Report Number: WUCSE-2007-22

2007

Splice: A Standardized Peripheral Logic and Interface Creation Engine

Justin Thiel

Recent advancements in FPGA technology have allowed manufacturers to place general-purpose processors alongside user-configurable logic gates on a single chip. At first glance, these integrated devices would seem to be the ideal deployment platform for hardware-software co-designed systems, but some issues, such as incompatibility across vendors and confusion over which bus interfaces to support, have impeded adoption of these platforms. This thesis describes the design and operation of Splice, a software-based code generation tool intended to address these types of issues by providing a bus-independent structure that allows end-users to easily integrate their customized peripheral logic into embedded systems.... **Read complete abstract on page 2.**

Splice: A Standardized Peripheral Logic and Interface Creation Engine

Justin Thiel

Complete Abstract:

Recent advancements in FPGA technology have allowed manufacturers to place general-purpose processors alongside user-configurable logic gates on a single chip. At first glance, these integrated devices would seem to be the ideal deployment platform for hardware-software co-designed systems, but some issues, such as incompatibility across vendors and confusion over which bus interfaces to support, have impeded adoption of these platforms. This thesis describes the design and operation of Splice, a software-based code generation tool intended to address these types of issues by providing a bus-independent structure that allows end-users to easily integrate their customized peripheral logic into embedded systems. To quantify the benefits of this approach, a comparison of a number of Splice-generated interfaces to functionally identical hand-coded mechanisms is provided in the context of a real-world use case scenario.

2007-22

Splice: A Standardized Peripheral Logic and Interface Creation Engine

Authors: Justin Thiel

Corresponding Author: jthiel@cse.wustl.edu

Abstract: Recent advancements in FPGA technology have allowed manufacturers to place general-purpose processors alongside user-configurable logic gates on a single chip. At first glance, these integrated devices would seem to be the ideal deployment platform for hardware-software co-designed systems, but some issues, such as incompatibility across vendors and confusion over which bus interfaces to support, have impeded adoption of these platforms. This thesis describes the design and operation of Splice, a software-based code generation tool intended to address these types of issues by providing a bus-independent structure that allows end-users to easily integrate their customized peripheral logic into embedded systems. To quantify the benefits of this approach, a comparison of a number of Splice-generated interfaces to functionally identical hand-coded mechanisms is provided in the context of a real-world use case scenario.

Type of Report: Other

WASHINGTON UNIVERSITY

SEVER INSTITUTE

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SPLICE: A STANDARDIZED PERIPHERAL LOGIC AND INTERFACE CREATION ENGINE

by

Justin Thiel B.S. Computer Engineering

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May 2007

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

SPLICE: A STANDARDIZED PERIPHERAL LOGIC AND INTERFACE CREATION ENGINE

by Justin Thiel

ADVISOR: Dr. Ron K. Cytron

May 2007
Saint Louis, Missouri

Recent advancements in FPGA technology have allowed manufacturers to place general-purpose processors alongside user-configurable logic gates on a single chip. At first glance, these integrated devices would seem to be the ideal deployment platform for hardware-software co-designed systems, but some issues, such as incompatibility across vendors and confusion over which bus interfaces to support, have impeded adoption of these platforms. This thesis describes the design and operation of Splice, a software-based code generation tool intended to address these types of issues by providing a bus-independent structure that allows end-users to easily integrate their customized peripheral logic into embedded systems. To quantify the benefits of this approach, a comparison of a number of Splice-generated interfaces to functionally identical hand-coded mechanisms is provided in the context of a real-world use case scenario.

For Dad, 50¢ never went so far....

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank the NSF for their funding through grant CNS-0313203 as well as The Boeing Company and the AFRL for the generous financial support they have provided to this project.

A heartfelt thanks to my fellow students in the Distributed Object Computing (DOC) Group for making the days fly by and reminding me that there is always time in the day to have a little fun. Much of my work over the past two years has been intertwined with various members of the Liquid Architecture Group, and I am grateful to all them for the insight and constructive criticism with which they have provided me. Never before have I met a group of students and professors so dedicated to their research goals and their unwavering committment has been a constant source of inspiration for myself.

Special thanks and recognition are also owed to my advisor, Dr. Ron K. Cytron, who has provided me with the freedom to work on projects that are of personal interest to myself and for the support he has given me throughout this journey.

In a similar vein, I would like to thank my undergraduate advisor Dr. Brad Noble for all of the advice with which he has provided me over the years. Without his hard work and dedication I might have never had the chance to attend graduate school at such a prestigious institution.

Over the years my parents have shown me nothing but patience as I studied for hours or days on end, and I owe them an eternal debt of gratitude for putting me through college and always believing in me. Beyond this, my wife Mindy has sacrificed so much for my dream and her eternal patience is a constant reminder of what a truly wonderful partner she is.

Last, but certainly not least, I would like to thank my beautiful daughter Olivia for always making me smile and for showing me what is truly important in life.

Justin Thiel

Washington University in Saint Louis
May 2007

Chapter 1

Introduction

As hardware development costs have skyrocketed, it has become economically infeasible for the developers of low-volume embedded devices to design and deploy **Application-Specific Integrated Circuit** (ASIC)s or other such specialized components in their systems. To overcome this cost barrier and yet still maintain the ability to include custom circuitry within their designs, many developers have shifted towards the use of system architectures that combine general-purpose processors with **Field-Programmable Gate Array** (FPGA)s. In doing so, they are able to selectively off-load portions of their system to dedicated hardware for a fraction of the cost that would be incurred in fabricating single-purpose circuitry.

Semiconductor manufacturers such as Xilinx and Altera are well aware of this trend and have begun to produce so-called "**System-on-a-Chip** (SoC)" devices that combine CPU and FPGA functionality on a single die. Such devices allow developers to tightly couple their customized logic with an on-board microprocessor and deploy functionality across the software-hardware barrier without having to rely on the use of external interfaces. Systems based on these architectures have the potential to replace an entire set of components with a single chip, thereby greatly simplifying the tasks typically associated with the design and development of embedded devices.

Despite their apparent advantages, the technology behind FPGA-based SoCs is still in its infancy. Issues such as cross-vendor incompatibility of designs and the inaccurate documentation of important procedures are well known amongst developers working with the technology. Furthermore, due to the rapid pace at which both vendor development tools and the chips themselves are changing, there is a great deal of uncertainty about whether or not a design produced today will still be viable five years from now. As a result of these complications and numerous others, the potential

performance and cost benefits offered by these devices tend to be counterbalanced by the steep learning curve required to produce viable hardware from them.

In an attempt to address these issues, we have created a software tool known as Splice that automates many of the common tasks associated with FPGA-based SoC design. The tool operates by forming an abstraction layer between a developer's customized logic and the underlying device it will be deployed on, thereby allowing for the portability of hardware designs from one development platform to another regardless of the bus interfaces or microprocessor architectures that are supported. In doing so, developers are free to focus solely on the implementation of their specialized hardware components without the need to concern themselves with the intricacies of any particular platform. A high-level diagram showing how the tool fits into common hardware/software co-design workflows can be found in Figure 1.1.

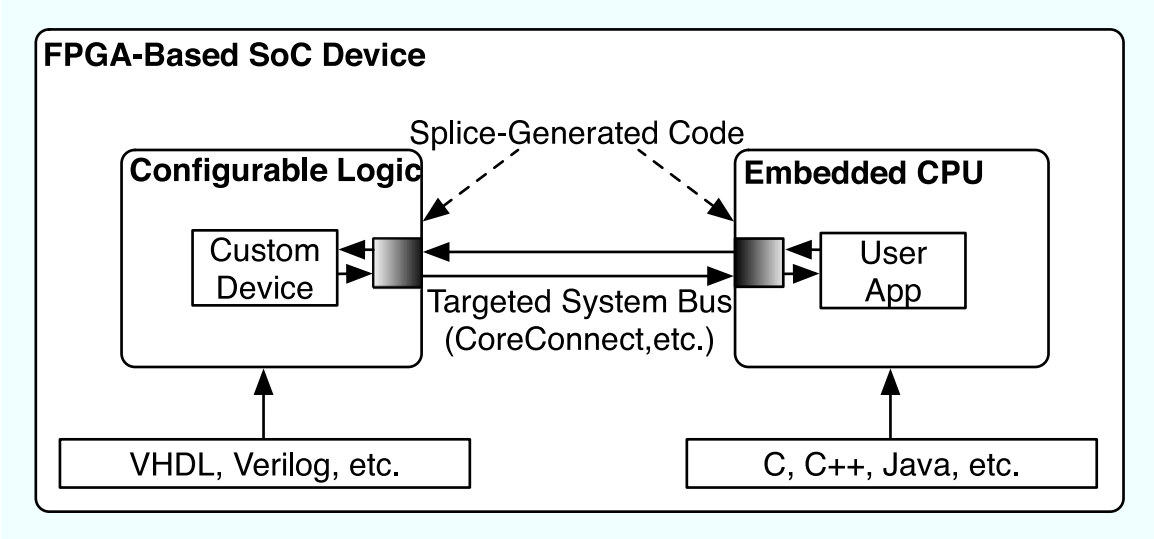

Figure 1.1: High Level Functionality of Splice

In the remainder of this document we discuss the design and implementation of Splice, with particular attention paid to how the tool can be readily integrated into common hardware-software co-design workflows. We start by providing the background details necessary to understand the design decisions made in developing Splice. With the work appropriately framed, we begin to describe the overall architecture of the tool, starting with its specialized syntax and and then moving on to the hardware and software generation capabilities of the program. This is followed

by a brief discussion of the user-expansion facilities provided by the tool and an in-depth description of a real-world use case. We then conclude by proposing a number of possible future enhancements to our design and reiterating the major contributions of the thesis.

Chapter 2

Background

Within this document there are a number of references to specific types of FPGAs, development boards, and system interfaces. Without a proper background in the field of interface design the meaning of these terms would likely be lost on the reader, leading to a difficulty in understanding the high-level significance of this work. In an effort to eliminate these issues and place all readers on an equal footing, this chapter presents an overview of the current FPGA-based SoC marketplace with a particular focus on the system interfaces commonly used within such systems. In conjunction with this information, a review of similar work in this area is also provided as a means of exploring the meaningful contributions of Splice in the realm of automated interface design.

2.1 FPGA-based SoCs

FPGA-based SoCs combine a general-purpose CPU and reconfigurable logic gates onto a single integrated silicon die. In doing so, the aim is to provide developers with a unified device through which customized peripheral logic and common application code can communicate with one another to accomplish well-defined tasks in a more efficient manner than would be possible with other types of computing devices. Often times, these devices are found in embedded systems where they serve as a sort of "poor man's ASIC" by providing the inherent flexibility of customized logic at a fraction of the cost required to build a high-speed dedicated circuit. Descriptions of the devices used in the development of Splice are presented below as a means of introducing the reader to this area of hardware design.

2.1.1 Xilinx Virtex2Pro/4FX

The Xilinx Virtex2Pro and Virtex4FX families of FPGA devices embed one or more a fully-fledged **PowerPC** (PPC) 405 processors directly into the routing fabric of their reconfigurable logic [24]. As a result, these chips are able to offer compatibility with a wide range of existing PPC software such as the Linux operating system and the **GNU C Compiler** (GCC) tool-chain. Furthermore, since the processor itself is implemented as an ASIC block device it is able to run at speeds up to 450 MHz, thereby allowing for general purpose code to be executed at a much higher rate than other devices in its class at the expense of additional power requirements and a larger cost per unit [21].

In terms of system interfaces, the device provides support for the entire IBM CoreConnect family (see Section 2.3.2) through a set of bridges that are implemented in reconfigurable logic and loaded onto the device upon startup. Through these interfaces customized peripheral designs can be interfaced with the on-board PowerPC and accessed in the same manner as any other standard system component.

2.1.2 Gaisler LEON2/3

The Gaisler LEON2/3 is a freely available, open-source **Sun Processor ARChitecture** (SPARC) V8 compatible processor that has gained a fair deal of traction in the SoC market. The device itself implements a *softcore* architecture, meaning that it can be constructed directly from the reconfigurable logic gates of an FPGA. This approach to processor design enables the LEON to offer a level of flexibility not seen in traditional systems. Parameters such as the cache replacement protocol that is used within the CPU can be modified at synthesis time, while support for non-essential components such as the **Memory Management Unit** (MMU) and **Floating Point Unit** (FPU) can be excluded to minimize resource requirements. In providing such flexibility, however, the overall performance of the LEON suffers somewhat in comparison to traditional CPUs, achieving clock speeds of only 125 MHz on even the most advanced Virtex4 devices [5].

In an effort to provide a fully "open" SoC solution, the LEON implements the royalty-free **Advanced Microcontroller Bus Architecture** (AMBA) interface specification (see Section 2.3.1). As is the case with the Xilinx Virtex4FX, support for these interconnects is provided through bridge devices that are loaded onto the underlying FPGA upon system startup. By attaching customized peripherals to these

bridges, user-designed logic can be linked into the processor and accessed directly from applications running on the system.

2.1.3 Xilinx Microblaze

The Xilinx Microblaze, much like the LEON, is a *softcore* processor implementation based on a 32-bit **Reduced Instruction Set Computer** (RISC) architecture. Contrary to the LEON, however, the overall design device is wholly proprietary in nature can only be used in conjunction with Xilinx-based FPGAs. While the device is reconfigurable in terms of the low-level features it supports (FPU, hardware multiplier, etc...) a monetary investment in expensive "logic cores" is often required to unlock high-end features of the CPU. Despite this, the device has become quite popular due to the fact it is bundled with the Xilinx **Embedded Development Kit** (EDK) software. Clock rates range from 100 MHz on a low-cost Spartan3 chip up to 210 MHz on the recently announced Virtex5 devices, providing reasonable performance for a wide variety of embedded systems tasks [18].

In an attempt to maintain a semblance of compatibility with Xilinx's high end Virtex4FX and Virtex2Pro SoC devices, the Microblaze provides support for the IBM CoreConnect **On-chip Peripheral Bus** (OPB) [18]. This, in turn, allows customized logic to be moved between the two processor types with minimal hardware-level changes, thereby extending the shelf-life of a given design. The level of "portability" that is truly achievable through this mechanism, however, is debatable due to fact that software drivers will still need to be rewritten to accommodate the unique instruction sets of the targeted architecture.

2.2 FPGA Development Platforms

When architecting an embedded system, it would be impractical to fabricate a new board design each time the hardware configuration was changed. In the realm of FPGA-based designs, such concerns are typically less of an issue due to the inherent pliability of the devices themselves. Despite this, it is often far more effective in terms of both time and cost to develop systems on a well-defined development platform when working with these devices, due to built-in peripherals and debugging interfaces that they tend to offer. In the course of designing Splice a pair of different development boards were used to verify and benchmark the various operations of the

tool. Descriptions of the capabilities of these boards and their role in the overall development process can be found in subsections below.

2.2.1 Xilinx ML403

The Xilinx ML-403 development board is a highly-integrated embedded systems platform designed around the Virtex4-FX12 FPGA. To facilitate the rapid development of design prototypes, the board contains a number of on-board peripherals including 64 MB of PC-2100 DDR-SDRAM, an RS-232 **Universal Asynchronous Receiever/Transmitter** (UART), and an Ethernet PHY. Through the use of the Xilinx EDK software, controllers for these various devices can be synthesized onto the FPGA and attached to the on-board PPC 405 processor via the IBM CoreConnect bus architecture. In this manner, flexible hardware platforms can be constructed via pre-configured "building blocks" and deployed alongside user logic to create hybrid systems which can run off-the-shelf software and be reconfigured in a fast and efficient manner [22].

In the development of Splice, the capabilities of this board were leveraged to create bus adapters for the **Fabric Co-Processor Bus** (FCB), **Processor Local Bus** (PLB), and OPB system interfaces. Furthermore, this board was used extensively in obtaining the various experimental results presented in Chapter 9 of this document. In both cases, the various debugging capabilities of the Xilinx EDK were utilized to load application software onto the board and retrieve testbench results from the custom hardware. By comparing the outcome of these experiments with the expected results, we were able to verify the functionality of Splice, as well as the assorted "example" devices that were constructed as a means of exercising the capabilities of the tool.

2.2.2 NuHorizons SP3-1500

The NuHorizons SP3-1500, much like the ML-403, is an integrated embedded systems development platform built around a Xilinx-supplied FPGA. Unlike the ML-403, however, which makes use of a complex Virtex4 part with an on-board hard-logic processor, the SP3-1500 employs a comparatively simpler Spartan3 1500 device that is composed almost entirely of reconfigurable logic gates [23]. In conjunction with this FPGA, the board offers a variety of integrated peripherals including 16 MB of PC-133 SDRAM and a number of RS-232 and Ethernet communications ports. As

a result, while the board is not suitable for use in high-end embedded development tasks, it is more than capable of implementing mid-range systems based on *softcore* microprocessor architectures [10].

For the purposes of developing Splice, the Spartan3 device on-board the SP3-1500 was loaded with the LEON2 processor core and an associated set of controller logic designed to facilitate interaction with the various peripheral devices provided by the platform. Using this system as a baseline, the AMBA communications standard implemented by the LEON was studied and a corresponding bus adapter for the **AMBA Peripheral Bus** (APB) interface was constructed. To verify the adapter, the UART and JTAG devices provided by the development platform were utilized to load testbench applications and Splice-based logic onto the system. Through these relatively simplistic debugging mechanisms, we were able to test all aspects of the tool and obtain cycle-accurate performance results for the APB that can be seen in Chapter 9 of this document.

2.3 Common Embedded Bus Interfaces

As embedded systems have grown in popularity and evolved in terms of the capabilities they offer, a variety of peripheral interfaces have been developed in an attempt to suit the ever-changing needs of developers. In doing so, a fragmented development landscape has evolved in which devices that are intended for use with one interface cannot easily be transitioned to another. Further complicating the issue is the fact that SoCs tend to implement only a subset of the interface standards that are available, due to cost constraints or other other architectural concerns. As a result, developers working in this design space often have little or no choice as to what interfaces they will deploy their logic across, and are simply forced to make use of what manufacturers have chosen to support.

In designing Splice, attempts have been made to address this issue and provide a bus-independent layer on top of which portable hardware logic can be deployed. To accomplish this goal, the tool implements bus adaption facilities for a number of well-known embedded systems interfaces. Descriptions of these interfaces are provided below along with additional information in regards to various interconnects for which support will be added in the near future. Using this information as a guide, it should be relatively easy to see that while all interfaces provide essentially the same

functionality, they often do so in manners that are wholly incompatible with one another.

2.3.1 Advanced Microcontroller Bus Architecture

When first introduced in 1995, the AMBA bus architecture was originally intended to provide an open and freely available interface standard for use in conjunction with **Advanced RISC Machine** (ARM) microprocessors. Since then, the standard has gained a great deal of traction in the general embedded marketplace and has seen deployment in a variety of devices ranging from network switches to cellular telephones. Along the way, the bus architecture itself has evolved and expanded its capabilities to provide support for bus transactions that are not tightly coupled to any particular microprocessor design [3].

Because the needs of the embedded development community are highly varied, it would be fairly difficult to create a unified system interface that met the requirements of every designer. To address this issue, the developers of the AMBA specification instead chose to define a number of different bus standards with feature sets suitable for the creation of both simple and complex devices. Each of these standards defines a fairly rigid communications protocol that governs all transactions across a given interface, as well as how optional features such as DMA transfers and interrupts should be handled. Using this information as a guide, developers can tune their own implementations to support only the operations they require, so long as compatibility with the specified transmission protocol is maintained [3].

In an attempt to better explain the capabilities of the various AMBA standards, descriptions of the AHB and APB interfaces are provided in the following subsections. Since Splice focuses primarily upon the creation of interface adapters for FPGA SoC devices, this information is presented with an emphasis on the bus implementation of the LEON2 softcore processor. Due to ambiguities in the AMBA specification, another device implementing these same interfaces could conceivably provide a somewhat different feature set than that which is outlined below.

AMBA High Speed Bus

The **AMBA High-speed Bus** (AHB) is a general purpose pseudo asynchronous interface that is designed to interact directly with a compatible microprocessor in order to facilitate high bandwidth communications at low latency. In the context of

the LEON2, the bus is capable of operating at a native transfer width of either 32 or 64 bits across a 32-bit memory address space. Devices attached to the interface are mapped directly into main memory where they can be accessed using the common load and store operations provided by the processor's **Instruction Set Architecture** (ISA) [3, 5].

When creating an AHB device, it can be configured to function as either a bus master or a bus slave. Bus masters are attached directly to the processor through an arbiter and have independent data and address lines through which to access the system, thereby allowing them to conduct **Direct Memory Access** (DMA) transactions. Bus slaves, on the other hand, are linked to a bus master and are forced to arbitrate on multiple levels before access is granted to the system resulting in high latency operations. In cases where DMA is used, chained transactions of up to 16 cycles are permitted, allowing up to 1024 bytes to be transferred via a single "logical" bus operation. Such capabilities are often used within LEON2 systems to pass network packets directly into memory bypassing the need to involve the CPU in each transaction [3, 5].

At it stands, Splice is currently unable to target the various designs it creates to work with the AHB interface. There are no technical hurdles preventing this from occurring in the future, however, and the omission is more a reflection of time limitations than a lack of desire to support the standard. As the tool grows and is enhanced, it can be expected that support for the AHB and the various features it supports will be among the first items that are added.

AMBA Peripheral Bus

In contrast to the AHB, The APB is a low-speed interface designed to facilitate communication with simplistic devices such as timers and UARTs. The bus itself is not connected directly to the processor, but it rather implemented as a bridge off of the AHB interface. Through this bridge a number of attachment points are provided for APB devices. As a result of this arrangement, peripherals must pass through multiple layers of arbitration to gain control of the shared global interconnect, causing a significant performance penalty in comparison to AHB devices [3].

Among the interfaces currently supported by Splice, the APB is unique in that it implements a strictly synchronous transmission protocol. The practical implication of this design decision is that devices attached to the interface are not allowed to pause the bus during read and write transactions [3]. This, in turn, implies that all

operations must be completed in the same cycle they are enacted, causing possible complications if a peripheral is not in the "ready" state when a transaction begins. Details on how Splice handles this issue can be found in Chapter 4 of this document.

2.3.2 IBM CoreConnect Interface Standard

The IBM CoreConnect architecture is a freely available and open bus standard that was designed to compete with the AMBA interface specification. While both standards offer a near identical feature set, CoreConnect has thus far failed to gain wide acceptance outside of realm of PPC-based systems. This is likely due to the fact that the architecture itself was released 4 years after its competitor and has thus had to compete for support from peripheral manufacturers whose devices had likely already implemented some form of the AMBA protocol. Despite this, the standard remains significant, especially in regards to FPGA devices, where it was chosen by Xilinx for use in conjunction with their embedded PPC-405 and Microblaze SoC designs [6, 18, 21].

As is the case with the AMBA specification, the CoreConnect framework does not attempt to specify a singular interface for use in all embedded systems, but rather defines a collection of different bus standards. As a result, developers are free to pick and choose which interfaces they wish to support depending on the bandwidth and clock rate constraints of their devices. In the case of the Xilinx PPC-405 platform, implementations of the FCB, PLB, and OPB standards are provided, whereas the simpler Microblaze design is capable of communicating with only OPB devices [18, 21]. A brief description of the feature sets of these three interfaces from the perspective of the Xilinx SoC architectures can be found in the subsections below.

Fabric Co-Processor Bus

In terms of functionality, the FCB is a pseudo asynchronous 32-bit bus that is intended to be used as a co-processor interconnect for a single device. The bus is not directly addressable through memory mappings, but can be accessed via a number of FCB-specific opcodes that are included in the instruction sets of compatible processors. In addition to simple single-word load and store operations, the FCB also has native support for double- and quad-word burst transmissions [17, 9]. Due to the fact that the interface is not memory-accessible, however, support for DMA transfers is not provided.

The implementation of the FCB provided by Splice is capable of orchestrating all transaction types supported by the interface. Furthermore, by leveraging the facilities offered by the tool, a developer can attach multiple independent functions to the bus so long as they are integrated within the confines of a single logical peripheral. In doing so, the high-speed and low latency transfers offered by the interface can be utilized by a potentially wider range of hardware designs that would have been possible if the "single device" restrictions imposed by the standard were maintained.

Processor Local Bus

The PLB, unlike the FCB, is a general-purpose interconnect designed to interface directly to a microprocessor and enable concurrent communications to take place between a number of different 32/64-bit devices. Access to the bus is controlled through an arbiter that allows only one device to transmit to the CPU on any given clock cycle. Transmissions to devices are accomplished through memory address mappings or via the various advanced mechanisms such as DMA that are supported by the bus [16]. One peculiarity of the bus is that although burst transactions are supported, explicit instruction-level support is required to activate such transactions from the CPU. In the absence of such instructions these types of transfers can only be enacted on a peripheral-to-peripheral basis, thereby limiting the potential performance gains of this feature.

Through Splice, hardware designs can be constructed that target either the 32 or 64 bit implementations of the PLB. Furthermore, the tool allows developers to include support for DMA transactions of up to 256 bytes at a time by simply setting a parameter within their design specification. As Splice is enhanced in future revisions, support for other features of the interface such as interrupts will likely be added, furthering expanding the range of transactions that can automated by the tool.

On-chip Peripheral Bus

The OPB is a 32-bit interface designed to facilitate communication among low-bandwidth peripherals within embedded systems. Devices attach to the bus through a shared-access arbiter that acts as a bridge to the PLB and allows data to be passed to and from the microprocessor in a pseudo asynchronous fashion. In addition to simple load and store operations, transactions based on the use of DMA and burst modes

can also be executed across via this mechanism, thereby providing feature equality with the more complex PLB albeit at a somewhat reduced level of performance [19].

Within Splice, the level of support that is currently offered for the OPB interface is somewhat limited by the fact that the tool is only capable of generating the logic necessary to handle simple read and write operations. In limiting the functionality provided, the assumption was that developers who required to use of DMA or burst transactions would instead use the more robust PLB in order to avoid the intrinsic latency penalties associated with the OPB. As the tool is enhanced in the future, however, support for these features will likely be added if only for the sake of completeness.

2.4 Related Work

In creating Splice, a great deal of inspiration was drawn from similar work in the areas of software interface generation and platform-independent application design. Furthermore, research that has been performed in the areas of **Hardware Description Language** (HDL) development and hardware interface standardization can also be linked to the tool, albeit in a somewhat less direct manner. The information in the following subsections provides comparisons of Splice to a number of related projects. In doing so, an emphasis is placed on the ways in which Splice differs from currently available solutions, as well as on how the tool is able to provide an highly-integrated system suitable for the design and development of flexible hardware/software interfaces.

2.4.1 CORBA

The Splice approach to bus-interface generation and linkage somewhat resembles the methods employed by distributed object specification systems such as **Common Object Request Broker Architecture** (CORBA) [11]. Like CORBA, Splice generates code based on relatively abstract interface definitions provided by the developer. However, Splice is fundamentally different in a number of ways. For instance, CORBA itself is a software-only system used to link applications seamlessly that are written in different programming languages or residing on different physical machines so that transparent calls can be made between two supported objects. As such, CORBA has no direct support for linking hardware and software together and cannot serve

the needs of the end-users that Splice is targeting. Furthermore, due to the complexities that such a system such CORBA must handle, it is forced to rely on fairly abstract bindings to handle each and every situation in which it might be deployed. In contrast, the structure of each interface file generated by Splice is governed by well defined protocols and microprocessor ISAs, allowing for full optimization of I/O transfers across the software-hardware boundary regardless of the bus interface that is targeted by the end-user.

2.4.2 SWIG

Simply stated, **Simplified Wrapper and Interface Generator** (SWIG) is a software application through which the bindings necessary to link code written in two disparate programming languages can be generated in a transparent manner. In doing so, the tool provides developers with the ability to make calls into custom-compiled libraries from code constructed in an interpreted language (Java, Perl, etc...). By leveraging this functionality, operations that cannot easily be performed in a given language can be off-loaded to such a library and then linked into the program at runtime to complete the specified task. This, in turn, allows developers to enjoy all of the benefits associated with coding in a high-level interpreted language, without sacrificing the power and flexibility offered by "classical" compiled languages such as C and C++ [14]

In comparing SWIG to Splice, it can be said that while both tools effectively provide the same basic functionality, they do so for entirely different segments of the development community. Whereas SWIG attempts to abstract the interfaces through which *software* applications communicate, Splice instead focuses on providing a unified framework though which *hardware* peripherals can be linked to systems in bus-agnostic manner. To accomplish these tasks, the tools must both operate within the strict confines of preexisting APIs and protocols to establish a sense of uniformity within the bindings they produce [14]. As a result, while the fundamental ideas behind SWIG have certainly had an impact on the high-level architecture of Splice, the capabilities offered by the two applications are far more complementary in nature, than they are competitive.

2.4.3 Alternative HDLs

Although not directly related, a case can certainly be made that the functionality provided by Splice is comparable to that offered by hardware-centric design automation languages such as Handel-C [4] or System-C [7]. Much like Splice, these types of systems operate upon C-like source files to create platform-independent hardware-based netlists and/or HDL files. In sharp contrast to Splice, however, both Handel-C and System-C infer not only interface mechanisms, but also calculation logic from the input source code. The downside to this approach to hardware design is that it is often unclear what type of calculation hardware will be generated from a particular set of developer input. Splice, on the other hand, does not attempt to infer calculation logic for the user-defined functions it operates upon, and is thus aimed at a somewhat different group of end-users than any other currently available software/hardware generation systems.

2.4.4 VCI

The **Virtual Component Interface** (VCI) is a "pseudo" bus standard created by the **Virtual Sockets Interface** (VSI) Alliance that attempts to define a universal interface for use in connecting peripheral devices to embedded systems. As is the case with Splice, this task is accomplished through the use of hardware adapters ("wrappers") that link the signaling protocols of a given native interface to those defined by the standard. By placing these adapters between the local bus and user logic, read and write requests to and from hardware components can be translated "on-the-fly", thereby facilitating communication between devices in a fairly bus-independent manner [15, 8].

While the VCI specifies a fairly complete transmission protocol, the standard has suffered from a lack of follow-through on the part of its developers. For instance, there is little in the way of tool support for the specification, meaning that the logic required to interact with the VCI protocol must be constructed from scratch for each and every device that is deployed across the interface. As a result, the need to deal with tedious handshaking code is not eliminated by this standard, but merely consolidated to the point where only a single set of transmission signals need to be considered. This method of hardware development stands in stark contrast to Splice where such tedium is automatically handled by the tool, allowing a focus to be placed

on the functional characteristics of a design rather than the structural requirements of the targeted system.

Chapter 3

Describing Hardware-Software Interfaces via Splice

Mainstream Hardware Description Languages (HDLs) such as Verilog and **VHSIC Hardware Description Language** (VHDL), provide a variety of high-level constructs through which the inner workings of devices can be defined in a largely platform-independent manner. In doing so, these languages have attempted to simplify the set of tasks commonly associated with system design by allowing for the same set of logic blocks to be ported between successive generations of hardware with little or no modification. In reality, however, the portability benefits obtained from these abstractions are often limited due to the inherent inflexibility of the purely structural syntax that interface designers are forced to use when describing the movement of information between communicating components. As a result, while the same peripherals can technically be deployed on any device, a great deal of foresight and careful planning is required to structure them in such a way that they are not physically bound to the system interconnect across which they were initially deployed.

The Splice approach to interface design attempts to address this shortcoming by creating an explicit separation between the implementation of a device and the means by which it is connected to a system. This is accomplished by providing a syntax through which interfaces are defined in a purely *functional* manner and then bound to a specific bus standard via *structural* directives. Since these constructs operate exclusively at the interface level, they have been designed in such a manner that they merely extend the functionality of existing HDLs as opposed to replacing them. In doing so, the tool is able to provide a means by which developers can

generate platform-agnostic interfaces, yet still maintain compatibility with the component libraries and language features that they have come to rely on. The specifics of how this syntax is implemented within Splice are presented in the remainder of this chapter.

3.1 Interface Declarations

To facilitate the creation of hardware-software interconnects in a purely functional manner, Splice provides a construct known as the interface declaration. Individually, these constructs represent little more than the combination of inputs and outputs required to drive a single set of calculation logic. By bundling sets of these declarations together, however, developers can convey the entire collection of communication pathways that are used by a given device, thereby replacing the need for explicit structural descriptions.

Interface declarations are composed in a form similar to that which is used to define function prototypes in the ANSI C programming language. By supporting this common syntax, the tool allows developers to define interfaces in terms of data types that have an intrinsic meaning (chars, floats, etc), as opposed to the somewhat ambiguous "wire-based" constructs that HDLs typically provide. This, in turn, eliminates the need to coordinate transmissions at the raw bit-level and allows for the formation of interfaces in a more functional manner [2].

Beyond these somewhat philosophical reasons, the use of ANSI C as a base language also offers the very practical benefit of providing compatibility with existing function prototypes. By leveraging this capability, developers can convert a software-only application into a hybrid system where selective portions of functionality are off-loaded to accelerated logic. In doing so, it is ensured that the software drivers generated by Splice will have calling conventions matching that of the original software routines, thus allowing the hardware implementation to be "dropped-in" to a design with little or no modifications to the existing code base.

While the advantages of having an interface declaration syntax based on ANSI C are numerous, its usage does present a problem in that it was never intended for use as an HDL. As a result, it lacks the syntactical support required to express hardware-specific constructs such as DMA or bounded array transfers in straightforward manner. Beyond this, full support for software-centric concepts such as pointers is not possible in hardware due to the inherent resource constraints associated with re

programmable devices. In an effort to remedy these issues, as well as various others, Splice implements a number of extensions to the standard ANSI C syntax. By using these extensions within interface declarations, low-level hardware features can be activated in a simple manner while still maintaining source code level compatibility with existing software prototypes.

The following subsections contain an in-depth description of the interface declaration mechanism provided by Splice, with particular attention paid to the various ANSI-C syntax extensions that have been implemented. This information is presented in a combination of formalized **Perl Compatible Regular Expression** (PCRE) declarations and casual text statements that precisely define how each language feature can be activated within the declarations that are passed to the tool [13]. For further information on how these interface declarations are transformed into functional software and hardware descriptions, the reader is advised to refer to Chapters 5 and 6 respectively.

3.1.1 Basic Transfers

As stated above, Splice makes use of a transaction syntax modeled after that used to define function prototypes in ANSI C header files. Essentially, these statements are a combination of three components: an interface name, a return type, and an optional set of inputs. Interface names are unique alphanumeric strings (tags) that are used to refer to declarations within the software drivers and hardware stubs generated by the tool. Return types, on the other hand, specify what (if anything) is passed back from the hardware, and consist of a single C language data type combined with any number of the hardware-specific extensions outlined in this chapter. Inputs to hardware are structured in a similar manner, with the further stipulation that each value (or set of values) returned must be both separated by a comma and associated with a alphanumeric tag that is unique within the constraints of the declaration.

As an example, to define an interface that takes in an unsigned 8-bit value and returns a signed 32-bit value, a statement such as `long get_status()` could be passed to the tool. At run time, the proper software driver and hardware interface required to activate the "get_status" function would then be generated, leaving the end-user with the sole task of filling in the calculation logic required to implement the functionality of the device. A formal declaration of this syntax can be seen below in Figure 3.1.

```
lower_case     := ['a'-'z']
upper_case     := ['A'-'Z']
digit          := ['0'-'9']
symbols        := _
alpha          := lower_case | upper_case
alphanumeric   := alpha | digit
identifier     := alpha (alphanumeric | '_') *
c_type         := int|short|char|bool|double|single|unsigned|void|float
c_decl         := c_type identifier
c_decl_list    := c_decl (',' c_decl)*

c_proto        := c_type '(' c_decl|_list?  ')'  ';'
```

Figure 3.1: Formal Declaration of the Baseline Splice Syntax

3.1.2 Pointer-Based Data Transfers

The most commonly used syntax extension defined by Splice is support for data pointers. In typical C applications, data pointers are extremely powerful and able to pass large amounts of data between functions with little effort on the part of the end-user. Hardware devices, however, cannot typically be passed pointers into memory or unbounded arrays due to the physical resource limitations of FPGAs. Therefore, when pointer-type inputs and outputs are required, the user must define how many items need to be transmitted across the bus interface to synchronize with the hardware.

Explicit Pointers

One method used to define pointer transmission is via an explicit numeric declaration. This type of declaration specifies precisely how many items of a particular data type need to be transferred from main memory into hardware and is supported on both input and output data structures. To make use of explicit transfers, standard pointer declarations need to be extended with the "colon" (or ':') operator. As an example, a declaration of `void some_function(int*:5 x)` would mean that 5 integers should be passed into the hardware implementation of '`some_function`' as input from a unbounded array (x) each time its corresponding driver is called. This syntax is expressed in a more formal fashion in Figure 3.2.

```
pointer        := '*'
expl_ref       := ':'  digit+
expl_ptr_decl  := c_type pointer expl_ref?  identifier
```

Figure 3.2: Formal Declaration of Explicit Pointer Syntax

Implicit Pointers

As an alternative to explicit pointer declarations, the end-user also has the option of using an implicit index value to define how many items should be passed into or out of a hardware function. Implicit array transfers reference the values of other inputs listed in the prototype to determine how many items should be transferred, allowing end-users to create "dynamic" hardware functions that operate on variable-length parameters. Much like explicit transfers, the "colon" operator extension is used to define an implicit declaration. As an example, a declaration of `void some_function(char x, int*:x y)` would imply that `some_function` takes in a total of 'x' integers transferred from the 'y' integer pointer array. A formal expression of this syntax can be seen in Figure 3.3.

```
impl_ref       := ':'  identifier
impl_ptr_decl  := c_type pointer impl_ref identifier
```

Figure 3.3: Formal Declaration of Implicit Pointer Syntax

3.1.3 "Packed" Data Transfers

In addition to providing support for explicit and implicit pointers, Splice also defines an additional language extension to assist in the packing of data values. In simple terms, data packing is a mechanism that allows a designer to transfer multiple data entries to or from a compatible target bus in a single transmission cycle. For instance, if provided with a 32-bit wide interface, a total of four 8-bit characters could be transferred across the bus in a single cycle if data packing is used, resulting in a 75% reduction in transmission time versus transferring one character at a time.

To enable the use of data packing within function declarations, the "plus" (or '+') extension was added to the Splice syntax. Use of this extension must be combined with either an explicit or implicit pointer declaration to be recognized by the tool. As an example, the statement "`void some_function(char* x:8+)`" would imply that

the user wants to transmit 8 characters across the bus in packed mode. Assuming that the targeted bus is 32 bits wide, this construct would allow 4 characters (at 8 bits/character) to packed into a single transmission cycle. This would enable all 8 values of 'x' to be transmitted in 2 cycles as opposed to 8, speeding up operation and freeing the bus sooner for additional transactions. The hardware designer would then be responsible for extracting each value from input data stream as they were sent into the peripheral. A formal declaration of this syntax can be found in Figure 3.4.

```
packed       := '+'
packed_decl  := c_type pointer (impl_ref | expl_ref) packed identifier
```

Figure 3.4: Formal Declaration of Packed Data Transfer Syntax

3.1.4 "Split" Data Transfers

While never explicitly defined within a prototype declaration, cases do arise when multiple transmission cycles will be required to send or receive a value across the system bus. As an example, when attempting to transmit a single 64-bit double across a 32-bit interface, two transmission cycles will be required to transmit the value. Similarly, an array of 16 doubles would take 32 transmission cycles to send. Furthermore, these types of transmissions can be further complicated when complex structs need to be passed across the bus.

To address these situations, Splice handles them transparently within the software drivers in a fashion similar to the method used to handle "packed" data transmissions. The end-user is then responsible for reassembling the split data transfers into a contiguous structure within the hardware. As such, developers need not modify the structure of their data to accommodate bus-width restraints. This, in turn, allows Splice-generated drivers to maintain compatibility with any pre-existing software functions that access the "split" data.

3.1.5 Direct Memory Access (DMA) Transfers

Along with extensions to handle various array operations, Splice also provides syntax enhancements that allow end-users to transfer input and output data parameters via DMA. Enabling this feature allows the designer to transfer information to and from a hardware peripheral without direct CPU-to-memory-to-bus interaction, thus freeing

up the processor to perform other tasks while data transmission takes place. This feature is especially helpful in cases where a function requires a large block of input to operate, and thus a large number of CPU cycles could be saved by automating said requests. Support for this feature is limited to use with bus interface types that have built-in physical support for DMA operations. In other words, Splice is not capable of providing DMA support to a bus that does not already have such capabilities.

Assuming that a DMA-supporting target bus is selected, the feature can be activated by including the "caret" (or '^') syntax extension within a user-defined function prototype. As is the case with packing, use of this extension is limited to those I/O parameters that make use of either explicit or implicit transmission. As an example, the statement "`void some_function(int*:8^ x)`" would imply that the user wants to create a hardware block that takes in 8 integers via DMA. This would enable `some_function` to be activated without the need for the CPU to physically pass data into the user-logic block. The formal syntax required to express DMA transactions is shown in Figure 3.5.

```
dma       := '^'
dma_decl  := c_type pointer (impl_ref | expl_ref) dma identifier
```

Figure 3.5: Formal Declaration of DMA Transfer Syntax

3.1.6 Creating Multiple Instances of a Function

Splice also provides syntax extensions to automatically generate multiple copies of the same hardware function from single prototype declaration. This feature is useful for a multi-threaded software application in which the developer wishes to have a copy of a specific peripheral hardware function available for each software thread to use. By having multiple copies of the same hardware function, developers can avoid the performance penalty of having to arbitrate access to a single resource and, as a result, drastically improve performance.

Similar to pointer declarations, this feature is activated by the "colon" operator, but the colon is included at the end of a declaration instead of being inserted on a per-parameter basis. As an example, a declaration of `void some_function(int x, int y):4` would generate four independent copies of `some_function` that could all execute calculations in parallel. Information on how individual instances of a hardware block can be selected and manipulated within user-created applications can be

found in Chapter 6, while a formal description of the syntax required to utilize this feature can be seen below in Figure 3.6.

```
multi_ref    := ':'  digit+
multi_proto  := c_proto multi_ref
```

Figure 3.6: Formal Declaration of Multiple Instance Syntax

3.1.7 Creating Non-Blocking Function Calls

By default, any prototypes defined in Splice are assumed to be synchronous operations, regardless of whether or not they actually return a value to the processor. That is, if a prototype specifies a 'void' return type then the generated driver for such a function will return control to the user application only after all hardware-related operations have completed. In doing so, the tool is able to guarantee that a given function has finished processing once the driver call has terminated and program execution is resumed.

This system works well for heavy-handed calculations that rely on the ordered execution of bus operations, but is inefficient for use in making simple "set-it-and-forget-it" calls into hardware. In an effort to enable support for these types of transactions within Splice, an extension for *non-blocking* hardware function calls was added to the tool. Support for this feature is enabled by setting the return type for a prototype that returns no value to 'nowait' instead of 'void'. As an example, the statement "`nowait some_function(int x, int y)`" would imply that the end-user wants to pass two integers (x and y) into a user-logic block and then immediately return control of the system back to the processor. A formal description of the non-blocking function call extension can be found in Figure 3.7.

```
block_type    := "nowait"
nowait_proto  := block_type '(' c_decl_list?  ')'  ';'
```

Figure 3.7: Formal Declaration of Non-Blocking Function Call Syntax

3.1.8 Combining Syntax Extensions

Although the syntax extensions provided by Splice are capable of expressing complex transactions in an abstract manner, there are many instances when the functionality

they expose is far too coarse-grained to be applicable in a given design space. As an example, when using DMA to orchestrate a transaction, it would easier to create generalized hardware logic if the amount of data sent via the transfer were based on an implicit value as opposed to a explicit numerical bound. Similarly, a declaration that makes use of both DMA and data packing could offer performance benefits not realizable through the application of any one language extension.

In an effort to address these issues and provide a highly flexible syntax, the tool allows for the use of multiple language extensions in a single I/O transaction. When creating such declarations, developers can simply chain together the various extensions that they wish to make use of and then terminate the "list" with either an explicit or implicit reference as a means of indicating the number of values that should be transmitted. As an example, the statement "`void some_function(char*:16^+ x)`" would indicate that sixteen characters should be transferred from the `x` array in a packed format over the DMA channels supplied by the targeted bus. The method via which other advanced transactions can be orchestrated in this manner is outlined in the formal syntax declaration (Figure 3.8) seen below.

```
multiple          := ':'  (digit+ | identifier)
extensions        := pointer packed?  dma?  multiple?
splice_type       := c_type | block_type
splice_decl       := type extensions?  identifier
splice_decl_list  := splice_decl (',' splice_decl)*

splice_proto      := splice_type extensions '(' splice_decl_list ')' multiple ';'
```

Figure 3.8: Formal Declaration of the Complete Splice Syntax

3.2 Target Specification

The language that has been described thus far defines a flexible framework through which interface declarations can be constructed in a purely functional manner. It does not, however, provide the mechanisms necessary to link a set of declarations together into a cohesive peripheral device. To remedy this issue, Splice offers an additional set of constructs known as the target specification. Through careful application of this syntax, developers can define the structure of the system bus across which they wish to deploy their designs, thereby providing a physical binding for the interface declarations that are passed to the tool.

In terms of structure, all target specification statements are preceded by a "%" or "percent" character which is then followed by a keyword and one or more modifiers. In cases where multiple modifiers are used, a comma (",") is used to separate each item. Depending on the types of interface declarations that are passed to the tool, one or more modifiers may need to be included to generate the proper hardware. In cases that an irregularity is detected, the tool will alert the end user of the error and allow them to address the problem.

The remainder of this section contains descriptions of the various target directives that Splice supports, with particular attention paid to the effects they have upon the various types of interface declarations that end-users might define. As was the case in Section 3.1, this information is provided in both a textual and structural format in an effort to provide maximum detail without introducing ambiguity in regards to how the constructs are defined.

3.2.1 Bus Structure Directives

Bus structure directives provide an essential bridge between the abstract interface declarations that Splice operates upon and the physical interconnect across which communication among user-defined functions will occur. In effect, these parameters bind the hardware logic and software drivers that the tool generates to a specific development platform or processor architecture. This, in turn, reduces the number of steps that a developer must go through to deploy a Splice-based peripheral within their system. The bus structure directives that are currently supported by Splice are outlined in the following sub sections.

Bus Type

The `bus_type` directive allows a developer to define the bus mechanism that Splice should target when configuring the layout of the various hardware and software interface files that are generated at run time. This directive must be defined in order to properly generate interface files from a set of input declarations, and is thus required for proper operation of the tool. When this is not done, the tool will generate an error message and refuse to proceed further until the issue has been addressed.

In terms of syntax, the directive takes in a single textual parameter that specifies the name of the interface to be targeted. As an example, the command `%bus_type some_interface` would indicate that the "some_interface" bus should be

targeted by the tool. Passing this directive into Splice along with a set of interface declarations would cause a VHDL-based interconnect for "some_interface" to be generated, which could then be attached to the other hardware and software files generated by the tool to form a complete peripheral implementation. A formal description of this syntax can be found below in Figure 3.9.

```
bus_type  := "%bus_type" identifier
```

Figure 3.9: Formal Declaration of the Bus Type Directive

By default, Splice is able to target the IBM CoreConnect PLB, FCB, and OPB interfaces, as well as the AMBA APB [3, 6]. This, in turn, allows developers to target a number of common FPGA-based SoCs such as the Xilinx Virtex4 FX and Virtex2 Pro devices, as well as any device capable of supporting the Xilinx Microblaze or Gaisler Research LEON2/3 soft-core architectures [21, 18, 5]. As additional interfaces are developed in the future, support for them can be added to the tool via development API described in Chapter 7 of this document.

Bus Width

When used in conjunction with the `bus_type` directive, the `bus_width` parameter allows an end user to define the size of the data pathway across which information will be transmitted to and from the various hardware logic stubs and software drivers that Splice generates. Due to the fact that the bus width has a significant effect upon the infrastructure required to support it, Splice requires that this parameter be explicitly defined before it will generate hardware logic and software drivers for any set of interface declarations. When this is not done, the tool will generate an error message and refuse to proceed further until the issue has been addressed.

In terms of syntax, this directive takes in a single integer parameter representing the number of bits wide the data interface specified by the `bus_type` parameter should operate upon each clock cycle. As an example, the statement "`%bus_width 32`" would imply that the targeted bus should have a native width of 32-bits and thus could transmit data values of that size or lower in a single clock cycle. A formal declaration of the syntax for this directive can be seen below in Figure 3.10.

The actual values that this directive can take will often be limited by the structure of the bus that is targeted. For instance, the FCB only supports 32-bit transfers and thus requires that a value of "32" be set for this parameter . Other interfaces,

```
bus_width   := "%bus_width" digit+
```

Figure 3.10: Formal Declaration of the Bus Width Directive

such as the PLB, however, can be configured for both 32 and 64 bit transfers, and thus would be capable of providing an interconnect of either width. In all cases, the amount of FPGA resources required to implement a given bus will scale in size with the physical width of the interconnect, and thus care should be taken to match the features of the interface with the needs of the hardware that will deployed across it.

Base Address

The `base_address` directive is used to define the initial memory address to which the hardware logic generated by Splice will be mapped when the device is deployed in hardware. Through the use of this directive, the tool is able to organize the structure of the various interface declarations that are passed to it and configure them such that they can all be attached to the targeted shared bus interface. The presence of this directive is required if the selected bus is accessible only via memory-mapping, and is ignored in cases where it is defined but not required.

In terms of syntax, this directive takes in a single hexadecimal parameter corresponding to the base address where the logic generated by Splice will be mapped in memory at the time of deployment. As an example, the statement "`%base_address 0x80000000`" would indicate that a peripheral's address space starts at 0x80000000, and continues upward (increasing) from that location. A formal declaration of this syntax can be seen below in Figure 3.11.

```
hex_digit     := digit | ['A'-'F'] | ['a'-'f']
base_address  := "%base_address" "0x" hex_digit+
```

Figure 3.11: Formal Declaration of the Base Address Directive

The types of hexadecimal values that are accepted by the tool for this directive will vary depending on the type of bus interface that is targeted. For instance, the PLB operates on 32 bit addresses, and Splice will therefore require that a base address of a similar width be passed in order to generate a proper interface. Furthermore, it is up the developer to ensure that the base address assigned to a peripheral is properly allocated within the system that the device will operate. In cases where this is not

done, the hardware will likely not function as expected, and the commands passed to it may inadvertently be redirected to another component on the bus resulting in the occurrence of undefined behavior.

3.2.2 Bus Feature Directives

Bus feature directives allow developers to selectively enable or disable support for the various options that are provided by a targeted interface. By manipulating these parameters, the resource requirements and control paths associated with an interconnect can be fine-tuned such that only the features required to implement a given set of interface declarations are present in the hardware logic generated by the tool. This, in turn, can lead to better hardware performance due to decreased latencies and the possible elimination of critical paths within a design.

For the most part, the use of these directives is entirely optional, and they are supported on a bus-by-bus basis. In cases where bus feature directives are undefined, Splice assumes that support for the associated options should not be provided and bypasses the task of generating the hardware necessary to do so. It should be noted, however, that developers can specify interface declarations that take advantage of features controlled by these parameters, regardless of whether the directives themselves are supported or enabled. In cases where this is done, the tool will generate an error message and refuse to proceed further until the issue has been resolved. Descriptions of the various bus feature directives that Splice supports are provided below.

Burst Transactions

The `burst_support` directive is used to enable or disable support for "bundled" transactions in which multiple values are transferred across a bus in a repeated fashion without the need for per-value handshaking on the opposite end of the interface. In peripherals where array-based transfers are common, enabling this feature can greatly reduce the number of clock cycles needed to complete a set of bus operations by lessening the amount of control overhead required to initiate said transactions. This, in turn, can lead to higher overall performance in cases where such transactions are executed frequently and support for the feature is provided by the targeted interface.

In terms of syntax, this directive takes in a boolean value ("true" or "false") denoting whether or not support for burst transactions should be enabled. As an

example, the statement "`%burst_support true`" would indicate that burst transactions should be allowed to occur across the targeted interface. A formal declaration of this syntax can be seen below in Figure 3.12.

```
boolean         := "true" | "false"
burst_support   := "%burst_support" boolean
```

Figure 3.12: Formal Declaration of the Burst Transaction Directive

When support for this feature is enabled, all array-based transactions defined in the interface declarations passed to Splice are replaced by comparable burst operations in the software drivers generated by the tool. The types of transactions that are used within these drivers will vary from bus to bus, depending on the capabilities of the chosen interface. For instance, a bus supporting 512 bit bursts would potentially be able to complete a set of transactions quicker than one with a burst length of only 256 bits. By taking advantage of the bursting features that each bus has to offer the tool can tune the code it generates such that a minimum amount of overhead is incurred in each array-based transaction.

DMA Transactions

The `dma_support` directive is used to enable or disable support for the execution of DMA-based transfers across a bus interface. In designs where high-volume array transactions to a peripheral are often followed by processor intensive operations, making use of this feature can allow said memory operations to proceed in parallel with other system tasks. Assuming that care is taken to balance the various operations within a system, these types of transfers can effectively "hide" the latencies associated with heavy-handed bus transactions, resulting in a higher level of overall system performance.

In terms of syntax, this directive takes in a boolean value ("true" or "false") denoting whether or not support for DMA transactions should be enabled. As an example, the statement "`%dma_support false`" would indicate that DMA transactions should not be allowed to occur across the targeted interface. A formal declaration of this syntax can be seen below in Figure 3.13.

For Splice to make use of DMA to complete an array-based transaction, the feature must be explicitly instantiated within the interface declarations (via the "caret" extension) passed to the tool. This is required primarily because DMA transfers rely

```
boolean      := "true" | "false"
dma_support  := "%dma_support" boolean
```

Figure 3.13: Formal Declaration of the DMA Support Directive

on the use of different software mechanisms to operate and thus require the generation of compatible drivers when enabled. In cases where declarations request the use of DMA and the `dma_support` directive is not enabled or supported, Splice will generate an error and refuse to proceed until the issue has been addressed.

Data Packing

The `packing_support` directive is used to enable or disable support for the use of "compacted" bus transfers in which multiple data values from the same array can be moved across a bus interface through a single command. When this directive is activated, data packing is enabled at a global level, but will only be implemented in cases where the size of the array entries defined in an interface declaration is "small" (i.e. 8-bit characters) in comparison to the width of the targeted bus. In designs where these types of operations are common, the use of data packing can greatly reduce the number of clock cycles required to complete said transmissions.

In terms of syntax, this directive takes in a boolean value ("true" or "false") denoting whether or not support for packed data transfers should be enabled. As an example, the statement "`%packing_support true`" would indicate that data packing should automatically be used for all transactions in which it possible to do so. A formal declaration of this syntax can be seen below in Figure 3.14.

```
boolean       := "true" | "false"
data_packing  := "%packing_support" boolean
```

Figure 3.14: Formal Declaration of the Packing Support Directive

Unlike the burst and DMA features discussed above, data packing can be enabled for any bus that a developer chooses to target. As a result, a collection of Splice-based calculation logic based on the use of data packing can be ported from one interface to another with minimal effort. In some cases, however, the use of data packing at a global level might not make sense due to resource limitations of devices or buffering concerns resulting from the need to process and store multiple data values in a single clock cycle. To remedy this, Splice allows developers to enable data packing

on a per-transfer basis through use the "plus" (+) interface declaration expansion, as discussed in Subsection 3.1.3. When this extension is used, the `packing_support` directive should be set to "false" in order to prevent the tool from inferring unwanted data packing operations from the declarations that are passed to it.

3.2.3 Syntactical Directives

Syntactical directives provide a mechanism by which the internal structure of the various files created by Splice can be modified without affecting the outward functionality of the interface implementations that are produced by the tool. In doing so, these directives essentially act as an abstraction layer through which the code generation facilities of the application can operate in syntax-agnostic manner. By leveraging this functionality, developers can create hardware logic and software drivers that are more amenable to the requirements of their pre-existing workflows, thereby reducing the amount of time and effort required to deploy a fully-functioning system.

Device Name

When creating hardware/software interfaces via Splice it is helpful to be able to easily identify and sort the files produced by the tool so that they do not get misplaced or inadvertently overwritten as changes are made to the design of the desired system. To assist in this task, the tool provides support for a `device_name` directive that can be used to mark the various hardware logic and software drivers that are produced from a given set of interface declarations. By making use of this parameter, developers can delineate the different "versions" of the interface files they have created, thereby avoiding possible issues associated with mixing different revisions of hardware and software.

In terms of syntax, this directive takes in a single alphanumeric string denoting the identifier that should be associated with a given design. As an example, the statement "`%device_name timer_v1`" would indicate that all files generated by Splice should be associated with the "timer_v1" device. The tool will then use this identifier to create a subdirectory of the same name and place all files related to the project in this location. All hardware logic and software drivers sources associated with the design will also have the same identifier inserted into them (via comments and other non-intrusive mechanisms) as means of differentiating them from other files produced

by the tool. A formal declaration of the syntax for this directive can be seen below in Figure 3.15.

```
device_name  := "%device_name" identifier
```

Figure 3.15: Formal Declaration of the Device Name Directive

Due to the fact that Splice uses the value associated with this directive to organize the output it produces, it is required that each set of input passed to the tool be accompanied by a `device_name` declaration. If this directive is omitted, the tool will generate an error and refuse to continue until it has been defined. In cases where the current working directory has a subdirectory of the same name as that associated with the directive, the tool will generate a warning and wait for the user to confirm their selection before beginning the process of interface generation.

Target HDL

The `target_hdl` directive is used to select the language syntax in which the various hardware logic files that are generated by Splice should be expressed. Currently, the purpose of this directive is purely ornamental, as the tool is only capable of generating files in the VHDL format. In the future, however, there are plans to add support for additional mainstream HDLs such as Verilog to the tool (see Section 10.2). As a result, while the use of this directive is not technically required at this point in time, it is strongly recommended that it be included in all interface specifications to ensure compatibility with future revisions of the application.

In terms of syntax, this directive takes in a single alphanumeric identifier corresponding to the name of the language syntax that should be used to structure the hardware logic files generated by the tool. As an example, the statement `%target_hdl vhdl` would imply that all hardware files produced for a given device should make be formatted in the VHDL language. A formal declaration of this syntax can be seen below in Figure 3.16.

```
valid_hdls   := "vhdl"
target_hdl   := "%target_hdl" valid_hdls
```

Figure 3.16: Formal Declaration of the Target HDL Directive

Custom Data Types

By including support for all native types defined by the language, Splice is able to maintain a high degree of compatibility with existing software-based function prototypes. In many cases, however, software developers make use of customized types in their applications when manipulating specialized segments of data. This presents an issue in that Splice has no notion of these custom types and thus cannot directly generate interfaces from function prototypes that contain them. To address this issue, support for a type-defining (typedef) mechanism was added to the tool.

Type definitions in Splice are structured in a manner similar to those used within ANSI C, but with the additional requirement that the size of the defined type be explicitly stated in the definition. This information is needed because the tool implements only a rudimentary parser and thus cannot directly infer the size of the type solely from its definition. As an example, to define a 64-bit unsigned integer type in ANSI C one would use statement similar to `typedef uint64 unsigned long long`. In Splice, however, this statement would be replaced by one with a somewhat different syntax: `%user_type uint64, unsigned long long, 64`, where "64" is the number of bits the data type contains. A formal syntax declaration for this directive can be found below in Figure 3.17.

```
user_type  := "%user_type" identifier ',' identifier+ ',' digit+
```

Figure 3.17: Formal Declaration of the Custom Data Typing Directive

Once defined, a custom type can be used within any interface declaration the end-user constructs. Furthermore, there is no limit to the number of type definition statements that can be passed to the tool. At run time, Splice simply collects all the definitions and generates proper software drivers and logic stubs for those interface declarations that make use of the custom types, along with a compatible hardware logic stub. In doing so, the calling conventions of the vast majority of pre-existing prototypes can be maintained, thus reducing the amount of work the developer must perform.

3.3 Current Limitations of the Splice Syntax

Although Splice is capable of generating a wide array of interfaces, the tool does suffer from a number of shortcomings in its present form. Often times, the effects of

these faults are somewhat minor and easily avoidable through the use of alternative language features or trivial syntax adjustments. In other cases, however, a simple workaround might not exist and more involved changes to a design may be required to implement the required interconnect. The paragraphs below serve as a description of the known limitations of the software and the workarounds that exist. This information should be consulted before attempting to generate interfaces via the tool.

Perhaps the most important concern to keep in mind when defining interfaces in Splice is that pointer declarations are handled by the tool in a slightly different manner than in traditional ANSI C compilers. For instance, when a software developer defines a pointer-based transfer within a function prototype, they are typically able to dictate whether the associated variable should be passed "by reference" or "by value". Through this functionality, subroutines that return more than a single type of value can be architected, thereby allowing for more flexibility in program design. In Splice, however, there is currently no support for this differentiation, and instead all pointer transfers are treated as if they are pass-by-value. As a result, interface declarations are able to return only a single value (or set of values) from hardware as specified by their return type.

Similarly, when a software developer defines a function that operates on a bounded pointer whose element count is based upon the value of another variable, there are typically no restrictions on how the parameters must be listed within the associated prototype. Splice, on the other hand, transfers parameters to hardware in the precise order they are listed within the interface declarations passed to the tool. Because of this, any implicit transfers included in an interface declaration are allowed to reference only those inputs that would be transmitted prior to themselves. As an example, the interface declaration `void some_function(int x, int*:x y)` is valid because the indexed value (`int x`) would be copied to hardware before the implicit transaction (`int*:x y`) that references it, whereas a declaration of `void some_function(int*:x y, int x)` would be rejected by the tool because it violates this restriction.

Beyond these pointer-related hazards, developers need to be careful of the possible portability issues associated with the use of certain syntax extensions or bus directives in their interface declarations. For example, when data packing is turned on (either globally or on a per-declaration basis), the number of a given data type that can be transmitted across a bus in each cycle will be directly proportional to the width of the interface that is targeted. Because of this, it can be somewhat

difficult to adapt a set of interface code optimized for packing across a specific bus for use across an interconnect with a different native width. To address this issue, conditional statements can be added to the generated hardware code to infer the proper data transfer routine based on the width of the interface across which a given device is deployed.

Additional portability issues can occur as a result of the fact that different bus architecture often provide somewhat unique feature sets, thereby making it difficult to port sophisticated peripheral logic between platforms. To address this issue, Splice expresses all complex transactions (DMA, burst, etc...) in the form of related groups of common read and write operations. In doing so, the tool is able to produce "generic" interface code that is effectively isolated from the physical implementation of any advanced features that are supported by a given bus. By leveraging this functionality, a set of interface declarations can be ported between two interconnects by simply swapping out any unsupported transaction types with equivalent operations that the targeted bus is capable of executing.

Chapter 4

The Splice Interface Standard

In order for a peripheral to interact with a specific system bus, both entities must agree on a standardized signaling protocol. Since most existing system interfaces make use of static transmission mechanisms, the end-user typically has to adapt their peripheral to work within the confines of the chosen bus. This results in the peripheral becoming dependent upon the targeted system bus to function, and essentially "locks" a design to a specific architecture. In an effort to eliminate such dependencies the **Splice Interface Standard** (SIS) was created.

Simply stated, the SIS functions as an intermediate interface between an external system bus and the user-logic files generated by Splice. All user-level logic stubs generated by the tool implement the protocols of the SIS, and operate under the assumption that any bus communications will be conducted within the confines of the abstract interface. In doing so, the SIS effectively provides a barrier between custom logic and the outside world, thereby acting as the "lynch-pin" of the entire Splice system. The remainder of this chapter presents the reasoning behind the creation of the standard, alongside descriptions of the structure and protocols that interconnects must support in order to interact with Splice-based peripherals.

4.1 The Need for a Standardized Protocol

As the demand for embedded devices continue to grow, it is not unreasonable to assume that the bus interfaces of today will eventually be replaced by more advanced implementations which can support the performance requirements of the future. In doing so, backwards compatibility with the devices of the past will be likely be sacrificed in the interest of providing cleaner and more efficient development platforms.

This, in turn, will leave developers in a bind, forcing them to either re-code their designs from scratch or adapt them each time a new interface emerges. In the worst case, the need to continually re-invent the wheel could eventually begin to dominate all other tasks leading to fewer innovations in future hardware revisions.

Splice attempts to address these issues by abstracting the process of interface design, and, as such, has been designed to operate outside the bounds of any particular bus specification. Furthermore, the syntax provided by the application does not attempt to pander to structural concerns and instead presents interface building blocks that work in a fairly functional manner. In turn, these constructs allow developers to design their interfaces at a higher level than is possible in typical mainstream HDLs. When converting a set of these generalized declarations into usable hardware, however, the complex issue of how one should go about connecting the various specified components to the targeted interface comes to the forefront.

At first glance, a method of direct conversion in which Splice would connect the user logic stubs it generates directly to the targeted interfaces might seem like an ideal solution. In reality, however, such a structure would require that an entirely separate version of the tool be created for each supported interface. Furthermore, this method of connection would not actually address the issue of hardware portability, due to the fact that direct handshaking would be needed to handle each bus transaction within a given device. To circumvent these problems, as well as various others, a method of indirect conversion based upon the use of the SIS was developed. A graphical description of how this mechanism functions can be seen below in Figure 4.1.

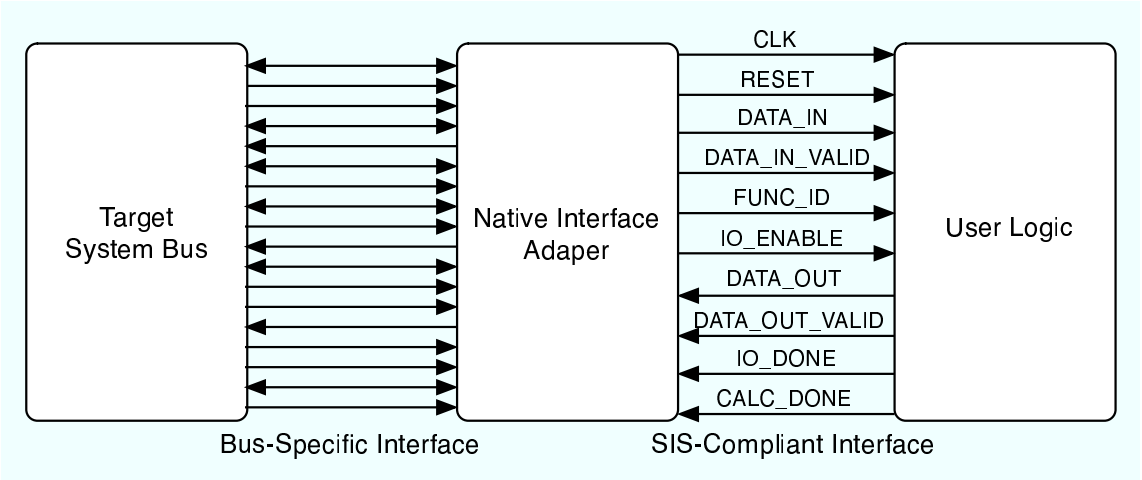

Figure 4.1: Overview of the SIS

Through indirect conversion, Splice is able to isolate user-logic from the constraints of any specific interface by conducting all transactions directly in terms of the SIS. So long as a native interface adapter exists for each platform that a developer wishes to target, device implementations can be moved freely without the need to make any modifications to their internal signaling protocols. In this manner, the tool is able to generate hardware for a variety of different platforms without the inherent scalability issues and inflexibility associated with direct one-to-one bus adaptation.

4.2 The SIS Transfer Protocol

The SIS consists of a collection of 10 interface signals that are used to redirect data from an external system bus into user-defined peripheral functions and vice versa. These signals combine to form a generic interface that can be used to define all the common types of read and write operations that a peripheral might require for both strictly synchronous and pseudo asynchronous interfaces. A table listing the functionality of each signal can be seen below in Figure 4.2.

While the signals of the SIS are well-defined in terms of the functions they serve, the exact methodology used to adapt a particular system bus to the protocol will vary depending on the complexity of the interface that is being targeted. To ensure interoperability among Splice logic, however, a number of communication axioms have been defined that serve to dictate how an SIS adapter should interact with code that is created via the tool. In doing so, the intent was to allow developers the freedom to adapt a foreign interface in any way they see fit so long as compatibility is maintained at the user-logic level.

The SIS protocol is designed to be lightweight in nature so that a minimal impact on hardware timing and performance is incurred when using Splice as an intermediary. As a result, advanced features such as DMA or burst transfers are not directly supported. Instead, these types of operations are converted into SIS-compliant interactions by the bus adapter that acts as an intermediary between the target bus and Splice-generated logic. This increases the complexity of the adapter somewhat, but results in a streamlined transmission protocol that can handle both strictly synchronous and pseudo asynchronous interfaces.

Signal Name	Type	Purpose
CLK	Broadcast	Global clock signal used to coordinate all bus transactions.
RST	Broadcast	Reset signal used to terminate current operations and return the user logic to a known state.
DATA_IN	Broadcast	Input data from the processor for use by the user logic.
DATA_IN_VALID	Broadcast	Used to signal that input data is valid and is waiting to be stored in the user logic.
IO_ENABLE	Broadcast	Used to signal the arrival of a new data request (read or write) in order to ensure proper timing of burst and DMA transactions.
FUNC_ID	Broadcast	Used to target a specific user logic function in the system and direct I/O requests across the SIS.
DATA_OUT	Per-Function	Output data from the user logic in response to a processor request.
DATA_OUT_VALID	Per-Function	Used to signal that output data is valid and is waiting to be read via the processor.
IO_DONE	Per-Function	Used to signal the SIS that the previous load or store operation sent to this function has completed.
CALC_DONE	Per-Function	Used to signal that the calculation operations performed by this function have all completed.

Figure 4.2: Listing of the Functionality of Each SIS Signal

4.2.1 Pseudo Asynchronous Bus Transfer Protocol

To support pseudo asynchronous bus transfers via the SIS, a simple generalized transmission protocol is defined. This protocol relies on the fact that most (if not all) pseudo asynchronous interfaces provide handshaking mechanisms that are used to signal the completion of each I/O operation. As a result, when a read operation is executed across a pseudo asynchronous bus, further bus operations pause until the read has been serviced. This ensures that all operations complete in-order and that proper timing between subsequent transmissions is maintained.

To pass data into a user function (write) across a pseudo asynchronous bus, an SIS-compliant interface must first place the data on the DATA_IN signal and set the

DATA_IN_VALID "active." At the same time, the interface must also set the FUNC_ID signal to match the pre-assigned identifier of the function for which the input data is destined, and the IO_ENABLE signal must be strobed for a single cycle. The DATA_IN, DATA_IN_VALID, and FUNC_ID lines must then remain static until the targeted hardware function raises its IO_DONE line for a single clock cycle to signal completion of the input operation. At this time, the bus interface can lower the DATA_IN, DATA_IN_VALID, and FUNC_ID signal lines to prepare for the next transaction.

Similarly, to request data (read) across a pseudo asynchronous bus from a user function, an SIS-compliant interface must set the FUNC_ID signal to match the pre-assigned identifier of the targeted function while strobing the IO_ENABLE signal for a single cycle. The FUNC_ID signal must then remain static until the targeted hardware block places data onto the DATA_OUT signal lines and raises both the DATA_OUT_VALID and IO_DONE signals. These values will then be held static for a single clock cycle, at end of which they are lowered again. At this point, the bus adaptor should pass the output data back to the system bus, lower the FUNC_ID signal, and prepare for the next transaction. The precise timing required to implement pseudo asynchronous input and output data transfers across the SIS can be seen in Figure 4.3.

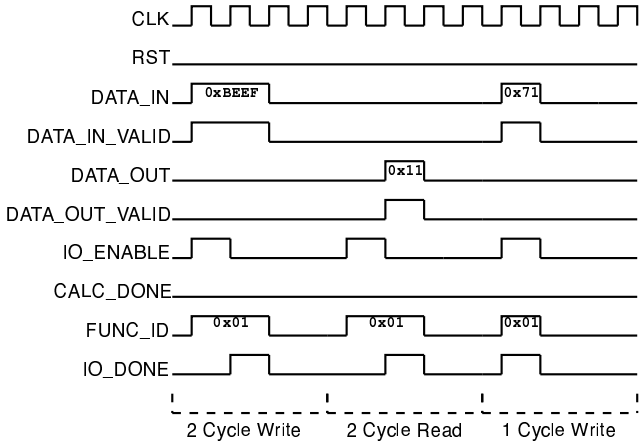

Figure 4.3: The SIS Pseudo Asynchronous Transmission Protocol

In examining the signal listing above, it may at first seem that the IO_ENABLE signal is somewhat redundant and that the functionality of this signal is actually duplicated by the FUNC_ID vector. In reality, however, the protocol of many common bus structures dictates that not all signals need to return to the idle state after a

transaction has occurred. As a result of this, issues could arise where the FUNC_ID vector could be signaled high even after a data transmission has completed, leading to complications when multiple data values need to be moved across the bus. To alleviate this issue and ensure proper timing of I/O transactions, the IO_ENABLE signal was created as a means of indicating when a *new* data transfer has been presented to the peripheral hardware. The specific details of how this signal is implemented depends largely upon the interface that is being targeted; an example of how this is done via the PLB can be found in Section 4.3.

4.2.2 Strictly Synchronous Bus Transfer Protocol

To support strictly synchronous bus transfers via the SIS a method similar to that employed to handle pseudo asynchronous transactions is used. The important difference being that strictly synchronous interfaces generally do not provide handshaking mechanisms that signal the completion of each I/O operation. As a result, when read operations are executed across a strictly synchronous bus, the interface cannot be paused until a result is available. To work around this limitation, the CALC_DONE signals defined in the SIS protocol are utilized as a barrier between write and read transactions to ensure that bus transactions are completed in-order.

To write to a user function across an strictly synchronous bus, an SIS-compliant essentially follows the same procedure as used for pseudo asynchronous bus transactions. The only significant difference being that the IO_DONE signal is unused for asynchronous transaction, because all write transactions must complete in a single cycle. Note that this limitation does not preclude the ability to support multi-value transfers, however, since a number of single-cycle transactions can be chained together.

Similarly, to execute read transactions across an strictly synchronous bus, the same basic procedure used for pseudo asynchronous transmissions is applied. In this case, the difference lies in the role of the CALC_DONE signal that each user function produces. Since there is no generalized method that can be used to determine when a peripheral attached to an strictly synchronous bus has completed operations and is ready to produce output, a polling approach is employed. Simply stated, the processor polls the current value of CALC_DONE signal array until the bit corresponding to the user function identifies with (as determined by the FUNC_ID) is raised. Once this is done, the processor knows that the function is in it's output state and that read

operations can proceed. The precise timing required to implement asynchronous input and output data transfers across the SIS can be seen in Figure 4.4.

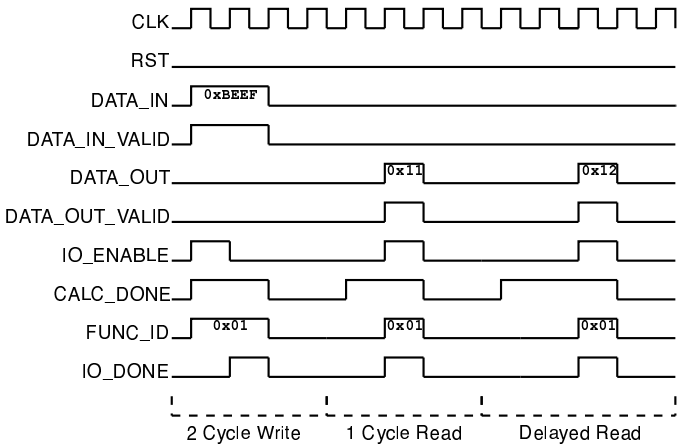

Figure 4.4: The SIS Strictly Synchronous Transmission Protocol

Since the raising and lowering of `CALC_DONE` signals is handled internal to the user function, the bus adapter arbitrating between the native and SIS transfer protocols needs to provide a mechanism to read the current value of the `CALC_DONE` signal at any given point. Therefore, the SIS standard dictates that function identifier zero be reserved for this purpose and mapped to the `CALC_DONE` vector (or "status" register) for all related reads that are enacted across the shared bus. Doing so ensures that Splice-generated software drivers will be provided with a standardized method by which to request status updates from any type of interface supported by the application.

4.3 Adapting Existing Bus Protocols to the SIS

To develop an efficient native interface adapter for an existing transmission protocol, a strong knowledge of the various quirks associated with using the underlying interface is required. In cases where complex features such as DMA or burst transactions need to be supported, the code required can become somewhat convoluted and difficult to debug. For most interfaces, however, an appreciable amount of functionality can be "ported" to the SIS through a few simplistic signal assignments. In an effort to illustrate this point, we present the steps required to create an interface adapter for the PLB.

4.3.1 The PLB Interface

As discussed in Section 2.3.2, the PLB is a member of the IBM CoreConnect specification that is often deployed as a high-speed pseudo asynchronous peripheral interface in sophisticated embedded systems. Specifically, this interface has seen extensive usage in Xilinx Virtex2Pro and Virtex4FX FPGA SoC devices where it is used to attach peripheral logic to the on-board PowerPC 405 processor. In these applications, the bus is mapped to an address space in main memory and typically runs at a clock rate of 100 MHz to match the requirements of the embedded CPU [21, 16].

By leveraging the flexibility inherent in FPGAs, Xilinx has created an implementation of the PLB that is highly configurable in nature and that has the the ability to support advanced features such as DMA and interrupt-driven I/O. The data width of the bus can also be configured to support a native transfer size of either 64 or 32 bits. Due to the internal architecture of the PowerPC 405, however, only 32-bit can be transferred directly from the CPU to a peripheral in a given cycle, thereby negating the benefits of the wider bus unless DMA is also enabled [16, 20, 21]. For the purposes of this "example" interface adapter, only the methods required to support simple read and write operations across a 32 bit PLB interface will be considered.

As is the case with most memory-mapped interfaces, the PLB allows developers to reserve a contiguous segment of addresses and deploy their design across them. When a memory address that is assigned to a peripheral is accessed, a single bit of either the `RD_CE` (read chip enable) or `WR_CE` (write chip enable) interface signals are triggered via one-hot encoding, to indicate which addresses the ensuing bus transaction is intended for. On that same cycle, or one closely thereafter the `BE` (byte enable) signal is also set to indicate how much data the transaction will operate upon. In a 32-bit PLB implementation this signal will always take on a value of "1111" when activated, indicating that a 32-bit (4 byte) transfer will follow [16, 20].

To enact a read transaction at this point, the PLB bus would strobe the `RD_REQ` (read request) for a single cycle and then wait for the targeted peripheral to respond. While waiting, the bus would keep the `RD_CE` and `BE` signals steady to ensure that the data request was received. The user-logic would then respond by placing a value on the `DATA_OUT` line and triggering the `RD_ACK` (read acknowledge) signal for a single clock cycle. This, in turn, would cause the bus to lower then `RD_CE` and `BE` lines, thereby signaling the end of the transaction. A graphical representation of this protocol can be seen in Figure 4.5.

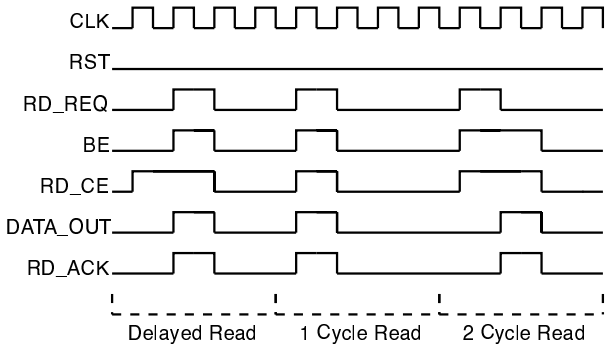

Figure 4.5: The PLB Read Protocol

Write transactions can be enacted in a similar fashion by strobing `WR_REQ` (write request) while simultaneously placing data on the `DATA_IN` bus line. The `DATA_IN`, `WR_CE` and `BE` signals would then be kept in a steady state until the user-logic responded via the `WR_ACK` (write acknowledge) signal. Once received, this event would trigger the lowering of the `WR_CE` and `BE` lines as an indication that the transaction had completed [16, 20]. The timing diagram in Figure 4.6 provides a formal declaration of this protocol.

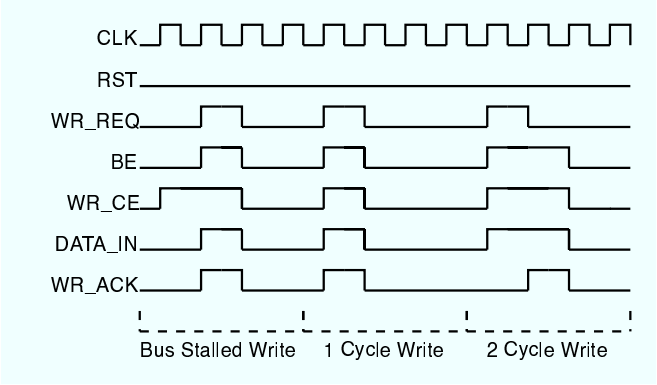

Figure 4.6: The PLB Write Protocol

4.3.2 Signal Adaptation

In comparing the PLB interface presented in the previous subsection to the pseudo asynchronous variant of the SIS shown in Figure 4.3, one would notice a great number of similarities between the two interconnects. In fact, at first glance it would likely appear that the PLB and the SIS made use of the same transmission protocol, but did

so across a somewhat different collection of signals [20]. If analogous comparisons were then conducted with the CoreConnect FCB or APB interfaces, the conclusions drawn in all cases would be largely identical [17, 3]. In other words, these resemblances are not entirely coincidental, but are rather a side-effect of modern design practices.

Simply stated, when communicating data between the various components in a system, there are only so many "unique" methods via which to conduct the transactions. Furthermore, because the area of interface design has been well-studied over the past 50 years, a general consensus among developers has emerged in regards to how information should be transmitted within a computer system. As a result, the vast majority of interfaces in use today tend to employ protocols that are functionally equivalent to one another and that conduct their transmissions in a largely identical manner.

The diagram shown in Figure 4.7 illustrates this concept by providing a signal-level comparison of the behaviors of the PLB of SIS protocols in response to a set of simple read operations. In the figure, the signals of the two interfaces have been arranged in such a manner that those on the same horizontal placement directly correspond to one another. For example, the PLB `RD_REQ` signal is shown across from the `IO_ENABLE` signal of the SIS interface because they both accomplish the same task by way of identical signal manipulations. Similar comparisons can be made for all other signals in the diagram with the exception of the `RD_CE` and `FUNC_ID` lines. To perform this particular adaptation, the one-hot `RD_CE` value from the PLB must first be transformed into a pure binary value and then linked to SIS `FUNC_ID` signal [16, 20].

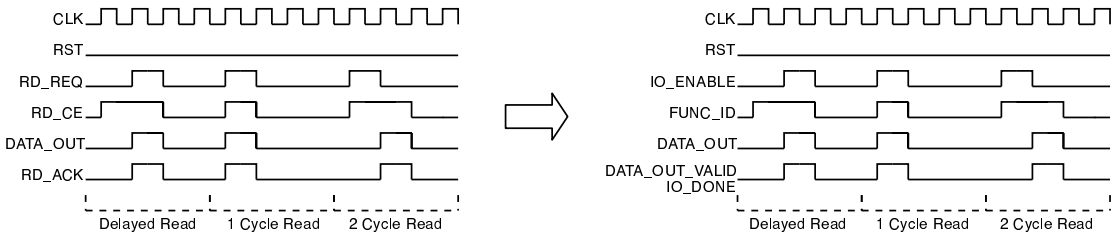

Figure 4.7: Adapting Between the PLB and SIS Read Protocols

For write transactions, a similar diagram can be constructed (see Figure 4.8), resulting in a comparable set of interface signal translations. By combining the relationships shown in both diagrams and converting them into a more formalized description, a native interface adapter for the PLB can be produced in the VHDL

format. This implementation can then be linked into Splice via the expansion API described in Chapter 7 of this document, after which time the bus can be selected as a hardware target by any developers who make use of the tool.

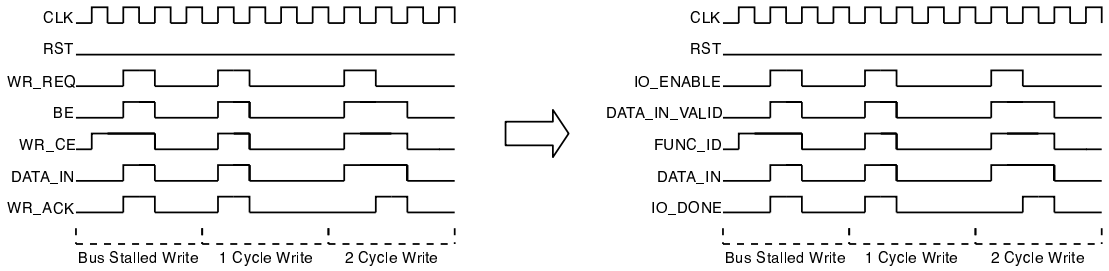

Figure 4.8: Adapting Between the PLB and SIS Write Protocols

Chapter 5

User Logic and Interface Generation

User logic and interface generation is a multi-step process that consists of three independent stages. In the first stage, one or more top-level interface files are generated to link user hardware blocks to the specified system bus. Next, a bus arbitration file is generated to handle the multiplexing of shared signals between each user-specified function. In the third and final stage, a separate user-logic stub file is generated for each hardware prototype that handles all related input and output operations automatically. The method by which these files are linked together to form a coherent system can be seen in Figure 5.1.

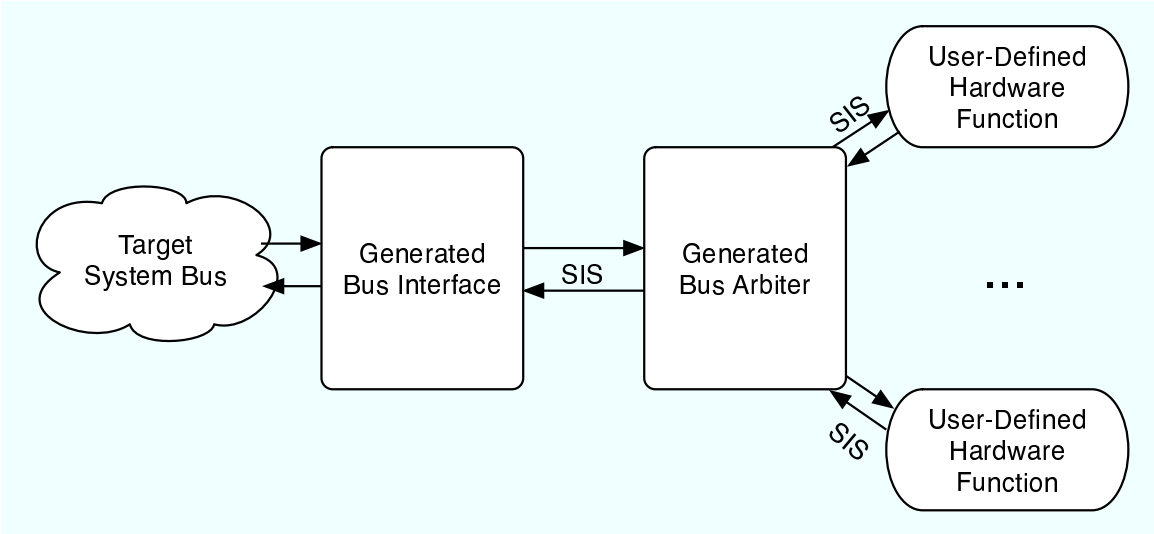

Figure 5.1: Interconnections Between Generated HDL Files

The contents of the logic, arbitration, and interface files that are generated will vary greatly from project to project. This is due to the fact that factors such as the format of the input function prototypes, the bus interface being targeted, and the types of "advanced" features the end-user requires all have an impact on the structure of the resulting stubs. What is constant, however, is the method used to generate the files and the relationships between them.

5.1 Bus Interface Generation

Generation of bus interfaces is accomplished by consulting a set of reference HDL files that describe the basic logical statements and wired connections that are required to implement the target system interface. These files essentially adapt the bus's native input signals into the format required by the SIS (as described in Chapter 4) to allow bus-independent hardware blocks to be created and deployed.

Embedded in these reference files are macro symbols of the form of '`%SYMBOL%`' that are parsed out by the generation routine and replaced with the logic required to generate a functionally-complete bus. For instance, a macro named '`%DMA_SUPPORT%`' might need to be replaced with the proper logic to handle DMA operations or simply ignored depending on what bus features the user has chosen to enable. Since certain bus structures might require multiple HDL source files to implement, a number of template files might need to be referenced to generate the entire target interface.

By virtue of the flexible transfer protocol defined by the SIS, Splice is able to support almost any type of bus interface that is currently available or will exist in the future. Since each interface the tool supports has it's own unique signaling protocol, the adaption techniques used tend to vary from bus-to-bus. Therefore, unlike the other modules covered in this chapter, it is somewhat difficult to pinpoint the exact structure of a "typical" Splice-generated bus interface. Further complicating the situation is the fact that Splice provides support for end-user define interfaces that are loaded via dynamic libraries referenced at run time. The specifics of this feature are explored in-depth in Chapter 7 of this document.

5.2 Arbitration Unit Generation

Once a proper interface file for a peripheral has been generated, Splice then creates an arbitration unit to sit between the bus and user-defined hardware functions in order

to multiplex access to the shared output signals defined by the SIS. To perform this arbitration, each user-defined function (or instance of a function if multiple instances are required) is assigned a unique function ID based upon a parameter that is specific to the target bus. Software drivers generated by the tool can then use these identifiers to target particular hardware functions and orchestrate calculations. This method of access control is required because all user-logic blocks are attached to a common connection point (the bus), and thus would corrupt the shared signaling lines if some method of arbitration were not employed.

Along with handling the multiplexing of control and data signals between each user-defined function, the arbitration unit is also responsible for maintaining transaction state when an strictly synchronous bus interface is targeted. This is done by collecting the various "calculation complete" (`CALC_DONE`) signals generated by each function and concatenating them into a single vector. This vector is then passed to the generated interface where it used to aid in the orchestration of read transfers across the shared strictly synchronous bus. In cases that pseudo asynchronous bus is targeted, the amalgamated signal is still passed to the generated interface, but can safely be ignored since other signals (namely `IO_DONE`) will handle the handshaking in a synchronized manner.

In theory, the multiplexers defined within the arbitration file could create a performance bottleneck if a set of functions has a high ratio of I/O to business-logic time. In reality, however, system interfaces are typically shared between a number of devices, yet can only be accessed by a single peripheral at any given time. As a result, having multiple components share the same bus interconnect merely shifts the point at which arbitration occurs and therefore should not create a performance bottleneck within the system that would not have otherwise existed. Furthermore, by sharing the same bus interface between all hardware functions, any additional connection points on the bus will be available for use by other peripherals, such as Ethernet devices or memory controllers. The impact that multiplexing I/O operations has upon performance is further explored in Chapter 9.

Beyond the access coordinating roles already specified, the arbitration unit is also responsible for instantiating the connection linkage for each user-defined prototype. If multiple instances of a specific function are required, then the arbitration unit will transparently handle the necessary signal routing and function ID assignment to enable said functionality. By handling the peripheral-to-bus linkage at this level, the

layout of the user-logic stubs can be simplified and thus made more easily portable between projects.

5.3 User-Logic Stub Generation

After interface and arbitration files have been generated for a particular peripheral device, Splice then creates user-logic stubs for each prototype that the end-user has defined. In an effort to modularize designs and enable the end-user to easily move hardware components between projects, a separate stub is created for each function. Since each generated file contains only the logical statements required to implement a single function, all hardware blocks in the system are able to operate concurrently. This means that while only one function can have control of the shared system bus at any given time, all other functions in the system can still perform calculations without interruption. This, in turn, minimizes the performance impact associated with using a shared-bus structure.

In terms of functionality, each generated stub contains a bare minimum of two distinct functional units known as the ICOB (input-calculation-output block) and the SMB (state machine block) These blocks work together to handle all bus interactions for a particular hardware function, as well as to provide support for any calculation logic that may be added by the end-user after the files are generated. Interaction between the two blocks is orchestrated via a set of state variables extracted from the prototypes passed into the tool that define the baseline input and output states for a given function. If the default functional units do not provide enough flexibility to implement a desired function, then the end-user is free to add any additional blocks that are required. The relationship between the ICOB and SMB is portrayed in Figure 5.2.

5.3.1 The Input-Calculation-Output Block (ICOB)

The ICOB is a clock-sensitive unit that is responsible for implementing all bus interactions for a specified hardware function in the order expressed within its corresponding prototype. By default, all input and output bus-level signaling, such as read and write verification (for both pseudo asynchronous and strictly synchronous interfaces), is handled by the ICOB using the protocols defined in Chapter 4. In contrast, the task of data storage and transfer is not automated within the ICOB,

but rather left up to the end-user to implement as they see fit. The reasoning behind this is that end-users may want to allocate their own system-specific structures such a Block RAM or register files for storage that are outside the scope of Splice.

As discussed in section 4.2, the pseudo asynchronous and strictly synchronous transfer protocols that Splice implements share many of the same interface signals. In fact, the only functional difference between the two transmission classes is that while the former only requires use of the `IO_DONE` line to signal the end of a transaction, the later must also toggle the `CALC_DONE` vector each time a given set of hardware operations is completed. By leveraging the similarities between these two protocols, the tool is able to generate user-logic stubs that support both transactions types with only a small amount of additional overhead. This is accomplished by instantiating the logic required to handle strictly synchronous "handshakes", regardless of the type of interconnect that is indicated by the target specification. Since pseudo asynchronous interface adapters simply ignore the `CALC_DONE` signal by convention, this simple modification is all that is required to allow for the portability of user logic implementations across both variants of the SIS protocol.

As information flows into user functions, state progression within the ICOB flows from input, to calculation, and finally to output in a manner determined by the SMB. The input stages within a function mimic the order and structure of those defined within the associated software prototype and are setup to flow from one to next as valid input is received by the peripheral. A single calculation stage is initially left blank for the end-user to fill in, and others can easily be added if support for multi-cycle operations is required. Finally, the single output state is responsible for passing data out of the function while handling all associated bus-level signaling. This includes the task of raising the "calculation complete" signal and keeping it stable until results are read from the function to support strictly synchronous interfaces.

When advanced bus features such as packed, split, or array-based (either implicit or explicit) transfers are used within a prototype, additional elements such as registers and comparators will also be included in the ICOB. In the case of packed or explicit array transfers, a tracking register and comparator are instantiated to track the number of values that are required to be passed into or out of the device to handle a particular input or output. For implicit array transfers, both tracking and storage registers are defined along with a comparator to track the number of entries required for a specific input or output depending upon changes in the value of index variable.

In all cases, the tracking register can also be used as an index into on-chip memory or a register file to simplify bulk data storage or transfers.

In the case that a blocking function that returns no output is defined, a special pseudo output state is created within the ICOB. This state is then used to report status signals back to the software driver associated with the function when all calculation states have been completed. In doing so, the associated software driver is able to issue a read command across the bus after all input states have been handled and effectively pause execution of the associated end-user application until a "completion" signal is received.

Special care is also taken within the ICOB to alert the user of cases in which "erroneous" values will be transferred across the bus. Most often these types of situations arise when a set of packed or split data items do not fit neatly into a integer number of bus transactions. As such, the last portion of the data transfer will be sent across the bus along with a number of bits of unknown data that does not correspond to any true data values. In most cases, the end-user can simply ignore these trailing bits and go about processing other inputs or handling calculations. It is not always obvious, however, under what conditions these types of transactions will arise. Therefore, in an effort to aid in the identification of such transmissions, descriptive comments are included above each ICOB state handling routine that note how many bits of each set of transactions can safely be ignored by the hardware.

The various support mechanisms defined above work together to ensure that all I/O operations will complete in the proper order, while also allowing for independent calculations to be launched in parallel. Therefore, by filling in the pre-defined calculation state and defining others as needed, the end-user can define a wide variety of calculation operations within the confines of ICOB. This, in turn, allows hardware to be developed in a more flexible and less error-prone manner.

5.3.2 The State Machine Block (SMB)

Working in tandem with the ICOB, the SMB is responsible for updating the "current state" signal that is used to move between the various input, calculation and output operations defined within the ICOB. Transitions within the SMB are accomplished via a set of combinational logic statements that are activated each time the ICOB requests a state transition. Since the ICOB is itself a clocked process it will only request a

state change a maximum of once per cycle, thus ensuring ordered completion of all function-related operations.

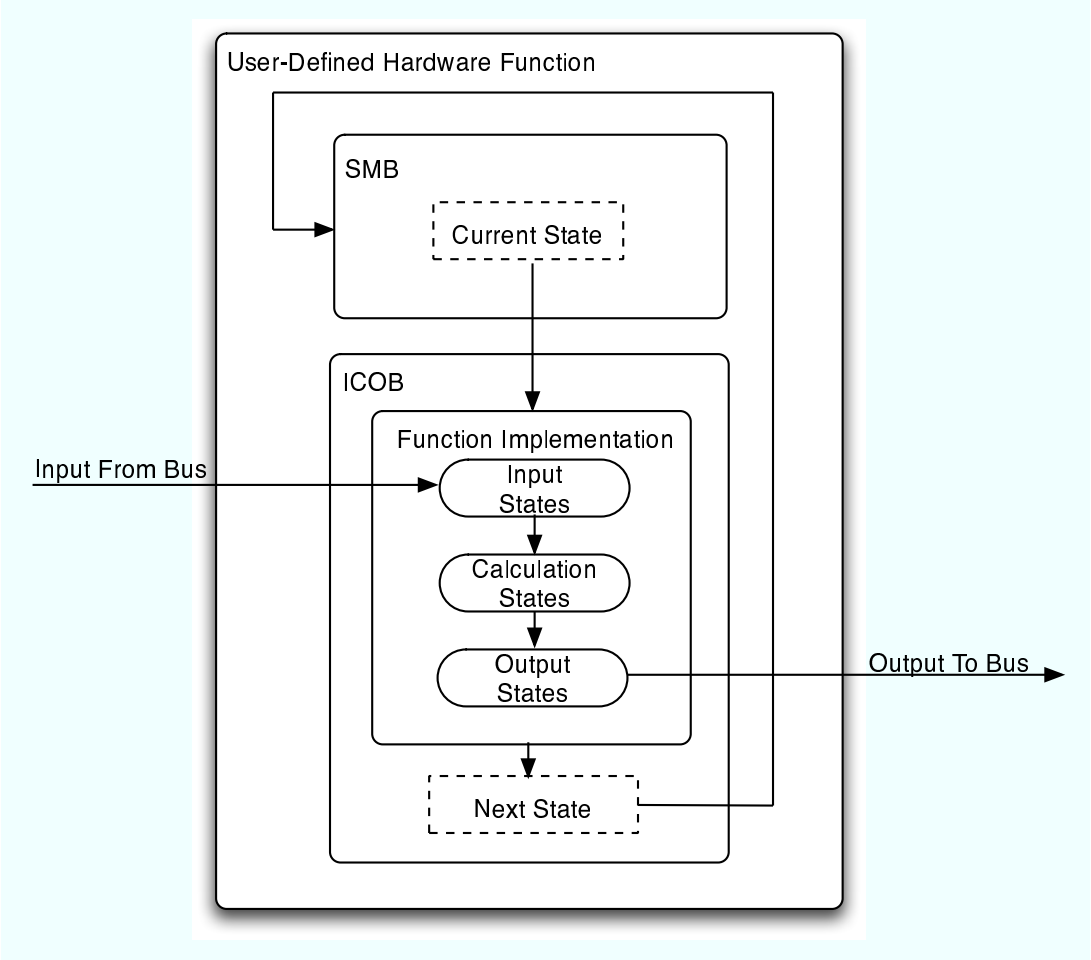

Figure 5.2: Layout of a Typical User Logic Stub

5.3.3 Additional User-Logic Stub Components

In addition to the ICOB and SMB, each user-logic stub also incorporates a number of pre-defined constants, variables, and non-synthesized functions that aid in handling various hardware-related tasks. For instance, the function ID assigned to an object is defined as a constant with the logic stub so that the ICOB can identify when a function-specific input or output request is made. Furthermore, if packed, split, or array-based transmissions are used within a function, a number of related variables and constants will be pre-defined in order to aid in the handling of such transactions. By combining these constants and variables along with the ICOB and SMB, a

functionally complete set of bus interactions for each user-defined prototype can be generated.

Chapter 6

Software Driver Generation

In an effort to streamline the process of integrating Splice-based hardware designs into embedded systems, the tool implements an extensive set of driver generation routines. Through these facilities, driver code abiding by the ANSI C language syntax can be produced that mimics the calling conventions of each interface declaration passed to the tool. By linking these drivers into new or existing software applications, calls to user-logic can be executed in a transparent manner, thereby freeing developers from the complications typically associated with orchestrating bus-related data transmissions. Within the remainder of this chapter, the various mechanisms that Splice employs to accomplish this task are described in detail with a particular focus on the abstract manner in which the tool structures the drivers it creates.

6.1 Coordinating Software Transactions

The means by which bus communication is accomplished within a given system is heavily dependent upon both the type of interface that is being targeted, as well as the underlying architecture of the processor through which data transfers will flow. As a result, it would be somewhat infeasible to directly generate customized drivers for each platform that the tool is capable of supporting. Instead, Splice implements a bus-independent software-side interface that handles all CPU-to-bus interactions for the various user functions in a system through a collection of generic functions that are defined in the form of macro operations. Each bus interface supported by the tool has it's own set of declarations for these macros stored in the form a standard C-syntax header file. By linking the abstract drivers produced by the tool to one or more

of these libraries, fully-functional software-to-hardware interfaces can be produced in a platform-agnostic manner.

To direct data to and from a specific hardware function, drivers make use of the identifiers assigned to each prototype during hardware generation by passing them into the macro operation defined for each supported interface. Since each user-logic object is assigned a unique function identifier (`FUNC_ID`), the driver can use these values to specifically target individual hardware blocks via the shared bus interface. This, in turn, ensures that any data passed across the bus reaches its proper destination, and that the various user-defined hardware blocks in the system can operate without interference.

6.1.1 Simple Transaction Macros

In order to provide support for simple data transactions, Splice requires that all supported bus mechanisms define macros for standard single-word load and store operations, as well as more advanced double and quad-word macros to more efficiently handle array-based transactions. Typically, these macro declarations consist of a set of assembly language instructions that define low-level transactions to the targeted bus mechanism. If a bus is unable to natively support one of the baseline declarations, then it must redefine the related macro in the form of transactions that are allowed to be conducted across the interface. For instance, if burst support is not provided by a specific bus, then the standard quad-word store routine could be defined in the form of four sequential single-word store operations. Doing so allows the driver generation routines within Splice to create highly optimized drivers for bus mechanisms that support fast transactions while still providing an equivalent level of functionality for simpler interfaces. A sample driver making use of these transaction macros can be seen in Figure 6.1. For a more complete description of the software macros currently supported by the tool, the reader is advised to consult Chapter 7 of this document.

When packing or splitting of data values is requested, these same transaction macros are coupled with a byte-wise incrementing pointer to handle the proper number of items each transmission cycle. Similarly, in the case of multi-valued outputs, the memory required to read multiple values from the hardware is automatically allocated and a pointer to the allocated array is passed back to the user program for processing after the proper results are read from the hardware via macro operations.

```
// ID Used to Target sample_function
#define SAMPLE_FUNCTION_ID 2

// Driver Used to Activate sample_function in HW
float sample_function(int* x, int y)
{
  // Allocate Storage for output and address
  float    result;
  unsigned func_addr;

  // Determine the Address of the Function
  func_addr = SET_ADDRESS(SAMPLE_FUNCTION_ID);

  // Transfer Two Values of x
  WRITE_DOUBLE(func_addr, &x[0]);

  // Transfer One Value of 'y'
  WRITE_SINGLE(func_addr, &y);

  // Wait for Calculations to Complete
  WAIT_FOR_RESULTS(func_addr);

  // Grab Result from Hardware
  READ_SINGLE(func_addr, &result);

  // Return Results to Calling Function
  return result;
}
```

Figure 6.1: Splice-based Driver Code for a Simple Hardware Function

Since drivers have no means of accessing the dynamic storage they allocate once control has returned to the application, end-users must remember to eventually "free" the data associated with the returned pointer or run the risk of developing significant memory leaks during the course of the programs execution.

Beyond simple data-related operations, Splice also dictates that proper timing of transactions across strictly synchronous bus mechanisms be managed within the software driver. This is done primarily to eliminate additional logic in the hardware, but also to streamline requirements for pseudo asynchronous bus mechanisms. To support these transactions, Splice inserts a "pausing" function call in each generated driver after the point at which all data has been written to the hardware but before any results are read back from the hardware. In accordance with the protocol defined in Chapter 4, a user logic function will raise a flag (`CALC_DONE`) when all calculations have completed and output is available for transfer.

To check for the arrival of the flag in software an asynchronous bus will typically implement a while loop that polls a dedicated address (via function identifier zero) on the shared bus for the current status of the calculation status register. The value retrieved can then be compared with the function identifier of the related hardware

block to see if they match. If so, then the loop can be stopped and results can be read from the hardware. If the values do not match, the while loop continues to poll the status register until calculations for the associated user function have completed. In the case of a pseudo asynchronous bus these tests are unnecessary, and can simply be replaced with a "NULL" statement in the associated macro definition.

6.1.2 Advanced Transaction Macros

Beyond the simple read and write operations outlined above, Splice is also able to generate drivers for complex interface declarations such as those that make use of the DMA or multi-instance syntax extensions that are provided by the tool. In doing so, the flexibility offered by a macro-based driver creation system is once again leveraged to produce software-to-hardware interface routines in a generalized manner. This, in turn, ensures that even the most complex bus operations can be encapsulated and abstracted, allowing for driver portability between platforms and reducing to amount of low-level software code that developers must produce to link hybrid systems together.

When a developer requires multiple copies of the same function, special considerations must be made to properly handle the individual hardware instances produced by the tool. In such cases, a single software driver is created to arbitrate access to all instances of the function. In addition to the I/O parameters specified in the original prototype, such drivers also take in an additional index parameter (`inst_index`) that is used to target a specific instance of the hardware block. This, in turn, allows the same software calling conventions to be used in all applications or threads that require concurrent access to specific instances of the hardware. The code segment provided below in Figure 6.2, extends the previous example as a means of demonstrating how such routines are structured.

Other advanced I/O features such as DMA transactions that may be required to implement a function are handled transparently within the driver via macros in a similar fashion to simple load and store transactions. For instance, any DMA-specific setup or timing is handled within the driver via specialized macro calls to ensure that bulk data transfers to and from hardware are executed properly. By providing support for such features internally, the software designer is free to focus on writing the actual application code to drive the hardware functions, as opposed to dealing with how to best implement complex bus transactions. As support for additional

features such as interrupts are added to the tool in the future, similar abstractions will be created to expose the functionality in a flexible manner.

```
// ID Used to Target sample_function
#define SAMPLE_FUNCTION_ID 2

// Driver Used to Activate sample_function in HW (w/ Multiple Instances)
float sample_function(int* x, int y, int inst_index)
{
  // Allocate Storage for output and address
  float    result;
  unsigned func_addr;

  // Determine the Address of the Specific Function Instance
  func_addr = SET_ADDRESS(SAMPLE_FUNCTION_ID + inst_index);

  // Transfer Two Values of x
  WRITE_DOUBLE(func_addr, &x[0]);

  // Transfer One Value of 'y'
  WRITE_SINGLE(func_addr, &y);

  // Wait for Calculations to Complete
  WAIT_FOR_RESULTS(func_addr);

  // Grab Result from Hardware
  READ_SINGLE(func_addr, &result);

  // Return Results to Calling Function
  return result;
}
```

Figure 6.2: Splice-based Driver Code for a Function with Multiple Hardware Instances

Chapter 7

Extending Splice

By default, the Splice distribution includes a set of built-in support libraries that allow end users to target their peripherals to a number of common embedded bus interfaces. The included libraries all abide by the SIS (as defined in Chapter 4, enabling the end user to take advantage of bus transparency. In the future, however, it is likely that new interfaces will be developed for use with the next generations of SoC FPGAs that are incompatible with the mechanisms Splice currently supports. As a result, peripherals created with the aid of Splice could, in effect, become unusable long before they were technically obsolete. Users who wish to move their existing designs forward to the new interfaces would then be forced to needlessly rewrite relevant portions of their code simply to maintain compatibility with the updated interfaces. Therefore, in an effort to prevent Splice-based designs from becoming technological dead ends, an API to aid in the creation of custom bus generation libraries was developed.

7.1 The Splice Interface API

The Splice bus interface API is formed by a set of hooks built into the tool that make use of external dynamic libraries written in C or C++. When the tool is activated, these hooks are linked (via function pointers) to specific routines contained in the external library, which are then used to generate key portions of the targeted bus interface. These libraries are then coupled with one or more bus-specific annotated HDL template files that are parsed and modified to create the desired bus interface file. Since user-logic and arbitration logic are both bus-independent entities, no external library functions are required to generate these types of files.

To simplify bus generation, all API-compliant external libraries are allowed to access the internal data structure (`splice_params`) that Splice uses to track the input specifications that are passed into the tool at run-time. A description of the contents of this structure can be found in Figure 7.3. By allowing developers to access this data, functions defined within external libraries can obtain information pertinent to the interface generation process, such as the type of bus features (bursting, DMA, etc...) the end-user has selected, the hardware functions that have been defined, and functions' inputs and outputs. With all of this information available, the external library can then generate a bus interface that matches the user's requirements without including unnecessary functionality that would slow down the rest of the system.

7.1.1 API Routines for Creating Native Bus Adapters

In essence, the native bus adapter is the keystone from which the entire functionality of Splice is derived. Without these components it would be impossible to link the abstract signals of the SIS into a physical system that is capable of making use of the various hardware logic and software driver files produced by the tool. The discussion provided in Section 4.3 of this document outlines how such an adapter might be constructed, but does not actually specify how the resulting file could be linked into Splice. To accomplish this goal, a templatized version of the adapter must be created using a set of macros provided by the tool.

The purpose of these macros is merely to provide the ability to generate native interface adapters in a flexible manner. For instance, the `%FUNC_ID_WIDTH%` macro is used to denote how many bits the `FUNC_ID` signal will contain. By inserting this macro in place of explicit signal sizes within the interface adapter, the code produced by Splice after parsing this file will reflect the correct signal width and ensure that the bus arbiter can function properly. Other macros, such as `%GEN_DATE%` simply produce helpful messages (a timestamp in this case) that can be used to identify the output produced by the tool. A listing of the hardware-related macros built into Splice can be found in Figure 7.1 along with descriptions of the various purposes they serve. In cases where one of these macros does not perform the specific functionality required by a given interface adapter, bus-specific markers can be added via the procedure outlined below in Section 7.1.2.

Macro Name	Replacement Text
COMP_NAME	Name assigned to the device (i.e. hw_timer).
BUS_WIDTH	Width of the targeted bus.
FUNC_ID_WIDTH	Width of the function identifier field.
BASE_ADDR	Base Address of the Hardware device.
GEN_DATE	Date and time at which files were generated.
DMA_ENABLED	Boolean value indicating if DMA is enabled.
FUNC_NAME	Name of the function currently being examined by the program.
MY_FUNC_ID	Numerical identifier assigned to the current function.
FUNC_INSTS	Numerical identifier that contains number of instances of function.
FUNC_CONSTS	HDL constants for the current function.
FUNC_SIGNALS	HDL signal definitions for the current function.
FUNC_FSM	HDL state machine process for the current function.
FUNC_STUB	HDL I/O handler stub process for the current function.
DATA_OUT_MUX	HDL multiplexer to arbitrate access to the `DATA_OUT` signal.
DATA_OUT_V_MUX	HDL multiplexer to arbitrate access to the `DATA_OUT_VALID` signal.
IO_DONE_MUX	HDL multiplexer to arbitrate access to the `IO_DONE` signal.
CALC_DONE_ENCODE	HDL encoder to bring together `CALC_DONE` values from each function.

Figure 7.1: Listing of Splice Hardware API Macros

7.1.2 API Routines for Generating Hardware Interfaces

While a native bus adapter template provides a high-level description of how a given interconnect can be adapted to the SIS, it does not describe the conditions under which such a translation can be performed. Since each bus has a somewhat different feature set and developers define their interfaces in a purely functional syntax, cases could arise where it is not physically possible to generate valid logic for a set of I/O declarations passed to the tool. To address this issue, Splice requires that all external libraries define a set of three ANSI C based API routines that serve to validate the input passed to the tool and guide the structure of the files that are produced. A description of the actions and responsibilities of each of these routines can be found below.

Parameter Checking Routine

The parameter checking routine is responsible for parsing the shared Splice configuration structure (`splice_params`) and determining if any of the features a user has requested support for are not supported by the target bus. This check is required because not all bus mechanisms are capable of providing support for advanced features such as DMA or burst transmissions. If an incompatibility is found, the parameter checking function is designed to alert the end user of the issue and then terminate bus generation. The end-user can then correct the error and attempt to regenerate the interface.

Marker Loader Routine

Another function the interface library is required to construct is a marker loader that defines any bus-specific macros that need to be parsed and replaced within the interface template HDL files that the library makes use of. Each macro that is defined within an external library must also be accompanied by a matching handler function that defines what the associated marker should be replaced with when it is found in an input template file. This system allows library developers to arbitrarily define replacement macros and handle their occurrence in any way that they see fit. By combining these bus-specific macros with the "standard" replacement functions (Figure 7.1) provided by the tool, native interface adapters that adapt to meet the needs of end-users can be constructed in an relatively straightforward manner.

Bus Interface Generator Routine

In addition to the aforementioned functions, each library also needs to define a bus interface generation routine that activates the HDL parser built into Splice. The built-in parser takes in the location of the associated bus adapter template file, along with the path where the output file should be placed. Once properly activated, the parser then searches through the input file, replacing all defined macros blocks by making calls to the appropriate macro handlers using the same procedure discussed in Chapter 5 of this document. At the completion of this operation, the parser then outputs a properly formatted bus interface file to the specified destination. If required, the user-defined arbiter function is allowed to make multiple calls to the parser to generate interfaces for complex bus mechanisms that require more than one HDL file to implement.

7.1.3 API Routines for Generating Software Drivers

Once the appropriate parameter checker, marker loader, and bus interface arbiter have been defined, the bus interface library can be generated and linked into Splice. To enable the creation of software drivers, however, the end-user must also create a C-based header file with definitions for a number of simple read and write macros that can be used to transfer data to and from the targeted bus interface. A listing of the macros supported by the tool and the functionality that they provide can be found in Figure 7.2. As discussed in Chapter 6 of this document, the functions that a developer is required to support are fairly trivial in nature, and can be implemented with only a rudimentary understanding of how the target bus operates. To support optional features such as DMA, however, a deeper knowledge of the underlying protocols of the associated interface will likely be required.

Macro Name	Type	Purpose
WRITE_SINGLE	Required	Used to issue a write transaction of the native bus width (or less).
WRITE_DOUBLE	Required	Used to issue a write transaction two times that of the native bus width (burst).
WRITE_QUAD	Required	Used to issue a write transaction four times that of the native bus width (burst).
READ_SINGLE	Required	Used to issue a read transaction of the native bus width (or less).
READ_DOUBLE	Required	Used to issue a read transaction two times that of the native bus width (burst).
READ_QUAD	Required	Used to issue a read transaction four times that of the native bus width (burst).
SET_ADDRESS	Required	Used to calculate the address of a hardware function.
WAIT_FOR_RESULTS	Required	Used to coordinate transactions across strictly synchronous interfaces.
WRITE_DMA	Optional	Used to issue a DMA write transaction.
READ_DMA	Optional	Used to issue a DMA read transaction.

Figure 7.2: Listing of Splice Software API Macros

7.2 Importing External Libraries into Splice

If provided with a copy of the associated dynamic library and various support files, any developer can deploy peripherals that target a user-created bus interface. In order to be recognized by Splice, however, the library must be named "`lib[x]_interface.so`", where 'x' is the name of the bus the library defines an interface generator for. Then, assuming that all support files are placed in their proper directories, an end user can target the bus by defining 'x' as the name of the desired output bus in their Splice configuration file (i.e. `%bus_type x`). Assuming that the library is structured properly, the interface files generated by Splice will be able to be used as drop-in replacements within any pre-existing project, thereby automating the process of porting devices between interfaces.

```
// Structure That Describes Module Names and Parameters
typedef struct
{
  char            *mod_name;        // Pointer to Name of the User HW Module
  bool            mod_name_f;       // Flag That Determine Whether Or Not the Module Name is Set

  int             hdl_type;         // Targeted HDL [0 = VHDL, 1 = Verilog]
  char            *bus_type;        // Pointer to Proper Name of the Bus
  unsigned int    base_addr;        // Defs Base Address of Device in Hardware
  int             data_width;       // Defs Width of Data Path for a Bus
  int             func_id_width;    // Defs the Maximum Bits to Reserve for the Function ID Field

  bool            packing_f;        // Enable or Disable Packing of Values Onto Higher-BW Buses
  bool            ld_burst_f;       // Enable or Disable LD Burst Operations
  bool            st_burst_f;       // Enable or Disable LD Burst Operations
  bool            dma_support_f;    // Enable or Disable DMA Memory Operations
  int             dma_width;        // Defs Native DMA Transfer Width
  int             dma_max_bits;     // Defs Max # of Bits That Can Be Sent in DMA Operation

  s_func_params   *funcs;           // Pointer to Set of User Functions
  int             nmbr_funcs;       // Lists the Number of Functions Code Will Be Generated For
  int             total_instances;  // Tracks Number of Total Function Instances Defined
} s_module_params;

// Structure That Describes I/O Parameters
typedef struct
{
  char            *io_name;         // Name of the Input                  [i.e. 'x']
  char            *io_type;         // String-Based Input Type            [i.e. 'int *']

  char            *index_var;       // Name of Variable Used As A Variable-Length Array Index
  bool            has_index;        // Denotes Whether or Not an Index Variable Is Used
  bool            used_as_index;    // Denotes Whether or Not Another Variable Uses This As An Index

  int             io_width;         // Defs the Bit-Width of the Input
  int             io_number;        // Defs Number of Entries to Transmit In/Out
  bool            is_pointer;       // Input is Defined as a Pointer
  bool            is_packed;        // Defs Whether or Not Packing Is Used [Per-Variable Packing]
  bool            is_dma;           // Defs Whether or Not DMA Access is To Be Used For This Param
} s_io_params;

// Structure That Describes Function Names and Parameters
typedef struct
{
  char            *func_name;       // Pointer to Name of User Function
  int             func_id;          // Numeric Function ID (Assigned by gThis)
  int             nmbr_instances;   // Used to Determine Number of Instances to Generate

  int             nmbr_inputs;      // Defines Total Number of Inputs
  s_io_params     *inputs;          // Link to Structure w/ Information About Inputs
  bool            has_output;       // Enable or Disable Value Returns
  s_io_params     *output;          // Link to Structure w/ Information About Output

  bool            splitting_f;      // Enable or Disable Splitting for a Function
  bool            indexing_f;       // Denotes Whether I/O Indexing is Used w/ a Function
} s_func_params;
```

Figure 7.3: Description of splice_params Data Structure

Chapter 8

Using Splice: A Real World Example

The previous chapters of this document have focused primarily upon the physical implementation Splice, with little emphasis on how the tool might be applied to the design of an actual hardware device. As a result, while the utility and structure of the application have been well-defined, the developers role in the process of hardware design has been largely ignored. In an effort to address this issue, this chapter provides a comprehensive walk through of the Splice workflow starting from a set of interface declarations and ending with a fully-functional collection of peripheral logic. Along the way, particular attention is placed on the various methods by which developers can leverage the capabilities of the tool in order to simplify the tasks commonly associated with hardware design.

8.1 Selecting a Device to Implement via Splice

Due to space constraints and other structural complications it would difficult to describe, in sufficient detail, the various steps required to create a sophisticated device via Splice within the confines of a single chapter of this document. Furthermore, if such a task were undertaken the result would likely be a collection of information that focused almost exclusively on the low-level details of the resulting hardware. In an effort to avoid such complications and better emphasize the functionality of the tool, we have instead decided to present the design of a conceptually simplistic device: the standard hardware timer.

In essence, a hardware timer does little more than count to a specified value (i.e. the "threshold") at a rate determined by an external clock, and then signals an event when the counting has completed. By intercepting these events, listening devices can activate themselves and execute tasks in a periodic manner. Depending on the implementation, the timer may reset itself automatically after firing and immediately begin counting again, or wait for outside intervention to occur before continuing operations. For the purposes of this example, the former method of operation will be considered.

When architecting a timer or other such generalized device, it is wise to plan the design in such a way that the resulting product can easily be integrated into a wide variety of hardware systems. For example, it would be somewhat wasteful to create a timer that was explicitly tied to a particular clock rate or that could only count to a particular threshold value. Furthermore, a timer that could not be enabled or disabled at will would be similarly limiting in terms of the situations under which it could be employed. In traditional interface designs the addition of such functionality would add a fair amount of complexity to the resulting device. By leveraging the flexibility of Splice, however, such features can be added to the timer without a great deal of effort.

8.2 Describing a Device in the Splice Syntax

Taking into account the design specification outlined above, one can imagine how such a timer might be implemented in the ANSI C language. Functions would be created to enable and disable the timer as well as the set and retrieve the threshold value assigned to the device. Beyond this, one might include the ability to retrieve the clock rate of the bus across which the timer was deployed. Doing so would enable the end-user of the device to accurately set the threshold of the device based upon the rate at which the counting would occur. Once all of the functionality was finally determined, a set of function prototypes similar to that seen in Figure 8.1 would likely emerge.

While the design specification presented above is fairly straightforward in nature, it does imply a few limitations about the structure of the timer device that bear mentioning. First and foremost, the use of the `llong` (`unsigned long long`) data type for the get and set threshold operations implies that the timer will operate off of a 64-bit counter. Such an implementation will take additional logic resources to

Prototype	Purpose
void disable()	Disables (or pauses) the timer and prevents further counting.
void enable()	Enables (or activates) the timer and starts (or continues) counting operations.
void set_threshold(llong thold)	Set the threshold value of the timer based on the input "thold" value.
llong get_threshold()	Retrieves the current threshold value of the timer from the hardware device.
llong get_snapshot()	Retrieves the current value of the timer's counter.
ulong get_clock()	Returns the clock rate of the system bus across which the timer is deployed.
ulong get_status()	Returns the current status of the timer (active, disabled, etc).

Figure 8.1: Function Prototypes for a C-Based Timer Implementation

implement compared to a 32-bit timer, but will also allow larger intervals to be used with the device. Considering that most modern embedded bus interfaces use a clock of around 100 MHz, and that the timer will increment once per clock cycle a 32-bit implementation would have an effective range of only 42.9 seconds. By extending this to 64-bits allows, however, a range of over 5849 *years* is achievable, which should be far more than is required for any type of common timing task.

One other design decision worth mentioning is that the `get_clock` function is configured to return a 32-bit `ulong` (`unsigned long`) value. Assuming a return value in Hz, this implies that the device will never operate across a system interface whose clock rate exceeds 4.29 GHz. Given that current embedded processors themselves struggle to reach even 1 GHz, however, and that interface speed tends to lag that of CPUs, such an assumption will likely be valid for at least the next 5 to 10 years. In other words, as specified, the design of the timer should be able to scale to meet the requirements of future interfaces and will be able to deployed across multiple generations of hardware without the need to rearchitect its internal operations.

In the realm of embedded design, it is common for the set of specifications for a device to be converted into a fully-functioning software implementation of the timer prior to starting work on a formalized hardware description. In doing so, any misconceptions about the design could be cleared up in software without having to resort to the often difficult task of identifying such issues through hardware-based

simulations. Furthermore, having a timer specification in software before implementation begins would allow developers to construct a test suite for the device before it was actually completed, resulting in an increase in the amount of design work that could be performed in parallel.

When developing hardware via Splice this method of design provides the added benefit of being able to take the resulting set of software prototypes and convert them directly into interface specifications from which a set of hardware descriptions that handle all I/O operations of the underlying device can be created. In doing so, developers can save a great deal of time and effort as compared to the use traditional HDLs directly, where syntactical limitations require that the hardware implementation of a given interface be constructed independently of the original software blueprint. In fact, the only additional input that Splice requires to begin the process of hardware generation, is a target specification that indicates the type of development platform upon which the resulting device will be deployed. Assuming the use of the PLB for the purposes of this example, would result in an interface specification similar to that seen in Figure 8.2.

```
// Target Specification
% name hw_timer
% hdl_type vhdl
% bus_type plb
% bus_width 32
% base_address 0x8000401C
% dma_support false
% user_type llong, unsigned long long, 64
% user_type ulong, unsigned long, 32

// Interface Directives
void disable{};
void enable{};
void set_threshold{llong thold};
llong get_threshold{};
llong get_snapshot{};
ulong get_clock{};
ulong get_status{};
```

Figure 8.2: Splice Specification for the Timer Device

8.3 Implementing a Splice-based Hardware Design

Passing the timer specification seen in Figure 8.2 into Splice, will cause the tool to generate the set of VHDL formatted hardware logic files listed in Figure 8.3. By combining these files together in a Xilinx **Integrated Synthesis Environment** (ISE) or Modelsim project, one can simulate the entire collection of I/O operations for the hardware timer. Since the generated files will contain no calculation logic at this point, however, the device will be largely useless and incapable of performing any timer-related tasks. The method in which these files can be filled in to implement the timer device is the primary topic of discussion in this section.

File Name	Purpose
plb_interface.vhd	Provides a PLB to SIS adapter with support for simple 32-bit transfers and a base memory address of 0x8000401C.
user_hw_timer.vhd	Bus arbiter for the `hw_timer` device that is used to pass information to and from each user function.
func_enable.vhd	Implements I/O Logic (a synchronous wait operation) for the `enable` function.
func_disable.vhd	Implements I/O Logic (a synchronous wait operation) for the `disable` function.
func_set_threshold.vhd	Implements I/O Logic (a 64-bit bus write operation) for the `set_threshold` function.
func_get_threshold.vhd	Implements I/O Logic (a 64-bit bus read operation) for the `get_threshold` function.
func_get_snapshot.vhd	Implements I/O Logic (a 64-bit read operation) for the `get_snapshot` timer function.
func_get_clock.vhd	Implements I/O Logic (a 32-bit bus read operation) for the `get_clock` timer function.
func_get_status.vhd	Implements I/O Logic (a 32-bit bus read operation) for the `get_status` timer function.

Figure 8.3: Listing of Hardware Files Generated By Splice for the Timer Device

In examining the file listing above, one will likely notice that each interface declaration that was passed into the tool resulted in the creation of an independent VHDL logic file. By default, each of these "stubs" contains only the I/O logic necessary to complete all bus transactions for their associated function: 32-bit reads for

get_clock and get_status, 64-bit reads for get_snapshot and get_threshold, a 64-bit write for set_threshold, and synchronous waits (no return value) for enable and disable. This, in turn, indicates that in the ideal case, no function should take more than two clock cycles (a 64-bit transaction split across a 32-bit PLB) to complete the set of bus operations with which they are associated.

8.3.1 Filling in User-Logic Stubs

Due to the isolated nature of each logic stub, it would be somewhat difficult to implement the entire timer device within the confines of any single function. Instead, it would be more efficient to instantiate an independent timer module in the user_hw_timer.vhd file, where it can reside alongside the component declarations for the various user logic functions supported by the device. In doing so, information can be transmitted between the timer and each logic stub via direct port mappings that require a minimum amount of effort to implement. The timer itself could then operate independently of other bus interaction and simply respond to requests from each user-defined function as they arrive.

With the above design considerations in mind, the control logic required to manipulate the timer device can be constructed in a fairly straightforward manner. Each logic stub can simply be configured to include a TIMER_ACTIVATE signal in their port listing that would be mapped directly to the timer and then triggered each time the function was activated. By coupling this functionality with a corresponding TIMER_CMD_DONE signal that is passed back from the timer and activated at the completion of each operation, a simplified handshaking mechanism can be formed within the confines of the peripheral. In cases where data needs to be passed into or out of the timer, additional wires can be instantiated to form the required data pathway.

An example of how such code might be injected into the set_threshold function can be found in Figure 8.4. In the context of this example, the handshaking code is offset from the statements generated by Splice via square brackets. By adding similar statements to the other logic stubs generated by the tool a common protocol for timer-to-function communication can be established within the device, thereby simplifying the task of routing data into and out of the peripheral.

```
-- Operate Based on the Current State
case (cur_state) is

  -- Handling 2 Write Operation(s) for time_interval
  when IN_time_interval =>

    -- Wait for Input Data Destined For This Function
    if (DATA_IN_VALID = '1' and FUNC_ID = MY_FUNC_ID) then

      -- Check to See if All Values for This Particular State Have Been Received
      if (time_interval_counter = time_interval_max_value) then
        [TIMER_INTERVAL_OUT(32 to 63) <= DATA_IN;]
        [TIMER_ACTIVATE               <= '1';]
        time_interval_counter         <= (others => '0');
        next_state                    <= WAIT_HANDSHAKE;
      else
        [TIMER_INTERVAL_OUT(0 to 31)  <= DATA_IN;]
        time_interval_counter         <= time_interval_counter + 1;
      end if;

      IO_DONE  <= '1';
    end if;

  -- Handshake Report Back From the Timer Module
  when WAIT_HANDSHAKE =>

    [if (TIMER_CMD_DONE = '1') then]
      next_state <= OUT_RESULT;
    end if;

  -- Handling 1 False Read Operation(s) for result
  when OUT_RESULT =>

    CALC_DONE <= '1';

    -- Wait for Timer to Complete and Then Pass An Output
    if (FUNC_ID = MY_FUNC_ID) then
      next_state     <= IN_time_interval;
      CALC_DONE      <= '0';
      IO_DONE      <= '1';
      DATA_OUT       <= (others => '0');
      DATA_OUT_VALID <= '1';
    end if;

end case;
```

Figure 8.4: Example Handshaking Code for the `set_threshold` Timer Function

8.3.2 Architecting the Timer Device

The shared-bus structure employed by Splice and implemented by `user_hw_timer.vhd` guarantees that no more than a single user-logic function will ever be active within a peripheral at any given point in time. Stated in terms of the handshaking protocol described above, this essentially means that under normal operation only one `TIMER_ACTIVATE` signal could ever be toggled high within the timer module. As such, there is no need for the timer to include the ability to arbitrate amongst concurrent requests or assign explicit priorities to any of the operations supported by the peripheral.

By concatenating the `TIMER_ACTIVATE` signals implemented by each user logic stub, a single `COMMAND` vector can be formed. This vector can then be passed into the timer module and analyzed as a one-hot encoded signal on the rising edge of each clock cycle to determine which (if any) function needs to be executed by the device. Once all communication with the activated logic stub has completed, the timer can raise the `TIMER_CMD_DONE` signal to update the state of the system. At this point, the user logic stub can handshake with the PLB to terminate the transaction and return control to the calling process. Figure 8.5 provides an implementation of this functionality in a somewhat abbreviated format.

Working in conjunction with this control logic, the counter process (as shown in Figure 8.6) can simply count from a value of zero up to the threshold value (`timer_threshold`) and then raise a flag (`timer_trigger`) to alert listening devices each time it fires. As commands come into the device, they will trigger events within the timer itself to modify the threshold, reset the counter, or accomplish other required tasks. By combining the control logic and counter process into a single VHDL file (i.e. "`timer.vhd`") and then integrating the design with the user logic stubs, a timer implementation that is both platform independent and highly flexible in nature can be synthesized with a minimum of coding effort on the part of the developer and without having to grapple with the esoteric protocols of any particular system interface.

```
case (COMMAND) is                              -- Operate Based on Command
  when OP_ENABLE =>                            -- Handle the Enable Operation
    timer_enabled   <= '1';

  when OP_DISABLE =>                           -- Handle the Disable Operation
    timer_enabled   <= '0';

  when OP_SET_INTERVAL =>                      -- Handle the Set Interval Operation
    timer_clear     <= '1';
    timer_threshold <= INTERVAL_IN;

  when OP_GET_INTERVAL =>                      -- Handle the Get Interval Operation
    INTERVAL_OUT    <= timer_threshold;

  when OP_GET_SNAPSHOT =>                      -- Handle the Get Snapshot Operation
    SNAPSHOT_OUT    <= timer_value;

  when OP_GET_CLOCK =>                         -- Handle the Get Clock Operation
    CLOCK_OUT       <= TIMER_CLOCK_RATE;

  when OP_GET_STATUS =>                        -- Handle the Get Status Operation
    STATUS_OUT      <= timer_status;

  when others =>                               -- Do Nothing For all Other Possibilities
    NULL;
end case;

if (COMMAND /= 0) then                         -- When a Command is Receieved; ACK It (Handshaking)
  TIMER_CMD_DONE <= '1';
else
  TIMER_CMD_DONE <= '0';
end if;
```

Figure 8.5: Function Handling Code for the Hardware Timer

```
counter_module: process (CLK)
begin
  if (CLK = '1' and CLK'EVENT) then
    if (RST = '1') then
      timer_trigger     <= '0';                    -- Clear Values on Reset
      timer_value <= (others => '0');
    else
      timer_trigger <= '0';                        -- Clear Trigger By Default

      if (timer_clear = '1') then                  -- Clear the Timer When Instructed to Do So
        timer_value     <= (others => '0');
      elsif(timer_enabled = '1') then              -- If Timer is Enabled; Then Do Some Counting

      if (timer_value = timer_threshold) then      -- If Threshold Reached Trigger; Else Continue
          timer_trigger   <= '1';
          timer_value     <= (others => '0');
        else
          timer_value     <= timer_value + 1;      -- Increment the Timer By Default
        end if;

      end if;
    end if;
  end if;
end process;
```

Figure 8.6: Counter Code for the Hardware Timer

8.4 Using Splice Generated Software Drivers

In addition to the hardware files generated by the tool for the timer hardware, Splice will also create a set of simplistic software drivers for the device that are intended to act as the starting point for a full systems integration. These drivers are pre-configured to act as "black boxes" that link into the peripheral logic across an abstract interface via a set of transaction macros to direct data to and from each user-logic stub associated with the design. By linking these macros into a library file provided by the tool (`splice_lib.h`) communications across the PLB (or any other interface) can be achieved, thereby providing a means by which the hardware functions can be triggered from any common C, C++, or Java application in a bus-independent manner. A listing of the software files that Splice would create for the original timer specification can be found in Figure 8.7.

File Name	Purpose
splice_lib.h	Implementation of software macros in in-line assembly that is used to transfer data to and from the hardware timer across the PLB.
hw_timer_driver.c	Contains software driver functions for each interface declaration associated with the hardware timer device.
hw_timer_driver.h	Listing of function prototypes for each driver defined in the hw_timer_driver.c file.

Figure 8.7: Listing of Software Files Generated By Splice for the Timer Device

By leveraging the drivers created by the tool, developers are able to avoid the tedious and error-prone task of writing hardware wrappers for their devices. The functions generated by Splice can simply be dropped into a standard C source file and combined with control flow statements to implement a simplistic test suite for the device. Through the use of debug statements, selected portions of the test suite can be executed to manipulate individual pieces (functions) of the device, allowing for incremental hardware changes to be tested in a modular fashion with little to no additional coding effort. Figure 8.8 provides an example test suite for the hardware timer device that makes extensive use of the drivers generated by the tool. Through the use of such a test suite, the hardware design presented in the previous section can be verified and then linked into a fully-featured application, thus completing the system implementation process.

```
#include <stdio.h>                      // Standard I/O Functions
#include <unistd.h>                     // Using "sleep()"
#include "splice_lib.h"                 // Splice Library Functions (PLB Interface)

int main()
{
  ulong clock_rate;                     // 32 Bit Value
  ulong status;                         //     "
  llong threshold;                      // 64 Bit Value
  llong get_threshold;                  //     "
  llong current_value;                  //     "

  disable();                            // Disable the Timer to Start

  clock_rate = get_clock();             // Retrieve Clock Speed of the Underlying Bus
  threshold  = clock_rate * 5;          // Setup a 5 Second Threshold (clock_rate = ticks/second)

  set_threshold(threshold);             // Setup the Timer to Operate Off a 5 Second Threshold
                                        // Also Resets the Timer

  enable();                             // Enable the Timer

  current_value = get_snapshot();       // Take A Snapshot of the Current Timer Value
  printf("Value: %ull\n",current_value);// Print the Value Retrieved (Should be close to 0)

  sleep(6);                             // Sleep 6 seconds; timer should fire.

  status = get_status();                // Grab the Status Value (Clears Internal Timer Fired Bit)
  printf("Status: %x\n", status);       // Print Status Value    (Bit 0 = Enabled, Bit 1 = "Fired")

  disable();                            // Disable the Timer

  get_threshold = get_thresold();       // Retrieve the Threshold Value (Should Be Same as Set Above)
  printf("Thold: %ull\n",get_threshold);// Print Threshold Value

  status = get_status();                // Grab the Status Value
  printf("Status: %x\n", status);       // Print Status Value    (Bit 0 = Enabled, Bit 1 = "Fired")

  return 0;                             // Terminate the Test
}
```

Figure 8.8: Example Software Test Suite For the Timer Device

Chapter 9

Evaluating Performance

While Splice is able to provide a wide-array of interface generation capabilities, the tool would be useless if the hardware it generated was substantially slower than that which could be implemented directly. Therefore, in an effort to quantify the performance impact that the bus interfaces and arbiters generated by Splice have upon overall hardware performance, a pre-existing hardware device was chosen and re-implemented to make use of a number of Splice-compliant interfaces. Specific details about the device that was used in this testing are presented below along with a selection of performance results.

9.1 The Scan Eagle UAV

The device selected for testing was a linear interpolator that is used within the Scan Eagle Unmanned Aerial Vehicle (UAV) to approximate continuous flight control data for the aircraft from a set of time-valued samples. The interpolated data is then used to steer the aircraft properly during the time periods in which sampled data is not being received. This device was chosen primarily for the following three reasons:

1. We have access to two pre-existing bus interconnects for the device that were coded by hand for use in previous research.

2. The calculation logic for the device runs in a predictable manner and requires the same numbers of clock cycles to produce results each time it is run, making it simpler to obtain reproducible performance results.

3. The interpolator can be run in four modes (scenarios), each of which require differing amounts of input to execute, thus providing the chance to test the performance of Splice-based interfaces under a wide range of usage patterns.

9.2 Experimental Configuration

In terms of bus traffic, the four usage scenarios of the linear interpolator each operate upon three sets of input values to generate a single output. The exact meanings of these values are not important for the purposes of this analysis since the amount of calculation done in each implementation is constant. As such, a full explanation of the inner workings of the interpolator is omitted for the sake of brevity. The precise number of inputs that each scenario requires is shown in Figure 9.1. It should be noted that since each set of values transferred to the hardware is contained in a separate data array, it would be impossible to transfer all items across the bus via a single burst or DMA transaction.

Scenario	Set 1	Set 2	Set 3	Total
1	2	1	2	5
2	4	2	4	10
3	8	3	6	16
4	16	4	8	28

Figure 9.1: Input Parameters Required for Each Scenario

9.2.1 Description of Interfaces used in Testing

For the purposes of testing, a total of five interface implementations for the linear interpolator were created and placed on the Xilinx ML-403 development board [22]. These interfaces, which consisted of two hand-coded variants and three Splice-generated implementations, were each attached to the on-board PowerPC 405 and manipulated through a standardized driver set. Of the hand-coded implementations, the PLB variant (referred to as "Simple PLB" in the figures provided below) was the product of the first attempt at generating an interface for the linear interpolation device. At the time the interface was coded, the designer was not aware of all of the intricacies of the PLB and thus the interface was not nearly as optimized as it could have been. Thus,

the performance results obtained from this interface should be considered representative of what an end-user who is unfamiliar with the protocols of a particular bus would likely create. Conversely, the hand-coded FCB interface ("Optimized FCB") is a highly optimized implementation that was created to replace the slower PLB interconnect.

All three Splice-generated interfaces are attached to an identical user-logic function that makes use of implicit pointer declarations to transfer the required number of values from each of the three datasets depending on the scenario that is run. One of the generated interfaces ("Splice PLB (Simple)"), is a minimally sized PLB interconnect that is capable of orchestrating single-word (32-bit) transmissions across the bus. The generated FCB interface ("Splice FCB"), on the other hand, is able to facilitate double and quad-word transfers for sets of data values that can benefit from such transmissions.

The third and final interface generated for testing via Splice for testing is an additional PLB interconnect ("Splice PLB (DMA)") that contains the supplementary control logic required to support DMA transactions. The use of DMA across the PLB bus is interesting in that it does not benefit transactions of four or fewer data values. This is due to the fact that the DMA circuitry requires a minimum of four bus transactions to setup and take down, thus negating any benefits for lesser transmissions [20].

9.3 Experimental Results

To perform the testing, a memory-accessible clock-cycle accurate timer was loaded onto the FPGA along with a small EDK project consisting of the bare minimum hardware to activate each peripheral interface and obtain results. The embedded PPC-405 was clocked at 300 MHz, while the on-chip PLB and FCB interconnects were clocked at 100 MHz. For the hand-coded PLB and FCB interfaces pre-existing drivers were used to transmit data to and from the interpolator, while Splice-created drivers were used for the three generated interfaces. The results of these tests for each interpolation scenario can be seen in Figure 9.2.

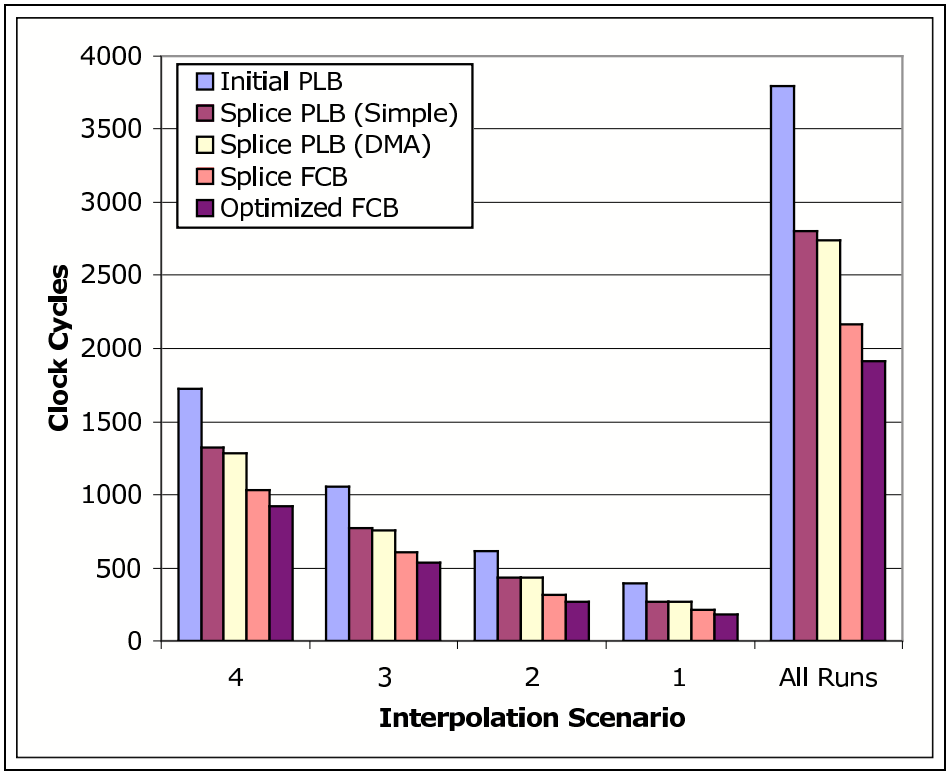

Figure 9.2: Clock Cycles Per Run By Each Implementation

9.3.1 Comparison of Transmission Time

The performance results show that Splice-generated interfaces compare favorably to those generated by hand. Overall, the Splice-generated simple PLB Interface is approximately 25% faster than the naïve hand-coded implementation. Furthermore, the Splice-generated FCB interface is approximately 43% faster than the naïve PLB implementation and only 13% slower than an optimized hand-coded FCB interconnect. For this particular hardware device, DMA transactions enacted via the PLB bus are not very beneficial, representing only a 1-4% performance increase versus a non-DMA implementation.

9.3.2 Comparison of Resource Usage

Besides pure performance results, there is also the issue of how many FPGA resources each implementation consumes. Although reconfigurable logic devices continue to grow in size at a rapid rate, there are still cases where it is difficult to fit all of the logic required for a particular design onto the chip. Resource usage statistics for each of the tested bus interconnects are provided in Figure 9.3.

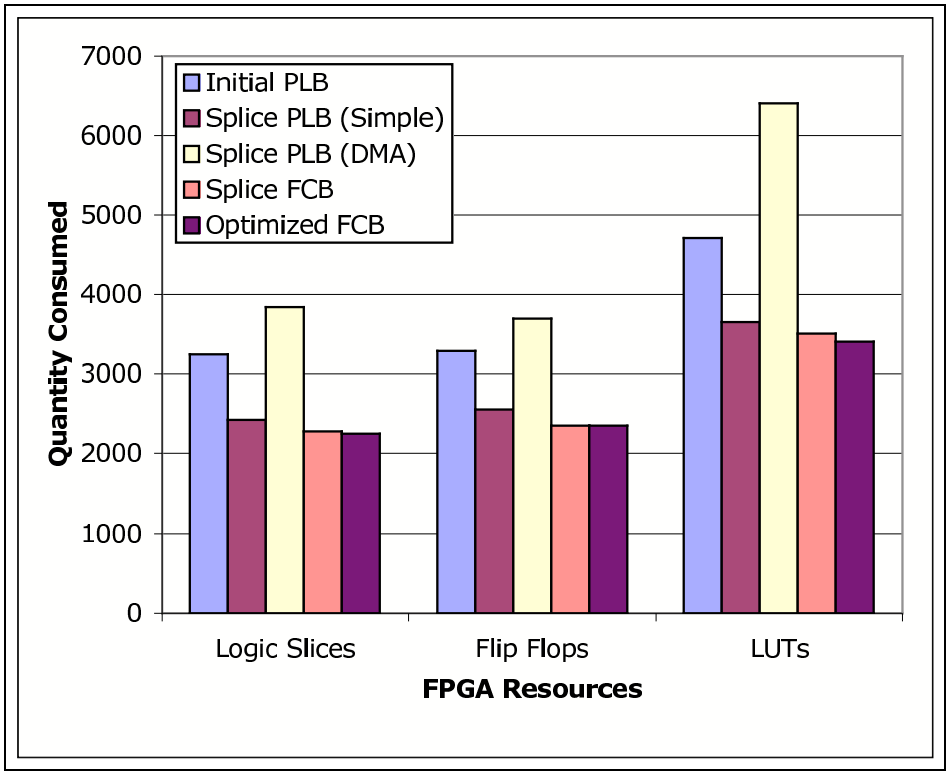

Figure 9.3: FPGA Resource Consumed By Each Implementation

The resource usage numbers obtained for each bus implementation are similar to that of the performance results. On average, the Splice-generated simple PLB interface consumes about 23% less FPGA resources than the naïve hand-coded implementation. Similarly, the Splice-generated FCB interface requires (on average) 28% less resources than the naïve PLB implementation, and only around 2% more resources than an optimized hand-coded FCB interconnect.

The most surprising usage statistic perhaps, is the astronomical resource usage that results from enabling DMA transactions for a PLB device. In this particular case, the DMA-supporting interface requires anywhere from 57-69% more FPGA resources to implement than the otherwise identical simple PLB interconnect. As such, when an end-user chooses to enable DMA support for a particular device they need to ensure that the possible performance benefit is worth the additional resource cost.

Chapter 10

Conclusion

10.1 Thesis in Review

As consumer demand for advanced technology continues to rise, the performance requirements imposed on embedded systems designers have become increasingly difficult to satisfy through the use of off-the-shelf components alone. To address this issue, developers have begun to design systems in which standard microprocessors are combined with custom-fabricated hardware capable of off-loading the specialized tasks of sophisticated software algorithms. This approach to system development is well-suited for use in high-volume markets, where the investment required to fabricate ASICs can be amortized over the lifespan of a product or lessened through licensing agreements with competitors. In emerging sectors, however, such up-front costs are often prohibitive in nature, leading developers to instead turn to the use of reprogrammable logic devices, which offer slightly less performance at a significantly lower price point. As a means of responding to this trend, manufacturers such as Xilinx and Altera have begun to introduce devices that combine an FPGA and general purpose CPU into a single SoC component. By leveraging these devices in their designs, budget-constrained developers can produce embedded systems that are capable of meeting the performance requirements of all but the most demanding workflows.

When developers first begin working with FPGA-based SoCs, the sheer number of choices that are available in the marketplace can be somewhat daunting. Each manufacturer typically provides their own proprietary system architecture along with a set of associated peripheral interfaces whose transmission protocols are wholly incompatible with one another. By coding their designs to match the specifications of a given interface, developers will typically be forced to work at the raw signal level,

handshaking directly with the bus to accomplish all transmissions. This, in turn, effectively binds the resulting logic to the hardware platform for which it was originally designed, making it difficult to create modular components and forcing a partial rewrite of the peripheral each time support for a new family of devices is added.

Splice attempts to address these issues by providing an abstraction layer on top of existing HDLs through which interface logic can be architected in an entirely functional manner using a syntax based on the ANSI C language. In doing so, the tool is able to isolate developers from the hardware platforms on which their designs are deployed, allowing them to instead code their logic with respect to a generalized system interconnect known as the SIS. By attaching logic coded in terms of the SIS to a compatible native interface adapter, a single implementation of a peripheral can be linked into a variety of hardware platforms by simply changing the set of parameters that are passed to Splice at runtime. To further extend the useful life span of designs, the tool also provides a development API through which adapters for additional interfaces can be constructed with minimal coding effort.

In terms of both performance and resource usage, Splice-generated interfaces compare favorably with manually created optimized interconnects and can often outperform "naïve" bus implementations by a significant margin. Furthermore, by virtue of the fact that the tool can generate interconnects almost instantly, Splice enables developers to experiment with the various bus mechanisms supported by their SoC without the need to manually code even a single interface. This, in turn, saves valuable development time and allows designers to focus on the task of creating optimized "business logic" for their peripherals without having to concern themselves with the complex communications protocols of the various platforms that they wish to target.

10.2 Future Work

Although Splice offers a great deal of functionality to embedded system developers in its present form, a number of architectural limitations prevent the tool from reaching its full potential. As work continues on this project, however, we hope to address these issues through a series of enhancements to the interface declaration constructs and code generation facilities provided by the tool. The modifications that are planned at this time are outlined in the remainder of this section. In making these changes, the aim is to bring new capabilities to the tool while still maintaining the ability to

produce easily adaptable interface logic and software drivers that can be deployed across multiple hardware platforms in a straightforward manner.

One issue that must be addressed above all others is the fact that Splice currently offers support for only a limited subset of the system interconnects that developers might encounter. Without such functionality it will be difficult to expand the potential audience of the tool and establish it as a viable solution in the area of cross-platform peripheral design. It is therefore our intent to create SIS adapters for common interfaces like the AHB as well as lesser-known alternatives such as Wishbone and Avalon [12, 1]. In a similar vein, the tool would also benefit by increasing the capabilities of existing interfaces adapters, as doing so would afford developers additional flexibility in expressing their designs. Since the level of support that is currently offered for the OPB is somewhat abysmal, any work in this regard would likely be focused on its associated interface adapter before changes to any others were considered.

Beyond the basic need for added platform support, there are also a number of deficiencies in the existing interface declaration syntax that inhibit the usability of the tool in its present form. This includes the inability to directly request interrupt lines as well as a lack of proper support for ANSI C `struct` declarations. Such issues can be handled through simple parser extensions and with minimal changes to the underlying structure of the tool. In fact, preliminary testing with the use of interrupts in conjunction with Splice-based PLB interfaces is currently under way, but the work has yet to be distilled into a form suitable for use by developers.

Finally, while Splice provides the ability to produce interface logic and software drivers for a variety of transactions, the code generator supplied by the tool lacks some functionality that would be useful to the general development community. On the hardware side, it would be helpful to provide support for packed data operations in a bus-agnostic manner as well as to allow for pointers to be passed into the tool "by reference". Such changes could likely be implemented with minimal changes to the overall structure of the tool and without adversely effecting compatibility with existing Splice-based designs. Beyond this, built-in support for generating interface logic in the Verilog syntax or producing driver code pre-targeted to the Linux operating system would also serve to expand the potential user base of the application. In the case of the later, such functionality could be added through simple physical-to-virtual memory mapping macros, while the former will likely require significant development effort in order to fully-integrate with the tool.

References

[1] Altera. *Avalon Memory-Mapped Interface Specification*, November 2006.

[2] ANSI. *ANSI C-99 Specification*, May 2005.

[3] ARM. *AMBA Specification*, May 1999.

[4] S. Chappell and C. Sullivan. Handel-C for co-processor and co-design of field programmable system on chip. In *Workshop on Reconfigurable Computing and Applications*, September 2002.

[5] J. Gaisler, S. Habnic, and E. Catovic. *GRLIB/LEON3 Users Manual*, August 2006.

[6] IBM. *IBM CoreConnect Specification Overview*, September 2006.

[7] IEEE. *IEEE Standard System C Language Reference Manual*, 2005.

[8] R.L. Lysecky, F. Vahid, and T.D. Givargis. Experiments with the PVCI. In *International Symposium on System Synthesis*, September 2000.

[9] H.H. Ng and L. Pillai. Accelerated system performance with the APU controller and extreme DSP slices. *Xilinx Application Notes*, September 2005.

[10] Nu Horizons. *Nu Horizons Spartan3 1500/200 Evaluation Platform Reference Manual*, January 2006.

[11] Object Management Group. *CORBA Specification*, January 2006.

[12] OpenCores Working Group. *Wishbone SoC Interconnection Architecture*, September 2002.

[13] Perldoc. *Perl Common Regular Expressions Reference Manual*, April 2006.

[14] SWIG Working Group. *SWIG Users Manual*, July 2006.

[15] Virtual Socket Interface Association. *On-Chip Bus Development Working Group Specification 1 Version 1.0 (OCB 1 1.0)*, 1998.

[16] Xilinx. *Processor Local Bus (PLB) Specification*, July 2003.

[17] Xilinx. *FCB Product Specification*, August 2005.

[18] Xilinx. *Microblaze Processor Reference Guide*, October 2005.

[19] Xilinx. *On-Chip Peripheral Bus (OPB) Specification*, December 2005.

[20] Xilinx. *PLB IP Interface Guide*, April 2005.

[21] Xilinx. *PowerPC 405 Processor Block Reference Guide*, 2.1 edition, July 2005.

[22] Xilinx. *ML401/ML402/ML403 Evaluation Platform User Guide*, May 2006.

[23] Xilinx. *Xilinx Spartan3 FPGA Family Overview*, April 2006.

[24] Xilinx. *Xilinx Virtex4 FPGA Family Overview*, January 2007.

Vita

Justin Thiel

Date of Birth October 13, 1983

Place of Birth Collinsville, IL

Degrees MS, Computer Engineering
Washington University in Saint Louis, May 2007

BS Magna Cum Laude, Computer Engineering
Southern Illinois University Edwardsville, May 2005

Publications Justin R. Thiel and Ron K. Cytron **Splice: A Standardized Peripheral Logic and Interface Creation Engine**. In *IEEE Computer Socieity Reconfigureable Architecture 2007 Workshop*, March 2007

May 2007