A GALS Solution Based on Highly Scalable, Low Latency, Crossbar Using Token Ring Arbitration

Tejpal Singh
Massachusetts Microprocessor Design Center, Intel
tejpal.singh@intel.com

Alexander Taubin
ECE Department, Boston University
taubin@bu.edu

Abstract - **This paper presents a new low latency Crossbar design that can be used to interface systems working at different frequencies. For case of multiple input ports contending for same output port contemporary designs provide localized arbitration solution. The arbitration circuitry itself is a major contributor to the area and latency of the crossbar. This hinders the scalability of contemporary designs.**

A crossbar using a distributed arbitration mechanism in the form of token rings, is presented in this paper. This implementation provides a highly scalable solution, in terms of both area and latency.

I. INTRODUCTION

With the increases in die size and clock frequency, it has become increasingly difficult to drive signals across die [1, 6]. To reduce clock skew and power, the general trend is towards the use of multiple clock domains on a single die. Inter clock domain communication can be enabled in both synchronous [10] and asynchronous [2, 3, 5, 7, 9] fashion. The International Technology Roadmap for Semiconductors (2005 Edition, ITRS05) [5] states a requirement of asynchronous global signaling (to handle multiple clock domains). GALS [9] (globally asynchronous locally synchronous) is a methodology that is supposed to address this problem according to ITRS05. It enables the use of clocked design for smaller scale functional unit (which has been industry standard approach). It also provides the ability to connect synchronous functional units using robust asynchronous interconnect.

The efficient design of an asynchronous crossbar is one of the most promising implementation of GALS methodology. This solution is particularly suitable for systems where multiple input ports need to communicate with multiple output ports. For the case where a single input port always communicates with a single output port, a simple FIFO is a better solution [3].

To resolve the issues related to multiple input ports contending for the same output port, contemporary designs [4] use an arbitration tree. An arbitration tree comprised of MUTEX elements is expensive in terms of the overall gate count (area, power), the number of gate delays (latency) to determine the overall winner and the number of transitioning nodes (power). These issues make the arbitration tree style of implementation less scalable.

Ring style arbitration [3, 13] provides a scalable alternative to tree style arbitration. Scalability of ring style arbitration comes from 1) Lower gate count compared to tree arbitration as the number of ports increases 2) lower latency under most distributed work loads because token ring has less number of transitioning nodes to determine a winner.

The latency of the proposed implementation is further improved by implementing asynchronous to synchronous and synchronous to asynchronous interface logic using bidirectional signal. These signals serve as both request and acknowledge and exhibit a very fast GasP like implementation [12] (however, unlike GasP the implementation is not self resetting).

Related work. Fulcrum Microsystems has presented an asynchronous crossbar interconnect, which uses the name Nexus [7]. Their implementation is 16-port, 36 bit asynchronous crossbar. Nine 16 by 16, four bit crossbars comprise a 36-bit datapath. Fulcrum Microsystems maintains a patent on this design [4]. The arbitration logic in this design is comprised of a binary tree of arbitration (MUTEX) and merge-elements. Irrespective of the timing of requests, the arbitration circuitry sits on the forward critical path.

II. THE CROSSBAR ARCHITECTURE

The crossbar presented in this paper is configured as 4 rows and 4 columns, with 4 bits of data transported from sender at the input ports to receiver at the output ports. The crossbar structure shown in figure 2.1 is subdivided into input ports, steering logic and output ports. Data is driven along the rows from the input ports. Each of the four columns supplies data to their respective output ports. Every intersection of rows and columns has steering logic that steers the data from an input port to a desired output port.

The input port communicates with the synchronous sender. It consists of a 4 bit data receiver (data port) circuit and 4 request ports. The data port converts the input to 4 phase dual rail protocol. When the input port is available, the request port receives a 1 hot request from sender. The input request port steers the data to an output port after winning the arbitration. The arbitration between input ports is implemented as a token ring (located in the input port) shown in figure 2.1. In the case where multiple input ports are contending for the same output port, the arbitration logic decides the winner.

Each input port has one request wire for each output port. In this way, a 4x4 cross bar has 16 request wires. Conversion of data to four phase dual rail protocol enables a QDI (quasi delay insensitive) implementation, which enables better scaling as we move into deep submicron processes.

Steering logic sits at every intersection of row and columns. This gives crossbar the ability to route data from any input port to any output port. A request from an input port will selectively enable an appropriate steering logic to connect an input port to the chosen output port.

An output port communicates with synchronous receiver. An output port also contains the completion detection circuit, for the data propagating through the crossbar. The completion detection circuitry indicates to the input port and the synchronous receiver that a valid data has been transported to the output port.

III. THE PROTOCOL

Depending on the system workload, crossbar can be operating in one of the following modes.

Mode 1: A request is waiting for the token. This mode should be the most common case and represents a fully loaded system. In this mode, the token arrival will enable the request to the output port.

Mode 2: The token is waiting for the request. This mode represents a lightly loaded system.

Mode 3: The token and the request arrive at same time. This is a corner case mode and should be rare enough that performance is not defined by it. The implementation will ensure that the appropriate request propagates through the crossbar.

In mode 1 the input protocol is as follows. The input request (Req_st[i]↑) will be waiting for the arrival of a token in the appropriate token ring as shown in Figure 2.1. The arrival of token (T[i]↑), in the input port with available request (from synchronous sender), will fire the input port request to desired output port (Req_ig[i]↑). Availability of data at the output port will be indicated by the data completion detection. The data completion (Reqout[i]↓) from the output port will enable a new request to synchronous sender (Req_st[i]↓), disable current request (Req_ti[i]↓, Req_ig[i]↓) and enable token propagation (T[i]↓ → T[i+1]↑). Req_st[i]↓ is an indication (acknowledge) to synchronous sender that data has propagated across the crossbar. When ready, the sender will enable a new request to crossbar (Req_st[i]↑). Arrival of data at the output port initiates the completion detection. This completion detection will arbitrate against the clock and assert a request to the synchronous receiver Reqout[i]↓. When available, the synchronous receiver accepts data by driving Reqout[i]↑. This, in turn, will enable the input request and data port.

IV. IMPLEMENTATION

The crossbar implementation is subdivided into: 1) Input port 2) Grid element 3) Output port.

A. Input port

The input port can be further sub divided into: 1) the circuit that interfaces to synchronous sender 2) the circuit that converts single rail data to 4 phase dual rail data 3) the circuit to propagate the request to the appropriate output port 4) the token ring to arbitrate among the synchronous senders.

Figure 4.1.1 shows the input request port implementation. The input port implementation in figure 4.1.1 is subdivided into; input request port, token ring and crossbar interface to the synchronous sender. The completion detection of data Reqout[i]↓ (figure 4.1.3) on the output port will disable the initiating input request port and concurrently generate acknowledge for the synchronous sender Req_st[i]↓ (figure 4.1.1 and equation 4.1.1).

if($\overline{\text{Req_im[i]}}$ **and** Reqout[i]↓) **then** Req_st[i]↓ (4.1.1)

The crossbar grid element component (represented with equations

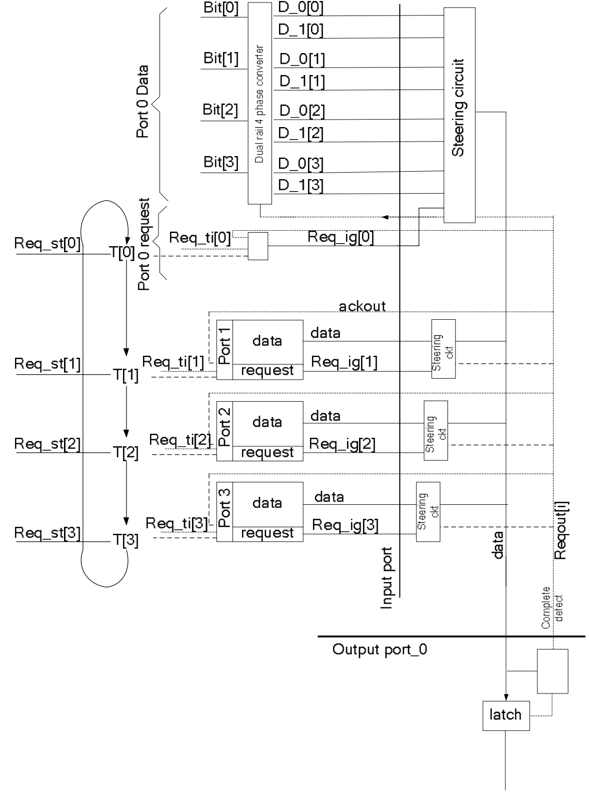

Figure 2.1. Crossbar interface showing one output port and four input port

4.1.2b and 4.1.2c) will steer the Reqout[i]↓ to the selected data port. The equation in 4.1.2a represents the inversion relationship between Ackout[i] and Reqout[i]. By steering the completion from an output port to the selected input data port (Ackdat[i]↓), enables NULL to be driven from the selected data converter port.

if (Reqout[i]↓) **then** Ackout[i]↑ (4.1.2a)

if (Ackout[i]↑ **and** Req_ig[i]) **then** Ackdat[i]↓ (4.1.2b)

if (Ackdat[i]↓) **then** ackc[i]↓ (4.1.2c)

Acknowledge from an input port (Req_st[i]↓), will propagate to the sender only when clock (clk) is high (shown in figure 4.1.1 with an inverted clock into the metastability filter (MF)). The MF is implemented as MUTEX from [11]. This implies that asynchronous crossbar is responsible for synchronization. The setup for the synchronous receiver is just before the rising edge of the local clock. The signal not switching in low phase of the local clock, guarantees that acknowledge will setup before next rising edge of the clock.

Req_st[i] is implemented as a bidirectional dynamic line, with its state defined by pulses of appropriate duration. The duration of pulses in this implementation is 3 gates. The pulsed nature of interconnect allows a single wire to send request and acknowledge across synchronous and asynchronous domain. The synchronous sender will check if it has received acknowledge (Req_st[i]↓) from the crossbar and both set Syn_req[i]↑ (figure 4.1.1) and place data on the interface. For a given input port, a one hot Ctl[i]↓ pulse (shown in figure 4.1.1 and equation 4.1.3), enabled by Syn_req[i] and clk↑, will enable the transmission gates that transfer data to the asynchronous crossbar. The same Ctl[i]↓

pulse will also enable a new request Req_st[i]↑ from the synchronous sender (equation 4.1.4).

if (Syn_req[i] **and** clk↑) **then** Ctl[i]↓ (4.1.3)

if (Ctl[i]↓) **then** Req_st[i]↑ (4.1.4)

A keeper circuit shown in figure 4.1.1 for request (applies to data as well), will be required because of the dynamic nature of Req_st[i] (and data). An input data port converts single rail data to 4 phase dual rail data. The Reqout[i]↑ indicates that the synchronous receiver has sunk the data. This in turn causes the data and the input request port to be enabled for a next possible request from the synchronous sender.

Figure 4.1.2 shows the token ring implementation. A valid request from the synchronous sender Req_st[i]↑, is one of enabling conditions for the request port (Req_ti[i]↑). The arrival of token (T[i]↑) after a valid request will cause transition on the output of the request port (figure 4.1.1) and send a request across the crossbar Req_ig[i]↑. The AE (arbitration element) is included in the ring to cover the corner case, when token arrival (T[i]↑) and valid request (Req_st[i]↑) transition at the same time. The disabling of valid request (Req_st[i]↓) will cause the token propagation (T[i]↑ to T[i+1]↑).

The availability of token in T[i+1] and T[i+2] will make T[i] NULL. Only T[i+1] needs to be NULL to enable AE corresponding to T[i]. A token will propagate through the ring, unless it hits a valid request. From this moment onwards the token will be gated by the request that is being serviced by the token. The token will be released into the ring to service next request, only after the current request has been serviced. When starting from reset, the starter complements AE to inject one token into the ring.

B. *Grid element*

The grid element can be subdivided into: 1) forward path data steering circuit 2) feedback path acknowledge steering circuit. The winning request port (Req_ig[i]↑ in figure 4.1.1) will direct the data from a requester input port to a destination output port using the forward path data steering circuit. The data steering circuit implementation requires just 2 transistors for each bit of data transfer to an output port. The feedback circuit directs acknowledge from an output port to the correct input data port. The steering of acknowledge is done by same request that steers the forward data. The input data and request port will be enabled on receiving Reqout[i]↑ from the output port, indicating that the synchronous receiver has sunk the data.

C. *Output port*

The output port can be subdivided into: 1) completion detection to indicate valid data at the input of output port C-element latch 2) interface logic to transfer data from asynchronous to synchronous domain. The availability of data token at the input of latch will be indicated by the completion detection circuit. The completion detection in an output port will trigger four concurrent activities 1) capture data into the latch 2) a pre-charge of data at the input of latch 3) Reqout[i]↓ to the input port, to drive a NULL token from the data and request port 4) send request to the synchronous receiver indicating availability of a valid data in the output port. The high level of concurrency and pulsed nature of interconnect (like GasP) makes this implementation a very low latency design.

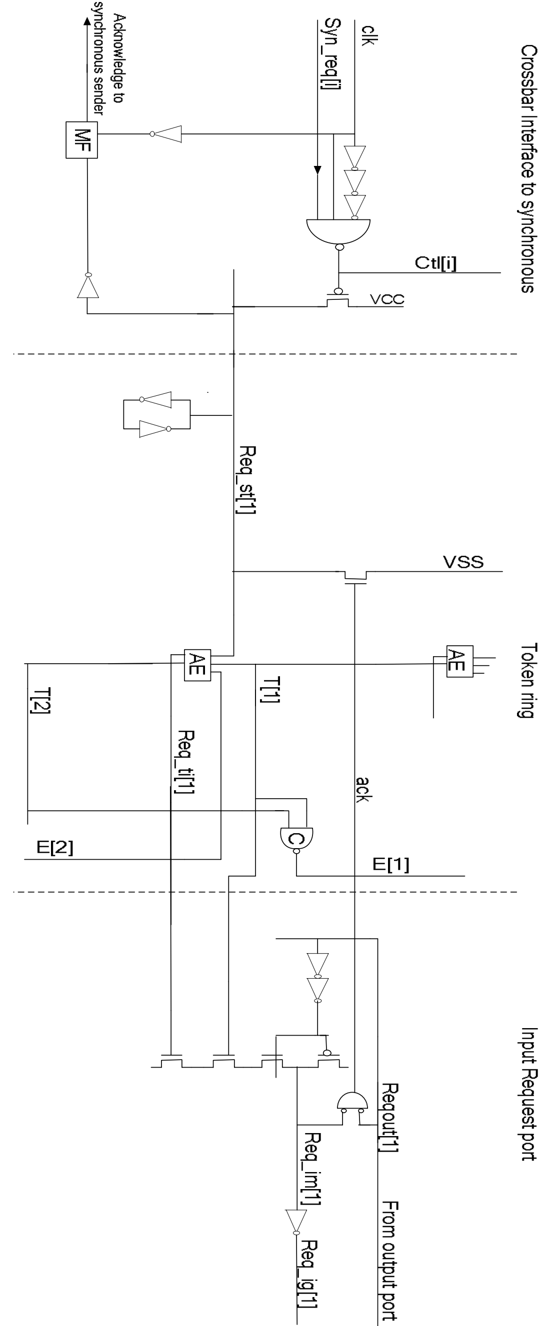

Figure 4.1.1. Crossbar input port

As shown in figure 4.1.3, asynchronous to synchronous request transfer is triggered off the completion detection in the output port Reqout[i]↓. The request is allowed to propagate to the output receiver if the clock (clk) is high. This guarantees the setup of data to next rising clk edge on the synchronous receiver flop.

The acknowledge from receiver (ackr↑) will: 1) enable the transfer of data from asynchronous crossbar output port to the synchronous receiver 2) enable the pre-charge (NULL) data value to be latched into the output port latch 3) enable input request and data port.

Figure 4.1.2. Token ring implementation

V. SIMULATION RESULTS AND COMPARISION TO NEXUS TREE ARBITRATION

The case simulated is Mode 1 from section 3. The arrival of token T[i]↑ at this input port will send a request across the crossbar to the output port. To evaluate performance of the crossbar presented in this paper the circuits were implemented using TSMC's 180nm technology. All the simulations were done at three voltages 1.5V, 1.8V, 2.1V, 25°C and TT corner using cadence® tools. All the side loadings and topology sizes were carefully incorporated into simulations to increase their accuracy.

To achieve realistic timing, the gates on critical path were layed out. The simulation setups included extracted capacitance and resistance for these gates. The layout of gates was also used to calculate dimensions of input port, output port and the length of interconnect. The wire (interconnect) capacitance was calculated based on minimum width fully shielded wires. The per unit wire capacitance was obtained from TSMC technology file.

The graph in figure 5.1 shows the delay through the tree arbitration circuit as a horizontal line. Same graph also shows the total delay through a token ring for various hops. For the token ring latency calculations, an 8 entry token ring was simulated at 1.8V, 25°C and TT corner. This gives a comparison point against

Figure 4.1.3. Output port

8 entry arbitration tree from [4]. From figure 5.1, for hop distance of 2.5 (average for ring with 4 places), token ring gives better arbitration latency (300ps) compared to the tree arbitration (455ps). Latency of the tree arbitration in figure 5.1 was calculated under best case assumptions. Accordingly, any delay associated with the loading buffers or feedback path in [4] was assumed to be zero. For tree arbitration, the delay associated with the buffers to drive high fan-out nets will be significant as the number of crossbar ports increase.

Table 5.1 shows latency scaling of the crossbar as the voltage is scaled for different modes of operation. The total latency through the crossbar corresponds to Lat_xb from equation 5.1. Grid latency (Lat_g) corresponds to latency from input of request port to the input of output port. The implementation in figure 4.1.1 (input request port) and steering logic are associated with this latency. The output port latency covers the latency associated with the pulse generator circuit and output port latch. The implementation in figure 4.1.3 is associated with this latency.

When operating in mode 1 (from section 3), the total forward latency of crossbar is defined as:

$$Lat_xb = Lat_tr + Lat_g + Lat_op \quad (5.1)$$

Where:

Lat_xb: is total crossbar latency
Lat_tr: is token ring latency
Lat_g: grid latency
Lat_op: is output port latency

Table 5.2 gives the gate count for token ring arbitration vs. the tree arbitration. For this gate count each inversion stage is considered as a gate. To get the gate count for tree arbitration, the tree arbitrator structure in [4] was scaled for 8x8, 16x16 and

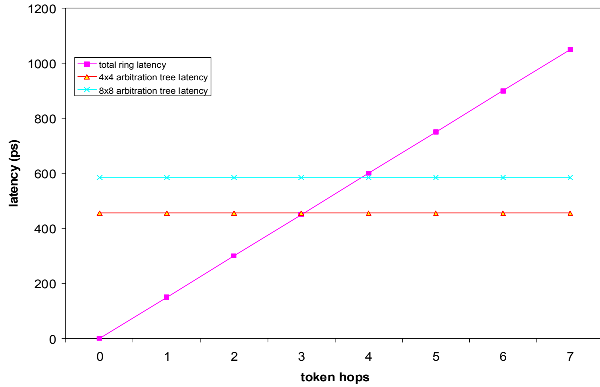

Figure 5.1. Ring latencies plotted against the tree latency for 4x4 an 8x8 crossbar.

Table 5.1 Total crossbar latency at different voltages

		latency(ps)			
voltage(volts)	hop distance	token ring	req/str	output port	total
1.5	0	0	675	200	875
	1	170	675	200	1045
	2	340	675	200	1215
	3	510	675	200	1385
1.8	0	0	550	160	710
	1	150	550	160	860
	2	300	550	160	1010
	3	450	550	160	1160
2.1	0	0	475	130	605
	1	105	475	130	710
	2	210	475	130	815
	3	315	475	130	920

32x32 configurations. The buffers (which will be significant part of total gate count, as the system scales) required for driving high fanout loads have not been rolled into total gate count for tree arbitration shown in table 5.2. The tree arbitration structure in [4] gives the encoded value of the winning port. The decode logic will be required at intersection of each row and column (ignored here) will be a significant contributor to the overall gate count. To get the gate count for the token ring arbitration the structure in figure 4.1.3 was scaled for 8x8, 16x16 and 32x32 configurations. The maximum fan-out for token ring is 2. Moreover, this fan-out is to the localized gates (next stage in token ring and feedback C element).

Table 5.2. Gate count for arbitrators with system scaling

	arbitration gate count	
crossbar size	token ring arbitration	tree arbitration
4x4	8	12
8x8	16	33
16x16	32	78
32x32	64	171

VI. CONCLUSIONS

A new low latency highly scalable crossbar design has been proposed. The proposed design has, input ports arbitrating for the output ports using distributed arbitration scheme, implemented as token rings.

The scalability of a design is a function of latency, bandwidth, area and power utilization of the design as the system size increases. The timing analysis and SPICE simulations, results demonstrate that under distributed (high bandwidth) workload conditions, proposed design provides a lower latency arbitration topology compared to contemporary designs utilizing arbitration trees. The nice feature of proposed design is the decrease in the crossbar latency as the system bandwidth increases and the average hop distance becomes smaller. This compares favorably with the tree arbitration which has constant forward path latency irrespective of the system bandwidth.

The gate count and area of a token ring arbiter increases at much lower rate compared to tree arbitration as the crossbar size increases. The token ring arbiter will have less leakage power because of less overall device area. Moreover, the token ring arbiter will also consume much less active power because of significantly lower fan-out and significantly lesser number of switching nodes.

VII. REFERENCES

[1] Bowman K. A., Duvall S.G. and Meindl J. D. Impact of die-to-die and within die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. IEEE Journal of solid-state circuits, 183-190, Feb. 2002.

[2] Chakraborty Ajanta and Greenstreet Mark R. Efficient self-timed interfaces for crossing clock domains. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 78-88. IEEE Computer Society Press, May 2003.

[3] Chelcea Tiberiu and Norwick Steven M.. .Low latency asynchronous FIFO using token rings. Advanced Research in Asynchronous Circuits and Systems, 2000, 210 – 220.

[4] Cummings et al. Asynchronous crossbar with deterministic or arbitrated control. United states patent application. Publications number US 2003/0146075 A1.

[5] Heath M. and Harris I. A deterministic Globally synchronous Locally Synchronous Microprocessor Architecture, Proceedings Microprocessor Test and Verification pp 119-124 May 29-30, 2003.)

[6] International Technology Roadmap for Semiconductors. 2005 Edition. http://www.itrs.net/Common/2005ITRS/Home2005.htm

[7] Lines Andrew. Asynchronous interconnect for synchronous SoC design. IEEE Micro, 32-41, 2004.

[8] Martin Alain J.. Asynchronous circuits for token-ring mutual exclusion. Computer science department California Institute of technology, 1990.

[9] Muttersbach J., Villiger T., Kaeslin H., Felber N. and Fichtner W. Globally-asynchronous locally-synchronous architectures to simplify the design on on-chip systems. Proc. of the 12th IEEE International ASIC/SOC conference, 317-321, 1999.

[10] Nose Koichi et al. NEC, Japan. Deterministic inter-core synchronization with periodically all-in-phase clocking for low-power multi-core SoCs. ISSCC 2005.

[11] Seitz C.L. system timing. Introduction to VLSI systems, chapter 7. Addison-Wesley, 1980.

[12] Sutherland I. and Fairbanks S. GasP: A minimal FIFO control. In the proceedings of the seventh international symposium on advanced research in asynchronous circuits and systems, 46-53, Apr. 2001.

[13] Yakovlev A., Varshavsky V., Marakhovsky V., Semenov A. Designing asynchronous pipeline token ring interface. Conference on Asynchronous Design Methodologies 1995.