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

2009

MSc THESIS

Protocol conversions for the Æthereal Networks-on-Chip

Jason de Windt

Abstract

CE-MS-2009-20

Bus based interconnects are commonly used to connect Intellectual Properties (IPs) on System-on-Chip (SoC). However, as the number of high performance IPs with large communication requirements in a Multi Processor SoC (MPSoC) increases, the bus interconnects become a communication bottleneck. To overcome this limitation, the bus based interconnects are replaced by Networks-on-Chip (NoC) as the interconnect for IPs in an MPSoC. The ability to support multiple protocols, both legacy and newer, is essential for leveraging the advantages of the NoC. This thesis describes the protocol conversions, or shells, we design for the Æthereal NoC, to provide a unified message format for the PLB, OPB and FSL protocols. Furthermore, to tolerate the latencies when accessing memory and to increase the throughput when possible, we add support for posted write and prefetch read in the shells and we design a mechanism to coalesce multiple single transactions into burst transaction when possible. To validate our design, we prototype it on a Virtex-II PRO FPGA. We use both synthetic applications as well as real life applications to benchmark and analyze the effect of factors, like computation-to-communication ratio, various link bandwidths and compiler loop unrolling on the performance of the system. The results show that reducing the link bandwidth up to a certain point does not affect the performance anymore, as the latency associated with NoC internals becomes dominant. Also the burst transaction can sustain the performance of the system up to a certain point, when the link bandwidth is reduced. The best result is obtained using a shell with posted write and prefetch read.

Faculty of Electrical Engineering, Mathematics and Computer Science

Protocol conversions for the Æthereal Networks-on-Chip
Providing interoperability with Coreconnect buses

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jason de Windt
born in Willemstad, Curaçao

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

Protocol conversions for the Æthereal Networks-on-Chip

by Jason de Windt

Abstract

Bus based interconnects are commonly used to connect Intellectual Properties (IPs) on System-on-Chip (SoC). However, as the number of high performance IPs with large communication requirements in a Multi Processor SoC (MPSoC) increases, the bus interconnects become a communication bottleneck. To overcome this limitation, the bus based interconnects are replaced by Networks-on-Chips (NoC) as the interconnect for IPs in an MPSoC. The ability to support multiple protocols, both legacy and newer, is essential for leveraging the advantages of the NoC. This thesis describes the protocol conversions, or shells, we design for the Æthereal NoC, to provide a unified message format for the PLB, OPB and FSL protocols. Furthermore, to tolerate the latencies when accessing memory and to increase the throughput when possible, we add support for posted write and prefetch read in the shells and we design a mechanism to coalesce multiple single transactions into burst transaction when possible. To validate our design, we prototype it on a Virtex-II PRO FPGA. We use both synthetic applications as well as real life applications to benchmark and analyze the effect of factors, like computation-to-communication ratio, various link bandwidths and compiler loop unrolling on the performance of the system. The results show that reducing the link bandwidth up to a certain point does not affect the performance anymore, as the latency associated with NoC internals becomes dominant. Also the burst transaction can sustain the performance of the system up to a certain point, when the link bandwidth is reduced. The best result is obtained using a shell with posted write and prefetch read.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-20

Committee Members :

Advisor:	
Chairperson:	Kees Goossens, CE , TU Delft
Member:	Arjan van Genderen, CE, TU Delft
Member:	Rene van Leuken, CE, TU Delft

I dedicate this thesis to my mother, father and closest relatives, for giving me unconditional love and support.

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I want to thank my advisors, Kees Goossens and Radu Stefan, for the excellent guidance I received during my thesis. Looking back, I can honestly say that I learned a lot by working with them.

Secondly, I want to thank my fellow Electrical Engineering students (most of whom I know since the beginning of my study in Delft), my roommates and my fraternity brothers, for their friendship, support, fun and good times throughout the years.

Last of all, a big thanks to all the professors, staff members and fellow students at the Computer Engineering department.

Jason de Windt
Delft, The Netherlands
September 9, 2009

Introduction 1

1.1 Trends

System-on-Chips are integrated electronic system [3] on a single chip. SoCs are widely used in embedded applications [26], [28] and are implemented on many integrated circuits (IC) technology [28], like ASIC (IBM Cell processors, smart phones) and FPGA (Xilinx Virtex platform).

1.1.1 SoC cost

The developments in the IC industry are driven by Moore's law, which states that each 18 months the number of transistors on a chip doubles. This increases the complexity associated with the design of the chip. Issues like testability, power management, and signal integrity become increasingly difficult to cope with.

To exploit the available computing power and to keep designing chips at the same rate, either the resources must double (design team) or the productivity must double for each generation. Else the time needed to design a chip doubles for every generation. A bigger design team each generation is not practical and feasible, due to factors like the mythical man-month [28]. Increasing productivity is challenging, as the chip is getting more complex each generation. Taking twice as long to design a chip is not really an option, due to economical factors like time to market. The cost for designing a chip also increases due to design, implementation and manufacturing challenges. The cost of a chip depends on:

- cost of manufacturing, includes the cost of the machines and packaging.
- cost of design (NRE), are one-time cost incurred for each new chip designed.
- number of chips sold

A solution to cope with the increasing chip cost is to design more efficiently. There are several possibilities to achieve this. The re-use of IPs and architecture enables the NRE cost to be spread over several products and reduces the complexity of the system, among others. Also, using a scalable architecture results in a system that is usable even if it consists of large number of components. Raising the abstraction level enables engineers to become more efficient. Lastly, using a system that is composable enables re-use of design testing and verification. Finally higher abstraction level enables engineers to utilize the exponential grow of transistors [3].

1.1.2 On-chip bus based interconnects

Traditionally, bus based on-chip interconnects [2], [18], [22], [24] are used to connect IPs on a SoC. The main advantages of using bus interconnect, described in [14], [20], are:

- low cost, as the IPs share the sets of wires.
- uniform interface to the IPs, facilitating system integration.

Current SoCs are complex, consisting of high performance IPs with large communication requirements, like CPU, DSP and video processors, and on-chip and off-chip memory. These systems are often referred to as Multi Processor SoC (MPSoC). The trend is toward MPSoC with massive computation power and communication requirements [3]. As the number of the IPs on a SoC increases, the bus interconnects become a communication bottleneck as their bandwidth is bounded. Some of the factors are as follows.

- Physical factors [1], [10], [14]: the IPs attached to the bus share the bandwidth. As the number of IPs increases, the capacitive loading on the bus grows, limiting the performance of the bus.
- Technology scaling [14], [15], [16]: the layout of a chip consists of a plane of transistors with planes of wires stacked on top of it, referred to as metal layers. The lower metal layers consists of short and thin wires for close communication (local wires). The top metal layers consist of long and fat wires for remote communication (global wires). For each generation the number of transistors doubles and the wires get thinner and slower, resulting in increasing delay for the wires. For the local wires, the effect is limited due to the short length. On contrary, for the global wires, the wire delay relative to the transistor delay increases for each generation. For each generation, the transistors become faster, achieving higher frequency and thus lower clock time. This constrains the performance of the global wires as it is harder for global signals to travel across the die in a single clock cycle.

As the result of technology scaling, the gap between on-chip computation and communication increases, resulting in a shift from a computation to a communication design scheme [3], [4].

1.1.3 Networks-on-Chip

Networks-on-Chip are proposed as interconnect for IPs on a SoC [5], to overcome the global communication limitations of bus based interconnects. Network-on-chips have the desirable property of using the global wires efficiently [3], [5]. This due to the fact that the IPs are not connected with each other through a full switch, but rather though several small switches [14]. As the wires between the IPs are not active simultaneously, they are shared on a single communication and on multiple communications. This results in fewer wires and better global wire utilization. Also, NoC decouple the IPs computation and communication, enabling the IPs to be reused more easily [3], [25]. Furthermore, NoC offer the following advantages, as described in [3], [5], [7], [8], [25].

- Flexibility: run time programmable, ability to offer different types of communication and QoS.
- Modular design: NoC building blocks are routers and network interfaces. The properties of the building blocks can be customized at design time. A NoC can realize any topology.
- Scalability: scalable in terms of number of IPs and bandwidth. The performance of the interconnect can be increased by using more links and routers. This increases the cost of the interconnect though.
- Protocol layering: The layer stack of the NoC architecture has several benefits. First, it enables the decomposition of problems into smaller and manageable pieces. Next, by using abstraction, the details of the layers are hidden. Each layer uses and offer services of the other layers. Protocols are used to implement a layer. Also, each layer can be optimized separately, without breaking the function of the other layers.
- Support for multiple communication protocol in use: supports both legacy as well as newer communication protocol.

Even with NoC, latency increases. This is due to latency associated with:

- the interaction of the NoC shells and bus interconnects.
- memory subsystems, including external memory controllers.
- internal network contention.

With the bus interconnect, once the master has control of the bus, the latency is the wire speed [3]. Increasing interconnect latencies have a negative effect on the system performance, especially for the memory-oriented microprocessors architectures [4], [14].

1.2 Problem statement

The network adaptation layer implements the interface of NoC to the IPs. The purpose of the adaptation layer is, among others, to:

- perform protocol conversions, enables re-use and compatibility of existing IPs.
- clock domain crossing, enables the IP to operate in a different clock domain.

The ability to support multiple protocols is essential for leveraging the advantages of the NoC.

For this thesis, we analyze the challenges of interconnecting existing IPs using different standards than the ones already available and explore options for lowering effective latency.

1.3 Solutions

To achieve interoperability with existing industry standards, we:

1. add the ability to interoperate with new protocols to the Æthereal NoC.

2. design protocol conversions for the Æthereal NoC

Furthermore, we evaluate the potential advantages of advanced techniques at the transport abstraction layer, which are challenges when the IPs do not support them. This for:

3. tolerating the latency when possible

4. increase in throughput when possible

5. definition of software protocols to facilitate the solutions of 3 and 4

We want to connect to IPs which use the IBM Coreconnect bus cores [18]. The Coreconnect buses are used and supported in a wide range of embedded products, including the Xilinx design suites.

1.4 Overview of this thesis

The remainder of this thesis is organized as follows. In chapter 2 we give an overview of a SoC and describe the IPs and interconnects we use in this thesis. Also, we describe the protocols we use. In chapter 3 we give an overview of the shells and also describe in detail the shell designs. Chapter 4 focuses on the various system-on-chips consisting of the shells and the Æthereal NoC and the software applications used. In chapter 5 we detail and analyze the results obtained from the experiments. In chapter 6 we present a summary of the thesis, our conclusions, and elaborate on the future work, related work and recommendations.

2 Background information

In this section we discuss the concepts of embedded system interconnects and underlying principles of SoC design and prototyping. We give particular attention to the set of technologies that we have used in this study: the CoreConnect bus interconnects, the Æthereal NoC and the Xilinx tools and components.

2.1 System overview

A SoC consists of, among others, IPs and interconnects. The IPs need to communicate with one another and can be connected directly to each other or through an interconnect.

For the case when the IPs are connected directly to each other, one IP is the initiator and always transmits. The other IP is the target and it always receives. A common scheme to move data, between two IPs connected directly to each other, is a handshake protocol. The initiator uses a request signal to initiate the transfer and the target uses an acknowledge signal to indicate to the initiator that the data was accepted.

For the case when the IPs are connected to each other through an interconnect, the communication is performed through transactions. For example, when the communication is done using bus interconnects, a transaction consists of sending the address and receiving or sending data. More specifically, we distinguish between a read and a write transaction, for example, between a processor and a memory. A read transaction transfers data from memory to the processor and a write transaction transfers data from the processor to the memory. Furthermore, we define masters as IPs that can initiate read or write transactions and slaves as IPs that are the target of read or write transactions. It is possible to have multiple masters and multiple slaves. Figure 2.1 shows an example of such a SoC consisting of multiple masters and slaves.

The IPs are grouped in subsystems or tiles. The two main subsystems are the master tile and the slave tile. The master tile contains the processing element (section 2.2.1) and the slave tile contains the storage element (section 2.2) and peripherals. For this example, the master tiles contain the μBlaze as the processing element together with optional components, like the local instruction and data memory and caches. The example slave tiles consists of on-chip memory, external memory controller to access off-chip memory and a peripheral.

2.1.1 Communication types

We distinguish two communication types. They are as follows.

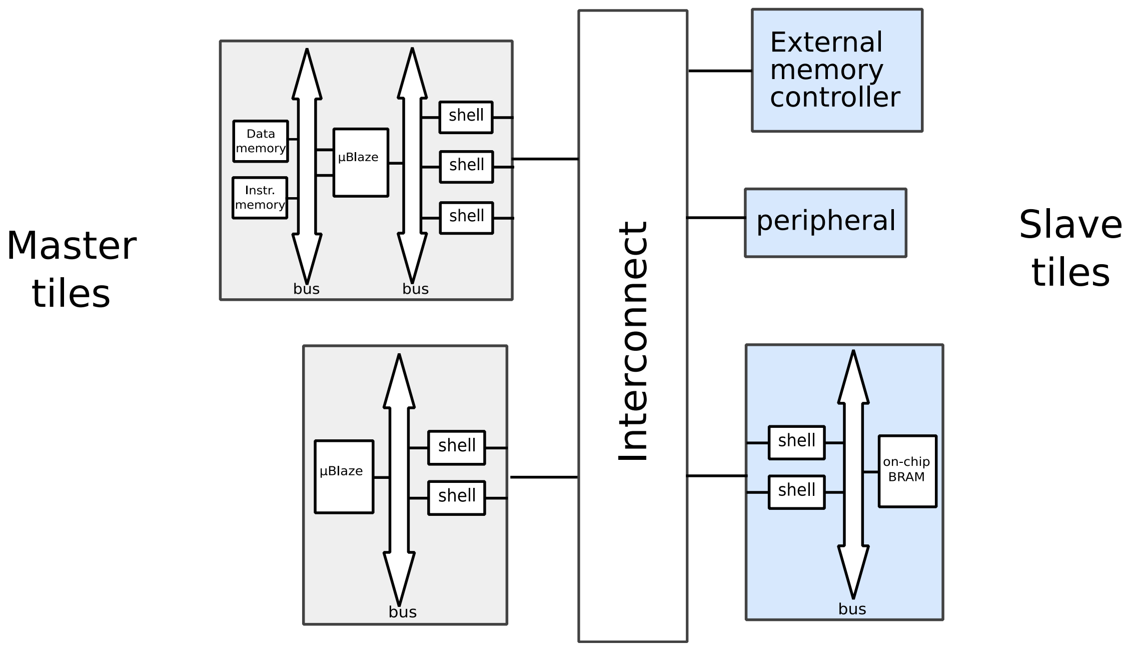


Figure 2.1: System overview example

- Direct streaming: one way communication from master to slave, requires only writing. It uses streaming protocol.

- Shared-memory communication [14], [20]: Communication occurs through a shared address space. This type enables more complex communication between multiple masters and multiple slaves attached to the interconnect. It requires both writing and reading. The master IP, in this case the processor, uses a bus-based I/O interface to communicate with peripherals and memory, by reading or writing to an address. There are 2 addressing schemes to accomplish this, known as memory-mapped I/O and standard I/O [14], [28]. In memory-mapped I/O, each peripheral occupies a specific address in the existing address space. In standard I/O the processor uses an additional signal to indicate whether the access is to memory or to a peripheral.

  Signal groups are used to transfer data between the IPs and the interconnect. If the logical address space is distributed among several memories, the communication type is referred to as distributed-shared-memory communication.

2.1.2 Multiprocessor system

Multiprocessor system, like the one of Figure 2.1, can be categorized, according to the parallelism in the instruction and data streams, as follows.

- SISD: single instruction stream, single data stream: a uniprocessor executes a single instruction stream using data available in a single memory

- SIMD: single instruction stream, multiple data streams. The same instruction is executed by multiple processors, using different data streams. Each processor has a local data.
- MISD: multiple instruction streams, single data stream. Each functional unit performs different operations on the same data set.
- MIMD: multiple instruction streams, multiple data streams: Each processor fetches its own instruction and performs operation on its own data

Each of the models have techniques for performing inter processor communication and for synchronization. An in depth coverage and analyses is provided in [14] and [20]. For this thesis we focus on a SISD model for experimentation.

2.1.3 Layered interconnect

The interconnect of Figure 2.1 is layered, consisting of:

- local buses, for local communication. For example, the bus connecting the μBlaze with its local data or instruction memory.
- global interconnect, for chip wide communication. For example, for the communication between a master tile and a slave tile.

For this thesis, we discuss two types of interconnect:

1. Bus based interconnects.

   The term bus refers to the entire collection of wires used for communication as well as the communication protocols over those wires [28] (section 2.4). These signals may be unidirectional or bidirectional and are generally grouped as follows.

   - Command signal group: these signals include the address, read/write and flags. They are used to indicate the type of the transaction
   - Write signal group: these signals consist of the write data and flags.
   - Read signal group: these signals consist of the read data and flags.

   The bus is a shared medium, which means several components take turns using it.

2. Network-on-chips.

   The NoC packetizes and routes the IPs transaction between source and destination in the network. A network consists of 2 main components.

   - Routers: The routers transport packets through the network. A router receives a packet on one of its input ports and forwards the packet to one of its output ports, which is connected to either another router or to a NI.
   - Network interfaces: The NIs are responsible for both packetizing the transaction of the IP and for un-packetizing the packets and presenting the transaction to the IP. The NIs also implement the connections and the services of the NoC.

2.2 Master and slave tiles

The master tile consists of a processor and its local memory and buses. We use the Xilinx μBlaze processor. The slave tile consists of a memory and its memory controller and buses.

2.2.1 μBlaze processor

The μBlaze embedded processor is a soft core reduced instruction set computer (RISC) and is available for the Xilinx FPGAs [11]. Figure 2.2 shows an overview of the μBlaze core. The μBlaze core is organized as a Harvard architecture with separate bus inter-

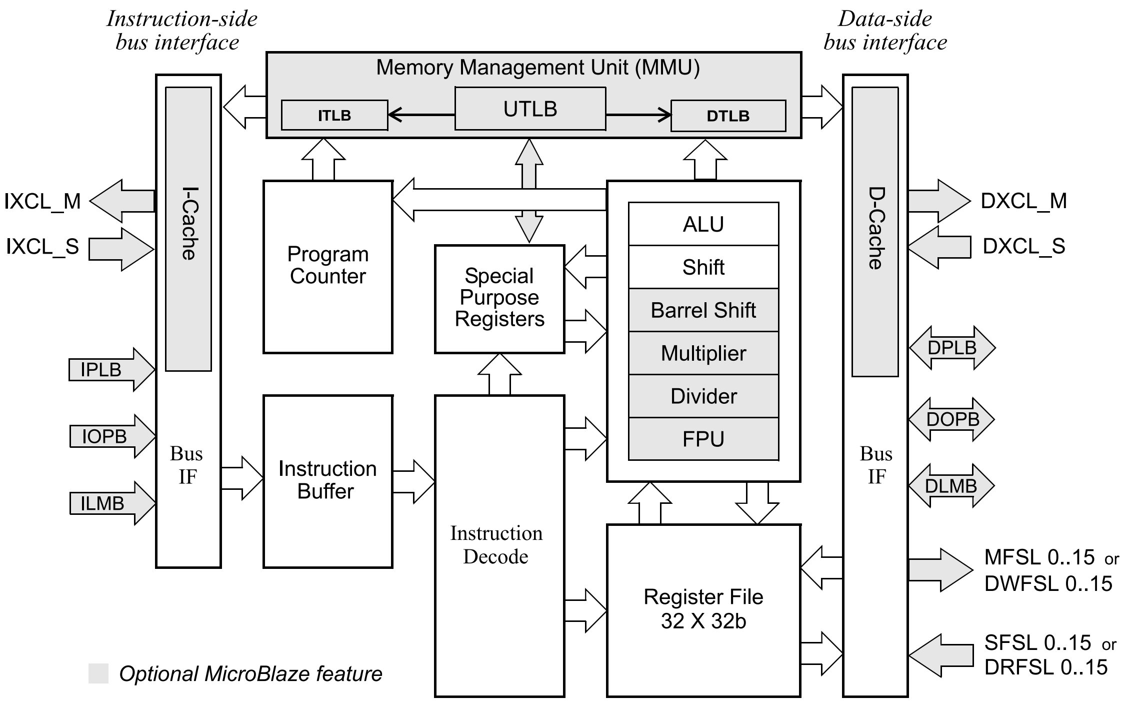


Figure 2.2: μBlaze core overview [11]

face units for data and instruction accesses. The following three memory interfaces are supported:

- Local Memory Bus (LMB) (Section 2.3.4), provides single-cycle access to on-chip dual-port block RAM.
- PLB or OPB (Sections 2.3.3 and 2.3.3), provide a connection to on-chip or off-chip peripherals and memory.
- Xilinx Cache Link (XCL), intended for use with specialized external memory controllers. The XCL is not used in this thesis.

Furthermore, The μBlaze uses the memory-mapped I/O addressing scheme.

2.2.2 On-chip memory

We use the on chip block RAM (BRAM) of the Xilinx FPGA. We access the on-chip memory through the Xilinx memory controller [31]. The memory controller supports memory sizes in the range of 8k to 256k bytes.

2.3 Bus based interconnect

The PLB and OPB bus are part of CoreConnect architecture bus, an IBM-developed on-chip bus-communications link [18]. The PLB is used to connect fast components to the CPU, while the OPB bus is used for slow peripherals. The slaves attached to the PLB and OPB bus are assigned a certain address range. They have to recognize their address on the bus. The following subsections present a short overview of the OPB, PLB and FSL bus.

2.3.1 OPB bus

The OPB bus is a non-split, non-burst bus. It supports up to 16 masters and 16 slaves. The OPB uses a single synchronous handshake protocol for the entire bus transaction. There is no separate handshake for the request and response transaction. The master puts the data, address and control signals on the bus and indicates a valid transaction (M1_select, Figure 2.3). Each slave checks the address on the bus. If the address is recognized, the transfer is completed with an acknowledge signal (Sl2_xferAck).

2.3.2 PLB bus

The PLB bus is a concurrent interconnect, it supports split transactions and also burst transactions (table 2.1). It supports a maximum of 16 masters. Each master has a separate address and data signal group Figure 2.4). For the slaves, the response signals are OR'ed together.

A PLB transaction consists of an address cycle and a data cycle. In the address cycle:

- the master drives its address and command groups and request ownership of the bus
- address and command signals are presented to the slave after the master has been granted ownership of the bus
- the slave latches the address and command groups and terminates the cycle with an acknowledgement.

In the data cycle:

- master drives the write data bus for write transactions or samples the read data bus for a read transaction.
- sample or drives the data acknowledgement signals.

Figures 2.5 gives an example of a PLB write transaction.

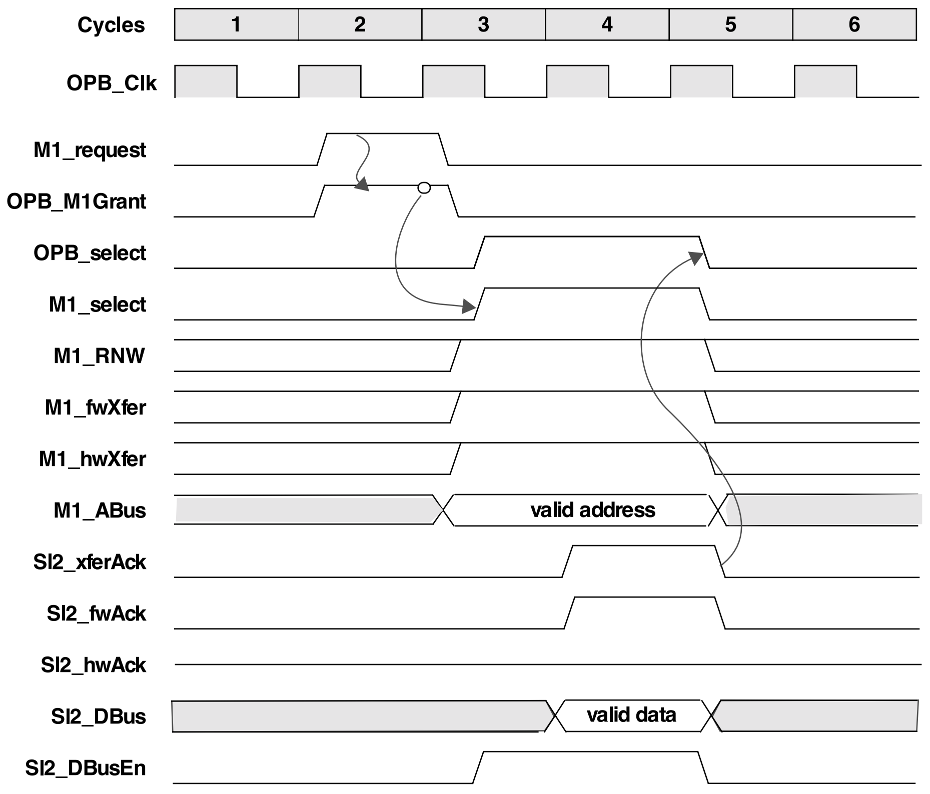


Figure 2.3: OPB basic data transfer [19]

2.3.3 FSL bus

The Fast Simplex Link (FSL) is a uni-directional point-to-point FIFO-based communication channel bus[30]. The link has a master side and a slave side. The master pushes data into the link and the slave consumes data from the link. The width of the FIFO link is 32 bits and the depth is design time configurable. The FSL uses a simple handshake streaming protocol for data transaction. Figure 2.6 details a write transaction [29]. The master pushes data into the link by asserting the "FSL_M_Write" signal. The FSL link is connected to the μBlaze through a dedicated port. The μBlaze supports up to 8 FSL link pairs in opposite directions, each with one master (write) and one slave(read) FSL interface. The μBlaze has special put and get instructions to write and read data to/from the FSL link. There are several variants of the instructions [11]. We use blocking version: the μBlaze will stall until the data from the FSL interface is valid.

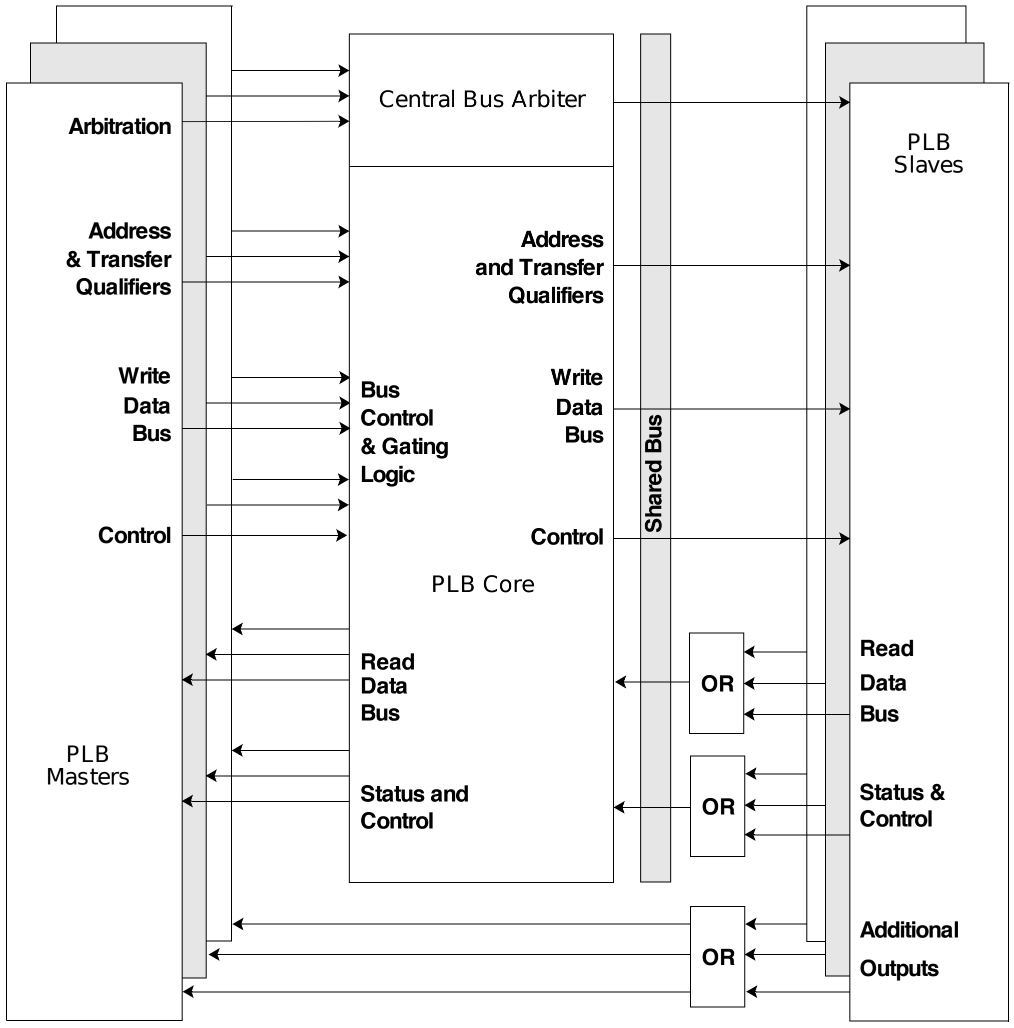


Figure 2.4: PLB bus overview [17]

2.3.4 LMB bus

The LMB bus [32] is the interconnect for the μBlaze instruction and data port with the local BRAM memory through the LMB BRAM interface controller [31]. The LMB bus supports one master, up to sixteen slaves and has no arbiter.

2.4 Protocols overview

To transfer data between two components, a sequence of steps needs to be followed [14]. Protocols describe the rules necessary for communication over an interconnect [28].

This section describes some advanced protocols used for tolerating the transaction latency.

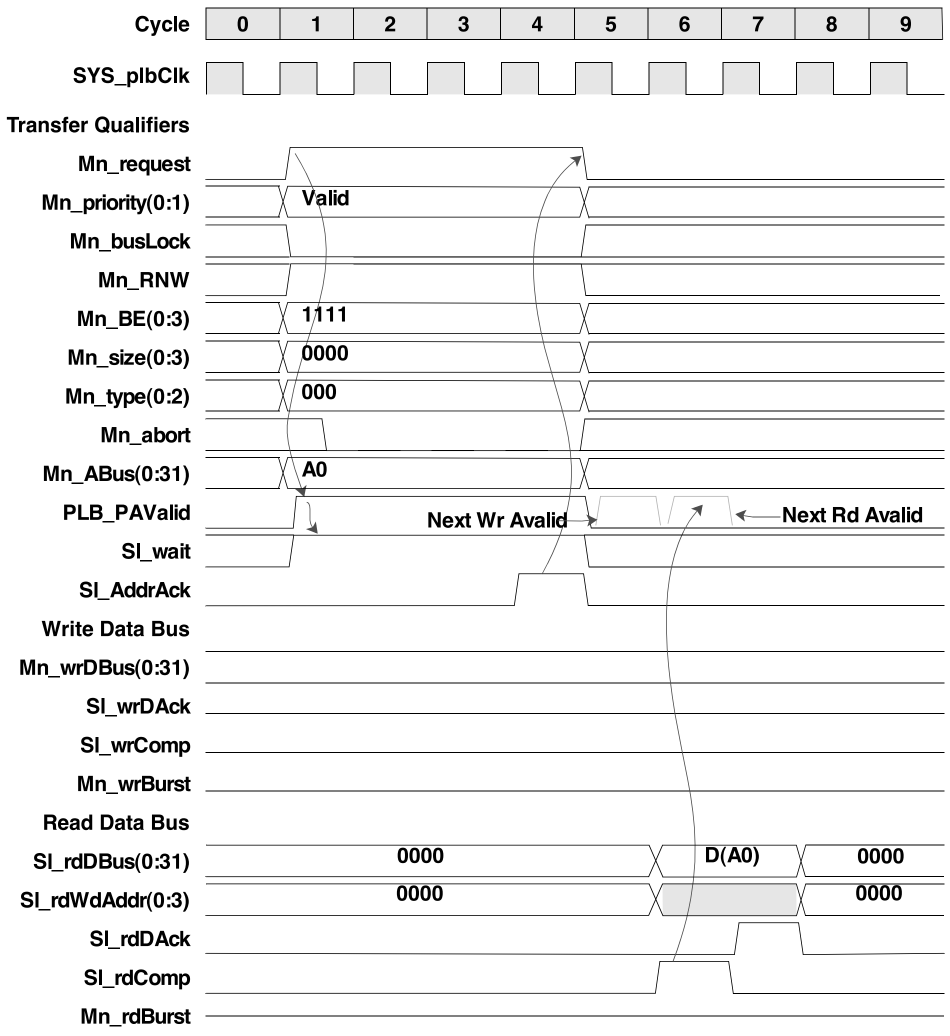


Figure 2.5: PLB write transaction [17]

2.4.1 Posted transaction

In general, the master has to wait for:

- The interconnect to transport the master request to the slave
- The slave to process the request and to acknowledge the transaction
- The interconnect to transports the slave response to the master

These steps are illustrated in figure 2.7.

In a posted or pipelined transaction, there are multiple outstanding requests of a single master. The interconnect acknowledges the transaction before it transports the request to the slave. Figure 2.8 shows a posted transaction. The benefit of this is that

Table 2.1: 64 bit Processor Local Bus (Xilinx)

Specification	
Address bus width	32 bits
Data bus width	64 bits
Architecture	(max 16) Master / (max 16) Slaves
Bus cycles	Read/write Back-to-back read/write Back-to-back read write read write Four-word line read/write Four-line read followed by four-line write Sequential burst-read/write transfer terminated by Master/Slave Fixed length burst read/write Back-to-back burst read - burst writes Locked data transfer Pipelined back-to-back read/write Pipelined back-to-back read and write Pipelined back-to-back read/write burst Pipelined back-to-back fixed length read burst
Arbitration	Central bus arbiter. Three cycle arbitration with four levels of dynamic master request priority. Fixed priority scheme can be implemented
Data bus width	64 bits
Burst transfers	Supported
Split transaction	Supported (separate address and data busses)
Pipelining	Address pipelining
Master/Slave signal interface	Input or output

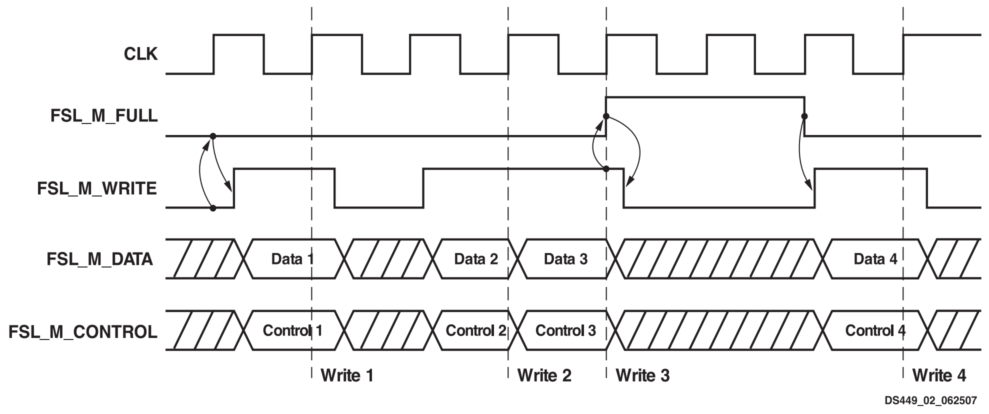


Figure 2.6: FSL write [29]

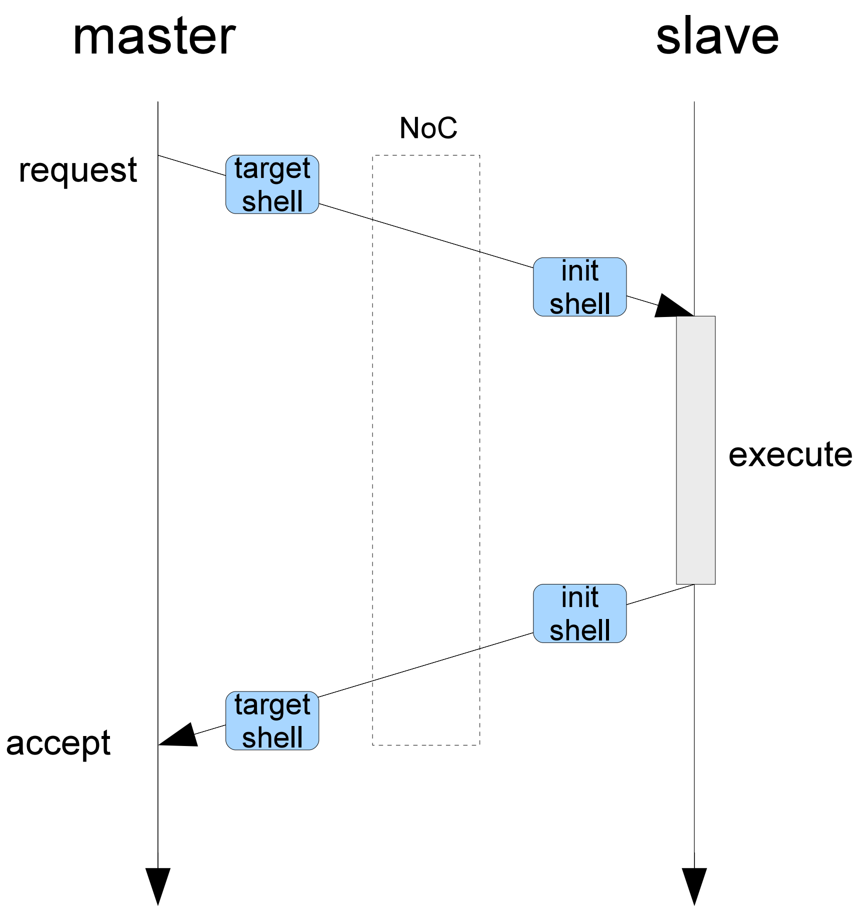


Figure 2.7: Non-posted transaction between master and slave

the master does not have to wait for the slave reply. The drawback of posted write is that the interconnect assumes that the slave can process the request successfully. If the slave cannot process the request, this assumption is not correct anymore and the interconnect, master and slave have to have a mechanism to solve this problem. The actual steps needed to fix the situation depend on the cause of the error. If, for example, the slave was busy and could not accept the request at that moment, the interconnect could wait for the slave to be available again and then retry the transaction.

The posted transaction can be applied to write transactions.

2.4.1.1 Posted write

For the posted write transaction, the interconnect acknowledges the write and then transport the write data to the slave. The slave acknowledge signals can be discarded by the interconnect as they are not needed anymore by the master, except on an error.

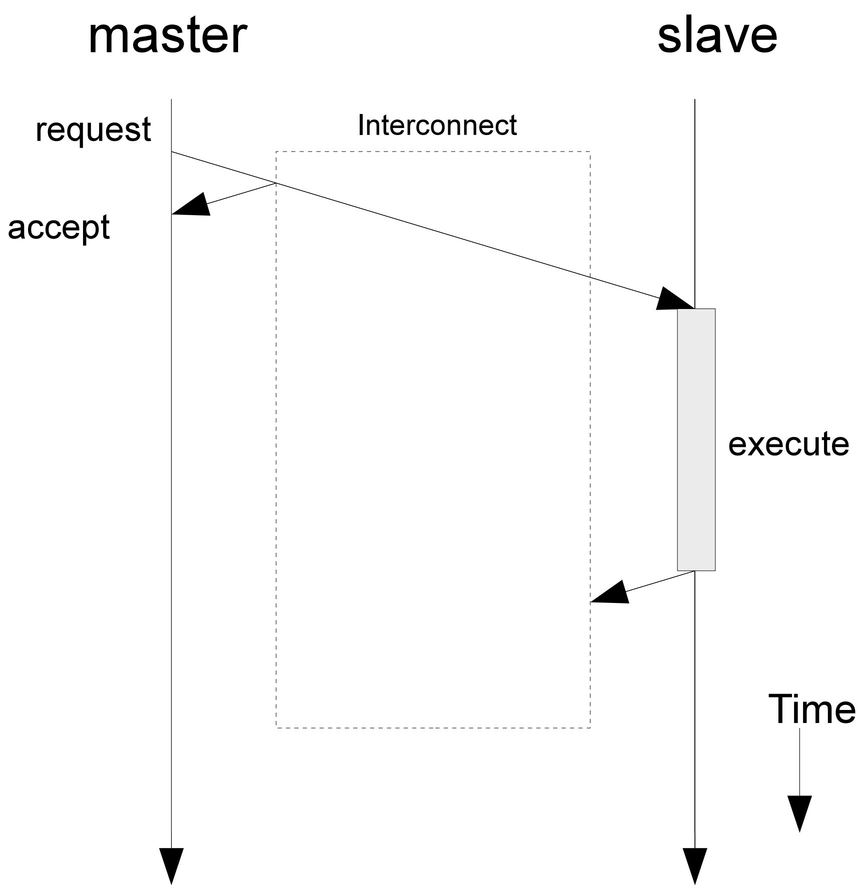


Figure 2.8: Posted transaction between master and slave

2.4.2 Split transaction

During a non-split transaction, between a master and a slave, the interconnect is held by the master. A slow slave will delay the response and hence have a negative effect on overall system performance. A method to avoid this situation is the split-transaction for multiple master systems. Consider the case of a master initiating a memory read to the slave. Between the time of a master request and the slave response, the interconnect is unavailable for other masters. A split-transaction interconnect allows multiple outstanding transfers by decoupling the request phase from the response phase. While the memory is busy reading the requested word, the interconnect is released and it can accept the next master request.

2.4.3 Burst transaction

For the case when there are multiple transactions between a master and a slave. The master can either:

- For each data word to be transferred, arbitrate for access on the bus and send command group data.
- Only arbitrate for the bus once and also send command group data once.

The first is called single transaction and the latter burst transaction. For an interconnect without support for burst transfers, only one word of data can be transferred per transaction. For an interconnect with support for burst transfers, multiple data words can be transferred per transaction. The requirement for a burst transfer is that the data words are located in sequential locations in the slave memory.

2.5 Prefetch read

For a read transaction, the master request data from the slave and must wait on:

- the interconnect to transport the request to the slave
- the slave to execute the request and produce a response
- the interconnect to transport the response to the master

A workaround for this is to do the request in advance, before the master actually needs it. To achieve this prefetch of data, a split read is used. First the address containing the read data location is acknowledged. After that the actual read data is acknowledged, when it is consumed by the processor. Figure 2.9 shows a prefetch read transaction.

Prefetching can be as classified as follows.

- hardware prefetching, the hardware performs and manages prefetches from memory. This implementation is transparent to the software.
- software prefetch: the user or compiler inserts instruction in the code to perform prefetches. The hardware provides support for this.
- Hardware/software prefetching: a combination of the above.

For this thesis, we use software prefetching. To support the read prefetch, we use a DMA transfer [28]: the processor signals to the DMA unit that it needs data from memory. The DMA unit is responsible for fetching the requested word from memory. Instead of using interrupts for signaling to the processor that the data has arrived, we leave it up to the processor to poll for data.

In a multiprocessor system, it is possible that the prefetch read data is updated after it was retrieved. This issue is important, as the prefetch read must not change the result of the computation [14]. A prefetch read can be either non-binding or binding.

- Nonbinding: the data is kept coherent. On read data access, the latest value written is loaded.
- Binding: the data is not kept coherent. On read data access, the value retrieved is loaded.

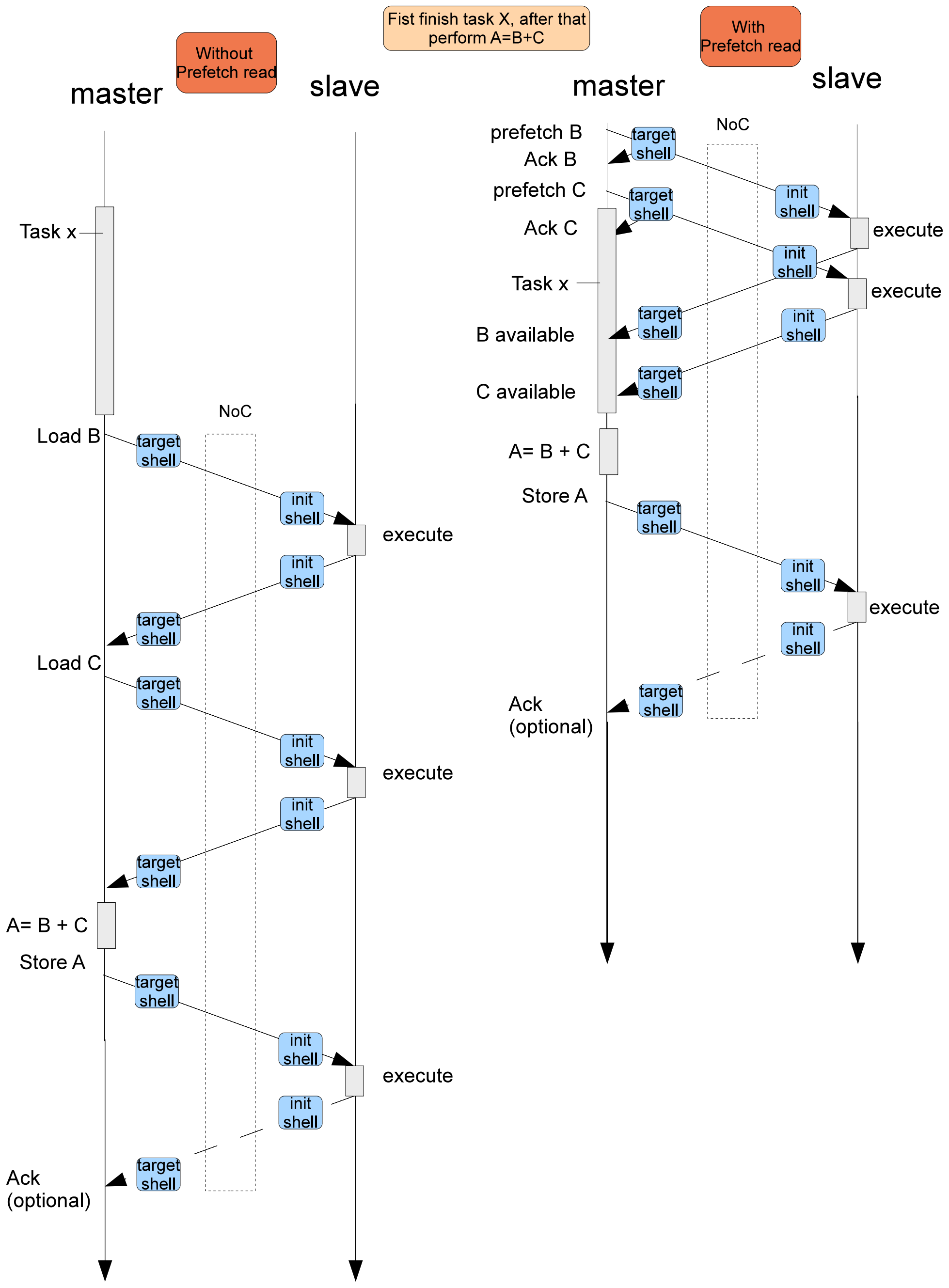


Figure 2.9: prefetch vs. non-prefetch read transaction

Nonbinding prefetch is essential in multiprocessors system, but it is more difficult to implement than binding prefetch. We use nonbinding prefetch as it is easier to implement. Therefore we must make sure that the data fetched from the prefetch unit is correct.

2.6 The Æthereal Network-on-Chip

The Æthereal network on chip consists of routers and network interfaces (NI), and tools to build and configure them [7]. Figure 2.10 shows an overview of the NoC.
The NI is split in two parts:

1. NI kernel, which implements flow control, schedule packets for sending to routers and implements connections.

2. NI shell, interfaces to the IPs.

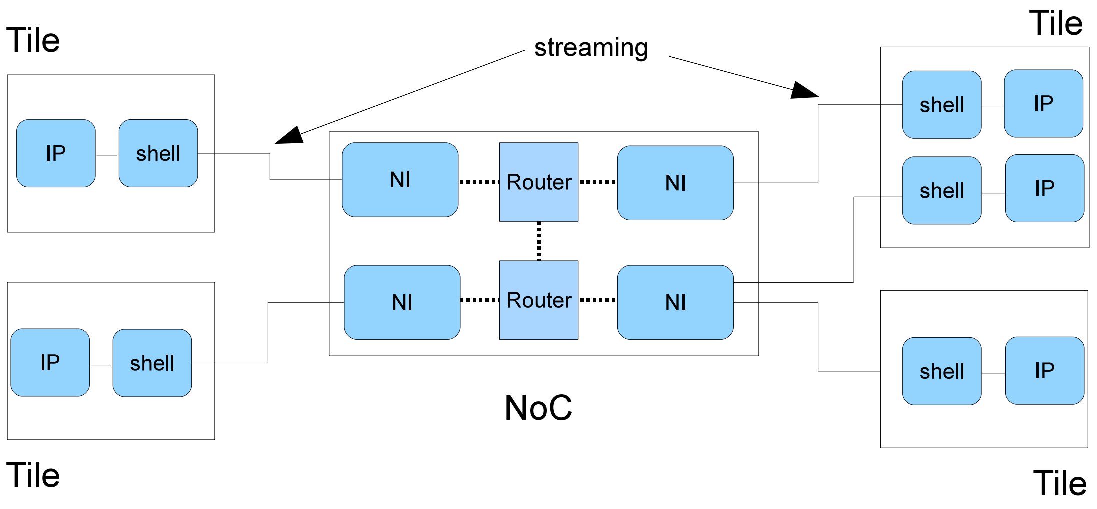


Figure 2.10: overview of the NoC

The main reason to split the NI into a shell and a kernel is for flexibility [25]. This way the shell can be re-used with different NI kernel versions. Also, to offer support for a new bus protocol, it is only necessary to implement a new shell.

The network provides two classes of quality of services (QoS) :

1. Guaranteed service (GS), channels use time division multiplexing circuit switching approach. Circuits are set up by reserving consecutive slots in the slot table. We use this QoS.

2. Best effort (BE), channels use buffered (worm-hole) flow control with input buffering. BE uses unused slots of the GS to transport data. We do not use this QoS

The following section details the functionality of the NI shell, the main topic of this thesis. Further details of the routers and the NI architectures are not described here, as they are documented in the specific literature [8], [25].

2.6.1 NI shell and messages

IPs use the bus for communication with other components (section 2.3). They interface with each other using memory mapped I/O. The exact communication details depend on the bus protocol being used. The NoC, on the other hand, serializes the bus transaction into messages. The messages are on streaming protocol and are independent of the bus type. The functionality of a NI shell is to convert bus protocols into NoC messages and vice versa. The presence of the NoC should be transparent to the IP. This way an IP can be plugged into a network without any modification. The NI shells interface to the IPs via a bus. As a transaction involves a master and a slave IP, the NI shells are located both at the master side and at the slave side. The functionality of the master shell is to serialize the master IP read or write requests, consisting of commands groups, address and optionally write data, into request messages. The master shell also has to de-serialize the response messages into IP read or write responses. Figure 2.11 illustrates an example of the master shell. At the slave side, the opposite operations occur. The

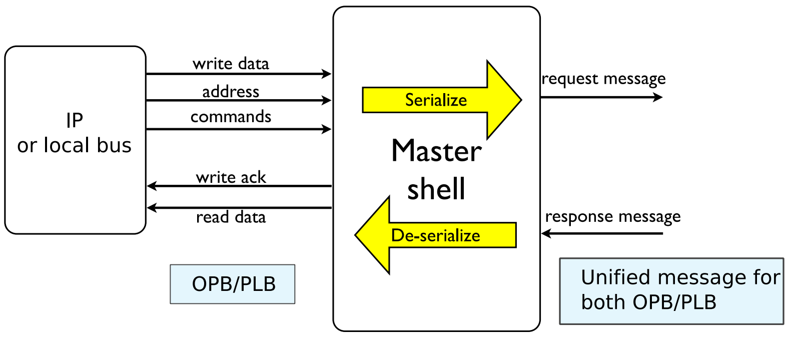


Figure 2.11: Master shell example

functionality of the slave shell is to de-serialize the request messages into the slave IP read or write requests. It also has to serialize the slave IP read or write responses, consisting of command groups, data and address, into response messages. Figure 2.12 illustrates an example of the slave shell. The messages consist of headers and payload.

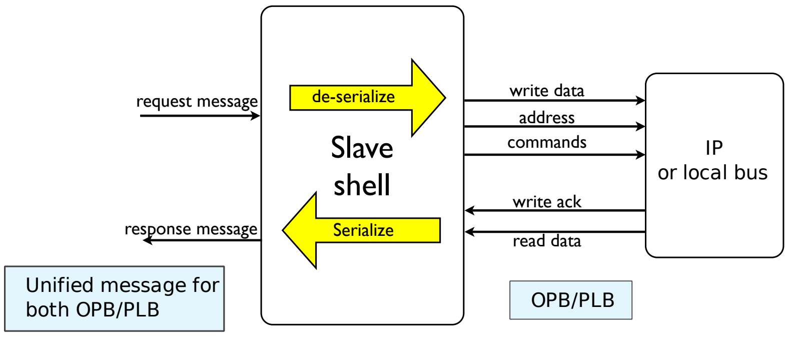


Figure 2.12: Slave shell example

The header contains the message type, the slave address and the length of the message. The payload consists of the actual data, byte enable and end of message (EOM). The whole message format is illustrated in figure 2.13. The actual message that is sent or received depends on the message type:

- write request messages consist of headers A,B and the data field(s).

- read request messages consist of headers A and B.
- write response messages consist of header A
- read response messages consist of header A and the data field(s)

Each message has an EOM bit. The messages are transferred to the NI kernel input queue with a simple handshake streaming protocol. The protocol consists of valid, accept and data lines. Data (the actual message) is only transferred between the shell and the kernel when both the valid and accept are valid on the rising edge of the clock signal.

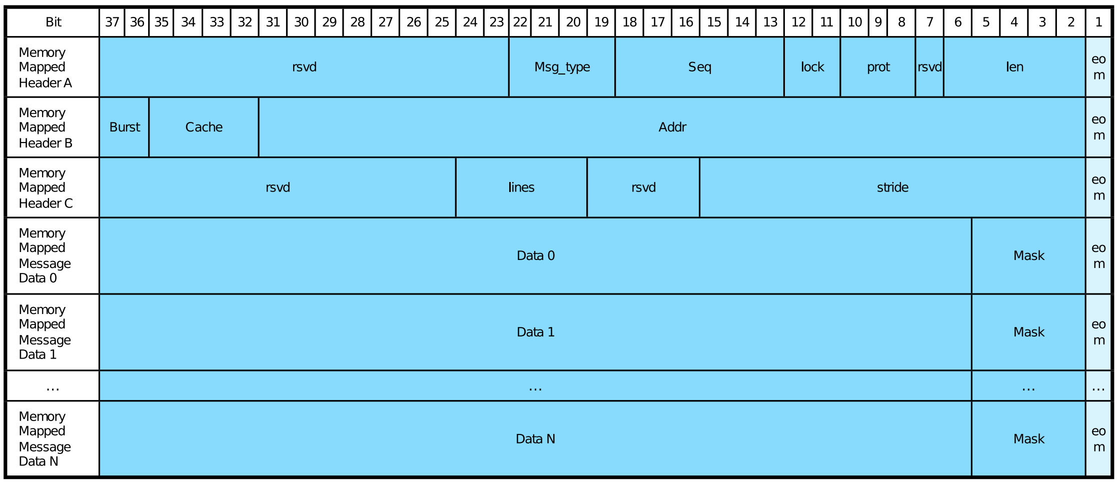


Figure 2.13: Æthereal message format

2.7 Æthereal runtime configuration

The NoC has to be configured before it is used. The configuration process consists of setting up connections between the NIs. This is accomplished by writing to the registers of the NIs. The registers of the NIs are accessible from the configuration port of the host NI. This way the configuration of the NoC is done through the network itself, there is no separate control network. The configuration operations are executed by (at least) one configuration master. The steps needed to configure a network for GT are as follows. First a connection is opened from the configuring NI (NIc) to the target NI, by writing messages to the NIc to:

- set up the flow control by initializing the remote space counter
- initialize the slot table
- set up the channel type, remote queue id and path from NIc to NIt

Next, a connection from the target NI (NIt) is opened to the NIc by writing messages to the NIt to:

- set up the channel type, remote queue id and path from NIt to NIc
- set up the flow control by initializing the remote space counter
- set up the slot table

Last, the NIc closes the configuration request connection to the NIt by:

- reading the remote space until all credits are returned

The NoC configuration process is detailed in [13].

2.8 NoC design tool flow

The Æthereal NoC consists of programs to generate the network topology and configurations files in SystemC, VHDL and C [13]. This section details the process to generate a NoC in VHDL from the tools provided.

2.8.1 NoC setup

The setup consists of specifying the NoC parameters. Some relevant parameters are as follows.

- Flow mode: the Unified MApping, Routing and Slot allocation (UMARS) is used. It is an algorithm for the mapping of cores and the routing of communication between the cores.
- Topology: specify the type of topology. The topology used is mesh topology. Also the number of NI per router can be specified here.
- Configuration service class: specify whether the network is configured using GT or BE. The default is GT.

Furthermore, some network specific parameters need to be set. They are specified in generated XML files or text files. The following subsections give an overview of the various options.

2.8.1.1 Architecture

The architecture parameters allow customizing, among others, the slot table size. We allocate a certain link bandwidth by varying the number of slots in the slot table. Apart from the slot table, all the ports connections between the shell and the NI kernel are specified in this section.

2.8.1.2 Communication

In this section, the usecase can be specified. The parameters of the connections include:

- QoS: we can specify whether we use GT of BE. We use GT
- Initiator and target port for the connection.
- Bandwidth and latency requirements. The tool computes the number of slots in the slot table from the information provided by the bandwidth and latency requirements.

Instead of using the generated slot table, we customize it according to the link bandwidth we want to use. Furthermore, we use the usecases to differentiate between the various link bandwidth setups we use for the experiments.

2.9 FPGA design flow

We use tools and components provided by Xilinx to build a System-on-Chip. We design the embedded system using the Xilinx Embedded Development Kit (EDK).

2.9.1 EDK

The EDK is an integrated software platform for designing embedded processing systems based on the Xilinx FPGAs. It contains descriptions of Xilinx and 3rd party boards. Different types of components are provided by libraries, like busses, controllers and processors. It is also possible to add custom components to the library, which can be imported and used in the embedded system. The EDK includes tools for writing and uploading applications to the embedded system. By default, the EDK compiles and uploads the application in the block RAM (BRAM) of the embedded processor. After the embedded system is designed, it is possible to simulate the whole design in a simulator. The EDK interacts with the simulator by generating simulation VHDL files and simulator compile scripts. These files are then used by the simulator. The supported simulators are ModelSim, NcSim and the ISE simulator. The EDK can automatically generate the addresses for the peripherals. It is also possible to give each peripheral a custom address.

For the hardware generation, we use the EDK to import the generated Æthereal NoC. After that we build the system by connecting the components. The last step is to specify the addresses of the shells and the memory on the local bus. For the software generation, we use the tools provided by the EDK to create software applications for the μBlaze. The software tools include compilers and linkers. When the whole design is ready, it is synthesized with Xilinx Synthesis Technology (XST).

2.9.2 Digilent XUPV2P board

Our design is implemented on the XUP Virtex-II Pro Development System, developed by Xilinx and manufactured by Digilent. The XUPV2P board provides a hardware platform that consists of a Virtex-II Pro FPGA surrounded by a collection of peripheral components that can be used to create a system [27]. The XUPV2P board is supported by the EDK. Figure 2.14 shows a block diagram of the board. After the bit stream is generated with the EDK tool, Xilinx impact application is used to upload the bit stream into the FPGA.

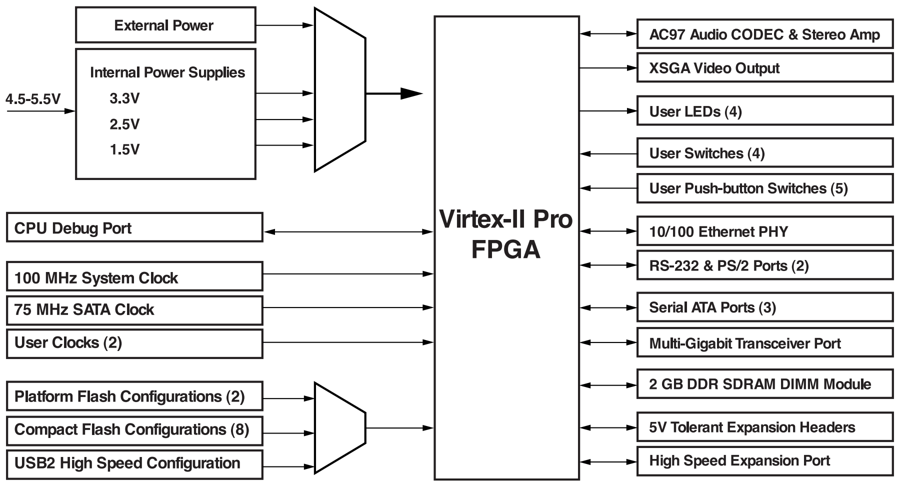

External Power
Internal Power Supplies
4.5-5.5V
3.3V
2.5V
1.5V
CPU Debug Port
100 MHz System Clock
75 MHz SATA Clock
User Clocks (2)
Platform Flash Configurations (2)
Compact Flash Configurations (8)
USB2 High Speed Configuration
Virtex-II Pro FPGA
AC97 Audio CODEC & Stereo Amp
XSGA Video Output
User LEDs (4)
User Switches (4)
User Push-button Switches (5)
10/100 Ethernet PHY
RS-232 & PS/2 Ports (2)
Serial ATA Ports (3)
Multi-Gigabit Transceiver Port
2 GB DDR SDRAM DIMM Module
5V Tolerant Expansion Headers
High Speed Expansion Port

Figure 2.14: Block diagram of the XUPV2P board

3 Design of the NI shells

This section details the design of the Æthereal OPB/PLB/FSL shells. The aim is to design a shell which provides interoperability and high performance i.e. hide the transaction latency and increase the data throughput when possible. All the shells have a pair of streaming ports, i.e. one for the read/write request and one for the read/write response. Therefore, the shells support one connection between an initiator and a target. We do not consider supporting separate read and write connection in the shells, as for the OPB and PLB, the command group for read and write is shared. For the shell designs, we choose an iterative design methodology. We first implement shells that support the basic functionality. Only there after we implement more advanced features. The shells are designed using VHDL.

3.1 Design methodology

We start by designing a shell for a bus with a simple protocol, the OPB bus. The OPB shell does not actually fit the idea of high performance shell, because the OPB bus itself is not suitable for interconnecting high performance components. The next step is to use a high performance bus, the PLB bus. The PLB bus has a more complex protocol than the OPB bus and it supports more advanced features. We convert the fully tested and functioning OPB shell into a PLB shell. The last step in achieving the high performance shell is to modify the resulting PLB shell to include optimizations. To further gain more performance, we use a dedicated bus, the FSL link, to support specific transactions. The high performance shells thus consist of the PLB shells and the FSL shell.

This design approach is necessary because our shells are components inside a larger system. Starting with designing a fully optimized shell supporting various features is a daunting task and error prone.

For the actual shell designs, we use a modular approach when possible. This has the advantage that each separate module can be fully tested and optimized without compromising the integrity of the whole shell. We use the modular design approach extensively for the design of the optimized PLB shell.

3.1.1 Test strategy

We develop a test strategy in order to:

- detect and fix design and logic errors. These include erroneous transitions in the state machines, incorrect output of components, etc.

- verify the functionality of the shells. These include shell/kernel timing discrepancies, message format mismatch, etc.

We use custom made test-benches, software applications and the actual Æthereal NoC for the tests. For each shell, we perform a set of tests. First, we setup a test-bench with the shell as device under test. This in order to verify the internal logic of the shell. After that, we use a setup consisting of the shells, the Æthereal NoC, a μBlaze and a on-chip memory. We test the shells by performing writes to the memory and reads from the memory. We verify that the write and read data are correct.

As these extra components are not part of the shell design, we do not consider them further in this thesis.

3.2 Shells overview

The OPB/FSL/PLB shells share common functionality. They differ in implementation details.

Table 3.1: Shells protocols overview

	PW	NPW	Burst w	rd	Burst rd	Pr rd	Burst pr rd	Split	Err
OPB target	x	√	x	√	x	x	x	x	√
OPB init	-	√	√	√	√	-	-	x	√
PLB target base	√	√	x	√	x	x	x	√	x
PLB target opt	√	x	√	√	x	x	x	√	x
PLB init	-	√	√	√	√	-	-	√	x
FSL shell	x	x	x	x	x	√	√	√	x

Table 3.1 gives an overview of the supported features of the various shells. The abbreviations used in the table are as follows.

- pw: posted write
- npw: non-posted write
- w: write
- rd: read
- pr: prefetch
- Err: error handling
- Split: uses address phase and data phase
- -: not applicable for the shell

- $\surd$: feature supported by the shell
- x: feature not supported by the shell

The next sections detail the design of each shell.

3.3 OPB shells

The OPB shells encode OPB bus protocol into the Æthereal messages and for decoding Æthereal messages into the OPB bus protocol. The OPB shells are designed to be used as the first stepping stone towards the high performance PLB shells. The reasons are:

- The OPB bus has a simple protocol, which is easy to interface to.
- We have the freedom to experiment with design concepts that we will not necessarily implement in the high performance shells. Example of this is the error handling mechanism.
- We can quickly develop a prototype shell to experiment with the NoC. This enables us to gain insight into the generation process of the NoC and to familiarize with the development kit.

Because of these considerations, the shell design is kept as simple as possible. Two OPB shell versions exist:

- OPB target, responsible for serializing the OPB bus request protocol into the Æthereal request message format and for de-serializing the Æthereal response message into the OPB bus response protocol. The OPB target is a slave on the OPB bus in the master tile.
- OPB initiator, responsible for de-serializing the Æthereal request message into the OPB bus request protocol and for serializing the OPB bus response into the Æthereal response message format. The OPB initiator is a master on the OPB bus in the slave tile.

3.3.1 OPB target shell

Figure 3.1 shows a diagram of the OPB target shell interface to the OPB bus. Not all of the signals shown are used. This due to the fact that some optional signals and transfer signals are not supported by the target shell or the NoC message format.

3.3.1.1 OPB target shell unused signals

The unused signals are as follows

- **Sln_hwAck, Sln_fwAck, Sln_dwAck** and **OPB_hwXfer, OPB_fwXfer, OPB_dwXfer**: OPB Transfer Size Acknowledge signals. These signals are used to implement Dynamic Bus Sizing on the OPB [19]. There are no corresponding fields in the NoC message format to support this feature.

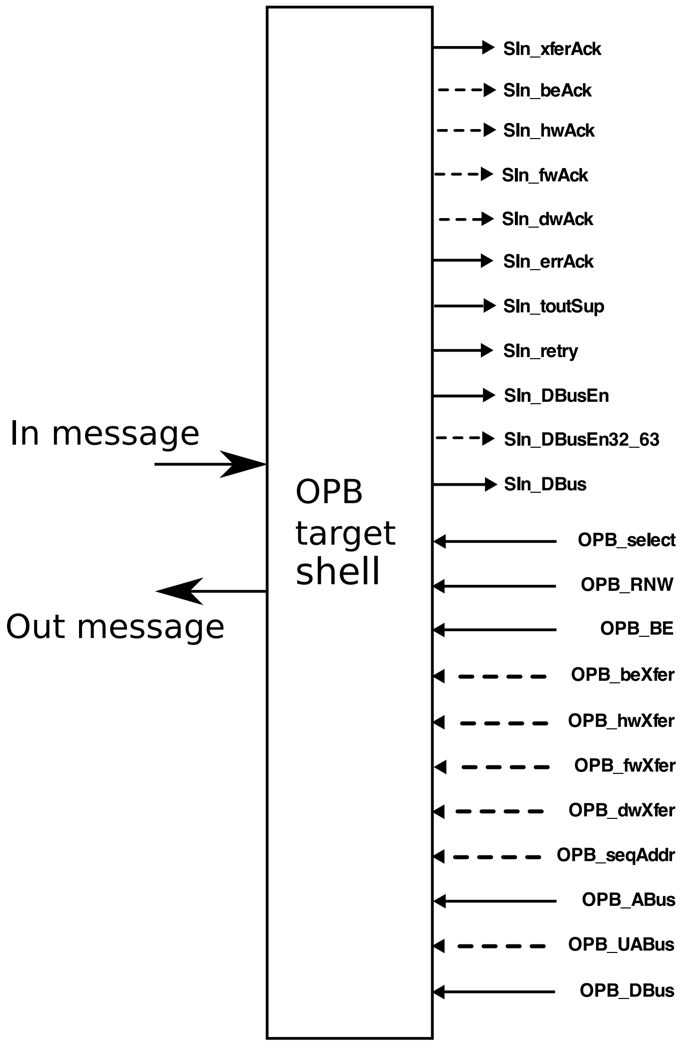


Figure 3.1: OPB target shell interface overview [19]. Dashed lines indicate unused signals

- **Sln_beAck, OPB_beXfer**: byte enable transfer request signals. We use the OPB_BE signals instead.

- **Sln_DBusEn32_63**: signal to enable a 64-bit slave devices data onto the OPB data bus(32:63) during read transfers. We instantiated the Æthereal NoC with a word width of 32 bits.

- **OPB_UABus(0:31)**: signals are used to form the most significant portion of a 64-bit address. We instantiated the Æthereal NoC with a word width of 32 bits.

- **OPB_seqAddr**: signal is used to reduce the latency for sequential addresses access, by allowing the slave to bypass the address decode cycle. This feature is not supported by the prototype target shell.

3.3.1.2 OPB target shell signals mapping

The OPB protocol request signals are mapped to the Æthereal message format. Figures 3.2 and 3.3 show the write request message and the read request message respectively, produced by the OPB target shell.

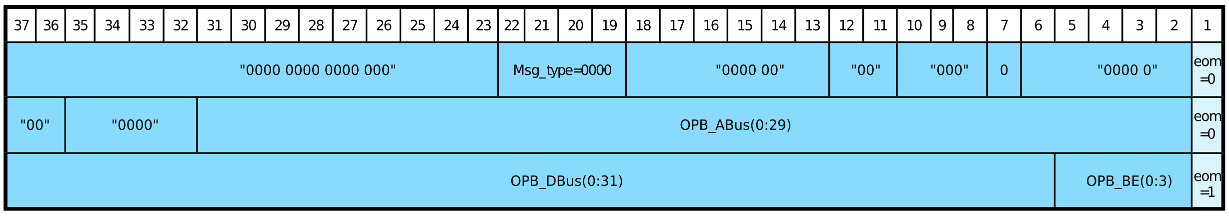


Figure 3.2: Write request message

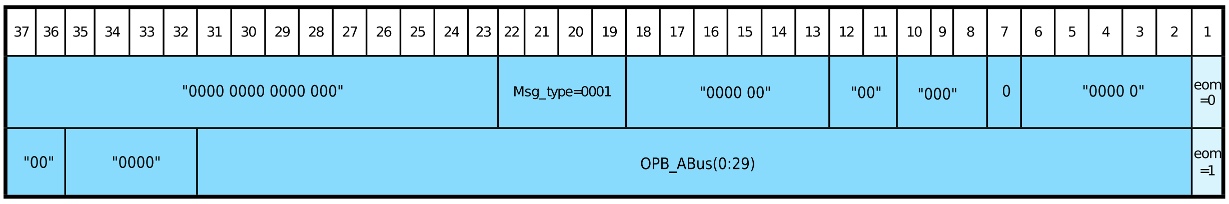


Figure 3.3: Read request message

The OPB bus signals used are as follows.

- **OPB_ABus(0:29)** $\mapsto$ read/write request **Header B(1:30)** address field: contains the address of the transaction. The 2 LSB bits of the address are omitted, due to the fact that the memory is word addressable.

- **OPB_DBus(0:31)** $\mapsto$ write request **Message data(5:36)** data field: contains the transaction write data.

- **OPB_BE(0:3)** $\mapsto$ write request **Message data(1:4)** mask field: contains the master byte enable signals of the transaction.

The rest of the signals are used for handshaking with the OPB bus.

3.3.1.3 OPB target shell overview

Figure 3.4 shows an overview of the OPB target shell. It consists of:

- FSM, to handle the handshake between the target and the shell and for loading data into the shift registers

- address decoder, to recognize the address on the bus.

- shift registers, for handling the handshake between the target and the NIK.

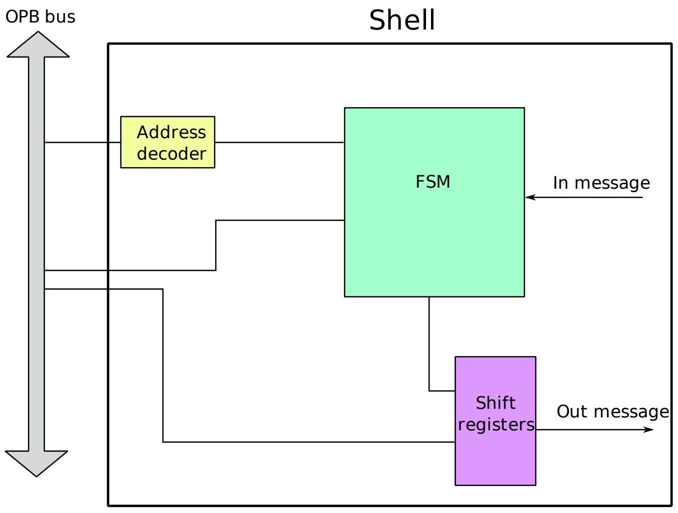


Figure 3.4: OPB target shell overview

Figure 3.5 shows the state diagram of the FSM. The OPB bus performs the address and data transaction in one handshake. When there is a valid transaction on the bus, the transaction command flags and write data are ready to be sampled. The FSM enables the shift register, resulting in the messages being loaded into the shift registers. As

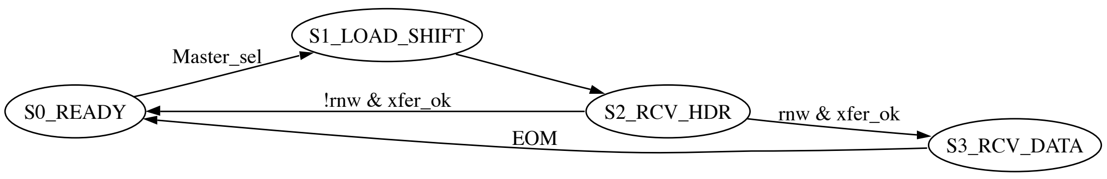


Figure 3.5: OPB target shell state diagram

the OPB bus does not support posted transactions, the FSM waits for the acknowledge message to return and acknowledges the transaction to the bus. If the transaction results in an error, the target shell raises the error acknowledge signal.

Figure 3.6 shows the shift register interface. When enabled, the shift register is loaded with the request messages, consisting of the serialized command signals, flags and write data, if the transaction was a write. After the shift register is loaded, it starts shifting the messages out to the NIK. The input of the shift register is a reshuffle of the OPB bus signals, with additional signals to complete the message format.

3.3.2 OPB initiator shell

Figure 3.7 shows an overview of the OPB initiator interface. As with the OPB target, not all signals are used. This due to the fact that some optional signals and transfer signals are not supported by the initiator shell or the NoC message format.

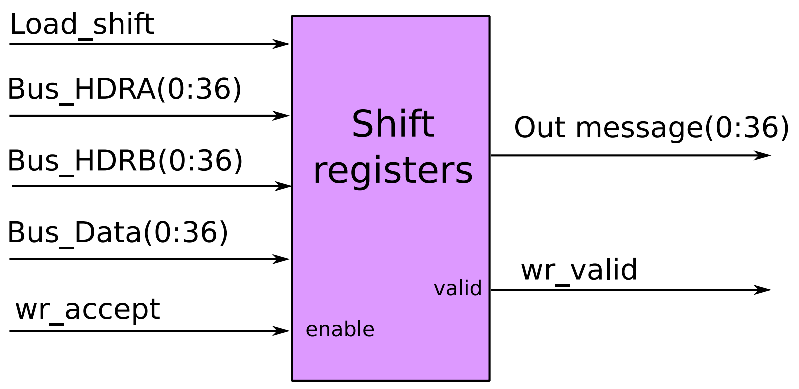


Figure 3.6: OPB target shell shift register interface

3.3.2.1 OPB initiator shell unused signals

The unused signals are as follows

- **Mn_hwXfer, Mn_fwXfer, Mn_dwXfer** and **OPB_hwAck, OPB_fwAck, OPB_dwAck**: OPB Transfer Size Acknowledge signals. These signals are used to implement Dynamic Bus Sizing on the OPB [19]. There are no corresponding fields in the NoC message format to support this feature.
- **OPB_beAck, Mn_beXfer**: byte enable transfer request signals. We use the Mn_BE signals instead.
- **Mn_DBusEn32_63**: signal to enable a 64-bit master devices data onto the OPB data bus(32:63) during write transfers. We instantiated the Æthereal NoC with a word width of 32 bits.
- **Mn_UABus(0:31)**: signals are used to form the most significant portion of a 64-bit address. We instantiated the Æthereal NoC to support an address width of 32 bits.
- **Mn_seqAddr**: signal is used to indicate to the slave sequential address access. This feature is not supported by the prototype initiator shell.
- **Mn_pendReqn**: signal is used to indicate to the master that other masters attached to the bus are requesting access to the bus. This feature is not supported by the prototype initiator shell.

3.3.2.2 OPB initiator shell signals mapping

The OPB protocol response signals are mapped to the Æthereal message format. Figure 3.8 shows the read response message, produced by the OPB initiator shell.

The OPB bus signals used are as follows.

- **Sln_Dbus(0:31)** $\mapsto$ read request **Message data(5:36)** data field: contains the transaction read data.

The rest of the signals are used for handshaking with the OPB bus.

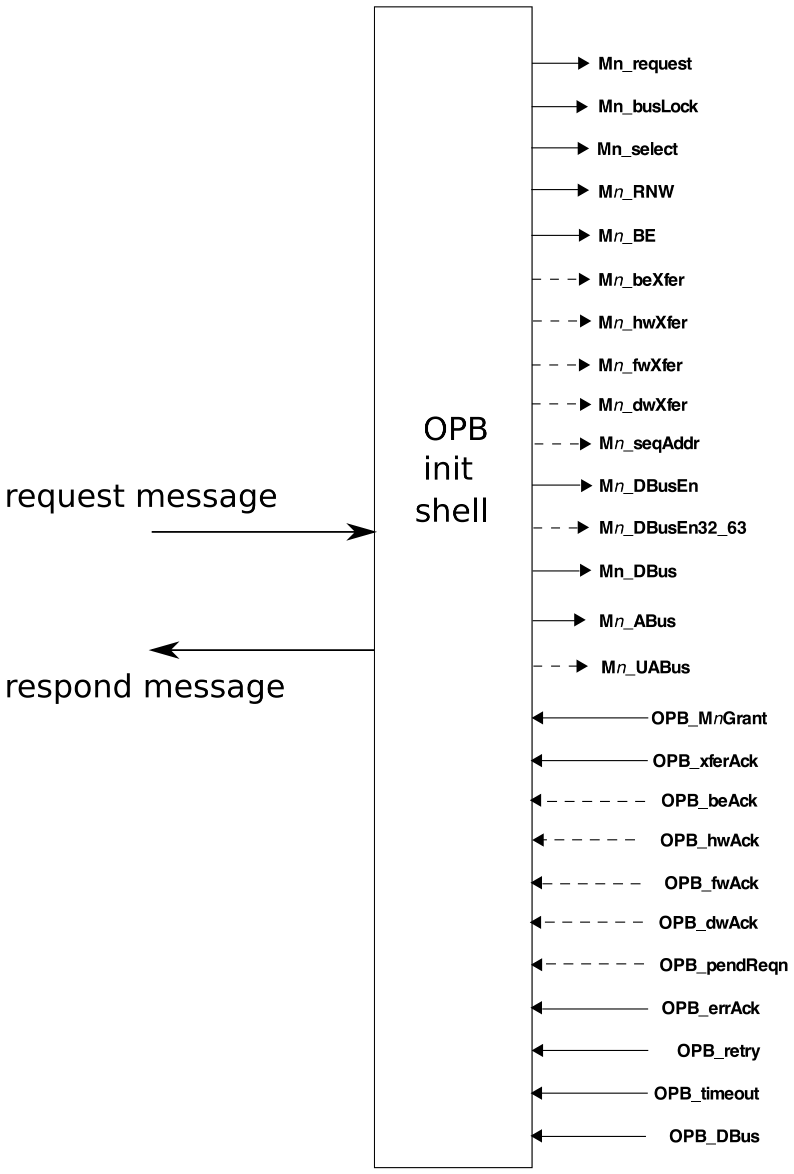
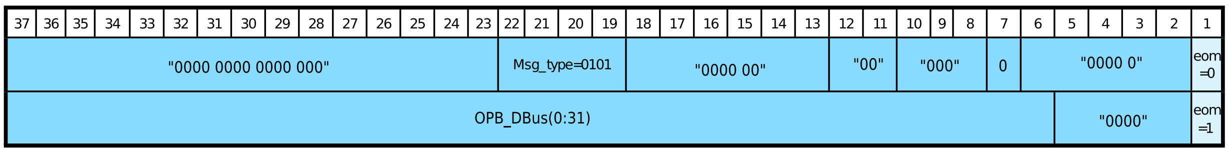


Figure 3.7: OPB initiator shell interface overview [19]. Dashed lines indicate unused signals



37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
"0000 0000 0000 000"															Msg_type=0101				"0000 00"						"00"		"000"			0	"0000 0"					eom =0
OPB_DBus(0:31)																																"0000"				eom =1

Figure 3.8: Read response message

3.3.2.3 OPB initiator shell overview

Figure 3.9 shows a diagram of the shell. It consists of:

- FIFOs, used to store the incoming messages and also for buffering the response messages.
- registers, used to hold the write response headers.
- FSM, controls the processing of the messages and handles the handshake with the OPB bus.

The shell starts processing the request messages only when there is a complete request ready. This enables the initiator shell to complete the transaction request to the bus without stalling, blocking other masters the access to the bus. For experimental purposes and to research burst transactions for the PLB shell, we made it possible for the initiator shell to handle burst transactions. It does this by splitting burst transactions into single transaction, as the OPB bus does not support burst transactions.

The OPB initiator has to take into account that the transaction may result in an error. For this reason, when there is a read transaction, the initiator shell must buffer the whole response and wait until the transaction finishes before it can generate and send the message header. This due to the fact that the message header contains the response type (acknowledge or error). For a burst read transaction, the shell waits until the last data read is fetched from the OPB bus before it starts sending the messages to the NIK. This due to the fact that the initiator shell must take another course of action in case of an error transaction compared with a normal transaction. The error invalidates all the previous data read of the burst read transaction in progress, as the Æthereal message format does not support reporting individual errors in block read response messages. The following steps are performed for a burst read transaction.

- Buffer the read data while burst is in progress
- If no errors occurs, send the acknowledge read header and the read data messages to the target shell. Burst read transaction is complete
- If an error occurs, first purge all the buffered read data from the FIFO, as they are not valid anymore. Then generate an error header and send it to the target shell. Burst read transaction failed.

The error condition puts a constrain on the condition for processing a message. Due to the fact that the send FIFO must be flushed when an error occurs during a burst read transaction, the send FIFO must be empty prior to processing the burst read transaction. The write transaction does not have these limitations, as the outcome of a write is either an acknowledge of an error.

3.3.2.4 Proposals for alternative solutions for error handling

To overcome the limitation imposed in case of an error, we propose three possible solutions. The first solution addresses the problem, while the second and third solutions offer a workaround.

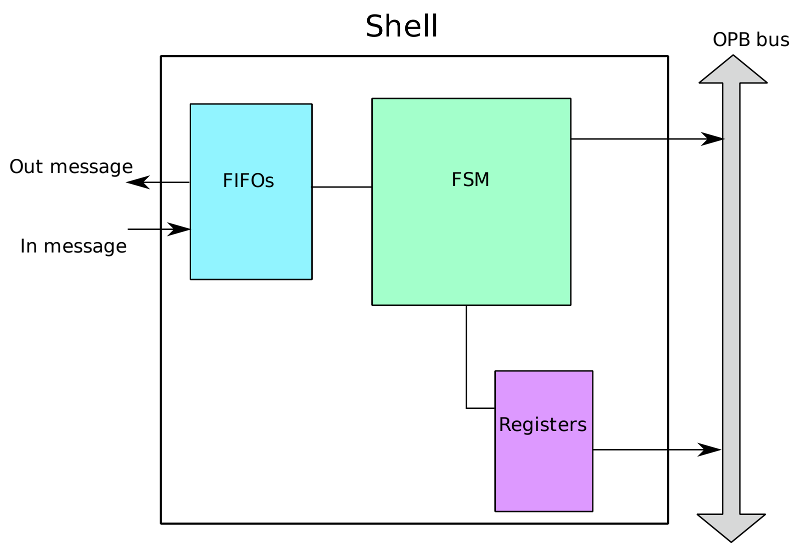


Figure 3.9: OPB initiator shell overview

1. Make the message format more flexible. The main problem with the burst read response message is that either the whole transaction succeeds or fails. Instead of using such a sharp distinction, for example, an extra field to the data messages can be added to indicate whether it is valid or not. This way the individual data messages contain the status instead of the message header. The disadvantage of this approach is that the message width must be adapted. Another approach is to send a footer for the burst read transaction, which contains masks to indicate whether a data message is valid or not. For example, in the footer, a '1' at bit position 'n' indicates that data message 'n' is valid, and so on. The disadvantage of this solution is that an extra message needs to be sent and that the target shell needs additional logic to decode the response messages. Also the target shell must wait until the last message arrives until it is able to process the burst read responses.

2. Split the burst read response message into multiple single read response messages. This workaround avoids the whole issue with the burst response message. There are a couple of disadvantages with this approach. First, the target shell does not expect single read response messages to return. This can lead to undefined behavior if the target shell uses the "len" field of the burst header to process burst transactions. It imposes restriction for all target shell on how to handle burst transactions. Also, extra headers need to be sent to the target shell.

3. Set the mask field to zero. In this case, a zero mask field indicates invalid data. This can also lead to undefined behavior at the target shell, as the mask field is not intended to be used as a status field.

3.4 PLB shells

The PLB shells are shells for encoding PLB bus protocol into Æthereal messages and for decoding Æthereal messages into PLB bus protocol. The PLB shells have two versions:

- PLB target, responsible for serializing the PLB request signals into Æthereal request messages and de-serializing the Æthereal response message into PLB response signals at the master tile.
- PLB initiator, responsible for de-serializing the Æthereal request messages into the PLB request signals and serializing the PLB response signals into Æthereal response messages at the slave tile.

For the target shell, we design two versions. The base design of the first version is the OPB shell. We derive our design out of the OPB target shell and make it suitable for the PLB bus. This way we reuse the working and fully tested shell design. This will provide an initial platform for testing. For the second, optimized version we start from scratch. The design focus of the optimal shell is on performance.

For the initiator shell, we decided to not include error handling as there is no efficient mechanism to report an error. The fact that the message type is in the header means that the initiator shell can only send the response message when it is complete. Meanwhile it has to buffer the messages. This increases the transaction latency. We choose not to implement the alternative solutions discussed in Section 3.3.2.4 because:

- we do not want to change the Æthereal NoC message format specification, as this has an impact on other existing shells (solution one).
- we strive for a structural solution and not a quick hack (solutions two and three).

The following sections detail the design of the PLB target and initiator shells.

3.4.1 PLB target shell

The PLB target interfaces through the PLB bus with the μBlaze. Figure 3.10 shows an overview of the target shell. The bus signals are grouped as follows.

1. Command signal group: these signals include the arbitration signals (valid address, etc) and the transaction signals (read/not write, address, etc). The transfer qualifiers are for initiating, handling and completing transactions. They contain all the necessary information of the transaction.
2. Write data signal group: this bus includes the master write data and the slave acknowledge signals.
3. Read data signal group: this bus includes the slave read data and acknowledge signals.

Not all of the PLB bus signals are used by the target shell. This because some PLB bus features are not supported by the shell, either due to the fact that the μBlaze does not use

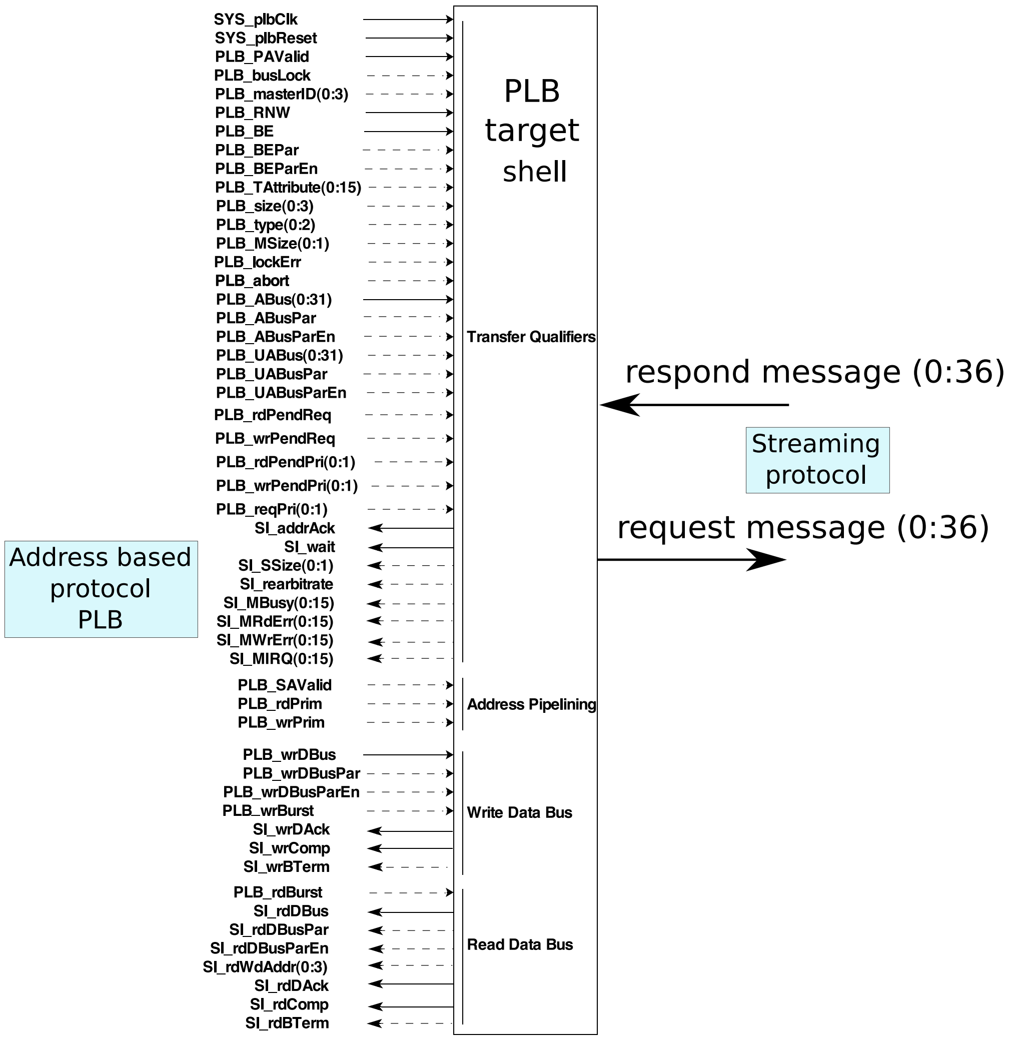


Figure 3.10: PLB target overview [17]. Dashed lines indicate unused signals

them or they are not implemented in the shell. Examples are the burst signals like and errors signals. For the burst case, the μBlaze does not support burst transaction. For this reason we coalesce multiple single transactions into burst transaction when possible in the optimized shell. For the error case, the PLB shell does not support error situations as the initiator shell does not generate error messages. Furthermore, the target shell address is set at design time. The disadvantage of this is that the target shell address cannot change during run time. In Section 3.4.2 we propose a solution.

3.4.1.1 PLB target shell unused signals

The unused signals are as follows [17]

- **PLB_masterID(0:3)**: PLB Master Identification, used by the slave to determine to which master the Sl_MBusy, Sl_MRdErr and Sl_MWrErr signals must be driven on the PLB [17]. We do not have support for errors and busy signals in the PLB shells. We use Sl_wait to indicate that the target shell needs more time to complete the transaction.

- **PLB_BEParEn, PLB_BEPar**: enables parity support for the Mn_BE signal. There is no adequate section in the NoC message format to support this feature.
- **PLB_size(0:3)**: indicate the size of the requested transfer. We instantiated the Æthereal NoC with a word width of 32 bits.
- **PLB_TAttribute (0:15)**: present specific transfer information to slaves. We do not have a use for these signals. Also, there are no adequate sections in the NoC message format to support this feature.
- **PLB_type (0:2)** : to indicate to the slave the type of transfer that is being requested. Two categories of transfer types exist: memory transfer and direct memory access (DMA) transfer. There is no adequate section in the NoC message format to support this feature.
- **PLB_MSize (0:1), Sl_SSize (0:1)**: master and slave size signals, indicate the data bus width of the associated master or slave. We instantiated the Æthereal NoC with a word width of 32 bits.
- **PLB_lockErr**: lock error status. We do not have support for errors in the PLB shells.
- **PLB_abort**: abort request. This request in not applicable in the scope of the NoC. There is no adequate section in the NoC message format to support this feature.
- **PLB_ABusParEn, PLB_ABusPar, PLB_UABusPar, PLB_UABusParEn**: enables parity support for the Mn_ABus and Mn_UABus signals. There is no adequate section in the NoC message format to support this feature.
- **PLB_UABus (0:31)**: allows for address expansion from 32-bits to 64-bits. We instantiated the Æthereal NoC to support an address width of 32 bits.
- **PLB_rdPendReq, PLB_rdPendPri (0:1), PLB_wrPendPri (0:1), PLB_reqPri (0:1)**: read or write request pending on the PLB bus. PLB masters and slave devices can use these signals to help resolve arbitration on the PLB or other buses that are attached to the PLB by a bridge or cross-bar switch [17]. We do not have a use for these signals. These requests are not applicable in the scope of the NoC.
- **Sl_rearbitrate, Sl_MBusy**: to indicate that the slave is unable to perform the currently requested transfer. We use Sl_wait to indicate that the target shell needs more time to complete the transaction.
- **Sl_MRdErr (0: n), Sl_MWrErr (0: n)**: slave error notification. We do not have support for errors in the PLB shells.

- **Sl_MIRQ (0: n)**: slave interrupt notification. We do not have support for interrupts in the PLB shells.
- **PLB_SAValid, PLB_rdPrim, PLB_wrPrim (0: n)**: used in address pipelining. We do not have support for address pipelining in the PLB shell.
- **PLB_wrDBusPar, PLB_wrDBusParEn, Sl_rdDBusPar, Sl_rdDBusParEn**: enables parity support for the Mn_wrDBus and the Sl_rdDBus signals. There is no adequate section in the NoC message format to support this feature.
- **PLB_wrBurst, Sl_wrBTerm, PLB_rdBurst, Sl_rdBTerm**: used for burst transfers. The μBlaze does not perform burst transfers.
- **Sl_rdWdAddr (0:3)**: used for line transfers. We do not support line transfers in the PLB shell.

3.4.1.2 PLB target shell signals mapping

The PLB protocol request signals are mapped to the Æthereal message format. Figures 3.11 and 3.12 show the write request message and the read request message respectively, produced by the PLB target shell.

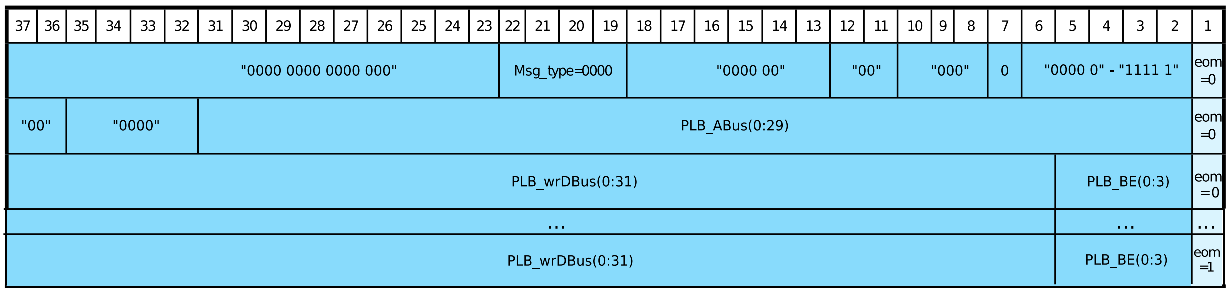


Figure 3.11: Write request message

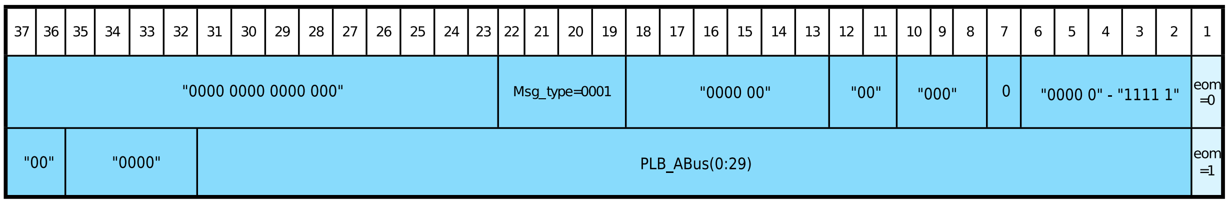


Figure 3.12: Read request message

The PLB bus signals used are as follows.

- **PLB_ABus (0:29)** $\mapsto$ read/write request **Header B (1:30)** address field: contains the address of the transaction. The 2 LSB bits of the address are omitted, due to the fact that the memory is word addressable.

- **PLB_wrDBus (031)** $\mapsto$ write request **Message data (5:36)** data field: contains the transaction write data.
- **PLB_BE (0:3)** $\mapsto$ write request **Message data (1:4)** mask field: contains the master byte enable signals of the transaction.

The rest of the signals are used for handshaking with the PLB bus.

3.4.1.3 PLB target base shell overview

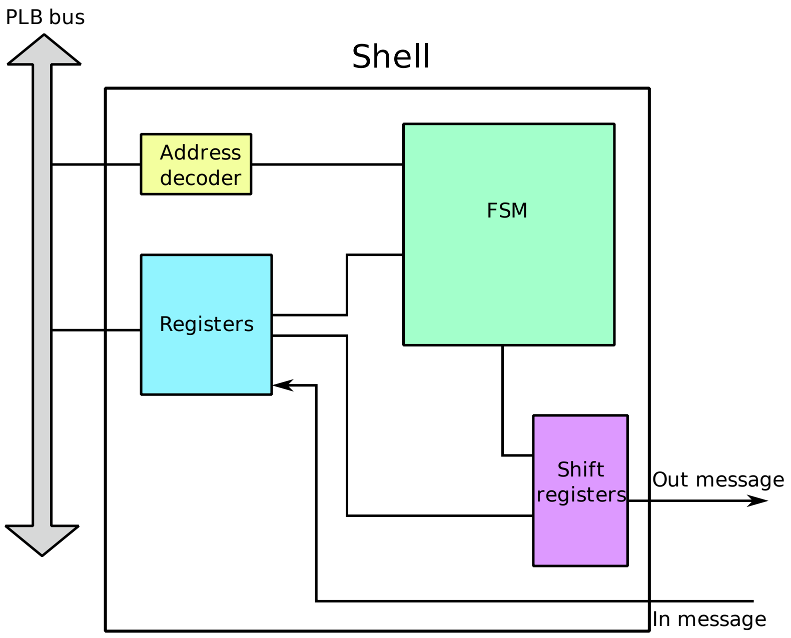


Figure 3.13: PLB target base shell overview

Figure 3.13 gives an overview of the base shell. It consists of:

- **FSM**, performs the handshake with the PLB bus, load the shift register with the messages and to control the de-serialization of the response messages.
- **address decoder**, is used to recognize the slave address from the PLB bus. This component generates a valid signal when a valid address is recognized. The valid signal goes straight to the registers and the FSM
- **registers**, stores the command signals and flags of the bus, to be used by the rest of the components
- **shift register**, performs the handshake with the NIK.

The target shell starts processing the transaction request when it detects a valid address on the bus.

Figure 3.14 shows a diagram of the FSM interface. The inputs 'EOM' and 'rd_valid' are from the incoming message and the NIK respectively. They are used to determine

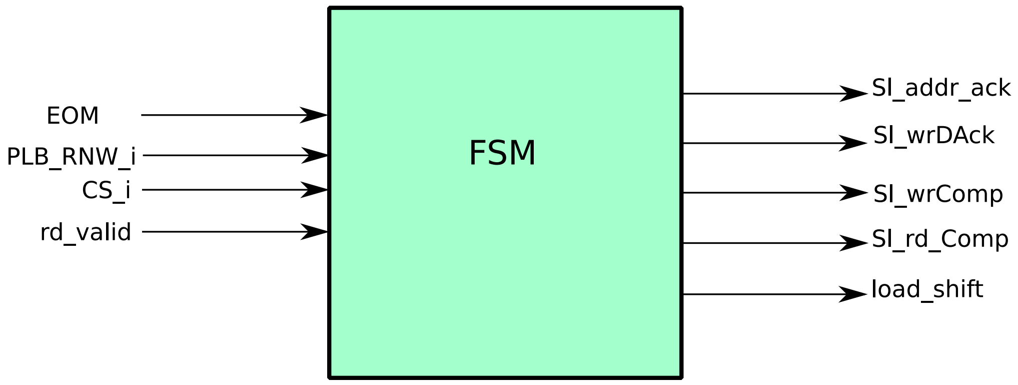


Figure 3.14: PLB target base shell data FSM

whether all the words of a message are received. The 'CS_i' signal is from the address decoder and is used to start the handshake between the FSM and the PLB bus. The 'PLB_RNW_i' is used for determining whether the transaction currently being processed is a read or a write. The FSM generates the slave handshake signals to the bus and it also indicates to the shift register when to load the headers and data messages.

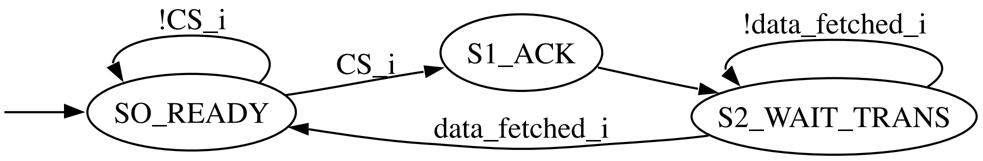


Figure 3.15: PLB target base shell address state diagram

The FSM is internally divided in two separate state machines. One state machine is responsible for the address handshake and the other is responsible for the data handshake. Figure 3.15 shows the state diagram of the address state machine. Only the relevant signals are displayed. When a valid address on the bus is recognized (CS_i=1), the state machine progresses from the initial state to the acknowledge state and the address is acknowledged. After that the state machine progresses to the wait state, where it waits until the data stored in the registers is fetched by the other state machine. This is necessary for the synchronization between the two state machines.

Figure 3.16 shows the state diagram for the data state machine. Again, for simplicity, only relevant signals are displayed. The state machine waits until the address on the bus is acknowledged to start to transition to the next state. Normally, the next state is the encode state, where the shift register is loaded with data. The only exception is when the shift register did not finish shifting data belonging to the previous transaction. This happens when the shells perform a posted write and the NIK did not accept all the data yet. After the shift register is loaded, the state machine transitions into the wait state, where it waits for the response messages to arrive. If the transaction was a posted write, the state machine proceeds to the write acknowledge state. In case of a read transaction,

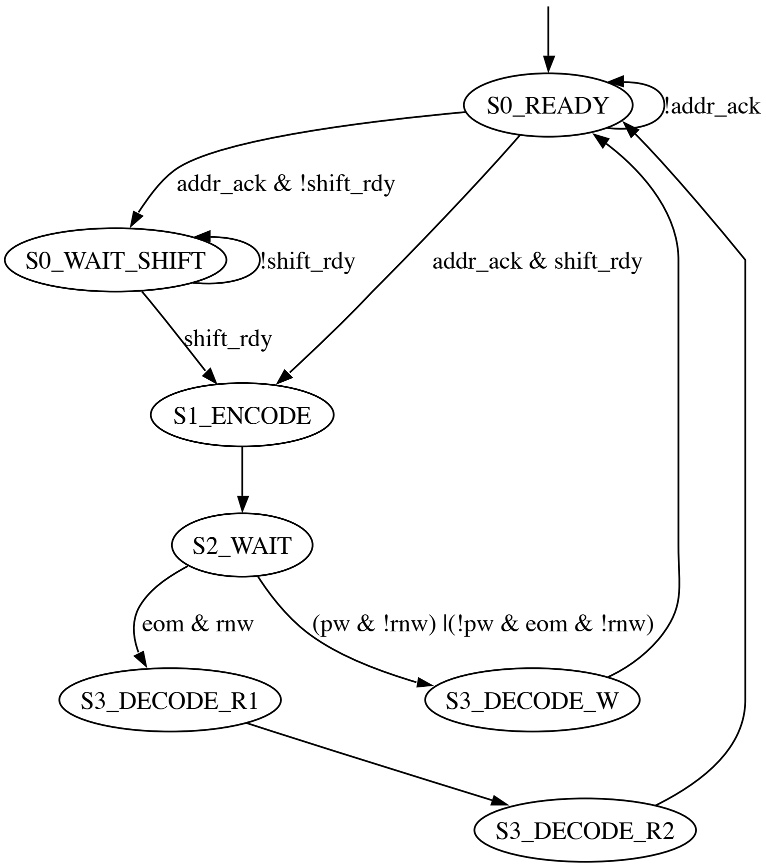


Figure 3.16: PLB target base shell data state diagram

if the data for the last message arrives (EOM=1), the state machine acknowledges the read transaction and presents the read data to the PLB bus.

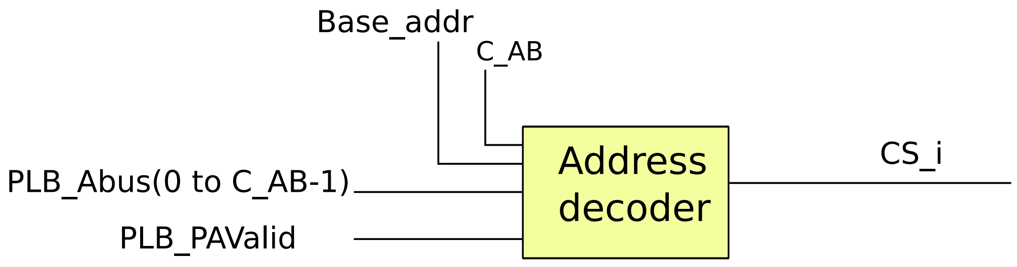


Figure 3.17: PLB target base shell address decoder

Figure 3.17 shows the interface of the address decoder. If the bus address is in range according to the base address, the address decoder will recognize the address as a valid address and generates a valid 'chip select' signal.

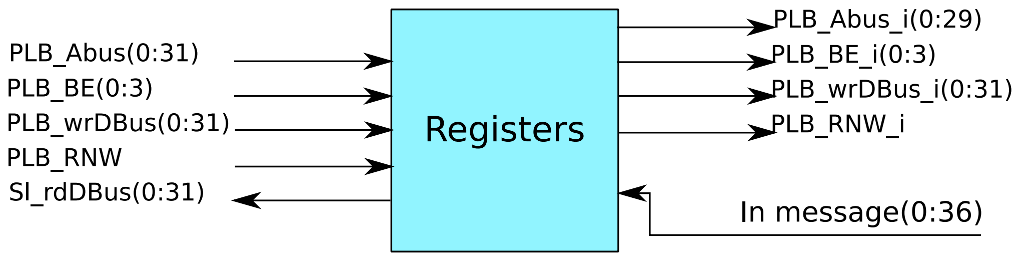


Figure 3.18: PLB target base shell registers

Figure 3.18 shows the base shell registers. They are used for preserving a copy of the command group and write data of the current transaction. This ensures that the transaction commands and data are always available while the shell processes the transaction. Without the registers, the shell may use updated commands or write data from the bus, resulting in erroneous transactions. It also stores the read request data for the PLB bus.

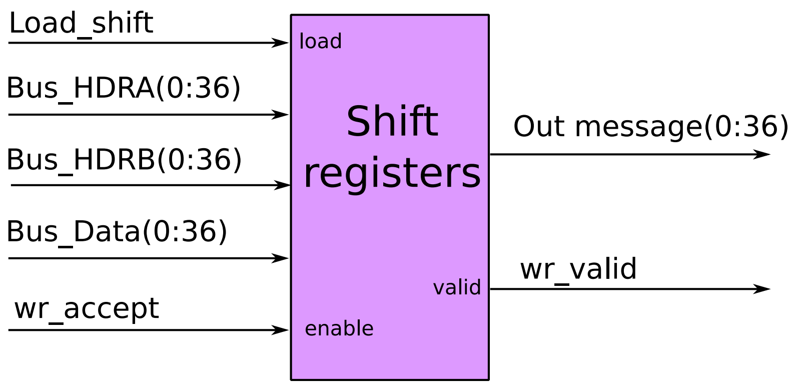


Figure 3.19: PLB target base shell shift registers

Figure 3.19 shows the base shell register. The 'Bus_HDRA', 'Bus_HDRB' and 'Bus_Data' signals are a combination of the registered PLB bus signals, extended with data corresponding to the Æthereal message format. The signals "wr_accept" and "wr_valid" are used for handshaking with the NIK. The "wr_accept" indicates that the NIK accepted the current word of the message, causing the shift register to shift the next word of the message to the output.

3.4.1.4 PLB target optimized shell overview

The aim of the optimized shell is to provide a high performance shell. This is accomplished in a couple of ways. First, the shell only performs posted write transactions and it contains logic to automatically coalesce multiple transactions to a burst transaction if possible. Next, the shell takes full advantage of the possibility to acknowledge the address and write data in the same clock cycle as the write request is made, in compliance with the PLB protocol [17].

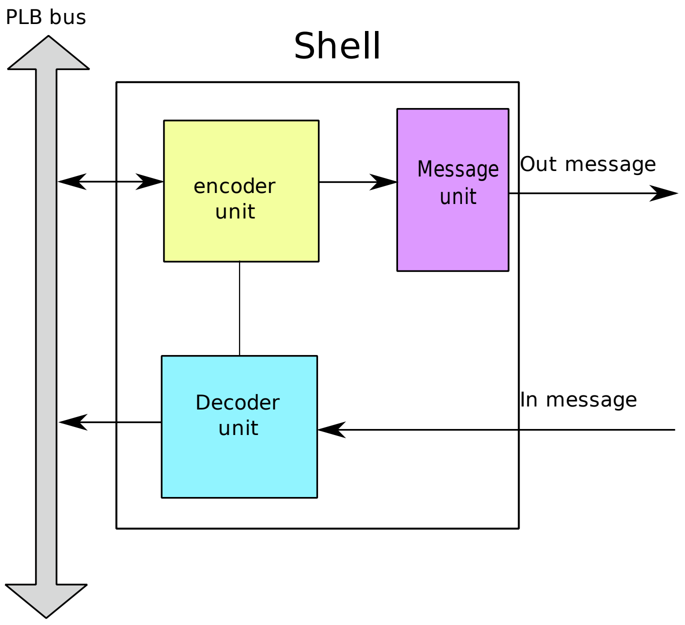


Figure 3.20: PLB target optimized shell overview

Figure 3.20 shows an overview of the optimized shell. The shell has a modular design. The shell consists of three main components. Each of the components performs an independent task.

- **Encoder unit**: encodes the PLB request transaction into Æthereal messages.
- **Message unit**: transmit messages to the NIK.
- **Decoder unit**: decodes the response message into the PLB protocol signals.

With this modular approach, it is possible to replace a unit and still have a fully functioning shell. When there is a valid transaction on the bus, the encoder unit serializes the bus signals into a message, stores it in FIFOs and provides it to the message unit for further handling. The message unit transmits the messages and handles the handshake with the NIK. The decoder unit processes the incoming messages and handles the read transaction handshake of the PLB bus.

We use FIFOs for sending messages to the NIK instead of shift registers. The difference between the FIFOs and the shift registers is that the former has a depth of 32 words (design time configurable) and the latter has a fix depth of 3 words. This allows the encoder to process the next transaction without it having to wait for the words of the message to be accepted by the NIK, as it can store the request messages into the FIFOs. Also, the FIFOs have a separate buffer for address words and data words of the message. The FIFOs are as follows.

- Header FIFO, contains the header word of the message, consisting of the transaction type.

- Address FIFO, contains the header word of the message, consisting of the transaction address.
- Data FIFO, contains the data words of the message.

The benefit of splitting the request process in encoder unit and message unit is as follows. In this setup, the encoder is free to create the header and data words of the request message in random order. This is possible as the encoder unit can store each of the generated words of the request message separately. It is the responsibility of the message unit to first transmit the header words and thereafter the data words of the request message. For example, for burst coalescing, the data word is created immediately. Only after the burst is complete does the encoder unit create the header words, which contains the burst size. The message unit detects that there is a header word present and starts sending the request message to the NIK. Figure 3.21 shows the interface of

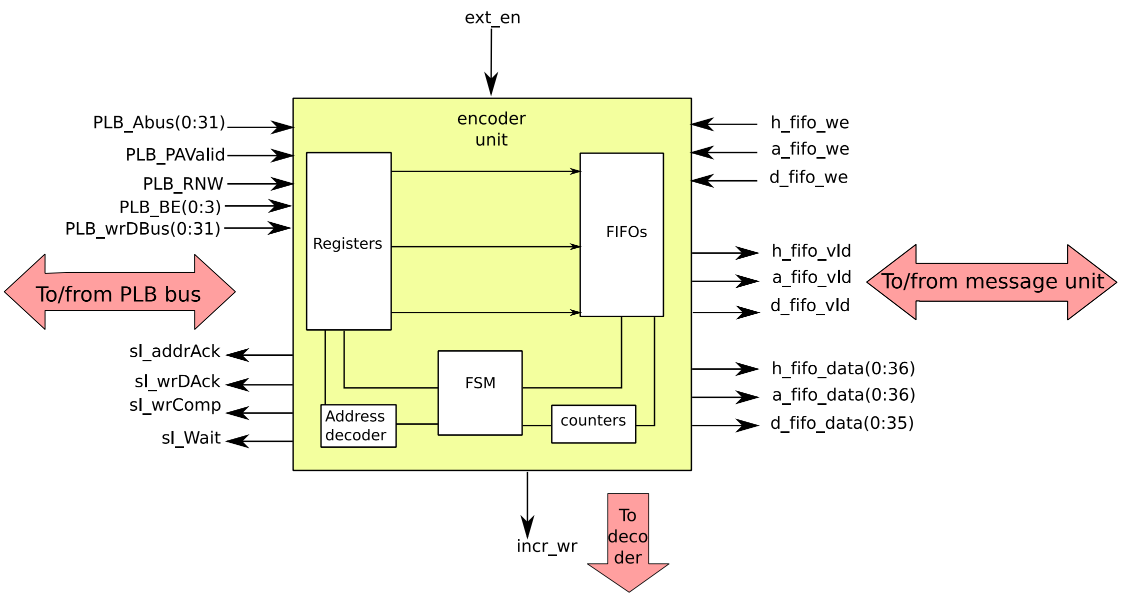


Figure 3.21: PLB target optimized shell encoder unit

the encoder unit. A write transaction is immediately acknowledged, with the write data word being stored in the FIFOs. The write transaction header words are generated depending whether the write transaction is a burst or not. For a read transaction, the header words are immediately sent to the FIFOs.

The address decoder determines whether the bus transaction is valid and generates a valid signal for the other components.

The registers are used to store the command group signals, flags and address. The FSM is responsible for loading the request header words into the FIFO and for enabling the burst counter. The encoder unit also keeps track of the outstanding write transactions. This allows synchronization with other units.

Figure 3.22 shows the FSM of the encoder. The shaded boxes indicate the states that recognize the write transaction. The logic, that detects a burst transaction, has to satisfy to the following conditions:

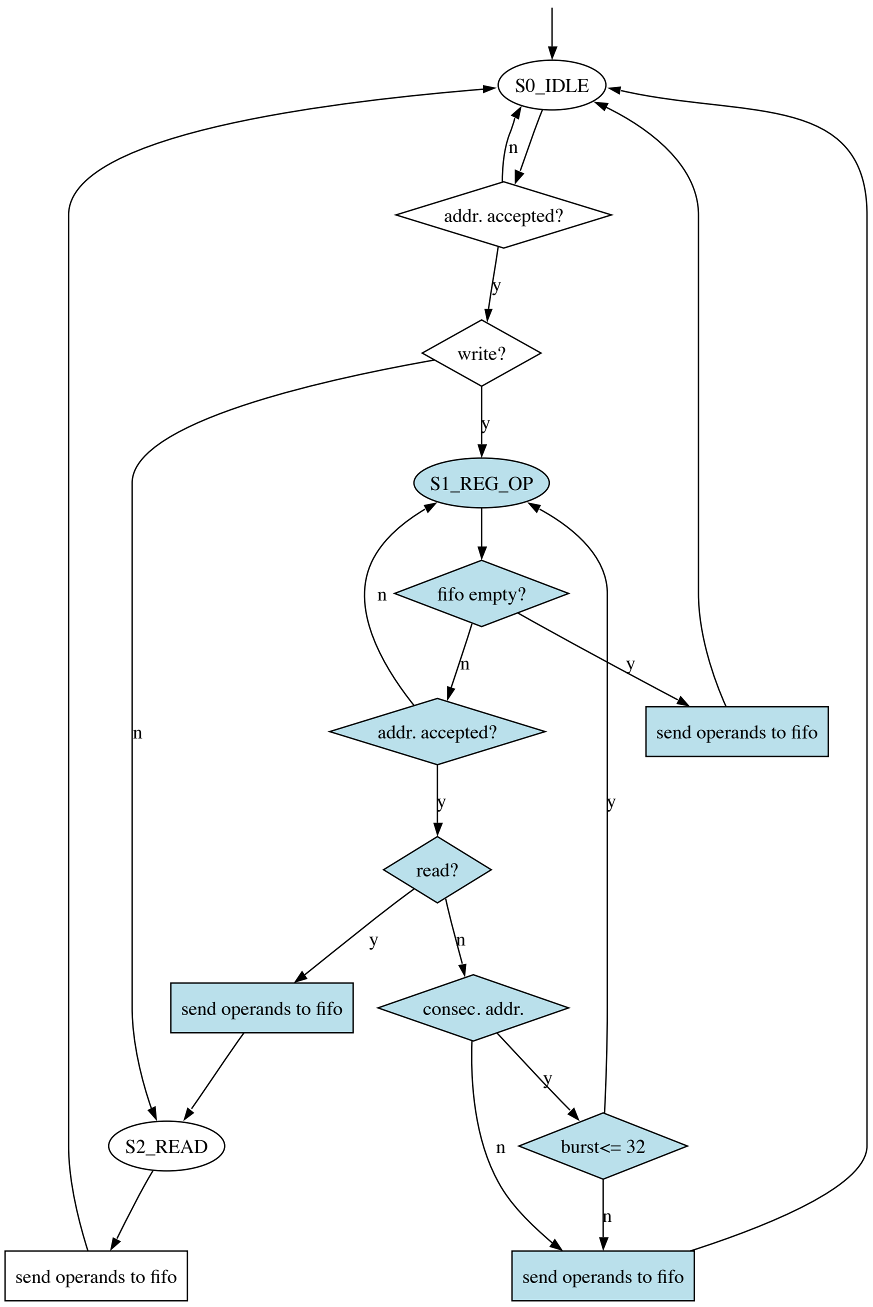


Figure 3.22: PLB target optimized shell encoder unit FSM

- state is 'S1_REG_OP', the write recognition state.
- the FIFOs must not be empty, because this indicates that the network is idle. When the network is idle, the overhead of sending headers is not relevant. When

it is not, we introduce extra delay by waiting.

- there is a valid write transaction.
- current address=previous address + 4. Thus the current address is a consecutive address.
- the burst counter must be smaller or equal to the maximum burst size supported (thirty two).

In the write state, if a burst is in progress, the base address will be stored and the consecutive addresses will be discarded. We store the base address, because we need to compare with consecutive addresses. If all these conditions are met, a burst transaction is in progress. If one of these conditions is not met, the FSM wraps up the transaction by sending the headers to the FIFOs and resetting the counter. The headers are created when the FSM stores the operands in the register. When the state machine is in the write detection state and a read occurs, it first wraps up the write transaction and then wraps up the read transaction by storing the read message headers immediately into the FIFOs.

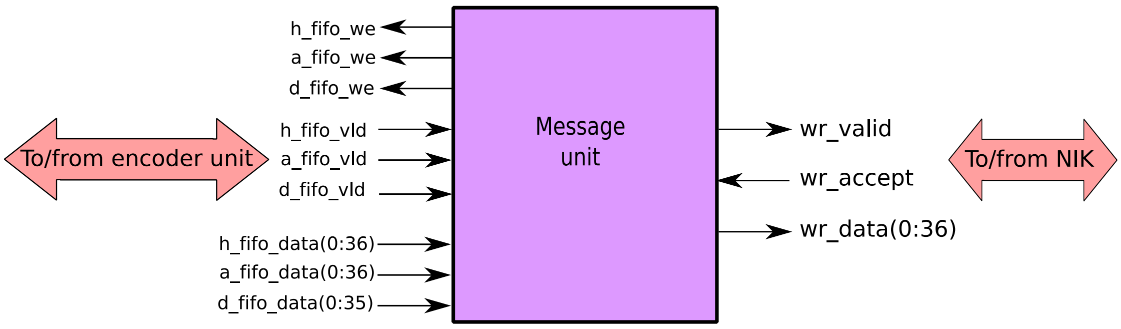


Figure 3.23: PLB target optimized shell message unit

Figure 3.23 shows the interface of the message unit. It receives the messages from the encoder unit and performs the handshake transaction with the NIK. The unit sends data to the NIK as soon as there is a valid header message. For a burst transaction, it keeps track of the number of messages it sends to the NIK. The data messages always arrive first in the FIFO. The last message to arrive is the header messages. When the last data word of the message is sent, it appends the EOM bit to the data word.

Figure 3.24 shows the decoder unit. The decoder unit is used for decoding the read response message and for bookkeeping on the outstanding writes. A read response message is decoded into the PLB read acknowledge and data signals. The encoder asserts the 'incr_wr' signal each time a write transaction occurs. This causes the decoder to increase the outstanding writes counter. While there is an outstanding write transaction, the 'ext_en' signal is de-asserted, causing other shells to stall. When a write acknowledge message arrives, the outstanding write counter is decremented. The 'ext_en' signal is asserted when there are no outstanding writes anymore.

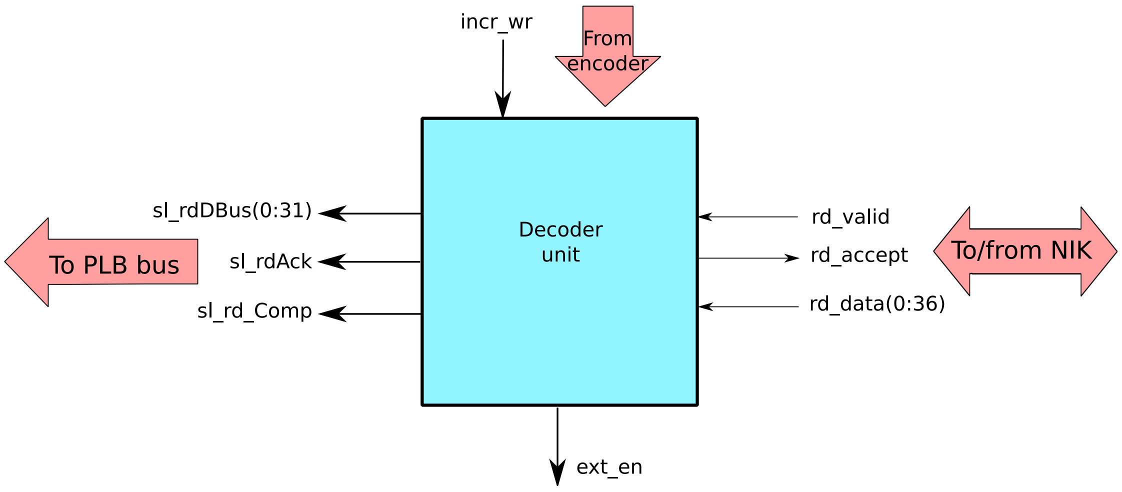


Figure 3.24: PLB target optimized shell decoder unit

3.4.2 Target shell address

It is desirable to change the target shell address at run time. We propose a method to achieve this. Currently we compare the bus address to the base address set at design time to recognize a valid address. To offer more flexibility, we propose to store the shell address in a register. This way a new address can be set by overwriting the value in the register. Figure 3.25 shows a possible setup. The address set at design time is used to

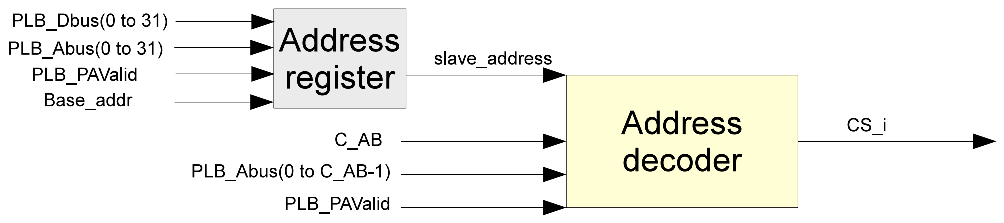


Figure 3.25: PLB target synchronization unit

address the register. During setup, a valid value must be written to this register. This value should be the slave address. During runtime, a different slave address can be set, by writing a new value to the register at the shell design time address. The address register uses the 'PLB_PAValid', 'PLB_Abus' and 'Base_addr' signals to determine whether it is addressed. If that is the case, it fetches and stores the data from the bus. This will be the new slave address, used by the address decoder.

3.4.3 PLB target optimized shell synchronization

The decision to only support posted writes has some drawbacks. In a multi-processor setup, posted write can lead to synchronization problems between the processors. To solve this, the optimized shell has support for synchronization between multiple shells.

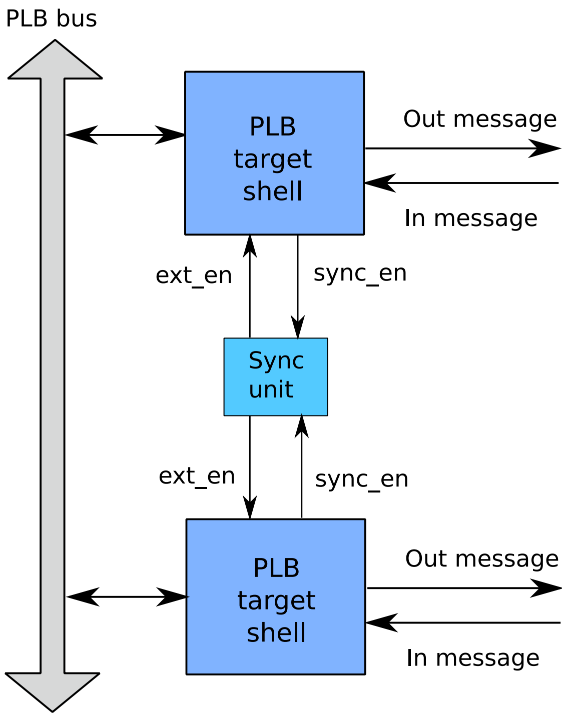


Figure 3.26: PLB target synchronization unit

Figure 3.26 shows this situation. The synchronization unit stalls the rest of the shells when there are outstanding write transactions in a shell.

3.4.4 PLB initiator shell

The PLB initiator interfaces through the PLB bus with the on-chip memory of any slave. The initiator shell is a master on the PLB bus. Figure 3.27 shows an overview of the shell interface with the PLB bus. The PLB signals are grouped in three categories:

- Request qualifiers signals, includes the transaction request, address and read/write signals.
- Write data signals, includes the write data and write acknowledge signals.
- Read data signals, includes the read data and the read acknowledge signals.

Like in the case of the target shell, not all signals are used by the shell.

3.4.4.1 PLB initiator shell unused signals

The unused signals are as follows [17]

- **PLB_MnTimeout**: indicates that the transaction timed out. We continue to drive the transaction request until we get an acknowledge from the slave.

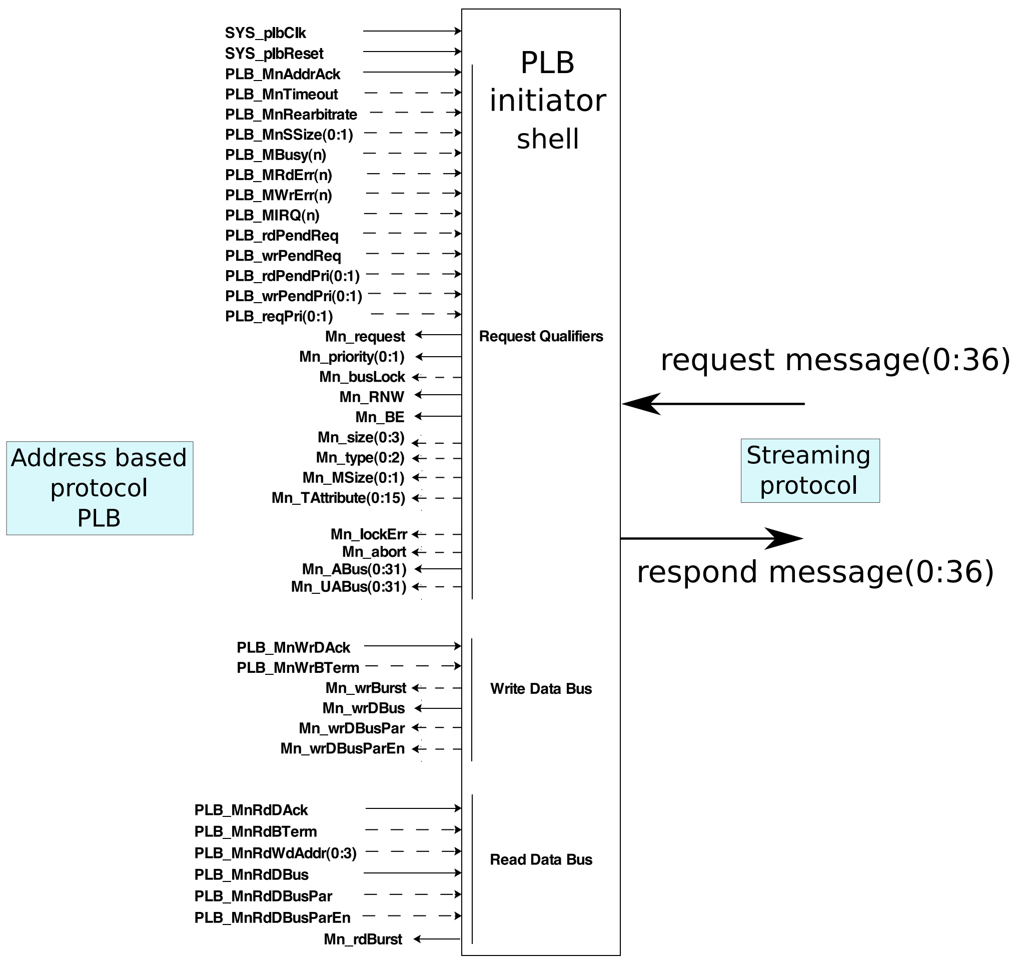


Figure 3.27: PLB initiator overview [17]. Dashed lines indicate unused signals

- **PLB_Mnrearbitrate, PLB_MBusy (n)**: used by the slave to indicate to the master that it is unable to perform the currently requested transfer. We continue to drive the transaction request until we get an acknowledge from the slave.

- **PLB_MRdErr(n), PLB_MWrErr(n)**: slave error notification. We do not have support for errors in the PLB shells.

- **PLB_rdPendReq, PLB_WrPendReq PLB_rdPendPri (0:1), PLB_wrPendPri (0:1), PLB_reqPri (0:1)**: read or write request pending on the PLB bus. PLB masters and slave devices can use these signals to help resolve arbitration on the PLB or other buses that are attached to the PLB by a bridge or cross-bar switch [17]. We do not have a use for these signals. These requests are not applicable in the scope of the NoC.

- **Mn_busLock**: acquire a lock on the bus. We do not lock the bus during a transaction.

- **Mn_size(0:3)**: indicate the size of the requested transfer. We instantiated the Æthereal NoC with a word width of 32 bits.
- **Mn_type (0:2)**: to indicate to the slave the type of transfer that is being requested. Two categories of transfer types exist: memory transfer and direct memory access (DMA) transfer. There is no adequate section in the NoC message format to support this feature.
- **Mn_MSize (0:1), PLB_MnSSize (0:1)**: master and slave size signals, indicate the data bus width of the associated master or slave. We instantiated the Æthereal NoC with a word width of 32 bits.
- **Mn_TAttribute (0:15)**: present specific transfer information to slaves. We do not have a use for these signals. Also, there are no adequate sections in the NoC message format to support this feature.
- **Mn_lockErr**: lock error status. We do not have support for errors in the PLB shells.
- **Mn_abort**: abort request. This request in not applicable in the scope of the NoC. There is no adequate section in the NoC message format to support this feature.
- **Mn_UABus (0:31)**: allows for address expansion from 32-bits to 64-bits. We instantiated the Æthereal NoC to support an address width of 32 bits.
- **PLB_MIRQ (0: n)**: interrupt notification. We do not have support for interrupts in the PLB shells.
- **Mn_wrDBusPar, Mn_wrDBusParEn, PLB_MnrdDBusPar, PLB_MnrdDBusParEn**: enables parity support for the Mn_wrDBus and the PLB_MnrdDBus signals. There is no adequate section in the NoC message format to support this feature.
- **Mn_wrBurst, PLB_MnWrBTerm, Mn_rdBurst, PLB_MnRdBTerm**: used for burst transfers. The μBlaze does not perform burst transfers.
- **PLB_MnRdWdAddr (0:3)**: used for line transfers. We do not support line transfers in the PLB shell.

3.4.4.2 PLB initiator shell signals mapping

The PLB protocol response signals are mapped to the Æthereal message format. Figure 3.28 shows the read response message, produced by the PLB initiator shell.

The PLB bus signals used are as follows.

- **PLB_MnRdDbus (0:31)** $\mapsto$ read request **Message data (5:36)** data field: contains the transaction read data.

The rest of the signals are used for handshaking with the PLB bus.

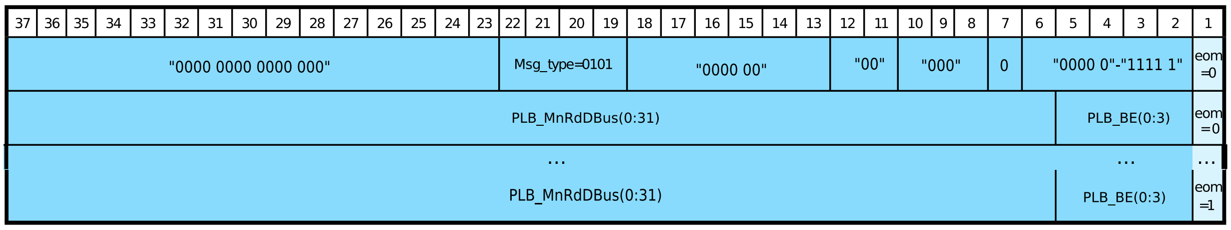


Figure 3.28: Read response message

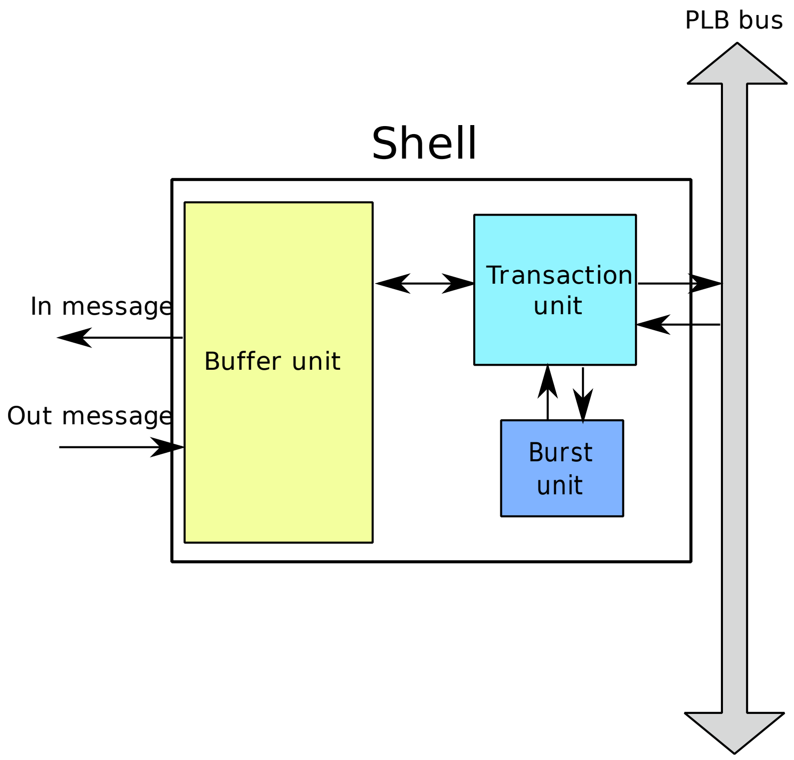


Figure 3.29: PLB initiator shell details

3.4.4.3 PLB initiator shell overview

Figure 3.29 gives an overview of the structure of the initiator shell. It consists of three main components.

- Buffer unit: decouples the message handling from the message processing and handles the handshake between the shell and the NIK
- Transaction unit: processes the messages, handles the handshake transaction between the shell and the bus.
- Burst unit: computes the next sequential address and keeps track of the burst length.

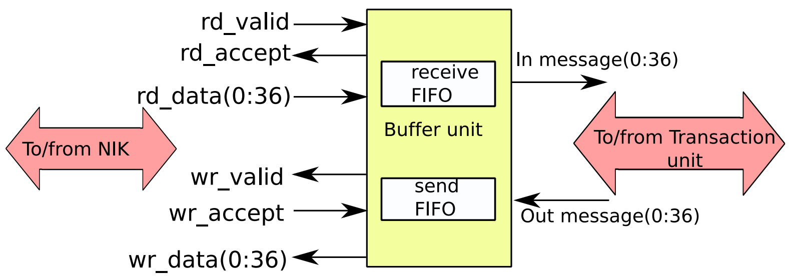


Figure 3.30: PLB initiator shell buffer unit

Figure 3.30 shows details of the buffer unit. It consists of two FIFOs, one for receiving data from the network, the other for sending data to the network. The FIFOs handle the handshake with the NIK. The receive FIFO provides the transaction unit with the messages for processing. The send FIFO transports the response messages of the transaction unit to the NIK.

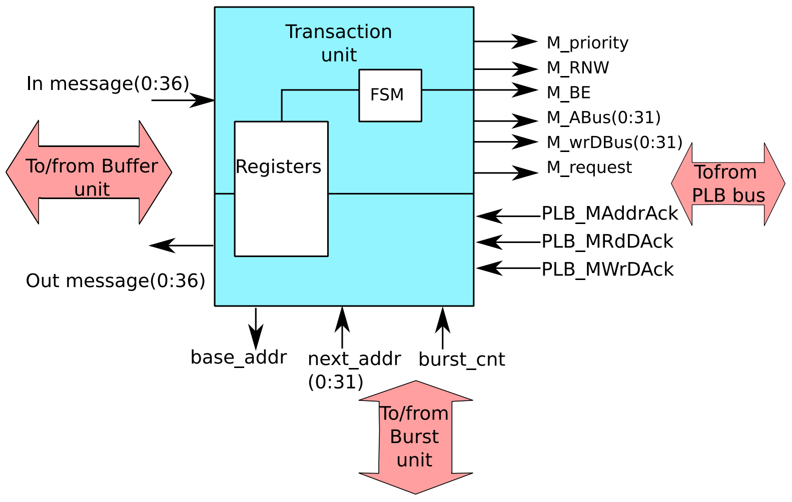


Figure 3.31: PLB initiator transaction unit

Figure 3.31 shows the transaction unit. It consists of registers and a FSM. The registers are used to hold the transaction data during the transaction. The FSM handles the handshake with the PLB bus and controls the loading of data into the registers. The FSM consists of two control units, one for handling the address and transfer qualifiers of the transaction and one for handling the transaction data.

The state diagram of the address state machine is illustrated in figure 3.32. Only relevant signals are shown. The first message to arrive is the header and contains information about whether the transaction is a read or write and the burst length. The next message is also a header, it contains the transaction address. If it is a write transaction, the next message(s) contain(s) write data. The state diagram shows these steps. First the read or write is decoded, after that the address. Depending whether the transaction is a read or a write, the next action is to start the transaction request or decode the

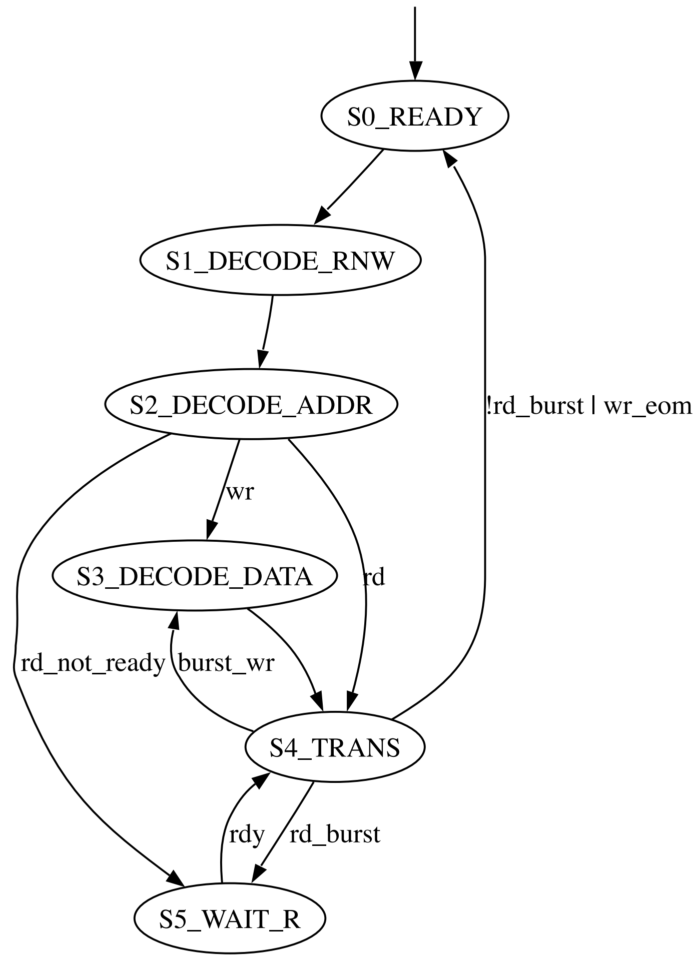


Figure 3.32: PLB initiator shell transaction unit address state diagram

write data.

A burst write or read request is handled as follows.

- **Burst read request message**: the initiator shell splits the burst read request message into multiple single request transactions to the PLB bus. This to ensure that all slaves can perform the transaction, even slaves that do not support burst transaction. The read request address is updated after each time the target memory acknowledges the transaction, and presented again to the bus. The burst read request terminates when the burst size is zero. For the response, the initiator shell gathers and converts all of the individual read data from the target memory into response data words and sends them as one burst response message to the NIK.
- **Burst write request message**: the initiator shell splits the burst write request message into multiple single write transactions to the PLB bus. This also to ensure

that all slaves can perform the transaction, even slaves that do not support burst transaction. The burst write is terminated by the message EOM. While there is no EOM, the shell will threat the next word as a write message data, uses the sequential address from the burst unit and presents the bus with the new transaction. For the response, each time the target memory acknowledges a write transaction, the initiator sends an acknowledge write response message to the NIK. For the write response, the same acknowledge message is used, regardless if it is a burst or not.

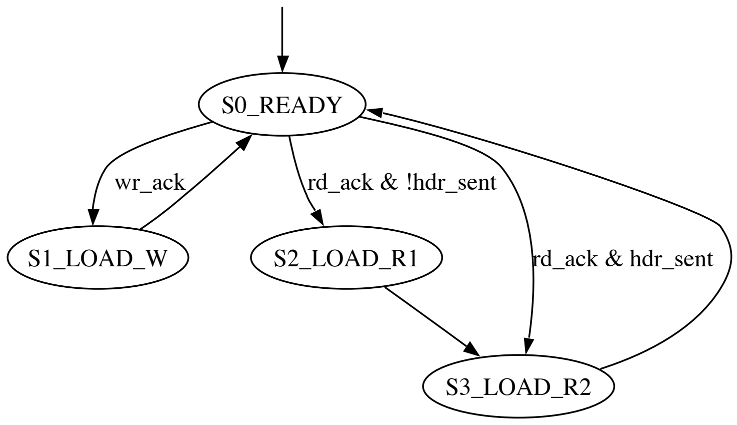


Figure 3.33: PLB initiator shell transaction unit data state diagram

Figure 3.33 shows the state diagram of the data transaction. The purpose of this state machine is to load the appropriate response messages. For the write transaction, when the write is acknowledged, the state machine creates a write acknowledge message header. A burst write is acknowledged with multiple single write acknowledges, according to the specification. The case for a read transaction is different. The read response message consists of message headers and the message read data. For a burst write, the header needs to be sent only once.

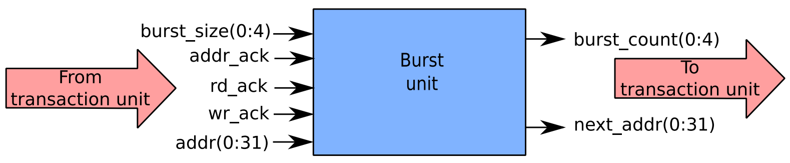


Figure 3.34: PLB initiator shell burst unit

Figure 3.34 shows the burst unit. The purpose of the burst unit is to provide the transaction unit with the burst related data. For a burst, the burst unit computes the next sequential address, based on the base address provided. It also keeps track of the number of burst left. It uses the PLB acknowledge signals to update the burst data.

3.5 FSL shell

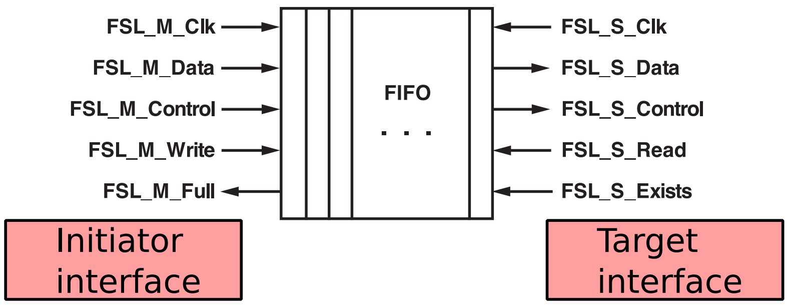


Figure 3.35: FSL master and slave interface

The FSL link is used as a dedicated prefetch read link. It can both perform single read as well as burst reads. The FSL is an unidirectional FIFO link, with two interfaces: initiator and target [30]. The initiator pushes data into the FSL and the target consumes data from the FSL. The FSL shell interfaces through a pair of FIFO links with the μBLaze. The shell has to interface with both the target and initiator interface of the FSL. Figure 3.35 shows the interface of the FSL. The difference between the FSL and the PLB is as follows. The PLB is a bus interconnect, uses an address based protocol for transaction, with command signals group, write signal group etc. The FSL is a FIFO, uses a streaming protocol for transaction, with handshake signals. This has an impact on the shell design, as it does not have to deal with various signals group of the bus. This enables the shell to have a flexible, simple design and interface. The streaming nature of the FSL enables us to explore more alternative designs, as we are not bound to a fixed bus protocol. In this section we present an FSL shell. In section 3.5.2 we propose an alternative method for using the FSL link.

3.5.1 FSL shell design

The FSL shell can be used for both read and write communication, but we only perform reads because they have the largest performance gain. We use the FSL shell to provide the μBlaze with the capability of performing prefetch reads (Section 2.5). The data flow from the μBlaze to the FSL shell and vice versa (Figure 3.36) is as follows.

- **Request from initiator to target**: the μBlaze performs a read request, single or burst, by pushing the read request address on the FIFO. The FSL shell reads the data from the other end of the FIFO, processes the address, creates an Æthereal read request message and send it to the NIK.

- **Response from target to initiator**: the FSL shell receives the response message from the NIK, decodes the message and pushes the response read data into the FIFO for the μBlaze to retrieve at the other end.

There is no need for a FSL initiator shell as a normal PLB initiator shell is used to process the request message.

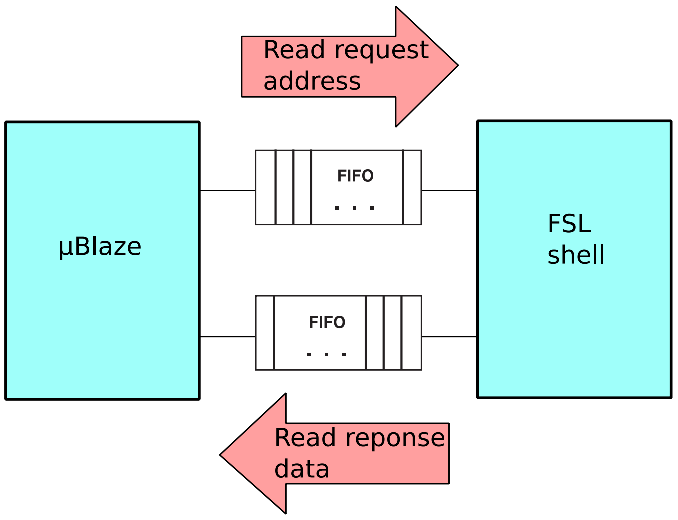


Figure 3.36: FSL link data flow

Furthermore, the FSL shell has two modes of operations. They are operation and configuration mode. In operation mode, the FSL functions as a DMA. Configuration mode is used to configure the burst size of the read request transaction, as the FSL shell is capable of performing both single as burst transactions. The details of the two modes are as follows.

- **Configuration mode**. The μBlaze specifies the 5 bit burst size of the read transaction. The shell stores the burst size in its internal registers.

- **Operation mode**. The μBlaze writes the 32 bit address it wants to read from to the FSL and the shell performs a read request.

The modes are selected by using the control signal of the handshake signals of the FSL. When the control bit is asserted, the mode of operation is configuration mode. When the control bit is de-asserted, the mode is operation mode.

The FSL shell has a relatively simple design. This because it only has to perform read transactions and it does not have to interface with a complicated bus protocol. Figure 3.37 shows an overview of the FSL shell components. It consists of registers to hold the read message headers and also the burst size. The FSM controls the registers and performs the handshake with the NIK.

The FSM consists of 2 state machines. One is used for handling the request message and the other one is used for handling the response message. This allows the shell to process both incoming and outgoing messages concurrently.

Figure 3.38 shows state diagrams for both the request and the response state machines. The request state machine is responsible for handling the handshake with the NIK. First the read message header is sent and after that the message data is sent. The wait state is for when the NIK is not accepting messages. The response state machine is responsible for controlling the write to the FSL link. It handles the FSL link master

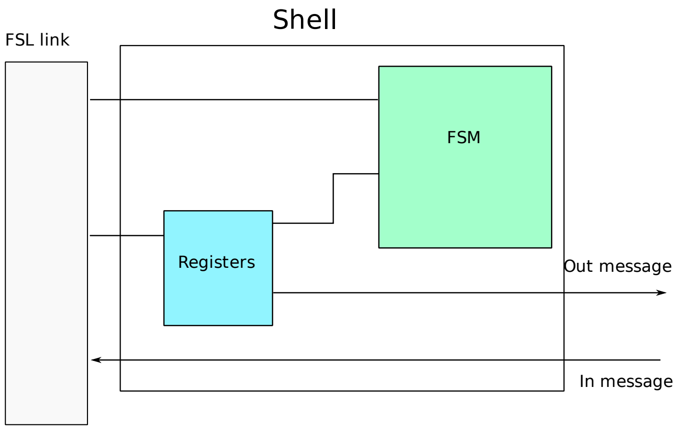


Figure 3.37: Overview of the FSL shell design

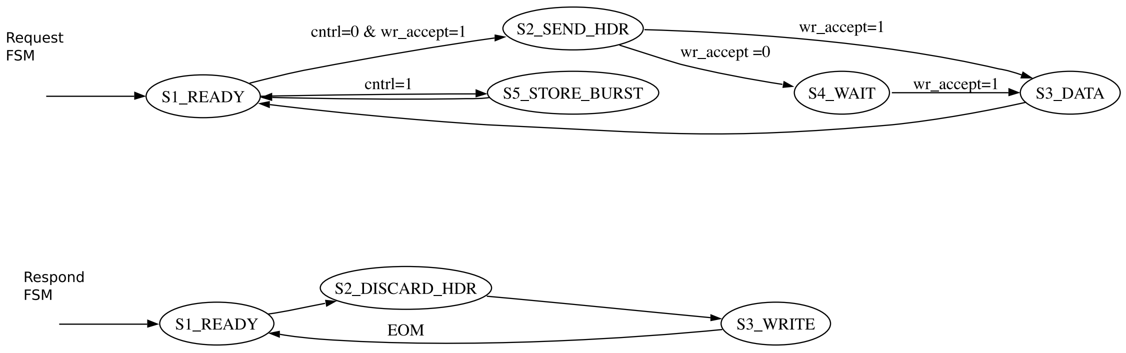


Figure 3.38: FSL shell state diagrams

interface. The response message header is discarded and the response message read data is written to the FSL link by enabling the FSL write control signal, FSL_M_WRITE.

3.5.2 Proposal for an alternative FSL utilization

We propose a method for performing read and write transaction, using the FSL, from the μBlaze directly to the NIK, i.e. without the need of an FSL shell. That is, writing and reading to/from the NIK in software, without the need of a protocol conversion hardware component. This is possible because the FSL has a simple handshake streaming protocol. First we present the design this "software NI shell", after that we discuss the advantages and disadvantages of this approach.

3.5.2.1 Design of the "software NI shell"

Figure 3.39 shows the design of the "software NI shell". The idea is to have the μBlaze perform read and writes to the FSL port and, after that, combine the transactions to form the header and payload of the message.

For this purpose, we use two pairs of the FSL link. One pair is used for the request message and the other pair is used for the response message. The streaming protocol of

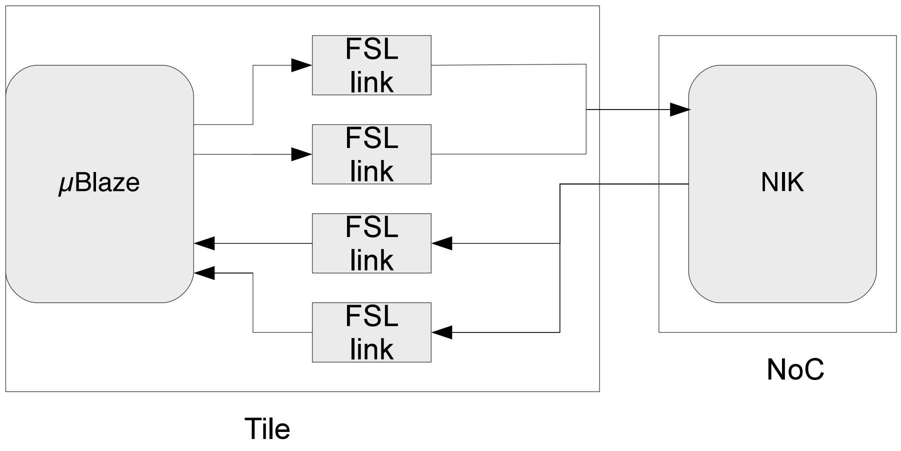


Figure 3.39: software NI shell

the FSL consists of a valid and acknowledge handshake. They work as follows.

- **Request transaction**: the two 32 bits outputs of the FIFOs target interface are concatenated to form the 37 bits word of the request message. The two "FSL_S_Exists" signals of the FIFOs target interface are AND'ed together to form a "ni_valid" signal (valid) for the NIK. The "FSL_S_Read" signal for the FIFOs is formed by demuxing the "ni_accept signal (acknowledge). Additional logic is used to ensure that the "FSL_S_READ" is not asserted before there is data in the FIFO, as this results in undefined behavior in the FSL.

- **Response transaction**: The 37 bits word of the response message from the NIK is split into two to form the input of the two FIFOs. The "ni_valid" signal of the NIK is demultiplexed to form the "FSL_M_WRITE" of the FSL. The FIFOs "FSL_M_FULL" are AND'ed together to form the "ni_accept" signal. Additional logic is used to ensure that the "FSL_M_WRITE" is not asserted if the FIFO is full, as this results in undefined behavior in the FSL.

All the transactions are performed in software. To form one word of the message, two writes to FSL (Section 3.6) are needed. To make this method more transparent for the user, we propose to encapsulate the communication with the FSL in software libraries.

3.5.2.2 Analyses of the "software NI shell"

The advantages of this approach are as follows.

- Flexibility to support all possible transactions. The IP can even perform transactions that are difficult or not supported at the time the IP was designed (for example burst transfers).

- Easy migration to future message formats, as long as the message width does not change. Only need to change the software mapping

- Compatible with not only the specific FSL streaming protocols, but all streaming protocol in general. May need minor adjustment in hardware logic and software for complete compatibility

The disadvantages of this approach are as follows.

- The transactions are performed in software. This results in more cycles per transaction in comparison to a dedicated hardware component. These extra cycles add to the transaction latency and have a negative effect on the performance of the IPs.
- Not all IPs have a streaming protocol. This limits the usability of this approach.

3.6 Impact on software

The OPB and PLB shell are memory-mapped addressable. The addresses of the shells are configured at design time. Generics are used to select features of the shell or to set certain shell parameters. The most common are:

- 'Base_addr', sets the base address of the shell
- 'C_AB' sets how many bits of the address needs to be compared before the slave recognizes the address
- Posted_Write, when enabled the target shell performs posted writes.

The base and high address of the target shell should be the same as that of the slave being addressed. There are no restrictions for the address range of the initiator shell.

3.6.1 Local vs. remote memory

We define the memory in the master tile as the local memory and the memory in the slave tile as the remote memory. We have to modify the software in order to communicate with the remote memory. The remote memory is reachable by writing/reading to/from the address range it occupies. This is accomplished by accessing the target shell in the master tile, as the address range of the remote memory is the same for the target shells.

For example, in the following code, the values 0 to 99 are written to consecutive address in the remote memory. We assume that the base address of the target shell and remote memory is 0xB0010000.

```
int i;
unsigned int *X;
X=(unsigned int *)0xB0010000;
for (i=0; i<100; i++)
{
          X[i] = i;
};
```

3.6.2 API for split pipelined reads

The FSL shell is not memory-mapped addressable. This is due to the fact that the FSL shell is not attached to a bus interconnect but to a FIFO link. The communication to the FSL shell is performed through special ports on the μBlaze. The EDK provides special macros to access the FSL links. We use the 'get' and 'set' instructions of the macro as follows.

- `putdfslx(var, id, FSL_CONTROL).`

  The control bit is asserted for this instruction. It is used to set the burst size in the FSL shell. The burst size is 'var', with 'id' being the FSL link pair id on the μBlaze.

- `putdfslx(var, id, FSL_DEFAULT).`

  This instruction is used for transferring the read address. The control bit is not set for this instruction. 'var' is the memory location of the address.

- `getdfslx(var, id, FSL_DEFAULT).`

  This instruction is used to fetch data from the FSL link. This performs a blocking read from the FSL. The data is put in the 'var'.

3.7 Conclusions

In this chapter we presented the design of the shells. We chose an iterative design methodology was as it enabled us to focus on a working prototype first and we considered more advanced features only thereafter. For the actual shell designs, we use a modular approach when possible. Furthermore, we developed a test strategy in order to detect and fix design and logic errors, and also to verify the functionality of the shells.

As for the shells, we first presented the OPB shell. This shell was used as a prototype design. We implemented error detection in this shell only, due to the impracticalities associated with error reporting. Next we presented the PLB shells. For the PLB target shell, we designed two versions. The first version was built on the OPB target shell. For the second version, we introduced a couple of novel features, like the possibility to coalesce multiple transactions to burst transactions. To further reduce the latency from the IPs point of view, we introduced a dedicated read link for performing prefetch reads. Last, to use these features, we described the necessary modifications needed for the software application.

Experimental setup 4

In this section we present various SoC configurations we have used in our experiments. The goals of these experiments are as follows.

- Verification: we want to verify that our shells function correctly.
- Evaluation: we want to quantify the effect of latency and bandwidth on the performance of the system.

In all of the configurations, a μBlaze processor was used for reading and writing to an on-chip memory. The resulting system configurations validate and demonstrate the functionality of our shell designs. We use the PLB bus, the FSL link and the Æthereal NoC as interconnects. We decided not to experiment with the OPB bus, as it provides no benefit over the PLB implementation. We present and analyze the results from the experiments in the next chapter.

4.1 NoC setup

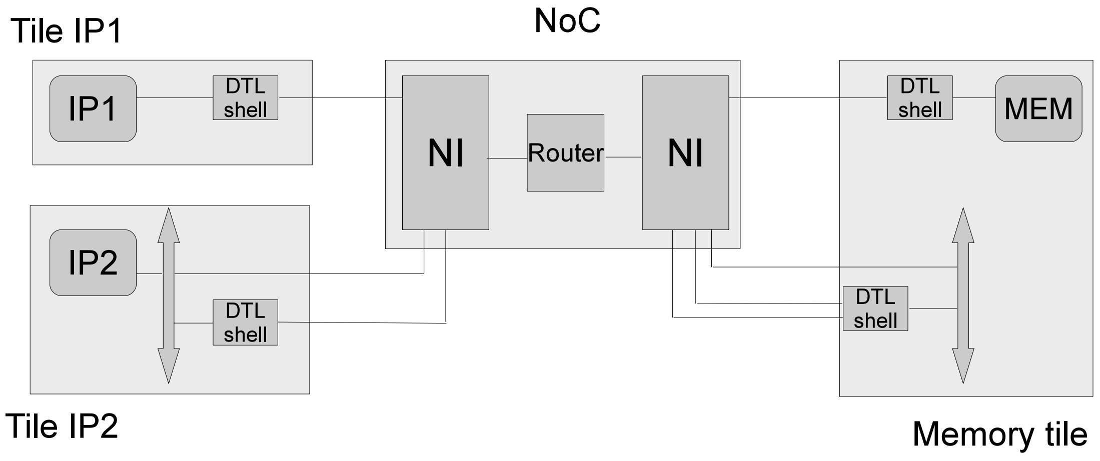


Figure 4.1: Generated Æthereal NoC

Figure 4.1 shows the generated NoC. To ease the integration of the NoC with the rest of the system, we modify the NoC as follows.

- We replaced the DTL shells with the PLB/FSL/OPB shells. This because our IPs don't use the DTL protocol

- We replaced the DTL bus on the configuration ports with a PLB/OPB bus. We connect our shells directly to the NI
- We also use the μBlaze as configuration master, instead of a dedicated IP.

Figure 4.2 shows the resulting NoC.

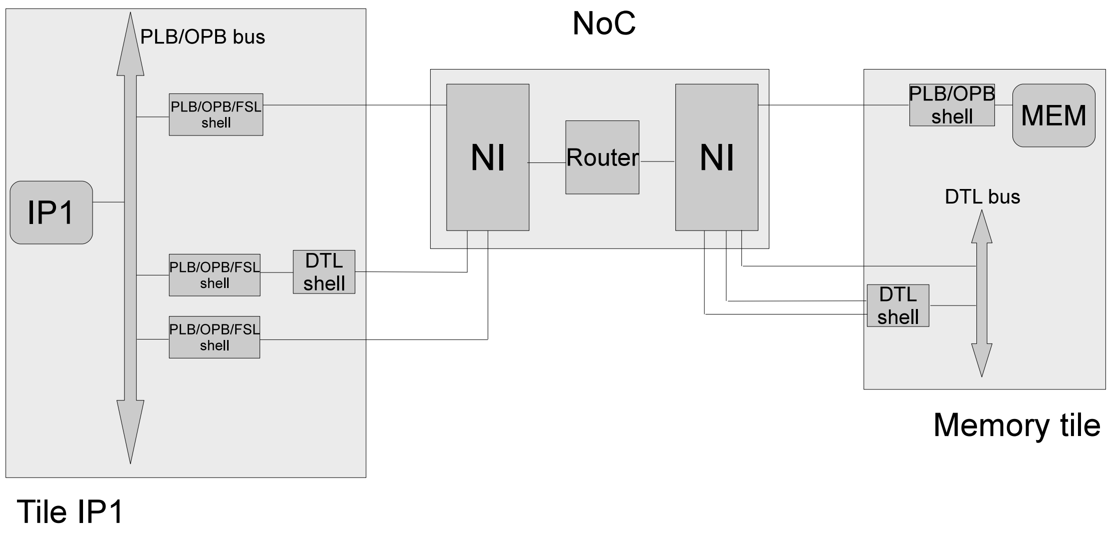


Figure 4.2: Modified Æthereal NoC

4.2 Embedded systems

We used a combination of the various shells to produce several SoCs. Each system supports certain features. We made the distinction between the basic PLB target, which has support for non-posted and posted writes, and the optimized PLB target, which has support for posted and burst write transactions. We defined, depending on the target shell used, a base and an optimized SoC. The base SoC uses the basic PLB target. The optimized SoC uses the posted burst PLB target. The NI connected to the μBlaze via a PLB bus, is the configuration NI. The μBlaze is the configuration master.

4.2.1 Base SoC

Figure 4.3 shows a simple system consisting of μBlaze, a local and remote memory. With this system, it is possible to have part of the application data in the remote memory. The application instruction still resides in the local memory.

Instead of attaching the remote memory directly to the local PLB bus, we used the Æthereal NoC as interconnect between the local PLB bus and the remote memory. The resulting system is illustrated in Figure 4.4. The Æthereal NoC consists of two NIs, shells and one router. In this system, like the simple SoC with remote memory, part of the application data resides in the remote memory. There are several configurations possible for the base SoC, depending on which target shell is used:

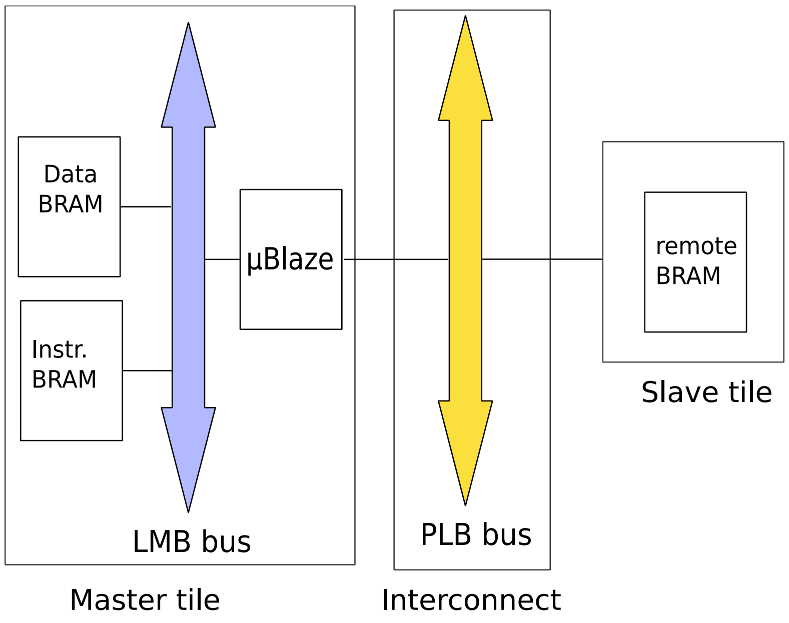


Figure 4.3: Simple SoC with remote memory

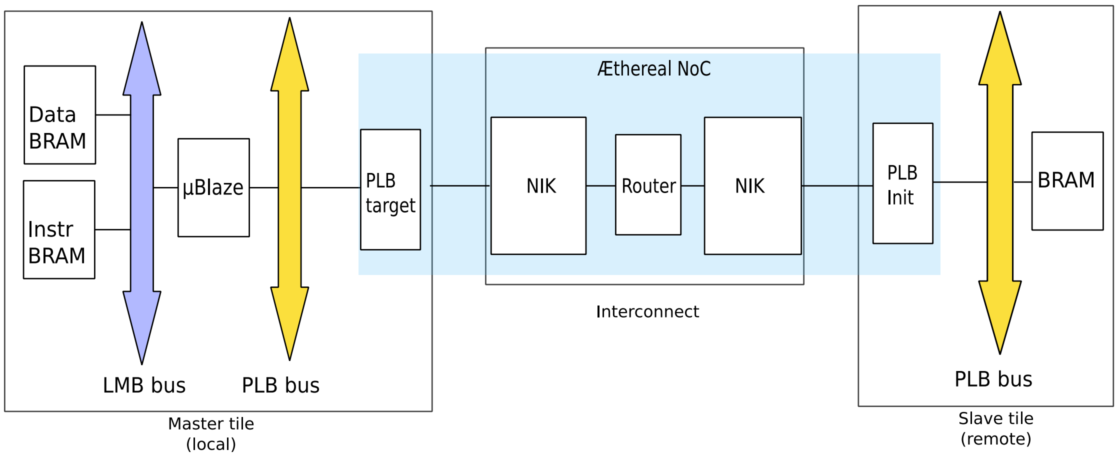


Figure 4.4: Base SoC

- Non-posted write, non-prefetch read version
- Non-posted write, prefetch read version
- Posted write, non-prefetch read version
- Posted write, prefetch read version

For the prefetch read, we used the FSL bus to prefetch the read data. Figure 4.5 illustrates the SoC with prefetch read. The utilization of the prefetch read expands the Æthereal with an additional NIK and FSL shell on the local side and an additional NIK and PLB initiator on the remote side.

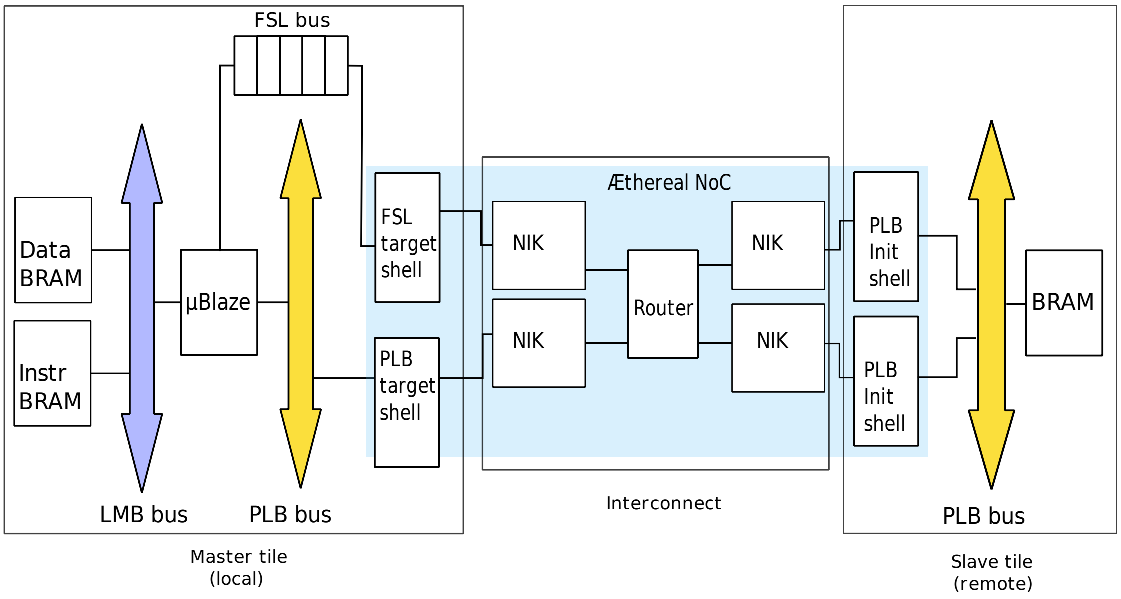


Figure 4.5: Base SoC with prefetch read

4.2.2 Optimized SoC

We also used the optimized version of the PLB target instead of the basic version. The optimized PLB target, when possible, automatically recognizes bursts. Figure 4.6 shows the system. The optimized PLB target only supports posted write transactions. No non-posted write transaction is possible. Again, like the simple SoC and the base SoC, part of the application data resides in the remote memory. Figure 4.7 illustrates the

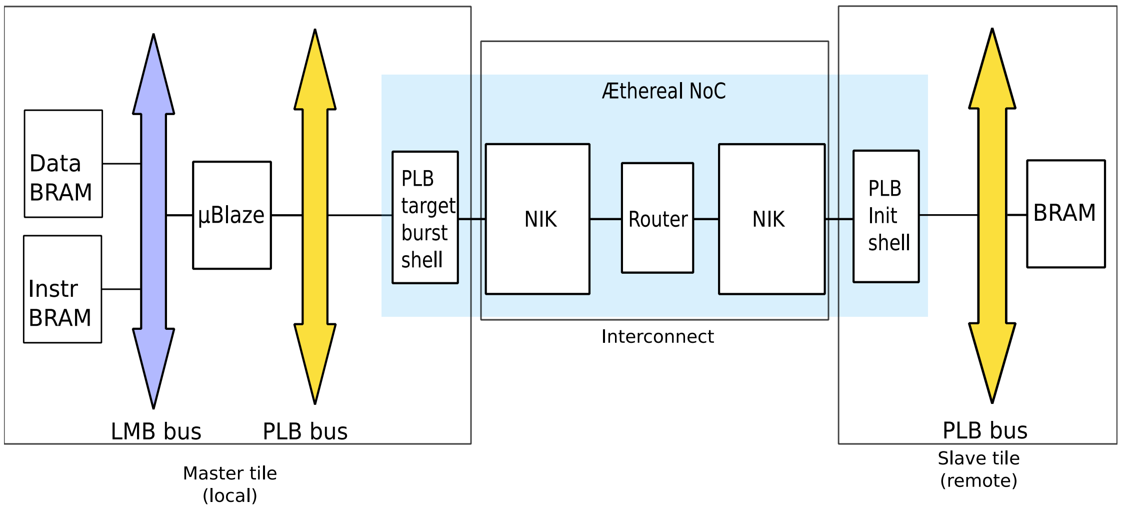


Figure 4.6: Optimized SoC

optimized system with the prefetch read. The read is performed by the FSL bus and the posted burst write is performed by the PLB target v2.

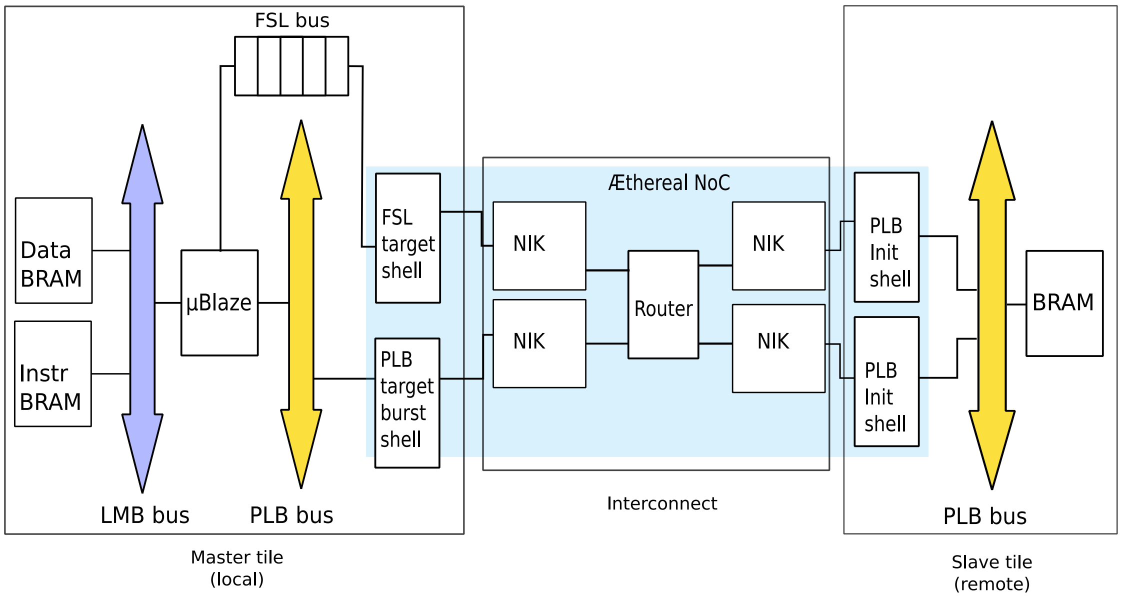


Figure 4.7: Optimized SoC with prefetch read

4.3 Software applications

We selected four software applications to run on the embedded system. They differ in the communication versus computation ratio and on the quantity of memory transactions performed. We modified each application to make them suitable to run on the SoC. All of the applications are written in C. We also created a test application in order to evaluate each feature of the shell separately. The applications are written for single processor implementation.

4.3.1 Test application

The code is included in Appendix A.5. The test application consists of two loops.

- Simple write loop: In this loop, a value is written to the remote memory in each loop iteration. The remote address is either consecutive or non-consecutive, depending on the preprocessor value defined.

- Simple read loop: In this loop, the value written to memory is read either with a non-prefetch read, a single prefetch read or a burst prefetch read.

We used additional code to verify whether the data being read is correct.

4.3.2 JPG decoder

The JPG decoder is an application that converts an image encoded according to the JPEG standard [9] into a bitmap image. It first starts by reading the JPG image from file into an input buffer in local memory. After that, it processes pieces of the image and then writes the result to the frame buffer located in the local memory. After the conversion is complete, the data from the frame buffer is written to a file. We made the following modifications to the application:

- The input buffer is located in the remote memory instead of the local memory.
- We do not read the image from a file into the input buffer. Instead, we load it with the program as an array of constant and then write it to the input buffer.
- The frame buffer is located in the remote memory instead of the local memory.
- Instead of writing the bitmap to a file, the bitmap is now sent to the serial port.

This application is computationally intensive, with a high ratio of computation to communication. The number of memory reads and writes performed depends on the JPG image size used.

4.3.3 Livermore loops

The Livermore loops [23] are a collection of kernel routines used in parallel computers benchmarks. They consist of a total of 24 kernels. We chose only four kernels to use for our experiments, because they need to be manually modified. We changed the data type of all the kernels from double to unsigned integer, as we instantiated the NoC with a word width of 32 bits. We modified the software as we described in Section 3.6. Appendix A.1 has the C code of the Livermore kernels.

4.3.3.1 Kernel 1: Hydro fragment

This routine consists of two loops, where equation 4.1 is computed.

$$x[k] = q + y[k] * (r * z[k + 10] + t * z[k + 11]) \tag{4.1}$$

Here x, y and z are one dimensional arrays and q, r and t are variables. We made the following modifications to the routine.

- We consider the case when the arrays x, y and z are placed in the remote memory and need to be accessed through the network.
- We modified the code to include the possibility to perform prefetch read, either single prefetch read or burst prefetch read.

4.3.3.2 Kernel 12: First difference

This routine consists of two loops, where equation 4.2 is computed.

$$X[k] = Y[k+1] - Y[k] \tag{4.2}$$

Here X and Y are one dimensional arrays. We made the following modifications to the routine.

- We consider the case when the arrays X and Y are placed in the remote memory and need to be accessed through the network.
- We modified the code to include the possibility to perform prefetch read, either single prefetch read or burst prefetch read.

4.3.3.3 Kernel 6: General linear recurrence equations

This routine consists of two loops, where equation 4.3 is computed.

$$W[i] = W[i] + B[k][i] * W[(i-k)-1] \tag{4.3}$$

Here W is a one dimensional array and B is a two dimensional array. We made the following modifications to the routine.

- We consider the case when the arrays W and B are placed in the remote memory and need to be accessed through the network. For the two-dimensional array B we defined a new data type as an unsigned integer vector. This guarantees that the two-dimensional array can be used in a similar way as the one-dimensional array.
- We modified the code to include the possibility to perform prefetch read. In this routine only single prefetch read is possible. The prefetch burst read was difficult to implement due to the fact that the inner loop has no fixed length but is dependent on the loop variable of the outer loop.

4.3.3.4 Matrix-Matrix multiplication

The matrix-matrix multiplication performs a product of two NxM matrixes, stored in the arrays PX, CX and VY. The routine consists of three loops. The inner-most loop performs the matrix-matrix multiplication. We made the following modifications to the application:

- We consider the case when the arrays PX, CX and VY are placed in the remote memory and need to be accessed through the network. For the two-dimensional arrays we defined a new data type as an unsigned integer vector. This guarantees that the two-dimensional array can be used in a similar way as the one-dimensional array.
- We modified the code to include the possibility to perform prefetch read. In this routine only single prefetch read is possible. The prefetch burst read is difficult to implement due to the fact that the read address is not sequential.

4.4 Experiment setup

In addition to the NoC related components, we also used a timer and the serial port for the experiments performed. Figure 4.8 shows the setup. We used the timer to precisely measure the cycle counts of the operations we are interested in. The serial port is used to send out the information collected during the execution of the C programs (print statements). We compiled all the software applications with the mb-gcc compiler -03" optimization level option. We did this because we are interested the performance of the system.

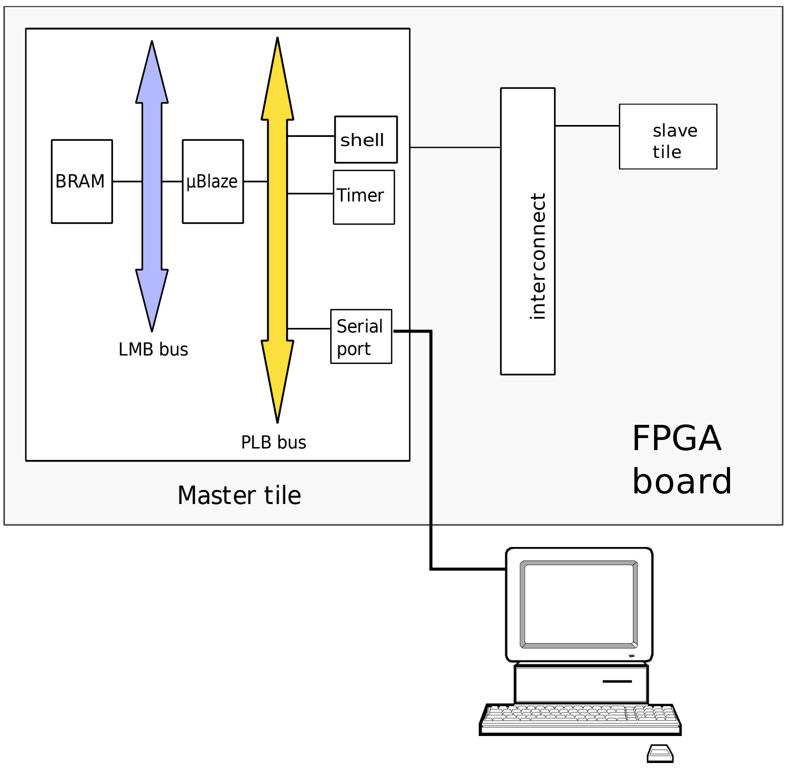


Figure 4.8: experiment setup

Furthermore, we made sure that the compiler does not optimize away the read and writes by placing print statements to the serial port before and after.

4.5 Conclusions

In this chapter we have described the hardware setups we are going to use to perform the experiments. Our goal is to verify the correct functionality of the shells and to evaluate the performance related indicators. We have also described the modification we made to the software applications, in order to make them run on the SoC we presented.

5 Experimental results and analysis

In this section we present the results of the experiments performed. We start with an overview of the SoC configurations. Then we elaborate the experiment parameters. After that we present and analyze the results obtained for the experiments. We end this section with concluding remarks.

5.1 SoC configurations

We experimented with the applications on all of the possible hardware combinations. Tables 5.1 - 5.2 show the configurations used to perform the experiments. On the SoC with the Æthereal NoC, we used GT for the application traffic. We use the following abbreviations:

- "$\surd$" indicates the link bandwidth used. We use the following BW: 100%, 50%, 25% and 12.5%
- "(n)pr" stands for (non) prefetch read
- "(n)pw" stands for (non) posted write

There are 64 configurations possible for the base SoC and 24 configurations possible for the optimized SoC. The total number of configurations possible is 88.

Table 5.1: Configurations for the base SoC

	HW configurations			
	Base: npr npw	Base: npr pw	Base: pr npw	Base: pr pw
Test application	$\surd$	$\surd$	$\surd$	$\surd$
JPG	$\surd$	$\surd$	$\surd$	$\surd$
Matrix vector mult	$\surd$	$\surd$	$\surd$	$\surd$
Livermore loops	$\surd$	$\surd$	$\surd$	$\surd$

5.2 Link bandwidth

We simulated the effects of having multiple processors in the system by varying the bandwidth allocated. The more processors in the SoC configuration, the less bandwidth

Table 5.2: Configurations for the optimized SoC

	HW configurations	
	Opt: npr pw	Opt: pr pw
Test application	$\surd$	$\surd$
JPG	$\surd$	$\surd$
Matrix vector mult	$\surd$	$\surd$
Livermore loops	$\surd$	$\surd$

each processor will get. A slot table of 33 slots is used, out of which 1 slot is used for configuration traffic and 32 slots are used for application traffic. We chose a slot table consisting of 33 slots, because it enables us to fine tune the bandwidth precisely enough for our experiments. The 32 slots for the application traffic to simulate a multi processor SoC configuration are used as follows.

- **BW: 100% → 1 processor**: occupies all slots in the slot table, all slots are assigned to the application traffic. That is 32 of the 33 slots available. The other slot is used by the configuration traffic.
- **BW: 50% → 2 processors**: 16 of 33 slots in the slot table are assigned to application traffic. Each processor effectively uses half of the available bandwidth
- **BW: 25% → 4 processors**: 8 of the 33 slots available are assigned to the application traffic. Each processor effectively uses a quarter of the available bandwidth.
- **BW: 12.5% → 8 processors**: 4 of the 33 slots available are assigned to the application traffic. Each processor effectively uses one eight of the available bandwidth.

The latency for each channel is tied to the channel bandwidth. Varying the channel bandwidth changes the channel latency. For example, the SoC configuration consisting of only 1 processor has all slots in the slot tables assigned for application traffic. Thus it can send data in each slot cycle. On the other hand, the SoC configuration consisting of 8 processors only has 4 slots reserved in the slot stable for application traffic. In the worst case, it has to wait for 29 slots before it can send data. The channel latency variation is still present if we had implemented a multiprocessor SoC configuration with fully filled slot table for each processor instead. In that situation, processors also need to contend for access to a single memory. The more processors in the SoC, the larger the latency will be. Moreover, the NoC serializes the bus protocols, instead of sampling the address, data and commands concurrently. This also introduces a latency.

5.3 Test application results

We used the test application in order to obtain detailed information of how each shell feature impacts the performance of the system. To accommodate all of the program data in the local memory, we increased the stack size section of the memory using a setting in XPS referred to as the project Executable and Linkable Format (ELF) file. The cycle count for the write and read transaction includes the transaction time itself, the time it takes to compute the next address, the time it takes to handle the loop iteration variables and the time it takes to access the timer attached to the PLB bus. Furthermore we experimented with the '-funroll-loops' option of the mb-gcc compiler. We did this to improve the performance of the processor. There is no data depence in the simple loops. With loop unrolling, the compiler optimizes the execution speed of the program by reducing loop test overhead [14]. First we used the base system to perform the experiments. After that we used the optimized system.

5.3.1 Posted write for base shell

This subsection details the results obtained from the base system configuration. First, we researched the effects of the posted write compared to non-posted write of the base shell. We did this by using the test application write loop and we measured the time in cycles it took to execute all of the writes. After that we performed experiments simulating a multiprocessor system. Figure 5.1 shows the results for the base write transactions. The results show that posted write improves the performance. The difference between

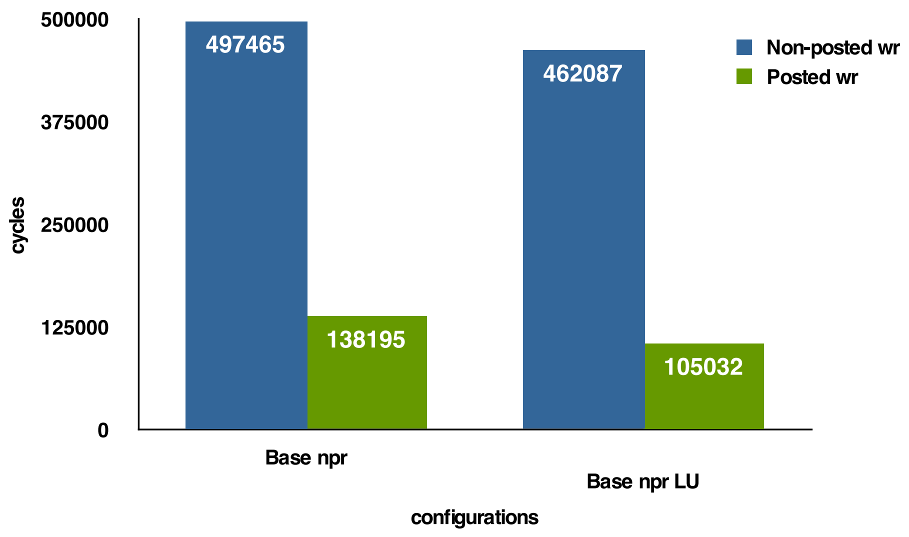


Figure 5.1: Base shell non-posted vs posted write

a posted write and a non-posted write is as follows. By using a posted write, the write transaction is not delayed by the roundtrip of the message from the shell to the memory. The only delay possible is between the NI shell and the NIK. This is the case when the input FIFO of the NIK or the buffer of the shell is full and it cannot accept new incoming messages yet. In this situation the base shell will stall, delaying the consequent write transactions until the NIK can accept messages again. The compilation of the source code with loop unrolling did increase the performance of the write transaction as expected.

5.3.1.1 Multi processor system

We simulated a multi-processor system with the base shell by varying the bandwidth and latency of the network communication channel. This was done by varying number of allocated the slots in the slot table of the NIK, as explained in section 5.1. Figure 5.2 shows the results obtained for the system of non-posted and posted writes. For the case of non-posted write, the performance deteriorates when the system consists of more processors. The decreased bandwidth effects the non-posted write transaction both when the request message travels form the base shell to the memory as from when the response message travels from the memory back to the base shell. The performance does not decrease linearly however, as the latency of the write transaction depends on the network conditions (slot available) and the status of the transaction (writing to memory). The posted write transaction shows less performance deterioration than the

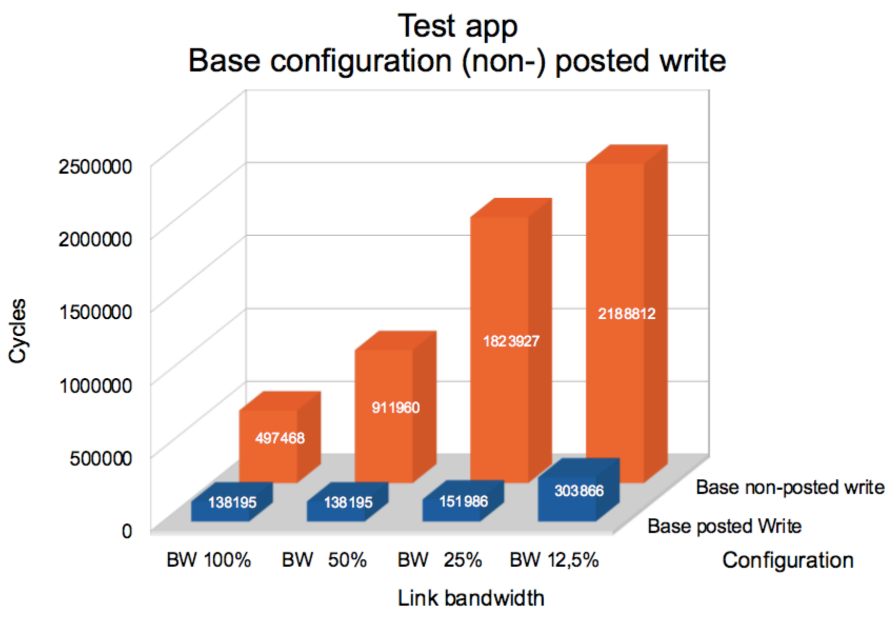


Figure 5.2: Base shell multiprocessor configuration for (non-) posted writes

non-posted write. This is because the posted write does not have to wait for the message to travel through the network. As long as the NIK FIFO is not full and the network transports the message fast enough to keep the FIFO filled but not full, the performance will stay stable (BW=100% and BW=50%). Eventually, as the bandwidth is decreased, the network can not keep up with the pace of the μBlaze, the NIK FIFOs become full and the performance starts to decrease (BW=25% and BW=12.5%).

5.3.2 Optimized shell burst write vs non-burst write

This subsection details the results obtained from the optimized system configuration. We were interested in the performance gain using the burst write, compared to non-burst write. The optimized shell automatically detects and performs burst transaction.

We performed experiments to determine whether the burst transaction does increase the performance as expected. We did this by using the test application and performing writes to consecutive and non-consecutive addresses in the remote memory. We also used in our experiment both the unrolled and the rolled loop. Figure 5.3 shows the results. The results show that the performance for both write transaction without the

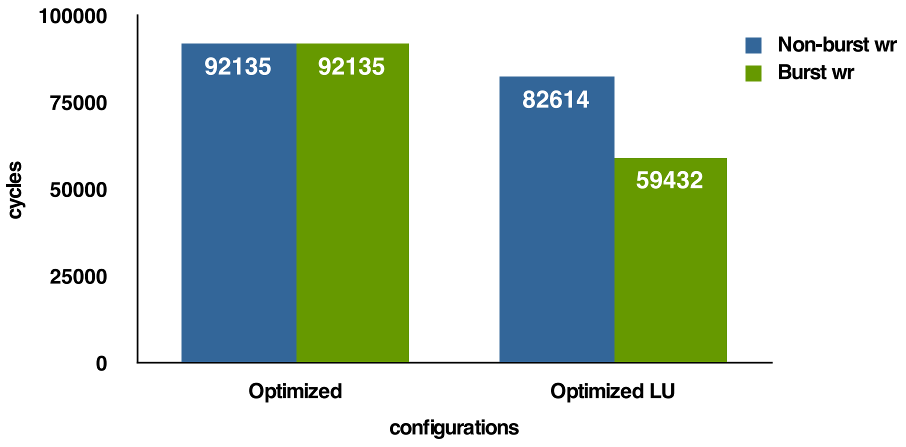


Figure 5.3: Burst write vs Non burst write

loop unrolling do not differ. The μBlaze cannot generate consecutive write transaction quick enough to take advantage of the burst transaction capability of the shell. The NoC consumes the messages faster than the μBlaze produces write transactions. This way the NoC is idle when the next transaction is ready and the burst transaction will not start, as intended. To demonstrate this we captured the waveforms of the simulation for consecutive address. Figure 5.4 shows the result for the optimized shell without loop unrolling. The 'plb_pa_valid' line indicates whether there is a valid write transaction

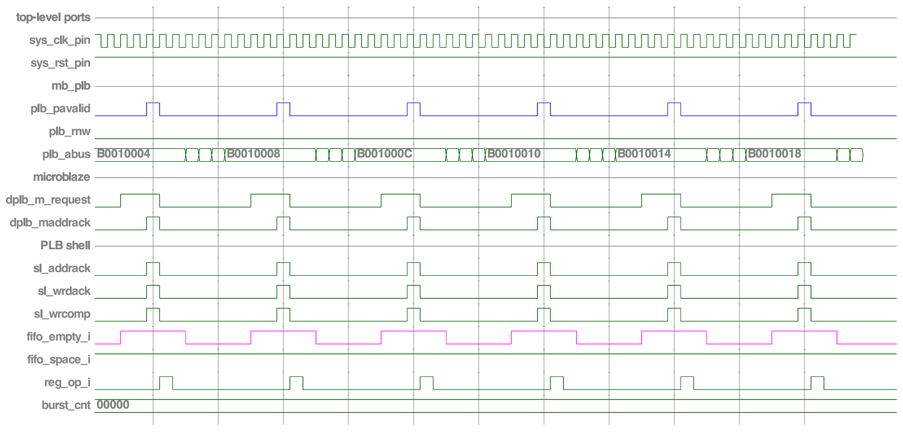


Figure 5.4: Simulation waveform for the optimized shell without loop unrolling

on the bus. The 'fifo_empty_i' line indicates whether the PLB shell FIFOs are empty. Each time there is a write transaction ('plb_pa_valid' line is '1'), the FIFOs are empty ('fifo_empty_i' line '1'). This indicates to the shell that the NoC is idle and that it must utilize the network. This way the burst mechanism is not activated.

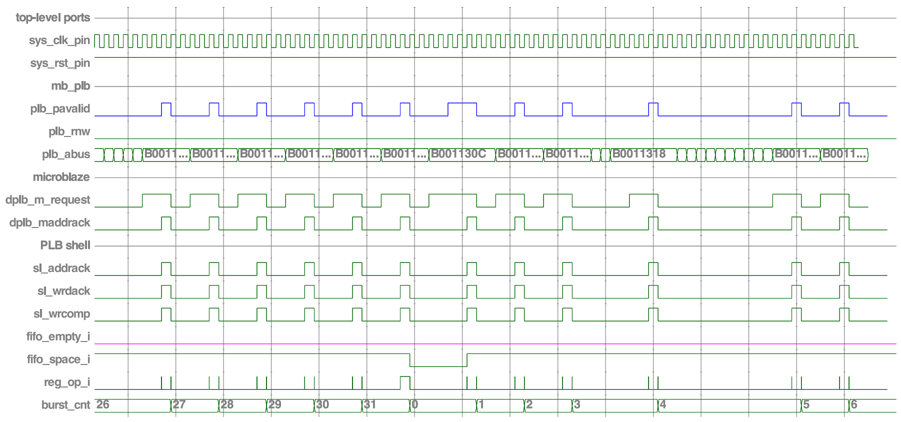


Figure 5.5: Simulation waveform for the optimized shell with loop unrolling

Figure 5.5 shows the waveform for the optimized shell with loop unrolling. The line "dplb_m_request" of the μBlaze is the transaction request line. It is immediately clear from the figure that the μBlaze in this case emits the write request faster than in the case without loop unrolling. The result is that when there is a valid transaction ('plb_pa_valid' line is '1') the shell is not empty ('fifo_empty_i' line is '0'). The burst mechanism kicks in (burst_cnt line) and the burst transaction is automatically performed, resulting in an improved performance.

5.3.2.1 Multi processor system write performance

As for the base system, we performed simulations for systems consisting of more than one processors. We experimented with the burst write and non-burst write without loop unrolling. Figure 5.6 shows the result. For the non-burst case, the performance stays stable for BW=100% and BW=50%. This due to the fact that the network can process the messages fast enough to avoid performance deterioration. For BW=25% and BW=12.5%, the performance decreases significantly. This is due to the fact that the NIK FIFOs and the shell FIFOs start to fill up and the network does not have enough bandwidth to process the messages fast enough to avoid the drop in performance. The performance decreases when the FIFOs are full and the shell stalls the write transactions.

For the burst case, the situation is different. For BW=50%, the burst mechanism starts to work because the shell FIFOs are not always empty anymore. This results in short burst as the network is able to frequently empty the shell FIFOs. The net effect of this is that the performance stays at the same level as for BW=100%. For BW=25%, the network is not able to process the messages fast enough anymore, similar to the non-burst case. The network performance starts to deteriorate, causing the shell FIFOs to be constantly filled. The result of this is that the shell starts performing long burst writes. The advantage of burst write is that fewer message headers need to be sent to the network. For example, for a burst transaction of length 32, the shell sends 34 total

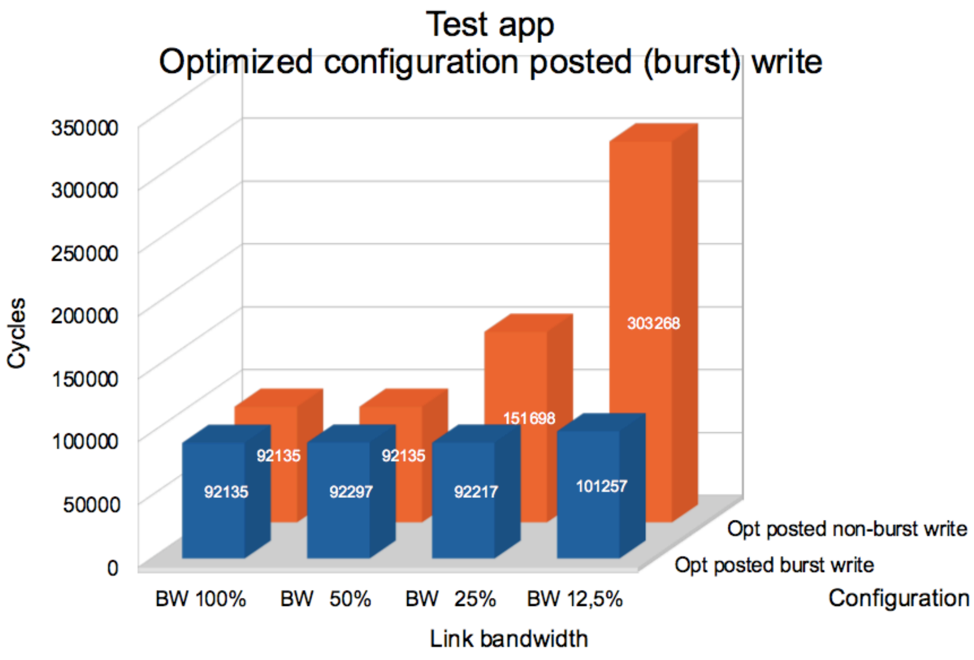


Figure 5.6: Optimized shell multiprocessor configuration for (non-) burst writes

messages to the network (2 headers, 32 data messages). A non-burst transaction for 32 data words requires a total of 96 messages (64 headers, 32 data messages) to be sent. For BW=25% the drop in bandwidth is compensated by the fact that fewer headers need to be sent. The net result is that the performance stays stable and does not drop as sharp as in the non-burst case. A similar analysis holds for the BW=12.5% case. The drop in bandwidth is compensated by the reduction of messages. However, the impact of less bandwidth is bigger than the reduction in messages. This results in a more performance decrease than for the BW=25% case.

5.3.3 Read prefetch

Next we researched the effects of the read prefetch on the performance. It must be noted that this option requires changes to the software. The experiments were performed using prefetch read with the FSL shell, compared to the non-prefetch read base shell version. For the application, the read loop was used. We performed the experiments for two types of read prefetch.

- Single read prefetch: We started by prefetching a certain number of data words in order to fill up the FIFO initially. After that, each loop prefetches another data word from memory. This way the FIFO link always has data readily available. The following code snippet shows this technique.

```
for(i=0, i<LOOP_MAX, i++)
{
        if (i==0)
4                        for(j=0;j<4;j++)
5                                putdfslx(Y+j,0,FSL_DEFAULT);  //Y[i]
6                if (i<LOOP_MAX-4)
7                        putdfslx(Y+i+4,0,FSL_DEFAULT);//Y[i]
8                getdfslx(temp_Y,0,FSL_DEFAULT);
9                X[i] = temp_Y + 100 ;
10       }
```

Assume that 'Y' is a pointer to the remote memory location. In the first iteration, line ③, prefetch 4 data words from memory (line ④ and ⑤). After that, in ⑥ and ⑦, for each loop iteration, prefetch a data word from memory. Continue up until Y[LOOP_MAX] is reached. Next, get the data word from the FIFO (⑧) and store it in temp_y. Lastly, use the prefetch data word in a calculation, ⑨, where X is an array of integers.

- Burst read prefetch: We used the prefetch to perform burst read memory transactions. We did use the maximum possible burst size allowed when performing the burst transaction. The following code snippet shows this technique.

```
1        for(i=0, i<LOOP_MAX, i++)
2        {
3                if ( i==0 || i%32==0 )
4                {
5                        burst=(LOOP_MAX-i >31)?31:(LOOP_MAX%i)-1;
6                        putdfslx(burst,0,FSL_CONTROL);
7                        putdfslx(Y+i,0,FSL_DEFAULT);
8                }
9                getdfslx(temp_Y,0,FSL_DEFAULT);
10               X[i] = temp_Y + 100 ;
11       }
```

Assume that 'Y' is a pointer to the remote memory location. First we determine the burst size ⑤, set the burst size ⑥, and perform the prefetch ⑦. We always use the maximum burst size possible. The maximum burst size allowed is 32 words. Next, in ⑨, we fetch the data from FSL. Lastly, in ⑩, we use the prefetch data word in the calculation, where X is an array of integers

Figure 5.7 shows the results obtained for both the single and burst prefetch read. The read prefetch improves performance by:

- Overlapping the read transaction with the computation. The read transaction is active while the program is executing other non-read instructions.
- Pipelining the read transactions. Each read transaction is acknowledge immediately. The read transactions do not have to wait for the completion of the previous active read transactions.

The results shows that the best performance is obtained with the prefetch burst read, using the loop unroll option for compilation. The disadvantage of the single prefetch is

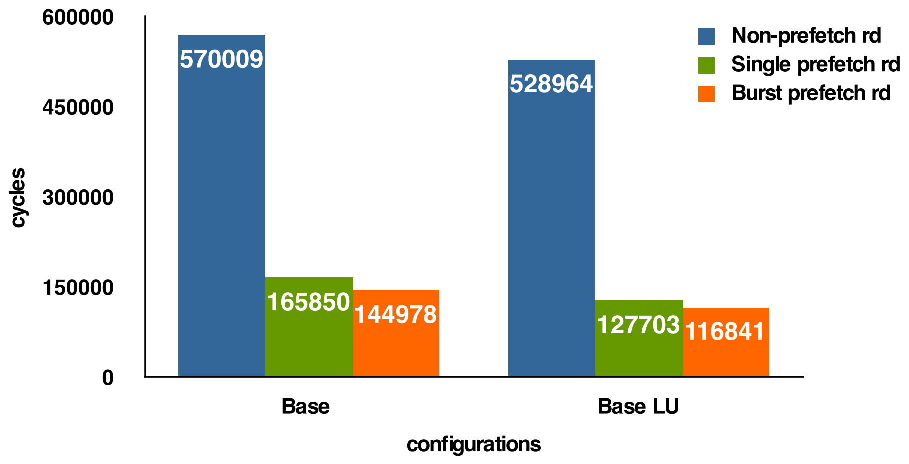


Figure 5.7: Base shell non-prefetch read vs prefetch read

that it has to perform a read request to the FSL shell in each loop iteration. The burst prefetch only has to request data once every certain amount of loop iteration. Loop unrolling improves the performance of the memory reads.

5.3.3.1 Multi processor system read performance

Figure 5.8 shows the results for the non-prefetch and single prefetch read transaction. For the non-prefetch read, the performance deteriorates as the system consists of more

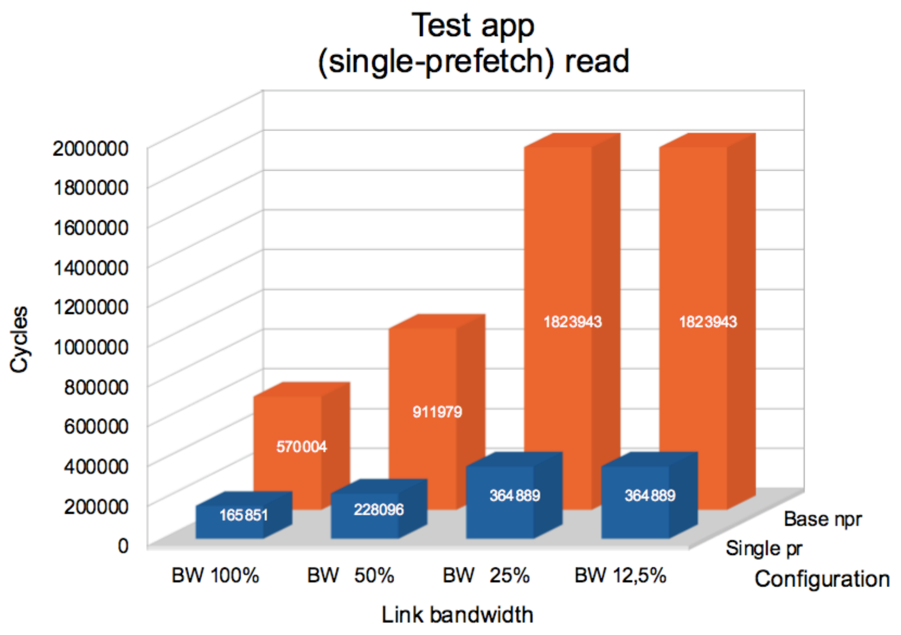


Figure 5.8: Base shell multiprocessor configuration for (non-) prefetch reads

processors up until when the link bandwidth is 25%. After that, the latency in sending back credits becomes dominant and the bandwidth reduction has no more influence on the performance. The same situation happens for the prefetch read. As the bandwidth is decreased, the latency becomes dominant for link bandwidth is 25%.

5.3.4 Difference in result between the base shell and the optimized shell

The write transaction results for the base shell differ from the results obtained from the optimized shell. The optimized shell has a better performance than the base shell. Apart from the automatic burst detection mechanism of the optimized shell, the two implementation of the shell differ in other areas. The optimized shell uses a FIFO to store the out going messages, instead of a shift register as in the case of the base shell. Also the optimized shell acknowledges, in case of a write, the address and the transaction in the same cycle, while the base shell has a delay of 2 cycles between the address acknowledgement and the write transaction acknowledgement. These differences explain the different results obtained for both shells. Figure 5.9 shows the waveform of the posted write version of the base shell. We see that there is a 14 cycles delay between

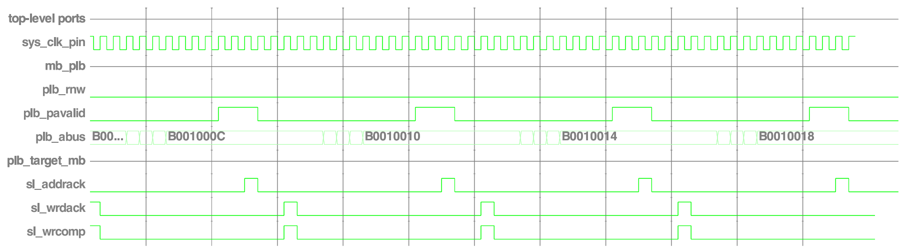


Figure 5.9: Wave form for the posted write base shell without loop unrolling

two consecutive writes (sl_wrcomp='1'). From figure 5.4 we see that the delay between two consecutive writes is 9 cycles.

5.4 JPG decoder

The JPG decoder is characterized as a computation intensive application, i.e. the communication to computation ratio is very low. There is no depence between read and write data. In this application more write transactions are performed than read. We experimented on the base system and on the optimized system with the JPG decoder.

5.4.1 Base system results

Figure 5.10 shows the results for the base configuration. All of the four base configurations have a similar performance for the JPG decoder. For all, the difference between performance is within 5.7%. This is due to the fact that the JPG decoder spent more cycle on computation than on communication. Even with the addition of a dedicated link for prefetching read does not impact the performance significantly. The figure shows, as

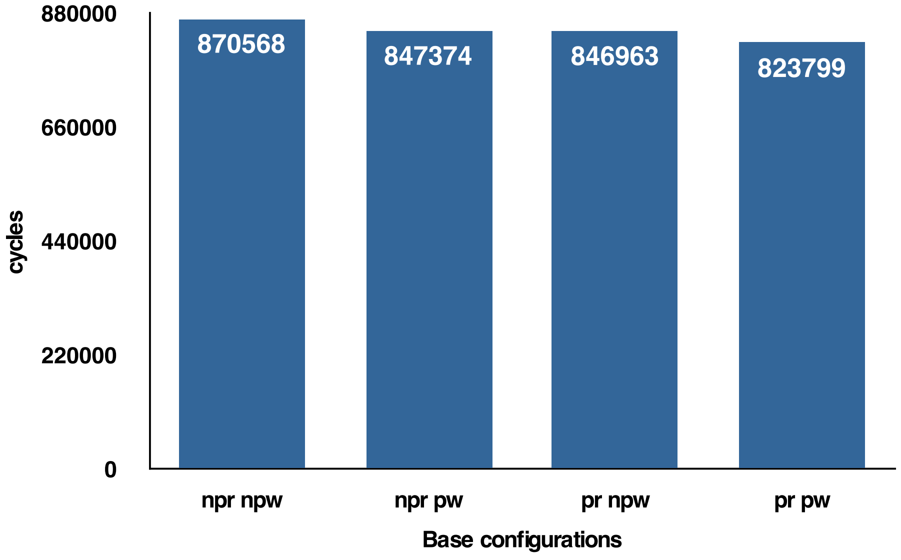


Figure 5.10: JPG decoder with the base shell

expected, that the performance is slightly improved with the usage of posted write and read prefetch

5.4.1.1 Base multi processor system read and write performance

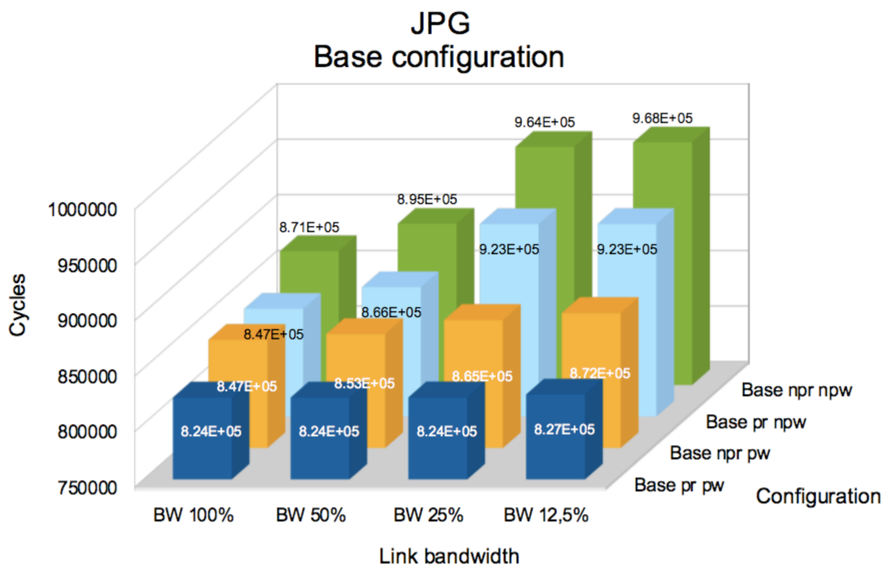


Figure 5.11: Base shell multiprocessor configuration for the JPG decoder

We also experimented with varying the bandwidth for the JPG application to simulate a multi processor setup. Figure 5.11 shows the result. The 'npr npw', 'npr pw' and 'pr npw' configurations show a general trend that the performance decreases when there is less bandwidth available, as expected from section 5.3. The performance for the 'pr pw' configuration remains stable. This is due to the fact that the read and write transactions are not frequent compared to the computation performed. This enables the posted write and the read prefetch to sustain a stable performance even when the bandwidth is decreased. In other words, there is sufficient time between the read and write transactions for the network to process the transaction traffic. Another interesting difference is between the 'pw' and the 'npw' configuration. This difference illustrates the advantage of using posted write when the write transactions dominate the read transactions. Although all configuration have similar performance when the full bandwidth is allocated for application traffic, they differ when the bandwidth is decreased. The penalty for waiting for a write transaction to return with reduced bandwidth even offsets the benefit of having a separate communication channel for reading. The worst case difference in performance (BW=12.5%) is 17% compared to 5.7% for the best case (BW=100%).

Table 5.3: Improvement for JPG by using Loop Unrolling for the base shell

	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	
BW 100%	2.80%	2.89%	2.88%	2.97%	**JPG**
BW 50%	2.87%	3.03%	2.81%	2.97%	
BW 25%	2.48%	2.81%	2.62%	2.97%	
BW 12.5%	2.65%	2.96%	2.62%	2.94%	

5.4.2 Optimized system results

Figure 5.12 shows the result for the optimized shell. As was the case with the base shell, the result obtained for both posted write and non-posted write do not show a large difference. Again, the application computation dominates the communication.

5.4.2.1 Optimized shell multi processor system read and write performance

Figure 5.13 shows the result for the various link bandwidths. For the prefetch read optimized shell, the performance stays at the same level. As previously explained, the limited number of writes and reads compared to the computation enables the posted shells to sustain a stable performance. For the non-posted read optimized shell, the performance slightly decreases as the read transaction is affected by the link bandwidth.

5.5 Livermore loops

We experimented on the various configurations with the livermore loops. We measured the amount of cycles for all the loop iterations. Table 5.5 shows the number of read and write transaction per iteration for the livermore loops, as well as the total number of iterations in the loop.

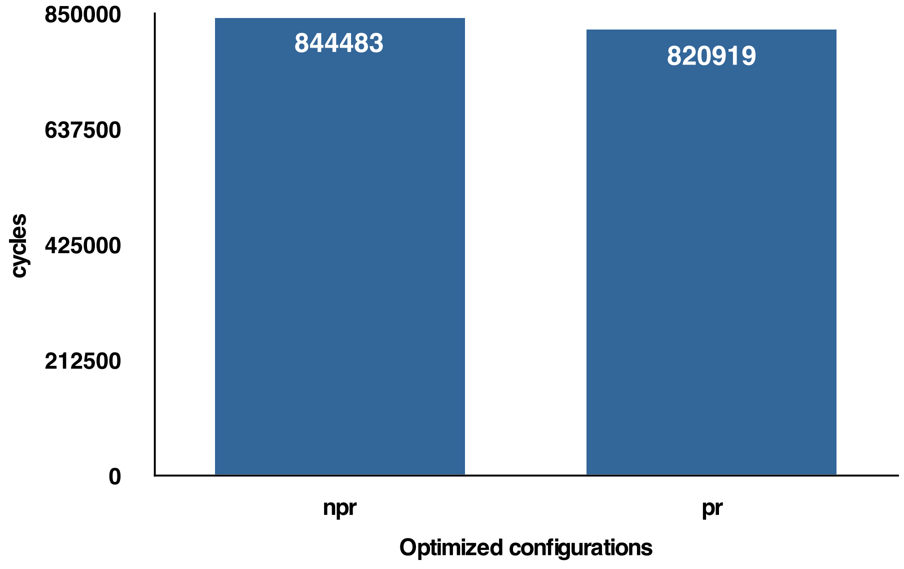


Figure 5.12: Non-posted and posted write for npr base shell

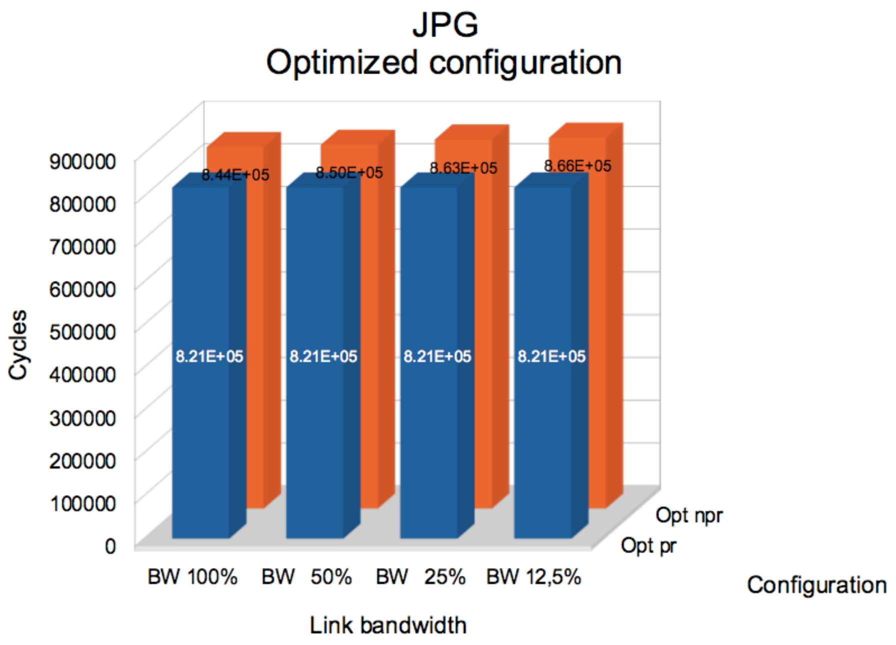


Figure 5.13: Optimized shell multiprocessor configuration for the JPG decoder

We researched the combined effects of the various shell features on the performance of the kernel routine. The burst is not considered for the Livermore Loops. The reason for this is that the burst reads cannot be performed in an efficient way. Contrary to the case of the test application, the loops in the Livermore kernels are more complicated, due to:

Table 5.4: Improvement for JPG by using Loop Unrolling for the optimized shell

	Opt pr	**Opt npr**	
BW 100%	2.98%	2.89%	**JPG**
BW 50%	2.98%	3.03%	
BW 25%	2.99%	2.83%	
BW 12.5%	2.98%	2.93%	

Table 5.5: Livermore loop information

	#Reads/iteration	**#Writes/iteration**	**#Loop iterations**
K1	3	1	7007
K12	2	1	12000
K6	3	1	< 900
K21	2(3)	1	63125

- variable number of loop iterations. This complicates the calculation of the burst size.
- non-consecutive data words locations in the memory. This limits the use of the burst, as the burst can only access consecutive word from the memory.
- multiple variables to fetch from memory. The problem with this issue is, that the variable data words cannot be used at the same instance. This is due to the fact that we store all the prefetch data in one FIFO.

An example of the problem with burst read and multiple variables:

```
    for(i=0, i<LOOP_MAX, i++)
    {
        if ( i==0 || i%31==0 )
        {
            burst=(LOOP_MAX-i >31)?31:(LOOP_MAX%i)-1;
            putdfslx(burst,0,FSL_CONTROL);
            putdfslx(Y+i,0,FSL_DEFAULT);
            putdfslx(Z+i,0,FSL_DEFAULT);
            for  (j=0;j<burst+1;j++)
                getdfslx(Y_burst[j],0,FSL_DEFAULT);
            for  (j=0;j<burst+1;j++)
                getdfslx(Z_burst[j],0,FSL_DEFAULT);
            Y_pnt=Y_burst;
            Z_pnt=Z_burst;
        }
        X[i] =(*Y_pnt) + (*Z_pnt);
```

```
17                  Y_pnt++;
18                  Z_pnt++;
19         }
```

In this example, data is retrieved from 2 remote memory location. Assume that 'Y' and 'Z' are pointers to remote memory locations. First we determine burst size and perform the prefetch read, in line ⑤ to ⑧. After that, we get the data words from the FSL in lines ⑨ to ⑫ . We store the burst data words in an array in the local memory. This is necessary, as we perform two burst transactions and we use the data words of the two transactions at the same time, like in ⑯ . As we have two arrays in the local memory containing the read data, we assign pointers to iterate through them, in lines ⑬ and ⑭. Lastly, we use the prefetch data word in a calculation (line ⑯) and update the pointers to point at the next data word (lines ⑰ and ⑱).

A work around for this issue is to use more FIFOs to store the response data. This way each variable has its own buffer. The disadvantage of this approach is that the whole NoC must be regenerated for this purpose. Also, this workaround is only possible for limited number as variables, as the μBlaze can only support 8 FSL link pair. Due to these issues, we decided not to perform burst prefetch read with the Livermore Loops. In the following subsections we present the results of the experiment and we also comment on the results obtained.

5.5.1 K1 Hydro fragment base configuration

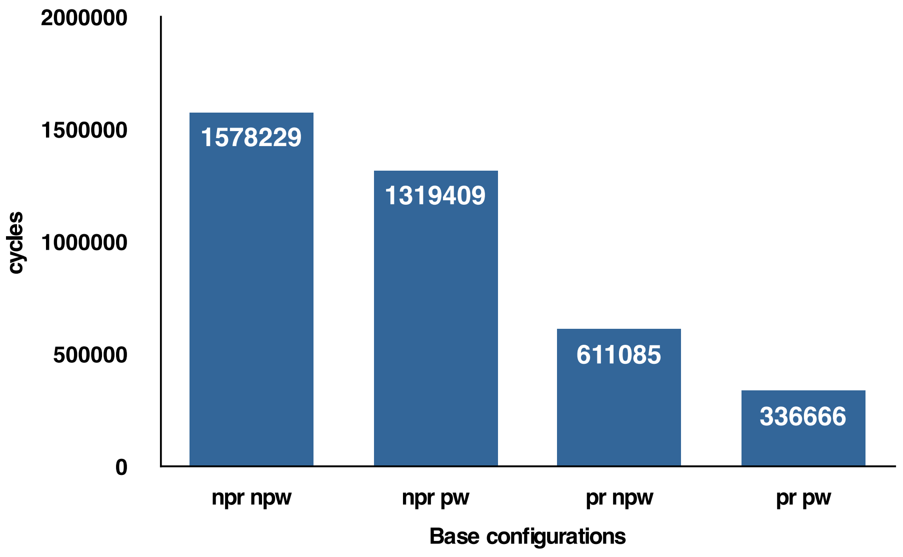


Figure 5.14: K1 hydro fragment results for the base shell BW=100%

This application performs three reads and one write transaction for each loop iteration. The read and writes have no depence. Figure 5.14 shows the results. The performance gain from using read prefetch is greater than posted write, as expected. This due to the fact that more reads than writes are performed for each loop iteration. The rest of the results are also as expected. Using posted write and read prefetch improves the performance.

5.5.1.1 Base shell multi processor system read and write performance

Figure 5.15 shows the results for the multiprocessor configuration. The benefits of a dedicated read channel for a read intensive application is clear from this figure. As

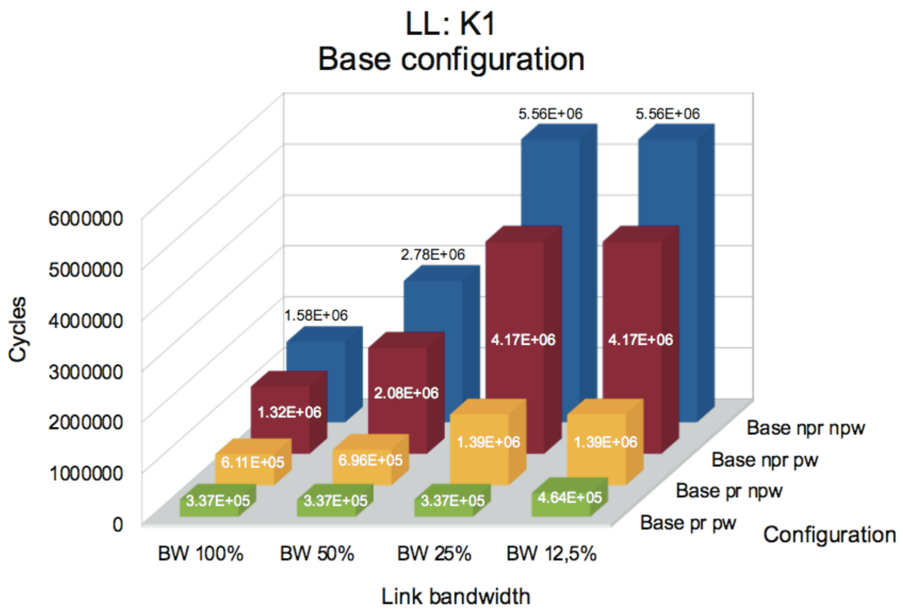


Figure 5.15: Base shell multiprocessor configuration for K1 Hydro fragment

the bandwidth decreases, the non-prefetched configurations clearly show a much greater performance deterioration than the prefetched read configurations. The performance of the 'pr pw' is fairly stable for BW=100% to BW=25%. This due to the fact that the network can process the transactions quick enough to avoid the performance decrease. The performance of the 'pr pw' configuration starts to deteriorate for BW=12.5%. At this point the network cannot keep up with the transactions, resulting in a performance decrease. For 'npr npw', 'npr pw' and 'pr npw' configurations, the latency in sending back credits becomes dominant at BW=25%. Table 5.6 shows the performance improvement obtained with using loop unrolling. For 'npr npw', 'npr pw' and 'pr npw' configurations, the performance is only improved when 100% link bandwidth is utilized. This is due to the fact that for BW=100%, the μBlaze is the bottleneck. With loop unrolling, the performance of the μBlaze is increased. With the BW <100%, the μBlaze is not the bottleneck anymore. The result of this is that there is no performance improvement. The case of the 'pr pw' configuration is different. For BW>12.5%, the μBlaze is the bottleneck. The situation changes when for BW=12.5%. Then the interconnect is the bottleneck.

Table 5.6: Improvement for K1 by using Loop Unrolling for the base shell

	Base npr npw	Base npr pw	Base pr npw	Base pr pw	K1
BW 100%	1.01%	2.31%	3.17%	4.33%	
BW 50%	0.00%	0.00%	0.00%	4.40%	
BW 25%	0.00%	0.00%	0.00%	3.82%	
BW 12.5%	0.00%	0.00%	0.00%	-0.01%	

5.5.2 K1 Hydro fragment optimized configuration

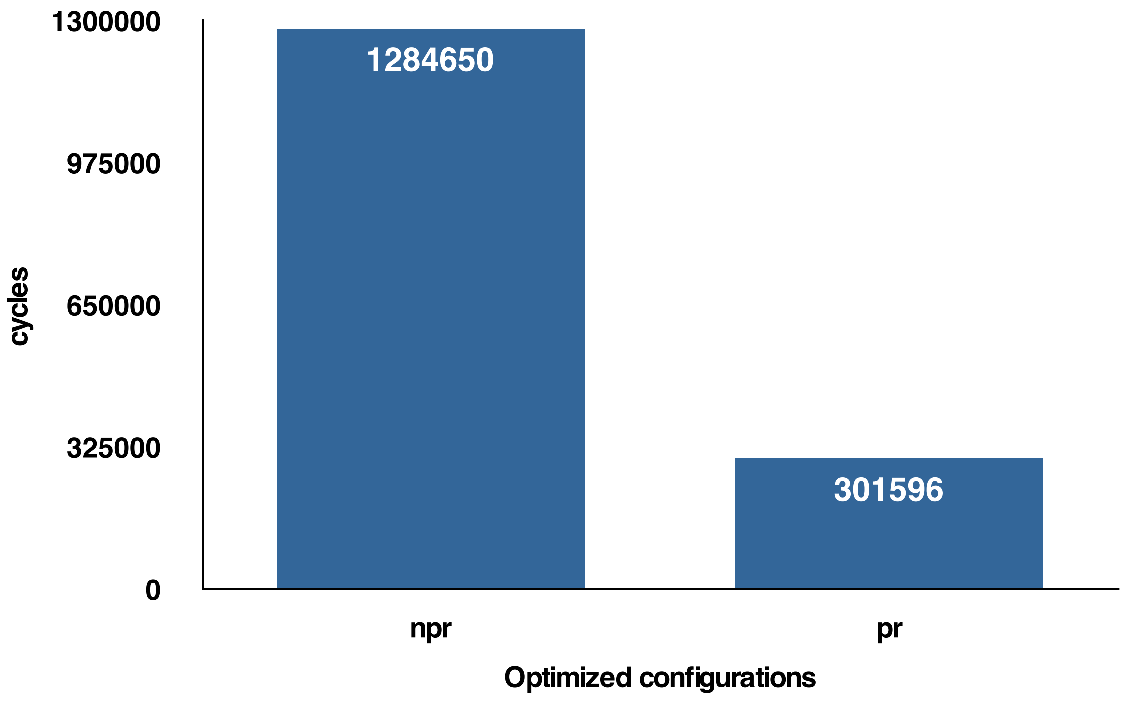


Figure 5.16: Optimized shell K1 results BW=100%

Figure 5.16 shows the results for the optimized shell. The results further show the benefit of the prefetch read for this routine. An performance increase of 326% is obtained with the use of the dedicated read link.

5.5.2.1 Optimized shell multi processor system read and write performance

Figure 5.17 shows the results for the multiprocessor experiment with the optimized shell. The 'npr pw' configuration is clearly affected by the lack of a dedicated read link. Like the base configuration case, the performance starts to decrease for BW=50% and the latency in sending back credits becomes dominant at BW=25%. The 'pw npr' configuration also shows a similar behavior as in the base configuration case. For BW=100%, BW=50%

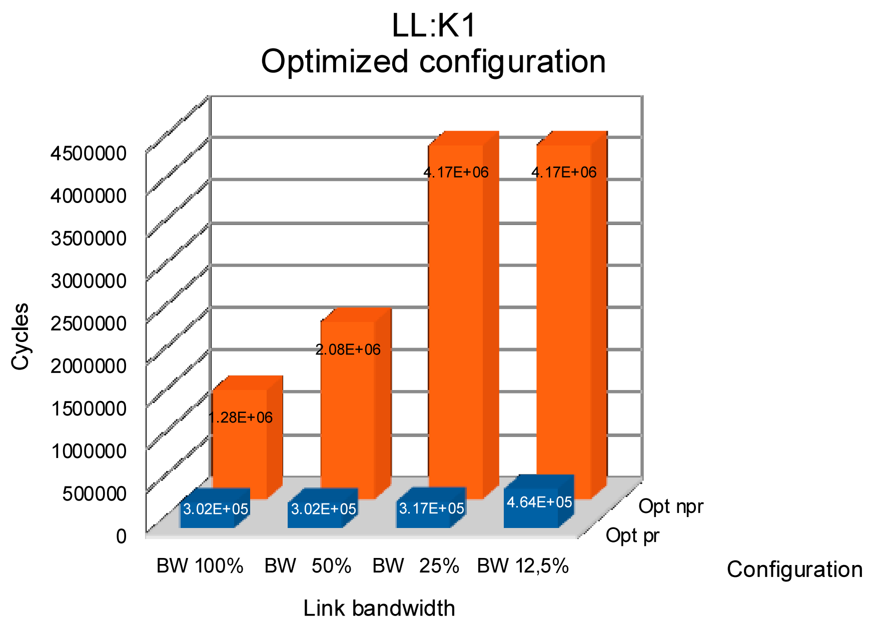


Figure 5.17: Optimized shell multiprocessor configuration for K1 Hydro fragment

and BW=25% the performance stays fairly stable. The performance starts to decrease for BW=12.5%.

Table 5.7: Improvement for K1 by using Loop Unrolling for the optimized shell

	Opt pr	**Opt npr**	
BW 100%	4.86%	1.45%	**K1**
BW 50%	4.78%	0.00%	
BW 25%	2.22%	0.00%	
BW 12.5%	0.01%	0.00%	

Table 5.7 shows the performance improvement obtained with using loop unrolling. The performance only improves if the μBlaze is the bottleneck (BW=100% for 'Opt npr' and BW=100%, 50% or 12.5% for 'Opt pr' configurations). When the interconnect is the bottleneck, there is no performance improvement by using loop unrolling.

5.5.3 K12 base configuration

This application performs two reads and one write transaction in each loop iteration. There is no depence between the reads and the writes transactions. Figure 5.18 shows the results for the base configurations. Again, like for the K1 Hydro fragment, the best performance is obtained with read prefetch, as more reads than writes are performed.

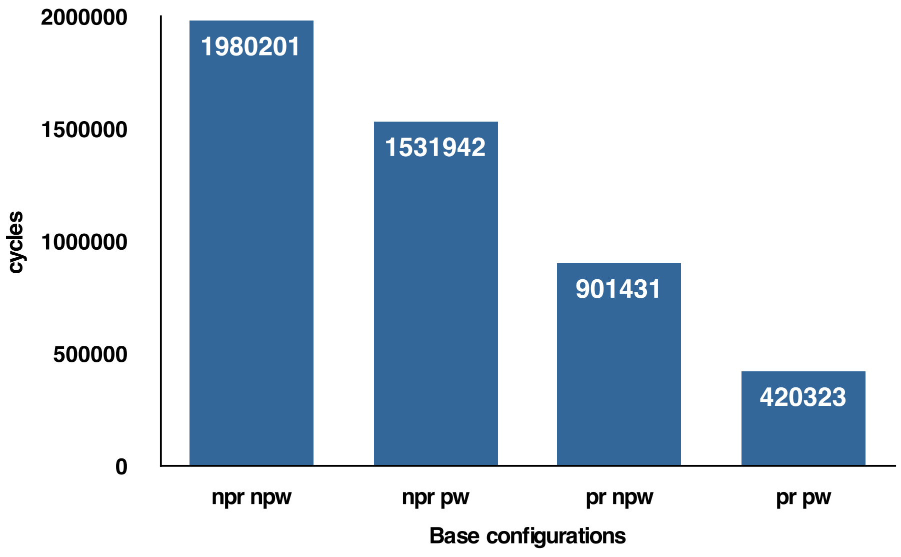


Figure 5.18: K12 base configuration results BW=100%

5.5.3.1 Base shell multi processor system read and write performance

Figure 5.19 shows the results for a multi processor setup. The figure shows similar characteristic as the hydro kernel multiprocessor configuration.

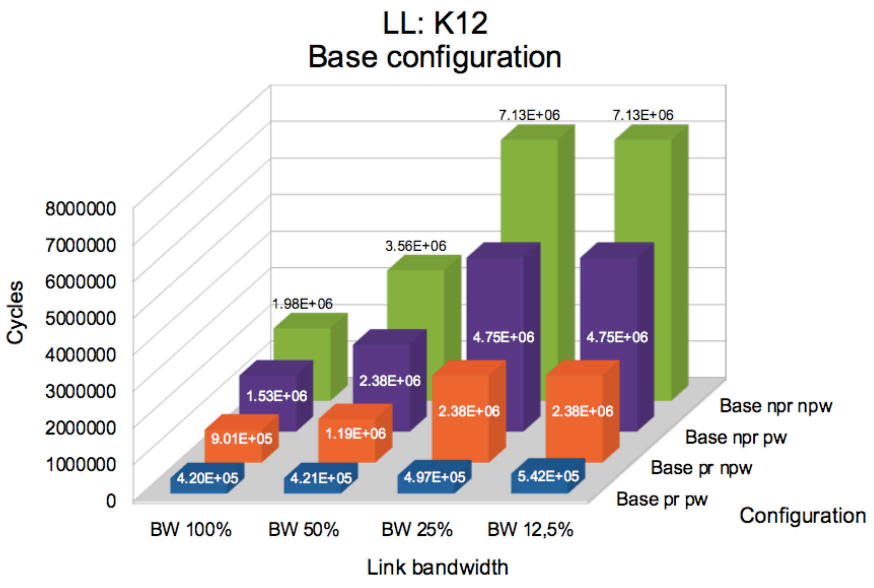


Figure 5.19: Base shell multiprocessor configuration for K12

Table 5.8 shows the performance improvement obtained with using loop unrolling. Again, for 'npr npw', 'npr pw' and 'pr npw' configurations, the performance is only improved when 100% link bandwidth is utilized, as the μBlaze is the bottleneck. For the

Table 5.8: Improvement for K12 by using Loop Unrolling for the base shell

	Base npr npw	Base npr pw	Base pr npw	Base pr pw	K12
BW 100%	1.27%	3.16%	6.18%	8.53%	
BW 50%	0.00%	0.00%	0.00%	7.75%	
BW 25%	0.00%	0.00%	0.00%	4.24%	
BW 12.5%	0.00%	0.00%	0.00%	2.25%	

'pr pw' configuration, the use of loop unrolling results in performance improvement for the all the link bandwidth used. Unlike for the K1 kernel, the μBlaze is the bottleneck even for BW=12.5%.

5.5.4 K12 optimized configuration

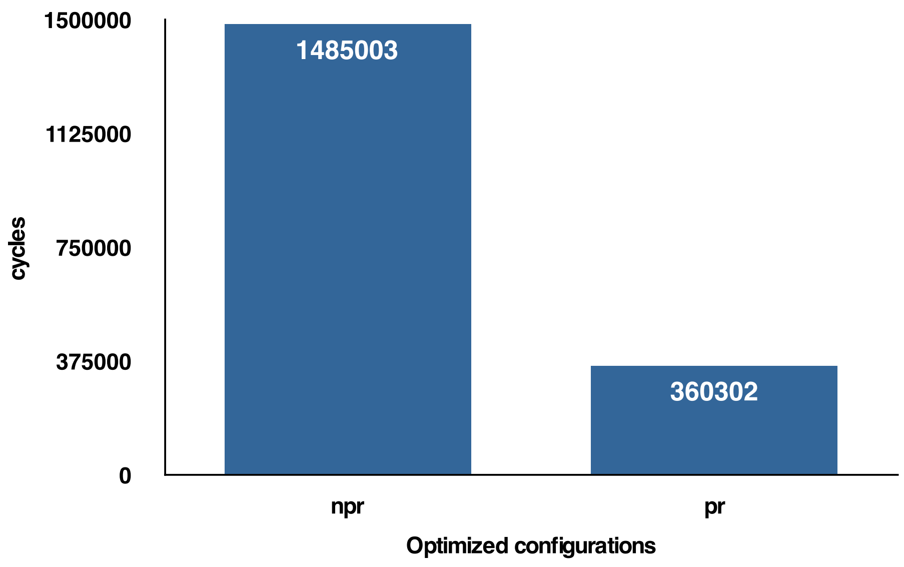


Figure 5.20: Optimized shell K12 results BW=100%

Figure 5.20 shows the results obtained for the the optimized shell. Again, due to the fact that the routine performs more reads than writes, the prefetch read improves the performance significantly. A performance increase of 312% is obtained when the dedicated prefetch link is used.

5.5.4.1 Optimized shell multi processor system read and write performance

Figure 5.21 shows the results for a multi processor setup. The figure shows a similar characteristic as the K1 hydro kernel multiprocessor configuration.

Table 5.9 shows the performance improvement obtained with using loop unrolling. The performance only improves if the μBlaze is the bottleneck (BW=100% for 'Opt

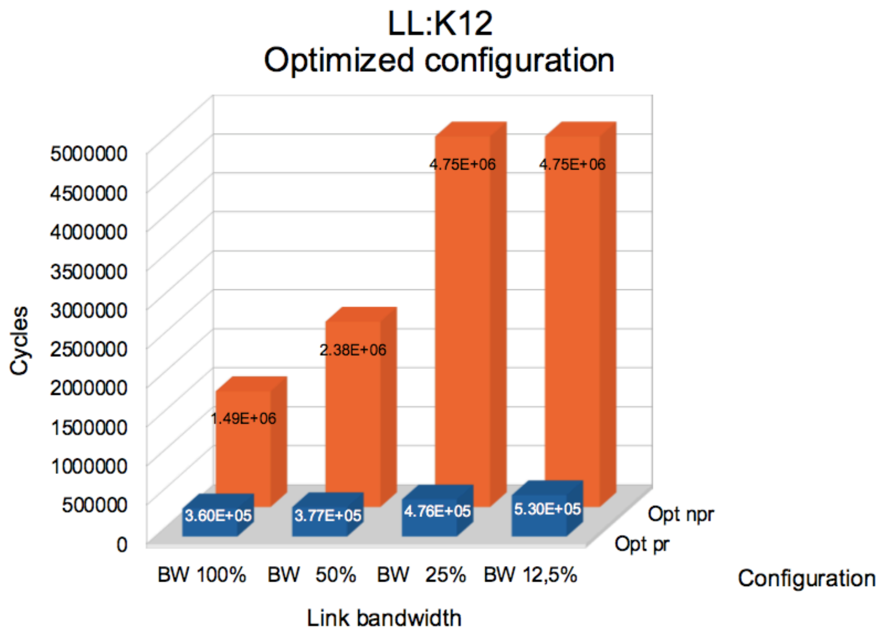


Figure 5.21: Optimized shell multiprocessor configuration for K12

Table 5.9: Improvement for K12 by using Loop Unrolling for the optimized shell

	Opt pr	**Opt npr**	
BW 100%	10.07%	2.27%	**K12**
BW 50%	1.25%	0.00%	
BW 25%	-0.01%	0.00%	
BW 12.5%	-0.01%	0.00%	

npr' and BW=100% and 50% for 'Opt pr' configurations). When the interconnect is the bottleneck, there is no performance improvement by using loop unrolling, like in the previous cases.

5.5.5 K6

Figure 5.22 shows the results obtained for the the base shell. This application performs three reads and one write transaction in each loop iteration. Unlike in the previous applications, there is a dependency between the writes and the reads. The write must complete before the read transaction in the next loop iteration begins.This limits the benefit of using the posted write or prefetch read. Figure 5.18 shows the results for the base configurations. The impact of the dependence is noticeable for the 'npr pw' and 'pr npw' configuration. Both have similar performance, with the difference being within 3.3%. This in contrast to the 59% to 70% difference in performance for the K1 and K12 routine.

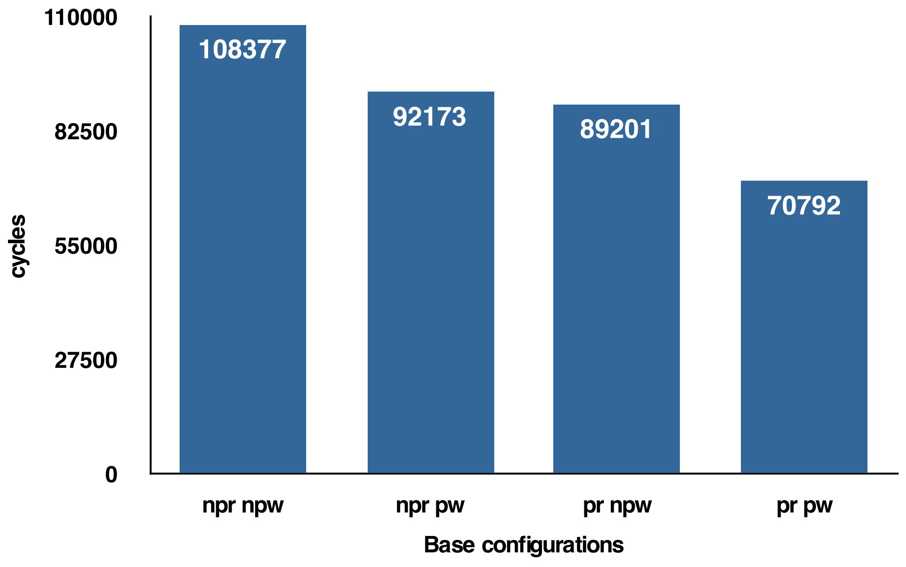


Figure 5.22: K6 base configuration results BW=100%

5.5.5.1 Base shell multi processor system read and write performance

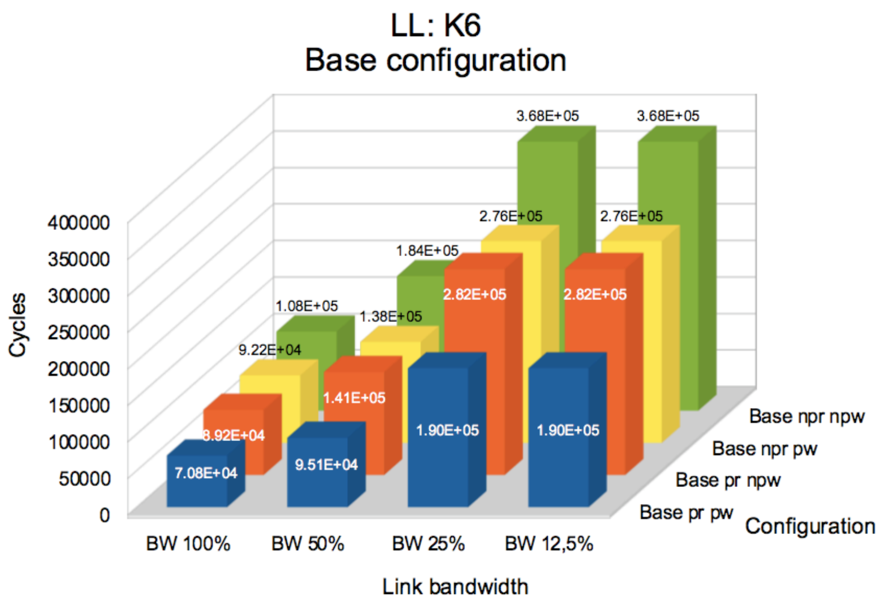


Figure 5.23: Base shell multiprocessor configuration for K6

Figure 5.23 shows the result for the multiprocessor configuration. Again, due to the dependence between the reads and writes, the configurations for 'npr pw' and 'pr npw' have a similar performance for the various link bandwidths. The 'pr pw' configurations show a different characteristic than in the previous routines. The performance deteriorates for BW=50% and the latency in sending back credits becomes dominant at

BW=25% and BW=12.5%.

Table 5.10: Improvement for K6 by using Loop Unrolling for the base shell

	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	**K6**
BW 100%	0.06%	2.20%	-0.05%	-0.37%	
BW 50%	0.02%	-0.05%	0.02%	0.04%	
BW 25%	-0.02%	0.01%	0.02%	0.02%	
BW 12.5%	0.01%	0.01%	-0.02%	0.02%	

Table 5.10 shows the performance improvement obtained with using loop unrolling. Unlike the case for the K1 and the K12 kernels, the effect of using loop unrolling is limited. The main reason for this is that number of loop iterations in this for-loop is variable. It ranges from '1' to '30' iterations. Again, due to this, the effect of loop unrolling is limited, as the benefit of using it is to reduce the end-of-loop test.

5.5.6 K6 optimized configuration

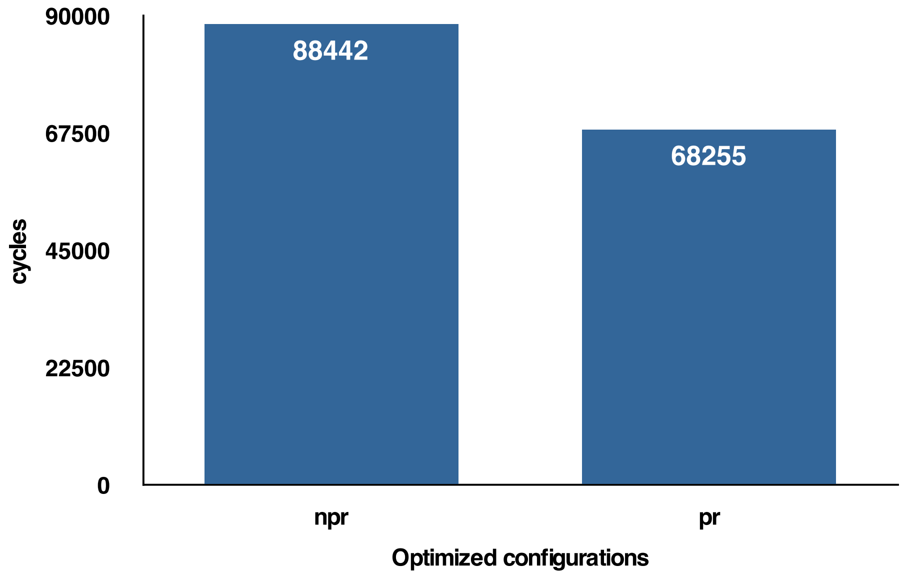


Figure 5.24: Optimized shell K6 results BW=100%

Figure 5.24 shows the result for the optimized shell. Again, similar to the case in of the base shell and unlike the previous kernels, the difference between non-prefetch read and prefetch read is small.

5.5.6.1 Optimized shell multi processor system read and write performance

Figure 5.25 shows the result for the multiprocessor configuration. The same analysis as in the case of multi processor configuration for the K1 and K12 kernels, is applicable for

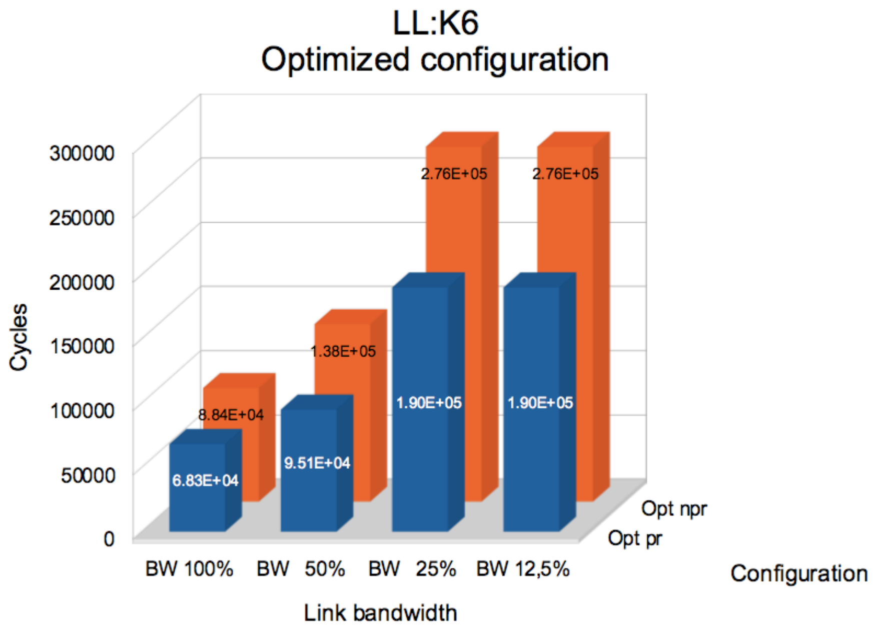


Figure 5.25: Optimized shell multiprocessor configuration for K6

this configuration.

Table 5.11 shows the performance improvement obtained with using loop unrolling. Again, the effect of loop unrolling is limited, due to the variable number of loop iterations.

Table 5.11: Improvement for K6 by using Loop Unrolling for the optimized shell

	Opt pr	**Opt npr**	
BW 100%	-1.60%	0.00%	**K6**
BW 50%	0.03%	0.03%	
BW 25%	0.02%	0.01%	
BW 12.5%	0.02%	0.01%	

5.5.7 K21

This application consists of two inner loops where two reads and one write transaction is performed in the innermost loop and one read transaction in the other inner loop. There are no dependence between reads and writes in the application. Figure 5.26 shows the results for the base configurations. The results shows similar characteristic for the configurations as was obtained with the K1 and K12 kernels.

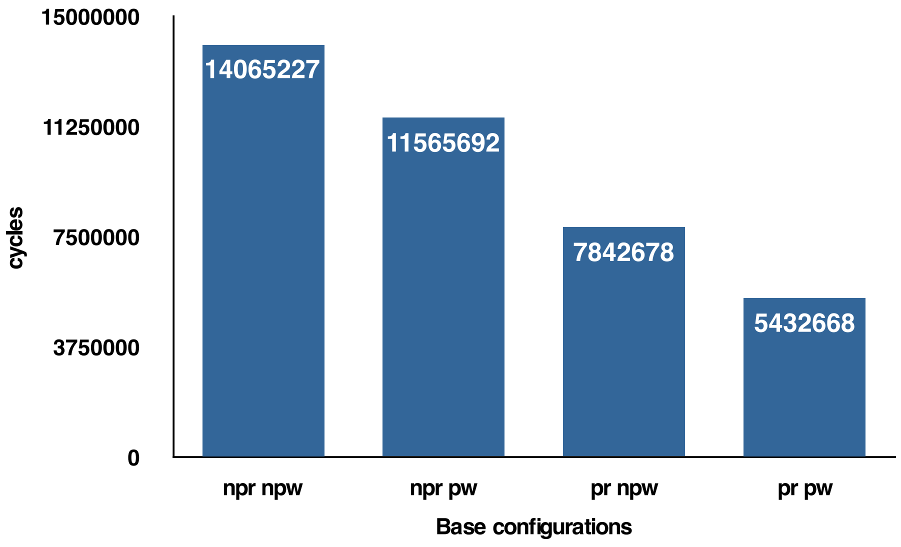


Figure 5.26: K21 base configuration results BW=100%

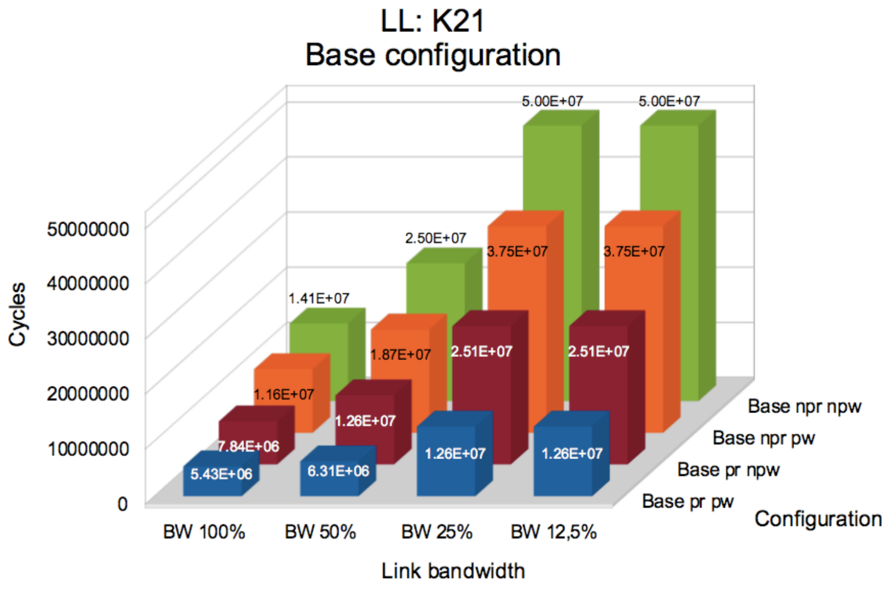


Figure 5.27: Base shell multiprocessor configuration for K21

5.5.7.1 Base shell multi processor system read and write performance

Figure 5.27 shows the result for the multiprocessor base configuration. Unlike for the case of the the K1 and K12 kernels, the performances of all the configurations do not deteriorate anymore at a link bandwidth of 25%. This is due to the large number of loop iterations in this kernel.

Table 5.12 shows the performance improvement obtained with using loop unrolling. Again, for 'npr npw', 'npr pw' and 'pr npw' configurations, the performance is only

Table 5.12: Improvement for K21 by using Loop Unrolling for the base shell

	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	**k21**
BW 100%	3.80%	2.79%	0.19%	1.08%	
BW 50%	0.00%	0.00%	0.00%	0.00%	
BW 25%	0.00%	0.00%	0.00%	0.00%	
BW 12.5%	0.00%	0.00%	0.00%	0.00%	

improved when 100% link bandwidth is utilized, as the μBlaze is the bottleneck. For the 'pr pw' configuration, the use of loop unrolling results in performance improvement for link BW=100%, 50% and 25%.

5.5.8 K21 optimized configuration

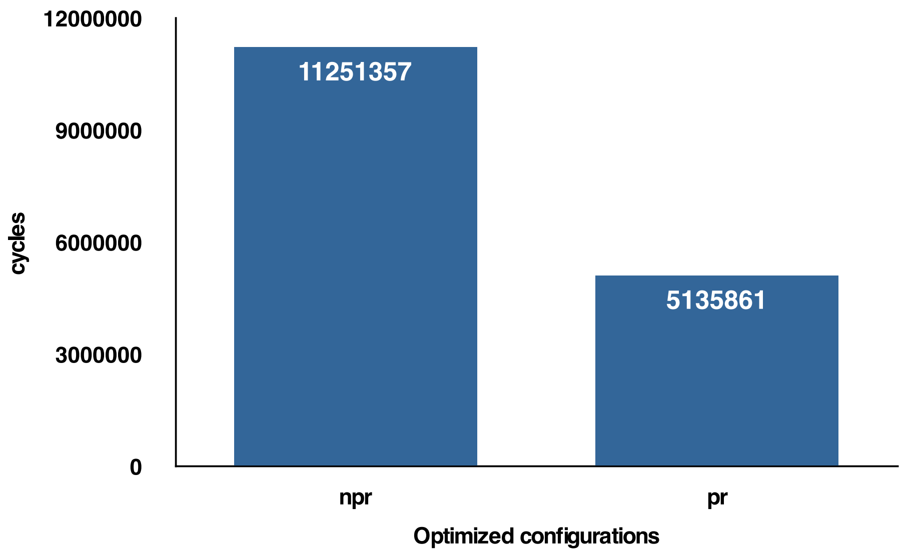


Figure 5.28: K21 optimized shell configuration BW=100%

Figure 5.28 shows the result for the multiprocessor base configuration. The same analysis as in the case of multi processor configuration for the K1 and K12 kernels, is applicable for this configuration.

5.5.8.1 Optimized shell multi processor system read and write performance

Figure 5.29 shows the result for the multiprocessor base configuration. Again, like the multiprocessor base configuration, the latency in sending back credits becomes dominant at link bandwidth of 25%.

Table 5.13 shows the performance improvement obtained with using loop unrolling. The loop unrolling only improves the performance when the link bandwidth is 100%. For

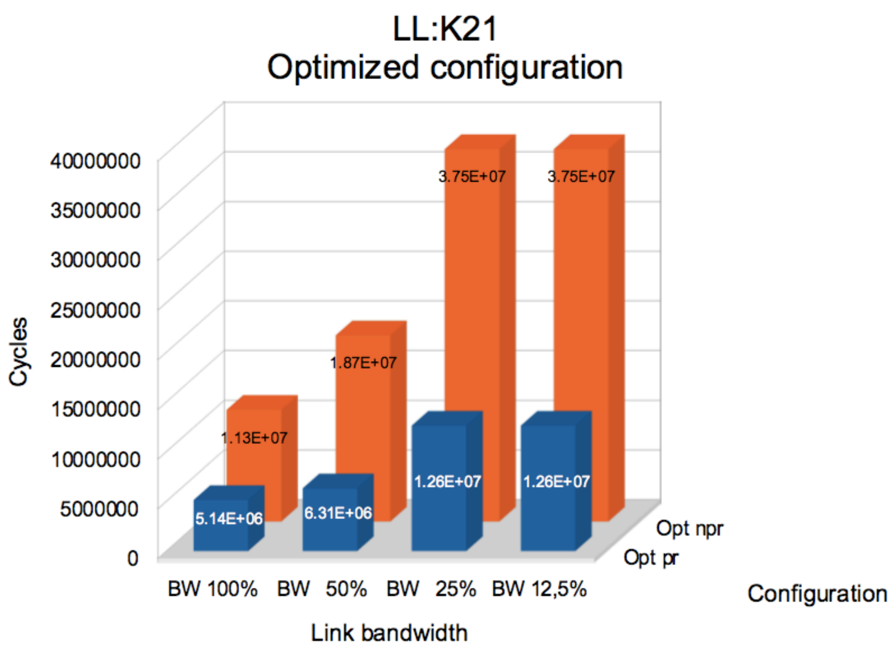


Figure 5.29: Optimized shell multiprocessor configuration for K21

the other link bandwidths, the interconnect is the bottleneck. The use of loop unrolling for those situations does not result in a performance increase.

Table 5.13: Improvement for K21 by using Loop Unrolling for the optimized shell

	Opt pr	**Opt npr**	
BW 100%	1.72%	1.87%	**K21**
BW 50%	0.00%	0.00%	
BW 25%	0.00%	0.00%	
BW 12.5%	0.00%	0.00%	

5.6 Conclusions

We presented the results of the various experiments performed. Furthermore, we analyzed the following factors.

- **Posted write**: improves the performance of the SoC. This is due to the fact that the write transactions do not have to wait for the acknowledge response to return from the network. The disadvantage of the posted write is that synchronization issues must be taken into account.

- **Prefetch read**: improves the performance of the SoC. This is achieved by overlapping the read transaction with the computation and pipelining the read transactions. The disadvantage of the prefetch read is that synchronization issues must be taken into account and that the software needs to be modified.

- **Burst transaction**: both the read and the write burst do improve the performance of the SoC. For the burst write, the improvement is achieved when the link bandwidth is decreased. This because less header words of a request message needs to be sent to the network. For the burst read, the performance improves due to the fact that the prefetch instruction is not executed in each iteration. However, a burst read is not always possible or does not always results in performance increase. This depends on the complexity of the application code. A workaround to overcome some of the issues concerning the prefetch burst read is to use more FSL shells, but this is not always desirable or possible.
- **Application Computation-to-Communication ratio**: has an impact on the effect of the shell on the performance of the SoC. The techniques used by the shell to improve performance only has an effect when the Computation-to-Communication ratio is low. For an application with a high Computation-to-Communication ratio, the usage of both the optimized shell as the base shell results in a small performance difference.
- **Loop unrolling**: with loop unrolling, the compiler optimizes the execution speed of the program by reducing loop test overhead. This results in an improved performance of the SoC, as long as the μBlaze is the bottleneck. Loop unrolling has its limitations as it cannot always be performed. For example, when the number of loop iterations is not constant, the use of loop unrolling does not improve the performance, even when the μBlaze is the bottleneck. Also when the μBlaze is not the bottleneck, loop unrolling does not add any improvements.
- **Synchronization**: the correct execution of an application needs to be verified when the prefetch read or the posted write is used. This because the bus interconnect does not support these features.
- **Latency**: depends on several factors. First, the type of application used has a great effect on the latency. Second of all, depending on the shell used, the latency may be improved. Last, the bandwidth affects the latency. A low allocated bandwidth and correspondingly a low number of slots results in a increased latency.
- **Link bandwidth**: affects the SoC performance. In general, when the link bandwidth is reduced, the performance of the SoC decreases. Burst write is able to sustain the performance of the shell up to certain point, by reducing overheads.

The results of the experiment show that the best performance is achieved for:

- base shell: prefetch read, posted write implementation.
- optimized shell: prefetch read version implementation.

Of these two, the optimized shell gives the best performance.

6 Conclusions

6.1 Summary

In this thesis we have researched and implemented protocol conversions for interconnecting existing IPs, which uses various protocols, with the Æthereal NoC. We accomplished this by creating network interface shells, both for encoding the bus protocols into Æthereal request messages and for decoding Æthereal response messages into bus protocols. Table 6.1 gives an overview of these shells.

Table 6.1: Protocol conversions

Component	**Protocol conversion**
OPB target shell	OPB protocol → Æthereal request message
	Æthereal response message → OPB protocol
OPB init shell	Æthereal request message → OPB protocol
	OPB protocol → Æthereal response message
PLB target shells	PLB protocol → Æthereal request message
	Æthereal response message → PLB protocol
PLB init shell	Æthereal request message → PLB protocol
	PLB protocol → Æthereal response message
FSL target shell	FSL protocol → Æthereal request message
	Æthereal response message → FSL protocol

The result of this work is that we have two versions of the PLB target shell (base and optimized), as consequence of different design considerations and features supported. Furthermore we have researched the possibility of improving the performance, both by coping with the transaction latency associated with memory access as well as increasing the throughput. This is necessary due to the fact that long transaction latencies have a negative impact on the performance of IPs. Table 6.2 gives an overview of the techniques we employed to improve the performance.
The challenges and difficulties we faced regarding these techniques was, that none of them are supported by the master IP, the μBlaze. First of all, we had to implement these features. We accomplished this task as follows. For posted write, we included support for this in both the OPB shell as in the PLB shell. For burst transactions, we designed a mechanism to automatically coalesce multiple transaction into a burst transaction when possible and implemented it in the optimized version of the PLB shell. For the prefetch

Table 6.2: Impact of several techniques on performance

Technique	*Effect*
Posted write **Prefetch read**	**Pros**: improves the performance of the SoC **Cons**: synchronization issues must be taken into account
Burst transaction	**Write burst** **Pros**: can improve the performance of the SoC **Cons**: Improves the performance only when the link bandwidth is decreased.
	Read burst **Pros**: can improve the performance of the SoC **Cons**: not always possible or does not always result in performance increase

read, we used a separate port on the μBlaze, build a custom unit to support the DMA transfer (FSL shell), and used a dedicated link to support and perform this transaction. Secondly, we implemented all these features without affecting the correct functioning of the system. The concern was to avoid synchronization issues, especially for prefetch read in combination with posted write configurations, due to the fact that we acknowledge the transaction before it is actually completed. To address this issues, we designed a simple component to prevent incorrect behavior due to outstanding write transactions.

We have also researched and analyzed in great detail the effects of several factors on the performance. Table 6.3 gives an overview of these factors.

Table 6.3: Impact of several factors on performance

Factor	*Effect*
C/C ratio	**High C/C ratio**: small performance difference between the optimized shell and the base shell **Low C/C ratio**: big performance difference between the optimized shell and the base shell
Loop unrolling	**Pros**: loop unrolling improves the performance of the SoC, as long as the μBlaze is the bottleneck. **Cons**: cannot always be performed. Also when the μBlaze is not the bottleneck, loop unrolling does not add any improvements.
Link bandwidth	performance of the SoC decreases when the link bandwidth is reduced

Also, we have performed a large number of experiments, using several applications, in order to analyze the combined effect of the several advanced techniques and factors on the performance of the SoC. We have focused on the PLB shell, as we are interested in the best performance possible. For the case when the link bandwidth is reduced, we concluded that:

- the latency, associated with sending back credits, becomes dominant and the link bandwidth reduction less influence on the performance

- a burst transaction can sustain the performance of the system, up to a certain point

Furthermore, the results of the experiment show that the best performance is achieved for:

- optimized shell: prefetch read version implementation.

6.2 Main contributions

The main contributions of this thesis are as follows.

- **Unified message format for PLB, OPB and FSL protocol**: We have designed protocol conversions to map the address based bus protocols of the Coreconnect bus architectures and the streaming protocol of the Xilinx FSL link into the streaming protocol of the Æthereal NoC.

- **Prototyping on FPGA**: We have ensured that our design is correct by implementing it on a Virtex-II Pro FPGA.

- **Quantitive evaluation of applications**: We have conducted experiments, using the Æthereal NoC as interconnect and various shell configurations, to obtain information on the performance of the application. Furthermore, we researched the effects of link bandwidth on the transaction latency.

- **Advanced techniques to improve performance**: We have researched the potential advantages and impact of several advanced techniques. Furthermore, we developed a hardware technique to coalesce transactions into burst transactions when possible.

- **Analyses of error reporting**: We have analyzed the difficulties associated with reporting error using the Æthereal NoC message format. Furthermore, we proposed several methods to address this issue in section 3.3.2.4.

6.3 Related work

For this thesis, we have used concepts related to the field of computer architecture as well as parallel computing. This section gives an overview of related work in those fields. We also discuss a couple of NI shells related to our work.

6.3.1 Prefetching

Prefetching techniques are used to tolerate memory latency. Numerous hardware and software schemes have been proposed to cope with the latency associated with memory access. We focus on software prefetching techniques. Furthermore, we distinguish:

- whether the prefetches are inserted manually or automatically into the code

- the type of hardware that is used to support the prefetch

Most of the software prefetching techniques proposed, focuses on compiler-controlled prefetching, as detailed in [20]. In [14], the prefetches are used to overlap the execution of instructions with activity in the memory hierarchy, in order reduce the cache miss penalty or miss rate. The compiler is used to insert the prefetch instructions. The authors distinguish between loading the data into a register or only into a cache. The hardware support consists of nonblocking or lockup-free caches. In [21], the author proposes and implements a compiler-controlled algorithm for inserting prefetches into code. The hardware support consists of a lockup-free cache. The differences between these and our approach is that we insert prefetches manually instead of automatically, and we use a simple DMA transfer instead of a lockup-free cache hardware to support the prefetching.

In [6], the authors used DMA mode for memory transfers, in combination with prefetch mechanisms. Furthermore, they integrated the prefetching in a framework that optimizes memory hierarchy, allocation and assignment. They have also presented an exhaustive list of prefetching mechanism. To support the DMA, they used an off-the-shelf DMA controller. The difference with our approach is that we analyze the code and insert the prefetches manually, instead of using a formalized technique. Also, we do not require the system to use the DMA I/O method and we use a simplified DMA transfer.

6.3.2 NI shells

In this section, we focus only on shells designed for the Æthereal NoC. We highlight the main differences between the shells. In [8] and [12], DTL, OCP and AXI shells are described for the Æthereal NoC. The OCP and AXI protocol have independent read and write channels. This makes it is possible to support two connections between an initiator and a target in the shell. We only provide support for one connection in our shells. For DTL, the bus forwards the request or response signals of the IP, based on the address, to the shells. It contains a reconfigurable address decoder. In our case, the shells are responsible to recognize the address on the bus and determine whether it is in its address range. Furthermore, we implemented in our PLB shell a mechanism to coalesce multiple single transactions into burst transactions when possible. The DTL, OCP and AXI shells do not have this feature.

6.4 Future work

We have the following recommendations for future work.

1. **Error reporting**: We have demonstrated in the prototype OPB shells a technique for handling transactions errors. However, our PLB and FSL shells do not support that technique as, in our opinion, the options that the Æthereal message format provides are not practical. Other techniques are needed for handling transaction errors. We have proposed alternative techniques, but they require changing the Æthereal message format.

2. **Hardware stride transfer detection**: We have developed a technique to coalesce multiple write transactions to consecutive addresses into burst transaction. An extension of this work is to also detect stride transfers and to coalesce them into burst transaction when possible.

3. **Prototyping using multiple μBlazes**: We have performed experiments with only one μBlaze, as the applications we have used are written for single processor implementation. The next step is to have a system consisting of multiple processors, running different applications concurrently. Issues like synchronization between multiple processors can be further researched.

4. **Direct interoperability between IPs and NIK**: We have designed hardware components (shells) to provide connectivity with IPs. Another possible solution is to have the IPs interoperate with the NoC directly, without the need of a specialized hardware components. This is suitable for IPs that support streaming protocols. We have proposed a technique to offer such interoperability between the NoC and a IP in section 3.5.2.

5. **Interoperability with other protocols**: In this study, we have provided connectivity with the Coreconnect bus cores. An extension to our work is to offer interoperability with other protocols, like the Wishbone bus architecture.

6. **Automated prefetches**: In our work, we have inserted the prefetches manually. This is a time consuming process, as the code needs to be analyzed, modified and verified. An automated technique potentially addresses these issues, resulting in a more structural method for performing prefetches.

7. **Run time address configuration**: The address of our shells can be configured only during design time. A desirable functionality is to configure the shell address also during run time. We proposed a method to configure the shell address at runtime in Section 3.4.2.

Bibliography

[1] Adrijean Adriahantenaina, Herve Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino, *Spin: A scalable, packet switched, on-chip micro-network*, DATE '03: Proceedings of the conference on Design, Automation and Test in Europe (Washington, DC, USA), IEEE Computer Society, 2003, p. 20070.

[2] ARM, *Amba system architecture*, `http://www.arm.com/products/solutions/AMBAHomePage.html`.

[3] Tobias Bjerregaard and Shankar Mahadevan, *A survey of research and practices of network-on-chip*, ACM Comput. Surv. **38** (2006), no. 1, 1.

[4] Luca Carloni and Alberto Sangiovanni-Vincentelli, *Coping with latency in soc design*, IEEE Micro, Special Issue on Systems on Chip **22** (2002), no. 5, 12.

[5] William J. Dally and Brian Towles, *Route packets, not wires: on-chip inteconnection networks*, DAC '01: Proceedings of the 38th conference on Design automation (New York, NY, USA), ACM, 2001, pp. 684–689.

[6] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis, *A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck*, IEEE Trans. Very Large Scale Integr. Syst. **14** (2006), no. 3, 279–291.

[7] John Dielissen, Andrei Radulescu, Kees Goossens, and Edwin Rijpkema, *Concepts and implementation of the philips network-on-chip*, In IP-Based SoC Design, 2003.

[8] Kees Goossens, John Dielissen, and Andrei Radulescu, *æthereal network on chip: Concepts, architectures, and implementations*, IEEE Des. Test **22** (2005), no. 5, 414–421.

[9] Joint Photographic Experts Group, *Jpeg standard*, `http://www.jpeg.org/jpeg/index.html`.

[10] Pierre Guerrier and Alain Greiner, *A generic architecture for on-chip packet-switched interconnections*, DATE '00: Proceedings of the conference on Design, automation and test in Europe (New York, NY, USA), ACM, 2000, pp. 250–256.

[11] MicroBlaze Processor Reference Guide, *Embedded development kit edk 10.1i.*

[12] Andreas Hansson, *A composable and predictable on-chip interconnect*, Ph.D. thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 2009.

[13] Andreas Hansson and Kees Goossens, *Trade-offs in the configuration of a network on chip for multiple use-cases*, In The 1st ACM/IEEE International Symposium on Networks-on-Chip, 2007.

[14] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[15] Ron Ho, Kenneth W. Mai, Student Member, and Mark A. Horowitz, *The future of wires*, Proceedings of the IEEE, 2001, pp. 490–504.

[16] Mark Horowitz, *Vlsi scaling for architects.*

[17] IBM, *128-bit processor local bus architecture specifications version 4.7.*

[18] ———, *Coreconnect bus architecture*, `http://www-03.ibm.com/technology/power/licensing/coreconnect/index.html`.

[19] ———, *On-chip peripheral bus architecture specications version 2.0.*

[20] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to parallel computing: design and analysis of algorithms*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[21] Todd C. Mowry, *Tolerating latency through software-controlled data prefetching*, Tech. report, Stanford, CA, USA, 1994.

[22] Opencores, *Wishbone system-on-chip (soc) interconnection architecture for portable ip cores.*

[23] Tim Peters, *Livermore loops*, `http://www.netlib.org/benchmark/livermorec`.

[24] Philips, *Coreuse 3.2 device transaction level (dtl) protocol specication.*

[25] Andrei Radulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage, *An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration*, IEEE Transactions on CAD of Integrated Circuits and Systems **24** (2004), 4–17.

[26] Sam Siewert, *Soc drawer: The resource view*, `http://www.ibm.com/developerworks/power/library/pa-soc1`.

[27] Xilinx University Program Virtex-II Pro Development System, *Hardware reference manual.*

[28] Frank Vahid and Tony D. Givargis, *Embedded system design: A unified hardware/-software introduction*, international student edition ed., Wiley, October 2001.

[29] Xilinx, *Fast simplex link (fsl) bus (v2.11a).*

[30] ———, *Fsl link*, `http://www.xilinx.com/products/ipcenter/FSL.htm`.

[31] ———, *Lmb bram interface controller (v2.10b) specification.*

[32] ———, *Local memory bus (lmb) v1.0 (v1.00a).*

Application C code A

A.1 kernel.c

```
 #include <stdio.h>
#include "kernel.h"

/*
C***********************************************************************
C***  KERNEL 1      HYDRO FRAGMENT
C***********************************************************************
C
*/
void K_1()
{
#define ARRAY_SZ 1013
#define BURST_SZ 20
unsigned int Q, R, T,burst;
unsigned int temp_Y,temp_Z1,temp_Z2 ;
unsigned int time_start,time_finish;

#ifdef fpga
unsigned int *X,*Y,*Z;
unsigned int Y_burst[32],Z_burst[32],*Y_pnt,*Z1_pnt,*Z2_pnt;

X=(unsigned int *)MEM_ADDR;
Y=X+ARRAY_SZ;
Z=X+(2*ARRAY_SZ);
#else
unsigned int X[1013], Y[1013], Z[1013];
#endif
//Initialization
unsigned int   i, L, k,j;

for (L = 0; L <= 1012; L++)
{ Y[L] = L*L; Z[L] = L+L%9; };

//make sure that all data is written to mem before using read prefetch
```

```
#ifdef fpga
xil_printf("init finish: 0x%08x\n\r",Y[0] );
#else
printf("init finish: 0x%08x\n\r",Y[0] );
write_file(&Y[0],0 );
#endif
Q = 3; R = 4; T = 5;

//real computation
#ifdef fpga
time_start=timer(1);
#endif
for (L = 1; L <= 7; L++)
{
for (k = 0; k <= 1001; k++)
{
#ifdef fsl

#ifdef BURST_RD
if ( k==0 || k%31==0 )
{
burst=(1002-k>31)?31:(1002%k)-1;

putdfslx(burst,0,FSL_CONTROL);
putdfslx(Y+k,0,FSL_DEFAULT);
putdfslx(Z+10+k,0,FSL_DEFAULT);
for (j=0;j<burst+1;j++)
{
getdfslx(Y_burst[j],0,FSL_DEFAULT);

}
for (j=0;j<burst+1;j++)
{
getdfslx(Z_burst[j],0,FSL_DEFAULT);
}
Y_pnt=Y_burst;
Z1_pnt=Z_burst;
Z2_pnt=Z1_pnt+1;
}
X[k] = Q + (*Y_pnt) * (R* (*Z1_pnt) + T*(*Z2_pnt));
Y_pnt++;
Z1_pnt++;
Z2_pnt++;
#else
if (k==0)
{
for(j=0;j<4;j++)
{
putdfslx(Y+j,0,FSL_DEFAULT);//Y[k]
putdfslx(Z+10+j,0,FSL_DEFAULT);//Z[k+10]
putdfslx(Z+11+j,0,FSL_DEFAULT);//Z[k+11]
}
}
if (k<=997)
{
putdfslx(Y+k+4,0,FSL_DEFAULT);//Y[k]
putdfslx(Z+14+k,0,FSL_DEFAULT);//Z[k+10]
putdfslx(Z+15+k,0,FSL_DEFAULT);//Z[k+11]
}
getdfslx(temp_Y,0,FSL_DEFAULT);
getdfslx(temp_Z1,0,FSL_DEFAULT);
getdfslx(temp_Z2,0,FSL_DEFAULT);

X[k] = Q + temp_Y*(R*temp_Z1 + T*temp_Z2);
#endif
#else
X[k] = Q + Y[k]*(R*Z[k+10] + T*Z[k+11]);
#endif
};
};
#ifdef fpga
time_finish=timer(0);
xil_printf("time=%d\n\r",time_finish);
#endif
//verify
for (i = 0; i <= 1011; i++)
{
#ifdef fpga
if (i%100==0)
{
xil_printf("    0x%08x\n\r",  X[i]);
}
#else
#ifdef FILE_S
 write_file(&X[i],0);//if (i%100==0)
#else
if (i%100==0) printf("    0x%08x\n\r",  X[i]);
write_file(&X[0],0 );
#endif
#endif
```

```
};

#ifdef fpga
xil_printf("start time=%d\r\n", time_start);
xil_printf("finish time=%d\r\n", time_finish);
xil_printf("total time=%d\r\n", time_finish-time_start);
#endif
#undef ARRAY_SZ
#undef BURST_SZ

//set burst to ZERO again !!!!
putdfslx(0,0,FSL_CONTROL);

}



/*
C
C***************************************************************************
C***  KERNEL 12     FIRST DIFF.
C***************************************************************************
*/
void K_12()
{
#define ARRAY_SZ 1002
#define BURST_SZ 20

#ifdef fpga
unsigned int *X,*Y;
unsigned int Y_burst[32],*Y1_pnt,*Y0_pnt;
unsigned int temp_Y,temp_Y1,offset;
unsigned int time_start,time_finish,burst;
X=(unsigned int *)MEM_ADDR;
Y=X+ARRAY_SZ;
offset=0;
#else
unsigned int X[1002], Y[1002];
#endif

int   L, k,j;
//preps
for (L = 0; L <= 1001; L++)
{
Y[L] = L*L*L;
```

```
};
//make sure that all data is written to mem before using read prefetch
#ifdef fpga
xil_printf("init finish: 0x%08x\n\r",Y[1] );
#else
printf("init finish: 0x%08x\n\r",Y[1] );
write_file(&Y[0],0 );
#endif


//real computation
#ifdef fpga
time_start=timer(1);
#endif
for (L = 1; L <= 12; L++)  /* 1000 */
{
for (k = 1; k <= 1000; k++)
{
#ifdef fsl
#ifdef BURST_RD
if ( k==1 || k%31==0 )
{
burst=(1001-k>31)?31:(1001%k);
if (k!=0) offset=k; else offset=0 ;
putdfslx(burst,0,FSL_CONTROL);
putdfslx(Y+offset,0,FSL_DEFAULT);
for (j=0;j<burst+1;j++)
{
getdfslx(Y_burst[j],0,FSL_DEFAULT);

}
Y0_pnt=Y_burst;
Y1_pnt=Y0_pnt+1;
}

X[k] =  *Y1_pnt - *Y0_pnt;
Y0_pnt++;
Y1_pnt++;
#else
if (k==1)
{
for(j=1;j<5;j++)
{
putdfslx(Y+j,0,FSL_DEFAULT); //Y[k]
putdfslx(Y+1+j,0,FSL_DEFAULT);//Y[k+1]
}
}
if (k<=996)
{
putdfslx(Y+k+4,0,FSL_DEFAULT); //Y[k]
putdfslx(Y+k+5,0,FSL_DEFAULT);//Y[k+1]
}
getdfslx(temp_Y,0,FSL_DEFAULT);
getdfslx(temp_Y1,0,FSL_DEFAULT);
X[k] =  temp_Y1 - temp_Y;
#endif
#else
X[k] =  Y[k+1] - Y[k];
#endif

};
};
#ifdef fpga
time_finish=timer(0);
xil_printf("time=%d\n\r",time_finish);
#endif

//Fb
for (L = 1; L <= 1000; L++)
{
#ifdef fpga
if (L%100==0)
{
xil_printf("    0x%08x\n\r",  X[L]);
}
#else
#ifdef FILE_S
 write_file(&X[L],0);//if (L%100==0)
#else
if (L%100==0) printf("   0x%08x\n",  X[L]);
#endif

#endif
};

#ifdef fpga
xil_printf("start time=%d\r\n", time_start);
xil_printf("finish time=%d\r\n", time_finish);
xil_printf("total time=%d\r\n", time_finish-time_start);
#endif
```

```
#undef ARRAY_SZ
#undef BURST_SZ

//set burst to ZERO again !!!!
putdfslx(0,0,FSL_CONTROL);
}




/*
*******************************************************************
*   Kernel 6 -- general linear recurrence equations
*******************************************************************
*/
void K_6()
{
#define n 30
unsigned int   L, k, i,residue;
unsigned int   temp,j,pf;
unsigned int time_start,time_finish;
#ifdef fpga
typedef unsigned int vec[n+1];
unsigned int *W;  //,*B[n];
vec *B;
unsigned int temp_W,temp_B,temp_W1;
W=(unsigned int *)MEM_ADDR;
//B is an array of pointers
B=(vec *)W+1;
#else
unsigned int B[n+1][n+1];
unsigned int W[101];
#endif

//init
for (L = 0; L <= n; L++)
{
for (i = 0; i <= n; i++)
{
B[L][i] = 1;
W[i]    = 1;

};
};
//make sure that all data is written to mem before using read prefetch
#ifdef fpga
```

```
xil_printf("init finish: 0x%08x\n\r",B[1][1] );
#else
printf("init finish: 0x%08x\n\r",B[1][1] );
write_file(&B[1][1],0 );
#endif


//real computation
#ifdef fpga
time_start=timer(1);
#endif
for ( i=1 ; i<=n ; i++ )
{
for ( k=0 ; k<i ; k++ )
{
#ifdef fsl
//#ifdef BURST_RD
//not possible
//#else
//fetch in each loop (changes constantly)
temp_W=W[i];
temp_W1=W[(i-k)-1];
if (k==0)
{
pf=(i<4)?i:4;
for(j=0;j<pf;j++)
{
putdfslx(&B[j][i],0,FSL_DEFAULT); //B[k][i]
}

}
if (i>4 && k<i-4)
{
putdfslx(&B[k+4][i],0,FSL_DEFAULT); //B[k][i]
}
getdfslx(temp_B,0,FSL_DEFAULT);
W[i] = temp_W+ temp_B * temp_W1;

//#endif
#else
W[i] = W[i]+ B[k][i] * W[(i-k)-1];
#endif
}
}
#ifdef fpga
```

```
time_finish=timer(0);
xil_printf("time=%d\n\r",time_finish);
#endif
//feedback
for (i = 1; i <=n; i++)
#ifdef fpga
if (i%1==0) xil_printf("   0x%08x\n\r",  W[i]);
#else
#ifdef FILE_S
if (i%1==0)  write_file(&W[i],0);
#else
if (i%1==0) printf("   0x%08x\n\r",  W[i]);
#endif
#endif
#ifdef fpga
xil_printf("start time=%d\r\n", time_start);
xil_printf("finish time=%d\r\n", time_finish);
xil_printf("total time=%d\r\n", time_finish-time_start);
#endif
#undef n

//set burst to ZERO again !!!!
putdfslx(0,0,FSL_CONTROL);
}




/*
*******************************************************************
*   Kernel 21 -- matrix*matrix product
*******************************************************************
 */
void K_21()
{
#define ROWS 26
#define COLS 102
unsigned int   L, i, k, j,m,rd;
#ifdef fpga
typedef unsigned int vec[COLS];
unsigned int *mem,temp_PX,temp_VY,temp_CX;
unsigned int time_start,time_finish;
vec *PX,*CX,*VY;
mem=(unsigned int *) MEM_ADDR;
```

```
PX=(vec *)mem;
CX=(vec *)mem+ ROWS;
VY=(vec *)mem+ 2*ROWS;
#else
unsigned int PX[26][102], CX[26][102], VY[26][102];
#endif

//init
for (i = 1; i <= 25; i++)
for (j = 1; j <= 101; j++)
{
PX[i][j] = 0;
VY[i][j] = i*j+j*j*i;
CX[i][j] = i*i*j+j*j*i;
};
//make sure that all data is written to mem before using read prefetch
#ifdef fpga
xil_printf("init finish: 0x%08x\n\r",VY[1][1] );
#else
printf("init finish: 0x%08x\n\r",VY[1][1] );
write_file(&VY[1][1],0);
#endif

//real computation
#ifdef fpga
time_start=timer(1);
#endif
for (L = 1; L <= 1; L++)  /* 1000 */
{
for (i = 1; i <= 25; i++)   /* 101 */
{
for (k = 1; k <= 25; k++)   /* 101 */
{
for (j = 1; j <= 101; j++)   /* 101 */
{
#ifdef fsl

//#ifdef BURST_RD
//#else
if (j==1)
{
putdfslx(&VY[i][k],0,FSL_DEFAULT);//VY[i][k]
for(m=1;m<5;m++)
{
```

```
putdfslx(&CX[k][m],0,FSL_DEFAULT);//CX[k][j]
}
//fetch only once in the loop
getdfslx(temp_VY,0,FSL_DEFAULT);
}
if (j<=97)
{

putdfslx(&CX[k][j+4],0,FSL_DEFAULT);//CX[k][j]
}

getdfslx(temp_CX,0,FSL_DEFAULT);

temp_PX=PX[i][j];
PX[i][j] = temp_PX + temp_VY * temp_CX;
//#endif
#else
PX[i][j] = PX[i][j] + VY[i][k] * CX[k][j];
#endif

};
};
};
};
#ifdef fpga
time_finish=timer(0);
xil_printf("time=%d\n\r",time_finish);
#endif

  for (i = 1; i <= 1; i++)
      for (j = 1; j <= 101; j++)
#ifdef fpga
 xil_printf("   0x%08x\n\r", PX[i][j]);
#else
#ifdef FILE_S
write_file(&PX[i][j],0);
#else
printf("   0x%08x\n\r", PX[i][j]);
#endif

#endif
#ifdef fpga
xil_printf("start time=%d\r\n", time_start);
xil_printf("finish time=%d\r\n", time_finish);
xil_printf("total time=%d\r\n", time_finish-time_start);
```

```
#endif

//set burst to ZERO again !!!!
putdfslx(0,0,FSL_CONTROL);
}

#ifndef fpga
void write_file(unsigned int *VAL,unsigned int text)
{
FILE *fpIN;
int *i;

fpIN=fopen("out_dbg","a");
if(fpIN == 0)
{
perror("out_dbg");
          return ;
}



if (text) fprintf(fpIN,"-------------------------------------------------------------------
else
fprintf(fpIN,"   0x%08x\n\r",*VAL);

fclose(fpIN);
}
#endif
```

A.2 kernel.h

```
   //
// C++ Interface: kernel
//
// Description:
//
//
// Author: Jason de Windt <dwindt65@forseti>, (C) 2009
//
// Copyright: See COPYING file that comes with this distribution
//
//
```

```
#ifdef fpga
#include "xilinx.h"
#endif
void write_file(unsigned int *,unsigned int );
void K_1();
void K_12();
void K_6();
void K_21();
```

A.3 xilinx.h

```
   #include "jpeg.h"
#ifdef fpga

#include "xparameters.h"
#include "xutil.h"
#include "xio.h"
#include "fsl.h"




#define MEM_ADDR 0xB0010000
#define MAX_ADDR 50



//start,stop and read a timer

int timer(int );


#endif
```

A.4 xilinx.c

```
 //
#include "xilinx.h"
#define MEM_TEST
#ifdef fpga
#define TIMER_START 0x00000001
//XTmrCtr TimerCounter;

int timer(int start)
{
int val;

val=XIo_In32(XPAR_TIMER_0_BASEADDR);

//start timer
if (start)
XIo_Out32(XPAR_TIMER_0_BASEADDR ,TIMER_START);

return val;

}
#endif
```

A.5 generator.c

```
 #include <stdio.h>
#ifndef fpga
 #include <stdlib.h>
#endif
#include "generator.h"

#ifdef fpga
int gen()
{
#else
int main()
{
#endif
```

```
//#ifndef fpga
 unsigned int result[LOOP_MAX];
//#endif

unsigned int i,j,offset;
float cmplx;
unsigned int mem_pr;
unsigned int  *mem;
unsigned int loop,burst;
unsigned int wr_st,wr_fn,rd_st,rd_fn,time,com_st,com_fn;
loop=LOOP_MAX;


#ifdef fpga
#ifdef N_BURST_WR
mem=(unsigned int *) HIGH_ADDR-1;
#else
mem=(unsigned int *) BASE_ADDR;
#endif
#else
 mem=(unsigned int *) malloc(100000);
 mem_pr=(unsigned int)mem;
#endif

//fill the mem with data
print("write\n\r");
#ifdef fpga
 wr_st=timer(1);
#endif

for(i=0;i<loop;i++)
{
#ifdef N_BURST_WR
 *mem=i;
 mem--;
#else
 *mem=i;
 mem++;
#endif
}

#ifdef fpga
 wr_fn=timer(0);
 print("wr=%d\nr",wr_fn);
#endif
```

```
print("finish write to mem\n\r");

//set the pointers to the correct location
#ifdef fpga
 mem=(unsigned int *) BASE_ADDR;
#else
 mem=(unsigned int *)mem_pr;
#endif

//decouple write and read
print("decouple wr and rd=%d\n\r",*(mem+1));

//set loop for read
loop=LOOP_MAX;

#ifdef fsl
 //set burst to zero again
 putdfslx(0,0,FSL_CONTROL);
#endif

//print("start read\n\r");
//perform the reads
#ifdef fpga
 rd_st=timer(1);
#endif

//print("before loop\n\r");
for(i=0;i<loop;i++)
{
//print("inside loop\n\r");
#ifdef fsl
//print("inside fsl\n\r");
#ifdef BURST_RD
if ( i==0 || i%32==0 )
{
burst=(loop-i>31)?31:(loop%i)-1;
putdfslx(burst,0,FSL_CONTROL);
putdfslx(mem+i,0,FSL_DEFAULT);
}
//get data from FSL
  getdfslx(result[i],0,FSL_DEFAULT);
//print("i=%d\tres=%d\n\r",i,result[i]);
#else
if (i==0)
```

```
{
//remote memory
putdfslx(mem,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+1,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+2,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+3,0,FSL_DEFAULT);
//mem++;
}
if(i<loop-4)
{
putdfslx(mem+4,0,FSL_DEFAULT);
mem++;
}

//get data from FSL
getdfslx(result[i],0,FSL_DEFAULT);
//print("i=%d\tj=%d\n\r",i,j);
#endif
#else
//print("reading\t");
  result[i]=*mem;
//print(":%d\n\r",j);
mem++;
#endif
}

#ifdef fpga
 rd_fn=timer(0);
 print("rd=%d\n\r",rd_fn);
 time=(rd_fn-rd_st) + (wr_fn-wr_st);
 print("rd=%d\twr=%d\ttot=%d\n\r",rd_fn,wr_fn,time);
#endif


//check for errors
#ifdef N_BURST_WR
for(i=0;i<loop;i++)
{
j=result[loop-1-i];

if (j!=i)
{
```

```
print("ERR=%d\texpected=%d\n\r",j,i);
break;
}
}
#else
for(i=0;i<loop;i++)
{
j=result[i];

if (j!=i)
{
print("ERR=%d\texpected=%d\n\r",j,i);
break;
}
}
#endif



#ifdef fsl
 //set burst to zero again
 putdfslx(0,0,FSL_CONTROL);
#endif


//-----------------------------------------------------------------

#ifdef fpga
mem=(unsigned int *) BASE_ADDR;
#else
 mem=(unsigned int *) malloc(100000);
 mem_pr=(unsigned int)mem;
#endif

//print("start read\n\r");
//perform the reads
#ifdef fpga
 com_st=timer(1);
#endif

//print("before loop\n\r");
for(i=0;i<loop;i++)
{
//print("inside loop\n\r");
#ifdef fsl
```

```
if (i==0)
{
//remote memory
putdfslx(mem,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+1,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+2,0,FSL_DEFAULT);
//mem++;
putdfslx(mem+3,0,FSL_DEFAULT);
//mem++;
}
if(i<loop-4)
{
putdfslx(mem+4,0,FSL_DEFAULT);
}

//get data from FSL
getdfslx(result[i],0,FSL_DEFAULT);
*mem=i;
mem++;
#else
  result[i]=*mem;
*mem=i;
mem++;
#endif
}

#ifdef fpga
 com_fn=timer(0);
 print("com=%d\n\r",com_fn);
 time=(com_fn-com_st);
 print("com=%d\ttot=%d\n\r",com_fn,time);
#endif


//check for errors
#ifdef N_BURST_WR
for(i=0;i<loop;i++)
{
j=result[loop-1-i];

if (j!=i)
{
print("ERR=%d\texpected=%d\n\r",j,i);
break;
}
}
#else
for(i=0;i<loop;i++)
{
j=result[i];

if (j!=i)
{
print("ERR=%d\texpected=%d\n\r",j,i);
break;
}
}
#endif


//------------------------------------------------------------------

#ifndef fpga
 free((unsigned int *)mem_pr);
#endif
print("Finish\n\r");
/**/
} ;
```

A.6 generator.h

```
#define HIGH_ADDR 0xb0018ff0
#define BASE_ADDR 0xb0010000
#define LOOP_MAX (HIGH_ADDR-BASE_ADDR)/4

#ifdef fpga
 #include "xilinx.h"
#endif
```

Experiment results B

B.1 Base shell results

This section contains all the results obtained for the base shell. The numbers are the total number of cycles that was needed to complete the operation. Table B.1 shows the result for the base shell without loop unrolling and Table B.2 shows the result for loop unrolling.

B.2 Optimized shell results

This section contains all the results obtained for the optimized shell. The numbers are the total number of cycles that was needed to complete the operation. Table B.3 shows the result for the optimized shell without loop unrolling and Table B.4 shows the result for loop unrolling.

Table B.1: Results for the base shell without loop unrolling

	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	
BW 100%	870568	847374	846963	823799	jpg
BW 50%	895225	852659	866457	823835	
BW 25%	964426	865314	923184	823928	
BW 12,5%	968485	871581	923283	826541	
	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	
BW 100%	1578229	1319409	611085	336666	k1
BW 50%	2777554	2083194	695772	337153	
BW 25%	5555022	4166329	1390109	337496	
BW 12,5%	5555043	4166328	1390184	463649	
	Base npr npw	**Base npr pw**	**Base pr npw**	**Base pr pw**	
BW 100%	1980201	1531942	901431	420323	k12
BW 50%	3563979	2376070	1189214	421148	
BW 25%	7127999	4752001	2378402	496589	
BW 12,5%	7128014	4752000	2378378	541733	
	Base pr pw	**Base pr npw**	**Base npr pw**	**Base npr npw**	
BW 100%	70792	89201	92173	108377	k6
BW 50%	95109	141090	138130	184163	
BW 25%	190117	282140	276230	368264	
BW 12,5%	190132	282110	276230	368279	
	Base pr pw	**Base pr npw**	**Base npr pw**	**Base npr npw**	
BW 100%	5432668	7842678	11565692	14065227	k21
BW 50%	6311265	12560644	18748148	24997519	
BW 25%	12622539	25121247	37496273	49994982	
BW 12,5%	12622551	25121217	37496270	49994997	

Table B.2: Results for the base shell with loop unrolling

	Base pr pw	**Base npr pw**	**Base pr npw**	**Base npr npw**	
BW 100%	800056	823606	823265	846825	jpg loop unrolling
BW 50%	800056	827583	842804	870285	
BW 25%	800192	841671	899630	941070	
BW 12,5%	802896	846527	899729	943446	
	Base pr pw	**Base pr npw**	**Base npr pw**	**Base npr npw**	
BW 100%	322678	592313	1289580	1562405	K1 loop unrolling
BW 50%	322948	695742	2083161	2777556	
BW 25%	325073	1390172	4166294	5555075	
BW 12,5%	463714	1390154	4166294	5555096	
	Base pr pw	**Base pr npw**	**Base npr pw**	**Base npr npw**	
BW 100%	387303	848925	1485017	1955434	K12 loop unrolling
BW 50%	390841	1189180	2376024	3564017	
BW 25%	476383	2378373	4751965	7127965	
BW 12,5%	529818	2378343	4751965	7127980	
	Base pr pw	**Base npr pw**	**Base pr npw**	**Base npr npw**	
BW 100%	71055	90188	89247	108307	K6 loop unrolling
BW 50%	95070	138200	141055	184129	
BW 25%	190081	276195	282090	368328	
BW 12,5%	190096	276195	282174	368244	
	Base pr pw	**Base pr npw**	**Base npr pw**	**Base npr npw**	
BW 100%	5374478	7827803	11251358	13550624	K21 loop unrolling
BW 50%	6311324	12560609	18748122	24997484	
BW 25%	12622503	25121212	37496238	49994947	
BW 12,5%	12622515	25121287	37496235	49994962	

Table B.3: Results for the optimized shell without loop unrolling

	Opt pr	**Opt npr**	
BW 100%	820919	844483	jpg
BW 50%	820970	849984	
BW 25%	821045	862543	
BW 12,5%	821045	866399	
	Opt pr	**Opt npr**	
BW 100%	301596	1284650	k1
BW 50%	302132	2083223	
BW 25%	316675	4166349	
BW 12,5%	463710	4166347	
	Opt pr	**Opt npr**	
BW 100%	360302	1485003	k12
BW 50%	376566	2375983	
BW 25%	476387	4752015	
BW 12,5%	529831	4752021	
	Opt pr	**Opt npr**	
BW 100%	68255	88442	k6
BW 50%	95073	138154	
BW 25%	190094	276254	
BW 12,5%	190094	276255	
	Opt pr	**Opt npr**	
BW 100%	5135861	11251357	k21
BW 50%	6311333	18748167	
BW 25%	12622516	37496202	
BW 12,5%	12622513	37496294	

Table B.4: Results for the optimized shell with loop unrolling

	Opt pr	**Opt npr**	
BW 100%	797176	820745	jpg loop unrolling
BW 50%	797176	825011	
BW 25%	797240	838841	
BW 12,5%	797257	841741	
	Opt pr	**Opt npr**	
BW 100%	287607	1266260	K1 loop unrolling
BW 50%	288337	2083174	
BW 25%	309803	4166313	
BW 12,5%	463676	4166313	
	Opt pr	**Opt npr**	
BW 100%	327325	1452021	K12 loop unrolling
BW 50%	371912	2376025	
BW 25%	476440	4751987	
BW 12,5%	529894	4751981	
	Opt pr	**Opt npr**	
BW 100%	69367	88445	K6 loop unrolling
BW 50%	95043	138115	
BW 25%	190059	276218	
BW 12,5%	190059	276218	
	Opt pr	**Opt npr**	
BW 100%	5049011	11045098	K21 loop unrolling
BW 50%	6311288	18748144	
BW 25%	12622483	37496254	
BW 12,5%	12622487	37496260	