1-1-2010

Modular, Configurable Bus Architecture for Ease of IP Reuse on System on Chip and ASIC Devices

Naveendran Balasingam
Portland State University

Modular, Configurable Bus Architecture Targeted for Ease of IP Reuse on System on Chip and ASIC Devices

by

Naveendran Balasingam

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Xiaoyu Song, Chair
Douglas V. Hall
Jingke Li

Portland State University
2010

Abstract

Integrated Circuit (IC) designs are increasingly moving towards Intellectual Property (IP) reuse for various targeted products and market segments. Therefore, there is a need to share and synergize internal bus architectures to enable the reuse of IP blocks for various ASIC and SoC applications. Due to the different market segments of various ASICs and SoCs, design teams and architects have opted to use customized internal bus architectures to suit the respective targeted features for their market segments. As a result, many ASIC and SoC companies that produce microprocessors for computers, microcontrollers for consumer electronics as well as memory and I/O controller chipsets have opted to use different internal interfaces, designs and IPs for the different products that they sell. A modular and configurable bus architecture that is flexible and capable of supporting IPs from various ASICs and SoCs would serve to solve many of the problems relating to IP reuse for various applications from a front end design perspective. There are several approaches to resolve this, for example, using a standard existing open source bus, a new all-encompassing bus that covers the needs of the majority of designs and a customization of a particular bus level such as the interface layer, where part of the bus features are fixed and the rest of them are determined by individual design groups. This research covers the analysis of existing bus architectures in industry and considers the various options for bus architecture optimization for design modularity, bus performance and IP reuse with existing technology. The architecture definition, design, logic simulation and performance comparisons of the proposed bus architecture on industry standard RTL design and validation tools was then conducted.

Acknowledgements

Dr Xiaoyu Song for his guidance during the course of the Master's program and continuous support of my efforts.

Professor Mark Faust for taking the time to review early revisions of the thesis and for general guidance during the course of the Master's program.

Renjit Retnamma for reviewing various revisions of my thesis and for providing important and timely feedback.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

1 AHB: ARM's Advanced High Performance Bus

2 AMBA: Advanced Microcontroller Bus Architecture

3 APB: ARM's Advanced Peripheral Bus

4 ASB: ARM's Advanced System Bus

5 ASIC: Application Specific Integrated Circuit

6 DCR: IBM Core Connect's Device Control Registers

7 FPGA: Field Programmable Gate Array

8 GA: Genetic Algorithm

9 HDL: Hardware Description Language

10 IBM: International Business Machines Corporation

11 IC: Integrated Circuit

12 IP: Intellectual Property

13 MCB: Modular Configurable Bus

14 Memctrl: Memory Controller

15 OCP: Open Core Protocol

16 OCP-IP: Open Core Protocol International Partnership

17 OPB: IBM Core Connect's On-Chip Peripheral Bus

18 PLB: IBM Core Connect's Processor Local Bus

19 RAM: Random Access Memory

20 RTL: Register Transfer Level

21 SB: Sideband

SD: Secure Digital

SDRAM: Synchronous Dynamic Random Access Memory

SoC: System On Chip

SPI: Serial Peripheral Interface

TDMA: Time Division Multiplexing Access

USB: Universal Serial Bus

MESI: Type of Cache Coherence Protocol

Chapter 1: Introduction and Motivation

Integrated Circuit (IC) designs have traditionally consisted of partitions containing a set of design units and the connectivity between these design units or blocks was either decided between the designers or standardized for a particular product team. The interconnectivity between the major design blocks could be point to point or shared and arbitrated by an internal bus arbiter within the design. In a CPU design for example, a branch prediction unit that needs to communicate with an instruction fetch/decode unit would have a point- to point connection while a power management controller that needs to communicate with the majority of the design units would have a shared bus connection to its targets.

The interconnects within these design units have typically been customized for a particular design target due to performance and timing considerations in the ASIC design flow. Furthermore, many design companies practice the use of inheritance of existing designs as a starting point for subsequent generations of products for a particular market segment, effectively carrying the problem of non-standardized connectivity and bus architecture forward. The evolution of the design flow process whilst still maintaining legacy design blocks in the current designs to reduce risk of logic and timing bugs has also added difficulty to reuse IPs from one design to another, thus diminishing the opportunity for design modularity. As a result, many of the designs targeted for a particular market segment or product group are diverged in terms of internal bus connectivity and validation methodologies from other product groups within the same

company or product division. Figure 1.1 illustrates the typical ASIC design flow used for modern day ICs. Due to the potential risk to timing, clock treeing and signal integrity for high performance products, many ICs maintain legacy design units and make incremental changes to design blocks that are considered timing risks. Furthermore, legacy features that are still supported by newer revisions of the IC but have no design changes are also maintained from earlier generations of the product.

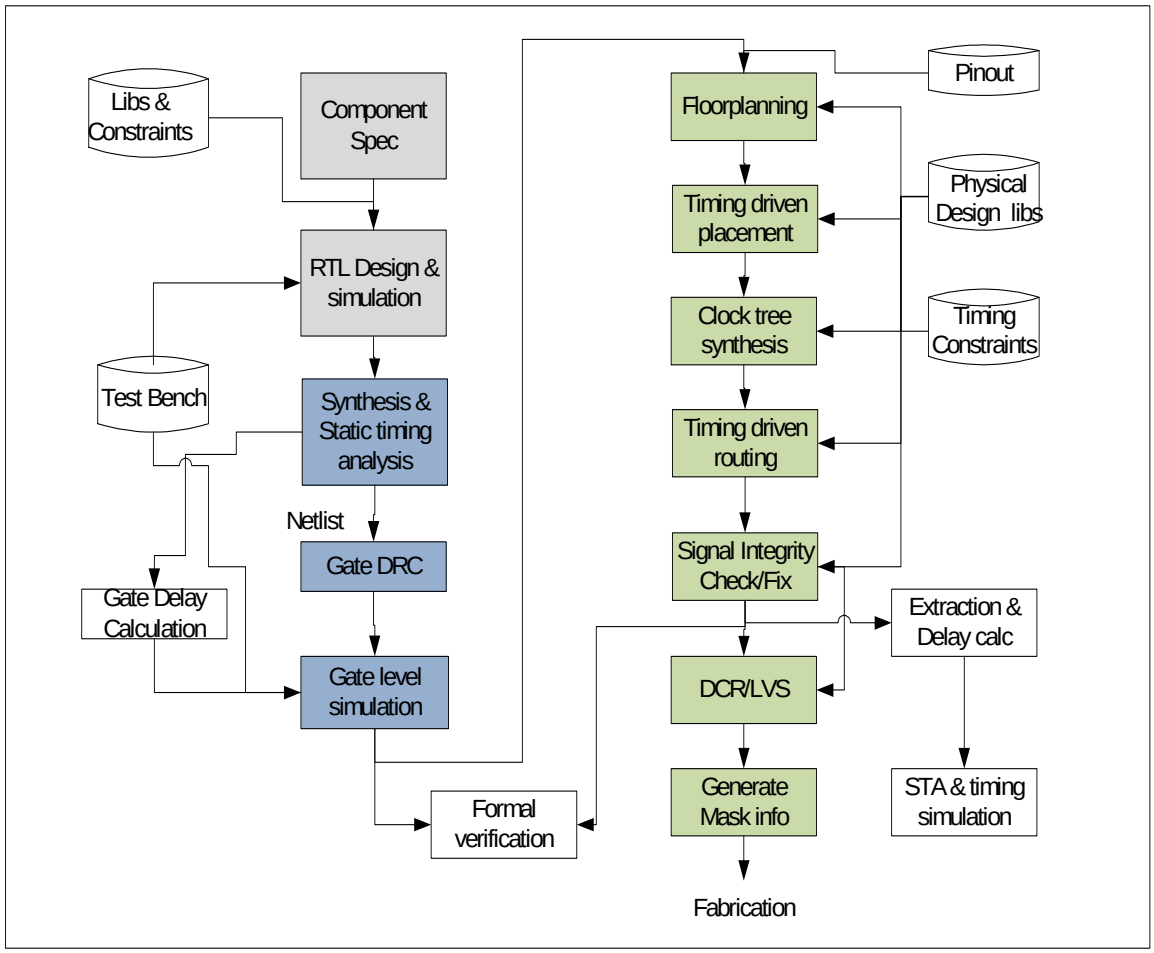

Figure 1.1: Standard ASIC Design Flow

The concept of IP reuse and the use of modular IP blocks that can be utilized by different designs is not a new one. In software there are many examples of code that are developed as functions or classes that can be reused by changing their parameters to suit the use of the intended programs. The conversion of HDL design blocks into smaller IP blocks does pose challenges and requires careful design consideration due to the complexity of the design flow. With the increasing complexity of ASIC and SoC designs as well as the reduction of the design and development phase of the product life cycle due to increasing demands for time to market for these products in industry, the ability to easily reuse existing designs or IPs has become increasingly important. Moreover, for large IC producers that design multiple flavors of a particular IP for different target products, it is more efficient to have one source of this IP that can be configured for reuse across the different products and flavors.

Furthermore, with the availability of hard and soft IPs that can be purchased from various IP producers, the cost of purchasing readily available IPs could be less than maintaining an entire design team. With large IC producers looking for ways to reduce product costs and increase profit margin, the potential for increase in design productivity with reuse of either in-house IP or externally purchased IP is a promising prospect.

There are, however, several challenges to modular design and IP reuse for ASIC and SoC designs. One of the major challenges faced is the connectivity and communication of these design units or IP blocks in a particular design. This is because, even if the majority

of large designs have a standard bus that they communicate through, there would typically be a set of stray sideband communication signals that are used for various functions such as power management, test/debug and interrupts. There are also additional complexities related to IP reuse such as performance, gate count and timing impacts to the design. For design units that have point to point connection to other units, there is less of a need to consider their interconnects while the majority of larger design units and IP blocks are typically required to interface with many different IPs and will require a standard bus interface and protocol for ease of reuse.

The focus of this research is to develop a modular, configurable bus architecture intended to enable simpler and more efficient IP reuse without impacting design performance and quality. The research also covers the bus protocol, connectivity and arbitration between IP blocks interconnected with other major IP blocks on this bus architecture. The research does not focus on connectivity between the smaller, more specific design units that would typically reside within a particular IP block. Therefore, the idea is that a standardized bus architecture that supports modular design and is configurable for various design requirements would greatly benefit the ability to reuse IP and design blocks used in ASIC and SoC products.

The thesis first researches existing solutions for IP reuse and modular design in industry, covered in chapter two. This covers research into existing bus architectures that advocate design reuse and configurability of interconnects to and from the respective busses. Since

System on Chip (SoC) products have been advocating design reuse of IP blocks for some years due to their requirement for lower time to market and for lower design costs; SoC busses in industry were analyzed to compare and contrast their complexities, limitations and features that exist to develop a modular, configurable bus architecture solution.

The thesis then goes on to describe the architecture of the Modular Configurable Bus (MCB) solution in chapter three. This chapter describes the features that were leveraged from the review of the existing bus solutions in industry before detailing the architecture features that are required to build a modular, configurable bus solution.
In chapter four, the reused IP blocks and system architecture to assess the functionality and effectiveness of the MCB solution is described. After which, a description of the re-used IPs, the required additions to the IP blocks and interconnect system for MCB and an outline of the simulated system are provided.

Chapter five covers the tabulation and analysis of simulation data for key test cases covering the data paths in the simulated system. The performance data, reusability analysis and gate count impact are then discussed and compared with the Wishbone bus architecture solution. Chapter six describes the conclusions that can be drawn from all the previous chapters and discusses areas of future research for further improvement of the MCB solution.

Chapter 2: Review of Existing Literature and Innovations in Bus Architecture

This section summarizes the research conducted to explore the existing methods and approaches currently available to overcome or mitigate the difficulties of integrating Intellectual Property (IP) blocks with existing bus architectures. There was also exploration and research in the area of bus architecture optimization for IP reuse particularly for SoC products.

The approach was to first analyze the existing bus architectures available that focused on the ease of IP reuse where the IBM core connect bus [2], ARM's AMBA bus [3] Wishbone bus[1], and Open Core Protocol International Partnership's OCP spec [4] were studied. These bus architectures were then compared and contrasted to determine their strengths, weaknesses and whether they are able to solve the problems of IP reuse of HDL design blocks. After this analysis was completed, the exploration into existing research and publications in the area of bus architecture for IP reuse was conducted where the Automated Bus Generation [6], Fast Exploration of Parameterized Bus Architectures [7] and Lottery Bus Architecture [8] publications were found and studied. These methods were also analyzed, compared and contrasted for their relative strengths and weaknesses.

IBM Core Connect Analysis

IBM Core connect consists of two main busses and one peripheral bus for its various IP requirements. There is a high speed Processor Local Bus (PLB), a lower speed On-chip Peripheral Bus (OPB) for lower speed device control and a Device Control Register (DRC) bus for static sideband signals such as configuration and DFT signals used by high speed and low speed IPs respectively.

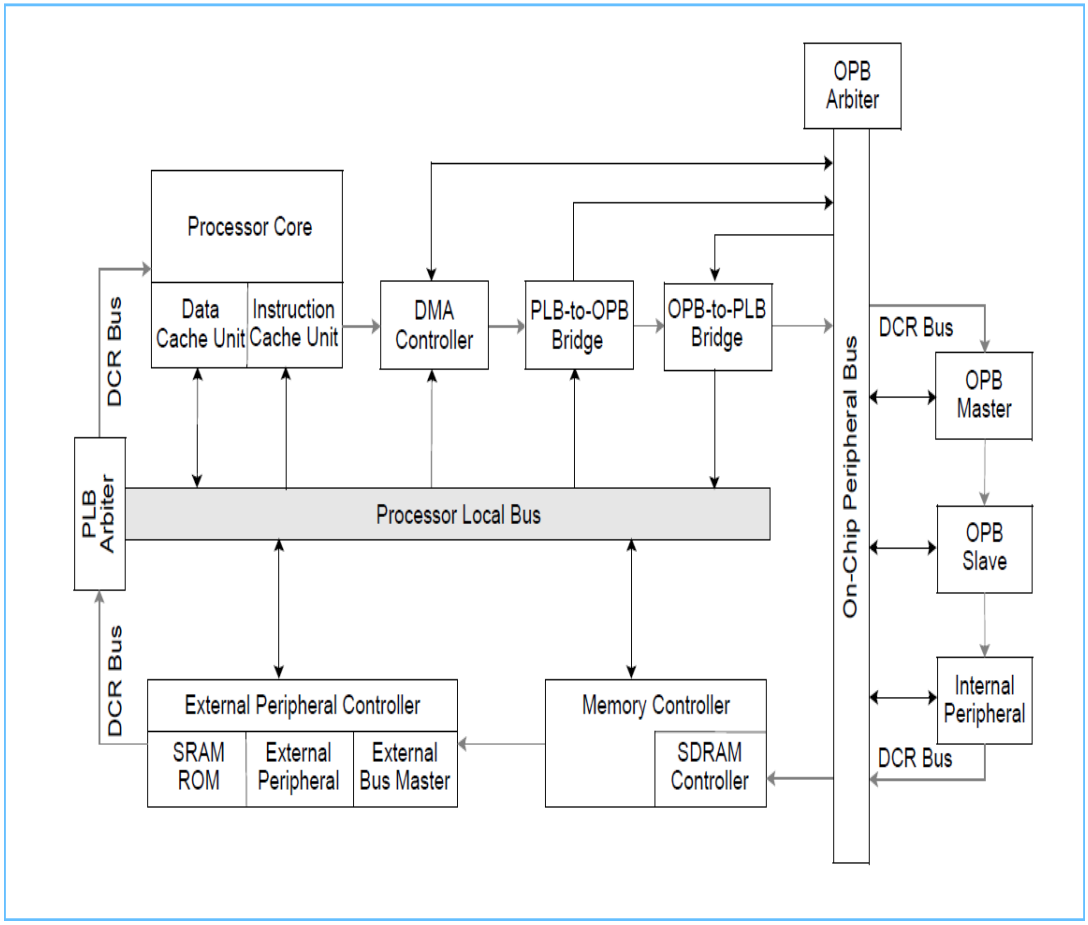

Figure 2.1: IBM Core Connect Implementation Example [5]

The Processor Local Bus (PLB) is the main system bus used for high speed and highest performance transactions. PLB supports data width of 32, 64, 128 and 256 bits on the bus and is made up of a separate set of read and write signals for the data bus. The PLB bus supports multi-master bus access with central arbitration as part of the PLB architecture. The On-Chip Peripheral Bus (OPB) is a lower speed, lower throughput bus used for peripheral devices or IPs. OPB also allows a configurable bus data width and supports multi-master arbitration between the bus agents. In order for the peripheral devices on OPB to connect with devices or IPs on the PLB bus, a PLB to OPB bridge is required. This is illustrated in Figure 2.1.

The IBM core connect architecture also includes a Device Control Register (DCR) Bus that is used to communicate configuration and control information to reduce these types of transactions on the PLB and OPB busses. The DCR bus is made up of a single master that targets various DCR registers throughout the system. The DCR can also be used to transfer DFT as well as other sideband information that function at lower speeds.

ARM's AMBA Bus Architecture

The AMBA bus has many similarities to the approach taken by the IBM core connect bus. There are also separate busses for high performance versus low performance IPs. The three types of busses used by AMBA are Advanced High Performance Bus (AHB), Advanced System Bus (ASB) and the Advanced

Peripheral Bus (APB). An example of the AMBA bus implementation is shown in Figure 2.2 [3]:

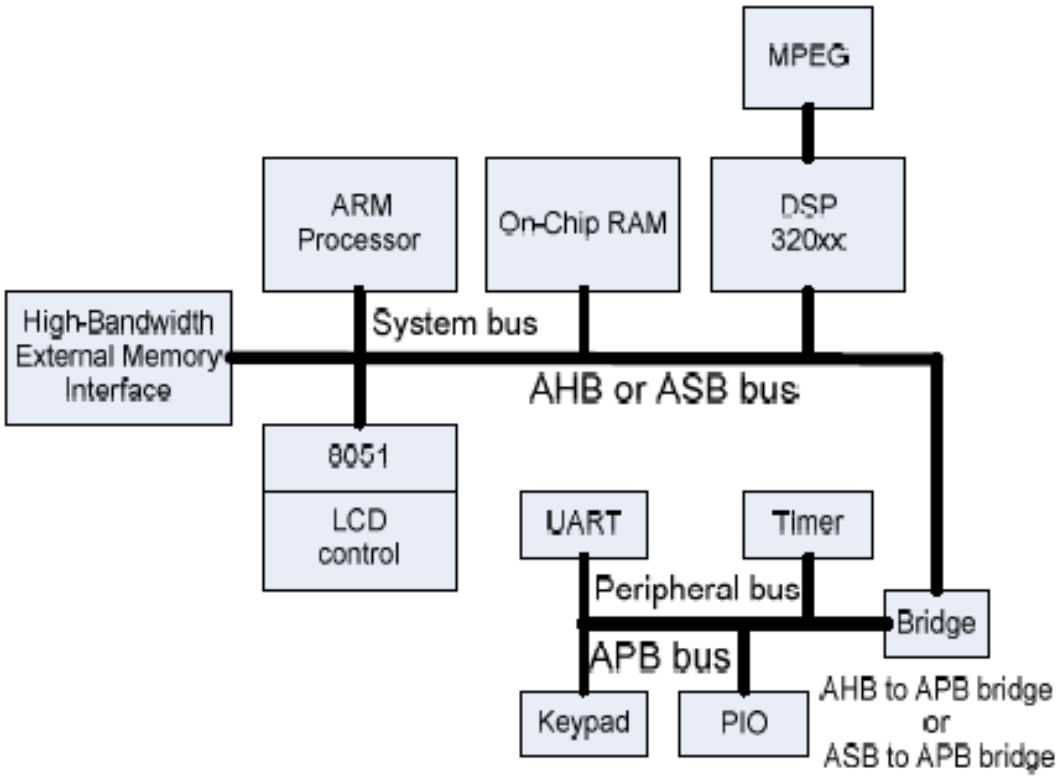

Figure 2.2: AMBA Bus Implementation Example [3]

The ARM AHB bus is a high performance, high bandwidth bus that is meant for high performance IPs much like IBM's PLB bus. The AHB bus is also catered for multi-master access with bus arbitration on the AHB. In addition, the AHB bus is capable of supporting burst transactions, bus pipelining and split transactions.

The ASB bus is also meant for high performance design or IP block interconnection although it has less features than AHB. ASB supports bus pipelining and multi-master access but doesn't support burst and split transactions. ARM's APB bus is a lower speed

bus with a lower expected throughput for the IPs and devices it is connected to. It also is a simpler interface compared to AHB and ASB and supports lesser features. In order to support the APB bus in a system that also uses AHB or ASB, a bus bridge is required to support transactions for bus agents on all of these busses.

The Wishbone Bus Architecture

The wishbone bus is a more evolved SoC bus compared to Core Connect and AMBA in terms of design modularity and configurability. It is also an open source architecture. There is a standardized connection to IP blocks although the internal IP core is dependent on IP core development team. It also can be used to support soft, firm and hard core IPs. There is only 1 bus specification for wishbone that can be used for various bus topologies as shown in the following figures [1]:

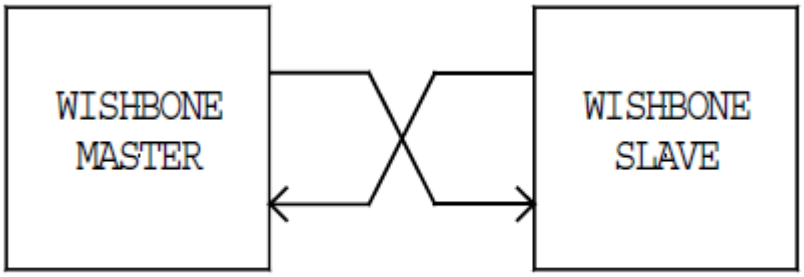

Figure 2.3: Wishbone Point to Point Topology [1]

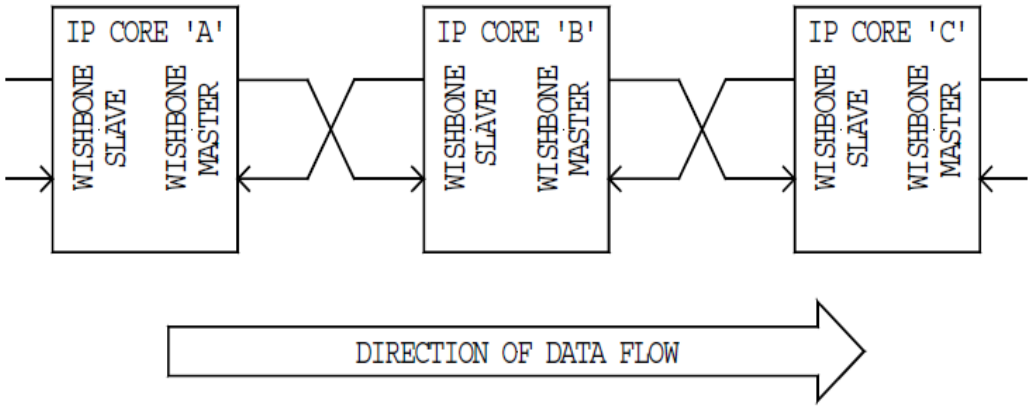

Figure 2.4: Wishbone Dataflow/Pipelined Bus Topology [1]

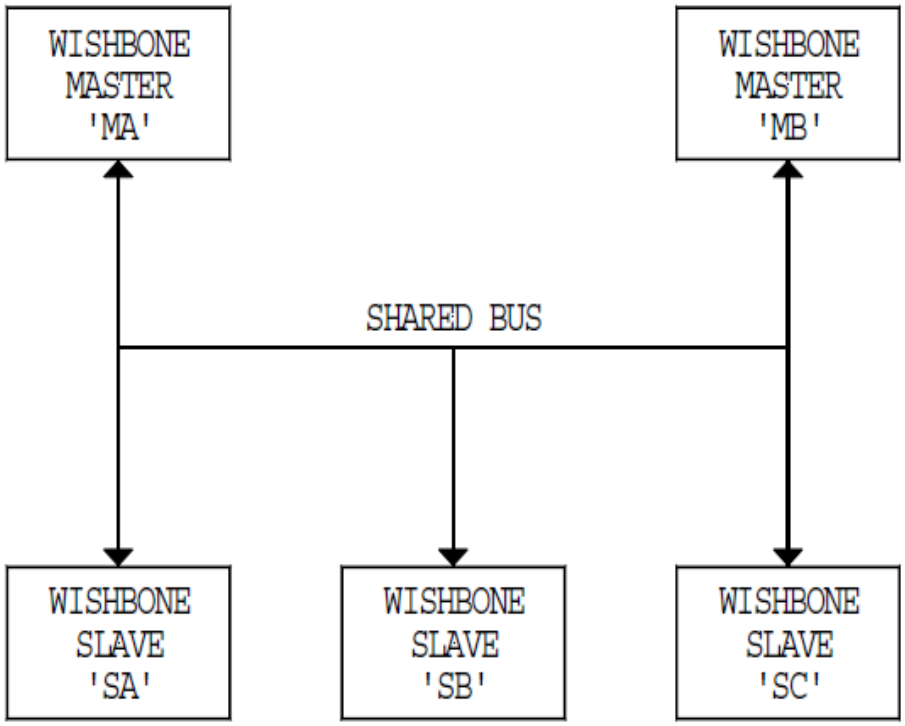

Figure 2.5: Wishbone Shared Multi-Master Bus Topology [1]

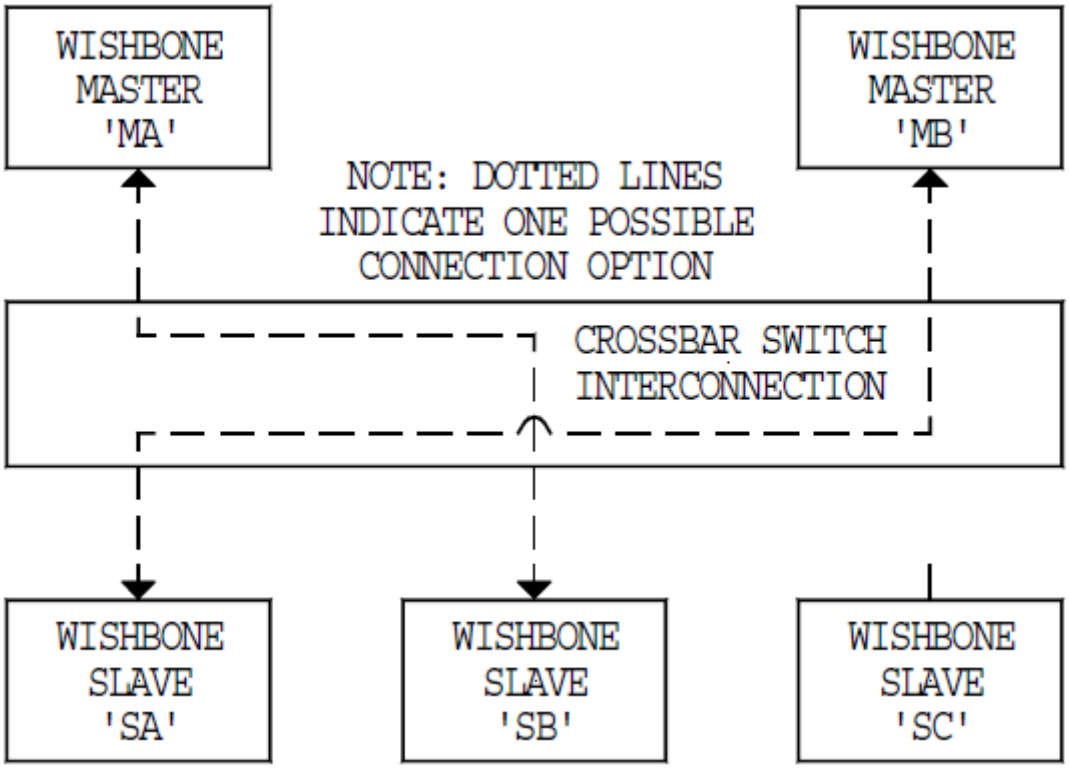

Figure 2.6: Wishbone Crossbar Switch Bus Topology [1]

As shown in Figures 2.3-2.6, the wishbone bus supports a variety of bus topologies, data widths and bus bandwidths with this single bus architecture. This means that the same wishbone bus can be used for high speed IP interconnect that would be on a high speed bus such as PLB as well as the lower speed bus such as OPB. The Wishbone architecture is able to achieve this because it is made up of a standard interface and a standardized data exchange protocol between the interconnect modules and IP blocks. Wishbone also supports widely used data transfers such as read, write, block and burst transfers. Wishbone bus agents are made up of masters and slaves and there is support for multi

master transactions. The frequency and power ratings of this bus are also configurable which increases its potential for reuse.

Open Core Protocol International Partnership's (OCPIP) OCP Bus Interface

The OCP methodology defines a system level integration of IP cores. The focus is on promoting IP reuse with defined boundaries for interface level connectivity without actually defining all the features of a bus. OCP allows for peer to peer communication, master to slave as well as multi threaded applications.

In the OCP spec, the bus system is not defined, it is up to the designer to convert bus system to OCP equivalent signals. OCP promotes a flexible implementation of a bus system, where bus features can be configured as chosen by designer. As such, OCP requires a timing interface file in a specified format to be defined since bus timing is not specified by OCP. Non OCP interfaces that want to use OCP need to also be described with an interface file. OCP's main focus is on interface boundary definition/design. Signals are completely de-multiplexed by dedicated unidirectional signals. OCP is a bus independent interface, with address decoding/device selection and arbitration scheme defined by design team. There is also extension to initial OCP spec to allow support for sideband interface and cache coherence signals.

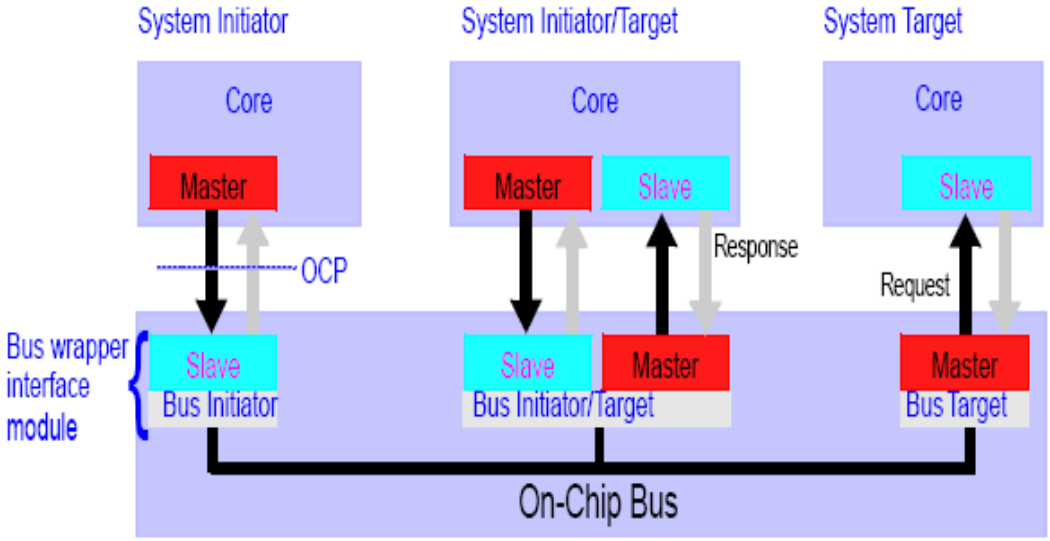

Figure 2.7: OCP implementation example [4]

A summary of the busses analyzed is given in Table 2.1 below:

Bus	**Feature Set**	**Limitation**
IBM Core Connect Bus	Busses split between high performance and low performance buses. There is no comprehensive single bus for all types of IPs. There is compatibility with AMBA bus, where a bridge can be used to connect to AMBA buses Accounts for a separate bus for sidebands (DCR) Synchronous bus design PLB high speed bus supports multi-master access	Not a comprehensive bus, needs three types to accomplish task of one bus Does not provide level of flexibility for designs of different kinds, i.e., limited width of add bus some key data bus capabilities need specific signal settings for enabling.

AMBA Bus	Busses split between high performance and low performance buses. There is no comprehensive single bus for all types of IPs. AHB Bus supports multiple bus masters, burst and split transactions. ASB supports multiple bus masters but not burst and split transactions Address bus width is limited APB (peripheral bus) is for low bandwidth and lower performance IPs Needs bridge to connect to AHB/ASB	No method of taking care of sideband signals, for example, test signals and configuration Not a comprehensive bus, needs three types of busses to accomplish task of one bus Does not provide required level of flexibility for designs of different kinds, i.e. limited width of address bus and some key data bus capabilities for flexibility not incorporated
Wishbone Bus	Consists of a standard interconnect of IPs to bus interconnect for data exchange. Bus architecture is independent of underlying HDL. Supports various data transfer types such as basic read, write and block transfers. Multi-master access supported Configurable address and data bus sizes with limit of 64bits. Works for on or off-chip busses	No obvious method of taking care of sideband signals for test signals and configuration data transfer that inhibits performance for designs with lots of configuration setup/DFT sidebands. No allocation for cache coherence signaling. Although good for first time design, can add complexity for legacy IP implementation since interface needs to change. Data bus width limited to 64bits.

OCP Bus	Employs a standard interconnect to the OCP bus for data exchange that is independent of underlying HDL. Supports wide variety of transfers although it is more of a socket interface architecture than a bus architecture	Requires additional effort in understanding syntax of timing interface file Designers required to learn syntax of the interface file to add to OCP definition. No direct way to bridge to this bus, requires bus to be defined, there is some talk of enabling automated feature to assist bridging but no examples given. Validation challenge since bus signals have a wide variety of use and flexibility

Table 2.1: Summary of Analyzed Bus Architectures

In addition, these four bus architectures were analyzed for IP reusability, performance, arbitration schemes, pin count and availability of error detection and correction to further assess their effectiveness. The reusability rating under the column "IP Reusability" is better if the number is higher.

Bus	**IP Reusability**	**Performance**	**Arbitration scheme**	**Pincount**	**Error Detection/ Correction**
IBM Core Connect Bus	2 out of 7 due to complexity and fixed arbiter	Split into high performance and low performance busses, supports up to 16 bytes of data transfers per clock cycle	fixed, uses internal arbiter	>400 pins on PLB bus due to large data width, address width and control signals for them	Supports error detection control bits

AMBA	3 out of 7, doesn't support flexible topology with single bus, fixed arbiter	Split into high performance and low performance busses, supports burst, typically 4 -8 bytes per clock cycle on high speed AHB bus	fixed, uses internal arbiter	Variable, typically 128 data,32 addresses , 40 control signals	ERR, RTY Signals for detection,
Wishbone Bus	5 out of 7, no sideband support	Supports high performance and low performance accesses, can support up to 8 bytes of data transfers per clock cycle for block transfers	Open	variable, maximum 128 data, 64 address, 12 controls	ERR, RTY Signals for detection, tag can hold correction data
OCP	5 out of 7, complex timing interface file information may not be available for reused IPs	Supports high performance and low performance accesses, can support variable data widths, typically 8 bytes of data transfers per clock cycle for burst transfers	Open	variable, typically 128 data, 64 address,9 controls for basic transactions	Slave response signals support error detection, some extendable data for error correction can be supported

Table 2.2 : Bus Architecture Feature and Performance Analysis

The research and exploration of bus architecture optimization techniques and advancements for IP reuse was then conducted. This was done by searching through publications of related material in computer engineering related journals and conferences in the field of bus architecture for ASICs and SoCs. The three most relevant papers related to this topic are summarized below:

Automated Bus Generation [6] analysis

This approach involves the use of an automated bus generation tool that requires the designer to follow a set of rules to specify the target bus types to generate. The rules and syntax of the tool were outlined in the Automatic Bus Generation paper [6] and performance of different busses generated by the tool was then analyzed. There were also different bus topologies and methods of connectivity that were demonstrated. The authors also claim that the bus generation tool generates synthesizable Verilog code of the different components of the bus architecture. The flow for generating the bus is given by Figure 2.8:

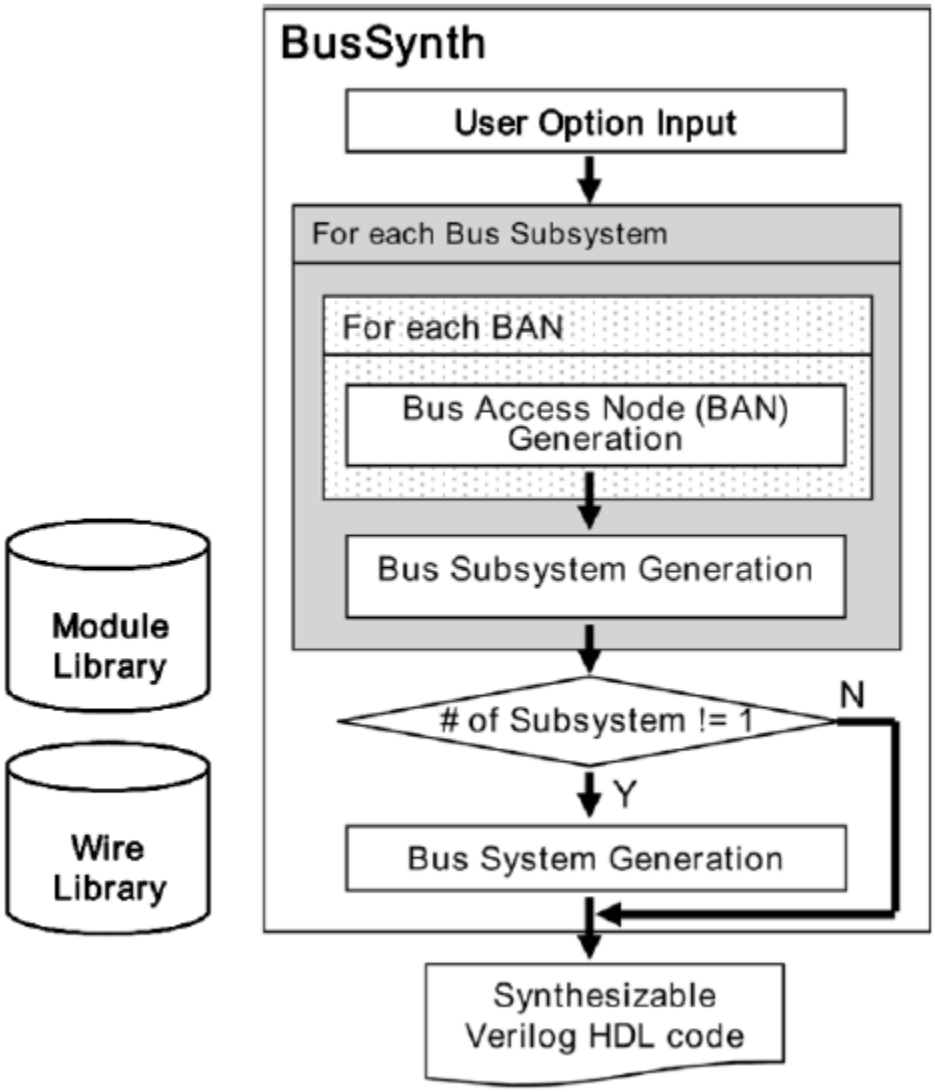

Figure 2.8: Autobus Generation Sequence [6]

Although this automatic generation of busses was able to give the designers the flexibility to generate different busses for different design requirements, it doesn't ease the work of IP reuse since the design team is required to learn a new tool to generate the desired bus architecture. Furthermore, the validation effort required to debug and properly test for a bus whose features can change easily based on inputs to an abstracted tool are not

addressed. The complexity of the BusSynth tool discussed in this paper can also increase exponentially with the variation of processing elements, ports and wires on the bus. It is also not clear how some of the key Verilog built in features such as adders, functions, etc. are to be generated by the BusSynth tool.

Fast Exploration of Parameterized bus Archietecture for SoCs [7] analysis

This paper is mainly concerned with optimization of bus configurations and system interconnects to achieve an optimum bus configuration for a particular bus system. The method advocated was to use a tool called ABC that employs a Genetic Algorithm (GA) to finalize the bus configuration of the system based on an input system configuration and the GA configuration file that needs to be fine-tuned for desired operation and optimal performance of the system. The results showed that an optimum set of parameters for a system configuration was found by the GA that matched the effort of an expert and completed in less time.

Although this approach seems promising, there are several drawbacks. Firstly, typical design engineers will not be familiar with what needs to be done to configure the GA tool for optimum performance such as population size and number of generations. Furthermore, this study mainly focuses on decisions that are typically done once or a few times in the design cycle and at the front end of the cycle; the cost of which can be amortized over the life cycle of the product since it is not repeated throughout the design cycle once completed. This paper also doesn’t directly focus on efforts to establish

modular, configurable bus architecture to solve the problem of IP reuse across different products and bus architectures in industry.

Lotterybus Architecture [8] analysis

This paper's focus was on improving bus protocol methodology to i) improve control of bandwidth allocated to each component within a bus and ii) improve performance of high priority communication on the bus. The paper begins by outlining the existing bus arbitration protocols such as priority-based, round-robin and time-division multiplexing access (TDMA) methods and continues to explain some of their disadvantages with different transactions. For example in the priority based protocol, the lower priority transaction obtains a disproportionately low percentage of the bandwidth compared to the rest of the transactions and TDMA can cause longer latencies for accessing high priority transactions.

The lottery bus uses an arbitration mechanism where a "lottery manager" assigns "tickets" to each bus requester and uses a probabilistic method to decide on a priority based on the number of requests and the range in which the requester falls. This happens after the Lotterybus generates a random number to enable the selection of a requester based on its location within the range of numbers. The performance of the lottery bus was then compared against TDMA and it was shown that lottery bus was able to successfully balance bandwidth between different prioritized tasks and was also able to reduce the latency of higher priority tasks compared to TDMA.

From this paper, it was learnt that the actual bus protocol employed by the architecture should not be limited and allowed to be chosen by the designers who intend to employ the architecture. This flexibility needs to be allowed by the bus spec to enable performance improvement for devices that employ the bus architecture.

Conclusion from Review of Existing Bus Architecture Solutions

After completing the research on existing literature and advancements in the bus architecture requirements for IP reuse and modular design support, it was clear that there was no direct solution to the problems of incorporating IPs or designs in a modular manner for SoC and ASIC products. It was also concluded that the solution would require i) a standardized interconnect ii) an easily understood and portable bus protocol iii) an open arbitration scheme iv) configurable interconnect widths and bus clock speeds v) ability to support a wide range of bus topologies vi) consideration for sideband bus for easier bridging capability and performance improvement vii) defined method for cache coherence support. Based on these criteria it was decided that the Wishbone bus would be a suitable baseline bus architecture for the modular configurable bus architecture due to its already existing support for criteria i, ii, iii, iv and v. The Wishbone bus can then be enhanced with additional capability for sideband bus support and cache coherence support to meet the remaining criteria for the creation of the modular configurable bus architecture.

Chapter 3: The Modular Configurable Bus Architecture Solution

Once the decision was made to use the Wishbone bus as a baseline for the Modular Configurable Bus (MCB), the capabilities and features within Wishbone that would prove useful for MCB were then analyzed in further detail. It was also clear from the review that the Wishbone bus signaling, handshaking protocol, bus transactions, error checking and data organization features were capable of being used as a baseline for MCB. The fact that Wishbone is an open source bus that has well documented rules and examples was also an added advantage. Wishbone also allows for signals outside its specified list to be used with any wishbone compliant IP that would also mean that any MCB additions would maintain its wishbone compliance as long as those signals were documented in the wishbone datasheet used for describing wishbone compliant devices.

Furthermore, there was also readily available Wishbone compliant IPs that could be used, with some modifications, for MCB. These IPs could be found from various sources, although the most useful resource for IPs was found to be maintained by the OpenCores website. In order to assess the feasibility of using these wishbone compliant IPs, one of these IPs was initially tested out for wishbone data transactions using the Modelsim RTL simulator. The availability of the IPs and the Modelsim simulation tool also enabled testing of these IPs for Wishbone and their intended functionality. The majority of the IPs and their tests were written in Verilog.

The MCB features that are required for a complete modular, configurable architecture are also an important consideration. These features and methods are discussed in detail after describing the key wishbone features used by MCB.

Firstly, it is important to look at the various criteria to build a modular configurable bus, the critical areas that Wishbone does not satisfy the criteria and how MCB intends to meet them. The next section covers the criteria for MCB, its key features and how it proposes to achieve improved modularity and configurability over Wishbone.

MCB Key Requirements and Goals:

As discussed in Chapter 2, among the key criteria for a modular, configurable bus for IP reuse are supporting configurable interconnect widths and bus clock speeds, a definition and consideration for sideband bus and a defined method for cache coherence support. The configurable interconnect is required in order to vary the number of IPs on the bus, to vary their data widths, to enable accessing of the different IPs by different bus agents and to arbitrate their accesses. Although Wishbone describes a standard interconnect and protocol, the interconnect module itself is open to various types of implementation schemes that limit its modularity and configurability. Therefore, it is important that the interconnect scheme used is able to be expanded and configured to be used with a variety of IPs and can be easily expanded or reduced if IPs need to be added or removed from the interconnect system. In addition, the arbitration scheme employed also needs to be

easily expandable and configured for the various IPs in the system. Without these features, key interconnect modularity and configurability will not be achieved.

In addition, IPs that are Wishbone compliant also have a set of signals that are not part of the main Wishbone bus and need to communicate to other IPs on the system using non-Wishbone compliant signals. In order to enable these stray signals and still support a modular bus, a sideband bus is required to standardize this signaling for them to be used in a uniform, predictable method. Furthermore, there are several busses reviewed in chapter two that require sideband bus support. If such IPs are to be integrated into a bus system that doesn't support sideband signals, the reuse of these types of IPs will not be possible. Therefore, a sideband bus support is also critical for ease of IP reuse.

Since the need for multi-processor systems is increasing for ASICs and SoCs due to performance and power benefits they provide; a cache coherence scheme is also another critical area that needs to be supported for a modular, configurable bus system.

Figure 3.1 illustrates the basic MCB architecture diagram. The blocks and connections colored in blue show the additional connections and blocks required for configurable interconnect, sideband support and cache coherence support. The Configurable Master/Slave Interconnect Module contains a configurable arbiter that can vary in width and size as the number of IPs increase or reduce. The SB Ctrl blocks denote the sideband controllers for sideband signaling. There is also a cache controller shown in this diagram that communicates cache coherence signaling for a multi processor scheme given by M0 and M1.

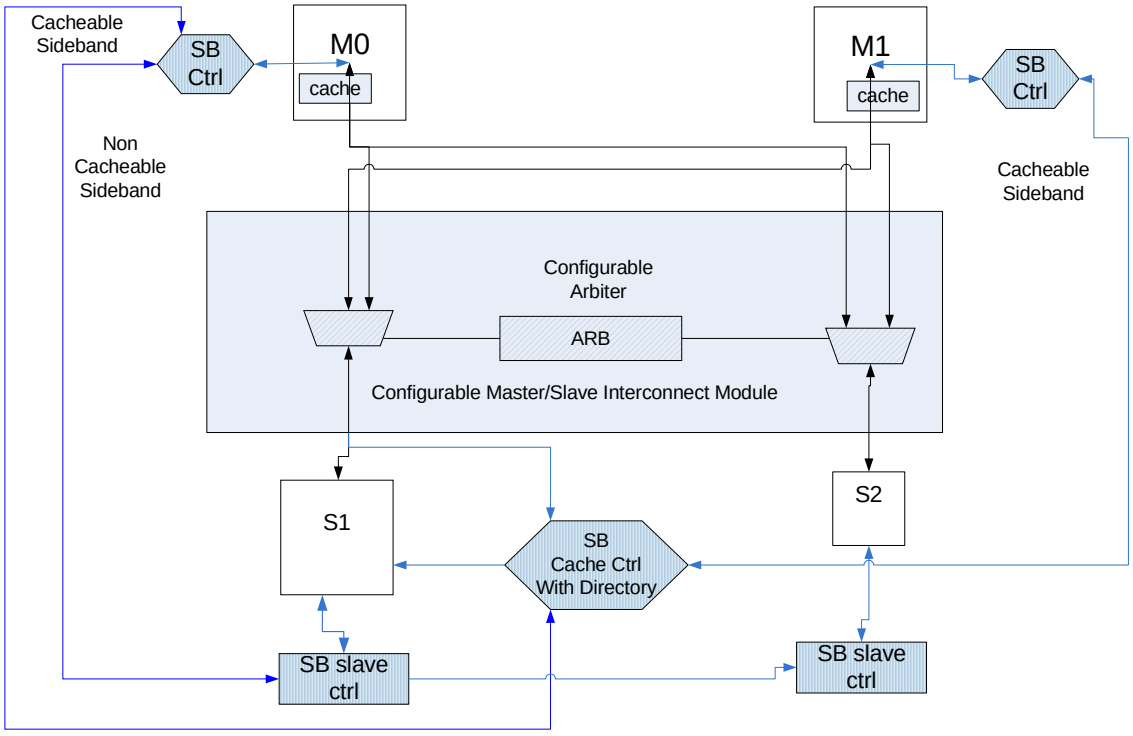

Figure 3.1: MCB Architecture Diagram

The following section describes the key Wishbone features that were used to build MCB.

Key Wishbone Features Leveraged by MCB:

The complete details and rules of the Wishbone bus are outlined in the Wishbone SoC Architecture Specification Document [1].

Wishbone's Signal Description

Wishbone bus agents that are interconnected through a system interconnect module consist of one or more Wishbone Master(s) and one or more Wishbone Slave(s). Masters initiate Wishbone transactions while slaves respond to the master initiated transactions.

Signal names ending with "_O" indicate output while "_I" indicates inputs.

Signal names ending with open and closed parentheses "()" indicates a greater than one bit width. The key wishbone signals used by MCB are described in Table 3.1.

Signal Name	Residing in	Description
CLK_O	Interconnect	System clock signal generated by a system interconnect module to provide the clock signal for all the Wishbone agents connected on the bus
RST_O	Interconnect	Reset signal generated by a system interconnect module
CLK_I	Master & Slave	System clock input signal for internal clocking of synchronous logic
RST_I	Master & Slave	Reset indicator for Wishbone logic
DAT_I ()	Master & Slave	Data bus input to masters or slaves, limited to 64 bits in Wishbone, not limited in MCB
DAT_O ()	Master & Slave	Data bus output from masters or slaves, again no limitation on MCB
TGD_I ()	Master & Slave	Carries tag information related to the data input bus (DAT_I). This information could be related to types of data transfer, error correction codes, or any user defined function
TGD_O ()	Master & Slave	Carries tag information related to the data output bus (DAT_O)
ACK_I	Master	Acknowledgement input to indicate a termination of a bus cycle
ADR_O ()	Master	Address bus output from masters to indicate targeted slave address
CYC_O	Master	Indicates valid bus cycle is in progress and is asserted for the duration of a bus cycle
STB_O	Master	Strobe output to indicate a valid data transfer, also used to qualify the master's output signals
WE_O	Master	Signal indicating a write cycle when asserted and a read signal when de-asserted
ERR_I	Master	Error input indicator to signal an abnormal cycle termination
RTY_I	Master	Indicates that the targeted bus agent is unable to accept or transfer data, data should be resent during this condition
SEL_O ()	Master	Select bus that indicates which data bits on the bus are valid on the data bus. Size of select bus depends on the granularity of data accesses. For example if an 8-bit granularity is used, then a 32 bit bus will have four SEL_O bits, one bit for each byte.
TAGA_O ()	Master	Address tag that stores information about the address such as memory, I/O, config, etc.
ACK_O	Slave	Acknowledgement output to indicate a termination of a bus cycle
ADR_I ()	Slave	Address bus input from masters to identify targeted slave address

CYC_I	Slave	Indicates a valid bus cycle is in progress, asserted for the duration of a bus cycle. During block transfers that have multiple data transfers in one bus cycle, this signal remains asserted during all the data transfers
STB_I	Slave	Strobe input to indicate a valid data transfer from master
WE_I	Slave	Indicates a write cycle when asserted and a read signal when de-asserted
ERR_O	Slave	Error indicator to signal an abnormal cycle termination
RTY_O	Slave	Indicates that the targeted bus agent is unable to accept or transfer data, data should be resent during this condition
SEL_I ()	Slave	Select bus that indicates which data bits on the bus are valid on the data bus.
TAGA_I ()	Slave	Address tag that stores information about the address

Table 3.1: Wishbone Signals Used by MCB

In Table 3.1, the first column describes the signal name, while the second column identifies where these signals reside and the third column describes the signals. It is important to note from Table 3.1 that output signals are clocked on the rising edge of the source clock while input signals are required to be stable before the rising edge of the source clock. Although the Wishbone spec specifies a DAT_O() and DAT_I () have a limit of 64 bits, MCB does not limit this bus for wider bus sizes to be supported.

Wishbone's Handshaking Method

As shown in Figure 3.2 [1] below, all Wishbone bus cycles use the STB_O and ACK_I relationship as a handshaking protocol between masters and slaves. A Wishbone master asserts the STB_O signal when it transfers data and this signal remains asserted until the slave terminates it with either one of the ACK_I, ERR_I or RTY_I signals. STB_O will terminate at the rising edge of the wishbone clock. There are additional requirements for handshaking for different clock domains that will be discussed in the "Features Added to Wishbone to Build MCB" section.

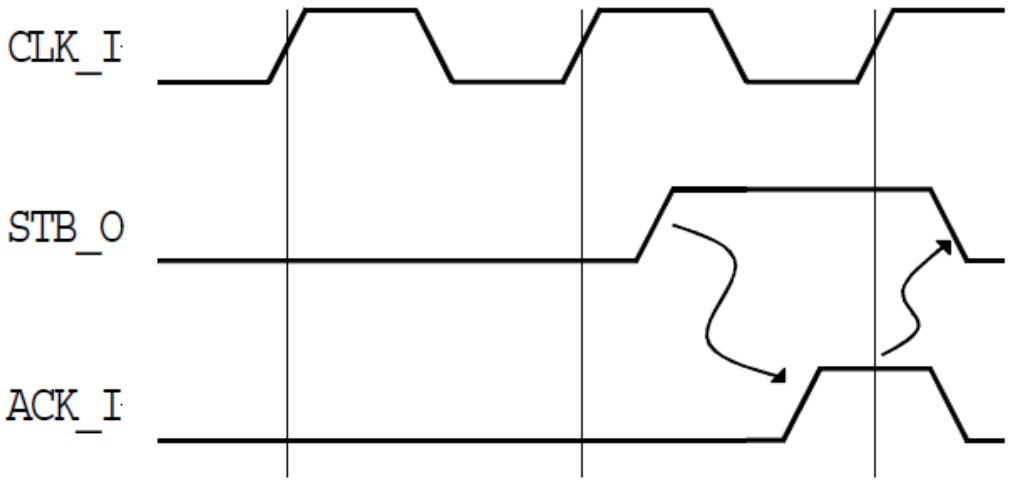

Figure 3.2: Wishbone Handshaking Protocol [1]

Wishbone Basic Read and Write cycles

The basic Wishbone read cycle will perform one data read for each bus cycle. When the Wishbone master is ready to read data from a slave, it asserts STB_O, sends the ADDR_O, SEL_O and sets WE_O to a low logic level to indicate a read transaction. These master output signals are synchronous to the first edge of CLK_I shown in Figure 3.3[1]. The Wishbone slave then decodes the data from master and asserts the ACK_I and presents valid read data on the DAT_I. Wishbone masters will latch inputs driven by slaves on CLK_I edge 1 given by Figure 3.3[1].

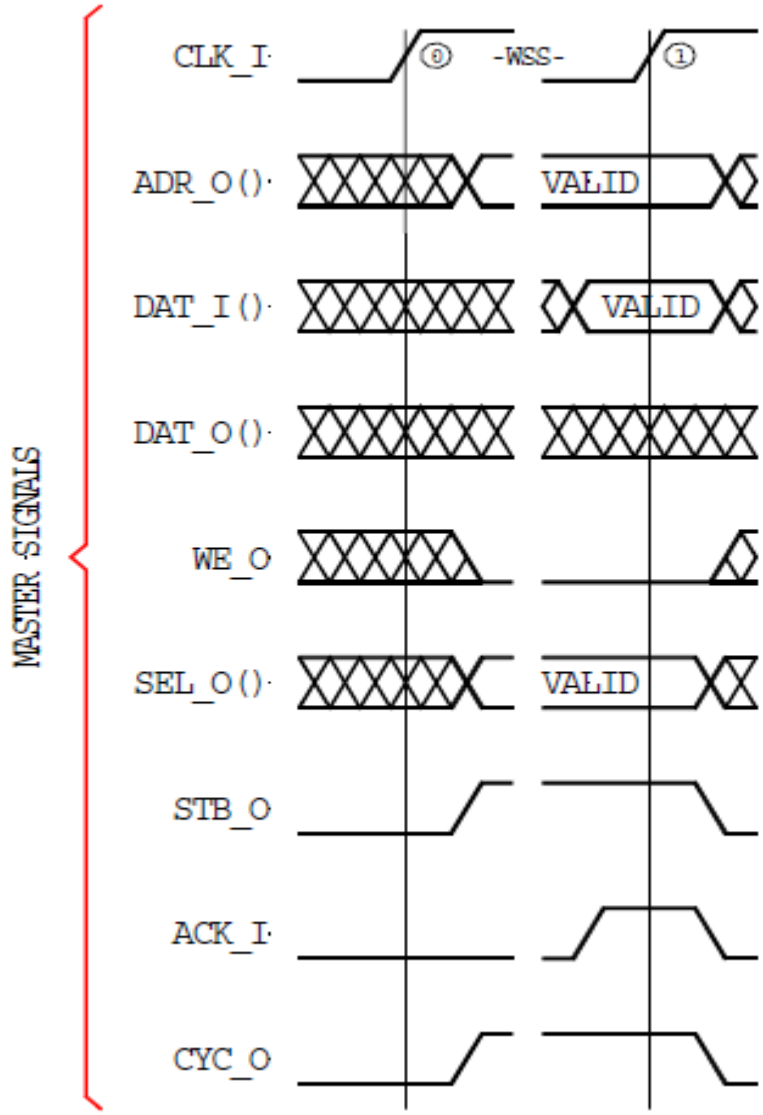

Figure 3.3: Wishbone Basic Read Cycle [1]

The basic Wishbone write cycle will perform one data write for each bus cycle. When the Wishbone master is ready to write data to a slave, it asserts STB_O, sends the ADDR_O, DAT_O, SEL_O and sets WE_O to high level to indicate a write transaction. These master output signals are synchronous to CLK_I edge 0 given in Figure 3.4[1].

The Wishbone slave then decodes the data from the master and latches the data given by DAT_O. The slave then asserts the ACK_I in response to STB_O to indicate data has been received. Wishbone slaves will latch DAT_O while masters will de-assert STB_O to indicate the end of write cycle on CLK_I edge 1 given by Figure 3.4[1]. The slave will then de-assert ACK_I in response to STB_O de-assertion.

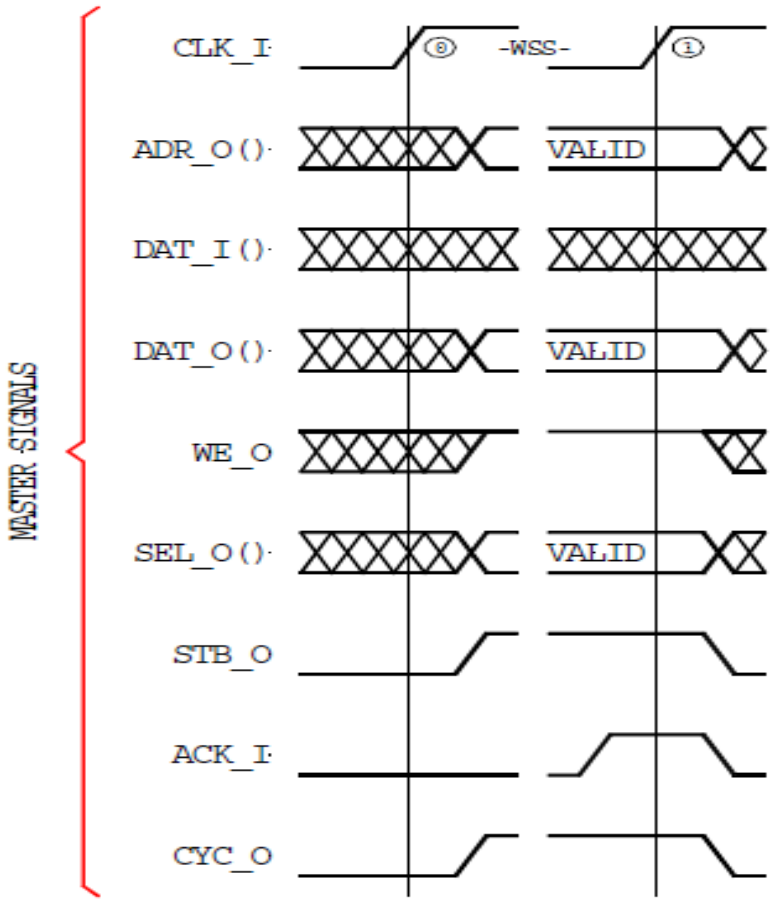

Figure 3.4: Wishbone Basic Write Cycle [1]

Wishbone Block Read and Writes

Block transfers allow for multiple data transfers on a single bus cycle. Block transfers perform read and write similar to basic read and write cycles with the difference being multiple read and write transfers can be combined into a single block. This allows for the

capability to perform data transfers at every clock cycle. To differentiate between basic reads/writes and block read/writes, address or data tags can be used.

Figure 3.5 [1] shows an example of a Wishbone block read transaction. On the first CLK_I cycle the master sends ADDR_O, SEL_O, asserts CYC_O to indicate start of a bus cycle and de-asserts WE_O to indicate a read. Before the edge of the second cycle, the slave decodes the master driven signals, asserts ACK_I and sends the read data through DAT_I. The Wishbone master will then latch DAT_I on the second CLK_I edge. The master then sends the subsequent address information during the second CLK_I cycle. The slave will then decode the new address and continues to assert ACK_I. The slave then sends the data associated with this transfer before the third CLK_I edge.

Figure 3.5 [1] also shows an example of cases where the STB_O and ACK_O are de-asserted during the block read cycle. These are due to wait states induced by masters and slaves. For example, if the master needs to induce a wait state, the master de-asserts STB_O and if a slave needs to induce a wait state, the slave de-asserts ACK_I.

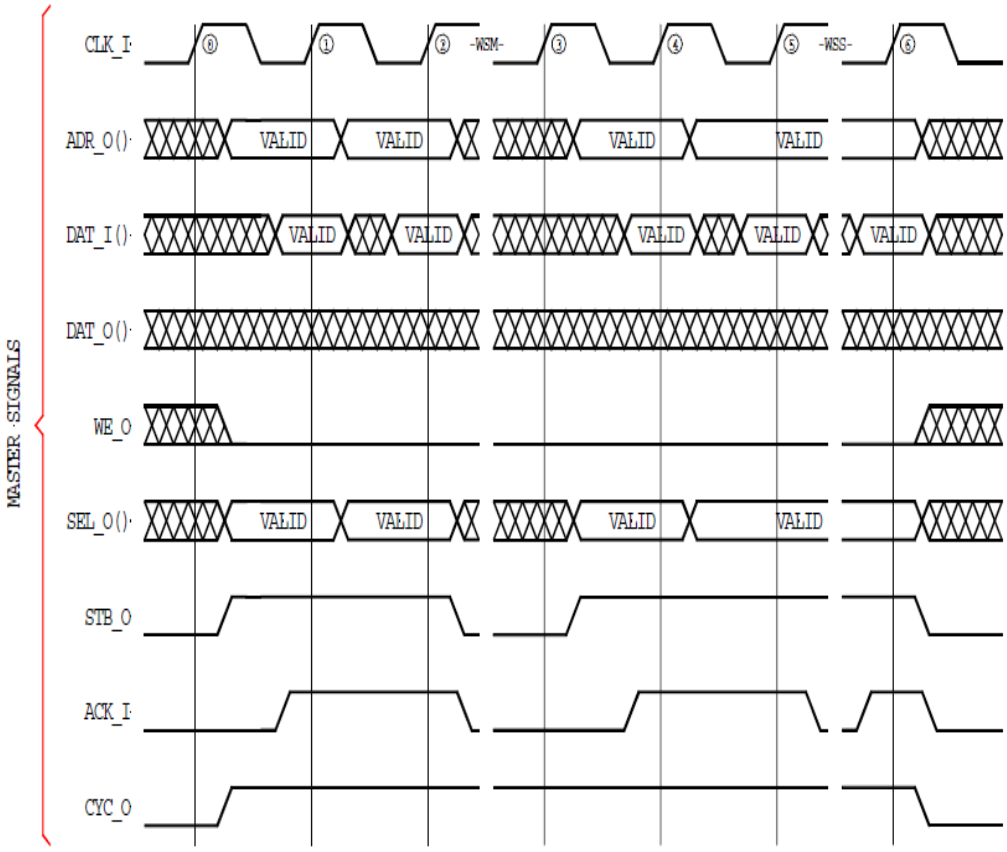

Figure 3.5: Wishbone Block Read Cycle [1]

For block writes, the Wishbone master sends ADDR_O, DAT_O, SEL_O information, asserts CYC_O to indicate start of a bus cycle and asserts WE_O to indicate a write transaction. Before the rising edge of the second CLK_I cycle the slave decodes the master driven signals and asserts ACK_I. The Wishbone slave will then latch DAT_O on the second CLK_I edge. The master then sends the subsequent address information during the second CLK_I cycle. The slave will then decode the address and continues to assert ACK_I. The slave then latches the data associated with this transfer on the third CLK_I edge. These transfers are shown in Figure 3.6[1].

Figure 3.6 [1] also shows an example of cases where the STB_O and ACK_O are de-asserted during the block write cycle. These are due to wait states induced by masters and slaves. For example, if the master needs to induce a wait state, the master de-asserts STB_O and if a slave needs to induce a wait state, the slave de-asserts ACK_I.

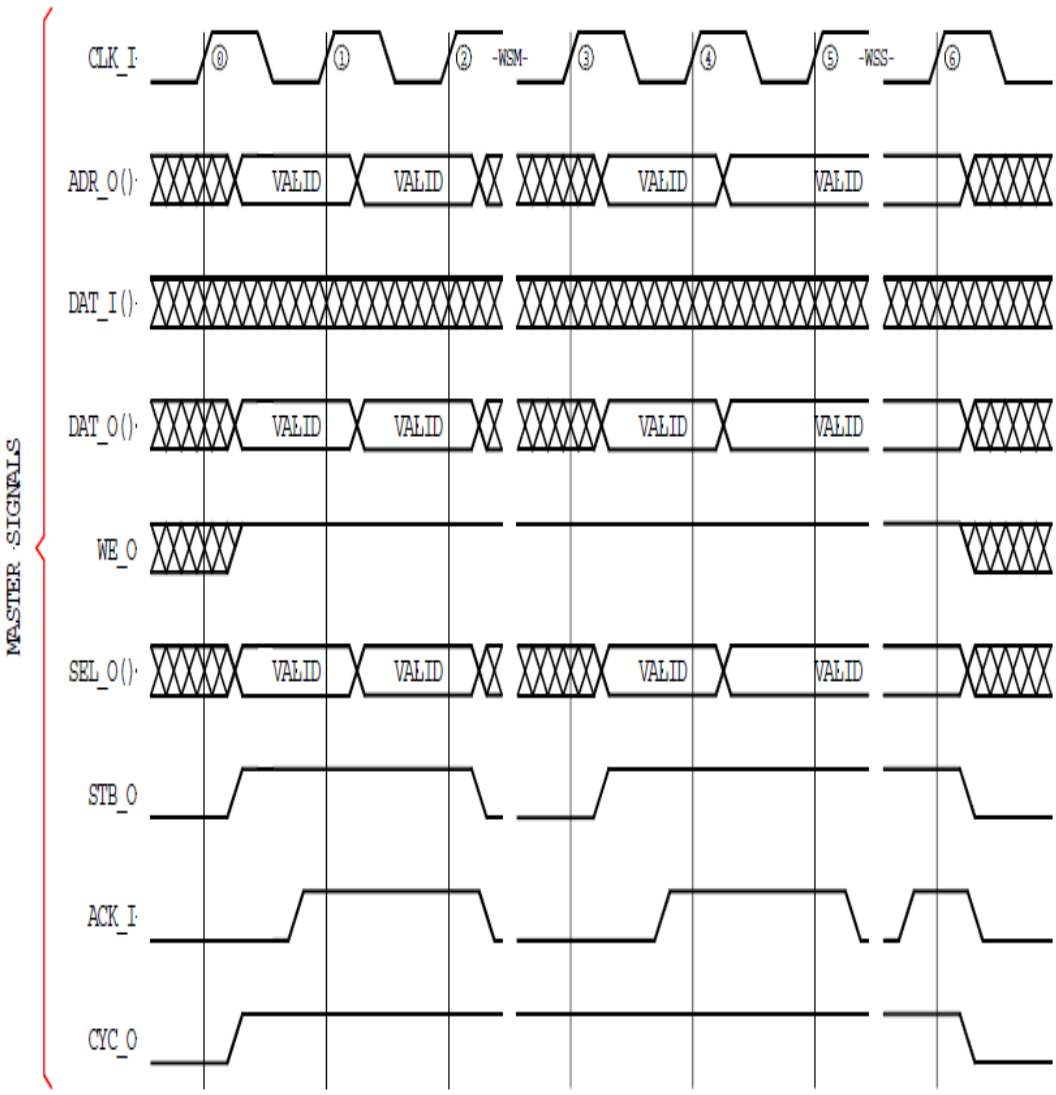

Figure 3.6: Wishbone Block Write Cycle [1]

Features Added to Wishbone to Build MCB:

This section describes the details of the features that were added to the existing Wishbone feature set in order to better support a variety of IP requirements for improved IP reuse, to improve bus performance and to increase configurability, flexibility and modularity of the existing Wishbone bus. These features are what differentiate MCB from Wishbone.

MCB Sideband Bus

Sideband signals typically exist in a system for transfer of signals that are not suitable for the primary bus due to their different usage model, different clock speed and for some cases the need to have a direct connection with the intended targeted logic. From the analysis conducted in Chapter 2, it was found that a number of the bus architectures surveyed supported a sideband bus for configuration, power management and test data transfers. Therefore, in order to build a bus solution for improved IP reuse and better performance, a sideband bus would not only be beneficial but also necessary.

The MCB sideband bus signals are described in Table 3.2 below. The MCB sideband bus was architected to serve the below functions:

1. Allow for easier bridging to busses that also have sideband channels such as IBM core connect.
2. Improved scalability for signals that would normally not plug into the main bus architectures such as power management, configuration and Design for Test (DFT) signals.
3. Improved bus performance since some of these signals, instead of being sent on main bus can now be sent through sideband bus.

The sideband bus on MCB is optional and would typically be used for bridging to external busses with sideband channels or for IPs that send out sideband signals not meant for main bus.

Signal name	Sideband Master Direction	Sideband Slave Direction	Signal Width: () denotes recommended width	Description
SB_clk	I	I	1	sideband clock
SB_rst	I	I	1	sideband reset
SB_Req	O	I	1	Request for Master to slave synchronization
SB_Ack	I	O	1	Ack for Master to slave synchronization
SB_Read	O	I	1	Read request on SB master
SB_Write	O	I	1	Write request on SB master
SB_Tmout_Hlt	I	O	1	Wait to holdoff timer on main bus
SB_Addr	O	I	configurable (32)	master's targeted address
SB_Tag	O	I	configurable (8)	Tag to determine type of transaction
SB_Datain	I	O	configurable (32)	input data into master
SB_Dataout	O	I	configurable (32)	output data from master

Table 3.2: MCB Sideband Signal Description

A brief description of these signals is given by the “Description” column of Table 3.2 and examples of the signal behavior during read and write transactions are described along with the timing diagrams in Figures 3.7 and 3.8 respectively.

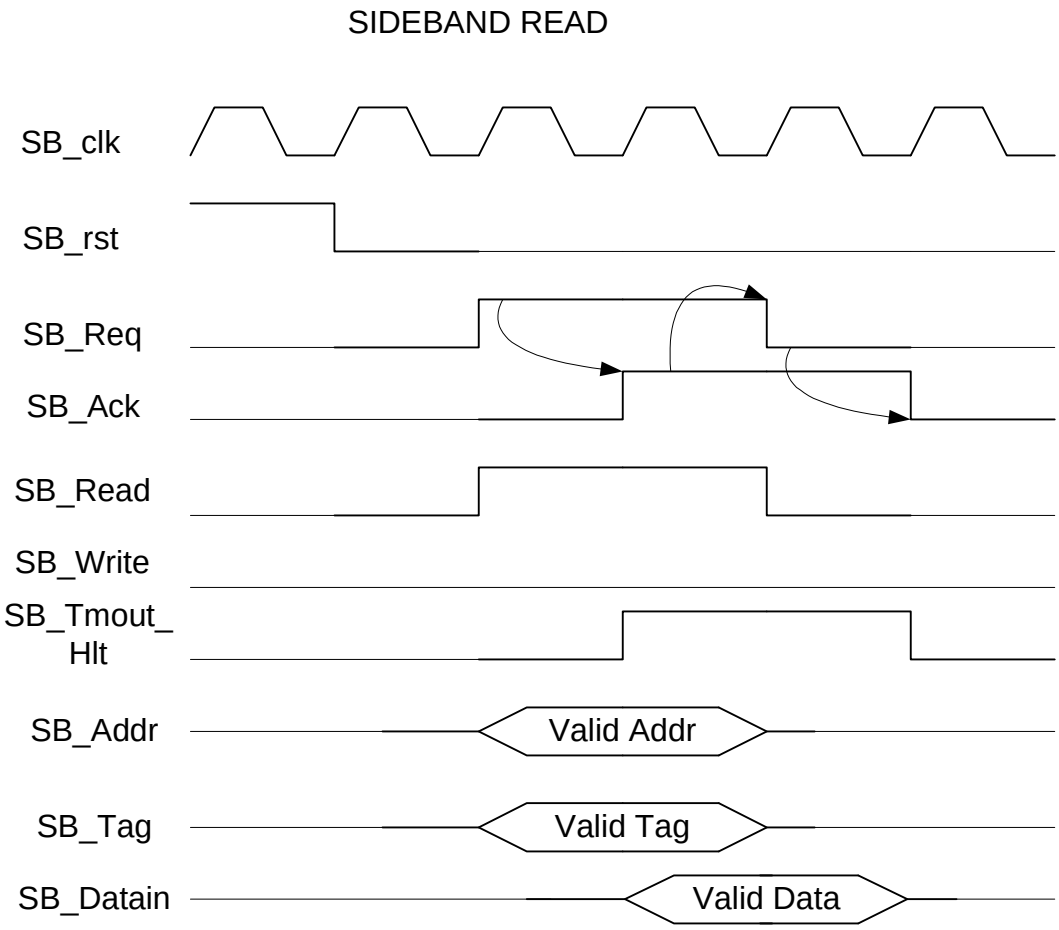

Figure 3.7: Sideband Read Transaction

Figure 3.7 depicts the sideband read phase timing behavior of the MCB bus with the defined sideband signals. The sideband read signal (SB_Read) assertion will also assert a sideband request signal (SB_Req). The sideband address (SB_Addr) will be sent from the sideband master to the sideband slave and an optional sideband tag (SB_Tag) bus can also be sent. The SB_Tag signal can be used to describe the type of data packet expected to be read. Once a request is made, a sideband acknowledge indicator (SB_Ack) will be monitored to de-assert SB_Req. The sideband Master's input data (SB_Datain) is received by the sideband master during the SB_Req phase and latched on the subsequent

Sb_clk rising edge. The SB_Req and SB_Read will then de-assert after SB_Ack is received from the sideband slave. There is also a timeout halt signal (SB_TMout_Hlt) to indicate to primary bus masters that the disabling of any timeout counters on the primary bus needs to take place. This would prove useful when sideband is running and primary bus is idle to avoid timing out during sideband transactions.

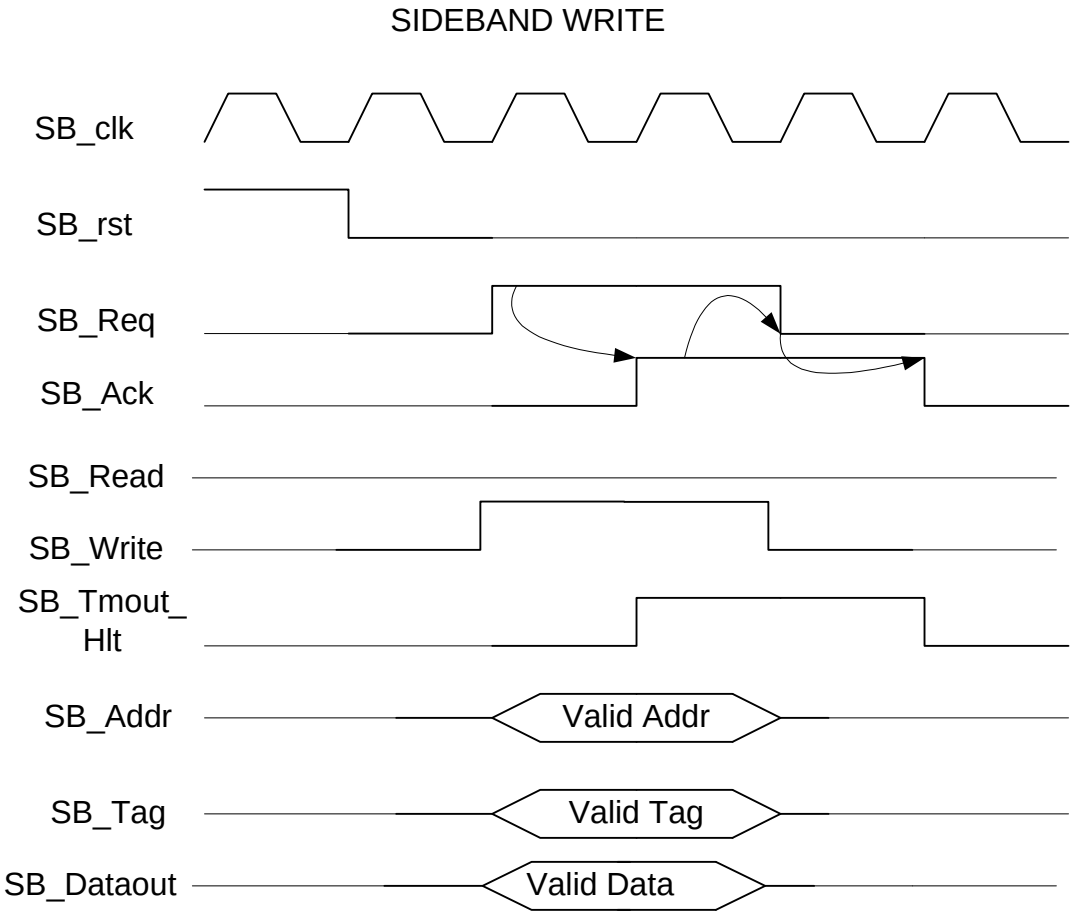

Figure 3.8: Sideband Write Transaction

Figure 3.8 depicts the sideband write phase timing behavior of the MCB bus with the defined sideband signals. The sideband write signal (SB_Write) assertion will also assert

a sideband request signal (SB_Req). The sideband address (SB_Addr) and output data (SD_Dataout) will be sent from the sideband master to the sideband slave and an optional sideband tag (SB_Tag) bus can also be sent. Once a request is made, a sideband acknowledge (SB_Ack) will be monitored to de-assert SB_Req. The sideband Master's output data (SB_Dataout) will be expected by the sideband slave during the duration of the SB_Req phase and latched on the subsequent Sb_clk rising edge. The SB_Req and SB_Write will then de-assert after SB_Ack is received from the sideband slave. There is also a timeout halt signal (SB_TMout_Hlt) to indicate to primary bus masters that the disabling of any timeout counters on the primary bus needs to take place as described in the read phase.

MCB Cache Coherence Support

In a multi-processor system, cache coherence support is critical to maintain data coherency between all processors containing caches and main memory. While implementing and architecting the sideband bus for the MCB bus requirements, it was discovered that the sideband bus could also be leveraged for the purpose of keeping track of data coherence between any caches that reside on primary bus masters and the main memory module. This is because the sideband tag signals (Sb_Tag) can be used to communicate the relevant cache transactions between the memory module and cache coherent masters while the SB_Add and SB_Dataout can be used to communicate the relevant cache addresses and data for cache writeback if necessary.

Some points to note about the cache coherence validation strategy to prove this capability are described below:

- For the purpose of the validation of cache coherence support for MCB, a MESI [9] protocol scheme was employed. Any of the other major cache coherence protocols can also be supported.
- Although in these examples only two masters are shown, the MCB cache directory was designed to support a parameterized number of cache coherent masters.
- A cache directory was modeled where the cache addresses can directly be used to index the directory to obtain the cache status information. The directory was not optimized to allow for complex address translation to compress its size since that capability is beyond the scope of this research. The cache directory was tested by only accessing the addresses within the ranges that it supports for cache coherence testing on MCB.
- Clock speed of cache coherence sideband signals is required to match the bus clock speed of the cache coherent modules. Any synchronization between internal memory accesses and the cache coherence signaling is handled within the respective cache coherent modules.

Examples of the expected usage model with common cache coherence scenarios with the MESI protocol are described as below.

1. Master 0 (M0) contains data in cache in Read state. Master 1 (M1) then reads same memory location from main memory. The behavior of the cache coherent masters and the cache directory is summarized in Table 3.3.

Trans	Master 0 Memory Transaction	SB Coherence Tag	M0 MESI State	M0 Cache Coherence Sideband transaction	Master 1 memory Transaction	SB Coherence Tag	M1 MESI State	M1 Sideband bus transaction	Cache directory SB Transaction	Cache Dir Status
1	MEM RD	N/A	E	N/A	N/A	N/A	I	N/A	N/A	RD, I
2	N/A	RDSHR	S	SBRD	MEM RD	RDSHR	S	SBRD	SBWR for RDSHR indication	RD/RD

Table 3.3: Cache Coherent Masters and Cache Directory Transactions example 1

In Table 3.3, the first transaction, given by the first row of the "Trans" column, is the initial memory read on M0 for a particular address. The "Master 0 Memory Transaction" field describes this as "MEM_RD". There are no sideband transactions required for this access although the cache directory is cognizant of this accesses by reading requests to memory controller and updates the directory for the M0 field for the accessed address. This is shown by the "Cache Dir Status" column in Table 3.3 where M 0's status is stored as a Read (RD). M1's directory status remains invalid which is given by the "I" in the "Cache Dir Status" column separated by a comma.

The following transaction given by second row of the "Trans" column shows the behavior when M1 reads the same address previously read by M0, as shown by "MEM_RD" on the "Master 0 Memory Transaction" column in Table 3.3 . When this is detected the cache directory transaction is a sideband write to both M0 and M1 to

indicate that both masters are sharing the data for the accessed address, given by the RDSHR indicator. As a result, both Masters are receiving sideband data given by the SBRD in their respective sideband transaction columns. This way, the cache coherence states can be maintained by both cache coherent masters as given by the "MESI State" columns in Table 3.3.

2. This specifies the scenario where Master 0 (M0) and Master 1 (M1) initially contain data from same address in Shared state. Master 0 then writes data to this location in its cache. The behavior of the cache coherent masters and the cache directory is summarized in Table 3.4.

Trans	Master 0 Memory Transaction	SB Coherence Tag	M0 MESI State	M0 Cache Coherence Sideband transaction	Master 1 memory Transaction	SB Coherence Tag	M1 MESI State	M1 Sideband bus transaction	Cache directory SB Transaction	Cache Dir Status
1	N/A	RDSHR	S	SBRD	MEM RD	RDSHR	S	SBRD	SBWR for RDSHR indication	RD/RD
2	CACHE WR	CACHE_WR	M	SBWR	N/A	INV	I	SBRD	SBWR for M1 invalidation, SBRD for M1 Cache status update	WR/I

Table 3.4: Cache Coherent Masters and Cache Directory Transactions example 2

As described in Table 3.4 above, when M0 writes to cache (as shown in Trans column 2), M0 also sends a sideband tag and address to indicate a cache write is taking place for the

address. The cache directory will update its state for M0 to a Write (WR) and sends a sideband transaction to M1 to indicate that the address written by M0 needs to be invalidated in M1's cache.

3. Master 0 (M0) contains data in cache in Write state. Master 1 (M1) then attempts to read same memory location from main memory. The behavior of the cache coherent masters and the cache directory is summarized in Table 3.5.

Trans	Master 0 Memory Transaction	SB Coherence Tag	M0 MESI State	M0 Cache Coherence Sideband transaction	Master 1 memory Transaction	SB Coherence Tag	M1 MESI State	M1 Sideband bus transaction	Cache directory SB Transaction	Cache Dir Status
1	CACHE WR	CACHE_WR	M	SBWR	N/A	INV	I	SBRD	SBWR for M1 invalidation, SBRD for M1 Cache status update	WR/I
2	N/A	WRBSHR	S	SBRD followd by SBWR	MEM RD	RDSHR	S	SBRD	SBWR to M0 to indicate WRBKSHR, SBWR to M1 and Main Mem for WRBSHR DATA	RD/RD

Table 3.5: Cache Coherent Masters and Cache Directory Transactions example 3

For this example, data that is in the Modified state in M0's cache is read by M1. The cache directory sends a writeback shared state indicator (WBSHR) to M0 to indicate that data writeback needs to take place. M0 then sends the WBSHR information to the directory controller in order for it to update the main memory and M1 with the modified

data. The directory will also update its information on the directory status of M0 and M1. The cache directory controller will also send an indicator through sideband tag to M1 to update its cache for this address to a shared state.

MCB Interconnect and Arbitration Requirements

In order to build a modular, configurable bus architecture, the interconnect module and arbitration scheme in particular need to be as flexible and easily configurable as possible. This is required in order to easily add new IPs onto the interconnect bus as system requirements can vary from project to project. Furthermore since multiple masters and slaves would reside on a given interconnect module, their respective accesses need to be arbitrated and the arbitration logic needs to also be easily configurable to a parameterized number of slaves.

A parameterized interconnect module was built using a Verilog RTL model by enabling inputs and output data widths to be controlled by Verilog header files. Due to a limitation in Verilog where input and output ports to a particular module need to be one dimensional, unlike internal registers that can be made up of multi-dimensional arrays, there was added complexity to the design of a parameterized interconnect module. For example, if a data bus from M0 in Figure 3.9 needs to be distinguished from M1, there would typically be two separate connections to the interconnect module that are uniquely named. This means that the addition of more masters and slaves on the bus would need changes to the interconnect module's RTL to support it. This would not be feasible for building the interconnect module for MCB. To overcome this, a single flattened bus

connecting all bus inputs for a particular signal type in a set order was used to connect with the interconnect module on the interface layer. Internally, this one dimensional port will be translated to internal registers that hold the unique bus information for each bus agent connected to the interconnect module. Using parameters for the bus width, number of masters and number of slaves, this information can be used to separate the information for interconnect outputs and combine the information for interconnect inputs. This allowed the MCB interconnect to continue adding IPs to the bus system without needing to update its RTL. All that is needed is to make the connection in the test bench and set the parameters for the added IPs. The final interconnect scheme used will be discussed further in Chapter 4.

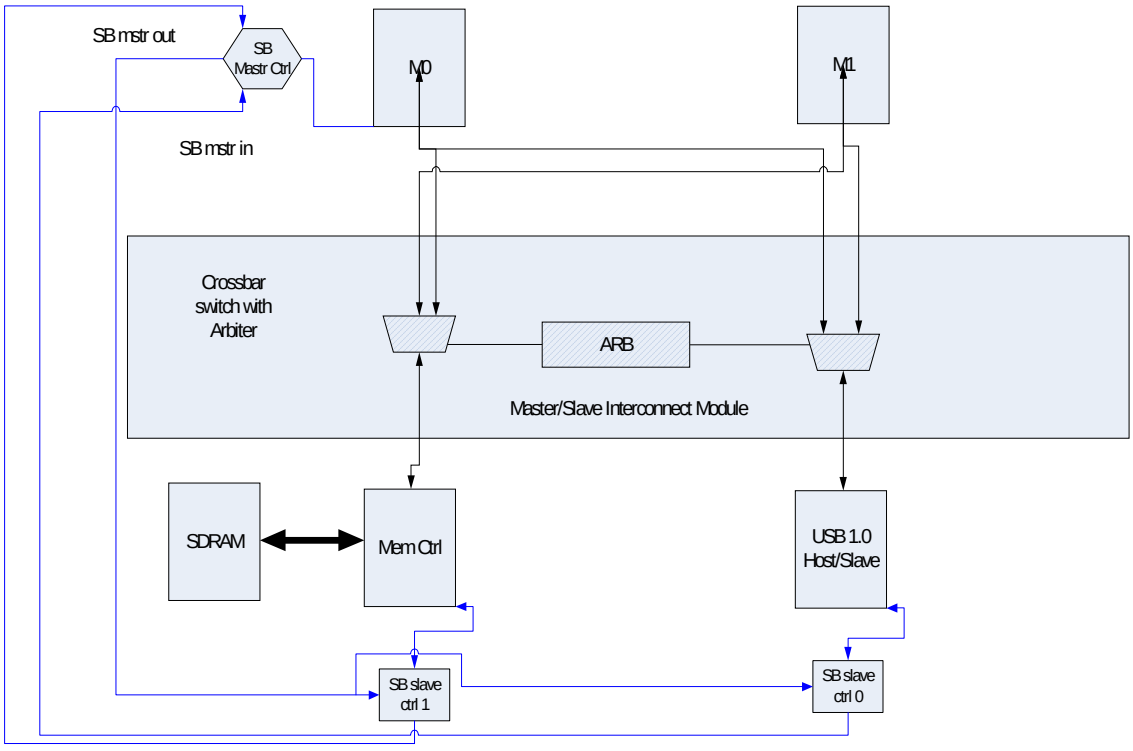

Figure 3.9: Example MCB Interconnect Module

Similarly, a scalable arbiter that supports multiple masters and slaves also needed to be designed. A simple round robin arbiter that supported a parameterized number of requests and grants was designed for this purpose. The grant priority for each requester would change depending on the last granted requester in the request list. These request and grant lists are parameterized in order to vary their sizes as more IPs are added to the system. It is also important to note that this module's coding style supported configurable sized Verilog registers and was accessible using loop parameters in order to determine the next requester on the list to be granted.

It is also important to note that just like Wishbone, the arbitration scheme for MCB is not fixed.

MCB Synchronization Support for Multiple Clock Domains

In order to support interconnect modules and IPs targeted for different clock frequencies on Wishbone compliant IPs, a workable synchronization scheme also needed to be developed in order to avoid missing requests from higher speed clock domains. In order to do this, the wishbone STB_O and ACK_I handshaking needed to be updated for clock crossing synchronization cases. This was particularly necessary for cases where MCB uses a primary bus of a high speed clock and connects to a secondary bus of a lower speed clock. In order to do this, the additional requirement where STB_O cannot assert until the ACK_I is de-asserted needed to be employed. This would effectively synchronize data and address transfers between the different clock domains.

See Figure 3.10 below for details. CLK_A denotes clock domain A, STB_A is strobe in clock domain A while CLK_B and ACK_B are the clock and acknowledge indicator for clock domain B.

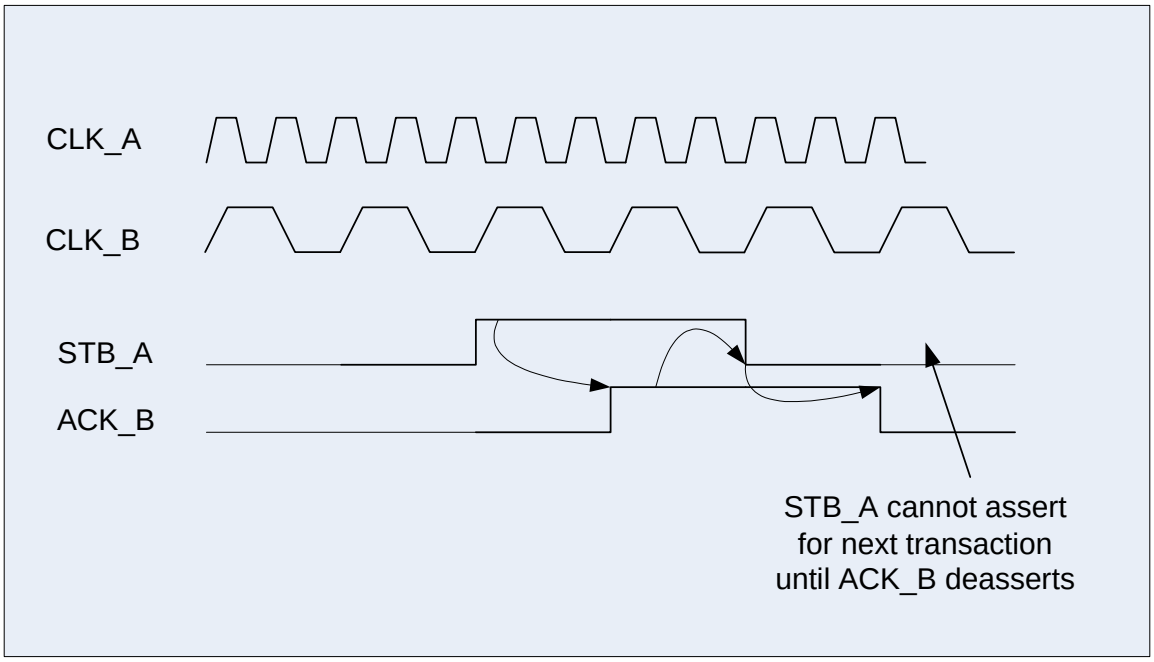

Figure 3.10: MCB Clock Domain Crossing Method

Chapter 4: IP Blocks Selected For Reuse and Architecture of Simulated System

To quantify the effectiveness of the Modular Configurable Bus architecture, a simulation of the RTL containing a number of IP blocks connected through interconnect module(s) was required. It was also important to select relevant IPs that were stable and for most of the cases, compliant to the Wishbone bus spec. This chapter covers the IPs used in the simulated system, the interconnect modules used to build these IPs into a simulated system, the additions made to the IPs for MCB, as well as the criteria used to decide on the interconnect scheme.

Description of IP Cores Inherited for Reuse:

All the IPs used were obtained by subscribing to the OpenCores, open source projects website. The OpenCores database contains many designed and published freely available hardware IP that were suitable for the purpose of building the simulated system. The selection criteria for the IPs were i) requires to be coded in Verilog ii) should be well documented iii) included tests for RTL simulation iv) is Wishbone compliant, unless the IP is intended to test for bridging to the Wishbone or MCB bus v) was relevant for ASIC or SoC systems.

USBHostSlave IP Core

Since USB is a widely used peripheral bus interface and supports a wide range of devices, it was important to simulate the system with a USB host and device capability. The USBHostSlave IP core can be configured as a USB host or a USB device. Both

instances were enabled for the MCB simulated system implementation. The USBHostSlave IP block consists of a serial interface engine, a multiplexer for Host and Slave selection block (HostSlaveMux), USB Host and Slave controllers and a Wishbone bus interface. These blocks are shown in Figure 4.1. The serial interface engine supports the USB 1.1 [10] protocol layer and contains the USB transmit and receive physical layer logic for connection to other USB hosts or devices. The HostSlaveMux allows the USB host controller and slave to share access to the serial interface engine. The USB Host and Slave controllers support all USB 1.1 [10] protocol layer specifications and transaction types such as bulk, setup, interrupt and isochronous transactions. The Wishbone bus interface enables the USB Host and Slave controllers to communicate to a Wishbone Master.

The tests for the USBHostSlave IP Core include reading the host and slave version numbers, configuration register reads and writes, configuration for different USB 1.1 [10] speeds, checking and cancellation of USB interrupts, multiple writes to USB transmitter and reads from USB receiver through receive FIFO.

The USBHostSlave IP core also includes interface support for AMBA AHB bus and IBM's OPB bus. These were useful for testing for bridging of these versions of USB with the MCB bus. Additionally the OPB version of the USBHostSlave IP was modified to support the DCR bus that was also used to assess IP reuse capability with MCB sideband bus.

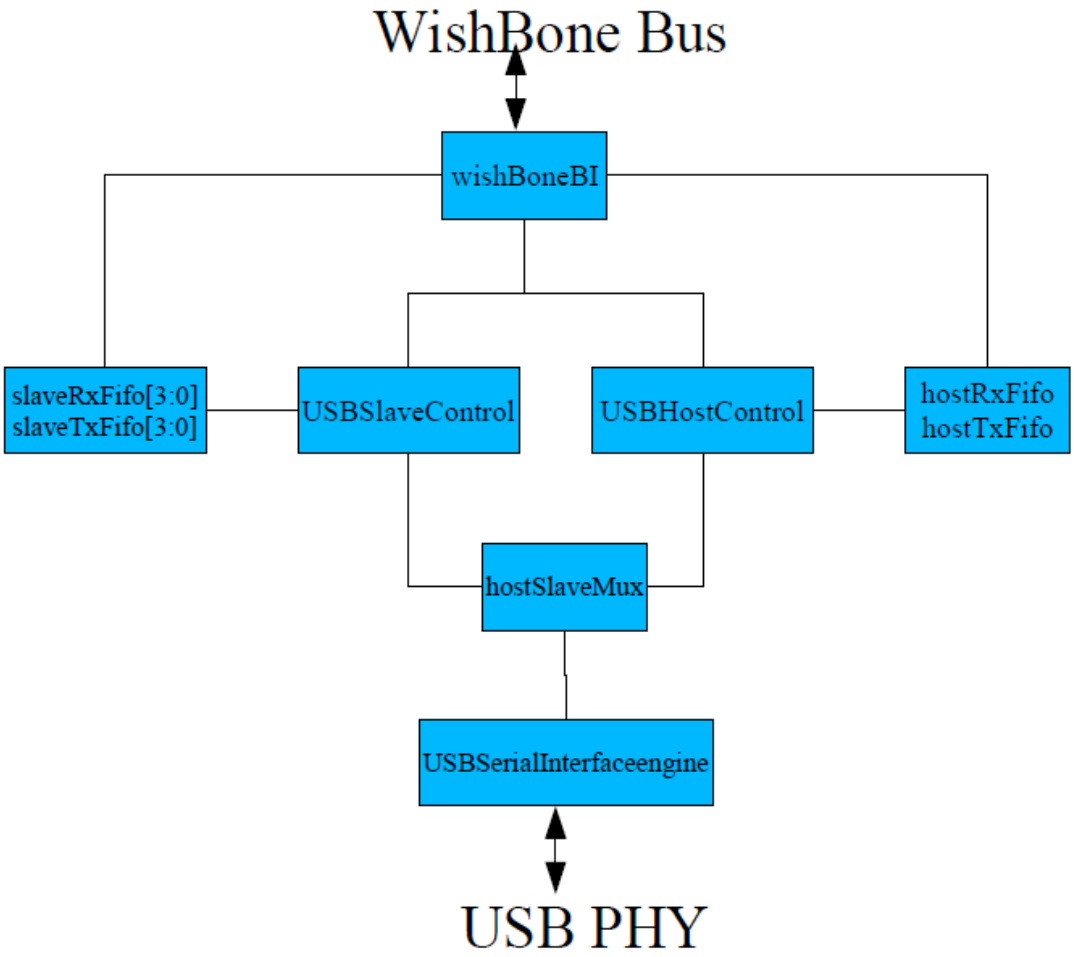

Figure 4.1: USB USBHostSlave IP Core Block Diagram [11]

Memory Controller IP Core

In order for the system to access main memory, a memory controller along with RAM blocks were required. This was achieved by implementing a wishbone compliant Memory Controller IP Core that supports SDRAM, SSRAM, FLASH among other devices. There is no DDR support. The overall architecture of the Memory Controller IP Core is shown in figure 4.2.

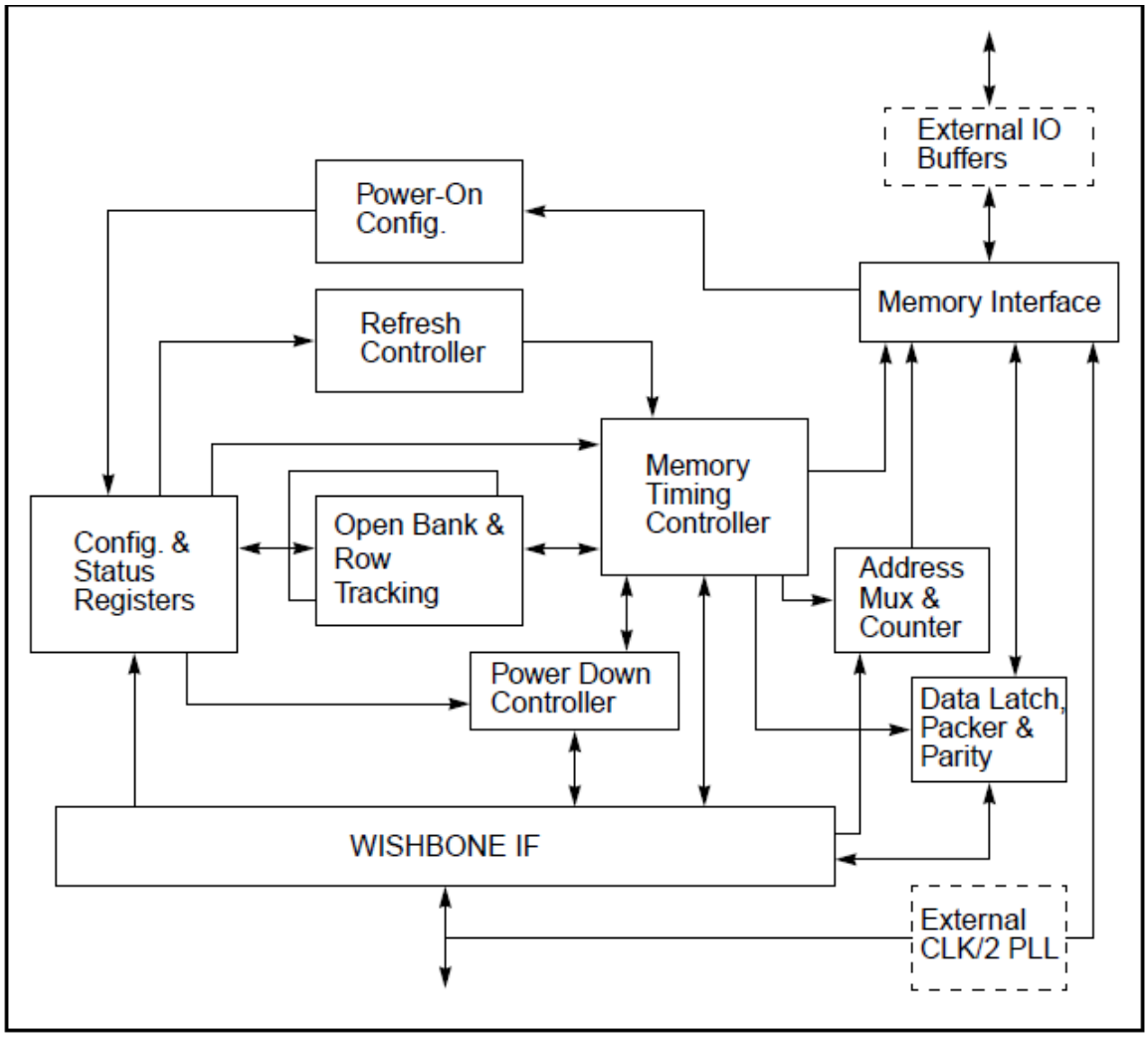

Figure 4.2: Memory Controller IP Core Block Diagram [12]

The Memory Controller connects to the Wishbone interface as a Wishbone slave where data can be written to and read from it through Wishbone masters. There are also configuration and status registers that can be configured by wishbone masters for targeted chip on the memory controller to memory interface. It is also important to note that the memory itself connects to the memory controller through a separate Memory Interface block that is not part of the wishbone interconnect. This is a direct connection between

the memory module and the memory controller. The SDRAM memory mode supported is a Micron 2 Megabytes by 32 giving 64Mbytes of RAM size. This SDRAM is also extendable since the memory controller is able to support connections to multiple SDRAMs where four of these SDRAMs are connected to the memory controller and each is selectable by a Wishbone accessible configuration register. Both a single 64MB SDRAM and a 256MB multi SDRAM setup were enabled in the simulated system. The memory controller module was tested by first configuring the internal registers for accessing SDRAMs, after which a bulk set of data was written to and read from the targeted SDRAM and compared for correctness.

SD Card Controller IP Core

The Secure Digital (SD) controller allows connection and control for the purposes of storage on an SD card. SD cards typically store up to 16GB of data and this is suitable for storage for the simulated system. The SD Card Controller IP supports a 32-bit wishbone bus interface, an SD bus interface for connection to a SD card, interrupts on completion of Data and Command transmissions among other features. The SD Card storage is also simulated to test for functionality of the SD Card Controller. Figure 4.3 shows the block diagram of the SD Card Controller IP block.

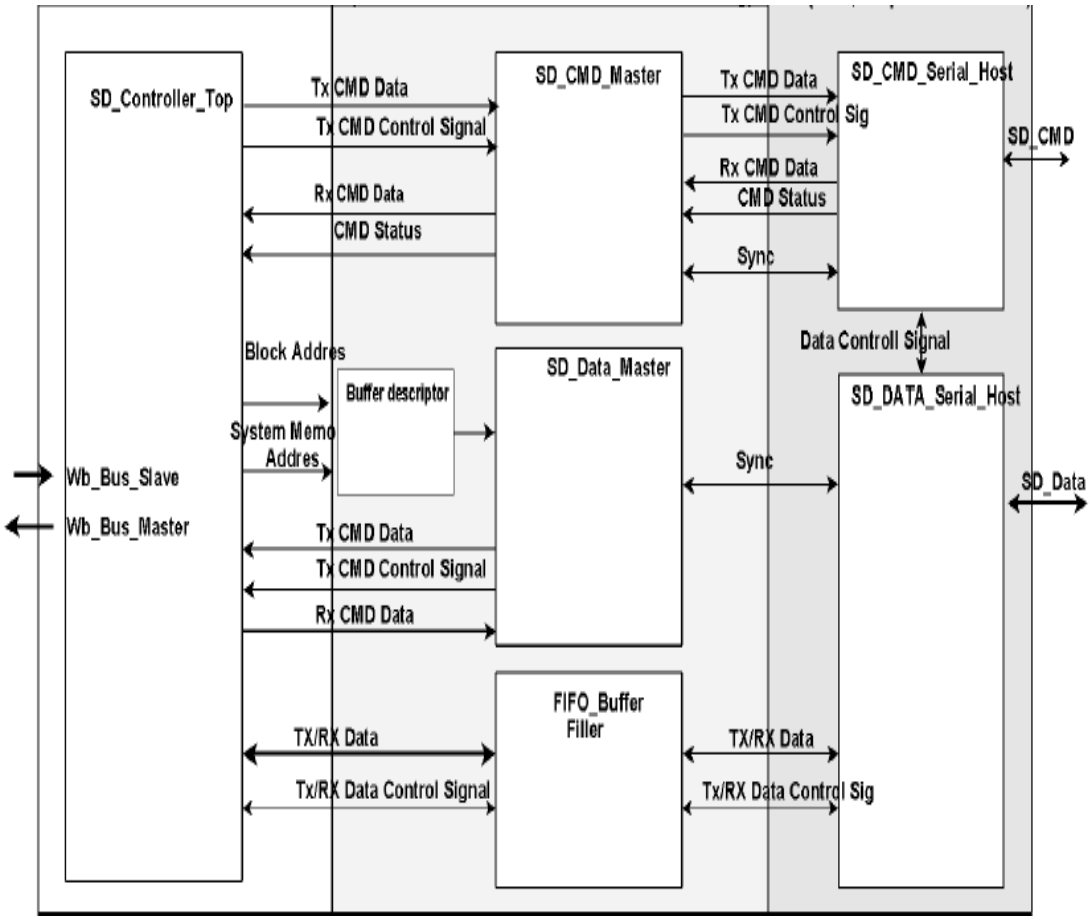

Figure 4.3: SD Card Controller IP Core Block Diagram [13]

The “SD_Controller_Top” block interfaces the SD controller core with the Wishbone bus. Internal registers and buffer descriptors are accessible through the Wishbone bus’ Slave interface. The “SD_CMD_Master” performs the tasks of reading registers accessible through “SD_Controller_Top” block and responds to the SD command host. The “SD_Data_Master” will check data from the Buffer Descriptor block and generates the required commands while waiting for the response after generating the buffer descriptor’s command. The SD command and data hosts is the physical layer interface to the SD card and checks for the physical sending and receiving of data on the SD

interface. The “FIFO_Buffer_Filler” block manages receive and transmit data between the Wishbone interface and the “SD_Data_Host” block. There is also a Wishbone bus master supported for accessing external memories for additional memory and access of data.

The SD controller tests exercise programming of the internal SD controller’s registers, sending commands for initialization sequence as well as data writes and reads into SD controller to the SD card.

SPI Controller IP Core

The Serial Peripheral Interface (SPI) is a serial, synchronous communication protocol that communicated through three wires for clock and data transmit and receive given by sck_o, miso_i, mosi_o in Figure 4.4. The SPI Controller IP core is a Wishbone slave that also controls an external SPI device using the SPI port connections. The SPI Controller also includes control and configuration registers, read/write FIFOs for data buffering and supports interrupt generation to the host system. Figure 4.4 shows the architecture block diagram of the SPI Controller IP Core.

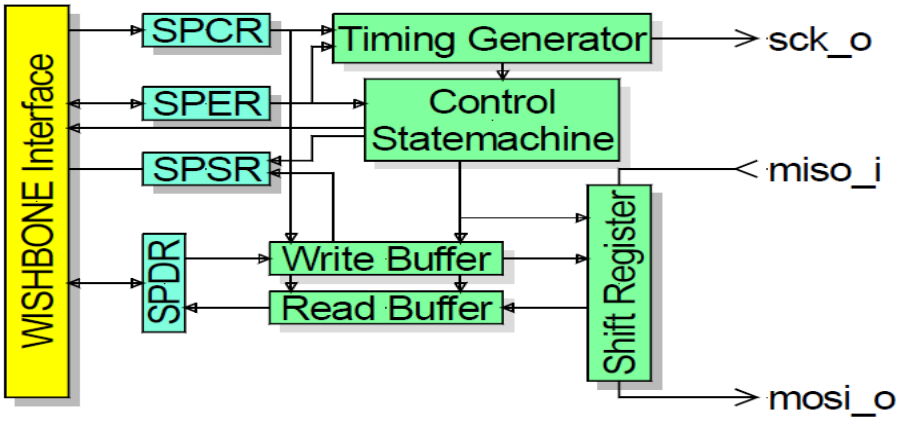

Figure 4.4: SPI Controller IP Core Block Diagram [14]

The SPI testing includes a simulated SPI device or slave connected to the sck_o, miso_i and mosi_o. The test begins by programming internal configuration and control registers to enable SPI transfers. A data write is initiated by writing to the Serial Peripheral Data Register (SPDR) that writes to the SPI FIFO (Write Buffer) that can then be shifted out to the mosi_o pin. Data bytes for read transactions are achieved by reading data from the Read Buffer through SPDR once data is received from miso_i.

Simulated Wishbone Master Module

In order to exercise wishbone master transactions that would typically be provided by a processor, a simulated wishbone master model was built. The Wishbone Master module contains the Verilog tasks for Wishbone transactions and connections for accessing the Wishbone slaves. The IP cores embed tasks that are initiated from the Wishbone master in these simulated models to provide the required Wishbone Master stimulus. Therefore, the final Wishbone master module contains all the tasks required for all reused IP cores

and was also extended for sideband bus support and cache coherence data generation as part of the enhancements for MCB. There can also be multiple Wishbone master modules instantiated in the simulated system and the Wishbone master modules will connect to the interconnect module.

Additional Components of Simulated MCB Architecture System:

Interconnect selection

There are multiple methods of building an interconnect system to enable multi-master, multi slave accesses. The two main methods studied were the shared bus and the crossbar switch interconnect schemes.

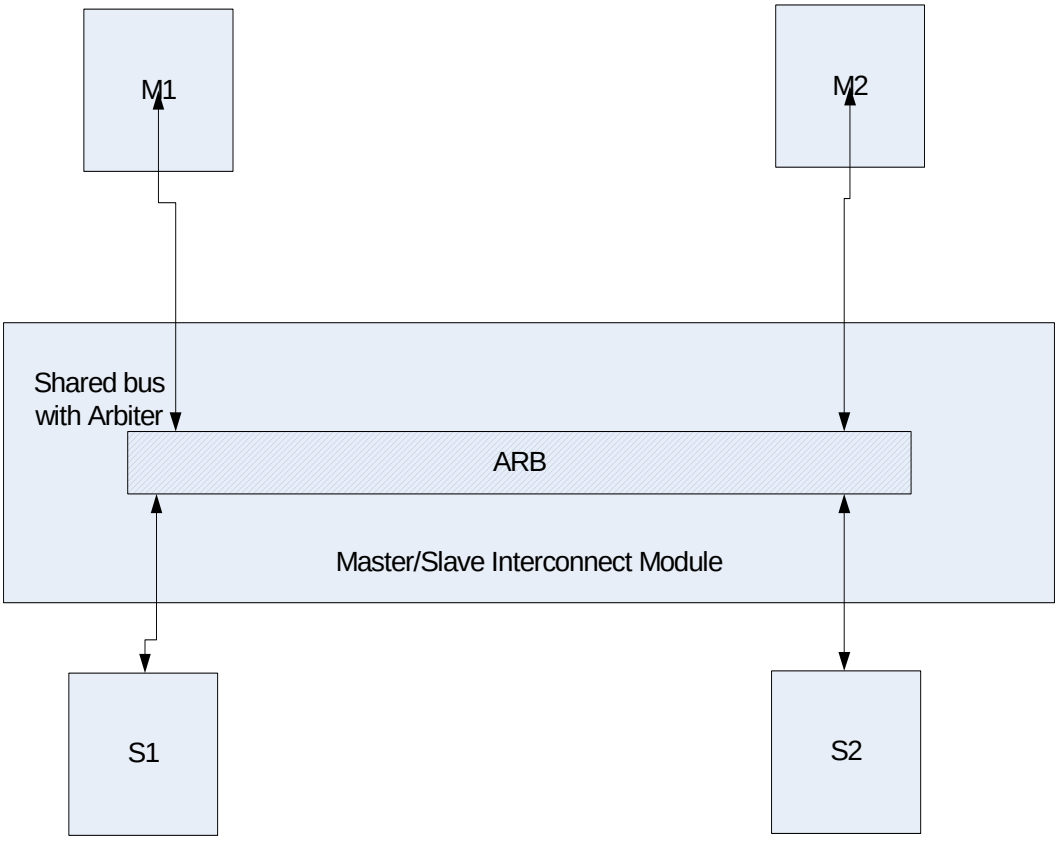

Figure 4.5: Shared Bus Interconnect Scheme

The shared bus interconnect scheme would only allow a single master (M1 or M2) to access one of the slaves at one bus cycle. The granting of a master is controlled by the arbiter. The advantage is that this is a simple scheme for interconnect implementation and has limited signal interface to the interconnect module and less impact to gate count in the system. The disadvantages of this scheme is that any possible parallel access between masters and slaves is not supported and this limits the scalability if there are multiple masters in the system due to performance constraints.

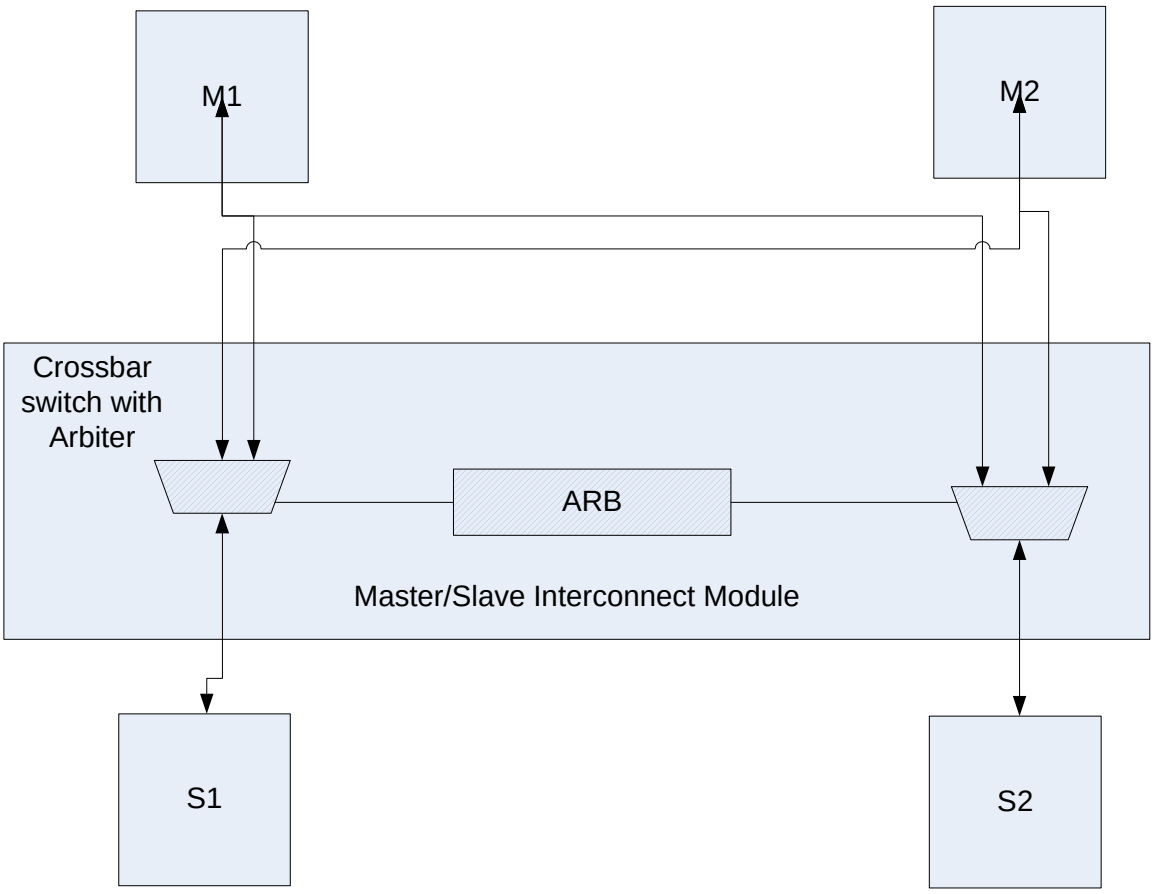

Figure 4.6: Crossbar Switch Interconnect Scheme

Figure 4.6 shows the Crossbar Switch implementation where each master has a dedicated channel to connect to an individual slave. This would allow for parallel access if different masters are accessing different slaves. The key advantage of the crossbar switch method is the opportunity for increased bus performance, particularly for multi master accesses to unique slaves. A key disadvantage is that it requires additional connectivity to the interconnect module.

In order to assess the two scheme's performance, a mini system with both schemes were built and simulated for performance. The system consisted of two masters and two slaves on a Wishbone interconnect system. The two slaves were made up of the Memory Controller IP Core and the USB HostSlave IP core. Table 4.1 summarizes the performance data:

Bus Configuration	Master 0 Mem Ctrl Data Access (Bytes)	Master 0 USB Data Access (Bytes)	Master 1 Mem Ctrl Data Access (Bytes)	Master 1 USB Data Access (Bytes)	Total Simulation time (ns)	Relative speedup
Wishbone with shared bus with round robin arbiter	1024	380	518	0	303060	1
Wishbone with cross bar switch & round robin arbiter	1024	380	518	0	215140	1.408664126

Table 4.1: Crossbar Switch Interconnect Scheme

Based on the analysis of the simulation with multi-master and multi-slave access, the relative speedup of 1.41 was found to be large enough to justify the additional signal connectivity and gate count incurred by the crossbar switch. The estimated additional

gate count was an increase of less than 10% for the crossbar switch scheme which is acceptable for the performance improvement.

Another important consideration is power consumption. With the accesses working in parallel there would theoretically be a higher dynamic power consumption in the crossbar switch method compared to the shared bus method. This can be offset by the earlier test completion where the crossbar switch scheme would be idle for 88µs in this case which would consume less power if clocks and idle logic are turned off. Therefore, the crossbar switch interconnect scheme was chosen for the simulated system.

Arbitration Scheme

As described in the "Features Added to Wishbone to Build MCB" section of Chapter 3, a scalable arbiter that supports multiple masters and slaves also needed to be designed. The Wishbone and MCB busses do not mandate the use of any particular arbitration scheme; hence, a simple, configurable round robin arbiter that supports a parameterized number of requests and grants was designed for arbitration between multi master accesses on the crossbar switch interconnect.

Sideband Extensions and Cache Coherence Support

All masters in the system were updated to support the MCB sideband signals. Additionally the Memory Controller IP and the USBOPB IP blocks were also modified to support the sideband signaling. Note that the USBOPB sideband signaling is a

conversion of the IBM DCR sideband bus to give an idea of how this can be bridged to the MCB sideband bus for reuse of IBM OPB IP cores that access DCR registers.

Secondary Interconnect Bus and Addressing Methods

For improved performance, a secondary interconnect bus was also built for use with lower speed interfaces. The SPI, USBHostSlave, USBOPB and SD Controller IPs were placed in the secondary interconnect bus due to their lower core frequencies and also longer response time due to their slower external interfaces. Note that this secondary interconnect appears as a slave to the primary interconnect module.

In order to address between these IP cores on the secondary interconnect module, they use different address ranges of their 32-bit address busses. In order to differentiate addresses of IPs between the primary and secondary busses, part of the TGA_O address tag signal is used to indicate the targeted bus. Additionally to differentiate main memory accesses intended for memory controller versus addresses of the rest of the IP cores, part of the TGA_O tag field is also used.

See table 4.2 for the addressing scheme and bus frequencies of the different IP cores in the system.

IP	Interconnect	TGA	Address Range	Interconnect Frequency	IP Core Frequency
Memctrl	Primary	04h	0000_0000h - FFFF_FFFFh	200Mhz	200Mhz
SPI	Secondary	01h	1000_0000h - 1FFFF_FFFFh	100Mhz	100Mhz
SD	Secondary	01h	2000_0000h - 2FFFF_FFFFh	100Mhz	100Mhz
USB	Secondary	01h	3000_0000h - 3FFFF_FFFFh	100Mhz	50MHz
USBOPB	Secondary	01h	4000_0000h - 4FFFF_FFFFh	100Mhz	50MHz

Table 4.2: IP Cores Tag, Addressing and Frequency Information in System

Similarly for sideband accesses, the tags on the sideband bus are used to distinguish between cacheable and non cacheable access as well as primary and secondary addresses.

Finalized Simulated MCB Architecture System:

The completed system architecture for MCB was built as shown in Figure 4.7.

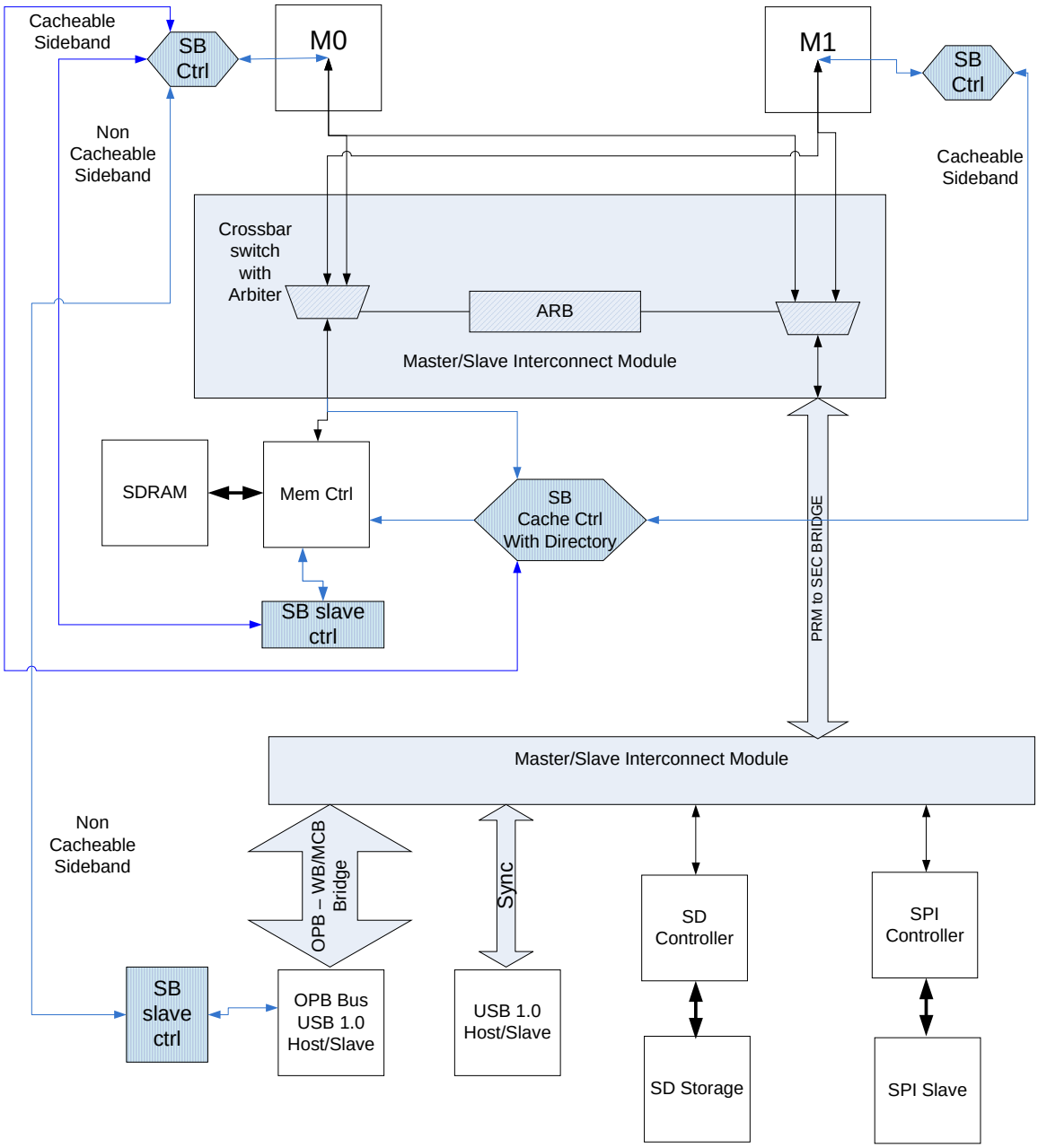

Figure 4.7: Finalized MCB System Architecture

As shown in Figure 4.7 all the IP Cores discussed earlier (Memory Controller, USB, SD Controller, SPI, etc.) were implemented in this system. The MCB sideband bus was also implemented for the Memory Controller, USB OPB and the cache controller containing the cache directory used to maintain data coherence between M0, M1 and the Memory Controller modules. There are also two crossbar switch interconnect modules to separate high speed and low speed IP blocks that were synchronized by the Primary to Secondary interconnect bridge as shown in Figure 4.7. The system supports two masters given by M0 and M1 that support wishbone master transactions and MCB sideband master and slave transactions. All these modules were connected together in a Verilog test bench that also contains the test stimuli for validating each IP block either individually or concurrently with other IP blocks. Each of the IP cores, MCB sideband blocks, and interconnect modules in Figure 4.7 was coded with synthesizable Verilog code while the test bench and the M0 and M1 modules are not synthesizable. This was done for future enhancements to synthesize a MCB system either with ASIC synthesis tools or on a FPGA.

Flow Diagram for Functionality of Finalized Simulated MCB System:

In order to better understand the functionality of the system described in Figure 4.7, the following flow diagrams and their respective descriptions can be used.

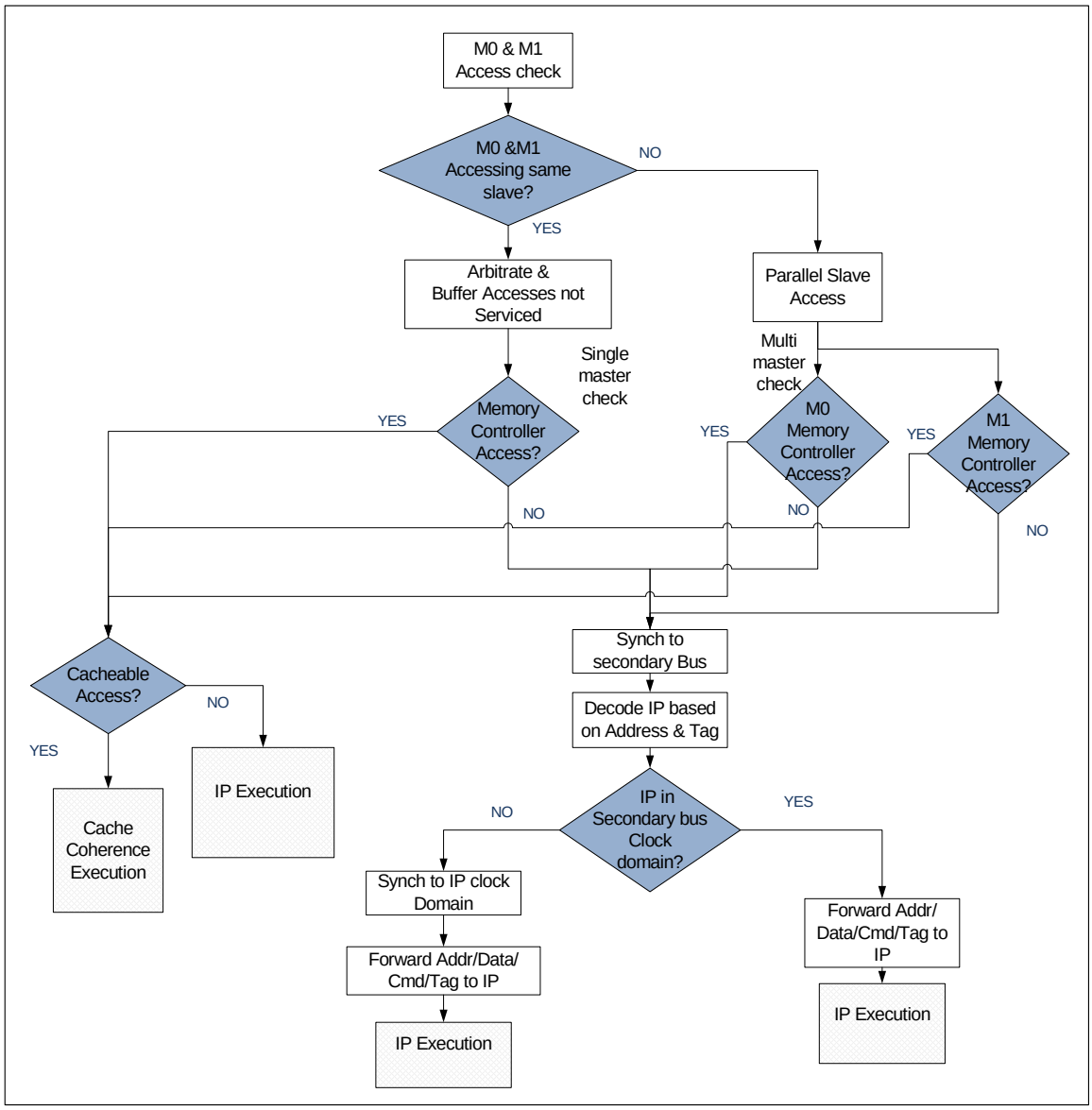

Figure 4. 8: MCB System Flow Diagram

The MCB System Flow Diagram as given in Figure 4.8 shows how the blocks in the MCB System Architecture Diagram given by Figure 4.7 work together. The first step in the flow is where the Master/Slave Interconnect Module determines the targeted slave for each master. Arbitration requirements and parallel slave access capabilities are then determined. If there is arbitration required, the Master which is not being serviced has its input data buffered until the system can service its request. For memory controller accesses which are cacheable, the cache directory, memory controller and each cacheable master will function according to the Cache Coherence Execution flow given by Figure 4.9. For non cacheable accesses to memory controller, such as configuration register accesses, the memory controller will function according to the IP Execution Flow given by Figure 4.10. These cacheable versus non-cacheable accesses to the memory controller are determined by the address range selected within the memory controller address space.

For non memory controller accesses, a synchronization of the clock domains needs to happen since all these slaves are on the secondary interconnect module which functions at half the frequency of the primary interconnect module. There is also further synchronization required if the IPs on the secondary bus are not in the same clock domain as the secondary interconnect module such as USB and USBOPB. Once the targeted slave is decoded based on address and tag provided by the requesting master, the address, data, tag and command information can be forwarded to it by the interconnect module for IP execution which is described in more detail by Figure 4.10.

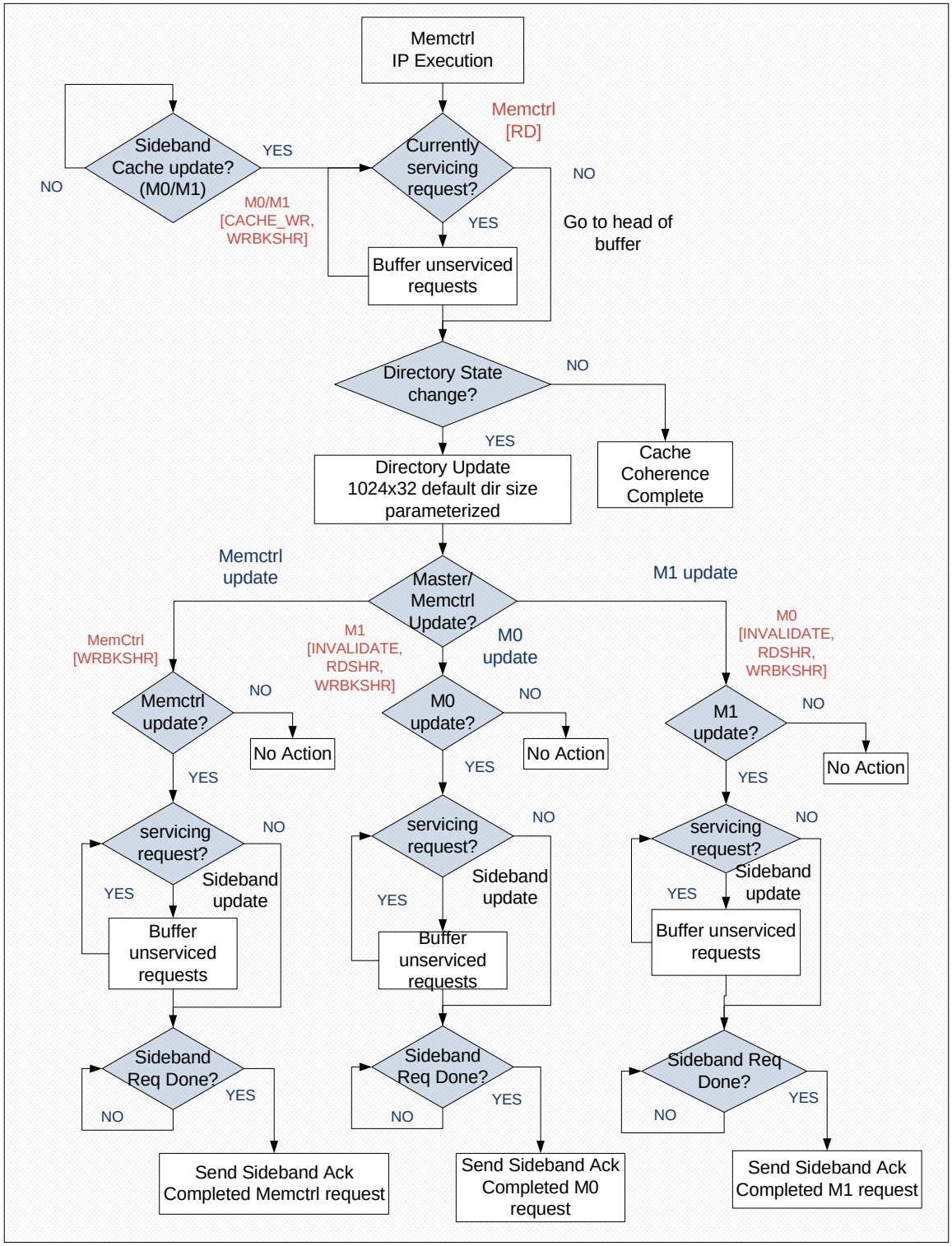

Figure 4.9: Cache Coherence Execution Flow Diagram

Figure 4.9 details the cache coherence execution functionality of the MCB system. The blocks which are involved in this flow are the memory controller, each cache coherent master (M0 and M1) and the sideband cache controller. When there is a cacheable access to the Memory controller, the memory controller completes its IP execution given by Figure 4.10. Note that in the IP execution flow, the sideband bus requests are also taken into account where sideband transactions will take priority over primary bus transactions. The sideband cache controller will service the sideband requests from each master and check for updates based on the memory controller's IP execution depending on whether there are outstanding cache coherence requests which need to be completed. Depending on the type of accesses, the status of the cache controller's directory will change state or remain the same. If there is a change in the directory status then the action taken to M0, M1 and memory controller will need to be determined based on the updated state of the directory.

The fonts in red in Figure 4.9 contain information of the blocks involved and the respective sideband tag which will accompany the sideband transaction for the cache coherence scheme. For example, the memory controller will be updated through sideband during the "writeback shared" stage (WRBKSHR) while each master will be updated during the "invalidate", "read share" or "writeback share" states. The destination of the sideband access is then determined and the sideband update can take place once all the outstanding sideband cache coherent requests are completed. The outstanding

requests are stored in a buffer in the cache controller and the buffer size defaults to a depth of 8 (to allow up to 8 outstanding transactions) which is also parameterized. Once the sideband request is completed the targeted sideband destination agent (either memory controller, M0 or M1) sends and acknowledge to indicate the completion of the cache coherent transaction.

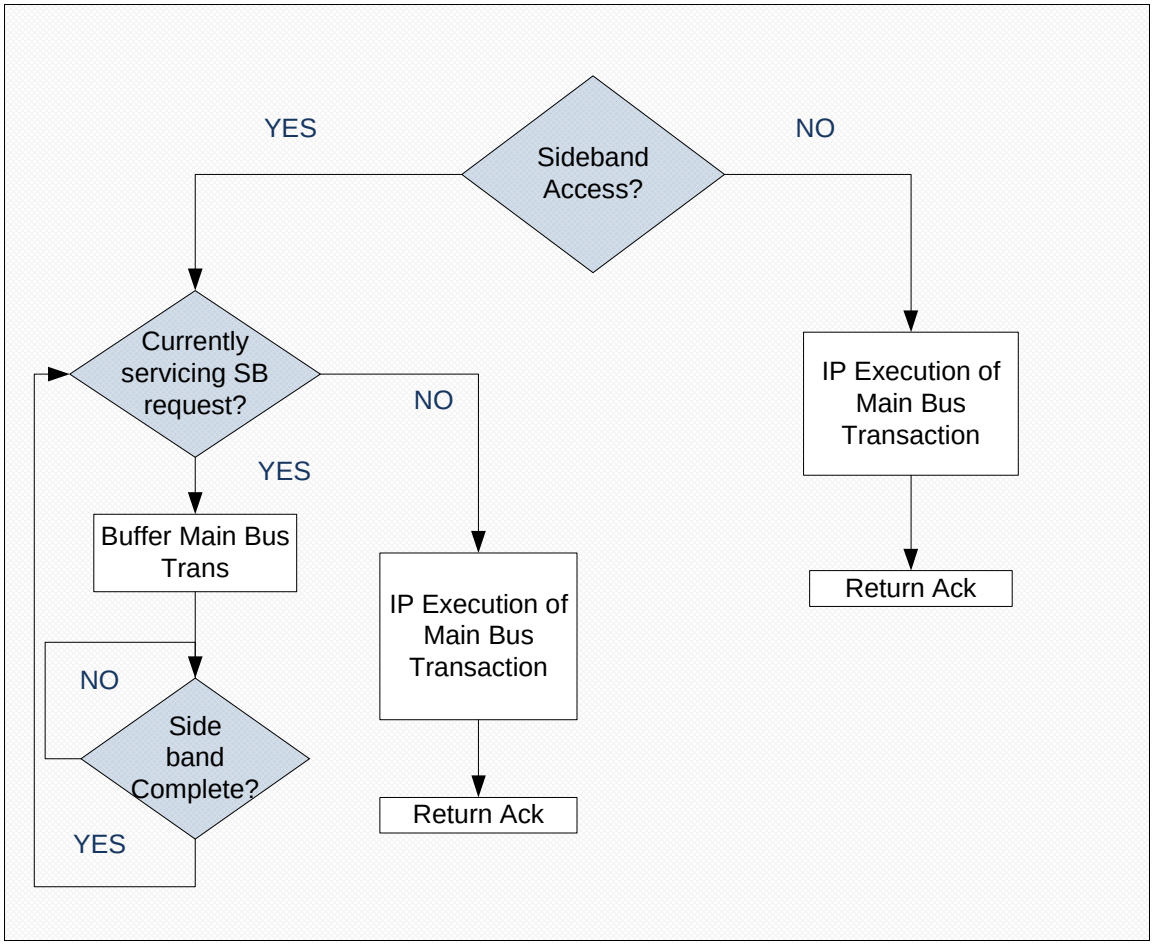

Figure 4.10: IP Execution Flow Diagram

Figure 4.10 shows the basic flow diagram of the IP execution for IPs supporting sideband and main bus transactions. If there are no sideband accesses pending then the IP executes

the main bus transaction. If there are sideband access pending then all the pending sideband access need to be complete before the main bus transactions can commence. The system as shown in Figure 4.7 and described by flow diagrams in Figures 4.8, 4.9 and 4.10 is now ready for simulation testing for various test cases. This testing, test scenarios and performance analysis are covered in the following “System Simulation Results and Analysis” chapter.

Chapter 5: System Simulation Results and Analysis

Once the system RTL model was ready and the test bench and tests were completed, the simulation of the system shown in Figure 4.7 was conducted using the Modelsim RTL simulation tool. This tool was chosen due to its various powerful debug features and ability to support Verilog RTL, tests and test bench features. It is also important to note that there was extensive testing conducted with various revisions of the Wishbone and MCB systems with the different IPs throughout the course of this research to qualify the IPs and the new MCB features. This section only covers the key test cases to focus on particular concurrent scenarios for the interest of brevity.

Sideband Cache Coherence Validation

In order to validate the MCB sideband cache coherence capability and cache directory features with the masters and memory controller, tests were written to cover the common cache coherence scenarios in a dual processor environment. The test conditions and results are summarized in Table 5.1. In order to better understand these test scenarios, the data paths can be deduced from Figure 5.1 that show the components and connections which were exercised for these tests.

Instruction Step	Master 0 memory Transaction	Address	SB Coherence Tag	M0 MESI State	M0 Sideband bus transaction	Master 1 memory Transaction	Address	SB Coherence Tag	M1 MESI State	M1 Sideband bus transaction	Cache directory SB Transaction	Cache Dir Status (M0,M1)
1	MEM RD	h0000_0001	N/A	E	N/A	N/A	N/A	N/A	I	N/A	N/A	RD,I
2	N/A	h0000_0001	RDSHR	S	SBRD	MEM RD	h0000_0001	RDSHR	S	SBRD	SBWR for RDSHR indication	RD,RD
3	CACHE WR	h0000_0001	CACHE_WR	M	SBWR	N/A	h0000_0001	INV	I	SBRD	SBWR for M1 invalidate	WR,I
4	N/A	h0000_0001	WRBSHR	S	SBWR	MEM RD	h0000_0001	RDSHR	S	SBRD	WRBSHR to M0, RDSHR to M1	RD,RD
5	N/A	h0000_0001	INV	I	SBRD	CACHE WR	h0000_0001	CACHE_WR	M	SBWR	Invalidate for M0	I,WR
6	N/A	N/A	N/A	I	N/A	MEM RD	h0000_0005	N/A	E	N/A	N/A	I,RD
7	MEM RD	h0000_0005	RDSHR	S	SBRD	N/A	h0000_0005	RDSHR	S	SBRD	SBWR for RDSHR indication	RD,RD
8	CACHE WR	h0000_0005	CACHE_WR	M	SBWR	N/A	h0000_0005	INV	I	SBRD	SBWR for M1 invalidate	WR,I
9	N/A	h0000_0005	WRBSHR	S	SBWR	MEM RD	h0000_0005	RDSHR	S	SBRD	WRBSHR to M0, RDSHR to M1	RD,RD
10	CACHE WR	h0000_0005	CACHE_WR	M	SBWR	N/A	CACHE WR	CACHE_WR	I	SBWR	Invalidate for M1	WR,I

Table 5.1: MCB Cache Coherence Validation Data

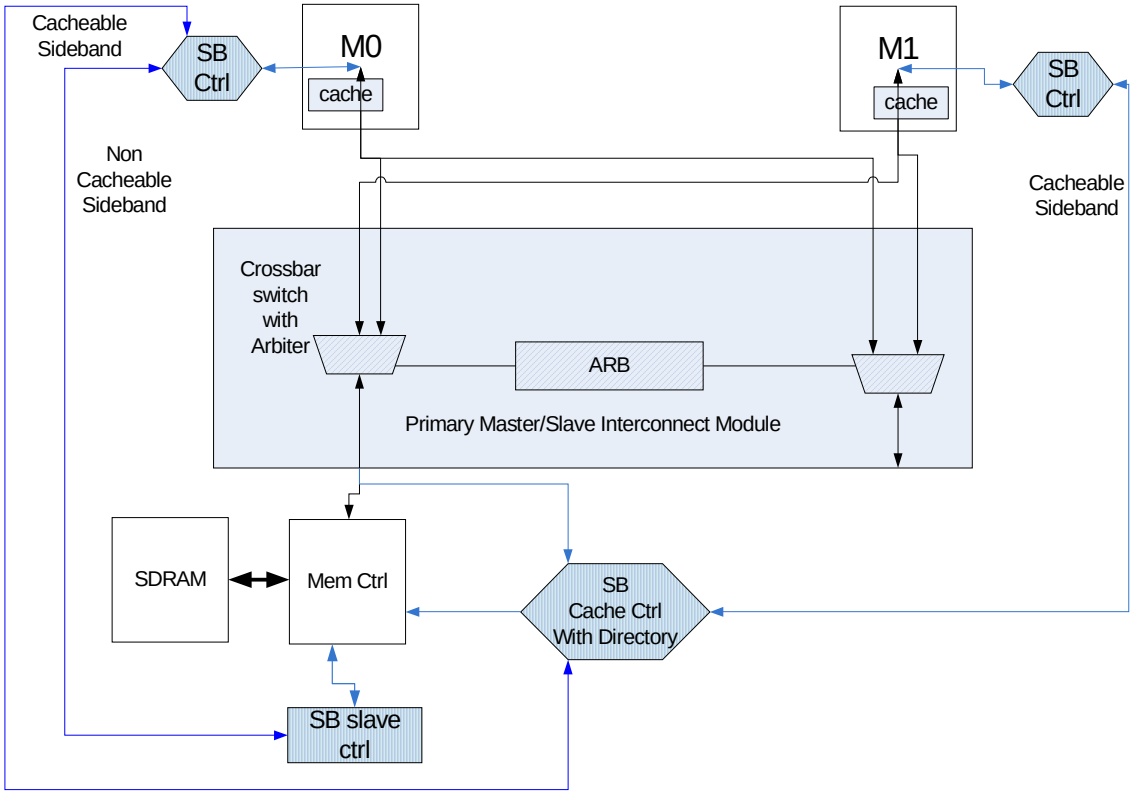

Figure 5.1: MCB Components Tested for Cache Coherence

- Instruction steps 1 and 2 in Table 5.1 cover the M0 and M1 data reads from SDRAM. These transactions are also monitored by the SB Cache Controller (SBCC) to update the cache directory. In test step 2, the SBCC sends sideband writes to M0 and M1 to indicate that the data in address 0000_0001h is shared.
- Instruction step 3 is a condition where M0 writes to address 0000_0001h in its cache. M0 sends the "CACHE_WR" indicator to the SBCC and the SBCC subsequently sends an invalidate indicator to M1. SBCC directory is also updated for this change of status.
- In instruction step 4, M1 reads address 0000_0001h from SDRAM again. The SBCC detects this read and sends the WRBSHR indicator to M0 that indicates that M0 needs to write back its modified data and change its state to shared. M0 will send the writeback data to SBCC with the WRBSHR tag. The SBCC will also send a sideband indicator to M1 to update its state to a RDSHR state and will also send the updated writeback data from M0 to M1. The Modified data will also be written to the SDRAM through the memory controller sideband from SBCC.
- In instruction step 5 M1 writes to address 0000_0001h in its cache and sends a CACHE_WR indicator to SBCC. SBCC then sends a sideband invalidate to M0 for M0 to invalidate its cacheline for this address.
- Instruction steps 6 to 9 cover similar scenarios for a different address

- Instruction step 10 covers the case where both M0 and M1 write to their caches for address 0000_0005h. In this case, the cache write for M0 will take priority over M1 and SBCC will send an invalidate indicator to M1.

More tests were then exercised to cover more data and address accesses to further test the cache coherence through sideband functionality. This effectively proved that cache coherence signaling and control is supportable through the MCB sideband bus and allows for improved configurability and scalability for multiprocessor environments.

MCB Test Scenarios for IP Concurrency and Performance Comparison

Tables 5.2 to 5.7 summarize the test results of six of the test cases used to compare performance and behavior of the Wishbone bus and MCB. To better analyze this data, Figure 5.2 showing the simulated system diagram will also be referenced.

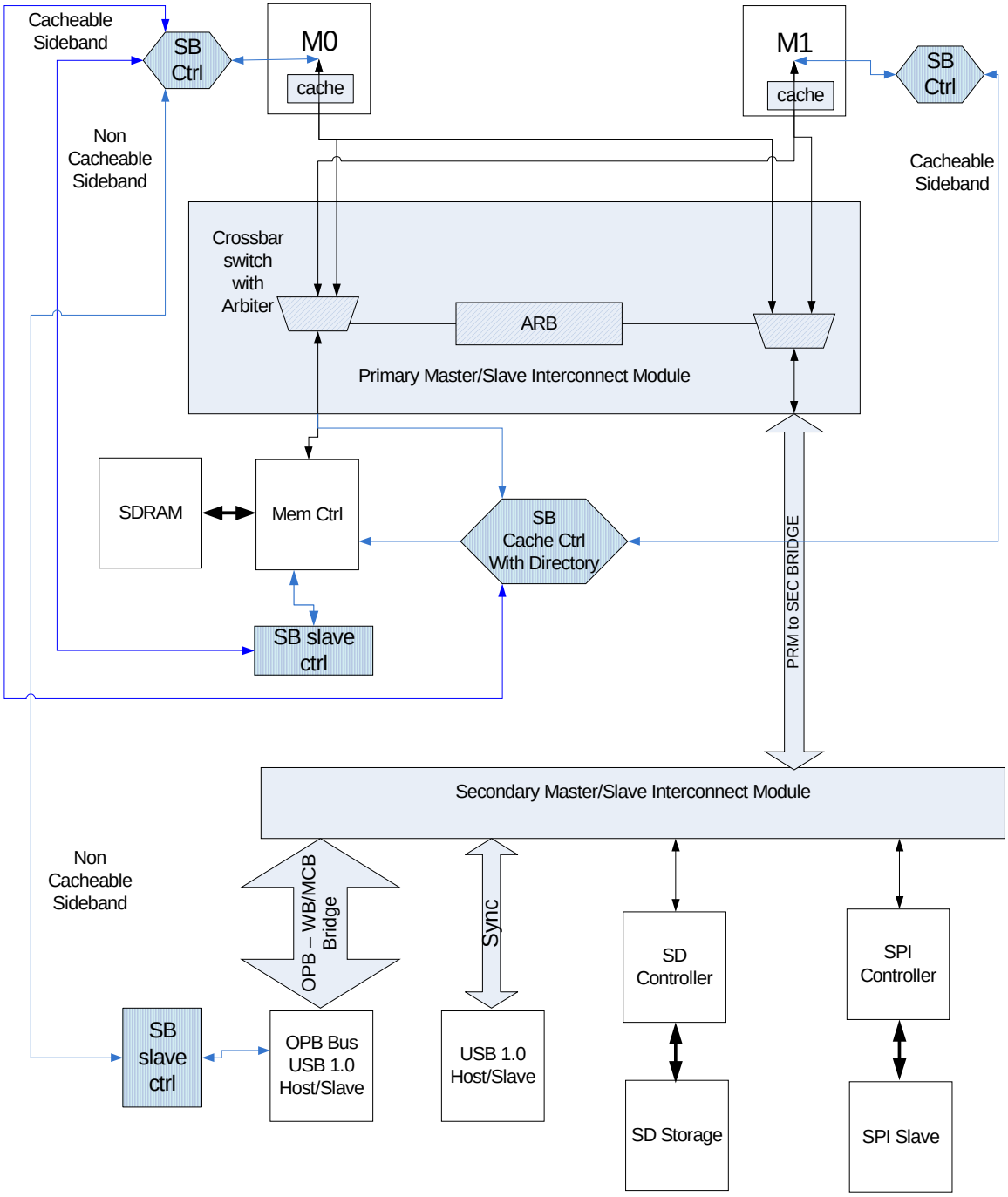

Figure 5.2: MCB System Architecture

TSTCASE1/ Bus Configuration	M0 memctrl accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 SD accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	1024	N/A	80	764	N/A	32	101971	1
MCB with cross bar switch & round robin arb, sideband	1024	80	N/A	764	32	N/A	90180	1.130749612

Table 5.2: Memory Controller and SD Controller Concurrent Access Comparison

Table 5.2 summarizes the data for the first test case that compares performance between the Wishbone and MCB for similar data accesses on the same round robin arbitration scheme. In this test case, M0 performs read and write accesses to the memory controller while M1 performs read and write accesses to the SD controller. These accesses occur in parallel since the two masters are accessing two different slaves through the crossbar switch interconnect module. There are also 80 bytes of sideband accesses supported by MCB for M0 and 32bytes of sideband accesses through M1 that also occur in parallel with the main bus transactions. The M0 sideband transactions are a combination of USBOPB, Memory Controller and cache coherence sideband accesses while the M1 sideband transactions are only for cache coherence sideband accesses. Since the Wishbone bus does not have sideband support, the equivalent accesses on the Wishbone bus were also simulated for comparison purposes. The data from table 5.2 shows a significant performance improvement given by the speedup of 1.13 for the MCB bus over Wishbone for this test case.

The selection of a small set of sideband accesses that is less than 10% of the main bus accesses was deliberately chosen to provide a conservative estimate for the speedup for

the MCB bus compared to Wishbone. With more sideband accesses, MCB will have an even higher speedup compared to Wishbone.

TSTCASE2/Bus Configuration	M0 USB accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 Mem Controller accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	120	N/A	80	1536	N/A	32	105626	1
MCB with cross bar switch & round robin arb, sideband	120	80	N/A	1536	32	N/A	105370	1.002429534

Table 5.3: USB and Memory Controller Concurrent Access Comparison

In the second test case shown by Table 5.3, M0 completes USB transactions while M1 completes memory controller transactions. There are also 80 bytes of sideband transactions on M0 and 32 bytes on M1 for MCB as in the first test case. In this test case, the M1 test is running a significantly larger number of total transactions compared to M0 and the M0 transactions finish earlier, therefore, the only significant difference between the Wishbone and MCB transactions are given by the 32 bytes of sideband transfers for M1. This is the reason for the lower speedup of 1.002 for the second test case compared to the first test case since there are less parallelized sideband transfers for the longer accesses for this test case. Furthermore, the selection of a small set of sideband accesses of less than 10% provides a conservative estimate for the speedup for the MCB bus compared to Wishbone which can increase with a higher number of sideband transactions.

TSTCASE3/ Bus Configuration	M0 USB OPB accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 Mem Controller accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	340	N/A	80	1536	N/A	32	151436	1
MCB with cross bar switch & round robin arb, sideband	340	80	N/A	1536	32	N/A	148805	1.017680857

Table 5.4: USB OPB and Memory Controller Concurrent Access Comparison

In the third test case shown by Table 5.4, M0 performs USB OPB accesses while M1 performs memory controller accesses. There are also 80 bytes of sideband transactions on M0 and 32 bytes of sideband transactions on M1 for the MCB bus as in the previous test cases. This test case shows a higher speedup on MCB compared to the second test case since there are more USB OPB transactions in this test which meant that the M0 transactions finished later than the M1 transactions. This is because there is a higher level of parallel transactions on the longer of the two tests, giving a higher speedup compared to the second test case.

TSTCASE4/ Bus Configuration	M0 memctrl accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 SPI accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	1024	N/A	80	368	N/A	32	91601	1
MCB with cross bar switch & round robin arb, sideband	1024	80	N/A	368	32	N/A	79810	1.147738379

Table 5.5: Memory Controller and SPI Concurrent Access Comparison

In the fourth test case shown by table 5.5, M0 performs Memory Controller Transactions in parallel with M1 performing SPI transactions. There are also 80 bytes of sideband transactions on M0 and 32 bytes of sideband transactions on M1 for MCB as in previous test cases. This test case also shows MCB performing significantly better than Wishbone due to the parallel sideband transactions on MCB versus the same transactions running serially in Wishbone.

TSTCASE5/ Bus Configuration	M0 USB accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 SD accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	120	N/A	80	412	N/A	32	74020	1
MCB with cross bar switch & round robin arb, sideband	120	80	N/A	412	32	N/A	71395	1.036767281

Table 5.6: USB and SD Controller Concurrent Access Comparison

In the fifth test case shown in Table 5.6, M0 performs USB accesses while M1 performs SD controller access. These accesses will happen serially and will be arbitrated by the round robin arbiter in the interconnect module as shown in Figure 5.2 since both of these IP blocks are connected to the Secondary Master/Slave Interconnect module. There are also 80 bytes of sideband transactions on M0 and 32 bytes of sideband transactions on M1 for MCB as in previous test cases. For this case as well, MCB shows a better performance compared to Wishbone for equivalent transactions given by the speedup of 1.037.

TSTCASE6/ Bus Configuration	M0 USB OPB accesses (Bytes)	M0 sideband accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 SPI accesses (Bytes)	M1 sideband accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	340	N/A	80	368	N/A	32	151516	1
MCB with cross bar switch & round robin arb, sideband	340	80	N/A	368	32	N/A	148885	1.017671357

Table 5.7: USBOPB and SPI Concurrent Access Comparison

In the sixth test case shown in Table 5.7, M0 performs USB OPB accesses while M1 performs SPI controller access. These accesses will happen serially and will be arbitrated by the round robin arbiter in the interconnect module as shown in Figure 5.2 since both of these IP blocks are connected to the Secondary Master/Slave Interconnect module. There are also 80 bytes of sideband transactions on M0 and 32 bytes of sideband transactions on M1 for MCB as in previous test cases. For this test case, MCB also shows a better performance compared to Wishbone for equivalent transactions given by the speedup of 1.018.

Serial Test for Combined Test Scenarios to Simulate System Behavior

There was also an additional test case which is beneficial for analysis consisting of a serial run of multiple accesses between the masters and slaves where the MCB system is also compared against the Wishbone system in terms of functionality and performance. This test case consists of separate Memory Controller, USBOPB and USB accesses by M0, and separate USB, Memory Controller, SPI, SD and USBOPB accesses by M1 where each master accesses their respective slave. Sideband transactions will occur concurrently on the MCB system while equivalent sideband access will occur serially for the Wishbone system. Tables 5.8 and 5.9 summarize the test results of the serial test run for Test Case 7 to compare performance and behavior of the Wishbone bus and MCB.

TSTCASE7/ Bus Configuration	M0 transactions	Number of accesses (Bytes)	M0 sideband equivalent accesses (Bytes)	M1 transactions	Number of accesses (Bytes)	M1 sideband equivalent accesses (Bytes)	Total execution time (ns)	Relative speedup
WB with cross bar switch & round robin arb	M0 memctrl accesses (Bytes)	1024	80	M1 SD accesses (Bytes)	764	32	797931	1.00
	M0 Mem controller accesses (Bytes)	1536	108	M1 USB accesses (Bytes)	120	32		
	M0 USB OPB accesses (Bytes)	340	80	M1 Mem Controller accesses (Bytes)	1536	32		
	M0 memctrl accesses (Bytes)	1024	80	M1 SPI accesses (Bytes)	368	32		
	M0 USB accesses (Bytes)	120	80	M1 SD accesses (Bytes)	412	32		
	M0 USB OPB accesses (Bytes)	340	80	M1 SPI accesses (Bytes)	368	32		
	M0 memctrl accesses (Bytes)	1024	80	M1 USB OPB accesses (Bytes)	120	32		
Total		5408	588		3688	224		

Table 5.8: Data for Serial Test Run with Multiple Accesses on Wishbone System

TSTCASE7/ Bus Configuration	M0 transactions	Number of accesses (Bytes)	M0 sideband accesses (Bytes)	M1 transactions	Number of accesses (Bytes)	M1 sideband accesses (Bytes)	Total execution time (ns)	Relative speedup over wishbone
MCB with cross bar switch & round robin arb, sideband	M0 memctrl accesses (Bytes)	1024	80	M1 SD accesses (Bytes)	764	32	743890	1.0726
	M0 Mem controller accesses (Bytes)	1536	108	M1 USB accesses (Bytes)	120	32		
	M0 USB OPB accesses (Bytes)	340	80	M1 Mem Controller accesses (Bytes)	1536	32		
	M0 memctrl accesses (Bytes)	1024	80	M1 SPI accesses (Bytes)	368	32		
	M0 USB accesses (Bytes)	120	80	M1 SD accesses (Bytes)	412	32		
	M0 USB OPB accesses (Bytes)	340	80	M1 SPI accesses (Bytes)	368	32		
	M0 memctrl accesses (Bytes)	1024	80	M1 USB OPB accesses (Bytes)	120	32		
Total		5408	588		3688	224		

Table 5.9: Data for Serial Test Run with Multiple Accesses on MCB System

From Tables 5.8 and 5.9 where M0 performs a total of 5408 transactions and M1 performs a total of 3688 transactions accessing various slaves; the MCB system continues to show a significant speedup of 7.3 percent.

MCB and Wishbone Comparison for Key Criteria

Based on the performance and test data obtained, the requirements for IP reuse and the additional logic considerations of MCB versus Wishbone; the effectiveness of the MCB architecture can be assessed. The performance data shows up to a 14% improvement in speedup for MCB versus Wishbone which is significant. Furthermore, the MCB architecture meets all the seven IP reuse requirement criteria outlined below while Wishbone meets five of them, which is another advantage of MCB versus Wishbone.

- Defines a standardized interconnect
- A portable, reusable and easily understood bus protocol
- Open arbitration scheme
- Configurable interconnect widths and bus clock speeds
- Support for a wide range of bus topologies
- Sideband support for easier bridging to external busses
- A defined method for cache coherence support

The additional gate count incurred for the MCB features in the MCB simulated system is estimated to be less than 3% of an increase compared to the Wishbone system which would also justify the features implemented for MCB.

Chapter 6: Conclusion and Areas for Further Study

Conclusion

The Modular, Configurable Bus (MCB) architecture is shown to be an effective architecture for optimizing IP reuse and bus performance. MCB also meets all the criteria for IP reuse requirements determined by surveys done on industry standard busses. The strategy of using the existing open source Wishbone architecture as a baseline was also an effective method to assess and compare its functionality with a commonly used bus scheme. Furthermore, the complete simulation system using industry standard simulation tools proved to be an effective method to validate the functionality, capabilities and performance of the MCB bus architecture.

Although this research covered many of the requirements to prove the capability of the MCB architecture, there are further areas that can be explored to strengthen its case. This is discussed in more detail in the following section.

Areas for Further Study

Power consumption analysis and comparison between the MCB and Wishbone busses would be an important consideration, particularly for implementation in low power SoCs. There was power analysis and consideration done during the course of this research where the parallel transactions of the MCB sideband is expected to increase power consumption slightly while the reduced total execution time can end up saving power consumption overall. Therefore, it is likely that there is little power consumption difference due to the MCB additional features. Although this may be the case, it would be a beneficial exercise to make this assessment either with pre synthesis or post synthesis power analysis tools.

In addition, the simulated system that contains synthesizable and non synthesizable RTL code can be further enhanced to include synthesizable code for all its components. This could then be used for timing and synthesis checking and closure to enable fabrication of the device. Alternatively the simulated system can also be programmed on an FPGA to assess its functionality in a real world scenario.

Lastly, the capability to bridge non Wishbone compatible IPs to MCB that needed additional RTL coding effort for use in the MCB simulated system can be explored for future automation. With the standardized interconnects defined for Wishbone and MCB, this exercise would be worthwhile since it can save design teams the effort to build bridges to IPs that are not meant for the Wishbone or MCB busses. This will allow design teams to opt for the cheapest, best performing IPs available without being constrained by a significant additional design effort.

References

[1] Silicore Corp [2002]. “WISHBONE SoC Architecture Specification, Revision B.3”, *http://cdn.opencores.org/downloads/wbspec_b3.pdf*

[2] IBM Corp [2006]. “IBM CoreConnect bus cores”, *www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/F175B826ECE6FDE08725711F00770F60/$file/G224-7587-01_coreconnect_pb.pdf*

[3] ARM Corp [1999]. “AMBA Specification, Revision 2.0”, *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html*

[4] OCP International Partnership[2009]. “Open Cores Protocol Specification rev 2.1”, *www.cvsi.fau.edu/download/attachments/852535/OpenCoreProtocolSpecification2.1.pdf?version=1*

[5] IBM Corp [2007]. “128-bit Processor Local Bus Architecture Specifications Version 4.7”, *https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4*

[6] Kyeong Keol Ryu, Vincent J. Mooney III [2004]. “Automated Bus Generation for Multiprocessor SoC Design”, *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 23, NO. 11.*

[7] Chulho Shin et all [2004]. “Fast Exploration of Parameterized Bus Architecture for Communication-Centric SoC Design”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE’04).*

[8] K. Lahiri, A. Ragunathan, G. Lakshminarayana [2001]. “LOTTERYBUS: A New High-Performance Communication Architecture for System-on-Chip Designs”, *ACM 1-58113-297-2/01/000.6*

[9] J. L. Hennessy, Patterson, D. A. [2007]. *Computer Architecture A Quantitative Approach*, fourth edition, Morgan Kaufmann, San Francisco.

[10] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, NEC Corporation [1998] “Universal Serial Bus Specification”, *http://mprolab.teipir.gr/vivlio80X86/usb11.pdf.*

[11] Steve Fielding [2008]. “USBHostSlave IP Core Specification Rev 1.2”, *http://www.opencores.org.*

[12] Rudolf Usselmann [2002]. “Memory Controller IP Core Rev 1.7”, *http://www.opencores.org.*

[13] Adam Edvardsson [2009]. “SD/MMC Controller Rev 0.1”, *http://www.opencores.org.*

[14] Richard Herveille [2003]. “SPI Core Specifications Rev 0.1”, *http://www.opencores.org.*

Appendices

Appendix A: Simulation Waveforms

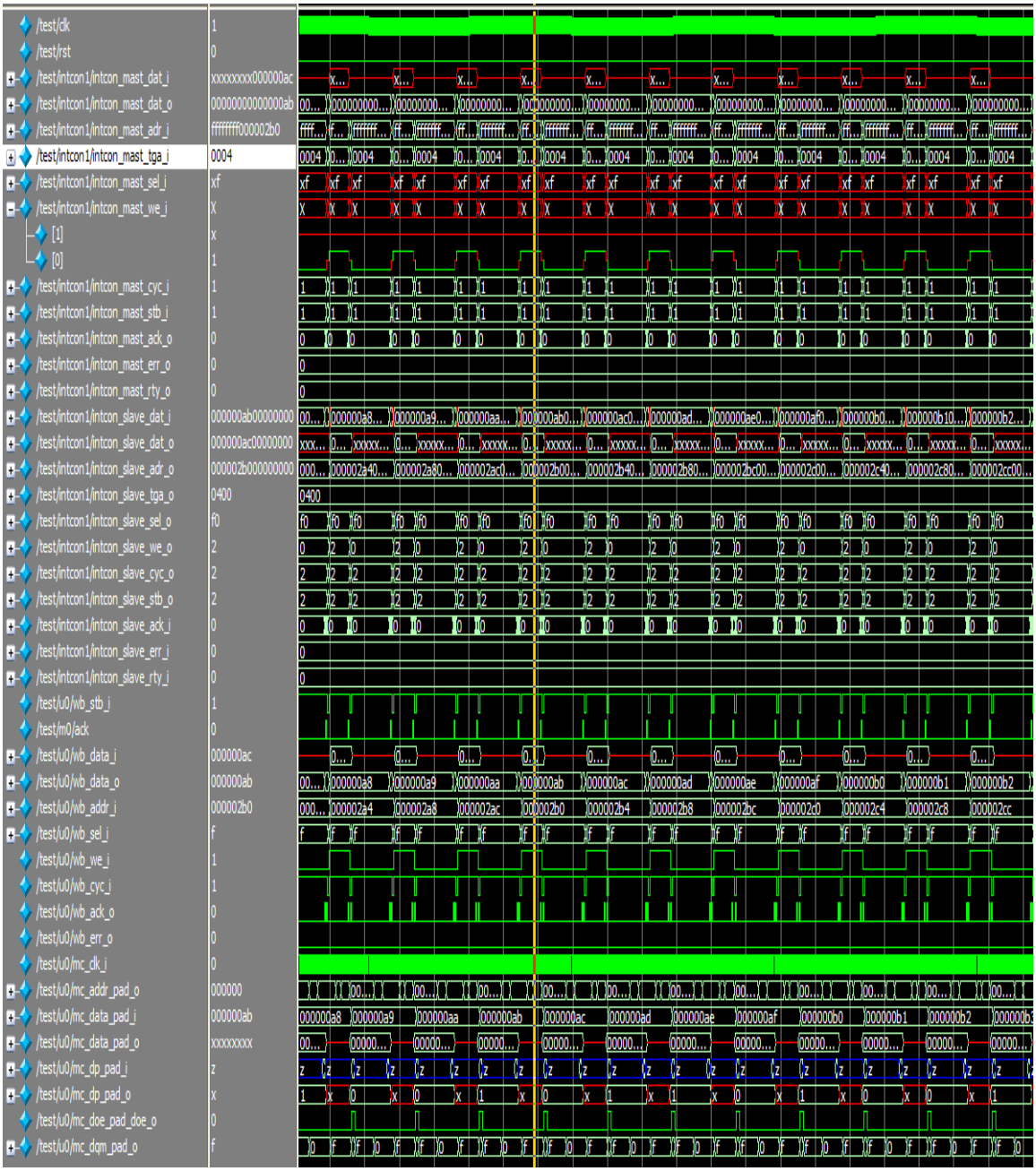

Figure A.1: Simulation Waveform For Memory Controller Access

Appendix B: RTL Snippet for Flexible Interconnect Module

```
input     clk;
input     rst;

//Flattened input bus for scalability
//master connections
input     [`DATA_WIDTH*num_mastr -1 :0] intcon_mast_dat_i;
output    [`DATA_WIDTH*num_mastr -1 :0] intcon_mast_dat_o;
input     [`ADDR_WIDTH*num_mastr -1 :0] intcon_mast_adr_i;
input     [`TAG_WIDTH*num_mastr -1 :0] intcon_mast_tga_i;
input     [`SEL_WIDTH*num_mastr -1 :0] intcon_mast_sel_i;
input     [num_mastr -1 :0] intcon_mast_we_i;
input     [num_mastr -1 :0] intcon_mast_cyc_i;
input     [num_mastr -1 :0] intcon_mast_stb_i;
output    [num_mastr -1 :0] intcon_mast_ack_o;
output    [num_mastr -1 :0] intcon_mast_err_o;
output    [num_mastr -1 :0] intcon_mast_rty_o;
//arbiter instantiated inside this module
//input   [num_mastr -1 :0] intcon_mast_cab_i;

//slave connections
input     [`DATA_WIDTH*num_slave -1 :0] intcon_slave_dat_i;
output    [`DATA_WIDTH*num_slave -1 :0] intcon_slave_dat_o;
output    [`ADDR_WIDTH*num_slave -1 :0] intcon_slave_adr_o;
output    [`TAG_WIDTH*num_slave -1 :0] intcon_slave_tga_o;
output    [`SEL_WIDTH*num_slave -1 :0] intcon_slave_sel_o;
output    [num_slave -1 :0] intcon_slave_we_o;
output    [num_slave -1 :0] intcon_slave_cyc_o;
output    [num_slave -1 :0] intcon_slave_stb_o;
input     [num_slave -1 :0] intcon_slave_ack_i;
input     [num_slave -1 :0] intcon_slave_err_i;
input     [num_slave -1 :0] intcon_slave_rty_i;
//arbiter instantiated inside this module
//output [num_slave -1 :0] intcon_slave_cab_o;



integer inc_i, inc_j, inc_cnt , inc_slv;
reg  [31:0] mast_indxs [num_slave -1 :0];

//local list of address & data
reg       [`DATA_WIDTH-1:0]  mast_dat_i_list [num_mastr -1 :0];
reg       [`DATA_WIDTH-1:0]  mast_dat_o_list [num_mastr -1 :0];
reg       [`ADDR_WIDTH-1:0]  mast_adr_i_list [num_mastr -1 :0];
reg       [`TAG_WIDTH-1:0]   mast_tga_i_list [num_mastr -1 :0];
reg       [`SEL_WIDTH-1:0]   mast_sel_i_list [num_mastr -1 :0];

reg       [`DATA_WIDTH-1:0]  slave_dat_i_list [num_slave -1 :0];
reg       [`DATA_WIDTH-1:0]  slave_dat_o_list [num_slave -1 :0];
reg       [`ADDR_WIDTH-1:0]  slave_adr_o_list [num_slave -1 :0];
reg       [`TAG_WIDTH-1:0]  slave_tga_o_list [num_slave -1 :0];
reg       [`SEL_WIDTH-1:0]   slave_sel_o_list [num_slave -1 :0];
```

```
//reg     [`ADDR_WIDTH-1:0]  slave_adr_range_list [num_slave -1 :0];
//disabled: set in header file
//wire    [`ADDR_WIDTH-1:0]  slave_adr_range_list [num_slave -1 :0];
reg       [num_mastr-1 :0] slave_arb_req_list [num_slave -1 :0];
reg       [num_mastr-1:0]  slave_arb_req_list_cmp [num_slave -1 :0];
reg       [num_mastr-1:0] slave_arb_req_cnt [num_slave -1 :0];


//connections to arbiter
reg       [num_mastr-1 :0] req_to_arb_list [num_slave -1 :0];
wire      [num_mastr-1 :0] gnt_fr_arb_list [num_slave -1 :0];



wire      [`DATA_WIDTH-1:0] mast_slave_deflt_data ;
wire    [`ADDR_WIDTH-1:0] mast_slave_deflt_adr ;
wire      deflt_sel, deflt_we, deflt_cyc, deflt_stb, deflt_ack, deflt_err, deflt_rty;

//registers to store output before sending to ports
//needed to overcome compile errors
//Master outputs
reg       [`DATA_WIDTH*num_mastr -1 :0] intcon_mast_dat_o;
reg       [num_mastr -1 :0] intcon_mast_ack_o;
reg       [num_mastr -1 :0] intcon_mast_err_o;
reg       [num_mastr -1 :0] intcon_mast_rty_o;

//Slave outputs
reg       [`DATA_WIDTH*num_slave -1 :0] intcon_slave_dat_o;
reg       [`ADDR_WIDTH*num_slave -1 :0] intcon_slave_adr_o;
reg       [`TAG_WIDTH*num_slave -1 :0] intcon_slave_tga_o;
reg       [`SEL_WIDTH*num_slave-1 :0]  intcon_slave_sel_o;
reg       [num_slave -1 :0] intcon_slave_we_o;
reg       [num_slave -1 :0] intcon_slave_cyc_o;
reg       [num_slave -1 :0] intcon_slave_stb_o;

//reassign inputs which go to "x"
reg       [num_mastr -1 :0] intcon_mast_stb_reg;



//assign defaults:

          //assign mast_slave_deflt_data = `DATA_WIDTH'bx;
          assign mast_slave_deflt_data = `DATA_BLANK;
          //assign mast_slave_deflt_adr = `ADDR_WIDTH'bx;
          assign mast_slave_deflt_adr = `ADDR_BLANK;
          //assign deflt_sel  = `SEL_WIDTH'bx;
          assign deflt_sel  = `SEL_BLANK;
          assign deflt_we   =  1'b0;//1'bx;
          assign deflt_cyc  = 1'b0;//1'bx;
          assign deflt_stb  = 1'b0;//1'bx;
```

```
assign  deflt_ack = 1'b0;//1'bx;
assign  deflt_err = 1'b0;//1'bx;
assign  deflt_rty = 1'b0;//1'bx;


//Logic for parameterized slave addressing & tag check
//This needs to be updated if number of slaves increases
wire [`ADDR_WIDTH-1:0]  slave_adr_base_list [`MAX_NUM_SLAVES -1 :0];
wire [`ADDR_WIDTH-1:0]  slave_adr_limit_list [`MAX_NUM_SLAVES -1 :0];
wire [`TAG_ADDR_WDTH-1:0] slave_adr_bus_tag [`MAX_NUM_SLAVES -1 :0];




always @ (*)
 begin
   if (rst)
      begin
            for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
                  intcon_mast_stb_reg[inc_i] = 1'b0;
      end
    else
      begin
            for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
              begin
               if ( intcon_mast_stb_i[inc_i] == 1'b1)
                  intcon_mast_stb_reg[inc_i] = intcon_mast_stb_i[inc_i] ;

                else
                  intcon_mast_stb_reg[inc_i] = 1'b0;
              end
      end
 end

//layer flattened master's  bus into a list
always @ (posedge clk)
 begin
          if (rst)
           begin
             for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
                  for (inc_j = 0 ; inc_j < `DATA_WIDTH ; inc_j = inc_j+1)
                        mast_dat_i_list[inc_i] [inc_j] = 1'b0; //bit wise assign
             for (inc_i = 0; inc_i < num_mastr*`DATA_WIDTH; inc_i= inc_i + 1)
                        intcon_mast_dat_o[inc_i] = 1'b0;
             for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
                  for (inc_j = 0 ; inc_j < `ADDR_WIDTH ; inc_j = inc_j+1)
                        mast_adr_i_list[inc_i][inc_j] = 1'b0;
             for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
                  for (inc_j = 0 ; inc_j < `TAG_WIDTH ; inc_j = inc_j+1)
                        mast_tga_i_list[inc_i][inc_j] = 1'b0;
             for (inc_i = 0; inc_i < num_mastr; inc_i= inc_i + 1)
                  for (inc_j = 0 ; inc_j < `SEL_WIDTH ; inc_j = inc_j+1)
```

```
                                mast_sel_i_list[inc_i] [inc_j] =1'b0;
                        end
                       else
                        begin
                         for (inc_i = 1; inc_i <= num_mastr; inc_i= inc_i + 1)
                          begin
                                //Need 2-for loops to do bitwise storage, cant do slice since boundaries
need to be constant expressions
                                inc_cnt = 0;
                                for (inc_j = (inc_i-1)*`DATA_WIDTH ; inc_j <= (inc_i*`DATA_WIDTH-1);
inc_j = inc_j+1)
                                   begin
                                        mast_dat_i_list[inc_i-1] [inc_cnt] = intcon_mast_dat_i[inc_j];
                                        intcon_mast_dat_o[inc_j] = mast_dat_o_list[inc_i-1] [inc_cnt] ;
                                        //$display ( " Master info 1 intcon_mast_dat_i: 0x%0h,
mast_dat_i_list: 0x%0h, intcon_mast_dat_o: 0x%0h, mast_dat_o_list:0x%0h ",
intcon_mast_dat_i[inc_j], mast_dat_i_list[inc_i-1] [inc_cnt], intcon_mast_dat_o[inc_j],
mast_dat_o_list[inc_i-1] [inc_cnt] );
                                        inc_cnt= inc_cnt+1;
                                   end
                                //NB Disable for debug
                                inc_cnt = 0;
                                for (inc_j = (inc_i-1)*`ADDR_WIDTH ; inc_j <= (inc_i*`ADDR_WIDTH-1);
inc_j = inc_j+1)
                                  begin
                                        mast_adr_i_list[inc_i-1] [inc_cnt] = intcon_mast_adr_i[inc_j];
                                        //$display ( " Master info 2 intcon_mast_adr_i : 0x%0h,
mast_adr_i_list : 0x%0h", intcon_mast_adr_i[inc_j], mast_adr_i_list[inc_i-1] [inc_cnt]);
                                        inc_cnt= inc_cnt+1;
                                  end

                                inc_cnt = 0;
                                for (inc_j = (inc_i-1)*`TAG_WIDTH ; inc_j <= (inc_i*`TAG_WIDTH-1);
inc_j = inc_j+1)
                                  begin
                                        mast_tga_i_list[inc_i-1] [inc_cnt] = intcon_mast_tga_i[inc_j];
                                        //$display ( " Master info 2 intcon_mast_tga_i : 0x%0h,
mast_tga_i_list : 0x%0h", intcon_mast_tga_i[inc_j], mast_tga_i_list[inc_i-1] [inc_cnt]);
                                        inc_cnt= inc_cnt+1;
                                  end


                                inc_cnt = 0;
                                for (inc_j = (inc_i-1)*`SEL_WIDTH ; inc_j <= (inc_i*`SEL_WIDTH-1); inc_j
= inc_j+1)
                                  begin
                                        mast_sel_i_list[inc_i-1] [inc_cnt] = intcon_mast_sel_i[inc_j];
                                        //$display ( " Master info 3 intcon_mast_sel_i: 0x%0h ,
mast_sel_i_list:0x%0h ", intcon_mast_sel_i[inc_j], mast_sel_i_list[inc_i-1] [inc_cnt]);
                                        inc_cnt= inc_cnt+1;
                                  end
                                  //$display ( "Master inc_i : %d", inc_i);
                        end
```

```
                //$display ( " Master information combined into lists, inc_i = %d  ", inc_i );
                end // end of else
        end

        //layer flattened slave's bus into a list
        always @ (posedge clk)
        begin
                if (rst)
                begin
                  for (inc_i = 0; inc_i < num_slave; inc_i= inc_i + 1)
                        for (inc_j = 0 ; inc_j < `ADDR_WIDTH ; inc_j = inc_j+1)
                                slave_dat_i_list[inc_i] [inc_j] = 1'b0; //bit wise assign
                  for (inc_i = 0; inc_i < num_slave*`DATA_WIDTH; inc_i= inc_i + 1)
                                intcon_slave_dat_o[inc_i] = 1'b0;
                  for (inc_i = 0; inc_i < num_slave*`ADDR_WIDTH; inc_i= inc_i + 1)
                                intcon_slave_adr_o[inc_i] = 1'b0;
                  for (inc_i = 0; inc_i < num_slave*`TAG_WIDTH; inc_i= inc_i + 1)
                                intcon_slave_tga_o[inc_i] = 1'b0;
                  for (inc_i = 0; inc_i < num_slave*`SEL_WIDTH; inc_i= inc_i + 1)

                                intcon_slave_sel_o[inc_i] =1'b0;
                end

                else
                begin
                 for (inc_i = 1; inc_i <= num_slave; inc_i =  inc_i+1)
                  begin
                        inc_cnt = 0;
                        for ( inc_j = (inc_i-1)*`DATA_WIDTH ; inc_j <= (inc_i*`DATA_WIDTH-1) ;
inc_j = inc_j+1)
                           begin
                                slave_dat_i_list[inc_i-1] [inc_cnt] = intcon_slave_dat_i[inc_j];
                                intcon_slave_dat_o[inc_j] = slave_dat_o_list[inc_i-1] [inc_cnt];
                                //$display ( " Slave info 1 intcon_slave_dat_i: 0x%0h,
slave_dat_i_list: 0x%0h, slave_dat_o_list: 0x%0h, intcon_slave_dat_o: 0x%0h ",
intcon_slave_dat_i[inc_j], slave_dat_i_list[inc_i-1] [inc_cnt], slave_dat_o_list[inc_i-1] [inc_cnt],
intcon_slave_dat_o[inc_j]);
                                inc_cnt = inc_cnt+1;
                           end

                //NB disabled for debug
                        inc_cnt = 0;
                        for ( inc_j = (inc_i-1)*`ADDR_WIDTH ; inc_j <= (inc_i*`ADDR_WIDTH-1)
;  inc_j = inc_j+1)
                           begin
                                intcon_slave_adr_o[inc_j] = slave_adr_o_list[inc_i-1] [inc_cnt];
                                //$display ( " Slave info 2 slave_adr_o_list: 0x%0h,
intcon_slave_adr_o: 0x%0h", slave_adr_o_list[inc_i-1] [inc_cnt], intcon_slave_adr_o[inc_j]);
                                inc_cnt = inc_cnt+1;
                           end

                        inc_cnt = 0;
```

```
                        for ( inc_j = (inc_i-1)*`TAG_WIDTH ; inc_j <= (inc_i*`TAG_WIDTH-1) ;
inc_j = inc_j+1)
                            begin
                                intcon_slave_tga_o[inc_j] = slave_tga_o_list[inc_i-1] [inc_cnt];
                                //$display ( " Slave info 2 slave_tga_o_list: 0x%0h,
intcon_slave_tga_o: 0x%0h", slave_tga_o_list[inc_i-1] [inc_cnt], intcon_slave_tga_o[inc_j]);
                                inc_cnt = inc_cnt+1;
                            end


                        inc_cnt = 0;
                        for ( inc_j = (inc_i-1)*`SEL_WIDTH ; inc_j <= (inc_i*`SEL_WIDTH-1) ;
inc_j = inc_j+1)
                            begin
                                intcon_slave_sel_o[inc_j] = slave_sel_o_list[inc_i-1] [inc_cnt];
                                //$display ( " Slave info 3 slave_sel_o_list: 0x%0h,
intcon_slave_sel_o: 0x%0h ", slave_sel_o_list[inc_i-1] [inc_cnt], intcon_slave_sel_o[inc_j]);
                                inc_cnt = inc_cnt+1;
                            end
                        //$display (" Slave inc_i: %d ", inc_i);
                    end // end for loop for inc_i...
                //$display ( " Slave information combined into lists, inc_i = %d  ", inc_i );
                end //closing else
        end
```