# AUTOMATED GENERATION OF ROUND-ROBIN ARBITRATION AND CROSSBAR SWITCH LOGIC

A Thesis
Presented to
The Academic Faculty

by

**Eung S. Shin**

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
November 2003

# AUTOMATED GENERATION OF ROUND-ROBIN ARBITRATION AND CROSSBAR SWITCH LOGIC

*"In his heart a man plans his course, but the LORD determines his steps...."*

*– Proverbs 16:9*

*To my parents*

# ACKNOWLEDGMENTS

During my Ph. D. study, there are many people in Georgia Tech to whom I am thankful. First of all, I would like to express enormous appreciation to my adviser, Dr. Vincent J. Mooney III, from the bottom of the heart. In addition to his enthusiasm and professionalism dedicated to all members of our Codesign group, Dr. Mooney has been supporting and encouraging me to develop my thesis. With our weekly regular meeting, he has been listening to my idea patiently, and we have been brainstorming by short question and answer session. He has been also helping me improve my writing with logical reasoning and has been correcting my English pronunciation. His technical acumen, integrity and concern for all members of Codesign are remarkable and exemplary.

Second of all, I am also grateful to all my committee members, Dr. George F. Riley, Dr. Sung Kyu Lim, Dr. Mary Ann Ingram and Dr. Santosh Pande. Especially, Dr. Riley is the coauthor of two of my papers and has been supportive to enhance our paper quality by developing a switch arbiter simulator.

Also, I have to express thank to all members of the Codesign group. Mohamed Shalan, the coauthor of one of my papers, has been helpful me with productive discussion of our research interest and the provision of constructive advice for simulations. Pramote Kuacharoen, Tankut Akgul and his wife Bilge Saglam Akgul have been encouraging me with helpful advice and hilarious joke and have been broadened my research interest. Jun Cheol Park and Kyeong-Keol Ryu have been supportive for every aspect besides the power estimation and debugging my design.

Finally, I wish to thank my parents who gave me a birth and have been supportive for my entire life. Without their encourage and support, I doubt that I have could bring my thesis into final form. They have believed in my ability to complete my thesis and have made me confident in pursuing my goal. I also thank to my brother, Eung Seok Shin and my sister, You Yong Shin, for their encouragement.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREBIATIONS

| | |
|---|---|
| **ASIC** | Application Specific Integrated Circuit |
| **BA** | Bus Arbiter |
| **CAD** | Computer-Aided Design |
| **CSMA/CD** | Carrier Sense Multiple Access with Collision Detection |
| **DMMU** | Dynamic Memory Management Unit |
| **DX-Gt** | **D**ynamic memory management unit and **X**bar **G**enera**t**or |
| **GUI** | Graphical User Interface |
| **HOL** | Head-of-Line |
| **IP** | Internet Protocol |
| **LAN** | Local Area Network |
| **MAC** | Medium Access Control |
| **NoC** | Network-on-a-Chip |
| **PCB** | Printed Circuit Board |
| **PE** | Processing Element |
| **PPA** | Ping-Pong Arbiter |
| **PPE** | Programmable Priority Encoder |
| **RAG** | Round-robin Arbiter Generator |
| **RTL** | Register Transfer Level |

| | |
|---|---|
| **SA** | Switch Arbiter |
| **SAIF** | Switching Activity Interchange Format |
| **SASim** | Switch Arbiter Simulator |
| **SDF** | Standard Delay Format |
| **SIP** | Silicon Intellectual Property |
| **SoC** | System-on-a-Chip |
| **TCP** | Transport Control Protocol |
| **VOQ** | Virtual Output Queue |
| **X-Gt** | **X**bar **G**enera**t**or |

# SUMMARY

The objective of this thesis is to automate the design of round-robin arbiter logic. The resulting arbitration logic is more than 1.8X times faster than the fastest prior state-of-the-art arbitration logic the author could find reported in the literature. The generated arbiter implemented in a single chip is fast enough in 0.25 $\mu$m CMOS technology to achieve terabit switching with a single chip computer network switch. Moreover, this arbiter is applicable to crossbar (Xbar) arbitration logic. The generated Xbar, customized according to user specifications, provides multiple communication paths among masters and slaves.

As the number of transistors on a single chip increases rapidly, there is a productivity gap between the number of transistors available in a chip and the number of transistors per hour a chip designer designs. One solution to reduce this productivity gap is to increase the use of Silicon Intellectual Property (SIP) cores. However, a SIP core should be customized before being used in a system different than the one for which it was designed. Thus, to reconfigure the SIP core, either an engineer must spend significant effort altering the core by hand or else an enhanced CAD tool can automatically customize the core according to customer specifications. In this thesis, we present SIP generator tools for arbiter and Xbar generation.

First, we introduce a Round-robin Arbiter Generator (RAG). The RAG can generate a hierarchical Bus Arbiter (BA) which is faster than all known previous approaches. RAG can also generate a hierarchical Switch Arbiter (SA) which is faster than all known previous approaches. Using a 0.25 $\mu$m TSMC standard cell library from LEDA Systems, we show the arbitration time of a 32x32 SA and demonstrate that our SA meets the time constraint to achieve terabit throughput. Furthermore, using a novel token-passing hierarchical arbitration scheme, our 32x32 SA performs better than the Ping-Pong Arbiter and Programmable Priority Encoder by factors of 1.8X and 2.3X, respectively, with less power dissipation.

Finally, we present an Xbar switch Generator (X-Gt) tool that automatically configures a crossbar for a multiprocessor System-on-a-Chip (SoC). An Xbar is generated in Register Transfer Level (RTL) Verilog HDL.

# CHAPTER I

# INTRODUCTION

## *1.1 Problem Statement*

In a multiprocessor System-on-a-Chip (SoC) environment, a silicon CMOS chip designer typically uses an arbiter to resolve conflicts on shared resources (i.e., bus or equivalent communication channels) among multiple bus masters (e.g, processors). In a bus-based system, processors could be stalled because of bus conflicts. Thus, a high-performance arbiter is needed to resolve bus contentions among bus masters; such a fast arbiter can also reduce processor stall time by shortening arbitration delays.

In computer networks, such a fast and efficient arbiter commands more attention to resolve contention for crossbar switch(es) of a fast network switch as the amount of user traffic continues to double every year [35]. If the network switch capacity fails to increase with user traffic, then internet service providers may have to increase the number of switches in their network each year. Alternatively, the capacity of a single network switch needs to increase instead, and a fast and efficient arbiter plays a key role in increasing such switch capacity. Considering power consumption, such an arbiter is also preferably implemented in a single chip since the power budget of a network switch is about 10 $kW$ per rack. A recent single rack of network switches, which aims at terabit switching, have already reached this limit [20]. As mentioned in [7, 20, 26, 29, 66], arbitration delay is one of the major obstacles to achieving terabit switching.

In the era of billion transistor chips, reducing the productivity gap between the number of transistors available in a chip and the number of transistors per hour that a designer can design is a challenging issue to a silicon CMOS chip designer. The productivity gap can be reduced by enhancing Silicon Intellectual Property (SIP) core reusability, e.g., by developing CAD tools that can automatically configure and customize a core according to customer specifications.

Primarily, this thesis focuses on round-robin arbiter design and automatic arbiter generation. Secondarily, we extend automatic arbiter generation to CAD tool development for a crossbar (Xbar) switch that employs our generated arbiter for its arbitration logic.

## *1.2 Thesis Contributions*

The contributions of this thesis are summarized as follows.

- In the era of multiprocessor SoC, the importance of fast and powerful arbiters commands more attention. We have designed a fast hierarchical round-robin arbiter, which can also be employed for a high-speed network switch. Our fast arbiter addresses arbitration delay, one major obstacle in the drive to achieve terabit switching in a single chip [7, 20, 26, 29, 66].

- As billion transistor chips begin to appear, the customization of Silicon Intellectual Property (SIP) cores by chip designers becomes much more complicated and difficult, resulting in longer time-to-market. Thus, the automatic generation of a hierarchical round-robin arbiter can reduce the time required to properly customize the arbiter. Also, to the best of our knowledge, we present the first published work on the automatic generation of a round-robin arbiter.

- We also developed a crossbar (Xbar) switch generator for on-chip communication. Since the generated Xbar is orthogonal to processor types, the Xbar is valuable in the sense that it can easily be integrated into a heterogeneous multiprocessor SoC. The automated customization of an Xbar according to customer specifications is presented in a tool called Xbar Generator (X-Gt).

## *1.3 Motivation*

In a year or so integrated circuits will appear with more than one billion transistors on a single chip [42]. Such chips give designers the opportunity to integrate many functionalities, each of which used to be implemented on different chips, into the same chip. In other words, a digital system that was previously implemented on a Printed Circuit Board (PCB) will be integrated into a single chip, i.e., System-on-a-Chip (SoC). One opportunity for such chips is building an SoC that has multiple processors of different types, large memory, custom digital logic and interfaces connected by one or more on-chip buses.

Most Processing Elements (PEs) in an SoC communicate with each other via buses and memory. As the number of bus masters increases in a single chip, the importance of fast and powerful arbiters commands more attention to achieve a high-speed on-chip bus. Especially, a fast arbiter is one of the more dominant factors for high-performance network switches [7, 20, 26, 29, 66]. Also, fast and efficient switch arbiters are needed to switch packets in a Network-on-Chip (NoC) [11, 18]. However, to design with high performance and fairness in arbitration is a very tedious and error-prone task for designers.

Fast arbitration schemes are intensively studied in computer networks. A major concern in computer networks today is the design of ultra high-speed switches, which provide a high speed and cost-effective contention resolution scheme when multiple packets from different input ports compete for the same output port. This issue is extremely important in order to provide multimedia services for future Broadband Integrated Services Digital Networks (B-ISDN) [48, 59]. We will show how our Round-robin Arbiter Generator (RAG) can help in the design of a terabit switch.

Most of the current prevailing buses such as VME [63] and PCI [38] were designed for system level buses. While VME connects PCBs in large systems, PCI connects discrete devices on a PCB. A System-on-a-Chip (SoC) allows designers to overcome the drawbacks of PCBs by implementing many or most chips of a PCB onto a single silicon chip. SoC technology allows one to take advantage of increased bus speed and decreased area compared with a PCB.

In the implementation of a multiprocessor SoC, an on-chip network comes to the forefront because the performance of the system is not dependent only on the CPU speed but also on the on-chip network, which may cause system degradation resulting from communication bottlenecks in the system. An efficient bus architecture and arbitration for reducing contention plays an important role in maximizing the performance of the system. We predict that in next five years multiprocessor SoCs will be dominated by designs with four to eight processors and on-chip SRAM or DRAM of 16Mbytes to 128Mbytes. In such multiprocessor SoCs, multiple communication channels may be desired so that communication among processors does not become a system bottleneck. We utilize a crossbar (Xbar) switch for an efficient on-chip network solution.

As the number of transistors on a single chip increases rapidly, there is a productivity gap between the number of transistors available in a chip and the number of transistors per hour a designer can design. In other words, it is almost impossible for human designers to cope with number of available transistors in a chip which doubles every 18 months by Moore's Law [65], while the number of transistors per hour that a designer designs increases 21% per year [42]. One solution to reduce this productivity gap is to increase the reusability of Silicon Intellectual Property (SIP) cores. However, an SIP core should be customized/configured before being used in a system different than the one for which it was designed. Thus, to reconfigure the SIP core, either an engineer must spend significant effort altering the core by hand or an enhanced CAD tool (SIP generator) can automatically configure and customize the core according to customer specifications. For example, memory and I/O generators by Artisan [2] and processor generators by ARC [1] and Tensilica [57] supply application-specific SIP cores that can be highly tuned for specific applications.

To the best of this author's knowledge, this thesis presents the first published work on automatic generation of an Xbar switch coupled with a round-robin arbiter. More specifically, reconfiguring an Xbar is more than reconfiguring bus parameters such as address bus width and data bus width. An MxN Xbar must be configured to support an exact number of masters (processors), M, and an exact number of slaves (memory), N. Thus, to configure an MxN Xbar, one must generate (i) an arbiter able to handle the exact number of requests in an Xbar and (ii) wires (address bus, data bus and some control lines) between masters and slaves.

# CHAPTER II

# TERMINOLOGY FOR AN ARBITER

In this chapter, we define terms that we use in this thesis. A reader who is already very familiar with computer network switch terminology may skip this chapter.

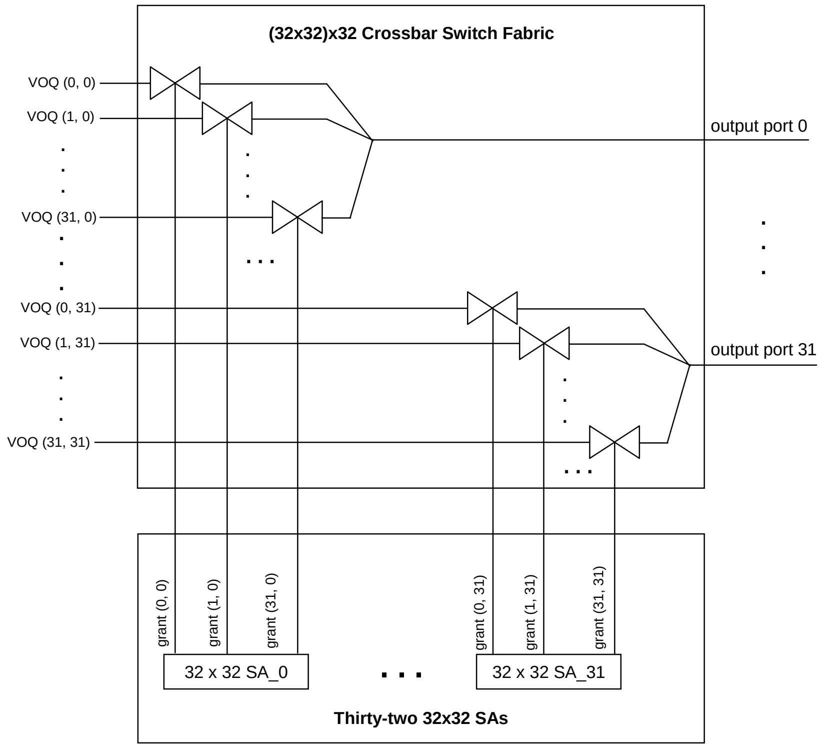


**Figure 1:** Internal structure of (32x32)x32 crossbar switch fabric and thirty-two 32x32 SAs of 32x32 network switch

We define terms to describe Figure 1. Figure 1 shows the inputs and outputs of the crossbar switch fabric in a 32x32 network switch. We briefly describe the network switch structure in order to later show how our generated arbiter is applied to a network switch. Note that we use the terms “switch” and “network switch” interchangeably throughout this thesis. Also, note that this thesis focuses only on an arbiter (equivalently, a scheduler) for a network switch or for a bus.

1. An **MxN switch** is an M-input by N-output switch. For example, a 32-input by 32-output device is a **"32x32" device**. Thus, there are 1024 ($32^2$) different possible connections where a "connection" is between a particular input port and a particular output port. A **switch** is able to pass data (packets) from any of the M inputs to any of the N outputs. A **network switch** is a switch that implements packet passing via a specific network protocol, e.g., the Internet Protocol (IP). All the switches we consider in this thesis are network switches.

2. **Virtual Output Queues (VOQs)** are typically employed in a packet switch to mitigate the head-of-line (HOL) blocking problem. HOL blocking occurs when a single FIFO input queue is used for each input port, and the packet at the head of the queue is blocked from being forwarded to its corresponding output port because of port contention, thereby blocking the entire FIFO. By using separate input queues for each input/output port pair, the HOL blocking problem can be solved [56].

3. **VOQ (m, n)**: m is the input port index, and n is the output port index. VOQ (1, 0), for example, is VOQ of input port 1 and queues packets destined to output port 0 as shown in Figure 1.

   **Example 2.1** Figure 2 describes HOL blocking problem (*Without VOQs*) and the solution of this problem (*With VOQs*). For this example, assume that *input port 1* is granted when output port contentions occur. A numbered rectangle in Figure 2 corresponds to a packet with the destination specified by the number. Thus, the packet numbered '1' indicates that this packet is destined to *output port 1.* For the *Without VOQs* case, *packet 1* in the queue at *input port 0* is blocked by *packet 0*

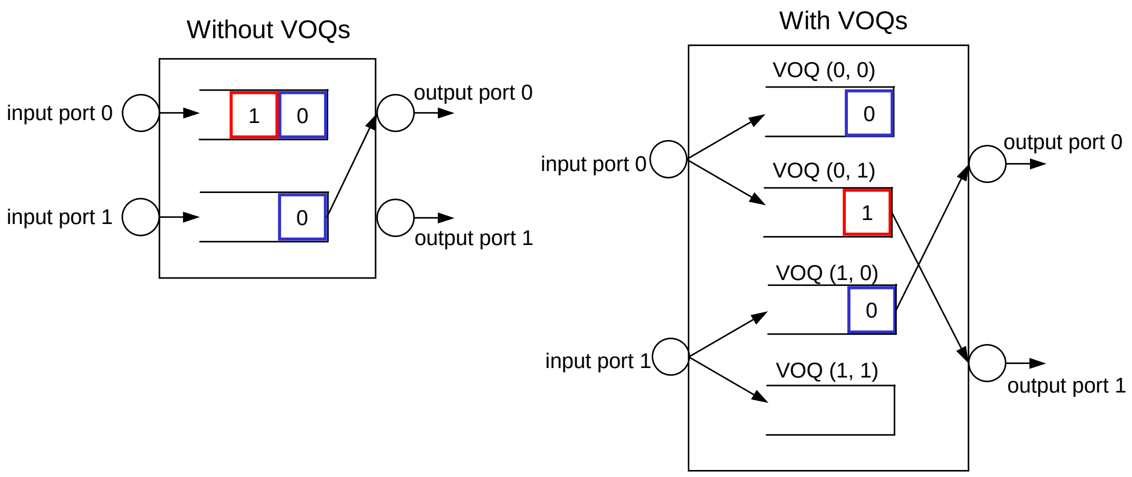


**Figure 2:** HOL blocking example: without VOQs and with VOQs

located at the head of the queue, even though *output port 1* is available at this point. Therefore, only *packet 0* is sent to *output port 1* in the current cycle. To remove HOL blocking, multiple VOQs are placed at input ports. In the *With VOQs* case, *packet 1* at *VOQ (0, 1)* is forwarded to *output port 1* simultaneously as *packet 0* at *VOQ (1, 0)* is delivered to *output port 0*. Consequently, multiple packets can be delivered to the appropriate unique destinations by employing VOQs. □

4. **(MxV)xN**: M is the number of input ports of an MxN switch. V is the number of VOQs per input port, and N is the number of output ports of an MxN switch. Note that the number of VOQs per input port (V) is typically equal to the total number of output ports (N) that can be requested from one input port - i.e., typically V equals N. The multiplicative product of M multiplied by V is the total number of VOQs in an MxN switch. As the name of Virtual Output Queue (VOQ) implies, an input port considers its V VOQs as output ports. Also, the VOQs dedicated to a certain input port have the same input port index as shown in Figure 3. For example, input port 0 as shown in Figure 3 has thirty-two VOQs with the same input port index: from VOQ (0, 0) to VOQ (0,

31). Theoretically, to completely remove the HOL block problem, each input port requires N dedicated VOQs.

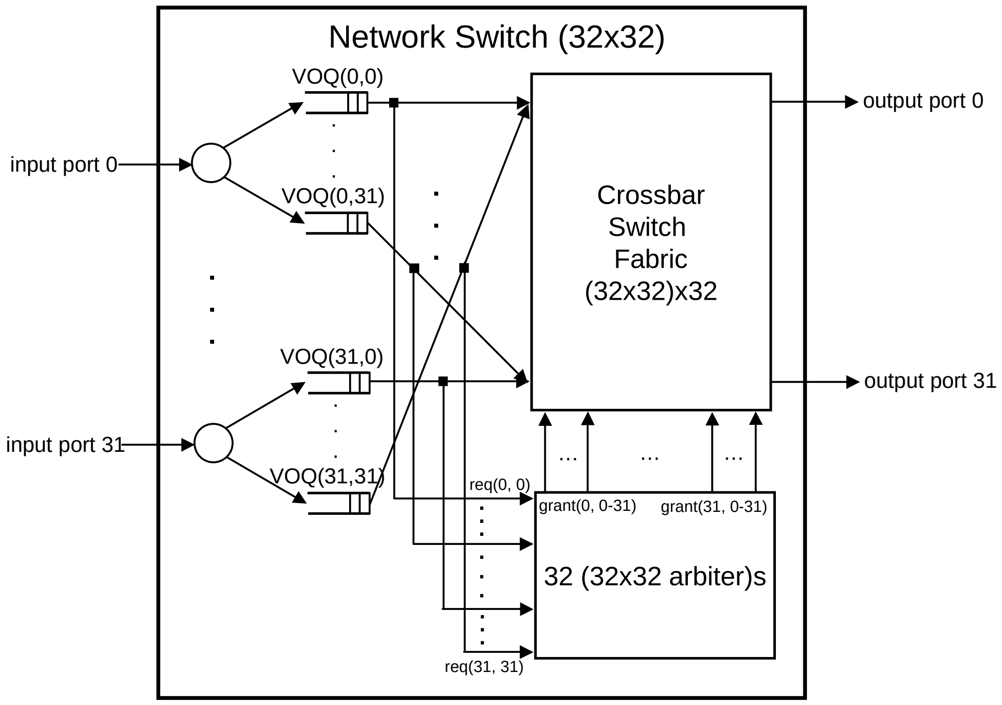


**Figure 3:** 32x32 network switch architecture

**Example 2.2** Suppose we design a 3x2 network switch. Each input port is allocated two VOQs since there are two output ports. The VOQs for *input port 0* are *VOQ (0, 0)* and *VOQ (0, 1)*; the VOQs for *input port 1* are *VOQ (1, 0)* and *VOQ (1, 1)*; and the VOQs for *input port 2* are *VOQ (2, 0)* and *VOQ (2, 1)*. Thus, the total number of VOQs for this 3x2 switch is equal to 6 ($= 3 * 2$). If we group VOQs based on the output port index as shown in Figure 1, *VOQ (0, 0)*, *VOQ (1, 0)* and *VOQ (2, 0)* are grouped together for output port 0 as shown in Figure 4. □

5. **(MxV)xN crossbar switch fabric**: There are connections between $M * V$ inputs (from VOQ (0, 0) to VOQ (M-1, V-1)) and N outputs, the number of output ports in the switch fabric. Again, VOQ ($l$, $m$) implies a VOQ at the

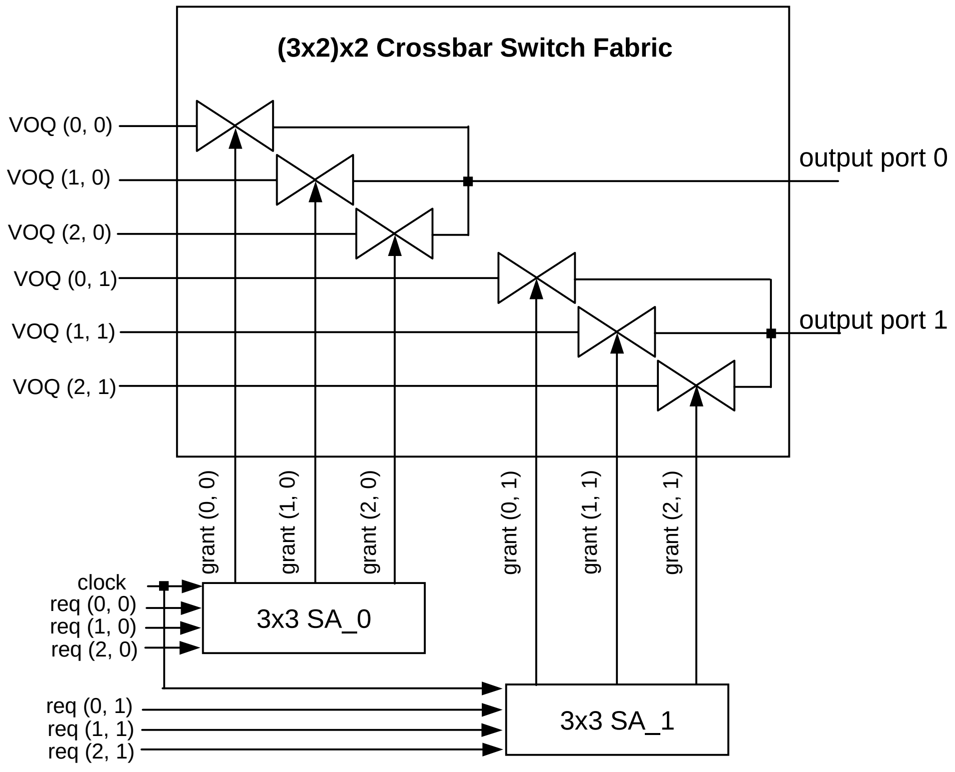


**Figure 4:** (3x2)x2 crossbar switch fabric and two 3x3 SAs (Note: reset signals for the SAs not shown)

$l^{th}$ input port destined to $m^{th}$ output port. As an example, Figure 1 shows a (32x32)x32 crossbar switch fabric[1].

6. An **MxM Switch Arbiter (SA)** is a part of an (MxV)xN switch with V=N. An MxM SA controls M specific transmission gates between M VOQs and a particular output port in an (MxN)xN switch. Thus, the number of requests is typically equal to the number of input ports, M. At most one transmission gate is turned on out of M transmission gates at a time. Hence, the number of grant signals controlling the M transmission gates to a particular output port is always equal to M. In Figure 1, for example, signal *grant (0, 31)* from

[1]The crossbar switch fabric shown in Figure 1 might be different from the switch fabric of a real network switch. We assume that control signals are required to turn on a connection between a specific VOQ and a particular output port.

*SA_31* turns on or off the transmission gate between *VOQ (0, 31)* and *output port 31*. Signals *grant (1, 31)* through *grant (31, 31)* control the other thirty-one transmission gates. The inputs of an MxM SA are *req[M-1:0]*, *clock* and *reset* signals, and the outputs of an MxM SA are *grant[M-1:0]* signals. The *clock* input is employed to rotate, in round-robin fashion, which request out of the M request signals (*req[M-1:0]*) receives the highest priority in a given clock cycle. The total number of MxM SAs needed for an (MxN)xN switch is equal to the number of output ports, N.

**Example 2.3** (Continued from Example 2.2) There are six ($= 3 * 2$) inputs (equivalently, the total number of VOQs in the switch) to the crossbar switch fabric in our 3x2 network switch. To resolve conflicts among *VOQ (0, 0)*, *VOQ (1, 0)* and *VOQ (2, 0)* in the case that all three request output port 0 (by *req (0, 0)*, *req (1, 0)* and *req (2, 0)*, respectively) in the same cycle, a 3x3 SA controls the connections as shown in Figure 4. Note that 3 ($= M$) is equal to the number of input ports of a 3x2 switch. The grant signals from the SA are concatenated as follows: *grant (2, 0)*, *grant (1, 0)*, *grant (0, 0)*. In Figure 4, if a grant signal from the *3x3 SA_0* is 3'b010, only the transmission gate between *VOQ (1, 0)* and *output port 0* is turned on. Also, to resolve conflicts among *VOQ (0, 1)*, *VOQ (1, 1)* and *VOQ (2, 1)*, another 3x3 SA (*3x3 SA_1*) is required for *output port 1* as shown in Figure 4. Thus, a total number of two 3x3 SAs are needed for this 3x2 switch. □

**Example 2.4** For a 2x3 network switch, one 2x2 SA is needed per output port. In Figure 5, *2x2 SA_0* is used to resolve contentions between *VOQ (0, 0)* and *VOQ (1, 0)* in the case that both request *output port 0* in the same cycle. Again, 2 ($= M$) is equal to the number of input ports of a 2x3 switch. □

7. An **MxM distributed SA**, equivalently an **MxM hierarchical SA**, fulfills the same role as and has exactly the same inputs and outputs as an MxM SA.

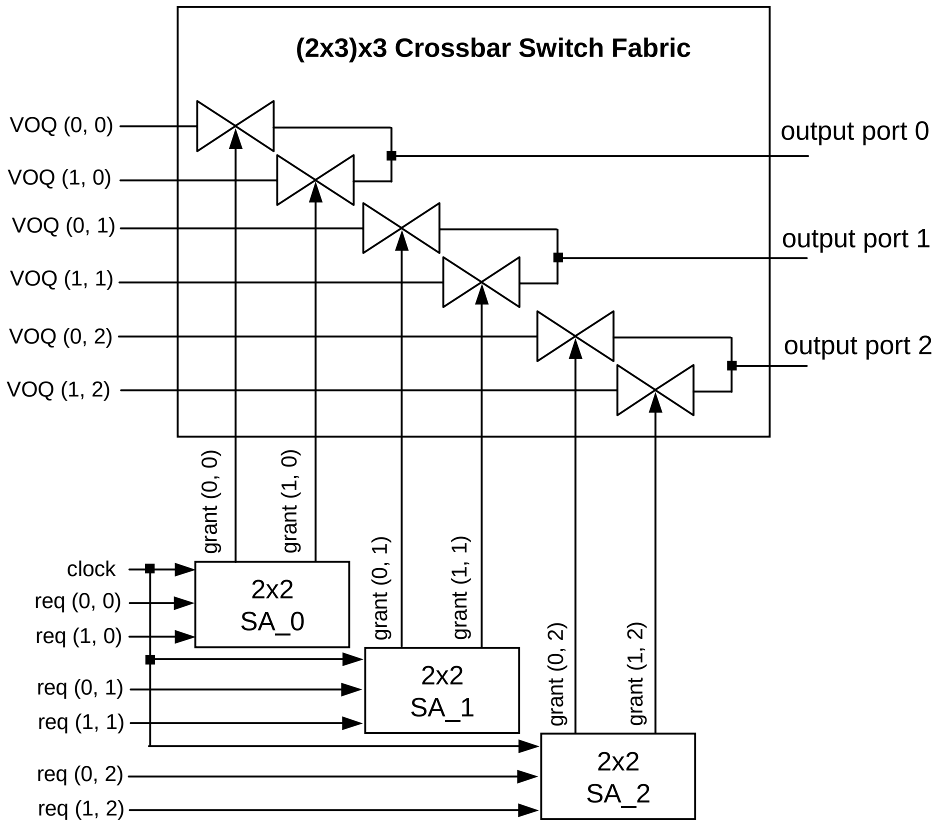


**Figure 5:** (2x3)x3 crossbar switch fabric and three 2x2 SAs (Note: reset signals for the SAs not shown)

However, an MxM hierarchical SA is composed of smaller SAs in the form of a hierarchical tree structure. We call smaller SAs switch arbiter blocks.

8. An **ack-req SA** block has an extra request ("*req*") output and an extra acknowledgment ("*ack*") input, as shown in Figure 6. We only use 2x2, 3x3 and 4x4 ack-req SA blocks in this thesis.

9. A **root SA** can be a 2x2, 3x3 or 4x4 SA. We call the 2x2 (3x3 or 4x4) SA a 2x2 (3x3 or 4x4) root SA for two reasons: (i) a root SA is built directly from logic gates (no hierarchy) as shown in Figure 7 for the 2x2 case, and (ii) a root SA is used as the "root" SA in the tree structure of an MxM hierarchical SA; this use will become more clear later in Section 4.2.

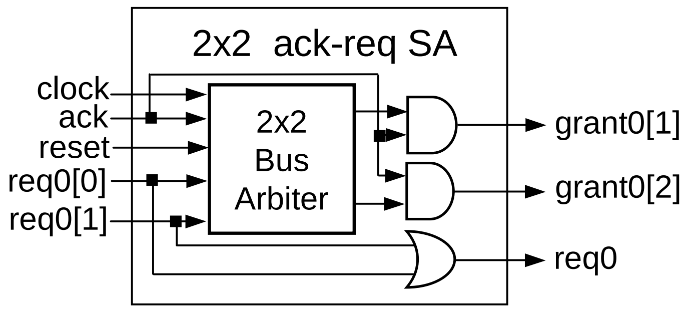


**Figure 6:** 2x2 ack-req SA block

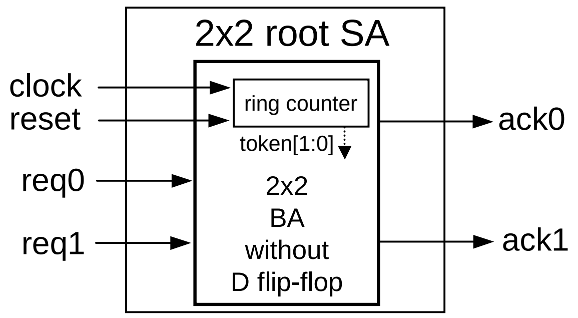


**Figure 7:** 2x2 root SA

10. We use a **switch arbiter block** as a superset of an ack-req SA, ack-req BA and a root SA.

11. A **Bus Arbiter (BA)** resolves bus conflicts when multiple bus masters request a bus in the same cycle. A BA allows access to a bus for the bus master whose request is granted. The input/output logic function of a BA and an ack-req SA block are the same except that an ack-req SA block has an extra "request" output. The use of this extra "request" output will become clear later in Section 4.2. The main difference between a BA and an ack-req SA block is in typical use: a BA typically arbitrates buses, while an ack-req SA block is a component of an MxM hierarchical SA that typically resolves conflicts between input ports and output ports in a switch.

12. A **hierarchical BA** is nearly the same as a hierarchical SA. The only difference is that a hierarchical BA has an extra *ack* input, taking the *ack* input from the owner of the bus (the bus master). In short, while a hierarchical SA rotates priorities every clock cycle, a hierarchical BA rotates priorities (which include rotating the highest priority among the potential bus requestors) only after a rising edge seen on the *ack* input. Thus, the bus requestor given control over the bus may use the bus for multiple bus clock cycles.

13. An **ack-req BA**, a component of a hierarchical BA, has the same functionality as an ack-req SA. However, the logic of an ack-req BA has extra gates to guarantee the possession of multiple bus clock cycles to the granted bus master.

In addition to an (MxV)xN crossbar switch fabric, the internal structure of an MxN network switch consists of VOQs and arbiters (there may be additional hardware components such as memory at the input port in case of the occurrence of VOQ overflows). In Figure 1, we intentionally delete request connections to the 32x32 Switch Arbiters (SAs) from VOQs to present a more compact and easy-to-read diagram. In Figure 9in Chapter 3, however, we show request connections to the SAs in more detail.

# CHAPTER III

# RELATED WORK

In this chapter, we present some of previous work in arbiter design and crossbar switch customization. For the arbiter design, we emphasize the Ping-Pong Arbiter (PPA) [7] and the Programmable Priority Encoder (PPE) [13] which implements the iSLIP algorithm [29]. The iSLIP algorithm is arguably the best current state-of-the-art arbitration algorithm in computer network theory and, from the author's informal search of the literature, seems to be the most referenced arbitration algorithm in recent network switch papers, including [7, 8, 66] . In Section 3.2, we focus on Smart Memory which is the closest related work in terms of the customized Xbar.

## *3.1 Arbiters*

### 3.1.1 Arbitration for Network Packet Switching: PPA, PPE and others

Current designs in Network-on-Chip (NoC) typically use standard round-robin token passing schemes for bus arbitration [11]. In computer network packet switching, previous research in round-robin algorithms have reported results on an iterative round-robin algorithm (iSLIP) [29] and a dual round-robin matching (DRRM) algorithm [8]. Furthermore, Chao *et al.* describe a design of a round-robin arbiter for a packet switch [7]. Chao *et al.* refer to their hardware design as a Ping Pong Arbiter (PPA).

In general, the goal of a switch arbiter in a packet switch is to provide control signals to the crossbar switch fabric as shown in Figure 3. In a packet switch design, one must keep in mind that each input port can potentially request connections to all output ports (e.g., in the case of broadcast). Theoretically, to avoid the HOL blocking problem in a packet switch with M input ports and N output ports, each input is allocated N VOQs (one per output) for a total of the multiplicative product of M times N VOQs in the packet switch [27, 56]. By employing VOQs, it is reported that the throughput of an input-queued switch increases from 58.6% to 100% [27, 30]. In general, an MxN switch can have fewer VOQs than N per input port to save cost and area at some slight cost of occasional HOL blocking. However, we assume $V = N$ VOQs per input port in an MxN switch in this thesis.

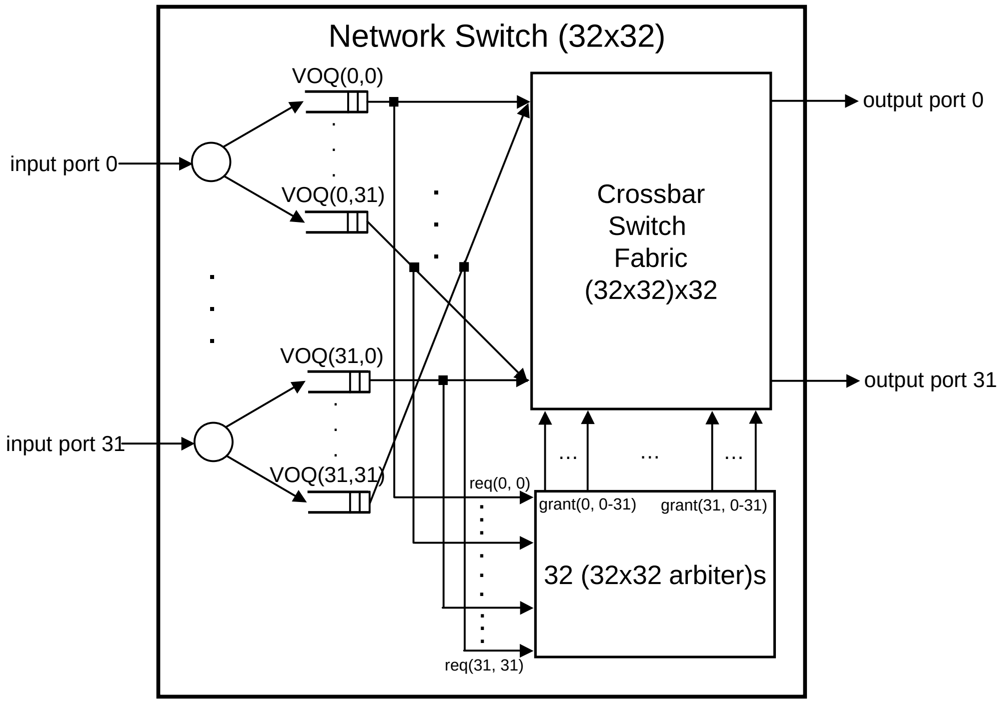


**Figure 8:** 32x32 network switch architecture (Note: Figure 8 is exactly same as Figure 3)

Figure 3 is repeated here as Figure 8: Figure 8 shows a 32x32 network switch with 32 input ports and 32 output ports. Each input port can request between zero (none) and thirty-two (all) connections to output ports. To accomplish this, thirty-two 32x32 Switch Arbiters (SAs), as shown in the bottom right hand side of Figure 3, take as input $32^2$ requests (*req (0, 0)*, *req (0, 1)*, . . . , *req (31, 30)*, *req (31, 31)* – 32 requests per input port, or one request per VOQ) and translates those requests into

$32^2$ (=4096) grant signals (one grant signal per possible VOQ to output connection) where at most one grant signal per output port is set to '1' on each clock cycle (thus, of the $32^2$ grant signals, at most 32 are set to '1' each clock cycle).

Figure 9 shows one 32x32 SA out of the thirty-two 32x32 SAs in Figure 3. Each SA grants one request out of at most thirty-two requests from thirty-two VOQs. Each input of the 32x32 SA in Figure 9 is connected to a specific VOQ (one per input port) which may request *output port 0*. The thirty-two outputs of the 32x32 SA are grant signals indicating which of the 32 VOQs is granted *output port 0* (note that if no VOQ requests the output port, then all grant signals will be '0' in this case). For example, *grant (31, 0)* can signal the crossbar switch fabric in Figure 3 to connect *VOQ (31, 0)* to *output port 0*.

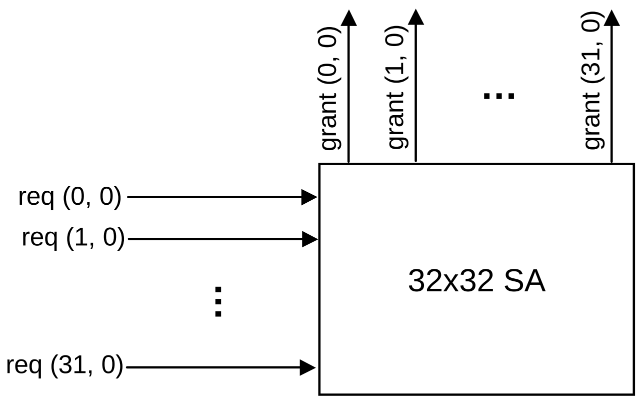


**Figure 9:** 32x32 Switch Arbiter (SA)

### *3.1.1.1 Network Packet Switching Arbitration: PPE*

The iSLIP algorithm [29] uses in its implementation MxM SAs. The iSLIP algorithm focuses on the efficient and fast scheduling of *best-effort* traffic and is developed to satisfy the following properties:

- High throughput for a network switch: an algorithm should keep the backlog low in the VOQs.

- Starvation-free VOQs: a nonempty VOQ should not remain unserved indefinitely.

- Fast arbitration scheme: to achieve a high speed network switch, an arbitration algorithm should not be a performance bottleneck.

- Simplicity of implementation: an arbiter plus some network switch components (e.g., VOQs and crossbar switch fabric) are preferably implemented in a single chip.

The iSLIP authors implement an MxM SA in hardware that they call a Programmable Priority Encoder (PPE) [13]. PPE was used in the Tiny Tera prototype [28]. Tiny Tera employs an input queued 32-port crossbar switch and a centralized scheduler, PPE. PPE is composed of an M-input simple Priority Encoder [64] and a thermometer encoding logic [13] which rotate priority levels of inputs. The thermometer encoding logic eliminates the long critical path resulting from the programmable priority level. A $\log_2$M-bit-wide vector **x** is translated into an M-bit-wide vector **y** by a thermometer encoding according to the following equation.

$$y[i] = 1 \textit{ iff } i < \textit{value(x) for all } 0 < i < M\text{, otherwise } y[i] = 0$$

Since the area and delay of an M-input priority encoder rapidly increases as M increases, the authors of PPE propose a recursive decomposition of PPE for large M. An M-input PPE can be decomposed into two M/2-input subblocks. One subblock takes care of inputs 0 through (M/2)-1, while the other serves inputs M/2 through and M-1. This idea can be extended to further decompose an (M/2)-input subblock into two (M/4)-input subblocks. Each decomposition adds one stage of multiplexers to combine final output. However, the authors do not show their detailed design nor area and delay comparison in [13]. Please note that after [13], subsequent papers from the authors of [13] do not have a logic diagram of PPE. Instead, subsequent publications of the authors of [13] focus on routing and packet classification algorithms [12, 14, 16, 17, 21, 36].

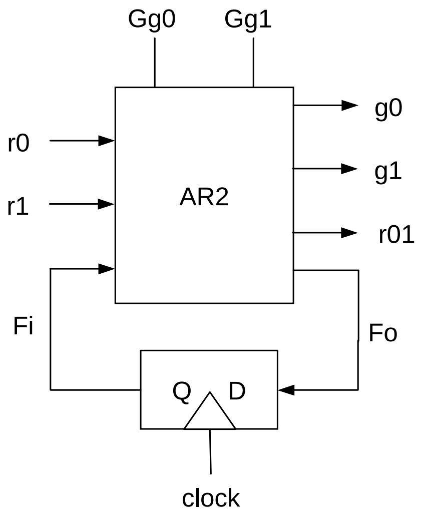
#### *3.1.1.2 Network Packet Switching Arbitration: PPA*

Chao *et al.* observe that a traditional arbiter (centralized arbiter) handles all request inputs together with arbitration delay increasing proportionally to the number of requests [7]. Consequently, a fixed amount of allowable arbitration delay limits the network switch capacity for large M. Therefore, in PPA, inputs are divided into groups and each group has its own arbiter. An arbiter in a group is named as AR2. Figure 10 shows the block diagram of *AR2* and its internal logic. *AR2* handles two request inputs (*r0* and *r1*) with two external grant inputs (*Gg0* and *Gg1*). The outputs of *AR2* are two grant signals (*g0* and *g1*), a request output (*r01*) and a flag signal (*Fo*) which is fed back to AR2 (*Fi*) via the *D flip flop*.

The request signals of each AR2 are summed as a group request signal by use of an OR gate (*r01* in Figure 10). PPA is binary tree structured as shown in Figure 11 and is composed of *AR2* in each layer of hierarchy.

Under the assumption of $M = 2^k$, Figure 11 describes a $k$-layer complete binary tree with each group handling two requests. In an *AR2*, an internal feedback signal indicates which input has a higher priority in the current cycle. If an input is granted in the current cycle, the other input has a higher priority in the next cycle. The priority is indicated by 1-bit flag; if flag is '0', input 0 of *AR2* (left input of AR2) in Figure 11 has a higher priority. There are $2^{k-1}$ *leaf AR2*s. The arbiter at the highest layer is called the *root AR2*. Other arbiters placed other than the first and the last layers are called *intermediate AR2*s.

The grant signal from an *AR2* in a location other than the lowest layer is fed back to the corresponding *AR2* located at the lower layer. Thus, each intermediate and leaf *AR2* has external grant inputs from an upper layer (*Gg0* and *Gg1* in Figure 10)



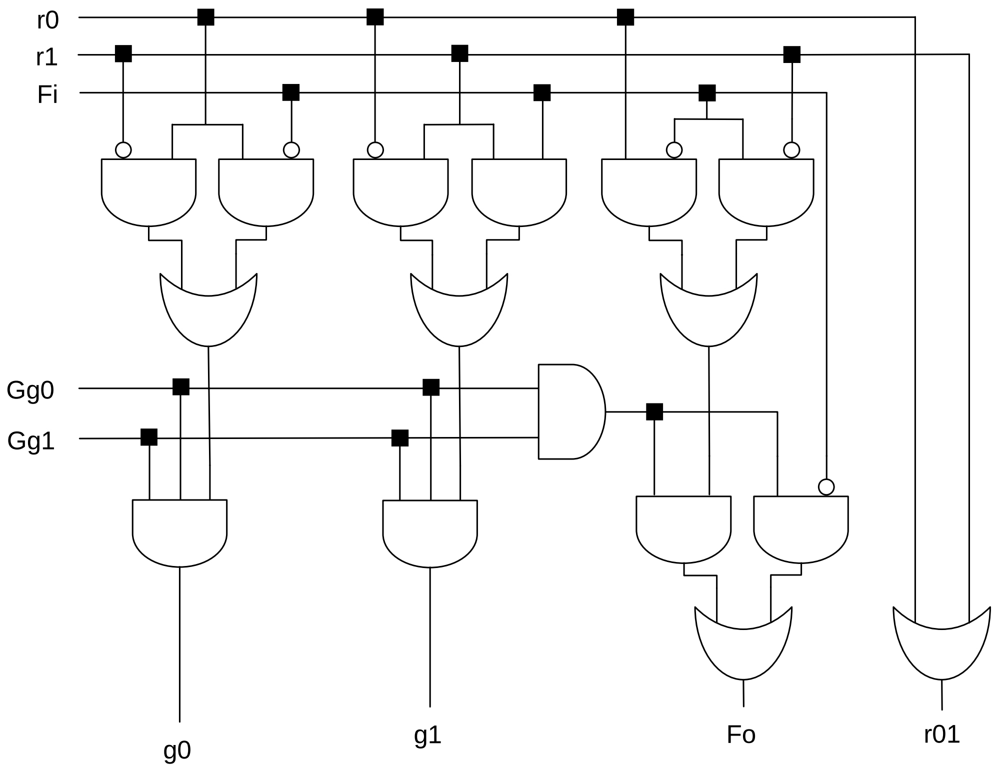


**Figure 10:** AR2: 2-input PPA and its internal logic

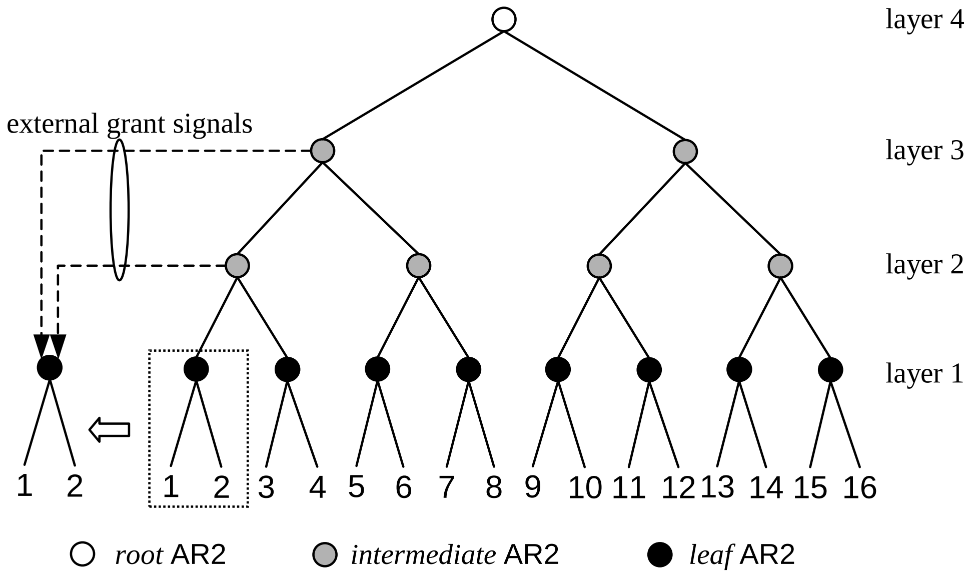


**Figure 11:** A binary tree structured PPA

which is ANDed with an *AR2*'s internal grant signal to reflect the arbitration result of an upper layer. The other important usage of an external grant is for a flag update. If an AR2 receives a valid external grant, the AR2's group request is granted from upper layers. Thus, the flag of the AR2 must be updated (toggled). However, the flag should be unchanged if the external grant is invalid to maintain the current priority.

The PPA structure shown in Figure 11 is for the case of $k = 4$. If all inputs request a grant, request inputs are granted in the order of 1→3→5→7→9→11→13→15→2→4 and so on which is in round-robin fashion.

Additional details of PPE and PPA will be described in Chapter 4 in the context of comparing PPE and PPA with our approach.

### 3.1.2 Logic Synthesis

Logic synthesis transforms a logic level description of a digital circuit into a gate level specification [3]. For large varieties of logic designs, logic synthesis provides fast design of logic at area and performance suitable for many Application-Specific Integrated Circuit (ASIC) designs [3, 15, 32]. For some classes of designs, such as multipliers and dividers, chip designers have devised custom logic structures that are much faster with much lower area, when compared to what a logic synthesis tool can deliver given the boolean description of the design.

The author could not find any prior research in logic synthesis focusing on arbitration logic or using token-passing in a multi-level logic network. *Thus, while this thesis only presents a small focused tool able to generate custom arbitration logic and is not a general-purpose logic synthesis tool, nevertheless no prior work on logic synthesis known to the author focuses on synthesis of specialized arbitration logic using token-passing.*

### 3.1.3 Token Rings

Since the 1970s, Local Area Networks (LANs) [24, 48] have been developed using a token ring networks. In a token ring, there are many stations which are connected by point-to-point links in a ring topology. Transmission links in a ring topology shared by many stations require Medium Access Control (MAC) which coordinates access to the shared medium in order to prevent collisions. Only the station which has possession of the token has the privilege to transmit at any given time. A "token," the key idea used in a token ring, is used for the token ring MAC and travels around the ring-topology network.

We apply the "token" concept to our hierarchical BA and SA designs in order to rotate the priority level among input request signals. The token enables a particular logic block in our design, which specifying the priority order in request signals. The details of exactly how we use a "token" are discussed in Chapter 4.

## 3.2 On-chip Communication

Considering multiple paths for on-chip communication, Karim *et al.* propose the "Octagon" bus which provides multiple on-chip communication channels for a multiprocessor SoC [19]. The octagon bus can support up to eight PEs and is scalable by sharing one port with another octagon bus as shown in Figure 12. There are bi-directional channels between ports, and each port has three queues: one for the route to cross, another for the route to the left and a third for the route to the right. Utilizing three queues is analogous to the VOQ concept in order to remove the HOL blocking problem. For applications whose communication patterns map closely to the Octagon bus structure, the Octagon bus can be an attractive and lower area alternative when compared to a full crossbar switch network.

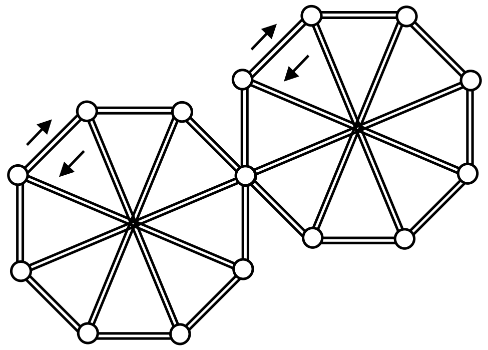


**Figure 12:** Octagon bus

There are few approaches to reduce design time for an SoC crossbar switch. Mai *et al.* propose reconfigurable crossbar switch and memory blocks [25]. In [25], a processing tile consists of a processor, crossbar interconnect and sixteen 8Kbytes SRAM blocks as shown in Figure 13. The processor is composed of two integer clusters and a floating point cluster. The processor block is connected to sixteen SRAMs via a crossbar switch such that a different number of SRAMs can be remapped to two integer clusters and a floating point cluster depending on an application. A *quad* has four processing tiles, and each processing tile communicates via a quad network. There are many 2.5 *mm* x 2.5 *mm* processing tiles with 0.1$\mu$m CMOS technology. However, the authors do not give details about their crossbar switch design.

Dally *et al.* propose a Network-on-Chip (NoC) in [11] which consists of network logic and tiles (similar to Mai *et al.* [28]). Dally *et al.* propose a tile structured NoC:

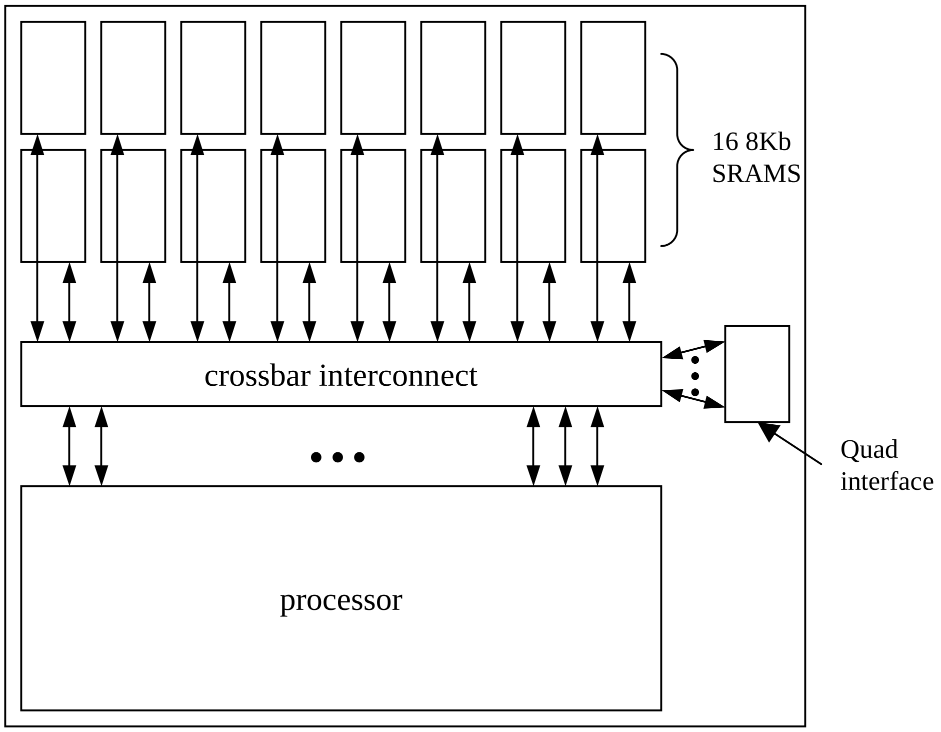


**Figure 13:** A processing tile with crossbar interconnect

sixteen 3 *mm* x 3 *mm* tiles in a 12 *mm* x 12 *mm* chip using 0.1$\mu$m CMOS technology with a 0.5$\mu$m minimum wire pitch. A tile can have client logic such as processors, DSPs, peripheral controllers and memory subsystems. With no top level connections other than the network logic, tiles communicate to one another by sending packets over the network logic. Each tile has one input controller at the west edge of a tile and four output controllers: one for the other directions (North, East and South) and one for the tile. The area overhead resulting from the on-chip network is estimated to be 6.6%. The arbitration scheme for a router is not explicitly mentioned in [11].

All literature discussed above present approaches to efficiently communicate on-chip. Compared with [25], which appears to be designed by hand, our MxN crossbar (Xbar) switch is automatically generated with bus parameters specified by a user to support an exact number of masters and an exact number of slaves. Also, our generated Verilog code for an Xbar is synthesizable, resulting in a reduction of design time. Thus, from the above discussion, an Xbar switch Generator (X-Gt: the subset of DX-Gt described in Chapter 6) provides the first automated approach to Xbar switch generation.

In this chapter we showed previous arbiter designs and approaches to on-chip communication with multiple channels from Karim *et al.* [19] and from Mai *et al.* [25]. In Chapter 4, we will show how our arbiter design is different from others, especially PPE and PPA. We will present how our arbiter is automatically generated in Chapter 5. In Chapter 6, we will describe how our Xbar is customized according to customer specifications.

# CHAPTER IV

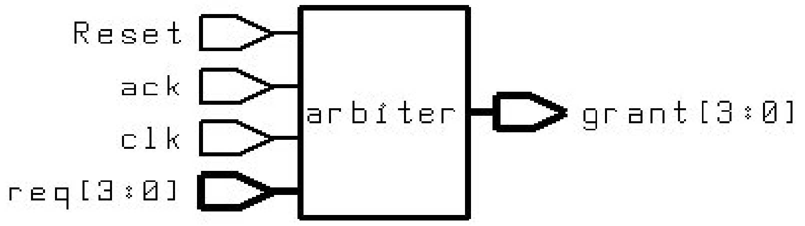
# ROUND-ROBIN ARBITER DESIGN

A round-robin token passing bus or switch arbiter guarantees avoidance of starvation among masters and allows any unused time slot to be allocated to a master whose round-robin turn is later but who is ready now. The protocol of a round-robin token passing bus or switch arbiter works as follows. In each cycle, one of the masters (in round-robin order) has the highest priority (i.e., owns the token) for access to a shared resource. If the token-holding master does not need the resource in this cycle, the master with the next highest priority who sends a request can be granted the resource. At the end of the cycle, the highest priority master then passes (in round-robin order) the token to the next master which then will have the highest priority for the next time slot.

Section 4.1 (the next section) shows the design of 2x2, 3x3 and 4x4 Bus Arbiters (BAs) generated by our tool. In Section 4.2, we present a sample design of a 32x32 hierarchical SA generated by our tool. In Section 4.3, we describe how a hierarchical SA is modified to become a hierarchical BA. We discuss how our arbiter design approach impacts on logic synthesis in Section 4.4. Finally, we discuss the fairness issue in arbitration in Section 4.5. MxM hierarchical SAs and BAs generated by our Round-robin Arbiter Generator (RAG) have a hierarchical structure for values of M greater than four.

## *4.1 2x2, 3x3 and 4x4 Bus Arbiter Design*

Figure 14 show a BA generated to handle four requests. The top figure of Figure 14 shows a BA block diagram for bus arbitration among four masters. To generate a



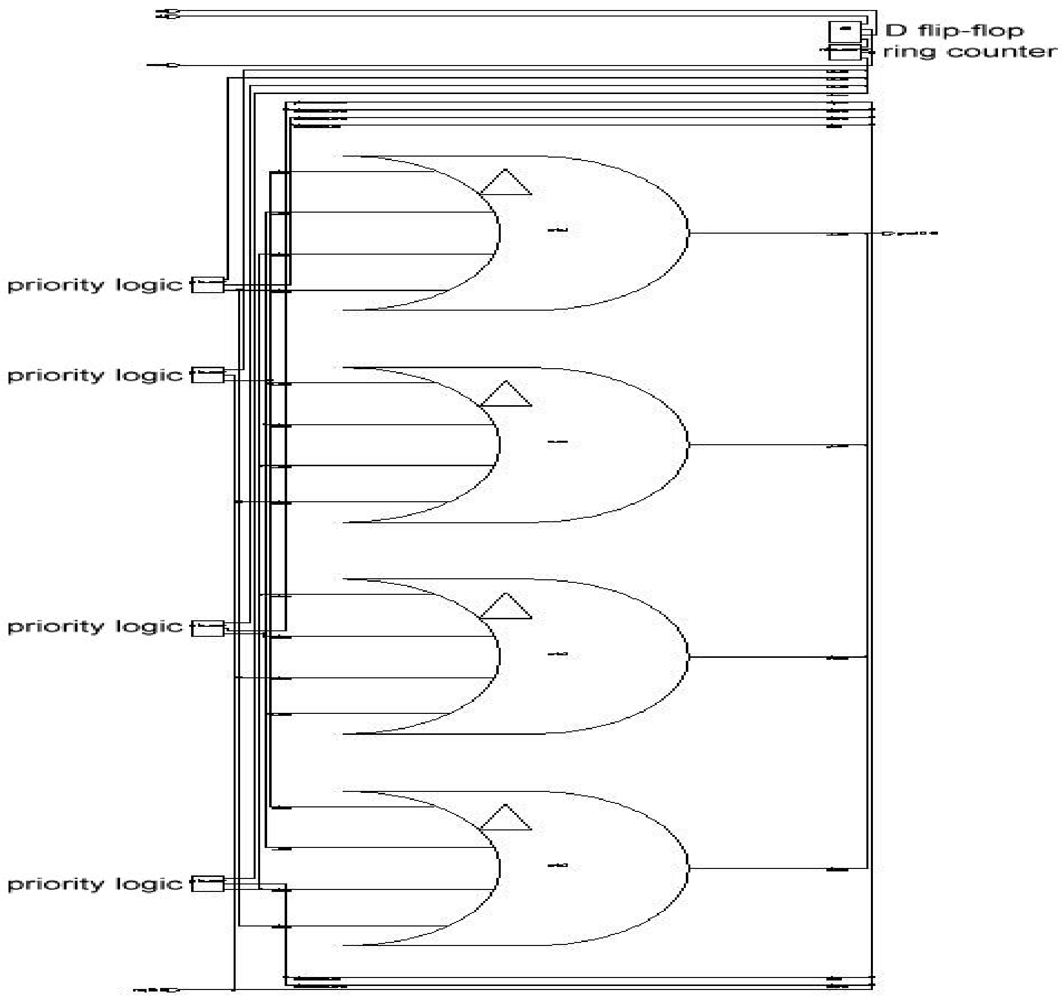


**Figure 14:** Block diagram and logic diagram of a 4x4 Bus Arbiter

hierarchical BA, RAG takes as input the number of masters and produces synthesizable Register Transfer Level (RTL) Verilog code. For synthesis of the logic for a four-master BA, the bottom of Figure 14 shows the logic synthesized by the Synopsys Design Compiler [51] with a TSMC 0.25$\mu$m standard cell library [60] from LEDA Systems [23] (now Qualcore Logic).

The four small blocks on the left side of the logic diagram in Figure 14 are the four "priority logic" blocks in Figure 15, and the block on the top right side of the logic diagram in Figure 14 is a ring counter. The functionality of a priority logic block is the same as that of a priority encoder [64] without output encoding; please see Table 1 for a truth table for a priority logic block for a 4x4 case.

The BA consists of a D flip-fop, priority logic blocks, an M-bit ring counter and M M-input OR gates as shown in Figure 15 where M=4. A 4x4 priority logic block is

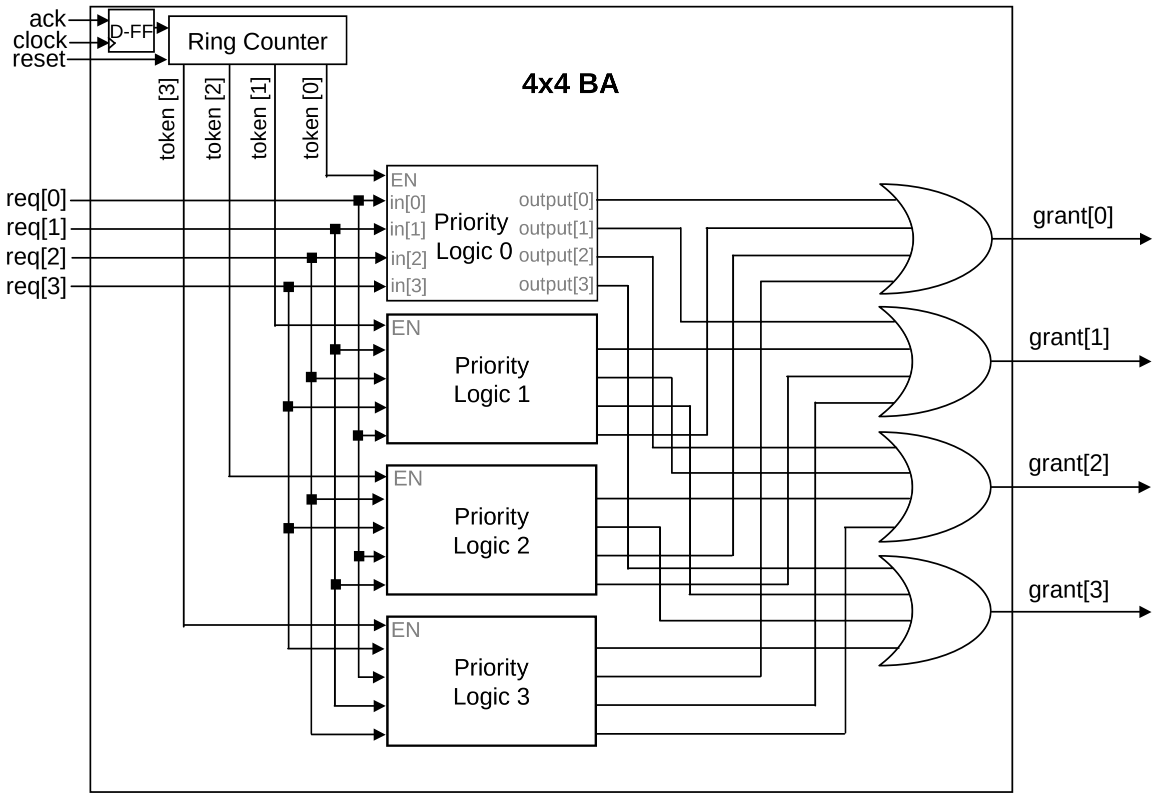


**Figure 15:** 4x4 Bus Arbiter (BA) architecture

**Table 1:** Truth table of a 4x4 priority logic block

| EN | in[0] | in[1] | in[2] | in[3] | output[0] | output[1] | output[2] | output[3] |
|---|---|---|---|---|---|---|---|---|
| 0 | X | X | X | X | 0 | 0 | 0 | 0 |
| 1 | 1 | X | X | X | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

implemented incombinational logic implementing the logic function of Table 1. The priority of inputs is placed in descending order from *in[0]* to *in[3]* in the priority logic blocks (*Priority Logic 0* through *3*) shown in Figure 15. Thus, *in[0]* has the highest priority, *in[1]* has the next priority, and so on. To implement a BA, we employ the token concept from a token ring in a network. The possession of the token allows a priority logic block to be enabled. Since each priority logic block has a different order of inputs (request signals), the priority of request signals varies with the chosen priority logic block. The token is implemented in a 4-bit ring counter as shown in Figure 15.

**Example 4.1** When token = 4'b0100, processor 2 (*req[2]*) has the token and thus has the highest priority in this arbitration cycle. In other words, *Priority Logic 2* in Figure 15 is enabled and *req[2]* has the highest priority because *req[2]* is connected to *in[0]* of the priority logic block, *Priority Logic 2.* □

The outputs (four bits) of the ring counter act as the enable signals to the priority logic blocks. Thus, only the single enabled priority logic block can assert a grant signal. The *ack* signal to the bus arbiter is clocked by a D flip-flop as shown in Figure 15. The *ack* signal pulls a trigger to the ring counter so that the content of the ring counter is rotated one bit for the next arbitration cycle. Thus, the token bit is rotated left each cycle with 4'b1000 rotating to 4'b0001 in Figure 15, and the token is initialized to one at the reset phase (e.g., 4'b0001 for a four-bit ring counter) so that there is exactly one '1' output by the ring counter. In the round-robin algorithm, each master must wait no longer than $(M-1)$ time slots, where a time slot is the period of time allocated to the chosen master, until the next time it receives the token (i.e., highest priority). The assigned time slot can also be yielded to another master if the owner of the time slot has nothing to send [47]. This round-robin protocol guarantees a dynamic priority assignment to bus masters (requestors) without starvation.

In Figure 15, request inputs are connected with different levels of priority so that the priority levels are equally distributed over all request signals. In other words, the probability of being the highest priority is 0.25 for all request signals (*req[0]* – *req[3]*). Likewise the probability of being the second highest priority is also 0.25, and so on.

**Example 4.2** In Figure 15, *req[0]* has the highest priority in the top priority logic block (*Priority Logic 0*) and has the lowest priority in the next priority logic block (*Priority Logic 1*). Also, *req[1]* has the second highest priority in the top priority logic block and has the highest priority in the next priority logic block. The outputs of priority logic blocks are ORed together in the same order of the request inputs to priority logic blocks. In other words, for example, *grant[0]* is the output of a 4-input OR gate whose four inputs are *output[0]* (corresponding to *req[0]* input to *in[0]* of *Priority Logic 0*), *output[3]* (corresponding to *req[0]* input to *in[3]* of *Priority Logic 1*), *output[2]* (corresponding to *req[0]* input to *in[2]* of *Priority Logic 2*) and *output[1]* (corresponding to *req[0]* input to *in[1]* of *Priority Logic 3*). □

**Example 4.3** Consider a scenario with four processors as bus masters connected to the same bus with one large shared memory as a slave on the bus as shown in Figure 16. Suppose the token is 4 (token = 4'b0100, which means processor 2 has the token), and only processor 0 (which uses *req[0]*) and processor 1 (*req[1]*) want to access the memory at this cycle. Token=4'b0100 enables only *Priority Logic 2* in Figure 15. In *Priority Logic 2*, the connection to *in[0]* (*req[2]* from processor 2) indicates the highest priority. Since *req[3]*

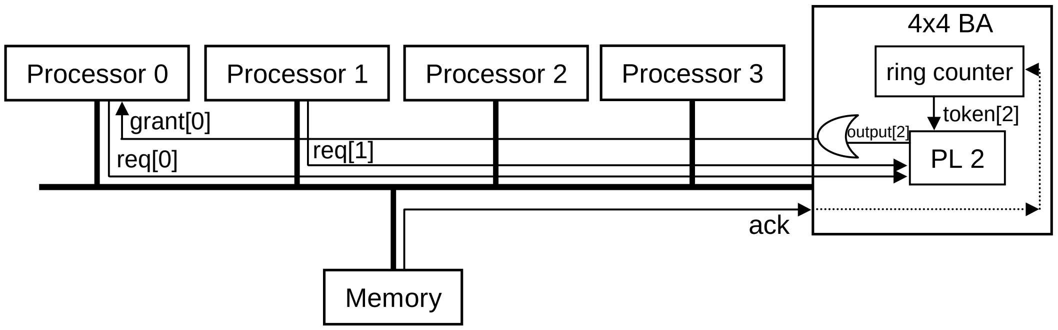
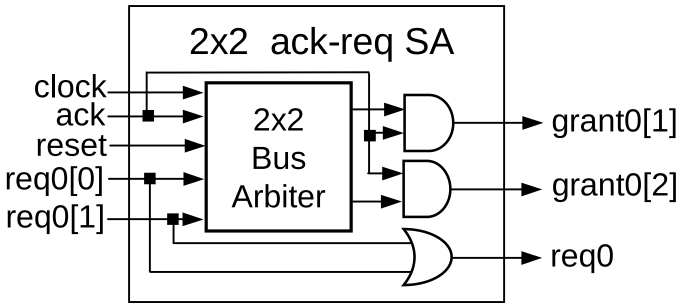


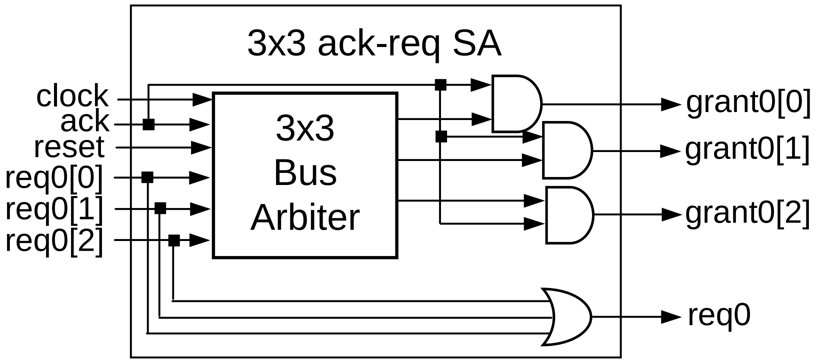
**Figure 16:** Four processors with a shared memory system (Note: bus and 4x4 BA details shown only as needed for Example 4.1.)

is connected to *in[1]* of *Priority Logic 2* in Figure 15, processor 3 has the next highest priority. However, since neither processor 2 nor processor 3 makes a request, *in[2]* which is connected to *req[0]* is next in line in priority. Thus, processor 0 is granted access to the memory. After processor 0 finishes, the memory controller of the accessed memory sends an *ack* signal, whose connection to the BA is shown in Figure 15, indicating when the memory transaction is successfully completed. Next, which could be several processor clock cycles later, the token is passed to processor 3 (the 4-bit ring counter is rotated left when the *ack* signal is received) in which case the token is 4'b1000. Note that only the asserted signals of this example are shown in Figure 16. □

## 4.2 Switch Arbiter Design

We use 2x2, 3x3 and 4x4 switch arbiter blocks as basic modules to implement an MxM hierarchical Switch Arbiter (SA). Figure 17 shows how 2x2, 3x3 and 4x4 bus arbiters are modified to implement 2x2, 3x3 and 4x4 ack-req SAs by adding some AND and OR gates to a BA (note that the *2x2 ack-req SA* in Figure 17 is exactly the same as Figure 6). In Figure 17, request input signals are ORed together to generate a single request output (whose use will become clear later), and grant signals are ANDed





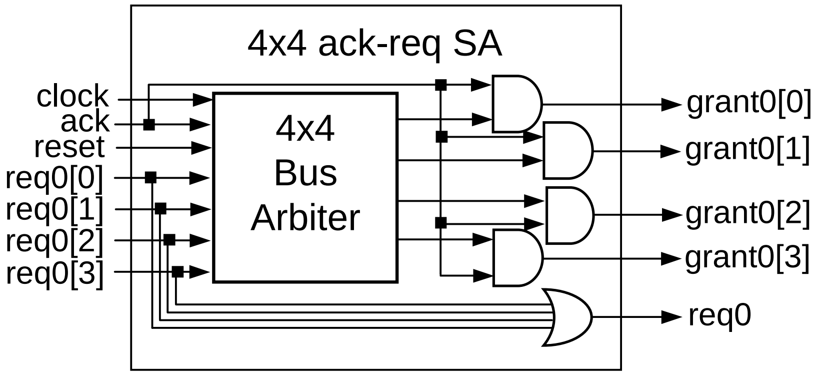


**Figure 17:** Ack-req SA blocks

together with an *ack* input (active high) which indicates that the corresponding ack-req SA is acknowledged in order to be enabled. Finally, the ack-req SA grants a master only if the *ack* input is asserted.

A root SA is placed at the top level of the hierarchy in a hierarchical SA. Specifically, a root SA is placed on the top of the hierarchical tree structure (an example of which will be shown in Figure 20). A root SA has no *ack* input nor *req* output. The input/output logic of a 2x2 root SA, a 3x3 root SA and a 4x4 root SA are the same as that of a 2x2 BA and a 4x4 BA except that there is no *ack* input and no D flip-flop in front of the ring counter as shown in Figure 18: thus, the *clock* input is used to rotate the content (the token bits) of the ring counter as shown in detail in Figure 19 for the case of a 4x4 root SA.

With the 2x2, 3x3 and 4x4 switch arbiter blocks shown in Figures 17 and 18, we can design an MxM hierarchical SA. Figure 20 shows how a 7x7 hierarchical SA is implemented and demonstrates a tree structure rotated clockwise by 90 degree. We call the 4x4 ack-req SA placed on the left side a "leaf" arbiter, and the final switch arbiter placed on the right side we call the "root" SA. We consider the "level" of a switch arbiter block to increase or go up moving toward the right. Thus, a "leaf" arbiter is always located at the lowest level. As shown in Figure 20, *req_0* and *req_1* signals act as request inputs to a switch arbiter block at the next highest level (in the case of Figure 20, the *2x2 root SA* is the next highest level). The grant signals output by the higher level switch arbiter block(s) – in Figure 20, the *2x2 root SA* is the only case – are used as inputs to the ack signals for lower level switch arbiter blocks – in Figure 20, *ack0[0]* and *ack0[1]*. Intuitively, the higher-level switch arbiter blocks activate, in round-robin fashion, lower-level switch arbiter blocks.

We can redraw Figure 20 with 4x4 BAs by placing AND gates and OR gates as shown in Figure 21 (the AND gates and OR gates shown explicitly in Figure 21 were previously contained in the 4x4 ack-req SAs in Figure 20). Note in Figure 21 that

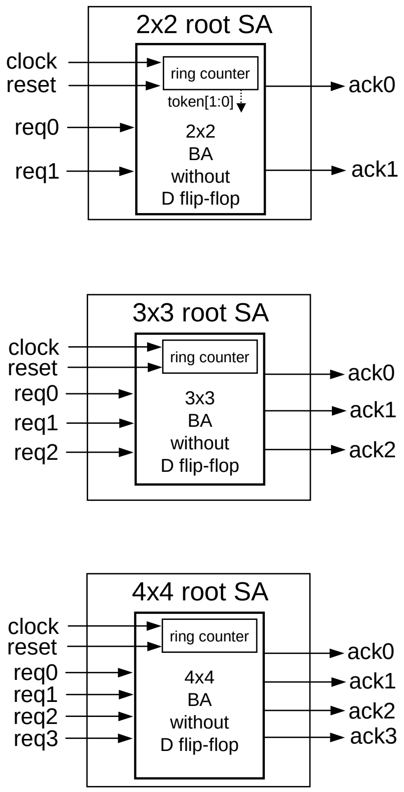


**Figure 18:** Root SA blocks

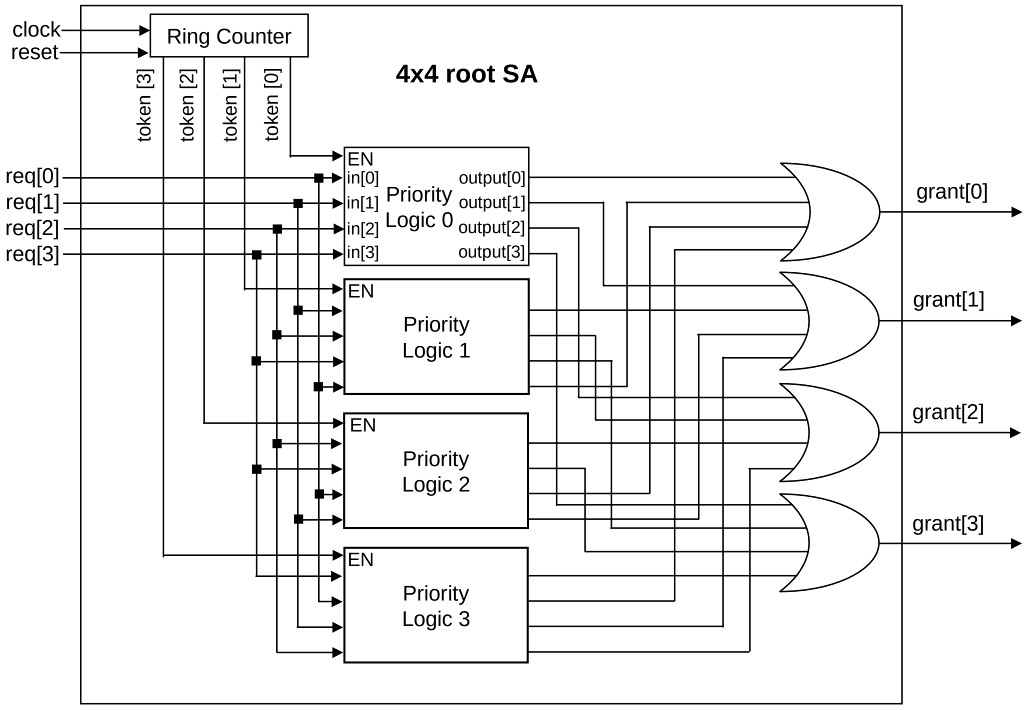


**Figure 19:** Detailed view of a 4x4 root Switch Arbiter

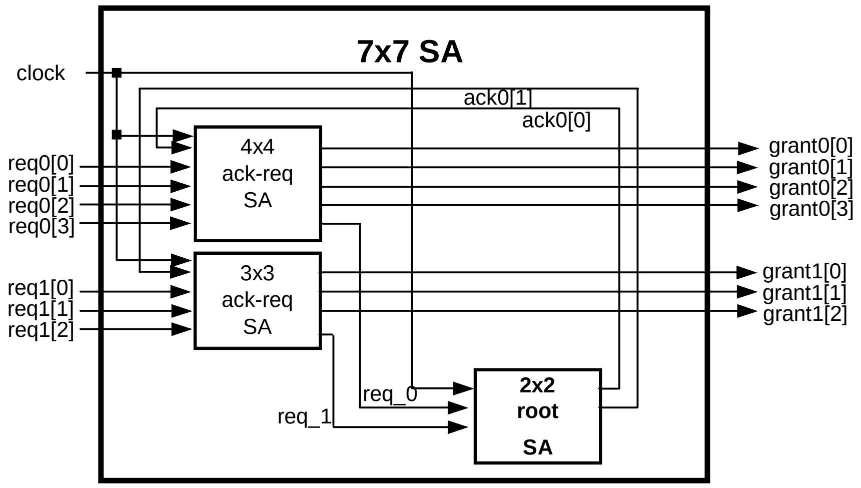


**Figure 20:** A 7x7 SA configuration (Note: reset signal not shown)

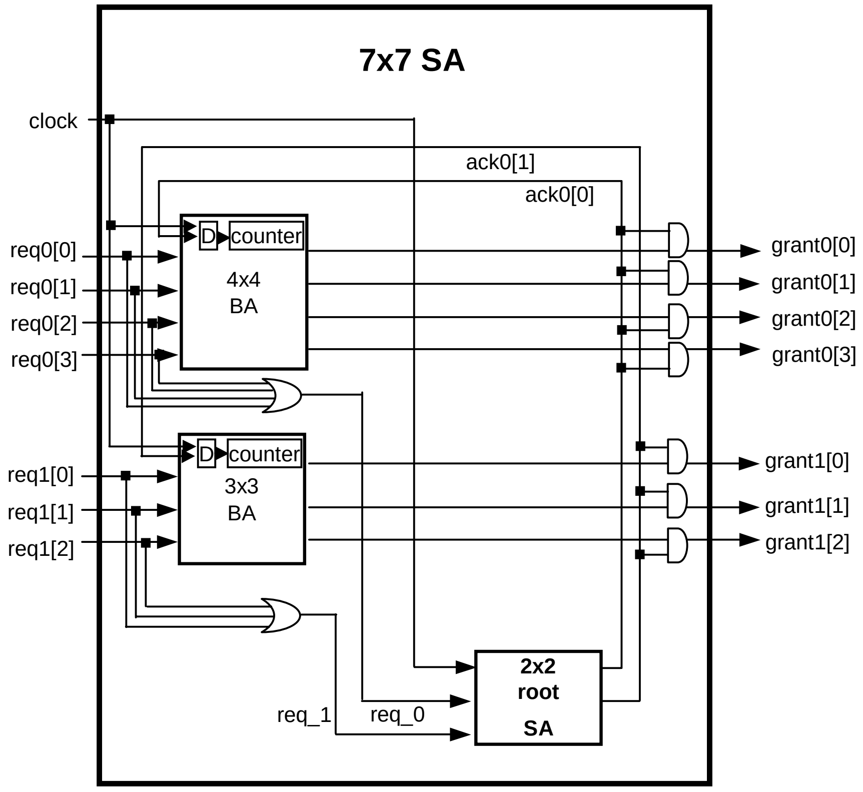


**Figure 21:** A 7x7 SA with a different placement of the AND gates (Note: reset signal not shown)

*ack0[0]* and *ack0[1]* are fed back to a D flip-flop for the next arbitration cycle to rotate the ring counter. However, the grant signals *grant0[3:0]* and *grant1[3:0]* are not fed back but instead are outputs in Figure 21.

Considering Figure 21, the two critical path candidates for the 7x7 SA are (i) a 4-input OR gate, a 2x2 root SA and an AND gate, or (ii) a 4x4 BA followed by a 2-input AND gate. It turns out that the critical path when using a TSMC 0.25$\mu$m standard cell library from LEDA Systems is (i) a 4-input OR gate followed by a 2x2 root SA followed by an AND gate. Note that *ack0* signals from the 2x2 root SA feed into D flip-flops in the 4x4 BAs and thus do not affect the present arbitration cycle.

In Chapter 5, we will give a formal algorithm to design a hierarchical SA. For now note that to reduce the number of levels in the hierarchical SA, we use as many 4x4 switch arbiter blocks as possible because both the speed and area of a 4x4 switch arbiter block are less than the speed and area of employing two levels of 2x2 switch arbiter blocks to handle four requests, for a total of three 2x2 switch arbiter blocks: two leaves and one root as shown in Figure 22. For example, the delay of a 4x4 switch arbiter block is 0.34 $ns$ in a TSMC 0.25$\mu$m library [60] from LEDA Systems [23] which

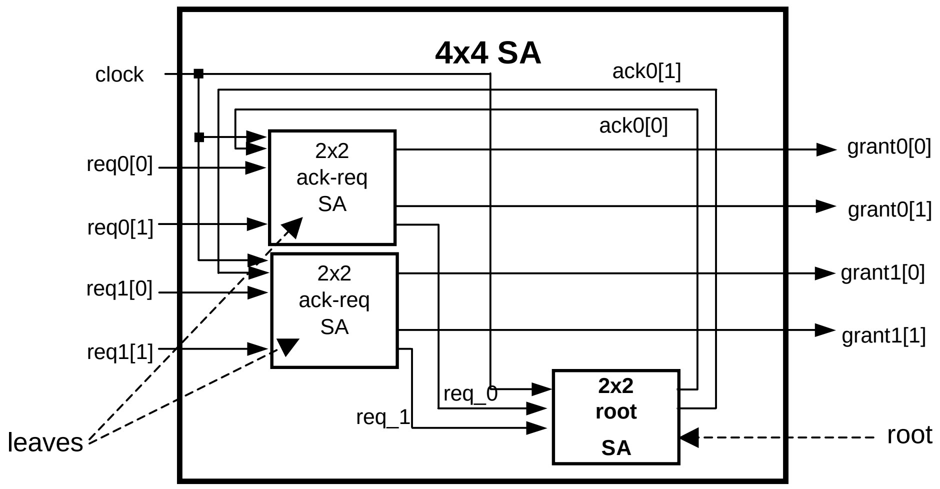


**Figure 22:** A 4x4 SA implemented by three 2x2 switch arbiter blocks (Note: reset signal not shown)

is less than the delay of two levels of 2x2 switch arbiters implementing Figure 22: 0.46 *ns* using the same TSMC 0.25$\mu$m library from LEDA Systems. Also, comparing a 16x16 SA versus a combination of 4x4 switch arbiter blocks implementing a 16x16 hierarchical SA yields the following: a 16x16 SA with a 16-input priority logic block synthesized using the Synopsys Design Compiler leads to 1.49 *ns* gate delay using a TSMC 0.25$\mu$m library from LEDA Systems, while a 16x16 hierarchical SA yields only 0.76 *ns* gate delay using the same standard cell library. So, apparently, for fast implementation, it is best to keep MxM hierarchical SAs built out of 2x2, 3x3 and 4x4 switch arbiter blocks, with a preference for 4x4 switch arbiter blocks. This will be discussed in more detail in Chapter 5.

For a larger example, consider the 32x32 hierarchical SA shown in Figure 23. The 2x2 and 4x4 switch arbiter blocks are composed into a tree structure in Figure 23 (the leftmost blocks are the leaves and the rightmost block is the root). Non-root 2x2 and 4x4 ack-req SA blocks receive an acknowledgment from a switch arbiter block at the next higher level (which translates to being further toward the right hand side of Figure 23) for the next arbitration cycle. Since the root switch arbiter block in the hierarchy does not receive an acknowledgment (because there is no higher level switch arbiter block), the root arbiter uses the clock input to pass the token (i.e., highest priority) to the next master in every arbitration cycle in round-robin order.

In Figure 23,"*l0.sa0*" denotes switch arbiter ("*sa*") 0 in level 0 (the lowest level in the hierarchy where the left hand side of Figure 23 is the lowest). The 4x4 ack-req SAs placed on the left side of Figure 23 are the lowest level (*l0.sa0* through *l0.sa7*) in the hierarchy, and the level goes up moving toward the right. At most one grant out of the 32 grants (outputs) is allowed to be set to logic '1' at a time. This hierarchical SA is a distributed switch arbiter whose individual 2x2 and 4x4 switch arbiter blocks operate in parallel with one another. In other words, the upper level switch arbiters (*l1.sa0*, *l1.sa2* and the *root* arbiter) only arbitrate their own ORed requests (for example,

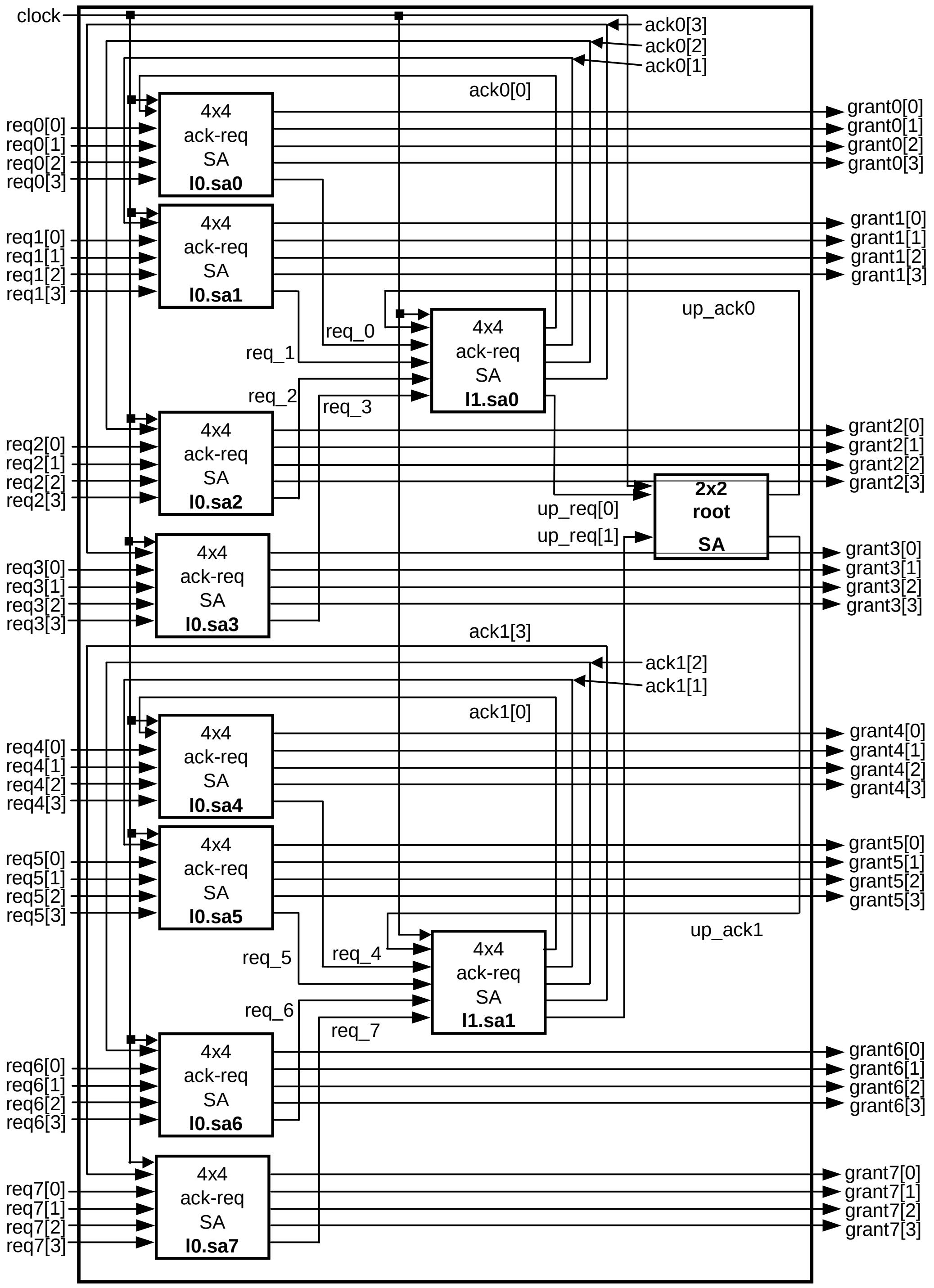


**Figure 23:** Hierarchical Switch Arbiter for 32 x 32 switch (Note: reset signal not shown)

*req_0* through *req_3* for *l1.sa0*) from the lower level switch arbiters (*l0.sa0* through *l0.sa3*) regardless of the *ack* signals from the higher level. Note that, for example, the ORed request *req_0* indicates whether any of *req0[3:0]* is currently making a request; thus, *req_0* indicates to *l1.sa0* that *l0.sa0* has at least one active request which *l0.sa0* can grant if *ack0[0]* is set high, enabling *l0.sa0*. Also, *ack* signals from higher levels are fed back to D flip-flops in each ack-req SA in order to potentially rotate the token bit in the next arbitration cycle.

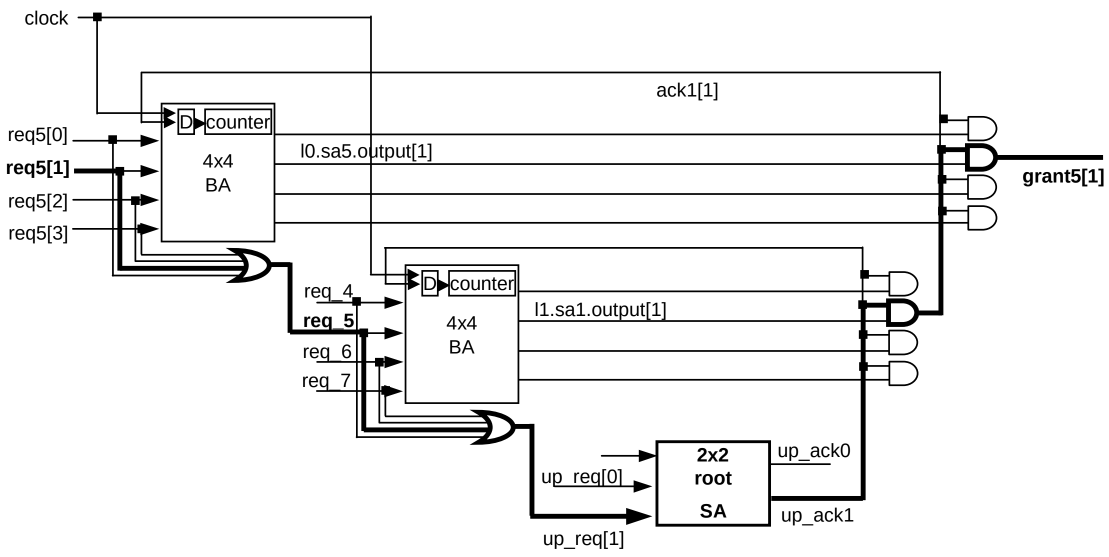


**Figure 24:** The critical path of Figure 23

For critical path considerations, we synthesized an RTL Verilog description of Figure 23 using the Synopsys Design Compiler [51] with a 0.25 $\mu$m TSMC standard cell library [60] from LEDA Systems [23]. The longest logic delay in Figure 23 given by the Synopsys Design Compiler [51] is shown in Figure 24: two levels of ORed requests *req_5* (ORed *req5[0]* through *req5[3]*) and *up_req[1]* (ORed *req_4* through *req_7*), the gate delay of the 2x2 root arbiter, *up_ack1* ANDed with *l1.sa1.output[1]* finishing with *ack1[1]* ANDed with *l0.sa5.output[1]* which produces signal *grant5[1]*. This critical path is indicated by the bold line in Figure 24.

This scheme of Figure 23 results in area savings and delay savings compared with a centralized arbiter. Even more, the design of a class of hierarchical SAs similar to Figure 23 is automated by our RAG tool, which will be described in Chapter 5.

We compare the performance and the area of our SA with those of the Programmable Priority Encoder (PPE), implementing iSLIP, and Ping-Pong Arbiter (PPA) in Chapter 7. We do not compare our SA with dual round-robin matching (DRRM). As mentioned in [7] by Chao, who is the first author of both DRRM [8] and PPA [7], PPA is proposed to reduce complexities due to the centralized arbitration algorithms like DRRM and iSLIP; thus, Chao claims that the arbitration of PPA is faster than that of DRRM. Hence, we do not compare our solution to DRRM but only to PPA, since, presumably according to Chao [7], if our solution is faster than PPA, then our solution is also faster than DRRM by the same or greater margin of speedup.

We chose to design our hierarchical SAs in the way shown in the previous two examples (Figure 20 and Figure 23) for two reasons. First, we want to reduce the number of levels in a hierarchical SA. As we showed in the comparison of a 4x4 SA with a 4x4 hierarchical SA composed of three 2x2 switch arbiter blocks as shown in Figure 22, a hierarchical 4x4 SA has longer logic delay than a 4x4 SA made using a single 4x4 BA (i.e., without any hierarchy). However, since PPA uses only 2x2 switch arbiters, PPA has more levels in its hierarchy resulting in longer logic delay than our SA. Hence, we prefer to use as many as 4x4 switch arbiter blocks possible in our hierarchical SA in order to reduce delay.

Employing priority encoders is one way of implementing an arbiter. However, the number of gates and the logic stages in a priority encoder rapidly increases as the number of masters increases, which leads to the longer critical path delay. Thus, we found that overall delay was the smallest when we limit the size of priority logic blocks to 2-input, 3-input or 4-input blocks to avoid the rapid increase in gate delay and area for PPE (which uses a single large priority encoder) as shown in Figure 45 of

**Algorithm 1** Pseudo code for token passing of the 2x2 root SA and a 4x4 switch arbiter blocks

```
2x2 root token
begin
 1: /*Initialization*/
 2: root token = 2'b01;
 3: if (positive edge of clock) then
 4:     rotate token;
 5: end if
end

4x4 ack-req SA token
begin
 1: /*Initialization*/
 2: ack-req token = 4'b0001;
 3: if (positive edge of clock) then
 4:     if (ack) then
 5:         rotate token;
 6:     end if
 7: end if
end
```

Chapter 7. Furthermore, we use 2x2, 3x3 and 4x4 switch arbiter blocks to implement an MxM SA. Note that 3x3 switch arbiter blocks are utilized only when M is not a power of two.

Algorithm 1 presents in pseudo code how the 2x2 root SA and an 4x4 ack-req SA tokens are passed. The 2x2 root token is rotated every clock cycle using a 2-bit ring counter, while the token of a 4x4 ack-req SA is rotated using a 4-bit ring counter whenever the 4x4 ack-req SA is acknowledged from the higher level.

**Example 4.4** If the current token state of *l1.sa1* (the ring counter outputs token[3:0], see Figure 15 which shows the 4x4 Bus Arbiter of 4x4 ack-req SA in Figure 17) in Figure 23 equals 4'b1000 implying that *req_7* has the highest priority, the token will be rotated to give *req_4* the highest priority in the next arbitration cycle if *up_ack1* is asserted. □

Figure 25 describes in pseudo code an algorithm for a subset of the 32x32 SA shown in Figure 23. Specifically, Figure 25 describes more details on the token values of l0.sa0 and l1.sa0. Since an asserted request is granted according to the priority

```
Switch_Arbiter{
if (root_token==1) {
/*l1.sa0 has the highest priority if           */
/*up_req[0] equals '1'. Otherwise l1.sa1 */
/*has the highest priority                     */
  if (up_req[0]) {
    up_ack0=1;
    if (l1.sa0.token==1) {
/*l0.sa0 has the highest priority              */
      if (req_0) {
/*if one of requests from l0.sa0 is '1'       */
        ack0[0]=1;
        if (l0.sa0.token==1) {
          if (req0[0]) grant0[0]=1;
          else if (req0[1]) grant0[1]=1;
          else if (req0[2]) grant0[2]=1;
          else grant0[3]=1;
        }
        else if (l0.sa0.token==2) {
          if (req0[1]) grant0[1]=1;
          else if (req0[2]) grant0[2]=1;
          else if (req0[3]) grant0[3]=1;
          else grant0[0]=1;
        }
        else if (l0.sa0.token==4) {
           req0[2] has the highest
          priority;
        }
        else if (l0.sa0.token==8) {
           req0[3] has the highest
          priority;
        }
      }//if (req_0)
      else if (req1) {
        ack0[1]=1;
        if (l0.sa1.token==1) {
          if (req1[0]) grant1[0]=1;
          else if (req1[1]) grant1[1]=1;
          else if (req1[2]) grant1[2]=1;
          else grant1[3]=1;
        }
        else if (l0.sa0.token==2) {
          req1[1] has the highest
          priority;
        }
        else if (l0.sa0.token==4) {
           req1[2] has the highest
          priority;
        }
        else if (l0.sa0.token==8) {
           req1[3] has the highest
          priority;
        }
      }//else if (req1)
      else if (req2) {
        ack0[2]=1;
        one of req2[3:0] has the
        privilege to be granted;
        req2[3:0] is granted according
        to the value of l0.sa2.token  in
        round-robin order;
      } // else if (req2)
      else {  //else if (req3)
        ack0[3]=1;
        one of req3[3:0] has the
        privilege to be granted;
        req3[3:0] is granted according
        to the value of l0.sa3.token  in
        round-robin order;
      } // else
      }//if (l0.sa0.token==1)
    else if (l1.sa0.token==2) {
      if (req1) {
        ack0[1]=1;
        one of req1[3:0] has the
        privilege to be granted;
        req1[3:0] is granted according
        to the value of l0.sa1.token  in
        round-robin order;
        } // if (req1)
      else if (req2) {
        ack0[2]=1;
        one of req2[3:0] has the
        privilege to be granted;
        req2[3:0] is granted according
        to the value of l0.sa2.token  in
        round-robin order;
      } // else if (req2)
      else if (req3) {
        ack0[3]=1;
        one of req3[3:0] has the
        privilege to be granted;
        req3[3:0] is granted according
        to the value of l0.sa3.token  in
        round-robin order;
      } // else if (req3)
      else { //else if (req_0)
        ack0[0]=1;
        one of req0[3:0] has the
        privilege to be granted;
        req0[3:0] is granted according
        to the value of l0.sa0.token  in
        round-robin order;
      } // else
    }//else if (l0.sa0.token==2)
    else if (l1.sa0.token=4) {
      req2[3:0] has the highest priority;
      req3[3:0] is the second  priority;
      req0[3:0] is the third priority;
      req1[3:0] has the lowest priority;
    }//else if (l0.sa0.token==4)
    else if (l1.sa0.token==8) {
      req3[3:0] has the highest priority;
      req0[3:0] is the second  priority;
      req1[3:0] is the third priority;
      req2[3:0] has the lowest priority;
    }//else if (l0.sa0.token==8)
  }//if (up_req[0])
  else if (up_req[1]) {
    up_ack0=1;
    if (l1.sa1.token==1) {
      req4[3:0] has the highest priority;
      req5[3:0] is the second  priority;
      req6[3:0] is the third priority;
      req7[3:0] has the lowest priority;
    }//if (l1.sa1.token==1)
    else if (l1.sa1.token==2) {
      req5[3:0] has the highest priority;
      req6[3:0] is the second  priority;
      req7[3:0] is the third priority;
      req4[3:0] has the lowest priority;
    }//else if (l1.sa1.token==2)
    else if (l1.sa1.token==4) {
      req6[3:0] has the highest priority;
      req7[3:0] is the second  priority;
      req4[3:0] is the third priority;
      req5[3:0] has the lowest priority;
    }//else if (l1.sa1.token==4)
    else if (l1.sa1.token==8) {
      req7[3:0] has the highest priority;
      req4[3:0] is the second  priority;
      req5[3:0] is the third priority;
      req6[3:0] has the lowest priority;
    }//else if (l1.sa1.token==8)
  }//else if (up_req[1])
  else
    set all grant signals to '0's;
}//if (root_token==1)
else { //root_token==2
  if (up_req[1]) {
    up_ack1=1;
    if (l1.sa1.token==1) {
      req4[3:0] has the highest priority;
      req5[3:0] is the second  priority;
      req6[3:0] is the third priority;
      req7[3:0] has the lowest priority;
    }//if (l1.sa1.token==1)
    else if (l1.sa1.token==2) {
      req5[3:0] has the highest priority;
      req6[3:0] is the second  priority;
      req7[3:0] is the third priority;
      req4[3:0] has the lowest priority;
    }//else if (l1.sa1.token==2)
    else if (l1.sa1.token==4) {
      req6[3:0] has the highest priority;
      req7[3:0] is the second  priority;
      req4[3:0] is the third priority;
      req5[3:0] has the lowest priority;
    }//else if (l1.sa1.token==4)
    else if (l1.sa1.token==8) {
      req7[3:0] has the highest priority;
      req4[3:0] is the second  priority;
      req5[3:0] is the third priority;
      req6[3:0] has the lowest priority;
    }//else if (l1.sa1.token==8)
  }//if (up_req[1])
  else if (up_req[0]) {
    up_ack0=1;
    if (l1.sa0.token==1) {
      req4[3:0] has the highest priority;
      req5[3:0] is the second  priority;
      req6[3:0] is the third priority;
      req7[3:0] has the lowest priority;
    }//if (l1.sa0.token==1)
    else if (l1.sa0.token==2) {
      req5[3:0] has the highest priority;
      req6[3:0] is the second  priority;
      req7[3:0] is the third priority;
      req4[3:0] has the lowest priority;
    }//else if (l1.sa0.token==2)
    else if (l1.sa0.token==4) {
      req6[3:0] has the highest priority;
      req7[3:0] is the second  priority;
      req4[3:0] is the third priority;
      req5[3:0] has the lowest priority;
    }//else if (l1.sa0.token==4)
    else if (l1.sa0.token==8) {
      req7[3:0] has the highest priority;
      req4[3:0] is the second  priority;
      req5[3:0] is the third priority;
      req6[3:0] has the lowest priority;
    }//else if (l1.sa0.token==8)
  }//else if (up_req[0])
  else
    set all grant signals to '0's;
}//else
```

**Figure 25:** Switch Arbiter algorithm

state of the corresponding switch arbiter block, we begin with an explanation of the priority state of a switch arbiter block. The priority state is determined by the token. We start by explaining the priority state of the root SA since the root SA is on the top of the hierarchy and acknowledges one of its children based on its token status. Figure 25 shows the following: which asserted request signal receives the highest priority in a given cycle is based on the token values of switch arbiter blocks. From a high level perspective, each switch arbiter block grants one of its requests according to its current token values and, at the same time, generates an ORed request to the corresponding switch arbiter block in the higher level; which request is granted by a switch arbiter block is independent of the outputs of other switch arbiter blocks. The output of a switch arbiter block only needs to be approved by the assertion of the *ack* signal input, which is equivalent to a "grant" signal from the higher level switch arbiter block.

The *2x2 root SA* has a token, token[1:0], which determines the priority state and is output from the ring counter shown in *2x2 root SA* in Figure 18. According to the token value of the *2x2 root SA*, one of the *2x2 root SA*'s children is granted if there is at least one asserted request from its children. Also, each 4x4 ack-req SA in Figure 23 has a token (the ring counter outputs token[3:0], see Figure 15 which shows the 4x4 Bus Arbiter of 4x4 ack-req SA in Figure 17) inside of the 4x4 Bus Arbiter in the ack-req SA. Similar to the grant operation of the *2x2 root SA*, a *4x4 ack-req SA* grants one of its request inputs according to its token value. In Figure 25, the top if-else block, *if(root_token==1)*, first checks the token of the *2x2 root SA* in Figure 23. Then, the *2x2 root SA* acknowledges one of its children (*l1.sa0* or *l1.sa1*) by granting one of request signals (*up_req[0]* or *up_req[1]*) according to the values of its token and request input signals. Likewise, the acknowledged 4x4 ack-req SA (say, *l1.sa0*) authorizes one of its children (say, *l0.sa0* through *l0.sa3*) by asserting one of *ack* signals (say, *ack0[0]*) depending on the values of its token and input requests

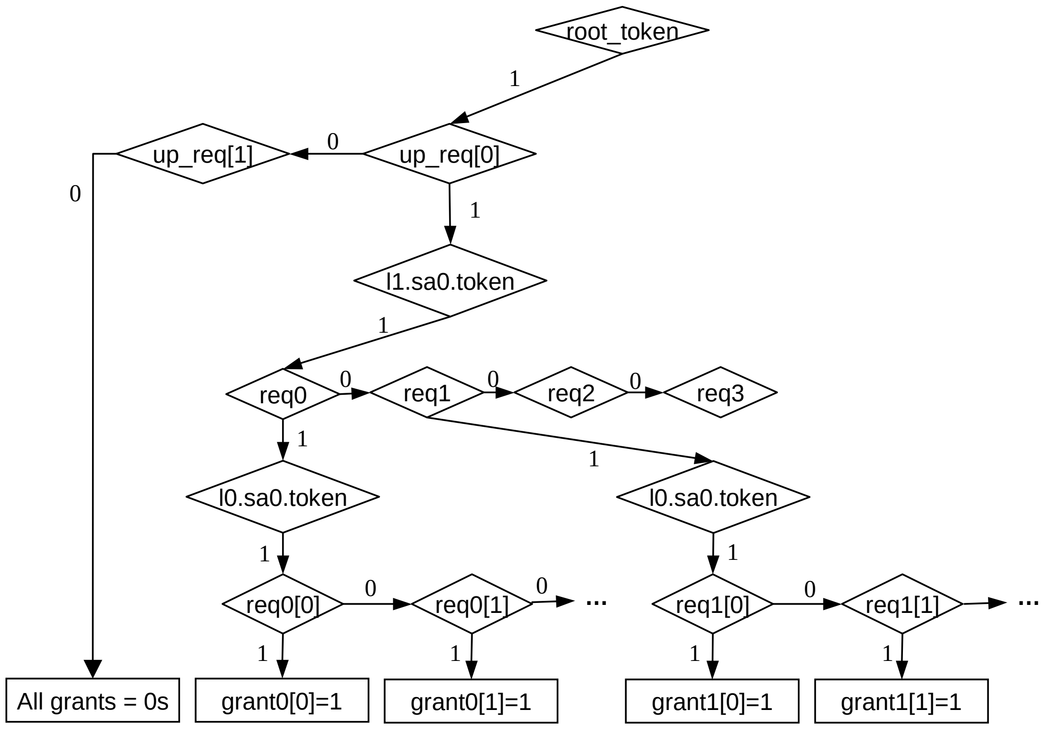


**Figure 26:** Flowchart of Switch Arbiter algorithm

(say, *req_0* through *req_3*). Similarly, the acknowledged 4x4 ack-req SA (say, *l0.sa0*) at the level 0, grants one request signal based on the token status and the values of input request signals (say, *req0[3:0]*). In general, the values of *ack* signals and *grant* signals are determined according to the value of input request signals (including ORed request) and the token status of each switch arbiter blocks. We explain in detail two specific examples (Example 4.5 and Example 4.6) with a simplified flowchart version of Figure 25 as shown in Figure 26.

**Example 4.5** We describe the case where *req0[0]* has the highest priority and all 32 request signals (*req0[3:0]* through *req7[3:0]*) in Figure 23 are asserted. We assume that in the previous cycle, no request was asserted and thus all grant signals (*grant0[3:0]* through *grant7[3:0]*) are zero. Suppose the token of the *2x2 root SA* in Figure 23 is equal to 1 (i.e., token[1:0]=2'b01); recall that token[1:0] is an internal value output by the two-bit ring counter of the *2x2 root SA* in Figure 23. Also, in order to give specific priority orders to all 4x4 ack-req SAs, assume that token (token[3:0]) of each 4x4 ack-req SA is equal to 4'b0001. Then the 32x32 hierarchical SA in Figure 23) works as follows. Since the token of the *2x2 root* SA is equal to 1 (2'b01), l1.sa0 has a higher priority than *l1.sa1* in Figure 23. Since all bits of *req0[3:0]* are asserted, *req_0* (the bit-wise OR of *req0[3:0]*) is equal to logic '1' resulting in *up_req[0]* set to '1'. In this case, the *2x2 root SA* grants *l1.sa0* by setting *up_ack0* equal to '1' since *up_req[0]* is equal to '1' and *l1.sa0* has higher priority. Thus, *l1.sa0* has been acknowledged and has the privilege to grant one of its children, *l0.sa0* through *l0.sa3*. Similarly, *l1.sa0* acknowledges *l0.sa0* by setting *ack0[0]* equal to '1' because *l0.sa0* has the highest priority since the token of *l1.sa0* is equal to 4'b0001. Then, *l0.sa0* grants one of requests signals (*req0[0]* through *req0[3]*) based on the priority. Since *req0[0]* is prior to other request inputs to *l0.sa0* (the token of *l0.sa0* is equal to 4'b0001) and *req0[0]* is equal to '1' by our assumption, *req0[0]* is the final winner and only *grant0[0]* is set to logic '1'. □

**Example 4.6** This example shows the same case as Example 4.5, except that only *req0[3:0]* are not asserted, under the same condition of priority orders (tokens) assumed in Example 4.5 (the token of the 2x2 root SA is equal to '1' and tokens of all 4x4 ack-req SAs are equal to 4'b0001). In the same way described in Example 4.5, *l1.sa0* is acknowledged to grant one of its children by the *up_ack0*. Since the highest priority request input, *req_0*, is negated, *l1.sa0* must look for a child switch arbiter in the descending order of priority. First, *l1.sa0* checks if *req_1* is '1'. If so, *l1.sa0* acknowledges *l0.sa1* by setting *ack0[1]* equal to '1'. Otherwise, *l1.sa0* checks whether *req_2* is '1'. Since *req_1* is equal to '1', *l1.sa0* sets *ack0[1]* equal to '1'. The acknowledged switch arbiter *l0.sa1* then grants one of its request signals (*req1[0]* – *req1[3]*) based on priority. Since *req1[0]* has the highest priority (because the token of *l0.sa1* is equal to 4'b0001), *req1[0]* is the final winner. □

## 4.3 Hierarchical Bus Arbiter Design

The basic logic of a hierarchical Bus Arbiter (BA) is the same as that of a hierarchical SA. However, a hierarchical BA has an extra *done* input which indicates the completion of a single use of the bus for the current bus owner. Thus, the current bus owner can hold a bus for multiple cycles in a hierarchical BA; in a hierarchical SA, on the other hand, the granted input port can take a specific output port only for a single cycle, and thus no *done* input is needed. Figure 27 shows an 8x8 hierarchical BA.

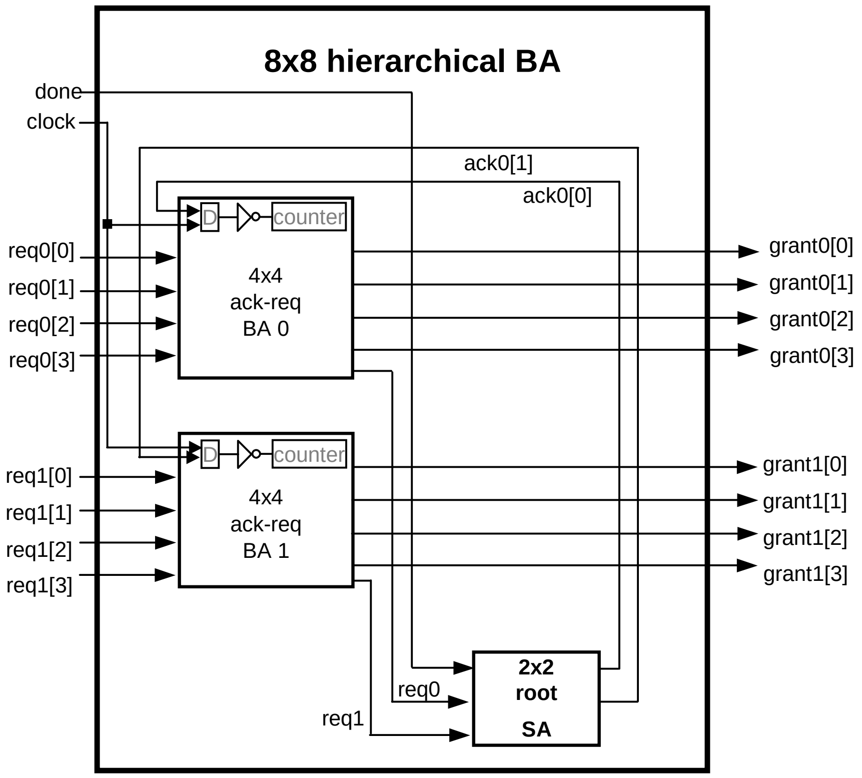


**Figure 27:** 8x8 hierarchical BA

Note that a hierarchical BA still uses a root SA. However, the root SA in a hierarchical BA takes an *done* input that rotates a root token only after a rising edge is seen on the *done* input, while a root SA in a hierarchical SA takes a *clock* input to rotate the root token every clock cycle. In addition, a hierarchical BA employs ack-req BA(s) instead of ack-req SA(s) in order for the granted bus master to hold a bus for multiple cycles. Figure 28 shows a 4x4 ack-req BA.

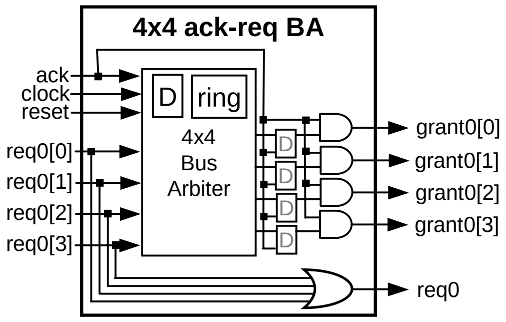


**Figure 28:** 4x4 ack-req BA

Let us briefly describe how an ack-req BA functions. First note that the assertion of an *ack* input (from a higher level in the hierarchy) to an ack-req BA acknowledges that one of its requests was granted. Thus, the token (the content of *ring counter* in Figure 28) of an ack-req BA must be updated after a data transfer completion noted by the acknowledgment (*ack0[0]* for *BA 0* or *ack0[1]* for *BA 1* in Figure 27). The acknowledgment remains asserted (logic '1') during the use of the bus. Unlike 4x4 ack-req SA, the *ring counter* of the *4x4 ack-req BA* takes the $\overline{Q}$ output from a *D flip-flop* and the *ring counter* consists of rising-edge D flip-flops. Thus, the falling edge of the acknowledgment (*ack0[0]* or *ack0[1]* in Figure 27) triggers the ring counter for a token update. The *4x4 ack-req BA* also has additional logic, four *D latch*es as shown on the right-hand-side of Figure 28, compared with an ack-req SA. These *D latch*es that are placed between a bus arbiter and AND gates to hold grant outputs from a bus arbiter until the rising edge of the acknowledgment signal; this is necessary because the output of a 4x4 Bus Arbiter, depending on request inputs, could change during the use of the bus. In this manner, a granted bus master can hold the bus until the completion of a single bus usage (which may consume multiple bus cycles).

**Example 4.7** Consider a scenario where eight processors as bus masters are connected to a single bus with one large shared memory and an 8x8 hierarchical BA that arbitrates bus requests from the eight masters. In the 8x8 hierarchical BA, there are two 4x4 ack-req BAs and one 2x2 root SA as shown in Figure 27. Suppose *req0[1]* from processor 1 is granted; thus, *grant0[1]* is asserted. Next, processor 0 and processor 4 request a bus by the assertions of *req0[0]* and *req1[0]*, respectively. The token of the *2x2 root SA* - currently, the token of the *2x2 root SA* is equal to 2'b01 - is not rotated until the *done* input from processor 1 is received. Thus, 4x4 ack-req BA 0 remains activated by the assertion of *ack0[0]*. Since *req0[0]* is asserted, the output of 4x4 Bus Arbiter in 4x4 ack-req BA 0 becomes 4'b0001 granting *req0[0]*. However, *grant0[1]* is still asserted because of the D latches in the 4x4 ack-req BA 0. After processor 1 completes its transfer by asserting the *done* input to the 8x8 hierarchical BA, the token of the root SA is rotated to 2'b10. Finally, the 2x2 root SA deactivates 4x4 ack-req BA 0 and activates 4x4 ack-req BA 1 which serves processor 4 by asserting *ack0[1]* since the token of the *2x2 root SA* becomes 2'b10. □

## 4.4 Impact on Logic Synthesis: Priority Logic Specification

As noted in Section 4.2, a 16x16 SA with a 16-input priority logic block synthesized using the Synopsys Design Compiler leads to critical path gate delay of 1.49 *ns* using a TSMC 0.25$\mu$m standard cell library from LEDA Systems, while a 16x16 hierarchical SA yields a critical path gate delay of only 0.76 *ns*. This large decrease – from 1.49 *ns* down to 0.76 *ns* in this case – holds in general for powers of two above $M = 4$ as will be shown empirically in Chapter 5 and Chapter 7 (for $M = 4$ and $M = 2$ the logic structure and thus delay is non-hierarchical and exactly the same).

### 4.4.1 Hierarchical SA versus PPE

Consider our hierarchical 32x32 SA shown in Figure 23 as compared to the equivalent 32x32 PPE Switch Arbiter, which is basically composed of a 32x32 priority encoder plus some associated control logic. Considered from a logic synthesis perspective [3, 15, 32], the **logic** of which grant signal is selected, given a sequence of requests, is roughly equivalent. Yet the worst-case delay of the logic synthesized for Figure 23 is less than half of the delay of the logic synthesized for a 32x32 PPE (see Chapter 7 for detailed results)! Clearly, then, the difference lies in the logic structure input to the logic synthesis tool.

Consideration of logic structure in PPE versus our hierarchical SA leads us to focus on what is by far the largest and most complicated logic component of each: priority encoder(s). In the priority encoder of PPE, a single priority encoder is used with a number of inputs equal to the number of inputs to the switch arbiter. In our hierarchical SA, on the other hand, the number of inputs to any priority logic block is limited to two, three or four; thus, in our case, the logic synthesis tool is biased toward considering two-input, three-input and four-input logic gates. With a single priority encoder with many (much greater than four) inputs, on the other hand, the logic synthesis heuristics must search for a logic structure to minimize delay and area given boolean equations with much greater than four boolean inputs; thus, more is asked of the logic synthesis heuristics in terms of finding the best mapping. In a sense, more freedom is given to the logic synthesis heuristics; in our hierarchical SA, on the other hand, a more constrained problem is given (namely, the synthesis of 2x2, 3x3 and 4x4 SAs with a clearly specified interconnect netlist). In short, this is one large difference: the logic equations given in the case of PPE are much larger, while the logic equations given in the case of our hierarchical SA are more constrained to a specific structure.

Consideration of logic structure in PPE versus our hierarchical SA leads to another perhaps less important but nonetheless very clear difference: the tokens and internal acknowledge signals in our hierarchical SA are simply not present in PPE. Note that none of the current logic synthesis algorithms such as heuristic logic minimizations [32, 15] consider the token passing method that we employed in this thesis. From a high level point of view, a parent token enables one of its children by assertion of an *ack* signal. Exemplified with Figure 23, the root SA (parent) acknowledges either *l1.sa0* or *l1.sa1* (child) by the *up_ack0* or *up_ack1* signal according to the token state. On the other hand, the token of each 2x2 or 4x4 switch arbiter block consequently grants one of its inputs by enabling one of its priority logic blocks according to the token state. Moreover, the token – updated by an *ack* signal from the higher level for ack-req SAs and the clock signal for a root SA– is a kind of state stored locally and updated by the next clock edge. Such local information gained by the passing of tokens back to internal inputs (states) seems to help to reduce the longest path delay. In a sense, our hierarchical SA keeps a distributed "state" in its tokens and acknowledge signals. This internally distributed state is updated each cycle and may also be a reason for achieving a shorter critical path as compared to PPE which does not save any internal state values at all. PPE employs thermometer encoding logic [13] which rotates the priority order based on the pointer value. This thermometer encoding logic causes extra delay in the current cycle compared with our token passing approach. Furthermore, this approach – namely, inclusion of specialized internal distributed state values – is definitely not an approach which is explored or in any way considered by traditional logic synthesis algorithms.

In short, the difference of our approach versus PPE lies in (i) a more constrained logic structure biased toward four-input logic blocks and (ii) a distributed internal state used as part of the decision of which input receives the grant.

### 4.4.2 Hierarchical SA versus PPA

PPA also uses hierarchy but limits its priority logic blocks to 2-input. Thus, the major difference of our approach versus PPA is that we use as many 4-input priority logic blocks as possible, thus biasing logic synthesis toward four-input logic gates. This is one main difference.

Another difference relates to the use of 4-input priority logic blocks can be seen in Figure 29. Figure 29 shows that a single 4-input priority logic structure, which is implemented as one base 4x4 unit (e.g., Figure 15 or Figure 19) with critical path delay through one 4-input priority encoder in our approach, results in a critical path delay through two *AR2*s in PPA. Thus, as the number of inputs becomes large (e.g., 16 or 32), the number of logic stages on the critical path in PPA is roughly twice the number of critical path stages in our approach.

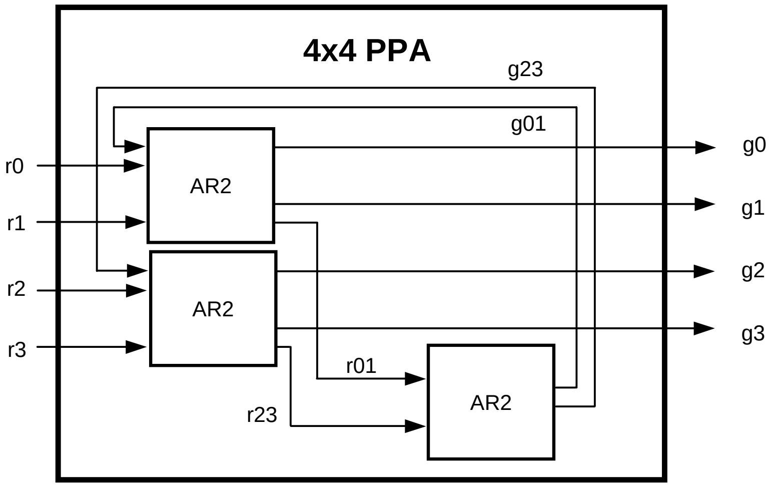


**Figure 29:** 4x4 PPA

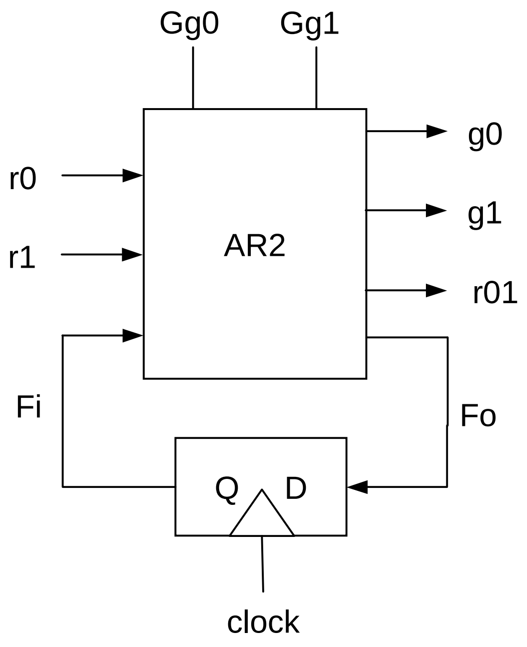


**Figure 30:** AR2 block diagram

Similar to our token-passing approach in our 2x2, 3x3 and 4x4 switch arbiter blocks, each *AR2* has a D flip-flop which keeps track of the previous granted input shown in Figure 30. Thus, for the top left *AR2* in Figure 29, *r0* has the higher priority over the other input (*r1*) if *r1* were granted in the previous cycle.

In conclusion, the hierarchical SA approach is better than the centralized SA approach (e.g., PPE) in a priority encoder based arbiter design. Also, a hierarchical SA with careful consideration of technology library mapping (e.g., a bias toward four-input logic gates) is preferable to the tendency toward two-input logic gates (e.g., the PPA approach) in logic synthesis. Detailed synthesis and timing comparisons are presented in Chapter 7. The goal of this section, Section 4.4, was to explain to the reader the main high-level differences in our hierarchical SA logic specification resulting in such dramatic speedups when the arbitration logic is synthesized.

## *4.5 Fairness in Arbitration*

In this section, we discuss the *absolute fairness* issue. We define an arbiter to be *absolutely fair* if the number of grants per request is equally distributed over $m$ requests in $m$ cycles, where $m \leq M$, for all cases. However, all published switch arbiters which are implemented based on priority encoder(s) show unfairness for certain cases. The following examples show unfairness occurrences in our SA, PPE implementing the iSLIP algorithm and PPA, respectively.

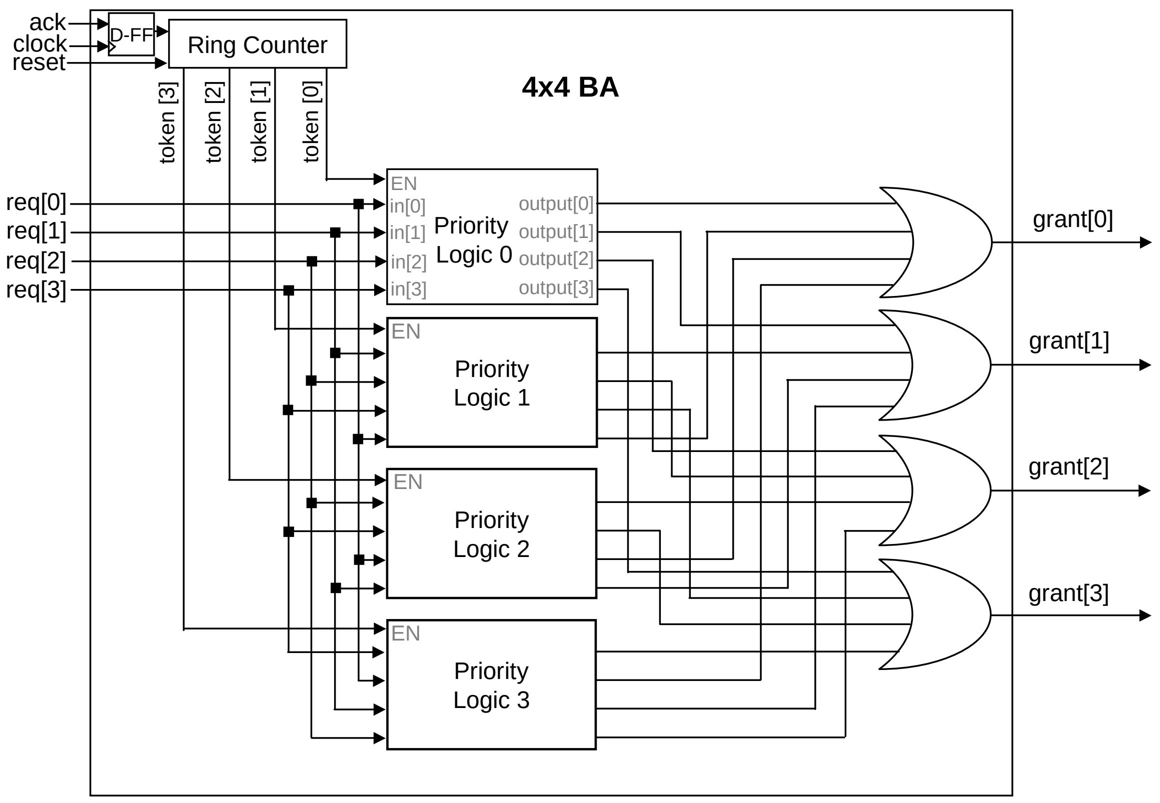


**Figure 31:** 4x4 Bus Arbiter (BA) architecture (Note: Figure 31 is exactly same as Figure 15)

To explain the unfairness case of our SA, we begin with the priority rotation of an ack-req SA. Figure 31 (repeated from Figure 15) shows a *4x4 BA* which is the internal logic of a 4x4 ack-req SA. The priority allocation scheme in Figure 31 is not perfectly fair for the case of 4x4 when less than four inputs to a 4x4 BA make requests. Nonetheless, in the worst case the number of grants per request is one in four cycles. Example 4.8 shows a case where the fairness problem occurs.

**Example 4.8** Suppose only *req[0]* and *req[1]* in Figure 31 request continuously. Furthermore, suppose that initially the content of token[3:0] is equal to 4'b0001. Then *req[0]* is granted in the first cycle. In the next cycle, *req[1]* is granted by the enabling of *Priority Logic 1* since token[3:0] is equal to 4'b0010. At the third and fourth cycles, *Priority Logics 2* and *3* are enabled, respectively, resulting in *req[0]* being granted in the next two consecutive cycles. Consequently, *req[1]* is granted once every four cycles while *req[0]* is granted three times per four cycles because *req[1]* has a higher priority than *req[0]* only when *Priority Logic 1* is enabled, and *Priority Logic 1* is enabled only when token[3:0] equals 4'b0010. □

A centralized arbiter, PPE implementing the iSLIP algorithm, also shows unfairness in arbitration for some cases. The pointer in iSLIP indicates which request signal has the highest priority. In [13], the pointer is incremented by one regardless of the latest granted request. We show unfairness occurrence for a case in Example 4.9.

**Example 4.9** Consider a 4x4 PPE where the pointer is incremented by one. Figure 32 shows how the pointer is updated as described in [29]. Numbers in Figure 32 correspond to request signal numbers. The request pointed to by the pointer has the highest priority and the priority order becomes lower clockwise. Suppose only input ports 0 and 1 compete for output port 0 continuously, and the pointer initially points to 0 indicating input port 0 has the highest priority at the *1st cycle*. Thus, input port 0 is granted in the *1st cycle* and the pointer is incremented by one. In the *2nd cycle* cycle, input port 1 is granted since the pointer points to 1. Now the pointer points to 2 and input port 2 has the highest priority

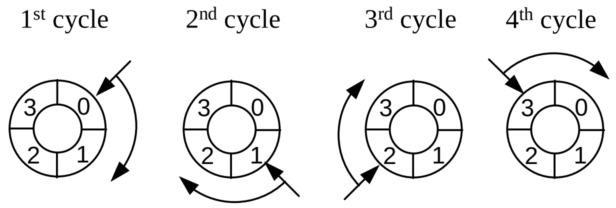


**Figure 32:** The pointer update scheme of PEE

but port 2 does not request output port 1. Thus, input port 0 is granted in the *3rd cycle* because input port 0 has a higher priority than input port 1 in the PPE. Similarly, input port 0 is granted again in the *4th cycle* where the pointer points to 3. Consequently, input port 0 is granted three times while input port 1 is granted once in four consecutive cycles. This unfair number of grants per request becomes worse for a larger PPE. For a 32x32 PPE with the same scenario, input port 0 is granted thirty-one times while input port 1 is granted only once in thirty-two consecutive cycles. □

The next example shows the fairness problem in a distributed arbiter such as PPA and our SA.

**Example 4.10** Consider a 4x4 PPA as shown in Figure 29. Suppose the 4x4 PPA resolves contention at output port 0, and only *r0*, *r1* and *r2* request output port 0 continuously. Each *AR2* is supposed to grant two request inputs one after another. Assume that initially the token state indicates that *r0*, *r2* and *r01* have the highest priority in each *AR2* shown in Figure 29. In the first cycle, *r01* (ORed request of *r0* and *r1*) and *r0* are granted resulting in the assertion of *g0*; thus, input port 0 wins in the competition. In the second cycle, *r23* has the highest priority because *r01* was granted in the previous cycle. Thus, *r2* is granted by the assertion of *g2* in the second cycle. Likewise, since *r01* has the higher priority than *r23* in the third cycle, *g1* is asserted to grant *r1*. Similar to the second cycle, *r23* has the higher priority than *r01* in the fourth cycle. Thus, *g2* is asserted to grant *r2* again. Consequently, input port 2 is granted twice in four consecutive cycles, while input port 0 and input port 1 are granted once. This unfairness becomes worse for larger cases. Consider a 32x32 PPA where all 16 request signals (*r0* − *r15*) are asserted from the top half and only one request, say *r16*, is asserted from the bottom half. Then, *r16* is granted sixteen times while each of 16 request signals (*r0* − *r15*) is granted only once in thirty-two consecutive cycles. The case described in this example also applies, with the same result, to the hierarchical SA presented in this thesis. □

From the above examples, we can see that none of the high-performance arbiters reported in the literature and known to this thesis author is absolutely fair when less than all inputs to a 4x4 arbiter make requests. However, note that under no circumstances do arbiters discussed in this section give priority to an input less often than in the fully congested case (i.e., all inputs requesting all the time). Therefore, all inputs achieve throughput at least as high as the fully congested case. In Chapter 7, we show that the impact of not having an *absolutely fair* arbiter turns out to be insignificant for uniform and burst traffic loads in a network switch.

In addition, the Transport Control Protocol/Internet Protocol (TCP/IP) itself does not have any mechanism associated with fairness issues [10]. TCP/IP involves five layers: application layer, transport layer associated with TCP, internet layer associated with IP, network access layer and physical layer [48]. When a host, say *A*, wants to send user data to another host, there are several steps which the user data have to go through involving the five layers. A host *A* first sends user data down to TCP with an appropriate encapsulation. Then, TCP sends a TCP segment down to IP. Next, IP hands an IP datagram down to the network access layer, where Ethernet is the most popular network access layer for a LAN system. Carrier-Sense

Multiple Access with Collision Detection (CSMA/CD) is the most common medium access control technique for Ethernet. In CSMA/CD, a station can transmit if the medium is idle. Otherwise a station listens to the medium until the channel is idle. If a collision occurs during transmission, the channel owner stops transmitting and backs off for a random amount of time. Obviously, there is no guarantee of fairness in the CSMA/CD protocol.

Consequently, as long as there is no potential occurrence of starvation (there is not), it is reasonable to conclude that the relative performances of PPA, PPE and our SA are roughly equivalent in terms of fairness: for an M-input PPA, PPE or our SA, each input always receives a grant at least $1/M$ of every M cycles, and, in the case that some inputs are not asserted, the assignment of the "extra" grants are distributed in a way that is not perfectly fair.

In this chapter, we have described how we design our hierarchical SA and BA. We also identified how we reduce the critical path delay of our design. We further discussed unfairness issues with our SA, PPE and PPA, concluding that while none is perfectly fair, no request is satisfied less often than the fully congested case (thus, none suffers from starvation either). In the following chapter, Chapter 5, we will present how our hierarchical SA and BA are automatically generated.

# CHAPTER V

# RAG: ROUND-ROBIN ARBITER GENERATOR

Guided by user specifications, our Round-robin Arbiter Generator (RAG) tool can generate synthesizable Verilog for an MxM hierarchical BA able to handle M simultaneous requests. RAG can also combine switch arbiter blocks (2x2, 3x3 and 4x4 switch arbiter blocks) to produce synthesizable Verilog for an MxM hierarchical SA according to user specifications.

Figure 33 shows the flow of RAG. First, the user chooses the arbiter type (bus or switch) and the number of masters for the chosen arbiter type. The truth table shown in Table 1 in Section 4.1 shows a regular pattern as M increases and is implemented with simple logic equations in Verilog for the generation of a Bus Arbiter (BA). From Figure 42 and Figure 43, it turns out that employing a hierarchical BA to implement the BA logic is better in terms of area and speed when the number of masters (M) is greater than 4. We will discuss area considerations in more detail in Section 7.1.1.

Using the function, *interpret_sa()* in Figure 33, the tool first calculates the number of levels in the hierarchy by dividing M by 4 for an MxM hierarchical Bus Arbiter (BA) or Switch Arbiter (SA). Then, the remainder of the previous division is divided by 4 iteratively until the quotient of the division reaches zero. The tool prefers to use as many 4x4 switch arbiter blocks as possible per each level. However, 3x3 switch arbiter blocks are utilized for a level in a hierarchy if the number of request inputs in the level is not a multiple of 4 but instead is a multiple of 3. If (the number of request inputs per level) modulo 4 is equal to 3, a 3x3 switch arbiter is used. If (the number of request inputs per level) modulo 4 is less than 3, a 2x2 switch arbiter is utilized. However, if (the number of request inputs per level) modulo 4 is less than 2

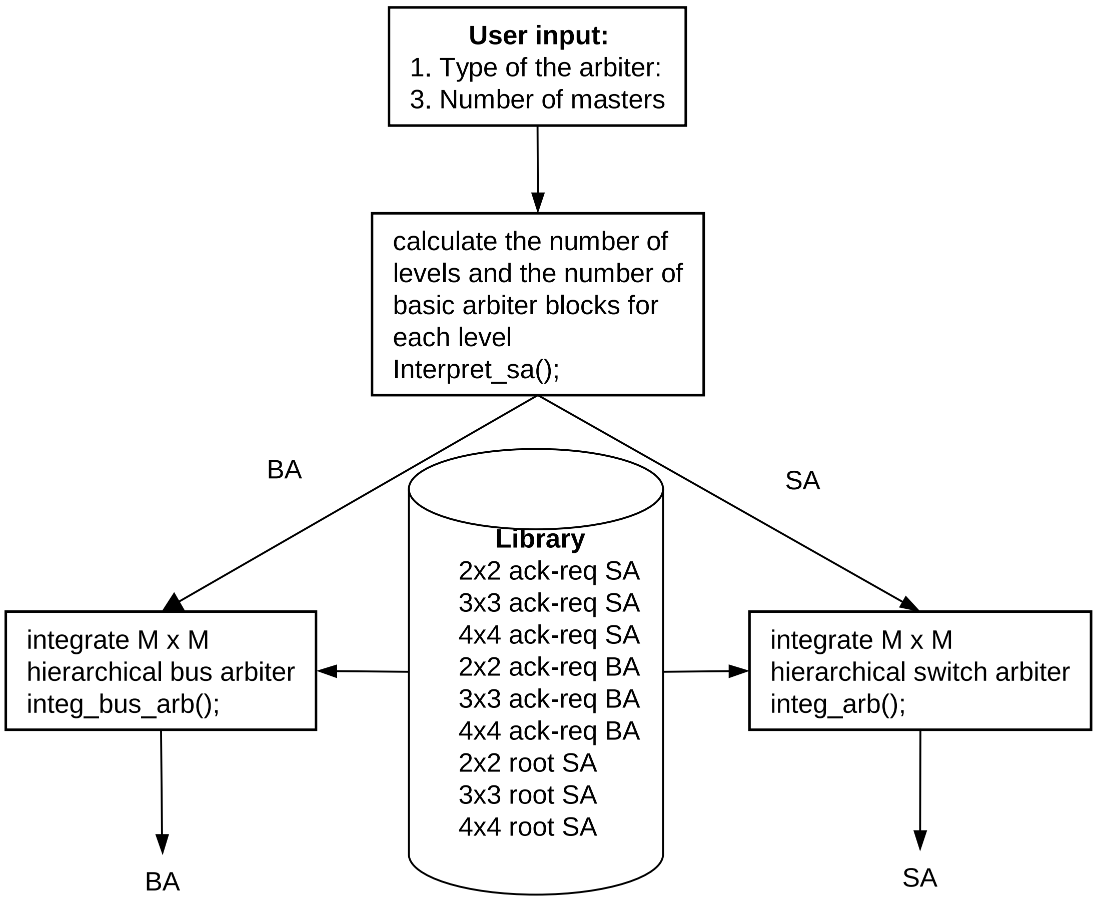


**Figure 33:** Flow of RAG tool

but not equal to 0, one remaining request signal is advanced to next highest level. In other words, RAG does not employ any switch arbiter block to serve one request.

**Example 5.1** This example shows how the number of levels in a hierarchy and the number of switch arbiter blocks per level are determined when $M = 7$. First, RAG divides 7 by 4. Then, the remainder equals 3. Since the remainder is not equal to 0, RAG divides 3 by 4. Now the quotient of the second division equals 0. Thus, RAG determines the number of levels to be 2 since divisions are executed twice.

After RAG determines the number of levels, RAG calculates the number of switch arbiter blocks per level. For the first level, RAG divides 7 by 4 and employs one 4x4 ack-req SA since the quotient is not zero. Then, since the remainder equals 3 which is not a multiple of 4 but instead is a multiple of 3, RAG utilizes one 3x3 ack-req SA after dividing the remainder (=3) by 3. After the calculation for the first level, RAG moves on to the second level. Since there are two ack-req SAs in the first level and there are two levels, RAG can utilizes neither a 4x4 root SA nor 3x3 root SA. Therefore, RAG uses a 2x2 root SA which serves the two ack-req SAs in the first level. □

Algorithm 2 describes the algorithm *Num_In_Level* called by the function *interpret_sa()* as shown in Figure 33. This algorithm calculates the number of levels in the hierarchy and the number of 4x4, 3x3 and 2x2 switch arbiter blocks employed in each level. The number of levels in the hierarchy is determined in the *while* loop (line 6 of Algorithm 2) that divides the number of masters by 4, recursively. The details of this algorithm are described in Algorithm 2 and Example 5.2. Finally, RAG generates a Verilog top file to integrate the selected 2x2, 3x3 and 4x4 switch arbiter blocks into an MxM hierarchical SA or BA. A top file instantiates 2x2, 3x3 and 4x4 switch arbiter blocks based on the data passed from the function, *interpret_sa()*.

**Algorithm 2** Pseudo code for the calculation of number of 2x2, 3x3 and 4x4 switch arbiter blocks and the number of levels in the hierarchy

```
Num_In_Level(number_of_masters)
begin
 1: /*Initialization*/
 2: number_of_levels = 0;
 3: dividend = number_of_masters;
 4: remainder = (dividend modulo 4);
 5: /*calculate the number of levels in the hierarchy*/
 6: while (dividend ! = 0) do
 7:    dividend = floor(dividend/4);
 8:    number_of_levels = number_of_levels + 1;
 9:    if ((dividend == 1) and (remainder == 0)) then
10:       break;
11:    end if
12: end while
13: /*calculate the number of 4x4, 3x3 and 2x2 SAs in each level*/
14: dividend = number_of_masters;
15: for (n = 0 to number_of_levels-1) do
16:    if (dividend>3) then
17:       /* if dividend is a multiple of 4*/
18:       if ((remainder = (dividend modulo 4)) == 0) then
19:          dividend = (dividend/4);
20:          /* 4by4_SA_in_level[n] is equal to the number of 4x4 SAs in level n */
21:          4by4_SA_in_level[n] = dividend;
22:          /* if dividend is a multiple of 3*/
23:       else if ((remainder = (dividend modulo 3)) == 0) then
24:          dividend = (dividend/3);
25:          /* 3by3_SA_in_level[n] is equal to the number of 3x3 SAs in level n */
26:          3by3_SA_in_level[n] = dividend;
27:          /*if number_of_masters is not a multiple of 4 nor 3*/
28:       else
29:          remainder = (dividend modulo 4);
30:          dividend = floor(dividend/4);
31:          4by4_SA_in_level[n] = dividend;
32:          if (remainder == 3) then
33:             3by3_SA_in_level[n] = 3by3_SA_in_level[n] + 1;
34:          else if (remainder == 2) then
35:             2by2_SA_in_level[n] = 2by2_SA_in_level[n] + 1;
36:          else
37:             unserved[n] = 1;
38:          end if
39:       end if
40:    else if (dividend > 2) then
41:       dividend = floor(dividend/3);
42:       3by3_SA_in_level[n] = dividend;
43:    else if (dividend > 1) then
44:       dividend = floor(dividend/2);
45:       2by2_SA_in_level[n] = dividend;
46:    else
47:       unserved[n] = 1;
48:    end if
49:    dividend = 4by4_SA_in_level[n] + 3by3_SA_in_level[n] + 2by2_SA_in_level[n] + unserved[n];
50: end for
end
```

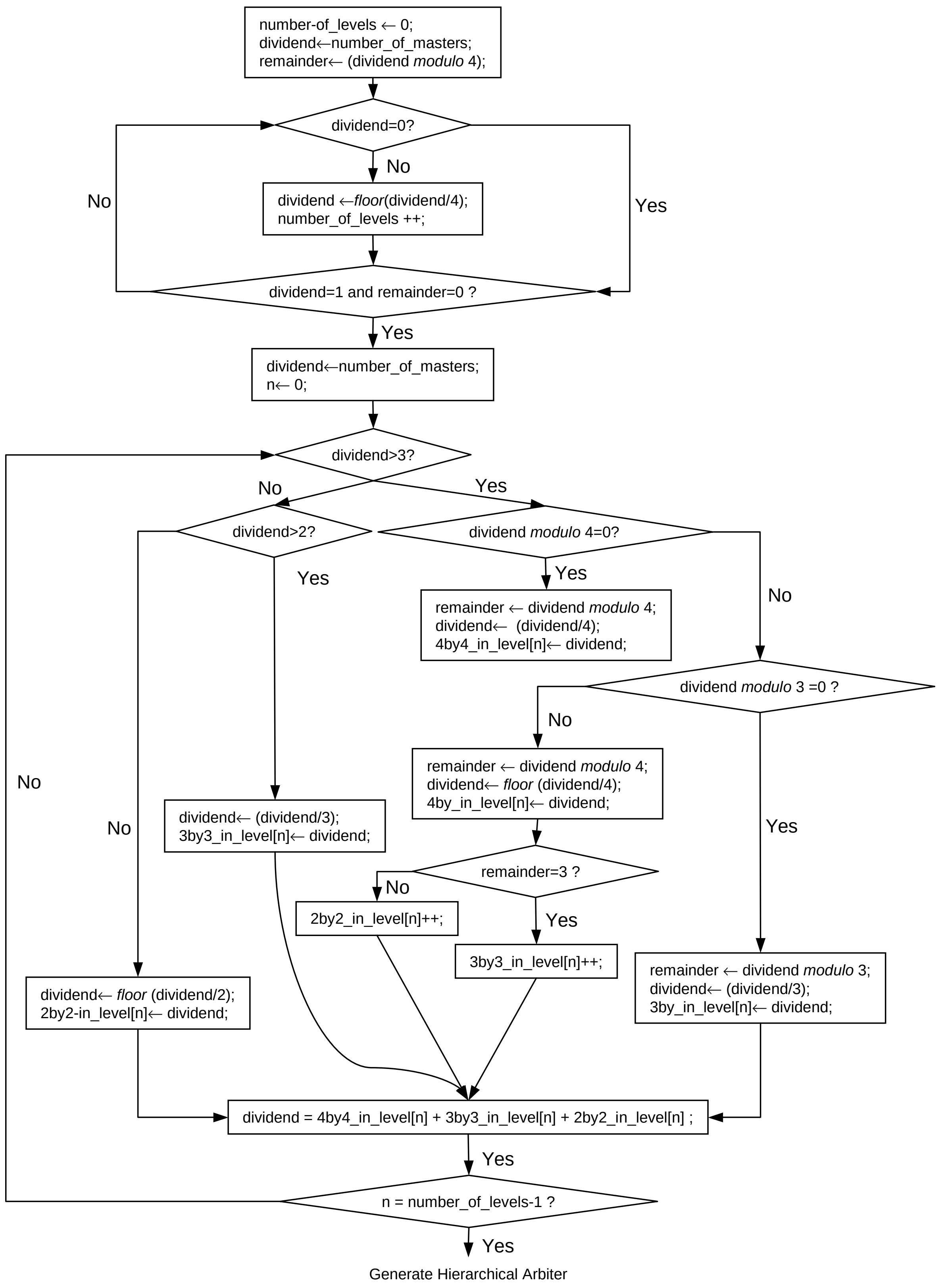

number-of_levels ← 0;
dividend←number_of_masters;
remainder← (dividend *modulo* 4);
dividend=0?
No
dividend ←*floor*(dividend/4);
number_of_levels ++;
No
Yes
dividend=1 and remainder=0 ?
Yes
dividend←number_of_masters;
n← 0;
dividend>3?
No
Yes
dividend>2?
dividend *modulo* 4=0?
Yes
Yes
remainder ← dividend *modulo* 4;
dividend← (dividend/4);
4by4_in_level[n]← dividend;
No
dividend *modulo* 3 =0 ?
No
remainder ← dividend *modulo* 4;
dividend← *floor* (dividend/4);
4by_in_level[n]← dividend;
No
No
dividend← (dividend/3);
3by3_in_level[n]← dividend;
Yes
remainder=3 ?
No
2by2_in_level[n]++;
Yes
3by3_in_level[n]++;
remainder ← dividend *modulo* 3;
dividend← (dividend/3);
3by_in_level[n]← dividend;
dividend← *floor* (dividend/2);
2by2-in_level[n]← dividend;
dividend = 4by4_in_level[n] + 3by3_in_level[n] + 2by2_in_level[n] ;
Yes
n = number_of_levels-1 ?
Yes
Generate Hierarchical Arbiter

**Figure 34:** Flowchart of Algorithm 2

Figure 34 shows the flowchart of Algorithm 2. We explain this flowchart with references to the line number(s) of Algorithm 2. In the flow chart, all variables are initialized to appropriate values (from line 2 to line 4 in Algorithm 2). The first loop in the flow chart implements the *while* loop in the Algorithm 2 (line 6). In the *while* loop, the *number_of_levels* value in a hierarchy is calculated iteratively by dividing *number_of_masters* iteratively by 4.

After the calculation of the number of levels in a hierarchy in the flowchart, *dividend* is set back to be equal to *number_of_masters* (line 14 in Algorithm 2). Then, the *for* loop (line 15 of Algorithm 2) is executed for *number_of_levels* times to calculate the number of 4x4, 3x3 and 2x2 switch arbiter blocks in each level. This *for* loop corresponds to the second loop in Figure 34. In the *for* loop, Algorithm 2 first check if the *dividend* is greater than or equal to 4 (line 16 in Algorithm 2). If a level requires more than one 4x4 switch arbiter, the algorithm checks whether *dividend* is a multiple of 4 or a multiple of 3 (line 18 and line 23, respectively). If *dividend* is a multiple of 4, *4by4_SA_in_level[n]* is set to the outcome of *dividend/4* (line 21). If *dividend* is a multiple of 3, *3by3_SA_in_level[n]* is set to of *dividend/3* (line 26). Otherwise (line 28), the algorithm sets *4by4_SA_in_level[n]* to *dividend/4* (line 31) in order to utilize as many 4x4 switch arbiter blocks per each level. If the *remainder* after division is equal to 3 (line 32), another 3x3 switch arbiter block is employed (line 33) to provide arbitration scheme for the remaining requests (*remainder*). If the *remainder* after division equals 2 (line 34), another 2x2 switch arbiter block is utilized. Otherwise (line 36), one remaining request (*unserved[n]*) for level $n$ is recorded.

**Example 5.2** This example describes the *Num_In_Level()* algorithm called by *interpret_sa()* to show how the number of levels is calculated for a 32x32 hierarchical SA. First, variable *number_of_levels* is initialized to zero (line 2) in Algorithm 2. Line 3 shows that *dividend* is set to *number_of_masters* (32 in this example). The variable *remainder* is initialized to *dividend modulo 4* in line 4. The *while* loop (line 6) iterates three times; (*dividend*=32)/4 is equal to 8 at the first iteration; (*dividend*=8)/4 is equal to 2 at the second iteration; finally, in the third iteration (*dividend*=2)/4 is equal to 0 by the *floor* function. Thus, *number_of_level*s is determined to be 3. After the *while* loop, *dividend* is set back to 32 (line 14). The *for* loop (line 15), iterates as many as the number of levels in the hierarchy. At the first iteration, since *dividend* is greater than 3, the outer *if* block is executed (line 16). Then, since 32 is a multiple of 4, dividend is set to 8 (line 19), which is assigned to *4by4_SA_in_level[0]* where *4by4_SA_in_level[0]* indicates the number of 4x4 switch arbiters needed for level 0. Since *remainder* is 0, no more switch arbiters are needed for level 0. Thus, *3by3_SA_in_level[0]* is equal to zero and *2by2_SA_in_level[0]* is equal to zero. In the second iteration, the outer *if* block is executed again, setting n =1. Next, the *if* block starting at line 18 is executed again and returns *4by4_SA_in_level[1]* as 2, *3by3_SA_in_level[1]* as 0, *2by2_SA_in_level[1]* as 0 and *dividend* as 2. In the last execution, the *elseif* block starting at line 43 is executed since dividend is not greater than 3 nor 2. *2by2_SA_in_level[2]* is set to 1 at line 45. Thus, *Num_In_Level()* determines that eight 4x4 switch arbiters are needed for level 0, two 4x4 switch arbiters are needed for level 1 and one 2x2 switch arbiter is needed for level 2 which is the root. □

After the tool determines the number of 2x2, 3x3 and 4x4 switch arbiter blocks for each level in the tree structure as described by Algorithm 2, the tool will integrate switch arbiters for each level and produce a 32x32 hierarchical SA.

The next example describes how RAG generates a hierarchical SA when M is not a power of two.

**Example 5.3** For an 11x11 hierarchical SA, RAG first determines the number of levels to be 2 in the *while* loop (line 6). The *for* loop (line 15) is executed twice because *number_of_levels* is equal to 2. For the level 0 (first iteration), the *else* block of line 28 is executed because *dividend* (=11) is greater than 3 and it is neither a multiple of 4 nor 3. At line 29, *remainder* is set to 3, and *4by4_SA_in_level[0]* is set to 2 in the following two lines. Since *remainder* is equal to 3, *3by3_SA_in_level[0]* is set to 1 at line 33 (note that *3by3_SA_in_level[0]* is initially set to 0). After the first iteration, *dividend* is set to 3 at line 49. Since *dividend* is equal to 3, in the second iteration the *else if* block (line 40) is executed resulting in setting *3by3_SA_in_level[1]* to 1. Thus, the final outcome of *Num_In_Level()* is that two 4x4 and one 3x3 switch arbiter blocks are needed for a level 0 and one 3x3 switch arbiter block (a root arbiter) is required for level 1 (note that the last highest level – in this case, level 1 – is always a root arbiter). □

Even though the algorithm of our tool is straightforward, RAG nevertheless quickly generates a hierarchical SA according to user specifications. Thus, the result is RTL Verilog for a hierarchical SA of custom size, without requiring rewriting of any Verilog code by hand. Even better, the resulting hierarchical SA arbitrates faster than PPE and PPA each of which need to be customized to a specific configuration by hand.

# CHAPTER VI

# X-GT: CROSSBAR SWITCH GENERATOR FOR MULTIPROCESSOR SOC

This chapter is focused on the design of a CAD tool for the generation of an MxN Crossbar (Xbar) switch. Xbar switch Generator (X-Gt) generates MxN Xbar switch based on user specifications. Figure 35 shows an example of a 4x4 Xbar switch generated by the X-Gt, which consists of four 4x1 switches. All switches directly interface to four PEs. Each switch compares physical addresses from PEs and judge if addresses belong to the address space of the attached memory block.

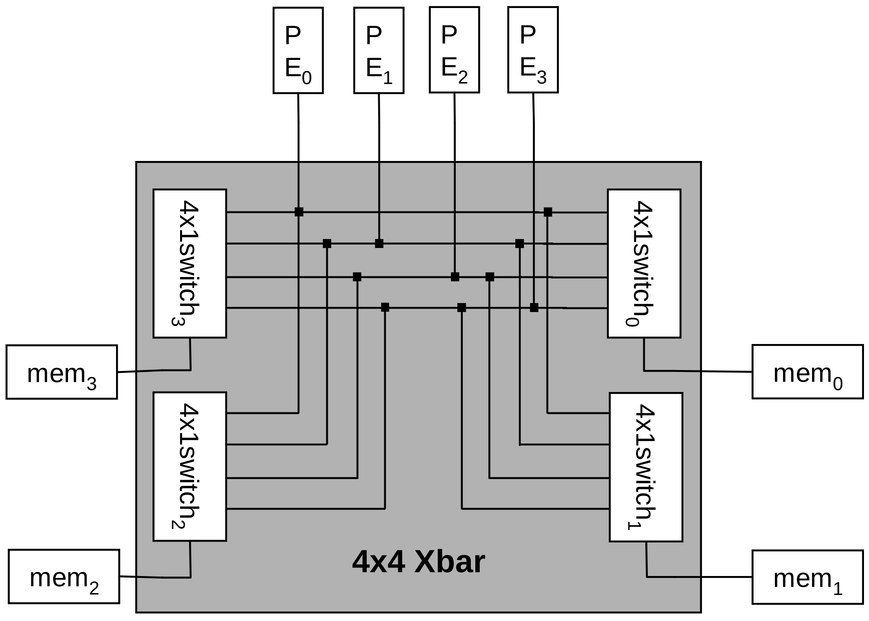


**Figure 35:** The example of 4x4 Xbar with four processors and four memory blocks each with a single port

Later in this chapter, we will show how X-Gt is integrated with an SoC Dynamic Memory Management Unit (DMMU). Shalan *et al.* has designed SoCDMMU [44, 43] and developed its generation tool.

## 6.1 The Xbar

Our generated MxN Xbar switch consists of N Mx1 switches. M is equal to the number of PEs and N equals the number of memory blocks. An Mx1 switch chooses one processor out of M processors to which to grant access to the attached memory block [46]. Figure 36 shows the internal structure of a 4x1 switch; in Figure 36, pe_ indicates wires from a processor. Also, a number appended to the signal name identifies a signal from the corresponding processor. The operation of an Mx1 switch is as follows. First, a comparator (*comp*) compares the M addresses from M processors (*pe_addr0[w-1:0]* through *pe_addr(M-1)[w-1:0]*) only if the corresponding *pe_req[m]* signal is asserted where $0 \leq m < M$ and w = address bus width. If a *pe_addr[m]* input whose *pe_req[m]* is asserted belongs to the address space of the attached memory block, *mem_req[m]* is asserted. The *mem_req[m]* signal asks an arbiter to grant a single bus attached to the corresponding memory block. An MxM hierarchical Bus Arbiter (BA) (see Section 4.3) is employed in an Mx1 switch. Thus, an MxM hierarchical BA handles up to M requests from M processors and grants one request in round-robin order by asserting the appropriate *mem_on[m]* signal. A *mem_on[m]* signal turns on switch blocks (shown in Figure 36 for the 4x1 case) (*addr bus switch*, *data bus switch*, *wire switch*es for read and write signals and *wire_ta* switch for memory transfer acknowledgment) which are Mx1 multiplexers with decoded select (*mem_on[M-1:0]*) signals in order to connect signals from the $m^{th}$ processor to the memory signals.

**Example 6.1** Suppose $PE_0$ and $PE_3$ both try to access $SRAM_0$, $PE_1$ tries to access $SRAM_2$, and $PE_2$ tries to access $SRAM_1$ in Figure 35. Then, pe_addr0 through pe_addr3 from the SoCDMMU are compared in the comparator of 4x1 $switch_0$. Consider Figure 36

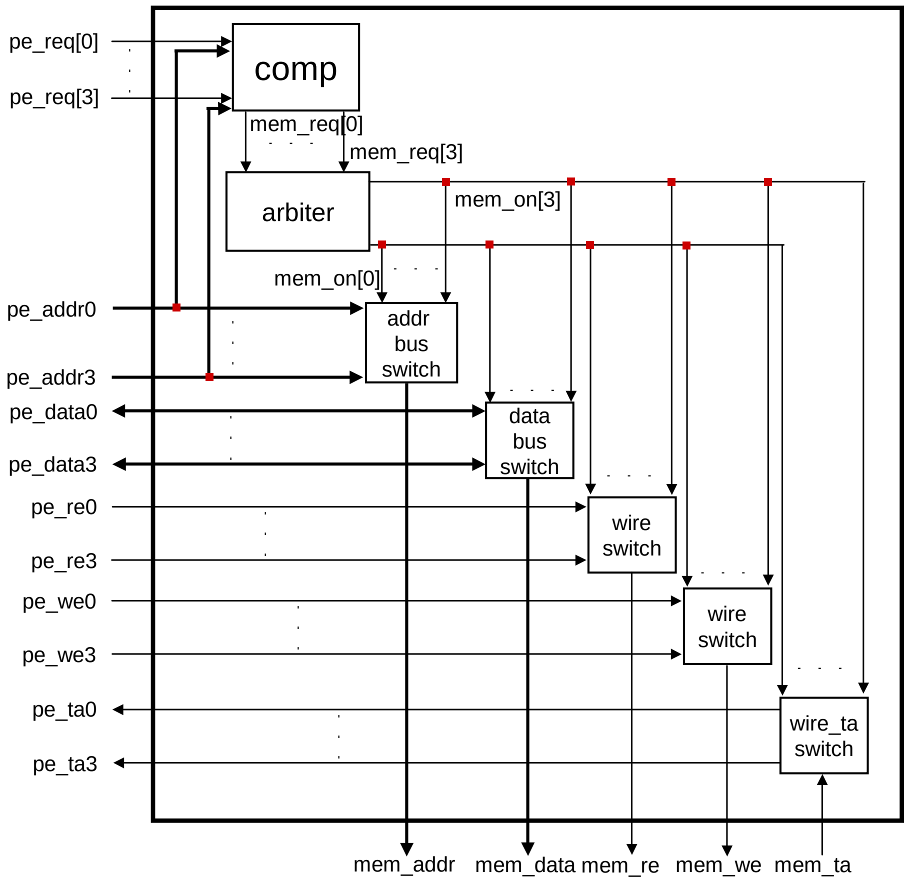


**Figure 36:** Internal structure of a 4x1 switch

to describe 4x1 switch$_0$. Then, in this case as described so far, only pe_addr0 and pe_addr3 are matched to the address space of SRAM$_0$ resulting in the assertion of mem_req[0] and mem_req[3]. Likewise, only mem_req[1] and mem_req[2] are asserted in 4x1 switch$_2$ and 4x1 switch$_1$, respectively. In this case, the arbiter of 4x1switch$_0$ grants the request mem_req[0]; mem_req[3] will be next in round-robin order. Thus, PE$_0$'s request is granted. Then, mem_on[0] turns on the corresponding switch blocks so that pe_addr0 is connected to mem_addr, pe_data0 to mem_data, pe_re0 to mem_re, pe_we0 to mem_we, and pe_ta0 to mem_ta.

At the same time, only mem_req[2] is asserted for 4x1 switch1 since only pe_addr2 is matched to SRAM$_1$. Likewise, only mem_req[1] is asserted for 4x1 switch$_2$ since only pe_addr1 is matched to $SRAM_2$. Thus, $PE_2$'s and $PE_1$'s memory access requests to $SRAM_1$ and $SRAM_2$ are both granted. Thus, in this example, three concurrent memory transfers are supported. □

## 6.2 Methodology

X-Gt processes the user inputs, validates them and generates an MxN Xbar switch in a Register Transfer Level (RTL) Verilog Hardware Description Language (HDL). The following is a list of the user specified parameters:

- The number of PEs that determines M in an MxN Xbar
- The number of the global on-chip memory blocks that determines N in an MxN Xbar
- The sizes of the global on-chip memory blocks that determine the address bus widths between Mx1 switches and memory blocks
- The total memory size that determines address bus widths between PEs and Mx1 switches
- The PE types that determine data bus width of Mx1 switches

The Xbar hardware is generated by calling the function *gen_xbar()*. The function *gen_xbar(bus parameters)*, as shown in Algorithm 3, generates N (number of memory blocks) Mx1 switches, where M is equal to the number of processors

The function *gen_xbar()* makes use of a linked-list to store the wire names for processors, memory blocks and bus lines. To generate an MxN Xbar, first the function *gen_xbar()* calls the function *gen_Mx1()* N times to generate N Mx1 switches; then, *gen_xbar()* integrates these N Mx1 switches (submodules in Verilog) into an Xbar by generating a top file.

To generate Mx1 switches, the function *gen_Mx1()* first fills the wire names linked-list with the processors' wire names, the memory blocks' wire names and the bus lines' wire names by calling the functions: *gen_proc_wires()*, *gen_mem_wires()* and *gen_bus_wires()*, respectively. Then, *gen_Mx1()* invokes a set of functions to generate address bus switches, data bus switches, wire switches and wire_ta switches (defined in Section 6.1) in Figure 36. These switches are hand-coded beforehand. Finally, *gen_Mx1()* calls *gen_arbiter(M)* to generate the appropriate arbiter using the Algorithm 2 described in Chapter 5. Note that specific user input for the arbiter is not required because the arbiter type is set to a bus arbiter and the number of masters is set to M as shown in Chapter 5.

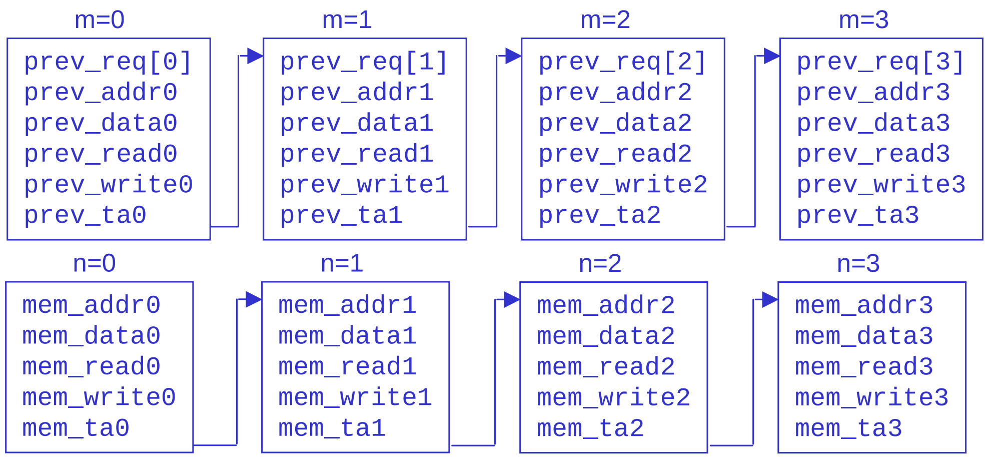


**Figure 37:** Linked-list data structure for Example 6.2

**Example 6.2** Suppose a user wants to build a system with two MPC750s and two ARM9TDMIs and four memory blocks with sizes 2Mbytes, 2Mbytes, 4Mbytes and 8Mbytes. X-Gt first determines the data bus widths to be 64 bits for MPC750 and 32 bits for ARM9TDMI. Also, the address bus widths are set to be 32 bits for both processors. The memory address bus widths of the four memory blocks are set to 21 bits, 21 bits, 22 bits and 23 bits. The order of memory blocks attached to the generated Xbar is clock-wise as shown in Figure 35.

**Algorithm 3** X-Gt Pseudo Code for the MxN Xbar Generation

```
gen_xbar(bus parameters)
begin
 1: n=0;
 2: while (n < N) do
 3:     gen_Mx1(bus parameters);
 4: end while
 5: integrate();
end

gen_Mx1(bus parameters)
begin
 1: gen_proc_wires(M);
 2: gen_mem_wires(N);
 3: gen_bus_wires(M);
 4: gen_addr_bus_switch(M);
 5: gen_data_bus_switch(M);
 6: gen_wire_switch();
 7: gen_wire_ta_switch();
 8: /* This function is the subset of RAG for a hierarchical BA generation. The detailed description
    of RAG is available in Chapter 5*/
 9: gen_arbiter(M);
10: gen_comp();
end

gen_proc_wires(M) begin
 1: m = 0;
 2: while m < M do
 3:     fill in data structure for processor wire names;
 4: end while
end

gen_mem_wires(N) begin
 1: n = 0;
 2: while n < N do
 3:     fill in data structure for memory wire names;
 4: end while
end

gen_bus_wires(M) begin
 1: m = 0;
 2: while m < M do
 3:     fill in data structure for processor wire names;
 4: end while
end
```

Next, X-Gt calls *gen_xbar()*. The function *gen_xbar()* calls *gen_Mx1()* which, as shown in Algorithm 3, fills the data structure for wire names by *gen_proc_wires()*, *gen_mem_wires()* and *gen_bus_wires()*. Figure 37 illustrates how wire names are stored in the data structure by the linked-list. Then, *gen_Mx1()* generates the switch blocks shown in Figure 36. *gen_Mx1()*, as seen in Algorithm 3, also invokes RAG as described in Chapter 5 to generate an arbiter handling 4 requests and generating a comparator comparing 4 addresses. The arbiter parameters (the number of requestors and the arbiter type) are passed to RAG. All generated submodules are connected together by wire names in *gen_Mx1()*. For example, pe_req signals are input signals to an Xbar, the top module, from an SoCDMMU. Also, pe_req signals are input signals to four 4x1 switches and are input signals to the comparator as well, the submodule of 4x1 switches as shown in Figure 36. Thus, pe_req signals are connected to all four 4x1 switches (submodules of an Xbar) in the top module and are connected to a comparator (the submodule of a 4x1 switch) in each 4x1 switch. In this way, the same wire names are connected. After $M = 4$ iterations, four 4x1 switches are generated with corresponding bus parameters. Finally, the function integrate() creates the top file so that the generated four 4x1 switches are wired together by bus wire names. $\square$

## 6.3 Integration with DMMU and its generation tool

We combine the crossbar (Xbar) generation tool (X-Gt)with Dynamic Memory Management (DMMU) generation tool developed by Mohamed Shalan [44, 43]. We name this tool the **D**ynamic memory management unit-**X**bar **G**enera**t**or (DX-Gt). The first contribution of DX-Gt is to provide an automatic generation of SoC Dynamic Memory Management Unit (SoCDMMU). The second contribution is the automatic generation of a crossbar (Xbar) switch. The Xbar generation is also integrated with an arbiter generation tool [46]. Both SoCDMMU and Xbar are generated in an RTL Verilog HDL. Our generated Silicon Intellectual Properties (SIPs) (SoCDMMU and Xbar) require interfaces for masters and slaves. We assume that the effort to connect a new SIP core to the custom SIP generated by DX-Gt is almost the same as that of SIP based design such as adding other SIP cores to CoreConnect.

### 6.3.1 Target Architecture

As shown in Figure 38, our target SoC architecture consists of multiple Processing Elements (PEs) of various types (i.e., general purpose processors, domain-specific CPUs such as DSPs, and custom hardware), large configurable global on-chip memory blocks and the SoC Dynamic Memory Management Unit (SoCDMMU) to manage the memory allocation and deallocation among the PEs. An SoCDMMU remaps processor addresses (virtual addresses) to physical addresses which are passed to Level 2 (L2) on-chip memory via an MxN Xbar. To achieve the maximum concurrent transfers between processors and memory blocks, N should be equal to M. The memory coherency problem due to concurrent accesses of global memory blocks is reduced by using the SoCDMMU [44]. The combination of the SoCDMMU and the Xbar for a multiprocessor SoC showed an overall speedup of 4.4X during application transition time when compared to a fully shared memory system with the same memory organization and number of processors [43].

Our SoC configuration tool can generate an architecture like that of Figure 38 with any number of processors and any number of memory modules of different types (e.g., SRAM and DRAM) and different numbers of ports. DX-Gt automatically configures both the SoCDMMU and the MxN Xbar. Currently, DX-Gt only supports two kinds of processors: MPC750 and ARM9TDMI. However, DX-Gt can be easily extended to additional processors. The SoCDMMU and the Xbar have a generic bus interface.

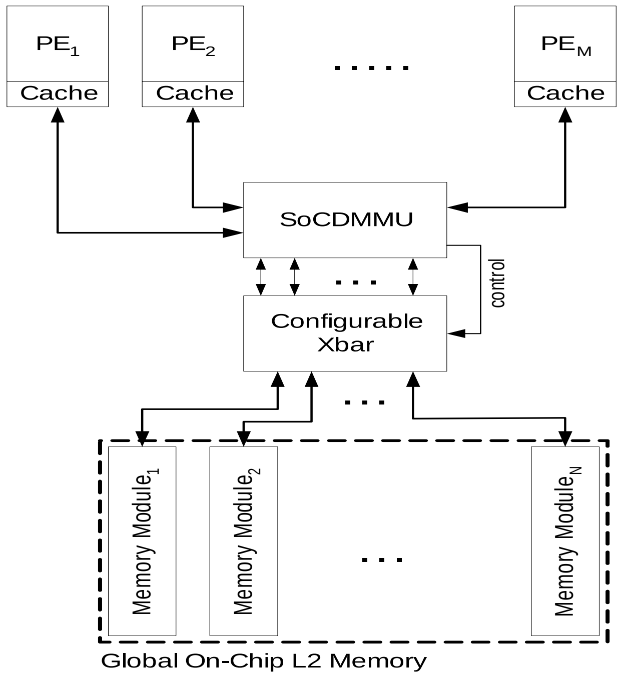


**Figure 38:** The SoC target architecture

Each PE in our target architecture has a wrapper that converts the PE's bus interface signals into the SoCDMMU and Xbar generic bus interface signals. Supporting a new PE is just a matter of developing a new wrapper for the PE's bus interface (which is an easy task).

Figure 39 shows a system that is generated by DX-Gt. Note that four 4x1 switches in Figure 39 are grouped together into the configurable Xbar (4x4 Xbar in this case) in Figure 38. A PE is connected to four 4x1 switches via an SoCDMMU (1-to-4 connections) and sends an address, data and controls (read, write and byte selection) to all four 4x1 switches at the same time. Each 4x1 switch translates each address to check if the address belongs to the corresponding address space of the attached SRAM. The system consists of four processors and four single port SRAM modules. Each processor block in Figure 39 can be either MPC750 or ARM9TDMI depending on the user input. In Figure 39, the generated 4x4 Xbar switch, which consists of four 4x1 switch blocks, provides four concurrent accesses to four SRAM modules by four

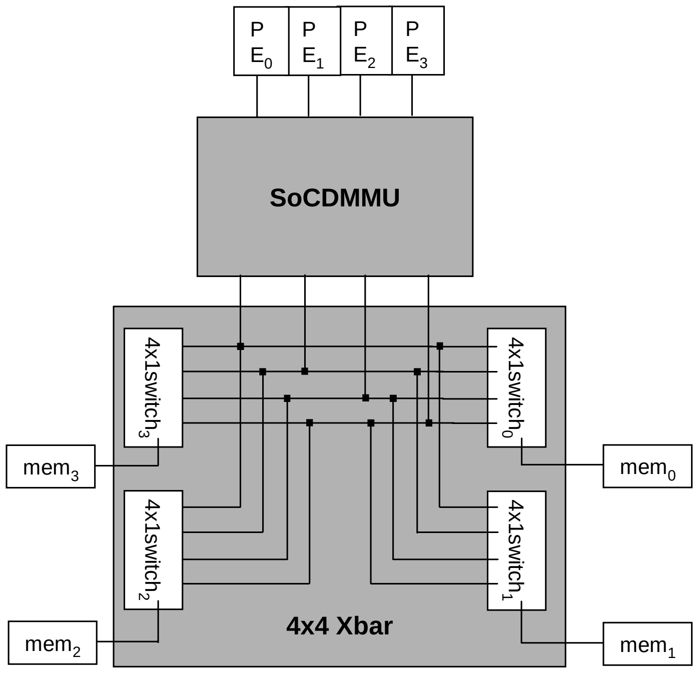


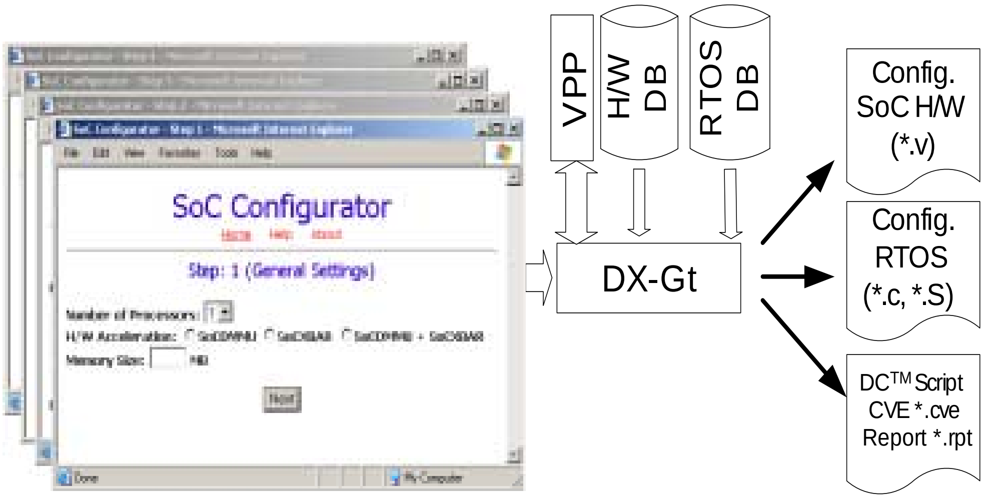
**Figure 39:** The target architecture of four processors and four memory blocks each with a single port

processors. The generated SoCDMMU manages the dynamic allocation/deallocation of memory in the four SRAM modules.

The target architecture runs the Atalanta Real-Time Operating System (RTOS) which is an open source RTOS developed at the Georgia Institute of Technology for a shared memory multiprocessor SoC [49]. DX-Gt can configure Atalanta to support the SoCDMMU – if used – and tune its different modules to reflect the user settings.

### 6.3.2 Tool Integration

Figure 40 gives an overview of the flow of our configuration tool. A Graphical User Interface (GUI), which consists of a set of HTML forms, captures the user's inputs and passes them to the **D**ynamic memory management unit and crossbar (**X**bar) switch **G**enera**t**or (DX-Gt) application (developed in C-Language). DX-Gt processes the user inputs, validates them and generates the SoC hardware files in an RTL Verilog.



Step: 1 (General Settings)

Number of Processors: 1

H/W Acceleration: SoCDMMU SoCDDAU SoCDMMU + SoCDDAU

Memory Size: MB

**Figure 40:** The SoC configuration tool flow

Moreover, DX-Gt generates Synopsys DC [51] synthesis scripts and a Mentor Graphics Seamless CVE [31] configuration file for simulation of the resulting SoC design.

The following is a partial list of the user specified parameters required for DX-Gt other than parameters for X-Gt specified in Section 6.2 :

- System wide parameters:
  - The number of PEs which determines M in an MxN Xbar and the number of the SoCDMMU ports
  - The number of the global on-chip memory *G_blocks* which determines the size of the SoCDMMU memory allocator
  - The sizes of the global on-chip memory *G_blocks* which determines the address bus widths between Mx1 switches and memory blocks
  - The number of memory ports which determines N in an MxN Xbar
  - The PE types which determines processor interfaces to SoCDMMU chosen from a hardware database
  - The memory type which determines the memory controller chosen from a hardware database

  - The choice of use of SoCDMMU, Xbar, both or none

- SoCDMMU related parameters:

  - The scheduling scheme to resolve concurrent memory requests from different PEs (first come first served scheme or priority scheme)
  - Memory *G_blocks* initially assigned to the PEs (initial memory assignment for the PEs)

In order to generate the hardware files, a "hardware database (HW DB)" of parameterized Verilog files of each system component – SoCDMMU sub-modules, processor bus wrappers, memory controller and Xbar switches and comparators in Figure 36 – is being used. The Xbar arbiter in Figure 36 is generated in a way that described in Chapter 4 and [46]. The Verilog files in the database are written in such a way that a custom version of the file can be generated using a Verilog PreProcessor (VPP).

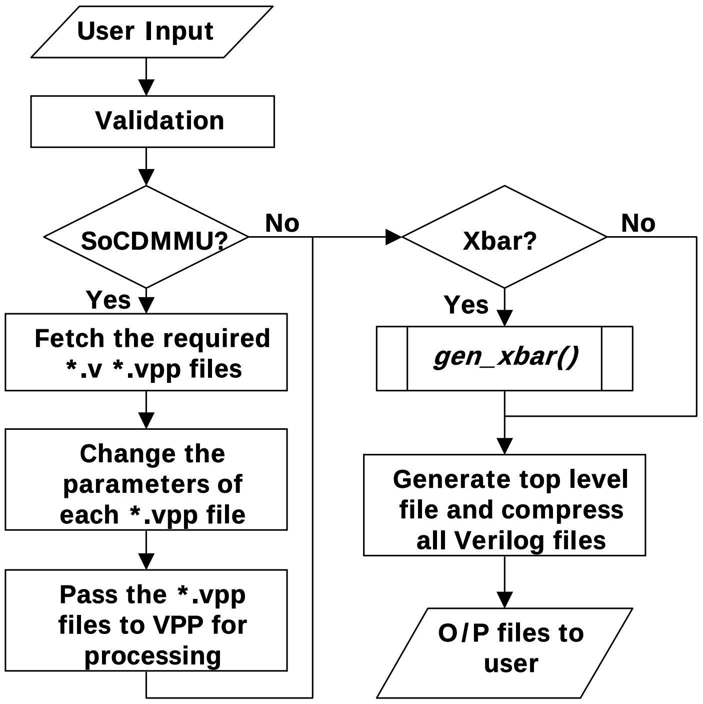

User Input
Validation
SoCDMMU?
No
Yes
Fetch the required *.v *.vpp files
Change the parameters of each *.vpp file
Pass the *.vpp files to VPP for processing
Xbar?
No
Yes
gen_xbar()
Generate top level file and compress all Verilog files
O/P files to user

**Figure 41:** Flowchart of DX-Gt

Once the user configurations and settings are captured using a set of HTML forms, DX-Gt selects from the database the hardware components that satisfy the user specified configurations. Next, DX-Gt sets the parameters of each component to reflect the user input. The hardware components (preprocessed Verilog files) are passed to the Verilog PreProcessor (VPP) [62] which processes them and generates new customized Verilog files. The detail use of VPP is described in [45]. Figure 41 shows the flowchart of DX-Gt.

We synthesize an Xbar in an RTL Verilog HDL, the output of the function *gen_xbar()* to report areas and delays. We also backward annotate Resistances and Capacitances of interconnect wires for the more accurate timing analysis. These results are presented in Chapter 7.

# CHAPTER VII

# EXPERIMENTAL RESULTS

This chapter consists of arbiter and Xbar experimental results. In Section 7.1, we consider area, delay and power dissipation of our hierarchical Bus Arbiter (BA) and Switch Arbiter (SA). We also show how our hierarchical SA achieves terabit switching and show various speedups our SA achieves over the-state-of-the-art arbiters. In order to prove that the lack of perfect fairness in arbitration does not significantly affect typical network traffic, we simulate our hierarchical SA with uniform, bursty and Transport Control Protocol (TCP) traffics.

For an Xbar, we report area and delay of an Xbar generated by X-Gt in the second subsection, Section 7.2. In order to report accurate delay, we perform back annotation by extracting Resistance and Capacitance (RC) values after layout.

## 7.1 *Arbiter Experiment*

In this section, we compare area and delay between a Ping-Pong Arbiter (PPA) [7], a Programmable Priority Encoder (PPE) [13] for iSLIP [29], and our generated Switch Arbiter (SA). Since PPA outperforms DRRM as mentioned in [7], we do not consider including dual round-robin matching (DRRM) algorithm in this comparison. Thus, we will show a speedup for our generated SA over a PPA and a PPE.

In Section 7.1.1, we compare areas and delays of a hierarchical arbiter versus a centralized arbiter to explain why we limit switch arbiter blocks to 2x2, 3x3 and 4x4 in RAG. In Section 7.1.2, we explain areas and delays of the three switch arbiters. Then, in the following section, we show speedups achieved by our 32x32 hierarchical SA generated by RAG for a 32x32 network switch. Finally, in Section 7.1.5, we present simulations which show that the lack of perfect fairness of a switch arbiter is not a problem under typical TCP/IP workloads.

### 7.1.1 BA Area and Delay Considerations

We synthesized different configurations of BAs and hierarchical BAs using the Synopsys Design Compiler [51] with a TSMC 0.25$\mu$m library [60] from LEDA Systems [23]. Figure 42 and Figure 43 show the area and delay, respectively, of BAs and hierarchical BAs. The area increases more than linearly as the number of masters (M) increases for a BA. As can be seen in Figure 43, the delay of a BA increases linearly as M increases, while the delay of a hierarchical BA increases less than linearly. The area and delay gaps between a BA and a hierarchical BA increase as the number of masters increases.

As shown in Figure 43, we found that limiting the size of the switch arbiter blocks (ack-req BAs and root SAs, for terminology clarification please see Chapter 2) used as the components of a hierarchical BA to 2x2, 3x3 and 4x4 yielded the fastest arbitration speeds. From this result, we use 2x2, 3x3 and 4x4 switch arbiter blocks for a hierarchical SA as well. Our RAG tool favors the utilization of 4x4 switch arbiter blocks rather than 2x2 and 3x3 switch arbiter blocks. Employing a 4x4 switch arbiter block gives 16% area savings and 36% gate delay reduction compared with using three 2x2 switch arbiter blocks (two for leaves and one for a root to implement a 4x4 switch arbiter block, see Figure 22 in Chapter 4).

### 7.1.2 SA Area and Delay Comparisons

In order to estimate area and delay, we synthesize our SAs generated by RAG using a 0.25$\mu$m TSMC standard cell library [60] from LEDA Systems [23]. For PPA, we modeled PPA by writing Verilog code based on the PPA logic diagram in [7] and synthesized PPA to estimate the area and delay using the aforementioned 0.25$\mu$m TSMC standard cell library from LEDA Systems. PPE reports results for synthesis

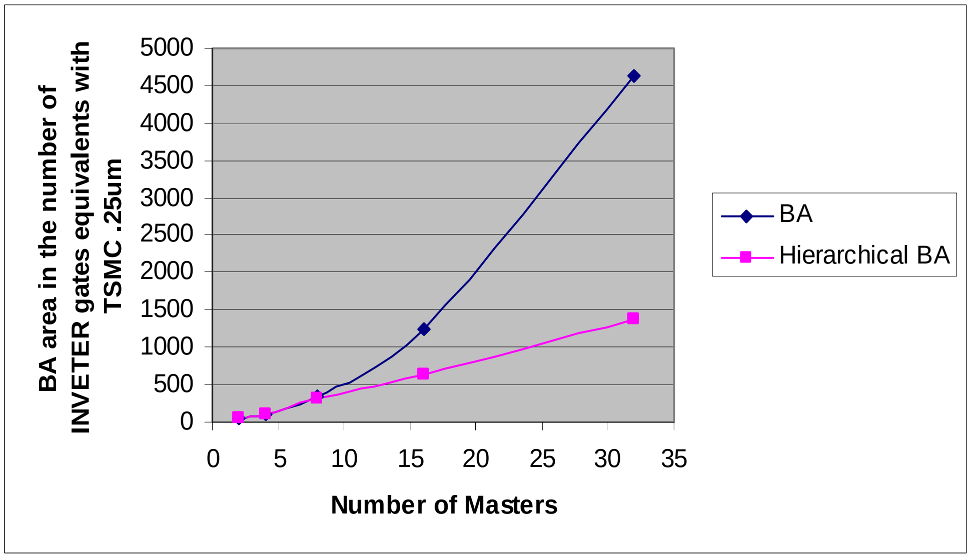


**Figure 42:** MxM Bus Arbiter area

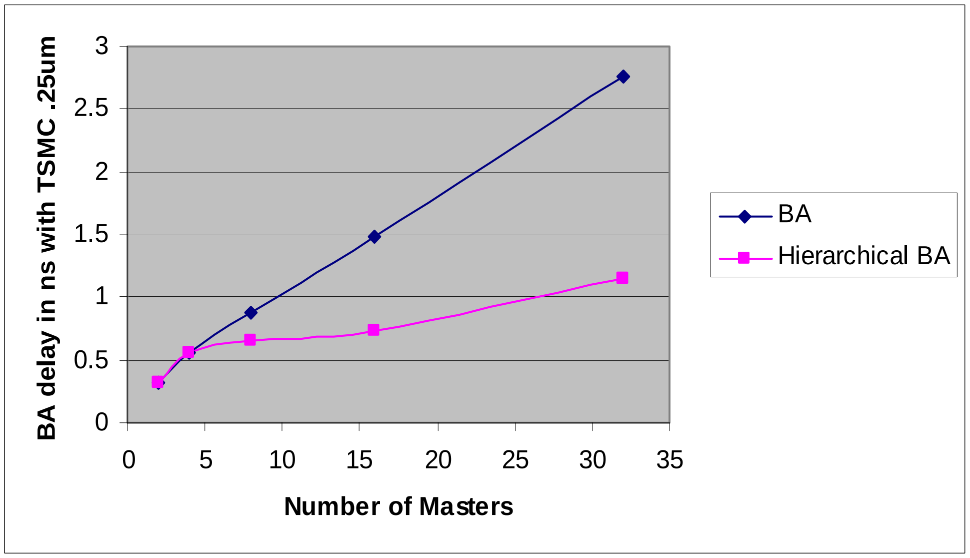


**Figure 43:** MxM Bus Arbiter delay

using Texas Instruments TSC5000 0.25$\mu$m technology; thus, for the area of PPE we simply report the area results stated in [13]. However, for PPE delay estimation we use the same library with which our SA and PPA are synthesized; thus, we modeled the priority encoder and additional logic shown for PPE in Figure 11 of [13] to measure the delay of PPE with TSMC 0.25$\mu$m technology. The delay we measured is well matched with Table 2 in [13] for PPE except for the case where M=32 (32% longer delay in our experiment)[1]. In all cases the Synopsys Design Compiler [51] is used.

Figure 44 and Figure 45 show the area and the longest switch arbiter logic delay for SA, PPE and PPA, respectively, for varying values of M equal to a power of two. PPA uses a 2x2 switch arbiter as a basic switch arbiter block. PPA applies 2x2 switch arbiters to a binary tree structure to form an MxM hierarchical SA. Whenever one master (say, *req0[0]*) is granted by a 2x2 switch arbiter, the other master (*req0[1]*) has the highest priority for the next cycle. The treatment of upper-level grant signals in the binary tree is similar to SA in Figure 23. The major difference is that every 2x2 switch arbiter of PPA receives acknowledgments from two higher levels when $M > 4$ and ANDs them together with the current level grants. If $M \leq 4$, PPA receives acknowledgment from at most one higher level.

In [7], Chao *et al.* compare the throughput of PPA with that of iSLIP [29]. The performance of PPA is very competitive with speedup $c = 2$ for both uniform and burst traffic as shown in Figure 2 in [7]. The speedup $c$ of the switch fabric is the ratio of the switch fabric bandwidth to the input link bandwidth. In this section, we first compare the area and the longest delay of our hierarchical SA with those of PPA and PPE for iSLIP in different MxM configurations. In Section 7.1.4, we compare the throughput of our hierarchical SA, PPA and PPE for a 32x32 network switch.

[1]We are not sure exactly why this difference exists for M=32. But it may have to do with differences in the TSC5000 0.25$\mu$m process and/or standard cell library used versus the TSMC 0.25$\mu$m process and/or standard cell library used. Unfortunately, we lack access to TSC5000.

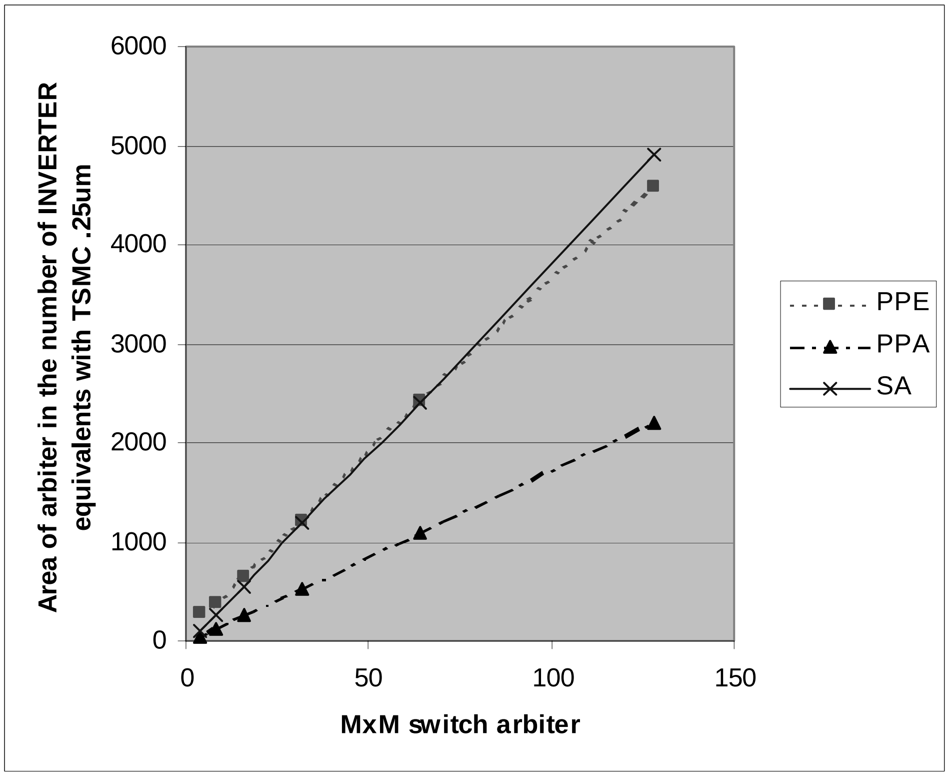


**Figure 44:** MxM Switch Arbiter area

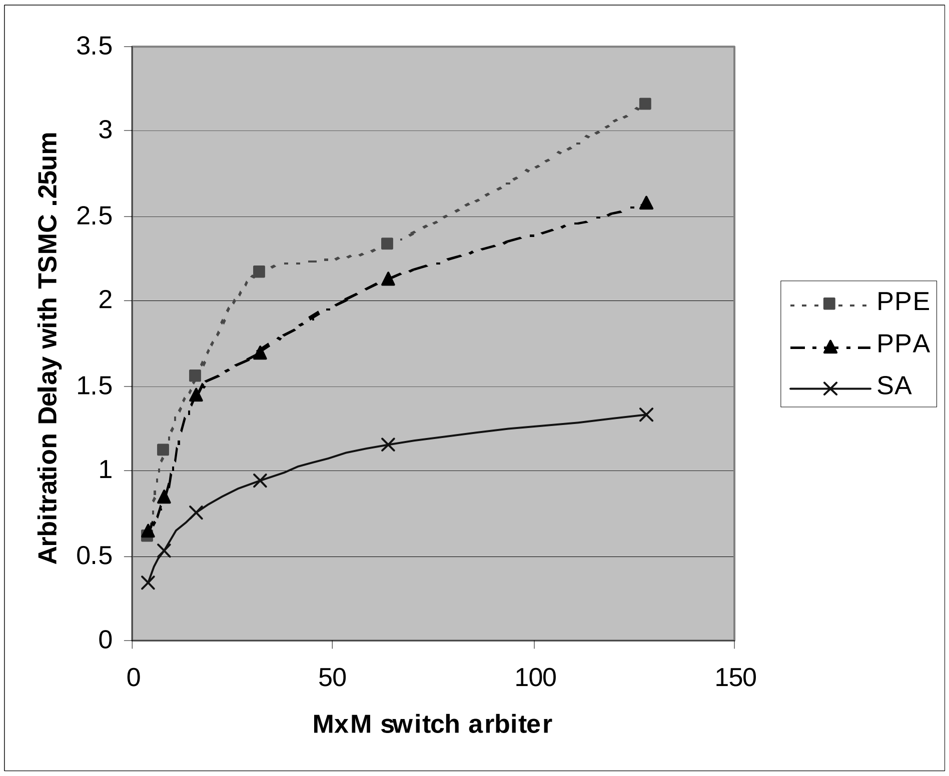


**Figure 45:** MxM Switch Arbiter longest delay

The areas of all three switch arbiters increase linearly as M increases. The area of SA is almost the same as that of PPE and increases more rapidly than PPA. However, SA shows the shortest logic delay when compared with PPA and PPE, and the logic delay increases the slowest (compared with PPA and PPE) as M increases. This is because we first limit the size of switch arbiter blocks to 2x2, 3x3 and 4x4 to reduce the critical path delay due to the expansion of priority logic blocks as M increases and apply the 2x2, 3x3 and 4x4 switch arbiter blocks to a distributed structure so that the number and delay of the switch arbiter block(s) in the critical path is minimized. Since PPE is a centralized switch arbiter, its delay is longest. Even though PPA is also a distributed switch arbiter, it has more levels than SA causing more gate delays to connect switch arbiters in different levels.

### 7.1.3 Power Dissipation of the Arbiters

We can classify a power dissipation into i) static power and ii) dynamic power [6]. A gate dissipates static power when it is not switching (inactive). The large portion of static power dissipation is from source-to-drain subthreshold leakage resulting from the reduction of threshold voltage which prevents a gate from being completely turned off. Static power is also dissipated from the current leakage between the diffusion layers and substrate. On the other hand, dynamic power is dissipated when the circuit is active. The dynamic power consists of switching power and internal power. The switching power is dissipated by charging and discharging of a load capacitance at the output of the cell. Internal power is dissipated by charging and discharging of any internal capacitances of the cell. Internal power also includes power dissipation of momentary short-circuit power between P and N transistors during signal transition.

Figure 46 illustrates the methodology we take for the power estimations of arbiters. First, an RTL Verilog file is read by the Synopsys Design Compiler [51]. Then, the Synopsys Power Compiler [53] creates a power model with the assigned technology

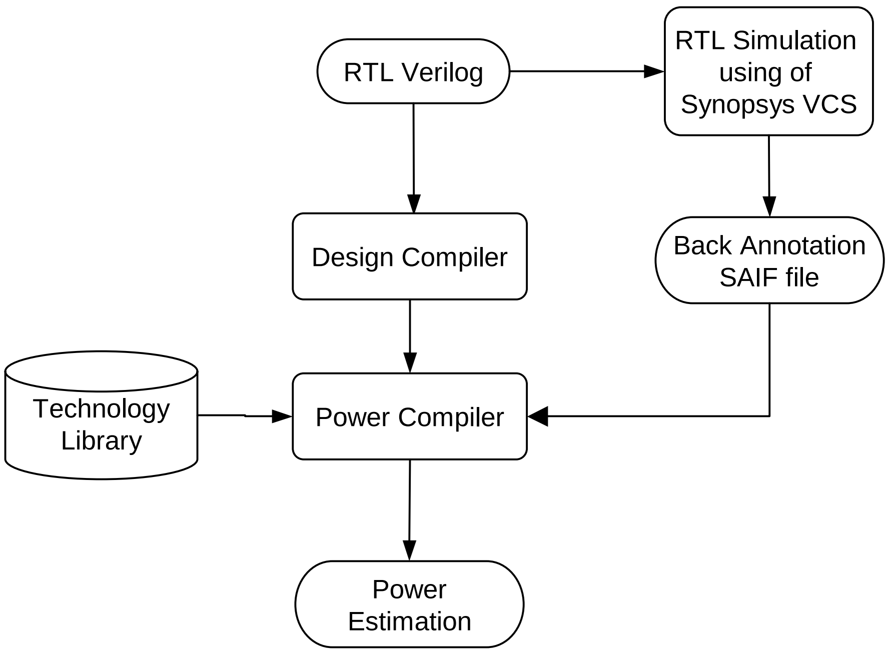


**Figure 46:** Methodology of power estimation

library mapping. We use a TSMC 0.25 $\mu$m standard cell library [60] from LEDA Systems [23] for the power estimation. In order to capture switching activity, an RTL Verilog file is simulated using Synopsys VCS [54] with stimulus input. The switching activities of arbiters are captured by VCS and are saved to a Switching Activity Interchange Format (SAIF) file.

#### *7.1.3.1 Hierarchical BA vs. BA*

In addition to area and delay comparisons in Section 7.1.1, we estimate power dissipation and compares power dissipation of hierarchical Bus Arbiters (BAs) with those of centralized BAs. For this estimation, we assert all request inputs of the MxM arbiters and randomly generate an acknowledgment signal so that the granted bus master can be an owner of the bus for a random number of cycles. Under this assumption, an SAIF file is generated by Synopsys VCS. The SAIF file is backward annotated to the Synopsys Power Compiler.

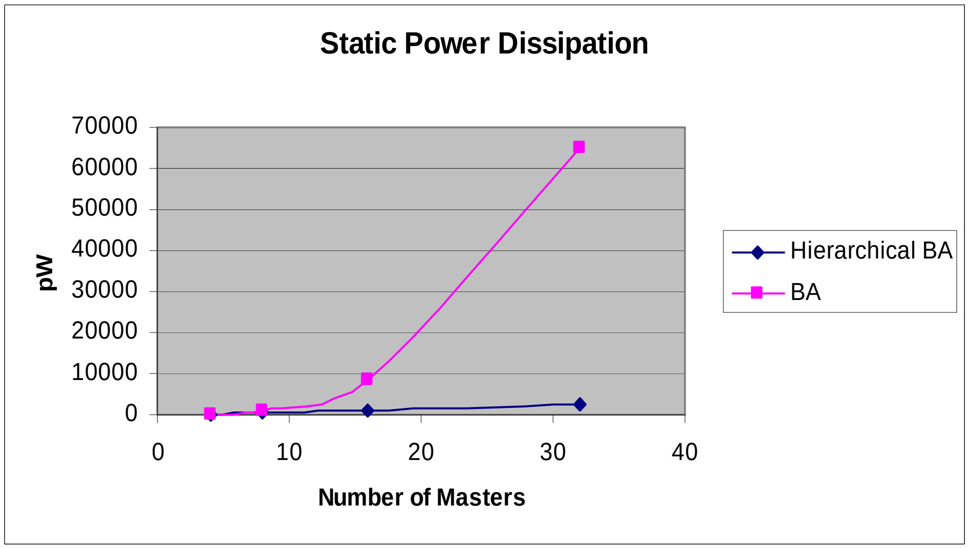


**Figure 47:** MxM hierarchical BA and BA static power dissipation

Figure 47 shows the static power dissipation of (flat, non-hierarchical) BAs and hierarchical BAs from $M = 4$ to $M = 32$. Note that the 4x4 BA design is exactly same as those of hierarchical BAs. Since the static power dissipation is proportional to area, our hierarchical BA, which uses far less area compared with BA, dissipates much less static power compared with BA.

However, a BA, regardless of its larger area compared with a hierarchical BA, dissipates less dynamic power as shown in Figure 48. This is because a hierarchical BA has more switching activity – switches more transistors – than a BA. In order words, "work done" by a hierarchical BA per second is larger than that by a BA per second.

Figure 49 shows total power dissipation of BA versus hierarchical BA. This figure is almost same as the plot of dynamic power dissipation since static power dissipation is very small (on the order of about $10^{-6}$) compared with the portion taken by dynamic power dissipation for TSMC 0.25 $\mu$m technology.

Since the power consumed by a device is equivalent to the "work done" by a device per second and energy consumption is important for portable and embedded

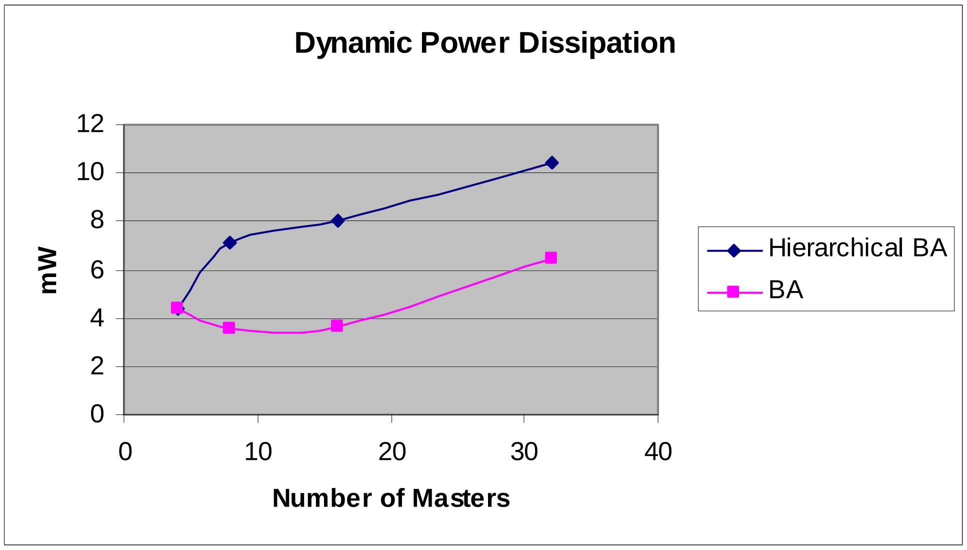


**Figure 48:** MxM hierarchical BA and BA dynamic power dissipation

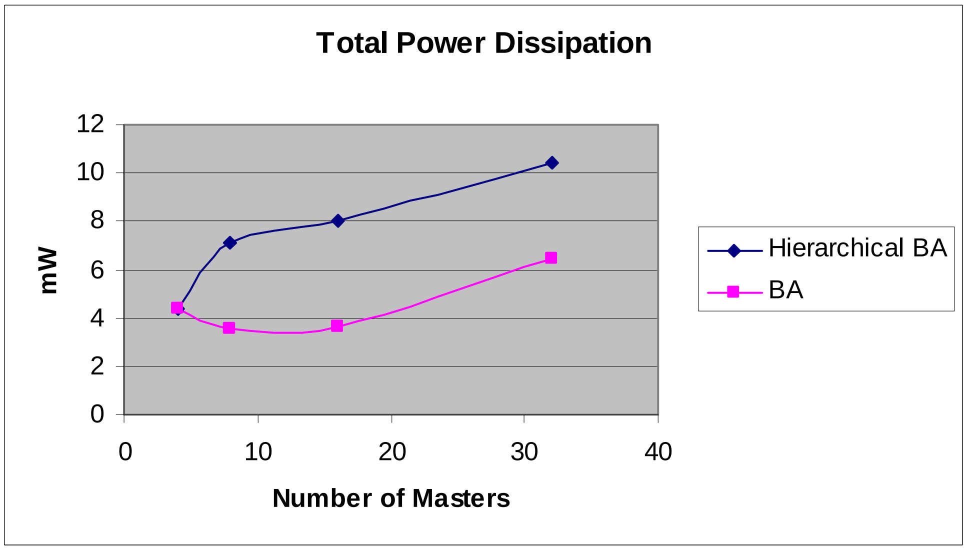


**Figure 49:** Total power dissipation of BA versus hierarchical BA

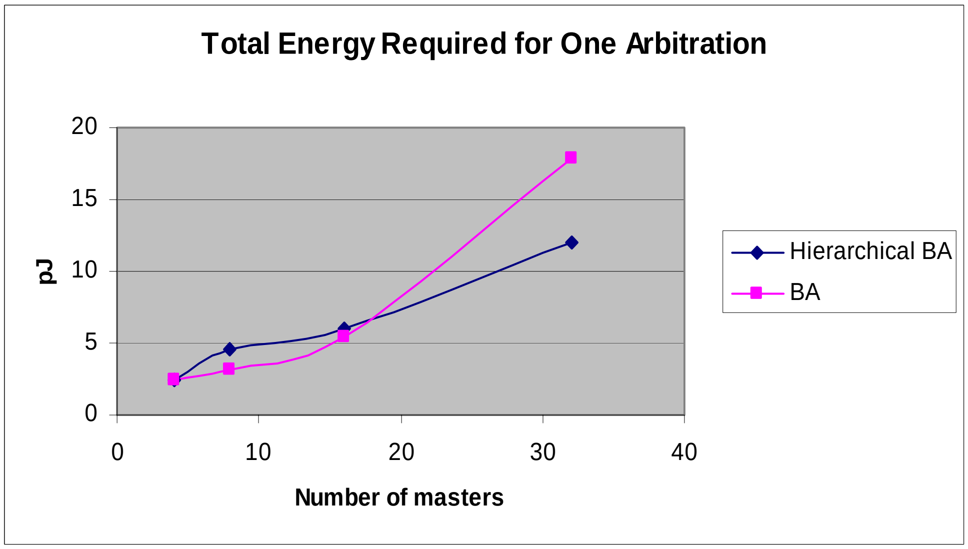


**Figure 50:** Total energy required for one arbitration of BA versus hierarchical BA

processors because a battery life is a key concern, it is important to compare energy consumption as well as power consumption [4]. Note that the overall energy consumption equals the power dissipation multiplied by the execution time (Figure 43). Figure 50 shows an energy required for one arbitration of BA versus hierarchical BA. Energy consumption increases as M increases for a BA and a hierarchical BA. The energy consumed by a hierarchical BA becomes less than that by a BA when M is larger than 16.

#### *7.1.3.2 Hierarchical SA vs. PPE and PPA*

In this section, we compare power dissipation of our hierarchical SA with those of PPE and PPA for values of M from 4 to 128. We estimate PPE's power dissipation with a Priority Encoder preceded by priority rotation logic implemented with ring counters (note that while this captures PPE's functionality, we are not sure if this is the exact actual PPE logic since the detailed logic design is not available in [13] nor in any subsequent publications [12, 14, 16, 17, 21, 36]). The output of a ring counter (only one output equals '1' at a time for an M-bit ring counter) is bitwise ANDed with request inputs. In this scheme, only one request input (the request pointed to by the pointer) to a priority encoder is asserted.

To capture a SAIF file, we utilize stimulus in which all input request signals are asserted continuously since it is reasonable to estimate power dissipation under the condition that all inputs of a network switch are saturated. This SAIF file is backward annotated to Power Compiler to have more accurate switching power estimation. After Power Compiler reads a SAIF file, power dissipation is estimated with separate reporting of static power and dynamic power dissipation.

We plot static power and dynamic power dissipation separately since the portion taken by static power dissipation of a hierarchical SA, PPE and PPAs is too small (on order of $10^{-6}$) compared with dynamic power dissipation in TSMC 0.25 $\mu$m technology. Figures 51 and 52 show static power dissipation and dynamic power dissipation, respectively. Static power dissipation increases linearly as M increases in the same way that the areas of hierarchical SA, PPE and PPA increase. Since our hierarchical SA takes more area than PPA as shown in Figure 44, our hierarchical SA dissipates more static power than PPA as expected. The static power dissipation of PPE is almost equal to that of PPA.

However, the increase in dynamic power dissipation depends both on clock speed and switching activity of an arbiter. Thus, the dynamic power dissipation does not increase linearly as M increases. Apparently, a faster clock speed which is determined by the logic delay of an arbiter results in more frequent occurrences of internal switching activity per second. However, our hierarchical SA shows much less switching activity compared with PPA and PPE even though our hierarchical SA operates much faster than PPA and PPE. Our hierarchical SA consists of 2x2, 3x3 and 4x4 switch arbiter blocks. In each switch arbiter blocks, there are multiple Priority Logic blocks. For the case of a 4x4 ack-req SA, there are four Priority Logic blocks in the *4x4 Bus Arbiter* (see Figures 17 and 15). Only one Priority Logic block is enabled at a time. Therefore

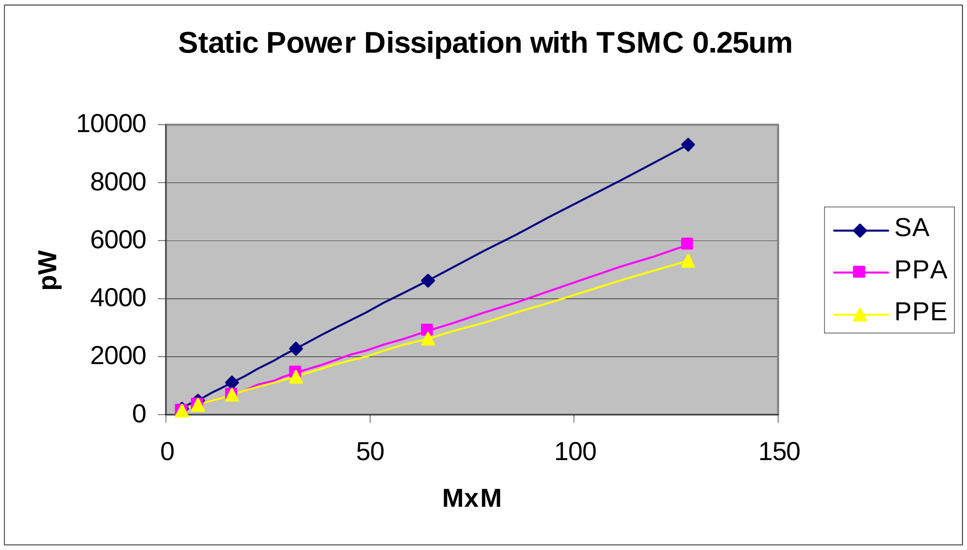


**Figure 51:** MxM hierarchical SA, PPE and PPA static power dissipation

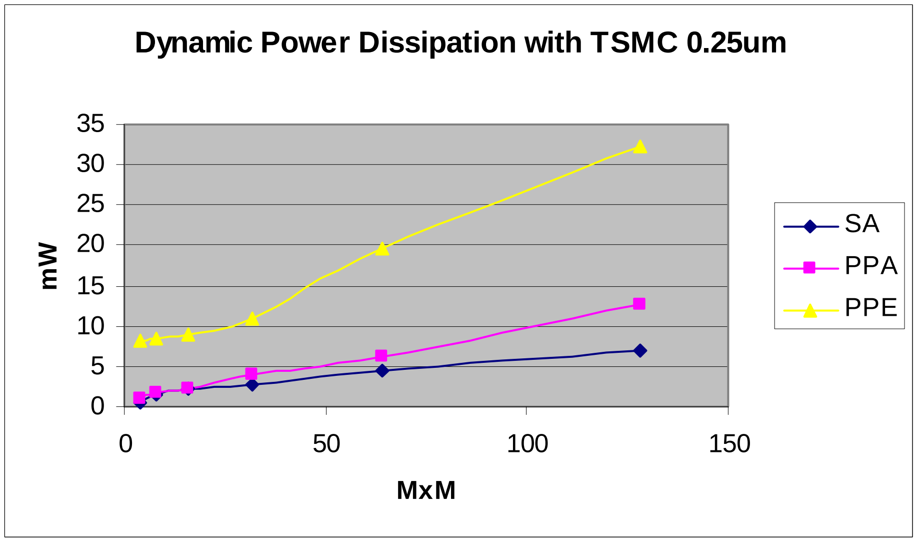


**Figure 52:** MxM hierarchical SA, PPE and PPA dynamic power dissipation

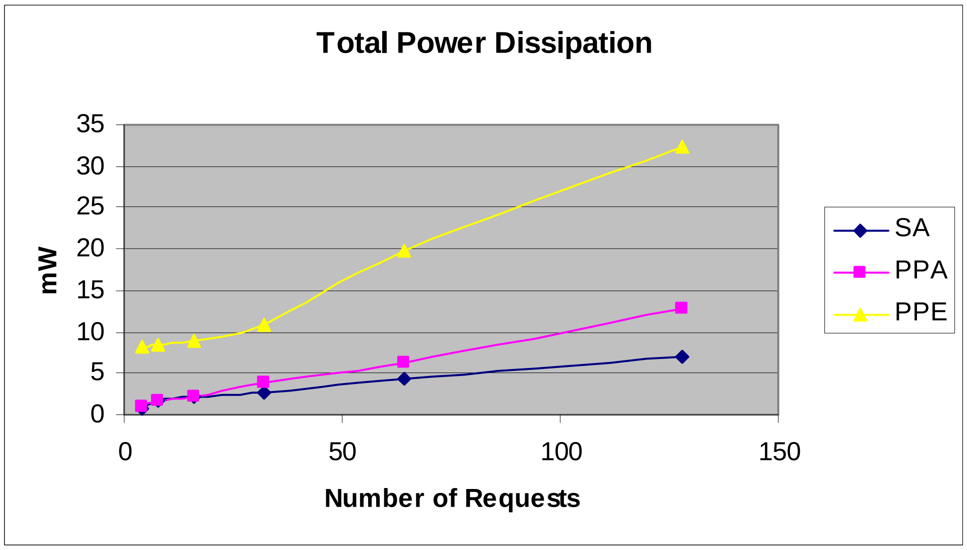


**Figure 53:** Total power dissipation of hierarchical SA, PPE and PPA

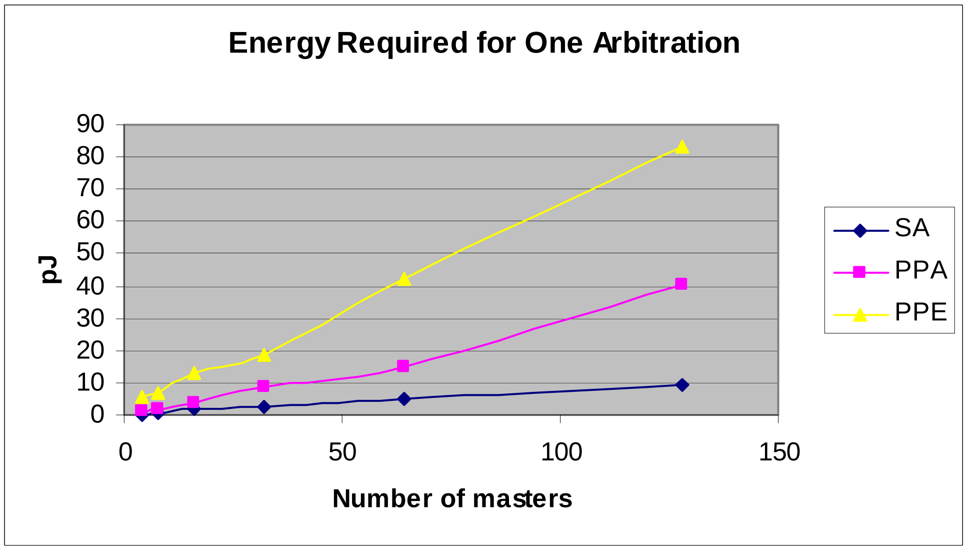


**Figure 54:** Energy required for one arbitration of MxM hierarchical SA, PPE and PPA

switching activity is associated with outputs of the Priority Logic block enabled in the current cycle and the Priority Logic block enabled in the previous cycle. Under the condition that all inputs are asserted continuously, the outputs of a Priority Logic block are switching every four cycles. However, the internal logic of AR2 (the basic switch arbiter block of PPA) switches every cycle. Thus, PPA dissipates larger dynamic power than our hierarchical SA due to more frequent switching activity. For PPE, it seems that the most switching activity caused by thirty-two ring counters that we add for the priority order rotation. Since a hierarchical BA consumes lower power than a BA shown in Section 7.1.3.1, an M-input proority encoder, the internal logic of PPE, presumably comsumes less power. Thus, our power estimation of PPE could be incorrect.

Figure 53 shows total power dissipation of hierarchical SA, PPE and PPA. The plot of total power dissipation is almost same as the plot of dynamic power dissipation since dynamic power contributes to most part of total power dissipation. However, static to dynamic power dissipation could be changed with deep submicron technologies since leakage becomes more significant in 0.13 $\mu$m and 0.09 $\mu$m technologies.

Figure 54 shows the energy required for one arbitration of hierarchical SA, PPE and PPA. Energy consumption increases as M increases for all three switch arbiters.

### 7.1.4 Speedup for a Chip Implementing a 32x32 Network Switch

In this section we consider the design of a 32x32 network switch chip. For comparison purposes with PPA and PPE, we first model a “64-bit 32x32 switch fabric” which consists of 64 32x32 crossbar switches, “input interconnect” (from network switch input ports – i.e., VOQs – to input ports of 64 crossbars), “output interconnect” (from output ports of 64 crossbars to network switch output ports) and control wires from SAs. A 32x32 crossbar switch connects a 1-bit wide input to a particular output according to control (grant) signals from a 32x32 SA. We will use 64 of these 32x32 crossbar switches in order to transmit a single 64-bit cell (a cell is a piece of a packet; we assume a packet is chopped into fixed size cells) per clock cycle.

Section 7.1.4.1 presents the delay through the 32x32 crossbar switch and the proposed floorplan for VOQs, SAs and 64-bit 32x32 switch fabric. From our experiment, it turns out that the delay of the input interconnect in the switch fabric contributes to the longest delay path. Thus, in Section 7.1.4.2, we show how we handle this longest delay path (input interconnect) with the insertions of pipeline registers. Since a cell travels through the input interconnect in a pipelined fashion, control (grant) signals from an SA should be delayed with D flip-flops in order to switch a specific cell to the corresponding output port at the right time. This is explained in detail in Section 7.1.4.3. Finally, in Section 7.1.4.4 we compare our hierarchical SA performance with PPE and PPA performances in terms of switching throughput based on a 32x32 network switch.

#### *7.1.4.1 32x32 Crossbar Switch Delay*

In order to estimate the delay through a crossbar switch, A 4x4 crossbar layout and then extract interconnect line lengths from the layout [55]. This layout is done using TSMC 0.25$\mu$m technology. By using of generic interconnect formulas [61], the delay calculation of a 32x32 crossbar switch is performed. It turns out than the longest delay path would be between the first input and the last output (e.g., between *I0* and *O31* in a 32x32 crossbar switch) as shown in Figure 55.

The C program written by Talpasanu iteratively generates the correct number of Resistance, Inductance, Capacitance (RLC interconnect model) and transistor components [55]. This RLC interconnect model is used to create an HSPICE input file to find the longest path delay. The HSPICE parameters used in this simulation are available from the MOSIS 0.25$\mu$m TSMC technology website [34]. These parameters include metal resistances per unit length, capacitances per unit area, capacitances per

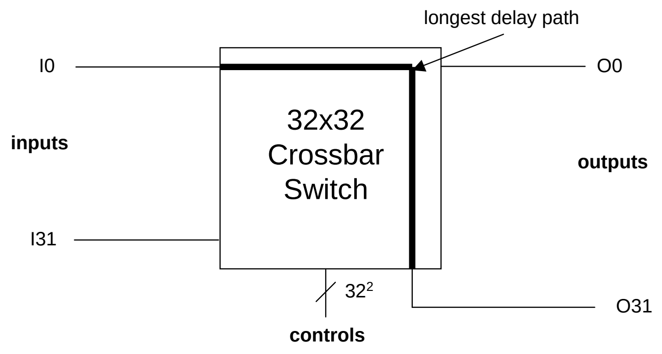


**Figure 55:** 32x32 crossbar switch and its longest delay path

unit length as well as other parameters needed in the HSPICE simulation. HSPICE level 49 transistor models are used in all delay simulations [34].

For a 64-bit cell switching, the 64-bit 32x32 switch fabric consists of 64 32x32 crossbars, input and output interconnects and control wires. For the 64-bit 32x32 switch fabric, a manufacturing process using six metal routing layers is assumed in order to approximate routing channel widths and lengths. A routing wire width of 1$\mu$m was chosen in order to reduce interconnect resistance. A routing pitch of 2$\mu$m is used in the calculation of routing width for interconnect channels. These parameters are used in the calculation of the 64-bit 32x32 switch fabric longest path delay. The delay through the input interconnect is reduced by inserting an optimal number of repeaters. The same optimization is applied to the output interconnect.

After an HSPICE simulation, the delay through one 32x32 crossbar switch is found to be 0.27 $ns$ which is measured. from the start of the input pulse to the 50% switching point for an output inverter. Following this simulation, the estimated area for a 32x32 crossbar switch is 0.153 $mm^2$ (542.45 $mm$ x 281.60 $mm$).

Figure 56 shows a floorplan of the 64-bit 32x32 switch fabric with 32 32x32 hierarchical SAs and $32^2$ VOQs. Each 64-bit wide VOQ maintains four precedence packets similar to a VOQ in Tiny Tera [28]. The areas of the thirty-two 32x32 SAs and the 322 VOQs are 1.40 $mm^2$ and 41.31 $mm^2$, respectively. Considering the 64-bit 32x32 switch fabric, the overall fabric area is approximately 125.64 $mm^2$ which is roughly 171 $mm^2$ smaller than the area of an Alpha 21364 processor [22]. This area approximation includes the input and output interconnects, 64 crossbars, and pipeline registers, but excludes I/O pads.

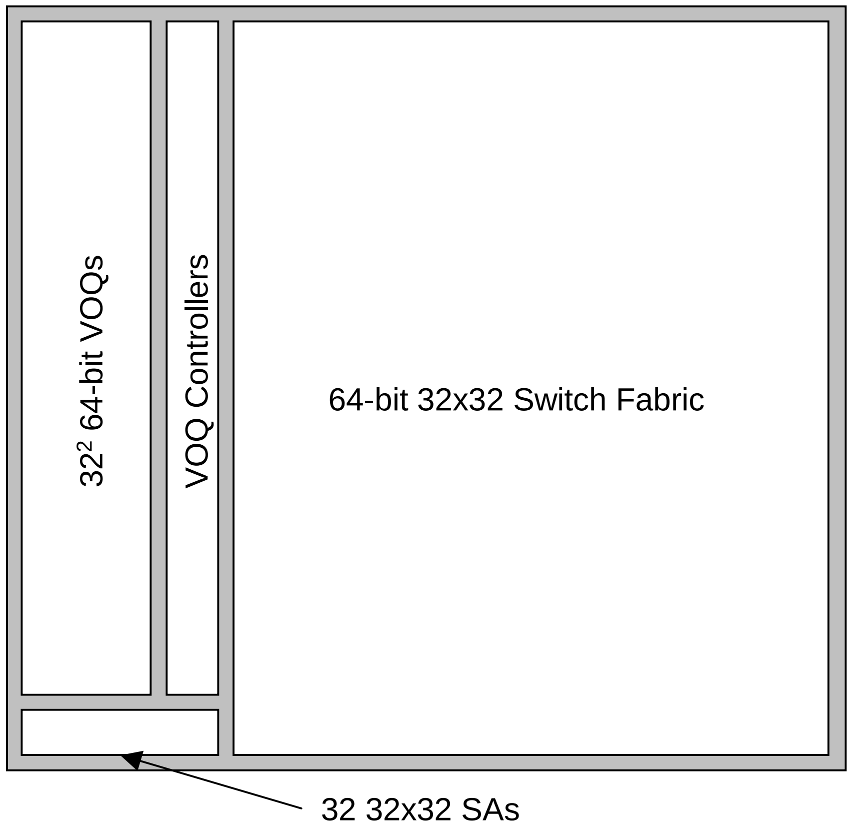


**Figure 56:** The floorplan of the 64-bit 32x32 switch fabric, VOQs, controllers and SAs

#### *7.1.4.2 Pipelined Input Interconnect for the 64-bit 32x32 Switch Fabric*

For the 64-bit 32x32 switch fabric, the input interconnect length is found to be 16.58 $mm$. This interconnect line is optimized by inserting two repeaters on the input interconnect line, and the delay is found to be 2.21 $ns$. The interconnect delay of the longest output path (length = 4.1 $mm$) is found to be 0.63 $ns$ with one repeater inserted [55].

Since the input interconnect from the VOQ to the 32x32 crossbar switch gives the longest delay (2.21 $ns$) in a 64-bit 32x32 switch fabric, the input interconnect delay contributions may be reduced by inserting pipeline registers periodically. Considering the longest delay path through the 64-bit 32x32 switch fabric to be through a one bit input interconnect, one 32x32 crossbar switch and through one bit output interconnect, the largest delay contributions would be those from the input interconnect. To minimize the delay through the input interconnect, a four-stage input interconnect pipeline is employed.

Figure 57 shows the input and output interconnects of the 64-bit 32x32 switch fabric including pipeline registers. The details of each "*4-stage interconnect*" box in Figure 57 are shown in Figure 58. Note that 4-stage interconnect in Figure 57 includes 32 "*4-stage pipeline*" blocks for all 32 input ports of a 32x32 crossbar. Each *4-stage pipeline* block consists of four D flip-flops arranged in series and spaced equally along an input interconnect. The individual stage delay for the input interconnect is approximately 0.63 $ns$.

#### *7.1.4.3 Pipelined control signals from SAs*

Since there are pipeline registers on the input interconnect, registers are also required to be inserted on the control lines (grant signals from thirty-two 32x32 SAs) which control the transmission gates of the crossbar switch. These registers are inserted so that the arbitration of the crossbar switch will happen in the same time slice in which the cell to be switched arrives at the input of the crossbar. Thus, four pipeline stages inserted on the control interconnect ensure that cells are being switched by the correct control signals. Each pipeline register is comprised of a D flip-flop triggered by a clock signal. The area of all pipeline registers is equal to 14.99 $mm^2$. This is included in the *64-bit 32x32 Switch Fabric* shown in Figure 56.

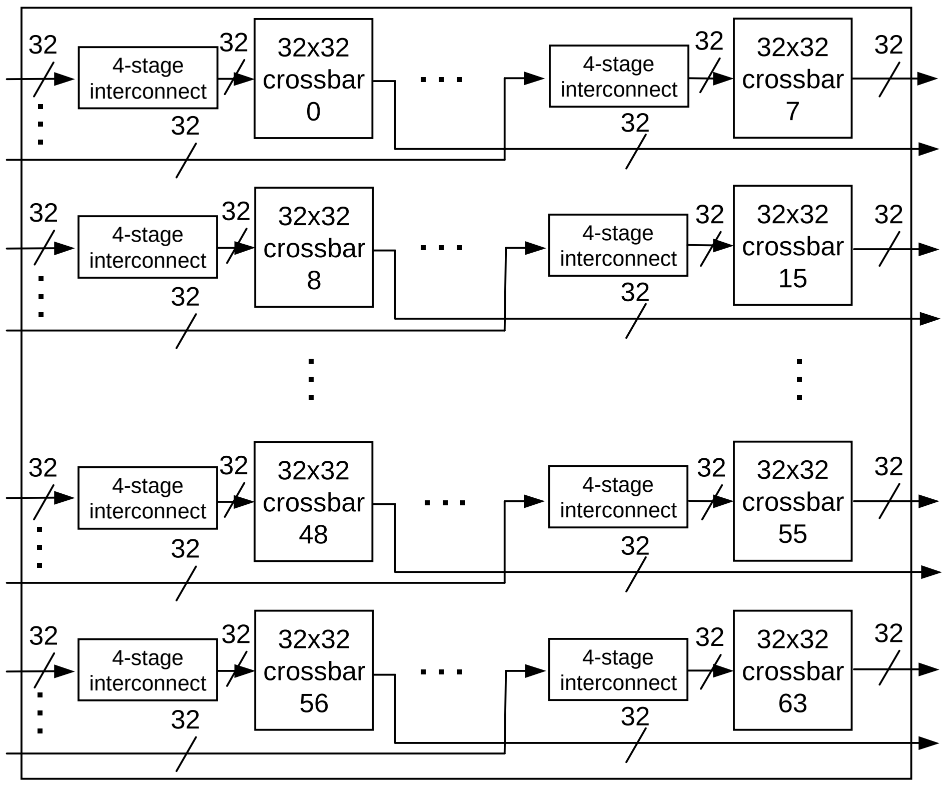


**Figure 57:** The internals of the 64-bit 32x32 switch fabric

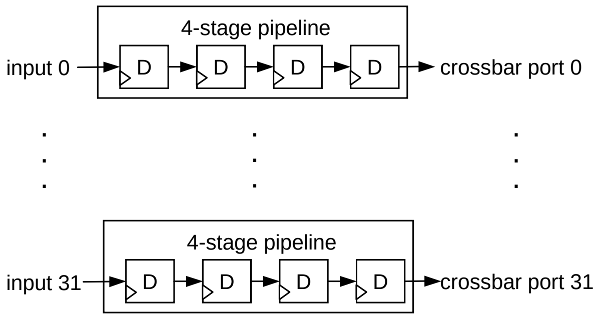


**Figure 58:** The internals of the 4-stage interconnect box

Accordingly, the different parts of the longest delay path are also pipelined. Namely, pipeline registers are inserted along the longest delay path. These pipeline stages are inserted before and after the 32x32 crossbars in order to reduce the overall 64-bit 32x32 switch fabric delay to the delay of a single pipeline stage.

Figure 59 shows the pipeline stages which would be inserted on every path to a 32x32 crossbar in order to reduce the delay. These stages are the SA delay, the

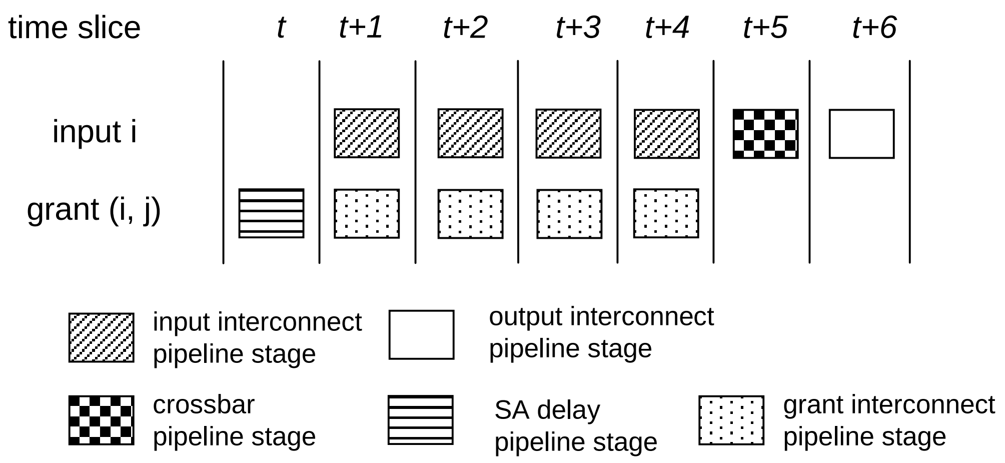

time slice
t
t+1
t+2
t+3
t+4
t+5
t+6
input i
grant (i, j)
input interconnect pipeline stage
output interconnect pipeline stage
crossbar pipeline stage
SA delay pipeline stage
grant interconnect pipeline stage

**Figure 59:** The grant-connect pipeline including an input interconnect and a control wire pipelines

segmented input interconnect (4 stages), the 32x32 crossbar stage, and the output interconnect stage. A cell from a VOQ can be forwarded to the first stage of an input interconnect after being granted. Note that *grant (i, j)* signal controls the specific transmission gate between the $i^th$ input port of a crossbar and the $j^th$ output port at time *t+5*.

Figure 60 shows how one control (grant) signal out of $32^2$ controls from 32 32x32 SAs in Figure 56 turns on the corresponding transmission gate in a 32x2 crossbar with a four stage pipeline delay. Four D flip-flops are utilized for a control wire pipeline.

In conclusion, with four stages of input and control interconnect as shown in Figure 59, the maximum throughput of the 32x32 network switch is limited either by the stage input interconnect delay (0.63 $ns$) or the arbitration delay.

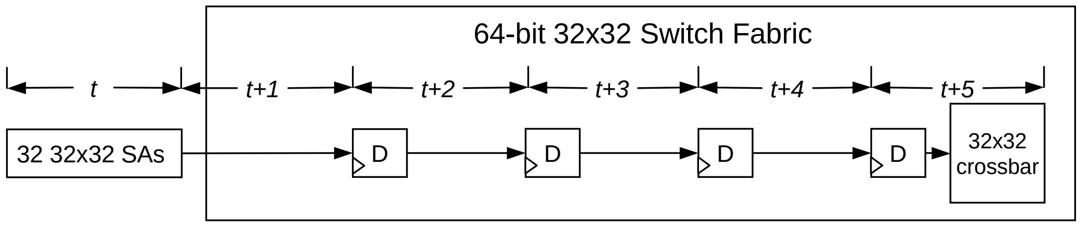


**Figure 60:** The 4-stage pipeline of a control wire

### *7.1.4.4 Switching Performance Comparison*

From our delay measurements of the 64-bit 32x32 switch fabric, the maximum throughput of this high-speed network switch is limited by the arbitration logic delay. Thus, the throughput of one port of the network switch with our 64-bit 32x32 switch fabric is equal to 64 bits (an eight-byte cell) divided by the 32x32 switch arbiter delay. The arbitration delay of our SA equals 0.94 *ns* where the arbitration delays of PPA and PPE are equal to 1.30 *ns* and 2.17 *ns*, respectively. Then, the aggregated bits per second (*bps*) capacities of 32x32 network switches with SA, PPA, and PPE are 2.18 *Tbps*, 1.20 *Tbps* and 0.94 *Tbps*, respectively. This can be verified by using the delays for SA, PPA and PPE shown in Figure 45. Thus, the RAG generated SA achieves speedups of 1.8X over PPA and of 2.3X over PPE, respectively.

Currently some commercial terabit switches are available. One is from Mindspeed, M21155 [33, 40]. M21155 is a 144-port x 144-port switch, each port delivering a data rate of up to 3.125 *Gbps*; the aggregate capacity is 0.45 *Tbps*. TeraCross, the partner of Mindspeed, built a 64x64 SA in a single chip (12 $mm^2$ die size) called TXS-1400 640Gbps Scheduler with 0.18$\mu$m Technology [58]. The other is PetaSwitch [39] from PetaSwitch Solutions, Inc. PetaSwitch claims that their chipset allows configuration of a switch for data rates from Gigabit Ethernet/OC-48 to OC-3072 and port numbers from 2x2 to 256x256. The aggregate bandwidth, PetaSwitch claims, can be configured from 40 *Gbps* to 10.24 *Tbps* depending on the number of ports and the data rate of port. Unfortunately, no information about the switch arbitration logic nor the process technology (e.g., 0.25$\mu$m) used is publicly available for either of these chips regardless of our effort (information was requested from the web and through phone calls, but no reply was ever received).

### 7.1.5 Fairness Simulation for Hierarchical SAs

In this section, we describe a simulator developed by G. F. Riley and Peng Cheng [9] to test the fairness of our hierarchical SA. The basic design of an arbiter, using priority encoder logic, is known to exhibit some unfairness if the order of request inputs to the priority encoder is fixed, thus giving a bias to lower numbered requests over higher ones in some cases. The goal of this simulation was to test the hypothesis that, under realistic network traffic assumptions, this theoretical unfairness would not be observed in any statistically significant way.

"Switch Arbiter Simulator" (SASim) implemented in C++ to model the behavior of our hierarchical SA design for an arbitrary number of inputs and outputs. SASim simulate a 32-port SA as shown in Figure 23, using a variety of input conditions for the network switch. Each of the experiments is described in detail below.

For the experiments, the measured metric is either the total number of grants or the "average delay per grant." The grant count metric is used for the first set of experiments, and the grant delay is used for the second set. Since we are interested in analyzing fairness under the priority allocation scheme described in Chapter 4, all of the results are summarized by the "4x4 ack-req SA input pin number." For a 32-input Switch Arbiter, 8 of the 32 inputs map to "input 0" on a 4x4 ack-req SA denoted by req$x$[0] in Figure 23; 8 of the 32 inputs map to "input 1" on a 4x4 ack-req SA denoted by req$x$[1] in Figure 23, etc. As we explained in Chapter 4, there is unfairness in the network protocol itself. Nevertheless, although we focus only on the unfairness of our SA which could occur within a single 4x4 ack-req SA, we can compare the grant rate and delay values for all of the "input 0" requests as an aggregate. All of our

**Table 2:** Simulation results, continuous requests

| | Run 1 | Run 2 | Run 3 |
|---|---|---|---|
| 4x4 ack-req input | Grants, All Asserted | Grants, 0, 1 Asserted | Grants, 0, 1, 2 Asserted |
| 0 | 250000 | 749000 | 499999 |
| 1 | 250000 | 250000 | 250000 |
| 2 | 250000 | 0 | 250000 |
| 3 | 249999 | 0 | 0 |

**Table 3:** Simulation results, bursty on-off traffic

| 4x4 ack-req input | Delay, $\rho = 2.0$ | Delay, $\rho = 1.0$ | Delay, $\rho = 0.9$ | Delay, $\rho = 0.5$ | Delay, $\rho = 0.1$ |
|---|---|---|---|---|---|
| 0 | 31.43 | 16.65 | 4.46 | 1.98 | 1.05 |
| 1 | 31.23 | 15.59 | 4.60 | 1.93 | 1.10 |
| 2 | 30.95 | 17.14 | 4.61 | 1.82 | 1.01 |
| 3 | 31.11 | 16.44 | 4.63 | 2.00 | 1.10 |

simulation experiments were executed for 1 million clock cycles, and all results are given in the tables below.

The first set of experiments simply asserted the requests continually. For the first experiment, all 32 inputs are asserted. Clearly, with this input pattern, the total number of grants should be approximately equal to the total number of clock cycles, and each input should receive an approximately equal number of grants. The results given in Run 1 of Table 2 confirm these hypotheses. The next experiment in this set asserts only those requests that map to “inputs 0” and “input1” on the 4x4 ack-req SAs. This is precisely the case that can cause some unfairness in the 4x4 ack-req SA, and the results shown in Run 2 of Table 2 demonstrate this clearly. The grants given to “inputs 0” is three times that of requests mapping to inputs 1. Then we asserted inputs 0, 1, and 2 (but not 3) continually, and again observed the unfairness with input 0 receiving more than the average grant count as shown in Run 3 of Table 2.

However, since the target application for our switch arbiter design is a high-speed internet router, we are more interested in request models based on internet traffic. To simulate this, we created a simulation model of bursty, on-off traffic with exponential distributions of on and off times and uniform distribution of packet sizes. The request pin for each switch input is driven by these on-off sources, with a separate source driving each of the 32 inputs. By varying the average off time of the sources, it is possible to vary the overall traffic load on the switch. This so-called traffic intensity (usually denoted $\rho$) is a measure of the demand on the switch relative to the maximum capacity of the switch. The 32x32 switch was simulated with $\rho$ values of 2.0, 1.0, 0.9, 0.5, and 0.1, and the results are shown in Table 3. For these experiments, the metric is the average delay (in clock cycles) per request.

Clearly, the traffic intensity of 2.0 is sending twice as much data to the switch than can be processed, and thus this experiment degenerates to the "all requests asserted continuously" that we described above. The results confirm this, showing in the second column of Table 3 the average delay per request to be nearly 32 clock cycles as expected. The lower $\rho$ values show correspondingly lower delays, also as expected. The $\rho$ value of 0.1 represents a lightly loaded switch, and shows an average grant delay of just over 1 clock cycle per request.

The important point to notice for the bursty traffic experiments is that the unfairness found in the 4x4 ack-req SA does not in fact result in unfairness in average request latency, when presented with realistic traffic requests.

Our 32x32 hierarchical SA was also simulated under GTNets which is a network simulator [41]. In essence, it functions like SASim. The difference is that SASim simulates the internal behavior of the switch arbiter [9], whereas GTNets is a comprehensive simulation of the entire internet network [41]. Protocol layers such as TCP and UDP are also embedded into GTNets. In this particular experiment, we use GTNets to test the functionality of a network switch that uses the round-robin

**Table 4:** Simulation results, TCP traffic using the GTNets log file

| 4x4 ack-req input | Delay, $\rho = 2.0$ | Delay, $\rho = 1.0$ | Delay, $\rho = 0.9$ | Delay, $\rho = 0.5$ | Delay, $\rho = 0.1$ |
|---|---|---|---|---|---|
| 0 | 18.12 | 16.75 | 11.68 | 1.95 | 1.10 |
| 1 | 19.91 | 19.48 | 11.29 | 1.95 | 1.10 |
| 2 | 20.44 | 19.71 | 10.69 | 1.93 | 1.10 |
| 3 | 19.22 | 17.54 | 12.13 | 1.94 | 1.10 |

switch arbiter. The network designed for simulation is as follows. There are 32 nodes (stations) in the network. All the 32 nodes are connected to a 32x32 router. For every client node, there are 31 applications requesting access other nodes; each application requests from a different server. Each node also has a server. This configuration allows each node to send packets to all the other nodes, and node is also receiving packets from all the other nodes. Note that the web-applications are requesting packets in a rate depending on the traffic intensity, $\rho$. The nodes have to send packets fast enough in order to reach certain traffic intensity. In GTNets, an application may make a request to a server, and then wait for a certain period, and then start to make another request to the same server.

For the first run of this simulation, the traffic intensity of the network switch is $\rho$ of 2.0 to see if SAs inside the network switch still maintain fairness under an overflow of packets. Then, the traffic intensities with 1.0, 0.9. 0.5 and 0.1 are simulated.

The results of TCP traffic simulations with different traffic intensities are displayed in Table 4. The average delay (in clock cycles) per request is used as the metric. For these experiments, first, SASim creates a 32x32 switch arbiter, and then it proceeds to parse through the log file to schedule packet arrivals. All the packets whose destination is node 1 are scheduled into SASim. For the case where $\rho$ approaches 1 (the maximum capacity of the switch), note that the maximum difference of the average delays occurs between req$x[0]$ and req$x[2]$ when $\rho = 1$. Also, the average delay per each input more slowly increases as $\rho$ increases from the maximum capacity of the switch ($\rho = 1$) because of transport control mechanism in TCP. The sliding window mechanism is one of the most well-known transport control protocol where a server puts packets into the network based on the window size. In the sliding window protocol, the window size is decreased multiplicatively if the network is congested (i.e., if the sender does not receive the acknowledgment in a certain time) and is increased additively when the congestion is relieved. Therefore, when the network becomes congested, the number of packets arrives at the switch becomes converged to the maximum switch capacity. The differences in the average delays among inputs with high traffic intensities (columns 2, 3 and 4 in Table 4) are not because of the lack of perfect fairness in our SA design but because of unfairness in TCP transport control. Low traffic intensities experiments (columns 5 and 6 in Table 4) confirm that our SA is fair enough for TCP traffic.

We conclude that the unfairness problem described in Section 4.5 does not show up under realistic traffic flows.

## 7.2 Xbar Synthesis

This section presents the synthesis results of various configurations of Xbars generated by X-Gt. We use the Synopsys Design Compiler [51] with a 0.25 $\mu m$ TSMC standard cell library [60] from Artisan Components [2]. We use the "TSMC25_Conservative" model for a wire load to provide more accurate area and delay results at the logic synthesis stage.

### 7.2.1 Xbar Area

In this section we first present the area of Mx1 switches to provide the guideline for X-Gt user to estimate the area of an MxN Xbar. Then, we show the areas of MxM Xbars estimated by Synopsys Design Compiler. We also perform the placement and route to report the accurate area including the area taken by interconnect area. We utilize

Cadence Silicon Ensemble [5] for the placement and route. For logic synthesis and physical synthesis we use TSMC 0.25 $\mu$m SAGE standard cell library from Artisan Components [2].

Figure 61 shows the synthesis results of the area of Mx1 switches for increasing number of processors. The total area of an MxN Xbar can be easily calculated by N times the area of an Mx1 switch, where N is the number of memory ports.

As shown in Figure 61, the area of an Mx1 switch increases almost linearly with the number of PEs except when $M = 8$. The area of Xbar, as shown in Figure 62, increases almost linearly as the number of PEs and the number of memory ports increase for an MxM Xbar configuration.

The area discrepancies between the area reported at the logic synthesis stage (noted as Cell Area) and the area reported after the placement and route (noted as Chip size) results from the interconnect areas between the wire area estimated at the logic synthesis stage using of "TSMC25_Conservative"wire load model and the actual wire area after route. The area difference between pre-layout and post-layout increases as M increases. The area discrepancies between logic syntheses and physical syntheses shown in Figure 62 are up to 23 % (when M=8). Thus, it turns out that using a wire load model at the logic synthesis stage for the interconnect area estimation provides insufficient accuracy for a wire-centric design such as an Xbar and a bus.

### 7.2.2 Xbar Delay

In the high level synthesis, the performance of a design is measured by summing delays of the operations on the critical path, while additional delay elements such as registers, multiplexers and buffers are considered in the RTL design [37]. However, interconnect delays are only included in the physical design. As deep submicron technology is evolving, interconnect delay becomes more significant portion of the total delay since

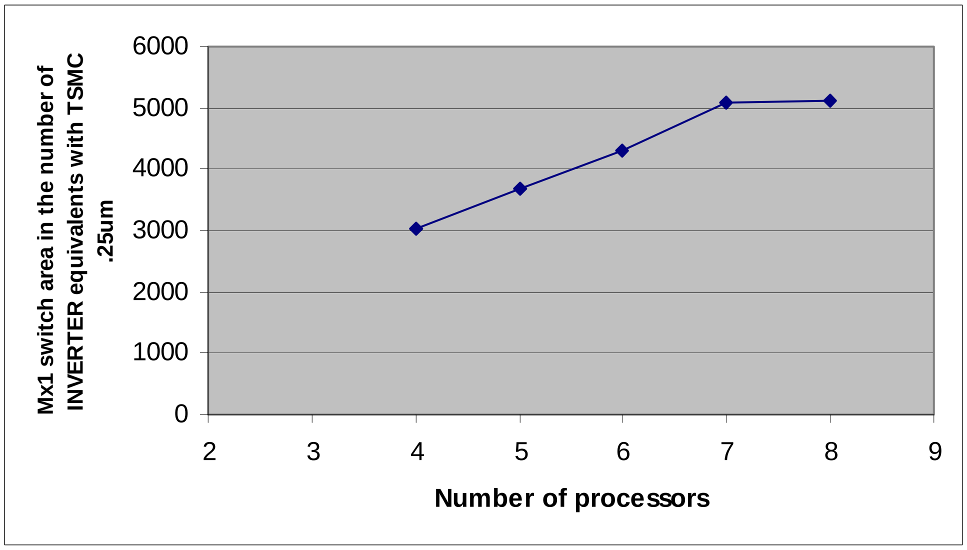


**Figure 61:** Area of Mx1 switch

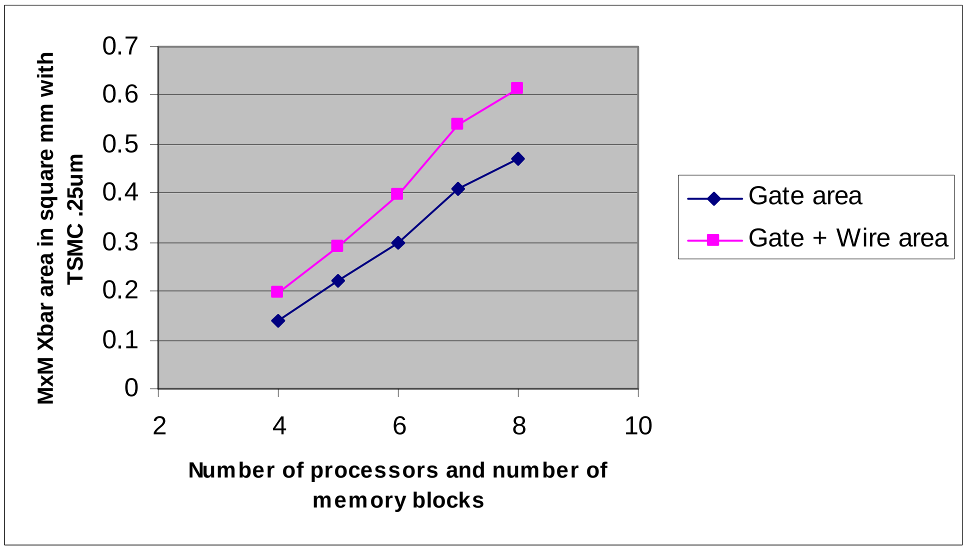


**Figure 62:** Area of MxM Xbar

interconnect delay stays constant while gate delay becomes shorter [42]. Thus, in this section, we perform timing analysis of generated Xbars in an RTL Verilog to minimize the gap between the high level synthesis and the physical level synthesis caused by added delays as a design goes down to lower level implementation.

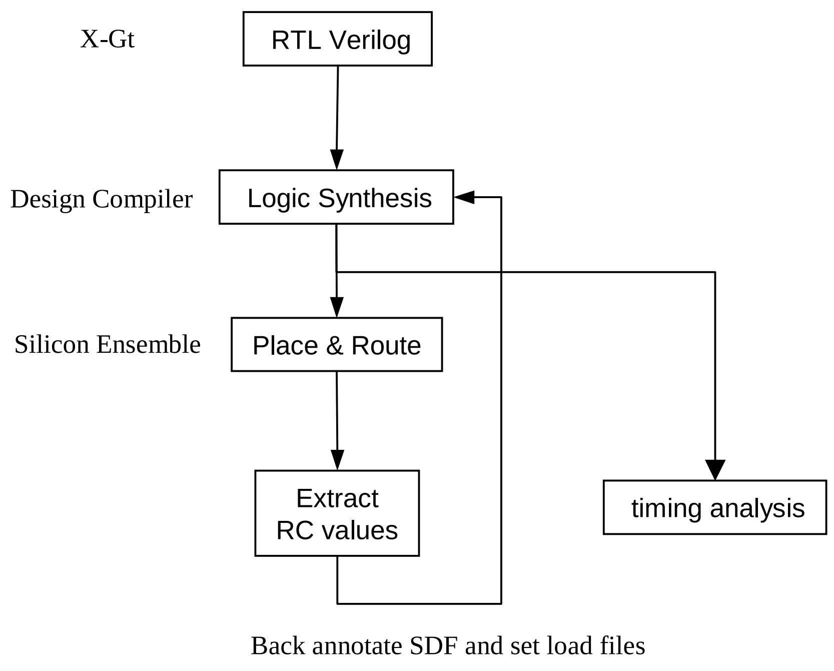


**Figure 63:** Back-annotation Flow

For delay analysis at the logic synthesis level, we first estimate the delay of an MxM Xbar with "TSMC25_Conservative" model for a wire load. Then, we extract Resistances and Capacitances (RC) values after place and route. We use Cadence Silicon Ensemble [5] for the placement and route. The RC values extracted from Silicon Ensemble are backward annotated to estimate more accurate wire delays in the form of Standard Delay Format (SDF) and set_load files. Figure 63 shows the flow of our methodology.

Design Compiler converts an Xbar in an RTL Verilog file into a netlist of registers and logic gates by technology mapping. The Xbar Verilog file compiled with "TSMC25_Conservative" wire load is an input to Silicon Ensemble in the form of the

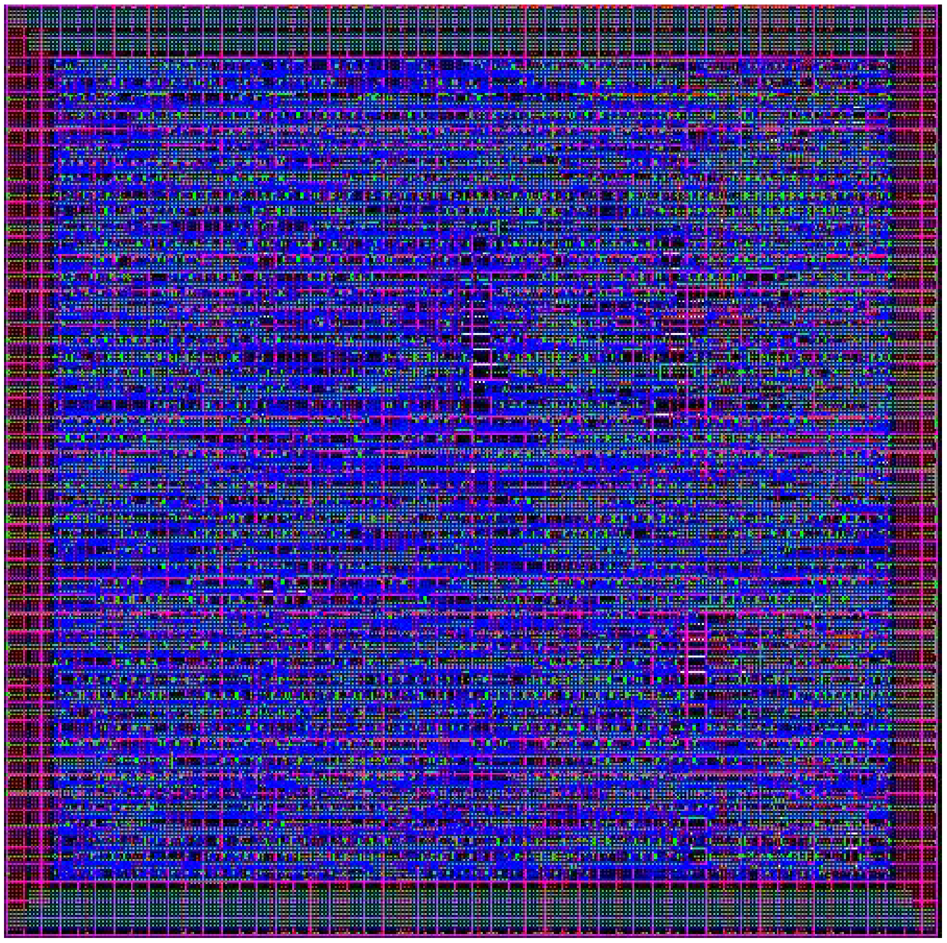


**Figure 64:** The snapshot of 4x4 Xbar layout

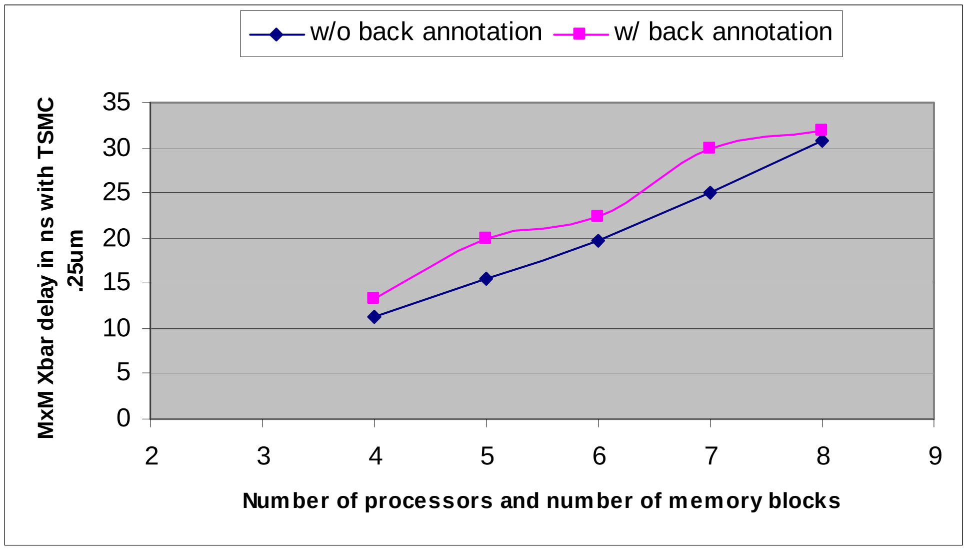


**Figure 65:** Delay of MxM Xbar

netlist. After place and route, layout information is exported to files in the type of *set_load* and SDF. In the resynthesis process, this *set_load* file is used to overwrite the capacitive net load values estimated with the wire load model by Design Compiler at the logic synthesis phase. In order to overwrite, the *include* command is used at the Design Compiler script prompt (dc_shell). The SDF file exported from Silicon Ensemble is also back annotated with rising delay and falling delay information of nets and ports. After replacing the estimated RC values with the exact RC values, we perform timing analysis of an Xbar again. Figure 64 shows the snapshot of 4x4 Xbar layout.

Figure 65 shows delays of Xbars for "without back-annotation" and "with back-annotation." In this figure, the differences are from the first iteration of the loop as shown in Figure 65. The maximum of 23% discrepancy occurs at the 5x5 Xbar.

Consequently, timing analysis with a wire load model at the logic synthesis phase gives inaccurate delay information for a long wire.

### 7.2.3 MP-SoC with Xbar and DMMU

Creating a high clock rate working chip containing millions of logic gates, whose functionality used to be implemented by multiple chips, is a time consuming task for Integrated Circuits (IC) designers. Usually designers start with floorplanning the chip by partitioning it into portions each of which presents a functional entity in the SoC (e.g., CPU, DRAM, caches or custom peripheral logic). Also, this step involves pin assignment, global routing and global clock tree generation. Tools such as Synopsys Chip Architect [50] can help the SoC designer to accomplish such tasks. The floorplanning and global routing provides useful timing information required for the RTL logical and physical synthesis of each functional entity using tools such as the Synopsys Design/Physical Compiler [51, 52] and the Cadence Silicon Ensemble [5]. Finally the layout of each unit is integrated into the SoC floorplan.

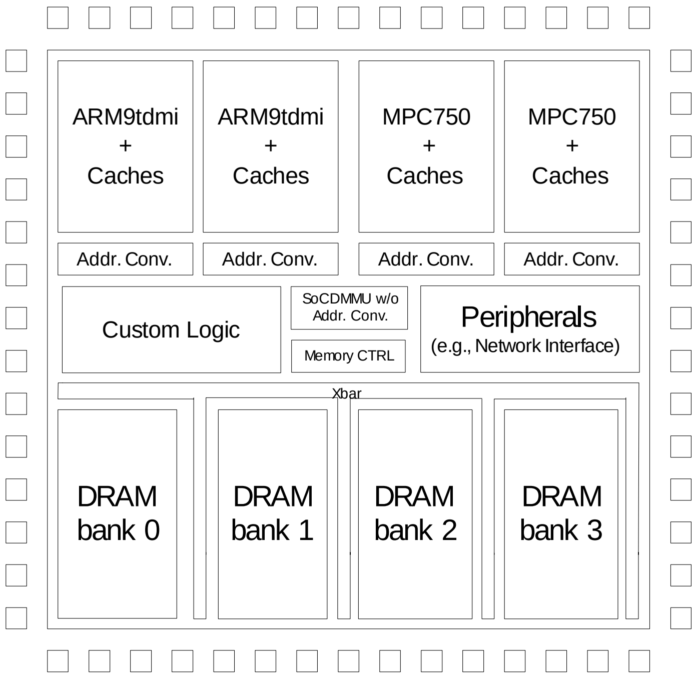


**Figure 66:** The floorplan of an SoC that utilizes the SoCDMMU and the Xbar

DX-Gt generates an RTL Verilog HDL of the SoCDMMU and the Xbar which are parts of the SoC presented in Figure 38. These RTL models with the RTL models of the other system components can be used to generate the layout of the SoC IC using the design flow described in the previous paragraph. As a sample effort to show part of the methodology described in the previous paragraph, we synthesized the RTL Verilog of the SoCDMMU (customized for 256 G_blocks and 4 PEs) and the Xbar and then placed and routed the layout using a 0.25 $\mu$m TSMC technology library. Figure 66 shows the floorplan of the system with four processors and four memory blocks of Example 6.2 including the layouts of the SoCDMMU and the Xbar. Since DRAM cores are not available, Figure 66 presumably describes how 4x4 Xbar is interleaved into DRAM modules. From our layout of a subset of Figure 66 (i.e., SoCDMMU plus Xbar), the area of 4x4 Xbar is 0.19 $mm^2$ and the area of SoCDMMU is 1.43 $mm^2$.

# CHAPTER VIII

# CONCLUSION

As the number of transistors on a single chip increases rapidly, there is a productivity gap between the number of transistors available on a chip and the number of transistors per hour a designer can design. In other words, it is almost impossible for human designers to cope with the number of transistors available in a chip which doubles every 18 months by Moore's Law [65], while the number of transistors per hour which a chip designer designs increase 21% per year [42]. One solution to reduce this productivity gap is to increase the reusability of Silicon Intellectual Property (SIP) cores. However, an SIP core should be customized/configured before being used in a system different than the one for which it was designed. Thus, to reconfigure the SIP core, either an engineer must spend significant effort altering the core by hand or else an enhanced CAD tool (SIP generator) can automatically configure and customize the core according to the customer specifications.

In a multiprocessor System-on-a-Chip (SoC) environment, a silicon CMOS chip designer should consider the need for an arbiter to resolve conflicts on shared resources (i.e., bus or equivalent communication channels) among multiple bus masters (e.g, processors). In a bus-based system, processors could be stalled because of bus conflicts. Thus, a high-performance arbiter is needed to resolve bus contentions among bus masters; such a fast arbiter can also reduce processor stall time by shortening arbitration delays.

In computer networks, such a fast and efficient arbiter commands more attention to resolve contention for crossbar switch(es) of a fast network switch as the amount of user traffic continues to double every year [35]. If the network switch capacity fails to increase with user traffic, then internet service providers may have to increase the number of switches in their network each year. Alternatively, the capacity of a single network switch needs to increase instead, and a fast and efficient arbiter plays a key role in increasing such switch capacity. Considering power consumption, such an arbiter plus other network switch components (e.g, VOQs and crossbar switch fabric) are preferably implemented in a single chip since the power budget of a network switch is about 10 $kW$ per rack. A recent single rack of network switches, which aims at terabit switching, have already reached this limit [20]. As mentioned in [7, 20, 26, 29, 66], arbitration delay is one of the major obstacles to achieving terabit switching.

The primary objective of this thesis is to automate the design of round-robin arbiter logic. The resulting arbitration logic is more than 1.8X times faster than the faster prior state-of-the-art arbiters the author could find reported in the literature. The generated arbiter implemented in a single chip is fast enough in 0.25 $\mu$m CMOS technology to achieve terabit switching with a single chip computer network switch. Moreover, this arbiter is applicable to crossbar (Xbar) switch arbitration logic. The generated Xbar, customized according to user specifications, provides multiple communication paths among masters and slaves.

In order to achieve our goal, we design and develop a tool for the automatic generation of a round-robin arbiter for a bus. Our arbiter is also applicable to high-speed network switches. We call this tool Round-robin Arbiter Generator (RAG). The RAG can generate a hierarchical Bus Arbiter (BA) which is faster than all known previous approaches in a Register Transfer Level (RTL) Verilog code for a bus. The RAG can also generate a hierarchical Switch Arbiter (SA), a high-speed network switch arbiter (scheduler) which is faster than all known previous approaches. Finally, a crossbar (Xbar) switch generation tool is developed that is integrated with RAG for the generation of its arbiter to resolve conflicts among masters. We name the tool **X**bar **G**enera**t**or (X-Gt) which generates an MxN Xbar in an RTL Verilog.

A hierarchical BA generated by RAG provides a fast and reasonably fair arbitration scheme for both on-chip and off-chip buses. We discussed the BA logic and showed the logic of 2x2, 3x3 and 4x4 switch arbiter block components. RAG can also generate a parallel hierarchical MxM SA. We presented how RAG utilizes 2x2, 3x3 and 4x4 switch arbiter blocks to produce an MxM hierarchical BA or an MxM hierarchical SA. We compared the area and delay of our generated SAs with PPA and PPE, respectively. It turns out that our distributed switch arbiters generated by RAG lead to significant delay improvement using a novel token-passing hierarchical arbitration scheme when compared with other switch arbiters such as PPA and PPE. Specifically, RAG can be used to generate a SA for a 32x32 terabit switch which achieves speedups of 1.8X over PPA and of 2.3X over PPE for the same 32x32 configuration. In general, the speedup of our generated SA over PPA and PPE increases slightly as M increases in an MxM switch arbiter; for example, for a 128x128 switch arbiter, our generated SA has a 1.9X speedup over PPA and 2.4X speedup over PPE.

We also compare the power dissipation of our generated SA with PPA and PPE. The experimental results shows that our SA dissipates less power compared with PPA and PPE even though the area of our SA is biggest.

To prove that the fairness issue of arbitration scheme is relatively insignificant, we demonstrate with a set of simulation experiments that the overall grant latency of our SA exhibits reasonable fairness when presented with a bursty and TCP traffic models.

For the SoC design aspect, we described an SoC SIP generation tool that enables a silicon CMOS chip designer to configure a crossbar, a subset of bus systems, to meet the design constraints with ease. Thus, our thesis is also focused on the provision of a CAD tool for an MxN Xbar switch named X-Gt. Hierarchical bus arbiter generator, the subset of RAG, is included in X-Gt to provide fast arbitration scheme for our Xbar.

Also, we showed the integration of the SoCDMMU and the Xbar into one system. The combination of the SoC Dynamic Memory Management Unit (DMMU) and the Xbar for a multiprocessor SoC has been shown to have an overall speedup of 4.4X during the application transition time when compared to a fully shared memory system with the same memory organization and number of processors [44]. Our X-Gt is integrated with SoCDMMU generator and is named as **D**ynamic memory management unit and **X**bar **G**enera**t**or (DX-Gt). DX-Gt automatically generates an RTL Verilog file. Also, DX-Gt generates a Verilog file required to enable standard EDA tools to implement the system.

As a billion transistors on a single chip begin to appear, the customization of SIP cores becomes much more complicated to a silicon CMOS chip designer, resulting in longer time-to-market. Thus, the automatic generation of a hierarchical round-robin arbiter and an Xbar will significantly reduce the time taken by customization. Automatic arbiter generation results in fast arbitration compared with standard logic synthesis techniques. A brief but interesting comparison with logic synthesis algorithms reveals that a key insight of our approach lies in our customized hierarchy. Our novel token-passing hierarchical arbitration scheme using priority logic blocks with at most four inputs yields the reported speedups. Not surprisingly, traditional logic synthesis algorithms do not consider our customized hierarchy; thus, given a logic description of a specific bus- or switch-arbiter, logic synthesis algorithms do not automatically consider our custom hierarchical logic structure. Also, to the best of our knowledge, we present the first published work on the automatic generation of a round-robin arbiter and an Xbar.

Since the generated Xbar is orthogonal to processor types, the Xbar is valuable in the sense that it can easily be integrated into a heterogeneous multiprocessor SoC. The automation of customizable and configurable versions of the Xbar according to the customer specifications: our thesis presents an SIP-generator, called X-Gt.

In conclusion, our hierarchical SA generated by RAG shows the improvements of performance and power dissipation compared with the hand-coded SIPs such as PPA and PPE. Also, upon the user specifications, X-Gt generates an Xbar on-the-fly for an multiprocessor SoC.

# REFERENCES

[1] ARC Inc., Available HTTP: http://www.arc.com.

[2] Artisan Components Inc., Available HTTP: http://www.artisan.com.

[3] Brayton, R. K. and Sangiovanni-Vincentelli, A. L., "Multilevel logic synthesis," *Proceedings of the IEEE*, Vol. 78, pp. 264–300, February 1990.

[4] Brook, D., Tiwari, V., and Martonosi, M., "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of ACM International Symposium on Computer Architecture*, pp. 83–94, June 2000.

[5] Cadence Silicon Ensemble. Available HTTP: http://www.cadence.com/products /sepks.html.

[6] Chandrakasan, A. and Brodersen, R. W., *Low-Power CMOS Design.* NY: Wiley-IEEE Press, 1998.

[7] Chao, H. J., Lam, C. H., and Guo, X., "A fast arbitration scheme for terabit packet switches," in *Proceedings of the IEEE Global Telecommunications Conference*, pp. 1236–1243, December 1999.

[8] Chao, H. J. and Park, J. S., "Centralized contention resolution schemes for a larger-capacity optical atm switch," in *Proceedings of the IEEE ATM Workshop*, pp. 11–16, May 1998.

[9] Cheng, P., Shin, E. S., Riley, G. F., and Mooney, V. J., "Sasim: Switch arbiter simulator," in *Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-03-38.* Available HTTP: http://www.cc.gatech.edu/tech_report s/index.03.html.

[10] Commer, D. E., *Internetworking with TCP/IP Principles, Protocols and Architecture, Vol.I.* NJ: Prentice Hall Inc, 1995.

[11] Dally, W. J. and Towels, B., "Route, packets, not wires: On-chip interconnection networks," in *Proceedings of the IEEE Design Automation Conference*, pp. 684–689, June 2001.

[12] Gupta, P., *Algorithms for Routing Lookups and Packet Classification.* CA: Ph. D. Thesis, Stanford Univeristy, 2000.

[13] Gupta, P. and McKeown, N., "Designing and implementing a fast crossbar scheduler," *IEEE Micro*, Vol. 19, pp. 20–28, January - February 1999.

[14] Gupta, P. and McKeown, N., "Algorithms for packet classification," *IEEE Network Special Issue*, Vol. 15, pp. 24–32, March – April 2001.

[15] Hachtel, G. D. and Somenzi, F., *Logic Synthesis and Verification Algorithms.* MA: Kluwer Academic Publishers, 1996.

[16] Iyer, S. and McKeown, N., "Maximum size matching and input queued switches," in *Proceedings of the Allerton Conference on Communication, Control and Computing.*

[17] Iyer, S. and McKeown, N., "Using constraint sets to achieve delay bounds in cioq switches," *IEEE Communications Letters*, Vol. 7, pp. 275–277, June 2003.

[18] Jantsch, A. and Tenhunen, H., *Networks on Chip.* MA: Kluwer Academic Publishers, 2003.

[19] Karim, F., Nguyen, A., Dey, S., and Rao, R., "On-chip communication architecture for OC-768 network processors," in *Proceedings of the IEEE Design Automation Conference*, pp. 678–683, June 2001.

[20] Keslassy, I., abd K. Yo, S.-T. C., Miller, D., Horowitz, M., Solgaard, O., and McKeown, N., "Scaling internet routers using optics," in *Proceedings of the ACM SIGCOMM*, pp. 189–199, August 2003.

[21] Keslassy, I., Zhang-Shen, R., and McKeown, N., "Maximum size matching is unstable for any packet switch," *IEEE Communications Letters*, Vol. 7, pp. 496–498, October 2003.

[22] Krewell, K., "Alpha ev7 processor: A high-performance tradition continue," in *In-Stat/MDR*, 2003. Available HTTP: http://h18003.www1.hp.com/hps/download/Compaq_EV7_Wp.pdf.

[23] LEDA Systems, Available HTTP: http://www.ledasys.com.

[24] Leon-Garcia, A. and Widjaja, I., *Communication Networks: Fundamental Concepts and Key Architecture.* OH: McGraw-Hill Inc., 2000.

[25] Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W., and Horowitz, M., "Smart memories: A modular reconfigurable architecture," in *Proceedings of ACM International Symposium on Computer Architecture*, pp. 161–171, June 2000.

[26] McKeown, N., "Optics inside routers," in *Proceedings of the European Conference and Exhibition on Optical Communucation*, September 2003.

[27] McKeown, N., Anantharam, V., and Warland, J., "Achieving 100% throughput in an input-queued switches," in *Proceedings of the IEEE INFOCOM*, pp. 13–27, March 1996.

[28] McKeown, N., Izzard, M., Mekkittikul, A., Ellersick, W., and Horowitz, M., "Tiny tera: A packet switch core," *IEEE Micro*, Vol. 17, pp. 26–33, January/February 1997.

[29] McKeown, N., Varaiya, P., and Warland, J., "The islip scheduling algorithm for input-queued switch," *IEEE/ACM Transaction on Networking*, Vol. 7, pp. 188–201, April 1999.

[30] Mekkittikul, A. and McKeown, N., "A practical scheduling algorithm for achieving 100% throughput in input-queued switches," in *Proceedings of the IEEE INFOCOM*, pp. 792–799, March 1998.

[31] Mentor Graphics Seamless, Available HTTP: http://www.mentor.com/seamless.

[32] Micheli, G. D., *Synthesis and Optimization of Digital Circuits.* NJ: McGraw-Hill Inc., 1994.

[33] Mindspeed, Available HTTP: http://www.mindspeed.com.

[34] The MOSIS Service. Available HTTP: http://www.mosis.org/products/fab/vendors/tsmc/tsmc025/index.html.

[35] Odlyzko, A. M., "Comments on the larry roberts and caspian networks study of internet traffic growth," in *The Cook Report on the Internet*, pp. 12–15, December 2001. Available HTTP: http://cookreport.com.

[36] P. Gupta, B. P. and Boyd, S., "Near-optimal routing lookups with bounded worst case performance," in *Proceedings of the IEEE INFOCOM*, pp. 1184–1192, March 2000.

[37] Park, S., Kim, K., Chang, H., Jeon, J., and Choi, K., "Backward-annotation of post-layout delay information into high-level synthesis process for performance optimization," in *Proceedings of the IEEE International Conference on VLSI and CAD*, pp. 25–28, October 1999.

[38] PCI Special Interest Group, Available HTTP: http://www.pcisig.com.

[39] PetaSwitch, Available HTTP: http://peta-switch.com.

[40] Rigby, P., "Mindspeed unveils terabit switch chip," in *Network World Fusion Newsletter*, December 2001. Available HTTP: http://www.nwfusion.com/newsletters/optical/2001/ 01142734.html.

[41] Riley, G. F., "The Georgia Ttech Network Simulator," in *Proceedings of Workshop on Models, Methods, and Tools for Reproducible Network Research*, p. to appear, August 2003.

[42] Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*, 2001. Available HTTP: http://www.semichips.org/pre_stat.cfm?ID=183.

[43] Shalan, M. and Mooney, V. J., "A dynamic memory management unit for embedded real-time system-on-a-chip," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 180–186, November 2000.

[44] Shalan, M. and Mooney, V. J., "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pp. 79–84, May 2002.

[45] Shalan, M. A., Shin, E. S., and Mooney, V. J., "Dx-gt: Memory management and crossbar switch generator for multiprocessor system-on-a-chip," in *Proceedings of Workshop on Synthesis And System Integration of MIxed Technologies*, pp. 357–364, April 2003.

[46] Shin, E. S., Mooney, V. J., and Riley, G. F., "Round-robin arbiter design and generation," in *Proceedings of the International Symposium on System Synthesis*, pp. 243–248, October 2002.

[47] Silberschatz, A., Galvin, P., and Gagne, G., *Applied Operating System Concepts*. NY: John Wiley and Sons Inc., 2000.

[48] Stalling, W., *Data and Computer Communications*. NJ: Prentice Hall Inc, 1999.

[49] Sun, D., Blough, D., and Mooney, V. J., "Atalanta: A new multiprocessor rtos kernel for system-on-a-chip applicationsn," in *Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-19*. Available HTTP: http://www.cc.gatech.edu/tech_reports/index.02.html.

[50] Synopsys Chip Architect, Available HTTP: http://www.synopsys.com/products/designplanning/designplanning.html.

[51] Synopsys Design Compiler. Available HTTP: http://www.synopsys.com/products/logic/design_comp_cs.html.

[52] Synopsys Physical Compiler. Available HTTP: http://www.synopsys.com/products/unified_synthesis/unified_synthesis.html.

[53] Synopsys Power Compiler. Available HTTP: http://www.synopsys.com/products/power/power_ds.html.

[54] Synopsys VCS Logic Simulator. Available HTTP: http://www.synopsys.com/products/ simulation/vcs_ds.html.

[55] Talpasanu, A., Davis, J. A., Shin, E. S., and Mooney, V. J., "Crossbar switch interconnect delay calculation," in *Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-03-37*. Available HTTP: http://www.cc.gatech.edu/tech_reports/index.03.html.

[56] Tamir, Y. and Chi, H.-C., “Dynamically allocated multi-queue buffers for vlsi communications switches,” *IEEE Transaction on Computers*, Vol. 41, pp. 725–737, June 1992.

[57] Tensilica Inc., Available HTTP: http://www.tensilica.com.

[58] Teracross, “TXS-1400 640 Gbps Scheduler,” Available HTTP: http://www.tera cross.com/web/pdfs/TXS-1400.pdf.

[59] Tobagi, F. A., “Fast packet switch architectures for broadband integrated services digital networks,” *Proceedings of the IEEE*, Vol. 78, pp. 133–167, January 1990.

[60] TSMC IP Services, Available HTTP: http://www.tsmc.com/design/ip.html.

[61] Uyemura, J., *Introduction to VLSI Circuits and Systems.* NY: John Wiley and Sons Inc., 2002.

[62] Verilog PreProcessor, Available HTTP: http://www.surefirev.com/vpp.

[63] VITA-VME bus International Trade Association. Available HTTP: http://www. vme.com.

[64] Wakerly, J., *Digital Design Principles and Practices.* NJ: Prentice Hall Inc., 1990.

[65] Wolf, W., *Modern VLSI Design: Systems on Silicon.* NJ: Prentice Hall Inc, 1998.

[66] Yun, K., “A terabit multiservice switch,” *IEEE Micro*, Vol. 21, pp. 58–70, January - February 2001.