Development and Verification of System-On-a-Chip Communication Architecture

A Dissertation Presented

by

Jian Liang

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

March 15, 2004

Department of Electrical and Computer Engineering

Development and Verification of System-On-a-Chip Communication Architecture

A Dissertation Presented

by

Jian Liang

Approved as to style and content by:

To the betterment of mankind

Acknowledgments

My sincere thanks go to my adviser Professor Russell Tessier. His support and guidance played a major role in this research. It has been a wonderful experience to be a part of Reconfigurable Computing Group for the past five years. I appreciate the help of Professor Dennis L. Goeckel. Weekly meetings with him helped me develop and improve the Turbo decoding algorithm. The committee members, Professor Wayne P. Burleson and Professor Charles C. Weems, have assisted me greatly with their valuable feedbacks on this dissertation work.

It has been a great pleasure for me to work with the researchers on the 3th floor of Knowles Engineering Building. Special thanks go to Andy Laffely and Srini Krishnamoorthy for their help on my research papers. I am also thankful to my colleagues in the Reconfigurable Computing Group for their friendship and supports.

Finally, I would thank my wife for her support. While I was working on my dissertation, she has manged our lives. Her selfless devotion to our family is inspirational.

Abstract

Development and Verification of System-On-a-Chip Communication Architecture

March 15, 2004

Jian Liang

B.E, Tsinghua University, Beijing, China

M.S., Tsinghua University, Beijing, China

Ph.D, University of Massachusetts Amherst

Directed by: Professor Russell Tessier

Recent advances in VLSI technology have led to a dramatic increase in the computation capacities of a single chip. Current industry estimates [55] indicate mass production of silicon devices containing over 1.4 billion transistors by 2013. This proliferation of on-chip resources enables the integration of complex system-on-a-chip (SoC) designs containing a wide range of intellectual property (IP) cores. To provide high performance, SoC integration tools must consider the design of individual IP cores, their on-chip interconnections, and application mapping approaches. In this dissertation, the latter two design issues are addressed through the introduction of a new on-chip communications architecture, adaptive System-on-Chip (aSoC), and its supporting compilation software. Our communications architecture is scalable to tens of heterogeneous cores and can be customized on a per-application basis.

In aSoC, each computation core is associated with a communication interface, which is connected to each other as a two-dimensional mesh. Communication between cores takes place via pipelined, point-to-point connections. The communications are restricted to short wires between cores to allow high speed data transfer and predictable delays. The interface design can be customized based on core data bandwidths and operating frequencies to allow for efficient use of resources. The power and clock are supplied individually in each tile for diverse working environments of heterogeneous cores.

A supporting software tool, AppMapper, was developed to map applications onto aSoC. Built upon existing compiler infrastructure and taking advantage of user interaction, AppMapper analyzes the high level language applications, schedules the on-chip communications, and configures the communication interfaces in aSoC. To augment the support for heterogeneous cores from third parties, AppMapper outputs the core codes in high level languages format such as C or Verilog, which allows the third-party compilers to handle the core specific optimization.

To evaluate the performance of aSoC, applications are mapped onto the aSoC model chips. The chosen applications include MPEG-II encoder/decoder, image smoothing filter, IIR filter, Doppler radar decoder, and orthogonal frequency division multiplexing (OFDM) modulator. In order to further test aSoC for substantial and complex applications, a Turbo decoding system is mapped onto aSoC devices. A novel adaptive Soft-output Viterbi algorithm (ASOVA) is developed for the Turbo decoder to reduce the computation complexity. By varying the number of survivor paths, ASOVA is capable of adapting its decoding complexity to speed up the decoding in the case of high signal to noise ratio (SNR).

These applications are mapped onto aSoC model devices and their performance is tested. The results are compared to other on-chip communication architectures with the same partitioning. The evaluated architectures include the hierarchical bus modeled from IBM CoreConnect on-chip communication and a dynamic routing model applying oblivious dynamic routing [30]. Based on the simulation results, aSoC outperforms the hierarchical bus model by a factor of five.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increase of gate count per chip makes the IP core based System-on-a-Chip (SoC) a feasible solution for chip design. Currently used bus-based on-chip communication architecture has become the limiting factor of SoCs for its un-scalable nature. A novel on-chip communication platform, the Adaptive System-on-a-Chip (aSoC), is developed to fulfill this requirement. ASoC applies a core-based mesh architecture using a pipelined, packet-switching communication network for heterogeneous cores. A compiler tool is developed to statically schedule the on-chip communications in aSoC.

1.1 Motivation

The dramatic increase in the transistor capacity of a single chip has led to the design concepts of System-on-a-Chip (SoC). Current industry estimates [55] indicate that mass production of silicon devices containing over 1.4 billion transistors would be possible by 2013. To reduce the exponential increase in the design complexity of such a huge hardware resource, SoC makes use of a core-based structure, in which the cores can be embedded processors, large soft elements and algorithmic subsystems. These block elements are the reusable Intellectual Property (IP) cores obtained from company libraries, prior internal designs or licensed from third parties. With well-designed IP cores, SoC designers can focus on the system level optimizations.

In such a core-based SoC architecture with numerous of IP cores, the on-chip communication is the focus of the whole system. One of the critical points to be considered by the core designers is the communication bandwidth with other cores or memory. With a highly parallelized and pipelined architecture, current processor cores are able to issue multiple instructions per cycle. Most instructions require two or three operands. The communication bandwidth requirements of such cores is generally compatible with the core kernel speed. When modern VLSI techniques allow SoC to integrate increasing number of IP cores, on-chip communication will become the bottleneck for SoC. Other recent studies [63] indicate that on-chip communication has already become the limiting factor in SoC performance.

Most current SoC designs apply a bus-based communication architecture [39, 78, 52, 4, 3]. In the near future, when the number of on-chip cores scales, these architectures will soon fail to provide the demanded communication ability on account of the following factors: dynamic allocation of communication resources, limited data bandwidth and the clock skew and capacitance along the lone bus lines.

- Significant amounts of dynamic *arbitration* for shared communication resources across even a small number of components can quickly form a performance bottleneck, especially for data-intensive, stream-based computation. The centralization of the arbitrary controller also restricts the number and physical location of the cores.

- At any given time, a bus can be occupied by a single bus master. Other bus masters have to wait until the end of the current transaction before they can compete for the bus control, which means low communication efficiency and, therefore, limited *bandwidth.*

- To operate reliably, the *clock* skew should be under 10%, i.e. less than 5ns in current techniques [105]. When the chip is large and the bus line is long, the clock speed for the bus has to be slowed down to make sure that the skew is within the reliable range. Even though repeaters can be used to reduce the signal skew, it is still a serious problem when the bus lines have to travel across several cores.

- An additional, the *capacitance* of the interfaces, will also limit the number of cores that a bus can drive.

The design process of SoC is made more complex by the need of heterogeneous cores of differing applications. The heterogeneous nature of cores in terms of clock speed, resources, and processing capability makes cost modeling difficult. Additionally, communication modeling for interconnection with long wires and variable arbitration protocols limits performance predictability required by computation and communication scheduling. To fulfill different cores' requirements, modern on-chip buses support more than one communication scheme, such as DMA, burst transfers, split transactions, priority arbitration, and so on [54, 39]. Besides the data bus, each core has to provide a set of control signals to the central bus controller, which significantly degrades the communication performance.

Realizing the problems of bus-based architecture, numerous approaches has been proposed lately for on-chip communication with differing topologies, such as 1D array [34], 2D meshes [74, 103, 24], hierarchical network [66], hexagon mesh [70], and so on. These architectures can be categorized by their scheduling schemes into two classes: the static network which is scheduled at compile-time, and the dynamic network-on-chip (NoC) architectures. The former [74, 70, 24, 104], which is named the *FPGA-like* architecture, relies on the compilers to schedule the communication which simplifies the hardware resources, but loses the flexibility to change the communication pattern in run time, which is required by many applications with data dependent communications. As a result, the SoCs with FPGA-like architectures usually target specific function or applications with fixed communication patterns like multi-media and DSP. The latter [12, 50, 31] presents the idea of using dynamic

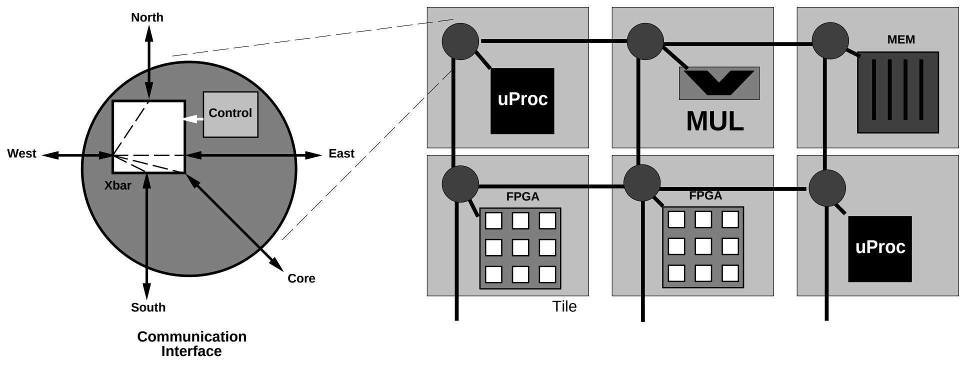


Figure 1.1. Adaptive System-on-a-Chip (aSoC)

networks for on-chip communication. These approaches provide full flexibility for communication, but the arbitration and scheduling totally rely on the network which results in high hardware costs. To better solve the dynamic routing problems, such as deadlock, congestion and buffering, more complicated circuitry will be required.

To better address the problems of prohibitive cross-chip communication on account of increasing die sizes, future SoCs will require a communication platform that can support a variety of diverse IP cores. This platform must also be scalable in terms of hardware resource cost, physical characteristics, application mapping, and software support.

1.2 Scalable aSoC Architecture

Our modular communications architecture, **a**daptive **s**ystem-**o**n-a-**c**hip (aSoC), provides a platform for next generation on-chip systems. The goal of aSoC is to build a scalable on-chip communication platform to solve the problems faced by the current bus-based architectures. By employing a pipelined packet-switching network, aSoC is able to achieve scalable data bandwidth. Furthermore, the static scheduling scheme makes aSoC a low-cost architecture for on-chip designs.

As shown in Figure 1.1, an aSoC device contains a two-dimensional mesh of computational *tiles*. Each tile consists of a core and an associated communication interface. The interface design can be customized based on core data bandwidths and operating frequencies to allow for efficient use of resources. The power and clock are supplied individually to each tile for diverse working environments of heterogeneous cores.

The neighboring tiles are connected using a pipelined, packet-switching network. To address the previously mentioned limiting factors of bus-based communication architecture, aSoC is designed with the following techniques:

- *Arbitration:* Static scheduling is employed in aSoC to eliminate the dynamic arbitration of bus-based architecture. While statically-scheduled data transfer is optimized in compile time, the software-based mechanism for data routing has been developed to reduce the congestion overhead. While most of the communications can be obtained by statically analyzing the codes, some data dependent communications can be handled by dynamic communication pattern switching of aSoC.

- *Bandwidth:* While the bandwidth of bus architecture is limited by the exclusive occupation of the bus master, the communication between aSoC nodes takes place via pipelined, point-to-point connections. Multiple data pieces are transferred simultaneously on the network on different pipelined stages. The number of stages is the number of tiles through which the data travels. As a result, the bandwidth is scalable with the size of the system. The wire length between cores is carefully chosen to balance the network speed, communication latency and core size.

- *Clock:* By limiting inter-core communication to short wires with predictable performance, high-speed communication can be achieved. Furthermore, the regularity of mesh architecture allows the network to work synchronously, which guarantees the simplicity and high speed of the communication network.

- *Capacitance:* ASoC has a pipelined architecture. At each stage, a communication interface drives a single core. As a result, the capacitance will not accumulate when system scales.

Compared to other scalable networks, aSoC is a low cost, heterogeneous flexible platform that fills the gap between the Network-on-Chip (NoC) and FPGA-like architectures. The aSoC platform provides a communication substrate for computation dense applications in the domains of DSP, multi-media and wireless communications. Most of these applications have highly predictable on-chip communication patterns, which enable a static scheduling network in aSoC. To reduce the hardware cost and improve the simplicity, aSoC reuses the wires using packet-switching based on static scheduling. Furthermore, aSoC combines the static network with certain dynamic scheme, which enables aSoC to adapt the network dynamically to the communication pattern changes and improves the flexibilities of aSoC. The static scheduling scheme with limited dynamic function makes aSoC simpler than the NoC approach.

1.3 Supporting Software

In aSoC, the static communication scheduling is applied to optimize its inter-core communication. While this job can be done manually for two or three cores, an automated software tool is required for larger aSoC chips. An application mapping tool, AppMapper, has been developed to translate high-level language application representations to aSoC devices.

The compilation is made more complicated by the heterogeneous cores. To maintain the flexibility of core selection, AppMapper leaves the core-related optimization for the third-part core compilers. AppMapper outputs the instructions for IP cores in the format of high level languages like C, Verilog or other control codes. They can be easily translated into core specific machine codes using the compilers provided

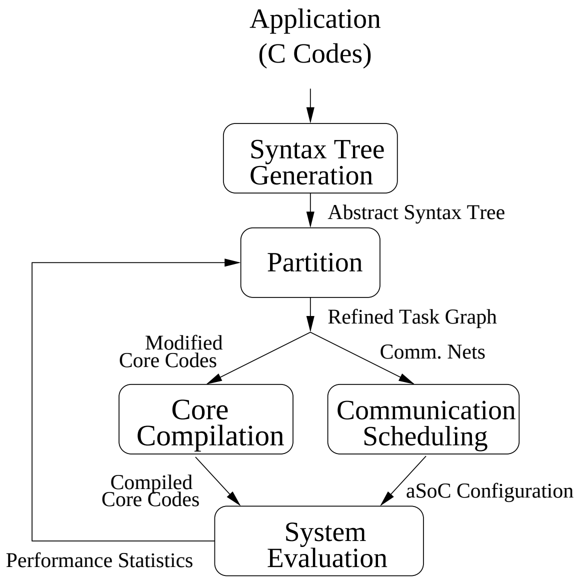


Figure 1.2. Flowchart of AppMapper

by the core vendors. As a result, users are able choose the cores appropriate for their applications without modifying the kernel of AppMapper.

As shown in Figure 1.2, compiler steps, including syntax tree generation, code partitioning, communication scheduling, system evaluation and core-dependent compilation are part of the AppMapper flow. Although each step has been fully automated, design interfaces for manual intervention are provided to improve mapping efficiency. Based on the performance statistics from system evaluation, manual interventions are allowed for better aSoC partition. Novel algorithms based on advanced cost functions have been developed for both partitioning and scheduling based on heuristic techniques.

A system-level simulator allows for performance evaluation prior to design implementation.

1.4 Applications

Key components of the aSoC architecture, including the communication interface architecture, have been simulated and implemented in 0.18 micron technology.

Experimentation results show that a communication network speed of 400MHz with a 6% communication area overhead to on-chip IP cores.

A set of applications have been developed and tested for the aSoC devices. The applications are chosen carefully from multi-media, DSP and communication domains, including MPEG-2 encoder/decoder, IIR filter and an image processing kernel, Doppler radar and OFDM modulator. These applications have been mapped to 9 and 16 core aSoC devices via the AppMapper tool.

While the above applications are relative small and regarded as DSP cores rather than applications, a Turbo decoding system has been mapped onto the aSoC platform. This turbo code simulation system includes a turbo encoder, a channel simulator, and a turbo decoder. The turbo decoder employs a novel adaptive soft-output Viterbi algorithm (ASOVA) which reduces the complexity of the Soft-Output Viterbi Algorithm [48] with a competitive error correction ability. In addition, ASOVA can adapt its decoding complexity according to the input signal to noise ratio (SNR). This Turbo decoder is mapped onto an Altera Stratix FPGA on the Nios development board [6] to verify its functionality. After obtaining the required parameters from the FPGA testing, the Turbo decoder is simulated on aSoC devices. The advantages of the high data bandwidth of aSoC are depicted in the comparison to other implementations of the same Turbo code simulation system on FPGA and bus-based architectures.

Performance comparison between aSoC implementations and other more traditional on-chip communication platforms, such as an on-chip bus, show an aSoC performance improvement by up to a factor of five. Simulation results also reveal that aSoC have better scalability for systems with more cores over bus architecture.

1.5 Contributions

The contribution of this dissertation includes four major parts.

- Construct the architecture of aSoC. A novel statically scheduling packet-switching on-chip network, aSoC, is designed for SoCs. ASOC applies a pipelined architecture which solves the problem of the inter-core communication bottleneck of next generation SoCs.

- Develop an automated compiling software and simulator, AppMapper, for aSoC. AppMapper is able to map high level language applications into aSoC silicon. The simulator traces the network cycle by cycle and provides the system performance information. Furthermore, it can also simulate other communication architectures, such as bus, hierarchical bus, dynamic networks, etc. with minor setting changes.

- Validate the aSoC architecture using software simulation. Based on the layout parameters, aSoC test chips with 9 cores and 16 cores are created. DSP applications are mapped onto these aSoC test chips. For the purpose of comparison, models are also established using the same topology with other communication architectures, namely, bus, hierarchical bus, and dynamic network. Experimental results show that the implementation using aSoC architecture runs up to five times faster than bus architecture, and outperforms other models.

- Develop a novel Turbo code decoding algorithm, ASOVA, which simplifies the complexity of SOVA [48] while maintaining the error correction ability. This Turbo code system is mapped onto aSoC to demonstrate the aSoC performance with larger applications.

Publications related to this dissertation include:

1. Jian Liang, Andrew Laffely, Sriram Srinivasan, and Russell Tessier, An Architecture and Compiler for Scalable On-Chip Communication, *Transaction of VLSI* (accepted).

2. Jian Liang, Russell Tessier, and Dennis L. Goeckel, A Dynamically-Reconfigurable, Power-Efficient Turbo Decoder, in *FCCM'04 (Field-Programmable Custom Computing Machines)* (submitted)

3. Jian Liang, Russell Tessier, Oskar Mencer, Floating Point Unit Generation and Evaluation for FPGAs, in *FCCM'03 (Field-Programmable Custom Computing Machines)*, Napa, CA. Apr. 2003.

4. Jian Liang, Sriram Swaminathan, Russell Tessier, aSoC: A Scalable, Single-Chip Communications Architecture, in *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques,* Philadelphia, PA. October 2000.

5. Andrew Laffely, Jian Liang, Wayne Burleson and Russell Tessier, Adaptive System on a Chip (aSoC): a Backbone for Power-Aware Signal Processing Cores, in *Proceedings of IEEE International Conference on Image Processing,* Barcelona, Spain, Sep. 2003.

6. Andrew Laffely, Jian Liang, Prashant Jain, Ning Weng, Wayne Burleson and Russell Tessier, Adaptive Systems on a Chip (aSoC) for Low-Power Signal Processing , in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers,* Monterey, California, November 2001.

7. Andrew Laffely, Jian Liang, Russell Tessier, C. Andras Moritz and Wayne Burleson, Power-Aware System on a Chip, in *the Boston Area Architecture Workshop,* Boston, MA, Jan. 2003

1.6 Overview

This dissertation presents a review of SoC-related communication architectures and the state-of-the-art SoC compiling tools in Chapter 2. Chapter 3 reveals the details of our communication architecture and explains its suitability for on-chip interconnection. Chapter 4 demonstrates the application mapping methodology and describes component algorithms. Simulation environment and experimental approach are described in Chapter 5, and Chapter 6 introduces the DSP applications that are mapped onto aSoC devices. To further test aSoC, a Turbo decoding system has been developed for aSoC. Chapter 7 reveals the adaptive Soft-output Viterbi algorithm (ASOVA), which is developed to reduce the decoding complexity. Chapter 8 explains the architecture to implement ASOVA in hardware. Chapter 9 verifies the FPGA implementation of the ASOVA Turbo decoder and maps it onto aSoC devices. The required parameters for aSoC implementation and performance results obtained from the FPGA implementation are presented. In Chapter 10, the experimental results for aSoC devices are presented. These results are compared against the performance of alternate interconnect approaches. Finally, Chapter 11 concludes this dissertation and proposes the potential research directions.

Chapter 2

Background

Various communication architectures are used on System-on-a-Chip (SoC) and embedded system designs. This chapter briefly reviews these on-chip communication architectures. An attempt is made to group the existing communication platforms by their structures. The advantages and pitfalls of each group are analyzed, and figures are shown to better illustrate the trade-offs between architecture groups and the position of aSoC in the design space. Furthermore, the supporting softwares for SoCs are also reviewed, and the difference from the software tool for aSoC, AppMapper, is pointed out.

2.1 On-Chip Communication Architectures

Examination of current on-chip protocols provides insight into the requirements of on-chip interconnect and its differences from traditional multiprocessor networks. Notable, common threads through these on-chip interconnect architectures include the *simplicity* of the required circuitry, the *flexibility* support for numerous interconnection topologies and the *connectivity* provided for each core. Simplicity is the resource cost to for the hardware, and the connectivity here means the average data bandwidth that the communication architecture can provide for each local core. The flexibility is defined as how easy a system can change its communication pattern to establish a new connection. For example, buses provide high flexibility for the connected cores, while fixed point-to-point connections have a limited flexibility since a new chip has to be designed to allowed for another connection.

Project	Cite	First Pub.	Arch.	Flex.	Conn.	Cores
Coral	[13]	2000	Fixed/Bus	Low	Low	Heter.
Daytona	[2]	2000	Bus	High	Low	Heter.
IDT	[52]	2000	Bus	High	Low	Heter.
AMBA	[39]	1997	Hier. Bus	High	Median	Heter.
CoreConnect	[54]	2000	Hier. Bus	High	Median	Heter.
PI Bus	[78]	1997	Hier. Bus	High	Low	Heter.
MATRIX	[74]	1996	2D Mesh	Median	Median	Homo.
CHESS	[70]	1999	Hex Array	Median	Median	Homo.
DP-FPGA	[24]	1994	FPGA-like	Median	Median	Homo.
Pleiades	[66]	1997	Hier. Crossbar	Median	Median	Heter.
RAW	[103]	1997	2D Mesh	High	High	Homo.
Dally	[31]	2000	2D Mesh	High	High	Homo.
aSoC	[67]	2000	network	Media	High	Heter.

Table 2.1. Summary of On-Chip Architectures

Since the cost of hardware resources heavily depends on the system size and architecture choices, the simplicity will be addressed later in this section while the trade-off between the connectivity and flexibility for the different communication architecture designed will be first discussed.

Since Numerous on-chip interconnect approaches have been proposed as a means to connect intellectual property (IP) cores, for better understanding this trade-off, these approaches can be classified based on the architecture to five groups: fixed point-to-point connection, arbitrated buses, hierarchical buses connected via bridges, FPGA-liked static switching networks and Network-on-Chip (NoC). A few examples and their classifications are listed in Table 2.1.

An overview of these architects on the design space of connectivity and flexibility is shown in Figure 2.1. The circles represent the approximate location of each group in the design space, however, due to the various trade-offs, designs in the same group might be scattering around and not restricted by the circle. The achievable trade-offs are given by the shaded area.

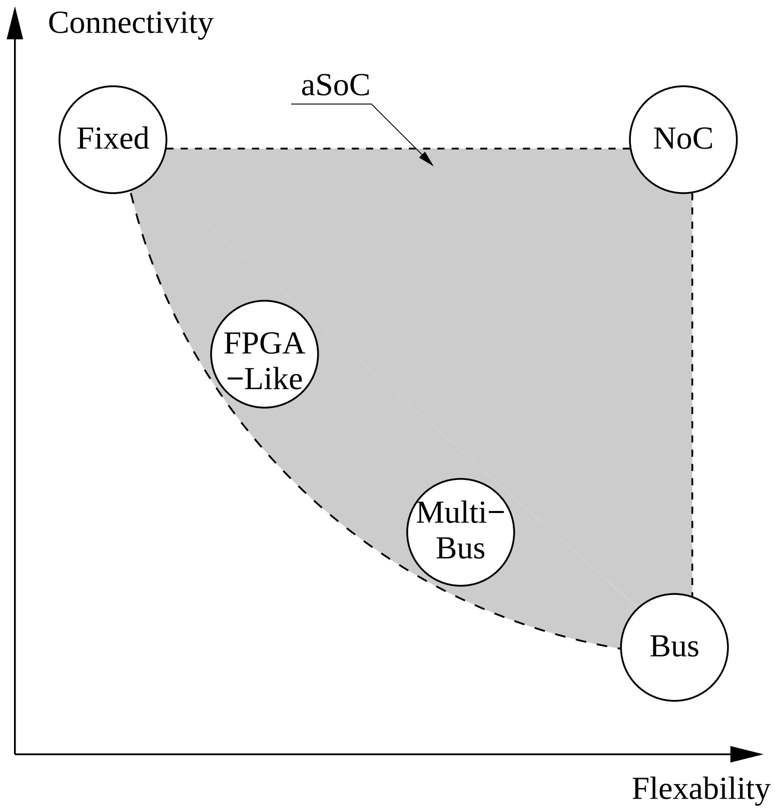


Figure 2.1. Design Space of SoC

2.1.1 Fixed Point-to-Point Connection

Using the fixed point-to-point connection architectures [32, 13], signals between cores are transferred by dedicated wires. These architectures are most used in small ASIC designs. All connections are decided at design time and can not be changed later.

Fixed connections have lowest flexibility and are located in the left-top corner in Figure 2.1. The connectivity of fixed connections is high, but it comes with un-affordable design cost. In addition, the required number of wires for such a system will increase exponentially when the number of cores goes up. It is unfeasible to route such a huge amount of wires except in a very regular mesh.

2.1.2 Arbitrated Bus

The bus architecture connects multiple cores using long bus lines. The bus lines are time-shared under the control of the bus arbitrator. The bus-based approaches include [52, 81, 94, 2]. In general, all of these architectures have similar arbitration characteristics to master/slave off-chip buses with several new features including data pipelining [52], replacement of tri-state drivers with multiplexers [81], and separated address/data buses due to the elimination of off-chip pin constraints.

These approaches are put in the right-bottom corner of Figure 2.1. While flexible, they have poor connectivity due to the centralized arbitration and the fixed bandwidth that has to be shared by all cores in a system.

2.1.3 Hierarchical Bus

Since an arbitrated bus is unable to fulfill the bandwidth requirement of multiple cores, hierarchical bus architectures are employed in many designs to increase connectivity. Hierarchical Bus combines several buses together using hierarchical nodes or bus bridges. Cores on the same bus can communicate without interfering with the cores on another bus, but communication across buses requires additional delay. Several current on-chip interconnects support the connection of hierarchical buses via bridges [4, 39], multiplexed memory/processor interconnect solutions [3], or variable topologies (e.g. partial crossbar, tree) [54].

AMBA [39] is a hierarchical bus model that consists high speed Processor Local Buses (PLB) and low speed On-chip Peripheral Buses (OPB). The PLB can be dedicated to the high-speed data transfers such as CPU and caches without interfering with slow peripheral IO units. A PLB can be connected to an OPB via a PLB-OPB bridge to allow data exchanges on these two buses. Other hierarchical bus architectures [4, 3, 54] apply similar schemes but utilize different protocols and hardware modules.

In Figure 2.1, Hierarchical bus is above the arbitrated bus. It increases the connectivity and allows the connections of more cores than a single bus architecture. On the other hand, while it still able to coordinate on-chip data paths among heterogeneous components as a bus architecture, certain flexibility is sacrificed since the some of the communications have to go across a bus bridge.

2.1.4 FPGA-Like Architecture

An alternate approach allows for static scheduling and programmable interconnect such as the type found in field-programmable gate arrays (FPGAs). To provide simplicity, this architecture employs a static network, whose configuration is decided at compile-time. With limited flexibility, the wires in such static networks are dedicated to a given communication link and does not support wire reuse. Most of these architectures can only change its communication network by rewriting the configurations [74, 24]. The connectivity of many FPGA-like architectures are also less than the fixed point-to-point connection since some nodes have to go across a couple switches to communicate. To simplify the architecture, the network is built for homogeneous small cores. Typical FPGA-like architectures can be found in MATRIX [74], CHESS Array [70], DP-FPGA [24] and Wan et al. [104]. As an exception, Pleiades [66] presents a hierarchical static crossbar-based network using heterogeneous coarse cores.

In Figure 2.1, it can be seen that the *FPGA-like* circle is located between the *Fixed* and *Bus*, which indicates that this architecture is not as flexible as bus or hierarchical bus, but better than fixed connections. On the other axle, FPGA-like architectures have better connectivity for its mesh communication network but weaker than fixed architectures due to the overhead of switches. This architecture is relatively cheaper in hardware resources with less flexibility and connectivity.

2.1.5 Network-on-Chip (NoC)

Network-on-chip (NoC) is a network of computational, storage and I/O resource, interconnected by a network of switches. Resources communicate with each other using addressed data packets routed to their destination by the switch fabric [50]. Similarly, researchers at Stanford University propose a SoC interconnect using packet-switching [31]. The idea of performing on-chip dynamic routing is described and simulated but is not yet implemented. MicroNetwork [106] provides on-chip communication via a pipelined interconnect. A rotating resource arbitration scheme is used to coordinate inter-node transfer for dynamic requests. This mechanism is limited by the need for extensive user interaction in design mapping.

The architecture of pipelined packet-switched interconnect has been used effectively for multiprocessor communication for over 25 years. Compile-time static routing of communication has been used effectively in a number of parallel processing systems. In iWarp [20], inter-processor communication patterns were determined during program compilation and were implemented with the aid of programmable, inter-processor buffers. This concept has been extended by the NuMesh project [92] to include collections of heterogeneous processing elements interconnected in a mesh topology. While pre-scheduled routing is appropriate for static data flows with predictable communication paths such as those found in DSP, most applications rely on at least minimal run-time support for data-dependent data transfer. Often, this support takes the form of complicated per-node dynamic routing hardware embedded within a communication fabric.

To reduce the affects of clock skew in large chips, the technique of Global Asynchronous Local Synchronous (GALS) are proposed [75, 105, 56, 77]. GALS provides asynchronous interfaces between cores which allows the cores to work on different clock speed. When solving the problems for data to travel across clock domain, GALS further increases the complexity of NoC architecture.

Recently, researcher in Fulcrum Microsystems presents the Nexus, a verified GALS interconnect ready for commercial usage [69], which is able to efficiently interconnect SoCs modules with different clock domains. The Nexus system features a crossbar with the IP cores around it. While heterogeneous IPs with different clock frequencies are allowed, clock-domain converters connect the local synchronous modules to the asynchronous crossbar. The crossbar, which is designed with the asynchronous-circuit techniques and based on the quasi-delay-insensitive (QDI) timing model, carry the data across the chip. All parts of the system are safely flow controlled and arbitration is employed to resolve the contention on output ports. The Nexus system supports 16 ports with each of 36-bit. In the 130-nm generic logic process, it operates at a frequency of 480 MHz at 1.5V and 25^oC. For a typical Nexus system, 4.15 mm^2 is used. The Nexus system provides an efficient on-chip interconnect architecture for SoCs with different clock domains. However, since the asynchronous crossbar can be occupied by only one port at a time, data congestion is unavoidable, which limits the scalability of the system.

To be applied in on-chip communications, NoC provides a highly scalable solution. For this reason, NoC is located on the far top of Figure 2.1. It requires complicated circuits for the dynamic routing, buffering, flow control and interfacing. This solution is proposed for next generation large systems having over one billion transistors [50]. To reduce the overhead of communication network, a coarse granularity is preferred.

A recent example of this approach can be found in the Reconfigurable Architecture Workstation (RAW) project [103], which provides both a static (determined at compile-time) and dynamic (determined at run-time) network. Limiting the dynamic routing and relying mostly on the static network, RAW simplifies its architecture and sacrifices flexibility.

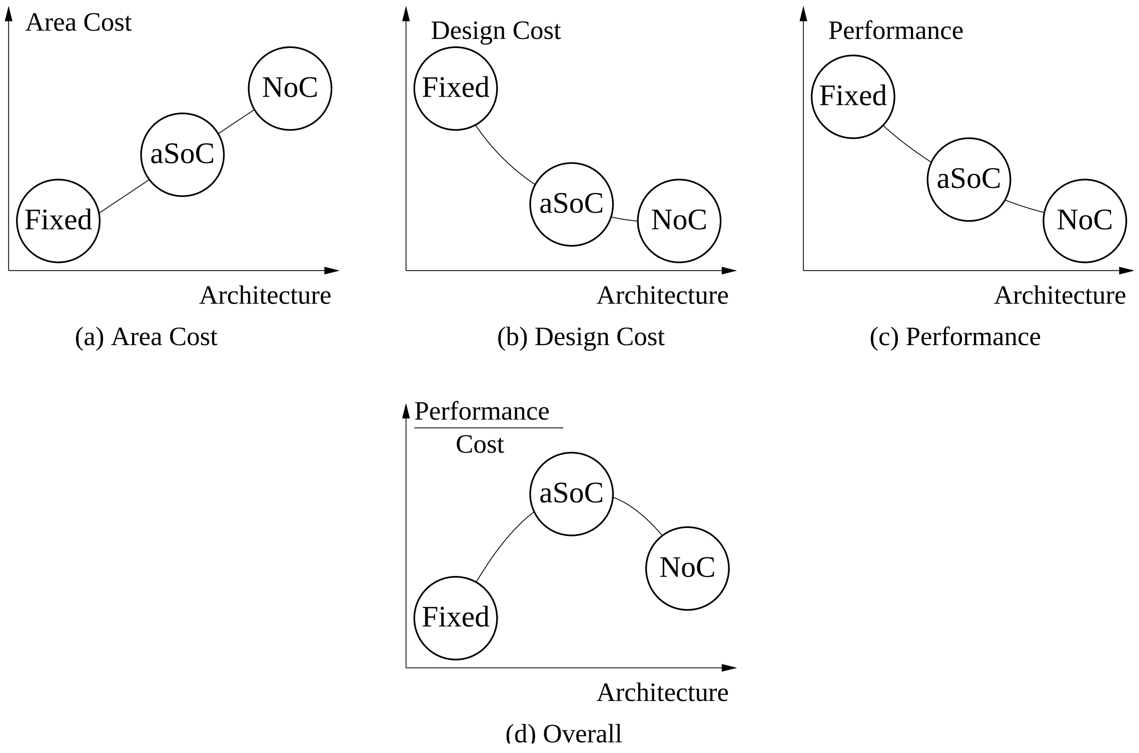


Figure 2.2. Illustration of Relative Tradeoff between Architectures

2.1.6 Comparisons on Performance and Cost

As shown in Figure 2.1, both fixed point-to-point architecture and NoC have high connectivity. The aSoC has been designed to have high connectivity in the design space. By trading flexibility, aSoC is able to provide required communication bandwidth with affordable circuit complexity. Further analysis has been done in the following to reveal the the trade-offs between these architectures.

Besides connectivity and flexibility, other important aspects of hardware architectures include simplicity and performance. Since quantitative comparison depends on the detailed architecture of a module, relative analysis is used to explore their relationships. Figure 2.2 presents a relative comparison between the three architectures of aSoC, NoC and fixed in terms of area cost, design time cost, and performance.

The fixed point-to-point architecture, as a customized design architecture, can achieve top performance with more efficient area usage than aSoC and NoC as shown conceptually in Figure 2.2(c) and Figure 2.2(a). However, as the system size increases, this advantage will be diluted by the huge amount of wires required for full connection. On the other hand, the design time cost of fixed point-to-point architecture is very expensive when any change in a system is required. The whole communication architecture has to be redesigned if a IP core is removed from or added into a system. The prohibitive design cost violates the divide and conquer idea of SoCs and generates poor performance to cost ratio as illustrated in Figure 2.2(d).

On the contrary, the design cost for NoC is almost ignorable. NoC employs a regular communication network for the IP cores. Once the network and the interfaces are set up, it costs little to put in another IP core into the system. As illustrated in Figure 2.2(b), NoC achieves the lowest design cost among the three architectures. As mentioned in Section 2.1.5, the NoC design comes with the cost of complex circuitry to deal with the packets and routing, which increases the area usage and reduces the performance.

2.1.7 Position of aSoC in the Design Space

As described above, present on-chip communication architectures have their pitfalls while providing some useful features. The NoC requires huge area resources to support its flexible communication network, and no physical design has been established as far as we know. The fixed point-to-point architecture has efficient area mapping, but its cost to design the system is prohibitive when IP cores need to be changed. FPGA-like architectures are limited by their flexibility and the bus architectures are limited by the bandwidth they can provide. To fulfill the requirements of the next generation on-chip communication, a new architecture has to be created.

ASoC is designed to fill in the gap between these existing communication architecture for the next generation SoCs. Compared to the above mentioned on-chip communication methodologies, aSoC provides a better overall performance to cost ratio. For SoCs, the cost means not only the area resources but also the design cost. The concept of SoC is proposed to reduce the prohibitive design cost for chips with large amount of transistors. A communication architecture for SoCs cannot ignore the design cost and just focus on other issues. As a result, the evaluated cost includes both the area resources and the design cost.

Several techniques are employed to guarantee aSoC a better performance to cost ratio. A packet-switching scheme is used in aSoC to provide high connectivity, and pipelined near-neighboring wires enable aSoC a high clock speed. By using static-scheduling, aSoC can be implemented with a simpler circuitry than NoC. To better understand the position of aSoC in the design space, the relationships between aSoC and other architectures in terms of connectivity, flexibility, cost and performance are discussed.

- *Connectivity.* One of most important goals for aSoC is to provide scalable communication bandwidth for SoCs. High connectivity is necessary for aSoC to achieve this goal. A regular 2-D mesh communication network is employed to provide the required bandwidth. In such an mesh, each core can communicate with its four neighbors in a clock cycle. A packet-switching scheme is used on the aSoC network to transfer the data between local cores, which allows wires to be shared by different communications at different time cycles so that limited number of wires are able to provide a connectivity competitive to fixed point-to-point architecture. As shown in Figure 2.1, aSoC is located at the same connectivity level of NoC and fixed point-to-point.

- *Flexibility.* ASoC has a higher flexibility than the FPGA-like and fixed point-to-point architectures. With a packet-switching scheme, the connections of the aSoC network is changed by control instructions every cycle in every network node. As a comparison, fixed point-to-point architecture can not change its communication pattern once the chip is fabricated, and the FPGA-like architecture can only modify its connecting settings once per application by reconfiguring the entire chip. In Figure 2.1, aSoC is to the left of both fixed point-to-point and the FPGA-like architecture.

  Compared to NoC, aSoC has lower flexibility in order to reduce circuit complexity. ASoC employs a static scheduling scheme, with which the communication patterns are pre-scheduled at compile time. The changes of the aSoC network connections are controlled by pre-set instructions, unlike the fully run-time dynamic changing in NoCs or buses. Further analysis indicates that the reduced flexibility of aSoC does not affect the performance of aSoC. ASoC is designed for the high data rate DSP applications. In many cases, the DSP applications are running with predictable communication patterns which are able to be determined by the static scheduling at compile time. In addition, aSoC also supports dynamic communication pattern switching with the dynamic data dependent control. With all the techniques, aSoC is flexible enough to support its target applications.

- *Area Cost.* As mentioned above, the loss in flexibility buys aSoC a lot of simplicity by employing a static scheduling and avoiding the dynamic arbitration. The concepts of NoC have been simulated in software to validation its performance, but, as far as we know, no hardware implementation has been built due to its resource requirement for arbitration and buffers. With static scheduling, aSoC is able to achieve high connectivity with a simple architecture, which will be described in detail in Chapter 3.

Simpler than NoC, aSoC is considered more expensive than a customized fixed point-to-point architecture in small system, which is illustrated in Figure 2.2(a). However, in large systems, fixed point-to-point architectures will lose because of their exponential increase in system size.

- *Design Cost.* ASoC provides a communication platform for heterogeneous SoCs. It allows SoCs to choose various IP cores from pre-designed libraries or third-part IP core vendors. With a regular mesh architecture and well-defined the interfaces between cores and the communication network, low design cost is required to design a SoC containing different cores. As a result, aSoC has a low design cost compared with the customized fixed point-to-point architecture, which is presented in Figure 2.2(b).

  While carefully compared to NoC architecture, aSoC supports heterogeneous cores. As a result, the aSoC interface has to communicate across clock domain to various local IP cores. It introduces some more design overhead compared to NoC. Once a proper core interface protocol is established, this overhead is negligible and results in a design cost very close to NoC. In addition, the overhead is worthwhile considering the performance benefits brought by the heterogeneous cores suitable for applications.

- *Performance.* ASoC architecture aims to provide scalable communication bandwidth for SoCs. Based on the common sense of scalability, aSoC should be able to improve performance with the size of the system increases. To achieve this goal, the packet-switching mesh network is employed to guarantee a high connectivity, the near-neighboring pipeline wires are used for high clock speed, and static-scheduling is designed to avoid data congestion. With all these schemes, aSoC can achieve a better performance than the NoC since it can run at a high clock speed and NoC requires the dynamic arbitration and suffers from the data congestion.

ASoC is not designed to beat the performance of fixed point-to-point architecture, since, with a customized design, fixed architecture can be refined for a given application with sufficient effort in designing. Figure 2.2(c) illustrates the relationship in performance between aSoC, NoC and fixed point-to-point architecture.

Combining the features mentioned above, aSoC achieves a better $performance/cost$ and fills in the gap between the NoC, FPGA-like and fixed point-to-point architectures. As illustrated in Figure 2.2(d), aSoC has better performance to cost ratio than fixed point-to-point architecture and NoC. The static scheduling packet-switching architecture of aSoC avoids the prohibitive design cost required in fixed point-to-point architectures. Compared to NoC, aSoC wins because of its smaller area resources requirement and better performance.

2.2 SoC Supporting Softwares

Generally, an application is specified in a high-level language(HLL) such as C/C++ or FORTRAN. A compiler converts the HLL into an intermediate representation(IR), which contains the necessary control and data dependency information. It is then applied to a more complex and machine-dependent phase that includes the following stages: instruction selection, scheduling, resource allocation, code optimizations/transformations, and code generation [95]. Unfortunately, the effectiveness of these stages depend heavily on the target architecture, chosen algorithm, and optimal ordering. So, for a core-changeable SoC, pre-existing core compilers are often applied in optimizing the codes for each core.

To date, few integrated compilation environments have been created for heterogeneous systems-on-a-chip. The MESCAL system [60] provides a high-level programming interface for embedded SoCs. Though flexible, this system is based on a communication protocol stack which may not be appropriate for data stream-based communication. Several projects [62] [63] have adapted embedded system compilers to SoC environments. These compilers target bus-based interconnect rather than a point-to-point network. Previous work in system synthesis provides some direction regarding compilation for SoC systems. Cost-based tradeoffs between hardware, software, and communication were evaluated by Wan et al. [104]. In Dick and Jha [84], partitioning was followed by a hill-climbing based task placement stage. In Lee et al. [64], a loop-based partitioner and space-time scheduler were used to isolate tasks to specific cores and to coordinate communication. This software system and the system described in [104] contain front-end interfaces to high-level languages through standard intermediate forms.

As another example, Pleiades[66] uses an energy-conscious methodology to guide its mapping and partitioning. Most commercial SoC compilers, such as PSoC[29] and CSoC[108], allow users to specify the task for each core. In Triscend CSoC compiler, FastChip, users can choose either software, dedicated hardware or configurable hardware to implement a given function. Either HLL(c,HDL) or net-lists can be used as the input. A built-in compiler is then applied to convert the application into configure data for CSoC.

But most compilers are unable to coordinate the inter-core communications to reduce potential congestion, since they target a bus or hierarchical bus template. Some compilers for packet-switched systems did reduce congestion. As mentioned above, RAW[64], NuMesh[92], and iWarp[20] support static scheduling and packet-switched communication. Their supporting software can pre-schedule the data communication at compile time to avoid congestion.

iWarp ConSet model schedules a phase-based communication model. The phase means a group of pathways which can be active at the same time. Each iWarp cell has 20 logic channels allowing up to 20 pathways at one time. Pathways in different phases can share the hardware resources. Extra control circuits and time are dedicated to phase changing. Because of the synchronization of the cells, the time required for phase changing is quite significant. Dijkstra's algorithm is applied in ConSet to find the shortest path for pathways. On NuMesh, a *C*ommunication *OP*erator language(COP[71]) is built to optimize the phase and data transfer organizations. The routing kernel of COP is developed from hierarchical commodities flow routing. The RAW machine assumes that the network contention is low. Most performance improvement focus on the scheduler. So, the dimension-ordered routing is employed.

As a compiler for aSoC, AppMapper differs from the above compilers as explained below:

- *No congestion:* Only static routing is used in AppMapper, which means that all data communications are scheduled by AppMapper in such a way that runtime congestion cannot happen.

- *Scalability:* Instead of hierarchical commodities flow algorithm, AppMapper's space-time routing is based on a shortest path search algorithm whose complexity is associated with the number of data streams not the system size.

- *Simpler:* aSoC communication architecture is more compact than multi-processor systems. All the data communication is scheduled in one loop. Without the overhead of phase-changing as in iWarp and NuMesh, the communication interface saves silicon area.

- *Decouple with cores:* Rather than being fixed, the cores in aSoC can be selected from a group of candidates.

- *Higher throughput:* Without the congestion of bus and the phase-changing and dynamic routing overhead of multiprocessor systems, AppMapper expects high throughput.

Chapter 3

aSoC Overview

3.1 aSoC Design Philosophy

Successful deployment of adaptive systems-on-a-chip (aSoC) requires the architectural development of an inter-node *communication interface*, the creation of supporting design mapping software, and the successful translation of target applications. Before discussing these issues, the basic operating model of aSoC interconnect is presented.

3.1.1 Design Overview

As shown in Figure 1.1, a standardized communication structure provides a convenient framework for the use of intellectual property (IP) cores. A simple core interface protocol, joining the core to the communication network, creates architectural modularity. By limiting inter-core communication to short wires exhibiting predictable performance, high-speed point-to-point transfer is achieved. Since heterogeneous cores can operate at a variety clock frequencies, the communication interface provides both data transport and synchronization between processing and communication clock domains.

Inter-core communication using aSoC takes place in the form of data *streams* [92] which connect data sources to destinations. To achieve the highest possible bandwidth, our architecture is targeted towards applications, such as video, communications, and signal processing, that allow most inter-core communication patterns to be extracted at compile time. By using the mapping tools, described

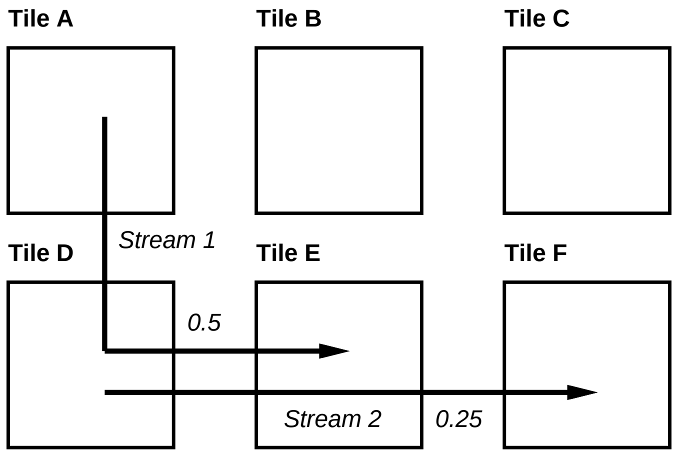


Figure 3.1. Multi-core data streams 1 and 2. This example shows data streams from Tile A to Tile E and from Tile D to Tile F. Fractional bandwidth usage is indicated in italics.

in Chapter 4, it is possible to determine how much bandwidth each inter-core data stream requires relative to available communication channel bandwidth. Since stream communication can generally be determined at compile time [92], our system can take advantage of minimized network congestion by scheduling data transfer in available data channels.

As seen in Figure 3.1, each stream requires a specific fraction of overall communication link bandwidth. For this example, *Stream 1* consumes $\frac{0.5}{1}$ of available bandwidth along links it uses and *Stream 2* requires $\frac{0.25}{1}$. This bandwidth is reserved for a stream even if it is not used at all times to transfer valid data. At specific times during the computation, data can be injected into the network at a lower rate than the reserved bandwidth, leaving some bandwidth unused. In general, the path taken by a stream may require data transfer on multiple consecutive clock cycles. On each clock cycle, a different stream can use the same communication resource. The assignment of streams to clock cycles is performed by a communication scheduler based on required stream bandwidth. Global communication is broken into a series of step-by-step hops that is coordinated by a distributed set of individual

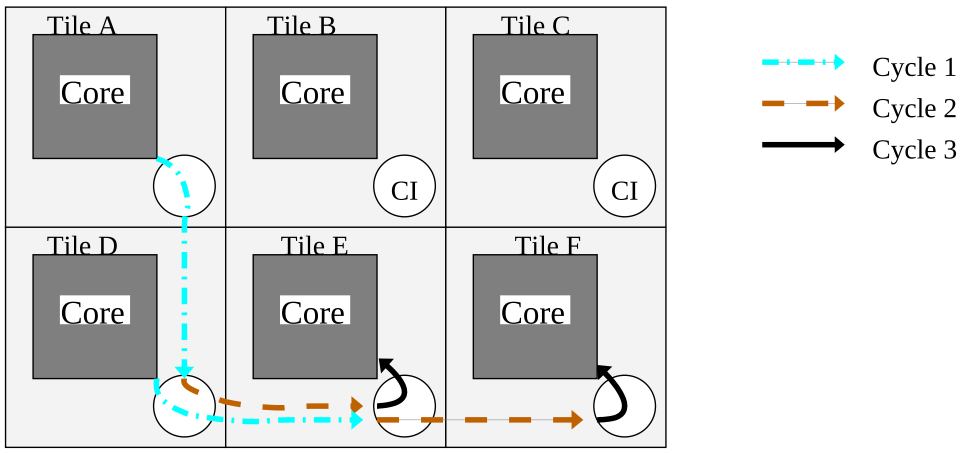


Figure 3.2. Pipelined stream communication across multiple communication interfaces

tile communication schedules. During communication scheduling, near-neighbor communication is coordinated between neighboring tiles. As a result of this bandwidth allocation, the dynamic timing of the core computation is decoupled from the scheduled timing of communications.

The cycle-by-cycle behavior of the two example data streams in Figure 3.1 is shown in Figure 3.2. For *Stream 2*, data from the core of *Tile D* is sent to the left (West) edge of *Tile E* during communication clock cycle 0 of a four-cycle schedule. During cycle 1, connectivity is enabled to transfer data from *Tile E* to the West edge of *Tile F*. Finally, in cycle 2 the data is moved to its destination, the core of *Tile F*. During four consecutive clock cycles, **two** data values are transmitted from *Tile A* to *Tile E* in a pipelined fashion forming *Stream 1*. Note that the data stream is pipelined and the physical link between *Tile D* and *Tile E* is shared between the two streams at different points in time. Stream transfer schedules are iterative. At the conclusion of the fourth cycle, the four-cycle sequence re-starts at cycle 0 for new pieces of data. The communication interface serves as a cycle-by-cycle switch for stream data. Switch settings for the four-cycle transfer in Figure 3.2 are shown in Table 3.1.

Cycle	Tile A	Tile D	Tile E	Tile F
0	core to south	core to east		
1	core to south	north to east	west to east	
2		north to east	west to core	west to core
3			west to core	

Table 3.1. Communication schedules for tiles in Figure 3.2

Stream-based routing differs from previous static routing networks [20]. Static networks demand that all communication patterns be known at compile time along with the *exact* time of all data transfers between cores and the communication network. Unlike static routing, stream-based routing only requires that bandwidth be allocated but not necessarily used during a specific invocation of the transfer schedule. Communication is set up as a pipeline from source to destination cores. This approach does not require the *exact* timing of all transfers, but rather, data only needs to be inserted into the correct stream by the core interface at a communication cycle allocated to the stream. Computation can be overlapped with communication in this approach since the injection of stream data into the network is decoupled from the arrival of stream data.

3.1.2 Flow Control

Since cores may operate asynchronously to each other, individual stream data values must be tagged to indicate validity. When a valid stream data value is inserted into the network by a source core at the time slot allocated for the stream, it is tagged with a single *valid* bit. As a result of communication scheduling, the allocated communication cycle for stream data arrival at a destination is predefined. The data valid bit can be examined during the scheduled cycle to determine if valid data has been received by the destination. If data production for a stream source temporarily runs ahead of data consumption at a destination, data for a specific stream can temporarily back up in the communication network. To avoid deadlock, data buffer storage is required in each intermediate communication interface for each stream passing through the interface. With buffering, if a single stream is temporarily blocked, other streams which use the affected communication interfaces can continue to operate unimpeded. A data buffer location for each stream is also used at each core-communication interface boundary for intermediate storage and clock synchronization.

The use of flow control bits and local communication interface buffering ensures data transfer with the following characteristics:

- All data in a stream follows the same source-destination path.
- All stream data is guaranteed to be transfered in order.
- In the absence of congestion, all stream data requires the same amount of time to be transfered from source to destination
- Computation is overlapped with communication.

3.1.3 Run-time Stream Management

For a number of real-time applications, inter-core communication patterns may vary over time. This requirement necessitates the capability to invoke and terminate streams at various points during application execution and, in some cases, to dynamically vary stream source and destination cores at run-time. In developing our architecture, we consider support for the following two situations: (1) all necessary streams required for execution are known at compile-time but are not all active simultaneously at run-time and, (2) some stream source-destination pairs can only be determined at run-time.

3.1.3.1 Asynchronous Global Branching

For some applications, it is desirable to execute a specific schedule of stream communication for a period of time, and then in response to the arrival of a data value at the communication interface, switch to a communication schedule for a different set of streams. This type of communication behavior has the following characteristics:

- All stream schedules are known at compile time.
- The order of stream invocation and termination is known, but the time at which switches are made is determined in a data-dependent fashion.
- A data value traverses all affected communication interfaces (tiles) over a series of communication cycles to allow for a *global* change in communication patterns.

This *asynchronous global branching* technique [64] across predetermined stream schedules has been shown [9] to support many stream-based applications, such as FFT, that exhibit time-varying communication patterns. The aSoC communication interface architecture supports these requirements by allowing local stream schedule changes based on the arrival of a specific data value at the communication interface. Depending on the value of the data, which is examined on a specific communication cycle, the previous schedule can be repeated or a new schedule, already stored in the communication interface, can be used. This technique does not require the loading of new schedules into the communication interface at run-time. Although our architecture supports run-time update of the schedule memory, our software does not currently exploit this capability. As a result, all required stream schedules must be loaded into the interface prior to run-time via an external interface and a shift chain.

Instruction No.	interface connection	next instruction	possible branch?	comment
0x0	core to north	0x1/-	N	*data transfer north*
0x1	core to interface control	0x0/0x2	Y	*test count value*
0x2	core to east	0x3/-	N	*data transfer east*
0x3	core to interface control	0x2/0x0	Y	*test count value*

Table 3.2. Data-dependent communication control branching for Tile D in Figure 3.3

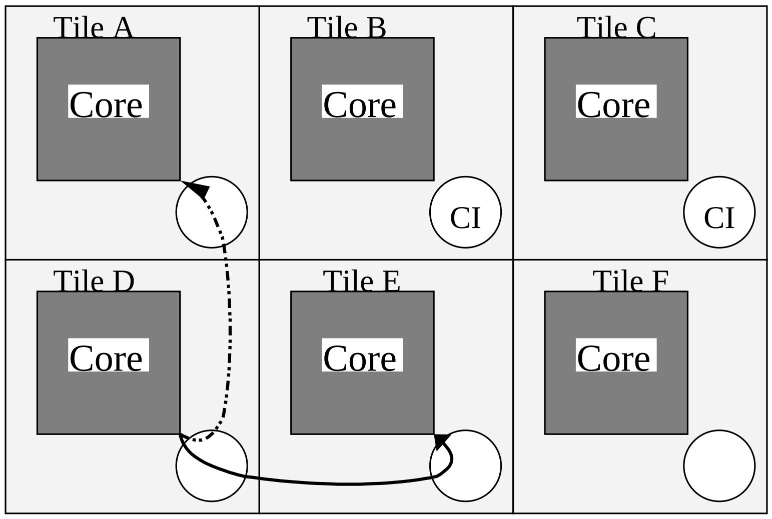


Figure 3.3. Example of distinct stream paths for two communication schedules which send data from a source to different destinations.

The use of these branching mechanisms can be illustrated through the use of a data transfer example. Consider a transfer pattern in which Tile D in Figure 3.3 is required to first send a fixed set of data to tile A and then send a different fixed set of data to Tile E. To indicate the need for a change in data destination, the Tile D core iteratively sends a value to its communication interface. When this value is decremented by the core to a value of 0, control for the communication schedule is changed to reflect a change in data destination. The two communication interface schedules for Tile D which supports this behavior are shown in Table 3.2. Each communication cycle is represented in the interface with a specific communication *instruction.* For cycles when data dependent schedule branching can take place, the target instruction for a taken branch is listed second under *next instruction.* In these cycles, data is examined by the interface control to determine if branching should occur. The Tile D - Tile A stream schedule uses instructions 0 and 1. The Tile D - Tile E stream schedule uses instructions 2 and 3.

3.1.3.2 Run-time Stream Creation

Given the simplicity of routing nodes and our goal to primarily support stream-based routing, communication hardware resources are not provided to route data from stream sources to destinations that have not been explicitly extracted at compile time (dynamic data). However, as we will show in Chapter 10, often streams extracted from the user program require only a fraction of the overall available stream bandwidth. As a result, a series of low-bandwidth streams between all nodes can be allocated at compile time via scheduling in a round robin fashion in otherwise unused bandwidth. Cores can take advantage of these out-of-band streams at run time by inserting dynamic data into a stream at the appropriate time so that data is transmitted to the desired destination core.

3.2 aSoC Architecture

The aSOC architecture augments each IP core with communication hardware to form a computational tile. As seen in Figure 3.4, tile resources are partitioned into an IP core and a communication interface (CI) to coordinate communication with neighboring tiles. The high level view of the communication interface reveals the five components responsible for aSOC communications functionality:

- **Interface Crossbar** - allows for inter-tile and tile-core transfer.
- **Instruction Memory** - contains schedule instructions to configure the interface crossbar on a cycle-by-cycle basis.

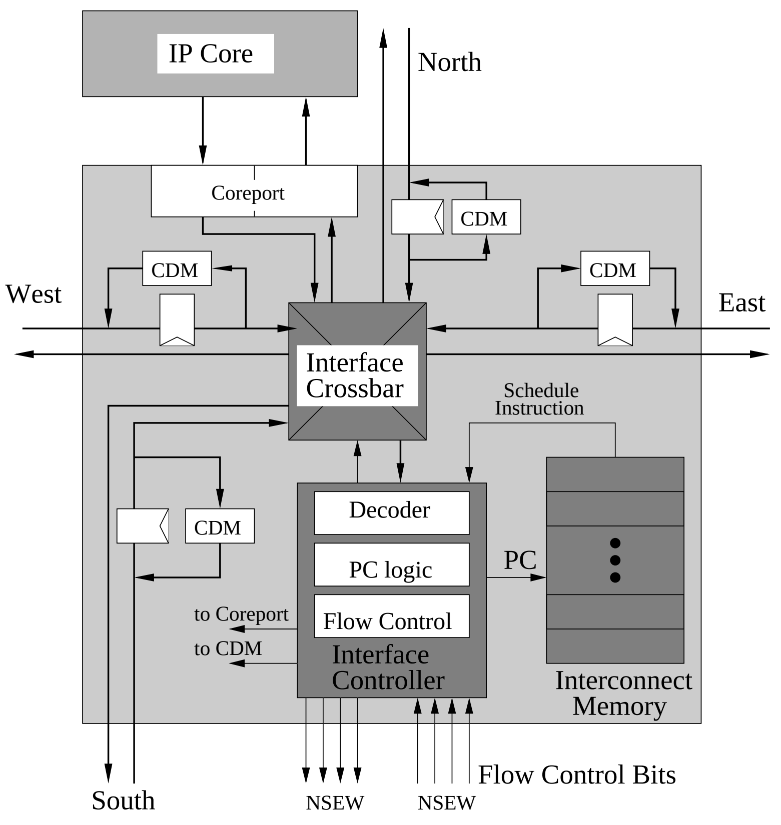


Figure 3.4. Core and Communication Interface

- **Interface Controller** - control circuitry to select an instruction from the instruction memory.
- **Coreport** - data interface and storage for transfers to/from the tile IP core.
- **Communication Data Memory (CDM)** - buffer storage for inter-tile data transfer.

The interface crossbar allows for data transfer from any input port (*North*, *South*, *East*, *West*, and *Coreport*) to any output port (five input directions and the port into the controller). The crossbar is configured to change connectivity every clock cycle under the control of the interface controller. The controller contains a program counter and operates as a microsequencer. If, due to flow control signals, it is not possible to transfer a data word on a specific clock cycle, data is stored

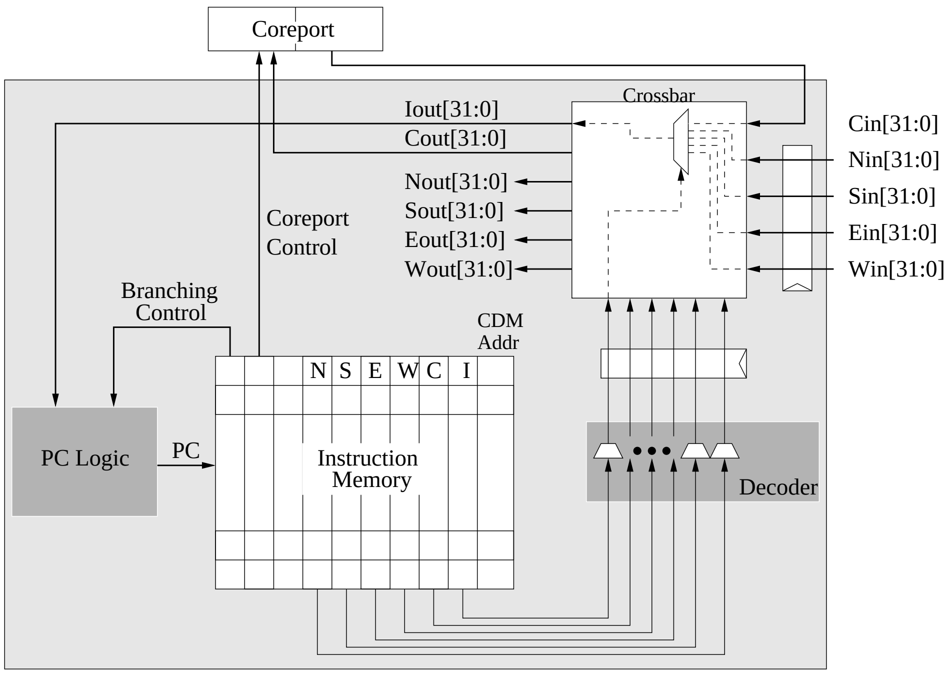


Figure 3.5. Detailed communication interface

in a communication data memory (CDM). For local transfers between the local IP core and its communication interface, the coreport provides data storage and clock synchronization.

3.2.1 Communication Interface

A detailed view of the communication interface appears in Figure 3.5. The programmable component of the interface is a 32-word SRAM-based instruction memory that dynamically configures the connectivity of the local interface crossbar on a cycle-by-cycle basis based on a pre-compiled schedule. This programmable memory holds binary code that is created by application mapping tools. Instruction memory bits are used to select the *source* port for each of the six interface destination ports (N_{out}, S_{out}, E_{out}, W_{out}, C_{out} for the core, I_{out} for the interface control). *CDM Addr* indicates the buffer location in the communication data memory (CDM), which is used to store intermediate routed values as described in Section 3.2.3. A program counter PC is used to control the instruction sequence. Branch control signals from

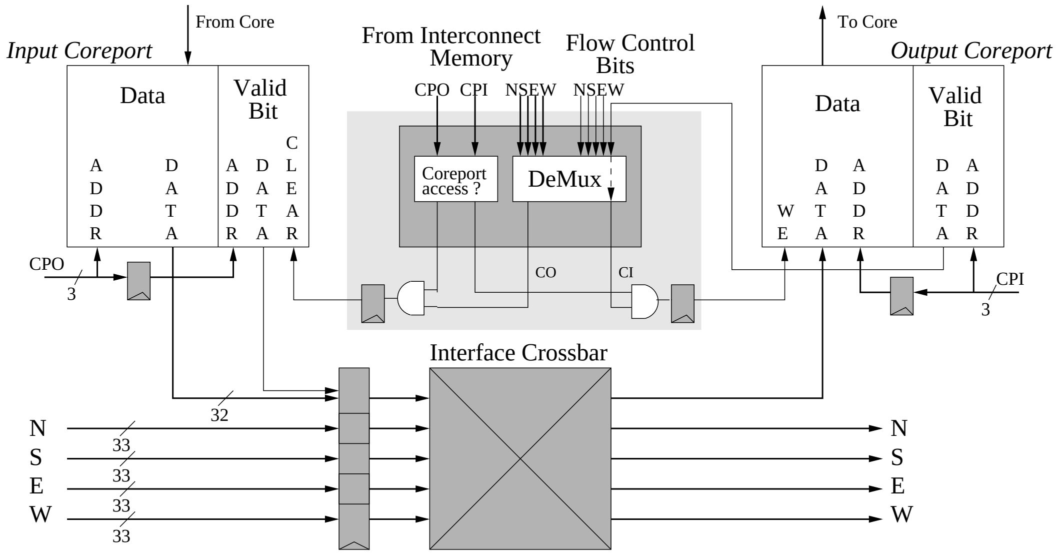


Figure 3.6. Input and output coreport interface

the instruction memory determine when data dependent schedule branching should occur. This control can include a comparison of the I_{out} crossbar output to a fixed value of 0 to initiate branching.

3.2.2 Coreports: Connecting Cores to the Network

The aSoC coreport provides a synchronization and buffering resource between a core and an associated communication interface. Coreport architecture is designed to permit interfacing to a broad range of cores with a minimum amount of additional hardware, much like a bus interface. Both core-to-interface and interface-to-core transfer are performed using asynchronous handshaking to provide support for differing computation and communication clock rates.

Figure 3.6 shows a high-level view of the coreport interface. Both *input* and *output* coreports contain dual-port memories (one input port, one output port). Each memory contains an addressable storage location for each individual streams, allowing multiple streams to be targeted to each core for both input and output. The structure of the ports allows other streams to continue transfer if an individual stream is blocked. Each data value in the input and output coreports is tagged with a *valid bit* to indicate valid data.

Coreport writes from the interface crossbar to the input coreport memory require two communication clock cycles. During the *first* transfer cycle the coreport input stream CPI is driven from the interconnect memory shown in Figure 3.5 to select the valid bit for the designated stream shown at the right in Figure 3.6. During the same cycle, the coreport source select bits (C in Figure 3.5) are used to select the source port and configure the interface crossbar. If the flow control bit attached to the source port data is valid and the valid bit in the input coreport memory is invalid, indicating available storage space, a successful coreport write can be completed on the following communication clock cycle. During the *second* transfer cycle the valid bit from the input coreport is used in conjunction with the flow control valid bits from the source to latch in the data. The control input bit CI is generated by the demultiplexer and conditioned by a decoded input stream indicator CPI to generate a write enable. After the write occurs, the input coreport valid bit is used to indicate successful transfer to a neighboring tile or CDM sending data. Note that, although not shown, a write to the input coreport will set the corresponding valid bit. Although coreport writes require two cycles, consecutive writes can be pipelined.

Reads from the output coreport also require two communication clock cycles. During the first clock cycle the data and the valid bit are read from the output coreport based on the coreport output stream indicator CPO from the interconnect memory. As shown in Figure 3.6, the data value is latched in the crossbar register at the start of the second communication clock cycle. During this cycle, the source select bits for the target destination port and associated flow control bits are demultiplexed to generate control output bit CO. This value indicates if storage space is available at the destination port. The control output CO is conditioned by

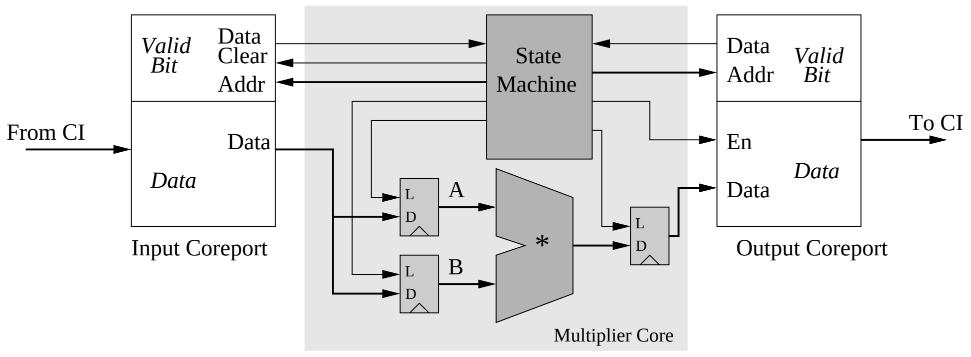


Figure 3.7. Multiplier coreport interface

a decoded stream indicator CPO to specify if the output coreport valid bit should be reset on the next rising clock edge, indicating a successful transfer.

The portion of the coreport closest to the core has been designed to be simple and flexible, like a traditional bus interface. This interface can easily be adapted to interact with a variety of IP cores. An example interface appears in Figure 3.7 for a multiplier core. Since coreport reads and writes occur independently, the network can operate at a rate that is different than that of individual cores. As shown in the figure, a state machine can control coreport/core interaction. A set of state machine operations that allow for the multiplication of the two data values includes:

- **State 0:** The valid bit of input data value A is accessed in the input coreport along with input data.
- **State 1:** If data is valid, data value A is latched into an input register and clear valid bit signal is asserted. If invalid, return to state 0.
- **State 2:** The valid bit of input data value B is accessed in the input coreport along with input data.
- **State 3:** If data is valid, data value B is latched into an input register and clear valid bit signal is asserted. If invalid, return to state 0.

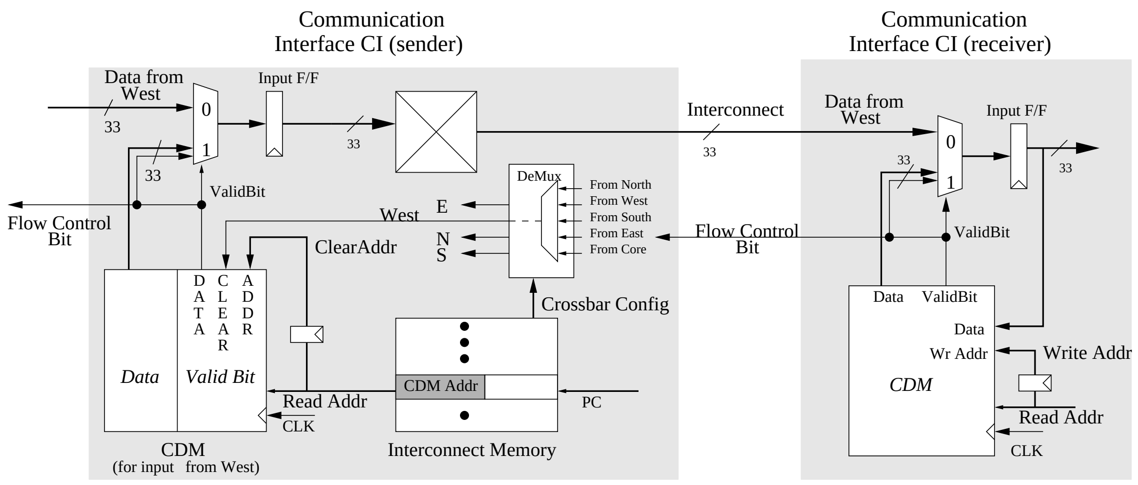


Figure 3.8. Flow Control Scheme

- **State 4:** The result from multiplication is latched and the valid bit at the stream location in output coreport is checked. If the location is full, a jump is made to State 4.

- **State 5:** The result is stored and the output coreport valid bit is set. A jump is made to state 0.

The sequenced nature of the data access prevents port data from being accessed out of order, even if network congestion delays data transfer. To maintain flexibility, the number of ports per coreport can be changed based on core requirements. While the above simple example provides an overview of coreport/core interfacing to illustrate transfer coordination, specific interfacing depends on the core. For example, as described in Section 5.2, a microprocessor can be interfaced to the coreport via a microprocessor bus.

3.2.3 Communication Data Memory

Due to network congestion or unequal source and destination core operating frequencies, it may be necessary to buffer data at intermediate communication interfaces. As shown in Figure 3.8, the communication data memory (CDM) provides one storage location for each stream that passes through a port of the communication interface. To facilitate interface layout, the memory is physically distributed across the N, S, E, W ports. On a given communication clock cycle, if a data value cannot be transfered successfully, it is stored in the CDM. The flow control bits that are transfered with the data can be used to indicate valid data storage. A full valid bit for a stored CDM value inhibits further transfer for the corresponding stream to the tile communication interface.

In aSoC devices, near-neighbor flow control and buffering is used. Figure 3.8 indicates the location of the communication data memory in relation to inter-tile data paths. On a given communication clock cycle, the $CDMAdr$ for each port (2 bits) indicates the stream that is to be transfered in the *next* cycle. Concurrently, the crossbar is configured by interconnect memory signals ($N..C$) to transfer the value stored in the crossbar register. This value is transfered to the receiver at the same time the receiver valid bit is sent to the transmitter. This bit indicates if the CDM at the receiver already has a buffered value for the transmitted stream. If a previous value is present at the receiver, the transmitted value is stored in the transmitter CDM using the write signals shown in the CDM at the right in Figure 3.8. A multiplexer at the input to the crossbar register determines if the transmitted or previously-stored value is loaded into the crossbar register on subsequent transfer cycles.

An example of inter-tile transfer from a sender tile to a receiver tile can be seen in Figure 3.8 for a two cycle transfer involving a data value stored in the CDM of the sender. During the first communication cycle stream data and valid bit are loaded in to the sender crossbar register. On the subsequent cycle this value is transfered to the receiver at the same time as the *flow control bit* from the receiver is sent to the sender. If this bit indicates that the CDM in the receiver is empty, the stream location in the sender is cleared. Note that the flow control (valid) bit also configures the multiplexer before the crossbar register in the receiver.

The use of stream based storage for the communication data memory allows for storage flexibility at the expense of increased communication data memory size. The decoupling of communication interface PC from the CDM allows for easier management of stream data across control jumps. Additionally, this approach is scalable with interconnect memory size.

C h a p t e r 4

AppMapper Compiler and Simulator

The aSoC application mapping environment, *AppMapper*, builds upon existing compiler infrastructure and takes advantage of user interaction and communication estimation during the compilation process. The mapping software contains a series of steps: front-end processing, basic block partitioning, computation and communication scheduling, and back-end device-specific compilation. Notable aspects of the mapping software include support for high-level language (C), an estensible library of compiler optimizations, a cost-based basic block partitioner, and output to a range of back-ends targeting a variety of cores. *AppMapper* tools and methodology follow the flow shown in Figure 4.1. Individual steps include:

- **Preprocessing/conversion to SUIF** - Following parsing, high-level C constructs are translated to a unified abstract syntax tree format (AST). After property annotation, AST representations are converted to a graph-based intermediate format (SUIF) [86] that represents functions at both high and low levels. This representation allows for rapid evaluation for both partitioning and communication cost estimation and the use of standard preprocessing passes such as dead-code elimination.

- **Basic block partitioning and assignment** - An annealing-based partitioner operates on basic blocks based on core computation and communication cost models. The partitioner isolates intermediate-form structures to locate inter-core communication. The result of this phase is a refined task graph where the nodes are clustered branches of the syntax tree assigned to available aSoC cores and the inter-node arcs represent communication. The number and the type of nodes in this task graph match the number and type of cores found in the device. Following partitioning and assignment to core types, core tasks are allocated to individual cores located in the aSoC substrate so that computation load is balanced.

- **Inter-core synchronization** - Once computation is assigned to core resources, communication points are determined. The blocking points allow for synchronization of stream-based communication and predictable bandwidth.

- **Communication scheduling** - Inter-core communication streams are determined through a heuristic space-time scheduler. This list-scheduling approach minimizes the overall critical path while avoiding communication congestion. Individual *interconnect memory* binaries are generated following communication scheduling.

- **Core-based compilation** - Core compilation and communication scheduling are analyzed in tandem through the use of feedback. Core functionality is determined by native core compilation technology (e.g. FPGA synthesis, RISC compiler). Communication calls between cores are provided through fine-grained send/receive operations.

- **Code generation** - As a final step, binary code for each core and communication interface is created.

These steps are presented in greater detail in subsequent subsections:

4.1 SUIF preprocessing

The AppMapper front-end is built upon the SUIF compiler infrastructure [86]. SUIF provides a kernel of optimizations and intermediate representations for high-

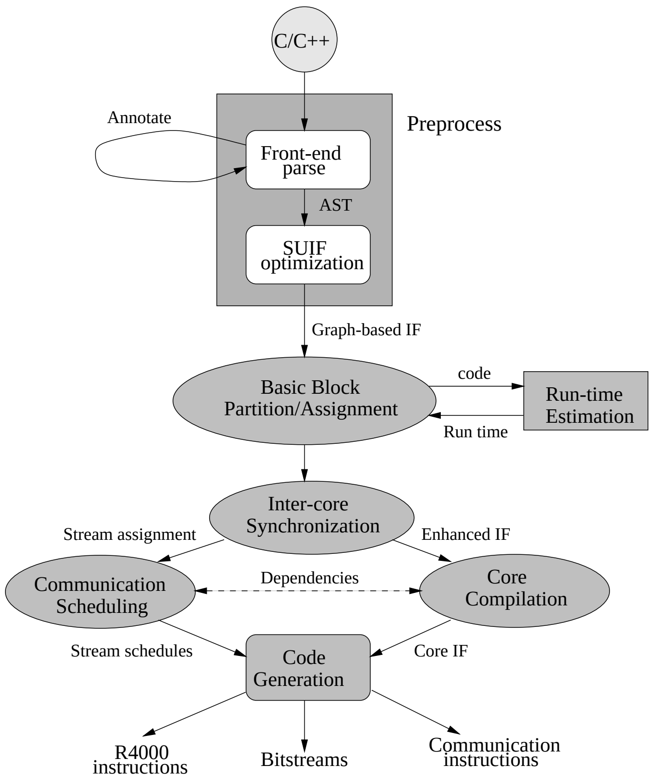


Figure 4.1. aSoC Application Mapping Flow

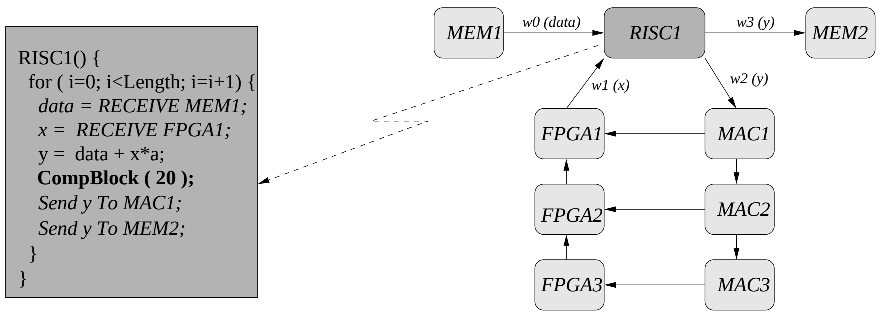
level C code structures. High-level representations are first translated into a language-independent abstract syntax tree format. This approach allows for object-oriented representation for loops, conditionals, and array accesses. AST representations are then converted into a graph-based intermediate form that represents functions at both a high and low level. Prior to partitioning, AppMapper takes advantage of several scalar SUIF optimization passes including constant propagation, forward propagation, constant folding, and scalar privatization [86]. The interprocedural representation supported in SUIF facilitates subsequent AppMapper partitioning, placement, and scheduling passes. SUIF supports interprocedural analysis rather than using procedural inlining. This representation allows for rapid evaluation of partitioning and communication cost and the invocation of dead-code elimination. Data references are tracked across procedures.

4.2 Basic Block Partitioning and Assignment

Following conversion to intermediate form, high-level code is presented as a series of basic blocks. These blocks represent sequential code, loop-level parallelism, and subroutine functions. Based on calling patterns, dataflow dependency between blocks is determined through both forward and reverse tracing of inter-block paths [10]. As a result of this dependence analysis, coarse-gained blocks can be scheduled to promote parallel computation. As shown in Figure 4.2b for an IIR filter, sub-function dependency forms a flowgraph of computation that can be scheduled. The most difficult part of determining this dependency is estimating the computation time of basic blocks across a range of cores to determine the core best suited for evaluation. The run time of each basic block is determined by parameters of the computation. These include:

- β **- run time** - execution time of a basic block on a specific core. Value β is based on the number of clock cycles, the speed of the core clock, and the amount of available parallelism.
- λ **- invocation frequency** - the number of times each basic block is invoked.

The parameters lead to an overall core run time of $\beta \times \lambda$ for each function. Core run-time estimates, β, are determined through instruction counts or through simulation, prior to compilation using techniques described in Section 5.2. Clock rates, which vary from core to core, are taken into account during this determination. A goal of design mapping is to maximize the throughput of stream computation while minimizing c_{total}, the inter-core transport time for basic block data. For a specific core, this value measures the shortest distance to another core of a different type.



```
RISC1() {
  for ( i=0; i<Length; i=i+1) {
    data = RECEIVE MEM1;
    x =  RECEIVE FPGA1;
    y =  data + x*a;
    CompBlock ( 20 );
    Send y To MAC1;
    Send y To MEM2;
  }
}
```

(a) Codes for RISC1 with communication primitives (b) Data streams of IIR application

Figure 4.2. Inter-core synchronization

Assignment of basic blocks to specific cores requires a cost model which takes both computation and communication into account. For AppMapper, this cost is represented as:

$$cost = x \times T_{compute} + y \times \frac{1}{T_{overlap}} + z \times c_{total}. \tag{4.1}$$

where $T_{compute}$ indicates combined computation time of all streams, $T_{overlap}$ indicates computational parallelism, c_{total} indicates combined stream communication time and x, y, and z are scaling constants. Minimization of this cost function forms the basis for basic block assignment. The value $T_{compute}$ is determined from β parameters for each core. Prior to basic block assignment, small code blocks are clustered using Equation 4.1 in an effort to minimize inter-core transfer. To support placement, a set of N bins are created, one per target core. During the clustering phase, communication time is estimated by the distance of the shortest path between the two types of target cores. At the end of clustering, a collection of N block-based clusters remains. Dataflow dependency is tracked through the creation of basic block data predecessor and successor lists.

For core assignment, clustered blocks are assigned to unoccupied cores so that the cost expressed in Equation 4.1 is minimized. *AppMapper* provides a file-based interface for users to manually assign basic blocks to specific cores, if desired. Following greedy basic block assignment to cores, a swapping step is used to exchange tasks across different types of cores subject to the cost function in Equation 4.1. This step attempts to minimize system cost and critical path length by load balancing parallel computation across cores. Load balancing is supported by the second term in Equation 4.1. Basic block assignment is complicated by the presence of multiple cores with the same functionality in an aSoC device. Following basic block assignment to a specific type of core, it is necessary to match the block to a specific core at a fixed location. Given the small number of each type of core available (typically less than 5), a full enumeration of all core assignments is possible. For later generation devices it may be possible to integrate this search with the basic block to core assignment phase.

4.3 Inter-core Synchronization

Synchronization between cores is required to ensure that data is consumed based on computational dependencies. Once basic blocks have been assigned to specific cores in the aSoC device, communication primitives are inserted into the intermediate form to indicate when communication should occur. These communications are blocking based on the transfer rate of the communication network. As shown in Figure 4.2a, the data transfer call to multiply-accumulate unit *MAC1* follows an assignment to y. As a result, *RISC1* processing can be overlapped with *MAC1* processing. Each inter-core communication represents a data stream, as indicated by w-labeled arcs in Figure 4.2b.

4.4 Communication Scheduling: Space-Time Routing

Following basic block assignment, the number of streams and their associated sources and destinations are known. Given a set of streams, communication scheduling assigns streams to communication links based on a fixed schedule. Inter-tile communication is broken into a series of time steps, each of which is represented by a specific instruction in a communication interface instruction memory. Schedule cycle assignment is made so that the schedule length does not exceed the instruction storage capacity of each communication interface instruction memory (32 instructions). Each unidirectional inter-tile channel can transmit one data value on each communication clock cycle. Only one stream can use a channel during a specific clock cycle. In general, the length of a schedule must be at least as long as the longest stream Manhattan path. During schedule execution, multiple source-destination data transfers may take place per stream. For example, two stream transfers take place per schedule in the example shown in Figure 3.2. To allow for flow control, all transfers for a stream must follow the same source-destination path.

Our communication scheduling algorithm forms multi-tile connections for all source-destination pairs in *space* through the creation of multi-tile routing paths. Sequencing in *time* is made by the assignment of data transfer to specific communication clock cycles. This space-time scheduling problem has been analyzed previously [64] in terms of static, but not stream-based scheduling. For our scheduler, the schedule length L is set to the longest Manhattan source-destination path in terms of tiles. Streams are ordered by required stream bandwidth per tile. The following set of operations are performed to create a source-destination path for each stream prior to scheduling transfers along the paths:

- The shortest source-destination path for each path is determined via maze routing using a per-tile cost function of $g_i = g_{i-1} + c_i$. In this equation, c_i is the cost of using a tile communication channel, g_{i-1} is the cost of the route from the path source to tile i, and g_i is the total cost of the path including tile i. The c_i cost value represents a combination of the amount of channel bandwidth required for the path in relation to the bandwidth available and the distance from the channel to the stream destination.

- For multi-fanout streams, a Steiner tree approximation is used to complete routing. After an initial destination is reached, maze routes to additional destinations are started from previously-determined connections.

Following the assignment of streams to specific paths, the assignment of stream data transfers to specific communication clock cycles is performed. Each transfer must be scheduled separately within the communication schedule. Scheduling is performed via the following algorithm:

1. Set the length of the schedule to the length of the longest Manhattan path distance, L. Specific schedule time slots range from 0 to $L-1$.
2. Order streams by required channel bandwidth.
3. For each stream:
    (a) Set start time slot s to 0.
    (b) For each transfer:
        i. Determine if inter-tile channels along source-destination path are available during n consecutive communication clock cycles, where n is the stream path length.
        ii. If bandwidth available, schedule transfer communication, increment start time s, and go to step 3.b to schedule next transfer.
        iii. Else increment start time s and go to step 3.b.i.

If any stream cannot fit into the length of the stream schedule L, the schedule length is incremented by one and the scheduling process is restarted.

In Section 3.1.3.1, a technique is described which allows run-time switching between multiple communication schedules. To support multiple schedules, the communication scheduling algorithm must be invoked multiple times, once per schedule, and the length of the *combined* schedules must fit within the communication interface instruction memory.

4.5 Core Compilation and Code Generation

Following assignment of basic blocks to cores and scheduling, basic block intermediate form code is converted to representations that can be compiled by tools for each core. Back-end formats include assembly-level code (R4000 processor) and Verilog (FPGA, multiplier). These tools also provide an interface to the simulation environment described in Section 5.2. The back-end step in AppMapper involves the generation of instructions for the R4000 and bitstreams for the FPGA. Each communication interface is configured through the generation of communication instructions.

4.6 Comparison to Previous Mapping Tools

To date, few integrated compilation environments have been created for heterogeneous systems-on-a-chip. The MESCAL system [60] provides a high-level programming interface for embedded SOCs. Though flexible, this system is based on a communication protocol stack which may not be appropriate for data stream-based communication. Several projects [62] [63] have adapted embedded system compilers to SOC environments. These compilers target bus-based interconnect rather than a point-to-point network. Cost-based tradeoffs between on-chip hardware, software, and communication were evaluated by Wan et al. [104]. In Dick and Jha [84], on-chip task partitioning was followed by a hill-climbing based task placement stage.

Our mapping system and these previous efforts have similarities to software systems which map applications to a small number of processors and custom devices (hardware/software co-design [33]), and parallel compilers which target a uniform collection of interconnected processors. Most codesign efforts [33] involve the migration of operational or basic block tasks from a single processor to custom hardware. The two primary operations performed in hardware/software codesign is the partitioning of operations and tasks between hardware and software and the scheduling of operations and communications [33]. The small number of devices involved (usually one or two processors and a small number of custom devices) allows for precise calculation of communication and timing requirements, facilitating partitioning and scheduling.

Our partitioning approach, which is based on task profiling and simulated annealing, extends earlier task-based codesign partitioning approaches [35, 37] to larger numbers of tasks and accurately models target processors and custom chips. Although all of these efforts require modeling of execution time, our approach addresses a larger number of target models and requires tradeoffs between numerous hardware/software partitions. This requires high-level modeling of both performance and partition size for a variety of different cores. Our approach to partition assignment of basic block tasks is slightly more complicated than typical codesign assignment. In general, the bus structure employed by codesign systems [35] limits the need for cost-based assignments. In contrast, our swap-based assignment approach for heterogeneous targets is simpler than the annealing based technique used to assign basic blocks to a large homogeneous array of processors [103]. Since blocks are assigned to specific target cores during partitioning, the assignment search is significantly more constrained and can be simplified.

Our stream-based scheduling differs from previous codesign processor/custom hardware communication scheduling [33]. These scheduling techniques attempt to identify exact communication latency between processors and custom devices to ensure worst-case performance across a variety of bus transfer modes (e.g. burst/non-burst) [62]. Often instruction scheduling is overlapped with communication scheduling to validate timing assumptions [33]. In contrast, our communication scheduling approach ensures stream throughput over a period of computation. Unlike parallel processing stream compilers for homogeneous multiprocessors [9, 45], the exact communication time of each piece of data is not required. Since heterogeneous devices with different clock speeds are used for our system, the scheduled reservation of bandwidth is sufficient to ensure required throughput.

Chapter 5

Experimental Methodology

5.1 Target aSoC Devices

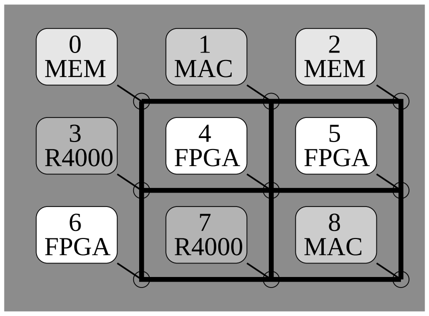
To validate the aSoC approach, target applications have been mapped to implemented aSoC devices and architectural models containing up to 49 cores. Parameters associated with the models are justified via a prototype aSoC device layout, described in Chapter 10. Examples of 9 and 16 core models are shown in Figure 5.1. The models consist of R4000 RISC microprocessors [27], FPGA blocks, 32Kx8 SRAM blocks (MEM), and multiply-accumulate (MAC) cores. The same core configurations were used for all benchmarks. The FPGA core contains 121 logic clusters, each of which consists of four 4-input look-up tables (LUTs) and flip flops [18]. The core population of all aSoC configurations are shown in Table 5.1.

5.2 Simulation Environment

To compare aSoC to a broad range of alternative on-chip interconnect approaches, including flat and hierarchical buses, a timing-accurate interconnect simulator was developed. This simulator is integrated with several IP core simulators to provide

Core Array Configuration	R4000	FPGA	Mem	MAC
3x3	2	3	2	2
4x4	4	4	2	6
5x5	9	2	4	10
6x6	13	2	6	15
7x7	18	3	8	20

Table 5.1. aSoC device configurations



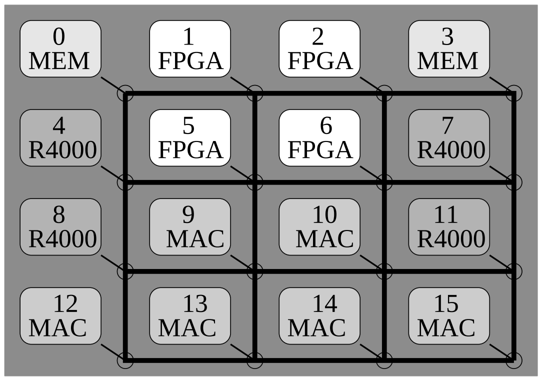


Figure 5.1. aSoC topologies: 9 and 16 cores

a complete simulation environment. The interaction between the computation and communication simulators provides a timing-accurate aSoC model that can be used to verify a spectrum of SoC architectures.

A flowchart of the simulator structure appears in Figure 5.2. For our modeling environment, simulation takes place in two phases. In phase 1, simulation determines the exact number of core clock cycles between data exchanges with the communication network coreport interface. In phase 2, core computation time is determined between send and receive operations via core simulation which takes core cycle time into account. Data communication time is simultaneously calculated based on data availability and network congestion. Both computation and communication times are subsequently combined to determine overall run-time.

During the first simulation phase, core computation is represented by C files created by AppMapper or user-created library files in C or HDL (Verilog). This compute information is used to determine the transfer times of core-network interaction. The execution times of core basic blocks are determined by invocation of individual core simulators. Cycle count results of these simulators are scaled based on the operating frequency of the cores. Specific simulators include:

- **Simplescalar** - This processor simulator [21] models instruction level execu-

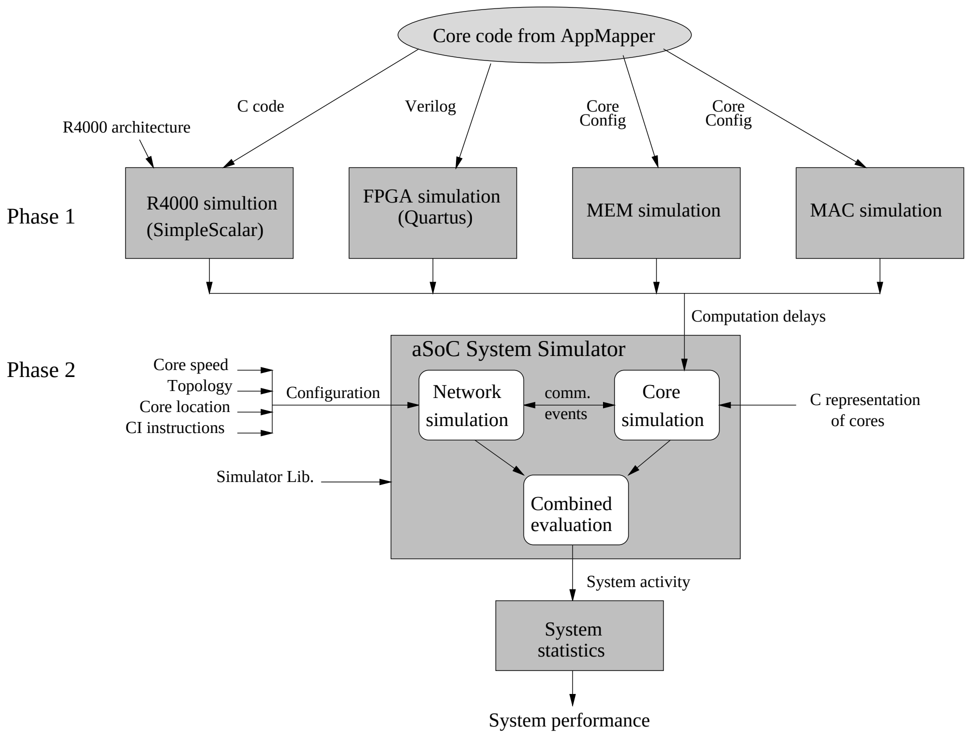


Figure 5.2. aSoC System Simulator

tion for the R4000 MIPs architecture. The simulator takes C code as input and determines the access order and number of execution cycles between coreport accesses. Cycle counts are measured through the use of breakpoint status information.

- **FPGA block simulation** - Unlike other cores, FPGA core logic is first created by the designer at the register-transfer level. The Verilog-XL simulator is then used to determine cycle counts between coreport transfers. To verify timing accuracy, all cores have been synthesized to four-input LUTs and flip flops using Synplicity Synplify.

- **Multiply-accumulate** - The multiply-accumulate core is modeled using a C-based simulator. Given the frequency of an input stream, the simulator determines the number of cycles between coreport interactions.

	Speed	Area (λ^2)	CI Overhead
Comm. interface	2.5 ns	2500×3500	-
MIPs R4000 (w/o cache)	5 ns	4.3×10^7 [27]	16.9%
MAC	5 ns	1500×1000	48.6%
FPGA	10 ns	27500×26500	1.0%
MEM	5 ns	7500×6500	8.8%

Table 5.2. Component Parameters

- **SRAM memory cores (MEM)** - SRAM cores are modeled using a C-based cycle-accurate simulator.

Layouts, described in Chapter 10, were used to determine per-cycle performance parameters of the FPGA, multiply-accumulate, and memory cores.

The second stage of the simulator determines communication delay based on core compute times and instruction memory instructions. Following core timing determination, aSoC network communication ordering and delay is evaluated via the communication simulator. The instruction memory instructions are used to perform simulation of each tile's communication interface. This part of the simulator takes in multiple interconnect memory instruction files. High-level C code represents core and communication interfaces. As shown in Figure 4.2a, core compute delay is replaced with compute cycle (CompBlock) delays determined from the first simulation stage. The second input file to the simulator is a configuration file, previously generated by AppMapper. This file contains core location and speed information, the details of the inter-core topology and the interconnection memory instructions for each communication interface. These files are linked with a simulator library to generate an executable. When the simulator is run, multiple core and communication interface processes are invoked in an event-driven fashion based on data movement, production, and consumption. For aSoC, the CDM and coreport storage is modeled to allow for accurate modeling of inter-tile storage.

The simulator can model a variety of communications architectures based on the input parameter file. This includes the aSoC interconnect architecture, the CoreConnect on-chip bus [54], a hierarchical CoreConnect bus, and a dynamic router. Parameters associated with aSoC, such as the core type, location, speed, and the communication interface configuration, can be configured by the designer to explore aSoC performance on applications.

Chapter 6

Target aSoC Applications

Four applications from communications, multimedia, and image processing domains have been mapped to the aSOC device models using the *AppMapper* flow described in Chapter 5. Mapped applications include MPEG-2 encoding [43], orthogonal frequency division multiplexing (OFDM) [61], Doppler radar signal analysis [79], and image smoothing (IMG). An IIR filter kernel was used for initial analysis.

6.1 MPEG-2 Encoder

An MPEG-2 encoder was parallelized from sequential code [43] to take advantage of concurrent processing available in aSOC. Three 128×128 pixel frames, distributed with the benchmark, were used for aSOC evaluation. For the 4×4 aSOC configuration, MPEG-2 computation is partitioned as shown in Figure 6.1. Thick arrows indicate video data flow and thin arrows illustrate control signal flow. Frame data blocks (16×16 pixels in size) in the Input Buffer core are compared against similarly-sized data blocks stored in the Reference Buffer and streamed into a series of multiply-accumulate cores via an R4000. These cores perform motion estimation by determining the accumulated difference across source and reconstructed block pixels, which is encoded by the DCT quantizer, implemented in an adjacent R4000. The data is then sent to the controller and Huffman coding is performed in preparation for transfer via a communication channel. Another copy of the DCT encoded data is transferred to the IDCT circuit, implemented in an R4000 core. The reconstructed data from the IDCT core is then stored in the Reference

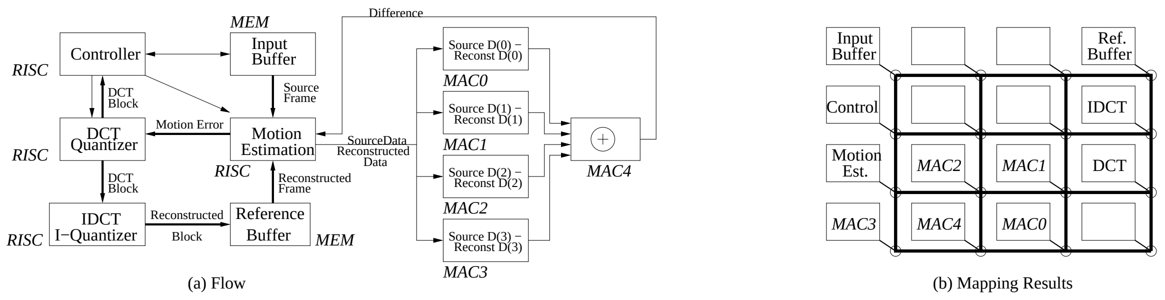


Figure 6.1. Partitioning of MPEG-2 encoder to a 4×4 aSOC configuration

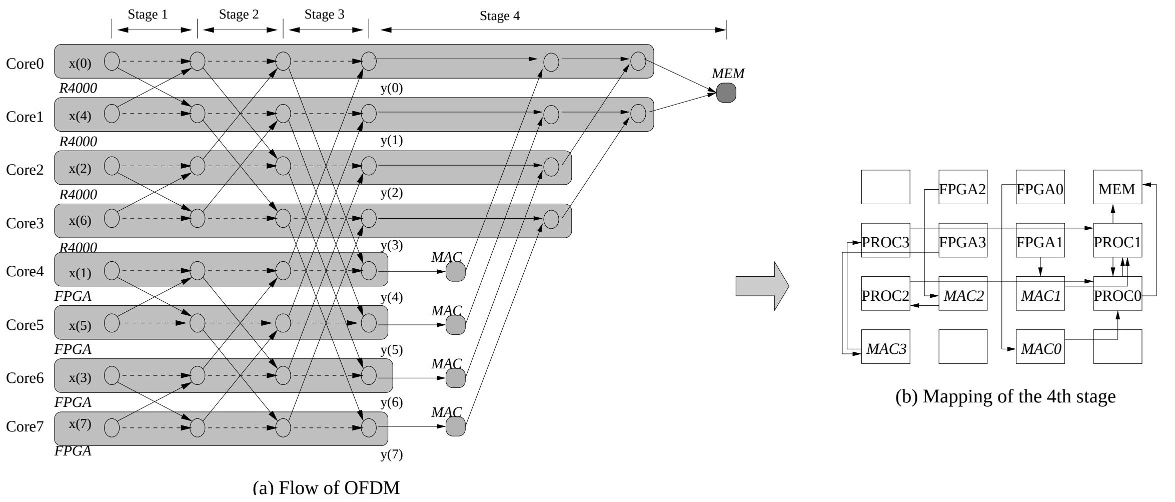
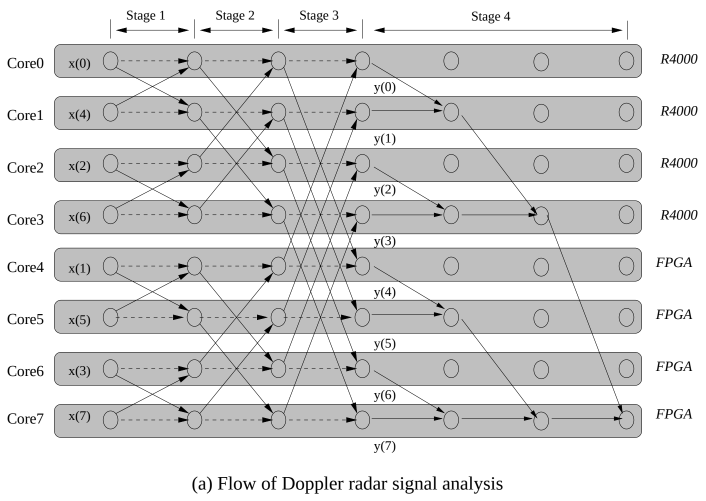


Figure 6.2. OFDM mapped to 16 core aSOC model

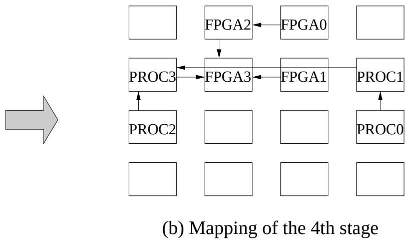
Buffer core for later use. Data transfer is scheduled so that all computation and storage is pipelined.

6.2 Orthogonal Frequency Division Multiplexing

OFDM is a wireless communication protocol that transmits data over a set of narrowband channels [61]. OFDM provides high communication bandwidth and is resilient to RF interference. Multi-frequency transmission using OFDM requires multiple processing stages including IFFT, normalization, and noise-tolerance guard value insertion. As shown in Figure 6.2, a 2048 subcarrier complex-valued OFDM transmitter has been implemented on an aSOC model. The IFFT portion of the computation is performed using four R4000 and four FPGA cores. Resulting com-



(a) Flow of Doppler radar signal analysis



(b) Mapping of the 4th stage

Figure 6.3. Doppler radar signal analysis mapped to 16 core aSOC model

plex values are normalized with four MAC and four R4000 cores. A total of 512 guard values are determined by R4000 cores and stored along with normalized data in memory. The OFDM application exhibits communication patterns which change during application execution, as shown in the four stages of computation illustrated in Figure 6.2. The run-time branching mechanism of the communication interface is used to coordinate communication branching for the four stages.

6.3 Doppler Radar Signal Analysis

A stream-based Doppler radar receiver [79] was implemented and tested using an aSOC device. In a typical Doppler radar system, a sinusoidal signal is transmitted by an antenna, reflects off a target object, and returns to the antenna. As a result of the reflection, the received signal exhibits a complex frequency shift. This shift can be used to determine the speed and distance of the target through the use of a Fourier analysis unit. As shown in Figure 6.3, the main components of the analysis include an FFT of complex input values, a magnitude calculation of FFT results and the selection of the largest frequency magnitude value. For the 16 core aSOC model, a 1024 point FFT, magnitude calculation, and frequency selection were performed by four R4000 and four FPGA cores. All calculation was performed on 64 bit complex values. Like OFDM, the Doppler receiver requires communication patterns which change during application execution. The run-time branching mechanism of the communication interface is used to coordinate communication branching for the four stages.

6.4 Image Smoothing

A linear smoothing filter was implemented in multi-core aSOC devices for images of size 800×600 pixels. The linear filter is applied to the image pixel matrix in a row-by-row fashion. The scalar value of each pixel is replaced by the average of the current value and its neighbors, resulting in local smoothing of the image and reducing the effects of noise. To take advantage of parallelism, each image is partitioned into horizontal slices and processed in separate pipelines. Data streams are sent from memory (MEM) cores to multiple R4000s, each accepting a single data stream. Inside each MAC, each pixel value is averaged with its eight neighbor values resulting in nine intermediate values. Later in the stream, an FPGA-based circuit sums the values to generate averaged results. These results are buffered in a memory core. This application was mapped to aSOC models ranging in size from 9 to 49 cores by varying the number of slices processed in parallel.

6.5 IIR Filter

A three-stage and a six-stage IIR filter were implemented using the 9 and 16 aSOC models, respectively. The data distribution and collection stages of the filter are implemented using R4000s. MACs and FPGA cores execute middle stages of multiplication and accumulation. SRAM cores (MEM) buffer both source data and computed results. The overall application data rate is limited by aSOC communication speed.

Chapter 7

Adaptive SOVA Turbo Code Decoder Application on aSoC

In the previous chapter, several DSP and multi-media applications have been mapped onto aSoC. While these applications are small and mostly the computation kernels of systems, it is necessary to further test the practicability of aSoC with large and practical systems.

Fully testing aSoC requires the mapping of an integral application system onto aSoC. ASoC targets a heterogeneous System-on-Chip for next generation techniques. A partial kernel core may not be able to fully reveal the ability of aSoC. Therefore, the target application should be a full system with the latest techniques.

To fulfill these requirements and better illustrate the performance of aSoC, an application system, the turbo codes simulator, is chosen from the wireless communication domain.

The turbo codes simulator includes the turbo encoder, channel simulator, turbo decoder and system evaluator. As a wireless communication system, it has a high data rate and its decoding throughput is an important parameter to be tested in aSoC.

In this dissertation, an adaptive soft-output Viterbi algorithm (ASOVA) was developed for the turbo decoder, which reduced the computation of the traditional SOVA [48], and could change its decoding complexity based on the input signal to noise ratio (SNR).

This turbo codes simulation system is mapped onto aSoC, FPGAs and other on-chip architectures. The performances on these architectures will be compared to each other to reveal the advantages of aSoC.

7.1 Introduction

The recently introduced turbo code [15], an error-correction code, is attractive because of its excellent performance. The turbo encoder is formed by two recursive systematic codes (RSC) [72] joined by an interleaver. The proposed decoder applies an iterative algorithm whose core is a maximum a posteriori (MAP) [11] decoder. Since the MAP decoding algorithm is complicated and difficult to pipeline, efforts are made to reduce the decoding complexity, thereby reducing the power dissipation and improving decoding speed.

Extended from MAP, a sub-optimum decoding algorithm, the Max-Log-MAP algorithm [85], reduces the complexity by sacrificing some performance. Another sub-optimum decoding algorithm, the soft-output Viterbi algorithm (SOVA) [48] has been shown to be a very efficient algorithm for turbo decoding. It has been shown that SOVA is about half as complex as the Max-Log-MAP algorithm, and the decoding ability of SOVA is close to that of MAP and Max-Log-MAP. The input signal to noise ratio (SNR) for SOVA to achieve a bit error rate (BER) of 10^{-4} is about 0.7dB more than the SNR for MAP [85]. The error correction ability of Max-Log-MAP is roughly between that of MAP and SOVA. The performance of SOVA can be further improved by taking the secondary errors into account [40], or scaling the outputs [80]. Furthermore, SOVA can be designed using a pipelined architecture, which improves the decoding throughput and reduces the memory requirement.

Some improvements are done in a higher level than the decoding algorithms. Since turbo decoder employs iterative decoding, the number of iterations plays an important role in the decoding speed and BER performance. Recent work has investigated exploiting the effects of the iteration termination schemes [65, 91]. The similar trade-off between decoding speed and error-correcting ability can also be achieved by varying the encoder and interleaver length [100, 101]. A soft-output adaptive Viterbi algorithm (SAVA) [22], which applies the scheme of adaptive Viterbi Algorithm [23] in the SOVA decoder, reduces the complexity by sacrificing certain Bit Error Rate (BER) performance.

The turbo decoder has been built in VLSI [14, 51], and has also been mapped onto FPGAs and reconfigurable resources for flexibility [49]. These implementations achieve some trade-offs by adapting the code parameters. However, the decoding complexity of the code is still high. Furthermore, while the VLSI implementation achieves high data rate with a fixed pipelined architecture and the FPGA is flexible to make use of adaptive algorithms, it is difficult for FPGA or VLSI substructure to support both the high data rate and reconfigurability concurrently, which restricts the code's performance.

In this dissertation, an Adaptive SOVA (ASOVA) is created to further reduce the decoding complexity in addition to allowing full range adaptation. The ASOVA will be mapped onto aSoC, which provides a very good platform for high bandwidth data transmission and adaptability.

The decoder complexity of ASOVA is reduced compared with that of SOVA. While SOVA traces all the possible survivor paths in decoding, ASOVA keeps only part of the survivor paths by introducing a threshold value and limiting the number of survivor paths. Pruning the survivor paths results in inaccurate or lost outputs of the ASOVA decoder. To compensate for this loss, the outputs apply a scaling factor and the lost data will be estimated using its expectation. Software simulation shows that the BER performance of ASOVA is better than Max-Log-MAP [36] and very close to Log-MAP algorithm [111], with about 1/4 of the computational complexity of SOVA.

To evaluate the adaptability and trade-off between performance and power, the ASOVA will be mapped onto an FPGA first. Given a BER requirement, the decoding speed can be reduced to trade for decreased power dissipation or smaller

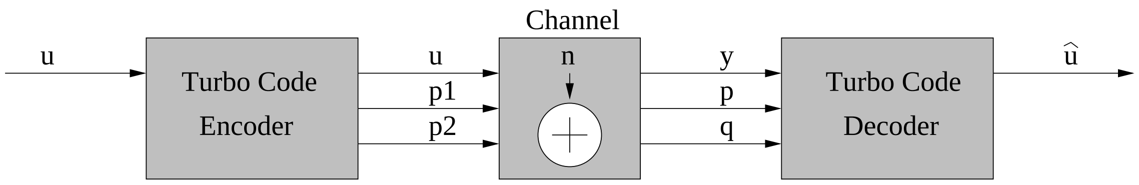


Figure 7.1. Turbo Code System

area. When the input SNR is high, the threshold and N_{max} can be adapted to reduce the complexity and speed up the decoding. Or, while maintaining a certain fixed decoding speed, some hardware resources can be freed for other usage or shut down to reduce power dissipation.

Finally, the ASOVA turbo codes system will be mapped onto an aSoC example chip to test its decoding speed. As mentioned previously in Chapter 3, aSoC consists of a high-bandwidth, adaptive communication network. When partitioned to separate IP cores in aSoC, the turbo codes system can be fully parallelized. Since the IP cores are connected with the high speed network, high decoding speed can be achieved.

In addition, the adaptive nature of aSoC will be able to ease the reconfiguration of ASOVA. The IP cores of aSoC are reconfigurable, which allows the aSoC implementation to perform the same adaptation as the FPGA implementation. While working in a SNR-changing channel, the adaptation allows the ASOVA to switch to a simpler scheme quickly. As a result, a high decoding rate will be guaranteed.

To test the performance with a high data rate, the aSoC implementation of ASOVA will be compared with systems using other communication architectures: bus, hierarchical bus and dynamic networks.

7.2 Turbo Codes and Decoding Algorithms

7.2.1 Turbo Codes

Turbo codes [15] are error-correction codes that are attractive for their superior error-correcting ability. By adding redundant information in the form of parity bits, turbo codes allow the receiver to correct some of the errors when the data signals are influenced by channel noise. Figure 7.1 presents a turbo codes system. Fundamentally, given an input sequence **u**, the turbo encoder generates two sequences of parity bits, **p1** and **p2**, where **p1** is based on **u** and **p2** is based on the sequence **u'**, which is a permutation of **u**. The sequences **u, p1** and **p2** become **y, p** and **q** when they are received by the decoder through the communication channel. In decoding, two component decoders are applied to decode the **u** or **u'**. These two component decoders interact for several times to improve the decoding ability. The parameters employed and the algorithm are explained as follows. For clarity, the variables in bold represent data vectors in this document.

- Input data sequence $\mathbf{u} = \{u_k, k = 1, ..., B\}$.
- **u'**: the sequence which is randomly permuted from **u**.
- Block Length B. It is the length of the input sequence. In the encoder, the B-bit **u** is permuted to generate the B-bit **u'**. Since the permutation unit is named Interleaver, the Block Length is also called the *Interleaver Length.*
- The parity bit sequences in Turbo Encoder, **p1** and **p2**. The sequence **p1** is generated from **u** and the sequence **p2** is generated from **u'**. Both **p1** and **p2** are B-bit sequences.
- Channel noise n. An additive white Gaussian noise (AWGN) channel is assumed. Let random variable $n \sim N(0, \sigma^2)$ represents the channel noise, where $N(0, \sigma^2)$ is a zero-mean Gaussian random number with a variance of $\sigma^2 = N_0/2$.
- Received sequences $\mathbf{y} = \{y_k\}, \mathbf{p} = \{p_k\}$ and $\mathbf{q} = \{q_k\}, k \in [1, B]$. In the turbo encoder, three B-bit sequences, **u, p1** and **p2** are sent to the channel.

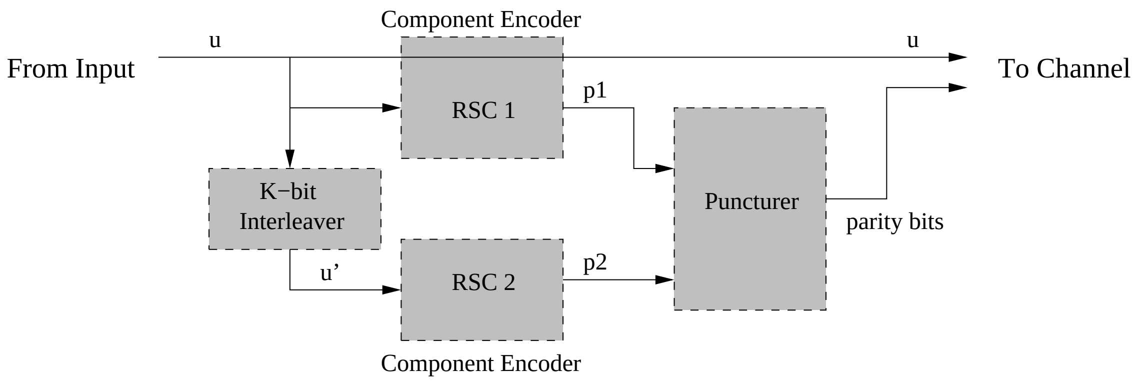


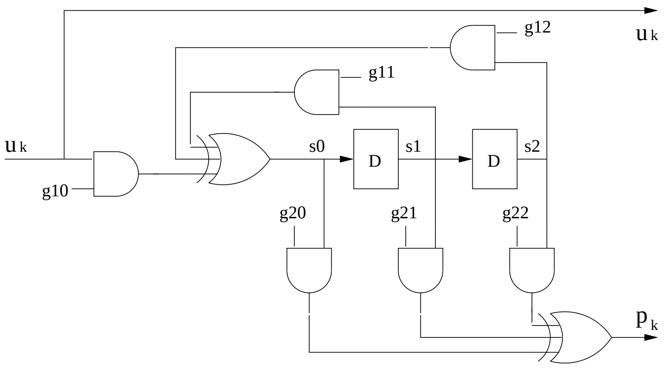
Figure 7.2. Turbo Code Encoder

After the channel noise is applied, the turbo decoder receives three sequences respectively from the communication channel: **y, p** and **q**.

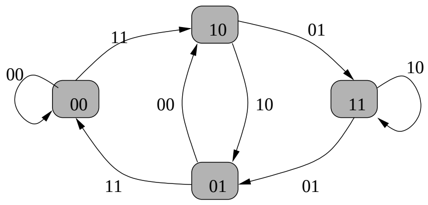
7.2.1.1 Turbo Code Encoder

A typical turbo codes architecture can be found in [87]. The turbo encoder, as described in Figure 7.2, consists of two Recursive Systematic Convolutional (RSC) [72] encoders that are connected by an interleaver. The RSC encoders, D1 and D2, are termed the *component encoders* of turbo codes. The interleaver permutes the input data sequence **u** to generate another data sequence **u'**. Taking the **u** and **u'** as inputs, the two component encoders generate the parity bits **p1** and **p2** respectively. The *code rate* is defined as the ratio of the number of information bits and the number of bits sent to the channel, which is the total length of **u, p1** and **p2** in turbo codes. In some systems, a puncturer is used to increase the code rate by skipping some of the parity bits in **p1** and **p2**. Without the puncturer, the code rate of turbo codes shown as Figure 7.2 is 1/3.

Figure 7.3 presents a typical RSC encoder [72] and its state diagram. The RSC encoder can be regarded as a state machine, which starts from state 0, takes the input bit u_k, and outputs the parity bit p_k together with the original input bit u_k at each step. Figure 7.3 shows a RSC encoder with memory M=2 bits, s_1 and s_2,



(a) RSC Encoder Arcitecture



(b) State Diagram of (g1,g2)=(7,5)

Figure 7.3. RSC Architecture and State Diagram

which are a series of shift registers. The *constraint length* of the convolutional code is defined as $(M+1)=3$, which is the number of output bits affected by a given input bit.

The *generator* of RSC code is defined as $G=(g_1,g_2)$ or $(1,g_1/g_2)$, where $g_1=\{g_{10},g_{11},...,g_{1M}\}$ and $g_2=\{g_{20},g_{21},...,g_{2M}\}$. The generator defines the functionality of the RSC encoder. The input to the first register, s_0, is given by Eq. 7.1 in which the binary coefficient g_{1i} decides if the state s_i is taken into account. The $\oplus$ represents XOR operation. Similarly, the output p_k is given by Eq. 7.2.

$$s_0=(g_{10}\&u_k)\oplus_{i\in[1,M]}(g_{1i}\&s_i) \tag{7.1}$$

$$p_k=\oplus_{i\in[0,M]}(g_{2i}\&s_i) \tag{7.2}$$

Generally, (g1,g2) is *expressed as an octal number.* In Figure 7.3, if $g_{11}=0$ when the others are 1, it forms a RSC code of $(g_1,g_2)=(7,5)$. The state diagram is shown in Figure 7.3(b), where the blocks represent the states with the number of (s_1s_2) inside, and the edges represent the stage transmission with the corresponding (u_kp_k).

Generally, in a code, the larger the minimum Hamming distance of any two codewords is, the better the code will be. Some (g_1,g_2) can generate good codes, which have strong error correction ability. The problem of constructing good codes is addressed in [68]. In this dissertation, the experiments will apply the codes which have been shown to have good performance: $(7,5), (15,13), (31,27)$, and $(65,57)$ [68, 107, 48].

7.2.1.2 Turbo Code Decoder

The architecture of the turbo decoder is shown in Figure 7.4. After data is transmitted through the channel, the three received sequences **y, p** and **q** correspond to the data sequences **u, p1** and **p2** from the encoder. In a system using a puncturer, a de-puncturer is employed to recover the sequence by putting the received bits into the proper positions of **y, p** and **q**. Two identical component decoders, *D1* and *D2*, are used, which correspond to the *RSC1* and *RSC2* in the encoder. *D1* takes **y** and **p** as input, and *D2* takes **y'** and **q**, where **y'** is interleaved from **y**. Their output sequences, $\mathbf{L_1}$ and $\mathbf{L_2}$, are *soft outputs*, whose signs are the decoded sequences of **u** and **u'** in Figure 7.2, and absolute values represent the reliability of their decoding decisions. The soft output from one decoder is fed into the other as the a-priori information through an Interleaver/De-Interleaver to help the latter decoder make a better decoding decision. The interleaver is the same as that in the encoder of Figure 7.2, and the de-interleavers are used to convert a sequence permuted by the Interleaver back to the original sequence. After a given number of decoding iterations, the final decision is made in the *Decide* block by combining the outputs from both decoders.

7.2.2 Turbo Code Decoding Algorithms

The MAP [11] algorithm was the first algorithm used in the component decoder of turbo codes. It is optimum with respect to bit error probability and it computes a cost value for each bit. However, it is too complicated for most practical uses. Max-

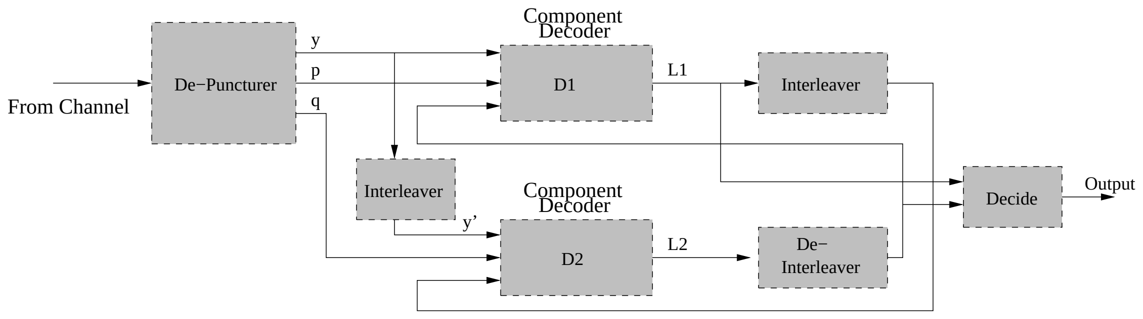


Figure 7.4. Iterative Decoder

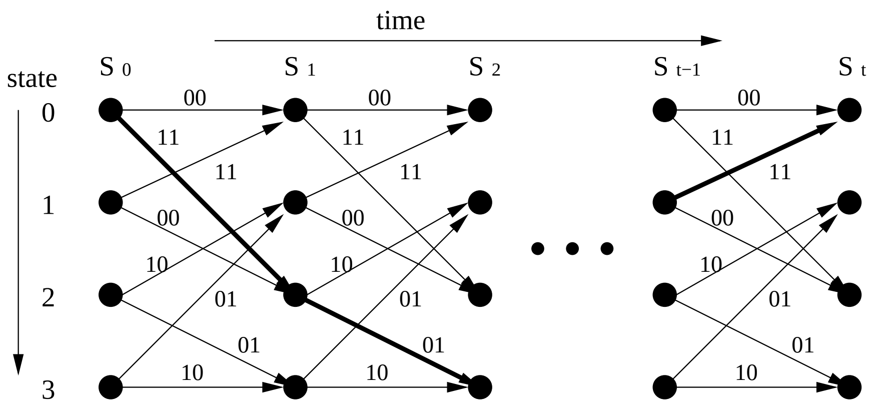


Figure 7.5. Trellis Diagram

Log-MAP [36] and soft-output Viterbi algorithm (SOVA) [48] significantly reduce the complexity, but they are sub-optimum in terms of Bit Error Rate (BER).

We focus on SOVA because it has half the complexity of Max-Log-MAP [85] and maintains competitive performance. A brief review of the SOVA decoding architecture is given in the next section. A more detailed explanation can be found in [48].

7.2.2.1 Viterbi Algorithm

Since the Viterbi Algorithm (VA) [102] is the kernel of SOVA, it will be described first. In the VA, the decoder takes the received sequences from the channel, and generates the decoded bit sequence for convolutional codes. For example, in Figure 7.4, if *D1* is a VA decoder, the sequence **y, p** would be its inputs.

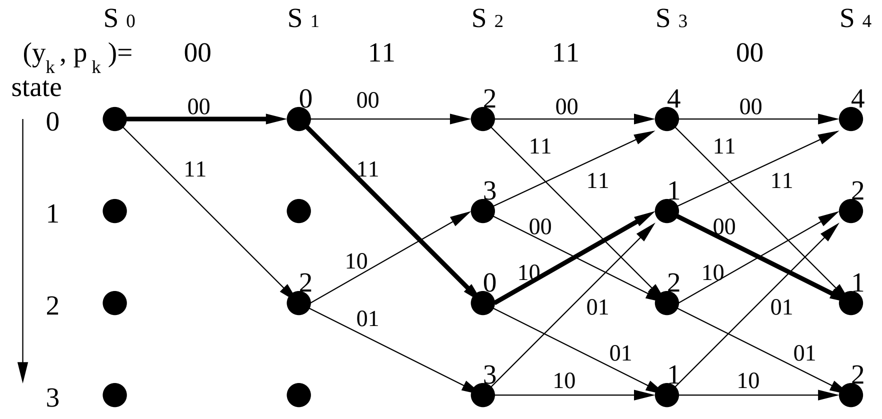


Figure 7.6. Hard-decision Viterbi Algorithm

To better understand the VA decoding, a *trellis diagram* is constructed. Taking the RSC encoder in Figure 7.3 as the example, when the state diagram of Figure 7.3(b) is evaluated successively over time, it leads to a *trellis diagram* as shown in Figure 7.5. The four states of $00, 01, 10$ and 11 in Figure 7.3(b) are mapped onto the vertical nodes of $0, 1, 2$ and 3 in Figure 7.5, and the nodes in a horizontal line represents the state at different stages over time. S_0 represents the states at stage 0, S_1 represents the states at stage 1, and so on. The edges and associated numbers in Figure 7.5 have the same meaning as in Figure 7.3(b).

To distinguish them from the variables (u_k, p_k) in the encoder, the source bit and parity bit on each edges of the decoder are represented using variables with a hat, $(\hat{u}_k, \hat{p}_k)$.

In the trellis diagram, a *path* is defined by a series of connected edges which represent stages transitions over time. For example, a path i, $\{S_0 = 0, S_1 = 2, S_2 = 3, ..., S_{t-1} = 1, S_t = 0\}$, is shown using dark lines in Figure 7.5. The bit sequence of $\{\hat{u}^i_k, k = 0, 1, ..., t\}$ associated with the sequence of edges in path i is the *decoded bit sequence* of this path.

Given an input sequence $\mathbf{u}$, the RSC encoder has a corresponding path in the trellis diagram. The VA decoder tries to reconstruct this path, from which a decoded sequence $\hat{\mathbf{u}}$ can be obtained, by evaluating the received sequences $\mathbf{y}$ and $\mathbf{p}$.

For ease of exposition, the *hard-decision decoding* [58] VA is introduced, in which the data $\{\mathbf{y}, \mathbf{p}\}$ received from the channel have been converted to binary numbers before the decoder. As shown in Figure 7.6, the received sequence $(\mathbf{y}, \mathbf{p}) = \{00, 11, 11, 00\}$ are shown on top of each trellis stage.

To reconstruct the input path, an accumulated cost value named *path metric* is determined at each stage to measure the distance between the possible paths and the received sequence. At a given stage k, the $(\hat{u}_k, \hat{p}_k)$ of each edge is compared with the received (y_k, p_k) in Hamming distance to obtain the *branch metric*. The path metric, which is shown on top of each node in Figure 7.6, is the accumulated branch metrics of the edges on this path. In Figure 7.6, at stage S_0, the received $(y_0, p_0) = 00$, and the edge from $S_0 = 0$ to $S_1 = 2$ has $(\hat{u}_0, \hat{p}_0) = 11$. The branch metric for this edge is 2, which is shown on the node $S_1 = 2$. Given a path $\{S_0 = 0, S_1 = 2, S_2 = 3\}$, the path metric is the number on node $S_1 = 2$, which is 2, plus the branch metric of edge $\{S_1 = 2, S_2 = 3\}$, which is 1. The result path metric 3 is shown in the node $S_2 = 3$. A formula is shown as Eq. 7.3 for the path metric, where the $M(S_k)$ is the path metric at node S_k.

$$M(S_k) = M(S_{k-1}) + HammingDistance(\{y_{k-1}, p_{k-1}\}, \{\hat{y}_{k-1}, \hat{p}_{k-1}\}) \qquad (7.3)$$

When multiple paths converge into the same node, the path with the minimum path metric is the *survivor* and its path metric is marked on the node. The other paths is the *competitive paths* and their path metrics are discarded. It is easy to see that the possible later stages of the survivor and discarded path are the same after they merge. If a discarded path has a good metric after a few stages, the survivor along these stages must have a better metric; thus, it is impossible for a discarded path to become the best path later. Since the VA looks for only the path with the lowest metric, discarding the un-optimal paths will not harm its performance.

After a series of stages, where the number of stages is the *truncation length* of VA, the path with the lowest path metric is named as the Maximum Likelihood (ML) path, and its associated bit sequence $\hat{u}_k^{ML}$ is the decoded output. For example, the ML path in Figure 7.6 is highlighted in bold.

7.2.2.2 Soft-Output Viterbi Algorithm

The SOVA uses the same idea as the VA except that it generates *soft outputs*, which represent the reliability of the bit decisions. For accurate computation, the input data of SOVA, **y, p**, is not hard-decoded. That is, each number of **y, p** is a continuous value. When used in turbo codes, the SOVA has another input $L(u_k)$ which is termed the *a-priori information*, and, in this turbo code model, it is the output from the other component decoder through a Interleaver or De-Interleaver. While VA uses Eq. 7.3, The path metric of SOVA is calculated using Eq. 7.4 [48]. This is a recursive transfer formula using the y_k and p_k which are received from the channel at stage k and the pair $(\hat{u}_k, \hat{p}_k)$ associated with this edge.

$$M(s_k) = M(s_{k-1}) + \frac{1}{2}\hat{u}_k L(u_k) + \frac{L_c}{2}(y_k\hat{u}_k + p_k\hat{p}_k) \tag{7.4}$$

where the L_c is the channel reliability measure and is given by Eq. 7.5, and $L(u_k)$ is the a-priori information.

$$L_c = 2/\sigma^2 \tag{7.5}$$

In SOVA, the path metric represents the likelihood that a path is the decoded path, and a larger metric implies an increased likelihood. As a result, when multiple paths converge, the highest metric path is preserved as the survivor, and other paths are discarded. Similarly, the ML path is defined as the path with the highest metric value.

Identifying the ML path, the decoded bit sequence $\hat{\mathbf{u}}$ is obtained by tracing back the edges along the ML path for the bit $\hat{u}_k^{ML}$ associated with the edges.

In addition to obtaining the decoded bit sequence, the SOVA needs to find the *soft output*. Without losing generality, we assume that the path represented by dark

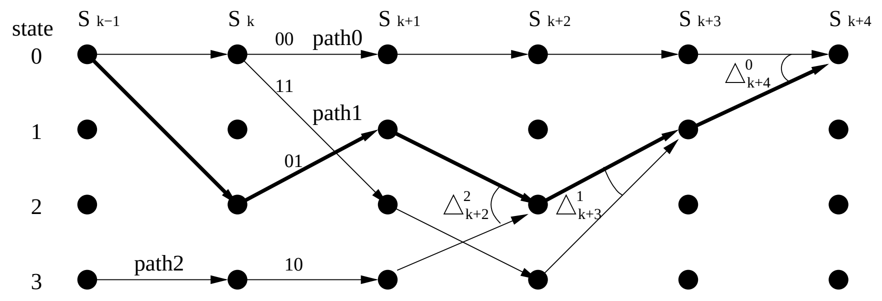


Figure 7.7. Trellis Diagram of SOVA

edges in Figure 7.7 is the ML path. The bit sequence $\hat{\mathbf{u}}^{ML} = \{\hat{u}_k^{ML}, k = 1, B\}$ is the decoded sequence for this ML paths. Now we need to find the reliability of the bit decisions for $\hat{u}_k^{ML}$.

Observations of the Viterbi algorithm have shown that all survivor paths at stage l would have come from the same path at some point before l. This point is generally within δ stages before l, where δ is the *truncation length* of the Viterbi Algorithm, which is usually set to five times the constraint length of the convolutional code [72].

The bit decision $\hat{u}_k$ associated with the edge from S_k to S_{k+1} may be different, if, instead of the ML path, the Viterbi algorithm had selected a path i in those paths that would merge with ML path up to δ stages later. Taking the path i will result in a wrong $\hat{u}_k$ only when the decoded bit of path i at stage k, which is $\hat{u}_k^i$, is different from the original $\hat{u}_k$. Based on this discussion, the reliability of $\hat{u}_k$ depends on the paths merging into the ML path from stage $t = k$ to $t = k + \delta$. For the ease of exposition, a set of paths P is defined such that any path i belonging to set P must merge with the ML path and $\hat{u}_k^i \neq \hat{u}_k$.

It has been shown in [46] that this soft-output of SOVA can be approximated by Eq. 7.6.

$$L_o(u_k) \approx \hat{u}_k \cdot \min_{t=k,\dots k+\delta, i\in P} \Delta_t^i \tag{7.6}$$

where Δ_t^i is the metric difference between the ML path with the path i, which is the path merging with ML path at stage t, and the $\hat{u}_k^i$ is the decoded bit of path i

at stage k. The minimization of Eq. 7.6 is over the paths in P only.

In the example shown in Figure 7.7, the difference between path 0 and the ML path, which is Δ^0_{k+4}, is not taken into account for the output at stage k, where path 0 has the same decoded bit as the ML path, i.e., $\hat{u}_k = \hat{u}^0_k = 0$. As the result, the soft-output of stage k is given by Eq. 7.7.

$$L_o(u_1) \approx \hat{u}_k \cdot \min\{\Delta^2_{k+2}, \Delta^1_{k+3}\} \tag{7.7}$$

7.2.3 Improvements on SOVA

The performance of SOVA has been improved since its publication. Robertson has pointed out the difference between SOVA and Max-Log-MAP in [85]. At a given stage k, while the Max-Log-MAP keeps trace of the ML path and its closest competitor, the SOVA can find the ML path but may not guarantee the closest competitor survived, where the *closest competitor* is the highest-metric path j whose decoded bit at stage k, $\hat{u}^j_k$, is different than the decoded bit of ML path $\hat{u}_k$.

It has been shown that a modified SOVA [40] can be equivalent with the Max-Log-MAP algorithm. While SOVA considers only those paths merging with the ML paths, which is termed the *first-level* paths for convenience, the modified SOVA takes the *second-level* paths into account, where the *second-level* paths are those paths that merge with the first-level paths. The first-level paths can be categorized into two path sets their its decoded bits at the evaluated stage k. The first set is P as defined in the previous section: $\{\text{path}\, i \in P | \hat{u}^i_k \neq \hat{u}_k\}$, and the second set Q is defined as $\{\text{path}\, j \in Q | \hat{u}^i_k = \hat{u}_k\}$.

In the original SOVA (as shown in Eq. 7.6), only the paths in P are taken into account. To consider the second-level paths, the output of SOVA is modified as in Eq. 7.8. The Δ^i_r and Δ^j_s are the metric differences between the ML path and the paths i and j, respectively.

Given a path $j \in Q$, a path set O is defined such that any path $l \in O$ must merge with path j and its decoded bit $\hat{u}_k^l \neq \hat{u}_k$. The variable Δ_t^l is the metric difference between path j and path l when they merge.

$$L_o(u_k) = \min_{i \in P, j \in Q, l \in O} \{\Delta_r^i, \Delta_s^j + \Delta_t^l\}; \tag{7.8}$$

From the above discussion, it is obvious that the SOVA output should be smaller when considering the second-level paths. In [80], it is found that the output of SOVA can be normalized with a scaling factor, which depends on the BER. With such a scaling factor, the performance can be improved by about 0.4 dB.

While the performance can be improved with complexity cost, a Soft-output adaptive Viterbi algorithm [22] is presented to reduce the computation complexity while yielding a little degradation in BER performance. Similar to the Adaptive Viterbi Algorithm [23], a threshold is introduced. At each stage, only those paths whose path metrics are higher than the threshold are preserved. As a result, the number of survivor paths is smaller, and therefore, it requires smaller memory storage and less computation. Using a 64-state code with $(g_1, g_2) = (115, 147)$, when the average survivor number is 31, the BER performance of SAVA is very close to that of the Viterbi Algorithm.

Similar reduced metric scheme are also presented in the MAP decoder for turbo codes [41]. Two strategies are explored. The *M-BCJR* algorithm keeps a fixed small number M of the best paths alive; the *T-BCJR* keeps all the paths above a certain threshold T alive. The results show that the latter is more efficient, in which the number of alive paths decrease at later iterations.

7.2.4 Turbo Code Adaptations

As mentioned above in Section 7.2.3, the metric states (survivor numbers) can be changed at run time. More adaptations can be done at the system level by varying the system parameters to realize the trade-offs in power, decoding speed and resource usage. An approach shown in [100] adapts the interleaver size to control the quality of service (QoS) in multimedia communications. Similar work involves changes in other system parameters, such as the iteration stop criterion, interleaver length, sliding window size, and puncturing rate [91, 100, 48]. The configuration of system parameters can be combined with other techniques for power savings. In [65, 44], Voltage Scaling is controlled by an iteration stopping criterion to save power. Other work [59] shows that the metric states can be made adaptive to conserve power.

The above adaptation algorithms have been mapped to silicon. A chip [14] built using $0.8\mu m$, 2-metal CMOS technology is able to decode turbo codes with varying puncture rates. A low-complexity multi-stage pipeline turbo encoder and decoder [51] are implemented in $0.6\mu m$ technology. For a 1Mbps data rate, the decoder power dissipation is $70mW$ with the supply voltage of $3.3V$.

A flexible turbo decoder was implemented in a ReConfigurable Processor Board (RCP) [49], a PCI board consisting six Altera FLEX 10K70 FPGAs and SRAM units. The implementation approximately uses 70,000 gates. It is able to configure the number of encoder states, interleaver length, sliding window size, decoding iterations and the quantization precision in software.

While it is easy to configure the system parameters, adapting the metric states inside the component decoder is more complicated. The reduced-search MAP algorithm was implemented using a Register Transfer Level (RTL) model for power saving [99]. It is synthesized using Synopsys Design Compiler with a $0.20\mu m$ standard cell library. The dynamic algorithm is compared with an iteration control strategy. It is shown that the latter approach is more efficient because it can power down the whole system or scale the system voltage. The dynamic algorithm can only reduce the memory power dissipation. Another approach to reduce memory power

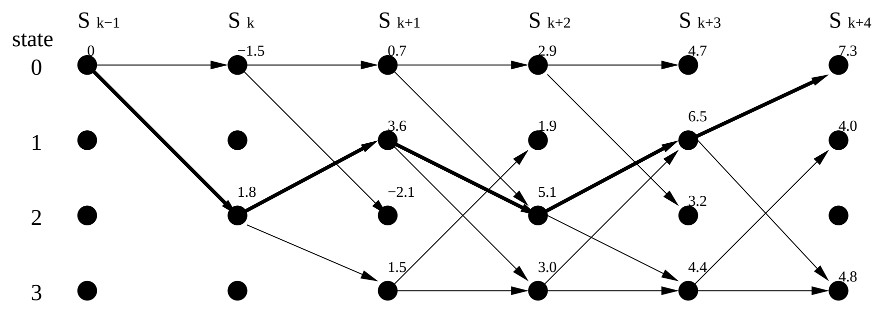


Figure 7.8. Trellis Diagram of Adaptive SOVA ($T = 1.0, N_{max} = 3$)

was presented in a SOVA turbo decoder using a novel orthogonal access memory structure [42].

All these implementations apply MAP-based decoders except [42]. Compared with SOVA, the hardware implementations of MAP-based algorithm were regarded as more efficient for its superior performance. With the presenting of ASOVA, the BER performance of ASOVA is competitive with Log-MAP, and it will be the preferred choice due to its lower complexity. While the previously mentioned parameter configuration schemes are still available, ASOVA allows further exploration in adapting the number of survivor paths to realize simpler and faster implementation.

7.3 Adaptive SOVA Decoding

In this section, an Adaptive SOVA (ASOVA) turbo decoder, whose performance is close to that of the Log-MAP algorithm, is described. The complexity of ASOVA is reduced, versus SOVA, by pruning the metric states. When compensated for the loss using the expectation and scaling factor, ASOVA loses little performance in terms of BER with about half complexity.

The ASOVA is developed based on the decoding trellis of SOVA [48]. As SOVA, the path metric of ASOVA, $M(S_k)$, is computed using Eq. 7.4 at each stage, and the reliability is given as Eq. 7.6. Two new parameters, T and N_{max}, are introduced in ASOVA to prune the metric paths. At each trellis stage, only those paths whose metric value satisfies Eq. 7.9 will be preserved. In the case that the number of preserved paths is more than N_{max}, only the best N_{max} paths will be kept. The N_{max} is defined as the maximum number of survivor paths allowed in the decoding trellis.

$$M(S_k) > \max\{M(S_{k_1})\} + T \tag{7.9}$$

An example is given in Figure 7.8, where the numbers on the node represent the paths metric of the survivor paths. Assume that $T = -2, N_{max} = 3$ and $M(S_{k-1} = 0) = 0$. At stage $(k+1)$, basing on the Eq. 7.9, the $M(S_{k+1})$ must be larger than $\max\{M(S_k)\} + T = 1.8 - 2.0 = -0.2$. As a result, the path $S_k = 2$ is pruned, whose path metric is -2.1. As shown in Figure 7.8 stage $(k+2)$, all four paths satisfy the threshold requirement, but only the paths at $S_{k+2} = 0, 2, 3$ are kept and the path at $S_{k+2} = 1$ is pruned since only the best $N_{max} = 3$ paths are allowed.

Since it is unlikely for a low metric path to become the ML path later, pruning the bad paths has only small possibility to change the ML path. As a result, the ASOVA still generates correct decoding sequence $\hat{\mathbf{u}}$ with a smaller number of metric paths, which reduces the decoding complexity and memory usage.

The soft output of ASOVA is generated by the metric difference between ML path and its closest competitor. When the paths with low metric are pruned, the competitors have higher possibility to be discarded because of its smaller path metrics than the ML path. Losing the closest competitors results in inaccurate soft outputs. In later decoding iterations, these soft outputs are the $L(u_k)$ when Eq. 7.4 is used to calculate the path metrics. The inaccurate soft outputs will result in incorrect path metrics and finally wrong decoding decisions.

The performance of a turbo decoder using reduced-state SOVA is shown in Figure 7.9. The experiment applies a (31,27) code which can 16 survivor paths at each stage

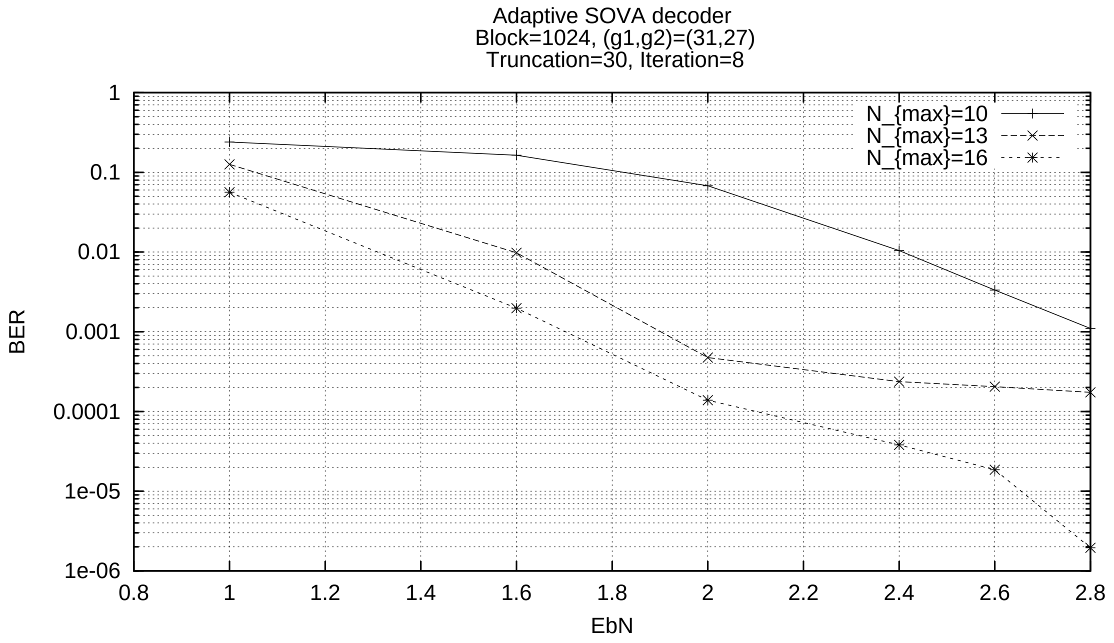


Figure 7.9. Performance with Varying N_{max}

without pruning. It can be seen that more than 0.8dB will be lost when the survivor number is restricted to $N_{max} = 13$.

To compensate the loss, two schemes are applied. The first scheme is to apply a scaling factor $\alpha < 1$ on the soft output, and the second scheme is to make use of the expectation of the soft outputs.

7.3.1 Scaling Factor

Generated using Eq. 7.6, the soft output is the minimum metric difference between ML path and the paths in *set P*. While pruning some paths, it must be $P_{ASOVA} \subseteq P_{SOVA}$, where P_{ASOVA} is the path set of ASOVA, and P_{SOVA} is the path set of SOVA. At a result, Eq. 7.10 is correct, i.e., the absolute value of ASOVA soft output is no smaller that that of the SOVA. A scaling factor $\alpha < 1$ is then used to compensate this error.

$$\min_{t=k,\ldots k+\delta, i \in P_{ASOVA}} \Delta_t^i \geq \min_{t=k,\ldots k+\delta, i \in P_{SOVA}} \Delta_t^i \tag{7.10}$$

When so many paths are pruned that no paths converge with the ML path within the truncation length, the set P_{ASOVA} is empty. Since the competitor is not available, the SOVA is unable to generate a soft output. While the decoded bit $\hat{u}_k$ is still available, the expectation of the absolute soft output, is used as its reliability value.

7.3.2 Expectation

The SOVA output is the Log Likelihood Ratio (LLR) of the received value y from the channel [80]. The LLR is given as Eq. 7.11.

$$Lo_y = ln \frac{P\{y|x=1\}}{P\{y|x=-1\}} \tag{7.11}$$

In an AWGN system, the received value y can be represented by the following formula:

$$y = u + n \tag{7.12}$$

where u is the input source with equal probability of {1,-1}, and n is the white Gaussian noise $N(0, \sigma^2)$.

It is easy to see that:

$$P\{y|u=1\} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-1)^2}{2\sigma^2}} \tag{7.13}$$

$$P\{y|u=-1\} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y+1)^2}{2\sigma^2}} \tag{7.14}$$

Make use of Eq. 7.11:

$$Lo_y = \frac{2}{\sigma^2} y \tag{7.15}$$

When it is decoded that $u = 1$, the expectation of LLR is

$$E\{Lo_y|u=1\} = \frac{2}{\sigma^2}(u+n) = \frac{2}{\sigma^2}(E\{u\} + E\{n\}) = \frac{2}{\sigma^2} \tag{7.16}$$

Eb/N_0	$2/\sigma^2$	$\sigma^2/2$
1.000000	1.678567	0.595746
1.200000	1.757676	0.568933
1.400000	1.840512	0.543327
1.600000	1.927253	0.518873
1.800000	2.018082	0.495520
2.000000	2.113191	0.473218
2.200000	2.212783	0.451920
2.400000	2.317068	0.431580
2.600000	2.426268	0.412156
2.800000	2.540615	0.393606

Table 7.1. Expectation of LLR

Otherwise, when $u = -1$,

$$E\{Lo_y|u = -1\} = -\frac{2}{\sigma^2} \tag{7.17}$$

Without losing generality, we discuss the absolute value of $E\{Lo_y\}$ only.

In my SOVA model, E_b is 3 since its bit rate is 1/3 and $E_s = 1$. We know that

$$\sigma^2 = N_0/2 \tag{7.18}$$

So the σ^2 can be calculated from the input signal noise ratio E_b/N_0, which is in dB, using the the following formula:

$$\sigma^2 = \frac{3}{10^{(E_b/N_0)/10} * 2} \tag{7.19}$$

$$|E\{Lo_y|u = \pm 1\}| = \frac{2}{\sigma^2} = \frac{4}{3} * 10^{(E_b/N_0)/10} \tag{7.20}$$

Some example expectation numbers are calculated in Table 7.1.

7.4 Simulation Results

A software turbo codes simulator was developed to test the performance of ASOVA using C++. The simulator consists of a turbo encoder, an AWGN channel,

Parameter	Default Value
Interleaver Length	1024 bit
Simulation Bit Number	$5 * 10^7$ bit
Channel Noise	1.5dB
Iteration Number	8
RSC Generator Formula	(31,27)
N_{max}	16
Threshold	-10
Truncation Length [1]	30
Code Rate	1/3

Table 7.2. Parameter Default Values

a turbo decoder and a performance evaluator. The turbo decoder is allowed to choose from six decoding algorithms, namely, Log-MAP, Max-Log-MAP, Adaptive MAP (AMAP, which prunes the state of Log-MAP), SOVA, Reg-SOVA, (which is SOVA using register exchange), and ASOVA.

Most key system parameters can be configured in this system. The default values used in the experiments are shown in Table 7.2.

The first set of experiments reveal the affect of the scaling factor, α, on the BER performance. A 10^7-bit randomly generated sequence is tested in each experiment to allow for enough BER accuracy. The performances of a (31,27) code with different α are tested and the results are shown in Figure7.10. The best BER is achieved when the scaling factor α is about 0.5. Similar values are obtained when other codes, such as (65,57), are used. As a result, the default value of α is set to be 0.5.

The BER values of ASOVA for different SNRs are shown in Figure 7.11. Similar values for the Log-MAP and MAX-LOG-MAP algorithms are also presented for comparison. It can be seen that the performance of ASOVA is between that of Log-MAX and Max-Log-MAP. When the *Threshold* increases, the average survivor number of ASOVA decreases. Generally, the higher the threshold is, the larger the BER will be.

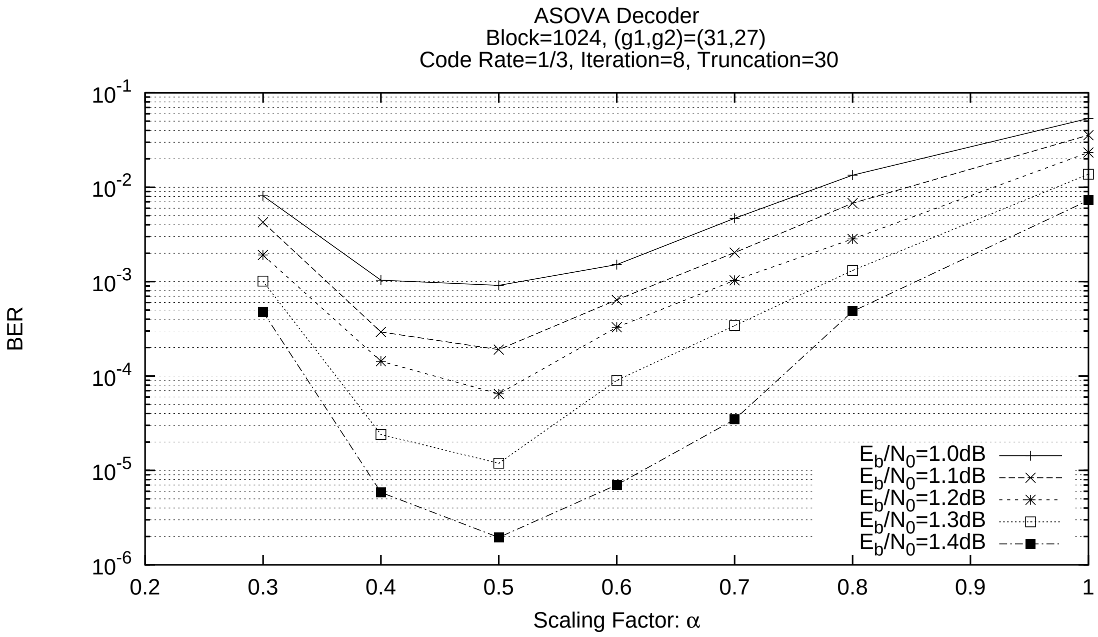


Figure 7.10. Performance with Varying α

The Figure 7.11 also shows that, when the average survivor states is reduced from full states (16 survivors) to 10, the BER numbers are very close. When the BER is 10^{-4}, the E_b/N_0 of Log-MAP is 0.9dB, that of Max-Log-MAP is 1.4dB, and those of ASOVA range ranges within 0.1dB from 1.15dB to 1.25dB.

The BER of ASOVA with varying Threshold and N_{max} is shown in Figure 7.12. It can be seen that, when $Threshold$ is greater than -8 and N_{max} is greater than 12, the BER is smaller than 10^{-4}.

The values of $Threshold$ and N_{max} to achieve a BER of 10^{-4} are obtained for other input SNR values through similar experiments. The resulting numbers are shown in Table 7.3. *Average States* is the average number of states per stage that have a survivor path. This number measures the actual computations required during decoding. It can be seen from Table 7.3 that, when E_b/N_0 is 1.5dB, the

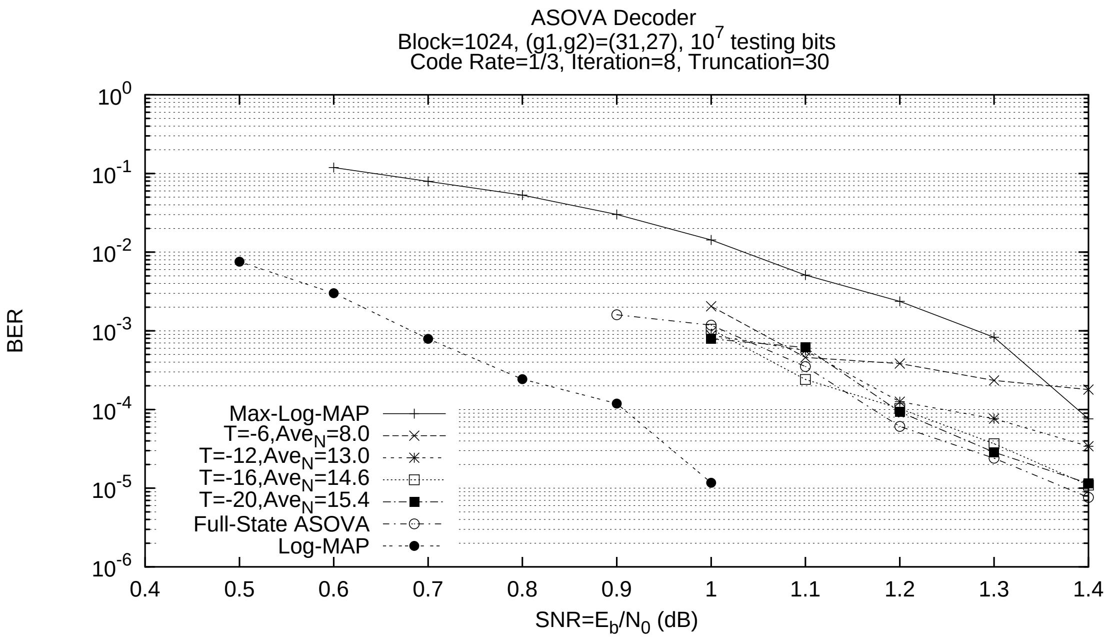


Figure 7.11. Performance of ASOVA

average states is around 6.55, which means that the computation is only $6.55/16 \approx$ 41% of SOVA, which has to deal with 16 survivor paths every stage. When E_b/N_0 is 5.0dB, the number decreases to $1.67/16 \approx 10\%$.

$E_b/N_0(dB)$	Threshold	N_{max}	Average States
1.5	-8	12	6.55360
2.0	-6	10	5.68129
2.5	-5	10	4.11785
3.0	-6	8	4.04745
3.5	-5	8	2.79940
4.0	-5	8	2.32015
4.5	-5	6	1.94780
5.0	-5	8	1.67064

Table 7.3. (31,27) Code at BER=10^{-4}

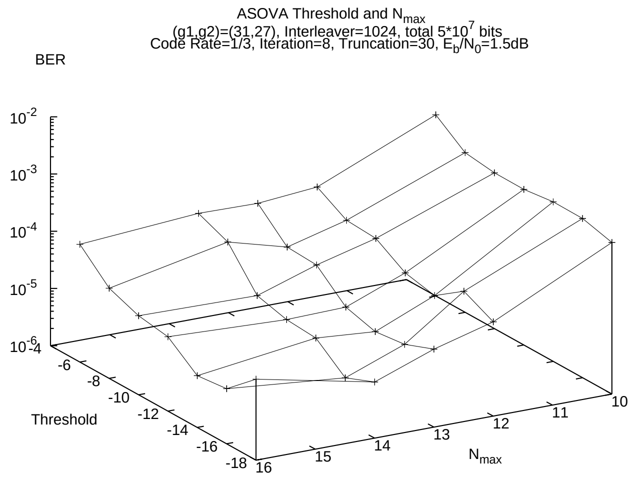


Figure 7.12. Affects of Threshold and N_{max}

Chapter 8

Hardware Implementation of ASOVA

To analyze the performance of aSoC for substantial applications with high data rate, a Turbo decoder using ASOVA is mapped onto aSoC. The theorem of ASOVA has been described in Chapter 7. In this chapter, the hardware implementation architecture is presented, especially, the trade-off between architectural choices and detailed construction of logic circuitry are revealed.

8.1 Architecture of Turbo Decoder

As shown in Figure 8.1, a typical turbo decoder [15] consists of two identical component decoders, *D1* and *D2*, interleaver/de-interleaver blocks, and an output decision block. The interleaver permutes the data bits to support the error correction algorithm. The output from one decoder is fed into the other as the a-priori information through an interleaver/de-interleaver to help the latter decoder make a better decoding decision in subsequent decoding iterations. Multiple iterations are

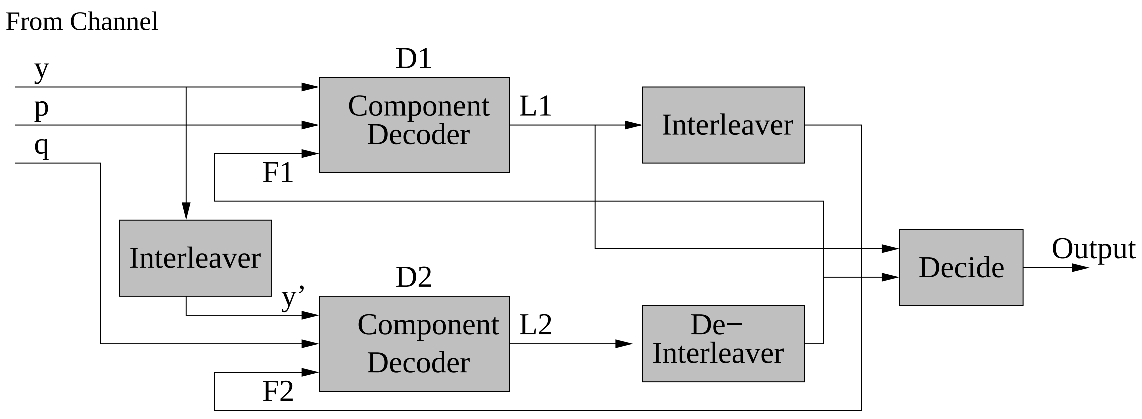


Figure 8.1. Block Diagram of Turbo Decoder

required before the decoder converges to a final result. After a pre-specified number of decoding iterations, the final decision is made in the *Decide* block by combining the outputs from both decoders.

To allow for pipelined decoding and achieve a high data rate, the interleaver and de-interleaver each include two data buffers. While one buffer receives new data, stored data can be read from the other. An entire block of code word can be stored in each buffer.

8.2 Architecture of Component Decoder

The component decoder is the key unit of the turbo decoder. Traditionally, a component decoder using SOVA was implemented using either *register-exchange* [47] or *memory traceback* [19]. The *register-exchange* approach employs a two-dimensional array to store the bit decisions of all possible paths over a certain time steps and constantly moves the bits through a pipeline. On the contrary, the *memory traceback* trades throughput, latency and memory size for circuit complexity by storing the decision bits into an SRAM in a constant position. Accessing only one data from the memory per clock cycle, it requires multiple cycles to complete the processing for each input data. With a slower throughput, *traceback* brings certain power benefits. However, in the case of a small number of states, little power advantage will be obtained because of the overhead of the peripheral circuitry and standard word addressing [19].

Since ASOVA stores only N_{max} paths when decoding, which results in a substantial difference path storage architecture than traditional SOVA, both *register-exchange* and *traceback* have to be re-evaluated. In the following two subsections, the architectures for both approaches are described.

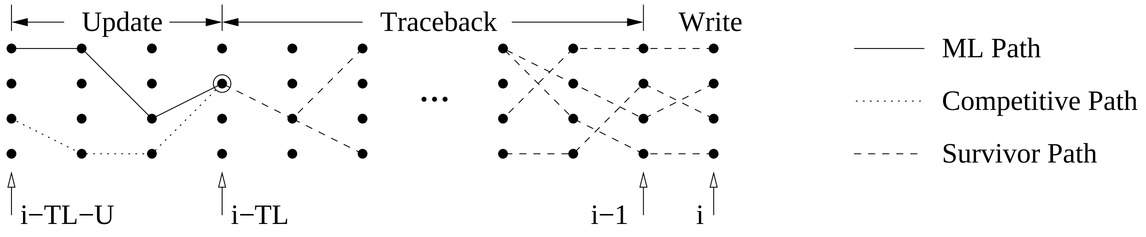


Figure 8.2. Traceback SOVA

8.2.1 Traceback Approach

8.2.1.1 Behavior of Traceback SOVA

The behavior of the *traceback* ASOVA component decoder can be described using Figure 8.2. The completion of a time step requires three phase as following:

1. Write the new path metric and metric difference Δ into path storage memory.

2. Traceback to locate the ML path.

3. Update the reliability information for the ML path.

As shown in Figure 8.2, without losing generality, assume that the new path metrics are written into column i of the path storage memory. Among the new path metrics, the largest path metric is determined to identify the Maximum Likelihood (ML) path. In Figure 8.2, the ML path is shown with solid lines. This ML path is obtained by a **Traceback** along the *truncation length* (TL) edges back to column *(i-TL)*. A typical value of the truncation length is five times the constraint length [82].

In addition to obtaining the decoded bit sequence, the SOVA algorithm determines reliability information Δ (also called soft output) for each decoded bit. Each Δ is determined by calculating the *difference* between the two path metrics that converge at each state node. An important phase of the SOVA algorithm is the dynamic **Update** of Δ in earlier trellis stages as later stages are reached [48]. As

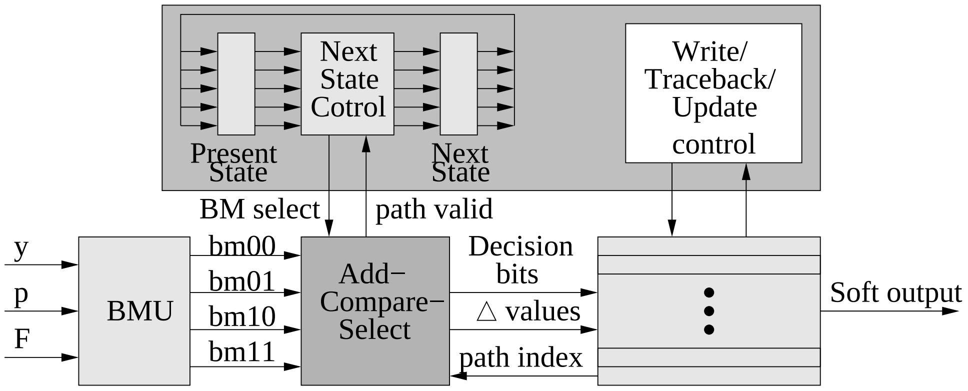


Figure 8.3. Architecture of Traceback Component Decoder

shown in Figure 8.2, after tracing back TL stages along the ML path from trellis state i, the next U stages are checked for Δ update via the following equation.

$$\Delta_t = \min_{\hat{u}_{ML(t)} \neq \hat{u}_{cmp(t)}} (\Delta_{i-TL}, \Delta_t), t = i - TL, ..., i - TL - U \tag{8.1}$$

where t is the stage index, and, $\hat{u}_{ML}$ and $\hat{u}_{cmp}$ are the decoded bit of ML path and the competitive path at the trellis stage and U is a parameter. It has been shown that U can be considerately less than TL without performance degradations [57].

To improve throughput, multiple *Write, Traceback* and *Update* phases can be used for one time step. Such scheme has been implemented for Viterbi algorithm [38].

8.2.1.2 Architecture of Traceback ASOVA

As shown in Figure 8.3, the component decoder consists of four parts: the branch metric unit (BMU), the add-compare-select (ACS) block, the survivor memory, and the control path. BMU and ACS are used to generate the new path metrics for each trellis stage. The survivor memory stores decoded bit u_k and soft output Δ_k values and performs **write**, **traceback**, and **update** operations. The control path

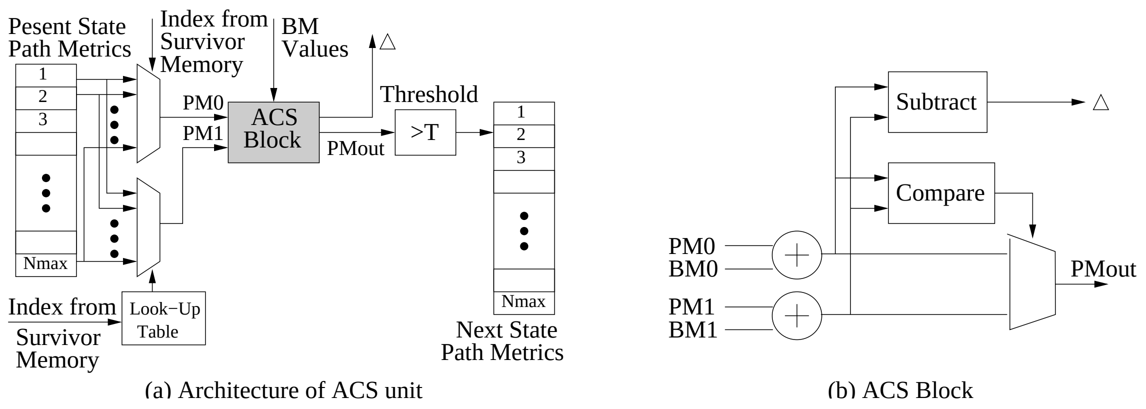


Figure 8.4. Add-Compare-Select Components

determines next state values and controls data flow between the other three units. Detailed architectural descriptions are provided in the following sections.

- Branch Metric Unit

  The BMU generates the branch metrics for all four possible encoder output pairs $\hat{u_k}, \hat{p_k}$. Received decoder input values y_k and p_k and the soft output feedback F from the alternate component decoder, (e.g. $F1$ and $F2$ in Figure 7.4) are used to generate the branch metric for each $\hat{u_k}, \hat{p_k}$ combination. For soft output decoders, channel values y_k and p_k are quantized to multi-bit values while $\hat{u_k}$ and $\hat{p_k}$ are single bits. For ASOVA, the branch metric of a given $\hat{u_k}$,$\hat{p_k}$ at trellis stage k with a soft output feedback F is:

$$bm(\hat{u}_k \hat{p}_k) = \frac{1}{2}\hat{u}_k F + \frac{L_c}{2}(y_k \hat{u}_k + p_k \hat{p}_k) \tag{8.2}$$

  where L_c is equivalent to $\frac{2}{\sigma^2}$, and σ is the standard deviation of the transmission channel noise. Since $\hat{u}_k$ and $\hat{p}_k$ are binary numbers, each BMU output requires two adders and a multiplier.

- Add-Compare-Select Unit

The goal of the add-compare-select unit is to add the branch metric for a trellis edge to the path metric of the present trellis state to create a new path metric for a *next state* in the next trellis stage. This metric is then compared to the computed path metric from the competitive path to determine the survivor for the next state. The ACS operation must be performed for at most N_{max} present state path metrics for each trellis stage.

The hardware architecture used to perform the ACS computation for each path is shown in Figure 8.4a. The correct index into the present state path metric array is obtained from the survivor memory and is used to select path metric $PM0$. This index is also used as an input to a pre-programmed look-up table to select the path metric $PM1$ for the competitive path. As shown in Figure 8.4b, the branch metric $BM0$, $BM1$ for each path is added to the appropriate path metric to create new path metrics for the next state. A subtractor takes the difference of the new metrics to generate the needed soft output Δ value for the next state. A comparator selects the larger of the two path metrics as $PMout$ for survival.

As shown in Figure 8.4, the ACS block employs the threshold $d_m + T$ to prune low cost paths. Only those paths whose metrics fulfill the threshold requirement are subsequently stored in the next state path metric array. The array index used to store the next state path metric is stored in the survivor memory. If more than N_{max} paths survive, the threshold T is dynamically increased and ACS computation is re-performed with the new T.

- Survivor Memory Unit

The survivor memory is a two dimensional memory array with N_{max} rows and $2{*}TL$ columns. The memory uses *traceback* pointers [25] so that data movement is limited. Each word in the survivor memory stores the decoded bit $\hat{u_k}$, metric difference Δ, and pointers to the previous trellis stage survivor memory values along the saved and competitive paths. As described above, the survivor memory supports three operations for up to N_{max} trellis states for each decoded input value: **write**, **traceback** and **update**. For a single trellis stage, a memory **write** requires a write port, **traceback** requires a read port, and the **update** phase requires two read ports to read ML and competitive path Δ values, and a write port for new Δ values. As a result, the survivor memory requires a total of 2 write ports and 3 read ports.

To facilitate FPGA implementation, our ASOVA memory is partitioned into banks. As shown in Figure 8.2, survivor memory **traceback** and **write** operations occur in portions of the memory that are isolated from the **update** phase. As a result, the survivor memory is partitioned vertically into eight separate banks. Four banks store the two path indexes and $\hat{u_k}$ values and the other four banks store the Δ values. The former four banks of memory requires two read-ports and one write-port since the **update** phase has to read the index for both the ML path and the competitive path. The memory of latter four banks can be implemented using general two-port RAM blocks. A single memory read and write operation is performed in one clock cycle.

8.2.2 Register Exchange Approach

8.2.2.1 Behavior of Register Exchange

In the *traceback* approach, the centralized memory architecture allows only a single read and write operation per clock cycle. As a result, it takes at least TL cycles to complete all operations for a time step. The *register exchange* approach employs a parallelized architecture, so that a time step can be completed in one clock cycle.

While *traceback* generates one path metric per cycle, the *register exchange* computes all the $2 \times N_{max}$ path metrics in one cycle using a Add-Select-Compare (ACS) Unit consisting of $2 \times N_{max}$ ACS blocks.

The $2 \times N_{max}$ new obtained path metrics and Δs are pruned down to N_{max} and then written into the path storage memory in a clock cycle. To complete multiple write operations in one cycle, a register array is used, with N_{max} rows and TL columns. The path metrics and Δ of a path are stored in the same row. As a result, indexes to previous path metric are not required to obtain the history of a given path. The traceback operation can access the whole row in one clock cycle.

With the register storage, the *Update* can happen the same time as the new metrics are written. When up to N_{max} survivor paths are determined by the ACS unit, each path is assigned to a row in the register array. The historical metrics and Δ of each path are copied from their old locations to the new row. That is the how the name of *register exchange* comes. During the exchanging, all the Δs of each path are updated following the Eq. 8.1.

The *register exchange* approach parallelizes and pipelines the ACS and update operation. It is able to achieve a data rate as high as one data per clock cycle. On the other hand, it requires a large amount of multiplexing circuitry between the registers to choose the metric and Δ data. In addition, all the storage registers are active during the decoding procedure, which consume a lot of power and become a drawback for power-constrained systems.

8.2.2.2 Architecture of Register Exchange

The architecture of the *register exchange* component decoder follows the blocks shown in Figure 8.5. It consists of four parts: the Branch Metric Unit (BMU), the Add-Compare-Select (ACS) Unit, the Survivor Memory Unit (SMU), and the Metric Difference Memory (MDM).

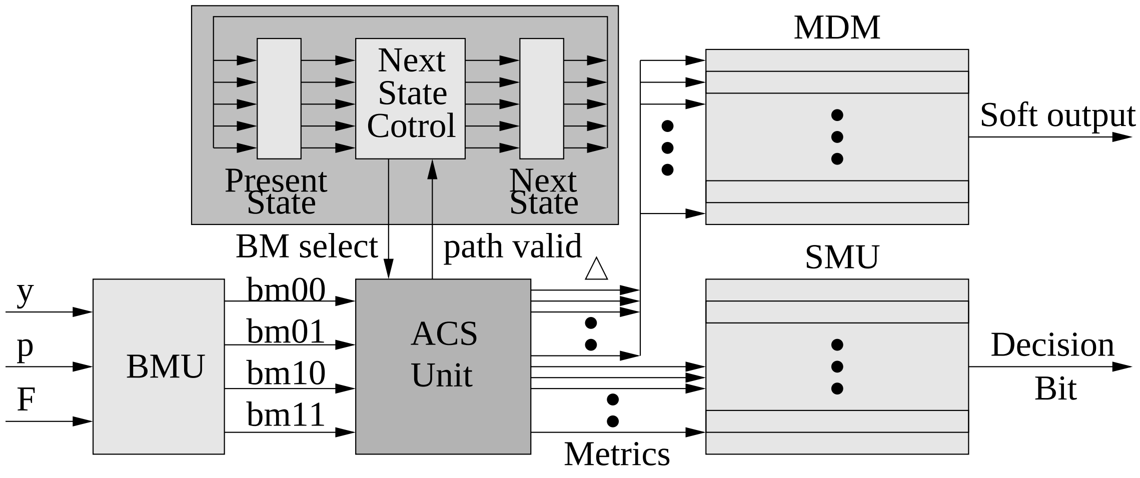


Figure 8.5. Block Diagram of Register Exchange ASOVA Component Decoder

The component decoder has a pipelined architecture, with BMU on the first stage, ACS on the second stage, and the third stage of SMU and MDM. For each decoding step, new branch metrics are generated by BMU. Combining the branch metrics with the path metrics obtained at the previous step, the ACS chooses the survivor paths and stores its decisions into the SMU. Assuming that the decoder is currently working on level i, the SMU stores the trellis array from level $(i - TL)$ to level i. For each step, the SMU shifts one level right to empty the last column for the new in-coming data from ACS. MDM, which stores the metric difference for the same part of trellis as the SMU, also shifts one level to the right. At the same time with shifting, it uses the ith level metric difference data from ACS to update the old metric difference for each path. The data shifted out from the right of MDM is the reliability information of the (i-TL)th level.

The SMU and MDM store TL levels of decision bits and metric differences. At each level, while traditional SOVA uses $2^{(K-1)}$ states, where K is the constraint length, only N_{max} states are required for ASOVA.

The detailed architecture of BMU, ACS, SMU and MDM are introduced as following:

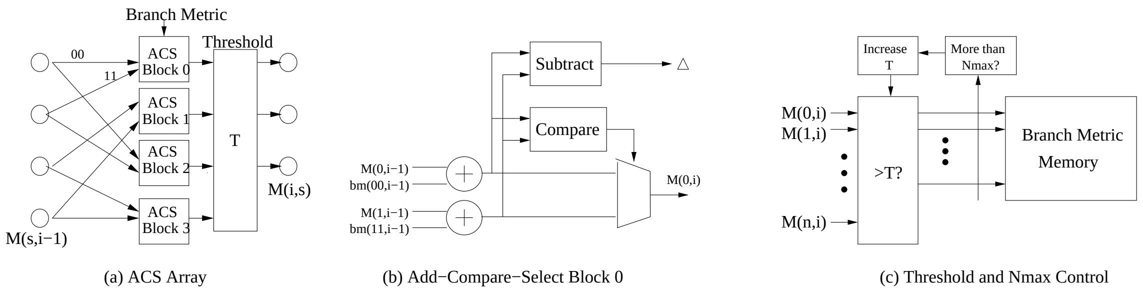


Figure 8.6. Architecture of Add-Compare-Select Unit

1. Branch Metric Unit

   The BMU is the same as the BMU in *traceback* approach. It generates the branch metric for all $(\hat{u}_k\hat{p}_k)$. It consists of four outputs, since there are four possible combinations of $(\hat{u}_k\hat{p}_k)$. As described as Eq. 8.2, $bm(\hat{u}_k\hat{p}_k)$ is the Branch Metric of a given $(\hat{u}_k\hat{p}_k)$ at level k, which is actually the last three terms of Eq. 7.4.

2. Add-Compare-Select Unit

   Given the branch metrics from BMU, the ACS blocks are applied to decide the survivors and the metric differences Δ between the survivor path and the discarded path. Unlike traditional SOVA, a threshold T is applied in the ACS of ASOVA to prune the bad paths and keeps at most N_{max} survivors.

   Figure 8.6(a) presents an ACS unit with 4 states and $N_{max} = 3$. $M(s, i-1)$ represents the path metric of state s level *(i-1)*. An ACS block computes the path metrics of the two paths that merge into the same state, and selects the survivor path for the next level. Each block consists of three adders and a multiplexer. Figure 8.6(b) shows the architecture of *ACS Block 0*, which compares the two paths merging into *state 0*. Since the two paths come from *state 0* and *state 1* with $(\hat{u}\hat{p}) = 00$ and 11 respectively, $M(0, i-1), bm(00, i-1)$ and $M(1, i-1), bm(11, i-1)$ are used as inputs.

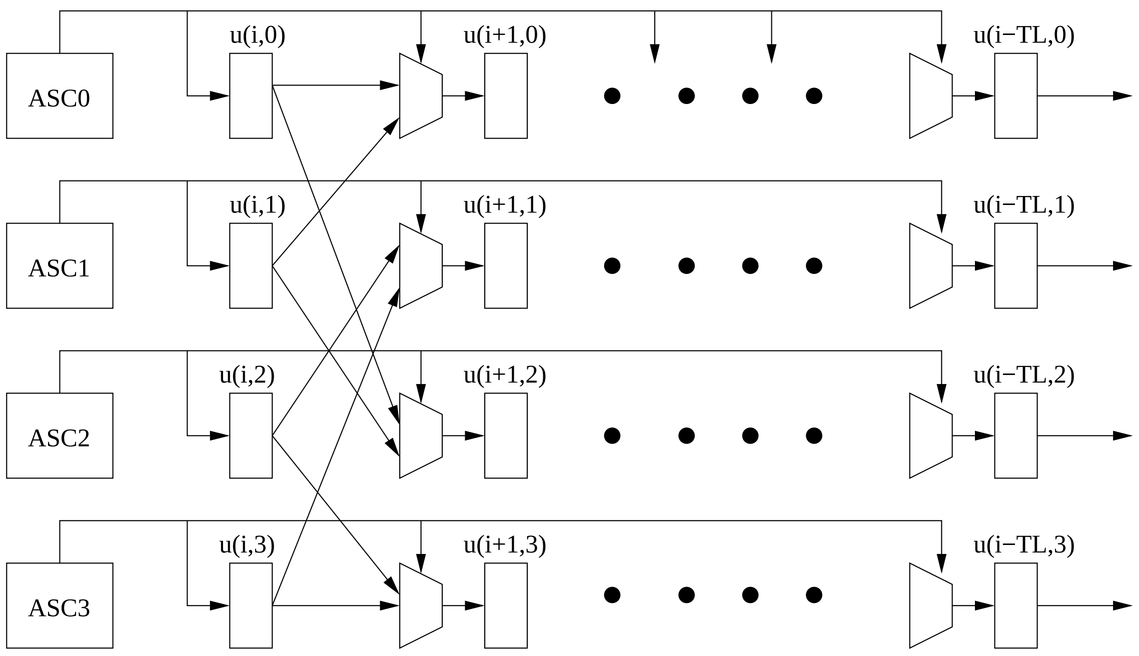


Figure 8.7. Architecture of Survivor-Memory-Unit

To find the best N_{max} paths and avoid the area-costly sorting circuit, an architecture was proposed in [96]. As shown in Figure 8.6(c), all the paths have to pass a pre-set threshold T. If the number of survivors is more than N_{max}, T will be increased until the survivor number is no greater than N_{max}.

In ASOVA, the memory records N_{max} states and each state can branch to two paths, so up to $2N_{max}$ comparers are used if $2N_{max} < 2^{(K-1)}$.

3. Survivor Memory Unit

Figure 8.7 shows a 4-state SMU. At each step, the ACS decision will be propagated to the whole row. The registers on this row are updated with the decisions of the survivor path. Since each state can only come from two possible states, a multiplexer controlled by the decision bit is enough to make the choice.

In ASOVA, only N_{max} but not all $2^{(K-1)}$ paths are recorded. The locations

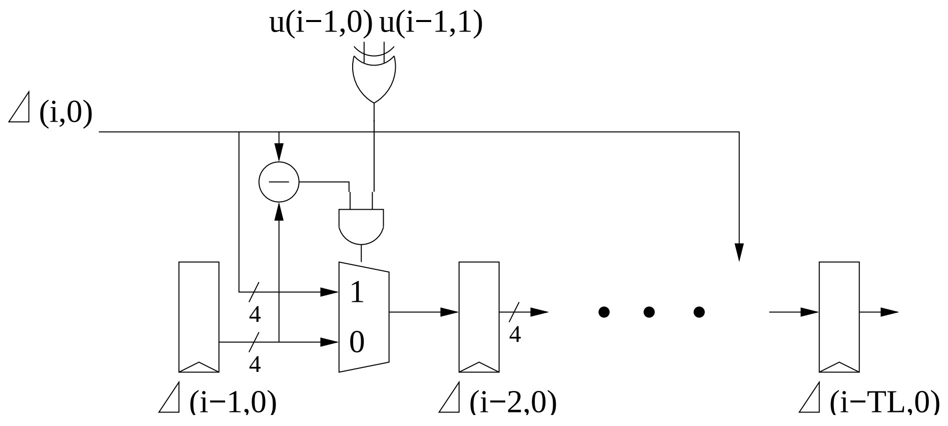


Figure 8.8. Metric Difference Memory Architecture

of the paths are dynamically determined, so the paths merging into a state are not coming from a fixed location. To solve this problem, extra $2N_{max}$ registers, which is named as *PathIdentify*, are used to record the state of the path. Detailed circuitry can be found in [96].

4. Metric Difference Memory

In addition to the bit decision information used for Viterbi Algorithm, SOVA also gives the reliability of each decoded bit, which is generated from the metric difference as Eq. 7.6. The computation of this reliability is done by the Metric Difference Memory.

MDM stores the metric difference Δ of the N_{max} survivors over the TL levels, covering the same range as SMU. At each step, when ACS make a decision of a path, this decision is propagated over the whole row to update the Δs of this path. As described in Chapter 7, the Δ will be getting close to the value of reliability of the decoding decision along with the updating.

Figure 8.8 gives an example of a row of MDM. At the (i-1)*th* level, if the decoded bit of path 0, $u(i-1,0)$, and path 1, $u(i-1,1)$, which are obtained from SMU, are different, the new $\Delta(i-1,0)$ will be the smaller of the original

$\Delta(i-1,0)$ and the new metric difference $\Delta(i,0)$ from ACS. When shifting one level to the right, the new $\Delta(i-1,0)$ will be stored into the location of where $\Delta(i-2,0)$ was.

All the reliability informations in the same row, which belong to the same path, are updated at the same time and shifted to the right by one level. The $\Delta(i-TL,0)$, which has been updated by TL times, is popped out from the right side of MDM as the reliability of path 0 at level $(i-TL)$.

While the MDM of traditional SOVA requires $2^{(K-1)}$ rows, ASOVA uses only N_{max} rows. The same as SMU, the ASOVA MDM is also a N_{max} by TL array, but the memory cell of MDM has 4 bits.

8.3 Experimental Approach of FPGA Implementation

Before the ASOVA Turbo decoder can be mapped onto aSoC, it has to be tested in FPGA. First of all, the ASOVA is a novel algorithm and requires totally different storage memory organization from traditional SOVA. The FPGA evaluation helps to choose the proper architecture between the *register exchange* and the *traceback* approaches. Second, proper parameters and performance numbers for individual functional blocks are required when the Turbo decoder is mapped onto aSoC. Third, the overall performance of FPGA implementation can be compared with the aSoC model.

8.3.1 FPGA Implementation Parameters

Parameterizable Turbo decoders with both *traceback* and *register exchange* approaches are developed with Verilog. Decoders for a variety of K and N_{max} values were synthesized to an Altera Stratix EP1S10 FPGA and downloaded to a Nios Development Board [6].

The bit width of the path metric and Δ depends on the accuracy requirement of the system. It was known that a word-length of 3 bits is almost optimum for the symbols in the case of BPSK modulation [25]. In our implementation, it was determined that a bit width of 4 bits is good enough for these experiments.

In the following experiments, turbo decoders were tested with 1024 bit data blocks and 6 decode iterations. In the *traceback* model, the survivor memory was constructed with eight memory banks which can hold $10 \times K$ trellis columns of $\hat{u_k}$ and Δ information ($5 \times K$ for **traceback**, $2.5 \times K$ for **update**, $2.5 \times K$ for rotating spare storage). In the *register exchange* model, the SMU and MDM are designed to hold $5 \times K$ trellis columns of $\hat{u_k}$ and Δ information.

8.3.2 Experimental System

To test the practicality of the reconfigurable ASOVA-based architecture, a hardware implementation of the decoder was tested as part of a communication system. This system contains blocks for data generation, encoding, transmission, and decoding. A random bit generator creates a bit sequence to model transmitted data. A turbo encoder, also shown in Figure 7.3, then encodes the data for transmission. A *modulator* converts the coded bits into real numbers: 0 -> 1, 1 -> -1 for the binary phase-shift keyed (BPSK) system employed. The output of the modulator is input to a *AWGN channel simulator*. This block simulates a noisy channel where white Gaussian noise is added to the transmitted signal. The amount of noise depends on the signal-to-noise ratio preset by the user. The symbols obtained from the AWGN channel model are quantized before being sent to the *decoder* as its input. On receiving the input, the decoder attempts to recover the original sequence. All software modeling of the communication system (except for the FPGA-based decoder) was performed using a 1.6 GHz Pentium IV PC.

The ASOVA-based decoder architecture was mapped to a Stratix EP1S10 FPGA located on an Altera NIOS Development Board [6]. This mapping allowed for in-field testing of turbo decoder designs for constraint lengths up to K=6. An

RTL level description of the turbo decoder was written in Verilog. The Verilog code was simulated using Altera Quartus II simulation tools. All designs were synthesized and mapped using Quartus II with timing constraints. The maximum operating frequencies of the FPGA were obtained from Quartus II compilation. Overall communication system decode rates were measured through profiling with the *time* utility on the PC.

Power consumption values for the turbo decoders were determined using the Quartus II power analyzer. To account for power consumption during EP1S10 reconfiguration, the power associated with reading the configuration bitstream from SDRAM and storing it in the FPGA was calculated. It was determined that approximately 125 mW of power are needed during reconfiguration to read the 3,534,640 EP1S10 configuration bits from 4M$\times$x32 Micron MT48LC4M32B2 SDRAM [73]. This value was determined by scaling the specified maximum power dissipation to 100 MHz, which is the required FPGA configuration speed. The amount of power required to reconfigure the EP1S10 was approximated by assuming the use of a on-chip reconfiguration shift chain. The power dissipated by the shift chain was determined by calculating the energy dissipated by a single shift in 0.13um technology with SPICE. This shift chain power value was scaled by the required 3,534,640 shifts and divided by configuration time to calculate FPGA reconfiguration power. It was calculated that 54.8 mW are required to reprogram the configuration bits of the EP1S10. To account for the reconfiguration time overhead, it was determined that a total time of 35mS was required to reconfigure a EP1S10 FPGA [6].

An experimental system is established as shown in Figure 8.9. The Turbo decoding board receives channel data from the receiver through Ethernet. In real system, the receiver might be a piece of hardware that receives channel signals from the remote antenna. In our experiment, a computer is used to send the data to the

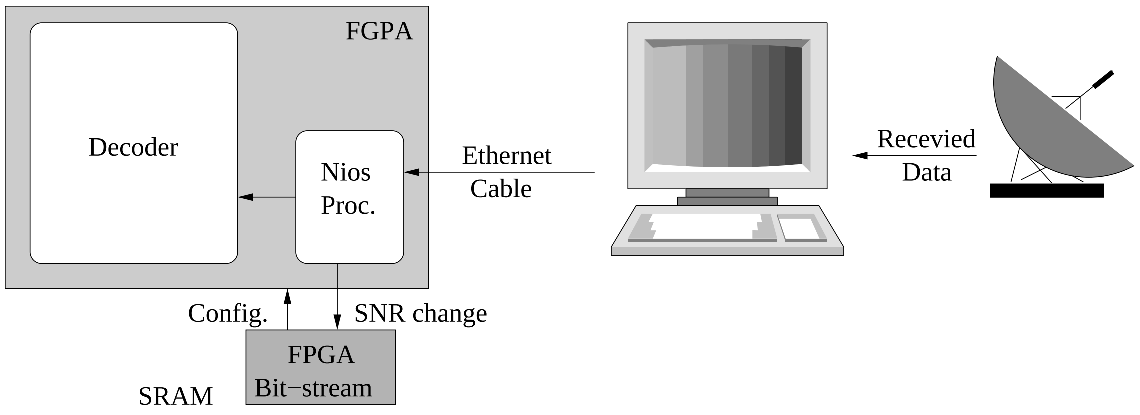


Figure 8.9. Turbo Code Experiment System

FPGA board for decoding. Insides the Stratix FPGA, a Nios micro processor [5] is implemented to measure the SNR when relaying the received data to the decoder. When the SNR changes, the processor picks up a proper bit-stream stored in the SRAM, reconfigures the FPGA and generates a new decoder appropriate for the current channel noise statistics.

Chapter 9

Map ASOVA Turbo Decoder onto aSoC

The ASOVA Turbo decoder has to be verified before mapping it onto aSoC devices. In this chapter, the implementation results of the ASOVA Turbo decoder on FPGAs are presented. Both *register exchange* and *traceback* approaches are evaluated and the proper architecture is chosen for later experiments. The functionality and performance of the ASOVA Turbo decoder was tested in the FPGA board, including decoding speed, resources usage and power consumption. Based on the performance and parameters obtained from the FPGA hardware implementation, the ASOVA Turbo decoder is mapped onto aSoC devices.

9.1 Results of FPGA Implementation

9.1.1 Register Exchange FPGA Turbo Decoder

As described in Chapter 8, Turbo decoders with both *register exchange* and *traceback* are implemented in Verilog and mapped onto a Stratix FPGA on the Nios board [6].

The mapping results of the Turbo decoder using *register exchange* is shown in Table 9.1. It can be seen from the results that this Turbo decoder can work on a clock frequency up to 96MHz, and have a decoding data rate up to about 8Mbps. Unfortunately, it takes huge resources when N_{max} is greater. When N_{max} is 4, it requires about 100K LUTs, which is more than the largest Stratix FPGA can provide. Since Quartus II is unable to map it onto any possible FPGAs, the number of resources and performance are unavailable. The huge resource usage is because of the multiplexing circuitry required in the ACS and SMU. In the ACS, it has to

N_{max}	LUTs	MEM (bit)	FFs	Max. Clk (MHz)	Power (mW)	Data Rate (Mbps)
2	55048	684	6144	96.68	360	7.9
4	83194	1247	12288	-	-	-

Table 9.1. Register Exchange Turbo Decoder Statistics

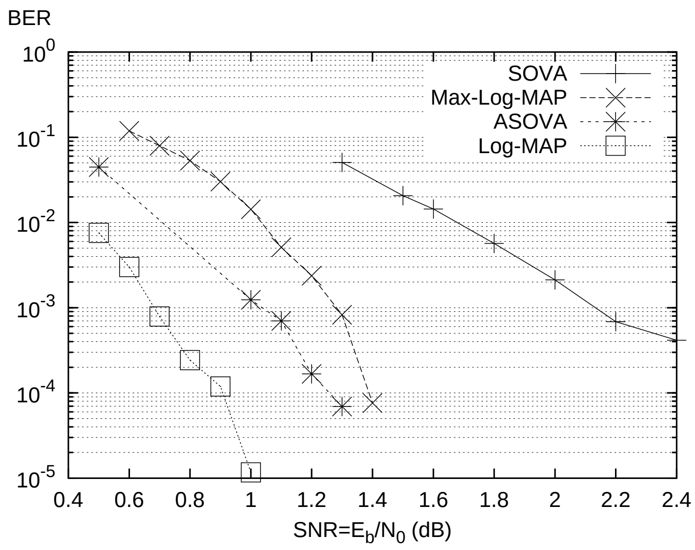


Figure 9.1. ASOVA Performance for a (31,27) code versus competing decoder algorithms

locate the two merging path from the randomly ranged N_{max} previous states for each next state, and there are up to $2 \times N_{max}$ next states. In the SMU and MDM, each metrics and metric difference Δ could potentially be updated by any of the other $N_{max} - 1$ states. In the whole storage array, there are $TL \times N_{max}$ storage units. As a result, since the *register exchange* approach, although with a high data rate, is not appropriate for the FPGA implementation of ASOVA Turbo decoder, the *traceback* approach is chosen.

9.1.2 Traceback FPGA Turbo Decoder

Prior to implementing the ASOVA component decoder in hardware, a set of simulations were performed to evaluate appropriate T and N_{max} values for the

ASOVA-based decoders. Via simulation it was determined that a value of $T = -10$ and the associated N_{max} values, shown in Table 9.2, were best suited for our decoders for a fixed BER of 10^{-4}. The signal-to-noise ratio (SNR) range supported by each tested decoder is shown in Table 9.2. For a constraint length K of $4, 5$, and 6, the codes of (15, 13), (31,27) and (65,57) was used respectively. When the SNR is high, a reduced N_{max} can be used to obtain the required BER.

Parameter values were used to evaluate the ASOVA decoders' error-correcting performance versus competing component decoders to verify the performance benefits of this approach. For comparison purposes, software versions of turbo decoders based on SOVA, Log-MAP, and Max-Log-MAP component decoders were developed. Figure 9.1 indicates that the BER performance of ASOVA is superior to the original SOVA without the scaling factor α, and approaches the performance of the computationally more expensive Log-MAP algorithm for the (31,27) code ($K = 5$). Other codes demonstrated similar results.

9.1.2.1 Traceback ASOVA Turbo Decoder Statistics

To test the power consumption and decoding speed of our ASOVA-based turbo decoders, a parameterizable decoder was written in Verilog. Decoders for a variety of K and N_{max} values were synthesized to an Altera Stratix EP1S10 FPGA and downloaded to a Nios Development Board [6]. In the following experiments, turbo decoders were tested with 1024 bit data blocks and 6 decode iterations. The survivor memory was constructed from eight memory banks with the capacity to hold $10 \times K$ trellis columns of $\hat{u_k}$ and Δ information ($5 \times K$ for **traceback**, $2.5 \times K$ for **update**, $2.5 \times K$ for rotating spare storage).

Table 9.2 illustrates the hardware resource usage of the decoders. Table 9.3 shows the decode rate and power consumption of the decoders for a range of K and N_{max} values. Two sets of power and decode rate values were determined: values at

K	N_{max}	SNR (dB)	LUTs	MEM (Kbit)	FFs
6	32	0-1.5	4611	135	2523
6	28	1.5-2.0	4407	133	2347
6	18	2.0-2.5	3814	126	1907
6	12	2.5-3.0	3100	65.0	1503
6	9	3.0-4.0	2851	63.1	1371
6	7	4.0-4.5	2524	37.2	1143
6	6	4.5-5.5	2406	36.6	1099
6	5	5.5-6.0	2317	36.0	1055
6	4	>6.0	1972	25.1	871
5	16	0-1.5	2809	67.6	1293
5	12	1.5-2.0	2587	65.0	1133
5	9	2.0-2.5	2392	63.1	1013
5	8	2.5-3.0	2202	37.9	897
5	6	3.0-4.0	2074	36.6	817
5	5	4.0-5.0	1996	26.0	777
5	4	5.0-6.5	1768	25.1	661
5	3	>6.5	1722	24.4	621
4	7	0-2.0	1896	26.8	687
4	6	2.0-2.5	1831	26.5	651
4	5	2.5-4.0	1752	26.2	615
4	4	4.0-5.5	1563	20.7	535
4	3	>5.5	1541	20.4	499

Table 9.2. ASOVA Decoder Statistics for BER=10^{-4} and T=-10

50 MHz, the clock speed of the NIOS board, and values for the maximum possible clock rate for the decoder. Clock speeds of nearly 90 MHz were found for smaller decoders. For a 50 MHz decoder it can seen from the table that for $K = 5$, there is a 51% power reduction for the N_{max} changing from 16 to 3, and for $K = 6$, there is a 67% power reduction across N_{max} values from 32 to 4.

9.1.2.2 ASOVA Dynamic Reconfiguration

A second set of experiments were used to determine power savings that could be achieved if the entire FPGA decoder was reconfigured at run-time to support

		50 MHz		Max speed		
K	N_{max}	Power (mW)	Speed (Kbps)	f_{max} (MHz)	Power (mW)	Speed (Kbps)
6	32	447.7	173.4	52.9	469.1	183.4
6	28	431.3	193.6	56.7	485.1	219.4
6	18	306.9	228.2	59.3	431.3	270.5
6	12	232.6	288.6	60.0	279.6	346.5
6	9	212.8	410.9	67.1	292.9	551.4
6	7	177.9	447.1	66.1	233.0	590.2
6	6	173.8	450.2	68.7	228.4	619.5
6	5	168.4	501.3	67.4	215.8	677.9
6	4	147.3	469.2	77.8	159.1	728.6
5	16	205.8	312.2	70.5	280.5	440.0
5	12	193.0	301.0	70.4	250.4	424.0
5	9	148.3	411.3	73.7	206.1	606.4
5	8	169.6	444.8	74.6	246.5	663.5
5	6	130.1	468.6	79.7	181.9	746.9
5	5	134.3	470.1	80.7	198.9	758.5
5	4	110.4	472.6	82.8	163.4	782.2
5	3	100.1	471.7	82.3	143.4	776.2
4	7	134.3	487.8	81.2	248.9	792.5
4	6	125.9	515.6	81.8	204.1	851.2
4	5	113.0	622.8	80.2	198.0	851.2
4	4	106.7	688.2	84.1	176.5	1216.0
4	3	98.6	734.6	89.0	185.2	1178.0

Table 9.3. ASOVA Performance on a Stratix EP1S10 FPGA

existent channel SNR requirements. Depending on the SNR, power savings are achieved by using a lower N_{max}, lower-power decoder for high SNR and a higher N_{max}, higher-power decoder for low SNR. The three constraint lengths K offered three separate SNR ranges for testing.

Wireless communication channels are affected by three propagation mechanisms: path-loss, shadowing, and multipath fading. The first, path-loss, represents the loss in signal strength with increasing distance of the receiver from the transmitter, and, hence, varies significantly only over relatively long time scales. The second effect,

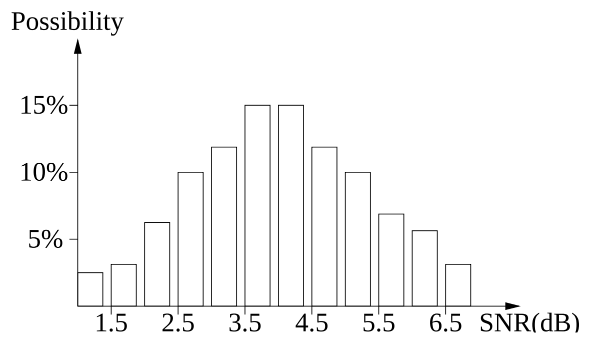


Figure 9.2. SNR Distribution of Log-Normal Shadowing Channel

shadowing, is caused by the presence of large objects between the transmitter and the receiver. The obstructions essentially form "dead zones", where the transmitted signal can be greatly attenuated from that expected at a given distance. Shadowing, which is often modeled instantaneously as a log-normal random variable, changes the average received signal-to-noise (SNR) ratio at moderate time scales, thus allowing it to be measured and used for system adjustment in the second- and third-generation wireless systems. Finally, multipath fading is caused by the constructive or destructive interference effects of the summation of the many reflected transmitted signals. Although multipath fading can cause significant fluctuations in signal strength, its rapid fluctuation allows the system to average its effects - thus greatly ameliorating its impact on the system in many cases.

In this dissertation, we focus on the effects of the *log-normal shadowing*. In particular, it is of interest to consider how measurements of fluctuations in the local average signal-to-noise ratio can be used dynamically to optimize system performance. This well matches the information available in many second and third generation wireless cellular systems [76].

A set of 10,000 SNR values were sampled for each K using a log-normal shadowing distribution [83] for a total transmission length of 2.5 billion bits. Figure 9.2 presents the distribution of the sampled SNRs. Based on the assumption that SNR can be sampled successfully every 250K bits [82], the FPGA was periodically reconfigured during the transmission process. Table 9.4 shows the number of required reconfigurations, the resulting decode rates at 50 MHz, and the average power dissipated. The average power consumption for the (31,27) code ($K = 5$) is 131 mW, a 36% improvement over a fixed $N_{max} = 16$ decoder. For a (65,57) code ($K = 6$), the average power of 216 mW is 52% less than the power of the fixed $N_{max} = 32$ decoder, 448 mW. Power and decode rate numbers include the time and power needed for FPGA reconfiguration, as described in Chapter 8.

9.1.2.3 Comparison to Microprocessor Implementations

Although the parallelism and memory structure of turbo decoders make efficient implementation on a microprocessor difficult, we contrasted the software performance of ASOVA on two microprocessors versus FPGA hardware implementations. Software results were determined using the 1.6 GHz Pentium IV PC (the host for the NIOS board) and the 50 MHz NIOS processor running on the FPGA board. The results for $K = 4$, 5, and 6 are shown in Table 9.5. For a given K and N_{max}, the 50 MHz FPGA decoder outperformed software implemented on the Pentium IV by over two orders of magnitude. The NIOS processor power consumption was approximately 630 mW for all decoders, and was 30% larger than the highest power consumption for an FPGA decoder (447 mW).

In a final experiment we performed a direct comparison between FPGA decode rates on the NIOS board (including PC-to-board transfer overheads) and Pentium IV PC decode rates. When 100 Mbps Ethernet PC-to-board delays are considered, the overall decode speed for a $K = 6$, $N_{max} = 18$ decoder is 211.1 Kbps and the overall decoder speed for a $K = 5$, $N_{max} = 12$ decoder is 229.7 Kbps. These values are still more than two orders of magnitude faster than corresponding Pentium IV PC decoders with data rates of 1.3 Kbps and 2.1 Kbps, respectively.

K	Avg. Speed (Kbps)	Reconfigures required	Avg. Power (mW)
4	598.6	6925/10000	111.6
5	429.4	6306/10000	131.7
6	359.1	8369/10000	216.2

Table 9.4. Dynamic Reconfiguration

9.1.2.4 Comparison to Digital Signal Processors

The FPGA implementation of ASOVA Turbo decoder is compared to a SA-1100 low-power processor [53], and a TMS320C6713 digital signal processor (DSP) [98], to further evaluate its performance.

The technology parameters of the compared architectures are presented in Table 9.6. The decoding speed and power consumption are shown in Table 9.7. The SA-1100 is evaluated by JouleTrack [93]. The decoding performance of TMS320C6713 is obtained from the simulation using the Code Composer Studio (CCStudio) Development Tools [97]. The power consumption of the TMS320C6713 is about 1.7W based [98]. Generally, the power consumption of DSPs does not vary too much for applications, which can also been seen from the power consumed by the SA-1100 in Table 9.7.

9.1.2.5 Comparison to Commercial FPGA Implementations

Recently, Altera and Xilinx published their commercial Turbo decoder cores using Log-MAP algorithm [8, 109]. A comparison is presented in Table 9.8. The performance of all decoders are based on a (31,27) code with 5 iterations. For a fair comparison, the N_{max} of ASOVA is set to be the full state of 8, so that the ASOVA decoder covers the same SNR range.

The Altera core uses 5-bit soft information, which mainly relates to the size of the required memory. A bitwidth of 6 is used for the soft information in the Xilinx

K	N_{max}	Pentium IV (Kbps)	NIOS (Kbps)	FPGA Hardware (Kbps)
6	32	0.784	0.003	173.4
6	18	1.344	0.005	228.2
6	12	2.064	0.007	288.6
6	9	2.730	0.009	410.9
6	7	3.382	0.011	447.1
6	6	3.615	0.013	450.2
6	5	4.227	0.015	501.3
6	4	5.294	0.019	469.2
6	3	6.239	0.024	471.4
5	16	1.589	0.006	312.2
5	12	2.048	0.008	301.0
5	9	2.661	0.010	411.3
5	8	3.013	0.012	444.8
5	6	4.160	0.015	468.6
5	5	4.899	0.019	470.1
5	4	5.824	0.022	472.6
4	7	3.177	0.014	487.8
4	6	3.404	0.017	515.6
4	5	4.193	0.020	622.8
4	4	5.241	0.024	688.2
4	3	7.381	0.030	734.6

Table 9.5. Decoding Speed of FPGA ASOVA Decoder versus Microprocessors

core. However, the function look-up-table, which is the kernel of the computation core, has 4-bit outputs. In Table 9.8, the results of ASOVA are based on a Turbo decoder with 5-bit soft information.

The power consumption of the Xilinx core was obtained from the Xilinx power

Architecture	Technology (μm)	Voltage (V)	Clock (MHz)
Stratix EP1S10	0.13	1.5	50
SA-1100	0.35	3.3	206
TMS320C6713	0.13	1.2	225

Table 9.6. Parameters of Stratix and DSPs

N_{max}	Decoding Speed (Kbps)			Power (mW)	
	EP1S10	SA1100	TMS320C6	EP1S10	SA1100
16	312.2	0.141	0.602	205.8	365.4
12	301.0	0.178	0.727	193.0	365.0
10	411.3	0.204	0.847	178.4	364.8
8	444.8	0.240	0.975	169.6	364.4
6	468.6	0.292	1.153	130.2	363.8
4	472.6	0.351	1.422	110.4	363.0

Table 9.7. Comparison to DSPs

Model	FPGA	Alg.	LUT	MEM EM	CLK (MHZ)	Power (mw)	Speed (bit/s)
Xilinx [109]	XC2V500	Log-MAP	2695×2	360K	66	970	2M
Altera [8]	EP1S10	Log-MAP	5644	400K	50	N/A	2M
ASOVA	EP1S10	ASOVA	2066	65K	76	248	1.35M

Table 9.8. Comparison Results of (15,13) Code

estimation tools [110] with an average activity rate of 30%. Note that the power consumed by the Xilinx core did not include the IO pads. The Xilinx core uses 2695 slices, where each slice has 2 LUTs. In terms of the LUT number, the dynamic ASOVA Turbo decoder is about 2 to 3 times smaller than the Xilinx core. The power consumption of Xilinx is about 8 times that of the dynamic ASOVA model.

9.2 Mapping ASOVA onto aSoC

After testing the ASOVA Turbo decoder in the FPGA board, the Turbo code system is mapped onto aSoC devices.

9.2.1 Partitioning of Functional Blocks

To run the Turbo decoder on the aSoC devices, the decoder must be partitioned into sub-tasks that can fit in aSoC cores. As introduced in Chapter 5, four types of cores are available in our aSoC model chips, namely, R4000, MEM, FPGA and

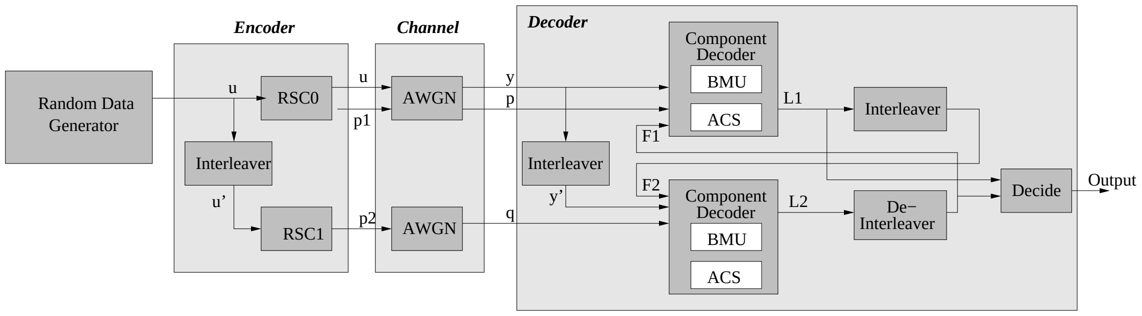


Figure 9.3. ASoC Partitioning of Turbo Decoder

MAC. In addition to matching the core functions, the partition should also allows for parallelism and pipelining.

To test the Turbo decoder, a Turbo encoder and a channel simulator were required in the system to provide the input data. The data rate of the encoder and the channel simulator should be no slower than the decoding speed when testing the decoder performance. Therefore, they also have to be integrated in the aSoC devices and carefully partitioned to provide enough input data rate.

The partitioning of the Turbo code system is shown in Figure 9.3, which includes a random data generator, a Turbo encoder, a channel simulator and a Turbo decoder. Each dark block in Figure 9.3 is mapped onto one aSoC core.

Random Data Generator generates information bit sequence u, which is the data to be sent to the receiver. It is a sequence of pseudo random bits of 0 or 1. Linear-feedback shift registers (LFSR) are employed to implement this random data generator. Since it is mainly bit wise operation, a FPGA core is used for the random data generator.

The Turbo encoder consists of three cores, two of which are encoders and the third is an *Interleaver*. The architecture of the RSC encoders described in Chapter 7 are implemented by two FPGA cores. The *Interleaver* functions to shuffle the order of the input data. The input data are buffered and output in a different order. A MEM core is used to accomplish this job.

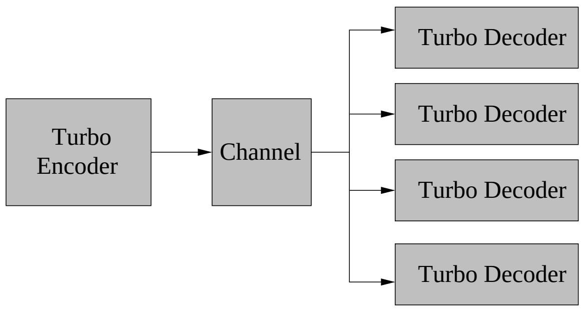


Figure 9.4. System with Four Turbo Decoders

The channel simulator functions as an additive white Gaussian noise (AWGN) channel. Each data that passes through the channel is added with a Gaussian noise. To allow for parallelization, two R4000 cores are used to perform the same function, with one on the data sequence of q, and the other one on sequences of y and p.

The Turbo decoder is implemented by three Interleavers and De-Interleavers, a decider and two component decoders. The Interleaver and De-Interleaver are built by MEM cores. An FPGA core is used for the decider. Each component decoder employs two cores. A MAC core works as the BMU and generate the branch metrics for each input, and a R4000 core takes care of the ACS and other computations since the aSoC FPGA core does not have enough hardware resources for this function. To sum up, two R4000 cores, one FPGA core, two MAC cores and three MEM cores are used for a Turbo decoder.

In order to further improve the decoding data rate, multiple Turbo decoders are mapped onto an aSoC chip. Figure 9.4 presents an example of a system with four parallel decoders. Each decoder works on separate code blocks. Given a block length of 1024 bits, the channel sends the first 1024 data to the first decoder, the second 1024 data to the second decoder, and so on. When all four decoders work parallelly, the decoding speed can be increased up to four times when compared to a system with only one decoder.

Table 9.9 presents the core requirements for the systems with various Turbo

Decoder #	Chip Size	R4000	FPGA	MEM	MAC
1	4×4	4	4	4	2
2	5×5	6	5	7	4
3	6×6	8	6	10	6
4	7×7	10	7	13	8

Table 9.9. Resources Usage of Turbo Decoder on aSoC

decoders. The first column is the number of decoders used in the system. The second column is the smaller chip size in terms of core number that can fit the Turbo decoder system. Other columns presents the number of used cores in each type.

9.2.2 Obtain aSoC Turbo Decoder Parameters

AppMapper is employed to map the Turbo decoder onto aSoC devices following the procedure described in Chapter 3.

A decoder for code (15,13) with constraint length $K = 4$ is chosen for the aSoC decoder. In all the experiments, a block size of 1024 bits and 6 iterations are used. The clock speed and the computation delay between communications are obtained from core simulation. The parameters of Decider, Interleaver and De-Interleaver are generated from the FPGA simulation. The part of ACS is simulated using SimpleScaler [21].

The effects of the log-normal shadowing is considered in the Turbo decoding system. For every 250,000 bits, the channel SNR is sampled, and the cores that handle the ACS decoding can select the appropriate N_{max} for the current SNR. A 2.5 billion information bit-sequence is tested in one experiment and the average decoding time for each 1024-bit code block is measured.

9.2.3 Comparison to FPGA Implementation

In this section, the performance and resource usage of the Turbo decoder on aSoC are compared with the FPGA implementation.

Table 9.10 presents the performance comparison between Turbo decoder using the Pentium IV micro-processor [26], the FPGA and aSoC. As mentioned in Section 9.2.1, multiple Turbo decoders can be mapped onto an aSoC device to improve the decoding rate. The row labeled with *Decoder Number* indicates the number of Turbo decoders in the system. In this experiment, a (31,27) code with constraint length of 5 was used. Similar results were obtained for the codes with other constraint length and not listed in the table.

It can be seen that the FPGA implementation has the highest decoding speed within the systems consisting one decoder. This speedup comes from the specialization and the parallelism of the FPGA implementation. The Verilog hardware design is customized for the FPGA. As described in Chapter 8, the architecture of the component decoders are specially designed to fit the Stratix FPGA resources to achieve the best performance. For example, the DSP blocks in the FPGA are employed for the BMU, and the memory in the SMU is divided into banks to make use of the SRAM blocks. In addition, the parameters used in the FPGA implementation are optimized to 4-bit values, which speeds up the computation. The general purpose processors such as the R4000 in aSoC and the Pentium IV have to use the given bitwidth of 32-bit or 16-bit, whether it is needed or not.

On the other hand, the FPGA implementation is fully parallel. An FPGA device has numerous computing resources which operate at the same time. With these distributed units, the FPGA implementation of the Turbo decoder can achieve a higher decoding speed. The general purpose processors have a centralized controller, a limited number of ALUs and share the same memory. All these features force the instructions to be executed sequentially. As a result, the decoding speed is

N_{max}	Pentium IV (Kbps)	FPGA (Kbps)	aSoC (Kbps)		
Decoder Number	1	1	1	2	3
16	1.589	312.2	197.1	287.2	337.8
12	2.048	301.0	223.5	327.5	380.9
10	2.532	398.2	238.4	352.9	409.2
8	3.013	444.8	256.3	377.0	435.6
6	4.160	468.6	322.5	476.7	552.8
4	5.824	472.6	436.2	648.3	708.9

Table 9.10. Decoding Speed of aSoC and FPGA Implementation on (31,27) Code

much slower than the FPGA implementation. The aSoC devices employ multiple heterogeneous IP cores, which allows more parallelism than a single processor and brings the performance closer to the FPGA implementation.

The speedup of FPGA implementation comes at the cost of larger resource usage. Since these devices are designed in different technologies, die size does not provide a fair comparison. As a result, the transistor count is used in the following comparison. Table 9.11 shows the transistor count of the Pentium IV, Stratix EP1S10 FPGA and the aSoC components used by the Turbo decoder. Pentium IV has about 42M transistors [26]. The exact transistor count of Stratix EP1S10 is unavailable from the public documents, but can be estimated closely with the algorithm in [17]. The Stratix EP1S10 FPGA consists of 10,570 logic elements (LEs) and about 1M bit of SRAM [7]. Each LE has about 900 transistors [16] and one memory bit requires 6 transistors. As a result, the number of overall transistors can be calculated as Eq 9.1.

$$900 \times 10,570 + 1M \times 6 = 15.5M \tag{9.1}$$

This number includes only the memory blocks and LEs. The other resources, such as the interconnect, IO pads and DSP blocks, are not included in this estimation. The above computation gives only a conservative number, and an EP1S10 FPGA has more than 15.5M transistors.

Architecture	Transistor Count
Stratix EP1S10	>15.5M
Turbo decoder on aSoC	11.2M
Pentium IV	42M [26]

Table 9.11. Transistor Count Comparison of Different Approaches

Core Type	Transistor Count
R4000	1M [27]
MEM	1.3M
MAC	100K
FPGA	1.1M
Communication Interface	500K

Table 9.12. Transistor Count of aSoC Cores

The transistor count of aSoC can be obtained by summing up the transistors of the IP cores and the communication interface. Table 9.12 presents the transistor count of the aSoC components. The data for the cores of MEM, MAC, FPGA and the aSoC communication interface are obtained from the simulation and synthesis. As mentioned previously in Section 9.2.1, a Turbo decoder employs two R4000 cores, one FPGA core, two MAC cores and three MEM cores. The transistor count sums up to 11.2M as shown in Eq. 9.2.

$$1M \times 2 + 1.1M + 100K \times 2 + 1.3M \times 3 + 500K \times 8 = 11.2M \tag{9.2}$$

The transistor count comparison is shown in Table 9.11. The Pentium IV has the largest transistor count because it consists a lot of units unnecessary to the Turbo decoder, for example, the graphic processing ability.

The performance and area comparison results mainly depend on the computation resources but not the communication architecture. To evaluate the aSoC communication architecture, a fair comparison system has to be established to eliminate the effect of the computation cores. This system is presented in more detail in Chapter 10.

Chapter 10

aSoC simulation results

To evaluate the benefits of aSoC versus other on-chip communication technologies, design mapping, simulation, and layout were performed. Benchmark simulation of 9 and 16 core models shown in Figure 5.1 were used to determine architectural parameters for aSoC prototype layout. Core model assumptions were subsequently validated via layout. As a final step, aSoC implementations of the benchmarks were compared to implementations using alternative on-chip interconnect approaches.

10.1 aSoC Parameter Evaluation and Layout

The benchmarks described in Chapter 6 and the Turbo decoding system introduced in Chapter 9 were evaluated using the aSoC simulator to determine aSoC parameters such as the required number of instructions per instruction memory. The cores listed in Table 5.2 were used in configurations described in Chapter 5. R4000 performance and area were obtained from MIPs [27]. Multiply-accumulate, memory, and FPGA performance numbers were determined through core layout using TSMC 0.18um library parameters [90].

Benchmark run-time statistics determined via simulation are summarized in Table 10.1. These statistics illustrate usage of various CI resources across a set of applications. The values were determined with parameters set to values which led to best-performance application mapping. Statistics which were used for CI architectural choices are highlighted in boldface. Although the maximum number of instructions per CI was relatively small for these designs (9), a depth of 32 was allocated in the aSoC prototype to accommodate expansion for future applications.

Design	Cores Number	Streams Number	**Max. CI Inst.**	**Max. Str. per CI**	Ave. Str. per CI	**Max. Coreport Depth**	Ave. Coreport Depth
IIR	9	11	2	5	3.5	2	2.0
IIR	16	20	2	5	3.5	4	2.7
IMG	9	8	2	3	2.0	2	1.5
IMG	16	15	4	4	3.5	4	1.9
IMG	25	20	4	7	2.0	4	2.1
IMG	36	28	4	7	1.8	4	1.5
IMG	49	36	4	7	2.6	4	1.5
Doppler	16	32	8	6	2.1	4	1.6
OFDM	16	39	9	6	2.2	4	1.5
MPEG	16	19	4	8	3.6	4	2.3
MPEG	25	37	5	8	3.2	4	3.0
MPEG	36	55	5	8	2.1	4	3.1
MPEG	49	73	5	8	5.3	4	3.0

Table 10.1. Benchmark statistics used to determine aSoC parameters

Since the maximum total number of streams per CI is 8, each of the four CDM buffers per CI could be restricted to a depth of 2 in the prototype. The coreport memory depth was set to four, the maximum value in terms of streams across all benchmarks.

A prototype SoC device, including aSoC interconnect, was designed and implemented based on experimentally-determined parameters. The 9 tile device layout in a 3×3 core configuration contains lookup-table (LUT) based FPGA cores with 121 clusters of 4 four-input LUTs, a complete communication interface, and clock and power distribution. Each tile fits a size of $30,000 \times 30,000\lambda^2$ with $2,500 \times 3,500\lambda^2$ assigned to the communication interface and associated control and clock circuitry (about 6% of device area). An *H-tree* clock distribution network is used to reduce clock skew between tiles. Layout was implemented using TSMC 0.18um library parameters resulting in a communication clock speed of 400MHz. The critical path of 2.5 ns in the communication interface involves the transfer of a flow control bit

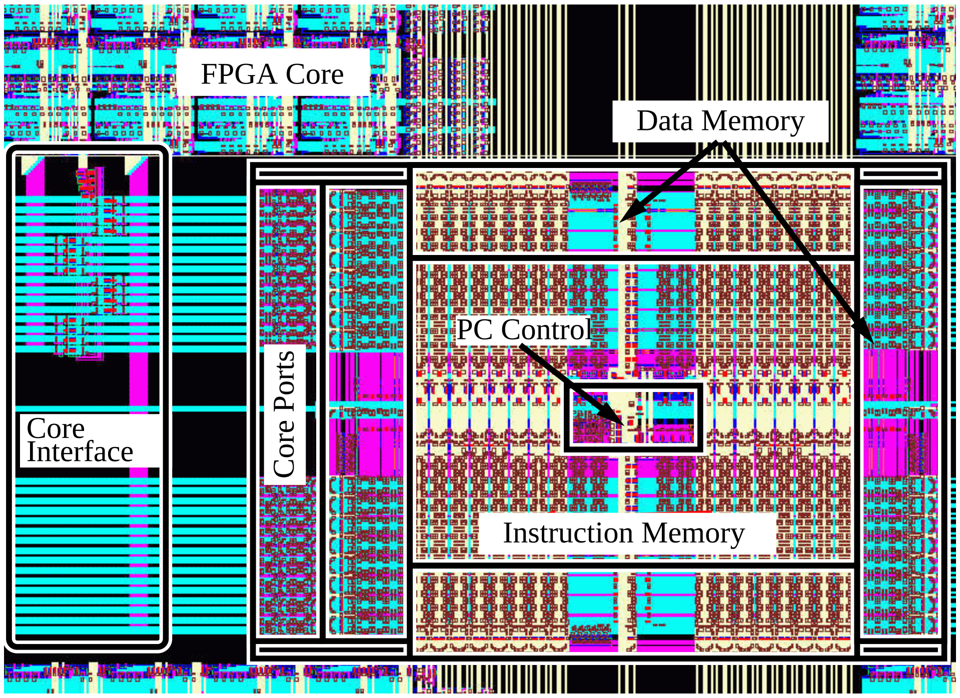


Figure 10.1. Layout of FPGA Core and Communication Interface

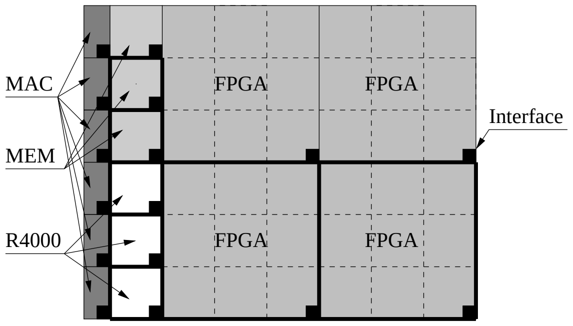


Figure 10.2. Non-uniform aSoC core configuration

from a CDM buffer to the read control circuitry of a neighboring CDM buffer, as shown in the right-to-left path in Figure 3.8. A layout snapshot of a communication interface, coreport, and a single FPGA cluster appears in Figure 10.1.

The layouts of the communication interface and associated cores support the creation of a non-uniform mesh structure which is populated to optimize space consumption. As shown in Figure 10.2, tile sizes range from $10,000 \times 10,000\lambda^2$ to

$30,000 \times 30,000\lambda^2$. The overhead percentage of the communication interface for each core is shown in Table 5.2. For comparison, an embedded Nios processor core [1] and its associated AMBA bus interface [39] were synthesized. A total of 206 out of 2904 total logic cells (7%) were required for the AMBA interface, with additional area required for bus wiring. This result indicates that the aSoC communication interface is competitive with on-chip bus architectures in terms of core overhead.

10.2 Performance Comparison with Alternative On-Chip Interconnects

A series of experiments were performed to compare aSoC performance against three alternative on-chip communication architectures: a standard CoreConnect on-chip bus [54], a hierarchical CoreConnect bus, and a hypothetical network based on run-time dynamic routing [30]. Performance was evaluated using the aSoC simulator described in Section 5.2. In these experiments, the IP cores with parameters shown in Table 5.2 were aligned in the 9 and 16 configurations shown in Figure 5.1. For each interconnect approach, the relative placement of cores and application partitioning was kept intact. Only the communication architecture which connects them together was changed for comparative results.

10.2.1 Comparison to CoreConnect Architecture

To evaluate aSoC bandwidth capabilities, a benchmark-based comparison is made for aSoC versus the IBM CoreConnect processor local bus (PLB) [54]. The PLB bus architecture allows for simultaneous 32-bit read and write operations at 133 MHz. When necessary, bus arbitration is overlapped with data transfer. The architecture requires two cycles to complete data transfer: one cycle to submit the address and a second cycle to transport the data. CoreConnect PLB supports burst transfers up to 16 words. Maximum possible speedup for burst transfer versus multiple single-word transfer is about 2X.

	9-Core Model		16-Core Model					
Execution Time (mS)	IIR	IMG	IIR	IMG	MPEG	Doppler	OFDM	Turbo
R4000	0.049	327.0	0.350	327	152	0.80	4.40	109
CoreConnect	0.012	22.0	0.016	30.5	173	0.13	0.21	0.323
CoreConnect (burst)	0.012	18.9	0.015	24.3	172	0.13	0.21	0.323
aSoC	0.006	9.6	0.006	7.3	83	0.11	0.18	0.102
aSoC Speedup vs. burst	2.0	2.3	2.5	3.3	2.1	1.2	1.2	3.2
Used aSoC links	8	8	33	27	41	26	45	33
aSoC max. link usage	10%	8%	37%	28%	25%	2%	4%	15%
aSoC ave. link usage	7%	7%	22%	25%	5%	2%	3%	8%
CoreConnect busy (burst)	91%	100%	100%	99%	67%	32%	37%	72%

Table 10.2. Comparison of aSoC and CoreConnect Performance

It can be seen in Table 10.2 that aSoC performance improvement over CoreConnect increases with a larger number of cores. Run times on a single 200 MHz R4000 are provided for reference. Relative aSoC improvement over CoreConnect burst transfer is indicated in the row labeled **aSoC speedup**. For most designs the CoreConnect implementation leads to saturated or nearly-saturated bus usage. The *CoreConnect busy* row indicates the fraction of total possible bandwidth used by the CoreConnect bus during the application.

10.2.2 Comparison to Hierarchical CoreConnect

A limiting factor with shared on-chip buses is scalability. To provide a fairer comparison to aSoC, a set of experiments was performed using a hierarchical version of the CoreConnect bus. Three separate CoreConnect PLBs connect **rows** of cores shown in Figure 5.1. A CoreConnect OPB bridge [54] joins three subbuses (for

	9-Core Model		16-Core Model				
Execution Time (mS)	IIR	IMG	IIR	IMG	MPEG	Doppler	OFDM
Hier. CoreConnect	0.013	26.0	15.7	37.4	178	0.15	0.22
aSoC	0.006	9.6	7.0	7.3	83	0.11	0.18
aSoC Speed-up	2.1	2.7	2.2	5.1	2.2	1.4	1.2
subbus 0 busy	85%	97%	99%	100%	94%	30%	30%
subbus 1 busy	72%	83%	99%	61%	94%	12%	27%
OPB bridge busy	40%	65%	81%	60%	93%	16%	36%

Table 10.3. Comparison of aSoC and Hierarchical CoreConnect Performance

9 cores) or four subbuses (for 16 cores). When a cross-subbus transfer request is made, the OPB bridge serves as a bus slave on the source subbus and a master for the sink subbus. The OPB bridge arbitrates for control of the sink subbus before acknowledging the source subbus transaction. When both subbuses are idle, a cross-subbus transaction can be set up within two bus clock cycles. As shown in Table 10.3, for all but one design, aSoC speedup versus the hierarchical CoreConnect bus is larger than speedup versus the standard CoreConnect bus. This effect is due to the overhead of setting up cross-bus data transfer.

10.2.3 Comparison to Oblivious Dynamic Routing

In a third set of experiments, the aSoC interconnect approach was compared to a hypothetical on-chip dynamic routing approach. This dynamic routing model applies oblivious dynamic routing [30] with one 400 MHz router allocated per IP core. Tile topology for the near-neighbor dynamic network is the same as shown in Figure 5.1. For each transmitted piece of data, a header indicating the coordinate of the target node is injected into the network, followed by up to 20 data packets. Dimension-order routing, which routes packets horizontally and then vertically, is used to prevent routing deadlock. To allow for a fair comparison to aSoC flow

	9-Core Model		16-Core Model				
Execution Time (mS)	IIR	IMG	IIR	IMG	MPEG	OFDM	Turbo
Dynamic Routing	0.008	14.4	8.7	9.7	162.0	0.19	0.154
aSoC	0.006	6.1	7.0	7.3	82.5	0.18	0.102
aSoC Speedup	1.3	2.4	1.3	1.3	2.0	1.1	1.5

Table 10.4. Comparison of aSoC and Dynamic Network Performance

control, the routing buffer in each dynamic router is set to be the maximum size required by an application.

The results in Table 10.4 indicate that the aSoC is up to 2.4 times faster than the dynamic model. Performance improvements are based on the removal of header processing and a tight aSoC coreport-router interaction.

10.2.4 Comparison to Published Results

Several experiments were performed to compare the results of aSoC interconnect versus previously-published on-chip interconnect results. An MPEG-2 decoder, developed from four Motorola PowerPC 750 cores interconnected with a CoreConnect bus, was reported in [88]. The four 83 MHz *compute nodes* require communication arbitration and contain an associated on-chip data and instruction cache. During decoding, frames of 16×16 pixels are distributed to all processors and results are collected by a single processor. To provide a fair performance comparison to this MPEG-2 decoder, our aSoC simulator was supplemented with SimpleScalar 3.0 for PowerPC [21] and applied to four PowerPC 750 core tiles interconnected with communication interfaces. The partitioning of computation was derived from [88], following consultation with the authors. In the experiment, the PowerPC cores run at 83 MHz and the aSoC communication network runs at its 400 MHz. Table 10.5 compares our results to previously published work. Unlike the 64-bit, 133 MHz CoreConnect model [88], aSoC avoids communication congestion by avoiding arbitration and providing a faster transfer rate (32 bits at 400 MHz).

MPEG-2 Decoder	Throughput (Mbps)
CoreConnect [88]	0.68
aSoC	2.88
OFDM	Throughput (Mbps)
CoreConnect [89]	2.19
aSoC	5.67

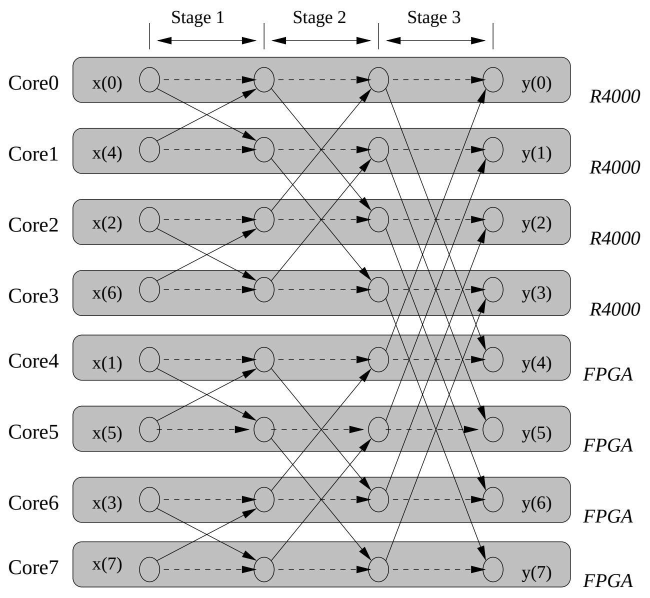
Table 10.5. Comparison to published work

In previously published work [89], OFDM was also implemented using four 83 MHz PowerPC 755 cores interconnected with a 64-bit, 133 MHz CoreConnect bus. Each packet of OFDM data contains a 2048-complex valued sample and a 512-complex valued guard signal. This application was partitioned into four stages: initiation, inverse FFT, normalization and guard signal insertion. Each stage was mapped onto a separate processor core. Like the MPEG-2 decoder described above, the same mapping of computation to 83 MHz PowerPC 755 cores was applied to aSoC and modeled using SimpleScalar and aSoC interconnect simulators. Results are shown in Table 10.5. The aSoC implementation achieves improved performance for this application by providing high bandwidth and pipelined transfer.

Unlike the results for MPEG-2 and OFDM shown in Tables 10.1 through 10.4, communication is not overlapped with computation during execution of the applications. This approach is consistent with the method used to obtain the previously-published results [88, 89].

10.3 Run-time Communication Branching

As discussed in Chapter 3.1.3, the conditional branch mechanism of the communication interface provides the capability of modifying aSoC communication patterns at run time. This is especially useful for algorithms with changing communication patterns, such as FFT. All source-destination paths for the computation are known at compile time and switches between communication patterns are coordinated by



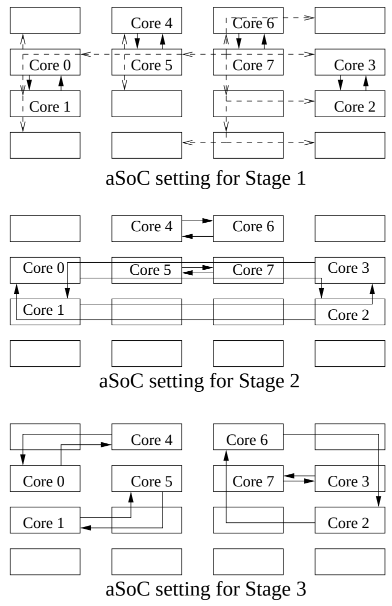


Figure 10.3. Mapping Result of FFT

Instruction Number	interface connection	next instruction	possible branch?	Comment
0x0	core to south	0x1/-	N	*data transfer south*
0x1	core to I_{out}	0x2/0x0	Y	*test control value*
0x2	core to east	0x3/-	N	*data transfer east*
0x3	core to I_{out}	0x4/0x2	Y	*test control value*
0x4	core to west	0x5/-	N	*data transfer west*
0x5	core to I_{out}	0x0/0x4	Y	*test control value*

Table 10.6. Control branching transfer example

a core-generated *count* value. When the *count* value reaches 0 and is transferred to the I_{out} port of the communication interface (shown in Figure 3.5) and a switch in communication patterns is performed.

A FFT kernel is mapped onto aSoC devices following the partition shown in the left part of Figure 10.3. As shown in Table 10.7, a number of multi-point FFTs were implemented to illustrate control branching. A single core performs all butterfly computations in a shaded row in the left portion of Figure 10.3. To

N	8	16	32	64	128	256	512	1024
R4000	23.0	34.0	50.0	89.0	176.0	263.0	792	1900
aSoC	0.4	0.8	1.7	4.7	14.6	38.7	94	230
CoreConnect	0.4	1.0	2.7	5.4	24.4	52.0	120	314

Table 10.7. FFT application run time for N points (times in uS)

complete required transfers, three communication patterns are scheduled for the three stages of FFT computation. The data streams scheduled for different stages are shown in the right part of Figure 10.3. As an example, the interconnect memory program of Core 4 in the 16 tile topology in Figure 10.3 is shown in Table 10.6. These instructions illustrate the transfer of a control value from the local core to port I_{out} on alternate instructions. When the control value reaches 0, control is switched to the subsequent communication instructions. This occurs when all data for the current phase (e.g. 8 points for a 64 point FFT) have been communicated.

Execution time results of the FFT application using CoreConnect and aSoC interconnect approaches are shown in Table 10.7. The benefits of aSoC over CoreConnect are due to the elimination of bus arbitration.

10.4 Architectural Scalability

An important aspect of a communication architecture is scalability. For interconnect architectures, a scalable interconnect can be defined as one that provides scalable bandwidth with reasonable (e.g. linear) latency increase as the number of processing nodes increase and as the computing problem size increases [28]. Under this definition, aSoC provides scalable bandwidth for many applications, including MPEG-2 encoding, Turbo decoding, image smoothing and OFDM application.

The MPEG-2 encoder in Figure 6.1 can be scaled by replicating core functionality, allowing for multiple frames to be simultaneously processed in separate threads. A bottleneck of this approach is a common *Input Buffer* and data collection

Threads	Core Configuration	Used Cores	Comm. Cycles	Throughput	
				pixel/uS	Mbps
1	4×4	12	33,002,480	0.60	7.15
2	5×5	23	34,906,796	1.13	13.51
3	6×6	34	34,916,246	1.69	20.27
4	7×7	45	35,311,402	2.23	26.72

Table 10.8. Scalability of the MPEG2 Encoder

buffer at the input and output of the encoder. Since the communication delay of distributing the data to threads can be overlapped with computation, communication congestion and data buffer contention can lead to performance degradation as design size scales. Table 10.8 illustrates scalable performance improvement for multiple MPEG-2 threads implemented on aSoC. Device sizes ranging between 16 and 49 cores were considered. Total communication cycles increased marginally to accommodate routing and *Input Buffer* contention.

Similar to the scalable MPEG II encoders, multiple Turbo decoders can be used in a decoding system to improve the decoding speed. As described in Chapter 9, when the sequence of 1024-bit code blocks are received from the channel, each code block will be assigned to a idle Turbo decoder. All Turbo decoders in a system can be working parallelly. In this experiment, up to 4 Turbo decoders are used in a system. Appropriate number and types of cores, as shown in Table 9.9, are chosen for the target aSoC devices. With different Turbo decoder numbers, the sizes of these devices range from 16 to 49 cores. Table 10.9 presents the scalable performance on aSoC devices. The decoding date rate of aSoC devices are compared to the system using CoreConnect bus, and the devices using dynamic network. With the number of decoders in a system increases, the aSoC devices and the model using dynamic network provides increasing decoding data rate. However, due to the congestion and bus arbitration overhead, more decoders do not help to achieve better performance

Device	16-core	25-core	36-core	49-core
Decoder Number	1	2	3	4
Used Core	14	22	30	38
aSoC Speed (Mbps)	1.66	2.51	2.89	3.28
CoreConnect Speed (Mbps)	0.529	0.689	0.615	0.602
Dyn. Speed (Mbps)	1.12	1.49	1.86	1.93
aSoC vs. CoreConnect	3.2	3.6	4.7	5.4

Table 10.9. Scalability of Turbo Decoder

Slices	Core Configuration	Used Cores	Execution Time (mS)
2	3×3	8	9.61
4	4×4	14	7.27
6	5×5	20	4.84
9	7×7	38	4.75

Table 10.10. Scalability of image smoothing for 800x600 pixel image

for the model using CoreConnect architecture. It can be seen that, as the number of Turbo decoders increases in a system, the aSoC devices obtains increasing speed-up over the CoreConnect bus architecture. It indicates that aSoC can achieve better performance over bus architecture when the size of the system gets large and the communication amount increases.

Using a similar multiprocessing technique, the image smoothing application was parallelized across a scaled number of cores using multiple threads applied to a fixed image size. Each 3-pixel high slice is handled by an R4000, a MAC and an FPGA. Table 10.10 illustrates the scalability of the application across multiple simultaneously-processed slices. The image source and destination storage buffers are shared across slices. Application execution time scales down with increased core count until contention inside the storage buffers eliminates further improvement.

In a final demonstration of architectural scalability, a number of multi-point Doppler evaluations were implemented on a 16 core aSoC model. Execution time

N	8	16	32	64	128	256	512	1024
R4000	23.0	34.0	50.0	89.0	176.0	263	792	1900
CoreConnect	0.68	1.6	3.7	7.4	28.2	60	130	340
aSoC	0.65	1.3	2.7	6.6	18.3	46	110	260

Table 10.11. Doppler run time for N points (times in uS)

results of the Doppler application using CoreConnect and aSoC interconnect approaches are shown in Table 10.11. The benefits of aSoC over CoreConnect are due to the elimination of bus arbitration.

Chapter 11

Conclusion and Future Work

This dissertation solves the problem of the scalable on-chip communication bandwidth requirement for near future System-on-a-Chip devices. Currently used bus architectures rely on centralized arbitration to dynamically choose the bus master, which restricts the scalability of the communication architecture. In addition, long bus wires limit the clock speed and consume a great deal of power to drive all cores on the bus. In this dissertation, a new communication substrate, the adaptive System-on-a-Chip (aSoC), for on-chip communication has been presented. There are four main contributions of this dissertation:

1. A scalable communication architecture has been constructed as shown in Chapter 3. Several features have been employed in the architecture to improve the data bandwidth and performance. The communication interface has been designed to support the packet-switching scheme so that centralized bus arbitration is avoided. The use of scheduled communication allows for predictable data transfer at a fast rate. Its distributed nature allows for scalable bandwidth. Near-neighboring pipelined wires are used to connect the communication interfaces, allowing for fast clock frequency. Although the architecture has been optimized for stream-based communication to allow for predictable data transfer at a fast rate, dynamic behavior is supported through communication control branching. The distributed nature of the interconnect allows for scalable bandwidth.

   The developed communication architecture includes the communication interfaces which forward the data packets to their destinations based on the prescheduled instructions, the pipelined near-neighboring wires, and the core-port which synchronizes the computation cores and the communication network.

2. Supporting mapping tools have been developed to aid in design translation to aSoC devices. The tools include an aSoC compiler and a system simulator. The compiler accepts high-level design representations, isolates basic blocks of code, and assigns blocks to specific cores. Data transfer times between cores are determined through heuristic scheduling. An integrated core and interconnect simulation environment allows for accurate system modeling prior to device fabrication.

3. A set of benchmarks have been developed for aSoC devices. The applications are chosen from the DSP, multi-media and communication domains to verify the performance of aSoC architecture, including the MPEG II encoder/decoder, image smoothing, Doppler radar, OFDM, and IIR filter. While the above applications are mostly DSP cores, a Turbo decoder is also mapped onto aSoC devices as an integral system. A novel adaptive Soft-output Viterbi algorithm is developed to improve the decoding data rate and reduce the computation complexity.

4. Experimentation was performed to validate the performance of an aSoC architecture. A nine-core prototype aSOC chip including both FPGA cores and associated communication interfaces was designed and constructed. The point-to-point nature of our architecture supports a communication clock speed of 400 MHz in 0.18 micron technology. The six application circuits have been tested on the aSoC devices and compared with alternative communication architectures. It was found that the aSOC interconnect approach outperforms the standard IBM CoreConnect on-chip bus protocol by up to a factor of

five and compares favorably to previously-published work. Experiments on the Turbo decoding system reveals increasing speedup over the CoreConnect bus architecture on aSoC devices with more cores, which indicates the superb scalability of aSoC on communication applications.

The current status of this dissertation has verified the performances of aSoC communication architecture. It provides a communication substrate for SoC designs. The concept of pipelined packet-switching on-chip communication has been justified by carefully chosen applications covering the communication, multi-media and DSP domains. In addition, the dissertation also opens some possible directions for future research interests.

- While the speed performance of aSoC has been validated in this dissertation, the power consumption, which is another critical issue for deep sub-micron devices, has not been addressed. The aSoC, as an on-chip communication substrate, provides not only the inter-core communications but also the clock tree for local computation cores. The highly active clock tree and long wire data communications consume a considerable portion of the power in modern devices. Further research can be done to reduce power consumption for on-chip communication so that a low-power aSoC architecture can be developed.

- Software tools that aid the mapping of applications onto aSoC are described in Chapter 4. The compiler is designed for high level language descriptions, which were created for sequential executions. New programming languages which can exploit the parallelism will be desirable for aSoC devices. The use of stream-based programming languages for aSoC also provides an opportunity for further investigation.

- Although the aSoC architecture is optimized for pre-scheduled stream-based architecture, dynamic routing is still a desirable feature for many applications.

The addition of some dynamic routing hardware to the communication interface will improve the flexibility of aSoC and extend the applications to a more flexible domain.

References

[1] *www.altera.com.* Altera Corporation, 2000.

[2] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. ONeill, J. Othmer, E. Sackinger, K. Singh, J. Sweet, C. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. *Journal of Solid-State Circuits*, 35(3), Mar. 2000.

[3] L. Adams. OverView of the CoreFrame Architecture. In *PalmChip Corporation white paper*, 2000. http://www.palmchip.com/coreframe.html.

[4] P. J. Aldworth. System-on-a-Chip Bus Architecture for Embedded Applications. In *IEEE International Conference on Computer Design*, Austin, Texas, Oct. 1999.

[5] Altera, Inc. Nios 3.0 cpu data sheet. In *http://www.altera.com/literature/ds/ds_nioscpu.pdf*, 2002.

[6] Altera, Inc. Nios development kit, stratix edition. In *http://www.altera.com/products/devkits/altera/kit-nios_1S10.html*, 2002.

[7] Altera, Inc. Stratix data sheet. In *http://www.altera.com/literature/ds/ds_stx.pdf*, Dec. 2002.

[8] Altera, Inc. MegaCore Function User Guide Turbo Encoder/Decoder. In *http://www.altera.com/literature/ug/turbo_ug.pdf*, 2003.

[9] J. Babb, M. Rinard, C. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications to Silicon. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1999.

[10] J. Babb, R. Tessier, M. Dahl, S. Hanono, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 609–626, June 1997.

[11] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate. *IEEE Transactions on Information Theory*, pages 284–287, Mar. 1974.

[12] L. Benini and G. D. Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1), Jan. 2002.

[13] R. Bergamaschi and W. Lee. Desiging systems-on-chip using cores. In *Proceedings, 37th Design Automation Conference*, Los Angeles, CA, USA, Jun. 2000.

[14] C. Berrou, P. Combelles, and B. Talibart. An IC for Turbo-Codes Encoding and Decoding. In *IEEE International Solid-State Circuits Conference*, pages 90–91, Feb. 1995.

[15] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. In *Proceedings of ICC'93*, pages 1064–1070, Geneve, Switzerland, May 1993.

[16] V. Betz and J. Rose. Directional Bias and Non-Uniformity in FPGA Global Routing Architectures. In *ICCAD*, San Jose, Ca, 1996.

[17] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *Custom Integrated Circuits Conference*, 1997.

[18] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers, Norwell, MA, 1999.

[19] P. Black and T. Meng. A 1-gb/s, four-state, sliding block viterbi decoder. *IEEE Journal of Solid-States Circuits*, 32(6), Jun 1997.

[20] S. Borkar. Supporting Systolic and Memory Communication in iWarp. In *Proceedings 17th International Symposium on Computer Architecture*, 1990.

[21] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin, Madison Computer Science Department*, June 1997. Technical Report 1342.

[22] F. Chan. Adaptive viterbi decoding of turbo codes with short frames. In *Communication Theory Mini-Conference*, pages 47–51, June 1999.

[23] F. Chan and D. Haccoun. Adaptive viterbi decoding of convolutional codes over memoryless channels. *IEEE Transactions on Communications*, 45(11):1389–1400, Nov. 1997.

[24] D. Cherepacha and D. Lewis. A datapath oriented architecture for fpgas. In *Proceedings, ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, Feb 1994.

[25] G. C. Clark and J. B. Cain. *Error-Correction Coding for Digital Communications.* Plenum Pub Corp., New York, 1981.

[26] I. Corporation. 2000: Intel Pentium 4 Processor. In *http://www.intel.com/intel/intelis/museum/Exhibits/hist_micro/hof/index.htm*, 2000.

[27] M. Corporation. *MIPS R4000 web page.* MIPS Corporation, 2000. http://www.mips.com/products/s2p6.html.

[28] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, San Francisco, CA, 1999.

[29] I. Cypress Semiconductor. Cyprss MicroSystems Unveils Programmable System-on-a-chip For Emgedded Internet, Communications and consumer Sysstems. In *http://www.cypress.com*, 2000.

[30] W. Dally and H. Aoki. Deadlock-free Adaptive Routing in Multicomputer Networks using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4), April 1993.

[31] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings, ACM/IEEE 38th Design Automation Conference*, June 2001.

[32] J. Darringer, R. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J. Morrell, I. Nair, P. Sagmeister, and Y. Shin. Early analysis tools for system-on-a-chip design. *IBM Journal of Research and Development*, 46(6), Nov. 2002.

[33] G. DeMicheli and R. Gupta. Hardware/Software Codesign. *Proceedings of the IEEE*, 85(3):349–365, Mar. 1997.

[34] C. Ebeling, D. C. Cronquist, and P. Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings, Sixth International Workshop on Field Programmable Logic and Applications*, Darmstadt, Germany, Sep. 1996.

[35] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems*, 2(1):5–32, Jan. 1997.

[36] J. A. Erfanian, S. Pasupathy, and G. Gulak. Reduced complexity symbol detectors with parallel structures for isi channels. *IEEE Transactions on Communications*, 42:1661–1671, Feb/Mar/Apr 1994.

[37] R. Ernst, J. Henkel, and T. Benner. Hardware Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, Dec. 1993.

[38] G. Feygin and P. G. Gulak. Architectural tradeoffs for survivor sequence memory management in viterbi decoder. *IEEE Transactions on Communications*, 41(3), March 1993.

[39] D. Flynn. AMBA: Enabling Reusable On-Chip Design. *IEEE Micro*, pages 20–27, July 1997.

[40] M. P. C. Fossorier, F. Burkert, S. Lin, and J. Hagenauer. On the Equivalence Between SOVA and Max-Log-MAP Decodings. *IEEE Communications Letters*, 2(5):137–139, May 1998.

[41] V. Franz and J. B. Anderson. Concatenated Decoding with a Reduced-Search BCJR Algorithm. *IEEE Journal on Selected Areas on Communication*, 16:186–195, Feb. 1998.

[42] D. Garrett and M. Stan. Low power architecture of the soft-output viterbi algorithm. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED'98)*, pages 262–267, Monterey, California, Aug. 1998.

[43] M. Ghanbari. *Video Coding.* The Institution of Electrical Engineers, London, England, 1999.

[44] F. Gilbert, A. Worm, and N. Wehn. Low power implementation of a turbo-decoder on programmable architectures. In *Proc. 2001 Asia South Pacific Design Automation Conference (ASP-DAC 01)*, pages 400–403, Yokohama, Japan, Jan. 2001.

[45] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2002.

[46] J. Hagenauer. Source-controlled channel decoding. *IEEE Transactions on Communications*, 43(9):2449–2457, Sept. 1995.

[47] J. Hagenauer and P. Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *Proceedings of GLOBECOM'89*, pages 47.1.1–47.1.7, Dallas, Texas, Nov. 1989.

[48] J. Hagenauer, E. Offer, and L. Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445, Mar. 1996.

[49] S. Halter, M. berg, P. M. Chau, and P. H. Siegel. Reconfigurable signal processor for channel coding & decoding in low SNR wireless communications. In *Proc. IEEE Workshop on Signal Processing Systems (SiPS'98)*, pages 260–274, Cambridge, Oct. 1998.

[50] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architercture for billion transistor era. In *Proceedings, IEEE NorChip Conference*, Finland, Nov. 2000.

[51] S. Hong, J. Yi, and W. E. Stark. VLSI design and implementation of low-complexity adaptive turbo-code encoder and decoder for wireless mobile communication applications. In *IEEE Workshops on Signal Processing Systems: Design and Implementation*, pages 233–242, Oct. 1998.

[52] IDT, Incorporated. IDT Peripheral Bus: Intermodule Connection Technology Enables Broad Range of System-Level Integration. In *http://www.idt.com*, 2000.

[53] Intel, Inc. Intel StrongARM SA-1100 Microprocessor. In *http://www.lart.tudelft.nl/27810525.pdf*, 2000.

[54] International Business Machines, Inc. IBM CoreConnect Information Web Site. In *http://www.chips.ibm.com/products/powerpc/cores*, 2000.

[55] International Technology Roadmap Semiconductors. *The International Technology Roadmap Semiconductors: 2002 Update.* International SEMATECH, 2002. http://public.itrs.net.

[56] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings, IEEE/ACM International Conference on Computer Aided Design*, San Jose, CA, USA, Nov. 2002.

[57] O. J. Joerssen, M. Vaupel, and H. Meyr. Soft-output Viterbi decoding: VLSI implementation issues. In *Proceedings of IEEE Vehicular Technology Conference*, pages 941–944, Secaucus, NJ, May 1993.

[58] G. D. F. Jr. The viterbi algorithm. In *Proceedings of IEEE*, pages 268–278, col. 61, Mar. 1973.

[59] J. Kaza and C. Chakrabarti. Energy-efficient turbo decoder. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2002.

[60] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), Dec. 2000.

[61] D. Kim and G. Stuber. Performance of Multiresolution OFDM on Frequency-selective Fading Channels. *IEEE Transactions on Vehicular Technology*, 48(5):1740–1746, 1999.

[62] P. Knudsen and J. Madsen. Integrating Communication Protocol Selection with Hardware/Software Codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8), Aug. 1999.

[63] K. Lahiri, A. Raghunathan, and S. Dey. Performance Analysis of Systems With Multi-Channel Communication. In *International Conference on VLSI Design*, Jan. 2000.

[64] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on RAW Machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Language and Operating Sustems*, Oct. 1998.

[65] O. Y.-H. Leung, C.-W. Yue, and C.-Y. Tsui. Reducing power consumption of turbo code decoder using adaptive iteration with variable supply voltage. In *ISLPED*, pages 36–41, San Diego, CA, 1999.

[66] S.-F. Li, M. Wan, and J. Rabaey. A Low-Energy Heterogeneous Reconfigurable DSP IC. In *Proceedings 1999 IEEE Workshop on Signal Processing Systems*, Taipei, Taiwan, Oct. 1999.

[67] J. Liang, S. Swaminathan, and R. Tessier. aSOC: A Scalable, Single-Chip Communications Architecture. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, Philadelphia, PA., Oct. 2000.

[68] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[69] A. Lines. Asynchronous interconnect for synchronous soc design. *IEEE Micro*, 24(1):32–41, Jan/Feb 2004.

[70] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings, ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Monterey, Ca, USA, Feb 1999.

[71] C. Metcalf. *Managing Scheduled Routing With A High-Level Communications Language.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1997.

[72] A. M. Michelson and A. H. Levesque. *Error-Control Techniques for Digital Communication.* John Wiley and Sons, New York, NY, 1985.

[73] Micron Technology, Inc. Mt48lc4m32b2 sdram data sheet. In *http://www.micron.com/products/dram/sdram/part.aspx?part=MT48LC4M32B2F5-6*, 2003.

[74] E. Mirsky and A. Dehon. MATRIX: A Reconfigurable Computing Architecture with Configurable Intruction Distribution and Deployable Resources. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1996.

[75] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner. Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems. In *Proceedings, IEEE International ASIC/SOC Conference*, Washington, DC, Sept. 1999.

[76] S. Nanda, K. Balachandran, and S. Kumar. Adaptation techniques in wireless packet data services. *IEEE Communications Magazine*, pages 54–64, Jan 2000.

[77] T. Njolstad, O. Tjore, K. Svarstad, L. Lundheim, T. O. Vedal, J. Typpo, T. Ramstad, L. Wanhammar, E. J. Aas, and H. Danielsen. A socket interface for gals using locally dynamic voltage scaling for rate-adaptive energy savings. In *Proceedings, Fourteeth Annual IEEE International ASIC/SOC Conference*, Arlington, VA, Sep. 2001.

[78] OMI. PI-Bus Peripheral Interconnect Bus - a bus for interconnecting micro-cells on chip. In *http://www.omimo.be*, Aug. 1997.

[79] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing.* Prentice-Hall, Inc., Upper Saddle River, NJ, 1999.

[80] L. Papke, P. Robertson, and E. Villebrun. Improved decoding with the sova in a parallel concatenated (turbo-code) scheme. In *Proceedings of ICC'96*, Dallas, Texas, June 1996.

[81] W. Peterson. Design Philosophy of the Wishbone SoC Architecture. In *Silicore Corporation*, 1999. http://www.silicore.net/wishbone.htm.

[82] J. Proakis. *Digital Communications.* McGraw-Hill, New York, NY, 1995.

[83] T. S. Rappaport. *Wireless Communications: Principles and Practice.* Prentice Hall PRT, Upper Saddle River, NJ, 1996.

[84] Robert P. Dick, Niraj K. Jha. MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis. In *In Proc. Design, Automation and Test in Europe, pages 263-270*, Mar. 1999.

[85] P. Robertson, E. Villebrun, and P. Hoeher. A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain. In *Proceedings of ICC'95*, pages 1009–1013, Seattle, Washington, June 1995.

[86] R.Wilson, R.French, C.Wilson, S.Amarasinghe, J.Anderson, S.Tjing, S.Liao, C.W.Tseng, M.Hall, M.Lam, and J.Hennessy. SUIF: An Infrastructure for Research on Paralleling and Optimizing Compilers. In *ACM SIGPLAN Notices, 29(12)*, Dec. 1996.

[87] W. E. Ryan. Concatenated Convolutional Codes and Iterative Decoding. *http://www.ece.arizona.edu/ ryan/*, May 2001.

[88] K. Ryu, E. Shin, and V. Mooney. A Comparison of Five Different Multiprocessor SoC Bus Architectures. In *Proceedings of the EUROMICRO Symposium on Digital Systems Design (EUROMICRO'01)*, pages 202–209, September 2001.

[89] K. K. Ryu and V. J. M. III. Automated bus generation for multiprocessor soc design. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*, pages 282–287, Munich, Germany, Mar. 2003.

[90] T. Schaffer, A. Stanaski, A. Glaser, and P. Franzon. The NCSU Design Kit for IC Fabrication through MOSIS. In *International Cadence User Group Conference*, Austin, Texas, Sept. 1998.

[91] C. Schurgers, L. V. der Perre, M. Engels, and H. D. Man. Adaptive turbo decoding for indoor wireless communication. In *URSI International Symposium on Signals, Systems, and Electronics (ISSSE'98)*, pages 107–111, Pisa, Italy, Sept. 1998.

[92] D. Shoemaker, C. Metcalf, and S. Ward. NuMesh: An Architecture Optimized for Scheduled Communication. *Journal of Supercomputing*, 10:285–302, 1996.

[93] A. Sinha and A. P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In *Proceedings of the 38th conference on Design automation*, 2001.

[94] Sonics, Incorporated. Corporate Web Site. In *http://www.sonicsinc.com*, 1999.

[95] S.S.Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[96] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson. A Dynamically Reconfigurable Adaptive Viterbi Decoder. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 2002.

[97] Texas Instruments, Inc. Development Tools: Code Composer Studio. In *http://dspvillage.ti.com/docs/catalog/devtools/selection.jhtml?templateId=5121&path=templatedata/cm/toolswovw/data/ccs_ovw&familyId=44&toolTypeId=30&toolTypeFlagId=2*, 2003.

[98] Texas Instruments, Inc. TMS320C6713, TMS320C6713B Floating-Point Digital Signal Processor. In *http://focus.ti.com/lit/ds/symlink/tms320c6713.pdf*, 2004.

[99] M. J. Thul, T. Vogt, F. Gilbert, and N. Wehn. Evaluation of algorithm optimizations for low-power turbo-decoder implementations. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2002.

[100] M. C. Valenti and B. D. Woerner. Variable latency turbo codes for wireless multimedia applications. In *Proc. of the International Symposium on Turbo Codes and Related Topics*, pages 216–219, Brest, France, 1997.

[101] S. Vishwanath and A. J. Goldsmith. Adaptive turbo coded modulation for flat fading channels. In *Proc. IEEE Vehicular Technology Conference*, Boston, MA, Sept. 2000.

[102] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, Apr. 1967.

[103] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997.

[104] M. Wan, Y. Ichikawa, D. Lidsky, and J. Rabaey. An energy-conscious exploration methodology for heterogeneous DSPs. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 111–117, Santa Clara, CA., May 1998.

[105] C. Weems. Asynchronous simd: An architectural concept for high performance image processing. In *Proc. IEEE Int'l Workshop on Computer Architecture for Machine Perception*, pages 235–243, Boston, MA, Oct 1997.

[106] D. Wingard. MicroNetwork-Based Integration for SOCs. In *Proceedings, ACM/IEEE 38th Design Automation Conference*, June 2001.

[107] J. P. Woodard and L. Hanzo. Comparative Study of Turbo Decoding Techniques: An Overview. *IEEE Transactions on Vehicular Technology*, 49(6):2208–2232, Nov. 2000.

[108] xess Corporation. myCSoC: Design Explorations With Your COnfigurable System on a chip. In *www.xess.com/manuals/myCSoC-1_3.pdf*, 2000.

[109] Xilinx, Inc. 3GPP Turbo Decoder. In *http://www.xilinx.com/products/logicore/alliance/sysonchip/sysonchip_3gpp_tbd.pdf*, 2001.

[110] Xilinx, Inc. Xilinx Virtex-II Web Power Tool Version 2.1.0. In *http://www.xilinx.com/cgi-bin/power_tool/web_power_tool.pl*, 2001.

[111] Y. Kaji, T. Fujiwara, T. Kasami and S. Lin. A Trellis-Based Recursive Log-MAP Algorithm for Binary Linear Block Codes. *Technical Report of IEICE*, IT96-79, Mar. 1997.