Automated Bus Generation for Multiprocessor SoC Design

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

Abstract

The performance of a system, especially 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 (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 two applications: an Orthogonal Frequency Division Multiplexing (OFDM) wireless transmitter and an MPEG2 decoder. 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 achieve superior performance when compared to a simple General Global Bus Architecture (GGBA) (e.g., 16.44% performance improvement in the case of OFDM transmitter) or when compared to the CoreConnect Bus Architecture (CCBA) (e.g., 15.54% peformance improvement in the case of MPEG2 decoder). 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.

1. Introduction

System-on-a-Chip (SoC) opens up new opportunities for hardware/software codesign. 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 one to take advantage of increased bus speeds and widths. Thus, an efficient bus architecture with optimal arbitration for reducing contention plays an important role in maximizing the performance of an SoC.

One issue for an SoC designer to consider is how to exchange data among the Processing Elements (PEs) in the SoC, e.g., should there be one bus or multiple buses and where should memory elements be placed? A second 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 Verilog 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) cores for a multiprocessor SoC. Using this methodology, five different bus systems are generated as examples in synthesizable Verilog HDL: 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). Bus system performance is evaluated using two applications: an Orthogonal Frequency Division Multiplexing (OFDM) wireless transmitter and an MPEG2 decoder. We will 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 will compare the performance with a simple General Global Bus Architecture (GGBA) and an industry standard on-chip bus (CoreConnect from IBM [1]), and show a 16% improvement with a customized bus architecture.

2. Related Work

Most SoC bus designs are based on Intellectual Property (IP) cores stitched together with various forms of data, address and control links. There are several efforts to make SoC bus systems from industry: Coreconnect from IBM [1] and AMBA from ARM [2]. FastForward for SiliconBackplane [3] and Connection Kit for CoreFrame [4] allow a designer to integrate IP modules and result in reduced design time for a bus system for an SoC. We take GGBA and CoreConnect as representative examples of these industry buses.

Gasteier *et al.* [5] describe the automatic generation of a communication topology using scheduling of data transfer operations to reduce the cost of a bus architecture. However, they only show support for a single type of bus topology (a single global bus topology). We, on the other hand, support multiple bus types.

Bergamaschi *et al.* [6] present automating the design of SoC using IP cores connected via CoreConnect. In their methodology for assembling IP, their approach checks the

compatibility of IP inputs/outputs and generates wires to connect the IP cores. Again, we, on the other hand, support a wider variety of bus types and architectures than CoreConnect.

Lyonnard *et al.* [7] introduce a design flow for the generation of application-specific multiprocessor architecture. Nicolescu *et al.* [8] and Gharsalli *et al.* [9] present a component-based design flow for a heterogeneous and multicore SoC, where the flow introduces a systematic method of wrapper generation for multicore SoC design. However, in the communication network design [7, 8 and 9], the flows presented only supports generation of a single bus type for the system (e.g., a shared bus or a point-to-point interconnection). We provide more flexible bus architecture templates such as supporting multiple and heterogeneous bus architectures (e.g., GBAVI, GBAVIII, BFBA, Hybrid, and SplitBA) in a system, and various optimized wrappers (e.g., CPU- bus interface, memory-bus interface and generic bus interface) generated or extracted from a module library file for the ease of interface and integration between modules.

3. 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.

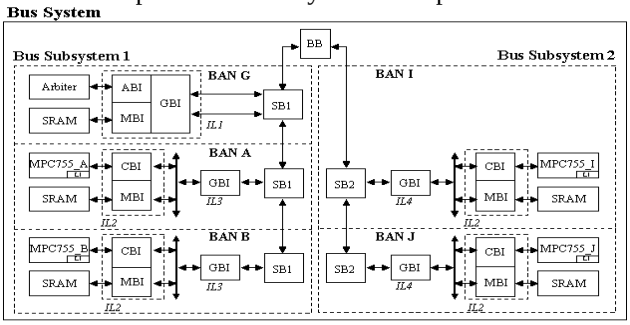
Definitions

1) Processing Element (PE): hardware that performs algorithmic processing – usually a CPU but may also be dedicated or reconfigurable logic. Currently, we support two types: MPC750 and MPC755.
2) Bus Bridge (BB): a controllable connection point between two buses – if the BB is enabled, the two buses are fully connected, otherwise the two buses are disconnected.
3) Global Bus Architecture (GBA): a type of bus architecture where BBs may be used to connect different sections of the bus.
4) Bi-FIFO Bus Architecture (BFBA): a type of bus architecture where bidirectional FIFOs are used to transmit and receive data between adjacent PEs.
5) 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).
6) Bus Access Node (BAN): processing or memory hardware together with associated bus access hardware and SB(s).
7) Module: one of BB, SB, Arbiter, SRAM or IL (in this paper, a PE is not a Module but instead is an IP core), where IL is Interface Logic that will be explained in more detail in Section 4. Note that it is possible to extend the definition of Module to include newly designed hardware units. For this paper, however, the definition given for Module suffices.
8) Bus Subsystem: one or more BANs connected together and using the same bus architecture or the combination of different bus architectures (in our case, either GBA, BFBA, or the combination of GBA and BFBA).
9) Bus System: one or more Bus Subsystems connected together.

4. Methodology for Bus System Generation

4.1. Bus System Structure

First, we will describe in more detail the bus components 1) through 9) defined in Section 3. Figure 1 shows a hierarchical example of a multiprocessor Bus System: a Bus System has Bus Subsystems, each Bus Subsystem includes BANs that are composed of PEs (MPC755s) and Modules, and the Bus Subsystems are connected with a Bus Bridge. The hierarchical definition allows a Bus System to have flexible and scaleable bus architecture. In addition to PEs (e.g., MPC755) and memories (e.g., SRAM) in the BANs of Figure 1, there are more Modules specified as Interface Logics (ILs): CPU or PE to Bus Interface (CBI), Memory to Bus Interface (MBI), and Generic Bus Interface (GBI). With these interface Modules, different BANs can have different types of PEs and memories because CBI and MBI Modules adapt the interface between the PE or memory and local bus respectively. Similarly, GBI also gives flexibility in being able to select various types of buses in a Bus Subsystem: GBA (GBAVI or GBAVIII, to be described in Section 4.2) and BFBA. Each BAN can access any other BAN's memory through the SBs. The repetition of the BANs makes a Bus Subsystem be a scalable structure and lets the multiprocessor Bus System be implemented fast.



BAN: Bus Access Node, IL: Interface Logic, SB: Segment of Bus, BB: Bus Bridge, MBI: Memory-Bus Interface, CBI: CPU/PE-Bus Interface, GBI: Generic Bus Interface, ABI: Arbiter-Bus Interface,

Figure 1. Bus System Example

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 part of a BAN: for example, the large SRAM in BAN G in Figure 1.

4.2. Bus System Examples

In this section, we show five custom Bus Systems to be generated by BusSyn automatically: BFBA, GBAVI, GBAVIII, Hybrid, and SplitBA. All Bus System examples

shown in Figures 2, 3, 4 and 5 have four processors and 32MB total of memory (all examples have approximately the same chip area because the area of the bus logic and wires is much smaller than CPU and memory area); however, BusSyn can generate a Bus System having any number of processors and any sizes of memories according to the user input that will be described in Section 4.3.2.

First, we give an 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 2(a) is a kind of global bus architecture (GBA), but the global bus is segmented with BBs separating each BAN. As shown in Figure 2(b), 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 [10]. The operations of GBAVI and BFBA are discussed in detail in [15].

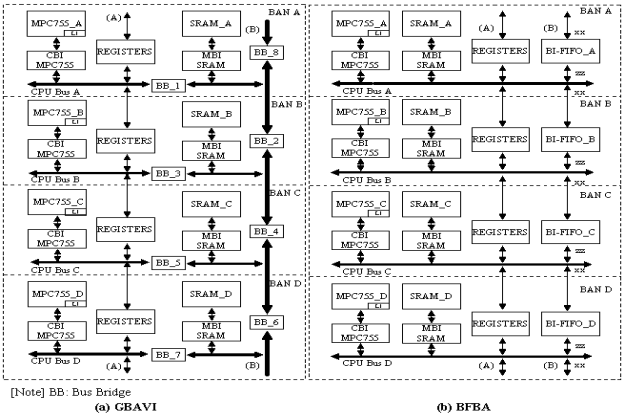


Figure 2. Diagrams of GBAVI and BFBA

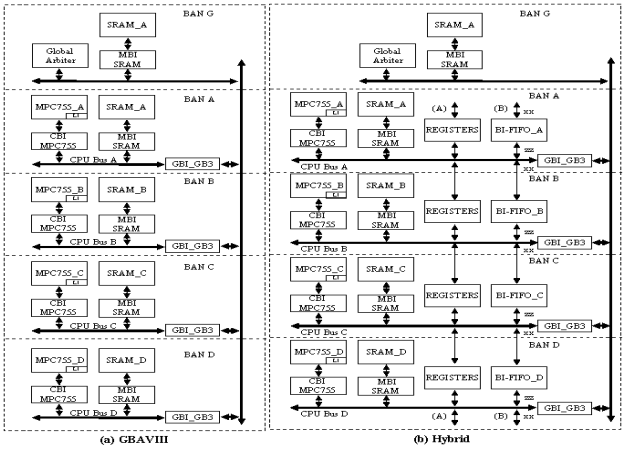


Figure 3. Diagrams of GBAVIII and Hybrid

GBAVIII shown in Figure 3(a) is a global bus architecture (GBA) having a local program and data memory, a global arbiter, and a global memory.

One example of a possible Bus System specified by Hybrid is the combination of BFBA and GBAVIII, as shown in Figure 3(b). This combination allows this bus architecture to have advantages of both the BFBA and the GBAVIII architectures: supplying a Bi-FIFO data transfer method between adjacent BANs and having a global memory area that can be accessed from all BANs.

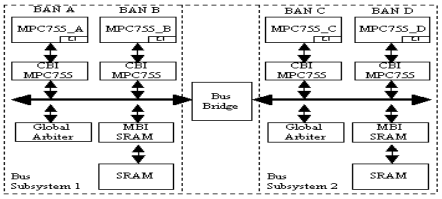


Figure 4. Diagram of SplitBA

Figure 4 shows SplitBA that is composed of two Bus Subsystems that have two MPC755s and a global memory respectively. The Bus Subsystems are connected through a bus bridge to exchange data between them.

GGBA and CCBA are shown in Figure 5. These bus architectures are a baseline for performance comparisons with other Bus Systems.

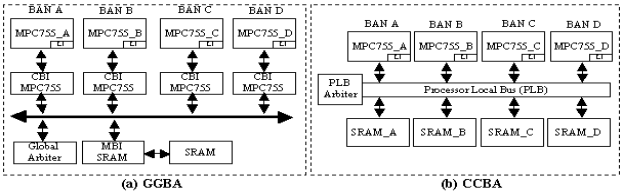


Figure 5. Diagrams of GGBA and CCBA

4.3. Bus System Generation

4.3.1. Libraries for Module Repository and Wiring.

BusSyn uses two libraries. One is a Module Library that is used for Modules to be configured in a BAN, and the other is a Wire Library for connecting the Modules in a BAN and for connecting the BANs in a Bus Subsystem.

The Module Library describes the input/output ports and behavior of each module in Register Transfer Level (RTL) Verilog. The library, for example, contains the following components: MPC755_IF as a processor core interface, MBI_SRAM for interface between memory and bus, SBFBA for a segment of bus of BFBA and GBI_BFBA for a generic bus interface.

The Wire Library contains all possible combinations of legal connections between bus elements (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 6, and there are several fields to specify connection information: wire name (w_name), wire width (w_width), module x name (mx_name), port name in module x (mx_pname), most significant bit (mx_wmsb) and least significant bit(mx_wlsb), where the 'x' is 1 or 2. Example 1 shows how to use the Wire Library.

```
%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 6. The Format in Wire Library

Example 1: As an example of wire connection in a BAN, MBI_SRAM supplies an address to SRAM_A in BAN A of Figure 2(b) BFBA. To specify, according to Figure 6, 20-bit

address wire 'w_addr[19:0]' between SRAM_A and MBI_SRAM, 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 22 3
%endwire
```

Another example of a wire connection between BANs in a Bus Subsystem is data bus wire 'w_data[63:0]' between BAN A and BAN B in Figure 2(b). To indicate the wire, the wire is specified as follows:

```
%wire subsys_bfba
w_data 64 BAN fifo_dq_dn 63 0 BAN fifo_dq_up 63 0
%endwire □
```

4.3.2. The Bus System Generation Sequence.

To design a custom bus, the user first inputs options that are described in the right hand side box of Figure 7. These options are input constraints used to generate a custom Bus System. The input sequence of user options is described in more detail in [15].

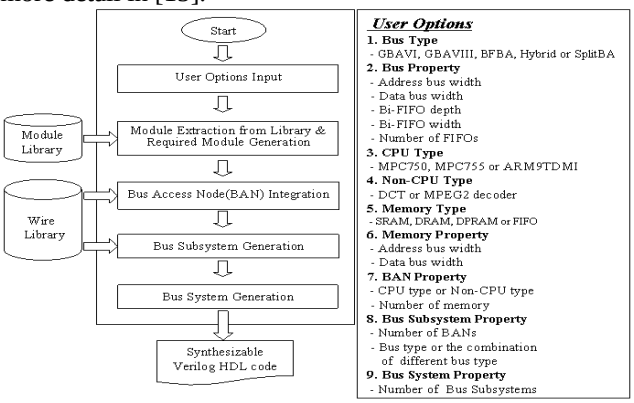


Figure 7. The Bus System Generation Sequence

```
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 8. Pseudo Code for BAN Generation

Figure 8 shows the pseudo code for BAN generation after the user options are specified. In Step 1 of the code, Modules required in each BAN are either extracted from the Module Library or generated. 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 Module-to-Module and Module-to-port but also to obtain exact I/O ports of the BAN to be generated, where both ends of a 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 instantiating the Modules and wiring the instantiated Modules, based on wires, wire connections and the ports that are decided from the previous Steps.

Example 2: Consider the wire 'w_addr' described in Example 1. For BAN A of BFBA shown in Figure 2(b), the required Modules are as follows: MPC755 Interface, MBI_SRAM, REGISTERS, CBI MPC755, SRAM_A and Bi-FIFO. Step 1 of BANGen() in Figure 8 extracts the first three Modules (MPC755 Interface, MBI_SRAM and REGISTERS), and the others are generated according to the user options: for example, SRAM parameters for SRAM_A. In Step 2 of Figure 8, BANGen() reads wire information (e.g., w_name 'w_addr', mx_name 'SRAM_A' and mx_pname 'sram_addr' in the format of Figure 6) from the Wire Library. In Step 3 of Figure 8, 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() instantiates the required Modules with the decided wires and writes Verilog HDL code describing BAN A. □

Bus Subsystem generation is done through an instantiation procedure of generating BANs according to the Bus Subsystem Property and a wiring procedure to integrate the BANs together. The pseudo code of the algorithm for the Bus Subsystem generation is shown in Figure 9. In Step 1 and Step 2 of Figure 9, wire information is read from the Wire Library, and port information is obtained from each BAN to be 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 instantiation of required BANs and wiring the instantiated BANs, based on the wires and wire connections that are decided in the previous Steps.

```
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 9. Pseudo Code for Bus Subsystem Generation

Example 3: Consider the wire 'w_data' and the port 'fifo_dq_up' described in Example 1. To generate Figure 2(b) BFBA's Bus Subsystem (which, as shown in Figure 2(b), is also a Bus System), in Step 2 of Figure 9 SubSysGen() reads wire information (e.g., 'w_data' and 'fifo_dq_up') from the Wire Library, obtains port information (e.g., 'fifo_dq_up') from BAN A generated in Example 2, 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 9 and writes Verilog HDL code describing the Bus Subsystem in Step 4. □

A Bus Subsystem can become 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 made by using BBs to connect generated Bus Subsystems. 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 in order to output synthesizable Verilog HDL code for a multiprocessor SoC.

5. Application Examples

Five kinds of bus architectures for a multiprocessor SoC were generated using BusSyn and then simulated to evaluate the performance with two applications: MPEG2 decoder [11] and Orthogonal Frequency Division Multiplexing (OFDM) [12], which is a wireless communication protocol.

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

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

Table 1. Function Assignment for each BAN in OFDM

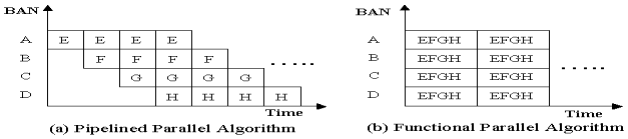


Figure 10. Software Programming Style in OFDM

Table 1 shows a list of functions in OFDM, and Figure 10 describes the computation in each processor according to programming style: pipelined parallel algorithm (PPA) and functional parallel algorithm (FPA). 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. One packet of OFDM data here contains a 2048-complex valued sample and a 512-complex valued guard signal. For the MPEG2 decoder, we exclusively used the FPA style because it yielded the fastest results. We used a very small picture (16 pixels x 16 pixels) because of the limitation of simulation time. The details of these applications are in [15].

6. Experimental Results

6.1. Experimental Environment

For the Bus System simulation, we use Seamless/CVE, a hardware/software co-simulation tool from Mentor Graphics [13], together with VCS, a Verilog HDL simulator from Synopsys [14]. We use the Synopsys Design Compiler to synthesize the Verilog HDL code to logic gates. More environment detail, including processors we used, is available in a technical report [15].

6.2. Comparison of Results

With the generated Bus Systems, Figures 2, 3 and 4, and hand-designed examples of GGBA and CCBA (shown in Figure 5), we evaluate the performance and verify the operation of each Bus System with the OFDM transmitter and MPEG2 decoder.

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. All Bus Systems run on four PowerPCs support instruction and data cache operations

Table 2. Evaluation Results in OFDM Transmitter

Table 2 shows the results of our evaluation using OFDM. 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. So, in SplitBA, it is more reasonable to use the FPA style. SplitBA (Case 7 in Table 2) 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 2 that the throughput of each Bus System is significantly affected by the bus types and programming style (PPA vs. FPA):

(1) In software programming style, FPA beats PPA in the OFDM transmitter application (e.g., Case 3 vs. 4 or Case 8 vs. 9 in Table 2). The reason is that, for OFDM, FPA balances the computational load better than PPA.
(2) Bus Systems using a shared memory for program and local data (e.g., GGBA) require more memory arbitration time than in Bus Systems having separate memories for program and local data for each BAN (e.g., GBAVIII). This arbitration time difference explains why GBAVIII outperforms GGBA.
(3) SplitBA relieves bus traffic congestion due to shared memory requests from each BAN. With this reason, SplitBA beats GGBA in our example (Case 7 vs. 8).
(4) 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).

In the MPEG2 decoder results shown in Table 3, Hybrid (Case 13) shows the best performance because Hybrid allows the use of both BFBA and GBAVIII's features such as fast data transactions between adjacent BANs using Bi-FIFOs and global data accesses in global memory from all BANs. Above all, the reason Hybrid and GBAVIII outperform CCBA is faster arbitration time in read operations (3 cycles as compared to 5 in CCBA). In Table 3, BFBA and GBAVI perform poorly because data

has to be passed from BAN A to each BAN sequentially to supply the global data to be processed in each BAN. The end result is that Hybrid, generated by BusSyn, outperforms CCBA by 15.54% in this example.

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

Table 3. Evaluation Results in MPEG2 Decoder

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 4. Generation Time and Gate Count in the Generated Bus Systems

Table 4 shows the generation time of the Bus Systems generated using BusSyn and 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, all Bus Systems shown in Table 4 take 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 one-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, but 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 impacting factors such as bus types, processor types and software programming style resulting in a system having high performance. This goal is accomplished through BusSyn, which leads the user to easily design Bus Systems in a matter of seconds.

7. Conclusion

In this paper, we have described a methodology to generate custom Bus Systems for multiprocessor SoC designs. BusSyn, a bus generation tool using this methodology, generates five different Bus Systems as examples: BFBA, GBAVI, GBAVIII, Hybrid and SplitBA. In Section 6, the Bus Systems are evaluated in performance and are verified in operation with two applications: OFDM transmitter and MPEG2 decoder. Our methodology gives us a great benefit in fast design space exploration of bus architectures across the performance impacting factors such as bus types and software programming style. We showed that BusSyn achieves performance better than the hand design of a simple GGBA and CCBA, but in a matter of seconds instead of weeks.

8. Acknowledgements

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.

9. References

[1] IBM, “CoreConnect Bus Architecture,” [Online document], Available HTTP: http://www.chips.ibm.com/products/ coreconnect/ docs/cron_wp.pdf.

[2] ARM, “AMBA Specification Overview,” [Online document], Available HTTP: http://www.arm.com/ Pro+Peripherals/AMBA.

[3] Sonics, “Sonics μNetwork Technical Overview,” [Online document], Available HTTP: http//:www.sonicsinc.com/ Documents/Overview.pdf.

[4] Bill Dittenhofer, “Connecting Multi-Source IP to a Standard On Chip Architecture,” [Online document], Available HTTP: http://www.palmchip.com/pdf/CP-9248P.pdf.

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

[6] R.A. Bergamaschi and William R. Lee, “Designing Systems-on-chip using cores,” *Proceedings of Design Automation Conference*, pp. 420-425, 2000.

[7] D. Lyonnard, Sungjoo Yoo, Amer Baghdadi and Ahmed A. Jerraya, “Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip,” *Proceedings of Design Automation Conference*, pp. 518-523, 2001.

[8] G. Nicolescu, S. Yoo, A. Bouchhima and A. Jerraya, “Validation in a Component-Based Design Flow for Multicore SoCs,” *Proceedings of the International Symposium on System Synthesis*, pp. 162-167, 2002.

[9] F. Gharsalli, D. Lyonnard, S. Meftali, F. Rousseau, 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, 2002.

[10] Pentek, “Operating manual for Model 4290 and 4291,” [Online document], Available HTTP: http//:www.pentek. com/products/GetDoc.CFM/80042900.pdf.

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

[12] 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, 1999.

[13] Mento Graphics, “Seamless Hardware/Software Co-Verification,” [Online document], Available HTTP: http:// www.mentor.com/Seamless/datasheets/seamless_ds.pdf

[14] Synopsys, “VCS data sheet,” [Online document], Available HTTP:http://www.synopsys.com/products/simulation/ vcs_ds.html

[15] Kyeong Ryu and Vincent Mooney, “Automated Bus Generation for Multiprocessor SoC Design,” Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-64, 2002, Available HTTP: http://www.cc.gatech. edu/tech_reports.