# Simplifying the Creation of Multi-core Processors: An Interconnection Architecture and Tool Framework

by

Samuel R. Grossman

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

The contribution of this thesis is two-fold: an on-chip interconnection architecture designed specifically for multi-core processors and a tool framework that simplifies the process of designing a multi-core processor. Both contributions primarily target ASIC fabrication, though prototyping on an FPGA is also supported. SG-Multi, the on-chip interconnection architecture, distinguishes itself from other interconnection architectures by emphasizing universal adaptability; that is, a primary design goal is to ensure compatibility with industry-supplied cores originally intended for other architectures. This goal is achieved through the use of bus adapters and without introducing clock cycle latency. SG-Multi is a multi-bus architecture that uses slave-side arbitration and supports multiple simultaneous transactions between independent devices. All transactions are pipelined in two stages, an address phase and a data phase, and for improved performance slave devices must signal their status for a given clock cycle at the beginning of that cycle. SG-Multi Designer, the tool framework which builds systems that use SG-Multi, provides a higher level of abstraction compared to other competing system-building solutions; the set of components with which a designer must be concerned is much more limited, and low-level details such as hardware interface compatibility are removed from active consideration. Experimental results demonstrate that the hardware cost of using SG-Multi is reasonable compared to using a processor's native bus architecture, although the current implementation of arbitration is identifiable as an area for future improvement. It is also shown that SG-Multi is scalable; the reference systems grow linearly with respect to the number of cores when tested for ASIC fabrication and slightly sublinearly when tested for FPGA prototyping, and the maximum achievable clock frequency remains almost constant as the number of cores grows beyond four. Because the reference systems tested are an accurate reflection of the types of systems SG-Multi Designer produces, it is concluded that the abstraction model used by SG-Multi Designer does not over-simplify the design process in a way that causes excessive performance degradation or increased hardware resource consumption.

**Acknowledgements**

I wish to thank my thesis reviewers, Dr. Catherine Gebotys and Dr. Hiren Patel, for their valuable feedback towards improving this thesis. I would also like to thank my supervisor, Dr. William Bishop, for his help and support throughout my degree program.

## Dedication

I dedicate this thesis to my wonderful grandmother, Lillian Bercuson. You have always been a special and beloved light in my life. Although you may be physically far away, remember that light can cross continents in but an instant.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The trend towards increasingly parallel computer systems marks the emergence of new research problems geared towards overcoming the performance-limiting communication bottlenecks, challenges not faced by designers of single-core processors [1]. In the simplest possible design, one would solve the basic logical contention issues and do nothing more, resulting in a system that performs only marginally better than a uniprocessor system and consequently wastes a significant amount of computing resources. Such a system would excel when faced with a set of completely independent tasks but would otherwise falter. On the other hand, one can envision an ideal parallel computing system, in which each processing element operates at its own maximum speed, and no processing element ever encounters communication delays longer than it would if it were the only processing element in the system. If it were possible to construct this type of system in a manner that scales to an arbitrary number of processing elements, the hardware architecture goals of parallel processing research would be achieved. Unfortunately, no such system exists; current research seeks to optimize parallel computing system hardware architectures to improve scalability and limit communication overhead.

As multi-core processors become increasingly pervasive [2], greater emphasis is placed on research towards improving multi-processor system architectures. In order to facilitate this type of research, it is important to be able to rapidly prototype and evaluate proposed architecture designs. Accordingly, a closely-related research area is the construction of tools and platforms capable of assisting with the implementation and verification of parallel computing architectures. These tools and platforms generally attempt to automate and

abstract away many of the lower-level details of the system's implementation so as to enable the researchers using them to focus on the higher-level design problems.

The research presented in this thesis contributes meaningfully to both of these areas. The primary goals and motivation are discussed in Section 1.1, and the main research contributions are explicitly outlined in Section 1.2. Section 1.3 describes the organization of the remainder of this thesis.

## 1.1 Goals and Motivation

The primary goal of this work is to create an integrated solution that facilitates the simple and rapid development of both homogeneous and heterogeneous multi-core processors in which cores are interconnected in a virtually arbitrary topology. While the processors built with this solution will be useful in a research setting, a key point of emphasis is ensuring that they are also specifically suitable for industrial applications.

This research contributes to two closely-related research areas: the design of a multi-core processor on-chip interconnection architecture and the creation of a tool framework to accelerate the development of complete systems. While an abundance of existing work exists in each of these research areas, none fully addresses the problems solved here; previous work emphasizes theoretical approaches that impose constraints, rendering them unsuitable beyond a research setting. For instance, proposed multi-core interconnection architectures generally require customized processing elements specifically tailored to that specific architecture, the purpose being to enable them to support the new signalling protocols or instructions introduced. Unless a vendor such as ARM adopts the proposed architecture or makes the required customizations, it is impossible for commercial products to be constructed that utilize it. In a similar fashion, existing rapid development tool sets and platforms are designed specifically for the furtherance of research due to their tendency to impose a particular type of underlying hardware on their users. Industry-designed tool sets also exist for the purpose of creating products more appropriate for commercial use, but what ultimately sets this work apart from existing solutions are the emphasis it places on high-level design and the simplicity of the abstraction model it presents to users.

The motivation for this work is to bridge the gap between multi-core architecture research and practical application. This is mainly achieved by avoiding constraints, re-

strictions, and assumptions that require infeasible modifications to existing widely-used hardware components and by building a solution that is separate from its final hardware implementation. In particular, the solution proposed here fully supports existing industry-produced processor cores and imposes no requirements on the underlying hardware, specifically targeting ASIC (application-specific integrated circuit) implementation but also supporting FPGAs (field-programmable gate arrays) as research and prototyping tools.

## 1.2 Research Contributions

The contribution of this work is two-fold: an on-chip interconnection architecture designed for parallel computing systems and a tool framework for rapidly constructing multi-core processors that make use of this architecture. Sections 1.2.1 and 1.2.2 describe each of these contributions, respectively, in more detail.

### 1.2.1 On-Chip Interconnection Architecture

The first contribution is *SG-Multi*, a scalable, general-purpose multi-core interconnection architecture and signalling protocol. The purpose of SG-Multi is to act as the underlying architecture within a multi-core processor for connecting processing elements to each other and to peripheral devices in a scalable, optimized fashion. The intent is to support both homogeneous and heterogeneous multi-core systems containing individual processor cores of various sizes and performance levels. In keeping with the goal of being suitable for use by industry, SG-Multi is designed specifically for use with existing industry-produced processor cores. Since not all such cores communicate according to the same signalling protocol, the SG-Multi signalling protocol is, to the greatest extent possible, universally adaptable; its design is governed by the notion that it must be possible to create adapter components to convert signals between it and the protocols used by each individual processor core.

At its core, SG-Multi is a multi-bus system that makes use of slave-side arbitration to allow multiple independent transactions to occur simultaneously. The physical wiring of a master device to a slave device is fixed at implementation time, but no restrictions are imposed on the designer; the set of slaves with which a particular master can com-

municate is architecturally independent of the sets of slaves with which other masters can communicate. SG-Multi also includes a performance-enhancing feature, known as "snooping," which effectively allows a slave device to service multiple incoming requests at once in some circumstances. While snooping methods are typically utilized for the purposes of maintaining cache coherency [3], SG-Multi allows them to be used for the completion of bus transactions.

A system that implements SG-Multi includes, in addition to the processing elements and peripheral devices themselves, several SG-Multi-specific hardware components. These components are listed and briefly described in Table 1.1. Chapter 3 provides a more detailed overview of SG-Multi.

Table 1.1: List and descriptions of SG-Multi core hardware components

| **Component** | **Description** |
|---|---|
| Master device wrapper | Implements the core SG-Multi-specific logic needed to connect a bus master device to an SG-Multi system. |
| Slave device wrapper | Implements the core SG-Multi-specific logic needed to connect a slave master device to an SG-Multi system. |
| Bus adapter | Required for bus master devices originally designed to use a signalling protocol other than SG-Multi. This component sits between the bus master device and the master device wrapper. |
| Arbiter | Resolves contention between bus master devices when they attempt to communicate simultaneously with a particular slave. This component is a sub-unit that exists within the slave device wrapper. |

Devices that support SG-Multi are not required to be aware of the existence of or give any consideration to the actions of other devices in the system. Master devices may be designed under the assumption that they are the only masters in the system, and similarly slave devices need not differentiate between masters. All of the logic required to route transactions correctly and handle contention is incorporated into the master and slave device wrapper components. Consequently, one of the distinguishing features of SG-Multi is its ability to create multi-core processors even out of individual cores not designed or intended for this purpose.

### 1.2.2 Multi-core Rapid Development Framework

The second contribution is a tool framework, *SG-Multi Designer*, that, given the high-level design of a multi-core processor, produces an SG-Multi implementation of the required interconnection fabric. An abstraction model serves as the basis for the tool framework, the sheer simplicity of which is the primary factor that distinguishes the tool framework from competing solutions. Experimentation will demonstrate the feasibility of providing an extremely simple abstraction model while neither over-simplifying the design process nor reducing the quality of the resulting hardware. At an extremely high level, SG-Multi Designer performs these steps in sequence when a user requests that a specified design be implemented:

1. Analyze the input design
2. Generate customized versions of each SG-Multi hardware component required to implement the design
3. Produce a top-level hardware module that connects all other hardware components according to the topology specified by the input design

SG-Multi Designer is designed to be modular. It consists of one tool for each type of SG-Multi hardware component, as listed in Table 1.1, modules representing specific devices that are available for inclusion in designs, and a unifying tool that controls the entire process and invokes the others. In essence, the lattermost completes steps 1 and 3, and the individual tools and modules each complete a portion of step 2. A fully-working processor is produced by combining the generated interconnection logic with the output of these modules, each of which represents a unit of intellectual property possibly supplied by a third-party. It is assumed that the user is in possession of the requisite intellectual property.

Input and output are human-readable to the greatest extent possible. Input designs are supplied using Extensible Markup Language (XML), and the tools each generate a Verilog file as output. A graphical front-end application serves as the unifying tool and is primarily used to simplify the process of creating XML files containing an input design, and the structure of the XML files is a reflection of the design of the abstraction model.

## 1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information on heterogeneous multi-core processors and their benefits as well as a detailed review of existing related work in the domains of research to which this thesis contributes. In addition to solutions proposed as a result of academic research, the scope of this chapter encompasses industry-produced solutions that are commercially available. Chapter 3 describes SG-Multi at a high level, illustrating how devices in a complete system connect to one another while avoiding discussion of the lower-level details. This is immediately followed by Chapter 4, which houses the technical discussion of SG-Multi. It should be noted that, whether taken individually or as a pair, neither of Chapters 3 and 4 are intended as protocol specifications; rather, they emphasize the architecture as a whole, the design of the individual components, and the rationale that underlies the signalling protocol. Chapter 5 shifts the focus from the architecture to the tool suite, outlining each component separately and demonstrating how they fit together in an integrated package. Chapter 6 describes and shows the results of experiments that were conducted as a means of evaluating SG-Multi and, by extension, SG-Multi Designer. Chapter 7 summarizes the contributions of and the conclusions drawn in this thesis and explores potential future research directions.

This thesis also includes two appendices containing supporting material. Appendix A provides a detailed protocol specification for SG-Multi. It is useful primarily as documentation for building devices that make use of SG-Multi. Appendix B defines the information exchange standards for SG-Multi Designer modules. Its purpose is to provide specifications on the content and format of the XML files that define the interface for communication with an SG-Multi Designer module.

# Chapter 2

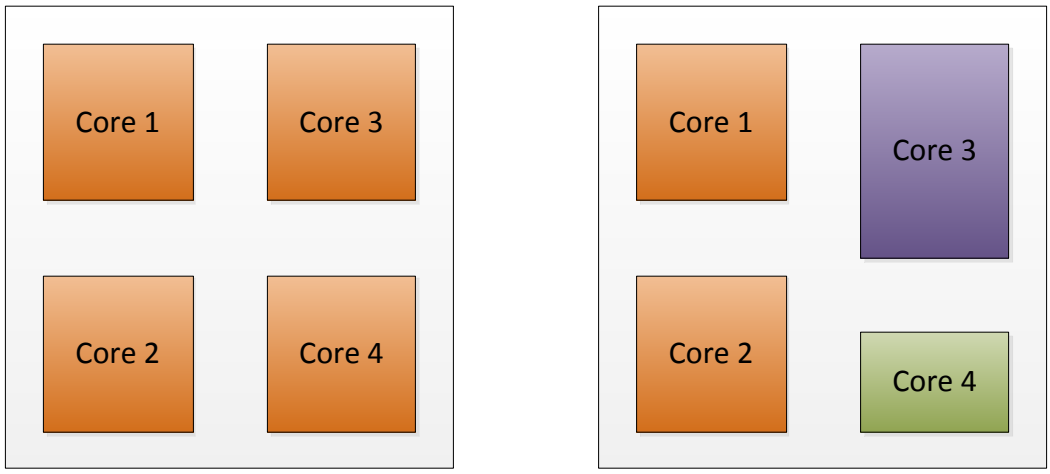
# Background

Research related to the advancement of parallel processing technology is abundant. Academic research includes projects designed to accelerate other higher-level research by abstracting away the intricacies involved with building a working multi-core processor. These works facilitate the rapid prototyping of multi-core processors by greatly simplifying the design and implementation process, in many ways offering similar functionality to that of SG-Multi-Designer. The SG-Multi signalling protocol itself lends itself well to comparison with competing industry-supplied interconnection architecture solutions, and SG-Multi Designer can similarly be compared to commercial products that simplify processor and system design.

This chapter begins by defining and differentiating between two types of multi-core processors. It then examines, in sequence, related work in each of the areas outlined previously.

## 2.1 Heterogeneous Multi-core Processors

A typical consumer-oriented multi-core processor, such as Intel's Core [4] processor family, is *homogeneous*. A defining characteristic of such a processor is that all of the individual cores are identical in design and performance [5]. By contrast, a *heterogeneous* multi-core processor contains individual cores that may vary in characteristics such as size, instruction set support, raw computational power, and special-purpose hardware optimizations. Figure 2.1 illustrates the difference between them.




(a) Homogeneous multi-core processor

(b) Heterogeneous multi-core processor

Figure 2.1: Illustration of the two different types of multi-core processors

The development and advancement of heterogeneous multi-core processors is motivated by the desire to improve efficiency and reduce power consumption. One approach towards achieving this involves integrating highly specialized processing elements, often called *accelerators*, into the design of the processor. Each accelerator is optimized to perform a small set of tasks more quickly and efficiently than a general-purpose processing element could, and accordingly the software must be designed to make use of each accelerator for its intended purpose [6]. As a result, accelerators are dependent on the existence of a general-purpose processor. The integration of accelerators allows heterogeneous multi-core processors to be custom-tailored in ways that make the results unsuitable for general-purpose computations but that greatly benefit the intended area of use. Heterogeneous multi-core processors designed in this fashion have applications in, for example, the areas of digital signal processing [7] and wireless communications [8].

Another approach stresses power consumption over performance, the aim being to reduce the system's overall power consumption while ensuring the performance penalty is as negligible as possible. This approach combines more powerful general-purpose processor cores with a less powerful—and less power-hungry—general-purpose processor core, where

all the cores support the same instruction set [9]. The assignment of tasks to processor cores happens on-the-fly. In the absence of tasks that demand high amounts of computational power, the more powerful cores are disabled, leaving functions such as those related to system management to be executed on the less powerful core. This type of heterogeneous multi-core processor has mainly seen applications in the design of processors destined for consumer mobile devices, such as NVIDIA's Tegra [10] series of mobile application processors.

## 2.2 Rapid Prototyping of Multi-core Processors

Rapid prototyping platforms greatly accelerate the process of constructing and implementing a working version of a multi-core processor. Compared to pure software simulation of a logic design, it has been shown that a rapid prototyping platform can realistically model the design and achieve 200-times speedup [11]. From a design process standpoint, no concrete figures are available as to the amount of time saved, but anecdotal evidence suggests it is possible to reduce bring-up and verification time from months to weeks [12].

Research-oriented platforms are designed to support experimentation with novel architectures and applications while also offering the performance benefits of rapid prototyping. Instead of relying on software simulators or fabricating real hardware, researchers can implement their designs on these reconfigurable platforms. Whereas commercial-grade platforms emphasize design verification and can therefore be used for general-purpose designs including those not related to multi-core processors, research-oriented platforms tend to be more special-purpose in nature. The rapid prototyping platforms presented in this section are all either designed for multi-core research or offered by industry.

A common feature of many rapid prototyping platforms is their reliance, at least in part, on FPGAs. A particularly well-known rapid prototyping platform, known as Research Accelerator for Multiple Processors (RAMP) [13], is entirely based on FPGAs. RAMP is designed to emulate multiprocessor systems in a cycle-accurate manner. It offers its own description language—RAMP Description Language (RDL)—as a way of recording a system's design such that it can be reconstructed in a way that provides cycle-for-cycle performance equivalence with the original system. In keeping with its focus on emulation, RAMP supports modeling different clock domains by allowing, on a per-component basis,

multiple physical clock cycles to correspond to a single logical clock cycle. For instance, a particular component may be clocked such that each physical clock cycle advances its emulated clock by one cycle, whereas a different component may require two or three physical clock cycles to advance its emulated clock by a single cycle. As with SG-Multi, RAMP supports the use of existing industry-supplied processor cores.

A more specialized rapid prototyping platform is the Flexible Architecture for Research Machine (FARM) [14]. Existing prototype systems that use RAMP, such as RAMP White [13] and RAMP Blue [15], are homogeneous multi-core processors, though RAMP does not specifically impose this requirement. FARM, on the other hand, is designed with heterogeneous multi-core processor applications in mind and, also unlike RAMP, does not implement all hardware components in FPGAs. The goal of FARM is to prototype multiprocessor systems consisting of multiple high-performance general-purpose processors connected to an FPGA that implements a hardware accelerator, where the FPGA includes a cache and participates in the same system-wide cache coherency protocols as do the general-purpose processors. FARM is less flexible than RAMP in that it specifies the overall topology of the system and limits reconfigurability to the FPGA part of the system, keeping the rest fixed. System performance would also be less indicative of that of real hardware, since timing behaviour is quite different when real processors interact with an FPGA than when real processors interact with a real accelerator.

A somewhat older rapid prototyping platform, which to a certain extent provides the basis for FARM [14], is the Rapid Prototyping Engine for Multiprocessors (RPM) [16]. Whereas both RAMP and FARM make use of FPGAs for implementing at least some of the processors in the system, the defining characteristic of RPM is that the processors are real off-the-shelf hardware components while the FPGAs are used for the caches, memory controllers, and other support elements. RPM fixes the processors and interconnection topology while allowing FPGA-controlled components to be customized; it is not a truly general-purpose rapid prototyping platform, but it does allow basic experimentation with, for example, different memory hierarchy configurations.

The rapid prototyping platforms described thus far are useful because they greatly accelerate the process of constructing a multiprocessor system that can be used for research activities such as experimentation. They do not, however, produce production-ready systems suitable for chip fabrication. Cadence makes available an FPGA-based commercial

rapid prototyping platform designed specifically for industrial use [17]. While such a system could theoretically be used for research, it is intended to accelerate the bring-up process for new system-on-chip designs that will ultimately be fabricated and sold commercially. Cadence provides a complete and comprehensive solution that encompasses all steps occurring after the chip's logic has been designed: compilation, partitioning across multiple FPGAs, insertion and configuration of debugging probes, and execution. Unlike the previously-introduced rapid prototyping platforms, Cadence's solution is not specifically optimized for multi-core processor design and can therefore be used in a more general-purpose fashion. However, it targets the verification portion of the design process, whereas the other rapid prototyping platforms target the entire process, from high-level design to verification.

In the rapid prototyping platform space, SG-Multi Designer fits between the research solutions (RAMP, FARM, and RPM) and the commercial one (Cadence), as represented visually in Figure 2.2. It produces complete systems that, when connected to processor cores and peripheral devices, can be downloaded to FPGAs and used either for research or for design verification. It is also the only of the systems discussed that focuses on the "high-level design" stage, as opposed to the "verification" stage, of the hardware design process.

Designs that make use of RAMP, FARM, or RPM are custom-tailored to those specific systems and require the hardware upon which those systems are built, whereas SG-Multi Designer imposes no hardware requirements. Similarly, unlike Cadence's platform, SG-Multi is not tied to a specific set of hardware and software tools; rather, users are free to select hardware and software vendors of their choice when implementing their SG-Multi system.

## 2.3 Existing Related Interconnection Architectures

Several existing interconnection architectures share design features with SG-Multi. They vary in their intended uses, particularly in terms of flexibility, scalability, and suitability for use as the basis of a multi-core processor. SG-Multi is motivated by existing signalling protocols but still contains features to distinguish itself; some of the most comparable existing work, all of which comes from industry, includes Altera's Avalon-MM, ARM's AHB-Lite, and IBM's CoreConnect. After discussing networks-on-chip, a modern but

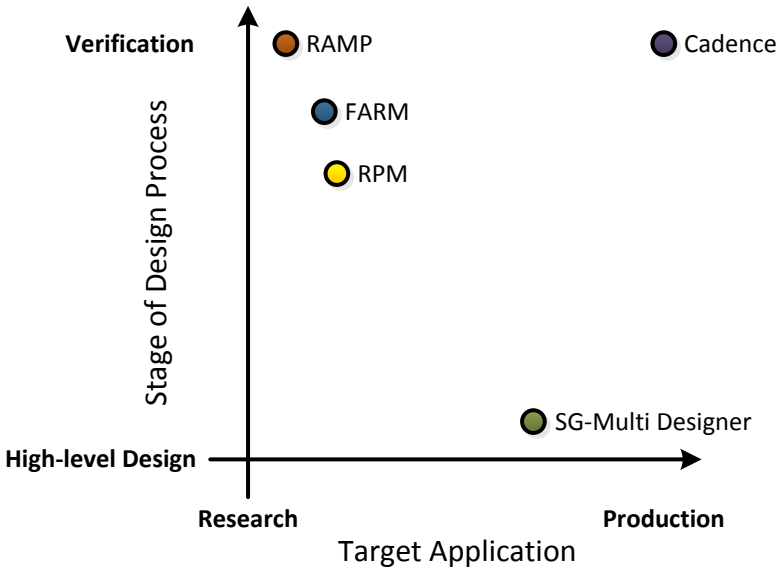



Figure 2.2: Map of application domains of various rapid prototyping platforms

indirectly related interconnection architecture paradigm, this section examines each of these interconnection architectures in sequence.

### 2.3.1 Networks-on-Chip

Unlike traditional system interconnection architectures which emphasize point-to-point connections between devices, a *network-on-chip* (NoC) shifts the interconnection architecture paradigm towards that of distributed, networked systems. The distinguishing characteristic of an NoC is that it employs packet-switching techniques to move data between components [18]. NoC-type architectures fall into a tangential but unrelated research area and strive to bring the scalability and performance characteristics inherent in large networked systems to the level of a single chip. Research progression typically involves overcoming problems related to implementation complexity of networking protocols and algorithms, which leads to power and performance penalties.

NoCs currently have a wide range of applications, many of which are research-oriented

and target FPGA devices. For example, an NoC exploration study is documented in [19], with specific emphasis on FPGAs. There are, however, also commercial NoC applications; Altera's latest system construction tool, Qsys, is based on a NoC-type architecture [20]. SG-Multi more closely resembles a traditional system interconnection architecture than it does an NoC and therefore is not faced with the same research challenges that an NoC designer must overcome.

### 2.3.2 Altera Avalon

Altera makes available several different variants of its Avalon interface specifications, including Avalon-ST for streaming and Avalon-MM for typical master/slave memory-mapped configurations [21]. Because SG-Multi is based on memory mapping, the latter is more directly relevant to it.

Avalon-MM's interconnection architecture has traditionally been based on a design similar to that of SG-Multi, in which independent transactions can proceed simultaneously. This performance-enhancing ability distinguishes Avalon-MM from single-bus architectures with centralized arbitration schemes, which permit only a single transaction at any given time. Support for simultaneous transactions is achieved by connecting each master directly to each slave, using multiplexors to ensure signals are routed between the correct devices, and performing slave-level arbitration instead of system-level arbitration [22]. Figure 2.3, adapted from [22], provides a simple example of a two-master system based on Avalon-MM; while the architecture supports the use of a tri-state bridge for communication with off-chip slave devices, only the on-chip interconnection portion is shown. Further details can be found in [21] and [22]. Because arbiters are inserted only as needed at the slave ports, it is possible for the processor to communicate with SRAM while the DMA controller communicates simultaneously with SDRAM.

Table 2.1 lists some of the most common Avalon-MM signals used in basic transactions [21]. Basic transactions in Avalon-MM take exactly one clock cycle each, but slave devices not capable of responding to requests in the same cycle as they are issued have the ability to extend the transaction to multiple cycles. This can be accomplished by setting a fixed wait time as a property of the slave or by using the `waitrequest` signal, which allows for a variable wait time. Pipelined transactions are also supported, causing each transaction to complete in at least two cycles and allowing each slave to have a variable

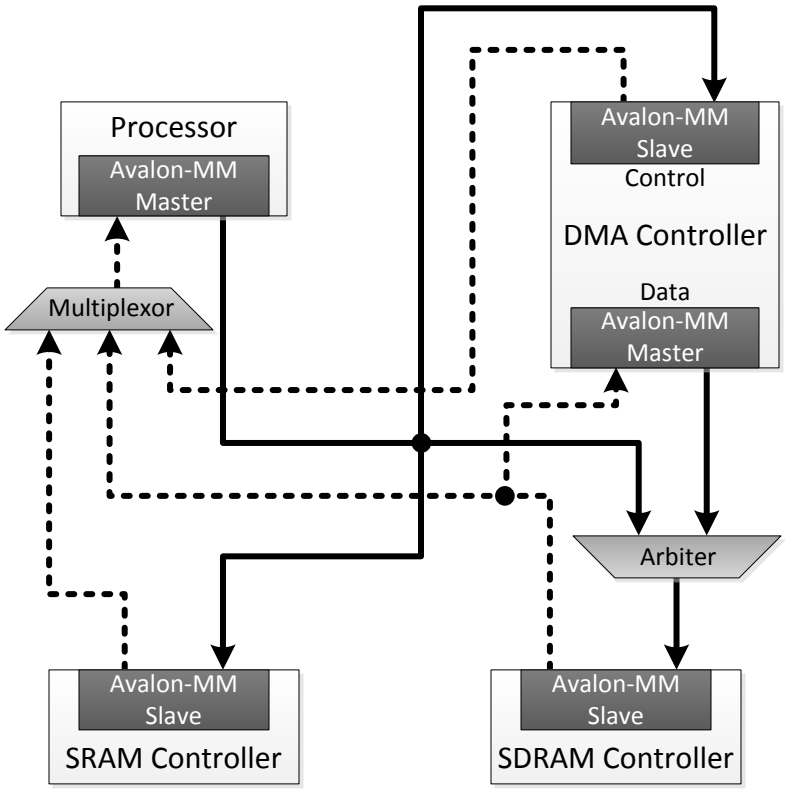



Figure 2.3: Two-master example of a system that uses Avalon-MM

number of outstanding transaction requests (the actual number is a property of an individual slave). A pipelined transaction is effectively split into two distinct phases: *address phase*, the first phase during which address and control information is supplied to the slave, and *data phase*, the second phase during which the slave processes the request. In pipelined transactions, both `waitrequest` and `readdatavalid` are used to indicate whether or not the slave has finished its processing, the former being used as in the unpipelined case to indicate delays and the latter being used specifically for read transactions [21].

Figure 2.4 shows an example of a set of simple unpipelined Avalon-MM transactions. The `waitrequest` signal is the only means by which a slave can extend a transaction beyond a single clock cycle, and a slave wishing to assert it is required to do so prior to

Table 2.1: List and descriptions of common Avalon-MM signals

| Signal Name | Description |
|---|---|
| `clk` | Clock signal used to drive transactions |
| `address` | Memory address of interest, supplied by master |
| `byteenable` | Mask to specify which of the bytes within the data bus are used in the current transaction |
| `read` | Specifies that the current transaction request is for the slave to read data and supply it to the master |
| `write` | Specifies that the current transaction request is for the slave to accept data from the master |
| `readdata` | Data resulting from a read operation, supplied by slave |
| `writedata` | Data to be written during a write operation, supplied by master |
| `waitrequest` | Indicates whether or not the slave needs more cycles to complete the current transaction |
| `readdatavalid` | For slaves that support pipelined transactions, indicates whether or not the slave has completed a read transaction |

the end of the clock cycle in which it receives its original transaction request [21].

Pipelined Avalon-MM transactions make use of the same set of signals, with the addition of read transactions which additionally use `readdatavalid`. Because pipelined Avalon-MM transactions are similar to standard transactions in the ARM AHB-Lite interconnection architecture, which is introduced in Section 2.3.3, an example is not shown here.

SG-Multi shares many design features with this version of Avalon-MM, including the use of slave-side arbitration, the pipelining of transactions, and the requirement that all slaves signal their wait requests immediately. Unlike Avalon-MM, however, it does not support unpipelined transactions, nor does it allow slaves to specify a fixed number of wait states; all transactions are separated into phases, and all slaves must make use of wait request signalling. Whereas Avalon-MM is designed primarily to be integrated with Altera software and used with its other intellectual property products, such as its NIOS II family of soft-core processors [23], SG-Multi's design emphasizes universal adaptability, allowing it to connect without loss of performance with master devices designed for other signalling protocols, including Altera-MM.

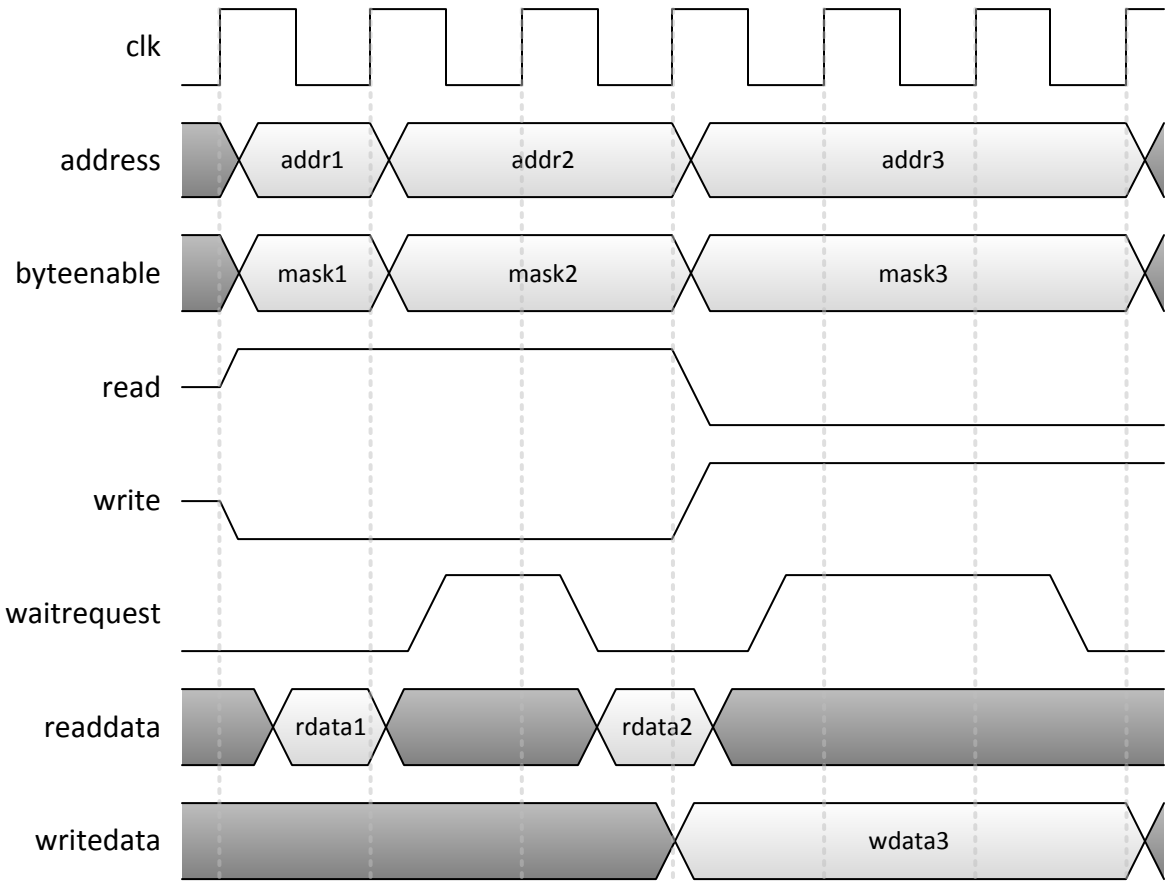



Figure 2.4: Examples of unpipelined Avalon-MM read and write transactions

### 2.3.3 ARM AHB-Lite

AHB-Lite is part of version 3 of ARM's Advanced Microcontroller Bus Architecture (AMBA) set of interconnection architecture standards [24]. Originally published in 2006, the AHB-Lite protocol specifications continue to be implemented even in some of ARM's more modern processors, particularly in those designed for simplicity and low power consumption, such as the ARM Cortex-M0 [25].

AHB-Lite is a stripped-down, simplified version of ARM's Advanced High-performance Bus (AHB) architecture, which was part of version 2 of AMBA [26]. AHB was originally designed to support multiple bus masters in a shared-bus structure using a central arbiter to determine which master controls the bus at any given time. The most significant difference between AHB and AHB-Lite is that the latter removes all of the multi-master support from AHB and operates on the premise that there is only one bus master in the system. A description of the most common AHB-Lite signals is provided in Table 2.2 [24].

Table 2.2: List and descriptions of common AHB-Lite signals

| **Signal Name** | **Description** |
|---|---|
| `HCLK` | Clock signal used to drive transactions |
| `HRESETn` | Active-low system-wide reset signal |
| `HADDR` | Memory address of interest, supplied by master |
| `HSEL` | Activation signal, sent from decoder to slave based on `HADDR` |
| `HSIZE` | Specifies the size of the current transaction, such as byte, word, or double-word |
| `HWRITE` | Specifies whether the current transaction is for reading (low) or writing (high) |
| `HTRANS` | Specifies the type of the current transaction |
| `HREADY` | Slave response signal used to indicate that the requested transaction is complete |
| `HRESP` | Slave response signal used to indicate that the slave encountered an error |
| `HRDATA` | Data resulting from a read operation, supplied by slave |
| `HWDATA` | Data to be written during a write operation, supplied by master |

In addition to master and slave components, AHB-Lite requires the use of a system-wide address decoder and a slave response multiplexor. Figure 2.5, adapted from [24],

shows a typical AHB-Lite system, including the decoder and the multiplexor components; for simplicity, some of the signals in Table 2.2 are omitted. The decoder generates exactly one `HSEL` signal based on `HADDR`, which in turn activates exactly one slave to respond to the transaction request. This is necessary because master-to-slave signals are broadcast to all slaves. The decoder also generates the multiplexor's selection signal. The multiplexor's purpose is to route the slave response signals from the active slave to the master.

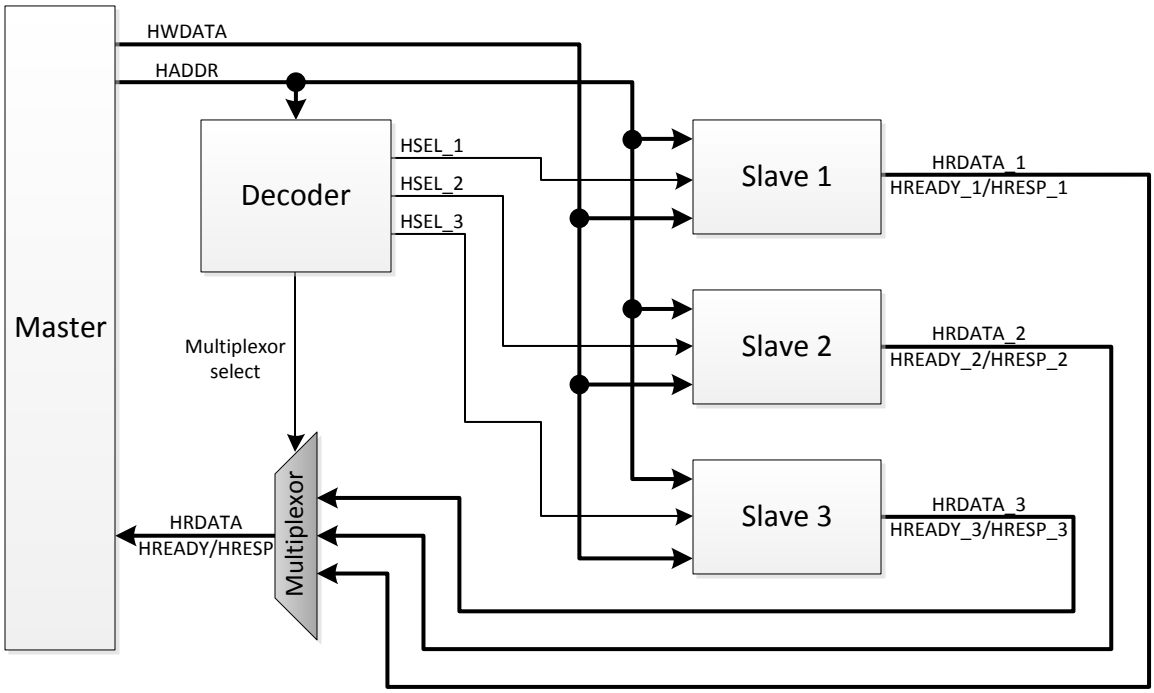



Figure 2.5: Example of a typical AHB-Lite system

All AHB-Lite bus transactions are pipelined; they are separated into two stages, the first being the *address phase* and the second being the *data phase*. Slaves use the `HREADY` signal to extend a data phase beyond just one cycle, which has the effect of extending the address phase of the next transaction. Figure 2.6 shows two back-to-back AHB-Lite transactions, the first a write with no wait states and the second a read with one wait state.

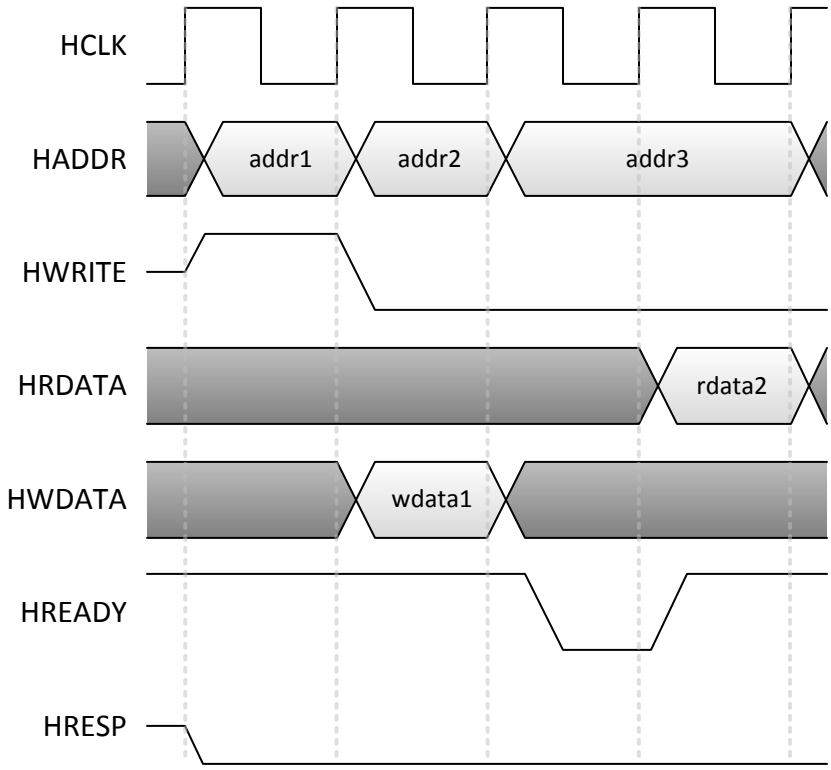



Figure 2.6: Two simple AHB-Lite transactions, one with a wait state

AHB-Lite also allows slaves to signal errors in the event that a transaction cannot be completed for some reason. A two-cycle error response is required, making use of both `HREADY` and `HRESP`. Figure 2.7 illustrates the error signalling functionality of AHB-Lite. Errors are signalled over two cycles in the data phase; in both cycles `HRESP` is held low, but the slave must hold `HREADY` low for the first cycle, effectively inserting a wait state.

The SG-Multi signalling protocol is based loosely on that of AHB-Lite; a comparison of transaction timing diagrams between AHB-Lite and SG-Multi would reveal a general similarity in terms of how they proceed. For instance, all SG-Multi transactions are pipelined, divided into the same address and data phases used in AHB-Lite. Because the system architecture of SG-Multi intrinsically incorporates some of the less-used AHB-Lite signal functionality, such as that offered by the most significant bit of `HPROT` [24], equivalent

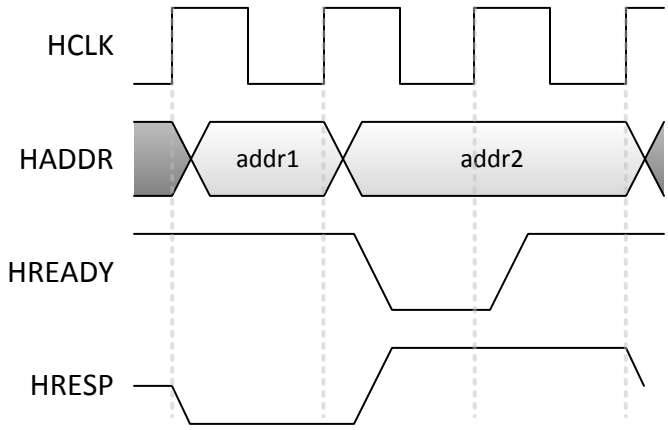



Figure 2.7: Illustration of AHB-Lite error signalling functionality

SG-Multi signals do not exist. Other SG-Multi signals, such as the equivalent of `HREADY`, have been modified for performance optimization purposes.

AHB-Lite can be extended to multiple levels, allowing multiple masters to exist in a single system. Multi-level AHB, an extension to AHB, is not a protocol specification in-and-of itself; rather, it is a set of ideas for extending AHB to more advanced configurations that overcome the limitations of a shared-bus system with centralized arbitration. One of the primary goals is to allow multiple masters to access independent slaves simultaneously [27]. The result is a system, based on the AHB and AHB-Lite protocols, that uses slave-side arbitration to handle contention at each slave but otherwise resembles an Avalon-MM system and, as a result, is also similar in overall design to an SG-Multi system. SG-Multi differentiates itself by its modified signalling protocol in ways that improve performance and facilitate the creation of bus adapters to enable support for a wide range of master devices, not just those that use AHB or AHB-Lite.

### 2.3.4 IBM CoreConnect

IBM makes available three architectures as part of its CoreConnect system: Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register (DCR) bus. PLB attaches directly to the processor cores and facilitates communication with slaves. OPB is

intended for low-speed peripheral devices, which communicate with PLB devices through bridges. DCR supports transactions involving accesses to control and status registers on the devices in the system; its use allows trivial administrative tasks such as these to be offloaded from PLB [28]. Of primary interest is PLB since it is the bus that ultimately connects directly to processor cores.

CoreConnect is by no means a new architecture. It is mostly used within IBM's own processor offerings, typically based on the POWER architecture, and until recently has been the architecture of choice for Xilinx's MicroBlaze soft-core processors [29]. Figure 2.8 shows a simplified layout of a system that uses CoreConnect. Typically such a system would include OPB bridges and DCR connections, but these have been omitted to emphasize the PLB portion.

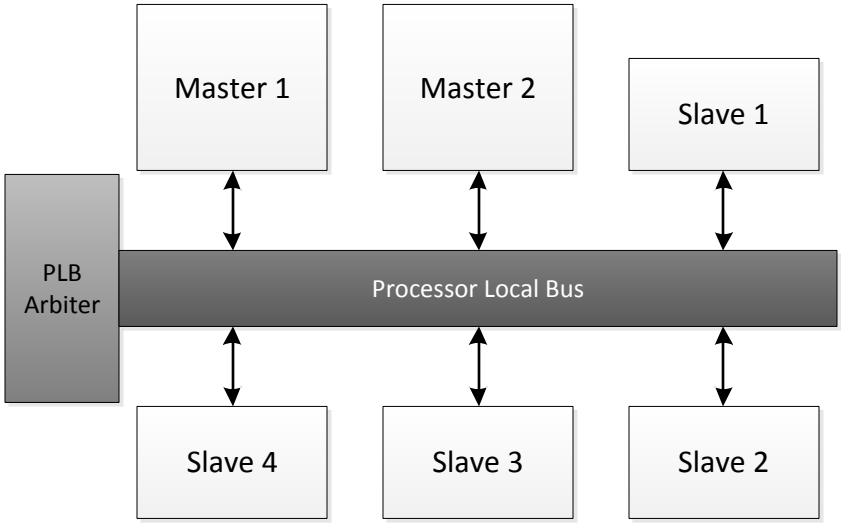



Figure 2.8: Simple example of a system based on IBM CoreConnect PLB

The CoreConnect architecture as a whole is quite different from those of Avalon and AHB-Lite, primarily due to its reliance on central arbitration, which effectively prohibits multiple transactions from occurring simultaneously between independent masters and slaves. In this regard Avalon, AHB-Lite, and SG-Multi all offer greater flexibility and, when used in a situation that involves multiple master devices, greater potential performance.

A key observation, however, is the fact that a CoreConnect master device may perform uninhibited when there are no other master devices in the system. Consequently, some of CoreConnect's inherent limitations could be overcome through the creation of a bus adapter device to allow a CoreConnect master to communicate with other devices on a system such as SG-Multi. A bridge for connecting AHB-compliant slaves to a CoreConnect system already exists [30], which demonstrates the feasibility of such an endeavour.

As CoreConnect is a mature architecture by comparison to SG-Multi, it supports numerous advanced features not currently supported by SG-Multi, such as the masking of bytes to allow unaligned or odd-sized transfers and parity protection to ensure transfer integrity. These features are discussed in [28] and are beyond the scope of this thesis. While not of particular interest in the context of a comparison to SG-Multi, the DCR bus also forms an important part of a CoreConnect system; the AHB bridge includes appropriate DCR functionality [30], and any SG-Multi bus adapter for CoreConnect would similarly need to do so.

## 2.4 Existing Related System Construction Tools

SG-Multi Designer provides the functionality required to construct a complete SG-Multi system while presenting its user with a high level of abstraction to simplify the process of doing so. Competing system construction tools of various forms also exist to assist a designer wishing to construct other types of systems. Common functionality shared by many system construction tools includes the ability to generate an interconnection architecture, instantiate it, and connect all system modules to it, though many tools provide additional functionality such as the customization of individual devices in the system. While academic solutions exist to provide some of this functionality, such as [31], the most complete—and therefore directly comparable—solutions all come from industry, examples of which include Altera Qsys (formerly SoPC Builder), Xilinx Platform Studio, and Tensilica Xtensa Processor Developer's Toolkit. Each of these will be examined in sequence.

### 2.4.1 Altera Qsys and SoPC Builder

Altera provides a system integration tool along with its hardware design software suite, Quartus II [32]. Both Qsys and SoPC Builder provide similar functionality, and both support Altera's intellectual property modules, known as "Megafunctions." Qsys was introduced in recent versions of Quartus II as the successor of SoPC Builder. Both tools rely on the same Avalon signalling interfaces described in Section 2.3.2, but SoPC Builder relies on an underlying architecture as described in that section, whereas the Qsys interconnect represents a shift to an FPGA-optimized network-on-chip [20].

The interfaces presented to the user are similar in both Qsys and SoPC Builder. As is the case with SG-Multi Designer, users are able to add devices to and remove devices from a design, specify connections between devices, and edit an individual device's properties. Compared to SG-Multi Designer, the level of abstraction is lower in Altera's tools; details of the underlying Avalon interfaces are exposed to the user, such as through timing diagrams [32], and Altera's abstraction model only supports devices that make use of these Avalon interfaces. While it is possible to add non-Avalon devices to the system, a user must add the requisite bridge components manually.

Altera's NIOS II family of soft core processors is a particularly common example of a device that supports the Avalon interface [23]. A designer who wishes to build a system containing NIOS II processors typically uses Qsys or SoPC Builder to do so [33]. While these tools support Megafunctions beyond those supplied by Altera and can work with other types of processors, the emphasis of these tools is building systems that use NIOS II processors. Furthermore, during the design process a user is asked to pick a target Altera device, and the system compilation process may optimize the output for that particular device. Hence, Qsys and SoPC Builder primarily target Altera devices for implementing the output hardware. This is unlike SG-Multi Designer, which is technology-independent and vendor-agnostic. The underlying architecture is designed specifically to support intellectual property modules from different vendors, and SG-Multi Designer provides the same interface and level of abstraction for all of them. Users are also not asked to select a target device or technology, and the output is pure HDL code which can be compiled by virtually any set of tools.

### 2.4.2 Xilinx Platform Studio

Xilinx offers a system construction tool similar to those offered by Altera, known as Xilinx Platform Studio [34]. The supported underlying architectures are IBM CoreConnect PLB [28] and ARM Advanced Extensible Interface (AXI) [35], allowing designers to create systems with not only Xilinx MicroBlaze soft core processors [29] but also numerous other industry-supplied intellectual property cores. Many higher-end processors, including the most recent revisions of MicroBlaze and the latest ARM processors, support AXI, greatly extending the scope of applicability for designs created in Platform Studio compared to designs created in Altera Qsys or SoPC Builder.

Platform Studio offers a similar level of abstraction to that of Altera Qsys and SoPC Builder. Users are able to add, remove, connect, disconnect, and otherwise customize devices in their system designs. Unlike SG-Multi Designer, however, the underlying interfaces remain a concern; users must manually add any bridges to the system should they wish to add a device that does not support AXI or CoreConnect PLB, and users are also required to differentiate between systems and components based on one architecture versus the other.

Xilinx markets Platform Studio for use specifically with its FPGA devices [34]. Although output is available in HDL form, the intention is for users to compile this HDL using Xilinx's tools and download the output to a Xilinx FPGA. As with Altera, Xilinx provides this tool to support its own products, whereas SG-Multi Designer separates hardware design from target implementation and offers a more flexible approach not tied to a specific vendor.

### 2.4.3 Tensilica Xtensa Processor Developer's Toolkit

Tensilica's system construction tool, known as Xtensa Processor Developer's Toolkit [36], emphasizes the creation of complete system solutions, including both hardware and software, based on its family of Xtensa processors. Unlike both Altera and Xilinx, whose tools primarily target their own respective FPGA devices, Tensilica's solution provides output appropriate for fabrication as an ASIC.

Xtensa processors are highly-customizable cores intended for a wide variety of applications, the goal being to enable hardware designers to produce application-specific Xtensa-

based systems [37]. The primary function of the Xtensa Processor Developer's Toolkit is to facilitate the customization of these Xtensa processors; configuration options range from the inclusion or exclusion of particular function units to using Tensilica's Instruction Extension (TIE) language to define new instructions [38]. To accommodate the large hardware variability between individual designs, including potential instruction set differences, Xtensa Processor Developer's Toolkit includes a customized software development toolchain in its output.

Xtensa Processor Developer's Toolkit targets Data Processing Units (DPU) in systems that consist of processors and executable code [36]. As a result, the abstraction model it employs is geared towards microarchitecture design and software development; there is no need to consider the interconnections between masters and slaves. Thus, instead of focussing on the interconnection details, a hardware designer customizes individual processor cores, specifies the memory map layout for each core included in the system, programs extensions to the instruction set, and develops the software to be executed. The focus of Xtensa Processor Developer's Toolkit is very different from that of SG-Multi Designer; in the case of the latter, while DPUs for use in processor-only systems are supported, the application domain is wider due to the inclusion of slaves as peripherals and the primary emphasis being on specifying the interconnections between devices. Additionally, Xtensa Processor Developer's Toolkit is only capable of producing systems based on the Xtensa architecture, whereas SG-Multi Designer is less restrictive in that it supports hardware from a wide variety of vendors.

# Chapter 3

# System Overview

The SG-Multi architecture is specifically geared towards accelerating communication between devices in a multi-core processor. Simultaneous inter-device communication is made possible through the use of direct master-slave connections and slave-side arbitration. However, there is no requirement that all masters be connected to all slaves in a fully-connected fashion; the set of slaves with which a particular master can communicate directly is architecturally independent of the sets of slaves to which other masters are connected.

The purpose of this chapter is to present a high-level overview of an SG-Multi system; lower-level signalling protocol details are left for Chapter 4. This chapter begins by illustrating the composition of an entire SG-Multi system, subsequently examining each part individually.

## 3.1 High-Level Description

An SG-Multi system is built from a combination of the components listed in Table 1.1 and actual devices. A *master device* initiates transactions and issues commands, whereas a *slave device* provides service. Master and slave device wrappers, respectively, provide the SG-Multi-specific functionality needed to connect them to the SG-Multi interconnection fabric. In keeping with SG-Multi's design goal of universal adaptability, *bus adapters* facilitate communication with master devices originally not designed to comply with the SG-Multi protocol specification. *Arbiters* are required at each slave to which multiple master devices connect. These devices resolve any contention issues that arise between

masters competing for simultaneous access to the same slave.

Individual devices in an SG-Multi are themselves designed with no knowledge of or consideration for the configuration of the system or even the fact that other devices exist within it. For instance, a processor core that communicates with an SG-Multi system is expected to assume itself to be the only master in the system and therefore make no attempt to synchronize accesses with other master devices. Similarly, a memory controller slave device need not identify the master device making a particular request. The hardware logic that enables devices configured in this manner to cooperate with one another is encapsulated separately in the master and slave device wrappers. A device wrapper provides all of the logic required to route transaction requests and responses as well as resolve contention. Figure 3.1 illustrates the separation between SG-Multi device and SG-Multi device wrapper. Separate wrappers exist for master devices and slave devices, each offering functionality specific to the type of device. Arbiters are enclosed within the slave wrapper component, thus achieving contention resolution without requiring slave devices to be aware of this process. For ease of illustration, shapes and colours are assigned to each component of the system, and these general notation conventions will be adopted for the remainder of this thesis.

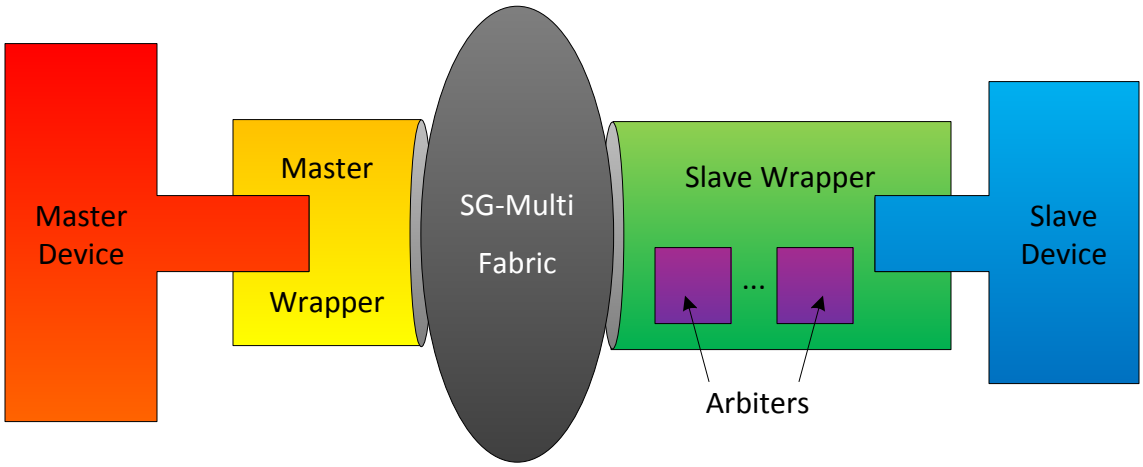



Figure 3.1: Overview of an SG-Multi system, illustrating the role of device wrappers

In addition to the components shown, a bus adapter might be present if it is a functional dependency for a particular master device. A bus adapter itself merely translates signals from one protocol to another; when considering a master device that requires a bus adapter, the bus adapter and master device are considered inseparably coupled together. Therefore, a master device that requires a bus adapter also requires a wrapper. Figure 3.2 provides a visual representation of an example SG-Multi system, in the same style as Figure 3.1, but this time showing the role of the bus adapter.

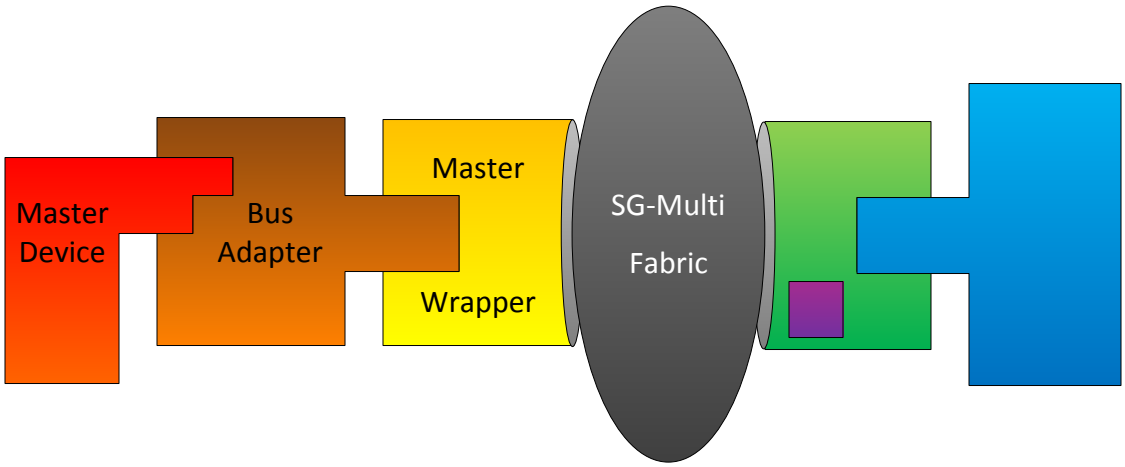



Figure 3.2: Overview of an SG-Multi system, illustrating the role of bus adapters

## 3.2 SG-Multi Device Wrappers

Two types of device wrappers exist in an SG-Multi system: one for master devices and one for slave devices. Both wrappers ensure signal timing complies with the SG-Multi protocol specifications, but each wrapper additionally performs functions appropriate for the type of device.

At a very high-level, a master device wrapper performs two tasks: transaction routing and snooping. Based on the input supplied by the master device, the wrapper identifies the slave with which to communicate and ensures both that the transaction request reaches

that slave and that the master device receives the slave's response. SG-Multi snooping emulates a successful transaction when another master completes a sufficiently similar transaction with the same slave. This is a performance-enhancing technique that can only be completed in very specific circumstances but, when possible, allows multiple masters to complete transactions with the same slave simultaneously. Figure 3.3 shows the high-level functionality of a master device wrapper for a master device connected to three slaves.

Slave device wrappers have a narrower scope of responsibilities. The primary purpose of a slave device wrapper is contention resolution through arbitration, though as a result transaction routing also becomes important. For slaves connected to multiple master devices, arbitration is the first step towards initiating a transaction; slave wrappers use the arbitration result to determine the master device from which transaction input should be accepted. Figure 3.4 illustrates the high-level functionality of a slave device wrapper for a slave connected to three masters.

Device wrappers form the core elements of the SG-Multi interconnection architecture, implementing all of the specific SG-Multi functionality that allows complete systems to be designed and built. They offer no functionality beyond what has been described, and the connections between device wrappers are simple wires, but in essence they comprise the SG-Multi interconnection fabric itself.

## 3.3 SG-Multi Arbiters

Arbiters are contained within slave device wrappers, the number per wrapper being determined by the number of master devices that connect to that particular slave. Each arbiter is a purely combinational circuit that connects both to a series of arbitration-specific common wires supplied by the slave wrapper and to "request" and "grant" lines specific to the particular master associated with that arbiter. While SG-Multi does not inherently restrict the arbitration scheme, all arbiter units within the slave wrapper must implement the same scheme. A reference arbiter, integrated into SG-Multi Designer and used to capture experimental results, employs a combination of static and dynamic priority. Each master is assigned a static priority level at design time. Dynamic priority is adjusted based on the number of times a particular master has lost arbitration, reset when that master wins and incremented otherwise up to a configurable but pre-determined maximum. The

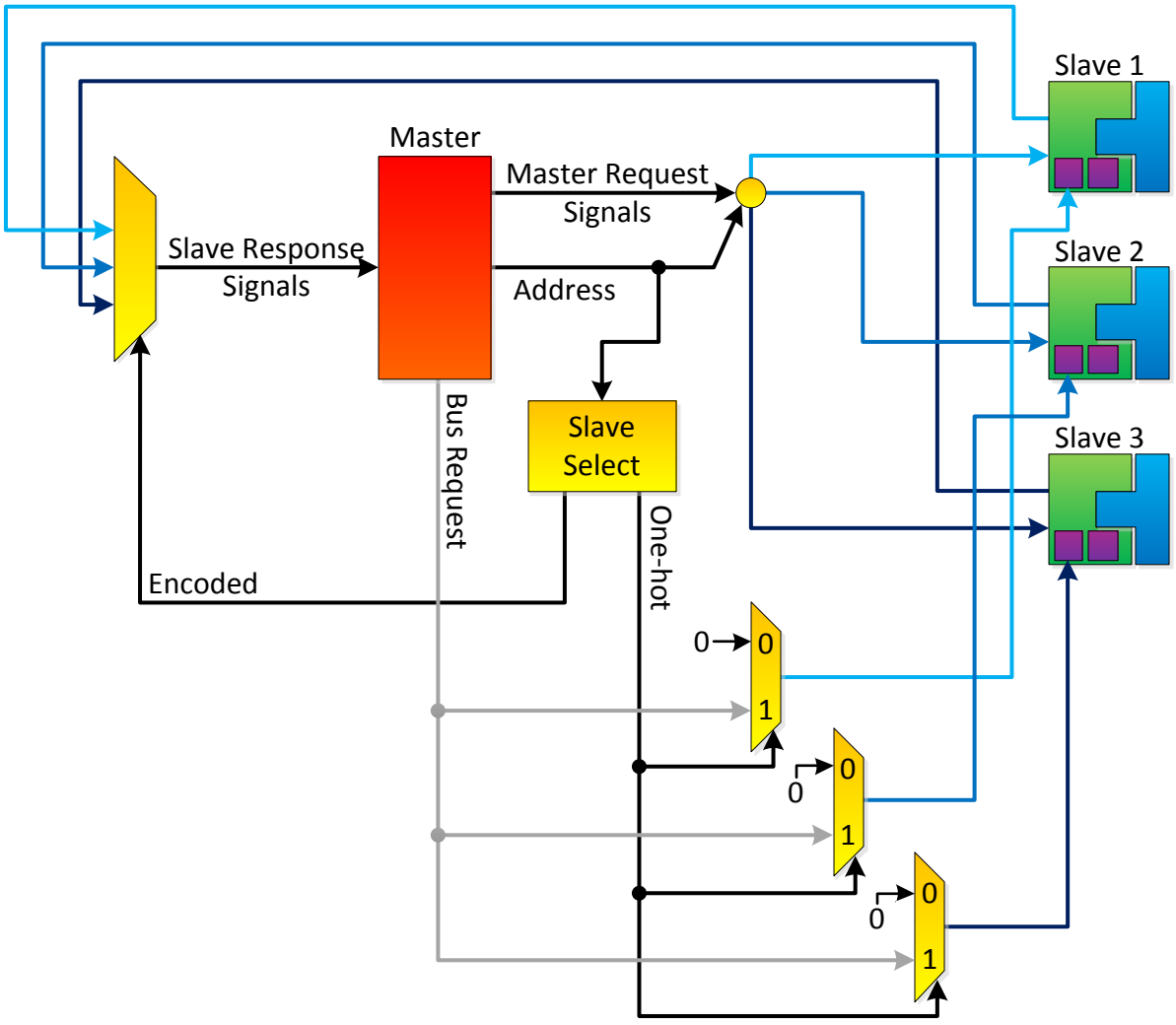



Figure 3.3: Hardware components that encapsulate a master device in SG-Multi

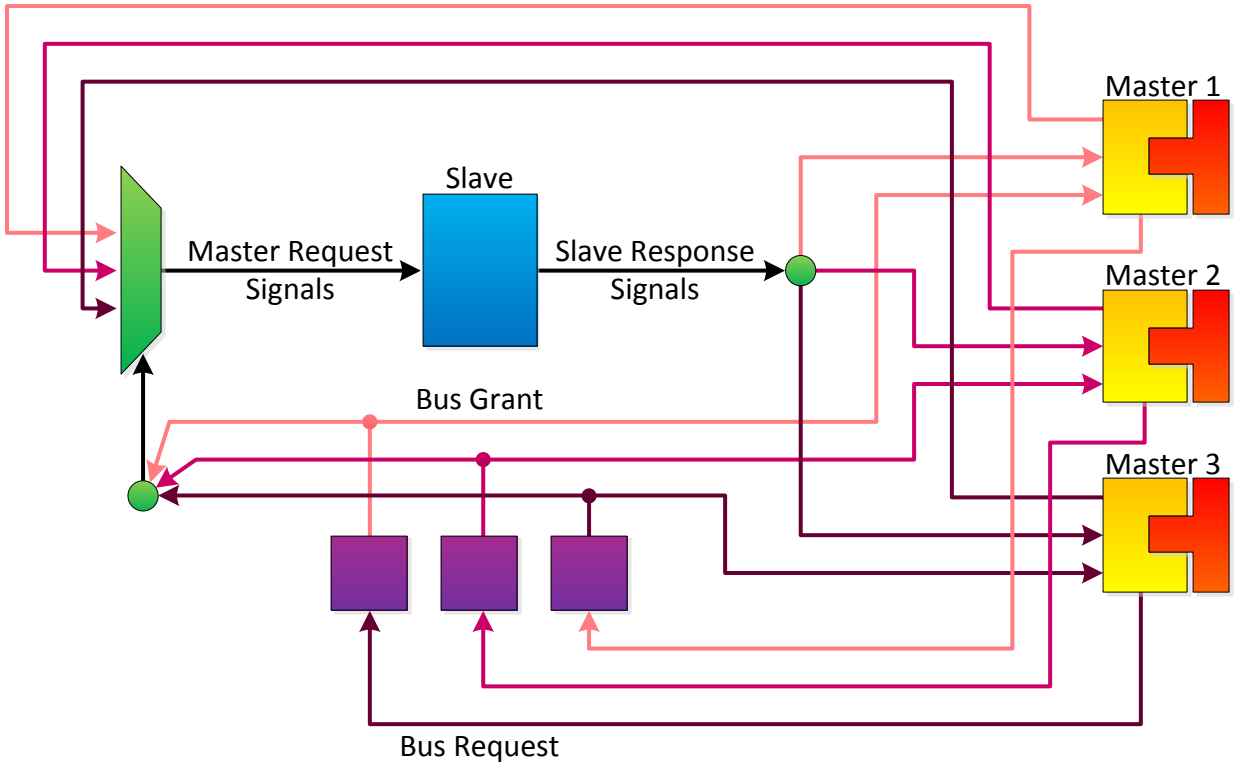



Figure 3.4: Hardware components that encapsulate a slave device in SG-Multi

winner is the master with the greatest level of dynamic priority; static priority level is used to resolve cases in which multiple competing masters are tied.

The number of static priority levels depends on the number of masters connected to a particular slave, but the number of dynamic priority levels is a design-time configurable parameter. If all connected masters wish to communicate with the slave all the time, then the arbitration scheme essentially becomes round-robin. The opposite can theoretically be achieved by setting the number of dynamic priority levels to 0, which implements an arbitration scheme that selects winners according to static priority levels.

# Chapter 4

# Architecture Design

This chapter describes SG-Multi's design in detail, including aspects of the signalling protocol and the hardware that implements the SG-Multi interconnection architecture. This includes the design of the signalling protocol itself, master device wrappers, slave device wrappers, and arbiters. Particular emphasis is placed on showing how the design decisions made contribute to the achievement of SG-Multi's design goals and the implementation SG-Multi-specific features.

Appendix A contains a complete protocol specification for SG-Multi. A basic familiarity with it is recommended, as this chapter does not provide the same level of protocol-related detail.

## 4.1 Basic Signal Definitions

Devices in an SG-Multi system communicate according to SG-Multi's signalling protocol. The interfaces of interest to an SG-Multi hardware device designer sit between the master and slave devices and their respective wrappers. Other signals are contained entirely within the interconnection fabric and are not exposed to any devices. SG-Multi also defines a small number of global signals, connected to each component in the system, including both wrappers and their associated devices. The subsections that follow list and briefly explain the purpose of each SG-Multi signal. Since reference system design and implementation made extensive use of the ARM Cortex-M0 processor [25], as detailed in Chapter 6, signal names and purposes are somewhat similar to those used in AHB-Lite [24].

### 4.1.1 Global Signals

There are two global signals in SG-Multi, as listed in Table 4.1.

Table 4.1: List and descriptions of global SG-Multi signals

| Signal Name | Description |
|---|---|
| SGCLK | Clock signal used to drive transactions |
| SGRESETn | Active-low reset signal |

All devices in an SG-Multi system share a common clock signal; thus, SG-Multi systems are completely synchronous by design. Reset signalling is active-low and asynchronous, designed to facilitate system-wide reset functionality which would typically happen on initial power-up and in limited circumstances thereafter, perhaps in response to a user request.

### 4.1.2 Master Interface Signals

Communication between a master device and its associated device wrapper is governed by the master interface specification. From a master device's perspective, incoming signals are responses from slaves, whereas outgoing signals form part of commands issued. The master interface signals are listed in Table 4.2.

Because master devices have no knowledge of or concern for the other devices in the system, all transactions must begin with the assertion of SGREQ and cannot proceed until the wrapper asserts SGGRANT in response. This process of arbitration must occur even if the selected slave device is connected exclusively to a particular master device, although in this case SGGRANT would be asserted almost immediately in response to SGREQ.

### 4.1.3 Slave Interface Signals

Table 4.3 lists the SG-Multi signals that exist at the slave interface. Many of the master interface signals are also present at the slave interface, though with opposite directionality.

While most of the master interface signals are also present at the slave interface, there are some important differences. First and most importantly, the slave interface lacks SGREQ and SGGRANT. This is because the slave does perform any arbitration on its own. All of

Table 4.2: List and descriptions of SG-Multi master interface signals

| Signal Name | Direction | Description |
|---|---|---|
| SGADDR | Outgoing | Memory address of interest |
| SGSIZE | Outgoing | Specifies the size of the current transaction |
| SGWnR | Outgoing | Specifies whether the current transaction is for reading (low) or writing (high) |
| SGRDATA | Incoming | Data resulting from a read operation |
| SGWDATA | Outgoing | Data to be written during a write operation |
| SGWAIT | Incoming | Indicates that more time is required to complete the current transaction |
| SGERROR | Incoming | Indicates that the slave encountered an error while processing the current transaction |
| SGREQ | Outgoing | Indicates that a master device wishes to start a transaction |
| SGGRANT | Incoming | Indicates that the master device has been granted permission to start a transaction |

Table 4.3: List and descriptions of SG-Multi slave interface signals

| Signal Name | Direction | Description |
|---|---|---|
| SGACTIVATE | Incoming | Activation signal, asserted when a transaction request involves a particular slave |
| SGADDR | Incoming | Memory address of interest |
| SGSIZE | Incoming | Specifies the size of the current transaction |
| SGWnR | Incoming | Specifies whether the current transaction is for reading (low) or writing (high) |
| SGRDATA | Outgoing | Data resulting from a read operation |
| SGWDATA | Incoming | Data to be written during a write operation |
| SGSNOOP | Outgoing | Indicates that the transaction currently underway supports bus snooping |
| SGWAIT | Outgoing | Indicates that more time is required to complete the current transaction |
| SGERROR | Outgoing | Indicates that the slave encountered an error while processing the current transaction |

the arbitration logic is encapsulated within the slave device wrapper, which simplifies the design of a slave device by allowing the hardware to be structured as though the slave will only accept commands from a single master. Second, `SGACTIVATE` and `SGSNOOP` are both present at the slave interface but not at the master interface. The former is produced by the slave device wrapper and the latter is consumed by master device wrappers, so the functionality associated with them is of no concern to individual master devices.

### 4.1.4 Internal Signals

Communication between master and slave device wrappers primarily occurs using the same signals defined at the interfaces, with the device wrappers merely forwarding signals received from the attached device to the other wrapper. However, SG-Multi's performance-enhancing bus snooping feature makes use of two additional signals. Since the device wrappers implement snooping and do not expose the details of this functionality to the devices at either end of a transaction, these signals do not form part of the SG-Multi signalling protocol specification and are considered internal to the interconnection fabric. They are listed in Table 4.4.

Table 4.4: List and descriptions of signals internal to SG-Multi's interconnection fabric

| Signal Name | Description |
|---|---|
| `SGSADDR` | Address of the current transaction available for bus snooping |
| `SGSSIZE` | Size of the current transaction available for bus snooping |

Both of these signals are broadcast by the slave device wrapper to all connected master device wrappers, which use them to determine whether it is appropriate to capture the result of the current transaction.

## 4.2 Signal Size Multiplexing

Through `SGSIZE`, a master specifies the size of a particular SG-Multi transaction. The sizes of the data busses `SGRDATA` and `SGWDATA` are fixed, so SG-Multi defines a particular manner in which bits of transaction data are transmitted across the data busses for transaction sizes less than the full size of the data bus. Transactions are aligned at addresses corresponding

to the size of the transaction; for instance, a 16-bit transaction is aligned on a 2-byte address boundary. The data bus itself is considered to be aligned at an address corresponding to its own size irrespective of the size of any given transaction. As an example, a 4-byte-aligned 16-bit transfer on a 32-bit (4-byte) data bus would use the lower 16 bit positions on the bus, and the same transfer aligned at a 2-byte (but not a 4-byte) boundary would use the upper 16 bit positions. The bit positions used by a transfer are considered the *active* bit positions. Figure 4.1 provides a visual illustration of signal size multiplexing.


(a) 8-bit transfer of `0x11` to a memory address ending in `0x1`

(b) 16-bit transfer of `0x3322` to a memory address ending in `0x2`

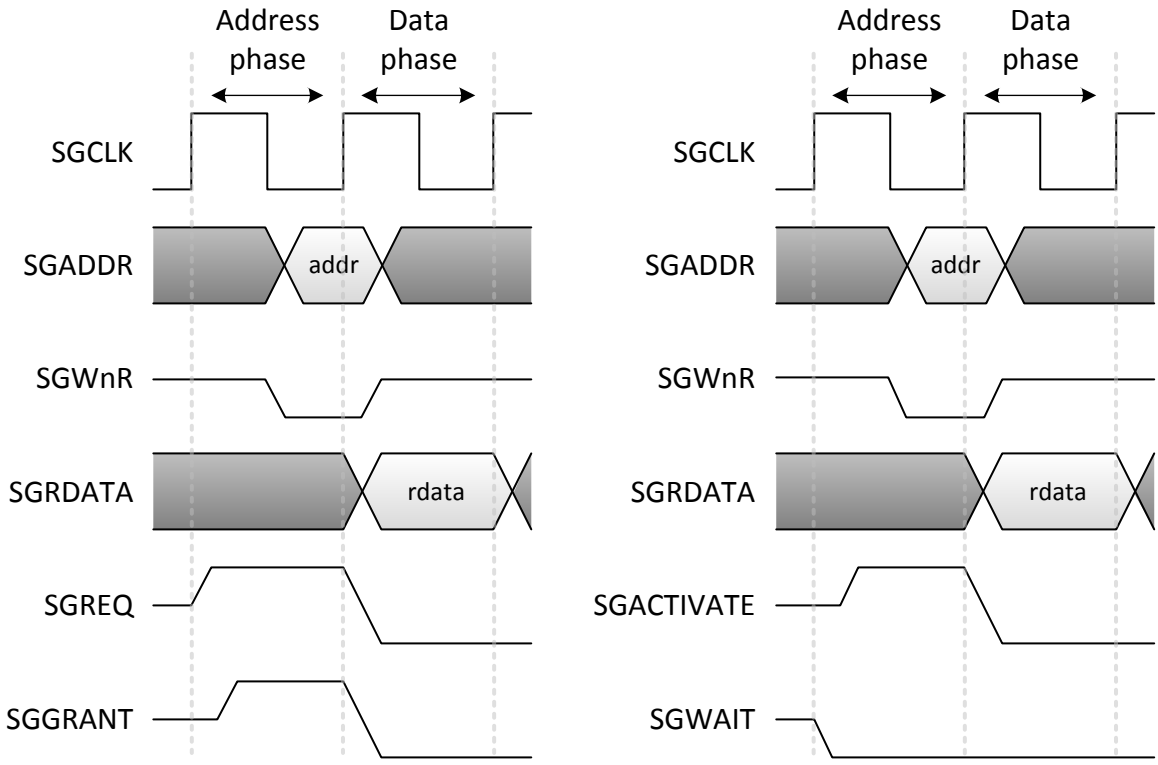
Figure 4.1: Signal size multiplexing illustration on a 32-bit data bus

Signal size multiplexing exists in AHB-Lite [24]. It is particularly beneficial in SG-Multi because it simplifies the logic required to implement the bus snooping feature, which is discussed in more detail in Section 4.4.1.

## 4.3 Transaction Details

All SG-Multi transactions are pipelined in two stages: an *address phase* and a *data phase*. The address phase is characterized by the exchange of control information, such as the memory address of interest and transaction parameters. Arbitration also occurs at the beginning of the address phase; there is no separate stage reserved for arbitration. During the data phase, the slave processes the transaction and produces a response if it is required

to do so. Figure 4.2 illustrates a simple read transaction, both at the master interface and at the slave interface. Some of the less-important signals have been omitted for simplicity.




(a) Master interface

(b) Slave interface

Figure 4.2: Simple SG-Multi read transaction at master and slave interfaces

Slaves signal their response to a transaction request using `SGWAIT` and `SGERROR`, the former for extending the length of a transaction's data phase and the latter for causing the transaction to be aborted due to an error of some kind. The SG-Multi signalling protocol requires all slaves to signal their response one cycle in advance, such that the value seen by masters at the rising edge is an indicator of what is to happen in the upcoming cycle as opposed to being status information resulting from the most recent one. A slave that asserts `SGWAIT` in time for a rising edge is indicating that the rising edge is the start of a wait cycle, and similarly if `SGWAIT` is deasserted at a rising edge then that rising edge is

the beginning of the final cycle required for a particular transaction. A new transaction may begin its arbitration in any cycle that begins with `SGWAIT` deasserted. This includes cases in which the slave uses `SGERROR` to signal an error, as the protocol requires `SGWAIT` to be asserted when `SGERROR` is asserted.

Other architectures, such as AHB-Lite [24], have different semantic behaviour for the signals equivalent to `SGWAIT` and `SGERROR`. In these architectures, slaves provide status information at the end of a clock cycle, such that a slave causes a cycle to begin with the `SGWAIT`-equivalent signal deasserted to signal the end of a transaction and the slave's immediate ability to begin a new data phase. For architectures that require no arbitration, such as those designed under the assumption that only one master will exist in the system, this is not a problem because any master is unconditionally able to begin a transaction with any slave at any time. In arbitration-based architectures, however, these alternative semantics dictate that a master device cannot start arbitration until after a transaction is fully completed. Systems such as AHB [26], the superset of AHB-Lite, address this problem by adding a separate cycle for arbitration before the address phase. SG-Multi avoids this performance loss by requiring slaves to signal their status in advance. This difference in behaviour, applied to `SGWAIT`, is illustrated visually in Figure 4.3. Each of the two signalling semantics are colour-coded: the green version for SG-Multi's signalling semantics and the red version for the alternate semantics. The circled points indicate the time at which a master device becomes aware of the second cycle being a wait cycle.

Earlier slave response signalling also contributes to the achievement of SG-Multi's design goal of universal adaptability. While of no consequence to master devices natively designed to communicate using the SG-Multi signalling protocol, bus adapters may benefit greatly from slaves providing information as soon as possible. It is not possible to consider every existing interconnection architecture individually when designing SG-Multi, and each such architecture will prescribe different signal timing behaviour. Signalling early allows a bus adapter to delay the transmission of information to the attached master device, but signalling late requires a bus adapter to provide information not in its possession.

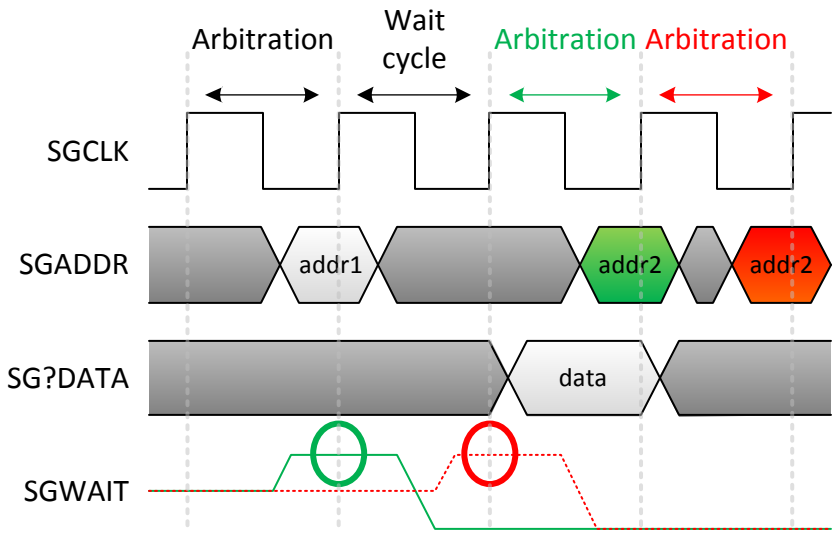



Figure 4.3: Illustration of different slave response signalling semantics

## 4.4 Device Wrapper Details

As discussed in Chapter 3, SG-Multi systems make use of device wrappers to encapsulate all of the functionality required to accommodate multiple master and slave devices in the same system. The subsections that follow describe the design of each of the device wrappers in detail.

### 4.4.1 Master Wrappers

A master device wrapper primarily provides two functions: signal routing and bus snooping. Signal routing determines, based on the input supplied by the attached master device, which slave to involve in a particular transaction. Due to SG-Multi's pipelined nature, a master device wrapper must properly time all signals routed to slaves to support simultaneous address and data phases with different slave devices. Bus snooping, a performance-enhancing feature of SG-Multi, is mostly implemented in the master wrapper component.

Signals are routed based on the memory address supplied by the bus master device. SG-Multi master wrappers use the upper four bits of `SGADDR` to determine the correct

slave. Each master wrapper has 16 slave "slots" and can connect to a total of 16 slaves; the use of the upper bits of the memory address effectively partitions the address space into 16 equal-sized blocks, one for each slave. The signal router is implemented as a decoder for master-to-slave signals and as a multiplexor for slave-to-master signals, as shown in Figure 4.4.

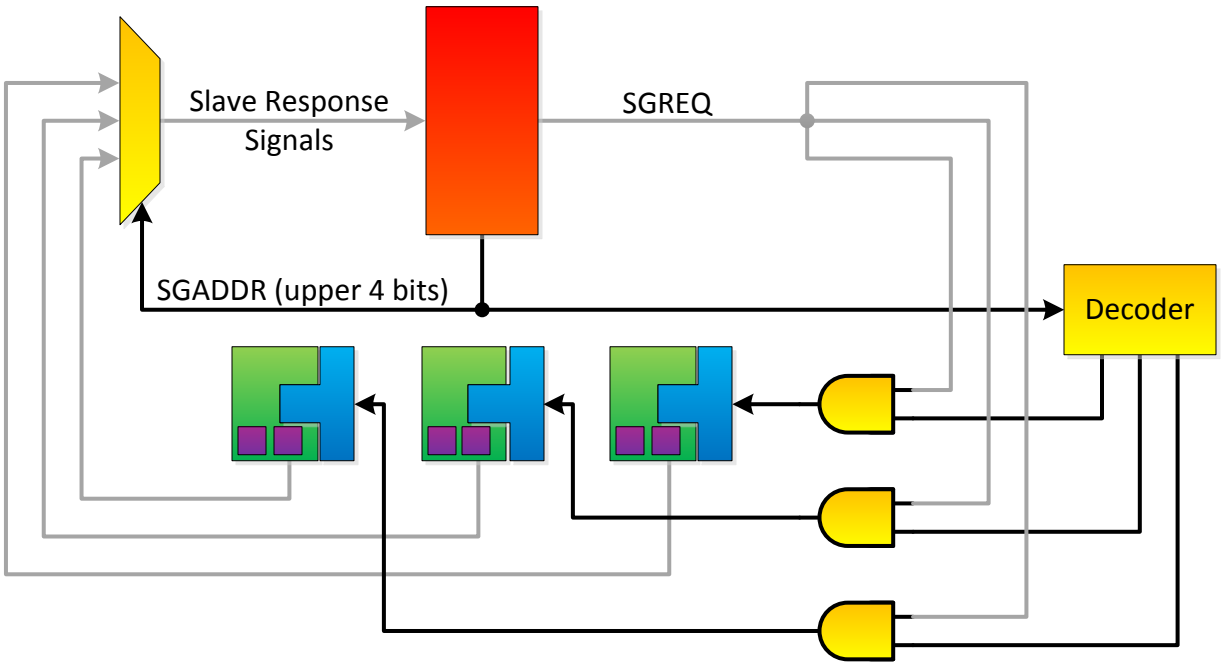



Figure 4.4: Master device wrapper signal routing implementation

Because slave device wrappers multiplex inputs from all master devices and only accept input from the master device that wins at arbitration, it is not necessary to route all master-to-slave SG-Multi signals. In fact, most signals such as `SGADDR` and `SGWDATA` can be broadcast to all slaves as a way of simplifying the logic of the master device wrapper implementation. `SGREQ` is the only signal that indicates a master's interest in a slave, so it is the only signal whose propagation to slaves must be controlled and is therefore the only signal that needs to be routed, as shown in Figure 4.4; all other signals are broadcasted.

The second major function that a master device wrapper provides is bus transaction

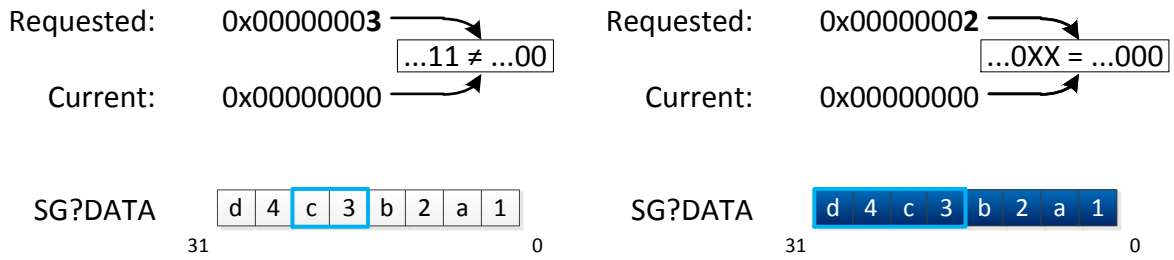
snooping. This performance-enhancing feature is activated whenever a master device attempting to perform a read transaction loses a round of arbitration, the slave asserts `SGSNOOP`, and the slave asserts `SGWAIT` for at least one cycle. During that first wait cycle, the master device wrapper compares `SGADDR` with `SGSADDR` and `SGSIZE` with `SGSSIZE` to determine if the requested transaction is sufficiently similar to the current transaction to allow snooping to be successful. This additional cycle is necessary due to the extra propagation delays and the use of comparators that, when instead added to the end of the address phase cycle, introduces a substantial delay into a cycle that already contains arbitration, slave selection, and slave response signalling. Informal experiments conducted at design time resulted in a maximum clock frequency reduction of approximately 20% when performing the transaction comparisons during the address phase cycle. Furthermore, snooping is more beneficial for longer transactions than for shorter ones, since a master device's average wait time is directly proportional to the time a slave takes to complete a single transaction.

Transaction comparison is the process of a master device wrapper verifying that the current transaction matches the transaction requested by the attached master device to a close enough extent. For snooping to provide the correct data, the master wrapper must be able to guarantee that the data provided by the slave in response to the current transaction will be sufficient to complete the requested transaction. The master device wrapper verifies that both the addresses and the transaction sizes are a match before proceeding to forward the results of the current transaction to the attached master device.

Address comparison ignores the upper four bits of the memory address because these are reserved for slave selection; all remaining bits may be used in the comparison. A naive address size comparison is simply to check for equality of `SGADDR` and `SGSADDR`. While this guarantees correctness, it is over-restrictive because it fails to take advantage of memory address alignment for larger transactions. If the current transaction is larger than the requested transaction and the memory addresses of interest in the request fall within the range of addresses encompassed by the current transaction, snooping remains a possibility, but the address equality check may fail. In the event of a size difference between the current and requested transactions, it is appropriate to drop the lower bits from the address comparisons. Where $n$ is the size of the current transaction in bytes, the number of bits to drop from the address comparison is given by $\log_2 n$. It follows that

the size comparison also need not be one of equality; rather, it is sufficient for the master wrapper to ensure that the requested transaction is smaller than or the same size as the current transaction; in other words, the master wrapper checks that SGSIZE $\leq$ SGSSIZE.

Address and size comparisons are shown in Figure 4.5, where the data bus and current transaction sizes are both 32 bits in size; outlined squares indicate the presence of the requested data, and dark shaded squares indicate what the master wrapper can make available to the master device. As demonstrated in Figure 4.5(b), if the master device wrapper snoops a transaction successfully, it forwards the entire result to the attached master, not just the portion of the result that the master actually requested. The signal size multiplexing described in Section 4.2 ensures that the master device will be able to extract the requested data, eliminating the need to include this logic in the master device wrapper.




(a) 8-bit request with no address bits dropped

(b) 16-bit request with lowest two address bits dropped

Figure 4.5: Address and size matching performed for bus snooping

The portion of bus snooping functionality implemented in the master device wrapper takes the form of a state machine. It contains four states, each of which is described in Table 4.5. If at any time the slave device signals SGERROR, the snooping operation is abandoned and the state machine returns to its initial state; this error-handling mechanism is present but has been omitted from descriptions presented here for the sake of simplicity.

The transitions between states are shown in Figure 4.6. References to preconditions or to transaction matching refer to the snooping preconditions and transaction matching criteria described previously in Table 4.5. A direct transition exists from $s_3$ to $s_1$ to accommodate cases where the last cycle of a snooped transaction's data phase coincides with the first cycle of another transaction's address phase and the latter transaction meets

Table 4.5: State description for master device wrapper bus snooping state machine

| State | Description |
|---|---|
| $s_0$ | Initial state. Waits for `SGWAIT`, `SGSNOOP`, and a read transaction request for which the attached master device loses arbitration. |
| $s_1$ | Analysis state. Compares the requested transaction to that described by `SGSADDR` and `SGSSIZE`. |
| $s_2$ | Waiting state. Snooping can proceed, but the slave is still asserting `SGWAIT`. |
| $s_3$ | Snooping state. This is the last cycle of the transaction, so the results are captured and forwarded to the master device. |

the preconditions for snooping. An unconditional transition from $s_3$ to $s_0$, the alternative, would miss these back-to-back snooping opportunities.

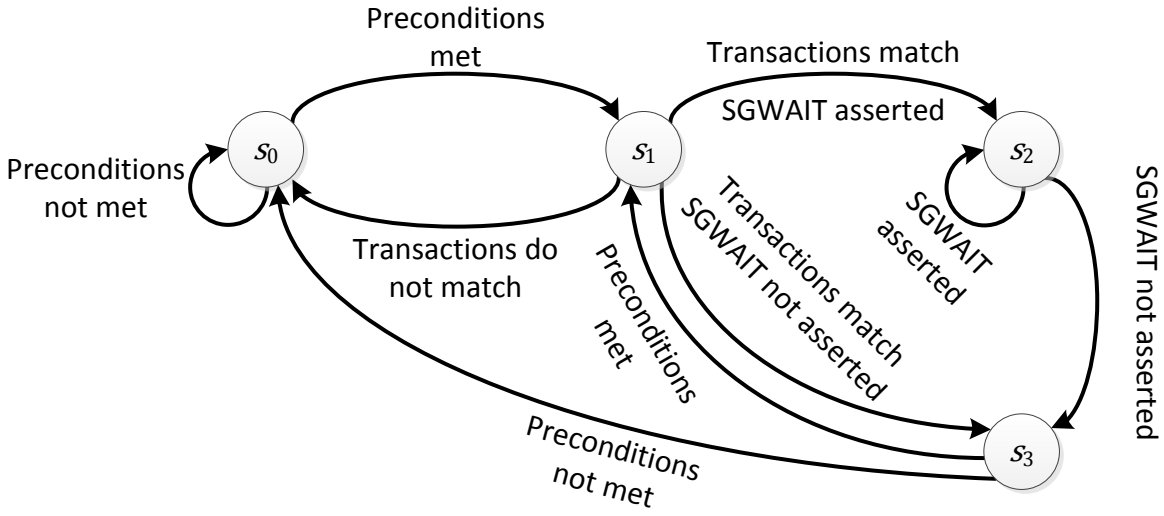



Figure 4.6: Bus snooping state machine in the master device wrapper

A master device is not able to distinguish transaction completion due to winning arbitration from transaction completion due to snooping. The master device continues to assert `SGREQ` until it receives `SGGRANT` from the master device wrapper. While typically a master device wrapper signals `SGGRANT` when it wins a round of arbitration, it will also generate `SGGRANT` during a transition to $s_3$ in the bus snooping state machine, marking the conclusion of a successful snoop operation.

### 4.4.2 Slave Wrappers and Arbiters

Some of the core functionality of a slave device wrapper is similar to that of a master device wrapper in that a slave device wrapper must similarly route certain outgoing signals and multiplex incoming signals. A slave device wrapper also contains arbiters and provides the supporting framework for them. Accordingly, much of the routing functionality of a slave device wrapper is controlled by the results of a round of arbitration. In a similar style to Figure 4.4, Figure 4.7 illustrates the general connections of components within a slave; for the sake of simplicity the connections of `SGGRANT` to each master have been omitted.

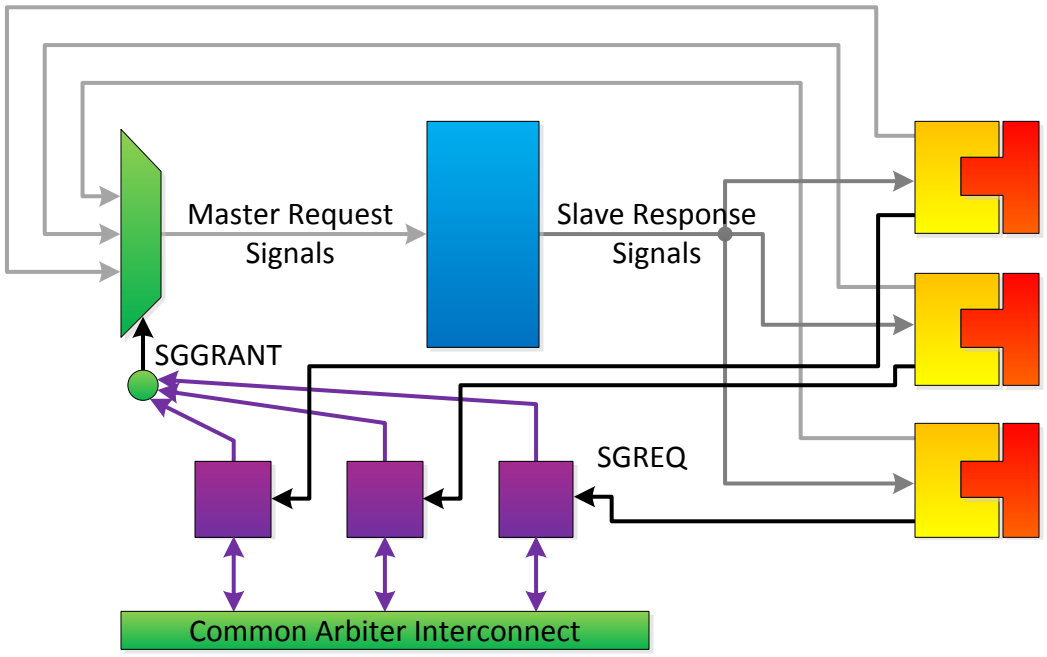



Figure 4.7: Slave device wrapper signal routing implementation

Like master device wrappers, slave device wrappers use the notion of a "slot" to describe a connection between it and another device in the system. However, slave device wrappers do not route signals based on memory addresses and thus can theoretically support an unlimited number of connections to other devices (although the slave device wrapper ref-

erence implementation limits this number to 16 to avoid over-complicating the hardware and to be consistent with the master device wrappers). The slot to which a particular master device is connected is important for arbitration, but beyond that there is no semantic significance to the order of the slots.

As introduced in Chapter 3, arbitration in SG-Multi is on the basis of a combination of static and dynamic priority levels, both of which are represented in one-hot form. Each slot in a slave device wrapper is assigned a static priority level that is unique within the scope of that slave device wrapper; as a result, the number of bits in a static priority level is equal to the total number of devices connected to the slave device in question. A master device's dynamic priority level begins at 0, increases each time it loses arbitration, and is reset to 0 when it finally wins arbitration. The number of dynamic priority levels, a design-time configurable parameter, determines the bit width of a slave device wrapper's dynamic priority level. Dynamic priority takes precedence when performing a round of arbitration; the static priority levels are used to resolve situations in which multiple master devices have the same level of dynamic priority.

Arbiters are purely combinational circuits. They are effectively divided into two stages based on priority level types being compared: dynamic arbitration and static arbitration. In each case, the first step is to generate a mask using the one-hot priority level as input. The desired output of this transformation is one in which the bits of greater significance than the position of the '1' in the priority level are '1' and the rest are '0'; for example, an input priority level of $(00100000)_2$ should produce a mask of $(11000000)_2$. This is achieved by:

1. Shifting the input bit pattern left by one position.
2. Adding the result to a string of '1' bits and discarding the output carry (for example, in the case of an 8-bit priority level, the addition is to the value $(11111111)_2$).
3. Inverting all the bits in the result of the addition.

An example of this mask transformation, using the sample input of $(00100000)_2$, is shown in Table 4.6.

Arbitration functionality is split between the slave device wrapper and the arbiter units themselves. Individual signal comparisons and `SGGRANT` generation happens within

Table 4.6: Mask transformation example for arbitration

| Step | Result |
|---|---|
| 0 | $(00100000)_2$ |
| 1 | $(01000000)_2$ |
| 2 | $(00111111)_2$ |
| 3 | $(11000000)_2$ |

the arbiter, but the slave device wrapper performs several important functions as part of the arbitration process:

- Combining the priority levels supplied by each individual arbiter and feeding the results back into the arbiters.
- Providing each arbiter with its static priority level as an input.
- Filtering bus requests and only allowing `SGREQ` at an arbiter's input to be asserted in clock cycles beginning with `SGWAIT` deasserted. This prevents bus arbitration from occurring in the middle of an ongoing transaction.

The complete logical circuit showing how an arbiter generates `SGGRANT` shown in Figure 4.8; thicker lines represent multi-bit signals and thinner ones represent single bits. Both the functionality of updating the dynamic priority level and the filtering of `SGREQ` are omitted for simplicity. Dynamic priority is stored in a register and incremented through a left shift operation each rising edge when `SGREQ` is asserted but `SGGRANT` is not. The "Common Arbiter Interconnect" referenced in Figure 4.7 consists of the green-shaded logic gates and their outputs.

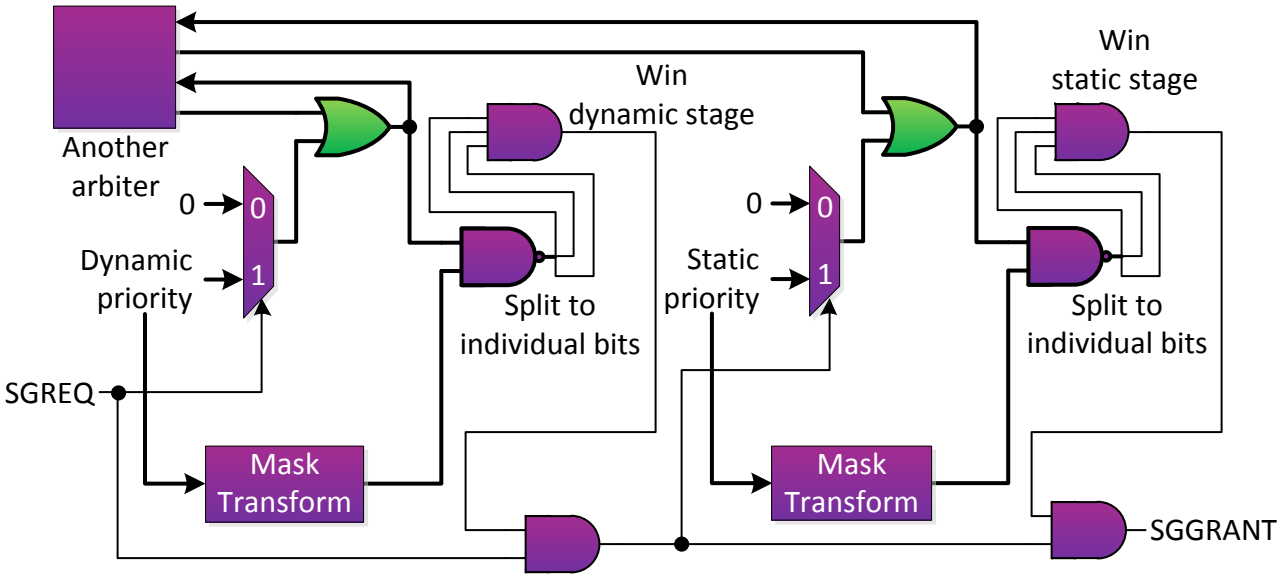



Figure 4.8: Combinational logic circuit implementation of an SG-Multi arbiter

# Chapter 5

# Tools Design

The SG-Multi Designer framework is composed of a unifying application—SG Multi Designer—that presents users with a graphical interface, a set of core tools for constructing the fundamental SG-Multi hardware components, and a set of modules, potentially provided by third-parties. Users provide the framework with input in the form of a design modelled using SG-Multi Designer's underlying abstraction model, which emphasizes simplicity while avoiding the problem of over-simplification.

This chapter begins by explaining the SG-Multi Designer abstraction model. It subsequently provides an overview of the composition of the framework as a whole and finishes with a brief presentation of the graphical interface by which the unifying application presents the abstraction model to the user.

## 5.1 Abstraction Model

SG-Multi Designer's underlying abstraction model emphasizes simplicity, the goal being to minimize the burden placed on the hardware designer. Many of the low-level details are abstracted to the point that a user simply adds devices to the system and specifies the set of interconnections between devices, without worrying about issues such as interface compatibility or individual bus signal connections. This is the essence of the simplicity of this model; instead of manipulating low-level signals, interface bridges, and other such components, SG-Multi Designer presents the hardware designer with a total of only six components with which to interact, as listed in Table 5.1.

Table 5.1: List and descriptions of components in the SG-Multi Designer abstraction model

| **Component name** | **Description** |
|---|---|
| Master | A device that issues commands and initiates inter-device transfers. |
| Slave | A device that responds to commands and services inter-device transfers. |
| Property | A design-time configurable parameter that governs the functionality of a particular instance of a device. The list of available properties varies by device type, and each property value is user-specified. |
| Device Connection | A connection between two devices, typically between a master and a slave. One of SG-Multi Designer's key distinguishing factors compared to competing solutions is that device connections have no configurable parameters associated with them; either they exist or they do not. |
| Connection Point | A named node that is accessible throughout the design. Connection points can represent external pins—inputs, outputs, or bidirectional signals—or internal wires. Connection points are user-defined, and SG-Multi Designer does not limit the number of connection points that can exist in any given design. |
| Extra Signal | A signal, outside of the standard SG-Multi interface signals, attached to a device, providing additional input or output. The list of available extra signals varies by device type. By default, extra signals are disconnected; explicit user action is required to connect them to connection points. Some extra signals provide crucial functionality and therefore must be connected, but typically their connection is optional. |

As an example, consider a small SG-Multi system consisting of just a single processor as the master, a read-only memory (ROM) unit as one slave whose purpose is to supply executable code, and a serial port controller as a second slave whose purpose is to interact with a user through a terminal application. The processor supports interrupts, which is modelled by defining the "interrupt request" (IRQ) inputs as extra signals. The ROM has no extra signals, but the serial port controller has three: a serial "transmit" line as output, a serial "receive" line as input, and an IRQ line as output. This example system, modelled using the abstraction model, is depicted in Figure 5.1. Large light rectangles are devices, smaller shaded rectangles are connection points, solid arrows represent device connections, and dashed arrows represent the connections of extra signals.

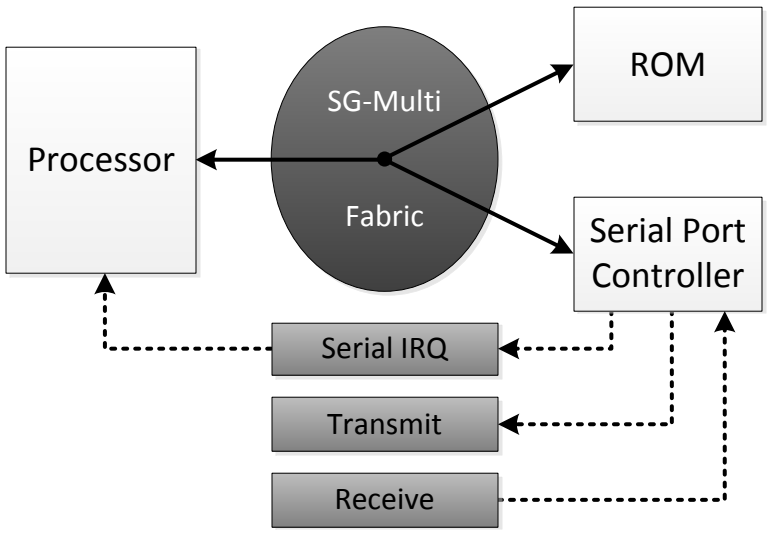



Figure 5.1: Example of a system design modelled using SG-Multi Designer's abstraction model

The degree of the abstraction model's simplicity becomes readily apparent when considering the precise list of steps a hardware designer would need to perform to create the system shown in Figure 5.1. The hardware designer must:

1. Add an instance of the processor core to the design.

2. Add an instance of the ROM unit to the design.

3. Add an instance of the serial port controller to the design.

4. Create two connection points representing external pins, one each for the serial port's "transmit" (output) and "receive" (input) lines.

5. Optionally create a connection point for the serial port IRQ line.

6. Specify the existence of a device connection between the processor and the ROM unit.

7. Specify the existence of a device connection between the processor and the serial port controller.

8. Connect the serial port controller's extra signals to the "transmit" and "receive" line connection points.

9. Optionally connect both the serial port's and the processor's extra signals to the serial port IRQ line.

The order of performance of these steps need not be exactly as shown; a hardware designer may complete them in any order so long as the devices and connection points exist before connections between them are inserted.

It is particularly noteworthy that the interface with which the processor was designed to be compatible is not specified. Because SG-Multi and, as a result, SG-Multi Designer support bus adapter devices that allow devices designed for different architectures to function as part of an SG-Multi system, the abstraction model encapsulates all the functionality required to connect these types of devices. For example, if a designer wishes to add a processor for which an appropriate bus adapter is available, SG-Multi Designer automatically generates the required components to allow the addition without placing that burden on the hardware designer. To the user, the processor core appears just as any other master device, complete with properties, extra signals, and device connections.

Device properties and extra signals help avoid the problem of over-simplification. A device that supports configurable properties offers the user the opportunity to fine-tune a device's functionality on a per-instance basis. Extra signals allow a device to be connected

to custom parts of the system not captured by the underlying interconnection architecture. Both properties and extra signals are pre-defined by the device being instantiated. SG-Multi Designer imposes no restrictions on the number of properties and extra signals a device can offer.

## 5.2 Framework Overview

The SG-Multi Designer framework consists of a unifying graphical application, a set of core tools for generating the fundamental SG-Multi components, and modules for generating any other devices that may be included in a system design. The framework's heart lies in the graphical application, itself called SG-Multi Designer; in addition interacting with users for the purpose of facilitating the construction of an SG-Multi system, it communicates with the other framework elements as required to implement the input design and generates the logic that instantiates and integrates the individual devices. SG-Multi Designer controls the flow of the framework's design process; its interface is guided entirely by the underlying abstraction model.

Each module provides the functionality needed to describe and generate a particular device that is available for users to add to system designs. For instance, one module might provide a particular type of processor core, and another might provide a memory controller. Modules act as "plug-ins" to SG-Multi Designer, each one extending the library of available devices that may be added to the system design. In this way, modules encapsulate intellectual property units and thus will often be supplied by a third-party. There are a total of four different types of modules the framework supports, as listed in Table 5.2.

Table 5.2: List and descriptions of the types of SG-Multi Designer modules

| **Module type** | **Description** |
|---|---|
| Master | Modules that generate SG-Multi master devices. |
| Slave | Modules that generate SG-Multi slave devices. |
| Bus adapter | Modules that generate bus adapter components required to attach non-SG-Multi master devices to the system. |
| Adapted master | Modules that generate non-SG-Multi master devices, which depend on the existence of a suitable bus adapter. |

A module consists of two parts:

- A **module description**, which is simply a file in XML format containing the information SG-Multi Designer needs to determine how to interact with the module and what options to present to the user.
- An **executable program**, which accepts parameters in the form of command-line arguments and, when executed, generates the Verilog code that implements device logic custom-tailored according to the parameters specified.

Based on the information each module supplies in its XML description and the design specified by the user, SG-Multi Designer determines the appropriate set of command-line arguments with which to invoke each module's executable program.

The abstraction model removes much of the low-level burden from the hardware designer using the framework, shifting it onto the module provider. A module is required to supply sufficient information in the XML file to uphold the abstraction model; Appendix B provides a complete set of XML file specifications. At a high level, the XML file for a master or slave device module must supply:

- A **name** for the module.
- A list of **properties**, their names, descriptions, data types, and optionally minimum and maximum values.
- A list of **extra signals** supported by the device.
- The name of the **executable file** that must be invoked to create a new variant of the device.
- An ordered list of **command-line arguments** for the executable file.

Core tools, which generate common SG-Multi logic, are similar to modules in that they are executable programs. However, unlike a standard module, the information needed to communicate with them is integrated into SG-Multi Designer due to their more fundamental role in the construction of an SG-Multi system. One core tool exists for each of the device types listed in Table 1.1, with the exception of bus adapters as these are treated as modules. Figure 5.2 shows a visual representation of the entire framework, including SG-Multi Designer, the core tools, and the device modules.

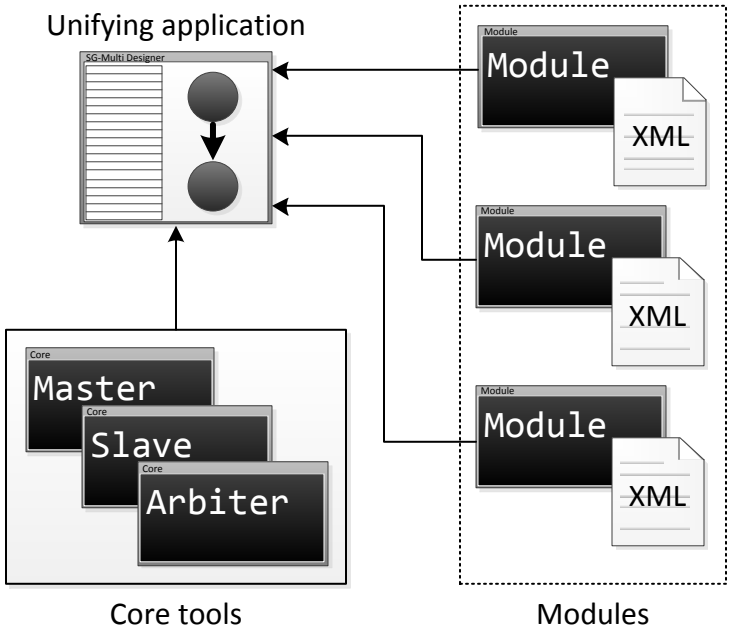



Figure 5.2: Overview of the SG-Multi Designer framework

## 5.3 User Interaction

Figure 5.3 shows a representation of the main window of SG-Multi Designer. This window on its own implements the majority of the abstraction model. Devices and connection points are listed in a tree view on the left pane, and when a device is selected its connections are shown on the right pane. Connecting and disconnecting a device is as simple as selecting it from a drop-down menu, no other action required. Extra signals and device properties may be modified on a separate window by selecting a device and clicking the "edit" button (represented as a gear); as with device connections, extra signals are connected and disconnected from connection points using a drop-down list.

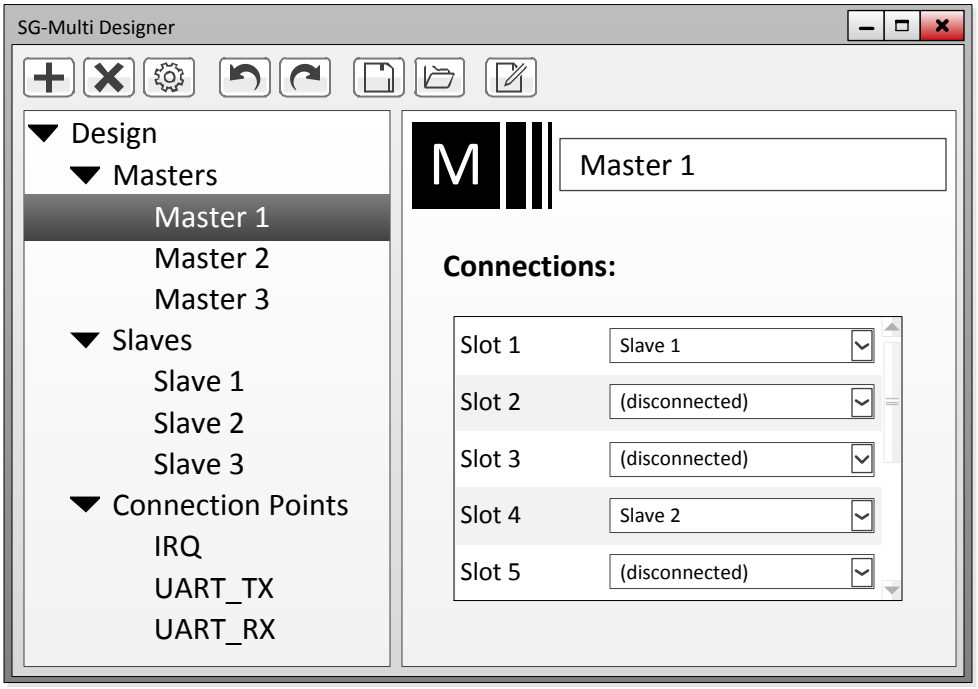



Figure 5.3: Representation of SG-Multi Designer's main window

# Chapter 6

# Experimental Results

Experiments used to evaluate SG-Multi and SG-Multi focus on many different characteristics of the systems under test. Since FPGAs are supported as prototyping tools, some of the initial experiments capture trends with respect to the FPGA resource demands of these systems, metrics which concern scalability of SG-Multi, particularly as the number of processor cores in the system increases. These scalability experiments also target potential ASIC fabrication, emphasizing trends in the area required to implement them with respect to the number of processor cores. The reference systems tested all require bus adapters, so other tests evaluate the ability of SG-Multi to achieve its goal of latency-free adaptability by comparing the number of clock cycles required to complete a benchmark application in an SG-Multi system versus a processor's native architecture. Finally, correct operation of SG-Multi-specific features such as transaction snooping is verified by comparing the number of cycles required to complete a benchmark with snooping enabled, with it disabled, and with only one processor core present. The subsections that follow provide details on each experiment conducted and present the results.

All experiments are based on a set of reference SG-Multi systems implemented in Verilog. FPGA-based tests involve prototyping these designs on an Altera Cyclone II DE2 board, developed by Terasic [39], after having been compiled using version 11.1 of Altera Quartus II software with the fitter's router timing optimization level set to "maximum" but all other settings left at their defaults. Since SG-Multi ultimately targets ASIC implementation, experiments also involve compilation with version E-2010.12-SP2 of Synopsys Design Compiler, using a 0.18 µm technology library supplied by Taiwan Semiconductor

Manufacturing Company (TSMC), to facilitate the extraction of performance and area measurements. Although fairly old, this library was readily available at the time of testing and, since only relative measurements are of interest, the choice of technology library is largely immaterial. The only constraint specified was a clock rate constraint, and all settings were left at defaults. In each system, the slave devices include an LED controller, a serial port controller, an SRAM controller, and one or more ROM devices containing executable code. The master devices are one or more ARM Cortex-M0 processors [25]. Examples of typical single-core and dual-core reference systems are shown in Figure 6.1; the gray lines and gray-bordered components are only present in a dual-core configuration of this system, and variations containing higher numbers of cores simply add additional Cortex-M0 instances, ROM device instances, and a subset of the other device connections.

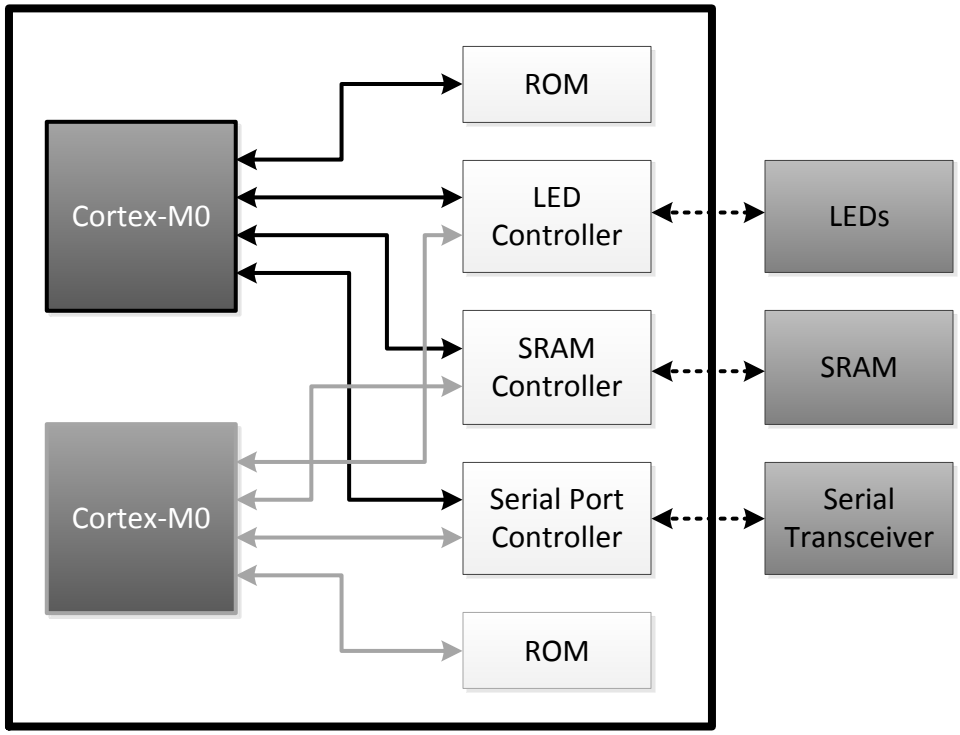



Figure 6.1: Single-core and dual-core reference system configurations for experiments

## 6.1 AHB-Lite Comparison

The Cortex-M0 processor communicates natively according to the AHB-Lite procotol [25], so an important test of basic correctness is ensuring that the combination of an AHB-Lite bus adapter and the SG-Multi interconnection fabric compares favourably in terms of performance and area to an identically-configured system based on AHB-Lite's interconnection architecture. While AHB-Lite systems can be constructed with multiple processors, AHB-Lite is designed primarily for single-master systems [24]; as a result, it is compared with the single-core variant of the SG-Multi reference system. The experiments shown in this section demonstrate how SG-Multi compares to AHB-Lite in terms of the number of clock cycles required to execute a benchmark application and the resources required to implement each system.

### 6.1.1 Benchmark Performance Comparison

A successful performance test would indicate no difference in the number of clock cycles required by each of the two systems to complete execution of a sequence of instructions. Accordingly, the performance test utilizes a synthetic benchmark application constructed specifically for this purpose. The application simply executes a busy-wait loop, checking and updating the value of a counter variable stored in SRAM each iteration. The single-core SG-Multi and AHB-Lite systems are binary-compatible; the same compilation tools were used for both, and the binaries supplied to each processor were identical. Both systems were augmented with cycle-counting circuitry that begins running on system power-up and stops counting once the processor indicates it has completed its execution of the benchmark application. The test was executed a total of 5 times for each system, with each test ranging from 1,000,000 to 15,000,000 iterations of the loop. The results are shown in Table 6.1.

The benchmark application has demonstrated that it is possible to achieve zero added latency in an SG-Multi system compared to using the individual processors' native architecture. The core interconnection components, combined with the AHB-Lite bus adapter used in these tests, do not introduce latency; the AHB-Lite bus adapter used in these tests has been designed to avoid the introduction of latency, though different bus adapters might not produce similar results. Ultimately the quality of the results and the elimination of additional latency depend upon the quality of the bus adapter.

Table 6.1: Latency comparison of AHB-Lite and SG-Multi systems

| Iterations | AHB-Lite Cycles | SG-Multi Cycles |
|---|---|---|
| 1,000,000 | 22,000,055 | 22,000,055 |
| 1,666,666 | 36,666,707 | 36,666,707 |
| 2,333,333 | 51,333,381 | 51,333,381 |
| 3,000,000 | 66,000,055 | 66,000,055 |
| 15,000,000 | 330,000,055 | 330,000,055 |

### 6.1.2 FPGA Hardware Comparison

FPGA resource requirements and maximum frequency values provide insight into the general added cost of using SG-Multi versus using a processor's native architecture. SG-Multi is more complicated than AHB-Lite and, as a result, is expected to have a lower maximum frequency and consume a larger amount of device resources; however, in order to be useful as an architecture, these added costs must reasonable. For example, AHB-Lite features no arbitration logic and no bus adaptation logic, most of which is combinational circuitry in SG-Multi, thus creating the expectation of higher usage of combinational functions. The values reported by Quartus II are shown in Table 6.2.

Table 6.2: FPGA-based comparison of AHB-Lite and SG-Multi systems

| Metric | AHB-Lite | SG-Multi | Difference |
|---|---|---|---|
| Logic Elements | 4,473 | 4,916 | +9.9% |
| Combinational Functions | 4,254 | 4,689 | +10.2% |
| Registers | 1,028 | 1,208 | +17.5% |
| Maximum Frequency | 58.29 MHz | 56.63 MHz | –2.8% |

Clock frequency is reduced by less than 3%, which may well be considered a negligible performance loss, depending on the specific application. Combinational functions and total logic elements increase in number by approximately 10%, but more significantly is the increase in the number of registers; at first glance, the 18% relative increase appears extremely high. However, the Cortex-M0 is designed for simplicity and small size [25]; naturally, the use of a larger, more powerful core would produce the same absolute change but a lower relative difference. Furthermore, the actual resource differences—443 logic elements, 435 combinational functions, and 180 registers—each account for less than 1.5%

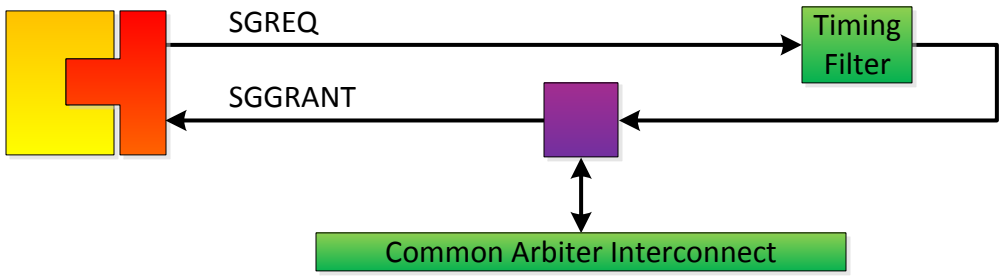
of the FPGA device's available resources, with the 180 extra registers occupying approximately 0.5%. In the case of a single-core system prototyped on an FPGA, it is clear that the extra hardware resource requirements imposed by using SG-Multi are well within reason.

### 6.1.3 ASIC Hardware Comparison

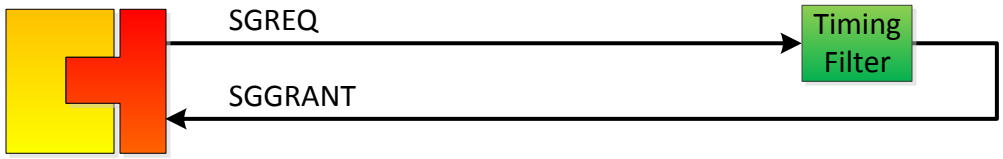
SG-Multi primarily targets ASIC implementation, so arguably more important than FPGA-based results are those obtained for potential ASIC fabrication. In this experiment, each system was compiled twice, once with the clock specified at an achievable 50 MHz and again with the clock specified at an unachievable value of 500 MHz. The maximum clock frequency can be derived from the worst-case negative slack reported in the latter case by simply adding it to the clock period specified in the constraint to obtain the minimum clock period for proper operation. The 50 MHz variation was used to capture area-related results, and the 500 MHz variation was used only for the maximum frequency test. Sensitivity to process variation is particularly important in ASIC design and fabrication, so the experiment was repeated three times, once for each of the technology library's process variation models: best-case, typical, and worst-case.

The addition of arbitration logic is one of the key differences between SG-Multi and AHB-Lite. Since SG-Multi arbitration circuitry consists of combinational logic that must be evaluated during the address phase, the cost of arbitration was also evaluated by performing each SG-Multi test both with and without arbiters. The former case is identical to the configuration of SG-Multi described in Chapters 3 and 4, and the latter case reflects an optimization SG-Multi Designer would perform for slaves only connected to a single master, in which the slave still filters incoming `SGREQ` signals for timing purposes but the actual arbitration circuitry is bypassed and `SGGRANT` is simply connected to the output of the slave device wrapper's usual filtering circuitry. This difference is illustrated in Figure 6.2.

Combinational area test results are shown in Figure 6.3. As expected, SG-Multi configurations require more area than the AHB-Lite system due to the added bus adapter and device wrapper circuitry. When the circuits were optimized for both the best-case and worst-case conditions, SG-Multi with arbiters requires approximately 25% more combinational area than AHB-Lite, and SG-Multi without arbiters requires approximately 20%. It




(a) Standard configuration containing arbitration circuitry




(b) Optimized configuration omitting arbitration circuitry

Figure 6.2: Experiment configuration of slaves, with and without arbiters

can therefore be concluded that the arbiters in SG-Multi, of which there are 16 in total in the single-core reference system, collectively consume approximately 6000 µm$^2$. Interestingly, the area results for SG-Multi configurations are more stable across process variations than those of AHB-Lite.

Noncombinational area test results are shown in Figure 6.4. The results essentially mirror those of the combinational area, with AHB-Lite unsurprisingly requiring less area and the difference being relatively quite similar. The same observation with respect to process variation sensitivity can also be observed here.

Total area test results, which simply combine the figures obtained in the previous two tests, are shown in Figure 6.5. The difference between the area consumed with and without arbiters is approximately 5.5%, which is not particularly significant, leading to a total area of approximately 15000 µm$^2$ consumed by arbiters.

Maximum frequency results are shown in Figure 6.6, with linear trend lines added to better illustrate sensitivity to process variation. It is here that the cost of arbitration is

especially clear; on average, AHB-Lite is able to achieve a clock frequency approximately 40% higher than SG-Multi with arbiters, but only 14% higher on average when arbiters are removed. The differences, however, are the least pronounced in the worst-case process variation condition. Sensitivity to process variation is represented in the slopes of the trend lines, with a shallower slope being more reflective of the desirable property of process variation insensitivity. In all tests, including the maximum frequency test, SG-Multi has been shown to be less sensitive to process variation than AHB-Lite, having a best-case $f_{\max}$ 142 MHz greater than its worst-case $f_{\max}$, compared to AHB-Lite's difference of 188 MHz.

These comparison results as a whole are reflective of the classic trade-off between area, performance, and functionality; it is generally not possible, without changing the underlying technology, to optimize a design for all three of these metrics. SG-Multi adds additional functionality largely in the form of combinational logic and therefore is expected to require a larger area and have a lower maximum clock frequency compared to AHB-Lite. Whereas the added area of SG-Multi is relatively minor—even when arbiters are present—the loss of performance is more pronounced. In the trade-off between area and performance, it is clear that SG-Multi favours smaller area over high clock frequency.

## 6.2 Hardware Scalability

SG-Multi is designed for multi-core processors, so it is imperative that the architecture be scalable in terms of both the hardware resources required and the maximum achievable clock frequency as the number of cores increases. Scalability tests involve compiling six variations of the reference SG-Multi system, ranging from the single-core case to an eight-core case, and recording the hardware resources consumed as well as the maximum clock frequency achievable. These tests evaluate the scalability of SG-Multi both for prototyping it on an FPGA and for fabricating it as an ASIC; the subsections that follow present the results.

In a system containing only processor cores and no interconnection logic, the introduction of additional cores causes a purely linear increase in the hardware resources consumed by the design; if a single core requires $x$ resources, then $n$ cores should require $nx$ resources since hardware is simply being duplicated. Adding slave devices shared among all the processor cores in the system requires a constant amount $s$ of extra resources irrespective of

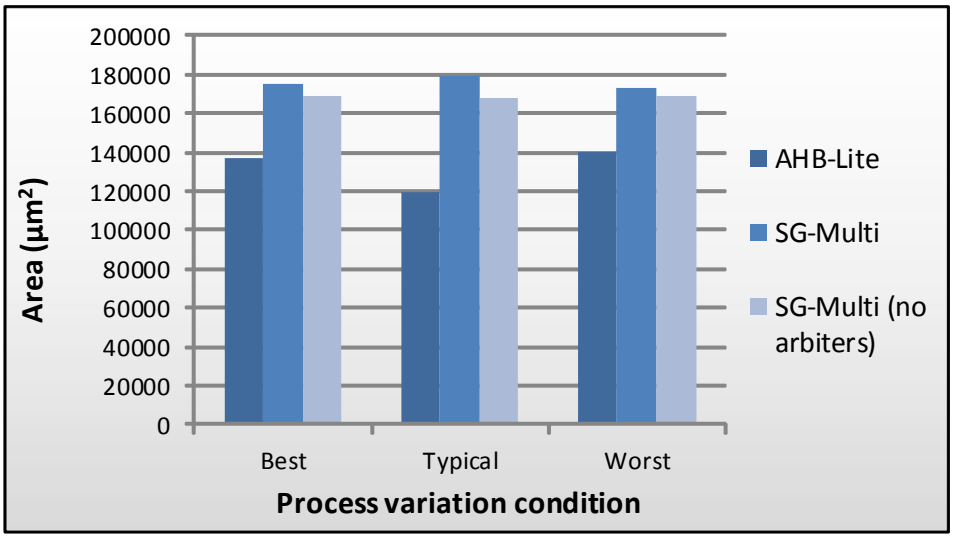



Figure 6.3: Combinational area results for comparing AHB-Lite and SG-Multi

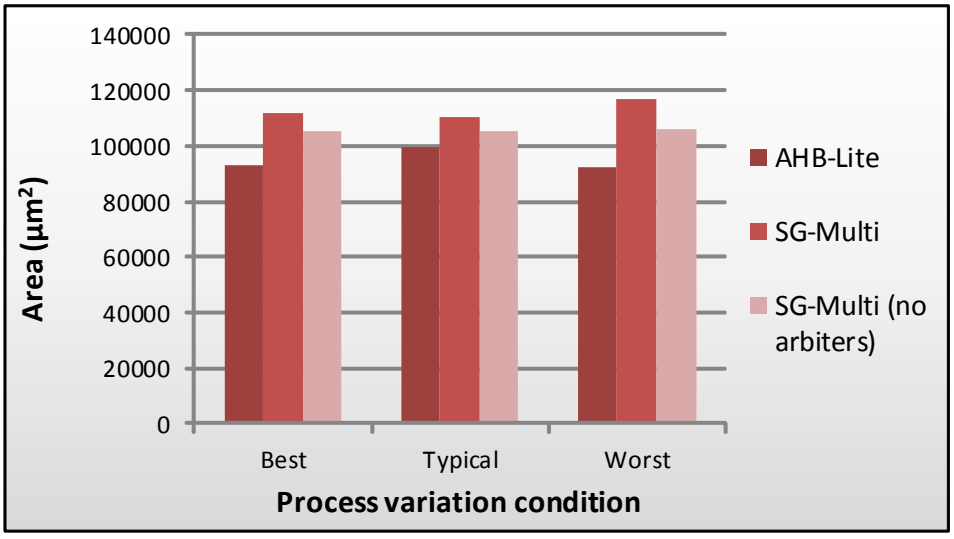



Figure 6.4: Noncombinational area results for comparing AHB-Lite and SG-Multi

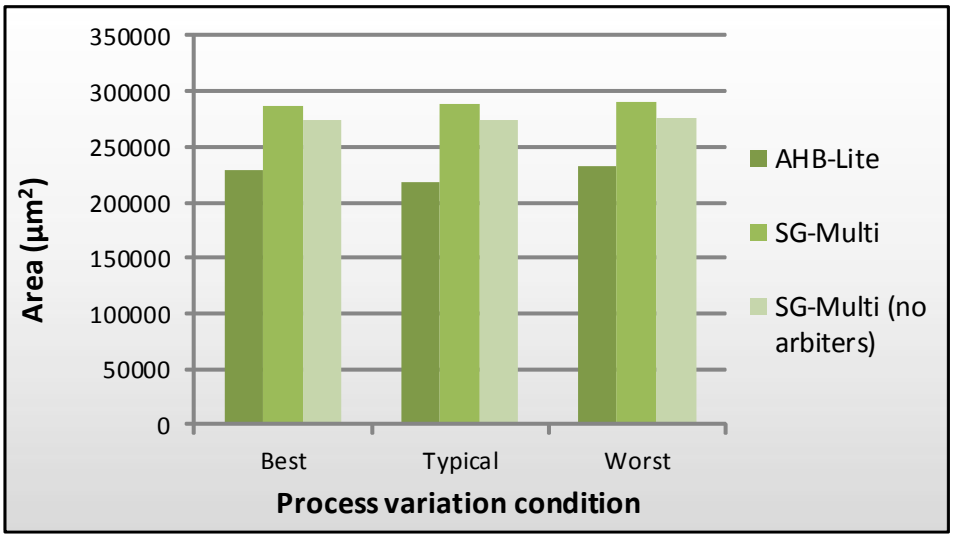



Figure 6.5: Total area results for comparing AHB-Lite and SG-Multi

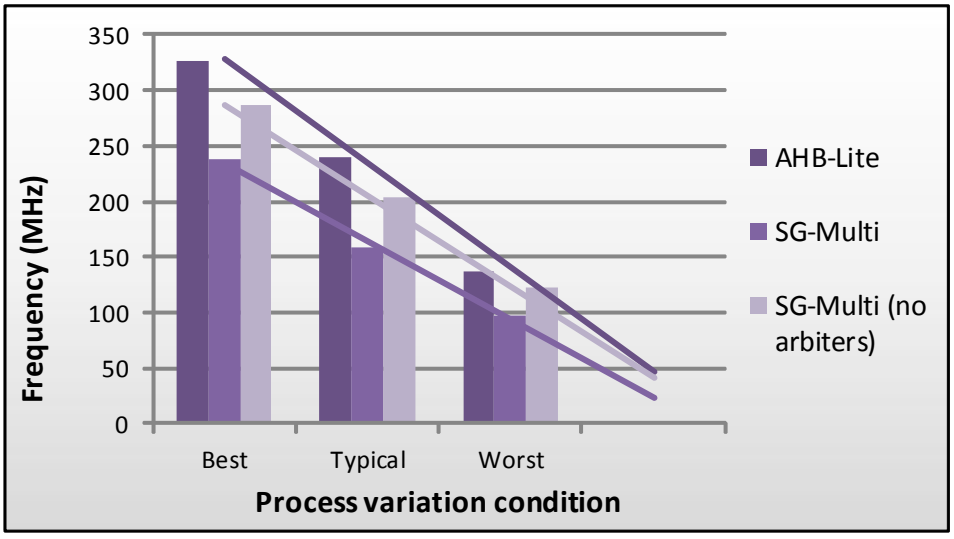



Figure 6.6: Maximum clock frequency results for comparing AHB-Lite and SG-Multi

the number of processor cores actually in the system. The expression then becomes $nx + s$ for the expected hardware resource growth rate in a scalable multi-core system. While this analysis does not consider overheads such as those related to interconnect and routing, it indicates that the scalability of an interconnection architecture can be judged preliminarily based on the system's overall hardware resource consumption growth rate as compared to linearity.

### 6.2.1 FPGA Scalability Results

The FPGA scalability test measures the growth rate of the number of combinational functions, registers, and total logic elements consumed by, as well as the change in the maximum achievable clock frequency in, an SG-Multi system as the number of cores grows. These results are shown in Figures 6.7, 6.8, 6.9, and 6.10, respectively. Each graph has been augmented with a dashed line showing a linear trend based on the first two data points captured.

Combinational functions and total logic elements appear to grow sublinearly as the number of cores increases to 6 and then to 8 ,which is encouraging from a scalability standpoint. Register growth is linear with a slight deviation when the number of cores is 6, likely resulting from an optimization fluke and inconsequential for interpreting the results. Maximum clock frequency results show an initial decline that stabilizes as the number of cores passes 3. From these results, it is concluded that SG-Multi scales well when prototyped on an FPGA.

### 6.2.2 ASIC Scalability Results

The ASIC scalability test measures the reference SG-Multi system's growth rate with respect to the number of cores in terms of its combinational area, noncombinational area, and total area, as well as performance changes measured based on the maximum achievable clock frequency. These results are shown in Figures 6.11, 6.12, 6.13, and 6.14, respectively. As with the FPGA results, each graph has been augmented with a dashed line showing linear growth.

Area-related test results show that all data points captured lie along the linear growth dashed line; SG-Multi scales linearly in terms of the area it consumes. Clock frequency

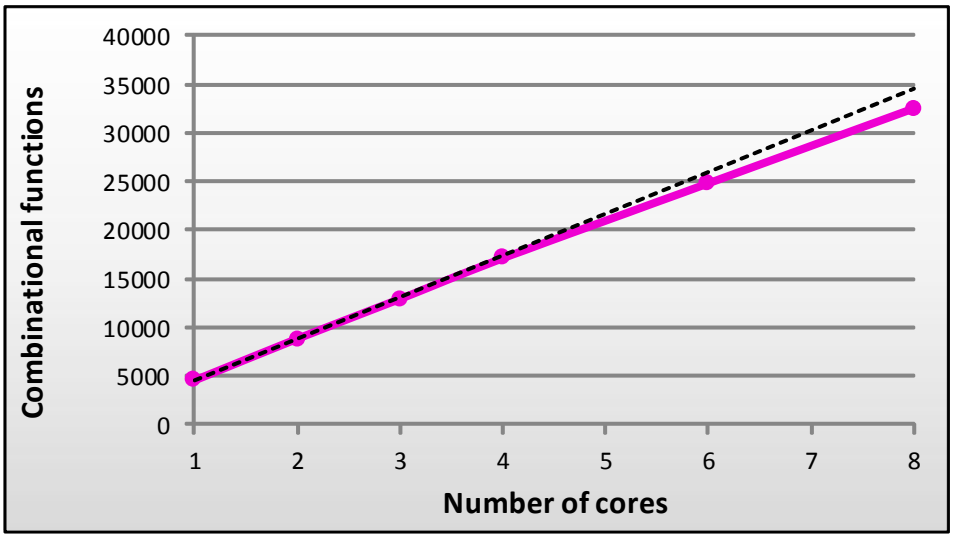



Figure 6.7: FPGA combinational functions vs. number of cores in an SG-Multi system

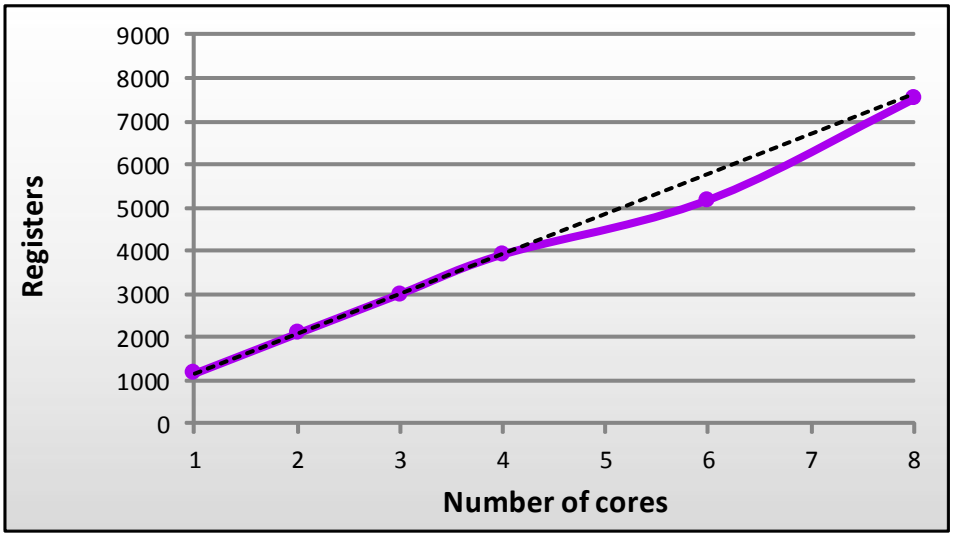



Figure 6.8: FPGA registers vs. number of cores in an SG-Multi system

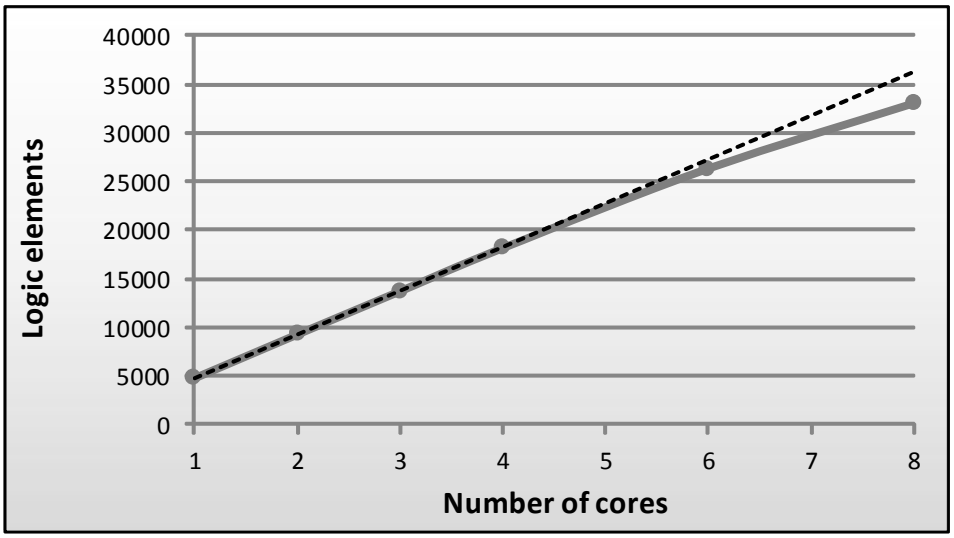



Figure 6.9: FPGA total logic elements vs. number of cores in an SG-Multi system

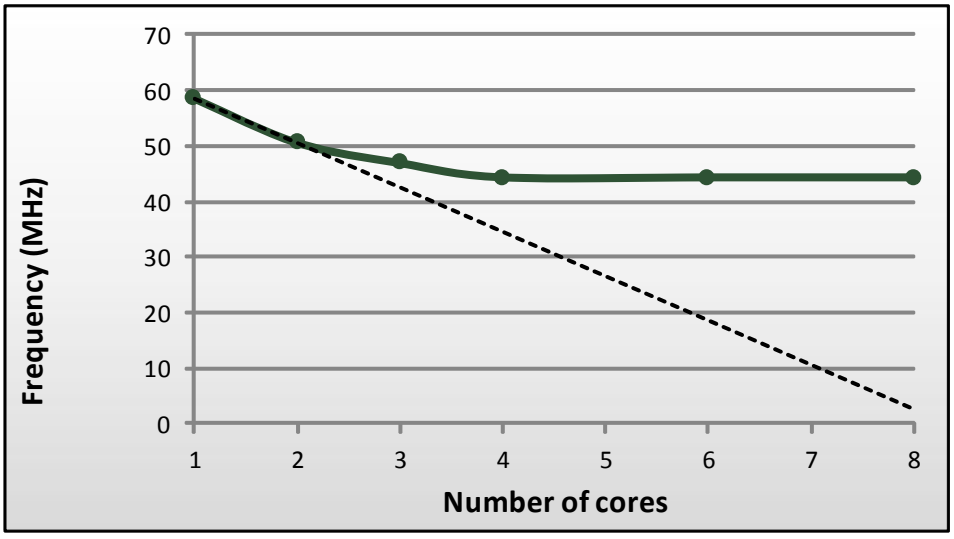



Figure 6.10: FPGA maximum clock frequency vs. number of cores in an SG-Multi system

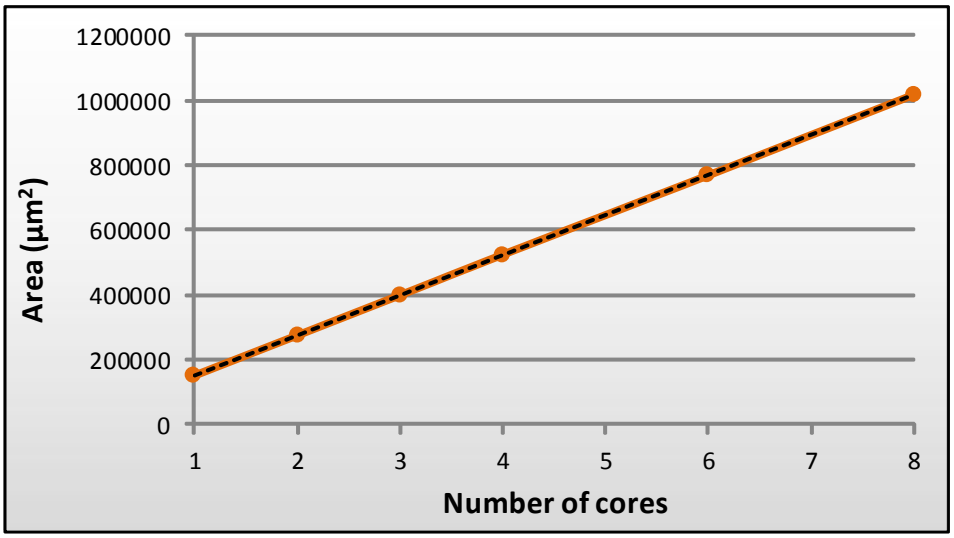



Figure 6.11: ASIC combinational area vs. number of cores in an SG-Multi system

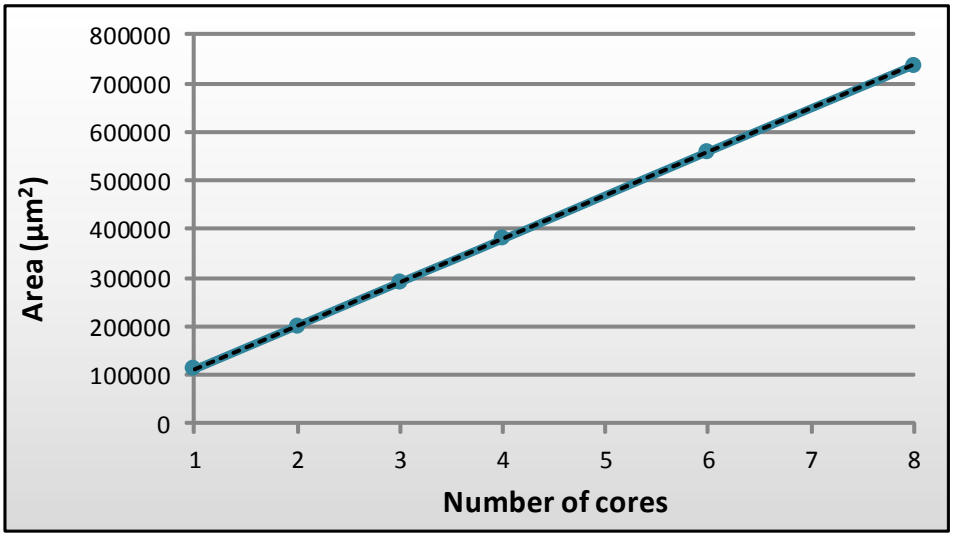



Figure 6.12: ASIC noncombinational area vs. number of cores in an SG-Multi system

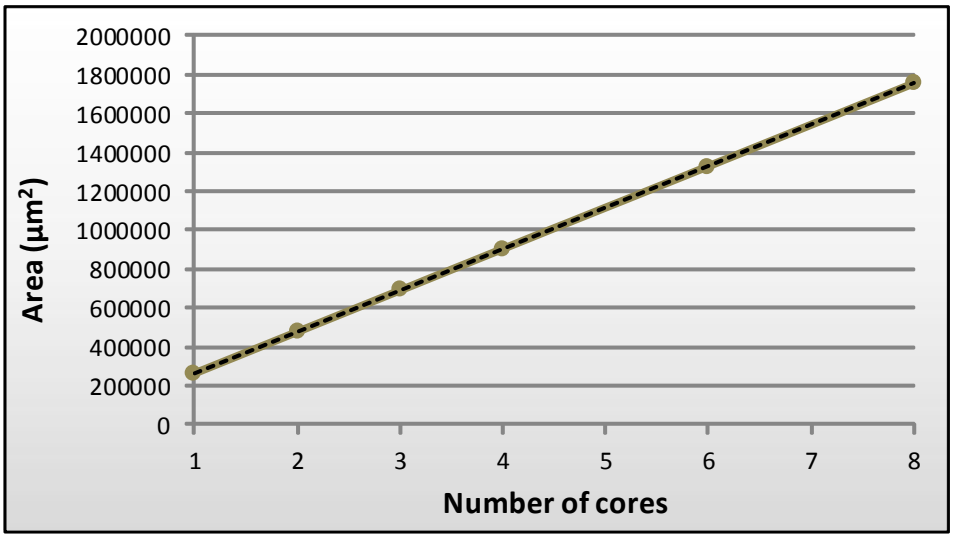



Figure 6.13: ASIC total area vs. number of cores in an SG-Multi system

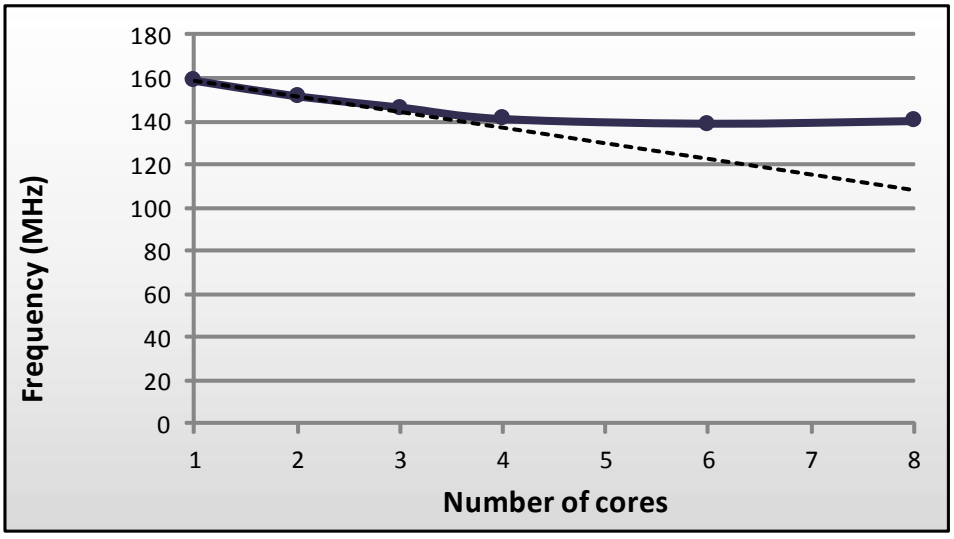



Figure 6.14: ASIC maximum clock frequency vs. number of cores in an SG-Multi system

results behave similarly to those of the FPGA tests, stabilizing around 140 MHz for higher numbers of cores. These results lead to the conclusion that SG-Multi is scalable when implemented as an ASIC.

## 6.3 Simultaneous Transaction Performance

The purpose of these tests is to demonstrate the functional correctness of the SG-Multi architecture and its performance-enhancing features. By design, SG-Multi is intended to support multiple simultaneous transactions with no slowdown, where each transaction occurs with a different slave device. Accordingly, the non-interfering transaction tests show the number of clock cycles required by a synthetic benchmark application when executed with a varying number of cores all interacting with different slaves. SG-Multi also includes a snooping feature that, under certain circumstances described in Chapter 4, permits multiple master devices to complete simultaneous transactions with the same slave. Thus, the interfering transaction tests are intended to demonstrate that the snooping functionality, when its prerequisite conditions are met, is able to remove all contention-related slowdown that would typically be present when multiple masters attempt to access the same slave.

These tests are based on the same SG-Multi reference systems used in other tests. The benchmarking circuitry is identical to that described in Section 6.1.1, and the benchmark application simply performs 100 32-bit read operations from SRAM. The SRAM controller requires 4 clock cycles per 32-bit transaction, so the actual loading time in a single-core system is expected to be 400 cycles. For non-interfering transaction tests, only one core executes this benchmark application while the others interact with the other slave devices in various ways. For interfering transaction tests, all cores involved in the test are connected to a private instance of the ROM component containing the same executable code implementing the test; as a result, all interference happens on attempts to access SRAM as opposed to instruction fetch operations.

### 6.3.1 Non-interfering Transactions

One of SG-Multi's most crucial features is its ability to support multiple simultaneous transactions between different slaves. The intention of this test is to verify that behaviour by showing that the benchmark application completes in the same number of clock cycles irrespective of the number of cores in the system and their interaction with slaves other than the SRAM controller. This experiment was conducted on SG-Multi systems containing a range of 1 to 4 processor cores, results shown in Table 6.3. As reflected in the results, the number of cycles remains constant across all tests, indicating that SG-Multi can correctly support multiple simultaneous non-interfering transactions.

Table 6.3: Non-interfering transaction benchmark results

| **Cores** | **Cycles** |
|---|---|
| 1 | 510 |
| 2 | 510 |
| 3 | 510 |
| 4 | 510 |

### 6.3.2 Interfering Transactions

Bus snooping is designed to remove the unnecessary waiting associated with losing arbitration when the other masters in the system are requesting the same transaction and thus the requested results are already being made available. This experiment quantifies the performance gain that can be realized from this technique, assuming its prerequisite conditions have been met. In theory, if all processors in the system continually request the same transaction, the amount of time spent should grow linearly with snooping disabled whereas the amount of time spent should remain constant with it enabled.

This experiment was conducted with SG-Multi systems containing a range of 1 to 4 processor cores. Each core is provided with an identical binary stored on a ROM device only accessible by that specific core, and the number of clock cycles taken to complete the benchmark is recorded. The test results are validated by performing the test both with snooping enabled and with it disabled. Cycle count measurements are shown in Table 6.4.

Table 6.4: Interfering transaction benchmark results

| **Cores** | **Cycles (snooping)** | **Cycles (no snooping)** | **Difference** |
|---|---|---|---|
| 1 | 510 | 510 | – |
| 2 | 510 | 811 | +301 |
| 3 | 510 | 1211 | +701 |
| 4 | 510 | 1611 | +1101 |

There is a consistent increment of 400 cycles as the number of cores increases to 3 and then again to 4. Since a single core spends 400 cycles performing memory load operations, the addition of another core whose execution is precisely timed to compete directly with the existing cores should in theory add an additional 400 cycles to the total latency of the benchmark, which matches with the results shown. The difference is less pronounced between the single-core and the dual-core cases because the measurement of 510 cycles includes the test set-up and tear-down code, the effect of which is removed when examining differences between results for higher numbers of cores.

With snooping enabled, the number of cycles is independent of the number of cores. Each core is able to complete its memory transaction at the same time as the others, so bus snooping demonstrably avoids all latency associated with waiting to win a round of arbitration.

# Chapter 7

# Conclusions

The primary goal of the work described in this thesis was to create an integrated solution that greatly simplifies and expedites the process of creating a multi-core processor whose internal components are interconnected in a virtually arbitrary arrangement. It was motivated by a desire to bridge the gap between research and practice, which led to the avoidance of constraints, restrictions, and assumptions that require infeasible modifications to existing widely-used hardware components and to the separation of high-level logic design from final hardware implementation. In particular, existing industry-supplied processor cores are supported, and all output is in pure HDL form containing only vendor-agnostic constructs. Experiments have demonstrated the feasibility, correctness, and scalability of the solution proposed in this thesis.

This chapter begins by summarizing the primary contributions of this work, SG-Multi and SG-Multi Designer. The former is an interconnection architecture optimized for multi-core processors and the latter is a tool framework to facilitate the creating of systems built on that architecture. It then briefly states the scope of this work's application domain, presents the major experimental conclusions, and discusses potential future work.

## 7.1 Contributions Summary

SG-Multi is a multi-bus system that employs slave-side arbitration to enable it to support multiple unrelated simultaneous transactions. In certain cases, its bus snooping performance-enhancing feature allows multiple masters to complete transactions at the

same time with the same slave device. Its design goal of universal adaptability is supported by its use of bus adapters, which experimentation has shown can be designed to introduce no clock cycle latency. All transactions in SG-Multi are pipelined in two stages– the address phase and the data phase–with arbitration taking place in the former instead of being separated into its own pipeline stage. To improve performance, the SG-Multi signaling protocol requires slave devices to signal their status information for a given clock cycle at the beginning of that clock cycle rather than at the end.

SG-Multi Designer is a tool framework that facilitates the rapid construction of multi-core processors based on SG-Multi. The abstraction model presented to the end-user simplifies the design process considerably, even to an extent greater than existing competitive solutions. Each hardware device available to be included in an SG-Multi system design is encapsulated in a module, which consists of an executable file and an XML file, the former producing HDL code implementing the device when executed and the latter describing how SG-Multi Designer's user interface should interact with the executable file.

This research project is intended to be applicable both for experimenting with multi-core systems in a research setting and for creating multi-core processors in an industrial setting; SG-Multi Designer targets hardware chip designers in both of these contexts. While the primary intention is to target systems that can be fabricated as an ASIC, implementing systems onto an FPGA for prototyping and testing purposes is also supported.

## 7.2 Experiment Conclusions

Initial tests sought to verify the potential for creating bus adapters that add no clock cycles of latency and to quantify the hardware cost of using SG-Multi versus using a processor's native interconnection architecture. Compared to AHB-Lite, the native bus architecture of the ARM Cortex-M0 processor featured in all SG-Multi reference systems, SG-Multi has been shown not to require any additional clock cycles when completing the same benchmark application. Thus, the AHB-Lite comparison tests affirm the possibility of creating high-quality bus adapters that introduce no clock-cycle latency. Furthermore, when both systems were prototyped on an FPGA, the difference in $f_{\max}$ was less than 3%, and the extra logic elements, combinational functions, and registers consumed by SG-Multi each accounted for less than 1.5% of the total device resources, well within reason given the

added functionality SG-Multi offers over AHB-Lite. When compiled for ASIC fabrication the differences were more pronounced; AHB-Lite consumes approximately 20% to 25% less total area, and the maximum clock frequency is between 35% and 50% higher with AHB-Lite. However, SG-Multi shows substantially less process variation than AHB-Lite in terms of maximum frequency, having a maximum frequency range 46 MHz smaller than that of AHB-Lite. Furthermore, the removal of arbiters reduces the clock frequency difference to between 10% and 20%, suggesting that the current implementation of arbitration is a prime candidate for future improvement.

Experiments with SG-Multi reference systems containing between 1 and 8 cores have shown that SG-Multi is scalable, both when prototyped on an FPGA and when compiled for fabrication as an ASIC. A scalable architecture grows linearly in terms of its hardware resource requirements when additional cores are added to the system, and the ASIC-oriented scalability test results precisely match this evaluation criterion for all three metrics: combinational area, noncombinational area, and total area. Better results were obtained with the FPGA-oriented scalability tests, with slightly sublinear growth observed as the number of cores increased past 3 or 4. In both cases, maximum achievable clock frequency dropped with the addition of the first few cores but stabilized and remained relatively constant with 4 or more cores in the system. It is therefore concluded that SG-Multi is scalable both when implemented on an FPGA and when compiled for fabrication as an ASIC.

The final set of tests targeted the functional correctness of SG-Multi's design features. SG-Multi's ability to support multiple simultaneous transactions was evaluated using a simple benchmark application executed on a single core while the variable number of other cores communicated with other slaves in the system. Irrespective of the number of cores present, the benchmark application took exactly 510 cycles to complete, successfully demonstrating SG-Multi's proper behaviour. The same benchmark was executed on multiple cores to test SG-Multi's bus snooping feature; while linear growth in terms of the cycles to completion was observed with snooping disabled, each core consistently required exactly 510 cycles to complete the benchmark with snooping enabled, clearly demonstrating both the potential benefits realizable by and the proper functionality of bus snooping.

Considering all of the experiments conducted to evaluate SG-Multi, it is concluded that this research project was successful. Although it is just a starting point for future growth

and development, SG-Multi is demonstrably capable of achieving its design goals. As it is the underlying architecture upon which SG-Multi Designer is built, and as the reference systems are an accurate representation of what SG-Multi Designer is capable of producing, it is further concluded that the abstraction model provided by SG-Multi Designer does not over-simplify the design process in a way that excessively degrades system performance or introduces unreasonable additional hardware resource requirements.

## 7.3 Future Research Directions

SG-Multi and SG-Multi Designer represent a starting point, demonstrating the feasibility of creating multi-core processors that consist of cores natively supporting a wide variety of interconnection architectures, including those originally designed with single-core operation in mind. In terms of hardware design, a highly simplified abstraction model is possible to achieve without over-simplifying the design process or causing scalability issues. There are, however, several ways to build upon this work in order to increase its utility in practice.

The first and most direct area for future work is performing additional optimization on the SG-Multi signalling protocol itself; experimental results, while positive, reveal arbitration as a specific area where such optimization is likely to be beneficial. Second, since the goal of universal adaptability depends on the existence of appropriate bus adapters, it follows that more bus adapters must be created for SG-Multi to reach its full potential. Third, SG-Multi currently makes no attempts to accommodate components operating at different clock frequencies. It is extremely likely that a system containing components of different performance levels will be required to accommodate this, so the addition of support for multiple clock domains is important. Fourth, SG-Multi could be made to support dynamically reconfigurable connections between master devices, for instance, two cores that communicate directly. Reconfigurability would allow a system to adjust itself based on the particular application to be executed, in essence becoming a dynamically reconfigurable application-specific accelerator. This section explores each of these areas for future work.

### 7.3.1 Basic Arbitration Improvements

Chapter 6 shows that the current implementation of arbitration is costly in terms of performance. It should be noted that the design of the arbiter component was not one of the main focal points in designing SG-Multi; however, as it is desirable to maximize the speed of an SG-Multi system, a re-examination of its arbitration portion likely to be beneficial. The current version of the arbiter component is a combinational circuit through which signals must propagate during the address phase cycle. This gives rise to two possible areas of emphasis: the combinational circuitry itself and its placement directly in the address phase.

Improving the combinational circuitry of the arbiter component would either involve selecting a new logical implementation of the hybrid priority scheme currently used, switching to a different arbitration scheme entirely, or some combination thereof. A combined approach to improvement is certainly viable; works such as [40], and more recently [41], demonstrate the feasibility of creating an extremely scalable arbiter that implements a variety of different arbitration schemes. For example, based on the SG-Multi reference systems, which have a maximum of 4 requestors per slave device, the type of arbiter described in [41] would introduce a delay of less than 1 ns and consume fewer than 100 2-input NAND gates.

The other alternative involves a re-evaluation of arbitration taking place in the address phase. It could be moved to its own cycle, similar AHB [26], but this would introduce latency into all transactions. Either a slave would have to signal `SGWAIT` one cycle earlier than it currently does–meaning that all transactions must take at least two cycles in the data phase–or certain bus adapters, such as the AHB-Lite bus adapter, would be forced to add a cycle of latency to all requested transactions. The suitability of this modification to SG-Multi depends on whether or not the clock frequency gain is high enough to offset the time cost of adding latency cycles to the majority of transactions.

### 7.3.2 Other Bus Adapters

The utility of SG-Multi depends largely on its ability to satisfy its design goals in a practical manner. To this end, an AHB-Lite bus adapter on its own is insufficient; while AHB-Lite is able to demonstrate that it is feasible to construct high-quality bus adapters for SG-

Multi, taking advantage of SG-Multi's adaptability depends upon the existence of other bus adapters. Creating these bus adapters is therefore a key step towards improving the SG-Multi system as a whole.

A logical first step towards achieving this is to begin with bus architectures that are similar by design to AHB-Lite; the existence of a bus adapter for AHB-Lite is a strong indicator that a bus adapter could be created for these architectures without adding latency. Chapter 2 describes some examples of such architectures, including Altera's Avalon-MM [21] and IBM's CoreConnect PLB [28]. While this may require either modifying SG-Multi to support some architecture-specific features or omitting these features entirely, the basic signalling protocols are generally similar in terms of completing transactions. In the lattermost case, it is clearly feasible to create a bus adapter, as evidenced by the availability of a bridge between CoreConnect PLB and ARM AHB [30].

Bus adapters for these similar architectures provide a solid groundwork upon which to build, but their creation does not mark an endpoint. A much greater applicability for SG-Multi can be realized by creating adapters for ARM AXI [35], a very popular architecture used in high-performance systems. The existence of a bridge between AXI and AHB-Lite [42] is a strong indication that such a bus adapter can be created and made to work. However, to maximize the performance of the adapter, it may be necessary to evolve SG-Multi to a form that can better accommodate AXI while also remaining fully compatible with AHB-Lite and others.

### 7.3.3 Multiple Clock Domains

A heterogeneous multi-core processor is characterized by its composition of non-identical cores. While some applications may suffice with this definition strictly referring to a distinction between general-purpose cores and specialized accelerators, it is not generally applicable. In particular, it is often a requirement that the system support components operating at different clock frequencies, such as having multiple cores all offering different amounts of computational power. As discussed in Chapter 2, this is the approach taken in NVIDIA's Tegra family of mobile processors [10]. The fact that the current form of SG-Multi makes no attempt to overcome clock frequency boundaries therefore places a limitation on its usefulness in certain applications.

This problem can be solved using either a synchronous or an asynchronous approach,

and solutions based on both approaches are readily available in the literature. In [43], for instance, a synchronous approach is suggested for bridging between 100 MHz AHB and 10 MHz Industry Standard Architecture (ISA), an older bus architecture for communicating with peripherals. The approach described involves a finite state machine coupled with a cycle counter used together not only to provide the clock domain boundary-crossing functionality but also to bridge the two architectures together. An SG-Multi approach based on this general framework would involve this clock domain-crossing logic being integrated into the bus adapters; while the framework uses an adapter to bridge signals from a faster controlling bus to a slower peripheral bus, it is conceivable that this directionality could be reversed if needed.

The alternative to a synchronous approach is an asynchronous approach, an example of which is presented in [44]. This solution places a high-performance asynchronous crossbar at the centre of the system, with each system component being clocked individually without any phase-locked loops or other components needed to synchronize the clocking. By design, the problem of crossing clock domains is essentially avoided altogether. One possibility for integrating this approach into SG-Multi is to change the architecture such that a large asynchronous crossbar switch becomes its basis. SG-Multi's emphasis on multi-core processors may lead to an excessively-sized crossbar switch being necessary in system designs featuring an abundance of cores, so the scalability of this solution would need to be studied. While intuition might suggest that a crossbar-based approach cannot scale to large multi-core systems, recent work, such as [45], suggests otherwise.

### 7.3.4 Reconfigurable Inter-core Communication

A reconfigurable processor in-and-of itself is not a completely new idea; existing literature already suggests how one might construct such a device. Building SG-Multi into a reconfigurable processor, however, approaches the problem from an entirely different perspective. SG-Multi was created with the idea that one should be able to integrate existing industry-supplied cores in the same system, even those not designed for such integration. It follows, then, that the reconfigurable extension should support the same thing. Reconfigurable processors in their current form are generally restricted for research use. A reconfigurable SG-Multi processor, however, would have a wide range of potential applications and, since it makes use of existing industry-supplied cores, can successfully bridge the gap between

research and practice.

Existing research on this subject can generally be divided into two categories. The first category involves integrating a physically reconfigurable hardware element with the rest of the system. In this sense, the *reconfigurable* element of the processor stems from the fact that an application can use the reconfigurable element in an application-specific manner; an example of such an approach is described in [46]. FPGAs are also a prime target of research related to reconfigurable computing. A system whose final form is implemented on an FPGA may be made reconfigurable if the FPGA supports partial dynamic reconfiguration. Xilinx is exploring this area for its own FPGA products and a specific approach is presented in [47], although research towards efficient and effective methods for implementing partial dynamic reconfiguration is ongoing. Modelling SG-Multi on this area of research would involve placing the interconnection architecture onto a dynamically reconfigurable hardware element and potentially including physically reconfigurable hardware in the place of master and slave devices. While potentially beneficial, this research direction is unlikely to positively impact SG-Multi at an architectural level.

The second category for reconfigurable processor research proposes architectures that require supporting functionality to be provided from the individual processing elements. In [48], for example, the proposed architecture depends on the existence of architecture-specific instructions that each processor core can execute. The limitation of solutions in this class is that, in order to be applicable in the real world, industry is required to implement the system in the design of its own processors; for instance, ARM must add support for these instructions and provide the logic necessary to implement that functionality. SG-Multi, on the other hand, requires no such changes, and the proposed vision for a reconfigurable version of it will similarly not require any modifications.

A reconfigurable version of SG-Multi would be one in which the interconnection between processor cores can be dynamically changed, a much coarser granularity for reconfiguration than has traditionally been studied. The primary benefit of making processors reconfigurable in this fashion is that they can be adapted on a per-application basis to suit virtually any possible type of algorithm model. For instance, an application designed to be pipelined can be greatly accelerated by executing it on a processor in which cores are interconnected in this arrangement. However, as existing industry multi-core processors are not generally reconfigurable in this manner, a processor would either be fixed with a particular model

in mind or contain hard-wired support for every model that the designers predict would be needed. Thus, a reconfigurable version of SG-Multi promises to enable industry-facing processors to take advantage of the benefits of reconfigurability without being held back by these limitations.

# References


[1] M. Stürmer, G. Wellein, G. Hager, H. Köstler, and U. Rüde, "Challenges and potentials of emerging multicore architectures," in *High Performance Computing in Science and Engineering, Garching/Munich 2007* (S. Wagner, M. Steinmetz, A. Bode, and M. Brehm, eds.), pp. 551–566, Springer Berlin Heidelberg, 2009.

[2] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pp. 67–72, November 2006.

[3] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," *SIGARCH Comput. Archit. News*, vol. 17, pp. 2–15, April 1989.

[4] Intel Corp., *2nd Generation Intel Core Processor Family Desktop Datasheet Vol. 1*, December 2011.

[5] A. Hung, W. Bishop, and A. Kennings, "Symmetric multiprocessing on programmable chips made easy," in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 240–245 Vol. 1, March 2005.

[6] C. Cascaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik, "A taxonomy of accelerator architectures and their programming models," *IBM Journal of Research and Development*, vol. 54, pp. 5:1–5:10, September–October 2010.

[7] X. Fang and S. Chen, "The design and algorithm mapping of a heterogeneous multicore processor for SDR," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pp. 1086–1089, November–December 2008.

[8] W. Han, Y. Yi, X. Zhao, M. Muir, T. Arslan, and A. Erdogan, "Heterogeneous multi-core architectures with dynamically reconfigurable processors for wireless communication," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 1–6, July 2009.

[9] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 81–92, December 2003.

[10] NVIDIA Corp., *NVIDIA Tegra Multi-processor Architecture*, February 2010.

[11] G. Paya-Vaya, J. Martin-Langerwerf, and P. Pirsch, "RAPANUI: A case study in rapid prototyping for multiprocessor system-on-chip," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pp. 215–221, August 2007.

[12] Cadence Design Systems Inc., *Concurrent Hardware/Software Development Platforms Speed System Integration and Bring-Up White Paper*, May 2011.

[13] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *Micro, IEEE*, vol. 27, pp. 46–57, March–April 2007.

[14] T. Oguntebi, S. Hong, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "FARM: A prototyping environment for tightly-coupled, heterogeneous architectures," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pp. 221–228, May 2010.

[15] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "RAMP Blue: A message-passing manycore system in FPGAs," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 54–61, August 2007.

[16] L. Barroso, S. Iman, M. Dubois, and K. Ramamurthy, "RPM: A rapid prototyping engine for multiprocessor systems," *Computer*, vol. 28, pp. 26–34, February 1995.

[17] Cadence Design Systems Inc., *Cadence Rapid Prototyping Platform Datasheet*, May 2011.

[18] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, 2001.

[19] G. Schelle and D. Grunwald, "Exploring FPGA network on chip implementations across various application and network loads," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 41–46, September 2008.

[20] Altera Corp., "Qsys interconnect," in *Quartus II Handbook, Volume 1: Design and Synthesis*, ch. 7, Altera Corp., November 2011.

[21] Altera Corp., *Avalon Interface Specifications*, May 2011.

[22] Altera Corp., *SOPC Builder User Guide*, December 2010.

[23] Altera Corp., *NIOS II Processor Reference Handbook*, May 2011.

[24] ARM Ltd., *AMBA 3 AHB-Lite Protocol Specification*, March 2010.

[25] ARM Ltd., *Cortex-M0 Devices Generic User Guide*, October 2009.

[26] ARM Ltd., *AMBA Specification (Rev. 2.0)*, May 1999.

[27] ARM Ltd., *Multi-layer AHB Overview*, May 2004.

[28] IBM Corp., *128-bit Processor Local Bus Architecture Specifications Version 4.7*, May 2007.

[29] Xilinx Inc., *MicroBlaze Processor Reference Guide UG081 (v9.0)*, January 2008.

[30] IBM Corp., *AHB to PLB Bridge Core*, October 2004.

[31] M. Winter and G. Fettweis, "Interconnection generation for system-on-chip design," in *System-on-Chip, 2006. International Symposium on*, pp. 1–4, November 2006.

[32] Altera Corp., "Creating a system with Qsys," in *Quartus II Handbook, Volume 1: Design and Synthesis*, ch. 5, Altera Corp., November 2011.

[33] Altera Corp., *NIOS II Hardware Development Tutorial*, May 2011.

[34] Xilinx Inc., "Xilinx Platform Studio (XPS)," 2012.

[35] ARM Ltd., *AMBA AXI and ACE Protocol Specification*, October 2011.

[36] Tensilica Inc., *Xtensa Processor Developer's Toolkit - Product Brief*, March 2011.

[37] Tensilica Inc., "Xtensa Configurable Processors - Overview," 2012.

[38] Tensilica Inc., *TIE - The Fast Path to High Performance Embedded SOC Processing*, 2009.

[39] Terasic Technologies Inc., "Altera DE2 Board."

[40] C. Savin, T. McSmythurs, and J. Czilli, "Binary tree search architecture for efficient implementation of round robin arbiters," in *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, vol. 5, pp. 333–336, May 2004.

[41] J. M. Jou, Y.-L. Lee, and S.-S. Wu, "Efficient design and generation of a multi-facet arbiter," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, pp. 111–114, June 2010.

[42] Xilinx Inc., *LogiCORE IP AXI to AHB-Lite Bridge (v1.00a)*, June 2011.

[43] S. Choi and S. Kang, "Implementation of an on-chip bus bridge between heterogeneous buses with different clock frequencies," in *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on*, pp. 530–534, July 2005.

[44] A. Lines, "Asynchronous interconnect for synchronous SoC design," *Micro, IEEE*, vol. 24, pp. 32–41, January–February 2004.

[45] B.-J. Hong, K.-S. Cho, S.-H. Kang, S.-Y. Lee, and J.-D. Cho, "On the configurable multiprocessor soc platform with crossbar switch," in *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, pp. 1087–1090, December 2006.

[46] L. Yan, B. Wu, Y. Wen, S. Zhang, and T. Chen, "A reconfigurable processor architecture combining multi-core and reconfigurable processing unit," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 2897–2902, June–July 2010.

[47] M. Santambrogio, P. Grassi, D. Candiloro, and D. Sciuto, "Analysis and validation of partially dynamically reconfigurable architecture based on xilinx fpgas," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–4, April 2010.

[48] M. Watkins and D. Albonesi, "ReMAP: A reconfigurable heterogeneous multicore architecture," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 497–508, December 2010.

# APPENDICES

# Appendix A

# SG-Multi Protocol Specification

This Appendix contains a complete specification for SG-Multi's signalling protocol. It assumes familiarity with Chapter 3, which provides a high-level overview of the entire SG-Multi architecture. The intended audience is the designer of a hardware device that supports communication with an SG-Multi system.

Signal descriptions are presented first, followed by the general rules for sending data along wide data paths. Transaction details are then provided, first for the master device interface and second for the slave device interface.

## A.1 Signal Descriptions

SG-Multi defines global signals shared by all devices and device-specific signals used for data exchange. Several of these signals are common to both the master device interface and the slave device interface, but some exist only at one of these interfaces. The subsections that follow list and describe all of these signals.

SG-Multi supports various widths for address and data busses, where each can be sized independently of the other. Supported sizes, measured in bits, are powers of 2 ranging from 32 to 256. When signal widths are provided, the symbol $A$ denotes the width of the address bus, and the symbol $D$ denotes the width of the data busses. When signal widths are omitted, the width is a single bit.

### A.1.1 Global Signals

Two global signals are used in SG-Multi, one as a clock source and one as an asynchronous reset. Names, types, and descriptions are listed in Table A.1.

Table A.1: List, types, and descriptions of global SG-Multi signals

| Signal Name | Type | Description |
|---|---|---|
| SGCLK | Input | Clock signal used to drive transactions |
| SGRESETn | Input | Asynchronous active-low reset signal |

### A.1.2 Common Signals

Several interface signals exist at both the master device interface and the slave device interface. Signals flow from master devices to slave devices or vice versa; thus, if a signal is an input for one type of device, it is the output for the other type. Interface signals common to both master and slave device interfaces are listed and described in Table A.2.

Table A.2: List, directionalities, and descriptions of common SG-Multi interface signals

| Signal Name | Direction | Description |
|---|---|---|
| SGADDR[$A-1$:0] | Master to slave | Memory address of interest |
| SGSIZE[2:0] | Master to slave | Specifies the size of the current transaction |
| SGWnR | Master to slave | Specifies whether the current transaction is for reading (low) or writing (high) |
| SGRDATA[$D-1$:0] | Slave to master | Data resulting from a read operation |
| SGWDATA[$D-1$:0] | Master to slave | Data to be written during a write operation |
| SGWAIT | Slave to master | Indicates that more time is required to complete the current transaction |
| SGERROR | Slave to master | Indicates that the slave encountered an error while processing the current transaction |

The transaction sizes supported range from 8 bits (1 byte) to the width of the data busses, with the actual size specified by SGSIZE. Possible values are listed in Table A.3.

Table A.3: Values for specifying the size of an SG-Multi transaction

| Size (bits) | Binary Value |
|---|---|
| 8 | 3'b000 |
| 16 | 3'b001 |
| 32 | 3'b010 |
| 64 | 3'b011 |
| 128 | 3'b100 |
| 256 | 3'b101 |

### A.1.3 Master Interface Signals

SG-Multi master-specific functionality is captured in additional interface signals only present at the master device interface. These signals are listed and described in Table A.4.

Table A.4: List, types, and descriptions of SG-Multi master-specific signals

| Signal Name | Type | Description |
|---|---|---|
| SGREQ | Output | Indicates that a master device wishes to start a transaction |
| SGGRANT | Input | Indicates that the master device has been granted permission to start a transaction |

Master devices are not expected to be concerned with the existence and status of other devices in the system when requesting transactions; SGREQ can be asserted whenever a master device is ready to begin a transaction. However, they are required to request permission using SGREQ and wait for SGGRANT to be asserted before proceeding with a transaction.

### A.1.4 Slave Interface Signals

SG-Multi slave-specific functionality is captured in additional interface signals only present at the slave device interface. These signals are listed and described in Table A.5.

The SGACTIVATE signal is used to inform a slave device that an incoming transaction request is available for processing. A slave is not to respond to transaction requests unless SGACTIVATE is asserted. If a slave determines that a particular transaction is safe to be snooped, it asserts SGSNOOP to signal this information. If a transaction has no side-effects,

Table A.5: List, types, and descriptions of SG-Multi slave-specific signals

| Signal Name | Type | Description |
|---|---|---|
| SGACTIVATE | Input | Activation signal, asserted when a transaction is being requested |
| SGSNOOP | Output | Indicates that the requested transaction is safe to snoop |

and the value of the address of interest does not change between consecutive read transactions, then in general it may be safe to snoop that transaction; for instance, if the slave is a memory storage device, then transactions might be safe to snoop. However, transactions with a hardware controller may only be safe to snoop in more limited circumstances, such as reading from or writing to the contents of a data register. In general, transactions that are data-destructive or that have slave-specific side-effects are not safe to be snooped and should have this functionality disabled.

### A.1.5 Data Bus Alignment

All transactions in SG-Multi are memory address-aligned according to the size of the transaction; for instance, a 16-bit (2-byte) transaction is aligned on a 2-byte address boundary, and a 32-bit (4-byte) transaction is aligned on a 4-byte address boundary. The data busses SGRDATA and SGWDATA are similarly considered to be aligned based on their width, and transactions smaller than the width of these data busses make use of specific bit positions within them based on how their alignment compares to that of the data busses. Table A.6 explains this mechanism in more detail for a 32-bit data bus width, though the principles contained therein extend to wider busses in a similar fashion. The address offset value is the offset, in bytes, from a memory address aligned to the size of the data bus width; in this example, it ranges from 0 to 3. It is recommended that the unused bit positions be driven to whatever values simplify the logic design of the device. This could mean, for instance, driving them to all '0' or replicating whatever is being driven to the bit positions in use.

Table A.6: Bit positions used for smaller transactions on wide data busses

| Transaction Size | Address Offset | Positions Used |
|---|---|---|
| 8 bits | 0 | [7:0] |
| | 1 | [15:8] |
| | 2 | [23:16] |
| | 3 | [31:24] |
| 16 bits | 0 | [15:0] |
| | 2 | [31:16] |
| 32 bits | 0 | [31:0] |

## A.2 Transaction Details

All transactions in SG-Multi are pipelined and consist of two stages: an *address phase* and a *data phase*. The address phase, the first part of a transaction, is when control information is exchanged between a master device and a slave device, and the data phase, which comes immediately after the address phase, is when the data transfer occurs. While the majority of a transaction involves common signals, there are some interface-specific differences between masters and slaves. The following subsections explain the procession of a transaction at each of the two interfaces.

### A.2.1 Master Interface Signalling

A master device request a transaction by initiating an address phase, which involves driving `SGADDR`, `SGWnR`, `SGSIZE`, and `SGREQ`. It must wait for `SGGRANT` in response before the transaction can proceed, which may occur in the same cycle as `SGREQ` is asserted or some cycle thereafter. A rising clock edge with `SGGRANT` asserted signifies that the slave has accepted the transaction, ending the address phase and beginning the data phase. During the data phase, the master must respond to status information the slave sends via `SGWAIT` and `SGERROR`; it must drive `SGWDATA` at this time if it requested a write transaction. It may also prepare for the next transaction, but it must not drive `SGREQ` until the final cycle of the current transaction's data phase. Figure A.1 shows a timing diagram for two consecutive transactions that complete in the shortest possible time, the first a read and the second a write. For the sake of simplicity the slave response signals `SGERROR` and `SGWAIT` are not

shown; these are deasserted throughout this example.

A master may not see SGGRANT immediately as shown in Figure A.1. In the event that SGGRANT is not asserted, the current address phase is extended; the master device must not deassert SGREQ and it cannot advance to the address phase for the following transaction. This delayed behaviour is shown in Figure A.2; for simplicity, only the directly relevant signals are shown.

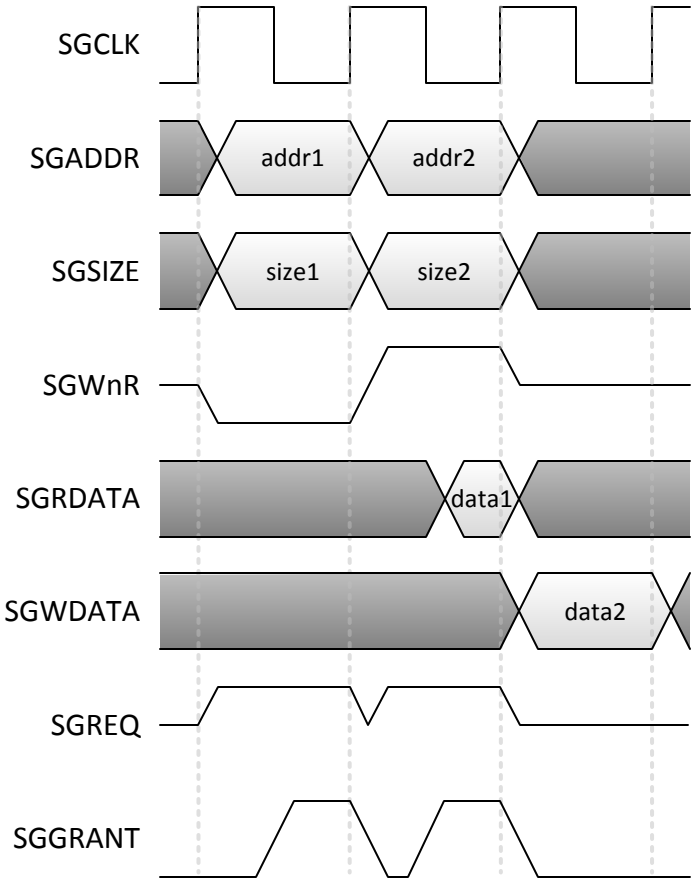



Figure A.1: Example read and write SG-Multi transactions at the master interface with no waiting

A master device may also be delayed at the request of a slave, through the SGWAIT signal. If this signal is asserted at the beginning of the clock cycle, then that cycle is a

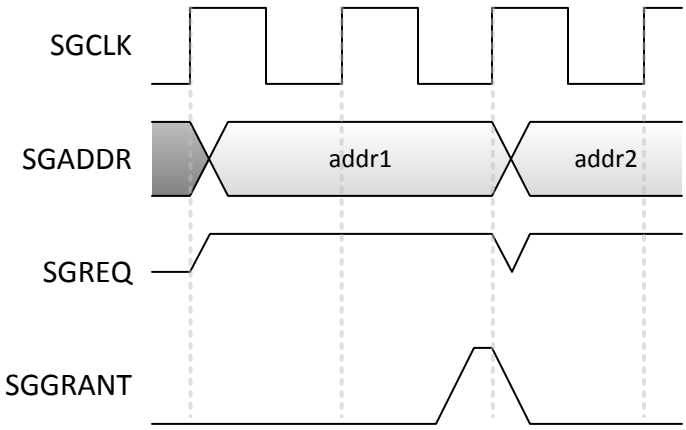



Figure A.2: Example of an SG-Multi master interface transaction with a delayed grant signal

wait cycle and the current transaction's data phase is extended. Conversely, if this signal is deasserted at the beginning of a clock cycle, then that cycle is the final cycle of the current transaction's data phase. In the case of a read transaction, the master must only sample `SGRDATA` at the end of a cycle that began with `SGWAIT` deasserted; an example of such a transaction is shown in Figure A.3. In the case of a write transaction, it must keep `SGWDATA` for the entire duration of the data phase, until the end of a cycle that began with `SGWAIT` deasserted; this is demonstrated in Figure A.4. A master must not begin arbitration for its next transaction in a clock cycle that begins with `SGWAIT` asserted, a fact demonstrated in Figures A.3 and A.4.

A slave may use `SGERROR` to signal to the master that an error has occurred during the current transaction. The `SGWAIT` and `SGERROR` signals are mutually exclusive; a slave will not signal both at the rising edge of a clock cycle. The signalling protocol is the same with and without `SGERROR`, and a master can begin a new transaction in a clock cycle that begins with `SGERROR` asserted, though the master should take note of the transaction error and perform any appropriate action in response to it.

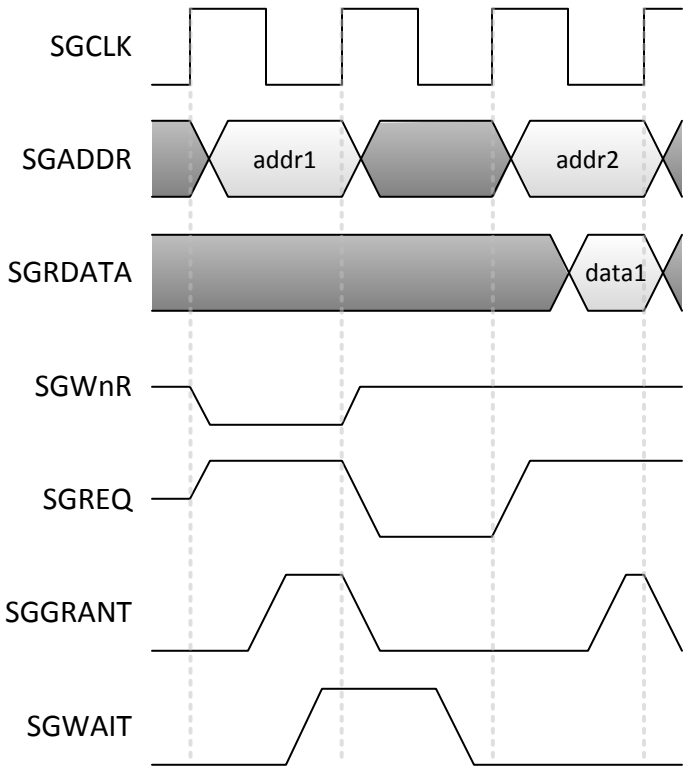



Figure A.3: Example of an SG-Multi master interface read transaction with a wait cycle

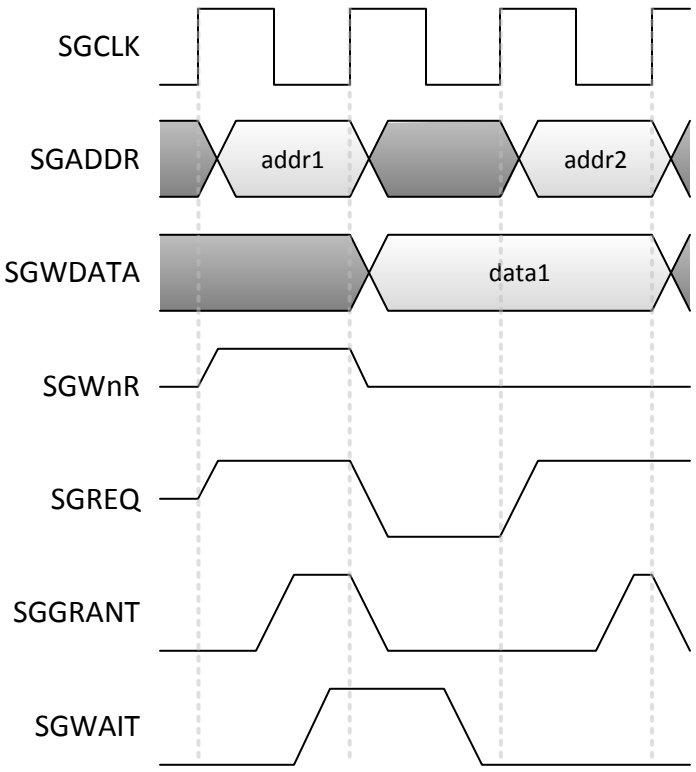



Figure A.4: Example of an SG-Multi master interface write transaction with a wait cycle

### A.2.2 Slave Interface Signalling

Transactions on the slave interface are controlled with SGACTIVATE. A cycle in which SGACTIVATE is asserted is a valid address phase for a transaction directed at that particular slave; if SGACTIVATE is not asserted, the slave must not accept or process a new transaction request. During this address phase cycle, the slave must determine if the transaction type is supported, if it will require any wait cycles, and whether or not it is safe for snooping. It must provide this information—via SGERROR, SGWAIT, and SGSNOOP, respectively—before the end of the address phase cycle. The next rising edge begins the data phase, and the slave must sample all relevant master control signals (such as SGADDR) at that edge.

Slave signalling protocol rules are illustrated in Figure A.5. Three points are circled to draw attention to the proper way for a slave to signal information to a master. The blue circle shows how a slave signals a wait cycle: by asserting SGWAIT at the beginning of the cycle, which in this example leads to the second cycle in the diagram being a wait cycle. The red circle shows how a slave signals an error: by deasserting SGWAIT and asserting SGERROR. A new transaction is supplied in the cycle immediately following the error, and the slave must be ready to accept a new transaction when it signals an error. The green circle shows how a slave permits a transaction to be snooped: by asserting SGSNOOP and SGWAIT, the latter being required because bus snooping only occurs with transactions that take at least two cycles to complete.

Error signalling may only happen immediately following a wait cycle. A slave is not permitted to signal an error once it has already signalled its ability to complete a transaction by beginning a cycle with SGWAIT deasserted. Figure A.6 differentiates between incorrect and correct slave error signalling. The exception to this rule is that a slave may signal an error before the start of a transaction; in Figure A.5, this case would involve the blue circle being replaced by SGWAIT deasserted and SGERROR asserted, a method of signalling a slave might use to reject an unsupported transaction.

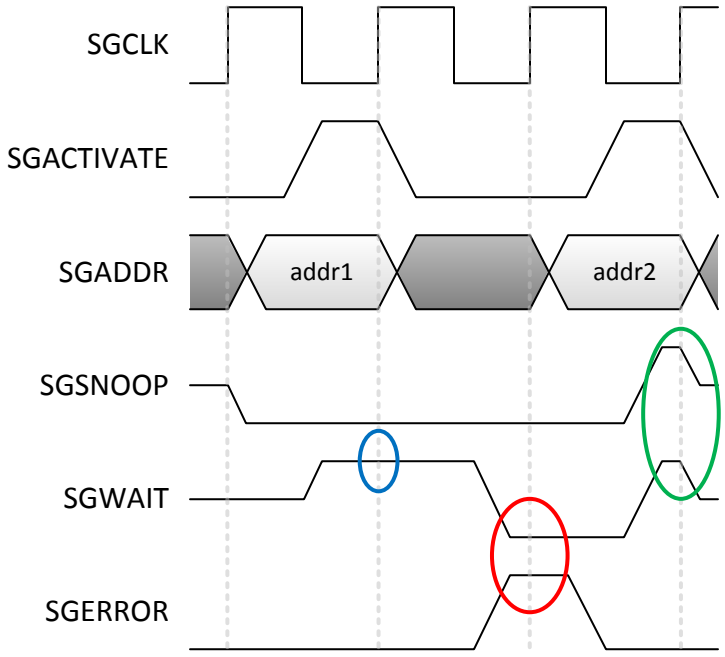
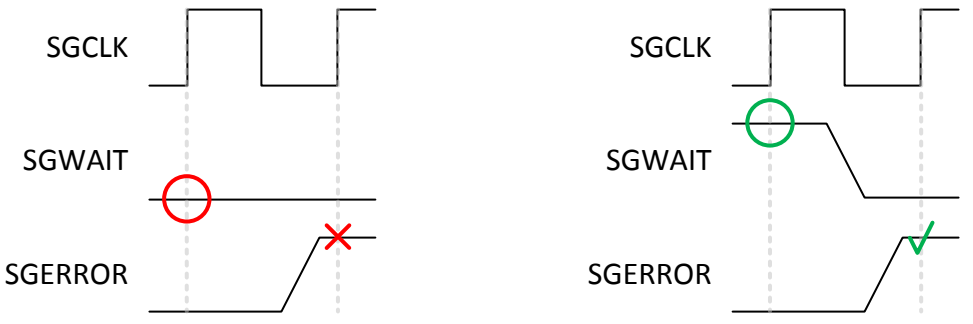



Figure A.5: Illustration of SG-Multi slave response signalling rules




(a) Incorrect error signalling

(b) Correct error signalling

Figure A.6: Differentiation between incorrect and correct SG-Multi slave error signalling

# Appendix B

# SG-Multi Designer XML Specifications

This Appendix contains the information exchange specifications for XML files that describe SG-Multi Designer modules. The intended audience is the designer of a hardware device that is to be provided to users of SG-Multi Designer for use in designs.

Descriptions of the supported types of XML files are presented first, followed by XML file support for variables and macros. Details on the information to be contained in each type of XML file are presented last.

## B.1 Supported XML File Types

SG-Multi Designer supports three primary types of modules: master device modules, slave device modules, and bus adapter modules. It differentiates between these three types based on the name given to the root node of the XML file. Supported root node names are listed in Table B.1. Each XML file may only describe a single module.

## B.2 Variables and Macros

SG-Multi Designer supports variable and macro references in place of constant values in its XML files. This allows hardware module designers to designate place-holders for values

Table B.1: XML file types supported by SG-Multi Designer

| Root Node Name | Device Type |
|---|---|
| `sgmulti-master` | Master device |
| `sgmulti-slave` | Slave device |
| `sgmulti-adapter` | Bus adapter |

that will only be known when it is time to invoke the hardware module's executable file. Variables are references to system-wide configurable parameters or device properties, and macros—also known as "instance parameters"—are references to values that SG-Multi Designer generates immediately before instantiating a device.

References to variables and macros are formatted as a special character followed immediately by the name of the variable or macro. Table B.2 lists the special characters and their meanings.

Table B.2: Special characters for variable or macro references in an XML file

| Character | Reference Type |
|---|---|
| `$` | Global variable |
| `%` | Device property |
| `#` | Instance parameter |

The current version of SG-Multi Designer supports two global variables whose values can be referenced from an XML file. These are named and described in Table B.3.

Table B.3: List and descriptions of global variables supported by SG-Multi Designer

| Name | Description |
|---|---|
| `ADDRESS_WIDTH` | Width, in bits, of the SG-Multi address bus `SGADDR` |
| `DATA_WIDTH` | Width, in bits, of the SG-Multi data busses `SGRDATA` and `SGWDATA` |

Instance parameters are assigned values immediately before SG-Multi invokes a tool to instantiate a device. The current version of SG-Multi Designer defines two instance parameters for all types of device modules. These are listed and described in Table B.4.

Table B.4: List and descriptions of instance parameters supported by SG-Multi Designer

| Name | Description |
|---|---|
| OUTPUT_FILE | File name of the desired output Verilog file |
| MODULE_NAME | Desired name of the Verilog module to be created |

## B.3 XML File Format Details

Every XML file supported by SG-Multi shares some common nodes, but beyond that most are specific to the type of device being represented. The subsections that follow provide details on the nodes that SG-Multi Designer requires to be present in a device module XML file.

### B.3.1 Common Nodes

SG-Multi Designer refers to devices, both internally and through its user interface, by name. Every hardware module must supply a valid name. Devices may also optionally specify a "friendly name" to be shown to the user instead of the internal name. If a device does not specify a friendly name, the internal name is shown instead. A device specifies its name and friendly name as shown in Figure B.1.

```
<name>internalname</name>
<friendlyname>Friendly Name</friendlyname>
```

Figure B.1: XML file format for specifying a device's name and friendly name

### B.3.2 Master and Slave Device Nodes

Master and slave device XML files share many common nodes since they both represent devices to be used in an SG-Multi system. These types of XML files must describe the device's configurable properties, its extra signals, and its executable tool interface.

Properties are defined in a `settings` block within the device's XML file. This block is optional; if a device has no configurable properties, this block does not need to be

present. Each property has its own `property` node, which defines the property's internal name, user-facing description, data type, and optionally minimum and maximum values. Property definitions are formatted as shown in Figure B.2.

```
<settings>
    <property name="p1" description="d1" type="boolean" />
    <property name="p2" description="d1" type="uint2" min="32"
        max="256" />
</settings>
```

Figure B.2: XML file format for specifying a device's configurable properties

Several different data types are supported for configurable properties. These are listed in Table B.5.

Table B.5: List and descriptions of data types for device properties in an XML file

| **Type String** | **Description** |
|---|---|
| `int` | Signed integer |
| `uint` | Unsigned integer |
| `uint2` | Unsigned integer, power of 2 |
| `string` | Character string |
| `verilog` | Character string, valid Verilog identifier |
| `boolean` | Boolean value |
| `file` | File name |

Extra signals are defined in an `extrasignals` block, which contains individual `signal` nodes, one per extra signal. This block is also optional; a device without any extra signals may omit it from its XML file description. Each extra signal must specify an internal name and may optionally specify a width, measured in bits, as well as a disconnected state. The disconnected state specifies the logic that drives an extra signal if a user does not explicitly connect it to a connection point; supported values are shown in Table B.6. If a width is omitted, SG-Multi Designer uses a default value of 1, and if a disconnected state is omitted, it is presumed to be high impedance. Figure B.3 illustrates how extra signals are formatted in an XML file.

Table B.6: Supported extra signal disconnected state specifiers

| Character | Description |
|---|---|
| 1 | Extra signal should be connected to constant logic '1' |
| 0 | Extra signal should be connected to constant logic '0' |
| Z | Extra signal should not be connected to anything (high impedance) |

```
<extrasignals>
    <signal name="sig1" width="16" />
    <signal name="sig2" disconnect="0" />
    <signal name="sig3" width="4" disconnect="1" />
</extrasignals>
```

Figure B.3: XML file format for specifying a device's extra signals

All devices must define the interface to the executable tool used to construct it. This information is captured in a `command` block. At the block level, the `tool` attribute specifies the path of the executable file that should be executed. Within the block, `param` nodes specify an ordered list of command-line arguments that should be appended to the executable file invocation to customize the device instance. Each parameter includes a `val` attribute, which specifies the source of the command-line argument's value, and an optional `prefix` attribute, which specifies a string constant to be inserted immediately before the value. For instance, a prefix of "-N" for a value of 6 would produce "-N6" as the final command-line argument passed to the tool. A command-line argument value specifier will likely be a variable or macro, as this facilitates device customization, though this is not a requirement. Figure B.4 shows the proper formatting of a `command` block in an XML file.

```
<command tool="exefile">
    <param val="value1" prefix="-v1" />
    <param val="value2" />
    <param val="value3" prefix="-v3" />
</command>
```

Figure B.4: XML file format for specifying a device's executable tool interface

Master devices—but not slave devices—may also specify dependence on a bus adapter. This is done through an `adapter` block, which must specify the internal name of the required adapter and provide values (variables or macros are also supported) for any instance parameters that the specific bus adapter requires. Instance parameters vary from adapter to adapter, so the designer of a master device module that depends on a bus adapter should consult the documentation supplied with that adapter module for a list of instance parameters. The proper format for indicating dependence on a bus adapter is shown in Figure B.5.

```
<adapter name="adaptername">
    <param name="param1" value="val1" />
    <param name="param2" value="val2" />
</adapter>
```

Figure B.5: XML file format for specifying a master's dependence on a bus adapter

### B.3.3 Bus Adapter Nodes

Bus adapter modules, like master and slave device modules, must define an interface to an executable tool using a `command` block. They do not, however, support extra signals or configurable properties; a master device that depends on a bus adapter supplies parameter values, which the bus adapter module can access through references to instance parameters.

XML files representing bus adapter modules must define the names of the native signals for the specific architecture being adapted. An `architecture` block serves this purpose, with each native signal being represented as a `nativesignal` node within it. Widths may be optionally specified using constant values or references to variables or macros; any signal without a specified width is presumed to have a width of a single bit. Figure B.6 shows the proper format for this block.

```
<architecture>
    <nativesignal name="sig1" width="11" />
    <nativesignal name="sig2" width="#REF" />
    <nativesignal name="sig3" />
</architecture>
```

Figure B.6: XML file format for specifying a bus adapter's native signal names

## B.4 Example XML Files

To better illustrate the format of XML files, two complete examples are provided: one for an AHB-Lite bus adapter and one for an ARM Cortex-M0 processor. Figure B.7 shows the AHB-Lite XML file, which defines all of the native AHB-Lite signals. The tool requires information from the standard SG-Multi Designer global variables and instance parameters but additionally refers to two additional instance parameters, "AWIDTH" and "DWIDTH," values for which must be supplied by any module that makes use of the AHB-Lite bus adapter. In the context of AHB-Lite, these instance parameters refer to the widths of the AHB-Lite address and data busses respectively, which may be different from the SG-Multi address and data bus sizes.

Figure B.8 shows the ARM Cortex-M0 XML file, with a dummy property added to illustrate how properties are used and accessed. Typically property values would be passed as input to the executable file, behaviour reflected in the third listed parameter for the command-line tool. The Cortex-M0 module also defines device-specific extra signals and provides values for the "AWIDTH" and "DWIDTH" instance parameters on the bus adapter.

```
<sgmulti-adapter>
    <name>ahblite</name>
    <friendlyname>AHB-Lite</friendlyname>
    <architecture>
        <nativesignal name="HCLK" />
        <nativesignal name="HRESETn" />
        <nativesignal name="HADDR" width="#AWIDTH" />
        <nativesignal name="HWRITE" />
        <nativesignal name="HSIZE" width="3" />
        <nativesignal name="HBURST" width="3" />
        <nativesignal name="HPROT" width="4" />
        <nativesignal name="HTRANS" width="2" />
        <nativesignal name="HMASTLOCK" />
        <nativesignal name="HRDATA" width="#DWIDTH" />
        <nativesignal name="HWDATA" width="#DWIDTH" />
        <nativesignal name="HREADY" />
        <nativesignal name="HRESP" />
    </architecture>
    <command tool="build_ahblite">
        <param val="#OUTPUT_FILE" prefix=""/>
        <param val="#MODULE_NAME" prefix=""/>
        <param val="$ADDRESS_WIDTH" prefix=""/>
        <param val="$DATA_WIDTH" prefix=""/>
        <param val="#AWIDTH" prefix=""/>
        <param val="#DWIDTH" prefix=""/>
    </command>
</sgmulti-adapter>
```

Figure B.7: Example XML file for an AHB-Lite bus adapter module

```
<sgmulti-master>
    <name>cortexm0</name>
    <friendlyname>ARM Cortex-M0</friendlyname>
    <adapter name="ahblite">
        <param name="AWIDTH" value="32" />
        <param name="DWIDTH" value="32" />
    </adapter>
    <settings>
        <property name="DUMMY" description="Dummy property"
          type="boolean" />
    </settings>
    <extrasignals>
        <signal name="NMI" disconnect="0" />
        <signal name="IRQ" width="16" disconnect="0" />
        <signal name="TXEV" />
        <signal name="RXEV" disconnect="0" />
        <signal name="LOCKUP" />
        <signal name="SYSRESETREQ" />
        <signal name="SLEEPING" />
    </extrasignals>
    <command tool="build_cortexm0">
        <param val="#OUTPUT_FILE" prefix="-o" />
        <param val="#MODULE_NAME" prefix="-m" />
        <param val="%DUMMY" />
    </command>
</sgmulti-master>
```

Figure B.8: Example XML file for an ARM Cortex-M0 master device module