AN IN-DEPTH LOOK AT PRIOR ART IN FAST ROUND-ROBIN ARBITER CIRCUITS

ÖZYEĞİN UNIVERSITY TECHNICAL REPORT: OZU-EF-2011-0001

Özyeğin University Technical Report: OZU-EF-2011-0001

An In-Depth Look at Prior Art in Fast Round-Robin Arbiter Circuits

H. Fatih Ugurdag
Özyeğin University
fatih@ugurdag.com

Onur Baskirt
Ericsson Turkey
onur_baskirt@yahoo.com

Abstract

Arbiters are found where shared resources exist such as busses, switching fabrics, processing elements. Round-robin is a fair arbitration method, where requestors get near-equal shares of a common resource or service. Round-robin arbitration (RRA) finds use in network switches/routers and processor boards/systems as well as many other applications that have concurrency. Today's electronic systems require arbiters with hundreds of ports (e.g., switching fabrics with virtual I/O queues) and clock speeds near the limits of even the latest microelectronics fabrication processes/libraries. Achieving high clock speeds in the presence of large number of ports is only possible with highly parallel arbiter architectures. This paper presents an in-depth literature survey of previous work on this problem. It looks at RRA work in the literature in a bigger context, then defines the typical RRA problem (RRA_typical), and specifically investigates work on fast architectures that solve the RRA_typical problem. There are five such works that are really competitive. This report takes a very in-depth look at these works. It explains each architecture and how/why it works from a unique perspective that cannot be found in the original publication of that architecture. It also proposes improvements to these architectures. We wrote generators for the improved versions of these architectures. We will share a summary of synthesis results in this report – although a detailed account of how these results were obtained and their analysis is the subject of another (upcoming) publication.

1. Introduction

This paper deals with fast (and preferably area-efficient) round-robin arbiter hardware architectures. Arbitration implies presence of concurrency (of requestors at the least). Most systems have concurrency – implicit or explicit. Explicit concurrency is when a system offers multiple services and/or serves multiple agents. Implicit concurrency, on the other hand, is when a system offers a single service but then uses internal multi-threading. If the concerned multiple threads/agents do not need service 100% of the time, then a design with shared resources is most often the best and sometimes the only viable approach. On top of sharing resources, if the agents also exhibit a dynamic (i.e., not a priori known) behavior in terms of when they require the shared resources, some sort of arbitration is needed. Although arbitration can be done by peer-to-peer protocols, a central arbiter offers better utilization of shared resources. And that is the subject of this paper.

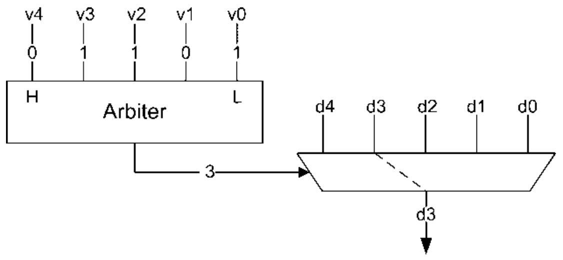

Fig. 1. An arbiter arbitrating a data bus. (The v's are valid flags and the d's are data.)

Fig. 1 shows an arbiter that arbitrates a multiplexed bus. Arbitration is needed, for example, when a system-on-chip (SOC) has one or more shared buses for its multiple IP blocks to communicate and access peripherals and shared memory [1,2]. Another example is when a switch fabric needs to concurrently route data between its multiple ingress (i.e., input) and multiple egress (i.e., output) ports. In this example, sharing is involved even when there are more egress ports than ingress ports – as different ingress ports may at the same time request the same egress port. Arbitration is part of the picture not only when there is shared interconnect but also any other kind of shared resources such as shared peripherals and shared processing elements. Arbitration needs to be fair (i.e., balanced) usually not from the point of view of the shared resources but from the point of view of the requestors sharing those resources. If there are different priority levels among the requestors, then fairness is needed within each priority level, and a certain service ratio may be required across priority levels.

1.1. What is Round-Robin Arbitration (RRA)?

RRA (Fig. 3) is a relatively simple way of providing fairness in a system with equal-priority requestors. Suppose there are 4 requestors, namely, A, B, C, and D. RRA keeps an ordered list of the requestors descending from high-priority to low-priority. It picks (i.e., grants) the highest priority requestor with a pending request. It indicates the requestor it picked (out of n request lines, i.e., bits) by outputting its index. The index can be expressed as a single number with ceiling($\log_2 n$) bits and an additional V bit (i.e., valid) – hence *n*2log*n* RRA. It is also possible to indicate the output of the RRA with a one-hot n-bit bit-vector (where no V bit is needed) – hence *n*2*n* arbiter as shown in Fig. 2. The RRA architectures studied here use the *n*2*n* style.

The *n*2log*n* RRA has two drawbacks:

(i) Excessive wiring at its output. Although its output port has fewer wires, every wire fans out to all requesters. Therefore, in reality, it has *n*log*n* wires, not log*n*.

(ii) Requires a comparator in each requestor to tell if the arbitration picked an index equal to the index of that particular requestor.

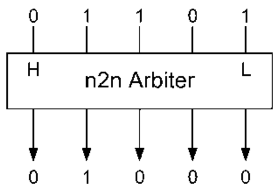

Fig. 2. An *n*2*n* arbiter.

RRA achieves fairness by rotating the "arbitrated requestor" to the low-priority end of the list. Suppose we start with an ordered list of {A, B, C, D}, as in Fig. 3a, where A is the highest priority requestor for the current clock cycle and D is the lowest. Let the request vector be 0101 (i.e., the inputs in Fig. 3a). RRA scans the request vector looking for a 1 starting from the high-priority bit and going in the direction of the arrow, and it stops at the first 1 (i.e., B). Therefore, B is arbitrated as the output of RRA block shows in Fig. 3a. To make it fair, the RRA now "rotates" the priority list so that B is the lowest priority in the list (as the L in Fig. 3b indicates). As a result, the list becomes {C, D, A, B} for the next clock cycle. This method may seem unfair as A has also been penalized although it did not use its turn when it was a high-priority requestor. However, simulations show that RRA is quite fair over a decent number of arbitration cycles.

With its simplified approach to priority, RRA can keep track of the current state of the above priority list with a 2-bit number ($\log_2 n$ with n requestors), which identifies the head or tail of the list. Instead of doing RRA, if we move only the arbitrated requestor to the end of the list (to be completely fair), then we end up with a priority list of {A, C, D, B}. In a scheme like this 4! (n! with n requestors) orderings are possible, which makes the complexity (hence the speed and hardware cost) of the arbitration much higher.

Going back to RRA, suppose we represent the priority list with a pointer to the head of the list (see headPtr in Fig. 3) and represent A with a pointer 3, B, C, D with 2, 1, 0, respectively. Then, initially (Fig. 3a) the list is represented with a pointer of 3 (pointing to A). After the first cycle of arbitration, B becomes tail of the list and C becomes head (as shown in Fig. 3b), and the list is represented by 1 (pointing to C). Given the index to the list's head as 1, we can easily construct the whole list as {1, 0, 3, 2} and hence {C, D, A, B}. The list is monotonically decreasing (in modulo *n*) starting with the requestor pointed by headPtr.

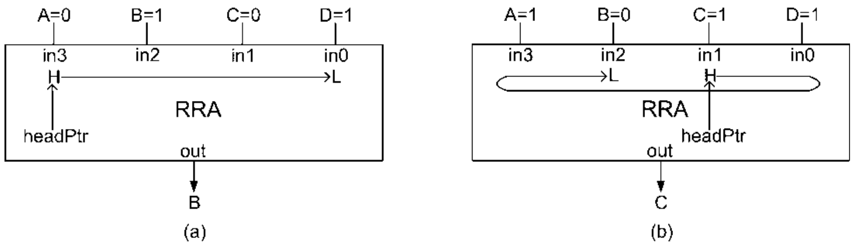

Fig. 3. (a) One cycle of an RRA. (b) Next cycle of the RRA (headPtr has been updated).

Note that the precedence of the requestors can go down from right to left instead of left to right (i.e., the direction of the arrow in Fig. 3). The requestor (i.e., input) indices may go up from left to right or right to left. Hence, there are 4 different versions of an RRA, and they are all equivalent in functionality as well as implementation timing/area. Internally, the RRA may keep a tail pointer instead of headPtr. That, however, may or may not impact the implementation timing/area depending on whether the pointer is coded one-hot or regular binary.

1.2. Background

There are arbiters in the literature claimed to be round-robin although they do not follow the typical (i.e., widely accepted) definition given in Section 1.1 (call it RRA_typical). The work of Shin et al. [3] is an example of such work and can be found in its entirety in [4]. This arbiter does a separate round-robin within little 4-input arbiters in its larger hierarchical arbitration tree. Chao et al. [5] have an earlier paper based on pretty much the same concept of doing ping-pong of local pointers in subblocks and hence is called Ping-Pong Arbiter. Zheng and Yang [6] state that these arbiters are not fair under non-uniformly distributed requests. There is also a very recent work by Jou et al. [7] (extended in [8]) that uses a very similar arbitration method. These arbiters are outside the scope of this paper's work.

This report deals with logic gate level (as opposed to transistor level) architectures for RRA_typical that have superior performance in timing (i.e., critical path). We will include their comparison in timing – using a logic synthesis based flow (with a library of static CMOS standard-cells).

Gupta and McKeown's work in [9,10] is the first work to give a detailed logic implementation of an RRA together with area and speed results from logic synthesis. Their arbiter design is called PPE – short for Programmable Priority Encoder. PPE specifically deals with the implementation of RRA_typical, not some arbitrary loosely round-robin scheme such as [3,5,7,11]. PPE work spurred interest in this specific problem, leading to [6,12,13,14]. We will look at PPE and these four works in detail in the following sections. The RRA designs proposed by these papers are 100% pin and policy compatible with PPE and are all at logic level. We wrote generators for PPE and all of these designs. We will summarize synthesis results we obtained from these generated designs in the Results section.

From the perspective of RRA_typical, previous work on Fixed Priority Encoders (FPEs) is also important. An FPE is sometimes simply called PE (Priority Encoder) and can be thought of as if it has a fixed headPtr (see Fig. 1 and Fig. 2). As Gupta calls it, an RRA is a Programmable PE (PPE). Moreover, Gupta's PPE has FPEs embedded in them, and as a result, high-performance FPE design can be critical when building an RRA. Most of the literature on FPEs that is worth to mention is at transistor level, and some even use dynamic CMOS as in [15]. However, their logic architectures can be used in a logic synthesis based standard-cell design flow.

Delgado-Frias and Nyathi [15] extend the carry lookahead technique found in fast adders to FPE design and do a lookahead of 4. They also propose doing a 2nd-level lookahead for further speed-up. Huang et al. [16] however present a really novel method of 3rd-level lookahead on top of [15]. Their 3rd-level lookahead utilizes shortcuts in the carry chain found by folding the layout in the middle each time the number units becomes a power of 2. Although this is quite a novel approach, it has serious handicaps in a logic synthesis flow. The shortcuts in this design speed up the design because with these shortcuts the topological longest path is no more the true critical path. That is, the shortcuts introduce many *false paths* in the design. In a synthesis based flow, these false paths need to be input to the synthesis tool since the tool constantly times the critical path to decide how to do technology mapping, cell sizing, and buffering so that it can meet the target timing. In the presence of too many false paths logic synthesis tools blow up. That is, they take an enormous amount of time to complete synthesis, and also the long synthesis times as well as broken paths limit the extent they can minimize logic area and timing. Abdel-hafeez and Harb [17] do a two-level lookahead (amounting to a lookahead of 8 positions) similar to [15] and implement two different FPEs in one logic network, namely, one FPE that scans from left to right and another that scans from right to left.

Preußer et al. [18] go further than [15,16] and observe that *parallel prefix networks* (PPNs), which constitute the state-of-the-art in fast adder circuits, can be used in arbiter design and present some results on FPGAs. Dimitrakopoulos et al. [13] make the same observation later and come up with a competitive design (see Section 5) and demonstrate it with results from a standard-cell based synthesis flow.

In the sections to follow, we will have a unique review of 5 competitive (i.e., fast) designs so far proposed in the literature that comply with RRA_typical. This is done with original diagrams and formulations instead of a dry summary of respective papers. These designs are pin and policy compatible with each other, and their improved versions (by us) are benchmarked in Section 7 mainly based on speed. Every section below belongs to a separate design, with the title indicating the particular RRA design's name, the institution or country the design work took place, first year it was published, and related references. The 5 designs are presented in chronological order, starting with the earliest.

2. PPE of Stanford, 1998 [9,10]

PPE[1] (Fig. 4) was proposed as a fast RRA implementation used within the ESLIP algorithm. ESLIP is based on *i*SLIP [19] and is the scheduling algorithm used in Stanford University's Tiny Tera (a 1 Tbps network switch) [20] and Cisco Systems, Inc. 12000 GSR router. Although PPE is part of a work that tries to solve a rather specific arbitration problem, it has the same I/O behavior as RRA_typical. The bigger arbitration problem in Tiny Tera and many other many-port and high-throughput core network routers/switches is as follows.

2.1. The Context of PPE

There is an n2n crossbar in the switch, and at any time, multiple input ports (IP) may request for the same output port (OP), i.e., ManyToOne problem, while a single IP may request for multiple OPs, i.e., OneToMany problem. ManyToOne problem can be solved with an arbiter (e.g., RRA_typical) at every OP. This arbitration may result in selection of the same IP for two or more (2+) OP. This is because an IP may have 2+ concurrent request, each one to a different OP. High-throughput switches buffer packets

[1] Note that what [9,10] actually call PPE is only the part of Fig. 4 in the dotted box.

pending at IPs in queues (Qs) and there is not one but *n* (number of OPs) Qs at each IP. The packet at the head of each Q (one Q per IP-OP pair) constitutes a concurrent request, and when 2+ Qs are non-empty at an IP, then there is a OneToMany problem. Separate Qs per IP eliminate *Head of Line Blocking* (HOL). (HOL: *Assume a single Q at each IP. Then if packet X at IP1 is asking for OP1 while the next packet Y at the same IP asks for OP2, there would be HOL. When X's turn comes, hence becomes head of line, it is possible that it is not arbitrated to OP1 because some other packet at some other IP wins the arbitration for OP1. However, if there is no packet in the switch arbitrating for OP2, Y will be unnecessarily delayed.*) Although separate Qs solve this problem, having physically separate Qs (i.e., separate memory blocks) is prohibitively expensive because then a Q cannot use the space not claimed by others. Also, breaking one memory block into *n* little pieces yields memory blocks with an area of significantly larger than 1/*n* (due to address decoding overhead). Also, the amount of wiring goes up to n^2 from *n*. That is why these multiple Qs at an IP are kept in a single memory block. They are called Virtual Output Queues (VOQ). They are virtual because they reside in the same memory and are called output Qs because there is one for each OP. However, since they are all in a single physical block, we can access only one of them in one arbitration slot and should hence arbitrate one out of them. The result is the OneToMany problem, and we have to pick one request per IP. Gao et al. [21] call this *Conflicting* Virtual Channel Router (VCR). In short, we have to pick a OneToOne set, out of all requests (i.e., IP-OP pairs), and we must come up with the largest OneToOne set as we are trying to maximize crossbar throughput. Mathematically, this problem is a *maximum-size bipartite graph matching* problem.

*i*SLIP solves the above matching problem by two sets of RRAs, one parallel set at the IPs and one parallel set at the OPs of the crossbar. However, it uses these RRAs multiple times because it is an iterative algorithm. It not only completes one (multiple-iteration) arbitration in multiple cycles but also does one RRA in multiple cycles. However, in their problem they still need to minimize the latency of an RRA just like RRA_typical. Note that if we are willing to pay for the area hit and put every input Q for every (IP, OP) pair in a separate memory block, then we only need a one-time arbitration at each OP. Gao et al. [21] call this *Non-conflicting* VCR Router (VCR). Either way (conflicting or non-conflicting VCR) a crossbar needs RRAs.

2.2. The Architecture of PPE

The top-level architecture of PPE (i.e., *i*SLIP's RRA building block) is given Fig. 4. PPE not only plays a role in the solution of the above arbitration problem but also solves RRA_typical. However, the timing results reported in [10] (initially presented in [9]) assume the path from Gnt (i.e., grant) output back to the flops (which store the pointer) is a false path because *i*SLIP does the pointer update of the RRAs only after all iterations are complete. We should note (as mentioned in Section 1.2) that in a synthesis based flow false paths are undesired. Also, if those false paths can become true paths with different register values, they make ATPG (Automatic Test Pattern Generation) very difficult.

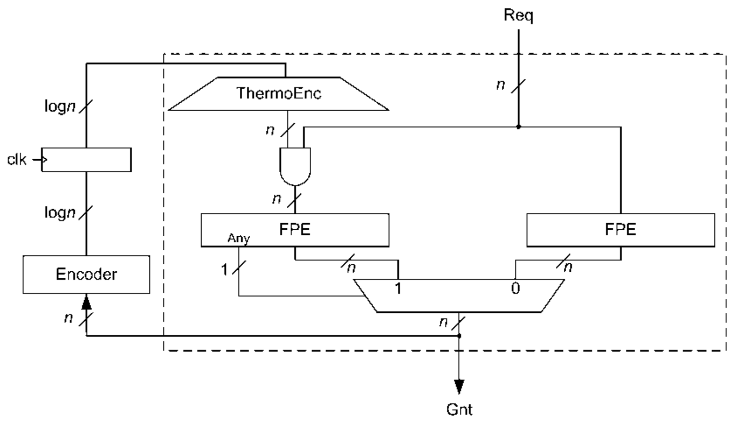

Fig. 4. PPE's architectural diagram.

The key innovation in PPE is that it breaks down the RRA_typical problem into 2 FPE problems by breaking up the wrap-around path in Fig. 3b from in0 back to in3. The arbitration problem in Fig. 3b can be looked at as follows. We have to first arbitrate in1 to in0 (*arbProb1* in Fig. 5). If there are no requests in that range, then arbitrate in3 to in2 (*arbProb2* in Fig. 5). Since in3 to in2 is looked at when there is no request between in1 to in0, the result would be the same if arbProb2 covers in3 to in0 (i.e., the full range). Hence, arbProb2 becomes nothing but a regular/full FPE problem. On the other hand, arbProb1 can be turned into an FPE when its in3 and in2 are masked out (i.e., zeroed). Hence, RRA_typical is equivalent to solving two FPEs as shown both in Fig. 5 (and also in Fig. 4).

In Fig. 4, the *n*-bit Gnt vector is one-hot. The position of the only 1 (if any) becomes the L pointer in the next arbitration cycle. The FPE on the left in Fig. 4 solves arbProb1 and hence needs the input request bits from the very left to the position where L points zeroed out. ThermoEnc and the AND gate in Fig. 4 do exactly that. The input to ThermoEnc is an *n*-bit vector that has a 1 at position L, and its output is all 0s from position *n*–1 to L and all 1s from position L+1 to 0. When this thermo-coded vector is ANDed with the request inputs, only the high priority half of the requests survive (i.e., can go through with non-zero values). The only other thing we have to do is put a MUX to use the result of the FPE for arbProb1 and if that has no 1s (ANY=0) then supply the output of arbProb2 FPE as the final Gnt output of the RRA.

$$\begin{array}{lllll} \text{RRA(xxxxxxxx)} = & \text{FPE(000xxxxx)} & = & \text{FPE(000xxxxx)} & \longleftarrow \text{arbProb1} \\ & \text{else FPE(xxx00000)} & & \text{else FPE(xxxxxxxx)} & \longleftarrow \text{arbProb2} \end{array}$$

Fig. 5. A PPE equals 2 FPE problems. **arbprob1:** over the High Priority Half (HPH). **arbprob2:** over the full set of requests.

Gupta [10] also discovered that RRA has a similar data flow to an "adder". However, they applied carry propagation techniques only to their earlier ill-defined architecture, which has a combinational loop. A combinational loop (even when it is a false loop) is unacceptable in real practice because it cannot be timed by the synthesis software and it has pitfalls in testing. Gupta uses a traditional carry-lookahead (CLA) approach and somehow only an n/4-bit CLA although n/2-bit CLA is also possible. Gupta does not mention using CLA in the FPEs of their fastest architecture (PPE in Fig. 4). FPEs can greatly benefit from smart carry propagation but we do not know how they are coded in PPE.

Another contribution of [10] is that it also talks about and benchmarks two straight-forward architectures for RRA, which are commonly used in the industry although they are not very efficient in timing and area. One is the *exhaustive* architecture (truth table approach) and the other is the more common rotate-FPE-rotateBack approach. They are both discussed in Weber's SNUG paper [22]. When the rotators are implemented as log*n*-level barrel shifters and the FPE is also done in log*n* levels, then this architecture is competitive in timing (3log*n* levels). However, the five architectures detailed here are between the orders of log*n* to 2log*n* in timing.

2.3. Our Improved/Modified PPE

As indicated earlier, we wrote Verilog RTL generators for all five designs including PPE and synthesized them through Synopsys Design Compiler. Additionally, we have implemented their logic in the best possible way when the internal structure of a subblock is not given – as in the case of PPE's FPEs. We have also picked the better implementation when there is ambiguity as to how a subblock needs to be implemented. In the case of PPE's FPEs, Gupta does not discuss FPE implementation. We have implemented them with the state-of-the-art PPNs [23,24,25,26], which do a carry lookahead over the full range of input bits. When our results for PPE are compared to Gupta's in [10], we have a savings of anywhere from 28 to 44%. However, up to 30% of this may be due to our using .18µm libraries as opposed to their using .25µm.

2.4. *PPE_C and PPE_N of Beihang University in China, 2006* [21]

PPE_C and PPE_N were inspired by the PPE architecture. Instead of making this work a separate section at the top level, we have decided to make it a subsection of PPE as it is not a fast architecture. PPE_C is only significant from an area reduction point of view.

The idea here is that we need only one FPE if we first check whether the requests in HPH contain any 1s. If they do, we mask the requests and only feed the requests of HPH (over a MUX) to the only FPE. If not, we feed the complete request vector to the FPE. It turns out the same idea was earlier proposed as part of a system level work done with FPGAs to build a 10 Gbps switch [27].

PPE_C simply offers area reduction but is slower than PPE due to an extra OR-tree introduced on the critical path. PPE_N, on the other hand, does not offer much area reduction over PPE but it is faster as it eliminates the Encoder and ThermoEnc by introducing a second FPE with thermo output (ThermoFPE). Gao et al., however, missed the point that they actually do not need the regular FPE as it can be constructed by attaching an edge detector to the output of ThermoFPE. Although ThermoFPE has a smaller and faster implementation than FPE (using a PPN), it is not clear if Gao et al. caught that point. Even then, PPE_N would not be too competitive in terms of timing due to the OR-tree but offer good area savings as we have shown in [28]. Since our focus in this paper is fast RRA architectures, we will not include PPE_C and PPE_N in our benchmarking.

3. PRRA and IPRRA of UT Dallas, 2002 [6,29]

PRRA is short for Programmable RRA, while IPRRA stands for Improved PRRA. This is the first work after PPE in the literature and solves the exact same problem as PPE (hence RRA_typical) – first published in [29] and then in [6] by Zheng and Yang. PRRA and especially IPRRA are quite competitive architectures in terms of both timing and area. IPRRA was proposed as an improvement over PRRA. However, based on our synthesis results, PRRA may also sometimes yield superior results.

3.1. *The Architecture of PRRA*

PRRA has a purely “recursive” structure. As can be seen in Fig. 6, the circuit is nothing but a binary tree in topology, or in other words, there are two identical subcircuits: left circuit (left subtree) and right circuit (right subtree) – each one for their half of the request inputs and grant outputs. Although this is a combinational circuit, it looks as if there are loops because neighboring subblocks (or nodes as depicted in Fig. 6) have signals pointing in both directions. However, this is a completely feed-forward circuit. Signals first propagate up the tree then propagate down. These *up-trace* (see *ut* nodes in Fig. 6b) and *down-trace* (see *dt* nodes in Fig. 6b) become explicit when PRRA is drawn as a “bowtie” (see Fig. 6b). In the up-trace at every level, a summary for each subtree is derived from the summaries of its left subtree and right subtree. The summary-signals characterize everything into 4 cases: The subtree

- **0:** has neither the headPtr (hP) nor an active request (req) – **hP=no, reqs=no**
- **1:** does not host the hP but has active req(s) – **hP=no, reqs=yes**
- **2:** hosts the hP but does not have any active req in the *High Priority Half* of reqs (HPHreqs) – **hP=yes, HPHreqs=no**
- **3:** hosts the hP and has active req(s) in HPHreqs – **hP=yes, HPHreqs=yes**

The signals first ascend in the binary tree (in Fig. 6a), summarizing requests and where the headPtr is. When we reach the root node, we start making arbitration decisions as we descend down the binary tree. Each node in the lower binary tree in Fig. 6b is a decision node and works when it is enabled (i.e., arbitrated) by its parent node. Then, it decides whether to enable its left subtree or right subtree depending on the summary information coming from the upper binary tree (i.e., up-trace) – hence the dotted lines in

Fig. 6b.

We had earlier come up with an architecture like this ourselves, called it “Bowtie Architecture”, and drew it just like Fig. 6b. However, we were not able to make it as efficient as PRRA because we were describing the summary information in 5 different cases as opposed to the 4 cases of PRRA. In our architecture, we were making a distinction between the cases where:

(i) a subtree has the headPtr and no active requests in the HPHreqs and

(ii) it has the headPtr and has an active request (i.e., at least one request is 1) in the Lower Priority Half (LPH).

PRRA does not discriminate between those two cases. It picks a subtree with the headPtr whether it has a request in LPH or no requests at all over a subtree that has no requests and no headPtr. Arbitrating a subtree with no requests in it is not a problem as PRRA ANDs the grant lines out of the lower binary tree of Fig. 6b (i.e., internal grants) with the request lines to obtain the “final grant outputs”. When we came up with 5 different cases in our version of this architecture, it resulted in 3 wires for a summary-signal and hence worse timing and worse area.

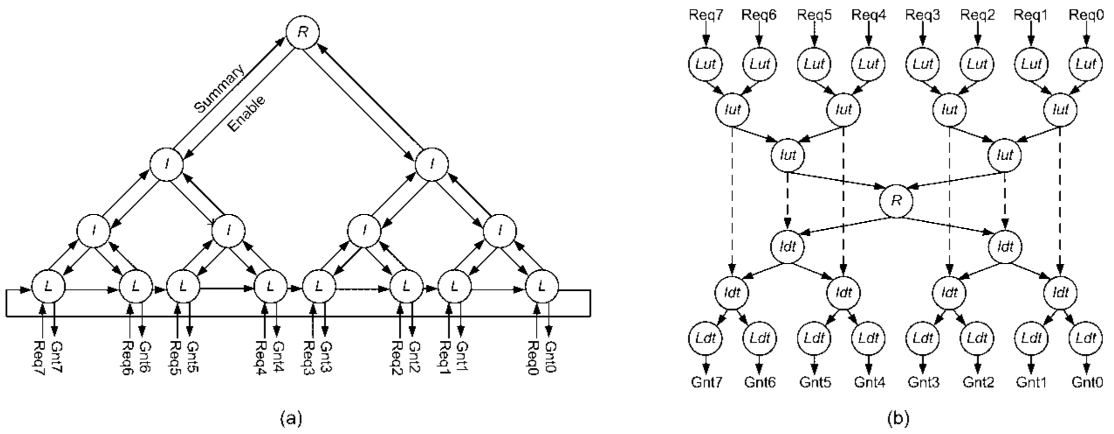
R
Summary
Enable
I
L
Req7 Gnt7 Req6 Gnt6 Req5 Gnt5 Req4 Gnt4 Req3 Gnt3 Req2 Gnt2 Req1 Gnt1 Req0 Gnt0
(a)
Req7 Req6 Req5 Req4 Req3 Req2 Req1 Req0
Lut
Iut
R
Idt
Ldt
Gnt7 Gnt6 Gnt5 Gnt4 Gnt3 Gnt2 Gnt1 Gnt0
(b)

Fig. 6. (a) PRRA. (b) PRRA shown in the form of a bowtie.

PRRA has 3 types of nodes: R (Root), I (Intermediate), L (Leaf). The R node is actually an I node with the parent side connections removed. L nodes, however, are the nodes where the headPtr is kept and hence are the only nodes where there are flops. L nodes are also the place where the internal grants are ANDed with the requests (to obtain the final grant) and the grant is shifted to obtain the new headPtr (so that the granted requesting agent becomes the tail).

3.2. The architecture of IPRRA and other variants

IPRRA improves the critical path through the down-trace, namely, the lower cone (LC) in Fig. 6b. It does this as follows. The function of the root node (R), which is at the juncture of the two cones, is to either enable the left subtree of LC or the right subtree. In PRRA, this is done by enabling/disabling the I nodes at the root of the two down-trace subtrees, which, in turn, do the same for their own subtrees. This creates a long path from the root node down the LC, which goes through every level. Therefore, the longest path traverses all levels through the upper cone (UC) as well as LC. In IPRRA, the subtrees operate without an enable signal and generate their own grant outputs, and the root node gates those outputs (ANDs them with its enable signal) as shown in Fig. 7a. As a result, the critical path through the root node has k ($=\log_2 n$, where n is the number of requestors) nodes on it in UC plus one node in LC. An I node in UC at level i has k–i nodes on its critical path and i+1 nodes in LC. Hence, in IPRRA there are many critical paths all with k+1 nodes, whereas PRRA has one critical path, which goes through the root node and 2k nodes in total. When the different contents of the I nodes in UC and LC are taken into account, the critical path of the IPRRA may seem to be $3\log_2 n$, whereas PRRA is $4\log_2 n$.

However, these formulas for critical path length are assuming “fanout” does not contribute to delay and a gate/node has the same delay in the circuit no matter how many and what gates it drives. The delay of a gate in reality depends on its load. Therefore, fanout contributes to delay even when wire delay (i.e., RC) is neglected. When fanout is taken into account, the speedup is not as much as it may seem. We have used Synopsys Design Compiler (DC) for logic synthesis, and it properly times circuits considering fanout and inserts a buffer tree so that load capacitance is within the characterization range of the cell library and the gate extrinsic delays are optimized.

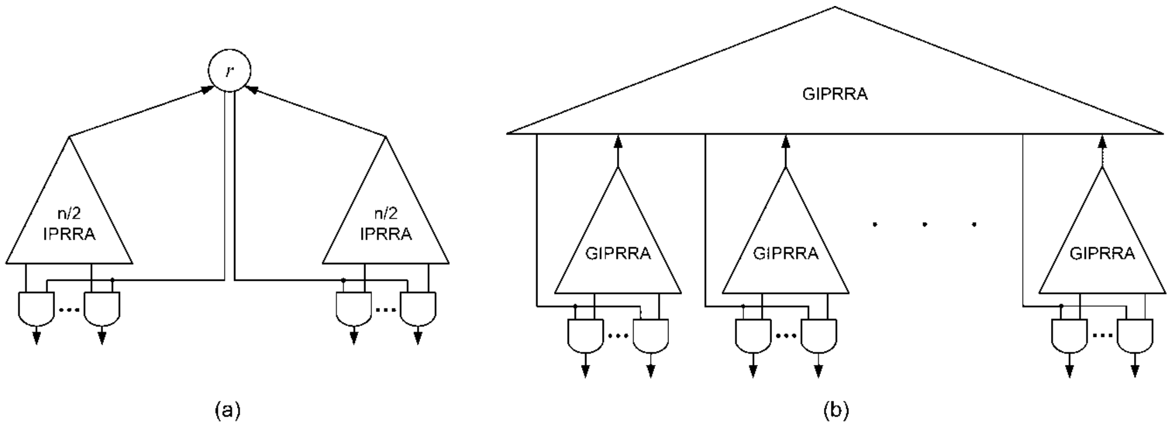

Fig. 7. (a) IPRRA. (b) GIPRRA.

For very large *n* and hence very large fanouts, Zheng proposes Grouped IPRRA (GIPRRA), which is a recursive architecture (shown in Fig. 7b) with an IPRRA architecture in the top level. Although [6] uses IPRRA for the leaf cones, we have realized that they can be GIPRRAs or even yet any RRA_typical that has the two extra up-trace summary-signals. On the other hand, the top level architecture that ties the little cones (i.e., little RRAs) can be PRRA as well instead of the IPRRA architecture shown in Fig. 7b. Although GIPRRA is a promising and flexible architecture, it requires a wise choice for the size of the little cones. Since Zheng does not report any implementation (i.e., synthesis) results for GIPRRA, we did not implement it.

3.3. Our Improved/Modified PRRA and IPRRA

It seems as if Zheng assumed PRRA/IPRRA would always be used in places that have heavy traffic. When traffic loads are low, there would be cycles when all request queues are empty, i.e., all requests inputs to the RRA are zero. In that case, PRRA/IPRRA generates an all-zero grant output and the headPtr also becomes all zeros, hence degrading PRRA/IPRRA into an FPE. This breaks the fairness of their RRA and favors the leftmost requestor by offering it lower latencies on the average. In order to fix this, we do not update the headPtr register when there is no request in our PRRA/IPRRA implementations as well as the other architectures we have implemented. As for the internals of a PRRA I node, the circuit level and the Boolean expression level implementations in [6] are different and the Boolean one has a significantly better timing. We have implemented the one with better timing. Our timing results for PRRA and IPRRA are significantly better than what was obtained in [6]. Our results for IPRRA (the better of the two) are better by between 35-60% (i.e., more than 2x speedup) compared to the IPRRA results in [6] although we have used the same cell libraries (TSMC .18µm) and the same synthesis tool – Synopsys DC – (only a newer version). Our improvements for PRRA have higher percentages.

4. BTS-RRA of Cisco, 2004 [12]

The name BTS-RRA stands for Binary Tree Search RRA and has been proposed by a team from the industry. This RRA_typical has a recursive mechanism just like a binary tree (see Fig. 8b). However, this recursive behavior is not obvious from the circuit topology given in [12]. On the other hand, our depiction of BTS-RRA in Fig. 8 makes the recursive topology obvious. BTS-RRA has a top-level architecture (see Fig. 8a) that consists of a mask unit (AND gate), a recursive core module (coreBTS – see Fig. 8b), and an edge detector (ED). The *n* to *n* core RRA can be implemented with two *n*/2 to *n*/2 versions of the same module (Left RRA and Right RRA) and a logic that massages their *n*/2-bit outputs to form an *n*-bit output (i.e., JoinLogic).

JoinLogic behaves as follows (in Verilog syntax):

```
assign omReq = omReq_right || omReq_left;
assign oReq = oReq_right || oReq_left;
assign gatingSignal = omReq ? omReq_left : oReg_left;
assign g[n–1:n/2] = g_left & {n/2{gatingSignal}}; // parallel bitwise ANDs, note that n/2-bit g_left is ANDed with n/2 copies of gatingSignal
assign g[n/2–1:0] = g_left | {n/2{gatingSignal}}; // parallel bitwise ORs
```

Let us explain how JoinLogic works. Assume the priority of inputs go down from left to right. Then, we need to either forward the output bits of the Left RRA to the output as is and make the output bits of Right RRA all 1s, or we need to zero out the outputs of the Left RRA and forward the outputs of the Right RRA as is. In order for this to happen, we need to gate the outputs of the Left RRA with ANDs (see the Verilog line with *parallel bitwise ANDs* above) and the outputs of the Right RRA with ORs (see *parallel bitwise ORs*). The parallel ANDs and ORs are driven by a logic (i.e., the first three lines of Verilog) that depends on the ANY flags that come from the two size *n*/2 RRAs. Each RRA (Left and Right) has one ANY output that signals whether any of its inputs is 1 (i.e., oReq for OR of requests) and another ANY that signals if any of its HPHreqs is 1 (i.e., omReq for OR of masked requests). As shown in Fig. 9, the placement of parallel AND gates have a Ladner-Fischer PPN topology [24] so do the OR gates.

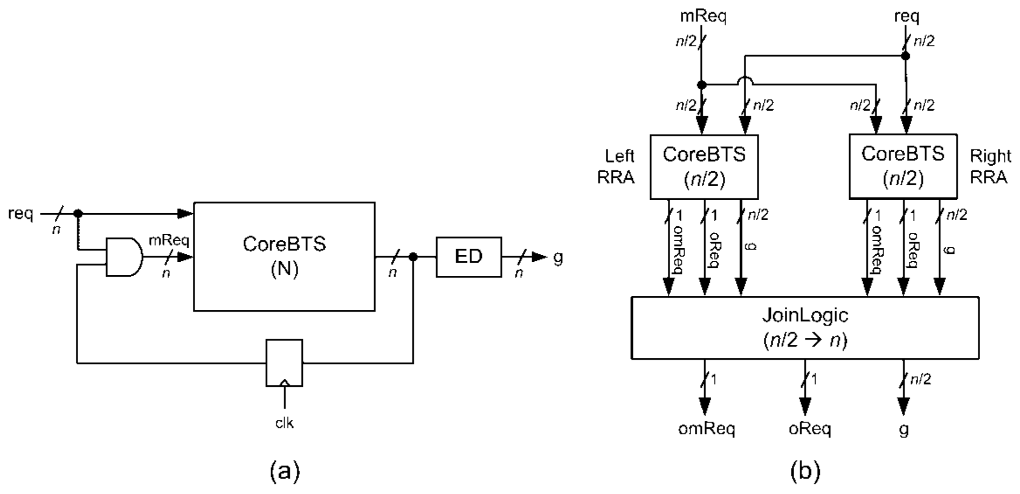

Fig. 8. (a) Top-level architecture of BTS-RRA. (b) An *n*-bit CoreBTS in terms of two *n*/2-bit CoreBTS modules.

Although a complete proof of why this always works is not included here (and neither in [12]), we will state that it is possible to prove it by enumerating it for 6 possible scenarios, which cover all possible cases. These scenarios can be coded as: {xH1, x}, {1H0, 0}, {xH0, 1}, and 3 more where the pairs in the curly braces are flipped. {xH1, x} means Left RRA has the headPtr (H), there is at least one active request to the right of H, requests to the left of H can be anything (x), and requests of the Right RRA can also be anything. {1H0, 0} means Left RRA has H, there is at least one active request to the left of H, and all other requests are 0. {xH0, 1} means Left RRA has H, there is at least one active request in Right RRA, and all other requests are 0. Hence, 0 means all requests are 0 in the corresponding range, and 1 means at least request equals 1 in the corresponding range.

CoreBTS produces a thermo-coded output, which is directly used as a headPtr (without the need of a ThermoEnc block like PPE). This internal *n*-bit pointer and grant output is sent to an "edge detector" (ED in Fig. 8b) to produce the final one-hot grant vector. BTS-RRA's claim was to come up with a very area efficient architecture because it uses a single encoder, not 2 FPEs like PPE. Although PPE uses 2 FPEs, PPE can also be very efficient in area as an FPE has much less complex nodes than BTS-RRA. Savin et al. [12] only offers timing and area estimates based on some formulas. Our work here has the contribution of synthesizing and offering real figures of merit for BTS-RRA.

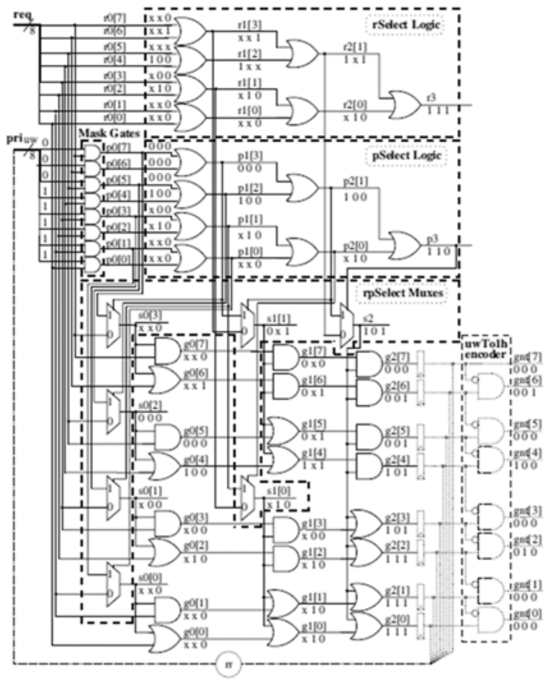

Fig. 9. The architecture of BTS-RRA as it is shown in [12].

5. Proposed-II (PII) of Greece, 2008 [13,30]

PII (see Fig. 10) turns the RRA problem into a circular PE problem and formulates it such that it is 100% identical to the carry lookahead problem. The authors, as a result, observe PPNs can be used to solve the problem. However, they end up proposing a solution based on only Kogge-Stone PPN [23]. In their first architecture, one can actually use any PPN. On the other hand, in PII (their second and better architecture) only Kogge-Stone seems to fit the bill out of the well-known PPNs as the PII has a very direct circular nature. Note that a non-cyclic Kogge-Stone PPN is pretty much the same as a Barrel Shifter architecture, while a cyclic Kogge-Stone is identical to a Barrel Rotator.

The dark nodes in Fig. 10 do the same as what nodes in a PPN do. The light color nodes are a simplified version of the dark nodes. The priorities go down as we go from right to left. The *P* bits show the pointer bits (one-hot), *R* request, and *G* grant. *AG* stands for Any-Grant (i.e., OR of all *G*'s). *G*'s shift to the left and become the *P*'s in the next cycle (not shown in Fig. 10). In other words, G0 is hardwired to P_1, G_1 to P_2, G_2 to P_3, and so on. That is because the *P* that is 1 is the highest priority input, while we have to make the *G* that is 1 the lowest priority.

Note that PII reduces the two FPEs in PPE (of Stanford) to a single PE. However, since the logic in the nodes are significantly more complicated, it turns out to be the design with the largest area – usually yielding twice the area of other designs for the same timing.

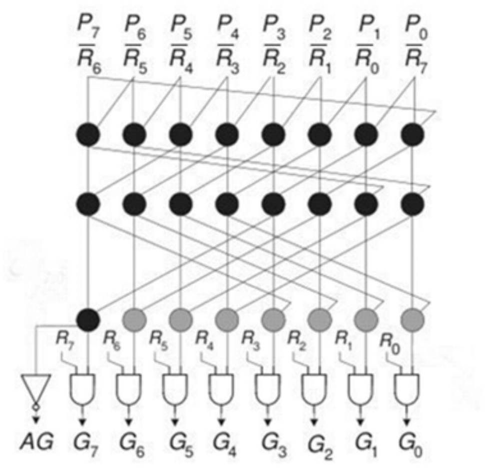
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
$\overline{R_6}$ $\overline{R_5}$ $\overline{R_4}$ $\overline{R_3}$ $\overline{R_2}$ $\overline{R_1}$ $\overline{R_0}$ $\overline{R_7}$
R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0
AG G_7 G_6 G_5 G_4 G_3 G_2 G_1 G_0

Fig. 10. An 8-bit PII as depicted in [13].

6. Jou & Lee (JL) of Taiwan, 2008 [14,31]

JL takes a very direct and low-level approach to the RRA_typical problem. They express each grant signal of the RRA in SOP form (i.e., Sum Of Products) and claim they have the Boolean expressions with the fewest number of minterms out of all possible SOP expressions. We below express the Boolean equations of JL in a formal and concise notation, while they are expressed in an ad hoc notation in [14,31]. In the equation below, g_i is the ith grant output, r_i is the ith request input, t_i is the ith pointer bit, and n is the total number of inputs as well as outputs and also pointer bits. The priorities go down as the index goes up.

$$g_i = r_i\left(\sum_{j=0}^{i-1} t_j \prod_{k=j}^{i-1} r_k' + t_i r_i + \prod_{k=0}^{i-1} r_k' \prod_{k=i+1}^{n-1} r_k' + \left(\prod_{k=0}^{i-1} r_k'\right)\left(\sum_{j=i+2}^{n-1} t_j \prod_{k=j}^{i-1} r_k'\right)\right) \quad \forall i = 0, n-1 \tag{1}$$

Although Eq. 1 above looks complicated, it is actually a straight-forward way to express RRA in a single Boolean expression. Although JL uses a simple-minded and brute-force approach to RRA, its synthesis yields competitive results in timing (but not in area). After n=128, however, its synthesis takes forever and never completes as the logic equation becomes too large.

7. Results

The table below summarizes our synthesis results. The timing figures are in the table are in nanoseconds (ns) and represent the period of the fastest clock possible assuming all inputs come from flops and all outputs go to flops. The numbers highlighted are the best (i.e., smallest) numbers for the respective number of ports. JL blows up in runtime and never completes for after 128 ports, hence no results for 256 ports and more for JL.

We used a 2009 version of Synopsys DC. In each synthesis run, we actually did 4 synthesis runs by cleverly adjusting the target clock period through a binary search. At every point, we have a lower bound and upper bound, and the new target clock period is set to the average of the two bounds. An unmet clock period target that is larger than the previous lower bound becomes the new lower bound. The target clock period minus the slack becomes an achievable clock period and bcomes the best clock period (and hence the upper bound) if it is lower than the previous best clock period. Note that we did not use wire-load.

The first row of percentages in the table shows the overall performance of the architectures for 8 to 128 ports. The second line shows their overall performance for all of them (8 to 512 ports). The percentages were calculated as follows. For each row, the smallest number was scaled to 1. That is, all numbers in a row were divided by the smallest number in that row. For each architecture, the scaled numbers were then averaged. The 39% for PPE for example means PPE's result was on the average 1.39 times the best result, hence within 39% of the optimum.

Table 1. Timing results (in ns) from logic synthesis.

#ports	**PPE**	**(I)PRRA**	**BTS-RRA**	**PII**	**JL**
8	1.450	1.225	1.290	1.295	**1.140**
16	1.740	1.460	1.430	1.508	**1.420**
32	2.275	1.640	1.730	1.708	**1.625**
64	2.580	**1.820**	1.856	1.930	1.895
128	2.980	**2.098**	2.125	2.145	2.130
	35%	***2%***	***5%***	***7%***	***1%***
256	3.385	2.330	**2.320**	2.325	
512	3.865	2.670	**2.575**	2.632	
	39%	***2%***	***3%***	***5%***	

(I)PRRA means we reported the best of PRRA and IPRRA. Although IPRRA is supposed to be better than PRRA according to [6], PRRA is pretty close to IPRRA and surpasses IPRRA in one case. We implemented PPE with 4 different PPNs, namely, Kogge-Stone [23], Ladner-Fischer (LF) [24], Brent-Kung [25], Han-Carlson (HC) [26] and entered the best result into the table. Out of the 7 cases, LF is best in 4 cases, HC is best in 2 cases, and it is a tie between those two in 1 case.

Although PPE triggered all other works, it is the worst performer by far. Before JL blows up, it is the best performer with a performance of within 1% of the optimum. Then (I)PRRA is the best. In overall performance (for all 7 cases), we cannot evaluate JL. Then, (I)PRRA is the best performer with within 2% of the optimum. Then, BTS-RRA is the runner-up with a performance of 3% within the optimum. JL is the top performer up to 64 ports, then (I)PRRA becomes the top performer for 64 and 128 ports, and for the highest number of ports (256 and 512) BTS-RRA is the winner.

The optimization criterion in this work is timing. However, area is still critical when it gets too large. In the area dimension, PPE, (I)PRRA, and BTS-RRA are pretty close and they are ±13% around their mean, while PII is between 1.5 to 3 times the size of their mean, and JL is 1.5 to 2 times their mean. The JL papers [14,31] report area results and admit that their area is significantly larger than their competition. However, PII papers [13,30] do not report on area.

PPE papers [9,10] report synthesis results for 8, 16, 32, and 64 ports. (I)PRRA papers [6,29] report synthesis results for 4, 8, 16, 32, 64, 128, and 256 ports. PII papers [13,30] report synthesis results only for 16 and 32 ports. JL papers [14,31] report synthesis results for 4, 8, 16, 32, and 64 ports. BTS-RRA paper [12], on the other hand, reports only theoretical estimations based on some formulas – no real synthesis results. We are the first in the literature to report results 512 ports. We did not report any results for 4 ports as it is such a small piece of logic and usually does not constitute a bottleneck in a bigger design.

8. Conclusion

In this work, we had a very detailed review of 5 very fast architectures in the literature that solve the RRA_typical problem. We also wrote generators for them and synthesized them under the same conditions: (i) same standard-cell library, (ii) same synthesis tool and version, and (iii) same synthesis. For any meaningful performance comparison to be made, equality in these three dimensions is critical. It seems like only (I)PRRA work [6,29] did a fair job in this department. Note that we were fair to all designs when we implemented them. As we mentioned before, this can be told from our improvement of between 35-60% (i.e., more than 2x speedup) of IPRRA. Let us wrap up with a summary of the results, JL is the best for between 8-32 ports (if we want to pay the sever area penalty up to 100%). (I)PRRA is best for 64 and 128 ports. BTS-RRA is best for large number of ports (256 and 512).

References

[1] J. Reed, N. Manjikian, A dual round-robin arbiter for split-transaction buses in system-on-chip implementations, in: Proc. Canadian Conf. Electrical and Computer Engineering, 2004, pp. 835–840 vol. 2.

[2] A. Zitouni, R. Tourki, Arbiter synthesis approach for SoC multi-processor systems, Computers and Electrical Engineering 34 (2008) 63–77.

[3] E.S. Shin, V.J. Mooney III, G.F. Riley, Round-robin arbiter design and generation, in: Proc. Int. Symp. System Synthesis (ISSS), 2002, pp. 243–248.

[4] E.S. Shin, Automated Generation of Round-Robin Arbitration and Crossbar Switch Logic, PhD Thesis, Georgia Inst. of Technology, 2003.

[5] Chao, H.J., Lam, C.H. & Guo, X., 1999. A fast arbitration scheme for terabit packet switches, in: Proc. Global Telecommunications Conf. (GLOBECOM), 1999, pp. 1236–1243.

[6] S.Q. Zheng, M. Yang, Algorithm-hardware codesign of fast parallel round-robin arbiters, IEEE Trans. Parallel and Distributed Systems 18 (1) (2007) 84–95.

[7] J.-M. Jou, S.-S. Wu, Y.-L. Lee, C. Chou, Y.-L. Jeang, New model-driven design and generation of multi-facet arbiters - Part I: From the design model to the architecture model, in: Proc. Design Automation Conf. (DAC), 2010, pp. 258–261.

[8] J.-M. Jou, Y.-L. Lee, S.-S. Wu, Efficient design and generation of a multi-facet arbiter, in: Proc. Symp. Application Specific Processors (SASP), 2010, pp. 111–114.

[9] P. Gupta, N. McKeown, Design and Implementation of a Fast Crossbar Scheduler, in: Proc. Hot Interconnects, 1998.

[10] P. Gupta, N. McKeown, Designing and implementing a fast crossbar scheduler, IEEE Micro 19 (1) (1999) 20–28.

[11] K. Lee, S.-J. Lee, H.-J. Yoo, A high-speed and lightweight on-chip crossbar switch scheduler for on-chip interconnection networks, in: Proc. European Solid-State Circuits Conf. (ESSCIRC), 2003, pp. 453–456.

[12] C.E. Savin, T. McSmythurs, J. Czilli, Binary tree search architecture for efficient implementation of round robin arbiters, in: Proc. Int. Conf. Acoustics Speech and Signal Processing (ICASSP), 2004, pp. 333–336 vol. 5.

[13] G. Dimitrakopoulos, N. Chrysos, K. Galanopoulos, Fast arbiters for on-chip network switches, in: Proc. Int. Conf. Computer Design (ICCD), 2008, pp. 664–670.

[14] J.-M. Jou, Y-.L. Lee, An optimal round-robin arbiter design, J. Inf. Sci. Eng. 26 (2010) 2047–2058.

[15] J.G. Delgado-Frias, J. Nyathi, A high-performance encoder with priority lookahead, IEEE Trans. Circuits Syst. I 47 (2000) 1390–1393.

[16] C.-H. Huang, J.-S. Wang, Y.-C. Huang, Design of high-performance CMOS priority encoders and incrementer/decrementers using multilevel lookahead and multilevel folding techniques, IEEE J. Solid-State Circuits 30 (1) (2002) 63–76.

[17] S. Abdel-hafeez, S. Harb, A VLSI high-performance priority encoder using standard CMOS library, IEEE Trans. Circuits and Systems–II: Express Briefs 53 (8) (2006) 597–601.

[18] T.B. Preußer, M. Zabel, R.G. Spallek, About carries and tokens re-using adder circuits for arbitration, in: Proc. IEEE Workshop Signal Processing Systems Design and Implementation (SiPS), 2005, pp. 59–64.

[19] N.McKeown, The iSLIP scheduling algorithm for input-queued switches, IEEE/ACM Trans. Networking 7 (2) (1999) 188–201.

[20] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, M. Horowitz, Tiny Tera: A packet switch core, IEEE Micro 17 (1) (1997) 26–33.

[21] X. Gao, Z. Zhang, X. Long, Round robin arbiters for virtual channel router, in: Proc. IMACS Multiconf. Computational Engineering in Systems Applications (CESA), 2006, pp. 1610–1614.

[22] M. Weber, Arbiters: Design Ideas and Coding Styles, in: Synopsys Users Group Meeting (SNUG), also available on citeseerx.ist.psu.edu, 2001.

[23] P.M. Kogge, H.S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence relations, IEEE Trans. Computers C-22 (8) (1973) 786–793.

[24] R.E. Ladner, M.J. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838.

[25] R.P. Brent, H.T. Kung, A regular layout for parallel adders, IEEE Trans. Computers C-31 (3) (1982) 260–264.

[26] T. Han, D.A. Carlson, Fast area-efficient VLSI adders, in: Proc. Symp. Computer Arithmetic, 1987, pp. 49–56.

[27] K. Yoshigoe, K. Christensen, A. Jacob, The RR/RR CICQ switch: Hardware design for 10-Gbps link speed, in: Proc. Performance Computing and Communications Conf., 2003, pp. 481–485.

[28] O. Baskirt, New logic architectures for round robin arbitration and their automatic RTL generation, MS Thesis, Bahcesehir University, 2008.

[29] S.Q. Zheng, M. Yang, J. Blanton, P. Golla, D. Verchere, A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling, in: Proc. Midwest Symp. Circuits and Systems, 2002, 671–674 vol. 2.

[30] N. Chrysos, G. Dimitrakopoulos, Practical High-Throughput Crossbar Scheduling, IEEE Micro 29 (4) (2009) 22–35.

[31] Y.-L. Lee, J.-M. Jou, Y.-Y. Chen, S.-S. Wu, Optimal arbiter design for NoC, in: Proc. Int. Computer Symp. (ICS), 2008, pp. 288–293.