2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems

Interfacing Synchronous and Asynchronous Domains for Open Core Protocol

Vikas S. Vij[1], Raghu Prasad Gudla[2], Kenneth S. Stevens[1]
[1]University of Utah, [2]Intel Corporation

***Abstract*—Intellectual property (IP) blocks are connected in a system on chip using a bus or network-on-chip (NoC). IP reuse is facilitated by the modularity that results when using common interfaces between the IP cores and the bus or NoC. This paper investigates and implements several versions of one of the common interfaces, the open core protocol (OCP). The paper addresses two new aspects of interface design. First, an approach is developed to partition the common protocol portion of the interface from the interface back-end which is specific to the particular IP. This is achieved with a component we call a *domain interface* at this boundary. Second, the domain interface is enhanced to synchronize between IP blocks and busses that use different clock frequencies or asynchronous (unclocked) logic. As a result IP operating at unrelated frequency and fully asynchronous (unclocked) blocks can more easily be integrated into a system. Results are reported for power, performance and area for these clocked and asynchronous implementations.**

I. Introduction

The complexity of current integrated circuits have resulted in a system-on-chip (SoC) revolution. Time to market, the need to limit design engineering costs, as well as other factors have resulted in the need for modularity and design reuse. Various system building blocks, called intellectual property (IP) blocks, are designed once and then used in multiple different systems. System-on-chip designs contain IP blocks like memory, general purpose processors, communication blocks such as busses or network-on-chip (NoC), and other specific function units and coprocessors [1].

There are a number of technical challenges to creating modular IP blocks that can be reliably and rapidly interconnected to form SoC designs that span multiple applications and process technology nodes. The size and diversity of operation of the function blocks results in many of the IP blocks requiring independent operating frequencies. Traditional clocked methodologies encourage a single operating frequency for the entire chip. Integrating many IP blocks into an SoC is efficiently performed by interconnecting the design blocks with a bus or network-on-chip. Due to design complexity and wire latencies, modern bus and NoC interfaces result in nondeterministic response delays. This often requires a complex interface to clocked systems which traditionally require that events occur on specific clock cycles.

Several common bus interface protocols have been created to enhance modularity and composability of IP blocks. These include protocols such as the the advanced microcontroller bus architecture (AMBA), wishbone, and open core protocol (OCP). If IP blocks and a bus (or NoC) have been enhanced to support such a protocol, then they can be directly connected to communicate across the bus (or NoC). Such designs can be directly reused in any system that supports the protocol. This paper implements a large subset of the full OCP protocol. The implementation is enhanced to facilitate multiple timing methodologies, simplify buffering, and to reduce the integration effort required to make an IP block OCP compliant.

This work assumes complete timing generality, including the implementation of a globally asynchronous locally synchronous (GALS) system, as well as the use of asynchronous IP blocks. The choice of timing and functionality of each IP block in an SoC needs to be carefully considered based on its specific power and performance targets. Without a modular design integration methodology this can lead to increased redesign effort, particularly for clockless asynchronous design or a new GALS architecture. Thus this method enables the use of clocked or asynchronous IP blocks, including asynchronous NoC designs.

The contributions of this work are as follows. The concept of a domain interface (DI) is introduced. The domain interface partitions the OCP protocol into two separate sections – a front end that implements the OCP master or slave protocol, and a back-end that interfaces with the specific IP block. The domain interface also serves as a location for interfacing different timing domains as well as providing buffering if necessary between the back-end and the OCP interface. The regions for clock domain crossing now become well defined and constrained in such a way that eases the design of systems with unrelated clock frequencies, asynchronous (unclocked) modules, or both. Given such a design, modularity is increased as the OCP master and slave front end blocks can be reused across all designs, including the network-on-chip. The design of specific IP back-ends is simplified. Both clocked and asynchronous OCP front ends have been designed. Domain interfaces with clock domain crossing and clocked to asynchronous interfaces have been implemented, as well as clocked and asynchronous front and back-ends.

II. Background

An overview of the OCP protocol and asynchronous bundled data design is provided.

A. Open Core Protocol (OCP)

Open core protocol (OCP) is a non-proprietary, open standard, core-centric protocol addressing IP core system-level integration requirements [2]. It is defined as a clocked system with unidirectional data transfer which assists in simplified core implementation, integration and timing analysis. OCP enables the design of IP cores independent of the other cores, thus enabling the reuse of IP. The goal of this protocol is to enhance the modularity and composability of the IPs without requiring redesign.

Fig. 1 shows a block diagram of the basic OCP implementation between a master IP core and a slave IP core communicating across an NoC. The IP cores have a OCP master/slave component directly integrated into their design.

The basic OCP implementation consists of two channels, a request channel and a response channel, as shown in Fig. 2. Any read command issued by the master IP core results in a

1063-9667/14 $31.00 © 2014 IEEE
DOI 10.1109/VLSID.2014.55

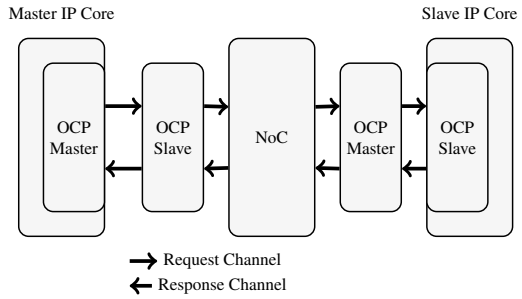

Fig. 1. OCP Implementation Block Diagram with native OCP Master and Slave

transfer on the request channel including the address associated with the read command. The slave IP core responds with data on the response channel for the master IP core. Similarly, write commands are issued across the request channel with the associated data and address. Various extensions are added to different versions of the OCP protocol such as the transfer of a burst of data, out-of-order responses, data handshake extensions, test control extension and a few more.

Some additional details of the OCP handshake protocol are introduced to help understand how traditional asynchronous handshakes can be mapped onto OCP[1]. The write command generated by the master IP core results in the OCP master interface defining a write command on the MCmd line, along with the Data and the Address information. The OCP slave block acknowledges the transaction by asserting the SCmdAccept signal on the request channel. Similarly for the read command, the master IP core sets the MCmd line to a read command and asserts the Address. The OCP slave acknowledges this read command with SCmdAccept, thus completing the handshake with the NoC slave interface. For a normal read, the OCP master then waits for an acknowledgment on the response channel before initiating a new command. If out-of-order reads are employed, the OCP master does not need to wait for responses and may immediately send the next command.

The basic handshake protocol for any request on the OCP channel involves an operation wherein the OCP master sets the MCmd wires on a request which is acknowledged by the SCmdAccept from the OCP slave. Similarly, there are other optional handshake signals that are asserted based on which OCP extensions being employed. For example, if the write data handshake extension is enabled then an extra SDataAccept handshake signal is used to indicate an acceptance of Data by the OCP slave. Similarly for the response channel, there is an optional MRespAccept signal generated by the OCP master for any response acknowledgment. Thus, flow control is implemented using OCP handshake protocols.

The asynchronous designs replace the clock by embedding request / acknowledgment handshake signals across the OCP master-slave interfaces. The OCP signals are considered bundled data signals in relation to the request. The SCmdAccept and MRespAccept signals can be entirely replaced with the asynchronous acknowledgment signal.

B. Bundled Data Asynchronous Circuits

A general 4-stage "bundled data" asynchronous linear pipeline design is shown in Fig. 3. It consists of a combinational logic datapath just as in a traditional clocked design,

[1]Refer to the OCP manuals for full protocol information [2].

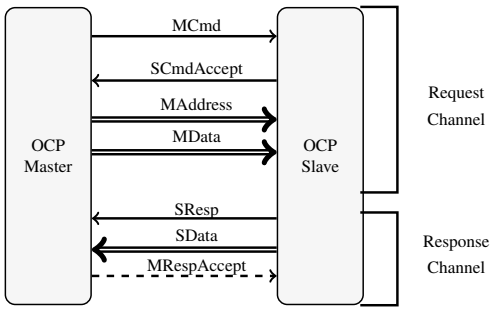

Fig. 2. Basic OCP Master and Slave Interface

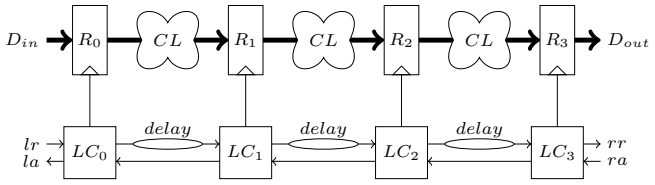

Fig. 3. Bundled data linear pipeline circuit with combinational logic datapath (Here lr and la are the request and acknowledge signal on left hand channel of a LC block and similarly rr and ra handshake on the right hand channel of a LC block.)

and an asynchronous control path that controls sequencing and pipeline frequency. The control path consists of linear control blocks (LC), which perform via handshaking the role of the clock. The datapath contains combinational logic (CL) and Registers (R). Handshake clocking generates the appropriate timing and sequencing for the design. It is elastic in nature and can stall if required. The latency through the control logic must be greater than the maximum delay of the combinational logic to fulfill the setup and hold time constraints at the register bank. Thus delay elements may be required between LC blocks.

The LC blocks are sequential circuits that implement an asynchronous handshake protocol. The LC design used for this paper is shown in Fig. 5, although any implementation may be used. Timing and sizing optimizations on these asynchronous circuits is performed using clocked CAD tools by constraining them using a flow based on Relative Timing [3].

III. Designs

A subset of the OCP protocol was implemented from the ground up to first develop a base synchronous implementation. The OCP subset used implemented normal read and write with extensions for burst mode as well as tags which enable out-of-order responses. Hence there are four modes that are possible for this OCP implementation which include normal read/write, burst read/write, normal read/write with out-of-order responses and burst read/write with out-of-order responses. By default, we have enabled the data byte extension, which allows enabling the data path in 8-bit segments. All these modes and selections are parameterized and can be changed easily during synthesis of the circuits.

Fig. 1 shows a block level implementation showing the communication network between a Master and a Slave IP with a OCP Master, OCP Slave and NoC. This standard implementation requires the OCP Master and Slave to integrate the OCP protocol into the IP and does not offer clear regions

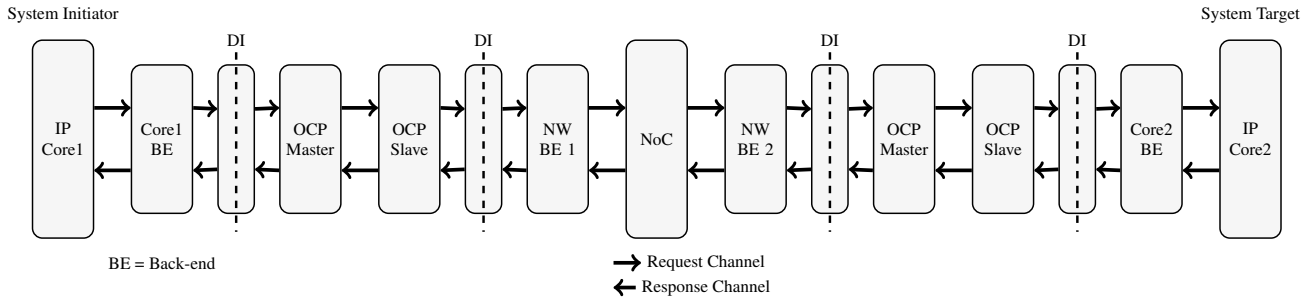

Fig. 4. OCP Implementation Block Diagram with Domain Interface (DI) and Back-end (BE) modules

for clock domain crossing (CDC). Hence, changes to an IP block can lead to redesign of others, if the CDC is moved into an adjacent block. To improve modularity, a *Domain Interface* (DI) is introduced. This block contains a simplified handshake protocol and confines clock domain crossings to this block. It separates OCP design integration into a custom back-end specific to the IP block, and an OCP master and slave block. This results in full reuse of a single OCP master and slave block for all IP designs. The DI also provides a good location for synchronization and buffering, when needed.

Fig. 4 shows the explicit partition created by the domain interface. Back-end modules interface between the cores or NoCs and the domain interface. This architecture allows the back-end (BE) blocks to take care of the interface between the DI and the IP operation without worrying about clock domain crossing. Thus back-end modules can be directly integrated into the IP core if advantageous. The DI is where frequency domains are synchronized. Any blocks, including IP cores and NoC, can be now implemented independently and then interfaced with this or any similar interface using a back-end module. The correct clock domain crossings and buffering is implemented directly into the DI.

A. Synchronous and Asynchronous OCP Implementation

The OCP protocol implements a handshake between the OCP master and OCP slave blocks, which prevents overwriting of the data and for flow control. Response times can arbitrarily vary, e.g. while communicating across a NoC due to traffic and congestion. Therefore, a stall capability needs to be implemented into the OCP to IP back-end to enable pausing of the cores to prevent overwriting of data when required by the OCP interface. The stall operations may require data buffering. This can result in significant complexity for traditional clocked IP blocks. Alternatively, asynchronous handshake modules natively allow for arbitrary stalls. Therefore the design of asynchronous back-ends are normally much easier to build, are smaller, and lower power than clocked back-ends. Further, certain OCP handshake signals, such as SCmdAccept are directly implemented with the asynchronous handshake signals and, thus, can be removed from the asynchronous implementation.

The initial clocked OCP master, OCP slave, and domain interface designs were converted into fully asynchronous designs. This resulted in an asynchronous implementation similar to the synchronous designs. These independent clocked and asynchronous blocks resulted in various configurations which can be evaluated for power, performance, area and flexibility of choosing the IPs as well as the NoC. This enables

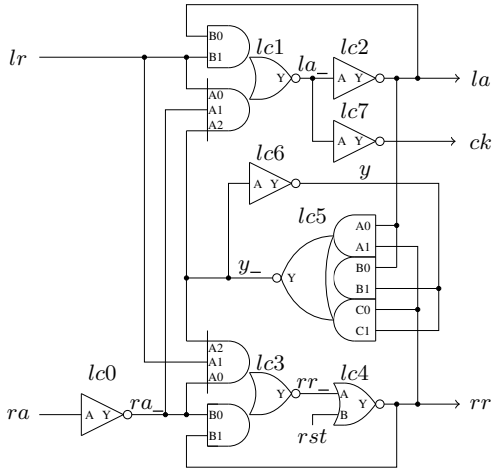

Fig. 5. LC circuit implementation

using either a synchronous or asynchronous component at any side of the DI and hence opening up many architectural options.

There are five different implementations which are investigated in this paper. First is a purely synchronous design with a single clock frequency controlling the operation of the IPs and the NoCs. Second is also a purely synchronous design but with IPs and NoCs operating at different frequencies. Third design is an unclocked asynchronous design. The fourth design has IPs which are synchronous while the OCP implementation and the NoCs are unclocked. The last design has a synchronous OCP implementation with the NoC operating at a fixed frequency combined with asynchronous IP blocks.

B. Domain Interface Designs

Five different domain interface designs are required to implement the five different implementations evaluated in this paper. The DI for the purely synchronous and asynchronous design is just combinational. (If needed, FIFO buffering can be added at the DI boundaries in these cases.) For the synchronous design, the DI steers the data forward without storing it. In case of the asynchronous design, the DI also needs to forward the request and acknowledge signal with the datapath signals. The complexity of the DI increases for the three other cases where different timing domains interact.

The domain interface when clock domain crossing occurs is defined in terms of an OCP request and response channel

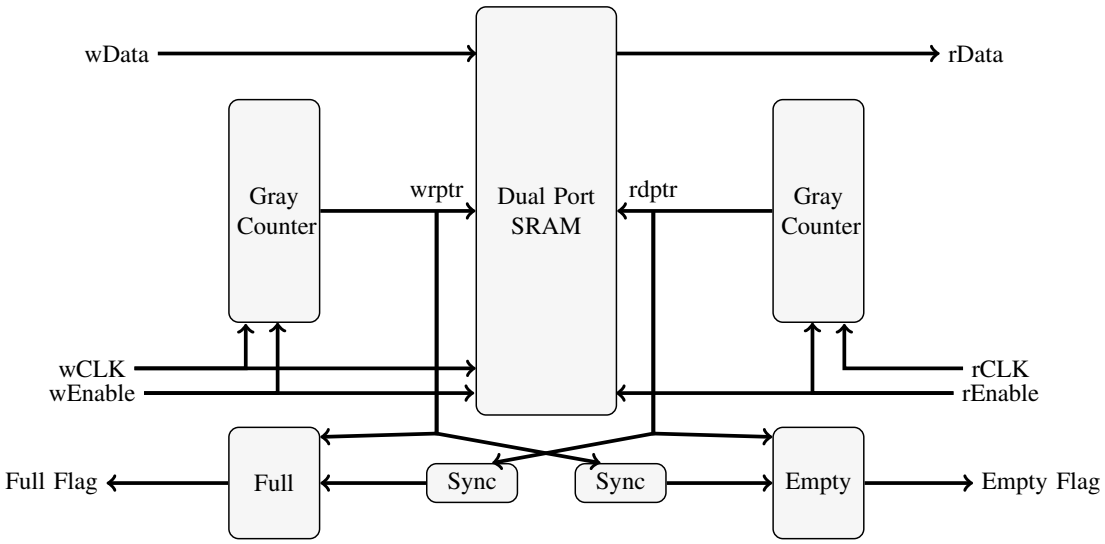

Fig. 6. *sync-sync* Domain Interface FIFO

(Fig. 2). Hence for every DI implementation there are two places where clock domain crossing can occur, once for the request channel and the other for the response channel. We use *sync* for synchronous or clocked interface and *async* for the asynchronous or unclocked interface. Thus synchronization uses the naming convention of a timing domain pair to identify the two synchronized domains.

There are three different cases for the DI design where different timing domains interact. They are the *sync-sync*, *async-sync* and *sync-async* domain interfaces. The *sync-async* DI requires a clocked to asynchronous request channel synchronization and a asynchronous to clocked response channel synchronization. The *async-sync* is its dual, having the clocked and asynchronous interfaces reversed.

Timing domain crossings which occur in the Domain Interfaces are now confined to three different designs: the *sync-sync*, *sync-async* and *async-sync* DI interfaces.

1) sync-sync DI FIFO: The core of this interface is a head and tail pointer FIFO that interfaces between two different clock domains. Fig. 6 shows an implementation of this design which is similar to the one described in [4], [5]. The benefit of this design is the easy synchronization and the generation of the full and empty status information before the arrival of the next rising edge of its domain CLK signal. Latency through this FIFO is dependent on the synchronizers which sample the empty and full status based on the read pointer (rdptr) and write pointer (wrptr). Reading and writing the FIFO can be done at each clock edge of their respective domain until the Empty or Full flag is set. A two flop synchronizer is used for the Sync block in Fig. 6. Therefore, it takes approximately two clock cycles after the next write or read to update the Empty or Full flag status respectively.

The domain interfaces specify Stall and Valid signals in order to indicate data validity (Stall and Valid signals are not shown but can be directly derived). Their behavior is similar to elastic systems [6]. The Stall signal on the write port is derived from the FIFO Full flag and the write Valid signal. Similarly, the Valid signal on the DI read port is derived from the FIFO Empty flag and the Stall signal on the read port. The wEnable signal is used to control the writing in the FIFO and the incrementing of the gray counter. It is derived from the write port Valid and stall signals. The rEnable signal is similarly derived from the Empty flag and the read port Stall signal.

2) async-sync DI FIFO: The FIFO for the *async-sync* FIFO is illustrated in Fig. 7. The clocked read port in this design is identical to that of the *sync-sync* FIFO. The write domain interface Valid signal is directly mapped to the asynchronous request signal (lr), and the Stall signal is directly mapped to the acknowledge (la) signal. (The Valid and Stall signals are not shown but can be directly derived.) The FIFO write port has been modified to generate the write clock wCLK from the write port Valid (lr) signal, and to directly generate the Stall (la) handshake signal.

The FIFO Full flag is causally generated from the assertion of the lr signal. A relative timing constraint is required to ensure proper operation. For this design to work correctly, the minimum delay between adjacent rising edges of the lr signal must be greater than the maximum delay from the rising of the lr signal to the assertion of the Full flag. The Full flag update time is the sum of the delay to shift the gray counter, which updates the write pointer wrptr and propagates its value through the Full flag logic.

No synchronization is required from the clocked port to the asynchronous port. Write operations to the FIFO are deferred while the FIFO is full by blocking the assertion of the wCLK signal with the SR Latch. If the FIFO is full and a pending data request exists, the lr signal will not propagate to the write clock signal wCLK so long as the full flag is asserted. As soon as data is read from a memory slot, the read pointer rdptr will update its value. This will result in the Full flag becoming unasserted. The SR Latch will then be released, allowing the lr signal to generate the write clock signal wCLK and acknowledge data write on la.

3) sync-async DI FIFO: The *sync-async* FIFO is similar to the *asyc-sync* FIFO. The design uses a SR Latch on the read port to generate the domain interface Valid signal. There is no synchronization that is needed to generate the read port Valid signal. Thus, asynchronous implementations will have half of the synchronization delay overhead compared to clocked

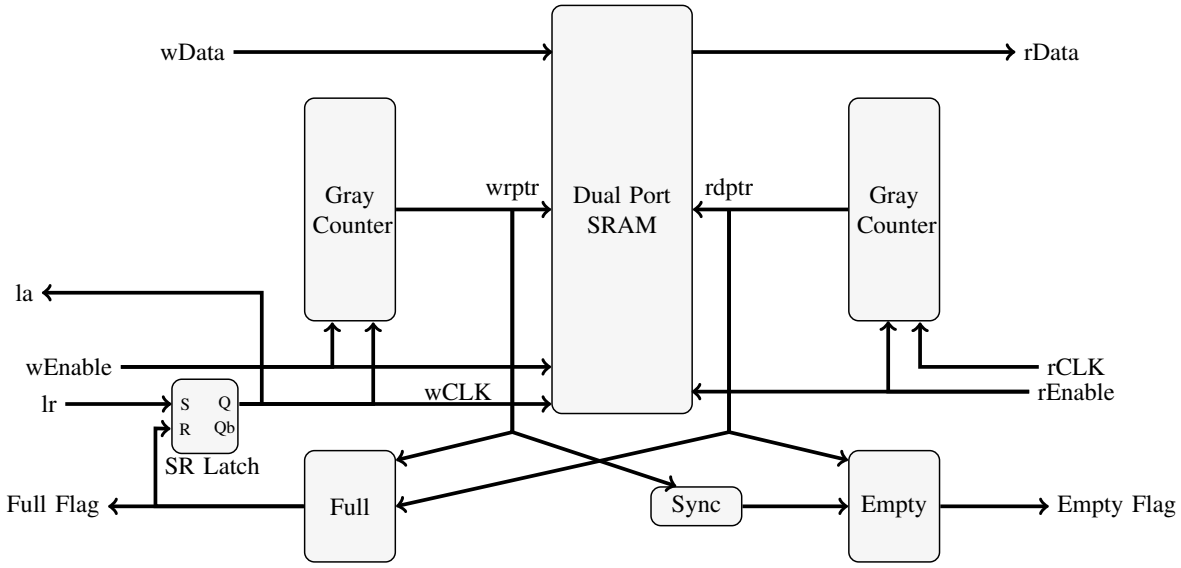

Fig. 7. *async-sync* Domain Interface FIFO

designs.

IV. Results

Fig. 4 shows the top level implementation for the test setup used to generate the various designs explained in this paper. The IP cores are abstracted out from the test setup. The test simulation directly drives signals into the IP back-end as though an IP were designed and directly connected. The NoC in the evaluation performed here is simply a point-to-point connection. It is implemented with nothing more than wires between NW_BE1 and NW_BE2. This was done to provide better area and power comparisons of the OCP blocks themselves by not adding in extra IP logic. All blocks are placed in close proximity for this evaluation. The IPs were considered to be on the same domain, i.e. either clocked at the same frequency, or unclocked asynchronous IP. For all the designs with multiple domains, the leftmost and rightmost DI in Fig. 4 were the domain crossing blocks.

There are five different cases implemented, which include: a clocked system with a single clock domain (*sync*), clocked circuit with a different clock frequency used for the NoC (*sync-sync-sync*), asynchronous unclocked circuit (*async*), and mixed clocked and asynchronous designs *async-sync-async* and *sync-async-sync* (a GALS architecture). For each individual case, the operation of the design was validated for four OCP design modes: normal read/write, burst read/write, read/write with out-of-order response and burst read/write with out-of-order response.

The reported results use the Artisan library for the IBM 65nm 10sf process using full layout and parasitic extraction. Design Compiler was used for synthesis, Modelsim was used for simulation, and SoC Encounter was used for place, route, and parasitic extraction. The power and delay numbers used SDF (standard delay format) parasitic back annotation into the Modelsim. The power numbers were generated using parasitic extraction and activity factors from a simulation run by importing a VCD (value change dump) file from Modelsim into SoC Encounter. The simulation runs a set of read and write commands to validate the functioning of the design. Post layout timing was validated using PrimeTime with extracted parasitics.

The operating frequency of all the designs with single clock domain, i.e. *sync*, *async-sync-async* and *sync-async-sync* was 667MHz. For the multiply clock domain design, i.e. *sync-sync-sync*, the operating frequency of the OCP and NoC domain was 570MHz while each IP back-end operated at 667MHz. For the asynchronous blocks the constraints for the handshake controllers were specified based on the amount of logic in each pipeline stage as described in [3]. The simulation testbench performs 22 transactions consisting of 12 writes and 10 reads.

Tab. I and II show the results for designs with and without clock domain crossing in the domain interfaces. Performance is based on the simulation time for the testbench. Energy is the average per transaction.

Designs which did not include extended capability did not have those features compiled into the logic. It can be seen that the addition of extra logic for the burst, tag or both burst and tag, results in an increase in the area and power consumed by each design with respect to the base design without any extensions. Respective areas and power in Tab. II are over 3× larger than those in Tab. I. The overhead of domain crossing is also increased due to the addition of extra buffering using FIFO structures. These DI FIFOs are all 8 words deep.

Tab. I allows us to compare purely clocked and asynchronous designs. Designs that only use asynchronous components, are substantially better than those which use only clocked components in terms of area, performance and power. Results in the table are all relative to the purely clocked design. The *async* design shows up to a 8.9× improvement in power, 3.1× improvement in performance, and 1.4× improvement in area over the *sync* design.

Tab. II compares designs with clock domain crossings. The bulk of the OCP logic lies in the network domain, since the clock domain crossing boundaries are in the far left and right domain interfaces. Therefore, these result are largely dominated by the network clocking methodology. As one would expect, the GALS design, the *sync-async-sync* design, is far superior to the clocked design operating at multiple frequen-

TABLE I
ENERGY, PERFORMANCE AND AREA COMPARISON FOR DESIGN WITH NO DOMAIN CROSSING

	Area	Simulation	Energy/trans.	Area	Performance	Energy
	(um^2)	Time (ns)	(pJ)	Benefit	Benefit	Benefit
sync Design						
Normal	15,822.0	309.75	94.43	1.00×	1.00×	1.00×
Burst	17,422.8	210.75	74.84	1.00×	1.00×	1.00×
Normal + Tag	16,432.8	144.75	52.40	1.00×	1.00×	1.00×
Burst + Tag	18,337.8	144.75	58.06	1.00×	1.00×	1.00×
async Design						
Normal	11,572.2	100.65	10.63	1.37×	3.08×	8.88×
Burst	14,218.8	81.36	13.09	1.23×	2.59×	5.72×
Normal + Tag	11,955.0	53.56	11.04	1.37×	2.70×	4.74×
Burst + Tag	13,518.0	52.62	12.49	1.36×	2.75×	4.65×

TABLE II
ENERGY, PERFORMANCE AND AREA COMPARISON FOR DESIGN WITH DOMAIN CROSSING

	Area	Simulation	Energy/trans.	Area	Performance	Energy
	(um^2)	Time (ns)	(pJ)	Benefit	Benefit	Benefit
sync-sync-sync Design						
Normal	59,419.8	401.25	414.02	1.00×	1.00×	1.00×
Burst	63,574.8	242.25	275.28	1.00×	1.00×	1.00×
Normal + Tag	61,737.6	138.75	160.19	1.00×	1.00×	1.00×
Burst + Tag	66,683.4	138.75	182.27	1.00×	1.00×	1.00×
sync-async-sync Design						
Normal	50,866.2	102.75	75.19	1.17×	3.91×	5.51×
Burst	55,956.0	98.25	79.05	1.14×	2.47×	3.48×
Normal + Tag	53,438.4	78.75	71.59	1.16×	1.76×	2.24×
Burst + Tag	57,557.4	78.75	81.97	1.16×	1.76×	2.22×
async-sync-async Design						
Normal	51,586.8	332.70	226.84	1.15×	1.21×	1.83×
Burst	55,584.0	235.58	186.32	1.14×	1.03×	1.48×
Normal + Tag	54,436.8	178.58	144.49	1.13×	0.78×	1.11×
Burst + Tag	58,363.8	178.59	155.96	1.14×	0.78×	1.17×

cies and the LAGS (*async-sync-async* – locally asynchronous globally synchronous) design. One of the primary benefit of asynchronous design is that it does not require synchronization when signals move into an asynchronous domain.

The performance penalty for the synchronous system in this work is due to the necessity to synchronize when moving into a new clock domain. However, it is also partly due to the implementation of elasticity and its stall mechanism. Hence analysis of better stall protocols might make the results better for designs that are partly or fully synchronous, but that is left for future work.

V. CONCLUSION

This paper reports on the design and implementation of OCP sockets for modular integration of IP blocks. The work implements four OCP operational modes including standard read write operation, burst transactions of up to eight words, bus transactions with out-of-order responses, and burst transactions with out-of-order responses. Additional bus byte enable commands are included.

A novel domain interface (DI) is introduced, which acts as a clear boundary for different clocked and unclocked domains and also to create more modularity when interfacing new IP cores to the OCP protocol. This DI concept enhances the modularity of the design by requiring only a single OCP master and slave front end to be shared between all IP components in a system-on-chip (SoC).

The domain interface supports SoC designs with IP blocks operating with independent frequencies. The IP blocks are implemented with either clocked or asynchronous circuit design timing methodologies. Clocked and asynchronous OCP master and slave components were designed and implemented. Five different domain interface blocks were implemented in order to enable all possible multi-synchronous clocked and asynchronous SoC designs, including globally asynchronous locally synchronous (GALS) implementations. The DI block enables the rapid design and evaluation of complete SoC designs, consisting of arbitrary timing methodologies within each IP block.

Five different designs were evaluated under a uniform test bench. These included designs with a single global clock, fully asynchronous, and multi-synchronous designs. Designs with substantial asynchronous components produced the best results in terms of area, performance, and energy per transfer. The purely asynchronous design has three times the performance and approximately one-ninth the energy of the clocked design. The GALS design also demonstrated almost four times the throughput at less than one-fifth the energy per transaction. Much of the performance is due to the decreased area and performance penalty for synchronization as well as the lower latency for asynchronous designs.

The presence of non-deterministic delays and any associated data-validity scheme that interfaces to clocked designs comes with a significant overhead. These costs are highlighted in the results. The impact on clocked IP blocks was not investigated in this study, but is left to future work. Likewise, the impact on design time and the ability for the domain interface to simplify back end integration will also be addressed in future work.

VI. ACKNOWLEDGMENTS

This material is based upon work supported by Semiconductor Research Corporation under task number 2235.001 and the National Science Foundation under Grant Number 0810408.

REFERENCES

[1] L. Benini and G. D. Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[2] *Open Core Protocol Specification Ver.3*, Open Core Protocol, http://www.ocpip.org/.

[3] K. S. Stevens, Y. Xu, and V. Vij, "Characterization of Asynchronous Templates for Integration into Clocked CAD Flows," in *15th International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2009, pp. 151–161.

[4] C. E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," in *SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers*, 2001, URL - http://www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf.

[5] ——, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," in *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, 2002, URL - http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.

[6] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the Digital Automation Conference (DAC06)*. IEEE, July 2006, pp. 657–662.