Karlsruhe Institute of Technology

Institut für Angewandte Informatik
und Formale Beschreibungsverfahren

STUDIENARBEIT

FPGA Crossbar Switch Architecture for Partially Reconfigurable Systems

von

Till Fischer

Eingereicht am 07.05.2010 beim
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
am Karlsruher Institut für Technologie

Referent: Prof. Dr. Hartmut Schmeck
Betreuer: Dr. Oliver Diessel (UNSW)

Heimatanschrift:
Hainstraße 35
65597 Hünfelden

Studienanschrift:
Wielandtstraße 16
76137 Karlsruhe

Studienarbeit am

Institut für Angewandte Informatik und Formale Beschreibungsverfahren

Thema: FPGA Crossbar Switch Architecture for Partially Reconfigurable Systems

Autor: Till Fischer

Till Fischer

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 07.05.2010 ....

Zusammenfassung

In den letzten Jahren haben sich durch immer höhere Integrationsdichten in der Chiptechnologie die verfügbaren Ressourcen auf FPGAs vervielfacht. Damit sind stetig komplexere Implementierungen bis hin zu ganzen System-on-Chips auf einem einzelnen FPGA möglich geworden. Gleichzeitig wurde die Fähigkeit zur Rekonfiguration immer flexibler, womit sich inzwischen genau definierte Bereiche auf dem Chip zur Laufzeit modifizieren lassen, ohne dass eine Unterbrechung der übrigen Hardware nötig ist (modulare dynamische partielle Rekonfiguration). Vorteile dieser Technik sind unter anderem Platz- und Energieeinsparungen durch Auslagern nicht benötigter Komponenten oder die vereinfachte Wartung und Aktualisierung von Hardware. Allerdings sind besondere Maßnahmen erforderlich, um den Betrieb eines derartigen partiell rekonfigurierbaren Systems mit geringem Overhead zu gewährleisten. Dazu zählen insbesondere spezielle Kommunikationsstrukturen, von welchen nun verlangt wird, dass sie sich den weniger statischen Anforderungen anpassen können. Eine derartige Struktur wird in dieser Arbeit vorgestellt.

Aktuelle Plattformen für partielle Rekonfiguration benutzen vorwiegend Busse oder feste Verbindungen, welche durch ihre Inflexibilität der partiellen Rekonfiguration nur bedingt gerecht werden. Daher schlagen neuere Forschungen Networks-on-Chip als sehr flexible Alternative vor. Das größte Problem von diesem Ansatz ist jedoch der hohe Overhead, welcher mit einer derartigen Netzwerkarchitektur einher geht. In dieser Arbeit wird daher eine angepasste Crossbar Switch Architektur vorgestellt, welche eine erhöhte Flexibilität gegenüber einfachen Bussen mit einem geringem Kontrolloverhead verbindet. Informationen werden hier auf dedizierten Kanälen übertragen, welche nach initialer Einrichtung nur ein Minimum an zusätzlichem Aufwand verursachen. Neue Konfigurationen für die Kanäle können in kürzester Zeit eingerichtet werden, es sind also trotzdem häufige Anpassungen möglich. Außerdem wird eine nebenläufige Übertragung auf allen Kanälen angeboten, es kommt also zu keiner gegenseitigen Blockierung der Kommunikationspartner. Da schließlich sämtliche Kommunikation gepuffert wird, ist ein asynchroner Betrieb der Komponenten möglich. Diese Eigenschaften müssen durch den Verbrauch von zusätzlichen Ressourcen erkauft werden. Allerdings berücksichtigt die Implementierung das spezielle Layout von FPGAs und es wird außerdem die allgemeine Crossbar Switch Architektur angepasst, um eine schlechte Skalierung zu vermeiden.

In dieser Arbeit werden zuerst einige alternative Lösungen zur on-chip Kommunikation in partiell rekonfigurierbaren Systemen vorgestellt, bevor der Entwurf und die Implementierung der neuen Architektur genauer beschrieben wird. Abschließend erfolgt eine Auswertung anhand des implementierten Prototyps sowie ein Vergleich mit den alternativen Lösungen. Es zeigt sich, dass der Flächenoverhead gut kontrolliert werden kann und für kleine Systeme eine höhere Performance erreicht wird. Für größere Systeme offenbart sich jedoch ein an Stärke zunehmender Leistungseinbruch. Hier wird neben anderen Vorschlägen zur Verbesserung auch eine einfache Erweiterung vorgestellt, welche dieses Problem behebt.

Abstract

Partial reconfiguration offers several benefits for a system on chip. But in order to take advantage of this technique, easy access and low overhead must be provided. One key issue therefore is the on-chip communication architecture. Current reconfigurable systems use buses and fixed links, which are quite inflexible; recent proposals suggest using networks on chip instead. The problem of the latter approach is the high overhead which is required to control the dataflow. In this work, XBar is presented: a customized crossbar switch architecture for reconfigurable systems on chip, which provides flexibility without high control overhead. Channels are established in order to transfer data, and once the channel is set up, data can be delivered with a minimum of additional effort. Concurrent transmission is possible on all channels, so interruptions due to blocking cannot occur. Furthermore, new channel configurations can be established in a very short period of time, so frequent changes are possible. Since communication is buffered, the infrastructure may be used asynchronously and components of different clock domains can be connected. The scalability is worse than with a packet-switched network on chip and it is not possible to provide the same amount of flexibility. However, better overall results can be achieved particularly for throughput and latency for small systems. Furthermore, the design of partially reconfigurable modules is eased.

Acknowledgements

First and foremost I would like to thank my supervisor, Dr Oliver Diessel, for his excellent support during my work. His guidance and the helpful notes he provided were a great assistance. Also I'd like to thank you for the offer to attend the FPT'09 in Sydney, which was a unique opportunity to experience the conference at close range.

Furthermore, I should like to thank Prof Hartmut Schmeck from the University of Karlsruhe, who brought me in contact with Dr Diessel. Without your help my internship at the University of New South Wales would not have been possible. My thanks also go to the University of New South Wales for the Practicum Exchange Program, which enabled me to study in Australia.

Last but not least I would like to thank my family, who supported me in any possible way, and of course I am deeply grateful for the enduring encouragements of my girlfriend Ursula during my far too long absence.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

In the course of time, available resources on FPGA devices have increased thanks to continuous advances in integration density. As a result, more and more complex designs can be implemented on a single FPGA, up to whole System-on-Chips (SoC). Simultaneously, reconfiguration capabilities of FPGAs were enhanced and became more flexible; partial reconfiguration evolved. This enabled the possibility to update, add or remove parts of a system while the rest continues to operate. Exploiting this feature has been an area of particular research interest in recent years and this still holds true today.

Obviously, there are several benefits of partial reconfiguration [**1**]. First of all, it is possible to change parts of the hardware, which might be required due to maintenance for example, without demanding a shutdown of the system. Such updates are possible at any time, both locally and remotely. As a consequence, updates can even be applied in critical situations, when access is not possible under conventional circumstances. Furthermore, hardware can be shared. Only those components, which are necessary at a certain point of time, must be available. This allows realization of even larger systems on smaller FPGA devices. This can lead to reductions in power consumption and overall costs. System performance may increase, since more specialized components can be used. Additionally, reconfiguration times can be reduced by difference-based partial reconfiguration [**2**]; bitstreams can remain smaller by containing configuration data for parts that effectively change only. Finally, it is expected that partial reconfiguration could significantly reduce iteration times for hardware development. If only small parts of the system had to be re-synthesized, this process could be completed within minutes rather than hours. This would allow use of fast software development methodologies for hardware design, too. However, this is not yet applicable at this point of time, since iteration cycles are still very long.

While the advantages of module-based partial reconfiguration are evident and benefits were shown for several applications, less attention was paid to the communication infrastructure, which is necessary to enable cooperation of separate components in a reconfigurable system. However, special requirements apply to such an infrastructure, since communication partners may change during runtime. Current reconfigurable platforms use buses and fixed links as interconnects which provides neither good performance nor flexibility.

Because of this, current research focuses on Networks-on-Chip (NoC) to satisfy the specific demands of a partially reconfigurable system. Most of these approaches utilize similar techniques to those used in common computer networks (local area networks, internet etc.). Data is split into several packets which are transferred independently between routers until the destination is reached. Because of this, all links in the network can be shared for all connections and a very high flexibility may be achieved. Furthermore, it is possible to serve rapidly changing communication needs: it makes no difference whether two succeeding packets are sent to the same target or if changing destinations are addressed.

However, very high control overhead is introduced at the same time, and in fact the benefits might not be exploited. Without question, highly flexible NoCs are superior to other solutions if completely unpredictable situations occur. But contrary to the computer networks addressed formerly, where nothing is known about the requirements of each participant, the communication needs of a specific system component can be predicted very well in most cases. Because of this, a different approach is considered in this work: a crossbar switch is evaluated as on-chip interconnection between partially reconfigurable modules. Crossbar switches can be treated as networks on chip as well, but instead of providing control headers with every message, connections are set up in advance. On the one hand, this significantly reduces the overhead when data is transferred as long as communication partners are fixed, and on the other hand, throughput will not suffer from concurrent communication, since data can be transferred in parallel. However, a crossbar switch is not a lightweight structure and it has to be examined whether an efficient implementation is possible and if the area overhead can be justified.

1.2 Contributions

This work focuses on how a crossbar switch might be designed to allow efficient implementation on Xilinx FPGA devices. The special requirements of a module-based partially reconfigurable system are taken into account, as well as resources and capabilities of partially reconfigurable FPGAs.

As part of this work a generator application is provided, which is capable of creating customized VHDL entities of the proposed communication infrastructure. A custom design flow is introduced, which explains how the crossbar switch can be instantiated in the static design of a partially reconfigurable system.

The proposed design is implemented on a certain device and evaluated for different parameters. Furthermore, a test environment for a specific parameterization is introduced which can be used to visualize the behavior for different configurations. Benefits and drawbacks are compared to other approaches and reasonable application scenarios are suggested. This work also identifies the weak points of the proposed architecture and recommends improvements to certain problems.

1.3 Outline

The background to this work is summarized in Chapter 2. Some basics about FPGA architectures are explained and the different development tools are introduced. Furthermore, an overview of the different communication models is given.

Chapter 3 reports on related work in the field of on-chip interconnects for reconfigurable systems. Three different approaches are presented for this purpose. Furthermore, a very efficient technique for implementing a crossbar switch on certain Xilinx FPGAs is shown.

In Chapter 4 the new crossbar switch is introduced. Based on several requirements, which are enumerated in the beginning of that chapter, the specific design decisions are explained. Finally, the resulting design is presented in detail.

Implementation specific details are addressed in Chapter 5. This also contains the transfer behavior for a precise example. A test environment is described in this chapter as well.

Chapter 6 provides the results of the evaluation for several different parameters and the proposed architecture is compared to the alternative approaches presented in Chapter 3. Scenarios that may benefit from this approach are identified. Several possible improvements are pointed out, which are discussed briefly in Chapter 7.

2 Fundamentals

This chapter explains the fundamentals which are referred to in more detail later on in this work. First, the Field Programmable Gate Array (FPGA), the basic hardware resource used for implementation, is introduced. Afterwards, the ability to partially reconfigure these devices is presented. Subsequently, the tools needed for partial reconfiguration and some more specific tools used for the implementation are described. These tools are provided by the chosen FPGA manufacturer, Xilinx Inc. Finally, an introduction to the different interconnection systems considered for the design or used in related works is given.

2.1 FPGAs

Historically, the main field of use for FPGAs is emulation, evaluation and rapid prototyping of application specific integrated circuits. More recently FPGAs have become the implementation device of choice for small volumes and have proven themselves to be capable of high performance and lowest power as well. They unite the advantages of Programmable Logic Devices (PLDs) and Gate Arrays. The former are logic components, which can be configured to realize a specific but rather simple function. The latter comprise a large number of fixed and non-programmable logic resources, but the routing between these can be defined by the user. The market leader for FPGA technology is Xilinx Inc., and some of the abbreviations and details in the context of FPGA architectures within this work may be specific to Xilinx devices. Actual FPGAs provide both a large number of simple programmable logic resources in Configurable Logic Blocks (CLBs) and a lot of routing resources to individually connect the components. A hardware description can be mapped into the resources of the FPGA to implement the described behavior.

The main component of the CLB is a so-called look-up table (LUT) which is used to store a logic function. For each possible combination of inputs one output value can be configured. So any logic function with the corresponding number of inputs can be implemented within a LUT. Additionally, other resources like multiplexers and flip-flops exist in the CLBs, which are used to combine functions and select combinational or sequential paths.

For interconnection a huge amount of wires must exist within the fabric. To individually connect distinct components, it must be possible to create custom connections. In the FPGA architecture of Xilinx, several *switch matrixes* and *switch boxes* are used for this purpose.

Which wires have to be connected by a switch, and which not, is stored together with the remaining configuration data of the FPGA.

Typically, configuration data is transferred to the FPGA using a bitstream and then stored in SRAM. This means that after cutting the power supply to the device, the data is lost. Because of this, many systems offer nonvolatile memory for configuration data and automatically configure the FPGA on boot up. Furthermore, additional resources that can be accessed from the FPGA are integrated in the most cases. For example, on the development board used in this work (ADM-XRC-4, see Figure 2.1), 6 SSRAM (Synchronous Static Random Access Memory) blocks are arrange around a Virtex-4 VLX160 FPGA.

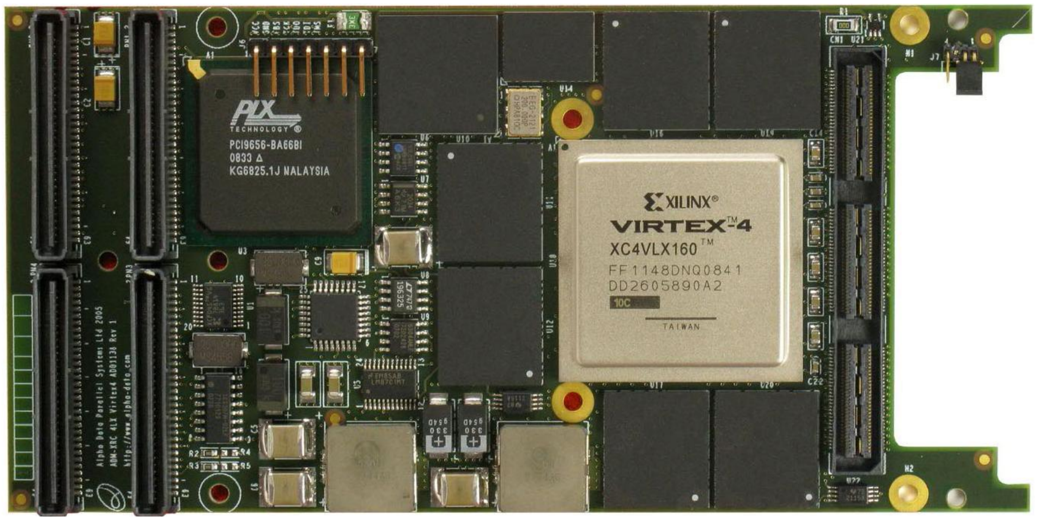


Figure 2.1: ADM-XRC-4 Board [3]

In addition to external resources, modern devices may also contain several dedicated components within the chip fabric. For example, Xilinx devices offer internal memory blocks (Block RAM), multipliers (DSP Slices), whole processor cores (PowerPC) and several other dedicated resources for quite some time.

Additionally, the capabilities of dynamic reconfiguration are further exploited. An actual research topic is so-called *partial reconfiguration* as it is described in the following section.

2.2 Partial Reconfiguration

Partial reconfiguration describes the ability to change the configuration of a certain area of the chip, while the rest of the implemented design continues operation. Partial reconfiguration became available to the user with the XC4000 device family. Theoretically, it was always possible to partially reconfigure FPGAs, if configuration data is stored in SRAM. However, if configuration bitstreams for the device have to be transferred as a whole, operation has to be stalled during the update process. This changed with the Virtex-II device family, where distinct *frames* could be addressed for reconfiguration. This was further enhanced with Virtex-4 devices and much more flexibility is provided nowadays: only 1-dimensional partial reconfiguration was possible with Virtex-II devices; more recent devices support almost any rectangular shape as a *partially reconfigurable region* (PRR). The smallest reconfiguration *frame* is now constrained only by the height of a clock region of the FPGA device (Virtex-4: 16 CLBs [**4**], Virtex-5: 20 CLBs [**5**], Virtex-6: 40 CLBs [**6**]) and can be as narrow as 1 CLB. The modules, which can be loaded into a PRR, are referred to as *partially reconfigurable modules* (PRMs).

An important issue when working with PRMs is to ensure correct communication between the reconfigurable regions and the remaining design. Xilinx handles the problem by enforcing every signal from/to a PRR to be routed through so-called *bus macros*. These are pre-routed components placed to fixed locations on the FPGA during design time. Basic bus macros are provided for all devices capable of partial reconfiguration, but creating a user-defined bus macro is possible as well [**7**].

The benefit of partial reconfiguration is obvious and several advantages were already listed in the introduction. For example, it is possible to add or remove parts of a system as necessary; similar to swapping out pages from main memory to the hard drive. As a result, more functionality can be implemented while utilizing the same area, and overall power consumption can be reduced – as long as removing a component does not disrupt correct operation of the system.

2.3 Xilinx Tools

Since in this work a Xilinx FPGA was used, the provided tools for implementation are briefly described in this section. The main development environment is the Integrated Software Environment (ISE) Design Suite described in 2.3.1. Because the communication infrastructure has to be embedded into a partially reconfigurable design, the Early Access Partial Reconfiguration (EA PR) overlay is needed.[1] In order to floorplan the PRRs and to create the distinct bitstreams for each module and the static design, PlanAhead has to be used. Anoth-

[1] During this work, Xilinx announced ISE 12.1, which will support partial reconfiguration designs out of the box and EA PR will become deprecated.

er utilized part of the ISE Design Suite is COREGenerator, which provides specialized IP-cores for Xilinx FPGAs.

In order to implement the low-level part of the design, the Xilinx Design Language (XDL) was used, which is described in section 2.3.2.

2.3.1 ISE Design Suite

The ISE Design Suite is a suite of tools supporting the complete design flow for integrated systems from a high-level description via synthesis and implementation down to final bit-stream generation for the FPGA device. Most of the tools can be accessed out of the Project Navigator which is the main interface to the user. Nevertheless, there are some additional tools like COREGenerator (2.3.1.1), PlanAhead (2.3.1.2) and FPGA Editor (2.3.1.3) which provide their own, complex interfaces.

However, the ISE Design Suite was intended for conventional (static) designs not involving partial reconfiguration, so some changes have to be made. Xilinx introduced the EA PR overlay for this purpose. The most important modification applied by this patch is to check whether for all signals, which pass the border of a PRR, a bus macro is used.

2.3.1.1 COREGenerator

COREGenerator is a design tool that delivers parameterizable IP-cores optimized for Xilinx FPGAs. The documentation for COREGenerator is split into several independent guides for the separate cores. For this work the FIFOGenerator [8] was used to create FIFO buffers as described in section 4.3.2. COREGenerator is an easy way to include IPs into an ISE design and provides a lot of licensed as well as free IP-cores.

2.3.1.2 PlanAhead

PlanAhead is a powerful layout and floorplanning tool for FPGA designs. The steps between synthesis and place-and-route are streamlined in order to provide more control over how the design is implemented. In particular, the complicated handling of partially reconfigurable designs is eased via a hierarchical design methodology.

2.3.1.3 FPGA Editor

The FPGA Editor provides a detailed graphical view of all FPGA components and the routing resources. It can be used to view and explore a placed and routed design, and facilitates low-level modifications on the logic configuration and interconnections.

2.3.2 XDL

XDL was intended to be a human readable low-level design language providing full control over all FPGA primitives and connections [9]. Although XDL was never developed further and all documentation vanished with the release of version 7 of the ISE suite, the original functionality and the conversion tool are still available in current releases. Basically, XDL provides the same level of access to FPGA resources as the FPGA Editor does, but unlike using the FPGA-Editor it is feasible to create large designs efficiently. XDL source code should not be written directly; instead any scripting language can be used to create the XDL code for the design.

The result can be converted to a Xilinx NCD file, which can contain all placement and routing information. If designed with care, direct conversion into a bitstream and configuration of the FPGA is possible. A more useful approach is to convert the NCD into a *hard macro* (NMC file). Hard macros can be instantiated as black box modules within ISE using any hardware description language like Verilog or VHDL. See section 4.4 for more information about this design flow.

2.4 Interconnects

Several possible interconnection methodologies exist to provide communication between different modules. Basically, they can be distinguished between four groups: point-to-point connections, shared resources, networks and hybrids of the former as sketched in Figure 2.2.

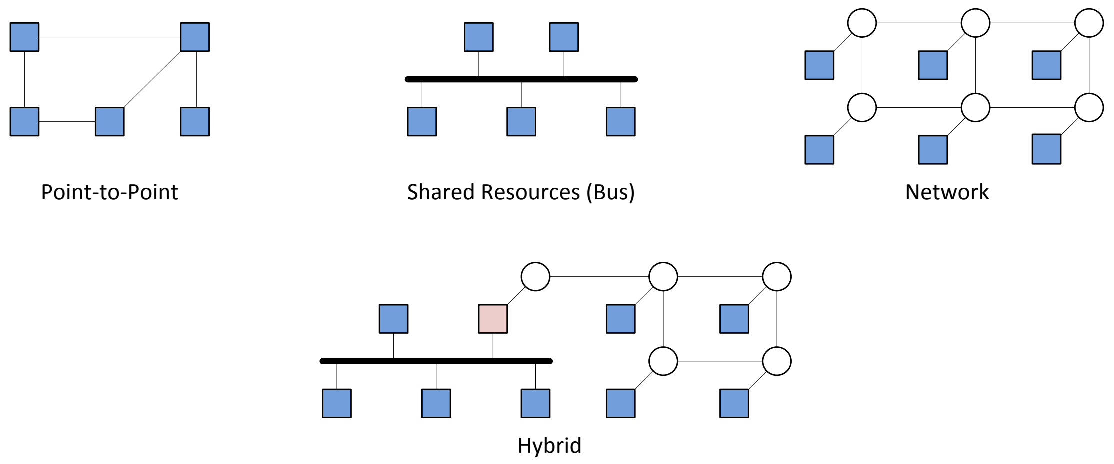


Figure 2.2: Interconnection methodologies

2.4.1 Point-to-Point Connections

The simplest interconnection methodology is to provide separate wires for every connection that has to be made. Since nothing has to be shared this approach promises to achieve the best performance. On the other hand a huge amount of resources is needed and this approach scales poorly for larger numbers of modules.

2.4.2 Shared Resource

A typical shared resource is a bus. All modules are connected to this shared element, but access is provided exclusively to one element only and arbitration is necessary. Furthermore, all communication is synchronized by a certain clock, so all modules are either restricted to use the same clock, or retiming has to be performed by the modules. Additionally, shared resources are characterized by poor scaling as well, since the infrastructure is saturated if a certain number of components is reached: throughput may be high for a small number of elements, but performance will significantly decrease when the components block each other from communication. Furthermore, assuring fairness is another issue that cannot be solved easily. Complex mechanisms like priority levels must be introduced; otherwise a less important module could prevent on-time delivery of critical messages just by continuously requesting the communication resource.

2.4.3 Network

In a network, the distinct modules are no longer connected directly by one or more wires, but a series of switching elements in between is introduced. Networks are much more flexible and scalable than point-to-point interconnects or shared resources. A popular example for a network involving a huge number of heterogeneous components is the internet. However, due to advances in integration density and increasing chip complexity networks are becoming an appropriate solution for on-chip interconnects as well. Such a network is referred to as Network-on-Chip (NoC).

The group of network-based interconnects can be further distinguished by mainly three characteristics as proposed by Salminen et al. in [**10**][2]: switching characteristic (2.4.3.1), topology (2.4.3.2) and routing (2.4.3.3).

[2] Salminen et al. refer to point-to-point connections and buses as network topologies. In this work they are not considered to be networks due to the lack of switching elements.

2.4.3.1 Switching Characteristics

The switching characteristics can either be *circuit*- or *packet-switching*. Circuit switching means that before sending data, a dedicated path which is referred to as *channel* has to be established from the source to the target. Every network segment used for a certain channel cannot be used for another channel at the same time. As long as no changes are made to this allocation, the network behaves and performs similarly to point-to-point interconnections. Nevertheless, there is some overhead to construct the channel.

When using packet-switching, the segments within the network are shared by several connections. Therefore, the data is split into several *packets* and additional information about the source, target and the content is added within a *header* to allow correct delivery. The switching elements forward packets according this information and are referred to as *routers*. This technique promotes high flexibility and optimizes the utilization of the resources: connections are not bound to a specific channel and alternative routes may be taken. Workload can be distributed evenly throughout the network, failing components can be avoided and advanced features such as different priority levels are easy to implement. However, this comes with a lot of overhead: with every packet, additional data for the header is required and the packets have to be processed by every router along the path. This results in low throughput, and latency increases. Furthermore, routers are rather complex and require more area and power than the switches of the former approach. Finally, additional issues like packets arriving in the wrong order may have to be considered.

2.4.3.2 Topology

A wide variety of network topologies has been studied. Some popular examples are given in Figure 2.3.

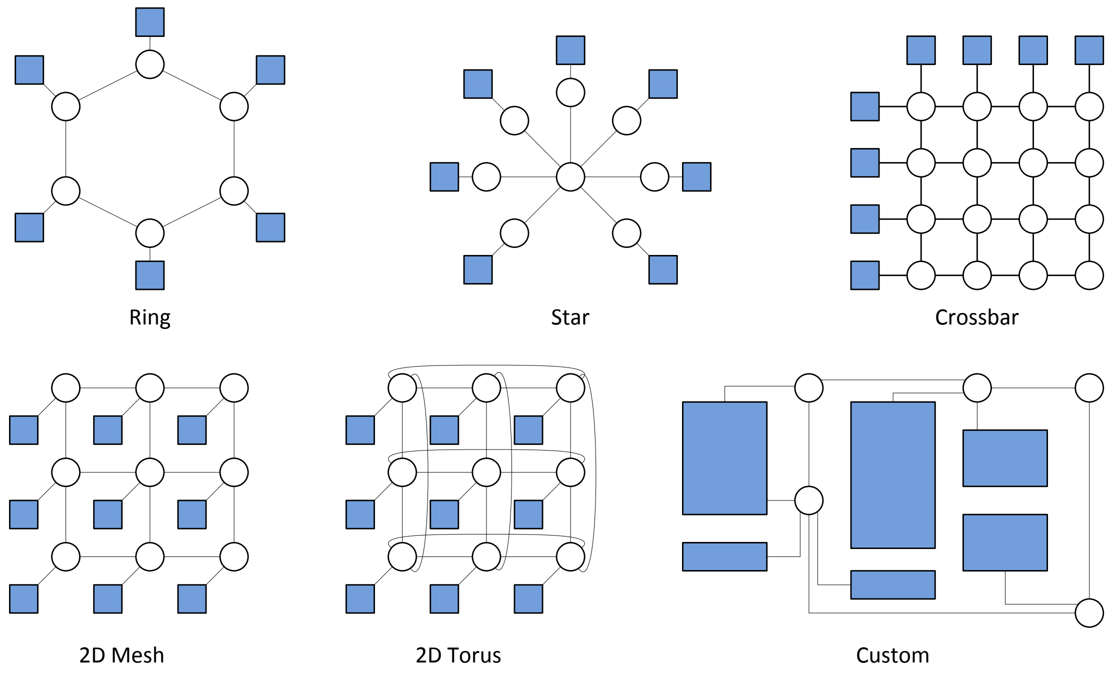


Figure 2.3: Network topologies

Each topology can be implemented using either circuit- or packet-switching. However, some topologies suggest a specific approach. The ring, for example, behaves poorly if implemented using circuit switching: if, without loss of generality, a channel is established to connect the modules on top and bottom, the remaining modules on the left or the right are completely blocked from communicating since there is no segment available to connect them. On the other hand, the crossbar as depicted in Figure 2.3 should be implemented using circuit-switching, since otherwise the overhead due to packet processing is needlessly large.

2.4.3.3 Routing

According to [10] routing can be *deterministic* or *adaptive*. Deterministic means that all data routed between two given modules always follows the same path through the network. When adaptive routing is implemented, different packets between the same modules may take different routes. It is obvious that routing in circuit switched networks is always deterministic, as long as the channels remain established. However, if a channel between two modules is torn down and then reestablished, other segments may be used. If this is the case, a circuit-switched network may be adaptive to a certain extent. As always, adaptive routing offers more flexibility and fault tolerance but also involves additional issues like out-of-order delivery as mentioned previously.

2.4.4 Hybrid Interconnects

Hybrid interconnects combine one or more of the former approaches. For example, in order to create a hierarchy within a multiprocessor system, a bus could be used to connect each processor with a local cache, peripherals and a network interface, and a network could be used to interconnect all of these interfaces to provide inter-processor communication.

The main advantage is that positive attributes of several architectures are combined. In the example above, the bus can provide low-latency access to the components, but only as long as the number is small. Thus, a network is introduced in order to overcome scaling problems of the bus architecture. For another example it may be assumed that a system requires high throughput for application data, but low latency is necessary for some synchronization messages as well. In this case several point-to-point connections could be used for the time-critical messages, while a high bandwidth network is responsible for the application data.

The main drawback is reduced flexibility of the architecture. When several methodologies are combined like in the examples above, the resulting interconnect is optimized for a certain situation but may be inappropriate for other cases.

3 Related Work

In the field of interconnects for partially reconfigurable systems, several solutions have already been proposed. Although a Network-on-Chip may be favored for flexibility and scalability reasons over rather simple structures like buses, actual platforms for partial reconfiguration like the Erlangen Slot Machine (ESM) [**11**] and the work of Ullmann et al. [**12**] are still using less complex approaches. The reason is the following: in order that benefits can outweigh the drawbacks introduced by a NoC, a system of more than just a few modules has to be designed: as long as the number of modules involved in communication remains small, the overhead introduced by additional switches and management is comparatively high. But actual systems being implemented on FPGAs currently do not utilize more than a low number of PRMs; using a NoC is disadvantageous not only in respect of implementation complexity but also in bandwidth, delay and area.

Because of this, the communication infrastructure used by the ESM is introduced in section 3.1 as an example. This infrastructure is called Reconfigurable Multiple Bus (RMB) [**13**] and was further developed into an easy-to-use solution for both dynamically reconfigurable and static System-on-Chips referred to as ReCoBus [**14**]. Subsequently, two different Network-on-Chips are analyzed. The first proposal by Marescaux et al. presented in section 3.2 was published in [**15**] 2002 and further developed in [**16**] one year later, while CoNoChi (Configurable Network-on-Chip, [**17**]) surveyed in section 3.3 is a more recent approach, which really exploits the capabilities of partial reconfiguration. Finally, in section 3.4 the implementation of a high-performance crossbar-switch targeting Xilinx FPGAs [**18**], just like the subject of this work but in a static context, is examined.

3.1 ReCoBus

The ReCoBus architecture was designed to provide inter-module communication within a partially reconfigurable platform, but can also be used in static SoCs. All classes of bus signals that follow established on-chip standards such as AMBA and CoreConnect (*shared* and *dedicated read/write*) are supported. The interface of the bus is depicted in Figure 3.1.

For each module, dedicated interrupt and selection signals are supplied. Furthermore, shared read/write data buses as well as an address bus exists, and one signal specifies whether in the current cycle the bus is driven by the master (write) or by slaves (read).

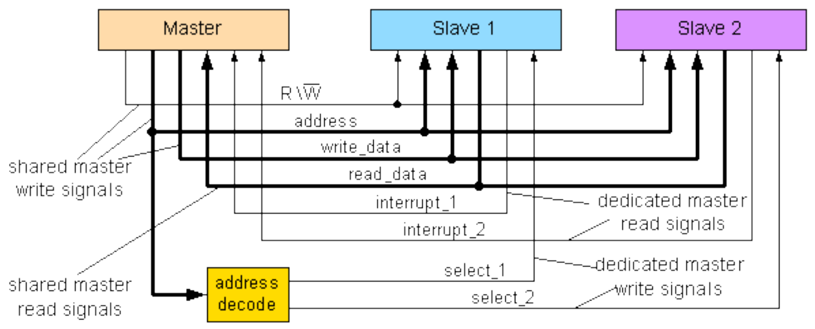

Master
Slave 1
Slave 2
R \W
address
write_data
read_data
interrupt_1
interrupt_2
shared master write signals
dedicated master read signals
dedicated master write signals
shared master read signals
address decode
select_1
select_2

Figure 3.1: ReCoBus interface [19]

For the PRRs into which modules can be placed, it is proposed to divide the reconfigurable FPGA area into several *slots*. These slots should be rather narrow, to provide a higher placement flexibility for modules that span one or more slots. Furthermore, narrow slots reduce the internal fragmentation of the modules, since the region boundaries can be tightly placed around the module. An interface has to be given at an exactly known location within each slot, so that any of them is universally valid for all arbitrary modules.

The ReCoBus is delivered as one single hard macro, which is completely placed and routed. In doing so, adding and removing modules is similar to plugging/unplugging boards into the fixed PCI slots of some desktop computer. This provides a simple abstraction for working with partially reconfigurable modules. On the other hand, the place and route process of the remaining static design is made more difficult, since several resources are unavailable. For example, if a certain hard macro wire is also required for a critical connection in the static design, it is not possible to consider reallocation of this wire. Upon other terms, different allocations of this wire could be evaluated in order to optimize overall performance.

3.1.1 Design and Implementation

A special issue has to be solved when implementing the shared read signal. For a shared read, a bus multiplexer has to be provided for each data line. A possible implementation for several slots can be a multiplexer chain as shown in Figure 3.2.

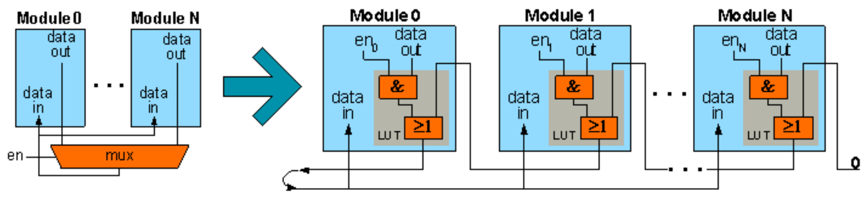


Figure 3.2: ReCoBus shared read multiplexer [19]

But because it is desired to have narrow slots, the slot count will be higher than the number of modules. As a consequence, the logic overhead for the chained structure is very high, and even more important, the combinational path is long and signal propagation takes more time. This issue was solved using interleaved multiplexer chains such as in Figure 3.3.

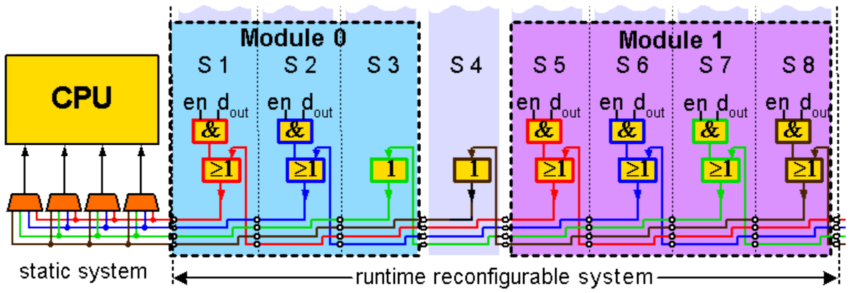


Figure 3.3: ReCoBus multiplexer interleaving [19]

In this example, four interleaved multiplexer chains are provided. Although 8 slots exist in the system, each chain is only 2 segments deep. It is also clear from this example, that in order to provide a specific interface width, the module has to span a minimum number of slots. Assuming that each multiplexer chain represents a set of 8 signal wires, the bus provides a total width of 32 bits and at least four slots have to be used by a module, to access the full bandwidth. However, this is considered to be noncritical, since practical experience showed that the interface width is usually related to the complexity of a module. So if a wide interface is needed, the module is probably complex (=large) and will utilize several slots anyway.

3.1.2 Evaluation

Despite using interleaved multiplexer chains, the worst-case delay is still very high. In [**14**] the propagation delay of this critical path for a 32-bit wide bus with 4 interleaved multiplexer chains is evaluated as shown in Figure 3.4. The entered parameter λ resolves to $\frac{R}{N} - 1$, whereby R is the number of available slots and N is the number of interleaved chains. W in the graph refers to the width of each slot in number of CLBs.

For a high-performance bus clocked at a frequency of at least 100 MHz, the propagation delay must be lower than 10 ns. With no further optimization, this bus could not be used to connect more than 16 slots on most FPGA devices. To overcome this problem, pipeline stages were added. After applying this enhancement, the propagation delay was reduced considerably as visible in Figure 3.5. A test system has been implemented on a Virtex-II device using the enhancements just mentioned. The system consisted of 8 slots being 2 CLBs wide and a bus of 32 bits. Each slot contained a very simple test module like a counter or an adder. The achieved data rate was reported to be 800 MB/s at a clock speed of roughly 100 MHz.

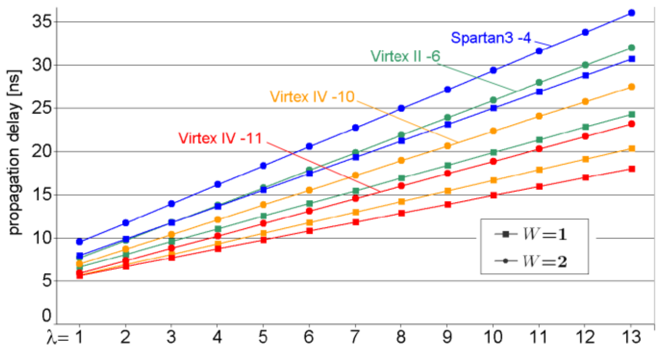


Figure 3.4: ReCoBus delay with no pipelining [14]

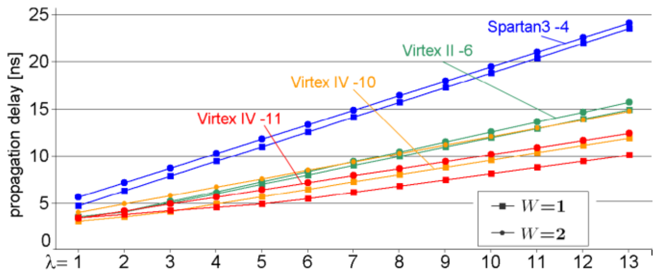


Figure 3.5: ReCoBus delay with pipelining [14]

3.2 NoC: Work of Marescaux et al.

The Network-on-Chip proposed by Marescaux et al. is intended to be a communication infrastructure within a SoC for application data only. This means control and configuration data for the system is supposed to use other infrastructures concurrently existing on the chip. Control of these networks is achieved by a so-called operating system for reconfigurable systems (OS4RS) [**20**].

Only the network for application data is examined here. It uses packet-switching and the topology can be chosen, but is fixed during runtime. In addition, a specific routing algorithm is realized. As always, the main network components are the routers. But besides these, separate network interfaces called *net cells* are used to wrap each module.

According to [**15**], the tightest constraint for an on-chip network is the limited space on the single chip. Most network interconnects realized in the past target multi-chip circuits and can be found in the world of multi-processor computing. In this case, however, the communication architecture has to be implemented together with all remaining components within a single FPGA. So the main challenge for the NoC design is to keep the resource overhead low. Therefore, the components were designed to be as simple as possible. The following design decisions are affected by these restrictions.

3.2.1 Routing

According to the classification in section 2.4.3.3, a deterministic routing algorithm is implemented; nevertheless a decent amount of adaptability is provided. This is offered by routing tables attached to each router (see section 3.2.2) which can be modified during run-time by the operating system. Virtual cut-through is used as the switching technique: when a packet arrives at some router, forwarding can already be started before the packet has been received completely. In this way, the latency of the network is reduced. Virtual cut-through is commonly implemented by splitting the packet into several so-called *flits*, prepended by a header. Wormhole routing is a very similar approach; the major difference is that with virtual cut-through it must always be guaranteed that a packet can be transferred as a whole. This allows flow control at packet level, but large buffers capable of containing a complete packet must exist throughout the network, since forwarding may be blocked: if packets from different sources have to be forwarded to the same target this cannot occur simultaneously. Because of this, each router has to check whether the target is ready to receive, before transfer can be started. This is done by dedicated handshake signals between adjacent routers.

3.2.2 Routers and Topology

As mentioned before, the routers are the main building blocks in the network. Each router is individually parameterizable in the number of input- and output-ports. All of these ports are fully identical; no distinction is made whether another router or the network interface of a module is connected to a given port. This means that all functionality required for higher levels of communication than physical addressing can be implemented within the network interface of the module, which keeps the routers simple. By leaving the number of ports customizable for every router, different topologies can be created. For example in order to realize a 2D-Torus as shown in Figure 2.3, every router would have 5 input and 5 output ports: 4 are needed to connect the adjacent routers, and one port is used for the interface of the corresponding module.

Packet forwarding to the next router is implemented by a crossbar switch, which is contained within every router and can connect any input to any output port. An example is given in Figure 3.6 for a 2 input/2 output router.

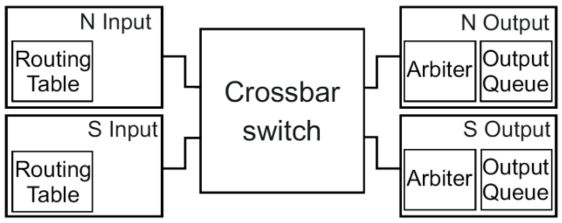


Figure 3.6: NoC router with 2 input and 2 output ports [16]

It is also shown in this picture that one arbiter and one output queue is attached to each output port. The arbiter is needed to select the input port, which is currently allowed to store data in the output queue if multiple inputs have to be connected to the same output. The output queue is necessary, since packets have to be stored when blocked. This can occur because of the chosen routing algorithm, which is also the reason for having one routing table attached to every input port.

3.2.3 Network Interface

The purpose of the network interface is to decouple the module from the communication network, which allows using different clock speeds for the distinct components of the system. Therefore, buffers capable of containing several messages are provided. Furthermore, high-level services like virtual addressing and packet redirection can be implemented in the interface. In the first version of this architecture, even the routing tables were included in the network interface and routers were forwarding the packets only by a given number of hops in X and Y direction.

3.2.4 Evaluation

Not much information was provided about the resources, used to implement one router on a Virtex-II Pro device. According to [**16**], somewhat more than 500 slices are used to implement one router having 5 input and 5 output ports, but additionally BlockRAMs are needed for the message queues. However, the amount of slices used increases quadratically with the number of ports due to the growing crossbar switch size and the arbiters needed for each output port. Unfortunately, nothing is said about the bit with of each provided port. Furthermore, no statistical data is given for the network interfaces (*net cells*).

The network latency is kept low by using virtual cut through switching. Processing the header flit including handshakes and setting the crossbar switch takes three clock cycles. Every following clock cycle one flit is transferred. So the latency is equal to three times the number of hops plus the number of flits for the packet. This appears to be rather low, but is valid only as long as the packet is never blocked. Latency decreases considerably, as soon as concurrent traffic occurs in the network. However, this is exactly what has to be expected when a packet-switched network is chosen instead of less flexible infrastructures. According to [**16**] the maximum clock speed for the network is 50 MHz, which results in data rates of around 100 MB/s between adjacent routers when a bit width of 16 is assumed per port.

3.3 NoC: CoNoChi

Contrary to the approach of Marescaux et al, CoNoChi is intended to be the only communication infrastructure within a given SoC. Thus, it has to be flexible enough to satisfy both the needs for application- and configuration data[3]. A high degree of flexibility is achieved via a complex routing mechanism, which includes several network layers and supports different *message classes* for prioritization and quality of service (QoS).

The routers are the main components of the network, again, and packet-switching is used to distribute the data as well. A separate network interface exists for each module, but not much is said about this element, apart from the fact that it is module-dependent and thereby implemented within the PRM.

However, the main difference to the former approach is adaptability of the topology. This is not only possible by parameterization during design time, but can be done dynamically during runtime. This allows the number of overall routers to be minimized in order to keep the area overhead low.

3.3.1 Routing

A much more complex network protocol is implemented compared to the work of Marescaux et al. First of all, two different addressing methodologies exist: physical addressing, which refers to a specific router within the network topology, and logical addressing, which specifies a so-called *processing unit*. This can either be a PRM or part of a PRM. Routing is based on the physical address only and similar to the former approach, routing tables are used again. Furthermore, a deterministic algorithm is implemented, and adaption is enabled by updating the tables. However, in this case the same network is used to distribute routing tables as for the application data. Packets containing configuration data belong to a special message class and can be distinguished from common application data. Routers recognize these packets and update their routing table if the physical address of the packet matches their own address. Some details are given about the algorithm that is used to calculate the

[3] While application data usually requires high bandwidth and may suffer from some latency, configuration data must be delivered in the shortest possible time but consists of only a few bits

routing tables: the Floyd-Warshall-Algorithm is executed by a dedicated control instance[4] which maintains a global view of the network. So routes are optimized in respect of the number of hops by this technique; however the algorithm is very complex[5]. But if the PowerPC, which is embedded on some Xilinx FPGAs, can be utilized, calculation time is noncritical, since updates of the routing tables are only necessary if some reconfiguration has to be done, which will take longer anyway [**17**].

Packets are always forwarded as a whole, and no technique like virtual cut-through or wormhole routing is implemented in order to reduce latency. Receiving, processing and forwarding the packet takes 5 clock cycles in any case.

3.3.2 Routers

The basic structure of the router is depicted in Figure 3.7, which is quite similar to the proposal of Marescaux et al. However, only the bit width provided at each port can be customized; the number of ports is restricted to the fixed value of 4 (both for inputs and outputs).

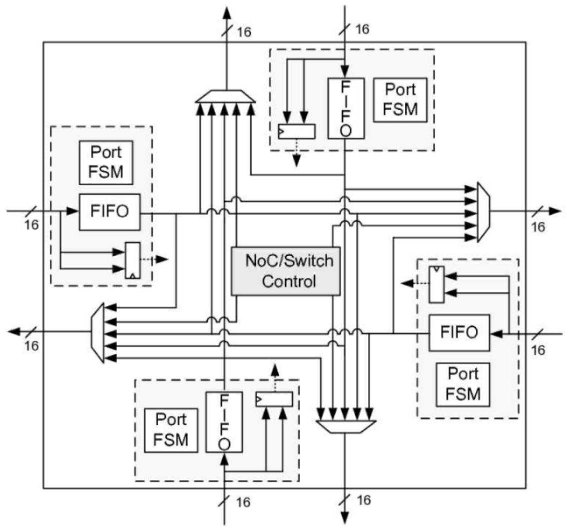


Figure 3.7: CoNoChi router [21]

Because of this, the topology is restricted to a 2 dimensional structure. On the other hand, this goes with the reconfiguration capabilities of recent FPGAs. This is important since the topology is supposed to be adapted during runtime as described in section 3.3.3. Furthermore, all ports are homogeneous and can be connected either to another router or the interface of one module. However, not more than one module can be connected to the same router.

[4] part of an OS4RS or a static hardware component
[5] $O(n^3)$, whereby n is the number of edges in the graph (λ links in a network)

Corresponding to the message queues in the work of Marescaux et al, FIFOs are provided as buffers, which are large enough to store a complete packet. However, packets are stored at the input and not at the output port. Thus only one arbiter is needed per router (labeled as *switch control* in Figure 3.7). Furthermore, there is only one routing table and no distinction on which port a packet has arrived is possible. But multiplexers are used again to connect every input port with all possible outputs.

3.3.3 Topology and Adaptability

As already stated in the previous section, only 2-dimensional topologies can be realized, since the number of ports is fixed. In return, CoNoChi supports adaption of the network topology during runtime by dynamically adding or removing routers from the network. Even the size and location of PRRs can be changed. A detailed description of how this technique is realized can be found in [**22**]; the specific design flow for Xilinx FPGAs is described in this paper as well. Since several limitations exist for the few tools available to create a partially reconfigurable design, it is an extensive process to create such a system. It might be argued that the additional effort is not justified by the gained flexibility.

The basic procedure is to initially divide the FPGA area into several PRRs of the same shape and size, which results in so-called *tiles* as shown in Figure 3.8. These tiles represent the smallest reconfiguration granularity and must be large enough to contain a router. In addition, some static components might reside in a separate region of the chip. Any tile can be configured to be one of four basic building blocks: horizontal or vertical connection wires, a router or the part of a module. Afterwards, bitstreams have to be created for every component and for every tile that is a possible target of this component.

3.3.4 Evaluation

A prototype of the network was implemented on a Virtex-4 device and revealed that 480 slices are needed for the router, which is hardly affected by the bit width provided. In addition, 4 BlockRAMs are required for the FIFOs and routing tables. Because of this, one tile was chosen to be 12 by 16 CLBs large, which is aligned to the reconfiguration capabilities of the Virtex-4 and provides enough resources to contain a router: 192 CLBs contain 768 slices; furthermore 8 BlockRAMs were available on the specific device per tile. However, a significant area overhead is introduced by this approach: especially if a tile contains horizontal or vertical connection wires, the resources are poorly exploited.

The maximum clock speed that could be achieved for the routers of the prototype system was 159 MHz.

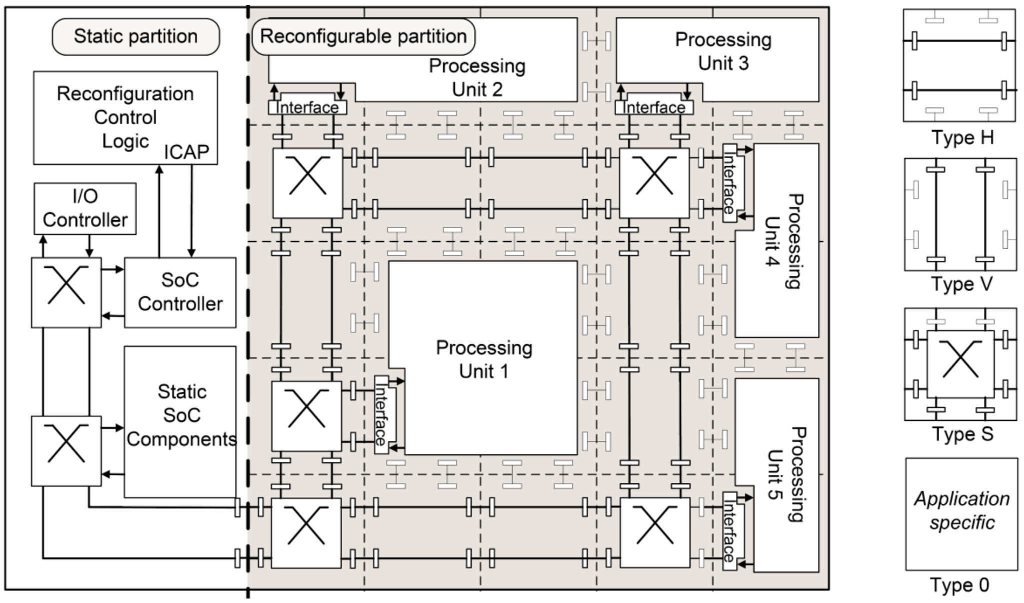


Figure 3.8: CoNoChi architecture [22]

3.4 Xilinx Crossbar Switch

The network architectures introduced so far use packet switching. As mentioned before, this technique introduces some control overhead with every packet that is sent. Circuit switching, on the contrary, introduces overhead only for establishing the channel, afterwards data is transferred more efficiently. A typical example for a circuit-switched network is the crossbar switch. In this work it is examined whether this approach is an adequate solution for partially reconfigurable systems, and a custom crossbar switch is designed for this purpose. However, there are several crossbar switch implementations available, which do not target a reconfigurable environment. The Xilinx Crossbar Switch introduced in this section was motivated by a customer's need for a very large crossbar switch, and a very efficient implementation technique was required.

3.4.1 Crossbar Switch basics

An n × m crossbar switch refers to a structure that is capable of connecting n inputs to m outputs in a matrix manner: traditionally, several switches are arranged in a matrix (see Crossbar in Figure 2.3) and can be switched to short-circuit a horizontal and a vertical wire in order to connect one output with one input. For an n × m crossbar being capable of connecting any input with any output simultaneously, the considerable number of n × m so-called *cross-points* must exist. A straightforward implementation uses large multiplexers for every output which is used to select one of the inputs. However, this requires enormous resources and when regular FPGA logic is utilized (slices and LUTs), limits are quickly reached: As described in [**18**], a 928 × 928 crossbar switch was required, which had to be capable of connecting any input with any output. Within one 4-input LUT a 2:1 multiplexer can be implemented by using one input as select line determining which of the two other inputs is mirrored to the output. Since there are 8 LUTs within one CLB, a 16:1 multiplexer can be implemented within this CLB. Accordingly 58 CLBs are needed for a 928:1 multiplexer. Replicating this multiplexer for every input, 58 × 928 = 53.824 CLBs would be needed. One of the largest FPGAs available at that time, a Virtex-II XC2V6000, provides 8.448 CLBs; the more recent Virtex-4 VLX160 used in this work, as well one of the largest devices in his family, has 16.896 CLBs. Obviously this is not enough for the required design.

In [**18**] the issue was solved using the routing fabric as logic resource. The main component responsible for flexible routing in Xilinx' FPGA architecture is the so-called *switch box*. Actually, this component is getting close to the original idea of a crossbar switch with horizontal and vertical wires. On Xilinx FPGAs these cross-points are called *programmable interconnect points* (PIP), and their state is selected within the configuration bitstream.

3.4.2 Design and Implementation

A detailed description of how the Xilinx Crossbar Switch was implemented can be found in [**23**]. Basically, by utilizing the switch box attached to every CLB, any of the 33 inputs can be connected to at least one pin of every LUT in the CLB. When the remaining inputs of the LUT are tied to logic "1" and the LUT is configured as an *and*-gate at the same time, any of the 33 inputs of the switch box can be selected for the LUT output. Following this technique, every LUT can represent a 33:1 multiplexer together with the switch box. If only 32 of the 33 inputs are utilized the 928 × 928 crossbar switch can be implemented using only 29 × 116 CLBs as depicted in Figure 3.9 (29 × 32 = 928 inputs, 116 × 8 = 928 outputs). However, this structure was further modified in order to improve the aspect ratio of the utilized are. As a result, a square region of 58 × 58 CLBs could be achieved. Furthermore, pipeline stages were included, which enabled higher clock speeds. The netlist for the specific FPGA was created using XDL (see section 2.3.2).

Since the cross-points are not implemented using conventional logic, updating the switch configuration cannot be realized by changing register values or by any other straightforward approach. Instead, the FPGA must be reconfigured in order to change the state of the PIPs and content of the LUTs. But using the convenient flow for partial reconfiguration (EA PR) is not possible: a bitstream had to be generated for every possible combination of switch settings (928^{928} altogether), which is not feasible. If bitstreams were generated in advance, a lot of space would be needed for storing. But complete bitstream generation during runtime is not feasible as well, since the place and route process is very complex and time consuming. The solution was to use JBits, which is a java-based application that is capable of manipulating bitstreams. However, JBits is deprecated and does not support bitstreams for devices more recent than Virtex-II.

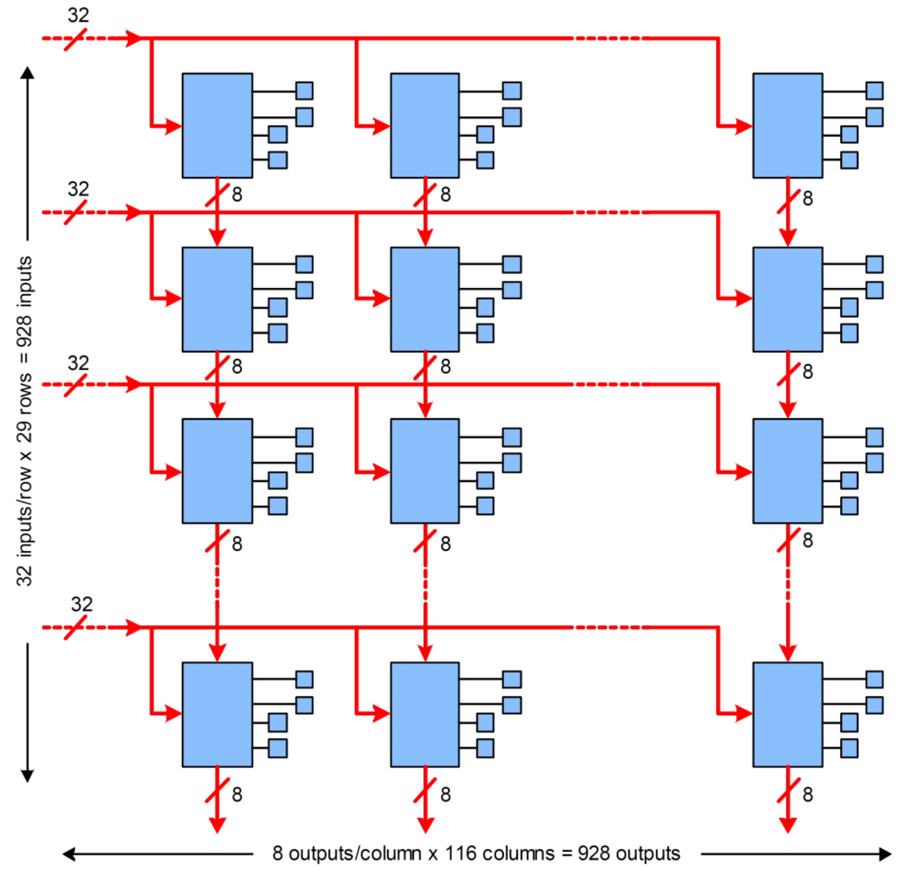

32
32
32
32
8
32 inputs/row x 29 rows = 928 inputs
8 outputs/column x 116 columns = 928 outputs

Figure 3.9: Xilinx Crossbar cascaded structure [23]

3.4.3 Evaluation

The final 928 × 928 crossbar switch was successfully implemented on a XC2V6000 device and occupied 60% of the resources on this FPGA model. A clock speed of 155.5 MHz could be achieved on all channels. This results in an aggregated throughput of 144.3 Gbits/sec, but unfortunately no value is specified for the latency of the structure. A new switch configuration could be written to the device within 220 µs.

3.5 Conclusion

Specific architectures were described in this chapter for the basic bus and network interconnection methodologies. It is apparent that each of the proposals introduces certain pros and cons.

The bus implementation stands out for implementing established standards beeing used in real partially reconfigurable platforms and exceeding the stage of a prototype. This corresponds with the fact that buses are the most widely used structure for static system-on-chips. There is no reason to expect a completely different trend for partially reconfigurable systems (PR systems) which are now emerging. It is merely logical that familiar structures are explored in the very beginning. Furthermore, a bus is quite a simple structure, causing a minimal area overhead and it is easy to use, since just a few fundamental protocol definitions have to be obeyed. But on the other hand, buses are neither scalable nor very flexible, as pointed out in 2.4.2. However, scalability becomes more important with larger SoCs which involve a larger number of more varied components. Flexibility, on the other hand, is needed particularly in a partially reconfigurable system: since certain parts can be changed during runtime, the communication structure should be able to satisfy changing requirements.

Because of this, Networks-on-Chip have appeared as another approach. They provide a much greater scalability and flexibility. Just like the architectures described in this chapter, most other NoC proposals use packet switching, which further enhances these characteristics. Obviously, the major drawback of this approach is the overhead that is introduced both for area and packet processing. But there are still other problems which have not yet been mentioned. Not only is the interconnection more extensive, but the complexity of the connected modules also increases, since the involved overhead applies to them as well. Besides identifying the source of a packet in order to recombine data belonging together, reordering might be necessary within the module, since just a few NoCs provide mechanisms for this. Accordingly, splitting messages into several parts to comply with the maximum packet size is often delegated to the module. The same applies to creating the appropriate headers. Furthermore, it is still an open question whether this higher flexibility is actually needed: it might be argued that even in a partially reconfigurable SoC the communication needs remain predictable to a certain extent, since modules are still working together to execute certain, well-known tasks. But packet-switched networks can play out their strengths particularly in the case of a very large number of participants being completely independent from each other. This does not apply to current PR designs, just because the required integration density for such large systems cannot yet be achieved on FPGAs.

Circuit-switched networks can be considered as an intermediate solution. In contrast to packet-switched networks, sophisticated module interfaces are not required and the control overhead is considerably lower. In addition, more flexibility is possible than with a bus. On the other hand, a new problem may arise as already stated in 2.4.3.2: depending on the network topology, certain components might be prevented from receiving or sending data, even though both communication partners are idle. The crossbar topology circumvents this situation. However, the area overhead is worse again, since it scales as the square of the number of connected modules. Otherwise, it was shown by Xilinx that implementing a very large crossbar switch is possible on FPGAs. Because of this, the behavior of a crossbar switch in a partially reconfigurable system is explored in more depth within this work.

Unfortunately, the Xilinx Crossbar Switch is not applicable for a practical analysis. First of all, it was not designed to be used in a PR design. While this issue could be solved by some modifications, another more serious problem prevents the usage of this solution: the Xilinx Crossbar Switch architecture is incompatible with recent FPGA devices, since partial reconfiguration has to be accessed in an extraordinary manner. As stated in section 3.4.2, bitstream modifications are done during runtime using JBits, which is deprecated. It seems to be rather futile to use an old Virtex-II with limited partial reconfiguration capabilities for present day investigations. Decoding the appropriate parts of the bitstream for the provided Virtex-4 FPGA was also considered to be going beyond the scope of this work. Instead, a custom architecture, as discussed in the next chapter, was designed.

4 Design

For the reasons given in section 3.5, it was chosen to investigate the use of a crossbar switch as an interconnection in a PR system. A new crossbar switch, referred to as *XBar*, was therefore designed and is described in this chapter. First, the requirements for the interconnection are explained, followed by the major design decisions and a reasoning of the choice. Subsequently, the architecture is explained in detail and information is provided on how to embed the crossbar switch into a custom design.

4.1 Requirements

Several requirements apply to the communication infrastructure. Some of them address the unique requirements of partial reconfiguration; others apply to conventional inter-module communication as well. Furthermore, some additional conditions were assumed, which have no general validity, but are required as a consequence of the current state of technological development. However, some requirements are mutually contradictory, and a trade-off has to be made.

Small area overhead. The FPGA area utilized for the communication infrastructure should remain as small as appropriate. Even though devices are getting larger, the possible modules to be implemented do so as well. Furthermore, less area overhead results in higher speed, since delays are reduced and lower costs follow when smaller FPGAs can be used. Finally, power consumption is reduced as well. On the other hand, constraining the available resources always affects the maximum bandwidth that can be achieved.

High bandwidth. Communication between modules may require high bandwidth. In particular, accelerators for multimedia or network applications are often processing data streams that cannot be buffered for a long period of time, since new data arrives constantly. Bandwidth can be increased by providing more wires per interface (= area overhead) or raising the clock speed, which requires additional pipeline stages (= higher latency).

Low latency. Communication between modules may require low latency. If modules perform handshakes for synchronization, the appropriate signals must have a low delay in order to work efficiently. For example, a buffer-full-signal is useless when it arrives several cycles too late.

Fast switching. The time, which is required to establish a new channel or change the destination of an existing one should be short in order to avoid large overheads when communication patterns change frequently. However, at this stage rather fixed channels are assumed, but more flexible switching to new configurations must not be prevented by the architecture (see also *external control* below).

Support for different communication patterns. There are four basic communication patterns that may occur between modules as depicted in Figure 4.1. Any of these cases should be supported without introducing high overheads like sending the same information multiple times.

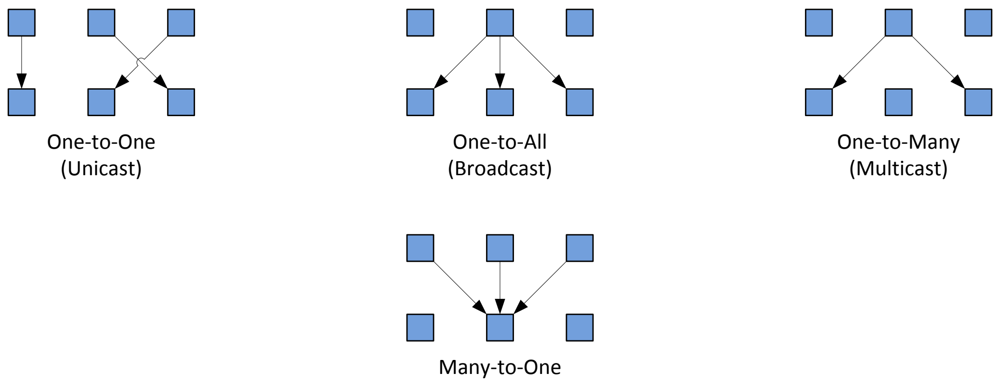


Figure 4.1: Communication Patterns

Support for different clock speeds. The connected modules might run with different clock speeds, so the communication infrastructure must provide asynchronous interfaces. Any module should be able to send or receive data depending on its local clock, and should not be constrained by a global communication clock.

Uniform PRRs. It should be possible to configure a module into any PRR using the same module implementation[6]. This can be achieved by providing a universally valid interface, which is independent of the associated region.

[6] This does not mean that the respective bitstreams are also equal. This may not be possible, because signals of the static design might cross the PRRs at different locations.

Limited number of PRRs. It is assumed that the number of partially reconfigurable regions does not exceed 8. The main reason for this assumption is to ease the prototype implementation, which would be much more complex if more regions would exist: several additional precautions would have to be taken which might be neglected with this simplification. Other reasons for this rather conservative assumption are the problems that occur when a higher number of PRRs is involved on a single FPGA. First of all, available resources are still a limiting factor and it is not possible to provide a high number of partially reconfigurable regions without the need to split some modules to span over more than one region. But this would lead to a complex module design, which should be avoided. Secondly, actual constraints that apply to partial reconfiguration require a distinct bitstream for every PRR. When the number of regions is increased, the number of bitstreams that has to be stored for every module grows accordingly. Furthermore, actual application scenarios for partial reconfiguration only involve a certain number of modules which operate *in parallel*. This assumption does not mean that the overall number of different modules has to be small.

External control. At this stage of the work it is assumed that reconfiguration management and setting up the connections is dedicated to an external controller. However, this is not applicable in all cases. It may be necessary that a module can establish a new channel to another module on demand and without introducing high communication overhead. The architecture of the crossbar should be designed in such a way that adding a module-driven control interface is feasible at a later date.

4.2 Design Decisions

The final design, which is described in section 4.3, is affected by some fundamental decisions which are discussed below. Predominantly they are justified by the requirements defined in the first section of this chapter.

Use circuit switching. Circuit switching was chosen as the switching characteristic of the design. This option is quite obvious because of the topology associated with a crossbar – there are many more interconnection wires and switches than modules. Actually, the number of cross-points and links grows quadratically. This only makes sense if communication is realized by exclusive channels between the modules: since alternative links exist, other components are not prevented from communicating despite the fact that some links are already assigned to other channels. In contrast, if packet-switching was used, interconnections could be shared, and the additional links would not be exploited efficiently.

Parameterize interface width and number of PRRs. As mentioned before, providing a high bandwidth affects the utilized area and vice versa. This issue cannot be solved in general and is tackled by providing a parameter for the bit width of the interface, which is the crucial factor. This allows the trade-off to be done by the user according to his constraints. However, the value applies to the interface of all PRRs, because all of them have to be equal in order that modules can be similarly placed in any PRR. The overall number of PRRs is parameterized for analogous reasons.

Bundle wires to reduce complexity. Channels cannot be established on a per-wire level. Instead, if one module should be connected with another one, a channel providing a bit width of a multiple of 7 must be established. This particular value results from some constraints arising during implementation: an efficient cross-point implementation exists for bundles of 8 wires (see section 5.1). However, one wire is needed for indicating data presence (see section 4.3.4). As a result, each interface of n bits is divided into $k = n \div 7$ bundles which can be switched independently. However, this removes some flexibility and channels might not be fully exploited, for example when a bundle is used to implement a single handshake signal. On the other hand, the overhead to manage connections is significantly reduced. An efficient implementation of the control logic would be hindered by a finer grained switching mechanism. Furthermore, it can be expected that the proportion of modules requiring only a very narrow channel will be low: very short messages usually contain control information, which are typically associated with some additional payload. Because of this, cannels often have to be wider anyway.

Remove certain cross-points to reduce area and wire delay. A crossbar switch consists of a large array of cross-points. If there is one of these switching elements for every pair of wires or wire groups, their number will be large: if N is the number of modules and k is the number of bundles per module, $k^2 \times N^2$ cross-points are needed. Unfortunately, every additional switch not only causes an increase in area, but also in propagation delay. However, it might be feasible to improve this structure: in the case that it does not have to be possible to connect every input with every output, several cross-points may be removed. Of course, this precondition does not apply without constraints: it must always be possible to connect any module with any other module, regardless in which PRR it is placed. Furthermore, the interface must remain universally valid: if bit n of interface i can be connected to bit m of interface j, it must be possible to access bit m of any other interface in some way as well. According to these requirements, certain cross-points were excluded as described in section 4.3. Afterwards, only $k \times N^2$ cross-points remain, which is still quadratically dependent on the number of modules, but only linearly dependent on the provided link width.

Use buffers between clock regions to allow asynchronous operation. In order to support asynchronous operation of the interconnected modules, data has to be buffered before and after being transferred through the switch. Simple FIFO message queues were chosen for this purpose, because the additional control cost is low. However, distinct FIFOs for the different bundles of each interface are required, because bundles may need to remain independent.

Do not use partial reconfiguration to update switch configuration. As pointed out in section 3.4.2, it is not possible to change the state of cross-points by partial reconfiguration following the common flow for PR designs. Nor is a solution like JBits available for recent FPGA devices. Since the exact content of the bitstream is kept a secret by Xilinx, manual modification seems not to be an alternative. Because of this it was chosen not to make use of partial reconfiguration to control the switch settings, although this is a serious drawback, since some potentially more efficient implementations are prevented.

Design certain elements at a low-level. It was decided to implement the switching structure in a low-level format and not to utilize some higher level design language like VHDL for this purpose. The first reason is that a higher logic density can be achieved by optimizing the design for the special logic resources of FPGAs. A second reason is that the array-like structure of the crossbar does not have to be abstracted to basic components like multiplexers before they are mapped to FPGA resources again, and the view of horizontal and vertical wires connected by cross-points remains valid. Furthermore, the implementation can be oriented towards the Xilinx Crossbar Switch. By this it is possible to introduce partial reconfiguration with less effort, when access to the bitstream does become available.

4.3 Architecture

This section describes the architectural design of the new crossbar switch called *XBar*. Basically, the design is divided into two parts. The first part is the higher-level design covering the interfaces and buffers, as well as the control logic. The other part is the low-level design of the switching structure, also providing separate internal interfaces, and can be viewed as a black-box module from the higher level. But first, the external view of the switch and the supported features are described. Subsequently, the interaction between the main building blocks is detailed.

4.3.1 Overview

When restricted to data links only, the XBar offers two distinct interfaces to every partially reconfigurable module: one for sending data, and one for receiving data. An abstraction of a partially reconfigurable region which is compatible to this (incomplete) XBar interface is given in Figure 4.2. As mentioned before, a set of wires is always bundled together, which results in having *k* bundles that can be switched independently. Because of this, every interface is also divided into *k ports*. Each of them can be utilized to establish a connection which is independent of the others, except for the fact that they are synchronized by the same clock related to the PRR. This limitation could be avoided, but assuming only one clock for each module seems to be reasonable. The number of input and output ports is always the same, because the amount of sent and received data within the whole system is assumed to be in similar dimensions. However, an asymmetric design might be considered (see chapter 7).

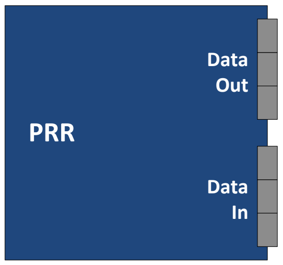


Figure 4.2: Abstraction of a PRR

The crossbar switch must provide the possibility to connect any output interface of the various PRRs with any input interface. In particular, this can also be the input of the same region, in order to provide some feedback to a module. A small system consisting of two PRRs with three ports per interface and the corresponding XBar in between is depicted in Figure 4.3. Every partially reconfigurable region is shown both at the top and on the left side of the switch. Of course, this is not the actual arrangement for implementation, but it is useful to illustrate the interconnections. Each region actually only exist once. Following this abstraction, all XBar inputs are on the top and all outputs are on the left. Inside the crossbar, basically a set of horizontal and vertical wires exist. It must be kept in mind that wires are bundled together, so each line in the picture below forms one of these bundles. As mentioned in section 4.2, cross-points do not exist for every pair of links. In Figure 4.3, every cross-point is represented by a black dot. It can be seen that every bundle coming from the top can be connected to one and only one horizontal bundle for each specific PRR. Furthermore, the appropriate horizontal bundle is always connected with the same interface port, regardless which PRR is addressed. Thereby it is ensured that interfaces are independent of the region that they belong to.

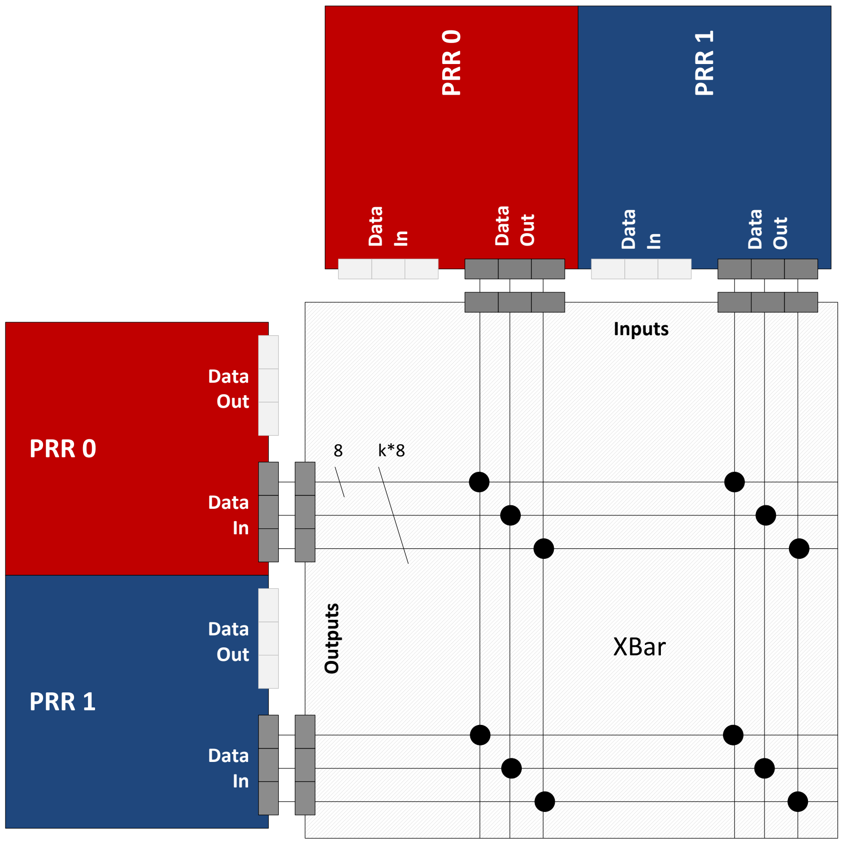


Figure 4.3: XBar schematic view

A problem introduced by removing cross-points might be that modules have to be designed in such a way that some functionality is provided at a specific port, since ports are not interchangeable any longer. In practice, this should not be a major problem, because the modules are usually designed for co-operation anyway. However, when IP blocks are supposed to be used, adjusting the interfaces might not be possible. This issue can be solved by introducing adapters between the PRR and the crossbar switch as shown in Figure 4.4. The adapter could be updated using reconfiguration every time another module is configured into the adjacent PRR. Thus, no change will have to be made to the given module.

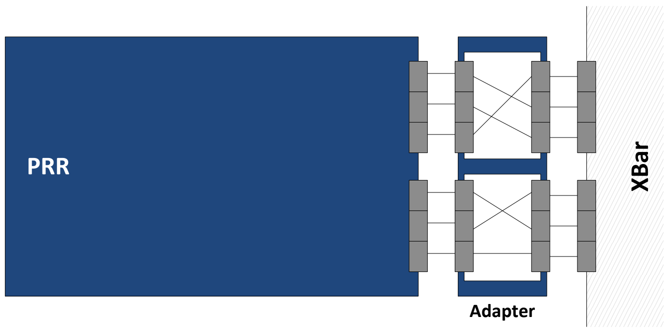


Figure 4.4: PRR with adapter

Furthermore, all the required communication patterns are supported. In the case of unicast, data can be transferred very efficiently since all available links may be utilized simultaneously and high bandwidth can be achieved. Multicast and broadcast obtain similar results since data has to be sent only once to reach all destinations: all horizontal wires can be driven by one module simultaneously. However, if data has to be received from several sources within the same module (according to the Many-to-One pattern), the corresponding interface has to be shared, since every horizontal wire may only be connected with one vertical wire at a time. Sharing can be realized in two different ways. First, the interface can be shared along the ports: any port can be connected to another module and data can be received from all modules in parallel. However, the maximum number of different sources is constrained by the number of ports in this case. Another solution is to share the interface in time: data from different sources is not received simultaneously; instead only one module is connected at a time. The complete interface may be used then and there is no limitation in the number of sources. But the control overhead to manage time sharing is high: an arbiter has to decide which module is allowed to access the shared ports and data sent by others in the meantime has to be buffered. Furthermore, the addressed module cannot distinguish between the different sources without additional information as might be provided by tagging the data.

Because the control overhead should be minimal, time sharing is not currently supported. Moreover, the drawback of the first approach is only minor, since the overall number of modules is supposed to be small and it is unlikely to expect a high number of different sources.

4.3.2 Interfaces

As mentioned before, the interface does not consist of pure data links as assumed in the previous section. The main issue in this context results from the fact that a PRM may run with a different clock speed than the crossbar switch. Data cannot simply be sent out every clock cycle; instead transfer has to be controlled. This should be achieved in a straight-forward manner in order to keep the overhead low.

FIFOs are used to allow asynchronous operation of the components. In the case of XBar inputs, the input of each FIFO is synchronized by the clock of the PRM, while the output is synchronized by the clock of the crossbar switch. Analogously, this is done exactly the other way round in the case of XBar outputs. So first of all, the clock of the PRM has to be distributed to the XBar. Furthermore, *read enable* (*rd_en*) and *write enable* (*wr_en*) signals are provided, which can be used immediately to control the FIFOs, which is the easiest way to control the dataflow. However, since data might be stored in a specific FIFO more frequently than it can be consumed, an *overflow* must be prevented. Therefore, the *full* signal of each FIFO is passed on to the PRM in order to prevent it from sending further data. Accordingly, FIFOs may run *empty*, which also has to be signaled to the module at the switch output.

Since any port should operate independently of the others, the FIFOs and signals have to be provided as part of the discrete ports. The resulting interface for one PRR is depicted in Figure 4.5. In this example, two input and two output ports exist and the maximum channel width will be 14 bits, because 7 data bits are associated with every port.

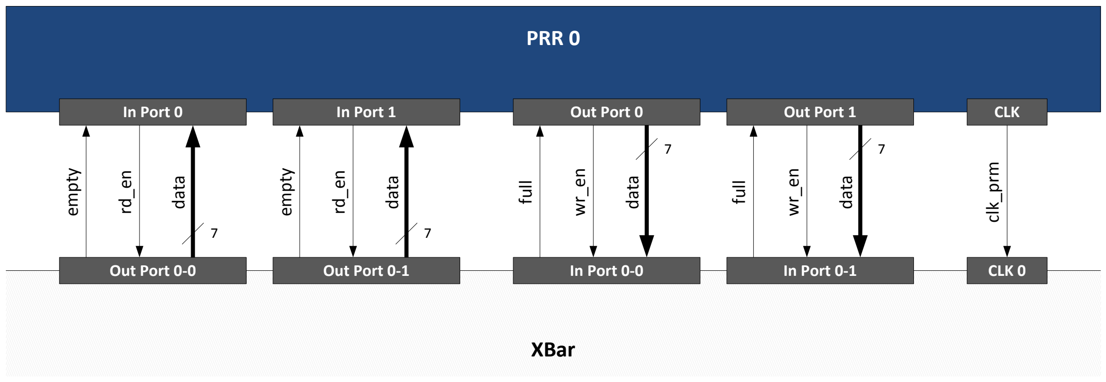


Figure 4.5: XBar interface

For the FIFOs themselves, IP-blocks are instantiated. The COREGenerator introduced in section 2.3.1.1 is used to create the IP. Several parameters can be specified, the most important are the following: *operation mode*, *implementation*, *data width*, *data depth*, provided *signals* and the *threshold values* for full and empty states.

For *operation mode* asynchronous is chosen, which will allow using the FIFO between two clock regions as assumed in this work. *Data width* must be 7, because this is the number of bits available for the payload at each port. *Implementation* refers to the choice if the FIFO should be realized using Block RAM or Distributed RAM. This property can be chosen by the user at design time of the crossbar switch. *Data depth* is another parameter that can be defined by the user. The value must be a power of 2 and the minimum is 16. Suggestions for appropriate settings can be found in section 5.2. The *signals* provided are chosen to match the description made previously in this section: *rd_en*, *wr_en*, *empty* and *full*.

The threshold value for *full* is $FIFO_{dept\,h} - 5$, since the control logic has to know the buffer state five cycles in advance in order to stop further transfers early enough: another five entries might still follow, although the full signal of the output buffer is asserted. This is because it takes the control logic one cycle to react to the FIFO states, and up to four entries per channel may be present in the switching structure simultaneously (see Section 5.1). Their delivery cannot be stopped anymore. The threshold value for *empty* is zero, so *empty* is asserted if and only if the corresponding buffer is really empty.

4.3.3 Switching Structure

The heart of XBar is the internal switching structure, which is designed to match the FPGA resources as closely as possible. Furthermore, it was decided not to use reconfiguration capabilities which would enable routing resources to be part of the logic implementation (see section 3.5). Instead the standard components, namely the CLBs, have to be used. It is possible to implement a cross-point between two bundles, represented by a black dot in Figure 4.3, within a single CLB (see section 5.1). Because of this, and the fact that CLBs are located uniformly throughout the FPGA area, a dense mesh of CLBs was used to implement the array of cross-point switches. Following this approach, a four by four cross-point structure can be mapped to an array of 16 CLBs as depicted in Figure 4.6. A more detailed description of the realization, especially how to keep the structure closely meshed after removing cross-points can be found in section 5.1.

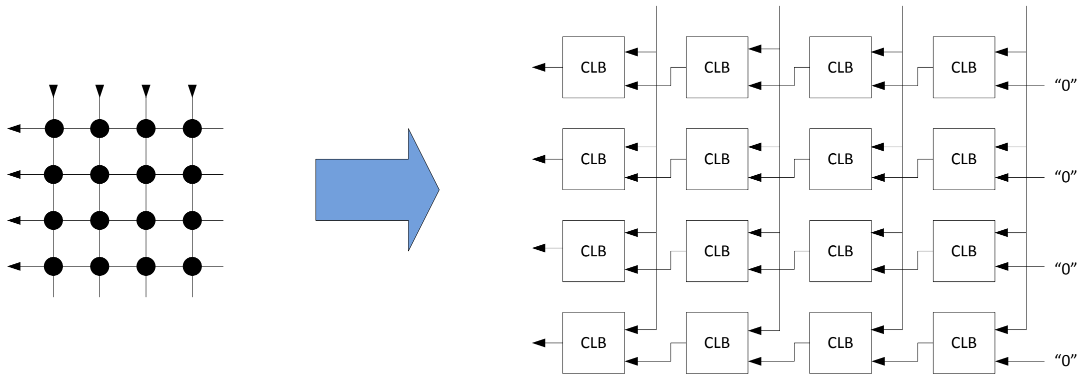


Figure 4.6: Basic switching structure

It is not possible to realize this structure using a common hardware description language, thus it is implemented independently of the other components and stored as a *hard macro*. How to merge this macro with the remaining implementation of the switch is explained in Section 4.4.

4.3.4 Control Logic

The hard macro introduced in the previous section is a very static structure. Data is forwarded, without further ado, based on the state of the cross-points. However, these states have to be configured to match the current communication needs. Control logic is introduced for this purpose. But as mentioned in Section 4.1, at this stage of the work the interconnection management is implemented within an external controller, which is also responsible for reconfiguration. Accordingly, the controller within the crossbar acts as an interface between an external host and the state selector of the individual cross-points (called *xconfig*) only.

Furthermore, the control logic must manage the internal dataflow. The necessary effort for one pair of FIFOs is shown in Figure 4.7.

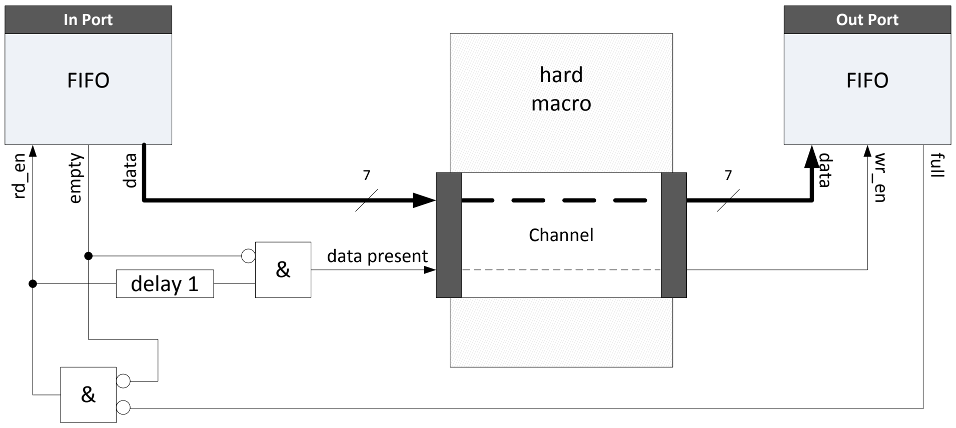


Figure 4.7: XBar FIFO control logic

For a better understanding, it is assumed that the depicted input FIFO is assigned permanently to the output FIFO of the figure. Of course this is not necessarily true, since the state of each cross-point may be changed at runtime. When data is available at a specific XBar port (or within the corresponding FIFO, respectively) it may be passed on to the switching structure by asserting *rd_en*. This signal is also propagated through the switching structure together with the payload, in order to signal the presence of data. Because of this, only 7 of 8 available wires per bundle can be used for the payload as mentioned before. After a certain delay, data will become valid at the hard macro output and *wr_en* of the destination

FIFO is set automatically. But since data from the input FIFO will not be valid before one additional cycle after *rd_en* was asserted, *rd_en* of the previous cycle must be considered. Furthermore, *rd_en* stays asserted by mistake for one cycle, if the last entry of the FIFO was just read. In order to deassert the data present signal on time, the *empty* signal must be taken into account, too (see timing diagram in Figure 5.5 for a better understanding). The result can be used to signal presence of data correctly. The reason for sending this control bit through the hard macro is that there is then no need for the internal controller to consider the exact latency of the switching structure.

Furthermore, data must not be transferred if the target FIFO is full. Because of this the appropriate *full* signal has to be evaluated by the control logic before *rd_en* of the source may be set. The *and* gate with two inverted inputs serves this purpose. Since it takes several cycles until data has passed the hard macro and the new FIFO state has been evaluated, the FIFOs are configured to advance *full* by five entries.

Figure 4.8 shows how the previously described components fit together. The parts visible in the diagram match the interface which is shown in Figure 4.5.

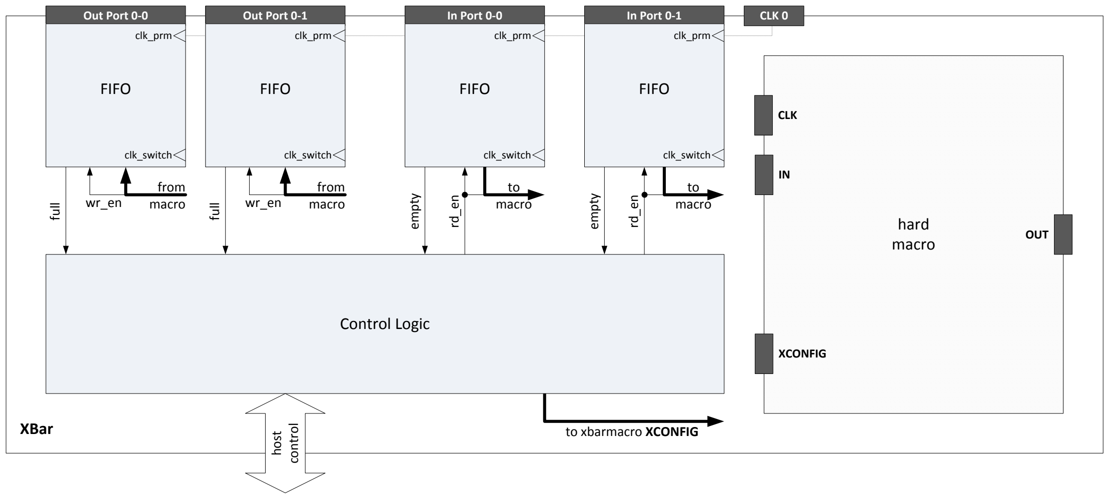


Figure 4.8: XBar block diagram

4.4 Design Flow

A standard design flow for a PR design was introduced by Xilinx with the EA PR overlay for the ISE design suite. However, the latest version supported by this overlay is ISE 9.2i, while the most recent version available is ISE 11.4. During this work it was officially announced that the EA PR flow, based on a modular design flow, as described below and in [**24**], will become deprecated with the release of ISE 12.1. EA PR will stay available for researchers in the Xilinx University Program (XUP) only.

The main difference between the EA PR flow and a conventional FPGA design is that mor than one pass of the implementation tools is necessary. The design is divided into discrete components and a separate synthesis, mapping, placement and routing process is required for each component. Every PR design is composed of the following parts:

Top-level design. Contains all user constraints (I/O pins, timing, etc.) and describes how the remaining parts belong together: PRRs are defined here as well and bus macros have to be placed at the boundaries for crossing signals

Static design. Contains all components not being reconfigured during runtime

PRMs. At least two PRMs must exist, in order for a PR design to make any sense

Merging the static part and the PRMs is eased by the PlanAhead tool. The different components can be imported into one project; the definition of PRRs and bus macro placement is simplified by a graphical user interface. However, it is recommended to analyze the interaction of system components by implementing a complementary non-PR design first. This is represented by the steps ○,1 to ○,4 in Figure 4.9, showing the complete EA PR design flow.

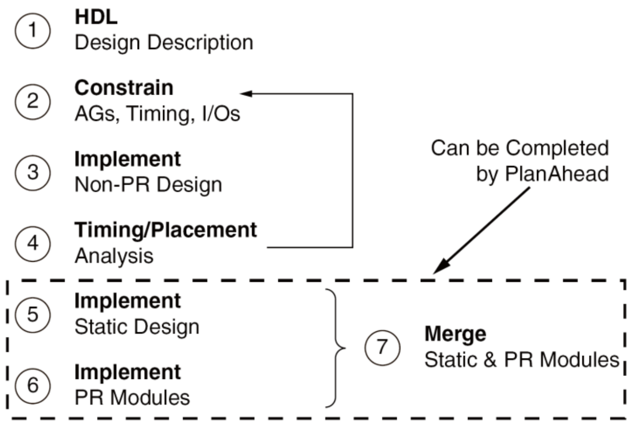


Figure 4.9: EA PR design flow [24]

The crossbar switch belongs to the static design. But the switch implementation itself consists of several parts again: the hard macro, designed using XDL, the FIFOs, which are IP-blocks and finally a VHDL entity description also containing the control logic. Because of this, a custom design flow as shown in Figure 4.10 was introduced, which also demonstrates how to merge the XBar with the remaining static components of a PR design.

The starting point of any new design is the XBarGenerator; a Windows based application, which is used to create a custom version of the XBar. After the parameters are defined by the user, several files based on these values are created:

XDL File. Contains the complete XDL design of the switching structure

SCR File. Contains additional information to convert the XDL design into a hard macro

VHD and PCF Files. The VHD files contain the entity description of the hard macro and XBar. Furthermore, a synthesizable VHDL implementation is provided. The PCF file is passed on to the place & route (PAR) tool and contains physical constraints to ensure the correct propagation delays within the hard macro. No additional modification needs to be done on these files by the user.

XCO File. Contains COREGenerator parameters which are used to create the appropriate FIFOs

Two additional steps are required before the XBar is ready to use: First, the XDL design has to be converted into a hard macro (NMC file). Therefore, the first step is to convert the XDL design into an equivalent NCD file, which is achieved by the `xdl` command provided with ISE. Unfortunately, it is not possible to convert the XDL design directly into an NMC file, since the xdl tool removes all *hard macro external pins*. However, they are an essential element, since they define the interface of the hard macro. So, in a second step the external pins must be restored. This is achieved by an FPGA Editor script (SCR file) parsed by the `fpga_edline` command. A batch script automates the NMC generation. The created hard macro contains all components and nets of the switching structure. However, the nets are not routed, in order to provide more flexibility in the place & route step for the merged design.

What remains is to import the resulting files into the ISE project of the static design. In addition, the FIFO IP-block has to be generated. This happens automatically if the XCO file was imported correctly into the ISE project. Subsequently, the crossbar switch is ready to be instantiated using common VHDL code.

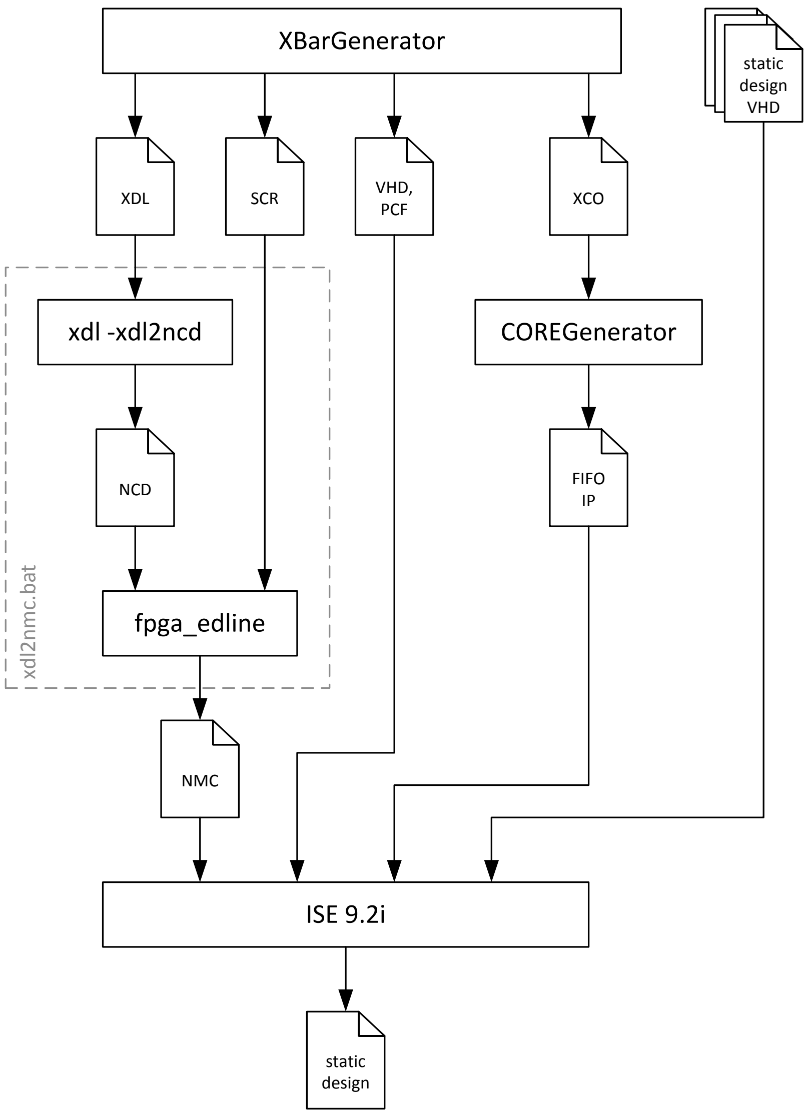
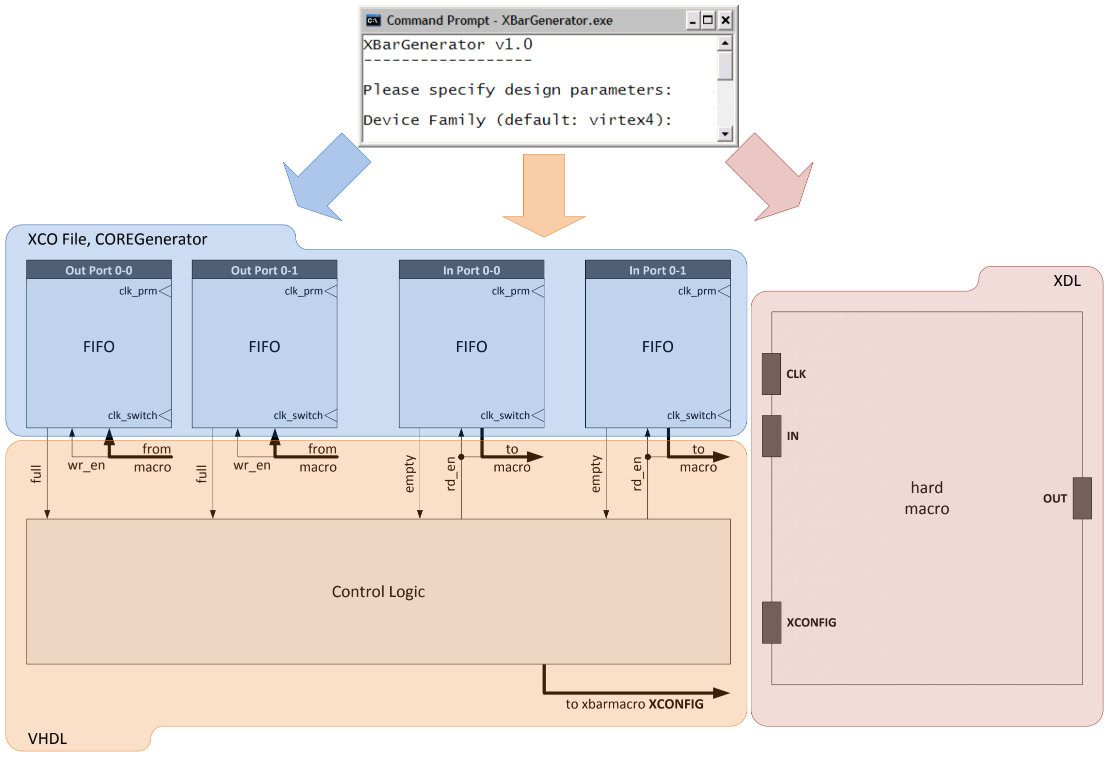


Figure 4.10: XBar design flow

5 Implementation

In this chapter the implementation part of this work is described in more detail. Following the design flow introduced in section 4.4, implementation can be divided into four discrete domains as shown in Figure 5.1: **VHDL** (XBar top-level entity and control logic) – **CORE-Generator** (FIFOs) – **XDL** (switching structure) – **C++** (XBarGenerator, which is creating the other sources).



```
XBarGenerator v1.0
------------------

Please specify design parameters:
Device Family (default: virtex4):
```

Figure 5.1: Implementation domains

The XBarGenerator is not described deeply, since it is a helpful application with no significance for the crossbar switch implementation, which is the main focus. In essence, the design parameters entered by the user are used to customize template files containing the basic source code. The remaining domains are described in the following sections. Finally, a test environment demonstrating the correct operation is presented.

5.1 XDL Hard Macro

The switching structure described in this section is the most complex part of the implementation. The fundamental idea was introduced in section 4.3.3 and is recapitulated shortly in the following. The switching structure provides the possibility to connect a bundle of vertical input wires with different bundles of horizontal output wires. A so-called cross-point may exist at any location at which horizontal and vertical bundles intersect. Each cross-point can have two states: either the intersecting wires are connected or they are left unconnected. One cross-point is implemented within one CLB as described in 5.1.1.

The complete structure forms a regular mesh of CLBs and can be moved to any FPGA location because of the uniform occurrence of CLBs. However, the crossbar switch as shown in Figure 4.3 differs from the initial situation as depicted in Figure 4.6, since certain cross-points are removed. In section 5.1.2 it is explained how the reduced cross-point allocation can be implemented in a very dense manner.

Finally, section 5.1.3 describes how the cross-point states are triggered and the remaining I/O for the hard macro, the *external pins*, are introduced. This involves another optimization affecting the hard macro layout. Afterwards the final structure with all modifications is presented.

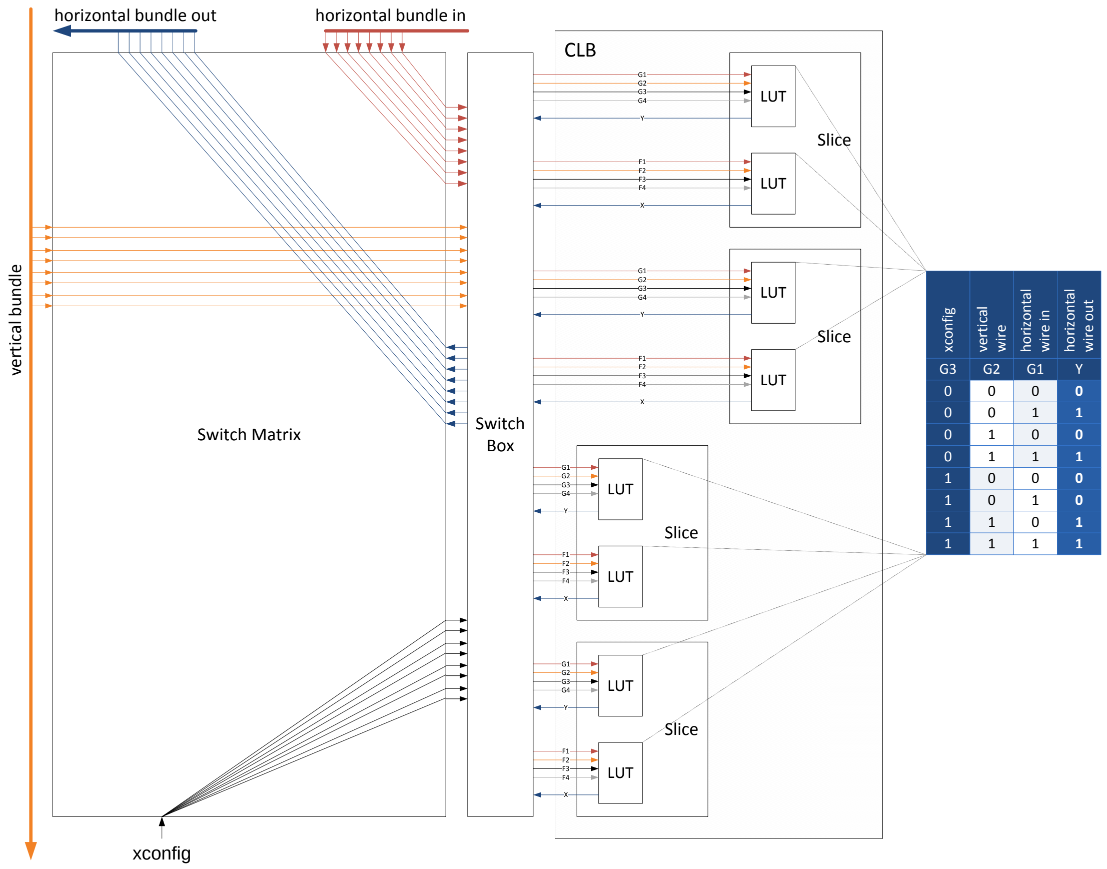
5.1.1 Cross-Points

As mentioned before, one cross-point can be implemented within a single CLB. A CLB can be further divided into four similar elements called *slices*. In turn, each slice contains two LUTs realizing any 4-input logic function, two flip-flops which can, among other things, be used to synchronize the LUT output, and several multiplexers to choose between different outputs or to enable features like input inverters, carry chains etc. Additionally, one switch matrix and one switch box are assigned to each CLB. The most important components for this design are illustrated in Figure 5.2.

All together there are 8 LUTs available, each of them provides one output. This is the reason for having 8 wires per bundle. It is not possible to implement a reliable and fast cross-point with a larger number of wires within one CLB as long as the switch matrix or switch box is considered to be fixed. This applies to the current situation, since partial reconfiguration is the only way to change these settings which is rejected for the reasons given earlier. Nevertheless, there is some other logic apart from the LUTs available within the CLB, which could be used for integration of additional wires. For example, some multiplexers would be appropriate. However, the behavior would be hardly predictable, since if other resources than LUTs were used, the switching characteristic for the distinct wires would differ: while LUT input-to-output delay (T_{ILO}) is always 0.2 ns, the fastest multiplexer that could be used

(F5MUX) has a delay (T_{IF5}) of 0.46 ns on the Virtex-4 device used in this work (Virtex-4 VLX160 Speed Grade -10) [25].

The input assignment of each LUT, the LUT content and switch matrix configuration is depicted in Figure 5.2. Each LUT is configured to realize a 2:1 multiplexer. One wire of the horizontal bundle and one wire of the vertical bundle are connected to inputs of each LUT. Additionally, one net distributing the state of the cross-point (*xconfig*) is connected as selector to the third LUT input. The fourth input remains open. According to the selector value, either the data on the horizontal wire or the vertical wire is forwarded.



xconfig	vertical wire	horizontal wire in	horizontal wire out
G3	G2	G1	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 5.2: Cross-Point implementation

Data transferred in a vertical direction is bypassed along the CLBs. Long wires may be used to distribute data to destinations further down the structure. By contrast, data transferred in a horizontal direction has to traverse all CLBs in between, and is delayed by T_{ILO} every time. Thus it is important to keep the number of hops in the horizontal direction low.

5.1.2 Maintaining Density

Maintaining the density of the switching structure is important for two reasons. First, the utilized FPGA area should be small to reduce overhead. Second, a smaller structure comes with shorter wires and propagation delays are reduced. Therefore, higher clock speeds can be achieved and throughput is improved. The basic structure is very dense in the case of a fully occupied crossbar switch (fully occupied means that there is a cross-point at every intersection of wires), but this is not the case in the current design. However, by interleaving the vertical links it is still possible to utilize all CLBs within a rectangular region for the switching structure. Figure 5.3 shows how this is possible for the cross-points of a design having 2 PRMs with 3 input and 3 output ports each (this is according to the setup in Figure 4.3). In principle the structure is just compacted horizontally, so that there is only one CLB column per PRM, while the number of CLB rows still matches the number of overall output ports.

As explained in the previous section, the number of CLB columns has a strong impact on the propagation delay. Because of this, it is expected that an increase of the number of modules will result in significantly lower clock speeds. This is confirmed in section 6.1.2 and a potential countermeasure is presented in chapter 7.

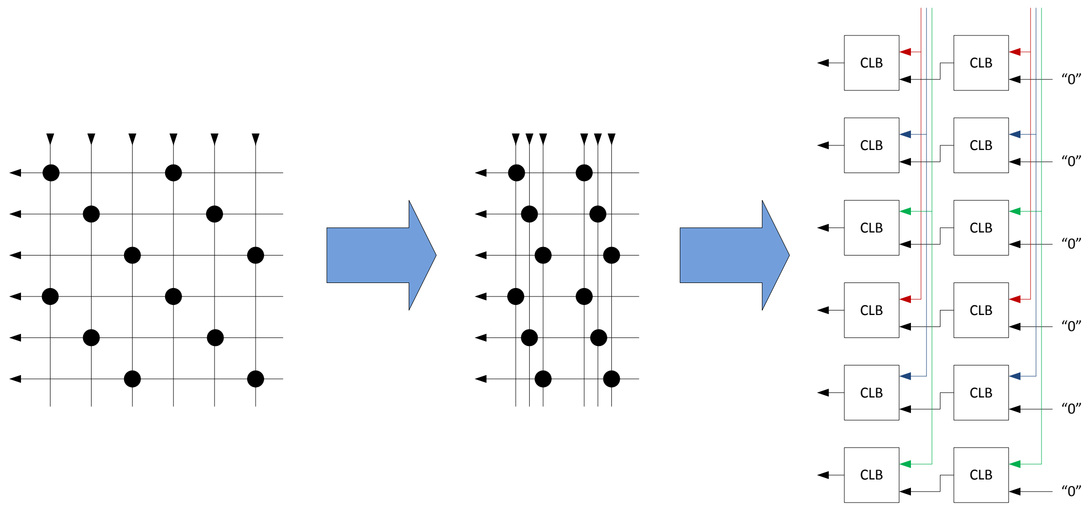


Figure 5.3: Interleaving vertical bundles

5.1.3 I/O, timing and final structure

In order to instantiate the hard macro, an interface must be defined. This is done by adding so-called *external pins*. They are similar to I/O pins of a chip fabric and they must be used for any I/O communication. External pins can be added to any component of the hard macro, for example the slice input associated with a LUT can be used. No restrictions regarding the location exist, though it is possible to add an external pin to any CLB in the switching structure. Nevertheless, external pins are kept at the border of the hard macro and separate CLBs are used for this purpose. The reason is that all cross-points are asynchronous components, which is required to propagate data with low latency. But at inputs and outputs synchronization is required, in order to ensure correct operation, since only data associated with a specific clock cycle may be transferred at a time.

Furthermore, it must be assured that data reaches the output within one clock cycle. Usually, correct timing behavior is checked automatically during the place & route step. However, this is not possible, because a hard macro is treated as a black-box module by the Xilinx timing tools and correct timing must be manually verified. Therefore, the propagation delay of the path through the switching structure from the input CLB to the corresponding output CLB must be examined. This propagation delay is given by

$$d_{propagation} = d_{vertical} + \lambda \cdot (d_{horizontal} + T_{ILO})$$

whereby $d_{horizontal}$ is the wire delay for the horizontal wire, $d_{vertical}$ is the wire delay for one vertical section between two cross-points, T_{ILO} is the LUT input-to-output delay and λ refers to the number of CLBs which have be traversed in vertical direction and corresponds to the number of modules in the worst case. The worst-case value for $d_{horizontal}$ and $d_{vertical}$ can be specified in XBarGenerator to ensure correct operation for a certain clock speed. A *physical constraint file* (PCF) based on these values is generated, which contains the appropriate constraint for each net and has to be merged with the PCF of the static design. The PAR tool will attempt to meet these constraints and throws a warning otherwise. However, this means that estimating the maximum clock speed for a specific design is complicated and it is not possible to provide exact values.

The output CLBs are placed next to the left most cross-point CLB column. Accordingly, all input CLBs could be placed at the top of the switching structure. However, this is modified by placing half of the inputs on the bottom edge to improve the arrangement of inputs. Because the structure tends to be narrow and high, having inputs only on the top might be a bottleneck. A lot of wires would need to be routed to the same location and input FIFOs could not be distributed evenly around the hard macro. Furthermore, placing the crossbar switch between several modules is eased, since it can be accessed from different sides.

Two additional groups of external pins exist. First, an interface for the *xconfig* state selector, which is used to set the state of each cross-point is needed. It is implemented similarly to the data inputs and placed in the rightmost CLB column. Finally, dummy drivers are needed for each vertical bundle. External pins are used here, but they are tied to logic 0 when the hard macro is instantiated. This forces PAR to connect the corresponding pin with the TIEOFF component residing next to every switch matrix.

Figure 5.4 shows the resulting structure with two input and two output ports for each of the four modules. On the left is the schematic view, each CLB is illustrated by a square as before. The picture on the right is a screenshot of the corresponding hard macro implementation as depicted in the FPGA Editor; CLBs having external pins are highlighted.

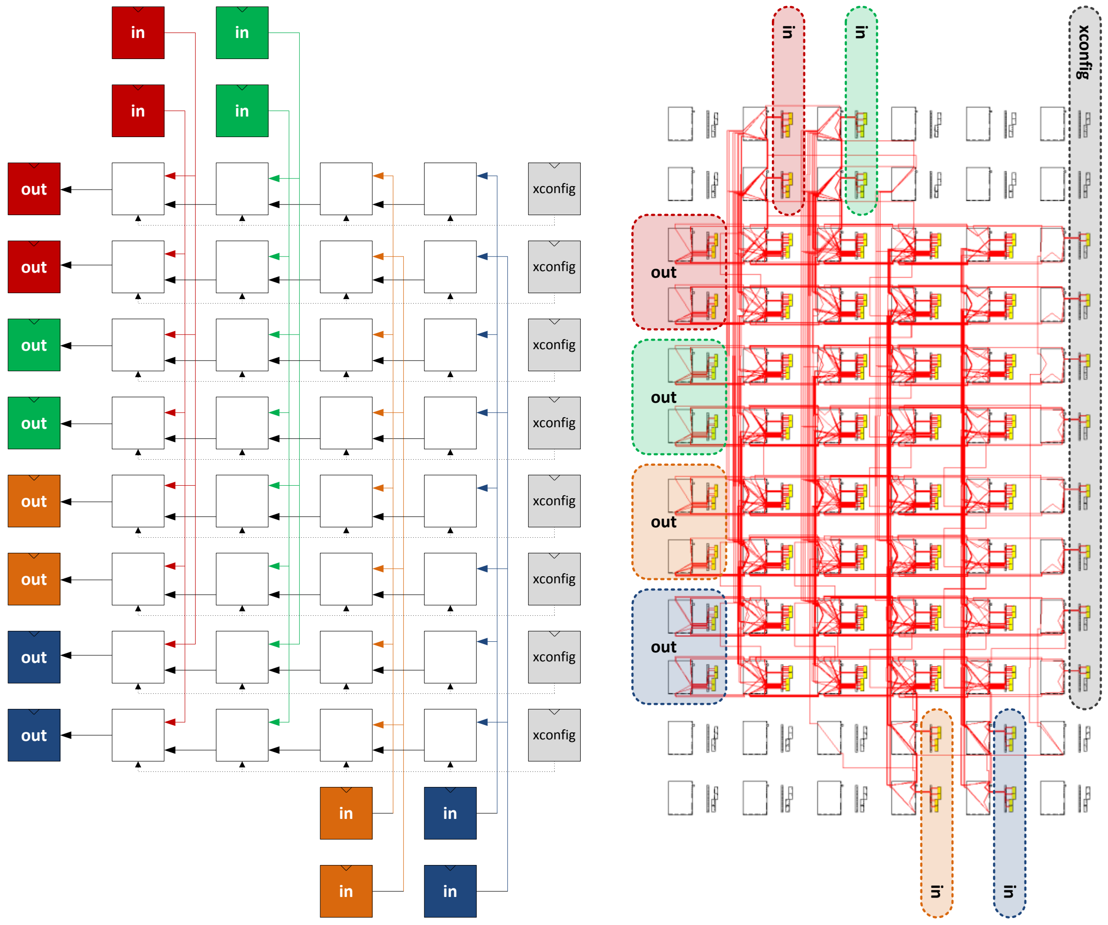


Figure 5.4: Final hard macro implementation

5.2 COREGenerator FIFOs

The implementation of the FIFOs is realized by the COREGenerator. However, the user may choose between different specifications. While most settings like data width can be derived directly from the properties of the remaining design, two parameters require application-dependent modifications and must be defined by the user at design time. First, it must be decided whether the FIFOs should be implemented using Block RAM or Distributed RAM and, second, the number of words stored in each FIFO must be configured. The choice made for the former may depend on the latter: while a large FIFO can be implemented efficiently using Block RAM, the number of slices needed for a similar FIFO using Distributed RAM is significantly higher (see section 6.1.1). On the other hand it may be preferred to keep the Block RAM available for modules. Furthermore, Block RAM is not available at all locations within the FPGA and the best achievable clock speed was found to be slightly higher when Distributed RAM was used – provided that the FIFO depth was very small (compare section 6.1). A larger FIFO (64 entries or more) performs better when implemented using Block RAM.

It is recommended to keep the depth small since large buffers pose several drawbacks and the benefits are rather low. Obviously, the area overhead increases with the size of the buffer. Furthermore, the main purpose of the FIFOs is to connect components of different clock domains. This is already achieved by a very small buffer. A reason for having a larger number of slots within one FIFO is to allow continuous data transmission, even though the target is blocked. But for the case that the target just consumes data more slowly than the source can provide it, a larger buffer does not make any sense: once the FIFO is full, the situation is identical for any depth. In the long term, large buffers are only beneficial if the target is able to catch up with the source again despite intervening interruptions. However, another problem arises when the buffers are large and modules have to be reconfigured: the corresponding buffers need to be emptied before the new configuration can start operation. But this may take longer for large FIFOs, of course.

5.3 VHDL Top-Level Entity

The top-level entity of the crossbar-switch is implemented using conventional VHDL code. All necessary parameters are declared in a VHDL package which is provided by XBarGenerator. Since most of the details were already described in section 4.3, only the precise transfer behavior is described in the following paragraph. Subsequently, a test environment used for verification is presented.

5.3.1 Behavior

Since all ports between a PRR and the top-level entity of XBar are equivalent to the pinout of the respective FIFO, the XBar transfer behavior as shown in Figure 5.5 correlates with the

FIFO read/write behavior described in [8]. The upper part of the diagram refers to an input FIFO of the crossbar switch; the lower part describes the behavior of the corresponding output FIFO. Both FIFOs are connected with two clock domains, one on the reader's side and one on the writer's side. The writer clock domain of the input FIFO and the reader clock domain of the output FIFO correspond to the clock domains of the connected modules. The background of the corresponding signals is shaded in the diagram. The remaining signals of both FIFOs are synchronous to the XBar clock. So altogether there are three clock domains: source module, destination module and control logic. The initial state of the source FIFO is *empty* while the destination FIFO is filled with 4 entries. The FIFO depth is assumed to be 9. It has to be considered that this value was chosen to keep the diagram small and is not a valid parameter. The *full* signal is always advanced by 5 cycles, so the output FIFO signals *full* to the control logic in the beginning.

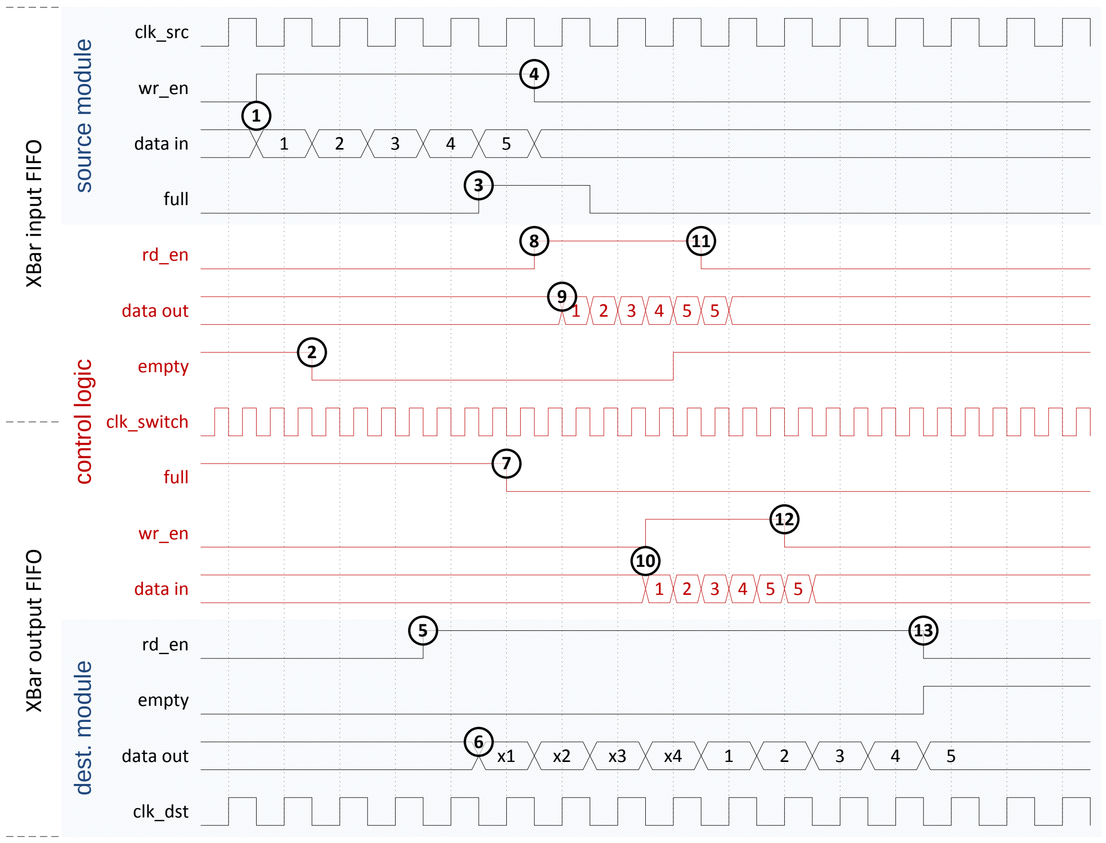


Figure 5.5: XBar transfer behavior

In this example, first *wr_en* is asserted by the source module and data is applied synchronously (①). Every clock cycle one entry can be stored in the FIFO. Additionally, in the cycle following the assertion of *wr_en*, *empty* of the input FIFO is deasserted (②), since the FIFO contains one entry then. Subsequently, after the fourth entry is written, the *full* signal of the FIFO is set (③). Since the source module cannot react before one additional cycle, one more entry might be written and *wr_en* is deasserted thereafter at ④. Supposing the destination module asserts *rd_en* at ⑤, the data stored in the output FIFO will be made available after the next rising clock edge of *clk_dst* (⑥). As a result, *full* of the output FIFO is deasserted after the following rising clock edge of *clk_switch* (⑦). Now it is possible to start the actual transfer and *rd_en* of the input FIFO is asserted by the control logic (⑧). Subsequently, the FIFO entries will become available at the output of the source FIFO (⑨), but it will take 3 additional cycles to traverse the switching structure before the destination FIFO is reached at ⑩. Starting transfer also results in resetting the *full* signal of the input FIFO, which was asserted at ③.

A signal to presence valid data is multiplexed with the payload in the switching structure (see section 4.3.4) and controls *wr_en* of the output FIFO. However, *rd_en* of the source will be deasserted too late (⑪), since one cycle is necessary to react on the corresponding assertion of empty. So precautions must be taken for the *data present* signal, in order to deassert *wr_en* on time such as at ⑫. Contrary to the solution presented in 4.3.4, FIFOs could be configured to provide a *valid* signal for data that could be used for this purpose. This would be beneficial at the output of the crossbar switch, too: the deassertion of *rd_en* as performed by the destination module at ⑬ will be possible only by coincidence. Instead, the empty signal of the previous cycle must be evaluated to ensure that the current entry is valid. On the other hand, the interface is smaller using the current approach. This was preferred in order to reduce the number of bus macros needed between PRRs and static design.

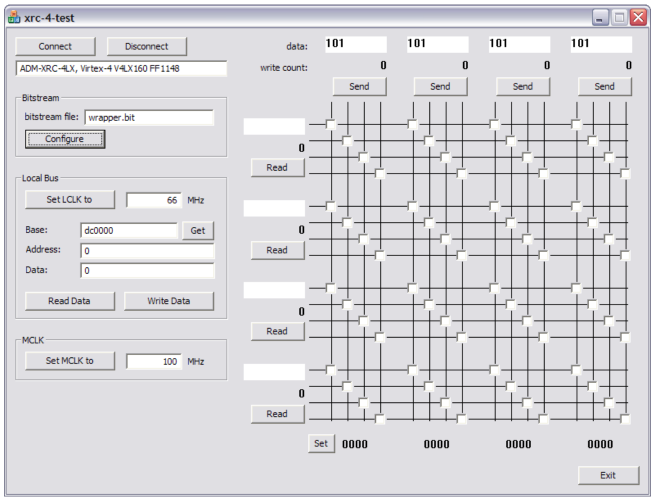
As stated previously, the full signal is asserted 5 cycles in advance. This is required to ensure that all data that was already read from the input FIFO but has not reached the destination yet can still be stored. Currently, the same IP core is used for FIFOs both at the output and the input of XBar, so some space might be wasted at the input side, since this FIFO signals full before it actually is in order to avoid overflow at the output side. However, there is no reason for this limitation and allowing a different threshold value for the *full* signal at the source is easy to implement. On the other hand, modules will benefit from advanced *full* notifications, too, and the impact of wasting a few slots is rather low.

Data is always sent as soon as possible in order to keep the FIFOs at the inputs free and at the outputs filled. However, it is not always possible to forward data, even though the target buffer is ready: in the case of more than one addressed port (multicast), the slowest target determines the maximum transfer speed. If at least one of the destinations is blocked, all transfer from the associated source has to be stopped by the control logic. If no destination is configured, data is still accepted by XBar and remains in the input FIFO until the cross-point configuration is updated and a target exists.

5.3.2 Exemplary Implementation

To verify the behavior, a test environment was implemented. It comprises a wrapper module containing XBar and a control application running on the host machine. The wrapper module is implemented in VHDL and corresponds to the static part of a PR design. It provides access to the XBar by utilizing the local bus of the XRC-4 development board, which is connected to the PCI bus of the host. The control application is a MFC based Windows application which allows the state of any cross-point to be selected in a user-friendly GUI (see Figure 5.6). Data can be read or written from or to any port of the crossbar switch. Furthermore, different clock speeds can be configured for the XBar. Nevertheless, performance cannot be evaluated using this setup, since the local bus has a maximum clock speed of 66 MHz and provides only 32 bits data width which is not enough to saturate the crossbar switch.

The floorplan of the associated FPGA implementation is shown in Figure 5.7. The depicted crossbar switch provides 4 interfaces with 4 input and 4 output ports each, and thereby provides channels of up to 28 bits per module. For behavioral verification, FIFOs are implemented using distributed RAM and the depth is configured to be 16. Wires of the switching structure are colored red, FIFOs at XBar inputs are highlighted yellow, FIFOs at XBar outputs are green and the control logic is white. Cyan wires belong to the wrapper module.


xrc-4-test

ADM-XRC-4LX, Virtex-4 V4LX160 FF1148

Bitstream
bitstream file: wrapper.bit

Local Bus
Set LCLK to 66 MHz
Base: dc0000
Address: 0
Data: 0

MCLK
Set MCLK to 100 MHz

data: 101 101 101 101
write count: 0 0 0 0

0
0
0
0

0000 0000 0000 0000

Figure 5.6: Test application on host machine

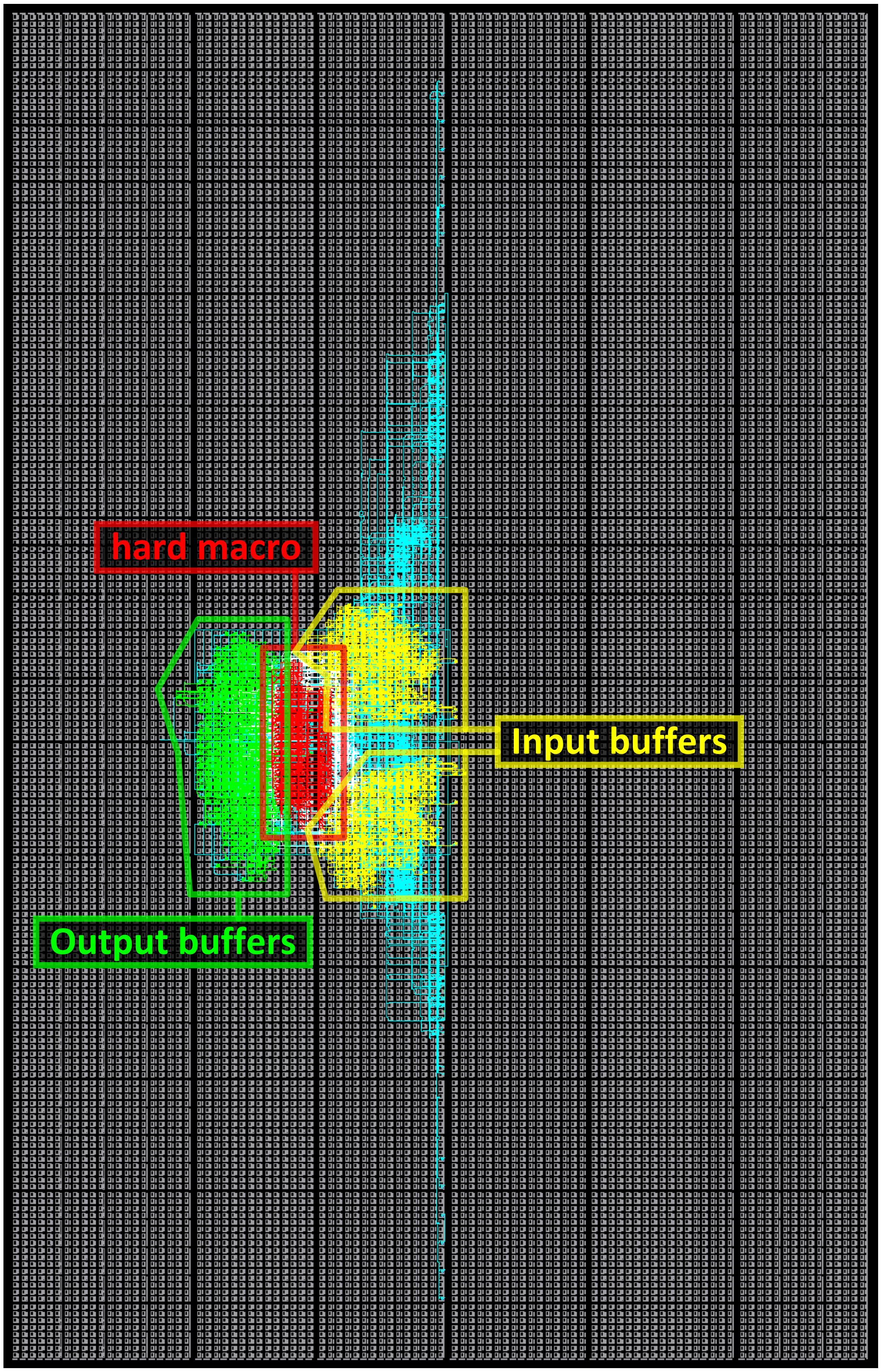


Figure 5.7: Test environment on Virtex-4 VLX160

6 Evaluation

In this chapter the proposed crossbar switch architecture is evaluated and compared to other on-chip communication infrastructures. In section 6.1 several designs for different parameters were implemented to determine the area utilization and performance. The resulting data is used to estimate scaling of the architecture. Subsequently, set-up time for a cross-point configuration is investigated. Afterwards, the amount of flexibility is evaluated and the architecture is compared to other approaches. Finally, possible applications of this architecture are discussed. This also includes reasons why this crossbar switch is not appropriate for certain applications and which features need to be added to expand the application domain.

6.1 Impact of design parameters

There are 4 major design parameters, which are explored in the following: 1. number of modules, 2. interface width, 3. FIFO implementation and 4. FIFO depth. Basically, no test case showed unexpected results. These are presented in two separate sections for area utilization and performance. In section 6.1.3 the results are compared with reference values of the related work.

6.1.1 Area

Calculating area utilization is delegated to the Xilinx tools and thus relatively straightforward. Figure 6.1 shows the number of slices needed by the XBar design for a varying per-module interface width. For a better comparison both 2-module and 4-module designs were implemented and the logic for all components (FIFOs, control logic, switching structure) is included in the figures. The FIFO depth was kept constant at 16 and Block RAM was used for their implementation. The graph shows the linear scaling which is expected. For every additional port, two additional FIFOs per module must be provided. And because several cross-points were removed from the basic crossbar switch layout, the switching structure (hard macro) is also growing linearly with an increasing number of ports. This is in contrast to the result of increasing the number of modules. In this case the area for the hard macro grows quadratically. This explains the slightly different course of the line in Figure 6.2, which is not linear anymore.

In order to evaluate FIFO resource usage, a crossbar switch providing 28 bit wide interfaces to 4 modules was implemented using both Block RAM and Distributed RAM. In the case of Block RAM, only few additional resources are needed to provide a larger FIFO. However, for each FIFO one Block RAM is used (on a Virtex-4 up to 1024 FIFO slots can be realized using one Block RAM). So in the case of this example 32 Block RAMs are required for the switch (4 modules × 8 ports). In the case of Distributed RAM a massive amount of FPGA logic is required to provide enough memory. Obviously, Distributed RAM is only an appropriate solution for very small buffers.

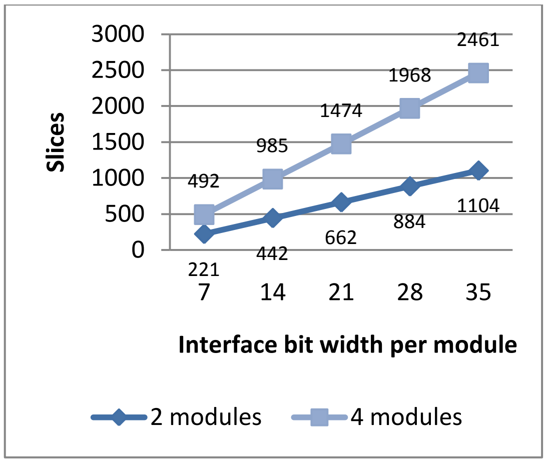


Figure 6.1: Used slices for varying interface width

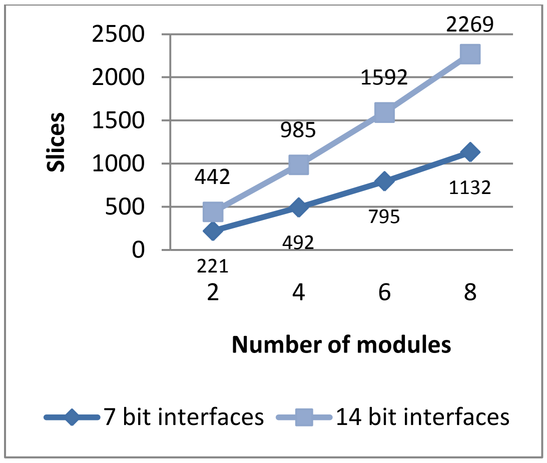


Figure 6.2: Used slices for varying number of modules

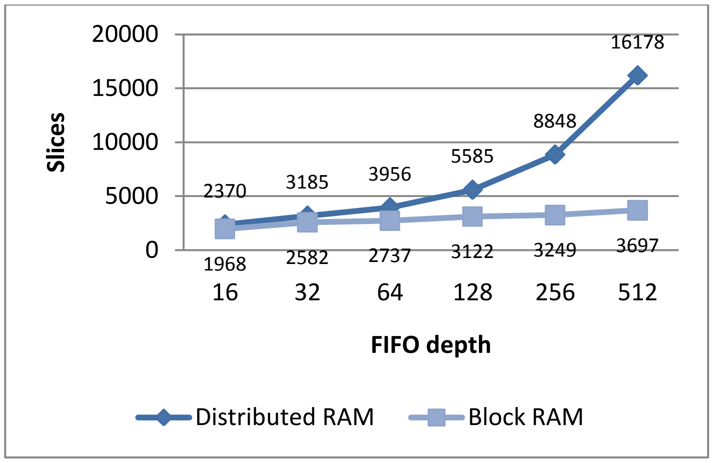


Figure 6.3: Used slices for varying FIFO depth

6.1.2 Performance

Performance is a rather unclear term and may refer to several different aspects. In this context, we are mainly concerned with latency and throughput. Of course, these metrics are usually antagonistic. As stated in 2.4.3, latency normally suffers as throughput is increased and vice versa. Since the latency within the hard macro is always 1 clock cycle, the latency of the crossbar switch is constant as well: four clock cycles are required to transfer data from the input buffer to the output buffer: 1. rd_en is asserted at the input FIFO, 2. the entry becomes available and is forwarded to the input register of the hard macro, 3. the hard macro is traversed, 4. the entry is written to the output FIFO. However, from the module perspective, the latency is higher: 2 more cycles (but of different clock domains) for writing/reading the FIFOs are required. Additionally, data may accumulate in these buffers and the latency for messages will increase further in this case. Contrary to latency, throughput is dependent on the design parameters and can be expressed by simple formulae, e.g. the maximum aggregated throughput is:

$$throughput_{max,agg} = width_{interface} \cdot \#_{modules} \cdot f_{max}$$

whereby $width_{interface}$ and $\#_{modules}$ are design parameters and only the maximal achievable frequency f_{max} remains to be estimated.

However, this is not as straightforward as identifying area utilization, since readings cannot simply be taken from the synthesis tools. The biggest problem concerning clock estimation arises when routing the hard macro. Timing constraints must be defined manually for each net; specifying a maximum input-to-output delay is not possible. Thereby, the place and route tools are prevented from exploring the whole design space. In addition, applying very tight constraints leads to much worse results than actually possible. Furthermore, nets of the remaining design narrow the scope for alternatives within the hard macro routing. The routing paths available to the switching structure are thus dependent on the remaining implementation and cannot be easily calculated. Finally, the specific target device and its speed grade have to be considered. All in all, finding correct values for net constraints is quite difficult.

Precise values for some different configurations can be found in Table 6.1. For the reasons given previously, better results may be possible, but degradation may also result if the XBar is embedded into a more complex design. As may be expected, the maximum clock speed that can be achieved drops as the interface width or the number of modules increases. It shows that the impact of the interface width is not as high as the impact of number of modules. This can be followed easily by looking at the switching structure: its width is related to the number of modules, but it is independent of the interface width. Thus, propagation delays increase more significantly as the number of modules increases. This can also be derived directly from the formula in section 5.1.3, where λ is equal to the number of modules.

For only two modules the critical path is not within the switching structure. However, this is no longer true as soon as the number of modules increases. It is possible to reduce this critical path by introducing pipeline stages within the switching structure as suggested in chapter 7. But doing so has a bad effect on latency.

Parameter	Max. frequency	Aggregated throughput
2 modules, 7 bit	400 MHz [7]	5.6 Gbit/sec
2 modules, 28 bit	345 MHz	19.32 Gbit/sec
2 modules, 56 bit	323 MHz	36.18 Gbit/sec
4 modules, 7 bit	222 MHz	6.22 Gbit/sec
4 modules, 28 bit	182 MHz	20.38 Gbit/sec
4 modules, 56 bit	161 MHz	36.06 Gbit/sec

Table 6.1: Clock speed and throughput for different designs

Another insight regarding maximum frequency involves the implementation methodology of the FIFOs. Naturally, higher clock speeds can be achieved for smaller buffers. However, while Block RAM shows much better performance for buffer sizes above 64 slots, smaller buffers run faster when using Distributed RAM. This is because Block RAM cannot be accessed as easily depending on their location on the FPGA and long wires may have to be introduced. But depending on the top-level layout of the PR design, this drawback may be insignificant, since PRRs will probably not abut the crossbar switch, anyway. The remaining design parameters for the FIFOs, which are not mentioned in the table above, were Distributed RAM for implementation and 16 slots for the depth.

6.1.3 Comparison

Since a crossbar switch is a rather heavy structure providing a large number of parallel wires, it might be expected that area overhead will be a big problem. Fortunately, this is not the case. Of course, compared to a bus, more resources are used by the communication infrastructure, but the introduced NoCs show worse results for settings as in the scope of this work. For example, when four CoNoChi routers are instantiated, which is the minimum to connect four modules, 3072 slices are used. A corresponding XBar configuration only uses 1968 slices and provides a significantly higher throughput and lower latency at the same time. Compared to this, buses perform less well: 6.4 Gbit/sec can be achieved by the ReCoBus architecture, but has to be shared by all modules. Similar data rates can be achieved between all modules connected to the crossbar switch in parallel. However, comparing pre-

[7] limited by the maximum frequency for this FPGA device

cise values is difficult here, since the architectures were implemented on different devices. Compared to the Xilinx Crossbar Switch, area utilization for the switching structure is high. As mentioned before it is a great advantage to use routing resources to implement the switching structure. However, only the switching structure could benefit from this approach. Since most logic is already needed for buffers and control logic, this improvement carries less weight for the current design. But it would be feasible to include all cross-points again, and much higher switching flexibility could be provided.

6.2 Reconfiguration and switching performance

Updating the cross-point configuration is very fast. Just two cycles after writing the configuration register, the new setting becomes valid. This is an advantage compared to the Xilinx crossbar switch solution, which uses partial reconfiguration capabilities of FPGAs to update switch settings. As mentioned in section 4.1, it is possible to provide a configuration interface to each module, which would allow establishing new channels with high frequencies at runtime. However, before a new configuration is applied, all buffers affected by the update should be empty, since otherwise data sent earlier might be delivered to the wrong destination.

6.3 Scaling and Flexibility

Crossbar scaling is a critical issue, since the number of cross-points and wires grows as the square of the numbers of inputs and outputs. However, this aspect is less problematic in this architecture, since critical resources were reduced by removing several cross-points. But it is still not possible to use this structure as interconnection for a large number of components. Even if additional improvements like pipeline stages were introduced, the main reason, why communication between hundreds of modules is not possible, remains: the centralized structure requires all components to be arranged around a single communication block. Components must be placed farther and farther away from this central point, but all data has to be sent through it. Obviously, this becomes inefficient. Nevertheless, compared to a bus, scaling is better. There is no critical point at which saturation is reached, since additional bandwidth is provided together with each additional module. Both for buses and the structure proposed in this work, scaling could be improved by introducing hierarchies: several independent instances of the same interconnection structure could be combined. This yields to architectures with hybrid properties, but reduces flexibility as explained in section 2.4.4.

Packet switched networks provide several routers operating independently and thereby do not suffer such scaling problems. Nonetheless, another issue arises and limits might be reached quickly as well: latency is the problem in this case. Small synchronization messages and handshakes cannot be handled efficiently if delivery takes 20 or more cycles – which may be the case as soon as just a few hops have to be taken. Of course, this problem can be tackled with the same technique as mentioned previously, such as providing a separate low-latency network.

A lot of flexibility is provided by XBar, but it must be said that the potential is not fully exploited. It is possible to establish multiple channels per module, the link width can be adapted at runtime, modules may send and receive data simultaneously, and messages are transferred in parallel once the connections are established, so no blocking occurs. These are great advantages over bus-like approaches, where any communication is limited to the static properties of the architecture. But in order to play out all the strengths of this flexibility, configuration must be accessible by modules and not only by some host controller as it is currently the case. Because such an external controller can consider pre-known communication needs only. If unexpected requirements arise for some module, no efficient mechanism exists to adjust the established channels, although these changes could be done very fast. Providing an appropriate controller within the switch could even be used to simulate packet-switched behavior by frequently changing the destinations. However, this will reintroduce the overhead which should be avoided by limiting to rather fixed channels during the lifetime of one module.

6.4 Field of Use

The proposed communication architecture was designed for inter-module communication. Special requirements of partially reconfigurable systems were taken into account and an appropriate amount of flexibility is provided. Nevertheless, this solution still suits better to data stream processing than concurrently interacting applications, because of the circuit switched approach. High throughput can be provided on several parallel channels which matches requirements of multimedia or network applications perfectly. Furthermore, the structure imposes creation of processing chains, since once a path is set up, data may rush through without further ado.

It might be considered to use the crossbar switch to manage access to peripherals or memory. A PRM can take over the role of an adapter by forwarding data to several XBar ports and may contain an arbiter to manage access to an exclusive resource. But particularly in the case of memory access, the connection flexibility might not be needed, since memory access should be provided to all PRMs. Because of this, it is more reasonable to provide this access separately by a dedicated interface. Another reason is that addresses must be supplied together with data, and the XBar interface is not laid out therefore.

If shared memory has to be implemented, a bus could be used as an additional communication infrastructure. However, higher performance can be achieved if distinct memory cells are accessed simultaneously, as it is possible with the development board used in this work. In order to increase performance, memory cells could be linked directly to a specific PRR, which would prevent memory sharing, or a memory management unit (MMU) can be introduced. The latter case is visualized in Figure 6.4. In this figure, local bus access is managed by the MMU as well. This is also proposed by Alpha Data, the manufacturer of the board used in this work, in order to provide consistent access to external resources [26].

At its current stage, the crossbar switch should be used for predictable communication patterns only. Cross-point configurations must be known in advance and set up before the corresponding component starts operation. Again, this suits the conditions that apply when data streams are processed. Furthermore, error detection is not implemented, so reliability is expected of the modules, as long as no further precautions are taken.

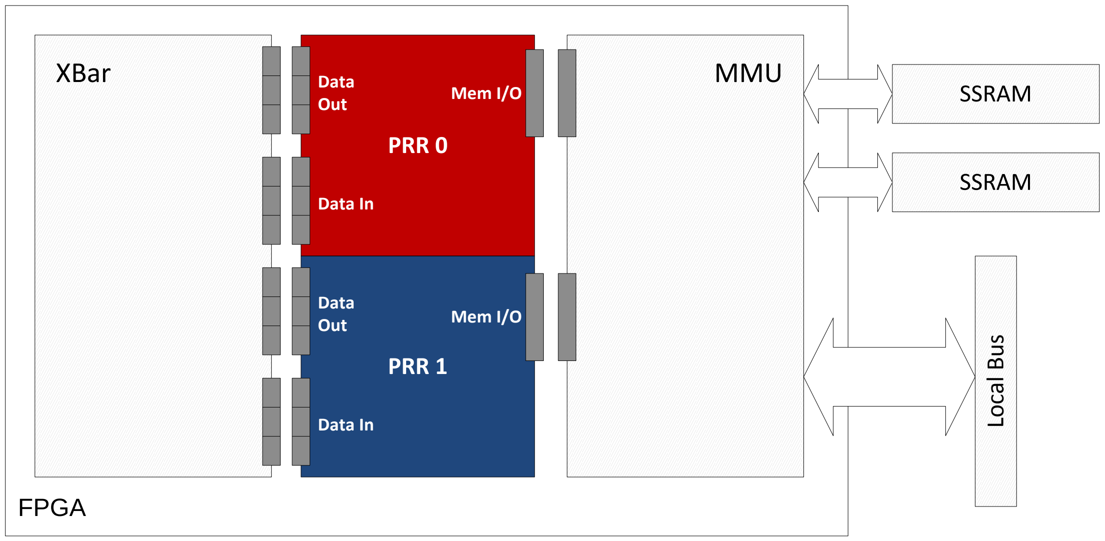


Figure 6.4: PR system with MMU

A simple application scenario involves compression- and encryption-modules, which are connected in series in order to speed-up transfer over a low speed channel that cannot be trusted (e.g. radio waves). The modules could be exchanged independently according to the currently required algorithm. Partial reconfiguration is beneficial in this scenario, since the algorithms might change frequently if the same accelerator is shared for several different connections. Furthermore, adding support for additional algorithms will be possible at any time because all that has to be done is providing the bitstream for a new PRM.

7 Conclusion and Future Work

In this work a crossbar switch architecture for inter module communication in a partially reconfigurable system, called XBar, was introduced. Design and implementation were explained in detail and correct operation of the implementation could be shown. The proposal is suitable for small configurations consisting of up to 8 partially reconfigurable regions. This matches what is actually possible on a single FPGA device using the tools currently available. On these terms high throughput can be achieved on multiple independent channels. Furthermore, a rather high amount of flexibility is provided within the infrastructure compared to bus architectures commonly in use. However, it showed that competing against packet switched approaches is not possible in this field. Neither is it possible to achieve better scaling behavior with the crossbar switch approach than with rather decentralized NoCs. On the other hand, these flexible networks have not yet been shown to be competitive against less complex structures such as buses regarding performance, since they introduce high overhead which reduces throughput and raises latency. And it is still uncertain which amount of flexibility is really needed. Furthermore, scaling is less important when limiting to smaller systems and last but not least, interfacing packet switched networks is much more complicated.

Some questions were raised during this work and several issues remain to be solved. First it might be asked whether the restriction to the very symmetric design is useful. While the interfaces of any pair of PRRs must remain equal to allow arbitrary placement of modules, having different input and output bit widths is no violation of this design requirement. As a result, additional flexibility could be introduced at design time and only minor changes on the current implementation would be necessary for realization.

A big problem is the performance drop for a higher number of modules. Pipeline stages are the most straightforward approach to tackle this problem. Implementing pipeline stages would be very easy: all that has to be done is placing additional registers on the critical path within the switching structure as depicted in Figure 7.1.

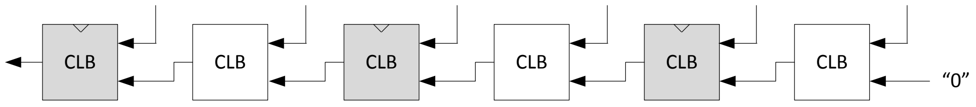


Figure 7.1: One CLB row of the switching structure with pipeline stages

In all shaded CLBs the synchronized slice output would be used for the horizontal nets instead of the asynchronous one. In this case, data is not forwarded along the path before the next clock cycle. The maximal propagation delay between two of such *pipeline CLBs* would be much smaller than the propagation delay for the whole path and remarkably higher frequencies could be achieved. But latency would differ for varying paths through the switching structure, which leads to both more complex control logic and more complicated module design. Another modification on the switching structure that might be considered is reducing the density by having "free" CLB columns in between to provide more wire candidates for vertical nets. Furthermore, better estimations for propagation delays would be favored in this context. Currently only worst-case values can be assumed for all segments of the path. As a result, the estimated propagation delay is much higher than it actually is.

As already stated before, the field of use for the XBar could be extended by introducing a more powerful internal controller. If an appropriate interface would be provided, modules could manage the cross-point configuration on their own. This would ease the design of self-organized and fault tolerant systems. Furthermore, fewer resources are wasted, since channels that are rarely needed may be destroyed immediately after use without notifying the host controller. So control overhead would be reduced in this example as well. However, the area overhead which is necessary to provide such functionality will increase.

Another issue arises when the top-level design for the partially reconfigurable system has to be implemented. Following the EA PR tool flow for a PR design, *bus macros* are needed as interface for every partially reconfigurable region. Since the standard *bus macros* provided by Xilinx are rather inflexible and must be instantiated with care, automatically generated *bus macros* corresponding to the XBar interface would speed up the system implementation. The XBarGenerator application could be extended easily for this purpose.

A. Source Code

Since several thousand lines of code were written for this work, it is not possible to include them in this document. All additional sources created during this work can be accessed via the web: *http://www.stud.uni-karlsruhe.de/~upbof*. The directory index of the filed archive is shown below.

Directory	Description
/bin	Compiled applications
/XBarGenerator	Windows XBarGenerator application, can be used to create all HDL sources for different parameters
/xrc-4-test	Windows GUI to interact with the ADM XRC-4 board and can be used to test the XBar behavior
/bit	Bitstreams for ADM XRC-4 board with VLX160 FPGA
/src	All source code
/cpp	C++ sources
/XBarGenerator	Visual Studio 2008 project: XBarGenerator application
/xrc-4-test	Visual Studio 2008 project: xrc-4-test application
/hdl	HDL sources
/xrc-4-test	ISE 9.1i_PR14 project: test environment corresponding to xrc-4-test application

Acronyms

CLB	Configurable Logic Block
CoNoChi	Configurable Network-on-Chip
EA PR	Early Access Partial Reconfiguration
ESM	Erlangen Slot Machine
FIFO	First In, First Out (memory)
FPGA	Field programmable Gate Array
GUI	Graphical User Interface
IP	Intellectual Property
ISE	Integrated Software Environment
LUT	Look-Up Table
MFC	Microsoft Foundation Classes
MMU	Memory Management Unit
NoC	Network-on-Chip
OS4RS	Operating System for Reconfigurable Systems
PAR	Place & Route
PIP	Programmable Interconnect Point
PLD	Programmable Logic Device
PR design	partially reconfigurable design
PR system	partially reconfigurable system
PRM	Partially Reconfigurable Module
PRR	Partially Reconfigurable Region
QoS	Quality of Service
RMB	Reconfigurable Multiple Bus
SoC	System-on-Chip
SSRAM	Synchronous Static Random Access Memory
VHDL	VHSIC HDL, Very High Speed Integrated Circuit Hardware Description Language
XDL	Xilinx Design Language
XUP	Xilinx University Program

Bibliography

[1] Xilinx, Inc., "Benefits of Partial Reconfiguration," *Xcell Journal*, 2005.

[2] Xilinx, Inc., "Difference-Based Partial Reconfiguration," *Xilinx Application Note XAPP290*, December 2007.

[3] Alpha Data, "ADM-XRC-4 Datasheet," April 2009.

[4] Xilinx, Inc., "Virtex-4 family overview," *Xilinx Datasheet DS112*, 2004.

[5] Xilinx, Inc., "Virtex-5 family overview: LX, LXT and SXT platforms," *Xilinx Datasheet DS100*, 2007.

[6] Xilinx, Inc., "Virtex-6 family overview: LX, LXT and SXT platforms," *Xilinx Datasheet DS150*, 2009.

[7] B. Zhang, M. Huebner, C. Schmutzler, J. Becker, W. Stechele, C. Claus, "An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems," in *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, Porto Alegre, Brazil, 2007.

[8] Xilinx, Inc., "LogiCORE™ FIFO Generator v4.2," *Xilinx User Guide UG175*, October 2007.

[9] Xilinx, Inc. (2000, July) Xilinx Design Language. [Online]. *xilinx6.1/help/data/xdl/xdl.html*

[10] A. Kulmala, T.D. Hämäläinen, E. Salminen, "Survey of Network-on-chip," in *OCP White Paper*, 2008.

[11] J. Teich, A. Ahmadinia, C. Bobda, M. Majer, "The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-Based Computer," in *Journal of VLSI Signal Processing Systems*, 2007, pp. 15-31.

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

[13] A.K. Somani, H. Schröder, H. Schmeck, A. Spray, H.A. ElGindy, "RMB - a reconfigurable multiple bus network," in *Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, San Jose, 2003, pp. 108-117.

[14] C. Haubelt, J. Teich, D. Koch, "Efficient Reconfigurable On-Chip Buses for FPGAs," in *16th International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 287-290.

[15] A. Bartic, D. Verkest, S. Vernalde, R.Lauwereins, T. Marescaux, "Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs," in *Field-Programmable Logic and Applications*, 2002, pp. 741-763.

[16] J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, R. Lauwereins, T.A. Bartic, "Highly scalable network on chip for reconfigurable systems," in *International Symposium on System-on-Chip*, 2003, pp. 79-82.

[17] R. Koch, C. Albrecht, T. Pionteck, "Applying Partial Reconfiguration to Networks-on-Chip," in *Field Programmable Logic and Applications*, Madrid, 2006, pp. 155-160.

[18] P. Alfke, C. Fewer, S. McMillan, B. Blodget, D. Levi, S. Young, "A high I/O reconfigurable crossbar switch," in *Field-Programmable Custom Computing Machines*, 2003, pp. 3-10.

[19] The ReCoBus Project Website. [Online]. *http://www.recobus.de*

[20] P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, V. Nollet, "Reconfigurable Architectures Workshop," 2003.

[21] T. Pionteck, C. Albrecht, B. Maehle, R. Koch, "An Adaptive System-on-Chip for Network Applications," in *13th Reconfigurable Architectures Workshop associated with the 20th International Parallel and Distributed Processing Symposium*, 2006.

[22] R. Koch, C. Albrecht, E. Maehle, T. Pionteck, "A Design Technique for Adapting Number and Boundaries of Reconfigurable Modules at Runtime," in *International Journal of Reconfigurable Computing*, 2009.

[23] C. Fewer, "Cross Bar Switch Implemented in FPGA," *Xilinx White Paper WP166*, September 2002.

[24] Xilinx, Inc., "Early Access Partial Reconfiguration User Guide," *Xilinx User Guide UG208*, September 2008.

[25] Xilinx, Inc., "Virtex-4 FPGA Data Sheet," *Xilinx Datasheet DS302*, 2009.

[26] Alpha Data, "ADM-XRC SDK 4.9.3 User Guide," 2009.