Università degli Studi di Cagliari
Facoltà di Ingegneria
Dipartimento di Ingegneria Elettrica ed Elettronica

Conception, Analysis, Design and Realization of a Multi-socket Network-on-Chip architecture and of the Binary Translation support for a VLIW core targeted to Systems-on-Chip

Ing. Gianni MEREU

Tesi per il conseguimento del Dottorato di Ricerca
in Ingegneria Elettronica ed Informatica

Relatore: Prof. Ing. Luigi RAFFO

Corso di Dottorato di Ricerca in Ingegneria Elettronica ed Informatica
XIX ciclo

Contents

1

List of Figures **2**

1 Presentation of the Thesis **4**

1.1 Introduction 4

1.2 Research effort in the *Computation* field 5

1.3 Research effort in the *Communication* field 5

1.4 Overview of the thesis 6

I SoC communication architectures: *Multi-socket support in STNoC architecture*. **7**

2 Traditional on-chip communication architectures **8**

2.1 State of the art bus-based interconnection systems 8

2.1.1 AMBA Bus 9

2.1.2 IBM CoreConnect 10

2.1.3 OCP Bus 13

2.1.4 AMBA AXI 15

2.1.5 STMicroelectronics STBus 17

3 Network-on-Chip technology overview **20**

3.1 NoC: an unifying concept 20

3.2 NoC basic concepts 22

3.2.1 Network Abstraction 23

3.3 NoC Research 24

3.3.1 Network Adapter 25

3.3.2 Network Level 26

3.3.3 Link Level 34

3.4 STNoC™ Network-on-Chip 35

4 Single socket and multiple socket NoC based SoCs. **37**

4.1 Network Adapter research. 37

4.2 Multiple socket NoC based SoCs. 38

4.3 Standard STNoC™ Network Adapter. 39

5 Multi-socket support in STNoC™ architecture **42**

5.1 Introduction 42

5.2 Comparison between STBus and AMBA AXI protocols 43

5.2.1 Support of the separate *Read* and *Write* channels of AMBA AXI protocol 43

5.2.2 Differences about the meaning of the *Size* of a transaction . . 44

5.2.3 Addressing Modes 45

5.2.4 Choice of the primitive operations to support inside the network. 45

5.2.5 Proposed format of the Transport Layer Opcode of the STNoC™ packet header. 46

5.2.6 The response signals. 47

5.2.7 Multi-socket STNoc™ packet. 48

5.2.8 Interruptibility of STBus and AXI transactions. 48

5.3 Main hurdles towards compatibility 50

5.3.1 Unaligned Addresses. 50

5.3.2 AXI transaction sizes not directly supported by STBus. 52

5.3.3 ID information. 53
5.3.4 Complexity of the AMBA AXI-STBus mapping problem. . . . 55
5.4 Mapping algorithms for AMBA AXI and STBus traffic integration. . 58
5.4.1 AXI-to-STBus traffic. 58
5.4.2 AXI-to-AXI traffic. 62
5.4.3 STBus-to-AXI and STBus-to-STBus traffic. 63
5.5 Implementation, results and future work. 65

II SoC computation architectures: *Extension of ST*230 *architecture to binary compatibility with ARM ISA.* **66**

6 Computation paradigms for Systems-on-Chip. 67
6.1 From the Embedded and General Purpose computing paradigms to the single System-on-Chip computing domain 67
6.2 System-on-Chip computing models. 68
6.3 Advanced processor architectures: Superscalar and VLIW. 70
6.4 VLIW processors. 73
6.4.1 Code generation for *VLIW* processors. 73
6.4.2 Introduction to binary compatibility. 76
6.4.3 Analysis of VLIW microarchitecture. 78
6.4.4 Compression of VLIW programs in memory. 80

7 ST230 architecture. 84
7.1 Fetch Packet. 85
7.2 Software load speculation. 86
7.3 Management of the Program Counter. 86
7.4 Lx Architecture and binary compatibility with ISA ARM. 86

8 Hardware and software techniques for binary translation of Instruction Sets. 88

8.1 Overview of binary translation techniques. 88

8.2 Supporting binary compatibility in hardware. 93

8.2.1 Hardware Binary Translation in superscalar processors. 94

8.2.2 Trace Cache. 97

8.2.3 Microcode sequencers. 98

8.3 Software-oriented approaches for binary compatibility. 100

8.3.1 DEC FX!32. 103

8.3.2 Examples of binary translation techniques in the context of VLIW processors. 104

9 Extension of ST230 architecture to binary compatibility with ARM ISA. 109

9.1 Analysis of architectural choices. 109

9.2 Implementation. 114

9.2.1 Examples of ARM ISA description with the Chorizo grammar. 115

9.3 Results and future work. 119

Appendix 121

A Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms. 121

Bibliography 137

List of Figures

2.1 Typical AMBA system. 9
2.2 Multiplexor Interconnection. 11
2.3 CoreConnect vs AMBA 2.0 features. 12
2.4 CoreConnect vs AMBA 2.0 features. 13
2.5 System Showing Wrapped Bus and OCP Instances. 14
2.6 Channel architecture of reads. 17
2.7 Channel architecture of writes. 18
2.8 STBus protocol layers. 19

3.1 Examples of communication structures in SoC. a) bus-based, b) dedicated point-to-point links, c) chip area network. 21
3.2 4-by-4 grid structured NoC. 22
3.3 Flow of data from source to destination across abstraction layers. . . 23
3.4 NoC research area classification. 24
3.5 The Network Adapter. 25
3.6 Typical network topologies. 26
3.7 Irregular network topologies. 27
3.8 Spidergon™ topology. 27
3.9 Generic Router model. LC= link controller. 29
3.10 Store-and-forward packet propagation. 30
3.11 Wormhole packet propagation. 31
3.12 Virtual cut-through packet propagation. 31

3.13 Virtual Channels. 33
3.14 Streams of different VCs can proceed even if another stream sharing some of the links is stalled. 33
3.15 Spidergon™ topology. 35
3.16 High Level view of STNoC™ Network Adapter. 36

4.1 System with single socket protocol. 39
4.2 Ideal system with multiple socket protocols. 39
4.3 Worst case system with multiple socket protocols. 40
4.4 High-level representation of standard STNoC™ Network Adapter. . . 40
4.5 STNoC™ packet layering. 41

5.1 AXI Write and Read channel sequentialization. 44
5.2 Comparison of available addressing mode options. 46
5.3 Interpretation of AXI response opcode. 47
5.4 Interpretation of STBus response opcode. 48
5.5 Proposed mapping between AXI and STBus response opcode values. 48
5.6 Proposed STNoC Response Packet Opcode. 49
5.7 STNoC request packet. 49
5.8 STNoC response packet. 49
5.9 Header at transport layer in request and response packets. 50
5.10 Propagation of packets generated by STBus Masters. 52
5.11 Propagation of packets generated by AXI Masters. 53
5.12 Packet splitting in AXI-to-STBus traffic scenarios. 54
5.13 Possible values for AXI SIZE and LENGTH parameters. 55
5.14 Splitting and reconstruction of incoming packets. 55
5.15 Generation of STNoC™ ID information in case of AXI Masters. . . . 56
5.16 Structure of the SOURCE ID field of STNoC™ request header. 56
5.17 Process of Mapping Selection. 57
5.18 Kernel of AXI-to-STBus wrapped algorithm 61
5.19 Synthesis data. 65

6.1 Computing platforms for Systems-on-Chip. 69
6.2 Logic block diagram of a *Superscalar* processor. 71
6.3 VLIW processor. 73
6.4 Scheduling across different basic blocks. 76
6.5 Selection of a Trace and scheduling of the instructions inside. 77
6.6 Substitution of the Trace with the Schedule and analysis of the situation at the edges. 78
6.7 Generation of the compensation code at the edge of the Trace. 79
6.8 Selection of the next Trace. 80
6.9 Execution *lanes* in a $VLIW$ processor. 81
6.10 Example of centralized architecture. 82
6.11 Example of architecture Clustered. 82
6.12 Format of the Fetch Packet for the TIC6XXX. 82
6.13 Possible configuration for the Fetch Packet of a TIC6XXX. 83
6.14 Schedule that results from the preceding pattern. 83

7.1 High level block diagram of ST230. 85

8.1 Conceptual representation of the operation of an emulator. 89
8.2 CISC to RISC ISA conversion in hardware in modern x86 processors. 91
8.3 Hardware approach to the binary conversion with software support for corner cases. 92
8.4 Binary conversion in the context of VLIW processors. 93
8.5 Front-end of a modern superscalar processor. 94
8.6 Typical parallel decoder. 95
8.7 Detailed block diagram of a parallel decoder. 96
8.8 Binary decoding with Trace Cache. 98
8.9 Micro Sequencer. 99
8.10 Managing control flow inside the microsequencer with branch uops. . 99
8.11 Managing control flow in microsequencer with next uop address field. 100
8.12 Simple two level ROM. 101

8.13 Micro Sequencer with intermediate instruction format. 101
8.14 General view of a software binary translation platform. 102
8.15 Block diagram of DEC FX!32. 104
8.16 Daisy's structure. 105
8.17 VMM control flow diagram. 106
8.18 Trace and treeregion 107

9.1 Mapping options for ARM architectural state. 112
9.2 Binary translation system architecture. 114
9.3 Development flow for emulation routines. 116

Chapter 1

Presentation of the Thesis

1.1 Introduction

The Challenges of today's Systems-on-Chip design complexity call for resorting to the old *divide and conquare* problem solving approach. It is no longer possible to undertake the design task in a monolithic way but there is the need to identify aspects of the design that can be approached and solved independently, without of course compromising the quality and performance of the overall result. There are two basic aspects the drive today's Systems-on-chip technology: 1) *Computation* and 2) *Communication*. Computation regards all the functionality that inside a SoC fulfills the system's requirements in terms of raw processing power. In this category fall components as microprocessor, digital signal processors, ASIPs, multimedia accelerators, etc. Communication deals with all the aspects of a System-on-Chip related to the distribution of the information across the chip. To achieve the best performance it is required to match the best of the two worlds. Availability of high performance processing resources would be worthless if not supported by an equally high performance interconnect system. At the same time an high performance interconnect system needs a great amount of processing power to meet the requirements of today's power hungry software applications. The evolution of IC manufacturing technology, which brings almost endless availability of silicon resources but at the same time brings new problems as the *Deep Submicron Effects* (DSM), is leading SoC design in a direction where *Computation* resource design becomes more and more independent from *Communication* resource design.In the course of this thesis have been investigated advanced solutions for improving technology both in the *Computation* and *Communication* fields. Both the research efforts have been conducted in partnership with the AST (Advanced System Technology) STMicroelectronics research

labs.

1.2 Research effort in the *Computation* field

The investigation conducted in this area has been focused on the extension of the architecture of an industrial VLIW processor, the STMicroelectronics ST230 processor, with the objective of providing binary compatibility among execution of native VLIW code and the execution of ARM code on the same processing core. The problem of binary translation of Instruction Sets, where already much effort has been spent in the field of high-performance general purpose computer architectures, is quite new in the context of Systems-on-Chip. The strong real time constraints that must be respected by processors in the context of SoC based systems require a completely new research effort in understanding which techniques and solutions for binary compatibility are best suited for the new environment. The purpose of the work accomplished in the context of the thesis is therefore to make a contribution in this direction, even with all the restrictions imposed by the strong industrial setting of the project. For confidentiality reasons much of the products and results which came out after this research effort cannot be disclosed. Anyway a detailed description of the solutions employed and of the reasons behind them has been given.

1.3 Research effort in the *Communication* field

The background of this part of the work stands on a new technology paradigm in the field of communication architectures for Systems-on-Chip, that is called Network-on-Chip. Network-on-chip architectures try to provide a better solution to the communication requirements of modern SoCs making extensive use of concepts like modularity and scalability. The extensive amount of research work that has been conducted in the last years in this area both in the academic and industrial community shows that the Network-on-Chip approach can bring real benefits to communication in the context of SoCs. The work conducted in this field in the context of the present thesis has focused on improving the already good modularity and scalability properties of Network-on-Chip architectures. In particular the objective of this part of the work has been to improve the architecture of an industrial Network-on-Chip, STMicroelectronics STNoC™, in order to reach higher performance levels in conditions were the underlying System-on-chip includes IP cores and subsystems which have heterogeneous legacy socket protocols. This is a quite sensitive topic in

the industry and has already received attention by the research community, but no ideal solutions have been already found. Therefore the goal of this part of the work has been to investigate the performance of an innovative approach to the so called *multi-socket* compatibility problem in the context of Networks-on-chip. As in the above case much of the product and the results of this work cannot be disclosed for confidentiality reasons. Again, all the architecture of the proposed solution is exposed, and some results regarding simplified configurations of the systems developed have been shown.

1.4 Overview of the thesis

The thesis is split in two parts. The first deals with the effort in the SoC Communication field, while the second deals with the SoC Computation field. Each one of the two parts starts with chapters introducing the state of the art in the field and the concepts behind that particular technological sector and ends with a chapter describing the architectures developed. At the end of the document has been included an appendix were some of the network traffic scenarios resulting from the application of the solutions developed in the context of the multi-socket NoC architecture problem have been placed.

Part I

SoC communication architectures:
Multi-socket support in STNoC architecture.

Chapter 2

Traditional on-chip communication architectures

2.1 State of the art bus-based interconnection systems

In today's SoC designs the model of interconnection architecture most widely used is the *shared bus*, in some of its variants. Usually not all the IP Cores inside a chip are interfaced to a single bus, but they are partitioned in terms of speed. One bus domain is typically used for example to connect cores that must sustain an high data rate, while cores with less demands are coupled through a slower bus. In this case the two bus domains are put in communication by means of a Bridge that allows data to pass from one domain to the other. This kind of architecture is called *hierarchical bus*. In a modern SoC there can be even much more that two bus domains. There are many examples of shared bus architectures available in the market. Among them AMBA [2](ARM bus architecture), widely adopted thanks to the strong presence of ARM processors in SoCs, in all its versions from APB to AHB and AXI [1]. There are also other bus architectures as STBus [5], STMicroelectronics proprietary bus system, IBM CoreConnect [6], OCP (Open Core Protocol)[4]. Each one of the bus architectures listed above is characterized by its own bus protocol (interface signals and handshake mechanism). There are many similarities among these protocols, like supported type of transactions, basic handshake mechanism, but in general every protocol is different from each other. These protocol differences are a legacy of the past, when almost often every vendor fabricated its chips using proprietary components. The exchange of components between different vendors was very limited.

The move that we have seen in the last couple of years where SoCs, which in the past where mainly custom built for a specific application, are becoming more and more full featured computer platforms embedding components as general purpose processors, digital signal processors, peripherals, multimedia hardware accelerators, has significantly increased the exchange of components among different vendors. For this reason there is the need to provide some means to bridge the protocol gap. Protocol converters are required which often degrade system performance. Today there are many efforts across the industry targeted at solving completely the protocol mismatch problem through the use of standards to overcome what many people in the industry calls "protocol madness". In the following paragraphs an overview of a selection of bus the most widely used protocols available today in the market is given. The major properties of the protocols are listed and their major differences are highlighted.

2.1.1 AMBA Bus

The Advanced Microcontroller Bus Architecture (AMBA 2.0) bus represents the first generation of interconnect systems developed by ARM[2]. The AMBA system is actually a collection of buses. There are three AMBA bus protocols: AHB, ASB and APB. The AHB bus is a high performance system. Supports high operating frequency. ASB is an alternative bus which can be used when the high performance features of AHB are not needed. The APB bus is optimized for minimal power consumption and reduced interface complexity. It can be used in conjunction with either of the preceding versions of AMBA.

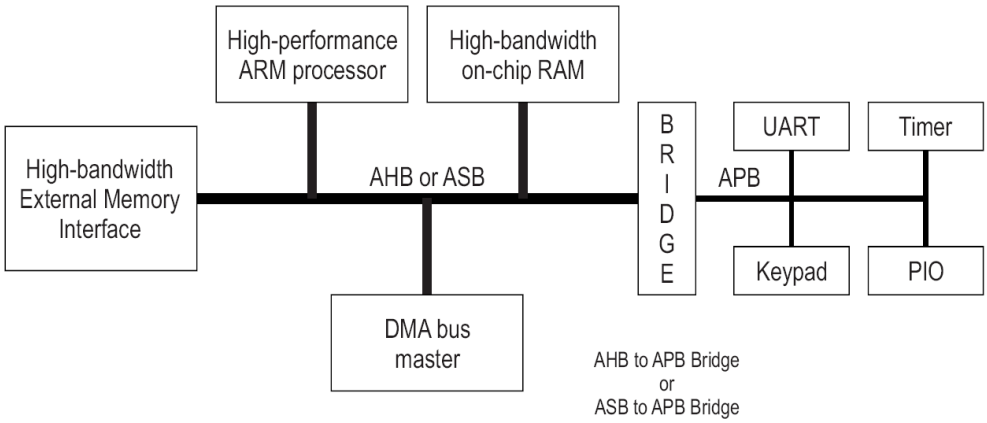


Figure 2.1: Typical AMBA system.

The AMBA AHB bus protocol is designed for being used with a central multiplexor interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer. AMBA AHB provides support for:

- burst transfers
- split transactions
- single cycle bus master handover
- single clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits)

2.1.2 IBM CoreConnect

The IBM CoreConnect [6] architecture provides three buses for interconnecting cores, library macros, and custom logic:

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)
- Device Control Register (DCR) Bus

The CoreConnect architecture shares many similarities with the Advanced Microcontroller Bus Architecture (AMBA) from ARM Ltd, as shown in Fig. 2.3. Both architectures support data bus widths of 32-bits and higher, utilize separate read and write data paths and allow multiple masters. CoreConnect and AMBA 2.0 provide both high performance features including pipelining, split transactions and burst transfers. Many custom designs utilizing the high performance features of the CoreConnect architecture are available in the marketplace today.

Figure (REF) illustrates how the CoreConnect architecture can be used to interconnect macros in a PowerPC 440 based SOC. High performance, high bandwidth

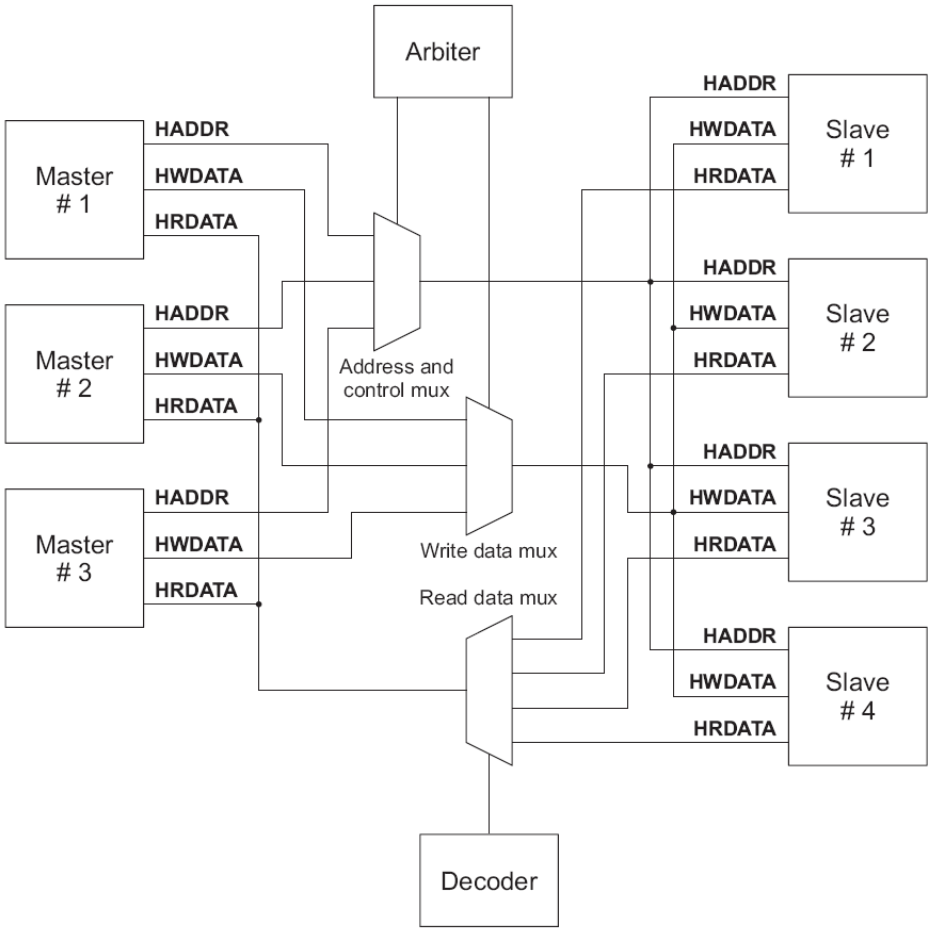


Figure 2.2: Multiplexor Interconnection.

blocks such as the PowerPC 440 CPU core, PCI-X Bridge and PC133/DDR133 SDRAM Controller reside on the PLB, while the OPB hosts lower data rate peripherals. The daisy-chained DCR bus provides a relatively low-speed data path for passing configuration and status information between the PowerPC 440 CPU core and other on-chip macros.

The PLB and OPB buses provide the primary means of data flow among macro elements. Because these two buses have different structures and control signals, individual macros are designed to interface to either the PLB or the OPB. Usually the PLB interconnects high-bandwidth devices such as processor cores, external memory interfaces and DMA controllers. The PLB addresses the high performance, low latency and design flexibility issues needed in a highly integrated SOC through:

	IBM CoreConnect Processor Local Bus	ARM AMBA 2.0 AMBA High-performance Bus
Bus Architecture	32-, 64-, and 128-bits Extendable to 256-bits	32-, 64-, and 128-bits
Data Buses	Separate Read and Write	Separate Read and Write
Key Capabilities	Multiple Bus Masters 4 Deep Read Pipelining 2 Deep Write Pipelining Split Transactions Burst Transfers Line Transfers	Multiple Bus Masters Pipelining Split Transactions Burst Transfers Line Transfers
	On-Chip Peripheral Bus	**AMBA Advanced Peripheral Bus**
Masters Supported	Supports Multiple Masters	Single Master: The APB Bridge
Bridge Function	Master on PLB or OPB	APB Master Only
Data Buses	Separate Read and Write	Separate or 3-state

Figure 2.3: CoreConnect vs AMBA 2.0 features.

- Decoupled address, read data, and write data buses with split transaction capability
- Concurrent read and write transfers yielding a maximum bus utilization of two data transfers per clock
- Address pipelining that reduces bus latency by overlapping a new write request with an ongoing write transfer and up to three read requests with an ongoing read transfer.
- Ability to overlap the bus request/grant protocol with an ongoing transfer
- non-tristate implementation

In addition to providing a high bandwidth data path, the PLB offers designers flexibility through the following features:

- Support for both multiple masters and slaves
- Four priority levels for master requests allowing PLB implementations with various arbitration schemes
- Deadlock avoidance through slave forced PLB rearbitration
- Master driven atomic operations through a bus arbitration locking mechanism
- Byte-enable capability, supporting unaligned transfers

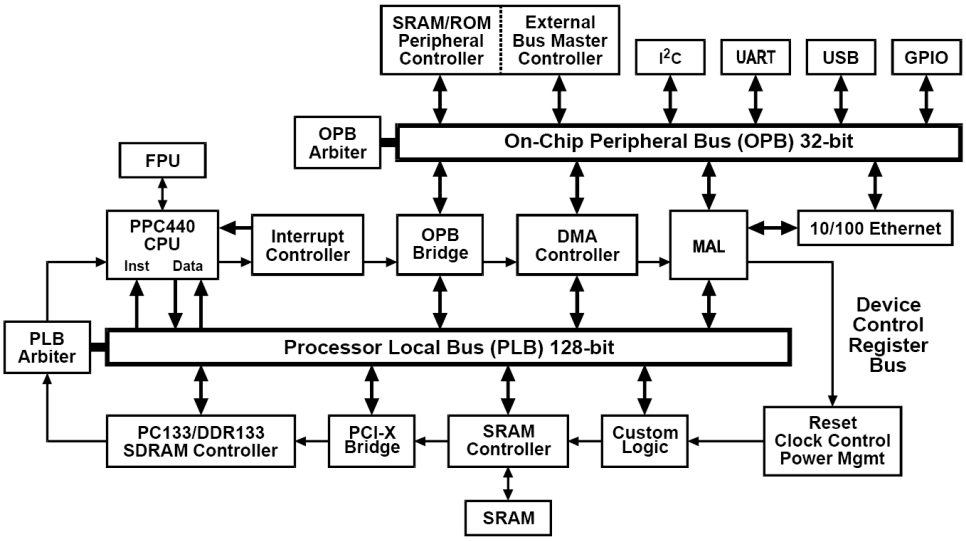


Figure 2.4: CoreConnect vs AMBA 2.0 features.

- A sequential burst protocol allowing byte, half-word, word and double-word burst transfers
- Support for 16-, 32- and 64-byte line data transfers
- Read word address capability, allowing slaves to return line data either sequentially or target word first
- DMA support for buffered, fly-by, peripheral-to-memory, memory-to-peripheral, and memory-tomemory transfers
- Guarded or unguarded memory transfers allow slaves to individually enable or disable prefetching of instructions or data
- Slave error reporting
- Architecture extendable to 256-bit data buses
- Fully synchronous

2.1.3 OCP Bus

The Open Core Protocol (OCP) [4] delivers the only non-proprietary, openly licensed, core-centric protocol that comprehensively describes the system-level integration requirements of intellectual property (IP) cores. The OCP supports very

high performance data transfer models ranging from simple request-grants through pipelined and multi-threaded objects. Higher complexity SOC communication models are supported using thread identifiers to manage out-of-order completion of multiple concurrent transfer sequences. The OCP defines a point-to-point interface between two communicating entities such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave. Figure 2.5 shows a simple system containing a wrapped bus and three IP core entities: one that is a system target, one that is a system initiator, and an entity that is both.

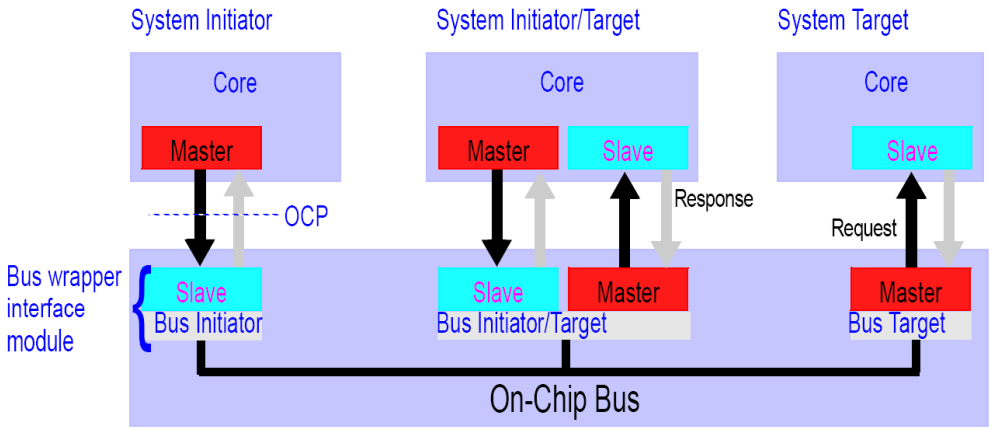


Figure 2.5: System Showing Wrapped Bus and OCP Instances.

The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP; the wrapper interface modules must act as the complementary side of the OCP for each connected entity. A transfer across this system occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a bus wrapper interface module). The interface module plays the request across the on-chip bus system. The OCP does not specify the embedded bus functionality. Instead, the interface designer converts the OCP request into an embedded bus transfer. The receiving bus wrapper interface module (as the OCP master) converts the embedded bus operation into a legal OCP command. The system target (OCP slave) receives the command and takes the

requested action. Each instance of the OCP is configured (by choosing signals or bit widths of a particular signal) based on the requirements of the connected entities and is independent of the others. For instance, system initiators may require more address bits in their OCP instances than do the system targets; the extra address bits might be used by the embedded bus to select which bus target is addressed by the system initiator. Main features of the OCP interface are:

- Point-to-Point Synchronous Interface
- Bus Independence
- Pipelining
- Separation between requests and responses
- Support of Bursts through annotation of transfers with burst information
- Support for transmission of in-band information. A typical use of in-band extensions is to pass cacheable information or data parity
- Out-of-order request and response delivery using multiple threads and tags

2.1.4 AMBA AXI

The AMBA AXI protocol (AMBA 3.0) [1] is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed submicron interconnect. The objectives of the latest generation AMBA interface are to:

- be suitable for high-bandwidth and low-latency designs
- enable high-frequency operation without using complex bridges
- meet the interface requirements of a wide range of components
- be suitable for memory controllers with high initial access latency
- provide flexibility in the implementation of interconnect architectures
- be backward-compatible with existing AHB and APB interfaces.

The key features of the AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes
- burst-based transactions with only start address issued
- separate read and write data channels to enable low-cost Direct Memory Access (DMA)
- ability to issue multiple outstanding addresses
- out-of-order transaction completion
- easy addition of register stages to provide timing closure

As well as the data transfer protocol, the AXI protocol includes optional extensions that cover signaling for low-power operation. The AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write data channel to the slave or a read data channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI protocol has an additional write response channel to allow the slave to signal to the master the completion of the write transaction. The AXI protocol enables:

- address information to be issued ahead of the actual data transfer
- support for multiple outstanding transactions
- support for out-of-order completion of transactions

Fig. 2.6 shows how a read transaction uses the read address and read data channels.

Each of the five independent channels consists of a set of information signals and uses a two-way VALID and READY handshake mechanism. The information source uses the VALID signal to show when valid data or control information is available on the channel. The destination uses the READY signal to show when it can accept the data. Both the read data channel and the write data channel also include a LAST signal to indicate when the transfer of the final data item within a transaction takes place. Read and write address channels Read and write transactions each have their own address channel. The appropriate address channel carries all of the required address and control information for a transaction. The AXI protocol supports the following mechanisms:

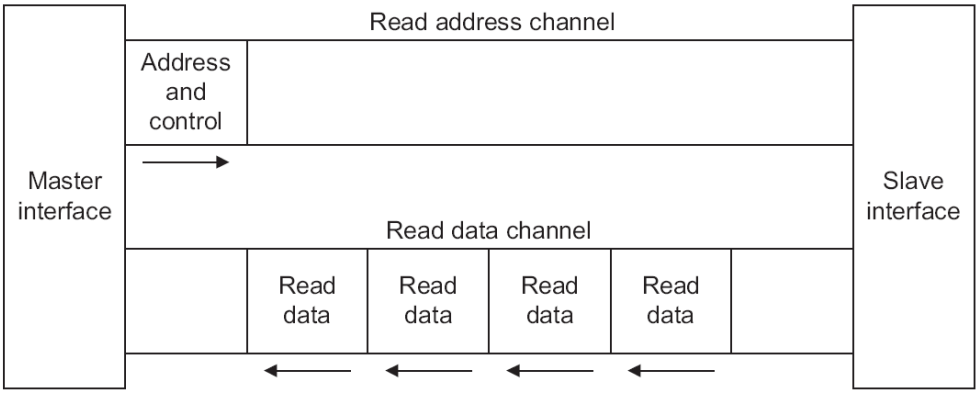


Figure 2.6: Channel architecture of reads.

- variable-length bursts, from 1 to 16 data transfers per burst
- bursts with a transfer size of 8-1024 bits
- wrapping, incrementing, and non-incrementing bursts
- atomic operations, using exclusive or locked accesses
- system-level caching and buffering control
- secure and privileged access

The AXI protocol supports also a number of advanced features. It offers support for three different burst types, suitable for normal memory accesses, wrapping cache line bursts and streaming data to peripheral FIFO locations. The cache support signal of the AXI protocol enables a master to provide to a system-level cache the bufferable, cacheable and allocate attributes of a transaction. Three levels of protection unit support ar provided, enabling both privileged and secure accesses. The AXI protocol defines also mechanisms for both exclusive and locked accesses. Moreover, to enhance the performance of the initial accesses within a burst, the AXI protocol supports unaligned burst start addresses.

2.1.5 STMicroelectronics STBus

The STBus [5] is a set of protocols, interfaces, primitives and architectures specifying an interconnect subsystem, versatile in terms of performance, architecture and

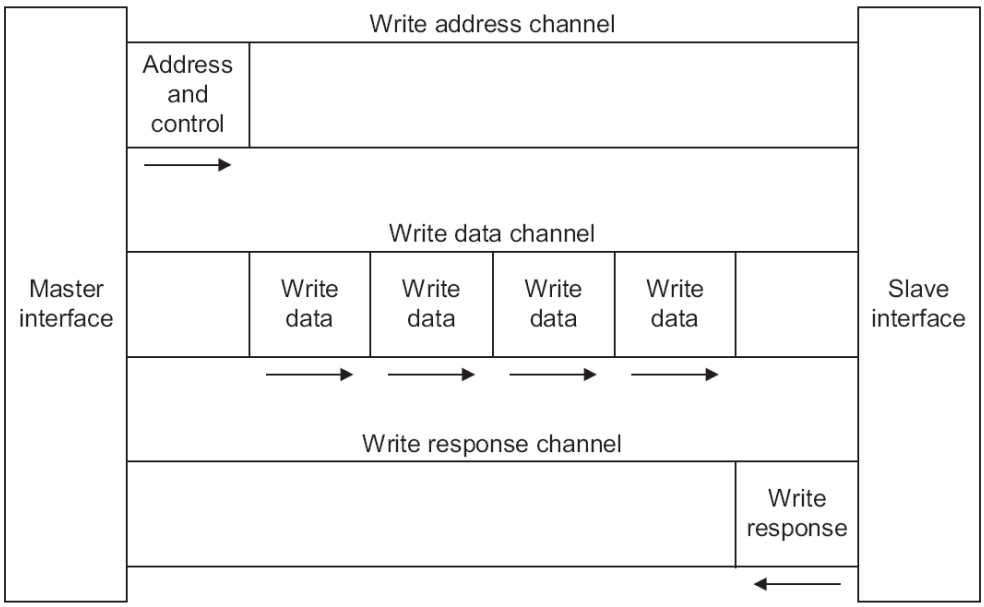


Figure 2.7: Channel architecture of writes.

implementation. The STBus is the result of the evolution of the interconnect subsystem developed for microcontrollers dedicated to consumer application, such as set top boxes, ATM networks, digital still cameras and others. Such an interconnect was born from the accumulation of ideas converging from different sources, such as the transputer (ST20), the Chameleon program (ST40, ST50), MPEG video processing and VCI (Virtual Component Interface) organization. Today the STBus is not only a communication system characterized by protocol, interfaces, transaction set and IPs, but also a technology allowing to design and implement communication networks for Systems On Chip with the support of a development environment including tools for system level design and architectural exploration, silicon design, physical implementation and verification. Three different types of the STBus protocols exist, each having a different level of complexity in terms of both performance and implementation:

- Type1 is the simplest and is intended to be used for peripherals registers access. No pipeline applies. It acts as a RG protocol. Load/store on 1/2/4/8 bytes are supported.
- Type 2 adds pipelines features. It is equivalent to the basic RGV protocol. It supports all operation code for ordered transactions. The number of the requesting cells (i.e. in a packet) is the same than the number of the response

ones.

- Type 3 is an advanced protocol implementing split transactions for high bandwidth requirements (high performance systems). It supports out of order executions. The packet response size might be different than the packet request size (the number of cells differs between request and response). The interfaces maps the STBus transaction set on a physical set of wires defined by this interface.

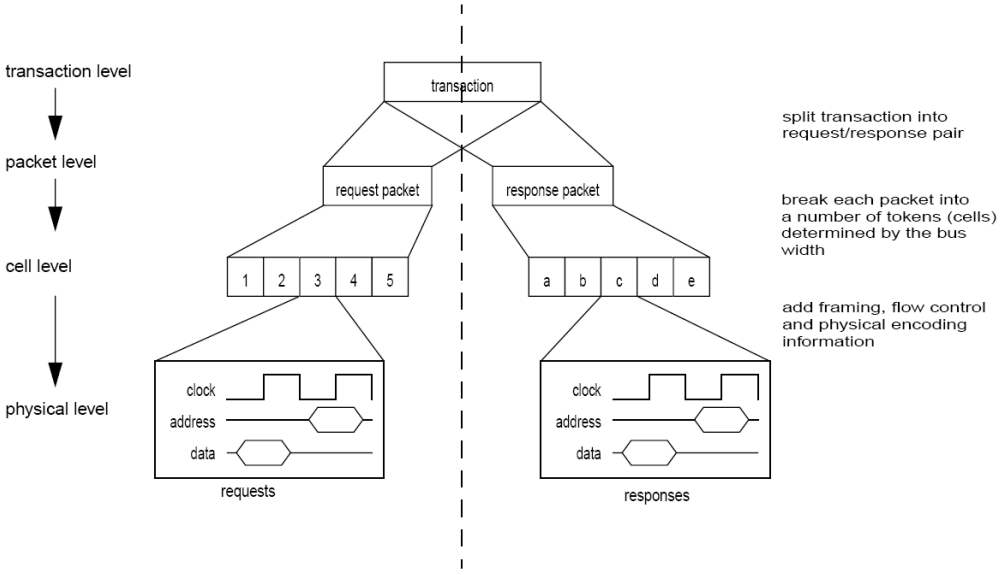


Figure 2.8: STBus protocol layers.

Fig. 2.8 describes the layered architecture of STBus protocol. First of all a single transaction is composed by a request phase and by a response phase. Each one of these transaction phases is in turn composed by a particular number of cells, or bus cycles. Depending on the Type of the STBus protocol, the number of cells for request and response transactions can be the same or not. At the lowest possible layer cells are transmitted by means of specific flow control and physical encoding.

Chapter 3

Network-on-Chip technology overview

3.1 NoC: an unifying concept

Chip design has four distinct aspects: computation, memory, communication and I/O. As processing power has increased and data intensive applications have emerged, the challenge of the communication aspect in single-chip systems, SoC, has attracted increasing attention. This Chapter treats an important concept for communication in SoCs known as Network-on-Chip (NoC). As will become clear in the following paragraphs, NoC does not constitute an explicit new alternative for intrachip communication but is rather a concept which presents a unification of on-chip communication solutions. The major driving factors for the development of global communication schemes are the ever increasing density of on-chip resources and the need to utilize these resource with a minimum of effort as well as the need to take into account the physical effects of DSM technologies. The preferred solution is to try taking advantage of economies of scale in the system design, dividing the processing resource into smaller pieces and reusing them as much as possible inside the overall design. This strategy helps also reducing design cycle time, since the entire chip design process can be divided into almost independent subproblems. The partitioning of the design allows also to use modular verification methodologies. Verification is performed at lower level of abstraction for the elementary modules and then verification at a more abstract level is performed for the design as whole. Working at an higher level of abstraction leads to a differentiation of local and global communication. As intercore communication is becoming the performance bottleneck in many multicore applications , there is a shift in design focus from a traditional processing-

centric to a communication-centric one. NoCs interconnection paradigm provides a standardized global communication scheme that coupled with the use of standard communication sockets for the IP cores would make a Lego brick-like plug-and-play design style possible, allowing good use of the available resources and fast product design cycles.

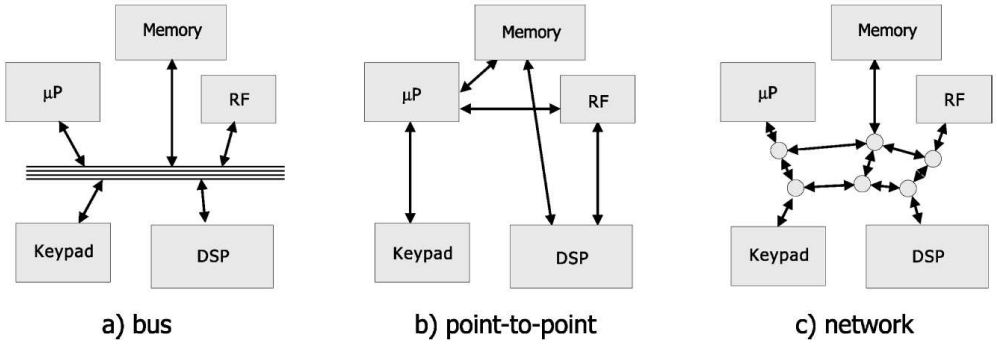


Figure 3.1: Examples of communication structures in SoC. a) bus-based, b) dedicated point-to-point links, c) chip area network.

Since the introduction of the SoC concept in the 90s, the solutions for SoC communication structures have generally been characterized by a mixture of buses and poit-to-point links, as depicted in Fig. 3.1. The major drawback of bus architectures is that in highly interconnected multicore systems it can quickly become a communication bottleneck. In fact, it is not ultimately scalable, since as more units are added to it, the power dissipation for bus access grows as a consequence of the increased capacitive load. Dedicated point-to-point links are optimal in terms of performance, but the number of links required increases exponentially with the number of cores, leading to a potential area and routing problem. For maximum flexibility and scalability it is generally accepted that a move towards a shared segmented global communication structure is needed. This definition leads in turn to a data-routing network consisting of communication links and routing nodes that are implemented on the chip. Such a distributed communication media scales well with chip size and complexity. Additional advantage include increased aggregated performance by exploitation of parallel operation. Interestingly, a similar solution is suggested also by silicon technology issues. In DSM chips long wires must be segmented in order to avoid signal degradation. The next natural step is to increase throughput by pipelining these structures. Wires become pipelines and bus-bridges become routing nodes. Therefore the distinction between different communication solutions is fading. In this context NoC can be considered as a unifying concept

rather than an explicit new communication model.

3.2 NoC basic concepts

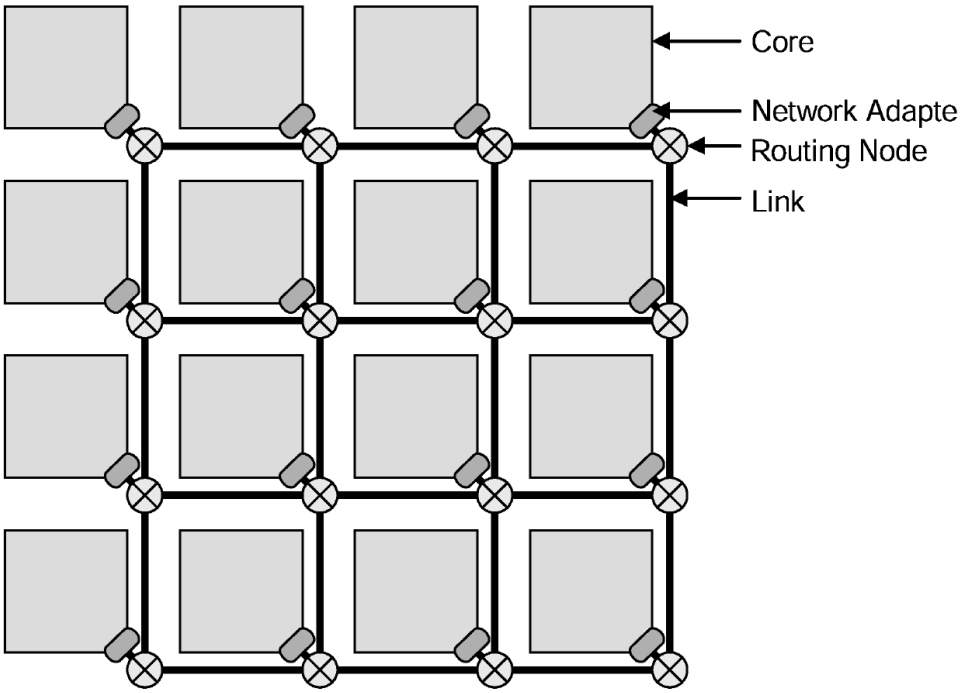


Figure 3.2: 4-by-4 grid structured NoC.

The three fundamental blocks found inside Networks-on-Chip are:

- **Network Adapters**. They implement the interface by which every single IP core connects to the NoC. Their function is to decouple computation (the cores) from communication (the network).
- **Routing Nodes**. They route data according to chosen protocols and implement the routing strategy.
- **Links**. Connect the nodes, providing the raw bandwidth. They may consist of one or more logical or physical channels.

Fig. 3.2 shows a sample NoC structured as a 4-by-4 grid which provides global chip level communication. In general, SoCs do not necessarily exhibit such a regular

architecture. System composition can be categorized by means of the degree of *homogenity* and *granularity* of the IP cores. The architecture of the communication network has to take into account the actual system composition. Therefore NoC-based systems implement a very high degree of variety in composition and in traffic diversity. This is different from what happens in traditional parallel computers, where the architecture is typically homogeneous and coarse grained.

3.2.1 Network Abstraction

In today's research the term NoC is used in a very broad sense, including topics that span from gate level physical implementation , across system layout aspects and applications, to design methodologies and tools. The major reason that stands behind the widespread adoption of typical network terminology lies on the readily available and widely accepted abstraction models of networked communications. The OSI model for layered network communication can be in fact easily adapted to NoCs, as done in [10]. To properly understand the research work done today in relation to NoC architectures is convenient to partition the spectrum of NoC research into four areas: 1) *system*, 2) *network adapter*, 3) *network* and 4) *link* research. Fig. 3.3 shows the relation between these research areas, the fundamental components of NoCs and the OSI layers.

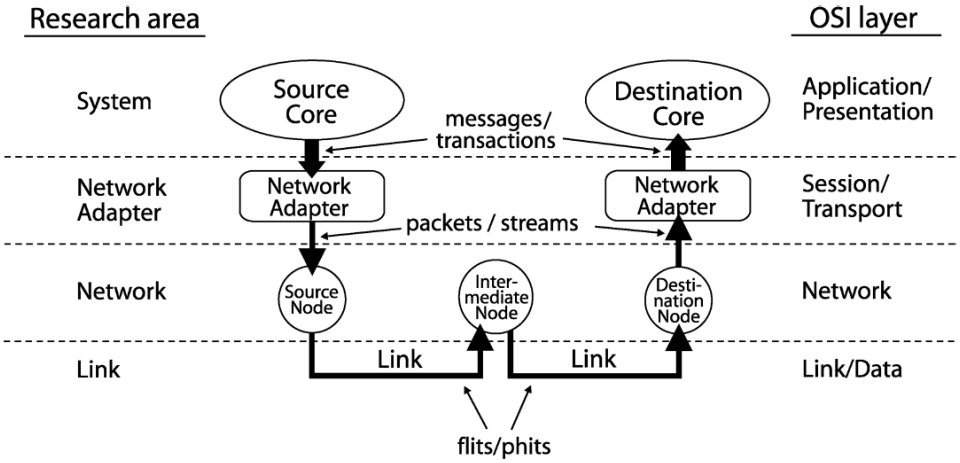


Figure 3.3: Flow of data from source to destination across abstraction layers.

The *system* area deals with applications (processes) and architecture (cores and network). At this level most of the implementation details are hidden. The *Network Adapter* bridges the gap between cores and network. It handles the end-to-end flow

control, through encapsulation of the messages generated by the IP core. These data are broken into packets, which may have or not information about their destination. In the latter case there must be a path setup phase prior of the actual packet transmission. The *Network Adapter* is the first level that is network aware. The *network* consists in the routing nodes and links defining the topology and implementing protocol and the nod-to-node flow control. The lowest level of abstraction is the *link* level. The basic datagrams at this level are flits (flow control units), from which packets are made up. Sometimes is defined also a further subdivision of flits into phits (physical units), which are the minimum piece of information that can be transmitted simultaneously on a link. Most commonly flits and phits are equivalent. Link-level research deals with encoding and synchronization issues. In general, in a NoC the abstraction layers are more closely bound than in a macronetwork. Issues at a physical level of abstraction affect to a great extent also the highest abstraction layers. Also, NoCs benefit from the system composition being completely static. This very often leads to higher performance.

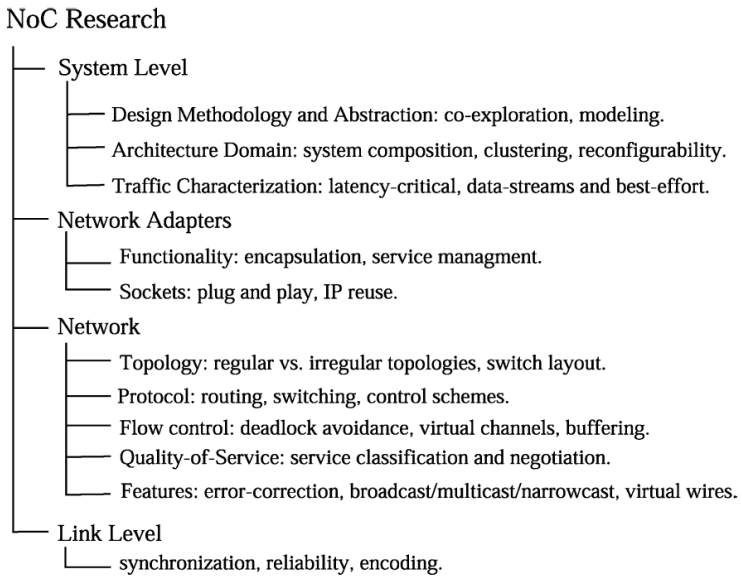


Figure 3.4: NoC research area classification.

3.3 NoC Research

In this section is provided a review of some of the approaches found in literature, listed according to the layers defined in the previous paragraph.

3.3.1 Network Adapter

The function of the *Network Adapter* is to interface the IP core to the network, managing at the same time to make communication services available with little effort from the core.

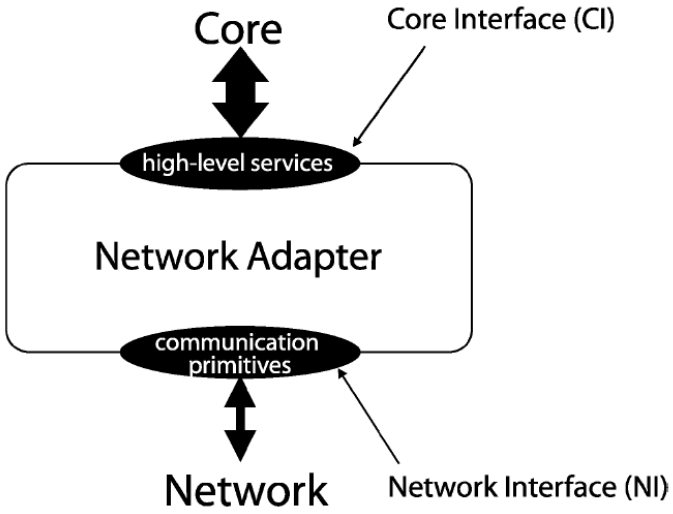


Figure 3.5: The Network Adapter.

Fig. 3.5 shows that the component exposes a core interface (CI) to the core and a Network Interface (NI) to the network side. The ultimate purpose of the NA is to provide the communication services needed by the core by means of primitive services provided directly by the network hardware. In this way the NA *decouples* the core from the network, enabling the implementation of a layered system design approach. The level of decoupling can significantly vary. Accordingly the core can be made more or less *network aware*. In the first case design and resource reuse are maximized, while in the latter there is the potential to make optimal use of the network resources. Typically the CI of the *Network Adapter* is implemented to adhere to a SoC *socket* standard. Socket standards are almost always identified with some legacy bus protocols, as the ones described in Chapter 2. Implementing a given socket standard as the CI of the *Network Adapter* allows in principle to attach to the network any IP core compliant to that given socket (a particular bus protocol). The *Network Adapter* performs encapsulation of the traffic for the underlying communication media. This may include global addressing and routing tasks, reorder buffering and data acknowledgement, buffer management to prevent network congestion, packet creation in a packet based NoC. The design of the *Network*

Adapter is a critical task in the overall NoC design process. Often this component handles tasks as frequency conversion and data size conversion between core side and network size, in order to improve flexibility.

3.3.2 Network Level

The role of the network is to provide the resources needed to deliver messages from their source to their destination. This is done giving hardware support for the basic communication primitives. On-Chip networks are mainly defined by their *Topology* and *Protocol*. The network's *Topology* describes how nodes and links are connected. *Protocol* specifies how these nodes and links are used.

Topology

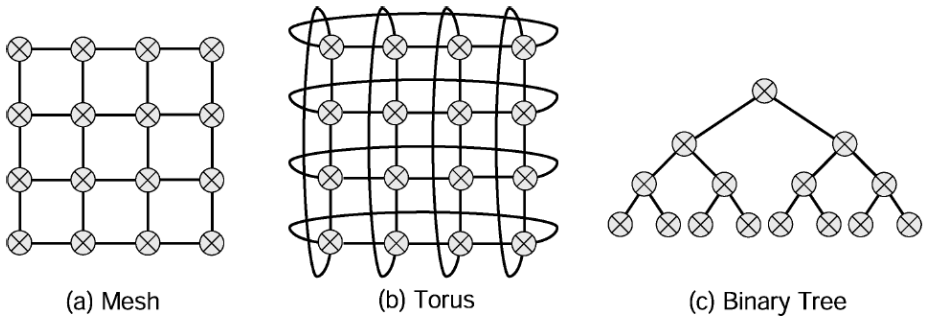


Figure 3.6: Typical network topologies.

A simple way to distinguish different regular topologies is in terms of $k-ary$ $n-cube$, where k is the degree of each dimension and n is the number of dimensions. The $k-ary\ tree$ and the $k-ary\ n-dimensional$ fat tree are two alternate regular networks explored in NoC (see Fig. 3.6). Most NoCs implement topologies that can be easily laid out on a chip surface. For example , k-ary 2-cube, typical grid topologies. Another topology used in the context of NoC is Spidergon™ [12], developed by STMicroelectronics (Fig. 3.8). Spidergon™ topology is derived by a ring by addition of more links. The new links are used to connect each node of the ring to the node directly across the ring's diameter. A more sound description of Spidergon™ and of STMicroelectronics Network-on-Chip technology will be given in the following paragraphs. Irregular forms of topologies are derived by mixing different forms in a hierarchical, hybrid or asymmetrical fashion, as seen in Fig. 3.7.

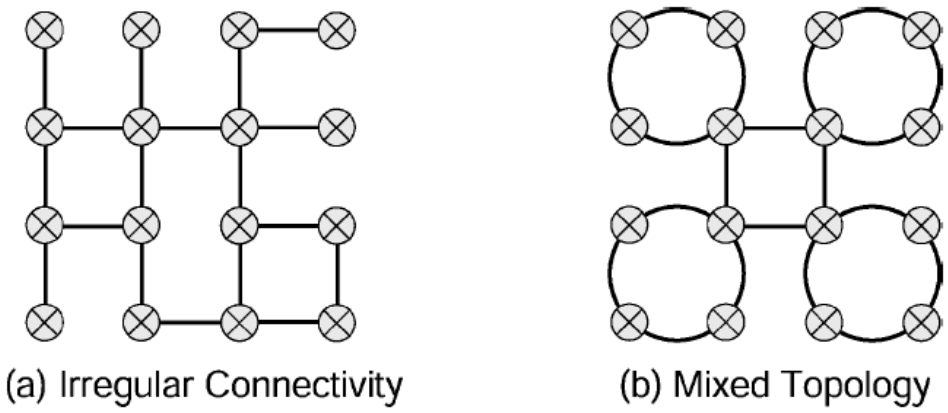


Figure 3.7: Irregular network topologies.

Networks where every node is connected to a source or a sink for the messages are called *direct networks*, while topologies that have a subset of nodes that are not connected to any source or sink are called *indirect networks*.

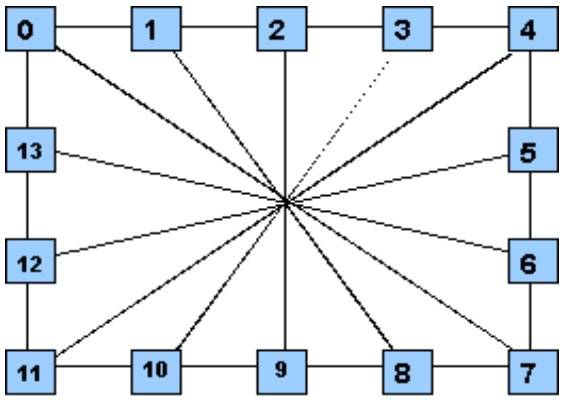


Figure 3.8: Spidergon™ topology.

Protocol

The protocol governs the way data is moved through the NoC. The protocol encompasses the concepts of *switching*, which is the mere transport of data, and *routing*, which determines the path followed by the data across the network. In the following will be discussed these and other aspects of protocol relevant for NoC.

Circuit vs packet switching. In circuit switching an entire circuit is setup from source to destination until the transport of data is complete. Packet switched traffic on the other hand flows on a per-hop basis. Each packet contains routing information as well as data.

Deterministic vs Adaptive routing. In deterministic routing the path for the packet is entirely specified by source and destination addresses. Popular deterministic routing strategies are source routing and X-Y routing. In source routing is the source core that specifies the path to destination. In X-Y routing the packet flows first on the horizontal direction and then in vertical direction, or vice versa. With adaptive routing the routing decision is taken at each hop. Adaptive mechanisms involve dynamic arbitration mechanisms, where the arbiter takes into account the local state of the network, for example the local link congestion. This results in a more complex router implementation but often offers benefits like load balancing.

Minimal vs nonminimal routing. A routing algorithm is minal if it chooses only among shortest paths between source and destination, otherwise is nonminimal.

Delay vs Loss. In the delay model datagrams (flits, phyts) are never lost. The worst thing that can happen is that the arrival of data is delayed. In the loss model instead, datagrams can be dropped. In this case means for data retransmission are required at the level of routers, introducing significant overhead. There are however some advantages with this model. For example dropping flits can be used for resolving network congestion.

Central vs distributed control. In centralized control systems routing decision are taken globally, for example by means of an arbiter. In distributed control instead routing decisions are made locally. NoCs usually employ the latter solution .

All the aspects involved in the network protocol described above have a direct impact on the node (router) implementation. Fig. 3.9 shows the major components of any routing node: buffers, switch, routing and arbitration unit and link controller.

The switch connects input buffers to output buffers, while the routing and arbitration unit implement the algorithm that endorses the routing policy. In centralized control systems the routing and arbitration unit would be common to all the nodes. In some cases [19] the routers include explicit time division multiplexing mechanisms, sometimes called TDN (Time Disjoint Networks). Packet that are in different TDN are guaranteed to not collide. As already said before the optimal design of a router

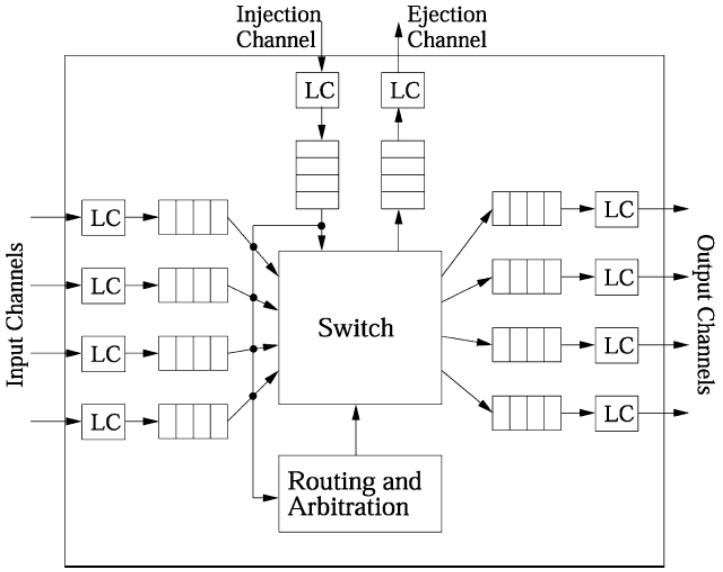


Figure 3.9: Generic Router model. LC= link controller.

is strictly related to the services it is supposed to provide. For example, support for adaptive bandwidth control can be provided simply adding to the basic architecture of Fig. 3.9 an additional bus, allowing to bypass the crossbar switch when congestion occurs. Some estimations say that in this case the router throughput can be improved by something around 27%. Many investigations in literature have focused on a comparison of the performance of adaptive versus deterministic protocols. A widespread opinion is that while adaptive protocol can speedup delivery of particular packets, deterministic protocols are superior by a global poit of view. The reason for this is that application of adaptive protocols tends to concentrate traffic in the center of the network, increasing congestion there. The most successful trends in NoC research today include packet switching, in contrast with circuit switching, and delay-based transmission, since the overhead of data retransmission is often considered unacceptable. The most common forwarding strategies studied in literature are: 1) *store-and-forward*, 2) *wormhole* and 3) *virtual cut through*. These will now be explained.

Store-and-forward. It is a packet switched protocol where any node stores the entire packet before forwarding it to the next node along the route. To find out to which output the data must be propagated the node looks to the header of the packet. This implies that transmission can be stalled when the node downstream does not have sufficient space on its internal buffers to hold the entire packet. See

Fig. 3.10 for a time space diagram of a packet switched message propagated with the store-and-forward algorithm.

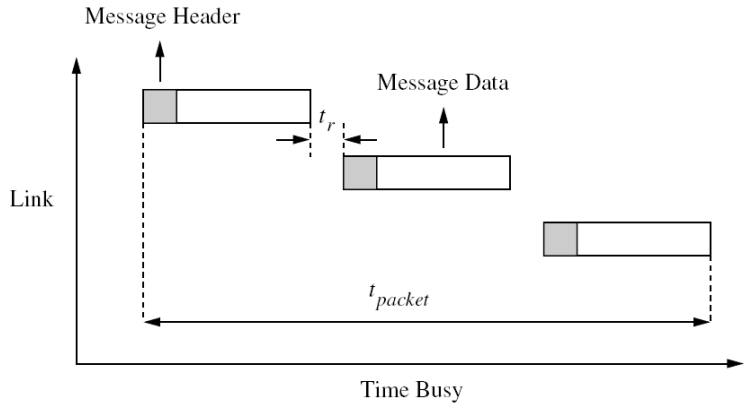


Figure 3.10: Store-and-forward packet propagation.

Wormhole. The major difference with store-and-forward is that in this case data propagation happens on a flit by flit basis. When a new packet arrives to a node the switch makes the routing decision upon the header information, usually placed inside the first flit. Subsequent flits are forwarded along the same route as soon as they arrive to the node. This implies that a single packet can span many nodes across the network, behaving somehow like a *worm*. The packet is actually pipelined through the network. Latency on the single node is significantly reduced in respect of that of store-and-forward. The major drawback is that stalling the packet has the unpleasant effect of stalling all the links occupied by the packet along the path. In the following we will see that the use of *Virtual Channels* can significantly alleviate this problem. Fig. 3.11 shows a time space diagram of a packet switched message propagated with the wormhole algorithm.

Virtual cut-through. Virtual cut-through has a forwarding mechanism similar to wormhole. The only difference is that the header is propagated to the next node only if the node itself guarantees to have enough space to hold all the packet. Otherwise propagation is stalled and all the packet is gathered at the current node. When the node down stream has enough room, the header and the following flits are forwarded as soon as they arrive. The advantage of this solution is that when the packet is stalled no links along the route are blocked. Fig. 3.12 shows a time space diagram of a packet switched message propagated with the virtual cut-through algorithm.

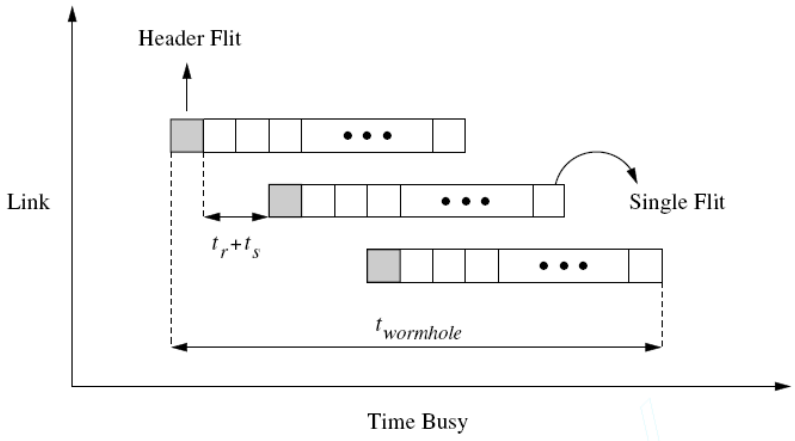


Figure 3.11: Wormhole packet propagation.

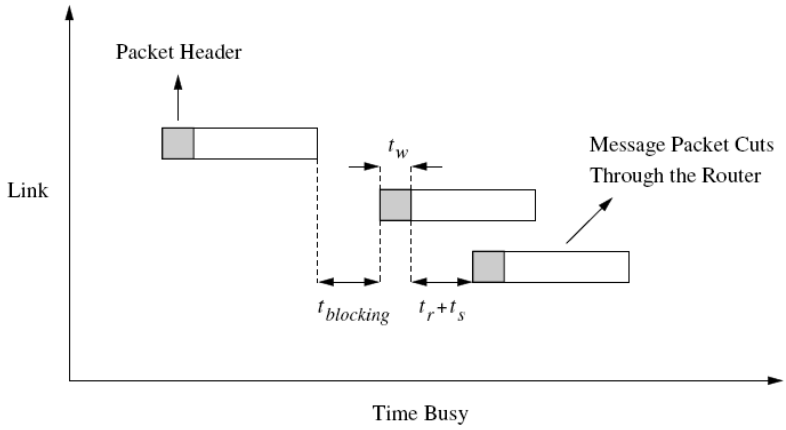


Figure 3.12: Virtual cut-through packet propagation.

For NoC the prevailing routing scheme is wormhole, while for macronetworks it is store-and-forward the preferred solution. The major advantages offered by wormhole are the low latency, no need to wait for the entire packet at each node, and the significant reduction in buffer space required at each node, which is the dominant cost factor for the implementation of routing nodes.

Flow Control

Flow control is the mechanism that determines the packet movement along the network path [20]. The basic purpose of flow control policies is to ensure correctness in the packet propagation process. In addition flow control deals also with optimiza-

tion of network resource usage, and with the offering of predictable performance to the network users. In the following a selection of the topics related to flow control will be discussed.

Virtual Channels. The concept of *Virtual Channels* deals with the sharing of a physical channel by several logically separated channels, which have individual and separated buffer queues (see Fig. 3.13). Some estimations specify that the number of VCs that can be used for a particular physical channel can be in the range between 2 and 16. Usage of *Virtual Channels* can cause significant implementation overhead, expecially for the hardware cost of additional buffer queues and the more sophisticate control logic of the physical channel, but it offers a number of important advantages. Among these are:

- **Deadlock avoidance.** It is possible to exploit the fact that VCs are not mutually dependent. In fact, adding VCs to links and choosing appropriately the routing policy it is possible to break cycles in the resource dependency graph [15].
- **Optimization of wire utilization.** In the future wire cost will dominate transistor cost [7]. Having several logical channels actually using a single physical channel enables a more efficient wire utilization. Also to mention the reduction in leakage power and wire routing congestion.
- **Performance improvement.** VCs minimize the possibility to have interresource dependency in the network. Some investigations have also shown [14] that dividing a fixed buffer size across a number of VCs improves the network performance at high loads.
- **Support for differentiated services.** VCs can be used to implement Quality-of-Service (QoS), by specifying different priorities for the different VCs tied to a single physical channel.

The most important task of any flow control mechanisms is to ensure *deadlock* and *livelock* avoidance. *Deadlock* is the situation that occurs when some of the resources inside the network are suspended waiting for other resources to be released, in a cyclic fashion. This situation occurs when one path is blocked waiting for another blocked as well. This condition can be avoided by breaking cyclic dependencies in the resource dependency graph. Actually this condition can be relaxed, as shown by [16]. It is in fact enough to require the existence of a channel subset

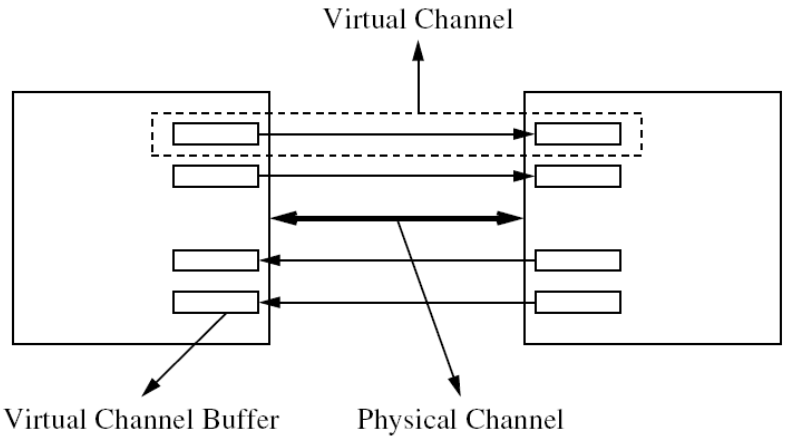


Figure 3.13: Virtual Channels.

which defines a connected routing subfunction with no cycles in the *extended* channel dependency graph. Livelock occurs when packets continue flowing around the network without ever reaching their final destination. Sometimes this can occur as a side effect of some forms of nonminimal adaptive routing policies. Fig. 3.14 shows how using VCs is sometimes possible to avoid stalls due to packets already blocked inside the network.

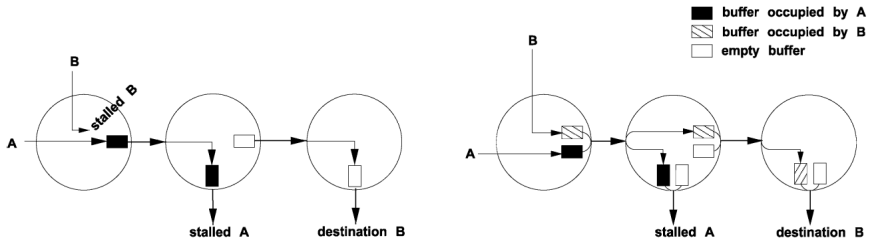


Figure 3.14: Streams of different VCs can proceed even if another stream sharing some of the links is stalled.

Quality of service

Quality of Service (QoS) in the context of NoC architectures defines the quantification of a particular service offered from the network to the cores. These services could be low latency, high throughput, low power, bounds on jitter, etc. Usually, two different QoS classes are identified: 1) *Best Effort* (BE) services, which offer no commitment and 2) *Guaranteed Services* which otherwise do. Also various

levels of commitment can be defined: 1) correctness of result, 2) completion of the transaction, 3) bounds on performance. In the NoC literature however traffic is defined as BE when only correctness and completion are guaranteed, while with GS additional guarantees are given, usually some bound on the performance of a transactions. The guarantees offered by NoC systems are almost always hard guarantees. In macronetworks instead service guarantees are often of statistical nature. Implementation of GS communication requires the allocation of resources that must be logically independent from other traffic in the system. In this case connections are instantiated as *virtual circuits* which use logically independent resources. The *virtual circuits* in turn can be implemented as *Virtual Channels*, time-slots, parallel switch fabric, etc. Most of NoCs that implement hard GS use variants of time division multiplexing (TDM).

3.3.3 Link Level

Link-level research studies the architectures of node-to-node links. These links consist of one or more channels, which can be virtual or physical. In the following will be presented two of the areas of interest in link-level research: 1) synchronization and 2) implementation.

Synchronization. For link-level synchronization in multiclock SoCs the crucial components are the multi-clock FIFO buffers. The design of these components is very critical. It is very important for the multi-clock FIFOs to be particularly robust with regards to metastability. Often reaching this goal requires losing something in terms of latency. In the context of link-level synchronization there is another concept that is gaining very much attention. It is the concept of *Globally Asynchronous Locally Synchronous* (GALS) design. In the GALS model, a system is built putting together a number of blocks that communicate with each other by means of asynchronous links, while internal communication is fully synchronous with a given local clock. There are examples of NoCs featuring asynchronous inter-node links [8]. Asynchronous logic implies some area and power overhead with respect to synchronous logic, due to the need to implement local handshake control. On the other hand, no power is consumed when the links are idle. One of the most widespread solutions for asynchronous links is the 1-of-4 encoding [9].

Implementation Issues. DSM physical issues (already described in one of the preceding paragraphs) have great impact on link delays and power consumption, which are both degraded. A number of techniques have been proposed in liter-

ature to improve the performance of NoC node-to-node links in the context of DSM technology. The first of these techniques is *wire segmentation*. A common solution has been for sometime to apply *repeaters* at regular intervals, in order to keep the delay linearly dependent on the length of the wire. Another technique widely used is *pipelining* of wire links. In this way the link throughput is effectively increased. Use of *pipelining* implies some overhead in terms of area, since pipeline stages are more complex that simple repeaters. But as in future DSM technology wire effects tend to dominate on area occupation, the overhead associated to pipelining is supposed to decrease.

3.4 STNoC™ Network-on-Chip

STNoC™ is an on-chip interconnection micro network designed to be a scalable and low overhead solution to the increasing SoC communication requirements. The vast majority of the solutions implemented in STNoC™ have been optimized for hardware efficiency. This can be seen for example looking at the way the routing algorithm is specified inside the router. Instead of using power and latency hungry routing tables, STNoC™ resorts to a direct combinational logic implementation within the router in order to decode the path that has to followed by the packet. The STNoC™ routing scheme is deterministic and it is based on Spidergon™ topology, already introduced in one of the preceding paragraphs but repeated in Fig. 3.15 for convenience.

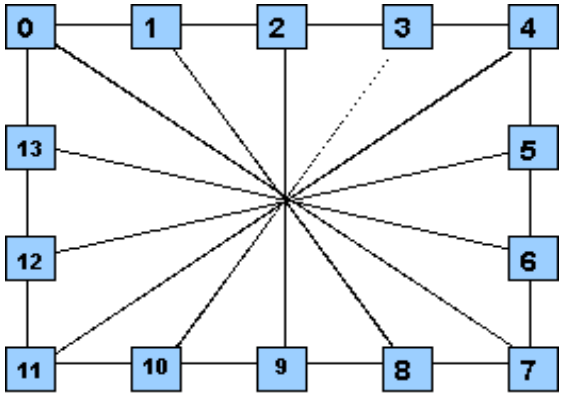


Figure 3.15: Spidergon™ topology.

The packet decoding scheme is light in terms of gate count, thanks to the high symmetry and regularity of the topology. In STNoC™ each router has a unique address I in the network, $0 \leq I \leq N$, where N is the network size in nodes. Since the

STNoC™ routing algorithm is local, identical for all router nodes, and the topology is both vertex and edge transitive, is possible to describe the routing algorithm at any node. For confidentiality reasons, the details of STNoC™ router microarchitecture and arbitration mechanism cannot be unveiled here. STNoC™ router proved to be a really cost effective solution, allowing clock frequency up to $1Ghz$ in 90nm ST technology, with a per link bandwidth of $8GB/sec$. One of the best properties of on chip networks is to hide the interconnect specific implementation details to the IP resources interfaced. All that is needed for an external IP to transmit data through the network is a specific designed Network Adapter. The architecture of the STNoC™ Network Adapters, as the architecture of the overall NoC, has been designed for modularity. It has two main components, *shell* and *kernel*. The *shell* handles the processing activity related to the signals coming from the IP resource side, while the *kernel* takes care of the activity related to the NoC side of the Network Adapter. Data and control passing between *shell* and *kernel* and viceversa happens basically through FIFOs. This partitioned NA architecture has been mainly conceived to minimize the effort needed to port the Network Adapters to IP resources compliant with different bus protocols. In fact in this way the only part that needs to be changed is the *shell*.

The architecture of the STNoC™ Network Adapters supports *kernel* and *shell* subsystems running at totally unrelated frequencies. The control circuitry of the FIFOs embedded inside the NIs allows for data to be read from one side and written from the other at completely unrelated frequencies (see Fig. 3.16). Depending on the bus domains that need to be interfaced to the NoC the internal FIFOs can be one or more. STNoC™ Network Adapters also support arbitrary data size conversion between IP and NoC side.

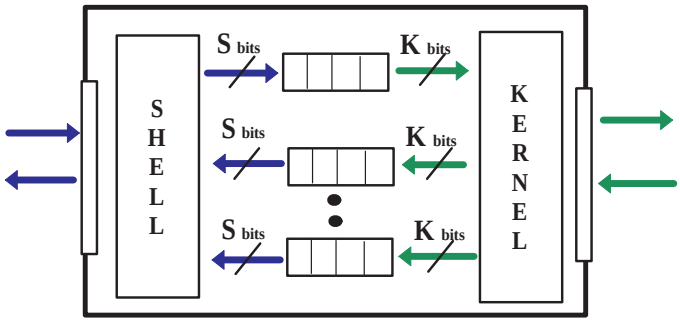


Figure 3.16: High Level view of STNoC™ Network Adapter.

Chapter 4

Single socket and multiple socket NoC based SoCs.

4.1 Network Adapter research.

As already declared in the previous chapters, one of the most important advantages of Network-on-Chip architectures is that they allow the decoupling of computation from communication, hiding interconnect specific implementation details to the IP resources interfaced. The *Network Adapter* is the component the actually takes care of separating the IP cores, where computation takes place, from the network, which provides communication services. Although a great amount of work has been published in literature in the last years on Network-on-Chip architectures, as partially exposed previously, not so many works focus on high-performance *Network Adapters*, which indeed are one of the most critical part of the whole interconnect system. In [21] a NA implementing standard sockets was presented for the Æthereal NoC. The *Network Adapter* for Æthereal supports both BE (best effort) and GS (guaranteed service) transaction streams. They also provide compatibility to more than one bus protocol (AXI, DTL, OCP), thanks to the modular architecture of the *Network Adapter*. The NA architecture is claimed to support system configurations that can be arbitrarily heterogeneous, comprising cores compliant with more that one bus protocol, but is not clear if to do this they have to give up supporting completely the protocols. Anyway, no specific details are disclosed about the techniques used to solve the issue. In [22] an OCP compliant adapter for the Xpipes NoC was described. The adapter has low area but supports only a single outstanding transaction. Both these works describe purely clocked designs. An OCP compliant *Network Adapter* for GALS-based SoC Design has been described in

[11]. This NA enables GALS-type (*Globally Asynchronous Locally Synchronous*) SoCs, in that the NA implements synchronization between the clocked OCP sockets and asynchronous or *clockless* MANGO NoC. The work in [13] addresses the topic of packet reordering on NA the support out-of-order transaction. In particular the work proposes using Gray-code tagging to decrease the overall latency. The paper from Arteris [17] addresses specifically the problem of $Virtual\ Component$ compatibility, closely related to much of the work developed in this thesis. Arteris NoC is claimed to have a transaction layer compatible with AMBA AHB, VCI flavors, AXI and OCP, but no details are given about the low level mechanisms employed to provide this support.

4.2 Multiple socket NoC based SoCs.

Configuring a Network-on-Chip in order to support simultaneously traffic coming from IP cores compliant with different sockets is not simple task. As remarked in the paper from Arteris[17] "*the intertwining of transaction, transport and physical levels within the standard interconnects, and the interconnect flexibility necessary to handle many application designs makes this ideal approach very difficult*". The problem is that the various sockets (IP bus protocols) have many incompatibilities in their basic features that need to be carefully addressed upfront in the NoC transaction layer. Addressing this issues in depth, in order to provide full protocol inter-operability, requires significant overhead in terms of both power and area. Fig. 4.1 shows a NoC based System-on-Chip where all the IP cores are compliant with the same socket (OCP in the figure).

Fig. 4.2 instead shows a system configuration where the IP cores attached comply with different sockets (AHB, OCP, AXI). But as the paper of Arteris and our experience suggest this is very close to be an ideal scenario. Typically the system will end up assuming the configuration depicted in Fig. 4.3.

In the next chapter will be described the work done in the context of the present thesis to add multi-socket support to the standard STMicroelectronics STNoC™ architecture, and in particular to the STNoC™*Network Adapters*. The focus has been put on adding to STNoC™ the simultaneous support of the AMBA AXI protocol and STBus Type 3 protocol. However the techniques and solutions developed can be extended also for other IP sockets. Before, a brief, for confidentiality reasons, description of the standard STNoC™ *Network Adapter* will be given in the next section. A significant part of the standard STNoC™ *Network Adapter*, supporting only the STBus protocol socket, has been developed in the context of this thesis [18].

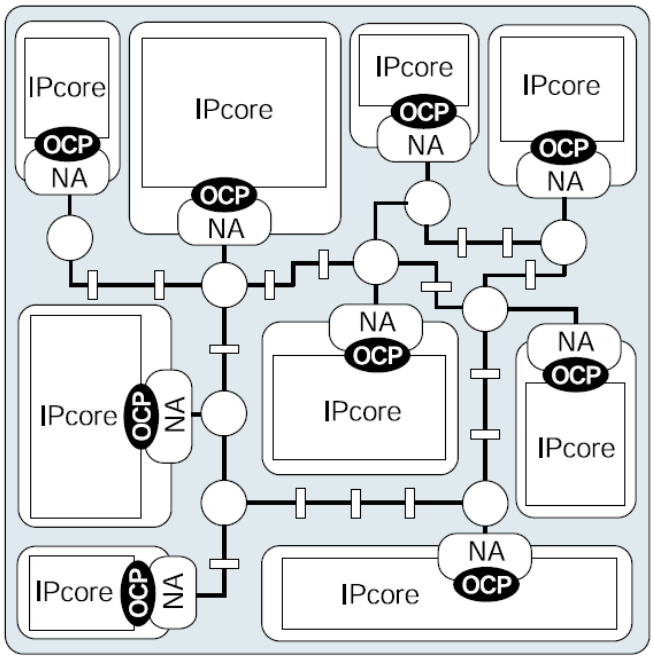


Figure 4.1: System with single socket protocol.

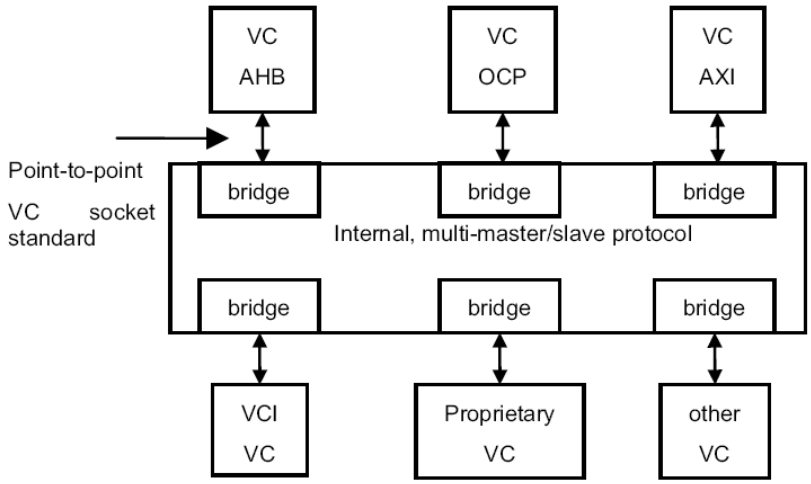


Figure 4.2: Ideal system with multiple socket protocols.

4.3 Standard STNoC™ Network Adapter.

The modular architecture of STNoC™ *Network Adapter* has been already introduced in the previous chapter. The STNoC™ *Network Adapter* architecture is also

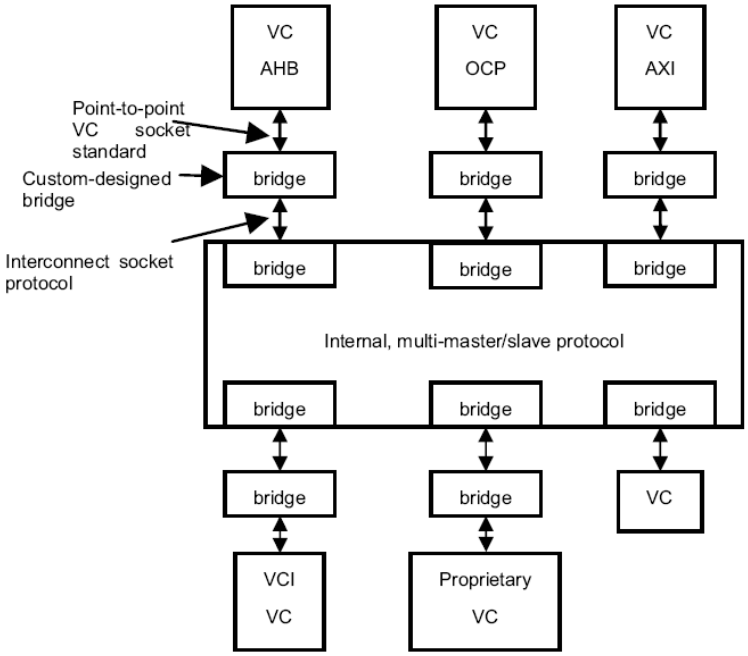


Figure 4.3: Worst case system with multiple socket protocols.

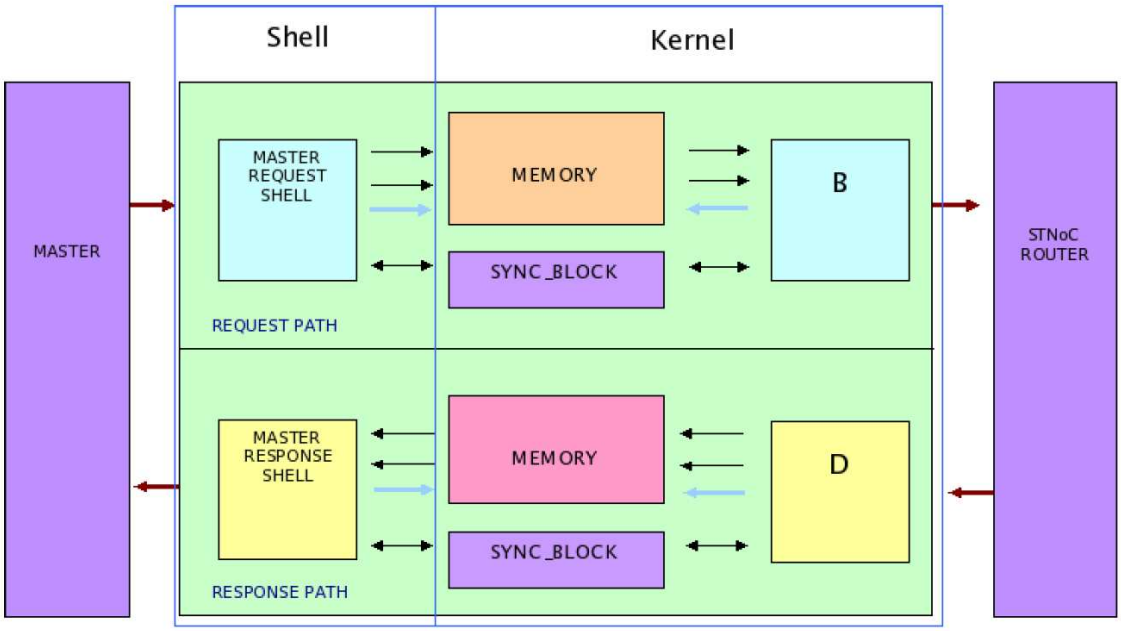


Figure 4.4: High-level representation of standard STNoC™ Network Adapter.

layered, because the *Shell* takes care of the *transport* layer of the OSI stack, while the *Kernel* takes care of the *network* layer of the stack. This layered architecture is reflected also in the structure of the STNoC™ packet, which is used to move data across the network. The detailed representation of the STNoC™ packet cannot be disclosed for confidentiality reasons, but Fig. 4.5 shows its layered nature. At each layer (Transport, Network, etc), information relevant to that layer is split in *header* and *payload*. In general the *header* refers to control information relevant at that particular layer, while *payload* refers to pure *data*. Control information that is relevant at a particular layer can be considered instead as simple data at a different layer, as Fig. 4.5 shows. Take for example the case where *Layer n* in the picture is the *Transport Layer* and *Layer* $(n-1)$ is the *Network Layer*. The only control information relevant at the *Network Layer* is the information that specifies the destination of the packet and how to get there. The *header* at *Transport Layer* can be therefore just considered as *payload* at this layer. This is what is normally referred to as message encapsulation. The STNoC™ *Network Adapter* provides support for both *frequency* and *size* conversion. This is accomplished through a very advanced implementation of the FIFO queues placed inside the *Kernel* block of the *Network Adapter*. It is not possible to disclose more information on the STNoC™ NA for confidentiality reasons.

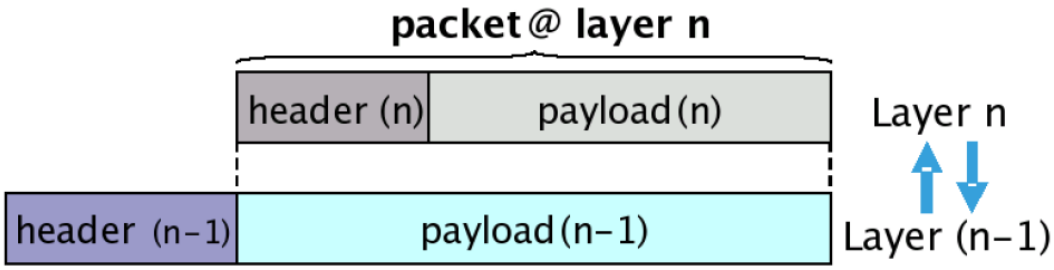


Figure 4.5: STNoC™ packet layering.

Chapter 5

Multi-socket support in STNoC™ architecture

5.1 Introduction

In this chapter will be disclosed the concepts behind the Multi-socket support in the STNoC™ architecture, with special focus on the problem of providing simultaneous supports for the sockets listed hereafter:

- AMBA AXI.
- STBus Type 3.

For confidentiality reasons, the actual hardware implementation of the Multi-socket architecture cannot be disclosed. This is only a first instantiation of the STNoC™ Multi-socket architecture. In the future also support for different sockets (like OCP) will be added. In Chapter 2 a brief introduction of the AMBA AXI and STBus socket interfaces (bus protocols) has already been given. For more information the reader is invited to refer to AMBA AXI[1] and STBus[5] specifications. In the discussion that follows the expression $Network\ Interface$ will be used as synonymous of $Network\ Adapter$. The basic assumption that stands behind the work described hereafter is that to achieve an efficient multi-socket support it is mandatory to provide the necessary *hooks* already at the level of the STNoC™ packet definition. Defining the network packet in a *multi-socket aware* fashion is the only way to avoid putting too much overhead inside the $Network\ Interfaces$ ($Network\ Adapters$). Our approach therefore is twofold, in that it comprises:

- Choosing a format for the Network-On-Chip packet that allows to provide from the ground up the most significant features of the sockets the interconnect architecture is supposed to support.
- Extending the basic architecture of the *Network Interfaces* in order to fill the compatibility gaps that still remain open.

The choice of the Network-On-Chip packet format should rely on the clear identification of a common semantic subset between the bus-protocols for which simultaneous support must be provided. These semantic objects would then be directly assigned to specific fields of the Transport Layer of the NoC packet.

5.2 Comparison between STBus and AMBA AXI protocols

5.2.1 Support of the separate *Read* and *Write* channels of AMBA AXI protocol

One of the distinctive features of the AMBA AXI bus-protocol is the availability of two separate channels for transmitting data, address and handshake information respectively for Load and Store type transactions. This allows to execute in parallel Load and Store transactions, as depicted in Fig. 5.1. This AXI feature is not supported neither by the STBus protocol nor by the STNoC™ Network-On-Chip internal protocol. It is therefore mandatory to provide means for performing a sort of serialization of the two channels outputted by a generic AMBA AXI node before the connection to the ST NoC system can be established. The Load and Store channels should never be simultaneously active. One possible solution is to serialize the Load and Store AXI channels inside the AXI Network Interfaces, using for example a Round-Robin arbiter to manage the allocation of the single output channel either to Load or Store transactions. It is also possible to use more clever algorithms in order to take advantage of the internal architecture of the Network Interfaces and to avoid as much as possible to stop data transmission from the AXI node.

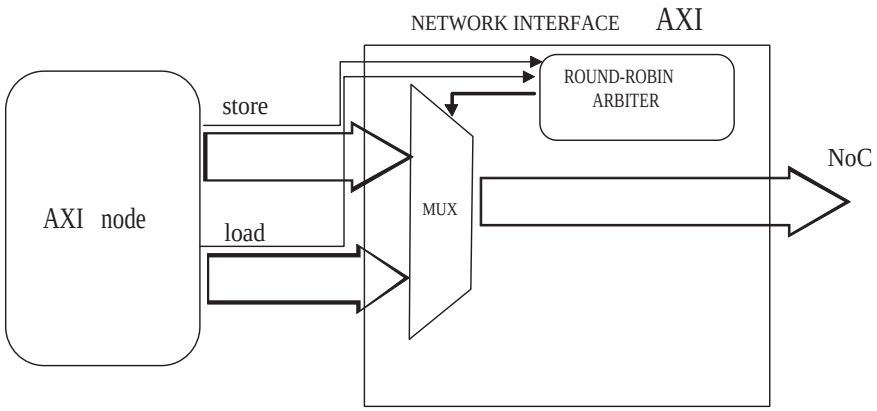


Figure 5.1: AXI Write and Read channel sequentialization.

5.2.2 Differences about the meaning of the *Size* of a transaction

STBus transactions transmit the information about the whole number of bytes the transaction itself is going to transfer. This information is encoded into the size field of the opcode, specifically into bits *opc* <6:4>. This field represents the maximum number of bytes that a single STBus transaction can transfer, 128 bytes. If the size of an STBus transfer is bigger than the size of the databus where the transfer is supposed to take place a multi-cell transaction will be scheduled. If for example the STBus transaction is a store16b and the databus has a width of 32 bits, 4 bytes, the transaction will generate 4 cells of 4 bytes. In AMBA AXI there is not a direct equivalent of the STBus opcode information. However there are two separate transaction parameters, SIZE and LENGTH, that coupled replace the opcode information. The correct names of this signals are $BLength$ <3:0> and $BSize$ <2:0>. The first specifies the number of transfers, beats, that will occur inside a particular transaction while the second specifies the size in bytes of every single transfer. Therefore the overall size of the transaction in the STBus way can be calculated by making the product of the $BLength$ <3:0> and $BSize$ <2:0> fields. Indeed the actual number of bytes that a transaction transfers can be different from that, depending on address alignment issues. The maximum amount of data that can be transferred with a single AXI transaction is larger by far than the maximum amount of data that a single STBus transaction can move. A single AXI transaction can move up to $\max\{BurstSize\} \times \max\{BurstLength\} \doteq 2048$ bytes , while a single STBus transaction can move no more than 128 bytes.

5.2.3 Addressing Modes

The AMBA AXI protocol provides a *Burst* <1:0> signal that encodes the type, i.e. the addressing mode, of the transaction. The protocol supports three different addressing modes: incrementing, wrapped and streaming. With the incrementing addressing mode the address of a transfer is calculated incrementing the previous one of a fixed number of bytes, the width of the single transfer. In wrapped mode the addresses are calculated as in the previous case until a fixed threshold is reached. Above this threshold the address wraps down to a specific starting point. The threshold value equals the product of the burst width and the burst length, and therefore the wrapping boundary for AXI transaction is the same used in STBus transactions to decide if adresses of unaligned transfers must be wrapped or not. In the last addressing mode, the streaming one, the value of the address for the different transfers of one transaction remains constant, pointing to a fixed memory location. Usually this addressing mode is used when accessing peripherals (FIFOs). On the other hand STBus provides only one default addressing mode that maps exactly into the incrementing/wrapping modes of AXI protocol. Indeed the wrapping behavior is hidden to the programmer. It is performed implicitly by the appropriate STBus interconnect components. AXI bursts that use the wrapped addressing mode are required to be aligned with the value of the AXI BSize signal and their length can be only of a power of two, ranging from 2 to 16. Providing support to the full spectrum of AMBA AXI addressing modes requires allocating 2 bits of information inside the STNoC™ packet format. Fig. 5.2 summarizes the relationship between AXI and STBus addressing modes.

5.2.4 Choice of the primitive operations to support inside the network.

The AMBA AXI protocol lacks an explicit opcode field as the one STBus and other bus-protocols have. This is because the distinction between Load and Store transactions is implicit for the presence of two fully parallel communication channels. The STBus protocol not only supports Load and Store transactions but also transactions as RMW and SWAP among others. Every different transaction maps to a particular configuration of STBus opcode field. An important property of the STBus protocol is that the behaviour of transactions like RMW and SWAP can be exactly reproduced using chunks of Load and Store transactions, where chunks are packets linked using a special signal named the lock signal. This property enables the Network-On-Chip to offer primitive support only to Load and Store type transactions. This

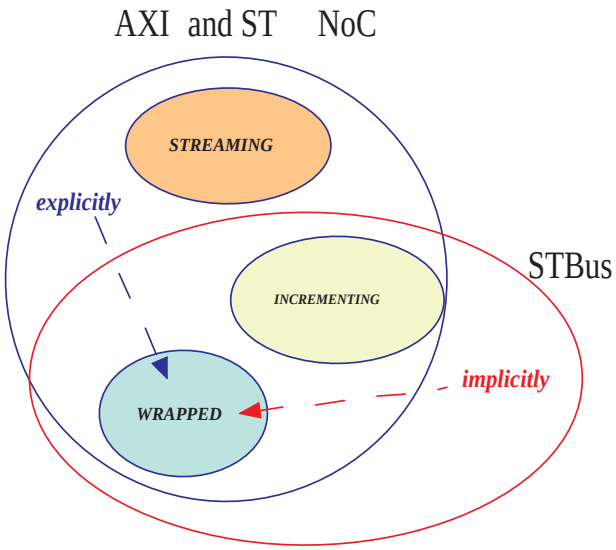


Figure 5.2: Comparison of available addressing mode options.

choice allows to allocate only 1 bit of the STNoC™ packet header for specifying the atomic operation. The conversion of the RMW or Swap transactions into chunks of Load and Store transactions would be in this case performed inside the Network Interfaces attached to an STBus Master Node.

5.2.5 Proposed format of the Transport Layer Opcode of the STNoC™ packet header.

From the discussions above a first proposal for the format of the Transport Layer section of the NoC packet can be outlined. The main fields inside the header's opcode will be allocated accordingly to what follows:

- 1 bit for Store/Read.
- 2 bits of Address Descriptor.
- 3 bits of SIZE.
- 4 bits of LENGTH.

The main choice that has been made is to keep all the features offered by the AMBA AXI protocol inside the STNoC™ packet, offering a direct mapping for the

most significant AXI signals,i.e. BurstType, BurstSize and BurstLength. This solution allows to simplify the support for the majority of traffic conditions that can occur inside the AXI aware STNoC™ system.

5.2.6 The response signals.

The most important signals on the AXI response path are RID <3:0> and $RRESP$ <1:0>. The first specifies the transaction ID. Its value should match the ID transmitted on the request phase. The second is used to send feedback information to the Initiator about the success of the transaction. The $RRESP$ <1:0> signal has 4 possible values that map to the 4 possible outcomes a transaction can have:

BRESP[1:0]	RESPONSE
b00	OKAY
b01	EXOKAY
b10	SLVERR
b11	DECERR

Figure 5.3: Interpretation of AXI response opcode.

The STBus protocol uses only 1 bit to transmit feedback information about the success of the transaction (r_opc[0]) :

It is therefore mandatory to establish a mapping between the feedback signals in AXI and those in STBus. The most obvious one is represented in Fig. 5.5.

At the level of STNoC™ packet format definition the preferred choice is again to keep the full AXI semantics, using 2 bits of the response opcode to transmit the feedback information, as shown in the Fig. 5.6:

r_opc[0]	RESPONSE
b0	OKAY
b1	ERROR

Figure 5.4: Interpretation of STBus response opcode.

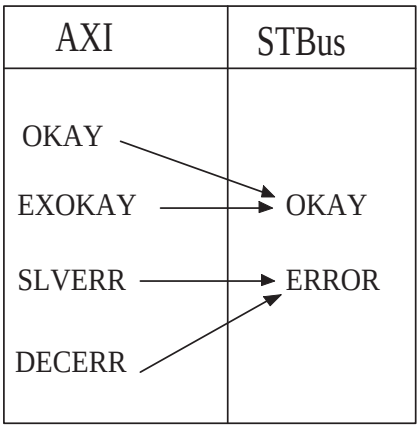


Figure 5.5: Proposed mapping between AXI and STBus response opcode values.

5.2.7 Multi-socket STNoc™ packet.

With the assumptions made above the format of the overall STNoC™ packet varies slightly in respect of its standard form (single protocol). Figures 5.7, 5.8 and 5.9 show respectively the high level format of request and response packet and the detailed configuration of the *header* at *Transport Layer*.

5.2.8 Interruptibility of STBus and AXI transactions.

AXI transactions cannot be interrupted, even if errors occur. Any transaction characterized by a given BSize and BLength parameters must absolutely complete the number of transfers the the transaction specifies. No matter if some of them are meaningless, as is the case in transfers having the write strobes all cleared. Also STBus transactions cannot be interrupted. Only STBus messages can be optionally

Figure 5.6: Proposed STNoC Response Packet Opcode.

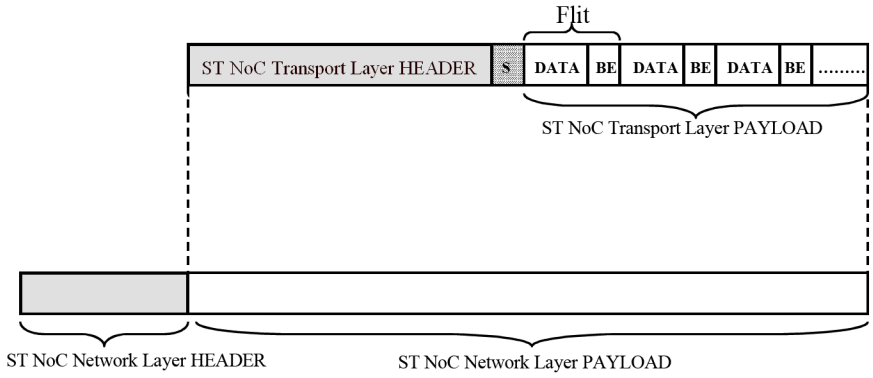


Figure 5.7: STNoC request packet.

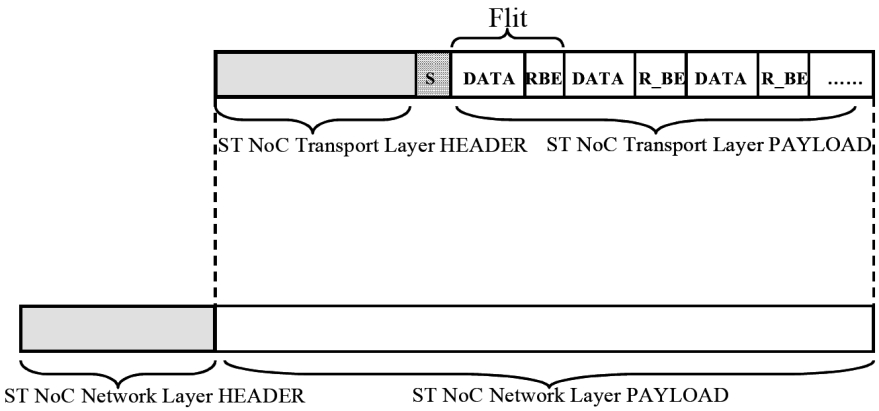


Figure 5.8: STNoC response packet.

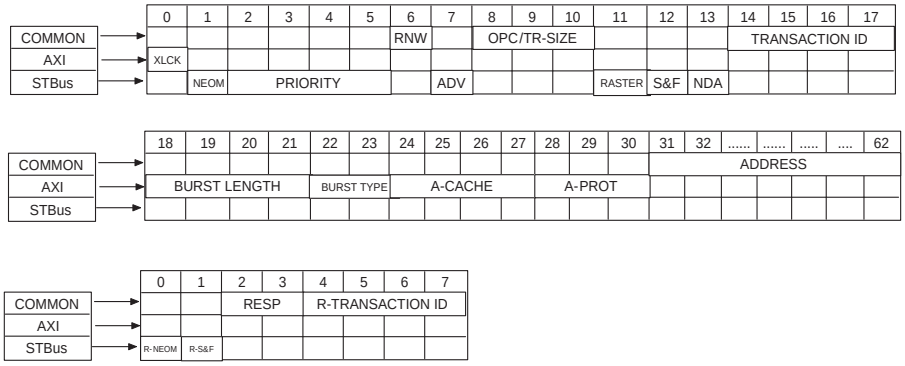


Figure 5.9: Header at transport layer in request and response packets.

interrupted, depending on how the interconnect is configured.

5.3 Main hurdles towards compatibility

From the concepts described above two aspects emerge as posing the major hurdles towards the achievement of full compatibility. The first is represented by the different way unaligned addresses are handled in the two protocols. The second is the wide number of data quantities that a single AXI transaction can transfer, compared to the relatively small number of choices offered by the STBus protocol. In particular the number of bytes that a single STBus transaction can transfer is only a small subset of the amounts an AMBA 3 AXI transaction can transfer. This two aspects will be further investigated in the following paragraphs.

5.3.1 Unaligned Addresses.

STBus Masters can only generate transactions aligned with their databus size. The protocol provides support only for transactions aligned with the target node, but in real systems sometimes this rule is violated, as when some types of size conversions must be taken into account. In those cases the interconnect system must be able to take appropriate actions. These actions consist in transparently performing a wrapping of the transaction's address, in order to avoid sending addresses not aligned with the size of the target. This wrapping process can be also performed inside some STBus interconnect IPs as bridges or interfaces. The way AMBA AXI systems handle unaligned accesses depends on the type of the transaction at hand. For incrementing transactions AMBA AXI provides full support for unaligned trans-

actions while for wrapping transactions the starting addresses must be aligned with the size parameter of the transaction itself, that is not the whole transaction size as intended in STBus terms. Wrapping AXI transactions must observe also an additional constraint on the length of the transfers, which must be a power of 2 value comprised between 2 and 16. If no explicit actions are taken the traffic generated by nodes that observe different bus protocols can not be integrated. The place where these corrective actions must be taken is inside some of the Network Interfaces of the STNoC™ system. The architectural extensions of the baseline STNoC™ architecture will enable support for 4 different traffic conditions, listed hereafter:

- Transactions generated by AMBA AXI Masters towards AMBA AXI slaves.
- Transactions generated by AMBA AXI Masters towards STBus slaves.
- Transactions generated by STBus Masters towards AMBA AXI slaves.
- Transactions generated by STBus Masters towards STBus slaves.

Transactions generated by STBus Masters.

The means provided to support unaligned transactions generated by STBus nodes (Fig. 5.10) consist in flagging these transactions as wrapping at the level of the STBus Master Network Interface, setting the corresponding field inside the Opcode of the STNoC™ packet Header. If the destination of the packet is another STBus node this action will produce no side effects, because the wrapping flag will be ignored by any STBus slave. The overall behavior will be anyway consistent with the meaning of the original transaction. If on the other hand, the destination is an AMBA AXI node, marking the incoming packet as of wrapping type would produce some desired effects. The address will in fact wrap around a threshold aligned with the quantity SIZExLENGTH. The transaction will therefore exhibit a behavior consistent with the semantics of the original STBus transaction, even if the Slave is actually an AXI node. Sometimes the only action of forcing a wrapping behavior for the addresses is not enough and some other actions are needed, like setting appropriately the values for the SIZE and LENGTH parameters.

Transactions generated by AMBA 3 AXI Masters.

Providing full support for transactions generated by AMBA AXI masters (Fig. 5.11) can be much more difficult. In case of unaligned addresses the behavior of the STBus

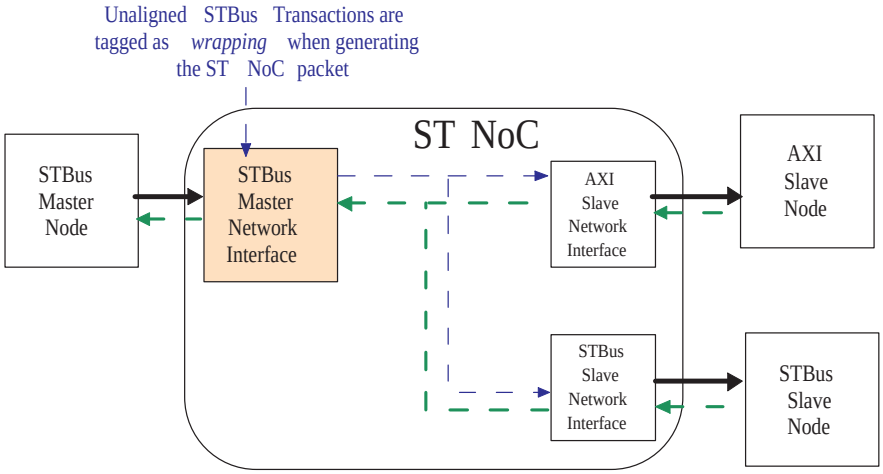


Figure 5.10: Propagation of packets generated by STBus Masters.

node would be to force the wrapping of some of the addresses, but this would in turn violate the original AXI semantics. It is therefore unavoidable to take some action at the level of the STBus Slave Network Interface. The solution proposed hereafter is to split the unaligned AXI transaction, that arrives to the slave as an unaligned STNoC™ packet, into 2 or more, depending on the size of the payload, STBus transactions, as shown in FIg. 5.12. To determine the exact number of mapping transactions that the Network Interface has to generate it is important to take into account not only the starting address but also the ratio between the databus sizes of Master and Slave. When transaction splitting is performed in the request path the same Network Interface has to provide means to perform the fusion of all the responses that the slave will generate into a single response packet. When the Slave is an AXI node instead there is the need to check if size conversion needs to be performed and if so, depending on the type of transaction, can again be required to perform transaction splitting and/or address manipulation.

5.3.2 AXI transaction sizes not directly supported by STBus.

The whole amount of bytes transferred by an AMBA AXI transaction is given by the product of the BurstSize and BurstLength fields. Actually, the number of bytes transferred can be smaller, in case of address unalignment. The BurstLength field can assume all the integer values comprised between 1 to 16. The value of the

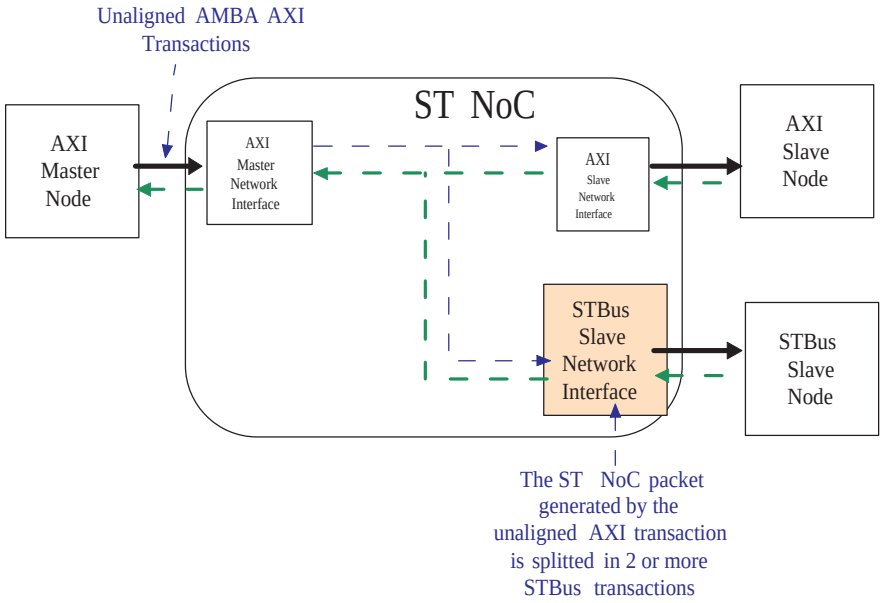


Figure 5.11: Propagation of packets generated by AXI Masters.

BurstSize field can be a power of two comprised between 1 and 128. The size of an STBus transaction is, as already said, specified by the value of opcode field. The sizes allowed are the power of two values between 1 and 128. Therefore the overall transaction size of STBus transactions maps exactly to the values of the SIZE parameter of AMBA AXI transactions. This implies that for example an AMBA AXI transaction with BurstSize=4 and BurstLength=5 can not be mapped directly to a single STBus transaction.

5.3.3 ID information.

Both the STBus Type 3 interface and the STNoC™ packet have a total amount of 12 bits dedicated to convey ID information. These bits are spread across 2 fields, *src* <7:0> and *tid* <3:0>. The src field is used to specify the Node that has generated the transaction while the tid field is used to identify different transactions generated by the same Node. The meaning of the src field depends on the actual configuration of the Initiator Node, i.e. if the Initiator is a single Node or if it is a Subsystem. In the first case the src field is just an identifier of the Node while in the second case part of the field is used to identify the Node/Interface that acts as Initiator towards the interconnect and the remaining bits are used to specify

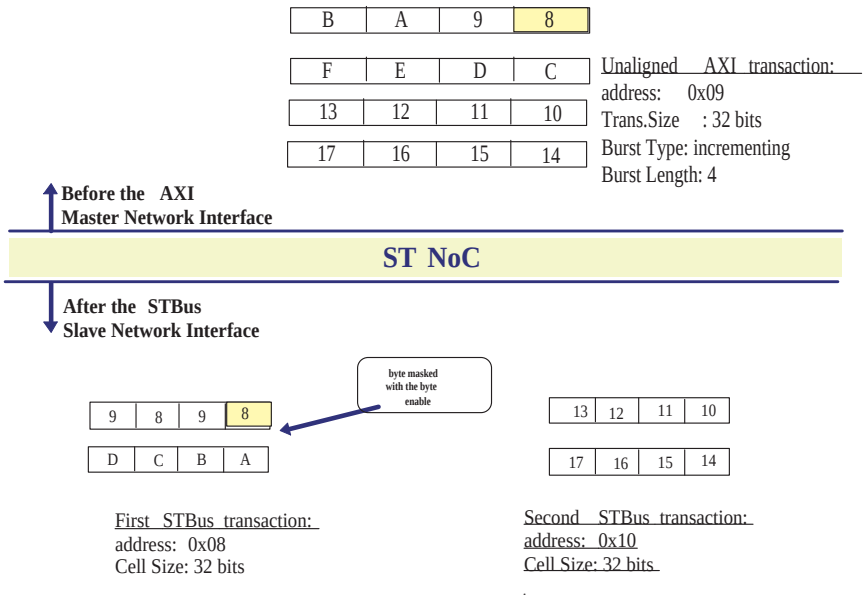


Figure 5.12: Packet splitting in AXI-to-STBus traffic scenarios.

which node inside the subsystems generated the transaction in the first place. In the AMBA AXI protocol things are a little bit different. The AXI Master issues only 4 bits of ID information inside the fields xID, where x stands for W or R, respectively in case of write and read transactions. This information is provided also in the address channel, enabling data interleaving in the Request Path. The meaning of this field is the same of the *tid* <3:0> field of STBus and STNoC™. The AXI interconnect, whenever an AXI Node issues a new transaction, appends a few ID bits to those directly generated by the initiator. The role of these new bits is to allow the correct dispatch of the transaction from the Master to the Slave Node. This implies that the number of ID bits that the Slave Interface sees is actually bigger than the number handled by the Master. To ensure interoperability of the STBus and AXI protocols inside the multi-socket system the AXI Master Network Interface needs to generate the ID information not provided by the AXI Master Node but required by the STNoC™ interconnect system to allow the correct dispatch of the transaction to the destination Node. That is because the AXI Node can not relay on an external arbiter that does the job in a standard AXI based interconnect. The bits the AXI Master Network Interface has to generate correspond to those of the *src* <7:0> field of STBus. Part of these bits will be "Interface Dependent" and the remaining will be "Subsystem Dependent", as shown in Fig. 5.16. The number of bits that will be of one type or of the other depends on the actual system

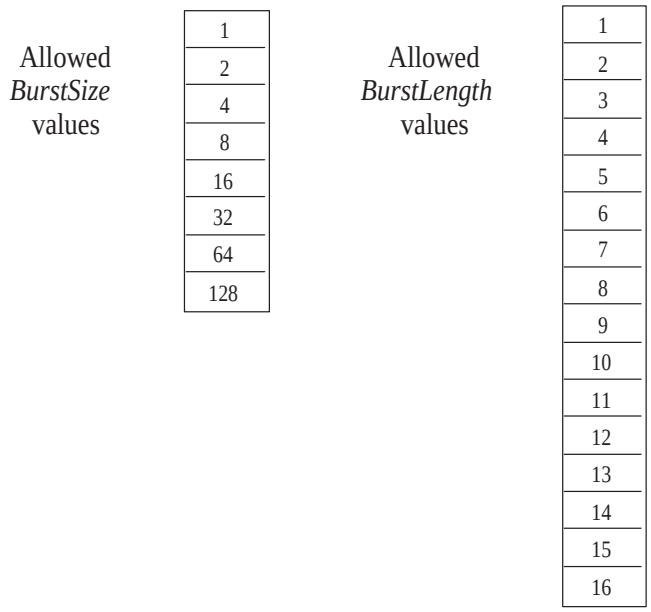


Figure 5.13: Possible values for AXI SIZE and LENGTH parameters.

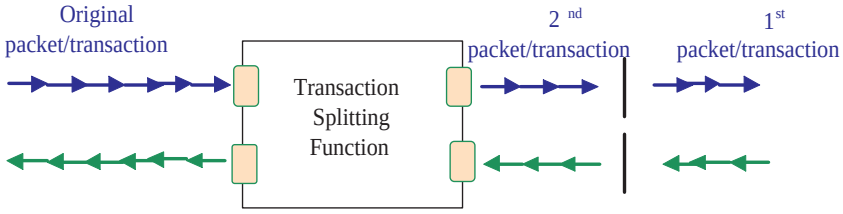


Figure 5.14: Splitting and reconstruction of incoming packets.

configuration, but their sum will never be more than 8, the width of the src field. For transactions generated from STBus Masters the SOURCE ID field is simply copied from the *src* <7:0> field of the incoming STBus transaction.

5.3.4 Complexity of the AMBA AXI-STBus mapping problem.

For what said above, the Slave Network Interface needs to map the incoming packet, representation of an AXI transaction, into one or more STBus transactions. The

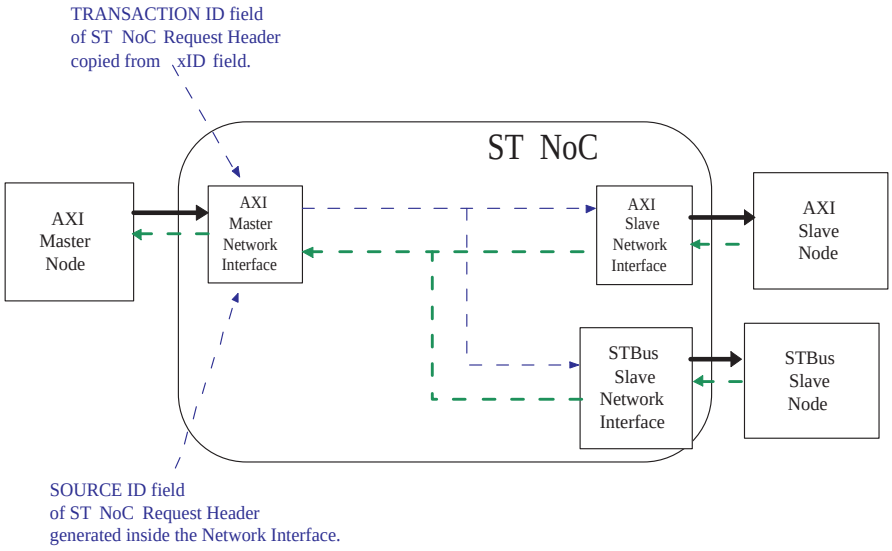


Figure 5.15: Generation of STNoC™ ID information in case of AXI Masters.

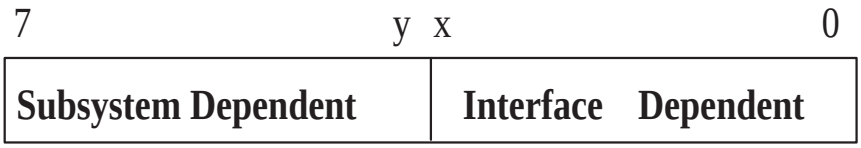


Figure 5.16: Structure of the SOURCE ID field of STNoC™ request header.

parameters that need to be chosen are the proper command (opcode) of the transaction/s, the byte enables and the final address of the outgoing STBus transaction. It is not possible to find a direct one-to-one mapping between all AMBA AXI and STBus type 3 transactions. If the incoming packet for example moves a total number of 12 bytes of data there is not any single STBus transactions that can do the job alone. In this case instead the Slave STBus Network Interface can take the decision to generate an 8 byte transaction followed by one of 4 bytes or otherwise can use 3 transactions of 4 bytes each and, just to name a few options. Furthermore the choice of the proper mapping can be influenced by other features of the incoming packet and by some other architectural features of the Slave Network Interface itself, as represented in Fig. 5.17. If for example the depth of the buffers used inside the Network Interface is not enough to store all the mapping sequence for a given input packet a good solution could be to avoid issuing STBus transactions which transfer large numbers of bytes, because the buffer overflow that would occur would require

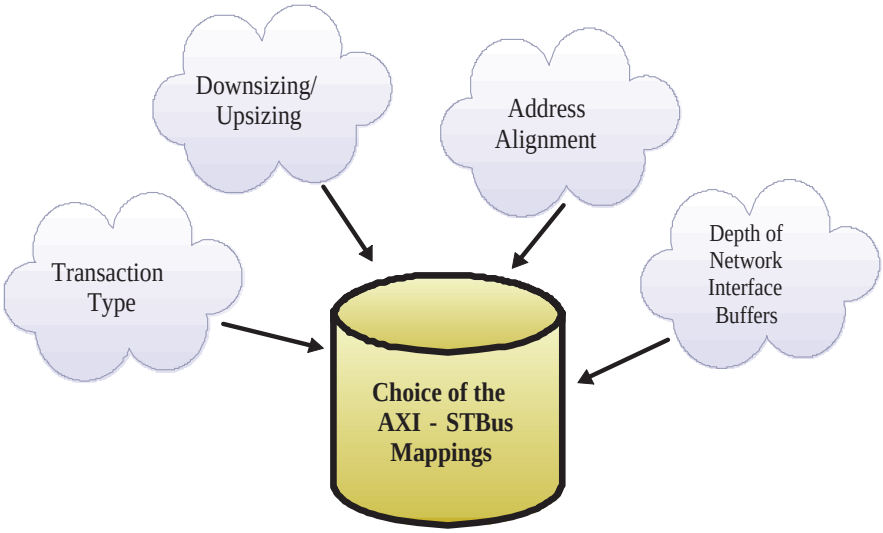


Figure 5.17: Process of Mapping Selection.

stalling the Network Interface, adding significant latency to the transaction. There are three possible actions that can be taken at the level of the STBus Slave Network Interface to perform a correct mapping between the incoming STNoC™ packet shape and the STBus transaction/s to transmit towards the Slave. These are *Transaction Splitting*, *Address Adaptation* and *Data Adaptation*. These actions can be deployed not only separately but also in combination to solve the mapping problem in critical corner cases. The need to perform Transaction Splitting can arise in a number of cases: STNoC™ packet sizes not supported by STBus, address alignment problem that requires the splitting of the transaction, size conversion. In each of the preceding conditions also Address Adaptation and Data Adaptation can be optionally required. There are times where Transaction Splitting is not required but one or both of Address Adaptation and Data Adaptation are. This occurs for example when the whole size in bytes of the packet would have a direct mapping in one STBus transaction but for example the packet address is unaligned in respect of the Slave Node databus size.

5.4 Mapping algorithms for AMBA AXI and STBus traffic integration.

5.4.1 AXI-to-STBus traffic.

AXI incrementing transactions.

The algorithm that handles AXI transactions of type incrementing basically tries to approximate the value of the entire packet size by means of the basic bricks represented by the available STBus transactions. In particular it checks at each step if it is better to use a transaction slightly smaller or bigger in size of the data amount that remains to be transferred. This check allows to minimize the total number of transactions needed to map a given packet (see Algorithm 1 for details).

Algorithm 1: AXI-to-STBus incrementing transactions

Input: PS =Packet Size (SIZE×LENGTH) ;
IA = Starting Address of incoming packet ;
T_{stbus} = Set of all STBus transactions ;
T_{SA_i} = Set of all STBus transactions aligned with SA_i;
W_i = Number of bytes actually transferred with the i^{th} transaction;
R_i = Number of bytes that remain to be transferred after the $(i-)^{th}$ transaction;
in_SIZE = Value of the size parameter of the incoming packet ;
in_LENGTH = Value of the length parameter of the incoming packet ;
in_ADD = Address of the incoming packet ;
$databus$ = Value of the size parameter of the incoming packet ;
Output: $T_0 \ldots\ldots T_N$ = STBus transactions of the mapping sequence ;
SA_i = Starting Address of mapping transactions ;
B = Vector of all byte enables for the current packet ;

begin

$x \longleftarrow IA \mod PS$;
$SA_0 \longleftarrow IA - x$;
$index \longleftarrow 0$;
$res \longleftarrow PS$;
while $res \geq 0$ **do**
$T_{down} \longleftarrow \max\{T \in T_{SA_{index}} : res \geq T\}$;
$D_{down} \longleftarrow res - T_{down}$;
$T_{up} \longleftarrow \min\{T \in T_{SA_{index}} : T \leq res\}$;
$D_{up} \longleftarrow T_{up} - res$;
if $D_{down} \geq D_{up}$ **then**
$T_{index} \longleftarrow T_{up}$;
else
$T_{index} \longleftarrow T_{down}$;
$W_{index} \longleftarrow T_{index} - x$;
$R_{index} \longleftarrow PS - \sum_{k=0}^{index} W_k$;
$SA_{index+1} \longleftarrow SA_{index} + T_{index}$;
$res \longleftarrow R_{index}$;
$index \longleftarrow index + 1$;
$B \longleftarrow [0, 0, \ldots, 0_{x-1}, 1, 1, \ldots, 1_{PS+x}, 0, 0, \ldots, 0_{\sum_{k=0}^{index-1} T_k}]$;

end

AXI wrapped transactions

Wrapping packets produced by AXI Masters which transfer a total number of bytes (SIZExLENGTH) that matches one of the STBus opcode values and whose initial address is aligned with the Slave databus size are just propagated to the STBus Slave, because the natural wrapping behavior of STBus interconnect will reproduce the desired wrapping semantics. Otherwise there is the need to perform a more detailed analysis. The number of different initial addresses allowed for a given wrapping AXI transaction (fixed SIZE and LENGTH parameters) equals the value of LENGTH. But as the SIZE parameter varies, the magnitude of each address interval varies at the same pace. For the constraints that the AXI specification imposes on the initial addresses of wrapped transactions, if we define a quantity named `sig` (signature) as: $sig = IA \mod PS$ we have that this quantity is invariant with respect to the SIZE parameter. To take into account the variation of the SIZE parameter (which does have an impact on the choice of the mapping sequence) a second quantity can be defined, `key`, as: $key = \frac{PS}{sig}$. For every possible AXI wrapping transaction (any combination of allowed address, SIZE and LENGTH parameters) there are only 16 possible values of `key`. Each one of these values corresponds to a different parametric sequence of STBus transactions that solves the mapping problem. Actually the number of independent `key` values can be further reduced to 9. The 9 parametric sequences of STBus transactions are listed in Fig. 5.18, along with their appropriate `key` value. The length of the parametric sequences can go from 1 to 5 STBus transactions. Every possible AXI wrapping transaction maps to one of the nine `key` values and therefore to a unique STBus mapping sequence (see Algorithm 2 and Fig. 5.18).

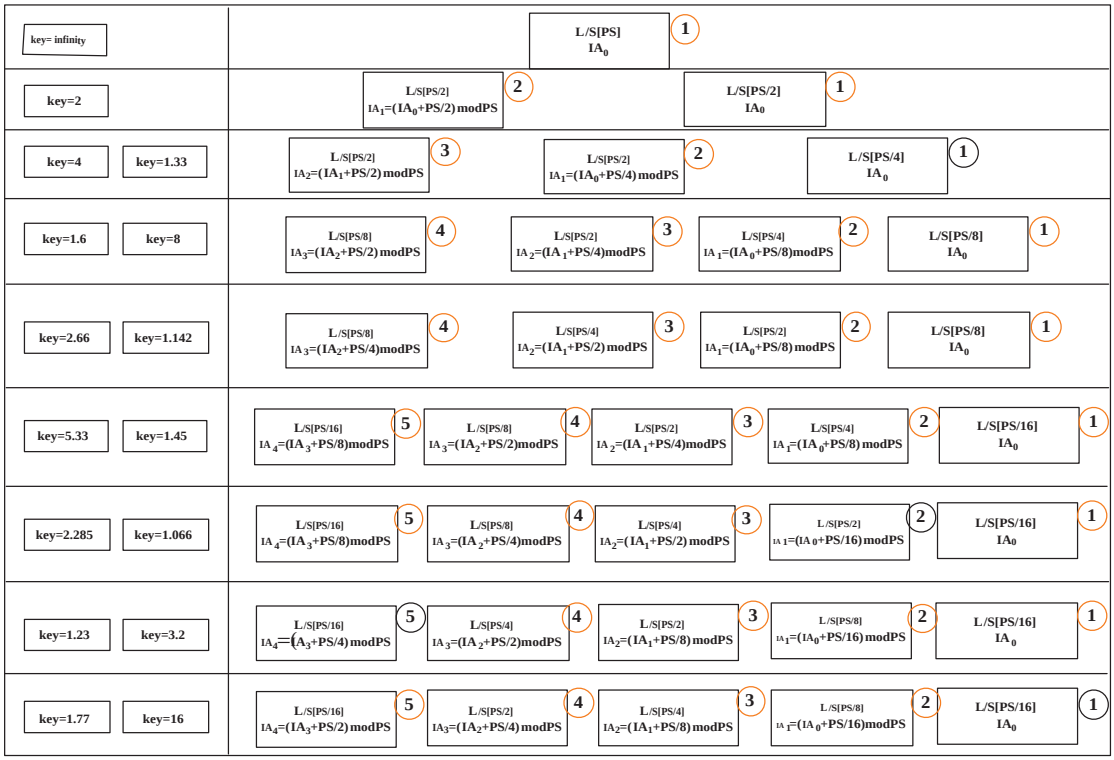


Figure 5.18: Kernel of AXI-to-STBus wrapped algorithm

Algorithm 2: Prologue of AXI-to-STBus wrapped algorithm

Input: IA =Initial Address of incoming packet ;
SS = Slave Databus size ;
in_SIZE = SIZE of incoming packet ;
in_LENGTH = LENGTH of incoming packet ;
Output: $out\_TRANSACT$ = Opcode of current STBus transaction emitted;
out_ADDR = Address of STBus transactions emitted ;

begin
  if $(is\_STBus(in\_SIZE \times in\_LENGTH) \&\& (IA \mod SS == 0))$ **then**
    $out\_ADDR \longleftarrow IA$;
    $out\_TRANSACT \longleftarrow in\_SIZE \times in\_LENGTH$;
  else
    GOTO the *kernel* part of the algorithm ;
end

AXI fixed transactions.

Every single transfer of an AXI fixed transaction that arrives to the STBus Slave Network Interface needs to be considered as an independent transaction. Therefore if one or more of these atomic transfers requires adaptation because of alignment or size conversion problems the standard algorithms for handling incrementing transactions will be applied.

5.4.2 AXI-to-AXI traffic.

The algorithm for mapping AXI-to-AXI traffic across the AXI aware STNoC™ is the same for incrementing and wrapping transactions. It takes also into account both downsize as well as upsize conversion ratios between the Master and Slave nodes. Indeed the mapping strategy employed in case of upsize conversion ratios comes out to be a special case of the one that provides support to downsize conversion ratios. The behavior of the AXI Slave Network Interface depends on two different features of the incoming transaction. The first is the ratio between the SIZE parameter of the incoming ST NoC packet and the size of the databus at the Slave interface side. The second is the value of the LENGTH parameter. If the SIZE parameter is equal or smaller than the databus the properties (LENGTH, SIZE, TYPE) of the transaction pass through unchanged across the Network Interface and propagate to the slave Node, that is the SIZE and LENGTH parameters as well as the address of the AXI transaction the Network Interface issues to the slave Node are the same of the incoming ST NoC packet. On the other hand, if the SIZE parameter of the incoming STNoC™ packet is bigger than the databus attached to the slave, and therefore can not be just copied, the SIZE parameter of the outgoing AXI transaction is fixed to the width of the databus. Since the overall size, in bytes transferred, of the AXI transaction must be kept constant, if we reduce the SIZE parameter we must increment in the same proportion the LENGTH parameter. So for example if we have an incoming packet SIZE of 8 and the databus is 4 bytes wide we are reducing the SIZE by a factor of 2 and we need to enlarge the LENGTH by the same factor. This can be done without any problem until the final value that we get for the LENGTH parameter is no more than 16. In case this condition is not met we are forced to perform some splitting on the incoming transaction, because 16 is the maximum available value for LENGTH. The fact that upsize conversion is included in the treatment of downsize conversion depends on the fact that when there is an upsize ratio between the Master and the slave, Slave bigger, the condition of having an incoming SIZE parameter bigger than the databus can never occur and then we

fall in the first of the scenarios described above.

AXI-to-AXI fixed transactions.

As usual for the handling of AXI fixed transactions traffic each fixed burst transfer is considered by the AXI slave Network Interface as an independent transaction. If no size conversions applies to the transfer the properties of the transfers will not vary. If instead size conversion needs to be accomplished the techniques developed for AXI-to-AXI incrementing traffic will be applied to emulate each single original transfer.

Additional considerations on AXI-to-AXI traffic support.

For the AXI-to-AXI mapping algorithm to work properly at least the AXI Master Network Interface needs to be aware of the databus size of the slave node that is sending data. This requirement can be satisfied providing every Master Network Interface with a small lookup table where for each possible destination node the size of its databus is reported. Knowing the exact size of the slave node is required because the STNoC™ response path does not provide byte strobe signal lines and slaves with different databus width transmit a different payload despite the fact that the actual transaction is the same. The ARM official documentation on AMBA AXI interconnect systems as well as documentation on Synopsys Design Ware model for AXI interconnect modules describes only system configurations where only downsize conversion ratios between Masters and Slaves are allowed. Upsize conversion seems to be never required. The mapping algorithm described above supports also scenarios where upsize conversion do occur. In those cases the mapping policy is just borrowed from downsizing scenarios. One possible reason that justifies the choice to avoid recurring to upsize conversion in AXI interconnect systems is that whenever upsize conversion occurs also in our AXI aware STNoC™ systems the best possible databus usage for the slave databus is reduced to 50%. This is because the smallest size ratio that we can have is 2, and the mapping algorithm in case of upsize conversion prescribes to keep the AXI transaction properties (i.e. SIZE,LENGTH and initial address) unchanged from Master to Slave.

5.4.3 STBus-to-AXI and STBus-to-STBus traffic.

Support for these two traffic types requires no special actions.

Algorithm 3: AXI-to-AXI transactions

Input: in_SIZE = Size of the incoming packet ;
in_LENGTH = Length of the incoming packet ;
in_ADD = Address of incoming transactions ;
$databus$ = Width of the slave databus ;
Output: out_SIZE = Size of the outgoing transaction ;
out_LENGTH = Length of the outgoing transaction ;
out_ADD = Address of the outgoing transaction ;
T_i = Generated AXI transactions;

begin

end

$ratio \longleftarrow \frac{in\_SIZE}{databus}$;
if $ratio \leq 1$ **then**
  $out\_SIZE \longleftarrow in\_SIZE$;
  $out\_LENGTH \longleftarrow in\_LENGTH$;
  $out\_ADD \longleftarrow in\_ADD$;
else
  $out\_SIZE \longleftarrow databus$;
  $tmp \longleftarrow ratio \times in\_LENGTH$;
  if $tmp > 16$ **then**
    if $incrementing$ **then**
      $\{T_0, T_1, \ldots, T_{ratio-1}\} \longleftarrow SPLIT(input\ trans.)$;
      $T_{-1} \longleftarrow 0$;
      for $i \longleftarrow 0$ **to** $ratio - 1$ **do**
        $out\_LENGTH_i \longleftarrow in\_LENGTH$;
        $out\_ADD_i \longleftarrow in\_ADD + \sum_{k=-1}^{i-1} T_k$;
        $out\_BTYPE \longleftarrow incrementing$;
    else
      if $(wrapping \&\& 16\_aligned)$ **then**
        $\{T_0, T_1, \ldots, T_{ratio-1}\} \longleftarrow SPLIT(input\ trans.)$;
        $T_{-1} \longleftarrow 0$;
        for $i \longleftarrow 0$ **to** $ratio - 1$ **do**
          $out\_LENGTH_i \longleftarrow 16$;
          $out\_ADD_i \longleftarrow \left(in\_ADD + \sum_{k=-1}^{i-1} T_k\right) \bmod (in\_LENGTH \times in\_SIZE)$;
          $out\_BTYPE \longleftarrow incrementing$;
      else
        $\{T_0, T_1, \ldots, T_{ratio}\} \longleftarrow SPLIT(incoming\ transaction)$;
        $T_{-1} \longleftarrow 0$;
        for $i \longleftarrow 0$ **to** $ratio - 1$ **do**
          $out\_LENGTH_i \longleftarrow 16$;
          $out\_ADD_i \longleftarrow \left(in\_ADD + \sum_{k=-1}^{i-1} T_k\right) \bmod (in\_LENGTH \times in\_SIZE)$;
          $out\_BTYPE \longleftarrow incrementing$;
        $out\_ADD_{ratio} = (in\_ADD - (in\_ADD \bmod (0x10))$;
  else
    $out\_LENGTH \longleftarrow temp$;

	area	max. freq
AXI MS	0.067843 mm^2	856 Mhz
AXI SL	0.074532 mm^2	797 Mhz
STBus MS	0.058823 mm^2	888 Mhz
STBus SL	0.105642 mm^2	678 Mhz

Figure 5.19: Synthesis data.

5.5 Implementation, results and future work.

A parametric VHDL model for the various *Network Adapter* encompassed by the preceding architectural specification has been implemented. For confidentiality reasons, neither the precise microarchitecture at the RTL level nor the actual VHDL source code can be disclosed. A number of synthesis runs have been conducted on simplified models of the *Network Adapter* in order to estimate the performance of the components. The simplifying assumptions include a full synchronous design, while the complete multi-socket NA architecture supports completely unrelated operation frequencies at the IP and network side. Moreover, traffic across the network is considered to be address aligned. This simplifies some of the mapping algorithms. The synthesis runs have been done with Synopsys Design Compiler and with the Virtual Silicon UMC 0.13 (normal conditions) library. Fig. 5.19 shows the synthesis data, area and timing, respectively for AXI Master NA, AXI Slave NA, STBus Master NA and STBus Slave NA.

The data shows as expected that the most challenging implementation is that of the STBus Slave NA which, as explained earlier in the chapter is in charge of performing the complex transaction splitting task. The need to support the complex State Machines required justifies the degradation in performance in respect of the other NAs. Implementation of the mapping algorithms in the most complex traffic scenarios needs to be further optimized. Future work will investigate the possible extension of the multi-socket support to other socket protocols (OCP). Furthermore techniques for the reduction of burstness of the traffic along the links will be investigated. Also a further optimized implementation of the Asynchronius FIFOs used inside the NA will be considered.

Part II

SoC computation architectures:
Extension of ST230 architecture to binary compatibility with ARM ISA.

Chapter 6

Computation paradigms for Systems-on-Chip.

6.1 From the Embedded and General Purpose computing paradigms to the single System-on-Chip computing domain

In the last years the scenario related to the embedded computation platforms has traversed deep changes. Actually to some years ago the aspects that assumed great relief were certainly those legacies to the choice of the components to integrate and to the scrupulous job of debugging general of the systems. Good part of the components were realized custom to be integrated inside a particular system and every change or evolution it often asked for times of at tended very long and an intense job of development. The field of embedded systems was completely separated by that of the personal computing, for which the exchange of technologies among the two fields was not very intense. Today embedded systems are definitely more "processor centric" and much more similar to the platforms employed in general purpose computing. Times where embedded systems didn't ask for sophisticated memory hierarchies, support for sophisticated operating systems, high level of interactivity with sophisticated user interfaces seem by now distant. Today those just listed are often essential characteristics inside many embedded systems, especially in the wireless and mobile computing fields. If from one side therefore it is assisted to a progressive convergence of embedded and personal computing worlds, from the other embedded systems maintain some distinctive prerogatives, as for example the great importance that inside them they assume again the occupation of area and the

dissipation of power over that the pure performances, that instead decidedly assume preponderant importance in the systems for personal computing. Anyway, these comparisons have to be handled with care, considering that for example power dissipation is assuming increasing importance also for personal computing systems. As a result, the technology of embedded systems is more and more approaching that of platforms for general purpose computing. However many architectural choices will be subject to specific restrictions. In the past, another property that distinguished embedded systems and personal computing systems was in the degree of integration required among the various components of the system. While in embedded was often necessary to integrate all on a single chip, in personal computing integration among the various parts of the system happened at board level. I single chip was usually reserved for the processor alone. In the last years however a technological drift is leading both the computing areas, embedded and general purpose, towards a single technological platform, the System-on-Chip platform. More and more systems designed for general purpose computing are now including on the same chip not only the processor but also first and second level cache memories, external memory controller, graphics accelerators, etc. Technologies that were originally developed for one of the above mentioned and once well separated domains are now converging on the field of System-on-Chip computing.

6.2 System-on-Chip computing models.

The architecture of the computing platforms available for System-on-Chips can vary depending on the application domain at hand. There are applications were only the management of sensors and primitive communication terminals is needed. For tasks like this is normally sufficient a simple microcontroller, with monitoring functions. In these cases the processor is not demanded to perform onerous computations. When instead the application requires higher levels of performances it is traditionally possible to resort to a number of different solutions, each of which has its own advantages and disadvantages that must carefully be evaluated in the design space exploration phase. In Systems-on-Chip are frequent the applications where is necessary to conduct some complex computations on some data that the system acquires in input at high rate, as is the case wireless systems for example. The solutions that can be adopted are usually three: 1) choose an high performance processor, 2) add a programmable coprocessor adapted for the execution of the task that requires more computational effort or 3) design an ASIC coprocessor specifically targeted to the execution of the critical task. These just listed represent only the traditional

solutions. There are also hybrid solutions, mainly based on the paradigm of the "customization" of flexible computation platforms that can be each time adapted to the requirements of the application at hand. This is the case for example of the reconfigurable coprocessors, that exhibit an highly modular architecture, that allows to not only vary its properties in substantial way in the design phase but often also at run-time. Another example is given by the "customizable" processors. They can be fine tuned both from the point of view of the microarchitecture and from that of the Instruction Set. These systems are highly modular, which advantages are twofold: 1) offer a good solution for the application at hand and 2) be flexible enough to be easily adapted if the systems requirements or the application change. The figure 1 illustrates many of the solutions described above.

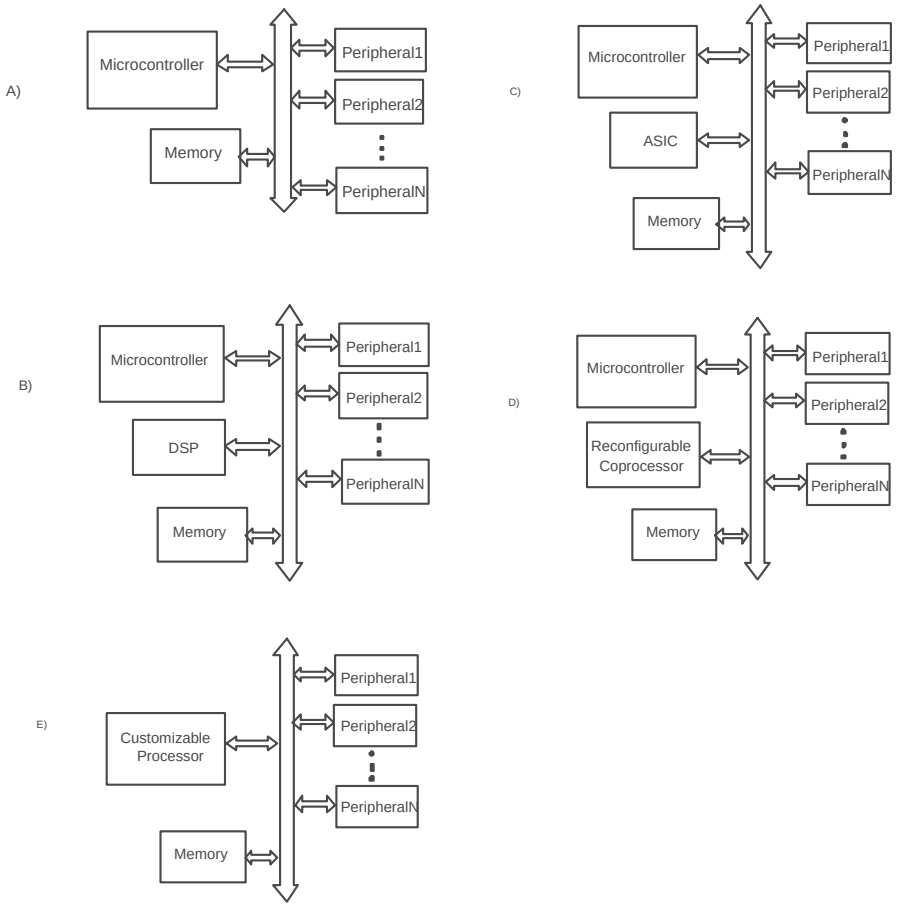


Figure 6.1: Computing platforms for Systems-on-Chip.

In the majority of the cases the programmable coprocessors are named Digital Signal Processors (DSP). Not always the distinction between a general purpose pro-

cessor and a DSP is neat. A DSP represents a processor optimized for the execution of particular tasks, usually some algorithm of digital signal processing. The optimization is achieved both through the addition of custom functional units dedicated to the accelerated execution of particular operations, and sometimes also through an extension of the instruction set, providing additional support for instructions particularly suited for the speed up of a given computational kernel. The spectrum of available solutions for handling computation tasks in the context of Systems-on-Chip is very diverse. At the same time the demand for computing power is continuously increasing, and probably is not going to stop in the future. Another property that computation platforms for future Systems-on-Chip must absolutely have is good scalability. The most promising way to increase the processing power available to SoCs seems to be that of taking full advantage of the various sources of parallelisms embedded in any software program and in particular in the software programs targeted to embedded applications. In the following paragraphs some ideas about the possible ways for the exploitation of this parallelism will be given.

6.3 Advanced processor architectures: Superscalar and VLIW.

Leaving for a moment the embedded field and focusing on general purpose computing were is definitely raw performance the principal requirement, the most widely used processor architecture is the *Superscalar* architecture. *Superscalar* processors are able to execute more than one instructions in parallel. To do this is necessary to modify the intrinsic order execution of computer programs, which is strictly sequential. At the same time there is the need to make sure that even if the order of execution of some of the instructions is changed the correctness of the program must not be compromised. A group of instructions can be executed in parallel only if there is not any dependence among them. Once the instructions potentially executable in parallel have been selected the need to be appropriately dispatched to the functional units available inside the processor. Usually the first task is called *dependency check* and the second *instruction scheduling*. In today's *Superscalar* processors also other techniques are used to improve the overall performance, optimizing the two aforementioned tasks. To perform an analysis of instruction dependencies there is the need to implement inside the processor structures able to maintain a "window" of instructions on which the analysis must be performed at run-time. This normally requires resorting to large memory buffers. Increasing the number of instructions that can be dynamically "watched" inside the instruction window it is more likely

the possibility to find a subset of them that can be executed in parallel . For what just said there is a trade off that must be carefully considered at design time between the performance expected from the system and the overhead in terms of area, latency and power consumption that can be afforded. In fact, large buffers as those needed to implement the instruction window and the dependency check can be very demanding in terms of resources. Unfortunately, in *Superscalar* processor there are not so many other ways to improve performance, if not incurring significant overheads. This certainly represents one of the greatest elements of weakness of *Superscalar* processors.

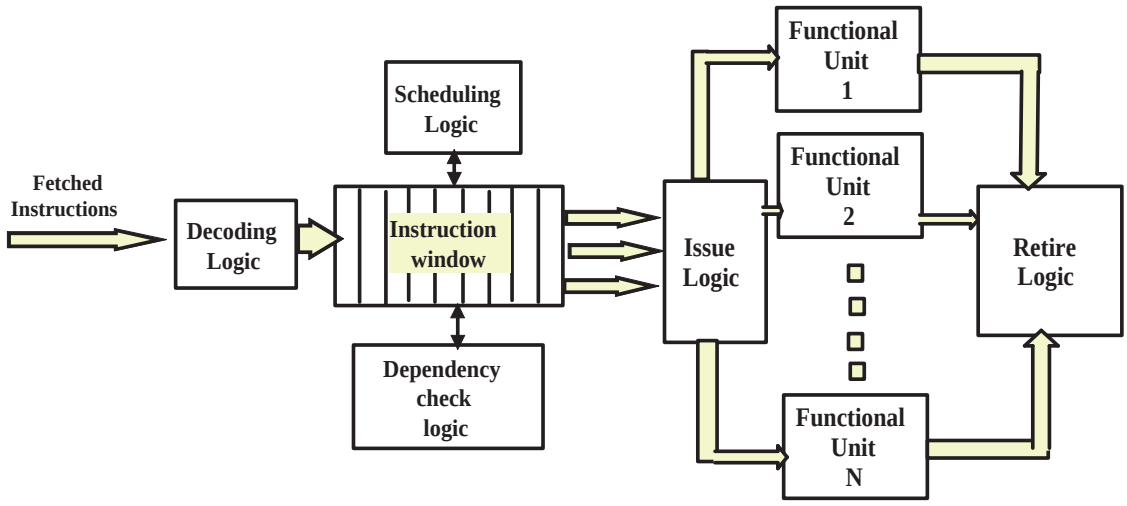


Figure 6.2: Logic block diagram of a *Superscalar* processor.

The process of extracting from the software program instructions to be executed in parallel can lead to a condition where the execution of one or more instructions does not respect anymore the original order ($out-of-order\ execution$). However, there are moments where the order of instruction execution as seen from outside (the programmer) must match exactly the original sequential order. This occurs for example when handling interrupts and exceptions. For this reason in most of *Superscalar* processors there is a functional unit called *Retire Unit*, but which is often referred to as *Reorder Buffer*, which is placed downstream of the functional units that has the sole purpose to guarantee that the system's architectural state, i.e. memories and registers, is changed only accordingly to the native sequential order of the program. Only in this way the processor can safely respect the condition of *Precise Interrupts*. Implementation of the *Retire Unit* is very expensive in terms of hardware resources and power dissipation. This is a further demonstration of the susceptibility of the efficiency of *Superscalar* architectures to implementation pa-

rameters. These problems explain why until now *Superscalar* processors have not being successfully introduced in embedded systems. They are very hungry of hardware resources and in embedded systems, in Systems-on-Chip, hardware resources are really expensive. Furthermore the waste of hardware resources in *Superscalar* processors does not scale very well with performance. Often to get only a small increase in performances the overhead in terms of hardware resources becomes unacceptable. For the problems described above, to solve the increasing performance hunger of Systems-on-Chip other solutions must be sought of. *Superscalar* architectures are not the only ones following the approach of pursuing more efficiency exploiting the inner parallelism of software programs. There are also other architectural processor models that try to exploit the same parallelism, just in a different, and maybe more efficient, way. These other processor architectures try to match the performance of *Superscalar* architectures without incurring in the significant overhead that characterizes their implementation. One of the best promising solutions in this context is represented by $VLIW$ (Very Long Instruction Word) architectures. The basic approach followed in $VLIW$ processors is to get a suitable exploitation of the parallelism assigning its discovering not at hardware resources but the software compilation tool chain . In this way the processor microarchitecture is freed from many the complexities typical of *Superscalars*, accepting to pay a little price in terms of performance, as will be better explained later. It is not required anymore to use large buffers for placing the instructions on which to conduct the dependency analyses on-the-fly. Moreover, instruction execution in hardware happens now in a strictly sequential order. In fact, the final execution sequence of instructions (the *schedule*) has been generated by the compiler. The processor core has to just strictly follow the execution schedule without taking any other action. Widespread adoption of this processor architecture in Systems-on-Chip has been until now prevented by the great level of complexity of the software tool chain that is required to support this particular processor architecture. Significant breackthrughts made recently in this field have allowed to overcame these difficulties. Since in VLIW processors much of the work is left to the compiler and optimization software, the quality of these components determines to a large extent the quality of the system as a whole. If for some reason the compiler is not good enough to extract all the available parallelism from the program and to put it in a fashion suitable for the optimal execution by the VLIW core, the core itself cannot do much more to improve performance. Another oddity of VLIW processors regards the way programs are stored in memory. A single instructions is actually the concatenation of a number of primitive operations that equals the number of independent execution *lanes* available in the hardware. These compound instructions usually take the name of *Long Words*, *molecules* or *bundles*.

Primitive operations belonging to a same macro-instruction are scheduled for execution in the same clock cycle. During the Fetch phase the processor withdraws a *Long Word* from the memory. Considering that it is now the compiler to give guarantees on the absence of bonds of dependence among the primitive operations of a single Long Word there is no need for dependency check in hardware, even if hardware could still directly manage specific types of conflicts. Direct consequence of the characteristics exposed above it is the possibility to use a more slender and rational microarchitecture compared to that required by *Superscalar* processors.

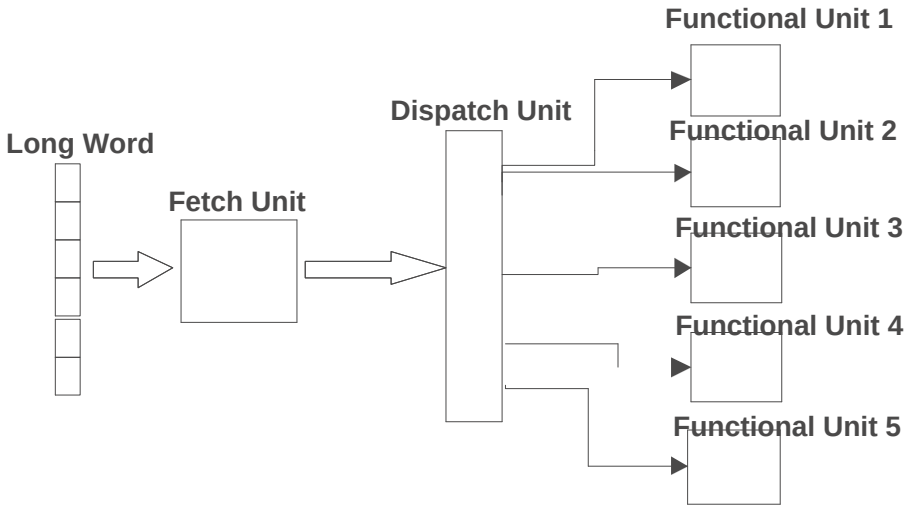


Figure 6.3: VLIW processor.

6.4 VLIW processors.

6.4.1 Code generation for $VLIW$ processors.

Every software program is composed of a number of *elementary bricks* called *basic blocks*. A basic block consists in a code fragment which does not embed any instructions that can determine variations of the control flow. At the same time this code fragment must have single-entry, it can be reached only by another basic block, but it is Multiple-exit, it can transfer control to more than one different basic blocks. The number of instructions included in a basic block varies according to the particular application domain. In general purpose applications basic blocks are usually formed at the most from 5-6 instructions. In software for embedded systems the length of the basic blocks is much bigger. The reason for this must be sought in the fact

that the general purpose applications are often much more branch-intensive than those for embedded systems. Among instructions belonging to different basic blocks exists an implicit dependency relation. If two instructions belong to different blocks it means that the basic block downstream is not necessarily executed every time the execution of the BB upstream is triggered. Therefore, execution of instructions belonging to different BB cannot be scheduled in the same cycle. However in modern compilers for $VLIW$ is indeed possible to produce some schedules which have instructions of different BB scheduled in the same cycle, but this is made possible only by the use of advanced optimization techniques consisting in manipulating the original program through the insertion of additional instructions that allow to guarantee the respect of the original program semantics under all conditions, also those less probable. One of the most successful optimization techniques implemented in advanced VLIW compilers is the *scheduling across branches*. Different flavors of scheduling across branches exist, that differ mainly for the type of *macroblocks* that is produced when different BB are merged. The basic approach in all the techniques of basic block fusion consists in making a number of assumptions on the dynamic behavior of the program. After the most likely behavior of a given program fragment has been established, the program execution is optimized across this path, also called program *hot spot*. In practice, performance in the most likely case is increased at the expense of performance in the less likely one. At any given branch point inside a program, a point where a change in control flow can take place, the compiler assigns a statistical weight. These weights are determined performing a profiling of the program. All this process happens statically, off line, no run time information is available at this stage. This techniques enable the compiler to look for instructions to execute in parallel along an entire *hot spot*, well beyond the boundaries of a single BB. This greatly enhances the likelihood of finding more parallel instructions.

There is one optimization technique based on scheduling across branches that has received particular success and that has been widely adopted in $VLIW$ available today in the market. It is the $Trace\ Scheduling$ technique. The purpose of the TS algorithm is to extract from a program fragment the segments which have higher execution probability, and then to perform an aggressive optimization of those fragments, taking care at the same time of the insertion of some form of recovery mechanism that deals with the need to take into account also the least frequent case in which the dynamic execution stream of the program abandons one of the hot spots. The $Trace\ Scheduling$ has the strong merit to be able to treat effectively code fragments of different types, and in particular to be useful also in areas of code full of branches. Hereafter the basic principles behind TS will be explained. After the front-end compiler optimizations have been performed and the instructions

have assumed the form of machine language the CFG (Control Flow Graph) of the program is passed to the TS (Trace Scheduler). The TS annotates the CFG with information regarding the execution statistics of the program, gathered by means of profiling runs. Then, the Trace Scheduler begins to execute the loop comprised of the following phases:

A. A sequence of instructions to be inserted inside the Schedule is selected. This sequence is called *trace*. The length of a single *trace* can be determined by various constraints: 1) end of functions (return / entry), 2) end of loops and 3) collision with fragments of code already scheduled.

B. The *trace* is removed from the CFG and passed to the Scheduler.

C. When the Scheduler emits the Schedule, it is inserted back into the CFG, exactly where the original Trace was placed. To take into account also the case where the assumptions made in building the *trace* need to be discarded, part of the scheduled instructions is replicated in the CFG.

D. The process is repeated until every instruction has been inserted in one of the Traces and that every *trace* has been replaced by the corresponding partial schedule.

E. The best possible order for the object code is chosen and the binary file is generated.

The best possible order for the object code is chosen and the binary file is generated.

Figs. 6.4, 6.5, 6.6, 6.7 and 6.8 describe the concepts just listed.

In *Trace Scheduling*, the final product of all the compilation and optimization process is the Schedule, the detailed execution plan for the instructions in the VLIW core. The core has only to execute the schedule with the best possible efficiency. As already said earlier *Superscalar* processors generate internally the schedule. Now that the details of the VLIW model of execution have been introduced is possible to answer to the following question: with reference to a sample fragment of code, is better the schedule generated by the compiler for a VLIW processor or the schedule generated in hardware by a *Superscalar Processor* ? The answer is that the process of profiling the software application for determining the execution statistics to plugged inside the Software schedule in case of VLIW processor is inherently inaccurate, because it does not take into account the dynamic behavior of the program. Aspects like the variation of the inputs cannot be predicted with accuracy. The instruction window hardwired inside a *Superscalar processor* instead

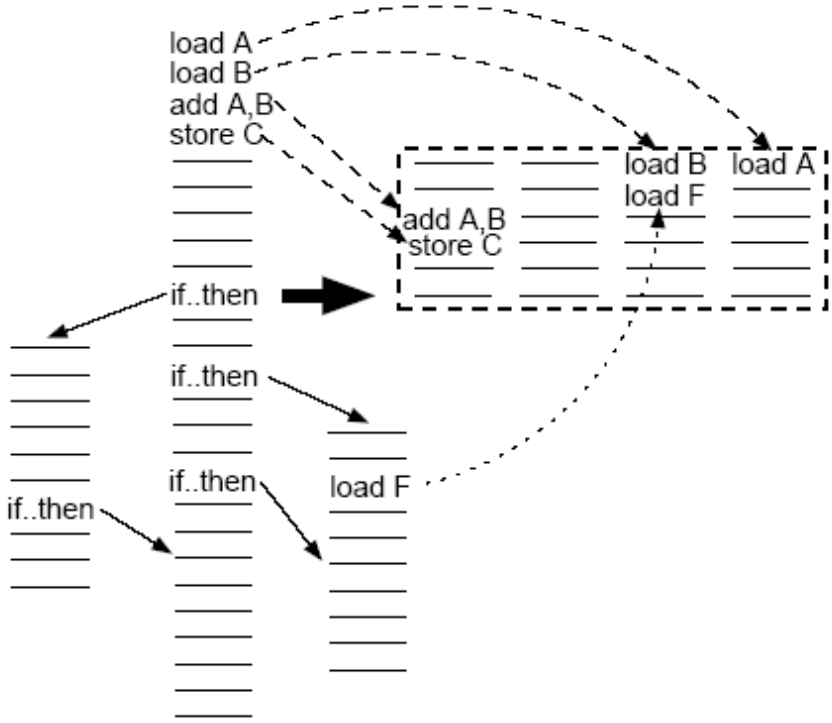


Figure 6.4: Scheduling across different basic blocks.

has access to the instructions just right before their execution. The only source of ambiguity in this latter case is due to the possible inclusion of instructions that have to be executed in a *speculative* way. On the other hand the $VLIW$ approach has the advantage that the code optimizations are performed on a large window of instructions, larger by far than the number of instruction a *Superscalar* processor can analyze in hardware, at least if the cost in terms of hardware resources must be kept into reasonable bounds. If a program has plenty of intrinsic parallelism the $VLIW$ approach is superior, while the *Superscalar* approach wins when the available parallelism is less.

6.4.2 Introduction to binary compatibility.

The VLIW model of computation assumes that the code generation and optimization tool chain has access to detailed information about the hardware microarchitecture of the processing core. The definition of the instruction schedule is completely up to the

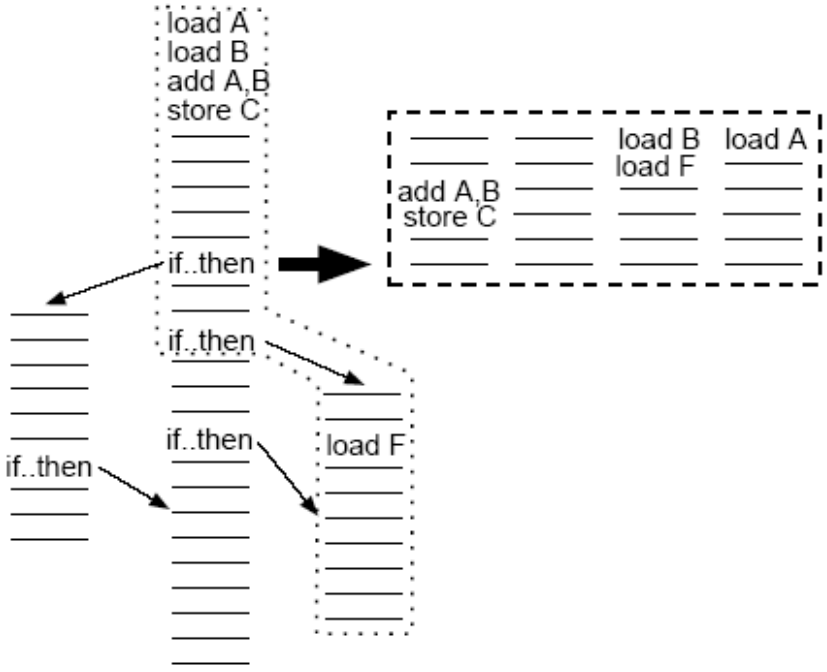


Figure 6.5: Selection of a Trace and scheduling of the instructions inside.

compiler and this can be done only if the compiler is fully aware of many of the details about the processor's microarchitecture. To establish that no resource conflict can exist between two arbitrary instructions, for example, the compiler needs to have complete knowledge of the latencies of each of the functional units inside the core. Without this information for the compiler would not be possible to find a schedule that is both *correct* and *optimal* for a given code fragment. A direct side effect of this important property is that a program compiled for a VLIW core having a given set of latencies for its functional units cannot in principle be executed correctly in a different core where some of the latencies of the functional units have been changed. This property of VLIW architectures poses serious concerns about the possibility to reuse binary code across different generation of the same processor family. Concerns like this will be better discussed in the continuation of this thesis, because these are exactly the issues that have been dealt with in the course of the thesis work. Moreover, for the optimization algorithms to be really effective, the compiler needs to have the best possible knowledge of the hardware core. Thus, there is again the need to find a good trade off between accuracy of the hardware representation available to the compiler and ease of porting of the binary code to other generations of the

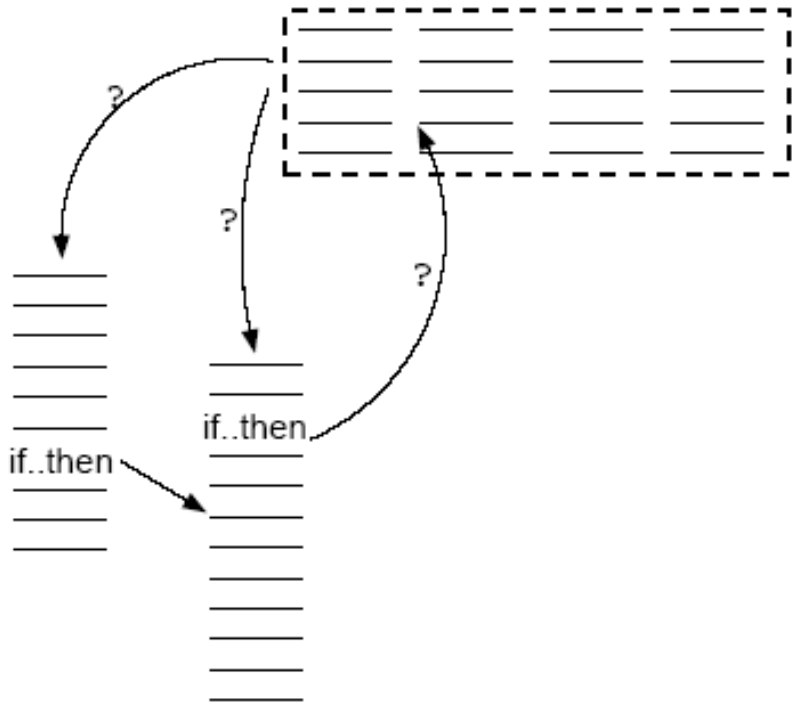


Figure 6.6: Substitution of the Trace with the Schedule and analysis of the situation at the edges.

processor's microarchitecture. In the light of what just said is possible to understand why adaption of VLIW architectures has started in embedded systems and not in general purpose systems. The reason is that in embedded systems recompilation of the software has rarely been a problem, while in the general purpose sector the capacity to execute legacy code is a must. However, in the last years issues like code reuse and binary compatibility are increasingly important also in the context of Systems-on-Chip, because also for SoCs the possibility to exchange compiled code across different platform and processor families is becoming an advantage as perceived by the market.

6.4.3 Analysis of VLIW microarchitecture.

In principle, a $VLIW$ processor is represented by a number of parallel *lanes*, each one dedicated to the execution of one of the many primitive operations (*uops*) that included inside a single Long Word fetched from memory.

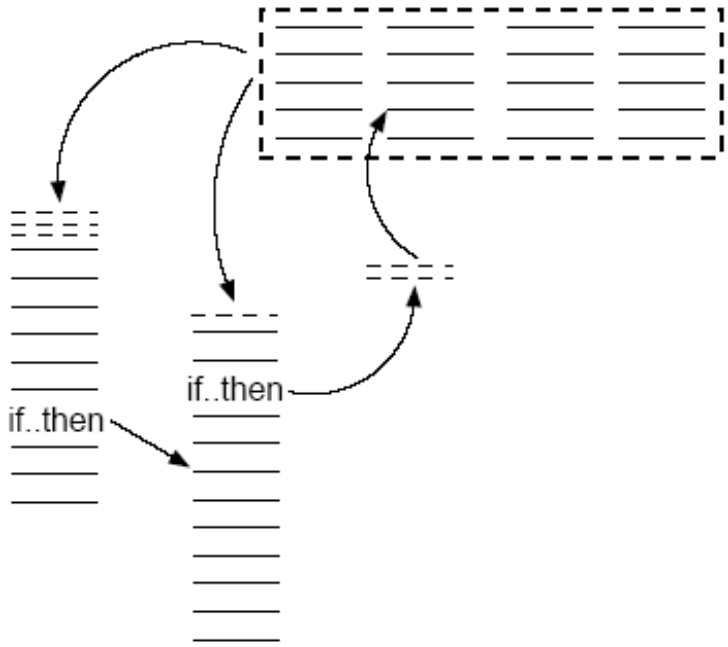


Figure 6.7: Generation of the compensation code at the edge of the Trace.

To fully exploit the parallelism inside a program fragment, the VLIW compiler needs to count on a large number of registers. Many registers are needed because two instructions can be considered to not conflict only if the registers sets they access in read or write mode are disjoint. Since optimization algorithms have to be applied to large windows of instructions, the demand for registers to assign to operands and results is very high. At the same time, from an hardware perspective having to deal with large monolithic register files poses serious treats to the system's performance. For this reason sometimes VLIW cores resort to using clustered register file configurations. Clustered VLIW configurations are considered to scale better with the performance required. Fig. 6.10 and Fig. 6.11 show respectively a monolithic and a clustered VLIW core.

In clustered architecture there is the need to provide some means to exchange data across the different register files. This can be done using specific uops or providing some hardwired mechanism inside the processor core. The intrinsic modularity of Clustered architectures allows to speed up the retargeting of the processor core for a different application domain, or just to increase the processing power. Changing

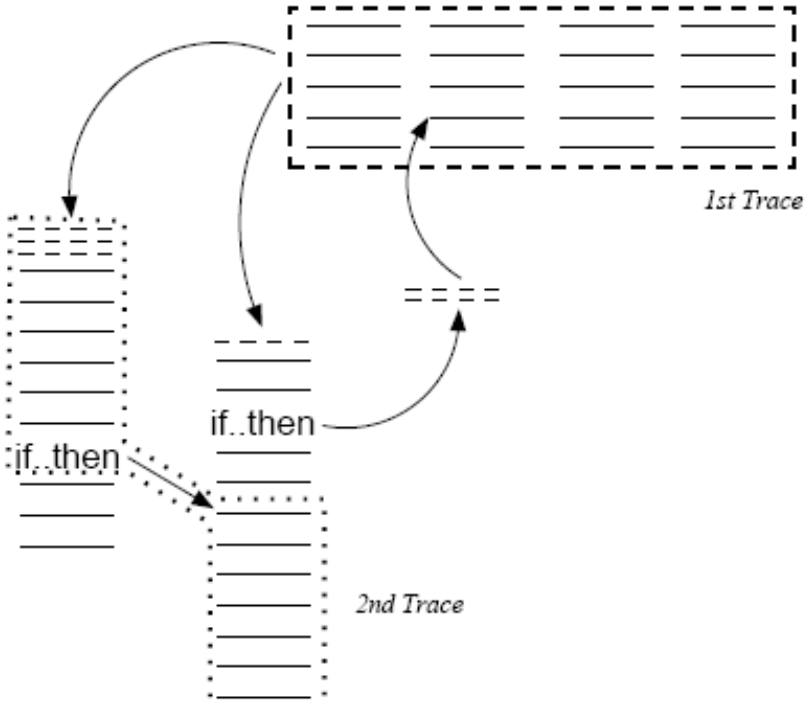


Figure 6.8: Selection of the next Trace.

the number of clusters inside the core the designer can meet different performance and application requirements, without having to deal with the complete redesign of the single cluster, that would take much more time.

6.4.4 Compression of VLIW programs in memory.

VLIW programs are made of Long Words, where all the uops the core has to execute in a single cycle are stored. If a given *lane* of the core in a particular cycle needs to be stalled to avoid resource conflict with another instruction, the corresponding slot inside the VLIW is filled by the compiler with a NOP tag. This NOP tag specifies that the functional unit assigned to that slot will be idle in that cycle. Since the situation where some of the slots inside a Long Word are filled with NOP is actually very common, storing a program in memory in this format would lead to a great waste of memory resources. For these reason, have been designed a number of techniques for the compression of VLIW programs in memory, where the basic approach is to avoid wasting space storing NOP information. The problem

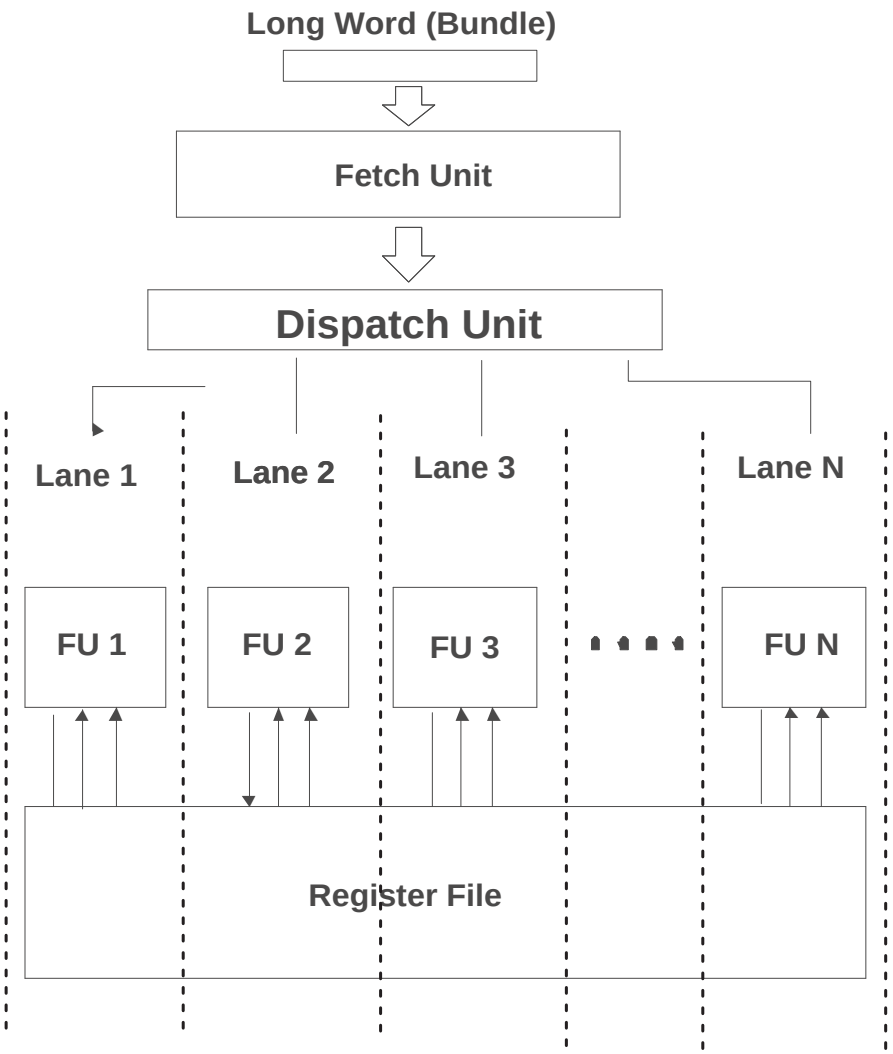


Figure 6.9: Execution *lanes* in a $VLIW$ processor.

is worsened by the fact that also the optimization techniques performed by VLIW compiler have as a side effect a significant increase on the size of the program. One the most widely used compression techniques consists in using a few additional bits embedded in the Long Word that specify if a specific slot has to execute some real instruction or not. A similar scheme is implemented inside the TIC6xxx, a VLIW processor from Texas Instruments. In this processor family the Long Word is referred to as $fetch\ packet$. The bits that implement the compression scheme are called $p - bits$.

When the Fetch packet is acquired from memory the p-bits are scanned from

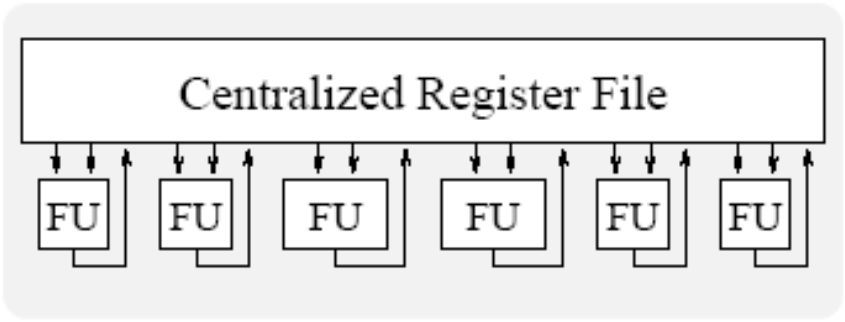


Figure 6.10: Example of centralized architecture.

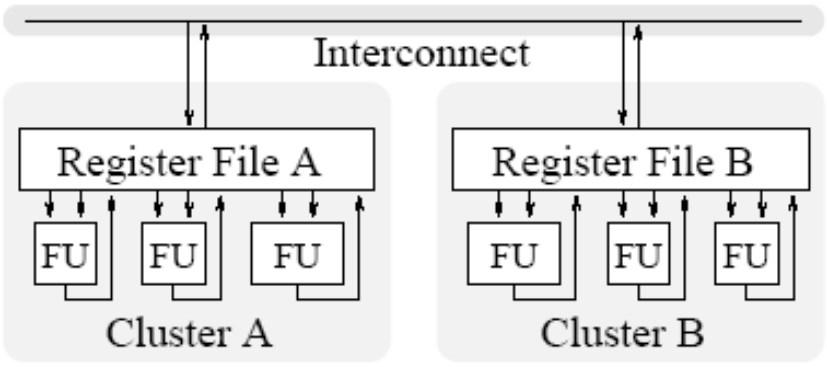


Figure 6.11: Example of architecture Clustered.

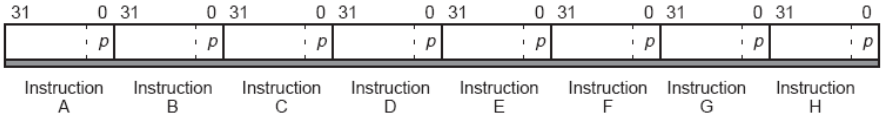


Figure 6.12: Format of the Fetch Packet for the TIC6XXX.

left to right. The semantics is the following: 1) if the p-bit for a given slot is 1 then also the uop specified in the next slot will be executed in the same cycle; 2) When instead the p-bit is 0 the uop placed on the next slot will be executed one cycle after. Fig. 6.13 and Fig. 6.14 show respectively a possible configuration of the Fetch Packet for the TIC6XXX processor and the resultant schedule.

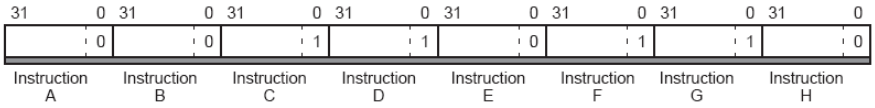


Figure 6.13: Possible configuration for the Fetch Packet of a TIC6XXX.

Cycle/Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

Figure 6.14: Schedule that results from the preceding pattern.

Chapter 7

ST230 architecture.

The ST230 belongs to the Lx family of processors, developed in partnership by STMicroelectronics and the HP labs. The Lx architecture is a VLIW architecture specifically targeted to Systems-on-chip. It builds upon a sophisticate code generation platform derived from the Multiflow compiler, one of the best VLIW compilers available in the industry that is the result of more than 20 years of research (CITE Multiflow). The Lx architecture has been designed from the beginning with scalability in mind, and this is way Lx designers adopted a clustered organization. Indeed, the ST230 represents one instance of the Lx architecture that includes one single computing cluster. The Lx Instruction Set Architecture (ISA) belongs to the class of RISC (Reduced Instruction Set Computer) ISAs. This means that Lx instructions are in principle quite simple, not including operations with complex semantics. It lacks also SIMD-like operations , that have been included in the Instruction Set of other VLIW processors for Systems-on-chip as the TIC6xxx. Fig. 7.1 shows an high level representation of the ST230 processor. Not many details can be disclosed about the ST230 architecture due to confidentiality reasons.

ST230 includes 4 Integer Units, 2 Multiply Units, a Load Store Unit and a Branch Unit. There are two different Register Files: the General Purpose Register File and the Branch Registers Register File. The first contains 64 registers of 32 bits each and the second contains 8 registers of 1 single bit each. The Latency of Integer Units is 1 cycle. They take in input 2 operands and produce in output 2 results: 1) a 32 bits value (the result) and 2) a condition bit. Multiply Units have a 3 stage pipeline and have a latency of 3 cycles, although they can accept a new instruction per cycle. The ST230 pipeline is partially exposed to the compiler. In particular the compiler has notion of the instruction latencies and of other microarchitectural details has the status of the bypass network, etc.

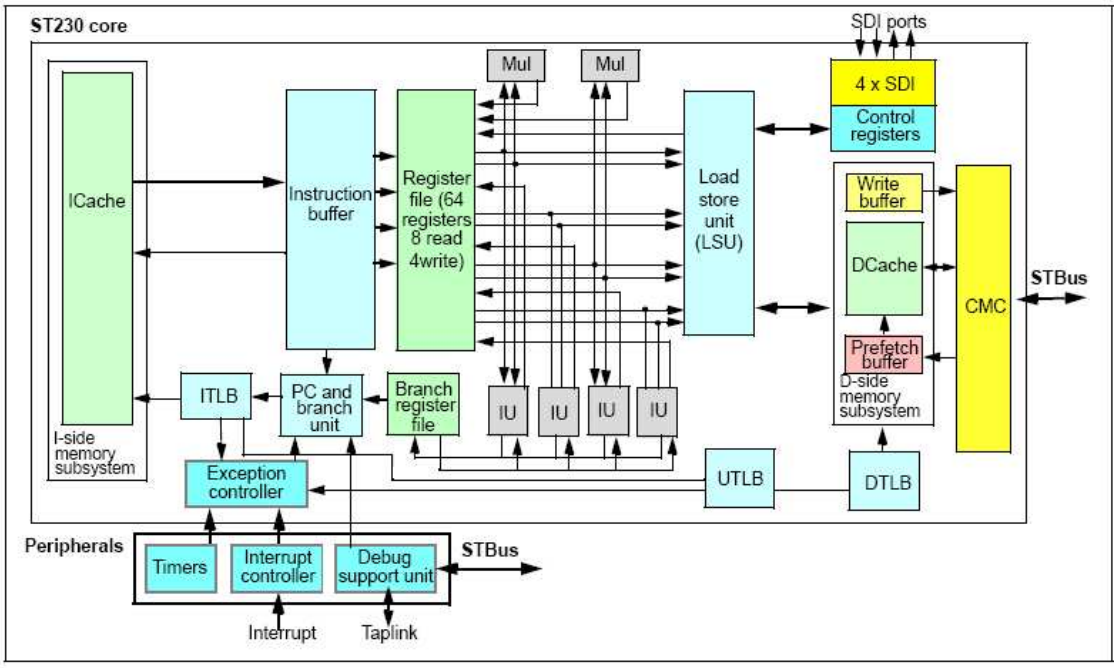


Figure 7.1: High level block diagram of ST230.

7.1 Fetch Packet.

The ST230 Fetch Packet is called *bundle*, and it is composed of 4 micro-instructions (syllables). Every syllable has a width of 32 bits.

The *stop bit* has function similar to the p-bit of the TIC6XXX. If the *stop bit* is 1 then the corresponding syllable is the last of its bundle. An ST230 bundle can include no more than 4 syllables. Unlike other VLIW processors the ST230 does not have a Dispatch Unit for issuing the syllables to the functional units. Instruction bundles are passed directly to the functional units which are directly in charge of understanding if a particular syllable has to be executed locally or belongs to another FU. For this reason bundles do not have any information specifying where a given syllable has to be executed. Control and Status registers are Memory mapped. When the core performs a memory access to the address space of the Control registers, the Load Store Unit drives the relevant information to the actual Control registers.

7.2 Software load speculation.

The ST230 processor supports the *speculative* execution of load instructions. The implementation of this feature gives more freedom to the compiler to implement aggressive optimizations of the code, allowing the execution of load instructions in advance. This kind of code transformation is often very useful to increase the available parallelism of a program fragment that can be then exploited by the parallelizing compiler. Support for this functionality is provided through inclusion in the Lx instruction set of *dismissible load* instructions. These load instructions behave exactly like normal load instructions, except when some of the following conditions occurs:

A. The address of the load instruction belongs to a critical address segment, for example to the address space of some peripherals. In this case also a read access can have dangerous effects.

B. When the execution of the load would have caused an exception condition.

C. If the speculative load determines a bus error the corresponding exception is executed anyway.

7.3 Management of the Program Counter.

In the ST230 the Program Counter (PC) is not handled like the other registers. Update of the program counter happens only as a side effect of the execution of some specific instruction classes. This is very different to what happens in other processors available for Systems-on-chip, as ARM processors. In ARM processors in fact the PC can be accessed exactly like any other general purpose register, even if with some limitations.

7.4 Lx Architecture and binary compatibility with ISA ARM.

ARM processors are the most widely used family of processors for Systems-on-chip. It is estimated the ARM processors have a market share around 70%, that is even higher if the analysis is restricted to wireless applications. This widespread diffusion of ARM processors has the side effect that a large part of the software

applications available for Systems-on-chip are available as ARM binary code. As already explained in the previous chapter the problem of binary code reuse and binary compatibility across families of processors is becoming relevant also in the context of Systems-on-Chip. In today's Systems-on-chip, many of the software programs that will actually run on the computing platform are developed by third party companies. For these companies, the possibility to reuse the same binary code across different system architectures and families of processors would allow to avoid the expensive process of porting and retargeting a software application to a different executing platform. It is well known for example that much of the success of the *wintel* platform in general purpose computing depends on the support for the large base of legacy applications and programs that are available for the x86 platforms. In this context the large base of applications available for the x86 platforms becomes a barrier against any one trying to push a platform different from x86 for general purpose computers. Providing direct support for ARM programs would allow to ST230 processors to have access to the large base of application in ARM code that exists for Systems-on-chip. For some applications it is even acceptable for the execution of the ARM code to be less efficient than the execution of native Lx code. This is for example the case of the plenty of little applications available for mobile phones like ring tones, games, etc. Relaxing the requirement of having the same performance when executing ARM code or native VLIW code it is possible to extend the number of available technical solutions that are available for implementing binary compatibility between the two instruction sets. The choice among the available solutions would then be driven also by other factors as cost in terms of hardware resources, time to market, minimization of the impact on the standard processor architecture.

Chapter 8

Hardware and software techniques for binary translation of Instruction Sets.

8.1 Overview of binary translation techniques.

The problem of executing programs compiled for a given machine on different machines has been dealt with for a long time both in the academic and in the industrial sector. Technologies that solve to a certain degree the aforementioned problem are available from many years. It is for example the case of the so called *emulators*. Emulators are programs that perform some kind of encapsulation (wrapping) of software programs written for s *target* machine in order to execute them on a different (*host*) machine. Typically, *target* and *host* machine may have different processors, different Operating Systems or a combination of both. The major weakness of this approach is that execution of a program under emulation is much slower than native execution (execution of programs directly compiled for the *host* machine), on the average by a factor of 10. See Fig. for a description of a typical emulator. Emulation is by definition a purely software technique.

Much work has been spent in trying to decrease the performance gap between native execution and execution under emulation. New techniques have been discovered, but pure emulation is still a slow process. This is why the research community has started searching for alternative techniques that could in principle speed up the process of executing a program compiled for a *target* machine on a different *host* machine. One of the proposed solutions has been to provide the *host* hardware of

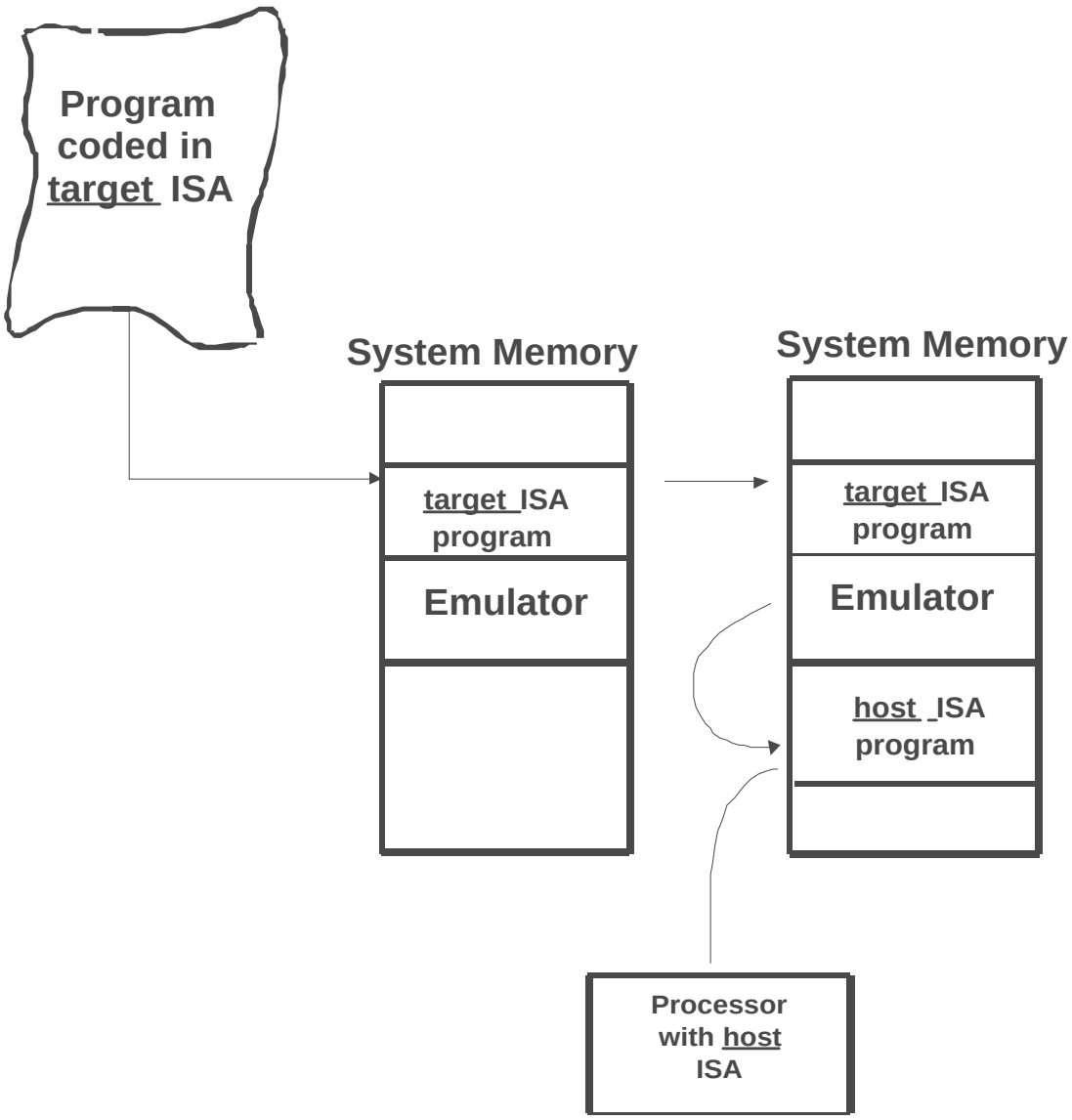


Figure 8.1: Conceptual representation of the operation of an emulator.

some additional resources that could enable the speed up of the execution of the *target* program. The hardware approach for binary compatibility among ISAs builds upon large work made in the past for solving a slightly different problem: providing solutions for achieving significant performance improvements of processor cores designed for the execution of CISC-like Instruction Sets. The distinguishing feature of CISC (Complex Instruction Set Computer) Instruction Sets Structures is that they include instructions of complex semantics, like instructions for performing some kind of computations on operands gathered directly from memory bypassing the Register

File. Implementation of this class of instructions, and of complex instructions in general, poses serious treats on the efficiency of the computing hardware, because complex logic is needed to support them in hardware. The implementation of such complex control structures in hardware affects the performance of the processing core as a whole, imposing for example un upper bound on the working frequency of the machine. So, if for example 80% of the processing core could run at a frequency of say 1Ghz and the remaining 20% could just run at a maximum frequency of say 500 Mhz, the overall maximum frequency for the design would be 500 Mhz. Moreover, the complex control logic required for the support of the Complex Instructions right inside the processor's functional unit requires significant power consumption and overhead in terms of area occupation. For this reason people has been searching for solutions that could allow in principle to avoid executing the complex instructions in hardware, but without changing at the same time the processor ISA has seen by the programmer, because this would compromise the support of legacy code. Changes in the microarchitecture of the processing core must be somehow hidden to the programmer. In this context, a good solution seemed to be to build the processing core around a RISC engine, not a CISC engine, and to add hardware structures in order to wrap this RISC processing engine to make the outside world (the code generation toolchain) believe it is a CISC engine. In this way there is not in principle any need to change the compilation toolchain. The change in the microarchitecture is not exposed to the compiler. The approach behind RISC Instruction Sets is exactly the opposite of that behind CISCs. If CISCs supported complex, intricate instructions, RISC processors provide only support for easy, fast and streamlined instructions, in order to support high operating frequencies. The microarchitecture of a RISC processor is typically much more efficient in terms of figures of merit as area occupation, latency of the functional units and power consumption. In particular the hardware implementation of the control structures is much more simple in RISC than in CISC processors. In order to effectively use this approach, the RISC processing engine must be completed with some kind of predecoding resources that must take care of the task of translating the instructions fetched from memory, of CISC type, in RISC instructions ready to be executed by the functional units. If the translation process is efficient enough, large improvements in terms of performance can be reached with this approach. This is the reason why today the most widely used x86, x86 is a CISC ISA, processors in the market as many Intel and AMD processor families resort to this technique for achieving high throughput at lower cost. In the framework just outlined, the *target* ISA would be the CISC ISA, while the *host* ISA is RISC.

Even the approach described above, superior to traditional CISC microarchi-

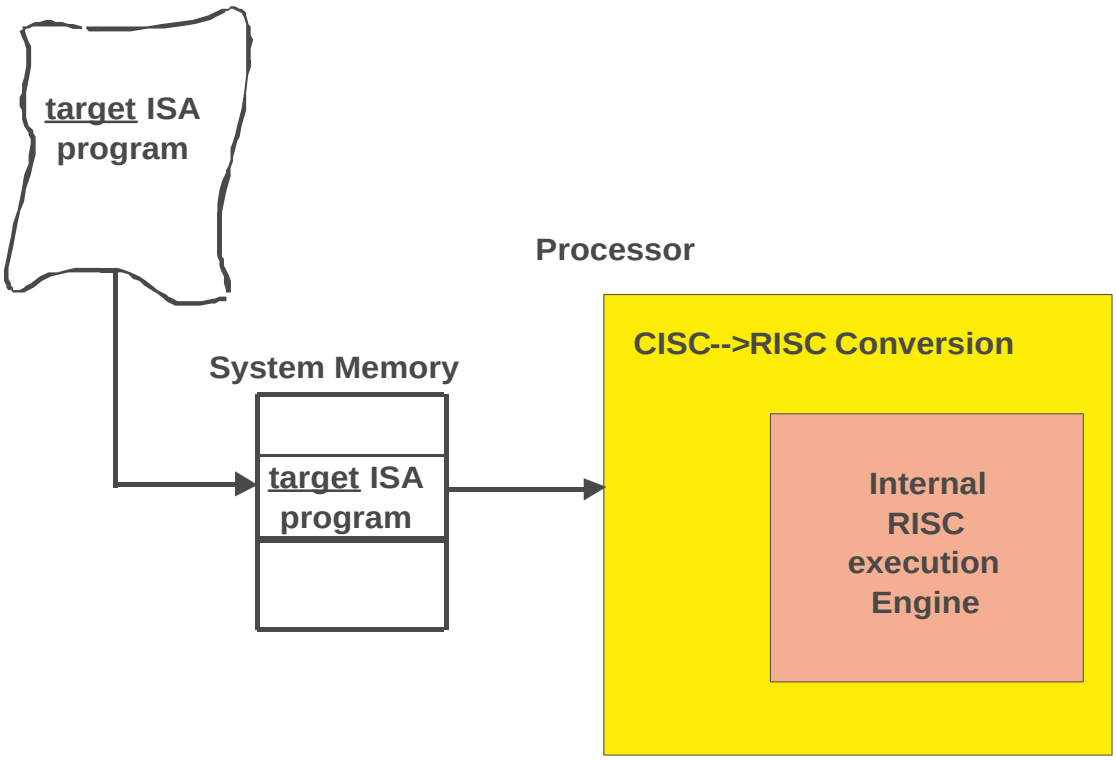


Figure 8.2: CISC to RISC ISA conversion in hardware in modern x86 processors.

tectures, has its own drawbacks. Sometimes also the *decoding logic* needed to map some of the complex CISC instructions (*target* instructions) into one or more RISC-like (*host*) instructions can be too complex. Some x86 instructions that need for example a long sequence of RISC instruction to reproduce their semantics. In these cases a possible optimization is to rely on software emulation routines for performing the mapping. The replacement of the more complex instructions with the RISC emulation routines could be performed by the compiler or directly in the hardware, using for example a ROM memory placed inside the decoder. Advantages are even more if the complex instructions have a low execution probability, because in this case implementing their mapping in hardware would be much like a waste of resources. Hybrid, hardware-software, solutions to the problem of mapping a *target* ISA in a *host* ISA like the one described above are becoming increasingly popular. However, exploiting the full potential of these techniques requires a careful partitioning among tasks to be assigned to hardware resources and tasks to be assigned to software.

The Instruction Sets supported by VLIW processors are almost always RISC ISAs. The Lx ISA of the ST230 processor described in the previous chapter is in

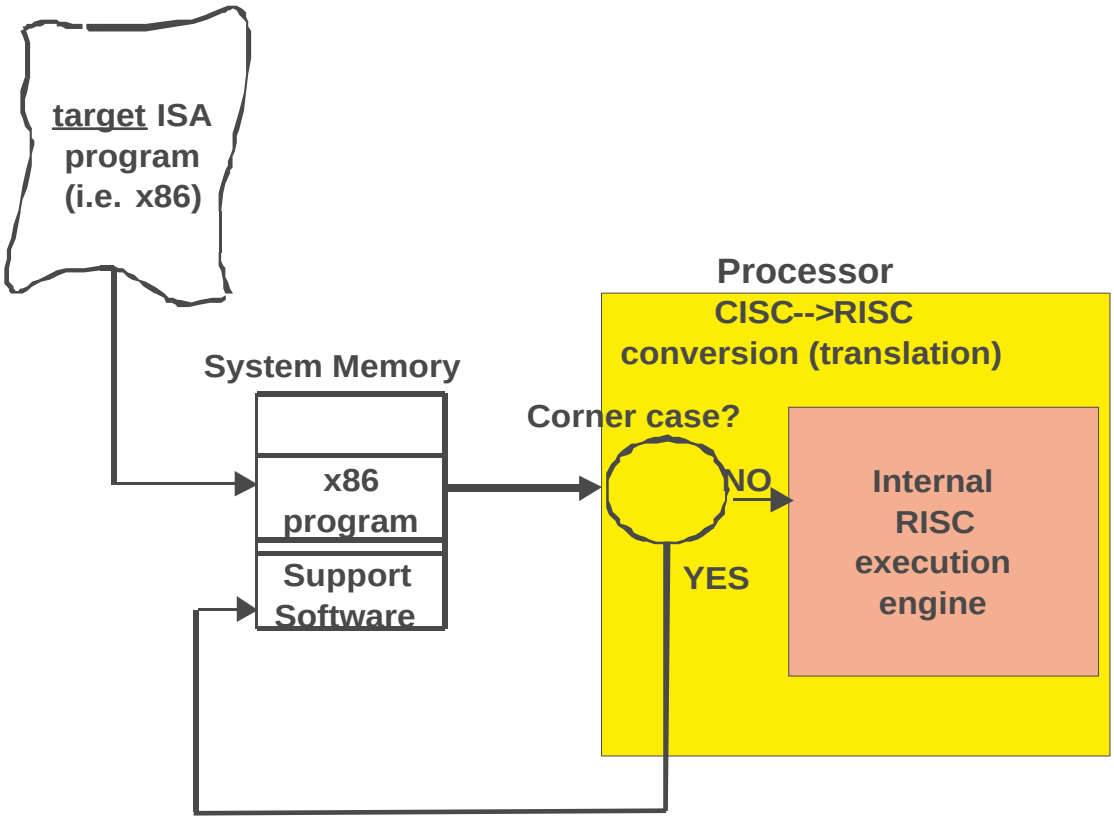


Figure 8.3: Hardware approach to the binary conversion with software support for corner cases.

fact a RISC ISA. Then, exactly in the same way as RISC engine can be wrapped to become a CISC processor, as described above, also a VLIW execution engine could be used to speed up execution of a CISC ISA. And in this latter case speed ups would be probably much more consistent. Also the aforementioned hybrid techniques can be easily applied to systems built around VLIW processing engines. An additional advantage of using VLIW engines is that when hybrid approaches are used, approaches where part of the problem is delegated to software algorithms, the sophisticate software infrastructure that surrounds VLIW processors can be exploited. Hybrid approaches for mapping a *target* ISA on a *host* ISA, process of *binary translation*, in the context of VLIW execution engines will be better discussed later.

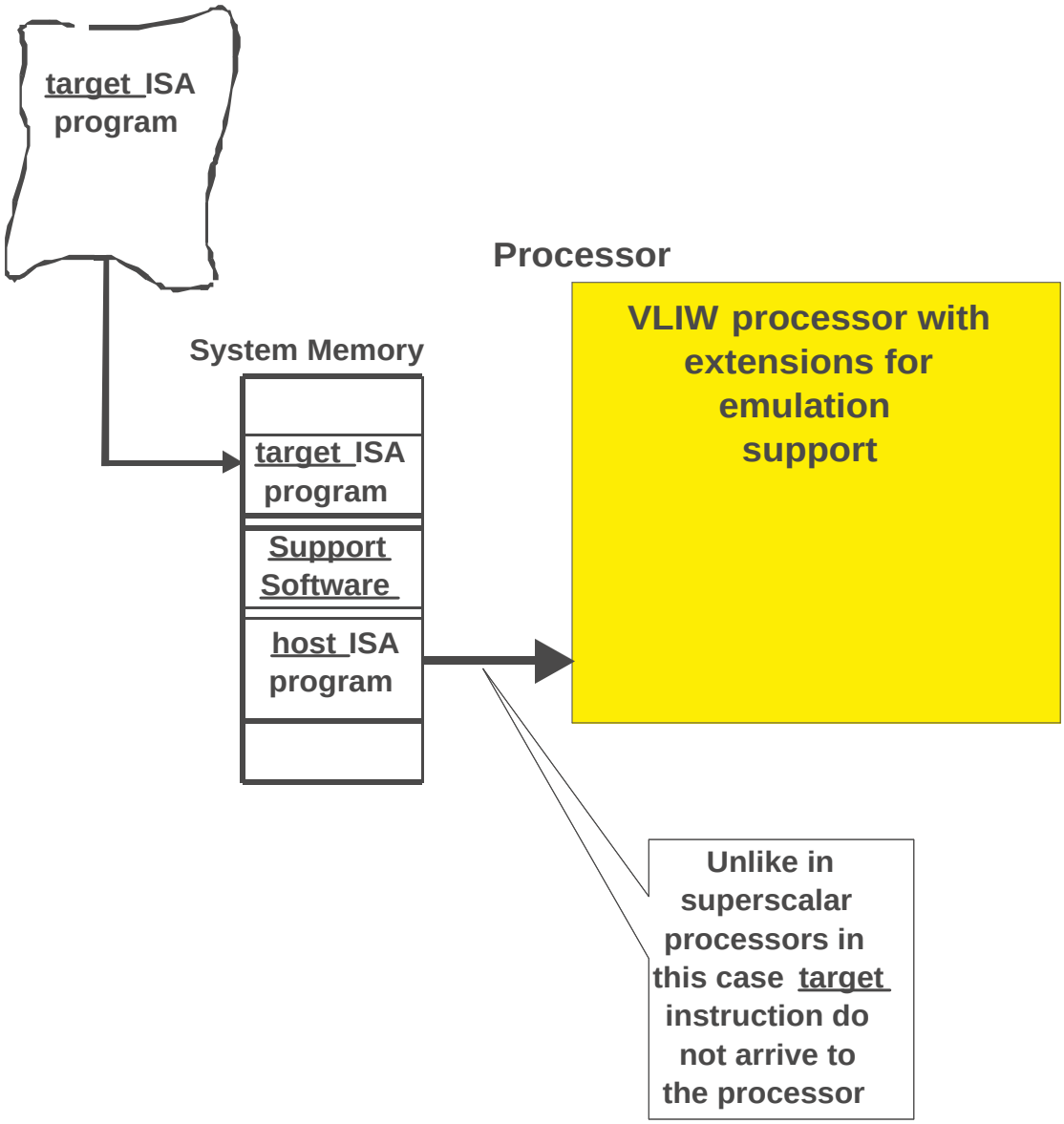


Figure 8.4: Binary conversion in the context of VLIW processors.

8.2 Supporting binary compatibility in hardware.

In the following paragraph will be given a more in depth treatment of hardware techniques for binary translation.

8.2.1 Hardware Binary Translation in superscalar processors.

Most of today's superscalar processors that support the x86 Instruction Set include both resources for HBT (Hardware Binary Translation) and for performing *Dynamic Scheduling* of instructions. The picture is completed by a parallel execution engine which can execute more instructions in parallel. *Target* ISA instructions, x86 instructions, are translated before being executed into one or more *host* instructions, RISC-like instructions often called *uops*, $micro-instructions$ or *operations*. Fig. 8.5 shows the typical front-end of a modern superscalar processor.

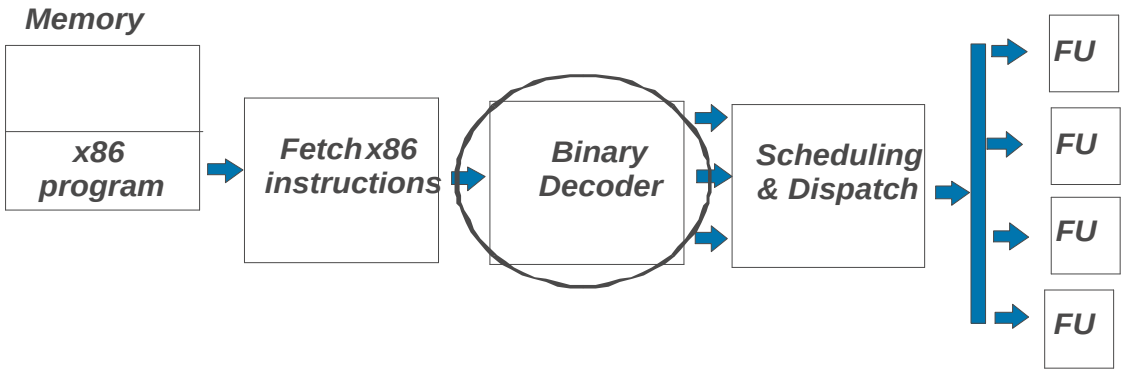


Figure 8.5: Front-end of a modern superscalar processor.

In what follows, description will focus on the inner workings of hardware translation units. There are two basic approaches in the design of hardware translation units: 1) based on combinational logic or 2) based on the so called $micro-sequencers$. Often the hardware translation units (binary decoders) are actually formed by a number of independent blocks working in parallel. Some of this blocks can be fast combinational blocks, while other can be based on sequential logic or microcontrollers. In this context the term microcontroller is not synonymous of processing core, as can be found somewhere, but it just identifies a microprogrammed control units. A microprogrammed control unit is a controller where the various configurations a particular set of control signals can assume over time is stored in a special memory, usually a read-only (ROM) memory. In principle the microprogrammed control unit is the grandfather of instruction based control units, which stands behind any modern microprocessor. This is why sometimes the term microcontroller is considered synonymous of microprocessor. In modular binary decoders usually happens that the fast combinational decoders are used to translate *target* instructions which do not require long sequences of *host* instructions for being emu-

lated, where micorcontrolled decoders come into play. Since the chance to encounter one of the complex instruction in a typical program is quite low, the increase in latency of ROM based decoders can be tolerated. Resorting to ROM-based decoders is required because the imp0lementation cost of implementing in a State Machine the complex control mechanisms required for the emulation of the most complex *target* instructions would be too high. Fig. 8.6 , extracted from an AMD patent, describes the high level architecture of a decoder of this kind.

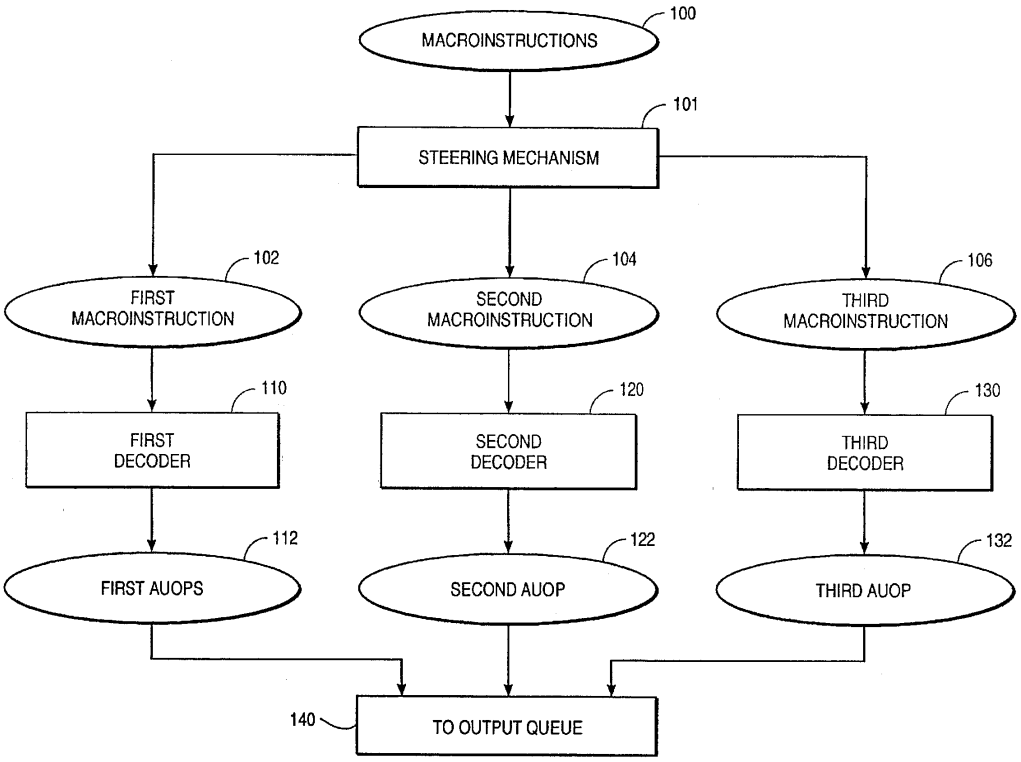


Figure 8.6: Typical parallel decoder.

In Fig. 8.6, *auops* represent *host* instructions while *macroinstructions* are the *target* instructions.

Fig. 8.7, taken from an Intel patent, shows a more detailed block diagram of an heterogeneous binary decoder as found in many superscalar processors.

With reference to Fig. 8.7 , apart of the x86 specific features as the need to provide support for instructions varying in length, is it possible to identify 6 different parallel decoders: 5 combinational (PLAs) and the 6th based on a microsequencer. The microsequencer comprises an Entry Point Generation Unit, a Microcode Se-

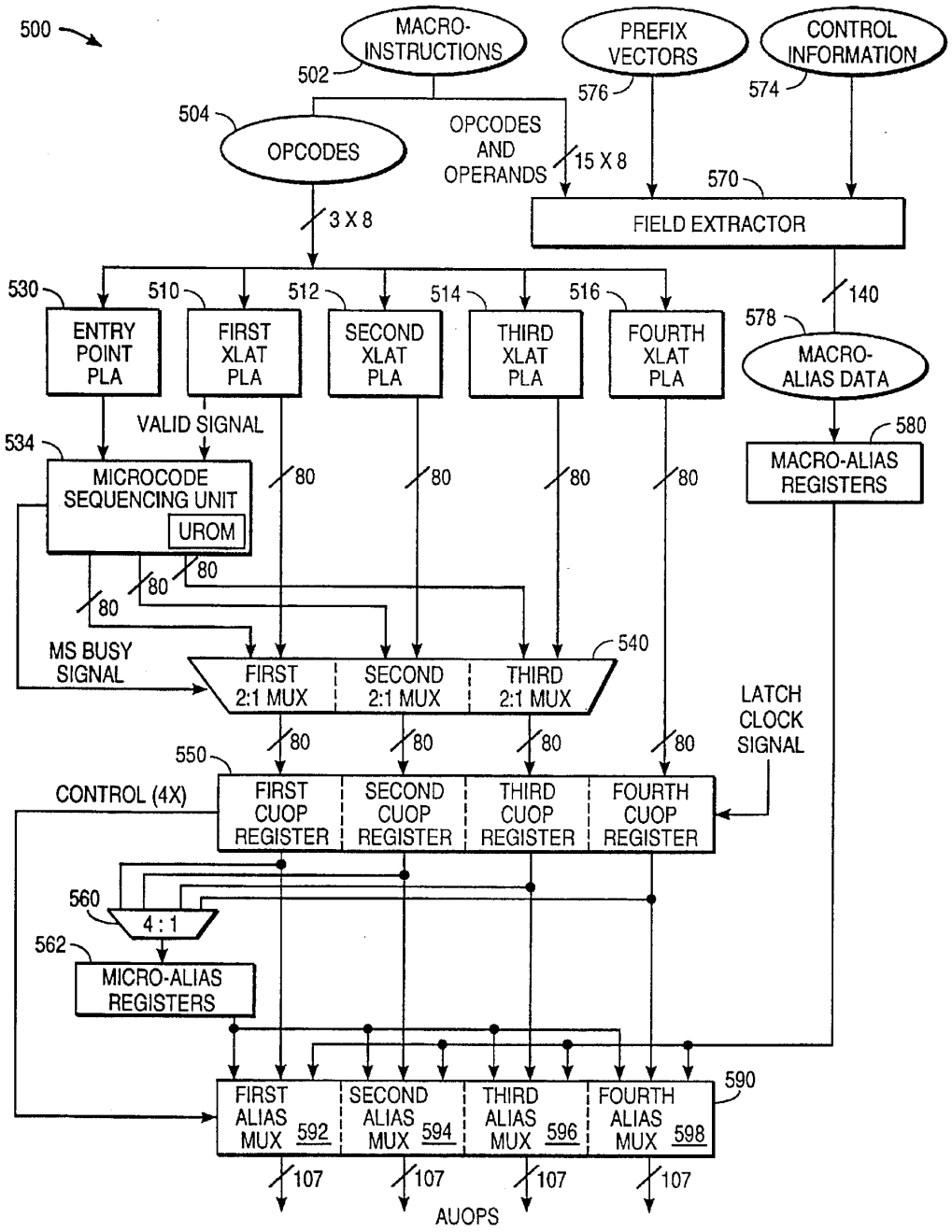


Figure 8.7: Detailed block diagram of a parallel decoder.

quencing Unit and a ROM. The Entry Point Generation Unit generates the starting address of the *emulation routine* stored in the ROM and the Microcode Sequencing

Unit regulates the flow of control of the same *emulation routine*. The *emulation routines* stored in the ROM are not actually sequences of full featured *host* instructions ready to be executed by the functional units. They are more *templates* of instructions. Emulation routines have to reproduce the semantics of a given *target* instruction in input. There are slots of the input instructions that depend on input data, the value of an *immediate* field for example, or the value of a memory address. For this reason there is the need to pass some data fields directly from the format of the input instruction to the *emulation routine*. Therefore, there is the need to provide a block that takes care of extracting those field from the input instruction (the Field Extractor in Fig. 8.7) and of some other block (typically a mux tree) for merging this data with the *emulation routine* template fetched from the ROM. In this context, the *emulation routine* specifies the behavior, while the input dependent information is just transferred from the *target* instruction in input. The task of the field extractor can be very complex sometimes. This is particularly the case when the format of the instructions of the *target* ISA is not regular. An Instruction Set is regular when specific bit slices inside the instruction word have the same meaning for all the instructions. For example, suppose to have an instruction format where the 4 most significant bits specify the instruction's opcode. If this is true for all the instructions then the instruction format is regular, at least with reference to the opcode. If this property holds for all the different slots that can be identified in the instruction format, for example the slots that specify input and output registers, the Instruction Set is said to be regular, and the task of Field Extraction can be accomplished without too much overhead. If instead there are slots in the instruction format that violate this rule the Field Extraction process can quickly became overly complex. The ARM ISA, the *target* Instruction Set in the context of the work carried on in the course of this thesis, exhibits good regularity apart from some corner case, where the extraction of critical data fields becomes more complex.

8.2.2 Trace Cache.

In the discussion above, *target* instructions are translated every time they are fetched from memory. Suppose that the processor is executing highly recurrent code, where the same instructions or code fragment is executed most of the time. In this case repeating the translation every time would be a waste both in terms of performance and power consumption. In terms of performance because the binary translation process has significant latency, sometimes there is the need to resort to pipelining, and in terms of power because the complex computations involved are very energy

consuming. Avoiding to repeat the translation process in these cases would definitely improve performance. Placing a dedicated high speed memory buffer downstream of the decoding logic for storing the code fragments (already translated) with higher execution rates and searching this buffer in all future program references to that code fragment would definitely improve performance. This is exactly what a *Trace Cache* does (see Fig. 8.8).

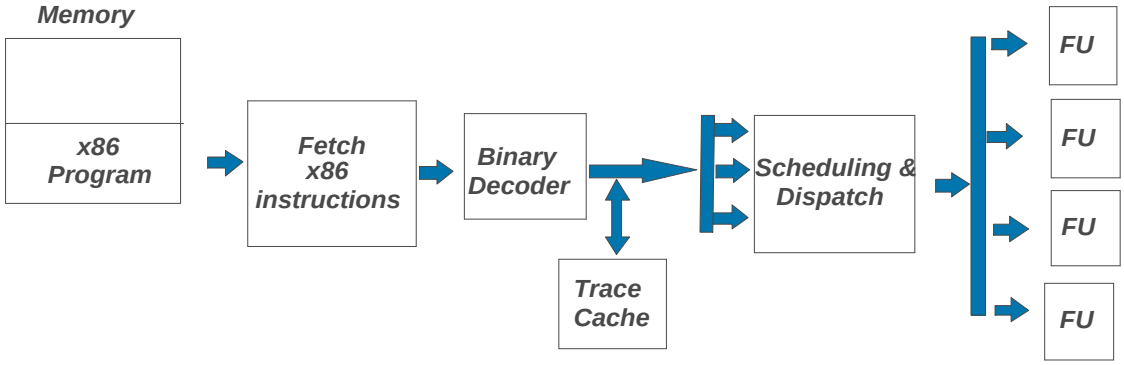


Figure 8.8: Binary decoding with Trace Cache.

The *Trace Cache* technique is widely used in modern superscalar processors. Translated instructions (in *host* format) are stored in the *Trace Cache* in execution order. However, maintaining a *Trace Cache* is not an easy task. If for example the Nth time a code fragment is fetched from memory the control flow inside the fragment changes, the core cannot continue fetching instructions from the *Trace Cache* because it is now inaccurate. There is the need to provide some recovery mechanisms in order to stop fetching instructions from the *Trace Cache* and restarting to translate the instructions fetched from memory. To be efficient, this process must be supported by some branch prediction hardware.

8.2.3 Microcode sequencers.

Microsequencers (Fig. 8.9)are used to translate complex *target* instructions into a sequence, often a long one, of *host* instructions.

Hereafter the basic principles of microsequencer operation are described. The input *target* instruction is scanned and the entry point (starting address) of the *emulation routine* is produced. The actual access to the ROM memory can be organized in different fashions, according to different design styles. Basically there are two alternative microsequencer organizations. The first one is based on the so

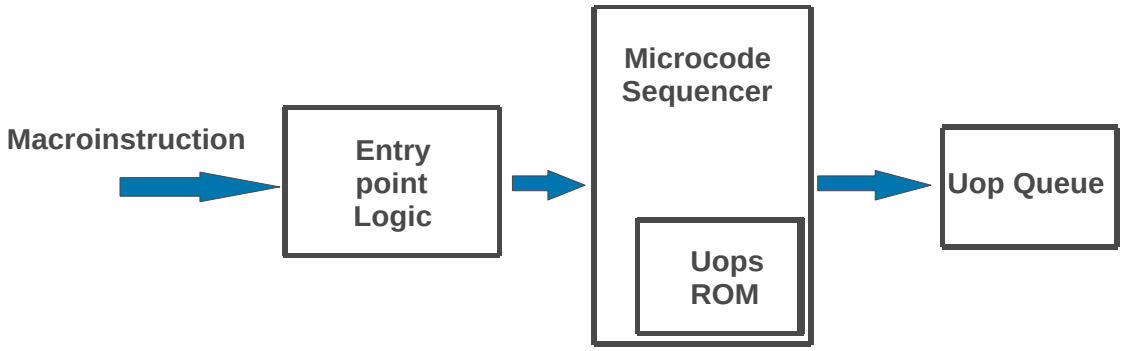


Figure 8.9: Micro Sequencer.

called *Branch Uops*, branch instructions that are only for internal use inside the microsequencer, as depicted in Fig. 8.10. They must not be confused with actual branch instruction that regulate the control flow of the software program. The second approach, described in Fig. 8.11, is to write beside to each location of the sequencer's ROM the address of the next ROM location to read. The choice among these two approaches is determined by the application at hand, the properties of the *target* and *host* Instruction Sets, etc. In general, the second solution is less flexible than the first.

In case *Branch Uops* are used, the default behavior of the *emulation routine* is to continue fetching from the next location inside the ROM.

In next uop configurations the end of the *emulation routine* is specified by a dedicated bit, which is stored in every ROM location. Otherwise in branch uops configurations the end is specified by a special configuration of the branch target field. A further distinction exists among ROM architecture with one of two levels of indirection. The difference is that in the second case the first access produces only an address for the second level ROM, as shown in Fig. 8.12. Also in this case the choice among the two available options needs to be tailored on the properties of the Instruction Sets involved in the translation process (*target* and *host* ISA).

8.3 Software-oriented approaches for binary compatibility.

Modern software-oriented approaches to binary translations try to improve performance of traditional emulation systems using aggressive optimization algorithms in order to fill as much as possible the performance gap with native code execution.

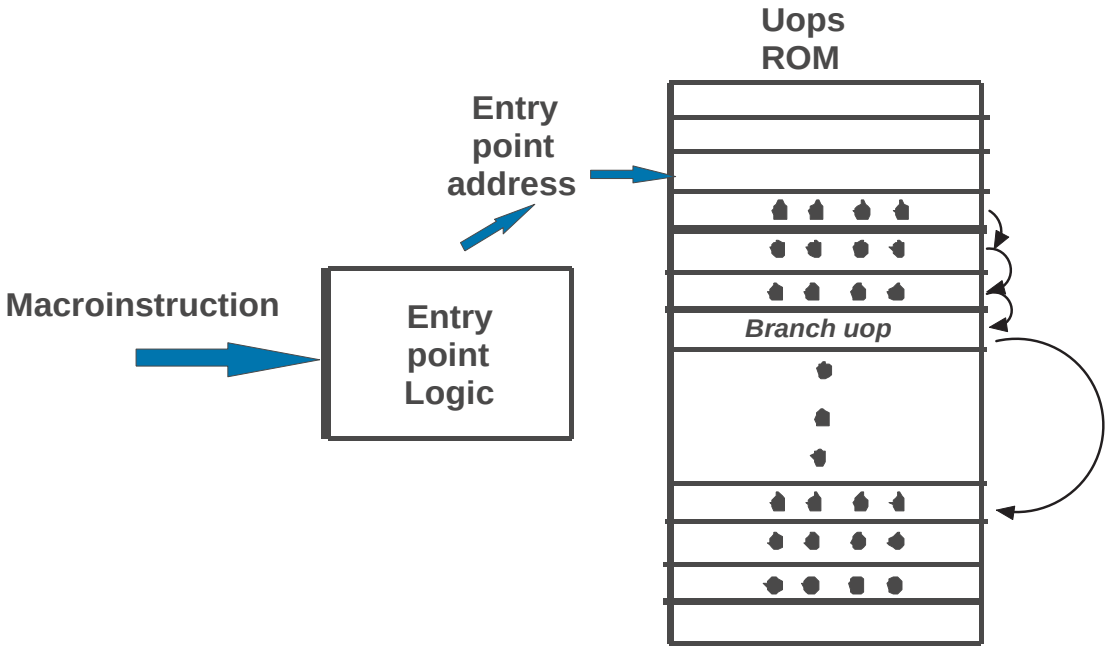


Figure 8.10: Managing control flow inside the microsequencer with branch uops.

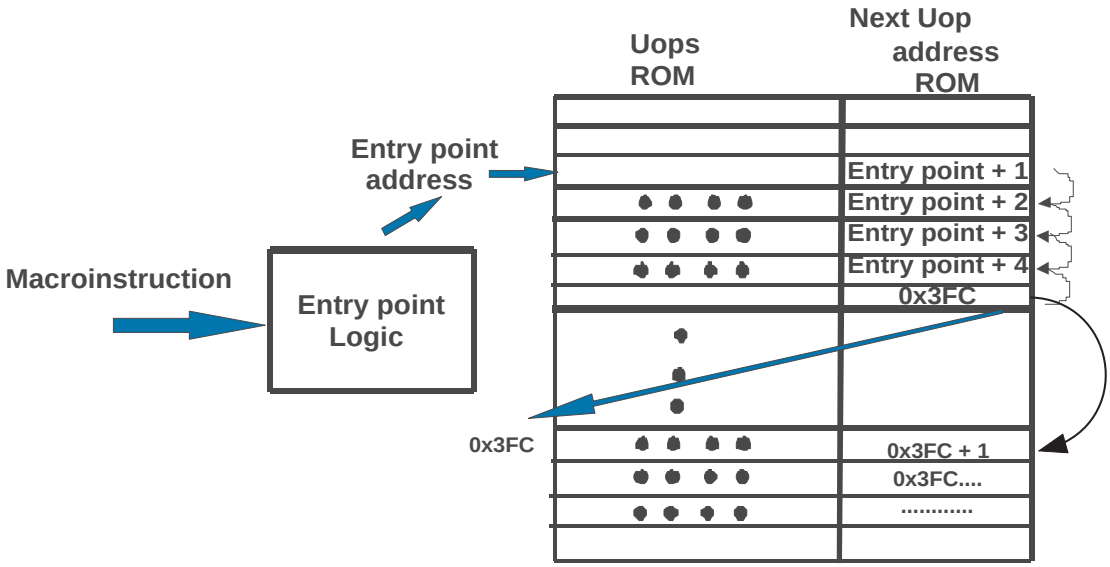


Figure 8.11: Managing control flow in microsequencer with next uop address field.

Typically a software binary translation framework uses a combination of traditional emulation with advanced techniques as caching of translated code and aggressive

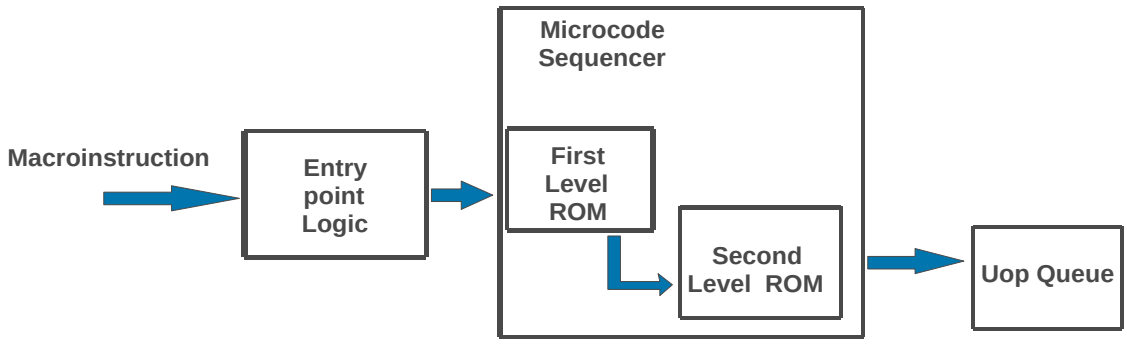


Figure 8.12: Simple two level ROM.

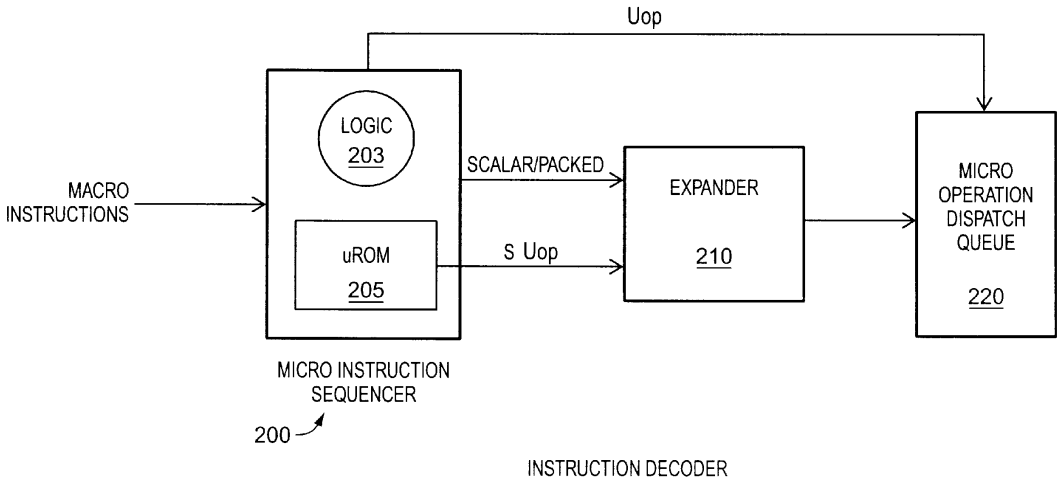


Figure 8.13: Micro Sequencer with intermediate instruction format.

optimization of translated code fragments. Moreover, caching of translated code fragments can be memory-based or supported by special hardware units like the *Trace Cache* introduced in the previous chapter.

Sometimes the distinction between *emulation* and *binary translation* is a bit confusing. In general *emulation* is a superset of *binary translation*. In fact, in typical emulation systems, once a given *target* instruction or code fragment has been translated it is immediately executed by the underlying machine. *Binary translation* instead can be performed on a given section of *target code* and transla-

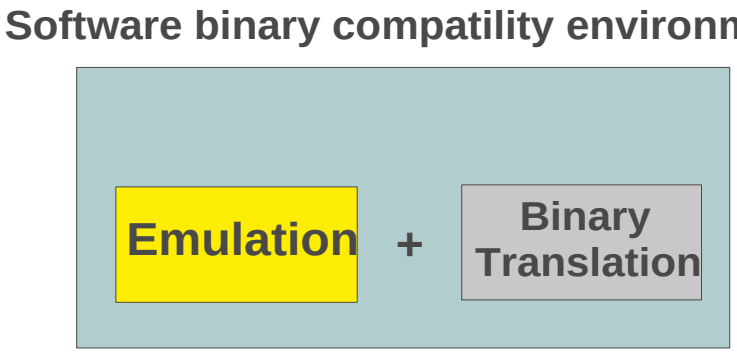


Figure 8.14: General view of a software binary translation platform.

tions can be placed in some memory buffer, in order to be executed later or further optimized. The memory area where translations are placed is sometimes called *Queues Cache* or *Fragment Memory*. The basic advantage of separating binary translation of a code fragment from its execution is that binary translation can be executed only once, the first time that particular code fragment is fetched from program memory, and being executed many times. In fact, when the program's control flow reaches again the point of the translated segment of code the system instead of translating again the instructions can jump to the appropriate translated image stored in the *Fragment Memory* and execute native *host* code. The higher is the execution rate of that particular program fragment the higher is the performance benefit of this optimization techniques over classical emulation techniques. If the rate of execution approaches infinity the performance of this software translation techniques matches that of native code execution. In this case in fact binary translation and optimization techniques can be combined with the already sophisticated software environment of VLIW processors, in order to boost the performance of both components. Typically the piece of software where binary translation and optimization techniques are implemented is called *Dynamic Compiler*. An important point that is worth remarking is that in this framework binary translation of *target* programs is something that happens at run-time, not statically at compilation time. *Dynamic Compilers* in conjunction with VLIW processor are particularly appealing exactly because the addition of a run-time software environment can significantly improve the performance of the optimization algorithms of VLIW compilers that in general have to accomplish their task counting only on statistical information, gathered profiling the software application, which is an inherently inaccurate process. As explained in Chapter 6, the VLIW optimizer in order to find the *hot spots* of the programs that need to be aggressively optimized relies only on statistical informa-

tion about the behavior of branch instructions. If the information on the behavior of branch instructions was instead gathered at run-time, upon the real execution of the code, it would be much more accurate. Therefore the chance of branch misprediction would be significantly reduced and accordingly the performance of the VLIW schedule would be increased. The more the behavior of branch instructions in unpredictable the higher is the benefit of relying on run-time information. If instead the behavior of branches is highly predictable, even at compilation time, the overhead of managing a complex software run-time monitoring environment is not justified. These properties depend basically on the particular software applications that have to be executed.

8.3.1 DEC FX!32.

The Digital FX!32 has been one of the first attempts where techniques of emulation and binary optimization have been used in combination. The objective of the project was to support the execution of x86 programs the DEC Alpha platform. The process of binary translation was executed in background, in order to minimize its overhead. Fig. 8.15 shows a block diagram of FX!32.

DEC FX!32 required that both the x86 *target* application and the underlying Alpha machine were used with the Windows NT operating system. Calls from x86 code to Windows NT were supported through a technique called *jacketing*, that consists in encapsulating operating system calls triggered by x86 code in the corresponding operating system calls of native Alpha code.

Management of x86 condition codes in DEC FX!32.

A very interesting technique has been provided in DEC FX!32 for handling the setting of x86 condition codes. Many of x86 instructions potentially can set condition bits, but statistically only a few instructions on a given program fragment actually use them. The technique used in DEC FX!32 tries to exploit this fact by means of a *lazy evaluation* mechanism. In this way, computation of unused condition bits is virtually avoided, with significant savings in terms of extension of the emulation routines.

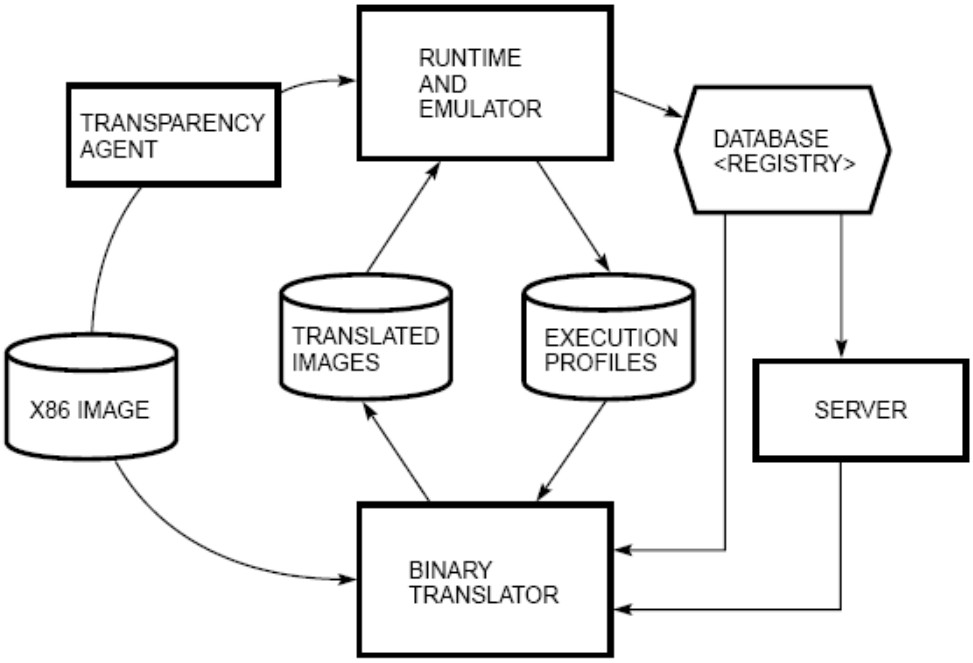


Figure 8.15: Block diagram of DEC FX!32.

8.3.2 Examples of binary translation techniques in the context of VLIW processors.

As already explained, benefits of binary translation techniques are larger when used in conjunction with VLIW processing cores. Further benefits can be achieved if the VLIW core supports in hardware particular features specifically designed for speeding up the binary translation/dynamic compilation process. In this section an enhanced VLIW architectures developed from IBM, designed for optimized support of dynamic translators, will be introduced.

DAISY.

The acronym Daisy stands for Dynamically Architected Instruction Set from Yorktown. Fig. 8.16 shows the high level architecture of a typical Daisy system.

The areas of Fig. 8.16 marked in black regard blocks of the system were execution of native VLIW code is involved. The *target* instruction set in Daisy is the PowerPC

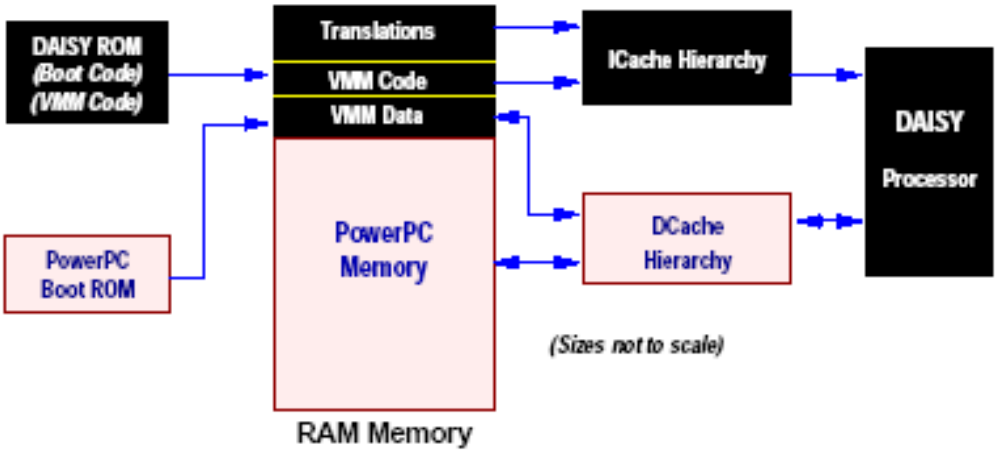


Figure 8.16: Daisy's structure.

ISA, while the *host* ISA is represented by the native VLIW ISA. The presence of the Dynamic compiler is completely hidden to the PowerPC software applications. The Dynamic compiler works seamlessly translating PowerPC instructions in native code. In the following the basic operation of the system is outlined. At system boot, control passes to a software module called VMM (Virtual Machine Display console). After initialization of the system has been finished control is then passed to the interpreter/emulator that begins translating and executing PowerPC code fragments included in the boot routine. The boot routine is executed under control of the VMM. After the boot routine finishes its execution, and kernel of the operating system has been loaded in memory, Daisy begins interpreting and executing the operating system code. In practice, execution of every PowerPC code fragment, application or operating system, happens under control of the Daisy dynamic compiler and run-time environment (VMM). The instructions coming from the boot ROM come in this case executed under the control of the VMM. The VMM supports two different execution scenarios: 1) when PowerPC code is interpreted and immediately executed (as in standard emulation) and 2) when the system executes directly from memory translated code fragments corresponding to PowerPC code translated previously. The more time a system of this kind spends executing already translated code from memory the better is the performance. Typically, translation of PowerPC code happens at *basic block* level. After the dynamic compiler identifies the boundaries of a *basic block* in the PowerPC code it starts generating the corresponding segment of translated code. At this point the VMM can apply further optimizations,

as merging translations corresponding to different basic blocks in order to maximize the parallelism available in the code and exploitable by the scheduler. Aggressive code optimizations must be performed with care, because the cost of the recovery mechanisms is relevant.

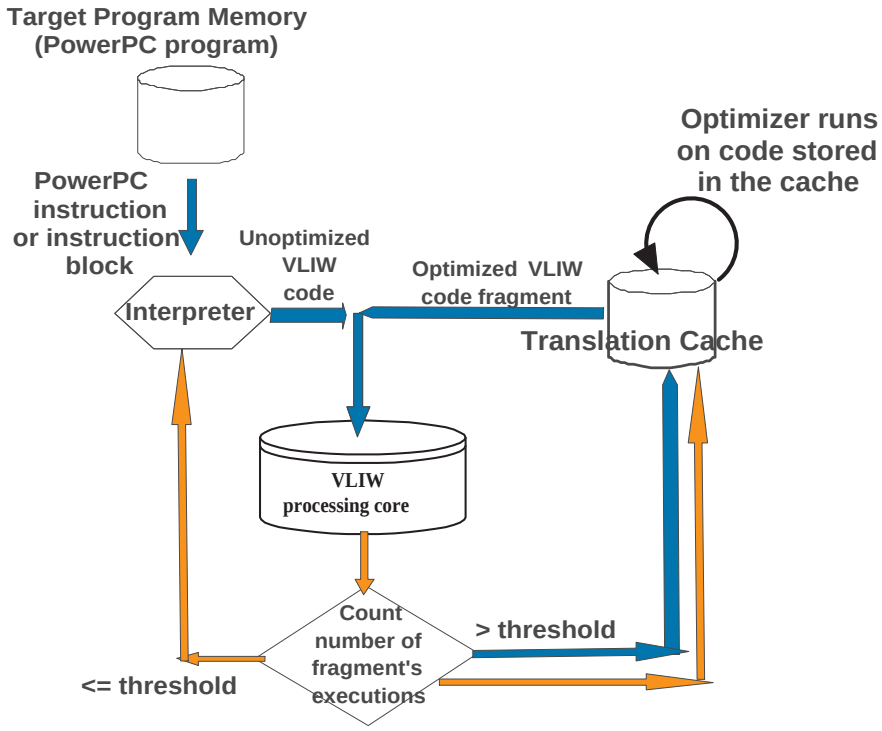


Figure 8.17: VMM control flow diagram.

Translation units used in Daisy are the *tree-regions*. Fig. 8.18 explains the differences between the tree-regions and the *traces*, introduced in Chapter 6. Unlike *traces*, *tree-regions* include segments of code that belong also to different paths of the program. This is done in order to decrease the overhead incurred in case of misprediction on the behavior of branch instructions, at the expense of reducing the efficiency of the code included in the translation units (because part of the translated instruction will be certainly never executed, belonging to different paths of the CFG).

In Daisy the entity of the optimizations applied to translated code is directly proportional to the execution rate of that code. If the code will be executed extensively, then it is worth spending time and resources for aggressive optimizations.

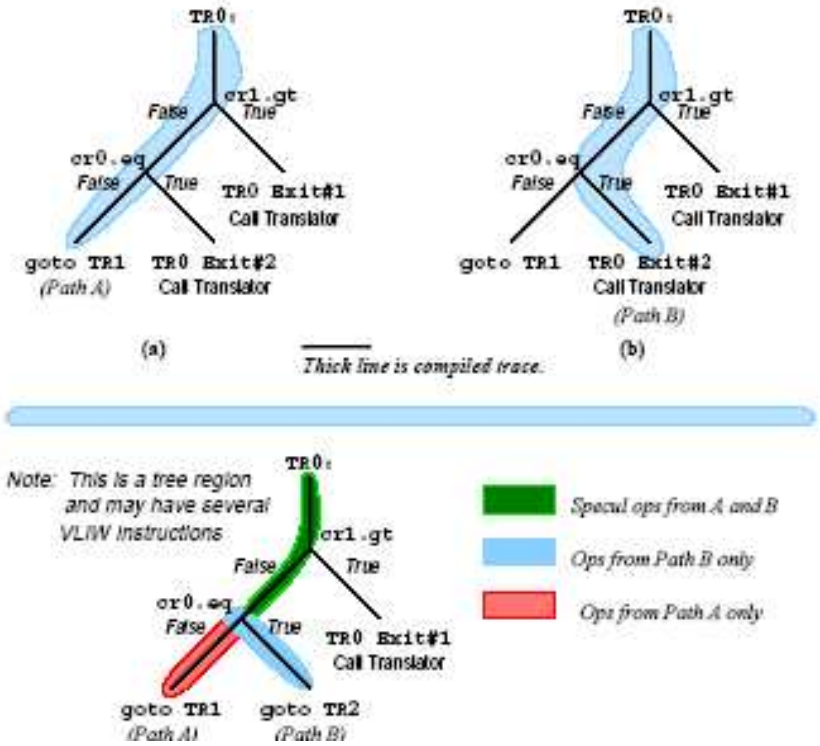


Figure 8.18: Trace and treeregion

Exception handling. A critical aspect of the Daisy system concerns the management of exceptions and interrupts triggered during execution of translated code (not during execution of the VMM itself). When an exception is encountered in a processor, is important for the system to be able to rollback to its state at the moment immediately preceding the exception or interrupt was collected. In this context, the optimization techniques that the dynamic compiler applies to translated instructions can in principle undermine the ability of the system to find out exactly which PowerPC instruction produced the exception and to go back to the state as it was before the exception or interrupt occurred. To avoid this problem in Daisy is employed the policy that no matter what optimizations the dynamic compiler applies to the translated code sequences, update of the system's architectural state happens in a way that respects the original order of PowerPC instructions. This is accomplished using two different sets of registers: 1) architectural registers and 2) working registers. Architectural registers are used to store the architectural (official) state of the system. Working registers instead are used to support the

aggressive optimization techniques applied by the dynamic compiler, which makes extensive use of register renaming for exploiting as much parallelism as possible. Safety during exceptions or interrupts is accomplished providing that the value of working registers is copied to the architectural registers only at PowerPC instruction boundaries. This process happens under direct control of the VMM.

Chapter 9

Extension of ST230 architecture to binary compatibility with ARM ISA.

The concepts and policies described in the second part of this thesis have been applied to the problem of providing support for the execution of ARM programs inside the ST230 processor of the Lx family, developed by STMicroelectronics. For confidentiality reasons the description that follows could not be much detailed. Moreover, since much of the work has been carried on with help of proprietary software tools and technologies developed by STMicroelectronics, and made available under NDA, it will not be possible to show material as sections of the source code developed, if not in rare cases. Therefore the following description will focus on the architecture, mostly at an high level of abstraction, of the proposed solutions for providing binary compatibility with ARM code inside the ST230 processor. For specific details about the properties of ARM instructions refer to the ARM Architecture Reference Manual [3].

9.1 Analysis of architectural choices.

The spectrum of possible solutions for providing binary compatibility among ARM and Lx instructions is quite large, as seen in the previous chapter. The main available options in synthesis are:

1) Full software oriented approach. It requires using a dynamic translation/compilation system, that in principle could run under the operating system

or above the operating system. ARM code fragments in this case are translated by the software in Lx code fragments. Thus, in this case no ARM instructions are actually fetched by the pipeline of the ST230 processor. Moreover, in this case we can have two different subcases:

- Specific extensions in the ST230 pipeline for speed up of translation process and forcing *precise exceptions* in hardware.
- No particular hardware support for binary translator in hardware, apart of the definition of a suitable mapping for the architectural space of the *target* (ARM) ISA.

2) Full Hardware oriented approach. It implies modifying hardware front end of the ST230 processor, with the addition of a custom designed *Binary Decoder* able to convert incoming ARM instructions into one or more Lx instructions. Unlike in the above case, now ARM instructions are fetched from the ST230 pipeline. Further choices available are:

- Implementation of a purely combinational decoder, single or parallel.
- Implementation of a parallel decoder comprising also a microsequencer for translation of complex ARM instructions.
- Full decoder implementation based on a microsequencer.

3) Hybrid hardware/software approaches. In this case part of the decoding process is accomplished by an hardware decoder and remaining part is done in software. Usually in hardware is done the decoding of the most commonly executed instructions, while in software is done only the decoding of the most rare and complex ones. There is also the possibility to decide how exceptions or interrupts triggered by the execution of translated code are handled:

- Support in hardware can be provided, including directly in the processors pipeline some place for storing the recovery routines that are in charge of handling exceptional conditions.
- A purely software strategy can be adopted, where recovery routines have to be fetched every time from the system memory.

To choose the right approach to the problem there is the need to understand the constraints posed by the features of the ST230 processor, by the properties of the software programs that are supposed to take advantage of the binary translation capabilities and by the price that can be payed in terms of performance of the system and hardware resources. The ST230 processor has been designed for applications

like multimedia Systems-on-chip targeted at the market sectors of set-top-boxes, mobile phones, multimedia content players, etc. Applications that are supposed to run on the ST230 processor are basically of two categories:

Heavyweight applications. These include the software tasks in charge of coding and decoding in real-time multimedia content, managing the communication channels in wireless devices, playing multimedia content to displays at an high data rate.

Lightweight applications. These include management of Graphical user interfaces and of all little application built around GUIs, as calculators, software for handling calendars and similar utilities. In this context is worth noting that while the user of the appliance is not usually aware of the Heavyweight tasks, because they are only experienced indirectly by the user in terms of performance of the system, it is instead directly exposed to the GUI and utilities. This is why the last ones tend to be standard across models proposed by different companies.

It can be safely assumed that software tasks that will take advantage of the ARM execution capabilities will belong to the second category. Applications that instead fall in the first category will be almost certainly compiled with the Lx toolchain and executed natively as Lx code, because the strict real-time constraints posed by their execution are not compatible with the overhead that is almost inherent to any binary translation mechanism. Furthermore, for the applications targeted by the ST230, the use of a complete dynamic compilation framework as the one used in Daisy seems not to be required at the moment (remember the Daisy was developed as a general purpose computer). STMicroelectronics has developed in partnership with HP labs a software run-time environment called DELI, which has many features in common with the VMM in Daisy. It is intended mainly for optimizing software execution in embedded platforms, but can also be configured to include dynamic compilers (also called *Just-in-time*, JIT, compilers) for binary translation tasks, even with all the overheads of a pure software approach. However, at the moment the approach of supporting the translation of ARM code plugging into DELI a proper *Just-in-time* compiler has been discarded, because a more efficient solution would be preferred. At the same time another strong constraint has been the requirement for the solution to minimize the impact on the ST230 microarchitecture. The reason for this is that today one of the most time consuming tasks when designing a System-on-chip and even more when designing a microprocessor is the verification task. Modifying critical parts of the processor that had already taken months for being validated could have been dangerous. For example any modification to the Load/Store unit (see Chapter 6) was strictly forbidden, because that block took several months

in verification time before being validated. This is why aggressive changes to the ST230 processor like adding new ad-hoc registers for mapping the ARM architectural state in order to optimize the execution of the emulation routines have soon been discarded (see Fig. 9.1 for an overview of the possible solutions for the mapping of ARM architectural state). Instead has been preferred solution *b* in Fig. 9.1.

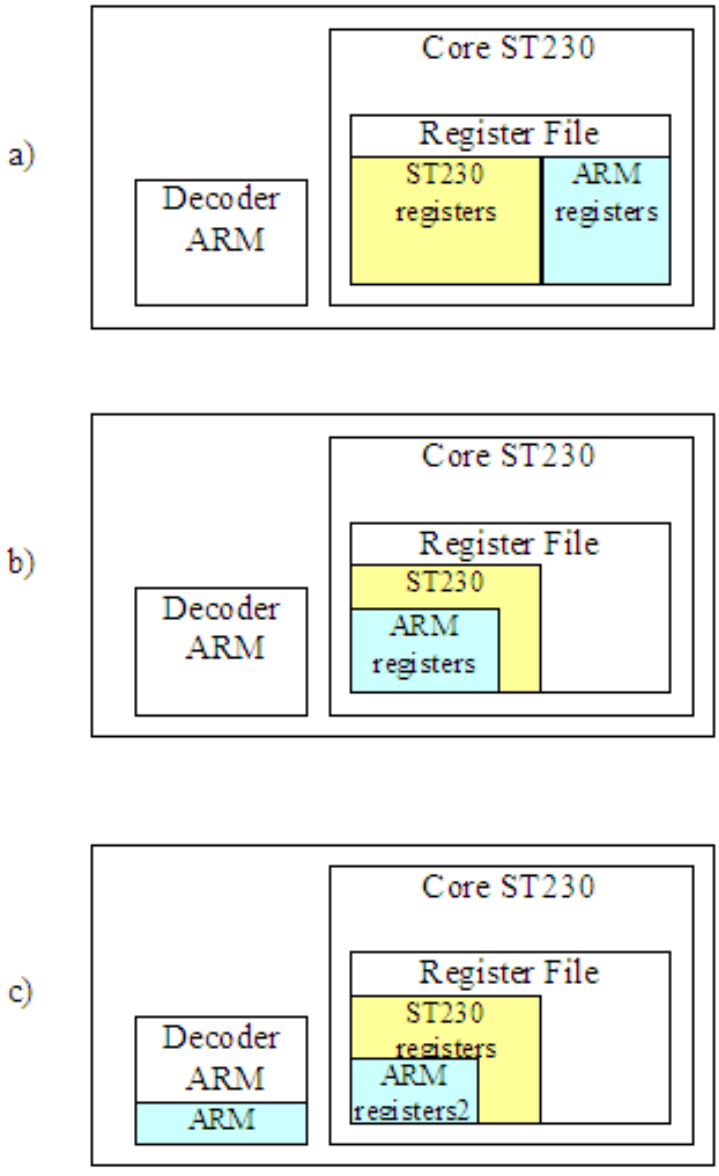


Figure 9.1: Mapping options for ARM architectural state.

Among the resources included in the ARM architectural state [3] is comprised also the ARM PC (the program counter that tracks the execution of the ARM code inside the ST230 processor). Unlike the program counter of the ST230 processor, that can be updated only as a side effect of the execution of few Lx instructions, the PC in ARM processors is included in the general purpose Register File and can be directly updated by many variants of common ARM instructions. Since the execution of ARM instructions inside the ST230 happens by means of the *emulation routines* generated by the ARM decoder, and this *emulation routines* once they exit the ARM decoder are just sequences of ordinary Lx instructions, the only way for the execution of ARM code fragments for properly updating the ARM PC is to reserve to the ARM PC one of the general purpose registers inside the ST230 Register File. Other solutions like placing an ad-hoc register inside the ARM decoder for acting like the ARM PC are not viable, because this new resource would not be visible to Lx instructions without major changes to the ST230 pipeline. For all these reasons the architecture configuration that has been chosen as a starting point includes an *ARM Binary Decoder* implemented in hardware as a Microcode sequencer. This hardware decoder will be connected to the front-end of the ST230 processor and will be in charge of the translations of the vast majority of ARM instructions. In the first version of the architecture support for the remaining ARM instructions as well as for the *interrupt* and *exception* emulation routine will be provided by means of calls to software routines stored in main memory. In future versions of the architecture will be explored the opportunity of including also support for these feature inside the Decoder in hardware. Refer to Fig. 9.2 for an high level view of the architecture chosen.

It is worth noting that although some of the solutions employed here are already known in the academic and industrial literature, as described in the previous chapter, this is to our knowledge one of the first attempts of employing hybrid hardware/software binary translation techniques in the field of Systems-on-chip. The strong real time constraints that must be respected by processors in the context of SoC based system require a completely new research effort in understanding which techniques and solutions for binary compatibility are best suited for the new environment. The purpose of this work accomplished in the context of the thesis is to make a contribution in this direction, even with all the restrictions imposed by the strong industrial setting of the project.

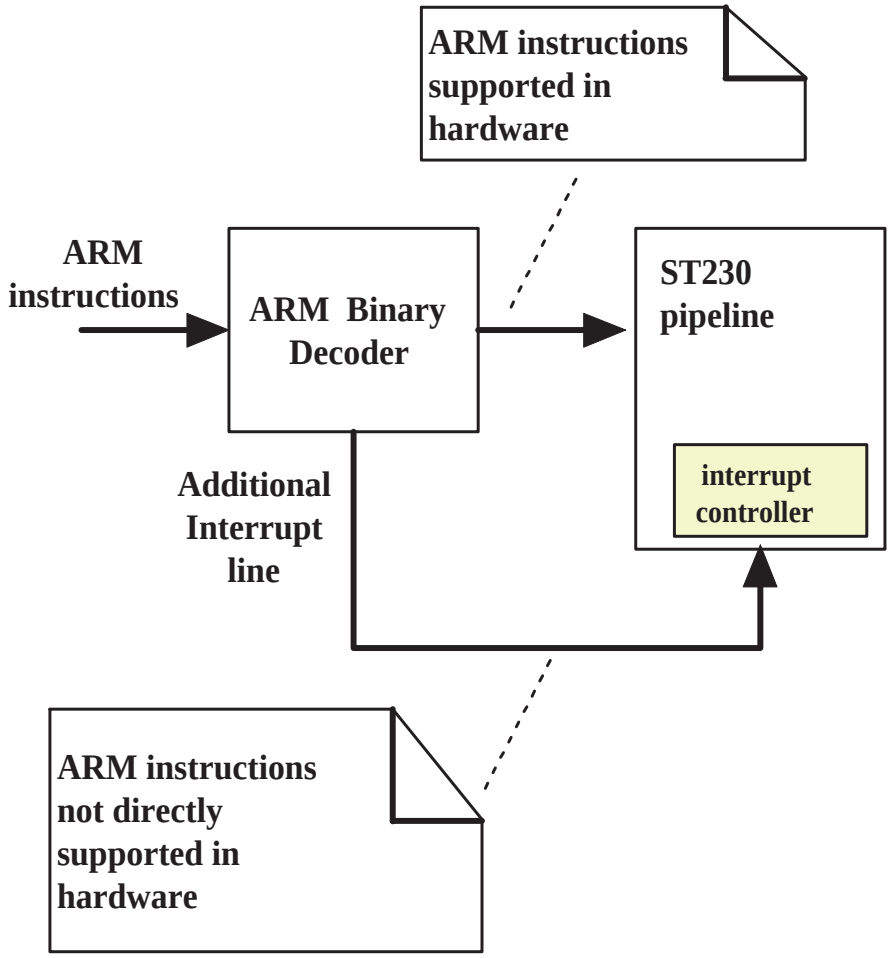


Figure 9.2: Binary translation system architecture.

9.2 Implementation.

Hardware support is provided only for part of the ARM ISA. When the *ARM Binary Decoder* fetches from memory an unsupported instructions it triggers an interrupt toward the ST230 *Interrup Controller*, that then processes the interrupt as a normal ST230 interrupt (call to *exception handlers* in memory, etc.). The implementation of the overall system undertaken in the course of this thesis involves the development of the major component of the binary translation system described above:

Hardware part. The development on the hardware side has required the implementation in the SystemC language, at an RTL level of abstraction, of the ARM binary decoder. The detailed microarchitecture and source code of the *ARM*

Binary Decoder cannot be disclosed for confidentiality reason, but it is basically based on a Microcode Sequencer. The work to be carried on in the hardware context regarded also the interfacing of the custom developed decoder with the ST230 model made available by STMicroelectronics under NDA. It is a model coded in the C language, but that provides support for detailed emulation of the ST230 processor architecture behavior.

Software part. The development on the software side of the project included mainly the creation of the *emulation routines* stored inside the *ARM Binary Decoder*. These emulation routines are sequences, sometimes quite long, of native Lx instructions in charge of reproducing precisely the behavior of the set of ARM instructions which have to be decoded in hardware. The development of the emulation routine was probably the hardest task. The work could not be undertaken by hand, therefore it was accomplished through the help of two advanced proprietary software tools of STMicroelectronics: 1) *Deli*, a complex run-time environment for the optimization of code execution in VLIW processor, but also suitable for off-line optimization of software functions, and 2) *Chorizo*, a powerful parametric generator of Instruction set Emulators.

In the following the discussion will focus mainly on the software development work. Fig. 9.3 describes the development flow for the *emulation routines*.

9.2.1 Examples of ARM ISA description with the Chorizo grammar.

A description of the ARMv5 (one the ARM ISA variants) was given with the grammar specified by the Chorizo emulator generator. Hereafter some code snippet taken from the ARM specification file will be shown. Although not all the instructions of ARMv5 ISA are still supported, no supports exists for example at this time for *Thumb* (see the ARM Reference Manual [3] for details about ARM ISA), the Instruction Set specification file is about 6000 lines long. The following code fragments shows the section of the specification where all the *slots* available on the ARM instruction format are specified.

```
1
2  code Fields {
3
4    // ARMv5 ISA fields . the naming conventions are
5    //the same used in the ARM Architecture
```

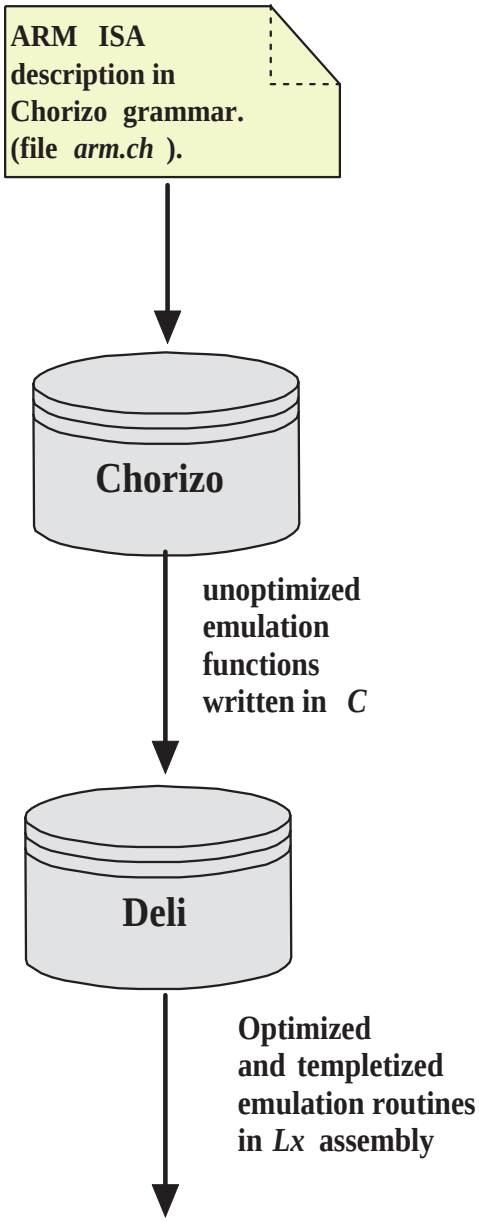


Figure 9.3: Development flow for emulation routines.

```
6    // Reference Manual .
7    //In this first version the Thumb instruction
8    //set is not implemented .
9
10   rotate_imm  4;  // see Addressing modes 1 , #immediate
11   immed_8     8;  // see Addressing modes 1 , #immediate
12   Rm          4;  // see Addressing modes 1
```

```
13  shift_imm   5;  // see Addressing modes 1
14  Rs          4;  // see Addressing modes 1
15  offset_12   12; // see Addressing modes 2
16  shift       2;  // see Addressing modes 2
17  immedH      4;  // see Addressing modes 3
18  immedL      4;  // see Addressing modes 3
19  //Ss            1;  // see Addressing modes 3
20  H           1;  // see Addressing modes 3
21  Rn          4;  // base register
22  Rd          4;  // destination register
23  L           1;  // link bit . tells if the branch instructions
24                  //update the link register
25  signed_immed_24 24; // holds the branch offset
26                      // in B and BL instructions
27  immed1      12; // holds the first part of the
28                  //immed field in BKPT instructions
29  immed2      4;  // holds the second part of the immed
30                  //field in BKPT instructions
31  //H         1;  // H bit in Branch with
32                  //Link and Exchange instructions
33  opcode_1    4;  // found in coprocessor instructions
34  opcode_2    4;  // found in coprocessor instructions
35  CRn         4;  // first register identifier
36                  //in coprocessor instructions
37  CRd         4;  // second register identifier
38                  //in coprocessor instructions
39  CRm         4;  // third register identifier in
40                  //coprocessor instructions
41  cp_num      4;  // number of the coprocessor the instruction refers to
42  eight_bit_word_offset   8;  // see Addressing mode 5
43                              //- Load and Store Coprocessor
44  //P         1;  // see Addressing mode 5 -
45                  //Load and Store Coprocessor and
46                  //also Addressing Mode 4
47  //U         1;  // see Addressing mode 5 - Load and Store
48                  //Coprocessor and also Addressing Mode 4
49  N           1;  // see Addressing mode 5 - Load and Store
50                  //Coprocessor and also Addressing Mode 4
51  W           1;  // see Addressing mode 5 - Load and Store
52                  //Coprocessor and also Addressing Mode 4
53  register_list   16; // holds the list of registers
54                  //to transfer in LDM and STM instructions
55  RdHi        4;  // register identifier used in the long multiplies
56  RdLo        4;  // register identifier used in the long multiplies
57  hbyte 4 ; // generic halfbyte
58
```

```
59   // fixed fields . for definition of
60   //fixed fields see chorizo user manual
61
62   dp_first_opcode 4 fixed;     // the first opcode field
63                                //in data processing instructions
64   dp_secnd_opcode 2 fixed;     // second opcode field in data
65                                // processing instructions
66   mems_opcode 2 fixed;     // principal opcode field for basic
67                            //memory transfer instructions
68   mem_opcode 3 fixed; // principal opcode field in some of
69                       //the memory block transfer instructions
70   mult_op1 7 fixed;   // first opcode field in multiply
71                       //and long multiply instructions
72   mult_op2 4 fixed;   // second opcode field in multiply
73                       //and long multiply instructions
74   br_opcode 3 fixed;  // opcode for branch instructions
75   bkpt_opcode1    8   fixed; // opcode fields of
76                               //breakpoint instructions
77   bkpt_opcode2    4   fixed;
78
79   I   1 fixed; // immediate bit . tells
80                //if the addressing mode uses an immediate value
81   S   1 fixed; // provvisoriamente posso forzare il bit S a zero
82   P   1; // choose between pre or post incrementing
83           //addressing mode in block transfer instructions
84   U   1; // decrement or increment the base
85           //register for calculating the other addresses
86   B   1;
87   // .
88
89   bit 1 fixed; // generic bit
90   hbyte_f 4   fixed;
91   byte_f  8   fixed;
92   threebit_f  3   fixed;
93   // conditional field.
94   cond_field 4    fixed;
95
96
```

The following code fragment shows instead a section of the file where the semantics of an ARM instruction is specified. In this case is an ADC (Add with carry), see [3] instruction with an immediate value as one of the operands.

```
1  c_imm  jitinfo 2 pipeinfo 1 // 1
2    debug  " ADC <REG_D> %d=0x%08x ,<REG_N> %d=0x%08x , <SHIFTER_OP>
```

```
3               0x%08x\n " literal "d,REG[d],n,REG[n],shifter_operand";
4   coded as  cond_field = 0b0000 + dp_secnd_opcode =
5               0b00 + I = 0b1 + dp_first_opcode = 0b0101 + S = 0b0
6               + Rn ~ n + Rd ~ d + rotate_imm ~ v_rotate_imm
7               + immed_8 ~ v_immed_8;
8   {
9
10      uint shifter_carry_out;
11      uint shifter_operand;
12      if (v_rotate_imm == 0)
13      {
14          shifter_operand = v_immed_8;
15          shifter_carry_out = CFLAG;
16      }
17      else
18      {
19          shifter_operand = ((v_immed_8 << (32-(v_rotate_imm * 2)))
20                             | (v_immed_8 >> (v_rotate_imm * 2)));
21          shifter_carry_out = shifter_operand & 0x10;
22      }
23      REG[d]= REG[n] + shifter_operand +CFLAG;
24  }
25  ;;
```

9.3 Results and future work.

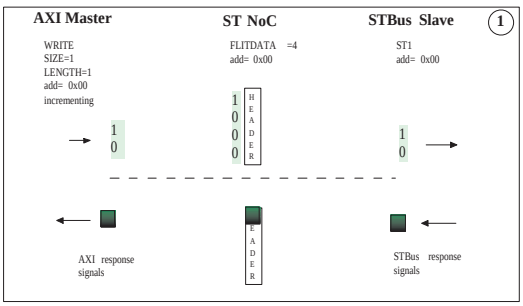
Preliminary results and characterization for the above described architecture cannot be disclosed for confidentiality reasons. Future work will regard extension of the ARM specification file to include also Thumb class instructions. Also the mechanisms for management of exceptions and interrupts generated by the execution of ARM code will be further optimized, maybe inserting some hooks in hardware for the speed up of processes like context-switching. A detailed performance comparison with binary translation approaches based on advanced dynamic compilers will also be undertaken.

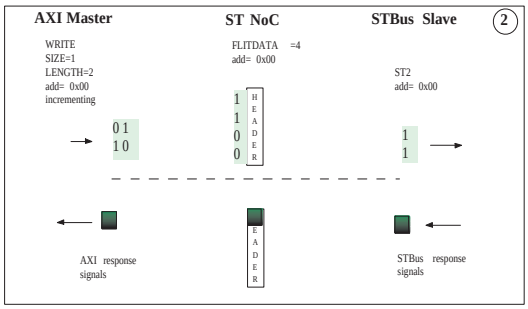
Appendix

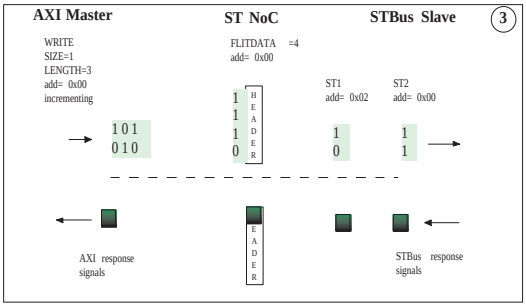
Appendix A

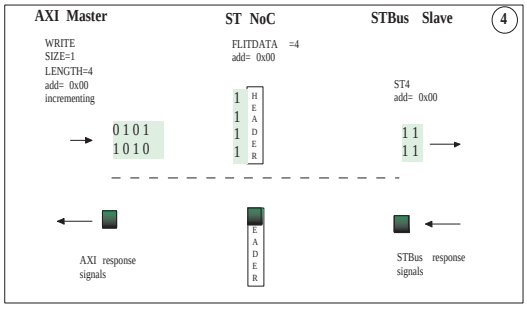
Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms.

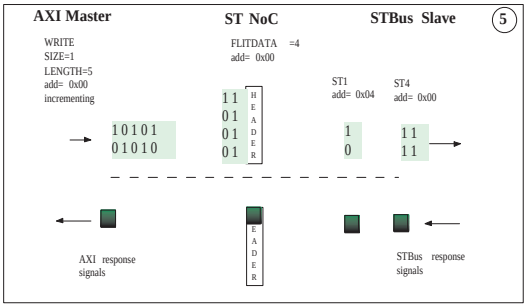
A. Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms.

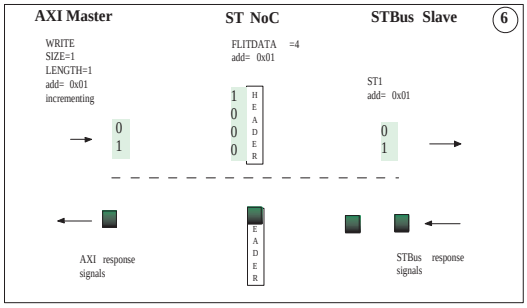


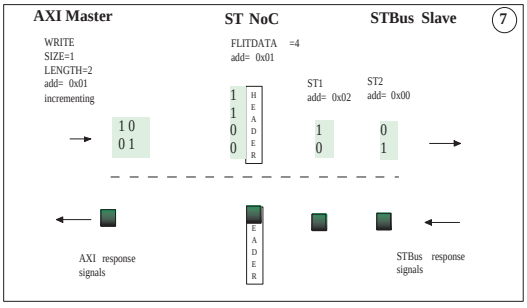


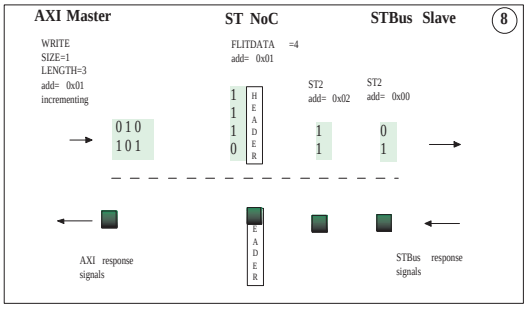


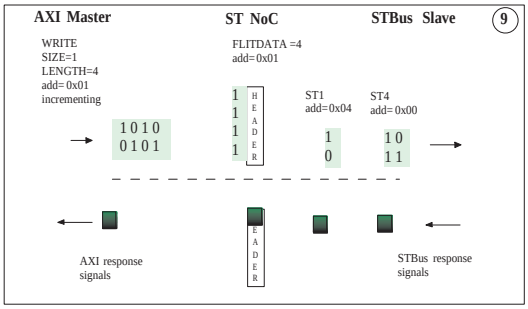


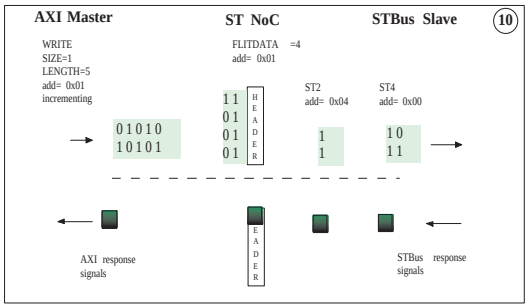


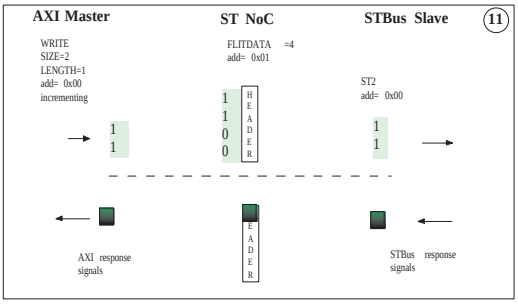


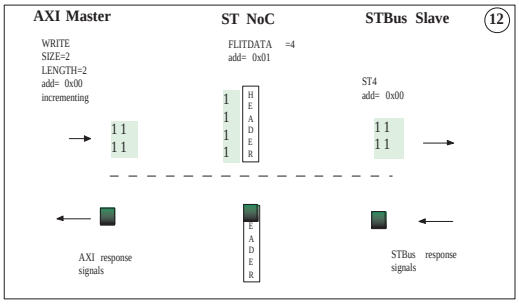


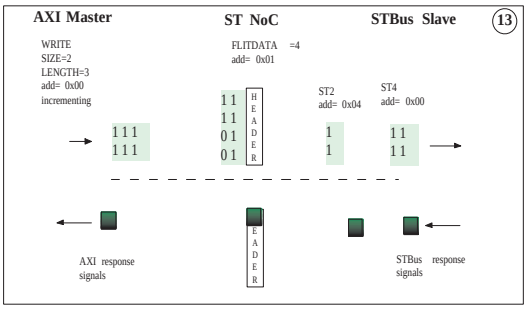


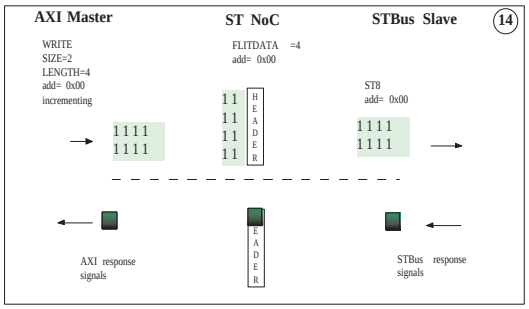

AXI Master
ST NoC
STBus Slave
9
WRITE
SIZE=1
LENGTH=4
add= 0x01
incrementing
1 0 1 0
0 1 0 1
FLITDATA =4
add=0x01
1
1
1
1
HEADER
ST1
add=0x04
1
0
ST4
add= 0x00
1 0
1 1
AXI response signals
HEADER
STBus response signals

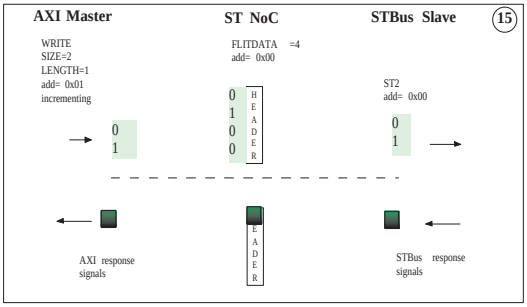

AXI Master
ST NoC
STBus Slave
10
WRITE
SIZE=1
LENGTH=5
add= 0x01
incrementing
0 1 0 1 0
1 0 1 0 1
FLITDATA =4
add= 0x01
1 1
0 1
0 1
0 1
HEADER
ST2
add= 0x04
1
1
ST4
add= 0x00
1 0
1 1
AXI response signals
HEADER
STBus response signals

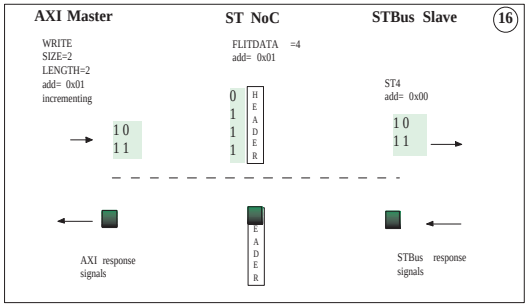

AXI Master
ST NoC
STBus Slave
11
WRITE
SIZE=2
LENGTH=1
add= 0x00
incrementing
1
1
FLITDATA =4
add= 0x01
1
1
0
0
HEADER
ST2
add= 0x00
1
1
AXI response signals
HEADER
STBus response signals

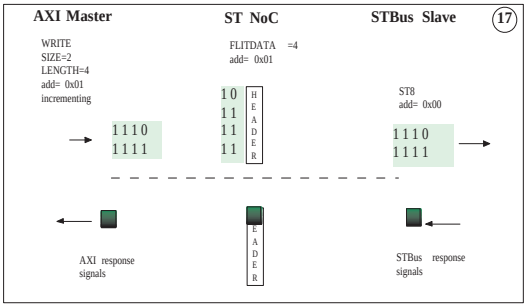

AXI Master
ST NoC
STBus Slave
12
WRITE
SIZE=2
LENGTH=2
add= 0x00
incrementing
1 1
1 1
FLITDATA =4
add= 0x01
1
1
1
1
HEADER
ST4
add= 0x00
1 1
1 1
AXI response signals
HEADER
STBus response signals

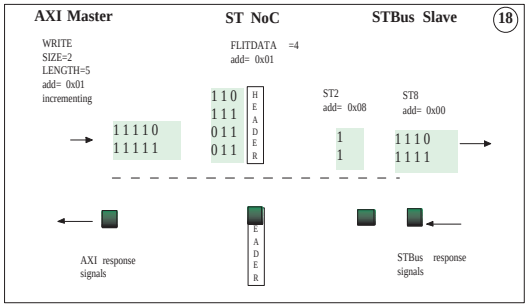

AXI Master
ST NoC
STBus Slave
13
WRITE
SIZE=2
LENGTH=3
add= 0x00
incrementing
1 1 1
1 1 1
FLITDATA =4
add= 0x01
1 1
1 1
0 1
0 1
HEADER
ST2
add= 0x04
1
1
ST4
add= 0x00
1 1
1 1
AXI response signals
HEADER
STBus response signals

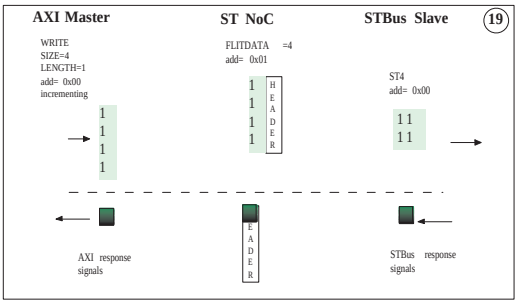

AXI Master
ST NoC
STBus Slave
14
WRITE
SIZE=2
LENGTH=4
add= 0x00
incrementing
1 1 1 1
1 1 1 1
FLITDATA =4
add= 0x00
1 1
1 1
1 1
1 1
HEADER
ST8
add= 0x00
1 1 1 1
1 1 1 1
AXI response signals
HEADER
STBus response signals

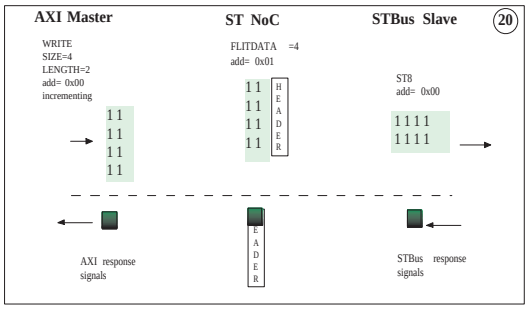

AXI Master
ST NoC
STBus Slave
15
WRITE
SIZE=2
LENGTH=1
add= 0x01
incrementing
0
1
FLITDATA =4
add= 0x00
0
1
0
0
HEADER
ST2
add= 0x00
0
1
AXI response signals
HEADER
STBus response signals

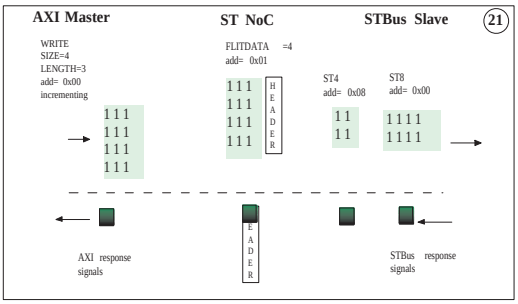

AXI Master
ST NoC
STBus Slave
16
WRITE
SIZE=2
LENGTH=2
add= 0x01
incrementing
1 0
1 1
FLITDATA =4
add= 0x01
0
1
1
1
HEADER
ST4
add= 0x00
1 0
1 1
AXI response signals
HEADER
STBus response signals

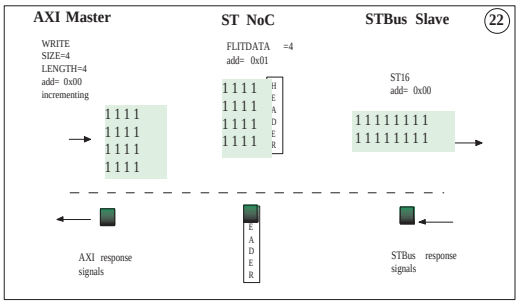

AXI Master
ST NoC
STBus Slave
17
WRITE
SIZE=2
LENGTH=4
add= 0x01
incrementing
FLITDATA =4
add= 0x01
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

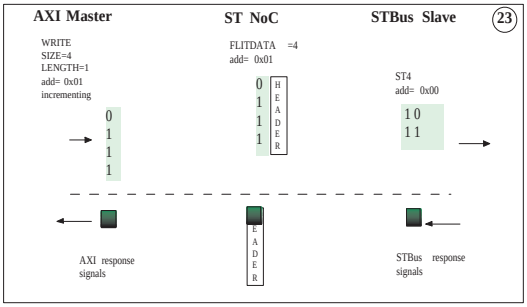

AXI Master
ST NoC
STBus Slave
18
WRITE
SIZE=2
LENGTH=5
add= 0x01
incrementing
FLITDATA =4
add= 0x01
ST2
add= 0x08
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

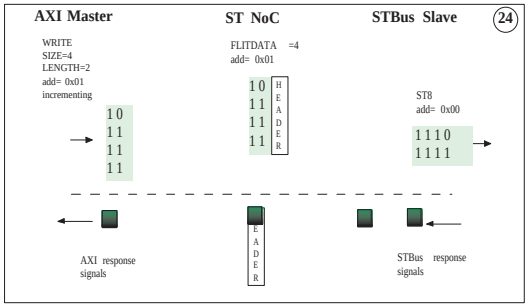

AXI Master
ST NoC
STBus Slave
19
WRITE
SIZE=4
LENGTH=1
add= 0x00
incrementing
FLITDATA =4
add= 0x01
ST4
add= 0x00
HEADER
AXI response signals
STBus response signals

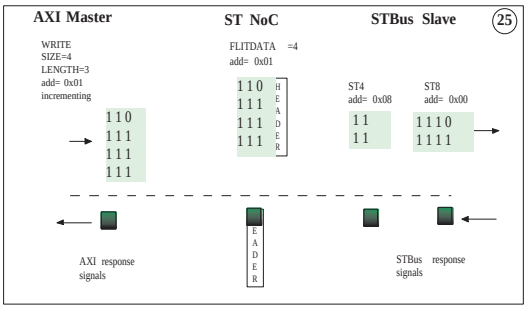

AXI Master
ST NoC
STBus Slave
20
WRITE
SIZE=4
LENGTH=2
add= 0x00
incrementing
FLITDATA =4
add= 0x01
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

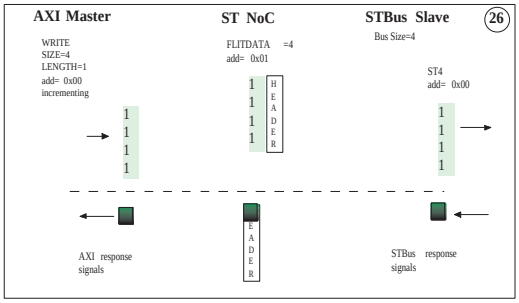

AXI Master
ST NoC
STBus Slave
21
WRITE
SIZE=4
LENGTH=3
add= 0x00
incrementing
FLITDATA =4
add= 0x01
ST4
add= 0x08
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

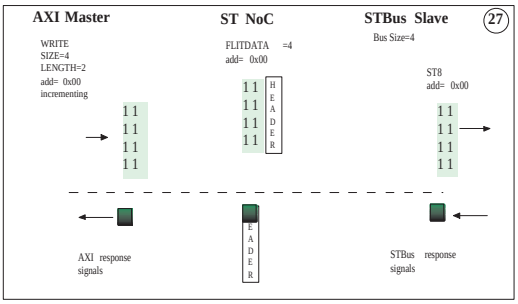

AXI Master
ST NoC
STBus Slave
22
WRITE
SIZE=4
LENGTH=4
add= 0x00
incrementing
FLITDATA =4
add= 0x01
ST16
add= 0x00
HEADER
AXI response signals
STBus response signals

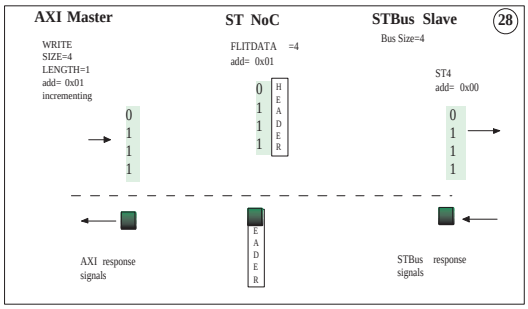

AXI Master
ST NoC
STBus Slave
23
WRITE
SIZE=4
LENGTH=1
add= 0x01
incrementing
FLITDATA =4
add= 0x01
ST4
add= 0x00
HEADER
AXI response signals
STBus response signals

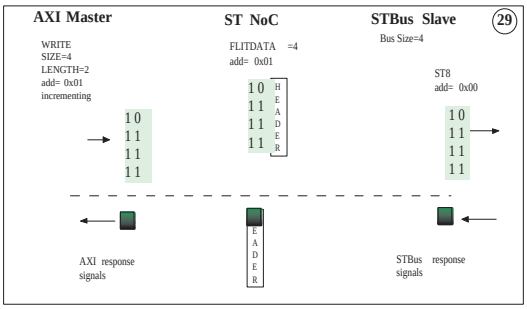

AXI Master
ST NoC
STBus Slave
24
WRITE
SIZE=4
LENGTH=2
add= 0x01
incrementing
FLITDATA =4
add= 0x01
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

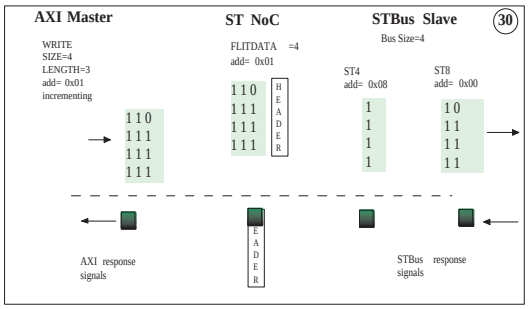

AXI Master
ST NoC
STBus Slave
25
WRITE
SIZE=4
LENGTH=3
add= 0x01
incrementing
FLITDATA =4
add= 0x01
HEADER
ST4
add= 0x08
ST8
add= 0x00
AXI response signals
STBus response signals

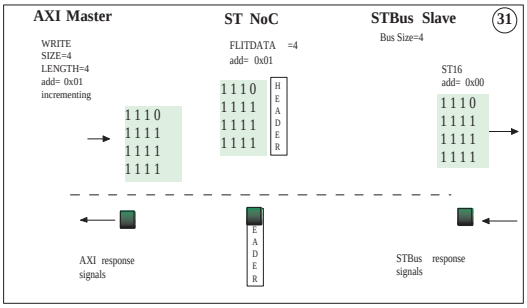

AXI Master
ST NoC
STBus Slave
26
WRITE
SIZE=4
LENGTH=1
add= 0x00
incrementing
FLITDATA =4
add= 0x01
Bus Size=4
HEADER
ST4
add= 0x00
AXI response signals
STBus response signals

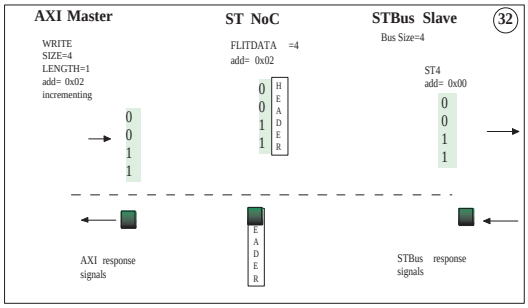

AXI Master
ST NoC
STBus Slave
27
WRITE
SIZE=4
LENGTH=2
add= 0x00
incrementing
FLITDATA =4
add= 0x00
Bus Size=4
HEADER
ST8
add= 0x00
AXI response signals
STBus response signals

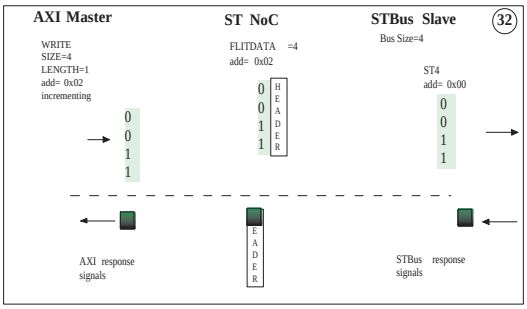

AXI Master
ST NoC
STBus Slave
28
WRITE
SIZE=4
LENGTH=1
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=4
HEADER
ST4
add= 0x00
AXI response signals
STBus response signals

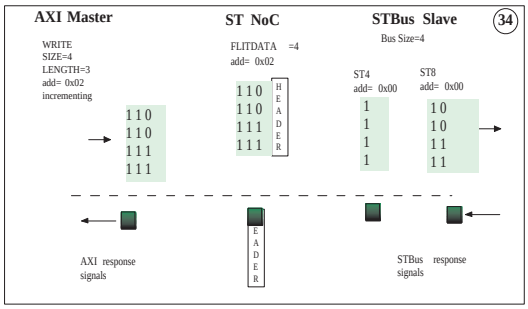

AXI Master
ST NoC
STBus Slave
29
WRITE
SIZE=4
LENGTH=2
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=4
HEADER
ST8
add= 0x00
AXI response signals
STBus response signals

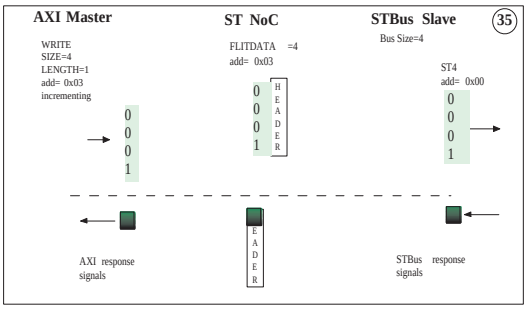

AXI Master
ST NoC
STBus Slave
30
WRITE
SIZE=4
LENGTH=3
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=4
HEADER
ST4
add= 0x08
ST8
add= 0x00
AXI response signals
STBus response signals

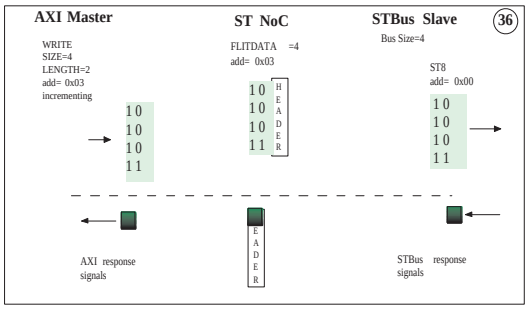

AXI Master
ST NoC
STBus Slave
31
WRITE
SIZE=4
LENGTH=4
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=4
HEADER
ST16
add= 0x00
AXI response signals
STBus response signals

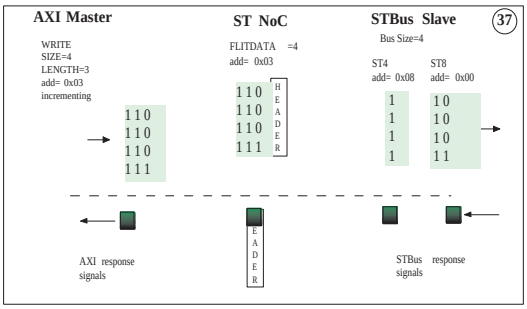

AXI Master
ST NoC
STBus Slave
32
WRITE
SIZE=4
LENGTH=1
add= 0x02
incrementing
FLITDATA =4
add= 0x02
Bus Size=4
HEADER
ST4
add= 0x00
AXI response signals
STBus response signals

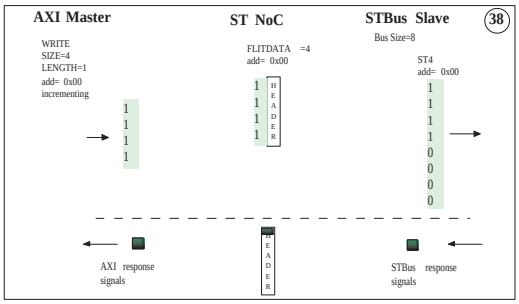

AXI Master
ST NoC
STBus Slave
32
WRITE
SIZE=4
LENGTH=1
add= 0x02
incrementing
FLITDATA =4
add= 0x02
HEADER
Bus Size=4
ST4
add= 0x00
AXI response signals
STBus response signals

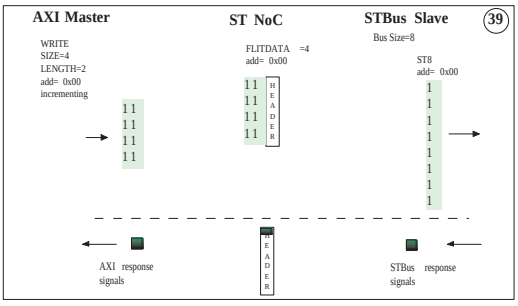

AXI Master
ST NoC
STBus Slave
34
WRITE
SIZE=4
LENGTH=3
add= 0x02
incrementing
FLITDATA =4
add= 0x02
HEADER
Bus Size=4
ST4
add= 0x00
ST8
add= 0x00
AXI response signals
STBus response signals

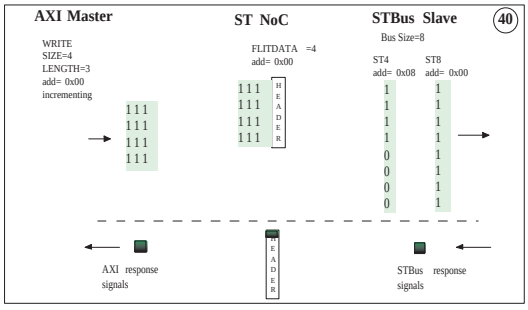

AXI Master
ST NoC
STBus Slave
35
WRITE
SIZE=4
LENGTH=1
add= 0x03
incrementing
FLITDATA =4
add= 0x03
HEADER
Bus Size=4
ST4
add= 0x00
AXI response signals
STBus response signals

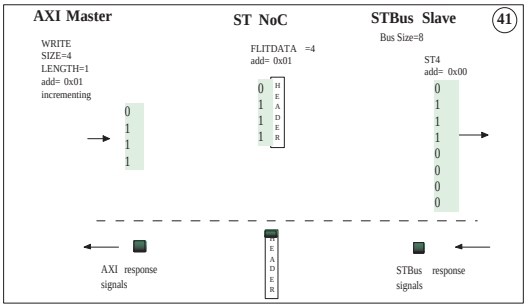

AXI Master
ST NoC
STBus Slave
36
WRITE
SIZE=4
LENGTH=2
add= 0x03
incrementing
FLITDATA =4
add= 0x03
HEADER
Bus Size=4
ST8
add= 0x00
AXI response signals
STBus response signals

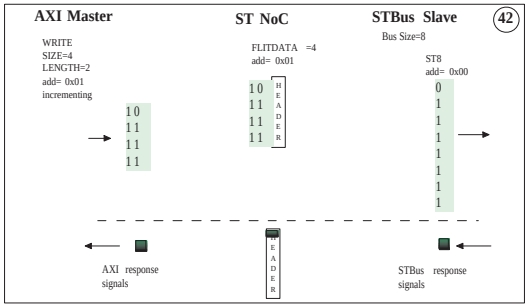

AXI Master
ST NoC
STBus Slave
37
WRITE
SIZE=4
LENGTH=3
add= 0x03
incrementing
FLITDATA =4
add= 0x03
HEADER
Bus Size=4
ST4
add= 0x08
ST8
add= 0x00
AXI response signals
STBus response signals

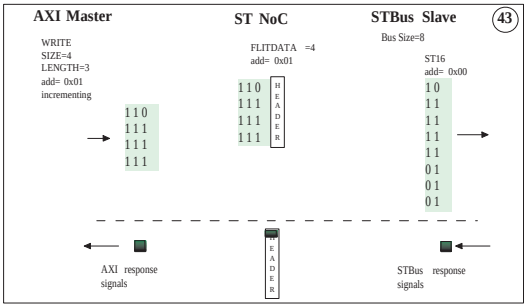

AXI Master
ST NoC
STBus Slave
38
WRITE
SIZE=4
LENGTH=1
add= 0x00
incrementing
FLITDATA =4
add= 0x00
HEADER
Bus Size=8
ST4
add= 0x00
AXI response signals
STBus response signals

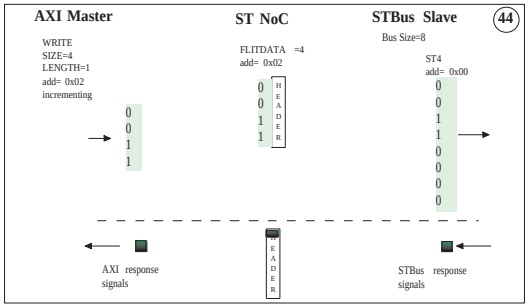

AXI Master
ST NoC
STBus Slave
39
WRITE
SIZE=4
LENGTH=2
add= 0x00
incrementing
FLITDATA =4
add= 0x00
HEADER
Bus Size=8
ST8
add= 0x00
AXI response signals
STBus response signals

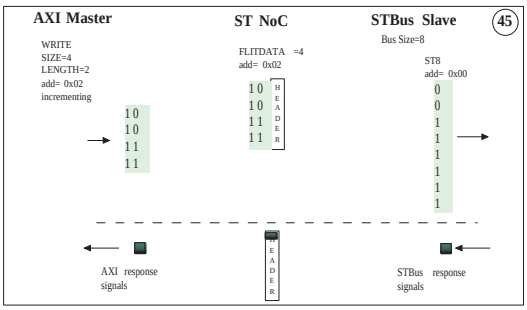

AXI Master
ST NoC
STBus Slave
40
WRITE
SIZE=4
LENGTH=3
add= 0x00
incrementing
FLITDATA =4
add= 0x00
HEADER
Bus Size=8
ST4
add= 0x08
ST8
add= 0x00
AXI response signals
STBus response signals

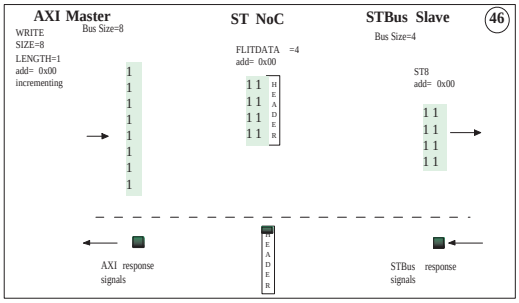

AXI Master
ST NoC
STBus Slave
41
WRITE
SIZE=4
LENGTH=1
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=8
ST4
add= 0x00
HEADER
AXI response signals
STBus response signals

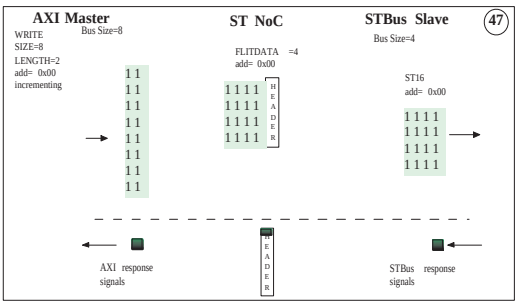

AXI Master
ST NoC
STBus Slave
42
WRITE
SIZE=4
LENGTH=2
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=8
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

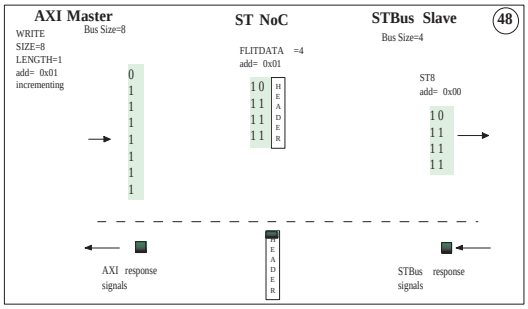

AXI Master
ST NoC
STBus Slave
43
WRITE
SIZE=4
LENGTH=3
add= 0x01
incrementing
FLITDATA =4
add= 0x01
Bus Size=8
ST16
add= 0x00
HEADER
AXI response signals
STBus response signals


AXI Master
ST NoC
STBus Slave
44
WRITE
SIZE=4
LENGTH=1
add= 0x02
incrementing
FLITDATA =4
add= 0x02
Bus Size=8
ST4
add= 0x00
HEADER
AXI response signals
STBus response signals

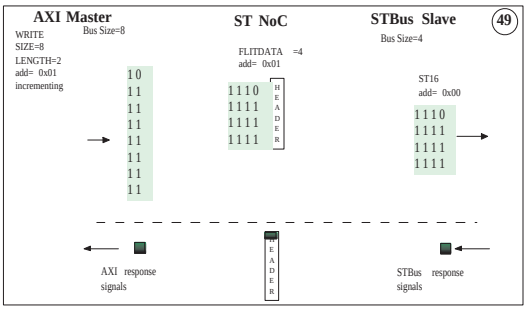

AXI Master
ST NoC
STBus Slave
45
WRITE
SIZE=4
LENGTH=2
add= 0x02
incrementing
FLITDATA =4
add= 0x02
Bus Size=8
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

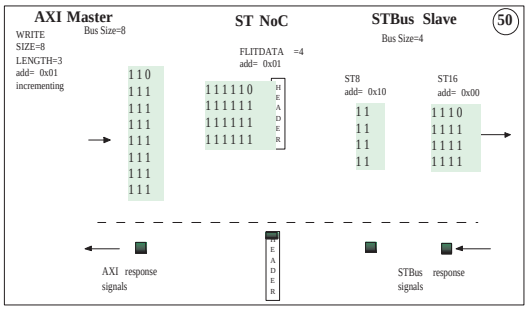

AXI Master
ST NoC
STBus Slave
46
WRITE
SIZE=8
LENGTH=1
add= 0x00
incrementing
Bus Size=8
FLITDATA =4
add= 0x00
Bus Size=4
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

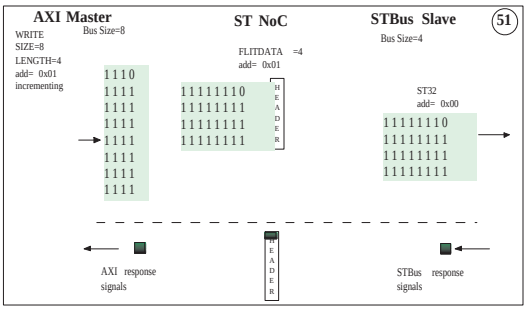

AXI Master
ST NoC
STBus Slave
47
WRITE
SIZE=8
LENGTH=2
add= 0x00
incrementing
Bus Size=8
FLITDATA =4
add= 0x00
Bus Size=4
ST16
add= 0x00
HEADER
AXI response signals
STBus response signals

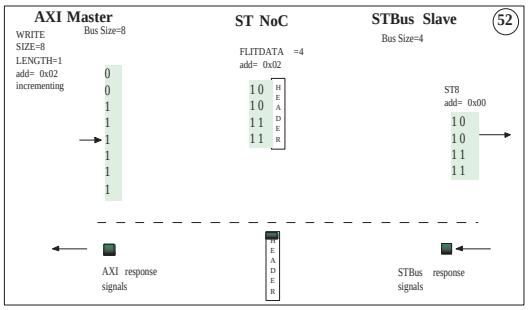

AXI Master
ST NoC
STBus Slave
48
WRITE
SIZE=8
LENGTH=1
add= 0x01
incrementing
Bus Size=8
FLITDATA =4
add= 0x01
Bus Size=4
ST8
add= 0x00
HEADER
AXI response signals
STBus response signals

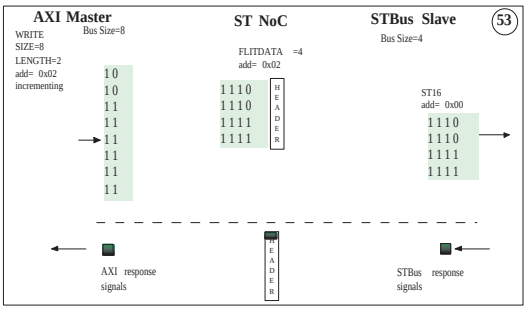
A. Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms.

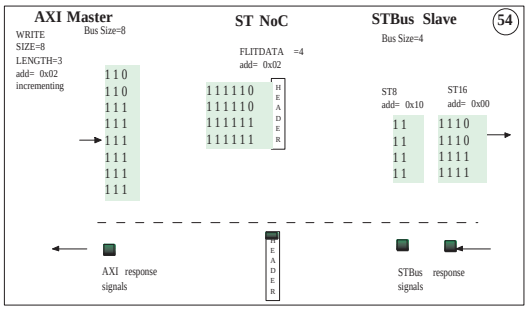


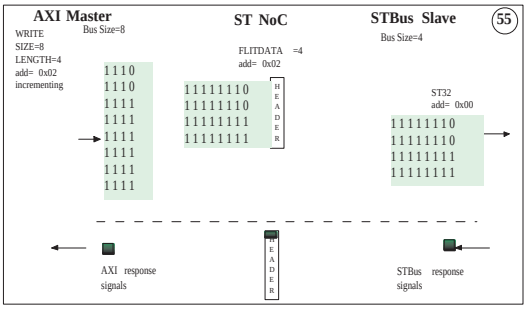


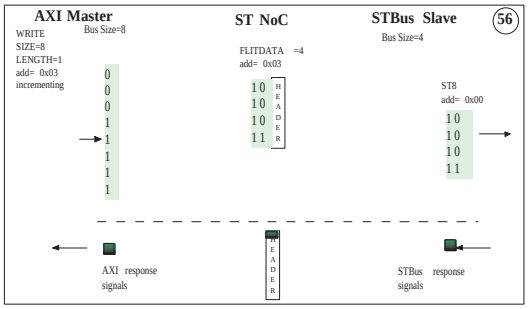


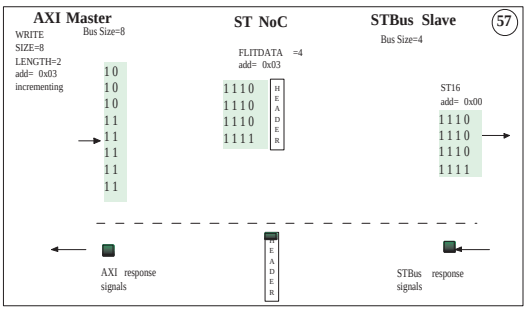


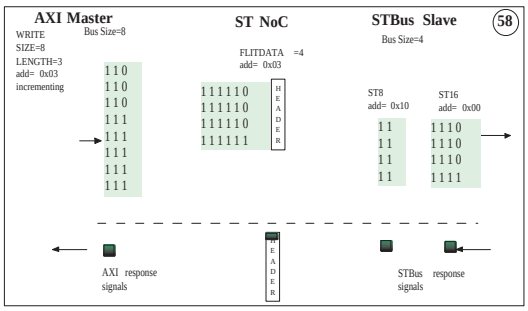


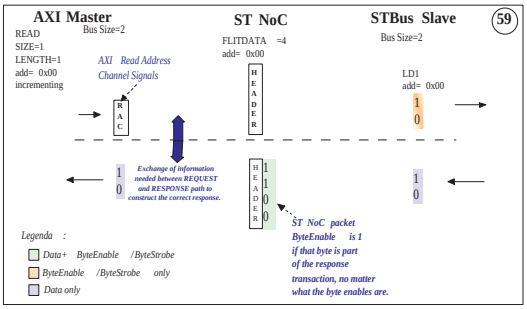


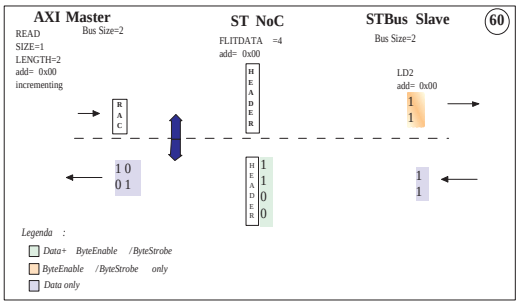


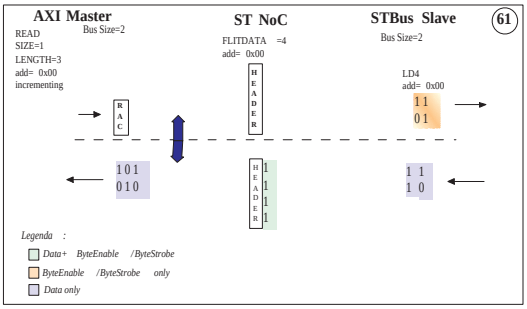


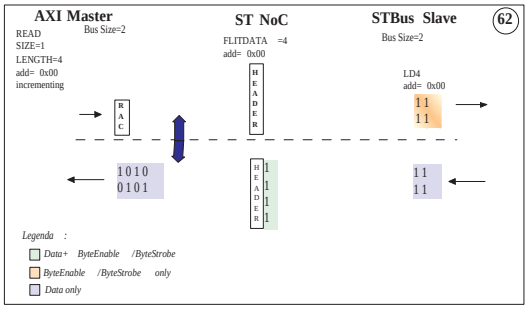

AXI Master
ST NoC
STBus Slave
57
WRITE
SIZE=8
LENGTH=2
add= 0x03
incrementing
Bus Size=8
FLITDATA =4
add= 0x03
Bus Size=4
ST16
add= 0x00
HEADER
AXI response signals
STBus response signals

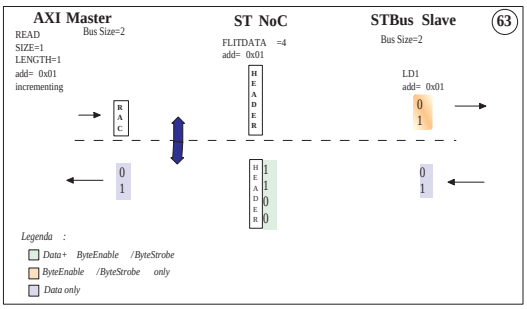

AXI Master
ST NoC
STBus Slave
58
WRITE
SIZE=8
LENGTH=3
add= 0x03
incrementing
Bus Size=8
FLITDATA =4
add= 0x03
Bus Size=4
ST8
add= 0x10
ST16
add= 0x00
HEADER
AXI response signals
STBus response signals

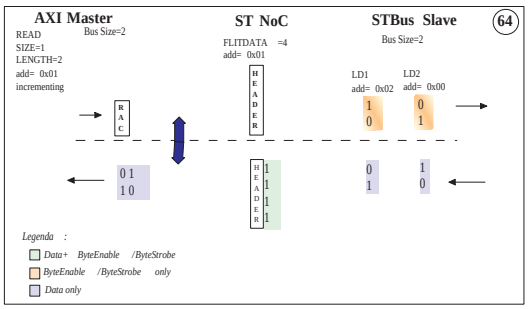

AXI Master
ST NoC
STBus Slave
59
READ
SIZE=1
LENGTH=1
add= 0x00
incrementing
Bus Size=2
AXI Read Address Channel Signals
FLITDATA =4
add= 0x00
Bus Size=2
LD1
add= 0x00
RAC
HEADER
Exchange of information needed between REQUEST and RESPONSE path to construct the correct response.
ST NoC packet ByteEnable is 1 if that byte is part of the response transaction, no matter what the byte enables are.
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

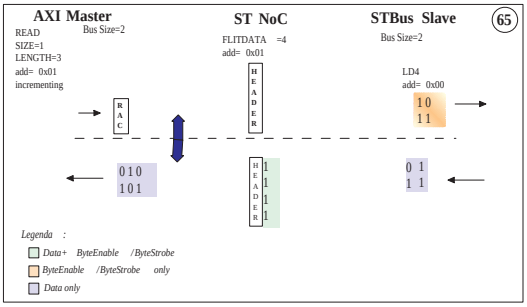

AXI Master
ST NoC
STBus Slave
60
READ
SIZE=1
LENGTH=2
add= 0x00
incrementing
Bus Size=2
FLITDATA =4
add= 0x00
Bus Size=2
LD2
add= 0x00
RAC
HEADER
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

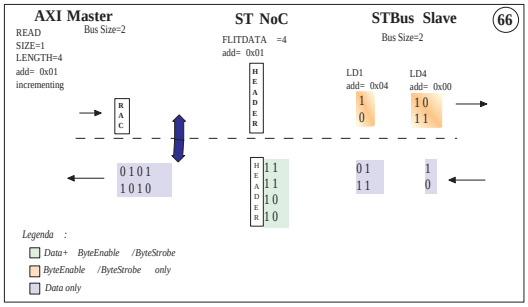

AXI Master
ST NoC
STBus Slave
61
READ
SIZE=1
LENGTH=3
add= 0x00
incrementing
Bus Size=2
FLITDATA =4
add= 0x00
Bus Size=2
LD4
add= 0x00
RAC
HEADER
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

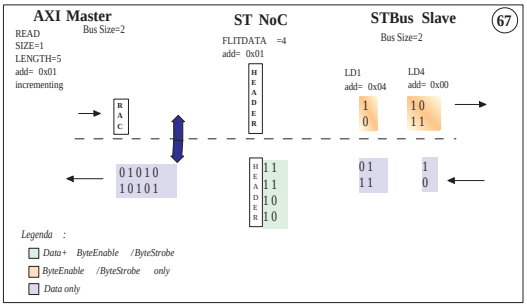

AXI Master
ST NoC
STBus Slave
62
READ
SIZE=1
LENGTH=4
add= 0x00
incrementing
Bus Size=2
FLITDATA =4
add= 0x00
Bus Size=2
LD4
add= 0x00
RAC
HEADER
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

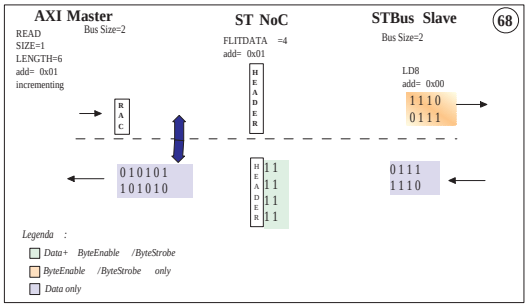

AXI Master
ST NoC
STBus Slave
63
READ
SIZE=1
LENGTH=1
add= 0x01
incrementing
Bus Size=2
FLITDATA =4
add= 0x01
Bus Size=2
LD1
add= 0x01
RAC
HEADER
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

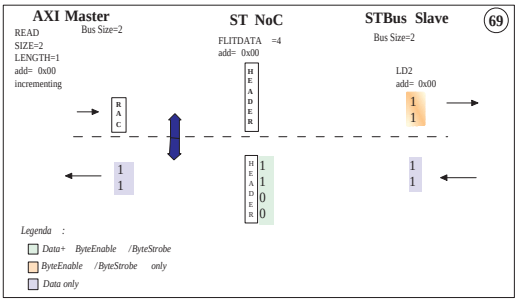

AXI Master
ST NoC
STBus Slave
64
READ
SIZE=1
LENGTH=2
add= 0x01
incrementing
Bus Size=2
FLITDATA =4
add= 0x01
Bus Size=2
LD1
add= 0x02
LD2
add= 0x00
RAC
HEADER
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

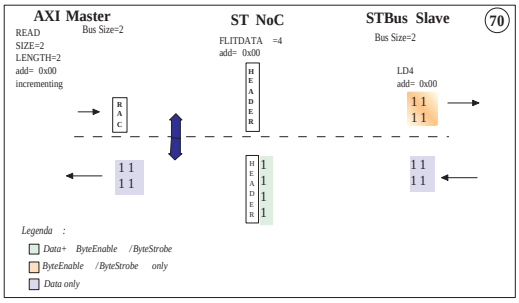

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=3
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD4
add= 0x00
65
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

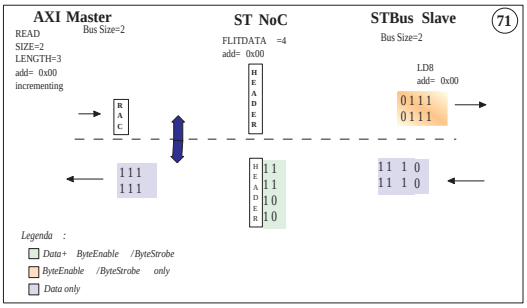

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=4
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD1
add= 0x04
LD4
add= 0x00
66
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

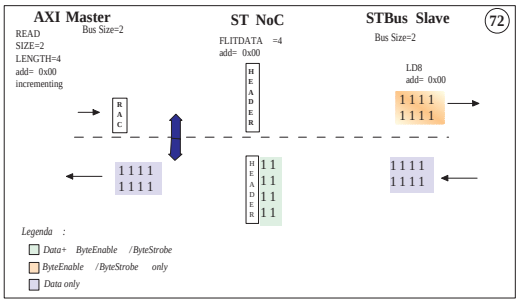

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=5
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD1
add= 0x04
LD4
add= 0x00
67
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

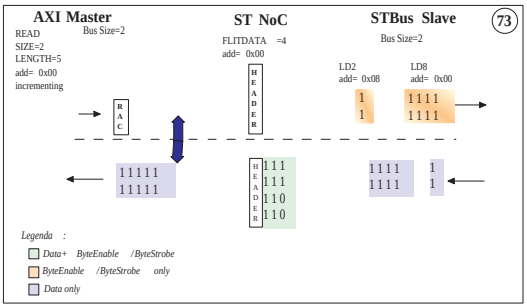

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=6
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD8
add= 0x00
68
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

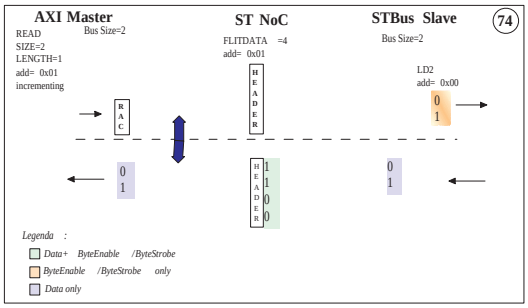

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=1
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=2
LD2
add= 0x00
69
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

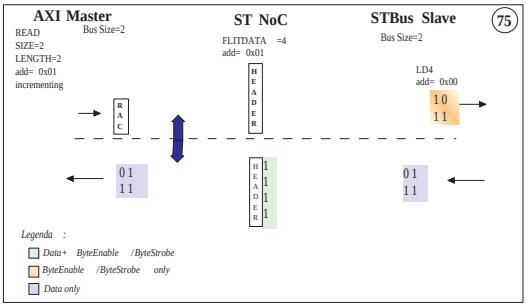

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=2
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=2
LD4
add= 0x00
70
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

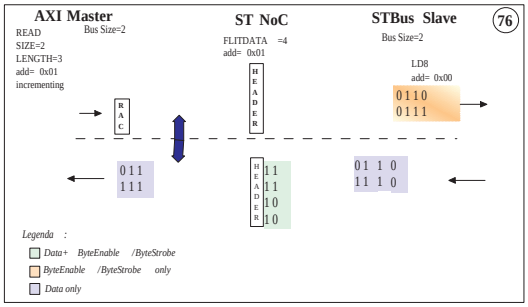

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=3
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=2
LD8
add= 0x00
71
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

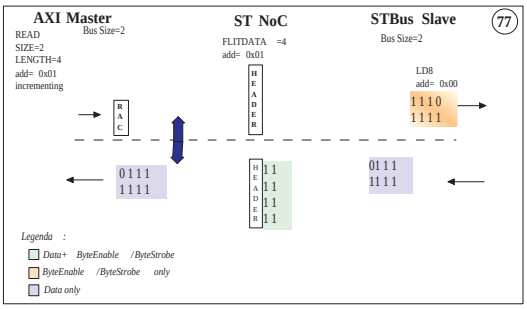

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=4
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=2
LD8
add= 0x00
72
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

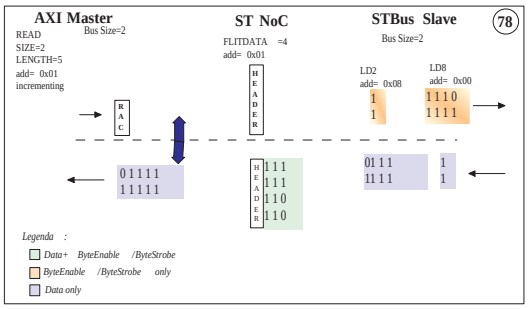

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=5
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=2
LD2
add= 0x08
LD8
add= 0x00
73
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

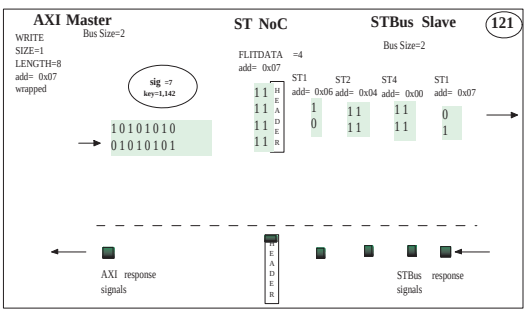

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=1
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD2
add= 0x00
74
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

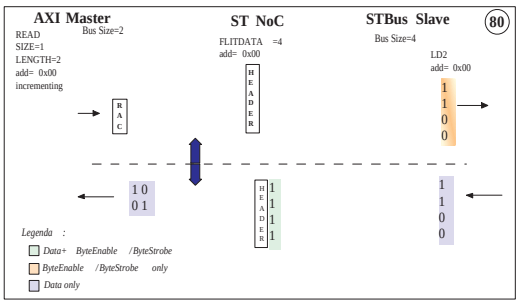

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=2
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD4
add= 0x00
75
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only


AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=3
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD8
add= 0x00
76
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

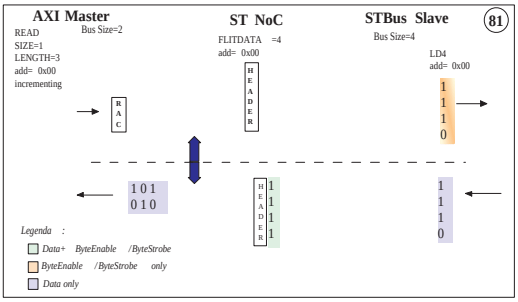

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=4
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD8
add= 0x00
77
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

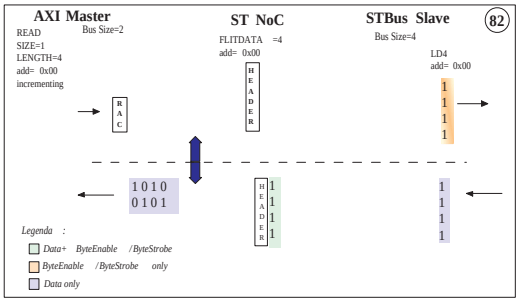

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=5
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=2
LD2
add= 0x08
LD8
add= 0x00
78
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

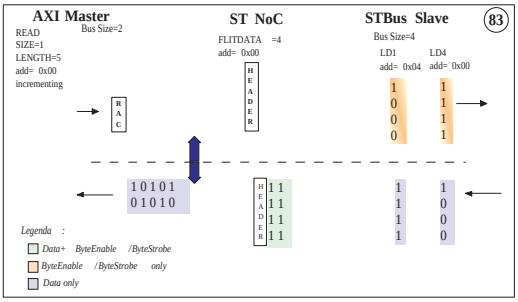

AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x07
wrapped
sig :=7
key=1,142
ST NoC
FLITDATA =4
add= 0x07
STBus Slave
Bus Size=2
ST1
add= 0x06
ST2
add= 0x04
ST4
add= 0x00
ST1
add= 0x07
121
AXI response signals
STBus response signals

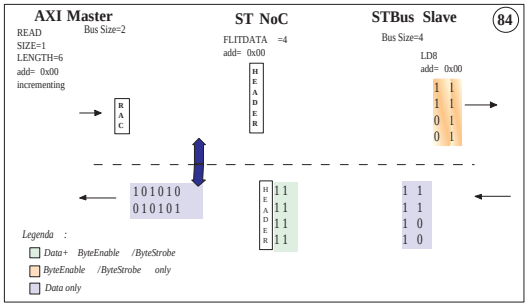

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=2
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=4
LD2
add= 0x00
80
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

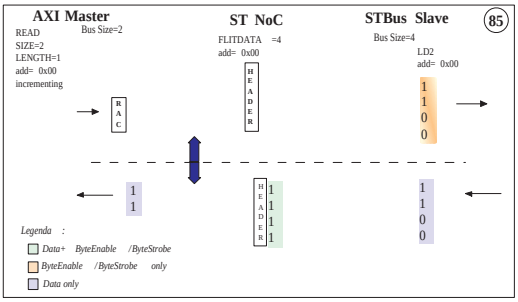
A. Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms.

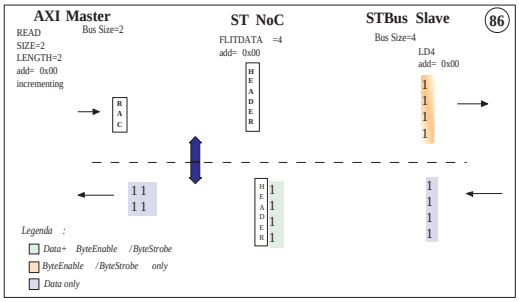


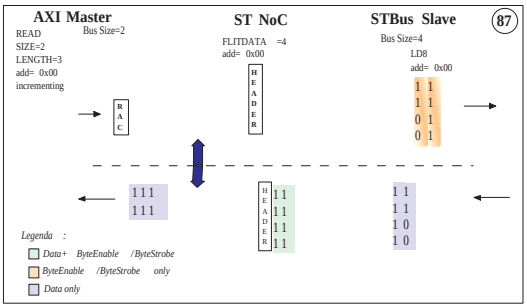


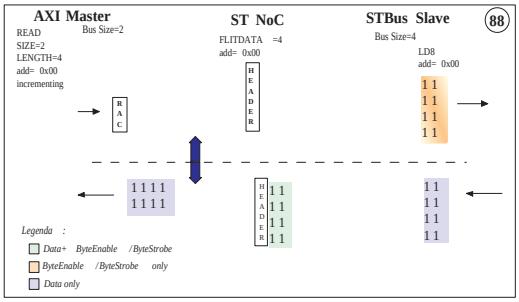


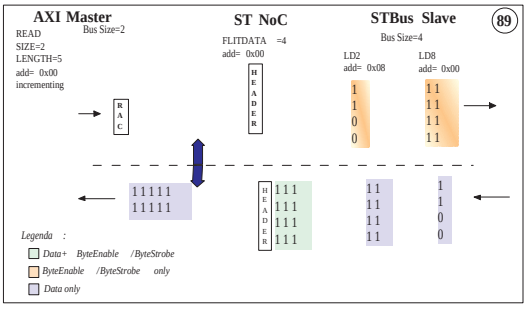


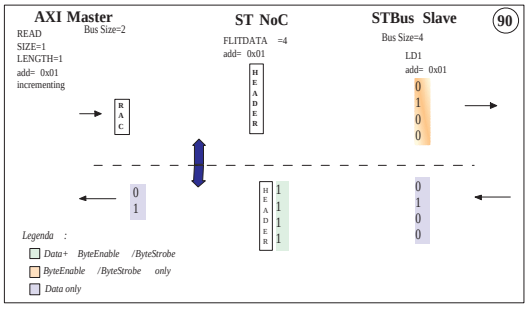


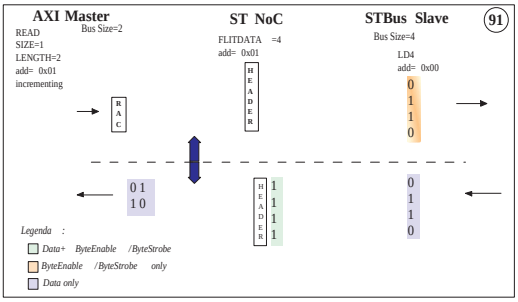


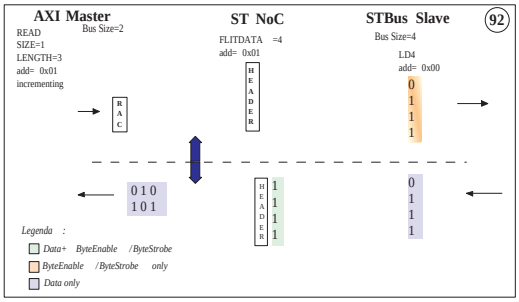


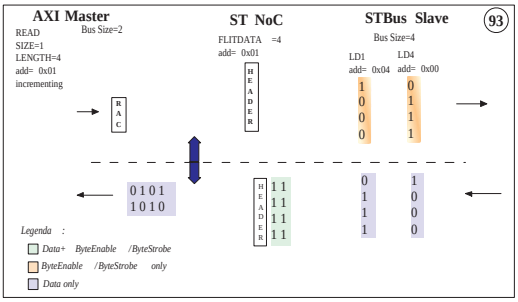


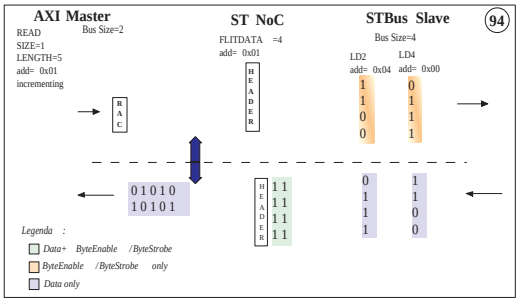

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=5
add= 0x00
incrementing
ST NoC
FLITDATA =4
add= 0x00
STBus Slave
Bus Size=4
LD2
add= 0x08
LD8
add= 0x00
89
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

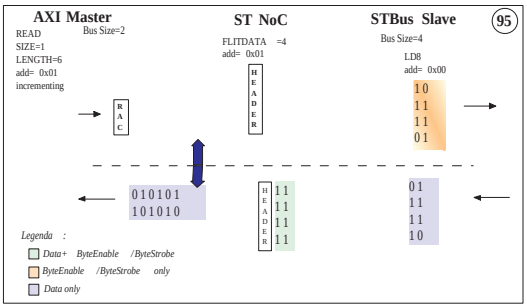

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=1
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD1
add= 0x01
90
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

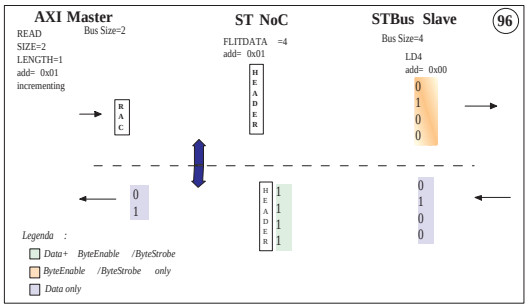

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=2
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD4
add= 0x00
91
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

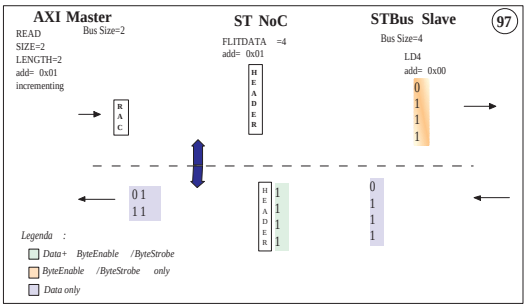

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=3
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD4
add= 0x00
92
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

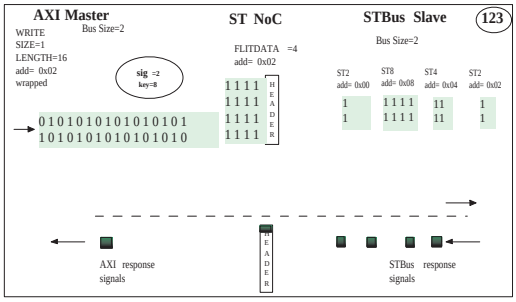

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=4
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD1
add= 0x04
LD4
add= 0x00
93
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

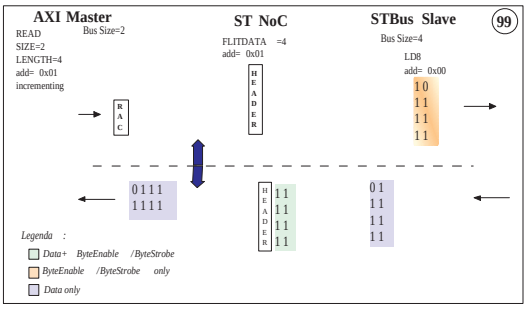

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=5
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD2
add= 0x04
LD4
add= 0x00
94
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

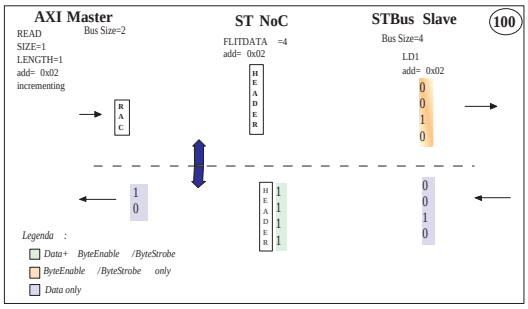

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=6
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD8
add= 0x00
95
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

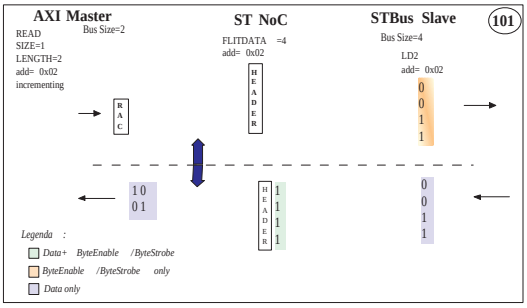

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=1
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD4
add= 0x00
96
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

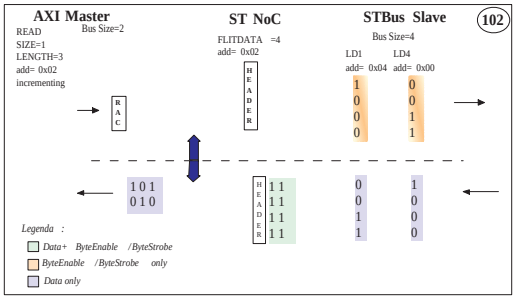

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=2
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD4
add= 0x00
97
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

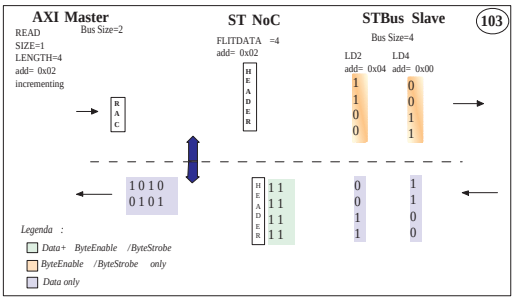

AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=16
add= 0x02
wrapped
sig =2
key=8
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=2
ST2
add= 0x00
ST8
add= 0x08
ST4
add= 0x04
ST2
add= 0x02
123
AXI response signals
STBus response signals

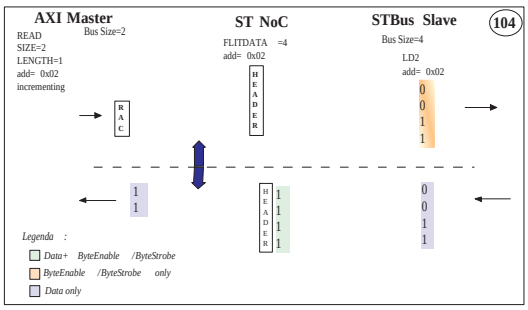

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=4
add= 0x01
incrementing
ST NoC
FLITDATA =4
add= 0x01
STBus Slave
Bus Size=4
LD8
add= 0x00
99
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only


AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=1
add= 0x02
incrementing
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=4
LD1
add= 0x02
100
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

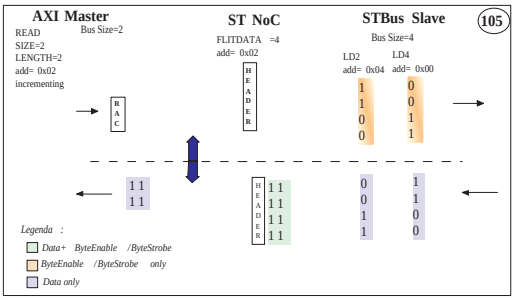

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=2
add= 0x02
incrementing
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=4
LD2
add= 0x02
101
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

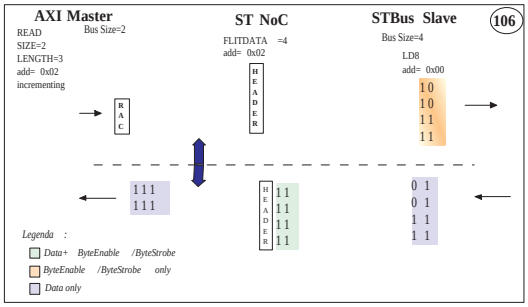

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=3
add= 0x02
incrementing
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=4
LD1
add= 0x04
LD4
add= 0x00
102
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

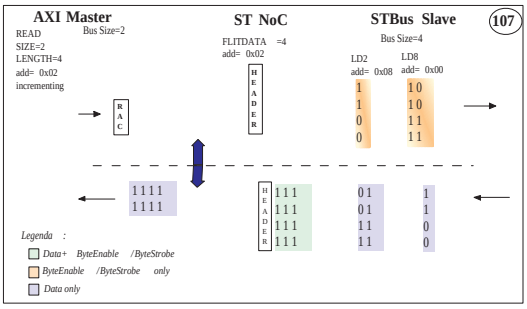

AXI Master
Bus Size=2
READ
SIZE=1
LENGTH=4
add= 0x02
incrementing
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=4
LD2
add= 0x04
LD4
add= 0x00
103
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

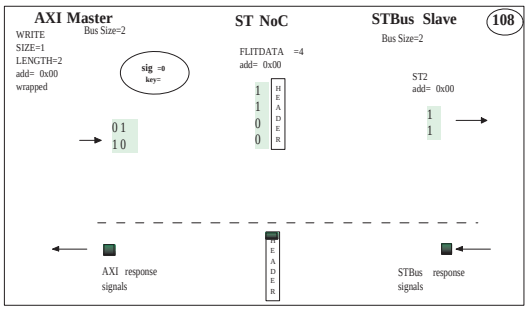

AXI Master
Bus Size=2
READ
SIZE=2
LENGTH=1
add= 0x02
incrementing
ST NoC
FLITDATA =4
add= 0x02
STBus Slave
Bus Size=4
LD2
add= 0x02
104
Legenda :
Data+ ByteEnable /ByteStrobe
ByteEnable /ByteStrobe only
Data only

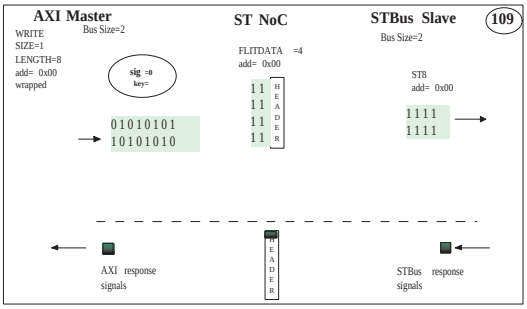
A. Selection of traffic scenarios generated by implementation of Multi-socket STNoC™ algorithms.

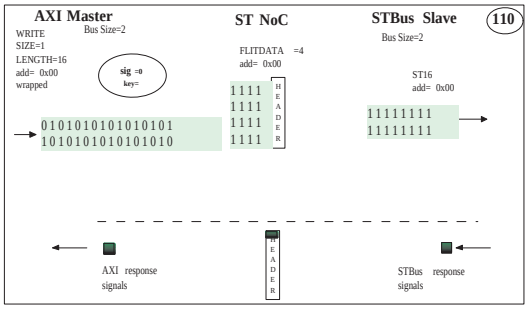


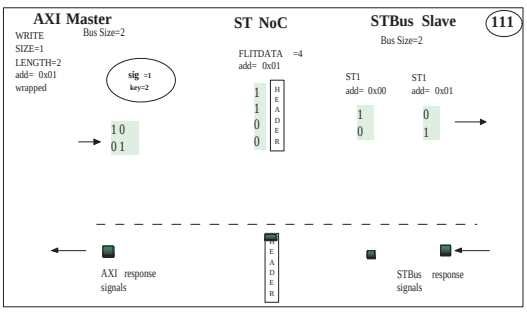


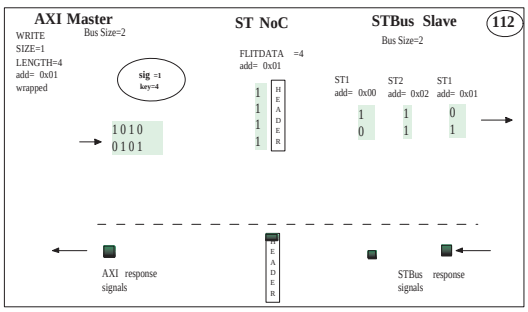


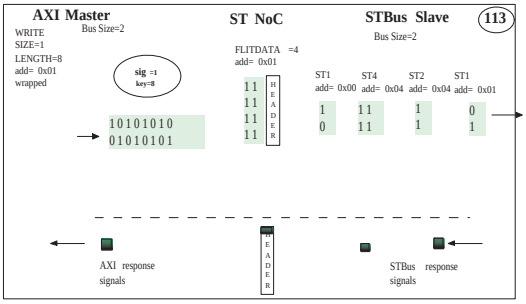


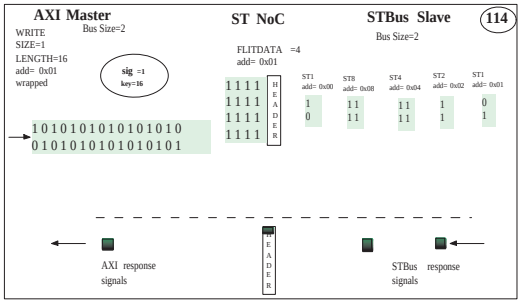


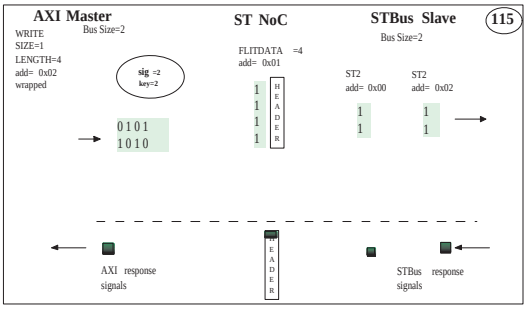


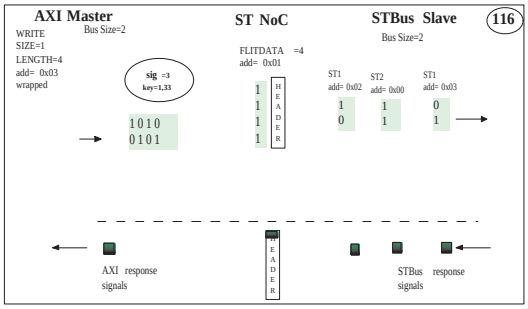


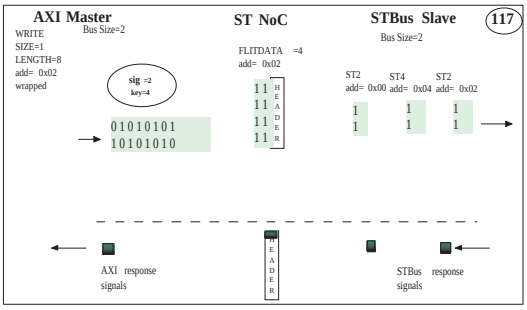


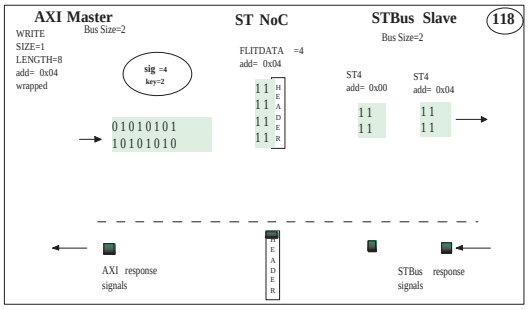

AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x01
wrapped
sig =1
key=8
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
ST NoC
Bus Size=2
FLITDATA =4
add= 0x01
HEADER
ST1 add= 0x00
ST4 add= 0x04
ST2 add= 0x04
ST1 add= 0x01
STBus Slave
Bus Size=2
113
AXI response signals
STBus response signals

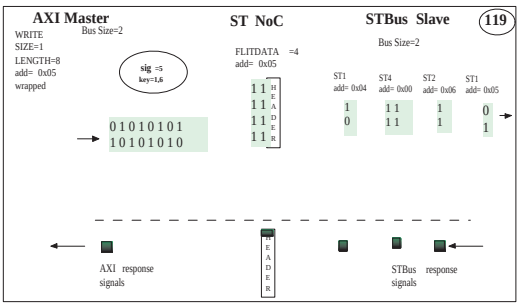

AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=16
add= 0x01
wrapped
sig =1
key=16
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
ST NoC
Bus Size=2
FLITDATA =4
add= 0x01
HEADER
ST1 add= 0x00
ST8 add= 0x08
ST4 add= 0x04
ST2 add= 0x02
ST1 add= 0x01
STBus Slave
Bus Size=2
114
AXI response signals
STBus response signals

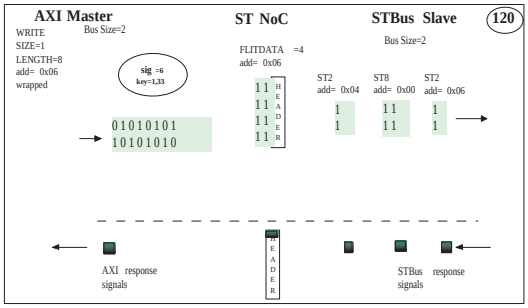

AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=4
add= 0x02
wrapped
sig =2
key=2
0 1 0 1
1 0 1 0
ST NoC
Bus Size=2
FLITDATA =4
add= 0x01
HEADER
ST2 add= 0x00
ST2 add= 0x02
STBus Slave
Bus Size=2
115
AXI response signals
STBus response signals


AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=4
add= 0x03
wrapped
sig =3
key=1,33
1 0 1 0
0 1 0 1
ST NoC
Bus Size=2
FLITDATA =4
add= 0x01
HEADER
ST1 add= 0x02
ST2 add= 0x00
ST1 add= 0x03
STBus Slave
Bus Size=2
116
AXI response signals
STBus response signals


AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x02
wrapped
sig =2
key=4
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
ST NoC
Bus Size=2
FLITDATA =4
add= 0x02
HEADER
ST2 add= 0x00
ST4 add= 0x04
ST2 add= 0x02
STBus Slave
Bus Size=2
117
AXI response signals
STBus response signals


AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x04
wrapped
sig =4
key=2
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
ST NoC
Bus Size=2
FLITDATA =4
add= 0x04
HEADER
ST4 add= 0x00
ST4 add= 0x04
STBus Slave
Bus Size=2
118
AXI response signals
STBus response signals


AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x05
wrapped
sig =5
key=1,6
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
ST NoC
Bus Size=2
FLITDATA =4
add= 0x05
HEADER
ST1 add= 0x04
ST4 add= 0x00
ST2 add= 0x06
ST1 add= 0x05
STBus Slave
Bus Size=2
119
AXI response signals
STBus response signals


AXI Master
Bus Size=2
WRITE
SIZE=1
LENGTH=8
add= 0x06
wrapped
sig =6
key=1,33
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
ST NoC
Bus Size=2
FLITDATA =4
add= 0x06
HEADER
ST2 add= 0x04
ST8 add= 0x00
ST2 add= 0x06
STBus Slave
Bus Size=2
120
AXI response signals
STBus response signals

Bibliography

[1] Amba™ axi protocol v1.0 specification. *http://www.arm.com*.

[2] Amba™ specification (rev.2.0). *http://www.arm.com*.

[3] Arm™ architecture reference manual. *http://www.arm.com*.

[4] Open core protocol specification, release 2.1. *http://www.ocpip.org*.

[5] Stbus communication system: Concepts and definitions. *STMicroelectronics internal document*.

[6] The coreconnect™ bus architecture. *http://www.chips.ibm.com/products/coreconnect*.

[7] ITRS. 2003. International technology roadmap for semiconductors. tech. rep., international technology roadmap for semiconductors.

[8] J. BAINBRIDGE and S. FURBER. Chain: A delay-insensitive chip area interconnect. *IEEE Micro 22*, (5 ,Oct):16–23, 2002.

[9] W. BAINBRIDGE and S. FURBER. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 118–126, 2001.

[10] L. BENINI and G. D. MICHELI. Powering network-on-chips. *The 14th International Symposium on System Synthesis (ISSS)*, pages 33–38, 2001.

[11] MAHADEVAN S. OLSEN R. G. BJERREGAARD, T. and J. SPARSØ. An ocp compliant network adapter for gals-based soc design using the mango network-on-chip. *Proceedings of International Symposium on System-on-Chip (ISSoC)*, 2005.

[12] LOCATELLI R. MARUCCIA G. PIERALISI L. SCANDURRA A. COPPOLA, M. Spidergon: a novel on-chip communication network. *Proceedings of International Symposium on System-on-Chip*, 2004.

[13] MANHO K. DAEWOOK, K. and G.E. SOBELMAN. Niugap: low latency network interface architecture with gray code for networks-on-chip. *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, 2006.

[14] W. J. DALLY. Virtual-channel flow control. *IEEE Trans. Parall. Distrib. Syst. 3*, 2,March:194–205, 1992.

[15] W. J. DALLY and C. L. SEITZ. Deadlock–free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput. 36*, (5,May):547–553, 1987.

[16] J. DUATO. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parall. Distrib. Syst. 4*, (12,Dec):1320–1331, 1993.

[17] P. MARTIN. Design of a virtual component neutral network-on-chip transaction layer. *Proceedings of the Design, Automation and Test in Europe Conference and Exibition (DATE)*, 2005.

[18] G. MARUCCIA. Stnoc network interface - functional specification. *STMicroelectronics internal document*, (ADCS 7785770), 2005.

[19] NILSSON E. THID R. MILLBERG, M. and A. JANTSCH. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network-on-chip. *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, pages 890–895, 2004.

[20] L.-S. PEH and W. J. DALLY. A delay model for router microarchitectures. *IEEE Micro 21*, pages 26–34, 2001.

[21] DIELISSEN J. GOOSSENS K. RIJPKEMA E. ANDWIELAGE P. RADULESCU, A. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, pages 878–883, 2004.

[22] ANGIOLINI F. CARTA S. RAFFO L. BERTOZZI D. STERGIOU, S. and G.D. DE MICHELI. Xpipes lite: A synthesis oriented design library for networks on chips. *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, 2005.