Round-robin Arbiter Design and Generation

Eung S. Shin, Vincent J. Mooney III and Georgey Riley

Abstract— **In this paper, we introduce a Round–robin Arbiter Generator (RAG) tool. The RAG tool can generate a design for a Bus Arbiter (BA). The BA is able to handle the exact number of bus masters for both on-chip and off-chip buses specified by the user of RAG. RAG can also generate a distributed and parallel hierarchical Switch Arbiter (SA). The first contribution of this paper is the automated generation of a round-robin token passing BA to reduce time spent on arbiter design. The generated arbiter is fair, fast, and has a low and predictable worst-case wait time. The second contribution of this paper is the design and integration of a distributed fast arbiter, e.g., for a terabit switch, based on 2x2 and 4x4 switch arbiters (SAs). Using a .25µ TSMC standard cell library from LEDA Systems [11, 15], we show the arbitration time of a 256x256 SA for a terabit switch and demonstrate that the SA generated by RAG meets the time constraint to achieve approximately six terabits of throughput in a typical network switch design. Furthermore, our generated SA performs better than the Ping-Pong Arbiter and Programmable Priority Encoder by a factor of 1.9X and 2.4X, respectively.**

Index Terms— **arbiter, distributed arbiter, round-robin token passing, synthesis, terabit switch**

I. INTRODUCTION

As the era of a billion transistors on a single chip fast approaches, more Processing Elements (PEs) can be placed on a System-on-a-Chip (SoC). Most 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. Especially, a fast arbiter is one of the most dominant factors for high performance network switches [5]. Also, fast and efficient switch arbiters are needed to switch packets in a Network-on-Chip (NoC) [1]. However, to design with high performance and fairness in arbitrations 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) [2, 14]. We will show how our Round-robin Arbiter Generator (RAG) can help in the design of a terabit switch.

The paper is organized as follows. In Section II, we define terms that might cause confusion between people who work in computer networks and people who work in the hardware chip design field. Next, we explain related work in Section III. Section IV shows the bus and switch arbiter logic designs for a Bus Arbiter (BA) and basic switch arbiter blocks for a Switch Arbiter (SA). Section IV also shows how basic switch arbiter blocks are integrated into an SA. In Section V, we explain the flow of our Round-robin Arbiter Generator (RAG) tool and the area and delay considerations for an SA. In Section VI, our experimental results show the performance comparison for a BA and

demonstrate how our generated 128x128 SA achieves terabit switching speeds using a TSMC 0.25µm standard cell library from LEDA Systems [11, 15]. Finally, we conclude the paper in Section VII.

II. TERMINOLOGY

In this section, we define terms to describe Figure 1. Figure 1 shows the inputs and outputs of the crossbar switch fabric in a 32x32 network switch. Note that we use the terms "switch" and "network switch" interchangeably throughout this paper.

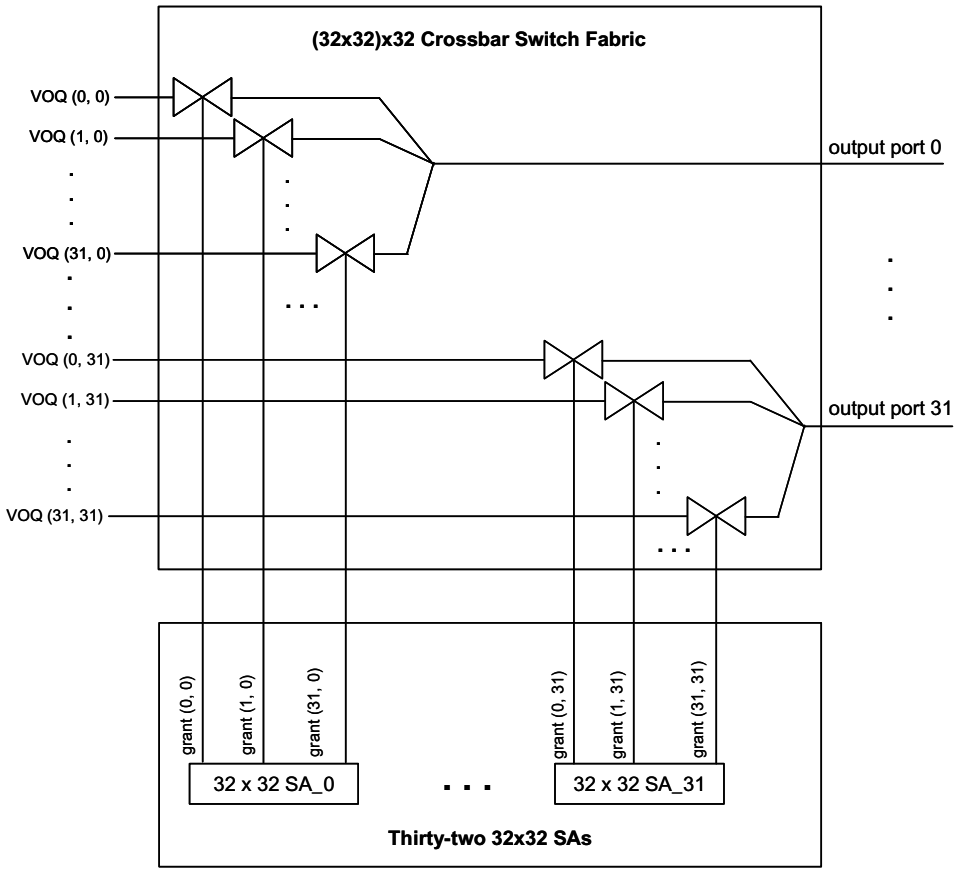

Figure 1. Internal structure of (32x32)x32 crossbar switch fabric and thirty-two 32x32 SAs of 32x32 network switch

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 paper are network switches.
2) **Virtual Output Queues (VOQs)** are typically employed in a packet switch to mitigate the head-of-line (HOL) block problem. HOL blocking occurs when a single FIFO input queue is used for each input port, and the packet in the head of the queue is blocked from being be forwarded to its corresponding output port

due to port contention. By using separate input queues for each input/output port pair, the HOL blocking problem is solved [6].

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.

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 2(a). For example, input port 0 as shown in Figure 2(a) 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.

Example 1. Suppose we design a 3x2 network switch. Each input port is allocated 2 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.

5) **(MxV)xN crossbar switch fabric**: There are connections between (MxV) 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 VOQ at the l^{th} input port destined to m^{th} output port. As an example, Figure 1 shows a (32x32)x32 crossbar switch fabric.

6) An **MxM Switch Arbiter (SA)** is a part of an (MxV)xN switch with M=V=N; thus, the number of requests (M) equals the number of grants (M). An MxM SA controls M specific transmission gates between M VOQs and a particular output port. At most one transmission gate is turned on at a time. In Figure 1, for example, signal grant (0, 31) from 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 total number of MxM SAs needed for an (MxM)xM switch is equal to the number of output ports, M.

Example 2. (Continued from Example 1.) There are six (=3∗2) inputs (equivalently, the total number of VOQs in the switch) to the crossbar switch fabric. To resolve conflicts among VOQ (0, 0), VOQ (1, 0), and VOQ (2, 0) in the case that all three request output port 0 in the same cycle, a 3x3 SA has to control the connections. The grant signals from the SA are concatenated as follows: {grant (2, 0), grant (1, 0), grant (0, 0)}. If a grant signal from the SA 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 is required for output port 1. Thus, a total number of two 3x3 SAs are needed for this 3x2 switch.

7) An **MxM distributed SA**, equivalently an **MxM hierarchical SA**, plays the same role as an MxM SA. However, an MxM distributed SA is composed of smaller SAs in the form of a hierarchical tree structure.

8) A **root SA** has the same input/output logic function as a regular MxM SA except that there is no request

signal output and no acknowledgement ("ack") input. 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 IV.B.

9) 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 SA are the same except that a SA has an extra "request" output. The use of this extra "request" output will become clear later in Section IV.B. The main difference between a BA and an SA is in typical use: a BA typically arbitrates buses while an SA typically resolves conflicts between input ports and output ports in a switch.

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 2, however, we show request connections to the SAs in more detail.

III. Related Work

Current designs in Network-on-Chip (NoC) typically use standard round-robin token passing schemes for bus arbitration [1]. In computer network packet switching, previous research in round-robin algorithms have reported results on an iterative round-robin algorithm (iSLIP) [3] and a dual round-robin matching (DRRM) algorithm [4]. Furthermore, Chao *et al*. describe a design of a round-robin arbiter for a packet switch [5]. 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 2(a). 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 block 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 N^2 VOQs in the packet switch. In general, an MxN switch can have fewer VOQs than N to save cost and area at some slight cost of occasional HOL blocking. However, we assume V=N VOQs in this paper.

Figure 2(a) 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), shown in the bottom right hand side of Figure 2(a), 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 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).

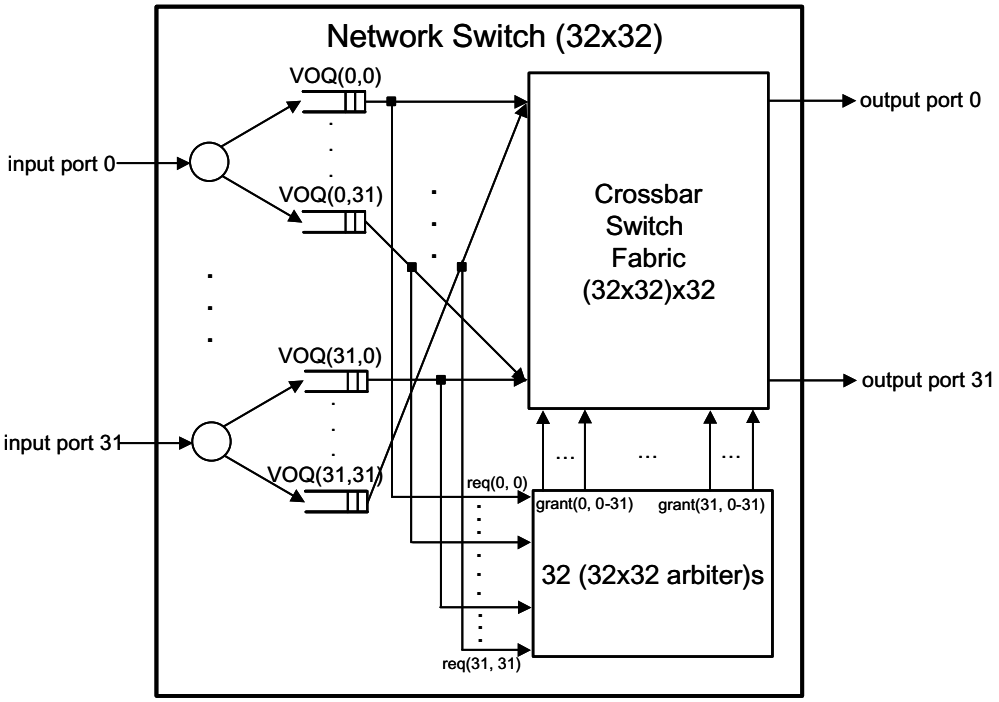

Figure 2(a). 32x32 network switch architecture.

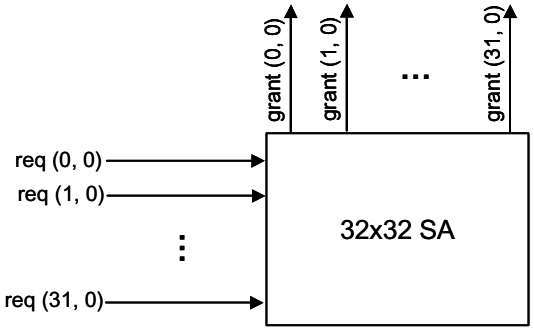

Figure 2(b). 32x32 Switch Arbiter (SA).

Figure 2(b) shows one 32x32 SA out of the thirty-two 32x32 SAs. Each SA grants one request out of at most 32 requests from thirty-two VOQs. Each input of the 32x32 SA in Figure 2(b) 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) signals the crossbar switch fabric in Figure 2(a) to connect VOQ (31, 0) to output port 0. Since the performance bottleneck of an MxN network switch is the delay of an MxM SA [5], we show how our tool can generate a fast and efficient MxM SA.

The iSLIP algorithm uses in its implementation MxM SAs. The iSLIP authors implement an MxM SA in hardware which they call a Programmable Priority Encoder (PPE) [8]. In Section VI, Experimental Results, we will compare a 128x128 SA generated by RAG to a PPE implementing a 128x128 SA and show a speedup of 2.4X. Similarly, we will compare a 128x128 SA generated by RAG to a 128x128 SA implemented by the PPA hardware described by Chao *et al.* [5], and we will show a speedup of 1.9X over PPA hardware.

Another contribution of this paper is the automated generation of a round-robin token passing Bus Arbiter (BA) to

reduce time spent on bus arbiter design. The BA is able to handle the exact number of bus masters for both on-chip and off-chip buses. The generated BA is fair, fast, and has a low and predictable worst-case wait time.

IV. Round-robin Arbiter Design

A round-robin token passing bus or switch arbiter guarantees fairness (no 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. A reliable prediction of the worst-case wait time is another advantage of the round-robin protocol. The worst-case wait time is equal to "time slot" times number of requestors minus one. The time slot is the time duration in time sharing system which is allocated to the granted requestor. 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, and the highest priority master then passes the token to the next master in round-robin order.

Section IV.A (the next section) shows the design of a Bus Arbiter (BA) generated by our tool, and Section IV.B presents a sample design of a 32x32 Switch Arbiter (SA) using our tool. A BA generated by our Round-robin Arbiter Generator (RAG) tool can handle any number of masters, while MxM SAs generated by RAG have a hierarchical structure.

IV.A. Bus Arbiter Design

Figures 3(a) and 3(b) show a BA generated to handle four requests. Figure 3(a) shows a BA block diagram for bus arbitration among four masters. To generate a BA, RAG takes as input the number of masters and produces synthesizable Verilog code at the RTL level. For synthesis of the logic generated for a four-master BA, Figure 3(b) shows the logic synthesized by the Synopsys Design Compiler [7] with a TSMC 0.25µm library from LEDA Systems [11, 15].

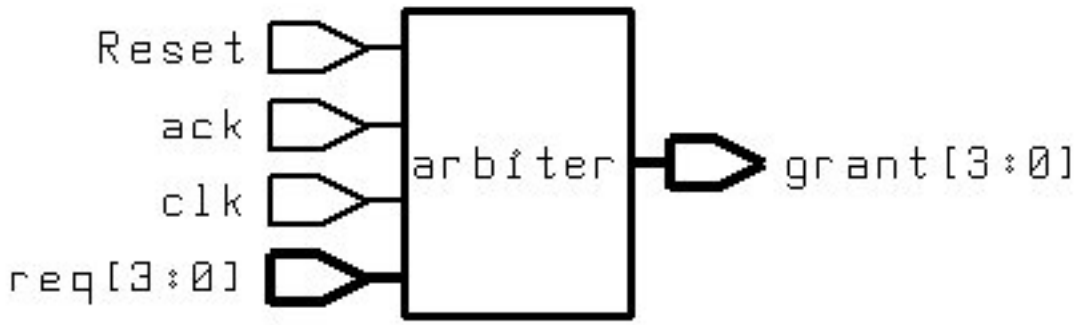

Figure 3(a). Bus arbiter block diagram.

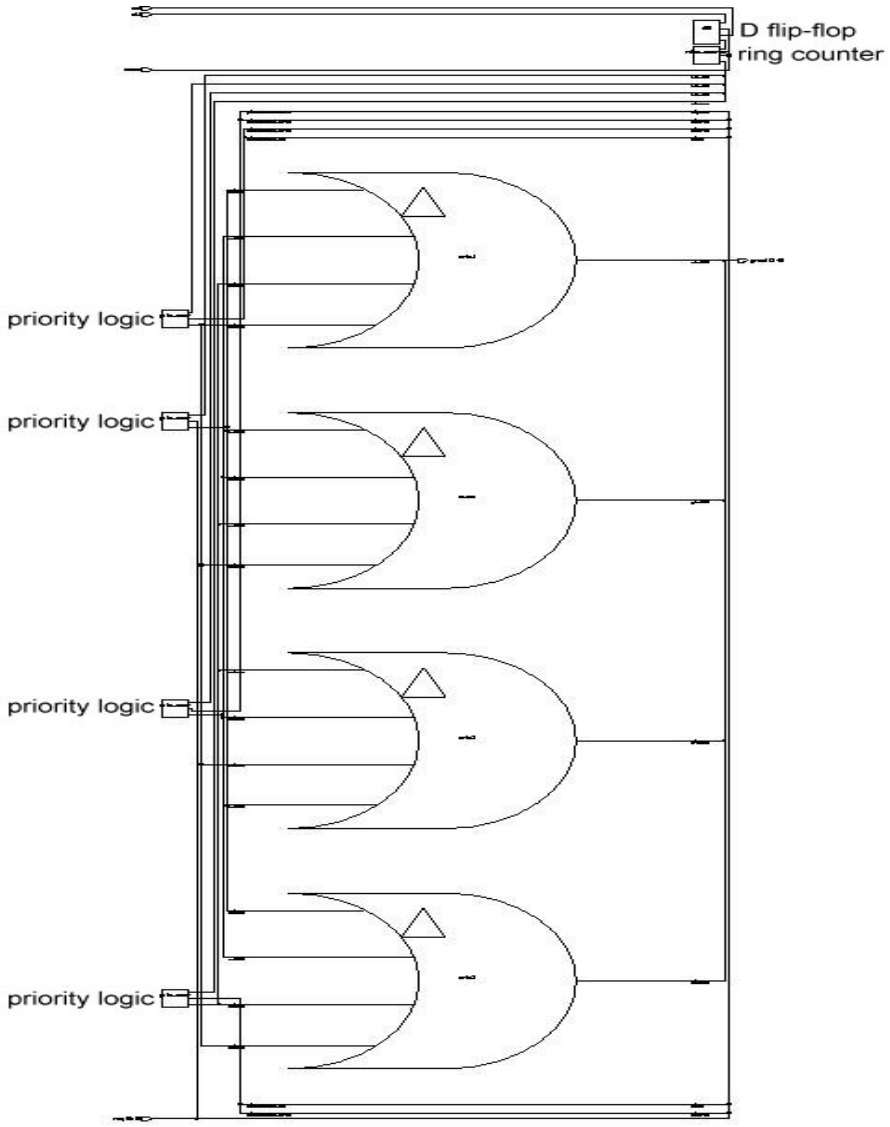

Figure 3(b). Logic diagram of 4x4 bus arbiter block.

The four small blocks on the left side of Figure 3(b) are four "priority logic" blocks in Figure 4, and the block on the top right side of Figure 3(b) is a ring counter. The functionality of a priority logic block is the same as that of a priority encoder [9] without output encoding; please see Table 1 for a truth table for a priority logic block for a 4x4 case.

The generated 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 4 where M=4. A 4x4 priority logic block is implemented in combinational logic implementing the logic function of Table 1. The priority of inputs are placed in descending order from in[0] to in[3] in the priority logic blocks (Priority Logic 0 through 3) shown in Figure 4. 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 4.

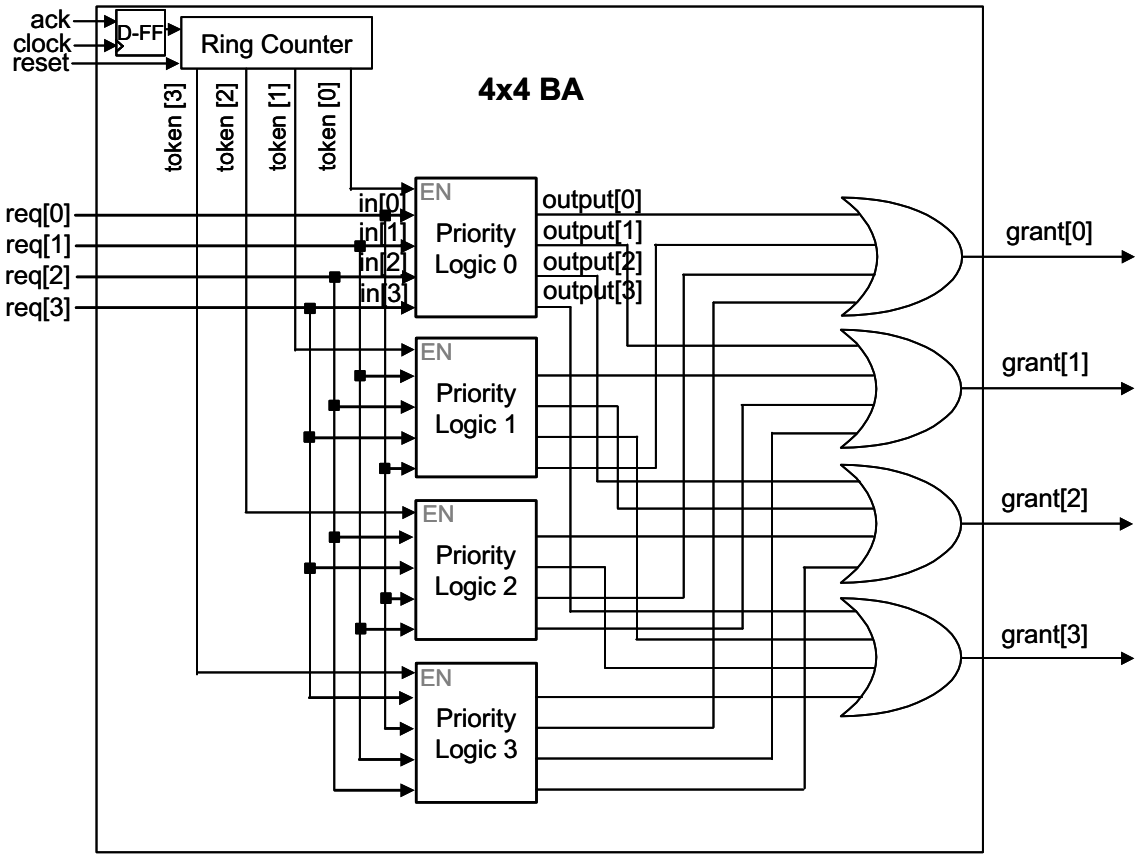

Figure 4. 4x4 Bus Arbiter (BA) architecture.

Example 3. When token = 4'b0100, processor 2 (req[2]) has the token and the highest priority in this arbitration cycle. In other words, Priority Logic 2 is enabled and req[2] has the highest priority because req[2] is connected to in[0] of the priority logic block.

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

The outputs (four bits) of the ring counter act as the enable signals to the priority logic blocks. Thus, only one enabled priority logic block can assert a grant signal. The ack signal to the bus arbiter is delayed one arbitration cycle by a D flip-flop as shown in Figure 4. The delayed 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 4, and the token is initialized to one at the reset phase (e.g., 4'b0001 for four-bit ring counter) so that there is only one '1' output by the ring counter. In the round-robin algorithm, each master must wait no longer than (M-1) time slots, 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 [13]. This round-robin protocol guarantees a dynamic priority assignment to bus masters (requestors) without starvation.

In Figure 4, 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 .25 for all request signals (req[0] – req[3]). Likewise the probability of being the second highest priority is also .25 and so on.

Example 4. In Figure 4, 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, grant[0] is the output of a 4-input OR gate whose four inputs are output[0] of Priority Logic 0, output[3] of Priority Logic 1, output[2] of Priority Logic 2 and output[1] of Priority Logic 3.

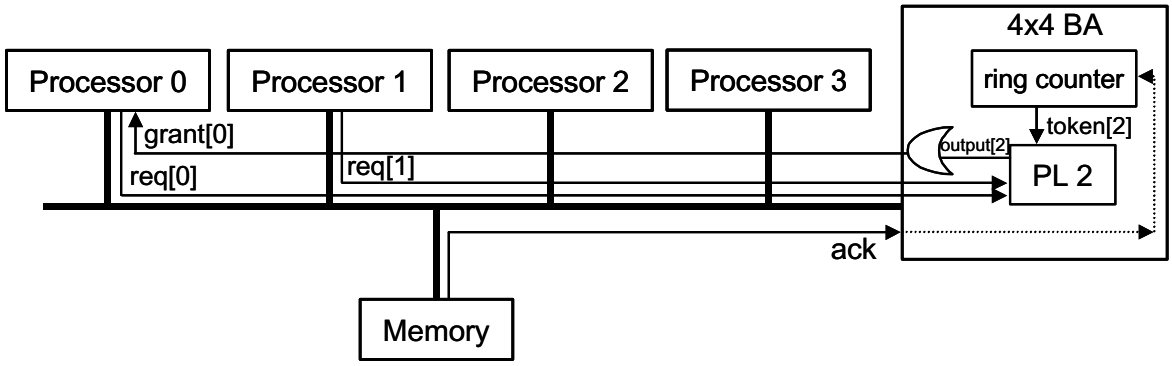

Figure 5. Four processors with a shared memory system.
(Note: bus details shown only as needed for Example 5.)

Example 5. Consider a scenario with four processors as bus masters connected to the same bus with one large shared memory on the bus as a slave as shown in Figure 5. Only the asserted signals are shown in Figure 5. 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 leads to the enabling of only Priority Logic 2 in Figure 4. In Priority Logic 2, the connection to in[0] (req[2] from processor 2) indicates the highest priority. Since req[3] is connected to in[1] of Priority Logic 2 in Figure 4, processor 3 has the next highest priority. However, since neither processor 2 nor processor 3 make 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, and then the memory controller of the accessed memory sends an ack signal, whose connection to the BA is shown in Figure 3(a), 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.

IV.B. Switch Arbiter Design

The SA generated by RAG uses 2x2 and 4x4 switch arbiter blocks as basic modules to implement an MxM switch arbiter. RAG is most efficient when M is a power of two. Figures 6(a) and 6(b) show how 2x2 and 4x4 bus arbiters are modified for switch arbiter implementation by adding some AND and OR gates to a BA. Request signals of the current level are ORed together to generate just one request to the higher level, and grant signals are ANDed together with an ack input (active high) from the higher level so that only the granted switch arbiter block can grant the corresponding master.

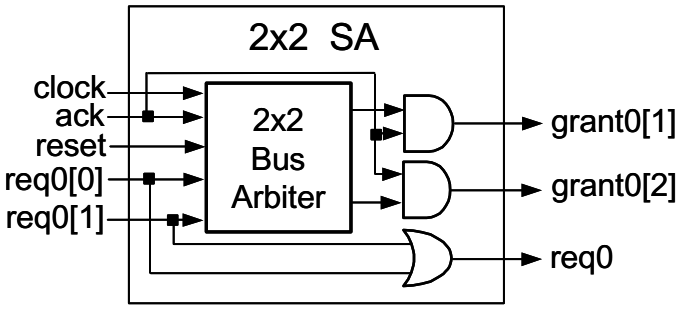

Figure 6(a). 2x2 switch arbiter block.

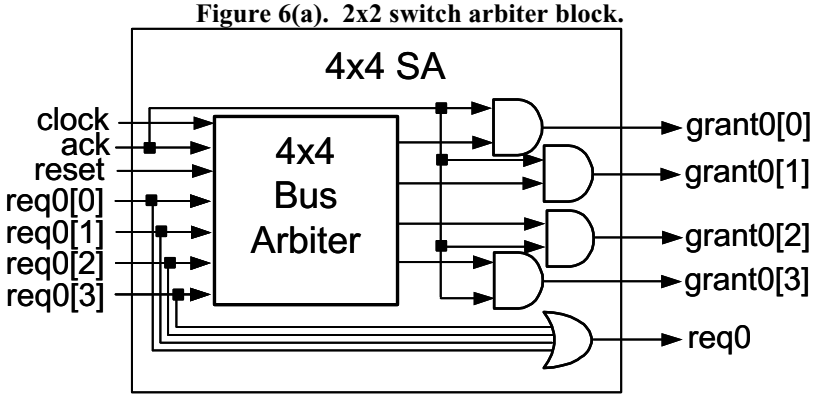

Figure 6(b). 4x4 switch arbiter block.

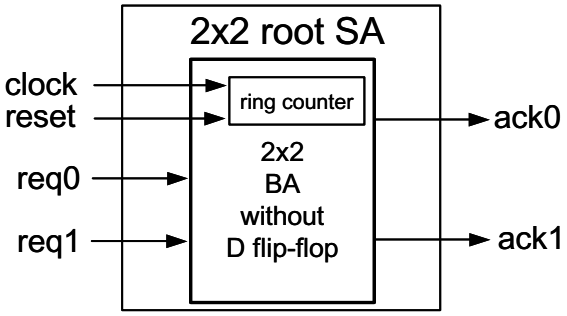

Figure 6(c). 2x2 root switch arbiter.

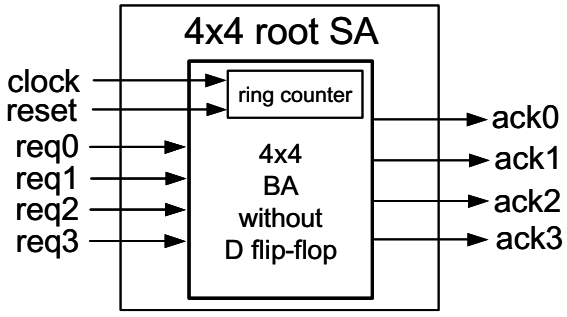

Figure 6(d). 4x4 root switch arbiter.

A root SA is placed at the top level in the hierarchy. Since there is no higher SA in the hierarchy, root SAs have no ack input nor req output. The input/output logic of a 2x2 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 Figures 6(c) and 6(d): thus, the clock input is used to rotate the content (the token bits) of the ring counter as shown in detail in Figure 6(e) for the case of a 4x4 root SA..

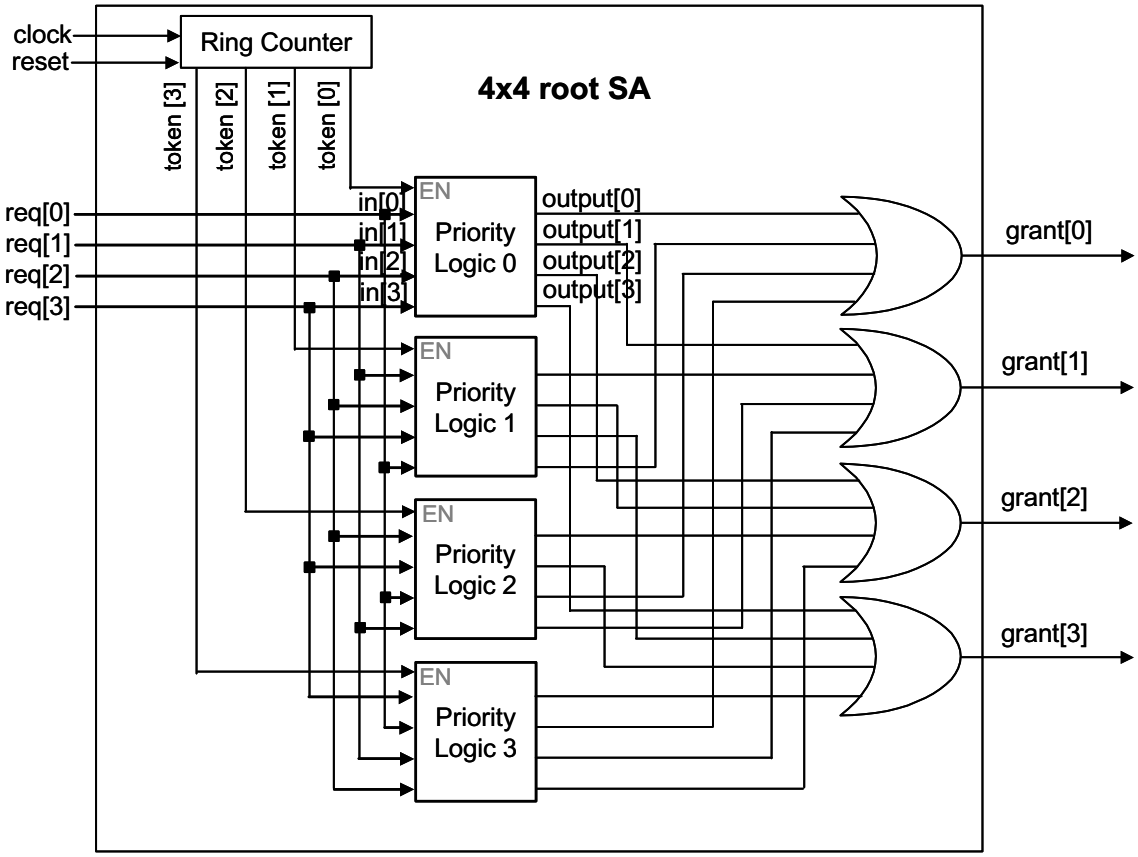

Figure 6(e). Detailed view of a 4x4 root switch arbiter.

With the 2x2 and 4x4 switch arbiter blocks as shown in Figure 6, any configuration of MxM is supported at the cost of *imperfect fairness* when M is a non-power-of-two. We define *perfect fairness* to mean that all M masters have equal probability to advance to the next round of arbitration in M arbitration cycles if all masters want to advance. The number of SA masters is preferred to be a power of 2 in order to guarantee fairest arbitration. If M is not a power of 2, *perfect fairness* is not guaranteed.

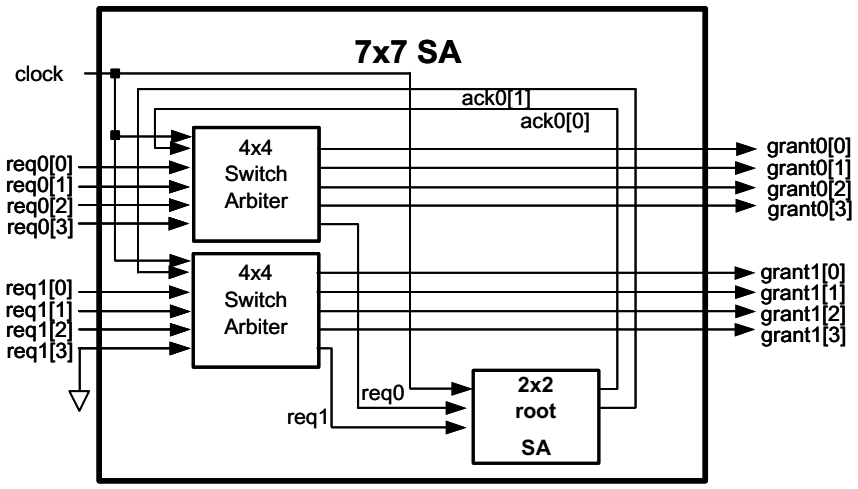

Figure 7. A 7x7 SA configuration (Note: reset signal not shown).

Example 6. Suppose a user needs a 7x7 SA. Then, the generated SA by RAG has two 4x4 switch arbiter blocks, and a 2x2 switch arbiter as shown in Figure 7. We refer to the 2x2 SA as the "root" arbiter because it is the top arbiter in the hierarchy. Figure 7 shows the tree structure which is rotated 90° to the right. The 3x3 switch arbiter block is implemented by a 4x4 switch arbiter block with the request of one input port, req1[3], set to logic 0 (see the bottom left

4x4 SA in Figure 7). The probability to be advanced to next round for request signals of a 4x4 leaf switch arbiter (req0[0] – req0[3]) is 1/4, while the probability for request signals of a 3x3 leaf switch arbiter (req1[0] – req1[2]) is approximately 1/3. Thus, the probability of acquiring the final grant from the 2x2 switch arbiter is 1/8 for request signals of a 4x4 leaf switch arbiter block and approximately 1/6 for request signals of a 3x3 leaf switch arbiter block. Hence, the probability to be granted is not equal; the probability to be granted should be 1/7 ideally (for perfect fairness).

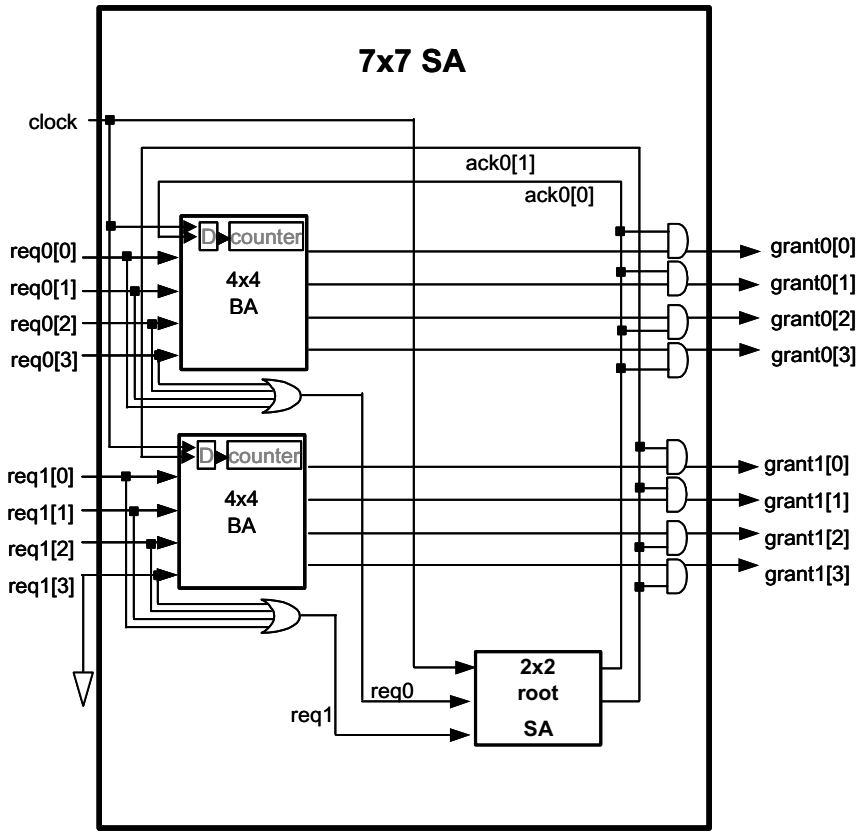

Figure 8. A 7x7 SA with the different placement of AND gates (Note: reset signal not shown).

We can redraw Figure 7 with 4x4 BAs by placing AND gates and OR gates as shown in Figure 8 (previously, the AND gates and OR gates were in the 4x4 SAs). Note that the 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 ANDed ack signals (i.e., the grant signals) are not fed back as can be verified in Figure 8. Thus, the two critical path candidates for a 7x7 SA are (i) a 4-input OR gate, a 2x2 switch arbiter 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 .25μm standard cell library from LEDA Systems is (i) a 4-input OR gate followed by a 2x2 SA followed by an AND gate. Note that ack0 signals from the root 2x2 SA feed into D flip-flops in the 4x4 BAs and thus do not affect the present arbitration cycle.

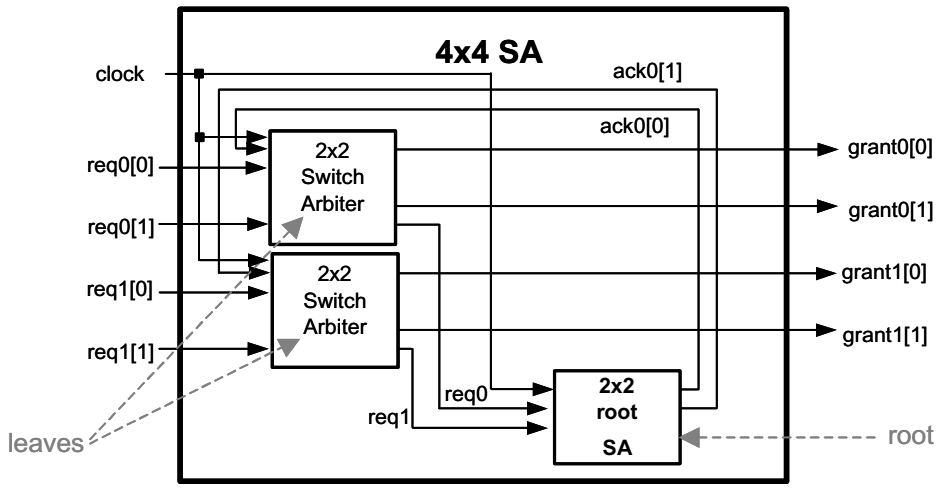

Figure 9. A 4x4 SA implemented by three 2x2 switch arbiters (Note: reset signal not shown).

To reduce the number of levels in the hierarchical Switch Arbiter (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 9. Figure 9 shows a tree structure rotated right by 90°, and we call a switch arbiter placed on the left side a "leaf" arbiter and the final switch arbiter placed on the right side we call the "root" arbiter. Moreover, the delay of a 4x4 switch arbiter block is 0.34ns in a TSMC 0.25µm library from LEDA Systems [11, 15] which is less than the delay of two levels of 2x2 switch arbiters implementing Figure 9: 0.46ns using the TSMC 0.25µm library from LEDA Systems. Also, comparing a 16x16 switch arbiter block with the combination of 4x4 switch arbiter blocks implementing a 16x16 SA, a 16x16 switch arbiter block with a 16-input priority logic block synthesized using the Synopsys Design Compiler leads to 1.49 ns gate delay using the TSMC 0.25µm library from LEDA Systems, while a combination of 4x4 (16x16 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 SAs built out of 2x2 and 4x4 basic switch arbiter blocks. This will be discussed in more detail in Section V.

These 2x2 and 4x4 switch arbiter blocks can be composed into a tree structure as shown in Figure 10(a) (the leftmost blocks are the leaves and the rightmost block is the root). Non-root 2x2 and 4x4 switch arbiter blocks receive an acknowledgement from a switch arbiter block at the next higher level (which translates to being further towards the right hand side of Figure 10(a)) for the next arbitration cycle. Since the root switch arbiter block in the hierarchy does not receive an acknowledgement (because there is no higher level switch arbiter block), the root arbiter takes the clock input so that a token is passed to the next master in every arbitration cycle in round-robin order.

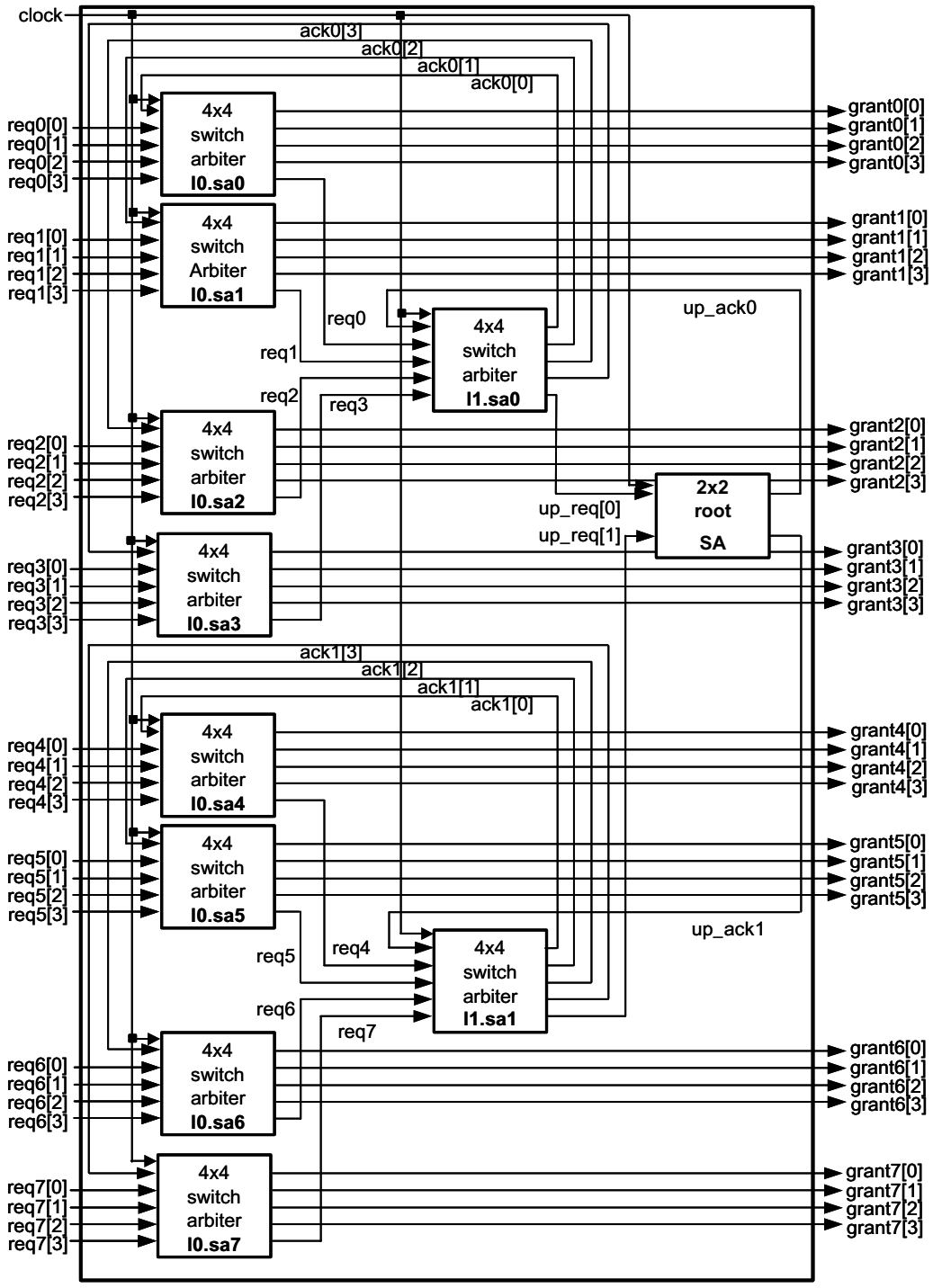

Figure 10(a). Hierarchical switch arbiter for 32 x 32 switch (Note: reset signal not shown).

Figure 10(a) shows the configuration of a 32x32 SA for a 32x32 fast switch. In Figure 10(a), "l0.sa0" denotes switch arbiter (sa) 0 in level 0 (the lowest level in the hierarchy where the left hand side of Figure 10(a) is the lowest). The 4x4 switch arbiters placed in the left side of the diagram are the lowest level (l0.sa0 through l0.sa7) in the hierarchy, and the level goes up moving towards 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 switch arbiter, SA, is a distributed switch arbiter whose individual 2x2 and 4x4 SAs operate in parallel with one another. In other words, the upper level switch arbiters (right-side arbiters, l1.sa0, l1.sa2 and root arbiter) only arbitrate their own ORed requests (for example, req0

through req3 for l1.sa0) from the lower level switch arbiters (l0.sa0 through l0.sa3) regardless of the ack signals from the higher level. The internals of ORed requests are shown in Figures 6 (a) and 6 (b). Also, ack signals from higher levels are fed back only to potentially rotate the token bit in the next arbitration cycle. Thus, the longest logic delay in Figure 10(a) is given by the Synopsys Design Compiler [7] to be as shown in Figure 10(b): two levels of ORed requests (ORed req5[0] through req5[3] and ORed req4 through req7), the gate delay of the 2x2 root arbiter, up_ack1 ANDed with ack1[1] and finishing with ack1[1] ANDed with grant5[1]. This critical path is indicated by the bold line in Figure 10(b).

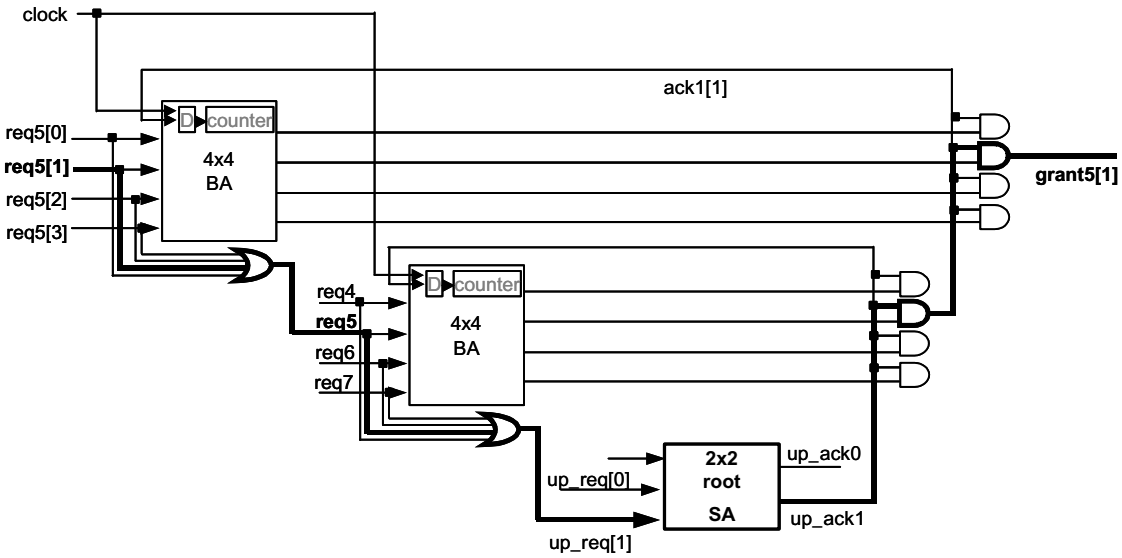

Figure 10(b). The critical path of Figure 10(a).

This scheme of Figure 10(a) 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 10(a) is automated by the RAG tool.

We compared the performance and the area of our SA with those of the Programmable Priority Encoder (PPE), implementing iSLIP, and Ping-Pong Arbiter (PPA) in Section VI. We do not compare our SA with dual round-robin matching (DRRM). As mentioned in [5] by Chao, who is the first author of both DRRM [4] and PPA [5], 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 [5], if our solution is faster than PPA, then our solution is also faster than DRRM by the same or greater margin of speedup. We choose our hierarchical SA 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 SAs as shown in Figure 9, 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 SAs 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 limit the size of

priority logic blocks (priority encoders without encoding) to 2-input or 4-input blocks to avoid the rapid increase in gate delay and area for PPE as shown in Figure 15(b) of Section VI of this paper. Furthermore, we use 2x2 and 4x4 basic SAs to implement an MxM SA.

Figure 11 describes in pseudo code of the algorithm of the 32x32 SA shown in Figure 10(a). The top if-else block is the case of the token of the 2x2 root arbiter in Figure 10(a); `if (root token =1) {` and `else {/*root token =2*/`. We show details in the pseudo code when the token of the 2x2 SA is equal to 1 (2'b01), which means l1.sa0 has the higher priority that l1.sa1in Figure 10(a).

Example 7. We describe the case when the token of the root arbiter in Figure 10(a) is equal to 1 and req0 is equal to 1. In this case, the 2x2 root arbiter sets up_ack0 equal to 1 since up_req[0] is equal to 1 and the token is equal to 1. In other words, l1.sa0 has been acknowledged and has the privilege to grant one of its child, l0.sa0 through l0.sa3. If the token of l1.sa0 is equal to 1, and req0 is not equal to logic 0, l1.sa0 acknowledge its child, l0.sa0 by setting ack0[0] equal to 1. Thus, l0.sa0 is chosen as the highest priority child switch arbiter. Then, l0.sa0 grants one of requests signals (req0[0] through req0[3]) based on a token status. Suppose the token of l0.sa0 is euqal to 4'b1000, and req0[3] is equal to 1. Then, req0[3] is the final winner and only grant0[3] is set to logic "1".

Example 8. Suppose the token of root arbiter is equal to 1 and so is the token of l1.sa0. So, up_ack0 is set to 1. Unlike Example 7, however, suppose req0 is equal to 0 meaning that all request input to l0.sa0 is equal to 0. Thus, l1.sa0 must look for the child switch arbiter in the descending order of priority. First, l1.sa0 check if req1 is 1. If so, l1.sa0 acknowledge l0.sa1 by setting ack0[1] equal to 1. Otherwise, l1.sa0 check whether req2 is 1. Consider req2 is equal to 1. Then l1.sa0 set ack0[2] euql to 1. The acknowledged switch arbiter l0.sa2, then, grants one of its request signals (req2[0] – req2[3]) based on its token.

```
Switch_Arbiter {
    if (root token = 1) {
/*I1.sa0 has higher priority*/
      up_ack0=1;
      if (I1.sa0 token = 1) {
/*I0.sa0 has higher priority*/
       if (req0) {
/*if one of requests from I0.sa0 is 1 */
        ack0[0]=1;
        if (I0,sa0 token =1) {
          if (req0[0]) grant0[0]=1;
          else if (req0[1]) grant0[1]=1;
          else if (req0[2]) grant0[2]=1;
          else if (req0[3]) grant0[3]=1;
        }
        else if (I0,sa0 token=2) {
          if (req0[1]) grant0[1]=1;
          else if (req0[2]) grant0[2]=1;
          else if (req0[3]) grant0[3]=1;
          else if (req0[0]) grant0[0]=1;
        }
                    .
                    .
                    .

        else if (I0,sa0 token=8) {
          if (req0[3]) grant0[3]=1;
          else if (req0[0]) grant0[0]=1;
          else if (req0[1]) grant0[1]=1;
          else if (req0[2]) grant0[2]=1;
        }
      }
      else if (req1) {
/*if one of requests from I0.sa1 is 1 */
        ack0[1]=1;
        if (I0,sa1 token =1) {
          if (req1[0]) grant1[0]=1;
          else if (req1[1]) grant1[1]=1;
          else if (req1[2]) grant1[2]=1;
          else if (req1[3]) grant1[3]=1;
        }
        else if (I0,sa1 token=2) {
          if (req1[1]) grant1[1]=1;
          else if (req1[2]) grant1[2]=1;
          else if (req1[3]) grant1[3]=1;
          else if (req1[0]) grant1[0]=1;
        }
                    .
                    .
                    .

        else if (I0,sa1 token=8) {
          if (req1[3]) grant1[3]=1;
          else if (req1[0]) grant1[0]=1;
          else if (req1[1]) grant1[1]=1;
          else if (req1[2]) grant1[2]=1;
        }
      }

      else if (req2)
/*if one of requests from I0.sa2 is 1 */
        ack0[2]=1;
        if (I0,sa2 token =1) {
          if (req2[0]) grant2[0]=1;
          else if (req2[1]) grant2[1]=1;
          else if (req2[2]) grant2[2]=1;
          else if (req2[3]) grant2[3]=1;
        }
        else if (I0,sa2 token=2) {
          if (req2[1]) grant2[1]=1;
          else if (req2[2]) grant2[2]=1;
          else if (req2[3]) grant2[3]=1;
          else if (req2[0]) grant2[0]=1;
        }
                    .
                    .
                    .

        else if (I0,sa2 token=8) {
          if (req2[3]) grant2[3]=1;
          else if (req2[0]) grant2[0]=1;
          else if (req2[1]) grant2[1]=1;
          else if (req2[2]) grant2[2]=1;
        }
      }
      else if (req3) {
/*if one of requests from I0.sa3 is 1 */
        ack0[3]=1;
        if (I0,sa3 token =1) {
          if (req3[0]) grant3[0]=1;
          else if (req3[1]) grant3[1]=1;
          else if (req3[2]) grant3[2]=1;
          else if (req3[3]) grant3[3]=1;
        }
        else if (I0,sa3 token=2) {
          if (req3[1]) grant3[1]=1;
          else if (req3[2]) grant3[2]=1;
          else if (req3[3]) grant3[3]=1;
          else if (req3[0]) grant3[0]=1;
        }
                    .
                    .
                    .

        else if (I0,sa3 token=8) {
          if (req3[3]) grant3[3]=1;
          else if (req3[0]) grant3[0]=1;
          else if (req3[1]) grant3[1]=1;
          else if (req3[2]) grant3[2]=1;
        }
      }
      else if (I1.sa0 token = 2) {
/*I0.sa1 has higher priority*/
        if (req1) {
/*if one of requests from I0.sa1 is 1 */
          ack0[1]=1;
          if (I0,sa1 token =1) {
            if (req1[0]) grant1[0]=1;
            else if (req1[1]) grant1[1]=1;
            else if (req1[2]) grant1[2]=1;
            else if (req1[3]) grant1[3]=1;
          }
          else if (I0,sa1 token=2) {
            if (req1[1]) grant1[1]=1;
            else if (req1[2]) grant1[2]=1;
            else if (req1[3]) grant1[3]=1;
            else if (req1[0]) grant1[0]=1;
          }
                      .
                      .
                      .

          else if (I0,sa1 token=8) {
            if (req1[3]) grant1[3]=1;
            else if (req1[0]) grant1[0]=1;
            else if (req1[1]) grant1[1]=1;
            else if (req1[2]) grant1[2]=1;
          }
        }
        else if (req2) {
/*if one of requests from I0.sa2 is 1 */
          ack0[2]=1;
          if (I0,sa2 token =1) {
            if (req2[0]) grant2[0]=1;
            else if (req2[1]) grant2[1]=1;
            else if (req2[2]) grant2[2]=1;
            else if (req2[3]) grant2[3]=1;
          }
          else if (I0,sa2 token=2) {
            if (req2[1]) grant2[1]=1;
            else if (req2[2]) grant2[2]=1;
            else if (req2[3]) grant2[3]=1;
            else if (req2[0]) grant2[0]=1;
          }
                      .
                      .
                      .

          else if (I0,sa2 token=8) {
            if (req2[3]) grant2[3]=1;
            else if (req2[0]) grant2[0]=1;
            else if (req2[1]) grant2[1]=1;
            else if (req2[2]) grant2[2]=1;
          }
        }
        else if (req3) {
/*if one of requests from I0.sa3 is 1 */
          ack0[3]=1;
          if (I0,sa3 token =1) {
            if (req3[0]) grant3[0]=1;
            else if (req3[1]) grant3[1]=1;
            else if (req3[2]) grant3[2]=1;
            else if (req3[3]) grant3[3]=1;
          }
          else if (I0,sa3 token=2) {
            if (req3[1]) grant3[1]=1;
            else if (req3[2]) grant3[2]=1;
            else if (req3[3]) grant3[3]=1;
            else if (req3[0]) grant3[0]=1;
          }
                      .
                      .
                      .

          else if (I0,sa3 token=8) {
            if (req3[3]) grant3[3]=1;
            else if (req3[0]) grant3[0]=1;
            else if (req3[1]) grant3[1]=1;
            else if (req3[2]) grant3[2]=1;
          }
        }
      }
      else if (I1.sa0 token = 4) {
/*I0.sa2 has higher priority*/
                      .
                      .
                      .
      }

    else { /*root token =2 */
/*I1.sa1 has higher priority*/
      up_ack1=1;
      if (I1.sa1 token = 1) {
/*I0.sa4 has higher priority*/
                      .
                      .
                      .
      }
      else if (I1.sa1 token = 2) {
/*I0.sa5 has higher priority*/
                      .
                      .
                      .
      }
      else if (I1.sa1 token = 4) {
/*I0.sa6 has higher priority*/
                      .
                      .
                      .
      }
      else if (I1.sa1 token = 8) {
/*I0.sa7 has higher priority*/
                      .
                      .
                      .
      }
```

Figure 11. Switch Arbiter Algorithm.

V. IMPLEMENTATION OF THE RAG TOOL

Our Round-robin Arbiter Generator (RAG) tool can generate synthesizable Verilog for a Bus Arbiter (BA) able to handle any number of requests. RAG can also combine switch arbiter blocks (2x2 and 4x4 switch arbiters) to

produce synthesizable Verilog for a hierarchical fast MxM switch arbiter.

V.A. RAG Tool

Figure 12 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 algorithm for a bus arbiter is straightforward because the tool just generates bus arbiter logic for the exact number of masters specified. The truth table shown in Table 1 in Section IV shows the regular pattern as M increases and is implemented with simple logic equations in Verilog for the generation of a Bus Arbiter (BA). From Figures 14 and 15, it turns out that employing a Switch Arbiter (SA) to implement the BA logic is better in terms of area and speed when the number of masters (M) is greater than 4 at the cost of possible waste of ports for the case that M is not a power of 2. We will discuss this in more detail in Section V.B. In short, a parametrizable Verilog version of Table 1 suffices to generate a BA with any specific number of masters.

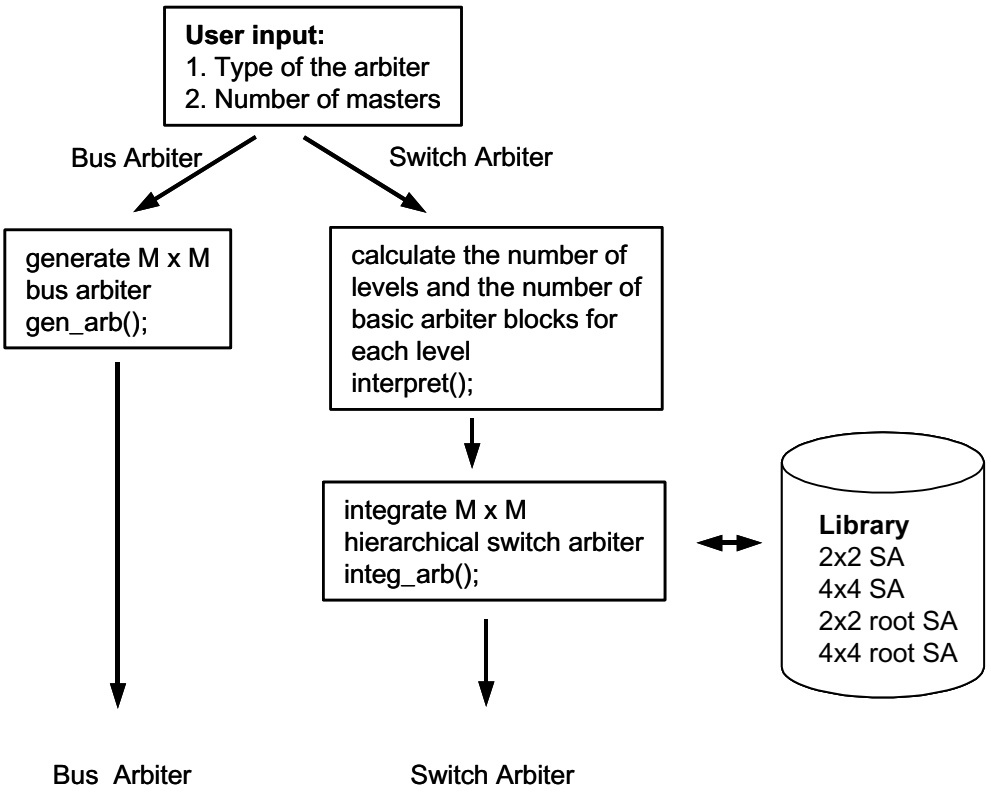

Figure 12. Flow of RAG tool.

For an MxM Switch Arbiter (SA), the tool divides M by 4 to decide the maximum allowable number of 4x4 switch arbiters that can be used in an MxM SA. First, 4x4 switch arbiters are employed as many as the quotient of M divided by 4. If M modulo 4 is not equal to 0, a 2x2 switch arbiter is used when the remainder is less than 3. Otherwise another 4x4 switch arbiter is employed with one unused request (one request signal, say req[0] in Figure 3(a), is set to '0'). Figure 13 describes the algorithm of the function interpret() which calculates the number of levels in the hierarchy and the number of 4x4 and 2x2 SAs employed in each level. Finally, RAG generates a Verilog top file to integrate basic switch arbiter block into an MxM SA. A top file instantiates 2x2 and 4x4 SAs based on the data passed from the function, interpret(). To get the optimum performance, M is preferred to be a

power of two.

```
Num_In_Level {
               /* initialization */
               set number_of_level equal to 0;
               set dividend equal to total number of master;
               set remainder equal to (dividend modulo 4);

               /* calculate the number of levels in the hierarchy */
               while (dividend != 0) {
                              set dividend equal to integer of (dividend /4);
                              increment number_of_level by 1;
                              if (dividend==1 && remainder ==0) break;
               }

               /* calculate the number of 4x4 and 2x2 SAs in each level */
               set dividend equal to total number of master again;
               for (n=0 to n<number_of_level) {
                              /* calculate the number of 4x4 SAs in level n */
                              if (dividend > 2) {
                                             set remainder equal to (dividend modulo 4);
                                             set dividend equal to integer of (dividend /4);
                                             set 4by4_SA_in_level(n) equal to dividend;
                                             /* if total number of masters is not a multiple of 4 */
                                             if (remainder !=0) {
                                                            /* if remainder is 3, then place one more 4x4 SA */
                                                            if (remainder > 2)
                                                                           increment 4by4_SA_in_level(n) by 1;
                                                            /* else place 2x2 SA */
                                                            else
                                                                           increment 2by2_SA_in_level(n) by 1;
                                             }
                                             increment n by 1;
                              }
                              /* calculate the number of 2x2 SAs in level n */
                              else {
                                             set remainder equal to (dividend modulo 2);
                                             set dividend equal to integer of (dividend /2);
                                             set 2by2_SA_in_level(n) equal to dividend;
                                             /* if total number of masters is not a multiples of 4 nor 2 */
                                             if (remainder != 0) {
                                                            if (remainder > 2)
                                                                           increment 4by4_SA_in_level(n) by 1;
                                                            /* else place 2x2 SA */
                                                            else
                                                                           increment 2by2_SA_in_level(n) by 1;
                                             }
                                             increment n by 1;
                              }

                              set dividend equal to sum of 4by4_SA_in_level(n) and 2by2_SA_in_level(n);
               }
}
```

Figure 13. Pseudo code for the calculation of number of 2x2 and 4x4 SA and the number of levels in the hierarchy.

Example 9. This example describe how the 32x32 SA as shown in Figure 10(a) handles 32 requests and grants one request signals. First, variables `number_of_level`, `dividend` and `remainder` initializes as `0`s, `total number of master` (32 in this example) and `dividend modulo 4`, respectively. `while` loop 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; (`dividend`=2)/4 is equal to 0 since this result is typecasted to integer at the third iteration. Thus, `number_of_level` is determined as 3.

`dividend` is set back to 32 after `while` block. `for` loop iterates as many as the number of levels in the hierarchy. At the first iteration, since `dividend` is greater 2, the outer `if` block is executed. `remainder` is determined to 0, and `dividend` is set to 8, which is the `4by4_SA_in_level(0)` in the level 0. Since `remainder` is 0 the number of 4x4 switch arbiters at the level 0 is determined to 8, and `dividend` is set to the sum of `4by4_SA_in_level(0)` (=8) and `2by2_SA_in_level(0)` (=0), which is 8. In the second iteration, the outer `if` block is executed again and returns `4by4_SA_in_level(1)` as 2, `2by2_SA_in_level(1)` as 0 and `dividend`

as 2. In the last execution, `else` block is executed since `dividend` is not greater than 2. First, `remainder` is set to 0 and `dividend` to 1 which is the same as `2by2_SA_in_level(2)`. Thus, `Num_In_Level` function returns eight 4x4 switch arbiters for the level0, two 4x4 switch arbiters for the level 1 and one 2x2 switch arbiter for level 2 which is the root.

After the tool decides the number of 2x2 and 4x4 switch arbiter blocks for each level in the tree structure, the tool will integrate switch arbiters for each level and produce an MxM SA.

V.B. Area and delay considerations

Figures 14(a) and 14(b) show the area and delay, respectively, of BAs generated by RAG. The area increases more than linearly as the number of masters (M) increases. As can be seen in Figure 14(b), delay increases linearly as M increases.

From Figure 14 and Figure 15, for minimal area and delay it is better to employ an SA when the number of masters is greater than 4 and is a power of two. However, when the number of masters is not a power of two, a user might prefer to choose the bus arbiter option in our tool because of possible waste of ports in the generated SA. Our tool can generate a BA that can handle the exact number of masters (including non-powers of two). Figure 14 and Figure 15 help a user to choose between BA and SA options with consideration of area and delay.

As shown in Figure 14(b), the increasing delay for our generated BA as M increases will limit the achievable switching speed in a fast switch. We found that limiting the size of the switch arbiter blocks, used as the components of a SA, to 2x2 and 4x4 yielded the fastest switching speeds. Our RAG tool is favored to utilize 4x4 switch arbiter blocks rather than 2x2 switch arbiter blocks. Employing 4x4 switch arbiter blocks 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 4x4 switch arbiter block, see Figure 9 in Section IV.B).

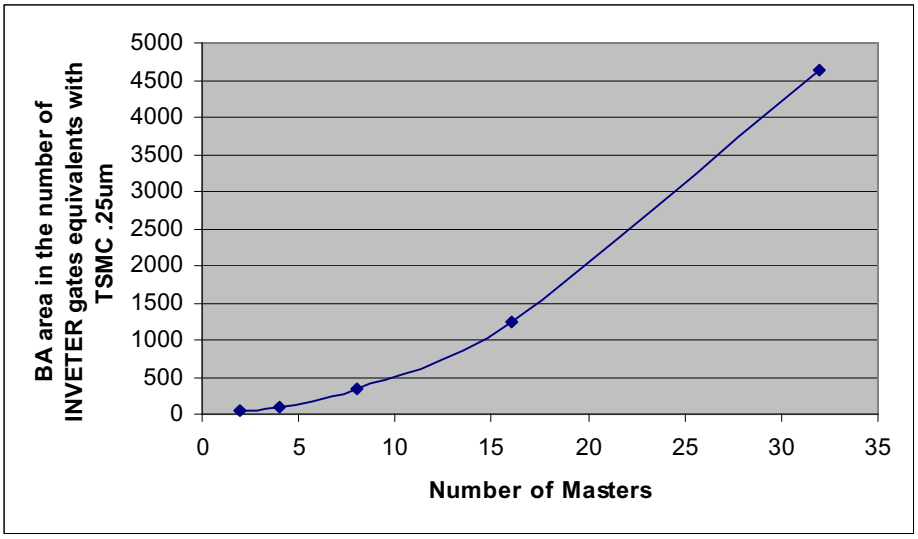

Figure 14(a). Area of MxM bus arbiter

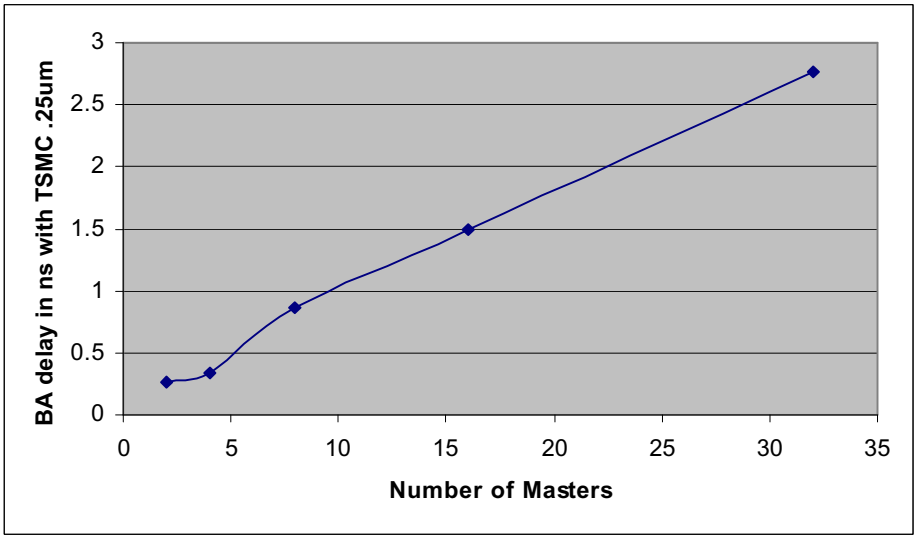

Figure 14(b). Gate delay of MxM bus arbiter

VI. EXPERIMENTAL RESULTS

In this section, we first compare area and delay between a Ping-Pong Arbiter (PPA) [5], a Programmable Priority Encoder (PPE) [8] for iSLIP [3], and our generated Switch Arbiter (SA). We do not consider including dual round-robin matching (DRRM) algorithm in this comparison as we mentioned in Section IV.B.

Then, we will show a speedup for our generated SA over a PPA and a PPE. First we explain areas and delays of the three switch arbiters, then speedups achieved by the SA generated by RAG.

VI.A. Areas and Delays

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 switch arbiter. 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 6(a). One difference is that every 2x2 switch arbiter of PPA receives acknowledgements from two higher levels and ANDs them together with the current level grants.

A PPE for iSLIP is a centralized switch arbiter. PPE adds a programmable functionality to the priority encoder so that the grant pointer can point to the next request after the current arbitration [8]. The delay and area of a priority encoder in PPE becomes larger as M increases and is shown in Figures 15 (a) and 15(b).

In [5], the authors compare PPA with iSLIP [3] and DRRM [4]. The performance of PPA is very competitive with speedup c=2. The speedup c of the switch fabric is the ratio of the switch fabric bandwidth and input link bandwidth. Since the arbitration protocol of SA is almost the same as that of PPA, we assume that the performance of a packet switch using SA and a packet switch using PPA are comparable with each other, with the major difference only in logic (arbitration) delay which was found to be the critical path in PPA. Thus, we just compare the area and the longest delay of SA with those of PPA and PPE for iSLIP in different MxM configurations.

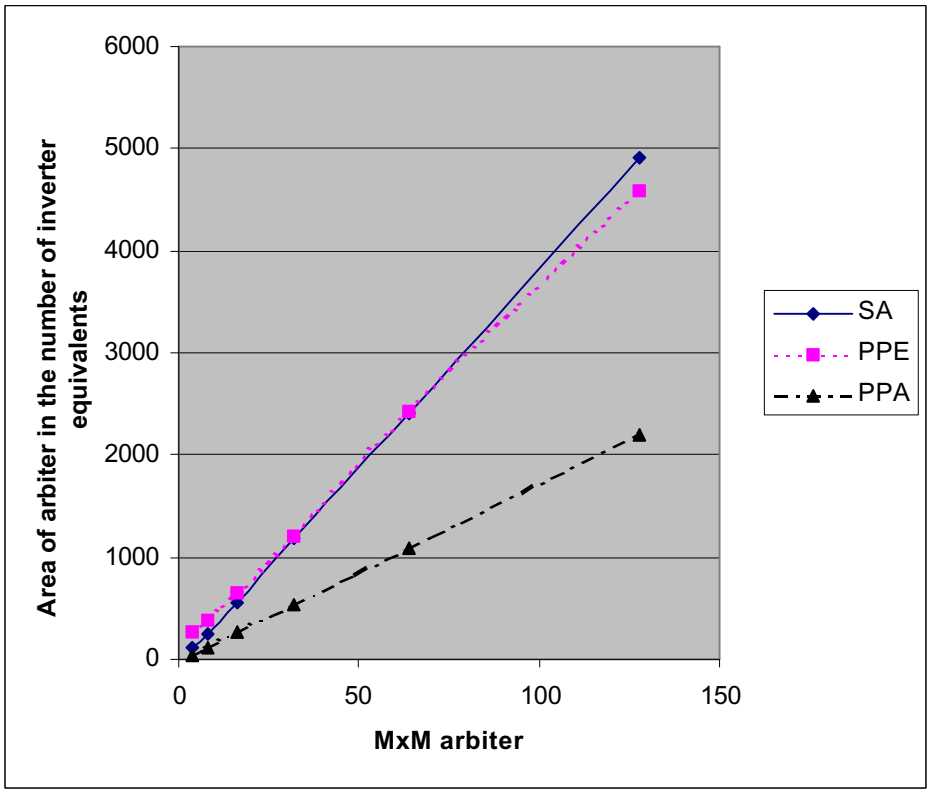

Figure 15(a). The area of MxM switch arbiter.

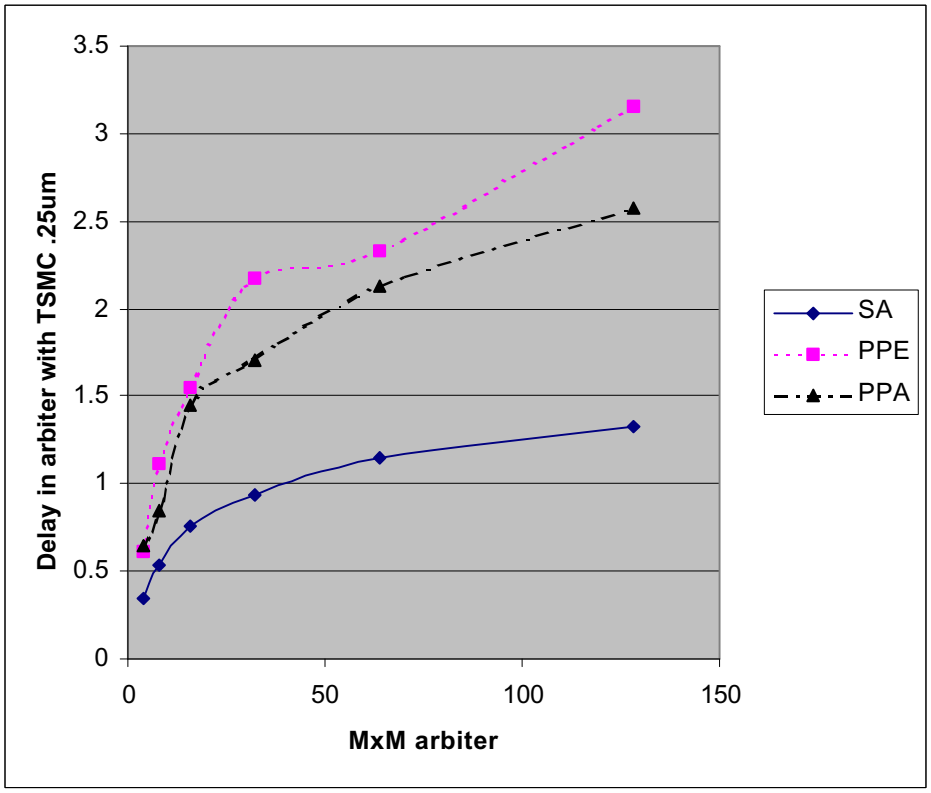

Figure 15(b). The longest delay of MxM switch arbiter.

Figures 15(a) and 15(b) show the area and the longest switch arbiter logic delays of SA, PPE and PPA, respectively. The area of PPE is synthesized and complied with Texas Instruments TSC5000 0.25µm

technology [3], while SA and PPA with TSMC 0.25μm technology both by Synopsys Design Compiler [7]. We modeled the priority encoder and added some gates as shown in Figure 11 of [8] to measure the delay with TSMC 0.25 technology. The delay measured here is well matched with Table 2 in [8] except for the case where M=32[1] (32% longer delay in our experiment). We modeled PPA by writing Verilog code based on the logic diagram in [5] and synthesized PPA with the Synopsys Design Compiler [7] to estimate the area and delay. The areas of all switch arbiters increase linearly as M increases. The area of SA is almost 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 from 32 to 128. This is because we first limit the size of basic switch arbiters to 2x2 and 4x4 to reduce the critical path delay due to the expansion of priority logic blocks as M increases and apply the 2x2 and 4x4 switch arbiters 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.

VI.B. Speedup for a Terabit switch

For comparison purposes with PPA and PPE, suppose we have a 128x128 switch with eight-byte cell size, and the speed of the serial link is wholly determined by the arbitration cycles. Thus, the serial link capacity is equal to 64-bit/128x128 switch arbiter delay. Then, the aggregated bps capacities of SA, PPA, and PPE are 6.16Tbps, 3.18Tbps, and 2.59Tbps, respectively. This can be verified by using the delays for SA, PPA, and PPE shown in Figure 15(b). Thus, for this comparison, the RAG generated arbiter achieves throughput 1.9X higher than PPA and 2.4X higher than PPE.

Currently some commercial terabit switches are available. One is from Mindspeed, M21155 [10, 16]. M21155 is a 144-port x 144-port switch, each port delivering a data rate of up to 3.125Gbps; the aggregate capacity is 0.45Tbps. The other is PetaSwitch [12] 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., .25μm) used is publicly available for either of these chips.

VII. CONCLUSION

In this paper, we introduced a Round–robin Arbiter Generator (RAG) tool. RAG can generate a BA to handle the exact number of bus masters for both on-chip and off-chip buses. RAG can also generate a parallel hierarchical MxM switch arbiter. We discussed the BA logic and showed the logic of 2x2 and 4x4 SA components. We also presented how RAG uses 2x2 and 4x4 switch arbiter blocks to produce a hierarchical MxM switch arbiter. The first contribution of this paper is the automated generation of a round-robin token passing Bus Arbiter (BA) to reduce bus

[1] We are not sure exactly why this difference exists for M=32. But it may have to do with differences in the TSC5000 .25um process and/or standard cell library used versus the TSMC .25um process and/or standard cell library used.

design time. The generated BA is fair, fast, and has a low and predictable worst-case wait time. The second contribution of this paper is the automated generation of an MxM 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 area and delay improvements when compared with other switch arbiters such as PPA and PPE. Specifically, RAG can be used to generate a switch arbiter for a 128x128 terabit switch which, assuming as stated in [5] that the critical path is the switch arbitration logic, achieves throughput 1.9X higher than PPA and 2.4X higher than PPE for the same 128x128 configuration.

ACKNOWLEDGMENT

We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, SUN, and Synopsys.

REFERENCES

[1] W. J. Dally and B. Towels, "Route, Packets, Not Wires: On-Chip Interconnection Networks," *Proceedings of IEEE Design Automation Conference*, 2001, pp. 684-689.
[2] F. A. Tobagi, "Fast Packet Switch Architecture for Broadband Integrated Services Digital Networks," *Proceedings of IEEE*, January 1990, pp. 133-167.
[3] N. Mckeown, P. Varaiya, and J. Warland, "The iSLIP Scheduling Algorithm for Input-Queued Switch," *IEEE Transaction on Networks*, 1999, pp. 188-201.
[4] H. J. Chao and J. S. Park, "Centralized Contention Resolution Schemes for a Larger-capacity Optical ATM Switch," *Proceedings of IEEE ATM Workshop*, 1998, pp. 11-16.
[5] H. J. Chao, C. H. Lam, and X. Guo, "A Fast Arbitration Scheme for Terabit Packet Switches," *Proceedings of IEEE Global Telecommunications Conference*, 1999, pp. 1236-1243.
[6] Y. Tamir and H-C. Chi, "High Performance Multi-queue buffers for VLSI Communications switches," *IEEE Transaction on Communications*, 1987, pp. 1347-1356.
[7] Synopsys, Design Compiler, Available HTTP: http://www.synopsys.com/products/logic/design_comp_cs.html.
[8] P. Gupta and N. Mckeown, "Designing and Implementing a Fast Crossbar Scheduler," IEEE Micro, 1999, pp. 20-28.
[9] J. Wakerly, *Digital Design Principles and Practices*, NJ: Prentice Hall, Inc., 1990, pp. 402-407.
[10] P. Rigby, "Mindspeed unveils terabit switch chip," Network World Fusion Newsletter, 12/12/01, Available HTTP:http://www.nwfusion.com/newsletters/optical/2001/01142734.html.
[11] TSMC, "IP Services," Available HTTP:http://www.tsmc.com/design/ip.html.
[12] PetaSwitch Product, Available HTTP: http://www.peta-switch.com.
[13] A. Silberschatz, P. Galvin, G. Gagne, *Applied Operation System Concepts*, NY: John Willey and Sons, Inc., 2000.
[14] W. Stallings, *Data and Computer Communications*, Fifth Edition, NJ: Prentice Hall, 1997.
[15] LEDA System, Available HTTP: http://www.ledasys.com.
[16] Mindspeed, Available HTTP: http//www.mindspeed.com.