ABSTRACT

Title of thesis: A HIGH-THROUGHPUT, LOW-POWER ASYNCHRONOUS MESH-OF-TREES INTERCONNECTION NETWORK FOR THE EXPLICIT MULTI-THREADING (XMT) PARALLEL ARCHITECTURE

Michael N. Horak
Master of Science, 2008

Thesis directed by: Professor Uzi Vishkin
Dept. of Electrical and Computer Engineering

and

Professor Steven Nowick
Dept. of Computer Science, Columbia University

As the number of processing cores per chip continues to grow, the on-chip network connecting processors to memory becomes increasingly crucial for performance. Future architectures will face scalability concerns as networks will require more die area and consume more power. The Mesh-of-Trees interconnection network is a high-throughput, low-latency network that uses pipelined routing decisions to achieve high performance for single-chip parallel processors that require high bandwidth to on-chip memory resources. The network has similar area requirements to other existing networks, but can utilize more bandwidth due to its unique topology.

Current single-chip parallel processors are developed as synchronous (clocked) circuits. A recent trend has emerged towards implementing GALS (globally asyn-

chronous, locally synchronous) architectures, which do not require a clock tree spanning the entire chip, thus avoiding the considerable challenges of design and managing power consumption.

This thesis presents an asynchronous (clockless) implementation of the Mesh-of-Trees network that features lower power and area demands, while maintaining the high throughput and low latency properties of the synchronous network. Two new asynchronous designs are proposed for the fundamental pipelined components of the Mesh-of-Trees network (routing and arbitration), which are optimized for power, area, latency and throughput. Performance and power consumption are evaluated for asynchronous components in isolation, as well as a projected full network layout.

Two issues top the agenda of CPU design in the emerging many-core era: programmers' productivity and power consumption. Through its reliance on the richest available theory of parallel algorithms, the eXplicit Multi-Threading (XMT) parallel architecture addresses programmers' productivity. The motivation for this work is to provide an effective interconnection network for the XMT architecture in terms of both performance and power consumption.

In order to provide communication between the asynchronous and synchronous timing domains, mixed-timing interfaces are implemented. The network, coupled with mixed-timing interfaces, can be used to implement a GALS architecture, where different timing domains communicate via the same asynchronous network. Performance of the XMT processor with the asynchronous network and mixed-timing interfaces is measured for several applications.

A HIGH-THROUGHPUT, LOW-POWER ASYNCHRONOUS MESH-OF-TREES INTERCONNECTION NETWORK FOR THE EXPLICIT MULTI-THREADING (XMT) PARALLEL ARCHITECTURE

by

Michael N. Horak

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2008

Advisory Committee:
Professor Uzi Vishkin, Co-Chair/Advisor
Professor Steven Nowick, Co-Chair, Columbia University
Associate Professor Bruce Jacob

Dedication

This thesis is dedicated to my wonderful family, for their unconditional love and support throughout the course of my studies. To my loving mother, sister, and grandparents, thank you for always believing in me even when I doubted myself. Finally, to my father, you are my hero and I truly would not be here without your guidance.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Uzi Vishkin, for the opportunities he provided me during my graduate studies. I am thankful for his encouragement, suggestions and support that made this work successful.

I am immensely grateful to Dr. Steven Nowick of Columbia University for introducing me to asynchronous design and his constant support, advice and wealth of knowledge.

I appreciate the helpful comments provided by Dr. Bruce Jacob as a member of the thesis committee.

I am thankful for the hard work of the entire XMT research team. I give special thanks to Aydin Balkan, for never hesitating to contribute his time and insightful advice. I am grateful to Xingzhi Wen for giving me a better understanding of XMT and graduate research as a whole. I also thank Fuat Keceli and Mary Kiemb Ma for their collaboration on the two ASIC projects.

Finally, I want to express my profound gratitude to my parents, grandparents, and sister for their understanding and continuous support throughout my studies.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

IOS	Independence-of-Order Semantics
LP	Latency-Oriented Primitive
LSRTM	Length of the Sequence of Round Trip to Memory
MoT	Mesh-of-Trees
MOUSETRAP	Minimal-Overhead Ultra-high-SpEed TRansition-signalling Asynchronous Pipeline
MWA	Merge-Without-Arbitration
PRAM	Parallel Random Access Machine
TPP	Throughput-Oriented Primitive
UMA	Uniform Memory Access
XMT	eXplicit Multi-Threading

Chapter 1

Introduction

The transition from the multi-core era, of dual and quad core processors, to the many-core era, with tens, hundreds or thousands of cores, presents new development challenges for programmers. Ease-of-programming needs to be addressed in order for programmers to harness the processing power provided through parallelism.

The "PRAM-on-Chip" project at the University of Maryland provides a solution to the ease-of-programming challenge for large-scale parallel processors. The Parallel Random Access Machine/Model (PRAM) is an easy model for parallel algorithmic thinking, accompanied by a rich body of algorithmic theory, second only to its serial counterpart. The eXplicit Multi-Threading (XMT) processor bridges the gap between PRAM algorithmic thinking and programming, providing the programmer with a PRAM-like performance model [34] for algorithmic development. Rather than focusing on synchronization between threads or architecture-specific optimizations, programmers are free to express the natural parallelism of an application.

The XMT performance model is comprised of three quantities [34]: (i) Computation Depth, given by the number of operations that have to be performed sequentially, either by a thread or while in serial mode. (ii) Length of Sequence of Round-Trips to Memory (or LSRTM) which represents the number of cycles on the

critical path spent by execution units waiting for data from memory. (iii) Queuing delay (or QD) which is caused by concurrent requests to the same memory location; the response time is proportional to the size of the queue.

Items (ii) and (iii) above can be optimized by employing a low-latency, high-bandwidth interconnection network between processing clusters and memory modules on chip. Such an interconnection network is critical for maintaining good performance in the XMT execution model. Earlier attempts to support PRAM were implemented as multi-chip multiprocessors (e.g. TERA-MTA, SBPRAM, NYU Ultracomputer and the IBMRP3), which were constrained by memory access performance and had limited success. The XMT processor, implemented as a single-chip parallel processor, can achieve much higher bandwidth using an on-chip interconnection network, thus pushing the performance bottleneck towards the computation depth of the algorithm and away from the latency of memory access.

The multi-threaded programming model for XMT algorithms uses "independence-of-order" semantics (IOS): every thread can proceed at its own pace, independent of other concurrent threads. IOS reduces synchronization requirements between threads, allowing processing to advance as data becomes available. This translates to better system performance. Another consequence of IOS is that processors do not require tight synchronization with memory modules [35].

The Mesh-of-Trees (MoT) interconnection network [4, 7, 6, 5] was developed for the XMT processor and provides high throughput and low latency for multi-threaded, shared-memory applications. The network, implemented using synchronous (clocked) logic, scales well and has good performance under the heavy

traffic conditions present in XMT. However, as the network grows, power consumption becomes a significant design constraint. Furthermore, chip area that could be used for larger caches or more processing cores must be devoted to the network.

To address the concern of power and area overheads from the interconnection network, an asynchronous (clockless) design is investigated. Designers have turned to asynchronous logic for implementing low-power, robust designs that are resilient to on-chip variations in the manufacturing process. In addition, asynchronous designs do not require the difficult task of distributing a high-fanout clock network, which accounts for a large portion of dynamic power consumption.

This thesis presents two new asynchronous designs for the fundamental pipelined components of the Mesh-of-Trees network (routing and arbitration), which are optimized for power, area, latency and throughput. The asynchronous designs are implemented using a standard cell methodology, with the exception of an arbiter component not found in standard cell libraries. The basic architecture and operation are described in detail in Chapter 3, and analytical performance equations are derived for latency and cycle time.

In order to provide interoperability with different timing domains on a single chip, mixed-timing FIFOs [12] for synchronous to asynchronous communication are incorporated at network ports. The mixed-timing FIFOs are implemented using a standard cell methodology, with the exception of data validity controllers originally implemented as dynamic logic. To provide communication between the interfaces of the network, which use a two-phase transition signaling protocol, and the asynchronous interfaces of the mixed-timing FIFO, which use a four-phase return-to-zero

signaling protocol, new protocol converters are developed and described in Chapter 4.

Asynchronous primitives are evaluated in isolation and compared with synchronous versions from [4, 5]. Area savings of 64% and 84% are reported for routing and arbitration network primitives. Power consumption was reduced by 7x and 10x respectively for the same asynchronous network primitives. Mixed-timing FIFO implementations with protocol converters are evaluted for latency and throughput. A projected 8-terminal network layout using asynchronous primitives is developed and evaluated. Limitations in current CAD tools add considerable challenges to creating a full-network layout, so the projection is created by adding appropriate wire delays based on a proposed floorplan, explained in detail in Chapter 5. Finally, the performance of XMT is measured for several applications with the synchronous and mixed-timing networks.

Chapter 2

Background

This chapter presents the background and motivation for the current research. First, the challenge of designing a network for a massively parallel on-chip architecture is discussed in the context of the PRAM-on-chip vision. Then, the topology, fundamental components, and advantages of the Mesh-of-Trees network [4, 6, 5] is presented. Finally, we discuss how asynchronous pipelines can be used to reduce power consumption while maintaining high performance in the Mesh-of-Trees network.

2.1 PRAM-On-Chip

The Parallel Random Access Machine (PRAM) is an abstract architecture consisting of multiple processors, each connected with uniform memory access (UMA) to a global shared memory. Accompanying this model is a rich algorithmic theory of PRAM algorithms that exploit the natural parallelism in applications. Programmers are freed from difficult decomposition or architecture-specific optimizations, making the PRAM model an easy-to-program parallel programming model.

PRAM research was very active during the 1980s and early 1990s, when theorists developed the second largest body of alrogithmic knowledge, second only to the serial RAM (Random Access Machine). At the time, large parallel machines were

implemented as multi-chip multiprocessors, and limited bandwidth and long latencies for communication hurt performance, and halted the advancement of PRAM research.

As we prepare to enter the era of one billion transistors per chip, new interest has sparked in single-chip parallel processors. The eXplicit Multi-Threading (XMT) project seeks to take advantage of the surplus of transistors to implement a PRAM on a single chip. With a high-bandwidth, low-latency on-chip interconnection network, processors and memory can overcome the limitations of earlier implementations, freeing researchers to pursue the PRAM-on-chip vision.

2.2 On-Chip Interconnection Networks

This section presents some background on on-chip interconnection networks. These networks are typically used to connect multiple processing elements to other processors, memory modules, and off-chip peripherals. First, we briefly examine some commonly-used interconnection networks. Then, the Mesh-of-Trees (MoT) interconnection network, developed for the eXplicit Multi-Threading (XMT) processor is presented in more detail.

2.2.1 Overview of Existing Networks

Researchers and manufacturers have developed many ways to connect processors and memory to implement parallel machines. This section presents several traditional interconnection networks of varying complexity.

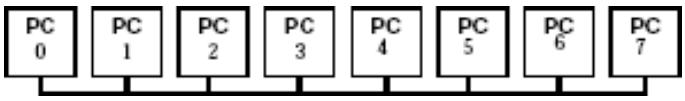


Figure 2.1: Bus with $N = 8$ Terminals (Fig. 2.2,[4])

- *Bus*: The bus is one of the simplest interconnection networks. In a bus architecture, a set of N elements share a common channel for communication. Figure 2.1 shows a bus with $N = 8$ terminals.

  In order to communicate using the bus, one sender, called the *master*, asserts new data and destination information onto the bus [15],[1]. Only one terminal is designated the master at any time. All terminals monitor the bus at every cycle, but only the specified recipient accepts data from the bus. An arbitration protocol decides which terminal is the master for the next cycle.

- *Ring*: A ring is a 1-dimensional, circular array of terminals. Each terminal is connected to two neighbors, with the first and last terminals as adjacent nodes, shown in Figure 2.2. Each processor has an interface to the ring, where new packets are injected into the network and traffic is monitored for new, incoming packets. Packets travel in unidirectional, linear paths from source to destination. Multiple ring networks were used recently in the Cell Processor [23] for high-speed communication.

- *2-D Mesh*: The 2-dimensional (2-D) mesh is the most common implementation of the mesh network. 2-D mesh networks connect processing elements arranged as a $m \times n$ grid, where m and n are rows and columns of the grid [15]. Figure 2.3 shows a 4×4 2-D mesh network.

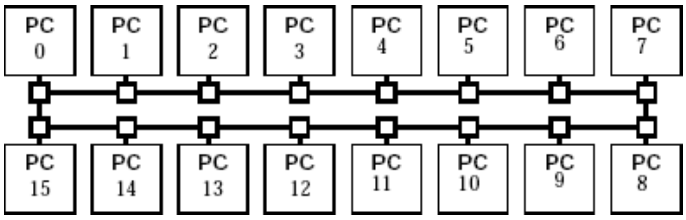


Figure 2.2: Ring Interconnection Network (Fig. 2.5a,[4])

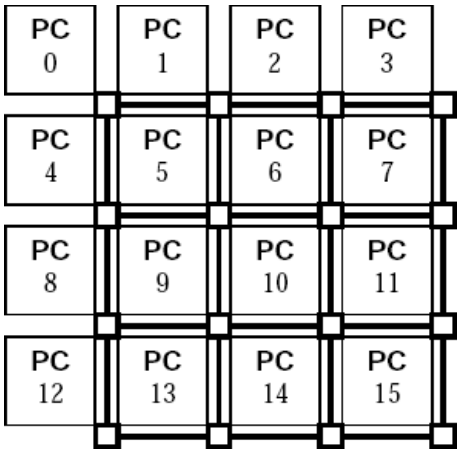


Figure 2.3: 4×4 2-D Mesh Interconnection Network (Fig. 2.5b,[4])

Each processor has a unique interface to the mesh for injecting and removing packets from the network. The nodes of the 2-D mesh route packets from source to destination, and execute an arbitration protocol when congestion occurs. The interconnection network of the TILE64 processor (Tilera) uses five 2-D mesh networks for on-chip communication [37].

- *Hypercube*: Hypercube is a member of the *cube* or *torus* family of networks [15]. A hypercube of dimension r has $n = 2r$ nodes, each of which has $r = \log n$ edges. Each node is numbered with a binary string of length r. Two nodes are connected by an edge if and only if they differ by exactly one bit. Connections between nodes are bidirectional and routing is non-unique between source and

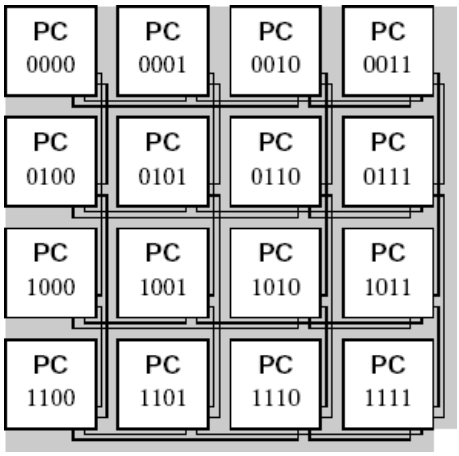


Figure 2.4: Physical Implementation of a 4-dimensional Hypercube (Fig. 2.6,[4])

destination. Figure 2.4 shows a physical layout for a 4-dimensional ($r = 4$) hypercube network.

- *Butterfly*: The butterfly network is one of the most commonly implemented interconnection networks [15]. Figure 2.5 shows a 2-ary 3-fly butterfly network that connects $2^3 = 8$ nodes. The butterfly network has logarithmic depth, which is appealing to designers. Unlike mesh and hypercube networks, butterfly has unique routing paths from source to destination. The routing path from node 6 to node 0 is shown in Figure 2.5.

2.2.2 Mesh-of-Trees Interconnection Network

An envisioned PRAM-on-chip requires a high-bandwidth, low-latency interconnection network capable of sustaining high throughput for transactions between parallel processing cores and global shared memory. The Mesh-of-Trees network (MoT) [4, 6, 5] was designed as part of the eXplicit Multi-Threading (XMT) project

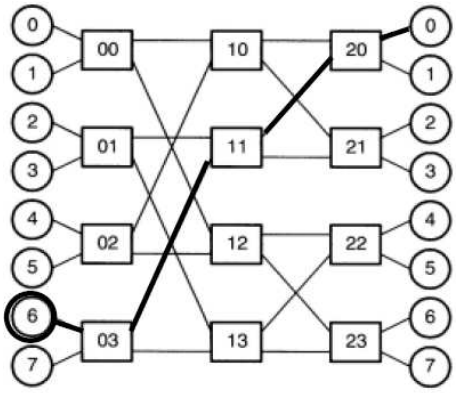


Figure 2.5: Butterfly Interconnection Network ($N = 8$) [15]

and provides such a network for massively parallel on-chip architectures.

A single Mesh-of-Trees network has N senders and N receivers. In the XMT context, senders are processing clusters and receivers are distributed memory modules [36]. Networks have unidirectional data flow, therefore two networks are embedded in XMT: one for requests from clusters and the other for responses from memory modules.

2.2.2.1 Topology

The network consists of set of N fan-out trees, rooted at the senders, and a set of N fan-in trees, rooted at the receivers. Each tree is a binary tree with depth $logN$, having N leaves, shown in Figure 2.6(b). Each of the leaves corresponds to a unique destination, one of the N receivers. A fan-in tree is also has depth $logN$, and N leaves corresponding to each of the N senders, shown in Figure 2.6(c). At this lowest level, the N^2 leaves are connected to form a unique path from each sender to receiver.

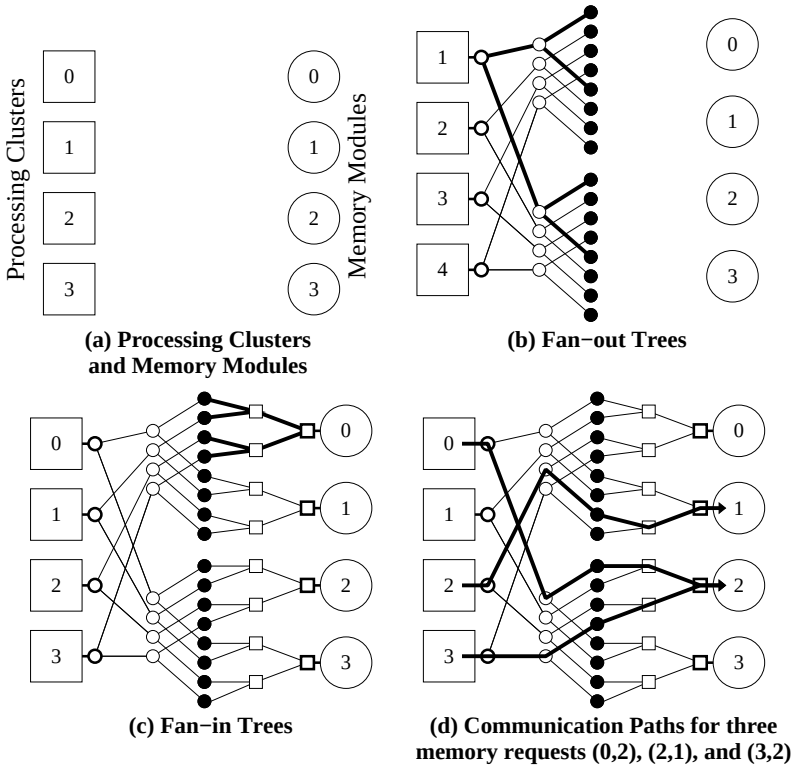


Figure 2.6: Mesh-of-Trees with 4 Clusters and 4 Memory Modules [4]

Routing decisions are made in a pipelined and decentralized fashion, making one part of the decision at each level of the fan-out tree. There are no routing decisions in the fan-in trees. In this way, directing traffic amounts to a binary decision per stage, up or down. For the example routing paths in Figure 2.6(d), an $N = 4$ network requires two decisions (one at the root and one at the first level) before reaching the leaves and continuing through the fan-in tree to the specified destination.

This approach has several key advantages. First, Mesh-of-Trees has low interference. Unless traffic in the network is extremely unbalanced, packets with different sources and destinations will not interfere. This avoids performance losses due to contention for shared resources in the network.

The Mesh-of-Trees network has logarithmic depth, requiring each packet to traverse $2 \cdot logN$ total stages from sender to receiver. This has the potential to provide low-latency communication even when scaling to very large networks. Most importantly, the Mesh-of-Trees network has decentralized routing decisions. Nodes of the tree communicate only with direct neighbors and require no global synchronization. Additionally, the decision made at each level amounts to a simple routing or arbitration decision, allowing for low logic depth and faster operation than alternative networks [4, 5].

2.2.2.2 Network Primitives

This section describes the basic funcationality of the three fundamental components, or primitives, of the MoT network: the routing, arbitration, and pipeline primitives.

Routing Primitive: The routing primitive, "Primitive A" in Figure 2.7, is used in the fan-out trees. Each routing primitive accepts data on a single input port, and directs that data to exactly one of the two output ports. The decision is made based on one bit of the destination address encoded in the packet. Each port executes a synchronous handshaking protocol with request, acknowledgment, and data signals. Stage N asserts "*ack*" to stage $N-1$ when it can accept new data. When stage $N-1$ asserts *Request*, data will be exchanged at the next clock edge, completing one transaction.

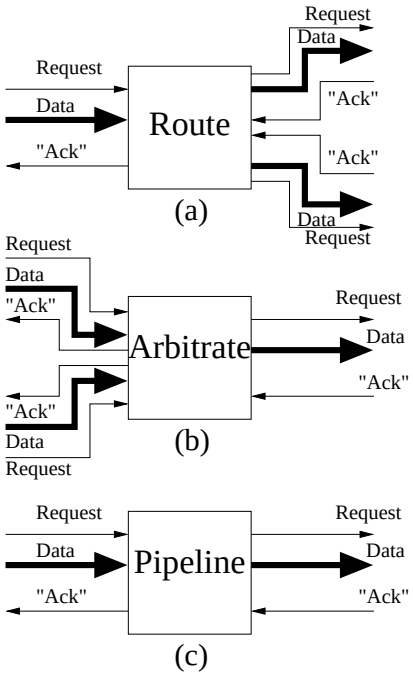


Figure 2.7: Synchronous Mesh-of-Trees Primitives [4]

Arbitration Primitive: The arbitration primitive, "Primitive R" in Figure 2.7, is used in the fan-in trees. Each arbitration primitive has two input ports and a single output port. During each clock cycle, one of the two competing inputs can be forwarded to the next stage. If there is no contention, then the arbitration primitive has a latency of one cycle. The arbitration primitives execute a strong fairness policy by updating the dynamic priority between input ports at every cycle [4].

In the context of XMT, the network traffic consists mainly of *load* and *store* operations between processing clusters and memory modules. Packets for *load* operations contain the desired address to be fetched, while *store* packets must contain both destination address and data. The data width in the network could be made large enough to store both address and data, but bandwidth would be wasted for

load operations. Instead, store packets are sent as consecutive flits, one carrying data and the other carrying address information.

Multi-flit packets can be handled by the senders and receivers, encoding flits of a multi-flit packet with identifiers at send-time and reconstructing when all flits are received. However, this places extra burden on the receiving environment to buffer incoming flits until all have arrived. Instead, the network is designed with the specification that multi-flit packets are delivered in consecutive cycles. This is achieved through "winner-take-all" arbitration, where flits 1 to $N-1$ of a multi-flit packet will bias the arbitration decision for the next cycle, ensuring that the next flit to advance through the arbitration stage will be from the same input port. Since all flits are injected into the network in consecutive cycles, this policy guarantees that multi-flit packets will be delivered in consecutive cycles.

Pipeline Primitive: The pipeline primitive, "Primitive B" in Figure 2.7, is the simplest of the three, with one input port and one output port. It executes the same synchronous handshaking protocol as the other primitives. When it can accept new data, the primitive asserts the "*Ack*" signal to the previous stage. After the previous stage has new data and asserts the *Request* signal, data will be registered in the pipeline primitive at the next clock edge. Pipeline primitives can be useful for adding buffering on long wires in the network.

2.3 Advantages of Asynchronous Design

As single chip parallel processors scale to increasing numbers of cores, the overheads of interconnection networks become first-class design constraints. Area overheads of networks reduce the available area that could be used for additional processors or caches. The added power overheads can be as much as a single core in a multi-core system [25], making a significant impact on the power budget. The Mesh-of-Trees network is not immune to these overheads.

One solution that answers the power concern is asynchronous (clockless) logic design. An asynchronous interconnection network would save on power by eliminating the need for global clock distribution. Since clock power consumption is a major portion total chip power [13, 20], this can represent significant savings. Instead, different localized timing domains can exist on a single chip, glued together by an asynchronous interconnect fabric. This is typically referred to as a globally asynchronous, locally synchronous (GALS) architecture [11]. While synchronous designers employ clock-gating as a method of reducing dynamic power for inactive components on a chip, the asynchronous timing domains naturally provide this funcationality, only transitioning nets when there is active computation, dubbed "perfect clock gating".

Asynchronous designs, since they are self-timed, are also more tolerant of on-chip variations. Communication is typically localized between neighboring modules, which are similarly affected by manufacturing process and temperature. This locality property reduces verification efforts for designers. During normal operation,

asynchronous circuits are more resilient to changes in temperature and voltage conditions and, unlike synchronous implementations, do not have to operate based on worst-case assumptions.

2.4 Asynchronous Signaling Protocols

As opposed to synchronous circuits, where events occur at specific times, asynchronous circuits are self-timed. Events occur as the result of local channel communication between adjacent modules. The basic interface between modules consists of a sender, generating requests (Req), and a receiver, responding with acknowledgments (Ack). In the case of the network, data is also being passed between adjacent modules.

There are several signaling conventions for control signals (*Req* and *Ack*) and for data. The selection between them depends on the application as well as physical constraints. This section discusses two signaling protocols for control signals, transition signaling (or 2-phase) and 4-phase, as well as two data signaling conventions, delay-insensitive communication and bundled data.

Transition Signaling: Transition signaling is the simplest form of communication between asynchronous modules. A transaction begins with both signal wires at the same level. The sender generates a request by causing a transition on the *Req* wire. The receiver eventually responds by causing a transition on the *Ack* wire. At this point, the transaction is complete. This protocol is also known as Two-Phase Signaling, since either the sender is waiting for acknowledgment, or the

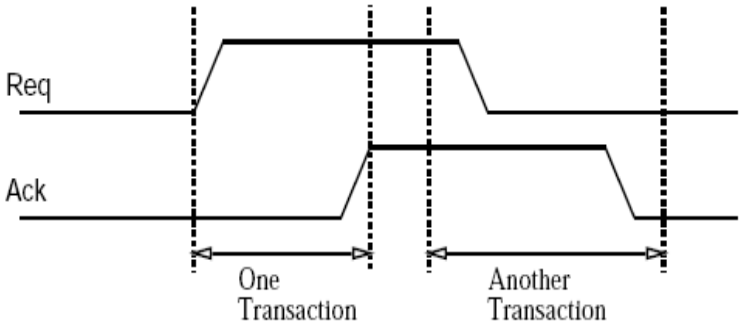


Figure 2.8: Transition Signaling Protocol [9]

receiver is waiting for the next request.

Figure 2.8 shows two consecutive transactions using transition signaling. Note that the level of the signals is not important, only whether they are both at the same level.

Four-phase (Return-to-Zero) Signaling: Four-phase (or return-to-zero) signaling adds two additional phases to each transaction, indicated by the level of the request and acknowledgment wires. The extra transitions ensure that the request and acknowledgment wires return to their quiescent state, where both are low, before the next transaction can begin. Hence the name "return-to-zero".

The four phases are referred to as *quiescent*, *requesting*, *acknowledging*, and *clearing* in [9]. Initially, both *req* and *ack* are low, indicating there is no pending transaction. Next, a rising edge on *req* will initiate a transaction between sender and receiver. The receiver acknowledges the request by asserting the *ack* wire. Then, by deasserting the *req* wire, the sender begins the clearing phase of the protocol. The receiver acknowledges the clearing signal by also lowering its *ack* wire, returning to the quiescent state.

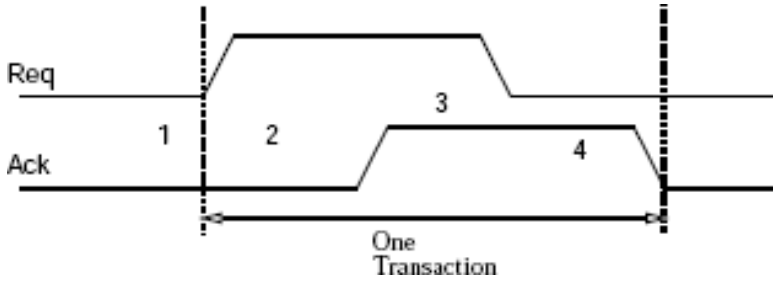


Figure 2.9: Four-phase Signaling Protocol [9]

Transition signaling typically has higher throughput for global communication than four-phase, because it requires half of the transitions. Therefore, only one roundtrip channel communication, rather than two in four-phase signaling. Additionally, more power is used for four-phase signaling since there are twice the number of transitions per transaction. On the other hand, logic blocks are typically much simpler and faster using four-phase signaling. Largely, the decision between the two is influenced by the functionality of the asynchronous module, which may or may not be conducive to one or the other.

Dual-Rail, Delay-Insensitive Data Communication: Delay-insensitive data communication between asynchronous modules uses multiple wires to communicate each bit of data. Data is only passed after an event occurs for each bit of data, allowing for correct operation in the face of arbitrary delays on data wires. This section describes four-phase, *dual-rail* data communication, a standard protocol for passing data asynchronously which combines four-phase signaling and delay-insensitive protocols. Two-phase dual-rail signaling can also be used, as well as other encodings such as 1-of-4 (see [9]).

In dual-rail, each bit of data is communicated using two wires, R_0 and R_1.

Acknowledgment from the receiver is combined into a single wire, A, for a total of $2N + 1$ wires used to communicate N bits of data. Since this is a four-phase protocol, initially all wires are deasserted. The protocol advances to the *requesting* phase when N transitions have occured, one for each request (R_0/R_1) pair. The receiver responds by asserting the single acknowledgment wire, A. Then, once all request wires have been deasserted, the receiver can deassert the acknowledgment wire, completing the passing of data.

Bundled Data: Bundled data is a standard protocol for asynchronous data communication. Data is communicated as one wire per bit accompanied by a single request signal, forming a signal bundle. First, the sender sets the values of the data wires, then initiates a request on the *Req* wire, either by transition signaling or some other convention. There is a timing constraint associated with this protocol, known as the *bundling constraint*. All data wires must be stable and at their correct values before the bundled *Req* arrives. Effectively, the bundled request has a setup constraint, and serves as a local completion signal that is transmitted along with the single-rail data bundle.

This scheme is advantageous because: 1) it simplifies the datapath wiring, using one wire per bit, and 2) it simplifies datapath control, having a single wire assert validity for an entire bundle of data wires. However, unlike delay-insensitive data communication, a local one-sided timing constraint must be enforced.

Signaling Protocols in the Asynchronous Network: For this research, we select transition signaling and bundled data protocols for the interfaces of network primitives. Transition signaling is chosen because it is fast, requiring only two

wire transitions per transaction (see Section 2.4). Bundled data is selected to reduce logic and wiring complexity on the datapath, thereby lowering area and power consumption of the network.

2.5 Asynchronous Transition-Signaling Pipelines

This section presents background on asynchronous transition-signaling pipelines, the type used to implement the asynchronous Mesh-of-Trees network. Pipelines are a series of storage elements, with processing logic in between, through which data can flow.

When no logic processing is required between stages of the pipeline, a FIFO (first-in, first-out) is implemented. FIFOs can be synchronous, operating according to a global clock, or asynchronous, with individual stages responding to local events. The input and output data rate may vary, allowing for flexible exchange of data from source to destination.

For this research, we focus on asynchronous pipelines. Asynchronous circuits are event-driven, data advances based on local decisions between adjacent stages, removing the need for global synchronization.

Transition signaling, or two-phase signaling (Section 2.4), is used as the communication protocol between pipeline stages. Each transaction consists of two transitions on wires, one for request and one for acknowledgment. The types of pipelines examined for this research use *bundled data*, so that valid data appears on data wires *before* the request and the acknowledgment is sent *after* data is safely stored (or *con-*

currently in the case of MOUSETRAP (see Section 3.4).

2.5.1 Background

In this section, we look at two transition-signaling asynchronous pipelines, one that uses phase conversion and one that does not. Each of the pipelines has performance and area overheads, and for those reasons they are not used in the design of the pipelined network primitives in the asynchronous network. However, it is important to understand alternative approaches to appreciate the significance of the results presented in this thesis.

The classic asynchronous pipeline is the *micropipeline* [32],[16]. Micropipelines use transition-signaling and bundled data protocols for pipeline stages. The basic control of micropipeline FIFO stages is shown in Figure 2.10. Each stage consists of a storage element, a capture-pass latch, and a Muller C-element.

The Muller C-element acts as an *AND* operation for events. In this case, data is captured in the current stage when a new request arrives and the current stage can accept new data. Later designs of micropipelines used standard latches instead of more complicated capture-pass latches [16].

There are significant drawbacks to using micropipelines for the asynchronous interconnection network. The first is the use of non-standard components. Micropipelines depend on Muller C-elements for control, which are not found in standard cell libraries. Additionally, *toggle elements* are used either as part of the capture-pass latch design [32] or elsewhere in the control logic [16].

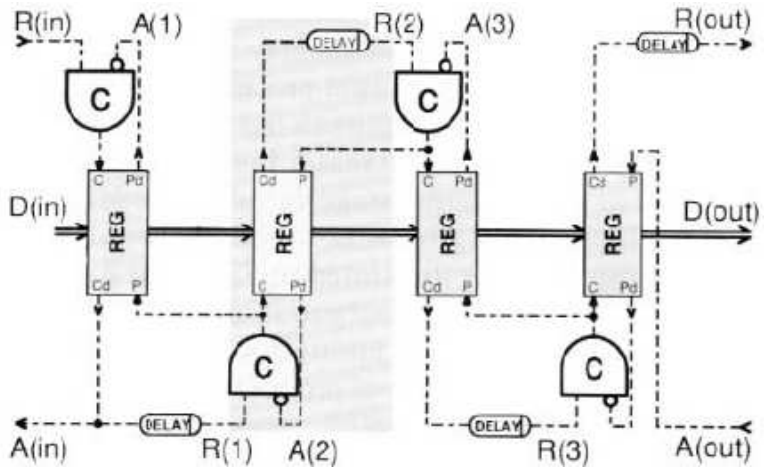


Figure 2.10: Micropipeline without Processing [32]

Another drawback is the throughput of micropipelines. Throughput is a key for performance of the interconnection network, as well as the performance of the external modules utilizing the network. Micropipelines provide very robust communication of data, but use a very conservative approach for generating request and acknowledgment signals. Another approach that trades some robustness for performance improvement may provide a more optimal solution.

The second approach to phase conversion is to use a storage element compatible with transition signaling control. There are several asynchronous FIFO implementations that use dual-edge-triggered D-flip-flops (DETDFF's) as storage elements [9],[38]. Figure 2.11 shows a branch and join FIFO designed using this methodology. The control logic consists of Muller C-elements and *merge elements*. Merge elements act as an *OR* operation for events, with output transitioning when either input transitions.

The function of each stage is to register new data following a request transition

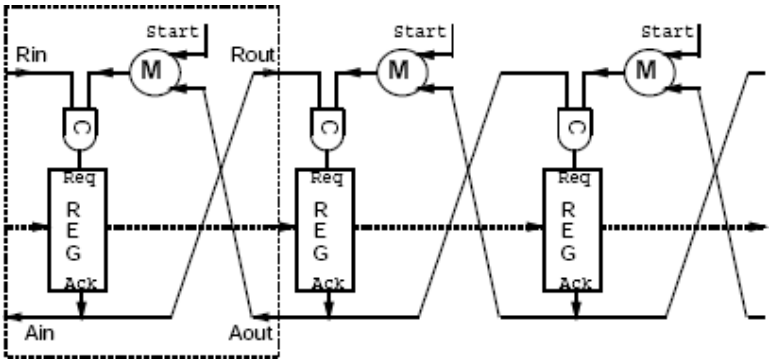


Figure 2.11: Branch and Join FIFO Control [9]

(*Rin*), given that the *Start* signal occurred at initialization. Then, new data is registered only after an acknowledgment is received from the next stage, transition on *Aout*, and new data is available from the previous stage, transition on *Rin*.

The downside to this approach is the dual-edge-triggered flip-flops used for storage. DETDFF's are larger than standard latches or capture-pass latches of micropipelines, and are also slower. While they avoid phase conversion on the control path, pipelines based on this style have power and performance overheads that make them less than ideal for a low-power, high-throughput interconnection network.

2.5.2 MOUSETRAP: High-Speed Asynchronous Pipelines

MOUSETRAP (Minimal Overhead Ultra-high-SpEed TRansition-signaling Asynchronous Pipeline) [31] is a high-performance, asynchronous pipeline that is excellent for fine-grained datapaths. Proposed by Singh and Nowick in [31], the pipeline modules provides for low-overhead movement of data, as the datapath uses standard transparent latches (rather than edge-triggered flip-flops) and the simplified

latch control consisting of a single combinational gate. The basic architecture and operation of a MOUSETRAP pipeline stage is described later in section 3.4.

The MOUSETRAP pipeline has several key advantages. Adjacent stages in the pipeline communicate via transition signaling, explained in section 2.4, a fast handshaking protocol consisting of a single request and acknowledge pair. Bundled data is used for the datapath (rather than dual-rail), reducing the logic overhead compared to delay-insensitive signaling, thus obtaining improved area and dynamic power consumption.

Additionally, each stage has very low latency, as the latches are kept normally transparent, and data advances quickly through the pipeline. Pipeline stages communicate only with neighboring stages, eliminating the need for complicated synchronization, and in comparison to other asynchronous pipelines, MOUSETRAP generates an earlier completion signal [31], improving throughput by reducing the time between subsequent requests.

The low area overheads and high-performance of the MOUSETRAP pipelines make them an ideal candidate for implementing the asynchronous Mesh-of-Trees network. Next, in Chapter 3, we discuss the fundamental asynchronous circuits of the network, that feature transition signaling and bundled data, compatible with high-speed MOUSETRAP pipeline stages.

2.6 GALS Architectures

Globally-Asynchronous Locally-Synchronous (GALS) architectures [11],[26] combine the advantages of synchronous and asynchronous design methodologies. In a GALS system, multiple synchronous "islands" communicate via an asynchronous interconnection network. Synchronous islands operate independently, with different and unrelated clock inputs. Synchronous modules are designed according to the proven design methodologies and tools available to synchronous designers. The goal of this thesis is to demonstrate a low-power, high-throughput asynchronous interconnection network capable of integration into a GALS architecture. The vision is to implement the various processing clusters and memory modules of XMT [36] as synchronous islands, creating a globally asynchronous system that is lower power and more flexible while maintaining high performance.

Designers are turning to GALS architectures for several reasons. First, globally asynchronous systems do not require global clock distribution. Implementing a low-skew clock tree for large-scale, single-clock systems has proven to be difficult and requires substantial verification and test resources. Without global synchronization, the operation of independent synchronous islands becomes *orthogonal* to the on-chip communication [24].

The GALS approach creates modularity and reusability for synchronous components that can be designed once, and reused in multiple designs. The advancements in process technology, to smaller and smaller feature sizes, has created a surplus of transistors, allowing for larger designs to fit on a single die. A growing

trend is implementing system-on-a-chip (SoC) designs that incorporate full systems, previous requiring several chips, into a single distributed system on a single chip. The GALS methodology allows developers to design and test synchronous modules individually and then later incorporate them into larger SoC's as intellectual property (IP).

Another advantage of globally asynchronous systems is that they incorporate the advantages of asynchronous design. Asynchronous logic is event-driven, spending dynamic power only as the result of requests from adjacent modules. Synchronous logic, without proper clock-gating, uses substantial dynamic power distributing clock transitions even when the module is at a quiescent state. In this sense, asynchronous logic provides "perfect clock gating" at all levels of implementation. Various studies of power consumption in GALS architectures [27],[22] find advantages to the approach. In addition, the asynchronous communication is more resilient to on-chip variations (PVT) and has reduced electromagnetic interference.

The current research focuses on a high-performance asynchronous interconnection network for a large-scale, single-chip parallel processor. Typically, asynchronous designs are implemented as extremely low-power systems that do not have high performance demands. Successful designs, such as Philips' asynchronous 80C51 microcontroller [18] appears in 100 million products from cell phones and pagers to electronic passports and smartcards.

However, asynchronous designs can have high-performance, even out-performing synchronous designs [33],[30],[8],[31]. Synchronous designs have the limitation of performance at the worst-case timing of all clocked paths. Asynchronous circuits,

on the other hand, are self-timed and operate according to the average-case performance. GALS architectures combine the flexibility and low-power properties of asynchronous with the proven ease-of-design and performance of synchronous, providing an exciting new direction for large-scale system design.

One of the challenges of implementing GALS architectures is the mixed-timing interfaces required for communication between synchronous and asynchronous modules. Many approaches have been researched, each having their own advantages and disadvantages. One method is by using locally-generated *pausible clocks* for interfacing synchronous and asynchronous modules [39]. The scheme proposed in [39] uses asynchronous finite-state machines to generate a local clock for inserting new data from a synchronous sender into an asynchronous FIFO. The locally generated clock may be paused or stretched in order to avoid setup or hold violations with the clocking signal in the synchronous domain. However, these designs require modifications to receiver clock domains and may have significant performance penalties when restarting receiver clocks. Additionally, modifications to synchronous environemnts, required by pausible clock designs, challenge the reusability property of GALS architectures that we would like to maintain.

Another approach to designing mixed-timing interfaces, proposed by Chelcea and Nowick [12], focuses on designing reusable components for high-throughput communication between different timing domains. Timing domains may be synchronous, with different and unrelated clocks, or asynchronous. The concatenation of *put* and *get* components forms four mixed-timing FIFOs. With proper capacity at steady-state, the FIFOs can provide throughput of one data item per cycle, with

put and *get* components operating independently with *no synchronization overhead*. In addition, the FIFOs do not require any modifications to synchronous domains. For these reasons, the mixed-timing FIFOs from [12] were chosen interfacing the asynchronous network and eXplicit Multi-Threading (XMT) processor.

Chapter 3

Asynchronous Primitives

3.1 Overview

The building blocks, or primitives, of the asynchronous Mesh-of-Trees network are now described in detail. There are three types of primitives used in the network: a *routing primitive* (used in the fan-out tree rooted at the sender), an *arbitration primitive* (used in the fan-in tree rooted at the receiver), and a *pipeline primitive* (used for added storage and buffering on long wires). In this chapter, we detail the function of each primitive and define analytical performance equations for latency and throughput.

3.2 Routing Primitive

The routing primitive accepts data from a single input port and forwards the data to exactly one of two output ports. The fan-out tree, consisting of routing primitives, is responsible for directing each packet to its appropriate fan-in tree. The asynchronous routing primitive was designed by Dr. Steven Nowick of Columbia University. As part of this new research, it is implemented and analytical equations for performance and timing constraints are derived. This section will discuss the architecture of the routing primitive, its basic operation, and performance equations.

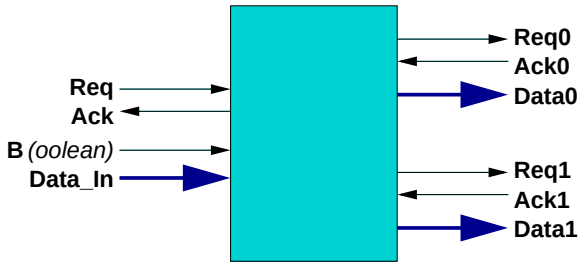


Figure 3.1: Block Diagram of an Routing Primitive

Block Diagram: The fanout primitive has one input port and two output ports, as shown in Figure 3.1. Each operates the transition-signalling protocol with bundled data. There is an additional input signal, B, which is used to determine the routing decision. When a new request arrives at the input, the request and data are conditionally forwarded to one of the outputs.

Architecture of the Routing Primitive: The routing primitive consists of two sets of data latches, latch controller logic, and toggle elements, as well as acknowledgment generation logic, shown in Figure 3.2. Standard D-type transparent latches are used instead of clock flip-flops. The latches are normally opaque (disabled), preventing data from passing through. Unlike MOUSETRAP [31] pipeline stages, which have latches that are normally transparent, the latches of the routing primitive are opened only after a new request is received. New data appears at the inputs of both banks of latches, but only one will be made transparent (enabled) for each new data item. The normally-opaque style guarantees that only correct data will be forwarded to subsequent stages.

Each bank of latches is accompanied by latch controller logic. The latch controllers are responsible for enabling and disabling the data latches. Each latch

controller takes as input handshaking signals from the input port (*Req*, *Ack*) and their respective output port (*Req0/Req1*, *Ack0/Ack1*). The handshaking signals use a 2-phase transition signaling protocol that can be in one of two phases: transaction pending or transaction complete (see section 2.4). The latch controller assesses the status of each port using XOR and XNOR gates (Figure 3.2), which function as equality testers for request and acknowledgment signals. The XOR gate partially enables the latch controller output when there is a pending transaction on the input port. The XNOR gate partially enables the latch controller output when there is a completed transaction on the corresponding output port.

The third enabling condition for the data latches depends on the value of B, the routing signal. B will partially enable exactly one bank of data latches. If the corresponding latch controller output is also enabled, then the latches will made transparent, and new data will pass through. The other bank of data latches will remain opaque, disabled by B.

The toggle element is used to convert an input *Req* transition to an output transition on the appropriate port. The toggle output for a specific port will transition once for every data item routed to that port. Like the data latches, the toggle element is enabled by the latch controller output and the routing signal, B. In Figure 3.2, there is an example design of the toggle element, implemented as a T latch. The toggle latch will only toggle when both the toggle, T, and enable, En, inputs are high.

The *Ack* signal to the left environment is generated by the XOR gate shown at the right in Figure 3.2. In this case, the XOR acts as a merge element [9]

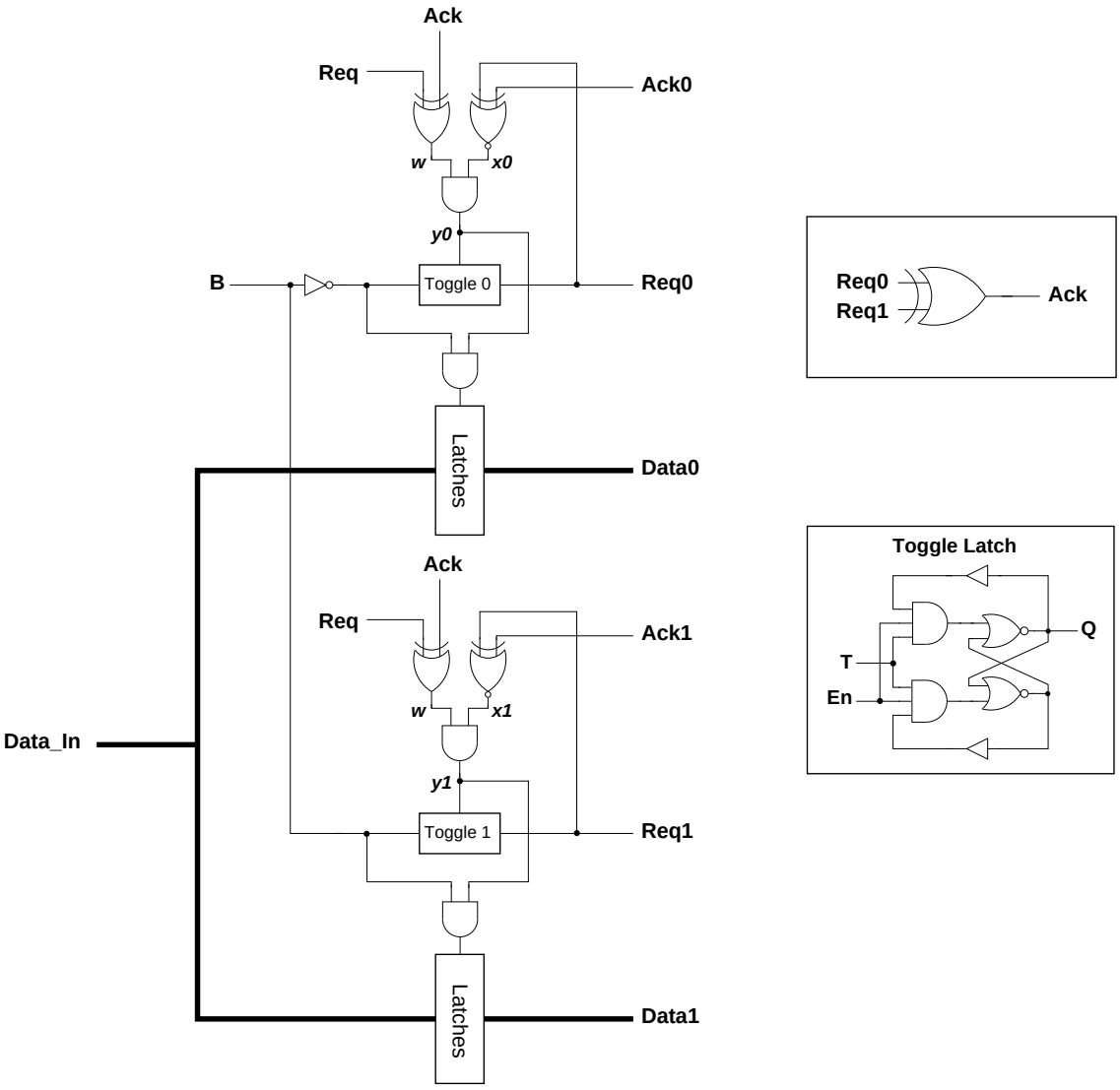


Figure 3.2: Routing Primitive

for two transition signals, *Req0* and *Req1*. Finally, the structure of the routing primitive allows for some concurrent operation between the two ports, described in the following section.

Basic Operation: We will consider a packet routed to port 0 for this example. The latch controller behavior is specified by three signal values: *w*, *x0*, and *y0*.

- Signal w is the output of an XOR gate that examines the left environment interface. The XOR is a *inequality tester* for the *Req* and *Ack* signals that generates a partial enable, w, to both sets of latches in the primitive. The latches are initally both opaque. When *Req* transitions on the left input channel, w is asserted high, partially enabling both latches. When the data passes through the latches to the appropriate output channel, *Ack* transitions, deasserting w and ensuring that the latches are opaque.

- Signal *x0* is the output of an XNOR gate that reflects the state of the right environment interface. Likewise, *x1* serves the same purpose for the bottom interface. The XNOR is an *equality tester* for the *Req0* and *Ack0* signals that partially enables the corresponding set of latches in the primitive. Initially, *x0* is asserted, since the right environment may accept a new request. When new data is forwarded to the right environment at the top interface (port 0), *Req0* transitions and *x0* is deasserted. When the request is acknowledged, *Ack0* transitions and *x0* is asserted, partially enabling the next operation.

- Signal *y0* combines the w and *x0* signals, and is asserted when a new input item has arrived and the following stage is ready to accept new data ($w = 1$ and $x0 = 1$) and deasserted after new data is routed to the following stage ($w = 0$ or $x0 = 0$).

Simulation: Initially, all inputs and outputs are low. w and *y0* are initially low since there are no pending input requests, thus disabling all toggle elements and data latches. *x0* is initially high since there are no pending output requests to port 0. An

important feature of this routing primitive is that, unlike MOUSETRAP pipeline stages [31], the latches in the routing primitive are *normally opaque* (disabled). The motivation for this approach is to prevent propagation of data bits through latches until after the routing decision is determined, thus saving dynamic power from unnecessary transitions.

First, new data and a stable B signal appear at the inputs. It is important that B is bundled with the *Req* transition according to the bundling constraint mentioned in section 2.4. Assuming that $B = 0$ for this data item, the toggle element and D-latches for port 0 will each be half enabled.

Next, *Req* transitions, then w is asserted. w and *x0* together fully enable the latch controller output, *y0*. With *y0* and the correct B signal asserted, the toggle element output transitions and the latches become transparent (enabled). Note that when w is asserted, it affects both latch controllers, and *y1* will also be enabled. However, since $B = 0$, the toggle and latches for port 1 will remain disabled.

The toggle output transition will cause four events to occur in parallel: (1) a *Req0* output transition is passed to the next stage, (2) the *Req0* transition is used as a feedback signal to disable *x0*, (3) an *Ack* transition is generated to the left environment, and (4) the *Ack* transition is used to disable w.

The end result is that *y0* will be deasserted, disabling the toggle and closing the latches. The data is now safely stored in the current stage, and the left environment (which was acknowledged) is free to send new data. There is a pending request to the next stage on port 0, awaiting an *Ack0* transition.

Concurrency Feature: An interesting feature of this design is that while a

request is pending on port 0, awaiting an *Ack0* transition, another request heading to port 1 is able to process a *full handshaking transaction*, thereby allowing this new input data to be routed even before the first item has been acknowledged. In fact, multiple successive input requests to port 1 can be processed in turn, even if port 0 remains stalled. This concurrency between ports can translate to performance benefits, since transactions to alternating destinations can be handled in parallel.

Reset: Resetting the routing primitive to an initial state is quite simple. It involves resetting both of the toggle elements to a 0 output. This will set both *Req0* and *Req1* to 0, as well as reset the *Ack* signal to 0. Along with inputs *Req*, *Ack0*, and *Ack1* set to 0 during reset, the routing primitive will be brought to the proper initialization state. Additionally, the T latch can be reset by adding extra gates on the feedback paths within the latch, which will not negatively impact the critical path delay through the primitive.

3.2.1 Performance Equations and Timing Constraints

This section presents an analytical evaluation of routing primitive performance and timing constraints. Performance is analyzed by looking at forward latency and cycle time. Latency is the delay through an empty primitive, and is important when looking at network performance for an initially empty network as well. Cycle time is the measure for operation under steady-state input conditions and reflects performance for a network with medium to high traffic. For cycle time, analytical equations for two distinct input patterns are created. The first case has consecutive

packets routed to the same destination, called the *single port* routing case. The second case has consecutive packets routed different destinations, called the *alternating port* routing case. The *alternating port* routing has better cycle time than the *single port* due to the concurrent operation of the two ports, and is described in detail below.

Timing constraints must be satisfied in order to guarantee correct operation of the routing primitive. These constraints specify some ordering of competing events and must be handled carefully in implementation. However, the timing constraints identified in the routing primitive are simple one-sided constraints that are not difficult to satisfy.

Performance: Performance is analyzed using two key metrics: forward latency and cycle time.

Forward latency is the time it takes a data item to pass through an initially empty primitive. For the routing primitive, this is the time from a *Req* transition to a corresponding *Req0* or *Req1* transition with valid data on the output channel. The example path is for data directed to port 0. The path consists of asserting *w*, *y0*, then, in parallel, a transition on the *Toggle0* element and opening of corresponding the data latches.

$$\mathbf{L} = T_{XOR_W\uparrow} + T_{AND_{Y0}\uparrow} + max(T_{Toggle0},\, T_{AND\uparrow} + T_{Latch}) \tag{3.1}$$

As described in the previous section, this path assumes that new data and a

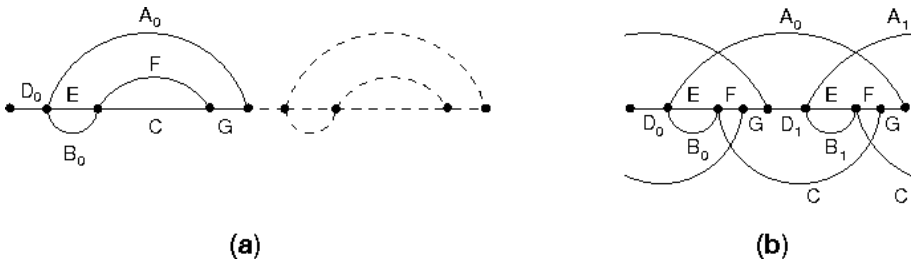


Figure 3.3: Cycle Time Diagrams for Routing Primitive directing consecutive packets to a) Single Port b) Alternating Ports

stable B signal were already present at the inputs of the primitive.

Cycle time is the time interval between successive packets passing through the primitive. A cycle consists of three events:

1. A *Req0* or *Req1* transition is passed to the right environment and a corresponding *Ack0* or *Ack1* transition is returned.

2. The latches are shut and the latch controller is reset, preparing for the next input.

3. An *Ack* transition is passed to the left environment and a *Req* transition with new data and B arrives at the input.

The routing primitive exhibits significantly different behavior depending on input patterns. Namely, cycle times may be different if consecutive packets are destined for alternating destinations. This is due to the fact that while one port is awaiting acknowledgment, the other is free to complete a full transaction. Figure 3.3 shows graphs representing the two types of cycle times. The variables in the equations correspond to arcs in the figure.

The primitives operate concurrently, with multiple paths active at the same

time. There are several synchronization points that require multiple threads to join in order to proceed. Each join in 3.3 is represented by a *m*ax expression in the cycle time equations. The longest of the joining paths will determine how execution continues, and ultimately affect the performance of the routing primitive.

In general, A paths, above the horizontal, are forward paths through the right environment, described above in (1). B, D, E, and F paths are internal to the primitive, and deal with setting and resetting the latch controller. C paths are reverse paths that cycle through the left environment with acknowledgment followed by new request plus data, mentioned in (3).

Paths with subscript 0 and 1 describe transactions on ports 0 and 1, respectively. The equations for cycle time are now presented for two simulation cases: successive routing to a single port and successive routing to alternating ports.

1. *Successive Routing to Single Port*: The cycle is measured as the amount of time between transition of *Req0* and the next, shown in Figure 3.3(a). The equation describes one full cycle on port 0 (input $B = 0$). The equation is the same for port 1, exchanging 0 for 1 in the subscripts.

$$\begin{aligned}
\mathbf{T}_{Single} &= D_0 + max(A_0,\, max(B_0, E) + max(C, F) + G) \qquad (3.2)\\
A_0 &= T_{RightEnv_0} + T_{XNOR_{X0}\uparrow}\\
B_0 &= T_{XNOR_{X0}\downarrow}\\
C &= T_{LeftEnv}\\
D_0 &= T_{AND_{Y0}\uparrow} + T_{Toggle0}\\
E &= T_{XOR_{Ack}}
\end{aligned}$$

$$F = T_{XOR_W\downarrow}$$

$$G = T_{XOR_W\uparrow}$$

2. *Successive Routing to Alternating Ports*: A full cycle of alternating ports is time between one transition of *Req0* and the next, shown in Figure 3.3(b). The equation describes two full cycles, one to port 0 immediately followed by one to port 1. The cycle time for a single flit in steady state is therefore half.

$$\mathbf{T}_{Alternating} = \frac{1}{2} \cdot max \left(\begin{array}{l} D_0 + max(A_0,\, S_A + max(S_C,\, C) + G), \\ \\ D_1 + max(A_1,\, S_B + max(S_D,\, C) + G) \end{array} \right) \quad (3.3)$$

$$A_0 = T_{RightEnv_0} + T_{XNOR_{X0}\uparrow}$$

$$A_1 = T_{RightEnv_1} + T_{XNOR_{X1}\uparrow}$$

$$B_0 = T_{XNOR_{X0}\downarrow}$$

$$B_1 = T_{XNOR_{X1}\downarrow}$$

$$C = T_{LeftEnv}$$

$$D_0 = T_{AND_{Y0}\uparrow} + T_{Toggle0}$$

$$D_1 = T_{AND_{Y1}\uparrow} + T_{Toggle1}$$

$$E = T_{XOR_{Ack}}$$

$$F = T_{XOR_W\downarrow}$$

$$G = T_{XOR_W\uparrow}$$

$$S_A = max(B_0,\, E)$$

$$S_B = max(B_1,\, E)$$

$$S_C = F + G + D_1 + S_1 + F$$

$$S_D \;=\; F + G + D_0 + S_0 + F$$

Clearly one can see the differences between Figure 3.3(a) and 3.3(b). The single port cycle is highly serial and requires all operations to complete for each cycle, before the next cycle can begin. The alternating port case allows for concurrency between adjacent cycles, improving performance.

Several conclusions can be drawn based on the performance equations. First, in the case of a very slow left environment, both scenarios evaluate to $D+max(B,\ E)+C+G$. This is expected, with the reverse path dominating the cycle time and setting the pace for execution.

In the case of a slow right environment, the single port case evaluates to $D+A$, while the alternating case evaluates to $\frac{1}{2}\cdot(D+A)$ on average. This is expected, since while awaiting an acknowledgment one port, the other is free to complete multiple full transactions.

For the case of both environments operating very quickly, both scenarios evaluate to $D + max(B,\ E) + F + G$. With fast responses from the right environments, the routing primitive can operate very efficiently, but as acknowledgments are generated with longer latency, the performance for the single port case quickly falls far behind. Therefore, the key to good performance at the root, which is critical for tree performance, is either generating fast acknowledgments from following stages or biasing the input packets to arrive with alternating destinations. Experimental results for these and other cases are presented in Chapter 5.

Timing Constraints: There are two simple, one-sided timing constraints

that are must be satisfied in order to guarantee correct operation.

The first is a bundling constraint on the input port, specifically regarding the B input. This signal is used as the routing decision for the data packet and should be stable when a *Req* transition occurs. Since B is part of the data packet, this should be guaranteed as part of the general bundling constraint, as described in 2.4.

The second constraint is on the toggle element, if a T latch is used (as indicated in Figure 3.2). The desired functionality is that the correct toggle element will toggle once for a corresponding *Req*. Since a T latch will continue to oscillate when enabled, it must be disabled after the first transition and before the second can occur.

$$T_{Toggle_Feedback} > T_{XNOR} + T_{AND} \tag{3.4}$$

To accomplish this, the feedback loops, pictured in Figure 3.2, must have adequate delay.

3.3 Arbitration Primitive

The routing primitive presented in the previous section accepts data on a single input port and conditionally routes that data to exactly one of two output ports. The arbitration primitive provides complementary functionality, accepting data from exactly one of two input ports and forwarding the data to a single output port, as shown in Figure 3.4.

One distinct challenge in designing an asynchronous arbitration primitive is that, unlike synchronous design, competing request inputs arrive in *continuous time*.

The asynchronous primitive must be able to select between competing requests that may arrive simultaneously or separated by large intervals. This functionality is performed by a mutual exclusion element (mutex), an analog arbiter circuit. The mutex grants access to a shared resource (storage latches) to exactly one of two competing requests.

Section 3.3.1 describes an earlier design for the arbitration primitive, proposed by Gill and Singh in [19], and a more optimized version proposed by Carlberg and Nowick in [3]. Section 3.3.2 discusses four new significant enhancements built on this framework for: datapath configuration, handling of large packets (i.e. multi-flit capability), power optimization, and performance-optimized insertion of reset logic. The challenge of adding functionality and making optimizations to the existing designs is minimizing the impact on performance. The enhancements are substantial contributions that add new functionality and reduce dynamic power consumption, to make the primitive practical for high-performance applications.

3.3.1 Previous Work

Carlberg and Nowick began initial work on the asynchronous interconnect at Columbia University [3]. The main focus was the development and optimization of the arbitration primitive. In the following sections, we discuss the two alternative basic designs developed at Columbia, a *throughput-oriented primitive* (TPP) and a *latency-oriented primitive* (LP), which serve as a foundation for this work. Then, in section 3.3.2, further enhancements developed at Maryland are described, as well

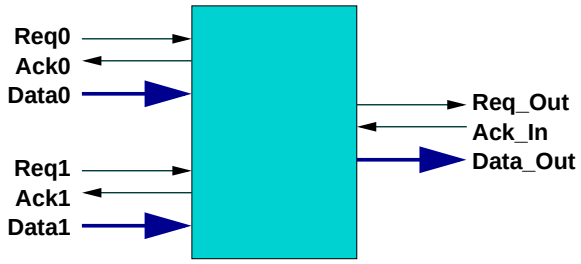


Figure 3.4: Block Diagram of an Arbitration Primitive

as analytical performance equations for latency, cycle time and timing constraints.

Block Diagram: An arbiter is used to control access to a shared resource by selecting exactly one pending request out of many. In the case of the arbitration primitive, the shared resource is the output port, and the requesters are two other primitives. As shown in Figure 3.4, the arbitration primitive has two input ports and a single output port. Transition signaling with bundled data is used for all communication.

Components: The arbitration primitive was designed as part of a MOUSETRAP-style pipeline to achieve the goal of low latency and high throughput. As a first-cut solution, two functional components were combined to form a single "merge-with-arbitrate" primitive (called throughout this thesis an "arbitration primitive"): 1) an *arbiter* and 2) a *merge-without-arbitration* (MWA) element, based on contributions from [19]. The arbiter component takes two inputs and allows only one, the winner, to pass to its respective output in a given operation. The losing input must wait until it is permitted to advance by the arbiter. The merge-without-arbitration component combines two input streams into a single output stream, on the condition that the inputs are already guaranteed to be mutually exclusive. Namely,

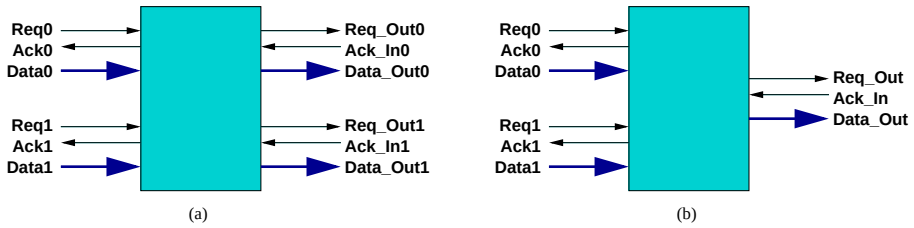


Figure 3.5: Block Diagrams of (a) an Arbiter and (b) Merge-without-Arbitration Components

a transaction on one port must fully complete before a request can appear on the other port.

Since the arbiter component guarantees mutually exclusive activity at its output, the two can naturally be combined to form an initial version of the desired composite arbitration primitive. Figure 3.5 shows how the the arbitration primitive can be formed by combining arbitration and merge-without-arbitration functions from [19].

The two arbitration primitive designs, *throughput-oriented primitive* (TPP) and *latency-oriented primitive* (LP), are presented below [3]. The TPP provides a faster acknowledgment on the input channels than LP, which may improve throughput for certain input conditions, however it uses more area, power, and has longer forward latency. The LP is an optimized version of TPP that reduces the area, power and forward latency, however has worse cycle time when accepting successive packets from a single port. Different applications may benefit from using either of the alternative designs. Each primitive was further optimized as part of this research. Enhancements are described in detail in section 3.3.2.

3.3.1.1 Throughput-Oriented Primitive (TPP)

This section presents the structure and function of the Throughput-Oriented Primitive (TPP), the first of the designs presented in [3]. This design is formed simply by concatenating the earlier Arbiter and MWA elements from [19]. It is called the TPP because it provides a fast acknowledgment to the left environment compared to the alternative design discussed in the next section. The faster acknowledgment completes the transaction with the previous stage faster, which can improvement throughput.

Architecture of the TPP: The design features seven transparent D-latches (numbered L1 through L7), as shown in Figure 3.6. Latches L1, L2, L5, L6, L7 are all normally transparent (enabled). Latches L3 and L4 are normally opaque (disabled).

There is a mutual exclusion element (ME), or mutex, that performs the arbitration functionality. The mutex is a four-phase, return-to-zero module that operates as follows: (1) Initially both inputs (*Req_ME*) and outputs (*Ack_ME*) are low (2) One (or both) *Req_ME* wire(s) transition to high, signaling a request (or requests) (3) Exactly one *Ack_ME* transitions high, corresponding to the winning request (4) After some time, the winning *Req_ME* wire transitions low, signaling the end of the transaction (5) The corresponding *Ack_ME* transitions low, returning the mutex to its initial state.

Note that during the time the *Ack_ME* wire is high, a request may arrive (or disappear) on the other port, but will have no effect on the state of the mutex. After

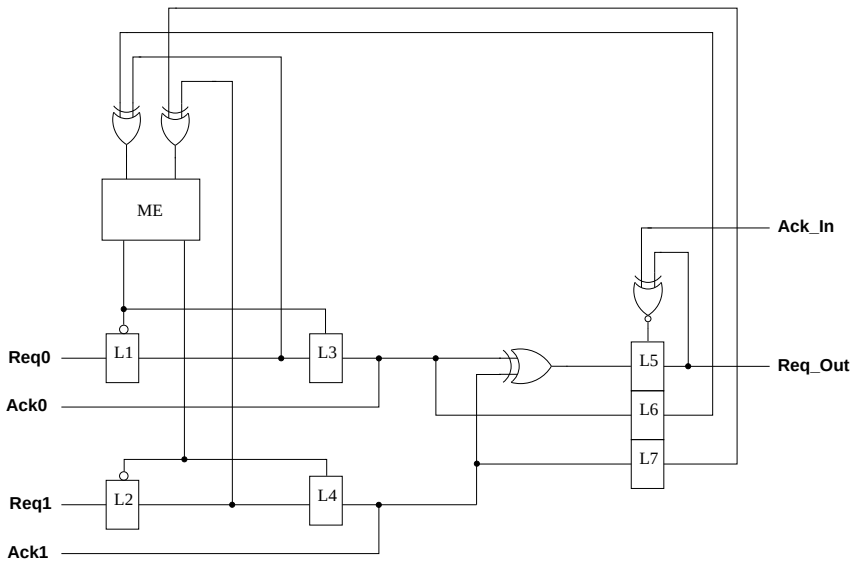


Figure 3.6: Basic Control of Throughput-Oriented Primitive (TPP)

a transaction is completed, the next transaction may begin immediately.

In addition to the latches and mutex, three XOR gates are used as merge elements for transition-signaling wires and an XNOR gate is used as a MOUSETRAP latch control.

Basic Operation: Initially, all inputs and outputs are low. The mutex has no pending requests, indicating that both mutex output wires are low. L3 and L4 are opaque, since mutex outputs are low, and are outputting low. All other latches are transparent (enabled), with output low. Therefore, all signal wires are low, except for the XNOR latch control output, which is high, enabling L5, L6, and L7.

Req0 transitions from low to high, indicating the start of a transaction. Since L1 and L2 are transparent, *Req0* passes through. It is halted at the input of L3, which is currently opaque. *Req0* continues to the input of the XOR, which causes a transition at its output, generating a request to the mutex. Since there are no

competing requests, the mutex responds with a transition from low to high on its acknowledgment output corresponding to *Req0*.

The rising acknowledgment wire performs two actions in parallel: (1) it closes L1, latching the current value of *Req0* and (2) opens L3, allowing *Req0* to pass through. The opening of L3 spawns three concurrent threads in the primitive: (1) L3 output is used as an acknowledgment (*Ack0*) to the previous stage; (2) the same output continues through a transparent L6, causing a transition on the XOR at the mutex input, and resetting the mutex; (3) it causes a transition on the XOR output at the input of L5, which it turn passes through L5, becoming *Req_Out* to the next stage, as well as closing the L5-7 latches through the feedback loop of the MOUSETRAP XNOR control.

At this point, the mutex lowers its acknowledgment output, completing its return-to-zero protocol. As a result, L1 becomes transparent and L3 is made opaque again. The primitive can now accept a second request on *Req0* through the transparent L1 latch. Note that at any time during this simulation, *Req1* is free to transition and make a request to the mutex. L2 remains transparent and the request can get all the way to the input of the mutex, but will be stopped at the input of L4, which provides protection to the MWA stage. An interesting property that results from this behavior is that the request on the opposing port will win the mutex as soon it is reset as part of the first transaction. In a heavily loaded scenario, the mutex defaults to a toggling behiavor that alternating between incoming requests.

Shortly after the *Req_Out* transitions, L5-7 are made opaque, retaining their values. The *Req_Out* transition will eventually be acknowledged by the next stage by

a transition on *Ack_In*, which will open L5-7, allowing new values to pass through. Note that a new input transaction can begin even if there is no acknowledgment from the right environment.

The primitive can complete two full transactions with the left environment when there is a stalled right environment. This is due to the fact that an acknowledgment to the left environment (*Ack0* or *Ack1*) is generated early in the cycle, at the opening of L3 or L4.

In the case where two *Reqs* occur simulataneously, the mutex will generate only one acknowledgment, and the operation will continue as described above.

Performance Equations: Analytical performance equations were presented in [3], but updated equations for latency and cycle time, as well as timing constraints, can be found later in section 3.3.3.

Datapath: No datapath for the TPP design was presented in the original report. Figure 3.11 in section 3.3.2 shows the new datapath configuration, including positioning of the data latches, as well as data mux allocation and control. This configuration was chosen amongst several alternatives as the best balance of area, power, and performance.

3.3.1.2 Latency-Oriented Primitive (LP)

This section presents an alternative earlier design, called the Latency-Oriented Primitive (LP), also presented in [3], that seeks to improve the forward latency of the primitive. The architecture and basic operation are discussed below. Analytical

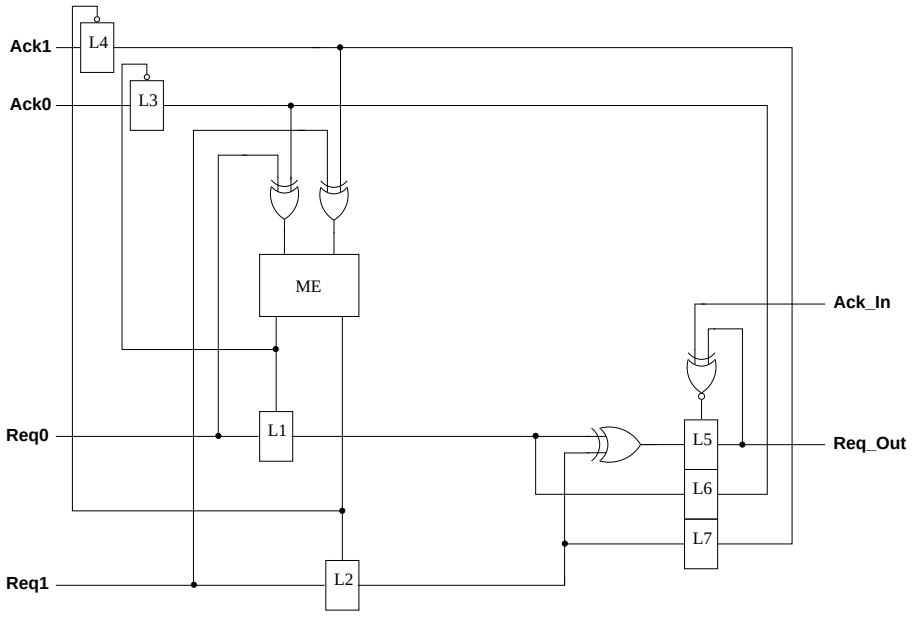


Figure 3.7: Basic Control of Latency-Oriented Primitive (LP)

equations for latency and throughput, as well as timing constraints, can be found in section 3.3.3.

Architecture of the LP: The design features seven transparent D-latches (numbered L1 through L7), as shown in Figure 3.7. Note that the forward input-to-outputs paths now each have one fewer latch than the previous TPP design, while latches are now added to the left *Ack0* and *Ack1* paths.

Latches L3, L4, L5, L6, L7 are all normally transparent (enabled). Latches L1 and L2 are normally opaque (disabled). There is a mutual exclusion element (ME), which executes the same four-phase return-to-zero protocol described in the previous section. Like the TPP design, three XOR gates are used as merge elements for transition-signaling wires and an XNOR gate is used as a MOUSETRAP latch control. The core structure of the TPP is preserved in this design, with two of the D-latches relocated.

Basic Operation: Initially, all inputs and outputs are low. The mutex has no pending requests, indicating that both mutex output wires are low. L1 and L2 are opaque, since mutex outputs are low, and are outputting low. All other latches are transparent (enabled), with output low. Therefore, all signal wires are low, except for the XNOR latch control output, which is high, enabling L5, L6, and L7.

Req0 transitions from low to high, indicating the start of a transaction. It is halted at the input to L1, since the latch is opaque. *Req0* also continues to the input of the XOR, which causes a transition at its ouput, generating a request to the mutex. Since there are no competing requests, the mutex responds with a transition from low to high on its acknowledgment output corresponding to *Req0*.

The rising acknowledgment wire performs two actions in parallel: (1) it opens L1, allowing *Req0* to pass through, and (2) closes L3, latching the current value of *Ack0*. The opening of L1 performs three operations in the primitive: (1) L1 output continues through a transparent L6, causing a transition on the XOR at the mutex input, and resetting the mutex; (2) the same L1 output appears at L3 input, which is currently opaque; (3) it causes a transition on the XOR output at the input of L5, which it turn passes through L5, becoming *Req_Out* to the next stage, as well as closing the L5-7 latches through the feedback loop of the MOUSETRAP XNOR control.

At this point, the mutex lowers its acknowledgment output, completing its return-to-zero protocol. As a result, L3 becomes transparent and L1 is made opaque. The opening of L3 causes a transition on its output, generating *Ack0* to the left environment, and completing the transaction. Note that at any time during this

simulation, *Req1* is free to transition and make a request to the mutex. L2 remains opaque the entire time, preventing *Req1* from entering the MWA stage. An interesting property that results from this behavior is that the request on the opposing port will win the mutex as soon it is reset as part of the first transaction. In a heavily loaded scenario, the mutex defaults to a toggling behiavor that alternating between incoming requests.

Performance Equations: Analytical performance equations for the basic LP primitive design were presented in [3]. In that report, Carlberg showed that LP saves 1 D-latch delay on the critical path, improving forward latency. Updated equations for latency and cycle time, as well as timing constraints, can be found in section 3.3.3.

Datapath: There are two basic operations that must take place on the datapath: (1) one of the two *Data* inputs must be selected to advance and (2) data must be latched to prevent overrun from the previous stage. The selection operation is performed by a multiplexer, with some logic to generate the select input. Carlberg presented two alternative approaches in [3] for datapath configuration and now we present the first, which is used as the foundation for the enhanced LP module.

Figure 3.8 shows the LP design with added datapath. The second output of the mutex (ME) is chosen for the multiplexer select input (*mux_sel*). If the *Req0* input wins the mutex, then *mux_sel* will remain low, allowing *Data0* to advance. If the *Req1* input wins the mutex, then *mux_sel* will transition to high, allowing *Data1* to advance. There is a timing constraint introduced in the design that challenges the bundling between selected data and the input to L5. This constraint is addressed

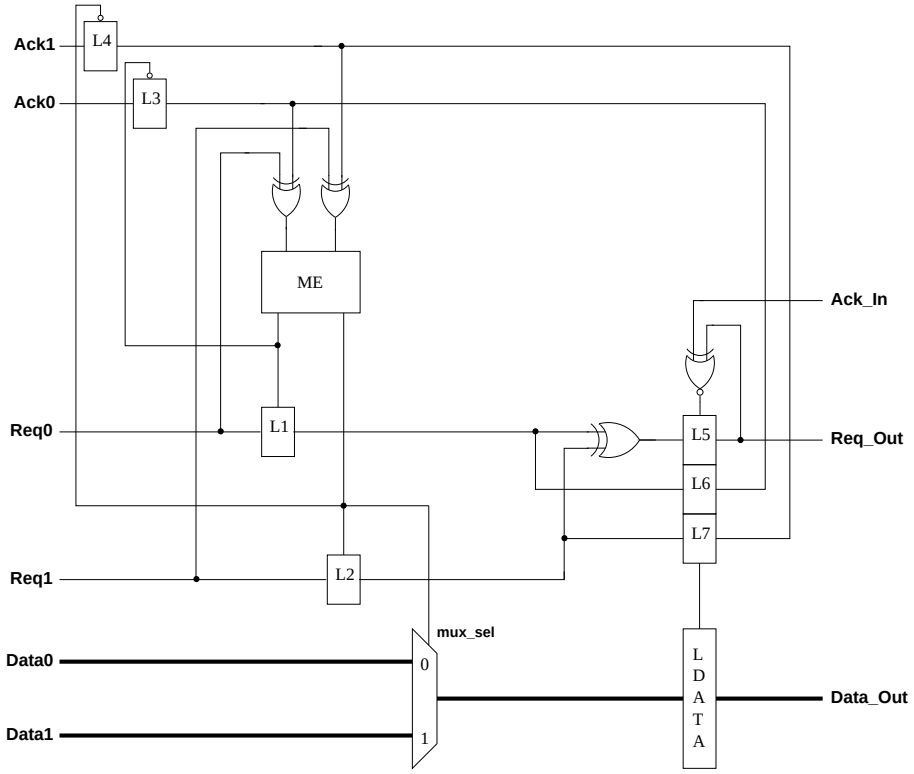


Figure 3.8: Basic Control and Datapath of Latency-Oriented Primitive (LP)

later in section 3.3.3.

3.3.2 Enhanced Primitives

The effort at Maryland focuses on incorporating the Mesh-of-Trees network as part of the PRAM-on-Chip project, led by Dr. Uzi Vishkin. The project developed an on-chip parallel processing architecture, called eXplicit Multi-Threading (XMT), which requires a high-throughput, low-latency interconnection network for performance. In this section, we present further optimizations to LP and TPP modules for use in XMT. We will use LP as the example for describing the implementation of the optimizations, then show how they are applied to TPP.

Four new enhancements presented in this section address the key areas of power consumption, added functionality, and initialization. The first three are optimizations that apply to both TPP and LP arbitration primitives. The fourth applies only to the TPP primitive.

- *Power Optimization*: The control logic for the datapath is modified to eliminate unnecessary transitions that increase the dynamic power consumption.

- *Multi-Flit Capability*: In XMT and other architectures, wide packets are sent as a series of narrower packets, called *flits*, to maximize bandwidth utilization. Multiple flits of a wide packet travel through the network as one contiguous multi-flit packet. Below, we show how multi-flit capability is added with support for an arbitrary number of flits per packet. This is a new contributionNo prior published work exists on transition-signaling asynchronous pipelines

- *Optimized Reset*: With any added functionality, performance may suffer as a result of added logic on critical paths. The addition of initialization logic, used to reset the network, is no exception. We present our optimized partial reset implementation that is sufficient to bring the arbitration primitive to an initial state.

- *TPP Datapath Configuration*: Datapath of the TPP arbitration primitive was not developed as part of [3]. Therefore, several alternative arrangements for storage latches and multiplexers of the datapath were evaluated for performance, area, and power. The resulting configuration, shown in Figure 3.11, represents the best balance of the three design constraints.

First, new optimizations for power, handling of multi-flit packets, and reset that apply to both TPP and LP primitives are introduced. Then, datapath configuration and application of these new enhancements for TPP are presented. Finally, section 3.3.3 defines performance equations for latency and cycle time, as well as timing constraints for each of the enhanced designs.

Power Optimization: Since the majority of cells in the primitive are on the datapath, reducing unnecessary transitions can deliver significant power savings. The datapath logic consists of multiplexers and transparent latches. The multiplexers select between the two data inputs, *Data0* and *Data1*, and provide input to the data latches. The multiplexer selection signal, *mux_sel*, is the focus of this optimization. Earlier designs of the latch-based *mux_sel* designs [3] for the arbitration primitive allowed the selection signal to transition at multiple times during

an operation. The power optimization presented below limits the transitions to once per cycle, thus reducing unnecessary transitions on multiplexers and latches.

The design in Figure 3.8 uses the second output from the mutex (ME) directly as *mux_sel*. While this is functionally correct, it is not power efficient because the mutex is reset during each cycle of operation. Any request the *Req1* port, will result in two transitions of *mux_sel*. The first transition from low to high occurs when the mutex acknowledges the request, and then another transition back to low occurs when the mutex is reset.

This behavior can cause unnecessary transitions for the multiplexer outputs. In the case of consecutive packets arriving on the *Req1* port, the right bank of data latches may also experience extra transitions due to the mutex being reset. If the packets are sufficiently spaced in time, a transparent bank of data latches on the right may propagate these transitions to future stages.

To eliminate this problem, an SR latch is introduced to drive *mux_sel*, shown in Figure 3.9. The set (S) and reset (R) inputs are connected to the second and first outputs of the mutex, respectively. When a request wins the mutex, the correct value of *mux_sel* will appear at the output of the SR latch. When the mutex is reset (with both outputs low), the SR latch output will keep the same state.

Note that this optimization applies to both LP and TPP modules.

Multi-Flit Capability: In many architectures, network packets may have different sizes. The interconnection network, however, has a fixed width for a specific implementation. Rather than designing a network for the widest packet, which wastes bandwidth when narrower packets are sent, wide packets can be sent as

a series of narrow packets, called *flits*. To ease the process of reconstructing the original packet at the destination, the entire multi-flit packet remains intact within the network, traveling one after the other, uninterrupted.

The earlier designs performed arbitration on individual packets which did not guarantee the order in which packets would advance through the fan-in tree. The goal of this enhancement is to bias the arbitration decision in a primitive to allow an entire multi-flit packet to advance intact through the fan-in tree of the network. This is a new contribution as there is no prior published work on implementing multi-flit protocols for high-performance asynchronous pipeline primitives.

One packet is defined as one or more flits, where a flit is the smallest granularity of data that can be sent through the network. In the XMT processor, for example, a flit contains one word (32 bits) of data, routing address, plus some extra bits used by the processor. The load word (lw) command requires one flit per packet, the requested address, while the store word (sw) command requires two flits, one for destination address and one for data. In the earlier arbitration primitive designs, reordering was free to occur within the fan-in tree, since arbitration has no explicit bias towards selecting one request from another. In order to accomodate multi-flit packets, hardware is added to detect and implement the multi-flit protocol.

Figure 3.9 shows the proposed enhanced design for LP with multi-flit capability. Each flit now contains an extra bit, called the *glue bit*, to denote whether the following flit on the same port is part of the same packet. A multi-flit packet, therefore, is defined as a series of flits with glue bit equal to one, followed by one flit with glue bit equal to zero. This definition is useful for a couple of reasons. First,

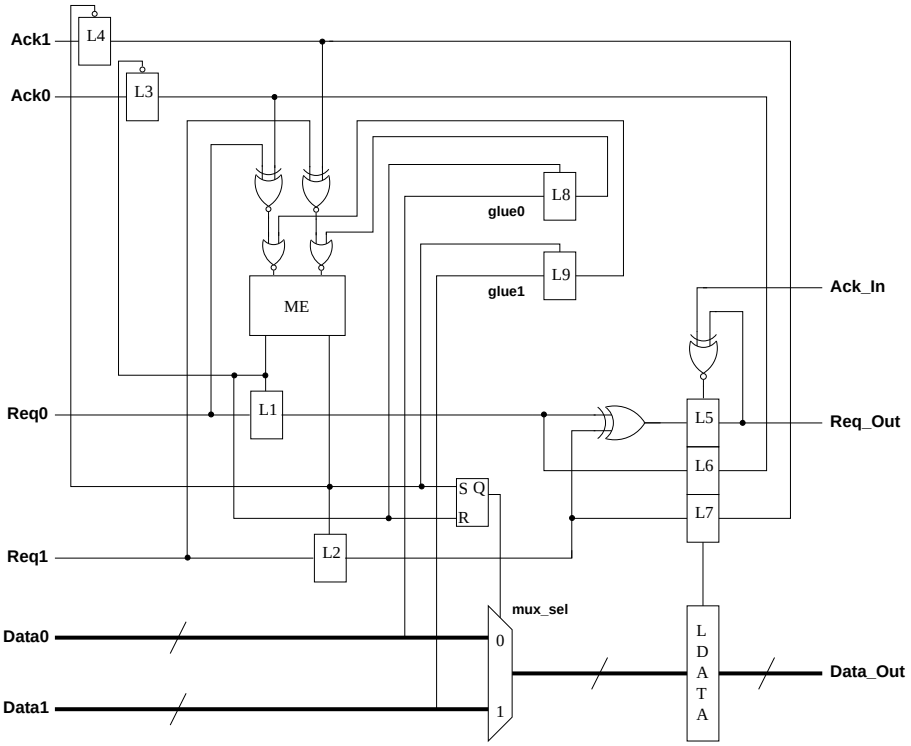


Figure 3.9: LP with Multi-Flit Capability and Power Optimization

defining a multi-flit packet in a distributed fashion, on a per-flit basis, simplifies the hardware. Second, by adding this functionality, the network can handle any size of packet that an application requires.

In order to bias the selection of the mutex, so the next flit of a multi-flit packet is guaranteed to advance, we employ a method that we dubbed "Kill your rival". When the first flit of a multi-flit packet wins the mutex, the opposing request input to the mutex is forced to zero, or "killed". This either prevents future requests on the other port from occuring, or in the case where a request was already pending, kills the opposing request until the entire multi-flit packet has passed through the arbitration primitive. Recall from the description of the mutex operation in section 3.3.1.1 that while the mutex has acknowledged one request, another request on the

opposing port can appear or disappear without affecting the operation. The kill function is achieved using a NOR gate located at the input of the mutex.

Once the mutex has made a new decision, one of the multi-flit latches, L8 or L9, is made transparent (enabled) by the corresponding mutex output. The input to the multi-flit latch is the *glue bit* from the corresponding *Data* input, which, if high, becomes a kill signal to the opposing *Req* at the mutex. When the mutex resets, the latch is closed. It is important that the glue bit reaches the NOR gate input before the mutex is reset, so in the case of a multi-flit packet, a rival input will be killed, and the next flit to advance will be from the same input port.

While the additional hardware is minimal, two transparent D-latches and two NOR gates, the NOR gates are on the critical execution path and do have some impact on performance, as explained in section 3.3.3. Also note that the multi-flit enhancement applies to both LP and TPP modules.

Optimized Reset: For correct operation, the arbitration primitive must be initialized so the latches are in the desired known state, the mutual exclusion element is reset, and request and acknowledgment outputs are deasserted. The addition of initialization logic can hurt performance if added on critical paths and increases the area requirements of the design. The goal of this reset implementation is to provide the necessary functionality while minimizing the performance and area overheads. To accomplish this goal, a partial reset of control latches is implemented, with some minor logic additions on non-critical paths. This approach limits performance and area overheads and is sufficient to bring the primitive to the desired initial state.

The arbitration primitive with added reset logic is shown in Figure 3.10. This

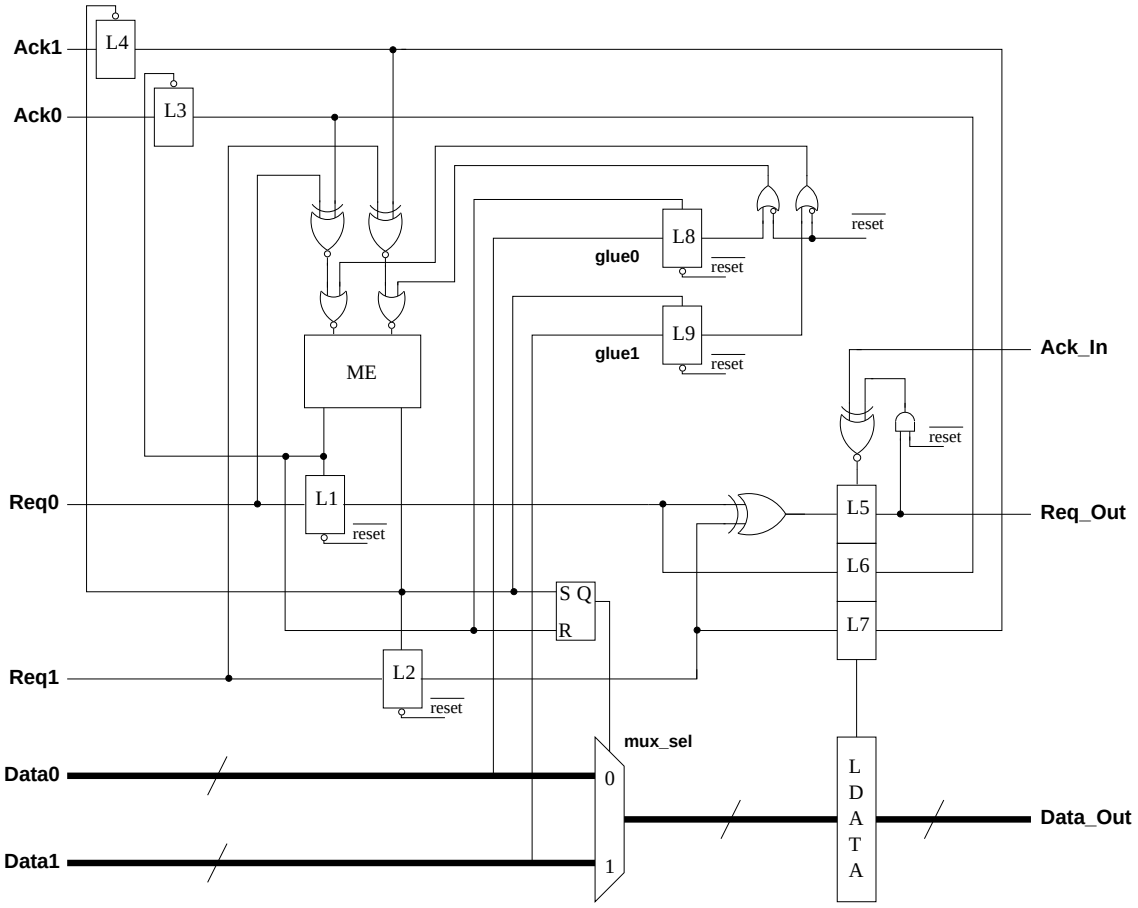


Figure 3.10: New Enhanced LP Arbitration Primitive with Power Optimization, Multi-Flit Capability and Optimized Active-Low Reset

functionality is desirable at power-on, to guarantee that the fan-in tree is ready to accept new packets. A specific application may also find it useful to flush the network of all packets.

Reset is accomplished by setting the nine control latches to a known state with known output values. We do this in three parts: resetting the mutex, modifying a partial set of latches enabled by the mutex, and making the set of latches on the right intially transparent.

First, the mutex is reset by exploiting the *kill your rival* functionality imple-

mented as part of multi-flit capability. The mutex outputs serve as latch enable signals for latches L1, L2, L3, L4, L8 and L9.

Next, the latches enabled by the mutex outputs are evaluated to decide which require an active-low asynchronous reset. Transparent latches with reset in the standard cell library [2] have more delay, and require higher area and power. Therefore, the effect on performance will be minimized by using the minimum number of latches with reset. The latches controlled by the mutex outputs can be divided into two groups: initially enabled and initially disabled. Only the latches that are initially disabled require an active-low reset, since their output value will not be known. The initially disabled latches are L1, L2, L8, and L9. The initially enabled latches (L3 and L4) are transparent, and will propagate values from their input to output. By limiting the latches with active-low reset, we minimize the performance penalty. L1 and L2 are the only latches with reset on the critical path of the LP arbitration primitive.

Finally, latches L5, L6, and L7 are enabled by setting the feedback input of the XNOR to low (the *Ack_In* will also be low at reset). By enabling these latches, the deasserted outputs of L1 and L2 will: (1) deassert *Req_Out* through the XOR and latch L5, (2) deassert *Ack0* and *Ack1* through L3 and L4 respectively, (3) reset the mutex XNOR controls, since request inputs *Req0* and *Req1* will also be deasserted at reset. This completes the optimized partial reset operation. When the reset signal transitions to high, the primitive will be in the initial state.

Enhanced TPP: The three enhancements mentioned in the previous sections all apply to TPP as well. However, the basic *throughput-oriented primitive*

(TPP) from [3] lacked datapath configuration for latches and multiplexing. Therefore, several options for datapath configuration were evaluated and one was chosen. This section presents the new fully-optimized TPP arbitration primitive with new datapath and enhancements.

There are three possible locations for the multiplexing of the two data inputs: before the first level of data latches, in-between the two levels, or after. We refer to these options as *early*, *mid*, and *late* muxing respectively.

Each of the options was evaluated for area and power costs, as well as ease-of-design. The *early* muxing was found to introduce difficult timing constraints, though it had the lowest area overhead. The *late* muxing had timing constraints similar to *mid*, but required an extra bank of data latches, increasing the area overhead. For the best balance of ease-of-design and area, the muxes were placed in-between according to the *mid* configuration.

Figure 3.11 shows the basic TPP module with *mid* muxing and data latches. Two banks of latches are used to store *Data0* and *Data1* values. Latch L1 and the *Data0* latches share the same enabling signal, and are normally transparent. The same applies for L2 and *Data1* latches. The new enhanced TPP arbitration primitive with power optimization, multi-flit capability and reset is shown in Figure 3.12.

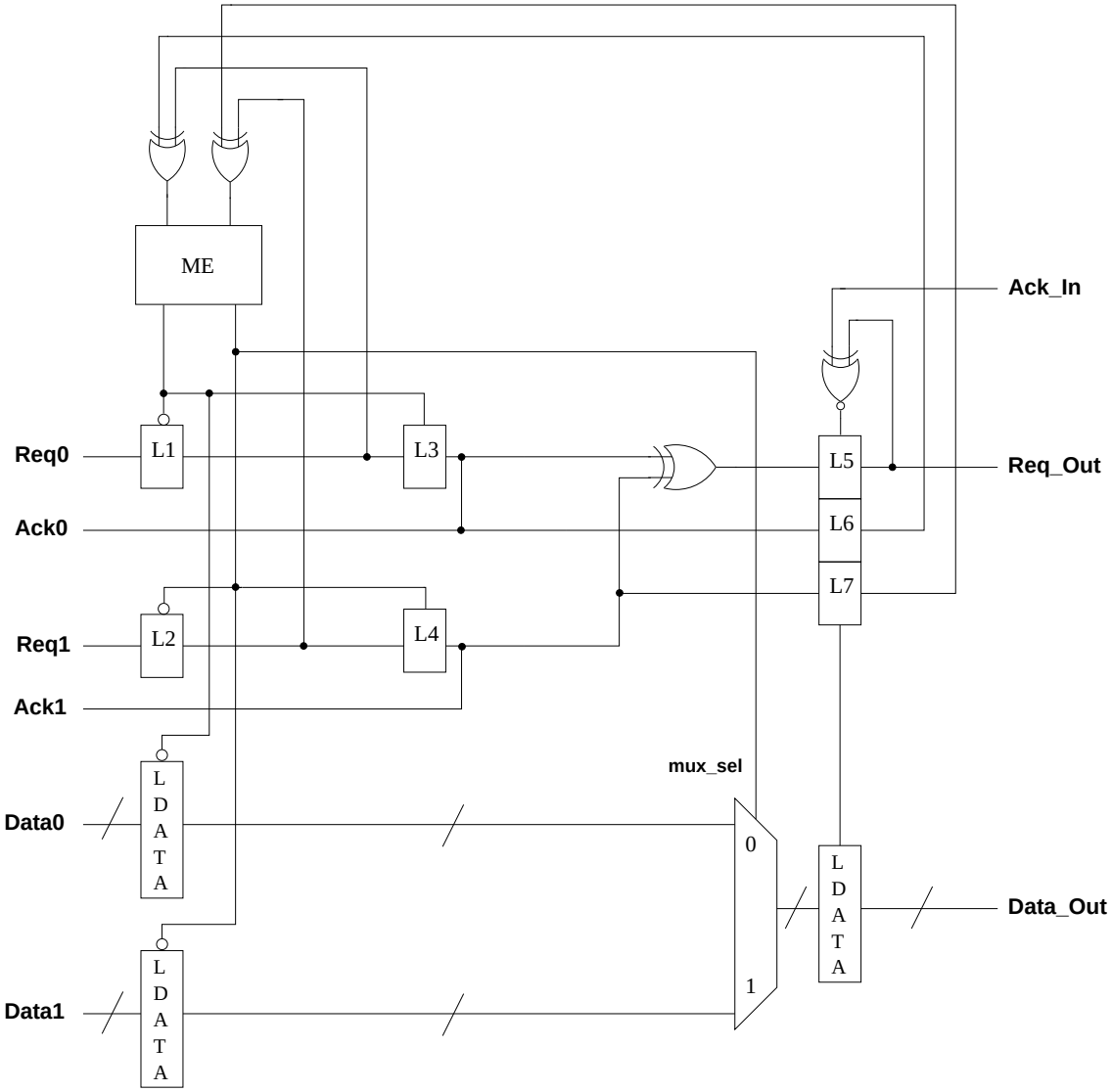


Figure 3.11: Basic Control and Datapath of Throughput-Oriented Primitive (TPP)

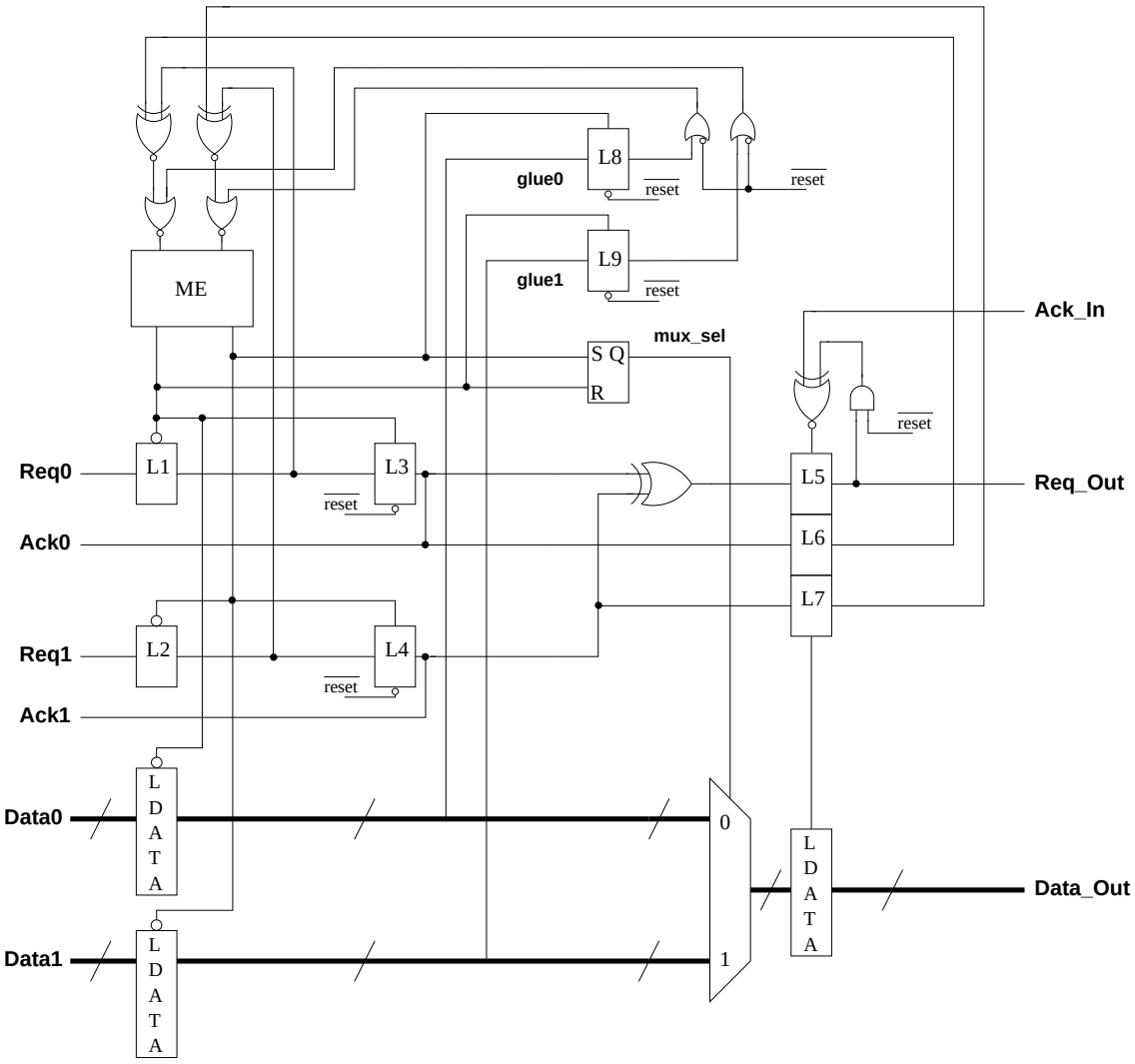


Figure 3.12: New Enhanced TPP Arbitration Primitive with Power Optimization, Multi-Flit Capability and Optimized Active-Low Reset

3.3.3 Performance Equations and Timing Constraints

This section presents an analytical evaluation of the arbitration primitive performance and timing constraints.

Performance is analyzed by looking at forward latency and cycle time. Latency is the delay through an empty primitive, and is important when looking at network performance for an initially empty network as well. Cycle time is the measure for operation under steady-state input conditions and reflects performance for a network with medium to high traffic. For cycle time, analytical equations for two distinct input patterns are created. The first case has packets arriving at the same input port, called the *single port* arbitration case. The second case has packets arriving at both input ports, called the *alternating port* arbitration case. Both the enhanced LP (Fig. 3.10) and TPP (Fig. 3.12) arbitration primitives are evaluated under these input conditions. The *alternating port* routing has better cycle time than the *single port* for both primitives due to concurrent operation between the two ports, and is described in detail below.

Timing constraints must be satisfied in order to guarantee correct operation of the routing primitive. These constraints specify some ordering of competing events and must be handled carefully in implementation. However, the timing constraints identified in the arbitration primitive are simple one-sided constraints that are not difficult in practice to satisfy.

Performance: Performance is analyzed using two key metrics: forward latency and cycle time.

Forward latency is the time it takes a data item to pass through an initially empty primitive. For the arbitration primitive, this is the time from a *Req* transition to a corresponding *Req_Out* transition. The path includes acquiring the mutex, and generating a new *Req_Out* transition. Latency equations for LP (3.5) and TPP (3.6) are presented below. The equations displayed are the same as those reported in [3].

$$\mathbf{L}_{LP} = T_{XNOR\uparrow} + T_{NOR\uparrow} + T_{ME\uparrow} + T_{L1_{G\rightarrow Q}} + T_{XOR} + T_{L5_{D\rightarrow Q}} \tag{3.5}$$

$$\mathbf{L}_{TPP} = T_{L1_{D\rightarrow Q}} + T_{XNOR\uparrow} + T_{NOR\uparrow} + T_{ME\uparrow} + T_{L3_{G\rightarrow Q}} + T_{XOR} + T_{L5_{D\rightarrow Q}} \tag{3.6}$$

Subtracting the two equations shows that the difference is one latch $D \rightarrow Q$ delay, and the reason LP is regarded as a latency-optimized primitive.

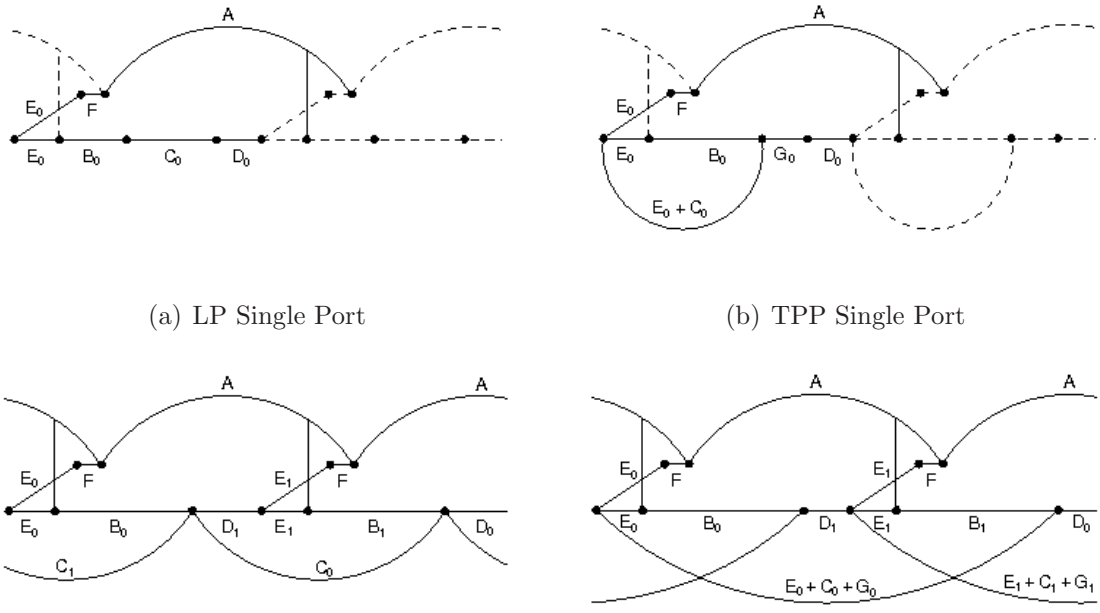
Cycle time is the time interval between successive flits passing through the primitive. A cycle of stage N consists of three events:

1. A *Req_Out* transition is passed to the right environment and an *Ack_In* transition is returned.

2. Stage N resets the mutex, preparing for the next input.

3. An *Ack* transition is passed to the left environment and a *Req* transition with new data arrives at stage N.

The arbitration primitives exhibit different behavior depending on input patterns. Namely, cycle times may be different if consecutive *Reqs* arrive at the same port versus arriving at alternating ports. The first case, which exercises a single port, may occur in a stage if many packets arrive from the same set of sources, also



(a) LP Single Port

(b) TPP Single Port

(c) LP Alternating Ports

(d) TPP Alternating Ports

Figure 3.13: Diagrams of Arbitration Primitive Cycle Times

in the case of multi-flit packets. The alternating case is likely in situations with heavy load and contention, most notably at the root of the fan-in tree.

The cycle times for the various input patterns are now discussed. The variables in the equations refer to arcs in Figure 3.13. The primitives operate concurrently, with multiple paths active at the same time. There are several synchronization points that require multiple threads to join in order to proceed. Each join in 3.13 is represented by a *max* expression in the cycle time equations. The longest of the joining paths will determine how execution continues, and ultimately affect the performance of the arbitration primitive.

In general, A paths, above the horizontal, are forward paths through the right environment, described above in (1). This path is shared by both input ports. B, D,

and E paths are internal to the primitive, and deal with resetting and setting of the mutex. C paths, below the horizontal, are reverse paths that cycle through the left environment with acknowledgment and new request plus data, mentioned in (3).

Paths with subscript 0 and 1 describe transactions on ports 0 and 1, respectively. The equations for dual operating modes of the LP and TPP are now presented, followed by a brief analysis.

1. *LP single port*: The cycle is measured as the amount of time between one rising edge of ME_0 output and the next, shown in Figure 3.13(a). The equation describes one full cycle on port 0.

$$\begin{aligned} \mathbf{T}_{LP\_Single} &= max(A,\, B_0 + C_0 + D_0 + E_0) \qquad (3.7) \\ A &= T_{L5_{D\rightarrow Q}} + T_{RightEnv} + T_{XNOR\uparrow} \\ B_0 &= T_{L6_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_0\downarrow} \\ C_0 &= T_{L3_{G\rightarrow Q}} + T_{LeftEnv_0} + T_{XNOR\uparrow} + T_{NOR\uparrow} \\ D_0 &= T_{ME_0\uparrow} \\ E_0 &= T_{L1_{G\rightarrow Q}} \\ F &= T_{XOR} \end{aligned}$$

2. *TPP single port*: The cycle is measured as the amount of time between one rising edge of the ME_0 output and the next, shown in Figure 3.13(b). The equation describes one full cycle on port 0.

$$\begin{aligned} \mathbf{T}_{TPP\_Single} &= max(A,\, E_0 + max(B_0,\, C_0) + G_0 + D_0) \qquad (3.8) \\ A &= T_{L5_{D\rightarrow Q}} + T_{RightEnv} + T_{XNOR\uparrow} \end{aligned}$$

$$
\begin{aligned}
B_0 &= T_{L6_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_0\downarrow} \\
C_0 &= T_{LeftEnv_0} \\
D_0 &= T_{ME_0\uparrow} \\
E_0 &= T_{L3_{G\rightarrow Q}} \\
F &= T_{XOR} \\
G_0 &= T_{L1_{D\rightarrow Q}} + T_{XNOR\uparrow} + T_{NOR\uparrow}
\end{aligned}
$$

3. *LP Alternating Ports*: A full cycle when alternating is the amount of time between one rising edge of the ME_0 output and the next, shown in Figure 3.13(c). The subscripts indicate the port associated with the path. The figure shows a full cycle, with one transaction on port 0 and the next on port 1. The cycle time for one flit at steady state, therefore, is half of the full cycle time.

$$
\begin{aligned}
\mathbf{T}_{LP\_Alternating} &= \frac{1}{2} \cdot max \left(\begin{array}{l} max(E_0 + B_0 + D_0,\, C_1) \\ \qquad + max(E_1 + B_1 + D_1,\, C_0), \\ max(A,\, B_0 + D_1 + E_1) \\ \qquad + max(A,\, B_1 + D_0 + E_0) \end{array} \right) \qquad (3.9) \\
A &= T_{L5_{D\rightarrow Q}} + T_{RightEnv} + T_{XNOR\uparrow} \\
B_0 &= T_{L6_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_0\downarrow} \\
B_1 &= T_{L7_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_1\downarrow} \\
C_0 &= T_{L3_{G\rightarrow Q}} + T_{LeftEnv_0} + T_{XNOR\uparrow} + T_{NOR\uparrow} \\
C_1 &= T_{L4_{G\rightarrow Q}} + T_{LeftEnv_1} + T_{XNOR\uparrow} + T_{NOR\uparrow} \\
D_0 &= T_{ME_0\uparrow}
\end{aligned}
$$

$$
\begin{aligned}
D_1 &= T_{ME_1\uparrow} \\
E_0 &= T_{L1_{G\rightarrow Q}} \\
E_1 &= T_{L2_{G\rightarrow Q}} \\
F &= T_{XOR}
\end{aligned}
$$

4. *TPP Alternating Ports*: A full cycle when alternating ports is the amount of time between one rising edge of the ME_0 output and the next, shown in Figure 3.13(d). The subscripts indicate the port associated with the path. The figure shows a full cycle, with one transaction on port 0 and the next on port 1. The cycle time for one flit at steady state, therefore, is half of the full cycle time.

$$
\begin{aligned}
\mathbf{T}_{TPP\_Alternating} &= \frac{1}{2} \cdot max \begin{pmatrix} S_A + S_B, \\ max(E_0 + S_A + B_1,\, S_C) + D_0, \\ max(E_1 + S_B + B_0,\, S_D) + D_1 \end{pmatrix} \quad (3.10) \\
A &= T_{L5_{D\rightarrow Q}} + T_{RightEnv} + T_{XNOR\uparrow} \\
B_0 &= T_{L6_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_0\downarrow} \\
B_1 &= T_{L7_{D\rightarrow Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} + T_{ME_1\downarrow} \\
C_0 &= T_{L3_{G\rightarrow Q}} + T_{LeftEnv_0} \\
C_1 &= T_{L4_{G\rightarrow Q}} + T_{LeftEnv_1} \\
D_0 &= T_{ME_0\uparrow} \\
D_1 &= T_{ME_1\uparrow} \\
E_0 &= T_{L3_{G\rightarrow Q}} + T_{XOR} \\
E_1 &= T_{L4_{G\rightarrow Q}} + T_{XOR}
\end{aligned}
$$

$$
\begin{aligned}
F_0 &= T_{L1_{D\rightarrow Q}} + T_{XNOR\uparrow} + T_{NOR\uparrow} \\
F_1 &= T_{L2_{D\rightarrow Q}} + T_{XNOR\uparrow} + T_{NOR\uparrow} \\
S_A &= max(A,\ B_0 + D_1 + E_1) \\
S_B &= max(A,\ B_1 + D_0 + E_0) \\
S_C &= E_0 + C_0 + G_0 \\
S_D &= E_1 + C_1 + G_1
\end{aligned}
$$

Several conclusions can be drawn based on the diagrams in Figure 3.13 and the equations 3.7 through 3.10. First, given a very slow right environment, all the cycle times evaluate to A. In the case of a slow left environment, which equates to light traffic in the network, the reverse paths, C, will dominate the cycle time. Both of these behaviors are expected in an handshaking pipeline, where each stage is dependent on receiving requests from the left and acknowledgments from the right.

Another interesting case is a very fast right environment and a very fast left environment. In this case, LP_Single operates very serially, evaluating to $B + C + D + E$. The cycle consists of acquiring the mutex, resetting it, then waiting for a new request from the left. The TPP_Single case, which has some concurrency between internal and reverse paths, will evaluate to $G + D + E + B$, cycling between accepting new data, acquiring the mutex, and resetting it. As expected, TPP will have better throughput than LP in this situation, benefiting from the early acknowledgment to the left.

The most interesting result in the analyzing the fast environments case is that TPP and LP both perform the same for alternating inputs, each evaluating

to $B + D + E$ for both ports. In nodes close to the root of the fan-in tree, where contention is more likely, this behavior can be exhibited, making both TPP and LP good candidates for the root primitive.

It is important to note that even under heavy load and a fast right environment, the single-port performance may be relevant, such as in the case of multi-flit packets. Given a very fast right environment, TPP will outperform LP in handling multi-flit packets, since each will operate according to their respective single-port equations while in multi-flit mode. But depending on the frequency of multi-flit packets in the traffic, this may not be the common case.

Timing Constraints: There are four timing constraints that must be satisfied for the correct operation of the primitive.

1. *Input Bundling Constraint*: There is a bundling constraint on the data input channels. The bundled data communication protocol, discussed in section 2.4, specifies that new, stable data should arrive at the input before the request transition. The request transition (*Req0* or *Req1*) then asserts validity for the data, as well as begins the transfer of data from one stage to the next.

2. *Fast Reset of Mutex*: There is a race condition between the setting of multi-flit mode and the resetting of the mutex for the next cycle. This mostly applies to the case where the right bank of data latches is transparent when the mutex asserts a new decision. Once the mutex raises an acknowledgment, in order to guarantee correctness, the multi-flit "kill" signal must be asserted or deasserted before the mutex can be reset. This ends up being a simple

constraint to satisfy:

$$T_{Latch_{G\to Q}} + T_{Latch_{D\to Q}} + T_{XNOR\downarrow} + T_{NOR\downarrow} > T_{Latch_{G\to Q}} + T_{NOR} \quad (3.11)$$

$$T_{Latch_{D\to Q}} + T_{XNOR\downarrow} > 0$$

This constraint applies to both LP and TPP primitives. For LP, this must hold for (L1, L6, L8) or (L2, L7, L9) used as the latches in the first equation. For TPP, substitute (L3, L6, L8) or (L4, L7, L9) for the latch terms in the first equation. This constraint should be met even with the addition of OR gates at the outputs of L8 and L9, as in Figures 3.10 or 3.12.

3. *Output Bundling Constraint*: Another timing constraint is a bundling constraint (Section 2.4) on the right bank of data latches, when they are transparent prior to a mutex decision. After a decision is made by the mutex, the winning *Req* continues through L5, where it is fed back to the MOUSETRAP XNOR latch control in order to close the right bank of latches. During this time, the correct data must be passed through the muxes and data latches, so the correct values are present when the latches are disabled, storing the data.

$$T_{Latch_{G\to Q}} + T_{XOR} + T_{L5_{D\to Q}} + T_{XNOR\downarrow} > T_{SRlatch} + T_{MUX} + T_{LData_{D\to Q}} \quad (3.12)$$

$$T_{Latch_{G\to Q}} + T_{XOR} + T_{XNOR\downarrow} > T_{SRlatch} + T_{mux}$$

This constraint should also be easily satisfied, and not require added delay. Although this should be verified in a post-layout design. The first latch variable may be (L1, L2) for LP or (L3, L4) for TPP.

4. *Data Overrun in TPP Latches*: The final constraint appears only in TPP and

concerns the L1 and L3 (or L2 and L4) latch enables. Since they are controlled by the same latch enable, the mutex output, there is a chance for data overrun when L1 is being opened and L3 is being closed.

$$T_{L1_{G\rightarrow Q}} > T_{hold_{L3}} \tag{3.13}$$

As long as the paths from mutex output are relatively matched, this should be easy to satisfy. The gate-to-output delay through the latch usually will exceed the hold time.

3.4 Pipeline Primitive

This section presents the pipeline primitive, the final element of the network. For this design, the existing MOUSETRAP high-speed pipeline design presented by Singh and Nowick in [31] is used. The structure and basic operation is described below for completeness. More detailed specification can be found in [31].

Block Diagram: The pipeline primitive accepts data from the left and forwards it to the right. MOUSETRAP stages can also be added for boosting performance, since they provide a fast acknowledgment to the left after accepting new data, and can offset inter-primitive latencies by providing buffering on long wires in the network.

Both interfaces of the pipeline primitive are transition signaling with bundled data, similar to the routing and arbitration primitives, shown in Figure 3.14. In fact, the routing and arbitration primitives were designed as MOUSETRAP-style stages for performance and exhibit similar latch controls as pipeline primitive.

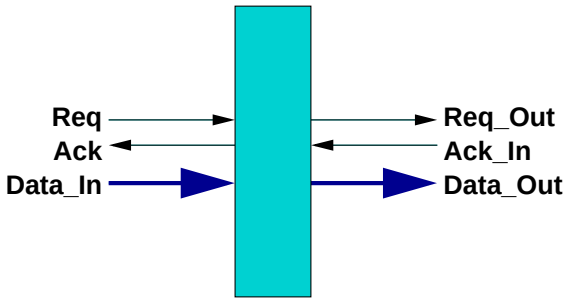


Figure 3.14: Block Diagram of the Pipeline Primitive

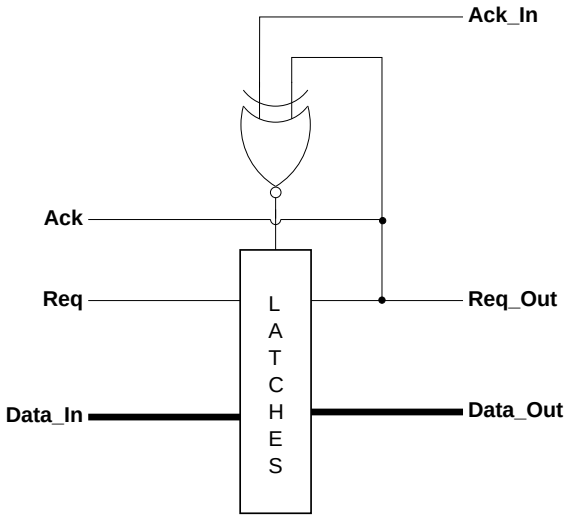


Figure 3.15: Pipeline Primitive

Architecture of the Pipeline Primitive: The MOUSETRAP pipeline stage consists of a set of transparent D-latches and a latch controller, which is comprised of a single standard logic gate, an XNOR.

Basic Operation: Initially, all inputs are low, and the XNOR latch controller is enabling the transparent latches. First, *Data* appears at the input of the latches and is allowed to pass through to the following stage. Sometime after the data is stable, a *Req* transition appears at the input and also passes through a transparent latch.

After the *Req* appears at the output, three events occur in parallel:

1. A *Req_Out* is generated to the right environment, signaling that new valid data is available.

2. An *Ack* is generated to the left environment, freeing it up to process a new data itme.

3. The latch controller in the current stage quickly disables the latches (making them opaque) to protect and store the current data.

The latches will remain opaque, storing the data, until an *Ack_In* is received from the right environment. At this point, the latches are made transparent and new data may arrive at the current stage. Detailed description of performance equations and timing constraints, as well as optimizations, can be found in [31].

Chapter 4

Mixed-Timing Network

This chapter presents the mixed-timing Mesh-of-Trees network which consists of the asynchronous network with mixed-timing interfaces at the source and destination ports. The mixed-timing network provides communication between synchronous domains and the asynchronous network on a single chip. The mixed-timing interfaces allow the network to communicate with multiple synchronous domains, providing flexibility in designing large, distributed systems on a chip by removing the need for global clock distribution. Such implementations are referred to as globally-asynchronous, locally-synchronous (GALS) systems [11], since multiple synchronous domains communicate asynchronously.

The goal of the mixed-timing network is to incorporate mixed-timing interfaces, while maintaining the same high throughput and low latency properties of the original asynchronous network. The mixed-timing interfaces should follow the same design methodology as the asynchronous network primitives, with an emphasis on standard cell implementations and use of commercial CAD tools. Additionally, the mixed-timing network should be modular and not require modification to existing synchronous environments. As a result, the mixed-timing network should be compatible with the synchronous interfaces of the XMT processing clusters and memory modules, allowing for easy integration.

To achieve these goals, mixed-timing FIFOs proposed by Chelcea and Nowick [12] are implemented. The mixed-timing FIFOs provide robust, low-latency communication between different timing domains, whether they be two domains operating under arbitrary (i.e. unrelated) clocks, two asynchronous domains, or a combination of the two. This family of mixed-timing FIFOs is designed in a modular fashion, and most importantly, do not require modifications to the sender or the receiver domain. Each FIFO is constructed with a reusable *put* and *get* component, either synchronous or asynchronous. For this research, mixed-timing FIFOs with three storage cells are used. Based on the results from [12], a 3-place FIFO will keep area overheads low and avoid frequent stalling from early 'empty' detection that a 2-place FIFO may encounter. In addition, a timing violation was discovered in the synchronous-asynchronous FIFO cell, and a new modification that is fixes the violation is proposed.

The chapter is organized as follows. First, the basic architecture and external interfaces of the mixed-timing FIFOs are discussed. Then the implementation of two FIFOs used with the network, the synchronous-asynchronous and the asynchronous-synchronous, are described in detail. The implementations follow the same standard cell approach as the rest of the network wherever possible. The asynchronous component of each of the two FIFOs uses a four-phase, return-to-zero handshaking protocol, while the network uses transition signaling. Therefore, we specify, design, and implement new *protocol converters* for each of the FIFOs to efficiently convert between two-phase and four-phase asynchronous signaling. These designs are shown in sections 4.2.1 and 4.3.1.

4.1 Overview

The primitives described in the previous chapter can be arranged to form a fully asynchronous Mesh-of-Trees network. However, in many applications, such as shared-memory parallel architectures, the drivers of the network will be synchronous. In order to provide communication between synchronous senders and receivers via the asynchronous network, we implement mixed-timing FIFOs proposed by Chelcea and Nowick in [12].

Figure 4.1 shows the structure of the mixed-timing network and integration with synchronous domains. Synchronous domains communicate directly with synchronous interfaces of mixed-timing FIFOs. The synchronous-asynchronous FIFOs are denoted by $S \rightarrow A$ and the asynchronous-synchronous FIFOs are denoted by $A \rightarrow S$. Each mixed-timing FIFO communicates to the asynchronous network via the two-phase to four-phase protocol converters, denoted by PC.

Basic Architecture of Mixed-Timing FIFOs: This section describes the structure of the mixed-timing FIFOs used in the mixed-timing Mesh-of-Trees interconnection network. Each mixed-timing FIFO is a circular array, or token ring, of identical storage cells that communicate with the put and get components on a shared bus. In addition, adjacent cells pass *put* and *get* tokens, which denote which cell is next for enqueuing or dequeuing data. Figure 4.2 shows the arrangement of two mixed-timing FIFOs, the synchronous-synchronous and the asynchronous-asynchronous as presented in [12]. The *put* and *get* components of each can be freely combined form the other two mixed-timing FIFOs which interface between

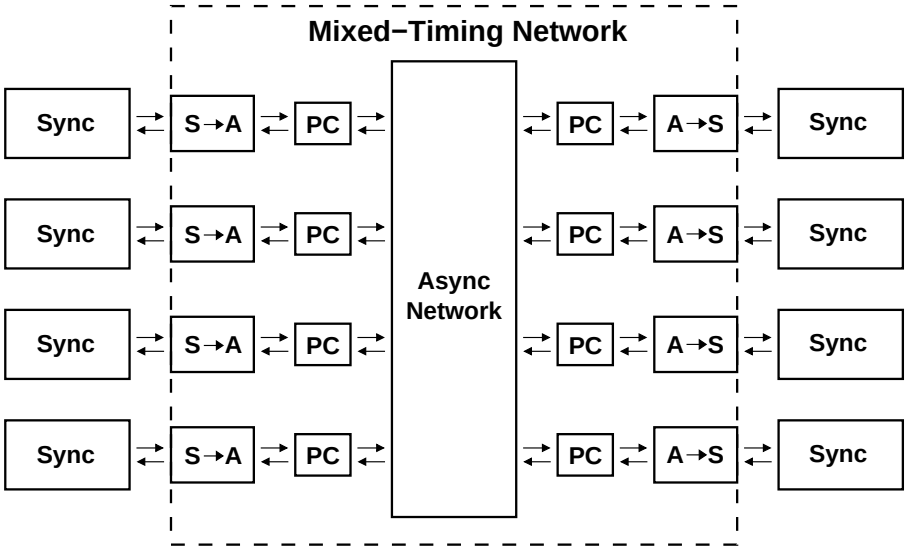
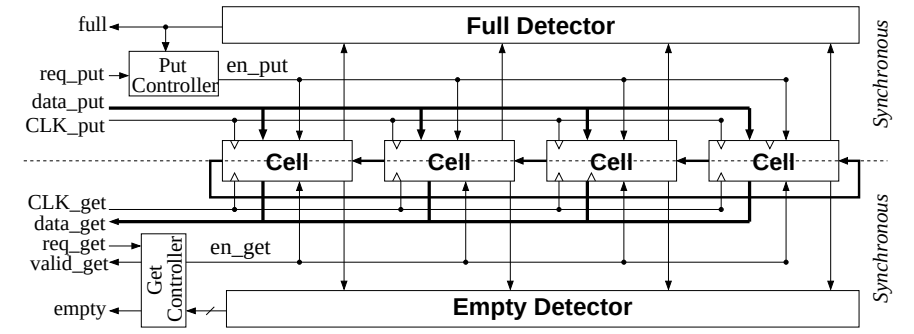


Figure 4.1: Mixed-Timing Network: Asynchronous Network with Mixed-Timing Interfaces and New Protocol Converters

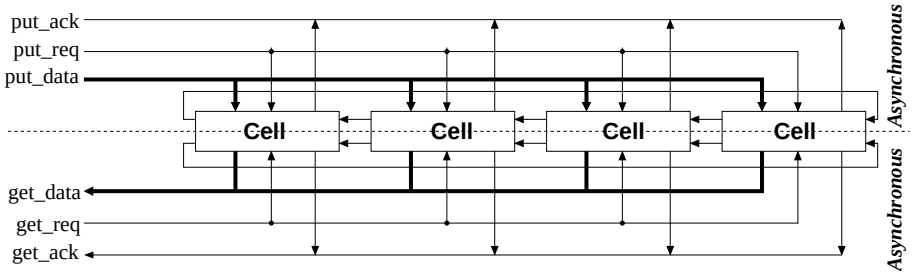
asynchronous and synchronous timing domains.

An important feature of the mixed-timing FIFOs is that data does not move. Once data is enqueued it remains in place, while the *put* and *get* tokens are passed between the cells. Since data is not passed between cells, the FIFOs have a potention for low latency and low-power operation. In addition, the modular design allows for high scalability with very few design modifications.

The synchronous put interface (Fig. 4.2(a)) is clocked by CLK_{put}. The signals req_{put} and data_{put} are the request and data inputs respectively. The *full* signal is asserted only when the FIFO is full, signalling to the synchronous interface that data cannot be inserted at that cycle. The synchronous get interface (Fig. 4.2(a)) is clocked by CLK_{get}. The signal req_{get} is the request input to the FIFO. The data_{get} signal is data output to the synchronous timing domain. The *empty* signal is asserted only when the FIFO is empty, signaling to the synchronous interface that valid data



(a)



(b)

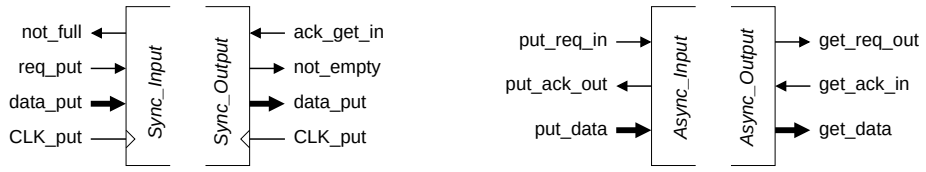
Figure 4.2: Architectures of two mixed-timing FIFOs: (a) Synchronous-Synchronous FIFO and (b) Asynchronous-Asynchronous FIFO [12]

is not available at that cycle. The valid$_{get}$ signal is not used for the mixed-timing interfaces implemented in this research.

The asynchronous interfaces (Fig. 4.2(b)) are different than the synchronous in that they are not synchronized to a clock input. The asynchronous put interface has put$_{req}$ and put$_{data}$ inputs that serve as the request and data inputs to the FIFO. Instead of a *full* output, the interface simply *withholds* the put$_{ack}$ signal until the FIFO can accept new data. The asynchronous get interface has a get$_{req}$ input that is the request for new data. The get$_{data}$ signal is the data output to the synchronous environment. Instead of an *empty* output, the interface simply withholds the get$_{ack}$ signal until the FIFO can accept new data.

For the network implementation, which is a unidirectional pipelined mesh, it is advantageous to have the convention that *req* and *data* signals have the same orientation (both inputs for *put* or outputs for *get*. This changes the functionality of the sync_get and async_get interfaces to sync_output and async_output respectively. For sync_output, the negated *empty* signal (*not_empty*) is used as a request output and the req$_{get}$ input is renamed to ack$_{get_in}$ from the synchronous environment.

For async_output, the get$_{req}$ and get$_{ack}$ should be reversed, with the caveat that an initial request must be generated at initialization. This way, get$_{ack}$ becomes a get$_{req_out}$ and get$_{req}$ becomes get$_{ack_in}$ from the asynchronous environment. In addition, the negated *full* signal (*not_full*) is used as a synchronous acknowledgment output. The new interface namings are shown in Figure 4.3 and are reflected in Figures 4.4 and 4.7 from [12].



(a) (b)

Figure 4.3: Synchronous and Asynchronous FIFO interfaces for asynchronous Mesh-of-Trees: (a) Synchronous FIFO interfaces and (b) Asynchronous FIFO interfaces

4.2 Synchronous-Asynchronous FIFO Implementation

The synchronous-asynchronous FIFO takes interfaces a synchronous sender and asynchronous receiver. This section describes how the synchronous-asynchronous FIFO was implemented for this research. It was determined that a timing violation may occur under certain scenarios, and a new simple fix is proposed to the design in [12]. Detailed description of the architecture and operation of the FIFO cells and interface as a whole can be found in [12].

A synchronous-asynchronous mixed-timing FIFO cell combines the synchronous put and asynchronous get components from [12], as shown in Figure 4.4. For the purposes of the network, where we would like *req* and *data* signals to have the same orientation, signals *get_req* and *get_ack* in 4.4 will be used to generate *get_ack_in* and *get_req_out* respectively. This does not represent a change in the individual cell, but in the FIFO as a whole.

Simulations with the synchronous-asynchronous FIFO cell revealed a potential hazardous scenario that may occur in the design presented in [12]. In certain cases, invalid or stale data could be delivered to the asynchronous environment. A simple

solution was identified and the proposed solution were confirmed by Chelcea and Nowick, and Figure 4.4 shows the modified FIFO cell. In particular, the clocking signal for REG was modified from CLK_PUT to *we*.

The consequence of modifying the clocking signal is a bundling constraint that the rising edge of *get_req_out* must occur only after there is valid data is present on *get_data*. This constraint should is easily satisfied in the current implementation.

$$we+ \rightarrow get\_req\_out+ > we+ \rightarrow get\_data$$

As shown in Figure 4.4, the synchronous-asynchronous FIFO consists of three main components: synchronous put, asynchronous get, and data validity controller (DVsa). The synchronous put component is implemented for this research as a edge-triggered D flip-flop clocked by CLK_PUT with *en_put* as an enable signal, an AND gate, and a bank of edge-triggered D flip-flops clocked by *we*. For initialization, the last cell (tail of the circular array of cells) should assert *ptok_out* so the first cell (head of the circular array of cells) is initialized with the put token.

The asynchronous get component consists of an asymmetric C-element with two 'plus' inputs, an obtain-get-token (OGT) circuit, and a bank of tri-state buffers at the outputs of the data registers from the synchronous put component. The asymmetric C-element, for the purposes of this research, was implemented with standard cell combinational gates, with logic function shown in Figure 4.5(a). This choice was made to facilitate simulating the mixed-timing FIFO in Verilog using ARM standard cell timing models for the IBM 90nm process.

The OGT circuit is a burst-mode asynchronous machine that controls the

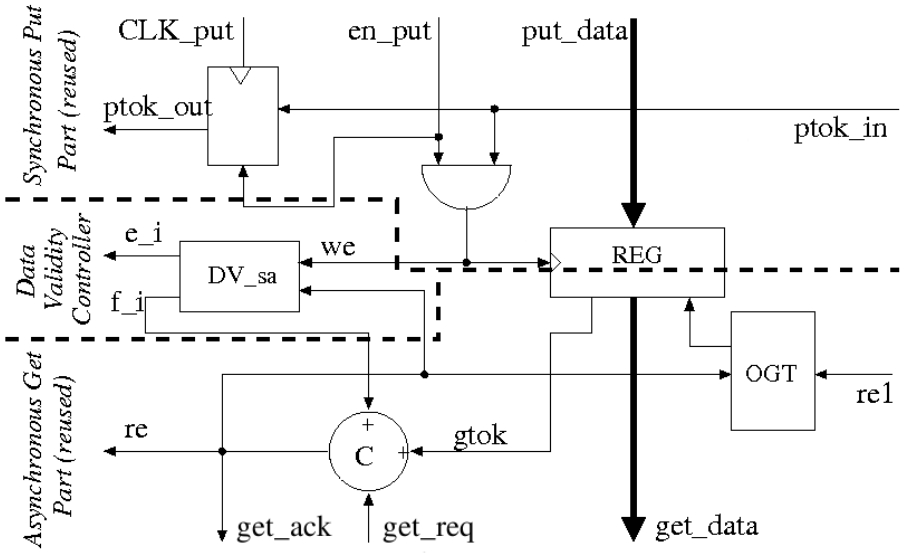
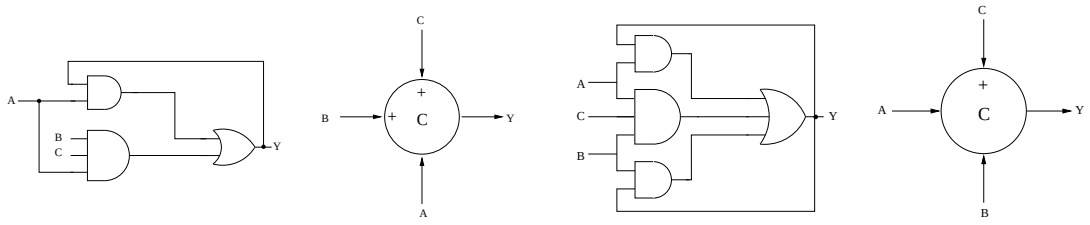


Figure 4.4: Synchronous-asynchronous FIFO cell with new modified data register clocking [12]

movement of the get token between FIFO cells. The burst-mode specifications are synthesized using the MINIMALIST [17] CAD tool as part of the CaSCADE asynchronous design environment [28]. During initialization, the first cell (head of the circular array of cells) should be given the get token by incorporating reset into the OGT implementation presented in [12]. When the get token is present in a cell (*gtok* is asserted), the bank of tri-state buffers in REG are enabled, allowing the data stored in the registers to be output onto the shared *get_data* bus. Only one cell can have *gtok* asserted at any given time.

The data validity controller (DVsa) for the synchronous-asynchronous cell in [12] was designed using the Petrify CAD tool [14]. For this research, the DVsa was modeled by a behavioral Verilog description that implemented the signal transition



(a) (b)

Figure 4.5: Gate-level Asymmetric C-element Implementations: (a) with two plus inputs (b) with one plus input

graph (STG) specification described in [12].

4.2.1 Two-Phase to Four-Phase Protocol Converter

The asynchronous get interface of the synchronous-asynchronous mixed-timing FIFO uses a four-phase, return-to-zero handshaking protocol. The asynchronous Mesh-of-Trees network, however, uses transition signaling, a two-phase handshaking protocol. In order to interface between the two components, a custom low-latency, asynchronous protocol converter is designed.

The two-phase to four-phase converter is synthesized as a burst-mode asynchronous machine using the MINIMALIST [17] CAD tool, part of the CaSCADE asynchronous design environment [28]. The burst-mode specification is a Mealy-type finite-state machine consisting of a set of states and arcs. A complete set of input transitions (called the input burst) will yield a complete set of output transitions (called the output burst) and a state transition.

The protocol converter is placed in between the synchronous-asynchronous FIFO and the root of a fan-out tree, an input to the network. In Figure 4.6(a), the

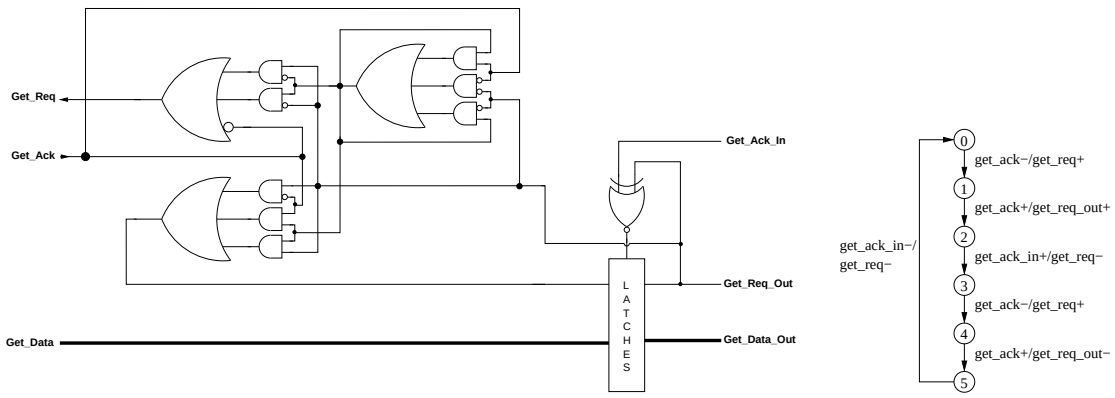
mixed-timing FIFO is placed to the left and the fan-out root is placed to the right.

For correct operation, the protocol converter is initialized to state S0 (Fig. 4.6(b)) during reset, where reset is asserted and all *req* and *ack* inputs are deasserted. When reset is deasserted, the burst-mode machine will have the necessary input burst (*get_ack*-) to cause the rising edge of *get_req*, a request to get new data from the FIFO, and make a transition to state S1. When the first request is acknowledged, it will cause the *get_req_out*+ output burst to the fan-out root, signaling that valid data is available. In this way, the initial request to the mixed-timing FIFO "primes the pump" for future transactions.

To maintain good performance, a MOUSETRAP pipeline stage [31] is added at the output to the right environment. The MOUSETRAP stage, when empty, will quickly store the data and acknowledge the mixed-timing FIFO (left interface), allowing the subsequent get request to the asynchronous get component. This shortens the delay for arcs S1→S2 and S4→S5, improving throughput in many situations, since the fan-out root will take longer to acknowledge than the MOUSETRAP pipeline stage.

4.3 Asynchronous-Synchronous FIFO Implementation

The asynchronous-synchronous FIFO takes interfaces an asynchronous sender and synchronous receiver. This section describes how the asynchronous-synchronous FIFO was implemented for this research. Detailed description of the architecture and operation of the FIFO cells and interface as a whole can be found in [12].



(a) (b)

Figure 4.6: (a) Protocol Converter and (b) Burst-Mode Specification for the Synchronous-Asynchronous Mixed-Timing FIFO

An asynchronous-synchronous mixed-timing FIFO cell combines the asynchronous put and synchronous get components from [12], as shown in Figure 4.7. The asynchronous-synchronous FIFO consists of three main components: asynchronous put, synchronous get, and data validity controller (DVas). The asynchronous put component consists of an asymmetric C-element with one 'plus' input, a bank of transparent D latches (part of REG), and an obtain-put-token (OPT) circuit. The asymmetric C-element, for the purposes of this research, was implemented with standard cell combinational gates, with logic function shown in Figure 4.5(b). This choice was made to facilitate simulating the mixed-timing FIFO in Verilog using ARM standard cell timing models for the IBM 90nm process.

The latches in REG are enabled when *we* is asserted, allowing *put_data* to pass through. When *we* is deasserted, the data is safely stored in the latches. The OPT circuit is a burst-mode asynchronous machine [17] that controls the movement of the put token between FIFO cells. During initialization, the first cell (head of the

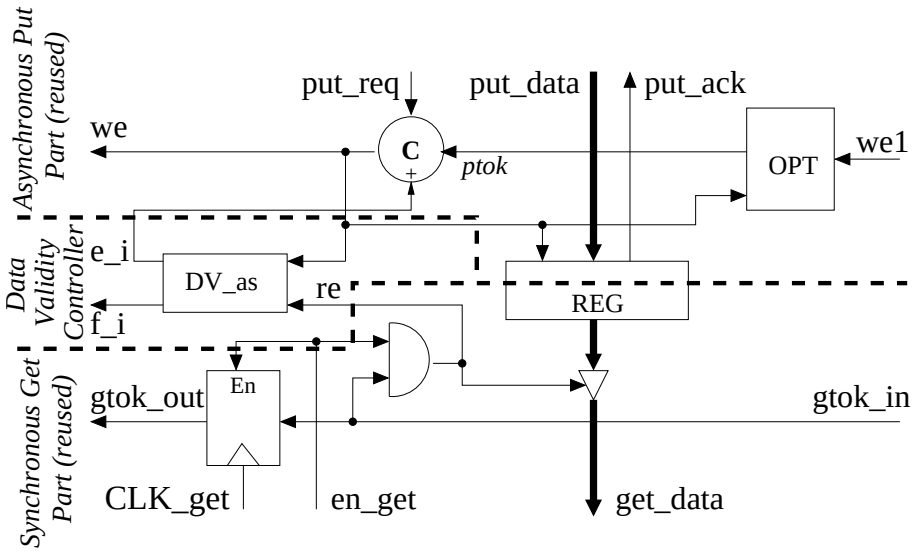


Figure 4.7: Asynchronous-synchronous FIFO cell [12]

circular array of cells) should be given the put token by incorporating reset into the OPT implementation presented in [12].

The synchronous get component consists of an edge-triggered D flip-flop clocked by CLK_get and enabled by *en_get*, an AND gate, and a bank of tri-state buffers. For initialization, the last cell (tail of the circular array of cells) should assert *gtok_out* so the first cell (head of the circular array of cells) is initialized with the get token.

The data validity controller (DVas) for the asynchronous-synchronous cell in [12] was designed using the Petrify CAD tool [14]. For this research, the DVas was implemented using a behavioral Verilog description that implemented the signal transition graph (STG) specification described in [12].

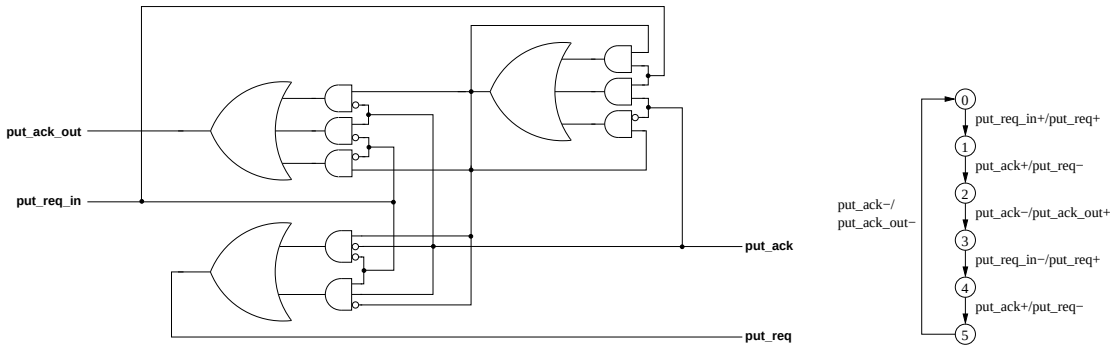
4.3.1 Two-Phase to Four-Phase Protocol Converter

The asynchronous put interface of the asynchronous-synchronous mixed-timing FIFO uses a four-phase, return-to-zero handshaking protocol. The asynchronous Mesh-of-Trees network, however, uses transition signaling, a two-phase handshaking protocol. In order to interface between the two components, a custom low latency, asynchronous protocol converter is designed.

The two-phase to four-phase converter is synthesized as a burst-mode asynchronous machine using the MINIMALIST [17] CAD tool. The burst-mode specification is a Mealy-type finite-state machine consisting of a set of states and arcs. A complete set of input transitions (called the input burst) will yield a complete set of output transitions (called the output burst) and a state transition.

The protocol converter is placed in between the root of the fan-in tree, an output of the network, and the asynchronous-synchronous FIFO interface. In Figure 4.8(a), the fan-in root is placed to left and the mixed-timing FIFO is placed to the right.

For correct operation, the protocol converter is initialized to state S0 (Fig. 4.8(b)) during reset, where reset is asserted and all *req* and *ack* inputs are deasserted. The first input burst will be a rising edge on the *put_req_in* signal, indicating there is valid data to place into the FIFO. Then, a full four-phase handshake will take place on the FIFO interface, and only after *put_ack* is deasserted, indicating the data is safely stored, will *put_ack_out* transition, completing the put operation.



(a) (b)

Figure 4.8: (a) Protocol Converter and (b) Burst-Mode Specification for the Asynchronous-Synchronous Mixed-Timing FIFO

Chapter 5

Experimental Results

This chapter presents measurements of performance, area, and power for the asynchronous network and its components. First, the asynchronous primitives are evaluated and compared to synchronous primitives from [4, 6, 5]. Then, simulation results for the asynchronous network and mixed-timing network are reported. Finally, the mixed-timing network is embedded in the XMT processor and evaluated using four benchmarks.

5.1 Overview

Simulation results are reported in increasing order of complexity. First, the asynchronous primitives are evaluated in isolation for latency, throughput, area, and power. Results for these key metrics are compared to results obtained from the synchronous primitives developed in [4, 6, 5]. Significant area and power savings are reported for the asynchronous, with latency and throughput comparable between the asynchronous and synchronous designs.

Next, small groups of routing and arbitration primitives, arranged as binary trees, are simulated. The simulations with binary trees give a more accurate indication of behavior in the full network. Results for maximum throughput under different input traffic conditions are reported. Maximum throughput remains the

same between isolated primitives and 3-level trees, indicating that performance of the tree is determined by the root. Successive *single port* routing and arbitration represent the worst-case steady-state operation in isolation for both primitives, and throughput for single port is further reduced in the 3-level trees.

Mixed-timing interfaces with *protocol converters* are evaluated in isolation for latency and maximum steady-state throughput, as they are in [12]. The simulation results presented in [12] are for 0.6-μm HP CMOS technology, while the current research uses IBM 90nm technology. Results appear consistent with those reported in [12], with an expected improvement in performance shifting to 90nm technology.

A full 8-terminal asynchronous network is simulated using a projected layout, as current commercial CAD tools posed limitations for creating a placed and routed design. To create the projected network layout, wire delays were added between primitives in an 8-terminal network to approximate the floorplan used for the synchronous Mesh-of-Trees network in [5]. The projected network layout is evaluated for latency and throughput at varying input traffic rates.

The mixed-timing network is evaluated by simulating the projected network layout, mixed-timing interfaces, and protocol converters. Maximum throughput of the mixed-timing network is reported at varying clock frequencies to evaluate possible bottlenecks for performance created by mixed-timing interfaces.

Finally, the mixed-timing network is embedded into the XMT processor model. Execution times for several benchmarks are measured and speedup versus the XMT processor with synchronous network is reported.

5.2 Tool Flow

One of the goals of this work was to design and simulate the asynchronous designs using available commercial CAD tools. Each of the circuits was designed using a standard cell methodology whenever possible. ARM 90nm (CMOS9SF) SAGE-X^{TM} standard cells were used for primitives and mixed-timing interfaces.

Designs were placed, routed and extracted using the Cadence Encounter® digital IC platform. Post-layout simulations were performed on gate-level netlists by Cadence NC-Verilog with Standard Delay Format (SDF) Annotation at the typical process corner (1.2V, 25°C). Power measurements were obtained using Cadence Voltagestorm, as part of the Encounter® platform. Burst-mode controllers for the mixed-timing FIFO protocol converters were synthesized using the MINIMALIST [17] CAD tool.

The mutual exclusion (ME) element of the arbitration primitive cannot be implemented using standard combinational gates. In order to incorporate the ME into the gate-level Verilog simulations, the cell was designed using transistor models from the IBM 90nm (CMOS9SF) PDK, and simulated in Cadence Spectre at 1.2V and 25°C. Delay information was used to create a behavioral Verilog model with similar timing behavior, described in detail in section 5.3.2.

The XMT processor Verilog model was developed by Xinghzi Wen [36] and used to simulate the XMT processor with the mixed-timing network. The mixed-timing network was designed to match the synchronous interfaces of processing clusters and memory modules in XMT.

5.3 Asynchronous Primitives

This section presents performance, area, and power measurements for the asynchronous primitives used in the network. Performance is analyzed based on two key metrics: latency and throughput. Latency is the delay from a request transition on the input interface to its appearance at the output in an empty primitive. Throughput is analyzed for various input conditions and is given as a fraction of the bandwidth, the maximum rate of data transfer in steady-state operation. Maximum throughput is measured for primitives in isolation as well as small groups of primitives.

Area measurements report the area occupied by all standard cells used in each design. Average power consumption is reported for asynchronous and synchronous primitives with various steady-state input traffic patterns.

5.3.1 Routing Primitive

The performance, area, and power consumption of the Mesh-of-Trees network are dependent on the individual network primitives. In this section, we evaluate the asynchronous routing primitive in isolation and compare to the synchronous routing primitive in terms of latency, throughput, area, and power consumption. Then, throughput in a 3-level binary *fan-out tree* of asynchronous routing primitives is reported.

Significant area and power savings are reported for the asynchronous. The asynchronous primitive uses 64% less area than the synchronous. At the steady-

state operating point with highest average power consumption (alternating routing destinations), the asynchronous consumed 85% less power on average than the synchronous.

The asynchronous routing primitive has competitive performance when compared with the synchronous primitive. Both asynchronous and synchronous primitives have similar latency. Maximum throughput of the asynchronous primitive is 12% less than synchronous at the maximum clock rate for the post-layout primitive.

Throughput is measured for routing primitives in a 3-level tree under various steady-state input traffic conditions, described below. The maximum throughput of the 3-level tree is the same as the routing primitive in isolation, indicating that fan-out tree performance is determined by the fan-out root primitive.

The routing primitive is comprised entirely of standard cells, as shown in Figure 3.2 and are placed, routed, and simulated according the tool flow described in section 5.2. All simulations in this section use an 8-bit datapath.

Routing Primitive in Isolation: The performance, area, and power consumption of the Mesh-of-Trees network are dependent on the individual network primitives. In this section, we examine the performance of an asynchronous routing primitive in isolation, which represents the upper bound for performance in a *fan-out tree* of the network. These results, as well as area and power measurements, are compared to the synchronous routing primitive. First, results for latency and throughput are presented. Then, area and power figures are reported.

Latency and Throughput: Table 5.1 shows latency and throughput measurements gathered from simulation of the routing primitive in isolation. Throughput

is measured for three steady-state input conditions. *Single* refers to consecutive packets routed to the same output port. *Alternating* refers to consecutive packets routed to different output ports. *Random* refers to consecutive packets with routing based on a uniformly random distribution. In the context of XMT, where addresses are hashed before being input to the network, *random* represents the common case.

For correctness in simulation, a 200ps margin was added between request and acknowledgment outputs and their subsequent, corresponding inputs. This margin is an optimistic value for time spent in the left or right environment.

Table 5.1 shows latency and throughput measurements for the asynchronous routing primitive in isolation. The latency is given in picoseconds, and is measured as the time between new request and data appearing at the inputs of an empty primitive and the same request and data appearing at the output. The maximum operating rate is given in giga-flits per second (Gfps), the number of flits that a routing primitive can advance per second. The bandwidth is computed by multiplying this value by the datapath width (bits per flit). In this case, the bandwidth is 13.6 Gbps for an 8-bit datapath. The relative throughput is the fraction of the bandwidth utilized under the listed input conditions.

Table 5.1: Latency and Throughput of Routing Primitive in Isolation

Type of Primitive	Latency (ps)	Max. Operating Rate (Gfps)	Relative Throughput		
			Single	Alternating	Random
Asynchronous	546	1.70	0.87	1.0	0.93
Synchronous	516	1.93	1.0	1.0	1.0

The throughput is calculated by multiplying the relative throughput by the

bandwidth. *Single* and *Alternating* define the worst and best throughput respectively. The uniformly random case, as expected, falls in the middle the two extremes, since each is equally likely to occur. The synchronous designs achieves the same throughput under all input conditions, advancing one flit per clock cycle.

Area and Power Consumption: Table 5.2 shows standard cell area and power consumption of the routing primitive with 8-bit datapath. The *single*, *alternating*, and *random* columns correspond to the input conditions in Table 5.1. The synchronous power measurements were made for a clock rate of 1.70 GHz.

Table 5.2: Area and Power Consumption of Routing Primitives

Type of Primitive	Cell Area (μm^2)	Average Power (mW)		
		Single	Alternating	Random
Asynchronous	358.44	0.410	0.502	0.455
Synchronous	988.55	3.065	3.404	3.317

Area: The asynchronous routing primitive has much lower area overhead than the synchronous, requiring 64% less cell area. Much of this area is saved on the datapath. While the asynchronous routing primitive requires two banks of transparent latches to store data, the synchronous uses two banks of edge-triggered flip-flops, according to the approach in [10], with levels of multiplexing and demultiplexing.

Power Consumption: Power consumption is reduced by 85% compared to the synchronous implementation in the *alternating case*, which has the highest average power consumption. This substantial reduction in power is attributed to lower overheads on the datapath for the asynchronous routing primitive.

Average power consumption is measured only for cases with successive routing

to specific ports. In these cases, the primitives remain active during the entire simulation. When the primitives are inactive, the asynchronous primitives do no consume dynamic power and are at a quiescent state. The synchronous primitives, however, will continue to spend dynamic power due to clock distribution. The measured clock power for the synchronous routing primitive was 0.381 mW, which is substantial compared to the total power consumed by an active asynchronous primitive.

Routing Primitives in a 3-level Fan-Out Tree: In the Mesh-of-Trees network, the routing primitives are arranged in a binary tree, called the *fan-out tree*. We examine the performance of a small group of routing primitives, arranged as a 3-level binary tree, as an indicator for performance in the full network. Results for throughput of the 3-level tree appear in Table 5.3.

Note that the 3-level tree has the same bandwidth as the root of the tree, the single routing primitive. However, the *single* case throughput has dropped considerably. The loss of performance is due to the $Req \rightarrow Ack$ latency in the second-level primitives of the tree. This path is longer than the 200ps margin used in the isolated primitive simulation and translates directly to a decline in throughput. The *random* case throughput is in the middle of the two extremes, which is expected since each case will occur half of the time in a uniformly random distribution.

Simulations are performed on post-layout asynchronous routing primitives. Simulations do not include inter-primitive wiring delays, and thus represent a lower bound for cycle time in a fan-out tree. Later, in section 5.5, simulation results for a proposed full network layout with inter-primitive delays are presented.

Table 5.3: Throughput of Routing Primitives in a 3-level Fan-Out Tree

Max. Operating Rate (Gfps)	Relative Throughput		
	Single	Alternating	Random
1.70	0.63	1.0	0.79

5.3.2 Arbitration Primitive

The previous section examined the routing portion of the Mesh-of-Trees network. In this section, simulation results for the arbitration portion are presented. We first evaluate both asynchronous arbitration primitives (LP and TPP) in isolation and compare to the synchronous arbitration primitive from [4, 6, 5]. Comparisons are made for the four key metrics of latency, throughput, area, and power consumption. Then, throughput is evaluated in a simple tree to understand system-level performance more accurately. In particular, a 3-level binary *fan-in tree* of asynchronous arbitration primitives is used in experiments.

The arbitration primitive is comprised of standard cells, with the exception of the mutual exclusion element. The LP and TPP arbitration primitives are placed, routed, and simulated according the tool flow described in section 5.2. All simulations in this section use an 8-bit datapath.

Significant area and power savings are reported for the asynchronous. The LP and TPP arbitration primitives use 84% and 74% less area than the synchronous primitive respectively. At the steady-state operating point with highest average power consumption, the asynchronous LP and TPP arbitration primitives consumed 91% and 87% less power on average than the synchronous.

Performance of the asynchronous primitives is competitive with the synchronous. The *latency-oriented primitive* (LP) has similar latency to the synchronous primitive. The *throughput-oriented primitive* (TPP) has worse latency than both the LP and synchronous designs, as expected from analytical performance equations reported in Chapter 3. Maximum throughput of both asynchronous primitives is competitive with the synchronous at the maximum clock rate for the post-layout primitive.

Throughput is measured for asynchronous arbitration primitives in a 3-level tree under various steady-state input traffic conditions, described below. The maximum throughput of the 3-level tree is the same as the arbitration primitive in isolation, indicating that fan-in tree performance is determined by the fan-in root primitive.

Simulating the Mutual Exclusion Element: The mutual exclusion element (mutex) is a key component of the asynchronous arbitration primitive. The mutex is an analog arbiter and is not part of the ARM 90nm standard cell library [2]. In order to evaluate the behavior of the mutex, a transistor model was designed using transistor models from the IBM 90nm (CMOS9SF) process, then simulated using the Cadence Spectre analog design environment.

The delay through the mutex varies depending on when input requests arrive. Two requests occuring close in time can send the arbiter into a metastable state that can persist for a long time. In order to incorporate this behavior, the delay information gathered from simulation was used to calibrate a behavioral Verilog description used in the gate-level simulations.

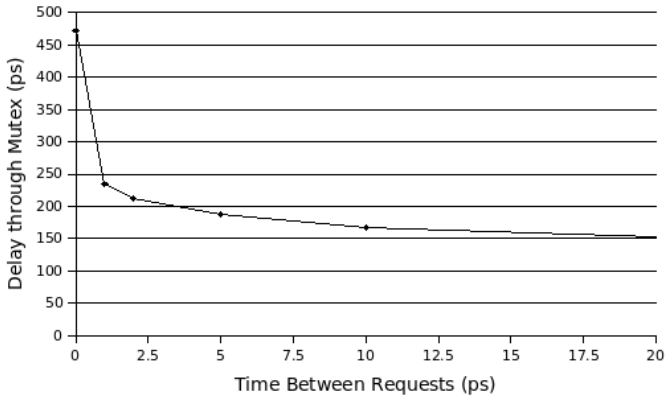


Figure 5.1: Plot of Mutex Delay Function

Since the transistor models and standard cell models have different simulated delays, even for the same PVT corner, the measured mutex delays were scaled by the ratio of fanout-of-4 (FO4) delays for the two timing models. This amounted to a derating factor of 1.66 applied to the measured mutex delays simulated by Spectre.

Arbitration Primitive in Isolation: In this section, we examine the performance of an asynchronous arbitration primitive in isolation. Results for latency and throughput, as well as area and power measurements, are compared to the synchronous arbitration primitive. First, results for latency and throughput are presented. Then, area and power figures are reported.

Latency and Throughput: Table 5.4 shows latency and throughput measurements gathered from simulation of the arbitration primitive in isolation. Throughput is measured for two input conditions. *Single* refers to consecutive packets arriving at the same input port. *Alternating* refers to packets arriving simultaneously on both ports, which will be served in alternating order. (Since both requests are allowed to reach the mutex, as soon as one transaction completes and the mutex is reset, the

opposing request will immediately be served.)

For the asynchronous primitives, as predicted by their performance equations given in section 3.3.3, LP has better latency than the TPP design. Both have the approximately the same throughput with in the *alternating* arbitration case, which is common near the root of the fan-in tree in a network with medium to heavy traffic.

However, in the case of consecutive packets arriving to a single port, LP has much worse throughput than TPP, operating at 56% of the maximum throughput. As shown in Figure 3.13, the operation of LP is very serial, and time spent in the left environment contributes directly to the single-port cycle time. TPP, on the other hand, generates an earlier acknowledgment, and this added concurrency between the left environment and the current cell allows for better throughput in the single-port case.

Table 5.4: Latency and Throughput of Arbitration Primitive in Isolation

Type of Primitive	Latency (ps)	Max. Operating Rate (Gfps)	Relative Throughput	
			Single	Alternating
Asynchronous - LP	489.5	2.04	0.56	1.0
Asynchronous - TPP	567.5	2.07	0.71	1.0
Synchronous	474	2.09	1.0	1.0

The maximum operating rate is given in gigaflits per second (Gfps), the number of flits the arbitration primitive can advance per second. The bandwidth is computed by multiplying this value by the datapath width (bits per flit). In this case, the bandwidth is 16.32 Gbps and 16.56 Gbps for LP and TPP respectively with

an 8-bit datapath. The relative throughput is the fraction of the bandwidth utilized under the listed input conditions. The throughput is calculated by multiplying the relative throughput by the bandwidth.

Table 5.5: Area and Power Consumption of Arbitration Primitives

Type of Primitive	Cell Area (μm^2)	Average Power (mW)	
		Single	Alternating
Asynchronous - LP	349.3	0.321	0.670
Asynchronous - TPP	584.2	0.607	0.933
Synchronous	2240.3	5.181	7.152

Area: Table 5.5 shows area and power consumption measurements for the asynchronous LP and TPP arbitration primitives. TPP uses three banks of latches while LP uses only one bank of latches. Each uses one bank of multiplexers. With larger datapath widths, the latches and muxes used for on the datapath will occupy nearly all of the total cell area of the primitive, so the ratio of total cell area for TPP versus LP will approach the 3-to-1 ratio of latch area.

The asynchronous primitives have an area savings of 84% and 74% for LP and TPP respectively. Much of this area is saved on the datapath. LP, for example, uses one bank of transparent latches and one bank of multiplexers to store and direct the flow of data. The synchronous primitive uses four banks of edge-triggered flip-flops and several levels of multiplexing and demultiplexing for the same purpose. As the datapath width increases, the area savings of the asynchronous primitives will also increase.

Power Consumption: Power consumption is reduced by 91% and 87% for

LP and TPP primitives respectively compared to the synchronous implementation in the *alternating case*, which has the highest average power consumption. This substantial reduction in power is attributed to lower overheads on the datapath for the asynchronous routing primitive, especially for the LP primitive. Measurements of average power consumption for the synchronous primitive were made for a clock frequency of 2.07 GHz.

Average power consumption is measured only for cases with successive routing to specific ports. In these cases, the primitives remain active during the entire simulation. When the primitives are inactive, the asynchronous primitives do no consume dynamic power and are at a quiescent state. The synchronous primitives, however, will continue to spend dynamic power due to clock distribution. The measured clock power for the synchronous routing primitive was 0.791 mW, which is substantial compared to the total power consumed by an active asynchronous primitive.

Arbitration Primitives in a 3-level Fan-In Tree:

In the Mesh-of-Trees network, the arbitration primitives are arranged in a binary tree, called the *fan-in tree*. We examine the performance of a small group of arbitration primitives, arranged as a 3-level binary tree, as an indicator for performance in the full network. Results for throughput of the 3-level tree appear in Table 5.6.

Table 5.6 shows throughput values for arbitration primitive performance in a 3-level fan-in tree. Similar to the fan-out tree results, the 3-level fan-in tree has bandwidth equal to the root of the tree, a single arbitration primitive. The 3-level

fan-in tree with TPP primitives has better throughput in the *single* case, which is expected based on the results in Table 5.4.

Table 5.6: Throughput of Arbitration Primitives in a 3-level Fan-In Tree

Type of Primitive	Max. Operating Rate (Gfps)	Relative Throughput	
		Single	Alternating
LP	2.04	0.53	1.0
TPP	2.07	0.64	1.0

Simulations do not include inter-primitive wiring delays, and thus represent a lower bound for cycle time in a fan-in tree. Later, in section 5.5, simulation results for a proposed network layout with inter-primitive delays are presented.

5.4 Mixed-Timing FIFOs

Mixed-timing FIFOs are added at the inputs and outputs of the network in order to interface the asynchronous network with multiple synchronous senders and receivers. The mixed-timing FIFO designs are based on the the work by Chelcea and Nowick [12]. Together, with the addition of new *protocol converters* described in Chapter 4, they form the mixed-timing Mesh-of-Trees network.

In this section, we evaluate latency and throughput of the mixed-timing FIFOs with protocol converters. From [12], latency of a mixed-timing FIFO is defined as the delay from the input of data on the put interface to its appearance at the output on the get interface in an empty FIFO. We define the maximum steady-state frequency as the clock rate where the synchronous put component will never be

full (for sync-async) or the synchronous get component will never be empty (for async-sync), given a fast asynchronous environment.

5.4.1 Synchronous-Asynchronous FIFO

The synchronous-asynchronous FIFO interfaces a synchronous sender and an input port of the asynchronous network. In the mixed-timing network, the synchronous-asynchronous FIFO has a new, low-latency *protocol converter* that interfaces the 4-phase handshaking of the FIFO and the 2-phase handshaking of the asynchronous network primitives. Performance is evaluated by examining latency and maximum steady-state frequency, the two metrics used for evaluation in [12].

Table 5.7 shows simulation results for the synchronous-asynchronous FIFO with protocol converter. The synchronous-asynchronous FIFO contains the synchronous put component, which interfaces with the synchronous environment, and the asynchronous get component, which interfaces with the *protocol converter* and root of a fan-out tree. Latency is reported in picoseconds and maximum steady-state frequency is given in megahertz (MHz).

Table 5.7: Performance of the Synchronous-Asynchronous FIFO with Protocol Converter

Latency (ps)	Maximum Steady-State Frequency (MHz)
967.3	932.8

5.4.2 Asynchronous-Synchronous FIFO

The asynchronous-synchronous FIFO interfaces an output port of the asynchronous network and a synchronous receiver. In the mixed-timing network, the asynchronous-synchronous FIFO has a new, low-latency *protocol converter* that interfaces the 2-phase handshaking of the asynchronous network primitive and the 4-phase handshaking of the FIFO. Performance is evaluated by examining latency and maximum steady-state frequency, the two metrics used for evaluation in [12].

Table 5.8 shows simulation results for the asynchronous-synchronous FIFO with protocol converter. The asynchronous-synchronous FIFO contains the asynchronous put component, which interfaces with root of the fan-in tree, and the synchronous get component, which interfaces with synchronous environment. Due to the function of the empty controller [12], it takes two full clock cycles after a put operation in an empty FIFO for *Empty* to be deasserted. Therefore, latency varies with the exact moment when data items are enqueued during the clock cycle, hence the *Min* and *Max* columns. Latency is reported in nanoseconds and maximum steady-state frequency is given in megahertz (MHz).

Table 5.8: Performance of the Asynchronous-Synchronous FIFO with Protocol Converter

Latency (ns)		Maximum Steady-State
Min	Max	Frequency (MHz)
2.95	3.56	843.2

5.5 Projected 8-Terminal Network Layout

This section presents simulation results for latency and throughput of an 8-terminal Mesh-of-Trees network. Limitations of commercial CAD tools make a full place and route of either network infeasible. Therefore, a network projection is created that adds wire delays extracted from a synchronous network design [5] between primitives to give an approximation of the performance of a post-layout asynchronous network.

First, we explain the methodology for creating the network projection. Then, the simulation environment used to evaluate latency and throughput of the network is presented. Finally, results are presented for two networks, the asynchronous and the mixed-timing.

5.5.1 Methodology for Network Projection

In order to approximate the performance of a placed and routed network, a projection is created. The projection is necessary because current CAD tools do not support optimizations needed for the asynchronous design.

First of all, current tools do not support bundling constraints or path-length matching [29], which is used extensively in the network. Additionally, current tools will not optimize, and therefore not perform proper buffer insertion, on paths without a specified clock or with combinational loops. The asynchronous network has no clocked elements and primitives have combinational loops. This provides non-trivial challenges for designing and verifying a full network layout.

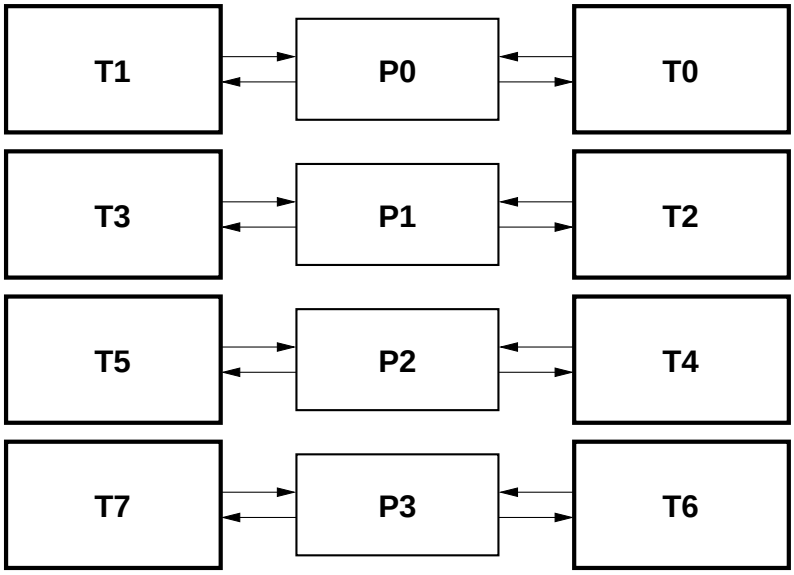


Figure 5.2: 8-Terminal Network ASIC Floorplan [4]

The projected full network layout consists of individual primitives, treated as hard macros, with appropriate wire delays inserted in between. Wire delays are assigned based on a floorplan similar to the one used in [5], where an 8-terminal synchronous Mesh-of-Trees network was fabricated in March 2007 using the same ARM standard cells and IBM 90nm technology as in this research.

ASIC Floorplan: Figure 5.2 shows the floorplan of the network ASIC. T0...T7 are synchronous terminals used to generate traffic and record throughput and latency measurements. P0...P3 are four partitions of the MoT network. Each partition was separately placed, routed, and optimized.

Partitions are formed by dividing the MoT network into four physical slices. Each partition contains two complete fan-in trees, corresponding to the two terminals that it interfaces with. Fan-out trees are split between partitions. Leaf primitives of the fan-out tree are placed in the partition corresponding to their two

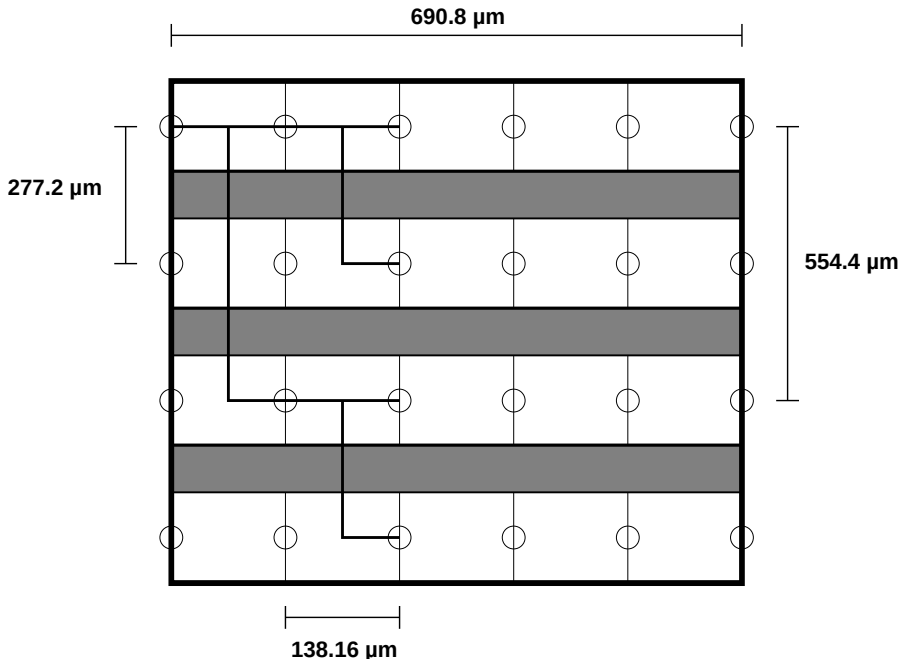


Figure 5.3: Projected 8-Terminal Network Floorplan

routing destinations. For example, a fan-out leaf primitive that routes to terminals T2 and T3 is placed in partition P1.

In the network ASIC, primitives are placed and optimized by the CAD tools. The location of each primitive is limited to the boundaries of the partition, but the placement within the partition is determined by the tools. For the projected layout used for the asynchronous network, we make a simplifying assumption to find reasonable delays for inter-primitive wires.

Projected Network Floorplan: Figure 5.3 shows the floorplan used to assign wire delays between primitives for the projected network layout. The dimensions are measured from the layout of the fabricated network ASIC. Since this is an 8-terminal network, fan-out and fan-in trees are 3 levels deep. The circles at intersections represent placement locations for routing and arbitration primitives. The shaded

areas are separation between the partitions.

The path drawn within the network shows fan-out tree connections for a tree rooted at T1. In this case, the children of the root are placed in partitions P0 and P2. As with each fan-out tree, the four leaf primitives are all in different partitions. This way, connections can be made to all eight fan-in trees, which exist entirely within a partition.

Wire delays are assigned by their distance traveled, using this simplified placement. In the synchronous network, network latency is constant for each source to each destination, since each packet must traverse a three-level fan-out tree and a three-level fan-in tree. The clock cycle determines the latency per stage.

Asynchronous circuits are self-timed. In other words, there is no global clock to synchronize the flow of data through the stages of the network. This means that the network latency from T1 to T6 will be longer than T1 to T1. In the context of XMT, which is a uniform memory access (UMA) parallel architecture, this may challenge the UMA status. However, the majority of time spent in the network is within the individual primitives, and each point-to-point communication traverses the same number of primitives.

5.5.2 Simulation Setup

Latency and throughput evaluation of the asynchronous network is conducted according to the standard interconnection network measurement setup described in [15]. In this setup, terminals are placed at each port of the network. *Terminal*

instrumentation is responsible three tasks:

1. Generating input packets at specific time intervals

2. Recording input packet count and timing information

3. Recording output packet count and timing information

The simulation is split into three distinct phases: warm-up, measurement, and drain. The warm-up phase is used to bring the network to equilibrium for the given input traffic rate. During the measurement phase, packets are tagged with their start time, and recorded when they exit the network. During the drain phase, packets are no longer generated, and all remaining packets in the network are recorded until the network is empty.

The following sections present simulation results for latency and throughput of the asynchronous network with and without mixed-timing FIFOs.

5.5.3 Asynchronous Network

This section presents average latency and throughput measurements of the asynchronous network using the *terminal instrumentation* described in the previous section. These are the standard evaluation metrics used for interconnection networks, and experiments are conducted according to [15].

Latency is the time required for a packet to traverse the network, and is measured as the time from creation of the packet, which occurs before insertion, until it exits the network at its destination. Throughput is the output data rate,

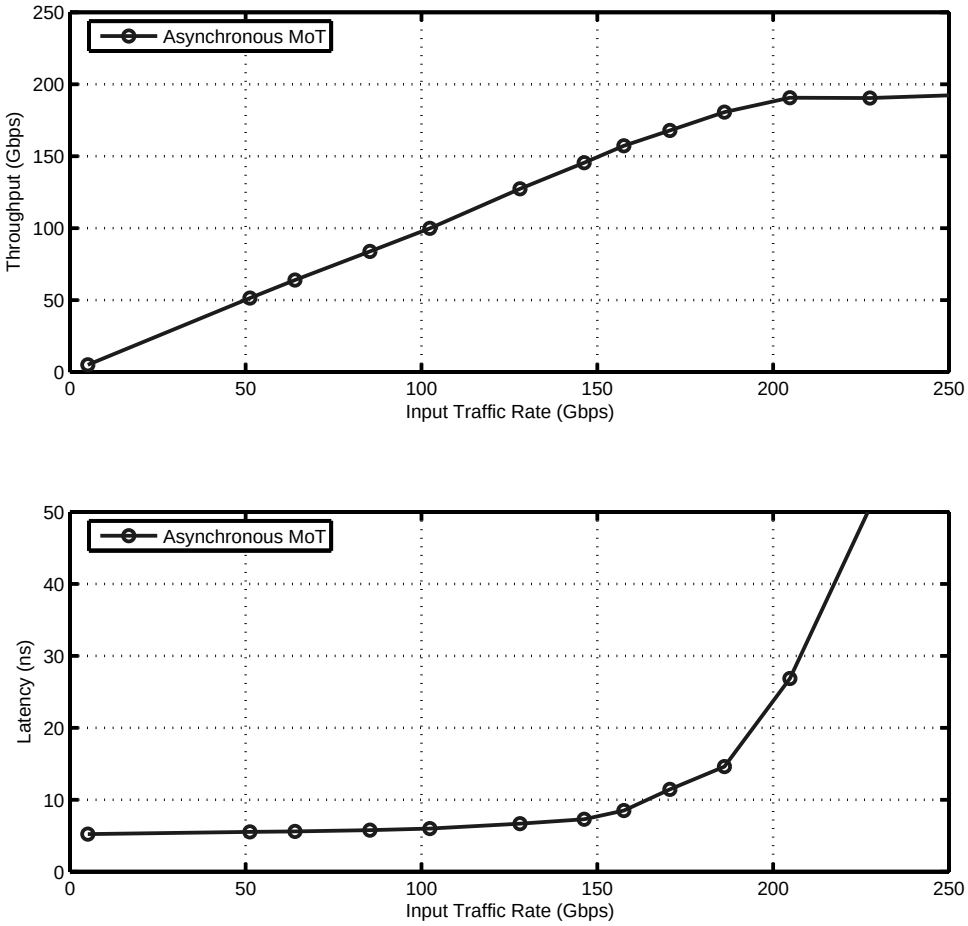


Figure 5.4: Throughput and Latency of Projected 8-terminal Asynchronous Network Layout

called the *accepted traffic rate*, and is measured as the number of packets exiting the network during the *measurement phase*.

Experiments are conducted using uniformly random traffic input into the network. Packets are generated at time invervals based on an exponential distribution, then inserted into a source queue until they can be inserted into the network. The mean of the exponential distribution is the *offered traffic rate*. The count of packets

exiting the network during the *measurement phase* is used to compute the *accepted traffic rate*. Throughput is plotted as accepted traffic versus offered traffic.

As packets exit the network, their latency from source to destination is recorded and this is used to assess the average latency of the network. Latency is plotted as average latency among all terminals versus offered traffic.

The asynchronous network reaches a maximum throughput of 190 Gb/s, with a 32-bit datapath and the projected 8-terminal layout. Increased input traffic beyond this point results in an exponential increase in the average latency per packet. This is the point of saturation. Thoughts on improving the maximum throughput of the network appear in section 6.2.

5.5.4 Mixed-Timing Network

In this section, we present simulation results for the asynchronous network with mixed-timing interfaces and protocol converters. Two metrics are measured at varying clock rates in the experiment: *maximum throughput* and *bandwidth utilization*. Maximum throughput is the highest *accepted traffic rate* measured for a given clock frequency, used for synchronous senders and receivers. Bandwidth utilization is the fraction of the bandwidth, the ideal maximum throughput, utilized during the experiment.

The purpose of the experiment is to observe how the change in clock frequency affects network performance. This is important because the behavior of the asynchronous network changes relative to the clock frequency of the external

synchronous environment. The asynchronous network has a self-timed, data-driven behavior where each primitive operates based on interactions with immediate neighbors, independent of the clock frequency used for synchronous senders and receivers. Therefore, changing the clock frequency creates different traffic patterns within the network, and this experiment highlights one of the differences.

The synchronous network, on the other hand, has a behavior dependent on the rising edge of the clock signal used for synchronous senders, receivers, and the network itself. Therefore, the behavior is the same regardless of the clock frequency, using the same amount of *clock cycles* to process the same input traffic. The behavior of the network, relative to a clock cycle, does not change as the frequency of the clock input changes.

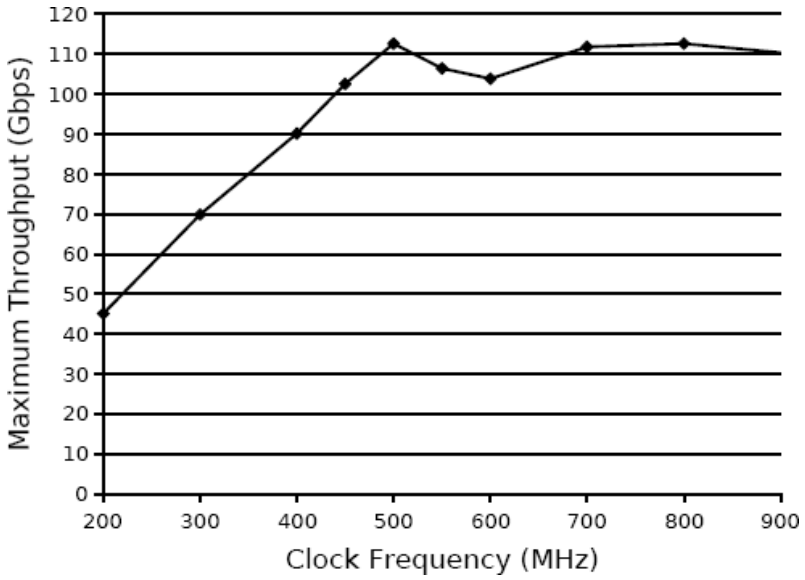
The simulations to measure maximum throughput and bandwidth utilization at varying clock frequencies use the *terminal instrumentation* described in section 5.5.2. The synchronous terminals are implemented in Verilog for use with the projected 8-terminal network layout and mixed-timing FIFOs. The maximum *offered traffic rate* is one flit per cycle per port. Packets are generated only at discrete event times, at the rising edge of the clock, based on a *binomial distribution* with probability equal to the desired fraction of the maximum offered traffic rate. For example, 80% offered traffic rate means that 80% of clock cycles should generate a new packet. All synchronous components are driven by the same clock for these experiments.

The expected results with the synchronous network are as follows. The maximum throughput will increase linearly as the clock frequency increases – a higher

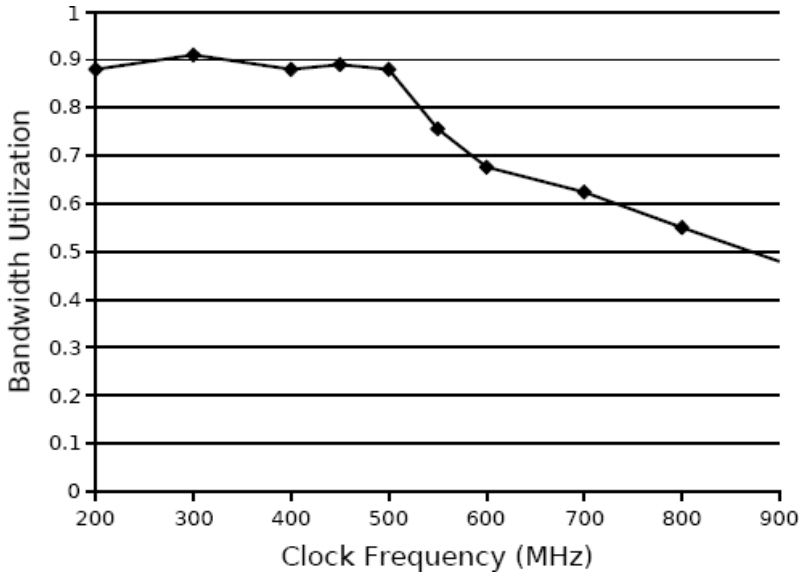
clock frequency increases the rate of data transfer throughout the network. By the same reasoning, the *bandwidth* of the network, the ideal maximum throughput, also increases linearly with the clock frequency. The *bandwidth utilization*, measured as the fraction of the bandwidth used by the maximum throughput at a given clock frequency, will therefore remain constant in the synchronous case. Both maximum throughput and bandwidth increase linearly with the clock frequency.

Figure 5.5 shows the simulation results for maximum throughput and bandwidth utilization of the mixed-timing network at varying clock frequencies. In the region between 200 MHz and 500 MHz, the results match the expected results from a synchronous network. The maximum throughput increases linearly and the bandwidth utilization remains constant. However, above 500 MHz, the maximum throughput ceases to increase and approaches an absolute maximum of 112 Gbps. This is a new type of saturation point, where the mixed-timing network can no longer deliver more traffic. The result is that bandwidth utilization will decrease linearly with increasing clock frequency, as bandwidth increases and maximum throughput remains constant.

The addition of mixed-timing FIFOs with protocol converters create performance bottlenecks for the mixed-timing network. One example is the *empty* detection for the asynchronous-synchronous FIFO, described in detail in [12]. In order to prevent underflow in the FIFO, the *empty detector* will trigger an early *Empty* signal. When this occurs, a stall cycle is inserted at the output of the network, and this hurts throughput. In the saturation region (above 500 MHz), this case is exercised regularly, indicating that even with high levels of input traffic, the asynchronous-



(a)



(b)

Figure 5.5: Projected 8-Terminal Network Layout with Mixed-Timing FIFOs (a) Maximum Throughput (b) Bandwidth Utilization of the for Different Clock Frequencies

synchronous FIFOs are not supplied with enough data.

Throughput optimizations to asynchronous network may help this situation. One possible method of increasing performance is adding pipeline stages near root nodes of the network. Pipeline stages, implemented as MOUSETRAP [31] asynchronous pipelines, provide fast acknowledgment, freeing the sender to start processing new data earlier. This provides a "pulling" effect in the network that helps enter data faster into the network at the fan-out roots, and queue new data for removal at the fan-in roots. Thoughts on improving the maximum throughput of the network are discussed further in section 6.2.

5.6 Simulation with the XMT processor

This section presents simulation results for the XMT processor with the mixed-timing network. We measure execution time for four benchmarks and report speedups relative to the XMT processor with the synchronous network. The goal of the experiments is to assess how the performance of XMT is affected with the substitution of the mixed-timing network. Also, XMT provides a real application and traffic load for the network, as opposed to uniformly random traffic used in previous experiments.

The performance of the XMT processor depends on network latency and the number of round-trips to memory required by each parallel thread. As shown in Figure 5.4, the latency of the network is affected by the amount of input traffic and the saturation throughput of the network. The simulations with four XMT benchmarks will indicate the advantages and disadvantages of the mixed-timing network

when used with the XMT processor. Additionally, the various traffic patterns of the four benchmarks will each have a different effect on the network performance, and potentially reveal areas for improvement and optimization.

Simulations were conducted using the XMT Verilog model developed by Xingzhi Wen [36] and the Verilog model for the mixed-timing network. The following four XMT benchmarks were simulated:

1. **Array Summation (add):** An array with 3 million elements is divided into sub arrays. Each parallel thread computes the sum of a sub array serially. The resulting sums are added to compute the total for the entire array.

2. **Matrix Multiplication (mmul):** The product of two 64×64 matrices is computed. Each parallel thread computes one row of the result matrix.

3. **Breadth-First Search (bfs):** The breadth-first search algorithm described in [36] is executed on a graph with 100k vertices and 1 million edges.

4. **Array Increment (a_inc):** Each element of an array with 32k elements is incremented. Each parallel thread increments 8 elements of the array.

Results for speedups with the mixed-timing network are normalized relative to performance with the synchronous network. Speedups of the four benchmarks are shown in Table 5.9. Simulations were performed for clock rates of 200 MHz, 400 MHz, and 700 MHz. The synchronous network has speedup of 1.0 in all cases.

The asynchronous network with mixed-timing performs similar to the synchronous network for all the 200 MHz and 400 MHz cases, but has increased execu-

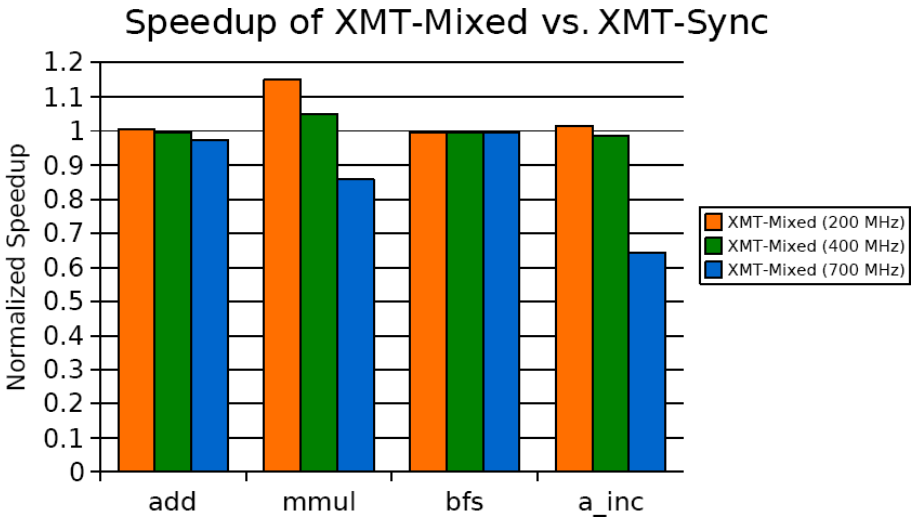


Figure 5.6: Speedups of XMT with Mixed-Timing Network vs. Synchronous Network for four XMT benchmarks

Table 5.9: Average Network Utilization for Four XMT Benchmarks

	add	mmul	bfs	a_inc
Avgerage Network Utilization	0.492	0.395	0.091	0.927

tion time on *mmul* and *a_inc* for 700 MHz simulations. This occurs because these applications put significant pressure on the network and the input traffic rate at times surpasses the saturation traffic rate (see Section 5.5.4). This helps make the case that performance in XMT depends on the network latency [34], which increases with higher input traffic. Once input traffic surpasses the saturation bandwidth, performance degrades.

The *add* benchmark provides steady traffic throughout the execution of the program that remains below the saturation throughput of the network with mixed-timing. Therefore, the performance remains comparable to an XMT with the synchronous network. The *mmul* benchmark has a lower average network utilization

than *add*; however, this does not reflect the true pattern of memory access. The traffic appears in bursts, causing very high traffic, followed by periods of low traffic as the processing clusters compute the results. In the 700 MHz case, the burst traffic exceeds the saturation throughput, indicated in Figure 5.5, thereby degrading the performance of the application.

Breadth-first search (*bfs*) has the lowest traffic rate of all benchmarks. Since it operates below the saturation throughput of the network, performance is comparable to the synchronous for all simulations. The extremely high traffic of the *a_inc* benchmark has a direct correlation to the increase in cycle time at 700 MHz execution. The average input traffic at 700 MHz will be 164.9 Gb/s, which exceeds the saturation throughput of 112 Gb/s. The network latency increases exponentially under that level of traffic, and the performance of the XMT processor suffers.

With further throughput optimizations, discussed later in section 6.2, the asynchronous network may provide enough throughput to maintain good performance for XMT all benchmarks.

Chapter 6

Discussion and Conclusion

This chapter presents a summary of the findings and future directions for asynchronous Mesh-of-Trees research. First, we present a brief evaluation of the asynchronous network using four key metrics: area, power, throughput, and latency. Then, future directions for asynchronous Mesh-of-Trees research are presented. Finally, we present our closing remarks.

6.1 Discussion

Summary of Contributions: The main contributions of this thesis are the development of enhanced asynchronous primitives, analytical performance and timing constraints equations, protocol converters for interfacing the asynchronous network and mixed-timing interfaces, implementation and detailed simulation.

The enhancements to the basic network primitives are essential for correct operation and integration into a larger system. The challenge of incorporating the optimizations is to minimize the performance penalties that may result from added logic. The addition of low-overhead multi-flit capability is a new contribution, not appearing in previous publications on transition-signaling asynchronous pipelines. Together with power optimizations to the basic design from [3], the enhanced network primitives form a foundation for high-performance, low-power pipelined asyn-

chronous interconnects.

Detailed simulation involved evaluation of isolated primitives and comparison to synchronous, as well as the development of a projected 8-terminal network layout for evaluation of a full network. Furthermore, incorporating the mixed-timing network into the eXplicit Multi-Threading (XMT) Verilog model for co-simulation was a significant contribution.

This research looked at the problem of developing a high-throughput, low-latency asynchronous network and compatible mixed-timing interfaces for use in a shared-memory single-chip parallel processing architecture. Simulations of the mixed-timing network with the XMT Verilog model represent a non-trivial context for evaluating the network where low-power and high-performance are first-class design constraints. A brief evaluation of the asynchronous network is presented below in terms of area, power, throughput and latency.

Summary of Results: This research demonstrated the advantages of a high-throughput, low-power asynchronous Mesh-of-Trees interconnection network. The asynchronous network primitives for routing and arbitration use significantly lower area than their synchronous counterparts presented in [4, 5]. The asynchronous primitive consume less power than the synchronous primitives by having lower area overheads for storage and eliminating the need for clock distribution.

In isolation, the asynchronous primitives have similar performance compared to the synchronous. In the projected network layout, the asynchronous network achieved high throughput, but performed worse than the highly-optimized synchronous network layout. Limitations of commercial CAD tools for asynchronous de-

signs and the projection itself are cited as explanations for the differences. Throughput optimizations to the network, such as pipeline insertion, should improve overall network throughput and is discussed in section 6.2.

Throughput of the mixed-timing network was evaluated for different clock frequencies of synchronous senders and receivers. The simulations revealed some performance bottlenecks created by mixed-timing interfaces with the unoptimized asynchronous network. However, throughput optimizations to the basic network topology should improve maximum throughput in the mixed-timing network.

Area: The asynchronous network has significant area savings compared to the synchronous network. Reducing area overheads of the network is important at the system level, as interconnection networks occupy valuable chip area that can be used for additional processors or caches. Additionally, savings in area contribute directly to reduced power consumption.

The asynchronous routing primitive used 64% less cell area than the synchronous. The asynchronous arbitration primitives used 84% and 74% less cell area than the synchronous, for LP and TPP respectively. The substantial area savings at the primitives should translate to substantial savings at the network level as well. Since the synchronous and asynchronous networks share the same same topology, the asynchronous network should have savings comparable to the primitive-level savings, assuming the same floorplan for the network and layout optimizations (repeater insertion, cell resizing, etc).

Much of the savings occur on the datapath of the primitives. The asynchronous primitives use standard transparent D-type latches to store data, compared to edge-

triggered flip-flops used by the synchronous. Each latch from the standard cell library uses 40% less area than the clock registers [2].

At the network-level, the mixed-timing network has mixed-timing interfaces that add some area overhead. However, as the network scales to larger sizes, the number of mixed-timing interfaces grows with $O(n)$, while the number of network primitives grows with $O(n^2)$. Therefore, the impact on total network area becomes less of a factor for large-scale networks. The synchronous network also must distribute the global clock tree at the network level, and uses considerable area for buffers and repeaters to maintain the low-skew properties necessary for correct operation. The asynchronous network, however, saves on area used for clock tree distribution, and proves to be area-efficient at each level of implementation.

Power: The asynchronous network has substantially lower power consumption than the synchronous network. Lower power consumption is important to chip designers, as high-power designs are increasingly difficult to package and cool effectively. Power consumption is a first-class design constraint for battery-powered devices as well. The power savings of the asynchronous network come from the following areas:

1. *Network Primitives*: The asynchronous primitives have lower average power consumption due to the structure of the datapath, cell selection on the datapath, and the absence of the clock tree. In both the asynchronous and synchronous networks, the elements of the datapath – storage elements and multiplexers – occupy the largest percentage of area.

In the synchronous network, data is stored using clock registers arranged using the latency-insensitive design methodology described in [10]. This method improves throughput in the synchronous primitives at the cost of additional power and area. The asynchronous primitives do not require additional storage or logic to maintain high throughput in the network.

The asynchronous primitives do not require a clock distribution, thereby saving dynamic power. Clock tree power accounts for 10% of the power consumed in the synchronous primitives in isolation. The combination of low-overhead datapath and absence of the clock tree yields 10x power savings for the arbitration primitives and 7x for the routing primitives.

2. *No Global Clock Distribution*: The asynchronous network does not require a global clock distribution. The global clock distribution in the synchronous network consumes approximately 20% of the total power. Without proper clock gating, the synchronous network will continue to consume dynamic power from the clock tree even when the network is at a quiescent state (no traffic). The asynchronous network, on the other hand, provides "perfect clock gating", where power is spent only as the result of activity in the network.

This work did not provide a detailed power study of a full network layout, due to limitations in the commercial CAD tools. However, based on the data collected, the asynchronous network should provide high throughput using less power than the synchronous network.

Throughput and Latency: Throughput and latency were measured for the network primitives in isolation as well as full 8-terminal networks for both synchronous and asynchronous designs. Simulation results showed that asynchronous and synchronous primitives had similar performance in isolation. The synchronous network maximum throughput, however, was much higher than the asynchronous network for the 8-terminal network simulations. There are proposed optimizations that should improve maximum throughput in the asynchronous network, described in section 6.2.

Based on the results presented in section 5.5, the maximum throughput of the projected 8-terminal network layout is 190 Gbps for a 32-bit datapath. The synchronous 8-terminal network has maximum throughput of 348 Gbps, operating at a clock frequency of 1.36 GHz with 32-bit datapath. There are a few explanations for the large difference.

First, the projected network layout is a pessimistic assumption for inter-primitive wire delays. Commerical CAD tools will frequently optimize the drive strength of cells to improve timing by reducing the transition time on the wires. The synchronous design took full advantage of these optimizations. The projected network layout for the asynchronous network approached primitives as hard macros, a pre-placed and routed module that is not optimized internally. Then, delays were added for wires connecting the primitives. This approach does not allow the cell-resizing optimizations available to the synchronous design at the layout level.

Second, the inter-primitive wire delays have a larger impact on the asynchronous network than the synchronous. For each cycle of the asynchronous, a

full handshake takes place between adjacent primitives. The asynchronous network uses *transition signaling*, meaning two inter-primitive wire delays are required per cycle, one for request and one for acknowledgment. In the synchronous, on the other hand, a timing path between clocked elements has at most one inter-primitive wire delay. Therefore, long wires between primitives have a larger negative impact on performance in the asynchronous.

The issue of pessimistic wire delays can be solved by the advancement of CAD tools for asynchronous design. Future tools that provide increased support for specifying timing constraints and optimizing placement and latency of clockless paths will make the design of large-scale asynchronous designs more feasible. Performance of the asynchronous network would improve beyond the projection with the aid of commerical CAD tools that could properly place, route, and optimize the layout.

Reducing the negative impact of long inter-primitive wiring delays can be achieved by inserting asynchronous pipeline stages as buffers for long wires. This optimization will have a positive impact on the throughput of the asynchronous network projection and future fully placed and routed asynchronous networks. Furthermore, adding more pipeline stages near the roots of fan-out and fan-in trees may provide additional throughput improvements. This and other enhancements are the focus of future work on the asynchronous network.

6.2 Future Work

This section presents future directions for research on the asynchronous Mesh-of-Trees interconnection network.

Full Layout and Power Study: As stated in the previous section, the projected network layout is only a first-cut solution. A full network layout, with careful attention paid to cell sizing and other gate-level optimizations, would provide a true measure of the performance of the asynchronous network. In addition, the asynchronous and synchronous network power consumption can be compared by assessing the distribution of between clock networks, primitives, and interconnects. Different size networks should also be placed, routed, and analyzed for performance and power consumption, CAD tools permitting.

Optimizing Root Nodes: Since the roots of the fan-out and fan-in trees determine the maximum performance, any optimizations on these primitives may provide performance benefits. For example, using low threshold voltage (LVT) cells from the standard cell library instead of the regular cells boost performance but at the cost of added power consumption. However, the number of root nodes grows as O(N), where N is the number of terminals of the network. The number of primitives grows as $O(N^2)$. Therefore, increasing the power consumption of these primitives will have less effect on network power consumption as the design scales up.

Throughput Optimization through Added Pipeline Stages: The key to improving throughput in the network is optimizing the performance of the root nodes. In the case of the fan-out root, providing fast acknowledgments to outgoing

requests allows the routing primitive to operating near maximum throughput for all input conditions, as shown in Table 5.1. For the fan-in root, best performance is achieved with requests pending on both inport ports, allowing for acceptance of alternating requests, shown in Table 5.4.

Adding pipeline stages to the network may improve the maximum throughput. Pipeline stages inserted at the outputs of the fan-out tree roots, will provide fast acknowledgment to the root while safely storing the data. This allows the root primitive to route another packet to the same output port faster than before, since the acknowledgment arrives faster, thus improving the input throughput. Adding multiple pipeline stages creates a queue where new packets can be inserted quickly into the network.

Pipeline stages inserted at the inputs of the fan-in tree roots form a queue of new data items to be output from the network. Once the first-level arbitration primitives transfer data to the pipeline stage(s), they receive a fast acknowledgment, freeing them to process the next data waiting data item.

Experiments should be conducted to assess where pipeline stages should be inserted to have maximum impact, as well as determine the optimal number of stages. The pipeline stages, implemented as MOUSETRAP [19] asynchronous pipelines, provide the desired functionality. Additionally, each stage is normally transparent (latches enabled), allowing request and data to advance through an empty queue with very low latency. Unlike synchronous pipeline stages, which add an entire clock cycle to the latency of the network, an asynchronous pipeline stage has a latency equal to one transparent latch when empty. Therefore, multiple stages could be

inserted with relatively low impact on network latency.

Experiments with Mixed Clock Rates: In chapter 5, the synchronous domains operating with the mixed-timing network all operated using a single clock. The mixed-timing network – asynchronous network with mixed-timing interfaces – is capable of interfacing with multiple synchronous domains that have different and unrelated clock rates. The proposal for future experiments is to examine the effects on network performance of multiple synchronous senders and receivers that operate at different clock frequencies.

Experiments with Dynamic Voltage/Frequency Scaling: Power optimizations such as dynamic voltage and frequency scaling can be applied to individual synchronous domains. With each optimization, the clock frequency of synchronous domains can change dynamically from software or hardware methods. The mixed-timing network allows flexibility at the high-level architecture for multiple synchronous domains operating at different and unrelated clocks. Furthermore, the mixed-timing network should gracefully handle dynamically changing frequencies of individual synchronous domains. Future work should explore the benefits of integrating the mixed-timing network into power-optimized synchronous architectures.

6.3 Conclusion

This thesis presented two new asynchronous designs for the fundamental pipelined components of the Mesh-of-Trees network. Both designs use transition signaling (two-phase) for handshaking signals, allowing for fast communication between mod-

ules since half the transitions are required compared to the more common four-phase handshaking protocols. Data is stored using standard transparent latches, which have lower area and power consumption overheads compared to edge-triggered flip-flops. The architecture and basic operation were presented, as well as analytical performance equations and timing constraints for both designs. The asynchronous routing and arbitration primitives were evaluated in isolation for area, power, throughput and latency, and compared to synchronous designs presented in [4, 5].

Due to limits in commercial CAD tools, implementation of a full network layout was not possible as part of this research. To obtain an approximation for performance of a post-layout full network, a projected 8-terminal network layout was developed. Arranged in the Mesh-of-Trees topology, the projected post-layout network was evaluated for maximum throughput and average latency under varying levels of input traffic.

To interface with systems in different timing domains, mixed-timing FIFOs were implemented based on the designs in [12]. New protocol converters were implemented to provide communication between transition-signaling routing and arbitration primitives and the four-phase signaling mixed-timing FIFOs. The mixed-timing FIFOs allow the asynchronous network to be embedded into the XMT parallel processor. Simulations are conducted to assess the performance of XMT with synchronous and asynchronous networks for four benchmark applications.

The eXplicit Multi-Threading (XMT) architecture, through independence-of-order semantics, operates with reduced synchronization between processors and memory. Execution proceeds as data becomes available, without any synchroniza-

tion between threads or tight scheduling for memory operations. This global asynchrony exists at all levels of the architecture, which was designed to fit the PRAM algorithms that it executes. The bridge from algorithms to hardware is taken one step further with the implementation of the asynchronous interconnection network. With the added benefits of lower power and area costs, the asynchronous Mesh-of-Trees network is a perfect fit for XMT.

Bibliography

[1] ARM Limited Inc., *Advanced Microcontroller Bus Architecture Specification*, rev. 2.0 edition, 1999.

[2] ARM Physical IP Inc. *CMOS9SF (90nm) RVT Process 1.2-Volt SAGE-X^{TM} v3.0 Standard Cell Library Databook*, rev 1.2 edition, May 2006.

[3] M. Carlberg and S.M. Nowick, *Progress Towards Designing an Asynchronous Interconnect Fabric for the XMT Parallel Processing Architecture*, Non-disclosed Research Write-Up, 2007.

[4] A. Balkan, *Mesh-of-Trees Interconnection Network for an Explicitly Multi-Threaded Parallel Computing Architecture*, Ph.D. Thesis, University of Maryland, College Park, 2008.

[5] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin, *Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing*, In Proceedings of IEEE Symposium on High Performance Interconnection Networks (Hot Interconnects), Stanford University, CA, August 2007.

[6] A. O. Balkan, G. Qu, and U. Vishkin, *A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing*, In Proceedings of Application-Specific Systems, Architectures and Processors (ASAP), pp. 73-80, 2006.

[7] A. Balkan, G. Qu, and U. Vishkin, *Arbitrate-and-Move Primitives for High Throughput On-Chip Interconnection Networks*, In Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), vol. 2, pp. 441-444, May 2004.

[8] M. Benes, S.M. Nowick, and A. Wolfe, *A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors*, IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 1998.

[9] E. Brunvand, *Translating Concurrent Communicating Programs into Asynchronous Circuits*, Ph.D. Dissertation, Carnegie Mellon University, 1991.

[10] L. P. Carloni, K. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, *A Methodology for Correct-by-Construction Latency Insensitive Design*, In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 301-315, 1999.

[11] D. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*, Ph.D. thesis, Stanford University, October 1984.

[12] T. Chelcea and S.M. Nowick, *Robust Interfaces for Mixed-Timing Systems*. IEEE Transactions on Very Large Scale Integration Systems, vol. 12, no. 8, pp. 857-873. Aug. 2004.

[13] R. Y. Chen, N. Vijaykrishnan, and M. J. Irwin, *Clock Power Issues in System-on-a-Chip Designs*, In Proceedings of the IEEE Computer Society Workshop on VLSI, pp. 48, 1999.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers*, IEICE Transactions on Information Systems, vol. E80-D, no. 3, pp. 315-325, Mar. 1997.

[15] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco, CA, 2004.

[16] P. Day and J. V. Woods, *Investigation into Micropipeline Latch Design Styles*, IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 3, no. 2, pp. 264-272, June 1995.

[17] R. M. Fuhrer, S.M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana, *MINIMALIST: An Environment for Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines*, Columbia Univ. Dept. of Computer Science, New York, Tech. Report CUCS-020-99, 1999.

[18] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters and G. Stegmann, *An Asynchronous Low-Power 80C51 Microcontroller*, IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 96-107, 1998.

[19] G. Gill and M. Singh, *Advanced MOUSETRAP Pipeline Circuits*, Non-disclosed Research Write-Up, 2007.

[20] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, D. Lundqvist, *Lowering Power Consumption in Clock by Using Globally Asynchronous Locally Synchronous Design Style*, Design Automation Conference (DAC), pp. 873-878, 1999.

[21] M. N. Horak, S.M. Nowick, U. Vishkin, Paper to appear, 2009.

[22] A. Iyer and D. Marculescu, *Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors*, Proceedings of the International Symposium on Computer Architecture, pp. 158-168, 2002.

[23] M. Kistler, M. Perrone, and F. Petrini, *Cell Multiprocessor Communication Network: Built for Speed*, IEEE Micro, vol. 26, issue 3, pp. 10-23, May-June 2006.

[24] K. Keutzer, A. R. Newton, J. M. Rabaey and A. Sangiovanni-Vincentelli, *System-Level Design: Orthogonalization of Concerns and Platform-Based Design*, IEEE Transactions on Computer-Aided Design, vol. 17, no. 2, pp. 1523-1543, April 2000.

[25] R. Kumar, V. Zyuban, and D. M. Tullsen, *Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling*, In Proceedings of the 32nd International Symposium on Computer Architecture, pp. 408-419, June 2005.

[26] J. Muttersbach, T. Villiger and W. Fiehter, *Practical Design of Globally-Asynchronous Locally-Synchronous Systems*, In Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 52-59, 1999.

[27] K. Niyogi and D. Marculescu, *System-Level Power and Performance Modelling of GALS Point-to-Point Communication Interfaces*, IEEE International Symposium on Low Power Electronics and Design, pp. 381-386, Aug. 2005.

[28] S.M. Nowick and P. Beerel, *CaSCADE: Columbia University and University of Southern California Asynchronous Design Environment*, Columbia University, Available: http://www.cs.columbia.edu/~nowick/asynctools/, 2007.

[29] B. Quinton, M. Greenstreet, and S.J.E. Wilton, *Asynchronous IC Interconnect Network Design and Implementation Using a Standard ASIC Flow*, In Proceedings of the 2005 International Conference on Computer Design, pp. 267-274, Oct. 2005.

[30] S. Rotem, K. S. Stevens, R. Ginosar, P. A. Beerel, C. J. Myers, K. Yun, R. Kol, C. Dike, M. Roncken and B. Agapiev, *RAPPID: An Asynchronous Instruction Length Decoder*, IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), April 1999.

[31] M. Singh and S. Nowick, *MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines*, IEEE Transactions on Very Large Scale Integration Systems, vol. 15, no. 6, pp. 684-698, June 2007.

[32] I. Sutherland, *Micropipelines*, Communications of the ACM, vol. 32, issue 6, pp. 720-738, June 1989.

[33] J. Tierno, A. Rylyakov, S. Rylov, M. Singh, P. Aspadu, S.M. Nowick, M. Immediato, and S. Gowda, *A 1.3 GSample/s 10-tap Full-rate Variable-Latency Self-timed FIR with Clocked Interfaces*, ISSCC: International Solid State Circuits Conference, 2002.

[34] U. Vishkin, G. Caragea, and B. Lee, *Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform*, Technical Report UMIACS-TR 2006-21, University of Maryland, College Park, 2006.

[35] U. Vishkin, and J. Nuzman, *Circuit Architecture for Reduced-Synchrony On-Chip Interconnect*, US Patent 6,768,336, filed June 11, 2002, and issued July 27, 2004.

[36] X. Wen, *Hardware Design, Prototyping, and Studies of the Explicit Multi-Threading (XMT) Paradigm*, Ph.D. thesis, University of Maryland, College Park, 2008.

[37] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J.F. Brown III, and A. Agarwal, *On-Chip Interconnection Architecture of the Tile Processor*, IEEE Micro, vol. 27, no. 5, pp. 15–31, 2007.

[38] K. Yun, P. Beerel, and J. Arceo, *High-Performance Asynchronous Pipeline Circuits*, IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 17-28, 1996.

[39] K. Yun and A.E. Dooply, *Pausible Clocking-Based Heterogeneous Systems*, IEEE Transactions on Very Large Scale Integration Systems, vol. 4, pp. 482-488, Dec. 1999.