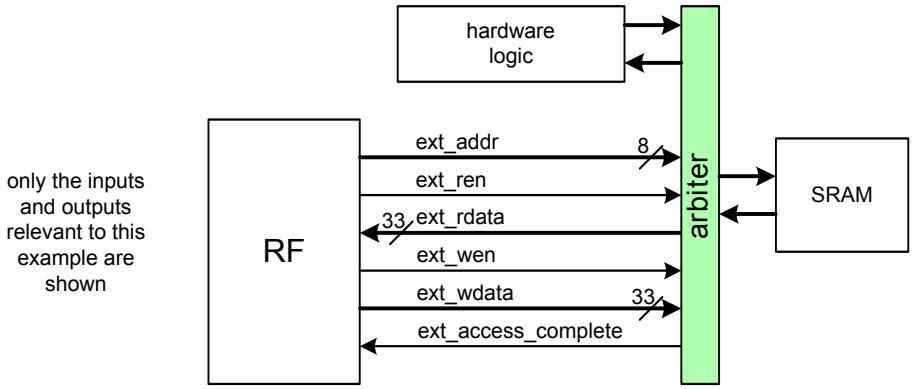


Figure 2-34: External Ramblock and Arbiter

One obvious application is to use the external ramblock interface to connect 3rd part RFs, which can be found in some IP blocks, because the interface can be translated easily for this purpose.

Due to the flexibility of the interface it allows implementing special features in the hardware with very small development effort. In a complex hardware unit it may, for example, be necessary to log more than one condition that causes errors. A data structure is required to store multiple of these condition, the easiest way is to use a FIFO for this purpose, because a FIFO provides a simple interface to store information.

A FIFO as depicted in figure 2-35 is a data structure that fulfills the task as described by its name. Data can be stored in a first in, first out fashion. The *shift_in* triggers the storage operation of data connected to the *d_in* (data input). The *empty* output indicates if the FIFO is empty or not, as soon as the *empty* output is 1'b0, valid data is available at the *d_out* (data output). Entries can be removed from the FIFO by asserting the *shift_out* signal.

With a FIFO wires that are relevant to debug the possible error condition can be connected to the data input of the FIFO and the error condition itself can trigger the *shift_in* of the FIFO. The FIFO could be full at some point, it is a design decision what should happen in that case, either the error logging could stop or old entries could be removed by using the full signal for shifting out old entries.

The RF interface that allows this implementation is instantiated in the RF with the following XML element. This example also shows the usage of an addrsize of "0". By setting the addrsize to "0" RFS will not provide an ***addr*** bus on the interface, because it would have no use in this case.

```
<ramblock name="elf" addrsize="0" ramwidth="64" sw="ro" hw=""
    external="1" desc="Error logging FIFO" />
```

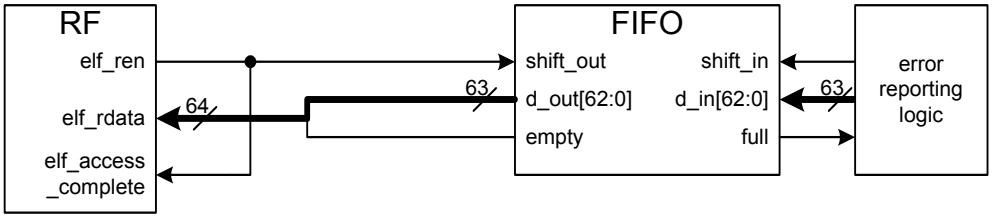


Figure 2-35: Error Logging FIFO

A block diagram of this mechanism is shown in figure 2-35. It shows also the limitation of this approach, the width of FIFO can only be 63, because the 64th bit of the interface is used to connect the empty signal. Thus, the software reading out the FIFO will be able to see the empty signal and can read from the memory address as long as the 64th bit is 1'b1.

Each external ramblock uses by default extra output registers for all interface signals, because this has positive effects on the timing. However, this does also require a lot of resources.

Thus, the possibility was added to share registers for multiple external ramblocks, this is done by setting the **shared_bus** attribute to "1". The default for this attribute is "0". By doing so, the *wdata* and *addr* registers for all external ramblocks, which have the shared_bus attribute set, are a shared. It is not possible to share the control signals.

The *wdata* and *addr* outputs may have different sizes of course, RFS is taking care that the shared registers are big enough and that the interface on the outside of the RF module is the same like without the shared_bus attribute.

2.4.6 Regroot / Regroot Instance

Each regroot has to be stored in an own file. Consequently resulting C header files and Verilog files can be named exactly like the XML file that contains the regroot. A regroot can contain the following elements:

- rrinst
- reg64
- aligner
- ramblock
- repeat
- placeholder

The regroot element itself does not have any attributes. In the example in chapter 2.4.1 it was already shown that regroots have to be instantiated with the rrinst element.

The task of the regroot instance (rrinst) is to be able to instantiate regroots. Rrinst is the short form for regroot instance. It is possible to instantiate regroots multiple times. The rrinst has two required attributes:

- name
- file

Each instance of a regroot must have a name, so that it can be addressed and put into the hierarchy. It can be named like the XML file that contains the regroot, but there is no direct relation, it can be named in any way.

The file attribute points to the XML file with the regroot, the filename can be either relative or a full path. Relative paths are from the location where the XML file is stored that has the rrinst element.

The optional **external** attribute is the most interesting one, when set to "1" the regroot will not be instantiated within the RF, where the rrinst element is used, but an interface will be provided to connect it externally. This is depicted in figure 2-36. The external attribute can be combined with the **shared_bus** attribute which was introduced in the last subsection. This attribute has in the case of external regroots the same functionality like for external ramblocks.

There are multiple advantages associated with using a regroot external:

- less wiring
- hardware modules are more portable
- place and route advantages

Verilog modules are connected by wires. Wiring is done in the Verilog file that contains the modules, and it can be extensive in big designs.

In figure 2-36 on the left side the six lines between unit1 and sub_RF1 as well as between unit2 and sub_RF2 symbolize the big amount of wiring that is often necessary. All this wires have to be routed through the top file.

On the right side of the figure the sub_RF1 and sub_RF2 are used as external instances and can therefore be integrated directly in the units. These sub-RFs are connected to their parent RF with the standard RF interface as described earlier.

Therefore, lots of tedious and error prone wiring is not needed at the top level. All wiring for the hwregs and ramblocks can stay within the actual module where the elements are required. So lots of work and code can be saved.

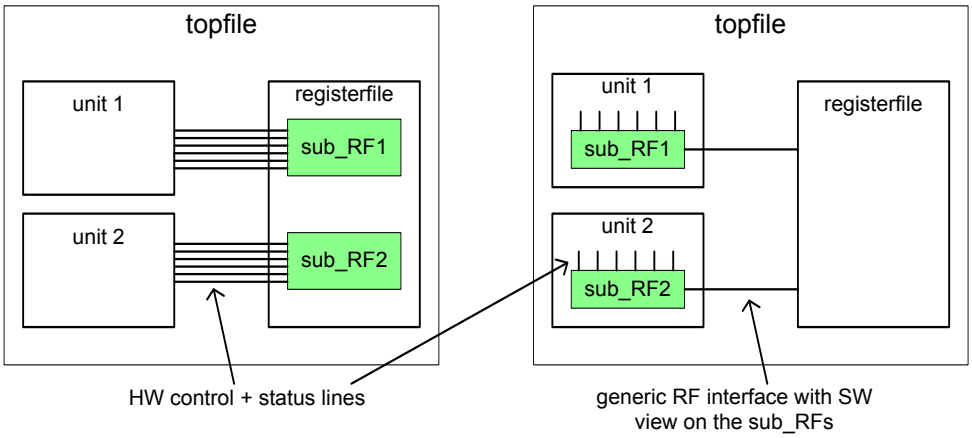


Figure 2-36: Comparison with and without External Regroots

When external regroots were introduced in the EXTOLL design the size of the top module was reduced from 3337 lines of Verilog to 2109 lines.

Additionally, in many cases the Verilog topfile can stay unchanged, if changes are made in the RF of a module, because the changes are contained within the module that instantiates the external sub-RF. The connection between the RF and its sub-RFs is parameterized, therefore even when the widths of the data or address paths change, no Verilog source changes are required.

Moreover, when the RF is inside the unit, then also the XML file can be maintained together with the unit. Consequently, units are more portable between different designs, because the unit itself and the XML definition of the RF are maintained together. Furthermore, the module and its RF can be changed without causing any effects on the code of the complete design.

Placing the RF components within the modules improves the results that can be achieved with the place and route tools.

For big ASICs designs partitioning is mandatory because the EDA tools can only handle logic up to a certain size. Components of the planned EXTOLL ASIC and a possible partitions scheme is depicted in figure 2-37. The hierarchical partitioning of the RF allows crossing the partitions with a relatively low number of wires. A part of the partitioning can be seen in the RF structure as shown in figure 2-45 on page 88, especially the big **nw** (network) sub tree can be seen there.

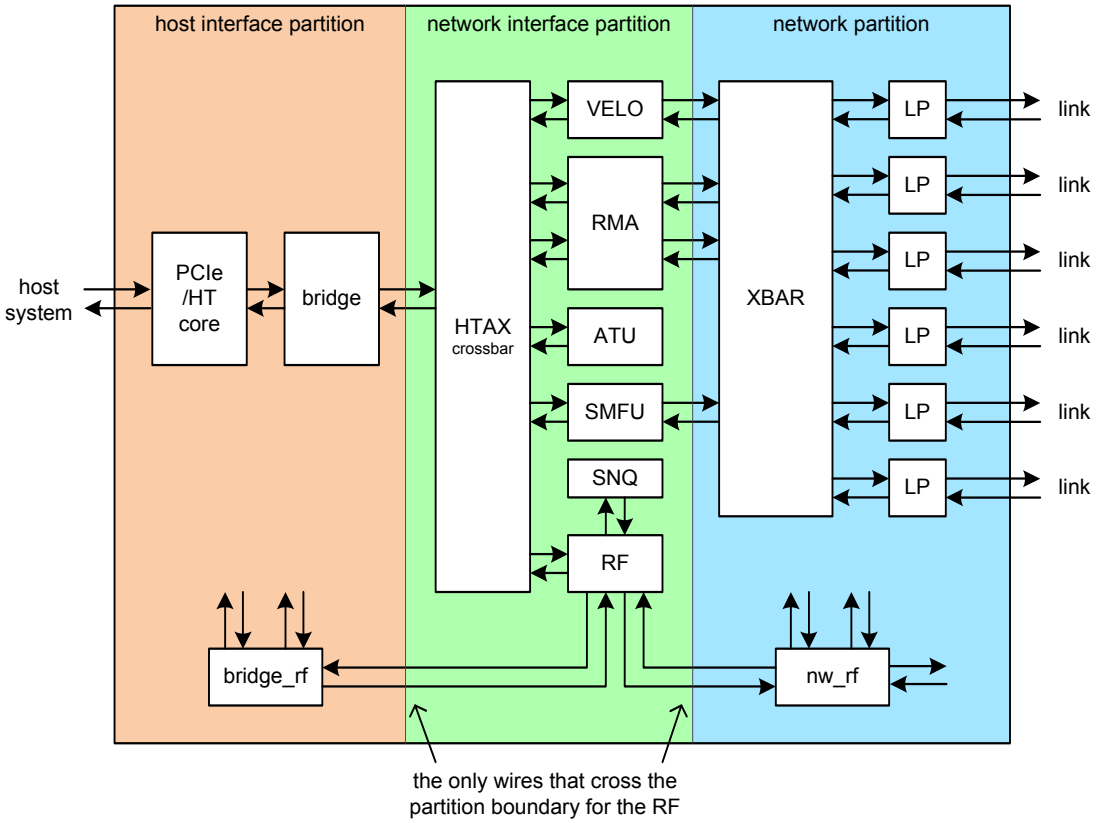


Figure 2-37: EXTOLL Partitions

The different partitions have to be connected and therefore the physical placements in the GDSII [122] have to be aligned so that the data paths that cross over the boundaries of partitions are connected. Figure 2-38 illustrates the issue. Therefore it is useful to have in each partition a single sub-RF that contains all further sub-RFs in the respective partition, because this reduces the number of connections between the partitions.

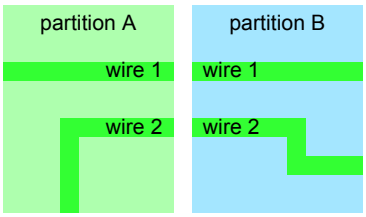


Figure 2-38: Physical Placement Illustration

In summary, the external regroots allow the hierarchical decomposition as described in the Introduction of this chapter and depicted in figure 2-10.

2.4.7 Multi Element Instantiation

Often designs have modules that are replicated multiple times and all these modules have to be controlled. For example EXTOLL has, depending on the underlying hardware, between four and seven link ports.

Another often seen design pattern is that one unit requires multiple registers with exactly the same properties.

The obvious solution is to replicate the entries multiple times and slightly change their name by adding for example a number. This approach has two disadvantages. First, the input XML is bigger and therefore more complex to maintain, this is only a minor disadvantage. Second, there will be no regular structure that the software can take advantage of.

The alternative to replicate the entries in the XML specification is to use a **repeat** block, because it provides the possibility to instantiate certain RF elements multiple times.

A repeat block can contain the following sub elements:

- reg64
- aligner
- placeholder
- ramblock

In the following example a repeat block will be shown that repeats the inner elements two times. The ramblock in this example is very small, because otherwise figure 2-39, which shows this repeat block on the left, would result in a too big figure.

```
<repeat name="rep" loop="2">
    <reg64 name="blue">
        <hwreg width="16" sw="rw" hw="ro" />
    </reg64>
    <ramblock name="green" addrsize="2" ramwidth="64" sw="wo" hw="ro" />
    <reg64 name="red">
        <hwreg width="16" sw="rw" hw="ro" />
    </reg64>
</repeat>
```

The most important attribute of the repeat block is the **loop**, because it decides how often the element in the repeat block will be instantiated. This XML code will instantiate in the Verilog: four reg64 named rep_0_blue, rep_0_red, rep_1_blue and rep_1_red; ramblocks named rep_0_green and rep_1_green. However, in the C header files the advantage of the repeat block will get apparent.

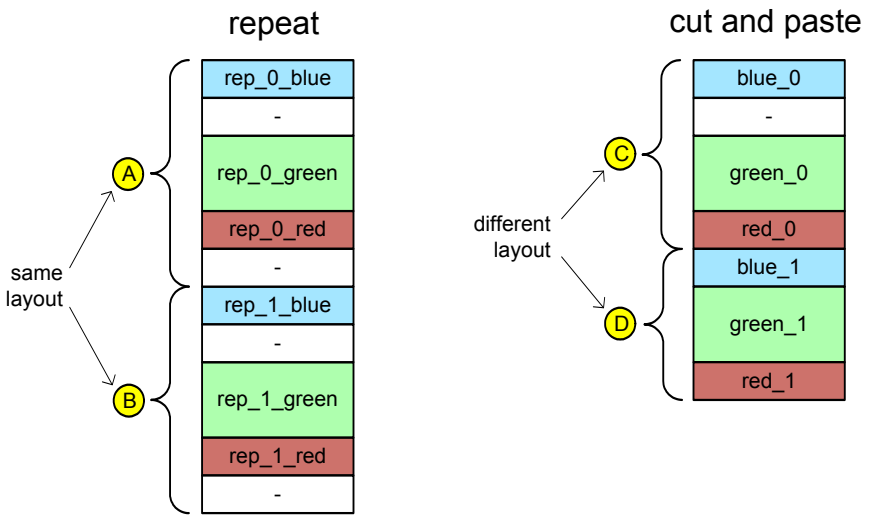


Figure 2-39: Repeat Block Alignment

```
struct rep {
    volatile uint64_t blue;
    char pad1[8];
    volatile uint64_t green[2];
    volatile uint64_t blue;
    char pad2[8];
};
...
struct rep rep[2];
...
```

The repeat block does appear in the C header files as an array of structs and therefore it is possible to use an index to access the different elements in the struct. This is the big difference from instantiating elements multiple times by cut and paste.

RFS does align the elements within the repeat block in a regular way so that it is possible to access the element with an array of structs. In figure 2-39 the two blocks of elements are marked with **(A)** and **(B)** with their alignment. Elements are instantiated by cut and paste in the following way:

```
<reg64 name="blue_0">
    <hwreg width="16" sw="rw" hw="ro" />
</reg64>
<ramblock name="green_0" addrsize="2" ramwidth="64" sw="wo" hw="ro" />
<reg64 name="red_0">
    <hwreg width="16" sw="rw" hw="ro" />
</reg64>
<reg64 name="blue_1">
    <hwreg width="16" sw="rw" hw="ro" />
</reg64>
<ramblock name="green_1" addrsize="2" ramwidth="64" sw="wo" hw="ro" />
<reg64 name="red_1">
    <hwreg width="16" sw="rw" hw="ro" />
</reg64>
```

As a result, the address setup will look like on the right side in figure 2-39, as it can clearly be seen, **(C)** and **(D)** do not have the same layout, because of the alignment that is necessary for the ramblock. Therefore, it is not possible to create structs in the C header files without the alignment that is done by the repeat block.

RF XML files are in some cases used in different designs that have slightly different requirements, and therefore the loop attribute may depend on the design. Consequently the following elements in the RF will start at different addresses. This is not always appreciated. There are different possibilities to solve this issue:

- the loop parameter is the same in all designs
- aligner
- special functionality

Keeping the loop parameter the same in all different designs, so that the RF components are added to the Verilog code will work of course, but it wastes resources. Though it depends on the exact circumstances, the synthesis may be able to optimize all surplus components away. Nevertheless, it is not a good idea to rely on optimization.

Another possibility is to use an aligner to align the next element, but this requires knowledge of the actual address setup. Of course this information can be extracted easily from the generated documentation, but it is contrary to the requirement that the developer should not have to be concerned about addressing.

Therefore, special functionality was required to solve this problem. The **maxloop** attribute does reserve space for the number of iterations given by the attribute. Thus, the start address of the next element in the RF stays constant independent from the actual loop attribute. In the C header files the respective amount of padding will be added. The value of the maxloop attribute has to be bigger or equal than the one loop attribute of course.

2.4.8 Placeholder

Hardware designs are often used on different hardware platforms, for example FPGAs and ASICs, but from a software perspective it is desirable to have the same address setup with all implementations. However, in some cases different implementation may require different RF elements. Thus, the placeholder element was added, which can be used to even out some of the differences between the implementations and keep the address setup consistent.

EXTOLL is going to be implemented for different platforms, and these cause also different requirements for the RF. Therefore, some implementations have more or other elements in the RF, but from a software perspective the RF interface should be everywhere the same.

Therefore, the placeholder element was added to RFS, because it allows exactly what the name suggests.

For example, one difference between the FPGA and the ASIC version of EXTOLL is that the ASIC version actually has to make use of the ***sec*** (single error corrected) and ***ded*** (double error detected) outputs of the RAMs; the interface of such a RAM module (ram_1w1r_1c) can be found in chapter 2.4.5. Therefore, the ASIC version should also provide the information about these errors in the respective RF of the module, but the FPGA version should not have the reg64 to avoid wasting resources.

With the help of the placeholder the problem can be solved in the XML definition.

```
#ifdef ASIC
    <reg64 name="ram_ecc_errors">
         <hwreg name="sec" sw="rw" hw="wo" width="1"
              sw_write_xor="1" sticky="1" />
         <hwreg name="dec" sw="rw" hw="wo" width="1"
              sw_write_xor="1" sticky="1" />
    </reg64>
#else
    <placeholder num_reg64="1" />
#endif
```

The placeholder will occupy the same amount of address space like one reg64, and therefore the address setup of the following RF elements will be the same in the FPGA and ASIC version. Besides the **num_reg64** attribute it is also possible to use the **addrsize** attribute to reserve the space for a ramblock of the given size.

When the placeholder is used with the addrsize attribute, then the alignment is exactly like in the case of a ramblock of the respective size. However, in the case of the num_reg64 attribute there is no alignment.

2.4.9 Trigger Events

In big hardware and software designs it is very useful to have a method to inform about events in the hardware. These events can be received either by hardware or software.

Details about handling such events in software and examples for this will be detailed in the next chapter about the System Notification Queue (SNQ). It will describe a special unit that will not only send an interrupt to the software to inform about an event, but also directly send information that is related to this event into a queue that is stored in the main memory.

All information that is interesting for the software or that can trigger events has to be accessible in the RF, especially when also the software is supposed to be able to access it. Accordingly, an event is nothing else than the change of a RF value. Therefore, the RF is the right place to detect these changes and trigger further actions.

Furthermore, the XML specification of the RF is the right place to define which changes should trigger an event, because the code that has to be generated to detect the changes is best placed in the RF.

The event is called trigger event (TE), each TE has a name for identification.

Only elements of the type hwreg can be used to generate TEs. The **te** attribute has to be used to mark hwregs as TE sources.

```
<hwreg sw="rw" hw="rw" name="test1" width="16" te="eventA"/>
```

The hardware will emit eventA, if this register is changed by the hardware. RFS does generate Verilog code that keeps a shadow register and compares the content of the register every cycle.

When the hwreg is implemented either as counter or with the hw_wen attribute, then the respective special inputs are used to detect the change. This saves the resources for the shadow register and the comparator. As a result an event will be emitted, even when the content of the register does not change, when the hw_wen attribute is set and the hw_wen input for the hwreg is asserted.

RFs can be instantiated from the same XML file in different places within the RF hierarchy, thus they will also have the same te attributes. These sub-RFs may be within different modules that should not share the same events. The first and obvious solution would be to copy the XML file to another filename and change the te attribute, so that it takes the place within the hierarchy into account. This solution would result in two slightly different XML files that both have to be maintained. Therefore, another solution was found, the te attribute does support two special strings:

- **%n** complete hierarchy
- **%i** iteration number

The %n will be replaced by the complete hierarchy name of the RF, where the te attribute is used, and the %i string is for repeat blocks, it will be replaced by the current iteration number.

A **te** attribute can also be responsible for multiple events by assigning multiple event names to the **te** attribute, separated by commas.

All TEs are written to an output file that is named snq_te.txt. Each line of this file is for one TE.

```
<number of the TE> <name of the TE>
```

The number of the TE corresponds to the wire in the *trigger* bus that will be provided by the top level RF. In figure 2-40 a RF is shown that makes use of TEs, the corresponding snq_te.txt file would have the following content.

```
0 A
1 B
2 C
3 D
```

Trigger signals are pulsed and therefore are asserted only for a single cycle.

To reduce the manual effort required to interconnect the modules. It was decided to use a single data path to transport the trigger signals between the different hierarchy levels of the RF. This is depicted in figure 2-40.

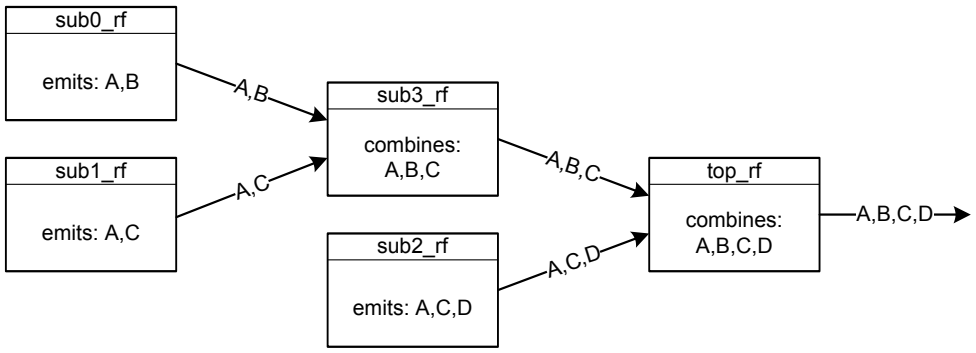


Figure 2-40: Trigger Tree

This figure does also show that the trigger signals are propagated from the sub-RFs to the **top_rf** and on the path the different event that the same name will be combined to a single trigger wire.

2.4.10 Annotated XML

The annotated XML is required by rgm and the register file driver generator (rfdrvgen) and it was already introduced shortly in chapter 2.3.3.

RFS is adding attributes to the elements in this file, which contain addressing information. All added attributes are prefixed with an underscore. Therefore the **_absoluteAddress** attribute is added to the following elements when they are outside of a repeat block:

- regroot
- repeat
- ramblock
- reg64

Elements that are within a repeat block do not have an absolute address, because they are used multiple times. Therefore, the repeat block has a **_iterSize** attribute that describes the size of a one complete iteration of the repeat block including alignment. All elements within the repeat block have an **_offset** attribute that describes the offset within the repeat block.

2.5 Implementation of RFS

The implementation of RFS requires the possibility to parse XML files and to write out XML files, these requirements can be fulfilled by quite a lot of programming languages and environments.

In the end C++ [73] in combination with QT [68] was chosen. The reason for this is the good prior experience with the XML functionality of the QT library.

For parsing of XML in C++ a viable alternative would have existed in the form of libxml2 [69], but QT also provides other useful functionality that makes C++ development easier.

Execution speed or memory requirement were not an important consideration in the course of the implementation, the priority was the implementation time and correctness. The task of RFS is for a modern CPU so simple that the execution time is only limited by the speed of the storage device where the inputs are read from and the outputs are written to. For example, the generation of the EXTOLL RF takes 0.2 seconds. In the end it does not matter if the RF generation takes 0.1 or 30 seconds, when considering the hours that will be required to generate a bit file in the case the code is employed in an FPGA or weeks for the ASIC implementation.

In the development process of EXTOLL R2 it got apparent that multiple different RFs will be required for different implementation targets. In the case of EXTOLL R2 there are at least the following development targets:

- ASIC
- Xilinx Virtex 6 FPGA with HyperTransport and 6 link ports
- Xilinx Virtex 6 FPGA with HyperTransport and 4 link ports
- Xilinx Virtex 6 FPGA with PCIe and 4 link ports
- FPGA small implementation to reduce the turn around speed

These different targets have different requirements regarding the RF, because for example the number of link ports is different, they can contain an HT core or not.

Therefore, a method had to be found that allows generating target specific RFs from a single set of XML files. For example, the following properties can change depending on the configuration:

- width of registers
- number of instantiations
- registers or whole sub-RFs have to included or excluded

Theoretically it would be possible to change RFS so that it could do this depending on a configuration file. However, it would be a lot of effort. Fortunately, the C preprocessor (CPP) can do exactly what is needed, because it can replace text and include or exclude parts with the help of #ifdef/#endif blocks. The only minor problem is that CPP is not able to understand Verilog defines, therefore a Perl script was developed that converts the Verilog style defines to C defines and combines all defines into a single file. This allows to use the same Verilog defines, which are used in the Verilog code, to control which elements are included in the RF. By using only a single configuration source for both, a possible source of errors is eliminated. The workflow is depicted in figure 2-41.

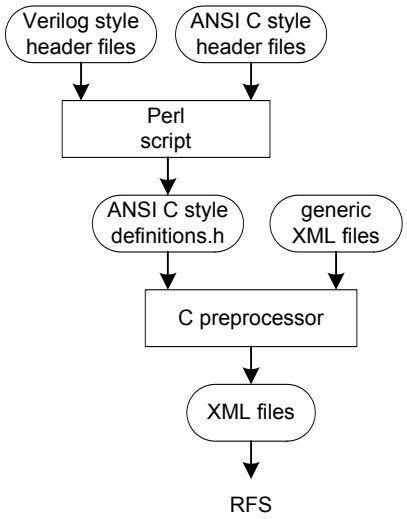


Figure 2-41: RFS Workflow with CPP

2.6 Supporting Infrastructure

The generated RFs are by themselves only of limited usability, they require infrastructure so that they can be used efficiently.

2.6.1 HyperTransport on Chip to RF Converter

In the designs, which are developed at the CAG, an on chip network is used to interconnect the different functional units with the host systems. The on chip network is used with the HToC protocol.

Thereby, a unit is required that translates this protocol to the interface of the RF, and this is done by the HT to RF converter.

In figure 2-42 the relevant components are shown, the converter translates between the HT protocol and the RF interface, and also provides a RF interface for special extra units. The converter will arbitrate the access to the actual RF.

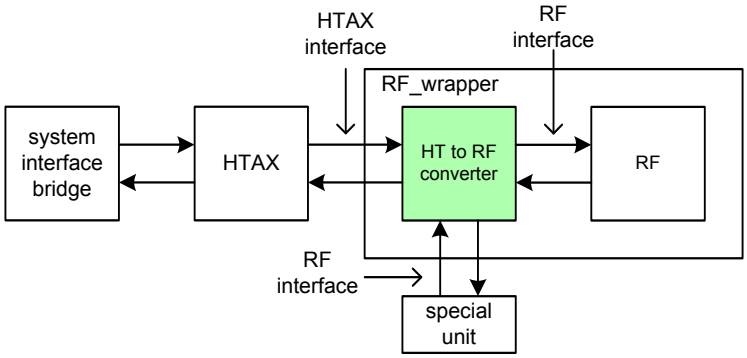


Figure 2-42: HT to RF Converter

The HT to RF converter also implements a time-out mechanism in the case the RF does not answer with the *access_complete* signal in time. In the generated RF itself this can not happen even when wrong addresses are used, but when an external RAM interface is used, then external logic has to handle the read and write access. This logic could have bugs, then the time-out prevents that the host system is affected by this bug. Otherwise, it is possible that a read that does not return will trigger a machine check exception (MCE).

When a write does time-out the converter just does nothing, because it has no way to report errors back, but when a read times out, then all bytes in the read response are set to 0xFF and an error bit is set in the HToC packet header.

2.6.2 Debug Port

The main development target of the EXTOLL project is the development of an ASIC. In such a chip there are many configuration options that can be accessed by the RF. The problem is that these can only be accessed by the host system if the interface to the host system is working.

Not only the PCIe but also the HT interface to the host system has numerous configuration options and it is, while the ASIC is under development, not entirely clear if the defaults will be correct.

Therefore, a way is required to be able to load parameters into the RF before the connection to the host system is working. Furthermore, it may also be necessary to access the RF for debugging purposes as long as the host interface is not functional.

Consequently, the debug port was developed and presented in [37]. It provides two possibilities to modify or access the RF without the interface to the host system:

- SPI
- I2C

It is able to read data from an **SPI** flash memory [123] with an SPI master controller. The SPI flash memory can contain addresses and data tuples that will be used to write entries in the RF. Additionally, it also provides an **I2C** bus slave that can be used to read and write RF entries.

The debug port uses the “special unit” interface of the HT to RF converter to access the RF as depicted in figure 2-43.

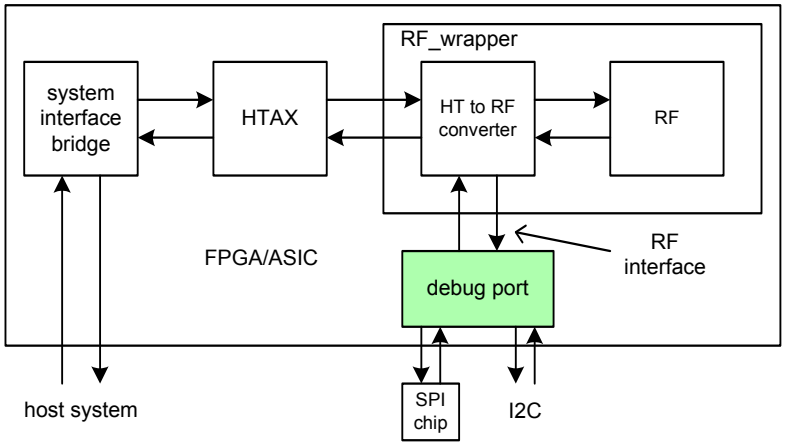


Figure 2-43: Debug Port

2.6.3 OVM Components

The verification environment has to be able to read and write the RF, exactly like the software that runs on the host system. Therefore, the verification environment should be able to use a naming scheme instead of addresses, because the addresses can change with any change in the RF.

For this purpose rgm can be used, it is the abbreviation for register and memory package. This tool uses the annotated XML file and generates OVM components.

2.6.4 RF Driver Generator

The sysfs [77] file system is a virtual file system of the Linux kernel and provides system information and settings. It does not use any storage device, and all files and directories are generated on the fly. These files can be handled like normal files, it is possible to read and write them. On a standard Linux systems the sysfs file system is mounted under /sys.

It is very useful for debugging purposes to be able to read and write registers without the need to modify, for instance, a C program each time.

The RF driver generator (rfdrvgen) uses the annotated XML file to generate a kernel module that provides a directory within this /sys hierarchy. This directory contains files for each reg64 and ramblock, so that these can be read and written.

The following example demonstrates how cat and echo can be used on the shell to read and write registers.

```
root@computer:/sys/../net_rf# cat tcp_rf_target_mac
addr: cd7e7a211b00; valid: 1;

root@computer:/sys/../net_rf# echo 0xAABBCCDDEE > tcp_rf_target_mac

root@computer:/sys/../net_rf# cat tcp_rf_target_mac
addr: aabbccddee; valid: 0;
```

The complete path to the net_rf directory was omitted for brevity. Additionally, the driver generator is also producing C, TCL and Python code to access the RF.

2.7 Application of RFS

Multiple designs were already done with the help of RFS. In the following two of these designs will be presented, specialties and results regarding their RF will be outlined.

2.7.1 Shift Register Read and Write Logic

Some hardware designs are using shift register chains instead of a register file as described in this chapter. There are multiple reasons for using them in a design, but on the other side there are of course also disadvantages.

The low usage of resources is the main advantage. Shift registers consist usually of flip-flop chains. Therefore, no big multiplexers are required to read out the different registers directly. It is the reason why shift registers are often used in custom analog designs, because it is relatively easy to implement them efficiently at such a low level.

Obviously, shift registers have the big disadvantage that they are very slow, because they handle only a single bit per cycle and furthermore in long shift register chains also many bits have to be read or written. So changing a single bit can take hundreds of cycles.

Hardware designs that make use of shift registers have a chain of flip-flops to get data in and out of the design, but usually the content of these flip-flops does not have a direct effect on the logic before an update is done.

Figure 2-44 tries to visualize such a setup. Writing of data is done bit for bit over the *bitin* input. Before the data in this chain has an effect on the design an *update* signal has to be asserted that will take over the data in the actual design. Reading data works similarly, first a scan signal has to be used to transport the data from actual logic into the chain, and then it can be read bit for bit by sampling the *bitout*.

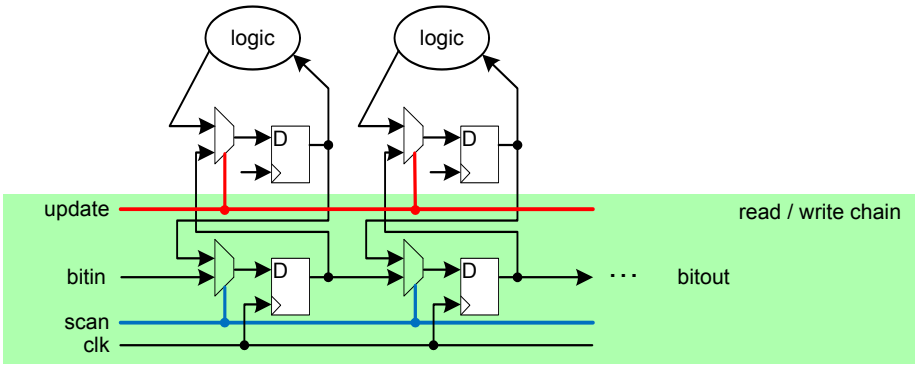


Figure 2-44: Shift Register Read/Write Chain

An example for such a design is the Spadic [88], a read out chip for physics experiments. The project website has more details and publications [97].

The features in RFS allow developing glue logic that has only a very small overhead to read and write such shift registers.

The RF for the interface consists of two elements. Foremost a control register, which is used to tell the glue logic if a read or a write is to be performed and the size of the respective chain. Additionally, an external ramblock is used to get the data to and from the chain.

```
<reg64 name="control">
    <hwreg name="mode" width="2" sw="wo" sw_written="1" hw="ro"/>
    <hwreg name="chain" width="1" sw="wo" hw="ro"/>
    <hwreg name="length" width="11" sw="wo" hw="ro"/>
</reg64>
<ramblock name="data" addrsize="5" ramwidth="16"
    sw="rw" hw="rw" external="1"/>
```

Using an external ramblock to write the data to the unit provides two crucial advantages. First, this allows making use of the *access_complete* signaling, and therefore it is no problem when the writing or reading takes multiple cycles. The ramblock is 16 bit wide, so every read or write to it will put 16 bit in the shift register or read out this amount of data.

Second, it allows providing consecutive addresses. Subsequently, this allows it to implement higher level units that do reads and writes bigger than the size of a single register. Such writes carry a start address, the size and the actual data payload.

Writing is done in the following way from the software interface of the register file:

1. write the control register with the right mode, chain and length information
2. write the amount of bits given in the length information to the ramblock

Reading works very similar:

1. write the control register with the right mode, chain and length information
2. read the amount of bits given in the length information from the ramblock

2.7.2 EXTOLL RF

To control all the features of EXTOLL a relatively big register file is required. This RF consists of many sub-RFs and makes use of most RFS features. The RFs for EXTOLL are the biggest RFs, which are currently generated with RFS.

There are multiple different implementations of EXTOLL, table 2-7 shows two of them, the 64 bit Xilinx Virtex 6 FPGA is the current main development platform and the one for the ASIC is at the moment the biggest one.

Resource type	64 bit FPGA	ASIC
Number of regroots	52	73
Number of reg64	494	1036
Number of hwreg elements (without counters)	1010	3219
Amount of bits in these hwregs	11769	26055
Number of counters	257	677
Amount of bits in these counters	6292	18440
Number of ramblocks	56	71
Amount of bits that can be addresses within these	601152	2818752

Table 2-7: Statistics for Two EXTOLL RFs

Figure 2-45 shows a few of the 73 regroots of the ASIC version in their hierarchical order. The FPGA version is very similar, but has for example fewer crossbar (XBAR) ports.

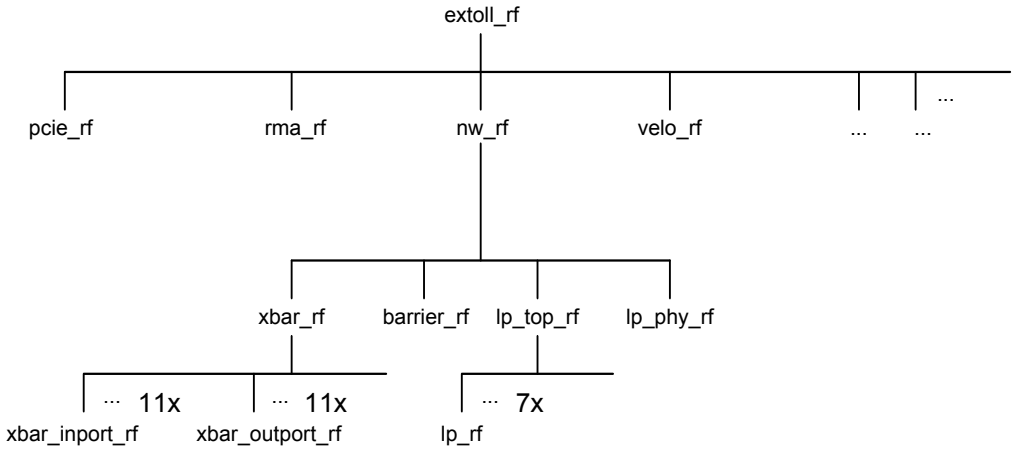


Figure 2-45: RF Hierarchy of EXTOLL

The numbers show that the average counter is in the FPGA version 24.5 bits wide. These 257 counters are implemented in DSP slices.

The screenshot from Xilinx PlanAhead in figure 2-46 shows the EXTOLL implementation for the Virtex 6 xc6vlx240t. Rectangles that are black, red or blue are used. The red ones are RF components, the blue ones delay elements and the black ones are the other logic. It can be observed that the RF components are distributed in between the other logic. Unfortunately, it is hard to see in the screenshot, but basically the complete FPGA is used; only the upper left and right corners are empty.

2.8 Optimization Efforts

In the course of the development of RFS it got apparent that the big amount of RF components has a relevant resource impact. Especially in the case of the FPGAs and big designs, for this reason, ways had to be searched that allow reducing these resource requirements. In the following, multiple techniques are explained and their impact on a big design is evaluated. The underlying technology is a Xilinx Virtex6 FPGA and the design is EXTOLL.

The crossbar of the EXTOLL design will be the object of this optimization effort, because it contains a lot of RF components and a big part of the resources, which are used by the crossbar, actually is occupied by them. Namely the crossbar contains one top RF and 16 sub-RFs and for timing reasons also two stages of delay elements between the top RF and the sub-RFs. Delay elements are modules that are pipelining the interface path with the help of registers to relax the timing, but of course they occupy flip-flops for this purpose. The

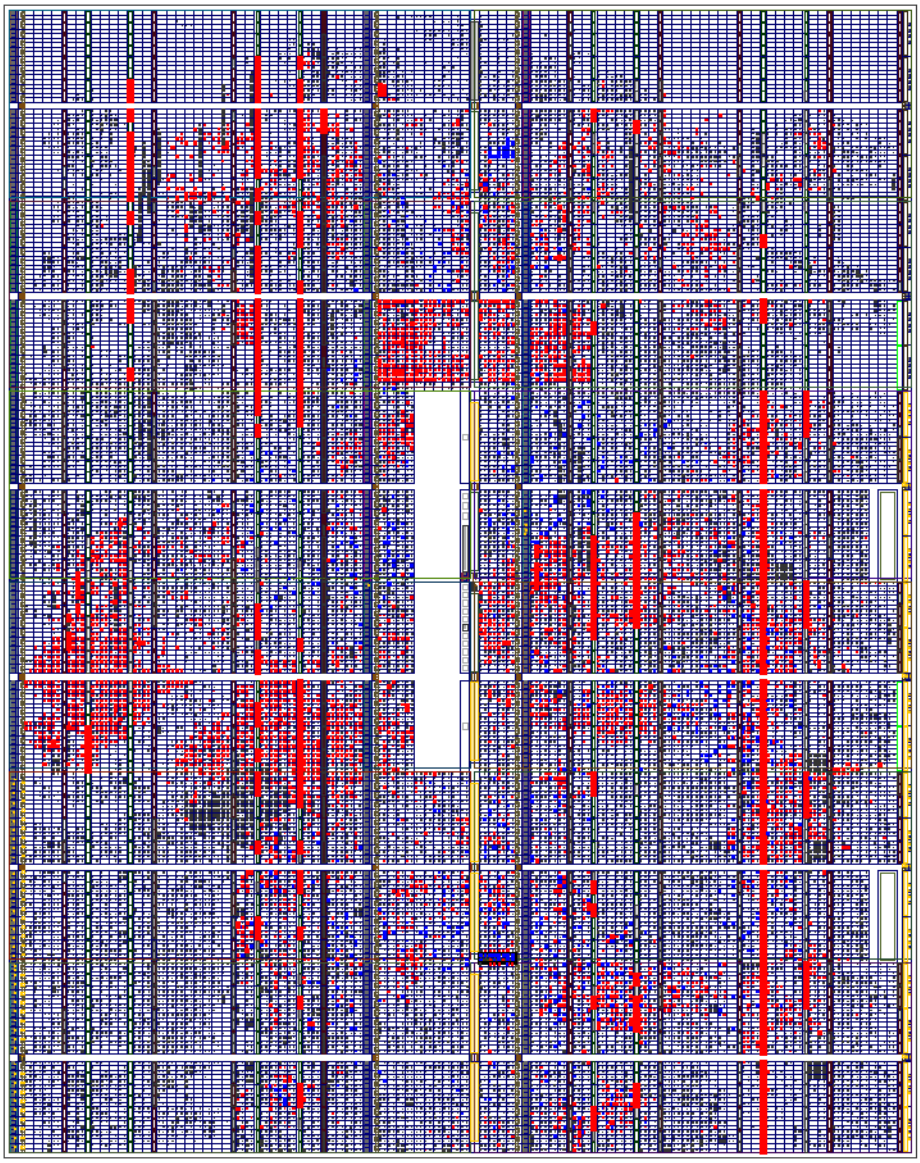


Figure 2-46: EXTOLL PlanAhead Screenshot

interfaces between a RF and its sub-RF do allow this, because read and write operations are initiated by the *read_en* or *write_en* control signals and the completion is indicated by the *access_complete* signal. Thus, each of these signals can be delayed without a problem when the accompanying data buses are delayed in the same way.

Figure 2-47 shows such a delay element; the widths in this figure are taken from the delay elements inside the crossbar. It is also possible to use multiple delay elements after each other. In the case of the EXTOLL crossbar two delay stages have been used, because otherwise it was not possible to reach timing closure.

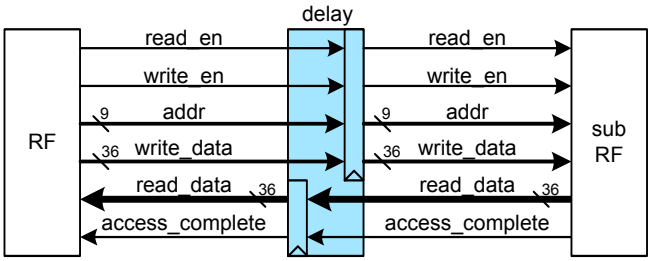


Figure 2-47: Delay Element

Delay elements are not instantiated automatically within the respective RFs, because this would reduce their effectiveness in reducing timing problems. Figure 2-48 shows a typical usage scenario for a delay element. It is between two placement bounding boxes, which may be far away from each other and one or more delay stages are used to relax the timing on the path between these two boxes. The Xilinx FPGA tools would not be able to do this if the delay element would be either part of the top RF or the sub_RF, because then the delay registers would be within either of the bounding boxes.

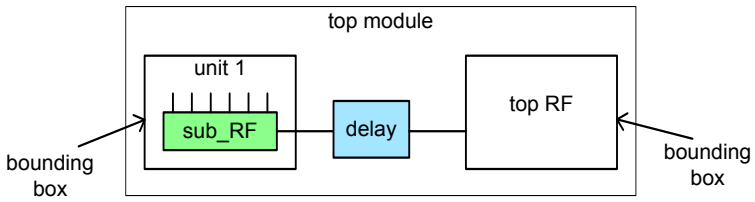


Figure 2-48: Placement Delay Module

Figure 2-49 depicts only the RF components inside the EXTOLL crossbar. There is one RF on the top level of the crossbar. The crossbar is an 8x8 design and consists of eight incoming (inport) and eight outgoing ports (outport), each of these ports has an own RF. These RFs are used for status information, statistics and settings. In the inports the RF is also used to initialize the routing tables. The crossbar together with 17 RFs and 32 delay modules requires a lot of resources.

This configuration is the baseline for the optimizations that are going to be performed to reduce the resource requirements. The resource requirements for this baseline and the following optimization steps are shown in table 2-8 on page 94.

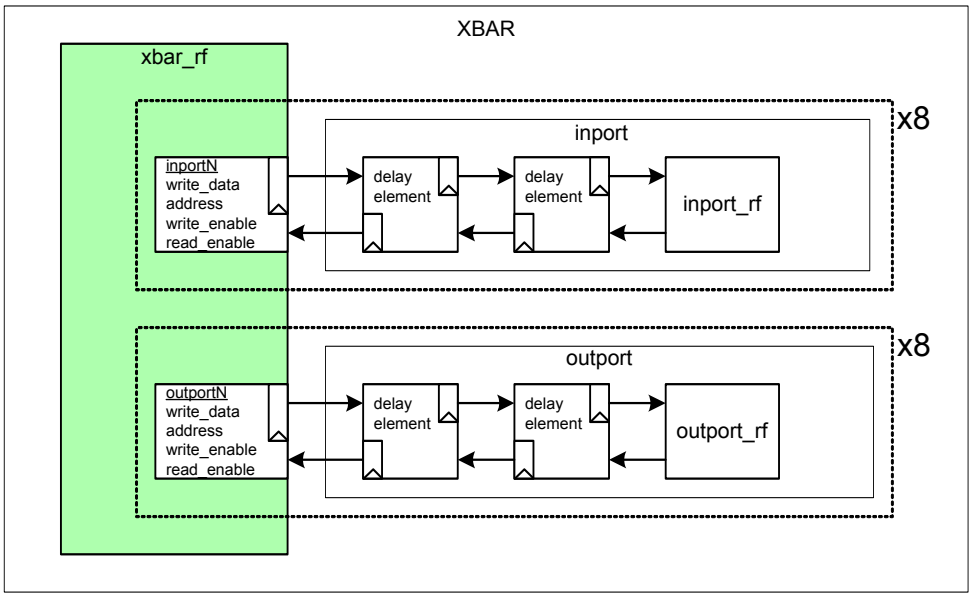


Figure 2-49: Crossbar RF Structure

The low number of flip-flops (FF) used is diluting, because it looks like it is no problem to use them, because there are two times more FFs than LUTs in the Xilinx Virtex 6 (301440 versus 150720), but due to the architecture of the FPGA the usage of a FF often means that the logic resources inside the slice can not be used. In Virtex6 FPGAs the SLICEs [112] are containing the components as they can be seen in the annotated screenshot from Xilinx PlanAhead [104] in figure 2-50. PlanAhead is a tool that can be used for floor planning and design analysis.

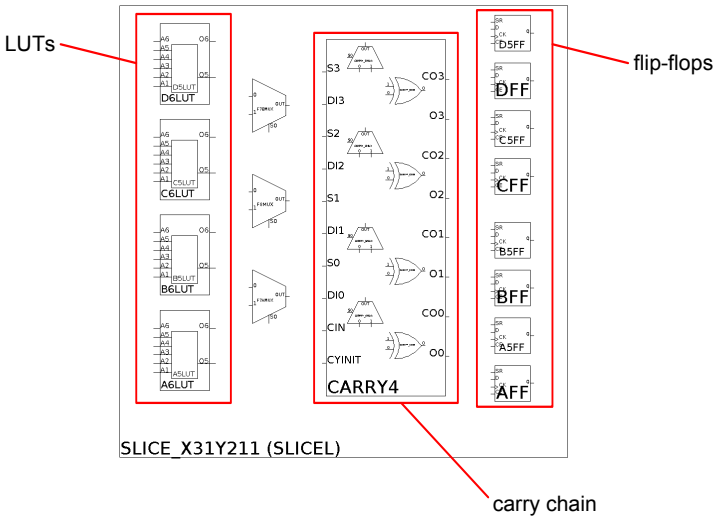


Figure 2-50: Xilinx Virtex 6 SLICEL

In figure 2-51 a typical section of the crossbar in the PlanAhead device view is shown. The red squares are FFs, and these are mainly delay elements. The other components of these slices are completely unused, because the delay elements do not need the logic resources. Consequently, the logic resources in these slices are lost and can not be used.

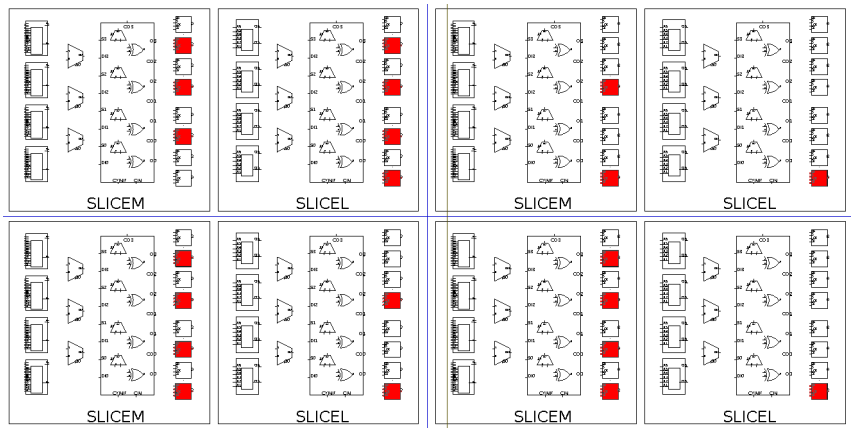


Figure 2-51: Slices Close up View

Therefore, it is important to reduce the usage of flip-flops. One possibility, which will be detailed in chapter 2.8.3, is to use multi cycle data path and pipelined synchronized control signals for the handshake.

2.8.1 Step 1

The baseline RF was created with an earlier version of RFS that generated all read and write paths 64 bit wide. In a first optimization step RFS was improved, and the widths of the different wires between RFs and their sub-RFs was limited only to the really required width. RFs can contain elements with different widths, but the software accessible parts do not have to be 64 bit in width, they also can be smaller. This optimization effort was expected to save resources, because fewer registers and fewer routing resources are required.

In the end no resources were saved as it can be seen in table 2-8, the reason for this is that the Xilinx tools were able to detect, which bits are required and optimize the hardware accordingly.

2.8.2 Step 2

The analysis of the RFs that are instantiated in the crossbar has shown that the RFs in the crossbar inports contain three reg64s and three ramblocks. All elements in the RF besides one reg64 are only 36 bit wide. This reg64 contains two 32 bit counters and is 64 bit wide, and therefore the read and write paths to this sub-RF have to be 64 bit wide. Consequently, also the read case has to produce a 64 bit wide output with a 64 bit wide multiplexer.

This motivated the idea to split up this reg64 into two, because this would limit the width of the *read_data* and *write_data* buses to 36 bit. The same is true of course also for the delay elements.

Subsequently, resources were saved as it can be seen in column “step 2” of table 2-8.

2.8.3 Step 3

There are two main possibilities to relax the timing in a design: pipelining and multi cycle paths. Pipelining is already done with the delay elements.

A multi cycle path ensures that after a certain time, for example two clock cycles, the signal is valid, but there is no way to ensure that different signals with the same property arrive at the same time. Therefore, if the *read_en* would also be constraint as a multi cycle path it could happen that this signal has a delay of 4 ns and the address of 8 ns. That would mean that at 200 MHz the *read_en* would arrive after one cycle and the address one cycle later. The result would be that the read in the sub-RF is performed on an invalid address.

The multi cycle constraint only enforces the end time of the signal propagation delay.

The solution to this problem is a mixture of pipelining and multi cycle paths. Multi cycle paths are used on the *addr*, *write_data* and *read_data* data paths, but the control signals are pipelined. This is depicted in figure 2-52.

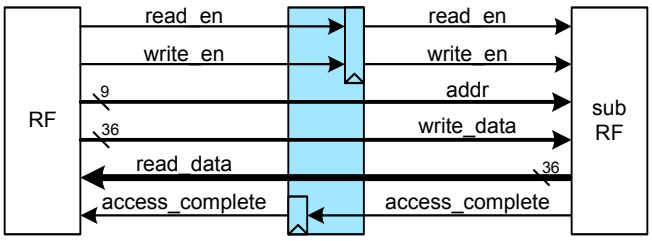


Figure 2-52: Delay Element for Multi Cycle

The promising results can be found in column “step 3” of table 2-8.

2.8.4 Step 4

For timing reasons every sub-RF was connected to own *write_data* and *address* registers, this way it is possible for the placer to place these registers closer to the corresponding sub-RFs. The disadvantage of this approach is obviously the resources usage for these registers, but it is possible to save the resources due to the multi cycle paths that were introduced in step 3. RFS implements the “shared_bus” feature for this reason also for regroots.

2.8.5 Results

Considerable amounts of resources can be saved with the right optimizations; especially the usage of multi cycles and the shared_bus feature contributes to the savings.

The results of the different steps are shown in table 2-8. It is important to note that these numbers are not only the RF components of the crossbar, but the whole crossbar.

	base-line	**step 1**	**step 2**	**step 3**	**step 4**
FD_LD (flip-flops)	22827	22828	21235	18883	14624
saved in comparison to last step	-	0%	7.0%	11.1%	22.6%
LUT	25789	25880	25496	25496	22975
saved in comparison to last step	-	-0.4%	1.5%	0%	9.9%
MUXFX	82	76	75	75	76
CARRY	1544	1544	1544	1544	880
BMEM	128	128	128	128	128
MULT	32	32	32	32	32
DMEM	496	496	496	496	496
OTHERS	456	456	456	456	456

Table 2-8: Crossbar Resource Usage

System Notification Queue

Chapter 3

3.1 Introduction

In complex hardware designs, error conditions and other events can occur that require attention by the controlling system software. Not only the information that something interesting has happened for the software, but also what has happened, while keeping a close time spatial correlation to the event.

A concrete example is an error on a network link. The software quickly has to make a decision if it has to reroute data connections over other links before the link failure has an effect on the whole network due to full buffers, or if it is possible to fix the link. Fixing the link could be done by resetting the link, reducing the number of lanes in use, or by reducing the data rate, because this reduces the error probability. For this decision it needs all available information about the link as quickly as possible.

In the classic scheme all status information is kept in registers and the hardware will issue an interrupt when certain registers change. Thus, the system software will get active and start reading out multiple registers to find out which issue or event caused the interrupt. The approach has two main problems:

Resources: Reading out multiple registers uses bandwidth and CPU time, because reads are blocking.

Latency: The time it takes from when the hardware issues the interrupt until the system will start reading out data is long and therefore it is possible that the status registers will by then have another value. Furthermore, there is also a time gap between the reads of the different status registers.

Measurements have shown a read latency of 256 ns when reading the RF from an HTX Ventoux board, which makes use of the HyperTransport interface.

In the following chapter a solution is proposed that aims to solve these problems in an efficient manner in conjunction with RFS. The system consists of hardware and software components. The hardware part is called System Notification Queue (SNQ). The configuration of this system is in parts automatically generated based on the XML input for the RF.

The general principle is depicted in figure 3-1. Logic causes a change of a hardware register (hwreg) in the RF and as a result the information about the change will be written by the hardware to the main memory of the host system, so that the software can access this information.

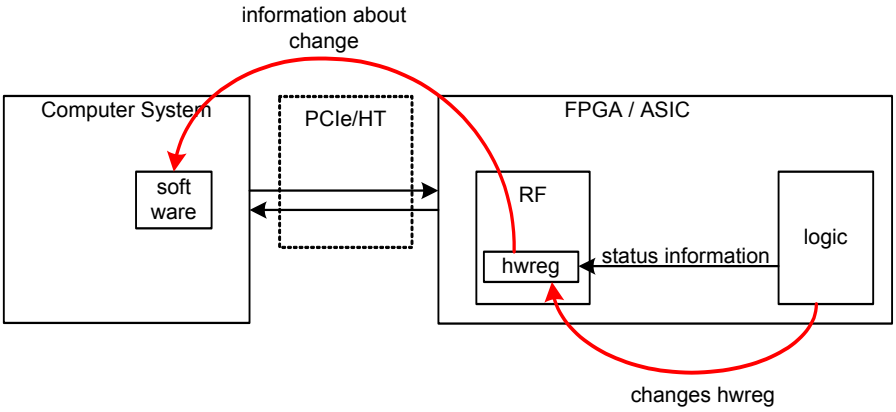


Figure 3-1: Overview

The first design that will make use of the SNQ is EXTOLL, therefore it is used as platform to explain the design decisions.

3.2 Motivation

The research into methods to report hardware states efficiently to the software was motivated by the development of the EXTOLL network, because it can take advantage of such functionality.

EXTOLL is a complex design with many components that can experience failure or require attention by the operating system (OS) of the host system. There are a few typical events that require attention:

- cable is detected or unplugged
- data transfer on link has errors
- packet arrival
- single and double bit errors in SRAMs
- unexpected error conditions

It is a typical event for a NIC that a cable is plugged in or unplugged, and in this case the software has to be informed. Every time this happens the management software of EXTOLL may have to change entries in routing tables.

VELO and RMA packets are usually not causing interrupts on arrival, but optionally it is possible. In that case the OS has to either handle the packet or wake up the process waiting for it. The problem is that the kernel needs to know which one of the up to 256 processes should be woken up. Possibly, multiple processes have to be woken up. Therefore, an efficient way is required to get this information from the hardware to the OS kernel.

Classical interrupt schemes are based on the method that after an interrupt, registers are read from the RF of the device. The amount of data that can be transported by a single processor read is only 64 bit, but VELO and RMA have 256 receive queues. Therefore, it would be either necessary to read four times for each unit or a more elaborate scheme to encode the IDs of the relevant queues in registers would have to be developed.

In a design, which is as complex as EXTOLL, hardware bugs can occur, especially in the phase of the FPGA bring up. Then unexpected error conditions have to be observed and reported with as much information as possible, so that the problem can be analyzed.

3.2.1 Performance Counters

Complex hardware designs often contain multiple performance counters that can be used to analyze certain characteristics of the design. These counters can be used to analyze performance or to debug problems. For example, there may be two counters; one counts the number of packets on a link and the other one the amount of bytes. With these counters the average size per packet can be calculated. To get a precise number it is beneficial to read out with minimal time between the readout of the different registers, and that is exactly what the SNQ is good for.

The EXTOLL design has hundreds of counters as it can be seen in table 2-7 on page 87.

3.3 Register File

For all of these points it is important that the overall system, consisting of the hardware and a host system, has the following properties:

- the information related to the issue has to be stored as quickly as possible
- the system has to be informed quickly about the issue, for this reason an interrupt has to be issued

All of the events that could lead to the necessity to inform the system are attended by the change of a register in the register file, because for example there is a register that has the information if a link is up or down. Therefore, the register file is the optimal place in the design to implement the required infrastructure to sense when a register has changed. This part of the overall architecture was already covered in chapter 2.4.9.

There are different possibilities in the design space as depicted in figure 3-2 to capture the data from the registers for the software.

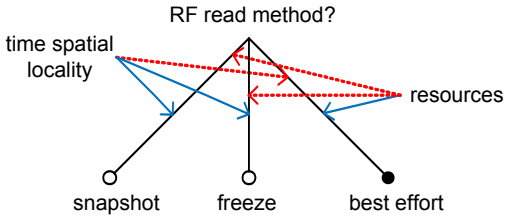


Figure 3-2: Design Space: RF Read Method

The first possibility is the use of **snapshot registers.** Extra registers could be added to the RF automatically by RFS for each register that is interesting. When the trigger event (TE) occurs then a copy of the register is made to the snapshot register. The SNQ would read out the snapshot register at some point.

Freezing of the software interface is a variation of the snapshot register. The software view of registers could be decoupled from the hardware view in the case of a TE. This would save resources in comparison, because no extra resources would be required to be able to read out the snapshot registers from the software view.

However, both of these solutions have the problem that they require extra resources and development effort for the RF, but they still can not assure that the snapshot or freeze takes place at exactly the right moment, because the snapshot would happen at least one cycle after the event that causes the snapshot. The reason for this is that the code that generates the event should have a registered output for timing reasons. Furthermore, the time difference could also be much bigger because of the RF hierarchy.

Therefore, in the end “best effort” was chosen. After a TE the SNQ will try to read out the information as quickly as possible. If this is not good enough, then it is up to the developer of the respective hardware module to write custom Verilog HDL, which provides a reliable mechanism that allows extracting state information that is exactly from the point in time when the event happens.

3.3.1 Trigger Dump Groups

It has been established in chapter 2.4.9 that the RF is the source of events, which can trigger further actions by the SNQ. These TEs are defined in the XML specification of the RF and will be listed in a snq_te.txt file that is generated by RFS.

Consequently, the question is which registers will be stored to the main memory when a specific event is triggered. It makes sense to define this also in the XML definition of the RF.

Information that is sent to the main memory is organized in trigger dump group (TDG). In contrast, to the TEs the TDGs are referring to reg64s instead of hwregs, because reg64s can be addressed from the software view.

To add a reg64 to a TDG the tdg attribute has to be added.

```
<reg64 name="exa1" tdg="examplegroup">
    <hwreg ..>
</reg64>
<reg64 name="exa2" tdg="examplegroup+opt1, othergroup">
    <hwreg ..>
</reg64>
```

Based on the tdg attributes RFS will generate a snq_tdg.txt file that contains all TDGs and the corresponding addresses. In this example the register **exa1** is at the address 0x100 and **exa2** at 0x108. The snq_tdg.txt would have the following content:

```
examplegroup 0x100 0x108+opt1
othergroup 0x108
```

One reg64 can also be in multiple TDGs by listing them in the tdg attribute separated by commas, as shown in the example of **exa2**. Moreover, it is possible to provide options to specific reg64 and TDG combinations with "+", which will be transferred to the snq_tdg.txt. An application for such an option will be shown in the next section.

Both, TEs and TDGs have names, with the help of these names they are matched to each other. There has to be a TE for each TDG, but a TE does not require the definition of a TDG.

TDGs are supporting the special **%n** and **%i** syntax exactly like the TEs as defines in chapter 2.4.9.

The SNQ compiler (SNQC), which will be detailed in a later section, will use the snq_te.txt and snq_tdg.txt to generate the microcode for the SNQ.

3.4 Functionality

The main responsibility of the SNQ is to react on changes in the RF, read out one or more registers (reg64) and write them to the main memory of the host system as outlined in the motivation section. However, it became clear quickly that more has to be done for an efficient solution and of course also more interesting features can be implemented to improve the overall architecture. These features are:

- interrupts
- reset of counters
- write back support for error bits
- flexibility
- possibility to trigger from software
- remote functionality

It is not enough that the SNQ can write data to the main memory, because otherwise the software would have to poll the memory to get aware of messages from the SNQ, it also has to send **interrupts.**

Big designs like EXTOLL have many counters and when they are read out it can make sense to **reset** them directly to their initial value, and not to wait for the software running on the host system to do that. How this has to be done depends on the configuration of the RF. It is either possible to directly write zero to the counter or the special rreinit (chapter 2.4.2) feature can be used to reset multiple counters at once.

Therefore, the SNQ has to be able to write to the RF in two ways, either by writing zero to a counter directly after reading it or by writing to the address of the rreinit element at a later time. The first case is supported with a TDG option. Attaching of "**+W0**" to the TDG will overwrite the reg64 with zero after reading it out. The SNQ also does provide functionality to write rreinit registers.

In the ASICs version of EXTOLL there are many SRAMs, which all feature outputs to indicate SECs (single error corrected) and DEDs (double error detected). Furthermore, there are of course also other error conditions. All these error bits are available in the RF. Therefore, the SNQ can be used to react and forward the error bits to the host system. However, the error bits also have to be set back so that they can detect the condition again. Especially in the case of SECs this is interesting, because this error has, thanks to the correction, on one side no direct negative impact, but on the other side when it occurs often then it is a clear indication that the SRAM is defective.

As outlined in chapter 2.4.2 it makes sense to define the registers for error bits or assertions in the following way.

```
<reg64 name="ecc_errors" tdg="ecc+WB">
    <hwreg name="sec" width="16" sw_write_xor="1" sticky="1" te="ecc"/>
    <hwreg name="dec" width="16" sw_write_xor="1" sticky="1" te="ecc"/>
</reg64>
```

Thus when the "ecc_errors" register, which contains the error bits for 16 SRAMs, was read the corresponding bits can be set back by **writing back** exactly the same value due to the sw_write_xor as introduced in chapter 2.4.2. This is achieved with the help of the "**+WB**" options, the SNQ will write back the register as soon at it did read it.

It has also been established that the SNQ will perform certain actions based on events that are triggered by changes in the RF, but especially in an ASIC it can be interesting to be able to change what actions are performed. Therefore, the SNQ has to be programmable to provide the necessary **flexibility**.

Using the Ventoux board, it takes around 256 ns for the software to read a single value from the RF. However, due to jitter on the host system this value can increase significantly. When values like multiple counters have a correlation then these big gaps are negative. The SNQ can be used to keep the time between the single reads lower and reduce the jitter in the read-out process. Therefore a way to **activate** the **operation of the SNQ from software** is of advantage.

EXTOLL network devices can also be used without a host node, then it would be good if the SNQ could be used. In the end it is even more important to have a way to push status information instead of reading it register by register, because the management software will run on a **remote** host and will therefore have a much higher access latency compared to locally running management software.

3.5 Architecture

The first question that has to be answered for the SNQ is the design of the architecture. A method to integrate it with the rest of the EXTOLL design has to be found. Accordingly, the interfaces have to be defined and a decision regarding the place inside the EXTOLL hierarchy has to be made. The place in the hierarchy also influences the interfacing to a certain degree.

The functionality of the SNQ is from a high level view: A register in the RF changes and the SNQ finds out about this change. As a result the OS of the host system is informed not only about the fact that a register has changed, but also about its content.

The method how the SNQ finds out about changed registers is already defined. In short, a bus is provided by the RF with one signal for each possible event, the signals are pulsed for a single cycle in case of such a change. These signals do not necessary refer to the change of a specific register, they inform about the change of any register that is part of the corresponding TE.

The question is how the SNQ is going to acquire the actual content of registers in the RF and to inform the software. This will bring up the question how the process is controlled.

First it has to be explained what types of data have to be transferred. In order to operate, all sources and destinations have to be known.

This general flow consists of five separate steps, and these are also labeled in figure 3-3.

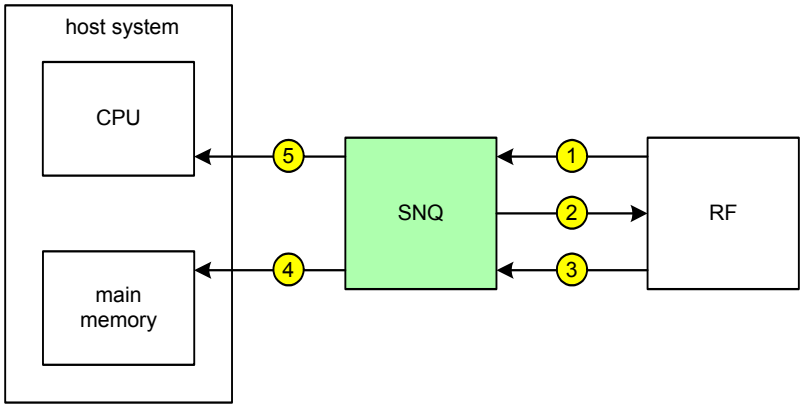


Figure 3-3: Schematic Overview

- **(1)** trigger signal
- **(2)** RF read request
- **(3)** RF read response
- **(4)** write to main memory
- **(5)** trigger interrupt in the host system

A change in a register inside the RF causes a TE, this information is transported (1) over a data path to the SNQ. Based on the event a read to the RF is issued (2) and the RF will answer this read request (3). Of course, also multiple reads can happen based on one TE. The SNQ will write the information from the RF to a ring buffer in the main memory (4) of the host system, which will be detailed in chapter 3.6.5. Consequently, also an interrupt to the host system has to be issued (5) so that the OS kernel is able to take actions based on the information that it received from the SNQ.

Hence, the SNQ has to be connected to the RF and the HTAX crossbar [38] so that it can not only read from the RF, but is also able to write to the main memory. The overall design of EXTOLL should be kept in mind, it is centered on the central HTAX crossbar that makes use of the HToC protocol. Furthermore, also the possibility to trigger interrupts in the host system is required. One consideration is also the latency of RF reads, done by the SNQ. Because these reads are triggered by a condition change it would be interesting to be able to read the value before it changes again. In Figure 3-4 three possible places for the SNQ are presented.

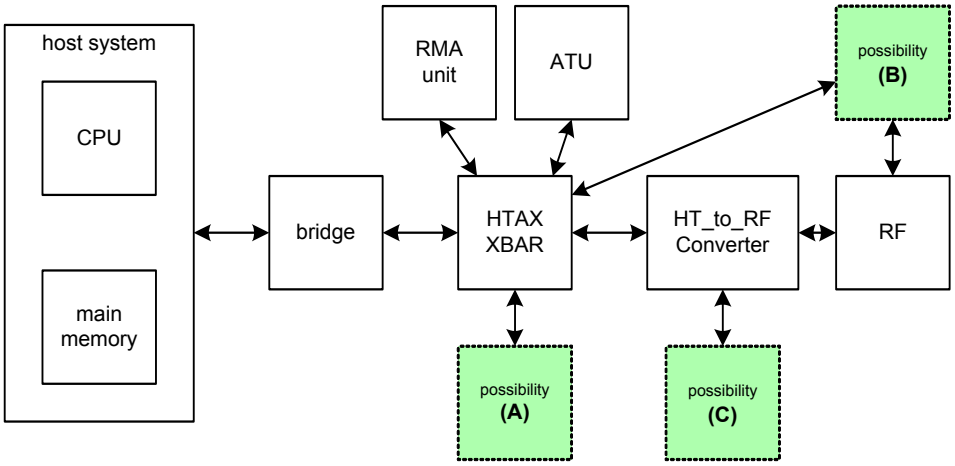


Figure 3-4: Possible Locations for the SNQ

One possibility **(A)** would be to connect the SNQ with an own HTAX port directly to the HTAX crossbar. The clear advantage is that this would be a clean design with clean interfaces, because also other modules are using an own HTAX interface. This would provide the SNQ not only with read and write access to the RF, but also to the main memory. Consequently, this would have the advantage that no changes are required to any other unit, the HTAX crossbar is generated by a script, and changing the number or ports is virtually no effort. However, the biggest problem with this approach is that the resource requirements of the HTAX crossbar are growing according to [48] quadratic with the number of ports.

Possibility **(B)** would involve an extra read path in the RF. This would have the big advantage that the interface does not have to be shared, and the distance in time between the change of the register and the time until it is read out could be smaller compared to (A). Unfortunately, this is not an option because it would require a lot of resources to implement a complete second read path within the RF, or, if it would be a limited read path, which can only access a subset of the registers, then this would limit the flexibility.

Option **(C)** would make use of the HT to RF converter to interface the SNQ with the RF and the HTAX. The converter features already an interface for the debug port, and this interface can as well be used for the SNQ. Furthermore, it has already access to the HTAX. Therefore, it can be enhanced to implement a feature that allows the SNQ to access the HTAX crossbar.

These three possibilities are summarized in figure 3-5.

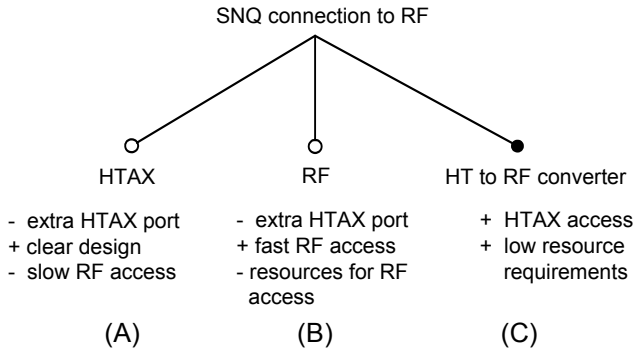


Figure 3-5: Design Space: SNQ Connection Possibilities

Subsequently, **(C)** was chosen, because it provides the best balance between RF access, HTAX access and resource requirements.

3.5.1 Control Logic

There are two main alternatives for the implementation of a control unit, which is able to fulfill the previously outlined tasks. The design space is shown in figure 3-6.

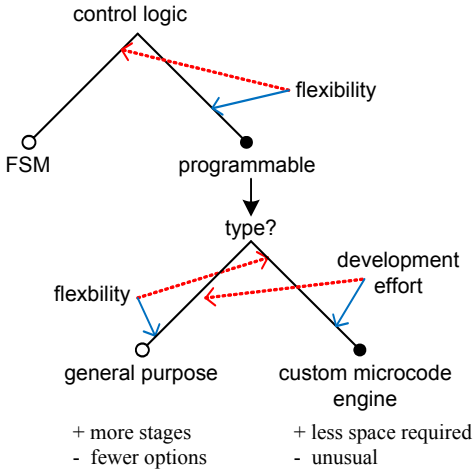


Figure 3-6: Design Space: Type of SNQ Control

A FSM, which can be designed and generated with the FSMdesigner [142], is not an option, because there is no way to encode the necessary RF addresses into the FSM. Furthermore, a FSM that is implemented in Verilog HDL would not allow changing the functionality of the SNQ at runtime and would thereby provide no flexibility. Thus, either an already existing general purpose CPU can be used or a special tailored microcode engine can be implemented.

The following features are desirable in both cases:

- technology agnostic
- high frequency

The implementation has to be **technology agnostic**, because it has to work on different FPGAs and on an ASIC.

Furthermore, it is important that the control logic can run with the same relatively **high frequency** like the rest of the EXTOLL design. If this was not the case, then it would be necessary to use an extra clock domain and this has several disadvantages:

- Slow: A safe way to cross a clock domain is to use an asynchronous FIFO, but this approach has a latency of six cycles.
- Resources: Extra resources are required, because asynchronous FIFOs need a certain amount of registers and logic.
- Complexity: The added complexity due to the crossing of clock domains increases the probability of implementation errors.
- Extra clock domain: Clock domains are a limited resource in FPGAs and complex to implement in ASICs.

A lower frequency for the control logic would have the advantage of a lower power usage. However, the control logic will be inactive most of the time. Hence, clock-gating [155] should be able to reduce the power usage significantly in the ASIC implementation. Clock gating deactivates the clock for idle parts of a design to save power.

In short, the control logic should be able to run at the same clock frequency like the rest of the design.

General Purpose Processors

General purpose processors are in the case of FPGAs often called soft cores and multiple different ones exist, as it can be seen in an overview like presented in [8]. The following architectures were reviewed:

- Xilinx MicroBlaze
- LatticeMico32

- Hephaestus

The MicroBlaze [16] [24] is a 32 bit soft core, which is a fully functional processor that even can be used to run Linux. This soft core is also widely used in academic research, for example in [18] it was used to conduct research that combines general purpose soft cores with specific logic to improve performance and power efficiency. It is easy to obtain information and examples to aid development, because MicroBlaze is used in a lot of places. Furthermore, there is a wide software support. MicroBlaze is for example an officially supported GNU Compiler Collection (GCC) target architecture [20].

LatticeMico32 [118] is also a CPU with a 32 bit data and instruction pipeline. According to [143] it is used in different system on a chip (SOC) projects. It is also supported by GCC.

In [74] the **Hephaestus** architecture is described, an architecture that makes it possible to configure custom application specific processors that can use multiple execution units. Such a complex architecture goes beyond the requirements of the SNQ.

The biggest problem with MicroBlaze is that its license only allows the usage on Xilinx FPGAs. On this platform there are no license costs [17]. However, despite Xilinx is the first target platform of EXTOLL, the design is designated to be implemented on other FPGAs and especially on an ASIC. Being able to use MicroBlaze on these target technologies would require that Xilinx provides a license. It was not tried to obtain such a license, because of the unlikeliness of success. The Nios II [144] from Altera shares the same problem.

The LatticeMico32 does not share the licensing problem and could be used.

However, in the end the decision was to not use a general purpose processor, because the rest of the RF is 64 bit wide, and this width of data has to be transported from the RF to the main memory. This does not prevent the use of the LatticeMico32, but to be able to handle the task efficient either the core would have to be modified or an extra unit that handles the actual data would have to be connected to the core with the slow wishbone interface.

Modifying the soft core could prevent binary compatibility with the standard version of the core and the advantage of having a C compiler would be lost. Attaching an extra unit for the data handling would introduce extra latency and complexity.

Microcode Engine

A microcode engine is a hardware unit that allows implementing programmable control logic.

Figure 3-7 shows a very simple variant of such an engine. It reads instructions from a program memory (or store) and a part of the instruction word is directly used to control logic. Another part of the instruction word is used in the execute logic to control the further flow.

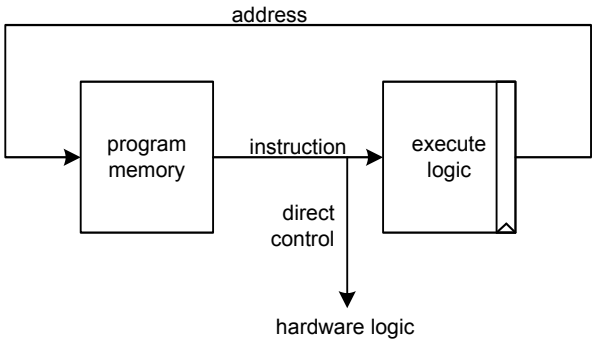


Figure 3-7: Simple Microcode Engine

One main variation is if a microcode engine uses vertical or horizontal encoding. Both variants are depicted in figure 3-8. The vertical variant can require multiple microcode words to produce one complete control vector for the hardware. Depending on the decoder output the microcode is used for different parts of the hardware. A horizontal version on the other side has the complete control vector in a single microcode word. Extensive details about microprogramming can be found in [156].

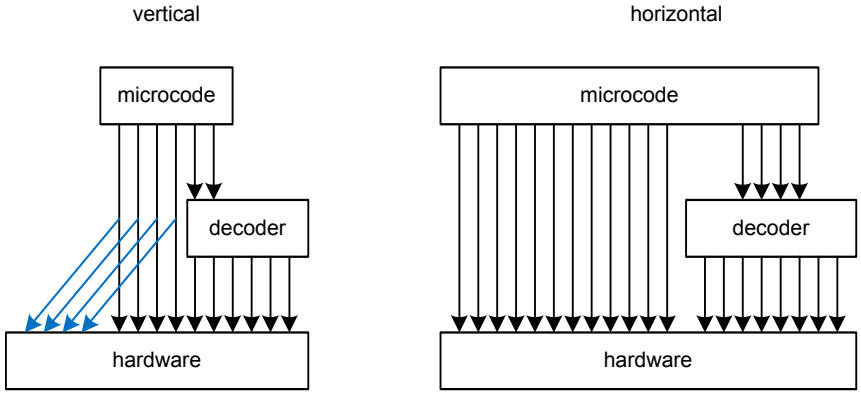


Figure 3-8: Vertical and Horizontal Instruction Format

For the microcode engine of the SNQ a horizontal approach has been chosen, because this has the advantage that a 64 bit wide program memory is big enough to contain one instruction and all necessary operands in one entry. Thus, the implementation uses a fixed size instruction format. Therefore, no further memory reads are required to lookup addresses or other information. This allows fast execution, which was one of the reasons to design the SNQ. Moreover, it is also reducing complexity.

The program memory is 64 bit wide and will be implemented as SRAM. In Xilinx Virtex6 FPGAs there are Block RAMs available [9] that are 72 bit wide and have 512 entries, this size is sufficient for most SNQ programs. Other high-performance FPGAs provide similar hardware components. In the case of an ASIC there are of course also 64 bit wide SRAMs available. Therefore the decision was to use a 64 bit wide instruction format.

The disadvantage of the horizontal instruction format is of course that a part of the program memory is wasted, because not all instructions need all operands.

More details about the microcode engine of the SNQ will be given in chapter 3.6.3.

3.6 Hardware Environment and Components

As a result of the discussion in the last section, the SNQ finds its place in the overall architecture as shown in figure 3-9.

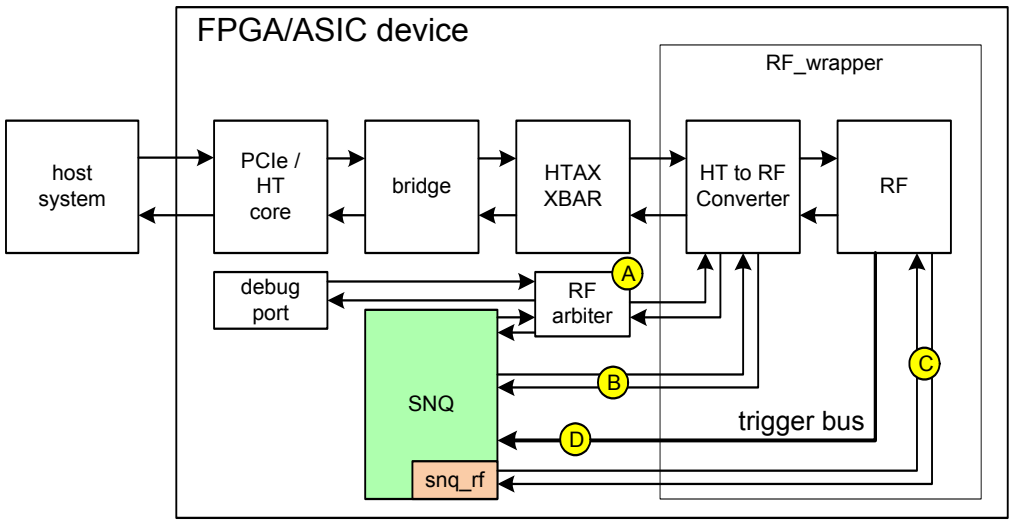


Figure 3-9: SNQ Environment Overview

The SNQ will perform reads and writes on the RF through the RF arbiter **(A)** that allows the SNQ and the debug port to access the RF. On the other side the sending of HToC packets to the HTAX crossbar will be done over an interface **(B)** to the HT to RF Converter. The snq_rf within the SNQ is used to control the SNQ and read out status information, this RF is connected **(C)** to the main RF.

Reacting on TEs is the main task of the SNQ, and the ***trigger*** bus **(D)** transports this information from the main RF to the SNQ.

The architecture of the SNQ has been chosen to be as simple as possible. This reduces the resource requirements and the probability of bugs during the development. Resource requirements had to be kept low not only in terms of hardware requirements, but also in terms of the development time.

In figure 3-10 the most important components of the SNQ are shown.

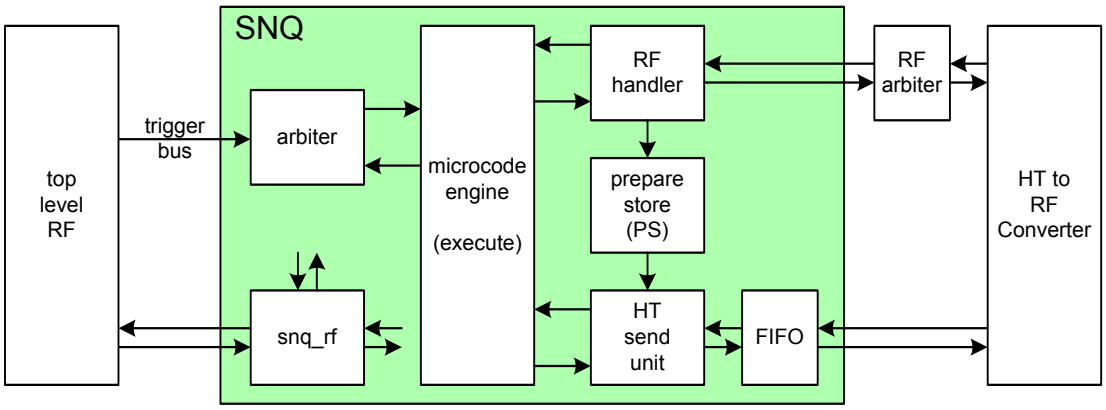


Figure 3-10: SNQ Control Flow

The arbiter decides which event should be processed next. Based on this decision the ME will start executing the corresponding program. The program is controlling the RF handler and the HT send unit.

In a typical operation the RF handler will read registers from the RF and write the data into the prepare store (PS). The PS consists of seven 64 bit registers PS0 to PS6. As soon as the RF handler is done, the microcode engine will instruct the HT send unit to prepare an HToC packet and forward it to the HT to RF converter.

3.6.1 HyperTransport on Chip to RF Converter

The converter unit was already introduced in chapter 2.6.1, but it has been extended for the SNQ. The extensions are shown in figure 3-11. There is an RF arbiter that is used to allow not only the debug port, but also the SNQ to access the RF interface of the HT to RF converter.

Furthermore, an extra interface was added to the HT to RF converter that allows sending packets to the HTAX crossbar. The interface is a FIFO interface. From the SNQ to the converter there is a data path that is 64 or 128 bits wide, depending on the width of the HTAX crossbar and an empty signal. The SNQ is storing complete HToC packets consisting of header and payload into the FIFO.

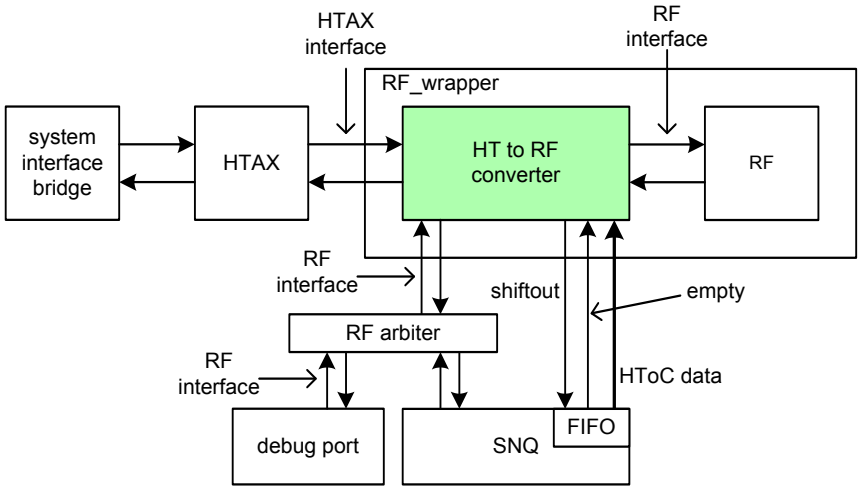


Figure 3-11: Extended HT to RF Converter

As soon as the empty signal indicates that the FIFO is not empty anymore the converter will start reading the data. The converter will interpret the header of the packet to send it to the correct destination port and VC on the HTAX crossbar. Moreover, it will also read the packet size from the header and will shift out the corresponding amount of data from the FIFO in the SNQ.

3.6.2 Trigger Arbiter and Control

As explained before, the *trigger* bus, which carries the information about the TEs, is provided by the RF to the SNQ. The bits of this bus are pulsed for a single cycle, and an arbitrary amount of bits in the *trigger* bus can be set at the same time.

Therefore, the arbiter unit has to arbitrate between the different possible TEs and store the remaining events. Moreover, the arbiter also has to provide methods to control which TEs are allowed to cause the execution of microcode in the SNQ and a way to trigger the execution of microcode from the host system.

Each bit on the *trigger* bus is accompanied by control registers as depicted in figure 3-12. These registers can be used to control the behavior of each trigger. There has to be a trigger control (TC) in the snq_rf for each event, which is provided by the RF, but there can be more TCs than bits on the *trigger* bus. The right side of figure 3-12 shows a small example.

There is a small naming problem, because the events that are emitted from the RF on the *trigger* bus are called TEs in the XML of the RF, but also the events within the SNQ are called TEs. The only difference is that the TEs within the SNQ are filtered by the trigger control logic and extra TEs, which can only be executed by software, can also exist within the SNQ.

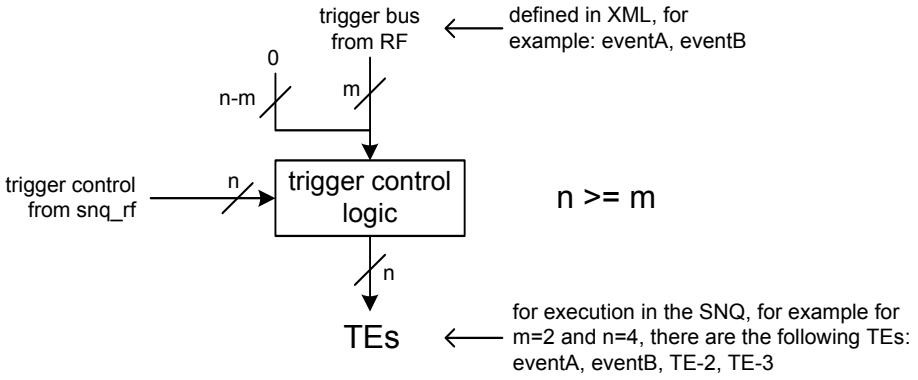


Figure 3-12: Trigger Control Logic

TEs have names that are defined in the XML of the RF as introduced in chapter 2.4.9. When there are more TCs than events specified in the XML, then these extra TEs are named TE-$number. The extra TEs can be used to start other SNQ programs from the host system with the trigger_now functionality, which will be introduced in this section. The other TC control bits do not matter for these extra TEs, because there is no *trigger* bus input bit for these and it is therefor assumed to be always 1'b0. Each TC consists of multiple control bits:

- trigger_enable
- trigger_now
- trigger_once_enable
- trigger_once

Figure 3-13 shows the control logic for a single TE. By default there are 32 and the BTS arbiter in the figure is thus performing arbitration between the outputs **(E)**.

Trigger_enable is generally enabling the TE, each time there is an event from the RF it will be used. When the *trigger_enable*, which is shown at **(A)** in figure 3-13, is not set, then there will be no event at all for the TE.

It is not always the right behavior to have a TE each time there is an event from the RF, because the result could be a tremendous amount of writes to the main memory. Some TEs should only happen once and have to be activated again by software. For this reason there is the ***trigger_once_enable***, as soon as it is set to 1'b1 the events from the RF do not trigger directly anymore due to the "**and**" gates at **(B)** in figure 3-13.

The ***trigger_once*** of that particular TE has to be written. It will allow exactly one TE and then it has to be activated again. The register *once_outstanding* **(C)** is storing the information that a “once” event is outstanding, it will be set back to 1'b0 if a there was such an event.

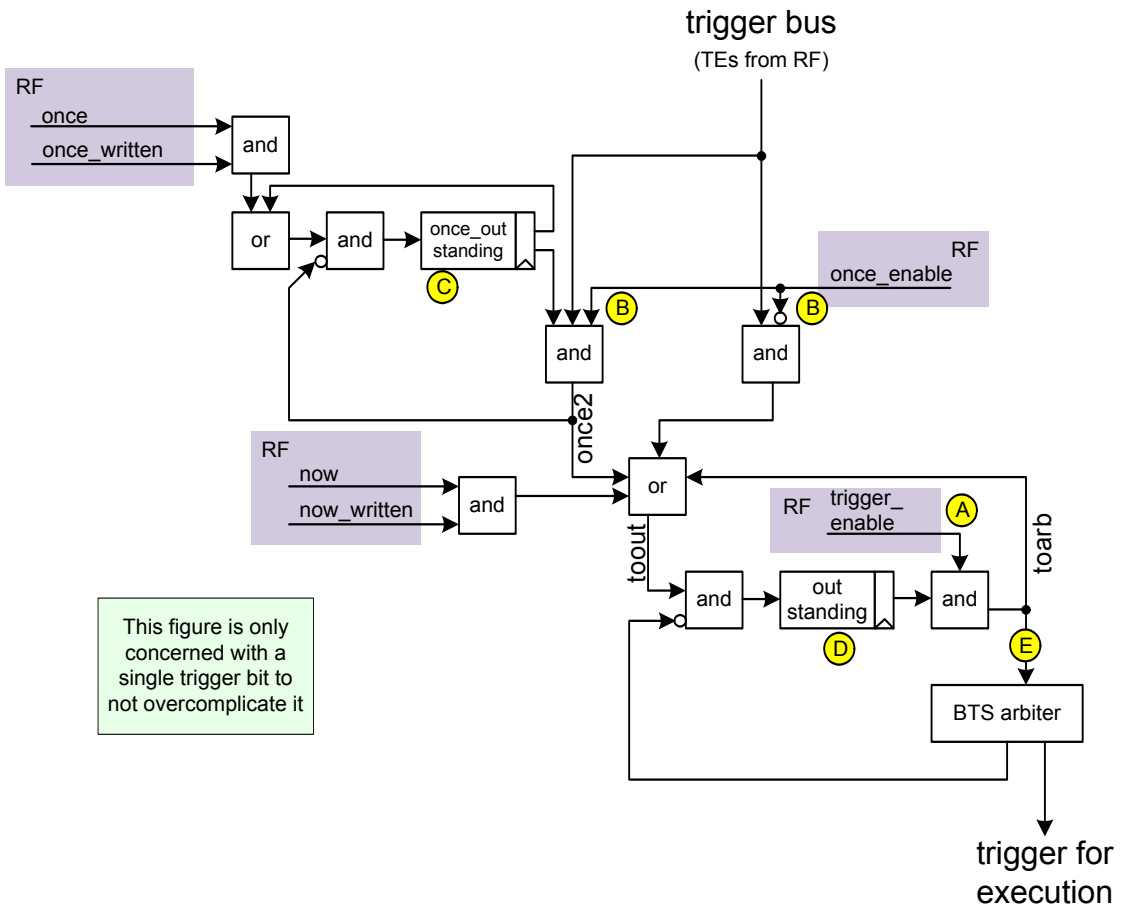


Figure 3-13: Schematic of the Trigger Control Logic

In the case the software intents to directly trigger a TE it can write the corresponding ***trigger_now*** bit. It will trigger each time the bit is written, if the *trigger_enable* is set.

The *outstanding* **(D)** register it storing the information about outstanding triggers that still have to be handled by the SNQ.

The decision which trigger signal should be selected next for execution is made with the help of an efficient binary search tree (BTS) arbiter. This arbiter is created with the help of a generator script [114].

In summary, to a use a TE the corresponding *trigger_enable* bit has to be set. If the TE should happen only one time, then the corresponding *trigger_once_enable* and *trigger_once* bits have to be enabled additionally.

3.6.3 Microcode Engine

The task of the microcode engine (ME) is to control the RF handler and the HT send unit with small programs that are started based on the arbitration decision of the arbiter.

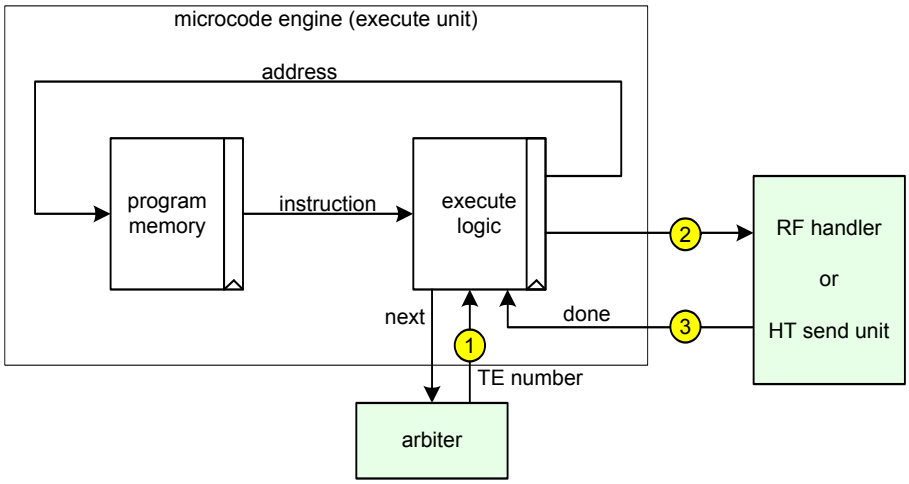


Figure 3-14: Microcode Engine

The simple design was chosen because the number of events is expected to be relatively small. It is using register pipelining to relax timing as it is shown in figure 3-14, but it does not use the pipelining to be able to execute multiple instructions in consecutive cycles. It would not be able to improve the performance significantly, because most commands require one or more reads from the RF and reads are, with at least five cycles, slow. Most reads in the EXTOLL design take nine cycles, due to the RF hierarchy.

In figure 3-14 it can also be seen that the output of the program memory is not directly used to drive the RF handler or HT send unit, because these units are driven by output registers of the execute logic. Therefore, it is possible to disable the output of the program memory SRAM, while the ME waits for the completion from the RF handler or HT send unit, to save power.

After a cold reset the content of the SRAM will be unknown, therefore by default the ME is disabled until the microcode load was performed by writing the program code and setting the enable bit. This is done with the help of the SNQ RF.

As soon as the ME is enabled it will start executing code. It will start by default with the last instruction in the program memory. It would have been logically to start executing at address 0x0, but that is the jump destination for the first TE, therefore the last address was chosen. By using the number of the TE **(1)** directly to jump into the respective code for the event, neither a calculation has to be made nor is an extra jump table necessary.

The instructions, which control the RF handler or the HT send unit, will produce output on the data paths **(2)** to the respective units output. As soon as this units have completed their task they will use the done **(3)** signal to inform the ME. The ME will then load the next instruction from the program memory.

The program flow is described in chapter 3.7.1 on page 123.

At some point it may be necessary to replace the microcode at runtime. This is possible by disabling the engine, replacing the microcode, setting the program counter and enabling the engine again. The program counter can be read with the RF. It is also possible to set it, while the ME is disabled.

The best method is to disable all trigger events and wait until the ME is at the start instruction, and then the code can be replaced without the danger of any hazard.

3.6.4 Register File Handler

The RF handler in figure 3-15 is responsible for reading from the RF and writing to the prepare store (PS), and furthermore it can write to the RF.

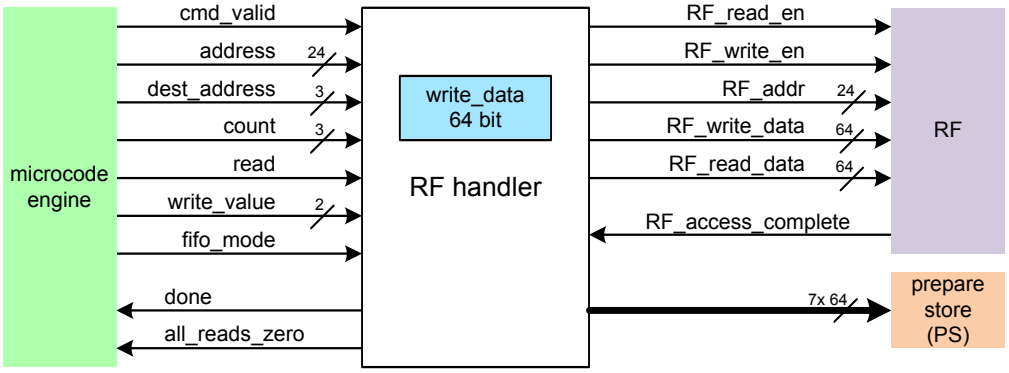


Figure 3-15: Register File Handler

Therefore, it provides an interface for the ME. The interface uses a *cmd_valid* signal to indicate that all signals are valid and the ME will keep them static until the RF handler indicated with the *done* signal that the task has been completed.

The RF handler can be used in a read and write mode. The read mode is activated by setting the read input to 1'b1. In both modes the *address* and the *count* inputs are describing the RF address and the number of additional repetitions that should be performed. Thus, it is possible to repeat a read or write up to eight times.

In the read mode the *dest_address* describes the 64 bit register destination register in the PS for reads. The RF address and the PS destination address will be increased for the additional reads that can be initiated by setting the count input to a value unequal to zero. The RF will not be increased when the FIFO mode is active.

The 64 bit *write_data* register contains the value of the last read. This register can be used either for write-back, or, in another activation of the RF handler in write mode the content of *write_data* can be written to an arbitrary RF address.

For optimization purposes there is the *all_read_zero* signal. It is valid when the done signal is set, and it is 1'b1 when all reads of the last command were zero.

The *write_value* input is used in the read mode for the write-back operation, or, in the write mode it defines which value will be written. The usage of these is detailed in the subsections that describe the LOAD_PS (chapter 3.7.3) and WRITE_RF (chapter 3.7.6) instructions.

3.6.5 HT Send Unit

The HT send unit has to perform writes to the SNQ mailbox (SNQM) and to issue interrupts by sending HToC packets to the HT to RF converter.

Furthermore, the unit in figure 3-16 does also provide a special RAW mode that allows sending HToC packets with completely user-defined content. The feature can be used for example to send data over the EXTOLL network.

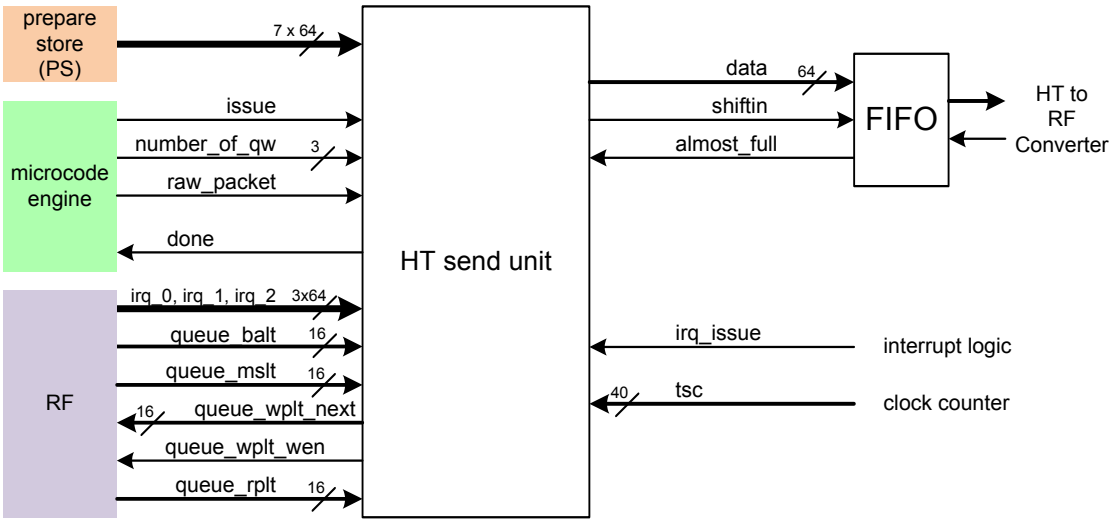


Figure 3-16: HT Send Unit

The size of the packet that should be send has to be provided by the execute unit and is stored in the microcode instructions, because the HT send unit has no information about which parts of the PS are relevant or valid.

System Notification Queue Mailbox

The mailbox is, as shown in figure 3-17, a ring buffer and stored in the main memory of the host system. The writes to the main memory are organized in notifications with 64 bytes each, because this is the cache line size on all current x86_64 CPUs and the maximum payload size of HyperTransport packets. The mailbox can have a size of up to 4 MB.

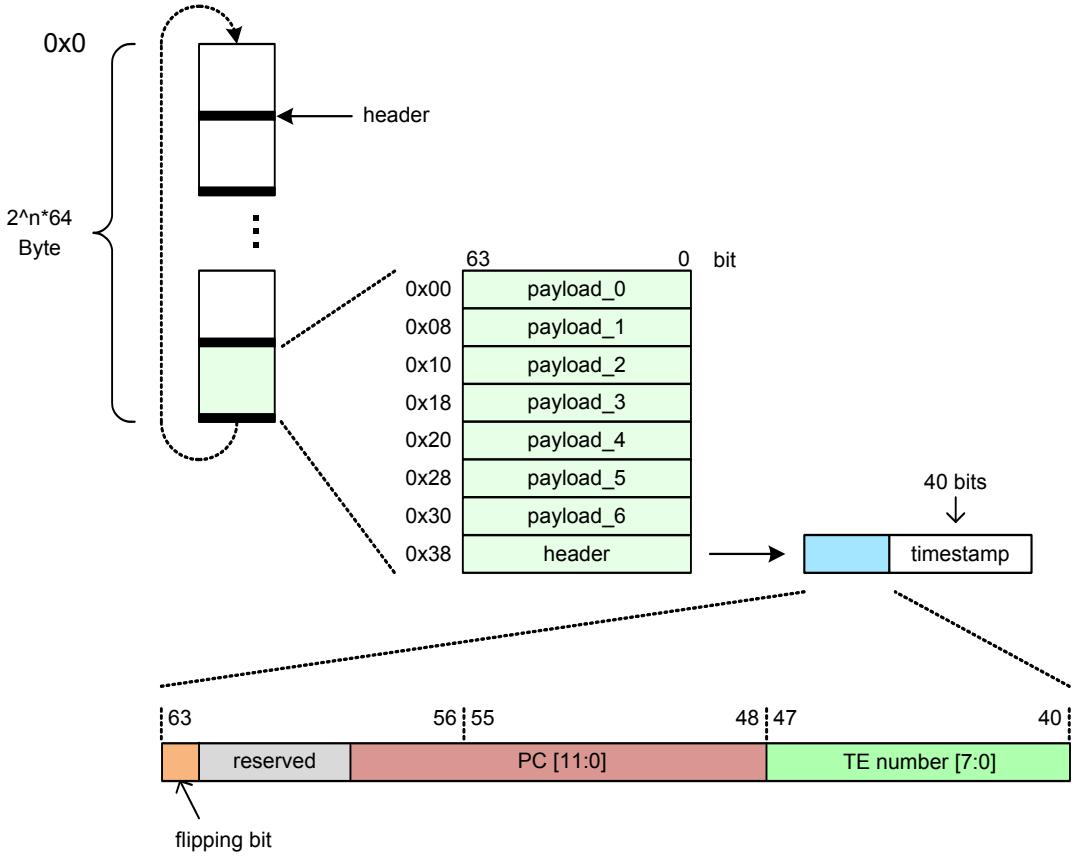


Figure 3-17: SNQ Mailbox Layout

The header is stored in the last quadword to be sure that it is written last. This way it is ensured that the whole packet is in a consistent state after a new SNQ messages was detected due to the header. If the header quadword would be in the beginning, then it could happen that the CPU reads the header, interprets it and reads the second quadword despite it has not yet been written. This could happen in the case the write to the main memory is split up in multiple smaller ones. Despite it is hard to imagine why this should be done by a PCI

express component the PCI express specification allows such a behavior, though the order would be kept. PCI express components are: PCI express cores, switches and root complexes such as host chipsets.

The header contains:

- timestamp: 40 bits
- TE number: 8 bits
- program counter: 12 bits
- flipping bit

The 40 bit timestamp allows the software to get accurate timing regarding the occurrence of an event; the timestamp counter is a global clock. For example, the EXTOLL design has such a clock. Furthermore, it contains the TE number of the program that caused the sending of this notification and additionally the program counter (PC) of the exact instruction that caused it. This allows correlating exactly which data is contained in the notification message, because one TE can cause the sending of multiple notifications. Consequently, the TE number is not really necessary, because the information can be derived from the PC, but in the header there was enough space for it.

There are usually two methods for the software to check for new entries in such a ring buffer. Either the write pointer can be read from the RF of the device or it can check if the header is unequal to zero. The first approach has the disadvantage that it is necessary to read from the RF and this takes hundreds of ns, and the second approach has the disadvantage that the software has to overwrite the header with zero to be able to detect new notification after the wrap-around.

Therefore, the SNQ is implementing the flipping bit mechanism, which was introduced in [99]. The header contains a single bit that is inverted every time the ring buffer wraps around. The software neither has to read the write pointer from the RF nor write to the memory, it just has to know if it has to expect a 0 or a 1 at the position of the flipping bit when it checks for new notifications.

When the software consumes a notification from the SNQM it has to write the read pointer in the RF to inform the hardware about the consumption of the notification, because the hardware is using the read pointer to ensure that it does not overflow the mailbox.

However, not all SNQ notifications require seven quadwords of payload, but the header still has to be in the last quadword, because the software is expecting it there. To save traffic the implementation does in that case not start writing from the first quadword in the 64 byte. Figure 3-18 shows an example for such a notification with only three quadwords of payload. The VELO [114] uses the same approach.

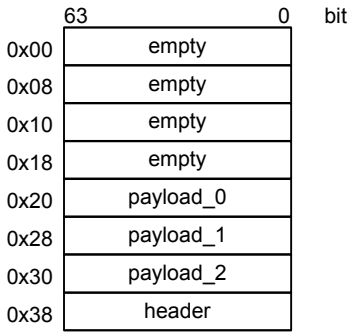


Figure 3-18: Partly Filled Mailbox Entry

Multiple notification queues were not considered, because there was no conclusive reason to pursue such a solution. Such a possibility would require more resources because for each of these queues the base address, read and write pointers would have to be kept. Furthermore, this would bring up the problem how to decide which events have to be written to which notification queue. The distribution of the notifications to different threads can be performed by software on the host system.

Interrupts

The SNQ has to be able to use interrupts to get attention by the software on the host system after writing to the SNQM. Two problems had to be solved:

- sending interrupts in a portable and flexible way
- ordering

Interrupts are, depending on the interface to the host system, triggered in different ways.

In the case of an HT core, as used for the Ventoux board, the interrupt is issued by a posted write. In the case of PCIe cores it seems to be common that a custom message signaled interrupt (MSI) interface is provided. This approach is for example followed by the PCI express cores from Xilinx [19], Altera [145] and Lattice [146].

The SNQ has to be able to send interrupts in all environments. Therefore, an approach is used that is very similar to what is done in the case of the HT core. The RF contains three registers for this purpose: irq_0, irq_1 and irq_2. These registers are set by the system software, and store the HToC header for a posted write in irq_0 and irq_1, and additionally 64 bit of payload in irq_2. The HT send unit assembled these three registers to an HToC packet and forwards it to the FIFO for the HT to RF converter.

Figure 3-19 shows the entities that are involved when the SNQ issues an interrupt. The SNQ generates an interrupt packet **(1)**. When an HT core is used then at **(2)** and **(3)** nothing has to be done, the interrupt is a normal posted write and the interrupt handler **(4)** will handle the interrupt in the end.

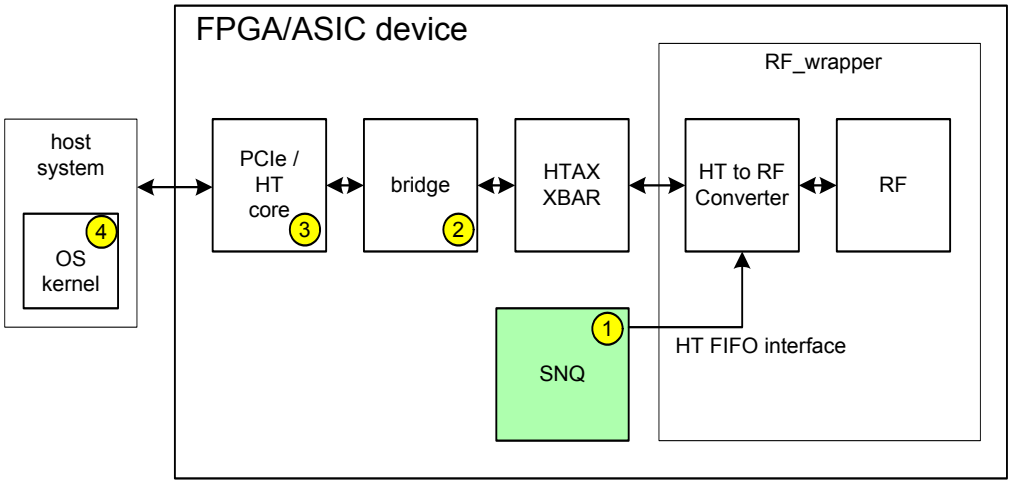


Figure 3-19: Components Involved in an Interrupt

However, in the case of PCIe the bridge **(2)** has to filter out interrupts that are sent by the SNQ and operate the extra interface **(3)** of the respective PCIe core.

Now it is clear how interrupts are issued, but not under which circumstances they are issued. The obvious possibility is that always when a notification is written and the interrupt is enabled, then an interrupt will be issued. When the interrupt was issued it has to be enabled again by the kernel driver.

However, the problem is that interrupts can pass writes on the way to the host system and then the interrupt handler runs and can not find anything in the SNQM.

The reason for this is depicted for the PCIe case in figure 3-20, where from the right to the left a unit sends packets over the HTAX to the bridge in the order: write1, write2 and as last packet it sends an interrupt. However, the problem is that an extra MSI interrupt interface is used and therefore the bridge has to decompose the in-order stream of packets and forward them over two different interfaces to the PCIe core. The PCIe core is suspected to have internally a FIFO for data packets and a packet generator for interrupts, both data streams are merged again. As a result, the order of the writes and the interrupt may change as shown on the left side of figure 3-20.

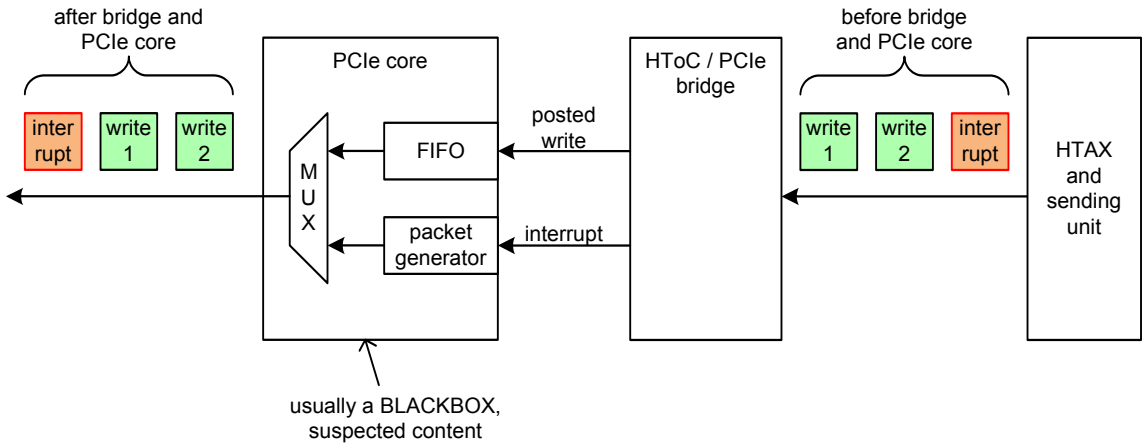


Figure 3-20: PCIe Core: Data and Interrupt Path

The Altera user guide [145] suggests to use the empty flag of a FIFO in the PCIe core or to wait a certain amount of cycles to ensure that the interrupt is sent after a specific packet. Furthermore, this behavior was observed with the Lattice PCIe core during another research project.

Consequently, it can not be assumed that there is ordering for writes and interrupts. Thus, when the interrupt handler assumes that there has to be a notification when there was an interrupt and it waits some time, then the question is how long it should wait. This depends on the size of the FIFOs inside the PCIe core, which might be responsible for the lost ordering and other factors like the speed of the PCIe interface. However considering the example from figure 3-20, this does not really help in the end, because the interrupt handler will not know that it has to wait for write2 too. The issue can be solved by reading the SNQM write pointer from the RF, but a RF read takes hundreds of ns.

However, there is another possible hazard which is depicted in figure 3-21:

1. SNQ writes two notification (write1 and write2) and an interrupt; the notification handler handles the notification and enables the interrupts again with a write to the RF
2. enable write and the write3 cross each other on the way
3. interrupt handler checks the SNQM before returning and can not find a new entry
4. write3 arrives in the main memory and is **not** handled
5. much later the interrupt handler will finally find write3 due to a new interrupt

This problem can also be avoided with reads to the RF.

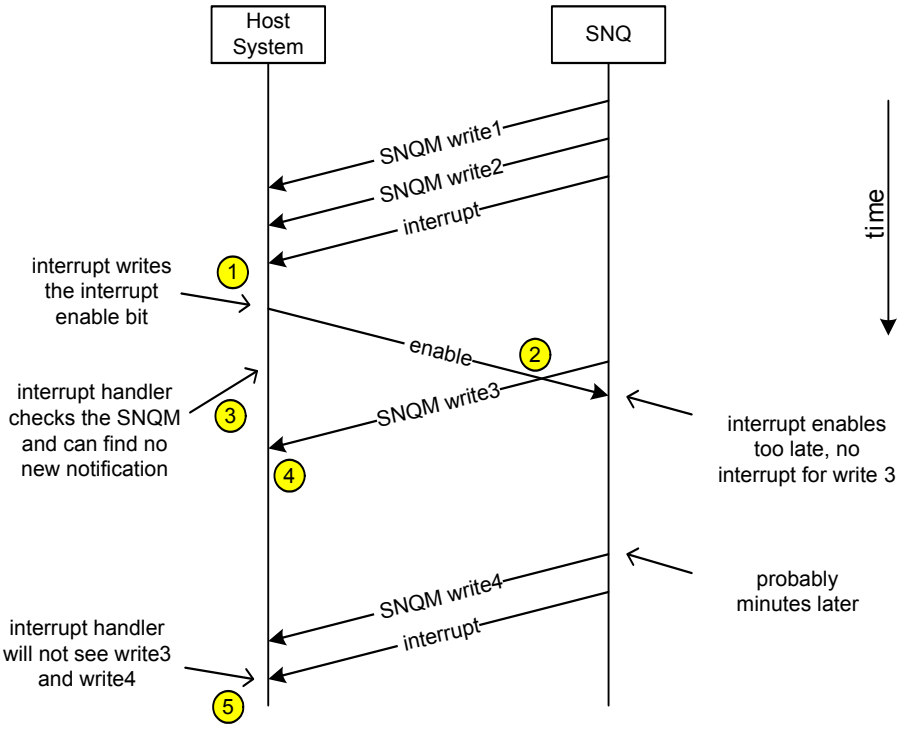


Figure 3-21: Interrupt Hazard

Therefore, a different solution is proposed which informs the hardware for which SNQM entry a new interrupt should be created. With this method only writes to the register file are required, and costly reads can be avoided.

The method is called “watermark” interrupts, because the term “level” is already occupied for interrupts. It was used before for a high speed flash device, which showed the ordering problem with the Lattice PCIe core. This type of interrupt is very simple and follows two rules:

- Watermark interrupts are enabled by a write to the corresponding register.
- Watermark interrupts are **not** edge triggered by the sending of notifications, but only by the current value of the SNQM write pointer.
  An interrupt is issued as soon as the SNQM write pointer is greater or equal than the watermark register.

This mechanism solves both problems that were introduced in figure 3-20 and figure 3-21. In the former case the interrupt handler would, after finding no new notification in the mail-box, activate interrupts again with writing the very same value to the watermark register. For this application the sw_written feature, introduced in chapter 2.4.2, has been added to RFS. Then the hardware would directly issue a new interrupt, because write1 and write2 were already done.

The crossing **(2)** in figure 3-21 is thus also solved, because at **(1)** the interrupt handler would write the "3" for write3 to the watermark register and the SNQ would send the interrupt.

The simplest method is that the software always sets the watermark register to the next expected entry after reading the currently available notification. This is the most useful method for the SNQ. It is also possible to use a bigger offsets, when the software for example expects a surge of notification. By doing so unnecessary interrupts can be avoided.

However, this makes more sense for other types of devices where the software submits lots of commands and waits for lots of notifications that indicate the completion of the commands.

RAW Mode

In the raw mode the HT send unit will forward the content of the PS directly as HToC packet to the HT to RF converter. This mode allows sending arbitrary HToC packets from the SNQ. It can, for example, be used to send packets from the SNQ to the EXTOLL network by using the VELO unit. This will be detailed in chapter 3.9.3.

3.6.6 SNQ Register File

The SNQ itself it also controlled and configured by a RF. It contains the following main functions:

- microcode engine program and enable
- trigger control
- SNQ mailbox addresses
- interrupt control registers

3.7 Microcode Engine Instruction Set

The instructions for the SNQ were designed with simplicity and speed in mind. All instructions are 64 bit wide as shown in figure 3-22, because this width gives enough room for all functionality.

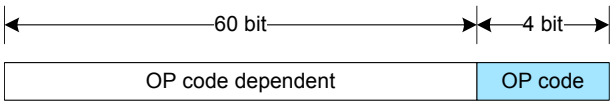


Figure 3-22: General Instruction Format

It is expected that in most cases only a single register has to be read from the RF and written to the SNQM, this functionality can be implemented with a single LOAD_PS (LOAD Prepare Store) instruction. The following instructions are supported by the ME and distinguished by their OP code:

- JUMP_ARBITER
- JUMP
- LOAD_PS
- WRITE_SNQM
- HT_RAW
- WRITE_RF

The instructions will be detailed in the following pages. All instructions despite the JUMP_ARBITER instruction include an explicit jump_destination. The jump_destination is the address of the instruction that should be executed as soon as the current instruction is finished.

Thus, the only instruction that is doing branching is JUMP_ARBITER.

3.7.1 JUMP Arbiter

The JUMP_ARBITER instruction in figure 3-23 waits for a TE from the arbiter and as soon as there is a valid trigger, it will use the trigger number and jump to this address in the microcode. The trigger number is also stored to a register to produce the header of the notification. This is the only instruction that does not have a jump destination in the instruction.

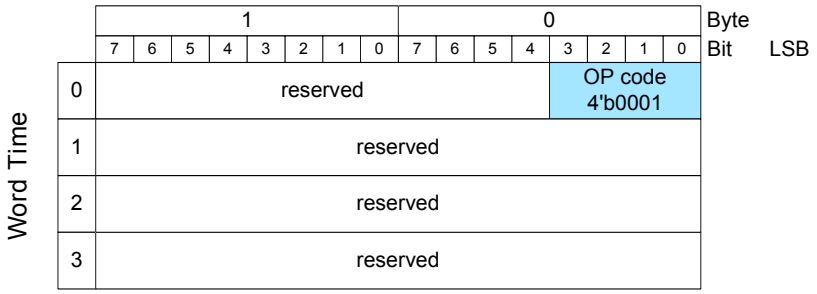


Figure 3-23: SNQ Instruction: JUMP_ARBITER

Thus, the JUMP_ARBITER is the first instruction that should be executed by the microcode engine and the one where it will wait for new events. Table 3-1 shows how the program flow is working. The microcode starts at address 0xFF and waits for a valid TE from the arbiter. The first instruction for each of the 32 TEs in this example is aligned to the addresses from 0x0 to 0x1F.

TE-0 is the simplest example, in that case the JUMP_ARBITER jumps to 0x0 and executes the single instruction, and then the program counter is set back to 0xFF. The case of TE-1 is more complex, because it consists of three instruction words the first one is at 0x1, and the two other ones are placed in this example at 0x20 and 0x21, because these are the first free microcode locations after the code for the 32 TEs. Thus, in the case of TE-1 the ME will execute the code at the microcode locations 0x1, 0x20 and 0x21. Then it will return to 0xFF and wait for further events.

address	description	jump_ destination
0x0	code for TE-0	0xFF
0x1	code for TE-1	0x20
0x2	code for TE-2	0xFF
...	...	...
0x1F	code for TE-31	0xFF
0x20	more code for TE-1	0x21
0x21	more code for TE-1	0xFF
...	...	...
0xFF	JUMP_ARBITER	

Table 3-1: Program Flow

3.7.2 JUMP

The unconditional jump instruction in figure 3-24 exists only for the case that there is a TE, but no code is provided for it, then this instruction can be used to jump back to the JUMP ARBITER instruction. The SNQ could get into an unknown state and stop working, if this program memory was in an unknown state.

The **jump_destination** is the address the microcode engine should jump to after the packet was sent. All following microcode instructions also have this operand.

Figure 3-24: SNQ Instruction: JUMP

3.7.3 LOAD Prepare Store

The main task of this instruction is to read one or more 64 bit values from the RF and write them to the PS. It provides several operands as illustrated in figure 3-25. For the read operation the following operands have to be set:

- **dest_address**: first destination address within the PS
- **count**: number of consecutive reads to be made
- **src_address**: 24 bit address within the RF, this is a quadword address. This means that it is actually a 27 bit address, but the last 3 bits are always zero. Therefore, RFs with a size of up to 128 MB are supported.

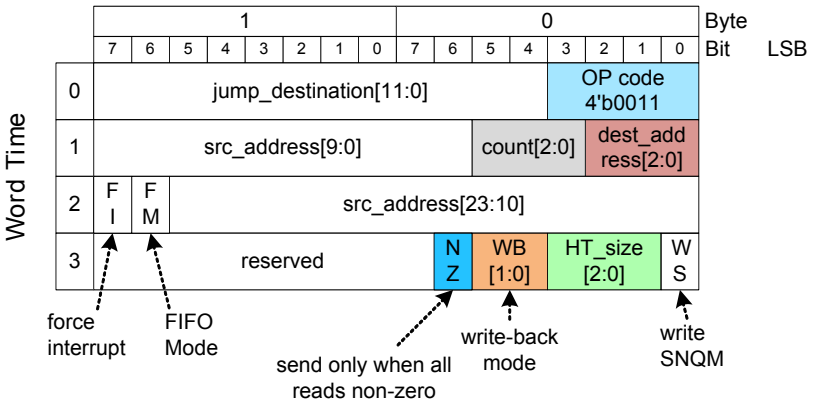


Figure 3-25: SNQ Instruction: LOAD_PS

Additionally, it is possible to make use of the write-back mode, which will directly write to the RF after each read. The value that is written depends on the **WB** operand:

- 2'b00: no write-back
- 2'b01: write 0
- 2'b10: all bits set to 1
- 2'b11: write the same value back if the read is unequal to zero

The write-back modes allow clearing registers in an efficient way as described in chapter 3.4.

It is not possible to combine any write-back mode with the FIFO mode (**FM**), which does allow performing multiple reads while not increasing the src_address. This mode allows reading out error logging FIFOs that are connected to the RF as introduced in chapter 2.4.5.

By setting the write SNQM (**WS**) bit a write to the mailbox will be done after the reads from the RF were completed. The **HT_size** has to be the complete size of the packet including the SNQ header. Counting starts at zero to save coding space. Thus, an HT_size of 3'h7 will cause a notification that includes the header and all seven PS registers. The HT_size can not be derived in all cases from the count, because it is in many cases necessary to perform multiple LOAD_PS instructions to load all necessary values into the PS, because they are not at consecutive addresses. The last of these LOAD_PS instructions can perform the SNQM write.

Setting the force interrupt (**FI**) bit will cause an interrupt independent from the watermark interrupt mechanism.

The **NZ** bit is a special bit that can be used to prevent writes to the SNQM that have no interesting content. It prevents the sending of a notification if all reads from the RF returned zero.

3.7.4 Write to SNQ Mailbox

The instruction in figure 3-26 offers a method to write to the SNQM without reading to the PS first. The **HT_size** is the amount of quadwords the SNQ is sending to the SNQM including the header, but counting starts at zero. Thus an HT_size of 3'd0 will write only the header into the SNQM. This is the main application for this instruction, because the LOAD_PS integrates the WRITE_SNQM functionality to save instruction words.

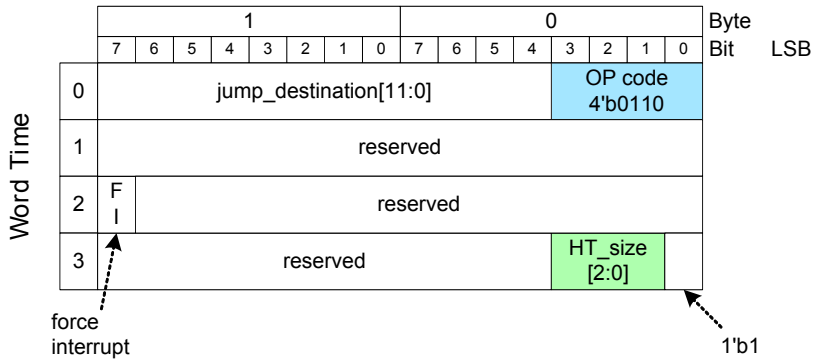


Figure 3-26: SNQ Instruction: WRITE_SNQM

3.7.5 Send RAW HToC Packet

This instruction, as depicted in figure 3-27, sends arbitrary HToC packets. The packet has to be prepared in the PS with the LOAD_PS instruction, and the HT_RAW instruction can be used to forward it over the HT to RF converter to the HTAX crossbar. The converter will read the destination port and virtual channel (VC) from the packet header.

Figure 3-27: SNQ Instruction: HT_RAW

HT_size is the number of quadwords that should be sent. With the **AH** bit set the standard notification header is appended at the end of the RAW packet by the HT send unit. This feature is very helpful for sending remote notifications over VELO as detailed in chapter 3.9.3.

3.7.6 Write RF

To be able to write to the RF the instruction in figure 3-28 can be used.

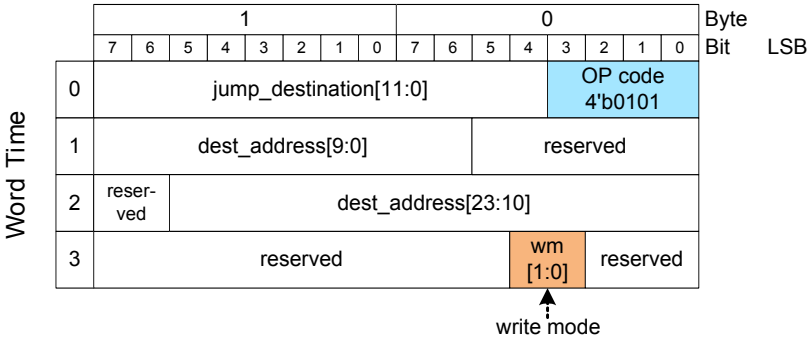


Figure 3-28: SNQ Instruction: WRITE_RF

The value that is written to the dest_address in the RF depends on the write mode:

- 2'b01: write 0
- 2'b10: all bits set to 1
- 2'b11: write the value from the write_data register in the RF the handler

The first possibility could be used for example to set a counter to zero, and the second one could be used to set enable bits.

However, there are also cases where arbitrary values have to be written to the RF, but the problem is that the instructions word is not big enough to embed a 64 bit value as operand. Consequently, two instructions have to be used in that case. First a LOAD_PS has to be done that reads the arbitrary value from the RF. The value could either be stored in unused segments of the SNQ program memory or at other locations in the RF. Then the value will be in the *write_data* register of the RF handler unit and can be written with the WRITE_RF instruction. The LOAD_PS instruction will of course also write to the PS, but this is not relevant in this case.

3.8 Microcode Generation

The microcode that runs in the SNQ is produced by a small compiler named SNQ Compiler (SNQC). At first it was unclear if the microcode should directly be generated by RFS or by an extra tool. To have a logical separation it was decided to pursue the microcode generation in an extra tool, because the register file implements the *trigger* bus in actual hardware and therefore they can not be changed easily. The TDGs and extra code that can be used in the SNQ can be changed later on and also at runtime.

3.8.1 Input Files

Three different inputs are used to compile the binary microcode output. Two of these files, namely snq_te.txt and snq_tdg.txt, are automatically generated by RFS. Automatically generated files should obviously not be changed by the user, because otherwise the modification would be necessary every time the file is regenerated. For this reason a third file (extra.asm) exists which can be used to overwrite and extend the auto generated code. The complete flow is also shown in figure 3-29.

All input files were designed and defined with simplicity in mind, so that they can be produced and parsed without unnecessary big development effort. At the end of this section an example will be shown that illustrates how the different components work together to generate the microcode.

snq_te.txt

Contains the trigger events that were generated by RFS, the format is very basic. Each line in that file represents a trigger. The line contains the number of the trigger and the name of the trigger.

```
<number of the trigger in decimal> <name of the trigger>
```

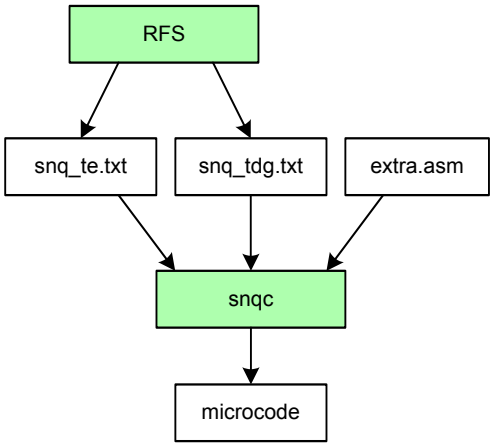


Figure 3-29: Microcode Generation Flow

snq_tdg.txt

This file contains the information which registers belong to a trigger event. Each line starts with the name of the trigger and is followed by a list of register addresses that must be stored to the SNQM.

```
<name of the trigger> <register address 1>...<register address n>
```

Each address can be followed by an option that is separated with a "+". The SNQC supports the following options:

- W0: write zero to the register after reading it
- W1: write 64 bit with all bits set to one to the register
- WB: write back exactly the same value that was read if it is not zero
- WBNZ: like WB, but the writing of the notification happens only if at least one value that was read is unequal to zero

Extra Assembler Code - extra.asm

It is necessary to have a method to supply extra code to the SNQC in addition to the auto generated code. The extra assembler code allows extending and replacing the programs that are auto generated.

There can be extra TEs in the SNQ that have no representation in the RF, and the extra code can be used to implement functionality for these. The SNQ instruction set offers more features than for just storing registers and the extra code is the way to provide these instructions to the SNQ.

Furthermore, there can be TEs in the RF that have no TDGs defined. In that case it also has to be possible to implement functionality.

The code consists of blocks. Each code block starts with a colon and a label:

```
:<TE name> [option]
```

The TE name is either as given in the TE attributes in the XML for RFS or TE-<number> and the number is also the bit number in the TC registers of the SNQ.

Code blocks without option can only be used when there are no TDG elements that match the TE. Otherwise, an option has to be supplied, which controls when this code block will be executed relative to the code that is auto generated based on the TDGs:

- pre: before the auto generated code
- replace: completely replaces the auto generated code
- post: after the auto generated code

The end of a code block has to be marked with a single line that contains only two colons:

```
::
```

Comments that are using two slashes and blank lines are allowed in the code to improve readability. The assembly instructions are detailed in the next subsection.

3.8.2 Assembly Instructions

A small DSL (Domain Specific Language) was developed for the extra assembler code. The description uses the following syntax:

```
INSTRUCTION [optional options] <mandatory parameter> [optional parameter]
```

The options begin with a "+".

LOAD

The LOAD instruction allows loading a value from an address in the RF and storing it to a location in the prepare store.

```
LOAD <PS dest address> <RF src address> [count]
```

The count value determines the number of consecutive RF addresses that should be read from the RF and written to the PS.

LOADI

There are two reasons to load immediate values, either to write them to the RF or to put together HToC packets in the PS. The immediate values are stored in the program memory at locations that are not used by the microcode programs and a LOAD microcode instruction is used to transport the data into the PS.

```
LOADI <PS dest address> <immediate value 0> ... [immediate value 6]
```

By loading multiple immediate values in one assembly instruction it is possible to store up to seven immediate values at consecutive addresses in the program memory to load them with a single LOAD microcode instruction into the PS.

WRITE_SNQM

This instruction allows writing count 64 bit registers from the PS to the SNQM.

```
WRITE_SNQM <count>
```

WRITE_HTOC_RAW

With this instruction it is possible to send count 64 bit registers from the PS as HToC packet to the HT to RF converter.

```
WRITE_HTOC_RAW [option] <count>
```

The only supported option is "+AH" it will append the standard notification header at the end of the packet.

WRITE

The WRITE assembler instruction can be used to write from the SNQ to the RF.

```
WRITE <RF address> <value>
```

The value can have one of the following values:

- 0: write 0
- 1: write with all 64 bits set to 1
- 3: write the value that was read last by LOAD or LOADI

FIFO

The FIFO instruction is very similar to the LOAD instruction, but it does not increase the RF address.

```
FIFO <PS address> <RF address> [count]
```

RAW

When a microcode instruction or an operand of a microcode instruction is not implemented as assembler instruction then the RAW microcode instruction can be used to place an arbitrary instruction in the microcode. However the jump_destination part is overwritten with the correct content.

```
RAW <hex value>
```

3.8.3 SNQC Implementation

Due to the design of the input files the actual process of generating the microcode in the SNQC can be divided in five parts.

1. Interpreting snq_te.txt and creating a data structure with all events
2. Parsing of snq_tdg.txt and adding the instructions to the corresponding events in the data structure.
3. The extra.asm is used in the same way like the snq_tdg.txt but there is extra markup that allows telling SNQC what should happen to code that is already supplied for an event by the snq_tdg.txt.
4. The data structure is transformed in code by resolving the addresses and bringing the code in the right order.
5. A report has to be generated so that the user level software can be programmed that evaluates the content of the SNQM.

SNQC is optimizing register file reads by combining consecutive reads with increasing addresses in a single LOAD_PS instruction. Hence, the size of the program code can be reduced.

3.9 The Integration and Application of the SNQ

The SNQC can be used to implement all the functionality that was defined in chapter 3.4. In the following it will be demonstrated how this can be archived with this defined and implemented architecture. These examples may also clear up the remaining open questions regarding the design.

The SNQ is a central element of the EXTOLL NIC, because it is not only used to read out status, performance and debug information, but also to implement the interrupt mechanism for the RMA and VELO units.

3.9.1 Interrupts

The necessary RF infrastructure for the RMA or VELO interrupts can be described in the XML definition in the following way:

```
<repeat name="intr" loop="4">
    <reg64 name="n" tdg="velo_intr+WBNZ">
         <hwreg sw="rw" hw="wo" width="64"
              sw_write_xor="1" sticky="1" te="velo_intr"/>
    </reg64>
</repeat>
```

The example is using the VELO, but for RMA everything is the same. It makes use of the repeat element of RFS to instantiate the contained reg64 multiple times. The result of the XML definition is that the RF will have four 64 bit wide inputs *intr_0_n* to *intr_3_n*, because of the **te** attribute a velo_intr event will be generated when any of these registers is changed. The advantages associated with using the sticky attributes in combination with the sw_write_xor attribute are detailed in chapter 2.4.2.

When a velo_intr event occurs then these four 64 bit register are read by the SNQ and if they are unequal to zero, then they are directly written back to clear the set bits.

The SNQ will store the values from these four reg64s to the SNQM only if any of these is unequal to zero, because of the WBNZ option in the tdg attribute. This is useful, because the arbiter unit in the SNQ is activated again as soon as the microcode engine starts executing the code for a TE. Therefore, it can happen that a new velo_intr TE is stored in the arbiter unit, while the SNQ is just starting to read out the four registers. Thus, the bit that caused the new TE is probably already set back, when the microcode starts reading out the four registers again. Hence, the WBNZ option prevents unnecessary notifications that contain only zeros under such circumstances.

Thus, 256 bits are available in the four hwregs, which is the amount of VPIDs that are supported at the moment, and the VELO only has to set the corresponding bit when there is an interrupt for a queue.

The rest of the interrupt handling is performed by the RF, the SNQ, the OS kernel and the device driver. Due to the efficient instruction format only a single LOAD_PS microcode instruction is required for this operation.

3.9.2 Counters

There are two ways to set back counters, when they are directly writeable, then the "+W0" option can be used in the TDG definition. The counter named cnt1 in the following example will be read out and set to zero when the eventA occurs.

```
<reg64 name="cnt1" TDG="eventA+W0">
    <hwreg counter="1" width="32" sw="rw" hw="" rreinit="0"/>
</reg64>
```

Another approach is necessary, when the rreinit feature is used, because the counters are not writeable. The following XML in the RF will generate sixteen 32 bit counters and when the TE eventB is emitted then the SNQ will read out all sixteen counters and write them to the main memory.

```
<reg64 name="initcnt"> <rreinit /></reg64>
<repeat loop="16" name="test">
    <reg64 name="cnt" TDG="eventB">
```

```
            <hwreg counter="1" width="32" sw="ro" hw="" rreinit="1"/>
        </reg64>
    </repeat>
```

To reset these counters it is necessary to write the **initcnt** reg64, under the assumption that the address of it is 0x1000 then this can be done with the following code block in the extra.asm file that is supplied to the SNQC.

```
:eventB post
WRITE 0x1000 0
::
```

This WRITE instruction will be appended at the end of the auto generated code for eventB, because of the "post" option. Thus, as soon as all sixteen reg64 are read and stored to the SNQM the initcnt reg64 at address 0x1000 will be written and all counters will be reset to zero.

3.9.3 Remote Notifications

There are, as mentioned in chapter 3.4, usage scenarios for EXTOLL NICs where there is no host system connected to an EXTOLL NIC. One example is using an EXTOLL NIC card as switch component.

VELO

The VELO was introduced in chapter 1.3.1 and can be used to send small messages into a receive queue of another node. Sending of messages is done by writing a VELO header and the data to the VELO FU, which is connected to the HTAX crossbar.

It is possible to generate the necessary HToC packet with a SNQ program. The following code demonstrates how such a program could be implemented for an eventC.

```
:eventC replace
LOADI 0 <HToC header 1> <HToC header 2> <VELO header>
LOAD 3 <some RF address>
WRITE_HTOC_RAW +AH 4
::
```

The LOADI lines puts the HToC and VELO headers in PS0 to PS2, and the LOAD line puts the content of some RF entry into PS3. Furthermore, the packet is sent with the WRITE_HTOC_RAW assembly instruction to the VELO. The count has to be 4, because the +AH option attaches the SNQ header at the end. It is not necessary to attach the SNQ header, but it includes the time and the instruction word that triggered the sending of the packet. Therefore it is useful for the software that evaluates the data on the other node.

Figure 3-30 shows the resulting events of such an implementation step by step.

1. RF sends a TE to the SNQ
2. SNQ reads from the RF and puts an HToC packet together in the PS

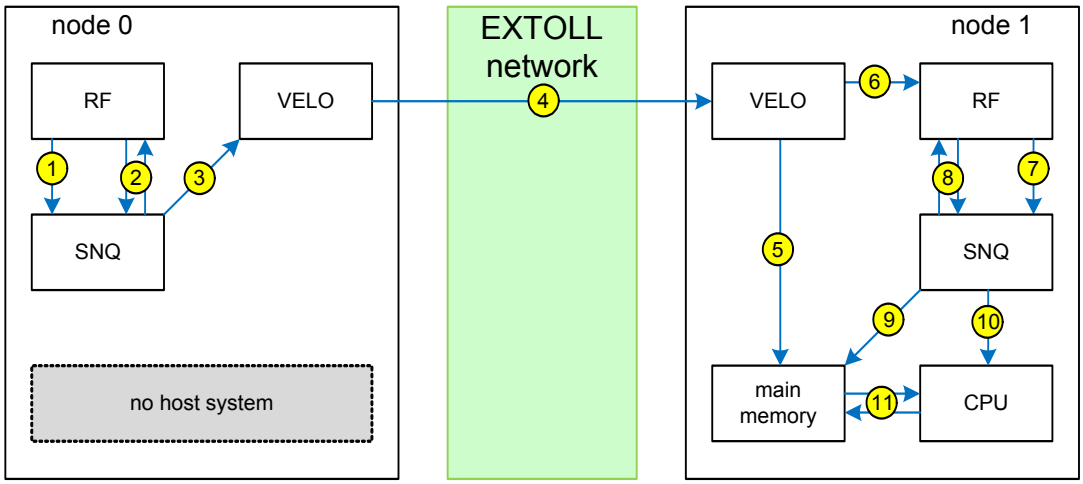


Figure 3-30: Remote Notifications with VELO

3. SNQ sends the prepared HToC packet to the VELO (node 0)
4. VELO (node 0) sends a network packet over the EXTOLL interconnect
5. VELO (node 1) writes received data to the main memory
6. VELO (node 1) sets corresponding interrupt bits in the RF
7. RF sends a TE to the SNQ
8. SNQ reads the VELO interrupt registers from RF
9. SNQ writes the VELO interrupt registers into the SNQM
10. SNQ issues an interrupt
11. Software reads the local SNQM to find out about the remote SNQ message in one of the VELO receive queues

3.9.4 Resource Usage

The resource requirements of the SNQ with an arbiter for 32 possible events and a program memory with 256 entries are shown in table 3-2. These numbers are for the SNQ within a EXTOLL design for Ventoux.

resource	amount
FFs (flip-flops)	1535
LUTs	2009
RAMB36	2
RAMB18	0

Table 3-2: SNQ Resource Requirements

Fast Virtual Address Translation

Chapter 4

4.1 Introduction

Interconnection networks for High Performance Computing (HPC) are built with the intention to provide fast data transfer with low latency between compute nodes. One option is to support remote memory reads and writes or Remote Memory Access (RMA). To implement RMA as efficient as possible user level applications have to be able to perform these operations, without the involvement of the operating system, from the user level.

For safety and security reasons, user level applications should not be allowed to directly access physical memory. Also user level applications can not be allowed to directly provide physical addresses to the NIC, because they would be able to read and manipulate the memory of other processes or the operation system. Even when there are no malicious applications it is still possible that programming errors occur. Such errors can be very hard to debug, because for example when a program overwrites by accident a part of an operating system data structure, the problem could surface only much later under certain conditions.

Therefore, an infrastructure is required that helps to avoid giving physical addresses into the hands of user level applications and allows separating the user level applications from each other based on a process ID.

This chapter describes a hardware unit called ATU2, which implements an address translation service and security checks. ATU2 is in the end a Memory Management Unit (MMU) for a NIC device. The chapter will start with a short overview about the state of the art and continues with a detailed overview of the tasks that have to be fulfilled by the ATU2. Furthermore, the motivations for this development will be detailed. It replaces the predecessor ATU1.

Subsequently, the environment of this new design will be introduced to prepare the explanation of the design principles and the reasons that lead to these approaches. Afterwards implementation details and results will be presented.

4.2 State of the Art

Address translation is of course not a new problem, for example most modern CPUs have an address translation mechanism called MMU.

The implementation of user level access to a NIC by the means of pinned down memory and an address translation mechanism is presented in [12]. In [13] it is demonstrated that the driver for an Ethernet NIC can be implemented in the user level without significant performance impact.

In [15] the influence of different TLB optimizations within CPUs is evaluated. Part of the optimization is an increase of the page size from 4 to 8 kB. This approach will also be used in ATU2.

A detailed discussion about the current state of art can be found in [6], reiterating about the different approaches that are used at the moment makes no sense.

4.2.1 ATU1

The address translation unit done for EXTOLL R1 is called ATU1 and was presented in depth in [6]. It is called ATU1 in the following to differentiate it from the new ATU2.

EXTOLL R2 required a redesign, because of the new functions, feature and performance requirements. Details on the new requirements and the methods that are used to fulfill them will be given in chapter 4.4.

4.3 Address Translation

The task of the ATU2 is the translation of network logical addresses (NLA) into physical addresses (PA) within the main memory. In the context of EXTOLL the NLA is the same like a virtual address (VA) in the context of an operating system (OS) kernel. The NLA can be used to transfer data in the network without using PAs. The exact addressing scheme will be detailed in a later section.

The RMA unit supports put and get operations as described in chapter 1.3.1. The flow of events in the case of a get operation is depicted in figure 4-1. This flow consists of the following ten simplified events:

1. active process on the CPU sends a **get** command to the requester
2. requester sends a data request to the responder
3. responder requests a translation from the ATU2
4. ATU2 sends a response to the responder with the PA

5. responder sends one or more read requests with the PA to the main memory (controller)
6. responder gets read response(s) with data
7. responder sends the requested data over the network to the completer
8. completer sends a translation request to the ATU2
9. ATU2 sends a response to the completer with the PA
10. completer writes the data to the PA in the main memory

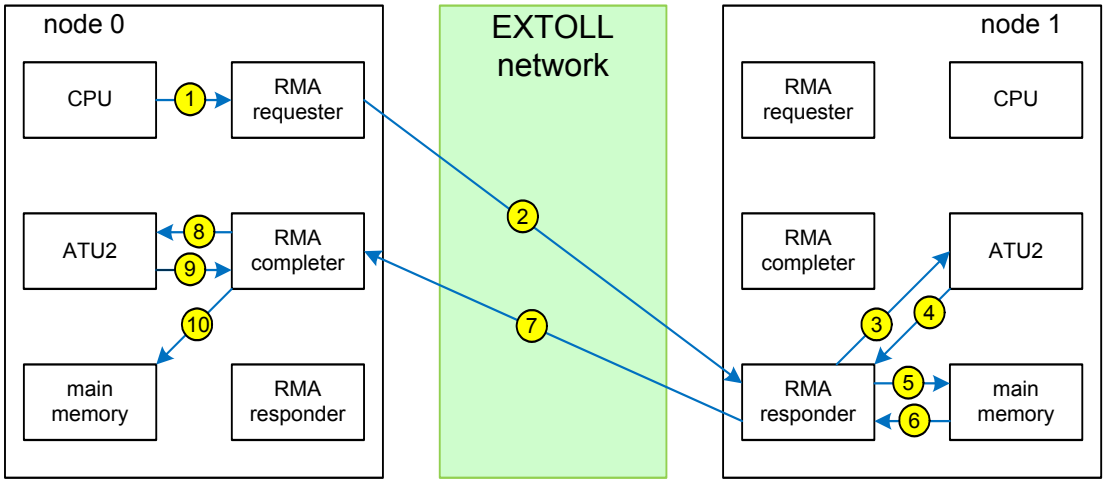


Figure 4-1: Get Operation

Thus, each RMA get operation causes at least two translations, one in **node 0** (source node) and one in **node 1** (destination node).

Every time a RMA operation crosses a 4 kB boundary in the main memory an ATU2 translation request is done. RMA operations can be up to 8 MB in size, and can therefore cause a lot of translation requests (up to 2049).

This figure is only a logical representation of the functionally involved units, of course there are more units involved in the transfer of actual data.

The OS assigns to each process, which makes use of the RMA, a virtual process ID (VPID), and the VPID is used to prevent memory accesses by unauthorized processes. It can not be changed by user level software. The RMA requester determines the VPID by the address range the application is using to write commands to the RMA units in hardware. Thus, the operating system is responsible for the mapping of the right address space to the corresponding process.

One of the reasons why the VPID method was chosen over the protection key [11] is because it has a lower overhead. In short, the protection key is for example a 64 bit key and when the sender includes the right one then the packet is allowed to be received. The VPID is with 8 bits much smaller. Furthermore, the protection key would also have to be stored in the hardware on the receiving side, because otherwise a comparison would not be possible.

The put operation is slightly simpler, but it does also cause two translations by the ATU2. An overview of the different steps that constitute a put operation is depicted in figure 4-2:

1. process on the CPU sends a **put** command to the RMA requester
2. requester sends a translation request to ATU2
3. ATU2 sends a response to the requester with the PA
4. requester sends one or more read requests to the main memory to obtain the data with the PA
5. requester receives read response(s) with data
6. requester combines the put command with the data into a packet that is sent over the network to the completer
7. completer sends a translation request to ATU2
8. ATU2 sends a translation response with the PA to the completer
9. completer writes data to the main memory by using the PA

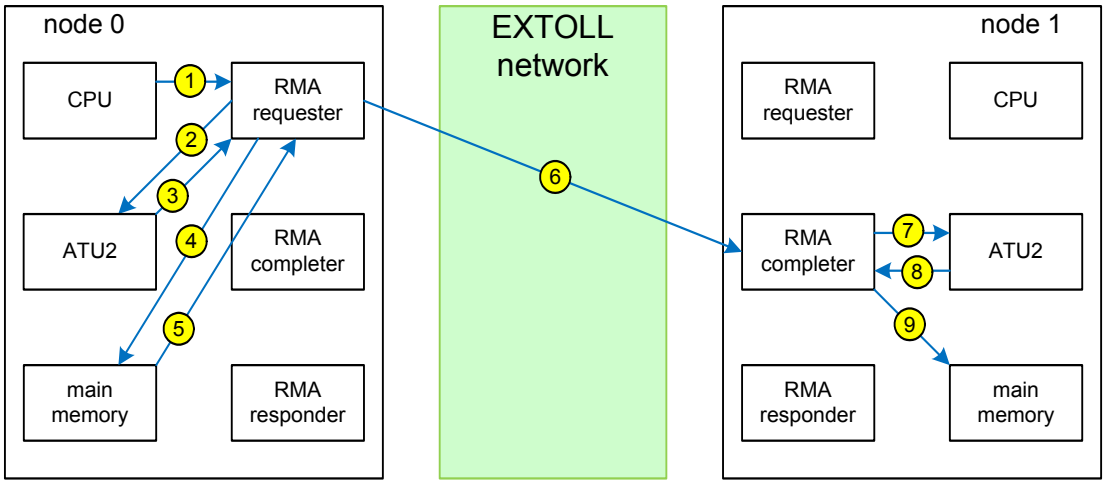


Figure 4-2: Put Operation

4.3.1 Translation Table

The information of how to translate from NLAs to the corresponding PAs has to be available for the ATU2, therefore a translation table is required. This translation table is called Global Address Table (GAT).

This table has to be stored somewhere. Figure 4-3 depicts the different design choices. It is assumed that ATU2 should be able to perform 2^{28} different translations, because given a translation granularity of 4 kB this equals to 1 TB of usable address space.

Support for 1 TB of memory seems to be sufficient, because as of this writing compute nodes with more than 256 GB are very expensive and therefore it seems unlikely that within the next three years the amount of memory per node will be higher than 1 TB.

Furthermore, 1 TB of memory, which can be mapped, is only the limit in the worst case with relatively small 4 kB pages, but the ATU2 is also supporting bigger page sizes. By making use of these, much more memory can be mapped.

Modern CPUs like recent Opterons have a physical address space of 48 bit [3]. However, because in the near future models may be released with support for bigger physical address spaces, it is assumed in this work that 52 bit addresses should be supported by ATU2. The GAT format, which will be detailed in a later section, requires 58 bits per entry.

When rounding up to the next power of two each translation entry requires 64 bits. Given 2^{28} translation entries this translates into a memory requirement of up to $8*2^{28}$ bytes or 2 GB for the translation tables.

Therefore, it is not possible to use the internal SRAM in the FPGA or ASIC to store the translation tables.

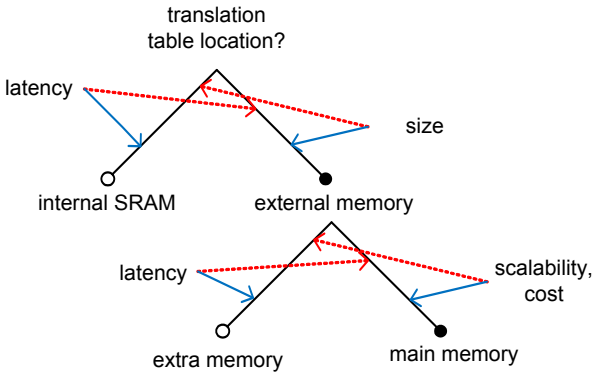


Figure 4-3: Design Space: Translation Table Location

Thus, external memory has to be used. There are two possibilities to have the required space of two GB, either extra memory chips could be added to each EXTOLL card or a part of the main memory could be used to store the tables.

Physical extra memory has the advantage that the latency is lower. Of course one of the reasons for this is that read requests and responses do not have to cross either a HyperTransport or PCI express link, but the bigger advantage is due to the lack of contention. There are two

possible sources for contention, the first one is in the EXTOLL design itself. Figure 4-4 depicts the problem where three units try to issue their read requests at the same time, but there is arbitration between these three. It does not only take time until the read request can be issued, but the read requests of the other units have to handled as well also by the main memory controller and this takes time.

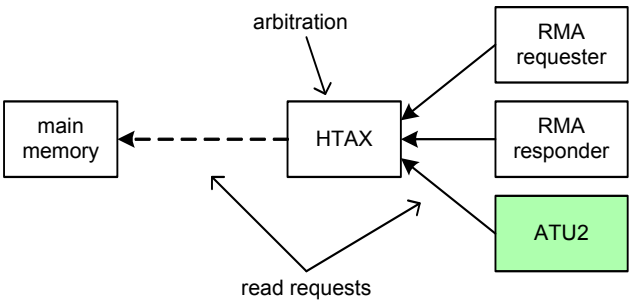


Figure 4-4: Contention

The next possible source for contention is the activity of the host system itself, because it can of course also use the main memory, and additionally it may also have a higher priority than the reads or writes from HT or PCIe devices.

Despite these concerns it was decided for reasons of portability and cost to use main memory. The EXTOLL design is supposed to be portable, so that it can be used on existing FPGA boards, which do not have enough onboard memory. Furthermore, the main memory solution also has the advantage to scale with the requirements. In most cases only a small fraction of the 2^{28} possible translations entries will be used, and thereby also less than the theoretical maximum of 2 GB translation tables will be required. If extra memory on the NIC PCB is used instead, then it has to be 2 GB in size to support enough translation entries under all circumstances.

A fast translation look-aside buffer (TLB) mechanism with a high hit rate is required to make up for the disadvantage due to the usage of the main memory as storage space for the GAT. The TLB is a cache that stores translations to reduce the latency. It is sometimes also called Translation Buffer (TB), Directory Look Aside Table (DLAT) or Address Translation Cache (ATC).

4.3.2 Functionality

To fulfill all required tasks ATU2 has to provide the following main operations:

- complete translation requests
- caching in the TLB

- remove entries from the TLB (flush)
- inform units about the invalidation of translations (fence)

Thus, the task of the ATU2 is to accept translation requests from the RMA and possibly other units. If a match for these requests is found in the TLB, then it is returned, otherwise the GAT has to be read to acquire the translation information. The most important data flows between the involved units are depicted in figure 4-5.

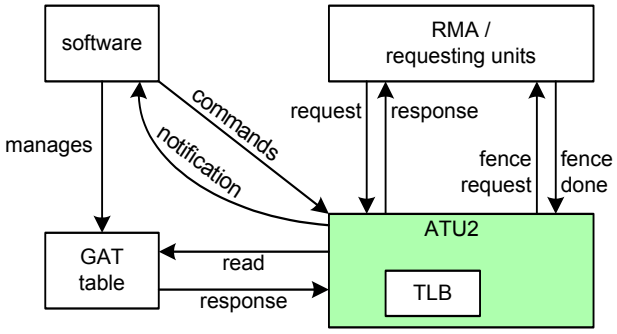


Figure 4-5: Data Flow

Not only the RMA requires translations, but also other units that can be connected to the EXTOLL NIC design with a special interface [48] may require translations. More information about this interface will be given later.

The GAT is managed by a kernel driver. When a process requires memory that can be used by the RMA unit, then the kernel will pin this memory and add an entry to the GAT. When the kernel pins memory this means that it marks the corresponding physical pages so that they will not be paged out to the swap area or moved around within the memory.

At some point it is necessary to delete entries in the translation table, because the memory is freed. This can happen for two reasons, either the application is releasing it, because it is not needed anymore or the application process ends. In the later case all translation entries for the process have to be removed.

The problem is that after a translation entry is removed from the GAT, there are still two places where the translation information could be cached:

- TLB
- RMA / requesting unit

Therefore, it has to be ensured that before memory is used again all translation entries that point to this piece of memory are invalidated. The process of removing entries from the TLB is called flushing.

Subsequently, when a translation is to be removed from the GAT then the following steps have to be performed, which are also depicted in figure 4-6:

1. software removes the entry from the GAT
2. software sends a FLUSH command for the entry and a FENCE command
3. ATU2 executed the FLUSH command and removes the TLB entry, if there is one
4. ATU2 sends a FENCE request to all units that make use of the translation service
5. fence done responses are sent to the ATU2
6. as soon as the ATU2 has received the fence done responses from all units it will issue a notification to the software

Only a single RMA/requesting unit is shown in figure 4-6, but of course there can more than one.

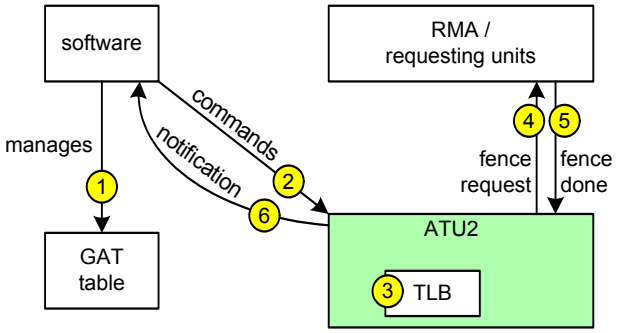


Figure 4-6: Flush and Fence

4.4 Motivation

The prior design ATU1 was done for EXTOLL R1 and it is working. However, the new R2 development has new requirements and a better performance would of course also be beneficial. The following issues are addressed with the new architecture:

- latency
- throughput
- non-blocking flushing
- support for 128 bit wide HTAX
- associativity
- different page sizes
- concurrent memory read requests avoidance

Every put/get RMA operation requires at least two translations by the ATU2, therefore the ATU2 is in the critical path and the **latency** of ATU2 is a significant factor for the overall performance of EXTOLL. Consequently, the latency should be improved relative to ATU1.

The new implementation is based on a pipeline and is able to perform up to one operation per cycle, this allows high **throughput**. However, the throughput is reduced when consecutive operations dependent on each other, because of possible read-modify-write hazards. In the worst case ATU2 can do one operation every five cycles.

An operation is one of the following activities: translation request, read response or control command.

The **flushing** of TLB entries has to be implemented in a non-blocking fashion, so that translations are still possible while the software is flushing entries from the TLB. Furthermore, the protocol to trigger flushing operations should be as simple as possible, and it should allow lock-less operation from multiple threads.

For EXTOLL R2 support for **128 bit wide HTAX** crossbars is required, therefore the new development has to take a width of 64 bit and 128 bit into account and it has to be developed in a way so that only few code is different between the two versions.

One component to improve the hit rate of the TLB is to implement **associativity**. Therefore, support for it is important in ATU2. In [82] reasons are given for direct caches, but the reasons are founded in the amount of used resources, and resources are by far not such a limiting factor anymore in comparison to 1988, when the paper was published.

This opens the way to the question of what type of associativity should be supported. Full and n-way set associativity are the two available possibilities, but full associativity is not an option because it requires content addressable memory (CAM). A small CAM can be implemented of course also in the FPGA, but it would either require a lot of resources, because it would have to make use of flip-flops and comparators instead of special CAM hardware structures, or, it would be slow, because it would use linear search like the CAMs that can be generated with the Xilinx core generator [141].

ATU2 is mainly designed to serve the purposes of the RMA, but it is also designed to provide translation capabilities for an extra unit (EXU) that can be connected to the EXTOLL design. An example for an EXU could be a unit for special calculations or collective operations. The EXTOLL design provides an extension interface that is named excellerate and described in [48]. By using this interface EXUs are able to communicate with the FUs of the EXTOLL design, the host system and the EXTOLL network.

The usual page size that is used by OS kernels on x86_64 systems is 4 kB, but 2 MB pages are also supported by the CPU architecture. Bigger page sizes allow higher performance for many workloads, as shown in [130] for the MIPS R10000 architecture. The reason for this is the reduced amount of TLB misses.

The same principle applies also to translation units for HPC NICs. Therefore, it is of course very interesting to support **different page sizes** in the ATU2. It saves translation table entries and thereby can reduce the number of TLB misses. The 2 MB pages are also called “huge pages” and are used at the moment more and more in applications that require a lot of memory, because they are not only preserving resource in the MMU, but are also reducing the number of required page table entries. ATU2 is also supporting intermediate translation sizes of 8 kB and 64 kB, because often more than 4 kB, but less than 2 MB of memory are required in one piece. Recent CPUs are also supporting page sizes of 1 GB. Consequently, ATU2 should also support this translation size.

Another important issue that has to be addressed with ATU2 is the **avoidance of unnecessary reads** to the main memory.

If a translation is not in the TLB, then a memory read has to be done to acquire the translation, such a memory read takes for example on the Ventoux board 250 ns. This number was obtained in an otherwise idle system; it can take much longer under load. Within this time frame multiple translation requests can arrive at the ATU2, which require a translation for the same page, but the TLB will have no valid translation at this point of time.

There are two reasons for such a situation. The RMA actually consists of three mostly independent units and they could operate at the same time on the same addresses, therefore requests for the same page could happen while the table read is still outstanding.

Due to the support for bigger pages this behavior also gets more likely, because when the pages are bigger then it is more likely that multiple units operate at the same time on it.

The other reason is that RDMA packets on the network have a maximum payload size of 240 bytes, but up to 8 MB puts or gets can be issued with a single RMA command, and the RMA completer will do a translation request for each packet. Hence, multiple consecutive packets with 240 bytes will likely cause a translation request for the same page.

RMA packets with 240 bytes of payload have including header and framing a size of 272 byte on the link and the links of the Ventoux HTX board are currently operating at 1600 MB/s. Therefore, it is possible that every 170 ns a new packet is arriving, and this is faster than the read from the GAT in main memory. Improved EXTOLL implementations will have much faster links, but the main memory access will not get much faster. In the case of PCIe devices the access time will be even higher.

As a consequence, multiple outstanding requests for a page, while the GAT read is outstanding, are common occurrences for every transfer that is bigger than 240 bytes.

Therefore, it was a central design target to avoid unnecessary reads with the help of this development, because positive effects on the overall performance are expected.

For all these reasons a completely new architecture had to be designed, that also takes the limitations of the different target architectures into account.

4.4.1 Technology

The first target platform for this new unit is the Ventoux HTX board with Xilinx Virtex 6 FPGA, which is shown in chapter 1.2.3. Later on, it should also be possible to synthesize the unit for other FPGAs and ASICs. The following three technologies are targeted by this design.

- Xilinx Virtex 6 at 200 MHz, this is the first architecture that will be used for prototyping. The ATU2 is designed with a higher target frequency of 300 MHz, because this way there is enough timing margin and ATU2 will not cause timing problems when integrated in the complete EXTOLL design.
- 65nm ASIC at 800 MHz, this seems to be a modest frequency for an ASIC, but this limitation is due to the available RAMs
- Other high end FPGAs like Altera Stratix IV [49], Stratix V [50] and Xilinx Virtex 7 [71] are also important targets

In all three cases there are SRAMs available with a common ground of one write and one read port. RAMs with different properties like one write port and two read ports have to be constructed from two native RAM blocks or the data path width is reduced. For example, the Virtex 6 BRAMs are only 36 bit instead of 72 bit wide if they are used in true dual-port mode. This would have a significant impact on the resource usage of the TLB.

In the case of an ASIC it should theoretically be no problem to construct such RAMs, but the available generator does not support dual-ported RAMs at the target frequency. However, even if true dual-port RAMs would be available for one technology that would not help, because that would require different implementations for the different target technologies. Therefore the assumption is that only RAMs with one read and one write port are available.

Thus, some interesting features are impossible to be implemented, for example answering two requests per cycle with the help of two read ports at the TLB RAMs. According to [51] it is a common feature for general purpose CPUs like the Intel Pentium or the MIPS R8000, because their caches are supporting multiple accesses per cycle. Luckily such a feature is not required for the ATU2, because the RMA does require much fewer translations.

The reason for this is the access granularity of the RMA, because get and put operations have a bigger granularity than the reads and writes usually done by CPUs. The smallest RMA packet that makes sense has a size of 64 bytes, because for smaller sizes the VELO is the right unit. Transferring 64 bytes over a 64 bit wide HTAX takes including header ten cycles. Therefore even when the different RMA units are doing reads and writes at the same time only every five cycles a new translations will be required, because the host interface will not be able to handle more packets to or from the RMA units.

4.5 Interface Architecture

Address translation is simple, from a very high level point of view. A unit wants to translate a network address, and therefore a request is sent to the ATU2. Either the translation is cached in the TLB or a read from the GAT in the main memory of the host system has to be done. If no translation is available the result is an invalid translation.

Therefore, interfaces have to be provided for the different units that may ask for translations. The design space allows different solutions, but the best one has to be determined before the internal structure of the ATU2 can be designed. The reason for this is that the design is interface driven, the type of interfaces and especially their flow control properties dictate the inner workings to a certain degree. On the other side the feasibility of an implementation has to be taken into account, it makes no sense if the interface description requires functionality, which is impossible to implement. Also unnecessary high effort in terms of development effort and resource usage is not acceptable.

4.5.1 Network Logical Address

The size of the Network Logical Address (NLA) is 64 bits, and it is used to address memory from the RMA unit or an EXU. It has to be decoded before it can be translated, because the actual translation is based on the Network Logical Page Address (NLPA). The NLPA is the logical address of a memory area, which is called Network Logical Page (NLP), this area can have the size of a single 4 kB page or it can be bigger.

The base page size of 4 kB was chosen because the target platform of EXTOLL is x86_64 and this is one of the supported page sizes of this platform. Furthermore, the architecture supports 2 MB pages. In recent implementations like the Opteron 1000/2000 and 8000

series also 1 GB pages are supported [78]. When the granularity of the data allows using such big page sizes this can save translation overhead. According to [78] recent AMD Opteron CPUs have 512 TLB entries for 4 kB pages and 8 for 2 MB pages.

ATU2 is supporting these three sizes. At first only 64 kB were supported as additional NLP size. This extra size was chosen because it is relatively likely that 64 kB can be allocated consecutive in the physical memory and also several translation entries can be saved in comparison to entries for 4 kB pages.

A similar method, for the Linux virtual memory system, called page clustering is described in [52]. It describes how the data structure, which is usually used only for a single page, can be used for multiple pages.

In terms of the encoding of the different data structures and development effort it would be no problem to support more different sizes in ATU2, but timing and hardware resource are a limiting factor.

Thus, the question came up how many resources are really required to support additional NLP sizes. Therefore, a version with support for four (4 kB, 64 kB, 2 MB, 1 GB) and a version with additional support for 8 kB NLPs were compared. Support for 8 kB is especially interesting because two 4 kB pages can be combined to save GAT entries. The results, as shown in table 4-1, are post-synthesis results for Xilinx Virtex 6 and as it can be seen, the amount of logic resources rises only by 1.3%. Therefore, support for 8 kB was added to ATU2.

Resource	four NLP sizes	with 8 kB added
Flip-Flops	2646	2646
LUTs	2558	2592

Table 4-1: Comparison Support for 4 and 5 NLP Sizes

Comparing post place and route results was not directly possible, because the difference between these two implementations is smaller than the differences between single runs for either implementation. Xilinx ISE uses heuristic methods for placement. Therefore, ten runs were made for each implementation and the average was calculated. The results are, with an increase of 1.4% in logic resources, comparable to the post-synthesis results.

Therefore the following five NLP sizes are supported by the hardware:

- 4 kB
- 8 kB
- 64 kB

- 2 MB
- 1 GB

The start address of the NLP in physical memory has to be aligned to a multiple of their size, because otherwise the offset could not be handled without calculation, and the pages would not match the ones supported by the host CPU and OS. Furthermore, the offset would also have to be stored.

TLB Index

The TLB index is the address that is used to access TLB RAMs.

If a TLB is only supporting a single page size, then the **index** within the TLB can be calculated very easily by selecting the least significant bits of the page base address and using them as index for the TLB, but when there are multiple different NLP sizes, then this is not possible.

For example in the case of a 4 kB page the 12 LSBs of the address are not used for page addressing, because they are the offset within the page. Consequently, the next 8 bits could be used as index for the TLB with 256 entries as depicted in figure 4-7. If exactly the same bits are used in the case of 2 MB pages, then it may happen that the TLB has in the end 256 times the same entry, because the TLB index was derived from within the page offset of the 2 MB page.

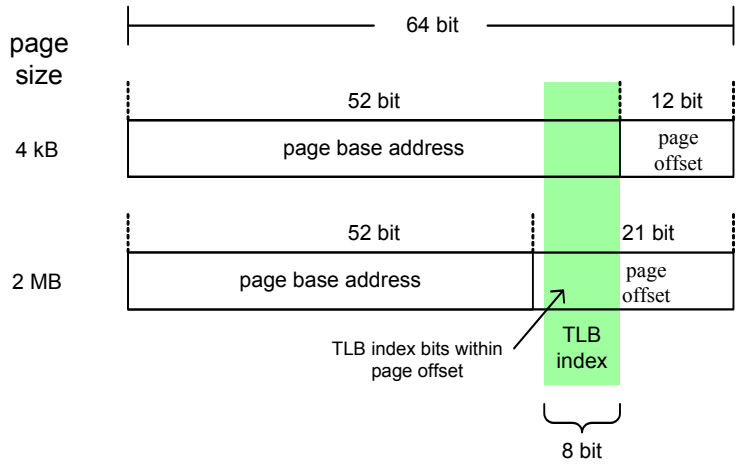


Figure 4-7: TLB Index

It is also no option to derive the TLB index from a range that fits a big page size, because if this approach would be followed in the case of the ATU2 with its NLP sizes of 4 kB and 1 GB then 2^{18} consecutive 4 kB pages would use the same TLB index.

In [131] this issue is discussed in detail for two different page sizes, the different design possibilities were of course also faced with ATU2, but due to the slightly different nature of the address translation in the case of a NIC, another possibility arises. The size could be encoded within the NLA. Furthermore, it should be kept in mind that ATU2 is using five instead of only two different page sizes.

The difference of the addressing in the case of RMA and ATU2 as compared to the one used in CPUs is that it is possible to decide how it is done. Therefore, all options regarding the implementation of a TLB with different page sizes are available.

First, the design choices, as presented in figure 4-8, should be considered for the case if the size is not encoded in the NLA.

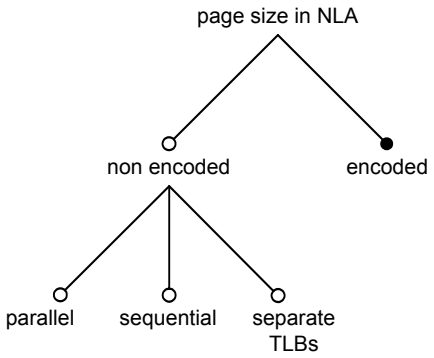


Figure 4-8: Design Space: Page Size Encoding

For different page sizes it makes sense to use different bits of the address as TLB index and these bits should be definitively not part of the offset within the page. Thus, different TLB entries have to be looked up to find out if an entry is contained within the TLB or not. It could be done in **parallel** by using TLB RAMs with multiple read ports, but as explained earlier this is not an option, because the available technologies for this design do not support multi-port RAMs efficiently.

Sequential access is of course possible, but it would increase the latency for a TLB hit from five to at least nine cycles. ATU2 performs a translation, which is a hit, with one read in five cycles, but would require four more cycles to perform a read for each of the five NLP sizes.

Having a TLB set for each page size would be inefficient, because the distribution of page sizes will be most likely uneven. For example, the 1 GB pages will be used most likely very seldom.

Consequently, the size of the NLP has to be encoded in the TLB. Encoding the size inside the NLA allows extracting the NLPA and using a part of the NLPA as TLB index.

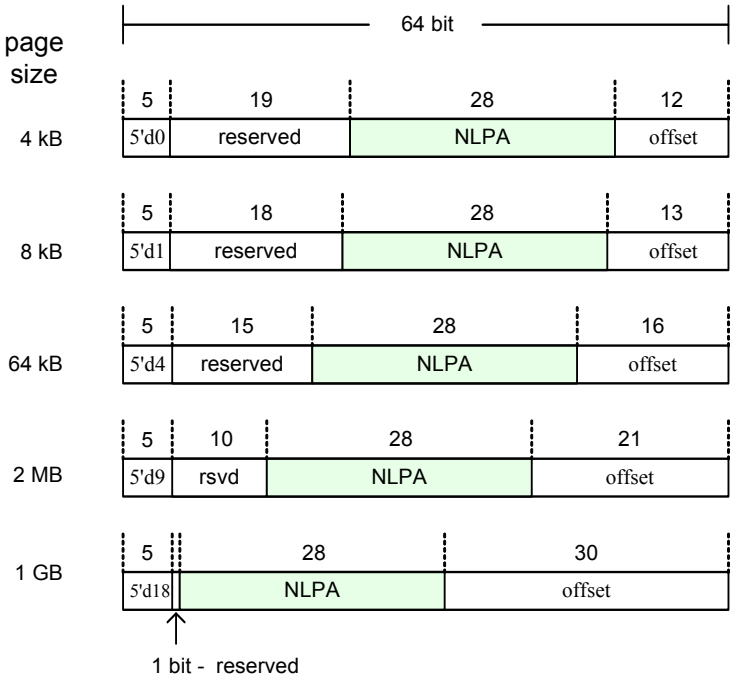


Figure 4-9: Five Possible NLP Sizes

The size is encoded in the five high order bits of the NLA. Calculating the size of the NLP is done with two to the power of this size multiplied with the page size of 4 kB. Thus, if the size field contains a value of nine this means that the NLP is 2^9 * 4 kB = 2 MB big.

The size of 28 bit for the NLPA was chosen because this allows addressing 1 TB of memory for networking purposes even in the worst case of small 4 kB page size. For a system that has and uses a lot of memory resources it is better to use bigger page sizes.

Figure 4-9 shows the five different possibilities on where the NLPA resides in the NLA, depending on the first five bits.

4.5.2 Virtual Process ID

In the EXTOLL architecture user level processes are able to initiate RMA operations, this is performed by writing commands to the hardware. These commands make use of the NLA addressing for their operation. Thus, ATU2 will translate the NLA addresses to physical addresses. This does prevent that user level processes can read or write arbitrary memory.

This could be done as well with an IOMMU [135] as available in modern chipsets. However, the EXTOLL design is using HT and PCIe, but in the case of HT there is no IOMMU available, because the NIC will be directly connected to one of the CPUs instead to a chipset.

Additionally, the usage of the NLA or an IOMMU is not preventing that processes can access the memory of other processes, which are also using the RMA unit, and therefore would also have valid translation entries.

To prevent this, the concept of VPIDs was introduced, which provides process separation. Each VPID marks a group of processes that can communicate with each other. Only processes of the same VPID can communicate with each other over RMA. The VPID is assigned by the management software and stored in the hardware. User level processes are not able to forge the VPID when sending RMA commands to the hardware.

The VPID is the security and safety mechanism for RMA operations. ATU2 supports eight bit wide VPIDs.

For RMA operation it depends on the workload how many VPIDs are required, the sensible maximum is one VPID per process. It is expected that EXTOLL will be used in systems with up to 256 processors. Therefore 256 VPIDs are enough and this is also the amount that is supported by the RMA unit.

However, the GAT format and the request/response interface of ATU2 is supporting 10 bit VPIDs for future extensions.

4.5.3 Request and Response

The format of the requests and responses to and from the ATU2 is independent from the way how they are transported. Either a direct interface is used in the case of the RMA or an HTAX interface can be used. In both cases the same information has to be transferred between the requesting units and ATU2.

The command format for the request is shown in figure 4-10. This translation request format supports up to 32 outstanding translations, because of the size of the source tag, which is five bits.

Source tags are statically assigned to different units in the overall EXTOLL design. The RMA consists of three separate units; each of these can make use of up to eight outstanding translations. The RMA units would also not be able to make use of more. For the EXU eight more tags are reserved. At the time of the writing it is unclear of what type the EXU will be or if such a unit will ever make use of the ATU2, therefore it seem appropriate to allow up to eight outstanding translation requests for this unit.

Requesting units have to set the two LSBs (source tag [1:0]) of the source tag corresponding to their type of unit:

- 2'b00: RMA requester
- 2'b01: RMA completer

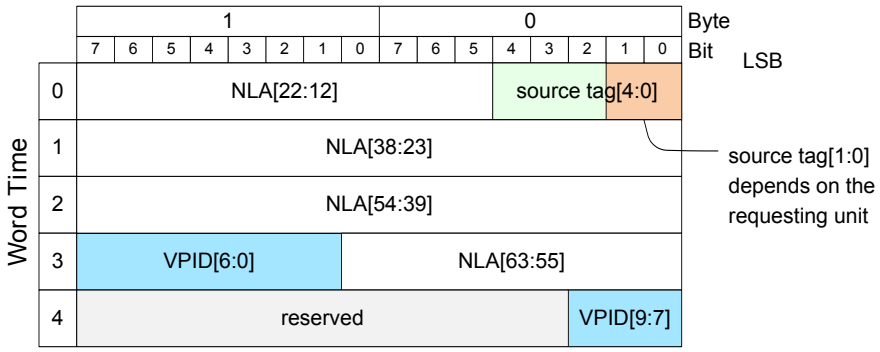


Figure 4-10: Translation Request

- 2'b10: RMA responder
- 2'b11: EXU

Moreover, the remaining three bits in the source tag have to be used in ascending order, because ATU2 will return responses in the same order. Hence it is not necessary for the requesting units to implement reordering for the responses.

The minimum size of an NLP is 4 kB, and therefore the last 12 bits of the NLA are not included in the request. Hence, the requesting unit has to store these 12 bits and combine them with the address given in the response. Subsequently, only 52 bits of the NLA are included in the request.

Furthermore, the VPID is included to check if the GTA entry and the request are matching.

Response

The response as shown in Figure 4-11 contains the PA without the last 12 bits. The response also does not have to incorporate the VPID, because it is known by the sending unit. If the valid bit is not set, then this means that no valid translation was found.

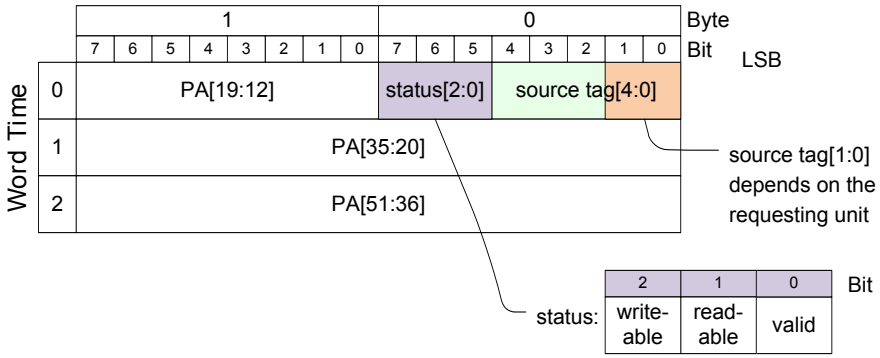


Figure 4-11: Translation Response

4.5.4 Interface to Requesting Units

Units that are making use of the translation service provided by the ATU2 have to be able to transport their translation requests and receive the responses. Therefore, interfaces between the different units have to be defined.

Figure 4-12 shows the different design options. The cleanest design would be to make use of an additional virtual channel (VC) on the HTAX crossbar. This would be clean in terms of interfaces, because an existing interface could be used. The advantage of this is not only of theoretical nature, because big designs have to be partitioned in different blocks to help the tool flow to reach timing closure. Thus, it is desirable to have as few connections as possible between different modules to reduce dependencies.

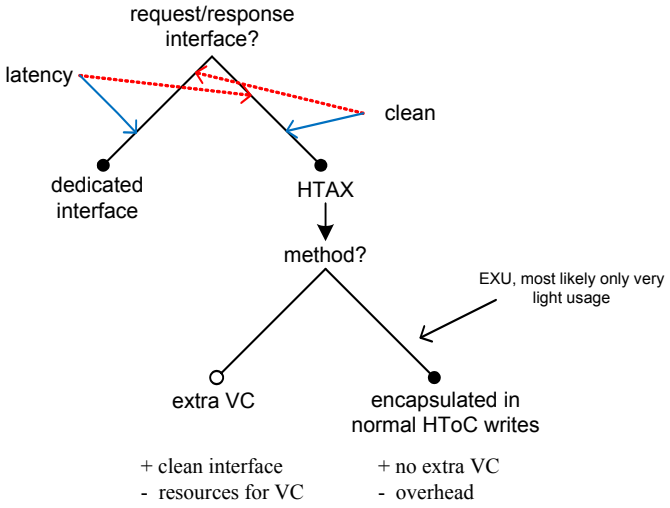


Figure 4-12: Design Space: Interfaces of Requesting Units

However, there are multiple disadvantages associated with this approach. First, the HTAX crossbar would get bigger and more complex, this would lead to a higher resource usage and worse timing in the FPGA or ASIC. Second, the HTAX crossbar does not allow more than one outstanding request of a destination port at the same time. This means that the ATU2 could either try to send a response or a memory read request to the main memory controller at the same time. This could lead to many small stops, because of congestion as explained before.

The biggest reason for the dedicated interface is latency, using the HTAX crossbar would incur at least one additional cycle of latency for each request or response.

Therefore, it was decided that the main path between RMA units and ATU2 should make use of a dedicated interface.

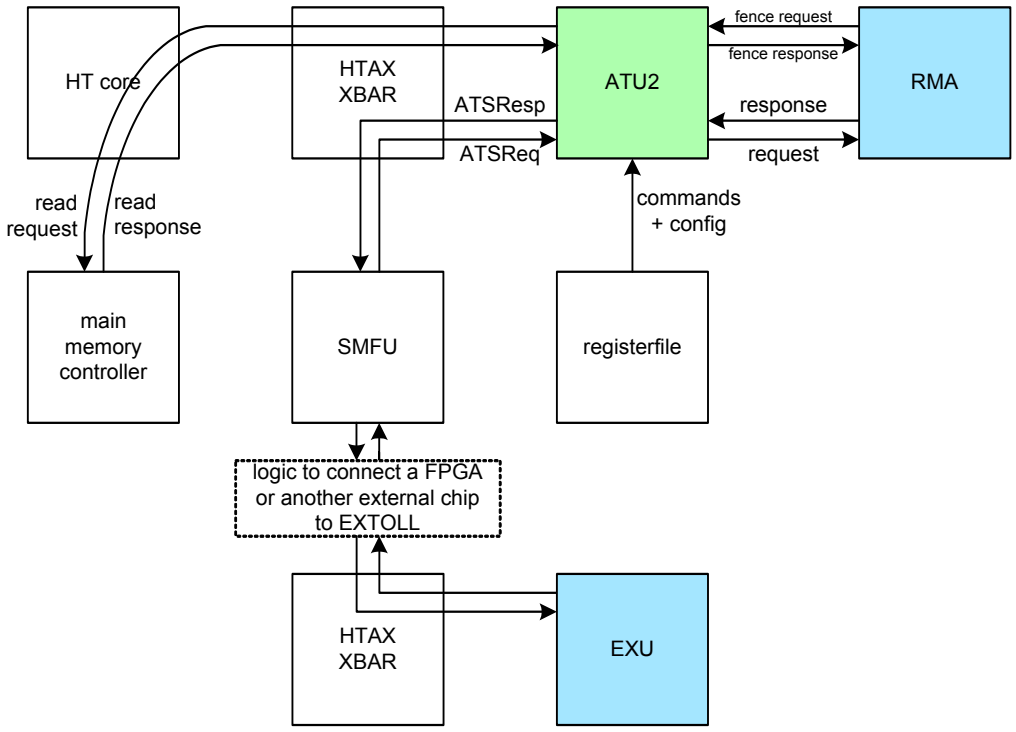


Figure 4-13: ATU2 Environment

Figure 4-13 gives an overview of the ATU2 environment. It also shows an EXU, this unit is connected by the excellerate [48] interface. Therefore it will not be able to make use of a dedicated interface to the ATU2, and thus will have to use the HTAX to use the address translation service (ATS) of the ATU2.

A translation request from the EXU is called ATSReq and the response is called ATSResp.

Despite the RMA consists of three relatively independent units there is only a single interface between the RMA units and the ATU2. The RMA design team has chosen to implement an ATU2 handler unit to channel the interaction between the ATU2 and the RMA units.

Requests and Responses over the HTAX

The EXU has to use the HTAX to submit requests to ATU2 and receive responses.

To keep the implementation of the HTAX interface (ATSReq/ATSResp) simple the approach was chosen to encapsulate the normal translation requests together with a 2 bit packet type indicator into normal posted HToC writes, as introduced in chapter 1.2.1. The disadvantage of this approach is that the packet is bigger than it would have to be when using an extra VC due to the header, in contrast to the approach to use a special VC for the translations. The framing format is shown in figure 4-14.

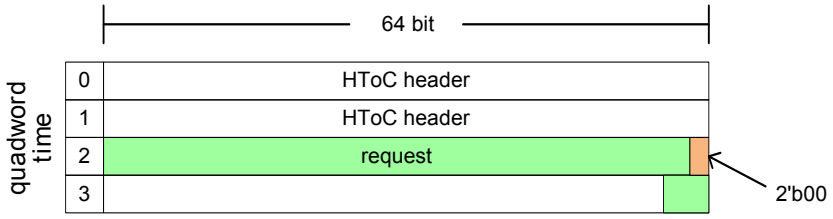


Figure 4-14: HToC ATSReq

But on the other side this way the translations can be handled like normal writes by the routing methods that have to be implemented anyway, because the excellerate unit needs to be able to communicate with the host system and the other FUs of the EXTOLL design.

Thereby, no resource consuming extra VC is required.

Responses, as represented in figure 4-15, are prepared in a similar way, but only one quadword of payload is required, because the response is shorter. The response is also encapsulated in a posted write, and the destination address and destination HTAX port for the responses are used from the RF of ATU2.

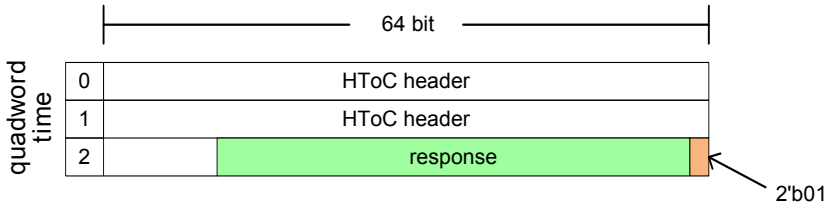


Figure 4-15: HToC ATSResp

Figure 4-14 and figure 4-15 show the encoding in the case of a 64 bit wide HTAX, in the case of a 128 bit wide HTAX the encoding will work as described in chapter 1.2.1.

4.5.5 Fence Interface

Furthermore, also the fence functionality has to be implemented, and therefore there is a *fence_request* signal as depicted in figure 4-16 from the ATU2 to the RMA units. It is drawn for one cycle and orders the RMA units to perform a fence operation. As a result, the *fence_done* signal will be set as soon as all translations in the RMA units before the *fence_request* are discarded.

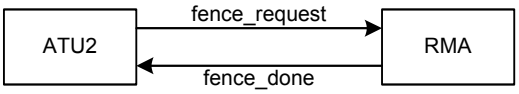


Figure 4-16: Fence Request

The *fence_request* line will not be drawn again as long as there is an outstanding fence request, because ATU2 is only able to handle one fence command at the same time.

Fence over HTAX

The EXU also has to complete the fence, because it could be caching translations as well. The fence request is sent in the same way like the translation response in the HTAX case. Figure 4-17 depicts the request.

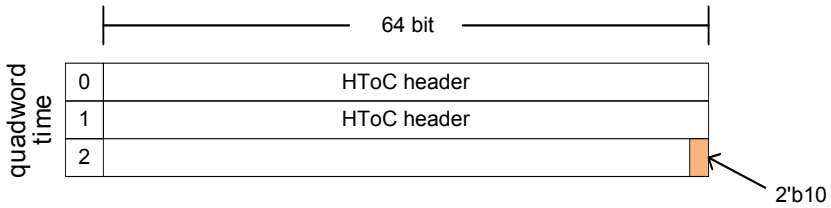


Figure 4-17: Fence Request over HTAX

ATU2 will only send a single fence request until the fence done response was received.

The fence done is sent in the same way like translation requests, and again only two bits of the payload quadword are used as it can be seen in figure 4-18.

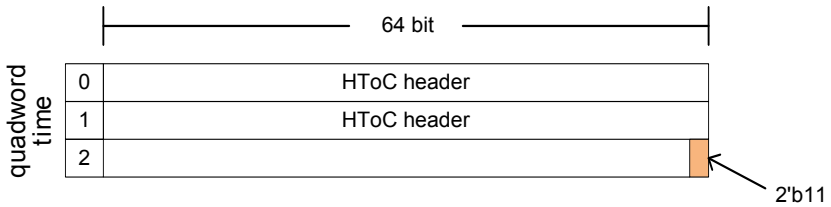


Figure 4-18: Fence Done over HTAX

4.5.6 Global Address Table Format and Lookup

System software is managing the translation tables in main memory, there has to be an entry for each possible translation.

Each of these entries carries multiple fields. For one, the physical address of the translation has to be stored, but also the VPID is required to check if the request is legitimate. Additionally, the size has to be stored in the table, because otherwise user level software could use bigger translation sizes than it was allowed to use, if ATU2 had no possibility to check it. Furthermore, two bits are used to indicate if the entry is valid and what the read/write access permissions are. To store all the information 58 bits are required, therefore each entry is 64 bit in size. This data structure is also shown at the bottom of figure 4-19.

The complete GAT consists of sub tables called Global Address Sub Table (GAST). Each GAST in memory is 2 MB in size. There are 2^{18} entries in each GAST, because each GAT entry requires 64 bit.

Thus, these tables with their 512k entries can be used for the translation between NLPA and PA. The VM subsystems of OS kernels are using smaller multi-level tables instead and also have to perform page table walks for the lookup.

ATU2 is using a two level translation scheme, and thus no latency intensive page walks have to be performed. The first level lookup for one of the up to 1024 GASTs, which are required to support the full 28 bit of NLPA space, is done in a fast SRAM.

There are certain usage scenarios that could take advantage of the possibility to lock entries in the TLB so that they are not evicted by other translations.

There were two possible design choices to implement this functionality. Either an extra TLB set could be used that contains only such entries. This set would be read only for the hardware and therefore it would be no problem if the software would directly control it. However, this would cause a fixed allocation of TLB resources between locked and normal TLB entries. Therefore, an extra status bit was added to the TLB instead, which can prevent the eviction an entry from the TLB.

The **no-eviction bit** in the GAT can be used to mark certain entries so that they will not be evicted anymore from the TLB automatically. It is up to the software to ensure that not too many entries have this flag set.

By adding the no-eviction bit the following possibilities arise:

- performance improvement
- optimization
- debug / fix

The most important application for this bit is to lock especially important translation entries within the TLB, so that they are always available **without delay**. This is of course especially interesting for the big 2 MB or 1 GB page sizes, because they span bigger ranges of memory.

ATU2 is by default featuring a 4-way associative TLB, but a direct or a 2-way associative TLB would save resources. The no-eviction bit can be used to lock complete TLB sets by entries that are of no use. Thus, it is possible to research the performance properties of TLBs with fewer sets without Verilog source code changes.

Furthermore, it would be possible to use the no-eviction bit as **fix** of last resort if the TLB had a bug in the ASIC. Either the number of sets could be limited or the whole TLB could be deactivated by locking complete sets with useless no-eviction bit entries.

Each GAT entry contains five fields with information:

- valid: 2 bits
- VPID: 10 bits
- PA: 40 bits
- no-eviction bit: 1 bits
- size of the NLP: 5 bits

There is no dedicated field that indicates the validity of the entry, one of the bits is indicating if the NLP is readable and the other one that it is writeable. The entry is valid if either of them is set. The possible values for the valid field are:

- 2'b00 invalid
- 2'b01 valid, read only
- 2'b10 valid, write only
- 2'b11 valid, read/write

GAST Lookup RAM

The hardware needs a table to find the corresponding GAST for a NLPA, for this purpose there is the GAST lookup RAM that has 1024 entries. This table resides in SRAM within the ATU2 and provides therefore very low access times in comparison to external memory, and does not introduce any additional latency in ATU2, because the lookup is done concurrently to the TLB lookup. The 32 bit wide GAST lookup RAM can be found on the left side in figure 4-19. Each entry consists of:

- GAST table address: 31 bits (GAST tables are aligned to 2 MB, therefore the last 21 bits of the PA are zero and omitted.)
- valid: 1 bit

Lookup

The lookup, as shown step by step in figure 4-19, of GAT entries begins with the NLA. Depending on the encoded size the NLPA is extracted at the correct position **(1)** of the NLA. The 10 MSBs of the NLPA are used to lookup the GAST base address in the GAST lookup table **(2)**. This address is combined with the 18 LSB of the NLPA to generate the address **(3)** that is used to read the GAT entry **(4)** from the main memory.

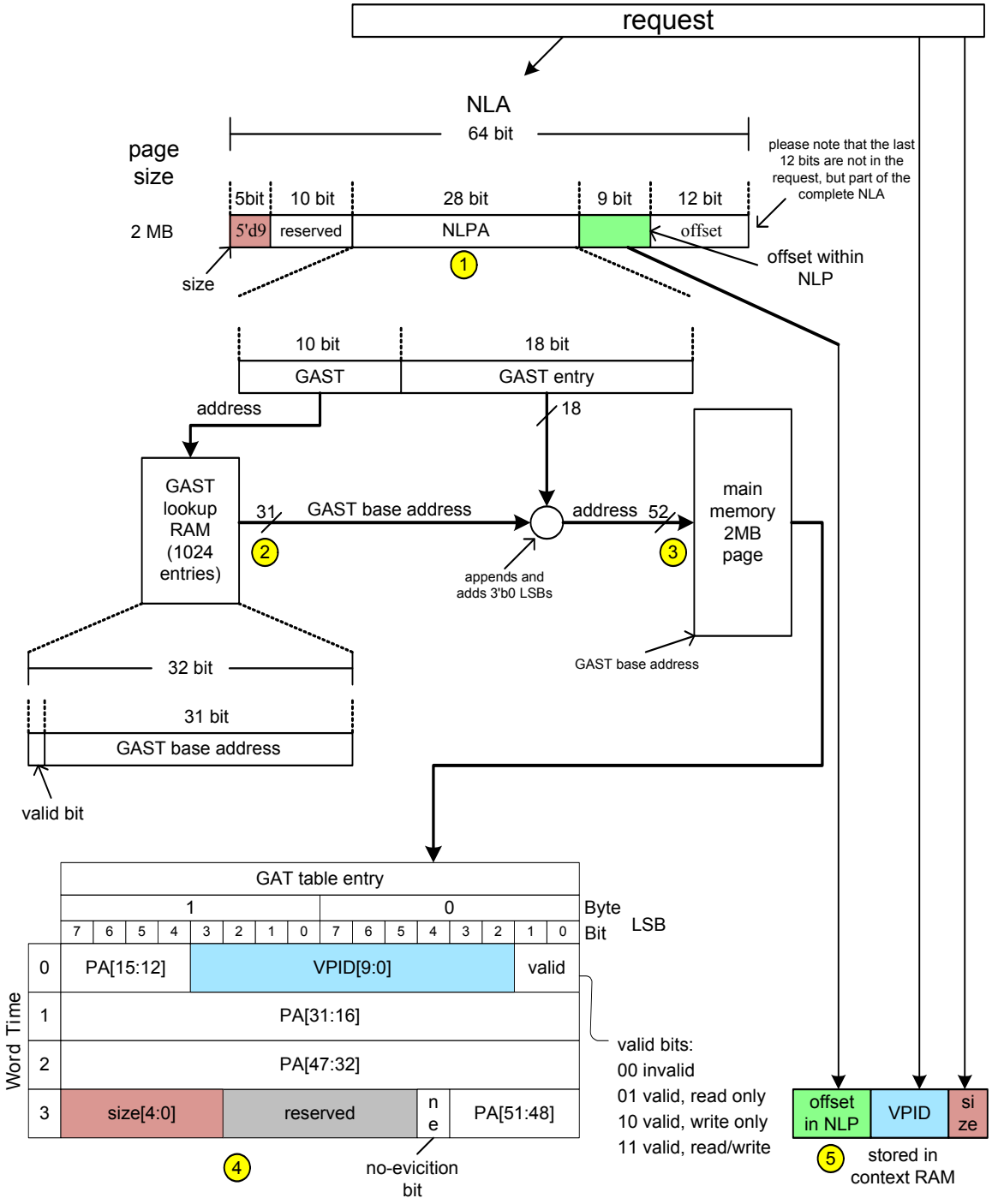


Figure 4-19: GAT Table Read

Additionally, some information is stored **(5)** in the context RAM, because it will be required for the translation.

4.5.7 Translation

After the GAT entry was received from the main memory it has to be used to generate the PA for the translation response as shown in figure 4-20. This is done by combining the PA from the GAT entry with the offset in the context RAM, the number of bits that have to be used from each of these two sources depends on the size of the NLP **(1)**. In the example a NLP size of 2 MB is assumed to stay consistent with figure 4-19. Additionally, the VPID and size have to be compared **(2)** to check if the request was valid.

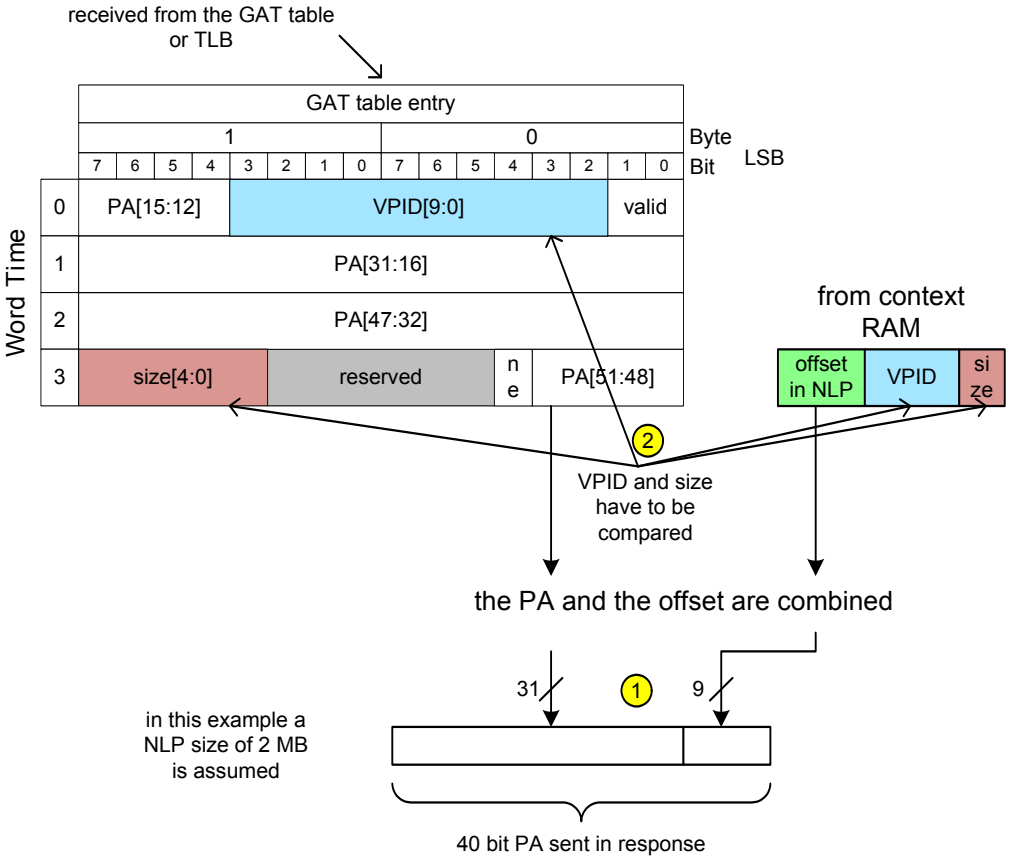


Figure 4-20: Translation

In the case of a TLB hit the translation process is very similar, despite that the GAT information is read from the TLB and the context information is taken directly from the translation request.

4.5.8 Configuration and Command Interface

Certain aspects of the ATU2 have to be controlled by the software, for this reason an efficient command interface was designed. The commands are for the flush and fence operations. Flow control is supported with the help of a notification command.

The command interface is controlled by the kernel, and therefore no protection mechanism is required, because the kernel can access all the memory anyway.

The commands are issued by writing them to the RF. A normal hwreg is used with the sw_written attribute, as described in chapter 2.4.2. By doing so the *cmd* output of the register can be connected to the data input of a FIFO and the *cmd_written* is used to shift the value into the FIFO as depicted in figure 4-21.

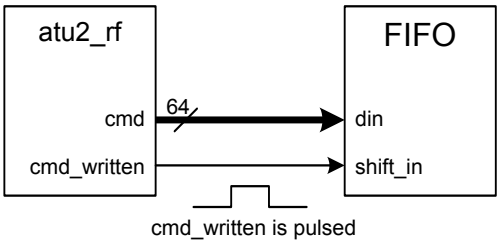


Figure 4-21: Command FIFO

Each command is 64 bit in size, therefore only a single write is necessary to issue a command. Thus, the writing of commands is atomic. This mechanism makes it also possible for multiple kernel threads to write their commands independently without any synchronization.

No flow control is implemented by the hardware. Hence, it is up to the software to ensure that never more than 512 commands are outstanding, because this is the size of the FIFO. Therefore, the software has to be informed when a certain amount of commands was completed, so that it can send more. Otherwise, it would not be possible to implement flow control in software.

There are a few options how hardware can inform software, which are shown in figure 4-22. The classic way for the hardware is to issue interrupts, but the problem with interrupts is not only that they are directed at the OS kernel and not to a single threads of the kernel, but additionally there are multiple reasons why the EXTOLL could generate an interrupt. Therefore, the interrupts handler would have to read out hardware registers that indicate the reasons, and this would introduce latency. The SNQ on the other hand could provide this information directly, but the information still would end up in an extra kernel thread that is responsible for the SNQ and would have to inform the thread, which is waiting for a notification.

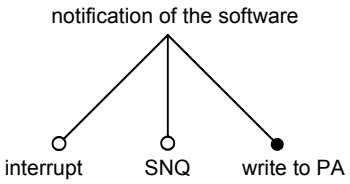


Figure 4-22: Design Space: Informing the Software

Therefore, the hardware provides another notification mechanism. It uses a very simple method. Notifications can either be triggered by a notification command or a fence command. Both commands include a PA within the main memory and as soon as such a command is executed then the memory location is overwritten with a 64 bit write with all bits set to one.

It is feasible in many cases to implement busy waiting, because most command sequences will be finished in only a few hundred ns.

For user level applications it would be for security and safety reasons a bad design decision to make use of physical addresses, but fortunately this is not an issue, because only kernel threads are using the interface.

The size of the FIFO has to be known by the software of course, and then it can write a certain amount of commands to the hardware and at the end a fence or a notification command. As soon as the memory location was overwritten by the hardware the software can be certain that all commands that were written to the hardware are finished.

In summary the notifications can be used to implement a credit based flow control mechanism.

There are different possibilities to implement this. Different threads could statically share the global number of commands or there could be an allocation and a deallocation mechanism.

Flush Commands

Efficient ways to remove entries from the TLB are required for two reasons. First the time the driver or management software is occupied with flushing entries from the TLB should be as low as possible. Second, also the other operations of the ATU2 should not be affected significantly by the flushing.

In figure 4-23 the three commands are shown that are provided with the purpose to efficiently removing entries from the TLB. When a mapping is removed from the GAT, so that the memory can be reused for other purposes, it has to be ensured that the mapping is not used anymore by the hardware, therefore the **Flush NLPA** command was implemented for single entries. It can be used to remove the cached entry from the TLB for a specific NLPA

if a count value of 0 is given. It is also possible to give a count value of up to 1023, and the result will be that the given NLPA and the following 1023 NLPA entries are removed from the TLB. The assumption is that the software will be using the NLPAs in ascending order, and therefore it will be able to take advantage of the count parameter and will require fewer commands when entries have to be flushed.

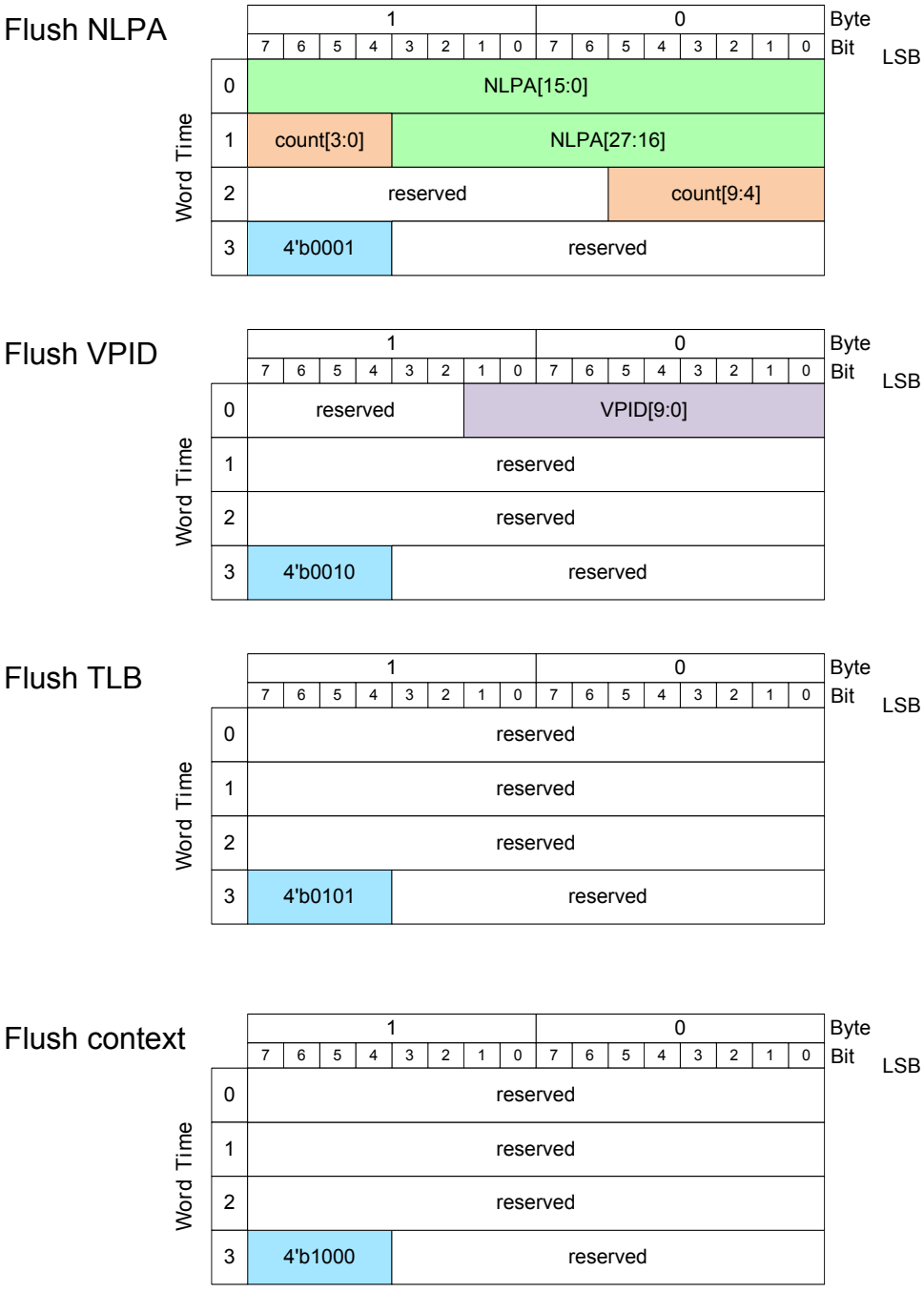


Figure 4-23: Flush Commands

While the Flush NLPA command is for removing single or several consecutive entries, there is also the **Flush VPID** command that is designed for mass removals. In the case a process is terminated and there is no process running anymore for a specific VPID, then all entries for this VPID can be removed from the TLB. The Flush VPID command allows doing this with a single command.

The **Flush TLB** command deletes all entries from the TLB. It can be useful for debugging and testing. Furthermore, if only the host system is rebooted, but the EXTOLL device is not put into the reset state, then the command can be used to bring the TLB into the state that it would have after the reset. This may sound to be only of limited use, because most node systems will reset all devices in the case of a reboot or a warm reset. Unfortunately, the PCI Express Specification does not require it. An example for such a system is the IBM X3650 server, which does not activate the reset line. Furthermore, the reset line could be deactivated willingly in node systems, because EXTOLL is a direct network and will stop working properly when single switches are down. Thus, when the EXTOLL device is not put into the reset state then node systems can be rebooted without affecting the network. Internally the ATU2 is using the very same command after a cold reset to bring the TLB into a known state.

The **Flush Context** command allows deleting the internal RAM that keeps the state of translation requests while the main memory reads are under way. This command is provided purely for debugging purposes, but it is also used internally after the cold reset to overwrite the complete context RAM with zeros before it is used.

Notification and Fence Commands

The group of commands that allow flow control and fences are shown in figure 4-24.

As explained before, the **Notification** command provides the functionality of credits. The software provides a PA in the command and as soon as this command is executed by the ATU2 the memory location is overwritten with ones. The software can be sure that all commands that were injected earlier in the command queue are completed when the memory location was overwritten.

After a flush command was used, to remove the respective entries from the TLB, it also has to be ensured that units that make use of the translation service of the ATU2 are not using the translation information anymore. Therefore, a fence, as described earlier, has to be issued to these units. The **Fence** command includes a PA that is used to write a notification to the main memory in exactly the same way like the notification command.

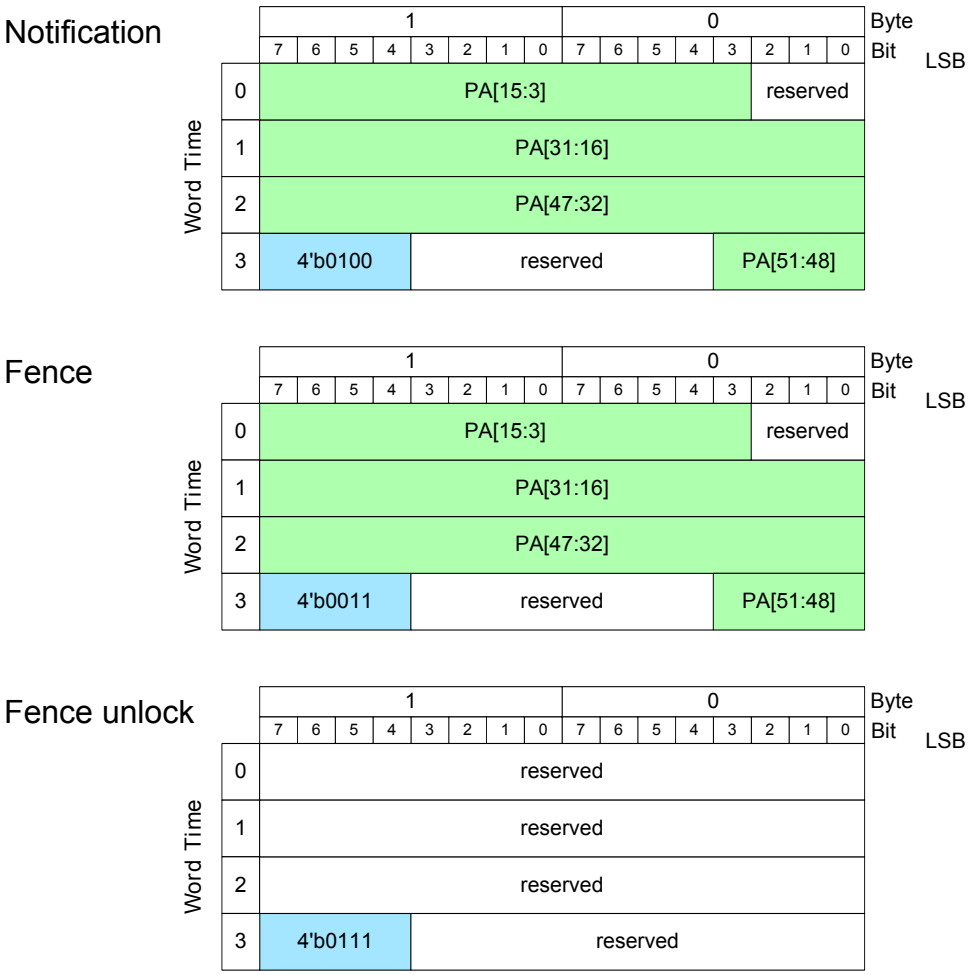


Figure 4-24: Notification and Fence Commands

In the case that one of the units is not responding to the fence ATU2 will be waiting for the responses forever, and thus it will not be able to do any new fences or notifications. If this situation happens it is clearly an error of the not responding unit, but there has to be a way to get the ATU2 out of this state. Therefore, a **Fence unlock** command was implemented.

This command is hopefully never used, because its usage makes only sense if there is a bug in one of the units.

4.5.9 Preload

While a translation is very fast with ATU2 when the translation is already cached in the TLB, it will take relatively long when the translation has to be read from the main memory. Hence, it would be good if there was a way to preload new translations into the TLB before they are needed.

There are two main approaches as shown in the design space diagram in figure 4-25 to load entries into the TLB, the first and most obvious one it to issue a pseudo translation request and the other one is an interface that allows writing the complete translation information directly into the TLB.

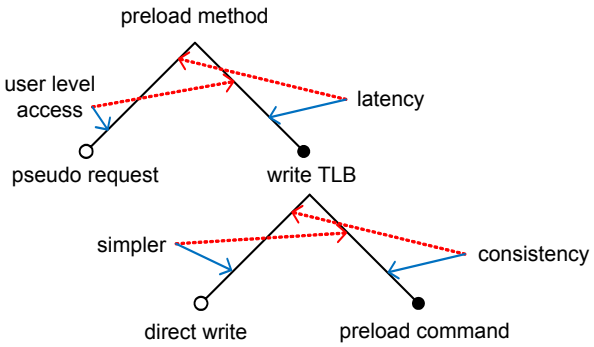


Figure 4-25: Design Space: Preload

A pseudo translation request is a request, which does not deliver a response. As a result the translation information is fetched from the main memory and stored into the TLB. The biggest advantage of the pseudo requests would be that user level processes could be allowed to send them to ATU2, because they could not cause a security problem with them. In contrast, it can of course not be allowed that normal user level processes write directly to the TLB.

This approach has two problems:

- source tags
- latency

ATU2 is designed to work with up to 32 source tags for requests/responses and also to perform memory read requests with the same source tags, if it is necessary to read translation information from the main memory. When the design of ATU2 was started 32 source tags were the limit for read requests to the main memory. Changing this, to support pseudo translation requests, would require a lot of development effort.

The second problem is latency. Figure 4-26 shows a sequence of events, from the host system a pseudo request is sent causing a memory read for the translation information. Furthermore, also an RMA operation is started that causes a translation request to the ATU2, but this translation request arrives before the memory response arrives at the ATU2. This is of course still better than without the pseudo request, but it is also not ideal.

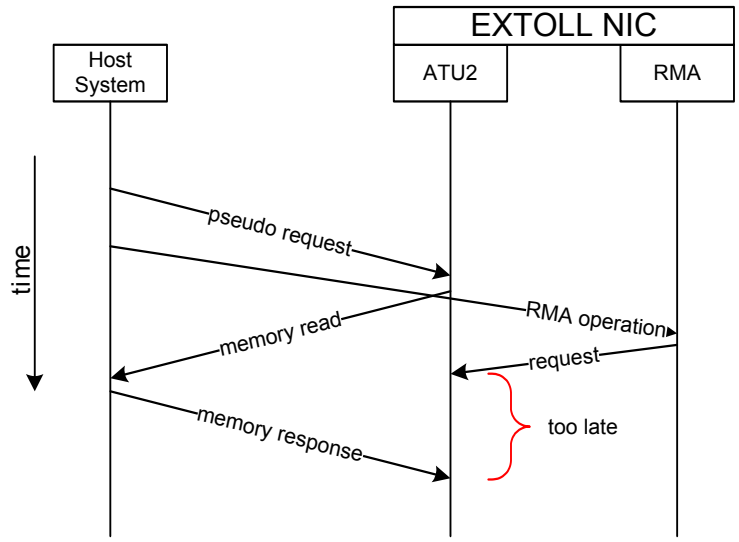


Figure 4-26: Sequence Diagram: Pseudo Request

Thus, the approach chosen is to directly preload the TLB. There are two paths possible for this, either the TLBs could be mapped directly into the RF so that they could be written or a command interface could be implemented.

Directly writing the TLB is not easily possible, because the TLB also does contain status information regarding translations that are waiting for read responses from the main memory. Avoiding possible hazards would incur complexity.

Therefore, the decision has been to make use of a command interface to allow the preloading of translation entries into the ATU2.

The amount of information that is required to preload one entry in the TLB consists of:

- NLPA: 28 bits
- NLPA size: 5 bits
- VPID: 10 bits
- PA: 40 bits
- read/write valid bits: 2 bits
- no-eviction bit: 1 bit

In the sum 86 bits have to be transported from the software towards the ATU2 to preload a single entry in the TLB. These are obviously more bits than fit into one quadword, so the same interface that is used for the flush commands can not be used.

Accordingly, one preload command will require two 64 bit writes, because 64 bit is the native size the EXTOLL RF operates on. Recent x86_64 CPUs provide the possibility to use SSE registers that are 128 bits in size with MOVNTDQ [136] to write 128 bits. However, the specification does not guarantee that the result will really be a single 128 bit write. Experiments have shown that on AMD Opterons it seems to be the case, but on Intel Xeon 5400 CPUs it was observed that the result of a MOVNTDQ can actually be two 64 bit writes in some cases.

The HT to RF converter is also not able to support writes bigger than 64 bits, therefore it is also not possible to use MOVNTDQ in the case EXTOLL is used in a system with AMD Opterons.

Consequently, two 64 bit writes have to be done for a single preload.

The problem is that it has to be possible to preload the TLB from multiple threads. However, writes from multiple threads can be intermixed, when they are arriving at the destination unit. Therefore, it is not possible to implement this in such a simple way as it was done for the command interface from the last subsection.

Figure 4-27 shows the problem with two threads, both try to transport their preload commands, consisting of two writes each, to the ATU2. In this example part0 from thread B is between the two parts that are sent by thread A.

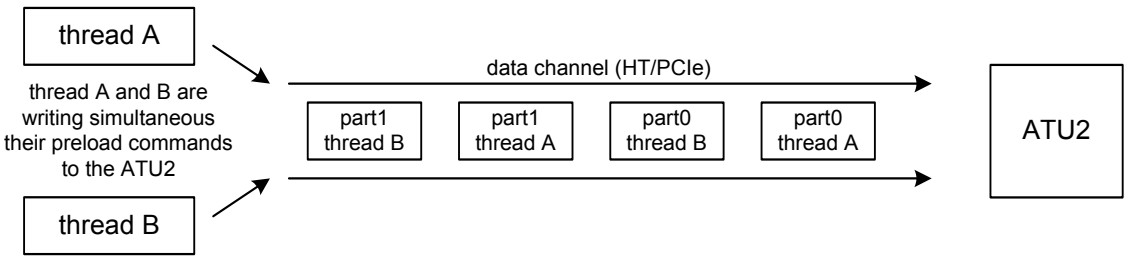


Figure 4-27: Two Threads and Intermixed Writes

Therefore, another approach is required to be able to preload from multiple threads, the fundamental design choices are depicted in figure 4-28. The simplest solution would be of course to use a mutex to prevent that multiple threads are writing at the same time. However, such a software solution would have a performance impact because a mutex would have to be acquired for every preload.

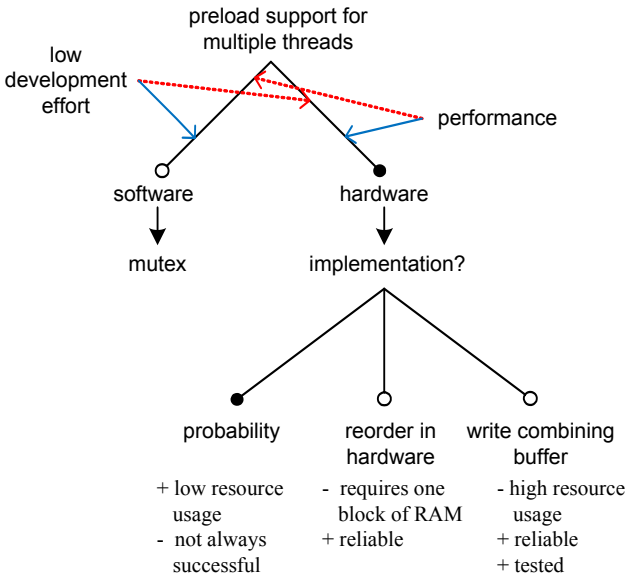


Figure 4-28: Design Space: Preload Interface

Consequently, a hardware solution should be preferred. One obvious choice would be the write combining buffer (WCB) [114], which is also used for the VELO and RMA units. It does have the big advantage that this unit does already exist and it is well tested, due to its usage in the other units of the EXTOLL design. However, it requires a lot of resources.

According to [75] reads and writes to memory regions that are marked as “uncachable” will not be reordered. The memory region of the RF is in such a region as opposed to the write combining memory regions that are used for the RMA and VELO. Therefore not all features of the WCB are required.

Furthermore, the preloading should make use of the RF infrastructure for reasons of simplicity.

In the case of the ATU2 there is the additional advantage, compared to the VELO or RMA, that it can handle the preloads much faster than they can arrive from the host system, because the ATU2 only has to store the content of the preload command into the TLB. Therefore, a very simple WCB can be built that basically only consists of a single SRAM and very few logic.

Thus, ordering is not a problem and an address space of 512 entries of 64 bit could be used for the preloading.

Each thread has to use an own area within that preload space of 512 entries. Each of these memory areas consists of two consecutive memory addresses named **part0** and **part1**. Therefore it is possible for up to 256 threads to use the preloading of ATU2 at the same time. To submit commands the threads just have to write the preload data in two 64 bit

writes to their respective memory area. This preload address space is pure virtual and can be generated in the XML definition of the ATU2 RF in the following way. A description of the ramblock element can be found in chapter 2.4.5.

```
<ramblock name="preload" addrsize="9" ramwidth="64"
    sw="rw" hw="ro" external="1" />
```

The hardware would operate very simple. If a write arrives with the least significant bit (LSB) 1'b0 then it would store this part0 into an SRAM with 256 entries as depicted in figure 4-29. If a write arrives with the LSB being 1'b1 then it would use this new part1 and combine it with the corresponding SRAM entry, containing part0, to form a complete preload command. Each software thread would simply have to write its preloads to its own two 64 bit addresses.

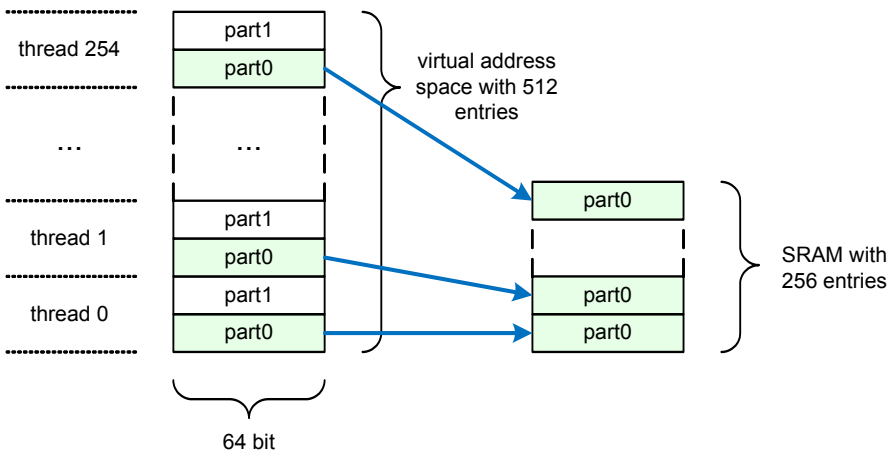


Figure 4-29: Virtual Address Space for Preloading

Despite the information that is necessary for a preload requires only 86 bits the actual preload command has a size of 95 bits, because it contains also a command identifier and it is consistent with the internal command format of ATU2. The 95 bits are split in 31 bits in part0 and 64 bits in part1, because this would allow using a SRAM which is only 31 bits wide.

However, preloads have a big advantage compared to other operations, because there purpose is only to reduce latency, it is functionally not a problem if some of them fail.

Therefore, ATU2 implements a probability method for the submission of preload commands, which does not require the 256 entry SRAM, but works very similar.

As soon as the hardware receives a write for any part0 it will store the written data and the address. Afterwards it will wait for the second part1 of the command by waiting for a write with the corresponding address. All other writes despite the expected write will be discarded. When part1 was received both will be combined and submitted to the execution pipeline of the ATU2.

The expectation is that the probability method works very well under normal workloads, but if this estimation is wrong, then it is still possible to resort to software locking.

As described before the preloading consists of writing two quadwords of data to the ATU2 RF, the necessary information as described before have to be encoded into the two writes, part0 and part1. The encoding is shown in figure 4-30. Please note that the MSB of the NLPA is in part1.

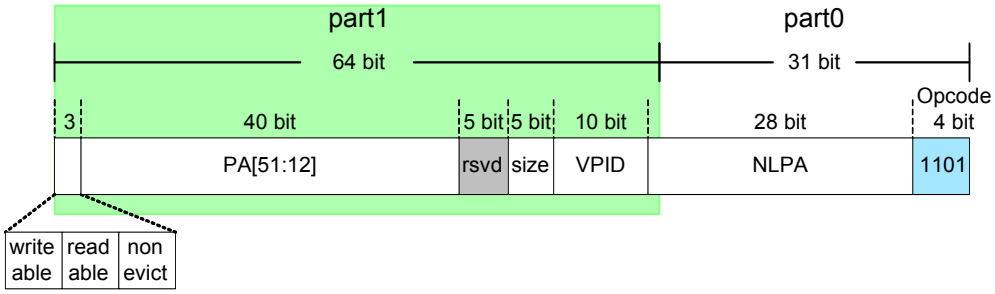


Figure 4-30: Preload Command Format

The encoding of the preload command is also used internally in the pipeline of the ATU2 without further modifications. Therefore, it is also possible to use the very same interface to submit any command that can be handled in the ATU2 pipeline. The instruction format of the pipeline is detailed in chapter 4.7.5 on page 187. Thus, this interface can be used for debugging purposes. However, when it is used for this purpose then it should be prevented that multiple threads write at the same time to avoid that instructions are discarded.

The ATU RF provides counters for successfully submitted instructions and discarded parts. These counters can be used to evaluate the probability mechanism and ensure that all instructions were submitted if the interface is used for debugging purposes.

4.6 Design

In the prior section the interfaces of the ATU2 were described. These interfaces and their properties largely influence the design, because they impose certain requirements due to the load that they can cause.

An important desired improvement as compared to ATU1 is the avoidance of unnecessary reads to the main memory.

There are mainly two possible ways to handle the following requests for the same NLPA. Either it is possible to not care about these in a special way and do more memory reads or the better way, which was chosen for ATU2, is to avoid the extra reads. A trivial solution for the problem would be to stall the ATU2, after every request, until the read response arrives, but this would, considering the read latency, not be a good solution, because in that time no other translation request could be handled.

Taking the avoidance of unnecessary reads into account early in the development is a central element of the design. This feature is complex, because different information has to be stored and managed. Two things have to be considered. First, the information that a translation is under way has to be stored, because otherwise a read can not be avoided. Second, following translation requests, which are waiting for this translation to finish, have to be stored in a data structure. Subsequently, the translation responses can be generated as soon as the required information for the translation arrives in the ATU2.

The only sensible place to store the information about outstanding GAT reads is the TLB. Therefore, a **busy** bit was added to the TLB, which indicates that a GAT read is under way for the particular NLPA.

Another data structure would be resource intensive, because it would require a data structure with 32 entries for NLPAs with main memory reads under way. This would have to be implemented in a CAM, which would have to be able to do one lookup per cycle, and furthermore it should be able to do the lookup quickly in one or two cycles. In the end 32 registers and comparators with 28 bit each would be required, due to the lack of special CAM hardware structures. Thus, the TLB is the right place for the information in terms of resources and development effort.

Accordingly, when a translation request arrives in the ATU2 and it is a miss, it is tried to allocate a TLB entry by possibly evicting an old translation for this NLPA. The new entry will carry the NLPA and the busy bit, and the memory read request will be sent.

If there is no TLB entry available then unnecessary reads can not be avoided. However, this should happen only very seldom at least when the NLPA distribution is, regarding the TLB index, not very unfortunate.

Storing Outstanding Requests

From a high level view ATU2 works as depicted in figure 4-31, the requests arrive in the pipeline and with the TLB index the entry is read out, then based of this entry either a response is generated or if a GAT read has to be done, then the TLB entry is marked busy. In the later case this is a read-modify-write operation.

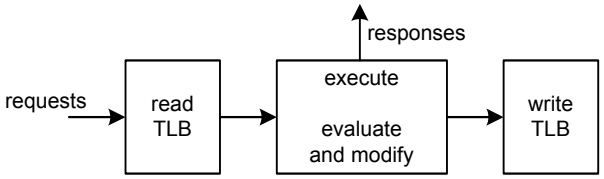


Figure 4-31: TLB Read-Modify-Write

As soon as another translation request arrives a TLB lookup will be done and when it is a hit for a translation that has the busy bit set, then it is not necessary to perform another GAT read. However, it is not sufficient to only store the information about the outstanding requests. It is also necessary to evaluate the information again as soon as the GAT read is completed and the necessary information to generate the translation response is available. Such a mechanism can implemented with the following methods:

- reservation station
- data structures in TLB entries
- replay FIFO

CPUs are using **reservation stations** [132] when an instruction waits for operands, the same principle could be applied to the ATU2, but given up to 32 outstanding translations it would be necessary to implement 31 reservation stations, which can wait with translation requests for read responses from the GAT. Additionally, when multiple reservation stations complete in the same cycle arbitration is required, because obviously it is not possible to send back multiple translations responses at the same time. Reservation stations do not fit the problem and would require too many resources.

As explained before the busy bit in the TLB indicates that a GAT read is in progress, therefore it would be possible to store the outstanding requests **in the TLB**. It was already mentioned that each translation request is stored in a context RAM. Therefore it is not necessary to store the complete requests within the TLB entries, because it is sufficient to store pointers to the context RAM.

There are two fundamental possibilities to do this, either with a bit field with one bit for each of the 32 possible entries in the context RAM or with a more complex data structure. The TLB entry could point to one entry within the context RAM and within the context RAM a linked list could be implemented.

So either storage space can be wasted or complexity has to be accepted.

However, there is also another possibility. When a translation has to wait because the TLB entry is busy the whole translation request could be stored in a **replay FIFO** and issued again. This is shown in figure 4-32. Consequently, requests will rotate through the replay FIFO as long as the TLB entry is busy. As soon as the TLB entry is updated by the read response from the GAT it is not busy anymore and the request is handled.

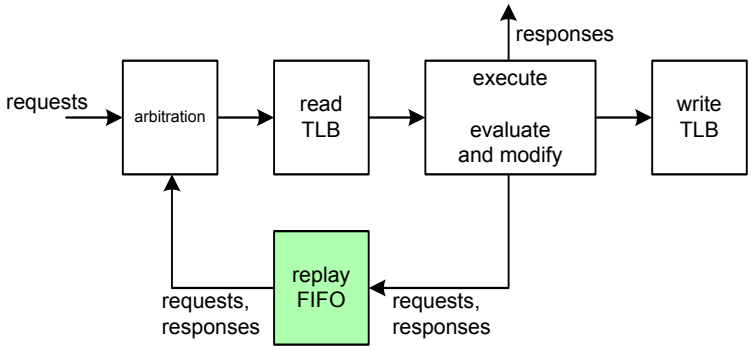


Figure 4-32: Replay FIFO

The replay FIFO additionally allows a very simple implementation of the reordering. As explained before the requesting units are expecting to receive the translation responses in the same order as the translation requests. They are using the source tags in ascending order, and therefore also the responses have to be in the same order.

However, ordering could be lost when there are a request A and B directly behind each other, but A is a TLB miss and B is a hit, then ATU2 has to delay the response for B until the it was possible to send the response for A.

To avoid complex extra hardware structures the concept of the replay FIFO was extended to also store responses. Thus, if a response would be out of order, then the execute unit can store the response in the replay FIFO for resubmission. Responses will just rotate through the pipeline until ordering is fulfilled again.

4.6.1 TLB

The TLB must contain all information that is necessary to translate requests without reading from the GAT. Therefore it stores the following information:

- NLA
- PA
- size
- VPID

Furthermore it is necessary that each entry contains the following status information:

- valid
- read / write bits
- no-eviction
- busy

It may happen, that at some point in time all TLB entries for a specific TLB index are occupied, then entries have to be replaced. Thus, it is necessary to choose a suitable replacement strategy.

The design space as depicted in figure 4-33 contains the replacement strategies that were considered for ATU2.

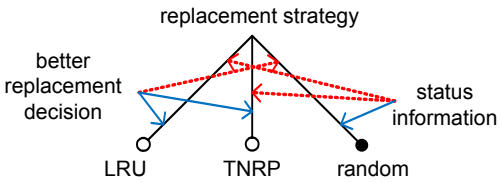


Figure 4-33: Design Space: Replacement Strategy

Least Recently Used (**LRU**) replaces the least used entry first.

Time of Next Reference Predictor (**TNRP**) is an example for a complex replacement algorithm and represented in [76]. The algorithm tries to predict, which page will be used next. Therefore, it stores the time of the last access and the number of page accesses between the last and the second last access.

The **random** replacement uses a fair round robin arbiter [114]. However, the replacement is “random” and not round robin because there is no round robin status for each possible TLB index, but the arbiter is running continuously. Thus, it ensures that no particular TLB set is replaced more often, because of the fairness property.

LRU and TNRP, both require significant amounts of status information and resources to evaluate this information. These algorithms are designed for the replacement of pages in operating systems or cache lines in CPU caches. The managed resources are in relation to the required extra status information much bigger than in the case of the ATU2 TLB. A page has 4096 bytes and a CPU cache line has 64 bytes, but an entry in the ATU2 TLB requires less than 80 bits.

In the case of TNRP a time and a counter would have to be stored, it is assumed to require 40 bit (32 bit for a timestamp and an eight bit access counter) of space and therefore it would add 50% storage overhead to the TLB and additionally it would add logic. LRU could be implemented with lower overhead.

Using TNRP would increase the TLB size by 50%. In summary, a 4-way set associative TLB with TNRP would require the same amount of storage like a 6-way (or 4-way with 50% more entries) TLB with random strategy.

If more resources should be allocated for the TLB, it seems to be feasible to invest them to increase the number of entries instead of using them to store status information of complex replacement strategies. Extra logic resources could be used to support more different NLP sizes to reduce the number of TLB entries that are required.

Therefore, the decision was in the end to use the random strategy.

It should be noted, that there is a big difference to a virtual memory subsystem or a CPU cache. The software, which manages the ATU2, is allocating the NLPAs. Therefore it can try to allocate NLPAs in a way to optimize TLB usage.

TLB Index Size

For the implementation of the ATU2 for the Ventoux board, which is the primary development platform, a decision had to be made regarding the size of a single TLB set. The amount of resources that are required inside the FPGA depends, due to the properties of the BRAMs, on how these hardware primitives can be used to implement the required amount of storage.

The 36 kbit BRAMs [9] in the Virtex 6 FPGAs have a maximum width of 72 bit if no ECC correction is used and are 512 entries deep. Other possible configurations are for instance:

- 36 bit * 1024
- 18 bit * 2048
- 9 bit * 4096

It is also possible for the FPGA tools to split the 36 kbit BRAMs into two 18 kbit BRAMs with half the width. Table 4-2 shows the required resources for different TLB index widths.

Index width	TLB width	amount of data in bits	BRAMs required	usage
8	78	19968	1 x 36 kbit + 1 x 18 kbit	36.1%
9	77	39424	1 x 36 kbit + 1 x 18 kbit	71.3%
10	76	77824	2 x 36 kbit + 1 x 18 kbit	84.4%
11	75	153600	4 x 36 kbit + 1 x 18 kbit	92.6%
12	74	303104	8 x 36 kbit + 1 x 18 kbit	96.7%

Table 4-2: Size Consideration for a Single TLB Set

In this table it can be seen that the TLB width depends on the index width, because the TLB entries only store the part of the NLPA that is not implicitly stored by the address of the TLB entry. The most interesting number is the usage. Obviously, it is very bad in the case of the 8 bit index width, because each BRAM has a depth of at least 512 entries, so half of the capacity is already wasted by this. Thus, an index width of 8 is out of question.

In the end the decision was to make use of a 4-way set associative TLB with an index width of 9 bits. The reason for this is that it is expected that a 4-way set associative TLB will allow a high hit rate and an index width of 9 bits requires the lowest amount of resources.

The TLB width is parameterized, therefore it can be easily changed as soon as the EXTOLL design is in a state that benchmark results can be generated for different TLB configurations on a FPGA test cluster.

4.7 Micro Architecture and Implementation

The most important functionality of the ATU2 is the generation of translation responses for requests from the RMA. In anticipation of a high hit rate the pipeline is designed around and optimized for the common path to deliver the lowest possible latency and a high throughput. Figure 4-34 shows the most likely path that is taken when there is a hit for a request from the RMA. Less than 5 pipeline stages are not possible at the targeted 300 MHz on a Xilinx Virtex 6 FPGA.

The task of the HT inport is to arbitrate between the different ways data can arrive in the ATU2. The RAM access takes two cycles, because the Xilinx SRAM blocks of the Virtex 6 have a high clock to output time of 1.79 ns [119]. The second RAM read cycle consists only of a register stage to relax the timing.

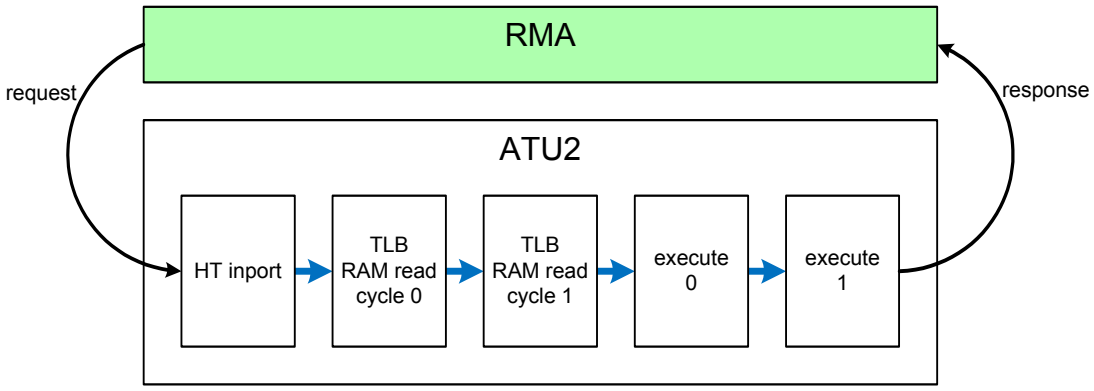


Figure 4-34: Optimized Five Cycle RMA/ATU2 Translation Path

However, a hit is only the most likely case. When a memory read request is made to read from the GAT, and there is a TLB entry available then it is marked busy and the context RAM stores the information which TLB RAM has to be updated. The busy information is used to prevent unnecessary GAT reads.

Searching a TLB entry and marking it busy is a read-modify-write operation of the TLB, and storing the information from the GAT read is a write operation.

A memory read can take quite some time, depending on the system interface and the hardware in use. A range from 100 ns (estimation for a high performance ASIC implementation connected to HyperTransport 3) to 600 ns (FPGA design connected to PCI express) is realistic for current technology. The read latency was measured in an AMD Opteron server system with HTX adapter board, which was using HT600 (HyperTransport at 600 MHz), and the result was 250 ns.

As soon as the read request to the GAT returns with a response and if the context RAM indicates that the TLB entry has to be updated, then the information from the GAT is stored in the TLB.

The Flush VPID command is also a read-modify-write operation, because it has to read out the TLB and overwrite the entries with a specific VPID.

Therefore it makes sense to have all data flow in a single execution pipeline, because otherwise it would be necessary to arbitrate the different accesses to the TLB RAM.

Consequently a complete ATU2 translation with a TLB miss should be described in the following with the help of figure 4-35, which outlines ATU2 and the important modules and their connections.

1. A translation request from the RMA arrives and the HT inport converts it to the

internal pipeline format

2. The TLB RAMs are read out for comparison with the NLPA in the translation request
3. It is a TLB miss and execute1 initiates a GAT read by sending the corresponding command to the HT outport; the information from the request is stored in the context RAM; if execute0 finds an available TLB entry, then a TLB entry is reserved for this NLPA and marked busy
4. The HT outport arbitrates the HTAX crossbar and sends the GAT read
5. In the HT inport the read response is converted into the internal pipeline format and forwarded to the pipeline
6. Based on the source tag of the read response the context RAM is read out
7. Execute1 receives the context information and the information from the GAT and evaluates the information
8. Based on the received information the context1 unit sends a translation response to the RMA and stores the translation information in the TLB

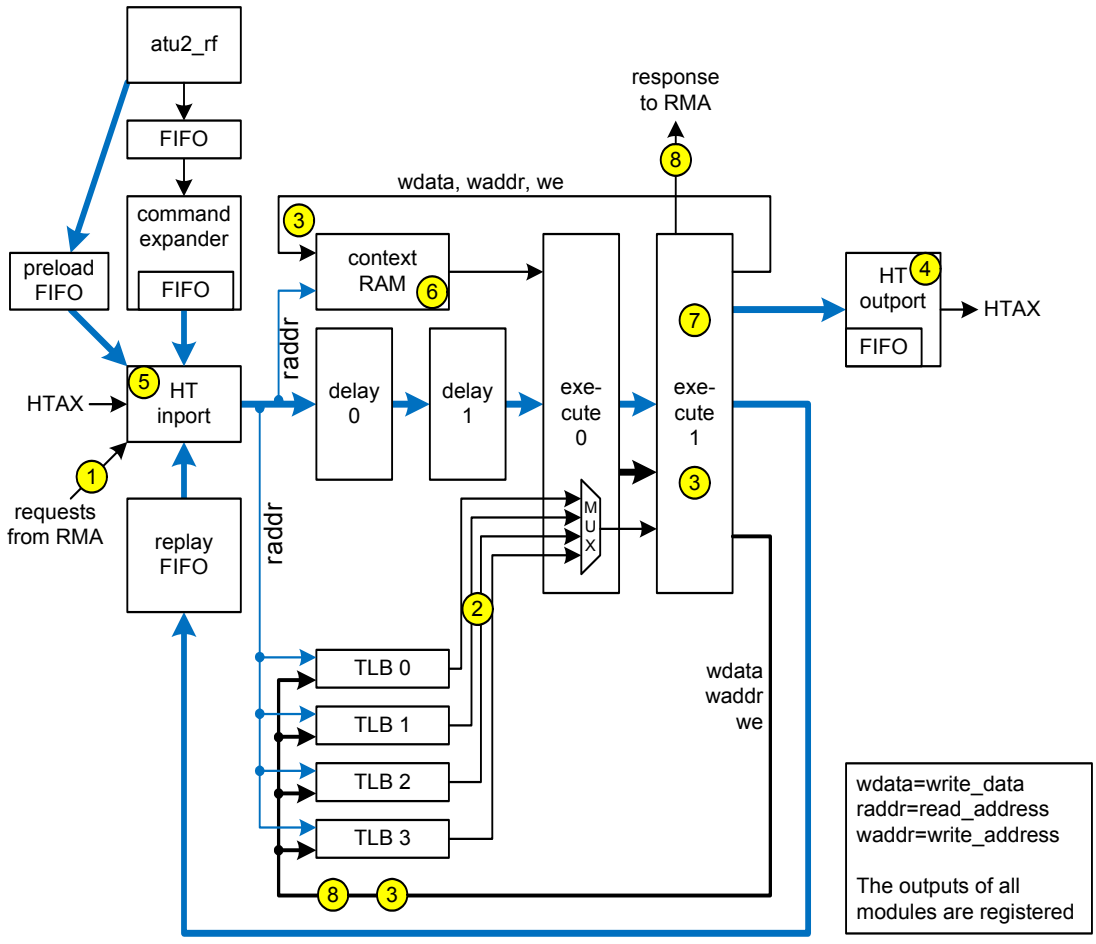


Figure 4-35: ATU2 Implementation Details

The instruction pipeline contains the delay stages to make up for the read latency of the TLB RAMs.

Such a pipeline implementation of course has different issues that have to be solved, namely the following ones:

- getting the different inputs into the pipeline
- preventing dependency problems
- flow control

These issues are addressed in the following two subsections.

4.7.1 Inputs and HT Inport

The HT inport can be seen as a five way multiplexer. It has five different data sources and schedules them for the execution pipeline. Additionally, it accepts the incoming data from the HTAX crossbar. For the different inputs there are different priority requirements due to the different interfaces.

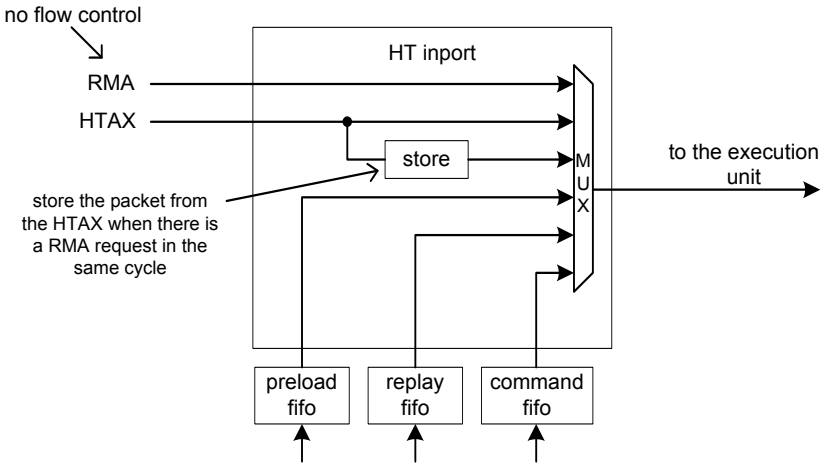


Figure 4-36: HT Inport Data Flow

The HT inport exists in a 64 and a 128 bit version to support both widths for different FPGA implementations and the ASIC.

There are, as depicted in figure 4-36, five sources of inputs in the HT inport:

- HTAX
- RMA interface
- command FIFO
- replay FIFO
- preload FIFO

The **HTAX** interface is equipped with a request/grant mechanism and therefore it provides flow control, but it should be avoided to stop granting, because stalling this interface can also have effects on other units. In the worst case a new packet can arrive from the HTAX every three cycles.

However, the difficulty in this unit is founded in the fact that the input from the **RMA** unit provides no flow control. Every request from the RMA has to be accepted and none can be discarded or delayed. A FIFO would solve the problem, but it would require resources.

The remaining three inputs from the **command**, **replay** and **preload** FIFO have of course flow control because they are FIFOs.

Translation requests have a higher priority than commands, because for example flushes are less timing critical. In the case of translation requests from the RMA it is even imperative, because there is no flow control.

Consequently, all new translation requests from the RMA are forwarded directly.

HTAX packets have the second highest priority, to not stall the HTAX crossbar. Therefore, the HT inport is able to store a single packet from the HTAX. The packet store can hold the content of a single HToC packet; it is a single register that stores one 95 bit wide pipeline instruction. The HTAX crossbar is using a request/grant scheme and the ATU2 is always granting, despite when there is already a packet in the store and at the same time there is a new translation request from RMA.

Command, replay and preload are scheduled fair when there is no new RMA request or input from the HTAX. It is not necessary to use more elaborate mechanisms to give for example the replay FIFO a higher priority, because in a realistic usage scenario there are not so many preloads or commands.

Preloads can only be sent by the OS kernel and in the worst case only one every ten cycles, because two writes to the RF take at least ten cycles. Furthermore, they will most likely be sent only when new memory is registered. A similar argument applies to the command interface, because it is used for flushing entries and fences, and the number of these is also not unlimited. ATU2 can handle in the worst case one instruction every five cycles, worst case dependencies are outlined in the next subsection. In reality a performance close to one instruction per cycle is expected. At least under assumption of “useful” usage scenarios, this can not be the bottleneck, because either the operation throughput of the RMA or the interface from and to the host system will be the bottleneck instead.

4.7.2 Dependencies and Flow Control

As explained before, the TLB is used in a read-modify-write flow within a pipeline, in each cycle a read and a write operation can take place. Such a setup can cause consistency problems, because reading the TLB takes two cycles and the execution stages also require two cycles. When there are two instructions A and B behind each other and both instructions read, modify and write the same TLB entry, then A and B will have the same data when reading. However, if both are also writing to the same TLB entry, then the write caused by A will not matter, because it will be overwritten a cycle later by B.

Thus, it has to be ensured for every pipeline instruction that the same TLB index was not used in the last four cycles.

This problem is avoided with two methods:

- arbitration
- no exec flag

The pipeline format, which will be detailed in chapter 4.7.5, has the TLB index always at the same position. Therefore, it is possible to store and compare the TLB index of the last four cycles without using a lot of resources to differentiate between the different possible instructions.

Figure 4-37 shows a reduced schematic with the parts relevant for this explanation.

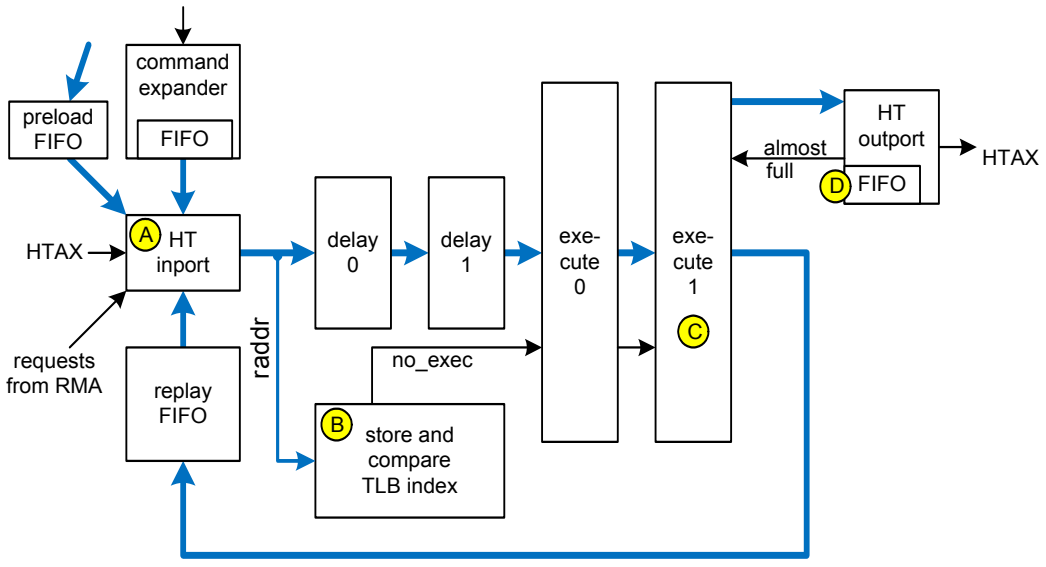


Figure 4-37: Dependency and Flow Control

The HT inport **(A)**, arbitrates between five different inputs, three of these are FIFOs and the HT inport will only take instructions from any of these FIFOs when the TLB index part is different from the one of the last four cycles.

Thus, there will never be an instruction from these FIFOs in the pipeline that may have a dependency problem.

However, the RMA request and the input from the HTAX are not out of a FIFO and there is at least in the RMA case no flow control. The obvious solution would have been to add two FIFOs to the HT inport that would store the instruction if there is a possible dependency problem, but this would have a resource impact due to the FIFOs, therefore another approach was chosen.

Extra logic **(B)** was added to the pipeline that does the same comparison as it is done in the HT inport for the three FIFOs and if there is a dependency problem then a *no_exec* signal is raised. The execute unit **(C)** will in this case simple do nothing with the instruction and forward it to the replay buffer.

The advantage of this approach is that no extra FIFOs are required and furthermore the pipeline also does not have to be stalled.

As a result the replay FIFO allows to avoid adding two FIFOs to the HT inport because in the case of a dependency problem the instruction can simple be forwarded to the replay FIFO.

In the worst case all instructions are for the same TLB index, then only every five cycles an instruction will be executed.

Flow Control

FIFOs require resources. Therefore, it was tried to keep the FIFOs as small as possible without causing problems.

The FIFO **(D)** in the HT outport, as depicted in figure 4-37, is only eight entries deep, because also in this case the replay FIFO can be used, as soon as the FIFO is *almost_full* a stop signal will be asserted. Based on this signal the execute unit will store the commands into the replay FIFO instead. This is no issue, because the execute unit will simple try to submit them again to the HT outport a few cycles later.

Of course also the size of the replay FIFO had to be chosen correctly, this was done by taking the worst case into account. In the worst case there can be 32 instructions circulating in the pipeline. Thus, the replay FIFO has to have a size of at least 32 entries.

4.7.3 Command Expander

The task of the command expander is to convert the commands, which are submitted by the software, into the internal pipeline format so that they can be executed.

The format of the commands is defined in chapter 4.5.8, depending on the command and its operands between one and 1024 pipeline instructions are generated for each command. Each generated instruction is put into a small FIFO and the *almost_full* signal of this FIFO is used in the state machine that generates the instructions.

Furthermore, this unit is also taking care that there is always only a single fence and the corresponding notification active. Otherwise the replay FIFO could overflow, because a fence notification has to wait and rotate until the fence was completed. Therefore, the command expander stops submitting instructions to the execution pipeline when there is already another fence or notification in the pipeline.

The reason for this is flow control, because otherwise many more notifications could otherwise be forwarded to the execution pipeline than can be handled in the HT outport, and then they would have to be stored to the replay FIFO. However, the size of the replay FIFO is limited.

Additionally, this unit is also responsible for resetting the TLB and the context RAM after a cold reset. It does this by generating pipeline instructions.

4.7.4 Execute

The task of the execute unit is to handle the pipeline instructions. This unit consists of two pipeline stages for timing reasons as depicted in figure 4-38.

In **execute0** the evaluation is made if a request is a hit and a MUX selects the output from the TLB RAM that is a hit, this reduces the logic complexity in **execute1**. For each TLB RAM there is a signal to indicate the following properties:

- NLPA matches
- VPID matches
- VPID and size matches

Furthermore, a TLB RAM for replacement is chosen, if there is one available. This is not always the case, because TLB entries with no-eviction or busy bit set can not be replaced. This information is transported with a one-hot bus named ***tlb_writable***. When the execute1 unit needs to write to a TLB it can assign this bus directly to the *write_enable* bus of the TLB RAMs.

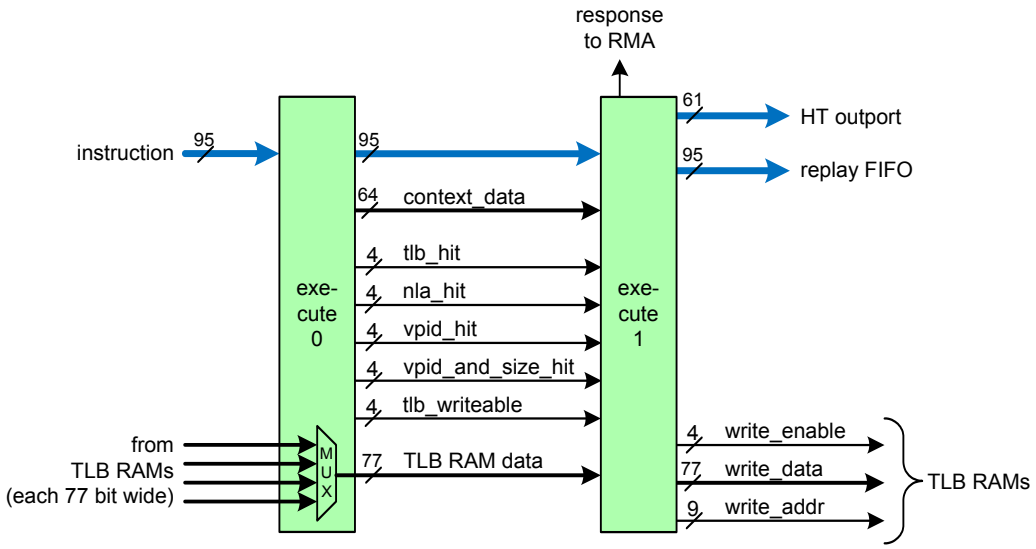


Figure 4-38: Execute Unit

Most instructions are handled in the execute1 unit. Here also the responses and GAT reads are constructed and forwarded to the respective receivers.

4.7.5 Pipeline Instruction Format

The ATU2 pipeline is using an instruction format that is consistent in all pipeline stages and additionally towards the HT outport. The format, as shown in figure 4-39, is using four bits to distinguish between the different instructions, and the operands of the instruction are stored in the rest of the instruction word and dependent on the respective instruction.

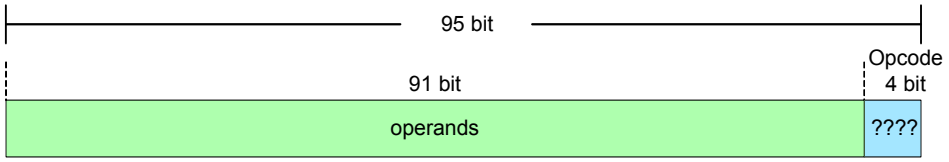


Figure 4-39: Generic Instruction Format

The maximum width of an instruction is 95 bit, but the actual data paths between the modules are only as wide as necessary to save resources. The paths from the command expander (53 bits) and to the HT outport (61 bits) are not as wide to save resources. All other instruction paths as depicted in figure 4-40 are 95 bit wide. Most instructions have a regular format. This is not only helpful for the development of the Verilog HDL, but it also saves resources, because the hardware structures are more regular and therefore the synthesis has more optimization opportunities.

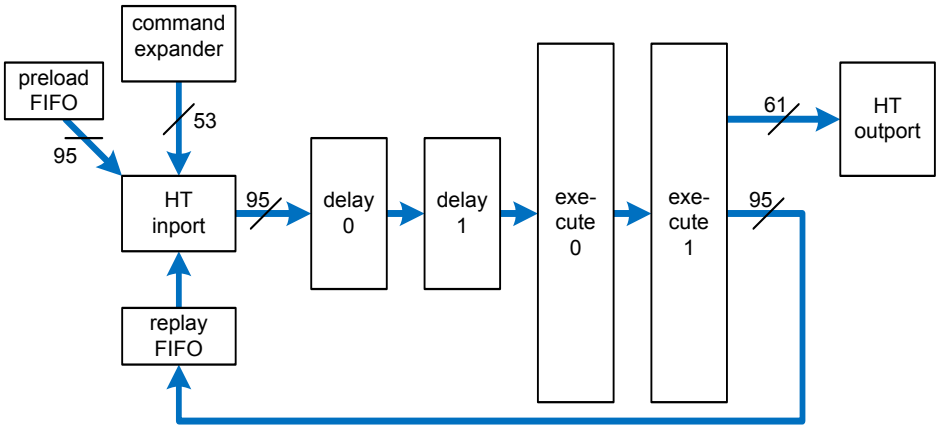


Figure 4-40: Instruction Format Using Units

In the following the different instructions and their operands will be shown. All of them have the NLPA at the same position. Consequently, also the respective portion of the NLPA that constitutes the TLB index is at the same position. The TLB index is crucial to prevent read-modify-write dependency problems, if not all instructions would have it at the same position, either as TLB index or as part of the NLPA, then the dependency analysis would be much more complex.

Flush NLPA

With the help of the Flush NLPA command in figure 4-41 a specific NLPA entry can be removed from the TLB.

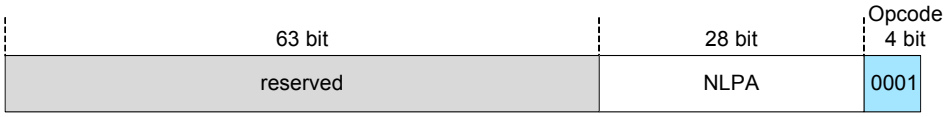


Figure 4-41: Flush NLPA Instruction

Flush VPID

When not only single NLPAs, but all entries for a specific VPID have to be removed from the TLB, then the instruction in figure 4-42 is the right one. There can be multiple entries for a VPID at the TLB index position, as given in the instruction, and this command will invalidate all of them.

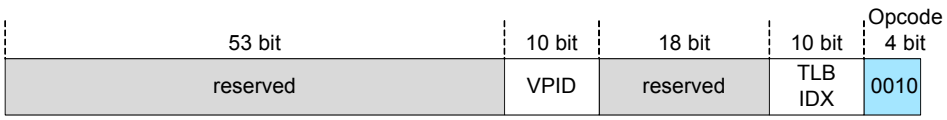


Figure 4-42: Flush VPID Instruction

The architecture as depicted in figure 4-38 with hit signals for different cases allows implementing this instruction in execute1 by assigning the *vpid_hit* signal to the *write_enable* of the TLB RAMs while assigning zero to *write_data*.

Flush IDX

The instruction, which is depicted in figure 4-43, can be used to delete all entries from the TLB that are at the index position in the operand. It is used for complete TLB flushes. This instruction is used after the cold reset to initialize the TLB RAMs.

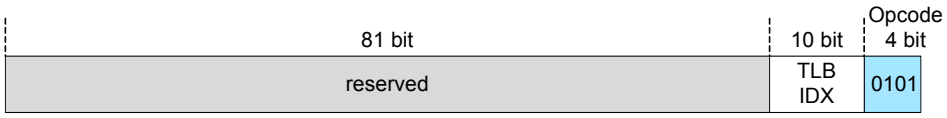


Figure 4-43: Flush IDX Instruction

Fence

Sends a fence request over the HTAX, the instruction in figure 4-44 is used when the excellerate interface is activated. It is generated by the command expander. The execute unit itself does not do anything with it besides forwarding it to the HT outport.

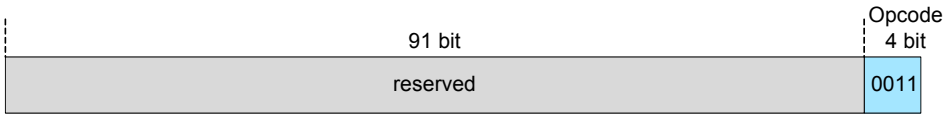


Figure 4-44: Fence Instruction

Notification

The notification causes a write to the PA location given in the instruction as shown in figure 4-45, when it is handled by the HT outport. However, it will be only forwarded by execute1 to the HT outport if there is no outstanding fence.

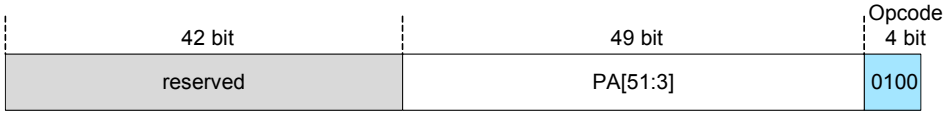


Figure 4-45: Notification Instruction

The execute unit will store it in the replay FIFO when there is an outstanding fence.

Address Translation Service Request

The HT inport encodes incoming translation requests from the RMA or HTAX into such an instruction as depicted in figure 4-46. A flow diagram for the execution of this instruction can be found in figure 4-52.

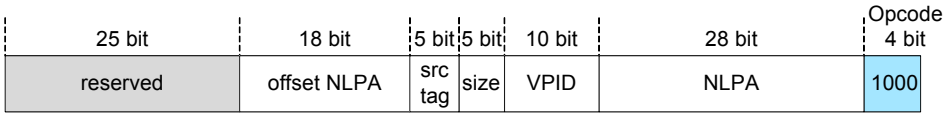


Figure 4-46: ATS Request Instruction

Main Memory Read

When it is necessary to perform a main memory read to obtain GAT entries then the execute1 unit will issue such an instruction to the HT outport, or, if the FIFO of the HT outport is full then it will be forwarded to the replay FIFO instead. Figure 4-47 depicts the necessary operands.

Figure 4-47: Memory Read Instruction

Read Response from Main Memory

The HT inport is receiving the read responses from the main memory and forms an instruction from this to inject it into the pipeline as depicted in figure 4-48.

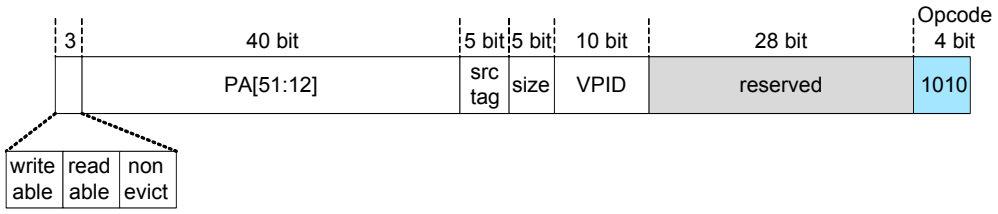


Figure 4-48: Read Response Instruction

Based on the information from the main memory the TLB can be updated and translation responses can be generated.

Address Translation Service Response

When the execute units was able to translate a request, then it emits a reply as depicted in figure 4-49.

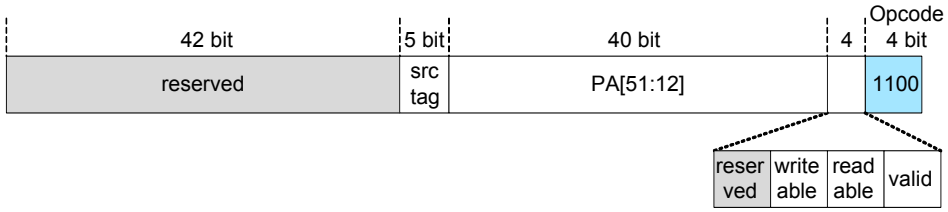


Figure 4-49: ATS Response Instruction

Preload Instruction

As described before it is one of the ATU2 features that the software is able to preload entries in the TLB. The instruction in figure 4-50 will try to store the given translation information, into the TLB to avoid the read latency for the GAT access.

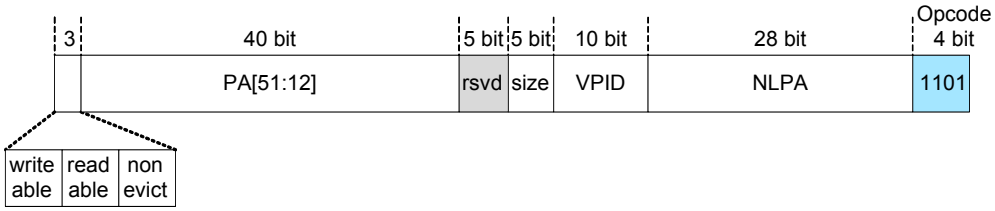


Figure 4-50: Preload Instruction

However, if there are no usable TLB entries, then the information is discarded. This can happen when all entries in the particular index have either the busy or the no-eviction bit set.

Context Flush

This instruction as shown in figure 4-51 is used after the reset to initialize the context RAM, so that it contains a valid state. Under normal circumstances it should not be required, but it could be used to initialize the context RAM without hardware cold reset.

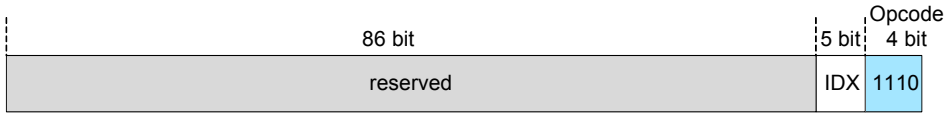


Figure 4-51: Flush Context Instruction

The context RAM has 32 entries and depending on the IDX the corresponding entry is overwritten with zeros.

Thus, 32 of these instructions with all possible IDX values have to be inserted into the execution pipeline to clear the context RAM completely.

Execution of Commands

Handling of requests is the central tasks of the ATU2, therefore for this particular instruction a flow diagram is presented in figure 4-52.

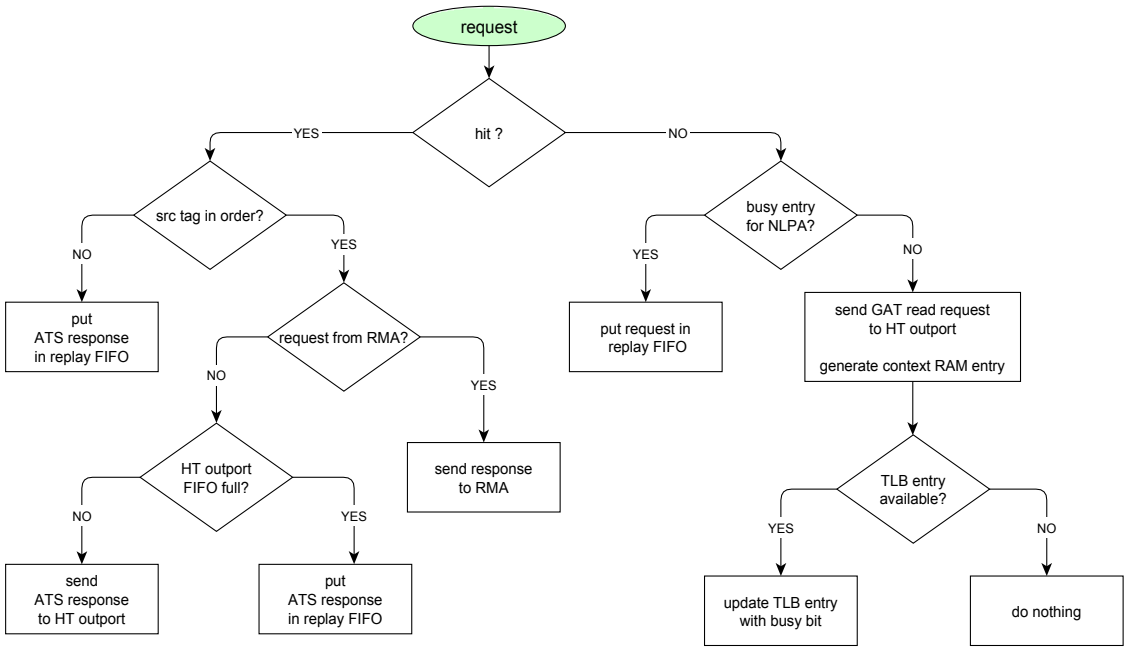


Figure 4-52: Request Flow Diagram

As it can be seen in this diagram, the execution of this instruction itself is causing, in some cases, the generation of new instructions.

4.7.6 HT Outport

The HT outport is responsible for sending outgoing packets over the HTAX crossbar. It receives instructions from the execute1 unit and converts them into packets in the HToC protocol to send them over the HTAX either to the host system or the EXU.

The design was done with the lowest possible latency in mind. Therefore it has a FIFO and non-FIFO path. When the module is idle, then the non-FIFO path is used. The module is idle when the interface to the HTAX is not used.

Thus, the latency is only a single clock cycle when the unit is idle, and in all other cases the instruction is stored in the FIFO. The schematic representation in figure 4-53 shows the two paths a command can use. In this representation it looks like each instruction would be stored in the FIFO, but this is of course not the case, the instruction is only stored if the interface to the HTAX is already in use.

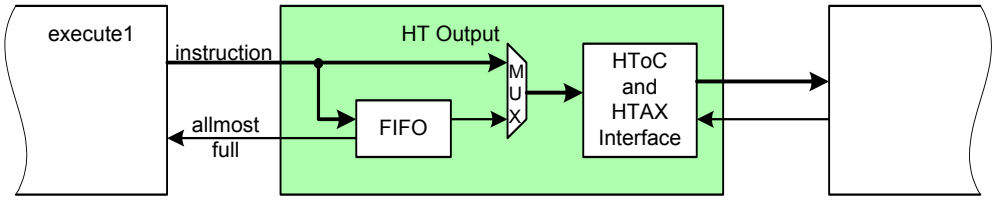


Figure 4-53: HT Outport

As depicted in figure 4-53 the execute1 unit is sending commands to the HT outport, back pressure flow control is implemented by using the *almost_full* signal from the FIFO. The threshold is chosen so that the execute1 unit has enough time to stop sending new commands.

The HT outport supports the following commands in the pipeline format, as described before:

- fence
- notification
- main memory read
- ATSResp (Address Translation Service Response)

4.8 Software Interface

To control the ATU2 and to be able to read out performance characteristics several registers are made available by a RF interface.

The setup of ATU2 for operation is done with the RF interface, before the ATU2 is able to perform translation the software has to provide GASTs and write the GAST lookup RAM with the RF interface.

4.8.1 Statistics Counters

HPC networks like EXTOLL are very complex designs. Therefore, it is necessary to have methods to analyze different performance properties, because these can help to optimize the hardware and software to reach higher performance. The ATU2 RF provides counters and values to observe and analyze the address translation.

In particular the following values are provided:

- number of translation requests
- number of memory reads

- for the preload interface the number of successfully received commands and the number of discarded quadwords is counted
- number of commands received overall
- number of NLPA flush commands
- number of fence commands
- number of notification commands
- number of flush VPID commands
- number of flush IDX commands

Furthermore fill level indicators and a register that stores the maximum fill level were added for the replay and the preload FIFO to allow an analysis of their usage patterns.

4.9 Results

To evaluate the performance properties of ATU2 certain values about the implementation were collected and set in perspective. The design is overall implemented in about 4000 lines of Verilog HDL including the 64 bit and 128 bit versions of the HT inport and outport, but without the automatically generated ATU2 RF.

4.9.1 Latency

ATU2 was designed from ground up to reduce the latency compared to ATU1. In the following table multiple basic performance characteristics are given. The numbers for ATU1 are from [6].

Path		ATU2 (in cycles)	ATU1 (in cycles)
TLB Hit	ATS request from RMA to response	5	9
TLB Miss	ATS request to read	6	11
	read response to ATS response	7	9
	complete	13 + main memory access	20 + main memory access

Table 4-3: ATU2 Performance Characteristics

The latency for a complete translation with ATU2 was measured in system and amounts to 63 cycles at 200 MHz or 315 ns. Therefore, the main memory access accounts for 250 ns, because 13 of these 63 cycles are spent in the ATU2.

4.9.2 Resource Requirements

Xilinx Virtex 6 is the main development platform as explained earlier, therefore the resource requirements are given for this architecture in table 4-4. An index size of 9 bit was used and a set associativity of four.

resource	amount
FFs (flip-flops)	3321
LUTs	3536
RAMB36	9
RAMB18	5

Table 4-4: ATU2 Resource Requirements

The design reached a maximum frequency after place and route of 304 MHz.

Trading Architecture and Simulation Environment

Chapter 5

5.1 Introduction

This chapter describes a new approach taken to develop a system for very fast and efficient trading at stock exchanges. For this application a low latency TCP/IP stack has been implemented in an FPGA to lower the latency. Special attention is given to the problem of testing the system and methods that allow accomplishing this task efficiently, despite of the tremendous complexity of the system.

The process of building a bit file for the FPGA from the HDL sources is time consuming and possibilities to debug the design in hardware are only very limited. Thus, hardware designs should be tested as much as possible in simulation before they are is tested in an FPGA. In the case of an ASIC it is of course out of question to produce a chip that is not sufficiently tested in simulation.

Testing is a hard problem for two reasons. To test a complex hardware design also a complex testing environment is required, this requires a lot of development effort. The second problem is the amount and type of testing that is required to reach a state that is "good enough" for the application. "Good enough" in the sense that the risk of a bug is either small enough or a failure has no catastrophically effect.

In the following chapter a short overview is given about the testing of hardware designs, a TCP implementation is explained and the approaches that were used to lift its quality to a production ready state with simulation.

5.1.1 High Frequency Trading

For this application a hardware/software system has been developed that allows reducing the latency of the trading as far as possible.

The target of the development is to reduce the overall latency of the time the trading server requires from the arrival of an Ethernet packet, with a market update, until the order is on the cable. Therefore, optimizations are possible on all levels of the network stack. The different processing steps in the trading server are depicted in figure 5-1.

In [98] and [99] we have shown methods to lower the latency of the reception of **market updates** and the decoding of FAST messages with the help of FPGAs. The **FAST** protocol [28] is an encoding format that is used to compress market update information. Promising results have motivated further research into the topic.

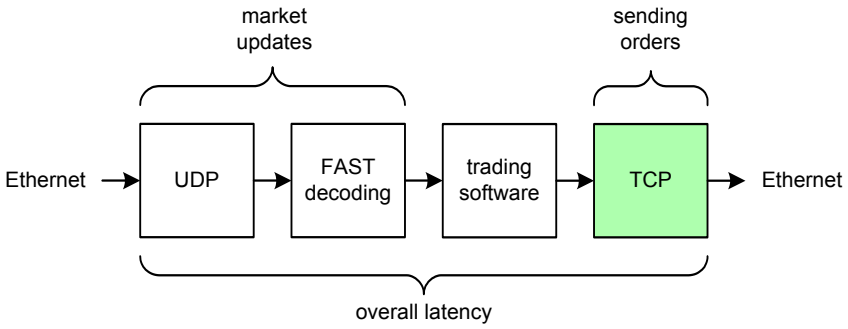


Figure 5-1: Processing Stages in the Trading Server

Optimizing the trading software has been out of scope. Therefore, the sending of orders should be improved with a TCP hardware unit.

5.1.2 Testing and Simulation Methods

Hardware designs have to be tested before it makes sense to use them. Test benches are a "classical" method of testing. These consist in some cases only of a single Verilog module that feeds a stimulus to the hardware that is to be tested, which is called Design Under Test (DUT), and checks the outputs. Figure 5-2 shows such a setup.

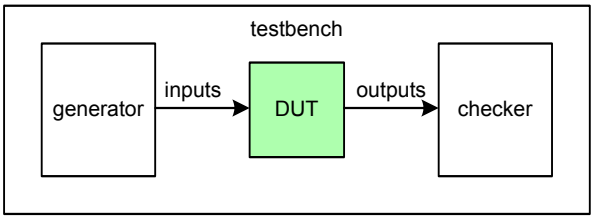


Figure 5-2: Generic Test Bench

Such a "classical" test bench can work very well for small units that do not have a lot of internal state.

A slight variation is the comparison with a reference model. Small arithmetic units are examples where this approach can work very well, if the input vector is small enough to test all combinations. For example, a special divider with fixed divisor [27] was tested with this method. Writing the test for all possible values that compares the results with a reference

model as shown in figure 5-3 took only few hours. The exhausting test had a runtime of one night on a desktop system. Verilator [157], which synthesizes Verilog to C++, was used for this.

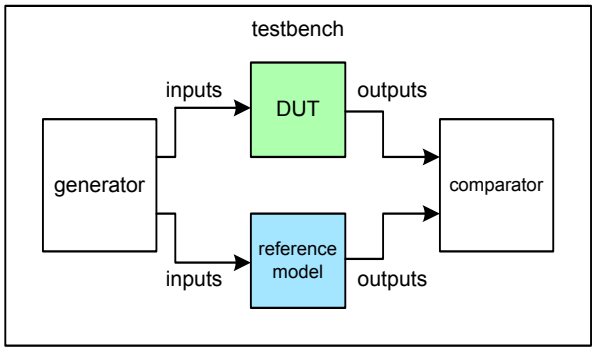


Figure 5-3: Test Bench with Reference Model

However, complex DUTs with complex interfaces and lots of internal state are very complex to test with simple test benches.

Therefore, verification environments are developed for complex designs. The idea behind these is to abstract the actual stimulus of the DUT and output of the DUT with a transaction model from the development of the tests.

The current “state of art” is the Universal Verification Methodology (UVM) [147]. It tries to motivate a development model that separates the verification environment in different reusable components called Universal Verification Components (UVC).

In figure 5-4 a verification environment with two UVCs is shown. The virtual sequencer drives the UVCs. These are translating the transactions into stimulus for the DUT and monitor the interface from the DUT. The scoreboard will be informed about both directions with transactions. Thus, the scoreboard can contain an abstract high level model of the DUT and evaluate with the transactions if the behavior is correct.

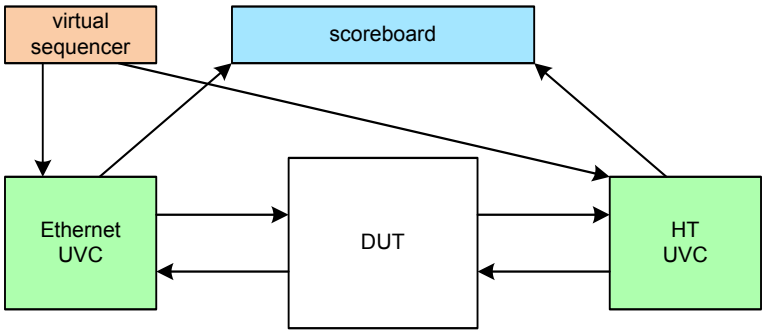


Figure 5-4: Verification Environment

Verification environments are usually combined with methods to collect coverage information. Coverage can be based either on source lines or on features points in the DUT.

5.2 TCP

TCP [5] usually is used synonymously with TCP/IP, because TCP relies usually on the IP protocol. It is part of the protocol stack, which is used in almost every device from cellular phone to supercomputer. Thus, a further introduction of the TCP protocol would be a superfluous exercise.

The implementation of TCP in hardware is interesting because it allows reducing the latency. It will be built application specific and therefore it is possible to accept certain limitations in comparison to a general implementation of an OS kernel like Linux [150]. One of the reasons why offloading TCP makes sense is that it allows reducing copy operations in the main memory for packet processing as stated in [125].

In [124] an analysis of the latencies for sending and receiving of TCP can be found. The times in the breakdown are for a 2.13 GHz processor and a 1G Ethernet card. Only minimum sized packets are transmitted, and therefore the difference to more recent CPUs and 10G Ethernet cards should be small. In the end, this setup required 4159 ns to get the data from the application on the cable.

TCP in hardware is not new; other implementations are presented in [1] and [26].

This new implementation is designed to allow sending with the lowest possible latency from user level software.

A full TCP implementation is a very complex undertaking, but for the trading application the following limitations are acceptable:

- only outgoing connections
- no retransmission
- no slow start and congestion avoidance
- only sending has to be fast and low latency
- no standard socket API

The major simplification is that the implementation only has to support outgoing connections and in the first implementation even only one single connection.

This special application does not require retransmission capabilities, because the sending of orders happens in a very clean network of fast switches, which have virtually no load. Therefore, packet lose is so seldom that it is not necessary to implement retransmission.

Furthermore, it would make no sense to retransmit an order, because the order would be too late at the order server, at least if the implementation uses a sensible retransmission time. Before a TCP stack retransmits a packet it should wait at least the usual time it takes for the other side to send an ACK otherwise many unnecessary retransmission would be under way.

TCP is using IP to transport its packets over network and IP provides neither reliability nor flow control. Buffers will fill up and packets will be dropped at some point, if the data rate it too high in an IP network. The TCP layer is only indirectly informed about the dropping of packets by the lack of acknowledgement (ACK) packets. Therefore elaborate algorithms have been developed for TCP to test the highest possible data rate between two nodes. Some of these methods are compared in [148].

However, such a mechanism is for this special application not required, because the network is very fast and only very few data is transferred over it.

Only the sending of packets has to be low latency, the return direction is only required for TCP (ACK) packets and reports from the stock exchange about the status of orders and it is not important to have the lowest possible latency for these reports.

The software in the trading server can be adapted. Therefore, it is not necessary to implement the standard UNIX socket API [79].

5.2.1 Implementation

There are different choices on how much of the low latency TCP stack has to be implemented in hardware and how much of it may be implemented in software.

A pure software implementation would have the big advantage that the development of software requires less effort than the development of hardware. Adding the TCP header to the data and the other necessary steps for sending could also be done quiet fast in software. Then the finished Ethernet packet could be transferred to the hardware. This would still have a lower latency than the standard solution in [124], because the software could be much simpler than a general TCP stack and furthermore PIO could be used instead of DMA to write the packet to the hardware. The advantage of PIO from the host CPU compared to DMA by the FPGA is that it does not involve an extra round-trip on the interface between FPGA and the host system. Adding the header in hardware has the advantage that fewer data has to be transferred to the hardware, and thus the transmission time can be reduced.

Therefore, the packet framing is done by the hardware, because it is the path that has to be optimized for latency. The receiving is not latency critical. Consequently, this part can be done in software. This is also the argument against a hardware only implementation, doing all the TCP processing in hardware is unnecessary development effort.

The different design choices are also summarized in figure 5-5.

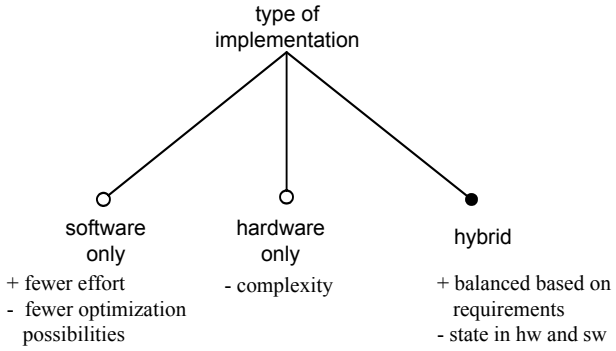


Figure 5-5: Design Space: Types of Possible Implementations

The implementation consists in the end of three hardware modules and a software library as depicted in figure 5-6. These components have to interact to fulfill the following three tasks:

- setup
- sending
- receiving

The **setup** of the connection is done by the software handler, it sets values in the RF and prepares the receive queue in the main memory of the host system.

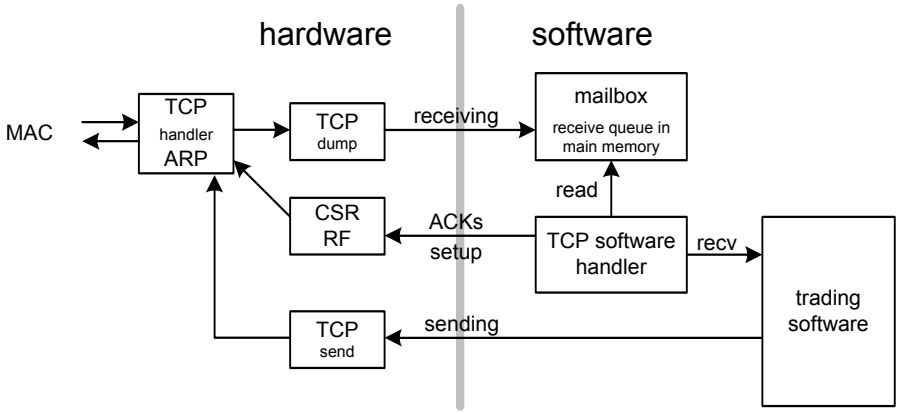


Figure 5-6: Hybrid TCP Stack Components

Sending is done by direct writes to the TCP sent unit without any framing in software.

Receiving involves multiple components. The TCP dump unit writes all received packets into a queue residing in the main memory, which is also called mailbox. The software handler reads the packets from the queue and sends ACKs if necessary. However, the software

handler is no extra thread and runs in the context of the trading software. Thus, the trading software has to call the software handler regularly, because it can otherwise not send ACKs. By polling the software handler the trading software will receive new data from the TCP connection.

5.3 Simulation Environment and Testing

The sending and receiving with TCP also does have to be tested. Therefore, first a UVM verification environment was developed, however it did not proof to be feasible for testing, because of several reasons.

The problem with such a full verification environment is that a tremendous amount of development time is required, because multiple components that are given in the environment of real hardware do not exist and have to be developed.

Figure 5-7 depicts the environment of the hardware device. When focusing on the TCP implementation there are two big components that interact with the FPGA device: the order server and the host system.

For the purpose of the simulation, the order server is a server system that accepts TCP connections, accepts data and sends data. The server system runs a standard Linux system.

The software part of the new TCP stack is mainly responsible for receiving data from the TCP receive queue and sending ACKs.

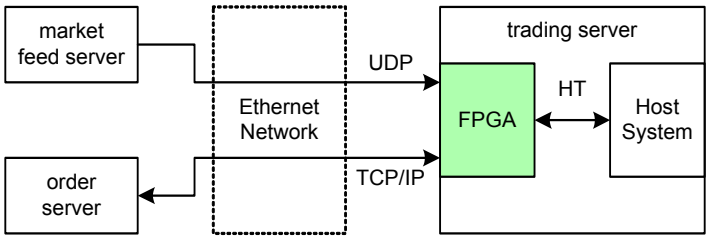


Figure 5-7: Device Environment

Thus, for testing TCP in the verification environment these two components have to be rebuilt: TCP/IP server and the software part of the hybrid TCP stack.

However, the problem has been that there are mismatches. Neither the simulated TCP server nor the reimplementation of the host software was working exactly like the real one. Thus, the system was not working correctly in hardware and a lot of debugging had to be

done in the FPGA. This is very time consuming and tedious, because the possibilities to debug a design in the FPGA are very limited. Furthermore, for every test a bit file has to be generated and this can take multiple hours.

Due to the complexity of the design it was decided to resort to a Correct by Application (CbA) [21] approach with testing only, because of the difficulties with the verification environment.

However, the ability to debug the system, if there is a problem, is mandatory. Therefore, the question had to be asked if it is possible to connect the hardware simulation to the real environment. This means that the simulation can be used with exactly the same software like the hardware and exactly the same TCP stack on the side of the order server. Thus, in the case of an error in hardware the simulation can be used to reproduce and analyze the problem.

Two extensions are required for the hardware simulation to make this possible:

- connection to a network
- method to interact with the software

The first requirement can be fulfilled with a TAP device and the later with a new component called Bus Function Model Server (BFMS).

5.3.1 TAP device

The TAP [127] device is a feature of the Linux kernel that allows simulating an Ethernet device at the user level. Other operating systems are offering similar capabilities.

Often TUN and TAP devices are mentioned together. The difference between them is that TUN is operating with layer 3 packets in the ISO 7-layer model [7], and is therefore on the IP packet level. However, the accelerator design will be connected to an Ethernet network and therefore it also has to support the Address Resolution Protocol (ARP) [2]. Thus, a TAP device has to be used, instead of TUN, because it handles packets on layer 2.

In figure 5-8 it is demonstrated how a TAP device can be used. The user level process **(A)** can open a tap device to use normal read and write system calls to either receive or send Ethernet packets. To the network layer it will look like packets from or to a real NIC. For the user level process **(C)** there is also no difference between a physical NIC and a TAP device.

TAP devices are usually either used for virtual private networks (VPN) or for virtual machines. By bridging between the device driver **(B)** and the TAP device either the VPN or the virtual machine are connected to the Ethernet network transparently. For example, OpenVPN is using tap devices for bridging Ethernet networks [151] over secure connections.

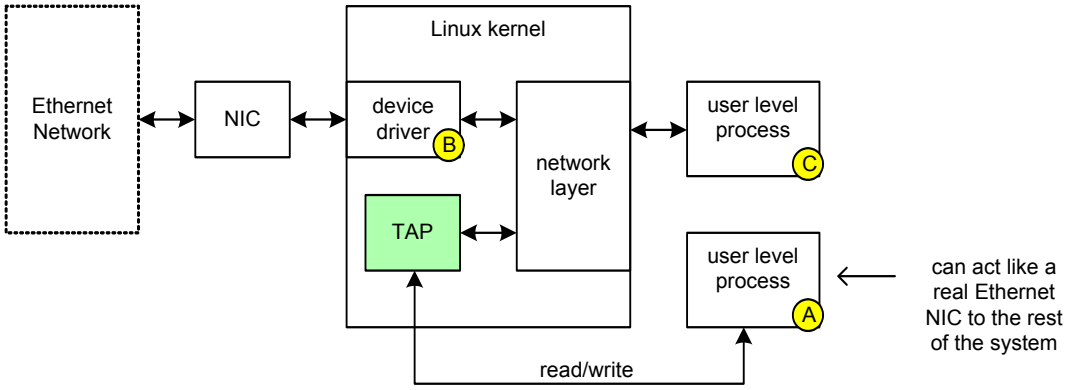


Figure 5-8: TAP Device

Consequently, it is also possible to use the TAP device to connect a hardware simulator to the network layer or a physical network.

5.3.2 BFMS

The complete design as shown in figure 5-9 is controlled from software by reads and writes. Furthermore, it is writing to the main memory of the host system to write to the TCP receive mailbox. The UDP and FAST functionality are using the same methods.

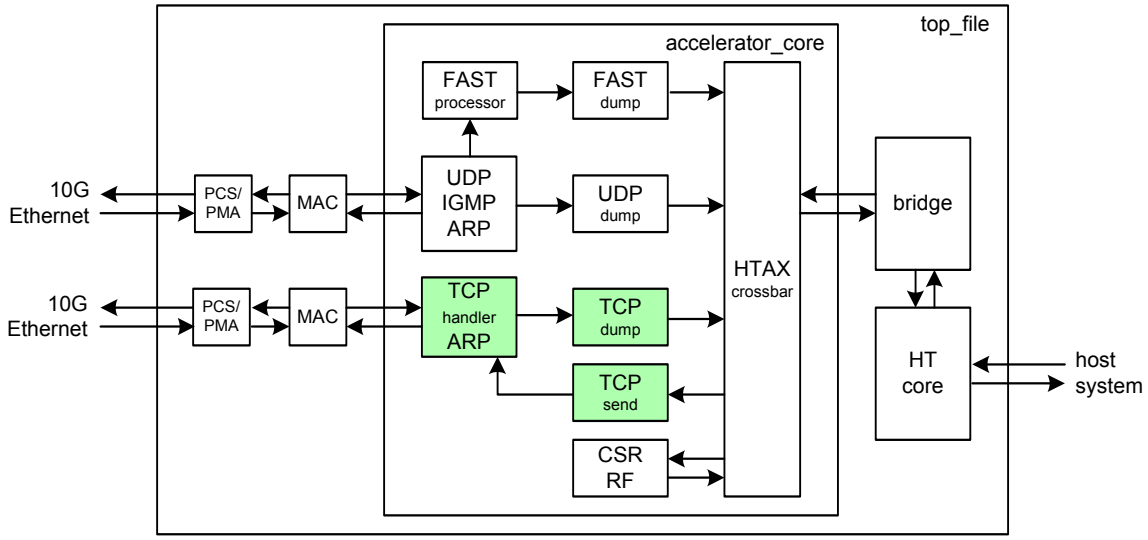


Figure 5-9: Overview Accelerator Architecture

There are two possibilities how the control software could be connected to the simulator. The first possibility is to connect a virtual machine to the simulator and the control software could run inside this virtual machine. QEMU [149] could be used exactly in the same way as in [116].

The other possibility is to modify the control software slightly so that it can run as normal user level process on the same system like the simulator.

After evaluating the two possibilities it was decided to implement the later possibility, because QEMU had the problem that it was not possible to perform 64 bit writes and reads from the emulated processor on emulated devices, but the TCP send unit relies on that.

To allow connecting the DUT with different normal user level programs it was decided to implement the Bus Functional Model Server (BFMS) as server that accepts TCP/IP socket connections. These connections should not be mixed up with the TCP hardware implementation that is tested in the simulation. Therefore the connections between BFMS and its clients are only called “socket” connection in the following. In [117] also sockets are used to connect system emulation and hardware simulation, but in that case only these two components are connected.

BFMS is acting as communication hub between the DUT and different clients. This offers flexibility, because different programs can be started and make use of the hardware exactly like when the real hardware is used. For instance, there is a “debuginfo” program that reads RF entries and evaluates them for debugging purposes. With BFMS it can be used on the DUT while tests are running.

The resulting setup is depicted in figure 5-10. The accelerator_core is exactly the same as shown in figure 5-9, but the Ethernet MACs and especially the HT core are not included, because including them would require development effort and it would slow down the simulation. The Network HyperTransport (NHT) client connects to BFMS, which acts as root complex in the setup, and client 1 and 2 can act similar to user level programs and perform reads and writes. There can be only a single NHT or DUT connected to BFMS, but up to 32 clients.

The NHT client and the TAP interfaces in figure 5-10 are implemented in C, and the SystemVerilog interface code calls these C functions with the DPI-C [153] interface. The hardware simulator that was used for the development is Cadence NCSIM [154], but there are no fundamental obstacles against using the environment with another simulator.

Four types of communication have to be supported:

- client reads from the RF of the DUT
- client writes to the RF of the DUT

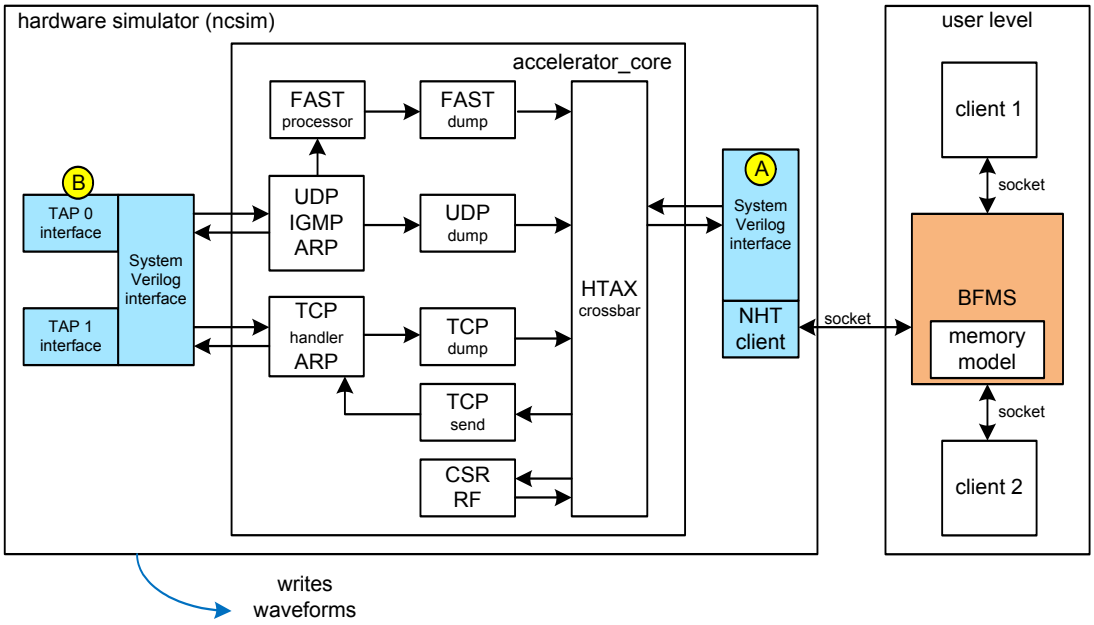


Figure 5-10: BFMS Simulation Environment

- client writes to the TCP send unit
- DUT writes to the main memory

The development target was to run the same control software on real hardware and with the simulator. Therefore, all read/write accesses to the RF were replaced in the C code by macros. Depending on if the BFMS define is set or not, either the RF is accessed directly or the bfms_client functions are used.

```
#ifndef BFMS
    #define RF_READ64(A) t->accel_rf->A
    #define RF_WRITE64(A,B) t->accel_rf->A=B
#else
    #define RF_READ64(A) bfms_client_read64(ADDRESS_OF(t->accel_rf->A))
    #define RF_WRITE64(A,B) bfms_client_write64(ADDRESS_OF(t->accel_rf->A),B)
#endif
```

The bfms_client provides these two and other functions in a single C file, this makes it possible to integrate these functions without a lot of effort. Writing to the TCP send unit is also done with the bfms_client_write64 function.

However, the reception of TCP packets is implemented by DMA writes from the TCP dump unit to the main memory. Thus, the accelerator_core as shown in figure 5-10 will emit HToC packets. The SystemVerilog wrapper **(A)** interprets these packets and forwards them over the socket interface to the BFMS, which has an internal memory model. The memory model mimics the main memory of the host system.

TCP software receives new packets by polling the mailbox and there are two possibilities to implement this:

- polling the memory model in BFMS
- bfms_client_mirror and polling normal memory

Clients can poll the memory model of BFMS by sending read requests to the BFMS. The disadvantage of this approach is that it is different from what the software is doing with real hardware, because there the mailboxes are mapped into the address space of the user level processes. Therefore, they can detect new packets and read the content of these packets transparently without calls to any functions. Consequently, it would be preferable if the software could operate in the same way with BFMS.

The bfms_client_mirror permits this. It runs as extra thread in the user level process and connects to BFMS to register itself for a range of memory. As a result, BFMS will forward all writes, from the DUT, to the specified memory range also to the bfms_client_mirror thread, which will write the data into the address space of the software, exactly like DMA hardware.

Receiving of UDP and FAST data works in the same way. BFMS supports multiple bfms_client_mirror threads concurrently.

Figure 5-11 summarizes the data flow in the whole setup. The red (curved) arrows symbolize the logical data flow, but all actual data or requests are transported by BFMS over the socket connections.

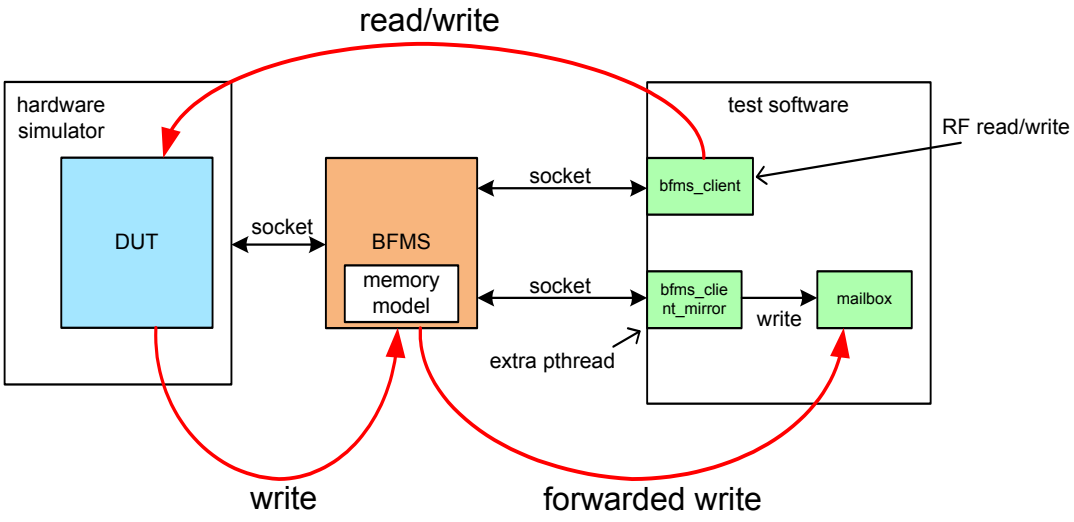


Figure 5-11: Data Flow

Using the simulation environment requires that the different components are started in the right order. First BFMS has to be started, then the hardware simulator which connects to BFMS. The clients, which make use of the DUT, can be started, as soon as the simulation is running.

Protocol

BFMS and its clients are using a straightforward protocol for communication, consisting of messages that are sent over the socket connection. Each message consists, as depicted in figure 5-12, of a 64 bit header and between 0 and 512 quadwords of payload. The header consists of the following fields:

- 1 byte: magic value 0x2A
- 1 byte: message type
- 1 byte: virtual channel
- 2 bytes: size
- 1 byte: source tag
- 2 bytes: reserved

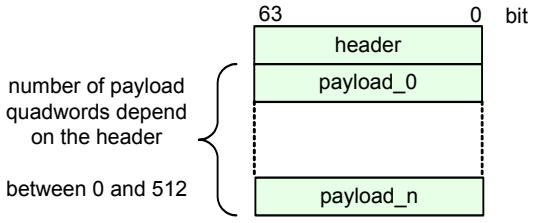


Figure 5-12: BFMS Message

After a client establishes a socket connection it has to send a login message. Depending on the message type BFMS detects the type of client. There are the following message types:

- 0x02: HT message type (not a login message)
- 0x03: flow control login
- 0xF1: DUT login
- 0xF2: bfms_client login
- 0xF3: bfms_client_mirror login

The HT message type supports a message size of up to 64 byte, despite the message format is designed to support bigger sizes. The extra bits are reserved for future extensions.

Figure 5-13 shows an example communication where the DUT and a bfms_client login to BFMS and the bfms_client performs a read and a write on the DUT.

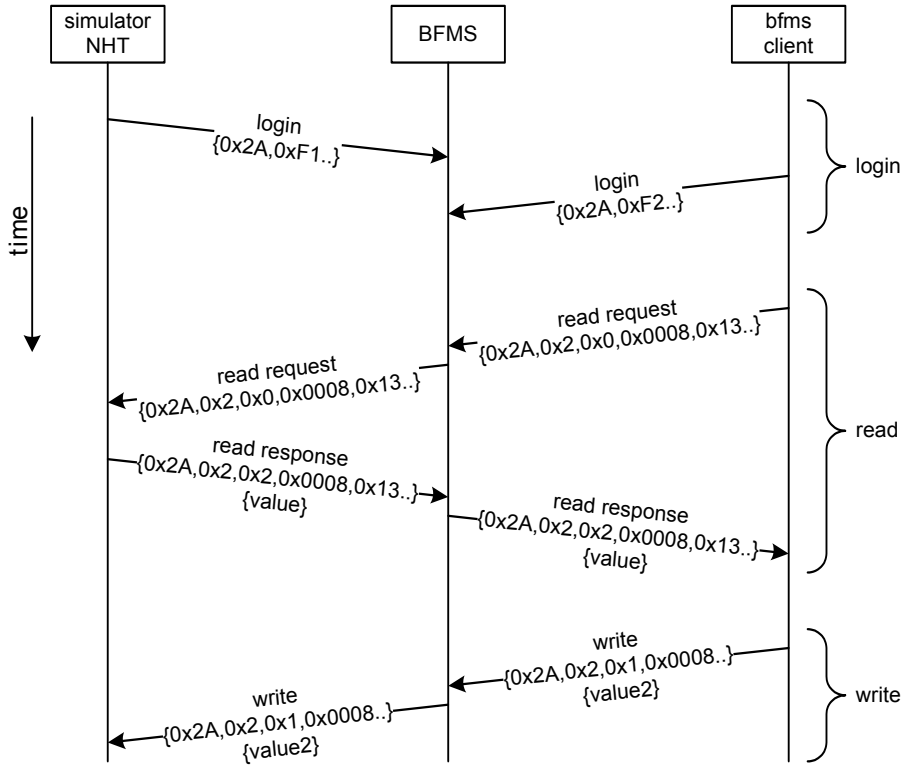


Figure 5-13: Communication between BFMS and Clients

Ethernet Flow Control

The trading accelerator as shown in figure 5-9 also contains units for receiving UDP packets and the decoding of FAST messages. These should also be usable in conjunction with BFMS.

The simulation is much slower than real hardware and therefore flow control is a problem, it is very easy for the test software to send so many packets to the hardware simulator that a very big backlog is the result. Two solutions to this problem are possible: slowing down and flow control

Slowing the stream down has the advantage that it is very easy to implement. Of course this directly brings up the question of how slow the stream has to be, so that the simulation can follow. This question is not easy to answer, but experience has shown that the simulation is between 10000 and 100000 times slower than the FPGA. However, slowing down the stream of UDP packets too much wastes a lot of time.

Therefore, flow control was implemented. This mechanism makes also use of BFMS. The tap interface function at **(B)** in figure 5-10 informs BFMS when the queue has a certain fill level, and the program that replays UDP streams requests the information from BFMS.

5.4 Benchmark

The trading accelerator is designed to receive information from UDP and send out orders over TCP as depicted in figure 5-14. To evaluate the performance of the new low latency TCP implementation an experiment was performed in the FPGA.

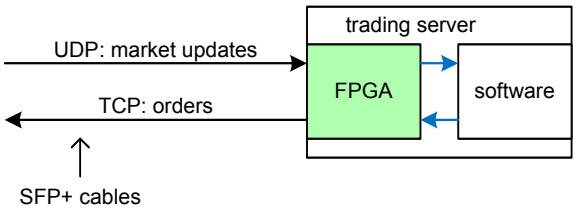


Figure 5-14: Benchmark Overview

Figure 5-15 shows the different components in the design and the path the data is flowing (curved red line). The test program that is running on the host system waits for an UDP packet. As soon as an UDP packet was received a 64 byte TCP packet will be send. The TCP connection was initiated before. The following stages were observed with the Xilinx ChipScope [158] FPGA logic analyzer:

- **(1)** UDP packet arrives at MAC
- **(2)** write UDP header to main memory
- **(3)** TCP data arrives in TCP send unit
- **(4)** data starts leaving MAC

Thus, the overall time that is spent in the logic and software is 1363 ns.

However, until that point only the latency from MAC to MAC was measured, but for an application it is of course more interesting how long it takes overall from cable to cable. There are more components between MAC and cable as it can be seen in figure 5-16.

The hardware that is used for this development is the Ventoux board as introduced in chapter 1.2.3. The board is not perfectly suited for 10G Ethernet, because its high speed serial links (SERDES) support only up to 6.6 Gbit/s, but modern 10G Ethernet networks make use of SFP+ cabling, which requires 10.3125 Gbit/s. Therefore a board with a XAUI transceiver [152] was developed to translate between SFP+ and XAUI with 4x3.125 Gbit/s.

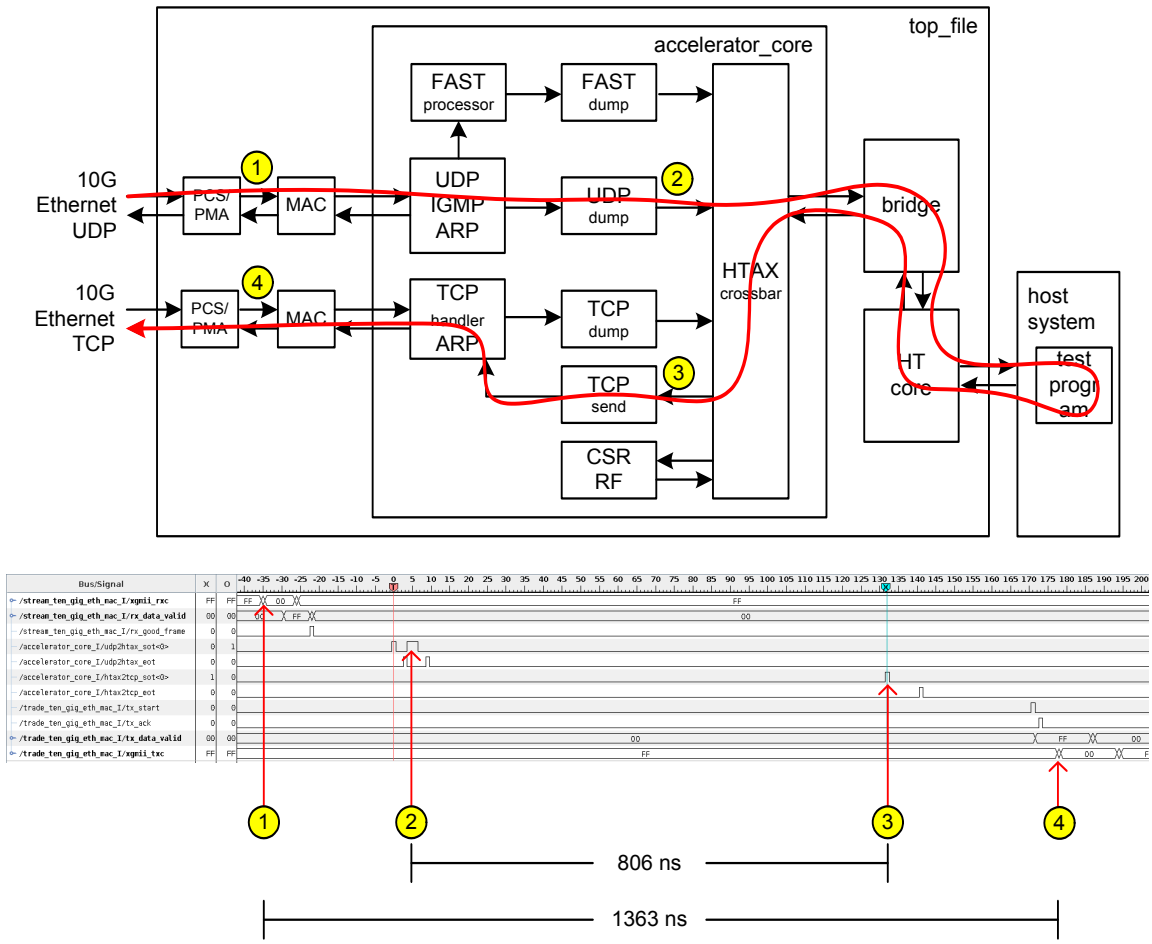


Figure 5-15: Benchmark Data Path

In figure 5-16 the complete path (curved red line) is shown and in the end it is only interesting how big the latency between **(A)** and **(C)** is. The latency, through the host system, on the path between **(B)** and **(D)** was already measured, but it is not possible to measure the latency between **(A)** and **(C)** directly, due to the lack of special equipment.

However, it is possible to measure the time between **(D)** and **(B)**, in the direction of the cables, by directly connecting the XAUI transceivers at **(A)** and **(C)** with a SFP+ cable. The result is 582.4 ns. This is the complete time and it is not possible to find out how big the difference in latency is between the send and the receive path, but in the end it does not matter, because both latencies are part of the overall latency.

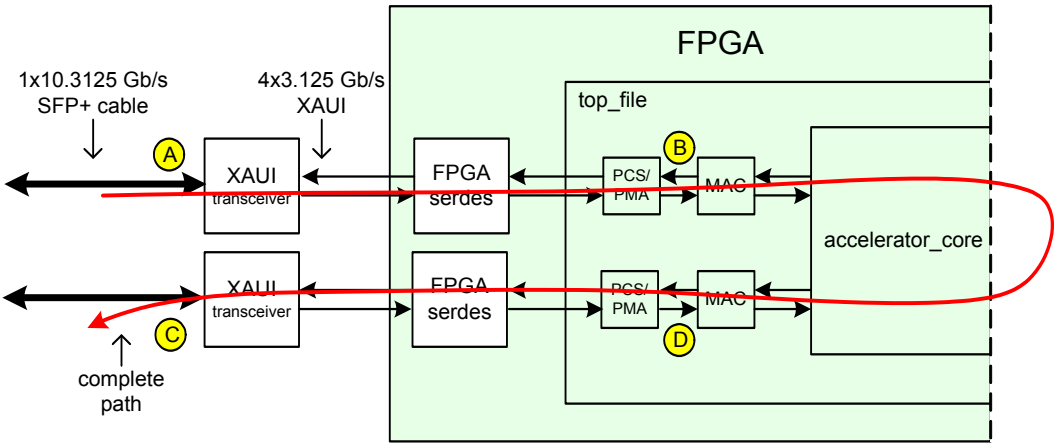


Figure 5-16: Benchmark Physical Interface

The result is that the lowest possible latency that can be achieved with this setup is cable to cable 1363 ns + 582 ns = 1945 ns. Real trading application would have a higher latency due to decoding, calculation and encoding which would have to be done in software.

It is not possible to measure, but it seems to be a reasonable assumption that half of the time is used for the TCP sending.

Chapter 6 Conclusion

The questions of design abstraction, implementation of pipelined micro architectures and simulation of complex application specific hardware structures have been discussed in this thesis based on four different projects.

The individual contributions of this work are an efficient generator for control and status register files (RF), a novel method to inform system software about state changes in the register file, a very efficient address translation unit and a new simulation environment for the co-simulation of complex hardware systems.

This work introduces a register file generator (RFS) which is used successfully in multiple projects. A high level description language has been designed for this purpose. It is based on XML and provides the means for an abstract description of RF structures.

The EXTOLL interconnection network makes extensive use of it and implements various different register files (RFs) depending on the target platform. Target platforms are different FPGA boards with HyperTransport and PCI express interface and also an ASIC. Despite these RFs are different they share the same input files with only very few differences, because parameterization allows to reuse the same XML files for different targets. Parameterization was implemented by using the standard C preprocessor markup language. The RF for the EXTOLL ASIC consists of a hierarchy of 73 single register files with 3967 elements like registers or counters, which are generated automatically by the RF generator. Thus, the design and verification time was reduced significantly. The Verilog code of this RF is 52607 lines long. Furthermore, RFS is also generating C header files, Verilog header files and documentation.

Hierarchical decomposition is one of the special features of RFS. It allows the separation of a RF in different sub-RFs. It is possible to integrate the corresponding sub-RFs directly into specific units of big designs by using special RF interfaces. This has the big advantage that the routing of the signals is simpler and therefore the timing is better. Additionally, the logical separation also reduces the wiring effort in the Verilog files.

One of the unique features is that RFS supports the automatic integration of counters in the RF for various applications. These counters are used in many hardware designs to analyze problems and performance properties. There are three types of counters and one of these uses edge detection to increase the value of the counter. This allows shorter and less error prone Verilog code when certain events have to be counted. Furthermore, the generator allows using special hardware structures on FPGAs, which allow resource preserving counter implementations.

Furthermore special mechanisms were implemented, which allow hazard free implementations of error reporting registers. Software can make use of the XOR operation to clear error bits, by doing so read-modify-write issues can be avoided.

The next contribution deals with the question what can be done to inform a host system faster in terms of latency and more efficient in terms of CPU time spent about changes in the previously introduced RF. Therefore, a novel system was implemented which consists of a hardware unit called system notification queue (SNQ) and software that uses special functionality from RFS to generate microcode automatically based on XML specifications. The automatically generated code can be extended with more functionality. The microcode of the SNQ can be exchanged at runtime and is able to perform reads and writes on the RF. Furthermore it is able to not only send data packets to the main memory but also over an interconnection network.

This unit is used within the EXTOLL NIC to implement an interrupt mechanism for the VELO and the RMA unit, each manages up to 256 queues and each of these can cause interrupts. Without the SNQ the hardware would issue an interrupt and then the software would perform multiple reads on the RF of the EXTOLL NIC to find out which sources did cause the interrupts. Reading 256 bits from a RF requires four reads and this takes about 1 microsecond, but with the SNQ this information will be pushed to the main memory and the access latency to the main memory is much smaller.

The SNQ can be used to read out different RF values after a hardware event with a very low latency between the reads. This permits a close time spatial correlation between the event and the different values from the RF. Furthermore, it is able to reset counters or error indicators in the RF after reading them out.

The next contribution of this work is an address translation unit (ATU2) with high throughput and low latency. This hardware unit implements a pipeline architecture that uses the concept of the replay FIFO to resolve read-modify-write hazards, flow control and ordering problems. The replay FIFO stores pipeline instructions, which can not be executed for resubmission. Thus, instructions will “rotate” through the pipeline until the conditions for

an execution of the instruction are fulfilled. This design provides efficient interfaces to remove entries from the translation cache and the possibility to preload entries from software.

Finally, a new method to implement the complex TCP protocol for a low latency trading application (HFT) was demonstrated. It is a hybrid implementation, which uses separation in a hardware and software part. The application requires sending of data with low latency, thus the sending part is implemented in hardware, but receiving is less performance critical and was implemented in software. This separation allows keeping the effort for the whole implementation low.

This complex software/hardware combination was tested with a novel combination of hardware simulation and in-system-test. A software component called Bus Functional Model Server (BFMS) was developed which serves as a communication hub between the different components of the simulation. The infrastructure allows using the software with small modification together the hardware simulation.

This solves two problems. First, it is possible to do full systems tests and debugging with the hardware design running in the simulator. Thus, the very limited debug possibilities in FPGAs and the time consuming generation of FPGA bit files can be avoided. Second, by using the hardware-software co-simulation it is not necessary to implement parts of the software again in the hardware test bench or verification environment, because the real software can be used instead.

The result is a fast system test environment that can be used to improve the design cycle time for complex hardware designs.

Representation Methods

Appendix A

Throughout this thesis multiple different representation methods are employed for descriptions. These include:

- design space diagrams
- representation of numbers and values
- hardware modules and their connections

Design Space Diagrams

For the purpose of visualizing the discussion of different design options, design space diagrams are employed throughout this work. The representation method that is used for these was inspired by [35] and [36]. Figure A-1 gives an overview of the naming convention for the different parts of such a diagram.

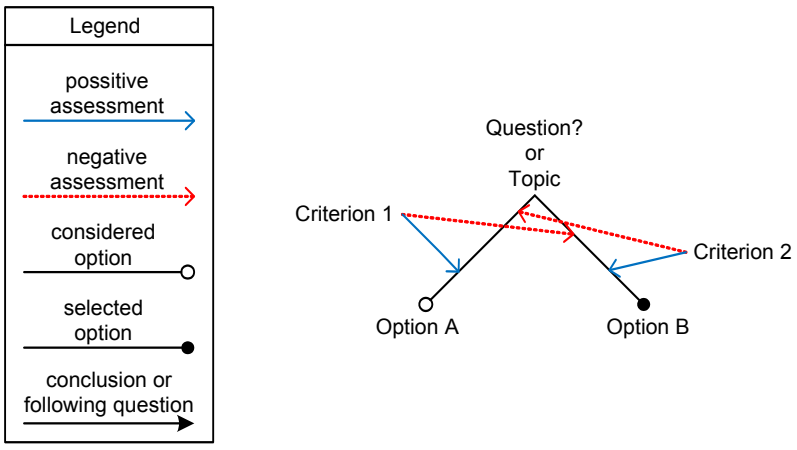


Figure A-1: Design Space Diagram Naming

In figure A-2 an example of such a design space diagram is shown. In this example the first question to be considered is “design space diagram?” and there are the two options no and yes. Either a design space diagram is used or not, there are two criteria that are considered in this case, the effort it takes to create a design space diagram on one side and on the other

side the overview that it will provide for the reader. After the positive decision to draw a design space diagrams a following question deals with the question in which orientation design space diagrams should be done. In this case the criteria are below the options because there is enough space.

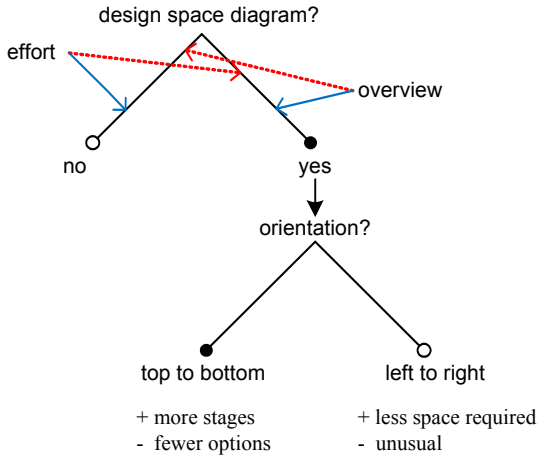


Figure A-2: Design Space Diagram Example

Presentation of Numbers

The Verilog format as described in [83] is used to represent numbers and bit patterns when it helps to prevent misunderstandings.

Values of XML attributes are given in the same presentation as in the XML files: "1"

Hardware modules

Hardware modules in this work are drawn as abstract boxes and only as detailed as necessary to support the writing. The terms hardware unit and module are used often synonymously in this work.

In figure A-3 two typical hardware modules and their connections are shown. Not all inputs and outputs are shown, especially the clock and reset inputs are omitted. Furthermore, also other signals, which are not significant for the descriptions, will not be shown.

The “1” in the yellow circle is used to mark points of special interest; text that refers to these will use **(1).**

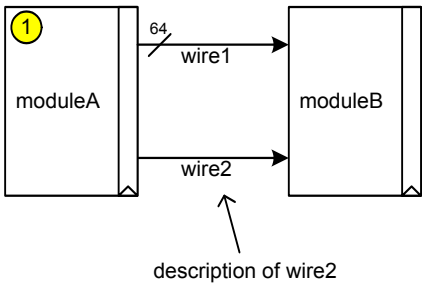


Figure A-3: Two Modules

It is assumed that all hardware modules have registered outputs, thus the output registers of modules are only shown if they are of relevance. Furthermore, hardware modules are connected by wires as it can be seen in figure A-3. In some cases the width is given like in the case of wire1. However, in most cases the width is omitted because it is not relevant or because the width is one.

The wires between these two modules are often called in this work synonymously: bus, data path or signal.

Some of the figures are more complex and wires are crossing, but wires are only connected when the intersection is market with a point as it can be seen in the case of A and B in figure A-4. The wires A and C in the figure are not connected.

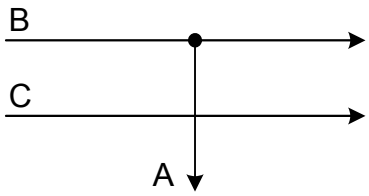


Figure A-4: Connection Between Wires

Acronyms

Appendix B

ACL	Access Control List
ASIC	Application Specific Integrated Circuit
ATC	Address Translation Cache
ATS	Address Translation Service
ATU	Address Translation Unit
BFM	Bus Functional Model
BFMS	Bus Functional Model Server
BRAM	Block RAM (e.g. Xilinx SRAM)
CAG	Computer Architecture Group (University of Heidelberg)
CAM	Content Addressable Memory
CbA	Correct by Application
CPP	C Pre-Processor
CPU	Central Processing Unit
DED	double error detected
DMA	Direct Memory Access
DSL	Domain Specific Language
DSP	Digital Signal Processor
DUT	Design Under Test
EDA	Electronic Design Automation
EXTOLL	EXTOLL R2 (Extended ATOLL)
EXU	EXtra Unit (unit that can be attached to EXTOLL)
FIFO	First In, First Out
FPGA	Field Programmable Gate Array

FU	Functional Unit
GAT	Global Address Table
GAST	Global Address Sub Table
HDL	Hardware Description Language
HFT	High Frequency Trading
HPC	High Performance Computing
HT	HyperTransport
HToC	HyperTransport On Chip
HVL	Hardware Verification Language
IOMMU	I/O Memory Management Unit
IP	Internet Protocol
kB	kilobyte
kbit	kilobit
LRU	Least Recently Used
LSB	Least Significant Bits
MCE	Machine Check Exception
ME	Microcode Engine
MMU	Memory Management Unit
MSB	Most Significant Bits
MSI	Message Signaled Interrupt
MTU	Maximum Transmission Unit
MUX	Multiplexer
NHT	Network HyperTransport
NIC	Network Interface Controller
NLA	Network Logical Address
NLP	Network Logical Page
NLPA	Network Logical Page Address
OS	Operating System
PA	Physical Address
PCI	Peripheral Component Interconnect

PCIe	PCI Express
PS	Prepare Store
RAM	Random Access Memory
RC	Register file Cell
RDL	Register Definition Language
RDMA	Remote Direct Memory Access
RF	Register File (plural: RFs)
RFS	Register File System
RMA	Remote Memory Access
RST	reStructuredText
RTL	Register Transfer Logic
SEC	single error corrected
SNQ	System Notification Queue
SNQC	SNQ Compiler
SNQM	SNQ Mailbox
SOC	System on a Chip
SPADIC	Self-triggered Pulse Amplification and Digitalization asIC
SRAM	Static Random Access Memory
TC	Trigger Control
TCP	Transmission Control Protocol
TDG	Trigger Dump Group
TE	Trigger Event
TLB	Translation Lookaside Buffer
UVM	Universal Verification Methodology
UVC	Universal Verification Component
VA	Virtual Address
VC	Virtual Channel
VELO	Virtualized Engine for Low Overhead
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

VPID	Virtual Process Identifier
VPN	Virtual Private Network
WCB	Write Combining Buffer
XBAR	Crossbar
XML	Extensible Markup Language

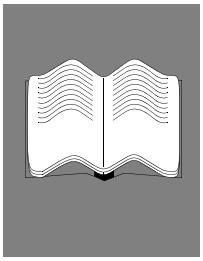
List of Figures

Appendix C

List of Tables

Appendix D

References



[1] David V. Schuehler and John W. Lockwood; A Modular System for FPGA-Based TCP Flow Processing in High-Speed Networks; Proceedings 14th International Conference, Field Programmable Logic and Applications (FPL), 2004, Leuven, Belgium.

[2] W. Richard Stevens, Gary R. Wright; TCP/IP Illustrated II: The Implementation; Addison-Wesley Longman, ISBN 978-0201633542, 1995, Amsterdam.

[3] AMD BIOS and Kernel Developer's Guide(BKDG) For AMD Family 10h Processors; http://support.amd.com/us/Processor_TechDocs/31116.pdf, retrieved Oct. 2011.

[4] Extensible Markup Language (XML) 1.0 (Fifth Edition); http://www.w3.org/TR/2008/PER-xml-20080205/ retrieved Oct. 2011.

[5] Floyd Wilder; A Guide to the TCP/IP Protocol Suite Second Edition; Artech House telecommunications library, 1998.

[6] Mondrian Nüßle; Acceleration of the Hardware - Software Interface of a Communication Device for Parallel Systems; Dissertation, Ph.D. Thesis, University of Mannheim, 2008.

[7] Douglas E. Comer; Internetworking with TCP/IP Volume: Vol. 1 Principles, Protocols, and Architectures This Edition; Prentice Hall, 1995, New Jersey, USA.

[8] Mikrocontroller.net website: FPGA Soft Core; http://www.mikrocontroller.net/articles/FPGA_Soft_Core, retrieved Nov. 2011.

[9] Xilinx Virtex-6 FPGA Memory Resources - User Guide - UG363 (v1.6) April 22, 2011; http://www.xilinx.com/support/documentation/user_guides/ug363.pdf, retrieved Nov. 2011.

[10] Xilinx Virtex-6 Family Overview - DS150 (v2.3) March 24, 2011; http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, retrieved Nov. 2011.

[11] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich and John Wilkes; An implementation of the Hamlyn sender-managed interface architecture; OSDI '96, Proceedings of the second USENIX symposium on Operating systems design and implementation, pages 245 - 259, 1996.

[12] Matt Welsh, Anindya Basu, Thorsten von Eicken; Incorporating Memory Management into User-Level Network Interfaces; Proceedings of Hot Interconnects V, 1997.

[13] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, Gernot Heiser; User-level Device Drivers: Achieved Performance; Journal of Computer Science and Technology, 2005.

[14] David Slogsnat, Alexander Giese, Mondrian Nuessle and Ulrich Bruening; An Open-Source HyperTransport Core; ACM Transactions on Reconfigurable Technology and Systems (TRETS) Volume 1 Issue 3, 2008.

[15] Todd M. Austin, Gurindar S. Sohi; High-bandwidth address translation for multiple-issue processors; 23rd Annual International Symposium on Computer Architecture (ISCA'96), 1996, Philadelphia, Pennsylvania, United States.

[16] MicroBlaze Soft Processor Core; http://www.xilinx.com/tools/microblaze.htm, retrieved Nov. 2011.

[17] MicroBlaze Soft Processor FAQs; http://www.xilinx.com/products/design_resources/proc_central/microblaze_faq.pdf, retrieved Nov. 2011.

[18] Roman Lysecky, Frank Vahid; Design and implementation of a MicroBlaze-based warp processor; ACM Transactions on Embedded Computing Systems (TECS), 2009, New York, NY, USA.

[19] Xilinx Virtex-6 FPGA Integrated Block for PCI Express User Guide, UG517 (v5.1) September 21, 2010; http://www.xilinx.com/support/documentation/user_guides/v6_pcie_ug517.pdf, retrieved Apr. 2012.

[20] Host/Target specific installation notes for GCC; http://gcc.gnu.org/install/specific.html, retrieved Nov. 2011.

[21] Amir Hekmatpour, Robert Devins, Dave Roberts, Michael Hale; An Integrated Methodology for SoC Design, Verification, and Application Development; Proceedings of The International Embedded Solution GSPx Conf., 2004.

[22] Wilson Snyder; 505 Registers or Bust; 2001 Boston SNUG, available from http://www.veripool.org/papers/vregs_snug.pdf, retrieved Nov. 2011.

[23] Vregs website; http://www.veripool.org/wiki/vregs, retrieved Nov. 2011.

[24] Daniel Mattsson, Marcus Christensson; Evaluation of synthesizable CPU cores; Master Thesis, Computer Science and Engineering Program, Chalmers University of Technology, Department of Computer Engineering, 2004, Gothenburg, Sweden.

[25] Mondrian Nüssle, Martin Scherer, Ulrich Brüning; A resource optimized remote-memory-access architecture for low-latency communication; The 38th International Conference on Parallel Processing (ICPP-2009), 2009, Vienna, Austria.

[26] Apostolos Dollas, Ioannis Ermis, Iosif Koidis, Ioannis Zisis, Christopher Kachris; An Open TCP/IP Core for Reconfigurable Logic; 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05), pages 297-298, 2005.

[27] Henry S. Warren; Hacker's Delight; Addison-Wesley Professional, ISBN 978-0201914658, 2002.

[28] FIX Adapted for STreaming - FAST Protocol; http://fixprotocol.org/fast retrieved May 2012.

[29] Gordon E. Moore; Cramming more components onto integrated circuits; Electronics, Volume 38, Number 8, 1965.

[30] HyperTransport I/O Link Specification Revision 3.10c; http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0035.pdf, retrieved Nov. 2011.

[31] Denali Blueprint Website; http://www.denali.com/en/products/blueprint.jsp, retrieved Nov. 2011.

[32] D. Crockford; The application/json Media Type for JavaScript Object Notation (JSON); IETF RFC 4627, retrieved Nov. 2011.

[33] JSON.org website; http://json.org/, retrieved Nov. 2011.

[34] Duolog Socrates, Bitwise website; http://www.duolog.com/products/bitwise/, retrieved Nov. 2011.

[35] Allan MacLean, Richard Young, Victoria Bellotti and Thomas Moran; Design Space Analysis: Bridging from Theory to Practice via Design Rationale; Proceedings of Esprit ’91 pages 720-730, 1991, Brussels, Belgium.

[36] Allan MacLean, Richard M. Young, Victoria M.E. Bellotti & Thomas P. Moran; Questions, Options, and Criteria: Elements of Design Space Analysis; Journal of Human–Computer Interaction, Volume 6, Issue 3, pages 201-250, 1991.

[37] Markus Müller; Exploring the Testability Methodology and the Development of Test and Debug Functions for a Complex Network ASIC; Diploma Thesis presented to the Computer Engineering Department, University of Heidelberg, 2011.

[38] Heiner Litz, Holger Fröning, Ulrich Bruening; A Novel Framework for Flexible and High Performance Networks-on-Chip; Fourth Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC) in conjunction with Hipeac, 2010, Pisa, Italy.

[39] Gordon E. Moore; Progress in Digital Integrated Electronics; Technical Digest, IEEE International, Electron Devices Meeting 21, 1975.

[40] Computer Architecture Group website; http://ra.ziti.uni-heidelberg.de/index.php, retrieved Dec. 2011.

[41] Holger Fröning, Mondrian Nüssle, David Slogsnat, Heiner Litz, Ulrich Brüning; The HTX-Board: A Rapid Prototyping Station; 3rd annual FPGAworld Conference, 2006, Stockholm, Sweden.

[42] Ulrich Brüning, Holger Fröning, Patrick R. Schulz, Lars Rzymianowicz; ATOLL: Performance and Cost Optimization of a SAN Interconnect; IASTED Conference: Parallel and Distributed Computing and Systems (PDCS), 2002, Cambridge, USA.

[43] Ajay V. Bhatt; Creating a Third Generation I/O Interconnect; Desktop Architecture Labs, Intel Corporation, http://all-electronics.de/ai/resources/63bc6da9d93.pdf, retrieved Dec. 2011.

[44] 82559ER Fast Ethernet PCI Controller Datasheet; http://pdos.csail.mit.edu/6.828/2008/readings/82559ER_datasheet.pdf, retrieved Dec. 2011.

[45] press release “Denali's Blueprint Employed by Atheros to Enhance SoC Design Productivity and Enable Rapid IP Reusability”; http://www.design-reuse.com/news/17902/esl-methodology.html, retrieved Dec. 2011.

[46] SystemRDL v1.0: A specification for a Register Description Language; Register Description Working Group, The SPIRIT Consortium; http://www.accellera.org/downloads/standards/SystemRDL_1.0.zip, retrieved Dec. 2011.

[47] Jan Decaluwe; MyHDL: a python-based hardware description language; Linux Journal, Issue 127, 2004, ISSN 1075-3583, http://www.linuxjournal.com/article/7542, retrieved Dec. 2011.

[48] Heiner Litz; Improving the Scalability of High Performance Computer Systems; Dissertation, Ph.D. Thesis, University of Mannheim, 2010.

[49] Altera website: Stratix IV; http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iv/stxiv-index.jsp, retrieved Dec. 2011.

[50] Altera website: Stratix V; http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp, retrieved Dec. 2011.

[51] Madhusudhan Talluri, Mark D. Hill; Surpassing the TLB performance of superpages with less operating system support; ASPLOS-VI Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, ACM New York, 1994, NY, USA.

[52] William Lee Irwin III; A 2.5 Page Clustering Implementation; Proceedings of the-Linux Symposium, 2003, Ottawa, Ontario, Canada.

[53] press release: Cypress Incorporates Denali Blueprint for CSR Automation Management; http://www.soccentral.com/results.asp?EntryID=26099, retrieved Dec. 2011.

[54] SystemVerilog website; http://www.systemverilog.org/, retrieved Dec. 2011.

[55] SynaptiCAD website: V2V; http://www.syncad.com/verilog_vhdl_translator.htm, retrieved Dec. 2011.

[56] X-Tek Corporation website: X-HDL; http://www.x-tekcorp.com/xhdl.html, retrieved Dec. 2011.

[57] Doxygen website: Doxygen Documentation, Generate documentation from source code; http://www.stack.nl/~dimitri/doxygen/index.html, retrieved Jan. 2012.

[58] Docbook website; http://www.docbook.org/, retireved Jan. 2012.

[59] Norman Walsh, Leonard Muellner; DocBook: the definitive guide; O'Reilly Media, ISBN 1-56592-580-7, 1999.

[60] W. Richard Stevens; Advanced Programming in the UNIX Environment; Addison Wesley, ISBN 0-201-56317-7, 1993.

[61] Bruce Schneier; Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition; Wiley, ISBN 978-0471117094, 1996.

[62] Igor Pavlow; LZMA SDK (software Development Kit); http://www.7-zip.org/sdk.html, retrieved Jan. 2012.

[63] Website HyperTransport Consortium; http://www.hypertransport.org/default.cfm?page=Technology, retrieved Jan. 2012.

[64] Ren´e J. Glaise; A two-step computation of cyclic redundancy code CRC-32 for ATM networks; IBM Journal of Research and Development, Volume 41, Issue 6, pages 705 - 710 , 1997.

[65] T. Grötker, S. Liao, G. Martin, S. Swan; System Design with SystemC; Springer, ISBN 1-4020-7072-1, 2002.

[66] Catapult C website; http://www.mentor.com/esl/catapult/details, retrieved Jan. 2012.

[67] Grant Martin, Gary Smith; High-Level Synthesis: Past, Present, and Future; Design & Test of Computers, IEEE, 2009.

[68] QT website; http://qt.nokia.com/, retrieved Jan. 2012.

[69] libxml2 website: The XML C parser and toolkit of Gnome; http://xmlsoft.org/, retrieved Jan. 2012.

[70] Xilinx: ISE Design Suite Overview; http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_c_overview.htm, retrieved Jan. 2012.

[71] Xilinx 7 Series FPGAs Overview; http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, retrieved Jan. 2012.

[72] William J. Dally, Charles L. Seitz; Deadlock Free Message Routing in Multiprocessor Interconnection Networks; IEEE Transactions on Computers, Volume 36, Issue 5, pages 547 - 553, May 1987.

[73] Bjarne Stroustrup; “The C++ Programming Language”; Addison-Wesley Professional, ISBN 978-0201889543, 1997.

[74] Jorge Ortiz; Synthesis Techniques for Semi-Custom Dynamically Reconfigurable Superscalar Processors; Ph.D. Thesis, Electrical Engineering & Computer Science, University of Kansas, 2009.

[75] Leong, P., Leong, M., Cheung, O., Tung, T., Kwok, C., Wong, M., and Lee, K.; Pilchard – a reconfigurable computing platform with memory slot interface; Proc. Symp. on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2001.

[76] Ben Juurlink; Approximating the optimal replacement algorithm; CF '04 Proceedings of the 1st conference on Computing frontiers, 2004.

[77] Patrick Mochel; The sysfs Filesystem; Proceedings of the 2005 Linux Symposium, 2005, Ottawa, Canada.

[78] J. Levesque, J. Larkin, M. Foster, J. Glenski, G. Geissler, S. Whalen, B. Waldecker, J. Carter, D. Skinner, H. He, H. Wasserman, J. Shalf, H. Shan, and E. Strohmaier; Understanding and Mitigating Multicore Performance Issues on the AMD Opteron Architecture; Lawrence Berkeley National Laboratory, 2007, http://escholarship.org/uc/item/9k38d8ms, retrieved Jan. 2012.

[79] Richard Stevens; UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI; Prentice Hall, ISBN 978-0134900124, 1998.

[80] Ellen Siever, Nathan Patwardhan, Stephen Spainhour; PERL in a Nutshell; O'Reilly & Associates, USA, ISBN 978-0596002411, 2002.

[81] Timo Reubold; Design, Implementation and Verification of a PCI Express to HyperTransport Protocol Bridge; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2008.

[82] Mark D. Hill; A Case for Direct-Mapped Caches; Computer, Volume 21, Issue 11, pages 25-40, 1988.

[83] Stuart Sutherland; Verilog HDL Quick Reference Guide; Sutherland HDL, Inc., 2001, http://www.sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf, retrieved Jan. 2012.

[84] Chris Sullivan, Alex Wilson, Stephen Chappell; Using C based logic synthesis to bridge the productivity gap; ASP-DAC '04 Proceedings of the Asia and South Pacific Design Automation Conference, 2004.

[85] LogiCORE IP DSP48 Macro v2.0; http://www.xilinx.com/support/documentation/ip_documentation/dsp48_macro_ds754.pdf, retrieved Jan. 2012.

[86] Virtex-6 FPGADSP48E1 Slice, User Guide, UG369 (v1.3) February 14, 2011; http://www.xilinx.com/support/documentation/user_guides/ug369.pdf, retrieved Jan. 2012.

[87] Altera, Stratix IV Device Handbook, Volume 1; http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf, retrieved Jan. 2012.

[88] Tim Armbruster, Peter Fischer and Ivan Peric; SPADIC - A Self-Triggered Pulse Amplification and Digitization ASIC; Nuclear Science Symposium Conference Record, 2010, Knoxville, USA.

[89] Heiner Litz, Holger Fröning, Mondrian Nüssle, Ulrich Brüning; VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers; 37th International Conference on Parallel Processing (ICPP-08), Sept. 08 - 12, 2008, Portland, Oregon, USA.

[90] Mondrian Nüssle, Benjamin Geib, Holger Fröning, Ulrich Brüning; An FPGA-based custom high performance interconnection network; 2009 International Conference on ReConFigurable Computing and FPGAs, December 9-11, 2009, Cancun, Mexico.

[91] Douglas Perry; VHDL : Programming By Example; McGraw-Hill Professional, 4th edition, ISBN 978-0071400701, 2002.

[92] Sphinx Website; http://sphinx.pocoo.org/, retrieved Jan. 2012.

[93] Wikipedia: Transistor Count and Moore's Law - 2011; http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg, retrieved Jan. 2012.

[94] C99 Standard, ISO/IEC 9899; draft available from http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, retrieved Jan. 2012.

[95] Spirit IP-XACT Standard, IEEE 1685-2009; IEEE Standards Association Corporate Advisory Group, ISBN 978-0-7381-6160-0, 2012, New York, USA.

[96] Getting Started with OVM, Tutorial 3 - The OVM Register Package; http://www.doulos.com/knowhow/sysverilog/ovm/tutorial_rgm_1/, retrieved Jan. 2012.

[97] The SPADIC Project Website; http://spadic.uni-hd.de/, retrieved Jan. 2012.

[98] Heiner Litz, Christian Leber, Benjamin Geib; DSL Programmable Engine for High Frequency Trading Acceleration; 4th Workshop on High Performance Computational Finance at SC11 (WHPCF 2011), co-located with SC11, November 13th, 2011, Seattle, USA.

[99] Christian Leber, Benjamin Geib, Heiner Litz; High Frequency Trading Acceleration using FPGAs; 21st International Conference on Field Programmable Logic and Applications (FPL 2011), September 5-7, 2011, Chania, Greece.

[100] Jose Duato, Sudhakar Yalamanchili, Lionel Ni; Interconnection Networks, The Morgan Kaufmann Series in Computer Architecture and Design, ISBN 1558608524, 2002.

[101] Cadence website, Adam Sherer: Scalable OVM Register and Memory Package; http://www.cadence.com/Community/blogs/fv/archive/2009/02/05/scalable-ovm-register-and-memory-package.aspx, retrieved Jan. 2012.

[102] Xilinx Virtex-6 Libraries Guide for HDL Designs, UG623 (v 12.3) September 21, 2010; http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/virtex6_hdl.pdf, retrieved Feb. 2012.

[103] Wei Hwang, Rajiv V. Joshi and Walter H. Henkels; A 500-MHz, 32-Word 64-Bit, Eight-Port Self-Resetting CMOS Register File; IEEE Journal of Solid-State Circuits, Volume 34, Issue 1, 1999.

[104] Xilinx PlanAhead User Guide UG632 (v13.3) October 19, 2011; http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/PlanAhead_UserGuide.pdf, retrieved Feb. 2012.

[105] XtremeDSP for Virtex-4 FPGAs User Guide, UG073 (v2.7) May 15, 2008; http://www.xilinx.com/support/documentation/user_guides/ug073.pdf, retrieved Feb. 2012.

[106] reStructuredText - Markup Syntax and Parser Component of Docutils; http://docutils.sourceforge.net/rst.html, retrieved Feb. 2012.

[107] pdfTeX website; http://www.tug.org/applications/pdftex/, retrieved Feb. 2012.

[108] PDTi - Register management simplified website; http://www.productive-eda.com/register-management/, retrieved Feb. 2012.

[109] Agnisys - IDesignSpec; http://agnisys.com/idesignspec, retrieved Feb. 2012.

[110] GenSys Registers; http://www.atrenta.com/solutions/gensys-family/gensys-registers.htm, retrieved Feb. 2012.

[111] Semifore CSRCompiler; http://www.semifore.com/index.php?option=com_content view=article&id=1&Itemid=10, retrieved Feb. 2012.

[112] Xilinx Virtex-6 FPGA Configurable Logic Block User Guide, UG364 (v1.2) February 3, 2012; http://www.xilinx.com/support/documentation/user_guides/ug364.pdf, retrieved Feb. 2012.

[113] Schemas of The SPIRIT Consortium; http://www.accellera.org/XMLSchema/SPIRIT, retrieved Feb. 2012.

[114] Benjamin Geib; Hardware Support for Efficient Packet Processing; Dissertation, Ph.D. thesis, University of Mannheim, 2012.

[115] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, Emmanuel Vaumorin; Industrial IP Integration Flows based on IP-XACT Standards; DATE '08 Proceedings of the conference on Design, automation and test in Europe, 2008.

[116] Màrius Montón, Antoni Portero, Marc Moreno, Borja Martínez, Jordi Carrabina; Mixed SW/SystemC SoC Emulation Framework; IEEE International Symposium on Industrial Electronics, 2007.

[117] Jing-Wun Lin, Chen-Chieh Wang, Chin-Yao Chang, Chung-Ho Chen, Kuen-Jong Lee, Yuan-Hua Chu, Jen-Chieh Yeh, Ying-Chuan Hsiao; Full System Simulation and Verification Framework; Proceedings of the 2009 Fifth International Conference on Information Assurance and Security, Volume 1, pages 165-168, 2009.

[118] Lattice Mico32 website; http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/, retrieved Feb. 2012.

[119] Xilinx Virtex-6 FPGA Data Sheet: DC and Switching Characteristics, DS152 (v3.4) January 12, 2012; http://www.xilinx.com/support/documentation/data_sheets/ds152.pdf, retrieved Feb. 2012.

[120] Subversion - an open-source version control system; http://subversion.apache.org/, retrieved Feb. 2012.

[121] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell; Pthreads Programming: A POSIX Standard for Better Multiprocessing (O'Reilly Nutshell); ISBN 1565921151, O'Reilly Media, 1996.

[122] Dan Clein; CMOS IC Layout: Concepts, Methodologies, and Tools; ISBN 0750671947; Newnes, 2000.

[123] Micron M25P64 datasheet; http://www.micron.com/~/media/Documents/Products/Data%20Sheet/NOR%20Flash/5980M25P64.ashx, retrieved Feb. 2012.

[124] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli and Siddharth Kulkarni; Architectural Breakdown of End-to-End Latency in a TCP/IP Network; International Journal of Parallel Programming, Springer Netherlands, 2009.

[125] Jeffrey C. Mogul; TCP offload is a dumb idea whose time has come; HOTOS'03 Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, 2003.

[126] Lattice PCI Express Endpoint IP Core (x1, x4) website; http://www.latticesemi.com/products/intellectualproperty/ipcores/pciexpress.cfm, retrieved Mar. 2012.

[127] Maxim Krasnyansky; Universal TUN/TAP device driver; http://kernel.org/doc/Documentation/networking/tuntap.txt, retrieved Mar. 2012.

[128] AMBA Open Specifications website; http://www.arm.com/products/system-ip/amba/amba-open-specifications.php, retrieved Mar. 2012.

[129] Xilinx Zynq-7000 Extensible Processing Platform Block Diagram and Product Features; http://www.xilinx.com/technology/roadmap/zynq7000/features.htm, retrieved Mar. 2012.

[130] Narayanan Ganapathy and Curt Schimmel; General Purpose Operating System Support for Multiple Page Sizes; USENIX Annual Technical Conference (NO 98), 1998.

[131] Madhusudhan Talluri, Shing Kong, Mark D. Hill, David A. Patterson; Tradeoffs in supporting two page sizes; ISCA '92 Proceedings of the 19th annual international symposium on Computer architecture, ACM New York, 1992, NY, USA.

[132] R.B. Tomasulo; An Efficient Algorithm for Exploiting Multiple Arithmetic Units; IBM Journal of Research and Development, Volume 11 Issue 1, pages 25-33, 1967.

[133] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, Tadashi Watanabe; The K computer: Japanese next-generation supercomputer development project; ISLPED '11 Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design, 2011.

[134] TOP500 Supercomputer Sites - Japan's K Computer Tops 10 Petaflop/s to Stay Atop TOP500 List; http://www.top500.org/lists/2011/11/press-release, retrieved Mar. 2012.

[135] Nadav Amit, Muli Ben-Yehuda, Ben-Ami Yassour; IOMMU: Strategies for Mitigating the IOTLB Bottleneck; WIOSCA '10: The Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture, 2010, Saint Malo, France.

[136] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-Z; http://download.intel.com/products/processor/manual/265667.pdf, retrieved April 2012.

[137] Xilinx Press Release # 0657, Xilinx Unveils 65nm Virtex-5 Family – Industry's Highest Performance Platform FPGAs; http://www.xilinx.com/prs_rls/2006/silicon_vir/0657v5family.htm, retrieved April 2012.

[138] Xilinx press release; New Xilinx Virtex-6 FPGA Family Designed to Satisfy Insatiable Demand for Higher Bandwidth and Lower Power Systems; http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1250609&highlight=, retrieved April 2012.

[139] Xilinx 7 Series FPGAs Slash Power Consumption by 50% and Reach 2 Million Logic Cells on Industry's First Scalable Architecture; http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsarticle&ID=1440020, retrieved April 2012.

[140] Lawrence C. Stewart and David Gingold; A New Generation of Cluster Interconnect; http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.7574, retrieved April 2012.

[141] Xilinx Content-Addressable Memory v6.1 Data Sheet, DS253; http://www.xilinx.com/support/documentation/ip_documentation/cam_ds253.pdf, retrieved April 2012.

[142] Frank Lemke; Fsmdesigner4 - development of a tool for interactive design and hardware description language generation of finite state machines; Diploma thesis, University of Mannheim, 2006.

[143] W. W. Terpstra; The Case for Soft-Cpus in Accelerator Control Systems; Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2011), 2011, Grenoble, France.

[144] Altera website, Nios II Processor: The World's Most Versatile Embedded Processor; http://www.altera.com/devices/processor/nios2/ni2-index.html, retrieved May 2012.

[145] Altera, IP Compiler for PCI Express User Guide May 2011; http://www.altera.com/literature/ug/ug_pci_express.pdf, retrieved May 2012.

[146] Lattice, PCI Express 2.0 x1, x4 Endpoint IP Core User's Guide, February 2012; http://www.latticesemi.com/documents/ipug75.pdf, retrieved May 2012.

[147] UVM (Standard Universal Verification Methodology); http://www.accellera.org/downloads/standards/uvm, retrieved May 2012.

[148] Kevin Fall and Sally Floyd; Comparisons of Tahoe, Reno, and Sack TCP; Computer Communication Review, Volume 26, pages 5-21, 1995.

[149] QEMU open source processor emulator website; http://wiki.qemu.org/Main_Page, retrieved May 2012.

[150] The Linux Kernel Archives website; http://www.kernel.org/, retrieved May 2012.

[151] OpenVPN: Ethernet Bridging; http://openvpn.net/index.php/open-source/documentation/miscellaneous/76-ethernet-bridging.html retrieved May 2012.

[152] Marvell Alaska 88X201x, Integrated 10 Gbps 802.3ae Compliant Ethernet Transceivers; http://www.marvell.com/transceivers/assets/Marvell-Alaska-X-88X201x-PHY.pdf, retrieved May 2012.

[153] SystemVerilog DPI; http://en.wikipedia.org/wiki/SystemVerilog_DPI, retrieved May 2012.

[154] Cadence - Incisive Design Team Simulator; http://www.cadence.com/products/ld/design_team_simulator/pages/default.aspx, retrieved May 2012.

[155] Frank Emnett and Mark Biegel; Power reduction through RTL clock gating; Synopsys Users Group (SNUG), 2000, San Jose, CA, USA.

[156] Mark Smotherman; A Brief History of Microprogramming; http://www.cs.clemson.edu/~mark/uprog.html, retrieved June 2012.

[157] Verilator website; http://www.veripool.org/wiki/verilator, retrieved June 2012.

[158] Xilinx ChipScope website; http://www.xilinx.com/tools/cspro.htm, retrieved June 2012.