AUTOMATED BUS GENERATION FOR MULTI-PROCESSOR SOC DESIGN

A Dissertation
Presented to
The Academic Faculty

by

Kyeong Keol Ryu

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
June 2004

AUTOMATED BUS GENERATION FOR MULTI-PROCESSOR SOC DESIGN

Approved by:

Dr. Vincent J. Mooney III, Adviser

Dr. Jeffrey A. Davis

Dr. Sudhakar Yalamanchili

Dr. Paul Benkeser

Dr. Thad Starner

Date Approved: June 11, 2004

Dedicated to my wife, Hyejung Hyeon, my parents,

and my parents-in-law

ACKNOWLEDGMENTS

This work could have not been finished without the support and sacrifice of many people I had to express my gratitude. First of all, I would like to deeply thank my adviser Vincent J. Mooney III. He has supported and encouraged me to develop my dissertation with his enthusiasm and professionalism throughout all stages of my Ph.D. program. He has been a great source of ideas and provided me with invaluable feedback. In addition, Dr. Mooney has been helping me improve my English skills with his consideration. I would also like to extend my appreciation to Dr. Jeffrey Davis, Dr. Sudhakar Yalamanchili, Dr. Paul Benkeser, and Dr. Thad Starner for serving on the committee and offering constructive comments.

I have to thank all Hardware/Software Codesign group members for their helps and friendship. It is obvious that, without many helps by them, my long journey at Georgia Tech would have been much harder and lonelier. Also, I wish to thank my friends, Dr. Chang-ho Lee, Dr. Jong-seung Moon, and Chang-hyuk Cho for their friendship.

Finally, I must thank my parents and my parents-in-law who have provided me with their enormous love, support and consideration for my life. I would also like to thank my brothers and my brothers-in-law for their consideration in all aspects. Most importantly, I cannot fail to thank my wire, Hyejung Hyeon. Without her unselfish devotion and her endless love, it would not have been possible to get through all the obstacles I have met during my study. I also thank my daughter Clair Seunghyun Ryu and my son Ryan Jihun Ryu who have given me great pleasure and their precious love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABI	Arbiter Bus Interface
AHB	Advanced High-performance Bus
AMBA	Advanced Micro-controller Bus Architecture
APB	Advanced Peripheral Bus
API	Application-specific Program Interface
ASB	Advanced System Bus
BAN	Bus Access Node
BB	Bus Bridge
BFBA	Bi-directional First-in-first-out Bus Architecture
BusSynth	Bus Synthesis tool
CAD	Computer-Aided Design
CBI	CPU Bus Interface
CCBA	CoreConnect Bus Architecture
DCR	Device Control Register
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FPA	Functional Parallel Algorithm
GBAVI	Global Bus Architecture Version I

GBAVII	Global Bus Architecture Version II
GBAVIII	Global Bus Architecture Version III
GBI	Generic Bus Interface
GGBA	General Global Bus Architecture
GUI	Graphic User Interface
HDL	Hardware Description Language
HybridBA	Hybrid Bus Architecture
IL	Interface Logic
IP	Intellectual Property
ISS	Instruction Set Simulator
JTAG	Joint Test Action Group
MBI	Memory Bus Interface
OCP	Open Core Protocol
OFDM	Orthogonal Frequency Division Multiplexing
OPB	On-chip Peripheral Bus
PE	Processing Element
PLB	Processor Local Bus
PPA	Pipelined Parallel Algorithm
RTL	Register Transfer Level
RTOS	Real-Time Operating System

SB	Segment of Bus
SoC	System-on-a-Chip
SplitBA	Split Bus Architecture
SRAM	Static Random Access Memory
TDMA	Time Division Multiplexed Access
UART	Universal Asynchronous Receiver-Transmitter
VCI	Virtual Component Interface
VLSI	Very Large Scale Integration

SUMMARY

The objective of this research is to provide a Computer Aided Design (CAD) tool with which the user can quickly explore System-on-a-Chip (SoC) bus design space in search of a high performance SoC bus system. From a straightforward description of the numbers and types of Processing Elements (PEs), non-PEs, memories and buses (including, for example, the address and data bus widths of the buses and memories), our Bus Synthesis tool, called BusSynth, generates a Register-Transfer Level (RTL) Verilog Hardware Description Language (HDL) description of the specified bus system. The user can utilize this RTL Verilog in bus-accurate simulations to more quickly arrive at an efficient bus architecture for a multi-processor SoC.

In the design of a multi-processor SoC, the bus architecture typically comes to the forefront because the system performance is not dependent only on the PE speed but also on the bus architecture in the system. An efficient bus architecture with effective arbitration for reducing contention on the bus plays an important role in maximizing performance. Therefore, among many issues of multi-processor SoC research, we focus on two issues related to bus architectures in this dissertation. One issue is how to quickly and easily design an efficient bus architecture for an SoC. The second issue is how to quickly explore the design space across performance influencing factors to achieve a high performance bus system.

To provide a solution to such system design issues, we propose a methodology to generate custom bus systems. The methodology was used for the implementation of BusSynth; thus, with BusSynth, designers can quickly and easily design a custom bus system to obtain high performance. During the design of a bus system,

BusSynth enables a user to customize many characteristics of each module (e.g., bus, PE and memory). Based on user options, BusSynth generates the required modules in the system using a module library, stitches the modules together to build a bus system and finally outputs synthesizable Verilog HDL code for the specified system. In this manner, BusSynth is capable of generating differently configured custom bus systems such as a Bi-directional First-In-First-Out (Bi-FIFO) Bus Architecture (BFBA), a Global Bus Architecture Version I (GBAVI), a Global Bus Architecture Version III (GBAVIII), a Hybrid Bus Architecture (HybridBA) and a Split Bus Architecture (SplitBA) as examples.

The methodology we propose gives designers a great benefit in fast design space exploration of bus systems across a variety of performance influencing factors such as bus types, PE types and software programming styles (e.g., pipelined parallel fashion or functional parallel fashion). We also show that BusSynth can efficiently generate bus systems in a matter of seconds as opposed to weeks of design effort to integrate together each system component by hand. Moreover, unlike the previous related work, BusSynth can support a wide variety of PEs, memory types and bus architectures (including a hybrid bus architecture) in search of a high performance SoC.

CHAPTER 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. The ability of the semiconductor industry to continually live up to Moore's prediction [26] makes it practical to put multiple Processing Elements (PEs) on a single chip. In particular, single-chip integration allows the designer to take advantage of increased bus speed and width. This is especially critical as the performance of a multi-processor SoC heavily depends on the efficiency of its bus architecture. This dissertation presents a methodology to generate a variety of custom bus systems using pre-designed reusable hardware modules for a multi-processor SoC.

1.1 Problem Statement and Motivation

In a few years, we will see an SoC with one billion transistors (memory chips with over a billion transistors already exist [58]), and we predict that the SoC will include many PEs. In the design of such a multi-processor SoC, the bus architecture of the SoC is an important performance factor due to multiple bus masters. Therefore, an efficient bus architecture with fast arbitration plays an important role in maximizing system performance. Moreover, when designing a multi-processor SoC including a bus architecture, users need to explore a diverse design space across performance-influencing factors in search of a high performance SoC. However, high performance system design via bus system design and design space exploration is very time-consuming since many performance-impacting factors are involved in the design: types of bus architectures, types of PEs, and types of memories. Thus, these issues motivate the development of

a design automation tool that is capable of generating customized SoC bus systems in a Hardware Description Language (HDL) and speeding up the SoC bus design space exploration for a high performance SoC.

1.2 Contributions

In this dissertation, we present a new methodology to generate custom bus systems. Unlike the previous research that will be described in Section 2, our methodology provides a more flexible bus system template to generate bus systems, and the template supports multiple and heterogeneous bus architectures and various optimized wrappers to attach Intellectual Property (IP) blocks to a bus so that the generated bus system is suitable for the desired applications. Please note that "IP block" and "IP core" are used interchangeably (i.e., with the same meaning) in this dissertation. The following items are the contributions of this research.

- **SoC Bus System Design Aid.** A bus mechanism in an SoC is a significant performance-impacting factor and gives many challenging points with regard to performance. Based on a generic bus system architecture that we proposed, a user can design an efficient and scalable bus system in an easy and fast manner for a multi-processor SoC system. We developed five different bus systems (GBAVI, GBAVIII, BFBA, HybridBA and SplitBA) as practical examples and verified the efficiency of our custom bus architecture. Therefore, our methodology as described in this dissertation can be an expert guide to design an SoC bus system.

- **Automated Bus Generation Tool.** How to easily and quickly design a multi-processor SoC bus system is an important issue in the increasing complexity of on-chip bus systems and in the context of ever shortening time-to-market demands. This dissertation presents an automated Bus Synthesis tool,

BusSynth, to meet this goal. Moreover, BusSynth generates a custom, application specific, configurable bus system for an SoC composed of multiple heterogeneous PEs, IP blocks, bus(es) and various types of memories. Based on the user specification, BusSynth can generate diverse custom bus systems (e.g., GBAVI, GBAVIII, BFBA and SplitBA), including a hybrid bus system (HybridBA), in synthesizable Verilog HDL. When compared to a typical global bus system (e.g., GGBA), the generated bus systems show superior performance (e.g., 41% reduction in execution time in the case of a database example, see Section 7.3).

- **Interconnect Delay Aware Bus Architecture Generation.** Due to the nature of SoC design, in which multiple IP blocks are placed together and connected with buses, interconnect delay plays a significant role in system performance as feature size is scaled down to the submicron level. In this dissertation, we describe a methodology to generate a custom bus architecture using accurate estimations of interconnect delay.

- **Case Studies.** This dissertation also delineates case studies of application examples of SoC designs in a component-based design approach that allows designers to explore efficient custom bus solutions with high performance. This research automatically integrates multiple and heterogeneous PEs, various types of buses (including a hybrid bus) and a variety of types of memories into an SoC. Custom bus systems in synthesizable Verilog HDL generated by BusSynth are evaluated in the context of three realistic applications: an Orthogonal Frequency Division Multiplexing (OFDM) transmitter, an MPEG2 decoder and a database example. We also use a Real-Time Operating System (RTOS) to run multitasking user applications on the integrated SoC.

- **Fast Design Space Exploration.** The methodology describing in this dissertation gives us a great benefit in fast SoC bus design space exploration across several important performance influencing factors (e.g., types of bus architectures, types of processing elements and types of memories) in search of a high performance SoC. Based on the user options, a bus system generated by BusSynth is designed in a matter of seconds instead of weeks for the hand design of a custom bus system.

CHAPTER II

RELATED WORK

We now present a review of previous work pertinent to this dissertation. First, we discuss several standard on-chip buses and standard bus interfaces, and then we depict several state-of-the-art commercial tools related to this research. Next, we show other related research in academia and industrial research labs.

2.1 SoC Bus Architectures

Most SoC designs are based on hardware blocks stitched together with bus signals, which are classified into groups of data, address, and control links. Several industries provide the following SoC bus architectures so that designers can easily integrate the IP blocks into a single silicon chip: CoreConnect, AMBA, CoreFrame, Wishbone, and SiliconBackplane μNetwork. Please note that all representations of commercial buses in this section are based upon publically available information at the time of publication of this thesis (June 2004).

2.1.1 CoreConnect

The IBM CoreConnect bus architecture [20] is an open standard and provides three levels of bus hierarchy: a Processor Local Bus (PLB), an On-chip Peripheral Bus (OPB), and a Device Control Register (DCR) bus. The PLB interconnects high-bandwidth devices such as PEs and memories since the PLB is a high performance and low latency processor bus with separate address bus, read data bus and write data bus for each bus master. The decoupled address and data buses support split-bus transaction capability for improved bandwidth. In contrast, the OPB provides separate low speed address and data buses for slow peripheral input/output (I/O)

devices such as serial ports, parallel ports, and Universal Asynchronous Receiver-Transmitters (UARTs). A bus bridge connects the PLB and the OPB together, and the bus bridge supports burst reads and writes as well as Direct Memory Access (DMA) transfers. The daisy-chained DCR bus offers a relatively low-speed data path for passing status and configuration information between CPU and IP blocks connected to a PLB. Since control registers in slave IP blocks that can be set from a master PE can be configured through the DCR bus, the use of the DCR bus lessens bus traffic on the PLB. To ease SoC design using CoreConnect, IBM provides design toolkits which support PLB and OPB functional models, bus monitors, and a Bus Functional Language (BFL) [20] for the control of the bus functional models.

2.1.2 AMBA

The Advanced Microcontroller Bus Architecture (AMBA) [1] from ARM provides an on-chip communication standard for designing a high-performance SoC. AMBA has three levels of bus hierarchy: Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advanced Peripheral Bus (APB). The AHB is a high-performance and high-speed bus connecting PEs, on-chip memories, and off-chip external memory interfaces. The ASB is a general-purpose system bus and is an older version which has been superseded by AHB. The ASB also interconnects PEs and system peripherals. The APB is a peripheral interconnection bus and is optimized for minimal power consumption. The APB can be connected to either AHB or ASB through a bus bridge. Thus, any latencies due to low performance peripherals connected to APB are buffered by the bridge to the high-performance buses, AHB and ASB.

AMBA and CoreConnect share many common features. However, unlike the CoreConnect bus, AMBA does not support features such as architecture extendability up to 256-bits and deep address pipelining. Furthermore, as of this writing (June

2004), AMBA only supports a single master of a peripheral bus while CoreConnect supports multiple masters [1] [20].

2.1.3 CoreFrame

CoreFrame [11] [34] from Palmchip is a bus architecture with two independent bus types: PalmBus and Mbus. PalmBus is designed for connecting low-speed peripherals and for accessing from CPU cores to peripheral blocks, while Mbus is designed for high-speed accesses to a shared memory block from CPU cores and peripherals. A processing node including a single CPU core, which is referred to as a *CPU subsystem* in CoreFrame, may contain local memories for its own use on its native CPU bus, links to PalmBus through a PalmBus interface, and links to MBus through a bus bridge. Since the *CPU subsystem* has dedicated local memories, the CPU can access its local memories without interaction with the rest of the system. This reduces bandwidth constraints on the shared memory. Furthermore, the use of a specialized "cache block" helps to minimize CPU accesses to the shared memory. To ease an SoC design, Palmchip provides an interface generation tool, CoreFrame Connect Kit, which assists users with the configuration of the interface modules for PalmBus and Mbus.

2.1.4 Wishbone

The Wishbone bus architecture [43] was developed by Silicore Corporation [42]. In August of 2002, Silicore placed the specification into the public domain via OpenCores [33], which is a organization that promotes the development of open IP cores. Thus, Wishbone is not copyrighted and may be freely copied and distributed as long as all modifications to Wishbone are also provided free for copy and distribution (please see the GNU General Public License (GPL) terms [16] which OpenCores uses [13]).

The Wishbone bus architecture is very simple since it defines only one bus. In a system that needs both a high-speed processor bus and a low-speed peripheral

bus, designers can use two Wishbones for buses, one operating at a high speed and one operating at a low speed. Thus, all cores are connected to the Wishbone buses by using the same bus interface. This way is simpler than using different buses for the high-speed bus versus the low-speed bus (e.g., PLB and OPB in the case of the CoreConnect). However, the Wishbone bus architecture supports various features in light of desired bus operations: multiple masters, single cycle read/write, block transfer cycles that systematically perform a set of single read cycles and/or a set of single write cycles, configurable address/data bus widths, and big versus little endian. Moreover, Wishbone supports various IP block interconnection methods: uni- and bi-directional buses, multiplexer based interconnects, tri-state based interconnections, off-chip I/O connections, and crossbar switches.

2.1.5 SiliconBackplane μNetwork

The SiliconBackplane μNetwork [45] from Sonics is an on-chip network that connects IP blocks in a system. The μNetwork isolates system IP from the network by requiring all system IP blocks to use a single bus interface protocol, the Open Core Protocol (OCP) (please see Section 2.2.1 for details). Thus, users can design and optimize the communication network knowing that all IP blocks which will utilize the communication network will do so using the same protocol (namely, OCP). Each IP block in a system communicates via a wrapper, which μNetwork calls an *agent*, using OCP, and the *agents* communicate with each other through μNetwork. Both OCP and the μNetwork protocol support modification of many system parameters in real time as system requirements change (e.g., arbitration scheme and address space), and the *agents* are generated by a tool called Fast Forward Development Environment provided by Sonics; therefore, designers can more easily implement an SoC that meets application requirements. The SiliconBackplane μNetwork offers fixed bandwidth by Time Division Multiplexed Access (TDMA)-based arbitration. This feature

is particularly suitable for real-time applications. In addition, since the μNetwork provides fixed latency, when a data transfer is not completed in time, it is retried later, and thus wait states are not inserted in the bus pipeline. This feature can help maintain predictable network bandwidth.

2.1.6 How We Differ

As compared to the buses presented in Section 2.1, our custom bus architectures (namely, GBAVI, GBAVIII, BFBA, HybridBA and SplitBA) generated based on our methodology using user options (please see the details in Sections 4 and 5) are more suitable for user specific applications. Therefore, we can obtain better performance when using one of our custom buses rather than aforementioned standard buses in an SoC. For example, in the context of a database example, SplitBA outperforms against a general global bus architecture by 41% reduction in execution time (see Section 7.3). The other performance evaluation results of the custom buses will also be shown in Section 7.3.

2.2 SoC Bus Interfaces

SoC design typically requires the mix and match of IP blocks on a single chip. Using a shared bus is one efficient way to connect the IP cores. However, because many types of buses are considered in high performance SoC designs, and because each bus type has different attributes, the introduction of a standard bus interface is useful so that each IP block can avoid having several interfaces to match to all available buses. Here, we describe two standard SoC bus interfaces as follows.

2.2.1 Open Core Protocol (OCP)

The Open Core Protocol (OCP) [44] developed by Sonics defines a bus interface for IP cores that connects the IP cores to on-chip buses. Communication requirements for an IP core can be described in this protocol format. The OCP interface is user-settable

so that designers can specify the interface's attributes (e.g., address and data bus widths). In OCP, there are four extensions beyond the Basic OCP version. The four extensions are Simple Extension, Complex Extension, Sideband Extension, and Debug and Test Interface Extension. Basic OCP includes only data flow signals, is based on a simple request and acknowledge protocol, and supports a unique data transfer on every clock cycle. However, the optional extensions support more functionality in control, verification, and test. In Simple Extension and Complex Extension, the protocols support burst transaction and pipelined writes; in addition, Sideband Extension also supports user-defined signals and synchronous resets. Moreover, Debug and Test Interface Extension supports Joint Test Action Group (JTAG) and clock control. Therefore, when OCP is integrated into an SoC, the protocol enables debugging and manufacturing test of IP blocks. OCP is available, potentially at no cost, under a license agreement that is agreed to over Sonics web site [32]. Products may use this standard without any royalty obligations.

2.2.2 Virtual Component Interface (VCI)

Like Sonics, the Virtual Socket Interface Alliance (VSIA) also supports the idea of a single bus interface for IP blocks and has a working group devoted to specifying such a protocol, the Virtual Component Interface (VCI) [8]. VCI defines a protocol for cycle-based and address mapped point-to-point communication. VCI is based on a handshake protocol in which each data transaction occurs on the rising edge of the clock when acknowledge and valid signals are high. Unlike OCP, VCI is a data-oriented protocol without the consideration of interrupt control and scan test signals. VCI is composed of three standards: Peripheral VCI (PVCI), Basic VCI (BVCI), and Advanced VCI (AVCI) protocol. PVCI is a subset of BVCI, which is also a subset of AVCI. PVCI and BVCI are for peripherals and for a simple processor system (e.g., a system supporting just a single read/write and DMA transfer), respectively.

In contrast, AVCI is for a more complex system (e.g., a system with a pipelined structure or specialized structure for graphics).

2.2.3 How We Differ

While OCP and VCI provide a generic interface between an IP block and an on-chip bus, in our approach we use a specialized wrapper for each specific IP block, providing a customized interface that is well matched to the IP block. For example, our methodology supports Memory Bus Interface (MBI) for memory, CPU Bus Interface (CBI) for a PE and Arbiter Bus Interface (ABI) for an arbiter. Use of these wrappers in a system provides more suitable interfaces due to their custom architectures and leads to a competitive system performance as will be shown in Section 7.3.

2.3 Commercial Tools Related to Bus Generation

We now describe several state-of-the-art commercial tools for automated bus generation for SoC designs. Since a bus provides a communication channel among IP blocks in an SoC, the tools typically support several bus architectures to integrate IP blocks. Please note that all representations of commercial tools in this section are based upon publically available information at the time of publication of this thesis (June 2004).

2.3.1 CoWare N2C

CoWare Napkin-to-Chip (N2C) [7] is a design environment for designing an SoC and a hardware platform at a system level. N2C provides a set of tools and methods for system-level design, hardware/software co-design, and IP block re-use. CoWare N2C uses C/C++/SystemC as a system-level description language and supports not only HDL design but also simulation capability. Automatic generation of glue logic and device drivers using an interface synthesis tool allows designers to integrate heterogeneous hardware/software functions into a system. In addition to such a hardware/software co-design capability, CoWare N2C provides a solution for the

decision of a bus architecture that is suitable for a user's application. A user may choose several options such as multi-layer, burst, or split transfer in two standard buses, AMBA or CoreConnect. With the CoWare N2C bus generator and simulator, the user can generate differently configured bus architectures and evaluate these generated buses with an application. Thus, users can explore a portion of the bus design space and choose a suitable bus architecture for their application.

2.3.2 Platform Express

Platform Express [25] from Mentor Graphics is a tool that uses IP blocks and on-chip buses, described in eXtensible Markup Language (XML), to automatically assemble heterogeneous components for an SoC design. Platform Express enables designers to quickly determine the suitability of platforms for system designs. To create a system, a designer just drags and drops library components (e.g., PEs, memories, and peripherals) in a graphical editor of a Graphic User Interface (GUI), and then the designer connects them to standard buses. After that, Platform Express automatically generates all the necessary connections among the components. Platform Express supports common on-chip standard buses such as AMBA from ARM and CoreConnect from IBM; these buses are used to link selected IP components. For design verification, Platform Express additionally invokes several verification tools (e.g., Seamless CVE, XRAY, and ModelSim from Mentor Graphics).

2.3.3 CoCentric System Studio

CoCentric System Studio from Synopsys provides a SystemC simulator and specification environment which enables users to verify and analyze hardware architectures and software algorithms at multiple levels of abstraction [47]. Since CoCentric System Studio supplies a unified design environment based on SystemC, users can seamlessly design a system from abstract algorithms to synthesizable SystemC. In a system architecture design, CoCentric System Studio works together with Synopsys DesignWare

SystemC AMBA IP blocks. Therefore, users can quickly integrate system IP blocks with AMBA, which provides a shared communication channel. Up to recently, modeling an architecture required Register Transfer Level (RTL) hardware description that requires great effort and tedious work to design and verify the model. CoCentric System Studio, on the other hand, supports Transaction-Level Modeling (TLM) where a communication channel is modeled based on its behavior and is expressed in terms of transactions [46]. Therefore, in the design and verification phase, simulation speed can be much faster than the simulation of a traditional RTL-based model at the cost of modeling-accuracy loss. For example, in communication modeling, while an RTL model is fully pin-accurate, data accurate and cycle accurate, TLM waives such low-level details, instead controlling inter-module communication by use of an Interface Method Call (IMC) between modules [56]. A system verified at the transaction-level can be synthesized to logic gates by SystemC Compiler and Design Compiler from Synopsys.

2.3.4 Magillem

Magillem from Prosilog is a tool for importing IP blocks and graphically creating SoC architectures [36]. For the generation of an SoC, Magillem supports two standard on-chip buses (AMBA and CoreConnect) and standard bus interfaces (OCP and VCI). After a user loads required IP blocks in a graphic editor and connects them together graphically, the tool automatically generates transaction level (e.g., SystemC) or RTL (e.g., Verilog) code, enabling designers to explore the system architecture. To customize the required IP blocks, the user can specify each IP block's options (e.g., data width and arbitration scheme) through a GUI. Furthermore, Prosilog provides IP Creator, as a part of Magillem, for integration and re-use of non-VCI or non-OCP compatible IP blocks by wrapping them in a structure compatible with OCP or VCI.

Thus, designers can assemble the IP blocks in an SoC that uses either the VCI or the OCP interface.

2.3.5 How We Differ

Unlike the commercial tools discussed above in Section 2.3, our bus synthesis tool, BusSynth (please see the details in Section 5) can generate SoC bus systems with standard bus architectures (such as CoWare N2C and Platform Express) as well as custom bus architectures; furthermore, based on user options, BusSynth generates a single bus architecture as well as multiple and hybrid bus architectures. For example, BusSynth generates GBAVI, GBAVIII, BFBA, HybridBA and SplitBA, which will be described in Sections 4.1 and 5.1. Moreover, BusSynth enables interconnect delay aware bus architecture generation that will be explained in Section 6.3. (However, please note that while we did not actually implement a large array of standard bus structures in BusSynth – please see Sections 4 and 5 for details about what we did implement – nevertheless any of the standard bus structures discussed so far can be integrated into BusSynth in a straightforward fashion.)

2.4 Additional Prior Work Related to Bus Generation

As additional research related to bus generation for SoC design, many papers present communication topology generation, IP block assembly for an SoC, and component-based SoC design as follows.

Gasteier *et al.* [12] describe 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 a single type of bus topology (a single global bus topology). Our method, on the other hand, supports multiple bus types and bus topologies.

Bergamaschi *et al.* [3] present design automation of an SoC using IP blocks connected via CoreConnect. In their methodology for assembling IP cores, their algorithm checks the compatibility of IP I/O ports and generates wires to connect the IP blocks. Again, we, on the other hand, support a wider variety of bus types and architectures than they present.

Pai Chou *et al.* [6] show an IP based approach to SoC building. An input description to their algorithm designates a bus topology that specifies how IP blocks 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 IP blocks communicate with each other by using a particular network protocol (e.g., I^2C or CAN) chosen. Our methodology, 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, our methodology targets SoC designs where direct, non-packet based connections are desired. For this reason, our methodology 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 [4] [5] [14] [15] [23] [30] [57] from TIMA laboratory present 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.* [23] introduce a design flow for the generation of an application-specific multi-processor architecture. They used a generic multi-processor architecture template to support two types of buses (e.g., a point-to-point connection and a shared

bus) and a communication coprocessor for the interface between a PE and a bus. To interface each heterogeneous component to other parts of the target system, they depict a generic wrapper architecture that adapts to different communication protocols based on automatic wrapper generation [14] [15] [57]. Cesário *et al.* [4] [5] and Nicolescu *et al.* [30] described a component-based design environment to enable an automatic wrapper generation tool to support various hardware interfaces, device drivers and Application Program Interfaces (APIs).

Shin *et al.* [41] show how an efficient configuration of a parameterized on-chip system bus could be found using a software tool they developed. They, however, do not discuss the generation of various bus communication topologies based on user requests; nor do they discuss any associated bus architecture generation.

Thepayasuwan *et al.* [52] describe layout conscious bus architecture synthesis. They use interconnect delay from a system layout to generate a single bus architecture operating at the maximum achievable bus speed considering worst-case interconnect delay. We, on the other hand, consider ways to alter the bus architecture – e.g., by splitting the bus into several separate buses connected by bus bridges – and, together with worst-case interconnect delay information, generate both a custom bus architecture as well as custom bus control logic able to handle, for example, multiple delays to different processors closer or further away in terms on interconnect delay. The details of how we do this are described in Section 6.3. In short, our methodology provides a more customized bus architecture that is suitable for a specific user's set of applications since the methodology generates the bus architecture based on various user-input options.

2.5 Summary

In this chapter, we first looked into several popular industry bus architectures (namely, CoreConnect, AMBA, CoreFrame, Wishbone, and SiliconBackplane μNetwork). Then,

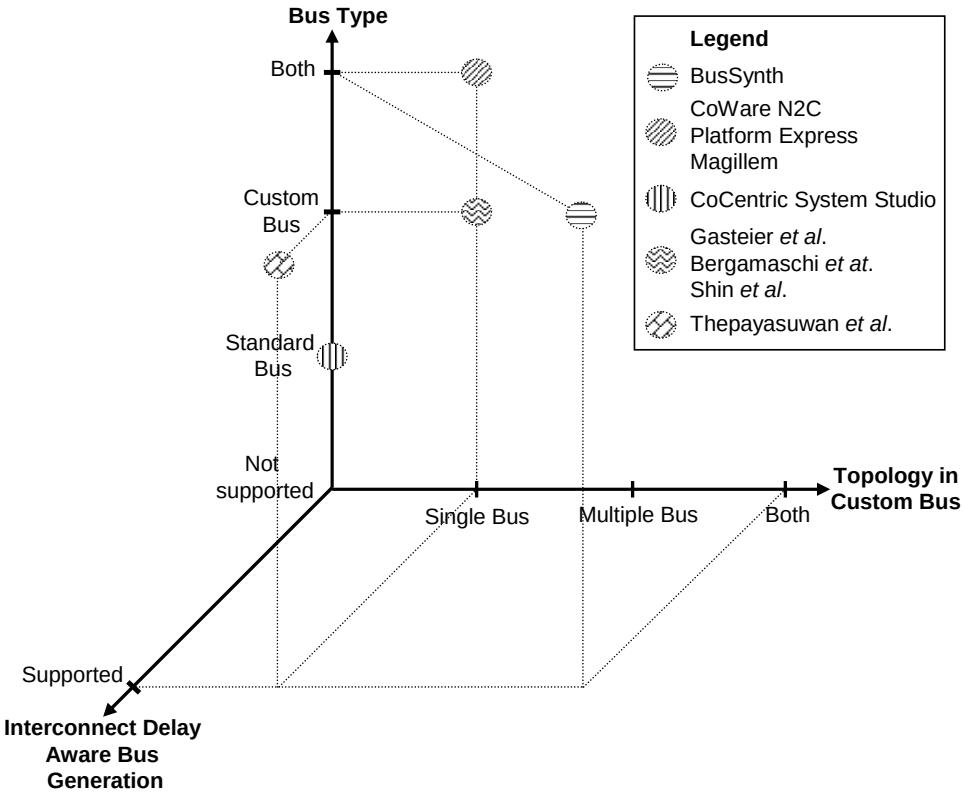


Figure 1: A Comparison of Bus Generation Tools

we discussed two bus interface protocols (namely, OCP and VCI) that allow custom IP blocks to match to the selected buses by providing a standardized protocol. Next, we described four commercial tools (namely, CoWare N2C, Platform Express, CoCentric System Studio, and Magillem) that enable bus architecture generation for an SoC design. Finally, we introduced additional prior work related to bus generation in academia and industrial research labs. In each of the previous sections (2.1, 2.2, 2.3 and 2.4), we described key differences with the research of this thesis. Figure 1 shows a summary of the key differences. As shown in Figure 1, BusSynth generates custom bus systems with a single bus topology as well as multiple bus topologies; furthermore, BusSynth supports interconnect delay aware bus generation. In the next section, we will describe our methodology to generate a customized bus architecture for a multi-processor SoC.

CHAPTER III

BUS SYSTEM STRUCTURE

In this chapter we will begin by defining some of terms that will be used throughout this dissertation. Then, we will describe our bus system structure to be used in our bus system generation.

3.1 Terminology for Bus System Generation

Before proceeding to discuss our Bus Synthesis tool (BusSynth), we first explain some of the terms we will be using to describe the different components of a bus architecture. Example 3.1 explains some of the terminology we have defined.

(a) Processing Element (PE): a hardware unit that performs algorithmic processing – usually a CPU, but it 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 (i.e., different bus clock frequencies) in buses connected by the BB (see Section 5.1.1 for details).

(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, BFBA).

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

(g) Local Bus (LB): a contiguous bus (no BBs) internal to a particular BAN that connects bus interface hardware unit(s) (in our case, CBI and/or MBI) attached to a PE, a memory or a hardware unit (in our case, non-CPU block), where the bus is composed of address, data and control wires.

(h) Module: a hardware unit such as PE, BB, SB, an arbiter, SRAM or interface logic blocks, where the specific interface logic blocks will be explained in more detail in Section 3.2. Note that it is possible to extend the definition of module to include newly designed hardware units that carry out specific functions. For this dissertation, however, the definition given for module suffices.

(i) Bus Subsystem: one or more BANs connected together using the same bus or a combination of different bus architectures without a bus bridge (in our case, either GBA, BFBA or the combination of GBA and BFBA).

(j) Bus System: a system that consists of one or more Bus Subsystems connected together by using one or more bus bridges.

Example 3.1 Terminology

Figure 2 shows an SoC consisting of four PEs, Motorola PowerPC (MPC) 755s [28], each with an L1 cache. Each MPC755 is an example of a PE. In the bottom right of Figure 2, the SB of BAN J is a segment of bus for Bi-FIFO Bus Architecture (BFBA). The SB is

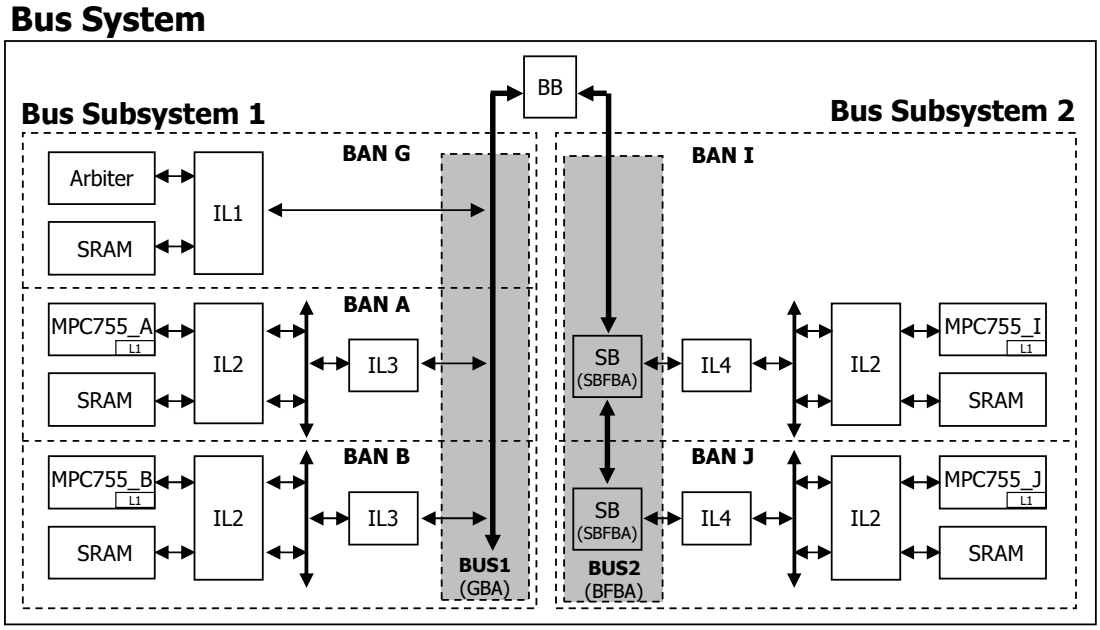


Figure 2: Example of a Bus System

used to connect BAN J to BAN I. The SB of BAN I is also another segment of bus for Bi-FIFO Bus Architecture (BFBA). Both SBs make a bus type, BFBA, by being connected each other. Note the use of Interface Logic blocks (IL2 and IL4) to connect MPC755_J to the SB. The bottom right of Figure 2 also shows MPC755_J connected to local SRAM and an SB to form Bus Access Node J (BAN J). In BANs A and B of Figure 2, LBs connecting ILs are shown between IL2 and IL3 of each BAN; similarly, in BANs I and J, LBs are shown between IL2 and IL4 of each of BAN. In BAN J, each block such as SRAM, IL2, IL4 or SB is a module. BAN J is adjacent to BAN I, and the BANs I and J together form a Bus Subsystem using bus type BFBA for communication. On the left-hand side of Figure 2, BANs A, B and G form another Bus Subsystem in which GBA is used for communication. A BB connects the two Bus Subsystems as shown in the top middle of Figure 2. On the whole, Figure 2 shows an example of a Bus System composed of two Bus Subsystems. □

3.2 Bus System Structure

In this section, we show a hierarchical structure in each Bus System to be generated. Figure 2 shows an example of a hierarchically structured multi-processor Bus System that has two Bus Subsystems with 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, hardware modules and/or memories together with associated control logic. Here, 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 a multi-processor SoC Bus System design.

Figure 3 depicts a more detailed version of the Bus Subsystem shown on the left-hand side of Figure 2. In addition to PEs (e.g., MPC755) and memories (e.g., SRAM) in the BANs of Figure 3, additional modules are specified as Interface Logic

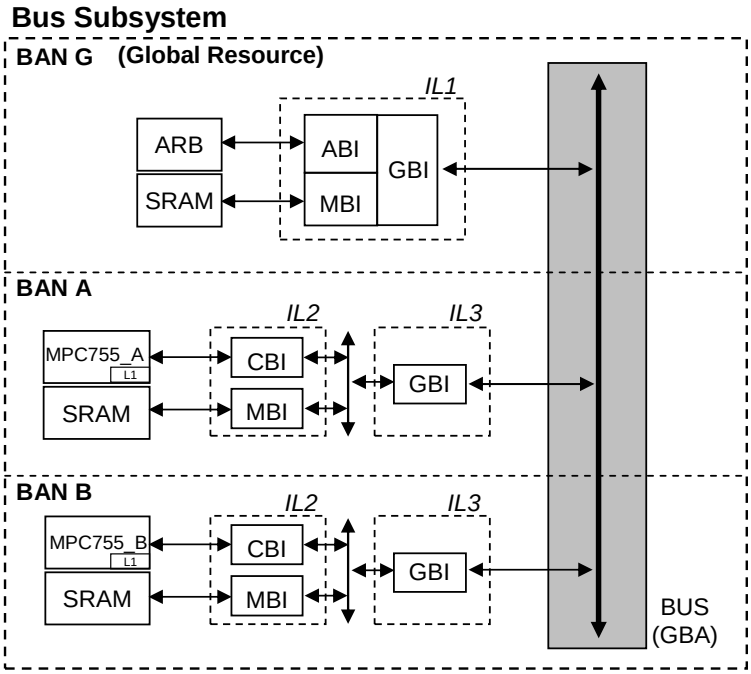


Figure 3: Example of a Bus Subsystem

(IL): CPU (or PE) to Bus Interface (CBI), Memory to Bus Interface (MBI), Generic Bus Interface (GBI) and Arbiter to Bus Interface (ABI).

With these ILs, each BAN can have different types of PEs, hardware modules and/or memories because the ILs enable the heterogeneous modules to adapt to one another. 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., GBAVIII 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 multi-processor Bus System can be implemented in an easy manner.

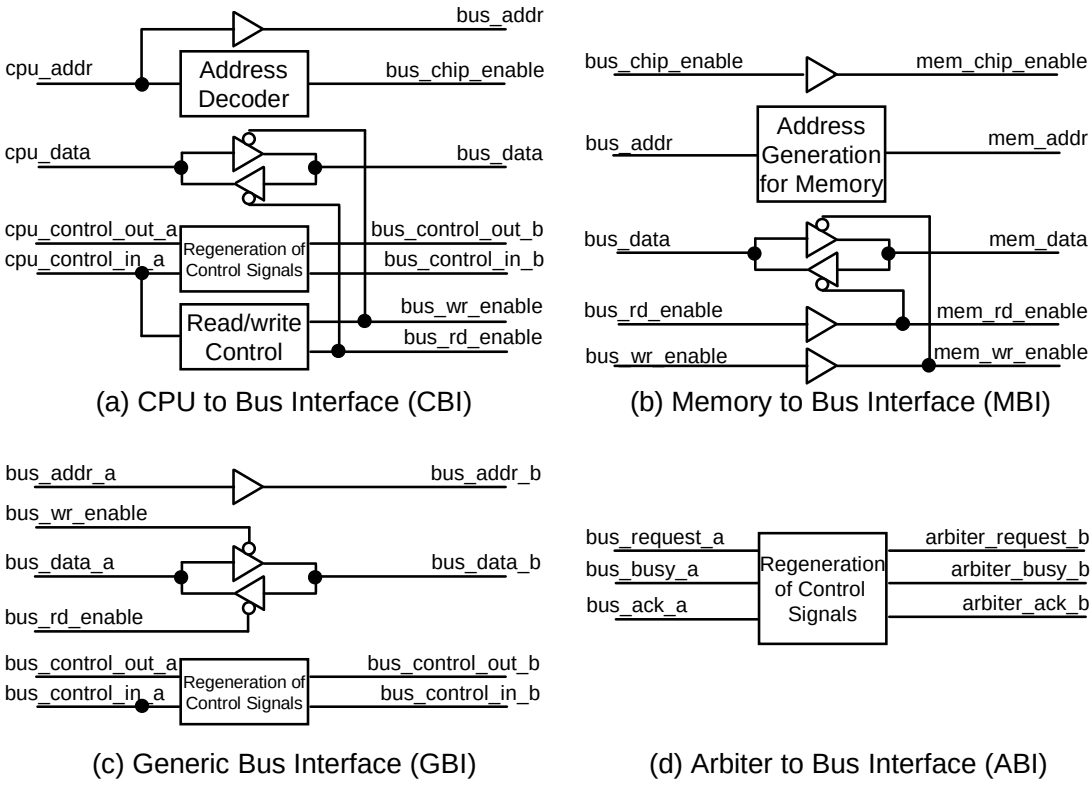


Figure 4: Block Diagrams of Interface Logic Blocks

Figure 4(a) shows the CBI block that adapts between a CPU (or PE) and a bus and that is composed of an address decoder, bi-directional data buffer, a block for regeneration of control signals (e.g., AACK_BAR and TA_BAR in PowerPC) and a block for generation of read/write enable signals. In Figure 4(a), for the CBI for MPC755, gate counts to implement each block are as follows: 128 NAND2 gate equivalents for the address decoder, 133 NAND2 gate equivalents for the bi-directional data buffer, 1027 NAND2 gate equivalents for the block for regeneration of control signals and 29 NAND2 gate equivalents for the block for generation of read/write enable signals (please note that we use TSMC 0.25μm technology to estimate the gate counts by using Synopsys Design Compiler [49]). The MBI block shown in Figure 4(b) adapts between a memory and a bus and is composed of a memory address generator, a bi-directional data buffer and buffers for read/write enable signals. For the MBI for SRAM shown in Figure 4(b), gate counts to implement each block are as follows: 21 NAND2 gate equivalents for the memory address generator and 133 NAND2 gate equivalents for the bi-directional data buffer. The GBI block shown in Figure 4(c) is composed of address bus buffer, a bi-direction data buffer and a block for regeneration of control signals (e.g., bus request and bus busy signals). Gate counts to implement the GBI for a global bus shown in Figure 4(c) are 133 NAND2 gate equivalents for the bi-direction data buffer and 46 NAND2 gate equivalents for the block for regeneration of control signals. Finally, the ABI block interface between an arbiter and a bus is shown in Figure 4(d), where the block regenerates arbiter control signals (e.g., arbiter request and arbiter acknowledge signals). In Figure 4(d), the ABI for a First-Come-First-Serve (FCFS) arbiter is implemented with 10 NAND2 gate equivalents.

Our current tool supports only two types of GBIs (namely, GBI_GBA and GBI_BFBA); however, more GBI types could easily be added to our tool after being defined. In the tool, the GBI_GBA can be used in a Bus System that use a Global Bus Architecture (GBA) to connect BANs in each Bus Subsystem while the GBI_BFBA can be used in

a Bus System with Bi-FIFO blocks to connect BANs in each Bus Subsystem. Thus, they have different functions to adapt to either a GBA or a Bi-FIFO bus architecture. Our tool currently supports two SB types: SB_GBA and SB_BFBA. However, more SB types could be added to the tool after being defined. SB_GBA is used to form a GBA that connects BANs in each Bus Subsystem while SB_BFBA is for a Bi-FIFO architecture connecting BANs in each Bus Subsystem.

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 3. On the other hand, the Bus System structure shown Figure 2 and the Bus Subsystem structure shown in Figure 3 allows a user to adapt a standard commercial bus architecture (e.g., AMBA). As shown in Figure 3, ILs adapt hardware units (e.g., arbiter, SRAM and MPC755) to specific buses (e.g., GBA). Thus, if our Module Library that will be described in Section 6.1 provides the wrappers (i.e., ILs) for the various possible buses (e.g., a global bus or an AMBA bus), our approach enables the user to choose a custom bus topology as a Bus System. In support of the choices of a user, BusSynth will generate custom Verilog HDL at the Register-Transfer Level (RTL) as will be described in Chapter 5.

3.3 Summary

In this chapter, we have defined some of the terms that we will use to describe Bus Systems throughout this dissertation. We have explained a basic Bus System structure used to generate Bus Systems with our tool BusSynth. In the next chapter (Chapter 4), based on a user-specified bus structure, we will show how to specify a wide variety of Bus Subsystems and our method to communicate among PEs in the specified Bus Subsystems. In the chapter after next (i.e., in Chapter 5), we will show how to specify Bus Systems (as opposed to just Bus Subsystems) using our methodology.

CHAPTER IV

BUS SUBSYSTEM SPECIFICATION

In this chapter, we show Bus Subsystem specification based on user input, where the specified Bus Subsystems can have various bus architectures. We target the ability to generate a wide variety of Bus Subsystems by using our bus synthesis tool, BusSynth. Then, we show how to communicate among PEs in the generated Bus Subsystems.

4.1 How to Specify Bus Subsystems

Before we describe in a later chapter (Chapter 6) our detailed methodology for Bus Subsystem generation, this chapter shows how to specify various Bus Subsystems, based on user options that are user inputs to BusSynth.

Several categories in user options are as follows:

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

(2) Bus Subsystem Property: number of BANs, address bus width and data bus width. Note that any Local Bus on a particular BAN is assumed to have the same address and data widths as the non-Local Bus(es) in the Bus Subsystem. Obviously, all buses in a Bus Subsystem have the same address and data widths.

(3) BAN Property: CPU type, Non-CPU type, number of global memories and number of local memories for each BAN, where the CPU type is one of NONE, MPC750, MPC755, MPC7410 or ARM9TDMI, and the Non-CPU type is one of NONE, DCT or MPEG2 decoder. Note that this can be easily extended to include new CPUs or additional predesigned reusable components (Non-CPUs).

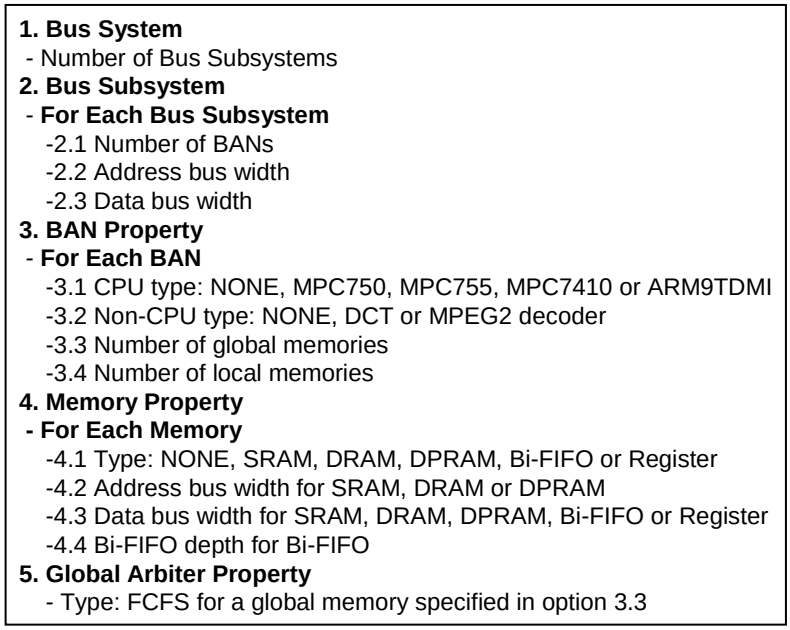


Figure 5: User Options to Configure a Custom Bus Subsystem

(4) Memory Property: memory type, address bus width, data bus width and Bi-FIFO depth, where the memory type is one of NONE, SRAM, DRAM, DPRAM, Bi-FIFO or Register, the address bus width is an option for SRAM, DRAM or DPRAM, and the Bi-FIFO depth is an option for Bi-FIFO. Note that this can easily be extended to include additional memory types.

(5) Global Arbiter Property: arbiter type. Currently, the type is only First-Come-First-Serve (FCFS); however, the type can easily be extended to include additional arbiter type (e.g., priority based arbiter). This property is selected only when a global memory is specified in the BAN property.

Figure 5 shows a summary of the user options (1) through (5) described above. The input sequence of the user options is as follows. First, the user enters the number of Bus Subsystems for a Bus System. Next, the user specifies the number of BANs, address bus width and data bus width for each Bus Subsystem (please note that a BAN has only one non-Local Bus that connects the BAN to the rest of the Bus

Subsystem). Then, for each BAN specified in Bus Subsystem Property 2.1, the user inputs CPU type, Non-CPU type, number of global memories and number of local memories in the BAN Property option if the user wants to have these resources in a BAN. Finally, in the BAN Property option, the user inputs Memory Property (namely, memory type, address bus width, data bus width and Bi-FIFO depth) for each selected memory if any memory is required in a BAN. How to use these options to generate various Bus Subsystems is shown in Examples 4.1, 4.2 and 4.3.

All Bus Subsystem examples shown in Figures 6, 7 and 8 have four PEs and a total of 256KB (64KB per PE) of L1 cache memory. Non-L1 cache memory size is 40MB for the examples shown in Figure 7 and 8 and 32MB for the example shown in Figure 6. The reason that Figure 6 has only 32MB is that Figure 6 does not have any global memory; in a Bi-FIFO based system, we found that a global memory tends to not increase performance at all in the applications we considered (see Chapter 7). Please note that, as defined in Chapter 3, a single Bus Subsystem can also be a Bus System, if the Bus Subsystem is not connected via bus bridge(s) to any other Bus Subsystem(s). While BusSynth can generate a Bus Subsystem having any number of PEs according to the user options, the examples presented in detail in this section all have the same number of PEs in order to provide a basis for fair comparisons later in Section 7.3 (please note that the examples shown in Figure 7 and Figure 8 have 40MB total of non-L1 cache memory; nevertheless, the bus examples of Figures 7 and 8 do not result in the best performance as shown in Table 10 of Chapter 7 Experiments and Results). In all examples in this thesis, we use the Motorola PowerPC (MPC755) for the PE core, which, however, can be changed to any other core simply by adding a CBI module for the new PE core (e.g., ARM9TDMI) to be operated in the Bus System.

Example 4.1 User Options to Generate BFBA

A user input sequence which specifies a Bus Subsystem we call the Bi-directional First-in-first-out Bus Architecture (BFBA) 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 5). Then, the user inputs "4" for the number of BANs (user option 2.1), "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3) in the Bus Subsystem property. Please note that the address and data bus widths in Figure 6 are all the exact same for all Local Buses (e.g., CPU Bus A) as well as buses to Bi-FIFOs; we could, if desired, easily update our current tool to specify particular widths for each Local Bus in each BAN separately. Next, the user inputs the fields of BAN Property for each BAN : "MPC755" for

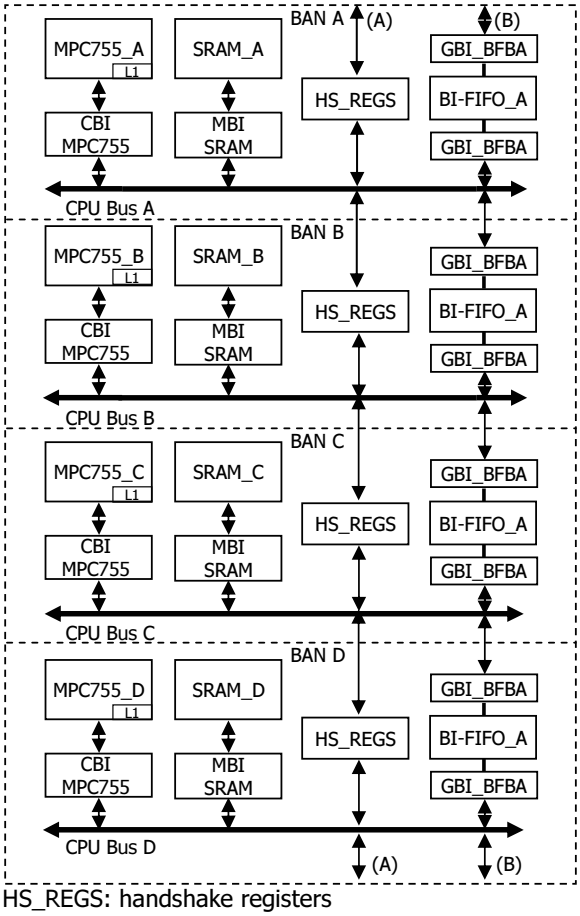


Figure 6: Diagram of BFBA

the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3) and "3" for the number of local memories (user option 3.4). Finally, the user inputs the Memory Property for three local memories chosen for each BAN. For the first local memory, the options are as follows: "SRAM" for the memory type (user option 4.1), "20" for the address bus width (user option 4.2) and "64" for the data bus width (user option 4.3) for 8MB SRAM in each BAN. For the second local memory, the user inputs "Bi-FIFO" for the memory type (user option 4.1), "64" for the data bus width (user option 4.3), "1024" for the Bi-FIFO depth (user option 4.4). After that, the user enters the third local memory property: "Register" for the memory type (user option 4.1) and "32" for the data bus width (user option 4.3). With these options, the generated Bus Subsystem BFBA is shown in Figure 6 where there are four equivalent BANs, each of which has an MPC755, an 8MB SRAM, a single 32-bit register, a 1024-entry Bi-FIFO and GBI_BFBAs as a Generic Bus Interfaces (GBI) for the Bi-FIFO block. In this example, since a Bi-FIFO is specified in each BAN, our tool automatically extracts GBI_BFBA from a library (namely, Module Library that will be described in Chapter 6 in detail) for the Bi-FIFO block connecting BANs. Here, we assume that any Bi-FIFOs specified by the user are intended to provide a path between BANs. □

As shown in Figure 6, Bi-directional First-in-first-out Bus Architecture (BFBA) has a Bi-FIFO between adjacent BANs. This design is similar to some commercially available multi-processor Printed Circuit Boards (PCBs) such as the Quad TMS320C6701 Processor VME Board from Pentek [35]. 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 6 will be presented in Section 4.2. Note that BFBA works well in a pipelined style of operation.

Example 4.2 User Options to Generate GBAVIII

User inputs to generate a Bus Subsystem we call Global Bus Architecture Version III

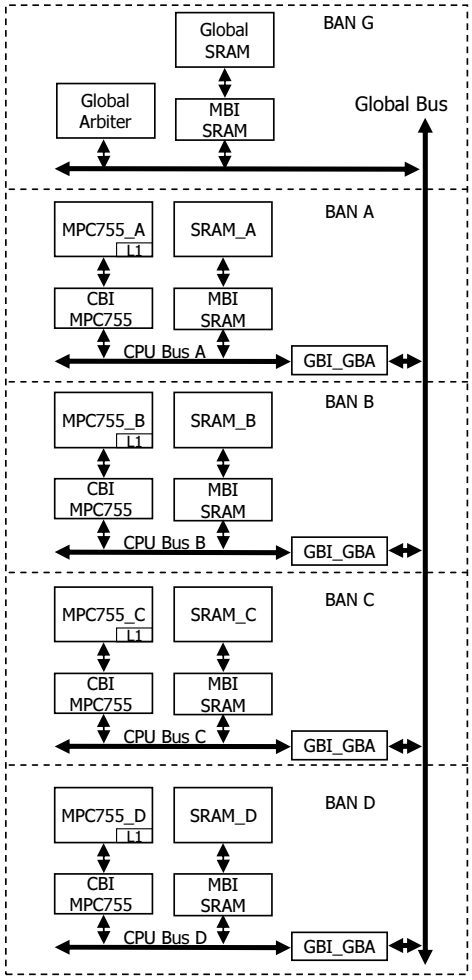


Figure 7: Diagram of GBAVIII

(GBAVIII) are as follows. The user first specifies the number of Bus Subsystems by entering a "1" in Bus System Property (user option 1 in Figure 5). Then, the user inputs "5" for the number of BANs (user option 2.1), "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3) in the Bus Subsystem property. Next, the user inputs the fields of BAN Property for four BANs (the bottom four BANs in Figure 7): "MPC755" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3) and "1" for the number of local memories (user option 3.4). Next, the Memory Property is input for a memory in each of the four BANs (the bottom for BANs in Figure 7): "SRAM" for the memory type (user option 4.1), "20" for the address bus width (user option 4.2) and "64" for the data bus width

(user option 4.3) for resulting in an 8MB memory size in each BAN. With these options, the bottom four BANs shown in Figure 7 are generated: each BAN has an MPC755 and an 8MB SRAM each with associated interface logic blocks CBI_MPC755, MBI_SRAM and GBI_GBA for the interface to the global bus. Please note that in our current tool, specified bus address and data widths (in user options 2.2 and 2.3) for a Bus Subsystem (e.g., GBAVIII, HybridBA or each Bus Subsystem in SplitBA shown in Figures 7, 8 and 18) are assumed to apply to all Local Buses as well as non-Local Buses. In this specific example, since no Bi-FIFOs are specified, the tool automatically extracts GBI_GBA from a library (namely, Module Library that will be described in Chapter 6 in detail) for the bus between BANs; however, as we explained in Example 4.1, the tool extracts GBI_BFBA from the library as a GBI for BFBA when a Bi-FIFO is specified; we assume that any Bi-FIFOs specified by the user are intended to provide a path between BANs.

Then, continuing with the generation of Figure 7, the user inputs the fields of BAN Property for one additional BAN (the top BAN in Figure 7) as a global memory block: "NONE" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "1" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4). Finally, the Memory Property is input for the global memory in the remaining BAN: "SRAM" for the memory type (user option 4.1), "20" for the address bus width (user option 4.2) and "64" for the data bus width (user option 4.3) for resulting in an 8MB memory size in the BAN. With these options, the top BAN in Figure 7, BAN G, is generated: an 8MB SRAM with a Global Arbiter. Please note that any BAN specified to contain *only* memory automatically infers a First-Come First-Serve (FCFS) arbiter for the number of masters (in Figure 7 there are four masters) specified in the Bus Subsystem. □

GBAVIII shown in Figure 7 is a Global Bus Architecture (GBA) with a global arbiter and a global memory (please note that in BusSynth, the FCFS Global Arbiter is generated when a user wants to have a global memory). When any BAN

tries to access the global memory through the global bus in Figure 7, the global arbiter resolves the case of multiple memory requests from the BANs. Currently, the only choice is an arbiter using a FIFO to implement a First-Come-First-Serve (FCFS) scheduling scheme; however, an arbiter having a different policy such as a priority-based protocol could easily be added to BusSynth (and, correspondingly, to the user options of Figure 5, e.g., as option 6). The Global SRAM in Figure 7 can also be replaced with another memory type (see option 4.1 in Figure 5) by using its corresponding MBI, which 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 7 will be shown in Section 4.2.

Please note that Global Bus Architecture Version I (GBAVI) is a Bus System with bus bridges and so will be presented in the next chapter. Also, please note that Global Bus Architecture Version II (GBAVII) was presented in [38] but was not chosen for automated generation in this dissertation because its bus architecture is almost the same (in both structure and achievable performance) as GBAVIII; in addition, GBAVII shows only a tiny (<1%) performance improvement over GBAVI to be presented in Section 5.1.1. However, if desired, the GBAVII bus could easily be added to our tool.

According to the user options shown in Figure 5, the user can customize any Bus Subsystem in our bus synthesis tool BusSynth. As one of the customized Bus Subsystems, the user might want to generate a bus mixing together both Bi-FIFO-based and GBA-based communication. Example 4.3 describes how to generate such a customized Bus Subsystem by the user options.

Example 4.3 User Options to Generate HybridBA

Suppose a user wants to generate a specialized bus using several of the custom buses explained earlier: specifically, a bus combining both the Bi-FIFO blocks from BFBA and the

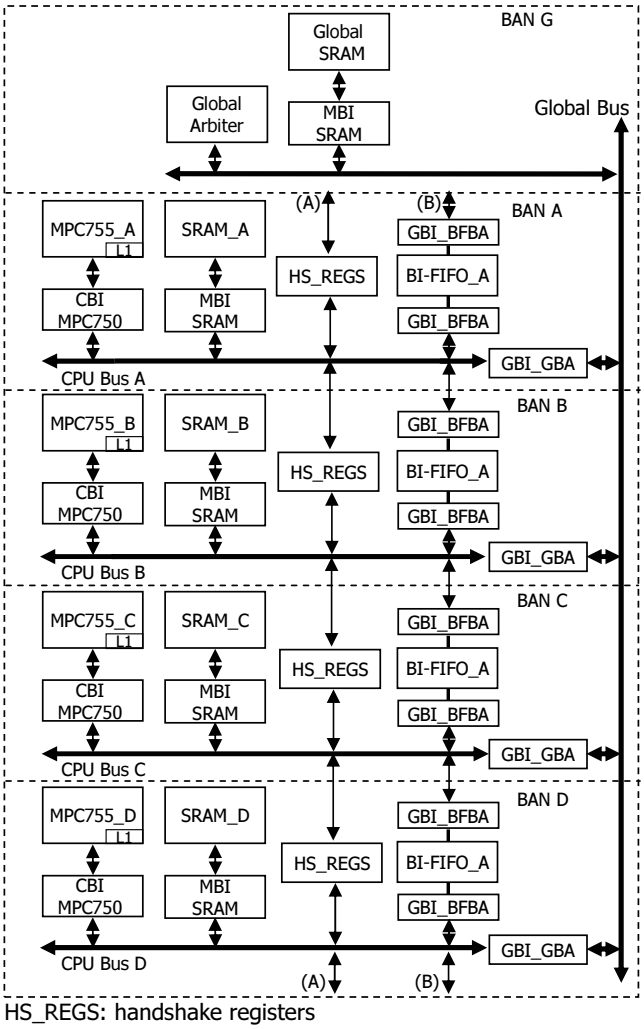


Figure 8: Diagram of HybridBA

global bus from GBAVIII. We call the Bus Subsystem having this combined bus type as Hybrid Bus Architecture (HybridBA). To generate such a combined bus, the user needs to input the user options shown in Figure 5 as follows.

First, the user enters "1" for the number of Bus Subsystems (user option 1 in Figure 5). Then, in the Bus Subsystem property, the user inputs "5" for the number of BANs (user option 2.1), "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3). Next, the user inputs the fields of BAN Property for four BANs (the bottom four BANs in Figure 8): "MPC755" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3)

and “3” for the number of local memories (user option 3.4). Next, the user inputs the Memory Property for the three local memories chosen to be in each of four BANs (BANs A through D). For the first local memory, the options are as follows: “SRAM” for the memory type (user option 4.1), “20” for the address bus width (user option 4.2) and “64” for the data bus width (user option 4.3) resulting in an 8MB SRAM. For the second local memory, the user inputs “Bi-FIFO” for the memory type (user option 4.1), “64” for the data bus width (user option 4.3), “1024” for the Bi-FIFO depth (user option 4.4). After that, the user enters the Memory Property for the third local memory: “Register” for the memory type (user option 4.1) and “32” for the data bus width (user option 4.3). With these options, each of the bottom four BANs in Figure 8 has MPC755 and three local memories, namely, 8MB SRAM, 1024-depth Bi-FIFO block and 32-bit handshake registers HS_REGS.

Then, the user inputs the fields of BAN Property for a remaining BAN (the top BAN in Figure 8) as a global memory block: “NONE” for the CPU Type (user option 3.1), “NONE” for the Non-CPU Type (user option 3.2), “1” for the number of global memories (user option 3.3) and “0” for the number of local memories (user option 3.4). Finally, the user inputs the Memory Property for the global memory in the remaining BAN as follows: “SRAM” for the memory type (user option 4.1), “20” for the address bus width (user option 4.2) and “64” for the data bus width (user option 4.3) resulting in an 8MB memory size in BAN G.

With these options, the generated Bus Subsystem HybridBA is shown in Figure 8, where each BAN has an 8MB SRAM, and a global arbiter in BAN G arbitrates global memory requests from the bottom four BANs and is generated since the user wants to have a global memory in BAN G. Handshake register blocks HS_REGSs in the bottom four BANs are for communication among MPC755s, and Bi-FIFOs in the bottom four BANs provide a fast data path between neighboring BANs. □

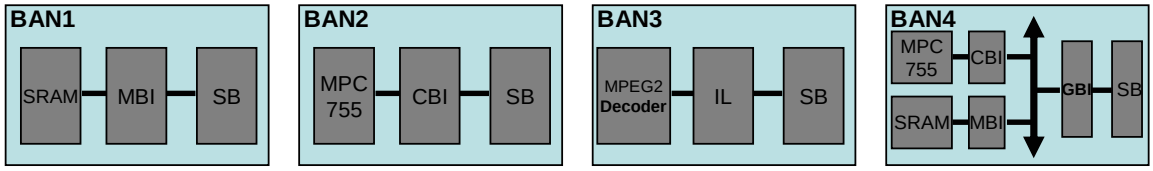
The Bus Subsystem we call HybridBA shown in Figure 8 is a combination of BFBA and GBAVIII. This combination allows the bus architecture to exploit the advantages of both BFBA and GBAVIII (i) by supplying a Bi-FIFO data transfer method between

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 14 in Section 7.3 for details).

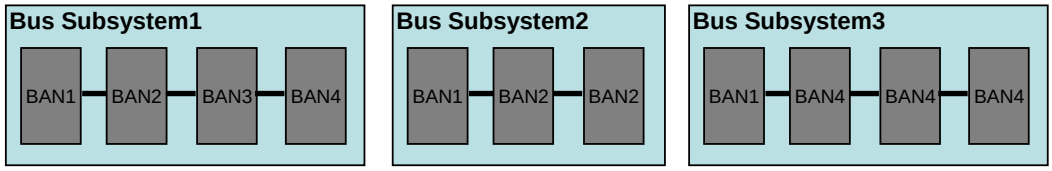
Different combination of bus components by the user options shown in Figure 5 is one way to make another Bus Subsystem as shown on the next page in Figure 9: (a) different combination of BAN components and (b) different combination of BANs. The detailed user options are introduced in Example 4.4.

Example 4.4 User Options for Different Combination of Bus Components

Figure 9(a) shows different combinations of BAN components. To generate BAN1, the user inputs "NONE" for CPU type (user option 3.1 in Figure 5), "NONE" for Non-CPU type (user option 3.2), "1" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4). Please note that SB in BAN1 inherits bus properties specified for the Bus Subsystem containing BAN1. Thus, in this case, the address and data bus widths of the SB were specified as follows: "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3). For BAN2, the user inputs "MPC755" for CPU type (user option 3.1), "NONE" for Non-CPU type (user option 3.2), "0" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4). SB in BAN2 is a segment of a bus with address bus width of "32" and data bus width of "64." Next, for BAN3, the user inputs "NONE" for CPU type (user option 3.1), "MPEG2 decoder" for Non-CPU type (user option 3.2), "0" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4). SB in BAN3 is a segment of a bus specified as follows: "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3). To generate BAN4 in Figure 9(a), the user inputs "MPC755" for CPU type (user option 3.1), "NONE" for Non-CPU type (user option 3.2), "0" for the number of global memories (user option 3.3) and "1" for the number of local memories (user option 3.4). SB in BAN4 is a segment of a bus with address bus width of "32" and data bus width of "64." GBI in BAN4 is



(a) Different Combination of BAN Components



(b) Different Combination of BANs

Note BAN: Bus Access Node, MBI: Memory Bus Interface, CBI: CPU Bus Interface, GBI: Generic Bus Interface, SB: Segment of Bus, IL: Interface Logic

Figure 9: Different Combination of Bus Components to Generate a New Bus Architecture

GBI_GBA – a GBI for a Global Bus Architecture (GBA) – that is put into BAN4 in order to interface the Local Bus connecting BAN4's MPC755 and SRAM to the non-Local Bus (i.e., a "global" bus) connecting BAN4 to other BANs.

Next, different combinations of BANs make different Bus Subsystems as shown in Figure 9(b). To generate Bus Subsystem1, the user enters "4" for the number of BANs (user option 2.1 in Figure 5), where the Bus Subsystem is composed of BAN1, BAN2, BAN3 and BAN4. As for Bus Subsystem2, the user inputs "3" for the number of BANs (user option 2.1), where the BANs are BAN1, BAN2 and BAN2. For Bus Subsystem3, the user inputs "4" for the number of BANs (user option 2.1), where the BANs are BAN1, BAN4, BAN4 and BAN4. □

We now introduce two other Bus Subsystem examples as shown in Figures 10 and 11: CoreConnect Bus Architecture (CCBA) from IBM and General Global Bus Architecture (GGBA). These Bus Subsystems are designed by hand rather than generated; CCBA and GGBA are used for the purpose of performance comparisons with the generated Bus Subsystems. In other words, we take CCBA and GGBA as examples of what a bus designer would typically do without a tool such as BusSynth. We

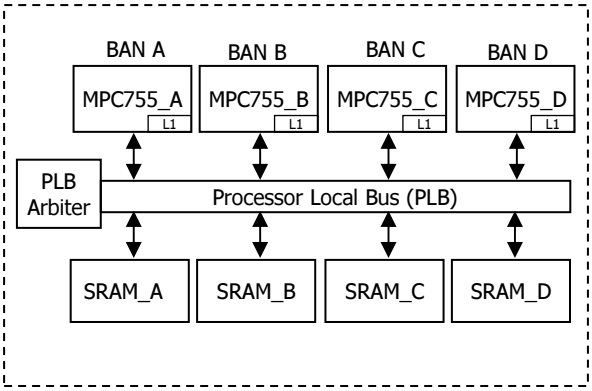


Figure 10: Diagram of CCBA

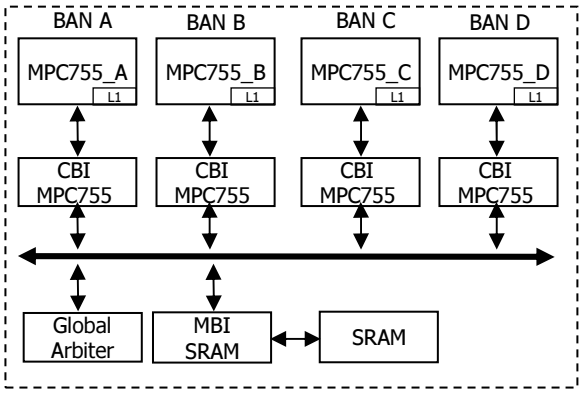


Figure 11: Diagram of GGBA

will show results of the performance evaluation between CCBA, GGBA and custom generated Bus Systems in Section 7.3. CoreConnect Bus Architecture (CCBA) shown in Figure 10 has a Processor Local Bus (PLB) that connects four MPC755s and four SRAMs (a total of 32MB size – 8MB per SRAM), where a PLB Arbiter arbitrates memory requests from the four MPC755s. General Global Bus Architecture (GGBA) shown in Figure 11 has a single global bus that provides a data path between four MPC755s and a single 32MB global memory SRAM. A Global Arbiter in Figure 11 arbitrates memory requests to the global SRAM from four MPC755s. While we use a total of 32MB of non-L1 cache memory in BFBA, CCBA and GGBA, GBAVIII and HybridBA have a total of 40MB memory; however, since each of our sample applications, which will be shown in Chapter 7, have instruction and data which completely

fit in the 32MB memory size, the memory size increase to 40MB for GBAVIII and HybridBA has no significant effect in the application performance (see Chapter 7 for details).

4.2 Communication among BANs

In a multi-processor 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 without any conflict in communication, which is what we desire, the overall system performance would be significantly increased in the parallel processing. In this section, 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 Subsystem in following sections (see Examples 4.7 and 4.8).

4.2.1 Our Basic Handshake Protocol

Our handshake protocol uses only two control signals. These signals are generated from two communicating PEs, a sender and a receiver. The protocol is different from a typical handshake protocol in that the typical handshake protocol needs three signals to control communication [17]. The typical handshake protocol uses three signals to keep 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, a receiver does not need to use condition (1) because a receiver needs to wait anyway until a sender has done its processing and saves data to a buffer for the receiver (please note that the receiver reads new raw data from the buffer after consuming old data so that old data in the receiver cannot be overwritten with the new data). Therefore, we eliminate condition (1), "read request," and thus use only two control signals for the conditions (2) and (3). The conditions are checked by the status (namely, "1" or "0") of each control signal, as shown in Example 4.5. 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 that may be necessary, and our generated bus architectures can support any such handshake protocols. However, in this dissertation we only show examples using the described protocol using only conditions (2) and (3).

Example 4.5 Handshake Control Registers

We denote two control registers as DONE_OP and DONE_RV, which output signals that correspond to condition (2) and condition (3), respectively. Each of the two registers has only one bit. 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 yet been received. After checking each condition, data is transferred from a sender to a receiver through a specific bus in each Bus Subsystem. □

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 4.6 in Section 4.2.2) 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 Subsystems containing a global bus style (e.g., GBAVIII, GGBA and CCBA), since we use control registers to generate the handshake control signals described before, and since multiple PEs can access the control registers at the same time, possible bus conflicts may occur. However, these possible conflicts can be resolved by exploiting an arbiter in the Bus Subsystem. The detailed communication procedures for each Bus Subsystem are shown in Examples 4.8 and 4.7. Please note that the specific handshaking protocol presented here can easily be replaced by a typical handshake protocol [17] or any other two-state or four-state handshake protocol with no effect whatsoever on the rest of the methodology presented in this dissertation.

4.2.2 Communication in GBAVIII

We introduce our handshake protocol for the communication in GBAVIII shown in Figure 7, repeated on the next page (for convenience) as Figure 12. 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.

In a pipelined parallel operation, output data from a PE is passed to the next PE for the subsequent operation in the application being executed. For the handshake protocol operation between BANs, GBAVIII shown in Figure 12 exploits global control variables saved in a specific region of a shared memory (e.g., Global SRAM in BAN G of Figure 12). Note that these variables work in a way similar to the control

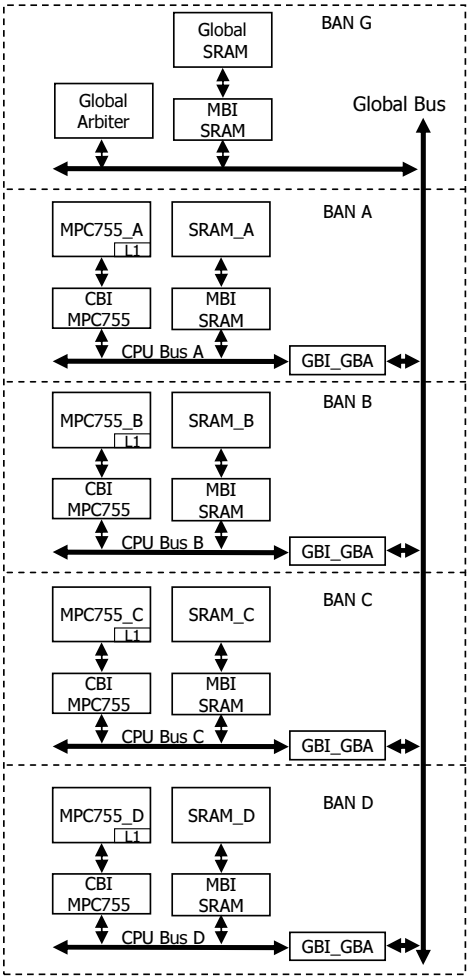


Figure 12: Diagram of GBAVIII (repeated from Figure 7 for convenience)

registers (e.g., DONE_RV and DONE_OP) introduced in Example 4.5. In this Bus Subsystem, 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 12. Example 4.6 shows passing processed data between BAN_B and BAN_C in GBAVIII working in a pipelined fashion. The other BANs in Figure 12 communicate in the same manner as shown in Example 4.6.

Example 4.6 Communication in GBAVIII Working in a Pipelined Parallel Fashion
We assume that BAN B and BAN C in GBAVIII, shown in Figure 12, execute an algorithm (e.g., OFDM transmitter that will be introduced in Section 7.1) in a pipelined fashion;

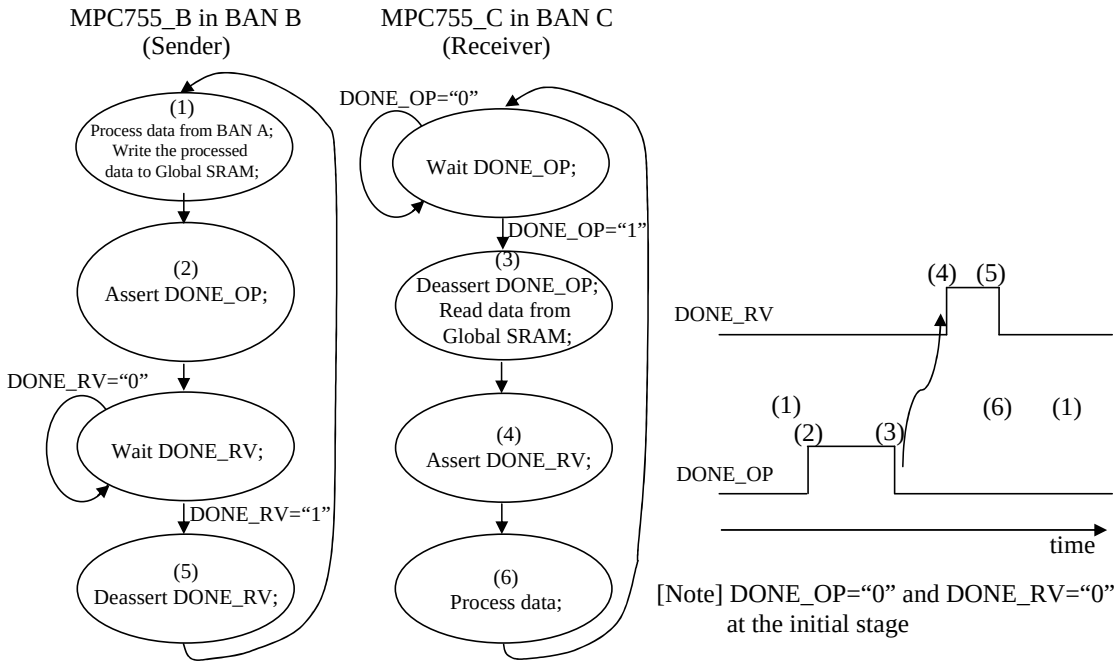


Figure 13: Communication between BANs in GBAVIII Working in a Pipelined Parallel Fashion

the result data from BAN B passes to BAN C through a shared memory Global SRAM in BAN G. To support our handshake protocol, handshake control variables saved in the shared memory work in a way similar to the the handshake control registers (e.g., DONE_RV and DONE_OP shown in Example 4.5). The control variables specified as DONE_RV and DONE_OP in this case can be accessed from both BAN B and BAN C. Note that the variables are initialized to "0," and the step numbers in the following procedure correspond to the numbers in Figure 13, which shows a communication state diagram. The procedure for data transfer from BAN B to BAN C is as follows.

(1) After MPC755_B processes BAN A's result data which was obtained using handshake control registers not shown in this example, MPC755_B writes 64 processed data words to Global SRAM (shown in Figure 12) starting from address 0x00000.

(2) MPC755_B sets DONE_OP to "1."

(3) MPC755_C resets DONE_OP to "0" after reading value "1" from DONE_OP. Using an API "mem_read(64, 0x000000, 0x400000)," MPC755_C reads the 64 words of data

from Global SRAM starting from address 0x000000 and stores the data to SRAM_C (shown in Figure 12) starting from address 0x400000.

(4) MPC755_C sets DONE_RV to "1."

(5) After MPC755_B reads "1" in DONE_RV, the MPC755_B resets DONE_OP to "0."

(6) MPC755_C processes stored data in SRAM_C in step (3).

□

By a functional parallel operation of GBAVIII shown in Figure 12, 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 dissertation, 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. Example 4.7 depicts the details of the communication procedure between BAN A and BAN B. The other BANs in Figure 7 can be handled in a similar fashion with additional handshake registers by extending the handshake algorithm presented in Example 4.7 in a straightforward fashion.

Example 4.7 Communication in GBAVIII Working in a Functional Parallel Fashion
We execute an MPEG2 decoder algorithm, which will be introduced in Section 7.1, in GBAVIII shown in Figure 7. 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 rather than in a pipelined operation; however, we still use data

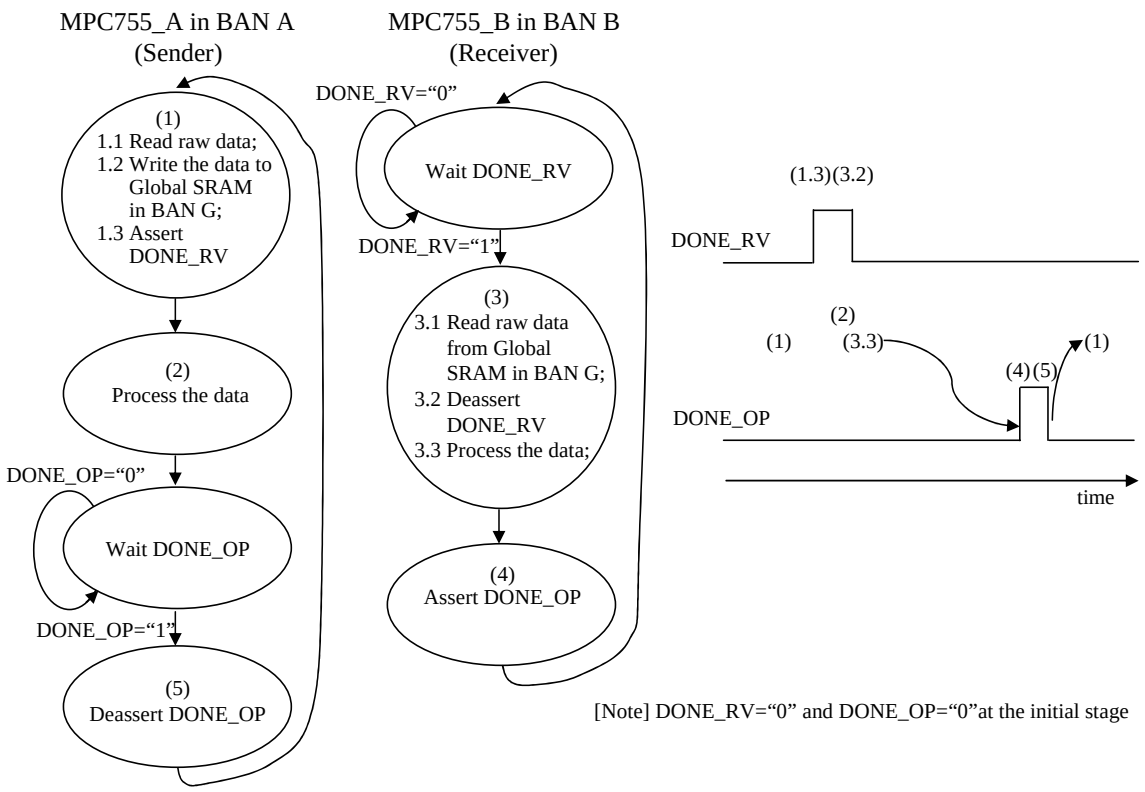


Figure 14: Communication between BANs in GBAVIII Working in a Functional Parallel Fashion

dependency in the handshake protocol to communicate between BAN A and BAN B since BAN B starts its data processing only after receiving raw data from BAN A, which works for raw data I/O as well as data processing, through a global memory Global SRAM in BAN G shown in Figure 4.2.2. BAN A reads a 1.47KB MPEG2 raw video stream, which is composed of Sequence Headers (SHs) and Groups Of Pictures (GOPs), from an external source and writes the stream data to an input buffer which is located in the global memory Global SRAM in BAN G shown in Figure 7. After such I/O processing, BAN A decodes the first SH and GOP while BAN B processes the second SH and GOP after reading the appropriate part of stream from the Global SRAM. In this manner, the video stream can be processed in parallel in each BAN (BAN A and BAN B). The step numbers in the following procedure correspond to the numbers in Figure 14. 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 7. As shown in Figure 14, which

depicts communication state diagram, the communication procedure between BAN A and BAN B is as follows.

(1) In BAN A, MPC755_A reads an MPEG2 video stream from a file, writes the stream to an "input buffer" (in the global memory Global SRAM of BAN G in Figure 14) for itself and for MPC755_B, and then sets the variable DONE_RV to "1."

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

(3) While MPC755_A computes as described in step (2), MPC755_B reads the second SH and GOP from the Global SRAM after reading a value "1" from DONE_RV, and then MPC755_B sets the variable DONE_RV to "0." After that, MPC755_B 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 in this example only BAN A and BAN B in Figure 14 perform MPEG2 decoding. While the above description was for the communication between BAN A and BAN B, additional handshaking (potentially requiring additional memory-mapped handshake registers in global memory) could be added in a straightforward fashion to include processing in BAN C and BAN D as well.

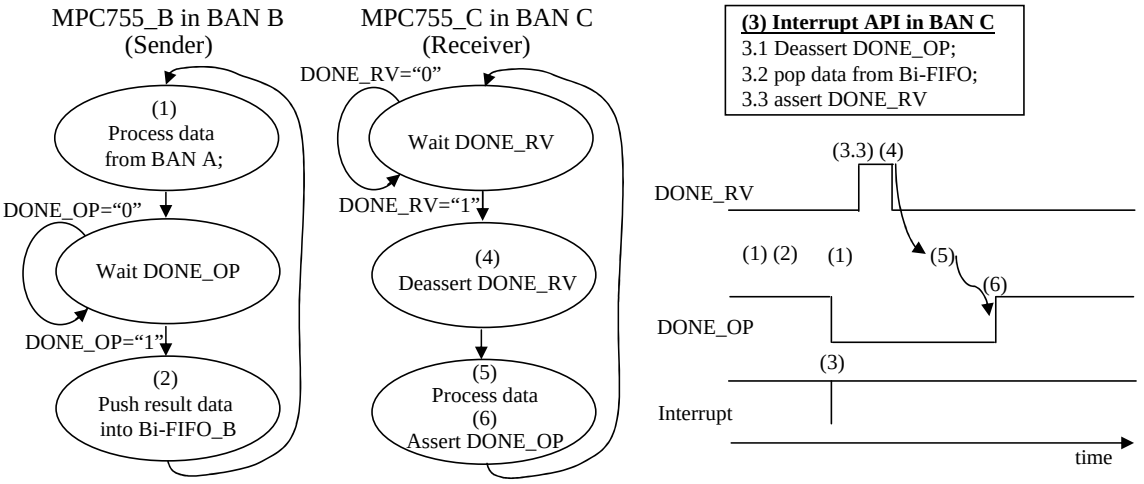
4.2.3 Communication in BFBA

PEs in BFBA shown in Figure 6 communicate using another adaptation of our handshake protocol in order to take advantage of an interrupt function. The handshake operation is implemented with an interrupt function and with two control registers DONE_OP and DONE_RV to generate the handshake control signals. These two

registers are contained in handshake registers' block "HS_REGS" in Figure 6, 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 6. As a sender pushes data into a Bi-FIFO memory in a receiver BAN, a Bi-FIFO counter in the controller of the receiving Bi-FIFO 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 PE so that an interrupt handler in the receiver PE 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 (i.e., using all intermediate PEs). In this case, the communication will incur some extra overhead; however, note that this Bus Subsystem also is suitable for a pipelined parallel style of operation, which usually has adjacent PEs communicating to each other. How to communicate between sender BAN B and receiver BAN C in Figure 6 is shown in Example 4.8. The other BANs' communication in Figure 6 works in the same manner as the procedure shown in Example 4.8.

Example 4.8 Communication in BFBA

We assume that BAN B and BAN C in BFBA, shown in Figure 6, execute an algorithm (e.g., OFDM transmitter that will be introduced in Section 7.1) in a pipelined fashion; the result data from BAN B passes to BAN C through Bi-FIFO_C in BAN C shown in Figure 6. 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 6). 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



[Note] 1. DONE_OP="1" and DONE_RV="0" at the initial stage
2. Pushing data to Bi-FIFO in (2) leads to the set (namely, "1") of DONE_RV (please see (2) and (3) steps)

Figure 15: Communication between BANs in BFBA

to the numbers in Figure 15, which shows a communication state diagram. As shown in Figure 15, the communication procedure between the BANs is as follows.

(1) MPC755_B in BAN_B processes BAN A's result data which was obtained using hand-shake control registers not shown in this example.

(2) MPC755_B pushes 64 words of processed data into Bi-FIFO_C in BAN C after reading "1" in register DONE_OP. Please note that MPC755_B's pushing of 64 words into Bi-FIFO_C leads to the setting (namely, "1") of DONE_RV (see the following step (3)).

(3) An interrupt handler API in MPC755_C runs based on Bi-FIFO_C filling up after MPC755_B has finished pushing the output data. As shown in the interrupt API in Figure 15, the API resets DONE_OP to "0," pops the sixty-four words of data from the Bi-FIFO_C and then sets DONE_RV to "1."

(4) MPC755_C resets DONE_RV to "0" after reading "1" in register DONE_RV.

(5) MPC755_C processes the popped data.

(6) MPC755_C sets DONE_OP to "1."

□

4.2.4 Communication in HybridBA

As introduced earlier, HybridBA shown in Figure 8 is a combined Bus Subsystem of BFBA (see Figure 6) and GBAVIII (see Figure 7); in other words, HybridBA has both the Bi-FIFO blocks and the global bus. Therefore, in the case of using the Bi-FIFO blocks in the HybridBA Bus Subsystem, the communication procedure is the same as the procedure shown in Example 4.8. On the other hand, when using the global bus in the HybridBA, the procedure is the same as that of GBAVIII as shown in Example 4.7.

4.3 Summary

In this chapter, based on the user options, we have shown how to specify various Bus Subsystems together with their detailed examples. We have also explained several ways to communicate among PEs in the generated Bus Subsystems. In the next chapter, we will show our specification method for Bus Systems together with several examples. Then, we will explain communication methods among PEs in the specified Bus Systems.

CHAPTER V

BUS SYSTEM SPECIFICATION

A Bus System is composed of one or more Bus Subsystems connected together with one or more bus bridges. Thus, a Bus Subsystem is a subset of a Bus System, but not vice versa. In this chapter, we show the specification of various Bus Systems, based on the user options described in Section 4.1, in detail.

5.1 Bus System Examples

We show two Bus System examples in this section: GBAVI and SplitBA, where we describe how to specify the Bus Systems by using the user options shown in Figure 16 (repeated here from Figure 5 of Section 4.1 for convenience).

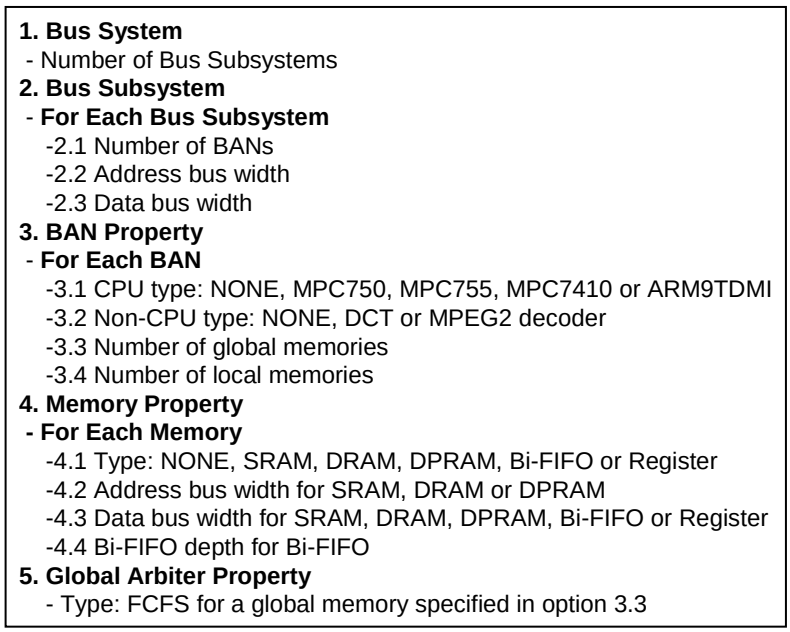


Figure 16: User Options to Configure a Custom Bus System (repeated from Figure 5 for convenience)

5.1.1 How to Generate Bus Systems

Here we generate a Bus System we call Global Bus Architecture Version I (GBAVI). Example 5.1 shows how to input the user options to generate GBAVI.

Example 5.1 User Options for the Generation of GBAVI

In this example, we generate a Bus System we call Global Bus Architecture Version I (GBAVI); to accomplish this, we specify the user options as follows. First, the user specifies the number of Bus Subsystems by entering an "8" in Bus System Property (user option 1 in Figure 5). Next, for each of the eight Bus Subsystems, the user inputs "1" for the number of BANs (user option 2.1), "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3). Please note that all Local Buses and all non-Local Buses in

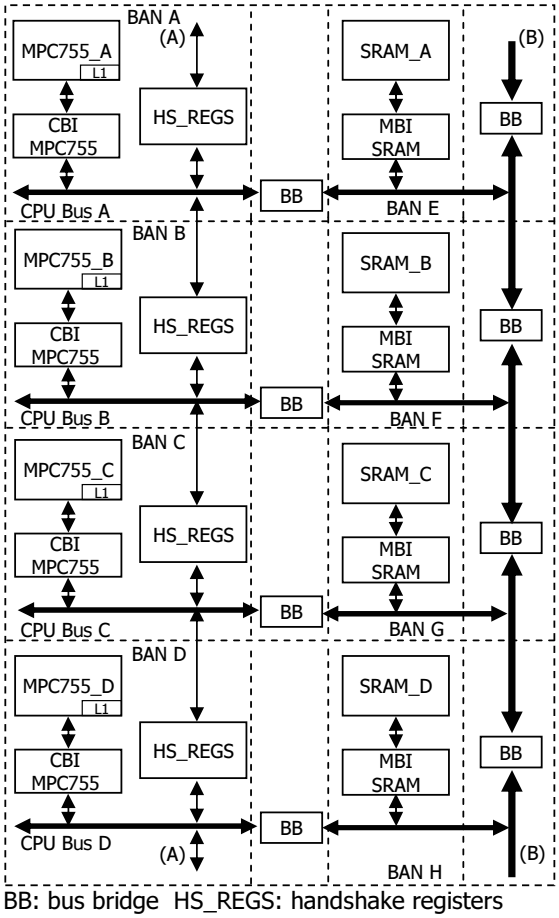


Figure 17: Diagram of GBAVI

Figure 17 have the same address and data widths; as mentioned earlier in Example 4.1, this could be easily changed if desired. Next, the user inputs the following BAN Property fields for each single BAN in four of the eight Bus Subsystems: "MPC755" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3) and "1" for the number of local memories (user option 3.4). Next, the user inputs the Memory Property for the local memory chosen for each BAN as follows: "Register" for the memory type (user option 4.1) and "1" for the data bus width (user option 4.3). With these options, BANs A, B, C and D in Figure 17 are generated, where each BAN is composed of MPC755, CBI MPC755 as an interface logic block and HS_REGS as a register block (please note that CBI MPC755 is integrated into each BAN as a result of the selection of MPC755 for each BAN).

Then, the user again inputs the fields of BAN Property for each single BAN in the remaining four Bus Subsystems: "NONE" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3) and "1" for the number of local memories (user option 3.4). Next, the user inputs the Memory Property for the local memory chosen for each BAN as follows: "SRAM" for the memory type (user option 4.1), "20" for the address bus width (user option 4.2) and "64" for the data bus width (user option 4.3) for 8MB SRAM in each BAN. With these options, BANs E, F, G and H in Figure 17 are generated, where each of the four BANs has an 8MB SRAM (please note that MBI SRAM in each of BANs E, F, G and H shown in Figure 17 is integrated into each BAN when the user specifies SRAM for the BAN).

Each Bus Subsystem in this example has a single BAN, and the Bus Subsystems need to be integrated to form a Bus System we call GBAVI. In this case, to connect each Bus Subsystem, our tool automatically inserts Bus Bridges (BBs) (e.g., the BB shown between BAN A and BAN E in Figure 17) between the Bus Subsystems. □

GBAVI shown in Figure 17 has a kind of global bus architecture (GBA), but the global bus is segmented by BBs separating each BAN, where the number of BANs is specified by the user. Each BAN has an SRAM block (e.g., SRAM_A, SRAM_B,

SRAM_C or SRAM_D). One BB in each BAN controls a 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 thanks to separation provided by the BBs. For example, in Figure 17, while BAN A and BAN B communicate with each other, BAN C and BAN D also can communicate at the same time without any bus conflict. Each group of two BANs in Figure 17 is synchronized by handshaking using shared registers (HS_REGS) between BANs (see Section 4.2 for a description of the handshake protocol). Note that GBAVI tends to work well in a pipelined style operation; for example, the output of a PE (e.g., MPC755_A) is passed to the next PE (e.g., MPC755_B).

As another custom Bus System, a user might want to have in a Bus System two Bus Subsystems, where each Bus Subsystem has PEs and a single global memory that are connected with a single global bus. We call this Bus System "SplitBA" (for Split Bus Architecture). Example 5.2 shows how to input the user options to generate SplitBA.

Example 5.2 User Options to Generate SplitBA

A user input sequence which specifies Split Bus Architecture (SplitBA) is as follows. The

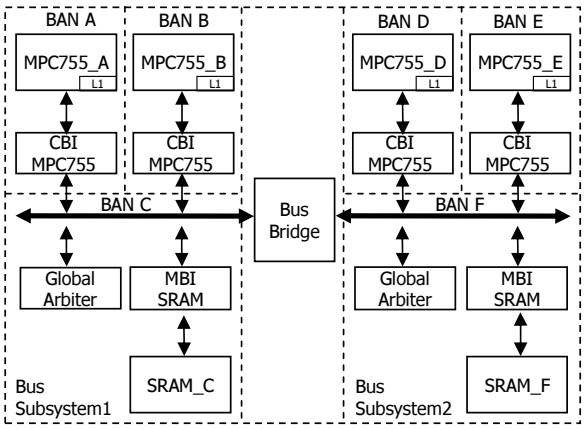


Figure 18: Diagram of SplitBA

user first specifies the number of Bus Subsystems by entering a "2" in Bus System Property (user option 1 in Figure 5). Then, for each of the two Bus Subsystems, the user inputs "3" for the number of BANs (user option 2.1), "32" for address bus width (user option 2.2) and "64" for data bus width (user option 2.3). Please note that all Local Buses and all non-Local Buses in Figure 18 have the same address and data widths; as mentioned earlier in Example 4.1, this could be easily changed if desired. Next, the user inputs the following fields of BAN Property for two BANs in each Bus Subsystem: "MPC755" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "0" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4). This results in BANs A, B, D and E in Figure 18. For the last remaining BAN in each Bus Subsystem, the user inputs "NONE" for the CPU Type (user option 3.1), "NONE" for the Non-CPU Type (user option 3.2), "1" for the number of global memories (user option 3.3) and "0" for the number of local memories (user option 3.4); the Memory Property for this global memory BAN is input as "SRAM" for the memory type (user option 4.1), "21" for the address bus width (user option 4.2) and "64" for the data bus width (user option 4.3). The results are two BANs each with 16MB SRAM as seen in BANs C and F in Figure 18. The total amount of non-cache memory is 32MB. With these options, the generated Bus System which we call SplitBA is shown in Figure 18. □

SplitBA shown in Figure 18 is composed of two Bus Subsystems each of which has two MPC755s and a 16MB memory. The two Bus Subsystems of Figure 18 are connected through a bus bridge to exchange data between them. Both Bus Subsystems in Figure 18 can operate at the same time without bus contention so that system performance is increased. In addition, in each Bus Subsystem, a bus length shorter relative to using a single GBA may allow the use of a faster bus clock, thus speeding up computation in the system; furthermore the shorter buses may even consume lower power due to lower parasitic resistance and capacitance in the buses in the SoC (please note that the power consumption in a bus bridge cannot

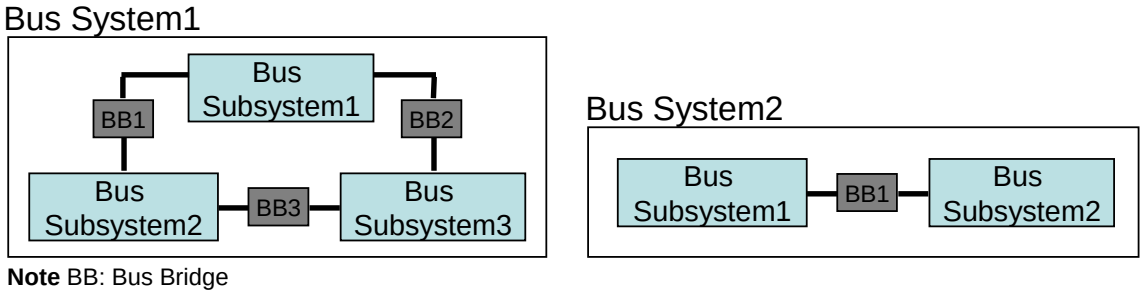


Figure 19: Different Combination of Bus Subsystems to Generate New Bus Architectures

be ignored) [19]. Due to its divided bus, SplitBA also relieves bus traffic congestion caused by shared memory requests from each BAN.

A different combination of Bus Subsystems by the user options shown in Figure 16 (repeated from Figure 5) results in a different (and possibly brand new) Bus System as shown in Figure 19. Example 5.3 shows how to generate another Bus Architecture based on the user options.

Example 5.3 User Options for Different Combinations of Bus Subsystems

Figure 19 shows examples where different combination of Bus Subsystems makes new Bus Systems. Bus System1 in Figure 19 has user option where the number of Bus Subsystems is "3" (user option 1 in Figure 5), and the Bus System is composed of Bus Subsystems 1, 2 and 3. The BBs (namely, BBs 1, 2 and 3) can be automatically selected in our tool according to the type of non-Local Bus architecture (e.g., GBA or BFBA) using each Bus Subsystem. To generate Bus System2, the number of Bus Subsystems is set to "2" (user option 1), where the system is composed of Bus Subsystems 1 and 2. Please note that while our current tool supports only one BB type (namely, BB_GBA), more BB types to support specific Bus Systems shown in Figure 19 could easily be added to our tool after being defined. □

As shown in Examples 5.1 and 5.2, If a Bus System has more than one Bus Subsystem, the Bus Subsystems need to be integrated to form a Bus System. In

that case, our tool BusSynth inserts a Bus Bridge (BB) between Bus Subsystems. Currently, BusSynth does not supports BBs that enable different bus speeds, which can be caused by different bus clocks, in buses connected by the BBs; however, this issue can be solved by integrating a memory (e.g., FIFO) into the BBs in order to adapt the different bus speeds, and such a BB could easily be added to Module Library, which will be described in Chapter 6 in detail.

5.1.2 Communication among BANs

Since user applications running on a multi-processor SoC are typically partitioned across multiple PEs for parallel processing, the system performance is heavily affected by method of communication among the PEs. In this section, we show a communication method among PEs in a Bus System, based on our basic handshake protocol described in Section 4.2.

5.1.2.1 Communication in GBAVI

We show how our specific handshake protocol used to communicate between PEs in the GBAVI Bus System shown in Figure 17. To support our protocol (i.e., to generate the handshake control signals), two control registers, DONE_OP and DONE_RV shown in Figure 20, reside in the handshake registers' block (HS_REGS) shown in each BAN of Figure 17. Each pair of neighboring PEs shares the registers (i.e., both a sender and a receiver can access the registers). When non-adjacent PEs have to communicate with each other (e.g., transferring data from MPC755_A to MPC755_C in Figure 17), 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., the BBs between MPC755_A and MPC755_C). Note that GBAVI, as implemented, tends to work well in a pipelined parallel style of operation that has a pattern in which output data from a PE is passed

to the next PE for the following operation. The communication procedure among PEs in GBAVI working in a pipelined fashion can be implemented in the same manner as the handshake protocol shown in Example 4.6 in Chapter 4. In GBAVI, two control registers (e.g., DONE_RV and DONE_OP) work in a way similar to handshake control variables (shown in Example 4.6) saved in the shared memory Global SRAM shown in Figure 7.

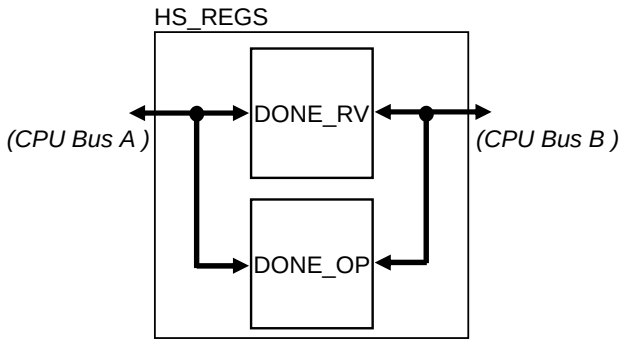


Figure 20: Detailed Diagram of HS_REGS in Figure 17

5.1.2.2 Communication in SplitBA

SplitBA shown in Figure 18 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 18). Through the shared memory, the communication procedure among PEs can be implemented in the same manner as shown in Examples 4.6 and 4.7 in Section 4.2.

5.1.3 Summary

Based on the user options, we have shown how to specify a Bus System; furthermore, we have included some detailed examples. We have also briefly described how to communicate among PEs in a generated Bus System. In the following chapter, we will depict detailed methodology of how to generate a user-specified Bus System,

where we will show our algorithm used in the methodology, our way of interconnect delay aware bus generation, and computational complexity of the algorithm.

CHAPTER VI

METHODOLOGY FOR BUS SYSTEM GENERATION

Based on the Bus System structure described in Chapter 3 and user inputs described in Chapters 4 and 5, our bus synthesis tool BusSynth generates a user-specific Bus System. We use two kinds of libraries which we refer to as Module Library and Wire Library. In this chapter, we show the methodology behind our approach to generate a custom user-specified Bus System. In Section 6.1, we show how the Module Library and the Wire Library are made and work in the tool. Then, in Section 6.2, we explain how to generate Bus Systems using the Libraries. Thus, Section 6.2 covers the main point of this chapter: detailed methodology and pseudo code for Bus System generation. Section 6.3 describes interconnect delay aware Bus System generation based on the methodology described in Sections 6.1 and 6.2. Finally, Section 6.4 ends with an analysis of the computational complexity of the algorithms shown in Section 6.2.

6.1 Libraries for Module Repository and Wiring

BusSynth uses two libraries to generate a Bus System. One is a Module Library that contains all modules currently supported for use inside a BAN, a Bus Subsystem and/or a Bus System. The other library is a Wire Library that contains many different specific wires for connecting the modules inside BANs, Bus Subsystems and a Bus System. The Module Library contains not only Input/Output (I/O) port information and behaviour of each module in RTL Verilog code but also many templates to generate specific modules (e.g., ARBITERs). Here, the templates have parameters

to configure each of the specific modules that the user wants through the user options that will be introduced in Section 6.2 in detail, and the modules are generated by assigning specific values to the parameters whose values are from the user input options, based on the user requests. Each library component is described in text in a file and starts and ends with a specific keyword, respectively: "%module <library name>" and "%endmodule <library name>." The parameters to be configured in a library component are specified with another specific keyword "@parameter@." These keywords are shown in Example 6.1 in detail. The Module Library contains the following components:

(a) <PE>: a processing element, where <PE> is one of MPC750, MPC755, MPC7410 or ARM9TDMI (note that more PE types could easily be added if desired)

(b) CBI_<PE>: an interface module between a PE (or CPU) and a bus

(c) <memory>_comp: a memory template to be used to generate any size of behavioral memory, where <memory> is one of SRAM, DRAM, DPRAM, Bi-FIFO or Register

(d) MBI_<memory>: an interface module between a <memory> and bus, where <memory> is one of SRAM, DRAM, DPRAM, Bi-FIFO or Register

(e) BB_<bb_type>: a bus bridge, where <bb_type> is GBA (please note that more BB types – e.g., BBs to support specific Bus Systems shown in Figure 19 – could easily be added after being defined)

(f) ARBITER_<arb_type>: an arbiter module, where <arb_type> is FCFS (please note that more ARBITER types could easily be added after being defined)

(g) ABI: an interface module between an arbiter and a bus, where the Module Library currently has only ABI_FCFS (please note that more ABI types could easily be added after being defined)

(h) GBI: a generic bus interface module, where the Module Library currently has only GBI_GBA and GBI_BFBA (please note that more GBI types could easily be added after being defined)

(i) SB: a module for Segment of Bus (SB), where the Module Library currently has only SB_GBA and SB_BFBA, where SB_GBA is for GBAVI, GBAVIII and a global bus architecture in HybridBA, and SB_BFBA is for BFBA and a Bi-FIFO bus architecture in HybridBA (please note that more SB types could easily be added after being defined)

In our current tool, GBI_GBA can be used for GBAVIII, HybridBA and SplitBA since these Bus Systems use a Global Bus Architecture (GBA) to connect BANs in each Bus Subsystem, GBI_BFBA is for BFBA as well as the Bi-FIFOs in HybridBA, and GBAVI does not need a GBI since the Bus Subsystem has a single BAN. If a user wants to generate a new Bus System (shown in Figure 19) without using a global bus architecture or a Bi-FIFO bus architecture, the user needs to add new components (e.g., a new GBI and/or a new SB) required to the Module Library. However, currently our tool works well to generate a Bus System with a global bus architecture, a Bi-FIFO bus architecture, a combined bus architecture with the global bus architecture and the Bi-FIFO bus architecture (e.g., HybridBA), or a combined bus architecture having more than one global bus (e.g., SplitBA). Actually, Bus Systems with one or more than one Global Bus Architecture (GBA) to connect BANs are a popular in industry today; thus we are able to address current needs quite well.

Example 6.1 shows an example of the Module Library including how the different parameters in each library component are taken into consideration when performing adaptation between heterogeneous hardware components (e.g., between a bus and an SRAM). Here, the different parameter values are based on the user input options.

```
%module MBI_SRAM
module mbi_sram(hrst_bar, abb_bar, cs_bar, sram_web,
                        // Skip I/Os
                        sram_oeb, sram_addr, sram_dq);

// Parameter definitions
parameter MEM_A_WIDTH = @parameter@;
parameter MEM_D_WIDTH = @parameter@;
parameter DLY_PE1 = @parameter@;
parameter DLY_PE2 = @parameter@;
parameter DLY_PE3 = @parameter@;
parameter DLY_PE4 = @parameter@;

// I/O definitions
input HRST_BAR;
input [0:3] ABB_BAR;
input [0:7] CS_BAR;
output sram_web;
output sram_oeb;
output [MEM_A_WIDTH-1:0]  sram_addr;
inout   [MEM_D_WIDTH-1:0]  sram_dq;
// Skip I/O definitions

// Register definitions
reg [0:3]    RnumRdDelay;
// Skip register definitions
// Assign delay values
  always @(cs_bar or hrst_bar)
    begin
      if (~hrst_bar)
              RnumRdDelay <= 4'h0;
      else if (~cs_bar)
            if(~abb_bar[0])
              RnumRdDelay <= DLY_PE1;
        else if (~abb_bar[1])
              RnumRdDelay <= DLY_PE2;
        else if (~abb_bar[2])
              RnumRdDelay <= DLY_PE3;
        else if (~abb_bar[3])
              RnumRdDelay <= DLY_PE4;
      else
             RnumRdDelay <= 4'h0;
    end

   // Skip verilog description

endmodule
%endmodule MBI_SRAM
```

Figure 21: MBI_SRAM Component in Module Library

Example 6.1 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 6, 7, 8, 17 and 18 when a user wants to attach an SRAM to a bus through the user options in Section 4.1. In Figure 21, the library component name is shown in the first line, "%module <library_name>," where <library_name> is MBI_SRAM. To specify the library's property, there are six different parameters: physical memory address width, memory data width and four interconnect delay parameters for PEs 1 to 4. These parameters are set in a module

generation procedure based on the user options and interconnect delay inputs, where the module generation procedure using the Module Library and interconnect delay aware module generation with interconnect delay input will be described in Section 6.2 and Section 6.3 in detail, respectively. For the interface between CPU bus A and the 8MB SRAM in BAN A of Figure 6, the first two parameters (namely, "MEM_A_WIDTH" and "MEM_D_WIDTH") shown in Figure 21 are set to "20" and "64", respectively. The parameter values are from the Memory Property (e.g., MEM_A_WIDTH: 20 and MEM_D_WIDTH: 64) in the user options 4.2 and 4.3 shown in Figure 16. The other four parameters (namely, DLY_PE1 to DLY_PE4)) are set to 3, 3, 4 and 5, repectively, when interconnect delay clocks to be inserted to memory access cycles are 3, 3, 4 and 5 cycles from PE1 to PE4 (see Section 6.3 for the detailed calculation of interconnect delay clocks). Please note that we assume that all addresses which appear on a bus are physical addresses. Any virtual address used by a program must be translated to a physical address prior to placement on a bus. □

The Wire Library contains all possible combinations of legal connections between hardware blocks (e.g., between modules in each BAN, between BANs in each Bus Subsystem) or between Bus Subsystems in a Bus System. This library is written in ASCII format as shown in Figure 22, 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 22: Wire Library Format

In the Wire Library format shown in Figure 22, two modules are connected by the wire, namely, module *m1_name* and module *m2_name*. To specify a single wire connecting three or more distinct ports, an additional wire entry is needed for each additional port beyond two. Please note that the *m1_pname* field specifies the port to which the wire connects in module *m1_name*, while the *m2_pname* specifies the port to which the wire connects in module *m2_name*. Thus, in a sense, two "ports" are specified in each Wire Library entry! These two ports are not, strictly speaking, "part of" the wire; nonetheless, since the wire connects the two ports, the two ports are part of the Wire Library format. Example 6.2 shows wire connections between two modules within the same BAN. Note that the *m1_name* and *m2_name* fields may be the same when a connection specifies either (1) a single wire between more than two ports on different modules (or BANs) or (2) a set of similarly-named wires (except for a suffix) forming a torus among more than two ports on different modules (or BANs). Example 6.3 shows such wire connections between different BANs. Please note that to specify a wire between/among BANs that have the same I/O ports in their pin names in a Bus Subsystem (e.g., the connection between BAN A and BAN B in Figure 6), *m1_name* and *m2_name* in Figure 22 need to be the same. This case is described in Example 6.3 in detail, where Figure 25(b) shows detailed blocks and I/O pins that are related to each BAN's I/O ports shown in Figure 25(a).

Example 6.2 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 23 (repeated here from Figure 6 for convenience). Figure 24

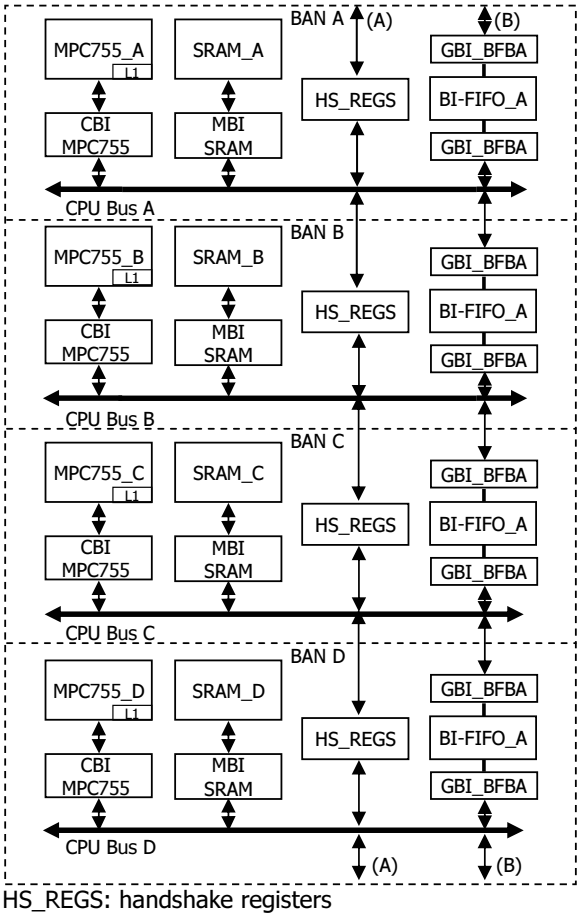


Figure 23: Diagram of BFBA (repeated from Figure 6 for convenience)

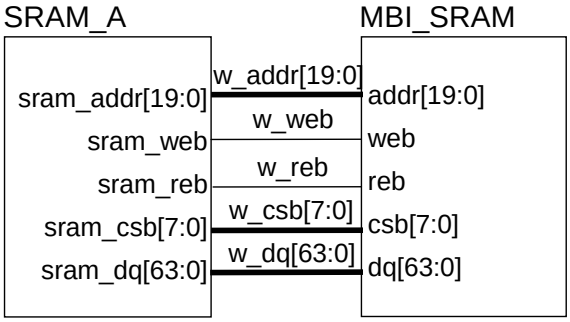


Figure 24: Wire Connection Example between SRAM_A and MBI_SRAM in Figure 6

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 24, 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

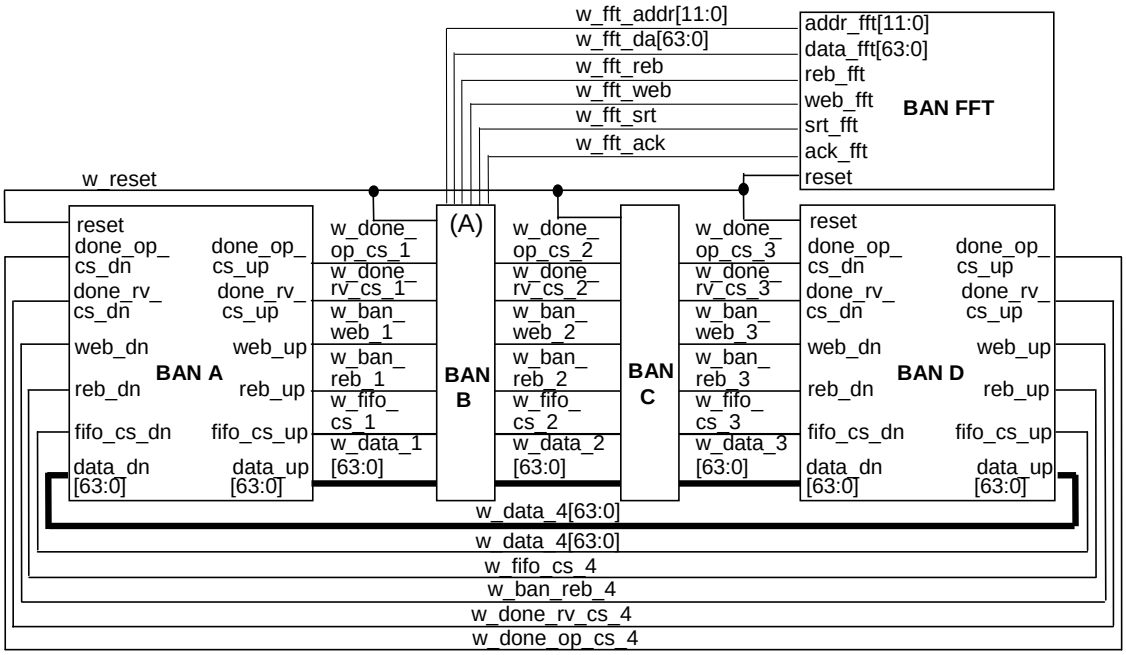
w_csb 8 SRAM_A sram_csb 7 0 MBI_SRAM csb 7 0

w_dq 64 SRAM_A sram_dq 63 0 MBI_SRAM dq 63 0

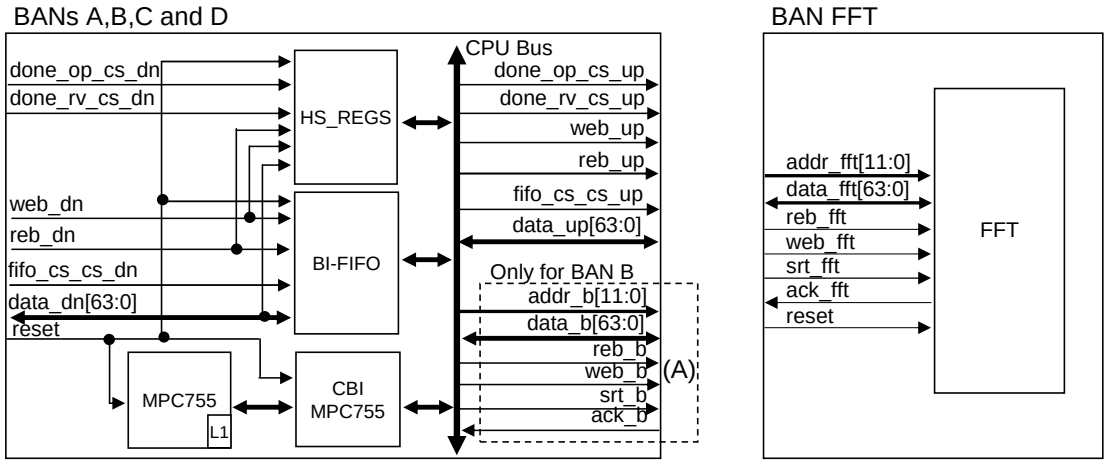
%endwire □

Example 6.3 Wire Connections between BANs in a Bus System

This example shows how to form wire connections between multiple (more than two) BANs in the Wire Library. BANs A, B, C and D are linked as in a chain as shown in Figure 25(a), and the connections of the I/O ports shown in the left box (with solid lines) of Figure 25(b) are repeated between the BANs. In this kind of wire connection, the names of the wires connecting the BANs have the same names but with different suffixes as shown Figure 25(a), except for one case: w_reset. The w_reset wire does not have a suffix, and w_reset is the only wire that connects to all the BANs with a single contiguous wire, as opposed to just connecting one BAN to another. In the example of Figure 25, it is not necessary that we specify all wires individually. Thus, although the Wire Library format technically only supports the specification of a wire connecting two ports (from up to two different BANs), nevertheless our tool supports wire specifications such as shown in this example. The wire specifications shown in this example result in the serial connections (wires) linking the specified BANs by generation of wires suffixed by an enumerated number as shown in Figure 25(a). For this purpose, wire connections between more than two BANs can be specified by the same module names in the *m1_name* field and the *m2_name* field in the Wire Library format as shown in Figure 22; in this example, the names are just "BAN[A,B,C,D]" as shown in the wire listing "wire subsystem_bfba" at the end of this example (on page 67).



(a) Connection among BANs



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

Figure 25: Wire Connection Example between BANs

A set of wires having the same module names in the *m1_name* field and the *m2_name* field and different port names in the the *m1_pname* field and the *m2_pname* field describe a torus network. Here, "BAN[A,B,C,D]" means that the specified wire connecting the named ports is applied for BANs A, B, C and D. On the other hand, the wires between BANs having connections other than a simple contiguous wire or a torus network have to be specified separately in the Wire Library; for example, as shown below, single explicit wires are specified connecting BAN B and BAN FFT. The connections between BAN B and BAN FFT in Figure 25(a) show the case where we assume that BAN B has another bus to BAN FFT in addition to the buses connecting BANs A, B, C and D. Here, BAN FFT is a BAN having a hardware Fast Fourier Transform (FFT) core.

Detailed wire connections between a pair of BANs A, B, C and D in Figure 25(a) are as follows: w_done_op_cs or w_done_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 25(a). In the connections between BAN B and BAN FFT, the wires are as follows: w_fft_ad for FFT buffer address, w_fft_data for the 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 Figure25(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_reset 1 BAN[A,B,C,D] reset 0 0 BAN[A,B,C,D] reset 0 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

w_reset BAN[B] reset 0 0 BAN[FFT] reset 0 0

%endwire □

As stated earlier, please note that the wire library contains at a minimum all legal connections among Modules (e.g., PEs, BBs, SBs, arbiters, memories and ILs including GBI), where by a "legal" connection we mean a connection which makes clear functional sense, e.g., between two 32-bit address ports. However, in a case where a specialized non-"legal" connection, e.g., from bit 3 of an address port in GBI to a clock input, is desired, such a case can be supported by manually entering the wire into the Wire Library.

To specify a port in a module, we use port direction (namely, input, output or inout), port name, MSB and LSB of the port width for each port of the module. Thus, a record of port information contains the four properties port direction, port name, MSB and LSB in a data structure. In order to specify the ports in the module, a record for each port is required. Example 6.4 shows an example of the port information.

Example 6.4 A Record of Port Information

Suppose that we want to describe a port "addr_fft[11:0]" of BAN FFT shown in Figure 25(b). A record for the port information contains "input," "addr_fft," "11" and "0" in a port data structure. □

6.2 Sequence of Bus System Generation

We now show how to generate a Bus System. First, we describe the overall flow of bus synthesis as shown in Figure 26. Next, we explain the user options to configure

the Bus System to be generated from our bus synthesis tool BusSynth. Third, we describe how to generate the wires to interconnect the modules of a specific hardware unit (e.g., a BAN or a Bus Subsystem) that is to be part of the specified Bus System. Fourth, we present our algorithm for Bus Subsystem generation. Finally, we describe our algorithm for Bus System generation.

6.2.1 Overall Flow of Bus System Generation

The flowchart shown in Figure 26 shows the Bus System generation sequence. First, BusSynth takes user input options for a Bus System to be generated, and then, based on the options, BusSynth generates the required BANs and assembles them into the required Bus Subsystems. After that, if the Bus System the user wants has more than one Bus Subsystem, the generated Bus Subsystems are integrated into the resulting Bus System. Otherwise, the generated single Bus Subsystem becomes

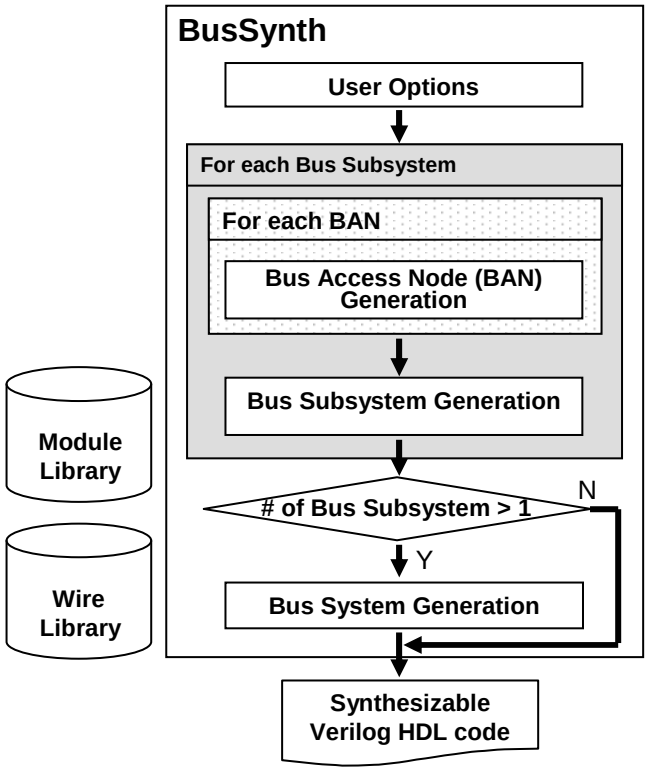


Figure 26: The Bus System Generation Sequence

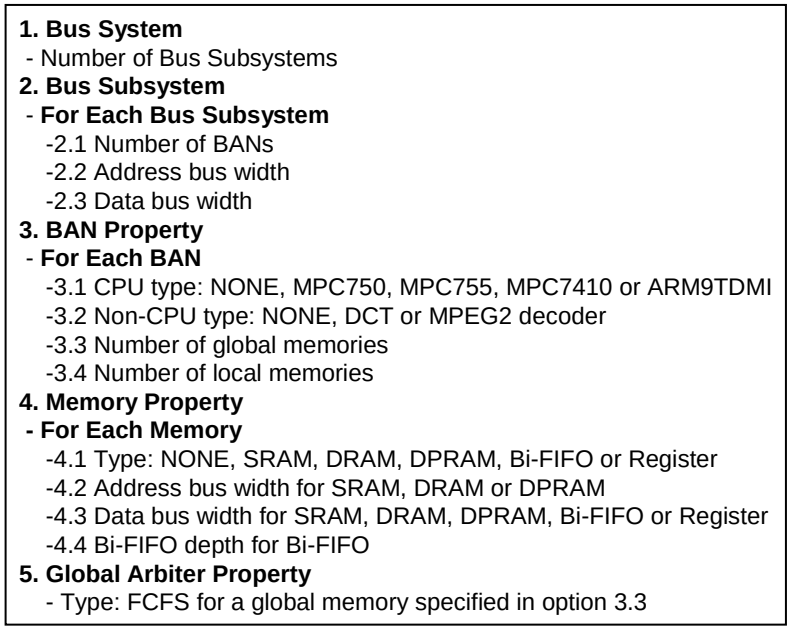


Figure 27: User Options to Configure a Custom Bus System (repeated from Figure 5 for convenience)

the Bus System. Finally, BusSynth writes synthesizable Verilog HDL code for the generated Bus System.

6.2.2 User Inputs to BusSynth

As the first step of the flowchart shown in Figure 26, to configure the custom Bus System, the user enters input options. These options are input constraints used to generate a custom Bus System. We already described in detail the user options in Chapter 4; the user options are summarized in Figure 27. Section 5.1 shows the input sequence of the user options and detailed examples of how to generate Bus Systems based on the user options.

6.2.3 Unit Generation

We introduce here an algorithm, UnitGen, which is a part of BusSynth's algorithms and is used to generate in HDL a hardware unit that is specified to be part of the Bus System desired by the user. In particular, given a list of modules as input,

UnitGen generates the wires needed to connect all the modules together appropriately. Depending on the input list (array) of module names, UnitGen can generate a BAN, a Bus Subsystem or a Bus System. UnitGen (short for "Unit Generator") is used by (called from) algorithms that will be introduced in Sections 6.2.4 and 6.2.5.

Listings 1 and 2 show the pseudo code of UnitGen. The input arguments are an array of module names, the name of the top hardware unit to be generated and a pointer to the Wire Library; the output is HDL code of the top hardware unit. The input array of module names contains all the names of all modules in a top hardware unit to be generated. Since UnitGen integrates modules specified in the module name array *M*, which is input (see Listing 1), such modules to be integrated are provided as separate HDL files (extracted from the Module Library shown as available in Figure 26). However, while UnitGen does not use the Module Library explicitly, UnitGen does use the Module Library implicitly by use of the Wire Library to generate wires for the specified design.

Example 6.5 An Array of Module Names Input to UnitGen

Consider the case where we generate a hardware unit, BAN A of BFBA shown in Figure 6. UnitGen, shown in Listing 1, takes as input an array of module names that contains MPC755, MBI_SRAM, HS_REGS, CBI_MPC755, SRAM_A and Bi-FIFO since these six modules are the components of BAN A. □

In lines 1 to 10 of Listing 1, to connect modules specified in an array of module names, UnitGen first extracts specific wires to connect between modules from a Wire Library file; this wire information is placed in a data structure LW1. Here, each record of LW1 is composed of the same fields as the fields shown in the Wire Library Format of Figure 22. In lines 14 to 21, port information of each module to be integrated is read from separate HDL files that were generated for each module in

Listing 1 *UnitGen():* Unit Generation.

```
Input: module_name_array M, top_unit_name_to_be_gen U and wire_library_file *W
Output: HDL code of an unit named U
begin
   /* lines 1 to 10: read wires from W that connect modules specified in M */
1: LW1 = φ; /* LW1 is a set Listing Wire information obtained from W; initially, LW1 is empty */
2: n_m = size(M);      /* n_m is the number of modules stored in M */
3: while(!end of line in W) do      /* each line of W contains a record of wire information */
                                    /* for one wire */
4:      for(i=1 to n_m, i=i+1) do
            /* the format of wire information has module name fields as shown in Figure 22 */
5:          if(current line pointed to by W contains M[i]) then
6:              LW1 = LW1 ∪ wire information in current line;
7:          end if
8:      end for
9:      go to next line in W;
10: end while

    /* lines 11 to 34: generate sets (LWPM and LP2) required for HDL generation */
11: LWPM = φ;      /* LWPM is a set Listing information of Wire-Port Mapping for U; */
                   /* here, the set is composed of five fields (wire name, wire LSB, wire MSB, */
                   /* port name and module name; initially, LWPM is empty */
12: LP2 = φ;       /* LP2 is a set Listing Port information that is composed of port name, */
                   /* port direction and port width for U; initially, LP2 is empty */
13: for(j=1 to n_m, j=j+1) do
        /* lines 14 to 21: read port information from M[j] module specified in M */
14:     LP1 = φ;       /* LP1 is a set Listing Port information for M[j] module, and each record */
                       /* of LP1 is composed of port direction, port name and port width */
15:     fp_m = open a file for M[j];      /* M[j] module written in HDL is provided in a file */
                /* in advance, and the file is opened and read based on a file pointer fp_m */
16:     while(!end of line in fp_m) do
17:         if (current line pointed by fp_m contains HDL port syntax) then
18:             LP1 = LP1 ∪ {port name, port direction and port width in current line};
19:         end if
20:         go to next line in fp_m;
21:     end while
        /* lines 22 to 33: compare between LW1 and LP1 and generate sets LWPM and LP2 */
22:     for each record of port information ∈ LP1 do
23:         flag = FALSE;
24:         for each record of wire information ∈ LW1 do
25:             if ((M[j] equals module name in current record of LW1) and (port name in current
                    record of LP1 equals port name in current record of LW1)) then
26:                 LWPM = LWPM ∪ {wire name, wire LSB and wire MSB of current record
                                   in LW1, port name of current record in LP1, and M[j]};
27:                 flag =TRUE;
28:             end if
29:         end for
                                                        /* continued in Listing 2 */
```

Listing 2 *UnitGen():* Unit Generation (continued from Listing 1)

```
                /* LP2 contains ports not currently connected to a wire (e.g., clock ports may */
                /* be wired up later) */
30:             if (flag equals FALSE) then
31:                 LP2 = LP2 ∪ current record of port information in LP1;
32:             end if
33:         end for
34: end for

    /* lines 35 to 50: write HDL code by instantiating the n_m modules and using sets */
    /* listing wires and ports */
35: for each record of port information ∈ LP2 do
36:         write ports in LP2 to U;
37: end for
38: for each record of wire information ∈ LW1 do
39:         write wires in LW1 to U;
40: end for
41: for (i=1 to n_m, i=i+1) do
42:         write the instantiation code for M[i] module to U;
43: end for
44: for (i=1 to n_m, i=i+1) do
45:         for each record of the information of wire-port mapping ∈ LWPM do
46:             if (module name in current record of LWPM equals M[i]) then
47:                 write code for wire-port mapping in LWPM to U;
48:             end if
49:         end for
50: end for
end
```

advance; the resulting port information extracted is placed in a data structure LP1. Here, each record of LP1 is composed of the following fields: port name, port direction and port width. Lines 22 to 33 of Listings 1 and 2 compare, for each module, (i) the port name of each wire contained in LW1 with (ii) each port name (corresponding to a specific port of a specific module) contained in LP1. Thus, UnitGen can decide required wire connections among the modules specified in the array of module names utilizing port information of the modules. With the comparison performed in the lines 22 to 33, UnitGen saves the wire-port mapping information for the specified modules to a linked list LWPM in line 26, where LWPM is composed of five fields: wire name, wire Least Significant Bit (LSB), wire Most Significant Bit (MSB), port name and module name. Ports with no internal connections – and thus definitely external ports for the hardware unit to be generated – are saved to a linked list LP2

in line 31 of Listing 2. Here, the fields of LP2 are the same as the ones of LP1. Finally, in lines 35 to 50 of Listing 2, UnitGen writes synthesizable Verilog HDL code by generating in a declarative fashion the instantiation code of the modules including all wires between modules. Example 6.6 shows how UnitGen generates a hardware unit in an HDL file.

Example 6.6 Unit Generation

Consider the generation of a hardware unit, BAN A of BFBA shown in Figure 6. As shown in Example 6.5, UnitGen first takes an array of module names that contains MPC755, MBI_SRAM, HS_REGS, CBI_MPC755, SRAM_A and Bi-FIFO. In lines 1 to 10 of Listing 1, UnitGen extracts specific wire data to connect between modules (e.g., one wire datum is w_name "w_addr," m1_name "SRAM_A" and m1_pname "sram_addr" in the format of Figure 22) from the Wire Library and saves the wire record (e.g., w_name "w_addr", m1_name "SRAM_A" and m1_pname "sram_addr") to LW1. In lines 14 to 21 of Listing 1, UnitGen obtains port information (e.g., port name "sram_addr") from the current module (e.g., "SRAM_A") and saves the port name to LP1. Next, in lines 22 to 33, to decide a specific wire that connects between modules, UnitGen compares, for the current module (e.g., "SRAM_A"), an associated port name (e.g., "sram_addr") field in LP1 with a port name (e.g., "sram_addr") field of LW1. If both the fields are equal, they need to be connected (by design, the Module and Wire Libraries are constructed to assign the same name to ports which can be connected), and UnitGen takes the wire information (e.g., "w_addr") in LW1, port information (e.g., "sram_addr") and current module name (e.g., "SRAM_A"), and saves them to LWPM in line 26. LWPM will be used later to generate wires in Verilog HDL. This procedure (from line 22 to line 33) is repeated for all ports in LP1. Finally,

```
module BAN_A(sysrstb, sysclk,
                    // skip some I/O
                    reb_dn, web_dn
);
// I/O definitions
input   sysrstb;
input   sysclk;
// skip some I/O definitions
output  reb_dn;
output  web_dn;

// Wire definitions
wire sysrstb;
wire [31:0] w_addr_cpu;
wire [63:0] w_dl_i;
wire [31:0] w_addr_cpu_o;
wire [20:0] w_addr;
// skip some wire definitions
wire [63:0] w_dq_sram;
wire w_reb_sram;
wire w_up_isr0_ctl_o;
wire [63:0] w_dl_o;
wire w_rv_done_csb_o;

// Wire-port mapping
MPC755 mpc755_0(              .sysrstb(sysrstb),
                              .A(w_addr_cpu),
                               // skip
                              .DL(w_dl_i));

CBI_MPC755 cbi_mpc755_0( .ADDR(w_addr_cpu),
                              .dl_i(w_dl_i),
                               // skip
                              .addr_o(w_addr_cpu_o));

SRAM_A sram_a_0(              .sram_addr(w_addr),
                              .sram_dq(w_dq_sram),
                               // skip
                              .sram_oeb(w_reb_sram));

MBI_SRAM mbi_sram_0(        .addr_local(w_addr_cpu_o),
                              .osram_addr(w_addr),
                               // skip
                              .sram_dq(w_dq_sram));

Bi-FIFO bi-fifo_0(            .sysrstb(sysrstb),
                               // skip
                              .up_isr0_ctl_o(w_up_isr0_ctl_o));

HS_REGS hs_regs_0(            .cpu_dl_i(w_dl_o),
                               // skip
                              .rv_done_csb_o(w_rv_done_csb_o));
endmodule
```

Figure 28: Top Verilog HDL Code of BAN A Generated from UnitGen

in lines 35 to 50, UnitGen generates the instantiation code for each module, including all wires, in the form of Verilog HDL code describing BAN A as shown in Figure 28.

The generated BAN A code shown in Figure 28 is composed of six modules: MPC755, CBI_MPC755, MBI_SRAM, Bi-FIFO and HS_REGS as shown in the array of module names, which is an input to UnitGen. The Verilog code is described in following order: I/O definitions, wire definitions and wire-port mappings for each instantiated hardware module. In the wire definitions, we can see the wire "w_addr," which was saved in LWPM earlier in this example. In the wire-port mapping shown in Figure 28, we also see the wire-port mappings related to the wire "w_addr," which connects between an SRAM_A port "sram_addr" and an MBI_SRAM port "osram_addr." □

6.2.4 Bus Subsystem Generation

As the second step of the flowchart in Figure 26 (shaded area), BusSynth generates required Bus Subsystems. Here, we explain the Bus Subsystem generation in which we use the algorithm BusSubSys shown in Listing 3 on page 78. Input arguments of the algorithm are an array of module names in each BAN of each Bus Subsystem, an array of BAN names in each Bus Subsystem, the number of Bus Subsystems, an array of number of modules in each BAN of each Bus Subsystem, an array of number of BANs in each Bus Subsystem, an array of parameters that specify the properties of each module, and, finally, a pointer to the Module Library. Example 6.7 shows an example of arguments in the pseudo code shown in Listing 3.

Example 6.7 Input Arguments in BusSubSys Algorithm

Consider the case where we generate BFBA Bus Subsystem in Figure 29 (repeated here for convenience from Figure 6). Please note that BFBA is called as a Bus System which is composed of a single Bus Subsystem. An array of module names for each BAN is

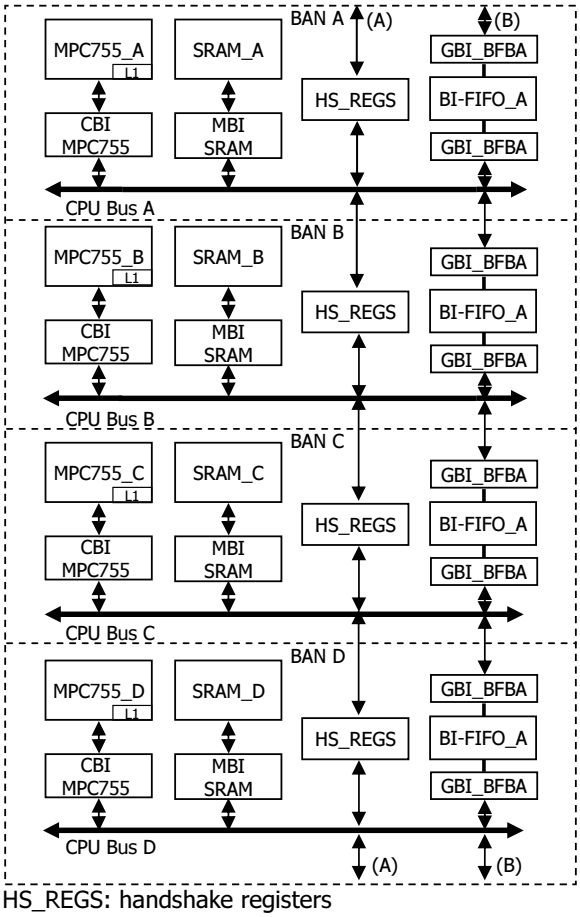


Figure 29: Diagram of BFBA (repeated here for convenience from Figure 6)

as follows: { {"MPC755_A", "CBI_MPC755", "SRAM_A", "MBI_SRAM", "HS_REGS", "BI-FIFO_A"}, {"MPC755_B", "CBI_MPC755", "SRAM_B", "MBI_SRAM", "HS_REGS", "BI-FIFO_B"}, {"MPC755_C", "CBI_MPC755", "SRAM_C", "MBI_SRAM", "HS_REGS", "BI-FIFO_C"}, {"MPC755_D", "CBI_MPC755", "SRAM_D", "MBI_SRAM", "HS_REGS", "BI-FIFO_D"} }. An array of BAN names is {"BAN A", "BAN B", "BAN C", "BAN D"}, n_{sub} is "1," an array of the number of modules is {"6", "6", "6", "6"}, an array of the number of BANs is {"4"}. To specify 32MB total of SRAM and a BI-FIFO with 1024-depth and 64-bit width, an array of parameters is { { {"20", "64"}, {"10", "64"} }, { {"20", "64"}, {"10", "64"} }, { {"20", "64"}, {"10", "64"} }, { {"20", "64"}, {"10", "64"} } }.

Here {"20", "64"} specifies the widths of address and data buses in an SRAM in each BAN, respectively, and {"10", "64"} specifies the depth and data bus width of the BI-FIFO in each BAN, respectively. □

First, modules in each BAN are extracted from the Module Library as shown in lines 4 to 14. Next, BANs in Bus Subsystem(s) are generated by calling UnitGen in line 15 of Listing 3. After this, the Bus Subsystem(s) are generated by connecting (choosing the appropriate wires for) the generated BANs via a call to UnitGen in line 17. Example 6.8 shows the generation of a sample BAN, and Example 6.9 shows how a Bus Subsystem is generated by BusSubSys shown in Listing 3.

Listing 3 *BusSubSys():* Bus Subsystem Generation.

```
Input: module_name_array MNA, ban_name_array BN, bus_subsystem_number n_sub,
       module_number_array NM, ban_number_array NB, parameter_array P,
       wire_library_file *WL, module_library_file *ML
Output: HDL code of Bus Subsystems
begin
1: for (i=1 to n_sub, i=i+1) do     /* n_sub is the number of Bus Subsystems in a Bus System */
       /* lines 2 to 16: BAN generation */
2:     n_b = NB[i];     /* n_b is the number of BANs in each Bus Subsystem */
3:     for (j=1 to n_b, j=j+1) do
           /* lines 4 to 14: look for modules specified in MNA in Module Library ML and */
           /* extract the corresponding RTL code for the module to a file */
4:         n_m = NM[i][j];     /* n_m is the number of modules in the current BAN of the */
                               /* current Bus Subsystem under consideration */
5:         for (k=1 to n_m, k=k+1) do
6:             l = 0;
7:             for each line of module MNA[i][j][k] in ML do /* to search a module in ML, */
               /* we have a look up table that specifies start and end lines for a module in ML */
8:                 if (current line contains "@parameter@") then
9:                     replace "@parameter@" in current line with P[i][j][k][l];
10:                    l=l+1;
11:                end if
12:                save the line to a file named MNA[i][j][k]
13:            end for
14:        end for
15:        Call UnitGen (&MNA[i][j], "ban_i_j", WL);     /* BAN generation */
16:    end for
17:    Call UnitGen (&BN[i], bus_subsystem_i WL);     /* Bus Subsystem generation */
18: end for
end
```

Example 6.8 BAN Generation

BusSubSys first takes arguments as shown in Example 6.7. For BAN A of BFBA shown in Figure 29, the required list of modules, which is an input (module_name_array) to BusSubSys shown in Listing 3, are as follows: MPC755, MBI_SRAM, HS_REGS, CBI_MPC755, SRAM_A and BI-FIFO (please see Example 6.7). In lines 4 to 14 in Listing 3, BusSubSys extracts four modules (MPC755, MBI_SRAM, HS_REGS and CBI_MPC755) from the Module Library, and in lines 7 to 13, the last two modules (SRAM_A and BI-FIFO) are generated with parameters in an array of parameters that is one of the input arguments. In other words, SRAM_A is generated with a 20-bit address bus width and a 64-bit data bus width, and BI-FIFO is generated with a 10-bit address bus width and a 64-bit data bus width. (Note that we assume standard tools from companies such as Synopsys [48], Artisan [2] and Virage Logic [55] are available.) Then, in line 15, BusSubSys calls UnitGen together with a pointer to Wire Library, a hardware unit name "BAN A" to be generated and an array of module names that contains MPC755, MBI_SRAM, HS_REGS, CBI_MPC755, SRAM_A and BI-FIFO. After the procedure shown in Example 6.6, UnitGen finally writes Verilog HDL code describing BAN A. □

Example 6.9 Bus Subsystem Generation

To generate BFBA Bus Subsystem shown in Figure 29 (repeated here for convenience from Figure 6; note that BFBA is also a Bus System), BusSubSys takes input arguments shown in Example 6.7. As shown in the input arguments in Example 6.7, BFBA has four BANs as specified in an array of the number of BANs, and required module names in each BAN are specified in an array of module names. Thus, in lines 2 to 16 of Listing 3, four BANs

("BAN A", "BAN B", "BAN C" and "BAN D" as shown in an array of BAN names in Example 6.7) are generated in the same fashion as shown in Example 6.8. Then, with the BAN name array having the four generated BAN names, top module name "BFBA" and a pointer to Wire Library, BusSubSys calls UnitGen in line 17 of Listing 3 to integrate the four generated BANs, and UnitGen outputs Verilog HDL code of BFBA Bus Subsystem in a manner similar to Example 6.6. □

6.2.5 Bus System Generation

As the final step of the flowchart in Figure 26, we now describe the generation of a Bus System. The generation is carried out after the generation of any necessary Bus Subsystem(s) as shown in Section 6.2.4 since the generated Bus Subsystem(s) is (are) integrated into a Bus System. Listing 4 shows the pseudo code (BusSys) for Bus System generation. First, BusSys takes three arguments: an array of Bus Subsystem names that specify Bus Subsystems in a Bus System, an array of the names of Bus Bridges (BBs) that connect the Bus Subsystems, and a pointer to the Module Library.

As shown in line 2, BusSys is performed only if a Bus System has multiple Bus Subsystems. The reason is that a Bus Subsystem becomes a Bus System if the user does not want to use Bus Bridges (BBs) anywhere in the bus architecture of the SoC. A Bus System is also formed by connecting generated Bus Subsystems through Bus Bridges (BBs). The module(s) (e.g., a BB or a FIFO) to connect multiple Bus Subsystems are extracted from the Module Library in lines 3 to 8 of Listing 4; then, in line 11, BusSys calls UnitGen to integrate the Bus Subsystems and modules to connect the Bus Subsystems. Example 6.10 shows the generation of a sample Bus System by BusSys shown in Listing 4.

Example 6.10 Bus System Generation

We generate a Bus System we call SplitBA shown in Figure 18. Following Listing 4, BusSys

Listing 4 *BusSys():* Bus System Generation.

```
Input: subsystem_name_array SS, bus_bridge_name_array MS, wire_library_file *WL,
       module_library_file *ML
Output: HDL code of a Bus System
begin
1: n_sub = size(SS);      /* n_sub is the number of Bus Subsystems in a Bus System */
2: if (n_sub > 1) then
       /* lines 3 to 8: look for MS[i] (e.g., bus bridge) in Module Library ML and */
       /* extract the corresponding RTL code */
3:     n_ms = size(MS);
4:     for (i=1 to n_ms, i=i+1) do
5:         for each line of module MS[j] in ML      /* to search a module in ML, we have */
                   /* a look up table that specifies start and end lines for a module in ML */
6:             save current line to a file named with MS[j];
7:         end for
8:     end for

9:     SSMN = φ;      /* SSMN is a character type array to save Bus Subsystem names */
                      /* and bus bridge names; initially, SSMN is empty */
10:    SSMN = {SS} ∪ {MS};
11:    Call UnitGen (&SSMN, bus_system, WL);
12:end if
end
```

first takes as input arguments the following: {"Bus Subsystem1", "Bus Subsystem2"} for the subsystem name array, {"Bus Bridge"} for the Bus Bridge (BB) name array, a pointer to Wire Library and a pointer to Module Library. Please note that the input of the Bus Subsystem name array was formed by suffixing "Bus Subsystem" with enumerated numbers (from 1 to the number of Bus Subsystems in the user options shown in Figure 27) – e.g., "Bus Subsystem1" and "Bus Subsystem2." In line 1 of Listing 4, the size of the subsystem name array is two, and thus a Bus Bridge (namely, Bus Bridge in Figure 18) is generated using Module Library in lines 2 to 8. Next, in line 11, BusSys call UnitGen together with a pointer to Wire Library, a hardware unit name "SplitBA" to be generated and an array of modules that contains Bus Subsystem1 and Bus Subsystem2. Finally, UnitGen integrates the two Bus Subsystems (namely, Bus Subsystem1 and Bus Subsystem2) and the

Bus Bridge (BB) using Wire Library and generates Verilog HDL code describing SplitBA.

□

As we have explained throughout this section, BusSynth can generate modules as well as do a syntactic translation from high-level input descriptions based on the user options to output synthesizable Verilog HDL code for a multi-processor SoC.

6.2.6 Summary

In Sections 6.1 and 6.2, we have shown our bus synthesis methodology in detail. At first, we described two libraries used in our bus synthesis tool BusSynth and the sequence of Bus System generation step by step in the methodology. Then, we have explained algorithms, UnitGen, BusSubSys and BusSys, together with detailed examples, where the algorithms are used in each generation step in the methodology. Our current tool supports only one BB type (namely, BB_GBA) and two GBI types (e.g., GBI_GBA and GBI_BFBA) in the Module Library; however, in the case of the generation of a new Bus System (e.g., a Bus System shown in Figure 19), more BB and GBI types to support the new Bus Systems could easily be added to the Module Library after being defined by hand.

6.3 Interconnect Delay Aware Bus System Generation

As feature size is scaled down to the submicron level, interconnect delay in the design of a high-speed System-on-a-Chip (SoC) becomes a major concern. This concern is especially acute for on-chip buses. In this section we describe interconnect delay aware bus system generation based on the methodology presented in Sections 6.1 and 6.2. Interconnect delay information is provided from an estimated chip layout. The delay estimates for the on-chip buses are used early in the design phase with a corresponding impact on system correctness and performance. As an example, this section shows

interconnect delay aware generation of an SoC bus system called General Global Bus Architecture (GGBA) and previously shown in Figure 11. Section 6.3.1 presents how to estimate interconnect delay in a system. Section 6.3.2 describes generation of a Memory Bus Interface module that efficiently takes into account the interconnect delay in system operation. Finally, Section 6.3.3 explains our interconnect delay aware bus system generation.

6.3.1 Interconnect Delay Estimation

The method used to estimate bus delay is to construct an estimated floorplan for the system, extract interconnect lengths from the floorplan, and model the respective global buses using circuit simulations tools.

6.3.1.1 General Global Bus Architecture Floorplan

The construction of an estimated floorplan for the GGBA is facilitated by obtaining die area estimates for four PowerPC processing elements (PEs) used in this system. This information is available from the Motorola [28] website. Another element used in the floorplan is the memory module. The area estimate for the SRAM module in the GGBA system is found using the UMC chip-sizer [53] available on the UMC website. The UMC chip-sizer tool outputs die size based on the input of either memory size or gate count. Thus, we estimate each die size of our bus components – e.g., arbiter, CPU Bus Interface (CBI) and Memory Bus Interface (MBI) – with gate counts (NAND2 gate equivalents) inputs after synthesizing the bus components using Synopsys Design Compiler [49].

An estimated floorplan of the GGBA architecture is shown in Figure 30. This floorplan was manually created with designer input, but could have been automated by a core placement tool such as MOCSYN [10]. Figure 30 illustrates the floorplan of a global bus connecting the four processing elements and a single memory element.

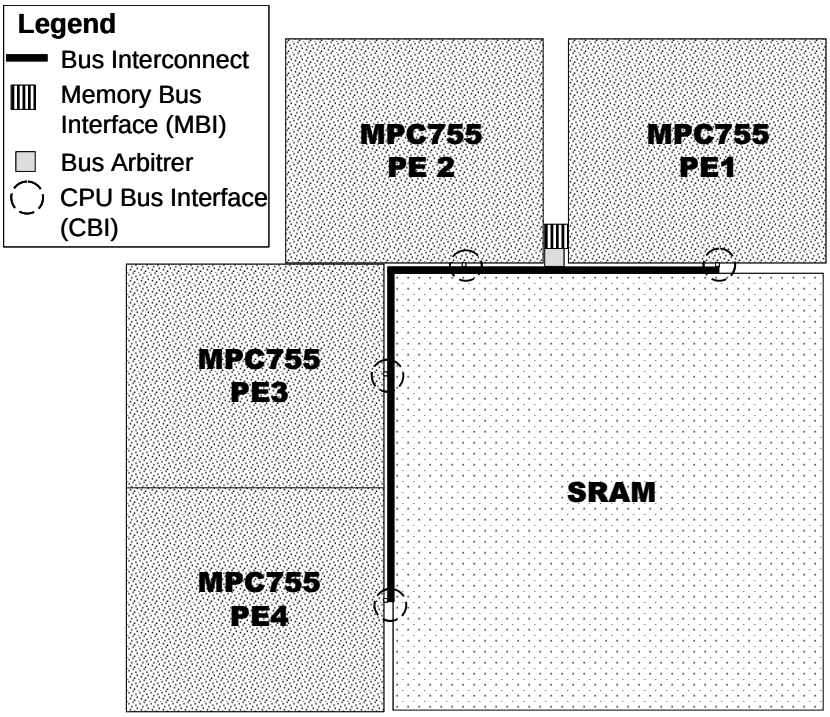


Figure 30: GGBA Estimated Layout

The GGBA floorplan was used to estimate PE-to-SRAM interconnect lengths; the results are listed in Table 1.

6.3.1.2 Bus Interconnect Physical Models

Bus interconnect physical models are designed by Alexandru Talpasanu [39], and this section presents a brief summary about the models in order to completely describe our methodology for interconnect delay aware bus generation. The details are available in a technical report [51].

Table 1: Interconnect Length Estimation for GGBA System

Processing Element (PE)	SRAM memory	
	Length [mm]	Delay [ns]
PE1	2.521	0.2848
PE2	6.143	0.5727
PE3	12.753	2.2882
PE4	19.363	3.0472

The bus interconnect shown in Figure 30 represents address and data buses connecting the four processing elements to the memory. The address and data bus widths for this GGBA system are 32 bits and 64 bits, respectively. Repeaters are not used in this design because it was found that they take up significant area while offering minuscule reductions in delay and crosstalk.

HSPICE simulations are performed on this bi-directional bus to calculate interconnect delay. The HSPICE wire models include resistance, capacitance and inductance values extracted from a MOSIS run [27] for the chosen TSMC 0.25μm technology as well as bus interconnect lengths from the GGBA system floorplan. A set of series connected RLC L-models is used to model each bus wire, with the total resistance [54], inductance and capacitance [22] [40] being derived from the total length of the bus. The interconnect length and HSPICE delay estimations between each processing element and the memory are shown in Table 1. The method used to estimate interconnect delay is automated by the use of shell-scripting and a C program [51].

6.3.2 Memory Bus Interface (MBI) Module Generation

One of the effects of interconnect delay insertion in an SoC is in the memory access cycle count of each PE. In this section, we describe an interconnect delay aware memory controller (an MBI module) for a system in its operation, and we also show automatic generation of the MBI module.

6.3.2.1 The Operation of an MBI Module

An MBI module in a system is an interface module operating as a memory controller which is located between a bus and a memory. The module generates PE control signals (e.g., *aack_bar* and *ta_bar* in PowerPC) related to memory access cycles and also generates memory control signals (e.g., *cs_bar*, *addr*, *we_bar* and *oe_bar* in PowerPC). Since moving data to or from memory in a system is affected by interconnect delay, suitable memory controller design for the system is required to account for the bus

delay so that the system operates without failure and with maximum performance. A method to ensure suitable memory control is to extend every memory access cycle according to the length of the bus interconnect delay. For example, we control two pins of the PowerPC MPC755 for the purpose of memory cycle extension: address acknowledge (*aack_bar*) and transfer acknowledge (*ta_bar*). Here, the *aack_bar* signal terminates an address bus cycle while the *ta_bar* terminates a data bus cycle. To extend each memory cycle, we delay the control signals in a memory access cycle by inserting dummy clock cycles in the memory controller. Example 6.11 shows how to extend memory access cycles in the case of PowerPC.

Example 6.11 Memory Access Cycle Extension

When four PEs (PowerPC MPC755s) sequentially access a shared memory SRAM in GGBA (shown in Figure 11) working at a 300MHz bus clock, we suppose that a burst memory access cycle (not involving interconnect delay) normally takes five bus clock cycles. In fact, a conservative approach would be to increase all memory access cycles in GGBA to include the global worst-case interconnect delay to guarantee correct operation in a real chip. As shown in Table 1, interconnect delays are 0.2848ns between PE1 and SRAM, 0.5727ns between PE2 and SRAM, 2.2882ns between PE3 and SRAM and 3.0472ns between PE4 and SRAM, and interconnect delays taken in order that memory access signals of each PE go to SRAM and return (i.e., roundtrip) are 0.5696ns between PE1 and SRAM, 1.1454ns between PE2 and SRAM, 4.5764ns between PE3 and SRAM and 6.0944ns between PE4 and SRAM. Access time of a 2MB SRAM in GGBA shown in Figure 11 is 8.0ns as estimated for 0.25μm technology by CACTI 3.0 [18], which is a cache modeling tool. Thus, total memory access delays are 8.5696ns between PE1 and SRAM, 9.1454ns between PE2 and SRAM, 12.5764ns for PE3 and SRAM and 14.0944ns for PE4 and SRAM. Since the bus clock (300MHz) period is 3.33ns, clock delays to access memory are 3 cycles for MPC755_A, 3 cycles for MPC755_B,

4 cycles for MPC755_C and 5 cycles for MPC755_D (actual calculated fractional values for the number of cycles needed are 2.5735, 2.7464, 3.7767 and 4.2326).

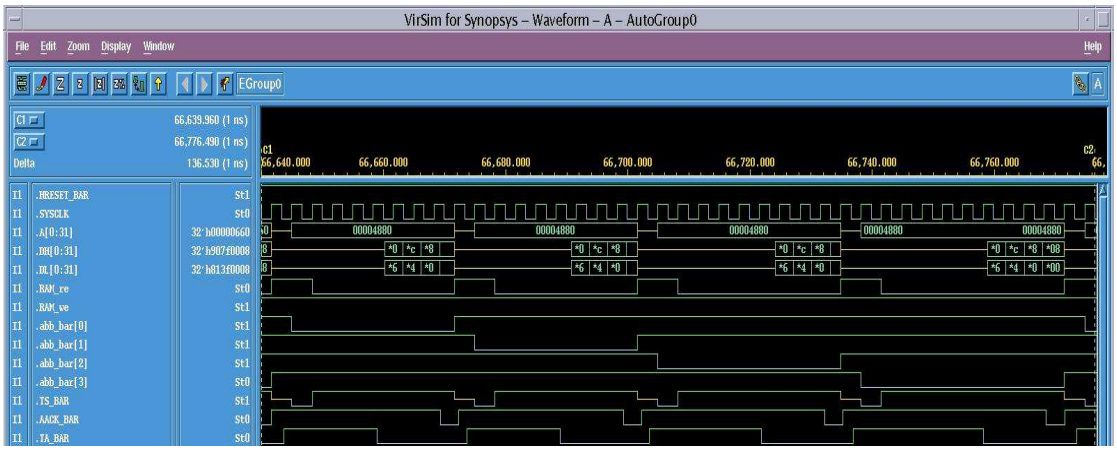


Figure 31: Waveform of Extended Memory Access Cycle

To do the cycle extension, the memory controller in the system outputs *aack_bars* and *ta_bars* to MPC755s A, B, C and D, where the signals are deferred by 3, 3, 4 and 5 clocks, respectively. Therefore, total number of cycles for the memory accesses becomes 8, 8, 9 and 10 cycles from MPC755_A to MPC755_D, respectively. Figure 31 shows these memory access cycles of MPC755s A to D from the left side of the waveform, respectively. Two signals in the bottom of Figure 31 are *aack_bar*, and *ta_bar*, which are PE control signals in the memory access. □

6.3.2.2 The Generation of an MBI Module

Before generating the MBI module, with regard to the estimated interconnect delays shown in Section 6.3.1, we calculate total delay including the time taken to move controls and data on the bus in two directions (e.g., from a PE to a memory and vice versa) and the time taken to access memory in a read operation. However, noticing that transmitting the signals for controls and data on the bus to a shared memory

has the same direction to the memory in a write operation, we only show here the effect of bus interconnect delay in a read operation since the read operation always requires a roundtrip (send address then receive data) thus typically requiring as much or more time than a write operation.

Table 2: Estimated Total Delay of Paths between Each PE and a Shared Memory

PE	Estimated bus delay between a PE and SRAM [ns]	Delay in a read operation (bus roundtrip) [ns]	SRAM (2Mbyte) access time [ns]	Total delay in a read operation [ns]
PE 1	0.2848	0.5696	8.00	8.5696
PE 2	0.5727	1.1454	8.00	9.1454
PE 3	2.2882	4.5764	8.00	12.5764
PE 4	3.0472	6.0944	8.00	14.0944

Note: The access time of a shared SRAM (2Mbytes) is estimated by CACTI 3.0 [18]

Table 2 shows estimated delays for the GGBA estimated layout shown in Figure 30. The second column shows estimated interconnect delays described in Section 6.3.1, and the third column shows bidirectional delays for a read operation. The fourth column shows memory access time for a 2MB SRAM, where the access time is estimated by using CACTI 3.0 [18], which is an integrated cache access time, cycle time, area, aspect ratio and power model. Finally, the fifth column is the summation of the previous two columns, that is, total delay for in a read operation.

Table 3 shows the number of clock delay cycles that will be inserted into a memory cycle for the cases that a GGBA system has three different bus clocks, respectively. The total delays shown in Table 2 are divided by each bus clock period in order to obtain the number of clock delays shown in Table 3.

Figure 32 describes the sequence of MBI module generation, which is a module generation procedure of our bus synthesis tool (BusSynth) that will be described in Section 6.3.3. With the input of interconnect delay shown in Table 1, the number of clock cycles required to be inserted for a memory access cycle is calculated in the

Table 3: Number of Clock Delays in Data Paths

PE	Number of clock delays in each PE for a read operation [clock]		
	100 MHz (10.00ns) system clock	200 MHz (5.00ns) system clock	300 MHz (3.33ns) system clock
PE 1	1 (0.8570)	2 (1.7139)	3 (2.57345)
PE 2	1 (0.9145)	2 (1.8291)	3 (2.74636)
PE 3	2 (1.2576)	3 (2.5153)	4 (3.77669)
PE 4	2 (1.4094)	3 (2.8189)	5 (4.23255)

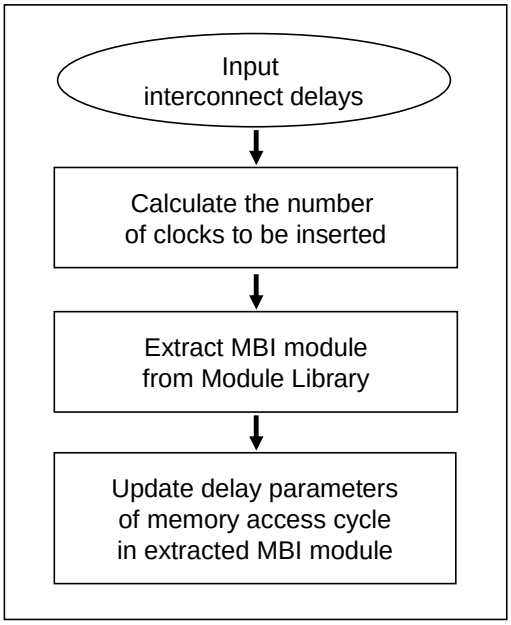


Figure 32: Sequence of MBI Module Generation

second step. Then, based on the user input options (please, see Section 6.2.2) that configure an SoC bus system with a shared memory, an MBI module is extracted from a Module Library that contains the respective module as a library component (see Section 6.1). The module is described in Verilog HDL and has pre-defined delay parameters which model corresponding clock delays and memory access cycles. Finally, the delay parameters are updated with the number of clocks calculated as described earlier and shown in Table 3.

```
module mbi_sram(hrst_bar, abb_bar, cs_bar, sram_web,
                        // Skip I/Os
                        sram_oeb, sram_addr, sram_dq);

// Parameter definitions
parameter MEM_A_WIDTH = 20;
parameter MEM_D_WIDTH = 64;
parameter DLY_PE1 = 4'h3;
parameter DLY_PE2 = 4'h3;
parameter DLY_PE3 = 4'h4;
parameter DLY_PE4 = 4'h5;

// I/O definitions
input HRST_BAR;
input [0:3] ABB_BAR;
input [0:7] CS_BAR;
output sram_web;
output sram_oeb;
output [MEM_A_WIDTH-1:0]  sram_addr;
inout   [MEM_D_WIDTH-1:0]  sram_dq;
// Skip I/O definitions

// Register definitions
reg [0:3]    RnumRdDelay;
// Skip register definitions
// Assign delay values
  always @(cs_bar or hrst_bar)
    begin
      if (~hrst_bar)
              RnumRdDelay <= 4'h0;
      else if (~cs_bar)
            if(~abb_bar[0])
                RnumRdDelay <= DLY_PE1;
        else if (~abb_bar[1])
                RnumRdDelay <= DLY_PE2;
        else if (~abb_bar[2])
                RnumRdDelay <= DLY_PE3;
        else if (~abb_bar[3])
                RnumRdDelay <= DLY_PE4;
      else
             RnumRdDelay <= 4'h0;
    end

  // Skip verilog description

endmodule
```

Figure 33: MBI Module with Updated Delay Clock Parameters

Example 6.12 Interconnect Delay Aware MBI Module Generation

MBI_SRAM in GGBA shown in Figure 11 interfaces between SRAM and a Global Bus Architecture (GBA). As shown in Example 6.1, The MBI_SRAM module has interconnect delay parameters that correspond to delay clocks to be inserted to memory access cycles in the GGBA. We generate the MBI_SRAM module with updated parameters of delay clock, based on the sequence of the MBI module generation shown in Figure 32. As shown in Example 6.11, the first two steps shown in Figure 32 calculate clock delays to access memory:

3 cycles for MPC755_A, 3 cycles for MPC755_B, 4 cycles for MPC755_C and 5 cycles for MPC755_D. Then, based on the user option that selects SRAM, MBI_SRAM is extracted from the Module Library in step 3, where the MBI module has clock delay parameters that correspond to clock delay to be inserted into memory access cycles of each PE. Finally, in step 4, the parameters are replaced with the delay clock (namely, 3, 3, 4 and 5 cycles) calculated in step 2 in our tool. Figure 33 shows the generated the MBI_SRAM module with the updated parameters (e.g., "DLY_PE1 = 4'h3" in delay parameter definitions). □

6.3.3 Interconnect Delay Aware Bus System Generation

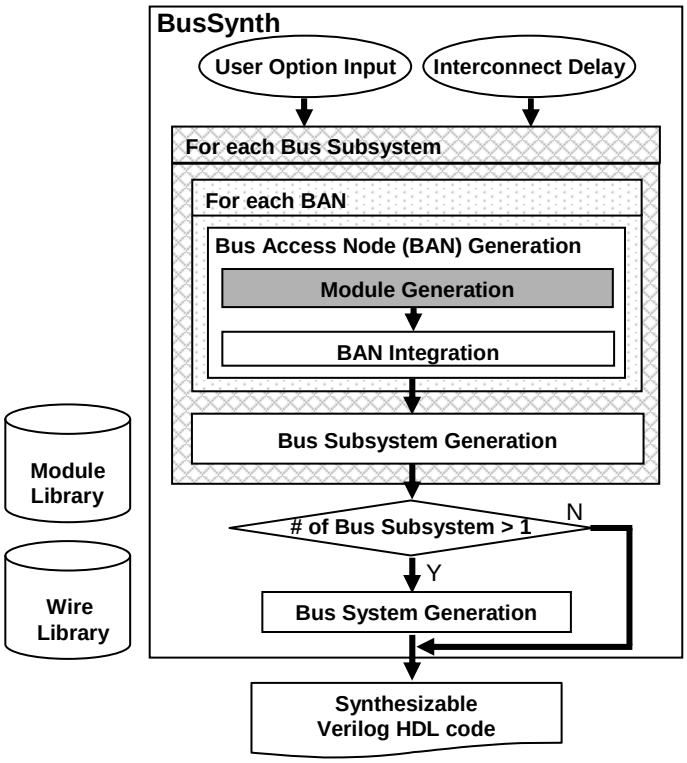


Figure 34: Sequence of an Interconnect Delay Aware Bus System Generation

The flowchart in Figure 34 shows the sequence for interconnect delay aware Bus System generation in our bus synthesis tool (BusSynth). First, BusSynth takes user input options and estimated interconnect delay for each PE to memory for a Bus

System to be generated (please note that we only implemented the case for a single memory and multiple PEs together in a single Bus Subsystem; however, we could extended our tool to support multiple memories in the interconnect delay aware Bus System generation). Based on these inputs, BusSynth generates the required Bus Access Nodes (BANs) after generating required modules for the BANs. The MBI module described in Section 6.3.2 is generated in the stage of the required module generation based on the user options. BusSynth subsequently assembles the BANs into required Bus Subsystems, each of which consists of one or more BANs connected together using bus components. After that, if the Bus System the user wants has more than one Bus Subsystem, the generated Bus Subsystems are integrated into a resulting Bus System. Otherwise, the generated single Bus Subsystem becomes a Bus System. Finally, BusSynth writes synthesizable Verilog HDL code for the generated Bus System.

6.4 Computational Complexity of Bus System Generation Algorithm

Now, we consider the computational complexity of our bus synthesis algorithm, which shows how it scales with increasing numbers of Bus Subsystems, BANs, modules, ports and wires. The BusSynth algorithm is shown in Figure 26 and consists of calls to UnitGen for BAN generation, BusSubSys for Bus Subsystem generation and BusSys for Bus System generation. We define several variables in Table 4 that are related to the computational complexity, and Table 5 shows an example of the numbers shown in Table 4 in our case. The values in Table 5 are based on our current two libraries (namely, Wire Library and Module Library) that support the generation of Bus Systems described in Sections 4.1 and 5.1; the values can be changed if the libraries are updated to support more varieties of bus components.

Table 4: The Numbers Related to Computational Complexity

Items	Number	Items	Max. Number
Bus Subsystems in a Bus System	n_{sub}	Ports of a module in a BAN	n_{pb}
Bus bridges (BBs) that connect Bus Subsystems in a Bus System	n_{ms}	HDL lines of a BAN	n_{lb1}
Wires that connect Bus Subsystems and BBs in a Bus System	n_{ws}	Lines of a library component in Module Library file	n_{lml1}
Ports of a Bus Subsystem in a Bus System	n_{ps}	Characters of a line of Module Library file	n_{clml}
Maximum number of BANs in any Bus Subsystem	n_b	Lines in Wire Library file	n_{lwl}
Wires that connect BANs in a Bus Subsystem	n_{wsub}	Characters of a parameter for a module	n_{cpml}
Ports of a BAN in a Bus Subsystem	n_{psub}	Characters of a line of Wire Library file	n_{clwl}
HDL lines of a Bus Subsystem	n_{lss1}	Characters of a unit name	n_{cun}
Modules in a BAN	n_m	Characters of HDL port keyword	n_{cport}
Wires that connect modules in a BAN	n_{wb}	Character of port name in a module	n_{cportn}

We first consider the computational complexity of the UnitGen algorithm shown in Listings 1 and 2 for each case of BAN, Bus Subsystem and Bus System generation. In the case of BAN generation, the upper bounds of each routine in the algorithm are shown in Table 6, and the complexity of the algorithm will be the worst case of cases 3 and 9 of Table 6 since both cases are performed sequentially. Therefore, as shown case 10 in Table 6, UnitGen has $O(\max[(n_{lwl} * n_m * n_{clwl} * n_{cun}), n_m *$

Table 5: Example of the Numbers in Table 4

Variable	Value	Variable	Value	Variable	Value	Variable	Value
n_{sub}	2	n_{wsub}	33	n_{pb}	41	n_{cpml}	4
n_{ms}	1	n_{psub}	46	n_{lb1}	399	n_{clwl}	81
n_{ws}	79	n_{lss1}	361	n_{lml1}	364	n_{cun}	32
n_{ps}	29	n_m	7	n_{clml}	131	n_{cport}	6
n_b	5	n_{wb}	88	n_{lwl}	86	n_{cportn}	16

Table 6: The Upper Bounds of UnitGen Algorithm in the Case of BAN Generation

Case	Line	Upper Bound	Case	Line	Upper Bound
1	5 to 6	$O(n_{clwl}*n_{cun})$	8	16 to 33	O(max[case 5, case 7])
2	4 to8	$O(n_m*n_{clwl}*n_{cun})$	9	11 to 34	$O(n_m*$max[case 5, case 7])
3	1 to 10	$O(n_{lwl}*n_m*n_{clwl}*n_{cun})$	10	1 to 34	O(max[case 3, case 9])
4	17 to 18	$O(n_{clml}*n_{cport})$	11	35 to 37	$O(n_{pb})$
5	16 to 21	$O(n_{lml1}*n_{clml}*n_{cport})$	12	38 to 40	$O(n_{wb})$
6	24 to 29	$O(n_{wb}*n_{cportn}^2)$	13	41 to 43	$O(n_m)$
7	22 to 33	$O(n_{wb}*n_{pb})$	14	44 to 50	$O(n_m*n_{pb}*n_{cun}^2)$

$\max[(n_{lml1} * n_{clml} * n_{cport}), (n_{wb} * n_{pb})]]$) in computational complexity in the case of BAN generation. Similarly, UnitGen has $O(\max[(n_{lwl} * n_b * n_{clwl} * n_{cun}), n_b * \max[(n_{lb1} * n_{clml} * n_{cport}), (n_{wsub} * n_{psub})]])$ and $O(\max[(n_{lwl} * (n_{sub}+n_{ms}) * n_{clwl} * n_{cun}), (n_{sub}+n_{ms}) * \max[(n_{lss1} * n_{clml} * n_{cport}), (n_{ws} * n_{ps})]])$ in the case of Bus Subsystem and Bus System generation, respectively.

We now consider the computational complexity of BusSynth where BusSubSys (Listing 3) and BusSys (Listing 4) are executed sequentially as shown in the flowchart of Figure 26. Table 7 shows the upper bounds of each routine in the BusSubSys algorithm shown in Listing 3. Case 9 of the table shows the upper bound of the algorithm; that is, the computational complexity is $O(n_{sub} * \max[(n_b * \max[(n_m * n_{lml1} * n_{clml} * n_{cpml}), (\max[(n_{lwl} * n_m * n_{clwl} * n_{cun}), n_m * \max[(n_{lml1} * n_{clml} * n_{cport}), (n_{wb} * n_{pb})]])]), (\max[(n_{lwl} * n_b * n_{clwl} * n_{cun}), n_b * \max[(n_{lb1} * n_{clml} * n_{cport}), (n_{wsub} * n_{psub})]])])$.

The upper bounds of each routine in the BusSys algorithm shown in Listing 4 are shown in Table 8. The upper bound of the BusSys algorithm is Case 4 in Table 8; that is, the computational complexity is $O(\max[(n_{ms} * n_{lml1}), (\max[(n_{lwl} * (n_{sub}+n_{ms}) * n_{clwl} * n_{cun}), (n_{sub}+n_{ms}) * \max[(n_{lss1} * n_{clml} * n_{cport}), (n_{ws} * n_{ps})]])])$.

As we discussed before, Case 9 of Table 7 and Case 4 of Table 8 show the upper bounds of BusSubSys and BusSys algorithms, respectively. Therefore, since those algorithms are executed sequentially, the overall complexity of BusSynth is

Table 7: The Upper Bounds of BusSubSys Algorithm

Case	Line	Upper Bound	Case	Line	Upper Bound
1	9	$O(n_{clml}*n_{cpml})$	6	3 to 16	$O(n_b*max[case3, case 4])$
2	7 to 13	$O(n_{lml1}*n_{clml}*n_{cpml})$	7	17	$O(max[(n_{lwl}*n_b*n_{clwl}*n_{cun}), n_b*max[(n_{lb1}*n_{clml}* n_{cport}), (n_{wsub}*n_{psub})]])$
3	5 to14	$O(n_m* n_{lml1}*n_{clml}*n_{cpml})$	8	3 to 17	O(max[case 6, case 7])
4	15	$O(max[(n_{lwl}*n_m*n_{clwl}* n_{cun}), n_m*max[(n_{lml1}* n_{clml}* n_{cport}), (n_{wb}*n_{pb})]])$	9	1 to 18	$O(n_{sub}*max[case 6, case 7])$
5	5 to 15	O(max[case 3, case 4])			

Table 8: The Upper Bounds of BusSys Algorithm

Case	Line	Upper Bound	Case	Line	Upper Bound
1	5 to 7	$O(n_{lml1})$	3	11	$O(max[(n_{lwl}*(n_{sub}+n_{ms})* n_{clwl}*n_{cun}), (n_{sub}+n_{ms})* max[(n_{lss1}*n_{clml}*n_{cport}), (n_{ws}*n_{ps})]])$
2	4 to 8	$O(n_{ms}*n_{lml1})$	4	1 to 12	O(max[case 2, case 3])

O(max[Case 9 in Table 7, Case 4 in Table 8]). That is to say, the computational complexity of the BusSynth algorithm is O(max[(n_{sub} * max[(n_b * max[(n_m * n_{lml1} * n_{clml} * n_{cpml}), (max[(n_{lwl} * n_m * n_{clwl} * n_{cun}), n_m * max[(n_{lml1} * n_{clml} * n_{cport}), (n_{wb} * n_{pb})]])]), (max[(n_{lwl} * n_b * n_{clwl} * n_{cun}), n_b * max[(n_{lb1} * n_{clml} * n_{cport}), (n_{wsub} * n_{psub})]])]), (max[(n_{ms} * n_{lml1}), (max[(n_{lwl} * (n_{sub}+n_{ms}) * n_{clwl} * n_{cun}), (n_{sub}+n_{ms}) * max[(n_{lss1} * n_{clml} * n_{cport}), (n_{ws} * n_{ps})]])])]). Here, the computational complexity seems to be quite complex and high. However, please note that the numbers specified in the variables above are highly constrained in realistic problems as shown in Table 5. For that reason, as shown in Table 14 of Section 7.3, BusSynth takes only a second or less to generate our examples in our experimental environment shown in Section 7.2.

Example 6.13 Upper Bound of BusSynth algorithm

We calculate computational complexity of BusSynth algorithm described Listings 1, 2, 3 and 4 in Sections 6.2.3, 6.2.4 and 6.2.5. Based on the real values of our case shown in Table 5, an upper computational bound of BusSynth algorithm, which is the worst-case complexity, is described as follows: max[(2 * max[(5 * max[(7 * 364 * 131 * 4), (max[(86 * 7 * 81 * 32), 7 * max[(364 * 131 * 6), (88 * 41)]])]), (max[(86 * 5 * 81 * 32), 5 * max[(399 * 131 * 6), (33 * 46)]])]), (max[(1 * 364), (max[(86 * (2 + 1) * 81 * 32), (2 + 1) * max[(361 * 131 * 6), (79 * 29)]])])]. Then, we can rewrite the upper bound: max[(2 * max[(5 * max[1335152, (max[1560384, 7 * max[286104, 3608]])]), (max[1114560, 5 * max[313614, 1518]])]), (max[364, (max[668736, 3 * max[283746, 2291]])])]. After the calculation, the upper bound of the running time in BusSynth in our case is 20,027,280 [operations] (please note that we assume straight-line code in which each line uses an "operation" without any method call). □

The main point of note is that while our algorithms have nontrivial polynomial time complexities, our algorithms are applied to situations with integers in the ten to one thousand range (as opposed to billions or more). For example, in implementing our practical case described in Section 7, we found that the number of "legal" wires is 686 for a Wire Library with 35 modules, 445 for a Wire Library with 23 modules and 369 for a Wire Library with 17 modules. While all possible wires between all modules, including all "legal" and "illegal" combinations, would clearly scale exponentially as the number of modules increases, as we can see the actual numbers of "legal" wires and modules scale somewhat linearly with each other. Thus, we posit that in most practical cases, the number of required "legal" wires scales in such a way that the

described algorithms of this section complete in seconds or less, as shown in all cases in Table 14 of Section 7.3.

6.5 Summary

In this chapter, we have explained our methodology to generate Bus Systems in detail. For the explanation of the methodology, we have described two libraries used in BusSynth and have shown our algorithm for BusSynth step by step. Then, we have shown our method of interconnect delay aware bus generation and have described the computational complexity of the algorithm. In the following chapter, we will evaluate the generated Bus Systems with three user applications and will show the evaluation results.

CHAPTER VII

EXPERIMENTS AND RESULTS

7.1 Application Examples

Five kinds of bus architectures for a multi-processor SoC were generated using BusSynth and then simulated to evaluate performance with three applications: an Orthogonal Frequency Division Multiplexing (OFDM) transmitter [21], which is used in wireless communications; an MPEG2 decoder [29] [37]; and a database example [31].

7.1.1 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 subcarriers is arranged such that they become orthogonal, and a Fast Fourier Transform (FFT) is used for digital modulation/demodulation of each subchannel. Figure 35 shows a simplified block diagram of an OFDM transmitter. First, the subchannels 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 36.

Figure 36 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

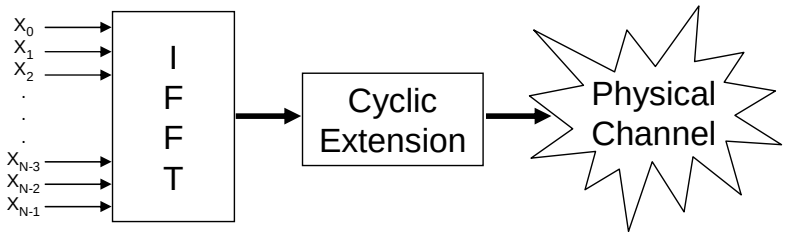


Figure 35: The Block Diagram of an OFDM Transmitter

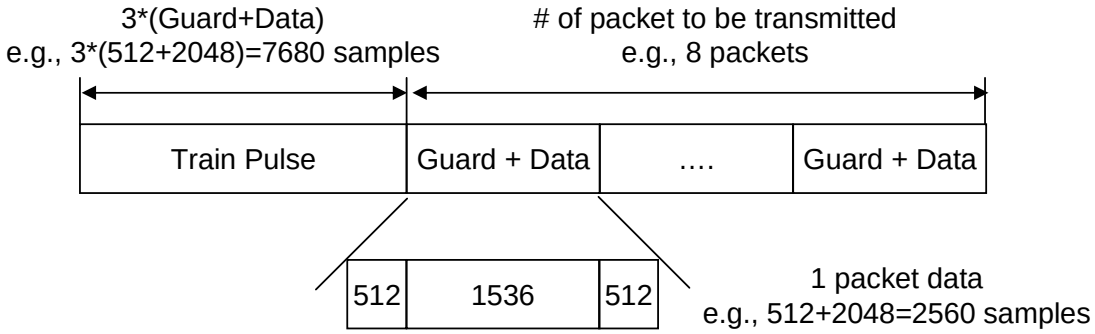


Figure 36: OFDM Data Format

valued sample and a 512-complex valued guard signal, where the size of guard data is usually a quarter of the data block. Figure 37 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 37 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.

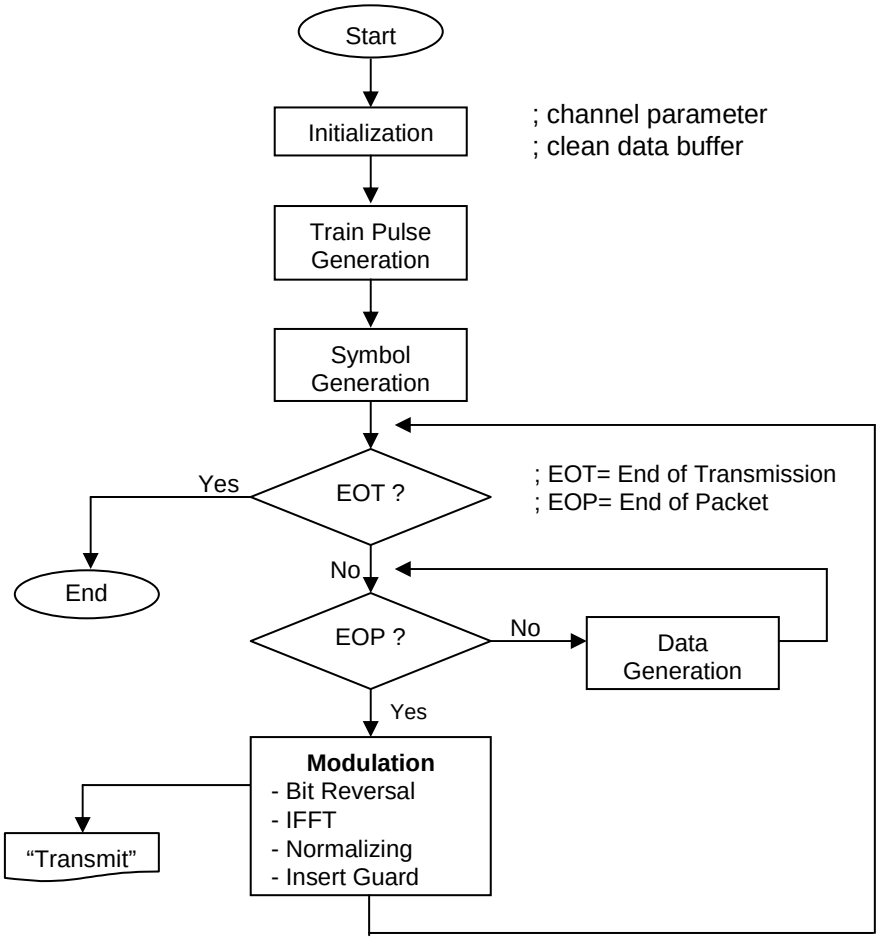


Figure 37: The Flowchart of the OFDM Transmitter

In the OFDM transmitter, 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 9 shows a list of functions in the OFDM transmitter and outlines the function assignment in each BAN. The functions assigned to BAN A may seem to require a lot of computation, but in fact BAN A is not the bottleneck of system performance because the first three functions listed for BAN A (italicized in Table 9) are executed only once. Only data generation, symbol mapping and bit reversal functions are iterated in BAN A. The function in BAN B, IFFT, unfortunately is difficult to split up due to the structure of the IFFT.

Table 9: 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 38 describes the computation performed by each PE according to programming styles: Pipelined Parallel Algorithm (PPA) and Functional Parallel Algorithm (FPA). Here E, F, G and H in Figure 38 indicate function groups shown in Table 9. We programmed the OFDM transmitter algorithm in both PPA style and FPA style to see how the software programming 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 2048 complex valued samples and 512 complex valued guard signals.

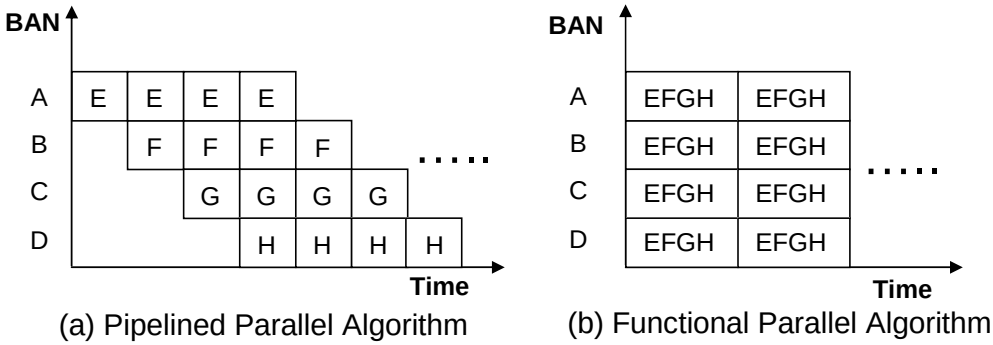


Figure 38: Software Programming Style in OFDM

7.1.2 MPEG2 Decoder

MPEG2 video is an ISO/IEC standard that specifies the syntax and the semantics of encoded video bit streams [37]. The data streams include parameters such as bit rates, picture sizes and resolutions. We modified an MPEG2 decoder program from the MPEG Software Simulation Group [29], resulting in C code having 8788 lines, in order to evaluate the generated Bus Systems.

Figure 39(a) shows input video frames, and Figure 39(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 39(a), and a Group Of Pictures (GOP) is composed of two frames (I and P). In our simulation, each frame size is specified with a very small picture, 16 pixels by 16 pixels, because of the limitation of simulation speed.

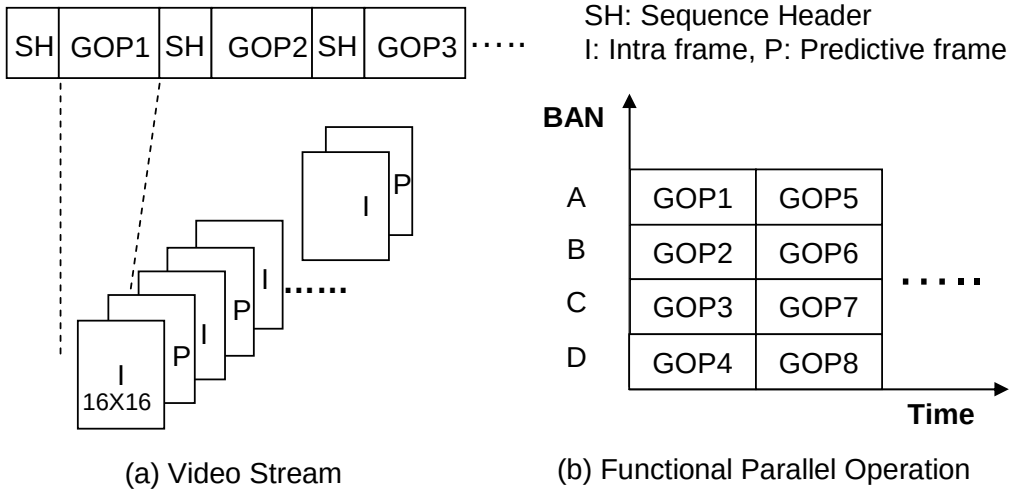


Figure 39: Input Video Stream and Functional Parallel Operation

For the MPEG2 decoder, we exclusively used the FPA style because it yielded significantly faster results. As shown in Figure 39(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 (all of our SoC examples have

four BANs) perform not only MPEG2 decoding but also raw data input and decoded data output, respectively.

7.1.3 Database Example

As for the last 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 40, a database system may use several transactions to access objects in the other tasks. For example, in Figure 40, 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 multi-processor system.

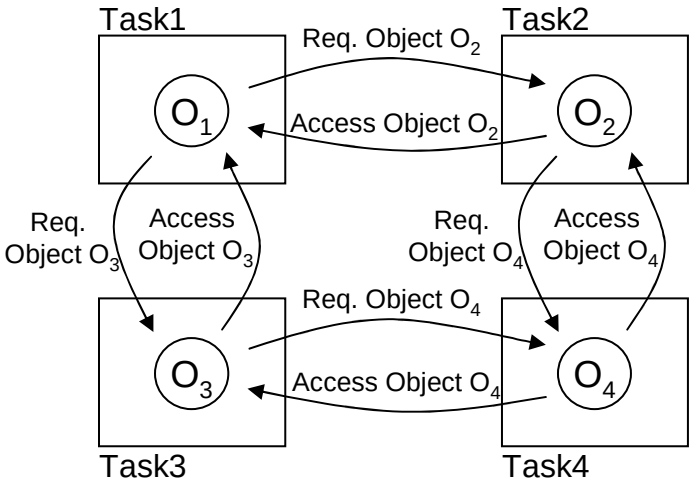


Figure 40: Transactions in Database Example

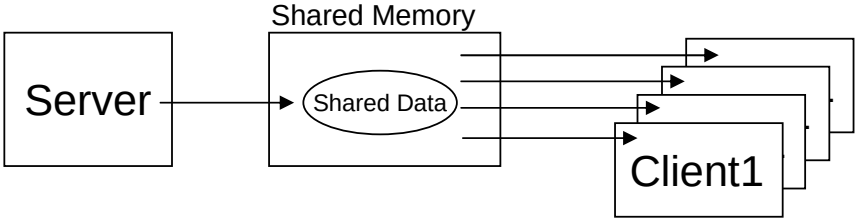


Figure 41: Data Transfer from a Server to Clients

As for the Real-Time Operating System (RTOS), we use Atalanta RTOS Version 0.4 developed at Georgia Tech [9]. We have installed the RTOS on each BAN, and tasks assigned to each BAN are executed on top of the RTOS. We simulated this database example in a variety of Bus Systems each with four PEs. A total of forty-one tasks that form a subset of the database example run: eleven on the PE in BAN A and ten on each PE in each of other BANs in the examples of Bus Systems shown in Sections 4.1 and 5.1. As shown in Figure 41, 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 writes (reads) one-hundred 32-bit words to (from) the shared memory. With this database example, each Bus System 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 the Bus Systems.

7.2 Experimental Setup

As shown in Figure 42, BusSynth takes the user input as described in Section 6.2 and interconnect delay described in Section 6.3, 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 [24] together with VCS, a Verilog HDL simulator from Synopsys [50]. In order to synthesize the Verilog HDL code to logic gates, we use the Synopsys Design Compiler. For this environment, we use a Sun workstation Ultra 60 having two 450MHz UltraSPARC II processors and 2GB of memory.

In this experiment, we set up four MPC755s in Seamless CVE; each BAN has one MPC755 with up to 300MHz external clock, *sysclk*. The maximum frequency of *sysclk* dictates the maximum bus speed (note that the internal MPC755 clock speed

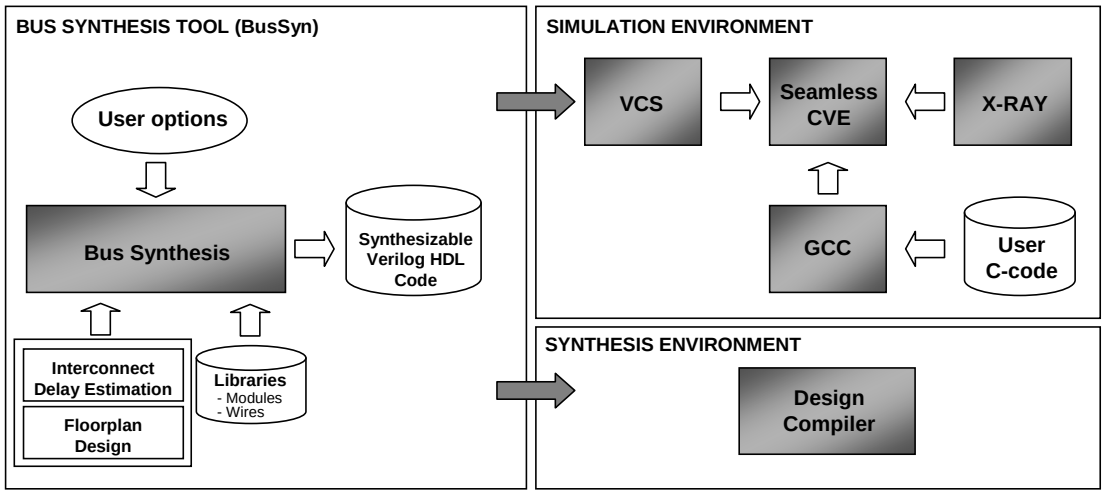


Figure 42: Experimental Environment

can be much faster, e.g., 500MHz) [28]. However, our results are equally applicable to much faster bus clock speeds. Note that the MPC755 Instruction Set Simulator (ISS) provided by Seamless CVE is instruction accurate, not cycle-accurate; nonetheless, external (non-cache) memory accesses are cycle accurate. In short, we have a bus functional simulation setup with cycle accuracy for all bus transactions.

7.3 Comparison of Results

Generated Bus Systems are evaluated by their performance in the context of the three applications described in Section 7.1 and in the experimental environment shown in Section 7.2. First, we compare performance among the generated Bus Systems. Next, we present the impact of interconnect delay prediction in the design phase by comparing the performance of Bus Systems generated without versus with each PE to memory delay customized to the exact interconnect delay of the particular distance. Finally, we show generation time of each Bus System from BusSynth and the gate counts of the generated Bus Systems.

7.3.1 Performance Comparison among Bus Systems

With the generated Bus Systems (shown in Figures 6, 7, 8, 17 and 18) and hand-designed examples of CCBA and GGBA (shown in Figures 10 and 11), we evaluate the performance and verify the operation of each Bus System with an OFDM transmitter, an MPEG2 decoder and a database example. Please note that many partitions of tasks to PEs were tried; we report only the best results obtained (i.e., the best possible partition found by hand for the given bus architecture). The Bus Systems except GBAVIII and HybridBA have 32MB total of non-L1 cache memory, respectively, and GBAVIII and HybridBA have 40MB total of non-L1 cache memory; however, since all the application code including instruction and data fits in 32MB memory size, the memory size increase to 40MB has no or tiny effect in our application performance. Each PE (MPC755) embedded in each Bus System has 32KB of L1 I-cache and 32KB of L1 D-cache.

Table 10: 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	HybridBA	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

Table 10 shows the results of our evaluation using an OFDM transmitter that in our example has 922 lines of C code for the algorithm implementation (7,909 lines of assembly code for the algorithm implementation) and 696 lines of assembly code

for PE 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 (BB), and the two Bus Subsystems operate independently. Therefore, in SplitBA, it is more reasonable to use the FPA style. SplitBA (Case 7 in Table 10) using the FPA style shows the best performance among the Bus Systems in our example: OFDM transmission reaches a rate of 5.1132Mbps, 16.44% faster than GGBA, which we take as representative of a typical commercial bus. We can see in Table 10 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 outperforms PPA in the OFDM transmitter application (e.g., Case 3 vs. 4, Case 5 vs. 6 and Case 8 vs. 9 in Table 10). 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 do Bus Systems having separate local memories for program and local data for each BAN (e.g., GBAVIII). 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 split its bus architecture into two Bus Subsystems, and thus each arbiter in each Bus Subsystem deals with only half the number of total memory requests of the application. With this reason, SplitBA outperforms GGBA in our example (Case 7 vs. 8).

(d) A fast data transfer method between BANs such as Bi-FIFOs used in BFBA and Bi-FIFOs used in HybridBA contributes to the performance improvement

observed for the PPA style (e.g., Case 1 = Case 6 > Case 4 > Case 9 > Case 2, in throughput).

Table 11: Evaluation Results in MPEG2 Decoder

Case	Bus System	Application Throughput [Mbps]	Software Programming Style
10	BFBA	0.8594	FPA
11	GBAVI	0.8271	FPA
12	GBAVIII	1.1444	FPA
13	HybridBA	1.1650	FPA
14	CCBA	1.0083	FPA

Note: Picture size: 16 x 16

Our MPEG2 decoder application has 8,788 lines of C code for its algorithm (26,430 lines of assembly code for the MPEG2 decoder algorithm) and 697 lines of assembly code for initialization routines and APIs. Due to the requirement of significant global memory interaction due to a large number of global variables in our MPEG decoder program, we could only use FPA effectively; thus, Table 11 reports results only for the FPA software programming style. In the results shown in Table 11, HybridBA (Case 13) shows the best performance because HybridBA 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 to global memory from all BANs. The results also show that HybridBA and GBAVIII outperform CCBA due to faster arbitration time in data read operations. In Table 11, 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 HybridBA, generated by BusSynth, outperforms CCBA by 15.54% in this example.

In the database application example, for multi-threaded operation, we employ the Atalanta RTOS [9], which requires a shared memory. We can support the use of the RTOS in GBAVI and BFBA; however, in this dissertation, we do not simulate these Bus Systems with this application because the current versions of these Bus

Table 12: Evaluation Results in a Database Example

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

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 a shared memory

Systems do not have such a shared memory. Furthermore, the database application is an example using only a shared memory without using local memories for data transactions between the server and the clients. Therefore, when we assume that, in this example, we do not use Bi-FIFO block(s) nor local memories, Bus Systems having a global memory and single global bus (e.g., GBAVIII, HybridBA and GGBA) have almost exactly (within 0.1%) the same performance in this example due to the same bus components. For that reason, we use one of these Bus Systems, GGBA (see Figure 11), as a baseline of performance comparison and compare the performance only with SplitBA (see Figure 18) in this application.

This example has total of 1700 lines of C code for the database algorithm (14,597 lines of assembly code for the database 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 18 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 12, SplitBA (Case 16 in Table 12) outperforms GGBA (Case 15 in Table 12) with a 41% reduction in application execution time. The performance of SplitBA is improved over GGBA because of following two reasons. The first one is that SplitBA has a better bus topology (e.g., split global bus connected by a BB) than GGBA, and thus bus traffic due to the shared memory requests is lessened. The second one is that SplitBA has a Global Bus Architecture (GBA) in

each Bus Subsystem so that all clients can easily access object data from the server. Please note that even though two of the clients (BANs D and E shown in Figure 18) in SplitBA are a BB away from the server (BAN A shown in Figure 18), apparently the two advantages listed above more than compensate.

7.3.2 Performance Comparison in Interconnect Delay Aware Bus Systems

To demonstrate possible impact of interconnect delay prediction in the design phase, we show three different configurations of a GGBA system: GGBA I, GGBA II and GGBA III. These three configurations have the same bus architecture, which is shown in Figure 11; nevertheless, the configurations vary in that each have different memory controllers. The first GGBA system, GGBA I, has a memory controller working with no regard to interconnect delay on the bus between each PE and the shared memory (thus, GGBA I may fail if implemented in a real SoC; nonetheless, GGBA I represents a typical initial simulation with communication across wires occurring instantaneously). The second GGBA system, GGBA II, is generated by BusSynth based on the methodology introduced in Section 6.3 and has a memory controller that works with different estimated interconnect delays on the shared bus. Here, the delays are provided from an estimated chip layout as introduced in Section 6.3.1, and the delay values are shown in Table 2. Finally, the third system, GGBA III, has a memory controller that operates with a maximum estimated delay on all connections between the PEs and the shared memory. In light of memory access, the third system is a non-optimized system that can be designed if we only use worst-case interconnect delay information in the design phase.

Table 13 shows execution times for an OFDM packet in GGBAs I, II and III, and their percentage comparison. Here, an OFDM packet consists of 128 real and imaginary data samples and 32 guard data samples. Note that in Table 13 simulations are performed with the bus clocked at 300MHz, 200MHz and 100MHz. For GGBA II

and III, both of which account for interconnect delay, the memory controller waits for an appropriate number of bus cycles based on the required delay and the bus cycle time; e.g., a 10ns delay requires only one bus cycle at 100MHz but requires three bus cycles at 300MHz.

7.3.2.1 Comparison I

In Comparison I of Table 13, GGBA I is used as a baseline for performance degradation according to altering the memory controller (MBI) to account for interconnect delay. In the case of (a) 300MHz bus clock in Table 13, the execution time shown in Comparison I increases up to 161.0% in GGBA III against the result of GGBA I. This increase is due to the fact that GGBA III uses overall worst-case interconnect delay. Here, the performance degradation results from inserting delay clocks into memory access cycles so that the system can operate without failure. In other words, while GGBA I would fail in a real SoC, GGBA III would work fine but with all bus delays set to accomodate the worst-case interconnect delay.

7.3.2.2 Comparison II

In Comparison II of Table 13, GGBA III is chosen as the baseline for performance improvement against the execution time of GGBA II. The impact of detailed interconnection delay estimation in the design phase results in a 35.3% reduction in execution time when we compare GGBA II, an interconnect delay aware GGBA system, with GGBA III, a non-optimized system with regard to memory access cycles. As shown in the cases of (a) 300MHz, (b) 200MHz and (c) 100MHz bus clocks in Table 13, different bus clocks result in different memory access patterns due to interconnect delays. Therefore, as the bus clock increases, the effect of detailed interconnect delay in a system is bigger as shown in Comparison II (and Comparison I) of Table 13. In short, comparing the two Bus Systems (GGBA II and GGBA III) which would work

Table 13: Performance Comparison

GGBA System	Execution Time [ns/packet]	Comparison I [increase in execution time]	Comparison II [decrease in execution time]
1. GGBA I (no interconnect delay, unrealistic)	1,218,455	0.0%	-
2. GGBA II (3, 3, 4 and 5 clock delays in each data path from PE 1 to PE 4)	2,057,487	68.9%	35.3%
3. GGBA III (5 clock delays in all data paths)	3,180,220	161.0%	0.0%

(a) 300 MHz Bus Clock

GGBA System	Execution Time [ns/packet]	Comparison I [increase in execution time]	Comparison II [decrease in execution time]
1. GGBA I (no interconnect delay, unrealistic)	1,825,751	0.0%	-
2. GGBA II (2, 2, 3 and 3 clock delays in each data path from PE 1 to PE 4)	2,323,670	27.3%	27.4%
3. GGBA III (3 clock delays in all data paths)	3,198,620	75.2%	0.0%

(b) 200 MHz Bus Clock

GGBA System	Execution Time [ns/packet]	Comparison I [increase in execution time]	Comparison II [decrease in execution time]
1. GGBA I (no interconnect delay, unrealistic)	3,644,003	0.0%	-
2. GGBA II (1, 1, 2 and 2 clock delays in each data path from PE 3 to PE 4)	3,862,686	6.0%	10.1%
3. GGBA III (2 clock delays in all data paths)	4,297,056	17.9%	0.0%

(c) 100 MHz Bus Clock

Table 14: 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 count
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
HybridBA	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

if implemented in reality, our bus controllers optimized for interconnect delay reduce application execution time by up to 35.3%.

7.3.3 Generation Time and Gate Counts of Each Bus System

Table 14 shows the generation time for the Bus Systems generated using BusSynth. Table 14 also shows the gate counts (in NAND2 gate equivalents) of the Bus System logic after synthesizing the logic using the LEDA TSMC 0.25 μm standard cell library with Synopsys Design Compiler. BusSynth can generate a Bus System having any number of PEs, but the table shows Bus Systems having a maximum of 24 PEs. In the generation time column, each Bus System shown in Table 14 takes less than one second to generate using BusSynth. 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 PEs and the modeling of required modules and their interfaces. Note that BusSynth achieves performance superior to the hand design of GGBA and CCBA; furthermore, the user specified 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, PE types and software programming style resulting in a system having higher performance. This goal is

accomplished through BusSynth, which allows the user to easily design a custom Bus System in a matter of seconds.

7.4 Summary

In this chapter, we have explained three user applications to evaluate the generated Bus Systems, and then we have described our experimental setup for the evaluation. After that, we have shown the evaluation results, the generation time of each Bus System from BusSynth and gate counts of each Bus System.

CHAPTER VIII

CONCLUSION

In this dissertation, we have described a methodology to generate custom Bus Systems for a multi-processor SoC design. We designed a bus synthesis tool BusSynth by exploiting this methodology. Using BusSynth, we have generated five different Bus Systems as examples: BFBA, GBAVI, GBAVIII, HybridBA and SplitBA. The BusSynth algorithms have been described in significant detail and have been shown to finish in reasonable time (under a second) in the practical cases shown. In Section 7, 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. We showed that BusSynth 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. In particular, we showed up to 41% reduction in application execution time with a customized bus architecture.

In the design of a high-speed SoC, interconnect delay becomes a major concern. we have described a methodology to generate a custom bus architecture based on accurate estimations of interconnect delay. The interconnect delay is provided from an accurate delay modeling established from an estimated chip floorplan. Our bus synthesis tool (BusSynth) generates a custom bus system (e.g., GGBA) that adapts to detailed interconnect delay predictions, and the generated systems are evaluated with a user application, an OFDM transmitter, in order to illustrate the impact of interconnect delay during the design phase. The results of our case study show that there is performance improvement due to suitable memory access control that adapts

predicted interconnect delay. In particular, we showed up to 35.3% reduction in application execution time for a customized bus architecture.

Finally, our methodology gives us a great benefit as follows: fast design space exploration of bus architectures across performance influencing factors (e.g., bus types, processing element types and software programming style), performance improvement due to the bus system generation by the user options that are suitable for the user application, and shortening SoC design time by quick bus architecture generation.

REFERENCES

[1] ARM, "AMBA Specification," Available HTTP: http://www.arm.com/ products/solutions/AMBAHomePage.html, 2004.

[2] Artisan, "Memory Generator," Available HTTP: http://www.artisan.com/ products/memory.html, 2004.

[3] Bergamaschi, R. A. and Lee, W., "Designing Systems-on-Chip Using Cores," *Proceedings of the 38th Design Automation Conference (DAC'00)*, pp. 420–425, June 2000.

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

[5] Cesário, W., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A., Gauthier, L., and Diaz-Nava, D., "Multiprocessor SoC Platforms: A Component-Based Design Approach," *IEEE Design & Test of Computers*, Vol. 19, pp. 62–63, November 2002.

[6] Chou, P., Ortega, R., and Borriello, G., "IPCHINOOK: An Integrated IP-based Design Framework for Distributed Embedded Systems," *Proceedings of the 37th Design Automation Conference (DAC'99)*, June 1999.

[7] CoWare, "CoWare N2C: Design Automation Technology for System-Level Design," Available HTTP: http://www.coware.com, 2004.

[8] Cyr, G., Bois, G., and Aboulhamid, M., "Syntehesis of Communication Interfaces for SoC using VSIA Recommendations," *Proceedings of Design, Automation and Test in Europe (DATE'01)*, pp. 155–159, March 2001.

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

[10] Dick, R. and Jha, N., "MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis," *Proceedings of Design, Automation and Test in Europe (DATE'99)*, pp. 263–270, March 1999.

[11] Dittenhofer, B., "Connecting Multi-source IP to a Standard On-Chip Architecture," Available HTTP: http://www.palmchip.com/pdf/CP-9248P.pdf, 2000.

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

[13] General Public License (GPL) in OpenCores. Available HTTP: http://www.opencores.org/faq.cgi/section/1/1.1#1.1.

[14] Gharsalli, F., Lyonnard, D., Meftali, S., Rousseau, F., and Jerraya, A., "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.

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

[16] GNU General Public License (GPL). Available HTTP: http://www.gnu.org/copyleft/gpl.html.

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

[18] Hewlett-Packard, "CACTI," Available HTTP: http://research.compaq.com/wrl/people/jouppi/CACTI.html, 2004.

[19] Hsieh, C. and Pedram, M., "Architectural Energy Optimization by Bus Splitting," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, pp. 408–414, April 2002.

[20] IBM, "The CoreConnect Bus Architecture," Available HTTP: http://www.chips. ibm.com/products/coreconnect, 2004.

[21] Kim, D. and Stuber, G., "Performance of Multiresolution OFDM on Frequency-selective Fading Channels," *IEEE Transaction on Vehicular Technology*, Vol. 48, pp. 1740–1746, September 1999.

[22] Lee, M., "A Fringing and Coupling interconnect Line Capacitance Model for VLSI On-Chip Wiring Delay and Crosstalk," *IEEE International Symposium On Circuits and Systems (ISCAS'96)*, Vol. 4, pp. 233–236, May 1996.

[23] Lyonnard, D., Yoo, S., Baghdadi, A., and Jerraya, A., "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," *Proceedings of the 39th Design Automation Conference (DAC'01)*, pp. 518–523, June 2001.

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

[25] Mentor Graphics Platform Express. Available HTTP: http://www.mentor.com/platform_ex, 2004.

[26] Moore, G., "Cramming More Components Onto Integrated Circuits," *Electronics*, Vol. 38, pp. 114–117, April 1965.

[27] The MOSIS Service, TSMC 0.25 Micron Process. Available HTTP: http://www.mosis.org/products/fab/vendors/tsmc/tsmc025/index.html, May 2003.

[28] Motorola, "MPC755A RISC Microprocessor Hardware Specification," Available HTTP: http://e-www.motorola.com/webapp/sps/site/prod_summary, 2004.

[29] MSSG, "Mpeg2encoder/moeg2decoder," Available HTTP: http://www.mpeg.org/MPEG/MSSG/Codec/readme.txt, 1996.

[30] Nicolescu, G., Yoo, S., Bouchhima, A., and Jerraya, A., "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.

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

[32] Open Core Protocol (OCP) Research License. Available HTTP: http://www.ocpip.org/socket/ocpspec/licensesignup.

[33] OpenCores. Available HTTP: http://www.opencores.org.

[34] Parmchip, "Overview of the CoreFrame Architecture," Available HTTP: http://www.palmchip.com/pdf/CP-9152W-1.01.pdf, 2004.

[35] Pentek, "Operating Manual for Model 4290 and 4291," Available HTTP: http://www.pentek.com, 2004.

[36] Prosilog, "Magillem Technical Feature," Available HTTP: http://www.prosilog.com, 2004.

[37] Rao, K. R. and Hwang, J. J., *Technique & Standards for Image Video & Audio Coding*. Upper Saddle River, New Jersey: Prentice Hall PTR, 1996.

[38] Ryu, K., Shin, E., and Mooney, V., "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.

[39] Ryu, K., Talpasanu, A., Mooney, V., and Davis, J., "Interconnect Delay Aware RTL Verilog Bus Architecture Generation for an SoC," *Advanced System Integrated Circuits (AP-ASIC'04)*, August 2004.

[40] Sakurai, T., "Closed-Form Expressions for Interconnection Delay, Coupling, and Crosstalk in VLSI's," *IEEE Transaction on Electron Devices*, Vol. 40, pp. 118–124, January 1993.

[41] Shin, C., Kim, Y., Chung, E., Choi, K., Kong, J., and Eo, S., "Fast Exploration of Parameterized Bus Architecture for Communication-Centric SoC Design," *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pp. 352–357, February 2004.

[42] Silicore, "Electronic Design – Sensors – IP Cores," Available HTTP: http://www.silicore.net, 2004.

[43] Silicore, "WISHBONE System-on-Chip(SoC) Interconnection Architecture for Portable IP Cores," Available HTTP: http://www.silicore.net/pdfiles/wishbone/specs/wbspec_b2.pdf, 2004.

[44] Sonics, "Open Core Protocol," Available HTTP: http://www.sonicsinc.com/sonics/products/opencoreprotocol, 2004.

[45] Sonics, "Sonics μnetwork technical overview," Available HTTP: http://www.sonicsinc.com/sonics/support/documentation/whitepapers/data/Overview.pdf, 2004.

[46] Synopsys, "Cocentric System Studio Enables Verification at Multiple Levels of Abstraction with SystemC," Available HTTP: http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html, 2004.

[47] Synopsys, "Data Sheet: CoCentric System Studio," Available HTTP: http://www.synopsys.com/products/cocentric_studio/cocentric_studio_ds.pdf, 2004.

[48] Synopsys, "DesignWare Library," Available HTTP: http://www.synopsys.com/products/designware/dwlibrary.html, 2004.

[49] Synopsys, "Synopsys Design Compiler," Available HTTP: http://www.synopsys.com/products/logic/design_compiler.html, 2004.

[50] Synopsys, "VCS Data Sheet," Available HTTP: http://www.synopsys.com/products/simulation/vcs_ds.html, 2004.

[51] Talpasanu, A. and Davis, J., "Bus interconnect structure for a system-on-a-chip multiprocessor system," [Online]. Available: http://www.cc.gatech.edu/tech_reports, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-04-03, December 2003.

[52] Thepayasuwan, N. and Doboli, A., "Layout Conscious Bus Architecture Synthesis for Deep Submicron Systems-on-Chip," *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pp. 108–113, February 2004.

[53] UMC, "Chip sizer," Available HTTP: http://eproject.umc.com/dse, 2004.

[54] Uyemura, J., *Introduction to VLSI Circuits and Systems*. New York: John Wiley & Sons, 2002.

[55] Virage Logic, Memory Compiler. Available HTTP: http://www.viragelogic.com/products/compilers, 2004.

[56] Wieferink, A., Kogel, T., Braun, G., and Nohl, A., "A System Level Processor/Communication Co-exploration Methodology for Multi-processor System-on-Chip Platforms," *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pp. 1256–1261, February 2004.

[57] Yoo, S., Nicolescu, G., Lyonnard, D., Baghdadi, A., and Jerraya, A., "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.

[58] Yoon, H., Yoon, J., Lee, H., Lim, K., and Hwang, C., "A 4Gb DDR SDRAM with Gain-Controlled Pre-Sensing and Reference Bitline Calibration Schemes in the Twisted Open Bitline Architecture," *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC'01)*, pp. 378–379, 467, February 2001.

PUBLICATIONS

This dissertation is based on and extends the work and results presented in the following publications:

[1] Ryu, K. and Mooney, V., "Automated Bus Generation for Multiprocessor SoC Design," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems* (TCAD'04), 2004.

[2] Ryu, K., Talpasanu, A., Mooney, V., and Davis, J., "Interconnect Delay Aware RTL Verilog Bus Architecture Generation for an SoC," *IEEE Asia-Pacific Conference on Advanced System Integrated Circuits* (AP-ASIC'04), August 2004.

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

[4] Ryu, K. and Mooney, V., "Automated Bus Generation for Multiprocessor SoC Design," [Online]. Available: http://www.cc.gatech.edu/tech_reports, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-64, December 2002.

[5] Ryu, K., Shin, E., and Mooney, V., "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.

The following publication is related but not covered in this dissertation.

[1] Lee, J., Ryu, K., and Mooney, V., "A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* (ERSA'02), pp. 31–37, June 2002.

POSTER PRESENTATIONS/DEMONSTRATIONS

This dissertation is based on and extends the work and results presented in the following posters and demonstration:

[1] Ryu, K. and Mooney, V., "Automated Bus Generation for Multiprocessor SoC design," Ph.D. Forum at the 40th Design Automation Conference (DAC'03), June 2003.

[2] Ryu, K., Shin, E., Lee, J., and Mooney, V., "A Framework for Automatic Generation of Bus Systems and a HW/SW RTOS for Multiprocessor SoC," University Booth at the 39th Design Automation Conference (DAC'02), June 2002.