Scalable and Partitionable Asynchronous Arbiter for Micro-threaded Chip Multiprocessors

Nabil Hasasneh
Department of Engineering
University of Hull
Hull, HU6 7RX
N.Hasasna@eng.hull.ac.uk

Ian Bell
Department of Engineering
University of Hull
Hull, HU6 7RX
I.M.Bell@hull.ac.uk

Chris Jesshope
Department of Computer Science
University of Amsterdam
Kruislaan 403, Amsterdam
Jesshope@science.uva.nl

ABSTRACT

This paper presents a scalable and partitionable asynchronous bus arbiter for use with chip multiprocessors *(CMP)* and its corresponding pre-layout simulation results using *VHDL*. The arbiter exploits the advantage of a concurrency control instruction *(Brk)* provided by the micro-threaded microprocessor model to set the priority processor and move the circulated arbitration token at the most likely processor to issue the create instruction. This mechanism provides latency hiding during token circulation by decoupling the micro-threaded processor from the ring's timing. It is shown that this arbiter can be extended easily to support large numbers of processors and can be used for chip multiprocessor arbitration purposes.

Keywords

Asynchronous, Synchronous, GALS, Micro-threaded, Concurrency, CMP, Broadcast Bus, Ring, Scalability, Partitioning, Insensitive Delay model.

1. INTRODUCTION

Integrated Circuits (IC's) in the future will require new efficient and powerful architecture designs to achieve the demands of many grand-challenge applications, such as weather and environmental modeling, computational physics and biomolecular simulation, as in near future, we will be able to integrate thousands of arithmetic units on a single chip [1].

Modern synchronous Chip Multiprocessor (CMP) architectures are based on a single clock domain with global synchronization and control signals. The designers must be very careful in their control signal design in order to satisfy the timing and synchronization requirements. These constraints will restrict the system on-chip performance and prevent overall system scalability. It is abundantly clear that clock-skew, large power consumption, and growing clock rate pose the most significant challenges for present and future systems on-chip.

The history of the IBM Power PC *(PPC)* processor shows that clock speed has increased at twice the predicted rate, i.e. from 33MHz to 1GHz over the last twelve years, but increases in system-level concurrency have not tracked the packing density [2]. Intel expects that its largest processor will reach 1.7 Billion transistors at the end of 2005, we may ask if these transistors are being effectively used. Evidence that all is not well is provided by the fact that Intel has cancelled its 4GHz Pentium 4 [3], because it has been proved that this processor has effectively reached the limit of its performance and has poor scaling properties. It can be argued that simply increasing the clock speed and using more and more transistors (enabled by smaller feature sizes) is a poor strategy for future generations of processor architectures and does not guarantee better performance.

Recently, asynchronous communication has proved to be a promising design method with good scaling properties. The Globally Asynchronous Locally Synchronous *(GALS)* design style is an approach to VLSI system design that holds the promise of combining the advantages of both synchronous and asynchronous operation [4]. This design approach eliminates the need for a centralized clock and minimizes the clock-skew problem. It also opens the door wide for system scalability and functional portioning, which both are the requirements for future CMP designs. The Semiconductor Industry Association *(SIA)* Roadmap recognizes that, by 2007, asynchronous techniques will be used in many designs [5]. Not every system can be decomposed into asynchronously communicating synchronous blocks easily. In CMP design, global communication is one of the most significant problems in both current and future systems [2].

The micro-threaded CMP architecture model [6, 7, 2], exploits the advantages of a GALS design approach by using a set of global buses all of which use fully asynchronous communications, creating independent clocking domains for each processor. One of these buses is the *broadcast bus*, which each micro-threaded processor uses in order to create a new family of micro-threads. The broadcast bus is also used to broadcast the global state to all processors. To avoid processor contention and to take the advantages of

asynchronous communication design methodology, this paper introduces an asynchronous arbiter design. The arbiter exploits the advantage of a concurrency control instruction *Brk*, provided by the micro-threaded microprocessor model to set a priority policy and to hide the token circulation time by decoupling the micro-threaded pipeline from the ring's timing. It also, provides multiple features, such as modularity, partitioning organization, and is starvation free.

The reset of the paper organized as follows: In the next section, we present a brief background and related work. Section 3 explains the micro-threaded approach, its concurrency controls and the micro-threaded chip multiprocessor architecture model. In section 4, the asynchronous arbiter organization and its mechanism are presented. The arbiter pre-layout simulation results using VHDL is described in section 5. Finally, we present a conclusion in section 6.

2. BACKGROUND AND RELATED WORK

Full asynchronous communication design is difficult but, one promising technique is to use a Globally-Asynchronous, Locally-Synchronous (GALS) clocking domains [4]. This design approach eliminates the global clocking problem and minimizes power consumption. GALS systems not only mitigate the clock distribution, power consumption, and the clock skew problems, but it also simplify the reuse of modules as the modules have asynchronous interface and do not need to use the same clock signal [8]. Recently, Hemani et.al. [9] compare the GALS architecture with the globally synchronous (GS) case. The results show that 70% power savings in clock distribution with negligible overheads can be achieved using GALS architecture design compared to the GS design case.

Delay modelling is one of the most significant elements of validating asynchronous design. One popular well-known approach that gives unbounded delays to both wire and gate elements is the delay-insensitive design approach. This design style avoids the need for the timing analysis, giving designs that operate correctly whatever the delay in the interconnecting wires [10]. It also has some benefits over bounded-delay methodologies in that the former delay model forces the designs to use conventions such as completion signals and transition signaling which are both important to good asynchronous circuit structure [11]. Furthermore, the delay-insensitive model allow the possibility of exploiting the average case delay rather than the worst case, which provides a significant saving with long interconnections [10]. There have been some processors used a delay-insensitive technique such as [12, 13, 14].

Asynchronous-synchronous interfaces using point to point GALS interconnect as described in [15] represent a very efficient and a suitable way to synchronize asynchronous and synchronous clock domains. The design we have described in this paper, has the advantage of asynchronous communication and provide asynchronous-synchronous interfaces using a point to point connection between arbiter modules. A delay-insensitive methodology is also applied on our arbiter by giving unbounded delays to both wires and logic gates.

It is very well-known that accessing the shared resource by two or more processors requires an arbitration mechanism to prevent contentions and to insure that only one processor can access the shared resource at a time. Many arbitration schemes have been proposed [16, 17, 18, 19, 20, 21] with different implementations characteristics. Arbiters can be centralized, decentralized, daisy chained, tree, round robin with fixed or dynamic priority, ring structure, etc. In fact, the degree of comparison between these mechanisms depends on a set of factors, such as: reusability, modularity, fairness in accessing the shared resource, avoiding starvation and minimizing both power consumption and logic area. That is, most of the arbitration mechanisms are only suitable for some cases and none of them is optimal for all cases.

Macii and Poncino [22] described a design of a scalable bus arbiter for a multiprocessor system using a ring architecture. This arbiter is synchronous in design and the priority level of each processor is reduced by one at every arbitration cycle to satisfy a rotating priority between the processors. Also, two signals (Bus_Busy and Token_Out) must be propagated through the ring network to circulate the token. Our arbiter also uses a ring structure but is a fully asynchronous design. It exploits the concurrency control instruction (Brk) provided by the micro-threaded microprocessor model to hide the token circulation time and to set a priority processor based on the processor that has succeed in executing this instruction. Also one grant signal (Gout) rather than two is propagated to circulate the grant token around the ring.

Valencia et. al. [19] presents a modular asynchronous design for an n-user linear array arbiter. In this design a centralized control signal is used to drive all the modules in the array. When this control signal is 0, the arbitration process takes place in such a way that this signal is not 1 until the requests have been granted in the same order of the module in the array. Also, the priority policy in this arbiter is dependent on the relative position of the component modules. This arbitration mechanism is not fair and provide a starvation situation if a large number of modules are used. Our arbiter has the advantage of being partitioned, where each arbiter can decide locally to access the global create bus or to wait. So, there is no need to propagate the control through all modules. Also, the priority policy described in our arbiter has multiple features, it provide fair communication and avoids processor starvation. It also, hides the token circulation time by moving the token to the most likely processor to issue the create instruction, which enable one processor to create a new family of micro-threads.

Moore et.al. [15] purposed an asynchronous-synchronous interfaces design for a point to point channel communication with independent clock domain. The authors suggested a new scheme by adding an asynchronous FIFO between producer and consumer modules to hide the waiting time during request and acknowledge synchronization path. From a hardware point of view, adding extra components means adding mechanisms and the complexity increases. Thus, this mechanism requires a complex control scheme, and in some cases, if the FIFO is deeper, the performance will be significantly degrade.

Work done in [21] described a design of asynchronous arbiters for on-chip communication system. The authors proposed a fixed and dynamic priority arbiter configuration. In

the fixed priority design, three blocks are used to handle the arbitration mechanism. These blocks are the loop control block to reactivate the arbiter after serving requests, the synchronizer block to sample the input requests and the fixed-priority block to determine the priority value based on a hardware coded priority mechanism. Dynamic priority design also has the same complexity of blocks, where n-request analyzer blocks and n-priority comparator blocks are required to handle n-requests. This arbiter has a complex arbitration design with centralized structure, which prevent partitioning. Also, many comparisons may be required to determine the priority values if the previous comparison failed in determining the priority value.

In contrast, our arbiter has less complexity and provides a simple arbitration mechanism for a large number of processors on-chip. It also, provides a simple mechanism to predetect the priority through a concurrency control instruction provided by micro-threaded microprocessor model to move the token to the most likely processor to issue the create instruction. Also, the arbiter we describe has the advantage of partitioned design; each micro-threaded GALS processor arbiter can decide locally to access the shared global create bus or to wait until the bus is free again. Therefore, there is no need to propagate the control through all processors.

Recently, Villiger et.al. [23] proposes a mechanism for transferring data between GALS modules using a self-time ring topology. This configuration provides a point-to-point communication between two adjacent GALS modules and provide a modular connectivity, which has full scalability in both bandwidth and area with increasing numbers of GALS modules. The design we described in this paper has the advantage of a ring organization that connects GALS micro-threaded processors with the broadcast bus in a circular fashion.

3. MICRO-THREADED CHIP MULTIPROCESSOR ARCHITECTURE MODEL

3.1 Micro-threaded Microprocessor Model

In this section we consider the micro-threaded microprocessor model and its features that support a future scalable and powerful CMP. This model was first introduced in 1996 [24], then extended in a set of papers [6, 7, 2, 25] to support systems with multiple processors on-chip. The model combines the advantages of blocked multi threading and interleaved multi-threading by interleaving the threads when one thread is blocked on a cycle-by-cycle basis using an explicit context switch instruction or tag, which is required when the compiler cannot guarantee that data will be available. The model exploits instruction level parallelism primarily across loop bodies, as the families of threads are defined on loops. The code for the thread is defined by the body of the loop. Other forms of instruction-level parallelism are captured using the *Cre* instruction but only the loop offers the efficiency of multiple threads created by one instruction with all threads sharing the same code.

Threads are reactivated after being suspended by a context switch when the data they were waiting for becomes available. Indeed the thread is suspended and awaits its data in the register that the compiler determined was non-

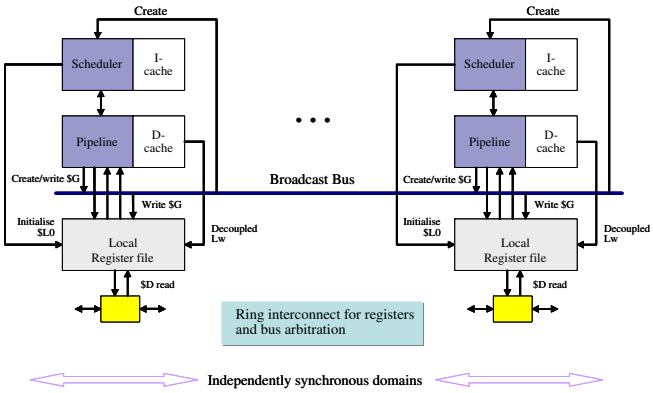

Figure 1: A chip multi-processor based on an asynchronous collection of microthreaded pipelines

deterministic. There is one other situation where the compiler may flag a context switch and that is following a branch instruction. In this case the thread is reactivated upon the computation of the branch target address. The concurrency controls used in this model provide a flexible and efficient mechanism for thread creation, context switching and synchronization. Context switching is compiler controlled by recognizing and tagging instructions which could fail synchronization.

The micro-threaded model provides instructions to create families of thread *(Cre)*, to explicitly context switch between threads *(Swch)*, to kill a thread *(Kill)* and two instructions for global synchronization, one a barrier synchronization *(Bsync)*, the other a form of a break instruction *(Brk)*, which forces a break from a loop executed concurrently. The Brk instruction terminates all other threads and leaves the issuing thread as the main thread. This instruction gives a hint that the processor needs to create a new family of micro-threads after a few cycles. Thus, a processor that has succeed in executing this instruction assert a high request signal through the Brk wire line to its arbiter to inform it that this processor will be requesting the broadcast bus. Based on this prediction, our asynchronous arbiters moves the grant token until it has reached the requesting module. This mechanism provides latency hiding and deadlock freedom during token circulation time.

3.2 Micro-threaded Chip Multiprocessor Model

A long-term vision is considered in the design and components organization of the micro-threaded CMP architecture model. This vision comes from the fact that most existing CMP designs suffer from hardware and software implementation problems. Thus micro-threaded CMPs avoid global clocking by supporting a GALS design approach, where each micro-threaded processor has its own local clock domain and accesses global resources asynchronous.

A block diagram of a micro-threaded chip multiprocessor is shown in figure 1. A set of shared components are used in this model to support the micro-threaded CMP. These com-

ponents are the *Broadcast Bus* which enables one processor to create a family of identical threads. This bus arbitrates between multiple processors and in each cycle one processor can access this bus to create a descriptor of a new family of micro-threads. The descriptor identified in the create is distributed to each scheduler, which uses that information to determine the subset of the family of threads it will execute.

The broadcast bus is also used to replicate what the compiler defines as global state to each processor's local register file instead of accessing a centralized register file for global variables. It is one of two mechanisms that allow the register file in micro-threaded model to be fully distributed between the multiple processors on a chip. The other is the *Shared Registers Ring Network*, which allows compiler-specified communications between pairs of threads, one of which produces data and the other which consumes it. This communication between the *shared* and *dependent thread* will be performed by the ring network if the threads are allocated to different processors. The justification for using a ring network is that it is scalable and, given sufficient resources, the model, as specified here, can adopt a schedule which ensures that any constant-strided, loop-carried dependency be mapped to a neighboring processor.

The two subsystems requiring global communication i.e. the broadcast bus, and the arbiter ring network use asynchronous signals, creating independent clocking domains for each processor. Therefore, the arbiter described in this paper has the advantage of an asynchronous global design and provides a good approach to preventing processor contention during broadcast bus access. Furthermore, it can be shown that the asynchronous arbiter provides a scalable and partitioned solution for a large number of processors on-chip.

The distribution of threads to pipelines, is deterministic and based on a simple scheduling algorithm. It is dynamic as it is determined by resource allocation and release (the concurrency exposed is parametric and not limited by the hardware resources). The instruction issue schedule is also dynamic and is scalable. Instructions can be issued from any micro-thread already allocated and active. The concurrency is limited only by the linearly growing hardware cost for a given chip area. Clearly if such a system could also give linear performance increases, then it can provide a solution to both CMP and ILP scalability [6].

4. ARBITER ORGANIZATION

Figure 2 shows the novel arbiter organization. Each processor has its own local control and a separate arbiter module in order to allow processor partitioning. Each arbiter module is linked to the next one in a ring arrangement and the processors are arranged in a grid layout as shown in figure 3a. Thus each arbiter can be linked to two physically adjacent ones to reduce propagation delays. Our arbiter has the optional capability of being usable in a dynamically partitionable processor array, assuming a suitable routing architecture is available. For example a possible reconfiguration of the processors in figure 3a onto two independent groups is also shown in figure 3b. However, a detailed description of the partitioning architecture is beyond the scope of this paper.

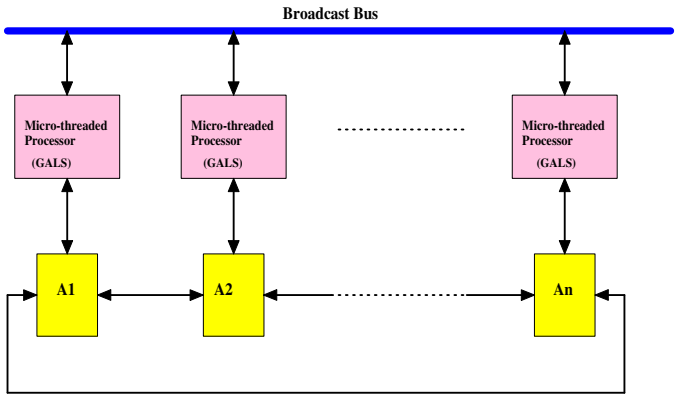

Figure 2: Asynchronous Arbiter Block Diagram

Figure 4 shows the arbiter input and output signals. As shown, the arbiters are linked by four lines comprising the request high (RH_i), which is the highest priority request, request low (RL_i), which is the lowest priority request, an acknowledgement signal (Ack_i) to release the bus, and the grant line (G_i) to grant requests and move the grant token towards the requesting module. The request and grant signals propagate in opposite directions around the ring. Also, one output wire ($Wout_i$) is required from each arbiter module to give processor P_i permission to access the broadcast bus.

There are three signals from each processor to its arbiter. The first is to inform the arbiter that the current processor has succeed in executing the Brk_i instruction, the next signal (D_i) is used to assert a demand request. The third is the local acknowledgement ($Ackl_i$) signal to inform the arbiter that a receiving processor has finished reading the data from the bus. Note that within the arbiter the Brk_i signal wire is assigned to the RH_i signal line with highest priority and the D_i signal assigned to RL_i line with low priority. Note that an initial (init) signal is also required to determine the initial location of the token. One arbiter is initialized with the token, the others without.

In order to release the bus a processor must receive an acknowledgement signal. To get that, every processor has to signal it has read the data, therefore we can reduce the acknowledgement signals back to the grantee by using the same ring connectivity to propagate the acknowledgement back until it reaches the processor that has currently reserved the broadcast bus. The required acknowledgment control circuit is shown in figure 5, where each processor asserts a high signal through its local acknowledgment ($ACKl_i$) line when that processor has read the data from the bus. A write (WR) signal is also required to control the propagation of the acknowledgment signal through the arbiter chain. Thus, the acknowledgement signal is propagated from one module to another until reached the processor that has reserved the broadcast bus. When that processor received an input acknowledgment ($Ackin_{i-1}$) signal from the previous arbiter module the processor releases the token and the arbiter responds by deasserting Wout.

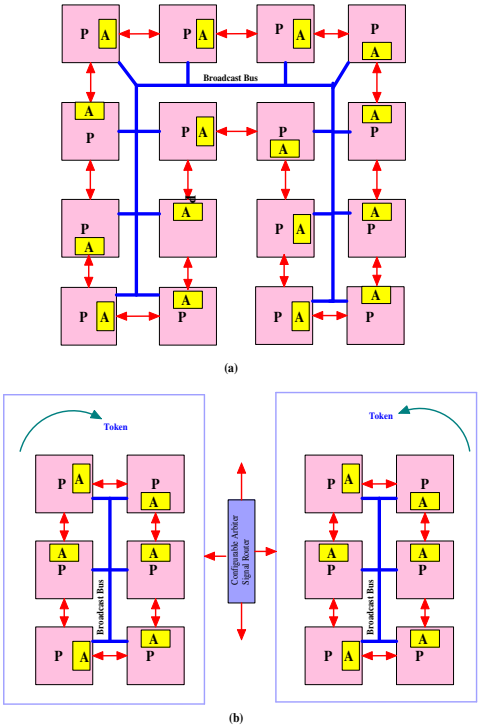

Figure 3: Asynchronous Arbiters with Different Partitioning. a)Grid Organization. b) Independent Group Organization

Our arbiter exploits the concurrency control instruction provided by the micro-threaded microprocessor model to set a priority policy based on the processor that has succeed in executing the Brk instruction, instead of just assigning the priority based on the position of the processor in the chain as described in [19]. Note that the micro-threaded pipeline executes the Brk instruction before executing the Cre instruction, which provides latency hiding during grant token circulation time.

Two levels of priority have been introduced in this paper, high and low priority. The highest priority is given to the processor that has succeed in executing the Brk instruction, while the low priority is assigned to a processor that has activated a demand request. Note that with the current micro-threaded CMP model, only one processor can succeed in executing the Brk instruction at a given time, which means there is no need for many levels of priority. However, the mechanism we described in this paper can be easily extended for many levels of priority and can be used to support any CMP arbitration model.

The arbiters operations can be described as follows, where we have N arbiter modules and only one processor can succeed in executing the Brk instruction at a given time.

- The arbiter are labelled using modulo arithmetic so for M arbiters A_{i+1} is A_0 for $i = M-1$ and A_{i-1} is A_{m-1} for M=1.
- Note that $init1 = 1$ and $init2$ to $init_m = 0$. This

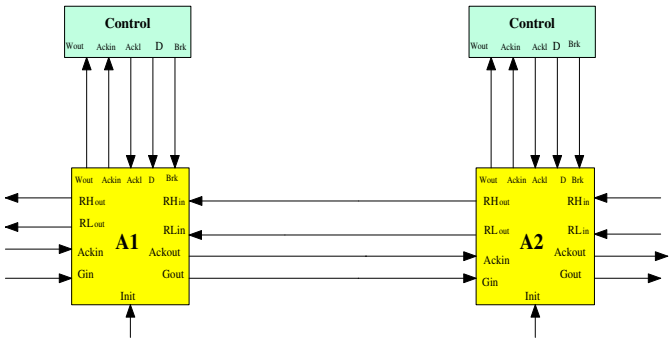

Figure 4: Asynchronous arbiter with require input and output signals

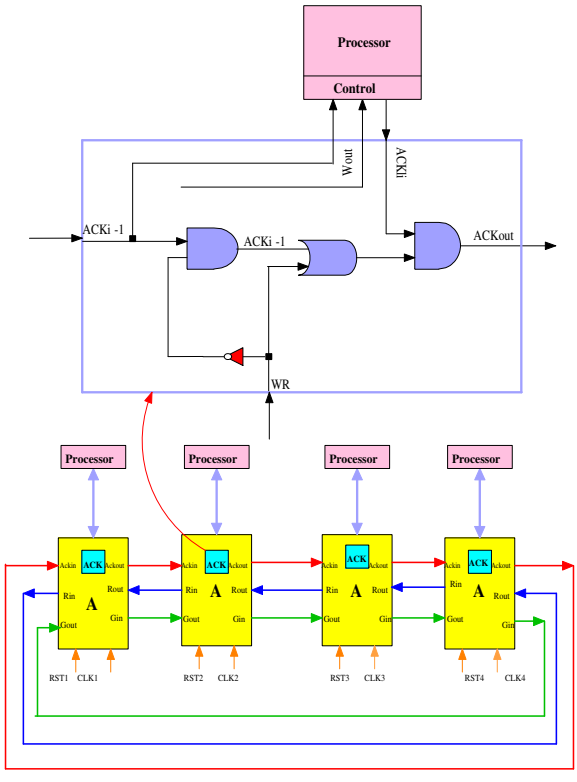

Figure 5: Released Control Circuit

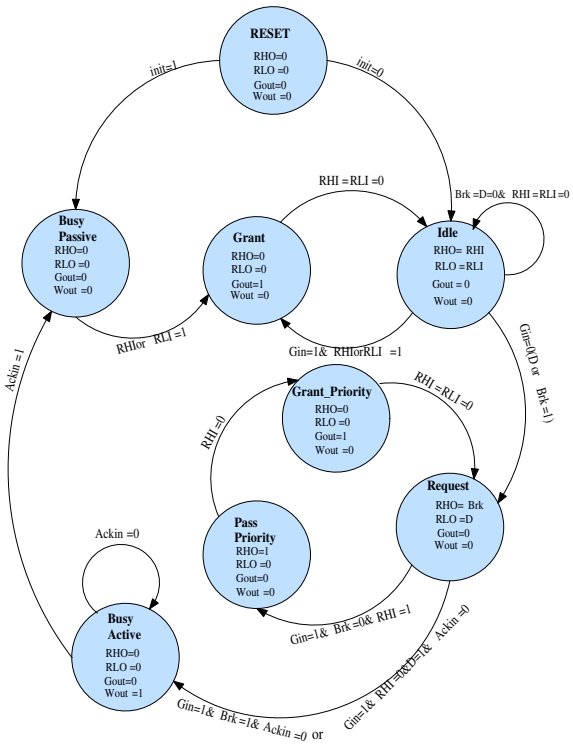

Figure 6: Arbiter State Transition Diagram

means that processor 1 would have a request acknowledged immediately after system initialization (reset) but other processors must wait for the grant to propagate ($A_1 to A_2 ..... to A_m$).

- If Brk_i =1, A_i outputs a high request to the next arbiter via $RHout_i$. The rest of the modules can also generate a demand request via $RLout_k$ where k can be any number from 1..N except i ($k \neq i$). If all Brk=0 any module can assert RLout.

- If Brk_i=0 and D_i=0, A_i propagates $RHin_i$ to $RHout_i$, $RLin_i$ to $RLout_i$ and Gin_i to $Gout_i$. This propagate RH_i and RL_i from A_i to A_{i-1} and G_i from A_i to A_{i+1}.

- If Brk_i=1 and Gin_i=1, and $Ackin_i$=0 then A_i asserts $Wout_i$ (read), which gives processor permission to access the broadcast bus.

- When a receiving processor has completed the bus transaction it asserts a local acknowledge signal $Ackl_i$=1, which also propagated through the ring until reached the module that has currently reserved the bus. Thus, when $Ackin_i$=1 and $Wout_i$=1, the token is released and the arbiter responds by deasserting Wout.

- If Brk_i=0, and the input line $RHin_i$=1 , then forward the grant to the next module irrespective of D. If D_i=1 assert $RLout_i$=1, else propagate $RLin_i$ to $RLout_i$.

- If Brk_i=0, and input line $RHin_i$=0, and demand request D_i=1 and $Ackin_i$=0, then activate the $Wout_i$, which gives the processor permission to access the broadcast bus.

- If Brk_i=0, and Gin_i=1 and $RHin_i$=0, and demand request D_i=0, and $RLin_i$=1, then forward the grant to the next module.

- When there is no request from any processor, then the RH_i, RL_i, G_i, Ack_i, and $Wout_i$ will all be 0.

It is clear from this mechanism that the highest priority is given always to the processor that asserts a high signal through its Brk output.

The state machine diagram for the arbiter module is shown in figure 6. As shown, there are eight states, however an asynchronous version of this machine can be minimized. Two states reset and grant priority can be eliminated, where the elimination of redundant stable states allows us to draw a simplified and minimized state machine.

The idle state receives the input requests from $RHin_i or RLin_i$ and if there is no input grant $Gin_i = 0$, it propagates the input requests to the next arbiter module via output request lines $RHout_i or RLout_i$. The request must be propagated until it reaches the module that currently holds the token. The token is stored in the busy passive state, from which high input requests from RHi or RLi cause a change to the grant state. In the grant state the machine waits for removal of the incoming request before returning to the idle state.

From the idle state an incoming bus demand from the processor (D=1 or Brk=1) causes a change to the request state. In the request state, if the input grant $Gin_i = 1$, and Brk_i= 1, and ($Ackin_{i-1} = 0$), then the state changes to busy active, which gives the processor permission to access the broadcast bus by activating the $Wout_i$ wire line. When the input acknowledge $Ackin_{i-1} = 1$ is received, this means that all processors complete accessing the bus and the state changes to busy passive. While if the input grant $Gin_i = 1$, $Brk_i = 0$ and the input request Rh_i=1, then the pass priority state is used to pass the request, ignoring the lower priority demand from this processor.

Our arbiter provides both starvation free and deadlock freedom. If we assume that the token is initially in module one and a demand requests to access the broadcast bus is encountered from all modules, then the token is given first to module one, which gives it access to the bus. When this module finished, it passes the token towards to the next module i.e. module two and so on. Thus, as described above the highest priority is given first to the module that has succeed in executing the Brk instruction, then the rest of the modules that has requested the bus is served based on the position in the ring and in sequence order. It is clear that this mechanism provides a fairness and is starvation free. As soon as the processor releases the bus the next module will be served directly.

4.1 Arbiter Partitioning

A partitionable design methodology will become one of the design requirements, which ensures low power and high performance in future processors [26, 27, 28]. It is one of the most important design issues, which is effective in block design and system verification [29]. This features makes the

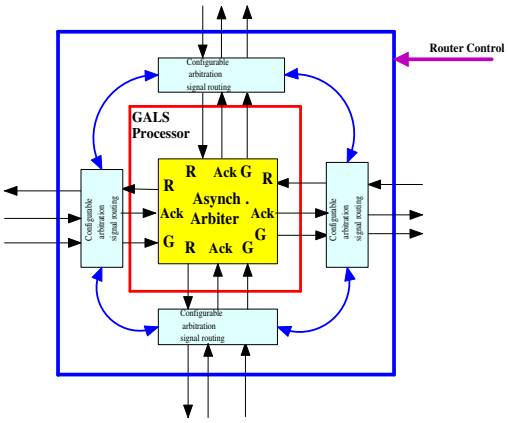

Figure 7: Asynchronous Arbiter with programmable routing for partitionable processor arrays

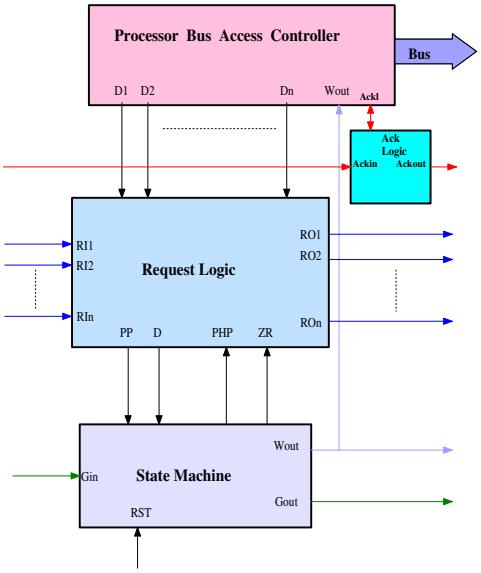

Figure 8: A block diagram for a scalable asynchronous arbiter design

design more flexible and provides a point-to-point communication between adjacent modules. Point-to-point communication in the GALS design approach provides low power and high performance [15]. It also offers a promising approach to fault tolerance problems and provides an independent communication between different system blocks.

As previously described, each arbiter connects to two other arbiters associated with adjacent processors to form an arbitration ring as shown in fig 3a. This arrangement could be hardwired, however by providing a routing architecture as shown in figure 7 a reconfiguration of processors and their buses can be achieved. So, for each arbiter and their associated global resources the processors can be partitioned into groups, where each group has a separate token.

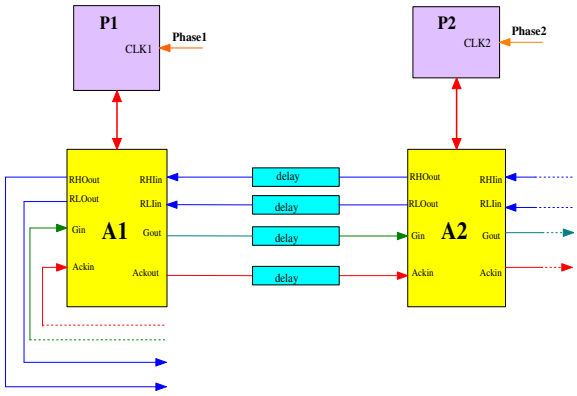

Figure 9: Asynchronous Arbiter Simulation Model

4.2 Arbiter with N-levels of Priority

Figure 8 shows a block diagram for a scalable asynchronous arbiter design with n-levels of priority. As illustrated, three blocks are required to handle n requests, which comprise the processor bus access controller block, request logic block and the state machine block. The function of the first block is to control and manipulate different levels of priority, where the priority levels can be determined by the compiler.

The second block determines whether the demand input signal has a high or low priority compared with the incoming requests. Thus if the demand line D has low priority, then a high signal is asserted to the state machine through PP wire line. Otherwise, if the demand has high priority, then PP=0 is asserted.

The state machine uses the input signals from the request logic block to decide whether to pass the grant line to the next module via Gout if the current module has a lowest priority; or to activate the Wout line, which allows the processor to access the bus. So, if the current module has the highest priority, then the pass-high-priority (PHP) signal is activated by the state machine to inform the request logic block that the bus access is given to the current module. Otherwise PHP=0 is asserted. The zero request line (ZR) can be used to control all output request RO, which block the propagation of output request RO if ZR=0, or to pass the request to the next module if ZR=1.

5. SIMULATION RESULTS

Our asynchronous arbiter is simulated using VHDL, and exploits the advantage of the generate statement provided by this language to generate N arbiter modules in the arbiter test bench. The arbiter is simulated with different processors having different clock phases with different frequencies in order to model their globally asynchronous nature. A different number of processors i.e. 2, 4, 8, 16, 32...etc. with different scenarios has been tested and verified.

Figure 9 shows the arbiter modules linked using arbitrary delay elements. The delay insensitive model uses unbounded delays on wires and gate elements and is a suitable method for analyzing the completion and transition signaling. Therefore, no matter how long the arbiter module waits for input

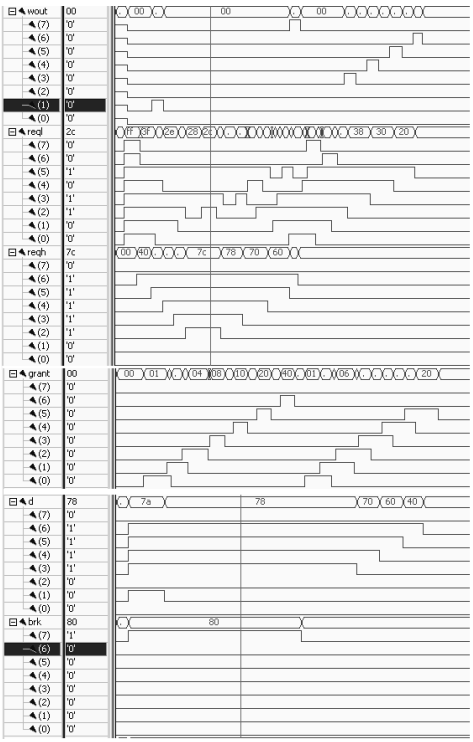

Figure 10: simulation result

changes when the arbiter sees activation of the input signals, the transition passed through the wire lines to the next module which will eventually know that a new input values has arrived.

Figure 10 shows a sample of results from simulating 8 arbiter modules. In this sample the following conditions apply: module 0 has initially reserved the token, module 7 receives a high input Brk signal line and modules 1, 2, 3, 4, 5, 6 have a high input demand requests lines. As illustrated, the request signal RL_1 reaches the token before RH_7, which means that broadcast bus access is given first to processor 1 (Wout is asserted). When processor 1 releases the token, the grant signals are propagated back to give processor 7 permission to use the broadcast bus before other low priority processors. The rest of the demand requests as shown are granted in sequence order and based on its position in the ring configuration.

6. CONCLUSION

In this paper we have discussed the design and the pre-layout simulation using VHDL of an asynchronous arbiter. The arbiter provides a very simple system architecture, where each module has a few wires connecting the next one and the last is connected to the first module in a circular fashion. Delay-insensitive methodologies with unbounded wire and gate delays were considered in the arbiter simulation procedures. The arbiter also has the advantages of GALS communication design and include the following features:

- The ring configuration to arbiter modules and the point-to-point communication between two adjacent arbiter modules provide a modular connectivity, which has full scalability in both bandwidth and area with increasing numbers of micro-threaded processors GALS modules.

- Each arbiter module has its own control signals and implements a self-timed model. Therefore, there is no need to propagate the control signals throughout all the arbiter modules.

- There are four wires connecting every arbiter module in the chain to the next one and the last to the first in a circular fashion. The latency of the wire delay is very small. Thus the decision is made locally by each arbiter module instead of using large wire delay, which gives it a partitioning properties.

- Each processor arbiter has a priority policy dependent on a processor successfully executing the concurrency control instruction Brk. This mechanism provides latency hiding by decoupling the micro-threaded processor from the token circulation time. It also, offer a fairness communication between processors and eliminates processors starvation.

7. REFERENCES

[1] Rixner, S. et al.: Register Organization for Media Processing. International Symposium on High Performance Computer Architecture, January 08-12, 2000, Toulouse, France (2000) 375-386

[2] Jesshope, C. R.: Scalable Instruction-level Parallelism. In Computer Systems: Architectures, Modeling and Simulation, 3rd and 4th International Workshops, 19-21 July SAMOS 2004, Samos, Greece, LNCS 3133, Springer (2004) 383-392

[3] shilov, A.: Intel to Cancel NetBurst Pentum 4 Xeon Evolution, http:/www.xbitlabs.com/news/cpu/display/20040507000306.html, (Accessed 7/1/2005), (2004)

[4] Shapiro, D.: Globally Asynchronous Locally Synchronous Circuits. Ph.D. Thesis, Report No. STAN-CS-84-1026, Stanford University, (1984)

[5] International Technology Roadmap for Semiconductors: ITRS, http://public.itrs.net, Accessed 4/2/2005

[6] Jesshope, C. R.: Multi-Threaded Microprocessors Evolution or Revolution. Proc. of the 8th Asia-Pacific Conference, September 23-26, ACSAC'2003, Aizu, Japan (2003) 21-45

[7] Jesshope, C. R.: Micro-grids The Exploitation of Massive on-Chip Concurrency. Cetraro HPC workshop, 31 May - 3 June, 2004, Cetraro, Italy (2004)

[8] Zhuang, S., Carlsson, W. L., Palmkvist, K. and Wanhammar, L.: An Asynchronous Wrapper with Novel Handshake Circuits for GALS Systems. In Proc. of the IEEE 2002 International Conference on Communications, Circuits and Systems, Cheungdu, China (2002) 1521-1525

[9] Hemani, A. et al.: Lowering Power Consumption in Clock by using Globally Asynchronous Locally Synchronous Design Style. In Proc. of ACM/IEEE design Automation Conference, New Orleans, Louisiana, United States (1999) 873-878

[10] Bainbridge, W. and Furber, S.: Delay Insensitive System-on-Chip Interconnect Using 1-of-4 Data Encoding. Proc. of the Seventh International Symposium on Asynchronous Circuits and Systems (2001) 118

[11] Hauck, S.: Asynchronous Design Methodologies: An Overview. Proc. of the IEEE, **83** (1995) 69-93

[12] Takamura, A. et al.: TITAC-2: An Asynchronous 32-bit Microprocessor Based on Scalable-Delay-Insensitive Model. Proc. of the 1997 International Conference on Computer Design (ICCD'97), Austin, TX, USA (1997) 288-294

[13] Martin, A. et al.: The Design of an Asynchronous MIPS R3000 Microprocessor. In Advanced Research in VLSI (1997) 164-181

[14] Huh, J., Burger, D. and Keckler, S.: Exploring the Design Space of Future CMPs. In Proc. of International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, (2001) 199-210

[15] Moore, S., Taylor, G., Mullins, R. and Robinson, P.: Point to Point GALS Interconnect. Proc. of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC'02) (2002) pp 69-75

[16] Guibaly, F.: Design and Analysis of Arbitration Protocols. IEEE Trans. Computers **38** (1989) 161-171

[17] Wilkinson, B.: Comments on Design and Analysis of Arbitration Protocols. IEEE Trans. Computers **41** (1992) 348-351

[18] Taub, D.: Corrected Setting Time of the Distributed Parallel Arbiters. Proc. of the IEEE **139** (1992) 348-354

[19] Valencia, M., Bellido, M., Huertas, J., Acosta, A. J. and Solano, S.: Modular Asynchronous Arbiter Insensitive to Metastability. IEEE Transaction on Computers **44** (1995) 1456-1461

[20] Josephs, M. and Yantchev, J.: CMOS Design of the Tree Arbiter Element. Proc. of the IEEE Trans. on VLSI Systems **4** (1996) 472-476

[21] Rigaud, J-B., Quartana, J., Fesquet, L. and Renaudin, M.: High-Level Modeling and Design of Asynchronous Arbiters for On-Chip Communication Systems. Proc. of the IEEE 2002 Design, Automation and test in Europe Conference and Exhibition, (2002) 1090

[22] Macii, E. and Poncino, M.: The Design of Easily Scalable Bus Arbiters with Different Dynamic Priority Schemes. Proc. of 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set), Pacific Grove, California (1995) 211-213

[23] Villiger, T., Kaslin, H., Gurkaynak, F. K., Oetiker, S. and Fichtner, W.: Self-timed Ring for Globally-Asynchronous Locally-Synchronous Systems. Proc. of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03), Vancouver, BC, Canada (2003)

[24] Bolychevsky, A., Jesshope, C. R. and Muchnick, V. Dynamic Scheduling in RISC Architectures. IEE Proc. Computer Digital Techniques **143** (1996) 309-317

[25] Jesshope, C. R.: Implementing an Efficient Vector Instruction Set in a Chip Multi-processor Using Micro-Threaded Pipelines. Proc. ACSAC 2001, January 29-30, 2001, Gold Coast, Queensland, Australia, IEEE Computer Society, Los Alimitos, CA, USA (2001) 80-88

[26] Yingmin, L., Brooks, D., Zhigang, H. and Skadron, K.: Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. Proc. of the 11th IEEE International Symposium on high performance computer architecture (HPCA), San Francisco, CA, USA (2005) 71-82

[27] El-Moursy, A., Garg, R., Albonesi, D. and Dwarkadas, S.: Partitioning Multi-threaded Processor with a Large Number of Threads. International Symposium on Performance Analysis of Systems and Software (2005)

[28] Wang, L., and Selvaraj, H.: A Scheduling and Partitioning Scheme for Low Power Circuit Operating at Multiple Voltage. Proc. of the Euromicro Symposium on Digital System Design (DSD'03), Belek-Antalya, Turkey (2003) 144-147

[29] Paterson, C. et al.: System-on-Chip for High Speed Communication Systems. Intel Corporation and Synopsys Inc. Report (2002) 1-25