Scientific Journal Impact Factor (SJIF) : 1.711
International Journal of Modern Trends in Engineering and Research
ISSN (ONLINE) : 2349 - 9745
ISSN (PRINT) : 2393 - 8161

ADVANCED INTERCONNECT PROTOCOLS USING HIGH-PERFORMANCE DEADLOCK-FREE ID ASSIGNMENT

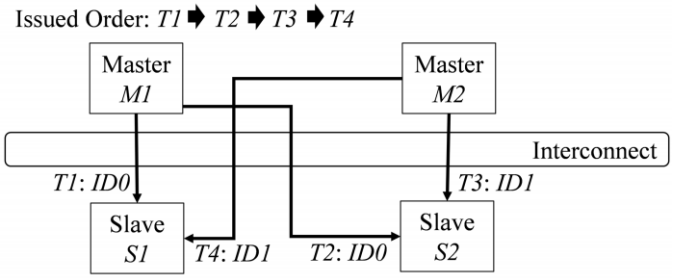
Sivasankari[1], Deepasri[2]
[1]*PG Scholar, VLSI Design, SNS College of Technology, Tamil Nadu, India*
[2]*Assistant professor, ECE Department, SNS College of Technology, Tamil Nadu ,India*

Abstract—In a latest network-on-chip design, several cores and intellectual properties can be integrated into a one chip. For high-performance interconnects, designers rapidly adopt advance interconnect protocols that support mechanisms of parallel accessing, including outstanding transactions and out-of-order complete of transactions. To produce these novel mechanisms, a master tag an ID to each transaction to decide in-order or out-of-order properties. These advanced protocols may lead to transaction deadlocks that do not occur in existing protocols. To prevent the deadlock problem, current solutions stall suspicious transactions and in some cases, many such stalls can incur serious performance penalty. Here, we propose a novel ID assignment mechanism that guarantee the issue transactions to be deadlock-free and results in rapid reduction in the number of transaction stalls issued by masters. Our experiment results show great performance improvements compared with last works with little hardware and power overheads.

Index Terms—Advanced interconnect protocol, deadlock, Network on chip (NoC)

I. INTRODUCTION

Advanced interconnect protocols allow the freedom to implement hardware architecture using shared link, crossbar, network on chip, or a hybrid of them. These protocols also support mechanisms of parallel access, including outstanding transactions and out-of-order complete transactions . In this way, a master can issue multiple transactions (such as write transactions) to slaves without waiting for outstanding (uncompleted) transactions to return. In addition, a slave can return the write responses or read data of outstanding transactions out of order, i.e., an earlier issued transaction by a master can be returned later and a later issued transaction can be returned earlier. The out-of-order mechanism allows a slave to perform more efficiently. For example, a memory slave can return transactions located in faster memory returned ID tags and a deadlock scenario. in order while transactions with different IDs can be returned out of order. Since the number of IDs impacts the hardware overhead of their out-of-order controller in an interconnect architecture, normally, the number of IDs is limited. Let us consider the example in Fig. 1



Masters *M1* and *M2* issue the following ordered sequence of transactions—*T1*, *T2*, *T3*, and *T4*—whose IDs are assigned to be *ID0*, *ID0*, *ID1*, and *ID1*, respectively. Since transactions *T1* and *T2* are issued from *M1* and tagged with the same ID, *T1* must be returned before *T2* and similarly transaction *T3* must be returned before *T4*.With the above out-of-order mechanism, interconnect

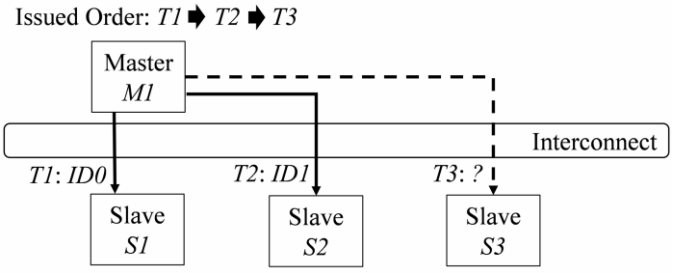
bandwidth and efficiency can be greatly improved; however, it may lead to the deadlock problem , which does not occur in traditional protocols supporting only in-order transactions. The conventional deadlock that occurs in traditional protocols is different from the ID deadlock that occurs in advanced bus protocols. In the conventional deadlock, the relation between masters and slaves is similar to that between processes and resources in an operating system where a deadlock occurs when there exists a circular wait-and-hold relation among a set of processes and resources. In advanced bus protocols, a deadlock occurs due to the out-of-order transactions that are tagged with IDs supported by these protocols .

According to the information provided by the industry, these kind of deadlocks occur very often in reality. For ease of understanding, we show an example of the ID deadlock scenario in Fig. 1.

We assume slave *S2* executes transactions in order, and thus transaction *T3* needs to wait until *T2* is returned. We also assume *S1* supports out-of-order execution and *S1* executes *T4* before *T1*. As a result, *T1* needs to wait until *T4* is returned. Moreover, since *T4* and *T3* are tagged with the same ID (*ID1*), *T4* needs to wait until *T3* is returned. Similarly because of the same ID (*ID0*), *T2* needs to wait until *T1* is returned. A deadlock occurs because each of all four transactions waits for the return of a certain transaction and the wait status forms a cycle *T1(S1)* _ *T4(S1)* _ *T3(S2)* _ *T2(S2)* _ *T1(S1)*,where *Ti*(*Sj*) means that transaction *Ti* is received by slave *Sj*. In addition, the symbol _ represents a wait condition, and there are two different wait conditions. When *T1* waits for *T4*, *T1* is waiting to be executed by the same slave. On the other hand, even when transaction *T4* is executed, *T4* still needs to wait until *T3* is returned, due to using the same ID. In this case, *T4* is waiting to be returned due to using the same ID. A deadlock cycle can occur because of the joint effect of the same device problem and the same ID problem. The above example shows that a deadlock may occur not only within one master but also among several masters. Detecting a deadlock within a master or among masters can be very costly because it may require communication from most other masters. As a result, all previous works implement their deadlock-avoidance methods in the interconnect controller, which can easily receive information from all masters. For example, ARM cyclic dependence avoidance schemes (CDAS) implements three ID checking schemes to prevent a deadlock in the interconnect controller.

II. DEADLOCK-FREE ID ASSIGNMENT

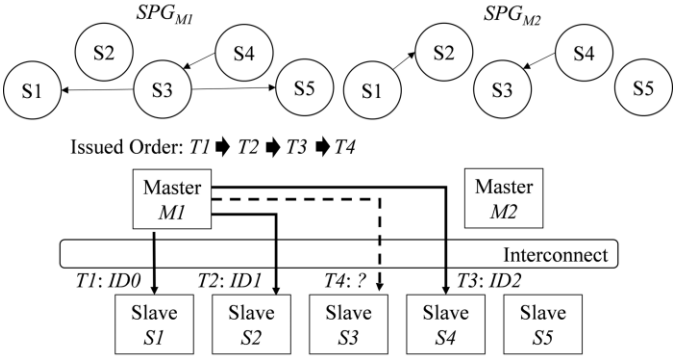
In this section, we discuss how to determine an ID for each transaction, while guaranteeing not to cause a deadlock cycle .An SOC design usually consists of several masters and slaves. Our assignment design will be embedded into each master. We will later show that as long as ID assignment of all masters follows certain rules, no deadlock will occur. We first describe two rules ,Rule 1 and Rule 2. Between every pair of slaves, our ID assignment must follow either Rule 1 or Rule 2.Before describing the rules, we assume that several transactions, whose IDs were assigned, have been issued but not returned (finished), and each unfinished transaction is called an outstanding transaction. In addition, we are interested in assigning ID for a new incoming transaction from a master. The first rule is described as follows.



Rule 1 (Exclusive Rule): Slaves *Si* and *Sj* are (mutually) exclusive if the ID assigned to a new transaction sent to *Si* (*Sj*) cannot be the same as IDs already assigned to outstanding transactions received by *Sj* (*Si*). For example, Fig. 2 shows an ID assignment example following the exclusive rule between slaves *S1* and *S2*. We assume that master *M1* can tag either *ID0* or *ID1* to a new

transaction. We can only assign a valid ID, *ID0*, to transaction *T3*, because *T2*(*S2*) has been tagged with *ID1*.The exclusive rule is strict and may cause stalls in certain situations. For example, in Fig. 3, when master *M1* issues transaction *T3*, *T3* cannot be assigned a valid ID and is stalled because all IDs, *ID0* and *ID1*, have been used by *T1*(*S1*) and *T2*(*S2*), respectively. To avoid stalls, we introduce the second rule called the priority rule.

Rule 2 (Priority Rule): Slave *Si* is prioritized over *Sj* or *Si* → *Sj* if *Si*'s new transaction can reuse one of the IDs of *Sj*'s outstanding transactions .Consider the example in Fig. 3 again, where *S3* is prioritized over *S2*. As a result, master *M1* can assign a valid ID (*ID1*) to new transaction *T3* though *ID1* has been used by *T2*(*S2*). Without adding the priority *S3* → *S2*, slave *S3* needs to wait for an available ID and the master has to be stalled. As a result of adding the priority of *S3* → *S2*, we reduce one stall when *T3* arrives. However, since the same ID is assigned to *T3* and *T2*, *T3* must wait for the return of *T2*. Notice that the priority rules are not transitive among slaves. For example, if *S1* is prioritized over *S2* and *S2* is prioritized over *S3*,it does not indicate *S1* is prioritized over *S3*.Our ID assignment follows either one of both rules between every pair of two slaves, and we can represent such conditions in a graph. These two rules can be represented as a directed graph called SPG *G* = *_V*, *E_*, where each vertex in *V* is a slave, and each directed edge from vertex *vi* to *vj* in *E* indicates *vi* is prioritized over *vj* otherwise, if there is no edge between *vi* and *vj*, *vi* and *vj* are exclusive. Fig. 4 shows an SPG with three slaves *S1*, *S2*, and *S3*.The edge from slave *S3* to *S2* means *S3* is prioritized over *S2*.In addition, since there is no edge between *S1* and *S2*, *S1* and *S2* are exclusive. Similarly, *S1* and *S3* are exclusive. However, the ID assignment guided by an arbitrarily constructed SPG does not guarantee that it will be deadlock-free. Luckily, we have developed the following two theorems.



Theorem 1: If all masters follow the same SPG and the SPG is acyclic, no deadlock will occur. An SoC design may contain heterogeneous masters (e.g., CPU, Graphics Processing Unit, and DSP) which have different behaviors .To improve the efficiency for each master, we should construct different SPGs for different masters. However, even though each SPG is acyclic, a deadlock may still occur. Still, we can use the following theorem.

Definition 1: A union graph *U* of two SPGs *G1* = *_V1*, *E1_* and *G2* = *_V2*, *E2_* is defined as *U* = *_V1* ∪ *V2*, *E1* ∪ *E2_*.*Theorem 2:* If the union graph of all masters' SPGs is acyclic, no deadlock will occur. For each master, the SPG it follows to assign ID is a sub graph of the union graph according to Definition 1. This indicates that all masters follow the union graph, which is acyclic. According to Theorem 1, no deadlock will occur. For example, Fig. 5(a) shows two SPGs, SPG*M*1 and SPG*M*2, for masters *M1* and *M2*, respectively. From Theorem 2, though the ID assignments for *M1* and *M2* use different SPGs, no deadlock will occur because the union graph of SPG*M*1 and SPG*M*2 shown in Fig. 5(b) is acyclic. Given an SPG, let us consider assigning an ID for slave *Si*'s new transaction. To satisfy the above two rules, it can be shown that we only need to find unused IDs for slaves that are exclusive to *Si*. Consider the example in Fig. 5(a). We assume that four IDs—*ID0*, *ID1*, *ID2*, *ID3*—are available. Further, transactions *T1*(*S1*), *T2*(*S2*), and *T3*(*S4*) are outstanding with *ID0*, *ID1*, and *ID2*, respectively. Let us consider a new transaction *T4* issued from master *M1* to *M1* and *M2*. (b) The union graph of all SPGs.ID assignment must guarantee that the rules between *S3* and other

slaves are satisfied. From SPG*M*1 in Fig. 5(a), we can find that *S3* is prioritized over *S1* and *S5*; i.e., *T4* can reuse IDs used by transactions related to *S1* and *S5*. For other slaves *S2* and *S4*, *T4* cannot reuse their IDs because *S3* and *S2* are exclusive and *S4* is prioritized over *S3*. Since *T2*(*S2*) and *T3*(*S4*) have already used two IDs, *ID1* and *ID2* as a result, *T4* can only use *ID0* or *ID3*. If *T4* is tagged with *ID0*; i.e., *T4* takes over *T1*'s ID (*ID0*), *T1* still needs to be returned before *T4*. After both transactions are finished, *ID0* is released for use by others. Since transactions to *S3* can always take over IDs used by *S1* (if there are no other choices), transactions to *S1* are stalled frequently. The proposed ID assignment method following the priority rule does not cause starvation. This is because the issuing order of transactions by masters cannot be changed. If a transaction *T* is stalled and cannot be issued by master *M*, none of the following transactions created by *M* can bypass *T* to be issued, even though their destination slaves are prioritized over the slaves of the previous transactions. This guarantees that outstanding transactions can eventually be executed, and *T* will get its turn to be executed.

III. SLAVE PRIORITY GRAPH CONSTRUCTION

In general, if slave *Si* is prioritized over *Sj*, transactions sent to slave *Si* have more flexibility in ID assignment and thus slave *Si* has better chance of not becoming stalled. On the other hand, slave *Sj*, which has lower priority, may suffer from having fewer valid IDs. Therefore, if an application has no special characteristics regarding its slaves, using SPGs may not result in additional benefit. In other words, to construct an efficient SPG, we need information about characteristics of slaves, such as access frequency, execution latency, and access order among slaves .The most efficient way to obtain the design knowledge is to perform profiling. Therefore, we discuss how to extract slave characteristics and construct initial SPGs by running application simulations. The construction procedure consists of two steps. In the first step, for each master we build an initial SPG which is a weighted graph summarizing the stall information during simulations. Then, with the initial SPGs of all masters, we remove some of their edges so that their union graph is acyclic with maximum weight.

A. Initial SPG

We first assume that the SPG for a master does not have any priority edge. In other words, all pairs of slaves under the master adopt the exclusive rule. Then, we simulate certain representative software test sets of an application. The test sets should be provided by designers. During the simulation, we record all stalling scenarios. When a new transaction sent to *Si* is stalled due to an outstanding transaction received by *Sj*, we increment the weight on the priority edge $Si \rightarrow Sj$. If edge $Si \rightarrow Sj$ does not exist, we insert edge $Si \rightarrow Sj$ and then set the edge weight depending on the number of IDs assigned to *Sj* and the number of stalled cycles due to waiting by *Si* [6]. After running the application simulation, we can obtain the initial SPG for each master with weights on its edges.

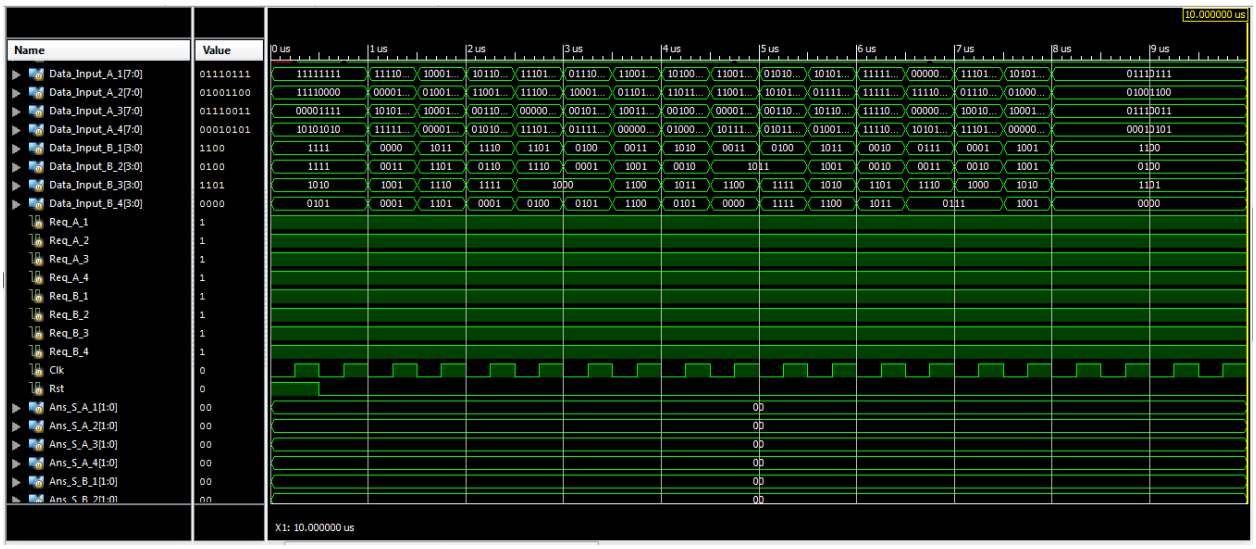
B. Acyclic SPG

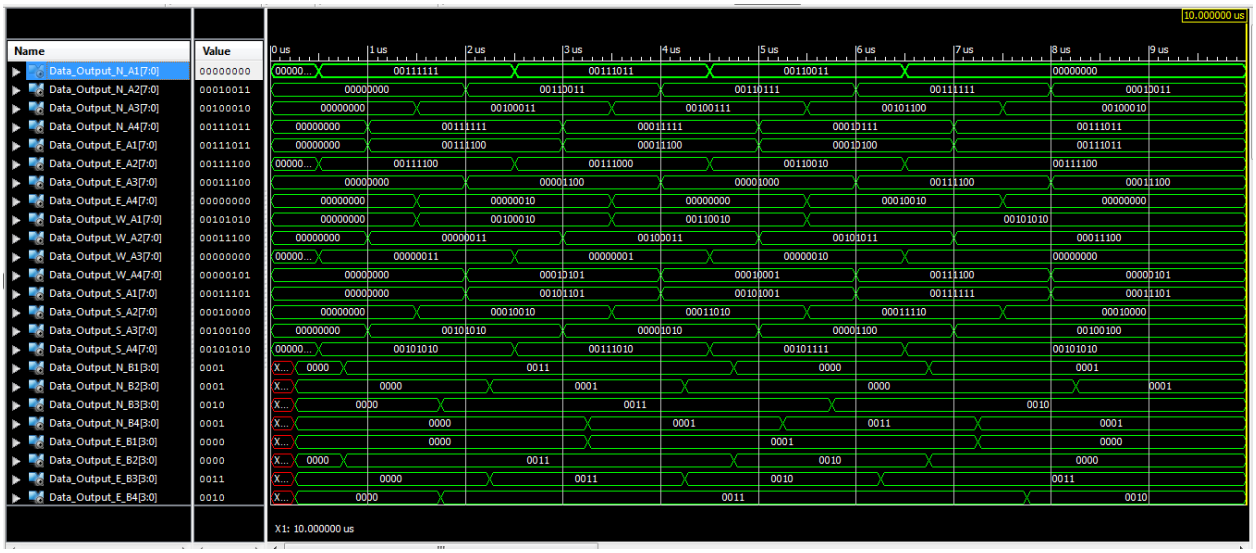
An initial SPG may contain many edges and may have cycles. Therefore, we try to remove edges to make each SPG acyclic. Since a weight reflects the ability to prevent stall cycles, we attempt to achieve an acyclic SPG with the maximum weight. Our problem formulation is described as follows.

Problem Formulation 1*:* Given an initial SPG, remove edges to create an acyclic sub graph with the maximum weight. The problem is known as the maximum acyclic sub graph problem. The maximum acyclic sub graph problem is NP-complete [2], and we adopt the algorithm in [2] to construct the SPG with maximum weight. As mentioned in Section II, masters can have different SPGs. To avoid deadlock for the entire system, we need to make sure the union graph of all SPGs has no cycle. To reduce the maximum number of stall cycles in a system, we revise our algorithm as follows. First, we create the union graph of all SPGs. The weight of each edge in the union graph is accumulated from

all SPGs. Then, we perform the algorithm in [2] to make the union SPG acyclic. Finally, we create the SPG of each master by removing the edges that do not belong to the master's SPG. As a result, we obtain an acyclic SPG for each master.If we let slave *Si* be prioritized over another slave *Sj*, we may prevent transactions sent to *Si* from being stalled. However, later transactions sent to *Sj* might be stalled. To resolve this problem, we allow a priority rule in an SPG to be dynamically changed to an exclusive rule. That is, for each edge *Si* → *Sj*, we can remove the edge to make *Si* and *Sj* exclusive. Later, if necessary, we can let *Si* be prioritized over *Sj* again by adding the edge back. Given an SPG, our dynamic SPG works as follows. Basically, we allow each priority edge in a given SPG to be present or not present. Initially, all pairs of slaves start using the exclusive rule. Whenever a transaction sent to slave *Si* is stalled, we change the exclusive rule to become the priority rule of *Si* → *Sj* if *Si* can be prioritized over a slave *Sj*. Then, we switch the priority rule back to the exclusive rule either when the transaction sent to *Si* is returned, or when the number of stalled transactions in *Sj* exceeds a given threshold number. Our experimental results show that the hardware overhead for inserting dynamic adjustment is marginal.

IV. EXPERIMENTAL RESULTS





V. CONCLUSION

Here, we present a deadlock-free ID assignment approach for advance interconnect architectures. Instead of using the existing check-and-stall approach, this proposed approach looks for the best ID assignment for each transaction to not only avoid deadlock but also rapidly reduce the number of stalled transaction. Two rules are first identified to break deadlock cycles caused by the joint waiting-for-execution and waiting-for-return effects. Equipped with both rules, SPGs constructed by guidelines or by simulation learning are used to lead the ID assignment process for each transaction. Efficient architecture and hardware are designed to support the proposed ID assignment approach, with fixed SPGs or adaptable (to working environments) SPGs, using little hardware and power overhead. Experimental results demonstrate that the feasibility of the proposed approach in that deadlock is completely avoided, and the number of stalled transactions is dramatically reduced, when compared with the traditional methods.

REFERENCE

[1] C.-Y. Chang and K.-J. Lee, "On deadlock problem of on-chip buses supporting out-of-order transactions," *IEEE Trans. Very Large Scale Integret. (VLSI) Syst.*, vol. 22, no. 3, pp. 484–496, Mar. 2014.

[2] (2013). *x264 Free Software Library*. [Online]. Available: http://www. videolan.org/developers/x264.html

[3] (2012). *Pin—A Dynamic Binary Instrumentation Tool*. [Online]. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentationtool
[4] M. R. Guthaus, J. S. Ringenberg ,. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. Int. Workshop Workload Characterizat.*, 2001,pp. 3–14.
[5] *LogiCORE AXI Interconnect IP*, Xilinx, San Jose, CA, USA, 2010.
[6] (2014). *On-Line Report*. [Online]. Available: https://sites.google.com/ site/nthuproject2014/
[7] (2009). *Open Core Protocol Specification*. [Online]. Available:http://www.ocpip.org
[8] (2012). *Pin—A Dynamic Binary Instrumentation Tool*. [Online]. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentationtool
[9] *Technical Reference Manual of CoreLink NIC-400 Network Interconnect*,ARM, Cambridge, U.K., 2012.
[10] (2013). *x264 Free Software Library*. [Online]. Available: http://www. videolan.org/developers/x264.html
[11] A. Silberschatz, P. B. Galvin, and G. Gagen, *Operating System Concepts*, 7th ed. New York, NY, USA: Wiley, 2009.

