MULTICORE SYSTEM DESIGN WITH XUM: THE EXTENSIBLE UTAH MULTICORE PROJECT

by

Benjamin Meakin

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2010

Copyright © Benjamin Meakin 2010

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Benjamin Meakin

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Ganesh Gopalakrishnan

Rajeev Balasubramonian

Ken Stevens

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Benjamin Meakin in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh Gopalakrishnan
Chair: Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Director

Approved for the Graduate Council

Charles A. Wight
Dean of The Graduate School

To my wife and girls.

ABSTRACT

With the advent of aggressively scaled multicore processors utilizing increasingly complex on-chip communication architectures, the need for efficient and standardized interfaces between parallel programs and the processors that run them is paramount. Hardware designs are constantly changing. This complicates the task of evaluating innovations at all system layers. Some of the most aggressively scaled multicore devices are in the embedded domain. However, due to smaller data sets, embedded applications must be able to exploit more fine grained parallelism. Thus, more efficient communication mechanisms are needed.

This thesis presents a study of multicore system design using XUM: the Extensible Utah Multicore platform. Using state-of-the-art FPGA technology, an 8-core MIPS processor capable of running bare-metal C programs is designed. It provides a unique on-chip network design and an instruction-set extension used to control it. When synthesized, the entire system utilizes only 30% of a Xilinx Virtex5 FPGA. The XUM features are used to implement a low-level API called MCAPI; the Multicore Association Communication API. The transport layer of a subset of this API has a total memory footprint of 2484 bytes (2264B code, 220B data). The implemented subset provides blocking message send and receive calls. Initial tests of these functions indicate an average latency of 310 cycles (from funtion call to return) for small packet sizes and various networking scenarios. It's low memory footprint and low latency function calls make it ideal for exploiting fine-grained parallelism in embedded systems.

The primary contributions of this work are threefold; First, it provides a valuable platform for evaluating the system level impacts of innovations related to multicore systems. Second, it is a unique case study of multicore system design in that it illustrates the use of an instruction set extension to interface a network-on-chip with a low level

communication API. Third, it provides the first hardware assisted implementation of MCAPI enabling fast message passing for embedded systems.

CONTENTS

ABSTRACT . . . . **v**

LIST OF FIGURES . . . . **x**

LIST OF TABLES . . . . **xii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Motivation and Objectives 1
1.2 Thesis Statement 2
1.3 Parallel Programming 2
1.3.1 Analogy to Project Management 2
1.3.2 Message Passing vs. Shared Memory 3
1.4 Parallel Computer Architecture 4
1.4.1 Embedded Systems 4
1.4.2 Networks-on-Chip 5
1.5 Challenges of Parallel System Design 6
1.5.1 Limited Parallelism 6
1.5.2 Locality 6
1.5.3 Code Portability 6
1.6 Overview of Project 7

2. DESIGN OF XUM MULTICORE PROCESSOR **8**

2.1 MIPS Core 9
2.1.1 Pipeline 10
2.1.2 Arithmetic-Logic Unit 10
2.1.3 Controlling Program Flow 12
2.1.4 Memory 12
2.1.5 Interrupts 13
2.1.6 Communication 15
2.2 XUM Network-on-Chip 15
2.2.1 Topology 16
2.2.2 Flow Control 17
2.2.3 Routing 18
2.2.4 Arbitration 19
2.2.5 Interfacing 20
2.2.6 Router Architecture 21
2.2.7 Extensibility 22

2.3 Instruction Set Extension 22
2.4 Tool-Chain 26
2.5 Synthesis Results 27

3. MULTICORE COMMUNICATION API (MCAPI) 29

3.1 MCAPI Introduction 29
3.2 Transport Layer Implementation 30
3.2.1 Initialization and Endpoints 30
3.2.2 Connectionless Communication 35
3.2.3 Connected Channels 37
3.3 Performance and Results 40
3.3.1 Hypothetical Baseline 41
3.3.2 Memory Footprint 42
3.3.3 Single Producer - Single Consumer 42
3.3.4 Two Producers - Single Consumer 42
3.3.5 Two Producers - Two Consumers 43
3.3.6 Discussion of Results 44

4. APPLICATION SPECIFIC NOC SYNTHESIS 49

4.1 NoC Synthesis Introduction 49
4.2 Related Work 50
4.3 Background 52
4.3.1 Assumptions 52
4.3.2 Optimization Metrics 52
4.4 Irregular Topology Generation 52
4.4.1 Workload Specification 52
4.4.2 Topology Generation 54
4.5 Routing Verification 56
4.5.1 Deadlock Freedom 56
4.5.2 Routing Algorithm 57
4.6 Node Placement 58
4.7 Network Simulator 60
4.7.1 Simulating an Irregular Network 60
4.7.2 Simulating Application Specific Traffic 60
4.8 Results 61
4.8.1 Test Case 61
4.8.2 Scalability Test 62
4.8.3 General Purpose Test 63
4.8.4 gpNoCSim Simulator 64
4.8.5 Results Summary 64

5. FUTURE DIRECTIONS 65

5.1 MCAPI Based Real-time Operating System 65
5.2 Scalable Memory Architectures 65
5.3 Parallel Programming Languages 66
5.4 Multicore Resource API (MRAPI) 67

5.5 Multicore Debugging 67

6. CONCLUSIONS **68**

6.1 Project Recap 68
6.2 Fulfillment of Project Objectives 68

APPENDIX: ADDITIONAL XUM DOCUMENTATION **70**

REFERENCES **76**

LIST OF FIGURES

2.1 XUM Block Layout 9
2.2 XUM Tile Layout 9
2.3 XUM Core - MIPS Pipeline 11
2.4 XUM Cache Organization 14
2.5 XUM Network Interface Unit 16
2.6 Packet NoC - Flow Control 17
2.7 Acknowledge NoC - Flow Control 18
2.8 Arbitration FSM 19
2.9 NoC Interfacing 20
2.10 Router Layout 21
2.11 Extending XUM 22
2.12 Packet Structure 24
2.13 ISA Example 25
3.1 MCAPI Subset 31
3.2 Data Structures 32
3.3 Interrupt Service Routine 33
3.4 MCAPI Initialize 34
3.5 MCAPI Create Endpoint 34
3.6 MCAPI Get Endpoint 35
3.7 MCAPI Message Send 36
3.8 MCAPI Message Receive 38
3.9 Algorithm for Managing Connected Channels 40
3.10 MCAPI Transport Layer Throughput 45
3.11 MCAPI Transport Layer Latency 46
3.12 MCAPI Throughput vs Baseline 48
4.1 Channel Dependency Graph 57
4.2 Results - Topology with Node Placement 61

4.3 Results - Scalability 63
5.1 MCAPI OS Kernel 66

LIST OF TABLES

2.1 ALU Instructions 11
2.2 Control Flow Instructions 12
2.3 Memory Instructions 13
2.4 Interrupt Instructions 14
2.5 Interrupt Sources 15
2.6 Routing Algorithm 18
2.7 ISA Extension 23
2.8 Address Space Partitioning 27
2.9 Results - Logic Synthesis 28
3.1 MCAPI Memory Footprint 42
3.2 MCAPI Latency: 1 Producer/1 Consumer 43
3.3 MCAPI Throughput: 1 Producers/1 Consumer 43
3.4 MCAPI Latency: 2 Producers/1 Consumer 43
3.5 MCAPI Throughput: 2 Producers/1 Consumer 43
3.6 MCAPI Latency: 2 Producers/2 Consumer 44
3.7 MCAPI Throughput: 2 Producers/2 Consumer 44
4.1 Example Workload 53
4.2 Results - Topology Synthesis 62
4.3 Results - General Purpose Traffic 63
4.4 Results - gpNoCSim Simulator 64
A.1 NIU Flags 71
A.2 SNDHD Variations 74

CHAPTER 1

INTRODUCTION

1.1 Motivation and Objectives

In recent years, the progress of computer science has been focused on the development of parallel computing systems (both hardware and software) and making those systems efficient, cost effective, and easy to program. Parallel computers have been around for many years. However, with the ability to now put multiple processing cores on a single chip the technologies of the past are in many cases obsolete. Technologies which have enabled parallel computing in the past must be re-designed with new constraints in mind. One important part of achieving these objectives has been to investigate various means of implementing mechanisms for collaboration between on-chip processing elements. This ultimately has lead to the use of on-chip communication networks [5, 1].

There has been a significant amount of research devoted to on-chip networks, also known as Networks-on-Chip (NoC). Much of this has focused on the development of network topologies, routing algorithms, and router architectures that minimize power consumption and hardware cost while improving latency and throughput [31, 6, 8, 22]. However, no one solution seems to be universally acceptable. It follows that standardized methods to provide communication services on a multicore system-on-chip (SoC) also remain unclear. This can only slow the progress of the adoption of concrete parallel programming practices, which software developers are desperately in need of.

This problem has clearly been recognized by the industry. The formation of an organization called the Multicore Association and its release of several important APIs is evidence of this. The Multicore Communication API (MCAPI) is one of the Multicore Associations most significant releases [17]. MCAPI is a perfect example of how technologies of the past are being rethought with the new constraints of multicore. However,

the only existing implementations of MCAPI are fairly high-level and depend on the services provided by an operating system. In addition, NoC hardware is constantly evolving and varies from chip to chip. Therefore, the interface between operating systems and low-level APIs with NoC hardware is an important area of research.

1.2 Thesis Statement

In addition to this motive, MCAPI aims to provide lightweight message passing that will reduce the overhead associated with parallel execution.

> In order to exploit fine grained parallelism, often present in embedded systems, better on-chip communication mechanisms are needed. An FPGA platform for studying such mechanisms and a hardware assisted implementation of an emerging communication API are timely enablers for finding solutions to this problem.

This thesis presents the design of a multicore processor called XUM: the Extensible Utah Multicore project. The scope of this project ranges from the development of individual processing cores, an interconnection network, and a low-level software transport layer for MCAPI. There are three major contributions of this work. First, it provides a valuable platform for evaluating the system level impacts of innovations related to multicore hardware design, verification and debugging, operating systems, and parallel programming languages. Second, it is a unique case study of multicore system design in that it illustrates the use of an instruction set extension to interface an NoC with system software. Third, it provides the first hardware assisted implementation of MCAPI.

1.3 Parallel Programming

1.3.1 Analogy to Project Management

Programming a parallel computer is not unlike managing a team of engineers working on a large project. Each individual is assigned one or more tasks to complete. These tasks can be created in one of two ways. They may be independent jobs that are each important to the project but are very different in their procedure. An example of this may be the

tasks of development and testing. On the other hand, they may be subtasks created as a result of partitioning one job into many very similar, but smaller jobs. This is analogous to task and data parallelism.

As a manager, there are many issues that one must address in order to ensure that this team of individuals is operating at their maximum level of productivity. Tasks must be divided such that the collaboration needed between individuals is not too frequent, yet no one task is too large for one person to complete in a reasonable amount of time. Individuals working on the same task must have quick access to one another in order to minimize the time spent collaborating instead of working. Tasks must also be assigned to the individuals that are most capable of completing them, since different people have different strengths. This must also be done while keeping everyone on the team busy.

In programming a parallel computer, one faces the same dilemmas in seeking good performance. Individuals in the above example represent processors. Each executes their given task sequentially, but their task assignment and ability to collaborate are the key factors in achieving optimal productivity. While task scheduling is equally important as inter-task collaboration, it is beyond the scope of this work. Therefore, the focus will be on collaboration; also referred to as communication.

1.3.2 Message Passing vs. Shared Memory

Traditionally, processors in a parallel computer collaborate through either shared memory or message passing [14]. In a shared memory system, processors have at least some shared address space with the other processors in the system. When multiple processors are working on the same task they often access the same data. To ensure correctness, accesses to the same data (or variable) must be mutually exclusive. Therefore, when one processor is accessing a shared variable, the others must wait for their turn. This typically works fine when there are only a few processors. However, when there are many, the time required to obtain access can become a serious performance limitation. Imagine a team of 30 writers who all must share the same note pad. By the time the note pad makes it to the writer whose turn it is to use it, what may have been

written has been forgotten. The same situation may be more practical if there are only 2 writers per note pad.

In such situations it makes more sense to copy the shared resource and send it to everyone that uses it. This means that more memory is needed. However, it is likely that better performance will result. This is known as message passing. Though, using message passing has its own drawbacks. Aside from requiring redundant copies of data, it takes time to transmit a copy. In a multicore chip where processors are physically located on the same device, it may be more efficient to transmit a pointer to shared memory than to copy and transmit a chunk of data. Therefore, a system that allows both forms of collaboration is desirable in a multicore device.

1.4 Parallel Computer Architecture

Parallel computers come in many varieties ranging from supercomputers with thousands of nodes connected by a high-speed fiber optic network to multicore embedded devices with several heterogeneous cores and I/O devices connected by an on-chip network. This work is primarily focused on closely distributed systems. Meaning systems in which the majority of communicating nodes are on the same chip. The architectural assumptions that follow are based on this application domain.

1.4.1 Embedded Systems

Embedded systems often have very unique constraints from that of high performance systems. For example, high performance systems often benefit from large data sets. This means that it is easier to amortize the cost of communication over long periods of independent computation in data parallel applications. In this case, a message passing mechanism does not need to offer very low latency in order for the application to perform well. On the other hand, embedded systems usually have much smaller data sets due to more limited memory resources. This means that concurrent processes will need to communicate more frequently. In other words, a smaller percentage of time will be spent doing computation if the same message passing mechanism is used. In order to see

an improvement in performance associated with parallel execution, an application must either have a large data set or very low latency communication.

Embedded systems also tend to benefit much more from task parallelism than high-performance systems. Super-computers nodes generally do not run more than one application at a time. Even personal computers do not have a lot of task parallelism because users switch between applications relatively infrequently. On the other hand, embedded devices are often running many independent tasks at the same time. Imagine a cell phone processor which must simultaneously handle everything from baseband radio processing to graphical user interfaces. Due to these constraints, many of the design decisions made in this work only make sense in the embedded domain.

1.4.2 Networks-on-Chip

It was made apparent in the discussion on shared-memory that any shared resource in a system becomes a performance bottleneck as the number of cores (sharers of the resource) increases. This is a major reason why shared busses have been replaced by networks-on-chip [5]. Unlike a buss, a network can support many transactions between interconnected components simultaneously. Therefore, to increase parallelism in multi-core devices, networking technology has been applied on-chip

On-chip networks have their own scalability issues however. For example, as the number of network nodes increases, so does the average number of hops (inter-router links) that a message must traverse, thus increasing latency. However, one cannot simply fully connect all nodes through a crossbar. This would result in a large critical path delay through the crossbar logic and long wires, which are increasingly problematic in deep submicron semiconductor processes. An efficient NoC will balance the network topology size with router complexity and wire lengths. NoCs are also characterized by the methods used for flow-control, routing algorithm, topology, and router architecture. These issues have been extensively researched. This work will give details of the NoC designed for XUM. It is a basic design that will allow for some of these more complex methods to be incorporated in future releases.

1.5 Challenges of Parallel System Design

1.5.1 Limited Parallelism

One of the major challenges in designing parallel systems comes from the fact that most applications have a very limited amount of parallelism. This is especially true of applications that heavily rely on algorithms that were designed assuming sequential execution. It is even truer in embedded systems due to smaller data sets, thus limiting data parallelism. Aside from rethinking algorithms, the key to being able to achieve good performance when parallelism is limited is to have efficient coordination mechanisms. Synchronization can then happen more frequently without compromising performance.

1.5.2 Locality

Perhaps the biggest issue with performance in parallel systems is with data locality [14]. Anytime a program accesses data there is a cost associated with that access. The further the physical location of the data is from the processor, the larger the cost is for accessing it. Caches and memory hierarchies help to combat this problem. However, as systems scale more aggressively the hardware that controls these hierarchies becomes more complex, power hungry, and expensive. Therefore, the programmer must have some control over locality. Through the tool chain, instruction set, or operating system the programmer should be able to specify the location of data structures and the location of the processor that will run the code that uses these data structures.

1.5.3 Code Portability

As computer scientists seek to find solutions to these various problems, architectures are constantly changing. With the cost of developing large applications being very considerable, programs must be portable across these constantly changing platforms. This leads to yet another challenge related to parallel system design. How can programs be made to run on variable hardware?

1.6 Overview of Project

Each of these challenges is addressed in this work. Limited parallelism is addressed by designing a highly efficient communication system, thus making a program with frequent synchronization points more capable of performing well. Data locality is also dealt with. Lightweight message passing provides a programmer with the ability to copy data to where it needs to be for efficient use. To avoid inefficient use of memory, tool-chain modifications provide the ability to statically allocate memory to only the core that uses it. Likewise, ISA modifications provide the programmer with the ability to specify which core runs which code. Code portability is addressed by standardizing the communication interface between cores with an ISA extension. The incorporation of an emerging communication API (MCAPI) further promotes portable system design.

This thesis represents a large body of work done by one student over the course of almost two years. Chap. 2 presents the design of XUM: the extensible Utah multicore processor. It describes the processor core designed, the network-on-chip, and the ISA extension used to interface them. XUM provides only heterogeneous cores and assumes general purpose applications. However, the system is designed with the assumption that future extensions would provide heterogeneous cores and be used to implement specific applications (as is common in the embedded space). Therefore, Chap. 4 presents a tool for automatically synthesizing custom on-chip networks for specific applications. Chap. 3 details the implementation of the MCAPI transport layer on top of the XUM platform. It discusses the techniques used to realize a subset of the API and proposes ideas for the remainder of the API. Since one of the purposes of XUM is to provide a multicore research platform, Chap. 5 discusses some possible research ideas and future directions. Lastly, Chap. 6 reviews the project and discusses the fulfillment of the stated objectives.

CHAPTER 2

DESIGN OF XUM MULTICORE PROCESSOR

In order to progress the state of the art with respect to parallel computing, innovations must be applied at all system layers. Research must encompass more than just hardware or just software. The system wide impacts of new ideas must be evaluated effectively. It follows that in order to study innovations of this scope there is clearly a need for new multicore research platforms. Intel's Teraflops 80-core research chip [11], UC Berkley's RAMP project [7, 12], and HP Labs open source COTSon simulator [2] are all examples of systems developed with this motive. At the University of Utah, a multicore MIPS processor called XUM has been developed on state of the art Xilinx FPGA's to provide a configurable multicore platform for doing parallel computer research. It differs from other such systems in that it is tailored for embedded systems and provides an instruction set extension which enables the implementation of efficient message passing over an on-chip network.

The basic architecture of XUM is depicted in Fig. 2.1. XUM is designed to be highly configurable for various applications. The complete system is partitioned into blocks and tiles (Fig. 2.2). Tiles can be heterogeneous modules, though XUM only provides one tile design consisting of the components indicated in Fig. 2.2. Tiles are interconnected by a scalable on-chip network which uses a handshaking protocol and FIFO buffering to interface with the tiles. This enables the use of a unique clock domain for each tile. The on-chip network provides two unique topologies, each supporting different features that are designed to be utilized by different parts of the XUM on-chip communication system. Each of these components and subsystems are described in detail throughout this chapter.

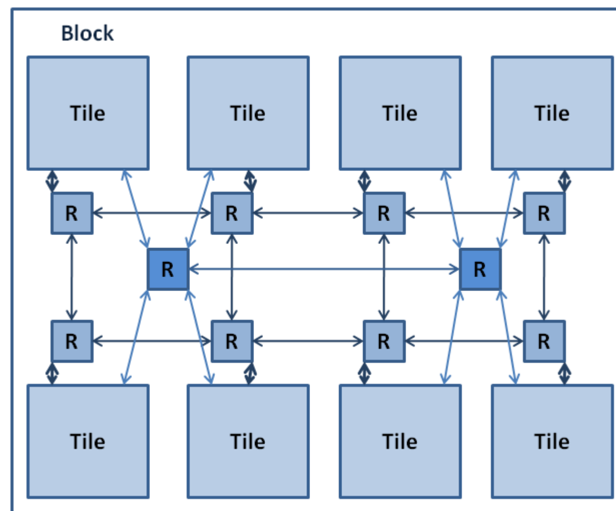

Figure 2.1. XUM Block Layout

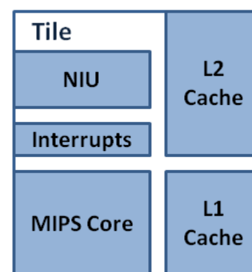

Figure 2.2. XUM Tile Layout

2.1 MIPS Core

The MIPS core used in XUM was designed from the ground up and implements the entire core instruction set (excluding the arithmetic and floating-point subsets). Each part of the processor is briefly described in this section. With the exception of XUM specific extensions, the MIPS specification available in [16] is strictly adhered to. However, one variation is that the data-path is only 16-bits wide. This was an important design decision. Since XUM is all about studying multicore computing, it is desirable for many processor cores to be available on a single FPGA. To maximize the number of blocks and tiles

that could be implemented on one FPGA, smaller 16-bit cores are used in place of the expected 32-bit MIPS cores.

2.1.1 Pipeline

The processor core uses a 6-stage in-order pipeline as shown in Fig. 2.3. Initially, a program counter is incremented in stage 1 to provide an address for the instruction fetch in stage 2. The cache architecture will be described in Sect. 2.1.4, but it is reasonable to assume that all instruction fetches will complete in one clock cycle. Stage 3 decodes the 32-bit instruction by extracting it's various fields and setting the appropriate control signals based on those fields. The register file is also accessed in this stage. Note that a register can be read in the same clock cycle that it is written to without conflict. Next, the functional units perform the computation for the bulk of the instruction set. Aside from the traditional arithmetic-logic unit (ALU), a network interface unit (NIU) appears as a co-processor (or additional functional unit) in the pipeline. This module is described in Sect. 2.1.6. It handles all communication oriented instructions that have been added to the MIPS ISA in this design. Accesses to data memory are handled in stage 5. Operations in this stage may stall the pipeline. Finally, results of memory or functional unit operations are written to their destination registers in stage 6.

Much of the basic design follows from the MIPS processor architecture presented in [25]. This includes the logic used for data forwarding, hazard detection, and pipeline organization. Though it has been heavily modified in this implementation, the design in [25] is a fundamental starting point for any student of this work.

2.1.2 Arithmetic-Logic Unit

The ALU provides addition, subtraction, bitwise, and logical operations. This does not include fixed-point multiplication and division or floating-point arithmetic. A future release of XUM will likely include a fixed-point multiplier/divider unit. At present, applications should use software based methods for doing fixed-point multiplication and division. Tab. 2.1 presents the instructions available in this release.

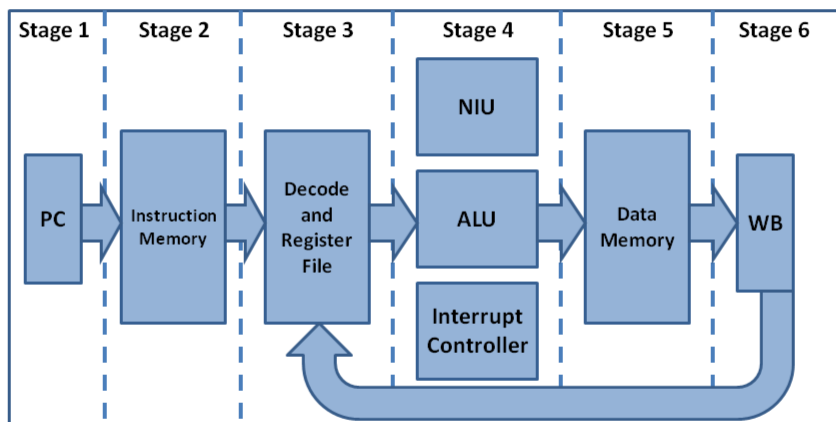

Figure 2.3. XUM Core - MIPS Pipeline

Table 2.1. ALU Instructions

Name	Assembly	Operation
Add Signed	add $rd,$rs,$rt	rd = rs + rt
Add Unsigned	addu $rd,$rs,$rt	rd = rs + rt
Add Signed Immediate	addi $rt,$rs,*Imm*	rt = rs + *Imm*
Add unsigned Immediate	addiu $rt,$rs,*Imm*	rt = rs + *Imm*
And	and $rd,$rs,$rt	rd = rs & rt
And Immediate	andi $rt,$rs,*Imm*	rt = rs & *Imm*
Nor	nor $rd,$rs,$rt	$rd = (rs \mid rt)$
Or	or $rd,$rs,$rt	$rd = rs \mid rt$
Or Immediate	ori $rt,$rs,*Imm*	$rt = rs \mid Imm$
Set On Less Than	slt $rd,$rs,$rt	$rd = (rs < rt)?1:0$
Set On Less Than Immediate	slti $rt,$rs,*Imm*	$rt = (rs < Imm)?1:0$
Set On Less Than Unsigned Immediate	sltiu $rt,$rs,*Imm*	$rt = (rs < Imm)?1:0$
Set On Less Than Unsigned	sltu $rd,$rs,$rt	$rd = (rs < rt)?1:0$
Shift Left Logical	sll $rd,$rs,*Shamt*	$rd = rs << Shamt$
Shift Left Logical Variable	sllv $rd,$rs,$rt	$rd = rs << rt$
Shift Right Arithmetic	sll $rd,$rs,*Shamt*	$rd = rs >>> Shamt$
Shift Right Arithmetic Variable	srav $rd,$rs,$rt	$rd = rs >>> rt$
Shift Right Logical	srl $rd,$rs,*Shamt*	$rd = rs >> Shamt$
Shift Right Logical Variable	srlv $rd,$rs,$rt	$rd = rs >> rt$
Subtract	sub $rd,$rs,$rt	rd = rs - rt
Subtract Unsigned	subu $rd,$rs,$rt	rd = rs - rt

Table 2.2. Control Flow Instructions

Name	Assembly	Latency
Branch On Equal	beq $rt,$rs,*BranchAddr*	+3 cycles
Branch On Not Equal	bne $rt,$rs,*BranchAddr*	+3 cycles
Jump	j *JumpAddr*	+1 cycle
Jump And Link	jal *JumpAddr*	+1 cycle
Jump Register	jr $rs	+3 cycles
Jump And Link Register	jalr $rs	+3 cycles

2.1.3 Controlling Program Flow

All of the MIPS flow control operations are implemented. They all support one branch delay slot, meaning that the instruction immediately following the branch or jump will always be executed. Subsequent instructions will then be flushed from the pipeline. Different operations have different cycle latencies. Tab. 2.2 gives all of the available control flow operations and the number of additional cycles required for the branch or jump to actually occur. No branch prediction is used in this implementation. Branches are assumed not taken. Therefore, to optimize performance it is recommended that the branch delay slot is used wherever possible.

2.1.4 Memory

All types of load and store operations are supported except unaligned memory accesses. Tab. 2.3 lists the available operations. Note that because this is a 16-bit variation of MIPS, which is a 32-bit instruction set, the word size is 2 bytes. Therefore, load/store half-word operations are the same as load/store word operations.

Each processor core is mated to a 2-level distributed cache hierarchy. The first level (L1) is a direct-mapped cache with 8 words per block (or line). In the current version of XUM, the second level (L2) is the last level in the memory hierarchy and is implemented in on-chip block RAM. Given that XUM is implemented on an FPGA, there is no advantage to having an L1 and L2 rather than simply having a larger L1. However, the logic is in place for the next release which will include an off-chip SDRAM controller. At that point, a sophisticated memory controller will take the place of the L2 block.

Table 2.3. Memory Instructions

Name	Assembly	
Load Byte	lb $rt,(*Imm*)$rs	rt = Mem[rs+*Imm*]
Load Half Word	lh $rt,(*Imm*)$rs	rt = Mem[rs+*Imm*]
Load Word	lw $rt,(*Imm*)$rs	rt = Mem[rs+*Imm*]
Store Byte	sb $rt,(*Imm*)$rs	Mem[rs+*Imm*] = rt
Store Half Word	sh $rt,(*Imm*)$rs	Mem[rs+*Imm*] = rt
Store Word	sw $rt,(*Imm*)$rs	Mem[rs+*Imm*] = rt

A data memory access is subject to taking multiple clock cycles. When a memory request cannot be immediately satisfied, the processor is stalled. The cache is organized as in Fig. 2.4. The read, write, address, and data input signals represent a memory request. If the desired address is cached in L1, then a cache hit results and processor execution proceeds as normal. However, if the request results in a miss, then the request is buffered and the pipeline is stalled. The control proceeds to read the cache line from L2 (or off chip memory) and write it into the L1. If the destination L1 cache line is valid then that line will be read from L1 and put into a write-back buffer simultaneously. Both the data and the corresponding address are stored in the buffer so that the controller can commit the write-back to L2 in between memory requests from the processor. The write-back buffer can be accessed to satisfy pending requests. Therefore, in the event that the data is in the write-back buffer, a request can be satisfied by operating directly on the buffer rather than waiting for the write-back to commit and subsequently re-loading that cache line into L1.

2.1.5 Interrupts

In order to provide the capabilities required for implementing MCAPI, it necessary for XUM to support interrupts. Each tile has its own interrupt controller. Interacting with the controller is accomplished through the use of the MIPS instructions listed in Tab. 2.4. The interrupt controller appears to the MIPS core as a co-processor. By moving data to and from the controller's registers, specific interrupts can be enabled/disabled, interrupt

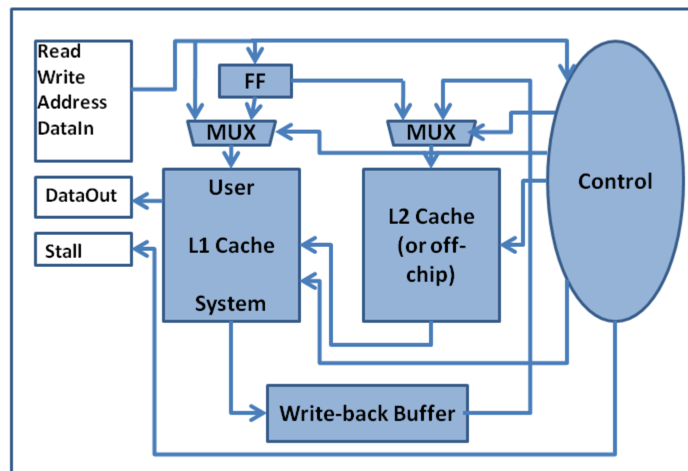

Figure 2.4. XUM Cache Organization

Table 2.4. Interrupt Instructions

Name	Assembly
Move From Co-Processor 0	mfc0 $rd,$cp
Move To Co-Processor 0	mtc0 $cp,$rs
Enable Interrupts	ei
Disable Interrupts	di
System Call	syscall

causes can be identified, and the value of the program counter at the time of the interrupt can be retrieved. The controller's three registers are EPC (0), cause (1), and status (2).

The interrupt controller registers are all 16-bits wide. The most significant bit of the cause and status registers is reserved for the interrupt enable bit. Therefore, there are a total of 15 possible sources. Several sources are currently provided and indicated in Tab. 2.5. These include one software interrupt (initiated by a *syscall* instruction), two interrupts caused by peripherals such as a timer or UART, two interrupts caused by the NoC (one for the *packet* network and one for the *acknowledge* network), and three hardware exceptions.

Table 2.5. Interrupt Sources

Number	Cause
cause[0]	System call (software interrupt)
cause[1]	Timer interrupt
cause[2]	NoC interrupt 1
cause[3]	NoC interrupt 2
cause[4]	Stack overflow exception
cause[5]	Address error exception
cause[6]	Arithmetic overflow exception
cause[7]	UART interrupt
cause[8-14]	Unused

2.1.6 Communication

Each of the previous sub-sections describes parts of the MIPS core that are not unique to this project. To this point, no feature has justified the development of a complete processor core. However, an ISA extension consisting of special instructions targeting on-chip communication is a key feature that enables XUM to provide hardware acceleration for lightweight message passing libraries.

A discussion of the instructions that make up this ISA extension is deferred to a later section. However, they are all implemented in a module called the network interface unit (NIU), shown in Fig. 2.5. The NIU appears as a functional unit in the MIPS pipeline. It takes two operands (A and B), an op-code, and returns a word width data value. This module interprets op-codes, maintains a set of operational flags, provides send and receive message buffering, and interfaces with two distinct on-chip networks which are each utilized for different types of communication.

2.2 XUM Network-on-Chip

Communication between XUM blocks and tiles is done over a network-on-chip (NoC). This communication structure is unique in that it uses two different network topologies for carrying different types of network traffic. These topologies are visualized in Fig. 2.1. These two topologies are referred to as the *packet* and *acknowledge* networks. Their

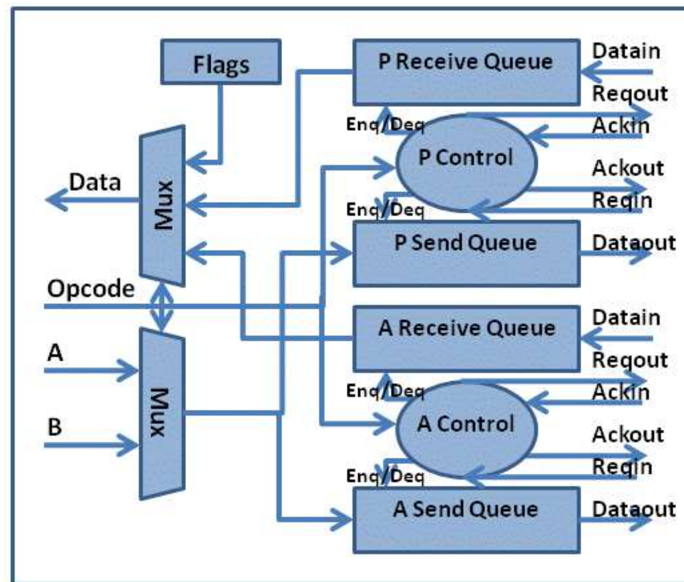

Figure 2.5. XUM Network Interface Unit

differences will be discussed here, while the advantages provided by those differences will become apparent in Chap. 3.

2.2.1 Topology

The *packet* network is designed to be used for point-to-point transmission of large chunks of data. It is a regular mesh topology. Therefore, it is easy to implement at the circuit level, has short uniform links, and provides good throughput. While a mesh network may not be fully utilized by some applications, it is a good general purpose solution and a good starting point for XUM communication. In future extensions, alternative topologies may be considered, especially if heterogeneous tiles are used. Chap. 4 discusses a method for automatically generating network topologies that are tailored for specific applications.

The *acknowledge* network is a different type of mesh network. Two routers are used to connect 8 tiles. This means that individual links will be more heavily utilized than in the *packet* network which has a router for every tile. This also means that throughput

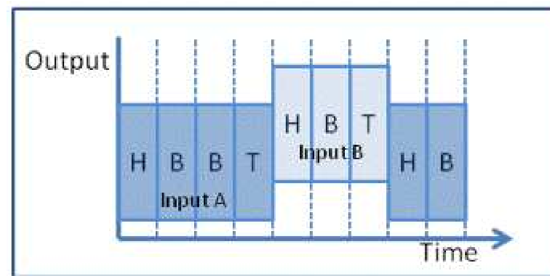

Figure 2.6. Packet NoC - Flow Control

is traded off for lower latency. Indeed, the *acknowledge* network is designed for fast transmission of small pieces of information. This network also supports broadcasts.

2.2.2 Flow Control

The differences between the two networks are not limited to their topologies. They also use different forms of network flow control. The *packet* network uses wormhole flow control. In this scheme, packets are divided into sub-packets called flits. The flit size is determined by the length of the packet header. Flits move through the network in a pipelined fashion with each flit following the header and terminated by a tail flit. These flits are also grouped together in their traversal of the network. Two different packets that must traverse the same link will not be interleaved. This is illustrated in Fig. 2.6, where different colored flits represent different packets. Once a packet header traverses a link, that link will not carry traffic belonging to any other packet until it has seen the tail.

The *acknowledge* network on the other hand, has no such association between flits. As is indicated in Fig. 2.7, flits are not grouped between headers and tails. Therefore, they can traverse a link in any order. This means that flits in the *acknowledge* network never need to wait very long in the link arbitration process, resulting in low latency. However, it is not ideal for transmission of large data streams since each flit must carry some routing information, resulting in significant overhead.

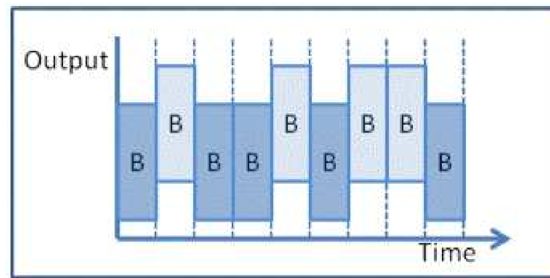

Figure 2.7. Acknowledge NoC - Flow Control

Table 2.6. Routing Algorithm

	Pkt X > Rtr X	Pkt X = Rtr X	Pkt X < Rtr X
Pkt Y > Rtr Y	East	North	West
Pkt Y = Rtr Y	East	Local	West
Pkt Y < Rtr Y	East	South	West

2.2.3 Routing

Routing of flits from source to destination follows a simple dimension order routing scheme in the *packet* network. This is advantageous because it is easy to implement on a mesh topology and is free of cyclic link dependencies which can cause network deadlock [4]. It is also deterministic. This determinism can be used to design network efficient software and simplify debugging. Tab. 2.6 represents the routing algorithm. Each tile is assigned an identifier with *X* and *Y* coordinates. The router that is local to that tile is assigned the same identifier for comparison with the destination of incoming packets. Based on this comparison, the packet is routed to the north, south, east, west, or local directions. Note that the correct *X* coordinate must be achieved prior to routing in the *Y* dimension. Routing in the *acknowledge* network is implemented in the same way. However, instead of the north, south, east, and west directions going to other routers they go directly to the adjacent tiles. The local direction is then used to link adjacent routers.

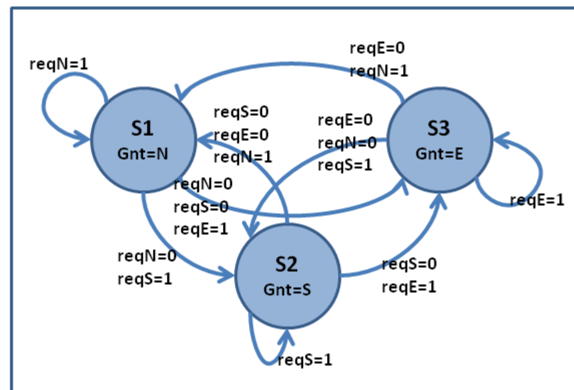

Figure 2.8. Arbitration FSM

2.2.4 Arbitration

When two or more packets need to use the same link they must go through an arbitration process to determine which packet should get exclusive access to the link. The arbiter must be fair, but must be capable of granting access very quickly. The arbitration process used in both network types is governed by the finite state machine (FSM) given in Fig. 2.8. For simplicity, only 3 input channels (north, south, and east) are used and state transitions to and from the initial state have been omitted. This arbitration FSM is instantiated for each output port. The directions (abbreviated by N,S, and E) in the diagram represent requests and grants to each input channel for use of the output channel associated with the FSM instantiation. The request signals are generated by the routing function and the direction that a packet needs to go. This signal only returns to zero when a tail flit is seen. Notice that each state gives priority to the channel that currently has the grant token. If that request returns to zero (from a tail flit), priority goes to the next channel in a round-robin sequence. If there are no pending requests then the FSM returns to the initial state. Arbitration in the *acknowledge* network is the same except that remaining in the same state is only allowed in the initial state. It should be noted that the request signals shown here are different from the request signals used to implement a handshake protocol that moves flits through the network (discussed in the next section).

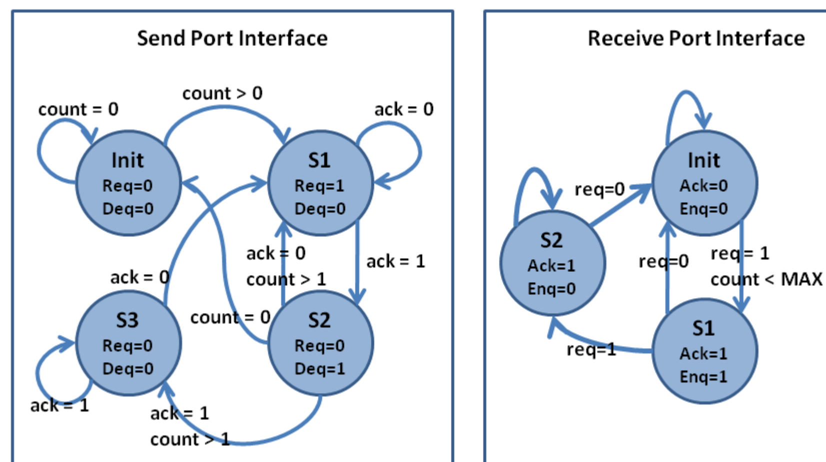

Figure 2.9. NoC Interfacing

2.2.5 Interfacing

Both the *packet* and *acknowledge* networks use a simple handshake protocol to move flits through the network. This simplifies the task of stalling a packet that is spread across several routers for arbitration and provides synchronization across clock domains. Any interface with the on-chip network must also implement this handshake protocol. The NIU discussed earlier is an example of such an interface. To communicate with the network, send and receive ports must implement the state-machines shown in Fig. 2.9.

On the send port side, the request signal is controlled and de-queuing from the send buffer is managed. If the send buffer has an element count of more than zero, than the request signal is set high until an acknowledgment is received. At this point, the data at the front of the queue is sent on the network and de-queued from the send buffer. If there are more elements in the buffer to send then the state machine must wait for the acknowledge signal to return to zero before sending another request. The receive port interface is simpler. If a request is received and the buffer element count is less than the buffer's maximum capacity, then the data is en-queued and an acknowledge signal is set high until the request returns to zero.

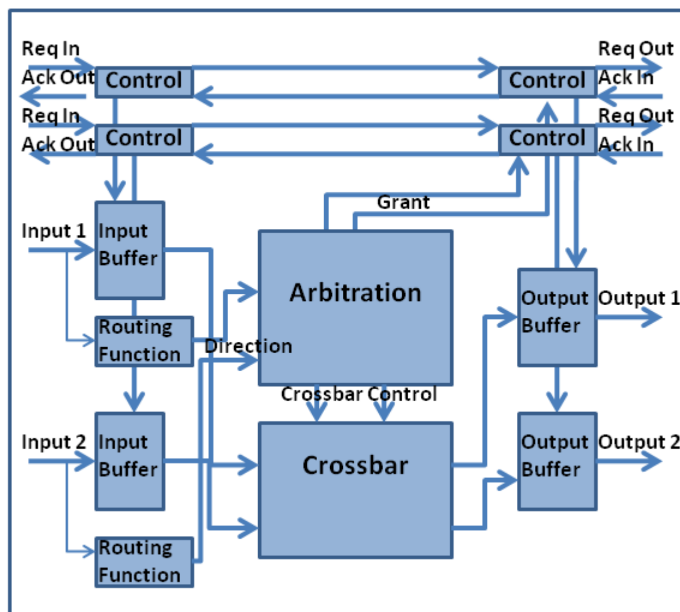

Figure 2.10. Router Layout

2.2.6 Router Architecture

All of the protocols for NoC operation that have been discussed in this section are implemented in the router module outlined in Fig.2.10. For simplicity, only a radix two router is shown. The router module is divided into two stages; input and output. Each input channel has its own corresponding flow controller. Therefore, if a packet on one input/output path gets stalled it doesn't affect packets on other input/output paths. In the input stage, the incoming flit is latched into the input buffer if it is free. If the flit is a header, then it's routing information is extracted by the routing function, which then generates a request for an output channel. If available, the arbiter then grants access to the requested output channel and sets the crossbar control signals. The crossbar then links the input to the desired output based on these control signals. After the flit passes through the crossbar it is latched at the output port and traverses the link to the next router.

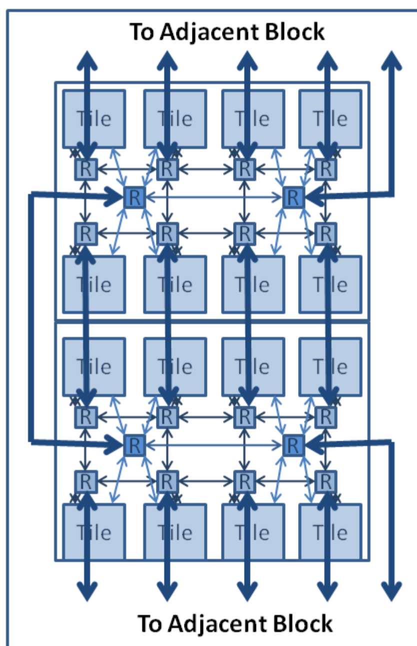

Figure 2.11. Extending XUM

2.2.7 Extensibility

The XUM network architecture is designed such that more cores can easily be added as FPGA technologies improve. By interconnecting blocks of 8 tiles, larger structures can be created. Fig.2.11 illustrates this concept. As with tiles, blocks need not be heterogeneous. They only need to implement the handshake protocol for interfacing with the NoC. For example, a block with only tiles that perform I/O is currently under development.

2.3 Instruction Set Extension

One important feature that XUM provides is explicit control over on-chip communication. Rather than relying on complex memory management hardware to move data around the processor, a programmer can build packets and send them anywhere on-chip at the instruction level. This is accomplished by providing an instruction set extension which is used to facilitate networking. The primary reason for using an instruction set extension is for good performance at a minimal cost. It is also to provide a standard

Table 2.7. ISA Extension

Name	Assembly
Send Header (with options)	sndhd $rd,$rs,$rt (.b, .s, .i, .l)
Send Word	sndw $rd,$rs or sndw $rd,$rs,$rt
Send Tail	sndtl $rd
Receive Header	rechd $rd
Receive Word	recw $rd
Receive Word Conditional	recw.c $rd, $rs
Send Acknowledge	sndack $rs
Broadcact	bcast $rs
Receive Acknowledge	recack $rd
Get Node ID	getid $rd
Get Operational Flag	getfl $rd,Immediate

interface to NoC hardware that can change as technologies improve. For code portability, communication should be standardized as part of an instruction set rather than relying on device specific memory-mapped control registers to utilize the NoC.

The advantages of providing explicit instruction-level control over the movement of data have already been recognized by researchers. The FLEET architecture presented in [30] is one such example. However, while FLEET goes to one extreme by providing only communication oriented instructions (computation is implicit), this work seeks a middle ground by extending a traditional ISA.

The specific instructions that make up the extension are given in Tab.2.7. Detailed documentation for each instruction is given in the appendix. However, the ISA extension can be broken down into instructions for sending packets (*sndhd, sndw, sndtl*), receiving packets (*rechd, recw, recw.c*), sending and receiving independent bytes of information (*sndack, bcast, recack*), and utility (*getid, getfl*).

The instructions for sending and receiving packets are used to build network packet flits according to the format given in Fig.2.12 and transmit them over the *packet* network described in the previous section. All of the flits generated from this category of instructions are 17 bits wide. Each flit carries a control bit that indicates to the NoC logic that a flit is a header/tail or body flit. The remaining 16-bits are data or control information, depending on whether it is a header/tail or body flit. The header

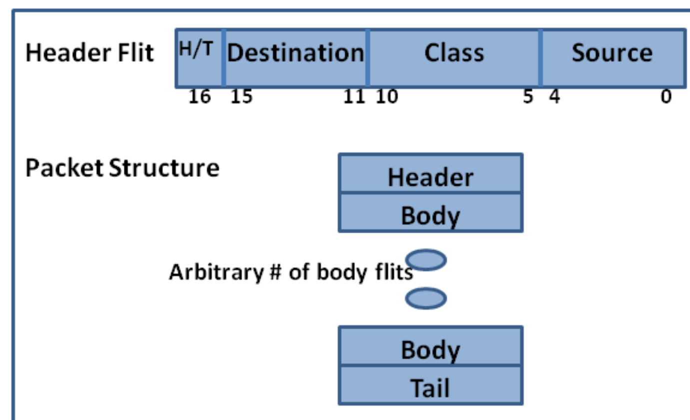

Figure 2.12. Packet Structure

has fields for the destination address, source address, and a packet class or type. The source and destination fields are provide through the two operands to the send header instruction. The packet class however, is taken from the function field of the send header instruction which can be specified by appending the instruction with one of the four options (.b, .s, .i, .l). Packet classes are used by the receiver to interpret the body flits. Body flits can represent raw bytes of information, portions of 32 and 64-bit data types, or pointers to buffers in shared memory. Therefore, the four send header options refer to buffer, short, integer, and long types respectively. A packet is terminated by called the send tail instruction, which simply sends another header with a packet class that is known to the network as a packet terminating token. Receiving these flits on the other end is accomplished by calling one of the receive instructions. Each of these removes incoming flits from the receive buffer and stores the data into a register. The receive word conditional instruction is unique in that it returns the value provided in the operand when there is either no data to receive or an error occurred.

The instructions for sending and receiving small bytes of information utilize the *acknowledge* network and are used primarily for synchronization. For example, when a packet has been received the receiver can notify the sender by sending an acknowledgement. Likewise, if an event occurs that all cores must be aware of, a broadcast can

ISA Extension	Alternative Solution

START:
li r1, 2
li r2, 1
li r3, 100
li r4, 101
 	
sndhd.s r5, r1, r2
sndw r5, r3
sndw r5, r4
sndtl r5
 	
bnez r5, START
	START:
	li r1, 2
	li r2, 1
	li r3, 100
	li r4, 101
	
	li r5, CONTROL_BASE
	sw r1, 0(r5)
	sw r2, 1(r5)
	li r6, SEND_PORT
	sw r3, 0(r6)
	sw r4, 0(r6)
	sw zero, 0(r5)
	sw zero, 1(r5)
	li r7, STATUS_BASE
	lw r3, 0(r7)
	
	bnez r3, START

Figure 2.13. ISA Example

be used to provide notification of that event. More importantly, these operations can be accomplished very quickly without flooding the *packet* network with traffic.

The two utility instructions provide simple features that are important for implementing any real system. The *getid* instruction reads a hardcoded core identifier into a register. This allows low-level software to ensure that certain functions execute on specific cores. The *getfl* instruction is used to obtain critical network operation parameters such as send/receive buffer status, errors, and the availability of received data.

To illustrate the advantages provided by the use of an ISA extension, consider an alternative in which memory addresses are mapped to control/status registers and send/receive ports inside the NIU. Using these registers, one could avoid the ISA extension by using load and store instructions. The psudo-assembly code in Fig.2.13 shows how one would send a packet with 2 body flits using both solutions.

It is clear that the ISA extension will result in faster code, completing the same operation in less than half as many instructions while using 2 fewer registers. In addition,

the hardware cost of adding these instructions to a standard MIPS processor is negligible. Sect.2.5 provides synthesis results that indicate the number of FPGA look-up-tables, flip-flops, and the clock period of the MIPS core with and without the ISA extension. Note that this does not include the NIU, which would be present in both solutions. Indeed, adding the memory-mapped control registers would also add logic to the base processor. The ISA extension also improves code portability since these memory mapped addresses will typically be device specific. Therefore, it is clear that the ISA extension has the advantage.

2.4 Tool-Chain

Programs can be built targeting XUM using a modified GNU tool-chain. After modifying the assembler source the ISA extension can be assembled into the MIPS executable, but without extending the C language the instructions can only be invoked via in-line assembly code. This is the method used in the implementation of MCAPI, discussed in Chap.3.

In Chap.1, the advantages and disadvantages of distributed versus shared memory were discussed. XUM utilizes a distributed memory architecture. Therefore, it is subject to one of the greatest weaknesses of such a configuration; which is poor memory utilization. This is particularly problematic in embedded systems (which XUM is targeted for). To address this problem, a custom linker script was written which defines the memory regions listed in Tab.2.8. Unique to this memory configuration are the local sections. Each local section is declared in an address space that gets mapped to the same physical region. This is because the linker is able to allocate data structures in address space that doesn't actually exist on the device. XUM is limited to 8KB of block RAM per core. Therefore, each local section occupies the first 4KB of it's associated core's block RAM since the higher order bits of the address are ignored. Data structures that get mapped into these sections appear to be in different memory regions, but in fact they are the same. This enables a C program to declare a global data structure with the attribute modifier, which maps the data structure into one of the eight local regions. Since only the address

Table 2.8. Address Space Partitioning

Section	Base	Size	Visibility
BSS	0x1000	2KB	All
Stack/Heap	0x17FE	2KB	All
Local0	0x0000	4KB	Core 0
Local1	0x2000	4KB	Core 1
Local2	0x4000	4KB	Core 2
Local3	0x6000	4KB	Core 3
Local4	0x8000	4KB	Core 4
Local5	0xA000	4KB	Core 5
Local6	0xC000	4KB	Core 6
Local7	0xE000	4KB	Core 7
Text	0x10000	4KB	All

space from 0x0000 to 0x2000 gets replicated across all cores, data structures placed in one of these local sections only use memory belonging to one core.

In the current version of XUM, programs are loaded by copying the text section of the executable into a VHDL initialization of a block RAM. This block RAM is then used as the instruction memory. This means that the project must be re-synthesized in order to run new programs. It also means that the initialized data section does not get used, so initialized global variables are not allowed. Obviously this is not ideal for any real use. An extension which allows executables to be dynamically loaded over a serial interface is planned for a future release.

2.5 Synthesis Results

All XUM modules are implemented in either VHDL or Verilog and are fully synthesizable. The target is a Xilinx Virtex-5 LX110T FPGA. This device was chosen because of it's massive amount of general purpose logic and block RAM resources. The top level XUM module consists of one 8-tile block, a simple timer, UART serial interface, and a 2x16-character LCD controller. Tab.4.2 gives the device utilization and clock speeds for the whole system, as well as some of the larger sub-systems. For comparison, the synthesis results of a basic MIPS core without the ISA extension or NIU is also

Table 2.9. Results - Logic Synthesis

Module	Registers (%)	LUT (%)	BRAM (%)	Max Clock Rate
Entire System	15523 (22%)	21929 (31%)	52 (35%)	117 MHz
Tile (w/ISA Ext.)	1673	2282	7	108 MHz
Tile (w/o ISA Ext.)	1589	1941	7	105 MHz
NIU	57	283	0	172 MHz
CacheCtrl	788	1026	5	194 MHz
Router (Pkt)	123	352	0	236 MHz
Router (Ack)	115	384	0	188 MHz

given. This is to indicate that the additional cost of extending the MIPS ISA is minimal (discussed in Sect.2.3).

It is clear that even though XUM is a large design, most of the device remains unused. This is to promote future extensions. There are sufficient resources to include an additional XUM block, several new I/O interfaces, and a SDRAM controller. These are all important extensions that will truly make XUM a cutting edge multicore research platform.

CHAPTER 3

MULTICORE COMMUNICATION API (MCAPI)

3.1 MCAPI Introduction

Coordination between processes in a multicore system has traditionally been accomplished either by using shared memory and synchronization constructs such as semaphores or by using heavy weight message passing libraries built on OS dependent I/O primitives such as sockets. It is clear that shared memory systems do not scale well. This is largely due to the overhead associated with maintaining a consistent global view of memory. Likewise, message passing libraries such as MPI have been in existence for many years and are not well suited for closely distributed systems, such as a multi-processor system-on-chip (MPSoC). This is especially true in embedded systems where lightweight software is very important in achieving good performance and meeting real-time constraints.

This dilemma has resulted in the development of a multicore communication API (MCAPI), recently released by the Multicore Association [17]. The API defines a set of lightweight communication primitives, as well as the concepts needed to effectively develop systems around them. These primitives essentially consist of connected and unconnected communication types in both blocking and non-blocking varieties. MCAPI aims to provide low latency, high throughput, low power consumption, and a low memory footprint.

However, the ability of MCAPI to achieve all of this depends on the implementation of its transport layer. Without careful consideration of new designs with MPSoCs in mind, the result will be another ill suited communication library. Just as distributed systems have downsized from clusters of server nodes interconnected over Ethernet to multicore chips with cores interconnected by NoCs, so too must libraries built on

complex OS constructs be downsized to bare-metal functions built on new processor features such as a communication oriented ISA extension.

The ISA extension provided by XUM is used for implementing the transport layer of MCAPI. Such a feature allows the implementation to avoid one of the major performance limitations of existing systems. That is, the implicit sharing of data. In a shared memory system, a load or store instruction is used to initiate a request for a cache line. The fulfillment of this request is dependent on very complicated memory management protocols which are hardwired into the system by the hardware manufacturer in order to maintain a consistent global view of memory. While convenient for the programmer, they may or may not be what is best for application performance. Likewise, in a traditional distributed system a request for data is initiated by a call to an MPI function. However, there is a lot going on ”under the hood” that is not under the control of the application. XUM's ISA extension allows a programmer to explicitly build a packet and transmit it over a NoC to a specified endpoint. This allows an MCAPI implementation, an OS, or bare-metal applications to share data and coordinate between processes utilizing custom algorithms which best meet the needs of the application. It also eliminates the overhead of cache hardware or a TCP/IP implementation.

In this release of XUM, a partial implementation of the MCAPI specification is provided. The interface to this API subset is given in Tab. 3.1. The techniques used in this implementation are described in this section. There are also strategies for implementing the remainder of the API given here.

3.2 Transport Layer Implementation

3.2.1 Initialization and Endpoints

The type of communication provided by MCAPI is point to point. In this implementation endpoints correspond to XUM tiles. Endpoints are grouped with other information that is vital to the correct operation of the various API functions. Fig. 3.2 shows how endpoints are represented at the transport layer.

A *trans_endpoint_t* is a structure consisting of an endpoint identifier, an endpoint attribute, and a connection. An endpoint identifier is a tuple of a node and a port,

```
mcapi_boolean_t mcapi_trans_initialize(mcapi_uint_t node_num);

mcapi_boolean_t mcapi_trans_finalize();

mcapi_boolean_t mcapi_trans_create_endpoint(mcapi_endpoint_t *endpoint,
                                            mcapi_uint_t port_num);

void mcapi_trans_get_endpoint(mcapi_endpoint_t *endpoint,
                              mcapi_uint_t node_num,
                              mcapi_uint_t port_num);

void mcapi_trans_delete_endpoint(mcapi_endpoint_t endpoint);

void mcapi_trans_get_endpoint_attribute(mcapi_endpoint_t endpoint,
                                        mcapi_uint_t attribute_num,
                                        void* attribute,
                                        size_t attribute_size);

void mcapi_trans_set_endpoint_attribute(mcapi_endpoint_t endpoint,
                                        mcapi_uint_t attribute_num,
                                        const void* attribute,
                                        size_t attribute_size);

mcapi_boolean_t mcapi_trans_msg_send(mcapi_endpoint_t send_ep,
                                     mcapi_endpoint_t receive_ep,
                                     char* buffer,
                                     size_t buffer_size);

mcapi_boolean_t mcapi_trans_msg_recv(mcapi_endpoint_t receive_ep,
                                     char* buffer,
                                     size_t buffer_size,
                                     size_t* received_size);

mcapi_uint_t mcapi_trans_get_node_num();
```

Figure 3.1. MCAPI Subset

which respectively refer to a tile and a thread running on that tile. An endpoint attribute is a variable that can be used in future extensions to direct buffer management. The connection structure is used to link a remote endpoint and associate a channel type for connected channel communication. In MCAPI, connected channels are used to provide greater throughput and lower latency, at the expense of higher cost and less flexibility. The current version of this work does not implement this part of the API. However, a discussion of implementation strategies for future versions is included in Sect. 3.2.3.

```
struct connection_t {
     mcapi_endpoint_t endpoint;
     channel_type type;
     mcapi_boolean_t open;
};

typedef struct {
     mcapi_endpoint_t endpoint;
     mcapi_uint_t attribute0;
     struct connection_t *conn;
} trans_endpoint_t;

volatile trans_endpoint_t endpoints[NUM_NODES][NUM_PORTS_PER_NODE];
```

Figure 3.2. Data Structures

Each XUM tile maintains an array of *trans_endpoint_t* structures. Since XUM is a distributed memory system, each tile sees a different version of this data structure. Some of the *trans_endpoint_t* members must be kept coherent for correct system operation, while others are only used by the local process. The *endpoint*, $conn \rightarrow endpoint$, and $conn \rightarrow open$ fields must be kept coherent across the whole system. This will be made clear in the discussions on connected and connectionless communication.

Notice that this data structure is declared with the *volatile* modifier. This is because coherence is maintained through an MCAPI interrupt service routine (ISR), with an interrupt being triggered by the *acknowledge* NoC via the *bcast* instruction. The ISR is given in Fig. 3.3. Note that this routine only includes code for managing coherence of the endpoint field, since connections are not supported in this version. The ISR receives data from the *acknowledge* network, identifies the data as an endpoint, and updates the local data structure. If the endpoint referred to has already been created then it is deleted, otherwise it is set to the received value.

Initialization of the *endpoints* array and registration of the ISR with the interrupt controller is done in the MCAPI initialize routine, shown in Fig. 3.4. All *trans_endpoint_t* (from this point referred to as endpoint) fields are first set to known values. This enables the ISR and other functions to know whether an endpoint has been created or not. This leads to the create endpoint code given in Fig. 3.5. This function creates an

```
void mcapi_ISR() {
  mcapi_uint_t icause, bc, cmd, dat;
  mcapi_uint_t msg;
  mcapi_uint_t node_num, port_num;
  trans_endpoint_t temp;
  volatile trans_endpoint_t* ep;

  asm volatile("mfc0 %0, $0" : "=r"(icause) : );
  asm volatile("recack %0" : "=r"(msg) : );

  bc = (msg & MASK_ACKSPC);
  cmd = (msg & MASK_ACKCMD);
  dat = msg & MASK_ACKDAT;

  if ((icause & MASK_INT_NOC1) != 0) {
    if (bc != 0) {
        if (cmd == 0) {
          node_num = dat >> BITS_FOR_PORT;
          port_num = dat & MASK_PORT;
          ep = &endpoints[node_num][port_num];
          temp.attribute0 = 0;
          temp.conn = UNCONNECTED;
          if (ep->endpoint == NULL_ENDPOINT) {
            temp.endpoint = dat;
          }
          else {
            temp.endpoint = NULL_ENDPOINT;
          }

          endpoints[node_num][port_num] = temp;
        }
    }
  }
}
```

Figure 3.3. Interrupt Service Routine

endpoint on the local node with the given port number by concatenating the node and port identifiers to form the endpoint tuple, updating the local *endpoints* data structure, and then broadcasting the coherent data to all other XUM tiles. In order to send a message to another tile, the endpoint on the receive end must be retrieved by calling the MCAPI get endpoint function, shown in Fig. 3.6. This function simply waits until the desired endpoint is created and the local *endpoints* data structure is updated by the

```
mcapi_boolean_t mcapi_trans_initialize(mcapi_uint_t node_num)
{
	mcapi_uint_t i, j;
	trans_endpoint_t temp;
	temp.endpoint = NULL_ENDPOINT;
	temp.attribute0 = 0;
	temp.conn = UNCONNECTED;
	for (i = 0; i < NUM_NODES; i++) {
	   for (j = 0; j < NUM_PORTS_PER_NODE; j++) {
	       endpoints[i][j] = temp;
	   }
	}

	OS_DisableInterrupts();
	OS_RegisterISR(INT_NOC1, mcapi_ISR);
	OS_EnableInterrupts();

	return MCAPI_TRUE;
}
```

Figure 3.4. MCAPI Initialize

```
mcapi_boolean_t mcapi_trans_create_endpoint(mcapi_endpoint_t *endpoint,
                                            mcapi_uint_t port_num)
{
	trans_endpoint_t temp;
	mcapi_uint_t node_num = mcapi_trans_get_node_num();
	mcapi_uint_t id = node_num;
	id = id << BITS_FOR_PORT;
	id = id | port_num;
	temp.endpoint = id;
	temp.attribute0 = 0;
	temp.conn = UNCONNECTED;
	endpoints[node_num][port_num] = temp;
	(*endpoint) = id;
	asm volatile("bcast %0" : : "r"(id));
	return MCAPI_TRUE;
}
```

Figure 3.5. MCAPI Create Endpoint

ISR. The application must ensure that it does not attempt to get and endpoint that is never created. This would result in a stalled process.

```
void mcapi_trans_get_endpoint(mcapi_endpoint_t *endpoint, mcapi_uint_t node_num,
                              mcapi_uint_t port_num)
{
    volatile trans_endpoint_t* ep = &endpoints[node_num][port_num];
    while (ep->endpoint == NULL_ENDPOINT) {};
    (*endpoint) = ep->endpoint;
}
```

Figure 3.6. MCAPI Get Endpoint

3.2.2 Connectionless Communication

The most flexible form of communication provided by MCAPI is a connectionless message. This type of communication also requires the least amount of hardware resources, implying that it generally will have a minimal adverse affect on the performance of other communicating processes in the system. Both blocking and non-blocking varieties of message send and receive functions are discussed here, though the current version of XUM only supports blocking calls.

The MCAPI blocking message send function is given in Fig. 3.7. The function first checks the status of the *network busy* flag. If this flag is set it implies that the send buffer is full and the application needs to back off of the network to allow the network time to consume data. To ensure that the tail follows immediately after the body of the packet, the bulk of the function is executed with interrupts disabled. Notice that the two operand variant of the *sndw* instruction is used. This allows two elements of the buffer to be sent in one cycle. If any send instruction fails, due to a backed-up network, then the function simply attempts to send the same bytes again. Also, the throughput of this function could be improved by sending multiples of 2 bytes at a time with several sequential calls to the *sndw* instruction within the loop body. However, this would mean that the buffer size would need to be aligned by the number of bytes sent in each loop iteration. Future extensions could utilize the endpoint attribute to specify the byte alignment of buffers to enable this optimization. After the entire buffer has been sent, the function terminates the message sequence with a tail and waits for the acknowledgement. Waiting for the acknowledgement provides synchronization between the two processes.

```
mcapi_boolean_t mcapi_trans_msg_send( mcapi_endpoint_t send_ep,
                mcapi_endpoint_t receive_ep, char* buffer, size_t buffer_size)
{
  mcapi_uint_t i = 0, res = 0, flag = 0, ack = 0;
  volatile trans_endpoint_t* ep;
  mcapi_uint_t node_num = get_node(send_ep);
  mcapi_uint_t port_num = get_port(send_ep);

  asm volatile ("getfl %0,%1" : "=r"(flag) : "i"(FLAG_NETBUSY));
  if (flag == 1)
     return MCAPI_FALSE;

  ep = &endpoints[node_num][port_num];

  OS_DisableInterrupts();

  asm volatile ("sndhd.s %0,%1,%2":"=r"(res):"r"(receive_ep),"r"(send_ep));
  while (i < buffer_size) {
     asm volatile ("sndw %0,%1,%2":"=r"(res):"r"(buffer[i]),"r"(buffer[i+1]));
     if (!res)
       i += 2;
  }
  asm volatile ("sndtl %0" : "=r"(res) : );

  while (flag == 0) {
     asm volatile ("getfl %0,%1" : "=r"(flag) : "i"(FLAG_ACK));
  }
  asm volatile ("recack %0" : "=r"(ack) : );

  OS_EnableInterrupts();
  return MCAPI_TRUE;
}
```

Figure 3.7. MCAPI Message Send

The MCAPI blocking message receive function is given in Fig. 3.8. A blocking receive starts by waiting for a message by checking the status of the *scalar available* flag. Note that since the current XUM release uses a purely distributed memory system, the type of data transmitted through MCAPI messages is scalar data. Pointer passing will be made possible in a future release. Once data is available, the header is retrieved and the function enters a loop that utilizes the *recw.c* instruction, which provides error checking by returning the value of the operand when an error has occurred. The program can then check the flags to discover the error source. For a message receive, there are

two sources of error. Either the network receive queue is empty, or a tail has been seen and the message sequence has completed. In the later case, the receive function discovers the arrival of a tail by checking the *receiver idle* flag, which is only non-zero between packets. If the tail has not been seen, then the buffer index is backed up and the function attempts to receive more data. Once the entire message has been received then an acknowledgement is sent to the sending endpoint.

Non-blocking variants of these functions are very similar. A non-blocking send would return after the tail of the packet has been queued up in the send buffer, instead of waiting for an acknowledgement. This generally will be very quick, unless the message is large enough to exceed the capacity of the send buffer. In this case, the function wouldn't return until the message starts to move through the network. Additional data structures would need to be added for tracking the status of MCAPI requests. A non-blocking send would create a request object. Then when the acknowledgement is received, which would cause an interrupt, the ISR could check for pending requests corresponding to the endpoint from which the acknowledgement was received. The request would then be satisfied. The non-blocking receive function would simply create a request object and return. When a packet is received, an interrupt would trigger and the ISR would see that a receive request was created. It would then call a function that would look similar to the blocking call to satisfy the request. Non-blocking functions are advantageous because they would permit message pipelining. In other words, a second message could be sent before the first one completes. This would improve throughput and network utilization. However, it complicates synchronization and introduces the possibility of deadlock. This makes programming more difficult. However, a formal verification tool for checking MCAPI applications is under development and will ease the task of writing correct MCAPI applications [28].

3.2.3 Connected Channels

Connected communication channels are different than connectionless messages in that they form an association between two endpoints prior to the transmission of data between them. For this reason, they are obviously less flexible than connectionless mes-

```
mcapi_boolean_t mcapi_trans_msg_recv( mcapi_endpoint_t  receive_endpoint,
                  char* buffer, size_t buffer_size, size_t* received_size) {
   mcapi_uint_t flag1 = 0;
   mcapi_uint_t i = 0, recHeader = 0, tmp = 0;
   mcapi_uint_t err = RECW_ERR_CONS;
   OS_DisableInterrupts();
   while (flag1 == 0) {
      asm volatile ("getfl %0,%1" : "=r"(flag2) : "i"(FLAG_SCLAVAIL));
   }

   asm volatile ("rechd %0" : "=r"(recHeader) : );
   while (1) {
      while (tmp != RECW_ERR_CONS && i < buffer_size) {
          asm volatile ("recw.c %0,%1" : "=r"(tmp) : "r"(err));
          buffer[i] = (tmp & 0x00FF);
          buffer[i+1] = (tmp >> 8);
          i += 2;
      }

      asm volatile ("getfl %0,%1" : "=r"(flag1) : "i"(FLAG_RECVIDL));
      if (tmp == RECW_ERR_CONS) {
         i -= 2;
         buffer[i] = 0;
         buffer[i+1] = 0;
         if (flag1)
            break;
      }
      else {
         if (flag1)
            break;
         OS_EnableInterrupts();
         return MCAPI_FALSE;
     }
     (*received_size) = i;
   }
   recHeader = recHeader & MASK_SENDPOINT;
   recHeader = recHeader >> BITS_FOR_PORT;
   asm volatile ("sndack %0" : : "r"(recHeader));
   OS_EnableInterrupts();
   return MCAPI_TRUE;
}
```

Figure 3.8. MCAPI Message Receive

sages. However, they do provide an opportunity for greater throughput and lower latency, depending on the implementation. Given the XUM architecture, the implementation strategy for connected channels is very similar to that of connectionless messages. In a

sense, sending a header opens a channel and sending a tail closes a channel. Channels simply decouple the process of sending headers/tails from sending data. Since a header arbitrates through the on-chip network and reserves routing channels it is the only part of a packet that must really be subject to significant network latency. By doing this decoupling, a programmer can reserve network resources in anticipation of transferring data. After a channel is opened, subsequent calls to send data can be done very quickly without contention on the network.

There is however, a significant cost associated with doing this. The two endpoints that are connected will enjoy a fast link, but other pairs of endpoints could suffer in their quality of service. Consider a situation in which a channel is open between two endpoints and another message needs to temporarily use a link occupied by this channel. This message could get stalled for a very long time, especially if the programmer is not attentive to releasing resources by closing the channel.

One possible solution to this is to use virtual channels in the NoC routers. Virtual channels are discussed in Chap. 4. It is sufficient though to simply state that adding virtual channels will increase the complexity of the routers and require more hardware to implement. In addition, the problem will still exist as the number of MCAPI channels used increases. Therefore, the following algorithm is proposed as a solution (see Fig. 3.9). Since the channel status (open or closed) and receiving endpoint are fields in the MCAPI data structure that are kept coherent across all tiles, all processes are aware of channel connections. In addition, since the routing scheme used by the *packet* network is a deterministic dimension order routing algorithm, all processes can determine which links are occupied by an open channel based on the channel's start and end points. Therefore, if an open channel will prevent a message from being sent (step 1) the message sender needs only to send a *close channel notification* to the channel start point (step 2). The message can then be sent across the link that was just occupied by the open channel (step 3). When the message has been received an *open channel notification* can then be sent to the channel start point (step 4). This re-opens the channel for use by the send point (step 5). The advantages of this technique are that messages can still be sent in the presence of open channels. The disadvantage is that data transmission

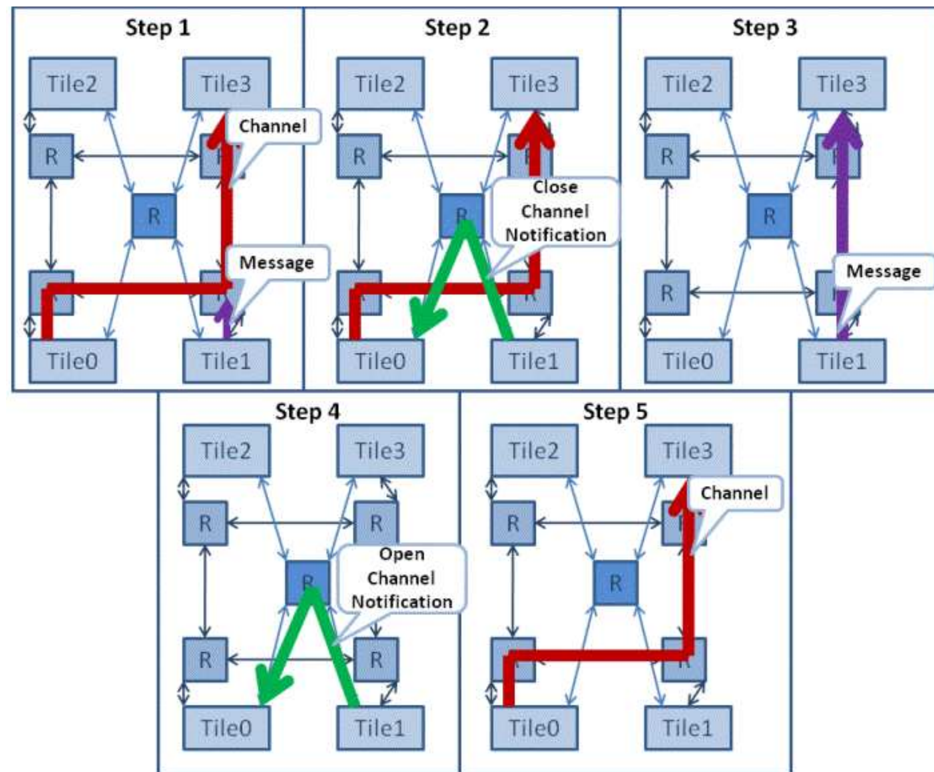

Figure 3.9. Algorithm for Managing Connected Channels

over open channels will occasionally be subject to a lag in performance. However, since messages generally only occupy links for a relatively short time, this lag will not be a significant limitation. Even more so, open channels often sit idle while the application operates on independent sections of code in between data transmission. This means that in practice the difference in average case performance of a channel with and without this implementation technique will be negligible.

3.3 Performance and Results

This MCAPI implementation is very unique in that it is built on a custom hardware platform, uses hardware primitives to perform communication, and runs below the operating system layer. Due to this, it is impossible to fairly compare the performance of this implementation with any existing message passing system. The only other available MCAPI implementation relies on shared memory and cannot run on XUM. In addition,

there are many factors that affect performance. XUM uses simple processing cores without branch prediction or complicated out-of-order instruction schedulers, which affects instruction throughput. XUM also runs on an FPGA with clock speeds much slower than fabricated ASIC's. The results reported in this section are given at face value. Little comparison to alternative solutions is given for the sake of comparing "apples to apples." The comparisons that are made are only for reference. No conclusions can be drawn since they are based on very different systems.

All throughput and latency measurements apply to the application layer. They are affected by, but do not represent the maximum throughput or latency possible by the XUM NoC. Latency is measured by counting the number of clock cycles from the time the function is called until it returns. Throughput is measure by calling send and receive functions on separate nodes within an infinite loop. The number of cycles that occur between the beginning of the first receive function and the end of the 10th is used to divide the total number of bytes transmitted in that period. Note that since these are blocking calls messages are not pipelined.

3.3.1 Hypothetical Baseline

Though no real baseline system exists for comparison, this section provides a brief description of a hypothetical baseline system from which this work would improve upon. The ideal baseline system would be an alternative MCAPI implementation running on XUM without the instruction-set extension. Polycore Software Inc. produces an MCAPI implementation called Poly-Messenger [26]. This MCAPI implementation is integrated with the RTXC Quadros RTOS from Quadros Systems [27]. This system running on XUM would provide the ideal baseline. While the Quadros/Polycore solution provides lightweight message passing for embedded systems it is not tied to any particular hardware platform. Therefore, it does not take advantage of custom hardware acceleration for message passing. Comparing with this baseline would allow for accurate evaluation of the XUM ISA extension's effectiveness at enabling fast and lightweight implementations of MCAPI.

Table 3.1. MCAPI Memory Footprint

Object File	Code Size	Data Size
mini_mcapi.o	1504 B	192 B
xum.o	244 B	28 B
boot.o	516 B	0 B
Total	2264 B	220 B

3.3.2 Memory Footprint

One very important parameter of any MCAPI implementation is its memory footprint. This is due to the fact that MCAPI targets embedded systems, where memory constraints are often very tight. A large message passing library like MPI is not well suited for most embedded devices simply due to its significant memory requirements.

This MCAPI implementation is able to achieve a very low-memory footprint, thanks in part to the XUM ISA extension. Tab. 3.1 lists the code size in bytes of all object files needed to run a bare-metal MCAPI application. Also important is the size of global data structures used by the system software. This information is also given in Tab. 3.1. Only statically allocated data is used by MCAPI and the XUM system software.

3.3.3 Single Producer - Single Consumer

In order to evaluate the communication performance of this MCAPI implementation several test cases were developed to model different communication scenarios. Each scenario stresses a different part of the system. Best and average case performance in terms of latency and throughput is evaluated. The first scenario models the best case in which there is a single producer, a single consumer, and no other contention for network resources. All tests require that producers generate message data and then send it repeatedly, while consumers repeatedly receive these messages. Tab. 3.2 gives the average latency of MCAPI calls while Tab. 3.3 gives the throughput.

3.3.4 Two Producers - Single Consumer

This example is similar to the prior case but two producers are used to simultaneously send messages to one consumer. The cores that operate as producers and consumers are

Table 3.2. MCAPI Latency: 1 Producer/1 Consumer

Function	10B Msg	20B Msg	30B Msg	40B Msg
Send	180 cycles	250 cycles	320 cycles	390 cycles
Recv	182 cycles	245 cycles	322 cycles	385 cycles

Table 3.3. MCAPI Throughput: 1 Producers/1 Consumer

	10B Msg	20B Msg	30B Msg	40B Msg
Throughput	0.053B / cycle	0.078B / cycle	0.091B / cycle	0.101B / cycle

Table 3.4. MCAPI Latency: 2 Producers/1 Consumer

Function	10B Msg	20B Msg	30B Msg	40B Msg
Send	265 cycles	405 cycles	545 cycles	685 cycles
Recv	126 cycles	196 cycles	266 cycles	336 cycles

chosen such that their respective routing paths on the *packet* network share a link. This is to model a more realistic scenario. Most parallel programs that run on XUM will share network links between multiple communicating paths. Tab. 3.4 gives the average latency of MCAPI message send and receive calls for the given packet sizes. Tab. 3.5 gives the throughput. Since throughput is measured per communication channel, the results are half of the total number of received bytes per cycle since there are two channels.

3.3.5 Two Producers - Two Consumers

In the previous example the performance bottleneck is clearly the consumer. A more common scenario that puts the stress on the network is a two producer/two consumer case in which the communicating paths share a link. This test assigns tiles 0 and 6 as

Table 3.5. MCAPI Throughput: 2 Producers/1 Consumer

	10B Msg	20B Msg	30B Msg	40B Msg
Throughput	0.037B / cycle	0.049B / cycle	0.055B / cycle	0.058B / cycle

Table 3.6. MCAPI Latency: 2 Producers/2 Consumer

Function	10B Msg	20B Msg	30B Msg	40B Msg
Send	188 cycles	258 cycles	330 cycles	400 cycles
Recv	187 cycles	254 cycles	322 cycles	398 cycles

Table 3.7. MCAPI Throughput: 2 Producers/2 Consumer

	10B Msg	20B Msg	30B Msg	40B Msg
Throughput	0.051B / cycle	0.075B / cycle	0.088B / cycle	0.094B / cycle

one producer/consumer pair and tiles 1 and 3 as another. Given the X/Y dimension-order routing scheme used by the XUM *packet* NoC, the link from router 1 to 2 will be shared by both routing paths. In this example, the performance bottleneck is the router's ability to arbitrate between two messages for one link. Tab. 3.6 provides the results for latency while Tab. 3.7 gives the throughput. Note that the throughput results given are the average of the two communication paths.

3.3.6 Discussion of Results

These results indicate that this implementation of MCAPI holds to the aspirations of the API specification. It has a very low memory footprint of only 2484 bytes (2264B code, 220B data). This is off course, only a subset of the transport layer. The full transport layer would be about 2 times that. In addition, the full API also includes a fair amount of error checking code above the transport layer. A full MCAPI implementation on XUM would probably require about 7-8KB of memory. This is a reasonable amount for most embedded systems.

One very important objective of this MCAPI implementation is to provide the ability to exploit fine grained parallelism. In a large scale distributed system using MPI, good performance is only achieved when there are large chunks of independent data and computation such that the cost of communication can be justified. However, many applications (especially in embedded systems) do not have this attribute. For these systems, it must be possible to transmit small messages with low-latency and high throughput.

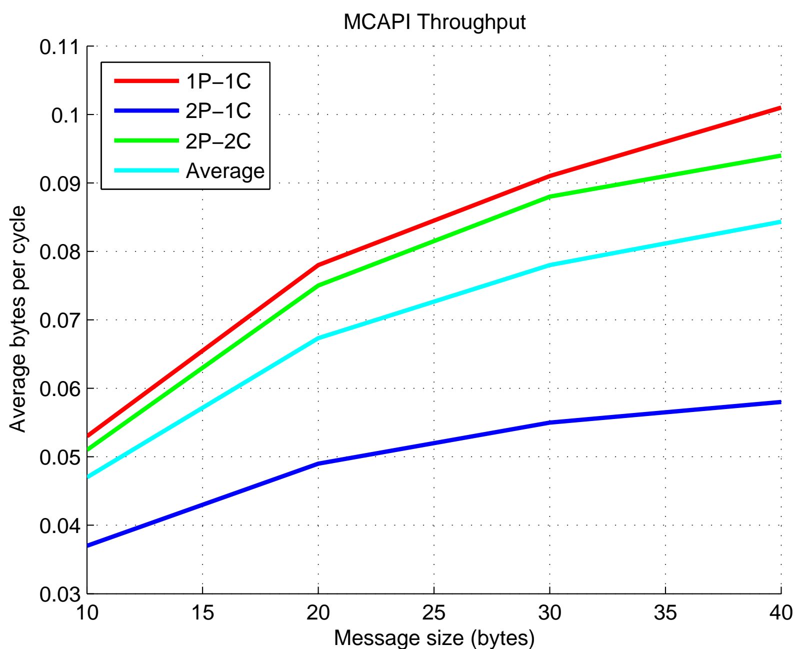

Figure 3.10. MCAPI Transport Layer Throughput

The average throughput for all scenarios and message sizes given in the previous sections is about 0.07 B/cycle. At a clock rate of 100MHz that is 7MB/S. Recall that this is the throughput that the application can expect to achieve with blocking message send and receive calls between two endpoints. It does not represent the throughput of the on-chip network. Fig. 3.10 shows a plot of the measured throughput for the various communication scenarios and their average. Notice that all throughput measurments increase as message sizes increase, but begin to taper off at around 40B message sizes. This happens sooner in scenarios where there is a shared link. For comparison, the maximum throughput possible in a point-to-point connection with no network contention on XUM is about 0.42B/cycle (42MB/S at 100MHz). MCAPI messages have a fair amount of overhead and will not exceed about 25-30% of this limit, regardless of message size. Non-blocking calls that can be pipelined will get closer to this limit, as will connected channels in a future release.

Perhaps even more important than throughput is the latency associated with calling MCAPI functions. The average latency of a blocking MCAPI function call is about 310

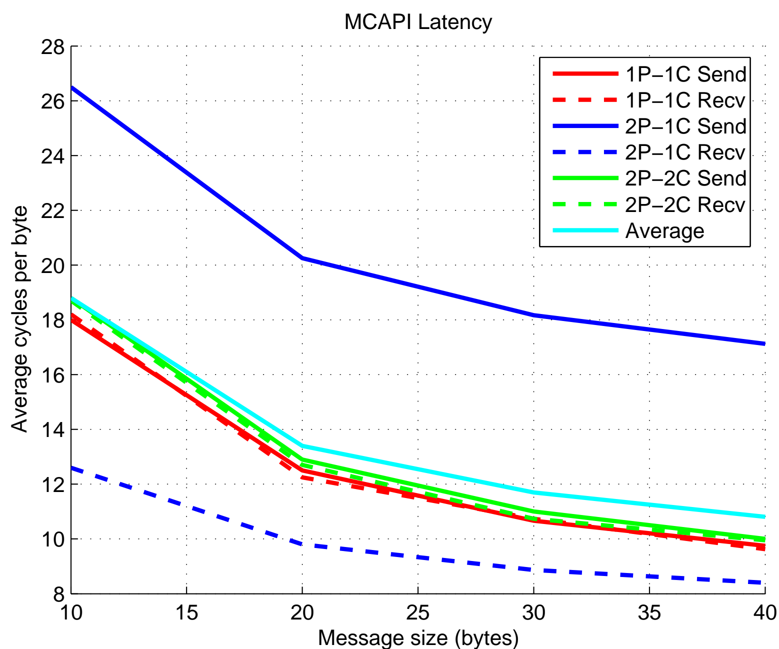

Figure 3.11. MCAPI Transport Layer Latency

cycles. This is assuming that there are no last level cache misses (XUM has no off-chip memory interface so this is guaranteed in these tests). The number of cycles required to send a byte of data goes down drastically as message sizes increase (as shown in Fig. 3.11). However, a lower limit of about 8-10 cycles per byte is reached.

For comparison with a recent high-performance system, consider the MPI performance results given in [24]. These results were obtained by running MPI on nodes with dual Intel quad-core processors running at 2.33 GHz with communication over Infiniband. For throughput, non-blocking message send and receive MPI calls were used. In addition, messages were pipelined. Within one of these Intel chips, throughput is measured at about 100-300 MB/S with message sizes from 16-64B. At 2.33 GHz, XUM would average between 110-196 MB/S with blocking calls (not pipelined) and message sizes between 10-40B. For latency, the Intel based system achieves about 0.4 us per blocking MPI call with message sizes under 64B. At 2.33 GHz, XUM would average about 0.133 us per blocking MCAPI call with message sizes between 10-40B. With this

baseline, MCAPI on XUM keeps up in terms of throughput and does much better in terms of latency, even though it is a far simpler design.

For comparison with a vanilla MCAPI implementation, consider the results given in [9]. These benchmarks are run on an example MCAPI implementation provided by the Multicore Association. It's transport layer is based on shared-memory. Therefore, it is fundamentally very different from the implementation presented in this work. However, for comparison the throuput results are plotted in Fig. 3.12. The results from [9] are obtained by running a producer/consumer test with blocking MCAPI message send/receive functions, similar to the test described in Sec. 3.3.3. The hardware used is a high-end Unix workstation with a 2.6 GHz quad-core processor. Results of the single producer/consumer test in Sec. 3.3.3 are plotted for comparison. The baseline data is a linear approximation of the data presented in [9]. It is clear from the figures that a significant improvement in bits per second is achieved for small message sizes. The improvement is even more significant considering the much slower clock rate of the test hardware. However, since the baseline MCAPI implementation is essentially a wrapper for shared-memory communication mechanisms it can not be fairly compared with a true closely distributed message passing system.

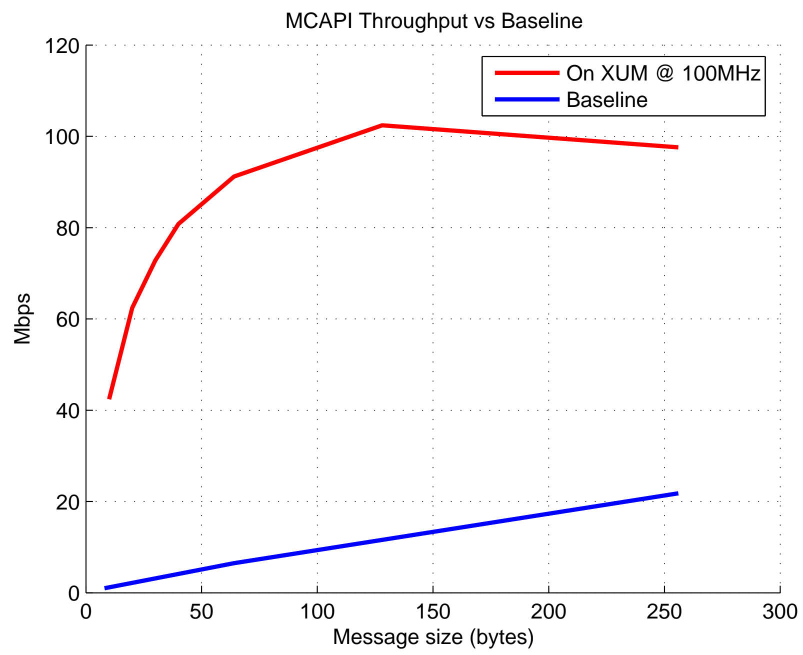

Figure 3.12. MCAPI Throughput vs Baseline

CHAPTER 4

APPLICATION SPECIFIC NOC SYNTHESIS

4.1 NoC Synthesis Introduction

Most of the current network-on-chip solutions are based on regular network topologies, which provide little to no optimization for specific communication patterns. The most highly concurrent chips on the market today lie in the embedded domain where chips are designed to run a small number of applications very well. In these cases, the on-chip communication demands are typically well known. Graphics and network packet processing chips are perfect examples of this, each using multiple cores to execute their intended application in a pipeline parallel manner. The communication demands are even more identifiable in heterogeneous chips where general purpose cores are integrated with DSP cores, along with various accelerators and I/O interfaces. It is obvious that in such systems the most efficient topology is one where the most frequently communicating devices are placed closest together.

Regular network topologies such as mesh, ring, torus, and hypercubes suffer from additional limitations. There are often under utilized links due to the routing function and network traffic patterns. This results in wasted power, chip area, and hurts overall network performance by over complicating the hardware. Regular topologies are also not very scalable in terms of communication latency (albeit much better than buses). Without some design consideration to workloads, the average number of network hops from one core to another will go up significantly as the number of cores increases.

However, the use of irregular networks on-chip (INoC) with heterogeneous cores introduces several complications. For example, deadlock free packet routing is more difficult to design and prove in such communication structures. Wire lengths are more difficult to manage compared to regular mesh topologies. If regular topologies are not

used then automatic synthesis of a custom topology for a given workload is essential to reducing design costs and time-to-market for various chip designs.

This chapter presents a synthesis tool that addresses these complications. The tool presented provides workload driven synthesis and verification of INoC topologies and their associated routing functions. The novel contribution is the approach to synthesizing INoC topologies and routing functions such that latency and power are minimized through reduced network hops. While this is not directly related to the primary objectives of this thesis, it is related to NoC technology and embedded systems. Therefore, it is consistent with the technological thrust of this body of work.

Other papers have been published in this area and are outlined with their various contributions and limitations in Sect. 4.2. In addition, various assumptions are made that effect the methodology presented here. These are described in Sect. 4.3. The algorithms used to generate a topology (Sect. 4.4) and deadlock free routing function (Sect. 4.5) are then presented. In addition, the physical placement of network nodes is briefly discussed in Sect. 4.6. An existing NoC traffic simulator is used and extended to simulate application specific traffic on the customized INoC topology. The extensions to this tool are described (Sect. 4.7). The synthesis results are given for several examples and compared with a baseline in Sect. 4.8.

4.2 Related Work

Most of the related work in this area takes one of two approaches to synthesizing a custom NoC for a given workload. The first approach is to analyze the communication demands of the workload, select a predesigned regular NoC topology, and then find an optimal mapping of communication nodes onto that topology. The work presented in [20] and [3] fall into this category. Both papers are based on a tool called SUNMAP, which implements this first approach to NoC synthesis. While these papers present very effective solutions, they are limited to predesigned topologies. Therefore, it is tha author's beleif that more efficient solutions are possible with irregular topologies; particularly when heterogeneous embedded systems are considered.

The second dominant approach to custom NoC synthesis is to start with an initial topology which ensures that a correct routing path exists, and then add links to improve performance for a workload. The work presented in [21] and [23] fall into this category. This approach does generate a custom irregular topology for the workload. However, the requirement of an initial topology that ensures a deadlock free routing path between all nodes is considerable overhead. Deadlock is a condition that may occur very infrequently. Therefore, it is inefficient to have dedicated hardware to handling this rare occurrence. It would be better if the links added for performance could themselves be fabricated into a topology that provides deadlock free routing by itself.

The work in [29] attempts to solve this problem, as does this paper. However, the optimization objectives of [29] are different than those of this paper. The objectives of [29] are to minimize power (primary goal) and area (secondary goal), while meeting the given architectural constraints. Our objectives are to minimize latency (primary goal) and power (secondary goal). Therefore, the algorithms presented here are different. There is value to studying both techniques as some applications may be more sensitive to one set of metrics than the other.

Our method offers the following features. It accepts a workload specification consisting of pairs of endpoints between which a communication channel is to be established, along with the estimated bandwidth and priority. Assuming a radix bound on the router at each node, the initial phase of our algorithm generates an optimum customized topology. It then enters a second phase where it determines the best deadlock-free routing by minimizing the number of virtual channel introductions in the routes chosen. To ensure that the performance gained by providing a direct link between the most frequently communicating nodes is not negated by wire delays, a core placement algorithm is then executed to find an effective placement of network nodes. Algorithmically, all of the individual phases of our algorithm run polynomially, and in practice we have generated networks with 430 channels and 64 nodes in under 20 seconds on a laptop PC.

This chapter presents the NoC synthesis problem at a very high level of abstraction. However, it forms a key part of the complete NoC design flow presented throughout this thesis.

4.3 Background

4.3.1 Assumptions

The following assumptions are made here; It is assumed that a workload is a set of on-chip communication requirements for a specific type of application. These requirements can be obtained by either profiling a real application or by a high level model of an application. Links are assumed to have uniform bandwidth. However, this work would provide an engineer implementing a network with the applications bandwidth requirements on individual links. Thus, enabling the engineer to make informed decisions about implementing non-uniform bandwidth links. Lastly, a network is referred to as consistent if it meets two conditions: there is a path between every pair of nodes in the network and that the shortest path between these nodes is free of a cyclic channel dependence that causes deadlock.

4.3.2 Optimization Metrics

An on-chip router generally consists of three main parts; buffers, arbiters, and a crossbar switch. The hardware complexity of a router is generalized to be proportional to the router radix; that is, the number of bidirectional links supported by the router. As the router radix goes up, the logic delay through the crossbar and arbitration logic goes up. Therefore, one synthesis objective is to minimize the average router size in the network by generating a topology with an upper bound on the maximum router radix.

In order to reduce power and latency, the number of network hops per flit must be minimized. This is the primary optimization objective. This is justified by the fact that the power per flit is proportional to the number of hops as well as the wire lengths traversed by the flits. Also, the latency per flit is proportional to the number of hops [4].

4.4 Irregular Topology Generation

4.4.1 Workload Specification

The topology generation algorithm will now be described starting with the input workload specification. The example workload is given in Tab. 4.1.

Table 4.1. Example Workload

Startpoint	Endpoint	Bandwidth	Priority
cpu1	cpu2	512	2
cpu1	cpu3	128	1
cpu3	cpu4	128	1
gpu1	gpu2	2048	3
gpu1	cpu3	256	1
gpu2	cpu3	256	1
dsp1	dsp2	2048	3
dsp1	gpu1	2048	2
dsp2	gpu2	2048	2
dsp1	cpu3	256	1
dsp2	cpu3	256	1
cpu1	cache1	1024	2
cpu2	cache1	1024	2
cpu3	cache2	1024	2
cpu4	cache2	1024	2
cache1	cache2	256	2
cache1	mem	256	2
cache2	mem	256	2
dsp3	io1	256	1
io2	dsp3	256	1
cpu4	dsp3	512	2
net	dsp3	512	2
cpu4	net	128	1
io3	mem	256	2
mem	dsp1	128	2
mem	dsp2	128	2

In this specification the first and second columns represent start and endpoints to model chip resource communication. The third column gives the desired bandwidth in MB/s between endpoints. The fourth column gives a priority associated with the communication path. A priority is used because a communication channel may not need much bandwidth, but it's time sensitivity requires that latency be minimized. The input workload is required to specify a path between every pair of endpoints, even if the desired bandwidth is 0. These lines are omitted from Tab. 4.1.

The links specified in the workload represent unidirectional paths and are assigned a level of optimization effort E calculated by 4.1. This equation states that the optimization effort is equal to the evenly weighted sum of the normalized bandwidth and priority.

$$E = \frac{BW}{BWnorm} + \frac{PR}{PRnorm} \tag{4.1}$$

4.4.2 Topology Generation

A network is represented as a graph data structure, where nodes represent routers and edges represent unidirectional links. The algorithm for generating a custom topology is a greedy algorithm. It is given in Alg. 1. Before *BuildNetwork()* is called, the communication channels specified by the workload are sorted according to their optimization effort. The result of this sort is the input to this algorithm. It starts by iterating through all channels. The channel endpoints are classified as nodes and are added to a network, if they are not already present.

Once all of the nodes have been added to the network the algorithm attempts to add links between them. It iterates through the channels in the order of their level of optimization effort. A shortest path computation is performed on the channel endpoints. Execution of the add link code is only done if there is no path between the two nodes, or if the there is a path and the communication bandwidth on that path is greater than a percentage of the maximum channel bandwidth. This prevents links from being added for channels where a path exists, but the number of flits that traverse that channel is low. Hence, the benefit of adding a link to minimize hops on that path is not justified.

Links are then added between two nodes such that the channel endpoint nodes will be connected. The ability to add a link depends on one condition, which is that the maximum router radix has not been reached. As mentioned in Sect. 4.3, network routers perform poorly as the radix increases. For a network topology to be implemented efficiently at the circuit level it must have small routers. Yet, in terms of network hops, the most best topology is one where all nodes have a direct link to every other node. Therefore, to balance both objectives a maximum radix must be specified.

Algorithm 1 Build Network

Require: $Channels \quad \leftarrow \quad set \quad of \quad all \quad c : c \in Workload \quad and \quad c \quad \leftarrow (N_s, N_e) : N\ is\ an\ endpoint\ node$
Require: $Network \leftarrow \emptyset$
  for all $c \in Channels$ **do**
    if $c.start \notin Network$ **then**
      $Network \leftarrow Network \cup c.start$
    end if
    if $c.end \notin Network$ **then**
      $Network \leftarrow Network \cup c.end$
    end if
  end for
  for all $c \in Channels$ **do**
    $path \leftarrow$ ***getShortestPath***$(Network.get(c.start), Network.get(c.end))$
    if $path.size() < 2 \cap path.size() \geq 2 \cup c.bandwidth > (MAXBW * TOL)$ **then**
      $depth \leftarrow ceil(c_prev.bandwidth * \frac{(c_prev.path.size()+COSTOFINSERT)}{c.bandwidth)}$
      $start \leftarrow$ ***getNextAvailableNode***$(Network.get(c.start), depth)$
      $end \leftarrow$ ***getNextAvailableNode***$(Network.get(c.end), depth)$
      if $start \neq NULL \cup end \neq NULL$ **then**
        $start.AddLink(end)$
      else if $start \neq NULL \cup start.HasTwoFreeLinks()$ **then**
        $link \leftarrow Network.get(c.end).GetLowestBWLink()$
        $Network(c.end).InsertNodeInLink(start, link)$
      else if $end \neq NULL \cup end.HasTwoFreeLinks()$ **then**
        $link \leftarrow Network.get(c.start).GetLowestBWLink()$
        $Network(c.start).InsertNodeInLink(end, link)$
      else
        return $ERROR$
      end if
    end if
    for all $n \in Path$ **do**
      $n.IncrementBW(n_next, c.bandwidth)$
    end for
  end for
  $Network.Cleanup()$
  return

Once the maximum radix is reached for a node *A*, it can no longer add a link to another node *B*. To establish a communication path between *A* and *B*, node *B* must link to another node that is already adjacent to node *A*. The algorithm uses a breadth first search with node *A* as the root to find an adjacent node to which node *B* may be linked. This functionality is provided by the function call to *getNextAvailableNode*. Since the channels are sorted initially, channels with the highest optimization effort are placed closest together while less important channels are subjected to traversing more network hops in order to minimize the hardware complexity.

Given a radix bound, it is possible that a network could not be connected by this greedy algorithm. Consider a case where three nodes *A, B,* and *C*, have been added to a network with a max radix of 2. Suppose node *D* needs to be added to the network. In this case, we use a heuristic that inserts this node into a higher priority path. The heuristic ensures that the added cost to the higher priority path will be tolerable. It is still possible for the algorithm to fail, but this reduces the number of cases in which failure would occur. If the algorithm fails, then the radix bound must be increased to generate the network.

Once a link has been added, the resulting path between the two endpoint node is traversed. The bandwidth used on the corresponding path links is incremented. This value is initially 0. At the end of the routine, all links that carry no bandwidth are removed from the network.

4.5 Routing Verification

Once topology has been generated a verification algorithm is executed to ensure that the shortest path between any two network nodes does not introduce a cyclic dependence that could result in a deadlock.

4.5.1 Deadlock Freedom

The method used to prove that routing paths are free of deadlock is modified from the method introduced in [21]. A channel dependency graph is generated, which is a directed graph where channels are characterized as either increasing or decreasing; depending on

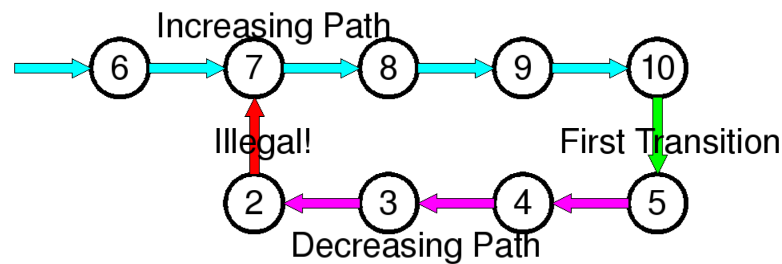

Figure 4.1. Channel Dependency Graph

a numbering of network nodes. A routing path is deadlock free if packets are restricted to traveling on only increasing channels and then decreasing channels. Packets are not allowed to travel on decreasing channels and then increasing channels, as shown in Fig. 4.1. This eliminates a cyclic channel dependence. Packets can however switch back to increasing channels if they switch to a higher virtual channel plane [4].

4.5.2 Routing Algorithm

Instead of numbering network nodes arbitrarily, the proposed method here numbers nodes as they are seen in a depth first search of the generated topology. This is an attempt to find a numbering that allows all communication paths to be consistent (given the aforementioned routing restriction) while leaving the generated network topology unchanged. The algorithm for routing the network is given in Alg. 2 and 3.

At this time no general purpose numbering scheme that results in the lowest cost solutions for all topologies is known. Therefore, the algorithm uses the depth first search numbering with a different root node for each iteration of the code in Alg. 2. The cost of the network is then calculated in terms of the total number of virtual channels in the resulting network.

After the network nodes have been numbered the *makeRoutingConsistent* function is called, as shown in Alg. 3. For each pair of nodes in the network the shortest path is computed using Dijkstra's algorithm. Then for each step in that path channels are characterized as either increasing or decreasing. Anytime there is a change from increasing to decreasing or visa versa a variable is incremented. If the number of changes

Algorithm 2 Route Network

Require: $Best \leftarrow Network$
  numberNodes(0, $Best$)
  makeRoutingCorrect($Best$)
  for all $n \in Best$ **do**
    $Temp \leftarrow Network$
    numberNodes(n, $Temp$)
    makeRoutingConsistent($Temp$)
    if *getNetworkCost*($Temp$) $<$ *getNetworkCost*($Best$) **then**
      $Best \leftarrow Temp$
    end if
    $Temp \leftarrow \emptyset$
  end for
  $Network \leftarrow Best$
  return

is two or more then a higher virtual channel plane is sought to ensure deadlock freedom as defined in Sect. 4.5.1. If the current node does not have a higher virtual channel on the required input port then one is added.

In general, adding virtual channels to a network has a lower cost than adding links. While both increase the buffer requirements and arbitration time, links also increase the crossbar radix, routing logic, and wiring between nodes. This is why the design methodology here seeks first to minimize the links between nodes and then ensures consistency by adding virtual channels. It is true that more than a few virtual channels on one input will result in poor performance due to arbitration delays. However, in practice consistency is ensured with no more than 2-4 virtual channels on any one router input.

4.6 Node Placement

The intelligent placement of network nodes is critical to the scalability of INoCs as wire delays increase. This is not a trivial problem in these types of on-chip networks for two reasons; heterogeneous cores vary in size and the links required to implement the topology may not conform to a structure easily fabricated on a 2-dimensional die. The problem of placing network nodes is addressed with far more detail and completeness in [19]. A crude placement algorithm has been developed for this work simply to ensure that the irregular topologies generated can be placed in a structure that can be fabricated

Algorithm 3 makeRoutingConsistent

```
Require: Channels ← set of all c : c ∈ Workload and c ← (N_s, N_e) : N is an endpoint node
Require: Network ← set of all linked network nodes
  for all c ∈ Channels do
    Path ← Dijkstra(Network, c.start, c.end)
    previous ← INCREASING
    current ← INCREASING
    changes ← 0
    vcplane ← 1
    for all n ∈ Path do
      if n.GetValue() > n.previous.GetValue() then
        current ← INCREASING
      else
        current ← DECREASING
      end if
      if current ≠ previous then
        changes ← changes + 1
      end if
      if changes ≥ 2 then
        vcplane ← vcplane + 1
        if n.previous.GetVirtualChannels() < vcplane then
          n.previous.AddVirtualChannel()
        end if
        changes ← 0
        previous ← INCREASING
      else
        previous ← current
      end if
    end for
  end for
  return
```

without wire delays significant enough to impact average performance. However, in this writing the details will be foregone and only results will be presented.

It should be stated that the results of this algorithm will often not result in good utilization of chip area; since the focus is on minimizing wire latencies and power consumption. Therefore, this tool should be used in early design stages when node sizes can be re-allotted and refined to improve area utilization.

4.7 Network Simulator

At present, this tool is not integrated with the XUM NoC hardware modules described in Chap. 2. Therefore, simulation is used to evaluate throughput and latency of realistic network traffic. The *gpNoCSim* simulator tool presented in [10] provides a platform for simulating uniform network traffic on a variety of topologies including mesh, torus, and tree structures. This platform was extended to provide two additional capabilities; to simulate network traffic on a custom INoC topology and to simulate application specific communication patterns.

4.7.1 Simulating an Irregular Network

The main modification needed to simulate INoC topologies was to change the routing logic so that packets could be routed in a scheme where the node address alone doesn't identify the physical location in the topology as it does with a regular topology. An INoC topology is not a known structure at the time the individual hardware modules are designed, thus routing computations can not be performed dynamically by a specific routing function. Rather, routing information is computed for each node at the time of topology synthesis and loaded into a small ROM at each node. The routing function then simply looks up the next network hop needed to reach the specified destination.

4.7.2 Simulating Application Specific Traffic

To simulate uniform network traffic, packets are generated at a specified rate and their destination is random with a uniformly distributed probability of each possible destination being selected. Since the work presented here involves generating NoCs for specific types of communication patterns, these patterns must be simulated to accurately evaluate the effectiveness of the synthesized topology. To do this the packet generation mechanism was modified such that the probability of a destination being selected is based on a distribution generated from the workload.

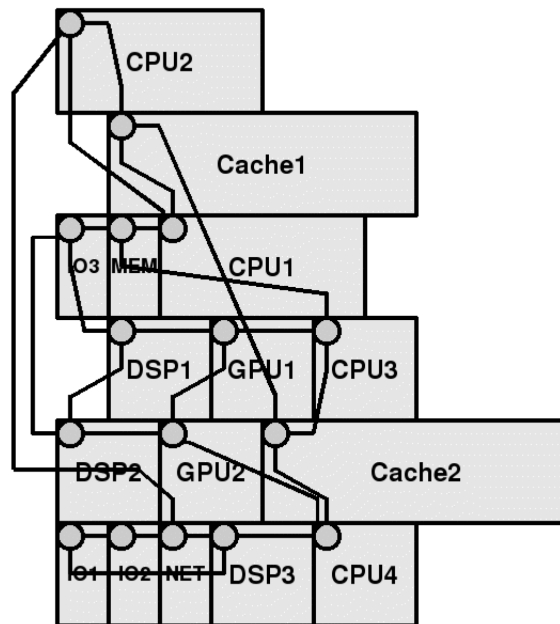

Figure 4.2. Results - Topology with Node Placement

4.8 Results

4.8.1 Test Case

The workload specified in Tab. 4.1 is used as the primary benchmark. Synthesizing the workload with a maximum router radix of three and approximate node sizes yields the topology and placement given in Fig. 4.2.

As a baseline, a mesh network has been generated with 100 different random mappings of cores to the topology. The mapping that results in the fewest network hops is shown for comparison. This number is assumed to approximately represent the improvement obtained by considering workloads in core placement as is done in [20].

Tab. 4.2 shows the average number of hops per flit and the cost of the network hardware in terms of links, virtual channels, average router radix, and the average wire distance traveled per flit. It compares the results for a fully connected network (where every node is connected to every other node), the INoC synthesized by this tool, the mesh network with the best mapping of cores, and the average mesh network.

Table 4.2. Results - Topology Synthesis

Parameter	Fully Connected	INoC	Best Mesh	Avg. Mesh
Avg. Hops / flit	1.0	1.157	2.202	2.638
Tot. Links	256	46	48	48
Tot. VCs	256	59	48	48
Avg. Radix	16	3.88	4	4
Avg. Distance	NA	55.4 microns	118 microns	154 microns

The average hops is computed with the assumption that a flit is one byte wide and that all bytes specified by the workload bandwidths are injected into the network every second. The average radix includes one link to a local node. Also note that the number of virtual channels given is the minimum number of virtual channels needed to ensure deadlock free routing.

4.8.2 Scalability Test

To test the scalability of the networks generated by this tool, several random workloads were generated and the average number of hops per flit between any two endpoints in the network was calculated on an average mesh, best mesh, and INoC topologies. This was done for 4, 8, 16, 32, and 64 cores. The workloads were generated by creating communication channels between each core and up to $sqrt(N)$ other cores (N is the total number of cores). Each core has a variable probability of being selected as an endpoint. This is to model the heterogeneous nature of communicating cores and is similar to the approach taken in [21].

The results of this test are given in Fig. 4.3. These indicate that not only does the INoC topology give lower average hops per flit for all workloads, but its scalability in terms of average hops approaches a linear scale while the mesh topologies are quadratic. Note that for this test the maximum router radix was fixed at 3. This means that the cost of the network in terms of total links is less than or equal to the mesh topologies; especially when more cores are used.

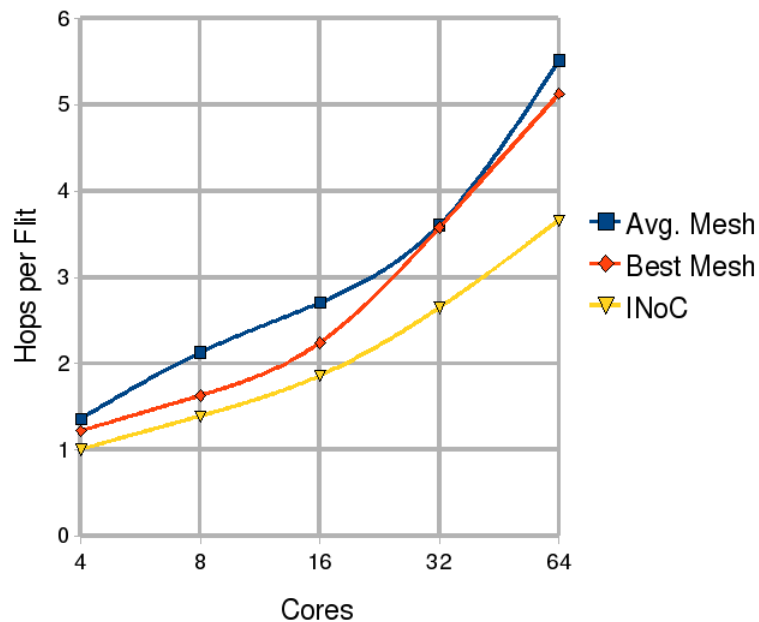

Figure 4.3. Results - Scalability

Table 4.3. Results - General Purpose Traffic

Param	INoC	Best Mesh	Avg. Mesh
Avg. Hops / flit	2.55	2.67	2.93

4.8.3 General Purpose Test

To further illustrate the advantages of using an INoC topology generated via the algorithms presented here, a test that models general purpose communication on an application specific topology was conducted. Ten different topologies of 16 cores were synthesized using randomly generated workloads and a maximum router radix of 3. The average hops per flit between every pair of nodes (rather than only those specified by the workload) is computed with equal bandwidths for each node pair. This is to model general purpose network traffic where specific patterns are unobservable. The results are given in Tab. 4.3. It is shown that for 16 core networks, the synthesized topology performs at least as well as the best core placement in the mesh topology for general purpose traffic.

Table 4.4. Results - gpNoCSim Simulator

Param	INoC	Mesh
Avg. Hops / flit	1.23	2.83
Avg. Packet Delay	412	595
Throughput	0.27	0.23

4.8.4 gpNoCSim Simulator

All of the aforementioned tests and results do not represent the actual performance metrics of an implementation of the synthesized NoC topologies, they are only theoretical. To obtain a more accurate view of how the synthesized topology compares with a generic mesh NoC, the modified gpNoCSim simulator was utilized. Using the extended tool to simulate the workload defined network traffic resulted in the numbers provided in Tab. 4.4. Note that the computation of performance metrics follows the equations given in [10]. Both topologies were simulated with the same input parameters and the same workload used previously.

4.8.5 Results Summary

The results indicate that the generated topology runs the workload with a significant advantage in terms of average hops per flit with a comparable number of links. The INoC topologies are also more scalable in terms of the average number of hops per flit between any two communicating endpoints. In addition, they perform at least as well as mesh topologies for general purpose applications. The placement algorithm is able to find node placements such that the advantage in terms of hops is not nullified by significantly longer wires. It is therefore shown that the INoC topologies synthesized by this tool offer an advantage over traditional mesh communication networks at a lower hardware cost than other INoC designs.

CHAPTER 5

FUTURE DIRECTIONS

One of the main contributions of this work is to provide a platform for multicore research. It follows that the author would like to propose the following future directions for this work.

5.1 MCAPI Based Real-time Operating System

From the XUM hardware and the bare-metal MCAPI implementation, the next level of abstraction in an embedded multicore system is a real-time operating system (RTOS). Since the MCAPI implementation runs directly on the hardware it provides a useful means for performing the thread-scheduling and synchronization tasks that are required of an RTOS. Fig. 5.1 presents a simple diagram illustrating how an asymmetric thread scheduler could be implemented. This design uses a master-slave approach with one master kernel using MCAPI to deliver thread control blocks to the various cores. The slave cores run a lightweight kernel that simply checks for incoming threads and performs simple scheduling of its own small pool of threads, while the master core handlers load balancing, assigning threads to cores, and thread creation. This minimizes the amount of control oriented code that must run on the slave cores, which may be DSPs, GPUs, or other number crunching cores that are not well suited for running control code.

5.2 Scalable Memory Architectures

XUM's NoC illustrates how cores can be interconnected without the bottleneck of a shared resource, such as a bus. However, there is still the problem of a shared memory controller. If an off-chip memory controller is simply inserted into a multicore system as a node in the on-chip network then the latency associated with accessing memory will be exaggerated, because each core will have to arbitrate for use of the memory

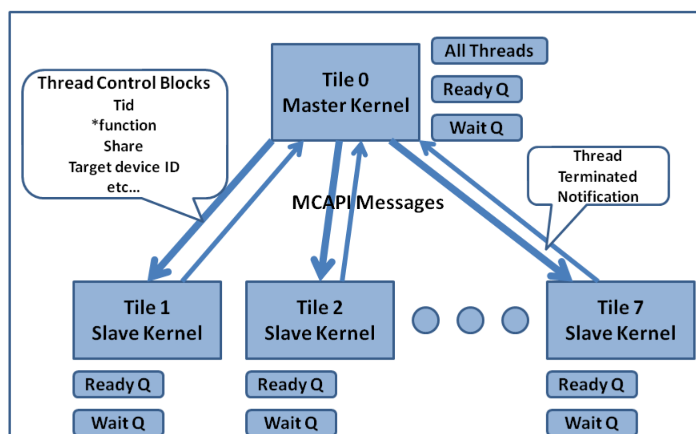

Figure 5.1. MCAPI OS Kernel

controller. Ideally, each core would have it's own interface to it's own off-chip memory. Unfortunately, this requires the number of pins on a chip to scale with the number of cores, which is not possible. One possible improvement could be to use a tree structure where the root(s) are the memory controllers that actually have the off-chip interface, leafs are tiles, and nodes are sub-memory controllers that service the requests of only a few sub-trees or leafs. This would simplify the task of servicing requests at each level and eliminate long wires.

5.3 Parallel Programming Languages

The features provided by the communication oriented ISA extension provided by XUM are only partially utilized in a language like C. Since they can only be invoked via inline assembly code, most users will only use the instructions by calling low-level API functions that use them, like MCAPI. The capabilities of the ISA extension could be most effectively used by language constructs. There currently exist many parallel programming languages (ZPL, Chapel, NESL, etc). However, they are not well known or widely used [14]. There could be great opportunity for progress by developing a language for an architecture that provides communication instructions, rather than depending on libraries such as OpenMP, MPI, and Pthreads.

5.4 Multicore Resource API (MRAPI)

The closely related cousin to MCAPI is the Multicore Resource API (MRAPI). It too could greatly benefit from being implemented on XUM's unique ISA. MRAPI is responsible for managing synchronization constructs such as locks and semaphores, shared and distributed memory regions, and hardware information available at run-time called meta-data [18]. A future XUM extension should consider implementing MRAPI at the same level the MCAPI has been in this work.

5.5 Multicore Debugging

The facilities provided by XUM could be used to investigate algorithms and techniques for increasing the internal visibility of multicore systems at run-time. Special debugging message types could be introduced to allow a third-party to observe the behavior of a parallel program and check for erroneous conditions. This third party could simply be a process running on one of the cores. In addition, since XUM is implemented on an FPGA custom hardware extensions for debugging could be added to assist run-time verification algorithms.

CHAPTER 6

CONCLUSIONS

6.1 Project Recap

Each chapter of this thesis has provided details on different areas of a project with a borad scope. Chap. 1 served as an introduction. It presented the project objectives and motivated them with an introduction to multicore system design. Parallel programming, message passing, and embedded systems are all introduced. Chap. 2 details the design of the XUM multicore processor. It describes the processor architecture, interconnection network, instruction-set extension, tool-chain, and synthesis for FPGA implementation. Chap. 3 introduces MCAPI; a lightweight message passing library for embedded multicores. It illustrates the use of XUM's ISA extension by providing an efficient transport layer implementation with a low memory footprint. Given that XUM and MCAPI apply primarily to embedded systems and are based on programmable logic, Chap. 4 presents a tool for synthesizing optimized irregular network topologies for specific communication workloads. While it is not yet integrated into the XUM design flow (and is a seperate project altogether), it is closely related and illustrates possible optimizations to the NoC system presented in Chap. 2. Several possible future directions are discussed in Chap. 5.

6.2 Fulfillment of Project Objectives

This thesis has presented a large body of original work which has sought to achieve the project objectives. To reiterate, the major objectives of this project have been

- To provide a research platform for evaluating the system level impacts of innovations related to multicore systems;

- To provide a standardization of the interface between NoC hardware and low-level communication API's with an ISA extension;
- To provide the first hardware assisted implementation of MCAPI.

Some of the biggest problems with embedded multicore systems are programmability, portability, the evaluation of new ideas, and the task of exploiting fine grained parallelism. These objectives seek to help solve these problems.

The multicore research platform that is provided is known as XUM, presented in detail in Chap. 2. Since XUM provides open HDL source code, new hardware designs can be inserted into a working system for evaluation. It provides a level of visibility that is not possible with commercial hardware and difficult to model with simulators. XUM's on-chip network and FPGA target provide a great deal of extensibility.

With constantly changing NoC hardware there needs to be some standard interface between the hardware and the communication software that uses it. The ISA extension presented in Sect. 2.3 provides this interface. NoC architectures can continue to evolve, but the basic operations of sending and receiving flits will provide a standard mechanism for implementing higher level network protocols.

In addition to the benefit of standardization, this ISA extension enables very lightweight implementation of low-level communication API's. Chap. 3 presents the first hardware assisted implementation of MCAPI. Though only a partial implementation, it has a very low memory footprint, low latency API calls, and runs without the use of an operating system. This makes it an ideal solution for implementing higher level systems, which is what MCAPI was intended for [17].

This project clearly meets it's intended objectives. However, it is clearly a vehicle for future work. It is the author's hope that many will find this work useful in advancing the state-of-the-art of the field of parallel computing.

APPENDIX

ADDITIONAL XUM DOCUMENTATION

A.1 MIPS Instruction Set Extension

This document standardizes the MIPS instruction set extension implemented in XUM. This instruction set extension is designed for the purpose of controlling XUM's network-on-chip (NoC) communication architecture and for implementing message passing primitives. Throughout this document, the term *packet* network refers to XUM's 2-D mesh NoC. The network is designed for explicit transfer of small to large sized packets of data. It is a 2-byte wide data path and supports multiple clock domains. The term *acknowledge* network refers to XUM's lightweight interconnect for low-latency signaling. It is designed for fast multi-casting of single bytes of information. For definitions of endpoints, packets classes, and other terms, please refer to the documentation on the MCAPI transport layer.

A.1.1 BCAST: Broadcast

Op: 011111 31:26	RS 25:21	unused 20:11	Funct: 00000000001 10:0

Forms a 9-bit message and broadcasts it on the *acknowledge* network. There is no reflection, so the sender does not see the broadcasted message. Messages are formatted as follows:

[1, RS[15:0]]

Unlike packets sent on the *packet* network, there is no association between subsequent messages on the *acknowledge* network. It is designed for the implementation of parallel algorithms that need to quickly synchronize and/or share small pieces of data.

A.1.2 GETID: Get core identifier

Op: 011101	unused	RD	Funct: 00000000000
31:26	25:16	15:11	10:0

Gets a hardcoded constant that serves as a processor core/tile identifier and stores it in register RD. This instruction is necessary for determining which core is executing the given code.

A.1.3 GETFL: Get operational flag

Op: 011110	unused	RT	Immediate
31:26	25:21	20:16	15:0

Gets the value of an operational flag specified by the immediate field and stores it in register RT. Values of operational flags are either 1 or 0 to indicate various error or buffer status conditions. The available flags are given in Tab. A.1.

Table A.1. NIU Flags

Flag	Address	Description (if RT == 1 then)
headF	0	A head flit is available
dataF	1	A data flit is available
netBusyF	5	The send buffer is full
recvIdleF	7	Receiver is idle (in between packets)
ackF	8	An acknowledge message is available

Op: 011111	unused	RD	Function: 00000010000
31:26	25:16	15:11	10:0

A.1.4 RECACK: Receive acknowledge

Removes an 8-bit acknowledge message from the receive queue and stores it in the low order 8-bits of register RD.

A.1.5 RECHD: Receive header

Op: 010101	unused	RD	Function: 00000000000
31:26	25:16	15:11	10:0

Removes a 16-bit header from the receive queue and stores it in register RD. This can then be used by the receiver to determine the packet class and the sender. Note that the RECHD and RECW instructions do exactly the same thing. They are given different names to follow the convention of the instruction set and to assist the programmer in forming a logical association of the received data.

A.1.6 RECW: Receive word

Op: 010101	unused	RD	Function: 00000000000
31:26	25:16	15:11	10:0

Removes 2-bytes of data from the receive queue and stores it in register RD. The received data forms the body of a packet. Note that the RECHD and RECW instructions do exactly the same thing. They are given different names to follow the convention of the instruction set and to assist the programmer in forming a logical association of the received data.

Op: 010101 31:26	RS 25:21	unused 20:16	RD 15:11	Function: 00000000001 10:0

A.1.7 RECW.C Receive word conditional

Conditionally removes 2-bytes of data from the receive queue and store it in register RD. If there is no data in the queue or if a tail has been seen then the value of RS is returned and stored in RD. This is to indicate that the receive operation failed, thus the value of RS should be some error constant. The error constant should be chosen such that the received data can never have the same value. Otherwise, a value could be correctly received but interpreted as a failure. A subsequent GETF operation may be used to determine the exact cause of a failure.

A.1.8 SNDACK: Send acknowledge

Op: 011111 31:26	RS 25:21	RT 20:16	unused 15:11	Function: 00000000000 10:0

Forms a 9-bit acknowledgment message and buffers it on the send queue of the *acknowledge* network. This is a self contained, point-to-point message where the destination endpoint is specified by RS and the source endpoint is specified by RT. This message has the following form:

[0, RS[3:0], RT[3:0]]

A.1.9 SNDHD: Send header

Op: 010100 31:26	RS 25:21	RT 20:16	RD 15:11	Function 10:0

Forms a 2 byte wide packet header and buffers it on the send queue for transmission on the *packet* network. Headers are formatted as follows:

Table A.2. SNDHD Variations

Instruction	Function	Description
sndhd.b	0	Buffer packet class
sndhd.p	1	Packet channel class
sndhd.s	2	Scalar channel short class
sndhd.i	3	Scalar channel integer class
sndhd.l	4	Scalar channel long class

[1, RS[4:0], 000, Funct[2:0], RT[4:0]]

RS specifies the destination endpoint, RT specifies the sending endpoint, and Funct specifies the packet class. The result of the operation is stored in register RD. If the send was successful then RD will be 0. Otherwise, RD will be 1. This could be due to the send buffer being full. If this happens, back off and give the network time to consume packets. Then try again.

Several variations of this instruction are available to specify the packet class. They are given in Tab. A.2.

A.1.10 SNDW: Send word

Op: 010110 31:26	RS 25:21	unused 20:16	RD 15:11	Function:00000000000 10:0

Forms a 2-byte body flit and buffers it on the send queue. A SNDW must be preceded by a SNDHD operation. If this does not happen then the body flit will be dropped by the network. Body flits are formatted as follows:

[0, RS[15:0]]

An arbitrary number of SNDW instructions may execute following a SNDHD. However, this sequence must be followed by a SNDTL operation in order to release the network resources being reserved for this message sequence. The result of the operation is stored in register RD. If the send was successful then RD will be 0. Otherwise, RD will be 1.

This could be due to the send buffer being full. If this happens, back off and give the network time to consume packets. Then try again.

A.1.11 SNDTL: Send tail

Op: 011100	unused	RD	Function:00000000000
31:26	25:16	15:11	10:0

Forms a 2-byte tail flit and buffers it on the send queue. A SNDTL must be preceded by a SNDHD operation. If this does not happen then the tail flit will be dropped by the network. Tail flits are formatted as follows:

$$[\ 1000001000000000\ ]$$

A SNDTL is used to release network resources that are reserved for a message sequence by a SNDHD. The result of the operation is stored in register RD. If the send was successful then RD will be 0. Otherwise, RD will be 1. This could be due to the send buffer being full. If this happens, back off and give the network time to consume packets. Then try again.

REFERENCES

[1] M. Ali, M. Welzl, and M. Zwicknagl. Networks on chips: Scalable interconnects for future systems on chips. ECCSC, 2008.

[2] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: Infrastructure for full system simulation. ACM, 2009.

[3] D. Bertozzi, A. Jalabert, S. Murali, and et al. Noc synthesis flow for customized domain specific multi-processor systems-on-chip. IEEE Transactions on Parallel and Distributed Systems, 2005.

[4] D. Cullen, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.

[5] W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. DAC, 2001.

[6] R. Das, S. Eachempati, A. Mishra, V. Narayanan, and C. Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps. HPCA, 2009.

[7] G. Gibeling, A. Shultz, and K. Asanovic. The ramp architecture and description language. WARFP, 2006.

[8] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S. Keckler, and D. Burger. On-chip interconnection networks of the trips chip. IEEE Micro, 2007.

[9] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, and P. Griffin. Software standards for the multicore era. IEEE Micro, 2009.

[10] H. Hossain, M. Ahmed, A. Al-Nayeem, T. Islam, and M. Akbar. Gpnocsim: A general purpose simulator for network-on-chip. ICICT, 2007.

[11] Intel Corp. *Intel's Teraflops Research Chip Overview*, 2007. http://www.intel.com/go/terascale.

[12] A. Krasnov, A. Shultz, J. Wawrzynek, G. Gibeling, and P. Droz. Ramp blue: A message-passing manycore system in fpgas. Proceedings of International Conference on Field Programmable Logic and Applications, 2007.

[13] Lawrence Livermore National Laboratory. *MPI Performance Topics*. https://computing.llnl.gov/tutorials/mpi_performance.

[14] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison Wesley, 2008.

[15] B. Meakin and G. Gopalakrishnan. Hardware design, synthesis, and verification of a multicore communication api. SRC TECHCON, 2009.

[16] MIPS Technologies, Inc. *The MIPS32 Instruction Set*, 2009.

[17] Multicore Association. *Multicore Communications API Specification Version 1*, 2008. http://www.multicore-association.org.

[18] Multicore Association. *Multicore Resource API Specification Version 1*, 2009. http://www.multicore-association.org.

[19] Murali, Srinivasan, Meloni, and et al. Designing application-specific networks on chips with floorplan information. ICCAD, 2006.

[20] S. Murali and G. DeMicheli. Sunmap: A tool for automatic topology selection and generation for nocs. DAC, 2004.

[21] C. Neeb and N. Wehn. Designing efficient irregular networks for heterogeneous systems-on-chip. EUROMICRO Conference on Digital System Design, 2006.

[22] I. Nousias and T. Arslan. Wormhole routing with virtual channels using adaptive rate control for network-on-chip. NASA/ESA Conference on AHS, 2006.

[23] G. Palermo, G. Mariani, C. Silvano, R. Locatelli, and M. Coppola. Mapping and topology customization approaches for application-specific stnoc designs. ASAP, 2007.

[24] D. Panda. Mvapich: Optimization of mpi intra-node communication for multicore systems. http://mvapich.cse.ohio-state.edu/performance/, 2008.

[25] D. Patterson and J. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann, 1997.

[26] PolyCore Software, Inc. *Poly-Messenger/MCAPI Datasheet*, 2009. http://www.polycoresoftware.com.

[27] Quadros Systems, Inc. *RTXC/mp - Multiprocessor/Multicore RTOS*, 2009. http://www.quadros.com.

[28] S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt. Mcc - a runtime verification tool for mcapi user applications. FMCAD, 2009.

[29] K. Srinivasan, K. Chatha, and G. Konjevod. An automated technique for topology and route generation of application specific on-chip interconnection networks. ICCAD, 2005.

[30] I. Sutherland. Fleet - a one-instruction computer. http://fleet.cs.berkeley.edu/docs/, 2005.

[31] M. C. Y. Lan, A. Su, Y. Hu, and S. Chen. Flow maximization for noc routing algorithms. ISVLSI, 2008.