Automated Bus Generation for Multiprocessor SoC Design

Kyeong Keol Ryu and Vincent J. Mooney III
Georgia Institute of Technology
School of Electrical and Computer Engineering, Atlanta, GA 30332, USA
{kkryu, mooney}@ece.gatech.edu

***Abstract*—The performance of a multiprocessor system heavily depends upon the efficiency of its bus architecture. This paper presents a methodology to generate a custom bus system for a multiprocessor System-on-a-Chip (SoC). Our bus synthesis tool, which we call BusSyn, uses this methodology to generate five different bus systems as examples: Bi-FIFO Bus Architecture (BFBA), Global Bus Architecture Version I (GBAVI), Global Bus Architecture Version III (GBAVIII), Hybrid bus architecture (Hybrid) and Split Bus Architecture (SplitBA). We verify and evaluate the performance of each bus system in the context of three applications: an Orthogonal Frequency Division Multiplexing (OFDM) wireless transmitter, an MPEG2 decoder and a database example. This methodology gives the designer a great benefit in fast design space exploration of bus architectures across a variety of performance impacting factors such as bus types, processor types and software programming style. In this paper, we show that BusSyn can generate buses that, when compared to a typical General Global Bus Architecture (GGBA), achieve superior performance (e.g., 41% reduction in execution time in the case of a database example). In addition, the bus architecture generated by BusSyn is designed in a matter of seconds instead of weeks for the hand design of a custom bus system.**

***Index Terms*—Bus architecture, bus generation, design space exploration, Intellectual Property (IP), System-on-a-Chip (SoC), synthesis.**

I. INTRODUCTION

State-of-the-art chip design technology enables System-on-a-Chip (SoC) to open up new opportunities for Very Large Scale Integration (VLSI) hardware design. For example, SoC allows the designer to overcome some performance drawbacks of Printed Circuit Boards (PCBs) by implementing on a single chip many or most of the chips previously on a PCB. In particular, single-chip integration allows the designer to take advantage of increased bus speeds and widths. Another trend that SoC enables is for the designer to integrate multiple Processing Elements (PEs) on a single chip. Thus, an effective bus architecture with efficient arbitration for reducing contention among multiple PEs plays an important role in maximizing the performance of an SoC.

In the design of an SoC, one obvious issue for an SoC designer to consider is how to exchange data among multiple

PEs in the SoC. For instance, should there be one bus or multiple buses and where should memory elements be placed? Another issue for an SoC designer to consider is how to easily and quickly design a bus system considering the increasing complexity of on-chip bus systems and in the context of ever shortening time to market demands. These issues motivate the introduction of a design automation tool that is capable of generating customized SoC bus systems in Hardware Description Language (HDL) code to speed up a user's design space exploration in search of a high performance bus system.

This paper presents a methodology to generate custom bus systems using Intellectual Property (IP) for a multiprocessor SoC. Using this methodology, five different bus systems [1][2] are generated as examples in synthesizable Verilog HDL: Bidirectional First-In-First-Out (Bi-FIFO) Bus Architecture (BFBA), Global Bus Architecture Version I (GBAVI), Global Bus Architecture Version III (GBAVIII), Hybrid bus architecture (Hybrid) that combines BFBA and GBAVIII, and Split Bus Architecture (SplitBA). Each bus system performance is evaluated using three applications: an Orthogonal Frequency Division Multiplexing (OFDM) wireless transmitter, an MPEG2 decoder and a database example. We also show that our Bus Synthesis (BusSyn) tool can efficiently generate a large variety of bus systems in a matter of seconds (as opposed to weeks of design effort to put together each bus system by hand). Furthermore, we compare the performance of each bus system with a simple General Global Bus Architecture (GGBA) or an industry standard on-chip bus (CoreConnect from IBM [3]), showing up to 41% reduction in application execution time with a customized bus architecture.

This paper is organized as follows. Section II shows related work, and Section III explains some of the terminology applied to describe our approach. Section IV depicts a bus system structure, several custom bus system examples and bus-based data communication among multiple PEs for each example. Section V presents a detailed description of the methodology. In Section VI, we explain the applications used to evaluate the generated bus systems and then show experimental results. Finally, we conclude this paper in Section VII.

II. RELATED WORK

The ability to design a VLSI multiprocessor SoC is required in order for the design to be adapted to following several constraints: shorter time to market, ease of design, correctness of design, huge gate counts and high performance. Specifically, shorter time to market, ease of design and correctness of design have been hot issues,

which have been approached through design automation using computer-aided design (CAD) tools.

Most SoC bus designs are based on IPs stitched together with various forms of data, address and control links. Several efforts from industry provide platforms to connect the IP blocks used in an SoC. For example, CoreConnect has three levels of hierarchy: Processor Local Bus (PLB), On-chip Peripheral Bus (OPB) and Device Control Register bus (DCR) [3]. PLB provides a high performance and low latency processor bus with separate read and write transactions, while OPB provides a low speed bus with separate read and write data buses to reduce bottleneck effects caused by slow I/O devices such as serial ports, parallel ports and UARTs. The daisy-chained DCR offers a relatively low-speed data path for passing status and configuration information. The Advanced Micro-controller Bus Architecture (AMBA) from ARM has two levels of hierarchy: the Advanced High-performance Bus (AHB), which is similar to PLB, and the Advanced Peripheral Bus (APB), which is similar to OPB [4]. CoreConnect and AMBA, which are pipelined buses, both require bridges between the high performance bus and the low speed bus for data transfer between the buses. CoreFrame from Palmchip Company is currently advertised as a non-pipelined bus architecture that has two independent bus types: Mbus for memory transfer and Palmbus for I/O devices [5]. The SiliconBackplane from Sonic Inc. attempts to guarantee fixed bandwidth and latency by Time Division Multiplexed Access (TDMA) based arbitration [6]. These buses and their additional CAD tools such as FastForward for SiliconBackplane and Connection Kit [7] for CoreFrame allow a designer to integrate IP modules in an easy way and thereby result in reduced design time for an SoC. We take the IBM CoreConnect bus architecture as a representative example of these industry buses. We will show in Section VI.C how our tool outperforms CoreConnect by 15.54% in the case of an MPEG2 decoder running on a four-PowerPC SoC.

As another related research for bus system generation for an SoC, several papers represent communication topology generation, IP assembling for an SoC and component-based SoC design as follows.

Gasteier *et al.* [8] described the automatic generation of a communication topology by using scheduling of data transfer operations to reduce the cost (e.g., area) of a bus architecture. However, their algorithm only supports for a single type of bus topology (a single global bus topology). Our method, on the other hand, supports multiple bus types.

Bergamaschi *et al.* [9] presented design automation of an SoC using IP cores connected via CoreConnect. In their methodology for assembling IP, their algorithm checks the compatibility of IP input/output ports and generates wires to connect the IP cores. Again, we, on the other hand, support a wider variety of bus types and architectures than they

presented.

Pai Chou *et al*. [10] showed an IP based approach to SoC building. An input description to their algorithm designates a bus topology that specifies how IPs are connected with each other and which bus protocol is used. Communication synthesis in their tool implements the bus topology together with the generation of device drivers, message routers and communication devices, so that the IPs communicate with each other by using a network protocol (e.g., I^2C or CAN) that they assume. BusSyn, on the other hand, assumes that high-performance direct on-chip bus connections are desired rather than using a complicated network protocol such as I^2C or CAN. Thus, BusSyn targets SoC designs where direct, non-packet based connections are desired. For this reason, BusSyn focuses on generating hardware blocks of dedicated bus logic for application specific communication including handshake registers and bus arbiters for a customized bus architecture. This contrasts with the work of Pai Chou *et al*., which did not generate customized SoC bus architectures but rather assumed that such bus architectures are already available (e.g., a CAN bus).

Several efforts [11][12][13][14][15][16][17] from TIMA Lab presented a component-based design flow for a heterogeneous multi-core SoC. Their design flow introduces a systematic method of wrapper generation for multi-core SoC design based on architectural parameters extracted from a high-level system specification. Lyonnard *et al.* [11] introduced a design flow for the generation of application-specific multiprocessor architecture. They used a generic multiprocessor architecture template to support two types of buses (e.g., point-to-point connection and shared bus) and a communication coprocessor for the interface between processor and bus. To interface each heterogeneous component to the other part of system, they depicted a generic wrapper architecture that adapts to different communication protocols and abstraction based on automatic wrapper generation [11, 12, 13]. W. Cesário *et al.* [14, 15] and Nicolescu *et al*. [17] described a component-based design environment to enable an automatic wrapper generation tool to support for hardware interfaces, device drivers and Application-specific Program Interfaces (APIs).

We, on the contrary, focus on bus architecture in the component-based SoC design and provide more flexible bus architecture template to generate bus systems. The template supports multiple and heterogeneous bus architectures (e.g., GBAVI, GBAVIII, BFBA, Hybrid and SplitBA) in a system and various optimized wrappers (e.g., CPU-, memory- and generic-bus interface).

III. TERMINOLOGY

Before proceeding to discuss our Bus Synthesis tool (BusSyn), we first explain some of the terms we will be using

to describe the different components of a bus architecture. Example 1 explains the terminology we have defined.

Definitions

(A) Processing Element (PE): a hardware unit that performs algorithmic processing – usually a CPU but may also be dedicated or reconfigurable logic.

(B) Bus Bridge (BB): a hardware unit that is an on-off controllable connection point between two buses – if the BB is enabled, the two buses are fully connected; otherwise the two buses are disconnected. Note that our BB does not currently support different bus speeds in buses connected by the BB.

(C) Global Bus Architecture (GBA): a type of bus architecture having a bus through which all PEs can access shared resource(s), where BBs may be used to connect different sections of the bus.

(D) Bi-FIFO Bus Architecture (BFBA): a type of bus architecture where bidirectional FIFOs are used to transmit and receive data between adjacent PEs.

(E) Segment of Bus (SB): a contiguous bus (no BBs) consisting of address, data and control (e.g., read enable, write enable, request and acknowledge) wires specific to a particular bus type (in our case, GBA or BFBA).

(F) Bus Access Node (BAN): An integrated hardware block that is composed of at most one PE, custom IP blocks and/or memory hardware together with associated bus access hardware and SB(s).

(G) Module: a hardware unit such as BB, SB, an arbiter, SRAM or IL (in this paper, a PE is not a Module but instead is an IP core), where IL is a specific Interface Logic that will be explained in more detail in Section IV.A. Note that it is possible to extend the definition of Module to include newly designed hardware units that carry out specific functions. For this paper, however, the definition given for Module suffices.

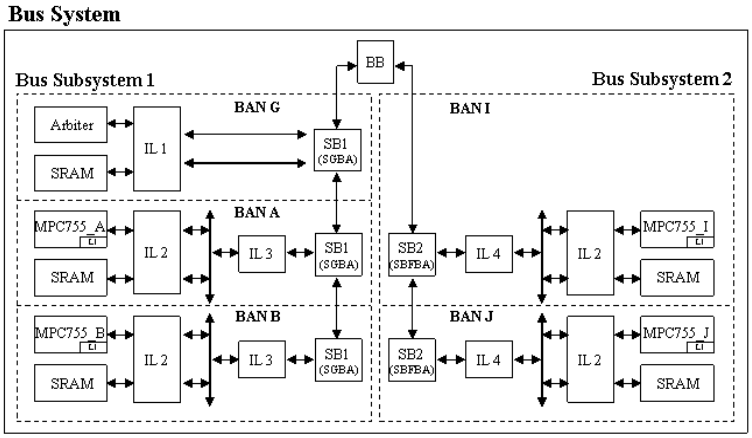
(H) Bus Subsystem: a subsystem that consists of one or more BANs connected together using the same bus or the combination of different bus architectures (in our case, either GBA, BFBA or the combination of GBA and BFBA).

(I) Bus System: a system that consists of one or more Bus Subsystems connected together.

Example 1: Terminology

Figure 1 shows an SoC consisting of four PEs (MPC755s), each with an L1 cache. Each MPC755 is an example of a PE. In the bottom left of Figure 1 can be seen an SB−SB1−used to connect BAN B to the rest of the SoC. Note the use of an Interface Logic blocks (ILs) to connect MPC755_B to the Bus System. The bottom left of Figure 1 also shows

MPC755_B connected to local SRAM and an SB−SB1−to form Bus Access Node B (BAN B). In BAN B, each block such as SRAM, IL2, IL3 and SB1 is a Module. BAN B is adjacent to BAN A which is adjacent to BAN G. BANs A, B and G together form a Bus Subsystem using bus type GBA for communication. On the right hand side of Figure 1, BANs I and J form another Bus Subsystem in which BFBA is used for communication. A BB connects the two Bus Subsystems as shown in the top middle of Figure 1. On the whole, Figure 1 shows an example of a Bus System composed of two Bus Subsystems.



IL : Interface Logic, SB: Segment of Bus, BB: Bus Bridge, SGBA and SBFBA: the specific example of SB for GBA and BFBA

Figure 1. Bus System Example

□

IV. BUS SYSTEM STRUCTURE AND EXAMPLES

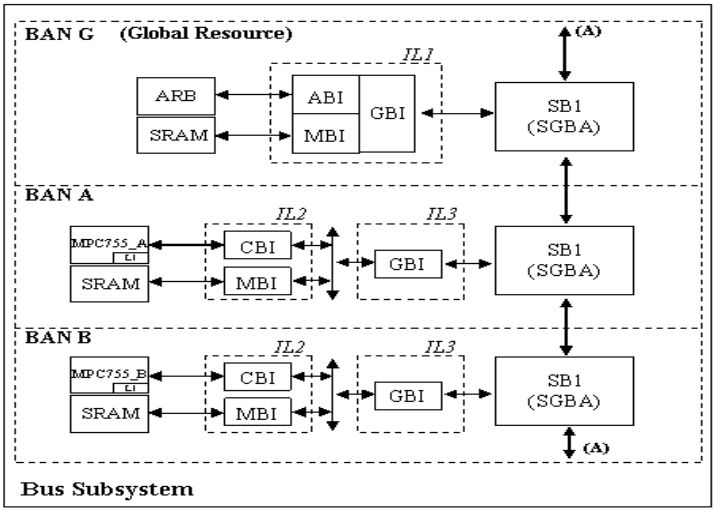
We now first describe a Bus System structure as a basement to generate several Bus Systems, which are exemplified in the following subsection, and then, we discuss communication among BANs in each of the Bus System example.

A. Bus System Structure

Figure 1 shows a hierarchical structure example of a multiprocessor Bus System that has two Bus Subsystems having two and three BANs, respectively. A Bus System is composed of one or more Bus Subsystems, and each Bus Subsystem includes one or more BANs, each of which is composed of PEs, IPs and/or memories together with associated hardware logic. The Bus Subsystems are connected through Bus Bridges. This kind of hierarchical definition allows a Bus System to have a flexible and scalable bus architecture in multiprocessor SoC Bus System design. Figure 2 depicts the Bus Subsystem shown on the left-hand-side of Figure 1 in more detail. In addition to PEs (e.g., MPC755) and memories (e.g., SRAMs) in the BANs of Figure 2, there are more Modules specified as Interface Logic (IL): CPU (or PE) to Bus Interface (CBI), Memory to Bus Interface (MBI) and Generic Bus Interface (GBI).

With these ILs, each BAN can have different types of PEs, IPs and/or memories because the ILs enables the heterogeneous modules to adapt to each other. For example, BAN A can have MPC755 and SRAM while BAN B can have ARM9TDMI and DRAM. Similarly, GBI also provides flexibility in selecting various types of buses for a Bus Subsystem (e.g., GBA−GBAVI or GBAVIII to be described in Section IV.B−and BFBA). Each BAN can access any other BAN's memory through a bus integrated with several SBs. Based on the Bus System structure, by simply repeating generated BANs, a Bus Subsystem can be a scalable structure, and a multiprocessor Bus System can be implemented in an easy manner.

When a Bus Subsystem has a global resource such as a large global memory to be accessed from all BANs, the resource is also defined as a BAN: for example, BAN G in Figure 2.



[Note] MEM: Memory, MBI: Memory-Bus Interface, CBI: CPU/PE-Bus Interface GBI: Generic Bus Interface, SB: Segment of Bus, ABI: Arbiter-Bus Interface, ARB: Arbiter, BAN: Bus Access Node, IL: Interface Logic

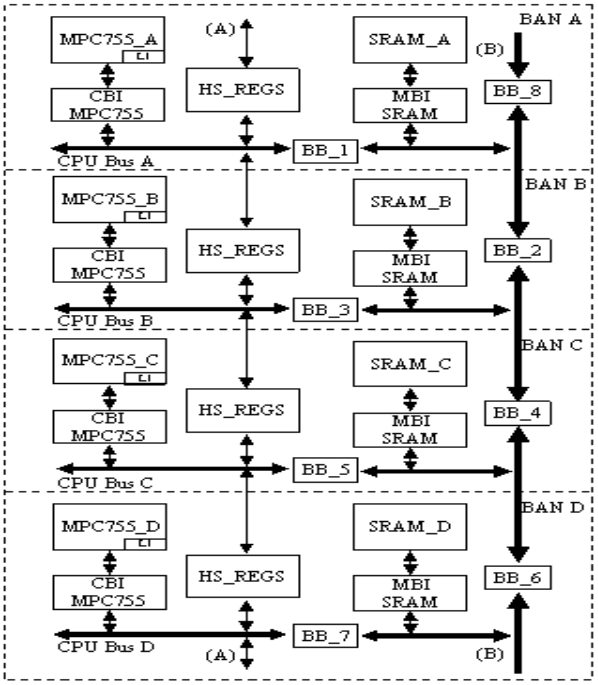
Figure 2. Example of a Bus Subsystem

B. Bus System Examples

In this section, we show five custom bus systems generated by BusSyn automatically: BFBA, GBAVI, GBAVIII, Hybrid and SplitBA. All Bus System examples shown in Figures 3, 4, 5, 6, 7, 8 and 9 have four PEs and total 32MB of memory (all examples have approximately the same chip area because the area of the bus logic and wires is much smaller than PE and memory area). Ordinarily, BusSyn can generate a Bus System having any number of PEs and any sizes of memories according to the user options (in Section V.B, we will describe how the user inputs the options). In

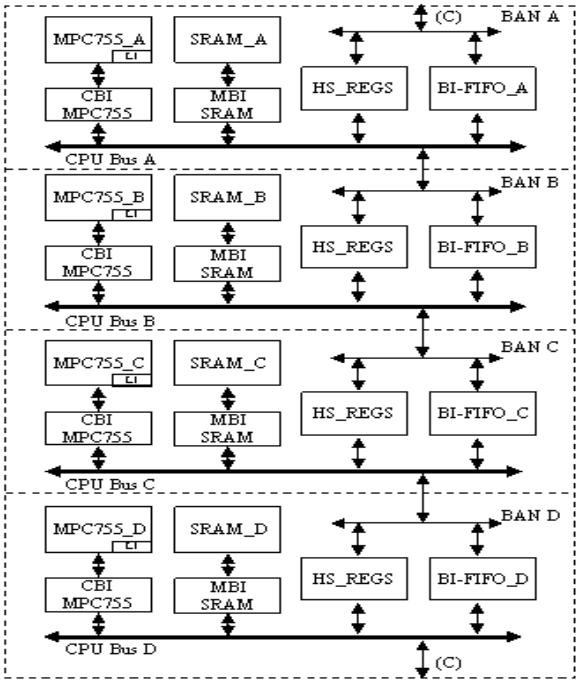
these examples, we use the Motorola PowerPC (MPC755) for the PE core, which, however, can be changed to other cores simply by adding a CBI Module for the new PE core (e.g., ARM9TDMI) to be operated in the Bus System.

First, we give a detailed explanation of the five sample custom bus architectures generated by BusSyn in this paper (BusSyn can generate a very large number of custom bus architectures). GBAVI shown in Figure 3 is a kind of global bus architecture (GBA), but the global bus is segmented by BBs(e.g., BB_2, _4, _6 and _8) separating each BAN, where the number of BANs is specified by the user. Each BAN has a SRAM block (e.g., SRAM_A, SRAM_B, SRAM_C or SRAM_D). One BB in each BAN controls possible bus connection between the PE side bus and the SRAM side bus in each BAN: BB_1 between CBI MPC755 and MBI SRAM in BAN A. Thus, in GBAVI, a group of two adjacent BANs can exchange data without any bus conflict with the other BANs in the SoC at the same time due to separation by BBs. For example, in Figure 3, while BAN A and BAN B communicate with each other, BAN C and BAN D also can do at the same time without any bus conflict. Each group of two BANs in Figure 3 is synchronized by handshaking using shared registers' block (HS_REGS) between BANs (see Section IV.C). Note that GBAVI tend to work well in a pipelined style operation; for example, output of a.PE (e.g., MPC755_A) is passed to the next PE (e.g., MPC755_B).



[Note] BB: bus bridge and HS_REGS: handshake registers

Figure 3. Diagram of GBAVI



[Note] HS_REGS: handshake registers

Figure 4. Diagram of BFBA

As shown in Figure 4, BFBA has a Bi-FIFO between adjacent BANs. This design is similar to some commercially available multiprocessor PCBs such as the Quad TMS320C6701 Processor VME Board from Pentek [18]. One BAN can push data into a Bi-FIFO while an adjacent BAN can read the data from the Bi-FIFO. In this way, the PEs can carry on successive functions for a pipelined operation. A specific way to communicate over the PEs in Figure 4 will be presented in Section IV.C. Note that BFBA also works well in a pipelined style operation.

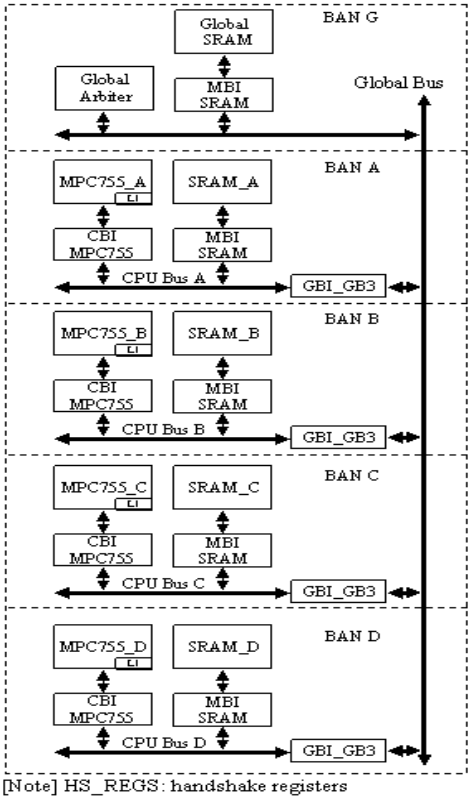


Figure 5. Diagram of GBAVIII

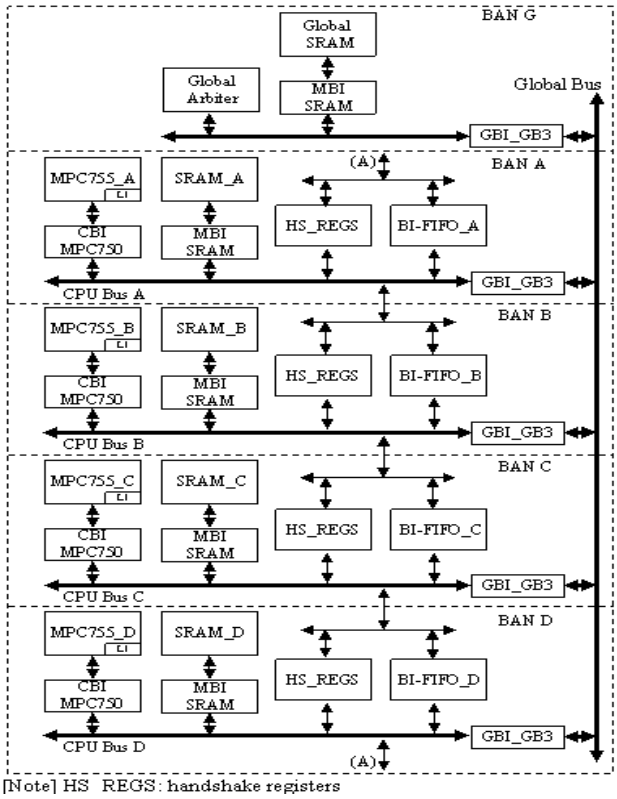


Figure 6. Diagram of Hybrid

GBAVIII shown in Figure 5 is a global bus architecture (GBA) having a global arbiter and a global memory. While any BAN tries to access the global memory through the global bus, the global arbiter resolves multiple memory requests from the BANs. The arbiter has a first-come-first-serve (FCFS) scheduling scheme using a FIFO, but the arbiter can have a different policy such as a priority-based protocol. The Global SRAM in Figure 5 also can be replaced with another memory type by using its corresponding MBI that adapts the interface between the memory and the bus. The local memory in each BAN can be used for relatively faster memory access than the global memory due to arbitration time. How to communicate among BANs in Figure 5 will be shown in Section IV.C. Also, please note that Global Bus Architecture Version II (GBAVII) was presented in [1] but was not chosen for automated generation

in this paper due to space constraints; however, if desired, the GBAVII bus could easily be added to our tool.

Another possible Bus System, Hybrid, is the combination of BFBA and GBAVIII, as shown in Figure 6. This combination allows the bus architecture to exploit advantages of both BFBA and GBAVIII (i) by supplying a Bi-FIFO data transfer method between adjacent BANs and (ii) by having a global memory area that can be accessed from all BANs. This combination of features gives flexibility in communication and thus results in a higher performance, although a penalty is paid in increased chip area (see Table V in Section VI.C for details).

Figure 7 shows SplitBA that is composed of two Bus Subsystems that have two MPC755s and a global memory respectively. Two Bus Subsystems are connected through a bus bridge to exchange data between them. Both Bus Subsystems in Figure 7 can operate at the same time without bus contention so that system performance is increased. In addition, in each Bus Subsystem, a bus length relatively shorter than using a single GBA makes the system be speedy and even consume lower power due to lower parasitic resistance and capacitance in the buses in the SoC [19]. SplitBA also relieves bus traffic congestion caused by shared memory requests from each BAN due to divided bus.

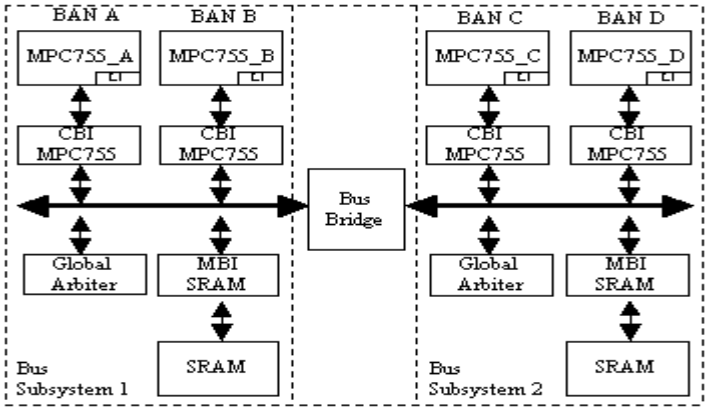


Figure 7. Diagram of SplitBA

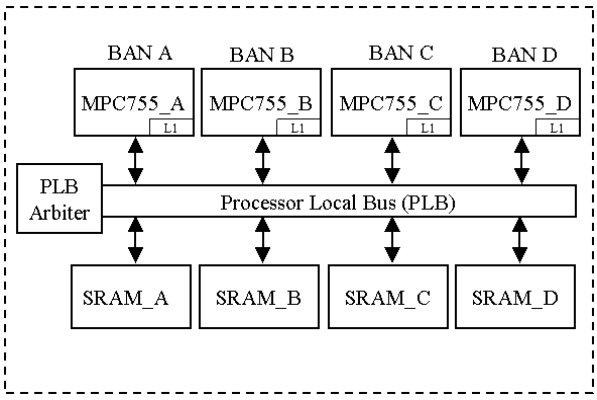


Figure 8. Diagram of CCBA

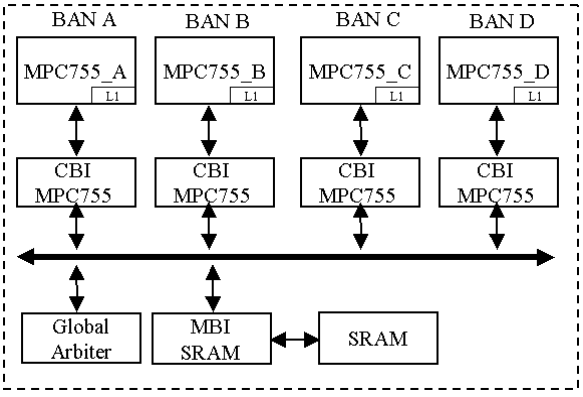


Figure 9. Diagram of GGBA

CCBA from IBM and GGBA are shown in Figures 8 and 9, respectively. These bus architectures are designed by hand and are used as a baseline for performance comparisons with our generated Bus Systems.

C. Communication among BANs

In a multiprocessor SoC, applications are typically partitioned across multiple PEs for parallel processing. As a consequence, the communication method among the PEs considerably influences on the system performance. If all PEs in the system could cooperate together without any conflict in communication, which is what we desire, the overall system performance would be significantly increased in the parallel processing.

In this subsection, we introduce a communication method that calls for minimal conflicts in bus-based communication. Specifically, we introduce a handshake protocol for the bus-based communication because the protocol is simple in operation and straightforward in implementation. We first describe our basic handshake protocol and then show the adaptation of the protocol to each specific Bus System in following subsections (see Examples 3, 4 and 5).

Our handshake protocol uses only two control registers. These registers are accessed by two communicating PEs, a sender and a receiver. The protocol is different from a typical handshake protocol in a sense that the typical protocol needs three registers to control communication [21]. The typical handshake protocol keeps track of the following three conditions or states: (1) read request, (2) data ready and (3) acknowledge. Here, condition (1) indicates a read request from a receiver to a sender; condition (2) specifies that data is now ready to be accessed; and condition (3) is used to acknowledge conditions (1) and (2) of the other party. Our protocol, on the other hand, only needs to keep track of two of the conditions or states: (2) and (3). The reason is that we exploit a particular characteristic in parallel processing. That is, application functions running on all PEs have data dependencies among the functions when the application functions are partitioned across multiple PEs for parallel operation. Due to the data dependencies, it is not necessary for a receiver to use condition (1) because a receiver needs to wait anyway until a sender has done its processing and emits data to be used in the receiver. Therefore, we eliminate condition (1), “read request,” and thus use only two control registers to check the conditions (2) and (3). The conditions are checked by the value of each control register as shown in Example 2. Please note that in cases where condition (1) is needed, then obviously the handshake protocol can be altered to include condition (1) or indeed any other additional conditions which may be

necessary, and our generated bus architectures can support any such handshake protocols. However, in this paper we only show examples using the described protocol using only conditions (2) and (3).

Example 2: Handshake Control Registers

We denote two control registers as DONE_OP and DONE_RV, which correspond to condition (2) and condition (3), respectively. Each of two registers is one-bit width. The values of the registers have following meaning. While a value "1" of DONE_OP indicates that the sender has done its operation and thus is ready to send the processed data, a value "0" indicates that the sender is not ready yet. In the case of DONE_RV, a value "1" of DONE_RV shows that a receiver has received data from a sender, and a value "0" indicates that the data has not received yet. After checking each condition, data is transferred from a sender to a receiver through a specific bus in each Bus System shown in Subsection IV.B. □

For the sake of easy programming and program reliability, we developed APIs that are responsible for the communication procedure in software. The APIs (e.g., mem_read(), see Example 3) read an exact amount of data (specified by the user) from the user specified source area of the sender memory and store the data to the user specified target area of the receiver memory. To handle this kind of data transfer, the APIs have several parameters such as size of data, source address and target address.

In Bus Systems containing a global bus style (e.g., GBAVIII, SplitBA, GGBA and CCBA), possible bus conflicts may occur since multiple PEs can access the control registers at the same time. However, these possible conflicts can be resolved by exploiting an arbiter in the Bus System. The detailed communication procedures for each Bus System are shown in Examples 3, 4 and 5. Please note that the specific handshaking protocol presented here can easily be replaced by a typical handshake protocol [21] or any other 2-state or 4-state handshake protocol with no effect whatsoever on the rest of the methodology presented in this paper.

C.1 Communication in GBAVI

We now show how our handshake protocol to adapt for the communication between PEs in the GBAVI Bus System shown in Figure 3. To support the protocol, two control registers, DONE_OP and DONE_RV introduced in Example 2, reside in the handshake registers' block (HS_REGS) shown in each BAN of Figure 3. Each pair of neighboring PEs shares the registers (i.e., both a sender and a receiver can access the registers). When it is necessary for non-adjacent PEs to communicate with each other (e.g., transferring data from MPC755_A to MPC755_C in

Figure 3), currently we only support the case where all PEs (e.g., MPC755_B) between a sender (e.g., MPC755_A) and a receiver (e.g., MPC755_C) relay the data to the destination PE sequentially. However, our implementation could be extended to support direct communication through bus bridges (e.g., BB_2 and BB_4 between MPC755_A and MPC755_C). Note that GBAVI, as implemented, tends to work well in a pipelined style operation that has a pattern in which output data from a PE is passed to the next PE for the following operation. Example 3 shows how BAN A and BAN B communicate with each other in GBAVI. The other BANs in Figure 3 communicate in the same manner as shown in Example 3.

Example 3: Communication in GBAVI

We assume that BAN A and BAN B in GBAVI shown in Figure 3 execute an algorithm (e.g., OFDM transmitter that will be introduced in Section VI.A) in a pipelined operation fashion; the result data from BAN A passes to BAN B through a shared memory, which in this case is SRAM_A in BAN A. Figure 10 shows detailed handshake control registers (DONE_RV and DONE_OP) inside the "HS_REGS" block shown in BAN B of Figure 3. The registers can be accessed from both BAN A and BAN B. Note that the registers DONE_RV and DONE_OP are initialized to "0", and the step numbers in the following procedure correspond to the numbers in Figure 11, which shows a communication state diagram. The procedure for data transfer from BAN A to BAN B is as follows.

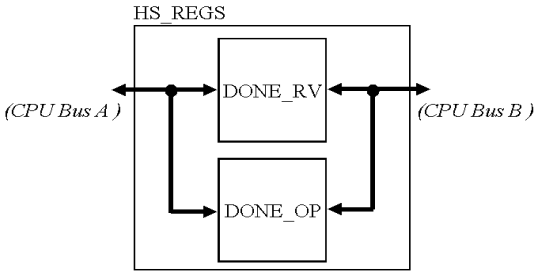


Figure 10. Detailed Diagram of HS_REGS in Figure 3

(1) Initial raw data from input source is read.

(2) After MPC755_A processes the data, the PE writes 64 processed data words to SRAM_A starting from address 0x00000 and sets DONE_OP to "1".

(3) MPC755_B resets register DONE_OP to "0" after reading value "1" in the register. Using an API "mem_read(64, 0x000000, 0x400000)", MPC755_B reads the 64 words of data from SRAM_A starting from address 0x000000 and stores the data to SRAM_B starting from address 0x400000.

(4) MPC755_B sets register DONE_RV to "1".

(5) After MPC755_A reads "1" in register DONE_RV, the PE resets the register to "0".

(6) MPC755_B processes the transferred data after step (4).

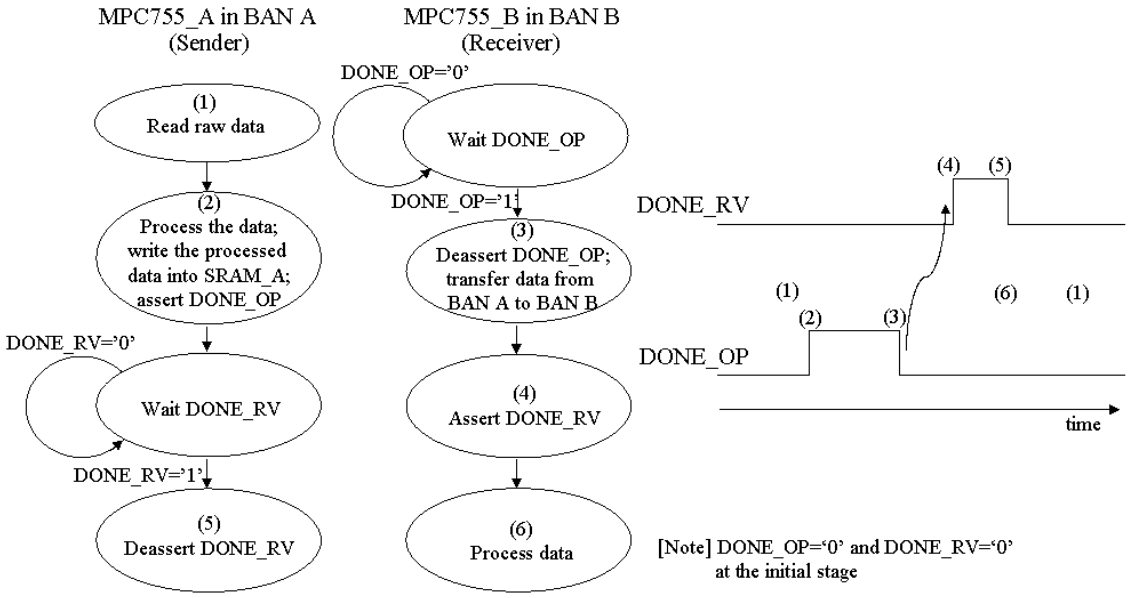


Figure 11. Communication between BANs in GBAVI

□

C.2 Communication in BFBA

PEs in BFBA Bus System shown in Figure 4 communicate using another adaptation of our handshake protocol since this communication takes advantage of an interrupt function. The handshake operation is implemented with an interrupt function and with two control registers DONE_OP and DONE_RV introduced earlier. These two registers are contained in handshake registers' block "HS_REGS" in Figure 4, and a threshold register in each Bi-FIFO controller specifies the size of data to be transferred and is set by a sender. Here, the Bi-FIFO controller is a hardware unit that controls Bi-FIFO memory in each Bi-FIFO block (e.g., Bi-FIFO_A, _B, _C or _D) shown in each BAN of Figure 4. As a sender pushes data into a Bi-FIFO memory in a receiver BAN, a Bi-FIFO counter in the controller is increased in hardware automatically, and then an interrupt signal is generated when the counter value is equal to the threshold register's value. The interrupt signal stimulates the receiver so that an interrupt handler in the receiver is executed. Functions in the interrupt handler are as follows: resetting DONE_OP to "0", popping received data from Bi-FIFO memory and setting DONE_RV to "1". In the communication between non-adjacent PEs, the PEs between the sender and the receiver have to relay the data to the destination PE sequentially. In this case, the communication might have a little overhead in time; however, note that this Bus System also is suitable for a pipelined style operation, which usually has adjacent PEs' communication. How to communicate between sender BAN A and receiver BAN B

in Figure 4 is shown in Example 4. The other BANs' communication in Figure 4 works in the same manner as the procedure shown in Example 4.

Example 4: Communication in BFBA

We assume that BAN A and BAN B in BFBA shown in Figure 4 execute an algorithm (e.g., OFDM transmitter that will be introduced in Section VI.A) in a pipelined fashion; the result data from BAN A passes to BAN B through Bi-FIFO_B in BAN B shown in Figure 4. Note that at the initial time, register DONE_OP is set to "1" while DONE_RV is set to "0" (these registers are in the "HS_REGS" block in BAN B of Figure 4). We also assume that the sender initially sets the threshold register in the Bi-FIFO controller to "64" to transfer sixty-four words of data at a time. The step numbers in the following procedure correspond to the numbers in Figure 12, which shows a communication state diagram. As shown in Figure 12, the communication procedure between the BANs is as follows.

(1) Initial raw data from input source is read and processed in BAN A.

(2) MPC755_A pushes 64 words of processed data into Bi-FIFO_B in BAN B after reading "1" in register DONE_OP.

(3) An interrupt handler API in MPC755_B runs after MPC755_A has finished pushing the output data; as shown in Figure 12**,** the interrupt API resets DONE_OP to "0", pops the sixty-four words of data from the Bi-FIFO_B and then sets DONE_RV to "1".

(4) MPC755_B resets DONE_RV to "0" after reading "1" in register DONE_RV.

(5) MPC755_B processes the popped data.

(6) MPC755_B sets DONE_OP to "1".

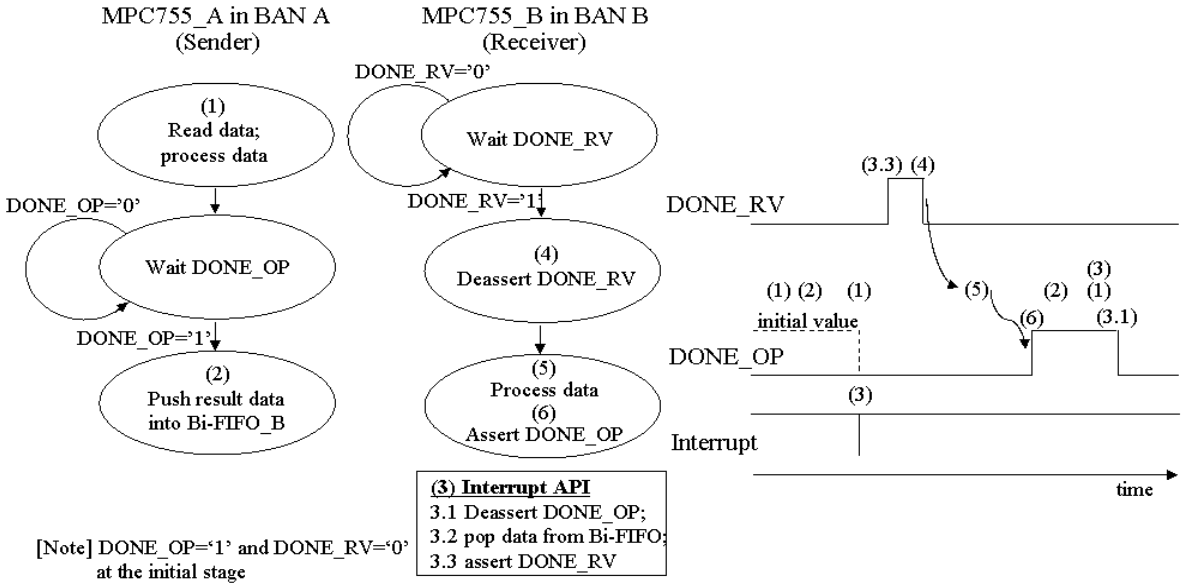


Figure 12. Communication between BANs in BFBA

□

C.3 Communication in GBAVIII

We introduce another adaptation of our handshake protocol for the communication in GBAVIII shown in Figure 5. GBAVIII is appropriate for both pipelined and functional parallel operation since a global memory, Global SRAM in BAN G, is employed as a communication buffer, which can be accessed from all PEs. By functional parallel operation, we refer to a parallel operation in which all PEs execute the same code for a complete algorithm but have different raw data to be processed. In this case, one of the PEs reads a chunk of raw data from the input source and writes the data to the global memory so that each PE can process its own assigned portion of the raw data. Please note that a Direct Memory Access (DMA) device can also work for such reading and writing functions, and the device can be supported in GBAVIII. In GBAVIII as presented in this paper, however, one of the PEs performs such functions rather than using DMA. In this functional parallel operation, there exists a dependency between one PE distributing the raw data and the other PEs receiving the data. This dependency enables our handshake protocol to adapt to GBAVIII for such a functional parallel operation.

For the handshake protocol operation, the GBAVIII Bus System shown in Figure 5 exploits global control variables saved in a specific region of a shared memory (e.g., Global SRAM in BAN G of Figure 5). Note that these variables work in a way similar to the control registers (e.g., DONE_RV and DONE_OP in HS_REGS block) employed in the previous subsections IV.C.1 and IV.C.2.

In this Bus System, the shared memory is used as a buffer not only for raw data from the input source but also for processed data from each BAN shown in Figure 5. Example 5 depicts the details of the communication procedure between BAN A and BAN B, and the other BANs in Figure 5 work in the same manner.

Example 5: Communication in GBAVIII

We execute an MPEG2 decoder algorithm, which will be introduced in Section VI.A in GBAVIII, Bus System shown in Figure 5. We first focus on describing the communication between BAN A and BAN B. We assume that the BANs execute the algorithm in the functional parallel operation style and start the operation at the same time. BAN A reads an MPEG2 raw video stream in the size of 1.47 KB, which is composed of Sequence Headers (SHs) and Group Of Pictures (GOPs), from an external source and writes the stream data to input buffer, which is located in the global memory Global SRAM in BAN G shown in Figure 5. After such an I/O processing, BAN A decodes the first SH and GOP while BAN B processes the second SH and GOP after reading the part of stream from the Global SRAM. In this manner, the video stream can be processed in parallel in each BAN. The step numbers in the following procedure correspond to the numbers in Figure 13. Note that the variables DONE_RV and DONE_OP in the Global SRAM are all initially set to “0”.

The variables are located in the variable area of the Global SRAM in BAN G of Figure 5. As shown in Figure 13, which depicts communication state diagram, the communication procedure between the BANs is as follows.

(1) In BAN A, MPC755_A reads MPEG2 video stream from an external source and writes the stream to input buffer (in the global memory Global SRAM of BAN G in Figure 5) for each PE, and then sets the variable DONE_RV to "1".

(2) MPC755_A processes the first SH and GOP and writes the processed data to output buffer in Global SRAM.

(3) While MPC755_A works as the step (2), MPC755_B reads the second SH and GOP from the Global SRAM after reading a value "1" from DONE_RV, and then the PE sets the variable DONE_RV to "0". After that, the PE starts processing its video stream.

(4) MPC755_B sets variable DONE_OP to "1" after finishing the data processing and writes the processed data to the output buffer in Global SRAM.

(5) MPC755_A resets DONE_OP to "0" after reading value "1" in variable DONE_OP.

Please note that all four BANs in Figure 5 process the MPEG2 decoding. While the above description was for the communication between BAN A and BAN B, the other BANs' communication works in the same manner as the procedure described above.

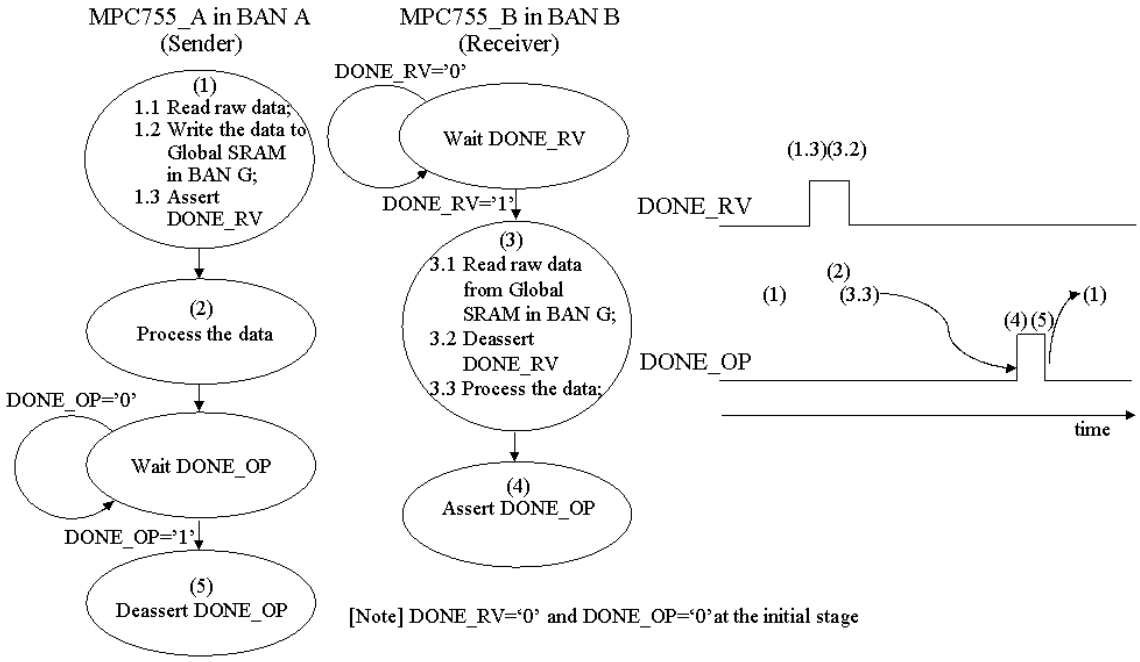


Figure 13. Communication between BANs in GBAVIII

□

C.4 Communication in Hybrid and SplitBA

As introduced earlier, the Hybrid Bus System shown in Figure 6 is a combined Bus System of BFBA (see Figure 4)

and GBAVIII (see Figure 5); in other words, the Hybrid has both the Bi-FIFO bus and the global bus. Therefore, in the case of using the Bi-FIFO bus in the Hybrid Bus System, the communication procedure is the same as the procedure shown in Example 4. On the other hand, when using the global bus in the Hybrid, the procedure is the same as that of GBAVIII as shown in Example 5.

The SplitBA shown in Figure 7 also has shared memory blocks (e.g., SRAM in each Bus Subsystem) that can be accessed from all PEs (e.g., MPC755_A, MPC755_B, MPC755_C and MPC755_D in Figure 7). Through the shared memory, the communication procedure among PEs can be implemented in the same manner as shown in Example 5.

V. METHODOLOGY FOR BUS SYSTEM GENERATION

Based on the Bus System structure described in Subsection IV.A, our bus synthesis tool BusSyn generates the Bus System examples shown in Subsection IV.B using two kinds of Libraries: Module Library and Wire Library. In this section, we show how the Libraries are made and work in the tool in the first subsection. Then, using the Libraries, we will show how to generate the Bus Systems in the next subsection.

A. Libraries for Module Repository and Wiring

BusSyn uses two libraries to generate a Bus System. One is a Module Library that contains Modules to be configured inside each BAN, and the other is a Wire Library for connecting the Modules inside a BAN and for connecting the BANs inside a Bus Subsystem.

The Module Library contains not only input/output port information and behaviour of each module in Register Transfer Level (RTL) Verilog but also many templates to generate specific modules (e.g., SRAM). Here, the templates have parameters to configure each of the specific modules, and the modules are generated by assigning specific values to the parameters. The Module Library contains the following components:

(A) <PE>: a processing element, where <PE> is one of MPC750, MPC755, MPC7410 or ARM9TDMI

(B) CBI_<PE>: an interface Module between a PE (or CPU) and bus

(C) <memory>_comp: a memory template to be used to generate any size of behavioural memory, where <memory> is one of SRAM or DRAM

(D) MBI_<memory>: an interface Module between a <memory> and bus, where <memory> is one of SRAM or DRAM

(E) BB_<bb_type>: a bus bridge, where <bb_type> is one of GBAVI or SplitBA

(F) ARBITER_<arb_type>: an arbiter Module, where <arb_type> is one of "Round Robin" or "Priority"

(G) ABI: an interface Module between arbiter and bus

(H) GBI_<bus_type>: a generic bus interface Module, where <bus_type> is one of GBAVI, GBVAIII or BFBA

(I) SB_<bus_type>: a Module for segment of bus (SB), where <bus_type> is one of GBAVI, GBVAIII or BFBA

Example 6: Module Library

As an example of a Module Library component, MBI_SRAM is shown in Figure 14. This component is for the interface between an SRAM and a bus as shown in Figures 3, 4, 5, 6 and 7. In Figure 14, the library component name is shown in the first line, "%module <library_name>", where <library_name> is MBI_SRAM. To specify MBI_SRAM's property, there are three parameters, @MEM_A_WIDTH@ for physical memory address width, @MEM_D_WIDTH@ for memory data width and @BIT_DIFFERENCE@ for difference in bit width between CPU data bus width and memory data bus width. For the interface between CPU bus A and 8MB SRAM in BAN A of Figure 4, the parameters are set as follows: @MEM_A_WIDTH@=20, @MEM_D_WIDTH@=64 and @BIT_DIFFERENCE@=0. In this library, control signals for reading from and writing to SRAM are decided by pin names: reb_local, sram_reb, web_local and sram_web. Please note that we assume that all addresses, which appear on a bus, are physical addresses. Any virtual addresses used by programs must be translated to physical addresses prior to placing them on the bus.

```
%module MBI_SRAM
module mbi_sram_bfgb2(addr_local, web_local, reb_local, dh, dl,
                      sram_addr, sram_web, sram_oeb, sram_dq );
parameter MEM_A_WIDTH = @MEM_A_WIDTH@;
parameter MEM_D_WIDTH = @MEM_D_WIDTH@;

input   [MEM_A_WIDTH-1:0]  addr_local;
input                      web_local;
input                      reb_local;
inout   [31:0]   dh;
inout   [31:0]   dl;
output  [MEM_A_WIDTH-1:0]  sram_addr;
output                     sram_web;
output                     sram_oeb;
inout   [MEM_D_WIDTH-1:0]  sram_dq;

assign  sram_addr = addr_local;
assign  sram_web = web_local;
assign  sram_oeb = reb_local;
assign  sram_dq = (~web_local) ? {dh, dl}: @MEM_D_WIDTH@'bz;
assign  {dh, dl} = (~reb_local) ? {@BIT_DIFFERENCE@'b0, sram_dq[MEM_D_WIDTH-1:0]}: 64'bz;
endmodule
%endmodule MBI_SRAM
```

Figure 14. MBI_SRAM component in Module Library

□

The Wire Library contains all possible combinations of legal connections between hardware blocks (e.g., between Modules in each BAN and between BANs in each Bus Subsystem). This library is written in ASCII format as shown in Figure 15, and there are several fields to specify connection information:

(A) wire name (w_name)

(B) wire width (w_width)

(C) module name (mx_name), where x indicates the module number, 1 or 2

(D) port name in module x (mx_pname)

(E) most significant bit (MSB) of wire connected to a module x (mx_wmsb)

(F) least significant bit (LSB) of wire connected to a module x (mx_wlsb)

```
%wire <library name>
w_name w_width m1_name m1_pname m1_wmsb m1_wlsb  \
        m2_name m2_pname m2_wmsb m2_wlsb
%endwire
```

Figure 15. Wire Library Format

In the Wire Library of Figure 15, m1_name and m2_name fields need to be different when a connection specifies a wire between different Modules or BANs. Examples 7 and 8 show the wire connection of these cases. However, to specify a wire between/among BANs that have same I/O ports in their pin names in a Bus Subsystem (e.g., the connection between BAN A and BAN B in Figure 4), m1_name and m2_name in Figure 15 need to be the same. This case is described in Example 8 in detail, where Figure 17(b) shows detailed blocks and I/O pins that are related to each BAN's I/O ports shown in Figure 17(a).

Example 7: Wire Connections in a BAN

As an example of a wire connection in a BAN, consider the wires between MBI_SRAM and SRAM_A in BAN A of Figure 4 BFBA. Figure 16 shows the detailed wires connecting SRAM_A to MBI_SRAM: w_addr for address bus, w_web for write enable, w_reb for read enable, w_csb for chip selection and w_dq for data bus. To specify the wires in Figure 16, the wire information in the Wire Library is as follows:

```
%wire ban_bfba
w_addr 20 SRAM_A sram_addr 19 0 MBI_SRAM addr 19 0
w_web 1 SRAM_A sram_web 0 0 MBI_SRAM web 0 0
w_reb 1 SRAM_A sram_reb 0 0 MBI_SRAM reb 0 0
```

w_csb 8 SRAM_A sram_csb 7 0 MBI_SRAM csb 7 0

w_dq 64 SRAM_A sram_dq 63 0 MBI_SRAM dq 63 0

%endwire

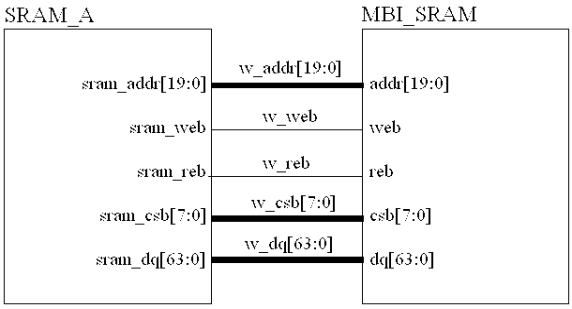


Figure 16. Wire Connection Example between SRAM A and MBI_SRAM in Figure 4

□

Example 8: Wire Connections between BANs in a Bus System

This example shows how to form wire connections between BANs in the Wire Library. BANs A, B, C and D are linked as in a chain as shown in Figure 17(a), and the connections of the I/O ports shown in the left box of Figure 17(b) are repeated between the BANs. In this kind of wire connections, the wire names and their connections between the BANs have the same names but with different suffixes as shown Figure 17(a). In this case, it is not necessary that we specify all wires and their connections related to the BANs. Thus, we specify wires and their connections between only two BANs in the Wire Library; however, our tool serially connects the linked BANs by generating wires suffixed by an enumerated number as shown in Figure 17(a). For that purpose, wire connections between BANs are specified by the same module names in the m1_name field and the m2_name field in the Wire Library as shown in Figure 15; in this example, the names are just "BAN[A,B,C,D]" as shown below. Here, "BAN[A,B,C,D]" means that the specified wire is applied for BANs A, B, C and D. On the other hand, the wires between BANs having different connections from the link have to be specified in the Wire Library as shown below. The connections between BAN B and BAN FFT in Figure 17(a) show the case where we assume that BAN B has another bus to BAN FFT in addition to the bus connecting BANs A, B, C and D. Here, the BAN FFT is a BAN having a hardware Fast Fourier Transform (FFT) IP.

Detailed wire connections between a pair of BANs A, B, C and D in Figure 17(a) are as follows: w_done_op(or rv)_cs for handshake register selection, w_ban_web for write enable, w_ban_reb for read enable, w_fifo_cs for FIFO chip selection and w_data for data bus as shown in Figure 17(a). In the connections between BAN B and BAN FFT, the wires are as follows: w_fft_ad: address for FFT buffer, w_fft_data: data bus, w_fft_reb for read enable, w_fft_web for write enable, w_fft_srt for FFT start control and w_fft_ack for acknowledge of FFT end. The wire connections among the BANs shown in Figure 17(a) are specified in the Wire Library as follows:

%wire subsys_bfba

w_done_op_cs 2 BAN[A,B,C,D] done_op_cs_dn 1 0 BAN[A,B,C,D] done_op_cs_up 1 0

w_done_rv_cs 2 BAN[A,B,C,D] done_rv_cs_dn 1 0 BAN[A,B,C,D] done_rv_cs_up 1 0

w_ban_web 1 BAN[A,B,C,D] web_dn 0 0 BAN[A,B,C,D] web_up 0 0

w_ban_reb 1 BAN[A,B,C,D] reb_dn 0 0 BAN[A,B,C,D] reb_up 0 0

w_fifo_cs 1 BAN[A,B,C,D] fifo_cs_dn 0 0 BAN[A,B,C,D] fifo_cs_up 0 0

w_data 64 BAN[A,B,C,D] data_dn 63 0 BAN[A,B,C,D] data_up 63 0

w_fft_ad 12 BAN[B] addr_b 11 0 BAN[FFT] addr_fft 11 0

w_fft_data 64 BAN[B] data_b 63 0 BAN[FFT] data_fft 63 0

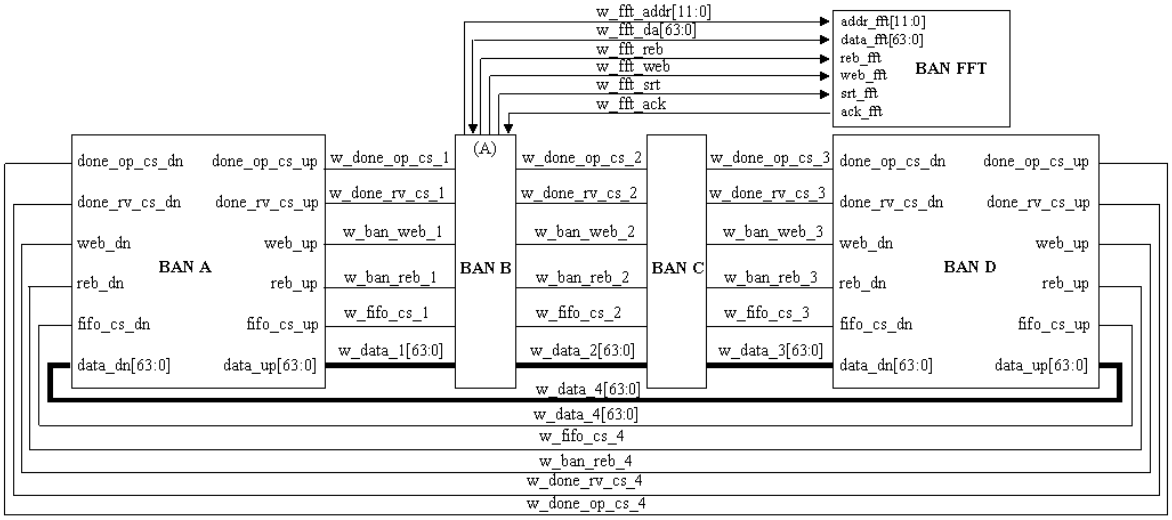
w_fft_reb 1 BAN[B] reb_b 0 0 BAN[FFT] reb_fft 0 0

w_fft_web 1 BAN[B] web_b 0 0 BAN[FFT] web_fft 0 0

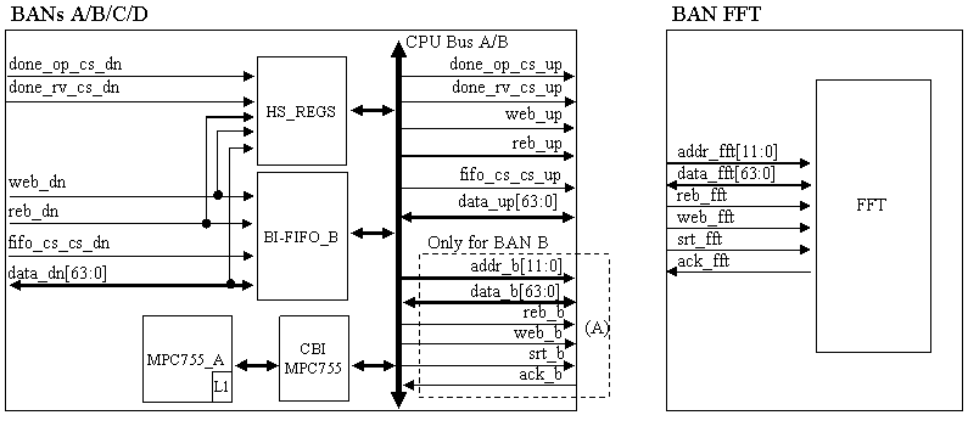
w_fft_srt 1 BAN[B] srt_b 0 0 BAN[FFT] srt_fft 0 0

w_fft_ack 1 BAN[B] ack_b 0 0 BAN[FFT] ack_fft 0 0

%endwire



(a) Connection among BANs



(b) I/O pins and blocks in each BAN in detail

Figure 17. Wire Connection Example between BANs in Figure 4

□

B. Bus System Generation Sequence

We now show how to generate the Bus Systems shown in Section IV.B. Before that, we explain user options to configure the user specific Bus System to be generated from our bus synthesis tool BusSyn. Then we detail the algorithms of BusSyn.

To configure a custom Bus System in BusSyn, the user first enters input options as shown in the right hand side box of Figure 18. These options are input constraints used to generate a custom Bus System. Several categories in these options are as follows:

(A) Bus System Property: number of Bus Subsystems in a Bus System.

(B) Bus Subsystem Property: number of BANs, number of buses and bus type in each bus, where the bus type is one of GBAVI, GBAVIII, BFBA or SplitBA.

(C) Bus Property: address bus width, data bus width and Bi-FIFO depth for each bus type specified in each Bus Subsystem, where the Bi-FIFO depth is available only for BFBA and Hybrid.

(D) BAN Property: CPU type or Non-CPU type and number of memories for each BAN, where the CPU type is one of MPC750, MPC755 or ARM9TDMI, and the Non-CPU type is one of DCT or MPEG2 decoder.

(E) Memory Property: memory type, address bus width and data bus width for each memory specified in each BAN, where the memory type is one of SRAM, DRAM, DPRAM or FIFO. Note that this can easily be extended to include additional memory types.

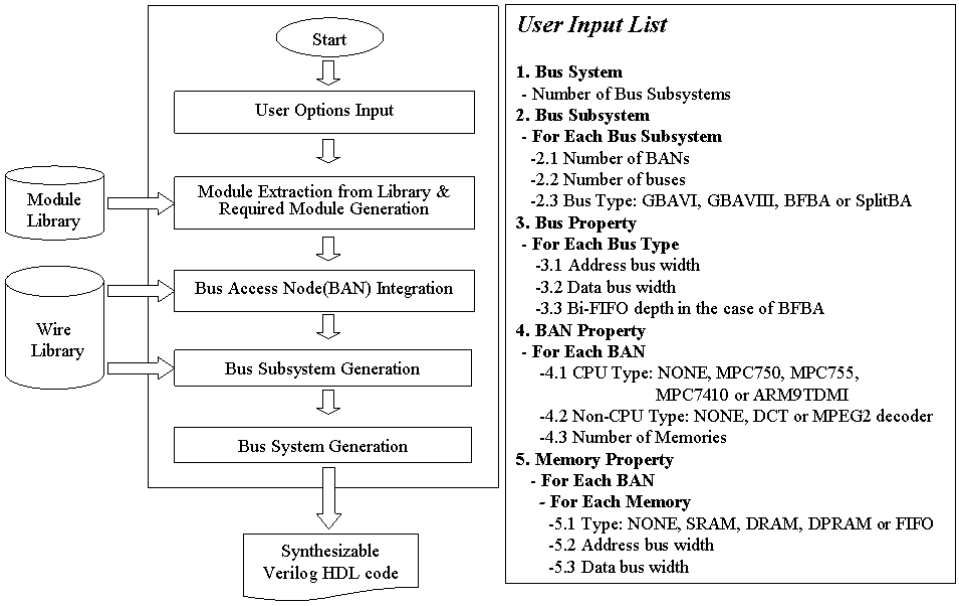


Figure 18. The Bus System Generation Sequence

The input sequence of user options is as follows. First, the user enters the number of Bus Subsystems for a Bus System and specifies the number of BANs and a bus type for each Bus Subsystem. For the bus types selected in the Bus Subsystem Property option, the user inputs Bus Property options for each bus type. The CPU Type or Non-CPU Type and the number of memories are inputs in the BAN Property option if the user wants to have these resources in a BAN. Finally, the user inputs Memory Property for each selected memory in the BAN Property if any memory is required in a BAN. How to use each option in a Bus System is shown in Example 9.

Example 9: User Input Option to Configure Bus System to Be Generated

A sequence of user input which specifies the BFBA Bus System shown in Figure 4 is as follows. The user first specifies the number of Bus Subsystems by entering a "1" in Bus System Property (user option 1 in Figure 18) and inputs "4" for the number of BANs (user option 2.1) that are BANs A, B, C and D in Figure 4. The user also inputs "BFBA" for the bus type (user option 2.2) to specify the Bus Subsystem. For the bus type BFBA, the user assigns the fields of Bus Property as follows: "32" for address bus width (user option 3.1), "64" for data bus width (user option 3.2) and "1024" for Bi-FIFO depth (user option 3.3). Next, the user inputs the fields of BAN Property for each BAN specified in the Bus Subsystem Property: "MPC755" for the CPU Type (user option 4.1), none for the Non-CPU Type (user option 4.2), and "1" for the number of memories (user option 4.3). Finally, the Memory Property is inputted for one memory in each BAN: "SRAM" for the memory type (user option 5.1), "20" for the address bus width (user option 5.2) and "64" for the data bus width (user option 5.3) for 8 MB SRAM in each BAN–BAN A, B, C or D. The four SRAMs are labeled SRAM_A, _B, _C and _D in Figure 4. The total amount of non-cache memories in Figure 4 is 32 MB. □

According to the user options shown in the right hand box of Figure 18, the user can customize the bus architecture of a Bus System in our bus synthesis tool BusSyn. As one of the customized bus architectures, the user might want to generate a mixed bus architecture by using the bus architectures we defined (e.g., GBAVI, BFBA and GBAVIII). Following Example 10 describes how to generate the customized bus architecture for a Bus System by the user options in BusSyn.

Example 10: Customized Bus Architecture (Hybrid)

Suppose a user wants to generate a combined bus architecture from the custom buses explained earlier: specifically, the combined bus architecture of both the Bi-FIFO bus from BFBA and the global bus from GBAVIII. As shown in Figure 6, we call the Bus System having the combined bus architecture as Hybrid. To generate such a Bus System, the user needs to input the user options shown in the right hand box of Figure 18 as follows.

First, the user enters “1” for the number of Bus Subsystems (user option 1). Then the Bus Subsystem property (user option 2) is specified as follows: “4” for the number of BANs (user option 2.1), “2” for the number of buses (user option 2.2), “BFBA” for the first bus type (user option 2.3), “GBAVIII” for the second bus type (user option 2.3). For the specified buses, BFBA and GBAVIII, the user enters their properties, respectively. In the BFBA bus property, address bus width (user option 3.1) is set to “32”, data bus width (user option 3.2) is inputted with “64”, and Bi-FIFO depth (user option 3.3) is entered with “1024”. And in the GBAVII bus property, address bus width (user option 3.1) is “32”, and data bus width (user option 3.2) is “64”. Next, the user enters each BAN Property; the user specified that the Bus Subsystem has four BANs in the user option 2. In each BAN (e.g., BAN A, B, C and D) Property, CPU type (user option 4.1) is set to “MPC755”, Non-CPU type (user option 4.2) is entered with “NONE”, and the number of memories (user option 4.3) is inputted with “1”. Based on the user input entered so far, each BAN has a single memory block, and thus total four memories are in the Bus Subsystem (see user option 2.1). Finally, Memory Property for each memory block (e.g., SRAM_A, _B, _C and _D shown in Figure 6) is entered as follows: “SRAM” for memory type (user option 5.1), “20” for address bus width (user option 5.2) for 8 MB size, “64” for data bus width (user option 5.3). □

```
BANGen(module name array, user option array, ban name){
   /* Step 1 */
   For each module name i in module name array, look up
      module name i in the Module Library and extract
      or generate the corresponding RTL Verilog code
      for each module i;
   /* Step 2 */
   Read wire information from Wire Library for each BAN;
   For each module i,
      /* Step 3 */
      Read port information from module i;
      /* Step 4 */
      While port j is not last port{
         While wire k is not last wire{
            Compare wire k information with port j information;
            If wire k information is matched with port j information,
               add wire k, wire connection k and port j into each list;
         }
      }
   }
   /* Step 5 */
   Instiantiate the required Modules;
   Write Verilog HDL code for a BAN;
}
```

Figure 19. Pseudo Code for BAN Generation

We introduce an algorithm to generate BAN(s) inside a Bus Subsystem. Figure 19 shows the pseudo code of the algorithm. In Step 1 of the code, Modules required in each BAN are either extracted from the Module Library or generated based on the user specified inputs. After extracting and generating the Modules for a BAN, wire information from the Wire Library is read in Step 2 and port information from each required Module is read in Step 3. Step 4 of the code uses the wire and port information not only to decide required wire connections between Modules but also to obtain exact I/O ports of the BAN to be generated. Both ends of each wire are examined if the wire needs to be connected to a Module and/or to a port of the BAN. Finally, in Step 5, BANGen() writes Verilog HDL code after

generating the instantiation code of the Modules based on wires, wire connections and the ports which are decided from the previous Steps. Example 11 shows a sample BAN generation based on the BAN generation algorithm.

Example 11: BAN Generation

For BAN A of BFBA shown in Figure 4, the required Modules are as follows: MPC755, MBI_SRAM, HS_REGS, CBI_MPC755, SRAM_A and Bi-FIFO. Step 1 of BANGen() in Figure 19 extracts the first three Modules (MPC755, MBI_SRAM and HS_REGS), and the others are generated according to the user options: for example, SRAM_A is generated according to SRAM parameters input by the user. In Step 2 of Figure 19, BANGen() reads wire information (e.g., w_name "w_addr", mx_name "SRAM_A" and mx_pname "sram_addr" in the format of Figure 15) from the Wire Library. In Step 3 of Figure 19, BANGen() obtains port and Module information (e.g, "sram_addr" and "SRAM_A") from each Module. Next, during Step 4, BANGen() compares the wire information, the port and Module information to decide which wires (e.g., "w_addr") need to be connected between the Modules. Finally, in Step 5 BANGen() generates the instantiation code of the required Modules with the chosen wires and writes Verilog HDL code describing BAN A. □

```
SubSysGen(BAN name array, user option array, subsystem name){
   /* Step 1 */
   Read wire information from Wire Library for each Bus SubSystem;
   For each BAN i in BAN name array,
      /* Step 2 */
      Read port information from BAN i if BAN i is different
         from BAN (i-1);
      /* Step 3 */
      While port j is not last port{
         While wire k is not last wire{
            Compare wire k information with port j information;
            If wire k information is matched with port j information,
               add wire k and wire connection k into each list;
         }
      }
   }
   /* Step 4 */
   Instiantiate required BANs;
   Write Verilog HDL code for a Bus Subsystem;
}
```

Figure 20. Pseudo Code for Bus Subsystem Generation

Bus Subsystem generation is done through procedure that instantiates generated BANs according to the Bus Subsystem Property and then wires the BANs together. The pseudo code of the algorithm for Bus Subsystem generation is shown in Figure 20. In Step 1 and Step 2 of Figure 20, wire information is read from the Wire Library, and port information is obtained from each BAN generated. Step 3 compares the ports and the wires so that required wires and wire connections between BANs are decided for a Bus Subsystem. In Step 4, SubSysGen() writes Verilog HDL code after generating the instantiation code of required BANs including the wires and wire connections which were decided in the previous Steps. Example 12 shows the procedure of Bus Subsystem generation based on pseudo

code shown in Figure 20.

Example 12: Bus Subsystem Generation

Consider the wire "w_data" and the port "fifo_dq_up" described in Example 8. To generate Figure 4 BFBA Bus Subsystem (which, as shown in Figure 4, is also a Bus System), SubSysGen() reads wire information (e.g., "w_data" and "fifo_dq_up") from the Wire Library in Step 1 of Figure 20 and obtains port information (e.g., "fifo_dq_up") from BAN A generated in Example 11 in Step 2. After that, SubSysGen() compares the port information with the wire information and decides which wires (e.g., "w_data") will be connected to the appropriate ports (e.g., "fifo_dq_up") of BAN A in Step 3. With the same method, in Step 3, SubSysGen() decides wires (e.g., "w_data") to be connected to the appropriate ports (e.g., "fifo_dq_dn") of BANs B, C and D. Finally, SubSysGen() instantiates BANs A, B, C and D with the wires and wire connections that are decided upon in Step 3 of Figure 20 and writes Verilog HDL code describing the Bus Subsystem in Step 4. □

A Bus Subsystem becomes a Bus System if the user wants a single bus architecture for the entire chip instead of multiple bus architectures in the SoC. A Bus System is also formed by connecting generated Bus Subsystems through bus bridges (BBs). As we have explained throughout this section, BusSyn can generate Modules as well as do a syntactic translation from high-level input description based on the user options to output synthesizable Verilog HDL code for a multiprocessor SoC.

VI. EXPERIMENTAL RESULTS

A. Application Examples

Five kinds of bus architectures for a multiprocessor SoC were generated using BusSyn and then simulated to evaluate the performance with three applications: a database example [22], an MPEG2 decoder [23][24] and an Orthogonal Frequency Division Multiplexing (OFDM) transmitter [25], which is used in wireless communications.

1) Database Example

As for the first application example to show the performance achievable with a custom Bus System, we have developed a database example having many tasks. This example is written in C code having 1700 lines. As shown in Figure 21, a database system may use several transactions to access objects in the other tasks. For example, in

Figure 21, task1 requests object O_2 in task2 and accesses O_2 after obtaining a lock. The lock is used to synchronize mutually exclusive accesses of the database objects in a multiprocessor system.

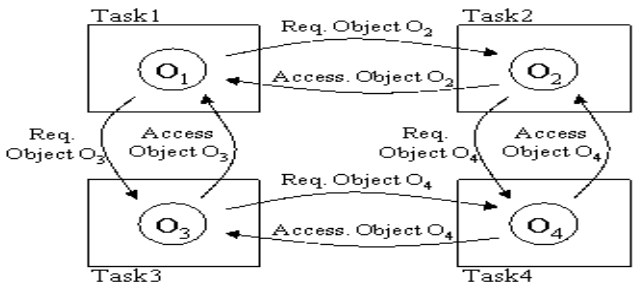


Figure 21. Transactions in Database Example

For the RTOS, we use ATALANTA RTOS Version 0.4 developed at Georgia Tech [26]. We have installed the RTOS on each BAN, and tasks assigned on each BAN are executed on top of the RTOS. We simulated this example in a variety of Bus Systems each with four processors. A total of forty-one tasks that form a subset of the database example run: eleven on the processor in BAN A and ten on each processor in each of other BANs in the examples of Bus Systems shown in Section IV.B. As shown in Figure 22 for the data transfer from a server to clients, a task in a server writes data requested from clients to a shared memory, and then tasks in the clients read the data from the shared memory and write the data to their local memories. Here, each task accesses one-hundred 32-bit words to or from the shared memory. With this database example, each one of Bus Systems has intensive bus traffic on its bus due to shared memory requests from each BAN, and thus we are able to observe a significant performance contrast among Bus Systems.

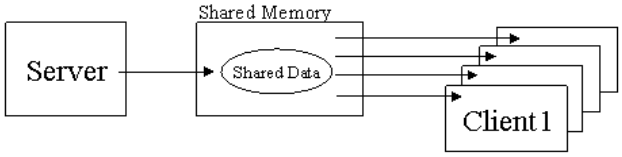


Figure 22. Data Transfer from Server to Clients

2) OFDM Transmitter

OFDM employs several parallel channels with low bit rates whose main lobes of carriers are orthogonal and side lobes of carriers are overlapping one another. This is an efficient way of carrying several subchannels in a fixed bandwidth. The subcarriers are not separated by bandwidth but rather overlap their side lobes with each other. The frequency spacing between the sub carriers is arranged such that they become orthogonal, and a Fast Fourier

Transform (FFT) is used for digital modulation/demodulation of each sub channel.

Figure 23 shows a simplified block diagram of an OFDM transmitter. First, the sub channels are modulated by an Inverse FFT (IFFT), and then a cyclic extension is added to avoid inter symbol interference caused by the physical channel. Here, the cyclic extension makes a packet of data be symmetric by attaching a block of head data to the data tail as shown in Figure 24.

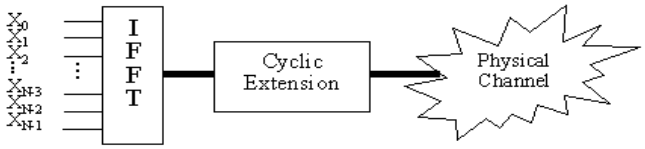


Figure 23. The Block Diagram of an OFDM Transmitter

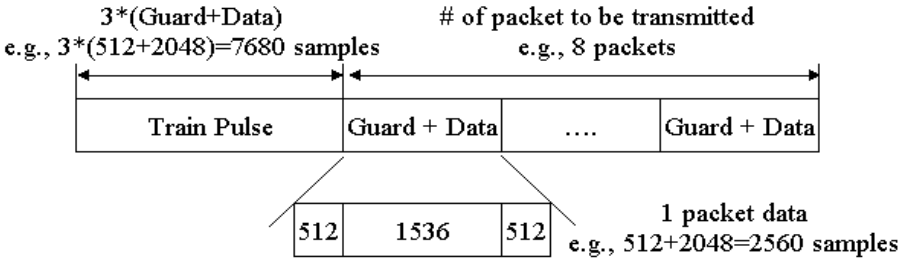


Figure 24. OFDM Data Format

Figure 24 shows the OFDM data format being transmitted. The OFDM data stream starts with a train pulse block, which allows a receiver to perform channel estimation and data synchronization, and guard and data packets follow the train pulse block. One packet of OFDM data we simulated here contains a 2048-complex valued sample and a 512-complex valued guard signal, where the size of guard data is usually a quarter of the data block.

Figure 25 shows the flow chart of the OFDM transmitter, which, in our example, is written in C code having 922 lines. The first three blocks (Initialization, Train Pulse Generation, and Symbol Generation) in Figure 25 are excluded in calculating throughput since these routines are executed only once at the startup. The End of Packet (EOP) loop controls data generation or data reading from an external device, which generates data to be transmitted. This EOP loop is repeated as many times as the size of the data packet; meanwhile, the outer loop is also repeated as many times as there are new data packets to be transmitted. The generated data is fed into the modulation block, which executes bit reversal, IFFT, normalization of IFFT output and insertion of the guard signal, sequentially.

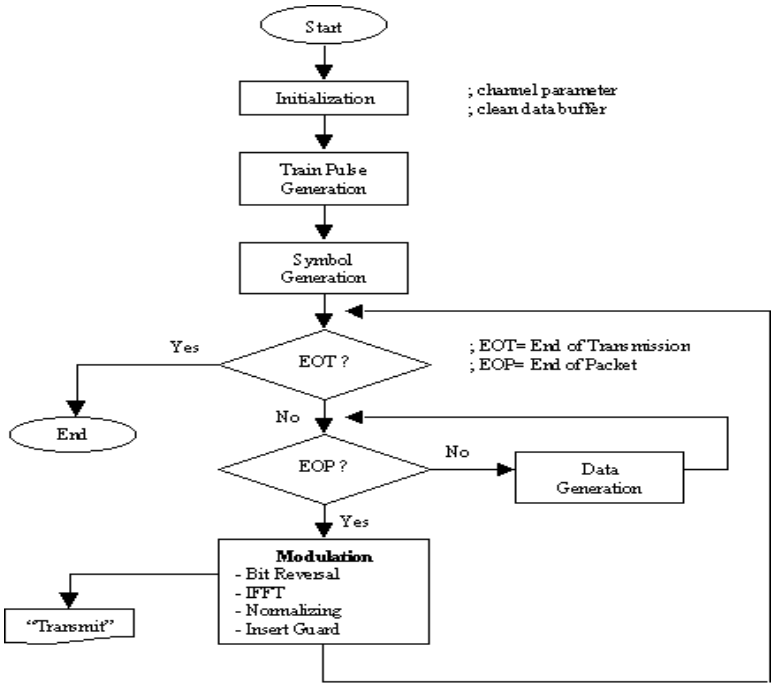


Figure 25. The Flowchart of the OFDM Transmitter

In OFDM, the function assignment to be processed in each BAN is decided after careful analysis of each function's computational load because a balanced load among BANs results in the fastest possible execution time. Table I shows a list of functions in OFDM and outlines the function assignment in each BAN. The assigned functions on BAN A seem heavier, but it is not the bottleneck of system performance because the first three functions listed for BAN A (italicized in Table I) are executed only once. Only data generation, symbol mapping and bit reversal functions are iterated in BAN A. The function on BAN B, IFFT, unfortunately is difficult to split up due to the structure of the IFFT.

Table I.
The Function Assignment in Each BAN

Function Group & Assigned BAN	Functions in OFDM Transmitter
E (BAN A)	*Initialization (channel parameters, etc)* *Train Pulse Generation* *Symbol Generation* Data Generation and Symbol Mapping Bit Reverse for Inverse FFT
F (BAN B)	Inverse FFT
G (BAN C)	Normalizing Inverse FFT
H (BAN D)	Normalization Insertion of Guard Signal Data Output

Note: Italicized functions are executed only once when starting OFDM system.

Figure 26 describes the computation in each processor according to programming styles: pipelined parallel algorithm (PPA) and functional parallel algorithm (FPA). Here E, F, G and H in Figure 26 indicate function groups shown in Table I. We programmed the OFDM transmitter algorithm in both PPA style and FPA style to see how the styles affect performance. The FPA style proved to be faster in most cases because of a more balanced load on each BAN. One packet of OFDM data here contains a 2048-complex valued samples and a 512-complex valued guard signals.

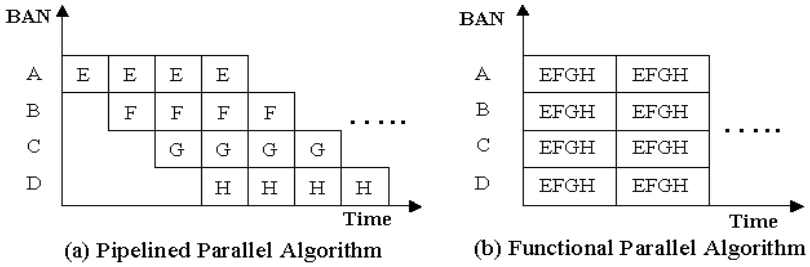


Figure 26. Software Programming Style in OFDM

3) MPEG2 Decoder

MPEG2 video is an ISO/IEC standard that specifies the syntax and the semantics of encoded video bit streams [23]. The data streams include parameters such as bit rates, picture sizes, resolutions. We borrowed an MPEG2 decoder code, which is written in C code having 8788 lines, from the MPEG Software Simulation Group [24] to evaluate the generated Bus Systems.

Figure 27(a) shows input video frames, and Figure 27(b) shows the functional parallel processing of the frames on each BAN. In the video stream data it is assumed that each Intra frame (I) is followed by Predictive frame (P) as shown in Figure 27(a), and a Group Of Picture (GOP) is composed of two frames (I and P). Each frame size is specified with very small picture, 16 pixels by 16 pixels, because of the limitation of simulation speed.

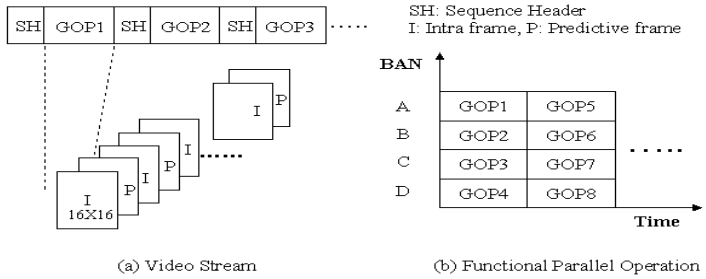


Figure 27. Input Video Stream and Functional Parallel Operation

For the MPEG2 decoder, we exclusively used the FPA style because it yielded the fastest results. As shown in Figure 27(b), each GOP is assigned to a particular BAN for functional parallel operation. All video frames fed to BAN A from an input source are distributed to each BAN, and each decoded frame is handed over to BAN D at the end. Here BAN A and BAN D perform not only MPEG2 decoding but also raw data input and decoded data output, respectively.

B. Experimental Environment

As shown in Figure 28, BusSyn takes the user input as described in Section V.B and outputs synthesizable Verilog HDL code for the specified custom Bus System. For the Bus System simulation, we use Seamless CVE, a hardware/software co-verification tool, and X-Ray debugger from Mentor Graphics [27] together with VCS, a Verilog HDL simulator from Synopsys [28]. We use the Synopsys Design Compiler to synthesize the Verilog HDL code to logic gates. For this environment, we use a Sun workstation Ultra 60 having two 450MHz UltraSPARC II processors and 2 GB of memory.

In this experiment, we use four MPC755s in Seamless CVE; each BAN has one MPC755 with 100 MHz external clock, SYSCLK. The maximum frequency of SYSCLK, which dictates the maximum bus speed, is limited to 100 MHz in the PowerPC Hardware Specification (note that the internal clock speed can be much faster, e.g., 500 MHz) [29]. However, our results are equally applicable to much faster bus clock speeds.

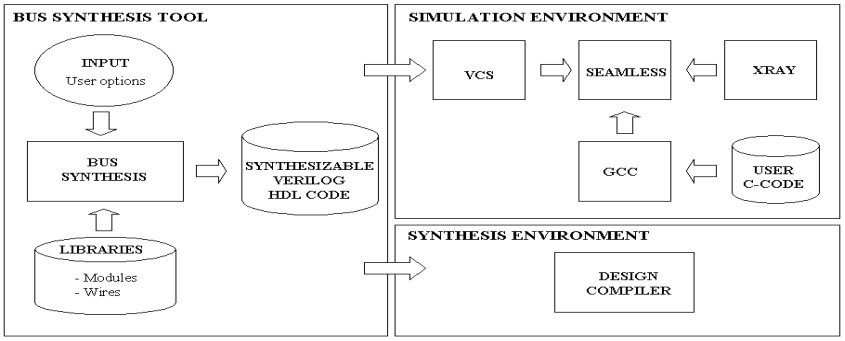


Figure 28. Experimental Environment

C. Comparison of Result

With the generated Bus Systems (shown in Figures 3, 4, 5, 6 and 7) and hand-designed examples of CCBA and GGBA (shown in Figures 8 and 9), we evaluate the performance and verify the operation of each Bus System with an

OFDM transmitter, an MPEG2 decoder and a database example. The Bus Systems have 32 MB total of memory, respectively, and each processor (MPC755) embedded in each Bus System has 32KB of L1 I-cache and 32KB of D-cache.

Table II.
Evaluation Results in OFDM Transmitter

Case	Bus System	Throughput [Mbps]	Software Programming Style
1	BFBA	2.6504	PPA
2	GBAVI	2.1087	PPA
3	GBAVIII	4.5599	FPA
4		2.2567	PPA
5	Hybrid	4.5599	FPA
6		2.6504	PPA
7	SplitBA	5.1132	FPA
8	GGBA	4.3913	FPA
9		2.1880	PPA

Note: 1. PPA: Pipelined Parallel Algorithm, FPA: Functional Parallel Algorithm
2. Data: 2048 complex samples and 512 guard complex samples per packet
3. Each Bus System having four PowerPCs supports instruction and data cache operations

Table II shows the results of our evaluation using an OFDM transmitter that in our example has 922 lines of C code for the algorithm implementation and 696 lines of assembly code for processor runtime initialization and APIs. The operation of BFBA and GBAVI is well matched to the PPA style because BFBA and GBAVI only have data transfer mechanisms between BANs instead of having a memory shared among all BANs. SplitBA is composed of two Bus Subsystems connected with a Bus Bridge, and the two Bus Subsystems operate independently. Therefore, in SplitBA, it is more reasonable to use the FPA style. SplitBA (Case 7 in Table II) using the FPA style shows the best performance among the Bus Systems in our example: OFDM transmission reaches a rate of 5.1132 Mbps, 16.44% faster than GGBA, which we take as representative of a typical commercial bus. We can see in Table II that the throughput of each Bus System is significantly affected by the bus types we described and programming style (PPA vs. FPA):

(A) In software programming style, FPA beats PPA in the OFDM transmitter application (e.g., Case 3 vs. 4 and Case 8 vs. 9 in Table II). The reason is that, for OFDM, FPA balances the computational load better than PPA does.

(B) Bus Systems using a shared memory for program and local data (e.g., GGBA) require more memory arbitration time than in Bus Systems having separated local memories for program and local data for each BAN (e.g., GBAVIII) does. This arbitration time difference explains why GBAVIII outperforms GGBA.

(C) SplitBA relieves bus traffic congestion due to shared memory requests from each BAN. The reason is the Bus System has spited bus architecture, and thus each arbiter in each Subsystem deals with only half number of total memory requests from each BAN. With this reason, SplitBA beats GGBA in our example (Case 7 vs. 8).

(D) A fast data transfer method between BANs such as Bi-FIFO of BFBA contributes to the performance improvement observed for the PPA style (e.g., Case 1 > Case 4 > Case 9 > Case 2, in throughput).

Table III.
Evaluation Results in MPEG2 Decoder

Case	Bus System	Throughput [Mbps]
10	BFBA	0.8594
11	GBAVI	0.8271
12	GBAVIII	1.1444
13	Hybrid	1.1650
14	CCBA	1.0083

[Note] 1.Picture size: 16 x 16
2.All Bus Systems run on four PowerPCs have Functional Parallel Algorithm

In our application, MPEG2 decoder has 8788 lines of C code for its algorithm and 697 lines of assembly code for initialization routines and APIs. In the results shown in Table III, Hybrid (Case 13) shows the best performance because Hybrid exploits both BFBA's and GBAVIII's bus features such as (i) fast data transactions between adjacent BANs using Bi-FIFOs and (ii) global data accesses in global memory from all BANs. The results also show that Hybrid and GBAVIII outperform CCBA due to faster arbitration time in data read operations (3 cycles as compared to 5 in CCBA). In Table III, BFBA and GBAVI perform poorly because the data to be processed in each BAN has to be passed from BAN A to each BAN sequentially. Note that Hybrid, generated by BusSyn, outperforms CCBA by 15.54% in this example.

Table IV
Evaluation Results in a Database Example

Case	Bus System	Execution Time [ns]
15	GGBA	2,241,100
16	SplitBA	1,317,804

[Note] 1. Each Bus System is composed of 1 server task and 40 client tasks.
2. Each task accesses one-hundred data to or from the shared memory.

In the database application example, for multi-thread operation, we employ the ATALNATA RTOS [26], which requires a shared memory. We can support the use of the RTOS in GBAVI and BFBA; however, in this paper, we do not simulate these Bus Systems with this application because the current versions of these Bus Systems do not have such a shared memory. Furthermore, this application is an example using only a shared memory without using local

memories in the data transaction between the server and the clients. Therefore, when we assume that, in this example, we do not use a Bi-FIFO bus nor local memories, Bus Systems having a global memory and single global bus (e.g., GBAVIII, Hybrid and GGBA) have the same performance factor due to the same bus components. For that reason, we use one of these Bus Systems, GGBA (see Figure 9), as a baseline of performance comparison and compare the performance only with SplitBA (see Figure 7) in this application. The performance of SplitBA is improved over GGBA because of following two reasons. The first one is that SplitBA has better bus topology (e.g., split global bus connected by a bus bridge (BB)) than GGBA have, and thus bus traffic due to the shared memory requests is lessened. The second one is that SplitBA has shared bus architecture in each Bus Subsystem so that all clients can easily access object data from the server. This example has total of 1700 lines of C code for the algorithm and runs on top of the ATALANTA RTOS. A total of forty-one tasks are executed for clients and a server; BAN A in Figure 7 has one server task and ten client tasks, and the other BANs in the figure each have ten client tasks, where each task accesses one-hundred words (32 bits per data word) to or from a shared memory in each Bus System. In the experiment of the database example shown in Table IV, SplitBA (Case 16 in Table IV) outperforms GGBA (Case 15 in Table IV) with a 41% reduction in application execution time.

Table V.
Generation Time and Gate Count in the Generated Bus Systems

Bus System	1 processor		8 processors		16 processors		24 processors	
	Time [ms]	Gate count	Time [ms]	Gate count	Time [ms]	Gate count	Time [ms]	Gate counts
BFBA	509	800	534	6,401	546	12,793	578	19,188
GBAVI	417	872	432	6,899	457	13,751	506	21,256
GBAVIII	513	2,070	542	14,746	563	30,798	590	48,395
Hybrid	763	2,973	859	21,869	928	44,847	983	69,697
SplitBA	N/A	N/A	413	4,297	440	8,605	491	16,110

[Note] Time: Bus generation time, N/A: Not Applicable
Gate count: NAND2 gate count in TSMC 0.25μm standard cell library

Table V shows the generation time for the Bus Systems generated using BusSyn. Table V also shows the gate counts of the Bus System logic after synthesizing the logic using the LEDA TSMC 0.25μm standard cell library with the Synopsys Design Compiler. Since our goal is cycle accurate hardware/software cosimulation, we do not include layout parameters such as wire area in our area estimates. Thus, after using our tool, extra work is required to obtain layout accurate area and timing estimates for the final chip implementation. BusSyn can generate a Bus System having any number of processors, but the table shows Bus Systems having a maximum of 24 processors. In the generation time column, each Bus System shown in Table V takes less than one second to generate using BusSyn. Our experience is that porting GGBA or CCBA to our application examples, on the other hand, took about one week. The

week was spent understanding signal functions of the processors and the modeling of required Modules and their interfaces. Note that BusSyn achieves performance superior to the hand design of GGBA and CCBA; furthermore, the custom bus architecture is designed in a matter of seconds instead of weeks. This means we have a major benefit that is fast design space exploration of bus architectures across performance influencing factors such as bus types, processor types and software programming style resulting in a system having higher performance. This goal is accomplished through BusSyn, which allows the user to easily design a custom Bus System in a matter of seconds.

VII. CONCLUSION

In this paper, we have described a methodology to generate custom Bus Systems for multiprocessor SoC designs. We designed a bus synthesis tool BusSyn by exploiting this methodology. Using BusSyn, we have generated five different Bus Systems as examples: BFBA, GBAVI, GBAVIII, Hybrid and SplitBA. In Section VI, the Bus Systems are evaluated according to their performance and are verified in operation with three applications: an OFDM transmitter, an MPEG2 decoder and a database example. Our methodology gives us a great benefit in fast design space exploration of bus architectures across performance influencing factors such as bus types and software programming style. We showed that BusSyn achieves performance superior to the hand design of a simple GGBA and CCBA, but in a matter of seconds instead of weeks for the hand design.

ACKNOWLEDGMENT

This research is funded by the State of Georgia under the Yamacraw initiative and by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, SUN and Synopsys.

REFERENCES

[1] K. Ryu, E. Shin and V. Mooney, "A Comparison of Five Different Multiprocessor SoC Bus Architectures," *Proceedings of the EUROMICRO Symposium on Digital Systems Design* (EUROMICRO' 01), pp. 202-209, September 2001.

[2] K. Ryu and V. Mooney, "Automated Bus Generation for Multiprocessor SoC Design," *Design, Automation and Test in Europe* (DATE' 03), pp. 282-287, March 2003.

[3] IBM, "CoreConnect Bus Architecture," [Online]. Available: http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, 2002.

[4] ARM, "AMBA Specification Overview," [Online]. Available: http://www.arm.com/armtech.nsf/html/AMBA? OpenDocument&style= AMBA, 2002.

[5] B. Cordan, "An Efficient Bus Architecture for System-on-a-Chip Design," *Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 623-626, May 1999.

[6] Sonics, "Sonics µNetwork Technical Overview," [Online]. Available: http://www.sonicsinc.com/sonics/support/documentation/whitepapers/data/Overview.pdf, 2002.

[7] B. Dittenhofer, "Connecting Multi-Source IP to a Standard On Chip Architecture," [Online]. Available: http://www.palmchip.com/pdf/CP-9248P.pdf.

[8] M. Gasteier and M. Glesner, "Bus-Based Communication Synthesis on System-Level," *Proceedings of 9th International Symposium on System Synthesis*, pp. 65-70, November 1996.

[9] R. A. Bergamaschi and W. Lee, "Designing Systems-on-chip using cores," *Proceedings of the 38th Design Automation Conference* (DAC'00), pp. 420-425, June 2000.

[10] P. Chou, R. Ortega and G. Borriello, "IPCHINOOK: An Integrated IP-based Design Framework for Distributed Embedded Systems," *Proceedings of the 37th Design Automation Conference*, June 1999.

[11] D. Lyonnard, S. Yoo, A. Baghdadi and A. A. Jerraya,, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," *Proceedings of the 39th Design Automation Conference*, pp. 518-523, June 2001.

[12] F. Gharsalli, D. Lyonnard, S. Meftali, F. Rousseau and A. A. Jerraya, "Unifying Memory and Processor Wrapper Architecture in Multiprocessor SoC Design," *Proceedings of the International Symposium on System Synthesis* (ISSS'02), pp. 26-31, October 2002.

[13] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi and A. Jerraya, "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign* (CODES'01), pp. 195-200, April 2001.

[14] F. Gharsalli, S. Meftali, F. Rousseau and A. Jerraya, "Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC," *Proceedings of the 40th Design Automation Conference* (DAC'02), pp. 596-601, June 2002.

[15] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya and M. Diaz-Nava, "Component-Based Design Approach for Multicore SoCs," *Proceedings of the 40th Design Automation Conference* (DAC'02), pp. 789-794, June 2002.

[16] W. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier and M. Diaz-Nava, "Multiprocessor SoC Platforms: A Component-Based Design Approach," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 62-63, November 2002.

[17] G. Nicolescu, S. Yoo, A. Bouchhima and A. A. Jerraya, "Validation in a Component-Based Design Flow for Multicore SoCs," *Proceedings of the International Symposium on System Synthesis* (ISSS'02), pp. 162-167, October 2002.

[18] Pentek, "Operating manual for Model 4290 and 4291," [Online]. Available: http://www.pentek.com/Products/Detail.cfm?Model=4291, 2002.

[19] C. Hsieh and M. Pedram, "Architectural Energy Optimization by Bus Splitting," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 4, pp. 408-414, April 2002.

[20] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design, the Hardware and Software Interface*, CA: Morgan Kaufmann Publishers, Inc, 1994.

[21] Ibid., pp. 551-558.

[22] M. A. Olson, "Selecting and Implementing and Embedded Database System," *IEEE Computer*, pp. 27-34, September 2000.

[23] K. R. Rao and J. J. Hwang, *Technique & Standards for Image Video & Audio Coding*, NJ: Prentice Hall PTR, 1996.

[24] MSSG, "mpeg2encoder / mpeg2decoder, " [online]. Available: http://www.mpeg.org/MPEG/MSSG/Codec/readme.txt, 1996.

[25] D. Kim and G. L. Stüber, " Performance of Multiresolution OFDM on Frequency-selective Fading Channels," *IEEE Transaction on Vehicular Technology*, vol. 48, no. 5, pp. 1740-1746, September 1999.

[26] S. Di-Shi, D. Blough and V. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications," Georgia Institute of Technology, College of Computing, Atlanta, GA. Tech. Rep. GIT-CC-02-19, March 2002, Available: http://www.cc.gatech.edu/tech_reports/.

[27] Mento Graphics, "Seamless Hardware/Software Co-Verification," [Online]. Available: http://www.mentor.com/seamless/datasheets/seamless_ds.pdf, 2002.

[28] Synopsys, "VCS data sheet," [Online]. Available: http://www.synopsys.com/products/simulation/vcs_ds.html, 2002.

[29] Motorola, "MPC 755A RISC Microprocessor Hardware Specification," [Online]. Available: http://e-www.motorola.com/webapp/sps/site/prod_summary.jsp?code=MPC755&nodeId= 04M0ylg8ZH6, 2002.