International Journal of Emerging Engineering Research and Technology
Volume 2, Issue 8, November 2014, PP 90-100
ISSN 2349-4395 (Print) & ISSN 2349-4409 (Online)

A Novel Approach to Solve Dead Lock Problem in AXI On-Chip Bus

Srikanth Manukonda
Master of Technology (VLSI System Design)
Malla Reddy Engineering College
JNTU Hyderabad, Secunderabad, Hyderabad
India
sreekanthactive@gmail.com

Santosh J
Assistant Professor, Department of ECE
Malla Reddy Engineering College
JNTU Hyderabad, Secunderabad, Hyderabad
India
Santosh.jonnala@mrec.ac.in

Abstract: *AXI (Advanced Extensible Interface) is an on chip protocol from AMBA. AXI which supports advanced data transactions burst base transactions and out of order transactions to improve communication efficiency. Burst transactions allow master to send different number of transactions without pairing with slave again and again. Out of order transactions that allow responses to be returned in an order different from their request order play an important role in this improvement. This kind of transactions faces the dead lock problem. In order to overcome the drawback we are implemented different kind of techniques and the techniques includes single slave methods and unique Id method and DALS (dead lock avoidance by least stalling method). In the paper using BSG (bus status graph) to indicate how dead lock is overcome by different methods. Each transaction can be executed with tags is called tagged transaction.AXI Bus interface is designed with the techniques and the results show that buses with the new technique are faster than those with the currently available techniques.*

Keywords: *Advanced extensible interface (AXI), bus dead- lock, on-chip bus, out-of-order transaction, tagged transaction, DALS (dead lock avoidance by least stalling method).*

1. INTRODUCTION

The Advanced High-performance Bus and advanced peripheral bus of advanced microcontroller bus architecture [1] are the most popular communication architectures for SOC designs. Both architectures are shared-link buses allowing only one master IP to access one slave IP at a time. Recently, the communication efficiency of shared link buses becomes insufficient because of their lack of parallel access capability. Some more advanced communication protocols such as Advanced eXtensible Interface (AXI) [1] that facilitate parallel communication are thus proposed. AXI define communication interface protocols and transaction types but leave communication architecture to interconnect designers. Interconnect designers thus have the freedom to implement the communication protocols using multilayer, crossbar, networks-on-chip [3], or a combination of these designs to increase the communication parallelism and designers can design their own interconnects and interfaces between the masters and slaves. AXI not only connects to respected slaves it may connects to other serial or parallel communication protocols through many interfaces called bridge networks.

AXI support various advanced transactions, including burst transactions, pipelined (outstanding) transactions, and out-of-order transactions [1], [2]. Among these transactions, the out-of order transactions serve as a major key to improve system performance. Out-of-order transactions can be executed more efficiently when a master IP, such as a processor, can handle out-of-order returns as it allows a slave core such as a dynamic random-access memory controller to service requests in the order that is most convenient, rather than the order in which requests are sent by the master [2].

2. BUS DEADLOCK

Bus deadlock is a problem that occurs when a set of IP cores communicating through a bus system is involved in a circular wait-and-hold state that cannot be resolved. This problem may crash a bus system as none of the IP cores involved in the deadlock can continue its functions. In [6], the authors invest the bus deadlock problem of a system that allows a master to execute a process only if the master is granted to access all required slaves of the process and each master will hold the slaves

granted to it until all its required slaves are granted to it. In AXI burst transactions master take control overs the slave for some time to do a read or write burst. In this situation master holds the slave and no other master can use slave till the master leaves the slave. A bus deadlock happens when each master in a set of masters is holding a slave and waiting for another slave held by another master in the set. In this type of bus deadlocks, the relation between masters and slaves is similar to that between processes and resources in an operating system (OS) where a deadlock occurs when there is a circular wait-and-hold relation among a set of processes and resources [7]. A resource allocation graph (RAG) is commonly utilized to represent the status of resource allocation in an OS [7].

A vertex in a RAG represents a process or a resource. A directed edge from a process vertex to a resource vertex denotes that the process is requesting the resource, and one from a resource vertex to a process vertex denotes that the resource is being held by the process. In an OS, a deadlock may occur when a cycle exists in the RAG [7]. In [6], the authors thus map the bus deadlock problem to the OS deadlock problem and propose a hardware approach which requires two to four clock cycles to detect the occurrence of a bus deadlock. In advanced bus/interface protocols such as AXI, a slave must return responses after some latency when it accepts a transaction request from a master. After the responses are returned and accepted, the transaction is completed and the slave must be released. As a result, a slave will be released from a master no matter whether the master is granted to access all its required slaves or not and thus the bus deadlock problem described in [6] does not happen in these protocols.

In AXI bus deadlock may occur because of the out-of-order transactions supported by these protocols. This is because in general a bus system supporting out of-order transactions also needs to support transactions that do not allow out-of-order execution. For example, a read operation must wait for a preceding write operation if they address to the same memory location. Such read-after-write operations are commonly encountered in a bus system and they clearly do not allow out-of-order execution. To distinguish the transactions allowing and not allowing out-of-order, AXI protocol define tagged transactions, in which a master tags an ID to each transaction in the way that the transactions whose responses must be returned in order are tagged with the same ID. A bus that fully supports the AXI protocol is responsible for assuring that the order of the responses returned to each master is the same as their request order if they have the same ID. Thus, the response returned by a slave may not be accepted by the bus and the slave has to wait until the order violation condition disappears. Tagged transaction must wait for the completion of an earlier issued transaction with the same ID, it is possible that a set of slaves are waiting for one another in a circular way and are all blocked by the bus, resulting in a bus deadlock. This type of bus deadlock is clearly different from that in an OS or that described in [6].

3. DATA TRANSITIONS IN AXI

Data transactions are single and burst transactions, where a single transaction is one that requests only one response whereas a burst transaction is one that requests multiple responses. These basic transactions can be either pipelined or non-pipelined. As commonly recognized, a pipelined transaction is one that can be issued before its preceding transaction is completed. As an out-of-order transaction is one that may be completed without waiting for its preceding transactions, all out-of-order transactions can be considered as pipelined ones. In AXI, the pipelined transactions can be further divided into tagged and untagged ones, where all untagged transactions must be executed in order, whereas the execution orders of the tagged ones depend on the IDs they are tagged. In this paper, we comply with the following order constraints [1], [2].

- All untagged transactions must be executed in order.
- All tagged transactions with the same tag ID must be executed in order.
- Two transactions tagged with different IDs can be executed out of order.
- There is no order restriction on the execution between one tagged transaction and an untagged transaction.

In addition to the above order restrictions, AXI require that a slave must return responses of transactions with the same tag ID in order. In this paper, we assume that the design of each slave complies with this protocol and do not consider it as one of our order restrictions.

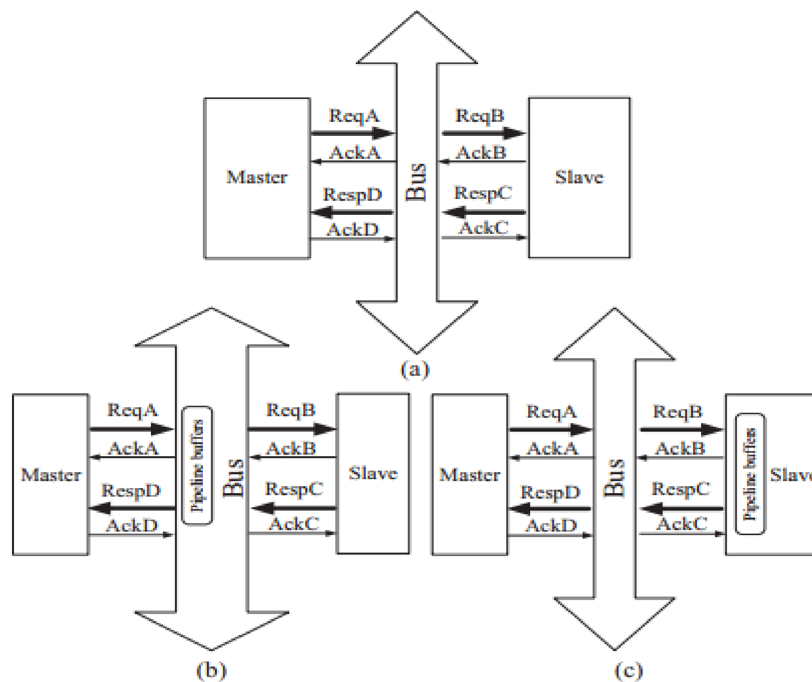


Fig. 2. Bus models for (a) basic transactions, (b) pipelined transactions with buffers in the bus, and (c) pipelined transactions with buffers in the slave.

Fig. 2(a) shows the basic bus model that is compliant with AXI [1] protocols. In this model, a complete bus transaction contains a request phase and a response phase. In a request phase, a master requests to read data from or write data to a slave, whereas in a response phase the slave responds with the read data for the read transaction or the completion status of the written data for the write transaction. We show how the basic non-pipelined transactions are executed in the bus model of Fig. 2(a). With a single transaction, a master requests to access a slave by issuing a request (ReqA) to the bus. If the bus arbiter determines that the master can access the slave, it will forward this request (ReqB) to the slave. As soon as the slave is available to accept this request, it captures the request (ReqB) and acknowledges (AckB) the bus. The bus then forwards the acknowledgment (AckA) to the master, which completes the request phase. After some access latency, the slave responds with the corresponding read data or the completion status of the written data (RespC) to the bus, which then forwards the response (RespD) to the master and waits for the acknowledgment. As soon as the master is available, it captures the response (RespD) and acknowledges (AckD) the bus. The bus can then acknowledge (AckC) the slave to complete the transaction. After the transaction is completed, the master is forced to abdicate the right to access

The slave, which implies that if the master wants to access the slave again, it must issue another request and wait for the acknowledgment. A burst transaction requires multiple data transfers. In AXI, a burst transaction is a single-request burst for which a master only specifies the request information (the access address, the burst length, the burst type, and so on) for the initial transfer. For a multi request burst transaction in AXI, a master needs to specify the request information for each transfer in the transaction. For a bus supporting pipelined transactions, successively issued and uncompleted transactions are buffered in the bus or in the slave. Because of the finite buffer space, the maximum number of transactions that a master can issue before previously issued transactions are completed is limited and known as the pipeline depth. Fig. 2(b) shows the case when the buffers are embedded in the bus, which is also the bus model used in this paper. When the master requests (ReqA) the bus, the bus will acknowledge (AckA) the master directly if there is available buffer space to buffer the transaction in the bus, and will simultaneously forward the request to the target slave. As long as the master is acknowledged, it can issue the next transaction. The remaining process, including the response phase, is similar to that of a non-pipelined transaction. The process for the case where the buffers are in the slave [see Fig. 2(c)] is similar to the above process except that the requests will be buffered in the slave. All transactions discussed so far are untagged and must be returned in the same order as their requests. When a master issue tagged transactions, the order of the returned responses for transactions with the same tag must be the same as their request order. Except for the order of the

responses, the requirement of buffers and the handshaking mechanism for tagged transactions are the same as non-tagged pipelined transactions.

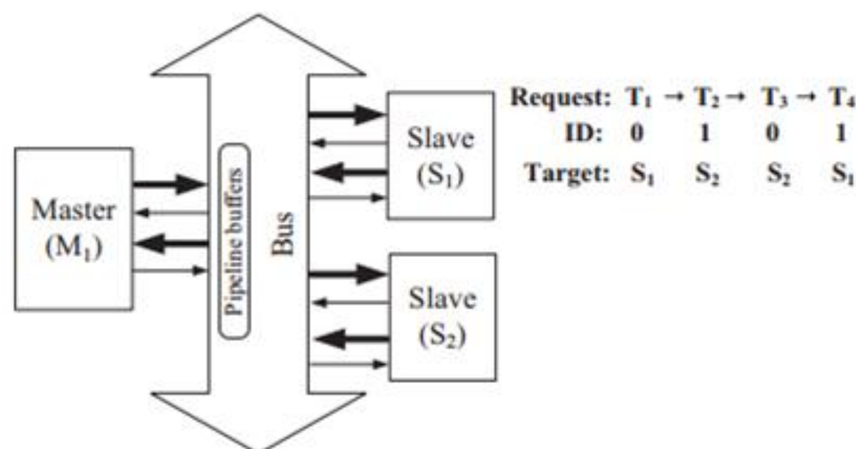


Fig. 3. Set of tagged transactions.

Fig. 3 explains the concept of IDs and the order constraints. The example is a system containing one master (M1) and two slaves (S1andS2), and supports out-of order execution. Assume that each transaction requested byM1 is tagged with an ID value of zero or one. In the example, M1 first issues a transaction requestT1to access S1 with ID=0, and thenT2 to access S2 with ID=1. After that, M1 issues requestT3to access S2with ID=0andthenT4to access S1 with ID=1. The order constraints restrict that the response of T3 must be returned after the response ofT1 because the two transactions are both tagged with zero. Similarly, the response ofT4 must be returned after that ofT2.

4. PREVIOUS WORK FOR BUS DEADLOCK

A bus deadlock is to set a timer that will expire some clock cycles later after a bus transaction is expected to be completed. However, to prevent the timer from expiring when there is no bus deadlock, the timer must be set to accommodate any possible latency of transactions. This will result in a long latency between the occurrence and the detection of a bus deadlock and thus the communication efficiency may be drastically affected. The method in [8] embeds a bus tracer in buses to monitors bus transactions. If the tracer detects a condition indicating some transaction is waiting or retrying, a counter with a predetermined number starts to count down. When the counter is decreased to zero, it is regarded as a deadlock occurs. With this method, a bus deadlock can be detected with few clock cycles. However, the predetermined number is difficult to determine appropriately. If the number is too small, designers may encounter the problem of over detection, whereas if it is too large, the latency between the occurrence and the detection of deadlocks is still long. Also, for both a timer and a bus tracer, after detecting the deadlock, some complicated mechanism is required to resume the bus system to the state before the deadlock, which may require large hardware overhead.

In [9], three methods are implemented to deal with the bus deadlock problem, namely the single-slave scheme, the unique ID scheme, and the hybrid scheme. The main idea of these methods is to stall some tagged transactions that may violate the order constraints. The single slave scheme stalls a tagged transaction targeting a slave if another slave is being accessed by an uncompleted tagged transaction. The unique ID scheme stalls requests whose IDs are already assigned to some uncompleted transactions. These two schemes, respectively, restrict that at any time, at most one slave can be accessed by tagged transactions, and each ID can be used by at most one tagged transaction. The hybrid scheme combines the two schemes by allowing any tagged transaction that satisfies either the single slave or the unique ID constraint. Though the communication efficiency can be improved, this scheme still over-stalls many tagged transactions that will not cause any bus deadlock. We will analyze this problem by using the proposed graph model later and show that our proposed technique can result in much better performance than these schemes.

5. ON-CHIP BUS DESIGN

This section describes the architecture of the on-chip bus used in this paper. We will describe how a transaction is processed inside the bus. As shown in Fig. 4, our bus supports single, burst, pipelined (outstanding), and tagged transactions. We divide the components of the bus system into three parts:

- The components for each master interface;
- The components for each slave interface; and
- The components in the center that will be shared by all masters and slaves.

Fig. 4 shows a bus system with one master and one slave. If l masters (m slaves) are to be employed, l (m) copies of the components in the master (slave) interface should be employed, whereas only one copy of the components in the center is needed.

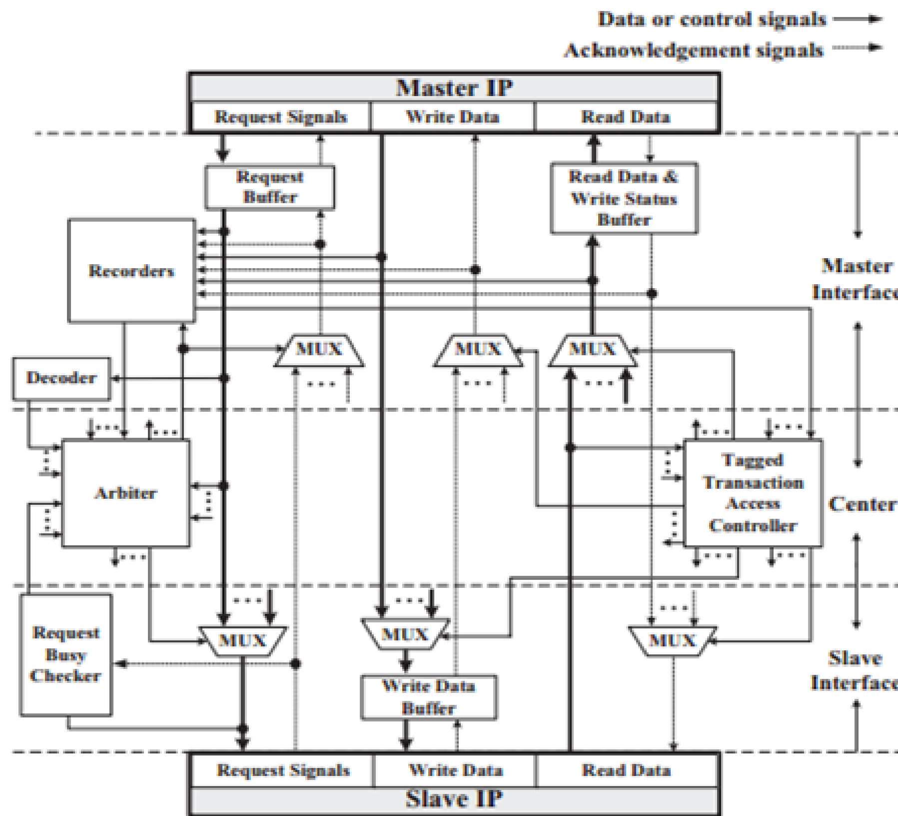


Fig. 4. Bus architecture supporting tagged transactions.

A tagged transaction starts with a master issuing a request with an ID to the bus. In the request phase, if the Request Buffer in the bus is not full, the bus acknowledges the master and the request is stored in the Request Buffer. The Decoder then decodes the transaction address of the request, and the Arbiter arbitrates whether the request can be granted to access the target slave. If it can be granted, the Arbiter forwards the request to the slave by controlling the corresponding multiplexors, and the index of the target slave is recorded in one of the Recorders, in the way that the transactions with the same tag are recorded in the same Recorder. The number of Recorders is equal to the number of IDs that the corresponding master can assign. Also the size of each Recorder is equal to the pipeline depth such that it is just enough to record all transactions that are not completed.

The Request Busy Checker checks whether the request is completed or not to assist the arbitration. For a write request, the corresponding write data are stored in the Write Data Buffer, and the Tagged Transaction Access Controller controls the multiplexors to decide from which master the write data are to be provided. After the slave accepts the request, it acknowledges the bus, and the acknowledgment is forwarded to the Request Buffer under the control of the Arbiter. In the response phase, the slave responds the bus with the read data for a read transaction or the completion status of the written data for a write transaction. If the responses can be accepted, they are stored in the Read Data and Write Status Buffer, which then acknowledges the slave. The tagged transaction access controller controls the multiplexors to decide from which slave the response is provided and from which read data and write status buffer the acknowledgment comes. Finally, the response is sent to the master when the master is able to receive it. In the bus model, the masters and slaves are connected by the bus in a crossbar manner and thus parallel transactions can be executed as long as no contention occurs. When more than one request from different masters to access the same slave arrive simultaneously, the arbiter will determine whether the request with the highest priority can be granted

or not. If it cannot be granted, the arbiter will start a new arbitration with possibly some priority updating (such as round robin). If it can be granted, the request will be forwarded to the corresponding slave. Once a request is granted, it can be processed in parallel with other granted requests. As a result, high communication parallelism can be achieved by this bus design.

6. BUS STATUS GRAPH

We now describe the proposed BSG model. A BSG contains two types of vertices, namely slave vertices and ID vertices, and two types of edges, namely prime edges and nonprime edges. Each vertex in a BSG represents a slave or a tag ID. Consider an SOC system containing m slaves and employing totally n tag IDs. Without loss of generality, we denote the set of slave vertices as S={S1, S2,...,Sm} and the set of ID vertices as ID={ID0,ID1,...,IDn−1}. The same ID values tagged by different masters are regarded as different IDs in the ID vertex set. A BSG representing this system will contain m+n vertices. To distinguish the slave and ID vertices, we use a circle and a square to represent a slave vertex and an ID vertex in a BSG. We do not model the masters in the graph. Later, the rationale for this will become clear.

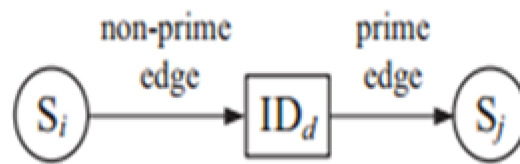


Fig 5. *Prime edge and non prime edge in BSG indicating S_i is waiting for S_j because of ID_d*

Each edge in the graph represents a transaction request that is accepted by a slave but is not completed. A prime edge is one from an ID vertex to a slave vertex and a nonprime edge is one from a slave to an ID. As shown in Fig. 5. A prime edge from vertex IDd to vertex Sj indicates that the transaction corresponding to the edge is requesting to access Sj and is the first accepted transaction among all currently accepted but not completed transactions with the tag IDd. A nonprime edge from vertexSi to vertex IDd indicates that the corresponding transaction is targeting Si but is not the first accepted transaction among the accepted but not completed transactions tagged with IDd. These definitions implies that at any time there can be at most one prime edge associated with an ID vertex, and each transaction corresponding to a nonprime edge must wait for the completion of the transaction corresponding to the prime edge with the same ID.

For a BSG, when a transaction with tag IDd is accepted by a slave Sj, a prime edge from vertex IDd to vertex Sj appears if there is no uncompleted transaction with the same IDd in the bus system. Otherwise, a nonprime edge from vertex Sj to IDdappears, which implies that the transaction must wait for some other transaction to complete. Using our waiting relation notation, in Fig. 5 we have the waiting relation Wdij =1. When one transaction corresponding to a prime edge is completed, the prime edge will disappear, and one of the nonprime edges (if any) will become a prime edge, i.e., the edge corresponding to the transaction that is accepted earliest among all the transactions that are waiting for the just completed transaction will become a prime one. Based on the definition of a BSG, we can see that a cycle in a BSG implies that a set of waiting relations exist in a circular way, which indicates that the bus system is in an unsafe state. Take the transactions in the bus system in Fig. 3 again as an example.

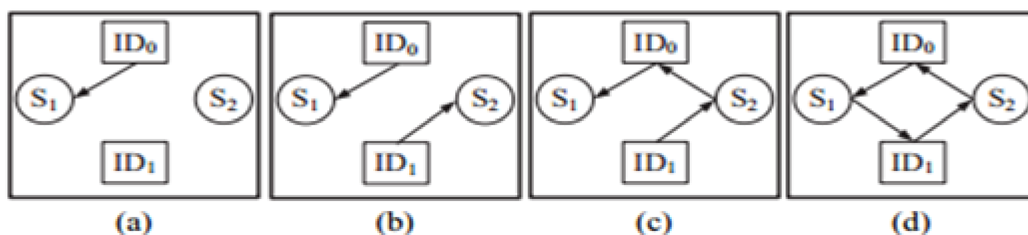


Fig 6. *Example of BSG indicating system is in unsafe state.*

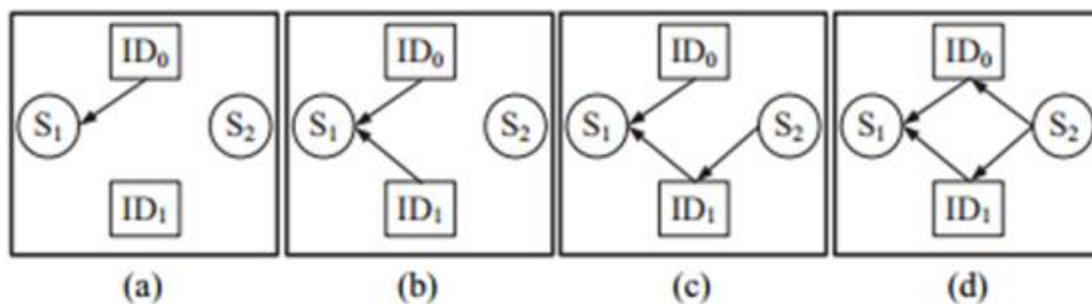


Fig 7. *Example of BSG indicating system is in safe state.*

The SOC system contains two slaves, so there are two vertices in the slave vertex set, namely S1 and S2. The IDs can be zero or one, so there are also two vertices in the ID vertex set, namely ID0and ID1 (respectively, refers to ID value zero and one). When the master in Fig. 3 first requestsT1, a prime edge from ID0to S1appears in the BSG as shown in Fig. 6(a). The master then requestsT2 which accesses S2 with ID1.A prime edge from ID1to S2 will appear afterT2is accepted by S2as shown in Fig. 6(b). When the master requestsT3which accesses S2 with ID0, there is one uncompleted transaction (T1) tagged with ID0. Thus Fig. 6(c) shows a nonprime edge from S2 to ID0 appears afterT3 is accepted by S2. Similarly, Fig. 6(d) shows that a nonprime edge fromS1 to ID1 will appear afterT4 is accepted by S1. In Fig. 6(d), a cycle exists in the BSG, and the bus system is in an unsafe state that may lead to a bus deadlock.

Now consider the case when the request orders of the transactions in Fig. 3 are $T1 \rightarrow T4 \rightarrow T2 \rightarrow T3$. As Fig. 7(a) and (b) shows, there is no uncompleted transactions tagged with ID0 or ID1 when $T1$ and $T4$ are accepted. Hence, two prime edges associated with $S1$ appear. Assume that when $T2$ and $T3$ are accepted, $T4$ and $T1$ are not completed yet. Then two nonprime edges will appear as shown in Fig. 7(c) and (d). In this case, no cycle exists in the BSG. Thus no matter how the responses from $S1$ and $S2$ are returned, no deadlock can occur.

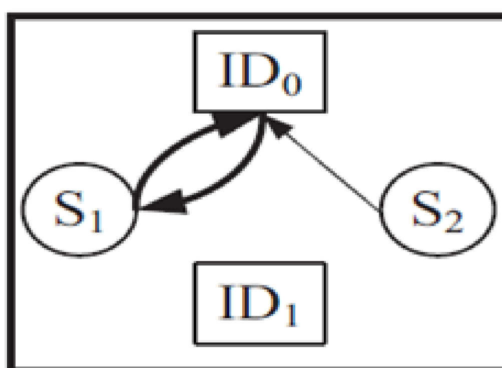


Fig 8. *A cycle composed of multiple edges does not imply that the bus system is in an unsafe state*

7. Bus Deadlock Solution

In this paper, we are using a technique to solve the bus deadlock problem based on BSG. Because no deadlock occurs if a bus system is always in a safe state, the proposed technique avoids deadlocks by stalling requests that will result in nontrivial cycles in the BSG. A similar request stalling concept to avoid deadlocks is also adopted in [9]. The main difference between our technique and those in [9] is that under the help of BSG we can stall much less requests but can still guarantee that no deadlock will occur.Before detailing our deadlock solution, we illustrate the techniques in [9] using BSGs to show how they avoid deadlocks.We will use the designs with two IDs and two slaves in the following description to illustrate the various deadlock avoidance schemes.

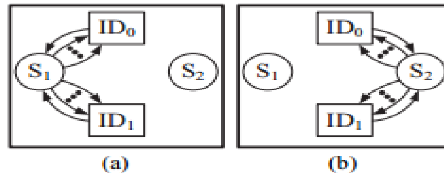


Fig. 9. Legal requests under single slave scheme.

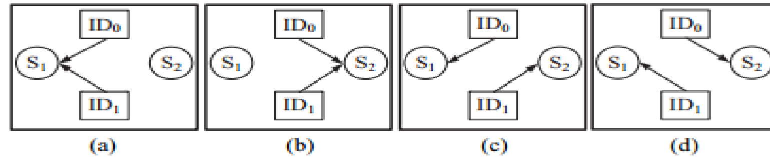


Fig. 10. Legal requests under unique ID scheme.

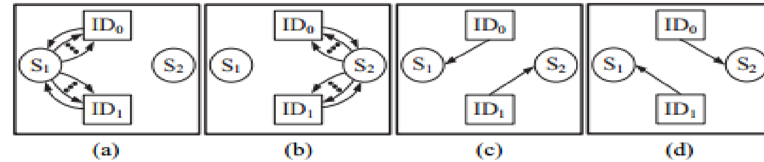


Fig. 11. Legal requests under hybrid scheme.

The single slave scheme [9] only allows tagged requests to access the same slave. Fig. 9(a) and (b) shows the legal requests under this scheme, i.e., the requests that will be forwarded to the slave but not stalled in the bus. Multiple edges between two vertices are allowed in this scheme. The unique ID scheme [9] only allows one tagged request for each ID. If there are k available IDs, at most k tagged requests can be accepted. Legal requests under the unique ID scheme are shown in Fig. 10(a)–(d). The hybrid scheme [9] combines the above two schemes and allows multiple requests to the same slave with the same ID in the unique ID basis, i.e., all requests tagged with the same ID must target the same slave. Legal requests under the hybrid scheme are shown in Fig. 11(a)–(d). From these graphs, we can see that many transactions that will not result in a deadlock cannot be forwarded to slaves. For example, in Fig. 11(a), a request to access S2 with ID0 or ID1 can still be issued and accepted by S2 without causing any deadlock. We next describe our deadlock solution. Our deadlock avoidance approach, called the deadlock avoidance by least stalling (DALS), stalls a request only if forwarding the request will form a nontrivial cycle in a BSG. Based on this idea, legal requests in a design with two IDs and two slaves under our DALS are shown in Fig. 12,

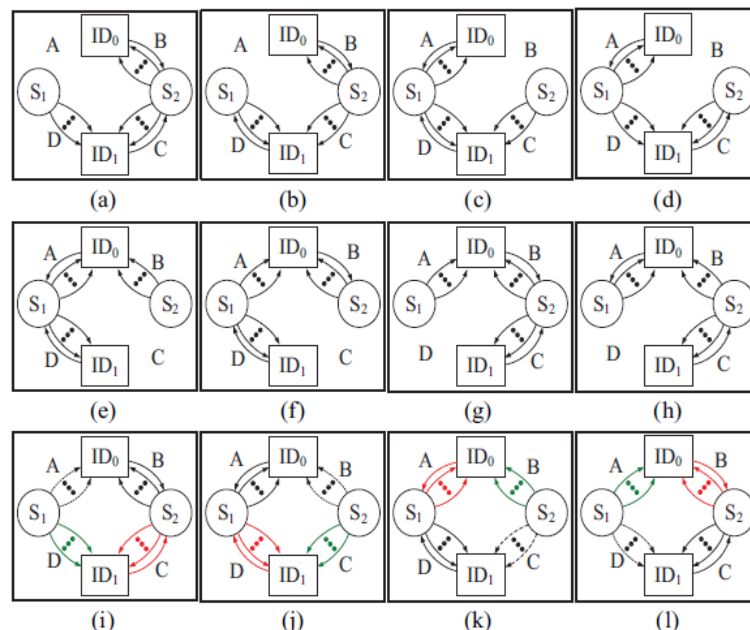


Fig. 12. Legal requests under our DALS.

Where we classify the edges corresponding to requests targetingS1 and S2with ID0into edge sets A and B, respectively, and classify those corresponding to requests targeting S2 andS1 with ID1 into edge sets C and D, respectively. If any of the four request sets is empty, no nontrivial cycle exists in the BSG. Fig. 12(a) and (b) shows the case of the edge set A being empty, where the first request tagged with ID1 targets S2 in Fig. 12(a) and the first request tagged with ID1 targets S1 in Fig. 12(b). In both cases, the first request tagged with ID0target S2.These figures indicates that any new request belonging to sets B, C, or D can be accepted with no risk of deadlock occurring. Fig. 12(c)–(h) shows the legal requests in the cases that edge sets B, C, and D are empty, respectively.

In addition, in some cases the proposed DALS also forwards requests in the fourth edge set. Consider Fig. 12(a) again. If a new request accessingS1 andtaggedwithID0 is accepted byS1, an edge in edge set A appears. This edge must be a nonprime edge fromS1to ID0because a prime edge associated with ID0already exists. Thus no counter-clockwise nontrivial cycle can occur in Fig. 12(a). Now if all the requests in C are accepted by S2 earlier than all the requests in D are accepted by S1, the edges in D must be nonprime edges before all requests in C are completed. Therefore, no clockwise nontrivial cycle can occur in Fig. 12(a) either. As a result, no deadlock can occur even if a new request in A is accepted. We show the results of this analysis in Fig. 12(i) where all requests in C are marked in red and all requests in D are marked in green with the order that all red edges are requested and accepted before all green edges. The dashed edges in A indicate the new requests that can be issued in this case. Similarly, as shown in Fig. 12(j)–(l), dashed requests in request sets B, C, and D can be, respectively, issued and accepted by slaves if the requests marked with red are accepted earlier than those marked with green. In our proposed DALS, we do not stall any dashed request in Fig. 12(i)–(l).

8. CONFIGURABLE ARBITER

The reconfigurable hybrid arbitration algorithms can be achieved by assigning different arbitration algorithms into functional blocks. For the evaluation of relative performances of different configuration states, the average waiting time, bus granted rate, throughput, and bus utilization can then be calculated.

To describe clearly the architecture of the proposed reconfigurable arbiter, we present the hardware architectures for fixed priority, round-robin, and random access, respectively. The functional block of the fixed priority is depicted in Fig.4.6. The request signals of Master 0 (M0), Master 1 (M1), Master 2 (M2), and Master 3 (M3) are assigned to BUSREQ0, BUSREQ0, BUSREQ0 and BUSREQ0, respectively. Then, the priority order is assigned to M0 > M1 > M2 > M3, where M0 owns the highest priority. The granted signals are Grant0, Grant1, Grant2, and Grant3 are for M0, M1, M2, and M3, respectively.

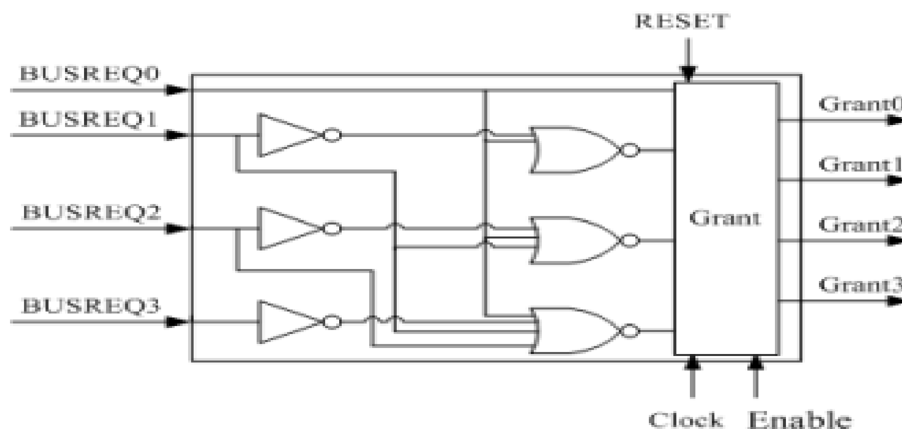


Fig13. *Fixed priority based arbitration*

The implementation of a round robin is shown in Fig. 4.7. A round-robin arbiter consists of a point controller, a timer, a de-multiplexer, and a grant register to memorize the granted master. The point controller will point to each master to check the bus request signal. If the master requests the use of the bus, the address of the master will be de-multiplexed and store at the grant register.

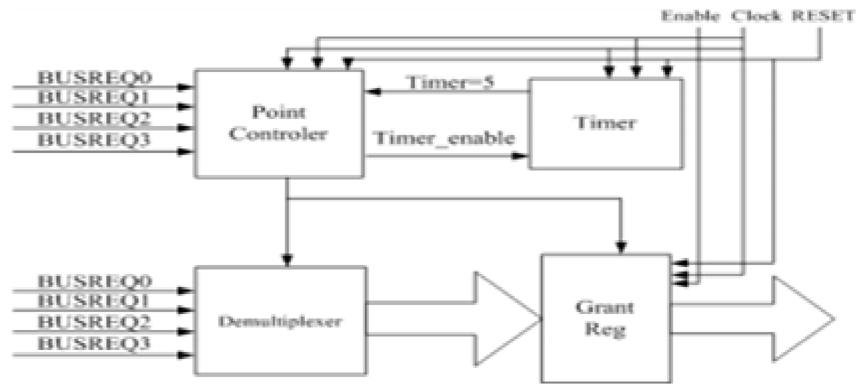
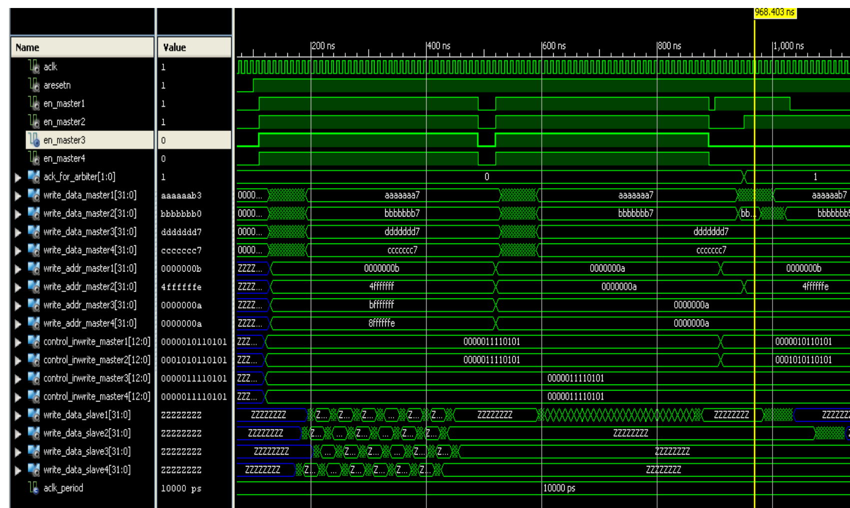


Fig14. *Round robbin arbitration*

9. RESULTS



10. CONCLUSION

Advanced communication protocols such as AXI can greatly improve the overall performance of a large VLSI system. However, the whole system may sink into a deadlock if designers do not carefully handle the advanced transactions. In this project, proposed a new graph model called the BSG to model the behavior of a bus system that can describe the necessary and sufficient condition for an unsafe state to occur. Then the proposed technique is used to solve the deadlock problem.The arbiter with hybrid arbitration algorithm for single layer shared bus system is used in this project. The simulation results not only provide performance analysis for the various combinations of the arbitration algorithms. These results can be feed into the reconfigurable arbiter to obtain the optimal condition under different system workloads. The reconfigurable arbiter can be custom-tuned to obtain high bandwidth utilization, low latency, and power effective for on-chip bus communication. Experimental results showed that our technique greatly outperforms previous work in both the numbers of stalled requests and the bus performance with affordable area overhead.

REFERENCES

[1] *Advanced Microcontroller Bus Architecture Specification*. (1997)[Online]. Available: http://www.arm.com

[2] *Open Core Protocol Specification*. (2006) [Online]. Available:http://www.ocpip.org/home

[3] A. T. Tran and B. M. Bass, "RoShaQ: High-performance on-chip routerwith shared queues," in *Proc. IEEE 29th Int. Conf. Comput. Design*,Oct. 2011, pp. 232–238.

[4] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism,"in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*,Feb. 2007, pp. 285–294.

[5] J. Pang, L. Yang, L. Shi, T. Zhang, D. Wang, and C. Hou, "A priorityexpression-based burst scheduling of memory reordering access," in*Proc. Int. Conf. Embedded Comput. Syst., Archit., Model.,Simul.*,Jul. 2008, pp. 203–209.

[6] X. Xiao and J. J. Lee, “A true O(1) parallel deadlock detection algorithmfor single-unit resource systems and its hardware implementation,” *IEEETrans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 4–19, Jan. 2010.

[7] A. Silberschatz, P. B. Galvin, and G. Gagen, *Operating System Concepts*,7th ed. New York, USA: Wiley, 1993.

[8] T. S. Cummins, “Method and apparatus for detecting a bus deadlock in an electronic system,” U.S. Patent 6 292 910, Sep. 18, 2001.

[9] *Technical Reference Manual of PrimeCell AXI Configurable Interconnect(PL300)*, ARM, Cambridge, U.K., 2010.

[10] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “The LOTTERYBUSon-chip communication architecture,” *IEEE Trans. Very LargeScale Integr. (VLSI) Syst.*, vol. 14, no. 6, pp. 596–608, Jun. 2006.

[11] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, “Dynamically configurable bus topologies for high-performance on-chip communication,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 10,pp. 1413–1426, Oct. 2008.

[12] Chin-Yao Chang, *Student Member, IEEE*, and Kuen-Jong Lee, *Member, IEEE ”On Dead Lock Problem of on-chip bus supporting out-of order transactions.*

AUTHORS’ BIOGRAPHY

Srikanth Manukonda is presently pursuing final semester M.Tech in VLSI Systems Design at Mallareddy Engineering College, Secunderabad. He received degree B.Tech in Electronics and communication from Bomma Institute of Technology and science. His areas of interest are VLSI systems, Digital design, Memory controllers and bus interface design.

Santosh J is presently working as an Assistant Professor in the department of Electronics and Communication Engineering, MREC, Secunderabad, India. He received degree B.Tech in Electronics and communication fromCVRCollegeOf Engineering, JNTU-H and M.Tech in VLSI SYSTEM DESIGN, SDES, JNTU-H. His areas of interest are VLSI systems design, Reconfigurable architecture and Design of Fault Tolerant systems.